Python Documentation contents

- What’s New in Python
    - What’s new in Python 3.14
        - Summary – Release highlights
        - New features
            - **PEP 649** & **PEP 749**: Deferred evaluation of annotations
            - **PEP 734**: Multiple interpreters in the standard library
            - **PEP 750**: Template string literals
            - **PEP 768**: Safe external debugger interface
            - A new type of interpreter
            - Free-threaded mode improvements
            - Improved error messages
            - **PEP 784**: Zstandard support in the standard library
            - Asyncio introspection capabilities
            - Concurrent safe warnings control
        - Other language changes
            - Built-ins
            - Command line and environment
            - PEP 758: Allow `except` and `except*` expressions without brackets
            - PEP 765: Control flow in `finally` blocks
            - Incremental garbage collection
            - Default interactive shell
        - New modules
        - Improved modules
            - argparse
            - ast

- asyncio
- calendar
- concurrent.futures
- configparser
- contextvars
- ctypes
- curses
- datetime
- decimal
- difflib
- dis
- errno
- faulthandler
- fnmatch
- fractions
- functools
- getopt
- getpass
- graphlib
- heapq
- hmac
- http
- imaplib
- inspect
- io
- json
- linecache
- logging.handlers
- math
- mimetypes
- multiprocessing
- operator
- os
- os.path
- pathlib
- pdb

  - pickle
  - platform
  - pydoc
  - re
  - socket
  - ssl
  - struct
  - symtable
  - sys
  - sys.monitoring
  - sysconfig
  - tarfile
  - threading
  - tkinter
  - turtle
  - types
  - typing
  - unicodedata
  - unittest
  - urllib
  - uuid
  - webbrowser
  - zipfile
- Optimizations
  - asyncio
  - base64
  - bdb
  - difflib
  - gc
  - io
  - pathlib
  - pdb
  - textwrap
  - uuid
  - zlib
- Removed

    - argparse
    - ast
    - asyncio
    - email
    - importlib.abc
    - itertools
    - pathlib
    - pkgutil
    - pty
    - sqlite3
    - urllib
- Deprecated
    - New deprecations
    - Pending removal in Python 3.15
    - Pending removal in Python 3.16
    - Pending removal in Python 3.17
    - Pending removal in Python 3.18
    - Pending removal in Python 3.19
    - Pending removal in future versions
- CPython bytecode changes
    - Pseudo-instructions
- C API changes
    - Python configuration C API
    - New features in the C API
    - Limited C API changes
    - Removed C APIs
    - Deprecated C APIs
        - Pending removal in Python 3.15
        - Pending removal in Python 3.16
        - Pending removal in Python 3.18
        - Pending removal in future versions
- Build changes
    - `build-details.json`
    - Discontinuation of PGP signatures
    - Free-threaded Python is officially supported

        - Binary releases for the experimental just-in-time compiler
    - Porting to Python 3.14
        - Changes in the Python API
        - Changes in annotations (**PEP 649** and **PEP 749**)
            - Implications for annotated code
            - Implications for readers of `__annotations__`
            - Related changes
            - `from __future__ import annotations`
        - Changes in the C API
    - Notable changes in 3.14.1
- What's New In Python 3.13
    - Summary – Release Highlights
    - New Features
        - A better interactive interpreter
        - Improved error messages
        - Free-threaded CPython
        - An experimental just-in-time (JIT) compiler
        - Defined mutation semantics for `locals()`
        - Support for mobile platforms
    - Other Language Changes
    - New Modules
    - Improved Modules
        - argparse
        - array
        - ast
        - asyncio
        - base64
        - compileall
        - concurrent.futures
        - configparser
        - copy
        - ctypes
        - dbm

- dis
- doctest
- email
- enum
- fractions
- glob
- importlib
- io
- ipaddress
- itertools
- marshal
- math
- mimetypes
- mmap
- multiprocessing
- os
- os.path
- pathlib
- pdb
- queue
- random
- re
- shutil
- site
- sqlite3
- ssl
- statistics
- subprocess
- sys
- tempfile
- time
- tkinter
- traceback
- types
- typing
- unicodedata

    - venv
    - warnings
    - xml
    - zipimport
- Optimizations
- Removed Modules And APIs
    - PEP 594: Remove “dead batteries” from the standard library
    - 2to3
    - builtins
    - configparser
    - importlib.metadata
    - locale
    - opcode
    - optparse
    - pathlib
    - re
    - tkinter.tix
    - turtle
    - typing
    - unittest
    - urllib
    - webbrowser
- New Deprecations
    - Pending removal in Python 3.14
    - Pending removal in Python 3.15
    - Pending removal in Python 3.16
    - Pending removal in Python 3.17
    - Pending removal in Python 3.18
    - Pending removal in Python 3.19
    - Pending removal in future versions
- CPython Bytecode Changes
- C API Changes
    - New Features
    - Changed C APIs
    - Limited C API Changes

        - Removed C APIs
        - Deprecated C APIs
            - Pending removal in Python 3.14
            - Pending removal in Python 3.15
            - Pending removal in Python 3.16
            - Pending removal in Python 3.18
            - Pending removal in future versions
    - Build Changes
    - Porting to Python 3.13
        - Changes in the Python API
        - Changes in the C API
    - Regression Test Changes
- What’s New In Python 3.12
    - Summary – Release highlights
    - New Features
        - PEP 695: Type Parameter Syntax
        - PEP 701: Syntactic formalization of f-strings
        - PEP 684: A Per-Interpreter GIL
        - PEP 669: Low impact monitoring for CPython
        - PEP 688: Making the buffer protocol accessible in Python
        - PEP 709: Comprehension inlining
        - Improved Error Messages
    - New Features Related to Type Hints
        - PEP 692: Using `TypedDict` for more precise `**kwargs` typing
        - PEP 698: Override Decorator for Static Typing
    - Other Language Changes
    - New Modules
    - Improved Modules
        - array
        - asyncio
        - calendar
        - csv
        - dis

  - fractions
  - importlib.resources
  - inspect
  - itertools
  - math
  - os
  - os.path
  - pathlib
  - platform
  - pdb
  - random
  - shutil
  - sqlite3
  - statistics
  - sys
  - tempfile
  - threading
  - tkinter
  - tokenize
  - types
  - typing
  - unicodedata
  - unittest
  - uuid
- Optimizations
- CPython bytecode changes
- Demos and Tools
- Deprecated
  - Pending removal in Python 3.13
  - Pending removal in Python 3.14
  - Pending removal in Python 3.15
  - Pending removal in Python 3.16
  - Pending removal in Python 3.17
  - Pending removal in future versions
- Removed
  - asynchat and asyncore

        - configparser
        - distutils
        - ensurepip
        - enum
        - ftplib
        - gzip
        - hashlib
        - importlib
        - imp
        - io
        - locale
        - smtpd
        - sqlite3
        - ssl
        - unittest
        - webbrowser
        - xml.etree.ElementTree
        - zipimport
        - Others
    - Porting to Python 3.12
        - Changes in the Python API
    - Build Changes
    - C API Changes
        - New Features
        - Porting to Python 3.12
        - Deprecated
            - Pending removal in Python 3.14
            - Pending removal in Python 3.15
            - Pending removal in Python 3.16
            - Pending removal in future versions
        - Removed
- What’s New In Python 3.11
    - Summary – Release highlights
    - New Features
        - PEP 657: Fine-grained error locations in tracebacks

    - PEP 654: Exception Groups and **`except*`**
    - PEP 678: Exceptions can be enriched with notes
    - Windows **`py.exe`** launcher improvements
- New Features Related to Type Hints
    - PEP 646: Variadic generics
    - PEP 655: Marking individual **`TypedDict`** items as required or not-required
    - PEP 673: **`Self`** type
    - PEP 675: Arbitrary literal string type
    - PEP 681: Data class transforms
    - PEP 563 may not be the future
- Other Language Changes
- Other CPython Implementation Changes
- New Modules
- Improved Modules
    - asyncio
    - contextlib
    - dataclasses
    - datetime
    - enum
    - fcntl
    - fractions
    - functools
    - gzip
    - hashlib
    - IDLE and idlelib
    - inspect
    - locale
    - logging
    - math
    - operator
    - os
    - pathlib
    - re
    - shutil

        - socket
        - sqlite3
        - string
        - sys
        - sysconfig
        - tempfile
        - threading
        - time
        - tkinter
        - traceback
        - typing
        - unicodedata
        - unittest
        - venv
        - warnings
        - zipfile
    - Optimizations
    - Faster CPython
        - Faster Startup
            - Frozen imports / Static code objects
        - Faster Runtime
            - Cheaper, lazy Python frames
            - Inlined Python function calls
            - PEP 659: Specializing Adaptive Interpreter
        - Misc
        - FAQ
            - How should I write my code to utilize these speedups?
            - Will CPython 3.11 use more memory?
            - I don’t see any speedups in my workload. Why?
            - Is there a JIT compiler?
        - About
    - CPython bytecode changes
        - New opcodes

        - Replaced opcodes
        - Changed/removed opcodes
      - Deprecated
        - Language/Builtins
        - Modules
        - Standard Library
      - Pending Removal in Python 3.12
      - Removed
      - Porting to Python 3.11
      - Build Changes
      - C API Changes
        - New Features
        - Porting to Python 3.11
        - Deprecated
        - Pending Removal in Python 3.12
        - Removed
      - Notable changes in 3.11.4
        - tarfile
      - Notable changes in 3.11.5
        - OpenSSL
    - What’s New In Python 3.10
      - Summary – Release highlights
      - New Features
        - Parenthesized context managers
        - Better error messages
          - SyntaxErrors
          - IndentationErrors
          - AttributeErrors
          - NameErrors
        - PEP 626: Precise line numbers for debugging and other tools
        - PEP 634: Structural Pattern Matching
          - Syntax and operations
          - Declarative approach
          - Simple pattern: match to a literal
            - Behavior without the wildcard

        - Patterns with a literal and variable
        - Patterns and classes
            - Patterns with positional parameters
        - Nested patterns
        - Complex patterns and the wildcard
        - Guard
        - Other Key Features
    - Optional `EncodingWarning` and `encoding="locale"` option
- New Features Related to Type Hints
    - PEP 604: New Type Union Operator
    - PEP 612: Parameter Specification Variables
    - PEP 613: TypeAlias
    - PEP 647: User-Defined Type Guards
- Other Language Changes
- New Modules
- Improved Modules
    - asyncio
    - argparse
    - array
    - asynchat, asyncore, smtpd
    - base64
    - bdb
    - bisect
    - codecs
    - collections.abc
    - contextlib
    - curses
    - dataclasses
        - __slots__
        - Keyword-only fields
    - distutils
    - doctest
    - encodings
    - enum
    - fileinput

    - faulthandler
    - gc
    - glob
    - hashlib
    - hmac
    - IDLE and idlelib
    - importlib.metadata
    - inspect
    - itertools
    - linecache
    - os
    - os.path
    - pathlib
    - platform
    - pprint
    - py_compile
    - pyclbr
    - shelve
    - statistics
    - site
    - socket
    - ssl
    - sqlite3
    - sys
    - _thread
    - threading
    - traceback
    - types
    - typing
    - unittest
    - urllib.parse
    - xml
    - zipimport
- Optimizations
- Deprecated
- Removed

        - Porting to Python 3.10
            - Changes in the Python syntax
            - Changes in the Python API
            - Changes in the C API
        - CPython bytecode changes
        - Build Changes
        - C API Changes
            - PEP 652: Maintaining the Stable ABI
            - New Features
            - Porting to Python 3.10
            - Deprecated
            - Removed
        - Notable security feature in 3.10.7
        - Notable security feature in 3.10.8
        - Notable changes in 3.10.12
            - tarfile
    - What’s New In Python 3.9
        - Summary – Release highlights
        - You should check for DeprecationWarning in your code
        - New Features
            - Dictionary Merge & Update Operators
            - New String Methods to Remove Prefixes and Suffixes
            - Type Hinting Generics in Standard Collections
            - New Parser
        - Other Language Changes
        - New Modules
            - zoneinfo
            - graphlib
        - Improved Modules
            - ast
            - asyncio
            - compileall
            - concurrent.futures

  - curses
  - datetime
  - distutils
  - fcntl
  - ftplib
  - gc
  - hashlib
  - http
  - IDLE and idlelib
  - imaplib
  - importlib
  - inspect
  - ipaddress
  - math
  - multiprocessing
  - nntplib
  - os
  - pathlib
  - pdb
  - poplib
  - pprint
  - pydoc
  - random
  - signal
  - smtplib
  - socket
  - time
  - sys
  - tracemalloc
  - typing
  - unicodedata
  - venv
  - xml
- Optimizations
- Deprecated
- Removed

        - Porting to Python 3.9
            - Changes in the Python API
            - Changes in the C API
            - CPython bytecode changes
        - Build Changes
        - C API Changes
            - New Features
            - Porting to Python 3.9
            - Removed
        - Notable changes in Python 3.9.1
            - typing
            - macOS 11.0 (Big Sur) and Apple Silicon Mac support
        - Notable changes in Python 3.9.2
            - collections.abc
            - urllib.parse
        - Notable changes in Python 3.9.3
        - Notable changes in Python 3.9.5
            - urllib.parse
        - Notable security feature in 3.9.14
        - Notable changes in 3.9.17
            - tarfile
    - What’s New In Python 3.8
        - Summary – Release highlights
        - New Features
            - Assignment expressions
            - Positional-only parameters
            - Parallel filesystem cache for compiled bytecode files
            - Debug build uses the same ABI as release build
            - f-strings support = for self-documenting expressions and debugging
            - PEP 578: Python Runtime Audit Hooks
            - PEP 587: Python Initialization Configuration

    - PEP 590: Vectorcall: a fast calling protocol for CPython
    - Pickle protocol 5 with out-of-band data buffers
- Other Language Changes
- New Modules
- Improved Modules
    - ast
    - asyncio
    - builtins
    - collections
    - cProfile
    - csv
    - curses
    - ctypes
    - datetime
    - functools
    - gc
    - gettext
    - gzip
    - IDLE and idlelib
    - inspect
    - io
    - itertools
    - json.tool
    - logging
    - math
    - mmap
    - multiprocessing
    - os
    - os.path
    - pathlib
    - pickle
    - plistlib
    - pprint
    - py_compile

    - shlex
    - shutil
    - socket
    - ssl
    - statistics
    - sys
    - tarfile
    - threading
    - tokenize
    - tkinter
    - time
    - typing
    - unicodedata
    - unittest
    - venv
    - weakref
    - xml
    - xmlrpc
- Optimizations
- Build and C API Changes
- Deprecated
- API and Feature Removals
- Porting to Python 3.8
    - Changes in Python behavior
    - Changes in the Python API
    - Changes in the C API
    - CPython bytecode changes
    - Demos and Tools
- Notable changes in Python 3.8.1
- Notable changes in Python 3.8.2
- Notable changes in Python 3.8.3
- Notable changes in Python 3.8.8
- Notable changes in Python 3.8.9
- Notable changes in Python 3.8.10
    - macOS 11.0 (Big Sur) and Apple Silicon Mac support

        - Notable changes in Python 3.8.10
            - urllib.parse
        - Notable changes in Python 3.8.12
            - Changes in the Python API
        - Notable security feature in 3.8.14
        - Notable changes in 3.8.17
            - tarfile
    - What’s New In Python 3.7
        - Summary – Release Highlights
        - New Features
            - PEP 563: Postponed Evaluation of Annotations
            - PEP 538: Legacy C Locale Coercion
            - PEP 540: Forced UTF-8 Runtime Mode
            - PEP 553: Built-in `breakpoint()`
            - PEP 539: New C API for Thread-Local Storage
            - PEP 562: Customization of Access to Module Attributes
            - PEP 564: New Time Functions With Nanosecond Resolution
            - PEP 565: Show DeprecationWarning in `__main__`
            - PEP 560: Core Support for `typing` module and Generic Types
            - PEP 552: Hash-based .pyc Files
            - PEP 545: Python Documentation Translations
            - Python Development Mode (-X dev)
        - Other Language Changes
        - New Modules
            - contextvars
            - dataclasses
            - importlib.resources
        - Improved Modules
            - argparse
            - asyncio
            - binascii

- calendar
- collections
- compileall
- concurrent.futures
- contextlib
- cProfile
- crypt
- datetime
- dbm
- decimal
- dis
- distutils
- enum
- functools
- gc
- hmac
- http.client
- http.server
- idlelib and IDLE
- importlib
- io
- ipaddress
- itertools
- locale
- logging
- math
- mimetypes
- msilib
- multiprocessing
- os
- pathlib
- pdb
- py_compile
- pydoc
- queue
- re

    - signal
    - socket
    - socketserver
    - sqlite3
    - ssl
    - string
    - subprocess
    - sys
    - time
    - tkinter
    - tracemalloc
    - types
    - unicodedata
    - unittest
    - unittest.mock
    - urllib.parse
    - uu
    - uuid
    - warnings
    - xml
    - xml.etree
    - xmlrpc.server
    - zipapp
    - zipfile
- C API Changes
- Build Changes
- Optimizations
- Other CPython Implementation Changes
- Deprecated Python Behavior
- Deprecated Python modules, functions and methods
    - aifc
    - asyncio
    - collections
    - dbm
    - enum

        - gettext
        - importlib
        - locale
        - macpath
        - threading
        - socket
        - ssl
        - sunau
        - sys
        - wave
    - Deprecated functions and types of the C API
    - Platform Support Removals
    - API and Feature Removals
    - Module Removals
    - Windows-only Changes
    - Porting to Python 3.7
        - Changes in Python Behavior
        - Changes in the Python API
        - Changes in the C API
        - CPython bytecode changes
        - Windows-only Changes
        - Other CPython implementation changes
    - Notable changes in Python 3.7.1
    - Notable changes in Python 3.7.2
    - Notable changes in Python 3.7.6
    - Notable changes in Python 3.7.10
    - Notable changes in Python 3.7.11
    - Notable security feature in 3.7.14
- What’s New In Python 3.6
    - Summary – Release highlights
    - New Features
        - PEP 498: Formatted string literals
        - PEP 526: Syntax for variable annotations
        - PEP 515: Underscores in Numeric Literals
        - PEP 525: Asynchronous Generators
        - PEP 530: Asynchronous Comprehensions

    - PEP 487: Simpler customization of class creation
    - PEP 487: Descriptor Protocol Enhancements
    - PEP 519: Adding a file system path protocol
    - PEP 495: Local Time Disambiguation
    - PEP 529: Change Windows filesystem encoding to UTF-8
    - PEP 528: Change Windows console encoding to UTF-8
    - PEP 520: Preserving Class Attribute Definition Order
    - PEP 468: Preserving Keyword Argument Order
    - New dict implementation
    - PEP 523: Adding a frame evaluation API to CPython
    - PYTHONMALLOC environment variable
    - DTrace and SystemTap probing support
- Other Language Changes
- New Modules
    - secrets
- Improved Modules
    - array
    - ast
    - asyncio
    - binascii
    - cmath
    - collections
    - concurrent.futures
    - contextlib
    - datetime
    - decimal
    - distutils
    - email
    - encodings
    - enum

- faulthandler
- fileinput
- hashlib
- http.client
- idlelib and IDLE
- importlib
- inspect
- json
- logging
- math
- multiprocessing
- os
- pathlib
- pdb
- pickle
- pickletools
- pydoc
- random
- re
- readline
- rlcompleter
- shlex
- site
- sqlite3
- socket
- socketserver
- ssl
- statistics
- struct
- subprocess
- sys
- telnetlib
- time
- timeit
- tkinter
- traceback

    - tracemalloc
    - typing
    - unicodedata
    - unittest.mock
    - urllib.request
    - urllib.robotparser
    - venv
    - warnings
    - winreg
    - winsound
    - xmlrpc.client
    - zipfile
    - zlib
- Optimizations
- Build and C API Changes
- Other Improvements
- Deprecated
    - New Keywords
    - Deprecated Python behavior
    - Deprecated Python modules, functions and methods
        - asynchat
        - asyncore
        - dbm
        - distutils
        - grp
        - importlib
        - os
        - re
        - ssl
        - tkinter
        - venv
    - xml
    - Deprecated functions and types of the C API
    - Deprecated Build Options
- Removed

        - API and Feature Removals
    - Porting to Python 3.6
        - Changes in ‘python’ Command Behavior
        - Changes in the Python API
        - Changes in the C API
        - CPython bytecode changes
    - Notable changes in Python 3.6.2
        - New `make regen-all` build target
        - Removal of `make touch` build target
    - Notable changes in Python 3.6.4
    - Notable changes in Python 3.6.5
    - Notable changes in Python 3.6.7
    - Notable changes in Python 3.6.10
    - Notable changes in Python 3.6.13
    - Notable changes in Python 3.6.14
- What’s New In Python 3.5
    - Summary – Release highlights
    - New Features
        - PEP 492 - Coroutines with async and await syntax
        - PEP 465 - A dedicated infix operator for matrix multiplication
        - PEP 448 - Additional Unpacking Generalizations
        - PEP 461 - percent formatting support for bytes and bytearray
        - PEP 484 - Type Hints
        - PEP 471 - os.scandir() function – a better and faster directory iterator
        - PEP 475: Retry system calls failing with EINTR
        - PEP 479: Change StopIteration handling inside generators
        - PEP 485: A function for testing approximate equality

    - PEP 486: Make the Python Launcher aware of virtual environments
    - PEP 488: Elimination of PYO files
    - PEP 489: Multi-phase extension module initialization
- Other Language Changes
- New Modules
    - typing
    - zipapp
- Improved Modules
    - argparse
    - asyncio
    - bz2
    - cgi
    - cmath
    - code
    - collections
    - collections.abc
    - compileall
    - concurrent.futures
    - configparser
    - contextlib
    - csv
    - curses
    - dbm
    - difflib
    - distutils
    - doctest
    - email
    - enum
    - faulthandler
    - functools
    - glob
    - gzip
    - heapq
    - http

- http.client
- idlelib and IDLE
- imaplib
- imghdr
- importlib
- inspect
- io
- ipaddress
- json
- linecache
- locale
- logging
- lzma
- math
- multiprocessing
- operator
- os
- pathlib
- pickle
- poplib
- re
- readline
- selectors
- shutil
- signal
- smtpd
- smtplib
- sndhdr
- socket
- ssl
    - Memory BIO Support
    - Application-Layer Protocol Negotiation Support
    - Other Changes
- sqlite3
- subprocess

        - sys
        - sysconfig
        - tarfile
        - threading
        - time
        - timeit
        - tkinter
        - traceback
        - types
        - unicodedata
        - unittest
        - unittest.mock
        - urllib
        - wsgiref
        - xmlrpc
        - xml.sax
        - zipfile
    - Other module-level changes
    - Optimizations
    - Build and C API Changes
    - Deprecated
        - New Keywords
        - Deprecated Python Behavior
        - Unsupported Operating Systems
        - Deprecated Python modules, functions and methods
    - Removed
        - API and Feature Removals
    - Porting to Python 3.5
        - Changes in Python behavior
        - Changes in the Python API
        - Changes in the C API
    - Notable changes in Python 3.5.4
        - New `make regen-all` build target
        - Removal of `make touch` build target
- What’s New In Python 3.4

- Summary – Release Highlights
- New Features
    - PEP 453: Explicit Bootstrapping of PIP in Python Installations
        - Bootstrapping pip By Default
        - Documentation Changes
    - PEP 446: Newly Created File Descriptors Are Non-Inheritable
    - Improvements to Codec Handling
    - PEP 451: A ModuleSpec Type for the Import System
    - Other Language Changes
- New Modules
    - asyncio
    - ensurepip
    - enum
    - pathlib
    - selectors
    - statistics
    - tracemalloc
- Improved Modules
    - abc
    - aifc
    - argparse
    - audioop
    - base64
    - collections
    - colorsys
    - contextlib
    - dbm
    - dis
    - doctest
    - email
    - filecmp
    - functools
    - gc

- glob
- hashlib
- hmac
- html
- http
- idlelib and IDLE
- importlib
- inspect
- ipaddress
- logging
- marshal
- mmap
- multiprocessing
- operator
- os
- pdb
- pickle
- plistlib
- poplib
- pprint
- pty
- pydoc
- re
- resource
- select
- shelve
- shutil
- smtpd
- smtplib
- socket
- sqlite3
- ssl
- stat
- struct
- subprocess
- sunau

    - sys
    - tarfile
    - textwrap
    - threading
    - traceback
    - types
    - urllib
    - unittest
    - venv
    - wave
    - weakref
    - xml.etree
    - zipfile
- CPython Implementation Changes
    - PEP 445: Customization of CPython Memory Allocators
    - PEP 442: Safe Object Finalization
    - PEP 456: Secure and Interchangeable Hash Algorithm
    - PEP 436: Argument Clinic
    - Other Build and C API Changes
    - Other Improvements
    - Significant Optimizations
- Deprecated
    - Deprecations in the Python API
    - Deprecated Features
- Removed
    - Operating Systems No Longer Supported
    - API and Feature Removals
    - Code Cleanups
- Porting to Python 3.4
    - Changes in ‘python’ Command Behavior
    - Changes in the Python API
    - Changes in the C API
- Changed in 3.4.3

        - PEP 476: Enabling certificate verification by default for stdlib http clients
- What’s New In Python 3.3
    - Summary – Release highlights
    - PEP 405: Virtual Environments
    - PEP 420: Implicit Namespace Packages
    - PEP 3118: New memoryview implementation and buffer protocol documentation
        - Features
        - API changes
    - PEP 393: Flexible String Representation
        - Functionality
        - Performance and resource usage
    - PEP 397: Python Launcher for Windows
    - PEP 3151: Reworking the OS and IO exception hierarchy
    - PEP 380: Syntax for Delegating to a Subgenerator
    - PEP 409: Suppressing exception context
    - PEP 414: Explicit Unicode literals
    - PEP 3155: Qualified name for classes and functions
    - PEP 412: Key-Sharing Dictionary
    - PEP 362: Function Signature Object
    - PEP 421: Adding sys.implementation
        - SimpleNamespace
    - Using importlib as the Implementation of Import
        - New APIs
        - Visible Changes
    - Other Language Changes
    - A Finer-Grained Import Lock
    - Builtin functions and types
    - New Modules
        - faulthandler
        - ipaddress
        - lzma

- Improved Modules
    - abc
    - array
    - base64
    - binascii
    - bz2
    - codecs
    - collections
    - contextlib
    - crypt
    - curses
    - datetime
    - decimal
        - Features
        - API changes
    - email
        - Policy Framework
        - Provisional Policy with New Header API
        - Other API Changes
    - ftplib
    - functools
    - gc
    - hmac
    - http
    - html
    - imaplib
    - inspect
    - io
    - itertools
    - logging
    - math
    - mmap
    - multiprocessing
    - nntplib
    - os
    - pdb

    - pickle
    - pydoc
    - re
    - sched
    - select
    - shlex
    - shutil
    - signal
    - smtpd
    - smtplib
    - socket
    - socketserver
    - sqlite3
    - ssl
    - stat
    - struct
    - subprocess
    - sys
    - tarfile
    - tempfile
    - textwrap
    - threading
    - time
    - types
    - unittest
    - urllib
    - webbrowser
    - xml.etree.ElementTree
    - zlib
- Optimizations
- Build and C API Changes
- Deprecated
    - Unsupported Operating Systems
    - Deprecated Python modules, functions and methods
    - Deprecated functions and types of the C API

        - Deprecated features
    - Porting to Python 3.3
        - Porting Python code
        - Porting C code
        - Building C extensions
        - Command Line Switch Changes
- What’s New In Python 3.2
    - PEP 384: Defining a Stable ABI
    - PEP 389: Argparse Command Line Parsing Module
    - PEP 391: Dictionary Based Configuration for Logging
    - PEP 3148: The **`concurrent.futures`** module
    - PEP 3147: PYC Repository Directories
    - PEP 3149: ABI Version Tagged .so Files
    - PEP 3333: Python Web Server Gateway Interface v1.0.1
    - Other Language Changes
    - New, Improved, and Deprecated Modules
        - email
        - elementtree
        - functools
        - itertools
        - collections
        - threading
        - datetime and time
        - math
        - abc
        - io
        - reprlib
        - logging
        - csv
        - contextlib
        - decimal and fractions
        - ftp
        - popen

  - select
  - gzip and zipfile
  - tarfile
  - hashlib
  - ast
  - os
  - shutil
  - sqlite3
  - html
  - socket
  - ssl
  - nntp
  - certificates
  - imaplib
  - http.client
  - unittest
  - random
  - poplib
  - asyncore
  - tempfile
  - inspect
  - pydoc
  - dis
  - dbm
  - ctypes
  - site
  - sysconfig
  - pdb
  - configparser
  - urllib.parse
  - mailbox
  - turtledemo
- Multi-threading
- Optimizations
- Unicode
- Codecs

    - Documentation
    - IDLE
    - Code Repository
    - Build and C API Changes
    - Porting to Python 3.2
  - What’s New In Python 3.1
    - PEP 372: Ordered Dictionaries
    - PEP 378: Format Specifier for Thousands Separator
    - Other Language Changes
    - New, Improved, and Deprecated Modules
    - Optimizations
    - IDLE
    - Build and C API Changes
    - Porting to Python 3.1
  - What’s New In Python 3.0
    - Common Stumbling Blocks
      - Print Is A Function
      - Views And Iterators Instead Of Lists
      - Ordering Comparisons
      - Integers
      - Text Vs. Data Instead Of Unicode Vs. 8-bit
    - Overview Of Syntax Changes
      - New Syntax
      - Changed Syntax
      - Removed Syntax
    - Changes Already Present In Python 2.6
    - Library Changes
    - **PEP 3101**: A New Approach To String Formatting
    - Changes To Exceptions
    - Miscellaneous Other Changes
      - Operators And Special Methods
      - Builtins
    - Build and C API Changes
    - Performance

    - Porting To Python 3.0
- What’s New in Python 2.7
    - The Future for Python 2.x
    - Changes to the Handling of Deprecation Warnings
    - Python 3.1 Features
    - PEP 372: Adding an Ordered Dictionary to collections
    - PEP 378: Format Specifier for Thousands Separator
    - PEP 389: The argparse Module for Parsing Command Lines
    - PEP 391: Dictionary-Based Configuration For Logging
    - PEP 3106: Dictionary Views
    - PEP 3137: The memoryview Object
    - Other Language Changes
        - Interpreter Changes
        - Optimizations
    - New and Improved Modules
        - New module: importlib
        - New module: sysconfig
        - ttk: Themed Widgets for Tk
        - Updated module: unittest
        - Updated module: ElementTree 1.3
    - Build and C API Changes
        - Capsules
        - Port-Specific Changes: Windows
        - Port-Specific Changes: Mac OS X
        - Port-Specific Changes: FreeBSD
    - Other Changes and Fixes
    - Porting to Python 2.7
    - New Features Added to Python 2.7 Maintenance Releases
        - Two new environment variables for debug mode

        - PEP 434: IDLE Enhancement Exception for All Branches
        - PEP 466: Network Security Enhancements for Python 2.7
        - PEP 477: Backport ensurepip (PEP 453) to Python 2.7
            - Bootstrapping pip By Default
            - Documentation Changes
        - PEP 476: Enabling certificate verification by default for stdlib http clients
        - PEP 493: HTTPS verification migration tools for Python 2.7
        - New `make regen-all` build target
        - Removal of `make touch` build target
    - Acknowledgements
- What’s New in Python 2.6
    - Python 3.0
    - Changes to the Development Process
        - New Issue Tracker: Roundup
        - New Documentation Format: reStructuredText Using Sphinx
    - PEP 343: The ‘with’ statement
        - Writing Context Managers
        - The contextlib module
    - PEP 366: Explicit Relative Imports From a Main Module
    - PEP 370: Per-user `site-packages` Directory
    - PEP 371: The `multiprocessing` Package
    - PEP 3101: Advanced String Formatting
    - PEP 3105: `print` As a Function
    - PEP 3110: Exception-Handling Changes
    - PEP 3112: Byte Literals
    - PEP 3116: New I/O Library
    - PEP 3118: Revised Buffer Protocol
    - PEP 3119: Abstract Base Classes
    - PEP 3127: Integer Literal Support and Syntax

    - PEP 3129: Class Decorators
    - PEP 3141: A Type Hierarchy for Numbers
        - The **`fractions`** Module
    - Other Language Changes
        - Optimizations
        - Interpreter Changes
    - New and Improved Modules
        - The **`ast`** module
        - The **`future_builtins`** module
        - The **`json`** module: JavaScript Object Notation
        - The **`plistlib`** module: A Property-List Parser
        - ctypes Enhancements
        - Improved SSL Support
    - Deprecations and Removals
    - Build and C API Changes
        - Port-Specific Changes: Windows
        - Port-Specific Changes: Mac OS X
        - Port-Specific Changes: IRIX
    - Porting to Python 2.6
    - Acknowledgements
- What's New in Python 2.5
    - PEP 308: Conditional Expressions
    - PEP 309: Partial Function Application
    - PEP 314: Metadata for Python Software Packages v1.1
    - PEP 328: Absolute and Relative Imports
    - PEP 338: Executing Modules as Scripts
    - PEP 341: Unified try/except/finally
    - PEP 342: New Generator Features
    - PEP 343: The 'with' statement
        - Writing Context Managers
        - The contextlib module
    - PEP 352: Exceptions as New-Style Classes
    - PEP 353: Using ssize_t as the index type
    - PEP 357: The '__index__' method
    - Other Language Changes

        - Interactive Interpreter Changes
        - Optimizations
    - New, Improved, and Removed Modules
        - The ctypes package
        - The ElementTree package
        - The hashlib package
        - The sqlite3 package
        - The wsgiref package
    - Build and C API Changes
        - Port-Specific Changes
    - Porting to Python 2.5
    - Acknowledgements
- What’s New in Python 2.4
    - PEP 218: Built-In Set Objects
    - PEP 237: Unifying Long Integers and Integers
    - PEP 289: Generator Expressions
    - PEP 292: Simpler String Substitutions
    - PEP 318: Decorators for Functions and Methods
    - PEP 322: Reverse Iteration
    - PEP 324: New subprocess Module
    - PEP 327: Decimal Data Type
        - Why is Decimal needed?
        - The `Decimal` type
        - The `Context` type
    - PEP 328: Multi-line Imports
    - PEP 331: Locale-Independent Float/String Conversions
    - Other Language Changes
        - Optimizations
    - New, Improved, and Deprecated Modules
        - cookielib
        - doctest
    - Build and C API Changes
        - Port-Specific Changes
    - Porting to Python 2.4
    - Acknowledgements

- What’s New in Python 2.3
    - PEP 218: A Standard Set Datatype
    - PEP 255: Simple Generators
    - PEP 263: Source Code Encodings
    - PEP 273: Importing Modules from ZIP Archives
    - PEP 277: Unicode file name support for Windows NT
    - PEP 278: Universal Newline Support
    - PEP 279: enumerate()
    - PEP 282: The logging Package
    - PEP 285: A Boolean Type
    - PEP 293: Codec Error Handling Callbacks
    - PEP 301: Package Index and Metadata for Distutils
    - PEP 302: New Import Hooks
    - PEP 305: Comma-separated Files
    - PEP 307: Pickle Enhancements
    - Extended Slices
    - Other Language Changes
        - String Changes
        - Optimizations
    - New, Improved, and Deprecated Modules
        - Date/Time Type
        - The optparse Module
    - Pymalloc: A Specialized Object Allocator
    - Build and C API Changes
        - Port-Specific Changes
    - Other Changes and Fixes
    - Porting to Python 2.3
    - Acknowledgements
- What’s New in Python 2.2
    - Introduction
    - PEPs 252 and 253: Type and Class Changes
        - Old and New Classes
        - Descriptors
        - Multiple Inheritance: The Diamond Rule

        - Attribute Access
        - Related Links
      - PEP 234: Iterators
      - PEP 255: Simple Generators
      - PEP 237: Unifying Long Integers and Integers
      - PEP 238: Changing the Division Operator
      - Unicode Changes
      - PEP 227: Nested Scopes
      - New and Improved Modules
      - Interpreter Changes and Fixes
      - Other Changes and Fixes
      - Acknowledgements
    - What’s New in Python 2.1
      - Introduction
      - PEP 227: Nested Scopes
      - PEP 236: __future__ Directives
      - PEP 207: Rich Comparisons
      - PEP 230: Warning Framework
      - PEP 229: New Build System
      - PEP 205: Weak References
      - PEP 232: Function Attributes
      - PEP 235: Importing Modules on Case-Insensitive Platforms
      - PEP 217: Interactive Display Hook
      - PEP 208: New Coercion Model
      - PEP 241: Metadata in Python Packages
      - New and Improved Modules
      - Other Changes and Fixes
      - Acknowledgements
    - What’s New in Python 2.0
      - Introduction
      - What About Python 1.6?
      - New Development Process
      - Unicode
      - List Comprehensions
      - Augmented Assignment

    - String Methods
    - Garbage Collection of Cycles
    - Other Core Changes
        - Minor Language Changes
        - Changes to Built-in Functions
    - Porting to 2.0
    - Extending/Embedding Changes
    - Distutils: Making Modules Easy to Install
    - XML Modules
        - SAX2 Support
        - DOM Support
        - Relationship to PyXML
    - Module changes
    - New modules
    - IDLE Improvements
    - Deleted and Deprecated Modules
    - Acknowledgements
- Changelog
    - Python next
        - Windows
        - Tools/Demos
        - Tests
        - Security
        - Library
        - Documentation
        - Core and Builtins
        - C API
        - Build
    - Python 3.14.2 final
        - Security
        - Library
        - Core and Builtins
        - Library
    - Python 3.14.1 final
        - Windows
        - Tools/Demos

    - Tests
    - Security
    - Library
    - IDLE
    - Documentation
    - Core and Builtins
    - Library
    - Core and Builtins
    - Library
    - Core and Builtins
    - Library
    - Core and Builtins
    - Library
    - Core and Builtins
    - Library
    - Core and Builtins
    - Library
    - Core and Builtins
    - C API
    - Build
- Python 3.14.0 final
    - macOS
    - Windows
    - Tools/Demos
    - Security
    - Library
- Python 3.14.0 release candidate 3
    - Windows
    - Tools/Demos
    - Security
    - Library
    - Core and Builtins
- Python 3.14.0 release candidate 2
    - macOS
    - Windows
    - Library

  - Documentation
  - Core and Builtins
  - C API
  - Build
- Python 3.14.0 release candidate 1
  - Tools/Demos
  - Security
  - Library
  - Core and Builtins
  - Library
  - Core and Builtins
  - Library
  - Core and Builtins
  - C API
  - Build
- Python 3.14.0 beta 4
  - Tools/Demos
  - Tests
  - Security
  - Library
  - Documentation
  - Core and Builtins
  - C API
  - Build
- Python 3.14.0 beta 3
  - Windows
  - Tests
  - Security
  - Library
  - Documentation
  - Core and Builtins
  - Library
  - Core and Builtins
  - Library
  - Core and Builtins
  - Library

    - Core and Builtins
    - C API
    - Build
- Python 3.14.0 beta 2
    - Windows
    - Tools/Demos
    - Tests
    - Security
    - Library
    - Core and Builtins
    - C API
    - Build
- Python 3.14.0 beta 1
    - Windows
    - Tools/Demos
    - Tests
    - Security
    - Library
    - IDLE
    - Documentation
    - Core and Builtins
    - Library
    - Core and Builtins
    - Library
    - Core and Builtins
    - Library
    - C API
    - Build
- Python 3.14.0 alpha 7
    - macOS
    - Windows
    - Tools/Demos
    - Tests
    - Security
    - Library
    - Documentation

    - Core and Builtins
    - Library
    - Core and Builtins
    - C API
    - Build
- Python 3.14.0 alpha 6
    - macOS
    - Windows
    - Tools/Demos
    - Tests
    - Security
    - Library
    - Documentation
    - Core and Builtins
    - Library
    - Core and Builtins
    - Library
    - Core and Builtins
    - C API
    - Build
- Python 3.14.0 alpha 5
    - macOS
    - Tools/Demos
    - Tests
    - Security
    - Library
    - IDLE
    - Documentation
    - Core and Builtins
    - Library
    - Core and Builtins
    - Library
    - Core and Builtins
    - Library
    - C API
    - Build

- Python 3.14.0 alpha 4
    - macOS
    - Tools/Demos
    - Tests
    - Library
    - Core and Builtins
    - Library
    - Core and Builtins
    - Library
    - Core and Builtins
    - C API
    - Build
- Python 3.14.0 alpha 3
    - Windows
    - Tools/Demos
    - Tests
    - Security
    - Library
    - Documentation
    - Core and Builtins
    - Library
    - Core and Builtins
    - C API
    - Build
- Python 3.14.0 alpha 2
    - Windows
    - Tools/Demos
    - Tests
    - Security
    - Library
    - Documentation
    - Core and Builtins
    - Library
    - Core and Builtins
    - Library
    - Core and Builtins

    - Library
    - C API
    - Build
- Python 3.14.0 alpha 1
    - macOS
    - Windows
    - Tools/Demos
    - Tests
    - Security
    - Library
    - IDLE
    - Documentation
    - Core and Builtins
    - Library
    - Core and Builtins
    - Library
    - Core and Builtins
    - Library
    - Core and Builtins
    - Library
    - Core and Builtins
    - Library
    - Core and Builtins
    - Library
    - Core and Builtins
    - Library
    - Core and Builtins
    - C API
    - Build
- Python 3.13.0 beta 1
    - Security
    - Core and Builtins
    - Library
    - Documentation
    - Build
    - Windows

    - macOS
    - IDLE
    - C API
- Python 3.13.0 alpha 6
    - Core and Builtins
    - Library
    - Documentation
    - Tests
    - Build
    - Windows
    - C API
- Python 3.13.0 alpha 5
    - Security
    - Core and Builtins
    - Library
    - Documentation
    - Tests
    - Build
    - Windows
    - macOS
    - IDLE
    - Tools/Demos
    - C API
- Python 3.13.0 alpha 4
    - Security
    - Core and Builtins
    - Library
    - Documentation
    - Tests
    - Build
    - Windows
    - macOS
    - IDLE
    - Tools/Demos
    - C API
- Python 3.13.0 alpha 3

    - Security
    - Core and Builtins
    - Library
    - Documentation
    - Tests
    - Build
    - Windows
    - macOS
    - IDLE
    - C API
- Python 3.13.0 alpha 2
    - Core and Builtins
    - Library
    - Tests
    - Build
    - Windows
    - macOS
    - IDLE
    - Tools/Demos
    - C API
- Python 3.13.0 alpha 1
    - Security
    - Core and Builtins
    - Library
    - Documentation
    - Tests
    - Build
    - Windows
    - macOS
    - IDLE
    - Tools/Demos
    - C API
- Python 3.12.0 beta 1
    - Security
    - Core and Builtins
    - Library

  - Documentation
  - Tests
  - Build
  - Windows
  - macOS
  - IDLE
  - Tools/Demos
  - C API
- Python 3.12.0 alpha 7
  - Core and Builtins
  - Library
  - Documentation
  - Tests
  - Build
  - Windows
  - Tools/Demos
  - C API
- Python 3.12.0 alpha 6
  - Security
  - Core and Builtins
  - Library
  - Documentation
  - Tests
  - Build
  - Windows
  - macOS
  - C API
- Python 3.12.0 alpha 5
  - Security
  - Core and Builtins
  - Library
  - Documentation
  - Tests
  - Build
  - Windows
- Python 3.12.0 alpha 4

  - Core and Builtins
  - Library
  - Documentation
  - Tests
  - Build
  - Windows
  - macOS
  - Tools/Demos
  - C API
- Python 3.12.0 alpha 3
  - Security
  - Core and Builtins
  - Library
  - Documentation
  - Tests
  - Build
  - Windows
  - macOS
  - Tools/Demos
  - C API
- Python 3.12.0 alpha 2
  - Security
  - Core and Builtins
  - Library
  - Documentation
  - Tests
  - Build
  - Windows
  - macOS
  - C API
- Python 3.12.0 alpha 1
  - Security
  - Core and Builtins
  - Library
  - Documentation
  - Tests

    - Build
    - Windows
    - macOS
    - IDLE
    - Tools/Demos
    - C API
- Python 3.11.0 beta 1
    - Security
    - Core and Builtins
    - Library
    - Core and Builtins
    - Build
    - Core and Builtins
    - Library
    - Documentation
    - Tests
    - Build
    - Windows
    - macOS
    - Tools/Demos
    - C API
- Python 3.11.0 alpha 7
    - Core and Builtins
    - Library
    - Core and Builtins
    - Library
    - Core and Builtins
    - Library
    - Documentation
    - Tests
    - Build
    - Windows
    - macOS
    - Tools/Demos
    - C API
- Python 3.11.0 alpha 6

  - Core and Builtins
  - Library
  - Core and Builtins
  - Library
  - Core and Builtins
  - Library
  - Core and Builtins
  - Library
  - Documentation
  - Tests
  - Build
  - Windows
  - IDLE
  - C API
- Python 3.11.0 alpha 5
  - Core and Builtins
  - Library
  - Core and Builtins
  - Library
  - Build
  - Core and Builtins
  - Library
  - Documentation
  - Tests
  - Build
  - Windows
  - macOS
  - IDLE
  - C API
- Python 3.11.0 alpha 4
  - Core and Builtins
  - Library
  - Core and Builtins
  - Library
  - Core and Builtins
  - Library

    - Core and Builtins
    - Library
    - Documentation
    - Tests
    - Build
    - Windows
    - macOS
    - C API
- Python 3.11.0 alpha 3
    - Core and Builtins
    - Library
    - Core and Builtins
    - Library
    - Documentation
    - Tests
    - Build
    - Windows
    - macOS
    - C API
- Python 3.11.0 alpha 2
    - Core and Builtins
    - Library
    - Documentation
    - Tests
    - Build
    - Windows
    - macOS
    - IDLE
    - C API
- Python 3.11.0 alpha 1
    - Security
    - Core and Builtins
    - Library
    - Core and Builtins
    - Library
    - Core and Builtins

    - Library
    - Core and Builtins
    - Library
    - Core and Builtins
    - Library
    - Core and Builtins
    - Library
    - Core and Builtins
    - Library
    - Core and Builtins
    - Library
    - Core and Builtins
    - Library
    - Core and Builtins
    - Library
    - Documentation
    - Tests
    - Build
    - Windows
    - macOS
    - IDLE
    - Tools/Demos
    - C API
- Python 3.10.0 beta 1
    - Security
    - Core and Builtins
    - Library
    - Core and Builtins
    - Library
    - Core and Builtins
    - Library
    - Documentation
    - Tests
    - Build
    - Windows
    - macOS

    - IDLE
    - C API
- Python 3.10.0 alpha 7
    - Security
    - Core and Builtins
    - Library
    - Core and Builtins
    - Library
    - Documentation
    - Tests
    - Build
    - Windows
    - IDLE
    - C API
- Python 3.10.0 alpha 6
    - Security
    - Core and Builtins
    - Library
    - Core and Builtins
    - Library
    - Core and Builtins
    - Library
    - Documentation
    - Tests
    - Build
    - Windows
    - macOS
    - IDLE
    - C API
- Python 3.10.0 alpha 5
    - Security
    - Core and Builtins
    - Library
    - Core and Builtins
    - Library
    - Documentation

    - Tests
    - Build
    - Windows
    - macOS
    - IDLE
    - C API
- Python 3.10.0 alpha 4
    - Core and Builtins
    - Library
    - Core and Builtins
    - Library
    - Core and Builtins
    - Library
    - Documentation
    - Tests
    - Build
    - macOS
    - Tools/Demos
    - C API
- Python 3.10.0 alpha 3
    - Security
    - Core and Builtins
    - Library
    - Core and Builtins
    - Library
    - Core and Builtins
    - Library
    - Core and Builtins
    - Library
    - Documentation
    - Tests
    - Build
    - Windows
    - macOS
    - IDLE
    - Tools/Demos

    - C API
- Python 3.10.0 alpha 2
    - Security
    - Core and Builtins
    - Library
    - Core and Builtins
    - Library
    - Documentation
    - Tests
    - Build
    - Windows
    - macOS
    - IDLE
    - C API
- Python 3.10.0 alpha 1
    - Security
    - Core and Builtins
    - Library
    - Core and Builtins
    - Library
    - Core and Builtins
    - Library
    - Core and Builtins
    - Library
    - Core and Builtins
    - Library
    - Core and Builtins
    - Library
    - Core and Builtins
    - Library
    - Core and Builtins
    - Library
    - Core and Builtins
    - Library
    - Core and Builtins
    - Library

  - Core and Builtins
  - Library
  - Core and Builtins
  - Library
  - Core and Builtins
  - Library
  - Core and Builtins
  - Library
  - Core and Builtins
  - Library
  - Documentation
  - Tests
  - Build
  - Windows
  - macOS
  - IDLE
  - C API
- Python 3.9.0 beta 1
  - Security
  - Core and Builtins
  - Library
  - Documentation
  - Tests
  - Build
  - Windows
  - macOS
  - Tools/Demos
  - C API
- Python 3.9.0 alpha 6
  - Security
  - Core and Builtins
  - Library
  - Documentation
  - Tests
  - Build
  - Windows

  - macOS
  - IDLE
  - Tools/Demos
  - C API
- Python 3.9.0 alpha 5
  - Security
  - Core and Builtins
  - Library
  - Documentation
  - Tests
  - Build
  - Windows
  - macOS
  - IDLE
  - Tools/Demos
  - C API
- Python 3.9.0 alpha 4
  - Security
  - Core and Builtins
  - Library
  - Documentation
  - Tests
  - Build
  - Windows
  - IDLE
  - C API
- Python 3.9.0 alpha 3
  - Core and Builtins
  - Library
  - Documentation
  - Build
  - IDLE
  - C API
- Python 3.9.0 alpha 2
  - Security
  - Core and Builtins

    - Library
    - Documentation
    - Tests
    - Build
    - Windows
    - macOS
    - IDLE
    - C API
- Python 3.9.0 alpha 1
    - Security
    - Core and Builtins
    - Library
    - Documentation
    - Tests
    - Build
    - Windows
    - macOS
    - IDLE
    - Tools/Demos
    - C API
- Python 3.8.0 beta 1
    - Security
    - Core and Builtins
    - Library
    - Documentation
    - Tests
    - Build
    - Windows
    - macOS
    - IDLE
    - Tools/Demos
    - C API
- Python 3.8.0 alpha 4
    - Security
    - Core and Builtins
    - Library

  - Documentation
  - Tests
  - Build
  - Windows
  - macOS
  - IDLE
  - Tools/Demos
  - C API
- Python 3.8.0 alpha 3
  - Security
  - Core and Builtins
  - Library
  - Documentation
  - Tests
  - Build
  - Windows
  - IDLE
  - Tools/Demos
  - C API
- Python 3.8.0 alpha 2
  - Core and Builtins
  - Library
  - Documentation
  - Tests
  - Windows
  - IDLE
- Python 3.8.0 alpha 1
  - Security
  - Core and Builtins
  - Library
  - Documentation
  - Tests
  - Build
  - Windows
  - macOS
  - IDLE

    - Tools/Demos
    - C API
- Python 3.7.0 final
    - Library
    - C API
- Python 3.7.0 release candidate 1
    - Core and Builtins
    - Library
    - Documentation
    - Build
    - Windows
    - IDLE
- Python 3.7.0 beta 5
    - Core and Builtins
    - Library
    - Documentation
    - Tests
    - Build
    - macOS
    - IDLE
- Python 3.7.0 beta 4
    - Core and Builtins
    - Library
    - Documentation
    - Tests
    - Build
    - Windows
    - macOS
    - IDLE
    - Tools/Demos
- Python 3.7.0 beta 3
    - Security
    - Core and Builtins
    - Library
    - Documentation
    - Tests

  - Build
  - Windows
  - macOS
  - IDLE
  - Tools/Demos
  - C API
- Python 3.7.0 beta 2
  - Security
  - Core and Builtins
  - Library
  - Documentation
  - Tests
  - Build
  - Windows
  - macOS
  - IDLE
  - Tools/Demos
- Python 3.7.0 beta 1
  - Core and Builtins
  - Library
  - Documentation
  - Tests
  - Build
  - Windows
  - macOS
  - C API
- Python 3.7.0 alpha 4
  - Core and Builtins
  - Library
  - Documentation
  - Tests
  - Windows
  - Tools/Demos
  - C API
- Python 3.7.0 alpha 3
  - Core and Builtins

    - Library
    - Documentation
    - Tests
    - Build
    - Windows
    - macOS
    - IDLE
    - Tools/Demos
    - C API
- Python 3.7.0 alpha 2
    - Core and Builtins
    - Library
    - Documentation
    - Build
    - IDLE
    - C API
- Python 3.7.0 alpha 1
    - Security
    - Core and Builtins
    - Library
    - Documentation
    - Tests
    - Build
    - Windows
    - IDLE
    - Tools/Demos
    - C API
- Python 3.6.6 final
- Python 3.6.6 release candidate 1
    - Core and Builtins
    - Library
    - Documentation
    - Tests
    - Build
    - Windows
    - macOS

    - IDLE
    - Tools/Demos
    - C API
- Python 3.6.5 final
    - Tests
    - Build
- Python 3.6.5 release candidate 1
    - Security
    - Core and Builtins
    - Library
    - Documentation
    - Tests
    - Build
    - Windows
    - macOS
    - IDLE
    - Tools/Demos
    - C API
- Python 3.6.4 final
- Python 3.6.4 release candidate 1
    - Core and Builtins
    - Library
    - Documentation
    - Tests
    - Build
    - Windows
    - macOS
    - IDLE
    - Tools/Demos
    - C API
- Python 3.6.3 final
    - Library
    - Build
- Python 3.6.3 release candidate 1
    - Security
    - Core and Builtins

  - Library
  - Documentation
  - Tests
  - Build
  - Windows
  - IDLE
  - Tools/Demos
- Python 3.6.2 final
- Python 3.6.2 release candidate 2
  - Security
- Python 3.6.2 release candidate 1
  - Security
  - Core and Builtins
  - Library
  - IDLE
  - C API
  - Build
  - Documentation
  - Tools/Demos
  - Tests
  - Windows
- Python 3.6.1 final
  - Core and Builtins
  - Build
- Python 3.6.1 release candidate 1
  - Core and Builtins
  - Library
  - IDLE
  - Windows
  - C API
  - Documentation
  - Tests
  - Build
- Python 3.6.0 final
- Python 3.6.0 release candidate 2
  - Core and Builtins

  - Tools/Demos
  - Windows
  - Build
- Python 3.6.0 release candidate 1
  - Core and Builtins
  - Library
  - C API
  - Documentation
  - Tools/Demos
- Python 3.6.0 beta 4
  - Core and Builtins
  - Library
  - Documentation
  - Tests
  - Build
- Python 3.6.0 beta 3
  - Core and Builtins
  - Library
  - Windows
  - Build
  - Tests
- Python 3.6.0 beta 2
  - Core and Builtins
  - Library
  - Windows
  - C API
  - Build
  - Tests
- Python 3.6.0 beta 1
  - Core and Builtins
  - Library
  - IDLE
  - C API
  - Tests
  - Build
  - Tools/Demos

    - Windows
- Python 3.6.0 alpha 4
    - Core and Builtins
    - Library
    - IDLE
    - Tests
    - Windows
    - Build
- Python 3.6.0 alpha 3
    - Security
    - Core and Builtins
    - Library
    - IDLE
    - C API
    - Build
    - Tools/Demos
    - Documentation
    - Tests
- Python 3.6.0 alpha 2
    - Security
    - Core and Builtins
    - Library
    - IDLE
    - Documentation
    - Tests
    - Windows
    - Build
    - C API
    - Tools/Demos
- Python 3.6.0 alpha 1
    - Security
    - Core and Builtins
    - Library
    - IDLE
    - Documentation
    - Tests

    - Build
    - Windows
    - Tools/Demos
    - C API
- Python 3.5.5 final
- Python 3.5.5 release candidate 1
    - Security
    - Core and Builtins
    - Library
- Python 3.5.4 final
    - Library
- Python 3.5.4 release candidate 1
    - Security
    - Core and Builtins
    - Library
    - Documentation
    - Tests
    - Build
    - Windows
    - C API
- Python 3.5.3 final
- Python 3.5.3 release candidate 1
    - Security
    - Core and Builtins
    - Library
    - IDLE
    - C API
    - Documentation
    - Tests
    - Tools/Demos
    - Windows
    - Build
- Python 3.5.2 final
    - Core and Builtins
    - Tests
    - IDLE

- Python 3.5.2 release candidate 1
    - Security
    - Core and Builtins
    - Library
    - IDLE
    - Documentation
    - Tests
    - Build
    - Windows
    - Tools/Demos
- Python 3.5.1 final
    - Core and Builtins
    - Windows
- Python 3.5.1 release candidate 1
    - Core and Builtins
    - Library
    - IDLE
    - Documentation
    - Tests
    - Build
    - Windows
    - Tools/Demos
- Python 3.5.0 final
    - Build
- Python 3.5.0 release candidate 4
    - Library
    - Build
- Python 3.5.0 release candidate 3
    - Core and Builtins
    - Library
- Python 3.5.0 release candidate 2
    - Core and Builtins
    - Library
- Python 3.5.0 release candidate 1
    - Core and Builtins
    - Library

  - IDLE
  - Documentation
  - Tests
- Python 3.5.0 beta 4
  - Core and Builtins
  - Library
  - Build
- Python 3.5.0 beta 3
  - Core and Builtins
  - Library
  - Tests
  - Documentation
  - Build
- Python 3.5.0 beta 2
  - Core and Builtins
  - Library
- Python 3.5.0 beta 1
  - Core and Builtins
  - Library
  - IDLE
  - Tests
  - Documentation
  - Tools/Demos
- Python 3.5.0 alpha 4
  - Core and Builtins
  - Library
  - Build
  - Tests
  - Tools/Demos
  - C API
- Python 3.5.0 alpha 3
  - Core and Builtins
  - Library
  - Build
  - Tests
  - Tools/Demos

    - Python 3.5.0 alpha 2
      - Core and Builtins
      - Library
      - Build
      - C API
      - Windows
    - Python 3.5.0 alpha 1
      - Core and Builtins
      - Library
      - IDLE
      - Build
      - C API
      - Documentation
      - Tests
      - Tools/Demos
      - Windows
- The Python Tutorial
  - 1. Whetting Your Appetite
  - 2. Using the Python Interpreter
    - 2.1. Invoking the Interpreter
      - 2.1.1. Argument Passing
      - 2.1.2. Interactive Mode
    - 2.2. The Interpreter and Its Environment
      - 2.2.1. Source Code Encoding
  - 3. An Informal Introduction to Python
    - 3.1. Using Python as a Calculator
      - 3.1.1. Numbers
      - 3.1.2. Text
      - 3.1.3. Lists
    - 3.2. First Steps Towards Programming
  - 4. More Control Flow Tools
    - 4.1. `if` Statements
    - 4.2. `for` Statements
    - 4.3. The `range()` Function
    - 4.4. `break` and `continue` Statements
    - 4.5. `else` Clauses on Loops

        - 4.6. `pass` Statements
        - 4.7. `match` Statements
        - 4.8. Defining Functions
        - 4.9. More on Defining Functions
            - 4.9.1. Default Argument Values
            - 4.9.2. Keyword Arguments
            - 4.9.3. Special parameters
                - 4.9.3.1. Positional-or-Keyword Arguments
                - 4.9.3.2. Positional-Only Parameters
                - 4.9.3.3. Keyword-Only Arguments
                - 4.9.3.4. Function Examples
                - 4.9.3.5. Recap
            - 4.9.4. Arbitrary Argument Lists
            - 4.9.5. Unpacking Argument Lists
            - 4.9.6. Lambda Expressions
            - 4.9.7. Documentation Strings
            - 4.9.8. Function Annotations
        - 4.10. Intermezzo: Coding Style
    - 5. Data Structures
        - 5.1. More on Lists
            - 5.1.1. Using Lists as Stacks
            - 5.1.2. Using Lists as Queues
            - 5.1.3. List Comprehensions
            - 5.1.4. Nested List Comprehensions
        - 5.2. The `del` statement
        - 5.3. Tuples and Sequences
        - 5.4. Sets
        - 5.5. Dictionaries
        - 5.6. Looping Techniques
        - 5.7. More on Conditions
        - 5.8. Comparing Sequences and Other Types
    - 6. Modules
        - 6.1. More on Modules
            - 6.1.1. Executing modules as scripts
            - 6.1.2. The Module Search Path

        - 6.1.3. “Compiled” Python files
    - 6.2. Standard Modules
    - 6.3. The `dir()` Function
    - 6.4. Packages
        - 6.4.1. Importing * From a Package
        - 6.4.2. Intra-package References
        - 6.4.3. Packages in Multiple Directories
- 7. Input and Output
    - 7.1. Fancier Output Formatting
        - 7.1.1. Formatted String Literals
        - 7.1.2. The String format() Method
        - 7.1.3. Manual String Formatting
        - 7.1.4. Old string formatting
    - 7.2. Reading and Writing Files
        - 7.2.1. Methods of File Objects
        - 7.2.2. Saving structured data with `json`
- 8. Errors and Exceptions
    - 8.1. Syntax Errors
    - 8.2. Exceptions
    - 8.3. Handling Exceptions
    - 8.4. Raising Exceptions
    - 8.5. Exception Chaining
    - 8.6. User-defined Exceptions
    - 8.7. Defining Clean-up Actions
    - 8.8. Predefined Clean-up Actions
    - 8.9. Raising and Handling Multiple Unrelated Exceptions
    - 8.10. Enriching Exceptions with Notes
- 9. Classes
    - 9.1. A Word About Names and Objects
    - 9.2. Python Scopes and Namespaces
        - 9.2.1. Scopes and Namespaces Example
    - 9.3. A First Look at Classes
        - 9.3.1. Class Definition Syntax
        - 9.3.2. Class Objects
        - 9.3.3. Instance Objects

        - 9.3.4. Method Objects
        - 9.3.5. Class and Instance Variables
      - 9.4. Random Remarks
      - 9.5. Inheritance
        - 9.5.1. Multiple Inheritance
      - 9.6. Private Variables
      - 9.7. Odds and Ends
      - 9.8. Iterators
      - 9.9. Generators
      - 9.10. Generator Expressions
    - 10. Brief Tour of the Standard Library
      - 10.1. Operating System Interface
      - 10.2. File Wildcards
      - 10.3. Command Line Arguments
      - 10.4. Error Output Redirection and Program Termination
      - 10.5. String Pattern Matching
      - 10.6. Mathematics
      - 10.7. Internet Access
      - 10.8. Dates and Times
      - 10.9. Data Compression
      - 10.10. Performance Measurement
      - 10.11. Quality Control
      - 10.12. Batteries Included
    - 11. Brief Tour of the Standard Library — Part II
      - 11.1. Output Formatting
      - 11.2. Templating
      - 11.3. Working with Binary Data Record Layouts
      - 11.4. Multi-threading
      - 11.5. Logging
      - 11.6. Weak References
      - 11.7. Tools for Working with Lists
      - 11.8. Decimal Floating-Point Arithmetic
    - 12. Virtual Environments and Packages
      - 12.1. Introduction
      - 12.2. Creating Virtual Environments

        - 12.3. Managing Packages with pip
    - 13. What Now?
    - 14. Interactive Input Editing and History Substitution
        - 14.1. Tab Completion and History Editing
        - 14.2. Alternatives to the Interactive Interpreter
    - 15. Floating-Point Arithmetic: Issues and Limitations
        - 15.1. Representation Error
    - 16. Appendix
        - 16.1. Interactive Mode
            - 16.1.1. Error Handling
            - 16.1.2. Executable Python Scripts
            - 16.1.3. The Interactive Startup File
            - 16.1.4. The Customization Modules
- Python Setup and Usage
    - 1. Command line and environment
        - 1.1. Command line
            - 1.1.1. Interface options
            - 1.1.2. Generic options
            - 1.1.3. Miscellaneous options
            - 1.1.4. Controlling color
        - 1.2. Environment variables
            - 1.2.1. Debug-mode variables
    - 2. Using Python on Unix platforms
        - 2.1. Getting and installing the latest version of Python
            - 2.1.1. On Linux
                - 2.1.1.1. Installing IDLE
            - 2.1.2. On FreeBSD and OpenBSD
        - 2.2. Building Python
        - 2.3. Python-related paths and files
        - 2.4. Miscellaneous
        - 2.5. Custom OpenSSL
    - 3. Configure Python
        - 3.1. Build Requirements
            - 3.1.1. Requirements for optional modules
        - 3.2. Generated files

    - 3.2.1. configure script
- 3.3. Configure Options
    - 3.3.1. General Options
    - 3.3.2. C compiler options
    - 3.3.3. Linker options
    - 3.3.4. Options for third-party dependencies
    - 3.3.5. WebAssembly Options
    - 3.3.6. Install Options
    - 3.3.7. Performance options
    - 3.3.8. Python Debug Build
    - 3.3.9. Debug options
    - 3.3.10. Linker options
    - 3.3.11. Libraries options
    - 3.3.12. Security Options
    - 3.3.13. macOS Options
    - 3.3.14. iOS Options
    - 3.3.15. Cross Compiling Options
- 3.4. Python Build System
    - 3.4.1. Main files of the build system
    - 3.4.2. Main build steps
    - 3.4.3. Main Makefile targets
        - 3.4.3.1. make
        - 3.4.3.2. make platform
        - 3.4.3.3. make profile-opt
        - 3.4.3.4. make clean
        - 3.4.3.5. make distclean
        - 3.4.3.6. make install
        - 3.4.3.7. make test
        - 3.4.3.8. make ci
        - 3.4.3.9. make buildbottest
        - 3.4.3.10. make regen-all
    - 3.4.4. C extensions
- 3.5. Compiler and linker flags
    - 3.5.1. Preprocessor flags
    - 3.5.2. Compiler flags
    - 3.5.3. Linker flags

- 4. Using Python on Windows
  - 4.1. Python install manager
    - 4.1.1. Installation
    - 4.1.2. Basic use
    - 4.1.3. Command help
    - 4.1.4. Listing runtimes
    - 4.1.5. Installing runtimes
    - 4.1.6. Offline installs
    - 4.1.7. Uninstalling runtimes
    - 4.1.8. Configuration
    - 4.1.9. Shebang lines
    - 4.1.10. Advanced installation
    - 4.1.11. Administrative configuration
    - 4.1.12. Installing free-threaded binaries
    - 4.1.13. Troubleshooting
  - 4.2. The embeddable package
    - 4.2.1. Python application
    - 4.2.2. Embedding Python
  - 4.3. The nuget.org packages
    - 4.3.1. Free-threaded packages
  - 4.4. Alternative bundles
  - 4.5. Supported Windows versions
  - 4.6. Removing the MAX_PATH limitation
  - 4.7. UTF-8 mode
  - 4.8. Finding modules
  - 4.9. Additional modules
    - 4.9.1. PyWin32
    - 4.9.2. cx_Freeze
  - 4.10. Compiling Python on Windows
  - 4.11. The full installer (deprecated)
    - 4.11.1. Installation steps
    - 4.11.2. Removing the MAX_PATH limitation
    - 4.11.3. Installing without UI
    - 4.11.4. Installing without downloading
    - 4.11.5. Modifying an install
    - 4.11.6. Installing free-threaded binaries

        - 4.12. Python launcher for Windows (deprecated)
            - 4.12.1. Getting started
                - 4.12.1.1. From the command-line
                - 4.12.1.2. Virtual environments
                - 4.12.1.3. From a script
                - 4.12.1.4. From file associations
            - 4.12.2. Shebang lines
            - 4.12.3. Arguments in shebang lines
            - 4.12.4. Customization
                - 4.12.4.1. Customization via INI files
                - 4.12.4.2. Customizing default Python versions
            - 4.12.5. Diagnostics
            - 4.12.6. Dry run
            - 4.12.7. Install on demand
            - 4.12.8. Return codes
    - 5. Using Python on macOS
        - 5.1. Using Python for macOS from `python.org`
            - 5.1.1. Installation steps
            - 5.1.2. How to run a Python script
        - 5.2. Alternative Distributions
        - 5.3. Installing Additional Python Packages
        - 5.4. GUI Programming
        - 5.5. Advanced Topics
            - 5.5.1. Installing Free-threaded Binaries
            - 5.5.2. Installing using the command line
            - 5.5.3. Distributing Python Applications
            - 5.5.4. App Store Compliance
        - 5.6. Other Resources
    - 6. Using Python on Android
        - 6.1. Adding Python to an Android app
        - 6.2. Building a Python package for Android
    - 7. Using Python on iOS
        - 7.1. Python at runtime on iOS
            - 7.1.1. iOS version compatibility
            - 7.1.2. Platform identification

            - 7.1.3. Standard library availability
            - 7.1.4. Binary extension modules
            - 7.1.5. Compiler stub binaries
        - 7.2. Installing Python on iOS
            - 7.2.1. Tools for building iOS apps
            - 7.2.2. Adding Python to an iOS project
            - 7.2.3. Testing a Python package
        - 7.3. App Store Compliance
            - 7.3.1. Incompatible code in the standard library
            - 7.3.2. Privacy manifests
    - 8. Editors and IDEs
        - 8.1. IDLE — Python editor and shell
        - 8.2. Other Editors and IDEs
- The Python Language Reference
    - 1. Introduction
        - 1.1. Alternate Implementations
        - 1.2. Notation
            - 1.2.1. Lexical and Syntactic definitions
    - 2. Lexical analysis
        - 2.1. Line structure
            - 2.1.1. Logical lines
            - 2.1.2. Physical lines
            - 2.1.3. Comments
            - 2.1.4. Encoding declarations
            - 2.1.5. Explicit line joining
            - 2.1.6. Implicit line joining
            - 2.1.7. Blank lines
            - 2.1.8. Indentation
            - 2.1.9. Whitespace between tokens
            - 2.1.10. End marker
        - 2.2. Other tokens
        - 2.3. Names (identifiers and keywords)
            - 2.3.1. Keywords
            - 2.3.2. Soft Keywords
            - 2.3.3. Reserved classes of identifiers

        - 2.3.4. Non-ASCII characters in names
    - 2.4. Literals
    - 2.5. String and Bytes literals
        - 2.5.1. Triple-quoted strings
        - 2.5.2. String prefixes
        - 2.5.3. Formal grammar
        - 2.5.4. Escape sequences
            - 2.5.4.1. Ignored end of line
            - 2.5.4.2. Escaped characters
            - 2.5.4.3. Octal character
            - 2.5.4.4. Hexadecimal character
            - 2.5.4.5. Named Unicode character
            - 2.5.4.6. Hexadecimal Unicode characters
            - 2.5.4.7. Unrecognized escape sequences
        - 2.5.5. Bytes literals
        - 2.5.6. Raw string literals
        - 2.5.7. f-strings
        - 2.5.8. t-strings
        - 2.5.9. Formal grammar for f-strings
    - 2.6. Numeric literals
        - 2.6.1. Integer literals
        - 2.6.2. Floating-point literals
        - 2.6.3. Imaginary literals
    - 2.7. Operators and delimiters
- 3. Data model
    - 3.1. Objects, values and types
    - 3.2. The standard type hierarchy
        - 3.2.1. None
        - 3.2.2. NotImplemented
        - 3.2.3. Ellipsis
        - 3.2.4. `numbers.Number`
            - 3.2.4.1. `numbers.Integral`
            - 3.2.4.2. `numbers.Real` (`float`)
            - 3.2.4.3. `numbers.Complex` (`complex`)
        - 3.2.5. Sequences

        - 3.2.5.1. Immutable sequences
        - 3.2.5.2. Mutable sequences
    - 3.2.6. Set types
    - 3.2.7. Mappings
        - 3.2.7.1. Dictionaries
    - 3.2.8. Callable types
        - 3.2.8.1. User-defined functions
            - 3.2.8.1.1. Special read-only attributes
            - 3.2.8.1.2. Special writable attributes
        - 3.2.8.2. Instance methods
        - 3.2.8.3. Generator functions
        - 3.2.8.4. Coroutine functions
        - 3.2.8.5. Asynchronous generator functions
        - 3.2.8.6. Built-in functions
        - 3.2.8.7. Built-in methods
        - 3.2.8.8. Classes
        - 3.2.8.9. Class Instances
    - 3.2.9. Modules
        - 3.2.9.1. Import-related attributes on module objects
        - 3.2.9.2. Other writable attributes on module objects
        - 3.2.9.3. Module dictionaries
    - 3.2.10. Custom classes
        - 3.2.10.1. Special attributes
        - 3.2.10.2. Special methods
    - 3.2.11. Class instances
        - 3.2.11.1. Special attributes
    - 3.2.12. I/O objects (also known as file objects)
    - 3.2.13. Internal types
        - 3.2.13.1. Code objects
            - 3.2.13.1.1. Special read-only attributes

        - 3.2.13.1.2. Methods on code objects
      - 3.2.13.2. Frame objects
        - 3.2.13.2.1. Special read-only attributes
        - 3.2.13.2.2. Special writable attributes
        - 3.2.13.2.3. Frame object methods
      - 3.2.13.3. Traceback objects
      - 3.2.13.4. Slice objects
      - 3.2.13.5. Static method objects
      - 3.2.13.6. Class method objects
- 3.3. Special method names
  - 3.3.1. Basic customization
  - 3.3.2. Customizing attribute access
    - 3.3.2.1. Customizing module attribute access
    - 3.3.2.2. Implementing Descriptors
    - 3.3.2.3. Invoking Descriptors
    - 3.3.2.4. __slots__
  - 3.3.3. Customizing class creation
    - 3.3.3.1. Metaclasses
    - 3.3.3.2. Resolving MRO entries
    - 3.3.3.3. Determining the appropriate metaclass
    - 3.3.3.4. Preparing the class namespace
    - 3.3.3.5. Executing the class body
    - 3.3.3.6. Creating the class object
    - 3.3.3.7. Uses for metaclasses
  - 3.3.4. Customizing instance and subclass checks
  - 3.3.5. Emulating generic types
    - 3.3.5.1. The purpose of *__class_getitem__*
    - 3.3.5.2. *__class_getitem__* versus *__getitem__*
  - 3.3.6. Emulating callable objects

        - 3.3.7. Emulating container types
        - 3.3.8. Emulating numeric types
        - 3.3.9. With Statement Context Managers
        - 3.3.10. Customizing positional arguments in class pattern matching
        - 3.3.11. Emulating buffer types
        - 3.3.12. Annotations
        - 3.3.13. Special method lookup
    - 3.4. Coroutines
        - 3.4.1. Awaitable Objects
        - 3.4.2. Coroutine Objects
        - 3.4.3. Asynchronous Iterators
        - 3.4.4. Asynchronous Context Managers
- 4. Execution model
    - 4.1. Structure of a program
    - 4.2. Naming and binding
        - 4.2.1. Binding of names
        - 4.2.2. Resolution of names
        - 4.2.3. Annotation scopes
        - 4.2.4. Lazy evaluation
        - 4.2.5. Builtins and restricted execution
        - 4.2.6. Interaction with dynamic features
    - 4.3. Exceptions
    - 4.4. Runtime Components
        - 4.4.1. General Computing Model
        - 4.4.2. Python Runtime Model
- 5. The import system
    - 5.1. `importlib`
    - 5.2. Packages
        - 5.2.1. Regular packages
        - 5.2.2. Namespace packages
    - 5.3. Searching
        - 5.3.1. The module cache
        - 5.3.2. Finders and loaders
        - 5.3.3. Import hooks
        - 5.3.4. The meta path

        - 5.4. Loading
            - 5.4.1. Loaders
            - 5.4.2. Submodules
            - 5.4.3. Module specs
            - 5.4.4. __path__ attributes on modules
            - 5.4.5. Module reprs
            - 5.4.6. Cached bytecode invalidation
        - 5.5. The Path Based Finder
            - 5.5.1. Path entry finders
            - 5.5.2. Path entry finder protocol
        - 5.6. Replacing the standard import system
        - 5.7. Package Relative Imports
        - 5.8. Special considerations for __main__
            - 5.8.1. __main__.__spec__
        - 5.9. References
    - 6. Expressions
        - 6.1. Arithmetic conversions
        - 6.2. Atoms
            - 6.2.1. Identifiers (Names)
                - 6.2.1.1. Private name mangling
            - 6.2.2. Literals
                - 6.2.2.1. String literal concatenation
            - 6.2.3. Parenthesized forms
            - 6.2.4. Displays for lists, sets and dictionaries
            - 6.2.5. List displays
            - 6.2.6. Set displays
            - 6.2.7. Dictionary displays
            - 6.2.8. Generator expressions
            - 6.2.9. Yield expressions
                - 6.2.9.1. Generator-iterator methods
                - 6.2.9.2. Examples
                - 6.2.9.3. Asynchronous generator functions
                - 6.2.9.4. Asynchronous generator-iterator methods
        - 6.3. Primaries

        - 6.3.1. Attribute references
        - 6.3.2. Subscriptions
        - 6.3.3. Slicings
        - 6.3.4. Calls
    - 6.4. Await expression
    - 6.5. The power operator
    - 6.6. Unary arithmetic and bitwise operations
    - 6.7. Binary arithmetic operations
    - 6.8. Shifting operations
    - 6.9. Binary bitwise operations
    - 6.10. Comparisons
        - 6.10.1. Value comparisons
        - 6.10.2. Membership test operations
        - 6.10.3. Identity comparisons
    - 6.11. Boolean operations
    - 6.12. Assignment expressions
    - 6.13. Conditional expressions
    - 6.14. Lambdas
    - 6.15. Expression lists
    - 6.16. Evaluation order
    - 6.17. Operator precedence
- 7. Simple statements
    - 7.1. Expression statements
    - 7.2. Assignment statements
        - 7.2.1. Augmented assignment statements
        - 7.2.2. Annotated assignment statements
    - 7.3. The `assert` statement
    - 7.4. The `pass` statement
    - 7.5. The `del` statement
    - 7.6. The `return` statement
    - 7.7. The `yield` statement
    - 7.8. The `raise` statement
    - 7.9. The `break` statement
    - 7.10. The `continue` statement
    - 7.11. The `import` statement
        - 7.11.1. Future statements

    - 7.12. The `global` statement
    - 7.13. The `nonlocal` statement
    - 7.14. The `type` statement
  - 8. Compound statements
    - 8.1. The `if` statement
    - 8.2. The `while` statement
    - 8.3. The `for` statement
    - 8.4. The `try` statement
      - 8.4.1. `except` clause
      - 8.4.2. `except*` clause
      - 8.4.3. `else` clause
      - 8.4.4. `finally` clause
    - 8.5. The `with` statement
    - 8.6. The `match` statement
      - 8.6.1. Overview
      - 8.6.2. Guards
      - 8.6.3. Irrefutable Case Blocks
      - 8.6.4. Patterns
        - 8.6.4.1. OR Patterns
        - 8.6.4.2. AS Patterns
        - 8.6.4.3. Literal Patterns
        - 8.6.4.4. Capture Patterns
        - 8.6.4.5. Wildcard Patterns
        - 8.6.4.6. Value Patterns
        - 8.6.4.7. Group Patterns
        - 8.6.4.8. Sequence Patterns
        - 8.6.4.9. Mapping Patterns
        - 8.6.4.10. Class Patterns
    - 8.7. Function definitions
    - 8.8. Class definitions
    - 8.9. Coroutines
      - 8.9.1. Coroutine function definition
      - 8.9.2. The `async for` statement
      - 8.9.3. The `async with` statement
    - 8.10. Type parameter lists
      - 8.10.1. Generic functions

        - 8.10.2. Generic classes
        - 8.10.3. Generic type aliases
      - 8.11. Annotations
    - 9. Top-level components
      - 9.1. Complete Python programs
      - 9.2. File input
      - 9.3. Interactive input
      - 9.4. Expression input
    - 10. Full Grammar specification
- The Python Standard Library
  - Introduction
    - Notes on availability
      - WebAssembly platforms
      - Mobile platforms
  - Built-in Functions
  - Built-in Constants
    - Constants added by the `site` module
  - Built-in Types
    - Truth Value Testing
    - Boolean Operations — `and`, `or`, `not`
    - Comparisons
    - Numeric Types — `int`, `float`, `complex`
      - Bitwise Operations on Integer Types
      - Additional Methods on Integer Types
      - Additional Methods on Float
      - Additional Methods on Complex
      - Hashing of numeric types
    - Boolean Type - `bool`
    - Iterator Types
      - Generator Types
    - Sequence Types — `list`, `tuple`, `range`
      - Common Sequence Operations
      - Immutable Sequence Types
      - Mutable Sequence Types
      - Lists
      - Tuples

    - Ranges
- Text and Binary Sequence Type Methods Summary
- Text Sequence Type — `str`
    - String Methods
    - Formatted String Literals (f-strings)
        - Debug specifier
        - Conversion specifier
        - Format specifier
    - Template String Literals (t-strings)
    - `printf`-style String Formatting
- Binary Sequence Types — `bytes`, `bytearray`, `memoryview`
    - Bytes Objects
    - Bytearray Objects
    - Bytes and Bytearray Operations
    - `printf`-style Bytes Formatting
    - Memory Views
- Set Types — `set`, `frozenset`
- Mapping Types — `dict`
    - Dictionary view objects
- Context Manager Types
- Type Annotation Types — Generic Alias, Union
    - Generic Alias Type
        - Standard Generic Classes
        - Special Attributes of `GenericAlias` objects
    - Union Type
- Other Built-in Types
    - Modules
    - Classes and Class Instances
    - Functions
    - Methods
    - Code Objects
    - Type Objects
    - The Null Object
    - The Ellipsis Object

      - The NotImplemented Object
      - Internal Objects
    - Special Attributes
    - Integer string conversion length limitation
      - Affected APIs
      - Configuring the limit
      - Recommended configuration
  - Built-in Exceptions
    - Exception context
    - Inheriting from built-in exceptions
    - Base classes
    - Concrete exceptions
      - OS exceptions
    - Warnings
    - Exception groups
    - Exception hierarchy
  - Text Processing Services
    - `string` — Common string operations
      - String constants
      - Custom String Formatting
      - Format String Syntax
        - Format Specification Mini-Language
        - Format examples
      - Template strings ($-strings)
      - Helper functions
    - `string.templatelib` — Support for template string literals
      - Template strings
      - Types
      - Helper functions
    - `re` — Regular expression operations
      - Regular Expression Syntax
      - Module Contents
        - Flags
        - Functions
        - Exceptions

        - Regular Expression Objects
        - Match Objects
        - Regular Expression Examples
            - Checking for a Pair
            - Simulating scanf()
            - search() vs. match()
            - Making a Phonebook
            - Text Munging
            - Finding all Adverbs
            - Finding all Adverbs and their Positions
            - Raw String Notation
            - Writing a Tokenizer
    - `difflib` — Helpers for computing deltas
        - SequenceMatcher Objects
        - SequenceMatcher Examples
        - Differ Objects
        - Differ Example
        - A command-line interface to difflib
        - ndiff example
    - `textwrap` — Text wrapping and filling
    - `unicodedata` — Unicode Database
    - `stringprep` — Internet String Preparation
    - `readline` — GNU readline interface
        - Init file
        - Line buffer
        - History file
        - History list
        - Startup hooks
        - Completion
        - Example
    - `rlcompleter` — Completion function for GNU readline
- Binary Data Services
    - `struct` — Interpret bytes as packed binary data
        - Functions and Exceptions
        - Format Strings

            - Byte Order, Size, and Alignment
            - Format Characters
            - Examples
        - Applications
            - Native Formats
            - Standard Formats
        - Classes
    - `codecs` — Codec registry and base classes
        - Codec Base Classes
            - Error Handlers
            - Stateless Encoding and Decoding
            - Incremental Encoding and Decoding
                - IncrementalEncoder Objects
                - IncrementalDecoder Objects
            - Stream Encoding and Decoding
                - StreamWriter Objects
                - StreamReader Objects
                - StreamReaderWriter Objects
                - StreamRecoder Objects
        - Encodings and Unicode
        - Standard Encodings
        - Python Specific Encodings
            - Text Encodings
            - Binary Transforms
            - Standalone Codec Functions
            - Text Transforms
        - `encodings` — Encodings package
        - `encodings.idna` — Internationalized Domain Names in Applications
        - `encodings.mbcs` — Windows ANSI codepage
        - `encodings.utf_8_sig` — UTF-8 codec with BOM signature
- Data Types
    - `datetime` — Basic date and time types
        - Aware and Naive Objects
        - Constants

  - Available Types
    - Common Properties
    - Determining if an Object is Aware or Naive
  - `timedelta` Objects
    - Examples of usage: `timedelta`
  - `date` Objects
    - Examples of Usage: `date`
  - `datetime` Objects
    - Examples of Usage: `datetime`
  - `time` Objects
    - Examples of Usage: `time`
  - `tzinfo` Objects
  - `timezone` Objects
  - `strftime()` and `strptime()` Behavior
    - `strftime()` and `strptime()` Format Codes
    - Technical Detail
- `zoneinfo` — IANA time zone support
  - Using `ZoneInfo`
  - Data sources
    - Configuring the data sources
      - Compile-time configuration
      - Environment configuration
      - Runtime configuration
  - The `ZoneInfo` class
    - String representations
    - Pickle serialization
  - Functions
  - Globals
  - Exceptions and warnings
- `calendar` — General calendar-related functions
  - Command-line usage
- `collections` — Container datatypes
  - `ChainMap` objects
    - `ChainMap` Examples and Recipes
  - `Counter` objects

    - `deque` objects
        - `deque` Recipes
    - `defaultdict` objects
        - `defaultdict` Examples
    - `namedtuple()` Factory Function for Tuples with Named Fields
    - `OrderedDict` objects
        - `OrderedDict` Examples and Recipes
    - `UserDict` objects
    - `UserList` objects
    - `UserString` objects
- `collections.abc` — Abstract Base Classes for Containers
    - Collections Abstract Base Classes
    - Collections Abstract Base Classes – Detailed Descriptions
    - Examples and Recipes
- `heapq` — Heap queue algorithm
    - Basic Examples
    - Other Applications
    - Priority Queue Implementation Notes
    - Theory
- `bisect` — Array bisection algorithm
    - Performance Notes
    - Searching Sorted Lists
    - Examples
- `array` — Efficient arrays of numeric values
- `weakref` — Weak references
    - Weak Reference Objects
    - Example
    - Finalizer Objects
    - Comparing finalizers with `__del__()` methods
- `types` — Dynamic type creation and names for built-in types
    - Dynamic Type Creation
    - Standard Interpreter Types

        - Additional Utility Classes and Functions
        - Coroutine Utility Functions
    - `copy` — Shallow and deep copy operations
    - `pprint` — Data pretty printer
        - Functions
        - PrettyPrinter Objects
        - Example
    - `reprlib` — Alternate `repr()` implementation
        - Repr Objects
        - Subclassing Repr Objects
    - `enum` — Support for enumerations
        - Module Contents
        - Data Types
            - Supported `__dunder__` names
            - Supported `_sunder_` names
        - Utilities and Decorators
        - Notes
    - `graphlib` — Functionality to operate with graph-like structures
        - Exceptions
- Numeric and Mathematical Modules
    - `numbers` — Numeric abstract base classes
        - The numeric tower
        - Notes for type implementers
            - Adding More Numeric ABCs
            - Implementing the arithmetic operations
    - `math` — Mathematical functions
        - Number-theoretic functions
        - Floating point arithmetic
        - Floating point manipulation functions
        - Power, exponential and logarithmic functions
        - Summation and product functions
        - Angular conversion
        - Trigonometric functions
        - Hyperbolic functions

    - Special functions
    - Constants
- `cmath` — Mathematical functions for complex numbers
    - Conversions to and from polar coordinates
    - Power and logarithmic functions
    - Trigonometric functions
    - Hyperbolic functions
    - Classification functions
    - Constants
- `decimal` — Decimal fixed-point and floating-point arithmetic
    - Quick-start tutorial
    - Decimal objects
        - Logical operands
    - Context objects
    - Constants
    - Rounding modes
    - Signals
    - Floating-point notes
        - Mitigating round-off error with increased precision
        - Special values
    - Working with threads
    - Recipes
    - Decimal FAQ
- `fractions` — Rational numbers
- `random` — Generate pseudo-random numbers
    - Bookkeeping functions
    - Functions for bytes
    - Functions for integers
    - Functions for sequences
    - Discrete distributions
    - Real-valued distributions
    - Alternative Generator
    - Notes on Reproducibility

            - Examples
            - Recipes
            - Command-line usage
            - Command-line example
        - `statistics` — Mathematical statistics functions
            - Averages and measures of central location
            - Measures of spread
            - Statistics for relations between two inputs
            - Function details
            - Exceptions
            - `NormalDist` objects
            - Examples and Recipes
                - Classic probability problems
                - Monte Carlo inputs for simulations
                - Approximating binomial distributions
                - Naive bayesian classifier
    - Functional Programming Modules
        - `itertools` — Functions creating iterators for efficient looping
            - Itertool Functions
            - Itertools Recipes
        - `functools` — Higher-order functions and operations on callable objects
            - `partial` Objects
        - `operator` — Standard operators as functions
            - Mapping Operators to Functions
            - In-place Operators
    - File and Directory Access
        - `pathlib` — Object-oriented filesystem paths
            - Basic use
            - Exceptions
            - Pure paths
                - General properties
                - Operators
                - Accessing individual parts
                - Methods and properties

    - Concrete paths
        - Parsing and generating URIs
        - Expanding and resolving paths
        - Querying file type and status
        - Reading and writing files
        - Reading directories
        - Creating files and directories
        - Copying, moving and deleting
        - Permissions and ownership
    - Pattern language
    - Comparison to the `glob` module
    - Comparison to the `os` and `os.path` modules
        - Corresponding tools
    - Protocols
- `os.path` — Common pathname manipulations
- `stat` — Interpreting `stat()` results
- `filecmp` — File and Directory Comparisons
    - The `dircmp` class
- `tempfile` — Generate temporary files and directories
    - Examples
    - Deprecated functions and variables
- `glob` — Unix style pathname pattern expansion
    - Examples
- `fnmatch` — Unix filename pattern matching
- `linecache` — Random access to text lines
- `shutil` — High-level file operations
    - Directory and files operations
        - Platform-dependent efficient copy operations
        - copytree example
        - rmtree example
    - Archiving operations
        - Archiving example
        - Archiving example with *base_dir*
    - Querying the size of the output terminal

- Data Persistence
    - `pickle` — Python object serialization
        - Relationship to other Python modules
            - Comparison with `marshal`
            - Comparison with `json`
        - Data stream format
        - Module Interface
        - What can be pickled and unpickled?
        - Pickling Class Instances
            - Persistence of External Objects
            - Dispatch Tables
            - Handling Stateful Objects
        - Custom Reduction for Types, Functions, and Other Objects
        - Out-of-band Buffers
            - Provider API
            - Consumer API
            - Example
        - Restricting Globals
        - Performance
        - Examples
        - Command-line interface
    - `copyreg` — Register `pickle` support functions
        - Example
    - `shelve` — Python object persistence
        - Restrictions
        - Example
    - `marshal` — Internal Python object serialization
    - `dbm` — Interfaces to Unix “databases”
        - `dbm.sqlite3` — SQLite backend for dbm
        - `dbm.gnu` — GNU database manager
        - `dbm.ndbm` — New Database Manager
        - `dbm.dumb` — Portable DBM implementation
    - `sqlite3` — DB-API 2.0 interface for SQLite databases
        - Tutorial

- Reference
    - Module functions
    - Module constants
    - Connection objects
    - Cursor objects
    - Row objects
    - Blob objects
    - PrepareProtocol objects
    - Exceptions
    - SQLite and Python types
    - Default adapters and converters (deprecated)
    - Command-line interface
- How-to guides
    - How to use placeholders to bind values in SQL queries
    - How to adapt custom Python types to SQLite values
        - How to write adaptable objects
        - How to register adapter callables
    - How to convert SQLite values to custom Python types
    - Adapter and converter recipes
    - How to use connection shortcut methods
    - How to use the connection context manager
    - How to work with SQLite URIs
    - How to create and use row factories
    - How to handle non-UTF-8 text encodings
- Explanation
    - Transaction control
        - Transaction control via the `autocommit` attribute
        - Transaction control via the `isolation_level` attribute

- Data Compression and Archiving
  - The `compression` package
  - `compression.zstd` — Compression compatible with the Zstandard format
    - Exceptions
    - Reading and writing compressed files
    - Compressing and decompressing data in memory
    - Zstandard dictionaries
    - Advanced parameter control
    - Miscellaneous
    - Examples
  - `zlib` — Compression compatible with **gzip**
  - `gzip` — Support for **gzip** files
    - Examples of usage
    - Command-line interface
      - Command-line options
  - `bz2` — Support for **bzip2** compression
    - (De)compression of files
    - Incremental (de)compression
    - One-shot (de)compression
    - Examples of usage
  - `lzma` — Compression using the LZMA algorithm
    - Reading and writing compressed files
    - Compressing and decompressing data in memory
    - Miscellaneous
    - Specifying custom filter chains
    - Examples
  - `zipfile` — Work with ZIP archives
    - ZipFile objects
    - Path objects
    - PyZipFile objects
    - ZipInfo objects
    - Command-line interface
      - Command-line options

        - Decompression pitfalls
            - From file itself
            - File system limitations
            - Resources limitations
            - Interruption
            - Default behaviors of extraction
    - `tarfile` — Read and write tar archive files
        - TarFile Objects
        - TarInfo Objects
        - Extraction filters
            - Default named filters
            - Filter errors
            - Hints for further verification
            - Supporting older Python versions
            - Stateful extraction filter example
        - Command-Line Interface
            - Command-line options
        - Examples
            - Reading examples
            - Writing examples
        - Supported tar formats
        - Unicode issues
- File Formats
    - `csv` — CSV File Reading and Writing
        - Module Contents
        - Dialects and Formatting Parameters
        - Reader Objects
        - Writer Objects
        - Examples
    - `configparser` — Configuration file parser
        - Quick Start
        - Supported Datatypes
        - Fallback Values
        - Supported INI File Structure
        - Unnamed Sections
        - Interpolation of values

        - Mapping Protocol Access
        - Customizing Parser Behaviour
        - Legacy API Examples
        - ConfigParser Objects
        - RawConfigParser Objects
        - Exceptions
    - `tomllib` — Parse TOML files
        - Examples
        - Conversion Table
    - `netrc` — netrc file processing
        - netrc Objects
    - `plistlib` — Generate and parse Apple `.plist` files
        - Examples
- Cryptographic Services
    - `hashlib` — Secure hashes and message digests
        - Hash algorithms
        - Usage
        - Constructors
        - Attributes
        - Hash Objects
        - SHAKE variable length digests
        - File hashing
        - Key derivation
        - BLAKE2
            - Creating hash objects
            - Constants
            - Examples
                - Simple hashing
                - Using different digest sizes
                - Keyed hashing
                - Randomized hashing
                - Personalization
                - Tree mode
            - Credits
    - `hmac` — Keyed-Hashing for Message Authentication

    - `secrets` — Generate secure random numbers for managing secrets
        - Random numbers
        - Generating tokens
            - How many bytes should tokens use?
        - Other functions
        - Recipes and best practices
- Generic Operating System Services
    - `os` — Miscellaneous operating system interfaces
        - File Names, Command Line Arguments, and Environment Variables
        - Python UTF-8 Mode
        - Process Parameters
        - File Object Creation
        - File Descriptor Operations
            - Querying the size of a terminal
            - Inheritance of File Descriptors
        - Files and Directories
            - Timer File Descriptors
            - Linux extended attributes
        - Process Management
        - Interface to the scheduler
        - Miscellaneous System Information
        - Random numbers
    - `io` — Core tools for working with streams
        - Overview
            - Text I/O
            - Binary I/O
            - Raw I/O
        - Text Encoding
            - Opt-in EncodingWarning
        - High-level Module Interface
        - Class hierarchy
            - I/O Base Classes
            - Raw File I/O
            - Buffered Streams

        - Text I/O
    - Static Typing
    - Performance
        - Binary I/O
        - Text I/O
        - Multi-threading
        - Reentrancy
- `time` — Time access and conversions
    - Functions
    - Clock ID Constants
    - Timezone Constants
- `logging` — Logging facility for Python
    - Logger Objects
    - Logging Levels
    - Handler Objects
    - Formatter Objects
    - Filter Objects
    - LogRecord Objects
    - LogRecord attributes
    - LoggerAdapter Objects
    - Thread Safety
    - Module-Level Functions
    - Module-Level Attributes
    - Integration with the warnings module
- `logging.config` — Logging configuration
    - Configuration functions
    - Security considerations
    - Configuration dictionary schema
        - Dictionary Schema Details
        - Incremental Configuration
        - Object connections
        - User-defined objects
        - Handler configuration order
        - Access to external objects
        - Access to internal objects
        - Import resolution and custom importers

        - Configuring QueueHandler and QueueListener
    - Configuration file format
- `logging.handlers` — Logging handlers
    - StreamHandler
    - FileHandler
    - NullHandler
    - WatchedFileHandler
    - BaseRotatingHandler
    - RotatingFileHandler
    - TimedRotatingFileHandler
    - SocketHandler
    - DatagramHandler
    - SysLogHandler
    - NTEventLogHandler
    - SMTPHandler
    - MemoryHandler
    - HTTPHandler
    - QueueHandler
    - QueueListener
- `platform` — Access to underlying platform’s identifying data
    - Cross platform
    - Java platform
    - Windows platform
    - macOS platform
    - iOS platform
    - Unix platforms
    - Linux platforms
    - Android platform
    - Command-line usage
- `errno` — Standard errno system symbols
- `ctypes` — A foreign function library for Python
    - ctypes tutorial
        - Loading dynamic link libraries
        - Accessing functions from loaded dlls

    - Calling functions
    - Fundamental data types
    - Calling functions, continued
    - Calling variadic functions
    - Calling functions with your own custom data types
    - Specifying the required argument types (function prototypes)
    - Return types
    - Passing pointers (or: passing parameters by reference)
    - Structures and unions
    - Structure/union layout, alignment and byte order
    - Bit fields in structures and unions
    - Arrays
    - Pointers
    - Thread safety without the GIL
    - Type conversions
    - Incomplete Types
    - Callback functions
    - Accessing values exported from dlls
    - Surprises
    - Variable-sized data types
- ctypes reference
    - Finding shared libraries
    - Listing loaded shared libraries
    - Loading shared libraries
    - Foreign functions
    - Function prototypes
    - Utility functions
    - Data types
    - Fundamental data types
    - Structured data types
    - Arrays and pointers
    - Exceptions

- Command-line interface libraries
  - `argparse` — Parser for command-line options, arguments and subcommands
    - ArgumentParser objects
      - prog
      - usage
      - description
      - epilog
      - parents
      - formatter_class
      - prefix_chars
      - fromfile_prefix_chars
      - argument_default
      - allow_abbrev
      - conflict_handler
      - add_help
      - exit_on_error
      - suggest_on_error
      - color
    - The add_argument() method
      - name or flags
      - action
      - nargs
      - const
      - default
      - type
      - choices
      - required
      - help
      - metavar
      - dest
      - deprecated
      - Action classes
    - The parse_args() method
      - Option value syntax
      - Invalid arguments

        - Arguments containing `-`
        - Argument abbreviations (prefix matching)
        - Beyond `sys.argv`
        - The Namespace object
    - Other utilities
        - Subcommands
        - FileType objects
        - Argument groups
        - Mutual exclusion
        - Parser defaults
        - Printing help
        - Partial parsing
        - Customizing file parsing
        - Exiting methods
        - Intermixed parsing
        - Registering custom types or actions
    - Exceptions
        - Argparse Tutorial
            - Concepts
            - The basics
            - Introducing Positional arguments
            - Introducing Optional arguments
                - Short options
            - Combining Positional and Optional arguments
            - Getting a little more advanced
                - Specifying ambiguous arguments
                - Conflicting options
            - How to translate the argparse output
            - Custom type converters
            - Conclusion
        - Migrating `optparse` code to `argparse`
- `optparse` — Parser for command line options

- Choosing an argument parsing library
- Introduction
- Background
    - Terminology
    - What are options for?
    - What are positional arguments for?
- Tutorial
    - Understanding option actions
    - The store action
    - Handling boolean (flag) options
    - Other actions
    - Default values
    - Generating help
        - Grouping Options
    - Printing a version string
    - How `optparse` handles errors
    - Putting it all together
- Reference Guide
    - Creating the parser
    - Populating the parser
    - Defining options
    - Option attributes
    - Standard option actions
    - Standard option types
    - Parsing arguments
    - Querying and manipulating your option parser
    - Conflicts between options
    - Cleanup
    - Other methods
- Option Callbacks
    - Defining a callback option
    - How callbacks are called
    - Raising errors in a callback
    - Callback example 1: trivial callback
    - Callback example 2: check option order

            - Callback example 3: check option order (generalized)
            - Callback example 4: check arbitrary condition
            - Callback example 5: fixed arguments
            - Callback example 6: variable arguments
        - Extending **`optparse`**
            - Adding new types
            - Adding new actions
        - Exceptions
    - **`getpass`** — Portable password input
    - **`fileinput`** — Iterate over lines from multiple input streams
    - **`curses`** — Terminal handling for character-cell displays
        - Functions
        - Window Objects
        - Constants
    - **`curses.textpad`** — Text input widget for curses programs
        - Textbox objects
    - **`curses.ascii`** — Utilities for ASCII characters
    - **`curses.panel`** — A panel stack extension for curses
        - Functions
        - Panel Objects
    - **`cmd`** — Support for line-oriented command interpreters
        - Cmd Objects
        - Cmd Example
- Concurrent Execution
    - **`threading`** — Thread-based parallelism
        - Introduction
        - GIL and performance considerations
        - Reference
            - Thread-local data

        - Thread objects
        - Lock objects
        - RLock objects
        - Condition objects
        - Semaphore objects
        - `Semaphore` example
        - Event objects
        - Timer objects
        - Barrier objects
    - Using locks, conditions, and semaphores in the `with` statement
- `multiprocessing` — Process-based parallelism
    - Introduction
        - The `Process` class
        - Contexts and start methods
        - Exchanging objects between processes
        - Synchronization between processes
        - Sharing state between processes
        - Using a pool of workers
    - Reference
        - Global start method
        - `Process` and exceptions
        - Pipes and Queues
        - Miscellaneous
        - Connection Objects
        - Synchronization primitives
        - Shared `ctypes` Objects
            - The `multiprocessing.sharedctypes` module
        - Managers
            - Customized managers
            - Using a remote manager
        - Proxy Objects
            - Cleanup
        - Process Pools
        - Listeners and Clients

            - Address Formats
        - Authentication keys
        - Logging
        - The **`multiprocessing.dummy`** module
    - Programming guidelines
        - All start methods
        - The *spawn* and *forkserver* start methods
    - Examples
- **`multiprocessing.shared_memory`** — Shared memory for direct access across processes
- The **`concurrent`** package
- **`concurrent.futures`** — Launching parallel tasks
    - Executor Objects
    - ThreadPoolExecutor
        - ThreadPoolExecutor Example
    - InterpreterPoolExecutor
    - ProcessPoolExecutor
        - ProcessPoolExecutor Example
    - Future Objects
    - Module Functions
    - Exception classes
- **`concurrent.interpreters`** — Multiple interpreters in the same process
    - Key details
    - Introduction
        - Multiple Interpreters and Isolation
        - Running in an Interpreter
        - Concurrency and Parallelism
        - Communication Between Interpreters
        - “Sharing” Objects
    - Reference
        - Interpreter objects
        - Exceptions
        - Communicating Between Interpreters
    - Basic usage
- **`subprocess`** — Subprocess management

        - Using the `subprocess` Module
            - Frequently Used Arguments
            - Popen Constructor
            - Exceptions
        - Security Considerations
        - Popen Objects
        - Windows Popen Helpers
            - Windows Constants
        - Older high-level API
        - Replacing Older Functions with the `subprocess` Module
            - Replacing **/bin/sh** shell command substitution
            - Replacing shell pipeline
            - Replacing `os.system()`
            - Replacing the `os.spawn` family
            - Replacing `os.popen()`
        - Legacy Shell Invocation Functions
        - Notes
            - Timeout Behavior
            - Converting an argument sequence to a string on Windows
            - Disable use of `posix_spawn()`
    - `sched` — Event scheduler
        - Scheduler Objects
    - `queue` — A synchronized queue class
        - Queue Objects
            - Waiting for task completion
            - Terminating queues
        - SimpleQueue Objects
    - `contextvars` — Context Variables
        - Context Variables
        - Manual Context Management
        - asyncio support
    - `_thread` — Low-level threading API
- Networking and Interprocess Communication

- `asyncio` — Asynchronous I/O
    - Runners
        - Running an asyncio Program
        - Runner context manager
        - Handling Keyboard Interruption
    - Coroutines and Tasks
        - Coroutines
        - Awaitables
        - Creating Tasks
        - Task Cancellation
        - Task Groups
            - Terminating a Task Group
        - Sleeping
        - Running Tasks Concurrently
        - Eager Task Factory
        - Shielding From Cancellation
        - Timeouts
        - Waiting Primitives
        - Running in Threads
        - Scheduling From Other Threads
        - Introspection
        - Task Object
    - Streams
        - StreamReader
        - StreamWriter
        - Examples
            - TCP echo client using streams
            - TCP echo server using streams
            - Get HTTP headers
            - Register an open socket to wait for data using streams
    - Synchronization Primitives
        - Lock
        - Event
        - Condition
        - Semaphore

    - BoundedSemaphore
    - Barrier
  - Subprocesses
    - Creating Subprocesses
    - Constants
    - Interacting with Subprocesses
      - Subprocess and Threads
      - Examples
  - Queues
    - Queue
    - Priority Queue
    - LIFO Queue
    - Exceptions
    - Examples
  - Exceptions
  - Call Graph Introspection
    - Low level utility functions
  - Event Loop
    - Event Loop Methods
      - Running and stopping the loop
      - Scheduling callbacks
      - Scheduling delayed callbacks
      - Creating Futures and Tasks
      - Opening network connections
      - Creating network servers
      - Transferring files
      - TLS Upgrade
      - Watching file descriptors
      - Working with socket objects directly
      - DNS
      - Working with pipes
      - Unix signals
      - Executing code in thread or process pools
      - Error Handling API
      - Enabling debug mode

        - Running Subprocesses
    - Callback Handles
    - Server Objects
    - Event Loop Implementations
    - Examples
        - Hello World with call_soon()
        - Display the current date with call_later()
        - Watch a file descriptor for read events
        - Set signal handlers for SIGINT and SIGTERM
- Futures
    - Future Functions
    - Future Object
- Transports and Protocols
    - Transports
        - Transports Hierarchy
        - Base Transport
        - Read-only Transports
        - Write-only Transports
        - Datagram Transports
        - Subprocess Transports
    - Protocols
        - Base Protocols
        - Base Protocol
        - Streaming Protocols
        - Buffered Streaming Protocols
        - Datagram Protocols
        - Subprocess Protocols
    - Examples
        - TCP Echo Server
        - TCP Echo Client
        - UDP Echo Server
        - UDP Echo Client
        - Connecting Existing Sockets

                - loop.subprocess_exec() and SubprocessProtocol
            - Policies
                - Getting and Setting the Policy
                - Policy Objects
                - Custom Policies
            - Platform Support
                - All Platforms
                - Windows
                    - Subprocess Support on Windows
                - macOS
            - Extending
                - Writing a Custom Event Loop
                - Future and Task private constructors
                - Task lifetime support
            - High-level API Index
                - Tasks
                - Queues
                - Subprocesses
                - Streams
                - Synchronization
                - Exceptions
            - Low-level API Index
                - Obtaining the Event Loop
                - Event Loop Methods
                - Transports
                - Protocols
                - Event Loop Policies
            - Developing with asyncio
                - Debug Mode
                - Concurrency and Multithreading
                - Running Blocking Code
                - Logging
                - Detect never-awaited coroutines
                - Detect never-retrieved exceptions
        - **`socket`** — Low-level networking interface

    - Socket families
    - Module contents
        - Exceptions
        - Constants
        - Functions
            - Creating sockets
            - Other functions
    - Socket Objects
    - Notes on socket timeouts
        - Timeouts and the `connect` method
        - Timeouts and the `accept` method
    - Example
- `ssl` — TLS/SSL wrapper for socket objects
    - Functions, Constants, and Exceptions
        - Socket creation
        - Context creation
        - Exceptions
        - Random generation
        - Certificate handling
        - Constants
    - SSL Sockets
    - SSL Contexts
    - Certificates
        - Certificate chains
        - CA certificates
        - Combined key and certificate
        - Self-signed certificates
    - Examples
        - Testing for SSL support
        - Client-side operation
        - Server-side operation
    - Notes on non-blocking sockets
    - Memory BIO Support
    - SSL session
    - Security considerations
        - Best defaults

                - Manual settings
                    - Verifying certificates
                    - Protocol versions
                    - Cipher selection
                - Multi-processing
            - TLS 1.3
        - `select` — Waiting for I/O completion
            - `/dev/poll` Polling Objects
            - Edge and Level Trigger Polling (epoll) Objects
            - Polling Objects
            - Kqueue Objects
            - Kevent Objects
        - `selectors` — High-level I/O multiplexing
            - Introduction
            - Classes
            - Examples
        - `signal` — Set handlers for asynchronous events
            - General rules
                - Execution of Python signal handlers
                - Signals and threads
            - Module contents
            - Examples
            - Note on SIGPIPE
            - Note on Signal Handlers and Exceptions
        - `mmap` — Memory-mapped file support
            - MADV_* Constants
            - MAP_* Constants
    - Internet Data Handling
        - `email` — An email and MIME handling package
            - `email.message`: Representing an email message
            - `email.parser`: Parsing email messages
                - FeedParser API
                - Parser API
                - Additional notes

    - `email.generator`: Generating MIME documents
    - `email.policy`: Policy Objects
    - `email.errors`: Exception and Defect classes
    - `email.headerregistry`: Custom Header Objects
    - `email.contentmanager`: Managing MIME Content
        - Content Manager Instances
    - `email`: Examples
    - `email.message.Message`: Representing an email message using the `compat32` API
    - `email.mime`: Creating email and MIME objects from scratch
    - `email.header`: Internationalized headers
    - `email.charset`: Representing character sets
    - `email.encoders`: Encoders
    - `email.utils`: Miscellaneous utilities
    - `email.iterators`: Iterators
- `json` — JSON encoder and decoder
    - Basic Usage
    - Encoders and Decoders
    - Exceptions
    - Standard Compliance and Interoperability
        - Character Encodings
        - Infinite and NaN Number Values
        - Repeated Names Within an Object
        - Top-level Non-Object, Non-Array Values
        - Implementation Limitations
    - Command-line interface
        - Command-line options
- `mailbox` — Manipulate mailboxes in various formats
    - `Mailbox` objects
        - `Maildir` objects
        - `mbox` objects
        - `MH` objects
        - `Babyl` objects
        - `MMDF` objects

        - `Message` objects
            - `MaildirMessage` objects
            - `mboxMessage` objects
            - `MHMessage` objects
            - `BabylMessage` objects
            - `MMDFMessage` objects
        - Exceptions
        - Examples
    - `mimetypes` — Map filenames to MIME types
        - MimeTypes objects
        - Command-line usage
        - Command-line example
    - `base64` — Base16, Base32, Base64, Base85 Data Encodings
        - RFC 4648 Encodings
        - Base85 Encodings
        - Legacy Interface
        - Security Considerations
    - `binascii` — Convert between binary and ASCII
    - `quopri` — Encode and decode MIME quoted-printable data
- Structured Markup Processing Tools
    - `html` — HyperText Markup Language support
    - `html.parser` — Simple HTML and XHTML parser
        - Example HTML Parser Application
        - `HTMLParser` Methods
        - Examples
    - `html.entities` — Definitions of HTML general entities
    - XML Processing Modules
        - XML security
    - `xml.etree.ElementTree` — The ElementTree XML API
        - Tutorial
            - XML tree and elements
            - Parsing XML
            - Pull API for non-blocking parsing

        - Finding interesting elements
        - Modifying an XML File
        - Building XML documents
        - Parsing XML with Namespaces
    - XPath support
        - Example
        - Supported XPath syntax
    - Reference
        - Functions
    - XInclude support
        - Example
    - Reference
        - Functions
        - Element Objects
        - ElementTree Objects
        - QName Objects
        - TreeBuilder Objects
        - XMLParser Objects
        - XMLPullParser Objects
        - Exceptions
- `xml.dom` — The Document Object Model API
    - Module Contents
    - Objects in the DOM
        - DOMImplementation Objects
        - Node Objects
        - NodeList Objects
        - DocumentType Objects
        - Document Objects
        - Element Objects
        - Attr Objects
        - NamedNodeMap Objects
        - Comment Objects
        - Text and CDATASection Objects
        - ProcessingInstruction Objects
        - Exceptions
    - Conformance

      - Type Mapping
      - Accessor Methods
    - `xml.dom.minidom` — Minimal DOM implementation
      - DOM Objects
      - DOM Example
      - minidom and the DOM standard
    - `xml.dom.pulldom` — Support for building partial DOM trees
      - DOMEventStream Objects
    - `xml.sax` — Support for SAX2 parsers
      - SAXException Objects
    - `xml.sax.handler` — Base classes for SAX handlers
      - ContentHandler Objects
      - DTDHandler Objects
      - EntityResolver Objects
      - ErrorHandler Objects
      - LexicalHandler Objects
    - `xml.sax.saxutils` — SAX Utilities
    - `xml.sax.xmlreader` — Interface for XML parsers
      - XMLReader Objects
      - IncrementalParser Objects
      - Locator Objects
      - InputSource Objects
      - The `Attributes` Interface
      - The `AttributesNS` Interface
    - `xml.parsers.expat` — Fast XML parsing using Expat
      - XMLParser Objects
      - ExpatError Exceptions
      - Example
      - Content Model Descriptions
      - Expat error constants
  - Internet Protocols and Support
    - `webbrowser` — Convenient web-browser controller
      - Command-line interface
      - Browser controller objects

- `wsgiref` — WSGI Utilities and Reference Implementation
    - `wsgiref.util` – WSGI environment utilities
    - `wsgiref.headers` – WSGI response header tools
    - `wsgiref.simple_server` – a simple WSGI HTTP server
    - `wsgiref.validate` — WSGI conformance checker
    - `wsgiref.handlers` – server/gateway base classes
    - `wsgiref.types` – WSGI types for static type checking
    - Examples
- `urllib` — URL handling modules
- `urllib.request` — Extensible library for opening URLs
    - Request Objects
    - OpenerDirector Objects
    - BaseHandler Objects
    - HTTPRedirectHandler Objects
    - HTTPCookieProcessor Objects
    - ProxyHandler Objects
    - HTTPPasswordMgr Objects
    - HTTPPasswordMgrWithPriorAuth Objects
    - AbstractBasicAuthHandler Objects
    - HTTPBasicAuthHandler Objects
    - ProxyBasicAuthHandler Objects
    - AbstractDigestAuthHandler Objects
    - HTTPDigestAuthHandler Objects
    - ProxyDigestAuthHandler Objects
    - HTTPHandler Objects
    - HTTPSHandler Objects
    - FileHandler Objects
    - DataHandler Objects
    - FTPHandler Objects
    - CacheFTPHandler Objects

    - UnknownHandler Objects
    - HTTPErrorProcessor Objects
    - Examples
    - Legacy interface
    - `urllib.request` Restrictions
- `urllib.response` — Response classes used by urllib
- `urllib.parse` — Parse URLs into components
    - URL Parsing
    - URL parsing security
    - Parsing ASCII Encoded Bytes
    - Structured Parse Results
    - URL Quoting
- `urllib.error` — Exception classes raised by urllib.request
- `urllib.robotparser` — Parser for robots.txt
- `http` — HTTP modules
    - HTTP status codes
    - HTTP status category
    - HTTP methods
- `http.client` — HTTP protocol client
    - HTTPConnection Objects
    - HTTPResponse Objects
    - Examples
    - HTTPMessage Objects
- `ftplib` — FTP protocol client
    - Reference
        - FTP objects
        - FTP_TLS objects
        - Module variables
- `poplib` — POP3 protocol client
    - POP3 Objects
    - POP3 Example
- `imaplib` — IMAP4 protocol client
    - IMAP4 Objects
    - IMAP4 Example

- `smtplib` — SMTP protocol client
  - SMTP Objects
  - SMTP Example
- `uuid` — UUID objects according to **RFC 9562**
  - Command-Line Usage
  - Example
  - Command-Line Example
- `socketserver` — A framework for network servers
  - Server Creation Notes
  - Server Objects
  - Request Handler Objects
  - Examples
    - `socketserver.TCPServer` Example
    - `socketserver.UDPServer` Example
    - Asynchronous Mixins
- `http.server` — HTTP servers
  - Command-line interface
  - Security considerations
- `http.cookies` — HTTP state management
  - Cookie Objects
  - Morsel Objects
  - Example
- `http.cookiejar` — Cookie handling for HTTP clients
  - CookieJar and FileCookieJar Objects
  - FileCookieJar subclasses and co-operation with web browsers
  - CookiePolicy Objects
  - DefaultCookiePolicy Objects
  - Cookie Objects
  - Examples
- `xmlrpc` — XMLRPC server and client modules
- `xmlrpc.client` — XML-RPC client access
  - ServerProxy Objects
  - DateTime Objects
  - Binary Objects

        - Fault Objects
        - ProtocolError Objects
        - MultiCall Objects
        - Convenience Functions
        - Example of Client Usage
        - Example of Client and Server Usage
    - `xmlrpc.server` — Basic XML-RPC servers
        - SimpleXMLRPCServer Objects
            - SimpleXMLRPCServer Example
        - CGIXMLRPCRequestHandler
        - Documenting XMLRPC server
        - DocXMLRPCServer Objects
        - DocCGIXMLRPCRequestHandler
    - `ipaddress` — IPv4/IPv6 manipulation library
        - Convenience factory functions
        - IP Addresses
            - Address objects
            - Conversion to Strings and Integers
            - Operators
                - Comparison operators
                - Arithmetic operators
        - IP Network definitions
            - Prefix, net mask and host mask
            - Network objects
            - Operators
                - Logical operators
                - Iteration
                - Networks as containers of addresses
        - Interface objects
            - Operators
                - Logical operators
        - Other Module Level Functions
        - Custom Exceptions
- Multimedia Services
    - `wave` — Read and write WAV files
        - Wave_read Objects

        - Wave_write Objects
    - `colorsys` — Conversions between color systems
- Internationalization
    - `gettext` — Multilingual internationalization services
        - GNU **gettext** API
        - Class-based API
            - The `NullTranslations` class
            - The `GNUTranslations` class
            - Solaris message catalog support
            - The Catalog constructor
        - Internationalizing your programs and modules
            - Localizing your module
            - Localizing your application
            - Changing languages on the fly
            - Deferred translations
        - Acknowledgements
    - `locale` — Internationalization services
        - Background, details, hints, tips and caveats
        - Locale names
        - For extension writers and programs that embed Python
        - Access to message catalogs
- Graphical user interfaces with Tk
    - `tkinter` — Python interface to Tcl/Tk
        - Architecture
        - Tkinter Modules
        - Tkinter Life Preserver
            - A Hello World Program
            - Important Tk Concepts
            - Understanding How Tkinter Wraps Tcl/Tk
            - How do I…? What option does…?
            - Navigating the Tcl/Tk Reference Manual
        - Threading model
        - Handy Reference

        - Setting Options
        - The Packer
        - Packer Options
        - Coupling Widget Variables
        - The Window Manager
        - Tk Option Data Types
        - Bindings and Events
        - The index Parameter
        - Images
    - File Handlers
- **`tkinter.colorchooser`** — Color choosing dialog
- **`tkinter.font`** — Tkinter font wrapper
- Tkinter Dialogs
    - **`tkinter.simpledialog`** — Standard Tkinter input dialogs
    - **`tkinter.filedialog`** — File selection dialogs
        - Native Load/Save Dialogs
    - **`tkinter.commondialog`** — Dialog window templates
- **`tkinter.messagebox`** — Tkinter message prompts
- **`tkinter.scrolledtext`** — Scrolled Text Widget
- **`tkinter.dnd`** — Drag and drop support
- **`tkinter.ttk`** — Tk themed widgets
    - Using Ttk
    - Ttk Widgets
    - Widget
        - Standard Options
        - Scrollable Widget Options
        - Label Options
        - Compatibility Options
        - Widget States
        - ttk.Widget
    - Combobox
        - Options
        - Virtual events
        - ttk.Combobox

    - Spinbox
        - Options
        - Virtual events
        - ttk.Spinbox
    - Notebook
        - Options
        - Tab Options
        - Tab Identifiers
        - Virtual Events
        - ttk.Notebook
    - Progressbar
        - Options
        - ttk.Progressbar
    - Separator
        - Options
    - Sizegrip
        - Platform-specific notes
        - Bugs
    - Treeview
        - Options
        - Item Options
        - Tag Options
        - Column Identifiers
        - Virtual Events
        - ttk.Treeview
    - Ttk Styling
        - Layouts
- IDLE — Python editor and shell
    - Menus
        - File menu (Shell and Editor)
        - Edit menu (Shell and Editor)
        - Format menu (Editor window only)
        - Run menu (Editor window only)
        - Shell menu (Shell window only)
        - Debug menu (Shell window only)
        - Options menu (Shell and Editor)

        - Window menu (Shell and Editor)
        - Help menu (Shell and Editor)
        - Context menus
    - Editing and Navigation
        - Editor windows
        - Key bindings
        - Automatic indentation
        - Search and Replace
        - Completions
        - Calltips
        - Code Context
        - Shell window
        - Text colors
    - Startup and Code Execution
        - Command-line usage
        - Startup failure
        - Running user code
        - User output in Shell
        - Developing tkinter applications
        - Running without a subprocess
    - Help and Preferences
        - Help sources
        - Setting preferences
        - IDLE on macOS
        - Extensions
    - idlelib — implementation of IDLE application
- `turtle` — Turtle graphics
    - Introduction
    - Get started
    - Tutorial
        - Starting a turtle environment
        - Basic drawing
            - Pen control
            - The turtle’s position
        - Making algorithmic patterns
    - How to…

    - Get started as quickly as possible
    - Automatically begin and end filling
    - Use the **`turtle`** module namespace
    - Use turtle graphics in a script
    - Use object-oriented turtle graphics
- Turtle graphics reference
    - Turtle methods
    - Methods of TurtleScreen/Screen
- Methods of RawTurtle/Turtle and corresponding functions
    - Turtle motion
    - Tell Turtle’s state
    - Settings for measurement
    - Pen control
        - Drawing state
        - Color control
        - Filling
        - More drawing control
    - Turtle state
        - Visibility
        - Appearance
    - Using events
    - Special Turtle methods
    - Compound shapes
- Methods of TurtleScreen/Screen and corresponding functions
    - Window control
    - Animation control
    - Using screen events
    - Input methods
    - Settings and special methods
    - Methods specific to Screen, not inherited from TurtleScreen
- Public classes
- Explanation
- Help and configuration

            - How to use help
            - Translation of docstrings into different languages
            - How to configure Screen and Turtles
        - `turtledemo` — Demo scripts
        - Changes since Python 2.6
        - Changes since Python 3.0
- Development Tools
    - `typing` — Support for type hints
        - Specification for the Python Type System
        - Type aliases
        - NewType
        - Annotating callable objects
        - Generics
        - Annotating tuples
        - The type of class objects
        - Annotating generators and coroutines
        - User-defined generic types
        - The `Any` type
        - Nominal vs structural subtyping
        - Module contents
            - Special typing primitives
                - Special types
                - Special forms
                - Building generic types and type aliases
                - Other special directives
            - Protocols
            - ABCs and Protocols for working with I/O
            - Functions and decorators
            - Introspection helpers
            - Constant
            - Deprecated aliases
                - Aliases to built-in types
                - Aliases to types in `collections`
                - Aliases to other concrete types

            - Aliases to container ABCs in `collections.abc`
            - Aliases to asynchronous ABCs in `collections.abc`
            - Aliases to other ABCs in `collections.abc`
            - Aliases to `contextlib` ABCs
    - Deprecation Timeline of Major Features
- `pydoc` — Documentation generator and online help system
- Python Development Mode
    - Effects of the Python Development Mode
    - ResourceWarning Example
    - Bad file descriptor error example
- `doctest` — Test interactive Python examples
    - Simple Usage: Checking Examples in Docstrings
    - Simple Usage: Checking Examples in a Text File
    - Command-line Usage
    - How It Works
        - Which Docstrings Are Examined?
        - How are Docstring Examples Recognized?
        - What’s the Execution Context?
        - What About Exceptions?
        - Option Flags
        - Directives
        - Warnings
    - Basic API
    - Unittest API
    - Advanced API
        - DocTest Objects
        - Example Objects
        - DocTestFinder objects
        - DocTestParser objects

    - TestResults objects
    - DocTestRunner objects
    - OutputChecker objects
  - Debugging
  - Soapbox
- `unittest` — Unit testing framework
  - Basic example
  - Command-Line Interface
    - Command-line options
  - Test Discovery
  - Organizing test code
  - Re-using old test code
  - Skipping tests and expected failures
  - Distinguishing test iterations using subtests
  - Classes and functions
    - Test cases
    - Grouping tests
    - Loading and running tests
      - load_tests Protocol
  - Class and Module Fixtures
    - setUpClass and tearDownClass
    - setUpModule and tearDownModule
  - Signal Handling
- `unittest.mock` — mock object library
  - Quick Guide
  - The Mock Class
    - Calling
    - Deleting Attributes
    - Mock names and the name attribute
    - Attaching Mocks as Attributes
  - The patchers
    - patch
    - patch.object
    - patch.dict
    - patch.multiple
    - patch methods: start and stop

        - patch builtins
        - TEST_PREFIX
        - Nesting Patch Decorators
        - Where to patch
        - Patching Descriptors and Proxy Objects
    - MagicMock and magic method support
        - Mocking Magic Methods
        - Magic Mock
    - Helpers
        - sentinel
        - DEFAULT
        - call
        - create_autospec
        - ANY
        - FILTER_DIR
        - mock_open
        - Autospeccing
        - Sealing mocks
    - Order of precedence of **`side_effect`**, **`return_value`** and *wraps*
- **`unittest.mock`** — getting started
    - Using Mock
        - Mock Patching Methods
        - Mock for Method Calls on an Object
        - Mocking Classes
        - Naming your mocks
        - Tracking all Calls
        - Setting Return Values and Attributes
        - Raising exceptions with mocks
        - Side effect functions and iterables
        - Mocking asynchronous iterators
        - Mocking asynchronous context manager
        - Creating a Mock from an Existing Object
        - Using side_effect to return per file content
    - Patch Decorators

        - Further Examples
            - Mocking chained calls
            - Partial mocking
            - Mocking a Generator Method
            - Applying the same patch to every test method
            - Mocking Unbound Methods
            - Checking multiple calls with mock
            - Coping with mutable arguments
            - Nesting Patches
            - Mocking a dictionary with MagicMock
            - Mock subclasses and their attributes
            - Mocking imports with patch.dict
            - Tracking order of calls and less verbose call assertions
            - More complex argument matching
    - **`test`** — Regression tests package for Python
        - Writing Unit Tests for the **`test`** package
        - Running tests using the command-line interface
    - **`test.support`** — Utilities for the Python test suite
    - **`test.support.socket_helper`** — Utilities for socket tests
    - **`test.support.script_helper`** — Utilities for the Python execution tests
    - **`test.support.bytecode_helper`** — Support tools for testing correct bytecode generation
    - **`test.support.threading_helper`** — Utilities for threading tests
    - **`test.support.os_helper`** — Utilities for os tests
    - **`test.support.import_helper`** — Utilities for import tests
    - **`test.support.warnings_helper`** — Utilities for warnings tests
- Debugging and Profiling
    - Audit events table

- `bdb` — Debugger framework
- `faulthandler` — Dump the Python traceback
    - Dumping the traceback
    - Dumping the C stack
        - C Stack Compatibility
    - Fault handler state
    - Dumping the tracebacks after a timeout
    - Dumping the traceback on a user signal
    - Issue with file descriptors
    - Example
- `pdb` — The Python Debugger
    - Command-line interface
    - Debugger commands
- The Python Profilers
    - Introduction to the profilers
    - Instant User’s Manual
    - `profile` and `cProfile` Module Reference
    - The `Stats` Class
    - What Is Deterministic Profiling?
    - Limitations
    - Calibration
    - Using a custom timer
- `timeit` — Measure execution time of small code snippets
    - Basic Examples
    - Python Interface
    - Command-Line Interface
    - Examples
- `trace` — Trace or track Python statement execution
    - Command-Line Usage
        - Main options
        - Modifiers
        - Filters
    - Programmatic Interface
- `tracemalloc` — Trace memory allocations

            - Examples
                - Display the top 10
                - Compute differences
                - Get the traceback of a memory block
                - Pretty top
                    - Record the current and peak size of all traced memory blocks
            - API
                - Functions
                - DomainFilter
                - Filter
                - Frame
                - Snapshot
                - Statistic
                - StatisticDiff
                - Trace
                - Traceback
    - Software Packaging and Distribution
        - **`ensurepip`** — Bootstrapping the **`pip`** installer
            - Command-line interface
            - Module API
        - **`venv`** — Creation of virtual environments
            - Creating virtual environments
            - How venvs work
            - API
            - An example of extending **`EnvBuilder`**
        - **`zipapp`** — Manage executable Python zip archives
            - Basic Example
            - Command-Line Interface
            - Python API
            - Examples
            - Specifying the Interpreter
            - Creating Standalone Applications with zipapp
                - Caveats
            - The Python Zip Application Archive Format

- Python Runtime Services
  - `sys` — System-specific parameters and functions
  - `sys.monitoring` — Execution event monitoring
    - Tool identifiers
      - Registering and using tools
    - Events
      - Local events
      - Deprecated event
      - Ancillary events
      - Other events
      - The STOP_ITERATION event
    - Turning events on and off
      - Setting events globally
      - Per code object events
      - Disabling events
    - Registering callback functions
      - Callback function arguments
  - `sysconfig` — Provide access to Python’s configuration information
    - Configuration variables
    - Installation paths
    - User scheme
      - `posix_user`
      - `nt_user`
      - `osx_framework_user`
    - Home scheme
      - `posix_home`
    - Prefix scheme
      - `posix_prefix`
      - `nt`
    - Installation path functions
    - Other functions
    - Command-line usage
  - `builtins` — Built-in objects
  - `__main__` — Top-level code environment
    - `__name__ == '__main__'`

        - What is the “top-level code environment”?
        - Idiomatic Usage
        - Packaging Considerations
    - `__main__.py` in Python Packages
        - Idiomatic Usage
    - `import __main__`
- `warnings` — Warning control
    - Warning Categories
    - The Warnings Filter
        - Repeated Warning Suppression Criteria
        - Describing Warning Filters
        - Default Warning Filter
        - Overriding the default filter
    - Temporarily Suppressing Warnings
    - Testing Warnings
    - Updating Code For New Versions of Dependencies
    - Available Functions
    - Available Context Managers
    - Concurrent safety of Context Managers
- `dataclasses` — Data Classes
    - Module contents
    - Post-init processing
    - Class variables
    - Init-only variables
    - Frozen instances
    - Inheritance
    - Re-ordering of keyword-only parameters in `__init__()`
    - Default factory functions
    - Mutable default values
    - Descriptor-typed fields
- `contextlib` — Utilities for `with`-statement contexts
    - Utilities
    - Examples and Recipes

        - Supporting a variable number of context managers
        - Catching exceptions from `__enter__` methods
        - Cleaning up in an `__enter__` implementation
        - Replacing any use of `try-finally` and flag variables
        - Using a context manager as a function decorator
    - Single use, reusable and reentrant context managers
        - Reentrant context managers
        - Reusable context managers
- `abc` — Abstract Base Classes
- `atexit` — Exit handlers
    - `atexit` Example
- `traceback` — Print or retrieve a stack traceback
    - Module-Level Functions
    - `TracebackException` Objects
    - `StackSummary` Objects
    - `FrameSummary` Objects
    - Examples of Using the Module-Level Functions
    - Examples of Using `TracebackException`
- `__future__` — Future statement definitions
    - Module Contents
- `gc` — Garbage Collector interface
- `inspect` — Inspect live objects
    - Types and members
    - Retrieving source code
    - Introspecting callables with the Signature object
    - Classes and functions
    - The interpreter stack
    - Fetching attributes statically

        - Current State of Generators, Coroutines, and Asynchronous Generators
        - Code Objects Bit Flags
        - Buffer flags
        - Command-line interface
    - `annotationlib` — Functionality for introspecting annotations
        - Annotation semantics
        - Classes
        - Functions
        - Recipes
            - Using annotations in a metaclass
        - Limitations of the `STRING` format
        - Limitations of the `FORWARDREF` format
        - Security implications of introspecting annotations
    - `site` — Site-specific configuration hook
        - `sitecustomize`
        - `usercustomize`
        - Readline configuration
        - Module contents
        - Command-line interface
- Custom Python Interpreters
    - `code` — Interpreter base classes
        - Interactive Interpreter Objects
        - Interactive Console Objects
    - `codeop` — Compile Python code
- Importing Modules
    - `zipimport` — Import modules from Zip archives
        - zipimporter Objects
        - Examples
    - `pkgutil` — Package extension utility
    - `modulefinder` — Find modules used by a script
        - Example usage of `ModuleFinder`
    - `runpy` — Locating and executing Python modules
    - `importlib` — The implementation of `import`

    - Introduction
    - Functions
    - `importlib.abc` – Abstract base classes related to import
    - `importlib.machinery` – Importers and path hooks
    - `importlib.util` – Utility code for importers
    - Examples
        - Importing programmatically
        - Checking if a module can be imported
        - Importing a source file directly
        - Implementing lazy imports
        - Setting up an importer
        - Approximating `importlib.import_module()`
- `importlib.resources` – Package resource reading, opening and access
    - Functional API
- `importlib.resources.abc` – Abstract base classes for resources
- `importlib.metadata` – Accessing package metadata
    - Overview
    - Functional API
        - Entry points
        - Distribution metadata
        - Distribution versions
        - Distribution files
        - Distribution requirements
        - Mapping import to distribution packages
    - Distributions
    - Distribution Discovery
    - Implementing Custom Providers
        - Example
- The initialization of the `sys.path` module search path
    - Virtual Environments
    - _pth files

    - Embedded Python
- Python Language Services
  - `ast` — Abstract syntax trees
    - Abstract grammar
    - Node classes
      - Root nodes
      - Literals
      - Variables
      - Expressions
        - Subscripting
        - Comprehensions
      - Statements
        - Imports
      - Control flow
      - Pattern matching
      - Type annotations
      - Type parameters
      - Function and class definitions
      - Async and await
    - `ast` helpers
    - Compiler flags
    - Command-line usage
  - `symtable` — Access to the compiler’s symbol tables
    - Generating Symbol Tables
    - Examining Symbol Tables
    - Command-Line Usage
  - `token` — Constants used with Python parse trees
  - `keyword` — Testing for Python keywords
  - `tokenize` — Tokenizer for Python source
    - Tokenizing Input
    - Command-Line Usage
    - Examples
  - `tabnanny` — Detection of ambiguous indentation
  - `pyclbr` — Python module browser support
    - Function Objects

            - Class Objects
        - `py_compile` — Compile Python source files
            - Command-Line Interface
        - `compileall` — Byte-compile Python libraries
            - Command-line use
            - Public functions
        - `dis` — Disassembler for Python bytecode
            - Command-line interface
            - Bytecode analysis
            - Analysis functions
            - Python Bytecode Instructions
            - Opcode collections
        - `pickletools` — Tools for pickle developers
            - Command-line usage
                - Command-line options
            - Programmatic interface
    - MS Windows Specific Services
        - `msvcrt` — Useful routines from the MS VC++ runtime
            - File Operations
            - Console I/O
            - Other Functions
        - `winreg` — Windows registry access
            - Functions
            - Constants
                - HKEY_* Constants
                - Access Rights
                    - 64-bit Specific
                - Value Types
            - Registry Handle Objects
        - `winsound` — Sound-playing interface for Windows
    - Unix-specific services
        - `shlex` — Simple lexical analysis
            - shlex Objects
            - Parsing Rules
            - Improved Compatibility with Shells

        - `posix` — The most common POSIX system calls
            - Large File Support
            - Notable Module Contents
        - `pwd` — The password database
        - `grp` — The group database
        - `termios` — POSIX style tty control
            - Example
        - `tty` — Terminal control functions
        - `pty` — Pseudo-terminal utilities
            - Example
        - `fcntl` — The `fcntl` and `ioctl` system calls
        - `resource` — Resource usage information
            - Resource Limits
            - Resource Usage
        - `syslog` — Unix syslog library routines
            - Examples
                - Simple example
    - Modules command-line interface (CLI)
    - Superseded Modules
        - `getopt` — C-style parser for command line options
    - Removed Modules
    - Security Considerations
- Extending and Embedding the Python Interpreter
    - Recommended third party tools
    - Creating extensions without third party tools
        - 1. Extending Python with C or C++
            - 1.1. A Simple Example
            - 1.2. Intermezzo: Errors and Exceptions
            - 1.3. Back to the Example
            - 1.4. The Module’s Method Table and Initialization Function
            - 1.5. Compilation and Linkage
            - 1.6. Calling Python Functions from C
            - 1.7. Extracting Parameters in Extension Functions

    - 1.8. Keyword Parameters for Extension Functions
    - 1.9. Building Arbitrary Values
    - 1.10. Reference Counts
        - 1.10.1. Reference Counting in Python
        - 1.10.2. Ownership Rules
        - 1.10.3. Thin Ice
        - 1.10.4. NULL Pointers
    - 1.11. Writing Extensions in C++
    - 1.12. Providing a C API for an Extension Module
- 2. Defining Extension Types: Tutorial
    - 2.1. The Basics
    - 2.2. Adding data and methods to the Basic example
    - 2.3. Providing finer control over data attributes
    - 2.4. Supporting cyclic garbage collection
    - 2.5. Subclassing other types
- 3. Defining Extension Types: Assorted Topics
    - 3.1. Finalization and De-allocation
    - 3.2. Object Presentation
    - 3.3. Attribute Management
        - 3.3.1. Generic Attribute Management
        - 3.3.2. Type-specific Attribute Management
    - 3.4. Object Comparison
    - 3.5. Abstract Protocol Support
    - 3.6. Weak Reference Support
    - 3.7. More Suggestions
- 4. Building C and C++ Extensions
    - 4.1. Building C and C++ Extensions with setuptools
- 5. Building C and C++ Extensions on Windows
    - 5.1. A Cookbook Approach
    - 5.2. Differences Between Unix and Windows

        - 5.3. Using DLLs in Practice
  - Embedding the CPython runtime in a larger application
    - 1. Embedding Python in Another Application
      - 1.1. Very High Level Embedding
      - 1.2. Beyond Very High Level Embedding: An overview
      - 1.3. Pure Embedding
      - 1.4. Extending Embedded Python
      - 1.5. Embedding Python in C++
      - 1.6. Compiling and Linking under Unix-like systems
- Python/C API Reference Manual
  - Introduction
    - Language version compatibility
    - Coding standards
    - Include Files
    - Useful macros
    - Objects, Types and Reference Counts
      - Reference Counts
        - Reference Count Details
      - Types
    - Exceptions
    - Embedding Python
    - Debugging Builds
    - Recommended third party tools
  - C API Stability
    - Unstable C API
    - Stable Application Binary Interface
      - Limited C API
      - Stable ABI
      - Limited API Scope and Performance
      - Limited API Caveats
    - Platform Considerations
    - Contents of Limited API
  - The Very High Level Layer

    - Available start symbols
    - Stack Effects
  - Reference Counting
  - Exception Handling
    - Printing and clearing
    - Raising exceptions
    - Issuing warnings
    - Querying the error indicator
    - Signal Handling
    - Exception Classes
    - Exception Objects
    - Unicode Exception Objects
    - Recursion Control
    - Exception and warning types
      - Exception types
      - OSError aliases
      - Warning types
    - Tracebacks
  - Defining extension modules
    - Multiple module instances
    - Initialization function
    - Multi-phase initialization
    - Legacy single-phase initialization
  - Utilities
    - Operating System Utilities
    - System Functions
    - Process Control
    - Importing Modules
    - Data marshalling support
    - Parsing arguments and building values
      - Parsing arguments
        - Strings and buffers
        - Numbers
        - Other objects
        - API Functions
      - Building values

        - String conversion and formatting
        - Character classification and conversion
        - PyHash API
        - Reflection
        - Codec registry and support functions
            - Codec lookup API
            - Registry API for Unicode encoding error handlers
            - Codec utility variables
        - PyTime C API
            - Types
            - Clock Functions
            - Raw Clock Functions
            - Conversion functions
        - Support for Perf Maps
    - Abstract Objects Layer
        - Object Protocol
        - Call Protocol
            - The *tp_call* Protocol
            - The Vectorcall Protocol
                - Recursion Control
                - Vectorcall Support API
            - Object Calling API
            - Call Support API
        - Number Protocol
        - Sequence Protocol
        - Mapping Protocol
        - Iterator Protocol
        - Buffer Protocol
            - Buffer structure
            - Buffer request types
                - request-independent fields
                - readonly, format
                - shape, strides, suboffsets
                - contiguity requests
                - compound requests

            - Complex arrays
                - NumPy-style: shape and strides
                - PIL-style: shape, strides and suboffsets
            - Buffer-related functions
    - Concrete Objects Layer
        - Fundamental Objects
            - Type Objects
                - Creating Heap-Allocated Types
            - The `None` Object
        - Numeric Objects
            - Integer Objects
                - Export API
                - PyLongWriter API
                - Deprecated API
            - Boolean Objects
            - Floating-Point Objects
                - Pack and Unpack functions
                    - Pack functions
                    - Unpack functions
            - Complex Number Objects
                - Complex Numbers as C Structures
                - Complex Numbers as Python Objects
        - Sequence Objects
            - Bytes Objects
            - Byte Array Objects
                - Type check macros
                - Direct API functions
                - Macros
            - Unicode Objects and Codecs
                - Unicode Objects
                    - Unicode Type
                    - Unicode Character Properties
                    - Creating and accessing Unicode strings
                    - Locale Encoding
                    - File System Encoding

            - wchar_t Support
        - Built-in Codecs
            - Generic Codecs
            - UTF-8 Codecs
            - UTF-32 Codecs
            - UTF-16 Codecs
            - UTF-7 Codecs
            - Unicode-Escape Codecs
            - Raw-Unicode-Escape Codecs
            - Latin-1 Codecs
            - ASCII Codecs
            - Character Map Codecs
            - MBCS codecs for Windows
        - Methods and Slot Functions
        - PyUnicodeWriter
        - Deprecated API
    - Tuple Objects
    - Struct Sequence Objects
    - List Objects
- Container Objects
    - Dictionary Objects
        - Dictionary View Objects
        - Ordered Dictionaries
    - Set Objects
        - Deprecated API
- Function Objects
    - Function Objects
    - Instance Method Objects
    - Method Objects
    - Cell Objects
    - Code Objects
    - Code Object Flags
    - Extra information
- Other Objects
    - File Objects
        - Deprecated API

        - Module Objects
        - Module definitions
            - Module slots
        - Creating extension modules dynamically
        - Support functions
            - Module lookup (single-phase initialization)
        - Iterator Objects
            - Range Objects
            - Builtin Iterator Types
            - Other Iterator Objects
        - Descriptor Objects
            - Built-in descriptors
        - Slice Objects
            - Ellipsis Object
        - MemoryView objects
        - Pickle buffer objects
        - Weak Reference Objects
        - Capsules
        - Frame Objects
            - Frame Locals Proxies
            - Legacy Local Variable APIs
            - Internal Frames
        - Generator Objects
            - Asynchronous Generator Objects
            - Deprecated API
        - Coroutine Objects
        - Context Variables Objects
        - Objects for Type Hinting
    - C API for extension modules
        - Curses C API
        - Internal data
        - DateTime Objects
        - Internal data
- Initialization, Finalization, and Threads
    - Before Python Initialization

        - Global configuration variables
        - Initializing and finalizing the interpreter
        - Process-wide parameters
        - Thread State and the Global Interpreter Lock
            - Detaching the thread state from extension code
            - Non-Python created threads
            - Supporting subinterpreters in non-Python threads
            - Cautions about fork()
            - Cautions regarding runtime finalization
            - High-level API
            - Low-level API
        - Sub-interpreter support
            - A Per-Interpreter GIL
            - Bugs and caveats
        - Asynchronous Notifications
        - Profiling and Tracing
        - Reference tracing
        - Advanced Debugger Support
        - Thread Local Storage Support
            - Thread Specific Storage (TSS) API
                - Dynamic Allocation
                - Methods
            - Thread Local Storage (TLS) API
        - Synchronization Primitives
            - Python Critical Section API
            - Legacy Locking APIs
        - Operating System Thread APIs
    - Python Initialization Configuration
        - PyInitConfig C API
            - Example
            - Create Config
            - Error Handling
            - Get Options
            - Set Options

        - Module
        - Initialize Python
      - Configuration Options
      - Runtime Python configuration API
      - PyConfig C API
        - Example
        - PyWideStringList
        - PyStatus
        - PyPreConfig
        - Preinitialize Python with PyPreConfig
        - PyConfig
        - Initialization with PyConfig
        - Isolated Configuration
        - Python Configuration
        - Python Path Configuration
      - Py_GetArgcArgv()
      - Delaying main module execution
    - Memory Management
      - Overview
      - Allocator Domains
      - Raw Memory Interface
      - Memory Interface
      - Object allocators
      - Default Memory Allocators
      - Customize Memory Allocators
      - Debug hooks on the Python memory allocators
      - The pymalloc allocator
        - Customize pymalloc Arena Allocator
      - The mimalloc allocator
      - tracemalloc C API
      - Examples
    - Object Implementation Support
      - Allocating Objects on the Heap
        - Deprecated aliases
      - Object Life Cycle
        - Life Events

            - Cyclic Isolate Destruction
            - Functions
        - Common Object Structures
            - Base object types and macros
            - Implementing functions and methods
            - Accessing attributes of extension types
                - Member flags
                - Member types
                - Defining Getters and Setters
        - Type Object Structures
            - Quick Reference
                - “tp slots”
                - sub-slots
                - slot typedefs
            - PyTypeObject Definition
            - PyObject Slots
            - PyVarObject Slots
            - PyTypeObject Slots
            - Static Types
            - Heap Types
            - Number Object Structures
            - Mapping Object Structures
            - Sequence Object Structures
            - Buffer Object Structures
            - Async Object Structures
            - Slot Type typedefs
            - Examples
        - Supporting Cyclic Garbage Collection
            - Controlling the Garbage Collector State
            - Querying Garbage Collector State
    - API and ABI Versioning
        - Build-time version constants
        - Run-time version
        - Bit-packing macros
    - Monitoring C API
    - Generating Execution Events

        - Managing the Monitoring State
- Installing Python Modules
    - Key terms
    - Basic usage
    - How do I …?
        - … install `pip` in versions of Python prior to Python 3.4?
        - … install packages just for the current user?
        - … install scientific Python packages?
        - … work with multiple versions of Python installed in parallel?
    - Common installation issues
        - Installing into the system Python on Linux
        - Pip not installed
        - Installing binary extensions
- Python HOWTOs
- Python Frequently Asked Questions
    - General Python FAQ
        - General Information
        - Python in the real world
    - Programming FAQ
        - General Questions
        - Core Language
        - Numbers and strings
        - Performance
        - Sequences (Tuples/Lists)
        - Objects
        - Modules
    - Design and History FAQ
        - Why does Python use indentation for grouping of statements?
        - Why am I getting strange results with simple arithmetic operations?
        - Why are floating-point calculations so inaccurate?
        - Why are Python strings immutable?

- Why must ‘self’ be used explicitly in method definitions and calls?
- Why can’t I use an assignment in an expression?
- Why does Python use methods for some functionality (e.g. list.index()) but functions for other (e.g. len(list))?
- Why is join() a string method instead of a list or tuple method?
- How fast are exceptions?
- Why isn’t there a switch or case statement in Python?
- Can’t you emulate threads in the interpreter instead of relying on an OS-specific thread implementation?
- Why can’t lambda expressions contain statements?
- Can Python be compiled to machine code, C or some other language?
- How does Python manage memory?
- Why doesn’t CPython use a more traditional garbage collection scheme?
- Why isn’t all memory freed when CPython exits?
- Why are there separate tuple and list data types?
- How are lists implemented in CPython?
- How are dictionaries implemented in CPython?
- Why must dictionary keys be immutable?
- Why doesn’t list.sort() return the sorted list?
- How do you specify and enforce an interface spec in Python?
- Why is there no goto?
- Why can’t raw strings (r-strings) end with a backslash?
- Why doesn’t Python have a “with” statement for attribute assignments?

    - Why don’t generators support the with statement?
    - Why are colons required for the if/while/def/class statements?
    - Why does Python allow commas at the end of lists and tuples?
  - Library and Extension FAQ
    - General Library Questions
    - Common tasks
    - Threads
    - Input and Output
    - Network/Internet Programming
    - Databases
    - Mathematics and Numerics
  - Extending/Embedding FAQ
    - Can I create my own functions in C?
    - Can I create my own functions in C++?
    - Writing C is hard; are there any alternatives?
    - How can I execute arbitrary Python statements from C?
    - How can I evaluate an arbitrary Python expression from C?
    - How do I extract C values from a Python object?
    - How do I use Py_BuildValue() to create a tuple of arbitrary length?
    - How do I call an object’s method from C?
    - How do I catch the output from PyErr_Print() (or anything that prints to stdout/stderr)?
    - How do I access a module written in Python from C?
    - How do I interface to C++ objects from Python?
    - I added a module using the Setup file and the make fails; why?
    - How do I debug an extension?
    - I want to compile a Python module on my Linux system, but some files are missing. Why?

        - How do I tell “incomplete input” from “invalid input”?
        - How do I find undefined g++ symbols __builtin_new or __pure_virtual?
        - Can I create an object class with some methods implemented in C and others in Python (e.g. through inheritance)?
    - Python on Windows FAQ
        - How do I run a Python program under Windows?
        - How do I make Python scripts executable?
        - Why does Python sometimes take so long to start?
        - How do I make an executable from a Python script?
        - Is a `*.pyd` file the same as a DLL?
        - How can I embed Python into a Windows application?
        - How do I keep editors from inserting tabs into my Python source?
        - How do I check for a keypress without blocking?
        - How do I solve the missing api-ms-win-crt-runtime-l1-1-0.dll error?
    - Graphic User Interface FAQ
        - General GUI Questions
        - What GUI toolkits exist for Python?
        - Tkinter questions
    - “Why is Python Installed on my Computer?” FAQ
        - What is Python?
        - Why is Python installed on my machine?
        - Can I delete Python?
- Deprecations
    - Pending removal in Python 3.15
    - Pending removal in Python 3.16
    - Pending removal in Python 3.17
    - Pending removal in Python 3.18
    - Pending removal in Python 3.19

    - Pending removal in future versions
    - C API deprecations
        - Pending removal in Python 3.15
        - Pending removal in Python 3.18
        - Pending removal in future versions
- Glossary
- About this documentation
    - Contributors to the Python documentation
- Dealing with Bugs
    - Documentation bugs
    - Using the Python issue tracker
    - Getting started contributing to Python yourself
- Copyright
- History and License
    - History of the software
    - Terms and conditions for accessing or otherwise using Python
        - PYTHON SOFTWARE FOUNDATION LICENSE VERSION 2
        - BEOPEN.COM LICENSE AGREEMENT FOR PYTHON 2.0
        - CNRI LICENSE AGREEMENT FOR PYTHON 1.6.1
        - CWI LICENSE AGREEMENT FOR PYTHON 0.9.0 THROUGH 1.2
        - ZERO-CLAUSE BSD LICENSE FOR CODE IN THE PYTHON DOCUMENTATION
    - Licenses and Acknowledgements for Incorporated Software
        - Mersenne Twister
        - Sockets
        - Asynchronous socket services
        - Cookie management
        - Execution tracing
        - UUencode and UUdecode functions
        - XML Remote Procedure Calls
        - test_epoll

- Select kqueue
- SipHash24
- strtod and dtoa
- OpenSSL
- expat
- libffi
- zlib
- cfuhash
- libmpdec
- W3C C14N test suite
- mimalloc
- asyncio
- Global Unbounded Sequences (GUS)
- Zstandard bindings

What’s New in Python

The “What’s New in Python” series of essays takes tours through the most important changes between major Python versions. They are a “must read” for anyone wishing to stay up-to-date after a new release.

- What’s new in Python 3.14
    - Summary – Release highlights
    - New features
    - Other language changes
    - New modules
    - Improved modules
    - Optimizations
    - Removed
    - Deprecated
    - CPython bytecode changes
    - C API changes
    - Build changes
    - Porting to Python 3.14
    - Notable changes in 3.14.1
- What’s New In Python 3.13
    - Summary – Release Highlights
    - New Features
    - Other Language Changes
    - New Modules
    - Improved Modules
    - Optimizations
    - Removed Modules And APIs
    - New Deprecations
    - CPython Bytecode Changes
    - C API Changes
    - Build Changes

    - Porting to Python 3.13
    - Regression Test Changes
- What’s New In Python 3.12
    - Summary – Release highlights
    - New Features
    - New Features Related to Type Hints
    - Other Language Changes
    - New Modules
    - Improved Modules
    - Optimizations
    - CPython bytecode changes
    - Demos and Tools
    - Deprecated
    - Removed
    - Porting to Python 3.12
    - Build Changes
    - C API Changes
- What’s New In Python 3.11
    - Summary – Release highlights
    - New Features
    - New Features Related to Type Hints
    - Other Language Changes
    - Other CPython Implementation Changes
    - New Modules
    - Improved Modules
    - Optimizations
    - Faster CPython
    - CPython bytecode changes
    - Deprecated
    - Pending Removal in Python 3.12
    - Removed
    - Porting to Python 3.11
    - Build Changes
    - C API Changes
    - Notable changes in 3.11.4
    - Notable changes in 3.11.5

- What’s New In Python 3.10
    - Summary – Release highlights
    - New Features
    - New Features Related to Type Hints
    - Other Language Changes
    - New Modules
    - Improved Modules
    - Optimizations
    - Deprecated
    - Removed
    - Porting to Python 3.10
    - CPython bytecode changes
    - Build Changes
    - C API Changes
    - Notable security feature in 3.10.7
    - Notable security feature in 3.10.8
    - Notable changes in 3.10.12
- What’s New In Python 3.9
    - Summary – Release highlights
    - You should check for DeprecationWarning in your code
    - New Features
    - Other Language Changes
    - New Modules
    - Improved Modules
    - Optimizations
    - Deprecated
    - Removed
    - Porting to Python 3.9
    - Build Changes
    - C API Changes
    - Notable changes in Python 3.9.1
    - Notable changes in Python 3.9.2
    - Notable changes in Python 3.9.3
    - Notable changes in Python 3.9.5
    - Notable security feature in 3.9.14

  - Notable changes in 3.9.17
- What’s New In Python 3.8
  - Summary – Release highlights
  - New Features
  - Other Language Changes
  - New Modules
  - Improved Modules
  - Optimizations
  - Build and C API Changes
  - Deprecated
  - API and Feature Removals
  - Porting to Python 3.8
  - Notable changes in Python 3.8.1
  - Notable changes in Python 3.8.2
  - Notable changes in Python 3.8.3
  - Notable changes in Python 3.8.8
  - Notable changes in Python 3.8.9
  - Notable changes in Python 3.8.10
  - Notable changes in Python 3.8.10
  - Notable changes in Python 3.8.12
  - Notable security feature in 3.8.14
  - Notable changes in 3.8.17
- What’s New In Python 3.7
  - Summary – Release Highlights
  - New Features
  - Other Language Changes
  - New Modules
  - Improved Modules
  - C API Changes
  - Build Changes
  - Optimizations
  - Other CPython Implementation Changes
  - Deprecated Python Behavior
  - Deprecated Python modules, functions and methods
  - Deprecated functions and types of the C API
  - Platform Support Removals

    - API and Feature Removals
    - Module Removals
    - Windows-only Changes
    - Porting to Python 3.7
    - Notable changes in Python 3.7.1
    - Notable changes in Python 3.7.2
    - Notable changes in Python 3.7.6
    - Notable changes in Python 3.7.10
    - Notable changes in Python 3.7.11
    - Notable security feature in 3.7.14
- What’s New In Python 3.6
    - Summary – Release highlights
    - New Features
    - Other Language Changes
    - New Modules
    - Improved Modules
    - Optimizations
    - Build and C API Changes
    - Other Improvements
    - Deprecated
    - Removed
    - Porting to Python 3.6
    - Notable changes in Python 3.6.2
    - Notable changes in Python 3.6.4
    - Notable changes in Python 3.6.5
    - Notable changes in Python 3.6.7
    - Notable changes in Python 3.6.10
    - Notable changes in Python 3.6.13
    - Notable changes in Python 3.6.14
- What’s New In Python 3.5
    - Summary – Release highlights
    - New Features
    - Other Language Changes
    - New Modules
    - Improved Modules
    - Other module-level changes

  - Optimizations
  - Build and C API Changes
  - Deprecated
  - Removed
  - Porting to Python 3.5
  - Notable changes in Python 3.5.4
- What’s New In Python 3.4
  - Summary – Release Highlights
  - New Features
  - New Modules
  - Improved Modules
  - CPython Implementation Changes
  - Deprecated
  - Removed
  - Porting to Python 3.4
  - Changed in 3.4.3
- What’s New In Python 3.3
  - Summary – Release highlights
  - PEP 405: Virtual Environments
  - PEP 420: Implicit Namespace Packages
  - PEP 3118: New memoryview implementation and buffer protocol documentation
  - PEP 393: Flexible String Representation
  - PEP 397: Python Launcher for Windows
  - PEP 3151: Reworking the OS and IO exception hierarchy
  - PEP 380: Syntax for Delegating to a Subgenerator
  - PEP 409: Suppressing exception context
  - PEP 414: Explicit Unicode literals
  - PEP 3155: Qualified name for classes and functions
  - PEP 412: Key-Sharing Dictionary
  - PEP 362: Function Signature Object
  - PEP 421: Adding sys.implementation
  - Using importlib as the Implementation of Import
  - Other Language Changes
  - A Finer-Grained Import Lock

    - Builtin functions and types
    - New Modules
    - Improved Modules
    - Optimizations
    - Build and C API Changes
    - Deprecated
    - Porting to Python 3.3
- What’s New In Python 3.2
    - PEP 384: Defining a Stable ABI
    - PEP 389: Argparse Command Line Parsing Module
    - PEP 391: Dictionary Based Configuration for Logging
    - PEP 3148: The `concurrent.futures` module
    - PEP 3147: PYC Repository Directories
    - PEP 3149: ABI Version Tagged .so Files
    - PEP 3333: Python Web Server Gateway Interface v1.0.1
    - Other Language Changes
    - New, Improved, and Deprecated Modules
    - Multi-threading
    - Optimizations
    - Unicode
    - Codecs
    - Documentation
    - IDLE
    - Code Repository
    - Build and C API Changes
    - Porting to Python 3.2
- What’s New In Python 3.1
    - PEP 372: Ordered Dictionaries
    - PEP 378: Format Specifier for Thousands Separator
    - Other Language Changes
    - New, Improved, and Deprecated Modules
    - Optimizations
    - IDLE
    - Build and C API Changes
    - Porting to Python 3.1

- What’s New In Python 3.0
    - Common Stumbling Blocks
    - Overview Of Syntax Changes
    - Changes Already Present In Python 2.6
    - Library Changes
    - **PEP 3101**: A New Approach To String Formatting
    - Changes To Exceptions
    - Miscellaneous Other Changes
    - Build and C API Changes
    - Performance
    - Porting To Python 3.0
- What’s New in Python 2.7
    - The Future for Python 2.x
    - Changes to the Handling of Deprecation Warnings
    - Python 3.1 Features
    - PEP 372: Adding an Ordered Dictionary to collections
    - PEP 378: Format Specifier for Thousands Separator
    - PEP 389: The argparse Module for Parsing Command Lines
    - PEP 391: Dictionary-Based Configuration For Logging
    - PEP 3106: Dictionary Views
    - PEP 3137: The memoryview Object
    - Other Language Changes
    - New and Improved Modules
    - Build and C API Changes
    - Other Changes and Fixes
    - Porting to Python 2.7
    - New Features Added to Python 2.7 Maintenance Releases
    - Acknowledgements
- What’s New in Python 2.6
    - Python 3.0
    - Changes to the Development Process
    - PEP 343: The ‘with’ statement

    - PEP 366: Explicit Relative Imports From a Main Module
    - PEP 370: Per-user `site-packages` Directory
    - PEP 371: The `multiprocessing` Package
    - PEP 3101: Advanced String Formatting
    - PEP 3105: `print` As a Function
    - PEP 3110: Exception-Handling Changes
    - PEP 3112: Byte Literals
    - PEP 3116: New I/O Library
    - PEP 3118: Revised Buffer Protocol
    - PEP 3119: Abstract Base Classes
    - PEP 3127: Integer Literal Support and Syntax
    - PEP 3129: Class Decorators
    - PEP 3141: A Type Hierarchy for Numbers
    - Other Language Changes
    - New and Improved Modules
    - Deprecations and Removals
    - Build and C API Changes
    - Porting to Python 2.6
    - Acknowledgements
- What’s New in Python 2.5
    - PEP 308: Conditional Expressions
    - PEP 309: Partial Function Application
    - PEP 314: Metadata for Python Software Packages v1.1
    - PEP 328: Absolute and Relative Imports
    - PEP 338: Executing Modules as Scripts
    - PEP 341: Unified try/except/finally
    - PEP 342: New Generator Features
    - PEP 343: The ‘with’ statement
    - PEP 352: Exceptions as New-Style Classes
    - PEP 353: Using ssize_t as the index type
    - PEP 357: The ‘__index__’ method
    - Other Language Changes
    - New, Improved, and Removed Modules
    - Build and C API Changes

  - Porting to Python 2.5
  - Acknowledgements
- What’s New in Python 2.4
  - PEP 218: Built-In Set Objects
  - PEP 237: Unifying Long Integers and Integers
  - PEP 289: Generator Expressions
  - PEP 292: Simpler String Substitutions
  - PEP 318: Decorators for Functions and Methods
  - PEP 322: Reverse Iteration
  - PEP 324: New subprocess Module
  - PEP 327: Decimal Data Type
  - PEP 328: Multi-line Imports
  - PEP 331: Locale-Independent Float/String Conversions
  - Other Language Changes
  - New, Improved, and Deprecated Modules
  - Build and C API Changes
  - Porting to Python 2.4
  - Acknowledgements
- What’s New in Python 2.3
  - PEP 218: A Standard Set Datatype
  - PEP 255: Simple Generators
  - PEP 263: Source Code Encodings
  - PEP 273: Importing Modules from ZIP Archives
  - PEP 277: Unicode file name support for Windows NT
  - PEP 278: Universal Newline Support
  - PEP 279: enumerate()
  - PEP 282: The logging Package
  - PEP 285: A Boolean Type
  - PEP 293: Codec Error Handling Callbacks
  - PEP 301: Package Index and Metadata for Distutils
  - PEP 302: New Import Hooks
  - PEP 305: Comma-separated Files
  - PEP 307: Pickle Enhancements
  - Extended Slices
  - Other Language Changes

  - New, Improved, and Deprecated Modules
  - Pymalloc: A Specialized Object Allocator
  - Build and C API Changes
  - Other Changes and Fixes
  - Porting to Python 2.3
  - Acknowledgements
- What’s New in Python 2.2
  - Introduction
  - PEPs 252 and 253: Type and Class Changes
  - PEP 234: Iterators
  - PEP 255: Simple Generators
  - PEP 237: Unifying Long Integers and Integers
  - PEP 238: Changing the Division Operator
  - Unicode Changes
  - PEP 227: Nested Scopes
  - New and Improved Modules
  - Interpreter Changes and Fixes
  - Other Changes and Fixes
  - Acknowledgements
- What’s New in Python 2.1
  - Introduction
  - PEP 227: Nested Scopes
  - PEP 236: __future__ Directives
  - PEP 207: Rich Comparisons
  - PEP 230: Warning Framework
  - PEP 229: New Build System
  - PEP 205: Weak References
  - PEP 232: Function Attributes
  - PEP 235: Importing Modules on Case-Insensitive Platforms
  - PEP 217: Interactive Display Hook
  - PEP 208: New Coercion Model
  - PEP 241: Metadata in Python Packages
  - New and Improved Modules
  - Other Changes and Fixes
  - Acknowledgements

- What’s New in Python 2.0
    - Introduction
    - What About Python 1.6?
    - New Development Process
    - Unicode
    - List Comprehensions
    - Augmented Assignment
    - String Methods
    - Garbage Collection of Cycles
    - Other Core Changes
    - Porting to 2.0
    - Extending/Embedding Changes
    - Distutils: Making Modules Easy to Install
    - XML Modules
    - Module changes
    - New modules
    - IDLE Improvements
    - Deleted and Deprecated Modules
    - Acknowledgements

The “Changelog” is an HTML version of the file built [https://pypi.org/project/blurb/] from the contents of the Misc/NEWS.d [https://github.com/python/cpython/tree/3.14/Misc/NEWS.d] directory tree, which contains *all* nontrivial changes to Python for the current version.

- Changelog
    - Python next
    - Python 3.14.2 final
    - Python 3.14.1 final
    - Python 3.14.0 final
    - Python 3.14.0 release candidate 3
    - Python 3.14.0 release candidate 2
    - Python 3.14.0 release candidate 1
    - Python 3.14.0 beta 4
    - Python 3.14.0 beta 3

- Python 3.14.0 beta 2
- Python 3.14.0 beta 1
- Python 3.14.0 alpha 7
- Python 3.14.0 alpha 6
- Python 3.14.0 alpha 5
- Python 3.14.0 alpha 4
- Python 3.14.0 alpha 3
- Python 3.14.0 alpha 2
- Python 3.14.0 alpha 1
- Python 3.13.0 beta 1
- Python 3.13.0 alpha 6
- Python 3.13.0 alpha 5
- Python 3.13.0 alpha 4
- Python 3.13.0 alpha 3
- Python 3.13.0 alpha 2
- Python 3.13.0 alpha 1
- Python 3.12.0 beta 1
- Python 3.12.0 alpha 7
- Python 3.12.0 alpha 6
- Python 3.12.0 alpha 5
- Python 3.12.0 alpha 4
- Python 3.12.0 alpha 3
- Python 3.12.0 alpha 2
- Python 3.12.0 alpha 1
- Python 3.11.0 beta 1
- Python 3.11.0 alpha 7
- Python 3.11.0 alpha 6
- Python 3.11.0 alpha 5
- Python 3.11.0 alpha 4
- Python 3.11.0 alpha 3
- Python 3.11.0 alpha 2
- Python 3.11.0 alpha 1
- Python 3.10.0 beta 1
- Python 3.10.0 alpha 7
- Python 3.10.0 alpha 6
- Python 3.10.0 alpha 5

- Python 3.10.0 alpha 4
- Python 3.10.0 alpha 3
- Python 3.10.0 alpha 2
- Python 3.10.0 alpha 1
- Python 3.9.0 beta 1
- Python 3.9.0 alpha 6
- Python 3.9.0 alpha 5
- Python 3.9.0 alpha 4
- Python 3.9.0 alpha 3
- Python 3.9.0 alpha 2
- Python 3.9.0 alpha 1
- Python 3.8.0 beta 1
- Python 3.8.0 alpha 4
- Python 3.8.0 alpha 3
- Python 3.8.0 alpha 2
- Python 3.8.0 alpha 1
- Python 3.7.0 final
- Python 3.7.0 release candidate 1
- Python 3.7.0 beta 5
- Python 3.7.0 beta 4
- Python 3.7.0 beta 3
- Python 3.7.0 beta 2
- Python 3.7.0 beta 1
- Python 3.7.0 alpha 4
- Python 3.7.0 alpha 3
- Python 3.7.0 alpha 2
- Python 3.7.0 alpha 1
- Python 3.6.6 final
- Python 3.6.6 release candidate 1
- Python 3.6.5 final
- Python 3.6.5 release candidate 1
- Python 3.6.4 final
- Python 3.6.4 release candidate 1
- Python 3.6.3 final
- Python 3.6.3 release candidate 1
- Python 3.6.2 final

  - Python 3.6.2 release candidate 2
  - Python 3.6.2 release candidate 1
  - Python 3.6.1 final
  - Python 3.6.1 release candidate 1
  - Python 3.6.0 final
  - Python 3.6.0 release candidate 2
  - Python 3.6.0 release candidate 1
  - Python 3.6.0 beta 4
  - Python 3.6.0 beta 3
  - Python 3.6.0 beta 2
  - Python 3.6.0 beta 1
  - Python 3.6.0 alpha 4
  - Python 3.6.0 alpha 3
  - Python 3.6.0 alpha 2
  - Python 3.6.0 alpha 1
  - Python 3.5.5 final
  - Python 3.5.5 release candidate 1
  - Python 3.5.4 final
  - Python 3.5.4 release candidate 1
  - Python 3.5.3 final
  - Python 3.5.3 release candidate 1
  - Python 3.5.2 final
  - Python 3.5.2 release candidate 1
  - Python 3.5.1 final
  - Python 3.5.1 release candidate 1
  - Python 3.5.0 final
  - Python 3.5.0 release candidate 4
  - Python 3.5.0 release candidate 3
  - Python 3.5.0 release candidate 2
  - Python 3.5.0 release candidate 1
  - Python 3.5.0 beta 4
  - Python 3.5.0 beta 3
  - Python 3.5.0 beta 2
  - Python 3.5.0 beta 1
  - Python 3.5.0 alpha 4
  - Python 3.5.0 alpha 3

    - Python 3.5.0 alpha 2
    - Python 3.5.0 alpha 1

What’s new in Python 3.14

Editors:: Adam Turner and Hugo van Kemenade

This article explains the new features in Python 3.14, compared to 3.13. Python 3.14 was released on 7 October 2025. For full details, see the changelog.

See also

PEP 745 [https://peps.python.org/pep-0745/] – Python 3.14 release schedule

Summary – Release highlights

Python 3.14 is the latest stable release of the Python programming language, with a mix of changes to the language, the implementation, and the standard library. The biggest changes include template string literals, deferred evaluation of annotations, and support for subinterpreters in the standard library.

The library changes include significantly improved capabilities for introspection in asyncio, support for Zstandard via a new `compression.zstd` module, syntax highlighting in the REPL, as well as the usual deprecations and removals, and improvements in user-friendliness and correctness.

This article doesn’t attempt to provide a complete specification of all new features, but instead gives a convenient overview. For full details refer to the documentation, such as the Library Reference and Language Reference. To understand the complete implementation and design rationale for a change, refer to the PEP for a particular new feature; but note that PEPs usually are not kept up-to-date once a feature has been fully implemented. See Porting to Python 3.14 for guidance on upgrading from earlier versions of Python.

Interpreter improvements:

- **PEP 649** [https://peps.python.org/pep-0649/] and **PEP 749** [https://peps.python.org/pep-0749/]: Deferred evaluation of annotations
- **PEP 734** [https://peps.python.org/pep-0734/]: Multiple interpreters in the standard library
- **PEP 750** [https://peps.python.org/pep-0750/]: Template strings
- **PEP 758** [https://peps.python.org/pep-0758/]: Allow except and except* expressions without brackets
- **PEP 765** [https://peps.python.org/pep-0765/]: Control flow in finally blocks
- **PEP 768** [https://peps.python.org/pep-0768/]: Safe external debugger interface for CPython
- A new type of interpreter
- Free-threaded mode improvements
- Improved error messages
- Incremental garbage collection

Significant improvements in the standard library:

- **PEP 784** [https://peps.python.org/pep-0784/]: Zstandard support in the standard library
- Asyncio introspection capabilities
- Concurrent safe warnings control
- Syntax highlighting in the default interactive shell, and color output in several standard library CLIs

C API improvements:

- **PEP 741** [https://peps.python.org/pep-0741/]: Python configuration C API

Platform support:

- **PEP 776** [https://peps.python.org/pep-0776/]: Emscripten is now an officially supported platform, at **tier 3** [https://peps.python.org/pep-0011/#tier-3].

Release changes:

- **PEP 779** [https://peps.python.org/pep-0779/]: Free-threaded Python is officially supported
- **PEP 761** [https://peps.python.org/pep-0761/]: PGP signatures have been discontinued for official releases
- Windows and macOS binary releases now support the experimental just-in-time compiler
- Binary releases for Android are now provided

New features

PEP 649 [https://peps.python.org/pep-0649/] & PEP 749 [https://peps.python.org/pep-0749/]: Deferred evaluation of annotations

The annotations on functions, classes, and modules are no longer evaluated eagerly. Instead, annotations are stored in special-purpose annotate functions and evaluated only when necessary (except if `from __future__ import annotations` is used).

This change is designed to improve performance and usability of annotations in Python in most circumstances. The runtime cost for defining annotations is minimized, but it remains possible to introspect annotations at runtime. It is no longer necessary to enclose annotations in strings if they contain forward references.

The new **`annotationlib`** module provides tools for inspecting deferred annotations. Annotations may be evaluated in the **`VALUE`** format (which evaluates annotations to runtime values, similar to the behavior in earlier Python versions), the **`FORWARDREF`** format (which replaces undefined names with special markers), and the **`STRING`** format (which returns annotations as strings).

This example shows how these formats behave:

```
>>> from annotationlib import get_annotations, Format
>>> def func(arg: Undefined):
...     pass
>>> get_annotations(func, format=Format.VALUE)
Traceback (most recent call last):
  ...
NameError: name 'Undefined' is not defined
>>> get_annotations(func, format=Format.FORWARDREF)
{'arg': ForwardRef('Undefined', owner=<function func at
0x...>)}
>>> get_annotations(func, format=Format.STRING)
{'arg': 'Undefined'}
```

The porting section contains guidance on changes that may be needed due to these changes, though in the majority of cases, code will continue working as-is.

(Contributed by Jelle Zijlstra in **PEP 749** [https://peps.python.org/pep-0749/] and gh-119180 [https://github.com/python/cpython/issues/119180]; **PEP 649** [https://peps.python.org/pep-0649/] was written by Larry Hastings.)

See also

PEP 649 **[https://peps.python.org/pep-0649/]**
Deferred Evaluation Of Annotations Using Descriptors

PEP 749 **[https://peps.python.org/pep-0749/]**
Implementing PEP 649

PEP 734 [https://peps.python.org/pep-0734/]: Multiple interpreters in the standard library

The CPython runtime supports running multiple copies of Python in the same process simultaneously and has done so for over 20 years. Each of these separate copies is called an ‘interpreter’. However, the feature had been available only through the C-API.

That limitation is removed in Python 3.14, with the new `concurrent.interpreters` module.

There are at least two notable reasons why using multiple interpreters has significant benefits:

- they support a new (to Python), human-friendly concurrency model
- true multi-core parallelism

For some use cases, concurrency in software improves efficiency and can simplify design, at a high level. At the same time, implementing and maintaining all but the simplest concurrency is often a struggle for the human brain. That especially applies to plain threads (for example, `threading`), where all memory is shared between all threads.

With multiple isolated interpreters, you can take advantage of a class of concurrency models, like Communicating Sequential Processes (CSP) or the actor model, that have found success in other programming languages, like Smalltalk, Erlang, Haskell, and Go. Think of multiple interpreters as threads but with opt-in sharing.

Regarding multi-core parallelism: as of Python 3.12, interpreters are now sufficiently isolated from one another to be used in parallel (see **PEP 684** [https://peps.python.org/pep-0684/]). This unlocks a variety of CPU-intensive use cases for Python that were limited by the GIL.

Using multiple interpreters is similar in many ways to **`multiprocessing`**, in that they both provide isolated logical “processes” that can run in parallel, with no sharing by default. However, when using multiple interpreters, an application will use fewer system resources and will operate more efficiently (since it stays within the same process). Think of multiple interpreters as having the isolation of processes with the efficiency of threads.

While the feature has been around for decades, multiple interpreters have not been used widely, due to low awareness and the lack of a standard library module. Consequently, they currently have several notable limitations, which are expected to improve significantly now that the feature is going mainstream.

Current limitations:

- starting each interpreter has not been optimized yet
- each interpreter uses more memory than necessary (work continues on extensive internal sharing between interpreters)
- there aren’t many options *yet* for truly sharing objects or other data between interpreters (other than `memoryview`)

- many third-party extension modules on PyPI are not yet compatible with multiple interpreters (all standard library extension modules *are* compatible)
- the approach to writing applications that use multiple isolated interpreters is mostly unfamiliar to Python users, for now

The impact of these limitations will depend on future CPython improvements, how interpreters are used, and what the community solves through PyPI packages. Depending on the use case, the limitations may not have much impact, so try it out!

Furthermore, future CPython releases will reduce or eliminate overhead and provide utilities that are less appropriate on PyPI. In the meantime, most of the limitations can also be addressed through extension modules, meaning PyPI packages can fill any gap for 3.14, and even back to 3.12 where interpreters were finally properly isolated and stopped sharing the GIL. Likewise, libraries on PyPI are expected to emerge for high-level abstractions on top of interpreters.

Regarding extension modules, work is in progress to update some PyPI projects, as well as tools like Cython, pybind11, nanobind, and PyO3. The steps for isolating an extension module are found at Isolating Extension Modules. Isolating a module has a lot of overlap with what is required to support free-threading, so the ongoing work in the community in that area will help accelerate support for multiple interpreters.

Also added in 3.14: concurrent.futures.InterpreterPoolExecutor.

(Contributed by Eric Snow in gh-134939 [https://github.com/python/cpython/issues/134939].)

See also

PEP 734 [https://peps.python.org/pep-0734/]

PEP 750 [https://peps.python.org/pep-0750/]: Template string literals

Template strings are a new mechanism for custom string processing. They share the familiar syntax of f-strings but, unlike f-strings, return an object representing the static and interpolated parts of the string, instead of a simple `str`.

To write a t-string, use a `'t'` prefix instead of an `'f'`:

```
>>> variety = 'Stilton'
>>> template = t'Try some {variety} cheese!'
>>> type(template)
<class 'string.templatelib.Template'>
```

`Template` objects provide access to the static and interpolated (in curly braces) parts of a string *before* they are combined. Iterate over **`Template`** instances to access their parts in order:

```
>>> list(template)
['Try some ', Interpolation('Stilton', 'variety', None, ''), '
cheese!']
```

It’s easy to write (or call) code to process **`Template`** instances. For example, here’s a function that renders static parts lowercase and `Interpolation` instances uppercase:

```
from string.templatelib import Interpolation

def lower_upper(template):
    """Render static parts lowercase and interpolations
uppercase."""
    parts = []
    for part in template:
        if isinstance(part, Interpolation):
            parts.append(str(part.value).upper())
        else:
            parts.append(part.lower())
    return ''.join(parts)

name = 'Wenslydale'
template = t'Mister {name}'
assert lower_upper(template) == 'mister WENSLYDALE'
```

Because **`Template`** instances distinguish between static strings and interpolations at runtime, they can be useful for sanitising user input. Writing a **`html()`** function that escapes user input in HTML is an exercise left to the reader! Template processing code can provide improved flexibility. For instance, a more advanced **`html()`** function could accept a **`dict`** of HTML attributes directly in the template:

```
attributes = {'src': 'limburger.jpg', 'alt': 'lovely cheese'}
template = t'<img {attributes}>'
assert html(template) == '<img src="limburger.jpg" alt="lovely
cheese" />'
```

Of course, template processing code does not need to return a string-like result. An even *more* advanced **`html()`** could return a custom type representing a DOM-like structure.

With t-strings in place, developers can write systems that sanitise SQL, make safe shell operations, improve logging, tackle modern ideas in web development (HTML, CSS, and so on), and implement lightweight custom business DSLs.

(Contributed by Jim Baker, Guido van Rossum, Paul Everitt, Koudai Aono, Lysandros Nikolaou, Dave Peck, Adam Turner, Jelle Zijlstra, Bénédikt Tran, and Pablo Galindo Salgado in gh-132661 [https://github.com/python/cpython/issues/132661].)

See also

PEP 750 [https://peps.python.org/pep-0750/].

PEP 768 [https://peps.python.org/pep-0768/]: Safe external debugger interface

Python 3.14 introduces a zero-overhead debugging interface that allows debuggers and profilers to safely attach to running Python processes without stopping or restarting them. This is a significant enhancement to Python's debugging capabilities, meaning that unsafe alternatives are no longer required.

The new interface provides safe execution points for attaching debugger code without modifying the interpreter's normal execution path or adding any overhead at runtime. Due to this, tools can now inspect and interact with Python applications in real-time, which is a crucial capability for high-availability systems and production environments.

For convenience, this interface is implemented in the `sys.remote_exec()` function. For example:

```
import sys
from tempfile import NamedTemporaryFile

with NamedTemporaryFile(mode='w', suffix='.py', delete=False)
as f:
    script_path = f.name
    f.write(f'import my_debugger;
my_debugger.connect({os.getpid()})')
```

```
# Execute in process with PID 1234
print('Behold! An offering:')
sys.remote_exec(1234, script_path)
```

This function allows sending Python code to be executed in a target process at the next safe execution point. However, tool authors can also implement the protocol directly as described in the PEP, which details the underlying mechanisms used to safely attach to running processes.

The debugging interface has been carefully designed with security in mind and includes several mechanisms to control access:

- A **`PYTHON_DISABLE_REMOTE_DEBUG`** environment variable.
- A **`-X disable-remote-debug`** command-line option.
- A **`--without-remote-debug`** configure flag to completely disable the feature at build time.

(Contributed by Pablo Galindo Salgado, Matt Wozniski, and Ivona Stojanovic in gh-131591 [https://github.com/python/cpython/issues/131591].)

See also

PEP 768 [https://peps.python.org/pep-0768/].

A new type of interpreter

A new type of interpreter has been added to CPython. It uses tail calls between small C functions that implement individual Python opcodes, rather than one large C `case` statement. For certain newer compilers, this interpreter provides significantly better performance. Preliminary benchmarks suggest a geometric mean of 3-5% faster on the standard `pyperformance` benchmark suite, depending on platform and architecture. The baseline is Python 3.14 built with Clang 19, without this new interpreter.

This interpreter currently only works with Clang 19 and newer on x86-64 and AArch64 architectures. However, a future release of GCC is expected to support this as well.

This feature is opt-in for now. Enabling profile-guided optimization is highly recommendeded when using the new interpreter as it is the only configuration that has been tested and validated for improved performance. For further information, see `--with-tail-call-interp`.

Note

This is not to be confused with tail call optimization [https://en.wikipedia.org/wiki/Tail_call] of Python functions, which is currently not implemented in CPython.

This new interpreter type is an internal implementation detail of the CPython interpreter. It doesn't change the visible behavior of Python programs at all. It can improve their performance, but doesn't change anything else.

(Contributed by Ken Jin in gh-128563 [https://github.com/python/cpython/issues/128563], with ideas on how to implement this in CPython by Mark Shannon, Garrett Gu, Haoran Xu, and Josh Haberman.)

Free-threaded mode improvements

CPython's free-threaded mode (**PEP 703** [https://peps.python.org/pep-0703/]), initially added in 3.13, has been significantly improved in Python 3.14. The implementation described in PEP 703 has been finished, including C API changes, and temporary workarounds in the interpreter were replaced with more permanent solutions. The specializing adaptive interpreter (**PEP 659** [https://peps.python.org/pep-0659/]) is now enabled in free-threaded mode, which along with many other optimizations greatly improves its performance. The performance penalty on single-threaded code in free-threaded mode is now roughly 5-10%, depending on the platform and C compiler used.

From Python 3.14, when compiling extension modules for the free-threaded build of CPython on Windows, the preprocessor variable `Py_GIL_DISABLED` now needs to be specified by the build backend, as it will no longer be determined automatically by the C compiler. For a running interpreter, the setting that was used at compile time can be found using `sysconfig.get_config_var()`.

The new `-X context_aware_warnings` flag controls if concurrent safe warnings control is enabled. The flag defaults to true for the free-threaded build and false for the GIL-enabled build.

A new `thread_inherit_context` flag has been added, which if enabled means that threads created with `threading.Thread` start with a copy of the `Context()` of the caller of `start()`. Most significantly, this makes the warning filtering context established by `catch_warnings` be “inherited” by threads (or asyncio tasks) started within that context. It also affects other modules that use context variables, such as the `decimal` context manager. This flag defaults to true for the free-threaded build and false for the GIL-enabled build.

(Contributed by Sam Gross, Matt Page, Neil Schemenauer, Thomas Wouters, Donghee Na, Kirill Podoprigora, Ken Jin, Itamar Oren, Brett Simmers, Dino Viehland, Nathan Goldbaum, Ralf Gommers, Lysandros Nikolaou, Kumar

Aditya, Edgar Margffoy, and many others. Some of these contributors are employed by Meta, which has continued to provide significant engineering resources to support this project.)

Improved error messages

- The interpreter now provides helpful suggestions when it detects typos in Python keywords. When a word that closely resembles a Python keyword is encountered, the interpreter will suggest the correct keyword in the error message. This feature helps programmers quickly identify and fix common typing mistakes. For example:

  ```
  >>> whille True:
  ...     pass
  Traceback (most recent call last):
    File "<stdin>", line 1
      whille True:
      ^^^^^^
  SyntaxError: invalid syntax. Did you mean 'while'?
  ```

  While the feature focuses on the most common cases, some variations of misspellings may still result in regular syntax errors. (Contributed by Pablo Galindo in gh-132449 [https://github.com/python/cpython/issues/132449].)

- **`elif`** statements that follow an **`else`** block now have a specific error message. (Contributed by Steele Farnsworth in gh-129902 [https://github.com/python/cpython/issues/129902].)

  ```
  >>> if who == "me":
  ...     print("It's me!")
  ... else:
  ...     print("It's not me!")
  ... elif who is None:
  ...     print("Who is it?")
  File "<stdin>", line 5
  ```

```
  elif who is None:
  ^^^^
SyntaxError: 'elif' block follows an 'else' block
```

- If a statement is passed to the Conditional expressions after **`else`**, or one of **`pass`**, **`break`**, or **`continue`** is passed before **`if`**, then the error message highlights where the **`expression`** is required. (Contributed by Sergey Miryanov in gh-129515 [https://github.com/python/cpython/issues/129515].)

```
>>> x = 1 if True else pass
Traceback (most recent call last):
  File "<string>", line 1
    x = 1 if True else pass
                       ^^^^
SyntaxError: expected expression after 'else', but
statement is given

>>> x = continue if True else break
Traceback (most recent call last):
  File "<string>", line 1
    x = continue if True else break
        ^^^^^^^^
SyntaxError: expected expression before 'if', but statement
is given
```

- When incorrectly closed strings are detected, the error message suggests that the string may be intended to be part of the string. (Contributed by Pablo Galindo in gh-88535 [https://github.com/python/cpython/issues/88535].)

```
>>> "The interesting object "The important object" is very
important"
Traceback (most recent call last):
SyntaxError: invalid syntax. Is this intended to be part of
the string?
```

- When strings have incompatible prefixes, the error now shows which prefixes are incompatible. (Contributed by

Nikita Sobolev in gh-133197 [https://github.com/python/cpython/issues/133197].)

```
>>> ub'abc'
  File "<python-input-0>", line 1
    ub'abc'
    ^^
SyntaxError: 'u' and 'b' prefixes are incompatible
```

- Improved error messages when using `as` with incompatible targets in:
  - Imports: `import ... as ...`
  - From imports: `from ... import ... as ...`
  - Except handlers: `except ... as ...`
  - Pattern-match cases: `case ... as ...`

  (Contributed by Nikita Sobolev in gh-123539 [https://github.com/python/cpython/issues/123539], gh-123562 [https://github.com/python/cpython/issues/123562], and gh-123440 [https://github.com/python/cpython/issues/123440].)

- Improved error message when trying to add an instance of an unhashable type to a **`dict`** or **`set`**. (Contributed by CF Bolz-Tereick and Victor Stinner in gh-132828 [https://github.com/python/cpython/issues/132828].)

```
>>> s = set()
>>> s.add({'pages': 12, 'grade': 'A'})
Traceback (most recent call last):
  File "<python-input-1>", line 1, in <module>
    s.add({'pages': 12, 'grade': 'A'})
    ~~~~~^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
TypeError: cannot use 'dict' as a set element (unhashable
type: 'dict')
>>> d = {}
>>> l = [1, 2, 3]
>>> d[l] = 12
Traceback (most recent call last):
  File "<python-input-4>", line 1, in <module>
```

```
    d[l] = 12
    ~^^^
TypeError: cannot use 'list' as a dict key (unhashable
type: 'list')
```

- Improved error message when an object supporting the synchronous context manager protocol is entered using `async with` instead of `with`, and vice versa for the asynchronous context manager protocol. (Contributed by Bénédikt Tran in gh-128398 [https://github.com/python/cpython/issues/128398].)

PEP 784 [https://peps.python.org/pep-0784/]: Zstandard support in the standard library

The new `compression` package contains modules `compression.lzma`, `compression.bz2`, `compression.gzip` and `compression.zlib` which re-export the `lzma`, `bz2`, `gzip` and `zlib` modules respectively. The new import names under `compression` are the preferred names for importing these compression modules from Python 3.14. However, the existing modules names have not been deprecated. Any deprecation or removal of the existing compression modules will occur no sooner than five years after the release of 3.14.

The new `compression.zstd` module provides compression and decompression APIs for the Zstandard format via bindings to Meta's zstd library [https://facebook.github.io/zstd/]. Zstandard is a widely adopted, highly efficient, and fast compression format. In addition to the APIs introduced in `compression.zstd`, support for reading and writing Zstandard compressed archives has been added to the `tarfile`, `zipfile`, and `shutil` modules.

Here's an example of using the new module to compress some data:

```
from compression import zstd
import math

data = str(math.pi).encode() * 20
compressed = zstd.compress(data)
ratio = len(compressed) / len(data)
print(f"Achieved compression ratio of {ratio}")
```

As can be seen, the API is similar to the APIs of the **`lzma`** and **`bz2`** modules.

(Contributed by Emma Harper Smith, Adam Turner, Gregory P. Smith, Tomas Roun, Victor Stinner, and Rogdham in gh-132983 [https://github.com/python/cpython/issues/132983].)

See also

PEP 784 [https://peps.python.org/pep-0784/].

Asyncio introspection capabilities

Added a new command-line interface to inspect running Python processes using asynchronous tasks, available via `python -m asyncio ps PID` or `python -m asyncio pstree PID`.

The `ps` subcommand inspects the given process ID (PID) and displays information about currently running asyncio tasks. It outputs a task table: a flat listing of all tasks, their names, their coroutine stacks, and which tasks are awaiting them.

The `pstree` subcommand fetches the same information, but instead renders a visual async call tree, showing coroutine relationships in a hierarchical format. This command is particularly useful for debugging long-running or stuck asynchronous programs. It can help developers quickly identify where a program is blocked, what tasks are pending, and how coroutines are chained together.

For example given this code:

```
import asyncio

async def play_track(track):
    await asyncio.sleep(5)
    print(f'🎵 Finished: {track}')

async def play_album(name, tracks):
    async with asyncio.TaskGroup() as tg:
        for track in tracks:
            tg.create_task(play_track(track), name=track)

async def main():
    async with asyncio.TaskGroup() as tg:
        tg.create_task(
          play_album('Sundowning', ['TNDNBTG', 'Levitate']),
          name='Sundowning')
        tg.create_task(
          play_album('TMBTE', ['DYWTYLM', 'Aqua Regia']),
          name='TMBTE')

if __name__ == '__main__':
    asyncio.run(main())
```

Executing the new tool on the running process will yield a table like this:

```
python -m asyncio ps 12345

tid        task id              task name            coroutine
stack                                    awaiter chain
awaiter name    awaiter id
---------------------------------------------------------------
---------------------------------------------------------------
------------------------------------------------------
1935500    0x7fc930c18050       Task-1
TaskGroup._aexit -> TaskGroup.__aexit__ -> main
0x0
```

```
1935500    0x7fc930c18230       Sundowning
TaskGroup._aexit -> TaskGroup.__aexit__ -> album
TaskGroup._aexit -> TaskGroup.__aexit__ -> main    Task-1
0x7fc930c18050
1935500    0x7fc93173fa50       TMBTE
TaskGroup._aexit -> TaskGroup.__aexit__ -> album
TaskGroup._aexit -> TaskGroup.__aexit__ -> main    Task-1
0x7fc930c18050
1935500    0x7fc93173fdf0       TNDNBTG              sleep ->
play                                     TaskGroup._aexit ->
TaskGroup.__aexit__ -> album   Sundowning     0x7fc930c18230
1935500    0x7fc930d32510       Levitate             sleep ->
play                                     TaskGroup._aexit ->
TaskGroup.__aexit__ -> album   Sundowning     0x7fc930c18230
1935500    0x7fc930d32890       DYWTYLM              sleep ->
play                                     TaskGroup._aexit ->
TaskGroup.__aexit__ -> album   TMBTE          0x7fc93173fa50
1935500    0x7fc93161ec30       Aqua Regia           sleep ->
play                                     TaskGroup._aexit ->
TaskGroup.__aexit__ -> album   TMBTE          0x7fc93173fa50
```

or a tree like this:

```
python -m asyncio pstree 12345

└── (T) Task-1
    └──  main example.py:13
        └──  TaskGroup.__aexit__ Lib/asyncio/taskgroups.py:72
            └──  TaskGroup._aexit Lib/asyncio/taskgroups.py:121
                ├── (T) Sundowning
                │   └──  album example.py:8
                │       └──  TaskGroup.__aexit__
Lib/asyncio/taskgroups.py:72
                │           └──  TaskGroup._aexit
Lib/asyncio/taskgroups.py:121
                │               ├── (T) TNDNBTG
                │               │   └──  play example.py:4
                │               │       └──  sleep
Lib/asyncio/tasks.py:702
                │               └── (T) Levitate
                │                   └──  play example.py:4
                │                       └──  sleep
Lib/asyncio/tasks.py:702
```

```
                └── (T) TMBTE
                    └──  album example.py:8
                        └──  TaskGroup.__aexit__
Lib/asyncio/taskgroups.py:72
                            └──  TaskGroup._aexit
Lib/asyncio/taskgroups.py:121
                                ├── (T) DYWTYLM
                                │   └──  play example.py:4
                                │       └──  sleep
Lib/asyncio/tasks.py:702
                                └── (T) Aqua Regia
                                    └──  play example.py:4
                                        └──  sleep
Lib/asyncio/tasks.py:702
```

If a cycle is detected in the async await graph (which could indicate a programming issue), the tool raises an error and lists the cycle paths that prevent tree construction:

```
python -m asyncio pstree 12345

ERROR: await-graph contains cycles - cannot print a tree!

cycle: Task-2 → Task-3 → Task-2
```

(Contributed by Pablo Galindo, Łukasz Langa, Yury Selivanov, and Marta Gomez Macias in gh-91048 [https://github.com/python/cpython/issues/91048].)

Concurrent safe warnings control

The **`warnings.catch_warnings`** context manager will now optionally use a context variable for warning filters. This is enabled by setting the **`context_aware_warnings`** flag, either with the `-X` command-line option or an environment variable. This gives predictable warnings control when using **`catch_warnings`** combined with multiple threads or asynchronous tasks. The flag defaults to true for the free-threaded build and false for the GIL-enabled build.

(Contributed by Neil Schemenauer and Kumar Aditya in gh-130010 [https://github.com/python/cpython/issues/130010].)

Other language changes

- All Windows code pages are now supported as 'cpXXX' codecs on Windows. (Contributed by Serhiy Storchaka in gh-123803 [https://github.com/python/cpython/issues/123803].)
- Implement mixed-mode arithmetic rules combining real and complex numbers as specified by the C standard since C99. (Contributed by Sergey B Kirpichev in gh-69639 [https://github.com/python/cpython/issues/69639].)
- More syntax errors are now detected regardless of optimisation and the **`-O`** command-line option. This includes writes to `__debug__`, incorrect use of **`await`**, and asynchronous comprehensions outside asynchronous functions. For example, `python -O -c 'assert (__debug__ := 1)'` or `python -O -c 'assert await 1'` now produce **`SyntaxError`**s. (Contributed by Irit Katriel and Jelle Zijlstra in gh-122245 [https://github.com/python/cpython/issues/122245] & gh-121637 [https://github.com/python/cpython/issues/121637].)
- When subclassing a pure C type, the C slots for the new type are no longer replaced with a wrapped version on class creation if they are not explicitly overridden in the subclass. (Contributed by Tomasz Pytel in gh-132284 [https://github.com/python/cpython/issues/132284].)

Built-ins

- The **`bytes.fromhex()`** and **`bytearray.fromhex()`** methods now accept ASCII **`bytes`** and bytes-like objects. (Contributed by Daniel Pope in gh-129349 [https://github.com/python/cpython/issues/129349].)
- Add class methods **`float.from_number()`** and **`complex.from_number()`** to convert a number to **`float`** or

`complex` type correspondingly. They raise a **`TypeError`** if the argument is not a real number. (Contributed by Serhiy Storchaka in gh-84978 [https://github.com/python/cpython/issues/84978].)

- Support underscore and comma as thousands separators in the fractional part for floating-point presentation types of the new-style string formatting (with **`format()`** or f-strings). (Contributed by Sergey B Kirpichev in gh-87790 [https://github.com/python/cpython/issues/87790].)
- The **`int()`** function no longer delegates to **`__trunc__()`**. Classes that want to support conversion to **`int()`** must implement either **`__int__()`** or **`__index__()`**. (Contributed by Mark Dickinson in gh-119743 [https://github.com/python/cpython/issues/119743].)
- The **`map()`** function now has an optional keyword-only *strict* flag like **`zip()`** to check that all the iterables are of equal length. (Contributed by Wannes Boeykens in gh-119793 [https://github.com/python/cpython/issues/119793].)
- The **`memoryview`** type now supports subscription, making it a generic type. (Contributed by Brian Schubert in gh-126012 [https://github.com/python/cpython/issues/126012].)
- Using **`NotImplemented`** in a boolean context will now raise a **`TypeError`**. This has raised a **`DeprecationWarning`** since Python 3.9. (Contributed by Jelle Zijlstra in gh-118767 [https://github.com/python/cpython/issues/118767].)
- Three-argument **`pow()`** now tries calling **`__rpow__()`** if necessary. Previously it was only called in two-argument **`pow()`** and the binary power operator. (Contributed by Serhiy Storchaka in gh-130104 [https://github.com/python/cpython/issues/130104].)
- **`super`** objects are now **`copyable`** and **`pickleable`**. (Contributed by Serhiy Storchaka in gh-125767 [https://github.com/python/cpython/issues/125767].)

Command line and environment

- The import time flag can now track modules that are already loaded (‘cached’), via the new **`-X importtime=2`**. When such a module is imported, the `self` and `cumulative` times are replaced by the string `cached`.

  Values above `2` for `-X importtime` are now reserved for future use.

  (Contributed by Noah Kim and Adam Turner in gh-118655 [https://github.com/python/cpython/issues/118655].)
- The command-line option **`-c`** now automatically dedents its code argument before execution. The auto-dedentation behavior mirrors **`textwrap.dedent()`**. (Contributed by Jon Crall and Steven Sun in gh-103998 [https://github.com/python/cpython/issues/103998].)
- **`-J`** is no longer a reserved flag for Jython [https://www.jython.org/], and now has no special meaning. (Contributed by Adam Turner in gh-133336 [https://github.com/python/cpython/issues/133336].)

PEP 758: Allow `except` and `except*` expressions without brackets

The **`except`** and **`except*`** expressions now allow brackets to be omitted when there are multiple exception types and the `as` clause is not used. For example:

```
try:
    connect_to_server()
except TimeoutError, ConnectionRefusedError:
    print('The network has ceased to be!')
```

(Contributed by Pablo Galindo and Brett Cannon in **PEP 758** [https://peps.python.org/pep-0758/] and gh-131831 [https://github.com/python/cpython/issues/131831].)

PEP 765: Control flow in `finally` blocks

The compiler now emits a `SyntaxWarning` when a `return`, `break`, or `continue` statement have the effect of leaving a `finally` block. This change is specified in **PEP 765** [https://peps.python.org/pep-0765/].

In situations where this change is inconvenient (such as those where the warnings are redundant due to code linting), the warning filter can be used to turn off all syntax warnings by adding `ignore::SyntaxWarning` as a filter. This can be specified in combination with a filter that converts other warnings to errors (for example, passing `-Werror -Wignore::SyntaxWarning` as CLI options, or setting `PYTHONWARNINGS=error,ignore::SyntaxWarning`).

Note that applying such a filter at runtime using the `warnings` module will only suppress the warning in code that is compiled *after* the filter is adjusted. Code that is compiled prior to the filter adjustment (for example, when a module is imported) will still emit the syntax warning.

(Contributed by Irit Katriel in gh-130080 [https://github.com/python/cpython/issues/130080].)

Incremental garbage collection

The cycle garbage collector is now incremental. This means that maximum pause times are reduced by an order of magnitude or more for larger heaps.

There are now only two generations: young and old. When **`gc.collect()`** is not called directly, the GC is invoked a little less frequently. When invoked, it collects the young generation and an increment of the old generation, instead of collecting one or more generations.

The behavior of **`gc.collect()`** changes slightly:

- `gc.collect(1)`: Performs an increment of garbage collection, rather than collecting generation 1.
- Other calls to **`gc.collect()`** are unchanged.

(Contributed by Mark Shannon in gh-108362 [https://github.com/python/cpython/issues/108362].)

Default interactive shell

- The default interactive shell now highlights Python syntax. The feature is enabled by default, save if **`PYTHON_BASIC_REPL`** or any other environment variable that disables colour is set. See Controlling color for details.

  The default color theme for syntax highlighting strives for good contrast and exclusively uses the 4-bit VGA standard ANSI color codes for maximum compatibility. The theme can be customized using an experimental API **`_colorize.set_theme()`**. This can be called interactively or in the **`PYTHONSTARTUP`** script. Note that this function has no stability guarantees, and may change or be removed.

  (Contributed by Łukasz Langa in gh-131507 [https://github.com/python/cpython/issues/131507].)

- The default interactive shell now supports import auto-completion. This means that typing `import co` and pressing `<Tab>` will suggest modules starting with `co`. Similarly, typing `from concurrent import i` will suggest submodules of `concurrent` starting with `i`. Note that autocompletion of module attributes is not currently supported. (Contributed by Tomas Roun in gh-69605 [https://github.com/python/cpython/issues/69605].)

New modules

- **`annotationlib`**: For introspecting annotations. See PEP 749 for more details. (Contributed by Jelle Zijlstra in gh-119180 [https://github.com/python/cpython/issues/119180].)
- **`compression`** (including **`compression.zstd`**): A package for compression-related modules, including a new module to support the Zstandard compression format. See PEP 784 for more details. (Contributed by Emma Harper Smith, Adam Turner, Gregory P. Smith, Tomas Roun, Victor Stinner, and Rogdham in gh-132983 [https://github.com/python/cpython/issues/132983].)
- **`concurrent.interpreters`**: Support for multiple interpreters in the standard library. See PEP 734 for more details. (Contributed by Eric Snow in gh-134939 [https://github.com/python/cpython/issues/134939].)
- **`string.templatelib`**: Support for template string literals (t-strings). See PEP 750 for more details. (Contributed by Jim Baker, Guido van Rossum, Paul Everitt, Koudai Aono, Lysandros Nikolaou, Dave Peck, Adam Turner, Jelle Zijlstra, Bénédikt Tran, and Pablo Galindo Salgado in gh-132661 [https://github.com/python/cpython/issues/132661].)

Improved modules

argparse

- The default value of the program name for **`argparse.ArgumentParser`** now reflects the way the Python interpreter was instructed to find the `__main__` module code. (Contributed by Serhiy Storchaka and Alyssa Coghlan in gh-66436 [https://github.com/python/cpython/issues/66436].)

- Introduced the optional *suggest_on_error* parameter to `argparse.ArgumentParser`, enabling suggestions for argument choices and subparser names if mistyped by the user. (Contributed by Savannah Ostrowski in gh-124456 [https://github.com/python/cpython/issues/124456].)
- Enable color for help text, which can be disabled with the optional *color* parameter to `argparse.ArgumentParser`. This can also be controlled by environment variables. (Contributed by Hugo van Kemenade in gh-130645 [https://github.com/python/cpython/issues/130645].)

ast

- Add `compare()`, a function for comparing two ASTs. (Contributed by Batuhan Taskaya and Jeremy Hylton in gh-60191 [https://github.com/python/cpython/issues/60191].)
- Add support for `copy.replace()` for AST nodes. (Contributed by Bénédikt Tran in gh-121141 [https://github.com/python/cpython/issues/121141].)
- Docstrings are now removed from an optimized AST in optimization level 2. (Contributed by Irit Katriel in gh-123958 [https://github.com/python/cpython/issues/123958].)
- The `repr()` output for AST nodes now includes more information. (Contributed by Tomas Roun in gh-116022 [https://github.com/python/cpython/issues/116022].)
- When called with an AST as input, the `parse()` function now always verifies that the root node type is appropriate. (Contributed by Irit Katriel in gh-130139 [https://github.com/python/cpython/issues/130139].)
- Add new options to the command-line interface: `--feature-version`, `--optimize`, and `--show-empty`. (Contributed by Semyon Moroz in gh-133367 [https://github.com/python/cpython/issues/133367].)

asyncio

- The function and methods named **`create_task()`** now take an arbitrary list of keyword arguments. All keyword arguments are passed to the **`Task`** constructor or the custom task factory. (See **`set_task_factory()`** for details.) The `name` and `context` keyword arguments are no longer special; the name should now be set using the `name` keyword argument of the factory, and `context` may be `None`.

  This affects the following function and methods: **`asyncio.create_task()`**, **`asyncio.loop.create_task()`**, **`asyncio.TaskGroup.create_task()`**.

  (Contributed by Thomas Grainger in gh-128307 [https://github.com/python/cpython/issues/128307].)

- There are two new utility functions for introspecting and printing a program's call graph: **`capture_call_graph()`** and **`print_call_graph()`**. See Asyncio introspection capabilities for more details. (Contributed by Yury Selivanov, Pablo Galindo Salgado, and Łukasz Langa in gh-91048 [https://github.com/python/cpython/issues/91048].)

calendar

- By default, today's date is highlighted in color in **`calendar`**'s command-line text output. This can be controlled by environment variables. (Contributed by Hugo van Kemenade in gh-128317 [https://github.com/python/cpython/issues/128317].)

concurrent.futures

- Add a new executor class, `InterpreterPoolExecutor`, which exposes multiple Python interpreters in the same process ('subinterpreters') to Python code. This uses a pool of independent Python interpreters to execute calls asynchronously.

  This is separate from the new `interpreters` module introduced by PEP 734. (Contributed by Eric Snow in gh-124548 [https://github.com/python/cpython/issues/124548].)

- On Unix platforms other than macOS, 'forkserver' is now the default start method for `ProcessPoolExecutor` (replacing 'fork'). This change does not affect Windows or macOS, where 'spawn' remains the default start method.

  If the threading incompatible *fork* method is required, you must explicitly request it by supplying a multiprocessing context *mp_context* to `ProcessPoolExecutor`.

  See forkserver restrictions for information and differences with the *fork* method and how this change may affect existing code with mutable global shared variables and/or shared objects that can not be automatically `pickled`.

  (Contributed by Gregory P. Smith in gh-84559 [https://github.com/python/cpython/issues/84559].)

- Add two new methods to `ProcessPoolExecutor`, `terminate_workers()` and `kill_workers()`, as ways to terminate or kill all living worker processes in the given pool. (Contributed by Charles Machalow in gh-130849 [https://github.com/python/cpython/issues/130849].)

- Add the optional *buffersize* parameter to **`Executor.map`** to limit the number of submitted tasks whose results have not yet been yielded. If the buffer is full, iteration over the *iterables* pauses until a result is yielded from the buffer. (Contributed by Enzo Bonnal and Josh Rosenberg in gh-74028 [https://github.com/python/cpython/issues/74028].)

configparser

- **`configparser`** will no longer write config files it cannot read, to improve security. Attempting to **`write()`** keys containing delimiters or beginning with the section header pattern will raise an **`InvalidWriteError`**. (Contributed by Jacob Lincoln in gh-129270 [https://github.com/python/cpython/issues/129270].)

contextvars

- Support the context manager protocol for **`Token`** objects. (Contributed by Andrew Svetlov in gh-129889 [https://github.com/python/cpython/issues/129889].)

ctypes

- The layout of bit fields in **`Structure`** and **`Union`** objects is now a closer match to platform defaults (GCC/Clang or MSVC). In particular, fields no longer overlap. (Contributed by Matthias Görgens in gh-97702 [https://github.com/python/cpython/issues/97702].)
- The **`Structure._layout_`** class attribute can now be set to help match a non-default ABI. (Contributed by Petr Viktorin in gh-97702 [https://github.com/python/cpython/issues/97702].)
- The class of **`Structure`**/**`Union`** field descriptors is now available as **`CField`**, and has new attributes to aid debugging and introspection. (Contributed by Petr

Viktorin in gh-128715 [https://github.com/python/cpython/issues/128715].)

- On Windows, the `COMError` exception is now public. (Contributed by Jun Komoda in gh-126686 [https://github.com/python/cpython/issues/126686].)
- On Windows, the `CopyComPointer()` function is now public. (Contributed by Jun Komoda in gh-127275 [https://github.com/python/cpython/issues/127275].)
- Add `memoryview_at()`, a function to create a `memoryview` object that refers to the supplied pointer and length. This works like `ctypes.string_at()` except it avoids a buffer copy, and is typically useful when implementing pure Python callback functions that are passed dynamically-sized buffers. (Contributed by Rian Hunter in gh-112018 [https://github.com/python/cpython/issues/112018].)
- Complex types, `c_float_complex`, `c_double_complex`, and `c_longdouble_complex`, are now available if both the compiler and the `libffi` library support complex C types. (Contributed by Sergey B Kirpichev in gh-61103 [https://github.com/python/cpython/issues/61103].)
- Add `ctypes.util.dllist()` for listing the shared libraries loaded by the current process. (Contributed by Brian Ward in gh-119349 [https://github.com/python/cpython/issues/119349].)
- Move `ctypes.POINTER()` types cache from a global internal cache (`_pointer_type_cache`) to the `_CData.__pointer_type__` attribute of the corresponding **`ctypes`** types. This will stop the cache from growing without limits in some situations. (Contributed by Sergey Miryanov in gh-100926 [https://github.com/python/cpython/issues/100926].)
- The `py_object` type now supports subscription, making it a generic type. (Contributed by Brian Schubert in gh-132168 [https://github.com/python/cpython/issues/132168].)
- **`ctypes`** now supports free-threading builds. (Contributed by Kumar Aditya and Peter Bierma in gh-127945 [https://github.com/python/cpython/issues/127945].)

curses

- Add the `assume_default_colors()` function, a refinement of the `use_default_colors()` function which allows changing the color pair `0`. (Contributed by Serhiy Storchaka in gh-133139 [https://github.com/python/cpython/issues/133139].)

datetime

- Add the `strptime()` method to the `datetime.date` and `datetime.time` classes. (Contributed by Wannes Boeykens in gh-41431 [https://github.com/python/cpython/issues/41431].)

decimal

- Add `Decimal.from_number()` as an alternative constructor for `Decimal`. (Contributed by Serhiy Storchaka in gh-121798 [https://github.com/python/cpython/issues/121798].)
- Expose `IEEEContext()` to support creation of contexts corresponding to the IEEE 754 (2008) decimal interchange formats. (Contributed by Sergey B Kirpichev in gh-53032 [https://github.com/python/cpython/issues/53032].)

difflib

- Comparison pages with highlighted changes generated by the `HtmlDiff` class now support 'dark mode'. (Contributed by Jiahao Li in gh-129939 [https://github.com/python/cpython/issues/129939].)

dis

- Add support for rendering full source location information of `instructions`, rather than only the line number. This feature is added to the following interfaces via the *show_positions* keyword argument:

  - `dis.Bytecode`
  - `dis.dis()`
  - `dis.distb()`
  - `dis.disassemble()`

  This feature is also exposed via `dis --show-positions`. (Contributed by Bénédikt Tran in gh-123165 [https://github.com/python/cpython/issues/123165].)

- Add the `dis --specialized` command-line option to show specialized bytecode. (Contributed by Bénédikt Tran in gh-127413 [https://github.com/python/cpython/issues/127413].)

errno

- Add the `EHWPOISON` error code constant. (Contributed by James Roy in gh-126585 [https://github.com/python/cpython/issues/126585].)

faulthandler

- Add support for printing the C stack trace on systems that support it via the new `dump_c_stack()` function or via the *c_stack* argument in `faulthandler.enable()`. (Contributed by Peter Bierma in gh-127604 [https://github.com/python/cpython/issues/127604].)

fnmatch

- Add `filterfalse()`, a function to reject names matching a given pattern. (Contributed by Bénédikt Tran in gh-74598 [https://github.com/python/cpython/issues/74598].)

fractions

- A `Fraction` object may now be constructed from any object with the `as_integer_ratio()` method. (Contributed by Serhiy Storchaka in gh-82017 [https://github.com/python/cpython/issues/82017].)
- Add `Fraction.from_number()` as an alternative constructor for `Fraction`. (Contributed by Serhiy Storchaka in gh-121797 [https://github.com/python/cpython/issues/121797].)

functools

- Add the `Placeholder` sentinel. This may be used with the `partial()` or `partialmethod()` functions to reserve a place for positional arguments in the returned partial object. (Contributed by Dominykas Grigonis in gh-119127 [https://github.com/python/cpython/issues/119127].)
- Allow the *initial* parameter of `reduce()` to be passed as a keyword argument. (Contributed by Sayandip Dutta in gh-125916 [https://github.com/python/cpython/issues/125916].)

getopt

- Add support for options with optional arguments. (Contributed by Serhiy Storchaka in gh-126374 [https://github.com/python/cpython/issues/126374].)
- Add support for returning intermixed options and non-option arguments in order. (Contributed by Serhiy Storchaka in gh-126390 [https://github.com/python/cpython/issues/126390].)

getpass

- Support keyboard feedback in the `getpass()` function via the keyword-only optional argument *echo_char*.

Placeholder characters are rendered whenever a character is entered, and removed when a character is deleted. (Contributed by Semyon Moroz in gh-77065 [https://github.com/python/cpython/issues/77065].)

graphlib

- Allow `TopologicalSorter.prepare()` to be called more than once as long as sorting has not started. (Contributed by Daniel Pope in gh-130914 [https://github.com/python/cpython/issues/130914].)

heapq

- The **`heapq`** module has improved support for working with max-heaps, via the following new functions:
  - `heapify_max()`
  - `heappush_max()`
  - `heappop_max()`
  - `heapreplace_max()`
  - `heappushpop_max()`

hmac

- Add a built-in implementation for HMAC (**RFC 2104** [https://datatracker.ietf.org/doc/html/rfc2104.html]) using formally verified code from the HACL* [https://github.com/hacl-star/hacl-star/] project. This implementation is used as a fallback when the OpenSSL implementation of HMAC is not available. (Contributed by Bénédikt Tran in gh-99108 [https://github.com/python/cpython/issues/99108].)

http

- Directory lists and error pages generated by the `http.server` module allow the browser to apply its default dark mode. (Contributed by Yorik Hansen in gh-123430 [https://github.com/python/cpython/issues/123430].)

- The **`http.server`** module now supports serving over HTTPS using the **`http.server.HTTPSServer`** class. This functionality is exposed by the command-line interface (`python -m http.server`) through the following options:

  - **`--tls-cert <path>`**: Path to the TLS certificate file.
  - **`--tls-key <path>`**: Optional path to the private key file.
  - **`--tls-password-file <path>`**: Optional path to the password file for the private key.

  (Contributed by Semyon Moroz in gh-85162 [https://github.com/python/cpython/issues/85162].)

imaplib

- Add **`IMAP4.idle()`**, implementing the IMAP4 `IDLE` command as defined in **RFC 2177** [https://datatracker.ietf.org/doc/html/rfc2177.html]. (Contributed by Forest in gh-55454 [https://github.com/python/cpython/issues/55454].)

inspect

- **`signature()`** takes a new argument *annotation_format* to control the **`annotationlib.Format`** used for representing annotations. (Contributed by Jelle Zijlstra in gh-101552 [https://github.com/python/cpython/issues/101552].)
- **`Signature.format()`** takes a new argument *unquote_annotations*. If true, string annotations are displayed without surrounding quotes. (Contributed by Jelle Zijlstra in gh-101552 [https://github.com/python/cpython/issues/101552].)

- Add function `ispackage()` to determine whether an object is a package or not. (Contributed by Zhikang Yan in gh-125634 [https://github.com/python/cpython/issues/125634].)

io

- Reading text from a non-blocking stream with `read` may now raise a `BlockingIOError` if the operation cannot immediately return bytes. (Contributed by Giovanni Siragusa in gh-109523 [https://github.com/python/cpython/issues/109523].)
- Add the `Reader` and `Writer` protocols as simpler alternatives to the pseudo-protocols `typing.IO`, `typing.TextIO`, and `typing.BinaryIO`. (Contributed by Sebastian Rittau in gh-127648 [https://github.com/python/cpython/issues/127648].)

json

- Add exception notes for JSON serialization errors that allow identifying the source of the error. (Contributed by Serhiy Storchaka in gh-122163 [https://github.com/python/cpython/issues/122163].)
- Allow using the `json` module as a script using the `-m` switch: **python -m json**. This is now preferred to **python -m json.tool**, which is soft deprecated. See the JSON command-line interface documentation. (Contributed by Trey Hunner in gh-122873 [https://github.com/python/cpython/issues/122873].)
- By default, the output of the JSON command-line interface is highlighted in color. This can be controlled by environment variables. (Contributed by Tomas Roun in gh-131952 [https://github.com/python/cpython/issues/131952].)

linecache

- `getline()` can now retrieve source code for frozen modules. (Contributed by Tian Gao in gh-131638 [https://github.com/python/cpython/issues/131638].)

logging.handlers

- `QueueListener` objects now support the context manager protocol. (Contributed by Charles Machalow in gh-132106 [https://github.com/python/cpython/issues/132106].)
- `QueueListener.start` now raises a `RuntimeError` if the listener is already started. (Contributed by Charles Machalow in gh-132106 [https://github.com/python/cpython/issues/132106].)

math

- Added more detailed error messages for domain errors in the module. (Contributed by Charlie Zhao and Sergey B Kirpichev in gh-101410 [https://github.com/python/cpython/issues/101410].)

mimetypes

- Add a public command-line for the module, invoked via **python -m mimetypes**. (Contributed by Oleg Iarygin and Hugo van Kemenade in gh-93096 [https://github.com/python/cpython/issues/93096].)

- Add several new MIME types based on RFCs and common usage:

  Microsoft and RFC 8081 [https://datatracker.ietf.org/doc/html/rfc8081.html] MIME types for fonts

  - Embedded OpenType: `application/vnd.ms-fontobject`

- OpenType Layout (OTF) `font/otf`
- TrueType: `font/ttf`
- WOFF 1.0 `font/woff`
- WOFF 2.0 `font/woff2`

RFC 9559 [https://datatracker.ietf.org/doc/html/rfc9559.html] MIME types for Matroska audiovisual data container structures

- audio with no video: `audio/matroska` (`.mka`)
- video: `video/matroska` (`.mkv`)
- stereoscopic video: `video/matroska-3d` (`.mk3d`)

Images with RFCs

- **RFC 1494** [https://datatracker.ietf.org/doc/html/rfc1494.html]: CCITT Group 3 (`.g3`)
- **RFC 3362** [https://datatracker.ietf.org/doc/html/rfc3362.html]: Real-time Facsimile, T.38 (`.t38`)
- **RFC 3745** [https://datatracker.ietf.org/doc/html/rfc3745.html]: JPEG 2000 (`.jp2`), extension (`.jpx`) and compound (`.jpm`)
- **RFC 3950** [https://datatracker.ietf.org/doc/html/rfc3950.html]: Tag Image File Format Fax eXtended, TIFF-FX (`.tfx`)
- **RFC 4047** [https://datatracker.ietf.org/doc/html/rfc4047.html]: Flexible Image Transport System (`.fits`)
- **RFC 7903** [https://datatracker.ietf.org/doc/html/rfc7903.html]: Enhanced Metafile (`.emf`) and Windows Metafile (`.wmf`)

Other MIME type additions and changes

- **RFC 2361** [https://datatracker.ietf.org/doc/html/rfc2361.html]: Change type for `.avi` to `video/vnd.avi` and for `.wav` to `audio/vnd.wave`

  - **RFC 4337** [https://datatracker.ietf.org/doc/html/rfc4337.html]: Add MPEG-4 `audio/mp4` (`.m4a`)
  - **RFC 5334** [https://datatracker.ietf.org/doc/html/rfc5334.html]: Add Ogg media (`.oga`, `.ogg` and `.ogx`)
  - **RFC 6713** [https://datatracker.ietf.org/doc/html/rfc6713.html]: Add gzip `application/gzip` (`.gz`)
  - **RFC 9639** [https://datatracker.ietf.org/doc/html/rfc9639.html]: Add FLAC `audio/flac` (`.flac`)
  - **RFC 9512** [https://datatracker.ietf.org/doc/html/rfc9512.html] `application/yaml` MIME type for YAML files (`.yaml` and `.yml`)
  - Add 7z `application/x-7z-compressed` (`.7z`)
  - Add Android Package `application/vnd.android.package-archive` (`.apk`) when not strict
  - Add deb `application/x-debian-package` (`.deb`)
  - Add glTF binary `model/gltf-binary` (`.glb`)
  - Add glTF JSON/ASCII `model/gltf+json` (`.gltf`)
  - Add M4V `video/x-m4v` (`.m4v`)
  - Add PHP `application/x-httpd-php` (`.php`)
  - Add RAR `application/vnd.rar` (`.rar`)
  - Add RPM `application/x-rpm` (`.rpm`)
  - Add STL `model/stl` (`.stl`)
  - Add Windows Media Video `video/x-ms-wmv` (`.wmv`)
  - De facto: Add WebM `audio/webm` (`.weba`)
  - ECMA-376 [https://ecma-international.org/publications-and-standards/standards/ecma-376/]: Add `.docx`, `.pptx` and `.xlsx` types
  - OASIS [https://docs.oasis-open.org/office/v1.2/cs01/OpenDocument-v1.2-cs01-part1.html#Appendix_C]: Add OpenDocument `.odg`, `.odp`, `.ods` and `.odt` types
  - W3C [https://www.w3.org/TR/epub-33/#app-media-type]: Add EPUB `application/epub+zip` (`.epub`)

(Contributed by Sahil Prajapati and Hugo van Kemenade in gh-84852 [https://github.com/python/cpython/issues/84852], by Sasha “Nelie” Chernykh and Hugo van Kemenade in gh-

132056 [https://github.com/python/cpython/issues/132056], and by Hugo van Kemenade in gh-89416 [https://github.com/python/cpython/issues/89416], gh-85957 [https://github.com/python/cpython/issues/85957], and gh-129965 [https://github.com/python/cpython/issues/129965].)

multiprocessing

- On Unix platforms other than macOS, ‘forkserver’ is now the default start method (replacing ‘fork’). This change does not affect Windows or macOS, where ‘spawn’ remains the default start method.

  If the threading incompatible *fork* method is required, you must explicitly request it via a context from **`get_context()`** (preferred) or change the default via **`set_start_method()`**.

  See forkserver restrictions for information and differences with the *fork* method and how this change may affect existing code with mutable global shared variables and/or shared objects that can not be automatically **`pickled`**.

  (Contributed by Gregory P. Smith in gh-84559 [https://github.com/python/cpython/issues/84559].)

- **`multiprocessing`**’s `'forkserver'` start method now authenticates its control socket to avoid solely relying on filesystem permissions to restrict what other processes could cause the forkserver to spawn workers and run code. (Contributed by Gregory P. Smith for gh-97514 [https://github.com/python/cpython/issues/97514].)

- The multiprocessing proxy objects for *list* and *dict* types gain previously overlooked missing methods:

  - **`clear()`** and **`copy()`** for proxies of **`list`**
  - **`fromkeys()`**, `reversed(d)`, `d | {}`, `{} | d`, `d |= {'b': 2}` for proxies of **`dict`**

  (Contributed by Roy Hyunjin Han for gh-103134 [https://github.com/python/cpython/issues/103134].)

- Add support for shared **`set`** objects via **`SyncManager.set()`**. The **`set()`** in **`Manager()`** method is now available. (Contributed by Mingyu Park in gh-129949 [https://github.com/python/cpython/issues/129949].)
- Add the **`interrupt()`** to **`multiprocessing.Process`** objects, which terminates the child process by sending **`SIGINT`**. This enables **`finally`** clauses to print a stack trace for the terminated process. (Contributed by Artem Pulkin in gh-131913 [https://github.com/python/cpython/issues/131913].)

operator

- Add **`is_none()`** and **`is_not_none()`** as a pair of functions, such that `operator.is_none(obj)` is equivalent to `obj is None` and `operator.is_not_none(obj)` is equivalent to `obj is not None`. (Contributed by Raymond Hettinger and Nico Mexis in gh-115808 [https://github.com/python/cpython/issues/115808].)

os

- Add the **`reload_environ()`** function to update **`os.environ`** and **`os.environb`** with changes to the environment made by **`os.putenv()`**, by **`os.unsetenv()`**, or made outside Python in the same process. (Contributed by Victor Stinner in gh-120057 [https://github.com/python/cpython/issues/120057].)
- Add the **`SCHED_DEADLINE`** and **`SCHED_NORMAL`** constants to the **`os`** module. (Contributed by James Roy in gh-127688

[https://github.com/python/cpython/issues/127688].)

- Add the **`readinto()`** function to read into a buffer object from a file descriptor. (Contributed by Cody Maloney in gh-129205 [https://github.com/python/cpython/issues/129205].)

os.path

- The *strict* parameter to **`realpath()`** accepts a new value, **`ALLOW_MISSING`**. If used, errors other than **`FileNotFoundError`** will be re-raised; the resulting path can be missing but it will be free of symlinks. (Contributed by Petr Viktorin for **CVE 2025-4517** [https://www.cve.org/CVERecord?id=CVE-2025-4517].)

pathlib

- Add methods to **`pathlib.Path`** to recursively copy or move files and directories:
  - **`copy()`** copies a file or directory tree to a destination.
  - **`copy_into()`** copies *into* a destination directory.
  - **`move()`** moves a file or directory tree to a destination.
  - **`move_into()`** moves *into* a destination directory.

  (Contributed by Barney Gale in gh-73991 [https://github.com/python/cpython/issues/73991].)

- Add the **`info`** attribute, which stores an object implementing the new **`pathlib.types.PathInfo`** protocol. The object supports querying the file type and internally caching **`stat()`** results. Path objects generated by **`iterdir()`** are initialized with file type information gleaned from scanning the parent directory. (Contributed by Barney Gale in gh-125413 [https://github.com/python/cpython/issues/125413].)

pdb

- The **`pdb`** module now supports remote attaching to a running Python process using a new **`-p PID`** command-line option:

  ```
  python -m pdb -p 1234
  ```

  This will connect to the Python process with the given PID and allow you to debug it interactively. Notice that due to how the Python interpreter works attaching to a remote process that is blocked in a system call or waiting for I/O will only work once the next bytecode instruction is executed or when the process receives a signal.

  This feature uses PEP 768 and the new **`sys.remote_exec()`** function to attach to the remote process and send the PDB commands to it.

  (Contributed by Matt Wozniski and Pablo Galindo in gh-131591 [https://github.com/python/cpython/issues/131591].)

- Hardcoded breakpoints (**`breakpoint()`** and **`set_trace()`**) now reuse the most recent **`Pdb`** instance that calls **`set_trace()`**, instead of creating a new one each time. As a result, all the instance specific data like **`display`** and **`commands`** are preserved across hardcoded breakpoints. (Contributed by Tian Gao in gh-121450 [https://github.com/python/cpython/issues/121450].)

- Add a new argument *mode* to **`pdb.Pdb`**. Disable the `restart` command when **`pdb`** is in `inline` mode. (Contributed by Tian Gao in gh-123757 [https://github.com/python/cpython/issues/123757].)

- A confirmation prompt will be shown when the user tries to quit **`pdb`** in `inline` mode. `y`, `Y`, `<Enter>` or `EOF` will confirm the quit and call **`sys.exit()`**, instead of raising **`bdb.BdbQuit`**. (Contributed by Tian Gao in gh-124704 [https://github.com/python/cpython/issues/124704].)

- Inline breakpoints like **`breakpoint()`** or **`pdb.set_trace()`** will always stop the program at calling frame, ignoring the `skip` pattern (if any). (Contributed by Tian Gao in gh-130493 [https://github.com/python/cpython/issues/130493].)

- `<tab>` at the beginning of the line in **`pdb`** multi-line input will fill in a 4-space indentation now, instead of inserting a `\t` character. (Contributed by Tian Gao in gh-130471 [https://github.com/python/cpython/issues/130471].)

- Auto-indent is introduced in **`pdb`** multi-line input. It will either keep the indentation of the last line or insert a 4-space indentation when it detects a new code block. (Contributed by Tian Gao in gh-133350 [https://github.com/python/cpython/issues/133350].)

- `$_asynctask` is added to access the current asyncio task if applicable. (Contributed by Tian Gao in gh-124367 [https://github.com/python/cpython/issues/124367].)

- **`pdb.set_trace_async()`** is added to support debugging asyncio coroutines. **`await`** statements are supported with this function. (Contributed by Tian Gao in gh-132576 [https://github.com/python/cpython/issues/132576].)

- Source code displayed in **`pdb`** will be syntax-highlighted. This feature can be controlled using the same methods as the default interactive shell, in addition to the newly added `colorize` argument of **`pdb.Pdb`**. (Contributed by Tian Gao and Łukasz Langa in gh-133355 [https://github.com/python/cpython/issues/133355].)

pickle

- Set the default protocol version on the **`pickle`** module to 5. For more details, see pickle protocols.
- Add exception notes for pickle serialization errors that allow identifying the source of the error. (Contributed by Serhiy Storchaka in gh-122213 [https://github.com/python/cpython/issues/122213].)

platform

- Add **`invalidate_caches()`**, a function to invalidate cached results in the **`platform`** module. (Contributed by Bénédikt Tran in gh-122549 [https://github.com/python/cpython/issues/122549].)

pydoc

- Annotations in help output are now usually displayed in a format closer to that in the original source. (Contributed by Jelle Zijlstra in gh-101552 [https://github.com/python/cpython/issues/101552].)

re

- Support `\z` as a synonym for `\Z` in **`regular expressions`**. It is interpreted unambiguously in many other regular expression engines, unlike `\Z`, which has subtly different behavior. (Contributed by Serhiy Storchaka in gh-133306 [https://github.com/python/cpython/issues/133306].)
- `\B` in **`regular expression`** now matches the empty input string, meaning that it is now always the opposite of `\b`. (Contributed by Serhiy Storchaka in gh-124130 [https://github.com/python/cpython/issues/124130].)

socket

- Improve and fix support for Bluetooth sockets.
    - Fix support of Bluetooth sockets on NetBSD and DragonFly BSD. (Contributed by Serhiy Storchaka in gh-132429 [https://github.com/python/cpython/issues/132429].)
    - Fix support for **`BTPROTO_HCI`** on FreeBSD. (Contributed by Victor Stinner in gh-111178 [https://github.com/python/cpython/issues/111178].)
    - Add support for **`BTPROTO_SCO`** on FreeBSD. (Contributed by Serhiy Storchaka in gh-85302 [https://github.com/python/cpython/issues/85302].)
    - Add support for *cid* and *bdaddr_type* in the address for **`BTPROTO_L2CAP`** on FreeBSD. (Contributed by Serhiy Storchaka in gh-132429 [https://github.com/python/cpython/issues/132429].)
    - Add support for *channel* in the address for **`BTPROTO_HCI`** on Linux. (Contributed by Serhiy Storchaka in gh-70145 [https://github.com/python/cpython/issues/70145].)
    - Accept an integer as the address for **`BTPROTO_HCI`** on Linux. (Contributed by Serhiy Storchaka in gh-132099 [https://github.com/python/cpython/issues/132099].)
    - Return *cid* in **`getsockname()`** for **`BTPROTO_L2CAP`**. (Contributed by Serhiy Storchaka in gh-132429 [https://github.com/python/cpython/issues/132429].)
    - Add many new constants. (Contributed by Serhiy Storchaka in gh-132734 [https://github.com/python/cpython/issues/132734].)

ssl

- Indicate through the **`HAS_PHA`** Boolean whether the `ssl` module supports TLSv1.3 post-handshake client authentication (PHA). (Contributed by Will Childs-Klein in gh-128036 [https://github.com/python/cpython/issues/128036].)

struct

- Support the float complex and double complex C types in the **`struct`** module (formatting characters `'F'` and `'D'` respectively). (Contributed by Sergey B Kirpichev in gh-121249 [https://github.com/python/cpython/issues/121249].)

symtable

- Expose the following **`Symbol`** methods:
    - **`is_comp_cell()`**
    - **`is_comp_iter()`**
    - **`is_free_class()`**

  (Contributed by Bénédikt Tran in gh-120029 [https://github.com/python/cpython/issues/120029].)

sys

- The previously undocumented special function **`sys.getobjects()`**, which only exists in specialized builds of Python, may now return objects from other interpreters than the one it's called in. (Contributed by Eric Snow in gh-125286 [https://github.com/python/cpython/issues/125286].)
- Add **`sys._is_immortal()`** for determining if an object is immortal. (Contributed by Peter Bierma in gh-128509 [https://github.com/python/cpython/issues/128509].)
- On FreeBSD, **`sys.platform`** no longer contains the major version number. It is always `'freebsd'`, instead of `'freebsd13'` or `'freebsd14'`. (Contributed by Michael

Osipov in gh-129393 [https://github.com/python/cpython/issues/129393].)

- Raise `DeprecationWarning` for `sys._clear_type_cache()`. This function was deprecated in Python 3.13 but it didn’t raise a runtime warning.
- Add `sys.remote_exec()` to implement the new external debugger interface. See PEP 768 for details. (Contributed by Pablo Galindo Salgado, Matt Wozniski, and Ivona Stojanovic in gh-131591 [https://github.com/python/cpython/issues/131591].)
- Add the `sys._jit` namespace, containing utilities for introspecting just-in-time compilation. (Contributed by Brandt Bucher in gh-133231 [https://github.com/python/cpython/issues/133231].)

sys.monitoring

- Add two new monitoring events, `BRANCH_LEFT` and `BRANCH_RIGHT`. These replace and deprecate the **`BRANCH`** event. (Contributed by Mark Shannon in gh-122548 [https://github.com/python/cpython/issues/122548].)

sysconfig

- Add `ABIFLAGS` key to `get_config_vars()` on Windows. (Contributed by Xuehai Pan in gh-131799 [https://github.com/python/cpython/issues/131799].)

tarfile

- `data_filter()` now normalizes symbolic link targets in order to avoid path traversal attacks. (Contributed by Petr Viktorin in gh-127987 [https://github.com/python/cpython/issues/127987] and **CVE 2025-4138** [https://www.cve.org/CVERecord?id=CVE-2025-4138].)

- `extractall()` now skips fixing up directory attributes when a directory was removed or replaced by another kind of file. (Contributed by Petr Viktorin in gh-127987 [https://github.com/python/cpython/issues/127987] and **CVE 2024-12718** [https://www.cve.org/CVERecord?id=CVE-2024-12718].)
- `extract()` and `extractall()` now (re-)apply the extraction filter when substituting a link (hard or symbolic) with a copy of another archive member, and when fixing up directory attributes. The former raises a new exception, `LinkFallbackError`. (Contributed by Petr Viktorin for **CVE 2025-4330** [https://www.cve.org/CVERecord?id=CVE-2025-4330] and **CVE 2024-12718** [https://www.cve.org/CVERecord?id=CVE-2024-12718].)
- `extract()` and `extractall()` no longer extract rejected members when `errorlevel()` is zero. (Contributed by Matt Prodani and Petr Viktorin in gh-112887 [https://github.com/python/cpython/issues/112887] and **CVE 2025-4435** [https://www.cve.org/CVERecord?id=CVE-2025-4435].)

threading

- `threading.Thread.start()` now sets the operating system thread name to `threading.Thread.name`. (Contributed by Victor Stinner in gh-59705 [https://github.com/python/cpython/issues/59705].)

tkinter

- Make `tkinter` widget methods `after()` and `after_idle()` accept keyword arguments. (Contributed by Zhikang Yan in gh-126899 [https://github.com/python/cpython/issues/126899].)
- Add ability to specify a name for `tkinter.OptionMenu` and `tkinter.ttk.OptionMenu`. (Contributed by Zhikang Yan in gh-130482 [https://github.com/python/cpython/issues/130482].)

turtle

- Add context managers for `turtle.fill()`, `turtle.poly()`, and `turtle.no_animation()`. (Contributed by Marie Roald and Yngve Mardal Moe in gh-126350 [https://github.com/python/cpython/issues/126350].)

types

- `types.UnionType` is now an alias for `typing.Union`. See below for more details. (Contributed by Jelle Zijlstra in gh-105499 [https://github.com/python/cpython/issues/105499].)

typing

- The `types.UnionType` and `typing.Union` types are now aliases for each other, meaning that both old-style unions (created with `Union[int, str]`) and new-style unions (`int | str`) now create instances of the same runtime type. This unifies the behavior between the two syntaxes, but leads to some differences in behavior that may affect users who introspect types at runtime:
  - Both syntaxes for creating a union now produce the same string representation in `repr()`. For example, `repr(Union[int, str])` is now `"int | str"` instead of `"typing.Union[int, str]"`.
  - Unions created using the old syntax are no longer cached. Previously, running `Union[int, str]` multiple times would return the same object (`Union[int, str] is Union[int, str]` would be `True`), but now it will return two different objects. Use `==` to compare unions for equality, not `is`. New-style unions have never been cached this way. This change could increase memory usage for some programs that use a large number of unions created by subscripting

`typing.Union`. However, several factors offset this cost: unions used in annotations are no longer evaluated by default in Python 3.14 because of **PEP 649** [https://peps.python.org/pep-0649/]; an instance of `types.UnionType` is itself much smaller than the object returned by `Union[]` was on prior Python versions; and removing the cache also saves some space. It is therefore unlikely that this change will cause a significant increase in memory usage for most users.

  - Previously, old-style unions were implemented using the private class `typing._UnionGenericAlias`. This class is no longer needed for the implementation, but it has been retained for backward compatibility, with removal scheduled for Python 3.17. Users should use documented introspection helpers like `get_origin()` and `typing.get_args()` instead of relying on private implementation details.
  - It is now possible to use `typing.Union` itself in `isinstance()` checks. For example, `isinstance(int | str, typing.Union)` will return `True`; previously this raised `TypeError`.
  - The **`__args__`** attribute of `typing.Union` objects is no longer writable.
  - It is no longer possible to set any attributes on `Union` objects. This only ever worked for dunder attributes on previous versions, was never documented to work, and was subtly broken in many cases.

(Contributed by Jelle Zijlstra in gh-105499 [https://github.com/python/cpython/issues/105499].)

- `TypeAliasType` now supports star unpacking.

unicodedata

- The Unicode database has been updated to Unicode 16.0.0.

unittest

- `unittest` output is now colored by default. This can be controlled by environment variables. (Contributed by Hugo van Kemenade in gh-127221 [https://github.com/python/cpython/issues/127221].)

- unittest discovery supports namespace package as start directory again. It was removed in Python 3.11. (Contributed by Jacob Walls in gh-80958 [https://github.com/python/cpython/issues/80958].)

- A number of new methods were added in the `TestCase` class that provide more specialized tests.

  - `assertHasAttr()` and `assertNotHasAttr()` check whether the object has a particular attribute.
  - `assertIsSubclass()` and `assertNotIsSubclass()` check whether the object is a subclass of a particular class, or of one of a tuple of classes.
  - `assertStartsWith()`, `assertNotStartsWith()`, `assertEndsWith()` and `assertNotEndsWith()` check whether the Unicode or byte string starts or ends with particular strings.

  (Contributed by Serhiy Storchaka in gh-71339 [https://github.com/python/cpython/issues/71339].)

urllib

- Upgrade HTTP digest authentication algorithm for `urllib.request` by supporting SHA-256 digest authentication as specified in **RFC 7616** [https://datatracker.ietf.org/doc/html/rfc7616.html]. (Contributed by Calvin Bui in gh-128193 [https://github.com/python/cpython/issues/128193].)

- Improve ergonomics and standards compliance when parsing and emitting `file:` URLs.

  In **`url2pathname()`**:

  - Accept a complete URL when the new *require_scheme* argument is set to true.
  - Discard URL authority if it matches the local hostname.
  - Discard URL authority if it resolves to a local IP address when the new *resolve_host* argument is set to true.
  - Discard URL query and fragment components.
  - Raise **`URLError`** if a URL authority isn't local, except on Windows where we return a UNC path as before.

  In **`pathname2url()`**:

  - Return a complete URL when the new *add_scheme* argument is set to true.
  - Include an empty URL authority when a path begins with a slash. For example, the path `/etc/hosts` is converted to the URL `///etc/hosts`.

  On Windows, drive letters are no longer converted to uppercase, and `:` characters not following a drive letter no longer cause an **`OSError`** exception to be raised.

  (Contributed by Barney Gale in gh-125866 [https://github.com/python/cpython/issues/125866].)

uuid

- Add support for UUID versions 6, 7, and 8 via `uuid6()`, `uuid7()`, and `uuid8()` respectively, as specified in **RFC 9562** [https://datatracker.ietf.org/doc/html/rfc9562.html]. (Contributed by Bénédikt Tran in gh-89083 [https://github.com/python/cpython/issues/89083].)
- `NIL` and `MAX` are now available to represent the Nil and Max UUID formats as defined by **RFC 9562** [https://datatracker.ietf.org/doc/html/rfc9562.html]. (Contributed by Nick Pope in gh-128427 [https://github.com/python/cpython/issues/128427].)
- Allow generating multiple UUIDs simultaneously on the command-line via `python -m uuid --count`. (Contributed by Simon Legner in gh-131236 [https://github.com/python/cpython/issues/131236].)

webbrowser

- Names in the `BROWSER` environment variable can now refer to already registered browsers for the `webbrowser` module, instead of always generating a new browser command.

  This makes it possible to set `BROWSER` to the value of one of the supported browsers on macOS.

zipfile

- Added `ZipInfo._for_archive`, a method to resolve suitable defaults for a `ZipInfo` object as used by `ZipFile.writestr`. (Contributed by Bénédikt Tran in gh-123424 [https://github.com/python/cpython/issues/123424].)
- `ZipFile.writestr()` now respects the `SOURCE_DATE_EPOCH` environment variable in order to better support reproducible builds. (Contributed by Jiahao Li in gh-91279 [https://github.com/python/cpython/issues/91279].)

Optimizations

- The import time for several standard library modules has been improved, including `annotationlib`, `ast`, `asyncio`, `base64`, `cmd`, `csv`, `gettext`, `importlib.util`, `locale`, `mimetypes`, `optparse`, `pickle`, `pprint`, `pstats`, `shlex`, `socket`, `string`, `subprocess`, `threading`, `tomllib`, `types`, and `zipfile`.

  (Contributed by Adam Turner, Bénédikt Tran, Chris Markiewicz, Eli Schwartz, Hugo van Kemenade, Jelle Zijlstra, and others in gh-118761 [https://github.com/python/cpython/issues/118761].)

- The interpreter now avoids some reference count modifications internally when it's safe to do so. This can lead to different values being returned from `sys.getrefcount()` and `Py_REFCNT()` compared to previous versions of Python. See below for details.

asyncio

- Standard benchmark results have improved by 10-20% following the implementation of a new per-thread doubly linked list for `native tasks`, also reducing memory usage. This enables external introspection tools such as python -m asyncio pstree to introspect the call graph of asyncio tasks running in all threads. (Contributed by Kumar Aditya in gh-107803 [https://github.com/python/cpython/issues/107803].)
- The module now has first class support for free-threading builds. This enables parallel execution of multiple event loops across different threads, scaling linearly with the number of threads. (Contributed by Kumar Aditya in gh-128002 [https://github.com/python/cpython/issues/128002].)

base64

- `b16decode()` is now up to six times faster. (Contributed by Bénédikt Tran, Chris Markiewicz, and Adam Turner in gh-118761 [https://github.com/python/cpython/issues/118761].)

bdb

- The basic debugger now has a `sys.monitoring`-based backend, which can be selected via the passing `'monitoring'` to the `Bdb` class's new *backend* parameter. (Contributed by Tian Gao in gh-124533 [https://github.com/python/cpython/issues/124533].)

difflib

- The `IS_LINE_JUNK()` function is now up to twice as fast. (Contributed by Adam Turner and Semyon Moroz in gh-130167 [https://github.com/python/cpython/issues/130167].)

gc

- The new incremental garbage collector means that maximum pause times are reduced by an order of magnitude or more for larger heaps.

  Because of this optimization, the meaning of the results of `get_threshold()` and `set_threshold()` have changed, along with `get_count()` and `get_stats()`.

  - For backwards compatibility, `get_threshold()` continues to return a three-item tuple. The first value is the threshold for young collections, as before; the second value determines the rate at which the old collection is scanned (the default is 10, and higher values mean that the old collection is

  scanned more slowly). The third value is now meaningless and is always zero.
  - `set_threshold()` now ignores any items after the second.
  - `get_count()` and `get_stats()` continue to return the same format of results. The only difference is that instead of the results referring to the young, aging and old generations, the results refer to the young generation and the aging and collecting spaces of the old generation.

In summary, code that attempted to manipulate the behavior of the cycle GC may not work exactly as intended, but it is very unlikely to be harmful. All other code will work just fine.

(Contributed by Mark Shannon in gh-108362 [https://github.com/python/cpython/issues/108362].)

io

- Opening and reading files now executes fewer system calls. Reading a small operating system cached file in full is up to 15% faster. (Contributed by Cody Maloney and Victor Stinner in gh-120754 [https://github.com/python/cpython/issues/120754] and gh-90102 [https://github.com/python/cpython/issues/90102].)

pathlib

- `Path.read_bytes` now uses unbuffered mode to open files, which is between 9% and 17% faster to read in full. (Contributed by Cody Maloney in gh-120754 [https://github.com/python/cpython/issues/120754].)

pdb

- `pdb` now supports two backends, based on either `sys.settrace()` or `sys.monitoring`. Using the pdb CLI or `breakpoint()` will always use the `sys.monitoring` backend. Explicitly instantiating `pdb.Pdb` and its derived classes will use the `sys.settrace()` backend by default, which is configurable. (Contributed by Tian Gao in gh-124533 [https://github.com/python/cpython/issues/124533].)

textwrap

- Optimize the `dedent()` function, improving performance by an average of 2.4x, with larger improvements for bigger inputs, and fix a bug with incomplete normalization of blank lines with whitespace characters other than space and tab.

uuid

- `uuid3()` and `uuid5()` are now both roughly 40% faster for 16-byte names and 20% faster for 1024-byte names. Performance for longer names remains unchanged. (Contributed by Bénédikt Tran in gh-128150 [https://github.com/python/cpython/issues/128150].)
- `uuid4()` is now c. 30% faster. (Contributed by Bénédikt Tran in gh-128150 [https://github.com/python/cpython/issues/128150].)

zlib

- On Windows, zlib-ng [https://github.com/zlib-ng/zlib-ng] is now used as the implementation of the `zlib` module in the default binaries. There are no known incompatibilities between `zlib-ng` and the previously-used `zlib` implementation. This should result in better performance at all compression levels.

It is worth noting that `zlib.Z_BEST_SPEED` (`1`) may result in significantly less compression than the previous implementation, whilst also significantly reducing the time taken to compress.

(Contributed by Steve Dower in gh-91349 [https://github.com/python/cpython/issues/91349].)

Removed

argparse

- Remove the *type*, *choices*, and *metavar* parameters of **`BooleanOptionalAction`**. These have been deprecated since Python 3.12. (Contributed by Nikita Sobolev in gh-118805 [https://github.com/python/cpython/issues/118805].)
- Calling **`add_argument_group()`** on an argument group now raises a **`ValueError`**. Similarly, **`add_argument_group()`** or **`add_mutually_exclusive_group()`** on a mutually exclusive group now both raise **`ValueError`**s. This ‘nesting’ was never supported, often failed to work correctly, and was unintentionally exposed through inheritance. This functionality has been deprecated since Python 3.11. (Contributed by Savannah Ostrowski in gh-127186 [https://github.com/python/cpython/issues/127186].)

ast

- Remove the following classes, which have been deprecated aliases of **`Constant`** since Python 3.8 and have emitted deprecation warnings since Python 3.12:
    - **`Bytes`**
    - **`Ellipsis`**
    - **`NameConstant`**

  - **`Num`**
  - **`Str`**

  As a consequence of these removals, user-defined `visit_Num`, `visit_Str`, `visit_Bytes`, `visit_NameConstant` and `visit_Ellipsis` methods on custom **`NodeVisitor`** subclasses will no longer be called when the **`NodeVisitor`** subclass is visiting an AST. Define a `visit_Constant` method instead.

  (Contributed by Alex Waygood in gh-119562 [https://github.com/python/cpython/issues/119562].)

- Remove the following deprecated properties on **`ast.Constant`**, which were present for compatibility with the now-removed AST classes:

  - **`Constant.n`**
  - **`Constant.s`**

  Use **`Constant.value`** instead. (Contributed by Alex Waygood in gh-119562 [https://github.com/python/cpython/issues/119562].)

asyncio

- Remove the following classes, methods, and functions, which have been deprecated since Python 3.12:

  - **`AbstractChildWatcher`**
  - **`FastChildWatcher`**
  - **`MultiLoopChildWatcher`**
  - **`PidfdChildWatcher`**
  - **`SafeChildWatcher`**
  - **`ThreadedChildWatcher`**
  - **`AbstractEventLoopPolicy.get_child_watcher()`**
  - **`AbstractEventLoopPolicy.set_child_watcher()`**
  - **`get_child_watcher()`**

  - **`set_child_watcher()`**

  (Contributed by Kumar Aditya in gh-120804 [https://github.com/python/cpython/issues/120804].)

- `asyncio.get_event_loop()` now raises a `RuntimeError` if there is no current event loop, and no longer implicitly creates an event loop.

  (Contributed by Kumar Aditya in gh-126353 [https://github.com/python/cpython/issues/126353].)

  There’s a few patterns that use `asyncio.get_event_loop()`, most of them can be replaced with `asyncio.run()`.

  If you’re running an async function, simply use `asyncio.run()`.

  Before:

  ```
  async def main():
      ...


  loop = asyncio.get_event_loop()
  try:
      loop.run_until_complete(main())
  finally:
      loop.close()
  ```

  After:

  ```
  async def main():
      ...

  asyncio.run(main())
  ```

  If you need to start something, for example, a server listening on a socket and then run forever, use `asyncio.run()` and an `asyncio.Event`.

Before:

```
def start_server(loop): ...

loop = asyncio.get_event_loop()
try:
    start_server(loop)
    loop.run_forever()
finally:
    loop.close()
```

After:

```
def start_server(loop): ...

async def main():
    start_server(asyncio.get_running_loop())
    await asyncio.Event().wait()

asyncio.run(main())
```

If you need to run something in an event loop, then run some blocking code around it, use `asyncio.Runner`.

Before:

```
async def operation_one(): ...
def blocking_code(): ...
async def operation_two(): ...

loop = asyncio.get_event_loop()
try:
    loop.run_until_complete(operation_one())
    blocking_code()
    loop.run_until_complete(operation_two())
finally:
    loop.close()
```

After:

```
async def operation_one(): ...
def blocking_code(): ...
async def operation_two(): ...

with asyncio.Runner() as runner:
    runner.run(operation_one())
    blocking_code()
    runner.run(operation_two())
```

email

- Remove `email.utils.localtime()`’s *isdst* parameter, which was deprecated in and has been ignored since Python 3.12. (Contributed by Hugo van Kemenade in gh-118798 [https://github.com/python/cpython/issues/118798].)

importlib.abc

- Remove deprecated `importlib.abc` classes:
    - **`ResourceReader`** (use `TraversableResources`)
    - **`Traversable`** (use `Traversable`)
    - **`TraversableResources`** (use `TraversableResources`)

  (Contributed by Jason R. Coombs and Hugo van Kemenade in gh-93963 [https://github.com/python/cpython/issues/93963].)

itertools

- Remove support for copy, deepcopy, and pickle operations from `itertools` iterators. These have emitted a `DeprecationWarning` since Python 3.12. (Contributed by Raymond Hettinger in gh-101588 [https://github.com/python/cpython/issues/101588].)

pathlib

- Remove support for passing additional keyword arguments to **`Path`**. In previous versions, any such arguments are ignored. (Contributed by Barney Gale in gh-74033 [https://github.com/python/cpython/issues/74033].)
- Remove support for passing additional positional arguments to **`PurePath.relative_to()`** and **`is_relative_to()`**. In previous versions, any such arguments are joined onto *other*. (Contributed by Barney Gale in gh-78707 [https://github.com/python/cpython/issues/78707].)

pkgutil

- Remove the **`get_loader()`** and **`find_loader()`** functions, which have been deprecated since Python 3.12. (Contributed by Bénédikt Tran in gh-97850 [https://github.com/python/cpython/issues/97850].)

pty

- Remove the **`master_open()`** and **`slave_open()`** functions, which have been deprecated since Python 3.12. Use **`pty.openpty()`** instead. (Contributed by Nikita Sobolev in gh-118824 [https://github.com/python/cpython/issues/118824].)

sqlite3

- Remove **`version`** and **`version_info`** from the **`sqlite3`** module; use **`sqlite_version`** and **`sqlite_version_info`** for the actual version number of the runtime SQLite library. (Contributed by Hugo van Kemenade in gh-118924 [https://github.com/python/cpython/issues/118924].)
- Using a sequence of parameters with named placeholders now raises a **`ProgrammingError`**, having been deprecated since Python 3.12. (Contributed by Erlend E. Aasland in gh-118928

[https://github.com/python/cpython/issues/118928] and gh-101693 [https://github.com/python/cpython/issues/101693].)

urllib

- Remove the **`Quoter`** class from **`urllib.parse`**, which has been deprecated since Python 3.11. (Contributed by Nikita Sobolev in gh-118827 [https://github.com/python/cpython/issues/118827].)
- Remove the **`URLopener`** and **`FancyURLopener`** classes from **`urllib.request`**, which have been deprecated since Python 3.3.

  `myopener.open()` can be replaced with **`urlopen()`**. `myopener.retrieve()` can be replaced with **`urlretrieve()`**. Customisations to the opener classes can be replaced by passing customized handlers to **`build_opener()`**. (Contributed by Barney Gale in gh-84850 [https://github.com/python/cpython/issues/84850].)

Deprecated

New deprecations

- Passing a complex number as the *real* or *imag* argument in the **`complex()`** constructor is now deprecated; complex numbers should only be passed as a single positional argument. (Contributed by Serhiy Storchaka in gh-109218 [https://github.com/python/cpython/issues/109218].)
- **`argparse`**:
  - Passing the undocumented keyword argument *prefix_chars* to the **`add_argument_group()`** method is now deprecated. (Contributed by Savannah

Ostrowski in gh-125563 [https://github.com/python/cpython/issues/125563].)
    - Deprecated the `argparse.FileType` type converter. Anything relating to resource management should be handled downstream, after the arguments have been parsed. (Contributed by Serhiy Storchaka in gh-58032 [https://github.com/python/cpython/issues/58032].)

- `asyncio`:

    - The **`asyncio.iscoroutinefunction()`** is now deprecated and will be removed in Python 3.16; use `inspect.iscoroutinefunction()` instead. (Contributed by Jiahao Li and Kumar Aditya in gh-122875 [https://github.com/python/cpython/issues/122875].)

    - The `asyncio` policy system is deprecated and will be removed in Python 3.16. In particular, the following classes and functions are deprecated:

        - `asyncio.AbstractEventLoopPolicy`
        - `asyncio.DefaultEventLoopPolicy`
        - `asyncio.WindowsSelectorEventLoopPolicy`
        - `asyncio.WindowsProactorEventLoopPolicy`
        - `asyncio.get_event_loop_policy()`
        - `asyncio.set_event_loop_policy()`

      Users should use `asyncio.run()` or `asyncio.Runner` with the *loop_factory* argument to use the desired event loop implementation.

      For example, to use `asyncio.SelectorEventLoop` on Windows:

      ```
      import asyncio

      async def main():
      ```

```
    ...

asyncio.run(main(),
loop_factory=asyncio.SelectorEventLoop)
```

(Contributed by Kumar Aditya in gh-127949 [https://github.com/python/cpython/issues/127949].)

- **`codecs`**: The **`codecs.open()`** function is now deprecated, and will be removed in a future version of Python. Use **`open()`** instead. (Contributed by Inada Naoki in gh-133036 [https://github.com/python/cpython/issues/133036].)

- **`ctypes`**:

    - On non-Windows platforms, setting **`Structure._pack_`** to use a MSVC-compatible default memory layout is now deprecated in favor of setting **`Structure._layout_`** to `'ms'`, and will be removed in Python 3.19. (Contributed by Petr Viktorin in gh-131747 [https://github.com/python/cpython/issues/131747].)
    - Calling **`ctypes.POINTER()`** on a string is now deprecated. Use incomplete types for self-referential structures. Also, the internal `ctypes._pointer_type_cache` is deprecated. See **`ctypes.POINTER()`** for updated implementation details. (Contributed by Sergey Myrianov in gh-100926 [https://github.com/python/cpython/issues/100926].)

- **`functools`**: Calling the Python implementation of **`functools.reduce()`** with *function* or *sequence* as keyword arguments is now deprecated; the parameters will be made positional-only in Python 3.16. (Contributed by Kirill Podoprigora in gh-121676 [https://github.com/python/cpython/issues/121676].)

- **`logging`**: Support for custom logging handlers with the *strm* argument is now deprecated and scheduled for removal in Python 3.16. Define handlers with the *stream* argument instead. (Contributed by Mariusz Felisiak in gh-115032 [https://github.com/python/cpython/issues/115032].)

- **`mimetypes`**: Valid extensions are either empty or must start with '.' for **`mimetypes.MimeTypes.add_type()`**. Undotted extensions are deprecated and will raise a **`ValueError`** in Python 3.16. (Contributed by Hugo van Kemenade in gh-75223 [https://github.com/python/cpython/issues/75223].)

- **`nturl2path`**: This module is now deprecated. Call **`urllib.request.url2pathname()`** and **`pathname2url()`** instead. (Contributed by Barney Gale in gh-125866 [https://github.com/python/cpython/issues/125866].)

- **`os`**: The **`os.popen()`** and **`os.spawn*`** functions are now soft deprecated. They should no longer be used to write new code. The **`subprocess`** module is recommended instead. (Contributed by Victor Stinner in gh-120743 [https://github.com/python/cpython/issues/120743].)

- **`pathlib`**: **`pathlib.PurePath.as_uri()`** is now deprecated and scheduled for removal in Python 3.19. Use **`pathlib.Path.as_uri()`** instead. (Contributed by Barney Gale in gh-123599 [https://github.com/python/cpython/issues/123599].)

- **`pdb`**: The undocumented `pdb.Pdb.curframe_locals` attribute is now a deprecated read-only property, which will be removed in a future version of Python. The low overhead dynamic frame locals access added in Python 3.13 by **PEP 667** [https://peps.python.org/pep-0667/] means the frame locals cache reference previously stored in this attribute is no longer needed. Derived debuggers should access `pdb.Pdb.curframe.f_locals` directly in Python 3.13 and later versions. (Contributed by Tian Gao in gh-124369 [https://github.com/python/cpython/issues/124369] and gh-125951 [https://github.com/python/cpython/issues/125951].)

- **`symtable`**: Deprecate **`symtable.Class.get_methods()`** due to the lack of interest, scheduled for removal in Python 3.16. (Contributed by Bénédikt Tran in gh-119698 [https://github.com/python/cpython/issues/119698].)
- **`tkinter`**: The **`tkinter.Variable`** methods **`trace_variable()`**, **`trace_vdelete()`** and **`trace_vinfo()`** are now deprecated. Use **`trace_add()`**, **`trace_remove()`** and **`trace_info()`** instead. (Contributed by Serhiy Storchaka in gh-120220 [https://github.com/python/cpython/issues/120220].)
- **`urllib.parse`**: Accepting objects with false values (like `0` and `[]`) except empty strings, bytes-like objects and `None` in **`parse_qsl()`** and **`parse_qs()`** is now deprecated. (Contributed by Serhiy Storchaka in gh-116897 [https://github.com/python/cpython/issues/116897].)

Pending removal in Python 3.15

- The import system:
  - Setting **`__cached__`** on a module while failing to set **`__spec__.cached`** is deprecated. In Python 3.15, **`__cached__`** will cease to be set or take into consideration by the import system or standard library. (gh-97879 [https://github.com/python/cpython/issues/97879])
  - Setting **`__package__`** on a module while failing to set **`__spec__.parent`** is deprecated. In Python 3.15, **`__package__`** will cease to be set or take into consideration by the import system or standard

library. (gh-97879 [https://github.com/python/cpython/issues/97879])

- `ctypes`:
  - The undocumented **`ctypes.SetPointerType()`** function has been deprecated since Python 3.13.
- `http.server`:
  - The obsolete and rarely used `CGIHTTPRequestHandler` has been deprecated since Python 3.13. No direct replacement exists. *Anything* is better than CGI to interface a web server with a request handler.
  - The **`--cgi`** flag to the **python -m http.server** command-line interface has been deprecated since Python 3.13.
- `importlib`:
  - `load_module()` method: use `exec_module()` instead.
- `locale`:
  - The `getdefaultlocale()` function has been deprecated since Python 3.11. Its removal was originally planned for Python 3.13 (gh-90817 [https://github.com/python/cpython/issues/90817]), but has been postponed to Python 3.15. Use `getlocale()`, `setlocale()`, and `getencoding()` instead. (Contributed by Hugo van Kemenade in gh-111187 [https://github.com/python/cpython/issues/111187].)
- `pathlib`:
  - `PurePath.is_reserved()` has been deprecated since Python 3.13. Use `os.path.isreserved()` to detect reserved paths on Windows.
- `platform`:
  - `java_ver()` has been deprecated since Python 3.13. This function is only useful for Jython support, has a confusing API, and is largely untested.
- `sysconfig`:
  - The *check_home* argument of `sysconfig.is_python_build()` has been deprecated since Python 3.12.

- `threading`:
    - `RLock()` will take no arguments in Python 3.15. Passing any arguments has been deprecated since Python 3.14, as the Python version does not permit any arguments, but the C version allows any number of positional or keyword arguments, ignoring every argument.
- `types`:
    - `types.CodeType`: Accessing `co_lnotab` was deprecated in **PEP 626** [https://peps.python.org/pep-0626/] since 3.10 and was planned to be removed in 3.12, but it only got a proper `DeprecationWarning` in 3.12. May be removed in 3.15. (Contributed by Nikita Sobolev in gh-101866 [https://github.com/python/cpython/issues/101866].)
- `typing`:
    - The undocumented keyword argument syntax for creating `NamedTuple` classes (for example, `Point = NamedTuple("Point", x=int, y=int)`) has been deprecated since Python 3.13. Use the class-based syntax or the functional syntax instead.
    - When using the functional syntax of `TypedDict`s, failing to pass a value to the *fields* parameter (`TD = TypedDict("TD")`) or passing `None` (`TD = TypedDict("TD", None)`) has been deprecated since Python 3.13. Use `class TD(TypedDict): pass` or `TD = TypedDict("TD", {})` to create a TypedDict with zero field.
    - The `typing.no_type_check_decorator()` decorator function has been deprecated since Python 3.13. After eight years in the `typing` module, it has yet to be supported by any major type checker.
- `wave`:
    - The `getmark()`, **`setmark()`**, and `getmarkers()` methods of the `Wave_read` and `Wave_write` classes have been deprecated since Python 3.13.
- `zipimport`:

  - `load_module()` has been deprecated since Python 3.10. Use `exec_module()` instead. (Contributed by Jiahao Li in gh-125746 [https://github.com/python/cpython/issues/125746].)

Pending removal in Python 3.16

- The import system:
  - Setting `__loader__` on a module while failing to set `__spec__.loader` is deprecated. In Python 3.16, `__loader__` will cease to be set or taken into consideration by the import system or the standard library.
- `array`:
  - The `'u'` format code (`wchar_t`) has been deprecated in documentation since Python 3.3 and at runtime since Python 3.13. Use the `'w'` format code (`Py_UCS4`) for Unicode characters instead.
- `asyncio`:
  - `asyncio.iscoroutinefunction()` is deprecated and will be removed in Python 3.16; use `inspect.iscoroutinefunction()` instead. (Contributed by Jiahao Li and Kumar Aditya in gh-122875 [https://github.com/python/cpython/issues/122875].)
  - `asyncio` policy system is deprecated and will be removed in Python 3.16. In particular, the following classes and functions are deprecated:
    - `asyncio.AbstractEventLoopPolicy`
    - `asyncio.DefaultEventLoopPolicy`
    - `asyncio.WindowsSelectorEventLoopPolicy`

        - `asyncio.WindowsProactorEventLoopPolicy`
        - `asyncio.get_event_loop_policy()`
        - `asyncio.set_event_loop_policy()`

    Users should use `asyncio.run()` or `asyncio.Runner` with *loop_factory* to use the desired event loop implementation.

    For example, to use `asyncio.SelectorEventLoop` on Windows:

    ```
    import asyncio

    async def main():
        ...

    asyncio.run(main(),
    loop_factory=asyncio.SelectorEventLoop)
    ```

    (Contributed by Kumar Aditya in gh-127949 [https://github.com/python/cpython/issues/127949].)

- **`builtins`**:

    - Bitwise inversion on boolean types, `~True` or `~False` has been deprecated since Python 3.12, as it produces surprising and unintuitive results (`-2` and `-1`). Use `not x` instead for the logical negation of a Boolean. In the rare case that you need the bitwise inversion of the underlying integer, convert to `int` explicitly (`~int(x)`).

- **`functools`**:

    - Calling the Python implementation of `functools.reduce()` with *function* or *sequence* as keyword arguments has been deprecated since Python 3.14.

- `logging`:

  Support for custom logging handlers with the *strm* argument is deprecated and scheduled for removal in Python 3.16. Define handlers with the *stream* argument instead. (Contributed by Mariusz Felisiak in gh-115032 [https://github.com/python/cpython/issues/115032].)

- `mimetypes`:
  - Valid extensions start with a '.' or are empty for `mimetypes.MimeTypes.add_type()`. Undotted extensions are deprecated and will raise a `ValueError` in Python 3.16. (Contributed by Hugo van Kemenade in gh-75223 [https://github.com/python/cpython/issues/75223].)

- `shutil`:
  - The `ExecError` exception has been deprecated since Python 3.14. It has not been used by any function in `shutil` since Python 3.4, and is now an alias of `RuntimeError`.

- `symtable`:
  - The `Class.get_methods` method has been deprecated since Python 3.14.

- `sys`:
  - The `_enablelegacywindowsfsencoding()` function has been deprecated since Python 3.13. Use the `PYTHONLEGACYWINDOWSFSENCODING` environment variable instead.

- `sysconfig`:

  - The **`sysconfig.expand_makefile_vars()`** function has been deprecated since Python 3.14. Use the `vars` argument of [`sysconfig.get_paths()`](#) instead.

- [`tarfile`](#):

  - The undocumented and unused **`TarFile.tarfile`** attribute has been deprecated since Python 3.13.

Pending removal in Python 3.17

- [`collections.abc`](#):

  - [`collections.abc.ByteString`](#) is scheduled for removal in Python 3.17.

    Use `isinstance(obj, collections.abc.Buffer)` to test if `obj` implements the buffer protocol at runtime. For use in type annotations, either use **`Buffer`** or a union that explicitly specifies the types your code supports (e.g., `bytes | bytearray | memoryview`).

    `ByteString` was originally intended to be an abstract class that would serve as a supertype of both **`bytes`** and **`bytearray`**. However, since the ABC never had any methods, knowing that an object was an instance of **`ByteString`** never actually told you anything useful about the object. Other common buffer types such as **`memoryview`** were also never understood as subtypes of **`ByteString`** (either at runtime or by static type checkers).

    See **PEP 688** [https://peps.python.org/pep-0688/#current-options] for more details. (Contributed by Shantanu Jain in gh-91896 [https://github.com/python/cpython/issues/91896].)

- `typing`:
    - Before Python 3.14, old-style unions were implemented using the private class `typing._UnionGenericAlias`. This class is no longer needed for the implementation, but it has been retained for backward compatibility, with removal scheduled for Python 3.17. Users should use documented introspection helpers like `typing.get_origin()` and `typing.get_args()` instead of relying on private implementation details.
    - `typing.ByteString`, deprecated since Python 3.9, is scheduled for removal in Python 3.17.

      Use `isinstance(obj, collections.abc.Buffer)` to test if `obj` implements the buffer protocol at runtime. For use in type annotations, either use `Buffer` or a union that explicitly specifies the types your code supports (e.g., `bytes | bytearray | memoryview`).

      `ByteString` was originally intended to be an abstract class that would serve as a supertype of both `bytes` and `bytearray`. However, since the ABC never had any methods, knowing that an object was an instance of **`ByteString`** never actually told you anything useful about the object. Other common buffer types such as `memoryview` were also never understood as subtypes of **`ByteString`** (either at runtime or by static type checkers).

      See **PEP 688** [https://peps.python.org/pep-0688/#current-options] for more details. (Contributed by Shantanu Jain in gh-91896 [https://github.com/python/cpython/issues/91896].)

Pending removal in Python 3.18

- `decimal`:
    - The non-standard and undocumented `Decimal` format specifier `'N'`, which is only supported in the **`decimal`** module's C implementation, has been deprecated since Python 3.13. (Contributed by Serhiy Storchaka in gh-89902 [https://github.com/python/cpython/issues/89902].)

Pending removal in Python 3.19

- `ctypes`:
    - Implicitly switching to the MSVC-compatible struct layout by setting `_pack_` but not `_layout_` on non-Windows platforms.

Pending removal in future versions

The following APIs will be removed in the future, although there is currently no date scheduled for their removal.

- `argparse`:
    - Nesting argument groups and nesting mutually exclusive groups are deprecated.
    - Passing the undocumented keyword argument *prefix_chars* to `add_argument_group()` is now deprecated.
    - The `argparse.FileType` type converter is deprecated.
- `builtins`:
    - Generators: `throw(type, exc, tb)` and `athrow(type, exc, tb)` signature is deprecated: use `throw(exc)` and `athrow(exc)` instead, the single argument signature.
    - Currently Python accepts numeric literals immediately followed by keywords, for example `0in x`, `1or x`, `0if 1else 2`. It allows confusing and ambiguous expressions like `[0x1for x in y]` (which

can be interpreted as `[0x1 for x in y]` or `[0x1f or x in y]`). A syntax warning is raised if the numeric literal is immediately followed by one of keywords **`and`**, **`else`**, **`for`**, **`if`**, **`in`**, **`is`** and **`or`**. In a future release it will be changed to a syntax error. (gh-87999 [https://github.com/python/cpython/issues/87999])
    - Support for `__index__()` and `__int__()` method returning non-int type: these methods will be required to return an instance of a strict subclass of **`int`**.
    - Support for `__float__()` method returning a strict subclass of **`float`**: these methods will be required to return an instance of **`float`**.
    - Support for `__complex__()` method returning a strict subclass of **`complex`**: these methods will be required to return an instance of **`complex`**.
    - Delegation of `int()` to `__trunc__()` method.
    - Passing a complex number as the *real* or *imag* argument in the **`complex()`** constructor is now deprecated; it should only be passed as a single positional argument. (Contributed by Serhiy Storchaka in gh-109218 [https://github.com/python/cpython/issues/109218].)
- **`calendar`**: `calendar.January` and `calendar.February` constants are deprecated and replaced by **`calendar.JANUARY`** and **`calendar.FEBRUARY`**. (Contributed by Prince Roshan in gh-103636 [https://github.com/python/cpython/issues/103636].)
- **`codecs`**: use **`open()`** instead of **`codecs.open()`**. (gh-133038 [https://github.com/python/cpython/issues/133038])
- **`codeobject.co_lnotab`**: use the **`codeobject.co_lines()`** method instead.
- **`datetime`**:
    - **`utcnow()`**: use `datetime.datetime.now(tz=datetime.UTC)`.
    - **`utcfromtimestamp()`**: use `datetime.datetime.fromtimestamp(timestamp, tz=datetime.UTC)`.

- **`gettext`**: Plural value must be an integer.
- **`importlib`**:
  - **`cache_from_source()`** *debug_override* parameter is deprecated: use the *optimization* parameter instead.
- **`importlib.metadata`**:
  - `EntryPoints` tuple interface.
  - Implicit `None` on return values.
- **`logging`**: the `warn()` method has been deprecated since Python 3.3, use **`warning()`** instead.
- **`mailbox`**: Use of StringIO input and text mode is deprecated, use BytesIO and binary mode instead.
- **`os`**: Calling **`os.register_at_fork()`** in multi-threaded process.
- **`pydoc.ErrorDuringImport`**: A tuple value for *exc_info* parameter is deprecated, use an exception instance.
- **`re`**: More strict rules are now applied for numerical group references and group names in regular expressions. Only sequence of ASCII digits is now accepted as a numerical reference. The group name in bytes patterns and replacement strings can now only contain ASCII letters and digits and underscore. (Contributed by Serhiy Storchaka in gh-91760 [https://github.com/python/cpython/issues/91760].)
- **`sre_compile`**, **`sre_constants`** and **`sre_parse`** modules.
- **`shutil`**: **`rmtree()`**’s *onerror* parameter is deprecated in Python 3.12; use the *onexc* parameter instead.
- **`ssl`** options and protocols:
  - **`ssl.SSLContext`** without protocol argument is deprecated.
  - **`ssl.SSLContext`**: **`set_npn_protocols()`** and **`selected_npn_protocol()`** are deprecated: use ALPN instead.
  - `ssl.OP_NO_SSL*` options
  - `ssl.OP_NO_TLS*` options
  - `ssl.PROTOCOL_SSLv3`

  - `ssl.PROTOCOL_TLS`
  - `ssl.PROTOCOL_TLSv1`
  - `ssl.PROTOCOL_TLSv1_1`
  - `ssl.PROTOCOL_TLSv1_2`
  - `ssl.TLSVersion.SSLv3`
  - `ssl.TLSVersion.TLSv1`
  - `ssl.TLSVersion.TLSv1_1`
- **`threading`** methods:
  - **`threading.Condition.notifyAll()`**: use **`notify_all()`**.
  - **`threading.Event.isSet()`**: use **`is_set()`**.
  - **`threading.Thread.isDaemon()`**, **`threading.Thread.setDaemon()`**: use **`threading.Thread.daemon`** attribute.
  - **`threading.Thread.getName()`**, **`threading.Thread.setName()`**: use **`threading.Thread.name`** attribute.
  - **`threading.currentThread()`**: use **`threading.current_thread()`**.
  - **`threading.activeCount()`**: use **`threading.active_count()`**.
- **`typing.Text`** (gh-92332 [https://github.com/python/cpython/issues/92332]).
- The internal class `typing._UnionGenericAlias` is no longer used to implement **`typing.Union`**. To preserve compatibility with users using this private class, a compatibility shim will be provided until at least Python 3.17. (Contributed by Jelle Zijlstra in gh-105499 [https://github.com/python/cpython/issues/105499].)
- **`unittest.IsolatedAsyncioTestCase`**: it is deprecated to return a value that is not `None` from a test case.
- **`urllib.parse`** deprecated functions: **`urlparse()`** instead
  - `splitattr()`
  - `splithost()`
  - `splitnport()`
  - `splitpasswd()`
  - `splitport()`
  - `splitquery()`
  - `splittag()`

  - `splittype()`
  - `splituser()`
  - `splitvalue()`
  - `to_bytes()`
- **`wsgiref`**: `SimpleHandler.stdout.write()` should not do partial writes.
- **`xml.etree.ElementTree`**: Testing the truth value of an **`Element`** is deprecated. In a future release it will always return `True`. Prefer explicit `len(elem)` or `elem is not None` tests instead.
- **`sys._clear_type_cache()`** is deprecated: use **`sys._clear_internal_caches()`** instead.

CPython bytecode changes

- Replaced the opcode **`BINARY_SUBSCR`** by the **`BINARY_OP`** opcode with the `NB_SUBSCR` oparg. (Contributed by Irit Katriel in gh-100239 [https://github.com/python/cpython/issues/100239].)
- Add the **`BUILD_INTERPOLATION`** and **`BUILD_TEMPLATE`** opcodes to construct new **`Interpolation`** and **`Template`** instances, respectively. (Contributed by Lysandros Nikolaou and others in gh-132661 [https://github.com/python/cpython/issues/132661]; see also PEP 750: Template strings).
- Remove the **`BUILD_CONST_KEY_MAP`** opcode. Use **`BUILD_MAP`** instead. (Contributed by Mark Shannon in gh-122160 [https://github.com/python/cpython/issues/122160].)
- Replace the **`LOAD_ASSERTION_ERROR`** opcode with **`LOAD_COMMON_CONSTANT`** and add support for loading **`NotImplementedError`**.
- Add the **`LOAD_FAST_BORROW`** and **`LOAD_FAST_BORROW_LOAD_FAST_BORROW`** opcodes to reduce reference counting overhead when the interpreter can prove that the reference in the frame outlives the

reference loaded onto the stack. (Contributed by Matt Page in gh-130704 [https://github.com/python/cpython/issues/130704].)

- Add the **`LOAD_SMALL_INT`** opcode, which pushes a small integer equal to the `oparg` to the stack. The **`RETURN_CONST`** opcode is removed as it is no longer used. (Contributed by Mark Shannon in gh-125837 [https://github.com/python/cpython/issues/125837].)
- Add the new **`LOAD_SPECIAL`** instruction. Generate code for **`with`** and **`async with`** statements using the new instruction. Removed the **`BEFORE_WITH`** and **`BEFORE_ASYNC_WITH`** instructions. (Contributed by Mark Shannon in gh-120507 [https://github.com/python/cpython/issues/120507].)
- Add the **`POP_ITER`** opcode to support ‘virtual’ iterators. (Contributed by Mark Shannon in gh-132554 [https://github.com/python/cpython/issues/132554].)

Pseudo-instructions

- Add the **`ANNOTATIONS_PLACEHOLDER`** pseudo instruction to support partially executed module-level annotations with deferred evaluation of annotations. (Contributed by Jelle Zijlstra in gh-130907 [https://github.com/python/cpython/issues/130907].)
- Add the **`BINARY_OP_EXTEND`** pseudo instruction, which executes a pair of functions (guard and specialization functions) accessed from the inline cache. (Contributed by Irit Katriel in gh-100239 [https://github.com/python/cpython/issues/100239].)
- Add three specializations for **`CALL_KW`**; **`CALL_KW_PY`** for calls to Python functions, **`CALL_KW_BOUND_METHOD`** for calls to bound methods, and **`CALL_KW_NON_PY`** for all other calls. (Contributed by Mark Shannon in gh-118093 [https://github.com/python/cpython/issues/118093].)

- Add the **`JUMP_IF_TRUE`** and **`JUMP_IF_FALSE`** pseudo instructions, conditional jumps which do not impact the stack. Replaced by the sequence `COPY 1`, `TO_BOOL`, `POP_JUMP_IF_TRUE/FALSE`. (Contributed by Irit Katriel in gh-124285 [https://github.com/python/cpython/issues/124285].)
- Add the **`LOAD_CONST_MORTAL`** pseudo instruction. (Contributed by Mark Shannon in gh-128685 [https://github.com/python/cpython/issues/128685].)
- Add the **`LOAD_CONST_IMMORTAL`** pseudo instruction, which does the same as **`LOAD_CONST`**, but is more efficient for immortal objects. (Contributed by Mark Shannon in gh-125837 [https://github.com/python/cpython/issues/125837].)
- Add the **`NOT_TAKEN`** pseudo instruction, used by **`sys.monitoring`** to record branch events (such as **`BRANCH_LEFT`**). (Contributed by Mark Shannon in gh-122548 [https://github.com/python/cpython/issues/122548].)

C API changes

Python configuration C API

Add a PyInitConfig C API to configure the Python initialization without relying on C structures and the ability to make ABI-compatible changes in the future.

Complete the **PEP 587** [https://peps.python.org/pep-0587/] PyConfig C API by adding **`PyInitConfig_AddModule()`** which can be used to add a built-in extension module; a feature previously referred to as the “inittab”.

Add **`PyConfig_Get()`** and **`PyConfig_Set()`** functions to get and set the current runtime configuration.

PEP 587 [https://peps.python.org/pep-0587/] ‘Python Initialization Configuration’ unified all the ways to configure Python’s initialization. This PEP also unifies the configuration of

Python’s preinitialization and initialization in a single API. Moreover, this PEP only provides a single choice to embed Python, instead of having two ‘Python’ and ‘Isolated’ choices (PEP 587), to further simplify the API.

The lower level PEP 587 PyConfig API remains available for use cases with an intentionally higher level of coupling to CPython implementation details (such as emulating the full functionality of CPython’s CLI, including its configuration mechanisms).

(Contributed by Victor Stinner in gh-107954 [https://github.com/python/cpython/issues/107954].)

See also

PEP 741 [https://peps.python.org/pep-0741/] and **PEP 587** [https://peps.python.org/pep-0587/]

New features in the C API

- Add `Py_PACK_VERSION()` and `Py_PACK_FULL_VERSION()`, two new macros for bit-packing Python version numbers. This is useful for comparisons with `Py_Version` or `PY_VERSION_HEX`. (Contributed by Petr Viktorin in gh-128629 [https://github.com/python/cpython/issues/128629].)
- Add `PyBytes_Join(sep, iterable)` function, similar to `sep.join(iterable)` in Python. (Contributed by Victor Stinner in gh-121645 [https://github.com/python/cpython/issues/121645].)
- Add functions to manipulate the configuration of the current runtime Python interpreter (PEP 741: Python configuration C API):

  - `PyConfig_Get()`
  - `PyConfig_GetInt()`
  - `PyConfig_Set()`
  - `PyConfig_Names()`

  (Contributed by Victor Stinner in gh-107954 [https://github.com/python/cpython/issues/107954].)

- Add functions to configure Python initialization (PEP 741: Python configuration C API):

  - `Py_InitializeFromInitConfig()`
  - `PyInitConfig_AddModule()`
  - `PyInitConfig_Create()`
  - `PyInitConfig_Free()`
  - `PyInitConfig_FreeStrList()`
  - `PyInitConfig_GetError()`
  - `PyInitConfig_GetExitCode()`
  - `PyInitConfig_GetInt()`
  - `PyInitConfig_GetStr()`
  - `PyInitConfig_GetStrList()`
  - `PyInitConfig_HasOption()`
  - `PyInitConfig_SetInt()`
  - `PyInitConfig_SetStr()`
  - `PyInitConfig_SetStrList()`

  (Contributed by Victor Stinner in gh-107954 [https://github.com/python/cpython/issues/107954].)

- Add `Py_fopen()` function to open a file. This works similarly to the standard C `fopen()` function, instead accepting a Python object for the *path* parameter and setting an exception on error. The corresponding new `Py_fclose()` function should be used to close a file. (Contributed by Victor Stinner in gh-127350 [https://github.com/python/cpython/issues/127350].)

- Add `Py_HashBuffer()` to compute and return the hash value of a buffer. (Contributed by Antoine Pitrou and Victor Stinner in gh-122854 [https://github.com/python/cpython/issues/122854].)

- Add `PyImport_ImportModuleAttr()` and `PyImport_ImportModuleAttrString()` helper functions to import a module and get an attribute of the module. (Contributed by Victor Stinner in gh-128911 [https://github.com/python/cpython/issues/128911].)

- Add `PyIter_NextItem()` to replace `PyIter_Next()`, which has an ambiguous return value. (Contributed by Irit Katriel and Erlend Aasland in gh-105201 [https://github.com/python/cpython/issues/105201].)

- Add `PyLong_GetSign()` function to get the sign of `int` objects. (Contributed by Sergey B Kirpichev in gh-116560 [https://github.com/python/cpython/issues/116560].)

- Add `PyLong_IsPositive()`, `PyLong_IsNegative()` and `PyLong_IsZero()` for checking if `PyLongObject` is positive, negative, or zero, respectively. (Contributed by James Roy and Sergey B Kirpichev in gh-126061 [https://github.com/python/cpython/issues/126061].)

- Add new functions to convert C `<stdint.h>` numbers to/from Python `int` objects:

    - `PyLong_AsInt32()`
    - `PyLong_AsInt64()`
    - `PyLong_AsUInt32()`
    - `PyLong_AsUInt64()`
    - `PyLong_FromInt32()`
    - `PyLong_FromInt64()`
    - `PyLong_FromUInt32()`
    - `PyLong_FromUInt64()`

(Contributed by Victor Stinner in gh-120389 [https://github.com/python/cpython/issues/120389].)

- Add a new import and export API for Python `int` objects (**PEP 757** [https://peps.python.org/pep-0757/]):
    - `PyLong_GetNativeLayout()`
    - `PyLong_Export()`
    - `PyLong_FreeExport()`
    - `PyLongWriter_Create()`
    - `PyLongWriter_Finish()`
    - `PyLongWriter_Discard()`

  (Contributed by Sergey B Kirpichev and Victor Stinner in gh-102471 [https://github.com/python/cpython/issues/102471].)

- Add `PyMonitoring_FireBranchLeftEvent()` and `PyMonitoring_FireBranchRightEvent()` for generating `BRANCH_LEFT` and `BRANCH_RIGHT` events, respectively. (Contributed by Mark Shannon in gh-122548 [https://github.com/python/cpython/issues/122548].)

- Add `PyType_Freeze()` function to make a type immutable. (Contributed by Victor Stinner in gh-121654 [https://github.com/python/cpython/issues/121654].)

- Add `PyType_GetBaseByToken()` and `Py_tp_token` slot for easier superclass identification, which attempts to resolve the type checking issue mentioned in **PEP 630** [https://peps.python.org/pep-0630/#type-checking]. (Contributed in gh-124153 [https://github.com/python/cpython/issues/124153].)

- Add a new `PyUnicode_Equal()` function to test if two strings are equal. The function is also added to the Limited C API. (Contributed by Victor Stinner in gh-124502 [https://github.com/python/cpython/issues/124502].)

- Add a new `PyUnicodeWriter` API to create a Python `str` object, with the following functions:

  - `PyUnicodeWriter_Create()`
  - `PyUnicodeWriter_DecodeUTF8Stateful()`
  - `PyUnicodeWriter_Discard()`
  - `PyUnicodeWriter_Finish()`
  - `PyUnicodeWriter_Format()`
  - `PyUnicodeWriter_WriteASCII()`
  - `PyUnicodeWriter_WriteChar()`
  - `PyUnicodeWriter_WriteRepr()`
  - `PyUnicodeWriter_WriteStr()`
  - `PyUnicodeWriter_WriteSubstring()`
  - `PyUnicodeWriter_WriteUCS4()`
  - `PyUnicodeWriter_WriteUTF8()`
  - `PyUnicodeWriter_WriteWideChar()`

  (Contributed by Victor Stinner in gh-119182 [https://github.com/python/cpython/issues/119182].)

- The `k` and `K` formats in `PyArg_ParseTuple()` and similar functions now use `__index__()` if available, like all other integer formats. (Contributed by Serhiy Storchaka in gh-112068 [https://github.com/python/cpython/issues/112068].)

- Add support for a new `p` format unit in `Py_BuildValue()` that produces a Python `bool` object from a C integer. (Contributed by Pablo Galindo in bpo-45325 [https://bugs.python.org/issue?@action=redirect&bpo=45325].)

- Add `PyUnstable_IsImmortal()` for determining if an object is immortal, for debugging purposes. (Contributed by Peter Bierma in gh-128509 [https://github.com/python/cpython/issues/128509].)

- Add `PyUnstable_Object_EnableDeferredRefcount()` for enabling deferred reference counting, as outlined in **PEP**

703 [https://peps.python.org/pep-0703/].

- Add `PyUnstable_Object_IsUniquelyReferenced()` as a replacement for `Py_REFCNT(op) == 1` on free threaded builds. (Contributed by Peter Bierma in gh-133140 [https://github.com/python/cpython/issues/133140].)

- Add `PyUnstable_Object_IsUniqueReferencedTemporary()` to determine if an object is a unique temporary object on the interpreter's operand stack. This can be used in some cases as a replacement for checking if `Py_REFCNT()` is `1` for Python objects passed as arguments to C API functions. (Contributed by Sam Gross in gh-133164 [https://github.com/python/cpython/issues/133164].)

Limited C API changes

- In the limited C API version 3.14 and newer, `Py_TYPE()` and `Py_REFCNT()` are now implemented as an opaque function call to hide implementation details. (Contributed by Victor Stinner in gh-120600 [https://github.com/python/cpython/issues/120600] and gh-124127 [https://github.com/python/cpython/issues/124127].)
- Remove the `PySequence_Fast_GET_SIZE`, `PySequence_Fast_GET_ITEM`, and `PySequence_Fast_ITEMS` macros from the limited C API, since they have always been broken in the limited C API. (Contributed by Victor Stinner in gh-91417 [https://github.com/python/cpython/issues/91417].)

Removed C APIs

- Creating `immutable types` with mutable bases was deprecated in Python 3.12, and now raises a `TypeError`. (Contributed by Nikita Sobolev in gh-119775 [https://github.com/python/cpython/issues/119775].)

- Remove `PyDictObject.ma_version_tag` member, which was deprecated in Python 3.12. Use the **`PyDict_AddWatcher()`** API instead. (Contributed by Sam Gross in gh-124296 [https://github.com/python/cpython/issues/124296].)
- Remove the private `_Py_InitializeMain()` function. It was a provisional API added to Python 3.8 by **PEP 587** [https://peps.python.org/pep-0587/]. (Contributed by Victor Stinner in gh-129033 [https://github.com/python/cpython/issues/129033].)
- Remove the undocumented APIs **`Py_C_RECURSION_LIMIT`** and **`PyThreadState.c_recursion_remaining`**. These were added in 3.13 and have been removed without deprecation. Use **`Py_EnterRecursiveCall()`** to guard against runaway recursion in C code. (Removed by Petr Viktorin in gh-133079 [https://github.com/python/cpython/issues/133079], see also gh-130396 [https://github.com/python/cpython/issues/130396].)

Deprecated C APIs

- The **`Py_HUGE_VAL`** macro is now soft deprecated. Use **`Py_INFINITY`** instead. (Contributed by Sergey B Kirpichev in gh-120026 [https://github.com/python/cpython/issues/120026].)
- The **`Py_IS_NAN`**, **`Py_IS_INFINITY`**, and **`Py_IS_FINITE`** macros are now soft deprecated. Use **`isnan`**, **`isinf`** and **`isfinite`** instead, available from `math.h` since C99. (Contributed by Sergey B Kirpichev in gh-119613 [https://github.com/python/cpython/issues/119613].)
- Non-tuple sequences are now deprecated as argument for the `(items)` format unit in **`PyArg_ParseTuple()`** and other argument parsing functions if *items* contains format units which store a borrowed buffer or a borrowed reference. (Contributed by Serhiy Storchaka in gh-50333 [https://github.com/python/cpython/issues/50333].)
- The `_PyMonitoring_FireBranchEvent` function is now deprecated and should be replaced with calls to

`PyMonitoring_FireBranchLeftEvent()` and `PyMonitoring_FireBranchRightEvent()`.
- The previously undocumented function `PySequence_In()` is now soft deprecated. Use `PySequence_Contains()` instead. (Contributed by Yuki Kobayashi in gh-127896 [https://github.com/python/cpython/issues/127896].)

Pending removal in Python 3.15

- The `PyImport_ImportModuleNoBlock()`: Use `PyImport_ImportModule()` instead.

- `PyWeakref_GetObject()` and `PyWeakref_GET_OBJECT()`: Use `PyWeakref_GetRef()` instead. The pythoncapi-compat project [https://github.com/python/pythoncapi-compat/] can be used to get `PyWeakref_GetRef()` on Python 3.12 and older.

- `Py_UNICODE` type and the `Py_UNICODE_WIDE` macro: Use `wchar_t` instead.

- `PyUnicode_AsDecodedObject()`: Use `PyCodec_Decode()` instead.

- `PyUnicode_AsDecodedUnicode()`: Use `PyCodec_Decode()` instead; Note that some codecs (for example, “base64”) may return a type other than `str`, such as `bytes`.

- `PyUnicode_AsEncodedObject()`: Use `PyCodec_Encode()` instead.

- `PyUnicode_AsEncodedUnicode()`: Use `PyCodec_Encode()` instead; Note that some codecs (for example, “base64”) may return a type other than `bytes`, such as `str`.

- Python initialization functions, deprecated in Python 3.13:

    - `Py_GetPath()`: Use `PyConfig_Get("module_search_paths")` (`sys.path`) instead.

  - `Py_GetPrefix()`: Use `PyConfig_Get("base_prefix")` (`sys.base_prefix`) instead. Use `PyConfig_Get("prefix")` (`sys.prefix`) if virtual environments need to be handled.
  - `Py_GetExecPrefix()`: Use `PyConfig_Get("base_exec_prefix")` (`sys.base_exec_prefix`) instead. Use `PyConfig_Get("exec_prefix")` (`sys.exec_prefix`) if virtual environments need to be handled.
  - `Py_GetProgramFullPath()`: Use `PyConfig_Get("executable")` (`sys.executable`) instead.
  - `Py_GetProgramName()`: Use `PyConfig_Get("executable")` (`sys.executable`) instead.
  - `Py_GetPythonHome()`: Use `PyConfig_Get("home")` or the `PYTHONHOME` environment variable instead.

  The pythoncapi-compat project [https://github.com/python/pythoncapi-compat/] can be used to get `PyConfig_Get()` on Python 3.13 and older.

- Functions to configure Python's initialization, deprecated in Python 3.11:

  - **`PySys_SetArgvEx()`**: Set `PyConfig.argv` instead.
  - **`PySys_SetArgv()`**: Set `PyConfig.argv` instead.
  - **`Py_SetProgramName()`**: Set `PyConfig.program_name` instead.
  - **`Py_SetPythonHome()`**: Set `PyConfig.home` instead.
  - `PySys_ResetWarnOptions()`: Clear `sys.warnoptions` and **`warnings.filters`** instead.

  The `Py_InitializeFromConfig()` API should be used with `PyConfig` instead.

- Global configuration variables:

  - `Py_DebugFlag`: Use `PyConfig.parser_debug` or `PyConfig_Get("parser_debug")` instead.

- `Py_VerboseFlag`: Use `PyConfig.verbose` or `PyConfig_Get("verbose")` instead.
- `Py_QuietFlag`: Use `PyConfig.quiet` or `PyConfig_Get("quiet")` instead.
- `Py_InteractiveFlag`: Use `PyConfig.interactive` or `PyConfig_Get("interactive")` instead.
- `Py_InspectFlag`: Use `PyConfig.inspect` or `PyConfig_Get("inspect")` instead.
- `Py_OptimizeFlag`: Use `PyConfig.optimization_level` or `PyConfig_Get("optimization_level")` instead.
- `Py_NoSiteFlag`: Use `PyConfig.site_import` or `PyConfig_Get("site_import")` instead.
- `Py_BytesWarningFlag`: Use `PyConfig.bytes_warning` or `PyConfig_Get("bytes_warning")` instead.
- `Py_FrozenFlag`: Use `PyConfig.pathconfig_warnings` or `PyConfig_Get("pathconfig_warnings")` instead.
- `Py_IgnoreEnvironmentFlag`: Use `PyConfig.use_environment` or `PyConfig_Get("use_environment")` instead.
- `Py_DontWriteBytecodeFlag`: Use `PyConfig.write_bytecode` or `PyConfig_Get("write_bytecode")` instead.
- `Py_NoUserSiteDirectory`: Use `PyConfig.user_site_directory` or `PyConfig_Get("user_site_directory")` instead.
- `Py_UnbufferedStdioFlag`: Use `PyConfig.buffered_stdio` or `PyConfig_Get("buffered_stdio")` instead.
- `Py_HashRandomizationFlag`: Use `PyConfig.use_hash_seed` and `PyConfig.hash_seed` or `PyConfig_Get("hash_seed")` instead.
- `Py_IsolatedFlag`: Use `PyConfig.isolated` or `PyConfig_Get("isolated")` instead.
- `Py_LegacyWindowsFSEncodingFlag`: Use `PyPreConfig.legacy_windows_fs_encoding` or `PyConfig_Get("legacy_windows_fs_encoding")` instead.
- `Py_LegacyWindowsStdioFlag`: Use `PyConfig.legacy_windows_stdio` or `PyConfig_Get("legacy_windows_stdio")` instead.

  - **`Py_FileSystemDefaultEncoding`**, **`Py_HasFileSystemDefaultEncoding`**: Use [`PyConfig.filesystem_encoding`] or [`PyConfig_Get("filesystem_encoding")`] instead.
  - **`Py_FileSystemDefaultEncodeErrors`**: Use [`PyConfig.filesystem_errors`] or [`PyConfig_Get("filesystem_errors")`] instead.
  - **`Py_UTF8Mode`**: Use [`PyPreConfig.utf8_mode`] or [`PyConfig_Get("utf8_mode")`] instead. (see [`Py_PreInitialize()`])

The [`Py_InitializeFromConfig()`] API should be used with [`PyConfig`] to set these options. Or [`PyConfig_Get()`] can be used to get these options at runtime.

Pending removal in Python 3.16

- The bundled copy of `libmpdec`.

Pending removal in Python 3.18

- The following private functions are deprecated and planned for removal in Python 3.18:
  - **`_PyBytes_Join()`**: use [`PyBytes_Join()`].
  - **`_PyDict_GetItemStringWithError()`**: use [`PyDict_GetItemStringRef()`].
  - **`_PyDict_Pop()`**: use [`PyDict_Pop()`].
  - **`_PyLong_Sign()`**: use [`PyLong_GetSign()`].
  - **`_PyLong_FromDigits()`** and **`_PyLong_New()`**: use [`PyLongWriter_Create()`].
  - **`_PyThreadState_UncheckedGet()`**: use [`PyThreadState_GetUnchecked()`].
  - **`_PyUnicode_AsString()`**: use [`PyUnicode_AsUTF8()`].

- **`_PyUnicodeWriter_Init()`**: replace `_PyUnicodeWriter_Init(&writer)` with `writer = PyUnicodeWriter_Create(0)`.
- **`_PyUnicodeWriter_Finish()`**: replace `_PyUnicodeWriter_Finish(&writer)` with `PyUnicodeWriter_Finish(writer)`.
- **`_PyUnicodeWriter_Dealloc()`**: replace `_PyUnicodeWriter_Dealloc(&writer)` with `PyUnicodeWriter_Discard(writer)`.
- **`_PyUnicodeWriter_WriteChar()`**: replace `_PyUnicodeWriter_WriteChar(&writer, ch)` with `PyUnicodeWriter_WriteChar(writer, ch)`.
- **`_PyUnicodeWriter_WriteStr()`**: replace `_PyUnicodeWriter_WriteStr(&writer, str)` with `PyUnicodeWriter_WriteStr(writer, str)`.
- **`_PyUnicodeWriter_WriteSubstring()`**: replace `_PyUnicodeWriter_WriteSubstring(&writer, str, start, end)` with `PyUnicodeWriter_WriteSubstring(writer, str, start, end)`.
- **`_PyUnicodeWriter_WriteASCIIString()`**: replace `_PyUnicodeWriter_WriteASCIIString(&writer, str)` with `PyUnicodeWriter_WriteASCII(writer, str)`.
- **`_PyUnicodeWriter_WriteLatin1String()`**: replace `_PyUnicodeWriter_WriteLatin1String(&writer, str)` with `PyUnicodeWriter_WriteUTF8(writer, str)`.
- **`_PyUnicodeWriter_Prepare()`**: (no replacement).
- **`_PyUnicodeWriter_PrepareKind()`**: (no replacement).
- **`_Py_HashPointer()`**: use `Py_HashPointer()`.
- **`_Py_fopen_obj()`**: use `Py_fopen()`.

The pythoncapi-compat project [https://github.com/python/pythoncapi-compat/] can be used to get these new public functions on Python 3.13 and older. (Contributed by Victor Stinner in gh-128863 [https://github.com/python/cpython/issues/128863].)

Pending removal in future versions

The following APIs are deprecated and will be removed, although there is currently no date scheduled for their removal.

- `Py_TPFLAGS_HAVE_FINALIZE`: Unneeded since Python 3.8.
- `PyErr_Fetch()`: Use `PyErr_GetRaisedException()` instead.
- `PyErr_NormalizeException()`: Use `PyErr_GetRaisedException()` instead.
- `PyErr_Restore()`: Use `PyErr_SetRaisedException()` instead.
- `PyModule_GetFilename()`: Use `PyModule_GetFilenameObject()` instead.
- `PyOS_AfterFork()`: Use `PyOS_AfterFork_Child()` instead.
- `PySlice_GetIndicesEx()`: Use `PySlice_Unpack()` and `PySlice_AdjustIndices()` instead.
- `PyUnicode_READY()`: Unneeded since Python 3.12
- `PyErr_Display()`: Use `PyErr_DisplayException()` instead.
- `_PyErr_ChainExceptions()`: Use `_PyErr_ChainExceptions1()` instead.
- `PyBytesObject.ob_shash` member: call `PyObject_Hash()` instead.
- Thread Local Storage (TLS) API:
  - `PyThread_create_key()`: Use `PyThread_tss_alloc()` instead.
  - `PyThread_delete_key()`: Use `PyThread_tss_free()` instead.
  - `PyThread_set_key_value()`: Use `PyThread_tss_set()` instead.
  - `PyThread_get_key_value()`: Use `PyThread_tss_get()` instead.
  - `PyThread_delete_key_value()`: Use `PyThread_tss_delete()` instead.
  - `PyThread_ReInitTLS()`: Unneeded since Python 3.7.

Build changes

- **PEP 776** [https://peps.python.org/pep-0776/]: Emscripten is now an officially supported platform at **tier 3** [https://peps.python.org/pep-0011/#tier-3]. As a part of this effort, more than 25 bugs in Emscripten libc [https://emscripten.org/docs/porting/emscripten-runtime-environment.html] were fixed. Emscripten now includes support for **`ctypes`**, **`termios`**, and **`fcntl`**, as well as experimental support for the new default interactive shell. (Contributed by R. Hood Chatham in gh-127146 [https://github.com/python/cpython/issues/127146], gh-127683 [https://github.com/python/cpython/issues/127683], and gh-136931 [https://github.com/python/cpython/issues/136931].)
- Official Android binary releases are now provided on python.org [https://www.python.org/downloads/android/].
- GNU Autoconf 2.72 is now required to generate `configure`. (Contributed by Erlend Aasland in gh-115765 [https://github.com/python/cpython/issues/115765].)
- `wasm32-unknown-emscripten` is now a **PEP 11** [https://peps.python.org/pep-0011/] tier 3 platform. (Contributed by R. Hood Chatham in gh-127146 [https://github.com/python/cpython/issues/127146], gh-127683 [https://github.com/python/cpython/issues/127683], and gh-136931 [https://github.com/python/cpython/issues/136931].)
- `#pragma`-based linking with `python3*.lib` can now be switched off with Py_NO_LINK_LIB. (Contributed by Jean-Christophe Fillion-Robin in gh-82909 [https://github.com/python/cpython/issues/82909].)
- CPython now enables a set of recommended compiler options by default for improved security. Use the **`--disable-safety`** `configure` option to disable them, or the **`--enable-slower-safety`** option for a larger set of compiler options, albeit with a performance cost.
- The `WITH_FREELISTS` macro and `--without-freelists` `configure` option have been removed.
- The new `configure` option **`--with-tail-call-interp`** may be used to enable the experimental tail call interpreter. See

A new type of interpreter for further details.

- To disable the new remote debugging support, use the **`--without-remote-debug`** `configure` option. This may be useful for security reasons.
- iOS and macOS apps can now be configured to redirect `stdout` and `stderr` content to the system log. (Contributed by Russell Keith-Magee in gh-127592 [https://github.com/python/cpython/issues/127592].)
- The iOS testbed is now able to stream test output while the test is running. The testbed can also be used to run the test suite of projects other than CPython itself. (Contributed by Russell Keith-Magee in gh-127592 [https://github.com/python/cpython/issues/127592].)

`build-details.json`

Installations of Python now contain a new file, `build-details.json`. This is a static JSON document containing build details for CPython, to allow for introspection without needing to run code. This is helpful for use-cases such as Python launchers, cross-compilation, and so on.

`build-details.json` must be installed in the platform-independent standard library directory. This corresponds to the 'stdlib' **`sysconfig`** installation path, which can be found by running `sysconfig.get_path('stdlib')`.

See also

PEP 739 [https://peps.python.org/pep-0739/] – `build-details.json` 1.0 – a static description file for Python build details

Discontinuation of PGP signatures

PGP (Pretty Good Privacy) signatures will not be provided for releases of Python 3.14 or future versions. To verify CPython artifacts, users must use Sigstore verification materials [https://www.python.org/downloads/metadata/sigstore/]. Releases have been signed using Sigstore [https://www.sigstore.dev/] since Python 3.11.

This change in release process was specified in **PEP 761** [https://peps.python.org/pep-0761/].

Free-threaded Python is officially supported

The free-threaded build of Python is now supported and no longer experimental. This is the start of phase II [https://discuss.python.org/t/37075] where free-threaded Python is officially supported but still optional.

The free-threading team are confident that the project is on the right path, and appreciate the continued dedication from everyone working to make free-threading ready for broader adoption across the Python community.

With these recommendations and the acceptance of this PEP, the Python developer community should broadly advertise that free-threading is a supported Python build option now and into the future, and that it will not be removed without a proper deprecation schedule.

Any decision to transition to phase III [https://discuss.python.org/t/37075], with free-threading as the default or sole build of Python is still undecided, and dependent on many factors both within CPython itself and the community. This decision is for the future.

See also

PEP 779 [https://peps.python.org/pep-0779/]

PEP 779’s acceptance [https://discuss.python.org/t/84319/123]

Binary releases for the experimental just-in-time compiler

The official macOS and Windows release binaries now include an *experimental* just-in-time (JIT) compiler. Although it is **not** recommended for production use, it can be tested by setting `PYTHON_JIT=1` as an environment variable. Downstream source builds and redistributors can use the `--enable-experimental-jit=yes-off` configuration option for similar behavior.

The JIT is at an early stage and still in active development. As such, the typical performance impact of enabling it can range from 10% slower to 20% faster, depending on workload. To aid in testing and evaluation, a set of introspection functions has been provided in the `sys._jit` namespace. `sys._jit.is_available()` can be used to determine if the current executable supports JIT compilation, while `sys._jit.is_enabled()` can be used to tell if JIT compilation has been enabled for the current process.

Currently, the most significant missing functionality is that native debuggers and profilers like `gdb` and `perf` are unable to unwind through JIT frames (Python debuggers and profilers, like `pdb` or `profile`, continue to work without modification). Free-threaded builds do not support JIT compilation.

Please report any bugs or major performance regressions that you encounter!

See also

PEP 744 [https://peps.python.org/pep-0744/]

Porting to Python 3.14

This section lists previously described changes and other bugfixes that may require changes to your code.

Changes in the Python API

- On Unix platforms other than macOS, *forkserver* is now the default start method for `multiprocessing` and `ProcessPoolExecutor`, instead of *fork*.

  See (1) and (2) for details.

  If you encounter `NameError`s or pickling errors coming out of `multiprocessing` or `concurrent.futures`, see the forkserver restrictions.

  This change does not affect Windows or macOS, where ‘spawn’ remains the default start method.

- `functools.partial` is now a method descriptor. Wrap it in `staticmethod()` if you want to preserve the old behavior. (Contributed by Serhiy Storchaka and Dominykas Grigonis in gh-121027 [https://github.com/python/cpython/issues/121027].)

- The garbage collector is now incremental, which means that the behavior of `gc.collect()` changes slightly:

  - `gc.collect(1)`: Performs an increment of garbage collection, rather than collecting generation 1.

  - Other calls to **`gc.collect()`** are unchanged.

- The [**`locale.nl_langinfo()`**]() function now temporarily sets the `LC_CTYPE` locale in some cases. This temporary change affects other threads. (Contributed by Serhiy Storchaka in [gh-69998]() [https://github.com/python/cpython/issues/69998].)

- [**`types.UnionType`**]() is now an alias for [**`typing.Union`**](), causing changes in some behaviors. See [above]() for more details. (Contributed by Jelle Zijlstra in [gh-105499]() [https://github.com/python/cpython/issues/105499].)

- The runtime behavior of annotations has changed in various ways; see [above]() for details. While most code that interacts with annotations should continue to work, some undocumented details may behave differently.

- As part of making the [**`mimetypes`**]() CLI public, it now exits with `1` on failure instead of `0` and `2` on incorrect command-line parameters instead of `1`. Error messages are now printed to stderr.

- The `\B` pattern in regular expression now matches the empty string when given as the entire pattern, which may cause behavioural changes.

- On FreeBSD, [**`sys.platform`**]() no longer contains the major version number.

Changes in annotations ([PEP 649]() [https://peps.python.org/pep-0649/] and [PEP 749]() [https://peps.python.org/pep-0749/])

This section contains guidance on changes that may be needed to annotations or Python code that interacts with or introspects annotations, due to the changes related to deferred evaluation of annotations.

In the majority of cases, working code from older versions of Python will not require any changes.

Implications for annotated code

If you define annotations in your code (for example, for use with a static type checker), then this change probably does not affect you: you can keep writing annotations the same way you did with previous versions of Python.

You will likely be able to remove quoted strings in annotations, which are frequently used for forward references. Similarly, if you use `from __future__ import annotations` to avoid having to write strings in annotations, you may well be able to remove that import once you support only Python 3.14 and newer. However, if you rely on third-party libraries that read annotations, those libraries may need changes to support unquoted annotations before they work as expected.

Implications for readers of `__annotations__`

If your code reads the **`__annotations__`** attribute on objects, you may want to make changes in order to support code that relies on deferred evaluation of annotations. For example, you may want to use **`annotationlib.get_annotations()`** with the **`FORWARDREF`** format, as the **`dataclasses`** module now does.

The external typing_extensions [https://pypi.org/project/typing_extensions/] package provides partial backports of some of the functionality of the **`annotationlib`** module, such as the **`Format`** enum and the **`get_annotations()`**

function. These can be used to write cross-version code that takes advantage of the new behavior in Python 3.14.

Related changes

The changes in Python 3.14 are designed to rework how `__annotations__` works at runtime while minimizing breakage to code that contains annotations in source code and to code that reads `__annotations__`. However, if you rely on undocumented details of the annotation behavior or on private functions in the standard library, there are many ways in which your code may not work in Python 3.14. To safeguard your code against future changes, only use the documented functionality of the `annotationlib` module.

In particular, do not read annotations directly from the namespace dictionary attribute of type objects. Use `annotationlib.get_annotate_from_class_namespace()` during class construction and `annotationlib.get_annotations()` afterwards.

In previous releases, it was sometimes possible to access class annotations from an instance of an annotated class. This behavior was undocumented and accidental, and will no longer work in Python 3.14.

`from __future__ import annotations`

In Python 3.7, **PEP 563** [https://peps.python.org/pep-0563/] introduced the `from __future__ import annotations` future statement, which turns all annotations into strings.

However, this statement is now deprecated and it is expected to be removed in a future version of Python. This removal will not happen until after Python 3.13 reaches its end of life in 2029, being the last version of Python without support for deferred evaluation of annotations.

In Python 3.14, the behavior of code using `from __future__ import annotations` is unchanged.

Changes in the C API

- **`Py_Finalize()`** now deletes all interned strings. This is backwards incompatible to any C extension that holds onto an interned string after a call to **`Py_Finalize()`** and is then reused after a call to **`Py_Initialize()`**. Any issues arising from this behavior will normally result in crashes during the execution of the subsequent call to **`Py_Initialize()`** from accessing uninitialized memory. To fix, use an address sanitizer to identify any use-after-free coming from an interned string and deallocate it during module shutdown. (Contributed by Eddie Elizondo in gh-113601 [https://github.com/python/cpython/issues/113601].)
- The Unicode Exception Objects C API now raises a **`TypeError`** if its exception argument is not a **`UnicodeError`** object. (Contributed by Bénédikt Tran in gh-127691 [https://github.com/python/cpython/issues/127691].)

- The interpreter internally avoids some reference count modifications when loading objects onto the operands stack by borrowing references when possible. This can lead to smaller reference count values compared to previous Python versions. C API extensions that checked **`Py_REFCNT()`** of `1` to determine if an function argument is not referenced by any other code should instead use **`PyUnstable_Object_IsUniqueReferencedTemporary()`** as a safer replacement.

- Private functions promoted to public C APIs:
  - `_PyBytes_Join()`: **`PyBytes_Join()`**
  - `_PyLong_IsNegative()`: **`PyLong_IsNegative()`**

  - `_PyLong_IsPositive()`: **`PyLong_IsPositive()`**
  - `_PyLong_IsZero()`: **`PyLong_IsZero()`**
  - `_PyLong_Sign()`: **`PyLong_GetSign()`**
  - `_PyUnicodeWriter_Dealloc()`: **`PyUnicodeWriter_Discard()`**
  - `_PyUnicodeWriter_Finish()`: **`PyUnicodeWriter_Finish()`**
  - `_PyUnicodeWriter_Init()`: use **`PyUnicodeWriter_Create()`**
  - `_PyUnicodeWriter_Prepare()`: (no replacement)
  - `_PyUnicodeWriter_PrepareKind()`: (no replacement)
  - `_PyUnicodeWriter_WriteChar()`: **`PyUnicodeWriter_WriteChar()`**
  - `_PyUnicodeWriter_WriteStr()`: **`PyUnicodeWriter_WriteStr()`**
  - `_PyUnicodeWriter_WriteSubstring()`: **`PyUnicodeWriter_WriteSubstring()`**
  - `_PyUnicode_EQ()`: **`PyUnicode_Equal()`**
  - `_PyUnicode_Equal()`: **`PyUnicode_Equal()`**
  - `_Py_GetConfig()`: **`PyConfig_Get()`** and **`PyConfig_GetInt()`**
  - `_Py_HashBytes()`: **`Py_HashBuffer()`**
  - `_Py_fopen_obj()`: **`Py_fopen()`**
  - `PyMutex_IsLocked()` : **`PyMutex_IsLocked()`**

The pythoncapi-compat project [https://github.com/python/pythoncapi-compat/] can be used to get most of these new functions on Python 3.13 and older.

Notable changes in 3.14.1

- Add **`PyUnstable_ThreadState_SetStackProtection()`** and **`PyUnstable_ThreadState_ResetStackProtection()`** functions to set the stack protection base address and stack protection size of a Python thread state. (Contributed by Victor Stinner in gh-139653 [https://github.com/python/cpython/issues/139653].)

What’s New In Python 3.13

Editors:: Adam Turner and Thomas Wouters

This article explains the new features in Python 3.13, compared to 3.12. Python 3.13 was released on October 7, 2024. For full details, see the changelog.

See also

PEP 719 [https://peps.python.org/pep-0719/] – Python 3.13 Release Schedule

Summary – Release Highlights

Python 3.13 is a stable release of the Python programming language, with a mix of changes to the language, the implementation and the standard library. The biggest changes include a new interactive interpreter, experimental support for running in a free-threaded mode (**PEP 703** [https://peps.python.org/pep-0703/]), and a Just-In-Time compiler (**PEP 744** [https://peps.python.org/pep-0744/]).

Error messages continue to improve, with tracebacks now highlighted in color by default. The `locals()` builtin now has defined semantics for changing the returned mapping, and type parameters now support default values.

The library changes contain removal of deprecated APIs and modules, as well as the usual improvements in user-friendliness and correctness. Several legacy standard library modules have now been removed following their deprecation in Python 3.11 (**PEP 594** [https://peps.python.org/pep-0594/]).

This article doesn't attempt to provide a complete specification of all new features, but instead gives a convenient overview. For full details refer to the documentation, such as the Library Reference and Language Reference. To understand the complete implementation and design rationale for a change, refer to the PEP for a particular new feature; but note that PEPs usually are not kept up-to-date once a feature has been fully implemented. See Porting to Python 3.13 for guidance on upgrading from earlier versions of Python.

Interpreter improvements:

- A greatly improved interactive interpreter and improved error messages.
- **PEP 667** [https://peps.python.org/pep-0667/]: The `locals()` builtin now has defined semantics when mutating the returned mapping. Python debuggers and similar tools may now more reliably update local variables in optimized scopes even during concurrent code execution.
- **PEP 703** [https://peps.python.org/pep-0703/]: CPython 3.13 has experimental support for running with the global interpreter lock disabled. See Free-threaded CPython for more details.
- **PEP 744** [https://peps.python.org/pep-0744/]: A basic JIT compiler was added. It is currently disabled by default (though we may turn it on later). Performance improvements are modest – we expect to improve this over the next few releases.

- Color support in the new interactive interpreter, as well as in tracebacks and doctest output. This can be disabled through the **`PYTHON_COLORS`** and **`NO_COLOR`** [https://no-color.org/] environment variables.

Python data model improvements:

- **`__static_attributes__`** stores the names of attributes accessed through `self.X` in any function in a class body.
- **`__firstlineno__`** records the first line number of a class definition.

Significant improvements in the standard library:

- Add a new **`PythonFinalizationError`** exception, raised when an operation is blocked during finalization.
- The **`argparse`** module now supports deprecating command-line options, positional arguments, and subcommands.
- The new functions **`base64.z85encode()`** and **`base64.z85decode()`** support encoding and decoding Z85 data [https://rfc.zeromq.org/spec/32/].
- The **`copy`** module now has a **`copy.replace()`** function, with support for many builtin types and any class defining the **`__replace__()`** method.
- The new **`dbm.sqlite3`** module is now the default **`dbm`** backend.
- The **`os`** module has a suite of new functions for working with Linux’s timer notification file descriptors.
- The **`random`** module now has a command-line interface.

Security improvements:

- **`ssl.create_default_context()`** sets **`ssl.VERIFY_X509_PARTIAL_CHAIN`** and **`ssl.VERIFY_X509_STRICT`** as default flags.

C API improvements:

- The `Py_mod_gil` slot is now used to indicate that an extension module supports running with the GIL disabled.
- The PyTime C API has been added, providing access to system clocks.
- `PyMutex` is a new lightweight mutex that occupies a single byte.
- There is a new suite of functions for generating **PEP 669** [https://peps.python.org/pep-0669/] monitoring events in the C API.

New typing features:

- **PEP 696** [https://peps.python.org/pep-0696/]: Type parameters (`typing.TypeVar`, `typing.ParamSpec`, and `typing.TypeVarTuple`) now support defaults.
- **PEP 702** [https://peps.python.org/pep-0702/]: The new `warnings.deprecated()` decorator adds support for marking deprecations in the type system and at runtime.
- **PEP 705** [https://peps.python.org/pep-0705/]: `typing.ReadOnly` can be used to mark an item of a `typing.TypedDict` as read-only for type checkers.
- **PEP 742** [https://peps.python.org/pep-0742/]: `typing.TypeIs` provides more intuitive type narrowing behavior, as an alternative to `typing.TypeGuard`.

Platform support:

- **PEP 730** [https://peps.python.org/pep-0730/]: Apple’s iOS is now an officially supported platform, at **tier 3** [https://peps.python.org/pep-0011/#tier-3].
- **PEP 738** [https://peps.python.org/pep-0738/]: Android is now an officially supported platform, at **tier 3** [https://peps.python.org/pep-0011/#tier-3].

- `wasm32-wasi` is now supported as a [**tier 2**] [https://peps.python.org/pep-0011/#tier-2] platform.
- `wasm32-emscripten` is no longer an officially supported platform.

Important removals:

- PEP 594: The remaining 19 "dead batteries" (legacy stdlib modules) have been removed from the standard library: **`aifc`**, **`audioop`**, **`cgi`**, **`cgitb`**, **`chunk`**, **`crypt`**, **`imghdr`**, **`mailcap`**, **`msilib`**, **`nis`**, **`nntplib`**, **`ossaudiodev`**, **`pipes`**, **`sndhdr`**, **`spwd`**, **`sunau`**, **`telnetlib`**, **`uu`** and **`xdrlib`**.
- Remove the **2to3** tool and **`lib2to3`** module (deprecated in Python 3.11).
- Remove the **`tkinter.tix`** module (deprecated in Python 3.6).
- Remove the **`locale.resetlocale()`** function.
- Remove the **`typing.io`** and **`typing.re`** namespaces.
- Remove chained **`classmethod`** descriptors.

Release schedule changes:

PEP 602 [https://peps.python.org/pep-0602/] ("Annual Release Cycle for Python") has been updated to extend the full support ('bugfix') period for new releases to two years. This updated policy means that:

- Python 3.9–3.12 have one and a half years of full support, followed by three and a half years of security fixes.
- Python 3.13 and later have two years of full support, followed by three years of security fixes.

New Features

A better interactive interpreter

Python now uses a new interactive shell by default, based on code from the PyPy project [https://pypy.org/]. When the user starts the REPL from an interactive terminal, the following new features are now supported:

- Multiline editing with history preservation.
- Direct support for REPL-specific commands like `help`, `exit`, and `quit`, without the need to call them as functions.
- Prompts and tracebacks with color enabled by default.
- Interactive help browsing using `F1` with a separate command history.
- History browsing using `F2` that skips output as well as the >>> and ... prompts.
- “Paste mode” with `F3` that makes pasting larger blocks of code easier (press `F3` again to return to the regular prompt).

To disable the new interactive shell, set the **`PYTHON_BASIC_REPL`** environment variable. For more on interactive mode, see Interactive Mode.

(Contributed by Pablo Galindo Salgado, Łukasz Langa, and Lysandros Nikolaou in gh-111201 [https://github.com/python/cpython/issues/111201] based on code from the PyPy project. Windows support contributed by Dino Viehland and Anthony Shaw.)

Improved error messages

- The interpreter now uses color by default when displaying tracebacks in the terminal. This feature can be controlled via the new **`PYTHON_COLORS`** environment variable as well as the canonical **`NO_COLOR`** [https://no-color.org/] and **`FORCE_COLOR`** [https://force-color.org/] environment variables. (Contributed by Pablo Galindo Salgado in gh-112730 [https://github.com/python/cpython/issues/112730].)

- A common mistake is to write a script with the same name as a standard library module. When this results in errors, we now display a more helpful error message:

  ```
  $ python random.py
  Traceback (most recent call last):
    File "/home/me/random.py", line 1, in <module>
      import random
    File "/home/me/random.py", line 3, in <module>
      print(random.randint(5))
            ^^^^^^^^^^^^^^
  AttributeError: module 'random' has no attribute 'randint'
  (consider renaming '/home/me/random.py' since it has the
  same name as the standard library module named 'random' and
  prevents importing that standard library module)
  ```

  Similarly, if a script has the same name as a third-party module that it attempts to import and this results in errors, we also display a more helpful error message:

  ```
  $ python numpy.py
  Traceback (most recent call last):
    File "/home/me/numpy.py", line 1, in <module>
      import numpy as np
    File "/home/me/numpy.py", line 3, in <module>
      np.array([1, 2, 3])
      ^^^^^^^^
  AttributeError: module 'numpy' has no attribute 'array'
  (consider renaming '/home/me/numpy.py' if it has the same
  name as a library you intended to import)
  ```

  (Contributed by Shantanu Jain in gh-95754 [https://github.com/python/cpython/issues/95754].)

- The error message now tries to suggest the correct keyword argument when an incorrect keyword argument is passed to a function.

```
>>> "Better error messages!".split(max_split=1)
Traceback (most recent call last):
  File "<python-input-0>", line 1, in <module>
    "Better error messages!".split(max_split=1)
    ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~^^^^^^^^^^^^^
TypeError: split() got an unexpected keyword argument
'max_split'. Did you mean 'maxsplit'?
```

(Contributed by Pablo Galindo Salgado and Shantanu Jain in gh-107944 [https://github.com/python/cpython/issues/107944].)

Free-threaded CPython

CPython now has experimental support for running in a free-threaded mode, with the global interpreter lock (GIL) disabled. This is an experimental feature and therefore is not enabled by default. The free-threaded mode requires a different executable, usually called `python3.13t` or `python3.13t.exe`. Pre-built binaries marked as *free-threaded* can be installed as part of the official Windows and macOS installers, or CPython can be built from source with the **`--disable-gil`** option.

Free-threaded execution allows for full utilization of the available processing power by running threads in parallel on available CPU cores. While not all software will benefit from this automatically, programs designed with threading in mind will run faster on multi-core hardware. **The free-threaded mode is experimental** and work is ongoing to improve it: expect some bugs and a substantial single-threaded performance hit. Free-threaded builds of CPython support optionally running with the GIL enabled at runtime using the environment variable **`PYTHON_GIL`** or the command-line option **`-X gil=1`**.

To check if the current interpreter supports free-threading, `python -VV` and `sys.version` contain “experimental free-threading build”. The new `sys._is_gil_enabled()` function can be used to check whether the GIL is actually disabled in the running process.

C-API extension modules need to be built specifically for the free-threaded build. Extensions that support running with the GIL disabled should use the `Py_mod_gil` slot. Extensions using single-phase init should use `PyUnstable_Module_SetGIL()` to indicate whether they support running with the GIL disabled. Importing C extensions that don’t use these mechanisms will cause the GIL to be enabled, unless the GIL was explicitly disabled with the `PYTHON_GIL` environment variable or the `-X gil=0` option. pip 24.1 or newer is required to install packages with C extensions in the free-threaded build.

This work was made possible thanks to many individuals and organizations, including the large community of contributors to Python and third-party projects to test and enable free-threading support. Notable contributors include: Sam Gross, Ken Jin, Donghee Na, Itamar Oren, Matt Page, Brett Simmers, Dino Viehland, Carl Meyer, Nathan Goldbaum, Ralf Gommers, Lysandros Nikolaou, and many others. Many of these contributors are employed by Meta, which has provided significant engineering resources to support this project.

See also

PEP 703 [https://peps.python.org/pep-0703/] “Making the Global Interpreter Lock Optional in CPython” contains rationale and information surrounding this work.

[Porting Extension Modules to Support Free-Threading](https://py-free-threading.github.io/porting/) [https://py-free-threading.github.io/porting/]: A community-maintained porting guide for extension authors.

An experimental just-in-time (JIT) compiler

When CPython is configured and built using the **`--enable-experimental-jit`** option, a just-in-time (JIT) compiler is added which may speed up some Python programs. On Windows, use `PCbuild/build.bat --experimental-jit` to enable the JIT or `--experimental-jit-interpreter` to enable the Tier 2 interpreter. Build requirements and further supporting information [are contained at](https://github.com/python/cpython/blob/main/Tools/jit/README.md) [https://github.com/python/cpython/blob/main/Tools/jit/README.md] `Tools/jit/README.md`.

The **`--enable-experimental-jit`** option takes these (optional) values, defaulting to `yes` if **`--enable-experimental-jit`** is present without the optional value.

- `no`: Disable the entire Tier 2 and JIT pipeline.
- `yes`: Enable the JIT. To disable the JIT at runtime, pass the environment variable `PYTHON_JIT=0`.
- `yes-off`: Build the JIT but disable it by default. To enable the JIT at runtime, pass the environment variable `PYTHON_JIT=1`.
- `interpreter`: Enable the Tier 2 interpreter but disable the JIT. The interpreter can be disabled by running with `PYTHON_JIT=0`.

The internal architecture is roughly as follows:

- We start with specialized *Tier 1 bytecode*. See [What’s new in 3.11](#) for details.

- When the Tier 1 bytecode gets hot enough, it gets translated to a new purely internal intermediate representation (IR), called the *Tier 2 IR*, and sometimes referred to as micro-ops (“uops”).
- The Tier 2 IR uses the same stack-based virtual machine as Tier 1, but the instruction format is better suited to translation to machine code.
- We have several optimization passes for Tier 2 IR, which are applied before it is interpreted or translated to machine code.
- There is a Tier 2 interpreter, but it is mostly intended for debugging the earlier stages of the optimization pipeline. The Tier 2 interpreter can be enabled by configuring Python with `--enable-experimental-jit=interpreter`.
- When the JIT is enabled, the optimized Tier 2 IR is translated to machine code, which is then executed.
- The machine code translation process uses a technique called *copy-and-patch*. It has no runtime dependencies, but there is a new build-time dependency on LLVM.

See also

PEP 744 [https://peps.python.org/pep-0744/]

(JIT by Brandt Bucher, inspired by a paper by Haoran Xu and Fredrik Kjolstad. Tier 2 IR by Mark Shannon and Guido van Rossum. Tier 2 optimizer by Ken Jin.)

Defined mutation semantics for `locals()`

Historically, the expected result of mutating the return value of `locals()` has been left to individual Python implementations to define. Starting from Python 3.13, **PEP 667** [https://peps.python.org/pep-0667/] standardises the historical behavior of CPython for most code execution scopes, but changes optimized scopes (functions, generators, coroutines, comprehensions, and generator expressions) to explicitly return independent snapshots of the currently assigned local variables, including locally referenced nonlocal variables captured in closures.

This change to the semantics of `locals()` in optimized scopes also affects the default behavior of code execution functions that implicitly target `locals()` if no explicit namespace is provided (such as `exec()` and `eval()`). In previous versions, whether or not changes could be accessed by calling `locals()` after calling the code execution function was implementation-dependent. In CPython specifically, such code would typically appear to work as desired, but could sometimes fail in optimized scopes based on other code (including debuggers and code execution tracing tools) potentially resetting the shared snapshot in that scope. Now, the code will always run against an independent snapshot of the local variables in optimized scopes, and hence the changes will never be visible in subsequent calls to `locals()`. To access the changes made in these cases, an explicit namespace reference must now be passed to the relevant function. Alternatively, it may make sense to update affected code to use a higher level code execution API that returns the resulting code execution namespace (e.g. `runpy.run_path()` when executing Python files from disk).

To ensure debuggers and similar tools can reliably update local variables in scopes affected by this change, `FrameType.f_locals` now returns a write-through proxy to the frame’s local and locally referenced nonlocal variables in these scopes, rather than returning an inconsistently updated shared `dict` instance with undefined runtime semantics.

See **PEP 667** [https://peps.python.org/pep-0667/] for more details, including related C API changes and deprecations. Porting notes are also provided below for the affected Python APIs and C APIs.

(PEP and implementation contributed by Mark Shannon and Tian Gao in gh-74929 [https://github.com/python/cpython/issues/74929]. Documentation updates provided by Guido van Rossum and Alyssa Coghlan.)

Support for mobile platforms

PEP 730 [https://peps.python.org/pep-0730/]: iOS is now a **PEP 11** [https://peps.python.org/pep-0011/] supported platform, with the `arm64-apple-ios` and `arm64-apple-ios-simulator` targets at tier 3 (iPhone and iPad devices released after 2013 and the Xcode iOS simulator running on Apple silicon hardware, respectively). `x86_64-apple-ios-simulator` (the Xcode iOS simulator running on older `x86_64` hardware) is not a tier 3 supported platform, but will have best-effort support. (PEP written and implementation contributed by Russell Keith-Magee in gh-114099 [https://github.com/python/cpython/issues/114099].)

PEP 738 [https://peps.python.org/pep-0738/]: Android is now a **PEP 11** [https://peps.python.org/pep-0011/] supported platform, with the `aarch64-linux-android` and `x86_64-linux-android` targets at tier 3. The 32-bit targets `arm-linux-androideabi` and `i686-linux-android` are not tier 3 supported platforms, but will have best-effort support. (PEP written and implementation contributed by Malcolm Smith in gh-116622 [https://github.com/python/cpython/issues/116622].)

See also

PEP 730 [https://peps.python.org/pep-0730/], **PEP 738** [https://peps.python.org/pep-0738/]

Other Language Changes

- The compiler now strips common leading whitespace from every line in a docstring. This reduces the size of the bytecode cache (such as `.pyc` files), with reductions in file size of around 5%, for example in **`sqlalchemy.orm.session`** from SQLAlchemy 2.0. This change affects tools that use docstrings, such as **`doctest`**.

  ```
  >>> def spam():
  ...     """
  ...         This is a docstring with
  ...           leading whitespace.
  ...
  ...         It even has multiple paragraphs!
  ...     """
  ...
  >>> spam.__doc__
  '\nThis is a docstring with\n  leading whitespace.\n\nIt
  even has multiple paragraphs!\n'
  ```

  (Contributed by Inada Naoki in gh-81283 [https://github.com/python/cpython/issues/81283].)

- Annotation scopes within class scopes can now contain lambdas and comprehensions. Comprehensions that are located within class scopes are not inlined into their parent scope.

  ```
  class C[T]:
      type Alias = lambda: T
  ```

  (Contributed by Jelle Zijlstra in gh-109118 [https://github.com/python/cpython/issues/109118] and gh-118160

[https://github.com/python/cpython/issues/118160].)

- Future statements are no longer triggered by relative imports of the **`__future__`** module, meaning that statements of the form `from .__future__ import ...` are now simply standard relative imports, with no special features activated. (Contributed by Jeremiah Gabriel Pascual in gh-118216 [https://github.com/python/cpython/issues/118216].)

- **`global`** declarations are now permitted in **`except`** blocks when that global is used in the **`else`** block. Previously this raised an erroneous **`SyntaxError`**. (Contributed by Irit Katriel in gh-111123 [https://github.com/python/cpython/issues/111123].)

- Add **`PYTHON_FROZEN_MODULES`**, a new environment variable that determines whether frozen modules are ignored by the import machinery, equivalent to the **`-X frozen_modules`** command-line option. (Contributed by Yilei Yang in gh-111374 [https://github.com/python/cpython/issues/111374].)

- Add support for the perf profiler working without frame pointers [https://en.wikipedia.org/wiki/Call_stack] through the new environment variable **`PYTHON_PERF_JIT_SUPPORT`** and command-line option **`-X perf_jit`**. (Contributed by Pablo Galindo in gh-118518 [https://github.com/python/cpython/issues/118518].)

- The location of a `.python_history` file can be changed via the new **`PYTHON_HISTORY`** environment variable. (Contributed by Levi Sabah, Zackery Spytz and Hugo van Kemenade in gh-73965 [https://github.com/python/cpython/issues/73965].)

- Classes have a new `__static_attributes__` attribute. This is populated by the compiler with a tuple of the class's attribute names which are assigned through `self.<name>` from any function in its body. (Contributed by Irit Katriel in gh-115775 [https://github.com/python/cpython/issues/115775].)

- The compiler now creates a **`__firstlineno__`** attribute on classes with the line number of the first line of the class definition. (Contributed by Serhiy Storchaka in gh-118465 [https://github.com/python/cpython/issues/118465].)

- The `exec()` and `eval()` builtins now accept the *globals* and *locals* arguments as keywords. (Contributed by Raphael Gaschignard in gh-105879 [https://github.com/python/cpython/issues/105879])

- The `compile()` builtin now accepts a new flag, `ast.PyCF_OPTIMIZED_AST`, which is similar to `ast.PyCF_ONLY_AST` except that the returned AST is optimized according to the value of the *optimize* argument. (Contributed by Irit Katriel in gh-108113 [https://github.com/python/cpython/issues/108113]).

- Add a `__name__` attribute on `property` objects. (Contributed by Eugene Toder in gh-101860 [https://github.com/python/cpython/issues/101860].)

- Add `PythonFinalizationError`, a new exception derived from `RuntimeError` and used to signal when operations are blocked during finalization. The following callables now raise **`PythonFinalizationError`**, instead of `RuntimeError`:

    - `_thread.start_new_thread()`
    - `os.fork()`
    - `os.forkpty()`
    - `subprocess.Popen`

(Contributed by Victor Stinner in gh-114570 [https://github.com/python/cpython/issues/114570].)

- Allow the *count* argument of `str.replace()` to be a keyword. (Contributed by Hugo van Kemenade in gh-106487 [https://github.com/python/cpython/issues/106487].)
- Many functions now emit a warning if a boolean value is passed as a file descriptor argument. This can help catch some errors earlier. (Contributed by Serhiy Storchaka in gh-82626 [https://github.com/python/cpython/issues/82626].)
- Added **`name`** and **`mode`** attributes for compressed and archived file-like objects in the `bz2`, `lzma`, `tarfile`, and `zipfile` modules. (Contributed by Serhiy Storchaka in gh-115961 [https://github.com/python/cpython/issues/115961].)

New Modules

- `dbm.sqlite3`: An SQLite backend for `dbm`. (Contributed by Raymond Hettinger and Erlend E. Aasland in gh-100414 [https://github.com/python/cpython/issues/100414].)

Improved Modules

argparse

- Add the *deprecated* parameter to the `add_argument()` and **`add_parser()`** methods, to enable deprecating command-line options, positional arguments, and subcommands. (Contributed by Serhiy Storchaka in gh-83648 [https://github.com/python/cpython/issues/83648].)

array

- Add the `'w'` type code (`Py_UCS4`) for Unicode characters. It should be used instead of the deprecated `'u'` type code. (Contributed by Inada Naoki in gh-80480 [https://github.com/python/cpython/issues/80480].)
- Register **`array.array`** as a **`MutableSequence`** by implementing the **`clear()`** method. (Contributed by Mike Zimin in gh-114894 [https://github.com/python/cpython/issues/114894].)

ast

- The constructors of node types in the **`ast`** module are now stricter in the arguments they accept, with more intuitive behavior when arguments are omitted.

  If an optional field on an AST node is not included as an argument when constructing an instance, the field will now be set to `None`. Similarly, if a list field is omitted, that field will now be set to an empty list, and if an **`expr_context`** field is omitted, it defaults to **`Load()`**. (Previously, in all cases, the attribute would be missing on the newly constructed AST node instance.)

  In all other cases, where a required argument is omitted, the node constructor will emit a **`DeprecationWarning`**. This will raise an exception in Python 3.15. Similarly, passing a keyword argument to the constructor that does not map to a field on the AST node is now deprecated, and will raise an exception in Python 3.15.

  These changes do not apply to user-defined subclasses of **`ast.AST`** unless the class opts in to the new behavior by defining the **`AST._field_types`** mapping.

  (Contributed by Jelle Zijlstra in gh-105858 [https://github.com/python/cpython/issues/105858], gh-117486 [https://github.com/python/cpython/issues/117486], and gh-118851 [https://github.com/python/cpython/issues/118851].)

- `ast.parse()` now accepts an optional argument *optimize* which is passed on to `compile()`. This makes it possible to obtain an optimized AST. (Contributed by Irit Katriel in gh-108113 [https://github.com/python/cpython/issues/108113].)

asyncio

- `asyncio.as_completed()` now returns an object that is both an asynchronous iterator and a plain iterator of awaitables. The awaitables yielded by asynchronous iteration include original task or future objects that were passed in, making it easier to associate results with the tasks being completed. (Contributed by Justin Arthur in gh-77714 [https://github.com/python/cpython/issues/77714].)

- `asyncio.loop.create_unix_server()` will now automatically remove the Unix socket when the server is closed. (Contributed by Pierre Ossman in gh-111246 [https://github.com/python/cpython/issues/111246].)

- `DatagramTransport.sendto()` will now send zero-length datagrams if called with an empty bytes object. The transport flow control also now accounts for the datagram header when calculating the buffer size. (Contributed by Jamie Phan in gh-115199 [https://github.com/python/cpython/issues/115199].)

- Add `Queue.shutdown` and `QueueShutDown` to manage queue termination. (Contributed by Laurie Opperman and Yves Duprat in gh-104228 [https://github.com/python/cpython/issues/104228].)

- Add the [`Server.close_clients()`] and [`Server.abort_clients()`] methods, which more forcefully close an asyncio server. (Contributed by Pierre Ossman in gh-113538 [https://github.com/python/cpython/issues/113538].)

- Accept a tuple of separators in [`StreamReader.readuntil()`], stopping when any one of them is encountered. (Contributed by Bruce Merry in gh-81322 [https://github.com/python/cpython/issues/81322].)

- Improve the behavior of [`TaskGroup`] when an external cancellation collides with an internal cancellation. For example, when two task groups are nested and both experience an exception in a child task simultaneously, it was possible that the outer task group would hang, because its internal cancellation was swallowed by the inner task group.

  In the case where a task group is cancelled externally and also must raise an [`ExceptionGroup`], it will now call the parent task's [`cancel()`] method. This ensures that a [`CancelledError`] will be raised at the next [`await`], so the cancellation is not lost.

  An added benefit of these changes is that task groups now preserve the cancellation count ([`cancelling()`]).

  In order to handle some corner cases, [`uncancel()`] may now reset the undocumented `_must_cancel` flag when the cancellation count reaches zero.

  (Inspired by an issue reported by Arthur Tacca in gh-116720 [https://github.com/python/cpython/issues/116720].)

- When [`TaskGroup.create_task()`] is called on an inactive [`TaskGroup`], the given coroutine will be closed (which prevents a [`RuntimeWarning`] about the given coroutine being never awaited). (Contributed by Arthur Tacca and Jason Zhang in gh-115957 [https://github.com/python/cpython/issues/115957].)

- The function and methods named `create_task` have received a new `**kwargs` argument that is passed through to the task constructor. This change was accidentally added in 3.13.3, and broke the API contract for custom task factories. Several third-party task factories implemented workarounds for this. In 3.13.4 and later releases the old factory contract is honored once again (until 3.14). To keep the workarounds working, the extra `**kwargs` argument still allows passing additional keyword arguments to **`Task`** and to custom task factories.

  This affects the following function and methods: **`asyncio.create_task()`**, **`asyncio.loop.create_task()`**, **`asyncio.TaskGroup.create_task()`**. (Contributed by Thomas Grainger in gh-128307 [https://github.com/python/cpython/issues/128307].)

base64

- Add **`z85encode()`** and **`z85decode()`** functions for encoding **`bytes`** as Z85 data [https://rfc.zeromq.org/spec/32/] and decoding Z85-encoded data to **`bytes`**. (Contributed by Matan Perelman in gh-75299 [https://github.com/python/cpython/issues/75299].)

compileall

- The default number of worker threads and processes is now selected using **`os.process_cpu_count()`** instead of **`os.cpu_count()`**. (Contributed by Victor Stinner in gh-109649 [https://github.com/python/cpython/issues/109649].)

concurrent.futures

- The default number of worker threads and processes is now selected using `os.process_cpu_count()` instead of `os.cpu_count()`. (Contributed by Victor Stinner in gh-109649 [https://github.com/python/cpython/issues/109649].)

configparser

- `ConfigParser` now has support for unnamed sections, which allows for top-level key-value pairs. This can be enabled with the new *allow_unnamed_section* parameter. (Contributed by Pedro Sousa Lacerda in gh-66449 [https://github.com/python/cpython/issues/66449].)

copy

- The new `replace()` function and the `replace protocol` make creating modified copies of objects much simpler. This is especially useful when working with immutable objects. The following types support the `replace()` function and implement the replace protocol:

    - `collections.namedtuple()`
    - `dataclasses.dataclass`
    - `datetime.datetime`, `datetime.date`, `datetime.time`
    - `inspect.Signature`, `inspect.Parameter`
    - `types.SimpleNamespace`
    - code objects

  Any user-defined class can also support `copy.replace()` by defining the `__replace__()` method. (Contributed by Serhiy Storchaka in gh-108751 [https://github.com/python/cpython/issues/108751].)

ctypes

- As a consequence of necessary internal refactoring, initialization of internal metaclasses now happens in `__init__` rather than in `__new__`. This affects projects that subclass these internal metaclasses to provide custom initialization. Generally:

  - Custom logic that was done in `__new__` after calling `super().__new__` should be moved to `__init__`.
  - To create a class, call the metaclass, not only the metaclass's `__new__` method.

  See gh-124520 [https://github.com/python/cpython/issues/124520] for discussion and links to changes in some affected projects.

- **`ctypes.Structure`** objects have a new **`_align_`** attribute which allows the alignment of the structure being packed to/from memory to be specified explicitly. (Contributed by Matt Sanderson in gh-112433 [https://github.com/python/cpython/issues/112433])

dbm

- Add **`dbm.sqlite3`**, a new module which implements an SQLite backend, and make it the default **`dbm`** backend. (Contributed by Raymond Hettinger and Erlend E. Aasland in gh-100414 [https://github.com/python/cpython/issues/100414].)
- Allow removing all items from the database through the new **`clear()`** methods of the GDBM and NDBM database objects. (Contributed by Donghee Na in gh-107122 [https://github.com/python/cpython/issues/107122].)

dis

- Change the output of **`dis`** module functions to show logical labels for jump targets and exception handlers, rather than offsets. The offsets can be added with the new **`-O`** command-line option or the *show_offsets* argument. (Contributed by Irit Katriel in gh-112137 [https://github.com/python/cpython/issues/112137].)
- **`get_instructions()`** no longer represents cache entries as separate instructions. Instead, it returns them as part of the **`Instruction`**, in the new *cache_info* field. The *show_caches* argument to **`get_instructions()`** is deprecated and no longer has any effect. (Contributed by Irit Katriel in gh-112962 [https://github.com/python/cpython/issues/112962].)

doctest

- **`doctest`** output is now colored by default. This can be controlled via the new **`PYTHON_COLORS`** environment variable as well as the canonical **`NO_COLOR`** [https://no-color.org/] and **`FORCE_COLOR`** [https://force-color.org/] environment variables. See also Controlling color. (Contributed by Hugo van Kemenade in gh-117225 [https://github.com/python/cpython/issues/117225].)
- The **`DocTestRunner.run()`** method now counts the number of skipped tests. Add the **`DocTestRunner.skips`** and **`TestResults.skipped`** attributes. (Contributed by Victor Stinner in gh-108794 [https://github.com/python/cpython/issues/108794].)

email

- Headers with embedded newlines are now quoted on output. The **`generator`** will now refuse to serialize (write) headers that are improperly folded or delimited, such that they would be parsed as multiple headers or joined with adjacent data. If you need to turn this safety feature off, set **`verify_generated_headers`**. (Contributed by Bas Bloemsaat and Petr Viktorin in gh-121650 [https://github.com/python/cpython/issues/121650].)

- **`getaddresses()`** and **`parseaddr()`** now return `('', '')` pairs in more situations where invalid email addresses are encountered instead of potentially inaccurate values. The two functions have a new optional *strict* parameter (default `True`). To get the old behavior (accepting malformed input), use `strict=False`. `getattr(email.utils, 'supports_strict_parsing', False)` can be used to check if the *strict* parameter is available. (Contributed by Thomas Dwyer and Victor Stinner for gh-102988 [https://github.com/python/cpython/issues/102988] to improve the **CVE 2023-27043** [https://www.cve.org/CVERecord?id=CVE-2023-27043] fix.)

enum

- **`EnumDict`** has been made public to better support subclassing **`EnumType`**.

fractions

- **`Fraction`** objects now support the standard format specification mini-language rules for fill, alignment, sign handling, minimum width, and grouping. (Contributed by Mark Dickinson in gh-111320 [https://github.com/python/cpython/issues/111320].)

glob

- Add **`translate()`**, a function to convert a path specification with shell-style wildcards to a regular expression. (Contributed by Barney Gale in gh-72904 [https://github.com/python/cpython/issues/72904].)

importlib

- The following functions in `importlib.resources` now allow accessing a directory (or tree) of resources, using multiple positional arguments (the *encoding* and *errors* arguments in the text-reading functions are now keyword-only):

  - `is_resource()`
  - `open_binary()`
  - `open_text()`
  - `path()`
  - `read_binary()`
  - `read_text()`

  These functions are no longer deprecated and are not scheduled for removal. (Contributed by Petr Viktorin in gh-116608 [https://github.com/python/cpython/issues/116608].)

- `contents()` remains deprecated in favor of the fully-featured `Traversable` API. However, there is now no plan to remove it. (Contributed by Petr Viktorin in gh-116608 [https://github.com/python/cpython/issues/116608].)

io

- The `IOBase` finalizer now logs any errors raised by the `close()` method with `sys.unraisablehook`. Previously, errors were ignored silently by default, and only logged in Python Development Mode or when using a Python debug build. (Contributed by Victor Stinner in gh-62948 [https://github.com/python/cpython/issues/62948].)

ipaddress

- Add the **`IPv4Address.ipv6_mapped`** property, which returns the IPv4-mapped IPv6 address. (Contributed by Charles Machalow in gh-109466 [https://github.com/python/cpython/issues/109466].)
- Fix `is_global` and `is_private` behavior in **`IPv4Address`**, **`IPv6Address`**, **`IPv4Network`**, and **`IPv6Network`**. (Contributed by Jakub Stasiak in gh-113171 [https://github.com/python/cpython/issues/113171].)

itertools

- **`batched()`** has a new *strict* parameter, which raises a **`ValueError`** if the final batch is shorter than the specified batch size. (Contributed by Raymond Hettinger in gh-113202 [https://github.com/python/cpython/issues/113202].)

marshal

- Add the *allow_code* parameter in module functions. Passing `allow_code=False` prevents serialization and de-serialization of code objects which are incompatible between Python versions. (Contributed by Serhiy Storchaka in gh-113626 [https://github.com/python/cpython/issues/113626].)

math

- The new function **`fma()`** performs fused multiply-add operations. This computes `x * y + z` with only a single round, and so avoids any intermediate loss of precision. It wraps the `fma()` function provided by C99, and follows the specification of the IEEE 754 “fusedMultiplyAdd” operation for special cases. (Contributed by Mark Dickinson and Victor Stinner in gh-73468 [https://github.com/python/cpython/issues/73468].)

mimetypes

- Add the **`guess_file_type()`** function to guess a MIME type from a filesystem path. Using paths with **`guess_type()`** is now soft deprecated. (Contributed by Serhiy Storchaka in gh-66543 [https://github.com/python/cpython/issues/66543].)

mmap

- **`mmap`** is now protected from crashing on Windows when the mapped memory is inaccessible due to file system errors or access violations. (Contributed by Jannis Weigend in gh-118209 [https://github.com/python/cpython/issues/118209].)
- **`mmap`** has a new **`seekable()`** method that can be used when a seekable file-like object is required. The **`seek()`** method now returns the new absolute position. (Contributed by Donghee Na and Sylvie Liberman in gh-111835 [https://github.com/python/cpython/issues/111835].)
- The new UNIX-only *trackfd* parameter for **`mmap`** controls file descriptor duplication; if false, the file descriptor specified by *fileno* will not be duplicated. (Contributed by Zackery Spytz and Petr Viktorin in gh-78502 [https://github.com/python/cpython/issues/78502].)

multiprocessing

- The default number of worker threads and processes is now selected using **`os.process_cpu_count()`** instead of **`os.cpu_count()`**. (Contributed by Victor Stinner in gh-109649 [https://github.com/python/cpython/issues/109649].)

os

- Add `process_cpu_count()` function to get the number of logical CPU cores usable by the calling thread of the current process. (Contributed by Victor Stinner in gh-109649 [https://github.com/python/cpython/issues/109649].)
- `cpu_count()` and `process_cpu_count()` can be overridden through the new environment variable `PYTHON_CPU_COUNT` or the new command-line option `-X cpu_count`. This option is useful for users who need to limit CPU resources of a container system without having to modify application code or the container itself. (Contributed by Donghee Na in gh-109595 [https://github.com/python/cpython/issues/109595].)
- Add a low level interface to Linux's *timer file descriptors [https://manpages.debian.org/timerfd_create(2)]* via `timerfd_create()`, `timerfd_settime()`, `timerfd_settime_ns()`, `timerfd_gettime()`, `timerfd_gettime_ns()`, `TFD_NONBLOCK`, `TFD_CLOEXEC`, `TFD_TIMER_ABSTIME`, and `TFD_TIMER_CANCEL_ON_SET` (Contributed by Masaru Tsuchiyama in gh-108277 [https://github.com/python/cpython/issues/108277].)
- `lchmod()` and the *follow_symlinks* argument of `chmod()` are both now available on Windows. Note that the default value of *follow_symlinks* in **`lchmod()`** is `False` on Windows. (Contributed by Serhiy Storchaka in gh-59616 [https://github.com/python/cpython/issues/59616].)
- `fchmod()` and support for file descriptors in `chmod()` are both now available on Windows. (Contributed by Serhiy Storchaka in gh-113191 [https://github.com/python/cpython/issues/113191].)
- On Windows, `mkdir()` and `makedirs()` now support passing a *mode* value of `0o700` to apply access control to the new directory. This implicitly affects `tempfile.mkdtemp()` and is a mitigation for **CVE 2024-4030** [https://www.cve.org/CVERecord?id=CVE-2024-4030]. Other values for *mode* continue to be ignored. (Contributed by Steve Dower in gh-118486 [https://github.com/python/cpython/issues/118486].)

- **`posix_spawn()`** now accepts `None` for the *env* argument, which makes the newly spawned process use the current process environment. (Contributed by Jakub Kulik in gh-113119 [https://github.com/python/cpython/issues/113119].)
- **`posix_spawn()`** can now use the **`POSIX_SPAWN_CLOSEFROM`** attribute in the *file_actions* parameter on platforms that support **`posix_spawn_file_actions_addclosefrom_np()`**. (Contributed by Jakub Kulik in gh-113117 [https://github.com/python/cpython/issues/113117].)

os.path

- Add **`isreserved()`** to check if a path is reserved on the current system. This function is only available on Windows. (Contributed by Barney Gale in gh-88569 [https://github.com/python/cpython/issues/88569].)
- On Windows, **`isabs()`** no longer considers paths starting with exactly one slash (`\` or `/`) to be absolute. (Contributed by Barney Gale and Jon Foster in gh-44626 [https://github.com/python/cpython/issues/44626].)
- **`realpath()`** now resolves MS-DOS style file names even if the file is not accessible. (Contributed by Moonsik Park in gh-82367 [https://github.com/python/cpython/issues/82367].)

pathlib

- Add **`UnsupportedOperation`**, which is raised instead of **`NotImplementedError`** when a path operation isn’t supported. (Contributed by Barney Gale in gh-89812 [https://github.com/python/cpython/issues/89812].)
- Add a new constructor for creating **`Path`** objects from ‘file’ URIs (`file:///`), **`Path.from_uri()`**. (Contributed by Barney Gale in gh-107465 [https://github.com/python/cpython/issues/107465].)

- Add **`PurePath.full_match()`** for matching paths with shell-style wildcards, including the recursive wildcard “`**`”. (Contributed by Barney Gale in gh-73435 [https://github.com/python/cpython/issues/73435].)
- Add the **`PurePath.parser`** class attribute to store the implementation of **`os.path`** used for low-level path parsing and joining. This will be either **`posixpath`** or **`ntpath`**.
- Add *recurse_symlinks* keyword-only argument to **`Path.glob()`** and **`rglob()`**. (Contributed by Barney Gale in gh-77609 [https://github.com/python/cpython/issues/77609].)
- **`Path.glob()`** and **`rglob()`** now return files and directories when given a pattern that ends with “`**`”. Previously, only directories were returned. (Contributed by Barney Gale in gh-70303 [https://github.com/python/cpython/issues/70303].)
- Add the *follow_symlinks* keyword-only argument to **`Path.is_file`**, **`Path.is_dir`**, **`Path.owner()`**, and **`Path.group()`**. (Contributed by Barney Gale in gh-105793 [https://github.com/python/cpython/issues/105793] and Kamil Turek in gh-107962 [https://github.com/python/cpython/issues/107962].)

pdb

- **`breakpoint()`** and **`set_trace()`** now enter the debugger immediately rather than on the next line of code to be executed. This change prevents the debugger from breaking outside of the context when **`breakpoint()`** is positioned at the end of the context. (Contributed by Tian Gao in gh-118579 [https://github.com/python/cpython/issues/118579].)
- `sys.path[0]` is no longer replaced by the directory of the script being debugged when **`sys.flags.safe_path`** is set. (Contributed by Tian Gao and Christian Walther in gh-111762 [https://github.com/python/cpython/issues/111762].)
- **`zipapp`** is now supported as a debugging target. (Contributed by Tian Gao in gh-118501

[https://github.com/python/cpython/issues/118501].)

- Add ability to move between chained exceptions during post-mortem debugging in **`pm()`** using the new **`exceptions [exc_number]`** command for Pdb. (Contributed by Matthias Bussonnier in gh-106676 [https://github.com/python/cpython/issues/106676].)
- Expressions and statements whose prefix is a pdb command are now correctly identified and executed. (Contributed by Tian Gao in gh-108464 [https://github.com/python/cpython/issues/108464].)

queue

- Add **`Queue.shutdown`** and **`ShutDown`** to manage queue termination. (Contributed by Laurie Opperman and Yves Duprat in gh-104750 [https://github.com/python/cpython/issues/104750].)

random

- Add a command-line interface. (Contributed by Hugo van Kemenade in gh-118131 [https://github.com/python/cpython/issues/118131].)

re

- Rename **`re.error`** to **`PatternError`** for improved clarity. **`re.error`** is kept for backward compatibility.

shutil

- Support the *dir_fd* and *follow_symlinks* keyword arguments in **`chown()`**. (Contributed by Berker Peksag and Tahia K in gh-62308 [https://github.com/python/cpython/issues/62308])

site

- `.pth` files are now decoded using UTF-8 first, and then with the locale encoding if UTF-8 decoding fails. (Contributed by Inada Naoki in gh-117802 [https://github.com/python/cpython/issues/117802].)

sqlite3

- A **`ResourceWarning`** is now emitted if a **`Connection`** object is not **`closed`** explicitly. (Contributed by Erlend E. Aasland in gh-105539 [https://github.com/python/cpython/issues/105539].)
- Add the *filter* keyword-only parameter to **`Connection.iterdump()`** for filtering database objects to dump. (Contributed by Mariusz Felisiak in gh-91602 [https://github.com/python/cpython/issues/91602].)

ssl

- The **`create_default_context()`** API now includes **`VERIFY_X509_PARTIAL_CHAIN`** and **`VERIFY_X509_STRICT`** in its default flags.

  Note

  `VERIFY_X509_STRICT` may reject pre-**RFC 5280** [https://datatracker.ietf.org/doc/html/rfc5280.html] or malformed certificates that the underlying OpenSSL implementation might otherwise accept. Whilst disabling this is not recommended, you can do so using:

  ```
  import ssl
  ```

```
ctx = ssl.create_default_context()
ctx.verify_flags &= ~ssl.VERIFY_X509_STRICT
```

(Contributed by William Woodruff in gh-112389 [https://github.com/python/cpython/issues/112389].)

statistics

- Add **`kde()`** for kernel density estimation. This makes it possible to estimate a continuous probability density function from a fixed number of discrete samples. (Contributed by Raymond Hettinger in gh-115863 [https://github.com/python/cpython/issues/115863].)
- Add **`kde_random()`** for sampling from an estimated probability density function created by **`kde()`**. (Contributed by Raymond Hettinger in gh-115863 [https://github.com/python/cpython/issues/115863].)

subprocess

- The **`subprocess`** module now uses the **`posix_spawn()`** function in more situations.

  Notably, when *close_fds* is `True` (the default), **`posix_spawn()`** will be used when the C library provides **`posix_spawn_file_actions_addclosefrom_np()`**, which includes recent versions of Linux, FreeBSD, and Solaris. On Linux, this should perform similarly to the existing Linux **`vfork()`** based code.

  A private control knob **`subprocess._USE_POSIX_SPAWN`** can be set to `False` if you need to force **`subprocess`** to never use **`posix_spawn()`**. Please report your reason and platform details in the issue tracker if you set this so that we can improve our API selection logic for everyone.

(Contributed by Jakub Kulik in gh-113117 [https://github.com/python/cpython/issues/113117].)

sys

- Add the `_is_interned()` function to test if a string was interned. This function is not guaranteed to exist in all implementations of Python. (Contributed by Serhiy Storchaka in gh-78573 [https://github.com/python/cpython/issues/78573].)

tempfile

- On Windows, the default mode `0o700` used by `tempfile.mkdtemp()` now limits access to the new directory due to changes to `os.mkdir()`. This is a mitigation for **CVE 2024-4030** [https://www.cve.org/CVERecord?id=CVE-2024-4030]. (Contributed by Steve Dower in gh-118486 [https://github.com/python/cpython/issues/118486].)

time

- On Windows, `monotonic()` now uses the `QueryPerformanceCounter()` clock for a resolution of 1 microsecond, instead of the `GetTickCount64()` clock which has a resolution of 15.6 milliseconds. (Contributed by Victor Stinner in gh-88494 [https://github.com/python/cpython/issues/88494].)
- On Windows, `time()` now uses the `GetSystemTimePreciseAsFileTime()` clock for a resolution of 1 microsecond, instead of the `GetSystemTimeAsFileTime()` clock which has a resolution of 15.6 milliseconds. (Contributed by Victor Stinner in gh-63207 [https://github.com/python/cpython/issues/63207].)

tkinter

- Add **`tkinter`** widget methods: **`tk_busy_hold()`**, **`tk_busy_configure()`**, **`tk_busy_cget()`**, **`tk_busy_forget()`**, **`tk_busy_current()`**, and **`tk_busy_status()`**. (Contributed by Miguel, klappnase and Serhiy Storchaka in gh-72684 [https://github.com/python/cpython/issues/72684].)
- The **`tkinter`** widget method **`wm_attributes()`** now accepts the attribute name without the minus prefix to get window attributes, for example `w.wm_attributes('alpha')` and allows specifying attributes and values to set as keyword arguments, for example `w.wm_attributes(alpha=0.5)`. (Contributed by Serhiy Storchaka in gh-43457 [https://github.com/python/cpython/issues/43457].)
- **`wm_attributes()`** can now return attributes as a **`dict`**, by using the new optional keyword-only parameter *return_python_dict*. (Contributed by Serhiy Storchaka in gh-43457 [https://github.com/python/cpython/issues/43457].)
- **`Text.count()`** can now return a simple **`int`** when the new optional keyword-only parameter *return_ints* is used. Otherwise, the single count is returned as a 1-tuple or `None`. (Contributed by Serhiy Storchaka in gh-97928 [https://github.com/python/cpython/issues/97928].)
- Support the “vsapi” element type in the **`element_create()`** method of **`tkinter.ttk.Style`**. (Contributed by Serhiy Storchaka in gh-68166 [https://github.com/python/cpython/issues/68166].)
- Add the **`after_info()`** method for Tkinter widgets. (Contributed by Cheryl Sabella in gh-77020 [https://github.com/python/cpython/issues/77020].)
- Add a new **`copy_replace()`** method to **`PhotoImage`** to copy a region from one image to another, possibly with pixel zooming, subsampling, or both. (Contributed by Serhiy

Storchaka in gh-118225 [https://github.com/python/cpython/issues/118225].)

- Add *from_coords* parameter to the **`PhotoImage`** methods **`copy()`**, **`zoom()`** and **`subsample()`**. Add *zoom* and *subsample* parameters to the **`PhotoImage`** method **`copy()`**. (Contributed by Serhiy Storchaka in gh-118225 [https://github.com/python/cpython/issues/118225].)
- Add the **`PhotoImage`** methods **`read()`** to read an image from a file and **`data()`** to get the image data. Add *background* and *grayscale* parameters to the **`write()`** method. (Contributed by Serhiy Storchaka in gh-118271 [https://github.com/python/cpython/issues/118271].)

traceback

- Add the **`exc_type_str`** attribute to **`TracebackException`**, which holds a string display of the *exc_type*. Deprecate the **`exc_type`** attribute, which holds the type object itself. Add parameter *save_exc_type* (default `True`) to indicate whether `exc_type` should be saved. (Contributed by Irit Katriel in gh-112332 [https://github.com/python/cpython/issues/112332].)
- Add a new *show_group* keyword-only parameter to **`TracebackException.format_exception_only()`** to (recursively) format the nested exceptions of a **`BaseExceptionGroup`** instance. (Contributed by Irit Katriel in gh-105292 [https://github.com/python/cpython/issues/105292].)

types

- **`SimpleNamespace`** can now take a single positional argument to initialise the namespace’s arguments. This argument must either be a mapping or an iterable of key-value pairs. (Contributed by Serhiy Storchaka in gh-108191 [https://github.com/python/cpython/issues/108191].)

typing

- **PEP 705** [https://peps.python.org/pep-0705/]: Add **`ReadOnly`**, a special typing construct to mark a **`TypedDict`** item as read-only for type checkers.
- **PEP 742** [https://peps.python.org/pep-0742/]: Add **`TypeIs`**, a typing construct that can be used to instruct a type checker how to narrow a type.
- Add **`NoDefault`**, a sentinel object used to represent the defaults of some parameters in the **`typing`** module. (Contributed by Jelle Zijlstra in gh-116126 [https://github.com/python/cpython/issues/116126].)
- Add **`get_protocol_members()`** to return the set of members defining a **`typing.Protocol`**. (Contributed by Jelle Zijlstra in gh-104873 [https://github.com/python/cpython/issues/104873].)
- Add **`is_protocol()`** to check whether a class is a **`Protocol`**. (Contributed by Jelle Zijlstra in gh-104873 [https://github.com/python/cpython/issues/104873].)
- **`ClassVar`** can now be nested in **`Final`**, and vice versa. (Contributed by Mehdi Drissi in gh-89547 [https://github.com/python/cpython/issues/89547].)

unicodedata

- Update the Unicode database to version 15.1.0 [https://www.unicode.org/versions/Unicode15.1.0/]. (Contributed by James Gerity in gh-109559 [https://github.com/python/cpython/issues/109559].)

venv

- Add support for creating source control management (SCM) ignore files in a virtual environment’s directory. By default, Git is supported. This is implemented as opt-in via the API, which can be extended to support other

SCMs (`EnvBuilder` and `create()`), and opt-out via the CLI, using **`--without-scm-ignore-files`**. (Contributed by Brett Cannon in gh-108125 [https://github.com/python/cpython/issues/108125].)

warnings

- **PEP 702** [https://peps.python.org/pep-0702/]: The new `warnings.deprecated()` decorator provides a way to communicate deprecations to a static type checker and to warn on usage of deprecated classes and functions. A `DeprecationWarning` may also be emitted when a decorated function or class is used at runtime. (Contributed by Jelle Zijlstra in gh-104003 [https://github.com/python/cpython/issues/104003].)

xml

- Allow controlling Expat >=2.6.0 reparse deferral (**CVE 2023-52425** [https://www.cve.org/CVERecord?id=CVE-2023-52425]) by adding five new methods:

    - `xml.etree.ElementTree.XMLParser.flush()`
    - `xml.etree.ElementTree.XMLPullParser.flush()`
    - `xml.parsers.expat.xmlparser.GetReparseDeferralEnabled()`
    - `xml.parsers.expat.xmlparser.SetReparseDeferralEnabled()`
    - **`xml.sax.expatreader.ExpatParser.flush()`**

  (Contributed by Sebastian Pipping in gh-115623 [https://github.com/python/cpython/issues/115623].)

- Add the **`close()`** method for the iterator returned by `iterparse()` for explicit cleanup. (Contributed by Serhiy Storchaka in gh-69893 [https://github.com/python/cpython/issues/69893].)

zipimport

- Add support for ZIP64 [https://en.wikipedia.org/wiki/Zip_(file_format)#ZIP64] format files. Everybody loves huge data, right? (Contributed by Tim Hatch in gh-94146 [https://github.com/python/cpython/issues/94146].)

Optimizations

- Several standard library modules have had their import times significantly improved. For example, the import time of the **`typing`** module has been reduced by around a third by removing dependencies on **`re`** and **`contextlib`**. Other modules to enjoy import-time speedups include **`email.utils`**, **`enum`**, **`functools`**, **`importlib.metadata`**, and **`threading`**. (Contributed by Alex Waygood, Shantanu Jain, Adam Turner, Daniel Hollas, and others in gh-109653 [https://github.com/python/cpython/issues/109653].)
- **`textwrap.indent()`** is now around 30% faster than before for large input. (Contributed by Inada Naoki in gh-107369 [https://github.com/python/cpython/issues/107369].)
- The **`subprocess`** module now uses the **`posix_spawn()`** function in more situations, including when *close_fds* is `True` (the default) on many modern platforms. This should provide a notable performance increase when launching processes on FreeBSD and Solaris. See the subprocess section above for details. (Contributed by Jakub Kulik in gh-113117 [https://github.com/python/cpython/issues/113117].)

Removed Modules And APIs

PEP 594: Remove “dead batteries” from the standard library

PEP 594 [https://peps.python.org/pep-0594/] proposed removing 19 modules from the standard library, colloquially referred to as ‘dead batteries’ due to their historic, obsolete, or insecure status. All of the following modules were deprecated in Python 3.11, and are now removed:

- **`aifc`**
  - standard-aifc [https://pypi.org/project/standard-aifc/]: Use the redistribution of `aifc` library from PyPI.
- **`audioop`**
  - audioop-lts [https://pypi.org/project/audioop-lts/]: Use `audioop-lts` library from PyPI.
- **`chunk`**
  - standard-chunk [https://pypi.org/project/standard-chunk/]: Use the redistribution of `chunk` library from PyPI.
- **`cgi`** and **`cgitb`**
  - **`cgi.FieldStorage`** can typically be replaced with **`urllib.parse.parse_qsl()`** for `GET` and `HEAD` requests, and the **`email.message`** module or the multipart [https://pypi.org/project/multipart/] library for `POST` and `PUT` requests.
  - **`cgi.parse()`** can be replaced by calling **`urllib.parse.parse_qs()`** directly on the desired query string, unless the input is `multipart/form-data`, which should be replaced as described below for **`cgi.parse_multipart()`**.

  - **`cgi.parse_header()`** can be replaced with the functionality in the **`email`** package, which implements the same MIME RFCs. For example, with **`email.message.EmailMessage`**:

    ```
    from email.message import EmailMessage

    msg = EmailMessage()
    msg['content-type'] = 'application/json;
    charset="utf8"'
    main, params = msg.get_content_type(), msg['content-
    type'].params
    ```

  - **`cgi.parse_multipart()`** can be replaced with the functionality in the **`email`** package, which implements the same MIME RFCs, or with the multipart [https://pypi.org/project/multipart/] library. For example, the **`email.message.EmailMessage`** and **`email.message.Message`** classes.

  - standard-cgi [https://pypi.org/project/standard-cgi/]: and standard-cgitb [https://pypi.org/project/standard-cgitb/]: Use the redistribution of `cgi` and `cgitb` library from PyPI.

- **`crypt`** and the private **`_crypt`** extension. The **`hashlib`** module may be an appropriate replacement when simply hashing a value is required. Otherwise, various third-party libraries on PyPI are available:

  - bcrypt [https://pypi.org/project/bcrypt/]: Modern password hashing for your software and your servers.
  - argon2-cffi [https://pypi.org/project/argon2-cffi/]: The secure Argon2 password hashing algorithm.
  - legacycrypt [https://pypi.org/project/legacycrypt/]: **`ctypes`** wrapper to the POSIX crypt library call and associated functionality.
  - crypt_r [https://pypi.org/project/crypt_r/]: Fork of the **`crypt`** module, wrapper to the *crypt_r(3)*

  [https://manpages.debian.org/crypt_r(3)] library call and associated functionality.
  - standard-crypt [https://pypi.org/project/standard-crypt/] and deprecated-crypt-alternative [https://pypi.org/project/deprecated-crypt-alternative/]: Use the redistribution of `crypt` and reimplementation of `_crypt` libraries from PyPI.

- **`imghdr`**: The filetype [https://pypi.org/project/filetype/], puremagic [https://pypi.org/project/puremagic/], or python-magic [https://pypi.org/project/python-magic/] libraries should be used as replacements. For example, the **`puremagic.what()`** function can be used to replace the **`imghdr.what()`** function for all file formats that were supported by **`imghdr`**.

  - standard-imghdr [https://pypi.org/project/standard-imghdr/]: Use the redistribution of `imghdr` library from PyPI.

- **`mailcap`**: Use the **`mimetypes`** module instead.

  - standard-mailcap [https://pypi.org/project/standard-mailcap/]: Use the redistribution of `mailcap` library from PyPI.

- **`msilib`**

- **`nis`**

- **`nntplib`**: Use the pynntp [https://pypi.org/project/pynntp/] library from PyPI instead.

  - standard-nntplib [https://pypi.org/project/standard-nntplib/]: Use the redistribution of `nntplib` library from PyPI.

- **`ossaudiodev`**: For audio playback, use the pygame [https://pypi.org/project/pygame/] library from PyPI instead.

- **`pipes`**: Use the [`subprocess`] module instead. Use [`shlex.quote()`] to replace the undocumented `pipes.quote` function.
    - standard-pipes [https://pypi.org/project/standard-pipes/]: Use the redistribution of `pipes` library from PyPI.
- **`sndhdr`**: The filetype [https://pypi.org/project/filetype/], puremagic [https://pypi.org/project/puremagic/], or python-magic [https://pypi.org/project/python-magic/] libraries should be used as replacements.
    - standard-sndhdr [https://pypi.org/project/standard-sndhdr/]: Use the redistribution of `sndhdr` library from PyPI.
- **`spwd`**: Use the python-pam [https://pypi.org/project/python-pam/] library from PyPI instead.
- **`sunau`**
    - standard-sunau [https://pypi.org/project/standard-sunau/]: Use the redistribution of `sunau` library from PyPI.
- **`telnetlib`**, Use the telnetlib3 [https://pypi.org/project/telnetlib3/] or Exscript [https://pypi.org/project/Exscript/] libraries from PyPI instead.
    - standard-telnetlib [https://pypi.org/project/standard-telnetlib/]: Use the redistribution of `telnetlib` library from PyPI.
- **`uu`**: Use the **`base64`** module instead, as a modern alternative.
    - standard-uu [https://pypi.org/project/standard-uu/]: Use the redistribution of `uu` library from PyPI.
- **`xdrlib`**

    - standard-xdrlib [https://pypi.org/project/standard-xdrlib/]: Use the redistribution of `xdrlib` library from PyPI.

(Contributed by Victor Stinner and Zachary Ware in gh-104773 [https://github.com/python/cpython/issues/104773] and gh-104780 [https://github.com/python/cpython/issues/104780].)

2to3

- Remove the **2to3** program and the **`lib2to3`** module, previously deprecated in Python 3.11. (Contributed by Victor Stinner in gh-104780 [https://github.com/python/cpython/issues/104780].)

builtins

- Remove support for chained **`classmethod`** descriptors (introduced in gh-63272 [https://github.com/python/cpython/issues/63272]). These can no longer be used to wrap other descriptors, such as **`property`**. The core design of this feature was flawed and led to several problems. To "pass-through" a **`classmethod`**, consider using the **`__wrapped__`** attribute that was added in Python 3.10. (Contributed by Raymond Hettinger in gh-89519 [https://github.com/python/cpython/issues/89519].)
- Raise a **`RuntimeError`** when calling **`frame.clear()`** on a suspended frame (as has always been the case for an executing frame). (Contributed by Irit Katriel in gh-79932 [https://github.com/python/cpython/issues/79932].)

configparser

- Remove the undocumented **`LegacyInterpolation`** class, deprecated in the docstring since Python 3.2, and at runtime since Python 3.11. (Contributed by Hugo van

Kemenade in gh-104886 [https://github.com/python/cpython/issues/104886].)

importlib.metadata

- Remove deprecated subscript (`__getitem__()`) access for EntryPoint objects. (Contributed by Jason R. Coombs in gh-113175 [https://github.com/python/cpython/issues/113175].)

locale

- Remove the **`locale.resetlocale()`** function, deprecated in Python 3.11. Use `locale.setlocale(locale.LC_ALL, "")` instead. (Contributed by Victor Stinner in gh-104783 [https://github.com/python/cpython/issues/104783].)

opcode

- Move **`opcode.ENABLE_SPECIALIZATION`** to **`_opcode.ENABLE_SPECIALIZATION`**. This field was added in 3.12, it was never documented, and is not intended for external use. (Contributed by Irit Katriel in gh-105481 [https://github.com/python/cpython/issues/105481].)
- Remove **`opcode.is_pseudo()`**, **`opcode.MIN_PSEUDO_OPCODE`**, and **`opcode.MAX_PSEUDO_OPCODE`**, which were added in Python 3.12, but were neither documented nor exposed through `dis`, and were not intended to be used externally. (Contributed by Irit Katriel in gh-105481 [https://github.com/python/cpython/issues/105481].)

optparse

- This module is no longer considered soft deprecated. While `argparse` remains preferred for new projects that aren’t using a third party command line argument processing library, there are aspects of the way `argparse` works that mean the lower level `optparse` module may provide a better foundation for *writing* argument processing libraries, and for implementing command line applications which adhere more strictly than `argparse` does to various Unix command line processing conventions that originate in the behaviour of the C **`getopt()`** function . (Contributed by Alyssa Coghlan and Serhiy Storchaka in gh-126180 [https://github.com/python/cpython/issues/126180].)

pathlib

- Remove the ability to use **`Path`** objects as context managers. This functionality was deprecated and has had no effect since Python 3.9. (Contributed by Barney Gale in gh-83863 [https://github.com/python/cpython/issues/83863].)

re

- Remove the undocumented, deprecated, and broken **`re.template()`** function and **`re.TEMPLATE`** / **`re.T`** flag. (Contributed by Serhiy Storchaka and Nikita Sobolev in gh-105687 [https://github.com/python/cpython/issues/105687].)

tkinter.tix

- Remove the **`tkinter.tix`** module, deprecated in Python 3.6. The third-party Tix library which the module wrapped is unmaintained. (Contributed by Zachary Ware in gh-75552 [https://github.com/python/cpython/issues/75552].)

turtle

- Remove the **`RawTurtle.settiltangle()`** method, deprecated in the documentation since Python 3.1 and at runtime since Python 3.11. (Contributed by Hugo van Kemenade in gh-104876 [https://github.com/python/cpython/issues/104876].)

typing

- Remove the `typing.io` and `typing.re` namespaces, deprecated since Python 3.8. The items in those namespaces can be imported directly from the `typing` module. (Contributed by Sebastian Rittau in gh-92871 [https://github.com/python/cpython/issues/92871].)
- Remove the keyword-argument method of creating `TypedDict` types, deprecated in Python 3.11. (Contributed by Tomas Roun in gh-104786 [https://github.com/python/cpython/issues/104786].)

unittest

- Remove the following `unittest` functions, deprecated in Python 3.11:

  - `unittest.findTestCases()`
  - `unittest.makeSuite()`
  - `unittest.getTestCaseNames()`

  Use `TestLoader` methods instead:

  - `loadTestsFromModule()`
  - `loadTestsFromTestCase()`
  - `getTestCaseNames()`

  (Contributed by Hugo van Kemenade in gh-104835 [https://github.com/python/cpython/issues/104835].)

- Remove the untested and undocumented `TestProgram.usageExit()` method, deprecated in Python

3.11. (Contributed by Hugo van Kemenade in gh-104992 [https://github.com/python/cpython/issues/104992].)

urllib

- Remove the *cafile*, *capath*, and *cadefault* parameters of the `urllib.request.urlopen()` function, deprecated in Python 3.6. Use the *context* parameter instead with an `SSLContext` instance. The `ssl.SSLContext.load_cert_chain()` function can be used to load specific certificates, or let `ssl.create_default_context()` select the operating system's trusted certificate authority (CA) certificates. (Contributed by Victor Stinner in gh-105382 [https://github.com/python/cpython/issues/105382].)

webbrowser

- Remove the untested and undocumented `MacOSX` class, deprecated in Python 3.11. Use the `MacOSXOSAScript` class (introduced in Python 3.2) instead. (Contributed by Hugo van Kemenade in gh-104804 [https://github.com/python/cpython/issues/104804].)
- Remove the deprecated `MacOSXOSAScript._name` attribute. Use the `MacOSXOSAScript.name` attribute instead. (Contributed by Nikita Sobolev in gh-105546 [https://github.com/python/cpython/issues/105546].)

New Deprecations

- User-defined functions:
  - Deprecate assignment to a function's `__code__` attribute, where the new code object's type does not match the function's type. The different types are: plain function, generator, async generator, and

coroutine. (Contributed by Irit Katriel in gh-81137 [https://github.com/python/cpython/issues/81137].)

- **`array`**:
  - Deprecate the `'u'` format code (**`wchar_t`**) at runtime. This format code has been deprecated in documentation since Python 3.3, and will be removed in Python 3.16. Use the `'w'` format code (**`Py_UCS4`**) for Unicode characters instead. (Contributed by Hugo van Kemenade in gh-80480 [https://github.com/python/cpython/issues/80480].)
- **`ctypes`**:
  - Deprecate the undocumented **`SetPointerType()`** function, to be removed in Python 3.15. (Contributed by Victor Stinner in gh-105733 [https://github.com/python/cpython/issues/105733].)
  - Soft-deprecate the **`ARRAY()`** function in favour of `type * length` multiplication. (Contributed by Victor Stinner in gh-105733 [https://github.com/python/cpython/issues/105733].)
- **`decimal`**:
  - Deprecate the non-standard and undocumented **`Decimal`** format specifier `'N'`, which is only supported in the **`decimal`** module’s C implementation. Scheduled to be removed in Python 3.18. (Contributed by Serhiy Storchaka in gh-89902 [https://github.com/python/cpython/issues/89902].)
- **`dis`**:
  - Deprecate the **`HAVE_ARGUMENT`** separator. Check membership in **`hasarg`** instead. (Contributed by Irit Katriel in gh-109319 [https://github.com/python/cpython/issues/109319].)
- **`gettext`**:
  - Deprecate non-integer numbers as arguments to functions and methods that consider plural forms in the **`gettext`** module, even if no translation was found.

  (Contributed by Serhiy Storchaka in gh-88434 [https://github.com/python/cpython/issues/88434].)

- `glob`:
  - Deprecate the undocumented **`glob0()`** and **`glob1()`** functions. Use `glob()` and pass a path-like object specifying the root directory to the *root_dir* parameter instead. (Contributed by Barney Gale in gh-117337 [https://github.com/python/cpython/issues/117337].)
- `http.server`:
  - Deprecate `CGIHTTPRequestHandler`, to be removed in Python 3.15. Process-based CGI HTTP servers have been out of favor for a very long time. This code was outdated, unmaintained, and rarely used. It has a high potential for both security and functionality bugs. (Contributed by Gregory P. Smith in gh-109096 [https://github.com/python/cpython/issues/109096].)
  - Deprecate the **`--cgi`** flag to the **python -m http.server** command-line interface, to be removed in Python 3.15. (Contributed by Gregory P. Smith in gh-109096 [https://github.com/python/cpython/issues/109096].)
- `mimetypes`:
  - Soft-deprecate file path arguments to `guess_type()`, use `guess_file_type()` instead. (Contributed by Serhiy Storchaka in gh-66543 [https://github.com/python/cpython/issues/66543].)
- `re`:
  - Deprecate passing the optional *maxsplit*, *count*, or *flags* arguments as positional arguments to the module-level `split()`, `sub()`, and `subn()` functions. These parameters will become keyword-only in a future version of Python. (Contributed by Serhiy Storchaka in gh-56166 [https://github.com/python/cpython/issues/56166].)
- `pathlib`:

    - Deprecate **`PurePath.is_reserved()`**, to be removed in Python 3.15. Use **`os.path.isreserved()`** to detect reserved paths on Windows. (Contributed by Barney Gale in gh-88569 [https://github.com/python/cpython/issues/88569].)
- **`platform`**:
    - Deprecate **`java_ver()`**, to be removed in Python 3.15. This function is only useful for Jython support, has a confusing API, and is largely untested. (Contributed by Nikita Sobolev in gh-116349 [https://github.com/python/cpython/issues/116349].)
- **`pydoc`**:
    - Deprecate the undocumented **`ispackage()`** function. (Contributed by Zackery Spytz in gh-64020 [https://github.com/python/cpython/issues/64020].)
- **`sqlite3`**:
    - Deprecate passing more than one positional argument to the **`connect()`** function and the **`Connection`** constructor. The remaining parameters will become keyword-only in Python 3.15. (Contributed by Erlend E. Aasland in gh-107948 [https://github.com/python/cpython/issues/107948].)
    - Deprecate passing name, number of arguments, and the callable as keyword arguments for **`Connection.create_function()`** and **`Connection.create_aggregate()`** These parameters will become positional-only in Python 3.15. (Contributed by Erlend E. Aasland in gh-108278 [https://github.com/python/cpython/issues/108278].)
    - Deprecate passing the callback callable by keyword for the **`set_authorizer()`**, **`set_progress_handler()`**, and **`set_trace_callback()`** **`Connection`** methods. The callback callables will become positional-only in Python 3.15. (Contributed by Erlend E. Aasland in gh-108278 [https://github.com/python/cpython/issues/108278].)
- **`sys`**:

    - Deprecate the [`_enablelegacywindowsfsencoding()`] function, to be removed in Python 3.16. Use the [`PYTHONLEGACYWINDOWSFSENCODING`] environment variable instead. (Contributed by Inada Naoki in gh-73427 [https://github.com/python/cpython/issues/73427].)
- [`tarfile`]:
    - Deprecate the undocumented and unused **`TarFile.tarfile`** attribute, to be removed in Python 3.16. (Contributed in gh-115256 [https://github.com/python/cpython/issues/115256].)
- [`traceback`]:
    - Deprecate the [`TracebackException.exc_type`] attribute. Use [`TracebackException.exc_type_str`] instead. (Contributed by Irit Katriel in gh-112332 [https://github.com/python/cpython/issues/112332].)
- [`typing`]:
    - Deprecate the undocumented keyword argument syntax for creating [`NamedTuple`] classes (e.g. `Point = NamedTuple("Point", x=int, y=int)`), to be removed in Python 3.15. Use the class-based syntax or the functional syntax instead. (Contributed by Alex Waygood in gh-105566 [https://github.com/python/cpython/issues/105566].)
    - Deprecate omitting the *fields* parameter when creating a [`NamedTuple`] or [`typing.TypedDict`] class, and deprecate passing `None` to the *fields* parameter of both types. Python 3.15 will require a valid sequence for the *fields* parameter. To create a NamedTuple class with zero fields, use `class NT(NamedTuple): pass` or `NT = NamedTuple("NT", ())`. To create a TypedDict class with zero fields, use `class TD(TypedDict): pass` or `TD = TypedDict("TD", {})`. (Contributed by Alex Waygood in gh-105566 [https://github.com/python/cpython/issues/105566] and gh-105570 [https://github.com/python/cpython/issues/105570].)

  - Deprecate the [`typing.no_type_check_decorator()`] decorator function, to be removed in Python 3.15. After eight years in the [`typing`] module, it has yet to be supported by any major type checker. (Contributed by Alex Waygood in gh-106309 [https://github.com/python/cpython/issues/106309].)
  - Deprecate [`typing.AnyStr`]. In Python 3.16, it will be removed from `typing.__all__`, and a [`DeprecationWarning`] will be emitted at runtime when it is imported or accessed. It will be removed entirely in Python 3.18. Use the new type parameter syntax instead. (Contributed by Michael The in gh-107116 [https://github.com/python/cpython/issues/107116].)
- [`wave`]:
  - Deprecate the [`getmark()`], **`setmark()`**, and [`getmarkers()`] methods of the [`Wave_read`] and [`Wave_write`] classes, to be removed in Python 3.15. (Contributed by Victor Stinner in gh-105096 [https://github.com/python/cpython/issues/105096].)

Pending removal in Python 3.14

- [`argparse`]: The *type*, *choices*, and *metavar* parameters of **`argparse.BooleanOptionalAction`** are deprecated and will be removed in 3.14. (Contributed by Nikita Sobolev in gh-92248 [https://github.com/python/cpython/issues/92248].)

- [`ast`]: The following features have been deprecated in documentation since Python 3.8, now cause a [`DeprecationWarning`] to be emitted at runtime when they are accessed or used, and will be removed in Python 3.14:

  - **`ast.Num`**
  - **`ast.Str`**
  - **`ast.Bytes`**

  - **`ast.NameConstant`**
  - **`ast.Ellipsis`**

  Use `ast.Constant` instead. (Contributed by Serhiy Storchaka in gh-90953 [https://github.com/python/cpython/issues/90953].)

- `asyncio`:
  - The child watcher classes **`asyncio.MultiLoopChildWatcher`**, **`asyncio.FastChildWatcher`**, **`asyncio.AbstractChildWatcher`** and **`asyncio.SafeChildWatcher`** are deprecated and will be removed in Python 3.14. (Contributed by Kumar Aditya in gh-94597 [https://github.com/python/cpython/issues/94597].)
  - **`asyncio.set_child_watcher()`**, **`asyncio.get_child_watcher()`**, **`asyncio.AbstractEventLoopPolicy.set_child_watcher()`** and **`asyncio.AbstractEventLoopPolicy.get_child_watcher()`** are deprecated and will be removed in Python 3.14. (Contributed by Kumar Aditya in gh-94597 [https://github.com/python/cpython/issues/94597].)
  - The `get_event_loop()` method of the default event loop policy now emits a `DeprecationWarning` if there is no current event loop set and it decides to create one. (Contributed by Serhiy Storchaka and Guido van Rossum in gh-100160 [https://github.com/python/cpython/issues/100160].)

- `email`: Deprecated the *isdst* parameter in `email.utils.localtime()`. (Contributed by Alan Williams in gh-72346 [https://github.com/python/cpython/issues/72346].)

- `importlib.abc` deprecated classes:

  - **`importlib.abc.ResourceReader`**
  - **`importlib.abc.Traversable`**
  - **`importlib.abc.TraversableResources`**

  Use `importlib.resources.abc` classes instead:

  - `importlib.resources.abc.Traversable`
  - `importlib.resources.abc.TraversableResources`

  (Contributed by Jason R. Coombs and Hugo van Kemenade in gh-93963 [https://github.com/python/cpython/issues/93963].)

- `itertools` had undocumented, inefficient, historically buggy, and inconsistent support for copy, deepcopy, and pickle operations. This will be removed in 3.14 for a significant reduction in code volume and maintenance burden. (Contributed by Raymond Hettinger in gh-101588 [https://github.com/python/cpython/issues/101588].)

- `multiprocessing`: The default start method will change to a safer one on Linux, BSDs, and other non-macOS POSIX platforms where `'fork'` is currently the default (gh-84559 [https://github.com/python/cpython/issues/84559]). Adding a runtime warning about this was deemed too disruptive as the majority of code is not expected to care. Use the `get_context()` or `set_start_method()` APIs to explicitly specify when your code *requires* `'fork'`. See Contexts and start methods.

- `pathlib`: `is_relative_to()` and `relative_to()`: passing additional arguments is deprecated.

- `pkgutil`: **`pkgutil.find_loader()`** and **`pkgutil.get_loader()`** now raise `DeprecationWarning`; use `importlib.util.find_spec()` instead. (Contributed by Nikita

Sobolev in gh-97850 [https://github.com/python/cpython/issues/97850].)

- `pty`:
  - `master_open()`: use `pty.openpty()`.
  - `slave_open()`: use `pty.openpty()`.
- `sqlite3`:
  - **`version`** and **`version_info`**.
  - `execute()` and `executemany()` if named placeholders are used and *parameters* is a sequence instead of a `dict`.
- `urllib`: **`urllib.parse.Quoter`** is deprecated: it was not intended to be a public API. (Contributed by Gregory P. Smith in gh-88168 [https://github.com/python/cpython/issues/88168].)

Pending removal in Python 3.15

- The import system:
  - Setting `__cached__` on a module while failing to set `__spec__.cached` is deprecated. In Python 3.15, **`__cached__`** will cease to be set or take into consideration by the import system or standard library. (gh-97879 [https://github.com/python/cpython/issues/97879])
  - Setting `__package__` on a module while failing to set `__spec__.parent` is deprecated. In Python 3.15, **`__package__`** will cease to be set or take into consideration by the import system or standard library. (gh-97879 [https://github.com/python/cpython/issues/97879])
- `ctypes`:

    - The undocumented `ctypes.SetPointerType()` function has been deprecated since Python 3.13.
- `http.server`:
    - The obsolete and rarely used `CGIHTTPRequestHandler` has been deprecated since Python 3.13. No direct replacement exists. *Anything* is better than CGI to interface a web server with a request handler.
    - The `--cgi` flag to the **python -m http.server** command-line interface has been deprecated since Python 3.13.
- `importlib`:
    - `load_module()` method: use `exec_module()` instead.
- `locale`:
    - The `getdefaultlocale()` function has been deprecated since Python 3.11. Its removal was originally planned for Python 3.13 (gh-90817 [https://github.com/python/cpython/issues/90817]), but has been postponed to Python 3.15. Use `getlocale()`, `setlocale()`, and `getencoding()` instead. (Contributed by Hugo van Kemenade in gh-111187 [https://github.com/python/cpython/issues/111187].)
- `pathlib`:
    - `PurePath.is_reserved()` has been deprecated since Python 3.13. Use `os.path.isreserved()` to detect reserved paths on Windows.
- `platform`:
    - `java_ver()` has been deprecated since Python 3.13. This function is only useful for Jython support, has a confusing API, and is largely untested.
- `sysconfig`:
    - The *check_home* argument of `sysconfig.is_python_build()` has been deprecated since Python 3.12.
- `threading`:
    - `RLock()` will take no arguments in Python 3.15. Passing any arguments has been deprecated since

  Python 3.14, as the Python version does not permit any arguments, but the C version allows any number of positional or keyword arguments, ignoring every argument.

- **`types`**:
  - **`types.CodeType`**: Accessing **`co_lnotab`** was deprecated in **PEP 626** [https://peps.python.org/pep-0626/] since 3.10 and was planned to be removed in 3.12, but it only got a proper **`DeprecationWarning`** in 3.12. May be removed in 3.15. (Contributed by Nikita Sobolev in gh-101866 [https://github.com/python/cpython/issues/101866].)
- **`typing`**:
  - The undocumented keyword argument syntax for creating **`NamedTuple`** classes (for example, `Point = NamedTuple("Point", x=int, y=int)`) has been deprecated since Python 3.13. Use the class-based syntax or the functional syntax instead.
  - When using the functional syntax of **`TypedDict`**s, failing to pass a value to the *fields* parameter (`TD = TypedDict("TD")`) or passing `None` (`TD = TypedDict("TD", None)`) has been deprecated since Python 3.13. Use `class TD(TypedDict): pass` or `TD = TypedDict("TD", {})` to create a TypedDict with zero field.
  - The **`typing.no_type_check_decorator()`** decorator function has been deprecated since Python 3.13. After eight years in the **`typing`** module, it has yet to be supported by any major type checker.
- **`wave`**:
  - The **`getmark()`**, **`setmark()`**, and **`getmarkers()`** methods of the **`Wave_read`** and **`Wave_write`** classes have been deprecated since Python 3.13.
- **`zipimport`**:
  - **`load_module()`** has been deprecated since Python 3.10. Use **`exec_module()`** instead. (Contributed by Jiahao Li in gh-125746 [https://github.com/python/cpython/issues/125746].)

Pending removal in Python 3.16

- The import system:

  - Setting `__loader__` on a module while failing to set `__spec__.loader` is deprecated. In Python 3.16, `__loader__` will cease to be set or taken into consideration by the import system or the standard library.

- `array`:

  - The `'u'` format code (`wchar_t`) has been deprecated in documentation since Python 3.3 and at runtime since Python 3.13. Use the `'w'` format code (`Py_UCS4`) for Unicode characters instead.

- `asyncio`:

  - `asyncio.iscoroutinefunction()` is deprecated and will be removed in Python 3.16; use `inspect.iscoroutinefunction()` instead. (Contributed by Jiahao Li and Kumar Aditya in gh-122875 [https://github.com/python/cpython/issues/122875].)

  - `asyncio` policy system is deprecated and will be removed in Python 3.16. In particular, the following classes and functions are deprecated:

    - `asyncio.AbstractEventLoopPolicy`
    - `asyncio.DefaultEventLoopPolicy`
    - `asyncio.WindowsSelectorEventLoopPolicy`
    - `asyncio.WindowsProactorEventLoopPolicy`
    - `asyncio.get_event_loop_policy()`
    - `asyncio.set_event_loop_policy()`

Users should use **`asyncio.run()`** or **`asyncio.Runner`** with *loop_factory* to use the desired event loop implementation.

For example, to use **`asyncio.SelectorEventLoop`** on Windows:

```
import asyncio

async def main():
    ...

asyncio.run(main(),
loop_factory=asyncio.SelectorEventLoop)
```

(Contributed by Kumar Aditya in gh-127949 [https://github.com/python/cpython/issues/127949].)

- **`builtins`**:
  - Bitwise inversion on boolean types, `~True` or `~False` has been deprecated since Python 3.12, as it produces surprising and unintuitive results (`-2` and `-1`). Use `not x` instead for the logical negation of a Boolean. In the rare case that you need the bitwise inversion of the underlying integer, convert to `int` explicitly (`~int(x)`).
- **`functools`**:
  - Calling the Python implementation of **`functools.reduce()`** with *function* or *sequence* as keyword arguments has been deprecated since Python 3.14.
- **`logging`**:

Support for custom logging handlers with the *strm* argument is deprecated and scheduled for removal in Python 3.16. Define handlers with the *stream* argument instead. (Contributed by Mariusz Felisiak in gh-115032 [https://github.com/python/cpython/issues/115032].)

- `mimetypes`:
    - Valid extensions start with a '.' or are empty for `mimetypes.MimeTypes.add_type()`. Undotted extensions are deprecated and will raise a `ValueError` in Python 3.16. (Contributed by Hugo van Kemenade in gh-75223 [https://github.com/python/cpython/issues/75223].)
- `shutil`:
    - The **`ExecError`** exception has been deprecated since Python 3.14. It has not been used by any function in **`shutil`** since Python 3.4, and is now an alias of `RuntimeError`.
- `symtable`:
    - The `Class.get_methods` method has been deprecated since Python 3.14.
- `sys`:
    - The `_enablelegacywindowsfsencoding()` function has been deprecated since Python 3.13. Use the `PYTHONLEGACYWINDOWSFSENCODING` environment variable instead.
- `sysconfig`:
    - The **`sysconfig.expand_makefile_vars()`** function has been deprecated since Python 3.14. Use the `vars`

argument of **`sysconfig.get_paths()`** instead.

- **`tarfile`**:
    - The undocumented and unused **`TarFile.tarfile`** attribute has been deprecated since Python 3.13.

Pending removal in Python 3.17

- **`collections.abc`**:
    - **`collections.abc.ByteString`** is scheduled for removal in Python 3.17.

      Use `isinstance(obj, collections.abc.Buffer)` to test if `obj` implements the buffer protocol at runtime. For use in type annotations, either use **`Buffer`** or a union that explicitly specifies the types your code supports (e.g., `bytes | bytearray | memoryview`).

      `ByteString` was originally intended to be an abstract class that would serve as a supertype of both **`bytes`** and **`bytearray`**. However, since the ABC never had any methods, knowing that an object was an instance of **`ByteString`** never actually told you anything useful about the object. Other common buffer types such as **`memoryview`** were also never understood as subtypes of **`ByteString`** (either at runtime or by static type checkers).

      See **PEP 688** [https://peps.python.org/pep-0688/#current-options] for more details. (Contributed by Shantanu Jain in gh-91896 [https://github.com/python/cpython/issues/91896].)

- **`typing`**:

  - Before Python 3.14, old-style unions were implemented using the private class `typing._UnionGenericAlias`. This class is no longer needed for the implementation, but it has been retained for backward compatibility, with removal scheduled for Python 3.17. Users should use documented introspection helpers like **`typing.get_origin()`** and **`typing.get_args()`** instead of relying on private implementation details.

  - **`typing.ByteString`**, deprecated since Python 3.9, is scheduled for removal in Python 3.17.

    Use `isinstance(obj, collections.abc.Buffer)` to test if `obj` implements the buffer protocol at runtime. For use in type annotations, either use **`Buffer`** or a union that explicitly specifies the types your code supports (e.g., `bytes | bytearray | memoryview`).

    `ByteString` was originally intended to be an abstract class that would serve as a supertype of both **`bytes`** and **`bytearray`**. However, since the ABC never had any methods, knowing that an object was an instance of **`ByteString`** never actually told you anything useful about the object. Other common buffer types such as **`memoryview`** were also never understood as subtypes of **`ByteString`** (either at runtime or by static type checkers).

    See **PEP 688** [https://peps.python.org/pep-0688/#current-options] for more details. (Contributed by Shantanu Jain in gh-91896 [https://github.com/python/cpython/issues/91896].)

Pending removal in Python 3.18

- **`decimal`**:
    - The non-standard and undocumented **`Decimal`** format specifier `'N'`, which is only supported in the **`decimal`** module's C implementation, has been deprecated since Python 3.13. (Contributed by Serhiy Storchaka in gh-89902 [https://github.com/python/cpython/issues/89902].)

Pending removal in Python 3.19

- **`ctypes`**:
    - Implicitly switching to the MSVC-compatible struct layout by setting **`_pack_`** but not **`_layout_`** on non-Windows platforms.

Pending removal in future versions

The following APIs will be removed in the future, although there is currently no date scheduled for their removal.

- **`argparse`**:
    - Nesting argument groups and nesting mutually exclusive groups are deprecated.
    - Passing the undocumented keyword argument *prefix_chars* to **`add_argument_group()`** is now deprecated.
    - The **`argparse.FileType`** type converter is deprecated.
- **`builtins`**:
    - Generators: `throw(type, exc, tb)` and `athrow(type, exc, tb)` signature is deprecated: use `throw(exc)` and `athrow(exc)` instead, the single argument signature.
    - Currently Python accepts numeric literals immediately followed by keywords, for example `0in x`, `1or x`, `0if 1else 2`. It allows confusing and ambiguous expressions like `[0x1for x in y]` (which can be interpreted as `[0x1 for x in y]` or `[0x1f or x in y]`). A syntax warning is raised if the numeric

  literal is immediately followed by one of keywords **`and`**, **`else`**, **`for`**, **`if`**, **`in`**, **`is`** and **`or`**. In a future release it will be changed to a syntax error. (gh-87999 [https://github.com/python/cpython/issues/87999])
  - Support for `__index__()` and `__int__()` method returning non-int type: these methods will be required to return an instance of a strict subclass of **`int`**.
  - Support for `__float__()` method returning a strict subclass of **`float`**: these methods will be required to return an instance of **`float`**.
  - Support for `__complex__()` method returning a strict subclass of **`complex`**: these methods will be required to return an instance of **`complex`**.
  - Delegation of `int()` to `__trunc__()` method.
  - Passing a complex number as the *real* or *imag* argument in the **`complex()`** constructor is now deprecated; it should only be passed as a single positional argument. (Contributed by Serhiy Storchaka in gh-109218 [https://github.com/python/cpython/issues/109218].)
- **`calendar`**: `calendar.January` and `calendar.February` constants are deprecated and replaced by **`calendar.JANUARY`** and **`calendar.FEBRUARY`**. (Contributed by Prince Roshan in gh-103636 [https://github.com/python/cpython/issues/103636].)
- **`codecs`**: use **`open()`** instead of **`codecs.open()`**. (gh-133038 [https://github.com/python/cpython/issues/133038])
- **`codeobject.co_lnotab`**: use the **`codeobject.co_lines()`** method instead.
- **`datetime`**:
  - **`utcnow()`**: use `datetime.datetime.now(tz=datetime.UTC)`.
  - **`utcfromtimestamp()`**: use `datetime.datetime.fromtimestamp(timestamp, tz=datetime.UTC)`.
- **`gettext`**: Plural value must be an integer.
- **`importlib`**:

  - `cache_from_source()` *debug_override* parameter is deprecated: use the *optimization* parameter instead.
- `importlib.metadata`:
  - `EntryPoints` tuple interface.
  - Implicit `None` on return values.
- `logging`: the `warn()` method has been deprecated since Python 3.3, use `warning()` instead.
- `mailbox`: Use of StringIO input and text mode is deprecated, use BytesIO and binary mode instead.
- `os`: Calling `os.register_at_fork()` in multi-threaded process.
- **`pydoc.ErrorDuringImport`**: A tuple value for *exc_info* parameter is deprecated, use an exception instance.
- `re`: More strict rules are now applied for numerical group references and group names in regular expressions. Only sequence of ASCII digits is now accepted as a numerical reference. The group name in bytes patterns and replacement strings can now only contain ASCII letters and digits and underscore. (Contributed by Serhiy Storchaka in gh-91760 [https://github.com/python/cpython/issues/91760].)
- **`sre_compile`**, **`sre_constants`** and **`sre_parse`** modules.
- `shutil`: `rmtree()`'s *onerror* parameter is deprecated in Python 3.12; use the *onexc* parameter instead.
- `ssl` options and protocols:
  - `ssl.SSLContext` without protocol argument is deprecated.
  - `ssl.SSLContext`: `set_npn_protocols()` and **`selected_npn_protocol()`** are deprecated: use ALPN instead.
  - `ssl.OP_NO_SSL*` options
  - `ssl.OP_NO_TLS*` options
  - `ssl.PROTOCOL_SSLv3`
  - `ssl.PROTOCOL_TLS`
  - `ssl.PROTOCOL_TLSv1`

  - `ssl.PROTOCOL_TLSv1_1`
  - `ssl.PROTOCOL_TLSv1_2`
  - `ssl.TLSVersion.SSLv3`
  - `ssl.TLSVersion.TLSv1`
  - `ssl.TLSVersion.TLSv1_1`
- `threading` methods:
  - **`threading.Condition.notifyAll()`**: use `notify_all()`.
  - **`threading.Event.isSet()`**: use `is_set()`.
  - **`threading.Thread.isDaemon()`**, `threading.Thread.setDaemon()`: use `threading.Thread.daemon` attribute.
  - **`threading.Thread.getName()`**, `threading.Thread.setName()`: use `threading.Thread.name` attribute.
  - **`threading.currentThread()`**: use `threading.current_thread()`.
  - **`threading.activeCount()`**: use `threading.active_count()`.
- `typing.Text` (gh-92332 [https://github.com/python/cpython/issues/92332]).
- The internal class `typing._UnionGenericAlias` is no longer used to implement `typing.Union`. To preserve compatibility with users using this private class, a compatibility shim will be provided until at least Python 3.17. (Contributed by Jelle Zijlstra in gh-105499 [https://github.com/python/cpython/issues/105499].)
- `unittest.IsolatedAsyncioTestCase`: it is deprecated to return a value that is not `None` from a test case.
- `urllib.parse` deprecated functions: `urlparse()` instead
  - `splitattr()`
  - `splithost()`
  - `splitnport()`
  - `splitpasswd()`
  - `splitport()`
  - `splitquery()`
  - `splittag()`
  - `splittype()`
  - `splituser()`

    - `splitvalue()`
    - `to_bytes()`
- **`wsgiref`**: `SimpleHandler.stdout.write()` should not do partial writes.
- **`xml.etree.ElementTree`**: Testing the truth value of an **`Element`** is deprecated. In a future release it will always return `True`. Prefer explicit `len(elem)` or `elem is not None` tests instead.
- **`sys._clear_type_cache()`** is deprecated: use **`sys._clear_internal_caches()`** instead.

CPython Bytecode Changes

- The oparg of **`YIELD_VALUE`** is now `1` if the yield is part of a yield-from or await, and `0` otherwise. The oparg of **`RESUME`** was changed to add a bit indicating if the except-depth is 1, which is needed to optimize closing of generators. (Contributed by Irit Katriel in gh-111354 [https://github.com/python/cpython/issues/111354].)

C API Changes

New Features

- Add the PyMonitoring C API for generating **PEP 669** [https://peps.python.org/pep-0669/] monitoring events:
    - **`PyMonitoringState`**
    - **`PyMonitoring_FirePyStartEvent()`**
    - **`PyMonitoring_FirePyResumeEvent()`**
    - **`PyMonitoring_FirePyReturnEvent()`**
    - **`PyMonitoring_FirePyYieldEvent()`**
    - **`PyMonitoring_FireCallEvent()`**
    - **`PyMonitoring_FireLineEvent()`**
    - **`PyMonitoring_FireJumpEvent()`**

  - `PyMonitoring_FireBranchEvent`
  - **`PyMonitoring_FireCReturnEvent()`**
  - **`PyMonitoring_FirePyThrowEvent()`**
  - **`PyMonitoring_FireRaiseEvent()`**
  - **`PyMonitoring_FireCRaiseEvent()`**
  - **`PyMonitoring_FireReraiseEvent()`**
  - **`PyMonitoring_FireExceptionHandledEvent()`**
  - **`PyMonitoring_FirePyUnwindEvent()`**
  - **`PyMonitoring_FireStopIterationEvent()`**
  - **`PyMonitoring_EnterScope()`**
  - **`PyMonitoring_ExitScope()`**

  (Contributed by Irit Katriel in gh-111997 [https://github.com/python/cpython/issues/111997]).

- Add **`PyMutex`**, a lightweight mutex that occupies a single byte, and the new **`PyMutex_Lock()`** and **`PyMutex_Unlock()`** functions. **`PyMutex_Lock()`** will release the GIL (if currently held) if the operation needs to block. (Contributed by Sam Gross in gh-108724 [https://github.com/python/cpython/issues/108724].)

- Add the PyTime C API to provide access to system clocks:

  - **`PyTime_t`**.
  - **`PyTime_MIN`** and **`PyTime_MAX`**.
  - **`PyTime_AsSecondsDouble()`**.
  - **`PyTime_Monotonic()`**.
  - **`PyTime_MonotonicRaw()`**.
  - **`PyTime_PerfCounter()`**.
  - **`PyTime_PerfCounterRaw()`**.
  - **`PyTime_Time()`**.
  - **`PyTime_TimeRaw()`**.

(Contributed by Victor Stinner and Petr Viktorin in gh-110850 [https://github.com/python/cpython/issues/110850].)

- Add the **`PyDict_ContainsString()`** function with the same behavior as **`PyDict_Contains()`**, but *key* is specified as a const char* UTF-8 encoded bytes string, rather than a PyObject*. (Contributed by Victor Stinner in gh-108314 [https://github.com/python/cpython/issues/108314].)
- Add the **`PyDict_GetItemRef()`** and **`PyDict_GetItemStringRef()`** functions, which behave similarly to **`PyDict_GetItemWithError()`**, but return a strong reference instead of a borrowed reference. Moreover, these functions return `-1` on error, removing the need to check **`PyErr_Occurred()`**. (Contributed by Victor Stinner in gh-106004 [https://github.com/python/cpython/issues/106004].)
- Add the **`PyDict_SetDefaultRef()`** function, which behaves similarly to **`PyDict_SetDefault()`**, but returns a strong reference instead of a borrowed reference. This function returns `-1` on error, `0` on insertion, and `1` if the key was already present in the dictionary. (Contributed by Sam Gross in gh-112066 [https://github.com/python/cpython/issues/112066].)
- Add the **`PyDict_Pop()`** and **`PyDict_PopString()`** functions to remove a key from a dictionary and optionally return the removed value. This is similar to **`dict.pop()`**, though there is no default value, and **`KeyError`** is not raised for missing keys. (Contributed by Stefan Behnel and Victor Stinner in gh-111262 [https://github.com/python/cpython/issues/111262].)
- Add the **`PyMapping_GetOptionalItem()`** and **`PyMapping_GetOptionalItemString()`** functions as alternatives to **`PyObject_GetItem()`** and **`PyMapping_GetItemString()`**

respectively. The new functions do not raise **`KeyError`** if the requested key is missing from the mapping. These variants are more convenient and faster if a missing key should not be treated as a failure. (Contributed by Serhiy Storchaka in gh-106307 [https://github.com/python/cpython/issues/106307].)

- Add the **`PyObject_GetOptionalAttr()`** and **`PyObject_GetOptionalAttrString()`** functions as alternatives to **`PyObject_GetAttr()`** and **`PyObject_GetAttrString()`** respectively. The new functions do not raise **`AttributeError`** if the requested attribute is not found on the object. These variants are more convenient and faster if the missing attribute should not be treated as a failure. (Contributed by Serhiy Storchaka in gh-106521 [https://github.com/python/cpython/issues/106521].)

- Add the **`PyErr_FormatUnraisable()`** function as an extension to **`PyErr_WriteUnraisable()`** that allows customizing the warning message. (Contributed by Serhiy Storchaka in gh-108082 [https://github.com/python/cpython/issues/108082].)

- Add new functions that return a strong reference instead of a borrowed reference for frame locals, globals, and builtins, as part of PEP 667:

  - **`PyEval_GetFrameBuiltins()`** replaces **`PyEval_GetBuiltins()`**
  - **`PyEval_GetFrameGlobals()`** replaces **`PyEval_GetGlobals()`**
  - **`PyEval_GetFrameLocals()`** replaces **`PyEval_GetLocals()`**

  (Contributed by Mark Shannon and Tian Gao in gh-74929 [https://github.com/python/cpython/issues/74929].)

- Add the **`Py_GetConstant()`** and **`Py_GetConstantBorrowed()`** functions to get strong or borrowed references to constants. For example, `Py_GetConstant(Py_CONSTANT_ZERO)` returns a strong reference to the constant zero.

(Contributed by Victor Stinner in gh-115754 [https://github.com/python/cpython/issues/115754].)

- Add the `PyImport_AddModuleRef()` function as a replacement for `PyImport_AddModule()` that returns a strong reference instead of a borrowed reference. (Contributed by Victor Stinner in gh-105922 [https://github.com/python/cpython/issues/105922].)
- Add the `Py_IsFinalizing()` function to check whether the main Python interpreter is shutting down. (Contributed by Victor Stinner in gh-108014 [https://github.com/python/cpython/issues/108014].)
- Add the `PyList_GetItemRef()` function as a replacement for `PyList_GetItem()` that returns a strong reference instead of a borrowed reference. (Contributed by Sam Gross in gh-114329 [https://github.com/python/cpython/issues/114329].)
- Add the `PyList_Extend()` and `PyList_Clear()` functions, mirroring the Python `list.extend()` and `list.clear()` methods. (Contributed by Victor Stinner in gh-111138 [https://github.com/python/cpython/issues/111138].)
- Add the `PyLong_AsInt()` function. It behaves similarly to `PyLong_AsLong()`, but stores the result in a C int instead of a C long. (Contributed by Victor Stinner in gh-108014 [https://github.com/python/cpython/issues/108014].)
- Add the `PyLong_AsNativeBytes()`, `PyLong_FromNativeBytes()`, and `PyLong_FromUnsignedNativeBytes()` functions to simplify converting between native integer types and Python `int` objects. (Contributed by Steve Dower in gh-111140 [https://github.com/python/cpython/issues/111140].)
- Add `PyModule_Add()` function, which is similar to `PyModule_AddObjectRef()` and `PyModule_AddObject()`, but

always steals a reference to the value. (Contributed by Serhiy Storchaka in gh-86493 [https://github.com/python/cpython/issues/86493].)

- Add the `PyObject_GenericHash()` function that implements the default hashing function of a Python object. (Contributed by Serhiy Storchaka in gh-113024 [https://github.com/python/cpython/issues/113024].)

- Add the `Py_HashPointer()` function to hash a raw pointer. (Contributed by Victor Stinner in gh-111545 [https://github.com/python/cpython/issues/111545].)

- Add the `PyObject_VisitManagedDict()` and `PyObject_ClearManagedDict()` functions. which must be called by the traverse and clear functions of a type using the `Py_TPFLAGS_MANAGED_DICT` flag. The pythoncapi-compat project [https://github.com/python/pythoncapi-compat/] can be used to use these functions with Python 3.11 and 3.12. (Contributed by Victor Stinner in gh-107073 [https://github.com/python/cpython/issues/107073].)

- Add the `PyRefTracer_SetTracer()` and `PyRefTracer_GetTracer()` functions, which enable tracking object creation and destruction in the same way that the `tracemalloc` module does. (Contributed by Pablo Galindo in gh-93502 [https://github.com/python/cpython/issues/93502].)

- Add the `PySys_AuditTuple()` function as an alternative to `PySys_Audit()` that takes event arguments as a Python `tuple` object. (Contributed by Victor Stinner in gh-85283 [https://github.com/python/cpython/issues/85283].)

- Add the `PyThreadState_GetUnchecked()` function as an alternative to `PyThreadState_Get()` that doesn't kill the process with a fatal error if it is `NULL`. The caller is responsible for checking if the result is `NULL`. (Contributed by Victor Stinner in gh-108867 [https://github.com/python/cpython/issues/108867].)

- Add the **`PyType_GetFullyQualifiedName()`** function to get the type's fully qualified name. The module name is prepended if **`type.__module__`** is a string and is not equal to either `'builtins'` or `'__main__'`. (Contributed by Victor Stinner in gh-111696 [https://github.com/python/cpython/issues/111696].)

- Add the **`PyType_GetModuleName()`** function to get the type's module name. This is equivalent to getting the **`type.__module__`** attribute. (Contributed by Eric Snow and Victor Stinner in gh-111696 [https://github.com/python/cpython/issues/111696].)

- Add the **`PyUnicode_EqualToUTF8AndSize()`** and **`PyUnicode_EqualToUTF8()`** functions to compare a Unicode object with a const char* UTF-8 encoded string and `1` if they are equal or `0` otherwise. These functions do not raise exceptions. (Contributed by Serhiy Storchaka in gh-110289 [https://github.com/python/cpython/issues/110289].)

- Add the **`PyWeakref_GetRef()`** function as an alternative to **`PyWeakref_GetObject()`** that returns a strong reference or `NULL` if the referent is no longer live. (Contributed by Victor Stinner in gh-105927 [https://github.com/python/cpython/issues/105927].)

- Add fixed variants of functions which silently ignore errors:
    - **`PyObject_HasAttrWithError()`** replaces **`PyObject_HasAttr()`**.
    - **`PyObject_HasAttrStringWithError()`** replaces **`PyObject_HasAttrString()`**.

  - **`PyMapping_HasKeyWithError()`** replaces **`PyMapping_HasKey()`**.
  - **`PyMapping_HasKeyStringWithError()`** replaces **`PyMapping_HasKeyString()`**.

The new functions return `-1` for errors and the standard `1` for true and `0` for false.

(Contributed by Serhiy Storchaka in gh-108511 [https://github.com/python/cpython/issues/108511].)

Changed C APIs

- The *keywords* parameter of **`PyArg_ParseTupleAndKeywords()`** and **`PyArg_VaParseTupleAndKeywords()`** now has type char *const* in C and const char *const* in C++, instead of char**. In C++, this makes these functions compatible with arguments of type const char *const*, const char**, or char *const* without an explicit type cast. In C, the functions only support arguments of type char *const*. This can be overridden with the **`PY_CXX_CONST`** macro. (Contributed by Serhiy Storchaka in gh-65210 [https://github.com/python/cpython/issues/65210].)

- **`PyArg_ParseTupleAndKeywords()`** now supports non-ASCII keyword parameter names. (Contributed by Serhiy Storchaka in gh-110815 [https://github.com/python/cpython/issues/110815].)

- The **`PyCode_GetFirstFree()`** function is now unstable API and is now named **`PyUnstable_Code_GetFirstFree()`**. (Contributed by Bogdan Romanyuk in gh-115781 [https://github.com/python/cpython/issues/115781].)

- The **`PyDict_GetItem()`**, **`PyDict_GetItemString()`**, **`PyMapping_HasKey()`**, **`PyMapping_HasKeyString()`**, `PyObject_HasAttr()`, `PyObject_HasAttrString()`, and `PySys_GetObject()` functions, each of which clears all errors which occurred when calling them now reports these errors using `sys.unraisablehook()`. You may replace them with other functions as recommended in the documentation. (Contributed by Serhiy Storchaka in gh-106672 [https://github.com/python/cpython/issues/106672].)

- Add support for the `%T`, `%#T`, `%N` and `%#N` formats to `PyUnicode_FromFormat()`:

  - `%T`: Get the fully qualified name of an object type
  - `%#T`: As above, but use a colon as the separator
  - `%N`: Get the fully qualified name of a type
  - `%#N`: As above, but use a colon as the separator

  See **PEP 737** [https://peps.python.org/pep-0737/] for more information. (Contributed by Victor Stinner in gh-111696 [https://github.com/python/cpython/issues/111696].)

- You no longer have to define the `PY_SSIZE_T_CLEAN` macro before including `Python.h` when using `#` formats in format codes. APIs accepting the format codes always use `Py_ssize_t` for `#` formats. (Contributed by Inada Naoki in gh-104922 [https://github.com/python/cpython/issues/104922].)

- If Python is built in debug mode or `with assertions`, `PyTuple_SET_ITEM()` and `PyList_SET_ITEM()` now check the index argument with an assertion. (Contributed by Victor Stinner in gh-106168 [https://github.com/python/cpython/issues/106168].)

Limited C API Changes

- The following functions are now included in the Limited C API:

  - `PyMem_RawMalloc()`
  - `PyMem_RawCalloc()`
  - `PyMem_RawRealloc()`
  - `PyMem_RawFree()`
  - `PySys_Audit()`
  - `PySys_AuditTuple()`
  - `PyType_GetModuleByDef()`

  (Contributed by Victor Stinner in gh-85283 [https://github.com/python/cpython/issues/85283] and gh-116936 [https://github.com/python/cpython/issues/116936].)

- Python built with `--with-trace-refs` (tracing references) now supports the Limited API. (Contributed by Victor Stinner in gh-108634 [https://github.com/python/cpython/issues/108634].)

Removed C APIs

- Remove several functions, macros, variables, etc with names prefixed by `_Py` or `_PY` (which are considered private). If your project is affected by one of these removals and you believe that the removed API should remain available, please open a new issue to request a public C API and add `cc: @vstinner` to the issue to notify Victor Stinner. (Contributed by Victor Stinner in gh-106320 [https://github.com/python/cpython/issues/106320].)

- Remove old buffer protocols deprecated in Python 3.0. Use Buffer Protocol instead.

  - **`PyObject_CheckReadBuffer()`**: Use `PyObject_CheckBuffer()` to test whether the object supports the buffer protocol. Note that `PyObject_CheckBuffer()` doesn't guarantee that `PyObject_GetBuffer()` will succeed. To test if the object is actually readable, see the next example of `PyObject_GetBuffer()`.

  - **`PyObject_AsCharBuffer()`**, **`PyObject_AsReadBuffer()`**: Use `PyObject_GetBuffer()` and `PyBuffer_Release()` instead:

    ```
    Py_buffer view;
    if (PyObject_GetBuffer(obj, &view, PyBUF_SIMPLE) < 0) {
        return NULL;
    }
    // Use `view.buf` and `view.len` to read from the
    buffer.
    // You may need to cast buf as `(const char*)view.buf`.
    PyBuffer_Release(&view);
    ```

  - **`PyObject_AsWriteBuffer()`**: Use `PyObject_GetBuffer()` and `PyBuffer_Release()` instead:

    ```
    Py_buffer view;
    if (PyObject_GetBuffer(obj, &view, PyBUF_WRITABLE) < 0)
    {
        return NULL;
    }
    // Use `view.buf` and `view.len` to write to the
    buffer.
    PyBuffer_Release(&view);
    ```

  (Contributed by Inada Naoki in gh-85275 [https://github.com/python/cpython/issues/85275].)

- Remove various functions deprecated in Python 3.9:

  - **`PyEval_CallObject()`**, **`PyEval_CallObjectWithKeywords()`**: Use `PyObject_CallNoArgs()` or `PyObject_Call()` instead.

    Warning

    In `PyObject_Call()`, positional arguments must be a `tuple` and must not be `NULL`, and keyword arguments must be a `dict` or `NULL`, whereas the removed functions checked argument types and accepted `NULL` positional and keyword arguments. To replace `PyEval_CallObjectWithKeywords(func, NULL, kwargs)` with `PyObject_Call()`, pass an empty tuple as positional arguments using `PyTuple_New(0)`.

  - **`PyEval_CallFunction()`**: Use `PyObject_CallFunction()` instead.

  - **`PyEval_CallMethod()`**: Use `PyObject_CallMethod()` instead.

  - **`PyCFunction_Call()`**: Use `PyObject_Call()` instead.

  (Contributed by Victor Stinner in gh-105107 [https://github.com/python/cpython/issues/105107].)

- Remove the following old functions to configure the Python initialization, deprecated in Python 3.11:

  - **`PySys_AddWarnOptionUnicode()`**: Use `PyConfig.warnoptions` instead.
  - **`PySys_AddWarnOption()`**: Use `PyConfig.warnoptions` instead.
  - **`PySys_AddXOption()`**: Use `PyConfig.xoptions` instead.
  - **`PySys_HasWarnOptions()`**: Use `PyConfig.xoptions` instead.
  - **`PySys_SetPath()`**: Set `PyConfig.module_search_paths` instead.
  - **`Py_SetPath()`**: Set `PyConfig.module_search_paths` instead.
  - **`Py_SetStandardStreamEncoding()`**: Set `PyConfig.stdio_encoding` instead, and set also maybe `PyConfig.legacy_windows_stdio` (on Windows).
  - **`_Py_SetProgramFullPath()`**: Set `PyConfig.executable` instead.

Use the new `PyConfig` API of the Python Initialization Configuration instead (**PEP 587** [https://peps.python.org/pep-0587/]), added to Python 3.8. (Contributed by Victor Stinner in gh-105145 [https://github.com/python/cpython/issues/105145].)

- Remove **`PyEval_AcquireLock()`** and **`PyEval_ReleaseLock()`** functions, deprecated in Python 3.2. They didn't update the current thread state. They can be replaced with:

  - `PyEval_SaveThread()` and `PyEval_RestoreThread()`;
  - low-level `PyEval_AcquireThread()` and `PyEval_RestoreThread()`;
  - or `PyGILState_Ensure()` and `PyGILState_Release()`.

  (Contributed by Victor Stinner in gh-105182 [https://github.com/python/cpython/issues/105182].)

- Remove the **`PyEval_ThreadsInitialized()`** function, deprecated in Python 3.9. Since Python 3.7, **`Py_Initialize()`** always creates the GIL: calling **`PyEval_InitThreads()`** does nothing and **`PyEval_ThreadsInitialized()`** always returns non-zero. (Contributed by Victor Stinner in gh-105182 [https://github.com/python/cpython/issues/105182].)

- Remove the **`_PyInterpreterState_Get()`** alias to `PyInterpreterState_Get()` which was kept for backward compatibility with Python 3.8. The pythoncapi-compat project [https://github.com/python/pythoncapi-compat/] can be used to get `PyInterpreterState_Get()` on Python 3.8 and older. (Contributed by Victor Stinner in gh-106320 [https://github.com/python/cpython/issues/106320].)

- Remove the private **`_PyObject_FastCall()`** function: use **`PyObject_Vectorcall()`** which is available since Python 3.8 (**PEP 590** [https://peps.python.org/pep-0590/]). (Contributed by

Victor Stinner in gh-106023 [https://github.com/python/cpython/issues/106023].)

- Remove the `cpython/pytime.h` header file, which only contained private functions. (Contributed by Victor Stinner in gh-106316 [https://github.com/python/cpython/issues/106316].)
- Remove the undocumented `PY_TIMEOUT_MAX` constant from the limited C API. (Contributed by Victor Stinner in gh-110014 [https://github.com/python/cpython/issues/110014].)
- Remove the old trashcan macros `Py_TRASHCAN_SAFE_BEGIN` and `Py_TRASHCAN_SAFE_END`. Replace both with the new macros `Py_TRASHCAN_BEGIN` and `Py_TRASHCAN_END`. (Contributed by Irit Katriel in gh-105111 [https://github.com/python/cpython/issues/105111].)

Deprecated C APIs

- Deprecate old Python initialization functions:
    - `PySys_ResetWarnOptions()`: Clear `sys.warnoptions` and `warnings.filters` instead.
    - `Py_GetExecPrefix()`: Get `sys.exec_prefix` instead.
    - `Py_GetPath()`: Get `sys.path` instead.
    - `Py_GetPrefix()`: Get `sys.prefix` instead.
    - `Py_GetProgramFullPath()`: Get `sys.executable` instead.
    - `Py_GetProgramName()`: Get `sys.executable` instead.
    - `Py_GetPythonHome()`: Get `PyConfig.home` or the `PYTHONHOME` environment variable instead.

  (Contributed by Victor Stinner in gh-105145 [https://github.com/python/cpython/issues/105145].)

- Soft deprecate the `PyEval_GetBuiltins()`, `PyEval_GetGlobals()`, and `PyEval_GetLocals()` functions, which return a borrowed reference. (Soft deprecated as part of **PEP 667** [https://peps.python.org/pep-0667/].)

- Deprecate the **`PyImport_ImportModuleNoBlock()`** function, which is just an alias to **`PyImport_ImportModule()`** since Python 3.3. (Contributed by Victor Stinner in gh-105396 [https://github.com/python/cpython/issues/105396].)

- Soft deprecate the **`PyModule_AddObject()`** function. It should be replaced with **`PyModule_Add()`** or **`PyModule_AddObjectRef()`**. (Contributed by Serhiy Storchaka in gh-86493 [https://github.com/python/cpython/issues/86493].)

- Deprecate the old `Py_UNICODE` and `PY_UNICODE_TYPE` types and the **`Py_UNICODE_WIDE`** define. Use the **`wchar_t`** type directly instead. Since Python 3.3, `Py_UNICODE` and `PY_UNICODE_TYPE` are just aliases to **`wchar_t`**. (Contributed by Victor Stinner in gh-105156 [https://github.com/python/cpython/issues/105156].)

- Deprecate the **`PyWeakref_GetObject()`** and **`PyWeakref_GET_OBJECT()`** functions, which return a borrowed reference. Replace them with the new **`PyWeakref_GetRef()`** function, which returns a strong reference. The pythoncapi-compat project [https://github.com/python/pythoncapi-compat/] can be used to get **`PyWeakref_GetRef()`** on Python 3.12 and older. (Contributed by Victor Stinner in gh-105927 [https://github.com/python/cpython/issues/105927].)

Pending removal in Python 3.14

- The `ma_version_tag` field in **`PyDictObject`** for extension modules (**PEP 699** [https://peps.python.org/pep-0699/]; gh-101193 [https://github.com/python/cpython/issues/101193]).

- Creating **`immutable types`** with mutable bases (gh-95388 [https://github.com/python/cpython/issues/95388]).

Pending removal in Python 3.15

- The **`PyImport_ImportModuleNoBlock()`**: Use **`PyImport_ImportModule()`** instead.

- **`PyWeakref_GetObject()`** and **`PyWeakref_GET_OBJECT()`**: Use **`PyWeakref_GetRef()`** instead. The pythoncapi-compat project [https://github.com/python/pythoncapi-compat/] can be used to get **`PyWeakref_GetRef()`** on Python 3.12 and older.

- **`Py_UNICODE`** type and the **`Py_UNICODE_WIDE`** macro: Use **`wchar_t`** instead.

- **`PyUnicode_AsDecodedObject()`**: Use **`PyCodec_Decode()`** instead.

- **`PyUnicode_AsDecodedUnicode()`**: Use **`PyCodec_Decode()`** instead; Note that some codecs (for example, “base64”) may return a type other than **`str`**, such as **`bytes`**.

- **`PyUnicode_AsEncodedObject()`**: Use **`PyCodec_Encode()`** instead.

- **`PyUnicode_AsEncodedUnicode()`**: Use **`PyCodec_Encode()`** instead; Note that some codecs (for example, “base64”) may return a type other than **`bytes`**, such as **`str`**.

- Python initialization functions, deprecated in Python 3.13:
  - **`Py_GetPath()`**: Use **`PyConfig_Get("module_search_paths")`** (**`sys.path`**) instead.
  - **`Py_GetPrefix()`**: Use **`PyConfig_Get("base_prefix")`** (**`sys.base_prefix`**) instead. Use **`PyConfig_Get("prefix")`** (**`sys.prefix`**) if virtual environments need to be handled.

  - `Py_GetExecPrefix()`: Use `PyConfig_Get("base_exec_prefix")` (`sys.base_exec_prefix`) instead. Use `PyConfig_Get("exec_prefix")` (`sys.exec_prefix`) if virtual environments need to be handled.
  - `Py_GetProgramFullPath()`: Use `PyConfig_Get("executable")` (`sys.executable`) instead.
  - `Py_GetProgramName()`: Use `PyConfig_Get("executable")` (`sys.executable`) instead.
  - `Py_GetPythonHome()`: Use `PyConfig_Get("home")` or the `PYTHONHOME` environment variable instead.

  The pythoncapi-compat project [https://github.com/python/pythoncapi-compat/] can be used to get `PyConfig_Get()` on Python 3.13 and older.

- Functions to configure Python's initialization, deprecated in Python 3.11:

  - **`PySys_SetArgvEx()`**: Set `PyConfig.argv` instead.
  - **`PySys_SetArgv()`**: Set `PyConfig.argv` instead.
  - **`Py_SetProgramName()`**: Set `PyConfig.program_name` instead.
  - **`Py_SetPythonHome()`**: Set `PyConfig.home` instead.
  - `PySys_ResetWarnOptions()`: Clear `sys.warnoptions` and **`warnings.filters`** instead.

  The `Py_InitializeFromConfig()` API should be used with `PyConfig` instead.

- Global configuration variables:

  - `Py_DebugFlag`: Use `PyConfig.parser_debug` or `PyConfig_Get("parser_debug")` instead.
  - `Py_VerboseFlag`: Use `PyConfig.verbose` or `PyConfig_Get("verbose")` instead.
  - `Py_QuietFlag`: Use `PyConfig.quiet` or `PyConfig_Get("quiet")` instead.

  - `Py_InteractiveFlag`: Use `PyConfig.interactive` or `PyConfig_Get("interactive")` instead.
  - `Py_InspectFlag`: Use `PyConfig.inspect` or `PyConfig_Get("inspect")` instead.
  - `Py_OptimizeFlag`: Use `PyConfig.optimization_level` or `PyConfig_Get("optimization_level")` instead.
  - `Py_NoSiteFlag`: Use `PyConfig.site_import` or `PyConfig_Get("site_import")` instead.
  - `Py_BytesWarningFlag`: Use `PyConfig.bytes_warning` or `PyConfig_Get("bytes_warning")` instead.
  - `Py_FrozenFlag`: Use `PyConfig.pathconfig_warnings` or `PyConfig_Get("pathconfig_warnings")` instead.
  - `Py_IgnoreEnvironmentFlag`: Use `PyConfig.use_environment` or `PyConfig_Get("use_environment")` instead.
  - `Py_DontWriteBytecodeFlag`: Use `PyConfig.write_bytecode` or `PyConfig_Get("write_bytecode")` instead.
  - `Py_NoUserSiteDirectory`: Use `PyConfig.user_site_directory` or `PyConfig_Get("user_site_directory")` instead.
  - `Py_UnbufferedStdioFlag`: Use `PyConfig.buffered_stdio` or `PyConfig_Get("buffered_stdio")` instead.
  - `Py_HashRandomizationFlag`: Use `PyConfig.use_hash_seed` and `PyConfig.hash_seed` or `PyConfig_Get("hash_seed")` instead.
  - `Py_IsolatedFlag`: Use `PyConfig.isolated` or `PyConfig_Get("isolated")` instead.
  - `Py_LegacyWindowsFSEncodingFlag`: Use `PyPreConfig.legacy_windows_fs_encoding` or `PyConfig_Get("legacy_windows_fs_encoding")` instead.
  - `Py_LegacyWindowsStdioFlag`: Use `PyConfig.legacy_windows_stdio` or `PyConfig_Get("legacy_windows_stdio")` instead.
  - **`Py_FileSystemDefaultEncoding`**, **`Py_HasFileSystemDefaultEncoding`**: Use `PyConfig.filesystem_encoding` or `PyConfig_Get("filesystem_encoding")` instead.

  - **`Py_FileSystemDefaultEncodeErrors`**: Use `PyConfig.filesystem_errors` or `PyConfig_Get("filesystem_errors")` instead.
  - **`Py_UTF8Mode`**: Use `PyPreConfig.utf8_mode` or `PyConfig_Get("utf8_mode")` instead. (see `Py_PreInitialize()`)

The `Py_InitializeFromConfig()` API should be used with `PyConfig` to set these options. Or `PyConfig_Get()` can be used to get these options at runtime.

Pending removal in Python 3.16

- The bundled copy of `libmpdec`.

Pending removal in Python 3.18

- The following private functions are deprecated and planned for removal in Python 3.18:
  - **`_PyBytes_Join()`**: use `PyBytes_Join()`.
  - **`_PyDict_GetItemStringWithError()`**: use `PyDict_GetItemStringRef()`.
  - **`_PyDict_Pop()`**: use `PyDict_Pop()`.
  - **`_PyLong_Sign()`**: use `PyLong_GetSign()`.
  - **`_PyLong_FromDigits()`** and **`_PyLong_New()`**: use `PyLongWriter_Create()`.
  - **`_PyThreadState_UncheckedGet()`**: use `PyThreadState_GetUnchecked()`.
  - **`_PyUnicode_AsString()`**: use `PyUnicode_AsUTF8()`.
  - **`_PyUnicodeWriter_Init()`**: replace `_PyUnicodeWriter_Init(&writer)` with `writer = PyUnicodeWriter_Create(0)`.
  - **`_PyUnicodeWriter_Finish()`**: replace `_PyUnicodeWriter_Finish(&writer)` with

    PyUnicodeWriter_Finish(writer).

  - **_PyUnicodeWriter_Dealloc()**: replace `_PyUnicodeWriter_Dealloc(&writer)` with `PyUnicodeWriter_Discard(writer)`.
  - **_PyUnicodeWriter_WriteChar()**: replace `_PyUnicodeWriter_WriteChar(&writer, ch)` with `PyUnicodeWriter_WriteChar(writer, ch)`.
  - **_PyUnicodeWriter_WriteStr()**: replace `_PyUnicodeWriter_WriteStr(&writer, str)` with `PyUnicodeWriter_WriteStr(writer, str)`.
  - **_PyUnicodeWriter_WriteSubstring()**: replace `_PyUnicodeWriter_WriteSubstring(&writer, str, start, end)` with `PyUnicodeWriter_WriteSubstring(writer, str, start, end)`.
  - **_PyUnicodeWriter_WriteASCIIString()**: replace `_PyUnicodeWriter_WriteASCIIString(&writer, str)` with `PyUnicodeWriter_WriteASCII(writer, str)`.
  - **_PyUnicodeWriter_WriteLatin1String()**: replace `_PyUnicodeWriter_WriteLatin1String(&writer, str)` with `PyUnicodeWriter_WriteUTF8(writer, str)`.
  - **_PyUnicodeWriter_Prepare()**: (no replacement).
  - **_PyUnicodeWriter_PrepareKind()**: (no replacement).
  - **_Py_HashPointer()**: use `Py_HashPointer()`.
  - **_Py_fopen_obj()**: use `Py_fopen()`.

The pythoncapi-compat project [https://github.com/python/pythoncapi-compat/] can be used to get these new public functions on Python 3.13 and older. (Contributed by Victor Stinner in gh-128863 [https://github.com/python/cpython/issues/128863].)

Pending removal in future versions

The following APIs are deprecated and will be removed, although there is currently no date scheduled for their removal.

- `Py_TPFLAGS_HAVE_FINALIZE`: Unneeded since Python 3.8.
- `PyErr_Fetch()`: Use `PyErr_GetRaisedException()` instead.
- `PyErr_NormalizeException()`: Use `PyErr_GetRaisedException()` instead.
- `PyErr_Restore()`: Use `PyErr_SetRaisedException()` instead.
- `PyModule_GetFilename()`: Use `PyModule_GetFilenameObject()` instead.
- `PyOS_AfterFork()`: Use `PyOS_AfterFork_Child()` instead.
- `PySlice_GetIndicesEx()`: Use `PySlice_Unpack()` and `PySlice_AdjustIndices()` instead.
- `PyUnicode_READY()`: Unneeded since Python 3.12
- `PyErr_Display()`: Use `PyErr_DisplayException()` instead.
- `_PyErr_ChainExceptions()`: Use `_PyErr_ChainExceptions1()` instead.
- `PyBytesObject.ob_shash` member: call `PyObject_Hash()` instead.
- Thread Local Storage (TLS) API:
    - `PyThread_create_key()`: Use `PyThread_tss_alloc()` instead.
    - `PyThread_delete_key()`: Use `PyThread_tss_free()` instead.
    - `PyThread_set_key_value()`: Use `PyThread_tss_set()` instead.
    - `PyThread_get_key_value()`: Use `PyThread_tss_get()` instead.
    - `PyThread_delete_key_value()`: Use `PyThread_tss_delete()` instead.
    - `PyThread_ReInitTLS()`: Unneeded since Python 3.7.

Build Changes

- `arm64-apple-ios` and `arm64-apple-ios-simulator` are both now **PEP 11** [https://peps.python.org/pep-0011/] tier 3 platforms. (PEP 730 written and implementation contributed by

Russell Keith-Magee in gh-114099 [https://github.com/python/cpython/issues/114099].)

- `aarch64-linux-android` and `x86_64-linux-android` are both now **PEP 11** [https://peps.python.org/pep-0011/] tier 3 platforms. (PEP 738 written and implementation contributed by Malcolm Smith in gh-116622 [https://github.com/python/cpython/issues/116622].)
- `wasm32-wasi` is now a **PEP 11** [https://peps.python.org/pep-0011/] tier 2 platform. (Contributed by Brett Cannon in gh-115192 [https://github.com/python/cpython/issues/115192].)
- `wasm32-emscripten` is no longer a **PEP 11** [https://peps.python.org/pep-0011/] supported platform. (Contributed by Brett Cannon in gh-115192 [https://github.com/python/cpython/issues/115192].)
- Building CPython now requires a compiler with support for the C11 atomic library, GCC built-in atomic functions, or MSVC interlocked intrinsics.
- Autoconf 2.71 and aclocal 1.16.5 are now required to regenerate the `configure` script. (Contributed by Christian Heimes in gh-89886 [https://github.com/python/cpython/issues/89886] and by Victor Stinner in gh-112090 [https://github.com/python/cpython/issues/112090].)
- SQLite 3.15.2 or newer is required to build the **`sqlite3`** extension module. (Contributed by Erlend Aasland in gh-105875 [https://github.com/python/cpython/issues/105875].)
- CPython now bundles the mimalloc library [https://github.com/microsoft/mimalloc/] by default. It is licensed under the MIT license; see mimalloc license. The bundled mimalloc has custom changes, see gh-113141 [https://github.com/python/cpython/issues/113141] for details. (Contributed by Dino Viehland in gh-109914 [https://github.com/python/cpython/issues/109914].)
- The `configure` option **`--with-system-libmpdec`** now defaults to `yes`. The bundled copy of `libmpdec` will be removed in Python 3.16.

- Python built with `configure` **`--with-trace-refs`** (tracing references) is now ABI compatible with the Python release build and debug build. (Contributed by Victor Stinner in gh-108634 [https://github.com/python/cpython/issues/108634].)
- On POSIX systems, the pkg-config (`.pc`) filenames now include the ABI flags. For example, the free-threaded build generates `python-3.13t.pc` and the debug build generates `python-3.13d.pc`.
- The `errno`, `fcntl`, `grp`, `md5`, `pwd`, `resource`, `termios`, `winsound`, `_ctypes_test`, `_multiprocessing.posixshmem`, `_scproxy`, `_stat`, `_statistics`, `_testconsole`, `_testimportmultiple` and `_uuid` C extensions are now built with the limited C API. (Contributed by Victor Stinner in gh-85283 [https://github.com/python/cpython/issues/85283].)

Porting to Python 3.13

This section lists previously described changes and other bugfixes that may require changes to your code.

Changes in the Python API

- PEP 667 introduces several changes to the semantics of **`locals()`** and **`f_locals`**:

  - Calling **`locals()`** in an optimized scope now produces an independent snapshot on each call, and hence no longer implicitly updates previously returned references. Obtaining the legacy CPython behavior now requires explicit calls to update the initially returned dictionary with the results of subsequent calls to **`locals()`**. Code execution functions that implicitly target **`locals()`** (such as `exec` and `eval`) must be passed an explicit namespace to access their

    results in an optimized scope. (Changed as part of **PEP 667** [https://peps.python.org/pep-0667/].)
  - Calling **`locals()`** from a comprehension at module or class scope (including via `exec` or `eval`) once more behaves as if the comprehension were running as an independent nested function (i.e. the local variables from the containing scope are not included). In Python 3.12, this had changed to include the local variables from the containing scope when implementing **PEP 709** [https://peps.python.org/pep-0709/]. (Changed as part of **PEP 667** [https://peps.python.org/pep-0667/].)
  - Accessing **`FrameType.f_locals`** in an optimized scope now returns a write-through proxy rather than a snapshot that gets updated at ill-specified times. If a snapshot is desired, it must be created explicitly with `dict` or the proxy's `.copy()` method. (Changed as part of **PEP 667** [https://peps.python.org/pep-0667/].)

- **`functools.partial`** now emits a **`FutureWarning`** when used as a method. The behavior will change in future Python versions. Wrap it in **`staticmethod()`** if you want to preserve the old behavior. (Contributed by Serhiy Storchaka in gh-121027 [https://github.com/python/cpython/issues/121027].)

- An **`OSError`** is now raised by **`getpass.getuser()`** for any failure to retrieve a username, instead of **`ImportError`** on non-Unix platforms or **`KeyError`** on Unix platforms where the password database is empty.

- The value of the **`mode`** attribute of **`gzip.GzipFile`** is now a string (`'rb'` or `'wb'`) instead of an integer (`1` or `2`). The value of the **`mode`** attribute of the readable file-like object returned by **`zipfile.ZipFile.open()`** is now `'rb'` instead of `'r'`. (Contributed by Serhiy Storchaka in gh-115961 [https://github.com/python/cpython/issues/115961].)

- **`mailbox.Maildir`** now ignores files with a leading dot (`.`). (Contributed by Zackery Spytz in gh-65559 [https://github.com/python/cpython/issues/65559].)
- **`pathlib.Path.glob()`** and **`rglob()`** now return both files and directories if a pattern that ends with "`**`" is given, rather than directories only. Add a trailing slash to keep the previous behavior and only match directories.
- The **`threading`** module now expects the **`_thread`** module to have an **`_is_main_interpreter()`** function. This function takes no arguments and returns `True` if the current interpreter is the main interpreter.

  Any library or application that provides a custom **`_thread`** module must provide **`_is_main_interpreter()`**, just like the module's other "private" attributes. (gh-112826 [https://github.com/python/cpython/issues/112826].)

Changes in the C API

- `Python.h` no longer includes the `<ieeefp.h>` standard header. It was included for the **`finite()`** function which is now provided by the `<math.h>` header. It should now be included explicitly if needed. Remove also the `HAVE_IEEEFP_H` macro. (Contributed by Victor Stinner in gh-108765 [https://github.com/python/cpython/issues/108765].)
- `Python.h` no longer includes these standard header files: `<time.h>`, `<sys/select.h>` and `<sys/time.h>`. If needed, they should now be included explicitly. For example, `<time.h>` provides the **`clock()`** and **`gmtime()`** functions, `<sys/select.h>` provides the **`select()`** function, and

`<sys/time.h>` provides the **`futimes()`**, **`gettimeofday()`** and **`setitimer()`** functions. (Contributed by Victor Stinner in gh-108765 [https://github.com/python/cpython/issues/108765].)

- On Windows, `Python.h` no longer includes the `<stddef.h>` standard header file. If needed, it should now be included explicitly. For example, it provides **`offsetof()`** function, and `size_t` and `ptrdiff_t` types. Including `<stddef.h>` explicitly was already needed by all other platforms, the `HAVE_STDDEF_H` macro is only defined on Windows. (Contributed by Victor Stinner in gh-108765 [https://github.com/python/cpython/issues/108765].)
- If the **`Py_LIMITED_API`** macro is defined, **`Py_BUILD_CORE`**, **`Py_BUILD_CORE_BUILTIN`** and **`Py_BUILD_CORE_MODULE`** macros are now undefined by `<Python.h>`. (Contributed by Victor Stinner in gh-85283 [https://github.com/python/cpython/issues/85283].)
- The old trashcan macros `Py_TRASHCAN_SAFE_BEGIN` and `Py_TRASHCAN_SAFE_END` were removed. They should be replaced by the new macros `Py_TRASHCAN_BEGIN` and `Py_TRASHCAN_END`.

  A `tp_dealloc` function that has the old macros, such as:

  ```
  static void
  mytype_dealloc(mytype *p)
  {
      PyObject_GC_UnTrack(p);
      Py_TRASHCAN_SAFE_BEGIN(p);
      ...
      Py_TRASHCAN_SAFE_END
  }
  ```

  should migrate to the new macros as follows:

```
static void
mytype_dealloc(mytype *p)
{
    PyObject_GC_UnTrack(p);
    Py_TRASHCAN_BEGIN(p, mytype_dealloc)
    ...
    Py_TRASHCAN_END
}
```

Note that `Py_TRASHCAN_BEGIN` has a second argument which should be the deallocation function it is in. The new macros were added in Python 3.8 and the old macros were deprecated in Python 3.11. (Contributed by Irit Katriel in gh-105111 [https://github.com/python/cpython/issues/105111].)

- PEP 667 introduces several changes to frame-related functions:
  - The effects of mutating the dictionary returned from **`PyEval_GetLocals()`** in an optimized scope have changed. New dict entries added this way will now *only* be visible to subsequent **`PyEval_GetLocals()`** calls in that frame, as **`PyFrame_GetLocals()`**, **`locals()`**, and **`FrameType.f_locals`** no longer access the same underlying cached dictionary. Changes made to entries for actual variable names and names added via the write-through proxy interfaces will be overwritten on subsequent calls to **`PyEval_GetLocals()`** in that frame. The recommended code update depends on how the function was being used, so refer to the deprecation notice on the function for details.
  - Calling **`PyFrame_GetLocals()`** in an optimized scope now returns a write-through proxy rather than a snapshot that gets updated at ill-specified times. If a snapshot is desired, it must be created explicitly

  (e.g. with `PyDict_Copy()`), or by calling the new `PyEval_GetFrameLocals()` API.
  - **`PyFrame_FastToLocals()`** and **`PyFrame_FastToLocalsWithError()`** no longer have any effect. Calling these functions has been redundant since Python 3.11, when `PyFrame_GetLocals()` was first introduced.
  - **`PyFrame_LocalsToFast()`** no longer has any effect. Calling this function is redundant now that `PyFrame_GetLocals()` returns a write-through proxy for optimized scopes.

- Python 3.13 removed many private functions. Some of them can be replaced using these alternatives:

  - `_PyDict_Pop()`: `PyDict_Pop()` or `PyDict_PopString()`;
  - `_PyDict_GetItemWithError()`: `PyDict_GetItemRef()`;
  - `_PyErr_WriteUnraisableMsg()`: `PyErr_FormatUnraisable()`;
  - `_PyEval_SetTrace()`: `PyEval_SetTrace()` or `PyEval_SetTraceAllThreads()`;
  - `_PyList_Extend()`: `PyList_Extend()`;
  - `_PyLong_AsInt()`: `PyLong_AsInt()`;
  - `_PyMem_RawStrdup()`: `strdup()`;
  - `_PyMem_Strdup()`: `strdup()`;
  - `_PyObject_ClearManagedDict()`: `PyObject_ClearManagedDict()`;
  - `_PyObject_VisitManagedDict()`: `PyObject_VisitManagedDict()`;
  - `_PyThreadState_UncheckedGet()`: `PyThreadState_GetUnchecked()`;
  - `_PyTime_AsSecondsDouble()`: `PyTime_AsSecondsDouble()`;
  - `_PyTime_GetMonotonicClock()`: `PyTime_Monotonic()` or `PyTime_MonotonicRaw()`;
  - `_PyTime_GetPerfCounter()`: `PyTime_PerfCounter()` or `PyTime_PerfCounterRaw()`;

  - `_PyTime_GetSystemClock()`: **`PyTime_Time()`** or **`PyTime_TimeRaw()`**;
  - `_PyTime_MAX`: **`PyTime_MAX`**;
  - `_PyTime_MIN`: **`PyTime_MIN`**;
  - `_PyTime_t`: **`PyTime_t`**;
  - `_Py_HashPointer()`: **`Py_HashPointer()`**;
  - `_Py_IsFinalizing()`: **`Py_IsFinalizing()`**.

The pythoncapi-compat project [https://github.com/python/pythoncapi-compat/] can be used to get most of these new functions on Python 3.12 and older.

Regression Test Changes

- Python built with `configure` **`--with-pydebug`** now supports a **`-X presite=package.module`** command-line option. If used, it specifies a module that should be imported early in the lifecycle of the interpreter, before `site.py` is executed. (Contributed by Łukasz Langa in gh-110769 [https://github.com/python/cpython/issues/110769].)

What’s New In Python 3.12

Editor:: Adam Turner

This article explains the new features in Python 3.12, compared to 3.11. Python 3.12 was released on October 2, 2023. For full details, see the changelog.

See also

PEP 693 [https://peps.python.org/pep-0693/] – Python 3.12 Release Schedule

Summary – Release highlights

Python 3.12 is a stable release of the Python programming language, with a mix of changes to the language and the standard library. The library changes focus on cleaning up deprecated APIs, usability, and correctness. Of note, the **`distutils`** package has been removed from the standard library. Filesystem support in **`os`** and **`pathlib`** has seen a number of improvements, and several modules have better performance.

The language changes focus on usability, as f-strings have had many limitations removed and ‘Did you mean …’ suggestions continue to improve. The new type parameter syntax and **`type`** statement improve ergonomics for using generic types and type aliases with static type checkers.

This article doesn’t attempt to provide a complete specification of all new features, but instead gives a convenient overview. For full details, you should refer to the documentation, such as the Library Reference and Language Reference. If you want to understand the complete implementation and design rationale for a change, refer to the PEP for a particular new feature; but note that PEPs usually are not kept up-to-date once a feature has been fully implemented.

New syntax features:

- PEP 695, type parameter syntax and the **`type`** statement

New grammar features:

- PEP 701, f-strings in the grammar

Interpreter improvements:

- PEP 684, a unique per-interpreter GIL
- PEP 669, low impact monitoring
- Improved ‘Did you mean …’ suggestions for **`NameError`**, **`ImportError`**, and **`SyntaxError`** exceptions

Python data model improvements:

- PEP 688, using the buffer protocol from Python

Significant improvements in the standard library:

- The **`pathlib.Path`** class now supports subclassing
- The **`os`** module received several improvements for Windows support
- A command-line interface has been added to the **`sqlite3`** module
- **`isinstance()`** checks against **`runtime-checkable protocols`** enjoy a speed up of between two and 20 times
- The **`asyncio`** package has had a number of performance improvements, with some benchmarks showing a 75% speed up.
- A command-line interface has been added to the **`uuid`** module
- Due to the changes in PEP 701, producing tokens via the **`tokenize`** module is up to 64% faster.

Security improvements:

- Replace the builtin **`hashlib`** implementations of SHA1, SHA3, SHA2-384, SHA2-512, and MD5 with formally verified code from the HACL* [https://github.com/hacl-star/hacl-star/] project. These builtin implementations remain as fallbacks that are only used when OpenSSL does not provide them.

C API improvements:

- PEP 697, unstable C API tier
- PEP 683, immortal objects

CPython implementation improvements:

- PEP 709, comprehension inlining
- CPython support for the Linux `perf` profiler
- Implement stack overflow protection on supported platforms

New typing features:

- PEP 692, using **`TypedDict`** to annotate **kwargs
- PEP 698, **`typing.override()`** decorator

Important deprecations, removals or restrictions:

- **PEP 623** [https://peps.python.org/pep-0623/]: Remove `wstr` from Unicode objects in Python's C API, reducing the size of every **`str`** object by at least 8 bytes.
- **PEP 632** [https://peps.python.org/pep-0632/]: Remove the **`distutils`** package. See **the migration guide** [https://peps.python.org/pep-0632/#migration-advice] for advice replacing the APIs it provided. The third-party Setuptools [https://setuptools.pypa.io/en/latest/deprecated/distutils-legacy.html] package continues to provide **`distutils`**, if you still require it in Python 3.12 and beyond.
- gh-95299 [https://github.com/python/cpython/issues/95299]: Do not pre-install `setuptools` in virtual environments created with **`venv`**. This means that `distutils`, `setuptools`, `pkg_resources`, and `easy_install` will no longer available by default; to access these run `pip install setuptools` in the activated virtual environment.

- The **asynchat**, **asyncore**, and **imp** modules have been removed, along with several **unittest.TestCase** method aliases.

New Features

PEP 695: Type Parameter Syntax

Generic classes and functions under **PEP 484** [https://peps.python.org/pep-0484/] were declared using a verbose syntax that left the scope of type parameters unclear and required explicit declarations of variance.

PEP 695 [https://peps.python.org/pep-0695/] introduces a new, more compact and explicit way to create generic classes and functions:

```
def max[T](args: Iterable[T]) -> T:
    ...

class list[T]:
    def __getitem__(self, index: int, /) -> T:
        ...

    def append(self, element: T) -> None:
        ...
```

In addition, the PEP introduces a new way to declare type aliases using the **type** statement, which creates an instance of **TypeAliasType**:

```
type Point = tuple[float, float]
```

Type aliases can also be generic:

```
type Point[T] = tuple[T, T]
```

The new syntax allows declaring **TypeVarTuple** and **ParamSpec** parameters, as well as **TypeVar** parameters with bounds or constraints:

```
type IntFunc[**P] = Callable[P, int]  # ParamSpec
type LabeledTuple[*Ts] = tuple[str, *Ts]  # TypeVarTuple
type HashableSequence[T: Hashable] = Sequence[T]  # TypeVar with
bound
```

```
type IntOrStrSequence[T: (int, str)] = Sequence[T]  # TypeVar with
constraints
```

The value of type aliases and the bound and constraints of type variables created through this syntax are evaluated only on demand (see lazy evaluation). This means type aliases are able to refer to other types defined later in the file.

Type parameters declared through a type parameter list are visible within the scope of the declaration and any nested scopes, but not in the outer scope. For example, they can be used in the type annotations for the methods of a generic class or in the class body. However, they cannot be used in the module scope after the class is defined. See Type parameter lists for a detailed description of the runtime semantics of type parameters.

In order to support these scoping semantics, a new kind of scope is introduced, the annotation scope. Annotation scopes behave for the most part like function scopes, but interact differently with enclosing class scopes. In Python 3.13, annotations will also be evaluated in annotation scopes.

See **PEP 695** [https://peps.python.org/pep-0695/] for more details.

(PEP written by Eric Traut. Implementation by Jelle Zijlstra, Eric Traut, and others in gh-103764 [https://github.com/python/cpython/issues/103764].)

PEP 701: Syntactic formalization of f-strings

PEP 701 [https://peps.python.org/pep-0701/] lifts some restrictions on the usage of f-strings. Expression components inside f-strings can now be any valid Python expression, including strings reusing the same quote as the containing f-string, multi-line expressions, comments, backslashes, and unicode escape sequences. Let’s cover these in detail:

- Quote reuse: in Python 3.11, reusing the same quotes as the enclosing f-string raises a `SyntaxError`, forcing the user to either use other available quotes (like using double quotes or triple quotes if the f-string uses single quotes). In Python 3.12, you can now do things like this:

```
>>> songs = ['Take me back to Eden', 'Alkaline', 'Ascensionism']
>>> f"This is the playlist: {", ".join(songs)}"
'This is the playlist: Take me back to Eden, Alkaline,
Ascensionism'
```

Note that before this change there was no explicit limit in how f-strings can be nested, but the fact that string quotes cannot be reused inside the expression component of f-strings made it impossible to nest f-strings arbitrarily. In fact, this is the most nested f-string that could be written:

```
>>> f"""{f'''{f'{f"{1+1}"}'}'''}"""
'2'
```

As now f-strings can contain any valid Python expression inside expression components, it is now possible to nest f-strings arbitrarily:

```
>>> f"{f"{f"{f"{f"{f"{1+1}"}"}"}"}"}"
'2'
```

- Multi-line expressions and comments: In Python 3.11, f-string expressions must be defined in a single line, even if the expression within the f-string could normally span multiple lines (like literal lists being defined over multiple lines), making them harder to read. In Python 3.12 you can now define f-strings spanning multiple lines, and add inline comments:

```
>>> f"This is the playlist: {", ".join([
...     'Take me back to Eden',  # My, my, those eyes like fire
...     'Alkaline',              # Not acid nor alkaline
...     'Ascensionism'           # Take to the broken skies at
last
... ])}"
'This is the playlist: Take me back to Eden, Alkaline,
Ascensionism'
```

- Backslashes and unicode characters: before Python 3.12 f-string expressions couldn’t contain any `\` character. This also affected unicode escape sequences (such as `\N{snowman}`) as these contain the `\N` part that previously could not be part of expression components of f-strings. Now, you can define expressions like this:

```
>>> print(f"This is the playlist: {"\n".join(songs)}")
This is the playlist: Take me back to Eden
Alkaline
Ascensionism
>>> print(f"This is the playlist: {"\N{BLACK HEART
SUIT}".join(songs)}")
This is the playlist: Take me back to Eden♥Alkaline♥Ascensionism
```

See **PEP 701** [https://peps.python.org/pep-0701/] for more details.

As a positive side-effect of how this feature has been implemented (by parsing f-strings with **the PEG parser** [https://peps.python.org/pep-0617/]), now error messages for f-strings are more precise and include the exact location of the error. For example, in Python 3.11, the following f-string raises a **`SyntaxError`**:

```
>>> my_string = f"{x z y}" + f"{1 + 1}"
  File "<stdin>", line 1
    (x z y)
     ^^^
SyntaxError: f-string: invalid syntax. Perhaps you forgot a comma?
```

but the error message doesn’t include the exact location of the error within the line and also has the expression artificially surrounded by parentheses. In Python 3.12, as f-strings are parsed with the PEG parser, error messages can be more precise and show the entire line:

```
>>> my_string = f"{x z y}" + f"{1 + 1}"
  File "<stdin>", line 1
    my_string = f"{x z y}" + f"{1 + 1}"
                   ^^^
SyntaxError: invalid syntax. Perhaps you forgot a comma?
```

(Contributed by Pablo Galindo, Batuhan Taskaya, Lysandros Nikolaou, Cristián Maureira-Fredes and Marta Gómez in gh-102856 [https://github.com/python/cpython/issues/102856]. PEP written by Pablo Galindo, Batuhan Taskaya, Lysandros Nikolaou and Marta Gómez).

PEP 684: A Per-Interpreter GIL

PEP 684 [https://peps.python.org/pep-0684/] introduces a per-interpreter GIL, so that sub-interpreters may now be created with a unique GIL per interpreter. This allows Python programs to take full advantage of multiple CPU cores. This is currently only available through the C-API, though a Python API is **anticipated for 3.13** [https://peps.python.org/pep-0554/].

Use the new `Py_NewInterpreterFromConfig()` function to create an interpreter with its own GIL:

```
PyInterpreterConfig config = {
    .check_multi_interp_extensions = 1,
    .gil = PyInterpreterConfig_OWN_GIL,
};
PyThreadState *tstate = NULL;
PyStatus status = Py_NewInterpreterFromConfig(&tstate, &config);
if (PyStatus_Exception(status)) {
    return -1;
}
/* The new interpreter is now active in the current thread. */
```

For further examples how to use the C-API for sub-interpreters with a per-interpreter GIL, see `Modules/_xxsubinterpretersmodule.c`.

(Contributed by Eric Snow in gh-104210 [https://github.com/python/cpython/issues/104210], etc.)

PEP 669: Low impact monitoring for CPython

PEP 669 [https://peps.python.org/pep-0669/] defines a new `API` for profilers, debuggers, and other tools to monitor events in CPython. It covers a wide range of events, including calls, returns, lines, exceptions, jumps, and more. This means that you only pay for what you use, providing support for near-zero overhead debuggers and coverage tools. See `sys.monitoring` for details.

(Contributed by Mark Shannon in gh-103082 [https://github.com/python/cpython/issues/103082].)

PEP 688: Making the buffer protocol accessible in Python

PEP 688 [https://peps.python.org/pep-0688/] introduces a way to use the buffer protocol from Python code. Classes that implement the `__buffer__()` method are now usable as buffer types.

The new `collections.abc.Buffer` ABC provides a standard way to represent buffer objects, for example in type annotations. The new `inspect.BufferFlags` enum represents the flags that can be used to customize buffer creation. (Contributed by Jelle Zijlstra in gh-102500 [https://github.com/python/cpython/issues/102500].)

PEP 709: Comprehension inlining

Dictionary, list, and set comprehensions are now inlined, rather than creating a new single-use function object for each execution of the comprehension. This speeds up execution of a comprehension by up to two times. See **PEP 709** [https://peps.python.org/pep-0709/] for further details.

Comprehension iteration variables remain isolated and don't overwrite a variable of the same name in the outer scope, nor are they visible after the comprehension. Inlining does result in a few visible behavior changes:

- There is no longer a separate frame for the comprehension in tracebacks, and tracing/profiling no longer shows the comprehension as a function call.
- The `symtable` module will no longer produce child symbol tables for each comprehension; instead, the comprehension's locals will be included in the parent function's symbol table.
- Calling `locals()` inside a comprehension now includes variables from outside the comprehension, and no longer includes the synthetic `.0` variable for the comprehension "argument".
- A comprehension iterating directly over `locals()` (e.g. `[k for k in locals()]`) may see "RuntimeError: dictionary changed size during iteration" when run under tracing (e.g. code coverage measurement). This is the same behavior already seen in e.g.

`for k in locals():`. To avoid the error, first create a list of keys to iterate over: `keys = list(locals()); [k for k in keys]`.

(Contributed by Carl Meyer and Vladimir Matveev in **PEP 709** [https://peps.python.org/pep-0709/].)

Improved Error Messages

- Modules from the standard library are now potentially suggested as part of the error messages displayed by the interpreter when a **`NameError`** is raised to the top level. (Contributed by Pablo Galindo in gh-98254 [https://github.com/python/cpython/issues/98254].)

  ```
  >>> sys.version_info
  Traceback (most recent call last):
    File "<stdin>", line 1, in <module>
  NameError: name 'sys' is not defined. Did you forget to import
  'sys'?
  ```

- Improve the error suggestion for **`NameError`** exceptions for instances. Now if a **`NameError`** is raised in a method and the instance has an attribute that's exactly equal to the name in the exception, the suggestion will include `self.<NAME>` instead of the closest match in the method scope. (Contributed by Pablo Galindo in gh-99139 [https://github.com/python/cpython/issues/99139].)

  ```
  >>> class A:
  ...    def __init__(self):
  ...        self.blech = 1
  ...
  ...    def foo(self):
  ...        somethin = blech
  ...
  >>> A().foo()
  Traceback (most recent call last):
    File "<stdin>", line 1
      somethin = blech
                 ^^^^^
  NameError: name 'blech' is not defined. Did you mean:
  'self.blech'?
  ```

- Improve the **`SyntaxError`** error message when the user types `import x from y` instead of `from y import x`. (Contributed by Pablo Galindo in gh-98931 [https://github.com/python/cpython/issues/98931].)

  ```
  >>> import a.y.z from b.y.z
  Traceback (most recent call last):
    File "<stdin>", line 1
      import a.y.z from b.y.z
      ^^^^^^^^^^^^^^^^^^^^^^^
  SyntaxError: Did you mean to use 'from ... import ...' instead?
  ```

- **`ImportError`** exceptions raised from failed `from <module> import <name>` statements now include suggestions for the value of `<name>` based on the available names in `<module>`. (Contributed by Pablo Galindo in gh-91058 [https://github.com/python/cpython/issues/91058].)

  ```
  >>> from collections import chainmap
  Traceback (most recent call last):
    File "<stdin>", line 1, in <module>
  ImportError: cannot import name 'chainmap' from 'collections'.
  Did you mean: 'ChainMap'?
  ```

New Features Related to Type Hints

This section covers major changes affecting **type hints** [https://peps.python.org/pep-0484/] and the **`typing`** module.

PEP 692: Using `TypedDict` for more precise `**kwargs` typing

Typing `**kwargs` in a function signature as introduced by **PEP 484** [https://peps.python.org/pep-0484/] allowed for valid annotations only in cases where all of the `**kwargs` were of the same type.

PEP 692 [https://peps.python.org/pep-0692/] specifies a more precise way of typing `**kwargs` by relying on typed dictionaries:

```
from typing import TypedDict, Unpack

class Movie(TypedDict):
  name: str
```

```
  year: int

def foo(**kwargs: Unpack[Movie]): ...
```

See **PEP 692** [https://peps.python.org/pep-0692/] for more details.

(Contributed by Franek Magiera in gh-103629 [https://github.com/python/cpython/issues/103629].)

PEP 698: Override Decorator for Static Typing

A new decorator **`typing.override()`** has been added to the **`typing`** module. It indicates to type checkers that the method is intended to override a method in a superclass. This allows type checkers to catch mistakes where a method that is intended to override something in a base class does not in fact do so.

Example:

```
from typing import override

class Base:
  def get_color(self) -> str:
    return "blue"

class GoodChild(Base):
  @override  # ok: overrides Base.get_color
  def get_color(self) -> str:
    return "yellow"

class BadChild(Base):
  @override  # type checker error: does not override Base.get_color
  def get_colour(self) -> str:
    return "red"
```

See **PEP 698** [https://peps.python.org/pep-0698/] for more details.

(Contributed by Steven Troxler in gh-101561 [https://github.com/python/cpython/issues/101561].)

Other Language Changes

- The parser now raises **`SyntaxError`** when parsing source code containing null bytes. (Contributed by Pablo Galindo in gh-96670 [https://github.com/python/cpython/issues/96670].)
- A backslash-character pair that is not a valid escape sequence now generates a **`SyntaxWarning`**, instead of **`DeprecationWarning`**. For example, `re.compile("\d+\.\d+")` now emits a **`SyntaxWarning`** (`"\d"` is an invalid escape sequence, use raw strings for regular expression: `re.compile(r"\d+\.\d+")`). In a future Python version, **`SyntaxError`** will eventually be raised, instead of **`SyntaxWarning`**. (Contributed by Victor Stinner in gh-98401 [https://github.com/python/cpython/issues/98401].)
- Octal escapes with value larger than `0o377` (ex: `"\477"`), deprecated in Python 3.11, now produce a **`SyntaxWarning`**, instead of **`DeprecationWarning`**. In a future Python version they will be eventually a **`SyntaxError`**. (Contributed by Victor Stinner in gh-98401 [https://github.com/python/cpython/issues/98401].)
- Variables used in the target part of comprehensions that are not stored to can now be used in assignment expressions (`:=`). For example, in `[(b := 1) for a, b.prop in some_iter]`, the assignment to `b` is now allowed. Note that assigning to variables stored to in the target part of comprehensions (like `a`) is still disallowed, as per **PEP 572** [https://peps.python.org/pep-0572/]. (Contributed by Nikita Sobolev in gh-100581 [https://github.com/python/cpython/issues/100581].)
- Exceptions raised in a class or type’s `__set_name__` method are no longer wrapped by a **`RuntimeError`**. Context information is added to the exception as a **PEP 678** [https://peps.python.org/pep-0678/] note. (Contributed by Irit Katriel in gh-77757 [https://github.com/python/cpython/issues/77757].)
- When a `try-except*` construct handles the entire **`ExceptionGroup`** and raises one other exception, that exception is no longer wrapped in an **`ExceptionGroup`**. Also changed in version 3.11.4. (Contributed by Irit Katriel in gh-103590 [https://github.com/python/cpython/issues/103590].)
- The Garbage Collector now runs only on the eval breaker mechanism of the Python bytecode evaluation loop instead of object allocations. The GC can also run when **`PyErr_CheckSignals()`** is called so C extensions that need to run for a long time without executing any Python code also have a

chance to execute the GC periodically. (Contributed by Pablo Galindo in gh-97922 [https://github.com/python/cpython/issues/97922].)

- All builtin and extension callables expecting boolean parameters now accept arguments of any type instead of just **`bool`** and **`int`**. (Contributed by Serhiy Storchaka in gh-60203 [https://github.com/python/cpython/issues/60203].)
- **`memoryview`** now supports the half-float type (the “e” format code). (Contributed by Donghee Na and Antoine Pitrou in gh-90751 [https://github.com/python/cpython/issues/90751].)
- **`slice`** objects are now hashable, allowing them to be used as dict keys and set items. (Contributed by Will Bradshaw, Furkan Onder, and Raymond Hettinger in gh-101264 [https://github.com/python/cpython/issues/101264].)
- **`sum()`** now uses Neumaier summation to improve accuracy and commutativity when summing floats or mixed ints and floats. (Contributed by Raymond Hettinger in gh-100425 [https://github.com/python/cpython/issues/100425].)
- **`ast.parse()`** now raises **`SyntaxError`** instead of **`ValueError`** when parsing source code containing null bytes. (Contributed by Pablo Galindo in gh-96670 [https://github.com/python/cpython/issues/96670].)
- The extraction methods in **`tarfile`**, and **`shutil.unpack_archive()`**, have a new a *filter* argument that allows limiting tar features than may be surprising or dangerous, such as creating files outside the destination directory. See tarfile extraction filters for details. In Python 3.14, the default will switch to `'data'`. (Contributed by Petr Viktorin in **PEP 706** [https://peps.python.org/pep-0706/].)
- **`types.MappingProxyType`** instances are now hashable if the underlying mapping is hashable. (Contributed by Serhiy Storchaka in gh-87995 [https://github.com/python/cpython/issues/87995].)
- Add support for the perf profiler through the new environment variable **`PYTHONPERFSUPPORT`** and command-line option **`-X perf`**, as well as the new **`sys.activate_stack_trampoline()`**, **`sys.deactivate_stack_trampoline()`**, and **`sys.is_stack_trampoline_active()`** functions. (Design by Pablo Galindo. Contributed by Pablo Galindo and Christian Heimes with contributions from Gregory P. Smith [Google] and Mark Shannon in gh-96123 [https://github.com/python/cpython/issues/96123].)

New Modules

- None.

Improved Modules

array

- The `array.array` class now supports subscripting, making it a generic type. (Contributed by Jelle Zijlstra in gh-98658 [https://github.com/python/cpython/issues/98658].)

asyncio

- The performance of writing to sockets in `asyncio` has been significantly improved. `asyncio` now avoids unnecessary copying when writing to sockets and uses `sendmsg()` if the platform supports it. (Contributed by Kumar Aditya in gh-91166 [https://github.com/python/cpython/issues/91166].)
- Add `asyncio.eager_task_factory()` and `asyncio.create_eager_task_factory()` functions to allow opting an event loop in to eager task execution, making some use-cases 2x to 5x faster. (Contributed by Jacob Bower & Itamar Oren in gh-102853 [https://github.com/python/cpython/issues/102853], gh-104140 [https://github.com/python/cpython/issues/104140], and gh-104138 [https://github.com/python/cpython/issues/104138])
- On Linux, `asyncio` uses **`asyncio.PidfdChildWatcher`** by default if `os.pidfd_open()` is available and functional instead of **`asyncio.ThreadedChildWatcher`**. (Contributed by Kumar Aditya in gh-98024 [https://github.com/python/cpython/issues/98024].)
- The event loop now uses the best available child watcher for each platform (**`asyncio.PidfdChildWatcher`** if supported and **`asyncio.ThreadedChildWatcher`** otherwise), so manually configuring a child watcher is not recommended. (Contributed by Kumar Aditya in gh-94597 [https://github.com/python/cpython/issues/94597].)
- Add *loop_factory* parameter to `asyncio.run()` to allow specifying a custom event loop factory. (Contributed by Kumar Aditya in

gh-99388 [https://github.com/python/cpython/issues/99388].)

- Add C implementation of **`asyncio.current_task()`** for 4x-6x speedup. (Contributed by Itamar Oren and Pranav Thulasiram Bhat in gh-100344 [https://github.com/python/cpython/issues/100344].)
- **`asyncio.iscoroutine()`** now returns `False` for generators as **`asyncio`** does not support legacy generator-based coroutines. (Contributed by Kumar Aditya in gh-102748 [https://github.com/python/cpython/issues/102748].)
- **`asyncio.wait()`** and **`asyncio.as_completed()`** now accepts generators yielding tasks. (Contributed by Kumar Aditya in gh-78530 [https://github.com/python/cpython/issues/78530].)

calendar

- Add enums **`calendar.Month`** and **`calendar.Day`** defining months of the year and days of the week. (Contributed by Prince Roshan in gh-103636 [https://github.com/python/cpython/issues/103636].)

csv

- Add **`csv.QUOTE_NOTNULL`** and **`csv.QUOTE_STRINGS`** flags to provide finer grained control of `None` and empty strings by **`reader`** and **`writer`** objects.

dis

- Pseudo instruction opcodes (which are used by the compiler but do not appear in executable bytecode) are now exposed in the **`dis`** module. **`HAVE_ARGUMENT`** is still relevant to real opcodes, but it is not useful for pseudo instructions. Use the new **`dis.hasarg`** collection instead. (Contributed by Irit Katriel in gh-94216 [https://github.com/python/cpython/issues/94216].)
- Add the **`dis.hasexc`** collection to signify instructions that set an exception handler. (Contributed by Irit Katriel in gh-94216 [https://github.com/python/cpython/issues/94216].)

fractions

- Objects of type `fractions.Fraction` now support float-style formatting. (Contributed by Mark Dickinson in gh-100161 [https://github.com/python/cpython/issues/100161].)

importlib.resources

- `importlib.resources.as_file()` now supports resource directories. (Contributed by Jason R. Coombs in gh-97930 [https://github.com/python/cpython/issues/97930].)
- Rename first parameter of `importlib.resources.files()` to *anchor*. (Contributed by Jason R. Coombs in gh-100598 [https://github.com/python/cpython/issues/100598].)

inspect

- Add `inspect.markcoroutinefunction()` to mark sync functions that return a coroutine for use with `inspect.iscoroutinefunction()`. (Contributed by Carlton Gibson in gh-99247 [https://github.com/python/cpython/issues/99247].)
- Add `inspect.getasyncgenstate()` and `inspect.getasyncgenlocals()` for determining the current state of asynchronous generators. (Contributed by Thomas Krennwallner in gh-79940 [https://github.com/python/cpython/issues/79940].)
- The performance of `inspect.getattr_static()` has been considerably improved. Most calls to the function should be at least 2x faster than they were in Python 3.11. (Contributed by Alex Waygood in gh-103193 [https://github.com/python/cpython/issues/103193].)

itertools

- Add `itertools.batched()` for collecting into even-sized tuples where the last batch may be shorter than the rest. (Contributed by Raymond Hettinger in gh-98363 [https://github.com/python/cpython/issues/98363].)

math

- Add `math.sumprod()` for computing a sum of products. (Contributed by Raymond Hettinger in gh-100485 [https://github.com/python/cpython/issues/100485].)
- Extend `math.nextafter()` to include a *steps* argument for moving up or down multiple steps at a time. (Contributed by Matthias Goergens, Mark Dickinson, and Raymond Hettinger in gh-94906 [https://github.com/python/cpython/issues/94906].)

os

- Add `os.PIDFD_NONBLOCK` to open a file descriptor for a process with `os.pidfd_open()` in non-blocking mode. (Contributed by Kumar Aditya in gh-93312 [https://github.com/python/cpython/issues/93312].)
- `os.DirEntry` now includes an `os.DirEntry.is_junction()` method to check if the entry is a junction. (Contributed by Charles Machalow in gh-99547 [https://github.com/python/cpython/issues/99547].)
- Add `os.listdrives()`, `os.listvolumes()` and `os.listmounts()` functions on Windows for enumerating drives, volumes and mount points. (Contributed by Steve Dower in gh-102519 [https://github.com/python/cpython/issues/102519].)
- `os.stat()` and `os.lstat()` are now more accurate on Windows. The `st_birthtime` field will now be filled with the creation time of the file, and `st_ctime` is deprecated but still contains the creation time (but in the future will return the last metadata change, for consistency with other platforms). `st_dev` may be up to 64 bits and `st_ino` up to 128 bits depending on your file system, and `st_rdev` is always set to zero rather than incorrect values. Both functions may be significantly faster on newer releases of Windows. (Contributed by Steve Dower in gh-99726 [https://github.com/python/cpython/issues/99726].)

os.path

- Add `os.path.isjunction()` to check if a given path is a junction. (Contributed by Charles Machalow in gh-99547 [https://github.com/python/cpython/issues/99547].)
- Add `os.path.splitroot()` to split a path into a triad `(drive, root, tail)`. (Contributed by Barney Gale in gh-101000

[https://github.com/python/cpython/issues/101000].)

pathlib

- Add support for subclassing `pathlib.PurePath` and `pathlib.Path`, plus their Posix- and Windows-specific variants. Subclasses may override the `pathlib.PurePath.with_segments()` method to pass information between path instances.
- Add `pathlib.Path.walk()` for walking the directory trees and generating all file or directory names within them, similar to `os.walk()`. (Contributed by Stanislav Zmiev in gh-90385 [https://github.com/python/cpython/issues/90385].)
- Add *walk_up* optional parameter to `pathlib.PurePath.relative_to()` to allow the insertion of `..` entries in the result; this behavior is more consistent with `os.path.relpath()`. (Contributed by Domenico Ragusa in gh-84538 [https://github.com/python/cpython/issues/84538].)
- Add `pathlib.Path.is_junction()` as a proxy to `os.path.isjunction()`. (Contributed by Charles Machalow in gh-99547 [https://github.com/python/cpython/issues/99547].)
- Add *case_sensitive* optional parameter to `pathlib.Path.glob()`, `pathlib.Path.rglob()` and `pathlib.PurePath.match()` for matching the path's case sensitivity, allowing for more precise control over the matching process.

platform

- Add support for detecting Windows 11 and Windows Server releases past 2012. Previously, lookups on Windows Server platforms newer than Windows Server 2012 and on Windows 11 would return `Windows-10`. (Contributed by Steve Dower in gh-89545 [https://github.com/python/cpython/issues/89545].)

pdb

- Add convenience variables to hold values temporarily for debug session and provide quick access to values like the current frame or the return value. (Contributed by Tian Gao in gh-103693 [https://github.com/python/cpython/issues/103693].)

random

- Add **`random.binomialvariate()`**. (Contributed by Raymond Hettinger in gh-81620 [https://github.com/python/cpython/issues/81620].)
- Add a default of `lambd=1.0` to **`random.expovariate()`**. (Contributed by Raymond Hettinger in gh-100234 [https://github.com/python/cpython/issues/100234].)

shutil

- **`shutil.make_archive()`** now passes the *root_dir* argument to custom archivers which support it. In this case it no longer temporarily changes the current working directory of the process to *root_dir* to perform archiving. (Contributed by Serhiy Storchaka in gh-74696 [https://github.com/python/cpython/issues/74696].)

- **`shutil.rmtree()`** now accepts a new argument *onexc* which is an error handler like *onerror* but which expects an exception instance rather than a *(typ, val, tb)* triplet. *onerror* is deprecated. (Contributed by Irit Katriel in gh-102828 [https://github.com/python/cpython/issues/102828].)

- **`shutil.which()`** now consults the *PATHEXT* environment variable to find matches within *PATH* on Windows even when the given *cmd* includes a directory component. (Contributed by Charles Machalow in gh-103179 [https://github.com/python/cpython/issues/103179].)

  `shutil.which()` will call `NeedCurrentDirectoryForExePathW` when querying for executables on Windows to determine if the current working directory should be prepended to the search path. (Contributed by Charles Machalow in gh-103179 [https://github.com/python/cpython/issues/103179].)

  `shutil.which()` will return a path matching the *cmd* with a component from `PATHEXT` prior to a direct match elsewhere in the search path on Windows. (Contributed by Charles Machalow in gh-103179 [https://github.com/python/cpython/issues/103179].)

sqlite3

- Add a command-line interface. (Contributed by Erlend E. Aasland in gh-77617 [https://github.com/python/cpython/issues/77617].)
- Add the `sqlite3.Connection.autocommit` attribute to `sqlite3.Connection` and the *autocommit* parameter to `sqlite3.connect()` to control **PEP 249** [https://peps.python.org/pep-0249/]-compliant transaction handling. (Contributed by Erlend E. Aasland in gh-83638 [https://github.com/python/cpython/issues/83638].)
- Add *entrypoint* keyword-only parameter to `sqlite3.Connection.load_extension()`, for overriding the SQLite extension entry point. (Contributed by Erlend E. Aasland in gh-103015 [https://github.com/python/cpython/issues/103015].)
- Add `sqlite3.Connection.getconfig()` and `sqlite3.Connection.setconfig()` to `sqlite3.Connection` to make configuration changes to a database connection. (Contributed by Erlend E. Aasland in gh-103489 [https://github.com/python/cpython/issues/103489].)

statistics

- Extend `statistics.correlation()` to include as a `ranked` method for computing the Spearman correlation of ranked data. (Contributed by Raymond Hettinger in gh-95861 [https://github.com/python/cpython/issues/95861].)

sys

- Add the `sys.monitoring` namespace to expose the new PEP 669 monitoring API. (Contributed by Mark Shannon in gh-103082 [https://github.com/python/cpython/issues/103082].)
- Add `sys.activate_stack_trampoline()` and `sys.deactivate_stack_trampoline()` for activating and deactivating stack profiler trampolines, and `sys.is_stack_trampoline_active()` for querying if stack profiler trampolines are active. (Contributed by Pablo Galindo and Christian Heimes with contributions from Gregory P. Smith [Google] and Mark Shannon in gh-96123 [https://github.com/python/cpython/issues/96123].)

- Add **`sys.last_exc`** which holds the last unhandled exception that was raised (for post-mortem debugging use cases). Deprecate the three fields that have the same information in its legacy form: **`sys.last_type`**, **`sys.last_value`** and **`sys.last_traceback`**. (Contributed by Irit Katriel in gh-102778 [https://github.com/python/cpython/issues/102778].)
- **`sys._current_exceptions()`** now returns a mapping from thread-id to an exception instance, rather than to a `(typ, exc, tb)` tuple. (Contributed by Irit Katriel in gh-103176 [https://github.com/python/cpython/issues/103176].)
- **`sys.setrecursionlimit()`** and **`sys.getrecursionlimit()`**. The recursion limit now applies only to Python code. Builtin functions do not use the recursion limit, but are protected by a different mechanism that prevents recursion from causing a virtual machine crash.

tempfile

- The **`tempfile.NamedTemporaryFile`** function has a new optional parameter *delete_on_close* (Contributed by Evgeny Zorin in gh-58451 [https://github.com/python/cpython/issues/58451].)
- **`tempfile.mkdtemp()`** now always returns an absolute path, even if the argument provided to the *dir* parameter is a relative path.

threading

- Add **`threading.settrace_all_threads()`** and **`threading.setprofile_all_threads()`** that allow to set tracing and profiling functions in all running threads in addition to the calling one. (Contributed by Pablo Galindo in gh-93503 [https://github.com/python/cpython/issues/93503].)

tkinter

- `tkinter.Canvas.coords()` now flattens its arguments. It now accepts not only coordinates as separate arguments (`x1, y1, x2, y2, ...`) and a sequence of coordinates (`[x1, y1, x2, y2, ...]`), but also coordinates grouped in pairs (`(x1, y1), (x2, y2), ...` and `[(x1, y1), (x2, y2), ...]`), like `create_*()` methods.

(Contributed by Serhiy Storchaka in gh-94473 [https://github.com/python/cpython/issues/94473].)

tokenize

- The `tokenize` module includes the changes introduced in **PEP 701** [https://peps.python.org/pep-0701/]. (Contributed by Marta Gómez Macías and Pablo Galindo in gh-102856 [https://github.com/python/cpython/issues/102856].) See Porting to Python 3.12 for more information on the changes to the `tokenize` module.

types

- Add `types.get_original_bases()` to allow for further introspection of User-defined generic types when subclassed. (Contributed by James Hilton-Balfe and Alex Waygood in gh-101827 [https://github.com/python/cpython/issues/101827].)

typing

- `isinstance()` checks against `runtime-checkable protocols` now use `inspect.getattr_static()` rather than `hasattr()` to lookup whether attributes exist. This means that descriptors and `__getattr__()` methods are no longer unexpectedly evaluated during `isinstance()` checks against runtime-checkable protocols. However, it may also mean that some objects which used to be considered instances of a runtime-checkable protocol may no longer be considered instances of that protocol on Python 3.12+, and vice versa. Most users are unlikely to be affected by this change. (Contributed by Alex Waygood in gh-102433 [https://github.com/python/cpython/issues/102433].)

- The members of a runtime-checkable protocol are now considered “frozen” at runtime as soon as the class has been created. Monkey-patching attributes onto a runtime-checkable protocol will still work, but will have no impact on `isinstance()` checks comparing objects to the protocol. For example:

```
>>> from typing import Protocol, runtime_checkable
>>> @runtime_checkable
... class HasX(Protocol):
...     x = 1
...
>>> class Foo: ...
...
>>> f = Foo()
>>> isinstance(f, HasX)
False
>>> f.x = 1
>>> isinstance(f, HasX)
True
>>> HasX.y = 2
>>> isinstance(f, HasX)  # unchanged, even though HasX now also
has a "y" attribute
True
```

This change was made in order to speed up `isinstance()` checks against runtime-checkable protocols.

- The performance profile of **`isinstance()`** checks against **`runtime-checkable protocols`** has changed significantly. Most `isinstance()` checks against protocols with only a few members should be at least 2x faster than in 3.11, and some may be 20x faster or more. However, `isinstance()` checks against protocols with many members may be slower than in Python 3.11. (Contributed by Alex Waygood in gh-74690 [https://github.com/python/cpython/issues/74690] and gh-103193 [https://github.com/python/cpython/issues/103193].)

- All **`typing.TypedDict`** and **`typing.NamedTuple`** classes now have the `__orig_bases__` attribute. (Contributed by Adrian Garcia Badaracco in gh-103699 [https://github.com/python/cpython/issues/103699].)

- Add `frozen_default` parameter to **`typing.dataclass_transform()`**. (Contributed by Erik De Bonte in gh-99957 [https://github.com/python/cpython/issues/99957].)

unicodedata

- The Unicode database has been updated to version 15.0.0. (Contributed by Benjamin Peterson in gh-96734 [https://github.com/python/cpython/issues/96734]).

unittest

Add a `--durations` command line option, showing the N slowest test cases:

```
python3 -m unittest --durations=3 lib.tests.test_threading
.....
Slowest test durations
---------------------------------------------------------------------
-
1.210s     test_timeout (Lib.test.test_threading.BarrierTests)
1.003s     test_default_timeout
(Lib.test.test_threading.BarrierTests)
0.518s     test_timeout (Lib.test.test_threading.EventTests)

(0.000 durations hidden.  Use -v to show these durations.)
---------------------------------------------------------------------
-
Ran 158 tests in 9.869s

OK (skipped=3)
```

(Contributed by Giampaolo Rodola in gh-48330 [https://github.com/python/cpython/issues/48330])

uuid

- Add a command-line interface. (Contributed by Adam Chhina in gh-88597 [https://github.com/python/cpython/issues/88597].)

Optimizations

- Remove `wstr` and `wstr_length` members from Unicode objects. It reduces object size by 8 or 16 bytes on 64bit platform. (**PEP 623** [https://peps.python.org/pep-0623/]) (Contributed by Inada Naoki in gh-92536 [https://github.com/python/cpython/issues/92536].)
- Add experimental support for using the BOLT binary optimizer in the build process, which improves performance by 1-5%.

(Contributed by Kevin Modzelewski in gh-90536 [https://github.com/python/cpython/issues/90536] and tuned by Donghee Na in gh-101525 [https://github.com/python/cpython/issues/101525])

- Speed up the regular expression substitution (functions `re.sub()` and `re.subn()` and corresponding `re.Pattern` methods) for replacement strings containing group references by 2–3 times. (Contributed by Serhiy Storchaka in gh-91524 [https://github.com/python/cpython/issues/91524].)
- Speed up `asyncio.Task` creation by deferring expensive string formatting. (Contributed by Itamar Oren in gh-103793 [https://github.com/python/cpython/issues/103793].)
- The `tokenize.tokenize()` and `tokenize.generate_tokens()` functions are up to 64% faster as a side effect of the changes required to cover **PEP 701** [https://peps.python.org/pep-0701/] in the `tokenize` module. (Contributed by Marta Gómez Macías and Pablo Galindo in gh-102856 [https://github.com/python/cpython/issues/102856].)
- Speed up `super()` method calls and attribute loads via the new `LOAD_SUPER_ATTR` instruction. (Contributed by Carl Meyer and Vladimir Matveev in gh-103497 [https://github.com/python/cpython/issues/103497].)

CPython bytecode changes

- Remove the `LOAD_METHOD` instruction. It has been merged into `LOAD_ATTR`. `LOAD_ATTR` will now behave like the old `LOAD_METHOD` instruction if the low bit of its oparg is set. (Contributed by Ken Jin in gh-93429 [https://github.com/python/cpython/issues/93429].)
- Remove the `JUMP_IF_FALSE_OR_POP` and `JUMP_IF_TRUE_OR_POP` instructions. (Contributed by Irit Katriel in gh-102859 [https://github.com/python/cpython/issues/102859].)
- Remove the `PRECALL` instruction. (Contributed by Mark Shannon in gh-92925 [https://github.com/python/cpython/issues/92925].)
- Add the `BINARY_SLICE` and `STORE_SLICE` instructions. (Contributed by Mark Shannon in gh-94163 [https://github.com/python/cpython/issues/94163].)
- Add the `CALL_INTRINSIC_1` instructions. (Contributed by Mark Shannon in gh-99005 [https://github.com/python/cpython/issues/99005].)
- Add the `CALL_INTRINSIC_2` instruction. (Contributed by Irit Katriel in gh-101799 [https://github.com/python/cpython/issues/101799].)

- Add the **`CLEANUP_THROW`** instruction. (Contributed by Brandt Bucher in gh-90997 [https://github.com/python/cpython/issues/90997].)
- Add the **`END_SEND`** instruction. (Contributed by Mark Shannon in gh-103082 [https://github.com/python/cpython/issues/103082].)
- Add the **`LOAD_FAST_AND_CLEAR`** instruction as part of the implementation of **PEP 709** [https://peps.python.org/pep-0709/]. (Contributed by Carl Meyer in gh-101441 [https://github.com/python/cpython/issues/101441].)
- Add the **`LOAD_FAST_CHECK`** instruction. (Contributed by Dennis Sweeney in gh-93143 [https://github.com/python/cpython/issues/93143].)
- Add the **`LOAD_FROM_DICT_OR_DEREF`**, **`LOAD_FROM_DICT_OR_GLOBALS`**, and **`LOAD_LOCALS`** opcodes as part of the implementation of **PEP 695** [https://peps.python.org/pep-0695/]. Remove the **`LOAD_CLASSDEREF`** opcode, which can be replaced with **`LOAD_LOCALS`** plus **`LOAD_FROM_DICT_OR_DEREF`**. (Contributed by Jelle Zijlstra in gh-103764 [https://github.com/python/cpython/issues/103764].)
- Add the **`LOAD_SUPER_ATTR`** instruction. (Contributed by Carl Meyer and Vladimir Matveev in gh-103497 [https://github.com/python/cpython/issues/103497].)
- Add the `RETURN_CONST` instruction. (Contributed by Wenyang Wang in gh-101632 [https://github.com/python/cpython/issues/101632].)

Demos and Tools

- Remove the `Tools/demo/` directory which contained old demo scripts. A copy can be found in the old-demos project [https://github.com/gvanrossum/old-demos]. (Contributed by Victor Stinner in gh-97681 [https://github.com/python/cpython/issues/97681].)
- Remove outdated example scripts of the `Tools/scripts/` directory. A copy can be found in the old-demos project [https://github.com/gvanrossum/old-demos]. (Contributed by Victor Stinner in gh-97669 [https://github.com/python/cpython/issues/97669].)

Deprecated

- **`argparse`**: The *type*, *choices*, and *metavar* parameters of **`argparse.BooleanOptionalAction`** are deprecated and will be removed in 3.14. (Contributed by Nikita Sobolev in gh-92248 [https://github.com/python/cpython/issues/92248].)

- **ast**: The following **ast** features have been deprecated in documentation since Python 3.8, now cause a **DeprecationWarning** to be emitted at runtime when they are accessed or used, and will be removed in Python 3.14:
  - `ast.Num`
  - `ast.Str`
  - `ast.Bytes`
  - `ast.NameConstant`
  - `ast.Ellipsis`

  Use **ast.Constant** instead. (Contributed by Serhiy Storchaka in gh-90953 [https://github.com/python/cpython/issues/90953].)

- **asyncio**:
  - The child watcher classes `asyncio.MultiLoopChildWatcher`, `asyncio.FastChildWatcher`, `asyncio.AbstractChildWatcher` and `asyncio.SafeChildWatcher` are deprecated and will be removed in Python 3.14. (Contributed by Kumar Aditya in gh-94597 [https://github.com/python/cpython/issues/94597].)
  - `asyncio.set_child_watcher()`, `asyncio.get_child_watcher()`, `asyncio.AbstractEventLoopPolicy.set_child_watcher()` and `asyncio.AbstractEventLoopPolicy.get_child_watcher()` are deprecated and will be removed in Python 3.14. (Contributed by Kumar Aditya in gh-94597 [https://github.com/python/cpython/issues/94597].)
  - The **get_event_loop()** method of the default event loop policy now emits a **DeprecationWarning** if there is no current event loop set and it decides to create one. (Contributed by Serhiy Storchaka and Guido van Rossum in gh-100160 [https://github.com/python/cpython/issues/100160].)

- **calendar**: `calendar.January` and `calendar.February` constants are deprecated and replaced by **calendar.JANUARY** and **calendar.FEBRUARY**. (Contributed by Prince Roshan in gh-103636 [https://github.com/python/cpython/issues/103636].)

- `collections.abc`: Deprecated `collections.abc.ByteString`.

  Use `isinstance(obj, collections.abc.Buffer)` to test if `obj` implements the buffer protocol at runtime. For use in type annotations, either use `Buffer` or a union that explicitly specifies the types your code supports (e.g., `bytes | bytearray | memoryview`).

  `ByteString` was originally intended to be an abstract class that would serve as a supertype of both `bytes` and `bytearray`. However, since the ABC never had any methods, knowing that an object was an instance of **`ByteString`** never actually told you anything useful about the object. Other common buffer types such as `memoryview` were also never understood as subtypes of **`ByteString`** (either at runtime or by static type checkers).

  See **PEP 688** [https://peps.python.org/pep-0688/#current-options] for more details. (Contributed by Shantanu Jain in gh-91896 [https://github.com/python/cpython/issues/91896].)

- `datetime`: `datetime.datetime`'s `utcnow()` and `utcfromtimestamp()` are deprecated and will be removed in a future version. Instead, use timezone-aware objects to represent datetimes in UTC: respectively, call `now()` and `fromtimestamp()` with the *tz* parameter set to `datetime.UTC`. (Contributed by Paul Ganssle in gh-103857 [https://github.com/python/cpython/issues/103857].)

- `email`: Deprecate the *isdst* parameter in `email.utils.localtime()`. (Contributed by Alan Williams in gh-72346 [https://github.com/python/cpython/issues/72346].)

- `importlib.abc`: Deprecated the following classes, scheduled for removal in Python 3.14:

  - **`importlib.abc.ResourceReader`**
  - **`importlib.abc.Traversable`**
  - **`importlib.abc.TraversableResources`**

  Use `importlib.resources.abc` classes instead:

  - `importlib.resources.abc.Traversable`

  - **`importlib.resources.abc.TraversableResources`**

  (Contributed by Jason R. Coombs and Hugo van Kemenade in gh-93963 [https://github.com/python/cpython/issues/93963].)

- **`itertools`**: Deprecate the support for copy, deepcopy, and pickle operations, which is undocumented, inefficient, historically buggy, and inconsistent. This will be removed in 3.14 for a significant reduction in code volume and maintenance burden. (Contributed by Raymond Hettinger in gh-101588 [https://github.com/python/cpython/issues/101588].)

- **`multiprocessing`**: In Python 3.14, the default **`multiprocessing`** start method will change to a safer one on Linux, BSDs, and other non-macOS POSIX platforms where `'fork'` is currently the default (gh-84559 [https://github.com/python/cpython/issues/84559]). Adding a runtime warning about this was deemed too disruptive as the majority of code is not expected to care. Use the **`get_context()`** or **`set_start_method()`** APIs to explicitly specify when your code *requires* `'fork'`. See contexts and start methods.

- **`pkgutil`**: **`pkgutil.find_loader()`** and **`pkgutil.get_loader()`** are deprecated and will be removed in Python 3.14; use **`importlib.util.find_spec()`** instead. (Contributed by Nikita Sobolev in gh-97850 [https://github.com/python/cpython/issues/97850].)

- **`pty`**: The module has two undocumented `master_open()` and `slave_open()` functions that have been deprecated since Python 2 but only gained a proper **`DeprecationWarning`** in 3.12. Remove them in 3.14. (Contributed by Soumendra Ganguly and Gregory P. Smith in gh-85984 [https://github.com/python/cpython/issues/85984].)

- **`os`**:

  - The `st_ctime` fields return by **`os.stat()`** and **`os.lstat()`** on Windows are deprecated. In a future release, they will contain the last metadata change time, consistent with other platforms. For now, they still contain the creation time, which is also available in the new `st_birthtime` field. (Contributed by Steve Dower in gh-99726 [https://github.com/python/cpython/issues/99726].)

  - On POSIX platforms, **`os.fork()`** can now raise a **`DeprecationWarning`** when it can detect being called from a multithreaded process. There has always been a fundamental incompatibility with the POSIX platform when doing so. Even if such code *appeared* to work. We added the warning to raise awareness as issues encountered by code doing this are becoming more frequent. See the **`os.fork()`** documentation for more details along with this discussion on fork being incompatible with threads [https://discuss.python.org/t/concerns-regarding-deprecation-of-fork-with-alive-threads/33555] for *why* we're now surfacing this longstanding platform compatibility problem to developers.

  When this warning appears due to usage of **`multiprocessing`** or **`concurrent.futures`** the fix is to use a different **`multiprocessing`** start method such as `"spawn"` or `"forkserver"`.

- **`shutil`**: The *onerror* argument of **`shutil.rmtree()`** is deprecated; use *onexc* instead. (Contributed by Irit Katriel in gh-102828 [https://github.com/python/cpython/issues/102828].)

- **`sqlite3`**:

  - default adapters and converters are now deprecated. Instead, use the Adapter and converter recipes and tailor them to your needs. (Contributed by Erlend E. Aasland in gh-90016 [https://github.com/python/cpython/issues/90016].)
  - In **`execute()`**, **`DeprecationWarning`** is now emitted when named placeholders are used together with parameters supplied as a sequence instead of as a **`dict`**. Starting from Python 3.14, using named placeholders with parameters supplied as a sequence will raise a **`ProgrammingError`**. (Contributed by Erlend E. Aasland in gh-101698 [https://github.com/python/cpython/issues/101698].)

- **`sys`**: The **`sys.last_type`**, **`sys.last_value`** and **`sys.last_traceback`** fields are deprecated. Use **`sys.last_exc`** instead. (Contributed by Irit Katriel in gh-102778 [https://github.com/python/cpython/issues/102778].)

- **`tarfile`**: Extracting tar archives without specifying *filter* is deprecated until Python 3.14, when `'data'` filter will become the default. See Extraction filters for details.

- **`typing`**:

  - **`typing.Hashable`** and **`typing.Sized`**, aliases for **`collections.abc.Hashable`** and **`collections.abc.Sized`** respectively, are deprecated. (gh-94309 [https://github.com/python/cpython/issues/94309].)
  - **`typing.ByteString`**, deprecated since Python 3.9, now causes a **`DeprecationWarning`** to be emitted when it is used. (Contributed by Alex Waygood in gh-91896 [https://github.com/python/cpython/issues/91896].)

- **`xml.etree.ElementTree`**: The module now emits **`DeprecationWarning`** when testing the truth value of an **`xml.etree.ElementTree.Element`**. Before, the Python implementation emitted **`FutureWarning`**, and the C implementation emitted nothing. (Contributed by Jacob Walls in gh-83122 [https://github.com/python/cpython/issues/83122].)

- The 3-arg signatures (type, value, traceback) of **`coroutine throw()`**, **`generator throw()`** and **`async generator throw()`** are deprecated and may be removed in a future version of Python. Use the single-arg versions of these functions instead. (Contributed by Ofey Chan in gh-89874 [https://github.com/python/cpython/issues/89874].)

- **`DeprecationWarning`** is now raised when **`__package__`** on a module differs from **`__spec__.parent`** (previously it was **`ImportWarning`**). (Contributed by Brett Cannon in gh-65961 [https://github.com/python/cpython/issues/65961].)

- Setting `__package__` or `__cached__` on a module is deprecated, and will cease to be set or taken into consideration by the import system in Python 3.14. (Contributed by Brett Cannon in gh-65961 [https://github.com/python/cpython/issues/65961].)

- The bitwise inversion operator (~) on bool is deprecated. It will throw an error in Python 3.16. Use `not` for logical negation of bools instead. In the rare case that you really need the bitwise inversion of the underlying `int`, convert to int explicitly: `~int(x)`. (Contributed by Tim Hoffmann in gh-103487 [https://github.com/python/cpython/issues/103487].)

- Accessing `co_lnotab` on code objects was deprecated in Python 3.10 via **PEP 626** [https://peps.python.org/pep-0626/], but it only got a proper `DeprecationWarning` in 3.12. May be removed in 3.15. (Contributed by Nikita Sobolev in gh-101866 [https://github.com/python/cpython/issues/101866].)

Pending removal in Python 3.13

Modules (see **PEP 594** [https://peps.python.org/pep-0594/]):

- **`aifc`**
- **`audioop`**
- **`cgi`**
- **`cgitb`**
- **`chunk`**
- **`crypt`**
- **`imghdr`**
- **`mailcap`**
- **`msilib`**
- **`nis`**
- **`nntplib`**
- **`ossaudiodev`**
- **`pipes`**
- **`sndhdr`**
- **`spwd`**
- **`sunau`**
- **`telnetlib`**
- **`uu`**

- **`xdrlib`**

Other modules:

- **`lib2to3`**, and the **2to3** program (gh-84540 [https://github.com/python/cpython/issues/84540])

APIs:

- **`configparser.LegacyInterpolation`** (gh-90765 [https://github.com/python/cpython/issues/90765])
- `locale.resetlocale()` (gh-90817 [https://github.com/python/cpython/issues/90817])
- **`turtle.RawTurtle.settiltangle()`** (gh-50096 [https://github.com/python/cpython/issues/50096])
- **`unittest.findTestCases()`** (gh-50096 [https://github.com/python/cpython/issues/50096])
- **`unittest.getTestCaseNames()`** (gh-50096 [https://github.com/python/cpython/issues/50096])
- **`unittest.makeSuite()`** (gh-50096 [https://github.com/python/cpython/issues/50096])
- **`unittest.TestProgram.usageExit()`** (gh-67048 [https://github.com/python/cpython/issues/67048])
- **`webbrowser.MacOSX`** (gh-86421 [https://github.com/python/cpython/issues/86421])
- **`classmethod`** descriptor chaining (gh-89519 [https://github.com/python/cpython/issues/89519])

Pending removal in Python 3.14

- **`argparse`**: The *type*, *choices*, and *metavar* parameters of **`argparse.BooleanOptionalAction`** are deprecated and will be removed in 3.14. (Contributed by Nikita Sobolev in gh-92248 [https://github.com/python/cpython/issues/92248].)

- **`ast`**: The following features have been deprecated in documentation since Python 3.8, now cause a **`DeprecationWarning`** to be emitted at runtime when they are accessed or used, and will be removed in Python 3.14:

  - **`ast.Num`**
  - **`ast.Str`**
  - **`ast.Bytes`**
  - **`ast.NameConstant`**
  - **`ast.Ellipsis`**

  Use **`ast.Constant`** instead. (Contributed by Serhiy Storchaka in gh-90953 [https://github.com/python/cpython/issues/90953].)

- **`asyncio`**:

  - The child watcher classes **`asyncio.MultiLoopChildWatcher`**, **`asyncio.FastChildWatcher`**, **`asyncio.AbstractChildWatcher`** and **`asyncio.SafeChildWatcher`** are deprecated and will be removed in Python 3.14. (Contributed by Kumar Aditya in gh-94597 [https://github.com/python/cpython/issues/94597].)
  - **`asyncio.set_child_watcher()`**, **`asyncio.get_child_watcher()`**, **`asyncio.AbstractEventLoopPolicy.set_child_watcher()`** and **`asyncio.AbstractEventLoopPolicy.get_child_watcher()`** are deprecated and will be removed in Python 3.14. (Contributed by Kumar Aditya in gh-94597 [https://github.com/python/cpython/issues/94597].)
  - The **`get_event_loop()`** method of the default event loop policy now emits a **`DeprecationWarning`** if there is no current event loop set and it decides to create one. (Contributed by Serhiy Storchaka and Guido van Rossum in gh-100160 [https://github.com/python/cpython/issues/100160].)

- **`email`**: Deprecated the *isdst* parameter in **`email.utils.localtime()`**. (Contributed by Alan Williams in gh-72346 [https://github.com/python/cpython/issues/72346].)

- **`importlib.abc`** deprecated classes:

  - **`importlib.abc.ResourceReader`**
  - **`importlib.abc.Traversable`**
  - **`importlib.abc.TraversableResources`**

  Use **`importlib.resources.abc`** classes instead:

    - `importlib.resources.abc.Traversable`
    - `importlib.resources.abc.TraversableResources`

  (Contributed by Jason R. Coombs and Hugo van Kemenade in gh-93963 [https://github.com/python/cpython/issues/93963].)

- `itertools` had undocumented, inefficient, historically buggy, and inconsistent support for copy, deepcopy, and pickle operations. This will be removed in 3.14 for a significant reduction in code volume and maintenance burden. (Contributed by Raymond Hettinger in gh-101588 [https://github.com/python/cpython/issues/101588].)

- `multiprocessing`: The default start method will change to a safer one on Linux, BSDs, and other non-macOS POSIX platforms where `'fork'` is currently the default (gh-84559 [https://github.com/python/cpython/issues/84559]). Adding a runtime warning about this was deemed too disruptive as the majority of code is not expected to care. Use the `get_context()` or `set_start_method()` APIs to explicitly specify when your code *requires* `'fork'`. See Contexts and start methods.

- `pathlib`: `is_relative_to()` and `relative_to()`: passing additional arguments is deprecated.

- `pkgutil`: **`pkgutil.find_loader()`** and **`pkgutil.get_loader()`** now raise `DeprecationWarning`; use `importlib.util.find_spec()` instead. (Contributed by Nikita Sobolev in gh-97850 [https://github.com/python/cpython/issues/97850].)

- `pty`:

    - `master_open()`: use `pty.openpty()`.
    - `slave_open()`: use `pty.openpty()`.

- `sqlite3`:

    - **`version`** and **`version_info`**.
    - `execute()` and `executemany()` if named placeholders are used and *parameters* is a sequence instead of a `dict`.

- **urllib**: **`urllib.parse.Quoter`** is deprecated: it was not intended to be a public API. (Contributed by Gregory P. Smith in gh-88168 [https://github.com/python/cpython/issues/88168].)

Pending removal in Python 3.15

- The import system:
  - Setting **`__cached__`** on a module while failing to set **`__spec__.cached`** is deprecated. In Python 3.15, **`__cached__`** will cease to be set or take into consideration by the import system or standard library. (gh-97879 [https://github.com/python/cpython/issues/97879])
  - Setting **`__package__`** on a module while failing to set **`__spec__.parent`** is deprecated. In Python 3.15, **`__package__`** will cease to be set or take into consideration by the import system or standard library. (gh-97879 [https://github.com/python/cpython/issues/97879])
- **ctypes**:
  - The undocumented **`ctypes.SetPointerType()`** function has been deprecated since Python 3.13.
- **http.server**:
  - The obsolete and rarely used **`CGIHTTPRequestHandler`** has been deprecated since Python 3.13. No direct replacement exists. *Anything* is better than CGI to interface a web server with a request handler.
  - The **`--cgi`** flag to the **python -m http.server** command-line interface has been deprecated since Python 3.13.
- **importlib**:
  - `load_module()` method: use `exec_module()` instead.
- **locale**:
  - The **`getdefaultlocale()`** function has been deprecated since Python 3.11. Its removal was originally planned for Python 3.13 (gh-90817 [https://github.com/python/cpython/issues/90817]), but has been postponed to Python 3.15. Use **`getlocale()`**, **`setlocale()`**, and **`getencoding()`** instead. (Contributed by Hugo van Kemenade in gh-111187 [https://github.com/python/cpython/issues/111187].)
- **pathlib**:

    - `PurePath.is_reserved()` has been deprecated since Python 3.13. Use `os.path.isreserved()` to detect reserved paths on Windows.
- `platform`:
    - `java_ver()` has been deprecated since Python 3.13. This function is only useful for Jython support, has a confusing API, and is largely untested.
- `sysconfig`:
    - The *check_home* argument of `sysconfig.is_python_build()` has been deprecated since Python 3.12.
- `threading`:
    - `RLock()` will take no arguments in Python 3.15. Passing any arguments has been deprecated since Python 3.14, as the Python version does not permit any arguments, but the C version allows any number of positional or keyword arguments, ignoring every argument.
- `types`:
    - `types.CodeType`: Accessing `co_lnotab` was deprecated in **PEP 626** [https://peps.python.org/pep-0626/] since 3.10 and was planned to be removed in 3.12, but it only got a proper `DeprecationWarning` in 3.12. May be removed in 3.15. (Contributed by Nikita Sobolev in gh-101866 [https://github.com/python/cpython/issues/101866].)
- `typing`:
    - The undocumented keyword argument syntax for creating `NamedTuple` classes (for example, `Point = NamedTuple("Point", x=int, y=int)`) has been deprecated since Python 3.13. Use the class-based syntax or the functional syntax instead.
    - When using the functional syntax of `TypedDict`s, failing to pass a value to the *fields* parameter (`TD = TypedDict("TD")`) or passing `None` (`TD = TypedDict("TD", None)`) has been deprecated since Python 3.13. Use `class TD(TypedDict): pass` or `TD = TypedDict("TD", {})` to create a TypedDict with zero field.
    - The `typing.no_type_check_decorator()` decorator function has been deprecated since Python 3.13. After eight years in the `typing` module, it has yet to be supported by any major type checker.
- `wave`:

  - The `getmark()`, `setmark()`, and `getmarkers()` methods of the `Wave_read` and `Wave_write` classes have been deprecated since Python 3.13.
- `zipimport`:
  - `load_module()` has been deprecated since Python 3.10. Use `exec_module()` instead. (Contributed by Jiahao Li in gh-125746 [https://github.com/python/cpython/issues/125746].)

Pending removal in Python 3.16

- The import system:
  - Setting `__loader__` on a module while failing to set `__spec__.loader` is deprecated. In Python 3.16, `__loader__` will cease to be set or taken into consideration by the import system or the standard library.
- `array`:
  - The `'u'` format code (`wchar_t`) has been deprecated in documentation since Python 3.3 and at runtime since Python 3.13. Use the `'w'` format code (`Py_UCS4`) for Unicode characters instead.
- `asyncio`:
  - `asyncio.iscoroutinefunction()` is deprecated and will be removed in Python 3.16; use `inspect.iscoroutinefunction()` instead. (Contributed by Jiahao Li and Kumar Aditya in gh-122875 [https://github.com/python/cpython/issues/122875].)
  - `asyncio` policy system is deprecated and will be removed in Python 3.16. In particular, the following classes and functions are deprecated:
    - `asyncio.AbstractEventLoopPolicy`
    - `asyncio.DefaultEventLoopPolicy`
    - `asyncio.WindowsSelectorEventLoopPolicy`
    - `asyncio.WindowsProactorEventLoopPolicy`
    - `asyncio.get_event_loop_policy()`

    - **[`asyncio.set_event_loop_policy()`]**

  Users should use **`asyncio.run()`** or **`asyncio.Runner`** with *loop_factory* to use the desired event loop implementation.

  For example, to use **`asyncio.SelectorEventLoop`** on Windows:

  ```
  import asyncio

  async def main():
      ...

  asyncio.run(main(), loop_factory=asyncio.SelectorEventLoop)
  ```

  (Contributed by Kumar Aditya in gh-127949 [https://github.com/python/cpython/issues/127949].)

- **`builtins`**:
  - Bitwise inversion on boolean types, `~True` or `~False` has been deprecated since Python 3.12, as it produces surprising and unintuitive results (`-2` and `-1`). Use `not x` instead for the logical negation of a Boolean. In the rare case that you need the bitwise inversion of the underlying integer, convert to `int` explicitly (`~int(x)`).
- **`functools`**:
  - Calling the Python implementation of **`functools.reduce()`** with *function* or *sequence* as keyword arguments has been deprecated since Python 3.14.
- **`logging`**:

  Support for custom logging handlers with the *strm* argument is deprecated and scheduled for removal in Python 3.16. Define handlers with the *stream* argument instead. (Contributed by Mariusz Felisiak in gh-115032 [https://github.com/python/cpython/issues/115032].)
- **`mimetypes`**:

  - Valid extensions start with a ‘.’ or are empty for `mimetypes.MimeTypes.add_type()`. Undotted extensions are deprecated and will raise a `ValueError` in Python 3.16. (Contributed by Hugo van Kemenade in gh-75223 [https://github.com/python/cpython/issues/75223].)

- `shutil`:

  - The `ExecError` exception has been deprecated since Python 3.14. It has not been used by any function in `shutil` since Python 3.4, and is now an alias of `RuntimeError`.

- `symtable`:

  - The `Class.get_methods` method has been deprecated since Python 3.14.

- `sys`:

  - The `_enablelegacywindowsfsencoding()` function has been deprecated since Python 3.13. Use the `PYTHONLEGACYWINDOWSFSENCODING` environment variable instead.

- `sysconfig`:

  - The `sysconfig.expand_makefile_vars()` function has been deprecated since Python 3.14. Use the `vars` argument of `sysconfig.get_paths()` instead.

- `tarfile`:

  - The undocumented and unused `TarFile.tarfile` attribute has been deprecated since Python 3.13.

Pending removal in Python 3.17

- `collections.abc`:

  - `collections.abc.ByteString` is scheduled for removal in Python 3.17.

    Use `isinstance(obj, collections.abc.Buffer)` to test if `obj` implements the buffer protocol at runtime. For use in type annotations, either use `Buffer` or a union that explicitly specifies the types your code supports (e.g., `bytes | bytearray | memoryview`).

    `ByteString` was originally intended to be an abstract class that would serve as a supertype of both `bytes` and `bytearray`. However, since the ABC never had any methods, knowing that an object was an instance of **`ByteString`** never actually told you anything useful about the object. Other common buffer types such as `memoryview` were also never understood as subtypes of **`ByteString`** (either at runtime or by static type checkers).

    See **PEP 688** [https://peps.python.org/pep-0688/#current-options] for more details. (Contributed by Shantanu Jain in gh-91896 [https://github.com/python/cpython/issues/91896].)

- `typing`:

  - Before Python 3.14, old-style unions were implemented using the private class `typing._UnionGenericAlias`. This class is no longer needed for the implementation, but it has been retained for backward compatibility, with removal scheduled for Python 3.17. Users should use documented introspection helpers like `typing.get_origin()` and `typing.get_args()` instead of relying on private implementation details.

  - `typing.ByteString`, deprecated since Python 3.9, is scheduled for removal in Python 3.17.

    Use `isinstance(obj, collections.abc.Buffer)` to test if `obj` implements the buffer protocol at runtime. For use in type annotations, either use `Buffer` or a union that explicitly specifies the types your code supports (e.g., `bytes | bytearray | memoryview`).

`ByteString` was originally intended to be an abstract class that would serve as a supertype of both **`bytes`** and **`bytearray`**. However, since the ABC never had any methods, knowing that an object was an instance of **`ByteString`** never actually told you anything useful about the object. Other common buffer types such as **`memoryview`** were also never understood as subtypes of **`ByteString`** (either at runtime or by static type checkers).

See **PEP 688** [https://peps.python.org/pep-0688/#current-options] for more details. (Contributed by Shantanu Jain in gh-91896 [https://github.com/python/cpython/issues/91896].)

Pending removal in future versions

The following APIs will be removed in the future, although there is currently no date scheduled for their removal.

- **`argparse`**:
  - Nesting argument groups and nesting mutually exclusive groups are deprecated.
  - Passing the undocumented keyword argument *prefix_chars* to **`add_argument_group()`** is now deprecated.
  - The **`argparse.FileType`** type converter is deprecated.
- **`builtins`**:
  - Generators: `throw(type, exc, tb)` and `athrow(type, exc, tb)` signature is deprecated: use `throw(exc)` and `athrow(exc)` instead, the single argument signature.
  - Currently Python accepts numeric literals immediately followed by keywords, for example `0in x`, `1or x`, `0if 1else 2`. It allows confusing and ambiguous expressions like `[0x1for x in y]` (which can be interpreted as `[0x1 for x in y]` or `[0x1f or x in y]`). A syntax warning is raised if the numeric literal is immediately followed by one of keywords **`and`**, **`else`**, **`for`**, **`if`**, **`in`**, **`is`** and **`or`**. In a future release it will be changed to a syntax error. (gh-87999 [https://github.com/python/cpython/issues/87999])
  - Support for `__index__()` and `__int__()` method returning non-int type: these methods will be required to return an

    instance of a strict subclass of **`int`**.
    - Support for `__float__()` method returning a strict subclass of **`float`**: these methods will be required to return an instance of **`float`**.
    - Support for `__complex__()` method returning a strict subclass of **`complex`**: these methods will be required to return an instance of **`complex`**.
    - Delegation of `int()` to `__trunc__()` method.
    - Passing a complex number as the *real* or *imag* argument in the **`complex()`** constructor is now deprecated; it should only be passed as a single positional argument. (Contributed by Serhiy Storchaka in gh-109218 [https://github.com/python/cpython/issues/109218].)
- **`calendar`**: `calendar.January` and `calendar.February` constants are deprecated and replaced by **`calendar.JANUARY`** and **`calendar.FEBRUARY`**. (Contributed by Prince Roshan in gh-103636 [https://github.com/python/cpython/issues/103636].)
- **`codecs`**: use **`open()`** instead of **`codecs.open()`**. (gh-133038 [https://github.com/python/cpython/issues/133038])
- **`codeobject.co_lnotab`**: use the **`codeobject.co_lines()`** method instead.
- **`datetime`**:
    - **`utcnow()`**: use `datetime.datetime.now(tz=datetime.UTC)`.
    - **`utcfromtimestamp()`**: use `datetime.datetime.fromtimestamp(timestamp, tz=datetime.UTC)`.
- **`gettext`**: Plural value must be an integer.
- **`importlib`**:
    - **`cache_from_source()`** *debug_override* parameter is deprecated: use the *optimization* parameter instead.
- **`importlib.metadata`**:
    - `EntryPoints` tuple interface.
    - Implicit `None` on return values.
- **`logging`**: the `warn()` method has been deprecated since Python 3.3, use **`warning()`** instead.
- **`mailbox`**: Use of StringIO input and text mode is deprecated, use BytesIO and binary mode instead.
- **`os`**: Calling **`os.register_at_fork()`** in multi-threaded process.
- **`pydoc.ErrorDuringImport`**: A tuple value for *exc_info* parameter is deprecated, use an exception instance.

- `re`: More strict rules are now applied for numerical group references and group names in regular expressions. Only sequence of ASCII digits is now accepted as a numerical reference. The group name in bytes patterns and replacement strings can now only contain ASCII letters and digits and underscore. (Contributed by Serhiy Storchaka in gh-91760 [https://github.com/python/cpython/issues/91760].)
- **`sre_compile`**, **`sre_constants`** and **`sre_parse`** modules.
- `shutil`: `rmtree()`'s *onerror* parameter is deprecated in Python 3.12; use the *onexc* parameter instead.
- `ssl` options and protocols:
  - `ssl.SSLContext` without protocol argument is deprecated.
  - `ssl.SSLContext`: `set_npn_protocols()` and **`selected_npn_protocol()`** are deprecated: use ALPN instead.
  - `ssl.OP_NO_SSL*` options
  - `ssl.OP_NO_TLS*` options
  - `ssl.PROTOCOL_SSLv3`
  - `ssl.PROTOCOL_TLS`
  - `ssl.PROTOCOL_TLSv1`
  - `ssl.PROTOCOL_TLSv1_1`
  - `ssl.PROTOCOL_TLSv1_2`
  - `ssl.TLSVersion.SSLv3`
  - `ssl.TLSVersion.TLSv1`
  - `ssl.TLSVersion.TLSv1_1`
- `threading` methods:
  - **`threading.Condition.notifyAll()`**: use `notify_all()`.
  - **`threading.Event.isSet()`**: use `is_set()`.
  - **`threading.Thread.isDaemon()`**, `threading.Thread.setDaemon()`: use `threading.Thread.daemon` attribute.
  - **`threading.Thread.getName()`**, `threading.Thread.setName()`: use `threading.Thread.name` attribute.
  - **`threading.currentThread()`**: use `threading.current_thread()`.
  - **`threading.activeCount()`**: use `threading.active_count()`.
- `typing.Text` (gh-92332 [https://github.com/python/cpython/issues/92332]).
- The internal class `typing._UnionGenericAlias` is no longer used to implement `typing.Union`. To preserve compatibility with users using this private class, a compatibility shim will be provided until at least Python 3.17. (Contributed by Jelle Zijlstra in gh-105499 [https://github.com/python/cpython/issues/105499].)

- **`unittest.IsolatedAsyncioTestCase`**: it is deprecated to return a value that is not `None` from a test case.
- **`urllib.parse`** deprecated functions: **`urlparse()`** instead
  - `splitattr()`
  - `splithost()`
  - `splitnport()`
  - `splitpasswd()`
  - `splitport()`
  - `splitquery()`
  - `splittag()`
  - `splittype()`
  - `splituser()`
  - `splitvalue()`
  - `to_bytes()`
- **`wsgiref`**: `SimpleHandler.stdout.write()` should not do partial writes.
- **`xml.etree.ElementTree`**: Testing the truth value of an **`Element`** is deprecated. In a future release it will always return `True`. Prefer explicit `len(elem)` or `elem is not None` tests instead.
- **`sys._clear_type_cache()`** is deprecated: use **`sys._clear_internal_caches()`** instead.

Removed

asynchat and asyncore

- These two modules have been removed according to the schedule in **PEP 594** [https://peps.python.org/pep-0594/], having been deprecated in Python 3.6. Use **`asyncio`** instead. (Contributed by Nikita Sobolev in gh-96580 [https://github.com/python/cpython/issues/96580].)

configparser

- Several names deprecated in the **`configparser`** way back in 3.2 have been removed per gh-89336 [https://github.com/python/cpython/issues/89336]:
  - **`configparser.ParsingError`** no longer has a `filename` attribute or argument. Use the `source` attribute and argument

instead.
  - ○ [`configparser`] no longer has a `SafeConfigParser` class. Use the shorter `ConfigParser` name instead.
  - ○ `configparser.ConfigParser` no longer has a `readfp` method. Use `read_file()` instead.

distutils

- Remove the **`distutils`** package. It was deprecated in Python 3.10 by **PEP 632** [https://peps.python.org/pep-0632/] "Deprecate distutils module". For projects still using `distutils` and cannot be updated to something else, the `setuptools` project can be installed: it still provides `distutils`. (Contributed by Victor Stinner in gh-92584 [https://github.com/python/cpython/issues/92584].)

ensurepip

- Remove the bundled setuptools wheel from `ensurepip`, and stop installing setuptools in environments created by `venv`.

  `pip (>= 22.1)` does not require setuptools to be installed in the environment. `setuptools`-based (and `distutils`-based) packages can still be used with `pip install`, since pip will provide `setuptools` in the build environment it uses for building a package.

  `easy_install`, `pkg_resources`, `setuptools` and `distutils` are no longer provided by default in environments created with `venv` or bootstrapped with `ensurepip`, since they are part of the `setuptools` package. For projects relying on these at runtime, the `setuptools` project should be declared as a dependency and installed separately (typically, using pip).

  (Contributed by Pradyun Gedam in gh-95299 [https://github.com/python/cpython/issues/95299].)

enum

- Remove `enum`'s `EnumMeta.__getattr__`, which is no longer needed for enum attribute access. (Contributed by Ethan Furman in gh-95083 [https://github.com/python/cpython/issues/95083].)

ftplib

- Remove **`ftplib`**’s `FTP_TLS.ssl_version` class attribute: use the *context* parameter instead. (Contributed by Victor Stinner in gh-94172 [https://github.com/python/cpython/issues/94172].)

gzip

- Remove the `filename` attribute of **`gzip`**’s **`gzip.GzipFile`**, deprecated since Python 2.6, use the **`name`** attribute instead. In write mode, the `filename` attribute added `'.gz'` file extension if it was not present. (Contributed by Victor Stinner in gh-94196 [https://github.com/python/cpython/issues/94196].)

hashlib

- Remove the pure Python implementation of **`hashlib`**’s **`hashlib.pbkdf2_hmac()`**, deprecated in Python 3.10. Python 3.10 and newer requires OpenSSL 1.1.1 (**PEP 644** [https://peps.python.org/pep-0644/]): this OpenSSL version provides a C implementation of **`pbkdf2_hmac()`** which is faster. (Contributed by Victor Stinner in gh-94199 [https://github.com/python/cpython/issues/94199].)

importlib

- Many previously deprecated cleanups in **`importlib`** have now been completed:
  - References to, and support for **`module_repr()`** has been removed. (Contributed by Barry Warsaw in gh-97850 [https://github.com/python/cpython/issues/97850].)
  - `importlib.util.set_package`, `importlib.util.set_loader` and `importlib.util.module_for_loader` have all been removed. (Contributed by Brett Cannon and Nikita Sobolev in gh-65961 [https://github.com/python/cpython/issues/65961] and gh-97850 [https://github.com/python/cpython/issues/97850].)

  - Support for `find_loader()` and `find_module()` APIs have been removed. (Contributed by Barry Warsaw in gh-98040 [https://github.com/python/cpython/issues/98040].)
  - `importlib.abc.Finder`, `pkgutil.ImpImporter`, and `pkgutil.ImpLoader` have been removed. (Contributed by Barry Warsaw in gh-98040 [https://github.com/python/cpython/issues/98040].)

imp

- The **`imp`** module has been removed. (Contributed by Barry Warsaw in gh-98040 [https://github.com/python/cpython/issues/98040].)

  To migrate, consult the following correspondence table:

imp	importlib
`imp.NullImporter`	Insert `None` into `sys.path_importer_cache`
`imp.cache_from_source()`	**`importlib.util.cache_from_source()`**
`imp.find_module()`	**`importlib.util.find_spec()`**
`imp.get_magic()`	**`importlib.util.MAGIC_NUMBER`**
`imp.get_suffixes()`	**`importlib.machinery.SOURCE_SUFFIXES`**, **`importlib.machinery.EXTENSION_SUFFIXES`**, and **`importlib.machinery.BYTECODE_SUFFIXES`**
`imp.get_tag()`	**`sys.implementation.cache_tag`**

imp	importlib
`imp.load_module()`	`importlib.import_module()`
`imp.new_module(name)`	`types.ModuleType(name)`
`imp.reload()`	`importlib.reload()`
`imp.source_from_cache()`	`importlib.util.source_from_cache()`
`imp.load_source()`	*See below*

Replace `imp.load_source()` with:

```
import importlib.util
import importlib.machinery

def load_source(modname, filename):
    loader = importlib.machinery.SourceFileLoader(modname,
filename)
    spec = importlib.util.spec_from_file_location(modname,
filename, loader=loader)
    module = importlib.util.module_from_spec(spec)
    # The module is always executed and not cached in
sys.modules.
    # Uncomment the following line to cache the module.
    # sys.modules[module.__name__] = module
    loader.exec_module(module)
    return module
```

- Remove **`imp`** functions and attributes with no replacements:
  - Undocumented functions:
    - `imp.init_builtin()`
    - `imp.load_compiled()`
    - `imp.load_dynamic()`
    - `imp.load_package()`

  - `imp.lock_held()`, `imp.acquire_lock()`, `imp.release_lock()`: the locking scheme has changed in Python 3.3 to per-module locks.
  - `imp.find_module()` constants: `SEARCH_ERROR`, `PY_SOURCE`, `PY_COMPILED`, `C_EXTENSION`, `PY_RESOURCE`, `PKG_DIRECTORY`, `C_BUILTIN`, `PY_FROZEN`, `PY_CODERESOURCE`, `IMP_HOOK`.

io

- Remove **`io`**'s `io.OpenWrapper` and `_pyio.OpenWrapper`, deprecated in Python 3.10: just use **`open()`** instead. The **`open()`** (**`io.open()`**) function is a built-in function. Since Python 3.10, **`_pyio.open()`** is also a static method. (Contributed by Victor Stinner in gh-94169 [https://github.com/python/cpython/issues/94169].)

locale

- Remove **`locale`**'s **`locale.format()`** function, deprecated in Python 3.7: use **`locale.format_string()`** instead. (Contributed by Victor Stinner in gh-94226 [https://github.com/python/cpython/issues/94226].)

smtpd

- The `smtpd` module has been removed according to the schedule in **PEP 594** [https://peps.python.org/pep-0594/], having been deprecated in Python 3.4.7 and 3.5.4. Use the aiosmtpd [https://pypi.org/project/aiosmtpd/] PyPI module or any other **`asyncio`**-based server instead. (Contributed by Oleg Iarygin in gh-93243 [https://github.com/python/cpython/issues/93243].)

sqlite3

- The following undocumented **`sqlite3`** features, deprecated in Python 3.10, are now removed:
  - `sqlite3.enable_shared_cache()`
  - `sqlite3.OptimizedUnicode`

If a shared cache must be used, open the database in URI mode using the `cache=shared` query parameter.

The `sqlite3.OptimizedUnicode` text factory has been an alias for **`str`** since Python 3.3. Code that previously set the text factory to `OptimizedUnicode` can either use `str` explicitly, or rely on the default value which is also `str`.

(Contributed by Erlend E. Aasland in gh-92548 [https://github.com/python/cpython/issues/92548].)

ssl

- Remove **`ssl`**'s **`ssl.RAND_pseudo_bytes()`** function, deprecated in Python 3.6: use **`os.urandom()`** or **`ssl.RAND_bytes()`** instead. (Contributed by Victor Stinner in gh-94199 [https://github.com/python/cpython/issues/94199].)
- Remove the **`ssl.match_hostname()`** function. It was deprecated in Python 3.7. OpenSSL performs hostname matching since Python 3.7, Python no longer uses the **`ssl.match_hostname()`** function. (Contributed by Victor Stinner in gh-94199 [https://github.com/python/cpython/issues/94199].)
- Remove the **`ssl.wrap_socket()`** function, deprecated in Python 3.7: instead, create a **`ssl.SSLContext`** object and call its **`ssl.SSLContext.wrap_socket`** method. Any package that still uses **`ssl.wrap_socket()`** is broken and insecure. The function neither sends a SNI TLS extension nor validates the server hostname. Code is subject to **CWE 295** [https://cwe.mitre.org/data/definitions/295.html] (Improper Certificate Validation). (Contributed by Victor Stinner in gh-94199 [https://github.com/python/cpython/issues/94199].)

unittest

- Remove many long-deprecated **`unittest`** features:
  - A number of **`TestCase`** method aliases:

Deprecated alias	Method Name	Deprecated in
`failUnless`	`assertTrue()`	3.1
`failIf`	`assertFalse()`	3.1
`failUnlessEqual`	`assertEqual()`	3.1
`failIfEqual`	`assertNotEqual()`	3.1
`failUnlessAlmostEqual`	`assertAlmostEqual()`	3.1
`failIfAlmostEqual`	`assertNotAlmostEqual()`	3.1
`failUnlessRaises`	`assertRaises()`	3.1
`assert_`	`assertTrue()`	3.2
`assertEquals`	`assertEqual()`	3.2
`assertNotEquals`	`assertNotEqual()`	3.2
`assertAlmostEquals`	`assertAlmostEqual()`	3.2
`assertNotAlmostEquals`	`assertNotAlmostEqual()`	3.2
`assertRegexpMatches`	`assertRegex()`	3.2

Deprecated alias	Method Name	Deprecated in
`assertRaisesRegexp`	**`assertRaisesRegex()`**	3.2
`assertNotRegexpMatches`	**`assertNotRegex()`**	3.5

You can use https://github.com/isidentical/teyit to automatically modernise your unit tests.

- Undocumented and broken **`TestCase`** method `assertDictContainsSubset` (deprecated in Python 3.2).
- Undocumented **`TestLoader.loadTestsFromModule`** parameter *use_load_tests* (deprecated and ignored since Python 3.5).
- An alias of the **`TextTestResult`** class: `_TextTestResult` (deprecated in Python 3.2).

(Contributed by Serhiy Storchaka in gh-89325 [https://github.com/python/cpython/issues/89325].)

webbrowser

- Remove support for obsolete browsers from **`webbrowser`**. The removed browsers include: Grail, Mosaic, Netscape, Galeon, Skipstone, Iceape, Firebird, and Firefox versions 35 and below (gh-102871 [https://github.com/python/cpython/issues/102871]).

xml.etree.ElementTree

- Remove the `ElementTree.Element.copy()` method of the pure Python implementation, deprecated in Python 3.10, use the **`copy.copy()`** function instead. The C implementation of **`xml.etree.ElementTree`** has no `copy()` method, only a `__copy__()`

method. (Contributed by Victor Stinner in gh-94383 [https://github.com/python/cpython/issues/94383].)

zipimport

- Remove **`zipimport`**’s `find_loader()` and `find_module()` methods, deprecated in Python 3.10: use the `find_spec()` method instead. See **PEP 451** [https://peps.python.org/pep-0451/] for the rationale. (Contributed by Victor Stinner in gh-94379 [https://github.com/python/cpython/issues/94379].)

Others

- Remove the `suspicious` rule from the documentation `Makefile` and `Doc/tools/rstlint.py`, both in favor of sphinx-lint [https://github.com/sphinx-contrib/sphinx-lint]. (Contributed by Julien Palard in gh-98179 [https://github.com/python/cpython/issues/98179].)
- Remove the *keyfile* and *certfile* parameters from the **`ftplib`**, **`imaplib`**, **`poplib`** and **`smtplib`** modules, and the *key_file*, *cert_file* and *check_hostname* parameters from the **`http.client`** module, all deprecated since Python 3.6. Use the *context* parameter (*ssl_context* in **`imaplib`**) instead. (Contributed by Victor Stinner in gh-94172 [https://github.com/python/cpython/issues/94172].)
- Remove `Jython` compatibility hacks from several stdlib modules and tests. (Contributed by Nikita Sobolev in gh-99482 [https://github.com/python/cpython/issues/99482].)
- Remove `_use_broken_old_ctypes_structure_semantics_` flag from **`ctypes`** module. (Contributed by Nikita Sobolev in gh-99285 [https://github.com/python/cpython/issues/99285].)

Porting to Python 3.12

This section lists previously described changes and other bugfixes that may require changes to your code.

Changes in the Python API

- More strict rules are now applied for numerical group references and group names in regular expressions. Only sequence of ASCII digits is now accepted as a numerical reference. The group name in bytes patterns and replacement strings can now only contain ASCII letters and digits and underscore. (Contributed by Serhiy Storchaka in gh-91760 [https://github.com/python/cpython/issues/91760].)

- Remove `randrange()` functionality deprecated since Python 3.10. Formerly, `randrange(10.0)` losslessly converted to `randrange(10)`. Now, it raises a **`TypeError`**. Also, the exception raised for non-integer values such as `randrange(10.5)` or `randrange('10')` has been changed from **`ValueError`** to **`TypeError`**. This also prevents bugs where `randrange(1e25)` would silently select from a larger range than `randrange(10**25)`. (Originally suggested by Serhiy Storchaka gh-86388 [https://github.com/python/cpython/issues/86388].)

- **`argparse.ArgumentParser`** changed encoding and error handler for reading arguments from file (e.g. `fromfile_prefix_chars` option) from default text encoding (e.g. **`locale.getpreferredencoding(False)`**) to filesystem encoding and error handler. Argument files should be encoded in UTF-8 instead of ANSI Codepage on Windows.

- Remove the `asyncore`-based `smtpd` module deprecated in Python 3.4.7 and 3.5.4. A recommended replacement is the **`asyncio`**-based aiosmtpd [https://pypi.org/project/aiosmtpd/] PyPI module.

- **`shlex.split()`**: Passing `None` for *s* argument now raises an exception, rather than reading **`sys.stdin`**. The feature was deprecated in Python 3.9. (Contributed by Victor Stinner in gh-94352 [https://github.com/python/cpython/issues/94352].)

- The **`os`** module no longer accepts bytes-like paths, like **`bytearray`** and **`memoryview`** types: only the exact **`bytes`** type is accepted for bytes strings. (Contributed by Victor Stinner in gh-98393 [https://github.com/python/cpython/issues/98393].)

- **`syslog.openlog()`** and **`syslog.closelog()`** now fail if used in subinterpreters. **`syslog.syslog()`** may still be used in subinterpreters, but now only if **`syslog.openlog()`** has already been called in the main interpreter. These new restrictions do not apply to the main interpreter, so only a very small set of users might be affected. This change helps with interpreter isolation. Furthermore, **`syslog`** is a wrapper around process-global resources, which are best managed from the main interpreter. (Contributed by Donghee Na in gh-99127 [https://github.com/python/cpython/issues/99127].)

- The undocumented locking behavior of **`cached_property()`** is removed, because it locked across all instances of the class, leading to high lock contention. This means that a cached property getter function could now run more than once for a single instance, if two threads race. For most simple cached properties (e.g. those that are idempotent and simply calculate a value based on other attributes of the instance) this will be fine. If synchronization is needed, implement locking within the cached property getter function or around multi-threaded access points.

- **`sys._current_exceptions()`** now returns a mapping from thread-id to an exception instance, rather than to a `(typ, exc, tb)` tuple. (Contributed by Irit Katriel in gh-103176 [https://github.com/python/cpython/issues/103176].)

- When extracting tar files using **`tarfile`** or **`shutil.unpack_archive()`**, pass the *filter* argument to limit features that may be surprising or dangerous. See Extraction filters for details.

- The output of the **`tokenize.tokenize()`** and **`tokenize.generate_tokens()`** functions is now changed due to the changes introduced in **PEP 701** [https://peps.python.org/pep-0701/]. This means that `STRING` tokens are not emitted any more for f-strings and the tokens described in **PEP 701** [https://peps.python.org/pep-0701/] are now produced instead: `FSTRING_START`, `FSTRING_MIDDLE` and `FSTRING_END` are now emitted for f-string “string” parts in addition to the appropriate tokens for the tokenization in the expression components. For example for the f-string `f"start {1+1} end"` the old version of the tokenizer emitted:

```
1,0-1,18:           STRING         'f"start {1+1} end"'
```

while the new version emits:

```
1,0-1,2:            FSTRING_START  'f"'
1,2-1,8:            FSTRING_MIDDLE 'start '
1,8-1,9:            OP             '{'
1,9-1,10:           NUMBER         '1'
1,10-1,11:          OP             '+'
1,11-1,12:          NUMBER         '1'
1,12-1,13:          OP             '}'
1,13-1,17:          FSTRING_MIDDLE ' end'
1,17-1,18:          FSTRING_END    '"'
```

Additionally, there may be some minor behavioral changes as a consequence of the changes required to support **PEP 701** [https://peps.python.org/pep-0701/]. Some of these changes include:

  - The `type` attribute of the tokens emitted when tokenizing some invalid Python characters such as `!` has changed from `ERRORTOKEN` to `OP`.
  - Incomplete single-line strings now also raise `tokenize.TokenError` as incomplete multiline strings do.
  - Some incomplete or invalid Python code now raises `tokenize.TokenError` instead of returning arbitrary `ERRORTOKEN` tokens when tokenizing it.
  - Mixing tabs and spaces as indentation in the same file is not supported anymore and will raise a `TabError`.

- The `threading` module now expects the **`_thread`** module to have an `_is_main_interpreter` attribute. It is a function with no arguments that returns `True` if the current interpreter is the main interpreter.

  Any library or application that provides a custom `_thread` module should provide `_is_main_interpreter()`. (See gh-112826 [https://github.com/python/cpython/issues/112826].)

Build Changes

- Python no longer uses `setup.py` to build shared C extension modules. Build parameters like headers and libraries are detected in `configure` script. Extensions are built by `Makefile`. Most extensions use `pkg-config` and fall back to manual detection. (Contributed by Christian Heimes in gh-93939 [https://github.com/python/cpython/issues/93939].)

- `va_start()` with two parameters, like `va_start(args, format)`, is now required to build Python. `va_start()` is no longer called with a single parameter. (Contributed by Kumar Aditya in gh-93207 [https://github.com/python/cpython/issues/93207].)

- CPython now uses the ThinLTO option as the default link time optimization policy if the Clang compiler accepts the flag. (Contributed by Donghee Na in gh-89536 [https://github.com/python/cpython/issues/89536].)

- Add `COMPILEALL_OPTS` variable in `Makefile` to override **`compileall`** options (default: `-j0`) in `make install`. Also merged the 3 `compileall` commands into a single command to build .pyc files for all optimization levels (0, 1, 2) at once. (Contributed by Victor Stinner in gh-99289 [https://github.com/python/cpython/issues/99289].)

- Add platform triplets for 64-bit LoongArch:

  - loongarch64-linux-gnusf
  - loongarch64-linux-gnuf32
  - loongarch64-linux-gnu

  (Contributed by Zhang Na in gh-90656 [https://github.com/python/cpython/issues/90656].)

- `PYTHON_FOR_REGEN` now require Python 3.10 or newer.

- Autoconf 2.71 and aclocal 1.16.4 is now required to regenerate `configure`. (Contributed by Christian Heimes in gh-89886 [https://github.com/python/cpython/issues/89886].)

- Windows builds and macOS installers from python.org now use OpenSSL 3.0.

C API Changes

New Features

- **PEP 697** [https://peps.python.org/pep-0697/]: Introduce the Unstable C API tier, intended for low-level tools like debuggers and JIT compilers. This API may change in each minor release of CPython without deprecation warnings. Its contents are marked by the `PyUnstable_` prefix in names.

  Code object constructors:

  - `PyUnstable_Code_New()` (renamed from `PyCode_New`)
  - `PyUnstable_Code_NewWithPosOnlyArgs()` (renamed from `PyCode_NewWithPosOnlyArgs`)

  Extra storage for code objects (**PEP 523** [https://peps.python.org/pep-0523/]):

  - `PyUnstable_Eval_RequestCodeExtraIndex()` (renamed from `_PyEval_RequestCodeExtraIndex`)
  - `PyUnstable_Code_GetExtra()` (renamed from `_PyCode_GetExtra`)
  - `PyUnstable_Code_SetExtra()` (renamed from `_PyCode_SetExtra`)

  The original names will continue to be available until the respective API changes.

  (Contributed by Petr Viktorin in gh-101101 [https://github.com/python/cpython/issues/101101].)

- **PEP 697** [https://peps.python.org/pep-0697/]: Add an API for extending types whose instance memory layout is opaque:

  - **`PyType_Spec.basicsize`** can be zero or negative to specify inheriting or extending the base class size.
  - **`PyObject_GetTypeData()`** and **`PyType_GetTypeDataSize()`** added to allow access to subclass-specific instance data.
  - **`Py_TPFLAGS_ITEMS_AT_END`** and **`PyObject_GetItemData()`** added to allow safely extending certain variable-sized types, including **`PyType_Type`**.

  - `Py_RELATIVE_OFFSET` added to allow defining `members` in terms of a subclass-specific struct.

  (Contributed by Petr Viktorin in gh-103509 [https://github.com/python/cpython/issues/103509].)

- Add the new limited C API function `PyType_FromMetaclass()`, which generalizes the existing `PyType_FromModuleAndSpec()` using an additional metaclass argument. (Contributed by Wenzel Jakob in gh-93012 [https://github.com/python/cpython/issues/93012].)
- API for creating objects that can be called using the vectorcall protocol was added to the Limited API:
  - `Py_TPFLAGS_HAVE_VECTORCALL`
  - `PyVectorcall_NARGS()`
  - `PyVectorcall_Call()`
  - `vectorcallfunc`

  The `Py_TPFLAGS_HAVE_VECTORCALL` flag is now removed from a class when the class's `__call__()` method is reassigned. This makes vectorcall safe to use with mutable types (i.e. heap types without the immutable flag, `Py_TPFLAGS_IMMUTABLETYPE`). Mutable types that do not override `tp_call` now inherit the `Py_TPFLAGS_HAVE_VECTORCALL` flag. (Contributed by Petr Viktorin in gh-93274 [https://github.com/python/cpython/issues/93274].)

  The `Py_TPFLAGS_MANAGED_DICT` and `Py_TPFLAGS_MANAGED_WEAKREF` flags have been added. This allows extensions classes to support object `__dict__` and weakrefs with less bookkeeping, using less memory and with faster access.

- API for performing calls using the vectorcall protocol was added to the Limited API:
  - `PyObject_Vectorcall()`
  - `PyObject_VectorcallMethod()`
  - `PY_VECTORCALL_ARGUMENTS_OFFSET`

  This means that both the incoming and outgoing ends of the vector call protocol are now available in the Limited API.

(Contributed by Wenzel Jakob in gh-98586 [https://github.com/python/cpython/issues/98586].)

- Add two new public functions, **`PyEval_SetProfileAllThreads()`** and **`PyEval_SetTraceAllThreads()`**, that allow to set tracing and profiling functions in all running threads in addition to the calling one. (Contributed by Pablo Galindo in gh-93503 [https://github.com/python/cpython/issues/93503].)
- Add new function **`PyFunction_SetVectorcall()`** to the C API which sets the vectorcall field of a given **`PyFunctionObject`**. (Contributed by Andrew Frost in gh-92257 [https://github.com/python/cpython/issues/92257].)
- The C API now permits registering callbacks via **`PyDict_AddWatcher()`**, **`PyDict_Watch()`** and related APIs to be called whenever a dictionary is modified. This is intended for use by optimizing interpreters, JIT compilers, or debuggers. (Contributed by Carl Meyer in gh-91052 [https://github.com/python/cpython/issues/91052].)
- Add **`PyType_AddWatcher()`** and **`PyType_Watch()`** API to register callbacks to receive notification on changes to a type. (Contributed by Carl Meyer in gh-91051 [https://github.com/python/cpython/issues/91051].)
- Add **`PyCode_AddWatcher()`** and **`PyCode_ClearWatcher()`** APIs to register callbacks to receive notification on creation and destruction of code objects. (Contributed by Itamar Oren in gh-91054 [https://github.com/python/cpython/issues/91054].)
- Add **`PyFrame_GetVar()`** and **`PyFrame_GetVarString()`** functions to get a frame variable by its name. (Contributed by Victor Stinner in gh-91248 [https://github.com/python/cpython/issues/91248].)
- Add **`PyErr_GetRaisedException()`** and **`PyErr_SetRaisedException()`** for saving and restoring the current exception. These functions return and accept a single exception object, rather than the triple arguments of the now-deprecated **`PyErr_Fetch()`** and **`PyErr_Restore()`**. This is less error prone and a bit more efficient.

(Contributed by Mark Shannon in gh-101578 [https://github.com/python/cpython/issues/101578].)

- Add `_PyErr_ChainExceptions1`, which takes an exception instance, to replace the legacy-API `_PyErr_ChainExceptions`, which is now deprecated. (Contributed by Mark Shannon in gh-101578 [https://github.com/python/cpython/issues/101578].)
- Add **`PyException_GetArgs()`** and **`PyException_SetArgs()`** as convenience functions for retrieving and modifying the **`args`** passed to the exception's constructor. (Contributed by Mark Shannon in gh-101578 [https://github.com/python/cpython/issues/101578].)
- Add **`PyErr_DisplayException()`**, which takes an exception instance, to replace the legacy-api **`PyErr_Display()`**. (Contributed by Irit Katriel in gh-102755 [https://github.com/python/cpython/issues/102755]).
- **PEP 683** [https://peps.python.org/pep-0683/]: Introduce *Immortal Objects*, which allows objects to bypass reference counts, and related changes to the C-API:
    - `_Py_IMMORTAL_REFCNT`: The reference count that defines an object

      as immortal.
    - `_Py_IsImmortal` Checks if an object has the immortal reference count.
    - `PyObject_HEAD_INIT` This will now initialize reference count to

      `_Py_IMMORTAL_REFCNT` when used with `Py_BUILD_CORE`.
    - `SSTATE_INTERNED_IMMORTAL` An identifier for interned unicode objects

      that are immortal.
    - `SSTATE_INTERNED_IMMORTAL_STATIC` An identifier for interned unicode

      objects that are immortal and static

  - `sys.getunicodeinternedsize` This returns the total number of unicode

    objects that have been interned. This is now needed for `refleak.py` to correctly track reference counts and allocated blocks

  (Contributed by Eddie Elizondo in gh-84436 [https://github.com/python/cpython/issues/84436].)

- **PEP 684** [https://peps.python.org/pep-0684/]: Add the new `Py_NewInterpreterFromConfig()` function and `PyInterpreterConfig`, which may be used to create sub-interpreters with their own GILs. (See PEP 684: A Per-Interpreter GIL for more info.) (Contributed by Eric Snow in gh-104110 [https://github.com/python/cpython/issues/104110].)

- In the limited C API version 3.12, `Py_INCREF()` and `Py_DECREF()` functions are now implemented as opaque function calls to hide implementation details. (Contributed by Victor Stinner in gh-105387 [https://github.com/python/cpython/issues/105387].)

Porting to Python 3.12

- Legacy Unicode APIs based on `Py_UNICODE*` representation has been removed. Please migrate to APIs based on UTF-8 or `wchar_t*`.

- Argument parsing functions like `PyArg_ParseTuple()` doesn't support `Py_UNICODE*` based format (e.g. `u`, `Z`) anymore. Please migrate to other formats for Unicode like `s`, `z`, `es`, and `U`.

- `tp_weaklist` for all static builtin types is always `NULL`. This is an internal-only field on `PyTypeObject` but we're pointing out the change in case someone happens to be accessing the field directly anyway. To avoid breakage, consider using the existing public C-API instead, or, if necessary, the (internal-only) `_PyObject_GET_WEAKREFS_LISTPTR()` macro.

- This internal-only `PyTypeObject.tp_subclasses` may now not be a valid object pointer. Its type was changed to void* to reflect this. We mention this in case someone happens to be accessing the internal-only field directly.

To get a list of subclasses, call the Python method `__subclasses__()` (using `PyObject_CallMethod()`, for example).

- Add support of more formatting options (left aligning, octals, uppercase hexadecimals, **`intmax_t`**, **`ptrdiff_t`**, **`wchar_t`** C strings, variable width and precision) in `PyUnicode_FromFormat()` and `PyUnicode_FromFormatV()`. (Contributed by Serhiy Storchaka in gh-98836 [https://github.com/python/cpython/issues/98836].)

- An unrecognized format character in `PyUnicode_FromFormat()` and `PyUnicode_FromFormatV()` now sets a `SystemError`. In previous versions it caused all the rest of the format string to be copied as-is to the result string, and any extra arguments discarded. (Contributed by Serhiy Storchaka in gh-95781 [https://github.com/python/cpython/issues/95781].)

- Fix wrong sign placement in `PyUnicode_FromFormat()` and `PyUnicode_FromFormatV()`. (Contributed by Philip Georgi in gh-95504 [https://github.com/python/cpython/issues/95504].)

- Extension classes wanting to add a `__dict__` or weak reference slot should use `Py_TPFLAGS_MANAGED_DICT` and `Py_TPFLAGS_MANAGED_WEAKREF` instead of `tp_dictoffset` and `tp_weaklistoffset`, respectively. The use of `tp_dictoffset` and `tp_weaklistoffset` is still supported, but does not fully support multiple inheritance (gh-95589 [https://github.com/python/cpython/issues/95589]), and performance may be worse. Classes declaring `Py_TPFLAGS_MANAGED_DICT` must call **`_PyObject_VisitManagedDict()`** and **`_PyObject_ClearManagedDict()`** to traverse and clear their instance’s dictionaries. To clear weakrefs, call `PyObject_ClearWeakRefs()`, as before.

- The `PyUnicode_FSDecoder()` function no longer accepts bytes-like paths, like `bytearray` and `memoryview` types: only the exact `bytes` type is accepted for bytes strings. (Contributed by Victor Stinner in gh-98393 [https://github.com/python/cpython/issues/98393].)

- The **`Py_CLEAR`**, **`Py_SETREF`** and **`Py_XSETREF`** macros now only evaluate their arguments once. If an argument has side effects, these side effects are no longer duplicated. (Contributed by Victor Stinner in gh-98724 [https://github.com/python/cpython/issues/98724].)

- The interpreter's error indicator is now always normalized. This means that **`PyErr_SetObject()`**, **`PyErr_SetString()`** and the other functions that set the error indicator now normalize the exception before storing it. (Contributed by Mark Shannon in gh-101578 [https://github.com/python/cpython/issues/101578].)

- `_Py_RefTotal` is no longer authoritative and only kept around for ABI compatibility. Note that it is an internal global and only available on debug builds. If you happen to be using it then you'll need to start using `_Py_GetGlobalRefTotal()`.

- The following functions now select an appropriate metaclass for the newly created type:

  - **`PyType_FromSpec()`**
  - **`PyType_FromSpecWithBases()`**
  - **`PyType_FromModuleAndSpec()`**

  Creating classes whose metaclass overrides **`tp_new`** is deprecated, and in Python 3.14+ it will be disallowed. Note that these functions ignore `tp_new` of the metaclass, possibly allowing incomplete initialization.

  Note that **`PyType_FromMetaclass()`** (added in Python 3.12) already disallows creating classes whose metaclass overrides `tp_new` (**`__new__()`** in Python).

  Since `tp_new` overrides almost everything `PyType_From*` functions do, the two are incompatible with each other. The existing behavior – ignoring the metaclass for several steps of type creation – is unsafe in general, since (meta)classes assume that `tp_new` was called. There is no simple general workaround. One of the following may work for you:

  - If you control the metaclass, avoid using `tp_new` in it:
    - If initialization can be skipped, it can be done in **`tp_init`** instead.
    - If the metaclass doesn't need to be instantiated from Python, set its `tp_new` to `NULL` using the **`Py_TPFLAGS_DISALLOW_INSTANTIATION`** flag. This makes it acceptable for `PyType_From*` functions.
  - Avoid `PyType_From*` functions: if you don't need C-specific features (slots or setting the instance size), create types by calling the metaclass.
  - If you *know* the `tp_new` can be skipped safely, filter the deprecation warning out using **`warnings.catch_warnings()`** from Python.

- **`PyOS_InputHook`** and **`PyOS_ReadlineFunctionPointer`** are no longer called in subinterpreters. This is because clients generally rely on process-wide global state (since these callbacks have no way of recovering extension module state).

  This also avoids situations where extensions may find themselves running in a subinterpreter that they don't support (or haven't yet been loaded in). See gh-104668 [https://github.com/python/cpython/issues/104668] for more info.

- **`PyLongObject`** has had its internals changed for better performance. Although the internals of **`PyLongObject`** are private, they are used by some extension modules. The internal fields should no longer be accessed directly, instead the API functions beginning `PyLong_...` should be used instead. Two new *unstable* API functions are provided for efficient access to the value of **`PyLongObject`**s which fit into a single machine word:

  - **`PyUnstable_Long_IsCompact()`**
  - **`PyUnstable_Long_CompactValue()`**

- Custom allocators, set via **`PyMem_SetAllocator()`**, are now required to be thread-safe, regardless of memory domain. Allocators that don't have their own state, including "hooks", are not affected. If your custom allocator is not already thread-safe and you need guidance then please create a new GitHub issue and CC `@ericsnowcurrently`.

Deprecated

- In accordance with **PEP 699** [https://peps.python.org/pep-0699/], the `ma_version_tag` field in `PyDictObject` is deprecated for extension modules. Accessing this field will generate a compiler warning at compile time. This field will be removed in Python 3.14. (Contributed by Ramvikrams and Kumar Aditya in gh-101193 [https://github.com/python/cpython/issues/101193]. PEP by Ken Jin.)
- Deprecate global configuration variable:
  - `Py_DebugFlag`: use `PyConfig.parser_debug`
  - `Py_VerboseFlag`: use `PyConfig.verbose`
  - `Py_QuietFlag`: use `PyConfig.quiet`
  - `Py_InteractiveFlag`: use `PyConfig.interactive`
  - `Py_InspectFlag`: use `PyConfig.inspect`
  - `Py_OptimizeFlag`: use `PyConfig.optimization_level`
  - `Py_NoSiteFlag`: use `PyConfig.site_import`
  - `Py_BytesWarningFlag`: use `PyConfig.bytes_warning`
  - `Py_FrozenFlag`: use `PyConfig.pathconfig_warnings`
  - `Py_IgnoreEnvironmentFlag`: use `PyConfig.use_environment`
  - `Py_DontWriteBytecodeFlag`: use `PyConfig.write_bytecode`
  - `Py_NoUserSiteDirectory`: use `PyConfig.user_site_directory`
  - `Py_UnbufferedStdioFlag`: use `PyConfig.buffered_stdio`
  - `Py_HashRandomizationFlag`: use `PyConfig.use_hash_seed` and `PyConfig.hash_seed`
  - `Py_IsolatedFlag`: use `PyConfig.isolated`
  - `Py_LegacyWindowsFSEncodingFlag`: use `PyPreConfig.legacy_windows_fs_encoding`
  - `Py_LegacyWindowsStdioFlag`: use `PyConfig.legacy_windows_stdio`
  - **`Py_FileSystemDefaultEncoding`**: use `PyConfig.filesystem_encoding`
  - **`Py_HasFileSystemDefaultEncoding`**: use `PyConfig.filesystem_encoding`
  - **`Py_FileSystemDefaultEncodeErrors`**: use `PyConfig.filesystem_errors`

  - **`Py_UTF8Mode`**: use `PyPreConfig.utf8_mode` (see `Py_PreInitialize()`)

  The `Py_InitializeFromConfig()` API should be used with `PyConfig` instead. (Contributed by Victor Stinner in gh-77782 [https://github.com/python/cpython/issues/77782].)

- Creating `immutable types` with mutable bases is deprecated and will be disabled in Python 3.14. (gh-95388 [https://github.com/python/cpython/issues/95388])

- The `structmember.h` header is deprecated, though it continues to be available and there are no plans to remove it.

  Its contents are now available just by including `Python.h`, with a `Py` prefix added if it was missing:

  - `PyMemberDef`, `PyMember_GetOne()` and `PyMember_SetOne()`
  - Type macros like `Py_T_INT`, `Py_T_DOUBLE`, etc. (previously `T_INT`, `T_DOUBLE`, etc.)
  - The flags `Py_READONLY` (previously `READONLY`) and `Py_AUDIT_READ` (previously all uppercase)

  Several items are not exposed from `Python.h`:

  - `T_OBJECT` (use `Py_T_OBJECT_EX`)
  - `T_NONE` (previously undocumented, and pretty quirky)
  - The macro `WRITE_RESTRICTED` which does nothing.
  - The macros `RESTRICTED` and `READ_RESTRICTED`, equivalents of `Py_AUDIT_READ`.
  - In some configurations, `<stddef.h>` is not included from `Python.h`. It should be included manually when using `offsetof()`.

  The deprecated header continues to provide its original contents under the original names. Your old code can stay unchanged, unless the extra include and non-namespaced macros bother you greatly.

  (Contributed in gh-47146 [https://github.com/python/cpython/issues/47146] by Petr Viktorin, based on earlier work by Alexander Belopolsky and Matthias Braun.)

- `PyErr_Fetch()` and `PyErr_Restore()` are deprecated. Use `PyErr_GetRaisedException()` and `PyErr_SetRaisedException()` instead. (Contributed by Mark Shannon in gh-101578 [https://github.com/python/cpython/issues/101578].)

- **`PyErr_Display()`** is deprecated. Use `PyErr_DisplayException()` instead. (Contributed by Irit Katriel in gh-102755 [https://github.com/python/cpython/issues/102755]).

- `_PyErr_ChainExceptions` is deprecated. Use `_PyErr_ChainExceptions1` instead. (Contributed by Irit Katriel in gh-102192 [https://github.com/python/cpython/issues/102192].)

- Using `PyType_FromSpec()`, `PyType_FromSpecWithBases()` or `PyType_FromModuleAndSpec()` to create a class whose metaclass overrides `tp_new` is deprecated. Call the metaclass instead.

Pending removal in Python 3.14

- The `ma_version_tag` field in `PyDictObject` for extension modules (**PEP 699** [https://peps.python.org/pep-0699/]; gh-101193 [https://github.com/python/cpython/issues/101193]).
- Creating `immutable types` with mutable bases (gh-95388 [https://github.com/python/cpython/issues/95388]).

Pending removal in Python 3.15

- The `PyImport_ImportModuleNoBlock()`: Use `PyImport_ImportModule()` instead.

- `PyWeakref_GetObject()` and `PyWeakref_GET_OBJECT()`: Use `PyWeakref_GetRef()` instead. The pythoncapi-compat project [https://github.com/python/pythoncapi-compat/] can be used to get `PyWeakref_GetRef()` on Python 3.12 and older.

- `Py_UNICODE` type and the **`Py_UNICODE_WIDE`** macro: Use **`wchar_t`** instead.

- `PyUnicode_AsDecodedObject()`: Use `PyCodec_Decode()` instead.
- `PyUnicode_AsDecodedUnicode()`: Use `PyCodec_Decode()` instead; Note that some codecs (for example, “base64”) may return a type other than `str`, such as `bytes`.
- `PyUnicode_AsEncodedObject()`: Use `PyCodec_Encode()` instead.
- `PyUnicode_AsEncodedUnicode()`: Use `PyCodec_Encode()` instead; Note that some codecs (for example, “base64”) may return a type other than `bytes`, such as `str`.
- Python initialization functions, deprecated in Python 3.13:
  - `Py_GetPath()`: Use `PyConfig_Get("module_search_paths")` (`sys.path`) instead.
  - `Py_GetPrefix()`: Use `PyConfig_Get("base_prefix")` (`sys.base_prefix`) instead. Use `PyConfig_Get("prefix")` (`sys.prefix`) if virtual environments need to be handled.
  - `Py_GetExecPrefix()`: Use `PyConfig_Get("base_exec_prefix")` (`sys.base_exec_prefix`) instead. Use `PyConfig_Get("exec_prefix")` (`sys.exec_prefix`) if virtual environments need to be handled.
  - `Py_GetProgramFullPath()`: Use `PyConfig_Get("executable")` (`sys.executable`) instead.
  - `Py_GetProgramName()`: Use `PyConfig_Get("executable")` (`sys.executable`) instead.
  - `Py_GetPythonHome()`: Use `PyConfig_Get("home")` or the `PYTHONHOME` environment variable instead.

  The pythoncapi-compat project [https://github.com/python/pythoncapi-compat/] can be used to get `PyConfig_Get()` on Python 3.13 and older.
- Functions to configure Python’s initialization, deprecated in Python 3.11:
  - `PySys_SetArgvEx()`: Set `PyConfig.argv` instead.
  - `PySys_SetArgv()`: Set `PyConfig.argv` instead.
  - `Py_SetProgramName()`: Set `PyConfig.program_name` instead.

  - `Py_SetPythonHome()`: Set `PyConfig.home` instead.
  - `PySys_ResetWarnOptions()`: Clear `sys.warnoptions` and `warnings.filters` instead.

  The `Py_InitializeFromConfig()` API should be used with `PyConfig` instead.

- Global configuration variables:
  - `Py_DebugFlag`: Use `PyConfig.parser_debug` or `PyConfig_Get("parser_debug")` instead.
  - `Py_VerboseFlag`: Use `PyConfig.verbose` or `PyConfig_Get("verbose")` instead.
  - `Py_QuietFlag`: Use `PyConfig.quiet` or `PyConfig_Get("quiet")` instead.
  - `Py_InteractiveFlag`: Use `PyConfig.interactive` or `PyConfig_Get("interactive")` instead.
  - `Py_InspectFlag`: Use `PyConfig.inspect` or `PyConfig_Get("inspect")` instead.
  - `Py_OptimizeFlag`: Use `PyConfig.optimization_level` or `PyConfig_Get("optimization_level")` instead.
  - `Py_NoSiteFlag`: Use `PyConfig.site_import` or `PyConfig_Get("site_import")` instead.
  - `Py_BytesWarningFlag`: Use `PyConfig.bytes_warning` or `PyConfig_Get("bytes_warning")` instead.
  - `Py_FrozenFlag`: Use `PyConfig.pathconfig_warnings` or `PyConfig_Get("pathconfig_warnings")` instead.
  - `Py_IgnoreEnvironmentFlag`: Use `PyConfig.use_environment` or `PyConfig_Get("use_environment")` instead.
  - `Py_DontWriteBytecodeFlag`: Use `PyConfig.write_bytecode` or `PyConfig_Get("write_bytecode")` instead.
  - `Py_NoUserSiteDirectory`: Use `PyConfig.user_site_directory` or `PyConfig_Get("user_site_directory")` instead.
  - `Py_UnbufferedStdioFlag`: Use `PyConfig.buffered_stdio` or `PyConfig_Get("buffered_stdio")` instead.
  - `Py_HashRandomizationFlag`: Use `PyConfig.use_hash_seed` and `PyConfig.hash_seed` or `PyConfig_Get("hash_seed")` instead.
  - `Py_IsolatedFlag`: Use `PyConfig.isolated` or `PyConfig_Get("isolated")` instead.

  - `Py_LegacyWindowsFSEncodingFlag`: Use `PyPreConfig.legacy_windows_fs_encoding` or `PyConfig_Get("legacy_windows_fs_encoding")` instead.
  - `Py_LegacyWindowsStdioFlag`: Use `PyConfig.legacy_windows_stdio` or `PyConfig_Get("legacy_windows_stdio")` instead.
  - **`Py_FileSystemDefaultEncoding`**, **`Py_HasFileSystemDefaultEncoding`**: Use `PyConfig.filesystem_encoding` or `PyConfig_Get("filesystem_encoding")` instead.
  - **`Py_FileSystemDefaultEncodeErrors`**: Use `PyConfig.filesystem_errors` or `PyConfig_Get("filesystem_errors")` instead.
  - **`Py_UTF8Mode`**: Use `PyPreConfig.utf8_mode` or `PyConfig_Get("utf8_mode")` instead. (see `Py_PreInitialize()`)

The `Py_InitializeFromConfig()` API should be used with `PyConfig` to set these options. Or `PyConfig_Get()` can be used to get these options at runtime.

Pending removal in Python 3.16

- The bundled copy of `libmpdec`.

Pending removal in future versions

The following APIs are deprecated and will be removed, although there is currently no date scheduled for their removal.

- `Py_TPFLAGS_HAVE_FINALIZE`: Unneeded since Python 3.8.
- `PyErr_Fetch()`: Use `PyErr_GetRaisedException()` instead.
- `PyErr_NormalizeException()`: Use `PyErr_GetRaisedException()` instead.
- `PyErr_Restore()`: Use `PyErr_SetRaisedException()` instead.
- `PyModule_GetFilename()`: Use `PyModule_GetFilenameObject()` instead.
- `PyOS_AfterFork()`: Use `PyOS_AfterFork_Child()` instead.
- `PySlice_GetIndicesEx()`: Use `PySlice_Unpack()` and `PySlice_AdjustIndices()` instead.
- `PyUnicode_READY()`: Unneeded since Python 3.12
- **`PyErr_Display()`**: Use `PyErr_DisplayException()` instead.
- **`_PyErr_ChainExceptions()`**: Use **`_PyErr_ChainExceptions1()`** instead.

- **`PyBytesObject.ob_shash`** member: call `PyObject_Hash()` instead.
- Thread Local Storage (TLS) API:
  - `PyThread_create_key()`: Use `PyThread_tss_alloc()` instead.
  - `PyThread_delete_key()`: Use `PyThread_tss_free()` instead.
  - `PyThread_set_key_value()`: Use `PyThread_tss_set()` instead.
  - `PyThread_get_key_value()`: Use `PyThread_tss_get()` instead.
  - `PyThread_delete_key_value()`: Use `PyThread_tss_delete()` instead.
  - `PyThread_ReInitTLS()`: Unneeded since Python 3.7.

Removed

- Remove the `token.h` header file. There was never any public tokenizer C API. The `token.h` header file was only designed to be used by Python internals. (Contributed by Victor Stinner in gh-92651 [https://github.com/python/cpython/issues/92651].)
- Legacy Unicode APIs have been removed. See **PEP 623** [https://peps.python.org/pep-0623/] for detail.
  - **`PyUnicode_WCHAR_KIND`**
  - **`PyUnicode_AS_UNICODE()`**
  - **`PyUnicode_AsUnicode()`**
  - **`PyUnicode_AsUnicodeAndSize()`**
  - **`PyUnicode_AS_DATA()`**
  - **`PyUnicode_FromUnicode()`**
  - **`PyUnicode_GET_SIZE()`**
  - **`PyUnicode_GetSize()`**
  - **`PyUnicode_GET_DATA_SIZE()`**
- Remove the `PyUnicode_InternImmortal()` function macro. (Contributed by Victor Stinner in gh-85858 [https://github.com/python/cpython/issues/85858].)

What’s New In Python 3.11

Editor:: Pablo Galindo Salgado

This article explains the new features in Python 3.11, compared to 3.10. Python 3.11 was released on October 24, 2022. For full details, see the changelog.

Summary – Release highlights

- Python 3.11 is between 10-60% faster than Python 3.10. On average, we measured a 1.25x speedup on the standard benchmark suite. See Faster CPython for details.

New syntax features:

- PEP 654: Exception Groups and except*

New built-in features:

- PEP 678: Exceptions can be enriched with notes

New standard library modules:

- **PEP 680** [https://peps.python.org/pep-0680/]: **`tomllib`** — Support for parsing TOML [https://toml.io/] in the Standard Library

Interpreter improvements:

- PEP 657: Fine-grained error locations in tracebacks
- New **`-P`** command line option and **`PYTHONSAFEPATH`** environment variable to disable automatically prepending potentially unsafe paths to **`sys.path`**

New typing features:

- PEP 646: Variadic generics
- PEP 655: Marking individual TypedDict items as required or not-required
- PEP 673: Self type
- PEP 675: Arbitrary literal string type

- PEP 681: Data class transforms

Important deprecations, removals and restrictions:

- **PEP 594** [https://peps.python.org/pep-0594/]: Many legacy standard library modules have been deprecated and will be removed in Python 3.13
- **PEP 624** [https://peps.python.org/pep-0624/]: Py_UNICODE encoder APIs have been removed
- **PEP 670** [https://peps.python.org/pep-0670/]: Macros converted to static inline functions

New Features

PEP 657: Fine-grained error locations in tracebacks

When printing tracebacks, the interpreter will now point to the exact expression that caused the error, instead of just the line. For example:

```
Traceback (most recent call last):
  File "distance.py", line 11, in <module>
    print(manhattan_distance(p1, p2))
          ^^^^^^^^^^^^^^^^^^^^^^^^^^
  File "distance.py", line 6, in manhattan_distance
    return abs(point_1.x - point_2.x) + abs(point_1.y - point_2.y)
                           ^^^^^^^^^
AttributeError: 'NoneType' object has no attribute 'x'
```

Previous versions of the interpreter would point to just the line, making it ambiguous which object was `None`. These enhanced errors can also be helpful when dealing with deeply nested **`dict`** objects and multiple function calls:

```
Traceback (most recent call last):
  File "query.py", line 37, in <module>
    magic_arithmetic('foo')
  File "query.py", line 18, in magic_arithmetic
    return add_counts(x) / 25
           ^^^^^^^^^^^^^
  File "query.py", line 24, in add_counts
    return 25 + query_user(user1) + query_user(user2)
                ^^^^^^^^^^^^^^^^^
  File "query.py", line 32, in query_user
    return 1 + query_count(db, response['a']['b']['c']['user'],
retry=True)
                               ~~~~~~~~~~~~~~~~~~^^^^^
TypeError: 'NoneType' object is not subscriptable
```

As well as complex arithmetic expressions:

```
Traceback (most recent call last):
  File "calculation.py", line 54, in <module>
    result = (x / y / z) * (a / b / c)
              ~~~~~~^~~
ZeroDivisionError: division by zero
```

Additionally, the information used by the enhanced traceback feature is made available via a general API, that can be used to correlate bytecode instructions with source code location. This information can be retrieved using:

- The `codeobject.co_positions()` method in Python.
- The `PyCode_Addr2Location()` function in the C API.

See **PEP 657** [https://peps.python.org/pep-0657/] for more details. (Contributed by Pablo Galindo, Batuhan Taskaya and Ammar Askar in bpo-43950 [https://bugs.python.org/issue?@action=redirect&bpo=43950].)

Note

This feature requires storing column positions in Code Objects, which may result in a small increase in interpreter memory usage and disk usage for compiled Python files. To avoid storing the extra information and deactivate printing the extra traceback information, use the `-X no_debug_ranges` command line option or the `PYTHONNODEBUGRANGES` environment variable.

PEP 654: Exception Groups and `except*`

PEP 654 [https://peps.python.org/pep-0654/] introduces language features that enable a program to raise and handle multiple unrelated exceptions simultaneously. The builtin types `ExceptionGroup` and `BaseExceptionGroup` make it possible to group exceptions and raise them together, and the new `except*` syntax generalizes `except` to match subgroups of exception groups.

See **PEP 654** [https://peps.python.org/pep-0654/] for more details.

(Contributed by Irit Katriel in bpo-45292 [https://bugs.python.org/issue?@action=redirect&bpo=45292]. PEP written by Irit Katriel, Yury Selivanov and Guido van Rossum.)

PEP 678: Exceptions can be enriched with notes

The `add_note()` method is added to `BaseException`. It can be used to enrich exceptions with context information that is not available at the time when the exception is raised. The added notes appear in the default traceback.

See **PEP 678** [https://peps.python.org/pep-0678/] for more details.

(Contributed by Irit Katriel in bpo-45607 [https://bugs.python.org/issue?@action=redirect&bpo=45607]. PEP written by Zac Hatfield-Dodds.)

Windows `py.exe` launcher improvements

The copy of the Python install manager included with Python 3.11 has been significantly updated. It now supports company/tag syntax as defined in **PEP 514** [https://peps.python.org/pep-0514/] using the `-V:<company>/<tag>` argument instead of the limited `-<major>.<minor>`. This allows launching distributions other than `PythonCore`, the one hosted on python.org [https://www.python.org].

When using `-V:` selectors, either company or tag can be omitted, but all installs will be searched. For example, `-V:OtherPython/` will select the "best" tag registered for `OtherPython`, while `-V:3.11` or `-V:/3.11` will select the "best" distribution with tag `3.11`.

When using the legacy `-<major>`, `-<major>.<minor>`, `-<major>-<bitness>` or `-<major>.<minor>-<bitness>` arguments, all existing behaviour should be preserved from past versions, and only releases from `PythonCore` will be selected. However, the `-64` suffix now implies "not 32-bit" (not necessarily x86-64), as there are multiple supported 64-bit platforms. 32-bit runtimes are detected by checking the runtime's tag for a `-32` suffix. All releases of Python since 3.5 have included this in their 32-bit builds.

New Features Related to Type Hints

This section covers major changes affecting **PEP 484** [https://peps.python.org/pep-0484/] type hints and the `typing` module.

PEP 646: Variadic generics

PEP 484 [https://peps.python.org/pep-0484/] previously introduced `TypeVar`, enabling creation of generics parameterised with a single type. **PEP 646** [https://peps.python.org/pep-0646/] adds `TypeVarTuple`, enabling parameterisation with an *arbitrary* number of types. In other words, a `TypeVarTuple` is a *variadic* type variable, enabling *variadic* generics.

This enables a wide variety of use cases. In particular, it allows the type of array-like structures in numerical computing libraries such as NumPy and TensorFlow to be parameterised with the array *shape*. Static type checkers will now be able to catch shape-related bugs in code that uses these libraries.

See **PEP 646** [https://peps.python.org/pep-0646/] for more details.

(Contributed by Matthew Rahtz in bpo-43224 [https://bugs.python.org/issue?@action=redirect&bpo=43224], with contributions by Serhiy Storchaka and Jelle Zijlstra. PEP written by Mark Mendoza, Matthew Rahtz, Pradeep Kumar Srinivasan, and Vincent Siles.)

PEP 655: Marking individual `TypedDict` items as required or not-required

`Required` and `NotRequired` provide a straightforward way to mark whether individual items in a `TypedDict` must be present. Previously, this was only possible using inheritance.

All fields are still required by default, unless the *total* parameter is set to `False`, in which case all fields are still not-required by default. For example, the following specifies a **`TypedDict`** with one required and one not-required key:

```
class Movie(TypedDict):
   title: str
   year: NotRequired[int]

m1: Movie = {"title": "Black Panther", "year": 2018}  # OK
m2: Movie = {"title": "Star Wars"}  # OK (year is not required)
m3: Movie = {"year": 2022}  # ERROR (missing required field title)
```

The following definition is equivalent:

```
class Movie(TypedDict, total=False):
   title: Required[str]
   year: int
```

See **PEP 655** [https://peps.python.org/pep-0655/] for more details.

(Contributed by David Foster and Jelle Zijlstra in bpo-47087 [https://bugs.python.org/issue?@action=redirect&bpo=47087]. PEP written by David Foster.)

PEP 673: `Self` type

The new `Self` annotation provides a simple and intuitive way to annotate methods that return an instance of their class. This behaves the same as the `TypeVar`-based approach **specified in PEP 484** [https://peps.python.org/pep-0484/#annotating-instance-and-class-methods], but is more concise and easier to follow.

Common use cases include alternative constructors provided as `classmethod`s, and `__enter__()` methods that return `self`:

```
class MyLock:
    def __enter__(self) -> Self:
        self.lock()
        return self

    ...

class MyInt:
    @classmethod
    def fromhex(cls, s: str) -> Self:
        return cls(int(s, 16))

    ...
```

`Self` can also be used to annotate method parameters or attributes of the same type as their enclosing class.

See **PEP 673** [https://peps.python.org/pep-0673/] for more details.

(Contributed by James Hilton-Balfe in bpo-46534 [https://bugs.python.org/issue?@action=redirect&bpo=46534]. PEP written by Pradeep Kumar Srinivasan and James Hilton-Balfe.)

PEP 675: Arbitrary literal string type

The new `LiteralString` annotation may be used to indicate that a function parameter can be of any literal string type. This allows a function to accept arbitrary literal string types, as well as strings created from other literal strings. Type checkers can then enforce that sensitive functions, such as those that execute SQL statements or shell commands, are called only with static arguments, providing protection against injection attacks.

For example, a SQL query function could be annotated as follows:

```
def run_query(sql: LiteralString) -> ...
    ...

def caller(
    arbitrary_string: str,
    query_string: LiteralString,
    table_name: LiteralString,
) -> None:
    run_query("SELECT * FROM students")       # ok
    run_query(query_string)                   # ok
    run_query("SELECT * FROM " + table_name)  # ok
    run_query(arbitrary_string)               # type checker error
    run_query(                                # type checker error
        f"SELECT * FROM students WHERE name = {arbitrary_string}"
    )
```

See **PEP 675** [https://peps.python.org/pep-0675/] for more details.

(Contributed by Jelle Zijlstra in bpo-47088 [https://bugs.python.org/issue?@action=redirect&bpo=47088]. PEP written by Pradeep Kumar Srinivasan and Graham Bleaney.)

PEP 681: Data class transforms

`dataclass_transform` may be used to decorate a class, metaclass, or a function that is itself a decorator. The presence of `@dataclass_transform()` tells a static type checker that the decorated object performs runtime “magic” that transforms a class, giving it **`dataclass`**-like behaviors.

For example:

```
# The create_model decorator is defined by a library.
@typing.dataclass_transform()
def create_model(cls: Type[T]) -> Type[T]:
    cls.__init__ = ...
    cls.__eq__ = ...
    cls.__ne__ = ...
    return cls

# The create_model decorator can now be used to create new model classes:
```

```
@create_model
class CustomerModel:
    id: int
    name: str

c = CustomerModel(id=327, name="Eric Idle")
```

See **PEP 681** [https://peps.python.org/pep-0681/] for more details.

(Contributed by Jelle Zijlstra in gh-91860 [https://github.com/python/cpython/issues/91860]. PEP written by Erik De Bonte and Eric Traut.)

PEP 563 may not be the future

PEP 563 [https://peps.python.org/pep-0563/] Postponed Evaluation of Annotations (the `from __future__ import annotations` future statement) that was originally planned for release in Python 3.10 has been put on hold indefinitely. See this message from the Steering Council [https://mail.python.org/archives/list/python-dev@python.org/message/VIZEBX5EYMSYIJNDBF6DMUMZOCWHARSO/] for more information.

Other Language Changes

- Starred unpacking expressions can now be used in **`for`** statements. (See bpo-46725 [https://bugs.python.org/issue?@action=redirect&bpo=46725] for more details.)
- Asynchronous comprehensions are now allowed inside comprehensions in asynchronous functions. Outer comprehensions implicitly become asynchronous in this case. (Contributed by Serhiy Storchaka in bpo-33346 [https://bugs.python.org/issue?@action=redirect&bpo=33346].)
- A **`TypeError`** is now raised instead of an **`AttributeError`** in **`with`** statements and **`contextlib.ExitStack.enter_context()`** for objects that do not support the context manager protocol, and in **`async with`** statements and **`contextlib.AsyncExitStack.enter_async_context()`** for objects not supporting the asynchronous context manager protocol. (Contributed by Serhiy Storchaka in bpo-12022 [https://bugs.python.org/issue?@action=redirect&bpo=12022] and bpo-44471 [https://bugs.python.org/issue?@action=redirect&bpo=44471].)
- Added **`object.__getstate__()`**, which provides the default implementation of the **`__getstate__()`** method. **`copy`**ing and **`pickle`**ing

instances of subclasses of builtin types `bytearray`, `set`, `frozenset`, `collections.OrderedDict`, `collections.deque`, `weakref.WeakSet`, and `datetime.tzinfo` now copies and pickles instance attributes implemented as slots. This change has an unintended side effect: It trips up a small minority of existing Python projects not expecting `object.__getstate__()` to exist. See the later comments on gh-70766 [https://github.com/python/cpython/issues/70766] for discussions of what workarounds such code may need. (Contributed by Serhiy Storchaka in bpo-26579 [https://bugs.python.org/issue?@action=redirect&bpo=26579].)

- Added a `-P` command line option and a `PYTHONSAFEPATH` environment variable, which disable the automatic prepending to `sys.path` of the script's directory when running a script, or the current directory when using `-c` and `-m`. This ensures only stdlib and installed modules are picked up by `import`, and avoids unintentionally or maliciously shadowing modules with those in a local (and typically user-writable) directory. (Contributed by Victor Stinner in gh-57684 [https://github.com/python/cpython/issues/57684].)
- A `"z"` option was added to the Format Specification Mini-Language that coerces negative to positive zero after rounding to the format precision. See **PEP 682** [https://peps.python.org/pep-0682/] for more details. (Contributed by John Belmonte in gh-90153 [https://github.com/python/cpython/issues/90153].)
- Bytes are no longer accepted on `sys.path`. Support broke sometime between Python 3.2 and 3.6, with no one noticing until after Python 3.10.0 was released. In addition, bringing back support would be problematic due to interactions between `-b` and `sys.path_importer_cache` when there is a mixture of `str` and `bytes` keys. (Contributed by Thomas Grainger in gh-91181 [https://github.com/python/cpython/issues/91181].)

Other CPython Implementation Changes

- The special methods `__complex__()` for `complex` and `__bytes__()` for `bytes` are implemented to support the `typing.SupportsComplex` and `typing.SupportsBytes` protocols. (Contributed by Mark Dickinson and Donghee Na in bpo-24234 [https://bugs.python.org/issue?@action=redirect&bpo=24234].)
- `siphash13` is added as a new internal hashing algorithm. It has similar security properties as `siphash24`, but it is slightly faster for long inputs. `str`, `bytes`, and some other types now use it as the

default algorithm for **`hash()`**. **PEP 552** [https://peps.python.org/pep-0552/] hash-based .pyc files now use `siphash13` too. (Contributed by Inada Naoki in bpo-29410 [https://bugs.python.org/issue?@action=redirect&bpo=29410].)
- When an active exception is re-raised by a **`raise`** statement with no parameters, the traceback attached to this exception is now always `sys.exc_info()[1].__traceback__`. This means that changes made to the traceback in the current **`except`** clause are reflected in the re-raised exception. (Contributed by Irit Katriel in bpo-45711 [https://bugs.python.org/issue?@action=redirect&bpo=45711].)
- The interpreter state's representation of handled exceptions (aka `exc_info` or `_PyErr_StackItem`) now only has the `exc_value` field; `exc_type` and `exc_traceback` have been removed, as they can be derived from `exc_value`. (Contributed by Irit Katriel in bpo-45711 [https://bugs.python.org/issue?@action=redirect&bpo=45711].)
- A new command line option, `AppendPath`, has been added for the Windows installer. It behaves similarly to `PrependPath`, but appends the install and scripts directories instead of prepending them. (Contributed by Bastian Neuburger in bpo-44934 [https://bugs.python.org/issue?@action=redirect&bpo=44934].)
- The **`PyConfig.module_search_paths_set`** field must now be set to `1` for initialization to use **`PyConfig.module_search_paths`** to initialize **`sys.path`**. Otherwise, initialization will recalculate the path and replace any values added to `module_search_paths`.
- The output of the **`--help`** option now fits in 50 lines/80 columns. Information about Python environment variables and **`-X`** options is now available using the respective **`--help-env`** and **`--help-xoptions`** flags, and with the new **`--help-all`**. (Contributed by Éric Araujo in bpo-46142 [https://bugs.python.org/issue?@action=redirect&bpo=46142].)
- Converting between **`int`** and **`str`** in bases other than 2 (binary), 4, 8 (octal), 16 (hexadecimal), or 32 such as base 10 (decimal) now raises a **`ValueError`** if the number of digits in string form is above a limit to avoid potential denial of service attacks due to the algorithmic complexity. This is a mitigation for **CVE 2020-10735** [https://www.cve.org/CVERecord?id=CVE-2020-10735]. This limit can be configured or disabled by environment variable, command line flag, or **`sys`** APIs. See the integer string conversion length limitation documentation. The default limit is 4300 digits in string form.

New Modules

- `tomllib`: For parsing TOML [https://toml.io/]. See **PEP 680** [https://peps.python.org/pep-0680/] for more details. (Contributed by Taneli Hukkinen in bpo-40059 [https://bugs.python.org/issue?@action=redirect&bpo=40059].)
- `wsgiref.types`: **WSGI** [https://peps.python.org/pep-3333/]-specific types for static type checking. (Contributed by Sebastian Rittau in bpo-42012 [https://bugs.python.org/issue?@action=redirect&bpo=42012].)

Improved Modules

asyncio

- Added the `TaskGroup` class, an asynchronous context manager holding a group of tasks that will wait for all of them upon exit. For new code this is recommended over using `create_task()` and `gather()` directly. (Contributed by Yury Selivanov and others in gh-90908 [https://github.com/python/cpython/issues/90908].)
- Added `timeout()`, an asynchronous context manager for setting a timeout on asynchronous operations. For new code this is recommended over using `wait_for()` directly. (Contributed by Andrew Svetlov in gh-90927 [https://github.com/python/cpython/issues/90927].)
- Added the `Runner` class, which exposes the machinery used by `run()`. (Contributed by Andrew Svetlov in gh-91218 [https://github.com/python/cpython/issues/91218].)
- Added the `Barrier` class to the synchronization primitives in the asyncio library, and the related `BrokenBarrierError` exception. (Contributed by Yves Duprat and Andrew Svetlov in gh-87518 [https://github.com/python/cpython/issues/87518].)
- Added keyword argument *all_errors* to `asyncio.loop.create_connection()` so that multiple connection errors can be raised as an `ExceptionGroup`.
- Added the `asyncio.StreamWriter.start_tls()` method for upgrading existing stream-based connections to TLS. (Contributed by Ian Good in bpo-34975 [https://bugs.python.org/issue?@action=redirect&bpo=34975].)
- Added raw datagram socket functions to the event loop: `sock_sendto()`, `sock_recvfrom()` and `sock_recvfrom_into()`. These have implementations in `SelectorEventLoop` and `ProactorEventLoop`. (Contributed by Alex Grönholm in bpo-46805 [https://bugs.python.org/issue?@action=redirect&bpo=46805].)

- Added `cancelling()` and `uncancel()` methods to `Task`. These are primarily intended for internal use, notably by `TaskGroup`.

contextlib

- Added non parallel-safe `chdir()` context manager to change the current working directory and then restore it on exit. Simple wrapper around `chdir()`. (Contributed by Filipe Laíns in bpo-25625 [https://bugs.python.org/issue?@action=redirect&bpo=25625])

dataclasses

- Change field default mutability check, allowing only defaults which are hashable instead of any object which is not an instance of `dict`, `list` or `set`. (Contributed by Eric V. Smith in bpo-44674 [https://bugs.python.org/issue?@action=redirect&bpo=44674].)

datetime

- Add `datetime.UTC`, a convenience alias for `datetime.timezone.utc`. (Contributed by Kabir Kwatra in gh-91973 [https://github.com/python/cpython/issues/91973].)
- `datetime.date.fromisoformat()`, `datetime.time.fromisoformat()` and `datetime.datetime.fromisoformat()` can now be used to parse most ISO 8601 formats (barring only those that support fractional hours and minutes). (Contributed by Paul Ganssle in gh-80010 [https://github.com/python/cpython/issues/80010].)

enum

- Renamed **`EnumMeta`** to `EnumType` (**`EnumMeta`** kept as an alias).
- Added `StrEnum`, with members that can be used as (and must be) strings.
- Added `ReprEnum`, which only modifies the `__repr__()` of members while returning their literal values (rather than names) for `__str__()` and `__format__()` (used by `str()`, `format()` and f-strings).
- Changed `Enum.__format__()` (the default for `format()`, `str.format()` and f-strings) to always produce the same result as `Enum.__str__()`: for enums inheriting from `ReprEnum` it will be the member's value; for all other enums it will be the enum and member name (e.g. `Color.RED`).
- Added a new *boundary* class parameter to `Flag` enums and the `FlagBoundary` enum with its options, to control how to handle out-of-

range flag values.
- Added the **`verify()`** enum decorator and the **`EnumCheck`** enum with its options, to check enum classes against several specific constraints.
- Added the **`member()`** and **`nonmember()`** decorators, to ensure the decorated object is/is not converted to an enum member.
- Added the **`property()`** decorator, which works like **`property()`** except for enums. Use this instead of **`types.DynamicClassAttribute()`**.
- Added the **`global_enum()`** enum decorator, which adjusts **`__repr__()`** and **`__str__()`** to show values as members of their module rather than the enum class. For example, `'re.ASCII'` for the **`ASCII`** member of **`re.RegexFlag`** rather than `'RegexFlag.ASCII'`.
- Enhanced **`Flag`** to support **`len()`**, iteration and **`in`**/**`not in`** on its members. For example, the following now works: `len(AFlag(3)) == 2 and list(AFlag(3)) == (AFlag.ONE, AFlag.TWO)`
- Changed **`Enum`** and **`Flag`** so that members are now defined before **`__init_subclass__()`** is called; **`dir()`** now includes methods, etc., from mixed-in data types.
- Changed **`Flag`** to only consider primary values (power of two) canonical while composite values (`3`, `6`, `10`, etc.) are considered aliases; inverted flags are coerced to their positive equivalent.

fcntl

- On FreeBSD, the **`F_DUP2FD`** and **`F_DUP2FD_CLOEXEC`** flags respectively are supported, the former equals to `dup2` usage while the latter set the `FD_CLOEXEC` flag in addition.

fractions

- Support **PEP 515** [https://peps.python.org/pep-0515/]-style initialization of **`Fraction`** from string. (Contributed by Sergey B Kirpichev in bpo-44258 [https://bugs.python.org/issue?@action=redirect&bpo=44258].)
- **`Fraction`** now implements an `__int__` method, so that an `isinstance(some_fraction, typing.SupportsInt)` check passes. (Contributed by Mark Dickinson in bpo-44547 [https://bugs.python.org/issue?@action=redirect&bpo=44547].)

functools

- **`functools.singledispatch()`** now supports **`types.UnionType`** and **`typing.Union`** as annotations to the dispatch argument.:

```
>>> from functools import singledispatch
>>> @singledispatch
... def fun(arg, verbose=False):
...     if verbose:
...         print("Let me just say,", end=" ")
...     print(arg)
...
>>> @fun.register
... def _(arg: int | float, verbose=False):
...     if verbose:
...         print("Strength in numbers, eh?", end=" ")
...     print(arg)
...
>>> from typing import Union
>>> @fun.register
... def _(arg: Union[list, set], verbose=False):
...     if verbose:
...         print("Enumerate this:")
...     for i, elem in enumerate(arg):
...         print(i, elem)
...
```

(Contributed by Yurii Karabas in bpo-46014 [https://bugs.python.org/issue?@action=redirect&bpo=46014].)

gzip

- The `gzip.compress()` function is now faster when used with the **mtime=0** argument as it delegates the compression entirely to a single `zlib.compress()` operation. There is one side effect of this change: The gzip file header contains an "OS" byte in its header. That was traditionally always set to a value of 255 representing "unknown" by the `gzip` module. Now, when using `compress()` with **mtime=0**, it may be set to a different value by the underlying zlib C library Python was linked against. (See gh-112346 [https://github.com/python/cpython/issues/112346] for details on the side effect.)

hashlib

- `hashlib.blake2b()` and `hashlib.blake2s()` now prefer libb2 [https://www.blake2.net/] over Python's vendored copy. (Contributed by Christian Heimes in bpo-47095 [https://bugs.python.org/issue?@action=redirect&bpo=47095].)

- The internal `_sha3` module with SHA3 and SHAKE algorithms now uses *tiny_sha3* instead of the *Keccak Code Package* to reduce code and binary size. The **`hashlib`** module prefers optimized SHA3 and SHAKE implementations from OpenSSL. The change affects only installations without OpenSSL support. (Contributed by Christian Heimes in bpo-47098 [https://bugs.python.org/issue?@action=redirect&bpo=47098].)
- Add **`hashlib.file_digest()`**, a helper function for efficient hashing of files or file-like objects. (Contributed by Christian Heimes in gh-89313 [https://github.com/python/cpython/issues/89313].)

IDLE and idlelib

- Apply syntax highlighting to `.pyi` files. (Contributed by Alex Waygood and Terry Jan Reedy in bpo-45447 [https://bugs.python.org/issue?@action=redirect&bpo=45447].)
- Include prompts when saving Shell with inputs and outputs. (Contributed by Terry Jan Reedy in gh-95191 [https://github.com/python/cpython/issues/95191].)

inspect

- Add **`getmembers_static()`** to return all members without triggering dynamic lookup via the descriptor protocol. (Contributed by Weipeng Hong in bpo-30533 [https://bugs.python.org/issue?@action=redirect&bpo=30533].)

- Add **`ismethodwrapper()`** for checking if the type of an object is a **`MethodWrapperType`**. (Contributed by Hakan Çelik in bpo-29418 [https://bugs.python.org/issue?@action=redirect&bpo=29418].)

- Change the frame-related functions in the **`inspect`** module to return new **`FrameInfo`** and **`Traceback`** class instances (backwards compatible with the previous named tuple-like interfaces) that includes the extended **PEP 657** [https://peps.python.org/pep-0657/] position information (end line number, column and end column). The affected functions are:

  - **`inspect.getframeinfo()`**
  - **`inspect.getouterframes()`**
  - **`inspect.getinnerframes()`**,
  - **`inspect.stack()`**

  - `inspect.trace()`

(Contributed by Pablo Galindo in gh-88116 [https://github.com/python/cpython/issues/88116].)

locale

- Add `locale.getencoding()` to get the current locale encoding. It is similar to `locale.getpreferredencoding(False)` but ignores the Python UTF-8 Mode.

logging

- Added `getLevelNamesMapping()` to return a mapping from logging level names (e.g. `'CRITICAL'`) to the values of their corresponding Logging Levels (e.g. `50`, by default). (Contributed by Andrei Kulakovin in gh-88024 [https://github.com/python/cpython/issues/88024].)
- Added a `createSocket()` method to `SysLogHandler`, to match `SocketHandler.createSocket()`. It is called automatically during handler initialization and when emitting an event, if there is no active socket. (Contributed by Kirill Pinchuk in gh-88457 [https://github.com/python/cpython/issues/88457].)

math

- Add `math.exp2()`: return 2 raised to the power of x. (Contributed by Gideon Mitchell in bpo-45917 [https://bugs.python.org/issue?@action=redirect&bpo=45917].)
- Add `math.cbrt()`: return the cube root of x. (Contributed by Ajith Ramachandran in bpo-44357 [https://bugs.python.org/issue?@action=redirect&bpo=44357].)
- The behaviour of two `math.pow()` corner cases was changed, for consistency with the IEEE 754 specification. The operations `math.pow(0.0, -math.inf)` and `math.pow(-0.0, -math.inf)` now return `inf`. Previously they raised `ValueError`. (Contributed by Mark Dickinson in bpo-44339 [https://bugs.python.org/issue?@action=redirect&bpo=44339].)
- The `math.nan` value is now always available. (Contributed by Victor Stinner in bpo-46917 [https://bugs.python.org/issue?@action=redirect&bpo=46917].)

operator

- A new function `operator.call` has been added, such that `operator.call(obj, *args, **kwargs) == obj(*args, **kwargs)`. (Contributed by Antony Lee in bpo-44019 [https://bugs.python.org/issue?@action=redirect&bpo=44019].)

os

- On Windows, **`os.urandom()`** now uses `BCryptGenRandom()`, instead of `CryptGenRandom()` which is deprecated. (Contributed by Donghee Na in bpo-44611 [https://bugs.python.org/issue?@action=redirect&bpo=44611].)

pathlib

- **`glob()`** and **`rglob()`** return only directories if *pattern* ends with a pathname components separator: **`sep`** or **`altsep`**. (Contributed by Eisuke Kawasima in bpo-22276 [https://bugs.python.org/issue?@action=redirect&bpo=22276] and bpo-33392 [https://bugs.python.org/issue?@action=redirect&bpo=33392].)

re

- Atomic grouping (`(?>...)`) and possessive quantifiers (`*+`, `++`, `?+`, `{m,n}+`) are now supported in regular expressions. (Contributed by Jeffrey C. Jacobs and Serhiy Storchaka in bpo-433030 [https://bugs.python.org/issue?@action=redirect&bpo=433030].)

shutil

- Add optional parameter *dir_fd* in **`shutil.rmtree()`**. (Contributed by Serhiy Storchaka in bpo-46245 [https://bugs.python.org/issue?@action=redirect&bpo=46245].)

socket

- Add CAN Socket support for NetBSD. (Contributed by Thomas Klausner in bpo-30512 [https://bugs.python.org/issue?@action=redirect&bpo=30512].)
- **`create_connection()`** has an option to raise, in case of failure to connect, an **`ExceptionGroup`** containing all errors instead of only raising the last error. (Contributed by Irit Katriel in bpo-29980 [https://bugs.python.org/issue?@action=redirect&bpo=29980].)

sqlite3

- You can now disable the authorizer by passing `None` to `set_authorizer()`. (Contributed by Erlend E. Aasland in bpo-44491 [https://bugs.python.org/issue?@action=redirect&bpo=44491].)
- Collation name `create_collation()` can now contain any Unicode character. Collation names with invalid characters now raise `UnicodeEncodeError` instead of `sqlite3.ProgrammingError`. (Contributed by Erlend E. Aasland in bpo-44688 [https://bugs.python.org/issue?@action=redirect&bpo=44688].)
- `sqlite3` exceptions now include the SQLite extended error code as `sqlite_errorcode` and the SQLite error name as `sqlite_errorname`. (Contributed by Aviv Palivoda, Daniel Shahaf, and Erlend E. Aasland in bpo-16379 [https://bugs.python.org/issue?@action=redirect&bpo=16379] and bpo-24139 [https://bugs.python.org/issue?@action=redirect&bpo=24139].)
- Add `setlimit()` and `getlimit()` to `sqlite3.Connection` for setting and getting SQLite limits by connection basis. (Contributed by Erlend E. Aasland in bpo-45243 [https://bugs.python.org/issue?@action=redirect&bpo=45243].)
- `sqlite3` now sets `sqlite3.threadsafety` based on the default threading mode the underlying SQLite library has been compiled with. (Contributed by Erlend E. Aasland in bpo-45613 [https://bugs.python.org/issue?@action=redirect&bpo=45613].)
- `sqlite3` C callbacks now use unraisable exceptions if callback tracebacks are enabled. Users can now register an `unraisable hook handler` to improve their debug experience. (Contributed by Erlend E. Aasland in bpo-45828 [https://bugs.python.org/issue?@action=redirect&bpo=45828].)
- Fetch across rollback no longer raises `InterfaceError`. Instead we leave it to the SQLite library to handle these cases. (Contributed by Erlend E. Aasland in bpo-44092 [https://bugs.python.org/issue?@action=redirect&bpo=44092].)
- Add `serialize()` and `deserialize()` to `sqlite3.Connection` for serializing and deserializing databases. (Contributed by Erlend E. Aasland in bpo-41930 [https://bugs.python.org/issue?@action=redirect&bpo=41930].)
- Add `create_window_function()` to `sqlite3.Connection` for creating aggregate window functions. (Contributed by Erlend E. Aasland in bpo-34916 [https://bugs.python.org/issue?@action=redirect&bpo=34916].)
- Add `blobopen()` to `sqlite3.Connection`. `sqlite3.Blob` allows incremental I/O operations on blobs. (Contributed by Aviv Palivoda and Erlend

E. Aasland in bpo-24905 [https://bugs.python.org/issue?@action=redirect&bpo=24905].)

string

- Add **`get_identifiers()`** and **`is_valid()`** to **`string.Template`**, which respectively return all valid placeholders, and whether any invalid placeholders are present. (Contributed by Ben Kehoe in gh-90465 [https://github.com/python/cpython/issues/90465].)

sys

- **`sys.exc_info()`** now derives the `type` and `traceback` fields from the `value` (the exception instance), so when an exception is modified while it is being handled, the changes are reflected in the results of subsequent calls to **`exc_info()`**. (Contributed by Irit Katriel in bpo-45711 [https://bugs.python.org/issue?@action=redirect&bpo=45711].)
- Add **`sys.exception()`** which returns the active exception instance (equivalent to `sys.exc_info()[1]`). (Contributed by Irit Katriel in bpo-46328 [https://bugs.python.org/issue?@action=redirect&bpo=46328].)
- Add the **`sys.flags.safe_path`** flag. (Contributed by Victor Stinner in gh-57684 [https://github.com/python/cpython/issues/57684].)

sysconfig

- Three new installation schemes (*posix_venv*, *nt_venv* and *venv*) were added and are used when Python creates new virtual environments or when it is running from a virtual environment. The first two schemes (*posix_venv* and *nt_venv*) are OS-specific for non-Windows and Windows, the *venv* is essentially an alias to one of them according to the OS Python runs on. This is useful for downstream distributors who modify **`sysconfig.get_preferred_scheme()`**. Third party code that creates new virtual environments should use the new *venv* installation scheme to determine the paths, as does **`venv`**. (Contributed by Miro Hrončok in bpo-45413 [https://bugs.python.org/issue?@action=redirect&bpo=45413].)

tempfile

- **`SpooledTemporaryFile`** objects now fully implement the methods of **`io.BufferedIOBase`** or **`io.TextIOBase`** (depending on file mode). This lets them work correctly with APIs that expect file-like objects, such as compression modules. (Contributed by Carey Metcalfe in gh-70363 [https://github.com/python/cpython/issues/70363].)

threading

- On Unix, if the `sem_clockwait()` function is available in the C library (glibc 2.30 and newer), the **`threading.Lock.acquire()`** method now uses the monotonic clock (**`time.CLOCK_MONOTONIC`**) for the timeout, rather than using the system clock (**`time.CLOCK_REALTIME`**), to not be affected by system clock changes. (Contributed by Victor Stinner in bpo-41710 [https://bugs.python.org/issue?@action=redirect&bpo=41710].)

time

- On Unix, **`time.sleep()`** now uses the `clock_nanosleep()` or `nanosleep()` function, if available, which has a resolution of 1 nanosecond (10^{-9} seconds), rather than using `select()` which has a resolution of 1 microsecond (10^{-6} seconds). (Contributed by Benjamin Szőke and Victor Stinner in bpo-21302 [https://bugs.python.org/issue?@action=redirect&bpo=21302].)
- On Windows 8.1 and newer, **`time.sleep()`** now uses a waitable timer based on high-resolution timers [https://docs.microsoft.com/en-us/windows-hardware/drivers/kernel/high-resolution-timers] which has a resolution of 100 nanoseconds (10^{-7} seconds). Previously, it had a resolution of 1 millisecond (10^{-3} seconds). (Contributed by Benjamin Szőke, Donghee Na, Eryk Sun and Victor Stinner in bpo-21302 [https://bugs.python.org/issue?@action=redirect&bpo=21302] and bpo-45429 [https://bugs.python.org/issue?@action=redirect&bpo=45429].)

tkinter

- Added method `info_patchlevel()` which returns the exact version of the Tcl library as a named tuple similar to **`sys.version_info`**. (Contributed by Serhiy Storchaka in gh-91827 [https://github.com/python/cpython/issues/91827].)

traceback

- Add **`traceback.StackSummary.format_frame_summary()`** to allow users to override which frames appear in the traceback, and how they are formatted. (Contributed by Ammar Askar in bpo-44569 [https://bugs.python.org/issue?@action=redirect&bpo=44569].)

- Add **`traceback.TracebackException.print()`**, which prints the formatted **`TracebackException`** instance to a file. (Contributed by Irit Katriel in bpo-33809 [https://bugs.python.org/issue?@action=redirect&bpo=33809].)

typing

For major changes, see New Features Related to Type Hints.

- Add **`typing.assert_never()`** and **`typing.Never`**. **`typing.assert_never()`** is useful for asking a type checker to confirm that a line of code is not reachable. At runtime, it raises an **`AssertionError`**. (Contributed by Jelle Zijlstra in gh-90633 [https://github.com/python/cpython/issues/90633].)
- Add **`typing.reveal_type()`**. This is useful for asking a type checker what type it has inferred for a given expression. At runtime it prints the type of the received value. (Contributed by Jelle Zijlstra in gh-90572 [https://github.com/python/cpython/issues/90572].)
- Add **`typing.assert_type()`**. This is useful for asking a type checker to confirm that the type it has inferred for a given expression matches the given type. At runtime it simply returns the received value. (Contributed by Jelle Zijlstra in gh-90638 [https://github.com/python/cpython/issues/90638].)
- **`typing.TypedDict`** types can now be generic. (Contributed by Samodya Abeysiriwardane in gh-89026 [https://github.com/python/cpython/issues/89026].)
- **`NamedTuple`** types can now be generic. (Contributed by Serhiy Storchaka in bpo-43923 [https://bugs.python.org/issue?@action=redirect&bpo=43923].)
- Allow subclassing of **`typing.Any`**. This is useful for avoiding type checker errors related to highly dynamic class, such as mocks. (Contributed by Shantanu Jain in gh-91154 [https://github.com/python/cpython/issues/91154].)
- The **`typing.final()`** decorator now sets the `__final__` attributed on the decorated object. (Contributed by Jelle Zijlstra in gh-90500 [https://github.com/python/cpython/issues/90500].)
- The **`typing.get_overloads()`** function can be used for introspecting the overloads of a function. **`typing.clear_overloads()`** can be used to clear all registered overloads of a function. (Contributed by Jelle Zijlstra in gh-89263 [https://github.com/python/cpython/issues/89263].)
- The **`__init__()`** method of **`Protocol`** subclasses is now preserved. (Contributed by Adrian Garcia Badarasco in gh-88970

[https://github.com/python/cpython/issues/88970].)

- The representation of empty tuple types (`Tuple[()]`) is simplified. This affects introspection, e.g. `get_args(Tuple[()])` now evaluates to `()` instead of `((),)`. (Contributed by Serhiy Storchaka in gh-91137 [https://github.com/python/cpython/issues/91137].)
- Loosen runtime requirements for type annotations by removing the callable check in the private `typing._type_check` function. (Contributed by Gregory Beauregard in gh-90802 [https://github.com/python/cpython/issues/90802].)
- **`typing.get_type_hints()`** now supports evaluating strings as forward references in PEP 585 generic aliases. (Contributed by Niklas Rosenstein in gh-85542 [https://github.com/python/cpython/issues/85542].)
- **`typing.get_type_hints()`** no longer adds **`Optional`** to parameters with `None` as a default. (Contributed by Nikita Sobolev in gh-90353 [https://github.com/python/cpython/issues/90353].)
- **`typing.get_type_hints()`** now supports evaluating bare stringified **`ClassVar`** annotations. (Contributed by Gregory Beauregard in gh-90711 [https://github.com/python/cpython/issues/90711].)
- **`typing.no_type_check()`** no longer modifies external classes and functions. It also now correctly marks classmethods as not to be type checked. (Contributed by Nikita Sobolev in gh-90729 [https://github.com/python/cpython/issues/90729].)

unicodedata

- The Unicode database has been updated to version 14.0.0. (Contributed by Benjamin Peterson in bpo-45190 [https://bugs.python.org/issue?@action=redirect&bpo=45190]).

unittest

- Added methods **`enterContext()`** and **`enterClassContext()`** of class **`TestCase`**, method **`enterAsyncContext()`** of class **`IsolatedAsyncioTestCase`** and function **`unittest.enterModuleContext()`**. (Contributed by Serhiy Storchaka in bpo-45046 [https://bugs.python.org/issue?@action=redirect&bpo=45046].)

venv

- When new Python virtual environments are created, the *venv* sysconfig installation scheme is used to determine the paths inside the environment. When Python runs in a virtual environment, the same installation scheme is the default. That means that downstream distributors can change the default sysconfig install scheme without changing behavior of virtual environments. Third party code that also creates new virtual environments should do the same. (Contributed by Miro Hrončok in bpo-45413 [https://bugs.python.org/issue?@action=redirect&bpo=45413].)

warnings

- `warnings.catch_warnings()` now accepts arguments for `warnings.simplefilter()`, providing a more concise way to locally ignore warnings or convert them to errors. (Contributed by Zac Hatfield-Dodds in bpo-47074 [https://bugs.python.org/issue?@action=redirect&bpo=47074].)

zipfile

- Added support for specifying member name encoding for reading metadata in a `ZipFile`'s directory and file headers. (Contributed by Stephen J. Turnbull and Serhiy Storchaka in bpo-28080 [https://bugs.python.org/issue?@action=redirect&bpo=28080].)
- Added `ZipFile.mkdir()` for creating new directories inside ZIP archives. (Contributed by Sam Ezeh in gh-49083 [https://github.com/python/cpython/issues/49083].)
- Added `stem`, `suffix` and `suffixes` to `zipfile.Path`. (Contributed by Miguel Brito in gh-88261 [https://github.com/python/cpython/issues/88261].)

Optimizations

This section covers specific optimizations independent of the Faster CPython project, which is covered in its own section.

- The compiler now optimizes simple printf-style % formatting on string literals containing only the format codes `%s`, `%r` and `%a` and makes it as fast as a corresponding f-string expression. (Contributed by Serhiy Storchaka in bpo-28307 [https://bugs.python.org/issue?@action=redirect&bpo=28307].)
- Integer division (`//`) is better tuned for optimization by compilers. It is now around 20% faster on x86-64 when dividing an `int` by a value smaller than `2**30`. (Contributed by Gregory P. Smith and Tim Peters in gh-90564 [https://github.com/python/cpython/issues/90564].)

- **`sum()`** is now nearly 30% faster for integers smaller than `2**30`. (Contributed by Stefan Behnel in gh-68264 [https://github.com/python/cpython/issues/68264].)
- Resizing lists is streamlined for the common case, speeding up **`list.append()`** by ≈15% and simple list comprehensions by up to 20-30% (Contributed by Dennis Sweeney in gh-91165 [https://github.com/python/cpython/issues/91165].)
- Dictionaries don't store hash values when all keys are Unicode objects, decreasing **`dict`** size. For example, `sys.getsizeof(dict.fromkeys("abcdefg"))` is reduced from 352 bytes to 272 bytes (23% smaller) on 64-bit platforms. (Contributed by Inada Naoki in bpo-46845 [https://bugs.python.org/issue?@action=redirect&bpo=46845].)
- Using **`asyncio.DatagramProtocol`** is now orders of magnitude faster when transferring large files over UDP, with speeds over 100 times higher for a ≈60 MiB file. (Contributed by msoxzw in gh-91487 [https://github.com/python/cpython/issues/91487].)
- **`math`** functions **`comb()`** and **`perm()`** are now ≈10 times faster for large arguments (with a larger speedup for larger *k*). (Contributed by Serhiy Storchaka in bpo-37295 [https://bugs.python.org/issue?@action=redirect&bpo=37295].)
- The **`statistics`** functions **`mean()`**, **`variance()`** and **`stdev()`** now consume iterators in one pass rather than converting them to a **`list`** first. This is twice as fast and can save substantial memory. (Contributed by Raymond Hettinger in gh-90415 [https://github.com/python/cpython/issues/90415].)
- **`unicodedata.normalize()`** now normalizes pure-ASCII strings in constant time. (Contributed by Donghee Na in bpo-44987 [https://bugs.python.org/issue?@action=redirect&bpo=44987].)

Faster CPython

CPython 3.11 is an average of 25% faster [https://github.com/faster-cpython/ideas#published-results] than CPython 3.10 as measured with the pyperformance [https://github.com/python/pyperformance] benchmark suite, when compiled with GCC on Ubuntu Linux. Depending on your workload, the overall speedup could be 10-60%.

This project focuses on two major areas in Python: Faster Startup and Faster Runtime. Optimizations not covered by this project are listed separately under Optimizations.

Faster Startup

Frozen imports / Static code objects

Python caches bytecode in the __pycache__ directory to speed up module loading.

Previously in 3.10, Python module execution looked like this:

```
Read __pycache__ -> Unmarshal -> Heap allocated code object -> Evaluate
```

In Python 3.11, the core modules essential for Python startup are “frozen”. This means that their Code Objects (and bytecode) are statically allocated by the interpreter. This reduces the steps in module execution process to:

```
Statically allocated code object -> Evaluate
```

Interpreter startup is now 10-15% faster in Python 3.11. This has a big impact for short-running programs using Python.

(Contributed by Eric Snow, Guido van Rossum and Kumar Aditya in many issues.)

Faster Runtime

Cheaper, lazy Python frames

Python frames, holding execution information, are created whenever Python calls a Python function. The following are new frame optimizations:

- Streamlined the frame creation process.
- Avoided memory allocation by generously re-using frame space on the C stack.
- Streamlined the internal frame struct to contain only essential information. Frames previously held extra debugging and memory management information.

Old-style frame objects are now created only when requested by debuggers or by Python introspection functions such as `sys._getframe()` and `inspect.currentframe()`. For most user code, no frame objects are created at all. As a result, nearly all Python functions calls have sped up significantly. We measured a 3-7% speedup in pyperformance.

(Contributed by Mark Shannon in bpo-44590 [https://bugs.python.org/issue?@action=redirect&bpo=44590].)

Inlined Python function calls

During a Python function call, Python will call an evaluating C function to interpret that function’s code. This effectively limits pure Python recursion to what’s safe for the C stack.

In 3.11, when CPython detects Python code calling another Python function, it sets up a new frame, and “jumps” to the new code inside the new frame. This avoids calling the C interpreting function altogether.

Most Python function calls now consume no C stack space, speeding them up. In simple recursive functions like fibonacci or factorial, we observed a 1.7x speedup. This also means recursive functions can recurse significantly deeper (if the user increases the recursion limit with `sys.setrecursionlimit()`). We measured a 1-3% improvement in pyperformance.

(Contributed by Pablo Galindo and Mark Shannon in bpo-45256 [https://bugs.python.org/issue?@action=redirect&bpo=45256].)

PEP 659: Specializing Adaptive Interpreter

PEP 659 [https://peps.python.org/pep-0659/] is one of the key parts of the Faster CPython project. The general idea is that while Python is a dynamic language, most code has regions where objects and types rarely change. This concept is known as *type stability*.

At runtime, Python will try to look for common patterns and type stability in the executing code. Python will then replace the current operation with a more specialized one. This specialized operation uses fast paths available only to those use cases/types, which generally outperform their generic counterparts. This also brings in another concept called *inline caching*, where Python caches the results of expensive operations directly in the bytecode.

The specializer will also combine certain common instruction pairs into one superinstruction, reducing the overhead during execution.

Python will only specialize when it sees code that is “hot” (executed multiple times). This prevents Python from wasting time on run-once code. Python can also de-specialize when code is too dynamic or when the use changes. Specialization is attempted periodically, and specialization attempts are not too expensive, allowing specialization to adapt to new circumstances.

(PEP written by Mark Shannon, with ideas inspired by Stefan Brunthaler. See **PEP 659** [https://peps.python.org/pep-0659/] for more information. Implementation by Mark Shannon and Brandt Bucher, with additional help from Irit Katriel and Dennis Sweeney.)

Operation	Form	Specialization	Operation speedup (up to)	Contributor(s)

Binary operations	`x + x`
	`x - x`
	`x * x`
		Binary add, multiply and subtract for common types such as `int`, `float` and `str` take custom fast paths for their underlying types.	10%	Mark Shannon, Donghee Na, Brandt Bucher, Dennis Sweeney

Subscript	`a[i]`	Subscripting container types such as `list`, `tuple` and `dict` directly index the underlying data structures.	10-25%	Irit Katriel, Mark Shannon

Operation	Form	Specialization	Operation speedup (up to)	Contributor(s)
		Subscripting custom `__getitem__()` is also inlined similar to Inlined Python function calls.		
Store subscript	`a[i] = z`	Similar to subscripting specialization above.	10-25%	Dennis Sweeney
Calls	`f(arg)` `C(arg)`	Calls to common builtin (C) functions and types such as `len()` and `str` directly call their underlying C version. This avoids going through the internal calling convention.	20%	Mark Shannon, Ken Jin
Load global variable	`print` `len`	The object's index in the globals/builtins namespace is cached. Loading globals and builtins require zero	[[1]]	Mark Shannon

Operation	Form	Specialization	Operation speedup (up to)	Contributor(s)
		namespace lookups.		
Load attribute	`o.attr`	Similar to loading global variables. The attribute's index inside the class/object's namespace is cached. In most cases, attribute loading will require zero namespace lookups.	[[2]]	Mark Shannon
Load methods for call	`o.meth()`	The actual address of the method is cached. Method loading now has no namespace lookups – even for classes with long inheritance chains.	10-20%	Ken Jin, Mark Shannon
Store attribute	`o.attr = z`	Similar to load attribute optimization.	2% in pyperformance	Mark Shannon

Operation	Form	Specialization	Operation speedup (up to)	Contributor(s)
Unpack Sequence	`*seq`	Specialized for common containers such as `list` and `tuple`. Avoids internal calling convention.	8%	Brandt Bucher

[1]

A similar optimization already existed since Python 3.8. 3.11 specializes for more forms and reduces some overhead.

[2]

A similar optimization already existed since Python 3.10. 3.11 specializes for more forms. Furthermore, all attribute loads should be sped up by bpo-45947 [https://bugs.python.org/issue?@action=redirect&bpo=45947].

Misc

- Objects now require less memory due to lazily created object namespaces. Their namespace dictionaries now also share keys more freely. (Contributed Mark Shannon in bpo-45340 [https://bugs.python.org/issue?@action=redirect&bpo=45340] and bpo-40116 [https://bugs.python.org/issue?@action=redirect&bpo=40116].)
- “Zero-cost” exceptions are implemented, eliminating the cost of `try` statements when no exception is raised. (Contributed by Mark Shannon in bpo-40222 [https://bugs.python.org/issue?@action=redirect&bpo=40222].)
- A more concise representation of exceptions in the interpreter reduced the time required for catching an exception by about 10%. (Contributed by Irit Katriel in bpo-45711 [https://bugs.python.org/issue?@action=redirect&bpo=45711].)

- **re**’s regular expression matching engine has been partially refactored, and now uses computed gotos (or “threaded code”) on supported platforms. As a result, Python 3.11 executes the pyperformance regular expression benchmarks [https://pyperformance.readthedocs.io/benchmarks.html#regex-dna] up to 10% faster than Python 3.10. (Contributed by Brandt Bucher in gh-91404 [https://github.com/python/cpython/issues/91404].)

FAQ

How should I write my code to utilize these speedups?

Write Pythonic code that follows common best practices; you don’t have to change your code. The Faster CPython project optimizes for common code patterns we observe.

Will CPython 3.11 use more memory?

Maybe not; we don’t expect memory use to exceed 20% higher than 3.10. This is offset by memory optimizations for frame objects and object dictionaries as mentioned above.

I don’t see any speedups in my workload. Why?

Certain code won’t have noticeable benefits. If your code spends most of its time on I/O operations, or already does most of its computation in a C extension library like NumPy, there won’t be significant speedups. This project currently benefits pure-Python workloads the most.

Furthermore, the pyperformance figures are a geometric mean. Even within the pyperformance benchmarks, certain benchmarks have slowed down slightly, while others have sped up by nearly 2x!

Is there a JIT compiler?

No. We’re still exploring other optimizations.

About

Faster CPython explores optimizations for CPython. The main team is funded by Microsoft to work on this full-time. Pablo Galindo Salgado is also funded by Bloomberg LP to work on the project part-time. Finally, many contributors are volunteers from the community.

CPython bytecode changes

The bytecode now contains inline cache entries, which take the form of the newly-added `CACHE` instructions. Many opcodes expect to be followed by an exact number of caches, and instruct the interpreter to skip over them at runtime. Populated caches can look like arbitrary instructions, so great care should be taken when reading or modifying raw, adaptive bytecode containing quickened data.

New opcodes

- `ASYNC_GEN_WRAP`, `RETURN_GENERATOR` and `SEND`, used in generators and co-routines.
- `COPY_FREE_VARS`, which avoids needing special caller-side code for closures.
- `JUMP_BACKWARD_NO_INTERRUPT`, for use in certain loops where handling interrupts is undesirable.
- `MAKE_CELL`, to create Cell Objects.
- `CHECK_EG_MATCH` and `PREP_RERAISE_STAR`, to handle the new exception groups and except* added in **PEP 654** [https://peps.python.org/pep-0654/].
- `PUSH_EXC_INFO`, for use in exception handlers.
- `RESUME`, a no-op, for internal tracing, debugging and optimization checks.

Replaced opcodes

Replaced Opcode(s)	New Opcode(s)	Notes
`BINARY_*` `INPLACE_*`	`BINARY_OP`	Replaced all numeric binary/in-place opcodes with a single opcode

`CALL_FUNCTION`
`CALL_FUNCTION_KW`
`CALL_METHOD`
	`CALL`
	`KW_NAMES`
	`PRECALL`
		Decouples argument shifting for methods from handling of keyword

Replaced Opcode(s)	New Opcode(s)	Notes
	`PUSH_NULL`	arguments; allows better specialization of calls

`DUP_TOP`
`DUP_TOP_TWO`
`ROT_TWO`
`ROT_THREE`
`ROT_FOUR`
`ROT_N`
	`COPY`
	`SWAP`
		Stack manipulation instructions

`JUMP_IF_NOT_EXC_MATCH`	`CHECK_EXC_MATCH`	Now performs check but doesn’t jump

`JUMP_ABSOLUTE`
`POP_JUMP_IF_FALSE`
`POP_JUMP_IF_TRUE`
	`JUMP_BACKWARD`
	`POP_JUMP_BACKWARD_IF_*`
	`POP_JUMP_FORWARD_IF_*`
		See [[3]]; `TRUE`, `FALSE`, `NONE` and `NOT_NONE` variants for each direction

`SETUP_WITH` `SETUP_ASYNC_WITH`	`BEFORE_WITH`	`with` block setup

[3]

All jump opcodes are now relative, including the existing `JUMP_IF_TRUE_OR_POP` and `JUMP_IF_FALSE_OR_POP`. The argument is now an offset from the current instruction rather than an absolute location.

Changed/removed opcodes

- Changed `MATCH_CLASS` and `MATCH_KEYS` to no longer push an additional boolean value to indicate success/failure. Instead, `None` is pushed on failure in place of the tuple of extracted values.
- Changed opcodes that work with exceptions to reflect them now being represented as one item on the stack instead of three (see gh-89874 [https://github.com/python/cpython/issues/89874]).

- Removed **`COPY_DICT_WITHOUT_KEYS`**, **`GEN_START`**, **`POP_BLOCK`**, **`SETUP_FINALLY`** and **`YIELD_FROM`**.

Deprecated

This section lists Python APIs that have been deprecated in Python 3.11.

Deprecated C APIs are listed separately.

Language/Builtins

- Chaining **`classmethod`** descriptors (introduced in bpo-19072 [https://bugs.python.org/issue?@action=redirect&bpo=19072]) is now deprecated. It can no longer be used to wrap other descriptors such as **`property`**. The core design of this feature was flawed and caused a number of downstream problems. To “pass-through” a **`classmethod`**, consider using the **`__wrapped__`** attribute that was added in Python 3.10. (Contributed by Raymond Hettinger in gh-89519 [https://github.com/python/cpython/issues/89519].)
- Octal escapes in string and bytes literals with values larger than `0o377` (255 in decimal) now produce a **`DeprecationWarning`**. In a future Python version, they will raise a **`SyntaxWarning`** and eventually a **`SyntaxError`**. (Contributed by Serhiy Storchaka in gh-81548 [https://github.com/python/cpython/issues/81548].)
- The delegation of **`int()`** to **`__trunc__()`** is now deprecated. Calling `int(a)` when `type(a)` implements **`__trunc__()`** but not **`__int__()`** or **`__index__()`** now raises a **`DeprecationWarning`**. (Contributed by Zackery Spytz in bpo-44977 [https://bugs.python.org/issue?@action=redirect&bpo=44977].)

Modules

- **PEP 594** [https://peps.python.org/pep-0594/] led to the deprecations of the following modules slated for removal in Python 3.13:

aifc	**chunk**	**msilib**	**pipes**	**telnetlib**
audioop	**crypt**	**nis**	**sndhdr**	**uu**

`cgi`	`imghdr`	`nntplib`	`spwd`	`xdrlib`
`cgitb`	`mailcap`	`ossaudiodev`	`sunau`	

(Contributed by Brett Cannon in bpo-47061 [https://bugs.python.org/issue?@action=redirect&bpo=47061] and Victor Stinner in gh-68966 [https://github.com/python/cpython/issues/68966].)

- The `asynchat`, `asyncore` and `smtpd` modules have been deprecated since at least Python 3.6. Their documentation and deprecation warnings have now been updated to note they will be removed in Python 3.12. (Contributed by Hugo van Kemenade in bpo-47022 [https://bugs.python.org/issue?@action=redirect&bpo=47022].)
- The `lib2to3` package and `2to3` tool are now deprecated and may not be able to parse Python 3.10 or newer. See **PEP 617** [https://peps.python.org/pep-0617/], introducing the new PEG parser, for details. (Contributed by Victor Stinner in bpo-40360 [https://bugs.python.org/issue?@action=redirect&bpo=40360].)
- Undocumented modules `sre_compile`, `sre_constants` and `sre_parse` are now deprecated. (Contributed by Serhiy Storchaka in bpo-47152 [https://bugs.python.org/issue?@action=redirect&bpo=47152].)

Standard Library

- The following have been deprecated in `configparser` since Python 3.2. Their deprecation warnings have now been updated to note they will be removed in Python 3.12:
  - the `configparser.SafeConfigParser` class
  - the `configparser.ParsingError.filename` property
  - the `configparser.RawConfigParser.readfp()` method

  (Contributed by Hugo van Kemenade in bpo-45173 [https://bugs.python.org/issue?@action=redirect&bpo=45173].)
- `configparser.LegacyInterpolation` has been deprecated in the docstring since Python 3.2, and is not listed in the `configparser` documentation. It now emits a `DeprecationWarning` and will be

removed in Python 3.13. Use `configparser.BasicInterpolation` or `configparser.ExtendedInterpolation` instead. (Contributed by Hugo van Kemenade in bpo-46607 [https://bugs.python.org/issue?@action=redirect&bpo=46607].)

- The older set of `importlib.resources` functions were deprecated in favor of the replacements added in Python 3.9 and will be removed in a future Python version, due to not supporting resources located within package subdirectories:
    - **`importlib.resources.contents()`**
    - **`importlib.resources.is_resource()`**
    - **`importlib.resources.open_binary()`**
    - **`importlib.resources.open_text()`**
    - **`importlib.resources.read_binary()`**
    - **`importlib.resources.read_text()`**
    - **`importlib.resources.path()`**
- The `locale.getdefaultlocale()` function is deprecated and will be removed in Python 3.15. Use `locale.setlocale()`, `locale.getpreferredencoding(False)` and `locale.getlocale()` functions instead. (Contributed by Victor Stinner in gh-90817 [https://github.com/python/cpython/issues/90817].)
- The **`locale.resetlocale()`** function is deprecated and will be removed in Python 3.13. Use `locale.setlocale(locale.LC_ALL, "")` instead. (Contributed by Victor Stinner in gh-90817 [https://github.com/python/cpython/issues/90817].)
- Stricter rules will now be applied for numerical group references and group names in regular expressions. Only sequences of ASCII digits will now be accepted as a numerical reference, and the group name in `bytes` patterns and replacement strings can only contain ASCII letters, digits and underscores. For now, a deprecation warning is raised for syntax violating these rules. (Contributed by Serhiy Storchaka in gh-91760 [https://github.com/python/cpython/issues/91760].)
- In the `re` module, the **`re.template()`** function and the corresponding **`re.TEMPLATE`** and **`re.T`** flags are deprecated, as they were undocumented and lacked an obvious purpose. They will be removed in Python 3.13. (Contributed by Serhiy Storchaka and Miro Hrončok in gh-92728 [https://github.com/python/cpython/issues/92728].)

- **`turtle.settiltangle()`** has been deprecated since Python 3.1; it now emits a deprecation warning and will be removed in Python 3.13. Use `turtle.tiltangle()` instead (it was earlier incorrectly marked as deprecated, and its docstring is now corrected). (Contributed by Hugo van Kemenade in bpo-45837 [https://bugs.python.org/issue?@action=redirect&bpo=45837].)

- `typing.Text`, which exists solely to provide compatibility support between Python 2 and Python 3 code, is now deprecated. Its removal is currently unplanned, but users are encouraged to use `str` instead wherever possible. (Contributed by Alex Waygood in gh-92332 [https://github.com/python/cpython/issues/92332].)

- The keyword argument syntax for constructing `typing.TypedDict` types is now deprecated. Support will be removed in Python 3.13. (Contributed by Jingchen Ye in gh-90224 [https://github.com/python/cpython/issues/90224].)

- **`webbrowser.MacOSX`** is deprecated and will be removed in Python 3.13. It is untested, undocumented, and not used by `webbrowser` itself. (Contributed by Donghee Na in bpo-42255 [https://bugs.python.org/issue?@action=redirect&bpo=42255].)

- The behavior of returning a value from a `TestCase` and `IsolatedAsyncioTestCase` test methods (other than the default `None` value) is now deprecated.

- Deprecated the following not-formally-documented `unittest` functions, scheduled for removal in Python 3.13:

  - **`unittest.findTestCases()`**
  - **`unittest.makeSuite()`**
  - **`unittest.getTestCaseNames()`**

  Use `TestLoader` methods instead:

  - `unittest.TestLoader.loadTestsFromModule()`
  - `unittest.TestLoader.loadTestsFromTestCase()`
  - `unittest.TestLoader.getTestCaseNames()`

  (Contributed by Erlend E. Aasland in bpo-5846 [https://bugs.python.org/issue?@action=redirect&bpo=5846].)

- `unittest.TestProgram.usageExit()` is marked deprecated, to be removed in 3.13. (Contributed by Carlos Damázio in gh-67048 [https://github.com/python/cpython/issues/67048].)

Pending Removal in Python 3.12

The following Python APIs have been deprecated in earlier Python releases, and will be removed in Python 3.12.

C APIs pending removal are listed separately.

- The `asynchat` module
- The `asyncore` module
- The entire distutils package
- The `imp` module
- The `typing.io` namespace
- The `typing.re` namespace
- `cgi.log()`
- `importlib.find_loader()`
- `importlib.abc.Loader.module_repr()`
- `importlib.abc.MetaPathFinder.find_module()`
- `importlib.abc.PathEntryFinder.find_loader()`
- `importlib.abc.PathEntryFinder.find_module()`
- `importlib.machinery.BuiltinImporter.find_module()`
- `importlib.machinery.BuiltinLoader.module_repr()`
- `importlib.machinery.FileFinder.find_loader()`
- `importlib.machinery.FileFinder.find_module()`
- `importlib.machinery.FrozenImporter.find_module()`

- **`importlib.machinery.FrozenLoader.module_repr()`**
- **`importlib.machinery.PathFinder.find_module()`**
- **`importlib.machinery.WindowsRegistryFinder.find_module()`**
- **`importlib.util.module_for_loader()`**
- **`importlib.util.set_loader_wrapper()`**
- **`importlib.util.set_package_wrapper()`**
- **`pkgutil.ImpImporter`**
- **`pkgutil.ImpLoader`**
- **`pathlib.Path.link_to()`**
- **`sqlite3.enable_shared_cache()`**
- **`sqlite3.OptimizedUnicode()`**
- **`PYTHONTHREADDEBUG`** environment variable
- The following deprecated aliases in `unittest`:

Deprecated alias	Method Name	Deprecated in
`failUnless`	`assertTrue()`	3.1
`failIf`	`assertFalse()`	3.1
`failUnlessEqual`	`assertEqual()`	3.1
`failIfEqual`	`assertNotEqual()`	3.1
`failUnlessAlmostEqual`	`assertAlmostEqual()`	3.1

Deprecated alias	Method Name	Deprecated in
`failIfAlmostEqual`	`assertNotAlmostEqual()`	3.1
`failUnlessRaises`	`assertRaises()`	3.1
`assert_`	`assertTrue()`	3.2
`assertEquals`	`assertEqual()`	3.2
`assertNotEquals`	`assertNotEqual()`	3.2
`assertAlmostEquals`	`assertAlmostEqual()`	3.2
`assertNotAlmostEquals`	`assertNotAlmostEqual()`	3.2
`assertRegexpMatches`	`assertRegex()`	3.2
`assertRaisesRegexp`	`assertRaisesRegex()`	3.2
`assertNotRegexpMatches`	`assertNotRegex()`	3.5

Removed

This section lists Python APIs that have been removed in Python 3.11.

Removed C APIs are listed separately.

- Removed the **`@asyncio.coroutine()`** decorator enabling legacy generator-based coroutines to be compatible with `async` / `await` code. The function has been deprecated since Python 3.8 and the removal was initially scheduled for Python 3.10. Use `async def` instead. (Contributed by Illia Volochii in bpo-43216 [https://bugs.python.org/issue?@action=redirect&bpo=43216].)

- Removed **`asyncio.coroutines.CoroWrapper`** used for wrapping legacy generator-based coroutine objects in the debug mode. (Contributed by Illia Volochii in bpo-43216 [https://bugs.python.org/issue?@action=redirect&bpo=43216].)
- Due to significant security concerns, the *reuse_address* parameter of **`asyncio.loop.create_datagram_endpoint()`**, disabled in Python 3.9, is now entirely removed. This is because of the behavior of the socket option `SO_REUSEADDR` in UDP. (Contributed by Hugo van Kemenade in bpo-45129 [https://bugs.python.org/issue?@action=redirect&bpo=45129].)
- Removed the **`binhex`** module, deprecated in Python 3.9. Also removed the related, similarly-deprecated **`binascii`** functions:
    - **`binascii.a2b_hqx()`**
    - **`binascii.b2a_hqx()`**
    - **`binascii.rlecode_hqx()`**
    - **`binascii.rldecode_hqx()`**

  The **`binascii.crc_hqx()`** function remains available.

  (Contributed by Victor Stinner in bpo-45085 [https://bugs.python.org/issue?@action=redirect&bpo=45085].)
- Removed the **`distutils`** `bdist_msi` command deprecated in Python 3.9. Use `bdist_wheel` (wheel packages) instead. (Contributed by Hugo van Kemenade in bpo-45124 [https://bugs.python.org/issue?@action=redirect&bpo=45124].)
- Removed the **`__getitem__()`** methods of **`xml.dom.pulldom.DOMEventStream`**, **`wsgiref.util.FileWrapper`** and **`fileinput.FileInput`**, deprecated since Python 3.9. (Contributed by Hugo van Kemenade in bpo-45132 [https://bugs.python.org/issue?@action=redirect&bpo=45132].)
- Removed the deprecated **`gettext`** functions **`lgettext()`**, **`ldgettext()`**, **`lngettext()`** and **`ldngettext()`**. Also removed the **`bind_textdomain_codeset()`** function, the **`NullTranslations.output_charset()`** and **`NullTranslations.set_output_charset()`** methods, and the *codeset*

parameter of **`translation()`** and **`install()`**, since they are only used for the **`l*gettext()`** functions. (Contributed by Donghee Na and Serhiy Storchaka in bpo-44235 [https://bugs.python.org/issue?@action=redirect&bpo=44235].)

- Removed from the `inspect` module:

    - The **`getargspec()`** function, deprecated since Python 3.0; use `inspect.signature()` or `inspect.getfullargspec()` instead.
    - The **`formatargspec()`** function, deprecated since Python 3.5; use the `inspect.signature()` function or the `inspect.Signature` object directly.
    - The undocumented **`Signature.from_builtin()`** and **`Signature.from_function()`** methods, deprecated since Python 3.5; use the `Signature.from_callable()` method instead.

    (Contributed by Hugo van Kemenade in bpo-45320 [https://bugs.python.org/issue?@action=redirect&bpo=45320].)

- Removed the `__class_getitem__()` method from `pathlib.PurePath`, because it was not used and added by mistake in previous versions. (Contributed by Nikita Sobolev in bpo-46483 [https://bugs.python.org/issue?@action=redirect&bpo=46483].)

- Removed the **`MailmanProxy`** class in the **`smtpd`** module, as it is unusable without the external **`mailman`** package. (Contributed by Donghee Na in bpo-35800 [https://bugs.python.org/issue?@action=redirect&bpo=35800].)

- Removed the deprecated **`split()`** method of **`_tkinter.TkappType`**. (Contributed by Erlend E. Aasland in bpo-38371 [https://bugs.python.org/issue?@action=redirect&bpo=38371].)

- Removed namespace package support from `unittest` discovery. It was introduced in Python 3.4 but has been broken since Python 3.7. (Contributed by Inada Naoki in bpo-23882 [https://bugs.python.org/issue?@action=redirect&bpo=23882].)

- Removed the undocumented private **`float.__set_format__()`** method, previously known as **`float.__setformat__()`** in Python 3.7. Its docstring said: “You probably don’t want to use this function. It exists mainly to be used in Python’s test suite.” (Contributed by

Victor Stinner in bpo-46852 [https://bugs.python.org/issue?@action=redirect&bpo=46852].)

- The `--experimental-isolated-subinterpreters` configure flag (and corresponding `EXPERIMENTAL_ISOLATED_SUBINTERPRETERS` macro) have been removed.

- Pynche [https://pypi.org/project/Pynche/] — The Pythonically Natural Color and Hue Editor — has been moved out of `Tools/scripts` and is being developed independently [https://gitlab.com/warsaw/pynche/-/tree/main] from the Python source tree.

Porting to Python 3.11

This section lists previously described changes and other bugfixes in the Python API that may require changes to your Python code.

Porting notes for the C API are listed separately.

- `open()`, `io.open()`, `codecs.open()` and `fileinput.FileInput` no longer accept `'U'` ("universal newline") in the file mode. In Python 3, "universal newline" mode is used by default whenever a file is opened in text mode, and the `'U'` flag has been deprecated since Python 3.3. The newline parameter to these functions controls how universal newlines work. (Contributed by Victor Stinner in bpo-37330 [https://bugs.python.org/issue?@action=redirect&bpo=37330].)
- `ast.AST` node positions are now validated when provided to `compile()` and other related functions. If invalid positions are detected, a `ValueError` will be raised. (Contributed by Pablo Galindo in gh-93351 [https://github.com/python/cpython/issues/93351])
- Prohibited passing non-`concurrent.futures.ThreadPoolExecutor` executors to `asyncio.loop.set_default_executor()` following a deprecation in Python 3.8. (Contributed by Illia Volochii in bpo-43234 [https://bugs.python.org/issue?@action=redirect&bpo=43234].)
- `calendar`: The `calendar.LocaleTextCalendar` and `calendar.LocaleHTMLCalendar` classes now use `locale.getlocale()`, instead of using `locale.getdefaultlocale()`, if no locale is specified. (Contributed by Victor Stinner in bpo-46659 [https://bugs.python.org/issue?@action=redirect&bpo=46659].)
- The `pdb` module now reads the `.pdbrc` configuration file with the `'UTF-8'` encoding. (Contributed by Srinivas Reddy Thatiparthy

(శ్రీనివాస్ రెడ్డి తాటిపర్తి) in bpo-41137 [https://bugs.python.org/issue?@action=redirect&bpo=41137].)

- The *population* parameter of `random.sample()` must be a sequence, and automatic conversion of `set`s to `list`s is no longer supported. Also, if the sample size is larger than the population size, a `ValueError` is raised. (Contributed by Raymond Hettinger in bpo-40465 [https://bugs.python.org/issue?@action=redirect&bpo=40465].)
- The *random* optional parameter of `random.shuffle()` was removed. It was previously an arbitrary random function to use for the shuffle; now, `random.random()` (its previous default) will always be used.
- In `re` Regular Expression Syntax, global inline flags (e.g. `(?i)`) can now only be used at the start of regular expressions. Using them elsewhere has been deprecated since Python 3.6. (Contributed by Serhiy Storchaka in bpo-47066 [https://bugs.python.org/issue?@action=redirect&bpo=47066].)
- In the `re` module, several long-standing bugs where fixed that, in rare cases, could cause capture groups to get the wrong result. Therefore, this could change the captured output in these cases. (Contributed by Ma Lin in bpo-35859 [https://bugs.python.org/issue?@action=redirect&bpo=35859].)

Build Changes

- CPython now has **PEP 11** [https://peps.python.org/pep-0011/] **Tier 3 support** [https://peps.python.org/pep-0011/#tier-3] for cross compiling to the WebAssembly [https://webassembly.org/] platforms Emscripten [https://emscripten.org/] (`wasm32-unknown-emscripten`, i.e. Python in the browser) and WebAssembly System Interface (WASI) [https://wasi.dev/] (`wasm32-unknown-wasi`). The effort is inspired by previous work like Pyodide [https://pyodide.org/]. These platforms provide a limited subset of POSIX APIs; Python standard libraries features and modules related to networking, processes, threading, signals, mmap, and users/groups are not available or don’t work. (Emscripten contributed by Christian Heimes and Ethan Smith in gh-84461 [https://github.com/python/cpython/issues/84461] and WASI contributed by Christian Heimes in gh-90473 [https://github.com/python/cpython/issues/90473]; platforms promoted in gh-95085 [https://github.com/python/cpython/issues/95085])

- Building CPython now requires:

  - A C11 [https://en.cppreference.com/w/c/11] compiler and standard library. Optional C11 features [https://en.wikipedia.org/wiki/C11_(C_standard_revision)#Optional_features] are not required. (Contributed by Victor Stinner in bpo-46656 [https://bugs.python.org/issue?@action=redirect&bpo=46656], bpo-45440 [https://bugs.python.org/issue?@action=redirect&bpo=45440] and bpo-46640 [https://bugs.python.org/issue?@action=redirect&bpo=46640].)
  - Support for IEEE 754 [https://en.wikipedia.org/wiki/IEEE_754] floating-point numbers. (Contributed by Victor Stinner in bpo-46917 [https://bugs.python.org/issue?@action=redirect&bpo=46917].)

- The **`Py_NO_NAN`** macro has been removed. Since CPython now requires IEEE 754 floats, NaN values are always available. (Contributed by Victor Stinner in bpo-46656 [https://bugs.python.org/issue?@action=redirect&bpo=46656].)

- The **`tkinter`** package now requires Tcl/Tk [https://www.tcl.tk] version 8.5.12 or newer. (Contributed by Serhiy Storchaka in bpo-46996 [https://bugs.python.org/issue?@action=redirect&bpo=46996].)

- Build dependencies, compiler flags, and linker flags for most stdlib extension modules are now detected by **configure**. libffi, libnsl, libsqlite3, zlib, bzip2, liblzma, libcrypt, Tcl/Tk, and uuid flags are detected by pkg-config [https://www.freedesktop.org/wiki/Software/pkg-config/] (when available). **`tkinter`** now requires a pkg-config command to detect development settings for Tcl/Tk [https://www.tcl.tk] headers and libraries. (Contributed by Christian Heimes and Erlend Egeberg Aasland in bpo-45847 [https://bugs.python.org/issue?@action=redirect&bpo=45847], bpo-45747 [https://bugs.python.org/issue?@action=redirect&bpo=45747], and bpo-45763 [https://bugs.python.org/issue?@action=redirect&bpo=45763].)

- libpython is no longer linked against libcrypt. (Contributed by Mike Gilbert in bpo-45433 [https://bugs.python.org/issue?@action=redirect&bpo=45433].)

- CPython can now be built with the ThinLTO [https://clang.llvm.org/docs/ThinLTO.html] option via passing `thin` to **`--with-lto`**, i.e. `--with-lto=thin`. (Contributed by Donghee Na and Brett Holman in bpo-44340 [https://bugs.python.org/issue?@action=redirect&bpo=44340].)

- Freelists for object structs can now be disabled. A new **configure** option `--without-freelists` can be used to disable all freelists except empty tuple singleton. (Contributed by Christian Heimes in bpo-45522 [https://bugs.python.org/issue?@action=redirect&bpo=45522].)
- `Modules/Setup` and `Modules/makesetup` have been improved and tied up. Extension modules can now be built through `makesetup`. All except some test modules can be linked statically into a main binary or library. (Contributed by Brett Cannon and Christian Heimes in bpo-45548 [https://bugs.python.org/issue?@action=redirect&bpo=45548], bpo-45570 [https://bugs.python.org/issue?@action=redirect&bpo=45570], bpo-45571 [https://bugs.python.org/issue?@action=redirect&bpo=45571], and bpo-43974 [https://bugs.python.org/issue?@action=redirect&bpo=43974].)

  Note

  Use the environment variables **`TCLTK_CFLAGS`** and **`TCLTK_LIBS`** to manually specify the location of Tcl/Tk headers and libraries. The **configure** options **`--with-tcltk-includes`** and **`--with-tcltk-libs`** have been removed.

  On RHEL 7 and CentOS 7 the development packages do not provide `tcl.pc` and `tk.pc`; use `TCLTK_LIBS="-ltk8.5 -ltkstub8.5 -ltcl8.5"`. The directory `Misc/rhel7` contains `.pc` files and instructions on how to build Python with RHEL 7’s and CentOS 7’s Tcl/Tk and OpenSSL.

- CPython will now use 30-bit digits by default for the Python **`int`** implementation. Previously, the default was to use 30-bit digits on platforms with `SIZEOF_VOID_P >= 8`, and 15-bit digits otherwise. It’s still possible to explicitly request use of 15-bit digits via either the **`--enable-big-digits`** option to the configure script or (for Windows) the `PYLONG_BITS_IN_DIGIT` variable in `PC/pyconfig.h`, but this option may be removed at some point in the future. (Contributed by Mark Dickinson in bpo-45569 [https://bugs.python.org/issue?@action=redirect&bpo=45569].)

C API Changes

New Features

- Add a new `PyType_GetName()` function to get type’s short name. (Contributed by Hai Shi in bpo-42035 [https://bugs.python.org/issue?@action=redirect&bpo=42035].)

- Add a new `PyType_GetQualName()` function to get type’s qualified name. (Contributed by Hai Shi in bpo-42035 [https://bugs.python.org/issue?@action=redirect&bpo=42035].)

- Add new `PyThreadState_EnterTracing()` and `PyThreadState_LeaveTracing()` functions to the limited C API to suspend and resume tracing and profiling. (Contributed by Victor Stinner in bpo-43760 [https://bugs.python.org/issue?@action=redirect&bpo=43760].)

- Added the `Py_Version` constant which bears the same value as `PY_VERSION_HEX`. (Contributed by Gabriele N. Tornetta in bpo-43931 [https://bugs.python.org/issue?@action=redirect&bpo=43931].)

- `Py_buffer` and APIs are now part of the limited API and the stable ABI:

    - `PyObject_CheckBuffer()`
    - `PyObject_GetBuffer()`
    - `PyBuffer_GetPointer()`
    - `PyBuffer_SizeFromFormat()`
    - `PyBuffer_ToContiguous()`
    - `PyBuffer_FromContiguous()`
    - `PyObject_CopyData()`
    - `PyBuffer_IsContiguous()`
    - `PyBuffer_FillContiguousStrides()`
    - `PyBuffer_FillInfo()`
    - `PyBuffer_Release()`
    - `PyMemoryView_FromBuffer()`
    - `bf_getbuffer` and `bf_releasebuffer` type slots

  (Contributed by Christian Heimes in bpo-45459 [https://bugs.python.org/issue?@action=redirect&bpo=45459].)

- Added the `PyType_GetModuleByDef()` function, used to get the module in which a method was defined, in cases where this information is not available directly (via `PyCMethod`). (Contributed by Petr Viktorin in bpo-46613 [https://bugs.python.org/issue?@action=redirect&bpo=46613].)

- Add new functions to pack and unpack C double (serialize and deserialize): `PyFloat_Pack2()`, `PyFloat_Pack4()`, `PyFloat_Pack8()`, `PyFloat_Unpack2()`, `PyFloat_Unpack4()` and `PyFloat_Unpack8()`. (Contributed by Victor Stinner in bpo-46906 [https://bugs.python.org/issue?@action=redirect&bpo=46906].)

- Add new functions to get frame object attributes: `PyFrame_GetBuiltins()`, `PyFrame_GetGenerator()`, `PyFrame_GetGlobals()`, `PyFrame_GetLasti()`.

- Added two new functions to get and set the active exception instance: `PyErr_GetHandledException()` and `PyErr_SetHandledException()`. These are alternatives to `PyErr_SetExcInfo()` and `PyErr_GetExcInfo()` which work with the legacy 3-tuple representation of exceptions. (Contributed by Irit Katriel in bpo-46343 [https://bugs.python.org/issue?@action=redirect&bpo=46343].)

- Added the `PyConfig.safe_path` member. (Contributed by Victor Stinner in gh-57684 [https://github.com/python/cpython/issues/57684].)

Porting to Python 3.11

- Some macros have been converted to static inline functions to avoid macro pitfalls [https://gcc.gnu.org/onlinedocs/cpp/Macro-Pitfalls.html]. The change should be mostly transparent to users, as the replacement functions will cast their arguments to the expected types to avoid compiler warnings due to static type checks. However, when the limited C API is set to >=3.11, these casts are not done, and callers will need to cast arguments to their expected types. See **PEP 670** [https://peps.python.org/pep-0670/] for more details. (Contributed by Victor Stinner and Erlend E. Aasland in gh-89653 [https://github.com/python/cpython/issues/89653].)

- `PyErr_SetExcInfo()` no longer uses the `type` and `traceback` arguments, the interpreter now derives those values from the exception instance (the `value` argument). The function still steals references of all three arguments. (Contributed by Irit Katriel in bpo-45711 [https://bugs.python.org/issue?@action=redirect&bpo=45711].)

- `PyErr_GetExcInfo()` now derives the `type` and `traceback` fields of the result from the exception instance (the `value` field). (Contributed by Irit Katriel in bpo-45711 [https://bugs.python.org/issue?@action=redirect&bpo=45711].)

- `_frozen` has a new `is_package` field to indicate whether or not the frozen module is a package. Previously, a negative value in the `size` field was the indicator. Now only non-negative values be used for `size`. (Contributed by Kumar Aditya in bpo-46608 [https://bugs.python.org/issue?@action=redirect&bpo=46608].)

- `_PyFrameEvalFunction()` now takes `_PyInterpreterFrame*` as its second parameter, instead of `PyFrameObject*`. See **PEP 523** [https://peps.python.org/pep-0523/] for more details of how to use this function pointer type.

- **`PyCode_New()`** and **`PyCode_NewWithPosOnlyArgs()`** now take an additional `exception_table` argument. Using these functions should be avoided, if at all possible. To get a custom code object: create a code object using the compiler, then get a modified version with the `replace` method.

- `PyCodeObject` no longer has the `co_code`, `co_varnames`, `co_cellvars` and `co_freevars` fields. Instead, use `PyCode_GetCode()`, `PyCode_GetVarnames()`, `PyCode_GetCellvars()` and `PyCode_GetFreevars()` respectively to access them via the C API. (Contributed by Brandt Bucher in bpo-46841 [https://bugs.python.org/issue?@action=redirect&bpo=46841] and Ken Jin in gh-92154 [https://github.com/python/cpython/issues/92154] and gh-94936 [https://github.com/python/cpython/issues/94936].)

- The old trashcan macros (`Py_TRASHCAN_SAFE_BEGIN`/`Py_TRASHCAN_SAFE_END`) are now deprecated. They should be replaced by the new macros `Py_TRASHCAN_BEGIN` and `Py_TRASHCAN_END`.

  A tp_dealloc function that has the old macros, such as:

  ```
  static void
  mytype_dealloc(mytype *p)
  {
      PyObject_GC_UnTrack(p);
      Py_TRASHCAN_SAFE_BEGIN(p);
      ...
      Py_TRASHCAN_SAFE_END
  }
  ```

  should migrate to the new macros as follows:

  ```
  static void
  mytype_dealloc(mytype *p)
  ```

```
{
    PyObject_GC_UnTrack(p);
    Py_TRASHCAN_BEGIN(p, mytype_dealloc)
    ...
    Py_TRASHCAN_END
}
```

Note that `Py_TRASHCAN_BEGIN` has a second argument which should be the deallocation function it is in.

To support older Python versions in the same codebase, you can define the following macros and use them throughout the code (credit: these were copied from the `mypy` codebase):

```
#if PY_VERSION_HEX >= 0x03080000
#  define CPy_TRASHCAN_BEGIN(op, dealloc) Py_TRASHCAN_BEGIN(op, dealloc)
#  define CPy_TRASHCAN_END(op) Py_TRASHCAN_END
#else
#  define CPy_TRASHCAN_BEGIN(op, dealloc) Py_TRASHCAN_SAFE_BEGIN(op)
#  define CPy_TRASHCAN_END(op) Py_TRASHCAN_SAFE_END(op)
#endif
```

- The **`PyType_Ready()`** function now raises an error if a type is defined with the **`Py_TPFLAGS_HAVE_GC`** flag set but has no traverse function (**`PyTypeObject.tp_traverse`**). (Contributed by Victor Stinner in bpo-44263 [https://bugs.python.org/issue?@action=redirect&bpo=44263].)

- Heap types with the **`Py_TPFLAGS_IMMUTABLETYPE`** flag can now inherit the **PEP 590** [https://peps.python.org/pep-0590/] vectorcall protocol. Previously, this was only possible for static types. (Contributed by Erlend E. Aasland in bpo-43908 [https://bugs.python.org/issue?@action=redirect&bpo=43908])

- Since **`Py_TYPE()`** is changed to a inline static function, `Py_TYPE(obj) = new_type` must be replaced with `Py_SET_TYPE(obj, new_type)`: see the **`Py_SET_TYPE()`** function (available since Python 3.9). For backward compatibility, this macro can be used:

```
#if PY_VERSION_HEX < 0x030900A4 && !defined(Py_SET_TYPE)
static inline void _Py_SET_TYPE(PyObject *ob, PyTypeObject *type)
{ ob->ob_type = type; }
#define Py_SET_TYPE(ob, type) _Py_SET_TYPE((PyObject*)(ob), type)
#endif
```

(Contributed by Victor Stinner in bpo-39573 [https://bugs.python.org/issue?@action=redirect&bpo=39573].)

- Since **`Py_SIZE()`** is changed to a inline static function, `Py_SIZE(obj) = new_size` must be replaced with `Py_SET_SIZE(obj, new_size)`: see the **`Py_SET_SIZE()`** function (available since Python 3.9). For backward compatibility, this macro can be used:

  ```
  #if PY_VERSION_HEX < 0x030900A4 && !defined(Py_SET_SIZE)
  static inline void _Py_SET_SIZE(PyVarObject *ob, Py_ssize_t size)
  { ob->ob_size = size; }
  #define Py_SET_SIZE(ob, size) _Py_SET_SIZE((PyVarObject*)(ob), size)
  #endif
  ```

  (Contributed by Victor Stinner in bpo-39573 [https://bugs.python.org/issue?@action=redirect&bpo=39573].)

- `<Python.h>` no longer includes the header files `<stdlib.h>`, `<stdio.h>`, `<errno.h>` and `<string.h>` when the `Py_LIMITED_API` macro is set to `0x030b0000` (Python 3.11) or higher. C extensions should explicitly include the header files after `#include <Python.h>`. (Contributed by Victor Stinner in bpo-45434 [https://bugs.python.org/issue?@action=redirect&bpo=45434].)

- The non-limited API files `cellobject.h`, `classobject.h`, `code.h`, `context.h`, `funcobject.h`, `genobject.h` and `longintrepr.h` have been moved to the `Include/cpython` directory. Moreover, the `eval.h` header file was removed. These files must not be included directly, as they are already included in `Python.h`: Include Files. If they have been included directly, consider including `Python.h` instead. (Contributed by Victor Stinner in bpo-35134 [https://bugs.python.org/issue?@action=redirect&bpo=35134].)

- The **`PyUnicode_CHECK_INTERNED()`** macro has been excluded from the limited C API. It was never usable there, because it used internal structures which are not available in the limited C API. (Contributed by Victor Stinner in bpo-46007 [https://bugs.python.org/issue?@action=redirect&bpo=46007].)

- The following frame functions and type are now directly available with `#include <Python.h>`, it’s no longer needed to add `#include <frameobject.h>`:

  - **`PyFrame_Check()`**

  - `PyFrame_GetBack()`
  - `PyFrame_GetBuiltins()`
  - `PyFrame_GetGenerator()`
  - `PyFrame_GetGlobals()`
  - `PyFrame_GetLasti()`
  - `PyFrame_GetLocals()`
  - `PyFrame_Type`

  (Contributed by Victor Stinner in gh-93937 [https://github.com/python/cpython/issues/93937].)

- The `PyFrameObject` structure members have been removed from the public C API.

  While the documentation notes that the `PyFrameObject` fields are subject to change at any time, they have been stable for a long time and were used in several popular extensions.

  In Python 3.11, the frame struct was reorganized to allow performance optimizations. Some fields were removed entirely, as they were details of the old implementation.

  `PyFrameObject` fields:

  - `f_back`: use `PyFrame_GetBack()`.
  - `f_blockstack`: removed.
  - `f_builtins`: use `PyFrame_GetBuiltins()`.
  - `f_code`: use `PyFrame_GetCode()`.
  - `f_gen`: use `PyFrame_GetGenerator()`.
  - `f_globals`: use `PyFrame_GetGlobals()`.
  - `f_iblock`: removed.
  - `f_lasti`: use `PyFrame_GetLasti()`. Code using `f_lasti` with `PyCode_Addr2Line()` should use `PyFrame_GetLineNumber()` instead; it may be faster.
  - `f_lineno`: use `PyFrame_GetLineNumber()`
  - `f_locals`: use `PyFrame_GetLocals()`.
  - `f_stackdepth`: removed.
  - `f_state`: no public API (renamed to `f_frame.f_state`).
  - `f_trace`: no public API.
  - `f_trace_lines`: use `PyObject_GetAttrString((PyObject*)frame, "f_trace_lines")`.
  - `f_trace_opcodes`: use `PyObject_GetAttrString((PyObject*)frame, "f_trace_opcodes")`.

  - `f_localsplus`: no public API (renamed to `f_frame.localsplus`).
  - `f_valuestack`: removed.

  The Python frame object is now created lazily. A side effect is that the **`f_back`** member must not be accessed directly, since its value is now also computed lazily. The **`PyFrame_GetBack()`** function must be called instead.

  Debuggers that accessed the **`f_locals`** directly *must* call **`PyFrame_GetLocals()`** instead. They no longer need to call **`PyFrame_FastToLocalsWithError()`** or **`PyFrame_LocalsToFast()`**, in fact they should not call those functions. The necessary updating of the frame is now managed by the virtual machine.

  Code defining `PyFrame_GetCode()` on Python 3.8 and older:

```
#if PY_VERSION_HEX < 0x030900B1
static inline PyCodeObject* PyFrame_GetCode(PyFrameObject *frame)
{
    Py_INCREF(frame->f_code);
    return frame->f_code;
}
#endif
```

  Code defining `PyFrame_GetBack()` on Python 3.8 and older:

```
#if PY_VERSION_HEX < 0x030900B1
static inline PyFrameObject* PyFrame_GetBack(PyFrameObject *frame)
{
    Py_XINCREF(frame->f_back);
    return frame->f_back;
}
#endif
```

  Or use the pythoncapi_compat project [https://github.com/python/pythoncapi-compat] to get these two functions on older Python versions.

- Changes of the **`PyThreadState`** structure members:

  - `frame`: removed, use **`PyThreadState_GetFrame()`** (function added to Python 3.9 by bpo-40429 [https://bugs.python.org/issue?@action=redirect&bpo=40429]). Warning: the function returns a strong reference, need to call **`Py_XDECREF()`**.
  - `tracing`: changed, use **`PyThreadState_EnterTracing()`** and **`PyThreadState_LeaveTracing()`** (functions added to Python 3.11 by

  bpo-43760 [https://bugs.python.org/issue?@action=redirect&bpo=43760]).
  - `recursion_depth`: removed, use `(tstate->recursion_limit - tstate->recursion_remaining)` instead.
  - `stackcheck_counter`: removed.

Code defining `PyThreadState_GetFrame()` on Python 3.8 and older:

```
#if PY_VERSION_HEX < 0x030900B1
static inline PyFrameObject* PyThreadState_GetFrame(PyThreadState
*tstate)
{
    Py_XINCREF(tstate->frame);
    return tstate->frame;
}
#endif
```

Code defining `PyThreadState_EnterTracing()` and `PyThreadState_LeaveTracing()` on Python 3.10 and older:

```
#if PY_VERSION_HEX < 0x030B00A2
static inline void PyThreadState_EnterTracing(PyThreadState *tstate)
{
    tstate->tracing++;
#if PY_VERSION_HEX >= 0x030A00A1
    tstate->cframe->use_tracing = 0;
#else
    tstate->use_tracing = 0;
#endif
}

static inline void PyThreadState_LeaveTracing(PyThreadState *tstate)
{
    int use_tracing = (tstate->c_tracefunc != NULL || tstate-
>c_profilefunc != NULL);
    tstate->tracing--;
#if PY_VERSION_HEX >= 0x030A00A1
    tstate->cframe->use_tracing = use_tracing;
#else
    tstate->use_tracing = use_tracing;
#endif
}
#endif
```

Or use the pythoncapi-compat project [https://github.com/python/pythoncapi-compat] to get these functions on old Python functions.

- Distributors are encouraged to build Python with the optimized Blake2 library libb2 [https://www.blake2.net/].

- The `PyConfig.module_search_paths_set` field must now be set to 1 for initialization to use `PyConfig.module_search_paths` to initialize `sys.path`. Otherwise, initialization will recalculate the path and replace any values added to `module_search_paths`.

- `PyConfig_Read()` no longer calculates the initial search path, and will not fill any values into `PyConfig.module_search_paths`. To calculate default paths and then modify them, finish initialization and use `PySys_GetObject()` to retrieve `sys.path` as a Python list object and modify it directly.

Deprecated

- Deprecate the following functions to configure the Python initialization:

    - **`PySys_AddWarnOptionUnicode()`**
    - **`PySys_AddWarnOption()`**
    - **`PySys_AddXOption()`**
    - **`PySys_HasWarnOptions()`**
    - **`PySys_SetArgvEx()`**
    - **`PySys_SetArgv()`**
    - **`PySys_SetPath()`**
    - **`Py_SetPath()`**
    - **`Py_SetProgramName()`**
    - **`Py_SetPythonHome()`**
    - **`Py_SetStandardStreamEncoding()`**
    - **`_Py_SetProgramFullPath()`**

    Use the new `PyConfig` API of the Python Initialization Configuration instead (**PEP 587** [https://peps.python.org/pep-0587/]). (Contributed by Victor Stinner in gh-88279 [https://github.com/python/cpython/issues/88279].)

- Deprecate the `ob_shash` member of the `PyBytesObject`. Use `PyObject_Hash()` instead. (Contributed by Inada Naoki in bpo-46864 [https://bugs.python.org/issue?@action=redirect&bpo=46864].)

Pending Removal in Python 3.12

The following C APIs have been deprecated in earlier Python releases, and will be removed in Python 3.12.

- **`PyUnicode_AS_DATA()`**
- **`PyUnicode_AS_UNICODE()`**
- **`PyUnicode_AsUnicodeAndSize()`**
- **`PyUnicode_AsUnicode()`**
- **`PyUnicode_FromUnicode()`**
- **`PyUnicode_GET_DATA_SIZE()`**
- **`PyUnicode_GET_SIZE()`**
- **`PyUnicode_GetSize()`**
- **`PyUnicode_IS_COMPACT()`**
- **`PyUnicode_IS_READY()`**
- **`PyUnicode_READY()`**
- **`PyUnicode_WSTR_LENGTH()`**
- **`_PyUnicode_AsUnicode()`**
- **`PyUnicode_WCHAR_KIND`**
- **`PyUnicodeObject`**
- **`PyUnicode_InternImmortal()`**

Removed

- **`PyFrame_BlockSetup()`** and **`PyFrame_BlockPop()`** have been removed. (Contributed by Mark Shannon in bpo-40222 [https://bugs.python.org/issue?@action=redirect&bpo=40222].)
- Remove the following math macros using the `errno` variable:
  - `Py_ADJUST_ERANGE1()`
  - `Py_ADJUST_ERANGE2()`
  - `Py_OVERFLOWED()`
  - `Py_SET_ERANGE_IF_OVERFLOW()`
  - `Py_SET_ERRNO_ON_MATH_ERROR()`

  (Contributed by Victor Stinner in bpo-45412 [https://bugs.python.org/issue?@action=redirect&bpo=45412].)
- Remove `Py_UNICODE_COPY()` and `Py_UNICODE_FILL()` macros, deprecated since Python 3.3. Use `PyUnicode_CopyCharacters()` or `memcpy()` (`wchar_t*` string), and `PyUnicode_Fill()` functions instead. (Contributed by Victor Stinner in bpo-41123 [https://bugs.python.org/issue?@action=redirect&bpo=41123].)

- Remove the `pystrhex.h` header file. It only contains private functions. C extensions should only include the main `<Python.h>` header file. (Contributed by Victor Stinner in bpo-45434 [https://bugs.python.org/issue?@action=redirect&bpo=45434].)
- Remove the `Py_FORCE_DOUBLE()` macro. It was used by the `Py_IS_INFINITY()` macro. (Contributed by Victor Stinner in bpo-45440 [https://bugs.python.org/issue?@action=redirect&bpo=45440].)
- The following items are no longer available when **`Py_LIMITED_API`** is defined:
  - **`PyMarshal_WriteLongToFile()`**
  - **`PyMarshal_WriteObjectToFile()`**
  - **`PyMarshal_ReadObjectFromString()`**
  - **`PyMarshal_WriteObjectToString()`**
  - the `Py_MARSHAL_VERSION` macro

  These are not part of the limited API.

  (Contributed by Victor Stinner in bpo-45474 [https://bugs.python.org/issue?@action=redirect&bpo=45474].)
- Exclude **`PyWeakref_GET_OBJECT()`** from the limited C API. It never worked since the **`PyWeakReference`** structure is opaque in the limited C API. (Contributed by Victor Stinner in bpo-35134 [https://bugs.python.org/issue?@action=redirect&bpo=35134].)
- Remove the `PyHeapType_GET_MEMBERS()` macro. It was exposed in the public C API by mistake, it must only be used by Python internally. Use the `PyTypeObject.tp_members` member instead. (Contributed by Victor Stinner in bpo-40170 [https://bugs.python.org/issue?@action=redirect&bpo=40170].)
- Remove the `HAVE_PY_SET_53BIT_PRECISION` macro (moved to the internal C API). (Contributed by Victor Stinner in bpo-45412 [https://bugs.python.org/issue?@action=redirect&bpo=45412].)
- Remove the **`Py_UNICODE`** encoder APIs, as they have been deprecated since Python 3.3, are little used and are inefficient relative to the recommended alternatives.

  The removed functions are:

- `PyUnicode_Encode()`
- `PyUnicode_EncodeASCII()`
- `PyUnicode_EncodeLatin1()`
- `PyUnicode_EncodeUTF7()`
- `PyUnicode_EncodeUTF8()`
- `PyUnicode_EncodeUTF16()`
- `PyUnicode_EncodeUTF32()`
- `PyUnicode_EncodeUnicodeEscape()`
- `PyUnicode_EncodeRawUnicodeEscape()`
- `PyUnicode_EncodeCharmap()`
- `PyUnicode_TranslateCharmap()`
- `PyUnicode_EncodeDecimal()`
- `PyUnicode_TransformDecimalToASCII()`

See **PEP 624** [https://peps.python.org/pep-0624/] for details and **migration guidance** [https://peps.python.org/pep-0624/#alternative-apis]. (Contributed by Inada Naoki in bpo-44029 [https://bugs.python.org/issue?@action=redirect&bpo=44029].)

Notable changes in 3.11.4

tarfile

- The extraction methods in `tarfile`, and `shutil.unpack_archive()`, have a new a *filter* argument that allows limiting tar features than may be surprising or dangerous, such as creating files outside the destination directory. See Extraction filters for details. In Python 3.12, use without the *filter* argument will show a `DeprecationWarning`. In Python 3.14, the default will switch to `'data'`. (Contributed by Petr Viktorin in **PEP 706** [https://peps.python.org/pep-0706/].)

Notable changes in 3.11.5

OpenSSL

- Windows builds and macOS installers from python.org now use OpenSSL 3.0.

What’s New In Python 3.10

Editor:: Pablo Galindo Salgado

This article explains the new features in Python 3.10, compared to 3.9. Python 3.10 was released on October 4, 2021. For full details, see the changelog.

Summary – Release highlights

New syntax features:

- **PEP 634** [https://peps.python.org/pep-0634/], Structural Pattern Matching: Specification
- **PEP 635** [https://peps.python.org/pep-0635/], Structural Pattern Matching: Motivation and Rationale
- **PEP 636** [https://peps.python.org/pep-0636/], Structural Pattern Matching: Tutorial
- bpo-12782 [https://bugs.python.org/issue?@action=redirect&bpo=12782], Parenthesized context managers are now officially allowed.

New features in the standard library:

- **PEP 618** [https://peps.python.org/pep-0618/], Add Optional Length-Checking To zip.

Interpreter improvements:

- **PEP 626** [https://peps.python.org/pep-0626/], Precise line numbers for debugging and other tools.

New typing features:

- **PEP 604** [https://peps.python.org/pep-0604/], Allow writing union types as X | Y
- **PEP 612** [https://peps.python.org/pep-0612/], Parameter Specification Variables
- **PEP 613** [https://peps.python.org/pep-0613/], Explicit Type Aliases
- **PEP 647** [https://peps.python.org/pep-0647/], User-Defined Type Guards

Important deprecations, removals or restrictions:

- **PEP 644** [https://peps.python.org/pep-0644/], Require OpenSSL 1.1.1 or newer
- **PEP 632** [https://peps.python.org/pep-0632/], Deprecate distutils module.
- **PEP 623** [https://peps.python.org/pep-0623/], Deprecate and prepare for the removal of the wstr member in PyUnicodeObject.
- **PEP 624** [https://peps.python.org/pep-0624/], Remove Py_UNICODE encoder APIs
- **PEP 597** [https://peps.python.org/pep-0597/], Add optional EncodingWarning

New Features

Parenthesized context managers

Using enclosing parentheses for continuation across multiple lines in context managers is now supported. This allows formatting a long collection of context managers in multiple lines in a similar way as it was previously possible with import statements. For instance, all these examples are now valid:

```
with (CtxManager() as example):
    ...

with (
    CtxManager1(),
    CtxManager2()
):
    ...

with (CtxManager1() as example,
      CtxManager2()):
    ...

with (CtxManager1(),
      CtxManager2() as example):
    ...

with (
    CtxManager1() as example1,
    CtxManager2() as example2
):
    ...
```

it is also possible to use a trailing comma at the end of the enclosed group:

```
with (
    CtxManager1() as example1,
    CtxManager2() as example2,
    CtxManager3() as example3,
):
    ...
```

This new syntax uses the non LL(1) capacities of the new parser. Check **PEP 617** [https://peps.python.org/pep-0617/] for more details.

(Contributed by Guido van Rossum, Pablo Galindo and Lysandros Nikolaou in bpo-12782 [https://bugs.python.org/issue?@action=redirect&bpo=12782] and bpo-40334 [https://bugs.python.org/issue?@action=redirect&bpo=40334].)

Better error messages

SyntaxErrors

When parsing code that contains unclosed parentheses or brackets the interpreter now includes the location of the unclosed bracket of parentheses instead of displaying *SyntaxError: unexpected EOF while parsing* or pointing to some incorrect location. For instance, consider the following code (notice the unclosed ‘{‘):

```
expected = {9: 1, 18: 2, 19: 2, 27: 3, 28: 3, 29: 3, 36: 4, 37:
4,
            38: 4, 39: 4, 45: 5, 46: 5, 47: 5, 48: 5, 49: 5,
54: 6,
some_other_code = foo()
```

Previous versions of the interpreter reported confusing places as the location of the syntax error:

```
File "example.py", line 3
    some_other_code = foo()
                    ^
SyntaxError: invalid syntax
```

but in Python 3.10 a more informative error is emitted:

```
File "example.py", line 1
    expected = {9: 1, 18: 2, 19: 2, 27: 3, 28: 3, 29: 3, 36: 4,
37: 4,
               ^
SyntaxError: '{' was never closed
```

In a similar way, errors involving unclosed string literals (single and triple quoted) now point to the start of the string instead of reporting EOF/EOL.

These improvements are inspired by previous work in the PyPy interpreter.

(Contributed by Pablo Galindo in bpo-42864 [https://bugs.python.org/issue?@action=redirect&bpo=42864] and Batuhan Taskaya in bpo-40176 [https://bugs.python.org/issue?@action=redirect&bpo=40176].)

`SyntaxError` exceptions raised by the interpreter will now highlight the full error range of the expression that constitutes the syntax error itself, instead of just where the problem is detected. In this way, instead of displaying (before Python 3.10):

```
>>> foo(x, z for z in range(10), t, w)
  File "<stdin>", line 1
    foo(x, z for z in range(10), t, w)
           ^
SyntaxError: Generator expression must be parenthesized
```

now Python 3.10 will display the exception as:

```
>>> foo(x, z for z in range(10), t, w)
  File "<stdin>", line 1
    foo(x, z for z in range(10), t, w)
           ^^^^^^^^^^^^^^^^^^^^
SyntaxError: Generator expression must be parenthesized
```

This improvement was contributed by Pablo Galindo in bpo-43914 [https://bugs.python.org/issue?@action=redirect&bpo=43914].

A considerable amount of new specialized messages for `SyntaxError` exceptions have been incorporated. Some of the most notable ones are as follows:

- Missing `:` before blocks:

  ```
  >>> if rocket.position > event_horizon
    File "<stdin>", line 1
      if rocket.position > event_horizon
                                        ^
  SyntaxError: expected ':'
  ```

(Contributed by Pablo Galindo in bpo-42997 [https://bugs.python.org/issue?@action=redirect&bpo=42997].)

- Unparenthesised tuples in comprehensions targets:

```
>>> {x,y for x,y in zip('abcd', '1234')}
  File "<stdin>", line 1
    {x,y for x,y in zip('abcd', '1234')}
     ^
SyntaxError: did you forget parentheses around the
comprehension target?
```

(Contributed by Pablo Galindo in bpo-43017 [https://bugs.python.org/issue?@action=redirect&bpo=43017].)

- Missing commas in collection literals and between expressions:

```
>>> items = {
... x: 1,
... y: 2
... z: 3,
  File "<stdin>", line 3
    y: 2
       ^
SyntaxError: invalid syntax. Perhaps you forgot a comma?
```

(Contributed by Pablo Galindo in bpo-43822 [https://bugs.python.org/issue?@action=redirect&bpo=43822].)

- Multiple Exception types without parentheses:

```
>>> try:
...     build_dyson_sphere()
... except NotEnoughScienceError, NotEnoughResourcesError:
  File "<stdin>", line 3
    except NotEnoughScienceError, NotEnoughResourcesError:
           ^
SyntaxError: multiple exception types must be parenthesized
```

(Contributed by Pablo Galindo in bpo-43149 [https://bugs.python.org/issue?@action=redirect&bpo=43149].)

- Missing `:` and values in dictionary literals:

```
>>> values = {
... x: 1,
... y: 2,
... z:
... }
  File "<stdin>", line 4
    z:
     ^
SyntaxError: expression expected after dictionary key and ':'

>>> values = {x:1, y:2, z w:3}
  File "<stdin>", line 1
    values = {x:1, y:2, z w:3}
                        ^
SyntaxError: ':' expected after dictionary key
```

(Contributed by Pablo Galindo in bpo-43823 [https://bugs.python.org/issue?@action=redirect&bpo=43823].)

- `try` blocks without `except` or `finally` blocks:

```
>>> try:
...     x = 2
... something = 3
  File "<stdin>", line 3
    something  = 3
    ^^^^^^^^^
SyntaxError: expected 'except' or 'finally' block
```

(Contributed by Pablo Galindo in bpo-44305 [https://bugs.python.org/issue?@action=redirect&bpo=44305].)

- Usage of `=` instead of `==` in comparisons:

```
>>> if rocket.position = event_horizon:
  File "<stdin>", line 1
    if rocket.position = event_horizon:
                       ^
SyntaxError: cannot assign to attribute here. Maybe you
meant '==' instead of '='?
```

(Contributed by Pablo Galindo in bpo-43797 [https://bugs.python.org/issue?@action=redirect&bpo=43797].)

- Usage of `*` in f-strings:

```
>>> f"Black holes {*all_black_holes} and revelations"
  File "<stdin>", line 1
    (*all_black_holes)
     ^
SyntaxError: f-string: cannot use starred expression here
```

(Contributed by Pablo Galindo in bpo-41064 [https://bugs.python.org/issue?@action=redirect&bpo=41064].)

IndentationErrors

Many `IndentationError` exceptions now have more context regarding what kind of block was expecting an indentation, including the location of the statement:

```
>>> def foo():
...    if lel:
...    x = 2
  File "<stdin>", line 3
    x = 2
    ^
IndentationError: expected an indented block after 'if'
statement in line 2
```

AttributeErrors

When printing `AttributeError`, **`PyErr_Display()`** will offer suggestions of similar attribute names in the object that the exception was raised from:

```
>>> collections.namedtoplo
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: module 'collections' has no attribute
'namedtoplo'. Did you mean: namedtuple?
```

(Contributed by Pablo Galindo in bpo-38530 [https://bugs.python.org/issue?@action=redirect&bpo=38530].)

Warning

Notice this won't work if **`PyErr_Display()`** is not called to display the error which can happen if some other custom error display function is used. This is a common scenario in some REPLs like IPython.

NameErrors

When printing `NameError` raised by the interpreter, **`PyErr_Display()`** will offer suggestions of similar variable names in the function that the exception was raised from:

```
>>> schwarzschild_black_hole = None
>>> schwarschild_black_hole
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'schwarschild_black_hole' is not defined. Did
you mean: schwarzschild_black_hole?
```

(Contributed by Pablo Galindo in bpo-38530 [https://bugs.python.org/issue?@action=redirect&bpo=38530].)

Warning

Notice this won’t work if `PyErr_Display()` is not called to display the error, which can happen if some other custom error display function is used. This is a common scenario in some REPLs like IPython.

PEP 626: Precise line numbers for debugging and other tools

PEP 626 brings more precise and reliable line numbers for debugging, profiling and coverage tools. Tracing events, with the correct line number, are generated for all lines of code executed and only for lines of code that are executed.

The `f_lineno` attribute of frame objects will always contain the expected line number.

The `co_lnotab` attribute of code objects is deprecated and will be removed in 3.12. Code that needs to convert from offset to line number should use the new `co_lines()` method instead.

PEP 634: Structural Pattern Matching

Structural pattern matching has been added in the form of a *match statement* and *case statements* of patterns with associated actions. Patterns consist of sequences, mappings, primitive data types as well as class instances. Pattern matching enables programs to extract information from complex data types, branch on the structure of data, and apply specific actions based on different forms of data.

Syntax and operations

The generic syntax of pattern matching is:

```
match subject:
    case <pattern_1>:
        <action_1>
    case <pattern_2>:
        <action_2>
    case <pattern_3>:
        <action_3>
    case _:
        <action_wildcard>
```

A match statement takes an expression and compares its value to successive patterns given as one or more case blocks. Specifically, pattern matching operates by:

1. using data with type and shape (the `subject`)
2. evaluating the `subject` in the `match` statement
3. comparing the subject with each pattern in a `case` statement from top to bottom until a match is confirmed.
4. executing the action associated with the pattern of the confirmed match
5. If an exact match is not confirmed, the last case, a wildcard _, if provided, will be used as the matching case. If an exact match is not confirmed and a wildcard case does not exist, the entire match block is a no-op.

Declarative approach

Readers may be aware of pattern matching through the simple example of matching a subject (data object) to a literal (pattern) with the switch statement found in C, Java or JavaScript (and many other languages). Often the switch statement is used for comparison of an object/expression with case statements containing literals.

More powerful examples of pattern matching can be found in languages such as Scala and Elixir. With structural pattern matching, the approach is “declarative” and explicitly states the conditions (the patterns) for data to match.

While an “imperative” series of instructions using nested “if” statements could be used to accomplish something similar to structural pattern matching, it is less clear than the “declarative” approach. Instead the “declarative” approach states the conditions to meet for a match and is more readable through its explicit patterns. While structural pattern matching can be used in its simplest form comparing a variable to a literal in a case statement, its true value for Python lies in its handling of the subject’s type and shape.

Simple pattern: match to a literal

Let’s look at this example as pattern matching in its simplest form: a value, the subject, being matched to several literals, the patterns. In the example below, `status` is the subject of the match statement. The patterns are each of the case statements, where literals represent request status codes. The associated action to the case is executed after a match:

```
def http_error(status):
    match status:
        case 400:
            return "Bad request"
        case 404:
            return "Not found"
        case 418:
            return "I'm a teapot"
        case _:
            return "Something's wrong with the internet"
```

If the above function is passed a `status` of 418, “I’m a teapot” is returned. If the above function is passed a `status` of 500, the case statement with _ will match as a wildcard, and “Something’s wrong with the internet” is returned. Note the last block: the variable name, _, acts as a *wildcard* and insures the subject will always match. The use of _ is optional.

You can combine several literals in a single pattern using `|` (“or”):

```
case 401 | 403 | 404:
    return "Not allowed"
```

Behavior without the wildcard

If we modify the above example by removing the last case block, the example becomes:

```
def http_error(status):
    match status:
        case 400:
            return "Bad request"
        case 404:
            return "Not found"
        case 418:
            return "I'm a teapot"
```

Without the use of _ in a case statement, a match may not exist. If no match exists, the behavior is a no-op. For example, if `status` of 500 is passed, a no-op occurs.

Patterns with a literal and variable

Patterns can look like unpacking assignments, and a pattern may be used to bind variables. In this example, a data point can be unpacked to its x-coordinate and y-coordinate:

```
# point is an (x, y) tuple
match point:
    case (0, 0):
        print("Origin")
    case (0, y):
        print(f"Y={y}")
    case (x, 0):
        print(f"X={x}")
    case (x, y):
        print(f"X={x}, Y={y}")
    case _:
        raise ValueError("Not a point")
```

The first pattern has two literals, `(0, 0)`, and may be thought of as an extension of the literal pattern shown above. The next two patterns combine a literal and a variable, and the variable *binds* a value from the subject (`point`). The fourth pattern captures two values, which makes it conceptually similar to the unpacking assignment `(x, y) = point`.

Patterns and classes

If you are using classes to structure your data, you can use as a pattern the class name followed by an argument list resembling a constructor. This pattern has the ability to capture instance attributes into variables:

```
class Point:
    def __init__(self, x, y):
        self.x = x
        self.y = y

def location(point):
    match point:
        case Point(x=0, y=0):
            print("Origin is the point's location.")
        case Point(x=0, y=y):
            print(f"Y={y} and the point is on the y-axis.")
        case Point(x=x, y=0):
            print(f"X={x} and the point is on the x-axis.")
        case Point():
            print("The point is located somewhere else on the
plane.")
        case _:
            print("Not a point")
```

Patterns with positional parameters

You can use positional parameters with some builtin classes that provide an ordering for their attributes (e.g. dataclasses). You can also define a specific position for attributes in patterns by setting the `__match_args__` special attribute in your classes. If it’s set to (“x”, “y”), the following patterns are all equivalent (and all bind the `y` attribute to the `var` variable):

```
Point(1, var)
Point(1, y=var)
Point(x=1, y=var)
Point(y=var, x=1)
```

Nested patterns

Patterns can be arbitrarily nested. For example, if our data is a short list of points, it could be matched like this:

```
match points:
    case []:
        print("No points in the list.")
    case [Point(0, 0)]:
        print("The origin is the only point in the list.")
    case [Point(x, y)]:
        print(f"A single point {x}, {y} is in the list.")
    case [Point(0, y1), Point(0, y2)]:
        print(f"Two points on the Y axis at {y1}, {y2} are in
the list.")
    case _:
        print("Something else is found in the list.")
```

Complex patterns and the wildcard

To this point, the examples have used _ alone in the last case statement. A wildcard can be used in more complex patterns, such as `('error', code, _)`. For example:

```
match test_variable:
    case ('warning', code, 40):
        print("A warning has been received.")
    case ('error', code, _):
        print(f"An error {code} occurred.")
```

In the above case, `test_variable` will match for (‘error’, code, 100) and (‘error’, code, 800).

Guard

We can add an `if` clause to a pattern, known as a “guard”. If the guard is false, `match` goes on to try the next case block. Note that value capture happens before the guard is evaluated:

```
match point:
    case Point(x, y) if x == y:
        print(f"The point is located on the diagonal Y=X at
{x}.")
    case Point(x, y):
        print(f"Point is not on the diagonal.")
```

Other Key Features

Several other key features:

- Like unpacking assignments, tuple and list patterns have exactly the same meaning and actually match arbitrary sequences. Technically, the subject must be a sequence. Therefore, an important exception is that patterns don’t match iterators. Also, to prevent a common mistake, sequence patterns don’t match strings.

- Sequence patterns support wildcards: `[x, y, *rest]` and `(x, y, *rest)` work similar to wildcards in unpacking assignments. The name after `*` may also be `_`, so `(x, y, *_)` matches a sequence of at least two items without binding the remaining items.

- Mapping patterns: `{"bandwidth": b, "latency": l}` captures the `"bandwidth"` and `"latency"` values from a dict. Unlike sequence patterns, extra keys are ignored. A wildcard `**rest` is also supported. (But `**_` would be redundant, so is not allowed.)

- Subpatterns may be captured using the `as` keyword:

  ```
  case (Point(x1, y1), Point(x2, y2) as p2): ...
  ```

  This binds x1, y1, x2, y2 like you would expect without the `as` clause, and p2 to the entire second item of the subject.

- Most literals are compared by equality. However, the singletons `True`, `False` and `None` are compared by identity.

- Named constants may be used in patterns. These named constants must be dotted names to prevent the constant from being interpreted as a capture variable:

  ```
  from enum import Enum
  class Color(Enum):
      RED = 0
      GREEN = 1
      BLUE = 2

  color = Color.GREEN
  match color:
      case Color.RED:
          print("I see red!")
      case Color.GREEN:
          print("Grass is green")
  ```

```
        case Color.BLUE:
            print("I'm feeling the blues :(")
```

For the full specification see **PEP 634** [https://peps.python.org/pep-0634/]. Motivation and rationale are in **PEP 635** [https://peps.python.org/pep-0635/], and a longer tutorial is in **PEP 636** [https://peps.python.org/pep-0636/].

Optional `EncodingWarning` and `encoding="locale"` option

The default encoding of `TextIOWrapper` and `open()` is platform and locale dependent. Since UTF-8 is used on most Unix platforms, omitting `encoding` option when opening UTF-8 files (e.g. JSON, YAML, TOML, Markdown) is a very common bug. For example:

```
# BUG: "rb" mode or encoding="utf-8" should be used.
with open("data.json") as f:
    data = json.load(f)
```

To find this type of bug, an optional `EncodingWarning` is added. It is emitted when `sys.flags.warn_default_encoding` is true and locale-specific default encoding is used.

`-X warn_default_encoding` option and `PYTHONWARNDEFAULTENCODING` are added to enable the warning.

See Text Encoding for more information.

New Features Related to Type Hints

This section covers major changes affecting **PEP 484** [https://peps.python.org/pep-0484/] type hints and the `typing` module.

PEP 604: New Type Union Operator

A new type union operator was introduced which enables the syntax `X | Y`. This provides a cleaner way of expressing 'either type X or type Y' instead of using **`typing.Union`**, especially in type hints.

In previous versions of Python, to apply a type hint for functions accepting arguments of multiple types, **`typing.Union`** was used:

```
def square(number: Union[int, float]) -> Union[int, float]:
    return number ** 2
```

Type hints can now be written in a more succinct manner:

```
def square(number: int | float) -> int | float:
    return number ** 2
```

This new syntax is also accepted as the second argument to **`isinstance()`** and **`issubclass()`**:

```
>>> isinstance(1, int | str)
True
```

See Union Type and **PEP 604** [https://peps.python.org/pep-0604/] for more details.

(Contributed by Maggie Moss and Philippe Prados in bpo-41428 [https://bugs.python.org/issue?@action=redirect&bpo=41428], with additions by Yurii Karabas and Serhiy Storchaka in bpo-44490 [https://bugs.python.org/issue?@action=redirect&bpo=44490].)

PEP 612: Parameter Specification Variables

Two new options to improve the information provided to static type checkers for **PEP 484** [https://peps.python.org/pep-0484/]'s `Callable` have been added to the **`typing`** module.

The first is the parameter specification variable. They are used to forward the parameter types of one callable to another callable – a pattern commonly found in higher order functions and decorators. Examples of usage can be found in `typing.ParamSpec`. Previously, there was no easy way to type annotate dependency of parameter types in such a precise manner.

The second option is the new `Concatenate` operator. It's used in conjunction with parameter specification variables to type annotate a higher order callable which adds or removes parameters of another callable. Examples of usage can be found in `typing.Concatenate`.

See `typing.Callable`, `typing.ParamSpec`, `typing.Concatenate`, `typing.ParamSpecArgs`, `typing.ParamSpecKwargs`, and **PEP 612** [https://peps.python.org/pep-0612/] for more details.

(Contributed by Ken Jin in bpo-41559 [https://bugs.python.org/issue?@action=redirect&bpo=41559], with minor enhancements by Jelle Zijlstra in bpo-43783 [https://bugs.python.org/issue?@action=redirect&bpo=43783]. PEP written by Mark Mendoza.)

PEP 613: TypeAlias

PEP 484 [https://peps.python.org/pep-0484/] introduced the concept of type aliases, only requiring them to be top-level unannotated assignments. This simplicity sometimes made it difficult for type checkers to distinguish between type aliases and ordinary assignments, especially when forward references or invalid types were involved. Compare:

```
StrCache = 'Cache[str]'  # a type alias
LOG_PREFIX = 'LOG[DEBUG]'  # a module constant
```

Now the `typing` module has a special value `TypeAlias` which lets you declare type aliases more explicitly:

```
StrCache: TypeAlias = 'Cache[str]'  # a type alias
LOG_PREFIX = 'LOG[DEBUG]'  # a module constant
```

See **PEP 613** [https://peps.python.org/pep-0613/] for more details.

(Contributed by Mikhail Golubev in bpo-41923 [https://bugs.python.org/issue?@action=redirect&bpo=41923].)

PEP 647: User-Defined Type Guards

`TypeGuard` has been added to the **`typing`** module to annotate type guard functions and improve information provided to static type checkers during type narrowing. For more information, please see **`TypeGuard`**'s documentation, and **PEP 647** [https://peps.python.org/pep-0647/].

(Contributed by Ken Jin and Guido van Rossum in bpo-43766 [https://bugs.python.org/issue?@action=redirect&bpo=43766]. PEP written by Eric Traut.)

Other Language Changes

- The **`int`** type has a new method **`int.bit_count()`**, returning the number of ones in the binary expansion of a given integer, also known as the population count. (Contributed by Niklas Fiekas in bpo-29882 [https://bugs.python.org/issue?@action=redirect&bpo=29882].)
- The views returned by **`dict.keys()`**, **`dict.values()`** and **`dict.items()`** now all have a `mapping` attribute that gives a **`types.MappingProxyType`** object wrapping the original dictionary. (Contributed by Dennis Sweeney in bpo-40890 [https://bugs.python.org/issue?@action=redirect&bpo=40890].)
- **PEP 618** [https://peps.python.org/pep-0618/]: The **`zip()`** function now has an optional `strict` flag, used to require that all the iterables have an equal length.

- Builtin and extension functions that take integer arguments no longer accept `Decimal`s, `Fraction`s and other objects that can be converted to integers only with a loss (e.g. that have the `__int__()` method but do not have the `__index__()` method). (Contributed by Serhiy Storchaka in bpo-37999 [https://bugs.python.org/issue?@action=redirect&bpo=37999].)
- If `object.__ipow__()` returns `NotImplemented`, the operator will correctly fall back to `object.__pow__()` and `object.__rpow__()` as expected. (Contributed by Alex Shkop in bpo-38302 [https://bugs.python.org/issue?@action=redirect&bpo=38302].)
- Assignment expressions can now be used unparenthesized within set literals and set comprehensions, as well as in sequence indexes (but not slices).
- Functions have a new `__builtins__` attribute which is used to look for builtin symbols when a function is executed, instead of looking into `__globals__['__builtins__']`. The attribute is initialized from `__globals__["__builtins__"]` if it exists, else from the current builtins. (Contributed by Mark Shannon in bpo-42990 [https://bugs.python.org/issue?@action=redirect&bpo=42990].)
- Two new builtin functions – `aiter()` and `anext()` have been added to provide asynchronous counterparts to `iter()` and `next()`, respectively. (Contributed by Joshua Bronson, Daniel Pope, and Justin Wang in bpo-31861 [https://bugs.python.org/issue?@action=redirect&bpo=31861].)
- Static methods (`@staticmethod`) and class methods (`@classmethod`) now inherit the method attributes (`__module__`, `__name__`, `__qualname__`, `__doc__`, `__annotations__`) and have a new `__wrapped__` attribute. Moreover, static methods are now callable as regular functions. (Contributed by Victor Stinner in bpo-43682 [https://bugs.python.org/issue?@action=redirect&bpo=43682].)

- Annotations for complex targets (everything beside `simple name` targets defined by **PEP 526** [https://peps.python.org/pep-0526/]) no longer cause any runtime effects with `from __future__ import annotations`. (Contributed by Batuhan Taskaya in bpo-42737 [https://bugs.python.org/issue?@action=redirect&bpo=42737].)
- Class and module objects now lazy-create empty annotations dicts on demand. The annotations dicts are stored in the object's `__dict__` for backwards compatibility. This improves the best practices for working with `__annotations__`; for more information, please see Annotations Best Practices. (Contributed by Larry Hastings in bpo-43901 [https://bugs.python.org/issue?@action=redirect&bpo=43901].)
- Annotations consist of `yield`, `yield from`, `await` or named expressions are now forbidden under `from __future__ import annotations` due to their side effects. (Contributed by Batuhan Taskaya in bpo-42725 [https://bugs.python.org/issue?@action=redirect&bpo=42725].)
- Usage of unbound variables, `super()` and other expressions that might alter the processing of symbol table as annotations are now rendered effectless under `from __future__ import annotations`. (Contributed by Batuhan Taskaya in bpo-42725 [https://bugs.python.org/issue?@action=redirect&bpo=42725].)
- Hashes of NaN values of both **`float`** type and **`decimal.Decimal`** type now depend on object identity. Formerly, they always hashed to `0` even though NaN values are not equal to one another. This caused potentially quadratic runtime behavior due to excessive hash collisions when creating dictionaries and sets containing multiple NaNs. (Contributed by Raymond Hettinger in bpo-43475 [https://bugs.python.org/issue?@action=redirect&bpo=43475].)
- A **`SyntaxError`** (instead of a **`NameError`**) will be raised when deleting the **`__debug__`** constant. (Contributed by

Donghee Na in bpo-45000 [https://bugs.python.org/issue?@action=redirect&bpo=45000].)

- `SyntaxError` exceptions now have `end_lineno` and `end_offset` attributes. They will be `None` if not determined. (Contributed by Pablo Galindo in bpo-43914 [https://bugs.python.org/issue?@action=redirect&bpo=43914].)

New Modules

- None.

Improved Modules

asyncio

Add missing `connect_accepted_socket()` method. (Contributed by Alex Grönholm in bpo-41332 [https://bugs.python.org/issue?@action=redirect&bpo=41332].)

argparse

Misleading phrase “optional arguments” was replaced with “options” in argparse help. Some tests might require adaptation if they rely on exact output match. (Contributed by Raymond Hettinger in bpo-9694 [https://bugs.python.org/issue?@action=redirect&bpo=9694].)

array

The `index()` method of `array.array` now has optional *start* and *stop* parameters. (Contributed by Anders Lorentsen and Zackery Spytz in bpo-31956 [https://bugs.python.org/issue?@action=redirect&bpo=31956].)

asynchat, asyncore, smtpd

These modules have been marked as deprecated in their module documentation since Python 3.6. An import-time `DeprecationWarning` has now been added to all three of these modules.

base64

Add `base64.b32hexencode()` and `base64.b32hexdecode()` to support the Base32 Encoding with Extended Hex Alphabet.

bdb

Add **`clearBreakpoints()`** to reset all set breakpoints. (Contributed by Irit Katriel in bpo-24160 [https://bugs.python.org/issue?@action=redirect&bpo=24160].)

bisect

Added the possibility of providing a *key* function to the APIs in the `bisect` module. (Contributed by Raymond Hettinger in bpo-4356 [https://bugs.python.org/issue?@action=redirect&bpo=4356].)

codecs

Add a `codecs.unregister()` function to unregister a codec search function. (Contributed by Hai Shi in bpo-41842 [https://bugs.python.org/issue?@action=redirect&bpo=41842].)

collections.abc

The `__args__` of the parameterized generic for `collections.abc.Callable` are now consistent with `typing.Callable`. `collections.abc.Callable` generic now flattens type parameters, similar to what `typing.Callable` currently does. This means that `collections.abc.Callable[[int, str], str]` will have `__args__` of `(int, str, str)`; previously this was `([int, str], str)`. To allow this change, **`types.GenericAlias`** can now be subclassed, and a subclass will be returned when subscripting the **`collections.abc.Callable`** type. Note that a **`TypeError`** may be raised for invalid forms of parameterizing **`collections.abc.Callable`** which may have passed silently in Python 3.9. (Contributed by Ken Jin in bpo-42195 [https://bugs.python.org/issue?@action=redirect&bpo=42195].)

contextlib

Add a **`contextlib.aclosing()`** context manager to safely close async generators and objects representing asynchronously released resources. (Contributed by Joongi Kim and John Belmonte in bpo-41229 [https://bugs.python.org/issue?@action=redirect&bpo=41229].)

Add asynchronous context manager support to **`contextlib.nullcontext()`**. (Contributed by Tom Gringauz in bpo-41543 [https://bugs.python.org/issue?@action=redirect&bpo=41543].)

Add **`AsyncContextDecorator`**, for supporting usage of async context managers as decorators.

curses

The extended color functions added in ncurses 6.1 will be used transparently by **`curses.color_content()`**, **`curses.init_color()`**, **`curses.init_pair()`**, and **`curses.pair_content()`**. A new function, **`curses.has_extended_color_support()`**, indicates whether extended color support is provided by the underlying ncurses library. (Contributed by Jeffrey Kintscher and Hans Petter Jansson in bpo-36982 [https://bugs.python.org/issue?@action=redirect&bpo=36982].)

The `BUTTON5_*` constants are now exposed in the **`curses`** module if they are provided by the underlying curses library. (Contributed by Zackery Spytz in bpo-39273 [https://bugs.python.org/issue?@action=redirect&bpo=39273].)

dataclasses

__slots__

Added `slots` parameter in **`dataclasses.dataclass()`** decorator. (Contributed by Yurii Karabas in bpo-42269 [https://bugs.python.org/issue?@action=redirect&bpo=42269])

Keyword-only fields

dataclasses now supports fields that are keyword-only in the generated __init__ method. There are a number of ways of specifying keyword-only fields.

You can say that every field is keyword-only:

```
from dataclasses import dataclass

@dataclass(kw_only=True)
class Birthday:
    name: str
    birthday: datetime.date
```

Both `name` and `birthday` are keyword-only parameters to the generated __init__ method.

You can specify keyword-only on a per-field basis:

```
from dataclasses import dataclass, field

@dataclass
class Birthday:
    name: str
    birthday: datetime.date = field(kw_only=True)
```

Here only `birthday` is keyword-only. If you set `kw_only` on individual fields, be aware that there are rules about re-ordering fields due to keyword-only fields needing to follow non-keyword-only fields. See the full dataclasses documentation for details.

You can also specify that all fields following a KW_ONLY marker are keyword-only. This will probably be the most common usage:

```
from dataclasses import dataclass, KW_ONLY

@dataclass
class Point:
    x: float
    y: float
    _: KW_ONLY
    z: float = 0.0
    t: float = 0.0
```

Here, `z` and `t` are keyword-only parameters, while `x` and `y` are not. (Contributed by Eric V. Smith in bpo-43532 [https://bugs.python.org/issue?@action=redirect&bpo=43532].)

distutils

The entire `distutils` package is deprecated, to be removed in Python 3.12. Its functionality for specifying package builds has already been completely replaced by third-party packages `setuptools` and `packaging`, and most other commonly used APIs are available elsewhere in the standard library (such as **`platform`**, **`shutil`**, **`subprocess`** or **`sysconfig`**). There are no plans to migrate any other functionality from `distutils`, and applications that are using other functions should plan to make private copies of the code. Refer to **PEP 632** [https://peps.python.org/pep-0632/] for discussion.

The `bdist_wininst` command deprecated in Python 3.8 has been removed. The `bdist_wheel` command is now recommended to distribute binary packages on Windows. (Contributed by Victor Stinner in bpo-42802 [https://bugs.python.org/issue?@action=redirect&bpo=42802].)

doctest

When a module does not define `__loader__`, fall back to `__spec__.loader`. (Contributed by Brett Cannon in bpo-42133 [https://bugs.python.org/issue?@action=redirect&bpo=42133].)

encodings

`encodings.normalize_encoding()` now ignores non-ASCII characters. (Contributed by Hai Shi in bpo-39337 [https://bugs.python.org/issue?@action=redirect&bpo=39337].)

enum

`Enum` **`__repr__()`** now returns `enum_name.member_name` and **`__str__()`** now returns `member_name`. Stdlib enums available as module constants have a **`repr()`** of `module_name.member_name`. (Contributed by Ethan Furman in bpo-40066 [https://bugs.python.org/issue?@action=redirect&bpo=40066].)

Add **`enum.StrEnum`** for enums where all members are strings. (Contributed by Ethan Furman in bpo-41816 [https://bugs.python.org/issue?@action=redirect&bpo=41816].)

fileinput

Add *encoding* and *errors* parameters in `fileinput.input()` and `fileinput.FileInput`. (Contributed by Inada Naoki in bpo-43712 [https://bugs.python.org/issue?@action=redirect&bpo=43712].)

`fileinput.hook_compressed()` now returns `TextIOWrapper` object when *mode* is “r” and file is compressed, like uncompressed files. (Contributed by Inada Naoki in bpo-5758 [https://bugs.python.org/issue?@action=redirect&bpo=5758].)

faulthandler

The `faulthandler` module now detects if a fatal error occurs during a garbage collector collection. (Contributed by Victor Stinner in bpo-44466 [https://bugs.python.org/issue?@action=redirect&bpo=44466].)

gc

Add audit hooks for `gc.get_objects()`, `gc.get_referrers()` and `gc.get_referents()`. (Contributed by Pablo Galindo in bpo-43439 [https://bugs.python.org/issue?@action=redirect&bpo=43439].)

glob

Add the *root_dir* and *dir_fd* parameters in `glob()` and `iglob()` which allow to specify the root directory for searching. (Contributed by Serhiy Storchaka in bpo-38144 [https://bugs.python.org/issue?@action=redirect&bpo=38144].)

hashlib

The hashlib module requires OpenSSL 1.1.1 or newer. (Contributed by Christian Heimes in **PEP 644** [https://peps.python.org/pep-0644/] and bpo-43669 [https://bugs.python.org/issue?@action=redirect&bpo=43669].)

The hashlib module has preliminary support for OpenSSL 3.0.0. (Contributed by Christian Heimes in bpo-38820 [https://bugs.python.org/issue?@action=redirect&bpo=38820] and other issues.)

The pure-Python fallback of `pbkdf2_hmac()` is deprecated. In the future PBKDF2-HMAC will only be available when Python has been built with OpenSSL support. (Contributed by Christian Heimes in bpo-43880 [https://bugs.python.org/issue?@action=redirect&bpo=43880].)

hmac

The hmac module now uses OpenSSL's HMAC implementation internally. (Contributed by Christian Heimes in bpo-40645 [https://bugs.python.org/issue?@action=redirect&bpo=40645].)

IDLE and idlelib

Make IDLE invoke `sys.excepthook()` (when started without '-n'). User hooks were previously ignored. (Contributed by Ken Hilton in bpo-43008 [https://bugs.python.org/issue?@action=redirect&bpo=43008].)

Rearrange the settings dialog. Split the General tab into Windows and Shell/Ed tabs. Move help sources, which extend the Help menu, to the Extensions tab. Make space for new options and shorten the dialog. The latter makes the dialog better fit small screens. (Contributed by Terry Jan Reedy in bpo-40468 [https://bugs.python.org/issue?@action=redirect&bpo=40468].) Move the indent space setting from the Font tab to the new Windows tab. (Contributed by Mark Roseman and Terry Jan Reedy in bpo-33962 [https://bugs.python.org/issue?@action=redirect&bpo=33962].)

The changes above were backported to a 3.9 maintenance release.

Add a Shell sidebar. Move the primary prompt ('>>>') to the sidebar. Add secondary prompts ('…') to the sidebar. Left click and optional drag selects one or more lines of text, as with the editor line number sidebar. Right click after selecting text lines displays a context menu with 'copy with prompts'. This zips together prompts from the sidebar with lines from the selected text. This option also appears on the context menu for the text. (Contributed by Tal Einat in bpo-37903 [https://bugs.python.org/issue?@action=redirect&bpo=37903].)

Use spaces instead of tabs to indent interactive code. This makes interactive code entries 'look right'. Making this feasible was a major motivation for adding the shell sidebar. (Contributed by Terry Jan Reedy in bpo-37892 [https://bugs.python.org/issue?@action=redirect&bpo=37892].)

Highlight the new soft keywords **`match`**, **`case`**, and **`_`** in pattern-matching statements. However, this highlighting is not perfect and will be incorrect in some rare cases, including some `_`-s in `case` patterns. (Contributed by Tal Einat in bpo-44010 [https://bugs.python.org/issue?@action=redirect&bpo=44010].)

New in 3.10 maintenance releases.

Apply syntax highlighting to `.pyi` files. (Contributed by Alex Waygood and Terry Jan Reedy in bpo-45447 [https://bugs.python.org/issue?@action=redirect&bpo=45447].)

Include prompts when saving Shell with inputs and outputs. (Contributed by Terry Jan Reedy in gh-95191 [https://github.com/python/cpython/issues/95191].)

importlib.metadata

Feature parity with `importlib_metadata` 4.6 (history [https://importlib-metadata.readthedocs.io/en/latest/history.html]).

importlib.metadata entry points now provide a nicer experience for selecting entry points by group and name through a new importlib.metadata.EntryPoints class. See the Compatibility Note in the docs for more info on the deprecation and usage.

Added importlib.metadata.packages_distributions() for resolving top-level Python modules and packages to their importlib.metadata.Distribution.

inspect

When a module does not define `__loader__`, fall back to `__spec__.loader`. (Contributed by Brett Cannon in bpo-42133 [https://bugs.python.org/issue?@action=redirect&bpo=42133].)

Add **`inspect.get_annotations()`**, which safely computes the annotations defined on an object. It works around the quirks of accessing the annotations on various types of objects, and makes very few assumptions about the object it examines. **`inspect.get_annotations()`** can also correctly un-stringize stringized annotations. **`inspect.get_annotations()`** is now considered best practice for accessing the annotations dict defined on any Python object; for more information on best practices for working with annotations, please see Annotations Best Practices. Relatedly, **`inspect.signature()`**, **`inspect.Signature.from_callable()`**, and **`inspect.Signature.from_function()`** now call **`inspect.get_annotations()`** to retrieve annotations. This means **`inspect.signature()`** and **`inspect.Signature.from_callable()`** can also now un-stringize stringized annotations. (Contributed by Larry Hastings in bpo-43817 [https://bugs.python.org/issue?@action=redirect&bpo=43817].)

itertools

Add `itertools.pairwise()`. (Contributed by Raymond Hettinger in bpo-38200 [https://bugs.python.org/issue?@action=redirect&bpo=38200].)

linecache

When a module does not define `__loader__`, fall back to `__spec__.loader`. (Contributed by Brett Cannon in bpo-42133 [https://bugs.python.org/issue?@action=redirect&bpo=42133].)

os

Add `os.cpu_count()` support for VxWorks RTOS. (Contributed by Peixing Xin in bpo-41440 [https://bugs.python.org/issue?@action=redirect&bpo=41440].)

Add a new function `os.eventfd()` and related helpers to wrap the `eventfd2` syscall on Linux. (Contributed by Christian Heimes in bpo-41001 [https://bugs.python.org/issue?@action=redirect&bpo=41001].)

Add `os.splice()` that allows to move data between two file descriptors without copying between kernel address space and user address space, where one of the file descriptors must refer to a pipe. (Contributed by Pablo Galindo in bpo-41625 [https://bugs.python.org/issue?@action=redirect&bpo=41625].)

Add `O_EVTONLY`, `O_FSYNC`, `O_SYMLINK` and `O_NOFOLLOW_ANY` for macOS. (Contributed by Donghee Na in bpo-43106 [https://bugs.python.org/issue?@action=redirect&bpo=43106].)

os.path

`os.path.realpath()` now accepts a *strict* keyword-only argument. When set to `True`, **`OSError`** is raised if a path doesn’t exist or a symlink loop is encountered. (Contributed by Barney Gale in bpo-43757 [https://bugs.python.org/issue?@action=redirect&bpo=43757].)

pathlib

Add slice support to **`PurePath.parents`**. (Contributed by Joshua Cannon in bpo-35498 [https://bugs.python.org/issue?@action=redirect&bpo=35498].)

Add negative indexing support to **`PurePath.parents`**. (Contributed by Yaroslav Pankovych in bpo-21041 [https://bugs.python.org/issue?@action=redirect&bpo=21041].)

Add **`Path.hardlink_to`** method that supersedes **`link_to()`**. The new method has the same argument order as **`symlink_to()`**. (Contributed by Barney Gale in bpo-39950 [https://bugs.python.org/issue?@action=redirect&bpo=39950].)

`pathlib.Path.stat()` and **`chmod()`** now accept a *follow_symlinks* keyword-only argument for consistency with corresponding functions in the **`os`** module. (Contributed by Barney Gale in bpo-39906 [https://bugs.python.org/issue?@action=redirect&bpo=39906].)

platform

Add **`platform.freedesktop_os_release()`** to retrieve operation system identification from freedesktop.org os-release [https://www.freedesktop.org/software/systemd/man/os-release.html] standard file. (Contributed by Christian Heimes in bpo-28468 [https://bugs.python.org/issue?@action=redirect&bpo=28468].)

pprint

`pprint.pprint()` now accepts a new `underscore_numbers` keyword argument. (Contributed by sblondon in bpo-42914 [https://bugs.python.org/issue?@action=redirect&bpo=42914].)

`pprint` can now pretty-print `dataclasses.dataclass` instances. (Contributed by Lewis Gaul in bpo-43080 [https://bugs.python.org/issue?@action=redirect&bpo=43080].)

py_compile

Add `--quiet` option to command-line interface of `py_compile`. (Contributed by Gregory Schevchenko in bpo-38731 [https://bugs.python.org/issue?@action=redirect&bpo=38731].)

pyclbr

Add an `end_lineno` attribute to the `Function` and `Class` objects in the tree returned by `pyclbr.readmodule()` and `pyclbr.readmodule_ex()`. It matches the existing (start) `lineno`. (Contributed by Aviral Srivastava in bpo-38307 [https://bugs.python.org/issue?@action=redirect&bpo=38307].)

shelve

The `shelve` module now uses `pickle.DEFAULT_PROTOCOL` by default instead of `pickle` protocol `3` when creating shelves. (Contributed by Zackery Spytz in bpo-34204 [https://bugs.python.org/issue?@action=redirect&bpo=34204].)

statistics

Add `covariance()`, Pearson’s `correlation()`, and simple `linear_regression()` functions. (Contributed by Tymoteusz Wołodźko in bpo-38490 [https://bugs.python.org/issue?@action=redirect&bpo=38490].)

site

When a module does not define `__loader__`, fall back to `__spec__.loader`. (Contributed by Brett Cannon in bpo-42133 [https://bugs.python.org/issue?@action=redirect&bpo=42133].)

socket

The exception **`socket.timeout`** is now an alias of **`TimeoutError`**. (Contributed by Christian Heimes in bpo-42413 [https://bugs.python.org/issue?@action=redirect&bpo=42413].)

Add option to create MPTCP sockets with `IPPROTO_MPTCP` (Contributed by Rui Cunha in bpo-43571 [https://bugs.python.org/issue?@action=redirect&bpo=43571].)

Add `IP_RECVTOS` option to receive the type of service (ToS) or DSCP/ECN fields (Contributed by Georg Sauthoff in bpo-44077 [https://bugs.python.org/issue?@action=redirect&bpo=44077].)

ssl

The ssl module requires OpenSSL 1.1.1 or newer. (Contributed by Christian Heimes in **PEP 644** [https://peps.python.org/pep-0644/] and bpo-43669 [https://bugs.python.org/issue?@action=redirect&bpo=43669].)

The ssl module has preliminary support for OpenSSL 3.0.0 and new option **`OP_IGNORE_UNEXPECTED_EOF`**. (Contributed by Christian Heimes in bpo-38820 [https://bugs.python.org/issue?@action=redirect&bpo=38820], bpo-43794 [https://bugs.python.org/issue?@action=redirect&bpo=43794], bpo-43788 [https://bugs.python.org/issue?@action=redirect&bpo=43788], bpo-43791 [https://bugs.python.org/issue?@action=redirect&bpo=43791], bpo-43799 [https://bugs.python.org/issue?@action=redirect&bpo=43799], bpo-43920 [https://bugs.python.org/issue?@action=redirect&bpo=43920], bpo-43789 [https://bugs.python.org/issue?

@action=redirect&bpo=43789], and bpo-43811 [https://bugs.python.org/issue?@action=redirect&bpo=43811].)

Deprecated function and use of deprecated constants now result in a **`DeprecationWarning`**. **`ssl.SSLContext.options`** has **`OP_NO_SSLv2`** and **`OP_NO_SSLv3`** set by default and therefore cannot warn about setting the flag again. The deprecation section has a list of deprecated features. (Contributed by Christian Heimes in bpo-43880 [https://bugs.python.org/issue?@action=redirect&bpo=43880].)

The ssl module now has more secure default settings. Ciphers without forward secrecy or SHA-1 MAC are disabled by default. Security level 2 prohibits weak RSA, DH, and ECC keys with less than 112 bits of security. **`SSLContext`** defaults to minimum protocol version TLS 1.2. Settings are based on Hynek Schlawack's research. (Contributed by Christian Heimes in bpo-43998 [https://bugs.python.org/issue?@action=redirect&bpo=43998].)

The deprecated protocols SSL 3.0, TLS 1.0, and TLS 1.1 are no longer officially supported. Python does not block them actively. However OpenSSL build options, distro configurations, vendor patches, and cipher suites may prevent a successful handshake.

Add a *timeout* parameter to the **`ssl.get_server_certificate()`** function. (Contributed by Zackery Spytz in bpo-31870 [https://bugs.python.org/issue?@action=redirect&bpo=31870].)

The ssl module uses heap-types and multi-phase initialization. (Contributed by Christian Heimes in bpo-42333 [https://bugs.python.org/issue?@action=redirect&bpo=42333].)

A new verify flag **`VERIFY_X509_PARTIAL_CHAIN`** has been added. (Contributed by l0x in bpo-40849 [https://bugs.python.org/issue?@action=redirect&bpo=40849].)

sqlite3

Add audit events for `connect()`, `enable_load_extension()`, and `load_extension()`. (Contributed by Erlend E. Aasland in bpo-43762 [https://bugs.python.org/issue?@action=redirect&bpo=43762].)

sys

Add `sys.orig_argv` attribute: the list of the original command line arguments passed to the Python executable. (Contributed by Victor Stinner in bpo-23427 [https://bugs.python.org/issue?@action=redirect&bpo=23427].)

Add `sys.stdlib_module_names`, containing the list of the standard library module names. (Contributed by Victor Stinner in bpo-42955 [https://bugs.python.org/issue?@action=redirect&bpo=42955].)

_thread

`_thread.interrupt_main()` now takes an optional signal number to simulate (the default is still `signal.SIGINT`). (Contributed by Antoine Pitrou in bpo-43356 [https://bugs.python.org/issue?@action=redirect&bpo=43356].)

threading

Add `threading.gettrace()` and `threading.getprofile()` to retrieve the functions set by `threading.settrace()` and `threading.setprofile()` respectively. (Contributed by Mario Corchero in bpo-42251 [https://bugs.python.org/issue?@action=redirect&bpo=42251].)

Add `threading.__excepthook__` to allow retrieving the original value of `threading.excepthook()` in case it is set to a broken or a different value. (Contributed by Mario Corchero in bpo-42308 [https://bugs.python.org/issue?@action=redirect&bpo=42308].)

traceback

The **`format_exception()`**, **`format_exception_only()`**, and **`print_exception()`** functions can now take an exception object as a positional-only argument. (Contributed by Zackery Spytz and Matthias Bussonnier in bpo-26389 [https://bugs.python.org/issue?@action=redirect&bpo=26389].)

types

Reintroduce the **`types.EllipsisType`**, **`types.NoneType`** and **`types.NotImplementedType`** classes, providing a new set of types readily interpretable by type checkers. (Contributed by Bas van Beek in bpo-41810 [https://bugs.python.org/issue?@action=redirect&bpo=41810].)

typing

For major changes, see New Features Related to Type Hints.

The behavior of **`typing.Literal`** was changed to conform with **PEP 586** [https://peps.python.org/pep-0586/] and to match the behavior of static type checkers specified in the PEP.

1. `Literal` now de-duplicates parameters.
2. Equality comparisons between `Literal` objects are now order independent.
3. `Literal` comparisons now respect types. For example, `Literal[0] == Literal[False]` previously evaluated to `True`. It is now `False`. To support this change, the internally used type cache now supports differentiating types.

4. `Literal` objects will now raise a **`TypeError`** exception during equality comparisons if any of their parameters are not hashable. Note that declaring `Literal` with unhashable parameters will not throw an error:

   ```
   >>> from typing import Literal
   >>> Literal[{0}]
   >>> Literal[{0}] == Literal[{False}]
   Traceback (most recent call last):
     File "<stdin>", line 1, in <module>
   TypeError: unhashable type: 'set'
   ```

(Contributed by Yurii Karabas in bpo-42345 [https://bugs.python.org/issue?@action=redirect&bpo=42345].)

Add new function **`typing.is_typeddict()`** to introspect if an annotation is a **`typing.TypedDict`**. (Contributed by Patrick Reader in bpo-41792 [https://bugs.python.org/issue?@action=redirect&bpo=41792].)

Subclasses of `typing.Protocol` which only have data variables declared will now raise a `TypeError` when checked with `isinstance` unless they are decorated with **`runtime_checkable()`**. Previously, these checks passed silently. Users should decorate their subclasses with the **`runtime_checkable()`** decorator if they want runtime protocols. (Contributed by Yurii Karabas in bpo-38908 [https://bugs.python.org/issue?@action=redirect&bpo=38908].)

Importing from the `typing.io` and `typing.re` submodules will now emit **`DeprecationWarning`**. These submodules have been deprecated since Python 3.8 and will be removed in a future version of Python. Anything belonging to those submodules should be imported directly from **`typing`** instead. (Contributed by Sebastian Rittau in bpo-38291 [https://bugs.python.org/issue?@action=redirect&bpo=38291].)

unittest

Add new method `assertNoLogs()` to complement the existing `assertLogs()`. (Contributed by Kit Yan Choi in bpo-39385 [https://bugs.python.org/issue?@action=redirect&bpo=39385].)

urllib.parse

Python versions earlier than Python 3.10 allowed using both `;` and `&` as query parameter separators in `urllib.parse.parse_qs()` and `urllib.parse.parse_qsl()`. Due to security concerns, and to conform with newer W3C recommendations, this has been changed to allow only a single separator key, with `&` as the default. This change also affects **`cgi.parse()`** and **`cgi.parse_multipart()`** as they use the affected functions internally. For more details, please see their respective documentation. (Contributed by Adam Goldschmidt, Senthil Kumaran and Ken Jin in bpo-42967 [https://bugs.python.org/issue?@action=redirect&bpo=42967].)

The presence of newline or tab characters in parts of a URL allows for some forms of attacks. Following the WHATWG specification that updates **RFC 3986** [https://datatracker.ietf.org/doc/html/rfc3986.html], ASCII newline `\n`, `\r` and tab `\t` characters are stripped from the URL by the parser in `urllib.parse` preventing such attacks. The removal characters are controlled by a new module level variable `urllib.parse._UNSAFE_URL_BYTES_TO_REMOVE`. (See gh-88048 [https://github.com/python/cpython/issues/88048])

xml

Add a `LexicalHandler` class to the `xml.sax.handler` module. (Contributed by Jonathan Gossage and Zackery Spytz in bpo-35018 [https://bugs.python.org/issue?@action=redirect&bpo=35018].)

zipimport

Add methods related to **PEP 451** [https://peps.python.org/pep-0451/]: **`find_spec()`**, **`zipimport.zipimporter.create_module()`**, and **`zipimport.zipimporter.exec_module()`**. (Contributed by Brett Cannon in bpo-42131 [https://bugs.python.org/issue?@action=redirect&bpo=42131].)

Add **`invalidate_caches()`** method. (Contributed by Desmond Cheong in bpo-14678 [https://bugs.python.org/issue?@action=redirect&bpo=14678].)

Optimizations

- Constructors **`str()`**, **`bytes()`** and **`bytearray()`** are now faster (around 30–40% for small objects). (Contributed by Serhiy Storchaka in bpo-41334 [https://bugs.python.org/issue?@action=redirect&bpo=41334].)
- The **`runpy`** module now imports fewer modules. The `python3 -m module-name` command startup time is 1.4x faster in average. On Linux, `python3 -I -m module-name` imports 69 modules on Python 3.9, whereas it only imports 51 modules (-18) on Python 3.10. (Contributed by Victor Stinner in bpo-41006 [https://bugs.python.org/issue?@action=redirect&bpo=41006] and bpo-41718 [https://bugs.python.org/issue?@action=redirect&bpo=41718].)
- The `LOAD_ATTR` instruction now uses new “per opcode cache” mechanism. It is about 36% faster now for regular attributes and 44% faster for slots. (Contributed by Pablo Galindo and Yury Selivanov in bpo-42093 [https://bugs.python.org/issue?@action=redirect&bpo=42093] and Guido van Rossum in bpo-42927 [https://bugs.python.org/issue?@action=redirect&bpo=42927], based on ideas implemented originally in PyPy and MicroPython.)

- When building Python with **`--enable-optimizations`** now `-fno-semantic-interposition` is added to both the compile and link line. This speeds builds of the Python interpreter created with **`--enable-shared`** with `gcc` by up to 30%. See this article [https://developers.redhat.com/blog/2020/06/25/red-hat-enterprise-linux-8-2-brings-faster-python-3-8-run-speeds/] for more details. (Contributed by Victor Stinner and Pablo Galindo in bpo-38980 [https://bugs.python.org/issue?@action=redirect&bpo=38980].)
- Use a new output buffer management code for **`bz2`** / **`lzma`** / **`zlib`** modules, and add `.readall()` function to `_compression.DecompressReader` class. bz2 decompression is now 1.09x ~ 1.17x faster, lzma decompression 1.20x ~ 1.32x faster, `GzipFile.read(-1)` 1.11x ~ 1.18x faster. (Contributed by Ma Lin, reviewed by Gregory P. Smith, in bpo-41486 [https://bugs.python.org/issue?@action=redirect&bpo=41486])
- When using stringized annotations, annotations dicts for functions are no longer created when the function is created. Instead, they are stored as a tuple of strings, and the function object lazily converts this into the annotations dict on demand. This optimization cuts the CPU time needed to define an annotated function by half. (Contributed by Yurii Karabas and Inada Naoki in bpo-42202 [https://bugs.python.org/issue?@action=redirect&bpo=42202].)
- Substring search functions such as `str1 in str2` and `str2.find(str1)` now sometimes use Crochemore & Perrin’s “Two-Way” string searching algorithm to avoid quadratic behavior on long strings. (Contributed by Dennis Sweeney in bpo-41972 [https://bugs.python.org/issue?@action=redirect&bpo=41972])
- Add micro-optimizations to `_PyType_Lookup()` to improve type attribute cache lookup performance in the common case of cache hits. This makes the interpreter 1.04 times faster on average. (Contributed by Dino Viehland

in bpo-43452 [https://bugs.python.org/issue?@action=redirect&bpo=43452].)

- The following built-in functions now support the faster **PEP 590** [https://peps.python.org/pep-0590/] vectorcall calling convention: `map()`, `filter()`, `reversed()`, `bool()` and `float()`. (Contributed by Donghee Na and Jeroen Demeyer in bpo-43575 [https://bugs.python.org/issue?@action=redirect&bpo=43575], bpo-43287 [https://bugs.python.org/issue?@action=redirect&bpo=43287], bpo-41922 [https://bugs.python.org/issue?@action=redirect&bpo=41922], bpo-41873 [https://bugs.python.org/issue?@action=redirect&bpo=41873] and bpo-41870 [https://bugs.python.org/issue?@action=redirect&bpo=41870].)
- `BZ2File` performance is improved by removing internal `RLock`. This makes `BZ2File` thread unsafe in the face of multiple simultaneous readers or writers, just like its equivalent classes in `gzip` and `lzma` have always been. (Contributed by Inada Naoki in bpo-43785 [https://bugs.python.org/issue?@action=redirect&bpo=43785].)

Deprecated

- Currently Python accepts numeric literals immediately followed by keywords, for example `0in x`, `1or x`, `0if 1else 2`. It allows confusing and ambiguous expressions like `[0x1for x in y]` (which can be interpreted as `[0x1 for x in y]` or `[0x1f or x in y]`). Starting in this release, a deprecation warning is raised if the numeric literal is immediately followed by one of keywords `and`, `else`, `for`, `if`, `in`, `is` and `or`. In future releases it will be changed to syntax warning, and finally to syntax error. (Contributed by Serhiy Storchaka in bpo-43833 [https://bugs.python.org/issue?@action=redirect&bpo=43833].)

- Starting in this release, there will be a concerted effort to begin cleaning up old import semantics that were kept for Python 2.7 compatibility. Specifically, **`find_loader()`/`find_module()`** (superseded by [`find_spec()`]), [`load_module()`] (superseded by [`exec_module()`]), **`module_repr()`** (which the import system takes care of for you), the `__package__` attribute (superseded by `__spec__.parent`), the `__loader__` attribute (superseded by `__spec__.loader`), and the `__cached__` attribute (superseded by `__spec__.cached`) will slowly be removed (as well as other classes and methods in [`importlib`]). [`ImportWarning`] and/or [`DeprecationWarning`] will be raised as appropriate to help identify code which needs updating during this transition.

- The entire `distutils` namespace is deprecated, to be removed in Python 3.12. Refer to the module changes section for more information.

- Non-integer arguments to [`random.randrange()`] are deprecated. The [`ValueError`] is deprecated in favor of a [`TypeError`]. (Contributed by Serhiy Storchaka and Raymond Hettinger in bpo-37319 [https://bugs.python.org/issue?@action=redirect&bpo=37319].)

- The various `load_module()` methods of [`importlib`] have been documented as deprecated since Python 3.6, but will now also trigger a [`DeprecationWarning`]. Use [`exec_module()`] instead. (Contributed by Brett Cannon in bpo-26131 [https://bugs.python.org/issue?@action=redirect&bpo=26131].)

- **`zimport.zipimporter.load_module()`** has been deprecated in preference for [`exec_module()`]. (Contributed by Brett Cannon in bpo-26131 [https://bugs.python.org/issue?@action=redirect&bpo=26131].)

- The use of `load_module()` by the import system now triggers an `ImportWarning` as `exec_module()` is preferred. (Contributed by Brett Cannon in bpo-26131 [https://bugs.python.org/issue?@action=redirect&bpo=26131].)

- The use of `importlib.abc.MetaPathFinder.find_module()` and `importlib.abc.PathEntryFinder.find_module()` by the import system now trigger an `ImportWarning` as `importlib.abc.MetaPathFinder.find_spec()` and `importlib.abc.PathEntryFinder.find_spec()` are preferred, respectively. You can use `importlib.util.spec_from_loader()` to help in porting. (Contributed by Brett Cannon in bpo-42134 [https://bugs.python.org/issue?@action=redirect&bpo=42134].)

- The use of `importlib.abc.PathEntryFinder.find_loader()` by the import system now triggers an `ImportWarning` as `importlib.abc.PathEntryFinder.find_spec()` is preferred. You can use `importlib.util.spec_from_loader()` to help in porting. (Contributed by Brett Cannon in bpo-43672 [https://bugs.python.org/issue?@action=redirect&bpo=43672].)

- The various implementations of `importlib.abc.MetaPathFinder.find_module()` (`importlib.machinery.BuiltinImporter.find_module()`, `importlib.machinery.FrozenImporter.find_module()`, `importlib.machinery.WindowsRegistryFinder.find_module()`, `importlib.machinery.PathFinder.find_module()`, `importlib.abc.MetaPathFinder.find_module()`), `importlib.abc.PathEntryFinder.find_module()` (`importlib.machinery.FileFinder.find_module()`), and `importlib.abc.PathEntryFinder.find_loader()` (`importlib.machinery.FileFinder.find_loader()`) now raise `DeprecationWarning` and are slated for removal in Python 3.12 (previously they were documented as deprecated in Python 3.4). (Contributed by Brett Cannon in bpo-42135 [https://bugs.python.org/issue?@action=redirect&bpo=42135].)

- **`importlib.abc.Finder`** is deprecated (including its sole method, **`find_module()`**). Both **`importlib.abc.MetaPathFinder`** and **`importlib.abc.PathEntryFinder`** no longer inherit from the class. Users should inherit from one of these two classes as appropriate instead. (Contributed by Brett Cannon in bpo-42135 [https://bugs.python.org/issue?@action=redirect&bpo=42135].)

- The deprecations of **`imp`**, **`importlib.find_loader()`**, **`importlib.util.set_package_wrapper()`**, **`importlib.util.set_loader_wrapper()`**, **`importlib.util.module_for_loader()`**, **`pkgutil.ImpImporter`**, and **`pkgutil.ImpLoader`** have all been updated to list Python 3.12 as the slated version of removal (they began raising **`DeprecationWarning`** in previous versions of Python). (Contributed by Brett Cannon in bpo-43720 [https://bugs.python.org/issue?@action=redirect&bpo=43720].)

- The import system now uses the `__spec__` attribute on modules before falling back on **`module_repr()`** for a module’s `__repr__()` method. Removal of the use of `module_repr()` is scheduled for Python 3.12. (Contributed by Brett Cannon in bpo-42137 [https://bugs.python.org/issue?@action=redirect&bpo=42137].)

- **`importlib.abc.Loader.module_repr()`**, **`importlib.machinery.FrozenLoader.module_repr()`**, and **`importlib.machinery.BuiltinLoader.module_repr()`** are deprecated and slated for removal in Python 3.12. (Contributed by Brett Cannon in bpo-42136 [https://bugs.python.org/issue?@action=redirect&bpo=42136].)

- `sqlite3.OptimizedUnicode` has been undocumented and obsolete since Python 3.3, when it was made an alias to **`str`**. It is now deprecated, scheduled for removal in Python 3.12. (Contributed by Erlend E. Aasland in bpo-42264 [https://bugs.python.org/issue?@action=redirect&bpo=42264].)

- The undocumented built-in function `sqlite3.enable_shared_cache` is now deprecated, scheduled for removal in Python 3.12. Its use is strongly discouraged by the SQLite3 documentation. See the SQLite3 docs [https://sqlite.org/c3ref/enable_shared_cache.html] for more details. If a shared cache must be used, open the database in URI mode using the `cache=shared` query parameter. (Contributed by Erlend E. Aasland in bpo-24464 [https://bugs.python.org/issue?@action=redirect&bpo=24464].)

- The following `threading` methods are now deprecated:

    - `threading.currentThread` => **`threading.current_thread()`**
    - `threading.activeCount` => **`threading.active_count()`**
    - `threading.Condition.notifyAll` => **`threading.Condition.notify_all()`**
    - `threading.Event.isSet` => **`threading.Event.is_set()`**
    - `threading.Thread.setName` => **`threading.Thread.name`**
    - `threading.thread.getName` => **`threading.Thread.name`**
    - `threading.Thread.isDaemon` => **`threading.Thread.daemon`**
    - `threading.Thread.setDaemon` => **`threading.Thread.daemon`**

  (Contributed by Jelle Zijlstra in gh-87889 [https://github.com/python/cpython/issues/87889].)

- **`pathlib.Path.link_to()`** is deprecated and slated for removal in Python 3.12. Use **`pathlib.Path.hardlink_to()`** instead. (Contributed by Barney Gale in bpo-39950 [https://bugs.python.org/issue?@action=redirect&bpo=39950].)

- `cgi.log()` is deprecated and slated for removal in Python 3.12. (Contributed by Inada Naoki in bpo-41139 [https://bugs.python.org/issue?@action=redirect&bpo=41139].)

- The following **`ssl`** features have been deprecated since Python 3.6, Python 3.7, or OpenSSL 1.1.0 and will be removed in 3.11:

  - **`OP_NO_SSLv2`**, **`OP_NO_SSLv3`**, **`OP_NO_TLSv1`**, **`OP_NO_TLSv1_1`**, **`OP_NO_TLSv1_2`**, and **`OP_NO_TLSv1_3`** are replaced by **`minimum_version`** and **`maximum_version`**.
  - **`PROTOCOL_SSLv2`**, **`PROTOCOL_SSLv3`**, **`PROTOCOL_SSLv23`**, **`PROTOCOL_TLSv1`**, **`PROTOCOL_TLSv1_1`**, **`PROTOCOL_TLSv1_2`**, and **`PROTOCOL_TLS`** are deprecated in favor of **`PROTOCOL_TLS_CLIENT`** and **`PROTOCOL_TLS_SERVER`**
  - **`wrap_socket()`** is replaced by **`ssl.SSLContext.wrap_socket()`**
  - **`match_hostname()`**
  - **`RAND_pseudo_bytes()`**, **`RAND_egd()`**
  - NPN features like **`ssl.SSLSocket.selected_npn_protocol()`** and **`ssl.SSLContext.set_npn_protocols()`** are replaced by ALPN.

- The threading debug (**`PYTHONTHREADDEBUG`** environment variable) is deprecated in Python 3.10 and will be removed in Python 3.12. This feature requires a debug build of Python. (Contributed by Victor Stinner in bpo-44584 [https://bugs.python.org/issue?@action=redirect&bpo=44584].)

- Importing from the `typing.io` and `typing.re` submodules will now emit **`DeprecationWarning`**. These submodules will be removed in a future version of Python. Anything belonging to these submodules should be imported directly from **`typing`** instead. (Contributed by Sebastian

Rittau in bpo-38291 [https://bugs.python.org/issue?@action=redirect&bpo=38291].)

Removed

- Removed special methods `__int__`, `__float__`, `__floordiv__`, `__mod__`, `__divmod__`, `__rfloordiv__`, `__rmod__` and `__rdivmod__` of the **`complex`** class. They always raised a **`TypeError`**. (Contributed by Serhiy Storchaka in bpo-41974 [https://bugs.python.org/issue?@action=redirect&bpo=41974].)
- The `ParserBase.error()` method from the private and undocumented `_markupbase` module has been removed. **`html.parser.HTMLParser`** is the only subclass of `ParserBase` and its `error()` implementation was already removed in Python 3.5. (Contributed by Berker Peksag in bpo-31844 [https://bugs.python.org/issue?@action=redirect&bpo=31844].)
- Removed the `unicodedata.ucnhash_CAPI` attribute which was an internal PyCapsule object. The related private `_PyUnicode_Name_CAPI` structure was moved to the internal C API. (Contributed by Victor Stinner in bpo-42157 [https://bugs.python.org/issue?@action=redirect&bpo=42157].)
- Removed the `parser` module, which was deprecated in 3.9 due to the switch to the new PEG parser, as well as all the C source and header files that were only being used by the old parser, including `node.h`, `parser.h`, `graminit.h` and `grammar.h`.
- Removed the Public C API functions `PyParser_SimpleParseStringFlags`, `PyParser_SimpleParseStringFlagsFilename`, `PyParser_SimpleParseFileFlags` and `PyNode_Compile` that were deprecated in 3.9 due to the switch to the new PEG parser.

- Removed the `formatter` module, which was deprecated in Python 3.4. It is somewhat obsolete, little used, and not tested. It was originally scheduled to be removed in Python 3.6, but such removals were delayed until after Python 2.7 EOL. Existing users should copy whatever classes they use into their code. (Contributed by Donghee Na and Terry J. Reedy in bpo-42299 [https://bugs.python.org/issue?@action=redirect&bpo=42299].)

- Removed the **`PyModule_GetWarningsModule()`** function that was useless now due to the **`_warnings`** module was converted to a builtin module in 2.6. (Contributed by Hai Shi in bpo-42599 [https://bugs.python.org/issue?@action=redirect&bpo=42599].)

- Remove deprecated aliases to Collections Abstract Base Classes from the **`collections`** module. (Contributed by Victor Stinner in bpo-37324 [https://bugs.python.org/issue?@action=redirect&bpo=37324].)

- The `loop` parameter has been removed from most of **`asyncio`**‘s high-level API following deprecation in Python 3.8. The motivation behind this change is multifold:

  1. This simplifies the high-level API.
  2. The functions in the high-level API have been implicitly getting the current thread’s running event loop since Python 3.7. There isn’t a need to pass the event loop to the API in most normal use cases.
  3. Event loop passing is error-prone especially when dealing with loops running in different threads.

  Note that the low-level API will still accept `loop`. See Changes in the Python API for examples of how to replace existing code.

(Contributed by Yurii Karabas, Andrew Svetlov, Yury Selivanov and Kyle Stanley in bpo-42392 [https://bugs.python.org/issue?@action=redirect&bpo=42392].)

Porting to Python 3.10

This section lists previously described changes and other bugfixes that may require changes to your code.

Changes in the Python syntax

- Deprecation warning is now emitted when compiling previously valid syntax if the numeric literal is immediately followed by a keyword (like in `0in x`). In future releases it will be changed to syntax warning, and finally to a syntax error. To get rid of the warning and make the code compatible with future releases just add a space between the numeric literal and the following keyword. (Contributed by Serhiy Storchaka in bpo-43833 [https://bugs.python.org/issue?@action=redirect&bpo=43833].)

Changes in the Python API

- The *etype* parameters of the **`format_exception()`**, **`format_exception_only()`**, and **`print_exception()`** functions in the **`traceback`** module have been renamed to *exc*. (Contributed by Zackery Spytz and Matthias Bussonnier in bpo-26389 [https://bugs.python.org/issue?@action=redirect&bpo=26389].)

- **`atexit`**: At Python exit, if a callback registered with **`atexit.register()`** fails, its exception is now logged. Previously, only some exceptions were logged, and the last exception was always silently ignored. (Contributed by Victor Stinner in bpo-42639 [https://bugs.python.org/issue?@action=redirect&bpo=42639].)

- `collections.abc.Callable` generic now flattens type parameters, similar to what `typing.Callable` currently does. This means that `collections.abc.Callable[[int, str], str]` will have `__args__` of `(int, str, str)`; previously this was `([int, str], str)`. Code which accesses the arguments via `typing.get_args()` or `__args__` need to account for this change. Furthermore, `TypeError` may be raised for invalid forms of parameterizing `collections.abc.Callable` which may have passed silently in Python 3.9. (Contributed by Ken Jin in bpo-42195 [https://bugs.python.org/issue?@action=redirect&bpo=42195].)
- `socket.htons()` and `socket.ntohs()` now raise `OverflowError` instead of `DeprecationWarning` if the given parameter will not fit in a 16-bit unsigned integer. (Contributed by Erlend E. Aasland in bpo-42393 [https://bugs.python.org/issue?@action=redirect&bpo=42393].)
- The `loop` parameter has been removed from most of `asyncio`‘s high-level API following deprecation in Python 3.8.

  A coroutine that currently looks like this:

  ```
  async def foo(loop):
      await asyncio.sleep(1, loop=loop)
  ```

  Should be replaced with this:

  ```
  async def foo():
      await asyncio.sleep(1)
  ```

  If `foo()` was specifically designed *not* to run in the current thread’s running event loop (e.g. running in another thread’s event loop), consider using **`asyncio.run_coroutine_threadsafe()`** instead.

(Contributed by Yurii Karabas, Andrew Svetlov, Yury Selivanov and Kyle Stanley in bpo-42392 [https://bugs.python.org/issue?@action=redirect&bpo=42392].)

- The **`types.FunctionType`** constructor now inherits the current builtins if the *globals* dictionary has no `"__builtins__"` key, rather than using `{"None": None}` as builtins: same behavior as **`eval()`** and **`exec()`** functions. Defining a function with `def function(...): ...` in Python is not affected, globals cannot be overridden with this syntax: it also inherits the current builtins. (Contributed by Victor Stinner in bpo-42990 [https://bugs.python.org/issue?@action=redirect&bpo=42990].)

Changes in the C API

- The C API functions `PyParser_SimpleParseStringFlags`, `PyParser_SimpleParseStringFlagsFilename`, `PyParser_SimpleParseFileFlags`, `PyNode_Compile` and the type used by these functions, `struct _node`, were removed due to the switch to the new PEG parser.

  Source should be now be compiled directly to a code object using, for example, **`Py_CompileString()`**. The resulting code object can then be evaluated using, for example, **`PyEval_EvalCode()`**.

  Specifically:

  - A call to `PyParser_SimpleParseStringFlags` followed by `PyNode_Compile` can be replaced by calling **`Py_CompileString()`**.

  - There is no direct replacement for `PyParser_SimpleParseFileFlags`. To compile code from a `FILE *` argument, you will need to read the file in C and pass the resulting buffer to `Py_CompileString()`.

  - To compile a file given a `char *` filename, explicitly open the file, read it and compile the result. One way to do this is using the `io` module with `PyImport_ImportModule()`, `PyObject_CallMethod()`, `PyBytes_AsString()` and `Py_CompileString()`, as sketched below. (Declarations and error handling are omitted.)

    ```
    io_module = Import_ImportModule("io");
    fileobject = PyObject_CallMethod(io_module, "open",
    "ss", filename, "rb");
    source_bytes_object = PyObject_CallMethod(fileobject,
    "read", "");
    result = PyObject_CallMethod(fileobject, "close", "");
    source_buf = PyBytes_AsString(source_bytes_object);
    code = Py_CompileString(source_buf, filename,
    Py_file_input);
    ```

  - For `FrameObject` objects, the `f_lasti` member now represents a wordcode offset instead of a simple offset into the bytecode string. This means that this number needs to be multiplied by 2 to be used with APIs that expect a byte offset instead (like `PyCode_Addr2Line()` for example). Notice as well that the **`f_lasti`** member of `FrameObject` objects is not considered stable: please use `PyFrame_GetLineNumber()` instead.

CPython bytecode changes

- The `MAKE_FUNCTION` instruction now accepts either a dict or a tuple of strings as the function’s annotations.

(Contributed by Yurii Karabas and Inada Naoki in bpo-42202 [https://bugs.python.org/issue?@action=redirect&bpo=42202].)

Build Changes

- **PEP 644** [https://peps.python.org/pep-0644/]: Python now requires OpenSSL 1.1.1 or newer. OpenSSL 1.0.2 is no longer supported. (Contributed by Christian Heimes in bpo-43669 [https://bugs.python.org/issue?@action=redirect&bpo=43669].)
- The C99 functions **`snprintf()`** and **`vsnprintf()`** are now required to build Python. (Contributed by Victor Stinner in bpo-36020 [https://bugs.python.org/issue?@action=redirect&bpo=36020].)
- **`sqlite3`** requires SQLite 3.7.15 or higher. (Contributed by Sergey Fedoseev and Erlend E. Aasland in bpo-40744 [https://bugs.python.org/issue?@action=redirect&bpo=40744] and bpo-40810 [https://bugs.python.org/issue?@action=redirect&bpo=40810].)
- The **`atexit`** module must now always be built as a built-in module. (Contributed by Victor Stinner in bpo-42639 [https://bugs.python.org/issue?@action=redirect&bpo=42639].)
- Add **`--disable-test-modules`** option to the `configure` script: don’t build nor install test modules. (Contributed by Xavier de Gaye, Thomas Petazzoni and Peixing Xin in bpo-27640 [https://bugs.python.org/issue?@action=redirect&bpo=27640].)
- Add **`--with-wheel-pkg-dir=PATH option`** to the `./configure` script. If specified, the **`ensurepip`** module looks for `setuptools` and `pip` wheel packages in this directory: if both are present, these wheel packages are used instead of ensurepip bundled wheel packages.

Some Linux distribution packaging policies recommend against bundling dependencies. For example, Fedora installs wheel packages in the `/usr/share/python-wheels/` directory and don’t install the `ensurepip._bundled` package.

(Contributed by Victor Stinner in bpo-42856 [https://bugs.python.org/issue?@action=redirect&bpo=42856].)

- Add a new **`configure --without-static-libpython option`** to not build the `libpythonMAJOR.MINOR.a` static library and not install the `python.o` object file.

  (Contributed by Victor Stinner in bpo-43103 [https://bugs.python.org/issue?@action=redirect&bpo=43103].)

- The `configure` script now uses the `pkg-config` utility, if available, to detect the location of Tcl/Tk headers and libraries. As before, those locations can be explicitly specified with the `--with-tcltk-includes` and `--with-tcltk-libs` configuration options. (Contributed by Manolis Stamatogiannakis in bpo-42603 [https://bugs.python.org/issue?@action=redirect&bpo=42603].)

- Add **`--with-openssl-rpath`** option to `configure` script. The option simplifies building Python with a custom OpenSSL installation, e.g. `./configure --with-openssl=/path/to/openssl --with-openssl-rpath=auto`. (Contributed by Christian Heimes in bpo-43466 [https://bugs.python.org/issue?@action=redirect&bpo=43466].)

C API Changes

PEP 652: Maintaining the Stable ABI

The Stable ABI (Application Binary Interface) for extension modules or embedding Python is now explicitly defined. C API Stability describes C API and ABI stability guarantees along with best practices for using the Stable ABI.

(Contributed by Petr Viktorin in **PEP 652** [https://peps.python.org/pep-0652/] and bpo-43795 [https://bugs.python.org/issue?@action=redirect&bpo=43795].)

New Features

- The result of `PyNumber_Index()` now always has exact type `int`. Previously, the result could have been an instance of a subclass of `int`. (Contributed by Serhiy Storchaka in bpo-40792 [https://bugs.python.org/issue?@action=redirect&bpo=40792].)
- Add a new `orig_argv` member to the `PyConfig` structure: the list of the original command line arguments passed to the Python executable. (Contributed by Victor Stinner in bpo-23427 [https://bugs.python.org/issue?@action=redirect&bpo=23427].)
- The `PyDateTime_DATE_GET_TZINFO()` and `PyDateTime_TIME_GET_TZINFO()` macros have been added for accessing the `tzinfo` attributes of `datetime.datetime` and `datetime.time` objects. (Contributed by Zackery Spytz in bpo-30155 [https://bugs.python.org/issue?@action=redirect&bpo=30155].)
- Add a `PyCodec_Unregister()` function to unregister a codec search function. (Contributed by Hai Shi in bpo-41842 [https://bugs.python.org/issue?@action=redirect&bpo=41842].)
- The `PyIter_Send()` function was added to allow sending value into iterator without raising `StopIteration`

exception. (Contributed by Vladimir Matveev in bpo-41756 [https://bugs.python.org/issue?@action=redirect&bpo=41756].)

- Add **`PyUnicode_AsUTF8AndSize()`** to the limited C API. (Contributed by Alex Gaynor in bpo-41784 [https://bugs.python.org/issue?@action=redirect&bpo=41784].)
- Add **`PyModule_AddObjectRef()`** function: similar to **`PyModule_AddObject()`** but don't steal a reference to the value on success. (Contributed by Victor Stinner in bpo-1635741 [https://bugs.python.org/issue?@action=redirect&bpo=1635741].)
- Add **`Py_NewRef()`** and **`Py_XNewRef()`** functions to increment the reference count of an object and return the object. (Contributed by Victor Stinner in bpo-42262 [https://bugs.python.org/issue?@action=redirect&bpo=42262].)
- The **`PyType_FromSpecWithBases()`** and **`PyType_FromModuleAndSpec()`** functions now accept a single class as the *bases* argument. (Contributed by Serhiy Storchaka in bpo-42423 [https://bugs.python.org/issue?@action=redirect&bpo=42423].)
- The **`PyType_FromModuleAndSpec()`** function now accepts NULL `tp_doc` slot. (Contributed by Hai Shi in bpo-41832 [https://bugs.python.org/issue?@action=redirect&bpo=41832].)
- The **`PyType_GetSlot()`** function can accept static types. (Contributed by Hai Shi and Petr Viktorin in bpo-41073 [https://bugs.python.org/issue?@action=redirect&bpo=41073].)
- Add a new **`PySet_CheckExact()`** function to the C-API to check if an object is an instance of **`set`** but not an instance of a subtype. (Contributed by Pablo Galindo in bpo-43277 [https://bugs.python.org/issue?@action=redirect&bpo=43277].)

- Add **`PyErr_SetInterruptEx()`** which allows passing a signal number to simulate. (Contributed by Antoine Pitrou in bpo-43356 [https://bugs.python.org/issue?@action=redirect&bpo=43356].)
- The limited C API is now supported if Python is built in debug mode (if the `Py_DEBUG` macro is defined). In the limited C API, the **`Py_INCREF()`** and **`Py_DECREF()`** functions are now implemented as opaque function calls, rather than accessing directly the **`PyObject.ob_refcnt`** member, if Python is built in debug mode and the `Py_LIMITED_API` macro targets Python 3.10 or newer. It became possible to support the limited C API in debug mode because the **`PyObject`** structure is the same in release and debug mode since Python 3.8 (see bpo-36465 [https://bugs.python.org/issue?@action=redirect&bpo=36465]).

  The limited C API is still not supported in the **`--with-trace-refs`** special build (`Py_TRACE_REFS` macro). (Contributed by Victor Stinner in bpo-43688 [https://bugs.python.org/issue?@action=redirect&bpo=43688].)
- Add the **`Py_Is(x, y)`** function to test if the *x* object is the *y* object, the same as `x is y` in Python. Add also the **`Py_IsNone()`**, **`Py_IsTrue()`**, **`Py_IsFalse()`** functions to test if an object is, respectively, the `None` singleton, the `True` singleton or the `False` singleton. (Contributed by Victor Stinner in bpo-43753 [https://bugs.python.org/issue?@action=redirect&bpo=43753].)
- Add new functions to control the garbage collector from C code: **`PyGC_Enable()`**, **`PyGC_Disable()`**, **`PyGC_IsEnabled()`**. These functions allow to activate, deactivate and query the state of the garbage collector from C code without having to import the **`gc`** module.

- Add a new **`Py_TPFLAGS_DISALLOW_INSTANTIATION`** type flag to disallow creating type instances. (Contributed by Victor Stinner in bpo-43916 [https://bugs.python.org/issue?@action=redirect&bpo=43916].)
- Add a new **`Py_TPFLAGS_IMMUTABLETYPE`** type flag for creating immutable type objects: type attributes cannot be set nor deleted. (Contributed by Victor Stinner and Erlend E. Aasland in bpo-43908 [https://bugs.python.org/issue?@action=redirect&bpo=43908].)

Porting to Python 3.10

- The `PY_SSIZE_T_CLEAN` macro must now be defined to use **`PyArg_ParseTuple()`** and **`Py_BuildValue()`** formats which use `#`: `es#`, `et#`, `s#`, `u#`, `y#`, `z#`, `U#` and `Z#`. See Parsing arguments and building values and **PEP 353** [https://peps.python.org/pep-0353/]. (Contributed by Victor Stinner in bpo-40943 [https://bugs.python.org/issue?@action=redirect&bpo=40943].)
- Since **`Py_REFCNT()`** is changed to the inline static function, `Py_REFCNT(obj) = new_refcnt` must be replaced with `Py_SET_REFCNT(obj, new_refcnt)`: see **`Py_SET_REFCNT()`** (available since Python 3.9). For backward compatibility, this macro can be used:

  ```
  #if PY_VERSION_HEX < 0x030900A4
  #  define Py_SET_REFCNT(obj, refcnt) ((Py_REFCNT(obj) =
  (refcnt)), (void)0)
  #endif
  ```

  (Contributed by Victor Stinner in bpo-39573 [https://bugs.python.org/issue?@action=redirect&bpo=39573].)
- Calling **`PyDict_GetItem()`** without GIL held had been allowed for historical reason. It is no longer allowed.

(Contributed by Victor Stinner in bpo-40839 [https://bugs.python.org/issue?@action=redirect&bpo=40839].)

- `PyUnicode_FromUnicode(NULL, size)` and `PyUnicode_FromStringAndSize(NULL, size)` raise `DeprecationWarning` now. Use **`PyUnicode_New()`** to allocate Unicode object without initial data. (Contributed by Inada Naoki in bpo-36346 [https://bugs.python.org/issue?@action=redirect&bpo=36346].)
- The private `_PyUnicode_Name_CAPI` structure of the PyCapsule API `unicodedata.ucnhash_CAPI` has been moved to the internal C API. (Contributed by Victor Stinner in bpo-42157 [https://bugs.python.org/issue?@action=redirect&bpo=42157].)
- **`Py_GetPath()`**, **`Py_GetPrefix()`**, **`Py_GetExecPrefix()`**, **`Py_GetProgramFullPath()`**, **`Py_GetPythonHome()`** and **`Py_GetProgramName()`** functions now return `NULL` if called before **`Py_Initialize()`** (before Python is initialized). Use the new Python Initialization Configuration API to get the Python Path Configuration. (Contributed by Victor Stinner in bpo-42260 [https://bugs.python.org/issue?@action=redirect&bpo=42260].)
- **`PyList_SET_ITEM()`**, **`PyTuple_SET_ITEM()`** and **`PyCell_SET()`** macros can no longer be used as l-value or r-value. For example, `x = PyList_SET_ITEM(a, b, c)` and `PyList_SET_ITEM(a, b, c) = x` now fail with a compiler error. It prevents bugs like `if (PyList_SET_ITEM (a, b, c) < 0) ...` test. (Contributed by Zackery Spytz and Victor Stinner in bpo-30459 [https://bugs.python.org/issue?@action=redirect&bpo=30459].)
- The non-limited API files `odictobject.h`, `parser_interface.h`, `picklebufobject.h`, `pyarena.h`, `pyctype.h`, `pydebug.h`, `pyfpe.h`,

and `pytime.h` have been moved to the `Include/cpython` directory. These files must not be included directly, as they are already included in `Python.h`; see Include Files. If they have been included directly, consider including `Python.h` instead. (Contributed by Nicholas Sim in bpo-35134 [https://bugs.python.org/issue?@action=redirect&bpo=35134].)

- Use the **`Py_TPFLAGS_IMMUTABLETYPE`** type flag to create immutable type objects. Do not rely on **`Py_TPFLAGS_HEAPTYPE`** to decide if a type object is mutable or not; check if **`Py_TPFLAGS_IMMUTABLETYPE`** is set instead. (Contributed by Victor Stinner and Erlend E. Aasland in bpo-43908 [https://bugs.python.org/issue?@action=redirect&bpo=43908].)
- The undocumented function `Py_FrozenMain` has been removed from the limited API. The function is mainly useful for custom builds of Python. (Contributed by Petr Viktorin in bpo-26241 [https://bugs.python.org/issue?@action=redirect&bpo=26241].)

Deprecated

- The `PyUnicode_InternImmortal()` function is now deprecated and will be removed in Python 3.12: use **`PyUnicode_InternInPlace()`** instead. (Contributed by Victor Stinner in bpo-41692 [https://bugs.python.org/issue?@action=redirect&bpo=41692].)

Removed

- Removed `Py_UNICODE_str*` functions manipulating `Py_UNICODE*` strings. (Contributed by Inada Naoki in bpo-41123 [https://bugs.python.org/issue?@action=redirect&bpo=41123].)

  - `Py_UNICODE_strlen`: use **`PyUnicode_GetLength()`** or **`PyUnicode_GET_LENGTH`**
  - `Py_UNICODE_strcat`: use **`PyUnicode_CopyCharacters()`** or **`PyUnicode_FromFormat()`**
  - `Py_UNICODE_strcpy`, `Py_UNICODE_strncpy`: use **`PyUnicode_CopyCharacters()`** or **`PyUnicode_Substring()`**
  - `Py_UNICODE_strcmp`: use **`PyUnicode_Compare()`**
  - `Py_UNICODE_strncmp`: use **`PyUnicode_Tailmatch()`**
  - `Py_UNICODE_strchr`, `Py_UNICODE_strrchr`: use **`PyUnicode_FindChar()`**

- Removed `PyUnicode_GetMax()`. Please migrate to new (**PEP 393** [https://peps.python.org/pep-0393/]) APIs. (Contributed by Inada Naoki in bpo-41103 [https://bugs.python.org/issue?@action=redirect&bpo=41103].)

- Removed `PyLong_FromUnicode()`. Please migrate to **`PyLong_FromUnicodeObject()`**. (Contributed by Inada Naoki in bpo-41103 [https://bugs.python.org/issue?@action=redirect&bpo=41103].)

- Removed `PyUnicode_AsUnicodeCopy()`. Please use **`PyUnicode_AsUCS4Copy()`** or **`PyUnicode_AsWideCharString()`** (Contributed by Inada Naoki in bpo-41103 [https://bugs.python.org/issue?@action=redirect&bpo=41103].)

- Removed `_Py_CheckRecursionLimit` variable: it has been replaced by `ceval.recursion_limit` of the **`PyInterpreterState`** structure. (Contributed by Victor Stinner in bpo-41834 [https://bugs.python.org/issue?@action=redirect&bpo=41834].)

- Removed undocumented macros `Py_ALLOW_RECURSION` and `Py_END_ALLOW_RECURSION` and the `recursion_critical` field of the **`PyInterpreterState`** structure. (Contributed by Serhiy Storchaka in bpo-41936 [https://bugs.python.org/issue?@action=redirect&bpo=41936].)

- Removed the undocumented `PyOS_InitInterrupts()` function. Initializing Python already implicitly installs signal handlers: see **`PyConfig.install_signal_handlers`**. (Contributed by Victor Stinner in bpo-41713 [https://bugs.python.org/issue?@action=redirect&bpo=41713].)

- Remove the `PyAST_Validate()` function. It is no longer possible to build a AST object (`mod_ty` type) with the public C API. The function was already excluded from the limited C API (**PEP 384** [https://peps.python.org/pep-0384/]). (Contributed by Victor Stinner in bpo-43244 [https://bugs.python.org/issue?@action=redirect&bpo=43244].)

- Remove the `symtable.h` header file and the undocumented functions:

  - `PyST_GetScope()`
  - `PySymtable_Build()`
  - `PySymtable_BuildObject()`
  - `PySymtable_Free()`
  - `Py_SymtableString()`
  - `Py_SymtableStringObject()`

  The `Py_SymtableString()` function was part the stable ABI by mistake but it could not be used, because the `symtable.h` header file was excluded from the limited C API.

  Use Python **`symtable`** module instead. (Contributed by Victor Stinner in bpo-43244 [https://bugs.python.org/issue?@action=redirect&bpo=43244].)

- Remove **`PyOS_ReadlineFunctionPointer()`** from the limited C API headers and from `python3.dll`, the library that provides the stable ABI on Windows. Since the function takes a `FILE*` argument, its ABI stability cannot be guaranteed. (Contributed by Petr Viktorin in bpo-43868 [https://bugs.python.org/issue?@action=redirect&bpo=43868].)

- Remove `ast.h`, `asdl.h`, and `Python-ast.h` header files. These functions were undocumented and excluded from the limited C API. Most names defined by these header files were not prefixed by `Py` and so could create names conflicts. For example, `Python-ast.h` defined a `Yield` macro which was conflict with the `Yield` name used by the Windows `<winbase.h>` header. Use the Python **`ast`** module instead. (Contributed by Victor Stinner in bpo-43244 [https://bugs.python.org/issue?@action=redirect&bpo=43244].)

- Remove the compiler and parser functions using `struct _mod` type, because the public AST C API was removed:

    - `PyAST_Compile()`
    - `PyAST_CompileEx()`
    - `PyAST_CompileObject()`
    - `PyFuture_FromAST()`
    - `PyFuture_FromASTObject()`
    - `PyParser_ASTFromFile()`
    - `PyParser_ASTFromFileObject()`
    - `PyParser_ASTFromFilename()`
    - `PyParser_ASTFromString()`
    - `PyParser_ASTFromStringObject()`

  These functions were undocumented and excluded from the limited C API. (Contributed by Victor Stinner in bpo-43244 [https://bugs.python.org/issue?@action=redirect&bpo=43244].)

- Remove the `pyarena.h` header file with functions:

    - `PyArena_New()`
    - `PyArena_Free()`
    - `PyArena_Malloc()`
    - `PyArena_AddPyObject()`

These functions were undocumented, excluded from the limited C API, and were only used internally by the compiler. (Contributed by Victor Stinner in bpo-43244 [https://bugs.python.org/issue?@action=redirect&bpo=43244].)

- The `PyThreadState.use_tracing` member has been removed to optimize Python. (Contributed by Mark Shannon in bpo-43760 [https://bugs.python.org/issue?@action=redirect&bpo=43760].)

Notable security feature in 3.10.7

Converting between **`int`** and **`str`** in bases other than 2 (binary), 4, 8 (octal), 16 (hexadecimal), or 32 such as base 10 (decimal) now raises a **`ValueError`** if the number of digits in string form is above a limit to avoid potential denial of service attacks due to the algorithmic complexity. This is a mitigation for **CVE 2020-10735** [https://www.cve.org/CVERecord?id=CVE-2020-10735]. This limit can be configured or disabled by environment variable, command line flag, or **`sys`** APIs. See the integer string conversion length limitation documentation. The default limit is 4300 digits in string form.

Notable security feature in 3.10.8

The deprecated **`mailcap`** module now refuses to inject unsafe text (filenames, MIME types, parameters) into shell commands. Instead of using such text, it will warn and act as if a match was not found (or for test commands, as if the test failed). (Contributed by Petr Viktorin in gh-98966 [https://github.com/python/cpython/issues/98966].)

Notable changes in 3.10.12

tarfile

- The extraction methods in [`tarfile`](), and [`shutil.unpack_archive()`](), have a new a *filter* argument that allows limiting tar features than may be surprising or dangerous, such as creating files outside the destination directory. See Extraction filters for details. In Python 3.12, use without the *filter* argument will show a `DeprecationWarning`. In Python 3.14, the default will switch to `'data'`. (Contributed by Petr Viktorin in **PEP 706** [https://peps.python.org/pep-0706/].)

What’s New In Python 3.9

Editor:: Łukasz Langa

This article explains the new features in Python 3.9, compared to 3.8. Python 3.9 was released on October 5, 2020. For full details, see the changelog.

See also

PEP 596 [https://peps.python.org/pep-0596/] - Python 3.9 Release Schedule

Summary – Release highlights

New syntax features:

- **PEP 584** [https://peps.python.org/pep-0584/], union operators added to `dict`;
- **PEP 585** [https://peps.python.org/pep-0585/], type hinting generics in standard collections;
- **PEP 614** [https://peps.python.org/pep-0614/], relaxed grammar restrictions on decorators.

New built-in features:

- **PEP 616** [https://peps.python.org/pep-0616/], string methods to remove prefixes and suffixes.

New features in the standard library:

- **PEP 593** [https://peps.python.org/pep-0593/], flexible function and variable annotations;

- `os.pidfd_open()` added that allows process management without races and signals.

Interpreter improvements:

- **PEP 573** [https://peps.python.org/pep-0573/], fast access to module state from methods of C extension types;
- **PEP 617** [https://peps.python.org/pep-0617/], CPython now uses a new parser based on PEG;
- a number of Python builtins (range, tuple, set, frozenset, list, dict) are now sped up using **PEP 590** [https://peps.python.org/pep-0590/] vectorcall;
- garbage collection does not block on resurrected objects;
- a number of Python modules (`_abc`, `audioop`, `_bz2`, `_codecs`, `_contextvars`, `_crypt`, `_functools`, `_json`, `_locale`, `math`, `operator`, `resource`, `time`, `_weakref`) now use multiphase initialization as defined by PEP 489;
- a number of standard library modules (`audioop`, `ast`, `grp`, `_hashlib`, `pwd`, `_posixsubprocess`, `random`, `select`, `struct`, `termios`, `zlib`) are now using the stable ABI defined by PEP 384.

New library modules:

- **PEP 615** [https://peps.python.org/pep-0615/], the IANA Time Zone Database is now present in the standard library in the `zoneinfo` module;
- an implementation of a topological sort of a graph is now provided in the new `graphlib` module.

Release process changes:

- **PEP 602** [https://peps.python.org/pep-0602/], CPython adopts an annual release cycle.

You should check for DeprecationWarning in your code

When Python 2.7 was still supported, a lot of functionality in Python 3 was kept for backward compatibility with Python 2.7. With the end of Python 2 support, these backward compatibility layers have been removed, or will be removed soon. Most of them emitted a **`DeprecationWarning`** warning for several years. For example, using `collections.Mapping` instead of `collections.abc.Mapping` emits a **`DeprecationWarning`** since Python 3.3, released in 2012.

Test your application with the **`-W`** `default` command-line option to see **`DeprecationWarning`** and **`PendingDeprecationWarning`**, or even with **`-W`** `error` to treat them as errors. Warnings Filter can be used to ignore warnings from third-party code.

Python 3.9 is the last version providing those Python 2 backward compatibility layers, to give more time to Python projects maintainers to organize the removal of the Python 2 support and add support for Python 3.9.

Aliases to Abstract Base Classes in the **`collections`** module, like `collections.Mapping` alias to **`collections.abc.Mapping`**, are kept for one last release for backward compatibility. They will be removed from Python 3.10.

More generally, try to run your tests in the Python Development Mode which helps to prepare your code to make it compatible with the next Python version.

Note: a number of pre-existing deprecations were removed in this version of Python as well. Consult the Removed section.

New Features

Dictionary Merge & Update Operators

Merge (`|`) and update (`|=`) operators have been added to the built-in **`dict`** class. Those complement the existing `dict.update` and `{**d1, **d2}` methods of merging dictionaries.

Example:

```
>>> x = {"key1": "value1 from x", "key2": "value2 from x"}
>>> y = {"key2": "value2 from y", "key3": "value3 from y"}
>>> x | y
{'key1': 'value1 from x', 'key2': 'value2 from y', 'key3':
'value3 from y'}
>>> y | x
{'key2': 'value2 from x', 'key3': 'value3 from y', 'key1':
'value1 from x'}
```

See **PEP 584** [https://peps.python.org/pep-0584/] for a full description. (Contributed by Brandt Bucher in bpo-36144 [https://bugs.python.org/issue?@action=redirect&bpo=36144].)

New String Methods to Remove Prefixes and Suffixes

`str.removeprefix(prefix)` and **`str.removesuffix(suffix)`** have been added to easily remove an unneeded prefix or a suffix from a string. Corresponding `bytes`, `bytearray`, and `collections.UserString` methods have also been added. See **PEP 616** [https://peps.python.org/pep-0616/] for a full description. (Contributed by Dennis Sweeney in bpo-39939 [https://bugs.python.org/issue?@action=redirect&bpo=39939].)

Type Hinting Generics in Standard Collections

In type annotations you can now use built-in collection types such as `list` and `dict` as generic types instead of importing the corresponding capitalized types (e.g. `List` or `Dict`) from `typing`. Some other types in the standard library are also now generic, for example `queue.Queue`.

Example:

```
def greet_all(names: list[str]) -> None:
    for name in names:
        print("Hello", name)
```

See **PEP 585** [https://peps.python.org/pep-0585/] for more details. (Contributed by Guido van Rossum, Ethan Smith, and Batuhan Taşkaya in bpo-39481 [https://bugs.python.org/issue?@action=redirect&bpo=39481].)

New Parser

Python 3.9 uses a new parser, based on PEG [https://en.wikipedia.org/wiki/Parsing_expression_grammar] instead of LL(1) [https://en.wikipedia.org/wiki/LL_parser]. The new parser's performance is roughly comparable to that of the old parser, but the PEG formalism is more flexible than LL(1) when it comes to designing new language features. We'll start using this flexibility in Python 3.10 and later.

The **`ast`** module uses the new parser and produces the same AST as the old parser.

In Python 3.10, the old parser will be deleted and so will all functionality that depends on it (primarily the **`parser`** module, which has long been deprecated). In Python 3.9 *only*, you can switch back to the LL(1) parser using a command line switch (`-X oldparser`) or an environment variable (`PYTHONOLDPARSER=1`).

See **PEP 617** [https://peps.python.org/pep-0617/] for more details. (Contributed by Guido van Rossum, Pablo Galindo and Lysandros Nikolaou in bpo-40334 [https://bugs.python.org/issue?@action=redirect&bpo=40334].)

Other Language Changes

- **`__import__()`** now raises **`ImportError`** instead of **`ValueError`**, which used to occur when a relative import went past its top-level package. (Contributed by Ngalim Siregar in bpo-37444 [https://bugs.python.org/issue?@action=redirect&bpo=37444].)

- Python now gets the absolute path of the script filename specified on the command line (ex: `python3 script.py`): the `__file__` attribute of the **`__main__`** module became an absolute path, rather than a relative path. These paths now remain valid after the current directory is changed by **`os.chdir()`**. As a side effect, the traceback also displays the absolute path for **`__main__`** module frames in this case. (Contributed by Victor Stinner in bpo-20443 [https://bugs.python.org/issue?@action=redirect&bpo=20443].)

- In the Python Development Mode and in debug build, the *encoding* and *errors* arguments are now checked for string encoding and decoding operations. Examples: **`open()`**, **`str.encode()`** and **`bytes.decode()`**.

  By default, for best performance, the *errors* argument is only checked at the first encoding/decoding error and the *encoding* argument is sometimes ignored for empty strings. (Contributed by Victor Stinner in bpo-37388 [https://bugs.python.org/issue?@action=redirect&bpo=37388].)

- `"".replace("", s, n)` now returns `s` instead of an empty string for all non-zero `n`. It is now consistent with

`"".replace("", s)`. There are similar changes for **`bytes`** and **`bytearray`** objects. (Contributed by Serhiy Storchaka in bpo-28029 [https://bugs.python.org/issue?@action=redirect&bpo=28029].)

- Any valid expression can now be used as a decorator. Previously, the grammar was much more restrictive. See **PEP 614** [https://peps.python.org/pep-0614/] for details. (Contributed by Brandt Bucher in bpo-39702 [https://bugs.python.org/issue?@action=redirect&bpo=39702].)
- Improved help for the **`typing`** module. Docstrings are now shown for all special forms and special generic aliases (like `Union` and `List`). Using **`help()`** with generic alias like `List[int]` will show the help for the correspondent concrete type (`list` in this case). (Contributed by Serhiy Storchaka in bpo-40257 [https://bugs.python.org/issue?@action=redirect&bpo=40257].)
- Parallel running of **`aclose()`** / **`asend()`** / **`athrow()`** is now prohibited, and `ag_running` now reflects the actual running status of the async generator. (Contributed by Yury Selivanov in bpo-30773 [https://bugs.python.org/issue?@action=redirect&bpo=30773].)
- Unexpected errors in calling the `__iter__` method are no longer masked by `TypeError` in the **`in`** operator and functions **`contains()`**, **`indexOf()`** and **`countOf()`** of the **`operator`** module. (Contributed by Serhiy Storchaka in bpo-40824 [https://bugs.python.org/issue?@action=redirect&bpo=40824].)
- Unparenthesized lambda expressions can no longer be the expression part in an `if` clause in comprehensions and generator expressions. See bpo-41848 [https://bugs.python.org/issue?@action=redirect&bpo=41848] and bpo-

43755 [https://bugs.python.org/issue?@action=redirect&bpo=43755] for details.

New Modules

zoneinfo

The **`zoneinfo`** module brings support for the IANA time zone database to the standard library. It adds **`zoneinfo.ZoneInfo`**, a concrete **`datetime.tzinfo`** implementation backed by the system's time zone data.

Example:

```
>>> from zoneinfo import ZoneInfo
>>> from datetime import datetime, timedelta

>>> # Daylight saving time
>>> dt = datetime(2020, 10, 31, 12,
tzinfo=ZoneInfo("America/Los_Angeles"))
>>> print(dt)
2020-10-31 12:00:00-07:00
>>> dt.tzname()
'PDT'

>>> # Standard time
>>> dt += timedelta(days=7)
>>> print(dt)
2020-11-07 12:00:00-08:00
>>> print(dt.tzname())
PST
```

As a fall-back source of data for platforms that don't ship the IANA database, the tzdata [https://pypi.org/project/tzdata/] module was released as a first-party package – distributed via PyPI and maintained by the CPython core team.

See also

PEP 615 [https://peps.python.org/pep-0615/] – Support for the IANA Time Zone Database in the Standard Library

PEP written and implemented by Paul Ganssle

graphlib

A new module, `graphlib`, was added that contains the `graphlib.TopologicalSorter` class to offer functionality to perform topological sorting of graphs. (Contributed by Pablo Galindo, Tim Peters and Larry Hastings in bpo-17005 [https://bugs.python.org/issue?@action=redirect&bpo=17005].)

Improved Modules

ast

Added the *indent* option to `dump()` which allows it to produce a multiline indented output. (Contributed by Serhiy Storchaka in bpo-37995 [https://bugs.python.org/issue?@action=redirect&bpo=37995].)

Added `ast.unparse()` as a function in the `ast` module that can be used to unparse an `ast.AST` object and produce a string with code that would produce an equivalent `ast.AST` object when parsed. (Contributed by Pablo Galindo and Batuhan Taskaya in bpo-38870 [https://bugs.python.org/issue?@action=redirect&bpo=38870].)

Added docstrings to AST nodes that contains the ASDL signature used to construct that node. (Contributed by Batuhan Taskaya in bpo-39638 [https://bugs.python.org/issue?@action=redirect&bpo=39638].)

asyncio

Due to significant security concerns, the *reuse_address* parameter of **`asyncio.loop.create_datagram_endpoint()`** is no longer supported. This is because of the behavior of the socket option `SO_REUSEADDR` in UDP. For more details, see the documentation for `loop.create_datagram_endpoint()`. (Contributed by Kyle Stanley, Antoine Pitrou, and Yury Selivanov in bpo-37228 [https://bugs.python.org/issue?@action=redirect&bpo=37228].)

Added a new coroutine **`shutdown_default_executor()`** that schedules a shutdown for the default executor that waits on the **`ThreadPoolExecutor`** to finish closing. Also, **`asyncio.run()`** has been updated to use the new coroutine. (Contributed by Kyle Stanley in bpo-34037 [https://bugs.python.org/issue?@action=redirect&bpo=34037].)

Added **`asyncio.PidfdChildWatcher`**, a Linux-specific child watcher implementation that polls process file descriptors. (bpo-38692 [https://bugs.python.org/issue?@action=redirect&bpo=38692])

Added a new coroutine **`asyncio.to_thread()`**. It is mainly used for running IO-bound functions in a separate thread to avoid blocking the event loop, and essentially works as a high-level version of **`run_in_executor()`** that can directly take keyword arguments. (Contributed by Kyle Stanley and Yury Selivanov in bpo-32309 [https://bugs.python.org/issue?@action=redirect&bpo=32309].)

When cancelling the task due to a timeout, **`asyncio.wait_for()`** will now wait until the cancellation is complete also in the case when *timeout* is <= 0, like it does with positive timeouts. (Contributed by Elvis Pranskevichus in bpo-32751 [https://bugs.python.org/issue?@action=redirect&bpo=32751].)

`asyncio` now raises `TypeError` when calling incompatible methods with an `ssl.SSLSocket` socket. (Contributed by Ido Michael in bpo-37404 [https://bugs.python.org/issue?@action=redirect&bpo=37404].)

compileall

Added new possibility to use hardlinks for duplicated `.pyc` files: *hardlink_dupes* parameter and –hardlink-dupes command line option. (Contributed by Lumír ‘Frenzy’ Balhar in bpo-40495 [https://bugs.python.org/issue?@action=redirect&bpo=40495].)

Added new options for path manipulation in resulting `.pyc` files: *stripdir*, *prependdir*, *limit_sl_dest* parameters and -s, -p, -e command line options. Added the possibility to specify the option for an optimization level multiple times. (Contributed by Lumír ‘Frenzy’ Balhar in bpo-38112 [https://bugs.python.org/issue?@action=redirect&bpo=38112].)

concurrent.futures

Added a new *cancel_futures* parameter to `concurrent.futures.Executor.shutdown()` that cancels all pending futures which have not started running, instead of waiting for them to complete before shutting down the executor. (Contributed by Kyle Stanley in bpo-39349 [https://bugs.python.org/issue?@action=redirect&bpo=39349].)

Removed daemon threads from `ThreadPoolExecutor` and `ProcessPoolExecutor`. This improves compatibility with subinterpreters and predictability in their shutdown processes. (Contributed by Kyle Stanley in bpo-39812 [https://bugs.python.org/issue?@action=redirect&bpo=39812].)

Workers in **`ProcessPoolExecutor`** are now spawned on demand, only when there are no available idle workers to reuse. This optimizes startup overhead and reduces the amount of lost CPU time to idle workers. (Contributed by Kyle Stanley in bpo-39207 [https://bugs.python.org/issue?@action=redirect&bpo=39207].)

curses

Added **`curses.get_escdelay()`**, **`curses.set_escdelay()`**, **`curses.get_tabsize()`**, and **`curses.set_tabsize()`** functions. (Contributed by Anthony Sottile in bpo-38312 [https://bugs.python.org/issue?@action=redirect&bpo=38312].)

datetime

The **`isocalendar()`** of **`datetime.date`** and **`isocalendar()`** of **`datetime.datetime`** methods now returns a **`namedtuple()`** instead of a **`tuple`**. (Contributed by Donghee Na in bpo-24416 [https://bugs.python.org/issue?@action=redirect&bpo=24416].)

distutils

The **upload** command now creates SHA2-256 and Blake2b-256 hash digests. It skips MD5 on platforms that block MD5 digest. (Contributed by Christian Heimes in bpo-40698 [https://bugs.python.org/issue?@action=redirect&bpo=40698].)

fcntl

Added constants **`fcntl.F_OFD_GETLK`**, **`fcntl.F_OFD_SETLK`** and **`fcntl.F_OFD_SETLKW`**. (Contributed by Donghee Na in bpo-38602 [https://bugs.python.org/issue?@action=redirect&bpo=38602].)

ftplib

`FTP` and `FTP_TLS` now raise a `ValueError` if the given timeout for their constructor is zero to prevent the creation of a non-blocking socket. (Contributed by Donghee Na in bpo-39259 [https://bugs.python.org/issue?@action=redirect&bpo=39259].)

gc

When the garbage collector makes a collection in which some objects resurrect (they are reachable from outside the isolated cycles after the finalizers have been executed), do not block the collection of all objects that are still unreachable. (Contributed by Pablo Galindo and Tim Peters in bpo-38379 [https://bugs.python.org/issue?@action=redirect&bpo=38379].)

Added a new function `gc.is_finalized()` to check if an object has been finalized by the garbage collector. (Contributed by Pablo Galindo in bpo-39322 [https://bugs.python.org/issue?@action=redirect&bpo=39322].)

hashlib

The `hashlib` module can now use SHA3 hashes and SHAKE XOF from OpenSSL when available. (Contributed by Christian Heimes in bpo-37630 [https://bugs.python.org/issue?@action=redirect&bpo=37630].)

Builtin hash modules can now be disabled with `./configure --without-builtin-hashlib-hashes` or selectively enabled with e.g. `./configure --with-builtin-hashlib-hashes=sha3,blake2` to force use of OpenSSL based implementation. (Contributed by Christian Heimes in bpo-40479 [https://bugs.python.org/issue?@action=redirect&bpo=40479])

http

HTTP status codes `103 EARLY_HINTS`, `418 IM_A_TEAPOT` and `425 TOO_EARLY` are added to **`http.HTTPStatus`**. (Contributed by Donghee Na in bpo-39509 [https://bugs.python.org/issue?@action=redirect&bpo=39509] and Ross Rhodes in bpo-39507 [https://bugs.python.org/issue?@action=redirect&bpo=39507].)

IDLE and idlelib

Added option to toggle cursor blink off. (Contributed by Zackery Spytz in bpo-4603 [https://bugs.python.org/issue?@action=redirect&bpo=4603].)

Escape key now closes IDLE completion windows. (Contributed by Johnny Najera in bpo-38944 [https://bugs.python.org/issue?@action=redirect&bpo=38944].)

Added keywords to module name completion list. (Contributed by Terry J. Reedy in bpo-37765 [https://bugs.python.org/issue?@action=redirect&bpo=37765].)

New in 3.9 maintenance releases

Make IDLE invoke **`sys.excepthook()`** (when started without '-n'). User hooks were previously ignored. (Contributed by Ken Hilton in bpo-43008 [https://bugs.python.org/issue?@action=redirect&bpo=43008].)

The changes above have been backported to 3.8 maintenance releases.

Rearrange the settings dialog. Split the General tab into Windows and Shell/Ed tabs. Move help sources, which extend the Help menu, to the Extensions tab. Make space for new options and shorten the dialog. The latter makes the dialog better fit small screens. (Contributed by Terry Jan Reedy in bpo-40468 [https://bugs.python.org/issue?@action=redirect&bpo=40468].) Move the indent space setting from the Font tab to the new Windows tab. (Contributed by Mark Roseman and Terry Jan Reedy in bpo-33962 [https://bugs.python.org/issue?@action=redirect&bpo=33962].)

Apply syntax highlighting to `.pyi` files. (Contributed by Alex Waygood and Terry Jan Reedy in bpo-45447 [https://bugs.python.org/issue?@action=redirect&bpo=45447].)

imaplib

`IMAP4` and **`IMAP4_SSL`** now have an optional *timeout* parameter for their constructors. Also, the **`open()`** method now has an optional *timeout* parameter with this change. The overridden methods of **`IMAP4_SSL`** and **`IMAP4_stream`** were applied to this change. (Contributed by Donghee Na in bpo-38615 [https://bugs.python.org/issue?@action=redirect&bpo=38615].)

`imaplib.IMAP4.unselect()` is added. **`imaplib.IMAP4.unselect()`** frees server's resources associated with the selected mailbox and returns the server to the authenticated state. This command performs the same actions as **`imaplib.IMAP4.close()`**, except that no messages are permanently removed from the currently selected mailbox. (Contributed by Donghee Na in bpo-40375 [https://bugs.python.org/issue?@action=redirect&bpo=40375].)

importlib

To improve consistency with import statements, **`importlib.util.resolve_name()`** now raises **`ImportError`** instead of **`ValueError`** for invalid relative import attempts. (Contributed by Ngalim Siregar in bpo-37444 [https://bugs.python.org/issue?@action=redirect&bpo=37444].)

Import loaders which publish immutable module objects can now publish immutable packages in addition to individual modules. (Contributed by Dino Viehland in bpo-39336 [https://bugs.python.org/issue?@action=redirect&bpo=39336].)

Added **`importlib.resources.files()`** function with support for subdirectories in package data, matching backport in `importlib_resources` version 1.5. (Contributed by Jason R. Coombs in bpo-39791 [https://bugs.python.org/issue?@action=redirect&bpo=39791].)

Refreshed `importlib.metadata` from `importlib_metadata` version 1.6.1.

inspect

`inspect.BoundArguments.arguments` is changed from `OrderedDict` to regular dict. (Contributed by Inada Naoki in bpo-36350 [https://bugs.python.org/issue?@action=redirect&bpo=36350] and bpo-39775 [https://bugs.python.org/issue?@action=redirect&bpo=39775].)

ipaddress

`ipaddress` now supports IPv6 Scoped Addresses (IPv6 address with suffix `%<scope_id>`).

Scoped IPv6 addresses can be parsed using **`ipaddress.IPv6Address`**. If present, scope zone ID is available through the **`scope_id`** attribute. (Contributed by Oleksandr Pavliuk in bpo-34788 [https://bugs.python.org/issue?@action=redirect&bpo=34788].)

Starting with Python 3.9.5 the **`ipaddress`** module no longer accepts any leading zeros in IPv4 address strings. (Contributed by Christian Heimes in bpo-36384 [https://bugs.python.org/issue?@action=redirect&bpo=36384]).

math

Expanded the **`math.gcd()`** function to handle multiple arguments. Formerly, it only supported two arguments. (Contributed by Serhiy Storchaka in bpo-39648 [https://bugs.python.org/issue?@action=redirect&bpo=39648].)

Added **`math.lcm()`**: return the least common multiple of specified arguments. (Contributed by Mark Dickinson, Ananthakrishnan and Serhiy Storchaka in bpo-39479 [https://bugs.python.org/issue?@action=redirect&bpo=39479] and bpo-39648 [https://bugs.python.org/issue?@action=redirect&bpo=39648].)

Added **`math.nextafter()`**: return the next floating-point value after *x* towards *y*. (Contributed by Victor Stinner in bpo-39288 [https://bugs.python.org/issue?@action=redirect&bpo=39288].)

Added **`math.ulp()`**: return the value of the least significant bit of a float. (Contributed by Victor Stinner in bpo-39310 [https://bugs.python.org/issue?@action=redirect&bpo=39310].)

multiprocessing

The **`multiprocessing.SimpleQueue`** class has a new **`close()`** method to explicitly close the queue. (Contributed by Victor Stinner in bpo-30966 [https://bugs.python.org/issue?@action=redirect&bpo=30966].)

nntplib

`NNTP` and **`NNTP_SSL`** now raise a **`ValueError`** if the given timeout for their constructor is zero to prevent the creation of a non-blocking socket. (Contributed by Donghee Na in bpo-39259 [https://bugs.python.org/issue?@action=redirect&bpo=39259].)

os

Added **`CLD_KILLED`** and **`CLD_STOPPED`** for **`si_code`**. (Contributed by Donghee Na in bpo-38493 [https://bugs.python.org/issue?@action=redirect&bpo=38493].)

Exposed the Linux-specific **`os.pidfd_open()`** (bpo-38692 [https://bugs.python.org/issue?@action=redirect&bpo=38692]) and **`os.P_PIDFD`** (bpo-38713 [https://bugs.python.org/issue?@action=redirect&bpo=38713]) for process management with file descriptors.

The **`os.unsetenv()`** function is now also available on Windows. (Contributed by Victor Stinner in bpo-39413 [https://bugs.python.org/issue?@action=redirect&bpo=39413].)

The **`os.putenv()`** and **`os.unsetenv()`** functions are now always available. (Contributed by Victor Stinner in bpo-39395 [https://bugs.python.org/issue?@action=redirect&bpo=39395].)

Added **`os.waitstatus_to_exitcode()`** function: convert a wait status to an exit code. (Contributed by Victor Stinner in bpo-40094 [https://bugs.python.org/issue?@action=redirect&bpo=40094].)

pathlib

Added **`pathlib.Path.readlink()`** which acts similarly to **`os.readlink()`**. (Contributed by Girts Folkmanis in bpo-30618 [https://bugs.python.org/issue?@action=redirect&bpo=30618])

pdb

On Windows now **`Pdb`** supports `~/.pdbrc`. (Contributed by Tim Hopper and Dan Lidral-Porter in bpo-20523 [https://bugs.python.org/issue?@action=redirect&bpo=20523].)

poplib

`POP3` and `POP3_SSL` now raise a `ValueError` if the given timeout for their constructor is zero to prevent the creation of a non-blocking socket. (Contributed by Donghee Na in bpo-39259 [https://bugs.python.org/issue?@action=redirect&bpo=39259].)

pprint

`pprint` can now pretty-print `types.SimpleNamespace`. (Contributed by Carl Bordum Hansen in bpo-37376 [https://bugs.python.org/issue?@action=redirect&bpo=37376].)

pydoc

The documentation string is now shown not only for class, function, method etc, but for any object that has its own `__doc__` attribute. (Contributed by Serhiy Storchaka in bpo-40257 [https://bugs.python.org/issue?@action=redirect&bpo=40257].)

random

Added a new `random.Random.randbytes()` method: generate random bytes. (Contributed by Victor Stinner in bpo-40286 [https://bugs.python.org/issue?@action=redirect&bpo=40286].)

signal

Exposed the Linux-specific `signal.pidfd_send_signal()` for sending to signals to a process using a file descriptor instead of a pid. (bpo-38712 [https://bugs.python.org/issue?@action=redirect&bpo=38712])

smtplib

`SMTP` and `SMTP_SSL` now raise a `ValueError` if the given timeout for their constructor is zero to prevent the creation of a non-blocking socket. (Contributed by Donghee Na in bpo-39259 [https://bugs.python.org/issue?@action=redirect&bpo=39259].)

`LMTP` constructor now has an optional *timeout* parameter. (Contributed by Donghee Na in bpo-39329 [https://bugs.python.org/issue?@action=redirect&bpo=39329].)

socket

The **`socket`** module now exports the **`CAN_RAW_JOIN_FILTERS`** constant on Linux 4.1 and greater. (Contributed by Stefan Tatschner and Zackery Spytz in bpo-25780 [https://bugs.python.org/issue?@action=redirect&bpo=25780].)

The socket module now supports the **`CAN_J1939`** protocol on platforms that support it. (Contributed by Karl Ding in bpo-40291 [https://bugs.python.org/issue?@action=redirect&bpo=40291].)

The socket module now has the **`socket.send_fds()`** and **`socket.recv_fds()`** functions. (Contributed by Joannah Nanjekye, Shinya Okano and Victor Stinner in bpo-28724 [https://bugs.python.org/issue?@action=redirect&bpo=28724].)

time

On AIX, **`thread_time()`** is now implemented with `thread_cputime()` which has nanosecond resolution, rather than `clock_gettime(CLOCK_THREAD_CPUTIME_ID)` which has a resolution of 10 milliseconds. (Contributed by Batuhan Taskaya in bpo-40192 [https://bugs.python.org/issue?@action=redirect&bpo=40192])

sys

Added a new **`sys.platlibdir`** attribute: name of the platform-specific library directory. It is used to build the path of standard library and the paths of installed extension modules. It is equal to `"lib"` on most platforms. On Fedora and SuSE, it is equal to `"lib64"` on 64-bit platforms. (Contributed by Jan Matějek, Matěj Cepl, Charalampos Stratakis and Victor Stinner in bpo-1294959 [https://bugs.python.org/issue?@action=redirect&bpo=1294959].)

Previously, **`sys.stderr`** was block-buffered when non-interactive. Now `stderr` defaults to always being line-buffered. (Contributed by Jendrik Seipp in bpo-13601 [https://bugs.python.org/issue?@action=redirect&bpo=13601].)

tracemalloc

Added **`tracemalloc.reset_peak()`** to set the peak size of traced memory blocks to the current size, to measure the peak of specific pieces of code. (Contributed by Huon Wilson in bpo-40630 [https://bugs.python.org/issue?@action=redirect&bpo=40630].)

typing

PEP 593 [https://peps.python.org/pep-0593/] introduced an **`typing.Annotated`** type to decorate existing types with context-specific metadata and new `include_extras` parameter to **`typing.get_type_hints()`** to access the metadata at runtime. (Contributed by Till Varoquaux and Konstantin Kashin.)

unicodedata

The Unicode database has been updated to version 13.0.0. (bpo-39926 [https://bugs.python.org/issue?@action=redirect&bpo=39926]).

venv

The activation scripts provided by **`venv`** now all specify their prompt customization consistently by always using the value specified by `__VENV_PROMPT__`. Previously some scripts unconditionally used `__VENV_PROMPT__`, others only if it happened to be set (which was the default case), and one used `__VENV_NAME__` instead. (Contributed by Brett Cannon in bpo-37663 [https://bugs.python.org/issue?@action=redirect&bpo=37663].)

xml

White space characters within attributes are now preserved when serializing **`xml.etree.ElementTree`** to XML file. EOLNs are no longer normalized to “n”. This is the result of discussion about how to interpret section 2.11 of XML spec. (Contributed by Mefistotelis in bpo-39011 [https://bugs.python.org/issue?@action=redirect&bpo=39011].)

Optimizations

- Optimized the idiom for assignment a temporary variable in comprehensions. Now `for y in [expr]` in comprehensions is as fast as a simple assignment `y = expr`. For example:

  ```
  sums = [s for s in [0] for x in data for s in [s + x]]
  ```

  Unlike the `:=` operator this idiom does not leak a variable to the outer scope.

  (Contributed by Serhiy Storchaka in bpo-32856 [https://bugs.python.org/issue?@action=redirect&bpo=32856].)

- Optimized signal handling in multithreaded applications. If a thread different than the main thread gets a signal, the bytecode evaluation loop is no longer interrupted at each bytecode instruction to check for pending signals which cannot be handled. Only the main thread of the main interpreter can handle signals.

Previously, the bytecode evaluation loop was interrupted at each instruction until the main thread handles signals. (Contributed by Victor Stinner in bpo-40010 [https://bugs.python.org/issue?@action=redirect&bpo=40010].)

- Optimized the **`subprocess`** module on FreeBSD using `closefrom()`. (Contributed by Ed Maste, Conrad Meyer, Kyle Evans, Kubilay Kocak and Victor Stinner in bpo-38061 [https://bugs.python.org/issue?@action=redirect&bpo=38061].)

- **`PyLong_FromDouble()`** is now up to 1.87x faster for values that fit into long. (Contributed by Sergey Fedoseev in bpo-37986 [https://bugs.python.org/issue?@action=redirect&bpo=37986].)

- A number of Python builtins (**`range`**, **`tuple`**, **`set`**, **`frozenset`**, **`list`**, **`dict`**) are now sped up by using **PEP 590** [https://peps.python.org/pep-0590/] vectorcall protocol. (Contributed by Donghee Na, Mark Shannon, Jeroen Demeyer and Petr Viktorin in bpo-37207 [https://bugs.python.org/issue?@action=redirect&bpo=37207].)

- Optimized **`set.difference_update()`** for the case when the other set is much larger than the base set. (Suggested by Evgeny Kapun with code contributed by Michele Orrù in bpo-8425 [https://bugs.python.org/issue?@action=redirect&bpo=8425].)

- Python’s small object allocator (`obmalloc.c`) now allows (no more than) one empty arena to remain available for immediate reuse, without returning it to the OS. This prevents thrashing in simple loops where an arena could be created and destroyed anew on each iteration. (Contributed by Tim Peters in bpo-37257 [https://bugs.python.org/issue?@action=redirect&bpo=37257].)

- floor division of float operation now has a better performance. Also the message of `ZeroDivisionError` for this operation is updated. (Contributed by Donghee Na in bpo-39434 [https://bugs.python.org/issue?@action=redirect&bpo=39434].)
- Decoding short ASCII strings with UTF-8 and ascii codecs is now about 15% faster. (Contributed by Inada Naoki in bpo-37348 [https://bugs.python.org/issue?@action=redirect&bpo=37348].)

Here’s a summary of performance improvements from Python 3.4 through Python 3.9:

```
Python version                       3.4     3.5     3.6
3.7     3.8    3.9
--------------                       ---     ---     ---     --
-     ---    ---

Variable and attribute read access:
    read_local                       7.1     7.1     5.4
5.1     3.9    3.9
    read_nonlocal                    7.1     8.1     5.8
5.4     4.4    4.5
    read_global                     15.5    19.0    14.3
13.6     7.6    7.8
    read_builtin                    21.1    21.6    18.5
19.0     7.5    7.8
    read_classvar_from_class        25.6    26.5    20.7
19.5    18.4   17.9
    read_classvar_from_instance     22.8    23.5    18.8
17.1    16.4   16.9
    read_instancevar                32.4    33.1    28.0
26.3    25.4   25.3
    read_instancevar_slots          27.8    31.3    20.8
20.8    20.2   20.5
    read_namedtuple                 73.8    57.5    45.0
46.8    18.4   18.7
    read_boundmethod                37.6    37.9    29.6
26.9    27.7   41.1
```

```
Variable and attribute write access:
    write_local                        8.7     9.3     5.5
5.3     4.3    4.3
    write_nonlocal                    10.5    11.1     5.6
5.5     4.7    4.8
    write_global                      19.7    21.2    18.0
18.0    15.8   16.7
    write_classvar                    92.9    96.0   104.6
102.1    39.2   39.8
    write_instancevar                 44.6    45.8    40.0
38.9    35.5   37.4
    write_instancevar_slots           35.6    36.1    27.3
26.6    25.7   25.8

Data structure read access:
    read_list                         24.2    24.5    20.8
20.8    19.0   19.5
    read_deque                        24.7    25.5    20.2
20.6    19.8   20.2
    read_dict                         24.3    25.7    22.3
23.0    21.0   22.4
    read_strdict                      22.6    24.3    19.5
21.2    18.9   21.5

Data structure write access:
    write_list                        27.1    28.5    22.5
21.6    20.0   20.0
    write_deque                       28.7    30.1    22.7
21.8    23.5   21.7
    write_dict                        31.4    33.3    29.3
29.2    24.7   25.4
    write_strdict                     28.4    29.9    27.5
25.2    23.1   24.5

Stack (or queue) operations:
    list_append_pop                   93.4   112.7    75.4
74.2    50.8   50.6
    deque_append_pop                  43.5    57.0    49.4
49.2    42.5   44.2
    deque_append_popleft              43.7    57.3    49.7
49.7    42.8   46.4

Timing loop:
    loop_overhead                           0.5     0.6     0.4
0.3     0.3    0.3
```

These results were generated from the variable access benchmark script at: `Tools/scripts/var_access_benchmark.py`. The benchmark script displays timings in nanoseconds. The benchmarks were measured on an Intel® Core™ i7-4960HQ processor [https://ark.intel.com/content/www/us/en/ark/products/76088/intel-core-i7-4960hq-processor-6m-cache-up-to-3-80-ghz.html] running the macOS 64-bit builds found at python.org [https://www.python.org/downloads/macos/].

Deprecated

- The distutils `bdist_msi` command is now deprecated, use `bdist_wheel` (wheel packages) instead. (Contributed by Hugo van Kemenade in bpo-39586 [https://bugs.python.org/issue?@action=redirect&bpo=39586].)

- Currently **`math.factorial()`** accepts **`float`** instances with non-negative integer values (like `5.0`). It raises a **`ValueError`** for non-integral and negative floats. It is now deprecated. In future Python versions it will raise a **`TypeError`** for all floats. (Contributed by Serhiy Storchaka in bpo-37315 [https://bugs.python.org/issue?@action=redirect&bpo=37315].)

- The **`parser`** and **`symbol`** modules are deprecated and will be removed in future versions of Python. For the majority of use cases, users can leverage the Abstract Syntax Tree (AST) generation and compilation stage, using the **`ast`** module.

- The Public C API functions **`PyParser_SimpleParseStringFlags()`**, **`PyParser_SimpleParseStringFlagsFilename()`**,

`PyParser_SimpleParseFileFlags()` and **`PyNode_Compile()`** are deprecated and will be removed in Python 3.10 together with the old parser.

- Using `NotImplemented` in a boolean context has been deprecated, as it is almost exclusively the result of incorrect rich comparator implementations. It will be made a `TypeError` in a future version of Python. (Contributed by Josh Rosenberg in bpo-35712 [https://bugs.python.org/issue?@action=redirect&bpo=35712].)

- The `random` module currently accepts any hashable type as a possible seed value. Unfortunately, some of those types are not guaranteed to have a deterministic hash value. After Python 3.9, the module will restrict its seeds to `None`, `int`, `float`, `str`, `bytes`, and `bytearray`.

- Opening the `GzipFile` file for writing without specifying the *mode* argument is deprecated. In future Python versions it will always be opened for reading by default. Specify the *mode* argument for opening it for writing and silencing a warning. (Contributed by Serhiy Storchaka in bpo-28286 [https://bugs.python.org/issue?@action=redirect&bpo=28286].)

- Deprecated the `split()` method of **`_tkinter.TkappType`** in favour of the `splitlist()` method which has more consistent and predictable behavior. (Contributed by Serhiy Storchaka in bpo-38371 [https://bugs.python.org/issue?@action=redirect&bpo=38371].)

- The explicit passing of coroutine objects to `asyncio.wait()` has been deprecated and will be removed in version 3.11. (Contributed by Yury Selivanov and Kyle Stanley in bpo-34790 [https://bugs.python.org/issue?@action=redirect&bpo=34790].)

- binhex4 and hexbin4 standards are now deprecated. The **`binhex`** module and the following [**`binascii`**](#) functions are now deprecated:

    - **`b2a_hqx()`**, **`a2b_hqx()`**
    - **`rlecode_hqx()`**, **`rledecode_hqx()`**

  (Contributed by Victor Stinner in bpo-39353 [https://bugs.python.org/issue?@action=redirect&bpo=39353].)

- **`ast`** classes `slice`, `Index` and `ExtSlice` are considered deprecated and will be removed in future Python versions. `value` itself should be used instead of `Index(value)`. `Tuple(slices, Load())` should be used instead of `ExtSlice(slices)`. (Contributed by Serhiy Storchaka in bpo-34822 [https://bugs.python.org/issue?@action=redirect&bpo=34822].)

- **`ast`** classes `Suite`, `Param`, `AugLoad` and `AugStore` are considered deprecated and will be removed in future Python versions. They were not generated by the parser and not accepted by the code generator in Python 3. (Contributed by Batuhan Taskaya in bpo-39639 [https://bugs.python.org/issue?@action=redirect&bpo=39639] and bpo-39969 [https://bugs.python.org/issue?@action=redirect&bpo=39969] and Serhiy Storchaka in bpo-39988 [https://bugs.python.org/issue?@action=redirect&bpo=39988].)

- The **`PyEval_InitThreads()`** and **`PyEval_ThreadsInitialized()`** functions are now deprecated and will be removed in Python 3.11. Calling **`PyEval_InitThreads()`** now does nothing. The GIL is initialized by **`Py_Initialize()`** since Python 3.7. (Contributed by Victor Stinner in bpo-39877 [https://bugs.python.org/issue?@action=redirect&bpo=39877].)

- Passing `None` as the first argument to the **`shlex.split()`** function has been deprecated. (Contributed by Zackery

Spytz in bpo-33262 [https://bugs.python.org/issue?@action=redirect&bpo=33262].)

- `smtpd.MailmanProxy()` is now deprecated as it is unusable without an external module, `mailman`. (Contributed by Samuel Colvin in bpo-35800 [https://bugs.python.org/issue?@action=redirect&bpo=35800].)

- The `lib2to3` module now emits a `PendingDeprecationWarning`. Python 3.9 switched to a PEG parser (see **PEP 617** [https://peps.python.org/pep-0617/]), and Python 3.10 may include new language syntax that is not parsable by lib2to3's LL(1) parser. The `lib2to3` module may be removed from the standard library in a future Python version. Consider third-party alternatives such as LibCST [https://libcst.readthedocs.io/] or parso [https://parso.readthedocs.io/]. (Contributed by Carl Meyer in bpo-40360 [https://bugs.python.org/issue?@action=redirect&bpo=40360].)

- The *random* parameter of `random.shuffle()` has been deprecated. (Contributed by Raymond Hettinger in bpo-40465 [https://bugs.python.org/issue?@action=redirect&bpo=40465])

Removed

- The erroneous version at `unittest.mock.__version__` has been removed.
- `nntplib.NNTP`: `xpath()` and `xgtitle()` methods have been removed. These methods are deprecated since Python 3.3. Generally, these extensions are not supported or not enabled by NNTP server administrators. For `xgtitle()`, please use `nntplib.NNTP.descriptions()` or `nntplib.NNTP.description()` instead. (Contributed by Donghee Na in bpo-39366 [https://bugs.python.org/issue?@action=redirect&bpo=39366].)

- **`array.array`**: `tostring()` and `fromstring()` methods have been removed. They were aliases to `tobytes()` and `frombytes()`, deprecated since Python 3.2. (Contributed by Victor Stinner in bpo-38916 [https://bugs.python.org/issue?@action=redirect&bpo=38916].)
- The undocumented `sys.callstats()` function has been removed. Since Python 3.7, it was deprecated and always returned **`None`**. It required a special build option `CALL_PROFILE` which was already removed in Python 3.7. (Contributed by Victor Stinner in bpo-37414 [https://bugs.python.org/issue?@action=redirect&bpo=37414].)
- The `sys.getcheckinterval()` and `sys.setcheckinterval()` functions have been removed. They were deprecated since Python 3.2. Use **`sys.getswitchinterval()`** and **`sys.setswitchinterval()`** instead. (Contributed by Victor Stinner in bpo-37392 [https://bugs.python.org/issue?@action=redirect&bpo=37392].)
- The C function `PyImport_Cleanup()` has been removed. It was documented as: "Empty the module table. For internal use only." (Contributed by Victor Stinner in bpo-36710 [https://bugs.python.org/issue?@action=redirect&bpo=36710].)
- `_dummy_thread` and `dummy_threading` modules have been removed. These modules were deprecated since Python 3.7 which requires threading support. (Contributed by Victor Stinner in bpo-37312 [https://bugs.python.org/issue?@action=redirect&bpo=37312].)
- `aifc.openfp()` alias to `aifc.open()`, `sunau.openfp()` alias to `sunau.open()`, and `wave.openfp()` alias to **`wave.open()`** have been removed. They were deprecated since Python 3.7. (Contributed by Victor Stinner in bpo-37320 [https://bugs.python.org/issue?@action=redirect&bpo=37320].)
- The **`isAlive()`** method of **`threading.Thread`** has been removed. It was deprecated since Python 3.8. Use **`is_alive()`** instead. (Contributed by Donghee Na in bpo-37804 [https://bugs.python.org/issue?@action=redirect&bpo=37804].)

- Methods `getchildren()` and `getiterator()` of classes **`ElementTree`** and **`Element`** in the **`ElementTree`** module have been removed. They were deprecated in Python 3.2. Use `iter(x)` or `list(x)` instead of `x.getchildren()` and `x.iter()` or `list(x.iter())` instead of `x.getiterator()`. (Contributed by Serhiy Storchaka in bpo-36543 [https://bugs.python.org/issue?@action=redirect&bpo=36543].)
- The old **`plistlib`** API has been removed, it was deprecated since Python 3.4. Use the **`load()`**, **`loads()`**, **`dump()`**, and **`dumps()`** functions. Additionally, the *use_builtin_types* parameter was removed, standard **`bytes`** objects are always used instead. (Contributed by Jon Janzen in bpo-36409 [https://bugs.python.org/issue?@action=redirect&bpo=36409].)
- The C function `PyGen_NeedsFinalizing` has been removed. It was not documented, tested, or used anywhere within CPython after the implementation of **PEP 442** [https://peps.python.org/pep-0442/]. Patch by Joannah Nanjekye. (Contributed by Joannah Nanjekye in bpo-15088 [https://bugs.python.org/issue?@action=redirect&bpo=15088])
- `base64.encodestring()` and `base64.decodestring()`, aliases deprecated since Python 3.1, have been removed: use **`base64.encodebytes()`** and **`base64.decodebytes()`** instead. (Contributed by Victor Stinner in bpo-39351 [https://bugs.python.org/issue?@action=redirect&bpo=39351].)
- `fractions.gcd()` function has been removed, it was deprecated since Python 3.5 (bpo-22486 [https://bugs.python.org/issue?@action=redirect&bpo=22486]): use **`math.gcd()`** instead. (Contributed by Victor Stinner in bpo-39350 [https://bugs.python.org/issue?@action=redirect&bpo=39350].)
- The *buffering* parameter of **`bz2.BZ2File`** has been removed. Since Python 3.0, it was ignored and using it emitted a **`DeprecationWarning`**. Pass an open file object to control how the file is opened. (Contributed by Victor Stinner in bpo-39357 [https://bugs.python.org/issue?@action=redirect&bpo=39357].)

- The *encoding* parameter of `json.loads()` has been removed. As of Python 3.1, it was deprecated and ignored; using it has emitted a `DeprecationWarning` since Python 3.8. (Contributed by Inada Naoki in bpo-39377 [https://bugs.python.org/issue?@action=redirect&bpo=39377])
- `with (await asyncio.lock):` and `with (yield from asyncio.lock):` statements are not longer supported, use `async with lock` instead. The same is correct for `asyncio.Condition` and `asyncio.Semaphore`. (Contributed by Andrew Svetlov in bpo-34793 [https://bugs.python.org/issue?@action=redirect&bpo=34793].)
- The **`sys.getcounts()`** function, the `-X showalloccount` command line option and the `show_alloc_count` field of the C structure `PyConfig` have been removed. They required a special Python build by defining `COUNT_ALLOCS` macro. (Contributed by Victor Stinner in bpo-39489 [https://bugs.python.org/issue?@action=redirect&bpo=39489].)
- The `_field_types` attribute of the `typing.NamedTuple` class has been removed. It was deprecated since Python 3.8. Use the `__annotations__` attribute instead. (Contributed by Serhiy Storchaka in bpo-40182 [https://bugs.python.org/issue?@action=redirect&bpo=40182].)
- The **`symtable.SymbolTable.has_exec()`** method has been removed. It was deprecated since 2006, and only returning `False` when it’s called. (Contributed by Batuhan Taskaya in bpo-40208 [https://bugs.python.org/issue?@action=redirect&bpo=40208])
- The **`asyncio.Task.current_task()`** and **`asyncio.Task.all_tasks()`** have been removed. They were deprecated since Python 3.7 and you can use `asyncio.current_task()` and `asyncio.all_tasks()` instead. (Contributed by Rémi Lapeyre in bpo-40967 [https://bugs.python.org/issue?@action=redirect&bpo=40967])
- The `unescape()` method in the `html.parser.HTMLParser` class has been removed (it was deprecated since Python 3.4).

`html.unescape()` should be used for converting character references to the corresponding unicode characters.

Porting to Python 3.9

This section lists previously described changes and other bugfixes that may require changes to your code.

Changes in the Python API

- **`__import__()`** and **`importlib.util.resolve_name()`** now raise **`ImportError`** where it previously raised **`ValueError`**. Callers catching the specific exception type and supporting both Python 3.9 and earlier versions will need to catch both using `except (ImportError, ValueError):`.
- The **`venv`** activation scripts no longer special-case when `__VENV_PROMPT__` is set to `""`.
- The **`select.epoll.unregister()`** method no longer ignores the **`EBADF`** error. (Contributed by Victor Stinner in bpo-39239 [https://bugs.python.org/issue?@action=redirect&bpo=39239].)
- The *compresslevel* parameter of **`bz2.BZ2File`** became keyword-only, since the *buffering* parameter has been removed. (Contributed by Victor Stinner in bpo-39357 [https://bugs.python.org/issue?@action=redirect&bpo=39357].)
- Simplified AST for subscription. Simple indices will be represented by their value, extended slices will be represented as tuples. `Index(value)` will return a `value` itself, `ExtSlice(slices)` will return `Tuple(slices, Load())`. (Contributed by Serhiy Storchaka in bpo-34822 [https://bugs.python.org/issue?@action=redirect&bpo=34822].)
- The **`importlib`** module now ignores the **`PYTHONCASEOK`** environment variable when the **`-E`** or **`-I`** command line options are being used.
- The *encoding* parameter has been added to the classes **`ftplib.FTP`** and **`ftplib.FTP_TLS`** as a keyword-only

parameter, and the default encoding is changed from Latin-1 to UTF-8 to follow **RFC 2640** [https://datatracker.ietf.org/doc/html/rfc2640.html].

- `asyncio.loop.shutdown_default_executor()` has been added to `AbstractEventLoop`, meaning alternative event loops that inherit from it should have this method defined. (Contributed by Kyle Stanley in bpo-34037 [https://bugs.python.org/issue?@action=redirect&bpo=34037].)
- The constant values of future flags in the `__future__` module is updated in order to prevent collision with compiler flags. Previously `PyCF_ALLOW_TOP_LEVEL_AWAIT` was clashing with `CO_FUTURE_DIVISION`. (Contributed by Batuhan Taskaya in bpo-39562 [https://bugs.python.org/issue?@action=redirect&bpo=39562])
- `array('u')` now uses **`wchar_t`** as C type instead of `Py_UNICODE`. This change doesn't affect to its behavior because `Py_UNICODE` is alias of **`wchar_t`** since Python 3.3. (Contributed by Inada Naoki in bpo-34538 [https://bugs.python.org/issue?@action=redirect&bpo=34538].)
- The `logging.getLogger()` API now returns the root logger when passed the name `'root'`, whereas previously it returned a non-root logger named `'root'`. This could affect cases where user code explicitly wants a non-root logger named `'root'`, or instantiates a logger using `logging.getLogger(__name__)` in some top-level module called `'root.py'`. (Contributed by Vinay Sajip in bpo-37742 [https://bugs.python.org/issue?@action=redirect&bpo=37742].)
- Division handling of `PurePath` now returns `NotImplemented` instead of raising a `TypeError` when passed something other than an instance of `str` or `PurePath`. This allows creating compatible classes that don't inherit from those mentioned types. (Contributed by Roger Aiudi in bpo-34775 [https://bugs.python.org/issue?@action=redirect&bpo=34775]).
- Starting with Python 3.9.5 the `ipaddress` module no longer accepts any leading zeros in IPv4 address strings. Leading zeros are ambiguous and interpreted as octal

notation by some libraries. For example the legacy function `socket.inet_aton()` treats leading zeros as octal notatation. glibc implementation of modern `inet_pton()` does not accept any leading zeros. (Contributed by Christian Heimes in bpo-36384 [https://bugs.python.org/issue?@action=redirect&bpo=36384]).

- `codecs.lookup()` now normalizes the encoding name the same way as `encodings.normalize_encoding()`, except that `codecs.lookup()` also converts the name to lower case. For example, `"latex+latin1"` encoding name is now normalized to `"latex_latin1"`. (Contributed by Jordon Xu in bpo-37751 [https://bugs.python.org/issue?@action=redirect&bpo=37751].)

Changes in the C API

- Instances of heap-allocated types (such as those created with `PyType_FromSpec()` and similar APIs) hold a reference to their type object since Python 3.8. As indicated in the “Changes in the C API” of Python 3.8, for the vast majority of cases, there should be no side effect but for types that have a custom `tp_traverse` function, ensure that all custom `tp_traverse` functions of heap-allocated types visit the object’s type.

  Example:

  ```
  int
  foo_traverse(PyObject *self, visitproc visit, void
  *arg)
  {
  // Rest of the traverse function
  #if PY_VERSION_HEX >= 0x03090000
      // This was not needed before Python 3.9 (Python
  issue 35810 and 40217)
      Py_VISIT(Py_TYPE(self));
  #endif
  }
  ```

If your traverse function delegates to `tp_traverse` of its base class (or another type), ensure that `Py_TYPE(self)` is visited only once. Note that only heap type are expected to visit the type in `tp_traverse`.

For example, if your `tp_traverse` function includes:

```
base->tp_traverse(self, visit, arg)
```

then add:

```
#if PY_VERSION_HEX >= 0x03090000
    // This was not needed before Python 3.9 (bpo-35810
and bpo-40217)
    if (base->tp_flags & Py_TPFLAGS_HEAPTYPE) {
        // a heap type's tp_traverse already visited
Py_TYPE(self)
    } else {
        Py_VISIT(Py_TYPE(self));
    }
#else
```

(See bpo-35810 [https://bugs.python.org/issue?@action=redirect&bpo=35810] and bpo-40217 [https://bugs.python.org/issue?@action=redirect&bpo=40217] for more information.)

- The functions `PyEval_CallObject`, `PyEval_CallFunction`, `PyEval_CallMethod` and `PyEval_CallObjectWithKeywords` are deprecated. Use **`PyObject_Call()`** and its variants instead. (See more details in bpo-29548 [https://bugs.python.org/issue?@action=redirect&bpo=29548].)

CPython bytecode changes

- The **`LOAD_ASSERTION_ERROR`** opcode was added for handling the **`assert`** statement. Previously, the assert statement would not work correctly if the **`AssertionError`** exception was being shadowed. (Contributed by Zackery Spytz in bpo-34880 [https://bugs.python.org/issue?@action=redirect&bpo=34880].)

- The **`COMPARE_OP`** opcode was split into four distinct instructions:
  - `COMPARE_OP` for rich comparisons
  - `IS_OP` for ‘is’ and ‘is not’ tests
  - `CONTAINS_OP` for ‘in’ and ‘not in’ tests
  - `JUMP_IF_NOT_EXC_MATCH` for checking exceptions in ‘try-except’ statements.

  (Contributed by Mark Shannon in bpo-39156 [https://bugs.python.org/issue?@action=redirect&bpo=39156].)

Build Changes

- Added `--with-platlibdir` option to the `configure` script: name of the platform-specific library directory, stored in the new **`sys.platlibdir`** attribute. See **`sys.platlibdir`** attribute for more information. (Contributed by Jan Matějek, Matěj Cepl, Charalampos Stratakis and Victor Stinner in bpo-1294959 [https://bugs.python.org/issue?@action=redirect&bpo=1294959].)
- The `COUNT_ALLOCS` special build macro has been removed. (Contributed by Victor Stinner in bpo-39489 [https://bugs.python.org/issue?@action=redirect&bpo=39489].)
- On non-Windows platforms, the **`setenv()`** and **`unsetenv()`** functions are now required to build Python. (Contributed by Victor Stinner in bpo-39395 [https://bugs.python.org/issue?@action=redirect&bpo=39395].)
- On non-Windows platforms, creating `bdist_wininst` installers is now officially unsupported. (See bpo-10945 [https://bugs.python.org/issue?@action=redirect&bpo=10945] for more details.)

- When building Python on macOS from source, `_tkinter` now links with non-system Tcl and Tk frameworks if they are installed in `/Library/Frameworks`, as had been the case on older releases of macOS. If a macOS SDK is explicitly configured, by using **`--enable-universalsdk`** or `-isysroot`, only the SDK itself is searched. The default behavior can still be overridden with `--with-tcltk-includes` and `--with-tcltk-libs`. (Contributed by Ned Deily in bpo-34956 [https://bugs.python.org/issue?@action=redirect&bpo=34956].)
- Python can now be built for Windows 10 ARM64. (Contributed by Steve Dower in bpo-33125 [https://bugs.python.org/issue?@action=redirect&bpo=33125].)
- Some individual tests are now skipped when `--pgo` is used. The tests in question increased the PGO task time significantly and likely didn’t help improve optimization of the final executable. This speeds up the task by a factor of about 15x. Running the full unit test suite is slow. This change may result in a slightly less optimized build since not as many code branches will be executed. If you are willing to wait for the much slower build, the old behavior can be restored using `./configure [..] PROFILE_TASK="-m test --pgo-extended"`. We make no guarantees as to which PGO task set produces a faster build. Users who care should run their own relevant benchmarks as results can depend on the environment, workload, and compiler tool chain. (See bpo-36044 [https://bugs.python.org/issue?@action=redirect&bpo=36044] and bpo-37707 [https://bugs.python.org/issue?@action=redirect&bpo=37707] for more details.)

C API Changes

New Features

- **PEP 573** [https://peps.python.org/pep-0573/]: Added `PyType_FromModuleAndSpec()` to associate a module with a class; `PyType_GetModule()` and `PyType_GetModuleState()` to retrieve the module and its state; and `PyCMethod` and `METH_METHOD` to allow a method to access the class it was defined in. (Contributed by Marcel Plch and Petr Viktorin in bpo-38787 [https://bugs.python.org/issue?@action=redirect&bpo=38787].)

- Added `PyFrame_GetCode()` function: get a frame code. Added `PyFrame_GetBack()` function: get the frame next outer frame. (Contributed by Victor Stinner in bpo-40421 [https://bugs.python.org/issue?@action=redirect&bpo=40421].)

- Added `PyFrame_GetLineNumber()` to the limited C API. (Contributed by Victor Stinner in bpo-40421 [https://bugs.python.org/issue?@action=redirect&bpo=40421].)

- Added `PyThreadState_GetInterpreter()` and `PyInterpreterState_Get()` functions to get the interpreter. Added `PyThreadState_GetFrame()` function to get the current frame of a Python thread state. Added `PyThreadState_GetID()` function: get the unique identifier of a Python thread state. (Contributed by Victor Stinner in bpo-39947 [https://bugs.python.org/issue?@action=redirect&bpo=39947].)

- Added a new public `PyObject_CallNoArgs()` function to the C API, which calls a callable Python object without any arguments. It is the most efficient way to call a callable Python object without any argument. (Contributed by Victor Stinner in bpo-37194 [https://bugs.python.org/issue?@action=redirect&bpo=37194].)

- Changes in the limited C API (if `Py_LIMITED_API` macro is defined):

  - Provide **`Py_EnterRecursiveCall()`** and **`Py_LeaveRecursiveCall()`** as regular functions for the limited API. Previously, there were defined as macros, but these macros didn't compile with the limited C API which cannot access `PyThreadState.recursion_depth` field (the structure is opaque in the limited C API).
  - `PyObject_INIT()` and `PyObject_INIT_VAR()` become regular "opaque" function to hide implementation details.

  (Contributed by Victor Stinner in bpo-38644 [https://bugs.python.org/issue?@action=redirect&bpo=38644] and bpo-39542 [https://bugs.python.org/issue?@action=redirect&bpo=39542].)

- The **`PyModule_AddType()`** function is added to help adding a type to a module. (Contributed by Donghee Na in bpo-40024 [https://bugs.python.org/issue?@action=redirect&bpo=40024].)

- Added the functions **`PyObject_GC_IsTracked()`** and **`PyObject_GC_IsFinalized()`** to the public API to allow to query if Python objects are being currently tracked or have been already finalized by the garbage collector respectively. (Contributed by Pablo Galindo Salgado in bpo-40241 [https://bugs.python.org/issue?@action=redirect&bpo=40241].)

- Added **`_PyObject_FunctionStr()`** to get a user-friendly string representation of a function-like object. (Patch by Jeroen Demeyer in bpo-37645 [https://bugs.python.org/issue?@action=redirect&bpo=37645].)

- Added **`PyObject_CallOneArg()`** for calling an object with one positional argument (Patch by Jeroen Demeyer in bpo-37483 [https://bugs.python.org/issue?@action=redirect&bpo=37483].)

Porting to Python 3.9

- `PyInterpreterState.eval_frame` (**PEP 523** [https://peps.python.org/pep-0523/]) now requires a new mandatory *tstate* parameter (`PyThreadState*`). (Contributed by Victor Stinner in bpo-38500 [https://bugs.python.org/issue?@action=redirect&bpo=38500].)
- Extension modules: **`m_traverse`**, **`m_clear`** and **`m_free`** functions of **`PyModuleDef`** are no longer called if the module state was requested but is not allocated yet. This is the case immediately after the module is created and before the module is executed (**`Py_mod_exec`** function). More precisely, these functions are not called if **`m_size`** is greater than 0 and the module state (as returned by **`PyModule_GetState()`**) is `NULL`.

  Extension modules without module state (`m_size <= 0`) are not affected.
- If **`Py_AddPendingCall()`** is called in a subinterpreter, the function is now scheduled to be called from the subinterpreter, rather than being called from the main interpreter. Each subinterpreter now has its own list of scheduled calls. (Contributed by Victor Stinner in bpo-39984 [https://bugs.python.org/issue?@action=redirect&bpo=39984].)
- The Windows registry is no longer used to initialize **`sys.path`** when the `-E` option is used (if **`PyConfig.use_environment`** is set to `0`). This is significant when embedding Python on Windows. (Contributed by Zackery Spytz in bpo-8901 [https://bugs.python.org/issue?@action=redirect&bpo=8901].)
- The global variable **`PyStructSequence_UnnamedField`** is now a constant and refers to a constant string. (Contributed by

Serhiy Storchaka in bpo-38650 [https://bugs.python.org/issue?@action=redirect&bpo=38650].)

- The **`PyGC_Head`** structure is now opaque. It is only defined in the internal C API (`pycore_gc.h`). (Contributed by Victor Stinner in bpo-40241 [https://bugs.python.org/issue?@action=redirect&bpo=40241].)
- The `Py_UNICODE_COPY`, `Py_UNICODE_FILL`, `PyUnicode_WSTR_LENGTH`, **`PyUnicode_FromUnicode()`**, **`PyUnicode_AsUnicode()`**, `_PyUnicode_AsUnicode`, and **`PyUnicode_AsUnicodeAndSize()`** are marked as deprecated in C. They have been deprecated by **PEP 393** [https://peps.python.org/pep-0393/] since Python 3.3. (Contributed by Inada Naoki in bpo-36346 [https://bugs.python.org/issue?@action=redirect&bpo=36346].)
- The `Py_FatalError()` function is replaced with a macro which logs automatically the name of the current function, unless the `Py_LIMITED_API` macro is defined. (Contributed by Victor Stinner in bpo-39882 [https://bugs.python.org/issue?@action=redirect&bpo=39882].)
- The vectorcall protocol now requires that the caller passes only strings as keyword names. (See bpo-37540 [https://bugs.python.org/issue?@action=redirect&bpo=37540] for more information.)
- Implementation details of a number of macros and functions are now hidden:
  - `PyObject_IS_GC()` macro was converted to a function.
  - The **`PyObject_NEW()`** macro becomes an alias to the `PyObject_New` macro, and the **`PyObject_NEW_VAR()`** macro becomes an alias to the `PyObject_NewVar` macro. They no longer access directly the `PyTypeObject.tp_basicsize` member.

  - **`PyObject_GET_WEAKREFS_LISTPTR()`** macro was converted to a function: the macro accessed directly the **`PyTypeObject.tp_weaklistoffset`** member.
  - **`PyObject_CheckBuffer()`** macro was converted to a function: the macro accessed directly the **`PyTypeObject.tp_as_buffer`** member.
  - **`PyIndex_Check()`** is now always declared as an opaque function to hide implementation details: removed the `PyIndex_Check()` macro. The macro accessed directly the **`PyTypeObject.tp_as_number`** member.

(See bpo-40170 [https://bugs.python.org/issue?@action=redirect&bpo=40170] for more details.)

Removed

- Excluded `PyFPE_START_PROTECT()` and `PyFPE_END_PROTECT()` macros of `pyfpe.h` from the limited C API. (Contributed by Victor Stinner in bpo-38835 [https://bugs.python.org/issue?@action=redirect&bpo=38835].)

- The `tp_print` slot of PyTypeObject has been removed. It was used for printing objects to files in Python 2.7 and before. Since Python 3.0, it has been ignored and unused. (Contributed by Jeroen Demeyer in bpo-36974 [https://bugs.python.org/issue?@action=redirect&bpo=36974].)

- Changes in the limited C API (if `Py_LIMITED_API` macro is defined):

  - Excluded the following functions from the limited C API:
    - `PyThreadState_DeleteCurrent()` (Contributed by Joannah Nanjekye in bpo-37878 [https://bugs.python.org/issue?@action=redirect&bpo=37878].)
    - `_Py_CheckRecursionLimit`

        - `_Py_NewReference()`
        - `_Py_ForgetReference()`
        - `_PyTraceMalloc_NewReference()`
        - `_Py_GetRefTotal()`
        - The trashcan mechanism which never worked in the limited C API.
        - `PyTrash_UNWIND_LEVEL`
        - `Py_TRASHCAN_BEGIN_CONDITION`
        - `Py_TRASHCAN_BEGIN`
        - `Py_TRASHCAN_END`
        - `Py_TRASHCAN_SAFE_BEGIN`
        - `Py_TRASHCAN_SAFE_END`
    - Moved following functions and definitions to the internal C API:
        - `_PyDebug_PrintTotalRefs()`
        - `_Py_PrintReferences()`
        - `_Py_PrintReferenceAddresses()`
        - `_Py_tracemalloc_config`
        - `_Py_AddToAllObjects()` (specific to `Py_TRACE_REFS` build)

  (Contributed by Victor Stinner in bpo-38644 [https://bugs.python.org/issue?@action=redirect&bpo=38644] and bpo-39542 [https://bugs.python.org/issue?@action=redirect&bpo=39542].)

- Removed `_PyRuntime.getframe` hook and removed `_PyThreadState_GetFrame` macro which was an alias to `_PyRuntime.getframe`. They were only exposed by the internal C API. Removed also `PyThreadFrameGetter` type. (Contributed by Victor Stinner in bpo-39946 [https://bugs.python.org/issue?@action=redirect&bpo=39946].)

- Removed the following functions from the C API. Call **`PyGC_Collect()`** explicitly to clear all free lists. (Contributed by Inada Naoki and Victor Stinner in bpo-37340 [https://bugs.python.org/issue?@action=redirect&bpo=37340],

bpo-38896 [https://bugs.python.org/issue?@action=redirect&bpo=38896] and bpo-40428 [https://bugs.python.org/issue?@action=redirect&bpo=40428].)

  - `PyAsyncGen_ClearFreeLists()`
  - `PyContext_ClearFreeList()`
  - `PyDict_ClearFreeList()`
  - `PyFloat_ClearFreeList()`
  - `PyFrame_ClearFreeList()`
  - `PyList_ClearFreeList()`
  - `PyMethod_ClearFreeList()` and `PyCFunction_ClearFreeList()`: the free lists of bound method objects have been removed.
  - `PySet_ClearFreeList()`: the set free list has been removed in Python 3.4.
  - `PyTuple_ClearFreeList()`
  - `PyUnicode_ClearFreeList()`: the Unicode free list has been removed in Python 3.3.

- Removed `_PyUnicode_ClearStaticStrings()` function. (Contributed by Victor Stinner in bpo-39465 [https://bugs.python.org/issue?@action=redirect&bpo=39465].)

- Removed `Py_UNICODE_MATCH`. It has been deprecated by **PEP 393** [https://peps.python.org/pep-0393/], and broken since Python 3.3. The **`PyUnicode_Tailmatch()`** function can be used instead. (Contributed by Inada Naoki in bpo-36346 [https://bugs.python.org/issue?@action=redirect&bpo=36346].)

- Cleaned header files of interfaces defined but with no implementation. The public API symbols being removed are: `_PyBytes_InsertThousandsGroupingLocale`, `_PyBytes_InsertThousandsGrouping`, `_Py_InitializeFromArgs`, `_Py_InitializeFromWideArgs`, `_PyFloat_Repr`, `_PyFloat_Digits`, `_PyFloat_DigitsInit`, `PyFrame_ExtendStack`, `_PyAIterWrapper_Type`, `PyNullImporter_Type`, `PyCmpWrapper_Type`, `PySortWrapper_Type`, `PyNoArgsFunction`. (Contributed by Pablo Galindo Salgado in bpo-39372 [https://bugs.python.org/issue?@action=redirect&bpo=39372].)

Notable changes in Python 3.9.1

typing

The behavior of **`typing.Literal`** was changed to conform with **PEP 586** [https://peps.python.org/pep-0586/] and to match the behavior of static type checkers specified in the PEP.

1. `Literal` now de-duplicates parameters.
2. Equality comparisons between `Literal` objects are now order independent.
3. `Literal` comparisons now respect types. For example, `Literal[0] == Literal[False]` previously evaluated to `True`. It is now `False`. To support this change, the internally used type cache now supports differentiating types.
4. `Literal` objects will now raise a **`TypeError`** exception during equality comparisons if any of their parameters are not hashable. Note that declaring `Literal` with mutable parameters will not throw an error:

```
>>> from typing import Literal
>>> Literal[{0}]
>>> Literal[{0}] == Literal[{False}]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unhashable type: 'set'
```

(Contributed by Yurii Karabas in bpo-42345 [https://bugs.python.org/issue?@action=redirect&bpo=42345].)

macOS 11.0 (Big Sur) and Apple Silicon Mac support

As of 3.9.1, Python now fully supports building and running on macOS 11.0 (Big Sur) and on Apple Silicon Macs (based on the `ARM64` architecture). A new universal build variant, `universal2`, is now available to natively support both `ARM64` and `Intel 64` in one set of executables. Binaries can also now be built on current versions of macOS to be deployed on a range of older macOS versions (tested to 10.9) while making some newer OS functions and options conditionally available based on the operating system version in use at runtime (“weaklinking”).

(Contributed by Ronald Oussoren and Lawrence D’Anna in bpo-41100 [https://bugs.python.org/issue?@action=redirect&bpo=41100].)

Notable changes in Python 3.9.2

collections.abc

`collections.abc.Callable` generic now flattens type parameters, similar to what **`typing.Callable`** currently does. This means that `collections.abc.Callable[[int, str], str]` will have `__args__` of `(int, str, str)`; previously this was `([int, str], str)`. To allow this change, **`types.GenericAlias`** can now be subclassed, and a subclass will be returned when subscripting the **`collections.abc.Callable`** type. Code which accesses the arguments via **`typing.get_args()`** or `__args__` need to account for this change. A **`DeprecationWarning`** may be emitted for invalid forms of parameterizing **`collections.abc.Callable`** which may have passed silently in Python 3.9.1. This **`DeprecationWarning`** will become a **`TypeError`** in Python 3.10. (Contributed by Ken Jin in bpo-42195 [https://bugs.python.org/issue?@action=redirect&bpo=42195].)

urllib.parse

Earlier Python versions allowed using both `;` and `&` as query parameter separators in `urllib.parse.parse_qs()` and `urllib.parse.parse_qsl()`. Due to security concerns, and to conform with newer W3C recommendations, this has been changed to allow only a single separator key, with `&` as the default. This change also affects **`cgi.parse()`** and **`cgi.parse_multipart()`** as they use the affected functions internally. For more details, please see their respective documentation. (Contributed by Adam Goldschmidt, Senthil Kumaran and Ken Jin in bpo-42967 [https://bugs.python.org/issue?@action=redirect&bpo=42967].)

Notable changes in Python 3.9.3

A security fix alters the `ftplib.FTP` behavior to not trust the IPv4 address sent from the remote server when setting up a passive data channel. We reuse the ftp server IP address instead. For unusual code requiring the old behavior, set a `trust_server_pasv_ipv4_address` attribute on your FTP instance to `True`. (See gh-87451 [https://github.com/python/cpython/issues/87451])

Notable changes in Python 3.9.5

urllib.parse

The presence of newline or tab characters in parts of a URL allows for some forms of attacks. Following the WHATWG specification that updates **RFC 3986** [https://datatracker.ietf.org/doc/html/rfc3986.html], ASCII newline `\n`, `\r` and tab `\t` characters are stripped from the URL by the parser in `urllib.parse` preventing such attacks. The removal characters are controlled by a new module level variable `urllib.parse._UNSAFE_URL_BYTES_TO_REMOVE`. (See gh-88048 [https://github.com/python/cpython/issues/88048])

Notable security feature in 3.9.14

Converting between **`int`** and **`str`** in bases other than 2 (binary), 4, 8 (octal), 16 (hexadecimal), or 32 such as base 10 (decimal) now raises a **`ValueError`** if the number of digits in string form is above a limit to avoid potential denial of service attacks due to the algorithmic complexity. This is a mitigation for **CVE 2020-10735** [https://www.cve.org/CVERecord?id=CVE-2020-10735]. This limit can be configured or disabled by environment variable, command line flag, or **`sys`** APIs. See the integer string conversion length limitation documentation. The default limit is 4300 digits in string form.

Notable changes in 3.9.17

tarfile

- The extraction methods in **`tarfile`**, and **`shutil.unpack_archive()`**, have a new a *filter* argument that allows limiting tar features than may be surprising or dangerous, such as creating files outside the destination directory. See Extraction filters for details. In Python 3.12, use without the *filter* argument will show a **`DeprecationWarning`**. In Python 3.14, the default will switch to `'data'`. (Contributed by Petr Viktorin in **PEP 706** [https://peps.python.org/pep-0706/].)

What’s New In Python 3.8

Editor:: Raymond Hettinger

This article explains the new features in Python 3.8, compared to 3.7. Python 3.8 was released on October 14, 2019. For full details, see the changelog.

Summary – Release highlights

New Features

Assignment expressions

There is new syntax `:=` that assigns values to variables as part of a larger expression. It is affectionately known as “the walrus operator” due to its resemblance to the eyes and tusks of a walrus [https://en.wikipedia.org/wiki/Walrus#/media/File:Pacific_Walrus_-_Bull_(8247646168).jpg].

In this example, the assignment expression helps avoid calling **`len()`** twice:

```
if (n := len(a)) > 10:
    print(f"List is too long ({n} elements, expected <= 10)")
```

A similar benefit arises during regular expression matching where match objects are needed twice, once to test whether a match occurred and another to extract a subgroup:

```
discount = 0.0
if (mo := re.search(r'(\d+)% discount', advertisement)):
    discount = float(mo.group(1)) / 100.0
```

The operator is also useful with while-loops that compute a value to test loop termination and then need that same value again in the body of the loop:

```
# Loop over fixed length blocks
while (block := f.read(256)) != '':
    process(block)
```

Another motivating use case arises in list comprehensions where a value computed in a filtering condition is also needed in the expression body:

```
[clean_name.title() for name in names
 if (clean_name := normalize('NFC', name)) in allowed_names]
```

Try to limit use of the walrus operator to clean cases that reduce complexity and improve readability.

See **PEP 572** [https://peps.python.org/pep-0572/] for a full description.

(Contributed by Emily Morehouse in bpo-35224 [https://bugs.python.org/issue?@action=redirect&bpo=35224].)

Positional-only parameters

There is a new function parameter syntax `/` to indicate that some function parameters must be specified positionally and cannot be used as keyword arguments. This is the same notation shown by `help()` for C functions annotated with Larry Hastings’ Argument Clinic [https://devguide.python.org/development-tools/clinic/] tool.

In the following example, parameters *a* and *b* are positional-only, while *c* or *d* can be positional or keyword, and *e* or *f* are required to be keywords:

```
def f(a, b, /, c, d, *, e, f):
    print(a, b, c, d, e, f)
```

The following is a valid call:

```
f(10, 20, 30, d=40, e=50, f=60)
```

However, these are invalid calls:

```
f(10, b=20, c=30, d=40, e=50, f=60)   # b cannot be a keyword
argument
f(10, 20, 30, 40, 50, f=60)           # e must be a keyword
argument
```

One use case for this notation is that it allows pure Python functions to fully emulate behaviors of existing C coded functions. For example, the built-in **`divmod()`** function does not accept keyword arguments:

```
def divmod(a, b, /):
    "Emulate the built in divmod() function"
    return (a // b, a % b)
```

Another use case is to preclude keyword arguments when the parameter name is not helpful. For example, the builtin **`len()`** function has the signature `len(obj, /)`. This precludes awkward calls such as:

```
len(obj='hello')  # The "obj" keyword argument impairs
readability
```

A further benefit of marking a parameter as positional-only is that it allows the parameter name to be changed in the future without risk of breaking client code. For example, in the **`statistics`** module, the parameter name *dist* may be changed in the future. This was made possible with the following function specification:

```
def quantiles(dist, /, *, n=4, method='exclusive')
    ...
```

Since the parameters to the left of `/` are not exposed as possible keywords, the parameters names remain available for use in `**kwargs`:

```
>>> def f(a, b, /, **kwargs):
...     print(a, b, kwargs)
...
>>> f(10, 20, a=1, b=2, c=3)         # a and b are used in two
ways
10 20 {'a': 1, 'b': 2, 'c': 3}
```

This greatly simplifies the implementation of functions and methods that need to accept arbitrary keyword arguments. For example, here is an excerpt from code in the **`collections`** module:

```
class Counter(dict):

    def __init__(self, iterable=None, /, **kwds):
        # Note "iterable" is a possible keyword argument
```

See **PEP 570** [https://peps.python.org/pep-0570/] for a full description.

(Contributed by Pablo Galindo in bpo-36540 [https://bugs.python.org/issue?@action=redirect&bpo=36540].)

Parallel filesystem cache for compiled bytecode files

The new **`PYTHONPYCACHEPREFIX`** setting (also available as **`-X`** `pycache_prefix`) configures the implicit bytecode cache to use a separate parallel filesystem tree, rather than the default `__pycache__` subdirectories within each source directory.

The location of the cache is reported in **`sys.pycache_prefix`** (**`None`** indicates the default location in `__pycache__` subdirectories).

(Contributed by Carl Meyer in bpo-33499 [https://bugs.python.org/issue?@action=redirect&bpo=33499].)

Debug build uses the same ABI as release build

Python now uses the same ABI whether it’s built in release or debug mode. On Unix, when Python is built in debug mode, it is now possible to load C extensions built in release mode and C extensions built using the stable ABI.

Release builds and debug builds are now ABI compatible: defining the `Py_DEBUG` macro no longer implies the `Py_TRACE_REFS` macro, which introduces the only ABI incompatibility. The `Py_TRACE_REFS` macro, which adds the **`sys.getobjects()`** function and the **`PYTHONDUMPREFS`** environment variable, can be set using the new **`./configure --with-trace-refs`** build option. (Contributed by Victor Stinner in bpo-36465 [https://bugs.python.org/issue?@action=redirect&bpo=36465].)

On Unix, C extensions are no longer linked to libpython except on Android and Cygwin. It is now possible for a statically linked Python to load a C extension built using a shared library Python. (Contributed by Victor Stinner in bpo-21536 [https://bugs.python.org/issue?@action=redirect&bpo=21536].)

On Unix, when Python is built in debug mode, import now also looks for C extensions compiled in release mode and for C extensions compiled with the stable ABI. (Contributed by Victor Stinner in bpo-36722 [https://bugs.python.org/issue?@action=redirect&bpo=36722].)

To embed Python into an application, a new `--embed` option must be passed to `python3-config --libs --embed` to get `-lpython3.8` (link the application to libpython). To support both 3.8 and older, try `python3-config --libs --embed` first and fallback to `python3-config --libs` (without `--embed`) if the previous command fails.

Add a pkg-config `python-3.8-embed` module to embed Python into an application: `pkg-config python-3.8-embed --libs` includes `-lpython3.8`. To support both 3.8 and older, try `pkg-config python-X.Y-embed --libs` first and fallback to `pkg-config python-X.Y --libs` (without `--embed`) if the previous command fails (replace `X.Y` with the Python version).

On the other hand, `pkg-config python3.8 --libs` no longer contains `-lpython3.8`. C extensions must not be linked to libpython (except on Android and Cygwin, whose cases are handled by the script); this change is backward incompatible on purpose. (Contributed by Victor Stinner in bpo-36721 [https://bugs.python.org/issue?@action=redirect&bpo=36721].)

f-strings support `=` for self-documenting expressions and debugging

Added an `=` specifier to f-strings. An f-string such as `f'{expr=}'` will expand to the text of the expression, an equal sign, then the representation of the evaluated expression. For example:

```
>>> user = 'eric_idle'
>>> member_since = date(1975, 7, 31)
>>> f'{user=} {member_since=}'
"user='eric_idle' member_since=datetime.date(1975, 7, 31)"
```

The usual f-string format specifiers allow more control over how the result of the expression is displayed:

```
>>> delta = date.today() - member_since
>>> f'{user=!s}  {delta.days=:,d}'
'user=eric_idle  delta.days=16,075'
```

The `=` specifier will display the whole expression so that calculations can be shown:

```
>>> print(f'{theta=}  {cos(radians(theta))=:.3f}')
theta=30  cos(radians(theta))=0.866
```

(Contributed by Eric V. Smith and Larry Hastings in bpo-36817 [https://bugs.python.org/issue?@action=redirect&bpo=36817].)

PEP 578: Python Runtime Audit Hooks

The PEP adds an Audit Hook and Verified Open Hook. Both are available from Python and native code, allowing applications and frameworks written in pure Python code to take advantage of extra notifications, while also allowing embedders or system administrators to deploy builds of Python where auditing is always enabled.

See **PEP 578** [https://peps.python.org/pep-0578/] for full details.

PEP 587: Python Initialization Configuration

The **PEP 587** [https://peps.python.org/pep-0587/] adds a new C API to configure the Python Initialization providing finer control on the whole configuration and better error reporting.

New structures:

- **`PyConfig`**
- **`PyPreConfig`**
- **`PyStatus`**
- **`PyWideStringList`**

New functions:

- **`PyConfig_Clear()`**
- **`PyConfig_InitIsolatedConfig()`**

- `PyConfig_InitPythonConfig()`
- `PyConfig_Read()`
- `PyConfig_SetArgv()`
- `PyConfig_SetBytesArgv()`
- `PyConfig_SetBytesString()`
- `PyConfig_SetString()`
- `PyPreConfig_InitIsolatedConfig()`
- `PyPreConfig_InitPythonConfig()`
- `PyStatus_Error()`
- `PyStatus_Exception()`
- `PyStatus_Exit()`
- `PyStatus_IsError()`
- `PyStatus_IsExit()`
- `PyStatus_NoMemory()`
- `PyStatus_Ok()`
- `PyWideStringList_Append()`
- `PyWideStringList_Insert()`
- `Py_BytesMain()`
- `Py_ExitStatusException()`
- `Py_InitializeFromConfig()`
- `Py_PreInitialize()`
- `Py_PreInitializeFromArgs()`
- `Py_PreInitializeFromBytesArgs()`
- `Py_RunMain()`

This PEP also adds `_PyRuntimeState.preconfig` (**`PyPreConfig`** type) and `PyInterpreterState.config` (**`PyConfig`** type) fields to these internal structures. `PyInterpreterState.config` becomes the new reference configuration, replacing global configuration variables and other private variables.

See Python Initialization Configuration for the documentation.

See **PEP 587** [https://peps.python.org/pep-0587/] for a full description.

(Contributed by Victor Stinner in bpo-36763 [https://bugs.python.org/issue?@action=redirect&bpo=36763].)

PEP 590: Vectorcall: a fast calling protocol for CPython

The Vectorcall Protocol is added to the Python/C API. It is meant to formalize existing optimizations which were already done for various classes. Any static type implementing a callable can use this protocol.

This is currently provisional. The aim is to make it fully public in Python 3.9.

See **PEP 590** [https://peps.python.org/pep-0590/] for a full description.

(Contributed by Jeroen Demeyer, Mark Shannon and Petr Viktorin in bpo-36974 [https://bugs.python.org/issue?@action=redirect&bpo=36974].)

Pickle protocol 5 with out-of-band data buffers

When `pickle` is used to transfer large data between Python processes in order to take advantage of multi-core or multi-machine processing, it is important to optimize the transfer by reducing memory copies, and possibly by applying custom techniques such as data-dependent compression.

The `pickle` protocol 5 introduces support for out-of-band buffers where **PEP 3118** [https://peps.python.org/pep-3118/]-compatible data can be transmitted separately from the main pickle stream, at the discretion of the communication layer.

See **PEP 574** [https://peps.python.org/pep-0574/] for a full description.

(Contributed by Antoine Pitrou in bpo-36785 [https://bugs.python.org/issue?@action=redirect&bpo=36785].)

Other Language Changes

- A **`continue`** statement was illegal in the **`finally`** clause due to a problem with the implementation. In Python 3.8 this restriction was lifted. (Contributed by Serhiy Storchaka in bpo-32489 [https://bugs.python.org/issue?@action=redirect&bpo=32489].)

- The **`bool`**, **`int`**, and **`fractions.Fraction`** types now have an **`as_integer_ratio()`** method like that found in **`float`** and **`decimal.Decimal`**. This minor API extension makes it possible to write `numerator, denominator = x.as_integer_ratio()` and have it work across multiple numeric types. (Contributed by Lisa Roach in bpo-33073 [https://bugs.python.org/issue?@action=redirect&bpo=33073] and Raymond Hettinger in bpo-37819 [https://bugs.python.org/issue?@action=redirect&bpo=37819].)

- Constructors of **`int`**, **`float`** and **`complex`** will now use the **`__index__()`** special method, if available and the corresponding method **`__int__()`**, **`__float__()`** or **`__complex__()`** is not available. (Contributed by Serhiy Storchaka in bpo-20092 [https://bugs.python.org/issue?@action=redirect&bpo=20092].)

- Added support of `\N{`*`name`*`}` escapes in **`regular expressions`**:

```
>>> notice = 'Copyright © 2019'
>>> copyright_year_pattern = re.compile(r'\N{copyright
sign}\s*(\d{4})')
```

```
>>> int(copyright_year_pattern.search(notice).group(1))
2019
```

(Contributed by Jonathan Eunice and Serhiy Storchaka in bpo-30688 [https://bugs.python.org/issue?@action=redirect&bpo=30688].)

- Dict and dictviews are now iterable in reversed insertion order using **`reversed()`**. (Contributed by Rémi Lapeyre in bpo-33462 [https://bugs.python.org/issue?@action=redirect&bpo=33462].)
- The syntax allowed for keyword names in function calls was further restricted. In particular, `f((keyword)=arg)` is no longer allowed. It was never intended to permit more than a bare name on the left-hand side of a keyword argument assignment term. (Contributed by Benjamin Peterson in bpo-34641 [https://bugs.python.org/issue?@action=redirect&bpo=34641].)
- Generalized iterable unpacking in **`yield`** and **`return`** statements no longer requires enclosing parentheses. This brings the *yield* and *return* syntax into better agreement with normal assignment syntax:

  ```
  >>> def parse(family):
  ...     lastname, *members = family.split()
  ...     return lastname.upper(), *members
  ...
  >>> parse('simpsons homer marge bart lisa maggie')
  ('SIMPSONS', 'homer', 'marge', 'bart', 'lisa', 'maggie')
  ```

  (Contributed by David Cuthbert and Jordan Chapman in bpo-32117 [https://bugs.python.org/issue?@action=redirect&bpo=32117].)
- When a comma is missed in code such as `[(10, 20) (30, 40)]`, the compiler displays a **`SyntaxWarning`** with a helpful

suggestion. This improves on just having a **`TypeError`** indicating that the first tuple was not callable. (Contributed by Serhiy Storchaka in bpo-15248 [https://bugs.python.org/issue?@action=redirect&bpo=15248].)

- Arithmetic operations between subclasses of **`datetime.date`** or **`datetime.datetime`** and **`datetime.timedelta`** objects now return an instance of the subclass, rather than the base class. This also affects the return type of operations whose implementation (directly or indirectly) uses **`datetime.timedelta`** arithmetic, such as **`astimezone()`**. (Contributed by Paul Ganssle in bpo-32417 [https://bugs.python.org/issue?@action=redirect&bpo=32417].)

- When the Python interpreter is interrupted by Ctrl-C (SIGINT) and the resulting **`KeyboardInterrupt`** exception is not caught, the Python process now exits via a SIGINT signal or with the correct exit code such that the calling process can detect that it died due to a Ctrl-C. Shells on POSIX and Windows use this to properly terminate scripts in interactive sessions. (Contributed by Google via Gregory P. Smith in bpo-1054041 [https://bugs.python.org/issue?@action=redirect&bpo=1054041].)

- Some advanced styles of programming require updating the **`types.CodeType`** object for an existing function. Since code objects are immutable, a new code object needs to be created, one that is modeled on the existing code object. With 19 parameters, this was somewhat tedious. Now, the new `replace()` method makes it possible to create a clone with a few altered parameters.

  Here’s an example that alters the **`statistics.mean()`** function to prevent the *data* parameter from being used as a keyword argument:

```
>>> from statistics import mean
>>> mean(data=[10, 20, 90])
40
>>> mean.__code__ =
mean.__code__.replace(co_posonlyargcount=1)
>>> mean(data=[10, 20, 90])
Traceback (most recent call last):
  ...
TypeError: mean() got some positional-only arguments passed
as keyword arguments: 'data'
```

(Contributed by Victor Stinner in bpo-37032 [https://bugs.python.org/issue?@action=redirect&bpo=37032].)

- For integers, the three-argument form of the **`pow()`** function now permits the exponent to be negative in the case where the base is relatively prime to the modulus. It then computes a modular inverse to the base when the exponent is `-1`, and a suitable power of that inverse for other negative exponents. For example, to compute the modular multiplicative inverse [https://en.wikipedia.org/wiki/Modular_multiplicative_inverse] of 38 modulo 137, write:

  ```
  >>> pow(38, -1, 137)
  119
  >>> 119 * 38 % 137
  1
  ```

  Modular inverses arise in the solution of linear Diophantine equations [https://en.wikipedia.org/wiki/Diophantine_equation]. For example, to find integer solutions for $4258x + 147y = 369$, first rewrite as $4258x \equiv 369 \pmod{147}$ then solve:

  ```
  >>> x = 369 * pow(4258, -1, 147) % 147
  >>> y = (4258 * x - 369) // -147
  >>> 4258 * x + 147 * y
  369
  ```

(Contributed by Mark Dickinson in bpo-36027 [https://bugs.python.org/issue?@action=redirect&bpo=36027].)

- Dict comprehensions have been synced-up with dict literals so that the key is computed first and the value second:

  ```
  >>> # Dict comprehension
  >>> cast = {input('role? '): input('actor? ') for i in
  range(2)}
  role? King Arthur
  actor? Chapman
  role? Black Knight
  actor? Cleese

  >>> # Dict literal
  >>> cast = {input('role? '): input('actor? ')}
  role? Sir Robin
  actor? Eric Idle
  ```

  The guaranteed execution order is helpful with assignment expressions because variables assigned in the key expression will be available in the value expression:

  ```
  >>> names = ['Martin von Löwis', 'Łukasz Langa', 'Walter
  Dörwald']
  >>> {(n := normalize('NFC', name)).casefold() : n for name
  in names}
  {'martin von löwis': 'Martin von Löwis',
   'łukasz langa': 'Łukasz Langa',
   'walter dörwald': 'Walter Dörwald'}
  ```

  (Contributed by Jörn Heissler in bpo-35224 [https://bugs.python.org/issue?@action=redirect&bpo=35224].)

- The `object.__reduce__()` method can now return a tuple from two to six elements long. Formerly, five was the limit. The new, optional sixth element is a callable with a `(obj, state)` signature. This allows the direct control over the state-updating behavior of a specific object. If not *None*, this callable will have priority over the object's `__setstate__()` method. (Contributed by Pierre Glaser and Olivier Grisel in bpo-35900 [https://bugs.python.org/issue?@action=redirect&bpo=35900].)

New Modules

- The new `importlib.metadata` module provides (provisional) support for reading metadata from third-party packages. For example, it can extract an installed package's version number, list of entry points, and more:

```
>>> # Note following example requires that the popular
"requests"
>>> # package has been installed.
>>>
>>> from importlib.metadata import version, requires, files
>>> version('requests')
'2.22.0'
>>> list(requires('requests'))
['chardet (<3.1.0,>=3.0.2)']
>>> list(files('requests'))[:5]
[PackagePath('requests-2.22.0.dist-info/INSTALLER'),
 PackagePath('requests-2.22.0.dist-info/LICENSE'),
 PackagePath('requests-2.22.0.dist-info/METADATA'),
 PackagePath('requests-2.22.0.dist-info/RECORD'),
 PackagePath('requests-2.22.0.dist-info/WHEEL')]
```

  (Contributed by Barry Warsaw and Jason R. Coombs in bpo-34632 [https://bugs.python.org/issue?@action=redirect&bpo=34632].)

Improved Modules

ast

AST nodes now have `end_lineno` and `end_col_offset` attributes, which give the precise location of the end of the node. (This only applies to nodes that have `lineno` and `col_offset` attributes.)

New function **`ast.get_source_segment()`** returns the source code for a specific AST node.

(Contributed by Ivan Levkivskyi in bpo-33416 [https://bugs.python.org/issue?@action=redirect&bpo=33416].)

The **`ast.parse()`** function has some new flags:

- `type_comments=True` causes it to return the text of **PEP 484** [https://peps.python.org/pep-0484/] and **PEP 526** [https://peps.python.org/pep-0526/] type comments associated with certain AST nodes;
- `mode='func_type'` can be used to parse **PEP 484** [https://peps.python.org/pep-0484/] “signature type comments” (returned for function definition AST nodes);
- `feature_version=(3, N)` allows specifying an earlier Python 3 version. For example, `feature_version=(3, 4)` will treat **`async`** and **`await`** as non-reserved words.

(Contributed by Guido van Rossum in bpo-35766 [https://bugs.python.org/issue?@action=redirect&bpo=35766].)

asyncio

`asyncio.run()` has graduated from the provisional to stable API. This function can be used to execute a coroutine and return the result while automatically managing the event loop. For example:

```
import asyncio

async def main():
    await asyncio.sleep(0)
    return 42

asyncio.run(main())
```

This is *roughly* equivalent to:

```
import asyncio

async def main():
    await asyncio.sleep(0)
    return 42

loop = asyncio.new_event_loop()
asyncio.set_event_loop(loop)
try:
    loop.run_until_complete(main())
finally:
    asyncio.set_event_loop(None)
    loop.close()
```

The actual implementation is significantly more complex. Thus, **`asyncio.run()`** should be the preferred way of running asyncio programs.

(Contributed by Yury Selivanov in bpo-32314 [https://bugs.python.org/issue?@action=redirect&bpo=32314].)

Running `python -m asyncio` launches a natively async REPL. This allows rapid experimentation with code that has a top-level **`await`**. There is no longer a need to directly call `asyncio.run()` which would spawn a new event loop on every invocation:

```
$ python -m asyncio
asyncio REPL 3.8.0
Use "await" directly instead of "asyncio.run()".
Type "help", "copyright", "credits" or "license" for more
information.
>>> import asyncio
>>> await asyncio.sleep(10, result='hello')
hello
```

(Contributed by Yury Selivanov in bpo-37028 [https://bugs.python.org/issue?@action=redirect&bpo=37028].)

The exception **`asyncio.CancelledError`** now inherits from **`BaseException`** rather than **`Exception`** and no longer inherits from **`concurrent.futures.CancelledError`**. (Contributed by Yury Selivanov in bpo-32528 [https://bugs.python.org/issue?@action=redirect&bpo=32528].)

On Windows, the default event loop is now **`ProactorEventLoop`**. (Contributed by Victor Stinner in bpo-34687 [https://bugs.python.org/issue?@action=redirect&bpo=34687].)

`ProactorEventLoop` now also supports UDP. (Contributed by Adam Meily and Andrew Svetlov in bpo-29883 [https://bugs.python.org/issue?@action=redirect&bpo=29883].)

`ProactorEventLoop` can now be interrupted by **`KeyboardInterrupt`** ("CTRL+C"). (Contributed by Vladimir Matveev in bpo-23057 [https://bugs.python.org/issue?@action=redirect&bpo=23057].)

Added **`asyncio.Task.get_coro()`** for getting the wrapped coroutine within an **`asyncio.Task`**. (Contributed by Alex Grönholm in bpo-36999 [https://bugs.python.org/issue?@action=redirect&bpo=36999].)

Asyncio tasks can now be named, either by passing the `name` keyword argument to **`asyncio.create_task()`** or the **`create_task()`** event loop method, or by calling the **`set_name()`** method on the task object. The task name is visible in the `repr()` output of **`asyncio.Task`** and can also be retrieved using the **`get_name()`** method. (Contributed by Alex Grönholm in bpo-34270 [https://bugs.python.org/issue?@action=redirect&bpo=34270].)

Added support for Happy Eyeballs [https://en.wikipedia.org/wiki/Happy_Eyeballs] to **`asyncio.loop.create_connection()`**. To specify the behavior, two new parameters have been added: *happy_eyeballs_delay* and *interleave*. The Happy Eyeballs algorithm improves responsiveness in applications that support IPv4 and IPv6 by attempting to simultaneously connect using both. (Contributed by twisteroid ambassador in bpo-33530 [https://bugs.python.org/issue?@action=redirect&bpo=33530].)

builtins

The **`compile()`** built-in has been improved to accept the `ast.PyCF_ALLOW_TOP_LEVEL_AWAIT` flag. With this new flag passed, **`compile()`** will allow top-level `await`, `async for` and `async with` constructs that are usually considered invalid syntax. Asynchronous code object marked with the `CO_COROUTINE` flag may then be returned. (Contributed by Matthias Bussonnier in bpo-34616 [https://bugs.python.org/issue?@action=redirect&bpo=34616])

collections

The **`_asdict()`** method for **`collections.namedtuple()`** now returns a **`dict`** instead of a **`collections.OrderedDict`**. This works because regular dicts have guaranteed ordering since Python 3.7. If the extra features of **`OrderedDict`** are required, the suggested remediation is to cast the result to the desired type: `OrderedDict(nt._asdict())`. (Contributed by Raymond Hettinger in bpo-35864 [https://bugs.python.org/issue?@action=redirect&bpo=35864].)

cProfile

The **`cProfile.Profile`** class can now be used as a context manager. Profile a block of code by running:

```
import cProfile

with cProfile.Profile() as profiler:
      # code to be profiled
      ...
```

(Contributed by Scott Sanderson in bpo-29235 [https://bugs.python.org/issue?@action=redirect&bpo=29235].)

csv

The **`csv.DictReader`** now returns instances of **`dict`** instead of a **`collections.OrderedDict`**. The tool is now faster and uses less memory while still preserving the field order. (Contributed by Michael Selik in bpo-34003 [https://bugs.python.org/issue?@action=redirect&bpo=34003].)

curses

Added a new variable holding structured version information for the underlying ncurses library: **`ncurses_version`**. (Contributed by Serhiy Storchaka in bpo-31680 [https://bugs.python.org/issue?@action=redirect&bpo=31680].)

ctypes

On Windows, **`CDLL`** and subclasses now accept a *winmode* parameter to specify flags for the underlying `LoadLibraryEx` call. The default flags are set to only load DLL dependencies from trusted locations, including the path where the DLL is stored (if a full or partial path is used to load the initial DLL) and paths added by **`add_dll_directory()`**. (Contributed by Steve Dower in bpo-36085 [https://bugs.python.org/issue?@action=redirect&bpo=36085].)

datetime

Added new alternate constructors `datetime.date.fromisocalendar()` and `datetime.datetime.fromisocalendar()`, which construct `date` and `datetime` objects respectively from ISO year, week number, and weekday; these are the inverse of each class's `isocalendar` method. (Contributed by Paul Ganssle in bpo-36004 [https://bugs.python.org/issue?@action=redirect&bpo=36004].)

functools

`functools.lru_cache()` can now be used as a straight decorator rather than as a function returning a decorator. So both of these are now supported:

```
@lru_cache
def f(x):
    ...

@lru_cache(maxsize=256)
def f(x):
    ...
```

(Contributed by Raymond Hettinger in bpo-36772 [https://bugs.python.org/issue?@action=redirect&bpo=36772].)

Added a new `functools.cached_property()` decorator, for computed properties cached for the life of the instance.

```
import functools
import statistics

class Dataset:
   def __init__(self, sequence_of_numbers):
      self.data = sequence_of_numbers

   @functools.cached_property
   def variance(self):
      return statistics.variance(self.data)
```

(Contributed by Carl Meyer in bpo-21145 [https://bugs.python.org/issue?@action=redirect&bpo=21145])

Added a new **`functools.singledispatchmethod()`** decorator that converts methods into generic functions using single dispatch:

```
from functools import singledispatchmethod
from contextlib import suppress

class TaskManager:

    def __init__(self, tasks):
        self.tasks = list(tasks)

    @singledispatchmethod
    def discard(self, value):
        with suppress(ValueError):
            self.tasks.remove(value)

    @discard.register(list)
    def _(self, tasks):
        targets = set(tasks)
        self.tasks = [x for x in self.tasks if x not in
targets]
```

(Contributed by Ethan Smith in bpo-32380 [https://bugs.python.org/issue?@action=redirect&bpo=32380])

gc

`get_objects()` can now receive an optional *generation* parameter indicating a generation to get objects from. (Contributed by Pablo Galindo in bpo-36016 [https://bugs.python.org/issue?@action=redirect&bpo=36016].)

gettext

Added `pgettext()` and its variants. (Contributed by Franz Glasner, Éric Araujo, and Cheryl Sabella in bpo-2504 [https://bugs.python.org/issue?@action=redirect&bpo=2504].)

gzip

Added the *mtime* parameter to `gzip.compress()` for reproducible output. (Contributed by Guo Ci Teo in bpo-34898 [https://bugs.python.org/issue?@action=redirect&bpo=34898].)

A `BadGzipFile` exception is now raised instead of `OSError` for certain types of invalid or corrupt gzip files. (Contributed by Filip Gruszczyński, Michele Orrù, and Zackery Spytz in bpo-6584 [https://bugs.python.org/issue?@action=redirect&bpo=6584].)

IDLE and idlelib

Output over N lines (50 by default) is squeezed down to a button. N can be changed in the PyShell section of the General page of the Settings dialog. Fewer, but possibly extra long, lines can be squeezed by right clicking on the output. Squeezed output can be expanded in place by double-clicking the button or into the clipboard or a separate window by right-clicking the button. (Contributed by Tal Einat in bpo-1529353 [https://bugs.python.org/issue?@action=redirect&bpo=1529353].)

Add “Run Customized” to the Run menu to run a module with customized settings. Any command line arguments entered are added to sys.argv. They also re-appear in the box for the next customized run. One can also suppress the normal Shell main module restart. (Contributed by Cheryl Sabella, Terry Jan Reedy, and others in bpo-5680 [https://bugs.python.org/issue?@action=redirect&bpo=5680] and bpo-37627 [https://bugs.python.org/issue?@action=redirect&bpo=37627].)

Added optional line numbers for IDLE editor windows. Windows open without line numbers unless set otherwise in the General tab of the configuration dialog. Line numbers for an existing window are shown and hidden in the Options menu. (Contributed by Tal Einat and Saimadhav Heblikar in bpo-17535 [https://bugs.python.org/issue?@action=redirect&bpo=17535].)

OS native encoding is now used for converting between Python strings and Tcl objects. This allows IDLE to work with emoji and other non-BMP characters. These characters can be displayed or copied and pasted to or from the clipboard. Converting strings from Tcl to Python and back now never fails. (Many people worked on this for eight years but the problem was finally solved by Serhiy Storchaka in bpo-13153 [https://bugs.python.org/issue?@action=redirect&bpo=13153].)

New in 3.8.1:

Add option to toggle cursor blink off. (Contributed by Zackery Spytz in bpo-4603 [https://bugs.python.org/issue?@action=redirect&bpo=4603].)

Escape key now closes IDLE completion windows. (Contributed by Johnny Najera in bpo-38944 [https://bugs.python.org/issue?@action=redirect&bpo=38944].)

The changes above have been backported to 3.7 maintenance releases.

Add keywords to module name completion list. (Contributed by Terry J. Reedy in bpo-37765 [https://bugs.python.org/issue?@action=redirect&bpo=37765].)

inspect

The **`inspect.getdoc()`** function can now find docstrings for `__slots__` if that attribute is a **`dict`** where the values are docstrings. This provides documentation options similar to what we already have for **`property()`**, **`classmethod()`**, and **`staticmethod()`**:

```
class AudioClip:
    __slots__ = {'bit_rate': 'expressed in kilohertz to one
decimal place',
                 'duration': 'in seconds, rounded up to an
integer'}
    def __init__(self, bit_rate, duration):
        self.bit_rate = round(bit_rate / 1000.0, 1)
        self.duration = ceil(duration)
```

(Contributed by Raymond Hettinger in bpo-36326 [https://bugs.python.org/issue?@action=redirect&bpo=36326].)

io

In development mode (**`-X`** `env`) and in debug build, the **`io.IOBase`** finalizer now logs the exception if the `close()` method fails. The exception is ignored silently by default in release build. (Contributed by Victor Stinner in bpo-18748 [https://bugs.python.org/issue?@action=redirect&bpo=18748].)

itertools

The **`itertools.accumulate()`** function added an option *initial* keyword argument to specify an initial value:

```
>>> from itertools import accumulate
>>> list(accumulate([10, 5, 30, 15], initial=1000))
[1000, 1010, 1015, 1045, 1060]
```

(Contributed by Lisa Roach in bpo-34659 [https://bugs.python.org/issue?@action=redirect&bpo=34659].)

json.tool

Add option `--json-lines` to parse every input line as a separate JSON object. (Contributed by Weipeng Hong in bpo-31553 [https://bugs.python.org/issue?@action=redirect&bpo=31553].)

logging

Added a *force* keyword argument to **`logging.basicConfig()`**. When set to true, any existing handlers attached to the root logger are removed and closed before carrying out the configuration specified by the other arguments.

This solves a long-standing problem. Once a logger or *basicConfig()* had been called, subsequent calls to *basicConfig()* were silently ignored. This made it difficult to update, experiment with, or teach the various logging configuration options using the interactive prompt or a Jupyter notebook.

(Suggested by Raymond Hettinger, implemented by Donghee Na, and reviewed by Vinay Sajip in bpo-33897 [https://bugs.python.org/issue?@action=redirect&bpo=33897].)

math

Added new function **`math.dist()`** for computing Euclidean distance between two points. (Contributed by Raymond Hettinger in bpo-33089 [https://bugs.python.org/issue?@action=redirect&bpo=33089].)

Expanded the **`math.hypot()`** function to handle multiple dimensions. Formerly, it only supported the 2-D case. (Contributed by Raymond Hettinger in bpo-33089 [https://bugs.python.org/issue?@action=redirect&bpo=33089].)

Added new function, **`math.prod()`**, as analogous function to **`sum()`** that returns the product of a ‘start’ value (default: 1) times an iterable of numbers:

```
>>> prior = 0.8
>>> likelihoods = [0.625, 0.84, 0.30]
>>> math.prod(likelihoods, start=prior)
0.126
```

(Contributed by Pablo Galindo in bpo-35606 [https://bugs.python.org/issue?@action=redirect&bpo=35606].)

Added two new combinatoric functions **`math.perm()`** and **`math.comb()`**:

```
>>> math.perm(10, 3)    # Permutations of 10 things taken 3 at
a time
720
>>> math.comb(10, 3)    # Combinations of 10 things taken 3 at
a time
120
```

(Contributed by Yash Aggarwal, Keller Fuchs, Serhiy Storchaka, and Raymond Hettinger in bpo-37128 [https://bugs.python.org/issue?@action=redirect&bpo=37128], bpo-37178 [https://bugs.python.org/issue?@action=redirect&bpo=37178], and bpo-35431 [https://bugs.python.org/issue?@action=redirect&bpo=35431].)

Added a new function **`math.isqrt()`** for computing accurate integer square roots without conversion to floating point. The new function supports arbitrarily large integers. It is faster than `floor(sqrt(n))` but slower than **`math.sqrt()`**:

```
>>> r = 650320427
>>> s = r ** 2
>>> isqrt(s - 1)         # correct
650320426
>>> floor(sqrt(s - 1))   # incorrect
650320427
```

(Contributed by Mark Dickinson in bpo-36887 [https://bugs.python.org/issue?@action=redirect&bpo=36887].)

The function **`math.factorial()`** no longer accepts arguments that are not int-like. (Contributed by Pablo Galindo in bpo-33083 [https://bugs.python.org/issue?@action=redirect&bpo=33083].)

mmap

The **`mmap.mmap`** class now has an **`madvise()`** method to access the `madvise()` system call. (Contributed by Zackery Spytz in bpo-32941 [https://bugs.python.org/issue?@action=redirect&bpo=32941].)

multiprocessing

Added new **`multiprocessing.shared_memory`** module. (Contributed by Davin Potts in bpo-35813 [https://bugs.python.org/issue?@action=redirect&bpo=35813].)

On macOS, the *spawn* start method is now used by default. (Contributed by Victor Stinner in bpo-33725 [https://bugs.python.org/issue?@action=redirect&bpo=33725].)

os

Added new function **`add_dll_directory()`** on Windows for providing additional search paths for native dependencies when importing extension modules or loading DLLs using **`ctypes`**. (Contributed by Steve Dower in bpo-36085 [https://bugs.python.org/issue?@action=redirect&bpo=36085].)

A new **`os.memfd_create()`** function was added to wrap the `memfd_create()` syscall. (Contributed by Zackery Spytz and Christian Heimes in bpo-26836 [https://bugs.python.org/issue?@action=redirect&bpo=26836].)

On Windows, much of the manual logic for handling reparse points (including symlinks and directory junctions) has been delegated to the operating system. Specifically, **`os.stat()`**

will now traverse anything supported by the operating system, while `os.lstat()` will only open reparse points that identify as “name surrogates” while others are opened as for `os.stat()`. In all cases, `os.stat_result.st_mode` will only have `S_IFLNK` set for symbolic links and not other kinds of reparse points. To identify other kinds of reparse point, check the new `os.stat_result.st_reparse_tag` attribute.

On Windows, `os.readlink()` is now able to read directory junctions. Note that `islink()` will return `False` for directory junctions, and so code that checks `islink` first will continue to treat junctions as directories, while code that handles errors from `os.readlink()` may now treat junctions as links.

(Contributed by Steve Dower in bpo-37834 [https://bugs.python.org/issue?@action=redirect&bpo=37834].)

os.path

`os.path` functions that return a boolean result like `exists()`, `lexists()`, `isdir()`, `isfile()`, `islink()`, and `ismount()` now return `False` instead of raising `ValueError` or its subclasses `UnicodeEncodeError` and `UnicodeDecodeError` for paths that contain characters or bytes unrepresentable at the OS level. (Contributed by Serhiy Storchaka in bpo-33721 [https://bugs.python.org/issue?@action=redirect&bpo=33721].)

`expanduser()` on Windows now prefers the **`USERPROFILE`** environment variable and does not use **`HOME`**, which is not normally set for regular user accounts. (Contributed by Anthony Sottile in bpo-36264 [https://bugs.python.org/issue?@action=redirect&bpo=36264].)

`isdir()` on Windows no longer returns `True` for a link to a non-existent directory.

`realpath()` on Windows now resolves reparse points, including symlinks and directory junctions.

(Contributed by Steve Dower in bpo-37834 [https://bugs.python.org/issue?@action=redirect&bpo=37834].)

pathlib

`pathlib.Path` methods that return a boolean result like **`exists()`**, **`is_dir()`**, **`is_file()`**, **`is_mount()`**, **`is_symlink()`**, **`is_block_device()`**, **`is_char_device()`**, **`is_fifo()`**, **`is_socket()`** now return `False` instead of raising **`ValueError`** or its subclass **`UnicodeEncodeError`** for paths that contain characters unrepresentable at the OS level. (Contributed by Serhiy Storchaka in bpo-33721 [https://bugs.python.org/issue?@action=redirect&bpo=33721].)

Added **`pathlib.Path.link_to()`** which creates a hard link pointing to a path. (Contributed by Joannah Nanjekye in bpo-26978 [https://bugs.python.org/issue?@action=redirect&bpo=26978]) Note that `link_to` was deprecated in 3.10 and removed in 3.12 in favor of a `hardlink_to` method added in 3.10 which matches the semantics of the existing `symlink_to` method.

pickle

`pickle` extensions subclassing the C-optimized **`Pickler`** can now override the pickling logic of functions and classes by defining the special **`reducer_override()`** method. (Contributed by Pierre Glaser and Olivier Grisel in bpo-35900 [https://bugs.python.org/issue?@action=redirect&bpo=35900].)

plistlib

Added new **`plistlib.UID`** and enabled support for reading and writing NSKeyedArchiver-encoded binary plists. (Contributed by Jon Janzen in bpo-26707 [https://bugs.python.org/issue?@action=redirect&bpo=26707].)

pprint

The **`pprint`** module added a *sort_dicts* parameter to several functions. By default, those functions continue to sort dictionaries before rendering or printing. However, if *sort_dicts* is set to false, the dictionaries retain the order that keys were inserted. This can be useful for comparison to JSON inputs during debugging.

In addition, there is a convenience new function, **`pprint.pp()`** that is like **`pprint.pprint()`** but with *sort_dicts* defaulting to `False`:

```
>>> from pprint import pprint, pp
>>> d = dict(source='input.txt', operation='filter',
destination='output.txt')
>>> pp(d, width=40)                  # Original order
{'source': 'input.txt',
 'operation': 'filter',
 'destination': 'output.txt'}
>>> pprint(d, width=40)              # Keys sorted
alphabetically
{'destination': 'output.txt',
 'operation': 'filter',
 'source': 'input.txt'}
```

(Contributed by Rémi Lapeyre in bpo-30670 [https://bugs.python.org/issue?@action=redirect&bpo=30670].)

py_compile

`py_compile.compile()` now supports silent mode. (Contributed by Joannah Nanjekye in bpo-22640 [https://bugs.python.org/issue?@action=redirect&bpo=22640].)

shlex

The new `shlex.join()` function acts as the inverse of `shlex.split()`. (Contributed by Bo Bayles in bpo-32102 [https://bugs.python.org/issue?@action=redirect&bpo=32102].)

shutil

`shutil.copytree()` now accepts a new `dirs_exist_ok` keyword argument. (Contributed by Josh Bronson in bpo-20849 [https://bugs.python.org/issue?@action=redirect&bpo=20849].)

`shutil.make_archive()` now defaults to the modern pax (POSIX.1-2001) format for new archives to improve portability and standards conformance, inherited from the corresponding change to the `tarfile` module. (Contributed by C.A.M. Gerlach in bpo-30661 [https://bugs.python.org/issue?@action=redirect&bpo=30661].)

`shutil.rmtree()` on Windows now removes directory junctions without recursively removing their contents first. (Contributed by Steve Dower in bpo-37834 [https://bugs.python.org/issue?@action=redirect&bpo=37834].)

socket

Added `create_server()` and `has_dualstack_ipv6()` convenience functions to automate the necessary tasks usually involved when creating a server socket, including accepting both IPv4 and IPv6 connections on the same socket. (Contributed by Giampaolo Rodolà in bpo-17561 [https://bugs.python.org/issue?@action=redirect&bpo=17561].)

The `socket.if_nameindex()`, `socket.if_nametoindex()`, and `socket.if_indextoname()` functions have been implemented on

Windows. (Contributed by Zackery Spytz in bpo-37007 [https://bugs.python.org/issue?@action=redirect&bpo=37007].)

ssl

Added `post_handshake_auth` to enable and `verify_client_post_handshake()` to initiate TLS 1.3 post-handshake authentication. (Contributed by Christian Heimes in bpo-34670 [https://bugs.python.org/issue?@action=redirect&bpo=34670].)

statistics

Added `statistics.fmean()` as a faster, floating-point variant of `statistics.mean()`. (Contributed by Raymond Hettinger and Steven D’Aprano in bpo-35904 [https://bugs.python.org/issue?@action=redirect&bpo=35904].)

Added `statistics.geometric_mean()` (Contributed by Raymond Hettinger in bpo-27181 [https://bugs.python.org/issue?@action=redirect&bpo=27181].)

Added `statistics.multimode()` that returns a list of the most common values. (Contributed by Raymond Hettinger in bpo-35892 [https://bugs.python.org/issue?@action=redirect&bpo=35892].)

Added `statistics.quantiles()` that divides data or a distribution in to equiprobable intervals (e.g. quartiles, deciles, or percentiles). (Contributed by Raymond Hettinger in bpo-36546 [https://bugs.python.org/issue?@action=redirect&bpo=36546].)

Added `statistics.NormalDist`, a tool for creating and manipulating normal distributions of a random variable. (Contributed by Raymond Hettinger in bpo-36018 [https://bugs.python.org/issue?@action=redirect&bpo=36018].)

```
>>> temperature_feb = NormalDist.from_samples([4, 12, -3, 2, 7,
14])
>>> temperature_feb.mean
6.0
>>> temperature_feb.stdev
6.356099432828281

>>> temperature_feb.cdf(3)            # Chance of being under 3
degrees
0.3184678262814532
>>> # Relative chance of being 7 degrees versus 10 degrees
>>> temperature_feb.pdf(7) / temperature_feb.pdf(10)
1.2039930378537762

>>> el_niño = NormalDist(4, 2.5)
>>> temperature_feb += el_niño        # Add in a climate effect
>>> temperature_feb
NormalDist(mu=10.0, sigma=6.830080526611674)

>>> temperature_feb * (9/5) + 32      # Convert to Fahrenheit
NormalDist(mu=50.0, sigma=12.294144947901014)
>>> temperature_feb.samples(3)        # Generate random samples
[7.672102882379219, 12.000027119750287, 4.647488369766392]
```

sys

Add new `sys.unraisablehook()` function which can be overridden to control how “unraisable exceptions” are handled. It is called when an exception has occurred but there is no way for Python to handle it. For example, when a destructor raises an exception or during garbage collection (`gc.collect()`). (Contributed by Victor Stinner in bpo-36829 [https://bugs.python.org/issue?@action=redirect&bpo=36829].)

tarfile

The `tarfile` module now defaults to the modern pax (POSIX.1-2001) format for new archives, instead of the previous GNU-specific one. This improves cross-platform portability with a consistent encoding (UTF-8) in a standardized and extensible format, and offers several other benefits. (Contributed by C.A.M. Gerlach in bpo-36268 [https://bugs.python.org/issue?@action=redirect&bpo=36268].)

threading

Add a new `threading.excepthook()` function which handles uncaught `threading.Thread.run()` exception. It can be overridden to control how uncaught `threading.Thread.run()` exceptions are handled. (Contributed by Victor Stinner in bpo-1230540 [https://bugs.python.org/issue?@action=redirect&bpo=1230540].)

Add a new `threading.get_native_id()` function and a `native_id` attribute to the `threading.Thread` class. These return the native integral Thread ID of the current thread assigned by the kernel. This feature is only available on certain platforms, see `get_native_id` for more information. (Contributed by Jake Tesler in bpo-36084 [https://bugs.python.org/issue?@action=redirect&bpo=36084].)

tokenize

The `tokenize` module now implicitly emits a `NEWLINE` token when provided with input that does not have a trailing new line. This behavior now matches what the C tokenizer does internally. (Contributed by Ammar Askar in bpo-33899 [https://bugs.python.org/issue?@action=redirect&bpo=33899].)

tkinter

Added methods `selection_from()`, `selection_present()`, `selection_range()` and `selection_to()` in the `tkinter.Spinbox` class. (Contributed by Juliette Monsel in bpo-34829 [https://bugs.python.org/issue?@action=redirect&bpo=34829].)

Added method **`moveto()`** in the **`tkinter.Canvas`** class. (Contributed by Juliette Monsel in bpo-23831 [https://bugs.python.org/issue?@action=redirect&bpo=23831].)

The **`tkinter.PhotoImage`** class now has **`transparency_get()`** and **`transparency_set()`** methods. (Contributed by Zackery Spytz in bpo-25451 [https://bugs.python.org/issue?@action=redirect&bpo=25451].)

time

Added new clock **`CLOCK_UPTIME_RAW`** for macOS 10.12. (Contributed by Joannah Nanjekye in bpo-35702 [https://bugs.python.org/issue?@action=redirect&bpo=35702].)

typing

The **`typing`** module incorporates several new features:

- A dictionary type with per-key types. See **PEP 589** [https://peps.python.org/pep-0589/] and **`typing.TypedDict`**. TypedDict uses only string keys. By default, every key is required to be present. Specify “total=False” to allow keys to be optional:

  ```
  class Location(TypedDict, total=False):
      lat_long: tuple
      grid_square: str
      xy_coordinate: tuple
  ```

- Literal types. See **PEP 586** [https://peps.python.org/pep-0586/] and **`typing.Literal`**. Literal types indicate that a parameter or return value is constrained to one or more specific literal values:

  ```
  def get_status(port: int) -> Literal['connected',
  'disconnected']:
      ...
  ```

- “Final” variables, functions, methods and classes. See **PEP 591** [https://peps.python.org/pep-0591/], **`typing.Final`** and **`typing.final()`**. The final qualifier instructs a static type checker to restrict subclassing, overriding, or reassignment:

  ```
  pi: Final[float] = 3.1415926536
  ```

- Protocol definitions. See **PEP 544** [https://peps.python.org/pep-0544/], **`typing.Protocol`** and **`typing.runtime_checkable()`**. Simple ABCs like **`typing.SupportsInt`** are now `Protocol` subclasses.

- New protocol class **`typing.SupportsIndex`**.

- New functions **`typing.get_origin()`** and **`typing.get_args()`**.

unicodedata

The **`unicodedata`** module has been upgraded to use the Unicode 12.1.0 [https://blog.unicode.org/2019/05/unicode-12-1-en.html] release.

New function **`is_normalized()`** can be used to verify a string is in a specific normal form, often much faster than by actually normalizing the string. (Contributed by Max Belanger, David Euresti, and Greg Price in bpo-32285 [https://bugs.python.org/issue?@action=redirect&bpo=32285] and bpo-37966 [https://bugs.python.org/issue?@action=redirect&bpo=37966]).

unittest

Added **`AsyncMock`** to support an asynchronous version of **`Mock`**. Appropriate new assert functions for testing have been added as well. (Contributed by Lisa Roach in bpo-26467 [https://bugs.python.org/issue?@action=redirect&bpo=26467]).

Added `addModuleCleanup()` and `addClassCleanup()` to unittest to support cleanups for `setUpModule()` and `setUpClass()`. (Contributed by Lisa Roach in bpo-24412 [https://bugs.python.org/issue?@action=redirect&bpo=24412].)

Several mock assert functions now also print a list of actual calls upon failure. (Contributed by Petter Strandmark in bpo-35047 [https://bugs.python.org/issue?@action=redirect&bpo=35047].)

`unittest` module gained support for coroutines to be used as test cases with `unittest.IsolatedAsyncioTestCase`. (Contributed by Andrew Svetlov in bpo-32972 [https://bugs.python.org/issue?@action=redirect&bpo=32972].)

Example:

```
import unittest


class TestRequest(unittest.IsolatedAsyncioTestCase):

    async def asyncSetUp(self):
        self.connection = await AsyncConnection()

    async def test_get(self):
        response = await
self.connection.get("https://example.com")
        self.assertEqual(response.status_code, 200)

    async def asyncTearDown(self):
        await self.connection.close()


if __name__ == "__main__":
    unittest.main()
```

venv

`venv` now includes an `Activate.ps1` script on all platforms for activating virtual environments under PowerShell Core 6.1. (Contributed by Brett Cannon in bpo-32718 [https://bugs.python.org/issue?@action=redirect&bpo=32718].)

weakref

The proxy objects returned by **`weakref.proxy()`** now support the matrix multiplication operators `@` and `@=` in addition to the other numeric operators. (Contributed by Mark Dickinson in bpo-36669 [https://bugs.python.org/issue?@action=redirect&bpo=36669].)

xml

As mitigation against DTD and external entity retrieval, the **`xml.dom.minidom`** and **`xml.sax`** modules no longer process external entities by default. (Contributed by Christian Heimes in bpo-17239 [https://bugs.python.org/issue?@action=redirect&bpo=17239].)

The `.find*()` methods in the **`xml.etree.ElementTree`** module support wildcard searches like `{*}tag` which ignores the namespace and `{namespace}*` which returns all tags in the given namespace. (Contributed by Stefan Behnel in bpo-28238 [https://bugs.python.org/issue?@action=redirect&bpo=28238].)

The **`xml.etree.ElementTree`** module provides a new function **`canonicalize()`** that implements C14N 2.0. (Contributed by Stefan Behnel in bpo-13611 [https://bugs.python.org/issue?@action=redirect&bpo=13611].)

The target object of **`xml.etree.ElementTree.XMLParser`** can receive namespace declaration events through the new callback methods `start_ns()` and `end_ns()`. Additionally, the **`xml.etree.ElementTree.TreeBuilder`** target can be configured to process events about comments and processing instructions to include them in the generated tree. (Contributed by Stefan Behnel in bpo-36676 [https://bugs.python.org/issue?@action=redirect&bpo=36676] and bpo-36673 [https://bugs.python.org/issue?@action=redirect&bpo=36673].)

xmlrpc

`xmlrpc.client.ServerProxy` now supports an optional *headers* keyword argument for a sequence of HTTP headers to be sent with each request. Among other things, this makes it possible to upgrade from default basic authentication to faster session authentication. (Contributed by Cédric Krier in bpo-35153 [https://bugs.python.org/issue?@action=redirect&bpo=35153].)

Optimizations

- The `subprocess` module can now use the `os.posix_spawn()` function in some cases for better performance. Currently, it is only used on macOS and Linux (using glibc 2.24 or newer) if all these conditions are met:

    - *close_fds* is false;
    - *preexec_fn*, *pass_fds*, *cwd* and *start_new_session* parameters are not set;
    - the *executable* path contains a directory.

  (Contributed by Joannah Nanjekye and Victor Stinner in bpo-35537 [https://bugs.python.org/issue?@action=redirect&bpo=35537].)

- `shutil.copyfile()`, `shutil.copy()`, `shutil.copy2()`, `shutil.copytree()` and `shutil.move()` use platform-specific “fast-copy” syscalls on Linux and macOS in order to copy the file more efficiently. “fast-copy” means that the copying operation occurs within the kernel, avoiding the use of userspace buffers in Python as in “`outfd.write(infd.read())`”. On Windows **`shutil.copyfile()`** uses a bigger default buffer size (1 MiB instead of 16 KiB) and a **`memoryview()`**-based variant of **`shutil.copyfileobj()`** is used. The speedup for copying a 512 MiB file within the same partition is about +26% on Linux, +50% on macOS and +40% on Windows. Also, much less CPU cycles are consumed. See Platform-dependent efficient copy operations section. (Contributed by Giampaolo Rodolà in bpo-33671 [https://bugs.python.org/issue?@action=redirect&bpo=33671].)

- **`shutil.copytree()`** uses **`os.scandir()`** function and all copy functions depending from it use cached **`os.stat()`** values. The speedup for copying a directory with 8000 files is around +9% on Linux, +20% on Windows and +30% on a Windows SMB share. Also the number of **`os.stat()`** syscalls is reduced by 38% making **`shutil.copytree()`** especially faster on network filesystems. (Contributed by Giampaolo Rodolà in bpo-33695 [https://bugs.python.org/issue?@action=redirect&bpo=33695].)

- The default protocol in the **`pickle`** module is now Protocol 4, first introduced in Python 3.4. It offers better performance and smaller size compared to Protocol 3 available since Python 3.0.

- Removed one **`Py_ssize_t`** member from `PyGC_Head`. All GC tracked objects (e.g. tuple, list, dict) size is reduced 4 or 8 bytes. (Contributed by Inada Naoki in bpo-33597 [https://bugs.python.org/issue?@action=redirect&bpo=33597].)

- **`uuid.UUID`** now uses `__slots__` to reduce its memory footprint. (Contributed by Wouter Bolsterlee and Tal Einat in bpo-30977 [https://bugs.python.org/issue?@action=redirect&bpo=30977])

- Improved performance of `operator.itemgetter()` by 33%. Optimized argument handling and added a fast path for the common case of a single non-negative integer index into a tuple (which is the typical use case in the standard library). (Contributed by Raymond Hettinger in bpo-35664 [https://bugs.python.org/issue?@action=redirect&bpo=35664].)

- Sped-up field lookups in `collections.namedtuple()`. They are now more than two times faster, making them the fastest form of instance variable lookup in Python. (Contributed by Raymond Hettinger, Pablo Galindo, and Joe Jevnik, Serhiy Storchaka in bpo-32492 [https://bugs.python.org/issue?@action=redirect&bpo=32492].)

- The `list` constructor does not overallocate the internal item buffer if the input iterable has a known length (the input implements `__len__`). This makes the created list 12% smaller on average. (Contributed by Raymond Hettinger and Pablo Galindo in bpo-33234 [https://bugs.python.org/issue?@action=redirect&bpo=33234].)

- Doubled the speed of class variable writes. When a non-dunder attribute was updated, there was an unnecessary call to update slots. (Contributed by Stefan Behnel, Pablo Galindo Salgado, Raymond Hettinger, Neil Schemenauer, and Serhiy Storchaka in bpo-36012 [https://bugs.python.org/issue?@action=redirect&bpo=36012].)

- Reduced an overhead of converting arguments passed to many builtin functions and methods. This sped up calling some simple builtin functions and methods up to 20–50%. (Contributed by Serhiy Storchaka in bpo-23867 [https://bugs.python.org/issue?@action=redirect&bpo=23867], bpo-35582 [https://bugs.python.org/issue?@action=redirect&bpo=35582]

and bpo-36127 [https://bugs.python.org/issue?@action=redirect&bpo=36127].)

- `LOAD_GLOBAL` instruction now uses new “per opcode cache” mechanism. It is about 40% faster now. (Contributed by Yury Selivanov and Inada Naoki in bpo-26219 [https://bugs.python.org/issue?@action=redirect&bpo=26219].)

Build and C API Changes

- Default **`sys.abiflags`** became an empty string: the `m` flag for pymalloc became useless (builds with and without pymalloc are ABI compatible) and so has been removed. (Contributed by Victor Stinner in bpo-36707 [https://bugs.python.org/issue?@action=redirect&bpo=36707].)

  Example of changes:

  - Only `python3.8` program is installed, `python3.8m` program is gone.
  - Only `python3.8-config` script is installed, `python3.8m-config` script is gone.
  - The `m` flag has been removed from the suffix of dynamic library filenames: extension modules in the standard library as well as those produced and installed by third-party packages, like those downloaded from PyPI. On Linux, for example, the Python 3.7 suffix `.cpython-37m-x86_64-linux-gnu.so` became `.cpython-38-x86_64-linux-gnu.so` in Python 3.8.

- The header files have been reorganized to better separate the different kinds of APIs:

  - `Include/*.h` should be the portable public stable C API.

  - `Include/cpython/*.h` should be the unstable C API specific to CPython; public API, with some private API prefixed by `_Py` or `_PY`.
  - `Include/internal/*.h` is the private internal C API very specific to CPython. This API comes with no backward compatibility warranty and should not be used outside CPython. It is only exposed for very specific needs like debuggers and profiles which has to access to CPython internals without calling functions. This API is now installed by `make install`.

  (Contributed by Victor Stinner in bpo-35134 [https://bugs.python.org/issue?@action=redirect&bpo=35134] and bpo-35081 [https://bugs.python.org/issue?@action=redirect&bpo=35081], work initiated by Eric Snow in Python 3.7.)

- Some macros have been converted to static inline functions: parameter types and return type are well defined, they don’t have issues specific to macros, variables have a local scopes. Examples:

  - `Py_INCREF()`, `Py_DECREF()`
  - `Py_XINCREF()`, `Py_XDECREF()`
  - **`PyObject_INIT`**, **`PyObject_INIT_VAR`**
  - Private functions: **`_PyObject_GC_TRACK()`**, **`_PyObject_GC_UNTRACK()`**, **`_Py_Dealloc()`**

  (Contributed by Victor Stinner in bpo-35059 [https://bugs.python.org/issue?@action=redirect&bpo=35059].)

- The **`PyByteArray_Init()`** and **`PyByteArray_Fini()`** functions have been removed. They did nothing since Python 2.7.4 and Python 3.2.0, were excluded from the limited API (stable ABI), and were not documented. (Contributed by Victor Stinner in bpo-35713 [https://bugs.python.org/issue?@action=redirect&bpo=35713].)

- The result of **`PyExceptionClass_Name()`** is now of type `const char *` rather of `char *`. (Contributed by Serhiy Storchaka in bpo-33818 [https://bugs.python.org/issue?@action=redirect&bpo=33818].)

- The duality of `Modules/Setup.dist` and `Modules/Setup` has been removed. Previously, when updating the CPython source tree, one had to manually copy `Modules/Setup.dist` (inside the source tree) to `Modules/Setup` (inside the build tree) in order to reflect any changes upstream. This was of a small benefit to packagers at the expense of a frequent annoyance to developers following CPython development, as forgetting to copy the file could produce build failures.

  Now the build system always reads from `Modules/Setup` inside the source tree. People who want to customize that file are encouraged to maintain their changes in a git fork of CPython or as patch files, as they would do for any other change to the source tree.

  (Contributed by Antoine Pitrou in bpo-32430 [https://bugs.python.org/issue?@action=redirect&bpo=32430].)

- Functions that convert Python number to C integer like **`PyLong_AsLong()`** and argument parsing functions like **`PyArg_ParseTuple()`** with integer converting format units like `'i'` will now use the **`__index__()`** special method instead of **`__int__()`**, if available. The deprecation warning will be emitted for objects with the `__int__()` method but without the `__index__()` method (like **`Decimal`** and **`Fraction`**). **`PyNumber_Check()`** will now return `1` for objects implementing `__index__()`. **`PyNumber_Long()`**, **`PyNumber_Float()`** and **`PyFloat_AsDouble()`** also now use the `__index__()` method if available. (Contributed by Serhiy Storchaka in bpo-36048 [https://bugs.python.org/issue?

@action=redirect&bpo=36048] and bpo-20092 [https://bugs.python.org/issue?@action=redirect&bpo=20092].)

- Heap-allocated type objects will now increase their reference count in `PyObject_Init()` (and its parallel macro `PyObject_INIT`) instead of in `PyType_GenericAlloc()`. Types that modify instance allocation or deallocation may need to be adjusted. (Contributed by Eddie Elizondo in bpo-35810 [https://bugs.python.org/issue?@action=redirect&bpo=35810].)

- The new function **`PyCode_NewWithPosOnlyArgs()`** allows to create code objects like **`PyCode_New()`**, but with an extra *posonlyargcount* parameter for indicating the number of positional-only arguments. (Contributed by Pablo Galindo in bpo-37221 [https://bugs.python.org/issue?@action=redirect&bpo=37221].)

- **`Py_SetPath()`** now sets `sys.executable` to the program full path (`Py_GetProgramFullPath()`) rather than to the program name (`Py_GetProgramName()`). (Contributed by Victor Stinner in bpo-38234 [https://bugs.python.org/issue?@action=redirect&bpo=38234].)

Deprecated

- The distutils `bdist_wininst` command is now deprecated, use `bdist_wheel` (wheel packages) instead. (Contributed by Victor Stinner in bpo-37481 [https://bugs.python.org/issue?@action=redirect&bpo=37481].)

- Deprecated methods `getchildren()` and `getiterator()` in the `ElementTree` module now emit a `DeprecationWarning` instead of `PendingDeprecationWarning`. They will be removed in Python 3.9. (Contributed by Serhiy

Storchaka in bpo-29209 [https://bugs.python.org/issue?@action=redirect&bpo=29209].)

- Passing an object that is not an instance of **`concurrent.futures.ThreadPoolExecutor`** to **`loop.set_default_executor()`** is deprecated and will be prohibited in Python 3.9. (Contributed by Elvis Pranskevichus in bpo-34075 [https://bugs.python.org/issue?@action=redirect&bpo=34075].)
- The **`__getitem__()`** methods of **`xml.dom.pulldom.DOMEventStream`**, **`wsgiref.util.FileWrapper`** and **`fileinput.FileInput`** have been deprecated.

  Implementations of these methods have been ignoring their *index* parameter, and returning the next item instead. (Contributed by Berker Peksag in bpo-9372 [https://bugs.python.org/issue?@action=redirect&bpo=9372].)
- The **`typing.NamedTuple`** class has deprecated the `_field_types` attribute in favor of the `__annotations__` attribute which has the same information. (Contributed by Raymond Hettinger in bpo-36320 [https://bugs.python.org/issue?@action=redirect&bpo=36320].)
- **`ast`** classes `Num`, `Str`, `Bytes`, `NameConstant` and `Ellipsis` are considered deprecated and will be removed in future Python versions. **`Constant`** should be used instead. (Contributed by Serhiy Storchaka in bpo-32892 [https://bugs.python.org/issue?@action=redirect&bpo=32892].)
- **`ast.NodeVisitor`** methods `visit_Num()`, `visit_Str()`, `visit_Bytes()`, `visit_NameConstant()` and `visit_Ellipsis()` are deprecated now and will not be called in future Python versions. Add the **`visit_Constant()`** method to handle all constant nodes. (Contributed by Serhiy

Storchaka in bpo-36917 [https://bugs.python.org/issue?@action=redirect&bpo=36917].)

- The **`@asyncio.coroutine`** decorator is deprecated and will be removed in version 3.10. Instead of `@asyncio.coroutine`, use **`async def`** instead. (Contributed by Andrew Svetlov in bpo-36921 [https://bugs.python.org/issue?@action=redirect&bpo=36921].)

- In **`asyncio`**, the explicit passing of a *loop* argument has been deprecated and will be removed in version 3.10 for the following: **`asyncio.sleep()`**, **`asyncio.gather()`**, **`asyncio.shield()`**, **`asyncio.wait_for()`**, **`asyncio.wait()`**, **`asyncio.as_completed()`**, **`asyncio.Task`**, **`asyncio.Lock`**, **`asyncio.Event`**, **`asyncio.Condition`**, **`asyncio.Semaphore`**, **`asyncio.BoundedSemaphore`**, **`asyncio.Queue`**, **`asyncio.create_subprocess_exec()`**, and **`asyncio.create_subprocess_shell()`**.

- The explicit passing of coroutine objects to **`asyncio.wait()`** has been deprecated and will be removed in version 3.11. (Contributed by Yury Selivanov in bpo-34790 [https://bugs.python.org/issue?@action=redirect&bpo=34790].)

- The following functions and methods are deprecated in the **`gettext`** module: **`lgettext()`**, **`ldgettext()`**, **`lngettext()`** and **`ldngettext()`**. They return encoded bytes, and it’s possible that you will get unexpected Unicode-related exceptions if there are encoding problems with the translated strings. It’s much better to use alternatives which return Unicode strings in Python 3. These functions have been broken for a long time.

  Function **`bind_textdomain_codeset()`**, methods **`NullTranslations.output_charset()`** and **`NullTranslations.set_output_charset()`**, and the *codeset* parameter of functions `translation()` and `install()` are also deprecated, since they are only used for the `l*gettext()` functions. (Contributed by Serhiy Storchaka in bpo-33710 [https://bugs.python.org/issue?@action=redirect&bpo=33710].)

- The **`isAlive()`** method of `threading.Thread` has been deprecated. (Contributed by Donghee Na in bpo-35283 [https://bugs.python.org/issue?@action=redirect&bpo=35283].)
- Many builtin and extension functions that take integer arguments will now emit a deprecation warning for `Decimal`s, `Fraction`s and any other objects that can be converted to integers only with a loss (e.g. that have the `__int__()` method but do not have the `__index__()` method). In future version they will be errors. (Contributed by Serhiy Storchaka in bpo-36048 [https://bugs.python.org/issue?@action=redirect&bpo=36048].)
- Deprecated passing the following arguments as keyword arguments:
  - *func* in `functools.partialmethod()`, `weakref.finalize()`, `profile.Profile.runcall()`, **`cProfile.Profile.runcall()`**, `bdb.Bdb.runcall()`, `trace.Trace.runfunc()` and `curses.wrapper()`.
  - *function* in `unittest.TestCase.addCleanup()`.
  - *fn* in the `submit()` method of `concurrent.futures.ThreadPoolExecutor` and `concurrent.futures.ProcessPoolExecutor`.
  - *callback* in `contextlib.ExitStack.callback()`, **`contextlib.AsyncExitStack.callback()`** and `contextlib.AsyncExitStack.push_async_callback()`.
  - *c* and *typeid* in the **`create()`** method of **`multiprocessing.managers.Server`** and **`multiprocessing.managers.SharedMemoryServer`**.

  - *obj* in [`weakref.finalize()`].

In future releases of Python, they will be positional-only. (Contributed by Serhiy Storchaka in bpo-36492 [https://bugs.python.org/issue?@action=redirect&bpo=36492].)

API and Feature Removals

The following features and APIs have been removed from Python 3.8:

- Starting with Python 3.3, importing ABCs from **`collections`** was deprecated, and importing should be done from **`collections.abc`**. Being able to import from collections was marked for removal in 3.8, but has been delayed to 3.9. (See gh-81134 [https://github.com/python/cpython/issues/81134].)
- The **`macpath`** module, deprecated in Python 3.7, has been removed. (Contributed by Victor Stinner in bpo-35471 [https://bugs.python.org/issue?@action=redirect&bpo=35471].)
- The function **`platform.popen()`** has been removed, after having been deprecated since Python 3.3: use **`os.popen()`** instead. (Contributed by Victor Stinner in bpo-35345 [https://bugs.python.org/issue?@action=redirect&bpo=35345].)
- The function **`time.clock()`** has been removed, after having been deprecated since Python 3.3: use **`time.perf_counter()`** or **`time.process_time()`** instead, depending on your requirements, to have well-defined behavior. (Contributed by Matthias Bussonnier in bpo-36895 [https://bugs.python.org/issue?@action=redirect&bpo=36895].)
- The `pyvenv` script has been removed in favor of `python3.8 -m venv` to help eliminate confusion as to what Python interpreter the `pyvenv` script is tied to. (Contributed by Brett Cannon in bpo-25427 [https://bugs.python.org/issue?@action=redirect&bpo=25427].)

- `parse_qs`, `parse_qsl`, and `escape` are removed from the **`cgi`** module. They are deprecated in Python 3.2 or older. They should be imported from the `urllib.parse` and `html` modules instead.
- `filemode` function is removed from the **`tarfile`** module. It is not documented and deprecated since Python 3.3.
- The **`XMLParser`** constructor no longer accepts the *html* argument. It never had an effect and was deprecated in Python 3.4. All other parameters are now keyword-only. (Contributed by Serhiy Storchaka in bpo-29209 [https://bugs.python.org/issue?@action=redirect&bpo=29209].)
- Removed the `doctype()` method of **`XMLParser`**. (Contributed by Serhiy Storchaka in bpo-29209 [https://bugs.python.org/issue?@action=redirect&bpo=29209].)
- “unicode_internal” codec is removed. (Contributed by Inada Naoki in bpo-36297 [https://bugs.python.org/issue?@action=redirect&bpo=36297].)
- The `Cache` and `Statement` objects of the **`sqlite3`** module are not exposed to the user. (Contributed by Aviv Palivoda in bpo-30262 [https://bugs.python.org/issue?@action=redirect&bpo=30262].)
- The `bufsize` keyword argument of **`fileinput.input()`** and **`fileinput.FileInput()`** which was ignored and deprecated since Python 3.6 has been removed. bpo-36952 [https://bugs.python.org/issue?@action=redirect&bpo=36952] (Contributed by Matthias Bussonnier.)
- The functions **`sys.set_coroutine_wrapper()`** and **`sys.get_coroutine_wrapper()`** deprecated in Python 3.7 have been removed; bpo-36933 [https://bugs.python.org/issue?@action=redirect&bpo=36933] (Contributed by Matthias Bussonnier.)

Porting to Python 3.8

This section lists previously described changes and other bugfixes that may require changes to your code.

Changes in Python behavior

- Yield expressions (both `yield` and `yield from` clauses) are now disallowed in comprehensions and generator expressions (aside from the iterable expression in the leftmost **`for`** clause). (Contributed by Serhiy Storchaka in bpo-10544 [https://bugs.python.org/issue?@action=redirect&bpo=10544].)
- The compiler now produces a **`SyntaxWarning`** when identity checks (`is` and `is not`) are used with certain types of literals (e.g. strings, numbers). These can often work by accident in CPython, but are not guaranteed by the language spec. The warning advises users to use equality tests (`==` and `!=`) instead. (Contributed by Serhiy Storchaka in bpo-34850 [https://bugs.python.org/issue?@action=redirect&bpo=34850].)
- The CPython interpreter can swallow exceptions in some circumstances. In Python 3.8 this happens in fewer cases. In particular, exceptions raised when getting the attribute from the type dictionary are no longer ignored. (Contributed by Serhiy Storchaka in bpo-35459 [https://bugs.python.org/issue?@action=redirect&bpo=35459].)
- Removed `__str__` implementations from builtin types **`bool`**, **`int`**, **`float`**, **`complex`** and few classes from the standard library. They now inherit `__str__()` from **`object`**. As result, defining the `__repr__()` method in the subclass of these classes will affect their string representation. (Contributed by Serhiy Storchaka in bpo-36793 [https://bugs.python.org/issue?@action=redirect&bpo=36793].)
- On AIX, **`sys.platform`** doesn't contain the major version anymore. It is always `'aix'`, instead of `'aix3'` .. `'aix7'`. Since older Python versions include the version number, so it is recommended to always use

`sys.platform.startswith('aix')`. (Contributed by M. Felt in bpo-36588 [https://bugs.python.org/issue?@action=redirect&bpo=36588].)

- **`PyEval_AcquireLock()`** and **`PyEval_AcquireThread()`** now terminate the current thread if called while the interpreter is finalizing, making them consistent with **`PyEval_RestoreThread()`**, **`Py_END_ALLOW_THREADS()`**, and **`PyGILState_Ensure()`**. If this behavior is not desired, guard the call by checking **`_Py_IsFinalizing()`** or **`sys.is_finalizing()`**. (Contributed by Joannah Nanjekye in bpo-36475 [https://bugs.python.org/issue?@action=redirect&bpo=36475].)

Changes in the Python API

- The **`os.getcwdb()`** function now uses the UTF-8 encoding on Windows, rather than the ANSI code page: see **PEP 529** [https://peps.python.org/pep-0529/] for the rationale. The function is no longer deprecated on Windows. (Contributed by Victor Stinner in bpo-37412 [https://bugs.python.org/issue?@action=redirect&bpo=37412].)
- **`subprocess.Popen`** can now use **`os.posix_spawn()`** in some cases for better performance. On Windows Subsystem for Linux and QEMU User Emulation, the **`Popen`** constructor using **`os.posix_spawn()`** no longer raises an exception on errors like “missing program”. Instead the child process fails with a non-zero **`returncode`**. (Contributed by Joannah Nanjekye and Victor Stinner in bpo-35537 [https://bugs.python.org/issue?@action=redirect&bpo=35537].)
- The *preexec_fn* argument of * **`subprocess.Popen`** is no longer compatible with subinterpreters. The use of the parameter in a subinterpreter now raises **`RuntimeError`**. (Contributed by Eric Snow in bpo-34651 [https://bugs.python.org/issue?@action=redirect&bpo=34651],

modified by Christian Heimes in bpo-37951 [https://bugs.python.org/issue?@action=redirect&bpo=37951].)

- The `imaplib.IMAP4.logout()` method no longer silently ignores arbitrary exceptions. (Contributed by Victor Stinner in bpo-36348 [https://bugs.python.org/issue?@action=redirect&bpo=36348].)
- The function **`platform.popen()`** has been removed, after having been deprecated since Python 3.3: use `os.popen()` instead. (Contributed by Victor Stinner in bpo-35345 [https://bugs.python.org/issue?@action=redirect&bpo=35345].)
- The `statistics.mode()` function no longer raises an exception when given multimodal data. Instead, it returns the first mode encountered in the input data. (Contributed by Raymond Hettinger in bpo-35892 [https://bugs.python.org/issue?@action=redirect&bpo=35892].)
- The `selection()` method of the `tkinter.ttk.Treeview` class no longer takes arguments. Using it with arguments for changing the selection was deprecated in Python 3.6. Use specialized methods like `selection_set()` for changing the selection. (Contributed by Serhiy Storchaka in bpo-31508 [https://bugs.python.org/issue?@action=redirect&bpo=31508].)
- The `writexml()`, `toxml()` and `toprettyxml()` methods of `xml.dom.minidom` and the `write()` method of `xml.etree.ElementTree` now preserve the attribute order specified by the user. (Contributed by Diego Rojas and Raymond Hettinger in bpo-34160 [https://bugs.python.org/issue?@action=redirect&bpo=34160].)
- A `dbm.dumb` database opened with flags `'r'` is now read-only. `dbm.dumb.open()` with flags `'r'` and `'w'` no longer creates a database if it does not exist. (Contributed by Serhiy Storchaka in bpo-32749 [https://bugs.python.org/issue?@action=redirect&bpo=32749].)
- The `doctype()` method defined in a subclass of `XMLParser` will no longer be called and will emit a `RuntimeWarning` instead of a `DeprecationWarning`. Define the `doctype()`

method on a target for handling an XML doctype declaration. (Contributed by Serhiy Storchaka in bpo-29209 [https://bugs.python.org/issue?@action=redirect&bpo=29209].)

- A **`RuntimeError`** is now raised when the custom metaclass doesn't provide the `__classcell__` entry in the namespace passed to `type.__new__`. A **`DeprecationWarning`** was emitted in Python 3.6–3.7. (Contributed by Serhiy Storchaka in bpo-23722 [https://bugs.python.org/issue?@action=redirect&bpo=23722].)
- The **`cProfile.Profile`** class can now be used as a context manager. (Contributed by Scott Sanderson in bpo-29235 [https://bugs.python.org/issue?@action=redirect&bpo=29235].)
- **`shutil.copyfile()`**, **`shutil.copy()`**, **`shutil.copy2()`**, **`shutil.copytree()`** and **`shutil.move()`** use platform-specific "fast-copy" syscalls (see Platform-dependent efficient copy operations section).
- **`shutil.copyfile()`** default buffer size on Windows was changed from 16 KiB to 1 MiB.
- The `PyGC_Head` struct has changed completely. All code that touched the struct member should be rewritten. (See bpo-33597 [https://bugs.python.org/issue?@action=redirect&bpo=33597].)
- The **`PyInterpreterState`** struct has been moved into the "internal" header files (specifically Include/internal/pycore_pystate.h). An opaque `PyInterpreterState` is still available as part of the public API (and stable ABI). The docs indicate that none of the struct's fields are public, so we hope no one has been using them. However, if you do rely on one or more of those private fields and have no alternative then please open a BPO issue. We'll work on helping you adjust (possibly including adding accessor functions to the public API). (See bpo-35886 [https://bugs.python.org/issue?@action=redirect&bpo=35886].)
- The **`mmap.flush()`** method now returns `None` on success and raises an exception on error under all platforms.

Previously, its behavior was platform-dependent: a nonzero value was returned on success; zero was returned on error under Windows. A zero value was returned on success; an exception was raised on error under Unix. (Contributed by Berker Peksag in bpo-2122 [https://bugs.python.org/issue?@action=redirect&bpo=2122].)

- **`xml.dom.minidom`** and **`xml.sax`** modules no longer process external entities by default. (Contributed by Christian Heimes in bpo-17239 [https://bugs.python.org/issue?@action=redirect&bpo=17239].)
- Deleting a key from a read-only **`dbm`** database (**`dbm.dumb`**, **`dbm.gnu`** or **`dbm.ndbm`**) raises **`error`** (**`dbm.dumb.error`**, **`dbm.gnu.error`** or **`dbm.ndbm.error`**) instead of **`KeyError`**. (Contributed by Xiang Zhang in bpo-33106 [https://bugs.python.org/issue?@action=redirect&bpo=33106].)
- Simplified AST for literals. All constants will be represented as **`ast.Constant`** instances. Instantiating old classes `Num`, `Str`, `Bytes`, `NameConstant` and `Ellipsis` will return an instance of `Constant`. (Contributed by Serhiy Storchaka in bpo-32892 [https://bugs.python.org/issue?@action=redirect&bpo=32892].)
- **`expanduser()`** on Windows now prefers the **`USERPROFILE`** environment variable and does not use **`HOME`**, which is not normally set for regular user accounts. (Contributed by Anthony Sottile in bpo-36264 [https://bugs.python.org/issue?@action=redirect&bpo=36264].)
- The exception **`asyncio.CancelledError`** now inherits from **`BaseException`** rather than **`Exception`** and no longer inherits from **`concurrent.futures.CancelledError`**. (Contributed by Yury Selivanov in bpo-32528 [https://bugs.python.org/issue?@action=redirect&bpo=32528].)
- The function **`asyncio.wait_for()`** now correctly waits for cancellation when using an instance of **`asyncio.Task`**. Previously, upon reaching *timeout*, it was cancelled and immediately returned. (Contributed by Elvis

Pranskevichus in bpo-32751 [https://bugs.python.org/issue?@action=redirect&bpo=32751].)
- The function `asyncio.BaseTransport.get_extra_info()` now returns a safe to use socket object when 'socket' is passed to the *name* parameter. (Contributed by Yury Selivanov in bpo-37027 [https://bugs.python.org/issue?@action=redirect&bpo=37027].)
- `asyncio.BufferedProtocol` has graduated to the stable API.

- DLL dependencies for extension modules and DLLs loaded with `ctypes` on Windows are now resolved more securely. Only the system paths, the directory containing the DLL or PYD file, and directories added with `add_dll_directory()` are searched for load-time dependencies. Specifically, **`PATH`** and the current working directory are no longer used, and modifications to these will no longer have any effect on normal DLL resolution. If your application relies on these mechanisms, you should check for `add_dll_directory()` and if it exists, use it to add your DLLs directory while loading your library. Note that Windows 7 users will need to ensure that Windows Update KB2533623 has been installed (this is also verified by the installer). (Contributed by Steve Dower in bpo-36085 [https://bugs.python.org/issue?@action=redirect&bpo=36085].)
- The header files and functions related to pgen have been removed after its replacement by a pure Python implementation. (Contributed by Pablo Galindo in bpo-36623 [https://bugs.python.org/issue?@action=redirect&bpo=36623].)
- `types.CodeType` has a new parameter in the second position of the constructor (*posonlyargcount*) to support positional-only arguments defined in **PEP 570** [https://peps.python.org/pep-0570/]. The first argument (*argcount*) now represents the total number of positional arguments (including positional-only arguments). The

new `replace()` method of [**`types.CodeType`**] can be used to make the code future-proof.

- The parameter `digestmod` for **`hmac.new()`** no longer uses the MD5 digest by default.

Changes in the C API

- The **`PyCompilerFlags`** structure got a new *cf_feature_version* field. It should be initialized to `PY_MINOR_VERSION`. The field is ignored by default, and is used if and only if `PyCF_ONLY_AST` flag is set in *cf_flags*. (Contributed by Guido van Rossum in bpo-35766 [https://bugs.python.org/issue?@action=redirect&bpo=35766].)

- The **`PyEval_ReInitThreads()`** function has been removed from the C API. It should not be called explicitly: use **`PyOS_AfterFork_Child()`** instead. (Contributed by Victor Stinner in bpo-36728 [https://bugs.python.org/issue?@action=redirect&bpo=36728].)

- On Unix, C extensions are no longer linked to libpython except on Android and Cygwin. When Python is embedded, `libpython` must not be loaded with `RTLD_LOCAL`, but `RTLD_GLOBAL` instead. Previously, using `RTLD_LOCAL`, it was already not possible to load C extensions which were not linked to `libpython`, like C extensions of the standard library built by the `*shared*` section of `Modules/Setup`. (Contributed by Victor Stinner in bpo-21536 [https://bugs.python.org/issue?@action=redirect&bpo=21536].)

- Use of `#` variants of formats in parsing or building value (e.g. **`PyArg_ParseTuple()`**, **`Py_BuildValue()`**, **`PyObject_CallFunction()`**, etc.) without `PY_SSIZE_T_CLEAN` defined raises `DeprecationWarning` now. It will be removed in 3.10 or 4.0. Read Parsing arguments and building values for detail. (Contributed by Inada Naoki in bpo-36381 [https://bugs.python.org/issue?@action=redirect&bpo=36381].)

- Instances of heap-allocated types (such as those created with `PyType_FromSpec()`) hold a reference to their type object. Increasing the reference count of these type objects has been moved from `PyType_GenericAlloc()` to the more low-level functions, `PyObject_Init()` and **`PyObject_INIT`**. This makes types created through `PyType_FromSpec()` behave like other classes in managed code.

  Statically allocated types are not affected.

  For the vast majority of cases, there should be no side effect. However, types that manually increase the reference count after allocating an instance (perhaps to work around the bug) may now become immortal. To avoid this, these classes need to call Py_DECREF on the type object during instance deallocation.

  To correctly port these types into 3.8, please apply the following changes:

  - Remove `Py_INCREF` on the type object after allocating an instance - if any. This may happen after calling `PyObject_New`, `PyObject_NewVar`, `PyObject_GC_New()`, `PyObject_GC_NewVar()`, or any other custom allocator that uses `PyObject_Init()` or **`PyObject_INIT`**.

    Example:

    ```
    static foo_struct *
    foo_new(PyObject *type) {
        foo_struct *foo = PyObject_GC_New(foo_struct,
    (PyTypeObject *) type);
        if (foo == NULL)
            return NULL;
    ```

```
#if PY_VERSION_HEX < 0x03080000
    // Workaround for Python issue 35810; no longer
necessary in Python 3.8
    PY_INCREF(type)
#endif
    return foo;
}
```

  - Ensure that all custom `tp_dealloc` functions of heap-allocated types decrease the type's reference count.

    Example:

    ```
    static void
    foo_dealloc(foo_struct *instance) {
        PyObject *type = Py_TYPE(instance);
        PyObject_GC_Del(instance);
    #if PY_VERSION_HEX >= 0x03080000
        // This was not needed before Python 3.8 (Python
    issue 35810)
        Py_DECREF(type);
    #endif
    }
    ```

  (Contributed by Eddie Elizondo in bpo-35810 [https://bugs.python.org/issue?@action=redirect&bpo=35810].)

- The `Py_DEPRECATED()` macro has been implemented for MSVC. The macro now must be placed before the symbol name.

  Example:

  ```
  Py_DEPRECATED(3.8) PyAPI_FUNC(int) Py_OldFunction(void);
  ```

  (Contributed by Zackery Spytz in bpo-33407 [https://bugs.python.org/issue?@action=redirect&bpo=33407].)

- The interpreter does not pretend to support binary compatibility of extension types across feature releases, anymore. A **`PyTypeObject`** exported by a third-party extension module is supposed to have all the slots expected in the current Python version, including **`tp_finalize`** (**`Py_TPFLAGS_HAVE_FINALIZE`** is not checked anymore before reading **`tp_finalize`**).

(Contributed by Antoine Pitrou in bpo-32388 [https://bugs.python.org/issue?@action=redirect&bpo=32388].)

- The functions **`PyNode_AddChild()`** and **`PyParser_AddToken()`** now accept two additional `int` arguments *end_lineno* and *end_col_offset*.
- The `libpython38.a` file to allow MinGW tools to link directly against `python38.dll` is no longer included in the regular Windows distribution. If you require this file, it may be generated with the `gendef` and `dlltool` tools, which are part of the MinGW binutils package:

```
gendef - python38.dll > tmp.def
dlltool --dllname python38.dll --def tmp.def --output-lib
libpython38.a
```

The location of an installed `pythonXY.dll` will depend on the installation options and the version and language of Windows. See Using Python on Windows for more information. The resulting library should be placed in the same directory as `pythonXY.lib`, which is generally the `libs` directory under your Python installation.

(Contributed by Steve Dower in bpo-37351 [https://bugs.python.org/issue?@action=redirect&bpo=37351].)

CPython bytecode changes

- The interpreter loop has been simplified by moving the logic of unrolling the stack of blocks into the compiler.

The compiler emits now explicit instructions for adjusting the stack of values and calling the cleaning-up code for `break`, `continue` and `return`.

Removed opcodes **`BREAK_LOOP`**, **`CONTINUE_LOOP`**, **`SETUP_LOOP`** and **`SETUP_EXCEPT`**. Added new opcodes **`ROT_FOUR`**, **`BEGIN_FINALLY`**, **`CALL_FINALLY`** and **`POP_FINALLY`**. Changed the behavior of **`END_FINALLY`** and **`WITH_CLEANUP_START`**.

(Contributed by Mark Shannon, Antoine Pitrou and Serhiy Storchaka in bpo-17611 [https://bugs.python.org/issue?@action=redirect&bpo=17611].)

- Added new opcode `END_ASYNC_FOR` for handling exceptions raised when awaiting a next item in an `async for` loop. (Contributed by Serhiy Storchaka in bpo-33041 [https://bugs.python.org/issue?@action=redirect&bpo=33041].)
- The `MAP_ADD` now expects the value as the first element in the stack and the key as the second element. This change was made so the key is always evaluated before the value in dictionary comprehensions, as proposed by **PEP 572** [https://peps.python.org/pep-0572/]. (Contributed by Jörn Heissler in bpo-35224 [https://bugs.python.org/issue?@action=redirect&bpo=35224].)

Demos and Tools

Added a benchmark script for timing various ways to access variables: `Tools/scripts/var_access_benchmark.py`. (Contributed by Raymond Hettinger in bpo-35884 [https://bugs.python.org/issue?@action=redirect&bpo=35884].)

Here’s a summary of performance improvements since Python 3.3:

```
Python version                       3.3     3.4     3.5
3.6     3.7     3.8
--------------                       ---     ---     ---     --
-     ---     ---

Variable and attribute read access:
    read_local                       4.0     7.1     7.1
5.4     5.1     3.9
    read_nonlocal                    5.3     7.1     8.1
5.8     5.4     4.4
    read_global                     13.3    15.5    19.0
14.3    13.6     7.6
    read_builtin                    20.0    21.1    21.6
18.5    19.0     7.5
    read_classvar_from_class        20.5    25.6    26.5
20.7    19.5    18.4
    read_classvar_from_instance     18.5    22.8    23.5
18.8    17.1    16.4
    read_instancevar                26.8    32.4    33.1
28.0    26.3    25.4
    read_instancevar_slots          23.7    27.8    31.3
20.8    20.8    20.2
    read_namedtuple                 68.5    73.8    57.5
45.0    46.8    18.4
    read_boundmethod                29.8    37.6    37.9
29.6    26.9    27.7

Variable and attribute write access:
    write_local                      4.6     8.7     9.3
5.5     5.3     4.3
    write_nonlocal                   7.3    10.5    11.1
5.6     5.5     4.7
    write_global                    15.9    19.7    21.2
18.0    18.0    15.8
    write_classvar                  81.9    92.9    96.0
104.6   102.1    39.2
    write_instancevar               36.4    44.6    45.8
40.0    38.9    35.5
    write_instancevar_slots         28.7    35.6    36.1
27.3    26.6    25.7

Data structure read access:
    read_list                       19.2    24.2    24.5
```

```
20.8    20.8    19.0
    read_deque                      19.9    24.7    25.5
20.2    20.6    19.8
    read_dict                       19.7    24.3    25.7
22.3    23.0    21.0
    read_strdict                    17.9    22.6    24.3
19.5    21.2    18.9

Data structure write access:
    write_list                      21.2    27.1    28.5
22.5    21.6    20.0
    write_deque                     23.8    28.7    30.1
22.7    21.8    23.5
    write_dict                      25.9    31.4    33.3
29.3    29.2    24.7
    write_strdict                   22.9    28.4    29.9
27.5    25.2    23.1

Stack (or queue) operations:
    list_append_pop                144.2    93.4   112.7
75.4    74.2    50.8
    deque_append_pop                30.4    43.5    57.0
49.4    49.2    42.5
    deque_append_popleft            30.8    43.7    57.3
49.7    49.7    42.8

Timing loop:
    loop_overhead                    0.3     0.5     0.6
0.4     0.3     0.3
```

The benchmarks were measured on an Intel® Core™ i7-4960HQ processor [https://ark.intel.com/content/www/us/en/ark/products/76088/intel-core-i7-4960hq-processor-6m-cache-up-to-3-80-ghz.html] running the macOS 64-bit builds found at python.org [https://www.python.org/downloads/macos/]. The benchmark script displays timings in nanoseconds.

Notable changes in Python 3.8.1

Due to significant security concerns, the *reuse_address* parameter of **`asyncio.loop.create_datagram_endpoint()`** is no longer supported. This is because of the behavior of the socket option `SO_REUSEADDR` in UDP. For more details, see the documentation for `loop.create_datagram_endpoint()`. (Contributed by Kyle Stanley, Antoine Pitrou, and Yury Selivanov in bpo-37228 [https://bugs.python.org/issue?@action=redirect&bpo=37228].)

Notable changes in Python 3.8.2

Fixed a regression with the `ignore` callback of **`shutil.copytree()`**. The argument types are now str and List[str] again. (Contributed by Manuel Barkhau and Giampaolo Rodola in gh-83571 [https://github.com/python/cpython/issues/83571].)

Notable changes in Python 3.8.3

The constant values of future flags in the **`__future__`** module are updated in order to prevent collision with compiler flags. Previously `PyCF_ALLOW_TOP_LEVEL_AWAIT` was clashing with `CO_FUTURE_DIVISION`. (Contributed by Batuhan Taskaya in gh-83743 [https://github.com/python/cpython/issues/83743])

Notable changes in Python 3.8.8

Earlier Python versions allowed using both `;` and `&` as query parameter separators in **`urllib.parse.parse_qs()`** and **`urllib.parse.parse_qsl()`**. Due to security concerns, and to conform with newer W3C recommendations, this has been changed to allow only a single separator key, with `&` as the default. This change also affects **`cgi.parse()`** and **`cgi.parse_multipart()`** as they use the affected functions internally. For more details, please see their respective documentation. (Contributed by Adam Goldschmidt, Senthil Kumaran and Ken Jin in bpo-42967 [https://bugs.python.org/issue?@action=redirect&bpo=42967].)

Notable changes in Python 3.8.9

A security fix alters the **`ftplib.FTP`** behavior to not trust the IPv4 address sent from the remote server when setting up a passive data channel. We reuse the ftp server IP address instead. For unusual code requiring the old behavior, set a `trust_server_pasv_ipv4_address` attribute on your FTP instance to `True`. (See gh-87451 [https://github.com/python/cpython/issues/87451])

Notable changes in Python 3.8.10

macOS 11.0 (Big Sur) and Apple Silicon Mac support

As of 3.8.10, Python now supports building and running on macOS 11 (Big Sur) and on Apple Silicon Macs (based on the `ARM64` architecture). A new universal build variant, `universal2`, is now available to natively support both `ARM64` and `Intel 64` in one set of executables. Note that support for “weaklinking”, building binaries targeted for newer versions of macOS that will also run correctly on older versions by testing at runtime for missing features, is not included in this backport from Python 3.9; to support a range of macOS versions, continue to target for and build on the oldest version in the range.

(Originally contributed by Ronald Oussoren and Lawrence D’Anna in gh-85272 [https://github.com/python/cpython/issues/85272], with fixes by FX Coudert and Eli Rykoff, and backported to 3.8 by Maxime Bélanger and Ned Deily)

Notable changes in Python 3.8.10

urllib.parse

The presence of newline or tab characters in parts of a URL allows for some forms of attacks. Following the WHATWG specification that updates **RFC 3986** [https://datatracker.ietf.org/doc/html/rfc3986.html], ASCII newline `\n`, `\r` and tab `\t` characters are stripped from the URL by the parser in **`urllib.parse`** preventing such attacks. The removal characters are controlled by a new module level variable `urllib.parse._UNSAFE_URL_BYTES_TO_REMOVE`. (See bpo-43882 [https://bugs.python.org/issue?@action=redirect&bpo=43882])

Notable changes in Python 3.8.12

Changes in the Python API

Starting with Python 3.8.12 the **`ipaddress`** module no longer accepts any leading zeros in IPv4 address strings. Leading zeros are ambiguous and interpreted as octal notation by some libraries. For example the legacy function **`socket.inet_aton()`** treats leading zeros as octal notation. glibc implementation of modern **`inet_pton()`** does not accept any leading zeros.

(Originally contributed by Christian Heimes in bpo-36384 [https://bugs.python.org/issue?@action=redirect&bpo=36384], and backported to 3.8 by Achraf Merzouki.)

Notable security feature in 3.8.14

Converting between **`int`** and **`str`** in bases other than 2 (binary), 4, 8 (octal), 16 (hexadecimal), or 32 such as base 10 (decimal) now raises a **`ValueError`** if the number of digits in string form is above a limit to avoid potential denial of service attacks due to the algorithmic complexity. This is a mitigation for **CVE 2020-10735** [https://www.cve.org/CVERecord?id=CVE-2020-10735]. This limit can be configured or disabled by environment variable, command line flag, or **`sys`** APIs. See the integer string conversion length limitation documentation. The default limit is 4300 digits in string form.

Notable changes in 3.8.17

tarfile

- The extraction methods in **`tarfile`**, and **`shutil.unpack_archive()`**, have a new a *filter* argument that allows limiting tar features than may be surprising or dangerous, such as creating files outside the destination directory. See Extraction filters for details. In Python 3.12, use without the *filter* argument will show a **`DeprecationWarning`**. In Python 3.14, the default will switch to `'data'`. (Contributed by Petr Viktorin in **PEP 706** [https://peps.python.org/pep-0706/].)

What’s New In Python 3.7

Editor:: Elvis Pranskevichus <elvis@magic.io>

This article explains the new features in Python 3.7, compared to 3.6. Python 3.7 was released on June 27, 2018. For full details, see the changelog.

Summary – Release Highlights

New syntax features:

- PEP 563, postponed evaluation of type annotations.

Backwards incompatible syntax changes:

- **`async`** and **`await`** are now reserved keywords.

New library modules:

- **`contextvars`**: PEP 567 – Context Variables
- **`dataclasses`**: PEP 557 – Data Classes
- importlib.resources

New built-in features:

- PEP 553, the new **`breakpoint()`** function.

Python data model improvements:

- PEP 562, customization of access to module attributes.
- PEP 560, core support for typing module and generic types.
- the insertion-order preservation nature of dict objects has been declared [https://mail.python.org/pipermail/python-

dev/2017-December/151283.html] to be an official part of the Python language spec.

Significant improvements in the standard library:

- The **`asyncio`** module has received new features, significant usability and performance improvements.
- The **`time`** module gained support for functions with nanosecond resolution.

CPython implementation improvements:

- Avoiding the use of ASCII as a default text encoding:
  - PEP 538, legacy C locale coercion
  - PEP 540, forced UTF-8 runtime mode
- PEP 552, deterministic .pycs
- New Python Development Mode
- PEP 565, improved **`DeprecationWarning`** handling

C API improvements:

- PEP 539, new C API for thread-local storage

Documentation improvements:

- PEP 545, Python documentation translations
- New documentation translations: Japanese [https://docs.python.org/ja/], French [https://docs.python.org/fr/], and Korean [https://docs.python.org/ko/].

This release features notable performance improvements in many areas. The Optimizations section lists them in detail.

For a list of changes that may affect compatibility with previous Python releases please refer to the Porting to Python 3.7 section.

New Features

PEP 563: Postponed Evaluation of Annotations

The advent of type hints in Python uncovered two glaring usability issues with the functionality of annotations added in **PEP 3107** [https://peps.python.org/pep-3107/] and refined further in **PEP 526** [https://peps.python.org/pep-0526/]:

- annotations could only use names which were already available in the current scope, in other words they didn't support forward references of any kind; and
- annotating source code had adverse effects on startup time of Python programs.

Both of these issues are fixed by postponing the evaluation of annotations. Instead of compiling code which executes expressions in annotations at their definition time, the compiler stores the annotation in a string form equivalent to the AST of the expression in question. If needed, annotations can be resolved at runtime using `typing.get_type_hints()`. In the common case where this is not required, the annotations are cheaper to store (since short strings are interned by the interpreter) and make startup time faster.

Usability-wise, annotations now support forward references, making the following syntax valid:

```
class C:
    @classmethod
    def from_string(cls, source: str) -> C:
        ...

    def validate_b(self, obj: B) -> bool:
        ...
```

```
class B:
    ...
```

Since this change breaks compatibility, the new behavior needs to be enabled on a per-module basis in Python 3.7 using a `__future__` import:

```
from __future__ import annotations
```

It will become the default in Python 3.10.

See also

PEP 563 [https://peps.python.org/pep-0563/] – Postponed evaluation of annotations
PEP written and implemented by Łukasz Langa.

PEP 538: Legacy C Locale Coercion

An ongoing challenge within the Python 3 series has been determining a sensible default strategy for handling the “7-bit ASCII” text encoding assumption currently implied by the use of the default C or POSIX locale on non-Windows platforms.

PEP 538 [https://peps.python.org/pep-0538/] updates the default interpreter command line interface to automatically coerce that locale to an available UTF-8 based locale as described in the documentation of the new `PYTHONCOERCECLOCALE` environment variable. Automatically setting `LC_CTYPE` this way means that both the core interpreter and locale-aware C extensions (such as `readline`) will assume the use of UTF-8 as the default text encoding, rather than ASCII.

The platform support definition in PEP 11 [https://peps.python.org/pep-0011/] has also been updated to limit full text handling support to suitably configured non-ASCII based locales.

As part of this change, the default error handler for **`stdin`** and **`stdout`** is now `surrogateescape` (rather than `strict`) when using any of the defined coercion target locales (currently `C.UTF-8`, `C.utf8`, and `UTF-8`). The default error handler for **`stderr`** continues to be `backslashreplace`, regardless of locale.

Locale coercion is silent by default, but to assist in debugging potentially locale related integration problems, explicit warnings (emitted directly on **`stderr`**) can be requested by setting `PYTHONCOERCECLOCALE=warn`. This setting will also cause the Python runtime to emit a warning if the legacy C locale remains active when the core interpreter is initialized.

While **PEP 538** [https://peps.python.org/pep-0538/]’s locale coercion has the benefit of also affecting extension modules (such as GNU `readline`), as well as child processes (including those running non-Python applications and older versions of Python), it has the downside of requiring that a suitable target locale be present on the running system. To better handle the case where no suitable target locale is available (as occurs on RHEL/CentOS 7, for example), Python 3.7 also implements PEP 540: Forced UTF-8 Runtime Mode.

See also

PEP 538 [https://peps.python.org/pep-0538/] – Coercing the legacy C locale to a UTF-8 based locale
PEP written and implemented by Nick Coghlan.

PEP 540: Forced UTF-8 Runtime Mode

The new **`-X`** `utf8` command line option and **`PYTHONUTF8`** environment variable can be used to enable the Python UTF-8 Mode.

When in UTF-8 mode, CPython ignores the locale settings, and uses the UTF-8 encoding by default. The error handlers for **`sys.stdin`** and **`sys.stdout`** streams are set to `surrogateescape`.

The forced UTF-8 mode can be used to change the text handling behavior in an embedded Python interpreter without changing the locale settings of an embedding application.

While **PEP 540** [https://peps.python.org/pep-0540/]'s UTF-8 mode has the benefit of working regardless of which locales are available on the running system, it has the downside of having no effect on extension modules (such as GNU `readline`), child processes running non-Python applications, and child processes running older versions of Python. To reduce the risk of corrupting text data when communicating with such components, Python 3.7 also implements PEP 540: Forced UTF-8 Runtime Mode).

The UTF-8 mode is enabled by default when the locale is `C` or `POSIX`, and the **PEP 538** [https://peps.python.org/pep-0538/] locale coercion feature fails to change it to a UTF-8 based alternative (whether that failure is due to `PYTHONCOERCECLOCALE=0` being set, `LC_ALL` being set, or the lack of a suitable target locale).

See also

PEP 540 [https://peps.python.org/pep-0540/] – Add a new UTF-8 mode

PEP written and implemented by Victor Stinner

PEP 553: Built-in `breakpoint()`

Python 3.7 includes the new built-in `breakpoint()` function as an easy and consistent way to enter the Python debugger.

Built-in `breakpoint()` calls `sys.breakpointhook()`. By default, the latter imports `pdb` and then calls `pdb.set_trace()`, but by binding `sys.breakpointhook()` to the function of your choosing, `breakpoint()` can enter any debugger. Additionally, the environment variable `PYTHONBREAKPOINT` can be set to the callable of your debugger of choice. Set `PYTHONBREAKPOINT=0` to completely disable built-in `breakpoint()`.

See also

PEP 553 [https://peps.python.org/pep-0553/] – Built-in breakpoint()
PEP written and implemented by Barry Warsaw

PEP 539: New C API for Thread-Local Storage

While Python provides a C API for thread-local storage support; the existing Thread Local Storage (TLS) API has used int to represent TLS keys across all platforms. This has not generally been a problem for officially support platforms, but that is neither POSIX-compliant, nor portable in any practical sense.

PEP 539 [https://peps.python.org/pep-0539/] changes this by providing a new Thread Specific Storage (TSS) API to CPython which supersedes use of the existing TLS API within the CPython interpreter, while deprecating the existing API.

The TSS API uses a new type `Py_tss_t` instead of int to represent TSS keys–an opaque type the definition of which may depend on the underlying TLS implementation. Therefore, this will allow to build CPython on platforms where the native TLS key is defined in a way that cannot be safely cast to int.

Note that on platforms where the native TLS key is defined in a way that cannot be safely cast to int, all functions of the existing TLS API will be no-op and immediately return failure. This indicates clearly that the old API is not supported on platforms where it cannot be used reliably, and that no effort will be made to add such support.

See also

PEP 539 [https://peps.python.org/pep-0539/] – A New C-API for Thread-Local Storage in CPython

PEP written by Erik M. Bray; implementation by Masayuki Yamamoto.

PEP 562: Customization of Access to Module Attributes

Python 3.7 allows defining `__getattr__()` on modules and will call it whenever a module attribute is otherwise not found. Defining `__dir__()` on modules is now also allowed.

A typical example of where this may be useful is module attribute deprecation and lazy loading.

See also

PEP 562 [https://peps.python.org/pep-0562/] – Module `__getattr__` and `__dir__`

PEP written and implemented by Ivan Levkivskyi

PEP 564: New Time Functions With Nanosecond Resolution

The resolution of clocks in modern systems can exceed the limited precision of a floating-point number returned by the `time.time()` function and its variants. To avoid loss of precision, **PEP 564** [https://peps.python.org/pep-0564/] adds six new “nanosecond” variants of the existing timer functions to the `time` module:

- `time.clock_gettime_ns()`
- `time.clock_settime_ns()`
- `time.monotonic_ns()`
- `time.perf_counter_ns()`
- `time.process_time_ns()`
- `time.time_ns()`

The new functions return the number of nanoseconds as an integer value.

Measurements [https://peps.python.org/pep-0564/#annex-clocks-resolution-in-python] show that on Linux and Windows the resolution of `time.time_ns()` is approximately 3 times better than that of `time.time()`.

See also

PEP 564 [https://peps.python.org/pep-0564/] – Add new time functions with nanosecond resolution

PEP written and implemented by Victor Stinner

PEP 565: Show DeprecationWarning in `__main__`

The default handling of `DeprecationWarning` has been changed such that these warnings are once more shown by default, but only when the code triggering them is running directly in the `__main__` module. As a result, developers of single file scripts and those using Python interactively should once again start seeing deprecation warnings for the APIs they use, but deprecation warnings triggered by imported application, library and framework modules will continue to be hidden by default.

As a result of this change, the standard library now allows developers to choose between three different deprecation warning behaviours:

- `FutureWarning`: always displayed by default, recommended for warnings intended to be seen by application end users (e.g. for deprecated application configuration settings).
- `DeprecationWarning`: displayed by default only in `__main__` and when running tests, recommended for warnings intended to be seen by other Python developers where a version upgrade may result in changed behaviour or an error.
- `PendingDeprecationWarning`: displayed by default only when running tests, intended for cases where a future version upgrade will change the warning category to `DeprecationWarning` or `FutureWarning`.

Previously both `DeprecationWarning` and `PendingDeprecationWarning` were only visible when running tests, which meant that developers primarily writing single file scripts or using Python interactively could be surprised by breaking changes in the APIs they used.

See also

PEP 565 [https://peps.python.org/pep-0565/] – Show DeprecationWarning in `__main__`
PEP written and implemented by Nick Coghlan

PEP 560: Core Support for `typing` module and Generic Types

Initially **PEP 484** [https://peps.python.org/pep-0484/] was designed in such way that it would not introduce *any* changes to the core CPython interpreter. Now type hints and the `typing` module are extensively used by the community, so this restriction is removed. The PEP introduces two special methods `__class_getitem__()` and `__mro_entries__()`, these methods are now used by most classes and special constructs in `typing`. As a result, the speed of various operations with types increased up to 7 times, the generic types can be used without metaclass conflicts, and several long standing bugs in `typing` module are fixed.

See also

PEP 560 [https://peps.python.org/pep-0560/] – Core support for typing module and generic types
PEP written and implemented by Ivan Levkivskyi

PEP 552: Hash-based .pyc Files

Python has traditionally checked the up-to-dateness of bytecode cache files (i.e., `.pyc` files) by comparing the source metadata (last-modified timestamp and size) with source metadata saved in the cache file header when it was generated. While effective, this invalidation method has its drawbacks. When filesystem timestamps are too coarse, Python can miss source updates, leading to user confusion. Additionally, having a timestamp in the cache file is problematic for build reproducibility [https://reproducible-builds.org/] and content-based build systems.

PEP 552 [https://peps.python.org/pep-0552/] extends the pyc format to allow the hash of the source file to be used for invalidation instead of the source timestamp. Such `.pyc` files are called “hash-based”. By default, Python still uses timestamp-based invalidation and does not generate hash-based `.pyc` files at runtime. Hash-based `.pyc` files may be generated with **`py_compile`** or **`compileall`**.

Hash-based `.pyc` files come in two variants: checked and unchecked. Python validates checked hash-based `.pyc` files against the corresponding source files at runtime but doesn’t do so for unchecked hash-based pycs. Unchecked hash-based `.pyc` files are a useful performance optimization for environments where a system external to Python (e.g., the build system) is responsible for keeping `.pyc` files up-to-date.

See Cached bytecode invalidation for more information.

See also

PEP 552 [https://peps.python.org/pep-0552/] – Deterministic pycs
PEP written and implemented by Benjamin Peterson

PEP 545: Python Documentation Translations

PEP 545 [https://peps.python.org/pep-0545/] describes the process of creating and maintaining Python documentation translations.

Three new translations have been added:

- Japanese: https://docs.python.org/ja/
- French: https://docs.python.org/fr/
- Korean: https://docs.python.org/ko/

See also

PEP 545 [https://peps.python.org/pep-0545/] – Python Documentation Translations

PEP written and implemented by Julien Palard, Inada Naoki, and Victor Stinner.

Python Development Mode (-X dev)

The new `-X` `dev` command line option or the new `PYTHONDEVMODE` environment variable can be used to enable Python Development Mode. When in development mode, Python performs additional runtime checks that are too expensive to be enabled by default. See Python Development Mode documentation for the full description.

Other Language Changes

- An `await` expression and comprehensions containing an `async for` clause were illegal in the expressions in formatted string literals due to a problem with the implementation. In Python 3.7 this restriction was lifted.
- More than 255 arguments can now be passed to a function, and a function can now have more than 255 parameters. (Contributed by Serhiy Storchaka in bpo-12844 [https://bugs.python.org/issue?@action=redirect&bpo=12844] and bpo-18896 [https://bugs.python.org/issue?@action=redirect&bpo=18896].)

- `bytes.fromhex()` and `bytearray.fromhex()` now ignore all ASCII whitespace, not only spaces. (Contributed by Robert Xiao in bpo-28927 [https://bugs.python.org/issue?@action=redirect&bpo=28927].)
- `str`, `bytes`, and `bytearray` gained support for the new `isascii()` method, which can be used to test if a string or bytes contain only the ASCII characters. (Contributed by INADA Naoki in bpo-32677 [https://bugs.python.org/issue?@action=redirect&bpo=32677].)
- `ImportError` now displays module name and module `__file__` path when `from ... import ...` fails. (Contributed by Matthias Bussonnier in bpo-29546 [https://bugs.python.org/issue?@action=redirect&bpo=29546].)
- Circular imports involving absolute imports with binding a submodule to a name are now supported. (Contributed by Serhiy Storchaka in bpo-30024 [https://bugs.python.org/issue?@action=redirect&bpo=30024].)
- `object.__format__(x, '')` is now equivalent to `str(x)` rather than `format(str(self), '')`. (Contributed by Serhiy Storchaka in bpo-28974 [https://bugs.python.org/issue?@action=redirect&bpo=28974].)
- In order to better support dynamic creation of stack traces, `types.TracebackType` can now be instantiated from Python code, and the `tb_next` attribute on tracebacks is now writable. (Contributed by Nathaniel J. Smith in bpo-30579 [https://bugs.python.org/issue?@action=redirect&bpo=30579].)
- When using the `-m` switch, `sys.path[0]` is now eagerly expanded to the full starting directory path, rather than being left as the empty directory (which allows imports from the *current* working directory at the time when an import occurs) (Contributed by Nick Coghlan in bpo-33053 [https://bugs.python.org/issue?@action=redirect&bpo=33053].)
- The new `-X` `importtime` option or the `PYTHONPROFILEIMPORTTIME` environment variable can be used to show the timing of each module import. (Contributed

by Inada Naoki in bpo-31415 [https://bugs.python.org/issue?@action=redirect&bpo=31415].)

New Modules

contextvars

The new `contextvars` module and a set of new C APIs introduce support for *context variables*. Context variables are conceptually similar to thread-local variables. Unlike TLS, context variables support asynchronous code correctly.

The `asyncio` and `decimal` modules have been updated to use and support context variables out of the box. Particularly the active decimal context is now stored in a context variable, which allows decimal operations to work with the correct context in asynchronous code.

See also

PEP 567 [https://peps.python.org/pep-0567/] – Context Variables

PEP written and implemented by Yury Selivanov

dataclasses

The new `dataclass()` decorator provides a way to declare *data classes*. A data class describes its attributes using class variable annotations. Its constructor and other magic methods, such as `__repr__()`, `__eq__()`, and `__hash__()` are generated automatically.

Example:

```
@dataclass
class Point:
    x: float
    y: float
    z: float = 0.0

p = Point(1.5, 2.5)
print(p)   # produces "Point(x=1.5, y=2.5, z=0.0)"
```

See also

PEP 557 [https://peps.python.org/pep-0557/] – Data Classes
PEP written and implemented by Eric V. Smith

importlib.resources

The new `importlib.resources` module provides several new APIs and one new ABC for access to, opening, and reading *resources* inside packages. Resources are roughly similar to files inside packages, but they needn’t be actual files on the physical file system. Module loaders can provide a `get_resource_reader()` function which returns a `importlib.abc.ResourceReader` instance to support this new API. Built-in file path loaders and zip file loaders both support this.

Contributed by Barry Warsaw and Brett Cannon in bpo-32248 [https://bugs.python.org/issue?@action=redirect&bpo=32248].

See also

importlib_resources [https://importlib-resources.readthedocs.io/en/latest/] – a PyPI backport for earlier Python versions.

Improved Modules

argparse

The new **`ArgumentParser.parse_intermixed_args()`** method allows intermixing options and positional arguments. (Contributed by paul.j3 in bpo-14191 [https://bugs.python.org/issue?@action=redirect&bpo=14191].)

asyncio

The **`asyncio`** module has received many new features, usability and performance improvements. Notable changes include:

- The new provisional **`asyncio.run()`** function can be used to run a coroutine from synchronous code by automatically creating and destroying the event loop. (Contributed by Yury Selivanov in bpo-32314 [https://bugs.python.org/issue?@action=redirect&bpo=32314].)
- asyncio gained support for **`contextvars`**. **`loop.call_soon()`**, **`loop.call_soon_threadsafe()`**, **`loop.call_later()`**, **`loop.call_at()`**, and **`Future.add_done_callback()`** have a new optional keyword-only *context* parameter. **`Tasks`** now track their context automatically. See **PEP 567** [https://peps.python.org/pep-0567/] for more details. (Contributed by Yury Selivanov in bpo-32436 [https://bugs.python.org/issue?@action=redirect&bpo=32436].)
- The new **`asyncio.create_task()`** function has been added as a shortcut to `asyncio.get_event_loop().create_task()`. (Contributed by Andrew Svetlov in bpo-32311 [https://bugs.python.org/issue?@action=redirect&bpo=32311].)

- The new `loop.start_tls()` method can be used to upgrade an existing connection to TLS. (Contributed by Yury Selivanov in bpo-23749 [https://bugs.python.org/issue?@action=redirect&bpo=23749].)
- The new `loop.sock_recv_into()` method allows reading data from a socket directly into a provided buffer making it possible to reduce data copies. (Contributed by Antoine Pitrou in bpo-31819 [https://bugs.python.org/issue?@action=redirect&bpo=31819].)
- The new `asyncio.current_task()` function returns the currently running `Task` instance, and the new `asyncio.all_tasks()` function returns a set of all existing `Task` instances in a given loop. The **`Task.current_task()`** and **`Task.all_tasks()`** methods have been deprecated. (Contributed by Andrew Svetlov in bpo-32250 [https://bugs.python.org/issue?@action=redirect&bpo=32250].)
- The new *provisional* `BufferedProtocol` class allows implementing streaming protocols with manual control over the receive buffer. (Contributed by Yury Selivanov in bpo-32251 [https://bugs.python.org/issue?@action=redirect&bpo=32251].)
- The new `asyncio.get_running_loop()` function returns the currently running loop, and raises a `RuntimeError` if no loop is running. This is in contrast with `asyncio.get_event_loop()`, which will *create* a new event loop if none is running. (Contributed by Yury Selivanov in bpo-32269 [https://bugs.python.org/issue?@action=redirect&bpo=32269].)
- The new `StreamWriter.wait_closed()` coroutine method allows waiting until the stream writer is closed. The new `StreamWriter.is_closing()` method can be used to

determine if the writer is closing. (Contributed by Andrew Svetlov in bpo-32391 [https://bugs.python.org/issue?@action=redirect&bpo=32391].)

- The new **`loop.sock_sendfile()`** coroutine method allows sending files using **`os.sendfile`** when possible. (Contributed by Andrew Svetlov in bpo-32410 [https://bugs.python.org/issue?@action=redirect&bpo=32410].)

- The new **`Future.get_loop()`** and `Task.get_loop()` methods return the instance of the loop on which a task or a future were created. **`Server.get_loop()`** allows doing the same for **`asyncio.Server`** objects. (Contributed by Yury Selivanov in bpo-32415 [https://bugs.python.org/issue?@action=redirect&bpo=32415] and Srinivas Reddy Thatiparthy in bpo-32418 [https://bugs.python.org/issue?@action=redirect&bpo=32418].)

- It is now possible to control how instances of **`asyncio.Server`** begin serving. Previously, the server would start serving immediately when created. The new *start_serving* keyword argument to **`loop.create_server()`** and **`loop.create_unix_server()`**, as well as **`Server.start_serving()`**, and **`Server.serve_forever()`** can be used to decouple server instantiation and serving. The new **`Server.is_serving()`** method returns `True` if the server is serving. **`Server`** objects are now asynchronous context managers:

  ```
  srv = await loop.create_server(...)

  async with srv:
      # some code

  # At this point, srv is closed and no longer accepts new
  connections.
  ```

(Contributed by Yury Selivanov in bpo-32662 [https://bugs.python.org/issue?@action=redirect&bpo=32662].)

- Callback objects returned by `loop.call_later()` gained the new `when()` method which returns an absolute scheduled callback timestamp. (Contributed by Andrew Svetlov in bpo-32741 [https://bugs.python.org/issue?@action=redirect&bpo=32741].)
- The `loop.create_datagram_endpoint()` method gained support for Unix sockets. (Contributed by Quentin Dawans in bpo-31245 [https://bugs.python.org/issue?@action=redirect&bpo=31245].)
- The `asyncio.open_connection()`, `asyncio.start_server()` functions, `loop.create_connection()`, `loop.create_server()`, `loop.create_accepted_socket()` methods and their corresponding UNIX socket variants now accept the *ssl_handshake_timeout* keyword argument. (Contributed by Neil Aspinall in bpo-29970 [https://bugs.python.org/issue?@action=redirect&bpo=29970].)
- The new `Handle.cancelled()` method returns `True` if the callback was cancelled. (Contributed by Marat Sharafutdinov in bpo-31943 [https://bugs.python.org/issue?@action=redirect&bpo=31943].)
- The asyncio source has been converted to use the `async`/`await` syntax. (Contributed by Andrew Svetlov in bpo-32193 [https://bugs.python.org/issue?@action=redirect&bpo=32193].)
- The new `ReadTransport.is_reading()` method can be used to determine the reading state of the transport. Additionally, calls to `ReadTransport.resume_reading()` and `ReadTransport.pause_reading()` are now idempotent.

(Contributed by Yury Selivanov in bpo-32356 [https://bugs.python.org/issue?@action=redirect&bpo=32356].)

- Loop methods which accept socket paths now support passing path-like objects. (Contributed by Yury Selivanov in bpo-32066 [https://bugs.python.org/issue?@action=redirect&bpo=32066].)
- In **`asyncio`** TCP sockets on Linux are now created with `TCP_NODELAY` flag set by default. (Contributed by Yury Selivanov and Victor Stinner in bpo-27456 [https://bugs.python.org/issue?@action=redirect&bpo=27456].)
- Exceptions occurring in cancelled tasks are no longer logged. (Contributed by Yury Selivanov in bpo-30508 [https://bugs.python.org/issue?@action=redirect&bpo=30508].)
- New `WindowsSelectorEventLoopPolicy` and `WindowsProactorEventLoopPolicy` classes. (Contributed by Yury Selivanov in bpo-33792 [https://bugs.python.org/issue?@action=redirect&bpo=33792].)

Several `asyncio` APIs have been deprecated.

binascii

The **`b2a_uu()`** function now accepts an optional *backtick* keyword argument. When it’s true, zeros are represented by ``'`'`` instead of spaces. (Contributed by Xiang Zhang in bpo-30103 [https://bugs.python.org/issue?@action=redirect&bpo=30103].)

calendar

The **`HTMLCalendar`** class has new class attributes which ease the customization of CSS classes in the produced HTML calendar. (Contributed by Oz Tiram in bpo-30095 [https://bugs.python.org/issue?@action=redirect&bpo=30095].)

collections

`collections.namedtuple()` now supports default values. (Contributed by Raymond Hettinger in bpo-32320 [https://bugs.python.org/issue?@action=redirect&bpo=32320].)

compileall

`compileall.compile_dir()` learned the new *invalidation_mode* parameter, which can be used to enable hash-based .pyc invalidation. The invalidation mode can also be specified on the command line using the new `--invalidation-mode` argument. (Contributed by Benjamin Peterson in bpo-31650 [https://bugs.python.org/issue?@action=redirect&bpo=31650].)

concurrent.futures

`ProcessPoolExecutor` and **`ThreadPoolExecutor`** now support the new *initializer* and *initargs* constructor arguments. (Contributed by Antoine Pitrou in bpo-21423 [https://bugs.python.org/issue?@action=redirect&bpo=21423].)

The **`ProcessPoolExecutor`** can now take the multiprocessing context via the new *mp_context* argument. (Contributed by Thomas Moreau in bpo-31540 [https://bugs.python.org/issue?@action=redirect&bpo=31540].)

contextlib

The new **`nullcontext()`** is a simpler and faster no-op context manager than **`ExitStack`**. (Contributed by Jesse-Bakker in bpo-10049 [https://bugs.python.org/issue?@action=redirect&bpo=10049].)

The new **`asynccontextmanager()`**, **`AbstractAsyncContextManager`**, and **`AsyncExitStack`** have been added to complement their synchronous counterparts. (Contributed by Jelle Zijlstra in bpo-29679 [https://bugs.python.org/issue?@action=redirect&bpo=29679] and bpo-30241 [https://bugs.python.org/issue?@action=redirect&bpo=30241], and by Alexander Mohr and Ilya Kulakov in bpo-29302 [https://bugs.python.org/issue?@action=redirect&bpo=29302].)

cProfile

The **`cProfile`** command line now accepts `-m module_name` as an alternative to script path. (Contributed by Sanyam Khurana in bpo-21862 [https://bugs.python.org/issue?@action=redirect&bpo=21862].)

crypt

The **`crypt`** module now supports the Blowfish hashing method. (Contributed by Serhiy Storchaka in bpo-31664 [https://bugs.python.org/issue?@action=redirect&bpo=31664].)

The **`mksalt()`** function now allows specifying the number of rounds for hashing. (Contributed by Serhiy Storchaka in bpo-31702 [https://bugs.python.org/issue?@action=redirect&bpo=31702].)

datetime

The new **`datetime.fromisoformat()`** method constructs a **`datetime`** object from a string in one of the formats output by **`datetime.isoformat()`**. (Contributed by Paul Ganssle in bpo-15873 [https://bugs.python.org/issue?@action=redirect&bpo=15873].)

The **`tzinfo`** class now supports sub-minute offsets. (Contributed by Alexander Belopolsky in bpo-5288 [https://bugs.python.org/issue?@action=redirect&bpo=5288].)

dbm

`dbm.dumb` now supports reading read-only files and no longer writes the index file when it is not changed.

decimal

The `decimal` module now uses context variables to store the decimal context. (Contributed by Yury Selivanov in bpo-32630 [https://bugs.python.org/issue?@action=redirect&bpo=32630].)

dis

The `dis()` function is now able to disassemble nested code objects (the code of comprehensions, generator expressions and nested functions, and the code used for building nested classes). The maximum depth of disassembly recursion is controlled by the new *depth* parameter. (Contributed by Serhiy Storchaka in bpo-11822 [https://bugs.python.org/issue?@action=redirect&bpo=11822].)

distutils

`README.rst` is now included in the list of distutils standard READMEs and therefore included in source distributions. (Contributed by Ryan Gonzalez in bpo-11913 [https://bugs.python.org/issue?@action=redirect&bpo=11913].)

enum

The `Enum` learned the new `_ignore_` class property, which allows listing the names of properties which should not become enum members. (Contributed by Ethan Furman in bpo-31801 [https://bugs.python.org/issue?@action=redirect&bpo=31801].)

In Python 3.8, attempting to check for non-Enum objects in `Enum` classes will raise a `TypeError` (e.g. `1 in Color`); similarly, attempting to check for non-Flag objects in a `Flag` member will raise `TypeError` (e.g. `1 in Perm.RW`); currently, both operations return `False` instead and are deprecated. (Contributed by Ethan Furman in bpo-33217 [https://bugs.python.org/issue?@action=redirect&bpo=33217].)

functools

`functools.singledispatch()` now supports registering implementations using type annotations. (Contributed by Łukasz Langa in bpo-32227 [https://bugs.python.org/issue?@action=redirect&bpo=32227].)

gc

The new `gc.freeze()` function allows freezing all objects tracked by the garbage collector and excluding them from future collections. This can be used before a POSIX `fork()` call to make the GC copy-on-write friendly or to speed up collection. The new `gc.unfreeze()` functions reverses this operation. Additionally, `gc.get_freeze_count()` can be used to obtain the number of frozen objects. (Contributed by Li Zekun in bpo-31558 [https://bugs.python.org/issue?@action=redirect&bpo=31558].)

hmac

The `hmac` module now has an optimized one-shot `digest()` function, which is up to three times faster than `HMAC()`. (Contributed by Christian Heimes in bpo-32433 [https://bugs.python.org/issue?@action=redirect&bpo=32433].)

http.client

`HTTPConnection` and **`HTTPSConnection`** now support the new *blocksize* argument for improved upload throughput. (Contributed by Nir Soffer in bpo-31945 [https://bugs.python.org/issue?@action=redirect&bpo=31945].)

http.server

`SimpleHTTPRequestHandler` now supports the HTTP `If-Modified-Since` header. The server returns the 304 response status if the target file was not modified after the time specified in the header. (Contributed by Pierre Quentel in bpo-29654 [https://bugs.python.org/issue?@action=redirect&bpo=29654].)

`SimpleHTTPRequestHandler` accepts the new *directory* argument, in addition to the new `--directory` command line argument. With this parameter, the server serves the specified directory, by default it uses the current working directory. (Contributed by Stéphane Wirtel and Julien Palard in bpo-28707 [https://bugs.python.org/issue?@action=redirect&bpo=28707].)

The new **`ThreadingHTTPServer`** class uses threads to handle requests using **`ThreadingMixIn`**. It is used when `http.server` is run with `-m`. (Contributed by Julien Palard in bpo-31639 [https://bugs.python.org/issue?@action=redirect&bpo=31639].)

idlelib and IDLE

Multiple fixes for autocompletion. (Contributed by Louie Lu in bpo-15786 [https://bugs.python.org/issue?@action=redirect&bpo=15786].)

Module Browser (on the File menu, formerly called Class Browser), now displays nested functions and classes in addition to top-level functions and classes. (Contributed by Guilherme Polo, Cheryl Sabella, and Terry Jan Reedy in bpo-1612262 [https://bugs.python.org/issue?@action=redirect&bpo=1612262].)

The Settings dialog (Options, Configure IDLE) has been partly rewritten to improve both appearance and function. (Contributed by Cheryl Sabella and Terry Jan Reedy in multiple issues.)

The font sample now includes a selection of non-Latin characters so that users can better see the effect of selecting a particular font. (Contributed by Terry Jan Reedy in bpo-13802 [https://bugs.python.org/issue?@action=redirect&bpo=13802].) The sample can be edited to include other characters. (Contributed by Serhiy Storchaka in bpo-31860 [https://bugs.python.org/issue?@action=redirect&bpo=31860].)

The IDLE features formerly implemented as extensions have been reimplemented as normal features. Their settings have been moved from the Extensions tab to other dialog tabs. (Contributed by Charles Wohlganger and Terry Jan Reedy in bpo-27099 [https://bugs.python.org/issue?@action=redirect&bpo=27099].)

Editor code context option revised. Box displays all context lines up to maxlines. Clicking on a context line jumps the editor to that line. Context colors for custom themes is added to Highlights tab of Settings dialog. (Contributed by Cheryl Sabella and Terry Jan Reedy in bpo-33642 [https://bugs.python.org/issue?@action=redirect&bpo=33642], bpo-33768 [https://bugs.python.org/issue?@action=redirect&bpo=33768], and bpo-33679 [https://bugs.python.org/issue?@action=redirect&bpo=33679].)

On Windows, a new API call tells Windows that tk scales for DPI. On Windows 8.1+ or 10, with DPI compatibility properties of the Python binary unchanged, and a monitor resolution greater than 96 DPI, this should make text and lines sharper. It should otherwise have no effect. (Contributed by Terry Jan Reedy in bpo-33656 [https://bugs.python.org/issue?@action=redirect&bpo=33656].)

New in 3.7.1:

Output over N lines (50 by default) is squeezed down to a button. N can be changed in the PyShell section of the General page of the Settings dialog. Fewer, but possibly extra long, lines can be squeezed by right clicking on the output. Squeezed output can be expanded in place by double-clicking the button or into the clipboard or a separate window by right-clicking the button. (Contributed by Tal Einat in bpo-1529353 [https://bugs.python.org/issue?@action=redirect&bpo=1529353].)

The changes above have been backported to 3.6 maintenance releases.

NEW in 3.7.4:

Add “Run Customized” to the Run menu to run a module with customized settings. Any command line arguments entered are added to sys.argv. They re-appear in the box for the next customized run. One can also suppress the normal Shell main module restart. (Contributed by Cheryl Sabella, Terry Jan Reedy, and others in bpo-5680 [https://bugs.python.org/issue?@action=redirect&bpo=5680] and bpo-37627 [https://bugs.python.org/issue?@action=redirect&bpo=37627].)

New in 3.7.5:

Add optional line numbers for IDLE editor windows. Windows open without line numbers unless set otherwise in the General tab of the configuration dialog. Line numbers for an existing window are shown and hidden in the Options menu. (Contributed by Tal Einat and Saimadhav Heblikar in bpo-17535 [https://bugs.python.org/issue?@action=redirect&bpo=17535].)

importlib

The **`importlib.abc.ResourceReader`** ABC was introduced to support the loading of resources from packages. See also importlib.resources. (Contributed by Barry Warsaw, Brett Cannon in bpo-32248 [https://bugs.python.org/issue?@action=redirect&bpo=32248].)

`importlib.reload()` now raises **`ModuleNotFoundError`** if the module lacks a spec. (Contributed by Garvit Khatri in bpo-29851 [https://bugs.python.org/issue?@action=redirect&bpo=29851].)

`importlib.util.find_spec()` now raises **`ModuleNotFoundError`** instead of **`AttributeError`** if the specified parent module is not a package (i.e. lacks a `__path__` attribute). (Contributed by Milan Oberkirch in bpo-30436 [https://bugs.python.org/issue?@action=redirect&bpo=30436].)

The new **`importlib.util.source_hash()`** can be used to compute the hash of the passed source. A hash-based .pyc file embeds the value returned by this function.

io

The new **`TextIOWrapper.reconfigure()`** method can be used to reconfigure the text stream with the new settings. (Contributed by Antoine Pitrou in bpo-30526 [https://bugs.python.org/issue?@action=redirect&bpo=30526] and INADA Naoki in bpo-15216 [https://bugs.python.org/issue?@action=redirect&bpo=15216].)

ipaddress

The new `subnet_of()` and `supernet_of()` methods of **`ipaddress.IPv6Network`** and **`ipaddress.IPv4Network`** can be used for network containment tests. (Contributed by Michel Albert and Cheryl Sabella in bpo-20825 [https://bugs.python.org/issue?@action=redirect&bpo=20825].)

itertools

`itertools.islice()` now accepts `integer-like objects` as start, stop, and slice arguments. (Contributed by Will Roberts in bpo-30537 [https://bugs.python.org/issue?@action=redirect&bpo=30537].)

locale

The new *monetary* argument to `locale.format_string()` can be used to make the conversion use monetary thousands separators and grouping strings. (Contributed by Garvit in bpo-10379 [https://bugs.python.org/issue?@action=redirect&bpo=10379].)

The `locale.getpreferredencoding()` function now always returns `'UTF-8'` on Android or when in the forced UTF-8 mode.

logging

`Logger` instances can now be pickled. (Contributed by Vinay Sajip in bpo-30520 [https://bugs.python.org/issue?@action=redirect&bpo=30520].)

The new `StreamHandler.setStream()` method can be used to replace the logger stream after handler creation. (Contributed by Vinay Sajip in bpo-30522 [https://bugs.python.org/issue?@action=redirect&bpo=30522].)

It is now possible to specify keyword arguments to handler constructors in configuration passed to `logging.config.fileConfig()`. (Contributed by Preston Landers in bpo-31080 [https://bugs.python.org/issue?@action=redirect&bpo=31080].)

math

The new **`math.remainder()`** function implements the IEEE 754-style remainder operation. (Contributed by Mark Dickinson in bpo-29962 [https://bugs.python.org/issue?@action=redirect&bpo=29962].)

mimetypes

The MIME type of .bmp has been changed from `'image/x-ms-bmp'` to `'image/bmp'`. (Contributed by Nitish Chandra in bpo-22589 [https://bugs.python.org/issue?@action=redirect&bpo=22589].)

msilib

The new **`Database.Close()`** method can be used to close the MSI database. (Contributed by Berker Peksag in bpo-20486 [https://bugs.python.org/issue?@action=redirect&bpo=20486].)

multiprocessing

The new **`Process.close()`** method explicitly closes the process object and releases all resources associated with it. **`ValueError`** is raised if the underlying process is still running. (Contributed by Antoine Pitrou in bpo-30596 [https://bugs.python.org/issue?@action=redirect&bpo=30596].)

The new **`Process.kill()`** method can be used to terminate the process using the **`SIGKILL`** signal on Unix. (Contributed by Vitor Pereira in bpo-30794 [https://bugs.python.org/issue?@action=redirect&bpo=30794].)

Non-daemonic threads created by **`Process`** are now joined on process exit. (Contributed by Antoine Pitrou in bpo-18966 [https://bugs.python.org/issue?@action=redirect&bpo=18966].)

os

`os.fwalk()` now accepts the *path* argument as `bytes`. (Contributed by Serhiy Storchaka in bpo-28682 [https://bugs.python.org/issue?@action=redirect&bpo=28682].)

`os.scandir()` gained support for file descriptors. (Contributed by Serhiy Storchaka in bpo-25996 [https://bugs.python.org/issue?@action=redirect&bpo=25996].)

The new `register_at_fork()` function allows registering Python callbacks to be executed at process fork. (Contributed by Antoine Pitrou in bpo-16500 [https://bugs.python.org/issue?@action=redirect&bpo=16500].)

Added `os.preadv()` (combine the functionality of `os.readv()` and `os.pread()`) and `os.pwritev()` functions (combine the functionality of `os.writev()` and `os.pwrite()`). (Contributed by Pablo Galindo in bpo-31368 [https://bugs.python.org/issue?@action=redirect&bpo=31368].)

The mode argument of `os.makedirs()` no longer affects the file permission bits of newly created intermediate-level directories. (Contributed by Serhiy Storchaka in bpo-19930 [https://bugs.python.org/issue?@action=redirect&bpo=19930].)

`os.dup2()` now returns the new file descriptor. Previously, `None` was always returned. (Contributed by Benjamin Peterson in bpo-32441 [https://bugs.python.org/issue?@action=redirect&bpo=32441].)

The structure returned by `os.stat()` now contains the `st_fstype` attribute on Solaris and its derivatives. (Contributed by Jesús Cea Avión in bpo-32659 [https://bugs.python.org/issue?@action=redirect&bpo=32659].)

pathlib

The new `Path.is_mount()` method is now available on POSIX systems and can be used to determine whether a path is a mount point. (Contributed by Cooper Ry Lees in bpo-30897 [https://bugs.python.org/issue?@action=redirect&bpo=30897].)

pdb

`pdb.set_trace()` now takes an optional *header* keyword-only argument. If given, it is printed to the console just before debugging begins. (Contributed by Barry Warsaw in bpo-31389 [https://bugs.python.org/issue?@action=redirect&bpo=31389].)

`pdb` command line now accepts `-m module_name` as an alternative to script file. (Contributed by Mario Corchero in bpo-32206 [https://bugs.python.org/issue?@action=redirect&bpo=32206].)

py_compile

`py_compile.compile()` – and by extension, **`compileall`** – now respects the **`SOURCE_DATE_EPOCH`** environment variable by unconditionally creating `.pyc` files for hash-based validation. This allows for guaranteeing reproducible builds [https://reproducible-builds.org/] of `.pyc` files when they are created eagerly. (Contributed by Bernhard M. Wiedemann in bpo-29708 [https://bugs.python.org/issue?@action=redirect&bpo=29708].)

pydoc

The pydoc server can now bind to an arbitrary hostname specified by the new `-n` command-line argument. (Contributed by Feanil Patel in bpo-31128 [https://bugs.python.org/issue?@action=redirect&bpo=31128].)

queue

The new **`SimpleQueue`** class is an unbounded FIFO queue. (Contributed by Antoine Pitrou in bpo-14976 [https://bugs.python.org/issue?@action=redirect&bpo=14976].)

re

The flags **`re.ASCII`**, **`re.LOCALE`** and **`re.UNICODE`** can be set within the scope of a group. (Contributed by Serhiy Storchaka in bpo-31690 [https://bugs.python.org/issue?@action=redirect&bpo=31690].)

`re.split()` now supports splitting on a pattern like `r'\b'`, `'^$'` or `(?=-)` that matches an empty string. (Contributed by Serhiy Storchaka in bpo-25054 [https://bugs.python.org/issue?@action=redirect&bpo=25054].)

Regular expressions compiled with the **`re.LOCALE`** flag no longer depend on the locale at compile time. Locale settings are applied only when the compiled regular expression is used. (Contributed by Serhiy Storchaka in bpo-30215 [https://bugs.python.org/issue?@action=redirect&bpo=30215].)

`FutureWarning` is now emitted if a regular expression contains character set constructs that will change semantically in the future, such as nested sets and set operations. (Contributed by Serhiy Storchaka in bpo-30349 [https://bugs.python.org/issue?@action=redirect&bpo=30349].)

Compiled regular expression and match objects can now be copied using **`copy.copy()`** and **`copy.deepcopy()`**. (Contributed by Serhiy Storchaka in bpo-10076 [https://bugs.python.org/issue?@action=redirect&bpo=10076].)

signal

The new *warn_on_full_buffer* argument to the **`signal.set_wakeup_fd()`** function makes it possible to specify whether Python prints a warning on stderr when the wakeup buffer overflows. (Contributed by Nathaniel J. Smith in bpo-30050 [https://bugs.python.org/issue?@action=redirect&bpo=30050].)

socket

The new `socket.getblocking()` method returns `True` if the socket is in blocking mode and `False` otherwise. (Contributed by Yury Selivanov in bpo-32373 [https://bugs.python.org/issue?@action=redirect&bpo=32373].)

The new `socket.close()` function closes the passed socket file descriptor. This function should be used instead of `os.close()` for better compatibility across platforms. (Contributed by Christian Heimes in bpo-32454 [https://bugs.python.org/issue?@action=redirect&bpo=32454].)

The `socket` module now exposes the socket.TCP_CONGESTION (Linux 2.6.13), socket.TCP_USER_TIMEOUT (Linux 2.6.37), and socket.TCP_NOTSENT_LOWAT (Linux 3.12) constants. (Contributed by Omar Sandoval in bpo-26273 [https://bugs.python.org/issue?@action=redirect&bpo=26273] and Nathaniel J. Smith in bpo-29728 [https://bugs.python.org/issue?@action=redirect&bpo=29728].)

Support for `socket.AF_VSOCK` sockets has been added to allow communication between virtual machines and their hosts. (Contributed by Cathy Avery in bpo-27584 [https://bugs.python.org/issue?@action=redirect&bpo=27584].)

Sockets now auto-detect family, type and protocol from file descriptor by default. (Contributed by Christian Heimes in bpo-28134 [https://bugs.python.org/issue?@action=redirect&bpo=28134].)

socketserver

`socketserver.ThreadingMixIn.server_close` now waits until all non-daemon threads complete.

`socketserver.ForkingMixIn.server_close` now waits until all child processes complete.

Add a new `socketserver.ForkingMixIn.block_on_close` class attribute to `socketserver.ForkingMixIn` and `socketserver.ThreadingMixIn` classes. Set the class attribute to `False` to get the pre-3.7 behaviour.

sqlite3

`sqlite3.Connection` now exposes the `backup()` method when the underlying SQLite library is at version 3.6.11 or higher. (Contributed by Lele Gaifax in bpo-27645 [https://bugs.python.org/issue?@action=redirect&bpo=27645].)

The *database* argument of `sqlite3.connect()` now accepts any path-like object, instead of just a string. (Contributed by Anders Lorentsen in bpo-31843 [https://bugs.python.org/issue?@action=redirect&bpo=31843].)

ssl

The `ssl` module now uses OpenSSL's builtin API instead of **`match_hostname()`** to check a host name or an IP address. Values are validated during TLS handshake. Any certificate validation error including failing the host name check now raises `SSLCertVerificationError` and aborts the handshake with a proper TLS Alert message. The new exception contains additional information. Host name validation can be customized with `SSLContext.hostname_checks_common_name`. (Contributed by Christian Heimes in bpo-31399 [https://bugs.python.org/issue?@action=redirect&bpo=31399].)

Note

The improved host name check requires a *libssl* implementation compatible with OpenSSL 1.0.2 or 1.1. Consequently, OpenSSL 0.9.8 and 1.0.1 are no longer supported (see Platform Support Removals for more details). The ssl module is mostly compatible with LibreSSL 2.7.2 and newer.

The `ssl` module no longer sends IP addresses in SNI TLS extension. (Contributed by Christian Heimes in bpo-32185 [https://bugs.python.org/issue?@action=redirect&bpo=32185].)

`match_hostname()` no longer supports partial wildcards like `www*.example.org`. (Contributed by Mandeep Singh in bpo-23033 [https://bugs.python.org/issue?@action=redirect&bpo=23033] and Christian Heimes in bpo-31399 [https://bugs.python.org/issue?@action=redirect&bpo=31399].)

The default cipher suite selection of the `ssl` module now uses a blacklist approach rather than a hard-coded whitelist. Python no longer re-enables ciphers that have been blocked by OpenSSL security updates. Default cipher suite selection can be configured at compile time. (Contributed by Christian Heimes in bpo-31429 [https://bugs.python.org/issue?@action=redirect&bpo=31429].)

Validation of server certificates containing internationalized domain names (IDNs) is now supported. As part of this change, the **`SSLSocket.server_hostname`** attribute now stores the expected hostname in A-label form (`"xn--pythn-mua.org"`), rather than the U-label form (`"pythön.org"`). (Contributed by Nathaniel J. Smith and Christian Heimes in bpo-28414 [https://bugs.python.org/issue?@action=redirect&bpo=28414].)

The `ssl` module has preliminary and experimental support for TLS 1.3 and OpenSSL 1.1.1. At the time of Python 3.7.0

release, OpenSSL 1.1.1 is still under development and TLS 1.3 hasn't been finalized yet. The TLS 1.3 handshake and protocol behaves slightly differently than TLS 1.2 and earlier, see TLS 1.3. (Contributed by Christian Heimes in bpo-32947 [https://bugs.python.org/issue?@action=redirect&bpo=32947], bpo-20995 [https://bugs.python.org/issue?@action=redirect&bpo=20995], bpo-29136 [https://bugs.python.org/issue?@action=redirect&bpo=29136], bpo-30622 [https://bugs.python.org/issue?@action=redirect&bpo=30622] and bpo-33618 [https://bugs.python.org/issue?@action=redirect&bpo=33618])

`SSLSocket` and `SSLObject` no longer have a public constructor. Direct instantiation was never a documented and supported feature. Instances must be created with `SSLContext` methods `wrap_socket()` and `wrap_bio()`. (Contributed by Christian Heimes in bpo-32951 [https://bugs.python.org/issue?@action=redirect&bpo=32951])

OpenSSL 1.1 APIs for setting the minimum and maximum TLS protocol version are available as `SSLContext.minimum_version` and `SSLContext.maximum_version`. Supported protocols are indicated by several new flags, such as `HAS_TLSv1_1`. (Contributed by Christian Heimes in bpo-32609 [https://bugs.python.org/issue?@action=redirect&bpo=32609].)

Added `ssl.SSLContext.post_handshake_auth` to enable and `ssl.SSLSocket.verify_client_post_handshake()` to initiate TLS 1.3 post-handshake authentication. (Contributed by Christian Heimes in gh-78851 [https://github.com/python/cpython/issues/78851].)

string

`string.Template` now lets you to optionally modify the regular expression pattern for braced placeholders and non-braced placeholders separately. (Contributed by Barry Warsaw in

bpo-1198569 [https://bugs.python.org/issue?@action=redirect&bpo=1198569].)

subprocess

The **`subprocess.run()`** function accepts the new *capture_output* keyword argument. When true, stdout and stderr will be captured. This is equivalent to passing **`subprocess.PIPE`** as *stdout* and *stderr* arguments. (Contributed by Bo Bayles in bpo-32102 [https://bugs.python.org/issue?@action=redirect&bpo=32102].)

The `subprocess.run` function and the **`subprocess.Popen`** constructor now accept the *text* keyword argument as an alias to *universal_newlines*. (Contributed by Andrew Clegg in bpo-31756 [https://bugs.python.org/issue?@action=redirect&bpo=31756].)

On Windows the default for *close_fds* was changed from `False` to `True` when redirecting the standard handles. It's now possible to set *close_fds* to true when redirecting the standard handles. See **`subprocess.Popen`**. This means that *close_fds* now defaults to `True` on all supported platforms. (Contributed by Segev Finer in bpo-19764 [https://bugs.python.org/issue?@action=redirect&bpo=19764].)

The subprocess module is now more graceful when handling **`KeyboardInterrupt`** during **`subprocess.call()`**, **`subprocess.run()`**, or in a **`Popen`** context manager. It now waits a short amount of time for the child to exit, before continuing the handling of the `KeyboardInterrupt` exception. (Contributed by Gregory P. Smith in bpo-25942 [https://bugs.python.org/issue?@action=redirect&bpo=25942].)

sys

The new `sys.breakpointhook()` hook function is called by the built-in `breakpoint()`. (Contributed by Barry Warsaw in bpo-31353 [https://bugs.python.org/issue?@action=redirect&bpo=31353].)

On Android, the new `sys.getandroidapilevel()` returns the build-time Android API version. (Contributed by Victor Stinner in bpo-28740 [https://bugs.python.org/issue?@action=redirect&bpo=28740].)

The new `sys.get_coroutine_origin_tracking_depth()` function returns the current coroutine origin tracking depth, as set by the new `sys.set_coroutine_origin_tracking_depth()`. `asyncio` has been converted to use this new API instead of the deprecated `sys.set_coroutine_wrapper()`. (Contributed by Nathaniel J. Smith in bpo-32591 [https://bugs.python.org/issue?@action=redirect&bpo=32591].)

time

PEP 564 [https://peps.python.org/pep-0564/] adds six new functions with nanosecond resolution to the `time` module:

- `time.clock_gettime_ns()`
- `time.clock_settime_ns()`
- `time.monotonic_ns()`
- `time.perf_counter_ns()`
- `time.process_time_ns()`
- `time.time_ns()`

New clock identifiers have been added:

- `time.CLOCK_BOOTTIME` (Linux): Identical to `time.CLOCK_MONOTONIC`, except it also includes any time that the system is suspended.
- `time.CLOCK_PROF` (FreeBSD, NetBSD and OpenBSD): High-resolution per-process CPU timer.

- **`time.CLOCK_UPTIME`** (FreeBSD, OpenBSD): Time whose absolute value is the time the system has been running and not suspended, providing accurate uptime measurement.

The new **`time.thread_time()`** and **`time.thread_time_ns()`** functions can be used to get per-thread CPU time measurements. (Contributed by Antoine Pitrou in bpo-32025 [https://bugs.python.org/issue?@action=redirect&bpo=32025].)

The new **`time.pthread_getcpuclockid()`** function returns the clock ID of the thread-specific CPU-time clock.

tkinter

The new **`tkinter.ttk.Spinbox`** class is now available. (Contributed by Alan Moore in bpo-32585 [https://bugs.python.org/issue?@action=redirect&bpo=32585].)

tracemalloc

`tracemalloc.Traceback` behaves more like regular tracebacks, sorting the frames from oldest to most recent. **`Traceback.format()`** now accepts negative *limit*, truncating the result to the `abs(limit)` oldest frames. To get the old behaviour, use the new *most_recent_first* argument to `Traceback.format()`. (Contributed by Jesse Bakker in bpo-32121 [https://bugs.python.org/issue?@action=redirect&bpo=32121].)

types

The new **`WrapperDescriptorType`**, **`MethodWrapperType`**, **`MethodDescriptorType`**, and **`ClassMethodDescriptorType`** classes are now available. (Contributed by Manuel Krebber and Guido van Rossum in bpo-29377 [https://bugs.python.org/issue?

@action=redirect&bpo=29377], and Serhiy Storchaka in bpo-32265 [https://bugs.python.org/issue?@action=redirect&bpo=32265].)

The new `types.resolve_bases()` function resolves MRO entries dynamically as specified by **PEP 560** [https://peps.python.org/pep-0560/]. (Contributed by Ivan Levkivskyi in bpo-32717 [https://bugs.python.org/issue?@action=redirect&bpo=32717].)

unicodedata

The internal `unicodedata` database has been upgraded to use Unicode 11 [https://www.unicode.org/versions/Unicode11.0.0/]. (Contributed by Benjamin Peterson.)

unittest

The new `-k` command-line option allows filtering tests by a name substring or a Unix shell-like pattern. For example, `python -m unittest -k foo` runs `foo_tests.SomeTest.test_something`, `bar_tests.SomeTest.test_foo`, but not `bar_tests.FooTest.test_something`. (Contributed by Jonas Haag in bpo-32071 [https://bugs.python.org/issue?@action=redirect&bpo=32071].)

unittest.mock

The `sentinel` attributes now preserve their identity when they are `copied` or `pickled`. (Contributed by Serhiy Storchaka in bpo-20804 [https://bugs.python.org/issue?@action=redirect&bpo=20804].)

The new `seal()` function allows sealing `Mock` instances, which will disallow further creation of attribute mocks. The seal is applied recursively to all attributes that are themselves mocks. (Contributed by Mario Corchero in bpo-30541 [https://bugs.python.org/issue?@action=redirect&bpo=30541].)

urllib.parse

`urllib.parse.quote()` has been updated from **RFC 2396** [https://datatracker.ietf.org/doc/html/rfc2396.html] to **RFC 3986** [https://datatracker.ietf.org/doc/html/rfc3986.html], adding `~` to the set of characters that are never quoted by default. (Contributed by Christian Theune and Ratnadeep Debnath in bpo-16285 [https://bugs.python.org/issue?@action=redirect&bpo=16285].)

uu

The `uu.encode()` function now accepts an optional *backtick* keyword argument. When it's true, zeros are represented by ``'`'`` instead of spaces. (Contributed by Xiang Zhang in bpo-30103 [https://bugs.python.org/issue?@action=redirect&bpo=30103].)

uuid

The new `UUID.is_safe` attribute relays information from the platform about whether generated UUIDs are generated with a multiprocessing-safe method. (Contributed by Barry Warsaw in bpo-22807 [https://bugs.python.org/issue?@action=redirect&bpo=22807].)

`uuid.getnode()` now prefers universally administered MAC addresses over locally administered MAC addresses. This makes a better guarantee for global uniqueness of UUIDs returned from `uuid.uuid1()`. If only locally administered MAC addresses are available, the first such one found is returned. (Contributed by Barry Warsaw in bpo-32107 [https://bugs.python.org/issue?@action=redirect&bpo=32107].)

warnings

The initialization of the default warnings filters has changed as follows:

- warnings enabled via command line options (including those for `-b` and the new CPython-specific `-X` `dev` option) are always passed to the warnings machinery via the `sys.warnoptions` attribute.
- warnings filters enabled via the command line or the environment now have the following order of precedence:
  - the `BytesWarning` filter for `-b` (or `-bb`)
  - any filters specified with the `-W` option
  - any filters specified with the `PYTHONWARNINGS` environment variable
  - any other CPython specific filters (e.g. the `default` filter added for the new `-X dev` mode)
  - any implicit filters defined directly by the warnings machinery
- in CPython debug builds, all warnings are now displayed by default (the implicit filter list is empty)

(Contributed by Nick Coghlan and Victor Stinner in bpo-20361 [https://bugs.python.org/issue?@action=redirect&bpo=20361], bpo-32043 [https://bugs.python.org/issue?@action=redirect&bpo=32043], and bpo-32230 [https://bugs.python.org/issue?@action=redirect&bpo=32230].)

Deprecation warnings are once again shown by default in single-file scripts and at the interactive prompt. See PEP 565: Show DeprecationWarning in __main__ for details. (Contributed by Nick Coghlan in bpo-31975 [https://bugs.python.org/issue?@action=redirect&bpo=31975].)

xml

As mitigation against DTD and external entity retrieval, the `xml.dom.minidom` and `xml.sax` modules no longer process external entities by default. (Contributed by Christian Heimes in gh-61441 [https://github.com/python/cpython/issues/61441].)

xml.etree

ElementPath predicates in the **`find()`** methods can now compare text of the current node with `[. = "text"]`, not only text in children. Predicates also allow adding spaces for better readability. (Contributed by Stefan Behnel in bpo-31648 [https://bugs.python.org/issue?@action=redirect&bpo=31648].)

xmlrpc.server

`SimpleXMLRPCDispatcher.register_function()` can now be used as a decorator. (Contributed by Xiang Zhang in bpo-7769 [https://bugs.python.org/issue?@action=redirect&bpo=7769].)

zipapp

Function **`create_archive()`** now accepts an optional *filter* argument to allow the user to select which files should be included in the archive. (Contributed by Irmen de Jong in bpo-31072 [https://bugs.python.org/issue?@action=redirect&bpo=31072].)

Function **`create_archive()`** now accepts an optional *compressed* argument to generate a compressed archive. A command line option `--compress` has also been added to support compression. (Contributed by Zhiming Wang in bpo-31638 [https://bugs.python.org/issue?@action=redirect&bpo=31638].)

zipfile

`ZipFile` now accepts the new *compresslevel* parameter to control the compression level. (Contributed by Bo Bayles in bpo-21417 [https://bugs.python.org/issue?@action=redirect&bpo=21417].)

Subdirectories in archives created by `ZipFile` are now stored in alphabetical order. (Contributed by Bernhard M.

Wiedemann in bpo-30693 [https://bugs.python.org/issue?@action=redirect&bpo=30693].)

C API Changes

A new API for thread-local storage has been implemented. See PEP 539: New C API for Thread-Local Storage for an overview and Thread Specific Storage (TSS) API for a complete reference. (Contributed by Masayuki Yamamoto in bpo-25658 [https://bugs.python.org/issue?@action=redirect&bpo=25658].)

The new context variables functionality exposes a number of new C APIs.

The new `PyImport_GetModule()` function returns the previously imported module with the given name. (Contributed by Eric Snow in bpo-28411 [https://bugs.python.org/issue?@action=redirect&bpo=28411].)

The new `Py_RETURN_RICHCOMPARE` macro eases writing rich comparison functions. (Contributed by Petr Victorin in bpo-23699 [https://bugs.python.org/issue?@action=redirect&bpo=23699].)

The new `Py_UNREACHABLE` macro can be used to mark unreachable code paths. (Contributed by Barry Warsaw in bpo-31338 [https://bugs.python.org/issue?@action=redirect&bpo=31338].)

The `tracemalloc` now exposes a C API through the new `PyTraceMalloc_Track()` and `PyTraceMalloc_Untrack()` functions. (Contributed by Victor Stinner in bpo-30054 [https://bugs.python.org/issue?@action=redirect&bpo=30054].)

The new import__find__load__start and import__find__load__done static markers can be used to trace module imports. (Contributed by Christian Heimes in bpo-31574 [https://bugs.python.org/issue?@action=redirect&bpo=31574].)

The fields **`name`** and **`doc`** of structures `PyMemberDef`, `PyGetSetDef`, `PyStructSequence_Field`, `PyStructSequence_Desc`, and **`wrapperbase`** are now of type `const char *` rather of `char *`. (Contributed by Serhiy Storchaka in bpo-28761 [https://bugs.python.org/issue?@action=redirect&bpo=28761].)

The result of `PyUnicode_AsUTF8AndSize()` and `PyUnicode_AsUTF8()` is now of type `const char *` rather of `char *`. (Contributed by Serhiy Storchaka in bpo-28769 [https://bugs.python.org/issue?@action=redirect&bpo=28769].)

The result of `PyMapping_Keys()`, `PyMapping_Values()` and `PyMapping_Items()` is now always a list, rather than a list or a tuple. (Contributed by Oren Milman in bpo-28280 [https://bugs.python.org/issue?@action=redirect&bpo=28280].)

Added functions `PySlice_Unpack()` and `PySlice_AdjustIndices()`. (Contributed by Serhiy Storchaka in bpo-27867 [https://bugs.python.org/issue?@action=redirect&bpo=27867].)

`PyOS_AfterFork()` is deprecated in favour of the new functions `PyOS_BeforeFork()`, `PyOS_AfterFork_Parent()` and `PyOS_AfterFork_Child()`. (Contributed by Antoine Pitrou in bpo-16500 [https://bugs.python.org/issue?@action=redirect&bpo=16500].)

The `PyExc_RecursionErrorInst` singleton that was part of the public API has been removed as its members being never cleared may cause a segfault during finalization of the interpreter. Contributed by Xavier de Gaye in bpo-22898 [https://bugs.python.org/issue?@action=redirect&bpo=22898] and bpo-30697 [https://bugs.python.org/issue?@action=redirect&bpo=30697].

Added C API support for timezones with timezone constructors `PyTimeZone_FromOffset()` and `PyTimeZone_FromOffsetAndName()`, and access to the UTC singleton with `PyDateTime_TimeZone_UTC`. Contributed by Paul

Ganssle in bpo-10381 [https://bugs.python.org/issue?@action=redirect&bpo=10381].

The type of results of **`PyThread_start_new_thread()`** and **`PyThread_get_thread_ident()`**, and the *id* parameter of **`PyThreadState_SetAsyncExc()`** changed from long to unsigned long. (Contributed by Serhiy Storchaka in bpo-6532 [https://bugs.python.org/issue?@action=redirect&bpo=6532].)

`PyUnicode_AsWideCharString()` now raises a **`ValueError`** if the second argument is `NULL` and the wchar_t* string contains null characters. (Contributed by Serhiy Storchaka in bpo-30708 [https://bugs.python.org/issue?@action=redirect&bpo=30708].)

Changes to the startup sequence and the management of dynamic memory allocators mean that the long documented requirement to call **`Py_Initialize()`** before calling most C API functions is now relied on more heavily, and failing to abide by it may lead to segfaults in embedding applications. See the Porting to Python 3.7 section in this document and the Before Python Initialization section in the C API documentation for more details.

The new **`PyInterpreterState_GetID()`** returns the unique ID for a given interpreter. (Contributed by Eric Snow in bpo-29102 [https://bugs.python.org/issue?@action=redirect&bpo=29102].)

`Py_DecodeLocale()`, **`Py_EncodeLocale()`** now use the UTF-8 encoding when the UTF-8 mode is enabled. (Contributed by Victor Stinner in bpo-29240 [https://bugs.python.org/issue?@action=redirect&bpo=29240].)

`PyUnicode_DecodeLocaleAndSize()` and **`PyUnicode_EncodeLocale()`** now use the current locale encoding for `surrogateescape` error handler. (Contributed by Victor Stinner in bpo-29240 [https://bugs.python.org/issue?@action=redirect&bpo=29240].)

The *start* and *end* parameters of **`PyUnicode_FindChar()`** are now adjusted to behave like string slices. (Contributed by Xiang Zhang in bpo-28822 [https://bugs.python.org/issue?@action=redirect&bpo=28822].)

Build Changes

Support for building `--without-threads` has been removed. The **`threading`** module is now always available. (Contributed by Antoine Pitrou in bpo-31370 [https://bugs.python.org/issue?@action=redirect&bpo=31370].).

A full copy of libffi is no longer bundled for use when building the **`_ctypes`** module on non-OSX UNIX platforms. An installed copy of libffi is now required when building `_ctypes` on such platforms. (Contributed by Zachary Ware in bpo-27979 [https://bugs.python.org/issue?@action=redirect&bpo=27979].)

The Windows build process no longer depends on Subversion to pull in external sources, a Python script is used to download zipfiles from GitHub instead. If Python 3.6 is not found on the system (via `py -3.6`), NuGet is used to download a copy of 32-bit Python for this purpose. (Contributed by Zachary Ware in bpo-30450 [https://bugs.python.org/issue?@action=redirect&bpo=30450].)

The **`ssl`** module requires OpenSSL 1.0.2 or 1.1 compatible libssl. OpenSSL 1.0.1 has reached end of lifetime on 2016-12-31 and is no longer supported. LibreSSL is temporarily not supported as well. LibreSSL releases up to version 2.6.4 are missing required OpenSSL 1.0.2 APIs.

Optimizations

The overhead of calling many methods of various standard library classes implemented in C has been significantly reduced by porting more code to use the `METH_FASTCALL` convention. (Contributed by Victor Stinner in bpo-29300 [https://bugs.python.org/issue?@action=redirect&bpo=29300], bpo-29507 [https://bugs.python.org/issue?@action=redirect&bpo=29507], bpo-29452 [https://bugs.python.org/issue?@action=redirect&bpo=29452], and bpo-29286 [https://bugs.python.org/issue?@action=redirect&bpo=29286].)

Various optimizations have reduced Python startup time by 10% on Linux and up to 30% on macOS. (Contributed by Victor Stinner, INADA Naoki in bpo-29585 [https://bugs.python.org/issue?@action=redirect&bpo=29585], and Ivan Levkivskyi in bpo-31333 [https://bugs.python.org/issue?@action=redirect&bpo=31333].)

Method calls are now up to 20% faster due to the bytecode changes which avoid creating bound method instances. (Contributed by Yury Selivanov and INADA Naoki in bpo-26110 [https://bugs.python.org/issue?@action=redirect&bpo=26110].)

The **`asyncio`** module received a number of notable optimizations for commonly used functions:

- The **`asyncio.get_event_loop()`** function has been reimplemented in C to make it up to 15 times faster. (Contributed by Yury Selivanov in bpo-32296 [https://bugs.python.org/issue?@action=redirect&bpo=32296].)
- **`asyncio.Future`** callback management has been optimized. (Contributed by Yury Selivanov in bpo-32348 [https://bugs.python.org/issue?@action=redirect&bpo=32348].)
- **`asyncio.gather()`** is now up to 15% faster. (Contributed by Yury Selivanov in bpo-32355 [https://bugs.python.org/issue?@action=redirect&bpo=32355].)
- **`asyncio.sleep()`** is now up to 2 times faster when the *delay* argument is zero or negative. (Contributed by

  Andrew Svetlov in bpo-32351 [https://bugs.python.org/issue?@action=redirect&bpo=32351].)
- The performance overhead of asyncio debug mode has been reduced. (Contributed by Antoine Pitrou in bpo-31970 [https://bugs.python.org/issue?@action=redirect&bpo=31970].)

As a result of PEP 560 work, the import time of `typing` has been reduced by a factor of 7, and many typing operations are now faster. (Contributed by Ivan Levkivskyi in bpo-32226 [https://bugs.python.org/issue?@action=redirect&bpo=32226].)

`sorted()` and `list.sort()` have been optimized for common cases to be up to 40-75% faster. (Contributed by Elliot Gorokhovsky in bpo-28685 [https://bugs.python.org/issue?@action=redirect&bpo=28685].)

`dict.copy()` is now up to 5.5 times faster. (Contributed by Yury Selivanov in bpo-31179 [https://bugs.python.org/issue?@action=redirect&bpo=31179].)

`hasattr()` and `getattr()` are now about 4 times faster when *name* is not found and *obj* does not override `object.__getattr__()` or `object.__getattribute__()`. (Contributed by INADA Naoki in bpo-32544 [https://bugs.python.org/issue?@action=redirect&bpo=32544].)

Searching for certain Unicode characters (like Ukrainian capital “Є”) in a string was up to 25 times slower than searching for other characters. It is now only 3 times slower in the worst case. (Contributed by Serhiy Storchaka in bpo-24821 [https://bugs.python.org/issue?@action=redirect&bpo=24821].)

The `collections.namedtuple()` factory has been reimplemented to make the creation of named tuples 4 to 6 times faster. (Contributed by Jelle Zijlstra with further improvements by INADA Naoki, Serhiy Storchaka, and Raymond Hettinger in bpo-28638 [https://bugs.python.org/issue?@action=redirect&bpo=28638].)

`datetime.date.fromordinal()` and `datetime.date.fromtimestamp()` are now up to 30% faster in the common case. (Contributed by Paul Ganssle in bpo-32403 [https://bugs.python.org/issue?@action=redirect&bpo=32403].)

The `os.fwalk()` function is now up to 2 times faster thanks to the use of `os.scandir()`. (Contributed by Serhiy Storchaka in bpo-25996 [https://bugs.python.org/issue?@action=redirect&bpo=25996].)

The speed of the `shutil.rmtree()` function has been improved by 20–40% thanks to the use of the `os.scandir()` function. (Contributed by Serhiy Storchaka in bpo-28564 [https://bugs.python.org/issue?@action=redirect&bpo=28564].)

Optimized case-insensitive matching and searching of `regular expressions`. Searching some patterns can now be up to 20 times faster. (Contributed by Serhiy Storchaka in bpo-30285 [https://bugs.python.org/issue?@action=redirect&bpo=30285].)

`re.compile()` now converts `flags` parameter to int object if it is `RegexFlag`. It is now as fast as Python 3.5, and faster than Python 3.6 by about 10% depending on the pattern. (Contributed by INADA Naoki in bpo-31671 [https://bugs.python.org/issue?@action=redirect&bpo=31671].)

The `modify()` methods of classes `selectors.EpollSelector`, `selectors.PollSelector` and `selectors.DevpollSelector` may be around 10% faster under heavy loads. (Contributed by Giampaolo Rodola’ in bpo-30014 [https://bugs.python.org/issue?@action=redirect&bpo=30014])

Constant folding has been moved from the peephole optimizer to the new AST optimizer, which is able perform optimizations more consistently. (Contributed by Eugene Toder and INADA Naoki in bpo-29469 [https://bugs.python.org/issue?@action=redirect&bpo=29469] and bpo-11549 [https://bugs.python.org/issue?@action=redirect&bpo=11549].)

Most functions and methods in **`abc`** have been rewritten in C. This makes creation of abstract base classes, and calling **`isinstance()`** and **`issubclass()`** on them 1.5x faster. This also reduces Python start-up time by up to 10%. (Contributed by Ivan Levkivskyi and INADA Naoki in bpo-31333 [https://bugs.python.org/issue?@action=redirect&bpo=31333])

Significant speed improvements to alternate constructors for **`datetime.date`** and **`datetime.datetime`** by using fast-path constructors when not constructing subclasses. (Contributed by Paul Ganssle in bpo-32403 [https://bugs.python.org/issue?@action=redirect&bpo=32403])

The speed of comparison of **`array.array`** instances has been improved considerably in certain cases. It is now from 10x to 70x faster when comparing arrays holding values of the same integer type. (Contributed by Adrian Wielgosik in bpo-24700 [https://bugs.python.org/issue?@action=redirect&bpo=24700].)

The **`math.erf()`** and **`math.erfc()`** functions now use the (faster) C library implementation on most platforms. (Contributed by Serhiy Storchaka in bpo-26121 [https://bugs.python.org/issue?@action=redirect&bpo=26121].)

Other CPython Implementation Changes

- Trace hooks may now opt out of receiving the `line` and opt into receiving the `opcode` events from the interpreter by setting the corresponding new **`f_trace_lines`** and **`f_trace_opcodes`** attributes on the frame being traced. (Contributed by Nick Coghlan in bpo-31344 [https://bugs.python.org/issue?@action=redirect&bpo=31344].)
- Fixed some consistency problems with namespace package module attributes. Namespace module objects

now have an `__file__` that is set to `None` (previously unset), and their `__spec__.origin` is also set to `None` (previously the string `"namespace"`). See bpo-32305 [https://bugs.python.org/issue?@action=redirect&bpo=32305]. Also, the namespace module object's `__spec__.loader` is set to the same value as `__loader__` (previously, the former was set to `None`). See bpo-32303 [https://bugs.python.org/issue?@action=redirect&bpo=32303].

- The **`locals()`** dictionary now displays in the lexical order that variables were defined. Previously, the order was undefined. (Contributed by Raymond Hettinger in bpo-32690 [https://bugs.python.org/issue?@action=redirect&bpo=32690].)
- The `distutils upload` command no longer tries to change CR end-of-line characters to CRLF. This fixes a corruption issue with sdists that ended with a byte equivalent to CR. (Contributed by Bo Bayles in bpo-32304 [https://bugs.python.org/issue?@action=redirect&bpo=32304].)

Deprecated Python Behavior

Yield expressions (both `yield` and `yield from` clauses) are now deprecated in comprehensions and generator expressions (aside from the iterable expression in the leftmost **`for`** clause). This ensures that comprehensions always immediately return a container of the appropriate type (rather than potentially returning a generator iterator object), while generator expressions won't attempt to interleave their implicit output with the output from any explicit yield expressions. In Python 3.7, such expressions emit **`DeprecationWarning`** when compiled, in Python 3.8 this will be a **`SyntaxError`**. (Contributed by Serhiy Storchaka in bpo-10544 [https://bugs.python.org/issue?@action=redirect&bpo=10544].)

Returning a subclass of **`complex`** from **`object.__complex__()`** is deprecated and will be an error in future Python versions.

This makes `__complex__()` consistent with **`object.__int__()`** and **`object.__float__()`**. (Contributed by Serhiy Storchaka in bpo-28894 [https://bugs.python.org/issue?@action=redirect&bpo=28894].)

Deprecated Python modules, functions and methods

aifc

`aifc.openfp()` has been deprecated and will be removed in Python 3.9. Use **`aifc.open()`** instead. (Contributed by Brian Curtin in bpo-31985 [https://bugs.python.org/issue?@action=redirect&bpo=31985].)

asyncio

Support for directly `await`-ing instances of **`asyncio.Lock`** and other asyncio synchronization primitives has been deprecated. An asynchronous context manager must be used in order to acquire and release the synchronization resource. (Contributed by Andrew Svetlov in bpo-32253 [https://bugs.python.org/issue?@action=redirect&bpo=32253].)

The **`asyncio.Task.current_task()`** and **`asyncio.Task.all_tasks()`** methods have been deprecated. (Contributed by Andrew Svetlov in bpo-32250 [https://bugs.python.org/issue?@action=redirect&bpo=32250].)

collections

In Python 3.8, the abstract base classes in **`collections.abc`** will no longer be exposed in the regular **`collections`** module. This will help create a clearer distinction between the concrete classes and the abstract base classes. (Contributed by Serhiy Storchaka in bpo-25988 [https://bugs.python.org/issue?@action=redirect&bpo=25988].)

dbm

`dbm.dumb` now supports reading read-only files and no longer writes the index file when it is not changed. A deprecation warning is now emitted if the index file is missing and recreated in the `'r'` and `'w'` modes (this will be an error in future Python releases). (Contributed by Serhiy Storchaka in bpo-28847 [https://bugs.python.org/issue?@action=redirect&bpo=28847].)

enum

In Python 3.8, attempting to check for non-Enum objects in **`Enum`** classes will raise a **`TypeError`** (e.g. `1 in Color`); similarly, attempting to check for non-Flag objects in a **`Flag`** member will raise **`TypeError`** (e.g. `1 in Perm.RW`); currently, both operations return **`False`** instead. (Contributed by Ethan Furman in bpo-33217 [https://bugs.python.org/issue?@action=redirect&bpo=33217].)

gettext

Using non-integer value for selecting a plural form in **`gettext`** is now deprecated. It never correctly worked. (Contributed by Serhiy Storchaka in bpo-28692 [https://bugs.python.org/issue?@action=redirect&bpo=28692].)

importlib

Methods **`MetaPathFinder.find_module()`** (replaced by **`MetaPathFinder.find_spec()`**) and **`PathEntryFinder.find_loader()`** (replaced by **`PathEntryFinder.find_spec()`**) both deprecated in Python 3.4 now emit **`DeprecationWarning`**. (Contributed by

Matthias Bussonnier in bpo-29576 [https://bugs.python.org/issue?@action=redirect&bpo=29576].)

The `importlib.abc.ResourceLoader` ABC has been deprecated in favour of `importlib.abc.ResourceReader`.

locale

`locale.format()` has been deprecated, use `locale.format_string()` instead. (Contributed by Garvit in bpo-10379 [https://bugs.python.org/issue?@action=redirect&bpo=10379].)

macpath

The `macpath` is now deprecated and will be removed in Python 3.8. (Contributed by Chi Hsuan Yen in bpo-9850 [https://bugs.python.org/issue?@action=redirect&bpo=9850].)

threading

`dummy_threading` and `_dummy_thread` have been deprecated. It is no longer possible to build Python with threading disabled. Use `threading` instead. (Contributed by Antoine Pitrou in bpo-31370 [https://bugs.python.org/issue?@action=redirect&bpo=31370].)

socket

The silent argument value truncation in `socket.htons()` and `socket.ntohs()` has been deprecated. In future versions of Python, if the passed argument is larger than 16 bits, an exception will be raised. (Contributed by Oren Milman in bpo-28332 [https://bugs.python.org/issue?@action=redirect&bpo=28332].)

ssl

`ssl.wrap_socket()` is deprecated. Use **`ssl.SSLContext.wrap_socket()`** instead. (Contributed by Christian Heimes in bpo-28124 [https://bugs.python.org/issue?@action=redirect&bpo=28124].)

sunau

`sunau.openfp()` has been deprecated and will be removed in Python 3.9. Use **`sunau.open()`** instead. (Contributed by Brian Curtin in bpo-31985 [https://bugs.python.org/issue?@action=redirect&bpo=31985].)

sys

Deprecated **`sys.set_coroutine_wrapper()`** and **`sys.get_coroutine_wrapper()`**.

The undocumented `sys.callstats()` function has been deprecated and will be removed in a future Python version. (Contributed by Victor Stinner in bpo-28799 [https://bugs.python.org/issue?@action=redirect&bpo=28799].)

wave

`wave.openfp()` has been deprecated and will be removed in Python 3.9. Use **`wave.open()`** instead. (Contributed by Brian Curtin in bpo-31985 [https://bugs.python.org/issue?@action=redirect&bpo=31985].)

Deprecated functions and types of the C API

Function **`PySlice_GetIndicesEx()`** is deprecated and replaced with a macro if `Py_LIMITED_API` is not set or set to a value in the range between `0x03050400` and `0x03060000` (not inclusive), or is `0x03060100` or higher. (Contributed by Serhiy Storchaka in bpo-27867 [https://bugs.python.org/issue?@action=redirect&bpo=27867].)

`PyOS_AfterFork()` has been deprecated. Use **`PyOS_BeforeFork()`**, **`PyOS_AfterFork_Parent()`** or **`PyOS_AfterFork_Child()`** instead. (Contributed by Antoine Pitrou in bpo-16500 [https://bugs.python.org/issue?@action=redirect&bpo=16500].)

Platform Support Removals

- FreeBSD 9 and older are no longer officially supported.
- For full Unicode support, including within extension modules, *nix platforms are now expected to provide at least one of `C.UTF-8` (full locale), `C.utf8` (full locale) or `UTF-8` (`LC_CTYPE`-only locale) as an alternative to the legacy `ASCII`-based `C` locale.
- OpenSSL 0.9.8 and 1.0.1 are no longer supported, which means building CPython 3.7 with SSL/TLS support on older platforms still using these versions requires custom build options that link to a more recent version of OpenSSL.

  Notably, this issue affects the Debian 8 (aka “jessie”) and Ubuntu 14.04 (aka “Trusty”) LTS Linux distributions, as they still use OpenSSL 1.0.1 by default.

  Debian 9 (“stretch”) and Ubuntu 16.04 (“xenial”), as well as recent releases of other LTS Linux releases (e.g. RHEL/CentOS 7.5, SLES 12-SP3), use OpenSSL 1.0.2 or later, and remain supported in the default build configuration.

CPython’s own CI configuration file [https://github.com/python/cpython/blob/v3.7.13/.travis.yml] provides an example of using the SSL compatibility testing infrastructure [https://github.com/python/cpython/tree/3.14/Tools/ssl/multissltests.py] in CPython’s test suite to build and link against OpenSSL 1.1.0 rather than an outdated system provided OpenSSL.

API and Feature Removals

The following features and APIs have been removed from Python 3.7:

- The `os.stat_float_times()` function has been removed. It was introduced in Python 2.3 for backward compatibility with Python 2.2, and was deprecated since Python 3.1.
- Unknown escapes consisting of `'\'` and an ASCII letter in replacement templates for **`re.sub()`** were deprecated in Python 3.5, and will now cause an error.
- Removed support of the *exclude* argument in **`tarfile.TarFile.add()`**. It was deprecated in Python 2.7 and 3.2. Use the *filter* argument instead.
- The **`ntpath.splitunc()`** function was deprecated in Python 3.1, and has now been removed. Use **`splitdrive()`** instead.
- **`collections.namedtuple()`** no longer supports the *verbose* parameter or `_source` attribute which showed the generated source code for the named tuple class. This was part of an optimization designed to speed-up class creation. (Contributed by Jelle Zijlstra with further improvements by INADA Naoki, Serhiy Storchaka, and Raymond Hettinger in bpo-28638 [https://bugs.python.org/issue?@action=redirect&bpo=28638].)

- Functions **`bool()`**, **`float()`**, **`list()`** and **`tuple()`** no longer take keyword arguments. The first argument of **`int()`** can now be passed only as positional argument.
- Removed previously deprecated in Python 2.4 classes `Plist`, `Dict` and `_InternalDict` in the **`plistlib`** module. Dict values in the result of functions **`readPlist()`** and **`readPlistFromBytes()`** are now normal dicts. You no longer can use attribute access to access items of these dictionaries.
- The `asyncio.windows_utils.socketpair()` function has been removed. Use the **`socket.socketpair()`** function instead, it is available on all platforms since Python 3.5. `asyncio.windows_utils.socketpair` was just an alias to `socket.socketpair` on Python 3.5 and newer.
- **`asyncio`** no longer exports the **`selectors`** and **`_overlapped`** modules as `asyncio.selectors` and `asyncio._overlapped`. Replace `from asyncio import selectors` with `import selectors`.
- Direct instantiation of **`ssl.SSLSocket`** and **`ssl.SSLObject`** objects is now prohibited. The constructors were never documented, tested, or designed as public constructors. Users were supposed to use **`ssl.wrap_socket()`** or **`ssl.SSLContext`**. (Contributed by Christian Heimes in bpo-32951 [https://bugs.python.org/issue?@action=redirect&bpo=32951].)
- The unused `distutils install_misc` command has been removed. (Contributed by Eric N. Vander Weele in bpo-29218 [https://bugs.python.org/issue?@action=redirect&bpo=29218].)

Module Removals

The `fpectl` module has been removed. It was never enabled by default, never worked correctly on x86-64, and it changed the Python ABI in ways that caused unexpected breakage of C extensions. (Contributed by Nathaniel J. Smith

in bpo-29137 [https://bugs.python.org/issue?@action=redirect&bpo=29137].)

Windows-only Changes

The python launcher, (py.exe), can accept 32 & 64 bit specifiers **without** having to specify a minor version as well. So `py -3-32` and `py -3-64` become valid as well as `py -3.7-32`, also the -*m*-64 and -*m.n*-64 forms are now accepted to force 64 bit python even if 32 bit would have otherwise been used. If the specified version is not available py.exe will error exit. (Contributed by Steve Barnes in bpo-30291 [https://bugs.python.org/issue?@action=redirect&bpo=30291].)

The launcher can be run as `py -0` to produce a list of the installed pythons, *with default marked with an asterisk*. Running `py -0p` will include the paths. If py is run with a version specifier that cannot be matched it will also print the *short form* list of available specifiers. (Contributed by Steve Barnes in bpo-30362 [https://bugs.python.org/issue?@action=redirect&bpo=30362].)

Porting to Python 3.7

This section lists previously described changes and other bugfixes that may require changes to your code.

Changes in Python Behavior

- **`async`** and **`await`** names are now reserved keywords. Code using these names as identifiers will now raise a **`SyntaxError`**. (Contributed by Jelle Zijlstra in bpo-30406 [https://bugs.python.org/issue?@action=redirect&bpo=30406].)

- **PEP 479** [https://peps.python.org/pep-0479/] is enabled for all code in Python 3.7, meaning that **`StopIteration`** exceptions raised directly or indirectly in coroutines and generators are transformed into **`RuntimeError`** exceptions. (Contributed by Yury Selivanov in bpo-32670 [https://bugs.python.org/issue?@action=redirect&bpo=32670].)

- **`object.__aiter__()`** methods can no longer be declared as asynchronous. (Contributed by Yury Selivanov in bpo-31709 [https://bugs.python.org/issue?@action=redirect&bpo=31709].)

- Due to an oversight, earlier Python versions erroneously accepted the following syntax:

  ```
  f(1 for x in [1],)

  class C(1 for x in [1]):
      pass
  ```

  Python 3.7 now correctly raises a **`SyntaxError`**, as a generator expression always needs to be directly inside a set of parentheses and cannot have a comma on either side, and the duplication of the parentheses can be omitted only on calls. (Contributed by Serhiy Storchaka in bpo-32012 [https://bugs.python.org/issue?@action=redirect&bpo=32012] and bpo-32023 [https://bugs.python.org/issue?@action=redirect&bpo=32023].)

- When using the **`-m`** switch, the initial working directory is now added to **`sys.path`**, rather than an empty string (which dynamically denoted the current working directory at the time of each import). Any programs that are checking for the empty string, or otherwise relying on the previous behaviour, will need to be updated accordingly (e.g. by also checking for `os.getcwd()` or `os.path.dirname(__main__.__file__)`, depending on why the code was checking for the empty string in the first place).

Changes in the Python API

- `socketserver.ThreadingMixIn.server_close` now waits until all non-daemon threads complete. Set the new `socketserver.ThreadingMixIn.block_on_close` class attribute to `False` to get the pre-3.7 behaviour. (Contributed by Victor Stinner in bpo-31233 [https://bugs.python.org/issue?@action=redirect&bpo=31233] and bpo-33540 [https://bugs.python.org/issue?@action=redirect&bpo=33540].)

- `socketserver.ForkingMixIn.server_close` now waits until all child processes complete. Set the new `socketserver.ForkingMixIn.block_on_close` class attribute to `False` to get the pre-3.7 behaviour. (Contributed by Victor Stinner in bpo-31151 [https://bugs.python.org/issue?@action=redirect&bpo=31151] and bpo-33540 [https://bugs.python.org/issue?@action=redirect&bpo=33540].)

- The `locale.localeconv()` function now temporarily sets the `LC_CTYPE` locale to the value of `LC_NUMERIC` in some cases. (Contributed by Victor Stinner in bpo-31900 [https://bugs.python.org/issue?@action=redirect&bpo=31900].)

- `pkgutil.walk_packages()` now raises a `ValueError` if *path* is a string. Previously an empty list was returned. (Contributed by Sanyam Khurana in bpo-24744 [https://bugs.python.org/issue?@action=redirect&bpo=24744].)

- A format string argument for `string.Formatter.format()` is now positional-only. Passing it as a keyword argument was deprecated in Python 3.5. (Contributed by Serhiy Storchaka in bpo-29193 [https://bugs.python.org/issue?@action=redirect&bpo=29193].)

- Attributes **`key`**, **`value`** and **`coded_value`** of class **`http.cookies.Morsel`** are now read-only. Assigning to them was deprecated in Python 3.5. Use the **`set()`** method for setting them. (Contributed by Serhiy Storchaka in bpo-29192 [https://bugs.python.org/issue?@action=redirect&bpo=29192].)
- The *mode* argument of **`os.makedirs()`** no longer affects the file permission bits of newly created intermediate-level directories. To set their file permission bits you can set the umask before invoking `makedirs()`. (Contributed by Serhiy Storchaka in bpo-19930 [https://bugs.python.org/issue?@action=redirect&bpo=19930].)
- The **`struct.Struct.format`** type is now **`str`** instead of **`bytes`**. (Contributed by Victor Stinner in bpo-21071 [https://bugs.python.org/issue?@action=redirect&bpo=21071].)
- **`cgi.parse_multipart()`** now accepts the *encoding* and *errors* arguments and returns the same results as **`FieldStorage`**: for non-file fields, the value associated to a key is a list of strings, not bytes. (Contributed by Pierre Quentel in bpo-29979 [https://bugs.python.org/issue?@action=redirect&bpo=29979].)
- Due to internal changes in **`socket`**, calling **`socket.fromshare()`** on a socket created by **`socket.share`** in older Python versions is not supported.
- `repr` for **`BaseException`** has changed to not include the trailing comma. Most exceptions are affected by this change. (Contributed by Serhiy Storchaka in bpo-30399 [https://bugs.python.org/issue?@action=redirect&bpo=30399].)
- `repr` for **`datetime.timedelta`** has changed to include the keyword arguments in the output. (Contributed by Utkarsh Upadhyay in bpo-30302 [https://bugs.python.org/issue?@action=redirect&bpo=30302].)

- Because **`shutil.rmtree()`** is now implemented using the **`os.scandir()`** function, the user specified handler *onerror* is now called with the first argument `os.scandir` instead of `os.listdir` when listing the directory is failed.

- Support for nested sets and set operations in regular expressions as in Unicode Technical Standard #18 [https://unicode.org/reports/tr18/] might be added in the future. This would change the syntax. To facilitate this future change a **`FutureWarning`** will be raised in ambiguous cases for the time being. That include sets starting with a literal `'['` or containing literal character sequences `'--'`, `'&&'`, `'~~'`, and `'||'`. To avoid a warning, escape them with a backslash. (Contributed by Serhiy Storchaka in bpo-30349 [https://bugs.python.org/issue?@action=redirect&bpo=30349].)

- The result of splitting a string on a **`regular expression`** that could match an empty string has been changed. For example splitting on `r'\s*'` will now split not only on whitespaces as it did previously, but also on empty strings before all non-whitespace characters and just before the end of the string. The previous behavior can be restored by changing the pattern to `r'\s+'`. A **`FutureWarning`** was emitted for such patterns since Python 3.5.

  For patterns that match both empty and non-empty strings, the result of searching for all matches may also be changed in other cases. For example in the string `'a\n\n'`, the pattern `r'(?m)^\s*?$'` will not only match empty strings at positions 2 and 3, but also the string `'\n'` at positions 2–3. To match only blank lines, the pattern should be rewritten as `r'(?m)^[^\S\n]*$'`.

[**`re.sub()`**] now replaces empty matches adjacent to a previous non-empty match. For example `re.sub('x*', '-', 'abxd')` returns now `'-a-b--d-'` instead of `'-a-b-d-'` (the first minus between ‘b’ and ‘d’ replaces ‘x’, and the second minus replaces an empty string between ‘x’ and ‘d’).

(Contributed by Serhiy Storchaka in bpo-25054 [https://bugs.python.org/issue?@action=redirect&bpo=25054] and bpo-32308 [https://bugs.python.org/issue?@action=redirect&bpo=32308].)

- Change **`re.escape()`** to only escape regex special characters instead of escaping all characters other than ASCII letters, numbers, and `'_'`. (Contributed by Serhiy Storchaka in bpo-29995 [https://bugs.python.org/issue?@action=redirect&bpo=29995].)
- **`tracemalloc.Traceback`** frames are now sorted from oldest to most recent to be more consistent with **`traceback`**. (Contributed by Jesse Bakker in bpo-32121 [https://bugs.python.org/issue?@action=redirect&bpo=32121].)
- On OSes that support **`socket.SOCK_NONBLOCK`** or **`socket.SOCK_CLOEXEC`** bit flags, the **`socket.type`** no longer has them applied. Therefore, checks like `if sock.type == socket.SOCK_STREAM` work as expected on all platforms. (Contributed by Yury Selivanov in bpo-32331 [https://bugs.python.org/issue?@action=redirect&bpo=32331].)
- On Windows the default for the *close_fds* argument of **`subprocess.Popen`** was changed from **`False`** to **`True`** when redirecting the standard handles. If you previously depended on handles being inherited when using **`subprocess.Popen`** with standard io redirection, you will have to pass `close_fds=False` to preserve the previous behaviour, or use **`STARTUPINFO.lpAttributeList`**.

- `importlib.machinery.PathFinder.invalidate_caches()` – which implicitly affects `importlib.invalidate_caches()` – now deletes entries in `sys.path_importer_cache` which are set to `None`. (Contributed by Brett Cannon in bpo-33169 [https://bugs.python.org/issue?@action=redirect&bpo=33169].)

- In `asyncio`, `loop.sock_recv()`, `loop.sock_sendall()`, `loop.sock_accept()`, `loop.getaddrinfo()`, `loop.getnameinfo()` have been changed to be proper coroutine methods to match their documentation. Previously, these methods returned `asyncio.Future` instances. (Contributed by Yury Selivanov in bpo-32327 [https://bugs.python.org/issue?@action=redirect&bpo=32327].)

- `asyncio.Server.sockets` now returns a copy of the internal list of server sockets, instead of returning it directly. (Contributed by Yury Selivanov in bpo-32662 [https://bugs.python.org/issue?@action=redirect&bpo=32662].)

- `Struct.format` is now a `str` instance instead of a `bytes` instance. (Contributed by Victor Stinner in bpo-21071 [https://bugs.python.org/issue?@action=redirect&bpo=21071].)

- `argparse` subparsers can now be made mandatory by passing `required=True` to `ArgumentParser.add_subparsers()`. (Contributed by Anthony Sottile in bpo-26510 [https://bugs.python.org/issue?@action=redirect&bpo=26510].)

- `ast.literal_eval()` is now stricter. Addition and subtraction of arbitrary numbers are no longer allowed. (Contributed by Serhiy Storchaka in bpo-31778 [https://bugs.python.org/issue?@action=redirect&bpo=31778].)

- `Calendar.itermonthdates` will now consistently raise an exception when a date falls outside of the `0001-01-01` through `9999-12-31` range. To support applications that cannot tolerate such exceptions, the new

`Calendar.itermonthdays3` and `Calendar.itermonthdays4` can be used. The new methods return tuples and are not restricted by the range supported by `datetime.date`. (Contributed by Alexander Belopolsky in bpo-28292 [https://bugs.python.org/issue?@action=redirect&bpo=28292].)

- `collections.ChainMap` now preserves the order of the underlying mappings. (Contributed by Raymond Hettinger in bpo-32792 [https://bugs.python.org/issue?@action=redirect&bpo=32792].)
- The `submit()` method of `concurrent.futures.ThreadPoolExecutor` and `concurrent.futures.ProcessPoolExecutor` now raises a `RuntimeError` if called during interpreter shutdown. (Contributed by Mark Nemec in bpo-33097 [https://bugs.python.org/issue?@action=redirect&bpo=33097].)
- The `configparser.ConfigParser` constructor now uses `read_dict()` to process the default values, making its behavior consistent with the rest of the parser. Non-string keys and values in the defaults dictionary are now being implicitly converted to strings. (Contributed by James Tocknell in bpo-23835 [https://bugs.python.org/issue?@action=redirect&bpo=23835].)
- Several undocumented internal imports were removed. One example is that `os.errno` is no longer available; use `import errno` directly instead. Note that such undocumented internal imports may be removed any time without notice, even in micro version releases.

Changes in the C API

The function `PySlice_GetIndicesEx()` is considered unsafe for resizable sequences. If the slice indices are not instances of

`int`, but objects that implement the **`__index__()`** method, the sequence can be resized after passing its length to **`PySlice_GetIndicesEx()`**. This can lead to returning indices out of the length of the sequence. For avoiding possible problems use new functions **`PySlice_Unpack()`** and **`PySlice_AdjustIndices()`**. (Contributed by Serhiy Storchaka in bpo-27867 [https://bugs.python.org/issue?@action=redirect&bpo=27867].)

CPython bytecode changes

There are two new opcodes: **`LOAD_METHOD`** and **`CALL_METHOD`**. (Contributed by Yury Selivanov and INADA Naoki in bpo-26110 [https://bugs.python.org/issue?@action=redirect&bpo=26110].)

The **`STORE_ANNOTATION`** opcode has been removed. (Contributed by Mark Shannon in bpo-32550 [https://bugs.python.org/issue?@action=redirect&bpo=32550].)

Windows-only Changes

The file used to override **`sys.path`** is now called `<python-executable>._pth` instead of `'sys.path'`. See Finding modules for more information. (Contributed by Steve Dower in bpo-28137 [https://bugs.python.org/issue?@action=redirect&bpo=28137].)

Other CPython implementation changes

In preparation for potential future changes to the public CPython runtime initialization API (see **PEP 432** [https://peps.python.org/pep-0432/] for an initial, but somewhat outdated, draft), CPython’s internal startup and configuration management logic has been significantly refactored. While these updates are intended to be entirely transparent to both embedding applications and users of the regular CPython CLI, they’re being mentioned here as the refactoring changes the internal order of various operations during interpreter startup, and hence may uncover previously latent defects, either in embedding applications, or in CPython itself. (Initially contributed by Nick Coghlan and Eric Snow as part of bpo-22257 [https://bugs.python.org/issue?@action=redirect&bpo=22257], and further updated by Nick, Eric, and Victor Stinner in a number of other issues). Some known details affected:

- **`PySys_AddWarnOptionUnicode()`** is not currently usable by embedding applications due to the requirement to create a Unicode object prior to calling `Py_Initialize`. Use **`PySys_AddWarnOption()`** instead.
- warnings filters added by an embedding application with **`PySys_AddWarnOption()`** should now more consistently take precedence over the default filters set by the interpreter

Due to changes in the way the default warnings filters are configured, setting **`Py_BytesWarningFlag`** to a value greater than one is no longer sufficient to both emit **`BytesWarning`** messages and have them converted to exceptions. Instead, the flag must be set (to cause the warnings to be emitted in the first place), and an explicit `error::BytesWarning` warnings filter added to convert them to exceptions.

Due to a change in the way docstrings are handled by the compiler, the implicit `return None` in a function body consisting solely of a docstring is now marked as occurring on the same line as the docstring, not on the function's header line.

The current exception state has been moved from the frame object to the co-routine. This simplified the interpreter and fixed a couple of obscure bugs caused by having swap exception state when entering or exiting a generator. (Contributed by Mark Shannon in bpo-25612 [https://bugs.python.org/issue?@action=redirect&bpo=25612].)

Notable changes in Python 3.7.1

Starting in 3.7.1, **`Py_Initialize()`** now consistently reads and respects all of the same environment settings as **`Py_Main()`** (in earlier Python versions, it respected an ill-defined subset of those environment variables, while in Python 3.7.0 it didn’t read any of them due to bpo-34247 [https://bugs.python.org/issue?@action=redirect&bpo=34247]). If this behavior is unwanted, set **`Py_IgnoreEnvironmentFlag`** to 1 before calling **`Py_Initialize()`**.

In 3.7.1 the C API for Context Variables was updated to use **`PyObject`** pointers. See also bpo-34762 [https://bugs.python.org/issue?@action=redirect&bpo=34762].

In 3.7.1 the **`tokenize`** module now implicitly emits a `NEWLINE` token when provided with input that does not have a trailing new line. This behavior now matches what the C tokenizer does internally. (Contributed by Ammar Askar in bpo-33899 [https://bugs.python.org/issue?@action=redirect&bpo=33899].)

Notable changes in Python 3.7.2

In 3.7.2, **`venv`** on Windows no longer copies the original binaries, but creates redirector scripts named `python.exe` and `pythonw.exe` instead. This resolves a long standing issue where all virtual environments would have to be upgraded or recreated with each Python update. However, note that this release will still require recreation of virtual environments in order to get the new scripts.

Notable changes in Python 3.7.6

Due to significant security concerns, the *reuse_address* parameter of **`asyncio.loop.create_datagram_endpoint()`** is no longer supported. This is because of the behavior of the socket option `SO_REUSEADDR` in UDP. For more details, see the documentation for `loop.create_datagram_endpoint()`. (Contributed by Kyle Stanley, Antoine Pitrou, and Yury Selivanov in bpo-37228 [https://bugs.python.org/issue?@action=redirect&bpo=37228].)

Notable changes in Python 3.7.10

Earlier Python versions allowed using both `;` and `&` as query parameter separators in **`urllib.parse.parse_qs()`** and **`urllib.parse.parse_qsl()`**. Due to security concerns, and to conform with newer W3C recommendations, this has been changed to allow only a single separator key, with `&` as the default. This change also affects **`cgi.parse()`** and **`cgi.parse_multipart()`** as they use the affected functions internally. For more details, please see their respective documentation. (Contributed by Adam Goldschmidt, Senthil Kumaran and Ken Jin in bpo-42967 [https://bugs.python.org/issue?@action=redirect&bpo=42967].)

Notable changes in Python 3.7.11

A security fix alters the **`ftplib.FTP`** behavior to not trust the IPv4 address sent from the remote server when setting up a passive data channel. We reuse the ftp server IP address instead. For unusual code requiring the old behavior, set a `trust_server_pasv_ipv4_address` attribute on your FTP instance to `True`. (See gh-87451 [https://github.com/python/cpython/issues/87451])

The presence of newline or tab characters in parts of a URL allows for some forms of attacks. Following the WHATWG specification that updates RFC 3986, ASCII newline `\n`, `\r` and tab `\t` characters are stripped from the URL by the parser `urllib.parse()` preventing such attacks. The removal characters are controlled by a new module level variable `urllib.parse._UNSAFE_URL_BYTES_TO_REMOVE`. (See gh-88048 [https://github.com/python/cpython/issues/88048])

Notable security feature in 3.7.14

Converting between `int` and `str` in bases other than 2 (binary), 4, 8 (octal), 16 (hexadecimal), or 32 such as base 10 (decimal) now raises a `ValueError` if the number of digits in string form is above a limit to avoid potential denial of service attacks due to the algorithmic complexity. This is a mitigation for **CVE 2020-10735** [https://www.cve.org/CVERecord?id=CVE-2020-10735]. This limit can be configured or disabled by environment variable, command line flag, or `sys` APIs. See the integer string conversion length limitation documentation. The default limit is 4300 digits in string form.

What’s New In Python 3.6

Editors:: Elvis Pranskevichus <elvis@magic.io>, Yury Selivanov <yury@magic.io>

This article explains the new features in Python 3.6, compared to 3.5. Python 3.6 was released on December 23, 2016. See the changelog [https://docs.python.org/3.6/whatsnew/changelog.html] for a full list of changes.

See also

PEP 494 [https://peps.python.org/pep-0494/] - Python 3.6 Release Schedule

Summary – Release highlights

New syntax features:

- PEP 498, formatted string literals.
- PEP 515, underscores in numeric literals.
- PEP 526, syntax for variable annotations.
- PEP 525, asynchronous generators.
- PEP 530: asynchronous comprehensions.

New library modules:

- `secrets`: PEP 506 – Adding A Secrets Module To The Standard Library.

CPython implementation improvements:

- The dict type has been reimplemented to use a more compact representation based on a proposal by Raymond Hettinger [https://mail.python.org/pipermail/python-dev/2012-December/123028.html] and similar to the PyPy dict implementation [https://morepypy.blogspot.com/2015/01/faster-more-memory-efficient-and-more.html]. This resulted in dictionaries using 20% to 25% less memory when compared to Python 3.5.
- Customization of class creation has been simplified with the new protocol.
- The class attribute definition order is now preserved.
- The order of elements in `**kwargs` now corresponds to the order in which keyword arguments were passed to the function.
- DTrace and SystemTap probing support has been added.
- The new PYTHONMALLOC environment variable can now be used to debug the interpreter memory allocation and access errors.

Significant improvements in the standard library:

- The **`asyncio`** module has received new features, significant usability and performance improvements, and a fair amount of bug fixes. Starting with Python 3.6 the `asyncio` module is no longer provisional and its API is considered stable.
- A new file system path protocol has been implemented to support path-like objects. All standard library functions operating on paths have been updated to work with the new protocol.
- The **`datetime`** module has gained support for Local Time Disambiguation.
- The **`typing`** module received a number of improvements.
- The **`tracemalloc`** module has been significantly reworked and is now used to provide better output for **`ResourceWarning`** as well as provide better diagnostics for

memory allocation errors. See the PYTHONMALLOC section for more information.

Security improvements:

- The new **`secrets`** module has been added to simplify the generation of cryptographically strong pseudo-random numbers suitable for managing secrets such as account authentication, tokens, and similar.
- On Linux, **`os.urandom()`** now blocks until the system urandom entropy pool is initialized to increase the security. See the **PEP 524** [https://peps.python.org/pep-0524/] for the rationale.
- The **`hashlib`** and **`ssl`** modules now support OpenSSL 1.1.0.
- The default settings and feature set of the **`ssl`** module have been improved.
- The **`hashlib`** module received support for the BLAKE2, SHA-3 and SHAKE hash algorithms and the **`scrypt()`** key derivation function.

Windows improvements:

- PEP 528 and PEP 529, Windows filesystem and console encoding changed to UTF-8.
- The `py.exe` launcher, when used interactively, no longer prefers Python 2 over Python 3 when the user doesn’t specify a version (via command line arguments or a config file). Handling of shebang lines remains unchanged - “python” refers to Python 2 in that case.
- `python.exe` and `pythonw.exe` have been marked as long-path aware, which means that the 260 character path limit may no longer apply. See removing the MAX_PATH limitation for details.
- A `._pth` file can be added to force isolated mode and fully specify all search paths to avoid registry and

environment lookup. See the documentation for more information.

- A `python36.zip` file now works as a landmark to infer **`PYTHONHOME`**. See the documentation for more information.

New Features

PEP 498: Formatted string literals

PEP 498 [https://peps.python.org/pep-0498/] introduces a new kind of string literals: *f-strings*, or formatted string literals.

Formatted string literals are prefixed with `'f'` and are similar to the format strings accepted by **`str.format()`**. They contain replacement fields surrounded by curly braces. The replacement fields are expressions, which are evaluated at run time, and then formatted using the **`format()`** protocol:

```
>>> name = "Fred"
>>> f"He said his name is {name}."
'He said his name is Fred.'
>>> width = 10
>>> precision = 4
>>> value = decimal.Decimal("12.34567")
>>> f"result: {value:{width}.{precision}}"  # nested fields
'result:      12.35'
```

See also

PEP 498 [https://peps.python.org/pep-0498/] – Literal String Interpolation.
PEP written and implemented by Eric V. Smith.

Feature documentation.

PEP 526: Syntax for variable annotations

PEP 484 [https://peps.python.org/pep-0484/] introduced the standard for type annotations of function parameters, a.k.a. type hints. This PEP adds syntax to Python for annotating the types of variables including class variables and instance variables:

```
primes: List[int] = []

captain: str  # Note: no initial value!

class Starship:
    stats: Dict[str, int] = {}
```

Just as for function annotations, the Python interpreter does not attach any particular meaning to variable annotations and only stores them in the `__annotations__` attribute of a class or module.

In contrast to variable declarations in statically typed languages, the goal of annotation syntax is to provide an easy way to specify structured type metadata for third party tools and libraries via the abstract syntax tree and the `__annotations__` attribute.

See also

PEP 526 [https://peps.python.org/pep-0526/] – Syntax for variable annotations.

PEP written by Ryan Gonzalez, Philip House, Ivan Levkivskyi, Lisa Roach, and Guido van Rossum. Implemented by Ivan Levkivskyi.

Tools that use or will use the new syntax: mypy [https://www.mypy-lang.org/], pytype [https://github.com/google/pytype], PyCharm, etc.

PEP 515: Underscores in Numeric Literals

PEP 515 [https://peps.python.org/pep-0515/] adds the ability to use underscores in numeric literals for improved readability. For example:

```
>>> 1_000_000_000_000_000
1000000000000000
>>> 0x_FF_FF_FF_FF
4294967295
```

Single underscores are allowed between digits and after any base specifier. Leading, trailing, or multiple underscores in a row are not allowed.

The string formatting language also now has support for the `'_'` option to signal the use of an underscore for a thousands separator for floating-point presentation types and for integer presentation type `'d'`. For integer presentation types `'b'`, `'o'`, `'x'`, and `'X'`, underscores will be inserted every 4 digits:

```
>>> '{:_}'.format(1000000)
'1_000_000'
>>> '{:_x}'.format(0xFFFFFFFF)
'ffff_ffff'
```

See also

PEP 515 [https://peps.python.org/pep-0515/] – Underscores in Numeric Literals
PEP written by Georg Brandl and Serhiy Storchaka.

PEP 525: Asynchronous Generators

PEP 492 [https://peps.python.org/pep-0492/] introduced support for native coroutines and `async` / `await` syntax to Python 3.5. A notable limitation of the Python 3.5 implementation is that it was not possible to use `await` and `yield` in the same function body. In Python 3.6 this restriction has been lifted, making it possible to define *asynchronous generators*:

```
async def ticker(delay, to):
    """Yield numbers from 0 to *to* every *delay* seconds."""
    for i in range(to):
        yield i
        await asyncio.sleep(delay)
```

The new syntax allows for faster and more concise code.

See also

PEP 525 [https://peps.python.org/pep-0525/] – Asynchronous Generators

PEP written and implemented by Yury Selivanov.

PEP 530: Asynchronous Comprehensions

PEP 530 [https://peps.python.org/pep-0530/] adds support for using `async for` in list, set, dict comprehensions and generator expressions:

```
result = [i async for i in aiter() if i % 2]
```

Additionally, `await` expressions are supported in all kinds of comprehensions:

```
result = [await fun() for fun in funcs if await condition()]
```

See also

PEP 530 [https://peps.python.org/pep-0530/] – Asynchronous Comprehensions

PEP written and implemented by Yury Selivanov.

PEP 487: Simpler customization of class creation

It is now possible to customize subclass creation without using a metaclass. The new `__init_subclass__` classmethod will be called on the base class whenever a new subclass is created:

```
class PluginBase:
    subclasses = []

    def __init_subclass__(cls, **kwargs):
        super().__init_subclass__(**kwargs)
        cls.subclasses.append(cls)

class Plugin1(PluginBase):
    pass

class Plugin2(PluginBase):
    pass
```

In order to allow zero-argument **`super()`** calls to work correctly from **`__init_subclass__()`** implementations, custom metaclasses must ensure that the new `__classcell__` namespace entry is propagated to `type.__new__` (as described in Creating the class object).

See also

PEP 487 [https://peps.python.org/pep-0487/] – Simpler customization of class creation

PEP written and implemented by Martin Teichmann.

Feature documentation

PEP 487: Descriptor Protocol Enhancements

PEP 487 [https://peps.python.org/pep-0487/] extends the descriptor protocol to include the new optional `__set_name__()` method. Whenever a new class is defined, the new method will be called on all descriptors included in the definition, providing them with a reference to the class being defined and the name given to the descriptor within the class namespace. In other words, instances of descriptors can now know the attribute name of the descriptor in the owner class:

```
class IntField:
    def __get__(self, instance, owner):
        return instance.__dict__[self.name]

    def __set__(self, instance, value):
        if not isinstance(value, int):
            raise ValueError(f'expecting integer in
{self.name}')
        instance.__dict__[self.name] = value

    # this is the new initializer:
    def __set_name__(self, owner, name):
        self.name = name

class Model:
    int_field = IntField()
```

See also

PEP 487 [https://peps.python.org/pep-0487/] – Simpler customization of class creation

PEP written and implemented by Martin Teichmann.

Feature documentation

PEP 519: Adding a file system path protocol

File system paths have historically been represented as `str` or `bytes` objects. This has led to people who write code which operate on file system paths to assume that such objects are only one of those two types (an `int` representing a file descriptor does not count as that is not a file path). Unfortunately that assumption prevents alternative object representations of file system paths like `pathlib` from working with pre-existing code, including Python’s standard library.

To fix this situation, a new interface represented by `os.PathLike` has been defined. By implementing the `__fspath__()` method, an object signals that it represents a path. An object can then provide a low-level representation of a file system path as a `str` or `bytes` object. This means an object is considered path-like if it implements `os.PathLike` or is a `str` or `bytes` object which represents a file system path. Code can use `os.fspath()`, `os.fsdecode()`, or `os.fsencode()` to explicitly get a `str` and/or `bytes` representation of a path-like object.

The built-in `open()` function has been updated to accept `os.PathLike` objects, as have all relevant functions in the `os` and `os.path` modules, and most other functions and classes in the standard library. The `os.DirEntry` class and relevant classes in `pathlib` have also been updated to implement `os.PathLike`.

The hope is that updating the fundamental functions for operating on file system paths will lead to third-party code to implicitly support all path-like objects without any code changes, or at least very minimal ones (e.g. calling `os.fspath()` at the beginning of code before operating on a path-like object).

Here are some examples of how the new interface allows for `pathlib.Path` to be used more easily and transparently with pre-existing code:

```
>>> import pathlib
>>> with open(pathlib.Path("README")) as f:
...     contents = f.read()
...
>>> import os.path
>>> os.path.splitext(pathlib.Path("some_file.txt"))
('some_file', '.txt')
>>> os.path.join("/a/b", pathlib.Path("c"))
'/a/b/c'
>>> import os
>>> os.fspath(pathlib.Path("some_file.txt"))
'some_file.txt'
```

(Implemented by Brett Cannon, Ethan Furman, Dusty Phillips, and Jelle Zijlstra.)

See also

PEP 519 [https://peps.python.org/pep-0519/] – Adding a file system path protocol

PEP written by Brett Cannon and Koos Zevenhoven.

PEP 495: Local Time Disambiguation

In most world locations, there have been and will be times when local clocks are moved back. In those times, intervals are introduced in which local clocks show the same time twice in the same day. In these situations, the information displayed on a local clock (or stored in a Python datetime instance) is insufficient to identify a particular moment in time.

PEP 495 [https://peps.python.org/pep-0495/] adds the new *fold* attribute to instances of `datetime.datetime` and `datetime.time` classes to differentiate between two moments in time for which local times are the same:

```
>>> u0 = datetime(2016, 11, 6, 4, tzinfo=timezone.utc)
>>> for i in range(4):
...     u = u0 + i*HOUR
...     t = u.astimezone(Eastern)
...     print(u.time(), 'UTC =', t.time(), t.tzname(), t.fold)
...
04:00:00 UTC = 00:00:00 EDT 0
05:00:00 UTC = 01:00:00 EDT 0
06:00:00 UTC = 01:00:00 EST 1
07:00:00 UTC = 02:00:00 EST 0
```

The values of the `fold` attribute have the value `0` for all instances except those that represent the second (chronologically) moment in time in an ambiguous case.

See also

PEP 495 [https://peps.python.org/pep-0495/] – Local Time Disambiguation

PEP written by Alexander Belopolsky and Tim Peters, implementation by Alexander Belopolsky.

PEP 529: Change Windows filesystem encoding to UTF-8

Representing filesystem paths is best performed with str (Unicode) rather than bytes. However, there are some situations where using bytes is sufficient and correct.

Prior to Python 3.6, data loss could result when using bytes paths on Windows. With this change, using bytes to represent paths is now supported on Windows, provided those bytes are encoded with the encoding returned by `sys.getfilesystemencoding()`, which now defaults to `'utf-8'`.

Applications that do not use str to represent paths should use `os.fsencode()` and `os.fsdecode()` to ensure their bytes are correctly encoded. To revert to the previous behaviour, set `PYTHONLEGACYWINDOWSFSENCODING` or call `sys._enablelegacywindowsfsencoding()`.

See PEP 529 [https://peps.python.org/pep-0529/] for more information and discussion of code modifications that may be required.

PEP 528: Change Windows console encoding to UTF-8

The default console on Windows will now accept all Unicode characters and provide correctly read str objects to Python code. `sys.stdin`, `sys.stdout` and `sys.stderr` now default to utf-8 encoding.

This change only applies when using an interactive console, and not when redirecting files or pipes. To revert to the previous behaviour for interactive console use, set `PYTHONLEGACYWINDOWSSTDIO`.

See also

PEP 528 [https://peps.python.org/pep-0528/] – Change Windows console encoding to UTF-8
PEP written and implemented by Steve Dower.

PEP 520: Preserving Class Attribute Definition Order

Attributes in a class definition body have a natural ordering: the same order in which the names appear in the source. This order is now preserved in the new class's `__dict__` attribute.

Also, the effective default class *execution* namespace (returned from type.__prepare__()) is now an insertion-order-preserving mapping.

See also

PEP 520 [https://peps.python.org/pep-0520/] – Preserving Class Attribute Definition Order
PEP written and implemented by Eric Snow.

PEP 468: Preserving Keyword Argument Order

`**kwargs` in a function signature is now guaranteed to be an insertion-order-preserving mapping.

See also

PEP 468 [https://peps.python.org/pep-0468/] – Preserving Keyword Argument Order
PEP written and implemented by Eric Snow.

New dict implementation

The dict type now uses a "compact" representation based on a proposal by Raymond Hettinger [https://mail.python.org/pipermail/python-dev/2012-December/123028.html] which was first implemented by PyPy [https://morepypy.blogspot.com/2015/01/faster-more-memory-efficient-and-more.html]. The memory usage of the new `dict()` is between 20% and 25% smaller compared to Python 3.5.

The order-preserving aspect of this new implementation is considered an implementation detail and should not be relied upon (this may change in the future, but it is desired to have this new dict implementation in the language for a few releases before changing the language spec to mandate order-preserving semantics for all current and future Python implementations; this also helps preserve backwards-compatibility with older versions of the language where random iteration order is still in effect, e.g. Python 3.5).

(Contributed by INADA Naoki in bpo-27350 [https://bugs.python.org/issue?@action=redirect&bpo=27350]. Idea originally suggested by Raymond Hettinger [https://mail.python.org/pipermail/python-dev/2012-December/123028.html].)

PEP 523: Adding a frame evaluation API to CPython

While Python provides extensive support to customize how code executes, one place it has not done so is in the evaluation of frame objects. If you wanted some way to intercept frame evaluation in Python there really wasn’t any way without directly manipulating function pointers for defined functions.

PEP 523 [https://peps.python.org/pep-0523/] changes this by providing an API to make frame evaluation pluggable at the C level. This will allow for tools such as debuggers and JITs to intercept frame evaluation before the execution of Python code begins. This enables the use of alternative evaluation implementations for Python code, tracking frame evaluation, etc.

This API is not part of the limited C API and is marked as private to signal that usage of this API is expected to be limited and only applicable to very select, low-level use-cases. Semantics of the API will change with Python as necessary.

See also

PEP 523 [https://peps.python.org/pep-0523/] – Adding a frame evaluation API to CPython
PEP written by Brett Cannon and Dino Viehland.

PYTHONMALLOC environment variable

The new `PYTHONMALLOC` environment variable allows setting the Python memory allocators and installing debug hooks.

It is now possible to install debug hooks on Python memory allocators on Python compiled in release mode using `PYTHONMALLOC=debug`. Effects of debug hooks:

- Newly allocated memory is filled with the byte `0xCB`
- Freed memory is filled with the byte `0xDB`
- Detect violations of the Python memory allocator API. For example, `PyObject_Free()` called on a memory block allocated by `PyMem_Malloc()`.
- Detect writes before the start of a buffer (buffer underflows)
- Detect writes after the end of a buffer (buffer overflows)
- Check that the GIL is held when allocator functions of `PYMEM_DOMAIN_OBJ` (ex: `PyObject_Malloc()`) and `PYMEM_DOMAIN_MEM` (ex: `PyMem_Malloc()`) domains are called.

Checking if the GIL is held is also a new feature of Python 3.6.

See the **`PyMem_SetupDebugHooks()`** function for debug hooks on Python memory allocators.

It is now also possible to force the usage of the **`malloc()`** allocator of the C library for all Python memory allocations using `PYTHONMALLOC=malloc`. This is helpful when using external memory debuggers like Valgrind on a Python compiled in release mode.

On error, the debug hooks on Python memory allocators now use the **`tracemalloc`** module to get the traceback where a memory block was allocated.

Example of fatal error on buffer overflow using `python3.6 -X tracemalloc=5` (store 5 frames in traces):

```
Debug memory block at address p=0x7fbcd41666f8: API 'o'
    4 bytes originally requested
    The 7 pad bytes at p-7 are FORBIDDENBYTE, as expected.
    The 8 pad bytes at tail=0x7fbcd41666fc are not all
FORBIDDENBYTE (0xfb):
        at tail+0: 0x02 *** OUCH
        at tail+1: 0xfb
        at tail+2: 0xfb
        at tail+3: 0xfb
        at tail+4: 0xfb
        at tail+5: 0xfb
        at tail+6: 0xfb
        at tail+7: 0xfb
    The block was made by call #1233329 to debug
malloc/realloc.
    Data at p: 1a 2b 30 00

Memory block allocated at (most recent call first):
  File "test/test_bytes.py", line 323
  File "unittest/case.py", line 600
  File "unittest/case.py", line 648
  File "unittest/suite.py", line 122
  File "unittest/suite.py", line 84
```

```
Fatal Python error: bad trailing pad byte

Current thread 0x00007fbcdbd32700 (most recent call first):
  File "test/test_bytes.py", line 323 in test_hex
  File "unittest/case.py", line 600 in run
  File "unittest/case.py", line 648 in __call__
  File "unittest/suite.py", line 122 in run
  File "unittest/suite.py", line 84 in __call__
  File "unittest/suite.py", line 122 in run
  File "unittest/suite.py", line 84 in __call__
  ...
```

(Contributed by Victor Stinner in bpo-26516 [https://bugs.python.org/issue?@action=redirect&bpo=26516] and bpo-26564 [https://bugs.python.org/issue?@action=redirect&bpo=26564].)

DTrace and SystemTap probing support

Python can now be built `--with-dtrace` which enables static markers for the following events in the interpreter:

- function call/return
- garbage collection started/finished
- line of code executed.

This can be used to instrument running interpreters in production, without the need to recompile specific debug builds or providing application-specific profiling/debugging code.

More details in Instrumenting CPython with DTrace and SystemTap.

The current implementation is tested on Linux and macOS. Additional markers may be added in the future.

(Contributed by Łukasz Langa in bpo-21590 [https://bugs.python.org/issue?@action=redirect&bpo=21590], based on patches by Jesús Cea Avión, David Malcolm, and Nikhil Benesch.)

Other Language Changes

Some smaller changes made to the core Python language are:

- A `global` or `nonlocal` statement must now textually appear before the first use of the affected name in the same scope. Previously this was a **`SyntaxWarning`**.
- It is now possible to set a special method to `None` to indicate that the corresponding operation is not available. For example, if a class sets **`__iter__()`** to `None`, the class is not iterable. (Contributed by Andrew Barnert and Ivan Levkivskyi in bpo-25958 [https://bugs.python.org/issue?@action=redirect&bpo=25958].)
- Long sequences of repeated traceback lines are now abbreviated as `"[Previous line repeated {count} more times]"` (see traceback for an example). (Contributed by Emanuel Barry in bpo-26823 [https://bugs.python.org/issue?@action=redirect&bpo=26823].)
- Import now raises the new exception **`ModuleNotFoundError`** (subclass of **`ImportError`**) when it cannot find a module. Code that currently checks for ImportError (in try-except) will still work. (Contributed by Eric Snow in bpo-15767 [https://bugs.python.org/issue?@action=redirect&bpo=15767].)
- Class methods relying on zero-argument `super()` will now work correctly when called from metaclass methods during class creation. (Contributed by Martin Teichmann in bpo-23722 [https://bugs.python.org/issue?@action=redirect&bpo=23722].)

New Modules

secrets

The main purpose of the new `secrets` module is to provide an obvious way to reliably generate cryptographically strong pseudo-random values suitable for managing secrets, such as account authentication, tokens, and similar.

Warning

Note that the pseudo-random generators in the `random` module should *NOT* be used for security purposes. Use `secrets` on Python 3.6+ and `os.urandom()` on Python 3.5 and earlier.

See also

PEP 506 [https://peps.python.org/pep-0506/] – Adding A Secrets Module To The Standard Library
PEP written and implemented by Steven D’Aprano.

Improved Modules

array

Exhausted iterators of `array.array` will now stay exhausted even if the iterated array is extended. This is consistent with the behavior of other mutable sequences.

Contributed by Serhiy Storchaka in bpo-26492 [https://bugs.python.org/issue?@action=redirect&bpo=26492].

ast

The new `ast.Constant` AST node has been added. It can be used by external AST optimizers for the purposes of constant folding.

Contributed by Victor Stinner in bpo-26146 [https://bugs.python.org/issue?@action=redirect&bpo=26146].

asyncio

Starting with Python 3.6 the `asyncio` module is no longer provisional and its API is considered stable.

Notable changes in the `asyncio` module since Python 3.5.0 (all backported to 3.5.x due to the provisional status):

- The `get_event_loop()` function has been changed to always return the currently running loop when called from coroutines and callbacks. (Contributed by Yury Selivanov in bpo-28613 [https://bugs.python.org/issue?@action=redirect&bpo=28613].)
- The `ensure_future()` function and all functions that use it, such as `loop.run_until_complete()`, now accept all kinds of awaitable objects. (Contributed by Yury Selivanov.)
- New `run_coroutine_threadsafe()` function to submit coroutines to event loops from other threads. (Contributed by Vincent Michel.)
- New `Transport.is_closing()` method to check if the transport is closing or closed. (Contributed by Yury Selivanov.)
- The `loop.create_server()` method can now accept a list of hosts. (Contributed by Yann Sionneau.)
- New `loop.create_future()` method to create Future objects. This allows alternative event loop implementations, such as uvloop [https://github.com/MagicStack/uvloop], to provide a faster `asyncio.Future` implementation. (Contributed by Yury

Selivanov in bpo-27041 [https://bugs.python.org/issue?@action=redirect&bpo=27041].)

- New **`loop.get_exception_handler()`** method to get the current exception handler. (Contributed by Yury Selivanov in bpo-27040 [https://bugs.python.org/issue?@action=redirect&bpo=27040].)
- New **`StreamReader.readuntil()`** method to read data from the stream until a separator bytes sequence appears. (Contributed by Mark Korenberg.)
- The performance of **`StreamReader.readexactly()`** has been improved. (Contributed by Mark Korenberg in bpo-28370 [https://bugs.python.org/issue?@action=redirect&bpo=28370].)
- The **`loop.getaddrinfo()`** method is optimized to avoid calling the system `getaddrinfo` function if the address is already resolved. (Contributed by A. Jesse Jiryu Davis.)
- The **`loop.stop()`** method has been changed to stop the loop immediately after the current iteration. Any new callbacks scheduled as a result of the last iteration will be discarded. (Contributed by Guido van Rossum in bpo-25593 [https://bugs.python.org/issue?@action=redirect&bpo=25593].)
- **`Future.set_exception`** will now raise **`TypeError`** when passed an instance of the **`StopIteration`** exception. (Contributed by Chris Angelico in bpo-26221 [https://bugs.python.org/issue?@action=redirect&bpo=26221].)
- New **`loop.connect_accepted_socket()`** method to be used by servers that accept connections outside of asyncio, but that use asyncio to handle them. (Contributed by Jim Fulton in bpo-27392 [https://bugs.python.org/issue?@action=redirect&bpo=27392].)
- `TCP_NODELAY` flag is now set for all TCP transports by default. (Contributed by Yury Selivanov in bpo-27456 [https://bugs.python.org/issue?@action=redirect&bpo=27456].)
- New **`loop.shutdown_asyncgens()`** to properly close pending asynchronous generators before closing the loop. (Contributed by Yury Selivanov in bpo-28003 [https://bugs.python.org/issue?@action=redirect&bpo=28003].)

- `Future` and `Task` classes now have an optimized C implementation which makes asyncio code up to 30% faster. (Contributed by Yury Selivanov and INADA Naoki in bpo-26081 [https://bugs.python.org/issue?@action=redirect&bpo=26081] and bpo-28544 [https://bugs.python.org/issue?@action=redirect&bpo=28544].)

binascii

The `b2a_base64()` function now accepts an optional *newline* keyword argument to control whether the newline character is appended to the return value. (Contributed by Victor Stinner in bpo-25357 [https://bugs.python.org/issue?@action=redirect&bpo=25357].)

cmath

The new `cmath.tau` (τ) constant has been added. (Contributed by Lisa Roach in bpo-12345 [https://bugs.python.org/issue?@action=redirect&bpo=12345], see **PEP 628** [https://peps.python.org/pep-0628/] for details.)

New constants: `cmath.inf` and `cmath.nan` to match `math.inf` and `math.nan`, and also `cmath.infj` and `cmath.nanj` to match the format used by complex repr. (Contributed by Mark Dickinson in bpo-23229 [https://bugs.python.org/issue?@action=redirect&bpo=23229].)

collections

The new `Collection` abstract base class has been added to represent sized iterable container classes. (Contributed by Ivan Levkivskyi, docs by Neil Girdhar in bpo-27598 [https://bugs.python.org/issue?@action=redirect&bpo=27598].)

The new `Reversible` abstract base class represents iterable classes that also provide the `__reversed__()` method. (Contributed by Ivan Levkivskyi in bpo-25987 [https://bugs.python.org/issue?@action=redirect&bpo=25987].)

The new `AsyncGenerator` abstract base class represents asynchronous generators. (Contributed by Yury Selivanov in bpo-28720 [https://bugs.python.org/issue?@action=redirect&bpo=28720].)

The `namedtuple()` function now accepts an optional keyword argument *module*, which, when specified, is used for the `__module__` attribute of the returned named tuple class. (Contributed by Raymond Hettinger in bpo-17941 [https://bugs.python.org/issue?@action=redirect&bpo=17941].)

The *verbose* and *rename* arguments for `namedtuple()` are now keyword-only. (Contributed by Raymond Hettinger in bpo-25628 [https://bugs.python.org/issue?@action=redirect&bpo=25628].)

Recursive `collections.deque` instances can now be pickled. (Contributed by Serhiy Storchaka in bpo-26482 [https://bugs.python.org/issue?@action=redirect&bpo=26482].)

concurrent.futures

The `ThreadPoolExecutor` class constructor now accepts an optional *thread_name_prefix* argument to make it possible to customize the names of the threads created by the pool. (Contributed by Gregory P. Smith in bpo-27664 [https://bugs.python.org/issue?@action=redirect&bpo=27664].)

contextlib

The `contextlib.AbstractContextManager` class has been added to provide an abstract base class for context managers. It provides a sensible default implementation for `__enter__()`

which returns `self` and leaves `__exit__()` an abstract method. A matching class has been added to the **`typing`** module as **`typing.ContextManager`**. (Contributed by Brett Cannon in bpo-25609 [https://bugs.python.org/issue?@action=redirect&bpo=25609].)

datetime

The **`datetime`** and **`time`** classes have the new **`fold`** attribute used to disambiguate local time when necessary. Many functions in the **`datetime`** have been updated to support local time disambiguation. See Local Time Disambiguation section for more information. (Contributed by Alexander Belopolsky in bpo-24773 [https://bugs.python.org/issue?@action=redirect&bpo=24773].)

The **`datetime.strftime()`** and **`date.strftime()`** methods now support ISO 8601 date directives `%G`, `%u` and `%V`. (Contributed by Ashley Anderson in bpo-12006 [https://bugs.python.org/issue?@action=redirect&bpo=12006].)

The **`datetime.isoformat()`** function now accepts an optional *timespec* argument that specifies the number of additional components of the time value to include. (Contributed by Alessandro Cucci and Alexander Belopolsky in bpo-19475 [https://bugs.python.org/issue?@action=redirect&bpo=19475].)

The **`datetime.combine()`** now accepts an optional *tzinfo* argument. (Contributed by Alexander Belopolsky in bpo-27661 [https://bugs.python.org/issue?@action=redirect&bpo=27661].)

decimal

New **`Decimal.as_integer_ratio()`** method that returns a pair `(n, d)` of integers that represent the given **`Decimal`** instance as a fraction, in lowest terms and with a positive denominator:

```
>>> Decimal('-3.14').as_integer_ratio()
(-157, 50)
```

(Contributed by Stefan Krah amd Mark Dickinson in bpo-25928 [https://bugs.python.org/issue?@action=redirect&bpo=25928].)

distutils

The `default_format` attribute has been removed from `distutils.command.sdist.sdist` and the `formats` attribute defaults to `['gztar']`. Although not anticipated, any code relying on the presence of `default_format` may need to be adapted. See bpo-27819 [https://bugs.python.org/issue?@action=redirect&bpo=27819] for more details.

email

The new email API, enabled via the *policy* keyword to various constructors, is no longer provisional. The **`email`** documentation has been reorganized and rewritten to focus on the new API, while retaining the old documentation for the legacy API. (Contributed by R. David Murray in bpo-24277 [https://bugs.python.org/issue?@action=redirect&bpo=24277].)

The **`email.mime`** classes now all accept an optional *policy* keyword. (Contributed by Berker Peksag in bpo-27331 [https://bugs.python.org/issue?@action=redirect&bpo=27331].)

The **`DecodedGenerator`** now supports the *policy* keyword.

There is a new **`policy`** attribute, **`message_factory`**, that controls what class is used by default when the parser creates new message objects. For the **`email.policy.compat32`** policy this is **`Message`**, for the new policies it is **`EmailMessage`**. (Contributed by R. David Murray in bpo-20476 [https://bugs.python.org/issue?@action=redirect&bpo=20476].)

encodings

On Windows, added the `'oem'` encoding to use `CP_OEMCP`, and the `'ansi'` alias for the existing `'mbcs'` encoding, which uses the `CP_ACP` code page. (Contributed by Steve Dower in bpo-27959 [https://bugs.python.org/issue?@action=redirect&bpo=27959].)

enum

Two new enumeration base classes have been added to the **`enum`** module: **`Flag`** and **`IntFlag`**. Both are used to define constants that can be combined using the bitwise operators. (Contributed by Ethan Furman in bpo-23591 [https://bugs.python.org/issue?@action=redirect&bpo=23591].)

Many standard library modules have been updated to use the **`IntFlag`** class for their constants.

The new **`enum.auto`** value can be used to assign values to enum members automatically:

```
>>> from enum import Enum, auto
>>> class Color(Enum):
...     red = auto()
...     blue = auto()
...     green = auto()
...
>>> list(Color)
[<Color.red: 1>, <Color.blue: 2>, <Color.green: 3>]
```

faulthandler

On Windows, the **`faulthandler`** module now installs a handler for Windows exceptions: see **`faulthandler.enable()`**. (Contributed by Victor Stinner in bpo-23848 [https://bugs.python.org/issue?@action=redirect&bpo=23848].)

fileinput

`hook_encoded()` now supports the *errors* argument. (Contributed by Joseph Hackman in bpo-25788 [https://bugs.python.org/issue?@action=redirect&bpo=25788].)

hashlib

`hashlib` supports OpenSSL 1.1.0. The minimum recommend version is 1.0.2. (Contributed by Christian Heimes in bpo-26470 [https://bugs.python.org/issue?@action=redirect&bpo=26470].)

BLAKE2 hash functions were added to the module. `blake2b()` and `blake2s()` are always available and support the full feature set of BLAKE2. (Contributed by Christian Heimes in bpo-26798 [https://bugs.python.org/issue?@action=redirect&bpo=26798] based on code by Dmitry Chestnykh and Samuel Neves. Documentation written by Dmitry Chestnykh.)

The SHA-3 hash functions `sha3_224()`, `sha3_256()`, `sha3_384()`, `sha3_512()`, and SHAKE hash functions `shake_128()` and `shake_256()` were added. (Contributed by Christian Heimes in bpo-16113 [https://bugs.python.org/issue?@action=redirect&bpo=16113]. Keccak Code Package by Guido Bertoni, Joan Daemen, Michaël Peeters, Gilles Van Assche, and Ronny Van Keer.)

The password-based key derivation function `scrypt()` is now available with OpenSSL 1.1.0 and newer. (Contributed by Christian Heimes in bpo-27928 [https://bugs.python.org/issue?@action=redirect&bpo=27928].)

http.client

`HTTPConnection.request()` and `endheaders()` both now support chunked encoding request bodies. (Contributed by Demian

Brecht and Rolf Krahl in bpo-12319 [https://bugs.python.org/issue?@action=redirect&bpo=12319].)

idlelib and IDLE

The idlelib package is being modernized and refactored to make IDLE look and work better and to make the code easier to understand, test, and improve. Part of making IDLE look better, especially on Linux and Mac, is using ttk widgets, mostly in the dialogs. As a result, IDLE no longer runs with tcl/tk 8.4. It now requires tcl/tk 8.5 or 8.6. We recommend running the latest release of either.

‘Modernizing’ includes renaming and consolidation of idlelib modules. The renaming of files with partial uppercase names is similar to the renaming of, for instance, Tkinter and TkFont to tkinter and tkinter.font in 3.0. As a result, imports of idlelib files that worked in 3.5 will usually not work in 3.6. At least a module name change will be needed (see idlelib/README.txt), sometimes more. (Name changes contributed by Al Swiegart and Terry Reedy in bpo-24225 [https://bugs.python.org/issue?@action=redirect&bpo=24225]. Most idlelib patches since have been and will be part of the process.)

In compensation, the eventual result with be that some idlelib classes will be easier to use, with better APIs and docstrings explaining them. Additional useful information will be added to idlelib when available.

New in 3.6.2:

Multiple fixes for autocompletion. (Contributed by Louie Lu in bpo-15786 [https://bugs.python.org/issue?@action=redirect&bpo=15786].)

New in 3.6.3:

Module Browser (on the File menu, formerly called Class Browser), now displays nested functions and classes in addition to top-level functions and classes. (Contributed by Guilherme Polo, Cheryl Sabella, and Terry Jan Reedy in bpo-1612262 [https://bugs.python.org/issue?@action=redirect&bpo=1612262].)

The IDLE features formerly implemented as extensions have been reimplemented as normal features. Their settings have been moved from the Extensions tab to other dialog tabs. (Contributed by Charles Wohlganger and Terry Jan Reedy in bpo-27099 [https://bugs.python.org/issue?@action=redirect&bpo=27099].)

The Settings dialog (Options, Configure IDLE) has been partly rewritten to improve both appearance and function. (Contributed by Cheryl Sabella and Terry Jan Reedy in multiple issues.)

New in 3.6.4:

The font sample now includes a selection of non-Latin characters so that users can better see the effect of selecting a particular font. (Contributed by Terry Jan Reedy in bpo-13802 [https://bugs.python.org/issue?@action=redirect&bpo=13802].) The sample can be edited to include other characters. (Contributed by Serhiy Storchaka in bpo-31860 [https://bugs.python.org/issue?@action=redirect&bpo=31860].)

New in 3.6.6:

Editor code context option revised. Box displays all context lines up to maxlines. Clicking on a context line jumps the editor to that line. Context colors for custom themes is added to Highlights tab of Settings dialog. (Contributed by Cheryl Sabella and Terry Jan Reedy in bpo-33642 [https://bugs.python.org/issue?@action=redirect&bpo=33642], bpo-33768 [https://bugs.python.org/issue?@action=redirect&bpo=33768], and bpo-33679 [https://bugs.python.org/issue?@action=redirect&bpo=33679].)

On Windows, a new API call tells Windows that tk scales for DPI. On Windows 8.1+ or 10, with DPI compatibility properties of the Python binary unchanged, and a monitor resolution greater than 96 DPI, this should make text and lines sharper. It should otherwise have no effect. (Contributed by Terry Jan Reedy in bpo-33656 [https://bugs.python.org/issue?@action=redirect&bpo=33656].)

New in 3.6.7:

Output over N lines (50 by default) is squeezed down to a button. N can be changed in the PyShell section of the General page of the Settings dialog. Fewer, but possibly extra long, lines can be squeezed by right clicking on the output. Squeezed output can be expanded in place by double-clicking the button or into the clipboard or a separate window by right-clicking the button. (Contributed by Tal Einat in bpo-1529353 [https://bugs.python.org/issue?@action=redirect&bpo=1529353].)

importlib

Import now raises the new exception **`ModuleNotFoundError`** (subclass of **`ImportError`**) when it cannot find a module. Code that current checks for `ImportError` (in try-except) will still work. (Contributed by Eric Snow in bpo-15767 [https://bugs.python.org/issue?@action=redirect&bpo=15767].)

`importlib.util.LazyLoader` now calls **`create_module()`** on the wrapped loader, removing the restriction that **`importlib.machinery.BuiltinImporter`** and **`importlib.machinery.ExtensionFileLoader`** couldn't be used with **`importlib.util.LazyLoader`**.

`importlib.util.cache_from_source()`, `importlib.util.source_from_cache()`, and `importlib.util.spec_from_file_location()` now accept a path-like object.

inspect

The `inspect.signature()` function now reports the implicit `.0` parameters generated by the compiler for comprehension and generator expression scopes as if they were positional-only parameters called `implicit0`. (Contributed by Jelle Zijlstra in bpo-19611 [https://bugs.python.org/issue?@action=redirect&bpo=19611].)

To reduce code churn when upgrading from Python 2.7 and the legacy **`inspect.getargspec()`** API, the previously documented deprecation of `inspect.getfullargspec()` has been reversed. While this function is convenient for single/source Python 2/3 code bases, the richer `inspect.signature()` interface remains the recommended approach for new code. (Contributed by Nick Coghlan in bpo-27172 [https://bugs.python.org/issue?@action=redirect&bpo=27172])

json

`json.load()` and `json.loads()` now support binary input. Encoded JSON should be represented using either UTF-8, UTF-16, or UTF-32. (Contributed by Serhiy Storchaka in bpo-17909 [https://bugs.python.org/issue?@action=redirect&bpo=17909].)

logging

The new `WatchedFileHandler.reopenIfNeeded()` method has been added to add the ability to check if the log file needs to be reopened. (Contributed by Marian Horban in bpo-24884 [https://bugs.python.org/issue?@action=redirect&bpo=24884].)

math

The tau (τ) constant has been added to the `math` and `cmath` modules. (Contributed by Lisa Roach in bpo-12345 [https://bugs.python.org/issue?@action=redirect&bpo=12345], see **PEP 628** [https://peps.python.org/pep-0628/] for details.)

multiprocessing

Proxy Objects returned by `multiprocessing.Manager()` can now be nested. (Contributed by Davin Potts in bpo-6766 [https://bugs.python.org/issue?@action=redirect&bpo=6766].)

os

See the summary of PEP 519 for details on how the `os` and `os.path` modules now support path-like objects.

`scandir()` now supports `bytes` paths on Windows.

A new `close()` method allows explicitly closing a `scandir()` iterator. The `scandir()` iterator now supports the context manager protocol. If a `scandir()` iterator is neither exhausted nor explicitly closed a `ResourceWarning` will be emitted in its destructor. (Contributed by Serhiy Storchaka in bpo-25994 [https://bugs.python.org/issue?@action=redirect&bpo=25994].)

On Linux, `os.urandom()` now blocks until the system urandom entropy pool is initialized to increase the security. See the **PEP 524** [https://peps.python.org/pep-0524/] for the rationale.

The Linux `getrandom()` syscall (get random bytes) is now exposed as the new `os.getrandom()` function. (Contributed by Victor Stinner, part of the **PEP 524** [https://peps.python.org/pep-0524/])

pathlib

`pathlib` now supports path-like objects. (Contributed by Brett Cannon in bpo-27186 [https://bugs.python.org/issue?@action=redirect&bpo=27186].)

See the summary of PEP 519 for details.

pdb

The **`Pdb`** class constructor has a new optional *readrc* argument to control whether `.pdbrc` files should be read.

pickle

Objects that need `__new__` called with keyword arguments can now be pickled using pickle protocols older than protocol version 4. Protocol version 4 already supports this case. (Contributed by Serhiy Storchaka in bpo-24164 [https://bugs.python.org/issue?@action=redirect&bpo=24164].)

pickletools

`pickletools.dis()` now outputs the implicit memo index for the `MEMOIZE` opcode. (Contributed by Serhiy Storchaka in bpo-25382 [https://bugs.python.org/issue?@action=redirect&bpo=25382].)

pydoc

The **`pydoc`** module has learned to respect the `MANPAGER` environment variable. (Contributed by Matthias Klose in bpo-8637 [https://bugs.python.org/issue?@action=redirect&bpo=8637].)

`help()` and **`pydoc`** can now list named tuple fields in the order they were defined rather than alphabetically. (Contributed by Raymond Hettinger in bpo-24879 [https://bugs.python.org/issue?@action=redirect&bpo=24879].)

random

The new **`choices()`** function returns a list of elements of specified size from the given population with optional weights. (Contributed by Raymond Hettinger in bpo-18844 [https://bugs.python.org/issue?@action=redirect&bpo=18844].)

re

Added support of modifier spans in regular expressions. Examples: `'(?i:p)ython'` matches `'python'` and `'Python'`, but not `'PYTHON'`; `'(?i)g(?-i:v)r'` matches `'GvR'` and `'gvr'`, but not `'GVR'`. (Contributed by Serhiy Storchaka in bpo-433028 [https://bugs.python.org/issue?@action=redirect&bpo=433028].)

Match object groups can be accessed by `__getitem__`, which is equivalent to `group()`. So `mo['name']` is now equivalent to `mo.group('name')`. (Contributed by Eric Smith in bpo-24454 [https://bugs.python.org/issue?@action=redirect&bpo=24454].)

`Match` objects now support **`index-like objects`** as group indices. (Contributed by Jeroen Demeyer and Xiang Zhang in bpo-27177 [https://bugs.python.org/issue?@action=redirect&bpo=27177].)

readline

Added **`set_auto_history()`** to enable or disable automatic addition of input to the history list. (Contributed by Tyler Crompton in bpo-26870 [https://bugs.python.org/issue?@action=redirect&bpo=26870].)

rlcompleter

Private and special attribute names now are omitted unless the prefix starts with underscores. A space or a colon is added after some completed keywords. (Contributed by Serhiy Storchaka in bpo-25011 [https://bugs.python.org/issue?@action=redirect&bpo=25011] and bpo-25209 [https://bugs.python.org/issue?@action=redirect&bpo=25209].)

shlex

The **`shlex`** has much improved shell compatibility through the new *punctuation_chars* argument to control which characters are treated as punctuation. (Contributed by Vinay Sajip in bpo-1521950 [https://bugs.python.org/issue?@action=redirect&bpo=1521950].)

site

When specifying paths to add to **`sys.path`** in a `.pth` file, you may now specify file paths on top of directories (e.g. zip files). (Contributed by Wolfgang Langner in bpo-26587 [https://bugs.python.org/issue?@action=redirect&bpo=26587]).

sqlite3

`sqlite3.Cursor.lastrowid` now supports the `REPLACE` statement. (Contributed by Alex LordThorsen in bpo-16864 [https://bugs.python.org/issue?@action=redirect&bpo=16864].)

socket

The **`ioctl()`** function now supports the **`SIO_LOOPBACK_FAST_PATH`** control code. (Contributed by Daniel Stokes in bpo-26536 [https://bugs.python.org/issue?@action=redirect&bpo=26536].)

The **`getsockopt()`** constants `SO_DOMAIN`, `SO_PROTOCOL`, `SO_PEERSEC`, and `SO_PASSSEC` are now supported. (Contributed by Christian

Heimes in bpo-26907 [https://bugs.python.org/issue?@action=redirect&bpo=26907].)

The **`setsockopt()`** now supports the `setsockopt(level, optname, None, optlen: int)` form. (Contributed by Christian Heimes in bpo-27744 [https://bugs.python.org/issue?@action=redirect&bpo=27744].)

The socket module now supports the address family **`AF_ALG`** to interface with Linux Kernel crypto API. `ALG_*`, `SOL_ALG` and **`sendmsg_afalg()`** were added. (Contributed by Christian Heimes in bpo-27744 [https://bugs.python.org/issue?@action=redirect&bpo=27744] with support from Victor Stinner.)

New Linux constants `TCP_USER_TIMEOUT` and `TCP_CONGESTION` were added. (Contributed by Omar Sandoval, bpo-26273 [https://bugs.python.org/issue?@action=redirect&bpo=26273]).

socketserver

Servers based on the **`socketserver`** module, including those defined in **`http.server`**, **`xmlrpc.server`** and **`wsgiref.simple_server`**, now support the context manager protocol. (Contributed by Aviv Palivoda in bpo-26404 [https://bugs.python.org/issue?@action=redirect&bpo=26404].)

The **`wfile`** attribute of **`StreamRequestHandler`** classes now implements the **`io.BufferedIOBase`** writable interface. In particular, calling **`write()`** is now guaranteed to send the data in full. (Contributed by Martin Panter in bpo-26721 [https://bugs.python.org/issue?@action=redirect&bpo=26721].)

ssl

`ssl` supports OpenSSL 1.1.0. The minimum recommend version is 1.0.2. (Contributed by Christian Heimes in bpo-26470 [https://bugs.python.org/issue?@action=redirect&bpo=26470].)

3DES has been removed from the default cipher suites and ChaCha20 Poly1305 cipher suites have been added. (Contributed by Christian Heimes in bpo-27850 [https://bugs.python.org/issue?@action=redirect&bpo=27850] and bpo-27766 [https://bugs.python.org/issue?@action=redirect&bpo=27766].)

`SSLContext` has better default configuration for options and ciphers. (Contributed by Christian Heimes in bpo-28043 [https://bugs.python.org/issue?@action=redirect&bpo=28043].)

SSL session can be copied from one client-side connection to another with the new **`SSLSession`** class. TLS session resumption can speed up the initial handshake, reduce latency and improve performance (Contributed by Christian Heimes in bpo-19500 [https://bugs.python.org/issue?@action=redirect&bpo=19500] based on a draft by Alex Warhawk.)

The new **`get_ciphers()`** method can be used to get a list of enabled ciphers in order of cipher priority.

All constants and flags have been converted to **`IntEnum`** and **`IntFlag`**. (Contributed by Christian Heimes in bpo-28025 [https://bugs.python.org/issue?@action=redirect&bpo=28025].)

Server and client-side specific TLS protocols for **`SSLContext`** were added. (Contributed by Christian Heimes in bpo-28085 [https://bugs.python.org/issue?@action=redirect&bpo=28085].)

Added **`ssl.SSLContext.post_handshake_auth`** to enable and **`ssl.SSLSocket.verify_client_post_handshake()`** to initiate TLS 1.3 post-handshake authentication. (Contributed by Christian Heimes in gh-78851 [https://github.com/python/cpython/issues/78851].)

statistics

A new **`harmonic_mean()`** function has been added. (Contributed by Steven D’Aprano in bpo-27181 [https://bugs.python.org/issue?

@action=redirect&bpo=27181].)

struct

[`struct`](#) now supports IEEE 754 half-precision floats via the `'e'` format specifier. (Contributed by Eli Stevens, Mark Dickinson in bpo-11734 [https://bugs.python.org/issue?@action=redirect&bpo=11734].)

subprocess

`subprocess.Popen` destructor now emits a **`ResourceWarning`** warning if the child process is still running. Use the context manager protocol (`with proc: ...`) or explicitly call the **`wait()`** method to read the exit status of the child process. (Contributed by Victor Stinner in bpo-26741 [https://bugs.python.org/issue?@action=redirect&bpo=26741].)

The **`subprocess.Popen`** constructor and all functions that pass arguments through to it now accept *encoding* and *errors* arguments. Specifying either of these will enable text mode for the *stdin*, *stdout* and *stderr* streams. (Contributed by Steve Dower in bpo-6135 [https://bugs.python.org/issue?@action=redirect&bpo=6135].)

sys

The new **`getfilesystemencodeerrors()`** function returns the name of the error mode used to convert between Unicode filenames and bytes filenames. (Contributed by Steve Dower in bpo-27781 [https://bugs.python.org/issue?@action=redirect&bpo=27781].)

On Windows the return value of the **`getwindowsversion()`** function now includes the *platform_version* field which contains the accurate major version, minor version and build number of the current operating system, rather than the version that is being emulated for the process (Contributed by Steve Dower in bpo-27932 [https://bugs.python.org/issue?@action=redirect&bpo=27932].)

telnetlib

`telnetlib.Telnet` is now a context manager (contributed by Stéphane Wirtel in bpo-25485 [https://bugs.python.org/issue?@action=redirect&bpo=25485]).

time

The `struct_time` attributes `tm_gmtoff` and `tm_zone` are now available on all platforms.

timeit

The new `Timer.autorange()` convenience method has been added to call `Timer.timeit()` repeatedly so that the total run time is greater or equal to 200 milliseconds. (Contributed by Steven D'Aprano in bpo-6422 [https://bugs.python.org/issue?@action=redirect&bpo=6422].)

`timeit` now warns when there is substantial (4x) variance between best and worst times. (Contributed by Serhiy Storchaka in bpo-23552 [https://bugs.python.org/issue?@action=redirect&bpo=23552].)

tkinter

Added methods `Variable.trace_add()`, `Variable.trace_remove()` and `trace_info()` in the `tkinter.Variable` class. They replace old methods `trace_variable()`, `trace()`, `trace_vdelete()` and `trace_vinfo()` that use obsolete Tcl commands and might not work in future versions of Tcl. (Contributed by Serhiy

Storchaka in bpo-22115 [https://bugs.python.org/issue?@action=redirect&bpo=22115]).

traceback

Both the traceback module and the interpreter's builtin exception display now abbreviate long sequences of repeated lines in tracebacks as shown in the following example:

```
>>> def f(): f()
...
>>> f()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 1, in f
  File "<stdin>", line 1, in f
  File "<stdin>", line 1, in f
  [Previous line repeated 995 more times]
RecursionError: maximum recursion depth exceeded
```

(Contributed by Emanuel Barry in bpo-26823 [https://bugs.python.org/issue?@action=redirect&bpo=26823].)

tracemalloc

The `tracemalloc` module now supports tracing memory allocations in multiple different address spaces.

The new `DomainFilter` filter class has been added to filter block traces by their address space (domain).

(Contributed by Victor Stinner in bpo-26588 [https://bugs.python.org/issue?@action=redirect&bpo=26588].)

typing

Since the `typing` module is provisional, all changes introduced in Python 3.6 have also been backported to Python 3.5.x.

The `typing` module has a much improved support for generic type aliases. For example `Dict[str, Tuple[S, T]]` is now a valid type annotation. (Contributed by Guido van Rossum in Github #195 [https://github.com/python/typing/pull/195].)

The `typing.ContextManager` class has been added for representing `contextlib.AbstractContextManager`. (Contributed by Brett Cannon in bpo-25609 [https://bugs.python.org/issue?@action=redirect&bpo=25609].)

The `typing.Collection` class has been added for representing `collections.abc.Collection`. (Contributed by Ivan Levkivskyi in bpo-27598 [https://bugs.python.org/issue?@action=redirect&bpo=27598].)

The `typing.ClassVar` type construct has been added to mark class variables. As introduced in **PEP 526** [https://peps.python.org/pep-0526/], a variable annotation wrapped in ClassVar indicates that a given attribute is intended to be used as a class variable and should not be set on instances of that class. (Contributed by Ivan Levkivskyi in Github #280 [https://github.com/python/typing/pull/280].)

A new `TYPE_CHECKING` constant that is assumed to be `True` by the static type checkers, but is `False` at runtime. (Contributed by Guido van Rossum in Github #230 [https://github.com/python/typing/issues/230].)

A new `NewType()` helper function has been added to create lightweight distinct types for annotations:

```
from typing import NewType
```

```
UserId = NewType('UserId', int)
some_id = UserId(524313)
```

The static type checker will treat the new type as if it were a subclass of the original type. (Contributed by Ivan Levkivskyi in Github #189 [https://github.com/python/typing/issues/189].)

unicodedata

The **`unicodedata`** module now uses data from Unicode 9.0.0 [https://unicode.org/versions/Unicode9.0.0/]. (Contributed by Benjamin Peterson.)

unittest.mock

The **`Mock`** class has the following improvements:

- Two new methods, **`Mock.assert_called()`** and **`Mock.assert_called_once()`** to check if the mock object was called. (Contributed by Amit Saha in bpo-26323 [https://bugs.python.org/issue?@action=redirect&bpo=26323].)
- The **`Mock.reset_mock()`** method now has two optional keyword only arguments: *return_value* and *side_effect*. (Contributed by Kushal Das in bpo-21271 [https://bugs.python.org/issue?@action=redirect&bpo=21271].)

urllib.request

If a HTTP request has a file or iterable body (other than a bytes object) but no `Content-Length` header, rather than throwing an error, **`AbstractHTTPHandler`** now falls back to use chunked transfer encoding. (Contributed by Demian Brecht and Rolf Krahl in bpo-12319 [https://bugs.python.org/issue?@action=redirect&bpo=12319].)

urllib.robotparser

`RobotFileParser` now supports the `Crawl-delay` and `Request-rate` extensions. (Contributed by Nikolay Bogoychev in bpo-16099 [https://bugs.python.org/issue?@action=redirect&bpo=16099].)

venv

`venv` accepts a new parameter `--prompt`. This parameter provides an alternative prefix for the virtual environment. (Proposed by Łukasz Balcerzak and ported to 3.6 by Stéphane Wirtel in bpo-22829 [https://bugs.python.org/issue?@action=redirect&bpo=22829].)

warnings

A new optional *source* parameter has been added to the **`warnings.warn_explicit()`** function: the destroyed object which emitted a **`ResourceWarning`**. A *source* attribute has also been added to **`warnings.WarningMessage`** (contributed by Victor Stinner in bpo-26568 [https://bugs.python.org/issue?@action=redirect&bpo=26568] and bpo-26567 [https://bugs.python.org/issue?@action=redirect&bpo=26567]).

When a **`ResourceWarning`** warning is logged, the **`tracemalloc`** module is now used to try to retrieve the traceback where the destroyed object was allocated.

Example with the script `example.py`:

```
import warnings

def func():
    return open(__file__)

f = func()
f = None
```

Output of the command `python3.6 -Wd -X tracemalloc=5 example.py`:

```
example.py:7: ResourceWarning: unclosed file <_io.TextIOWrapper
name='example.py' mode='r' encoding='UTF-8'>
  f = None
Object allocated at (most recent call first):
  File "example.py", lineno 4
    return open(__file__)
  File "example.py", lineno 6
    f = func()
```

The “Object allocated at” traceback is new and is only displayed if **`tracemalloc`** is tracing Python memory allocations and if the **`warnings`** module was already imported.

winreg

Added the 64-bit integer type **`REG_QWORD`**. (Contributed by Clement Rouault in bpo-23026 [https://bugs.python.org/issue?@action=redirect&bpo=23026].)

winsound

Allowed keyword arguments to be passed to **`Beep`**, **`MessageBeep`**, and **`PlaySound`** (bpo-27982 [https://bugs.python.org/issue?@action=redirect&bpo=27982]).

xmlrpc.client

The **`xmlrpc.client`** module now supports unmarshalling additional data types used by the Apache XML-RPC implementation for numerics and `None`. (Contributed by Serhiy Storchaka in bpo-26885 [https://bugs.python.org/issue?@action=redirect&bpo=26885].)

zipfile

A new `ZipInfo.from_file()` class method allows making a `ZipInfo` instance from a filesystem file. A new `ZipInfo.is_dir()` method can be used to check if the `ZipInfo` instance represents a directory. (Contributed by Thomas Kluyver in bpo-26039 [https://bugs.python.org/issue?@action=redirect&bpo=26039].)

The `ZipFile.open()` method can now be used to write data into a ZIP file, as well as for extracting data. (Contributed by Thomas Kluyver in bpo-26039 [https://bugs.python.org/issue?@action=redirect&bpo=26039].)

zlib

The `compress()` and `decompress()` functions now accept keyword arguments. (Contributed by Aviv Palivoda in bpo-26243 [https://bugs.python.org/issue?@action=redirect&bpo=26243] and Xiang Zhang in bpo-16764 [https://bugs.python.org/issue?@action=redirect&bpo=16764] respectively.)

Optimizations

- The Python interpreter now uses a 16-bit wordcode instead of bytecode which made a number of opcode optimizations possible. (Contributed by Demur Rumed with input and reviews from Serhiy Storchaka and Victor Stinner in bpo-26647 [https://bugs.python.org/issue?@action=redirect&bpo=26647] and bpo-28050 [https://bugs.python.org/issue?@action=redirect&bpo=28050].)
- The `asyncio.Future` class now has an optimized C implementation. (Contributed by Yury Selivanov and INADA Naoki in bpo-26081 [https://bugs.python.org/issue?@action=redirect&bpo=26081].)
- The `asyncio.Task` class now has an optimized C implementation. (Contributed by Yury Selivanov in bpo-28544 [https://bugs.python.org/issue?@action=redirect&bpo=28544].)

- Various implementation improvements in the **`typing`** module (such as caching of generic types) allow up to 30 times performance improvements and reduced memory footprint.
- The ASCII decoder is now up to 60 times as fast for error handlers `surrogateescape`, `ignore` and `replace` (Contributed by Victor Stinner in bpo-24870 [https://bugs.python.org/issue?@action=redirect&bpo=24870]).
- The ASCII and the Latin1 encoders are now up to 3 times as fast for the error handler `surrogateescape` (Contributed by Victor Stinner in bpo-25227 [https://bugs.python.org/issue?@action=redirect&bpo=25227]).
- The UTF-8 encoder is now up to 75 times as fast for error handlers `ignore`, `replace`, `surrogateescape`, `surrogatepass` (Contributed by Victor Stinner in bpo-25267 [https://bugs.python.org/issue?@action=redirect&bpo=25267]).
- The UTF-8 decoder is now up to 15 times as fast for error handlers `ignore`, `replace` and `surrogateescape` (Contributed by Victor Stinner in bpo-25301 [https://bugs.python.org/issue?@action=redirect&bpo=25301]).
- `bytes % args` is now up to 2 times faster. (Contributed by Victor Stinner in bpo-25349 [https://bugs.python.org/issue?@action=redirect&bpo=25349]).
- `bytearray % args` is now between 2.5 and 5 times faster. (Contributed by Victor Stinner in bpo-25399 [https://bugs.python.org/issue?@action=redirect&bpo=25399]).
- Optimize **`bytes.fromhex()`** and **`bytearray.fromhex()`**: they are now between 2x and 3.5x faster. (Contributed by Victor Stinner in bpo-25401 [https://bugs.python.org/issue?@action=redirect&bpo=25401]).
- Optimize `bytes.replace(b'', b'.')` and `bytearray.replace(b'', b'.')`: up to 80% faster. (Contributed by Josh Snider in bpo-26574 [https://bugs.python.org/issue?@action=redirect&bpo=26574]).
- Allocator functions of the **`PyMem_Malloc()`** domain (**`PYMEM_DOMAIN_MEM`**) now use the pymalloc memory

allocator instead of `malloc()` function of the C library. The pymalloc allocator is optimized for objects smaller or equal to 512 bytes with a short lifetime, and use `malloc()` for larger memory blocks. (Contributed by Victor Stinner in bpo-26249 [https://bugs.python.org/issue?@action=redirect&bpo=26249]).

- `pickle.load()` and `pickle.loads()` are now up to 10% faster when deserializing many small objects (Contributed by Victor Stinner in bpo-27056 [https://bugs.python.org/issue?@action=redirect&bpo=27056]).
- Passing keyword arguments to a function has an overhead in comparison with passing positional arguments. Now in extension functions implemented with using Argument Clinic this overhead is significantly decreased. (Contributed by Serhiy Storchaka in bpo-27574 [https://bugs.python.org/issue?@action=redirect&bpo=27574]).
- Optimized `glob()` and `iglob()` functions in the `glob` module; they are now about 3–6 times faster. (Contributed by Serhiy Storchaka in bpo-25596 [https://bugs.python.org/issue?@action=redirect&bpo=25596]).
- Optimized globbing in `pathlib` by using `os.scandir()`; it is now about 1.5–4 times faster. (Contributed by Serhiy Storchaka in bpo-26032 [https://bugs.python.org/issue?@action=redirect&bpo=26032]).
- `xml.etree.ElementTree` parsing, iteration and deepcopy performance has been significantly improved. (Contributed by Serhiy Storchaka in bpo-25638 [https://bugs.python.org/issue?@action=redirect&bpo=25638], bpo-25873 [https://bugs.python.org/issue?@action=redirect&bpo=25873], and bpo-25869 [https://bugs.python.org/issue?@action=redirect&bpo=25869].)
- Creation of `fractions.Fraction` instances from floats and decimals is now 2 to 3 times faster. (Contributed by Serhiy Storchaka in bpo-25971 [https://bugs.python.org/issue?@action=redirect&bpo=25971].)

Build and C API Changes

- Python now requires some C99 support in the toolchain to build. Most notably, Python now uses standard integer types and macros in place of custom macros like `PY_LONG_LONG`. For more information, see **PEP 7** [https://peps.python.org/pep-0007/] and bpo-17884 [https://bugs.python.org/issue?@action=redirect&bpo=17884].
- Cross-compiling CPython with the Android NDK and the Android API level set to 21 (Android 5.0 Lollipop) or greater runs successfully. While Android is not yet a supported platform, the Python test suite runs on the Android emulator with only about 16 tests failures. See the Android meta-issue bpo-26865 [https://bugs.python.org/issue?@action=redirect&bpo=26865].
- The `--enable-optimizations` configure flag has been added. Turning it on will activate expensive optimizations like PGO. (Original patch by Alecsandru Patrascu of Intel in bpo-26359 [https://bugs.python.org/issue?@action=redirect&bpo=26359].)
- The GIL must now be held when allocator functions of **`PYMEM_DOMAIN_OBJ`** (ex: **`PyObject_Malloc()`**) and **`PYMEM_DOMAIN_MEM`** (ex: **`PyMem_Malloc()`**) domains are called.
- New **`Py_FinalizeEx()`** API which indicates if flushing buffered data failed. (Contributed by Martin Panter in bpo-5319 [https://bugs.python.org/issue?@action=redirect&bpo=5319].)
- **`PyArg_ParseTupleAndKeywords()`** now supports positional-only parameters. Positional-only parameters are defined by empty names. (Contributed by Serhiy Storchaka in bpo-26282 [https://bugs.python.org/issue?@action=redirect&bpo=26282]).
- `PyTraceback_Print` method now abbreviates long sequences of repeated lines as `"[Previous line repeated {count} more times]"`. (Contributed by Emanuel Barry in

bpo-26823 [https://bugs.python.org/issue?@action=redirect&bpo=26823].)

- The new **`PyErr_SetImportErrorSubclass()`** function allows for specifying a subclass of **`ImportError`** to raise. (Contributed by Eric Snow in bpo-15767 [https://bugs.python.org/issue?@action=redirect&bpo=15767].)
- The new **`PyErr_ResourceWarning()`** function can be used to generate a **`ResourceWarning`** providing the source of the resource allocation. (Contributed by Victor Stinner in bpo-26567 [https://bugs.python.org/issue?@action=redirect&bpo=26567].)
- The new **`PyOS_FSPath()`** function returns the file system representation of a path-like object. (Contributed by Brett Cannon in bpo-27186 [https://bugs.python.org/issue?@action=redirect&bpo=27186].)
- The **`PyUnicode_FSConverter()`** and **`PyUnicode_FSDecoder()`** functions will now accept path-like objects.

Other Improvements

- When **`--version`** (short form: **`-V`**) is supplied twice, Python prints **`sys.version`** for detailed information.

```
$ ./python -VV
Python 3.6.0b4+ (3.6:223967b49e49+, Nov 21 2016, 20:55:04)
[GCC 4.2.1 Compatible Apple LLVM 8.0.0 (clang-800.0.42.1)]
```

Deprecated

New Keywords

`async` and `await` are not recommended to be used as variable, class, function or module names. Introduced by **PEP 492** [https://peps.python.org/pep-0492/] in Python 3.5, they will become proper keywords in Python 3.7. Starting in Python 3.6, the use of `async` or `await` as names will generate a **`DeprecationWarning`**.

Deprecated Python behavior

Raising the **`StopIteration`** exception inside a generator will now generate a **`DeprecationWarning`**, and will trigger a **`RuntimeError`** in Python 3.7. See PEP 479: Change StopIteration handling inside generators for details.

The **`__aiter__()`** method is now expected to return an asynchronous iterator directly instead of returning an awaitable as previously. Doing the former will trigger a **`DeprecationWarning`**. Backward compatibility will be removed in Python 3.7. (Contributed by Yury Selivanov in bpo-27243 [https://bugs.python.org/issue?@action=redirect&bpo=27243].)

A backslash-character pair that is not a valid escape sequence now generates a **`DeprecationWarning`**. Although this will eventually become a **`SyntaxError`**, that will not be for several Python releases. (Contributed by Emanuel Barry in bpo-27364 [https://bugs.python.org/issue?@action=redirect&bpo=27364].)

When performing a relative import, falling back on `__name__` and `__path__` from the calling module when `__spec__` or `__package__` are not defined now raises an **`ImportWarning`**. (Contributed by Rose Ames in bpo-25791 [https://bugs.python.org/issue?@action=redirect&bpo=25791].)

Deprecated Python modules, functions and methods

asynchat

The **`asynchat`** has been deprecated in favor of **`asyncio`**. (Contributed by Mariatta in bpo-25002 [https://bugs.python.org/issue?@action=redirect&bpo=25002].)

asyncore

The **`asyncore`** has been deprecated in favor of **`asyncio`**. (Contributed by Mariatta in bpo-25002 [https://bugs.python.org/issue?@action=redirect&bpo=25002].)

dbm

Unlike other **`dbm`** implementations, the **`dbm.dumb`** module creates databases with the `'rw'` mode and allows modifying the database opened with the `'r'` mode. This behavior is now deprecated and will be removed in 3.8. (Contributed by Serhiy Storchaka in bpo-21708 [https://bugs.python.org/issue?@action=redirect&bpo=21708].)

distutils

The undocumented `extra_path` argument to the `distutils.Distribution` constructor is now considered deprecated and will raise a warning if set. Support for this parameter will be removed in a future Python release. See bpo-27919 [https://bugs.python.org/issue?@action=redirect&bpo=27919] for details.

grp

The support of non-integer arguments in **`getgrgid()`** has been deprecated. (Contributed by Serhiy Storchaka in bpo-26129 [https://bugs.python.org/issue?@action=redirect&bpo=26129].)

importlib

The `importlib.machinery.SourceFileLoader.load_module()` and `importlib.machinery.SourcelessFileLoader.load_module()` methods are now deprecated. They were the only remaining implementations of `importlib.abc.Loader.load_module()` in `importlib` that had not been deprecated in previous versions of Python in favour of `importlib.abc.Loader.exec_module()`.

The `importlib.machinery.WindowsRegistryFinder` class is now deprecated. As of 3.6.0, it is still added to `sys.meta_path` by default (on Windows), but this may change in future releases.

os

Undocumented support of general bytes-like objects as paths in `os` functions, `compile()` and similar functions is now deprecated. (Contributed by Serhiy Storchaka in bpo-25791 [https://bugs.python.org/issue?@action=redirect&bpo=25791] and bpo-26754 [https://bugs.python.org/issue?@action=redirect&bpo=26754].)

re

Support for inline flags `(?letters)` in the middle of the regular expression has been deprecated and will be removed in a future Python version. Flags at the start of a regular expression are still allowed. (Contributed by Serhiy Storchaka in bpo-22493 [https://bugs.python.org/issue?@action=redirect&bpo=22493].)

ssl

OpenSSL 0.9.8, 1.0.0 and 1.0.1 are deprecated and no longer supported. In the future the `ssl` module will require at least OpenSSL 1.0.2 or 1.1.0.

SSL-related arguments like `certfile`, `keyfile` and `check_hostname` in **`ftplib`**, **`http.client`**, **`imaplib`**, **`poplib`**, and **`smtplib`** have been deprecated in favor of `context`. (Contributed by Christian Heimes in bpo-28022 [https://bugs.python.org/issue?@action=redirect&bpo=28022].)

A couple of protocols and functions of the **`ssl`** module are now deprecated. Some features will no longer be available in future versions of OpenSSL. Other features are deprecated in favor of a different API. (Contributed by Christian Heimes in bpo-28022 [https://bugs.python.org/issue?@action=redirect&bpo=28022] and bpo-26470 [https://bugs.python.org/issue?@action=redirect&bpo=26470].)

tkinter

The **`tkinter.tix`** module is now deprecated. **`tkinter`** users should use **`tkinter.ttk`** instead.

venv

The `pyvenv` script has been deprecated in favour of `python3 -m venv`. This prevents confusion as to what Python interpreter `pyvenv` is connected to and thus what Python interpreter will be used by the virtual environment. (Contributed by Brett Cannon in bpo-25154 [https://bugs.python.org/issue?@action=redirect&bpo=25154].)

xml

- As mitigation against DTD and external entity retrieval, the **`xml.dom.minidom`** and **`xml.sax`** modules no longer process external entities by default. (Contributed by Christian Heimes in gh-61441 [https://github.com/python/cpython/issues/61441].)

Deprecated functions and types of the C API

Undocumented functions **`PyUnicode_AsEncodedObject()`**, **`PyUnicode_AsDecodedObject()`**, **`PyUnicode_AsEncodedUnicode()`** and **`PyUnicode_AsDecodedUnicode()`** are deprecated now. Use the generic codec based API instead.

Deprecated Build Options

The `--with-system-ffi` configure flag is now on by default on non-macOS UNIX platforms. It may be disabled by using `--without-system-ffi`, but using the flag is deprecated and will not be accepted in Python 3.7. macOS is unaffected by this change. Note that many OS distributors already use the `--with-system-ffi` flag when building their system Python.

Removed

API and Feature Removals

- Unknown escapes consisting of `'\'` and an ASCII letter in regular expressions will now cause an error. In replacement templates for **`re.sub()`** they are still allowed, but deprecated. The **`re.LOCALE`** flag can now only be used with binary patterns.
- `inspect.getmoduleinfo()` was removed (was deprecated since CPython 3.3). **`inspect.getmodulename()`** should be used for obtaining the module name for a given path. (Contributed by Yury Selivanov in bpo-13248 [https://bugs.python.org/issue?@action=redirect&bpo=13248].)
- `traceback.Ignore` class and `traceback.usage`, `traceback.modname`, `traceback.fullmodname`, `traceback.find_lines_from_code`, `traceback.find_lines`, `traceback.find_strings`, `traceback.find_executable_lines` methods were removed from the **`traceback`** module. They

were undocumented methods deprecated since Python 3.2 and equivalent functionality is available from private methods.
- The `tk_menuBar()` and `tk_bindForTraversal()` dummy methods in **`tkinter`** widget classes were removed (corresponding Tk commands were obsolete since Tk 4.0).
- The **`open()`** method of the **`zipfile.ZipFile`** class no longer supports the `'U'` mode (was deprecated since Python 3.4). Use **`io.TextIOWrapper`** for reading compressed text files in universal newlines mode.
- The undocumented `IN`, `CDROM`, `DLFCN`, `TYPES`, `CDIO`, and `STROPTS` modules have been removed. They had been available in the platform specific `Lib/plat-*/` directories, but were chronically out of date, inconsistently available across platforms, and unmaintained. The script that created these modules is still available in the source distribution at Tools/scripts/h2py.py [https://github.com/python/cpython/blob/v3.6.15/Tools/scripts/h2py.py].
- The deprecated `asynchat.fifo` class has been removed.

Porting to Python 3.6

This section lists previously described changes and other bugfixes that may require changes to your code.

Changes in 'python' Command Behavior

- The output of a special Python build with defined `COUNT_ALLOCS`, `SHOW_ALLOC_COUNT` or `SHOW_TRACK_COUNT` macros is now off by default. It can be re-enabled using the `-X showalloccount` option. It now outputs to `stderr` instead of `stdout`. (Contributed by Serhiy Storchaka in bpo-23034 [https://bugs.python.org/issue?@action=redirect&bpo=23034].)

Changes in the Python API

- `open()` will no longer allow combining the `'U'` mode flag with `'+'`. (Contributed by Jeff Balogh and John O'Connor in bpo-2091 [https://bugs.python.org/issue?@action=redirect&bpo=2091].)
- `sqlite3` no longer implicitly commits an open transaction before DDL statements.
- On Linux, `os.urandom()` now blocks until the system urandom entropy pool is initialized to increase the security.
- When `importlib.abc.Loader.exec_module()` is defined, `importlib.abc.Loader.create_module()` must also be defined.
- `PyErr_SetImportError()` now sets `TypeError` when its **msg** argument is not set. Previously only `NULL` was returned.
- The format of the `co_lnotab` attribute of code objects changed to support a negative line number delta. By default, Python does not emit bytecode with a negative line number delta. Functions using `frame.f_lineno`, `PyFrame_GetLineNumber()` or `PyCode_Addr2Line()` are not affected. Functions directly decoding **`co_lnotab`** should be updated to use a signed 8-bit integer type for the line number delta, but this is only required to support applications using a negative line number delta. See `Objects/lnotab_notes.txt` for the **`co_lnotab`** format and how to decode it, and see the **PEP 511** [https://peps.python.org/pep-0511/] for the rationale.
- The functions in the `compileall` module now return booleans instead of `1` or `0` to represent success or failure, respectively. Thanks to booleans being a

subclass of integers, this should only be an issue if you were doing identity checks for `1` or `0`. See bpo-25768 [https://bugs.python.org/issue?@action=redirect&bpo=25768].

- Reading the **`port`** attribute of **`urllib.parse.urlsplit()`** and **`urlparse()`** results now raises **`ValueError`** for out-of-range values, rather than returning **`None`**. See bpo-20059 [https://bugs.python.org/issue?@action=redirect&bpo=20059].
- The **`imp`** module now raises a **`DeprecationWarning`** instead of **`PendingDeprecationWarning`**.
- The following modules have had missing APIs added to their **`__all__`** attributes to match the documented APIs: **`calendar`**, **`cgi`**, **`csv`**, **`ElementTree`**, **`enum`**, **`fileinput`**, **`ftplib`**, **`logging`**, **`mailbox`**, **`mimetypes`**, **`optparse`**, **`plistlib`**, **`smtpd`**, **`subprocess`**, **`tarfile`**, **`threading`** and **`wave`**. This means they will export new symbols when `import *` is used. (Contributed by Joel Taddei and Jacek Kołodziej in bpo-23883 [https://bugs.python.org/issue?@action=redirect&bpo=23883].)
- When performing a relative import, if `__package__` does not compare equal to `__spec__.parent` then **`ImportWarning`** is raised. (Contributed by Brett Cannon in bpo-25791 [https://bugs.python.org/issue?@action=redirect&bpo=25791].)
- When a relative import is performed and no parent package is known, then **`ImportError`** will be raised. Previously, **`SystemError`** could be raised. (Contributed by Brett Cannon in bpo-18018 [https://bugs.python.org/issue?@action=redirect&bpo=18018].)
- Servers based on the **`socketserver`** module, including those defined in **`http.server`**, **`xmlrpc.server`** and **`wsgiref.simple_server`**, now only catch exceptions derived from **`Exception`**. Therefore if a request handler raises an exception like **`SystemExit`** or **`KeyboardInterrupt`**,

`handle_error()` is no longer called, and the exception will stop a single-threaded server. (Contributed by Martin Panter in bpo-23430 [https://bugs.python.org/issue?@action=redirect&bpo=23430].)

- **`spwd.getspnam()`** now raises a **`PermissionError`** instead of **`KeyError`** if the user doesn't have privileges.

- The **`socket.socket.close()`** method now raises an exception if an error (e.g. `EBADF`) was reported by the underlying system call. (Contributed by Martin Panter in bpo-26685 [https://bugs.python.org/issue?@action=redirect&bpo=26685].)

- The *decode_data* argument for the **`smtpd.SMTPChannel`** and **`smtpd.SMTPServer`** constructors is now `False` by default. This means that the argument passed to **`process_message()`** is now a bytes object by default, and **`process_message()`** will be passed keyword arguments. Code that has already been updated in accordance with the deprecation warning generated by 3.5 will not be affected.

- All optional arguments of the **`dump()`**, **`dumps()`**, **`load()`** and **`loads()`** functions and **`JSONEncoder`** and **`JSONDecoder`** class constructors in the **`json`** module are now keyword-only. (Contributed by Serhiy Storchaka in bpo-18726 [https://bugs.python.org/issue?@action=redirect&bpo=18726].)

- Subclasses of **`type`** which don't override `type.__new__` may no longer use the one-argument form to get the type of an object.

- As part of **PEP 487** [https://peps.python.org/pep-0487/], the handling of keyword arguments passed to **`type`** (other than the metaclass hint, `metaclass`) is now consistently delegated to **`object.__init_subclass__()`**. This means that `type.__new__` and `type.__init__` both now accept arbitrary keyword arguments, but `object.__init_subclass__()` (which is called from `type.__new__`) will reject them by default. Custom metaclasses accepting additional keyword arguments will need to adjust their calls to `type.__new__` (whether direct or via `super`) accordingly.

- In `distutils.command.sdist.sdist`, the `default_format` attribute has been removed and is no longer honored. Instead, the gzipped tarfile format is the default on all platforms and no platform-specific selection is made. In environments where distributions are built on Windows and zip distributions are required, configure the project with a `setup.cfg` file containing the following:

  ```
  [sdist]
  formats=zip
  ```

  This behavior has also been backported to earlier Python versions by Setuptools 26.0.0.

- In the `urllib.request` module and the `http.client.HTTPConnection.request()` method, if no Content-Length header field has been specified and the request body is a file object, it is now sent with HTTP 1.1 chunked encoding. If a file object has to be sent to a HTTP 1.0 server, the Content-Length value now has to be specified by the caller. (Contributed by Demian Brecht and Rolf Krahl with tweaks from Martin Panter in bpo-12319 [https://bugs.python.org/issue?@action=redirect&bpo=12319].)

- The `DictReader` now returns rows of type `OrderedDict`. (Contributed by Steve Holden in bpo-27842 [https://bugs.python.org/issue?@action=redirect&bpo=27842].)

- The **`crypt.METHOD_CRYPT`** will no longer be added to `crypt.methods` if unsupported by the platform. (Contributed by Victor Stinner in bpo-25287 [https://bugs.python.org/issue?@action=redirect&bpo=25287].)

- The *verbose* and *rename* arguments for **`namedtuple()`** are now keyword-only. (Contributed by Raymond Hettinger in bpo-25628 [https://bugs.python.org/issue?@action=redirect&bpo=25628].)

- On Linux, **`ctypes.util.find_library()`** now looks in `LD_LIBRARY_PATH` for shared libraries. (Contributed by Vinay Sajip in bpo-9998 [https://bugs.python.org/issue?@action=redirect&bpo=9998].)

- The **`imaplib.IMAP4`** class now handles flags containing the `']'` character in messages sent from the server to improve real-world compatibility. (Contributed by Lita Cho in bpo-21815 [https://bugs.python.org/issue?@action=redirect&bpo=21815].)

- The **`mmap.mmap.write()`** function now returns the number of bytes written like other write methods. (Contributed by Jakub Stasiak in bpo-26335 [https://bugs.python.org/issue?@action=redirect&bpo=26335].)

- The **`pkgutil.iter_modules()`** and **`pkgutil.walk_packages()`** functions now return **`ModuleInfo`** named tuples. (Contributed by Ramchandra Apte in bpo-17211 [https://bugs.python.org/issue?@action=redirect&bpo=17211].)

- **`re.sub()`** now raises an error for invalid numerical group references in replacement templates even if the pattern is not found in the string. The error message for invalid group references now includes the group index and the position of the reference. (Contributed by SilentGhost,

Serhiy Storchaka in bpo-25953 [https://bugs.python.org/issue?@action=redirect&bpo=25953].)

- `zipfile.ZipFile` will now raise `NotImplementedError` for unrecognized compression values. Previously a plain `RuntimeError` was raised. Additionally, calling `ZipFile` methods on a closed ZipFile or calling the `write()` method on a ZipFile created with mode `'r'` will raise a `ValueError`. Previously, a `RuntimeError` was raised in those scenarios.

- when custom metaclasses are combined with zero-argument `super()` or direct references from methods to the implicit `__class__` closure variable, the implicit `__classcell__` namespace entry must now be passed up to `type.__new__` for initialisation. Failing to do so will result in a `DeprecationWarning` in Python 3.6 and a `RuntimeError` in Python 3.8.

- With the introduction of `ModuleNotFoundError`, import system consumers may start expecting import system replacements to raise that more specific exception when appropriate, rather than the less-specific `ImportError`. To provide future compatibility with such consumers, implementers of alternative import systems that completely replace `__import__()` will need to update their implementations to raise the new subclass when a module can’t be found at all. Implementers of compliant plugins to the default import system shouldn’t need to make any changes, as the default import system will raise the new subclass when appropriate.

Changes in the C API

- The `PyMem_Malloc()` allocator family now uses the pymalloc allocator rather than the system **`malloc()`**.

Applications calling **`PyMem_Malloc()`** without holding the GIL can now crash. Set the **`PYTHONMALLOC`** environment variable to `debug` to validate the usage of memory allocators in your application. See bpo-26249 [https://bugs.python.org/issue?@action=redirect&bpo=26249].

- **`Py_Exit()`** (and the main interpreter) now override the exit status with 120 if flushing buffered data failed. See bpo-5319 [https://bugs.python.org/issue?@action=redirect&bpo=5319].

CPython bytecode changes

There have been several major changes to the bytecode in Python 3.6.

- The Python interpreter now uses a 16-bit wordcode instead of bytecode. (Contributed by Demur Rumed with input and reviews from Serhiy Storchaka and Victor Stinner in bpo-26647 [https://bugs.python.org/issue?@action=redirect&bpo=26647] and bpo-28050 [https://bugs.python.org/issue?@action=redirect&bpo=28050].)
- The new **`FORMAT_VALUE`** and **`BUILD_STRING`** opcodes as part of the formatted string literal implementation. (Contributed by Eric Smith in bpo-25483 [https://bugs.python.org/issue?@action=redirect&bpo=25483] and Serhiy Storchaka in bpo-27078 [https://bugs.python.org/issue?@action=redirect&bpo=27078].)
- The new **`BUILD_CONST_KEY_MAP`** opcode to optimize the creation of dictionaries with constant keys. (Contributed by Serhiy Storchaka in bpo-27140 [https://bugs.python.org/issue?@action=redirect&bpo=27140].)
- The function call opcodes have been heavily reworked for better performance and simpler implementation. The **`MAKE_FUNCTION`**, **`CALL_FUNCTION`**, **`CALL_FUNCTION_KW`** and **`BUILD_MAP_UNPACK_WITH_CALL`** opcodes have been modified, the new **`CALL_FUNCTION_EX`** and **`BUILD_TUPLE_UNPACK_WITH_CALL`** have been added, and `CALL_FUNCTION_VAR`, `CALL_FUNCTION_VAR_KW` and `MAKE_CLOSURE` opcodes have been

removed. (Contributed by Demur Rumed in bpo-27095 [https://bugs.python.org/issue?@action=redirect&bpo=27095], and Serhiy Storchaka in bpo-27213 [https://bugs.python.org/issue?@action=redirect&bpo=27213], bpo-28257 [https://bugs.python.org/issue?@action=redirect&bpo=28257].)

- The new **`SETUP_ANNOTATIONS`** and **`STORE_ANNOTATION`** opcodes have been added to support the new variable annotation syntax. (Contributed by Ivan Levkivskyi in bpo-27985 [https://bugs.python.org/issue?@action=redirect&bpo=27985].)

Notable changes in Python 3.6.2

New `make regen-all` build target

To simplify cross-compilation, and to ensure that CPython can reliably be compiled without requiring an existing version of Python to already be available, the autotools-based build system no longer attempts to implicitly recompile generated files based on file modification times.

Instead, a new `make regen-all` command has been added to force regeneration of these files when desired (e.g. after an initial version of Python has already been built based on the pregenerated versions).

More selective regeneration targets are also defined - see Makefile.pre.in [https://github.com/python/cpython/tree/3.14/Makefile.pre.in] for details.

(Contributed by Victor Stinner in bpo-23404 [https://bugs.python.org/issue?@action=redirect&bpo=23404].)

Added in version 3.6.2.

Removal of `make touch` build target

The `make touch` build target previously used to request implicit regeneration of generated files by updating their modification times has been removed.

It has been replaced by the new `make regen-all` target.

(Contributed by Victor Stinner in bpo-23404 [https://bugs.python.org/issue?@action=redirect&bpo=23404].)

Changed in version 3.6.2.

Notable changes in Python 3.6.4

The `PyExc_RecursionErrorInst` singleton that was part of the public API has been removed as its members being never cleared may cause a segfault during finalization of the interpreter. (Contributed by Xavier de Gaye in bpo-22898 [https://bugs.python.org/issue?@action=redirect&bpo=22898] and bpo-30697 [https://bugs.python.org/issue?@action=redirect&bpo=30697].)

Notable changes in Python 3.6.5

The **`locale.localeconv()`** function now sets temporarily the `LC_CTYPE` locale to the `LC_NUMERIC` locale in some cases. (Contributed by Victor Stinner in bpo-31900 [https://bugs.python.org/issue?@action=redirect&bpo=31900].)

Notable changes in Python 3.6.7

`xml.dom.minidom` and **`xml.sax`** modules no longer process external entities by default. See also gh-61441 [https://github.com/python/cpython/issues/61441].

In 3.6.7 the **`tokenize`** module now implicitly emits a `NEWLINE` token when provided with input that does not have a trailing new line. This behavior now matches what the C tokenizer does internally. (Contributed by Ammar Askar in bpo-33899 [https://bugs.python.org/issue?@action=redirect&bpo=33899].)

Notable changes in Python 3.6.10

Due to significant security concerns, the *reuse_address* parameter of **`asyncio.loop.create_datagram_endpoint()`** is no longer supported. This is because of the behavior of the socket option `SO_REUSEADDR` in UDP. For more details, see the documentation for `loop.create_datagram_endpoint()`. (Contributed by Kyle Stanley, Antoine Pitrou, and Yury Selivanov in bpo-37228 [https://bugs.python.org/issue?@action=redirect&bpo=37228].)

Notable changes in Python 3.6.13

Earlier Python versions allowed using both `;` and `&` as query parameter separators in **`urllib.parse.parse_qs()`** and **`urllib.parse.parse_qsl()`**. Due to security concerns, and to conform with newer W3C recommendations, this has been changed to allow only a single separator key, with `&` as the default. This change also affects **`cgi.parse()`** and **`cgi.parse_multipart()`** as they use the affected functions internally. For more details, please see their respective documentation. (Contributed by Adam Goldschmidt, Senthil Kumaran and Ken Jin in bpo-42967 [https://bugs.python.org/issue?@action=redirect&bpo=42967].)

Notable changes in Python 3.6.14

A security fix alters the **`ftplib.FTP`** behavior to not trust the IPv4 address sent from the remote server when setting up a passive data channel. We reuse the ftp server IP address instead. For unusual code requiring the old behavior, set a `trust_server_pasv_ipv4_address` attribute on your FTP instance to `True`. (See gh-87451 [https://github.com/python/cpython/issues/87451])

The presence of newline or tab characters in parts of a URL allows for some forms of attacks. Following the WHATWG specification that updates RFC 3986, ASCII newline `\n`, `\r` and tab `\t` characters are stripped from the URL by the parser **`urllib.parse()`** preventing such attacks. The removal characters are controlled by a new module level variable `urllib.parse._UNSAFE_URL_BYTES_TO_REMOVE`. (See gh-88048 [https://github.com/python/cpython/issues/88048])

What’s New In Python 3.5

Editors:: Elvis Pranskevichus <elvis@magic.io>,
Yury Selivanov <yury@magic.io>

This article explains the new features in Python 3.5, compared to 3.4. Python 3.5 was released on September 13, 2015. See the changelog [https://docs.python.org/3.5/whatsnew/changelog.html] for a full list of changes.

See also

PEP 478 [https://peps.python.org/pep-0478/] - Python 3.5 Release Schedule

Summary – Release highlights

New syntax features:

- PEP 492, coroutines with async and await syntax.
- PEP 465, a new matrix multiplication operator: `a @ b`.
- PEP 448, additional unpacking generalizations.

New library modules:

- **`typing`**: PEP 484 – Type Hints.
- **`zipapp`**: PEP 441 Improving Python ZIP Application Support.

New built-in features:

- `bytes % args`, `bytearray % args`: PEP 461 – Adding `%` formatting to bytes and bytearray.
- New **`bytes.hex()`**, **`bytearray.hex()`** and **`memoryview.hex()`** methods. (Contributed by Arnon Yaari in bpo-9951 [https://bugs.python.org/issue?@action=redirect&bpo=9951].)
- **`memoryview`** now supports tuple indexing (including multi-dimensional). (Contributed by Antoine Pitrou in bpo-23632 [https://bugs.python.org/issue?@action=redirect&bpo=23632].)
- Generators have a new `gi_yieldfrom` attribute, which returns the object being iterated by `yield from` expressions. (Contributed by Benno Leslie and Yury Selivanov in bpo-24450 [https://bugs.python.org/issue?@action=redirect&bpo=24450].)
- A new **`RecursionError`** exception is now raised when maximum recursion depth is reached. (Contributed by Georg Brandl in bpo-19235 [https://bugs.python.org/issue?@action=redirect&bpo=19235].)

CPython implementation improvements:

- When the `LC_TYPE` locale is the POSIX locale (`C` locale), **`sys.stdin`** and **`sys.stdout`** now use the `surrogateescape` error handler, instead of the `strict` error handler. (Contributed by Victor Stinner in bpo-19977 [https://bugs.python.org/issue?@action=redirect&bpo=19977].)
- `.pyo` files are no longer used and have been replaced by a more flexible scheme that includes the optimization level explicitly in `.pyc` name. (See PEP 488 overview.)
- Builtin and extension modules are now initialized in a multi-phase process, which is similar to how Python modules are loaded. (See PEP 489 overview.)

Significant improvements in the standard library:

- **`collections.OrderedDict`** is now implemented in C, which makes it 4 to 100 times faster.

- The `ssl` module gained support for Memory BIO, which decouples SSL protocol handling from network IO.
- The new `os.scandir()` function provides a better and significantly faster way of directory traversal.
- `functools.lru_cache()` has been mostly reimplemented in C, yielding much better performance.
- The new `subprocess.run()` function provides a streamlined way to run subprocesses.
- The `traceback` module has been significantly enhanced for improved performance and developer convenience.

Security improvements:

- SSLv3 is now disabled throughout the standard library. It can still be enabled by instantiating a `ssl.SSLContext` manually. (See bpo-22638 [https://bugs.python.org/issue?@action=redirect&bpo=22638] for more details; this change was backported to CPython 3.4 and 2.7.)
- HTTP cookie parsing is now stricter, in order to protect against potential injection attacks. (Contributed by Antoine Pitrou in bpo-22796 [https://bugs.python.org/issue?@action=redirect&bpo=22796].)

Windows improvements:

- A new installer for Windows has replaced the old MSI. See Using Python on Windows for more information.
- Windows builds now use Microsoft Visual C++ 14.0, and extension modules should use the same.

Please read on for a comprehensive list of user-facing changes, including many other smaller improvements, CPython optimizations, deprecations, and potential porting issues.

New Features

PEP 492 - Coroutines with async and await syntax

PEP 492 [https://peps.python.org/pep-0492/] greatly improves support for asynchronous programming in Python by adding awaitable objects, coroutine functions, asynchronous iteration, and asynchronous context managers.

Coroutine functions are declared using the new **`async def`** syntax:

```
>>> async def coro():
...     return 'spam'
```

Inside a coroutine function, the new **`await`** expression can be used to suspend coroutine execution until the result is available. Any object can be *awaited*, as long as it implements the awaitable protocol by defining the **`__await__()`** method.

PEP 492 also adds **`async for`** statement for convenient iteration over asynchronous iterables.

An example of a rudimentary HTTP client written using the new syntax:

```
import asyncio

async def http_get(domain):
    reader, writer = await asyncio.open_connection(domain, 80)

    writer.write(b'\r\n'.join([
        b'GET / HTTP/1.1',
        b'Host: %b' % domain.encode('latin-1'),
        b'Connection: close',
        b'', b''
    ]))

    async for line in reader:
        print('>>>', line)

    writer.close()

loop = asyncio.get_event_loop()
try:
    loop.run_until_complete(http_get('example.com'))
finally:
    loop.close()
```

Similarly to asynchronous iteration, there is a new syntax for asynchronous context managers. The following script:

```
import asyncio

async def coro(name, lock):
    print('coro {}: waiting for lock'.format(name))
    async with lock:
        print('coro {}: holding the lock'.format(name))
        await asyncio.sleep(1)
        print('coro {}: releasing the lock'.format(name))

loop = asyncio.get_event_loop()
lock = asyncio.Lock()
coros = asyncio.gather(coro(1, lock), coro(2, lock))
try:
    loop.run_until_complete(coros)
finally:
    loop.close()
```

will output:

```
coro 2: waiting for lock
coro 2: holding the lock
coro 1: waiting for lock
coro 2: releasing the lock
coro 1: holding the lock
coro 1: releasing the lock
```

Note that both `async for` and `async with` can only be used inside a coroutine function declared with `async def`.

Coroutine functions are intended to be run inside a compatible event loop, such as the asyncio loop.

Note

Changed in version 3.5.2: Starting with CPython 3.5.2, `__aiter__` can directly return asynchronous iterators. Returning an awaitable object will result in a **`PendingDeprecationWarning`**.

See more details in the Asynchronous Iterators documentation section.

See also

PEP 492 [https://peps.python.org/pep-0492/] – Coroutines with async and await syntax

PEP written and implemented by Yury Selivanov.

PEP 465 - A dedicated infix operator for matrix multiplication

PEP 465 [https://peps.python.org/pep-0465/] adds the `@` infix operator for matrix multiplication. Currently, no builtin Python types implement the new operator, however, it can be implemented by defining **`__matmul__()`**, **`__rmatmul__()`**, and **`__imatmul__()`** for regular, reflected, and in-place matrix multiplication. The semantics of these methods is similar to that of methods defining other infix arithmetic operators.

Matrix multiplication is a notably common operation in many fields of mathematics, science, engineering, and the addition of `@` allows writing cleaner code:

```
S = (H @ beta - r).T @ inv(H @ V @ H.T) @ (H @ beta - r)
```

instead of:

```
S = dot((dot(H, beta) - r).T,
        dot(inv(dot(dot(H, V), H.T)), dot(H, beta) - r))
```

NumPy 1.10 has support for the new operator:

```
>>> import numpy

>>> x = numpy.ones(3)
>>> x
array([ 1., 1., 1.])

>>> m = numpy.eye(3)
>>> m
array([[ 1., 0., 0.],
       [ 0., 1., 0.],
       [ 0., 0., 1.]])

>>> x @ m
array([ 1., 1., 1.])
```

See also

PEP 465 [https://peps.python.org/pep-0465/] – A dedicated infix operator for matrix multiplication

PEP written by Nathaniel J. Smith; implemented by Benjamin Peterson.

PEP 448 - Additional Unpacking Generalizations

PEP 448 [https://peps.python.org/pep-0448/] extends the allowed uses of the `*` iterable unpacking operator and `**` dictionary unpacking operator. It is now possible to use an arbitrary number of unpackings in function calls:

```
>>> print(*[1], *[2], 3, *[4, 5])
1 2 3 4 5

>>> def fn(a, b, c, d):
...     print(a, b, c, d)
...

>>> fn(**{'a': 1, 'c': 3}, **{'b': 2, 'd': 4})
1 2 3 4
```

Similarly, tuple, list, set, and dictionary displays allow multiple unpackings (see Expression lists and Dictionary displays):

```
>>> *range(4), 4
(0, 1, 2, 3, 4)

>>> [*range(4), 4]
[0, 1, 2, 3, 4]

>>> {*range(4), 4, *(5, 6, 7)}
{0, 1, 2, 3, 4, 5, 6, 7}

>>> {'x': 1, **{'y': 2}}
{'x': 1, 'y': 2}
```

See also

PEP 448 [https://peps.python.org/pep-0448/] – Additional Unpacking Generalizations

PEP written by Joshua Landau; implemented by Neil Girdhar, Thomas Wouters, and Joshua Landau.

PEP 461 - percent formatting support for bytes and bytearray

PEP 461 [https://peps.python.org/pep-0461/] adds support for the `%` interpolation operator to **`bytes`** and **`bytearray`**.

While interpolation is usually thought of as a string operation, there are cases where interpolation on `bytes` or `bytearrays` makes sense, and the work needed to make up for this missing functionality detracts from the overall readability of the code. This issue is particularly important when dealing with wire format protocols, which are often a mixture of binary and ASCII compatible text.

Examples:

```
>>> b'Hello %b!' % b'World'
b'Hello World!'

>>> b'x=%i y=%f' % (1, 2.5)
b'x=1 y=2.500000'
```

Unicode is not allowed for `%b`, but it is accepted by `%a` (equivalent of `repr(obj).encode('ascii', 'backslashreplace')`):

```
>>> b'Hello %b!' % 'World'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: %b requires bytes, or an object that implements
__bytes__, not 'str'

>>> b'price: %a' % '10€'
b"price: '10\\u20ac'"
```

Note that `%s` and `%r` conversion types, although supported, should only be used in codebases that need compatibility with Python 2.

See also

PEP 461 [https://peps.python.org/pep-0461/] – Adding % formatting to bytes and bytearray

> PEP written by Ethan Furman; implemented by Neil Schemenauer and Ethan Furman.

PEP 484 - Type Hints

Function annotation syntax has been a Python feature since version 3.0 (**PEP 3107** [https://peps.python.org/pep-3107/]), however the semantics of annotations has been left undefined.

Experience has shown that the majority of function annotation uses were to provide type hints to function parameters and return values. It became evident that it would be beneficial for Python users, if the standard library included the base definitions and tools for type annotations.

PEP 484 [https://peps.python.org/pep-0484/] introduces a provisional module to provide these standard definitions and tools, along with some conventions for situations where annotations are not available.

For example, here is a simple function whose argument and return type are declared in the annotations:

```
def greeting(name: str) -> str:
    return 'Hello ' + name
```

While these annotations are available at runtime through the usual `__annotations__` attribute, *no automatic type checking happens at runtime*. Instead, it is assumed that a separate off-line type checker (e.g. mypy [https://mypy-lang.org]) will be used for on-demand source code analysis.

The type system supports unions, generic types, and a special type named `Any` which is consistent with (i.e. assignable to and from) all types.

See also

- `typing` module documentation
- **PEP 484 [https://peps.python.org/pep-0484/] – Type Hints**
  PEP written by Guido van Rossum, Jukka Lehtosalo, and Łukasz Langa; implemented by Guido van Rossum.
- **PEP 483 [https://peps.python.org/pep-0483/] – The Theory of Type Hints**
  PEP written by Guido van Rossum

PEP 471 - os.scandir() function – a better and faster directory iterator

PEP 471 [https://peps.python.org/pep-0471/] adds a new directory iteration function, `os.scandir()`, to the standard library. Additionally, `os.walk()` is now implemented using `scandir`, which makes it 3 to 5 times faster on POSIX systems and 7 to 20 times faster on Windows systems. This is largely achieved by greatly reducing the number of calls to `os.stat()` required to walk a directory tree.

Additionally, `scandir` returns an iterator, as opposed to returning a list of file names, which improves memory efficiency when iterating over very large directories.

The following example shows a simple use of `os.scandir()` to display all the files (excluding directories) in the given *path* that don't start with `'.'`. The `entry.is_file()` call will generally not make an additional system call:

```
for entry in os.scandir(path):
    if not entry.name.startswith('.') and entry.is_file():
        print(entry.name)
```

See also

PEP 471 [https://peps.python.org/pep-0471/] – os.scandir() function – a better and faster directory iterator

PEP written and implemented by Ben Hoyt with the help of Victor Stinner.

PEP 475: Retry system calls failing with EINTR

An `errno.EINTR` error code is returned whenever a system call, that is waiting for I/O, is interrupted by a signal. Previously, Python would raise `InterruptedError` in such cases. This meant that, when writing a Python application, the developer had two choices:

1. Ignore the `InterruptedError`.
2. Handle the `InterruptedError` and attempt to restart the interrupted system call at every call site.

The first option makes an application fail intermittently. The second option adds a large amount of boilerplate that makes the code nearly unreadable. Compare:

```
print("Hello World")
```

and:

```
while True:
    try:
        print("Hello World")
        break
    except InterruptedError:
        continue
```

PEP 475 [https://peps.python.org/pep-0475/] implements automatic retry of system calls on `EINTR`. This removes the burden of dealing with `EINTR` or `InterruptedError` in user code in most situations and makes Python programs, including the standard library, more robust. Note that the system call is only retried if the signal handler does not raise an exception.

Below is a list of functions which are now retried when interrupted by a signal:

- `open()` and `io.open()`;
- functions of the `faulthandler` module;
- `os` functions: `fchdir()`, `fchmod()`, `fchown()`, `fdatasync()`, `fstat()`, `fstatvfs()`, `fsync()`, `ftruncate()`, `mkfifo()`, `mknod()`, `open()`, `posix_fadvise()`, `posix_fallocate()`, `pread()`, `pwrite()`, `read()`, `readv()`, `sendfile()`, `wait3()`, `wait4()`, `wait()`, `waitid()`, `waitpid()`, `write()`, `writev()`;
- special cases: `os.close()` and `os.dup2()` now ignore `EINTR` errors; the syscall is not retried (see the PEP for the rationale);
- `select` functions: `devpoll.poll()`, `epoll.poll()`, `kqueue.control()`, `poll.poll()`, `select()`;
- methods of the `socket` class: `accept()`, `connect()` (except for non-blocking sockets), `recv()`, `recvfrom()`, `recvmsg()`, `send()`, `sendall()`, `sendmsg()`, `sendto()`;
- `signal.sigtimedwait()` and `signal.sigwaitinfo()`;
- `time.sleep()`.

See also

PEP 475 [https://peps.python.org/pep-0475/] – Retry system calls failing with EINTR

PEP and implementation written by Charles-François Natali and Victor Stinner, with the help of Antoine Pitrou (the French connection).

PEP 479: Change StopIteration handling inside generators

The interaction of generators and **`StopIteration`** in Python 3.4 and earlier was sometimes surprising, and could conceal obscure bugs. Previously, `StopIteration` raised accidentally inside a generator function was interpreted as the end of the iteration by the loop construct driving the generator.

PEP 479 [https://peps.python.org/pep-0479/] changes the behavior of generators: when a `StopIteration` exception is raised inside a generator, it is replaced with a **`RuntimeError`** before it exits the generator frame. The main goal of this change is to ease debugging in the situation where an unguarded **`next()`** call raises `StopIteration` and causes the iteration controlled by the generator to terminate silently. This is particularly pernicious in combination with the `yield from` construct.

This is a backwards incompatible change, so to enable the new behavior, a __future__ import is necessary:

```
>>> from __future__ import generator_stop

>>> def gen():
...     next(iter([]))
...     yield
...
>>> next(gen())
Traceback (most recent call last):
  File "<stdin>", line 2, in gen
StopIteration

The above exception was the direct cause of the following
exception:

Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
RuntimeError: generator raised StopIteration
```

Without a `__future__` import, a `PendingDeprecationWarning` will be raised whenever a `StopIteration` exception is raised inside a generator.

See also

PEP 479 [https://peps.python.org/pep-0479/] – Change StopIteration handling inside generators

PEP written by Chris Angelico and Guido van Rossum. Implemented by Chris Angelico, Yury Selivanov and Nick Coghlan.

PEP 485: A function for testing approximate equality

PEP 485 [https://peps.python.org/pep-0485/] adds the `math.isclose()` and `cmath.isclose()` functions which tell whether two values are approximately equal or “close” to each other. Whether or not two values are considered close is determined according to given absolute and relative tolerances. Relative tolerance is the maximum allowed difference between `isclose` arguments, relative to the larger absolute value:

```
>>> import math
>>> a = 5.0
>>> b = 4.99998
>>> math.isclose(a, b, rel_tol=1e-5)
True
>>> math.isclose(a, b, rel_tol=1e-6)
False
```

It is also possible to compare two values using absolute tolerance, which must be a non-negative value:

```
>>> import math
>>> a = 5.0
>>> b = 4.99998
>>> math.isclose(a, b, abs_tol=0.00003)
True
>>> math.isclose(a, b, abs_tol=0.00001)
False
```

See also

PEP 485 [https://peps.python.org/pep-0485/] – A function for testing approximate equality
PEP written by Christopher Barker; implemented by Chris Barker and Tal Einat.

PEP 486: Make the Python Launcher aware of virtual environments

PEP 486 [https://peps.python.org/pep-0486/] makes the Windows launcher (see PEP 397 [https://peps.python.org/pep-0397/]) aware of an active virtual environment. When the default interpreter would be used and the `VIRTUAL_ENV` environment variable is set, the interpreter in the virtual environment will be used.

See also

PEP 486 [https://peps.python.org/pep-0486/] – Make the Python Launcher aware of virtual environments
PEP written and implemented by Paul Moore.

PEP 488: Elimination of PYO files

PEP 488 [https://peps.python.org/pep-0488/] does away with the concept of `.pyo` files. This means that `.pyc` files represent both unoptimized and optimized bytecode. To prevent the need to constantly regenerate bytecode files, `.pyc` files now have an optional `opt-` tag in their name when the bytecode is optimized. This has the side-effect of no more bytecode file name clashes when running under either `-O` or `-OO`. Consequently, bytecode files generated from `-O`, and `-OO` may now exist simultaneously. `importlib.util.cache_from_source()` has an updated API to help with this change.

See also

PEP 488 [https://peps.python.org/pep-0488/] – Elimination of PYO files

PEP written and implemented by Brett Cannon.

PEP 489: Multi-phase extension module initialization

PEP 489 [https://peps.python.org/pep-0489/] updates extension module initialization to take advantage of the two step module loading mechanism introduced by PEP 451 [https://peps.python.org/pep-0451/] in Python 3.4.

This change brings the import semantics of extension modules that opt-in to using the new mechanism much closer to those of Python source and bytecode modules, including the ability to use any valid identifier as a module name, rather than being restricted to ASCII.

See also

PEP 489 [https://peps.python.org/pep-0489/] – Multi-phase extension module initialization

PEP written by Petr Viktorin, Stefan Behnel, and Nick Coghlan; implemented by Petr Viktorin.

Other Language Changes

Some smaller changes made to the core Python language are:

- Added the `"namereplace"` error handlers. The `"backslashreplace"` error handlers now work with decoding and translating. (Contributed by Serhiy Storchaka in bpo-19676 [https://bugs.python.org/issue?@action=redirect&bpo=19676] and bpo-22286 [https://bugs.python.org/issue?@action=redirect&bpo=22286].)
- The **`-b`** option now affects comparisons of **`bytes`** with **`int`**. (Contributed by Serhiy Storchaka in bpo-23681 [https://bugs.python.org/issue?@action=redirect&bpo=23681].)
- New Kazakh `kz1048` and Tajik `koi8_t` codecs. (Contributed by Serhiy Storchaka in bpo-22682 [https://bugs.python.org/issue?@action=redirect&bpo=22682] and bpo-22681 [https://bugs.python.org/issue?@action=redirect&bpo=22681].)
- Property docstrings are now writable. This is especially useful for **`collections.namedtuple()`** docstrings. (Contributed by Berker Peksag in bpo-24064 [https://bugs.python.org/issue?@action=redirect&bpo=24064].)
- Circular imports involving relative imports are now supported. (Contributed by Brett Cannon and Antoine Pitrou in bpo-17636 [https://bugs.python.org/issue?@action=redirect&bpo=17636].)

New Modules

typing

The new **`typing`** provisional module provides standard definitions and tools for function type annotations. See Type Hints for more information.

zipapp

The new **`zipapp`** module (specified in **PEP 441** [https://peps.python.org/pep-0441/]) provides an API and command line tool for creating executable Python Zip Applications, which were introduced in Python 2.6 in bpo-1739468 [https://bugs.python.org/issue?@action=redirect&bpo=1739468], but which were not well publicized, either at the time or since.

With the new module, bundling your application is as simple as putting all the files, including a `__main__.py` file, into a directory `myapp` and running:

```
$ python -m zipapp myapp
$ python myapp.pyz
```

The module implementation has been contributed by Paul Moore in bpo-23491 [https://bugs.python.org/issue?@action=redirect&bpo=23491].

See also

PEP 441 [https://peps.python.org/pep-0441/] – Improving Python ZIP Application Support

Improved Modules

argparse

The **`ArgumentParser`** class now allows disabling abbreviated usage of long options by setting allow_abbrev to `False`.

(Contributed by Jonathan Paugh, Steven Bethard, paul j3 and Daniel Eriksson in bpo-14910 [https://bugs.python.org/issue?@action=redirect&bpo=14910].)

asyncio

Since the `asyncio` module is provisional, all changes introduced in Python 3.5 have also been backported to Python 3.4.x.

Notable changes in the `asyncio` module since Python 3.4.0:

- New debugging APIs: `loop.set_debug()` and `loop.get_debug()` methods. (Contributed by Victor Stinner.)
- The proactor event loop now supports SSL. (Contributed by Antoine Pitrou and Victor Stinner in bpo-22560 [https://bugs.python.org/issue?@action=redirect&bpo=22560].)
- A new `loop.is_closed()` method to check if the event loop is closed. (Contributed by Victor Stinner in bpo-21326 [https://bugs.python.org/issue?@action=redirect&bpo=21326].)
- A new `loop.create_task()` to conveniently create and schedule a new `Task` for a coroutine. The `create_task` method is also used by all asyncio functions that wrap coroutines into tasks, such as `asyncio.wait()`, `asyncio.gather()`, etc. (Contributed by Victor Stinner.)
- A new `transport.get_write_buffer_limits()` method to inquire for *high-* and *low-* water limits of the flow control. (Contributed by Victor Stinner.)
- The **`async()`** function is deprecated in favor of `ensure_future()`. (Contributed by Yury Selivanov.)
- New `loop.set_task_factory()` and `loop.get_task_factory()` methods to customize the task factory that `loop.create_task()` method uses. (Contributed by Yury Selivanov.)

- New `Queue.join()` and `Queue.task_done()` queue methods. (Contributed by Victor Stinner.)
- The `JoinableQueue` class was removed, in favor of the `asyncio.Queue` class. (Contributed by Victor Stinner.)

Updates in 3.5.1:

- The `ensure_future()` function and all functions that use it, such as `loop.run_until_complete()`, now accept all kinds of awaitable objects. (Contributed by Yury Selivanov.)
- New `run_coroutine_threadsafe()` function to submit coroutines to event loops from other threads. (Contributed by Vincent Michel.)
- New `Transport.is_closing()` method to check if the transport is closing or closed. (Contributed by Yury Selivanov.)
- The `loop.create_server()` method can now accept a list of hosts. (Contributed by Yann Sionneau.)

Updates in 3.5.2:

- New `loop.create_future()` method to create Future objects. This allows alternative event loop implementations, such as uvloop [https://github.com/MagicStack/uvloop], to provide a faster `asyncio.Future` implementation. (Contributed by Yury Selivanov.)
- New `loop.get_exception_handler()` method to get the current exception handler. (Contributed by Yury Selivanov.)
- New `StreamReader.readuntil()` method to read data from the stream until a separator bytes sequence appears. (Contributed by Mark Korenberg.)
- The `loop.create_connection()` and `loop.create_server()` methods are optimized to avoid calling the system

`getaddrinfo` function if the address is already resolved. (Contributed by A. Jesse Jiryu Davis.)

- The `loop.sock_connect(sock, address)` no longer requires the *address* to be resolved prior to the call. (Contributed by A. Jesse Jiryu Davis.)

bz2

The `BZ2Decompressor.decompress` method now accepts an optional *max_length* argument to limit the maximum size of decompressed data. (Contributed by Nikolaus Rath in bpo-15955 [https://bugs.python.org/issue?@action=redirect&bpo=15955].)

cgi

The **`FieldStorage`** class now supports the context manager protocol. (Contributed by Berker Peksag in bpo-20289 [https://bugs.python.org/issue?@action=redirect&bpo=20289].)

cmath

A new function `isclose()` provides a way to test for approximate equality. (Contributed by Chris Barker and Tal Einat in bpo-24270 [https://bugs.python.org/issue?@action=redirect&bpo=24270].)

code

The `InteractiveInterpreter.showtraceback()` method now prints the full chained traceback, just like the interactive interpreter. (Contributed by Claudiu Popa in bpo-17442 [https://bugs.python.org/issue?@action=redirect&bpo=17442].)

collections

The **`OrderedDict`** class is now implemented in C, which makes it 4 to 100 times faster. (Contributed by Eric Snow in bpo-16991 [https://bugs.python.org/issue?@action=redirect&bpo=16991].)

`OrderedDict.items()`, **`OrderedDict.keys()`**, and **`OrderedDict.values()`** views now support **`reversed()`** iteration. (Contributed by Serhiy Storchaka in bpo-19505 [https://bugs.python.org/issue?@action=redirect&bpo=19505].)

The **`deque`** class now defines **`index()`**, **`insert()`**, and **`copy()`**, and supports the `+` and `*` operators. This allows deques to be recognized as a **`MutableSequence`** and improves their substitutability for lists. (Contributed by Raymond Hettinger in bpo-23704 [https://bugs.python.org/issue?@action=redirect&bpo=23704].)

Docstrings produced by **`namedtuple()`** can now be updated:

```
Point = namedtuple('Point', ['x', 'y'])
Point.__doc__ += ': Cartesian coordinate'
Point.x.__doc__ = 'abscissa'
Point.y.__doc__ = 'ordinate'
```

(Contributed by Berker Peksag in bpo-24064 [https://bugs.python.org/issue?@action=redirect&bpo=24064].)

The **`UserString`** class now implements the **`__getnewargs__()`**, **`__rmod__()`**, **`casefold()`**, **`format_map()`**, **`isprintable()`**, and **`maketrans()`** methods to match the corresponding methods of **`str`**. (Contributed by Joe Jevnik in bpo-22189 [https://bugs.python.org/issue?@action=redirect&bpo=22189].)

collections.abc

The **`Sequence.index()`** method now accepts *start* and *stop* arguments to match the corresponding methods of **`tuple`**, `list`, etc. (Contributed by Devin Jeanpierre in bpo-23086 [https://bugs.python.org/issue?@action=redirect&bpo=23086].)

A new `Generator` abstract base class. (Contributed by Stefan Behnel in bpo-24018 [https://bugs.python.org/issue?@action=redirect&bpo=24018].)

New `Awaitable`, `Coroutine`, `AsyncIterator`, and `AsyncIterable` abstract base classes. (Contributed by Yury Selivanov in bpo-24184 [https://bugs.python.org/issue?@action=redirect&bpo=24184].)

For earlier Python versions, a backport of the new ABCs is available in an external PyPI package [https://pypi.org/project/backports_abc/].

compileall

A new `compileall` option, `-j N`, allows running *N* workers simultaneously to perform parallel bytecode compilation. The `compile_dir()` function has a corresponding `workers` parameter. (Contributed by Claudiu Popa in bpo-16104 [https://bugs.python.org/issue?@action=redirect&bpo=16104].)

Another new option, `-r`, allows controlling the maximum recursion level for subdirectories. (Contributed by Claudiu Popa in bpo-19628 [https://bugs.python.org/issue?@action=redirect&bpo=19628].)

The `-q` command line option can now be specified more than once, in which case all output, including errors, will be suppressed. The corresponding `quiet` parameter in `compile_dir()`, `compile_file()`, and `compile_path()` can now accept an integer value indicating the level of output suppression. (Contributed by Thomas Kluyver in bpo-21338 [https://bugs.python.org/issue?@action=redirect&bpo=21338].)

concurrent.futures

The **`Executor.map()`** method now accepts a *chunksize* argument to allow batching of tasks to improve performance when **`ProcessPoolExecutor()`** is used. (Contributed by Dan O’Reilly in bpo-11271 [https://bugs.python.org/issue?@action=redirect&bpo=11271].)

The number of workers in the **`ThreadPoolExecutor`** constructor is optional now. The default value is 5 times the number of CPUs. (Contributed by Claudiu Popa in bpo-21527 [https://bugs.python.org/issue?@action=redirect&bpo=21527].)

configparser

`configparser` now provides a way to customize the conversion of values by specifying a dictionary of converters in the **`ConfigParser`** constructor, or by defining them as methods in `ConfigParser` subclasses. Converters defined in a parser instance are inherited by its section proxies.

Example:

```
>>> import configparser
>>> conv = {}
>>> conv['list'] = lambda v: [e.strip() for e in v.split() if
e.strip()]
>>> cfg = configparser.ConfigParser(converters=conv)
>>> cfg.read_string("""
... [s]
... list = a b c d e f g
... """)
>>> cfg.get('s', 'list')
'a b c d e f g'
>>> cfg.getlist('s', 'list')
['a', 'b', 'c', 'd', 'e', 'f', 'g']
>>> section = cfg['s']
>>> section.getlist('list')
['a', 'b', 'c', 'd', 'e', 'f', 'g']
```

(Contributed by Łukasz Langa in bpo-18159 [https://bugs.python.org/issue?@action=redirect&bpo=18159].)

contextlib

The new `redirect_stderr()` context manager (similar to `redirect_stdout()`) makes it easier for utility scripts to handle inflexible APIs that write their output to `sys.stderr` and don't provide any options to redirect it:

```
>>> import contextlib, io, logging
>>> f = io.StringIO()
>>> with contextlib.redirect_stderr(f):
...     logging.warning('warning')
...
>>> f.getvalue()
'WARNING:root:warning\n'
```

(Contributed by Berker Peksag in bpo-22389 [https://bugs.python.org/issue?@action=redirect&bpo=22389].)

csv

The `writerow()` method now supports arbitrary iterables, not just sequences. (Contributed by Serhiy Storchaka in bpo-23171 [https://bugs.python.org/issue?@action=redirect&bpo=23171].)

curses

The new `update_lines_cols()` function updates the `LINES` and `COLS` module variables. This is useful for detecting manual screen resizing. (Contributed by Arnon Yaari in bpo-4254 [https://bugs.python.org/issue?@action=redirect&bpo=4254].)

dbm

`dumb.open` always creates a new database when the flag has the value `"n"`. (Contributed by Claudiu Popa in bpo-18039 [https://bugs.python.org/issue?@action=redirect&bpo=18039].)

difflib

The charset of HTML documents generated by **`HtmlDiff.make_file()`** can now be customized by using a new *charset* keyword-only argument. The default charset of HTML document changed from `"ISO-8859-1"` to `"utf-8"`. (Contributed by Berker Peksag in bpo-2052 [https://bugs.python.org/issue?@action=redirect&bpo=2052].)

The **`diff_bytes()`** function can now compare lists of byte strings. This fixes a regression from Python 2. (Contributed by Terry J. Reedy and Greg Ward in bpo-17445 [https://bugs.python.org/issue?@action=redirect&bpo=17445].)

distutils

Both the `build` and `build_ext` commands now accept a `-j` option to enable parallel building of extension modules. (Contributed by Antoine Pitrou in bpo-5309 [https://bugs.python.org/issue?@action=redirect&bpo=5309].)

The `distutils` module now supports `xz` compression, and can be enabled by passing `xztar` as an argument to `bdist --format`. (Contributed by Serhiy Storchaka in bpo-16314 [https://bugs.python.org/issue?@action=redirect&bpo=16314].)

doctest

The **`DocTestSuite()`** function returns an empty **`unittest.TestSuite`** if *module* contains no docstrings, instead of raising **`ValueError`**. (Contributed by Glenn Jones in bpo-15916 [https://bugs.python.org/issue?@action=redirect&bpo=15916].)

email

A new policy option **`Policy.mangle_from_`** controls whether or not lines that start with `"From "` in email bodies are prefixed with a `">"` character by generators. The default is `True` for **`compat32`** and `False` for all other policies. (Contributed by Milan Oberkirch in bpo-20098 [https://bugs.python.org/issue?@action=redirect&bpo=20098].)

A new **`Message.get_content_disposition()`** method provides easy access to a canonical value for the *Content-Disposition* header. (Contributed by Abhilash Raj in bpo-21083 [https://bugs.python.org/issue?@action=redirect&bpo=21083].)

A new policy option **`EmailPolicy.utf8`** can be set to `True` to encode email headers using the UTF-8 charset instead of using encoded words. This allows `Messages` to be formatted according to **RFC 6532** [https://datatracker.ietf.org/doc/html/rfc6532.html] and used with an SMTP server that supports the **RFC 6531** [https://datatracker.ietf.org/doc/html/rfc6531.html] SMTPUTF8 extension. (Contributed by R. David Murray in bpo-24211 [https://bugs.python.org/issue?@action=redirect&bpo=24211].)

The **`mime.text.MIMEText`** constructor now accepts a **`charset.Charset`** instance. (Contributed by Claude Paroz and Berker Peksag in bpo-16324 [https://bugs.python.org/issue?@action=redirect&bpo=16324].)

enum

The **`Enum`** callable has a new parameter *start* to specify the initial number of enum values if only *names* are provided:

```
>>> Animal = enum.Enum('Animal', 'cat dog', start=10)
>>> Animal.cat
<Animal.cat: 10>
```

```
>>> Animal.dog
<Animal.dog: 11>
```

(Contributed by Ethan Furman in bpo-21706 [https://bugs.python.org/issue?@action=redirect&bpo=21706].)

faulthandler

The `enable()`, `register()`, `dump_traceback()` and `dump_traceback_later()` functions now accept file descriptors in addition to file-like objects. (Contributed by Wei Wu in bpo-23566 [https://bugs.python.org/issue?@action=redirect&bpo=23566].)

functools

Most of the `lru_cache()` machinery is now implemented in C, making it significantly faster. (Contributed by Matt Joiner, Alexey Kachayev, and Serhiy Storchaka in bpo-14373 [https://bugs.python.org/issue?@action=redirect&bpo=14373].)

glob

The `iglob()` and `glob()` functions now support recursive search in subdirectories, using the `"**"` pattern. (Contributed by Serhiy Storchaka in bpo-13968 [https://bugs.python.org/issue?@action=redirect&bpo=13968].)

gzip

The *mode* argument of the `GzipFile` constructor now accepts `"x"` to request exclusive creation. (Contributed by Tim Heaney in bpo-19222 [https://bugs.python.org/issue?@action=redirect&bpo=19222].)

heapq

Element comparison in **`merge()`** can now be customized by passing a key function in a new optional *key* keyword argument, and a new optional *reverse* keyword argument can be used to reverse element comparison:

```
>>> import heapq
>>> a = ['9', '777', '55555']
>>> b = ['88', '6666']
>>> list(heapq.merge(a, b, key=len))
['9', '88', '777', '6666', '55555']
>>> list(heapq.merge(reversed(a), reversed(b), key=len,
reverse=True))
['55555', '6666', '777', '88', '9']
```

(Contributed by Raymond Hettinger in bpo-13742 [https://bugs.python.org/issue?@action=redirect&bpo=13742].)

http

A new **`HTTPStatus`** enum that defines a set of HTTP status codes, reason phrases and long descriptions written in English. (Contributed by Demian Brecht in bpo-21793 [https://bugs.python.org/issue?@action=redirect&bpo=21793].)

http.client

`HTTPConnection.getresponse()` now raises a **`RemoteDisconnected`** exception when a remote server connection is closed unexpectedly. Additionally, if a **`ConnectionError`** (of which `RemoteDisconnected` is a subclass) is raised, the client socket is now closed automatically, and will reconnect on the next request:

```
import http.client
conn = http.client.HTTPConnection('www.python.org')
for retries in range(3):
    try:
        conn.request('GET', '/')
        resp = conn.getresponse()
    except http.client.RemoteDisconnected:
        pass
```

(Contributed by Martin Panter in bpo-3566 [https://bugs.python.org/issue?@action=redirect&bpo=3566].)

idlelib and IDLE

Since idlelib implements the IDLE shell and editor and is not intended for import by other programs, it gets improvements with every release. See `Lib/idlelib/NEWS.txt` for a cumulative list of changes since 3.4.0, as well as changes made in future 3.5.x releases. This file is also available from the IDLE *Help* ‣ *About IDLE* dialog.

imaplib

The **`IMAP4`** class now supports the context manager protocol. When used in a **`with`** statement, the IMAP4 `LOGOUT` command will be called automatically at the end of the block. (Contributed by Tarek Ziadé and Serhiy Storchaka in bpo-4972 [https://bugs.python.org/issue?@action=redirect&bpo=4972].)

The **`imaplib`** module now supports **RFC 5161** [https://datatracker.ietf.org/doc/html/rfc5161.html] (ENABLE Extension) and **RFC 6855** [https://datatracker.ietf.org/doc/html/rfc6855.html] (UTF-8 Support) via the **`IMAP4.enable()`** method. A new **`IMAP4.utf8_enabled`** attribute tracks whether or not **RFC 6855** [https://datatracker.ietf.org/doc/html/rfc6855.html] support is enabled. (Contributed by Milan Oberkirch, R. David Murray, and Maciej Szulik in bpo-21800 [https://bugs.python.org/issue?@action=redirect&bpo=21800].)

The **`imaplib`** module now automatically encodes non-ASCII string usernames and passwords using UTF-8, as recommended by the RFCs. (Contributed by Milan Oberkirch

in bpo-21800 [https://bugs.python.org/issue?@action=redirect&bpo=21800].)

imghdr

The **`what()`** function now recognizes the OpenEXR [https://www.openexr.com] format (contributed by Martin Vignali and Claudiu Popa in bpo-20295 [https://bugs.python.org/issue?@action=redirect&bpo=20295]), and the WebP [https://en.wikipedia.org/wiki/WebP] format (contributed by Fabrice Aneche and Claudiu Popa in bpo-20197 [https://bugs.python.org/issue?@action=redirect&bpo=20197].)

importlib

The **`util.LazyLoader`** class allows for lazy loading of modules in applications where startup time is important. (Contributed by Brett Cannon in bpo-17621 [https://bugs.python.org/issue?@action=redirect&bpo=17621].)

The **`abc.InspectLoader.source_to_code()`** method is now a static method. This makes it easier to initialize a module object with code compiled from a string by running `exec(code, module.__dict__)`. (Contributed by Brett Cannon in bpo-21156 [https://bugs.python.org/issue?@action=redirect&bpo=21156].)

The new **`util.module_from_spec()`** function is now the preferred way to create a new module. As opposed to creating a **`types.ModuleType`** instance directly, this new function will set the various import-controlled attributes based on the passed-in spec object. (Contributed by Brett Cannon in bpo-20383 [https://bugs.python.org/issue?@action=redirect&bpo=20383].)

inspect

Both the **`Signature`** and **`Parameter`** classes are now picklable and hashable. (Contributed by Yury Selivanov in bpo-20726 [https://bugs.python.org/issue?@action=redirect&bpo=20726] and bpo-20334 [https://bugs.python.org/issue?@action=redirect&bpo=20334].)

A new **`BoundArguments.apply_defaults()`** method provides a way to set default values for missing arguments:

```
>>> def foo(a, b='ham', *args): pass
>>> ba = inspect.signature(foo).bind('spam')
>>> ba.apply_defaults()
>>> ba.arguments
OrderedDict([('a', 'spam'), ('b', 'ham'), ('args', ())])
```

(Contributed by Yury Selivanov in bpo-24190 [https://bugs.python.org/issue?@action=redirect&bpo=24190].)

A new class method **`Signature.from_callable()`** makes subclassing of **`Signature`** easier. (Contributed by Yury Selivanov and Eric Snow in bpo-17373 [https://bugs.python.org/issue?@action=redirect&bpo=17373].)

The **`signature()`** function now accepts a *follow_wrapped* optional keyword argument, which, when set to `False`, disables automatic following of `__wrapped__` links. (Contributed by Yury Selivanov in bpo-20691 [https://bugs.python.org/issue?@action=redirect&bpo=20691].)

A set of new functions to inspect coroutine functions and coroutine objects has been added: **`iscoroutine()`**, **`iscoroutinefunction()`**, **`isawaitable()`**, **`getcoroutinelocals()`**, and **`getcoroutinestate()`**. (Contributed by Yury Selivanov in bpo-24017 [https://bugs.python.org/issue?@action=redirect&bpo=24017] and bpo-24400 [https://bugs.python.org/issue?@action=redirect&bpo=24400].)

The **`stack()`**, **`trace()`**, **`getouterframes()`**, and **`getinnerframes()`** functions now return a list of named tuples. (Contributed by

Daniel Shahaf in bpo-16808 [https://bugs.python.org/issue?@action=redirect&bpo=16808].)

io

A new `BufferedIOBase.readinto1()` method, that uses at most one call to the underlying raw stream's `RawIOBase.read()` or `RawIOBase.readinto()` methods. (Contributed by Nikolaus Rath in bpo-20578 [https://bugs.python.org/issue?@action=redirect&bpo=20578].)

ipaddress

Both the `IPv4Network` and `IPv6Network` classes now accept an `(address, netmask)` tuple argument, so as to easily construct network objects from existing addresses:

```
>>> import ipaddress
>>> ipaddress.IPv4Network(('127.0.0.0', 8))
IPv4Network('127.0.0.0/8')
>>> ipaddress.IPv4Network(('127.0.0.0', '255.0.0.0'))
IPv4Network('127.0.0.0/8')
```

(Contributed by Peter Moody and Antoine Pitrou in bpo-16531 [https://bugs.python.org/issue?@action=redirect&bpo=16531].)

A new `reverse_pointer` attribute for the `IPv4Address` and `IPv6Address` classes returns the name of the reverse DNS PTR record:

```
>>> import ipaddress
>>> addr = ipaddress.IPv4Address('127.0.0.1')
>>> addr.reverse_pointer
'1.0.0.127.in-addr.arpa'
>>> addr6 = ipaddress.IPv6Address('::1')
>>> addr6.reverse_pointer
'1.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.
0.ip6.arpa'
```

(Contributed by Leon Weber in bpo-20480 [https://bugs.python.org/issue?@action=redirect&bpo=20480].)

json

The `json.tool` command line interface now preserves the order of keys in JSON objects passed in input. The new `--sort-keys` option can be used to sort the keys alphabetically. (Contributed by Berker Peksag in bpo-21650 [https://bugs.python.org/issue?@action=redirect&bpo=21650].)

JSON decoder now raises `JSONDecodeError` instead of `ValueError` to provide better context information about the error. (Contributed by Serhiy Storchaka in bpo-19361 [https://bugs.python.org/issue?@action=redirect&bpo=19361].)

linecache

A new `lazycache()` function can be used to capture information about a non-file-based module to permit getting its lines later via `getline()`. This avoids doing I/O until a line is actually needed, without having to carry the module globals around indefinitely. (Contributed by Robert Collins in bpo-17911 [https://bugs.python.org/issue?@action=redirect&bpo=17911].)

locale

A new `delocalize()` function can be used to convert a string into a normalized number string, taking the `LC_NUMERIC` settings into account:

```
>>> import locale
>>> locale.setlocale(locale.LC_NUMERIC, 'de_DE.UTF-8')
'de_DE.UTF-8'
>>> locale.delocalize('1.234,56')
'1234.56'
>>> locale.setlocale(locale.LC_NUMERIC, 'en_US.UTF-8')
```

```
'en_US.UTF-8'
>>> locale.delocalize('1,234.56')
'1234.56'
```

(Contributed by Cédric Krier in bpo-13918 [https://bugs.python.org/issue?@action=redirect&bpo=13918].)

logging

All logging methods (**`Logger`** **`log()`**, **`exception()`**, **`critical()`**, **`debug()`**, etc.), now accept exception instances as an *exc_info* argument, in addition to boolean values and exception tuples:

```
>>> import logging
>>> try:
...     1/0
... except ZeroDivisionError as ex:
...     logging.error('exception', exc_info=ex)
ERROR:root:exception
```

(Contributed by Yury Selivanov in bpo-20537 [https://bugs.python.org/issue?@action=redirect&bpo=20537].)

The **`handlers.HTTPHandler`** class now accepts an optional **`ssl.SSLContext`** instance to configure SSL settings used in an HTTP connection. (Contributed by Alex Gaynor in bpo-22788 [https://bugs.python.org/issue?@action=redirect&bpo=22788].)

The **`handlers.QueueListener`** class now takes a *respect_handler_level* keyword argument which, if set to `True`, will pass messages to handlers taking handler levels into account. (Contributed by Vinay Sajip.)

lzma

The **`LZMADecompressor.decompress()`** method now accepts an optional *max_length* argument to limit the maximum size of decompressed data. (Contributed by Martin Panter in bpo-15955 [https://bugs.python.org/issue?@action=redirect&bpo=15955].)

math

Two new constants have been added to the **`math`** module: **`inf`** and **`nan`**. (Contributed by Mark Dickinson in bpo-23185 [https://bugs.python.org/issue?@action=redirect&bpo=23185].)

A new function **`isclose()`** provides a way to test for approximate equality. (Contributed by Chris Barker and Tal Einat in bpo-24270 [https://bugs.python.org/issue?@action=redirect&bpo=24270].)

A new **`gcd()`** function has been added. The **`fractions.gcd()`** function is now deprecated. (Contributed by Mark Dickinson and Serhiy Storchaka in bpo-22486 [https://bugs.python.org/issue?@action=redirect&bpo=22486].)

multiprocessing

`sharedctypes.synchronized()` objects now support the context manager protocol. (Contributed by Charles-François Natali in bpo-21565 [https://bugs.python.org/issue?@action=redirect&bpo=21565].)

operator

`attrgetter()`, **`itemgetter()`**, and **`methodcaller()`** objects now support pickling. (Contributed by Josh Rosenberg and Serhiy Storchaka in bpo-22955 [https://bugs.python.org/issue?@action=redirect&bpo=22955].)

New **`matmul()`** and **`imatmul()`** functions to perform matrix multiplication. (Contributed by Benjamin Peterson in bpo-21176 [https://bugs.python.org/issue?@action=redirect&bpo=21176].)

os

The new `scandir()` function returning an iterator of `DirEntry` objects has been added. If possible, `scandir()` extracts file attributes while scanning a directory, removing the need to perform subsequent system calls to determine file type or attributes, which may significantly improve performance. (Contributed by Ben Hoyt with the help of Victor Stinner in bpo-22524 [https://bugs.python.org/issue?@action=redirect&bpo=22524].)

On Windows, a new `stat_result.st_file_attributes` attribute is now available. It corresponds to the `dwFileAttributes` member of the `BY_HANDLE_FILE_INFORMATION` structure returned by `GetFileInformationByHandle()`. (Contributed by Ben Hoyt in bpo-21719 [https://bugs.python.org/issue?@action=redirect&bpo=21719].)

The `urandom()` function now uses the `getrandom()` syscall on Linux 3.17 or newer, and `getentropy()` on OpenBSD 5.6 and newer, removing the need to use `/dev/urandom` and avoiding failures due to potential file descriptor exhaustion. (Contributed by Victor Stinner in bpo-22181 [https://bugs.python.org/issue?@action=redirect&bpo=22181].)

New `get_blocking()` and `set_blocking()` functions allow getting and setting a file descriptor’s blocking mode (`O_NONBLOCK`.) (Contributed by Victor Stinner in bpo-22054 [https://bugs.python.org/issue?@action=redirect&bpo=22054].)

The `truncate()` and `ftruncate()` functions are now supported on Windows. (Contributed by Steve Dower in bpo-23668 [https://bugs.python.org/issue?@action=redirect&bpo=23668].)

There is a new `os.path.commonpath()` function returning the longest common sub-path of each passed pathname. Unlike the `os.path.commonprefix()` function, it always returns a valid path:

```
>>> os.path.commonprefix(['/usr/lib', '/usr/local/lib'])
'/usr/l'

>>> os.path.commonpath(['/usr/lib', '/usr/local/lib'])
'/usr'
```

(Contributed by Rafik Draoui and Serhiy Storchaka in bpo-10395 [https://bugs.python.org/issue?@action=redirect&bpo=10395].)

pathlib

The new **`Path.samefile()`** method can be used to check whether the path points to the same file as another path, which can be either another **`Path`** object, or a string:

```
>>> import pathlib
>>> p1 = pathlib.Path('/etc/hosts')
>>> p2 = pathlib.Path('/etc/../etc/hosts')
>>> p1.samefile(p2)
True
```

(Contributed by Vajrasky Kok and Antoine Pitrou in bpo-19775 [https://bugs.python.org/issue?@action=redirect&bpo=19775].)

The **`Path.mkdir()`** method now accepts a new optional *exist_ok* argument to match `mkdir -p` and **`os.makedirs()`** functionality. (Contributed by Berker Peksag in bpo-21539 [https://bugs.python.org/issue?@action=redirect&bpo=21539].)

There is a new **`Path.expanduser()`** method to expand `~` and `~user` prefixes. (Contributed by Serhiy Storchaka and Claudiu Popa in bpo-19776 [https://bugs.python.org/issue?@action=redirect&bpo=19776].)

A new **`Path.home()`** class method can be used to get a **`Path`** instance representing the user’s home directory. (Contributed by Victor Salgado and Mayank Tripathi in bpo-19777 [https://bugs.python.org/issue?@action=redirect&bpo=19777].)

New **`Path.write_text()`**, **`Path.read_text()`**, **`Path.write_bytes()`**, **`Path.read_bytes()`** methods to simplify read/write operations on files.

The following code snippet will create or rewrite existing file `~/spam42`:

```
>>> import pathlib
>>> p = pathlib.Path('~/spam42')
>>> p.expanduser().write_text('ham')
3
```

(Contributed by Christopher Welborn in bpo-20218 [https://bugs.python.org/issue?@action=redirect&bpo=20218].)

pickle

Nested objects, such as unbound methods or nested classes, can now be pickled using pickle protocols older than protocol version 4. Protocol version 4 already supports these cases. (Contributed by Serhiy Storchaka in bpo-23611 [https://bugs.python.org/issue?@action=redirect&bpo=23611].)

poplib

A new **`POP3.utf8()`** command enables **RFC 6856** [https://datatracker.ietf.org/doc/html/rfc6856.html] (Internationalized Email) support, if a POP server supports it. (Contributed by Milan OberKirch in bpo-21804 [https://bugs.python.org/issue?@action=redirect&bpo=21804].)

re

References and conditional references to groups with fixed length are now allowed in lookbehind assertions:

```
>>> import re
>>> pat = re.compile(r'(a|b).(?<=\1)c')
>>> pat.match('aac')
<_sre.SRE_Match object; span=(0, 3), match='aac'>
>>> pat.match('bbc')
<_sre.SRE_Match object; span=(0, 3), match='bbc'>
```

(Contributed by Serhiy Storchaka in bpo-9179 [https://bugs.python.org/issue?@action=redirect&bpo=9179].)

The number of capturing groups in regular expressions is no longer limited to 100. (Contributed by Serhiy Storchaka in bpo-22437 [https://bugs.python.org/issue?@action=redirect&bpo=22437].)

The **`sub()`** and **`subn()`** functions now replace unmatched groups with empty strings instead of raising an exception. (Contributed by Serhiy Storchaka in bpo-1519638 [https://bugs.python.org/issue?@action=redirect&bpo=1519638].)

The **`re.error`** exceptions have new attributes, **`msg`**, **`pattern`**, **`pos`**, **`lineno`**, and **`colno`**, that provide better context information about the error:

```
>>> re.compile("""
...     (?x)
...     .++
... """)
Traceback (most recent call last):
   ...
sre_constants.error: multiple repeat at position 16 (line 3,
column 7)
```

(Contributed by Serhiy Storchaka in bpo-22578 [https://bugs.python.org/issue?@action=redirect&bpo=22578].)

readline

A new **`append_history_file()`** function can be used to append the specified number of trailing elements in history to the given file. (Contributed by Bruno Cauet in bpo-22940 [https://bugs.python.org/issue?@action=redirect&bpo=22940].)

selectors

The new **`DevpollSelector`** supports efficient `/dev/poll` polling on Solaris. (Contributed by Giampaolo Rodola' in bpo-18931 [https://bugs.python.org/issue?@action=redirect&bpo=18931].)

shutil

The **`move()`** function now accepts a *copy_function* argument, allowing, for example, the **`copy()`** function to be used instead of the default **`copy2()`** if there is a need to ignore file metadata when moving. (Contributed by Claudiu Popa in bpo-19840 [https://bugs.python.org/issue?@action=redirect&bpo=19840].)

The **`make_archive()`** function now supports the *xztar* format. (Contributed by Serhiy Storchaka in bpo-5411 [https://bugs.python.org/issue?@action=redirect&bpo=5411].)

signal

On Windows, the **`set_wakeup_fd()`** function now also supports socket handles. (Contributed by Victor Stinner in bpo-22018 [https://bugs.python.org/issue?@action=redirect&bpo=22018].)

Various `SIG*` constants in the **`signal`** module have been converted into **`Enums`**. This allows meaningful names to be printed during debugging, instead of integer “magic numbers”. (Contributed by Giampaolo Rodola' in bpo-21076 [https://bugs.python.org/issue?@action=redirect&bpo=21076].)

smtpd

Both the **`SMTPServer`** and **`SMTPChannel`** classes now accept a *decode_data* keyword argument to determine if the `DATA` portion of the SMTP transaction is decoded using the `"utf-8"` codec or is instead provided to the **`SMTPServer.process_message()`** method as a byte string. The default is `True` for backward compatibility reasons, but will change to `False` in Python 3.6. If *decode_data* is set to `False`, the `process_message` method must be prepared to accept keyword arguments. (Contributed by Maciej Szulik in bpo-19662 [https://bugs.python.org/issue?@action=redirect&bpo=19662].)

The **`SMTPServer`** class now advertises the `8BITMIME` extension (**RFC 6152** [https://datatracker.ietf.org/doc/html/rfc6152.html]) if *decode_data* has been set `True`. If the client specifies `BODY=8BITMIME` on the `MAIL` command, it is passed to **`SMTPServer.process_message()`** via the *mail_options* keyword. (Contributed by Milan Oberkirch and R. David Murray in bpo-21795 [https://bugs.python.org/issue?@action=redirect&bpo=21795].)

The **`SMTPServer`** class now also supports the `SMTPUTF8` extension (**RFC 6531** [https://datatracker.ietf.org/doc/html/rfc6531.html]: Internationalized Email). If the client specified `SMTPUTF8 BODY=8BITMIME` on the `MAIL` command, they are passed to **`SMTPServer.process_message()`** via the *mail_options* keyword. It is the responsibility of the `process_message` method to correctly handle the `SMTPUTF8` data. (Contributed by Milan Oberkirch in bpo-21725 [https://bugs.python.org/issue?@action=redirect&bpo=21725].)

It is now possible to provide, directly or via name resolution, IPv6 addresses in the **`SMTPServer`** constructor, and have it successfully connect. (Contributed by Milan Oberkirch in bpo-14758 [https://bugs.python.org/issue?@action=redirect&bpo=14758].)

smtplib

A new **`SMTP.auth()`** method provides a convenient way to implement custom authentication mechanisms. (Contributed by Milan Oberkirch in bpo-15014 [https://bugs.python.org/issue?@action=redirect&bpo=15014].)

The **`SMTP.set_debuglevel()`** method now accepts an additional debuglevel (2), which enables timestamps in debug messages. (Contributed by Gavin Chappell and Maciej Szulik in bpo-16914 [https://bugs.python.org/issue?@action=redirect&bpo=16914].)

Both the **`SMTP.sendmail()`** and **`SMTP.send_message()`** methods now support **RFC 6531** [https://datatracker.ietf.org/doc/html/rfc6531.html] (SMTPUTF8). (Contributed by Milan Oberkirch and R. David Murray in bpo-22027 [https://bugs.python.org/issue?@action=redirect&bpo=22027].)

sndhdr

The **`what()`** and **`whathdr()`** functions now return a **`namedtuple()`**. (Contributed by Claudiu Popa in bpo-18615 [https://bugs.python.org/issue?@action=redirect&bpo=18615].)

socket

Functions with timeouts now use a monotonic clock, instead of a system clock. (Contributed by Victor Stinner in bpo-22043 [https://bugs.python.org/issue?@action=redirect&bpo=22043].)

A new **`socket.sendfile()`** method allows sending a file over a socket by using the high-performance **`os.sendfile()`** function on UNIX, resulting in uploads being from 2 to 3 times faster than when using plain **`socket.send()`**. (Contributed by Giampaolo Rodola’ in bpo-17552 [https://bugs.python.org/issue?@action=redirect&bpo=17552].)

The `socket.sendall()` method no longer resets the socket timeout every time bytes are received or sent. The socket timeout is now the maximum total duration to send all data. (Contributed by Victor Stinner in bpo-23853 [https://bugs.python.org/issue?@action=redirect&bpo=23853].)

The *backlog* argument of the `socket.listen()` method is now optional. By default it is set to `SOMAXCONN` or to `128`, whichever is less. (Contributed by Charles-François Natali in bpo-21455 [https://bugs.python.org/issue?@action=redirect&bpo=21455].)

ssl

Memory BIO Support

(Contributed by Geert Jansen in bpo-21965 [https://bugs.python.org/issue?@action=redirect&bpo=21965].)

The new `SSLObject` class has been added to provide SSL protocol support for cases when the network I/O capabilities of `SSLSocket` are not necessary or are suboptimal. `SSLObject` represents an SSL protocol instance, but does not implement any network I/O methods, and instead provides a memory buffer interface. The new `MemoryBIO` class can be used to pass data between Python and an SSL protocol instance.

The memory BIO SSL support is primarily intended to be used in frameworks implementing asynchronous I/O for which `SSLSocket`'s readiness model ("select/poll") is inefficient.

A new `SSLContext.wrap_bio()` method can be used to create a new `SSLObject` instance.

Application-Layer Protocol Negotiation Support

(Contributed by Benjamin Peterson in bpo-20188 [https://bugs.python.org/issue?@action=redirect&bpo=20188].)

Where OpenSSL support is present, the `ssl` module now implements the *Application-Layer Protocol Negotiation* TLS extension as described in **RFC 7301** [https://datatracker.ietf.org/doc/html/rfc7301.html].

The new `SSLContext.set_alpn_protocols()` can be used to specify which protocols a socket should advertise during the TLS handshake.

The new `SSLSocket.selected_alpn_protocol()` returns the protocol that was selected during the TLS handshake. The `HAS_ALPN` flag indicates whether ALPN support is present.

Other Changes

There is a new `SSLSocket.version()` method to query the actual protocol version in use. (Contributed by Antoine Pitrou in bpo-20421 [https://bugs.python.org/issue?@action=redirect&bpo=20421].)

The `SSLSocket` class now implements a **`SSLSocket.sendfile()`** method. (Contributed by Giampaolo Rodola' in bpo-17552 [https://bugs.python.org/issue?@action=redirect&bpo=17552].)

The **`SSLSocket.send()`** method now raises either the `ssl.SSLWantReadError` or `ssl.SSLWantWriteError` exception on a non-blocking socket if the operation would block. Previously, it would return `0`. (Contributed by Nikolaus Rath in bpo-20951 [https://bugs.python.org/issue?@action=redirect&bpo=20951].)

The `cert_time_to_seconds()` function now interprets the input time as UTC and not as local time, per **RFC 5280** [https://datatracker.ietf.org/doc/html/rfc5280.html]. Additionally, the return value is always an `int`. (Contributed by Akira Li in bpo-19940 [https://bugs.python.org/issue?@action=redirect&bpo=19940].)

New `SSLObject.shared_ciphers()` and `SSLSocket.shared_ciphers()` methods return the list of ciphers sent by the client during the handshake. (Contributed by Benjamin Peterson in bpo-23186 [https://bugs.python.org/issue?@action=redirect&bpo=23186].)

The `SSLSocket.do_handshake()`, `SSLSocket.read()`, `SSLSocket.shutdown()`, and `SSLSocket.write()` methods of the `SSLSocket` class no longer reset the socket timeout every time bytes are received or sent. The socket timeout is now the maximum total duration of the method. (Contributed by Victor Stinner in bpo-23853 [https://bugs.python.org/issue?@action=redirect&bpo=23853].)

The `match_hostname()` function now supports matching of IP addresses. (Contributed by Antoine Pitrou in bpo-23239 [https://bugs.python.org/issue?@action=redirect&bpo=23239].)

sqlite3

The `Row` class now fully supports the sequence protocol, in particular `reversed()` iteration and slice indexing. (Contributed by Claudiu Popa in bpo-10203 [https://bugs.python.org/issue?@action=redirect&bpo=10203]; by Lucas Sinclair, Jessica McKellar, and Serhiy Storchaka in bpo-13583 [https://bugs.python.org/issue?@action=redirect&bpo=13583].)

subprocess

The new `run()` function has been added. It runs the specified command and returns a `CompletedProcess` object, which describes a finished process. The new API is more consistent and is the recommended approach to invoking subprocesses in Python code that does not need to maintain compatibility with earlier Python versions. (Contributed by Thomas Kluyver in bpo-23342 [https://bugs.python.org/issue?@action=redirect&bpo=23342].)

Examples:

```
>>> subprocess.run(["ls", "-l"])  # doesn't capture output
CompletedProcess(args=['ls', '-l'], returncode=0)

>>> subprocess.run("exit 1", shell=True, check=True)
Traceback (most recent call last):
  ...
subprocess.CalledProcessError: Command 'exit 1' returned non-
zero exit status 1

>>> subprocess.run(["ls", "-l", "/dev/null"],
stdout=subprocess.PIPE)
CompletedProcess(args=['ls', '-l', '/dev/null'], returncode=0,
stdout=b'crw-rw-rw- 1 root root 1, 3 Jan 23 16:23 /dev/null\n')
```

sys

A new **`set_coroutine_wrapper()`** function allows setting a global hook that will be called whenever a coroutine object is created by an **`async def`** function. A corresponding **`get_coroutine_wrapper()`** can be used to obtain a currently set wrapper. Both functions are provisional, and are intended for debugging purposes only. (Contributed by Yury Selivanov in bpo-24017 [https://bugs.python.org/issue?@action=redirect&bpo=24017].)

A new **`is_finalizing()`** function can be used to check if the Python interpreter is shutting down. (Contributed by Antoine Pitrou in bpo-22696 [https://bugs.python.org/issue?@action=redirect&bpo=22696].)

sysconfig

The name of the user scripts directory on Windows now includes the first two components of the Python version. (Contributed by Paul Moore in bpo-23437 [https://bugs.python.org/issue?@action=redirect&bpo=23437].)

tarfile

The *mode* argument of the **`open()`** function now accepts `"x"` to request exclusive creation. (Contributed by Berker Peksag in bpo-21717 [https://bugs.python.org/issue?@action=redirect&bpo=21717].)

The **`TarFile.extractall()`** and **`TarFile.extract()`** methods now take a keyword argument *numeric_owner*. If set to `True`, the extracted files and directories will be owned by the numeric `uid` and `gid` from the tarfile. If set to `False` (the default, and the behavior in versions prior to 3.5), they will be owned by the named user and group in the tarfile. (Contributed by Michael Vogt and Eric Smith in bpo-23193 [https://bugs.python.org/issue?@action=redirect&bpo=23193].)

The **`TarFile.list()`** now accepts an optional *members* keyword argument that can be set to a subset of the list returned by **`TarFile.getmembers()`**. (Contributed by Serhiy Storchaka in bpo-21549 [https://bugs.python.org/issue?@action=redirect&bpo=21549].)

threading

Both the **`Lock.acquire()`** and **`RLock.acquire()`** methods now use a monotonic clock for timeout management. (Contributed by Victor Stinner in bpo-22043 [https://bugs.python.org/issue?@action=redirect&bpo=22043].)

time

The **`monotonic()`** function is now always available. (Contributed by Victor Stinner in bpo-22043 [https://bugs.python.org/issue?@action=redirect&bpo=22043].)

timeit

A new command line option `-u` or `--unit=U` can be used to specify the time unit for the timer output. Supported options are `usec`, `msec`, or `sec`. (Contributed by Julian Gindi in bpo-18983 [https://bugs.python.org/issue?@action=redirect&bpo=18983].)

The **`timeit()`** function has a new *globals* parameter for specifying the namespace in which the code will be running. (Contributed by Ben Roberts in bpo-2527 [https://bugs.python.org/issue?@action=redirect&bpo=2527].)

tkinter

The **`tkinter._fix`** module used for setting up the Tcl/Tk environment on Windows has been replaced by a private function in the **`_tkinter`** module which makes no permanent changes to environment variables. (Contributed by Zachary Ware in bpo-20035 [https://bugs.python.org/issue?@action=redirect&bpo=20035].)

traceback

New **`walk_stack()`** and **`walk_tb()`** functions to conveniently traverse frame and traceback objects. (Contributed by Robert Collins in bpo-17911 [https://bugs.python.org/issue?@action=redirect&bpo=17911].)

New lightweight classes: **`TracebackException`**, **`StackSummary`**, and **`FrameSummary`**. (Contributed by Robert Collins in bpo-17911 [https://bugs.python.org/issue?@action=redirect&bpo=17911].)

Both the `print_tb()` and `print_stack()` functions now support negative values for the *limit* argument. (Contributed by Dmitry Kazakov in bpo-22619 [https://bugs.python.org/issue?@action=redirect&bpo=22619].)

types

A new `coroutine()` function to transform generator and `generator-like` objects into awaitables. (Contributed by Yury Selivanov in bpo-24017 [https://bugs.python.org/issue?@action=redirect&bpo=24017].)

A new type called `CoroutineType`, which is used for coroutine objects created by `async def` functions. (Contributed by Yury Selivanov in bpo-24400 [https://bugs.python.org/issue?@action=redirect&bpo=24400].)

unicodedata

The `unicodedata` module now uses data from Unicode 8.0.0 [https://unicode.org/versions/Unicode8.0.0/].

unittest

The `TestLoader.loadTestsFromModule()` method now accepts a keyword-only argument *pattern* which is passed to `load_tests` as the third argument. Found packages are now checked for `load_tests` regardless of whether their path matches *pattern*, because it is impossible for a package name to match the default pattern. (Contributed by Robert Collins and Barry A. Warsaw in bpo-16662 [https://bugs.python.org/issue?@action=redirect&bpo=16662].)

Unittest discovery errors now are exposed in the `TestLoader.errors` attribute of the `TestLoader` instance.

(Contributed by Robert Collins in bpo-19746 [https://bugs.python.org/issue?@action=redirect&bpo=19746].)

A new command line option `--locals` to show local variables in tracebacks. (Contributed by Robert Collins in bpo-22936 [https://bugs.python.org/issue?@action=redirect&bpo=22936].)

unittest.mock

The **`Mock`** class has the following improvements:

- The class constructor has a new *unsafe* parameter, which causes mock objects to raise **`AttributeError`** on attribute names starting with `"assert"`. (Contributed by Kushal Das in bpo-21238 [https://bugs.python.org/issue?@action=redirect&bpo=21238].)
- A new **`Mock.assert_not_called()`** method to check if the mock object was called. (Contributed by Kushal Das in bpo-21262 [https://bugs.python.org/issue?@action=redirect&bpo=21262].)

The **`MagicMock`** class now supports **`__truediv__()`**, **`__divmod__()`** and **`__matmul__()`** operators. (Contributed by Johannes Baiter in bpo-20968 [https://bugs.python.org/issue?@action=redirect&bpo=20968], and Håkan Lövdahl in bpo-23581 [https://bugs.python.org/issue?@action=redirect&bpo=23581] and bpo-23568 [https://bugs.python.org/issue?@action=redirect&bpo=23568].)

It is no longer necessary to explicitly pass `create=True` to the **`patch()`** function when patching builtin names. (Contributed by Kushal Das in bpo-17660 [https://bugs.python.org/issue?@action=redirect&bpo=17660].)

urllib

A new `request.HTTPPasswordMgrWithPriorAuth` class allows HTTP Basic Authentication credentials to be managed so as to eliminate unnecessary `401` response handling, or to unconditionally send credentials on the first request in order to communicate with servers that return a `404` response instead of a `401` if the `Authorization` header is not sent. (Contributed by Matej Cepl in bpo-19494 [https://bugs.python.org/issue?@action=redirect&bpo=19494] and Akshit Khurana in bpo-7159 [https://bugs.python.org/issue?@action=redirect&bpo=7159].)

A new *quote_via* argument for the `parse.urlencode()` function provides a way to control the encoding of query parts if needed. (Contributed by Samwyse and Arnon Yaari in bpo-13866 [https://bugs.python.org/issue?@action=redirect&bpo=13866].)

The `request.urlopen()` function accepts an `ssl.SSLContext` object as a *context* argument, which will be used for the HTTPS connection. (Contributed by Alex Gaynor in bpo-22366 [https://bugs.python.org/issue?@action=redirect&bpo=22366].)

The `parse.urljoin()` was updated to use the **RFC 3986** [https://datatracker.ietf.org/doc/html/rfc3986.html] semantics for the resolution of relative URLs, rather than **RFC 1808** [https://datatracker.ietf.org/doc/html/rfc1808.html] and **RFC 2396** [https://datatracker.ietf.org/doc/html/rfc2396.html]. (Contributed by Demian Brecht and Senthil Kumaran in bpo-22118 [https://bugs.python.org/issue?@action=redirect&bpo=22118].)

wsgiref

The *headers* argument of the `headers.Headers` class constructor is now optional. (Contributed by Pablo Torres Navarrete and SilentGhost in bpo-5800 [https://bugs.python.org/issue?@action=redirect&bpo=5800].)

xmlrpc

The **`client.ServerProxy`** class now supports the context manager protocol. (Contributed by Claudiu Popa in bpo-20627 [https://bugs.python.org/issue?@action=redirect&bpo=20627].)

The **`client.ServerProxy`** constructor now accepts an optional **`ssl.SSLContext`** instance. (Contributed by Alex Gaynor in bpo-22960 [https://bugs.python.org/issue?@action=redirect&bpo=22960].)

xml.sax

SAX parsers now support a character stream of the **`xmlreader.InputSource`** object. (Contributed by Serhiy Storchaka in bpo-2175 [https://bugs.python.org/issue?@action=redirect&bpo=2175].)

`parseString()` now accepts a **`str`** instance. (Contributed by Serhiy Storchaka in bpo-10590 [https://bugs.python.org/issue?@action=redirect&bpo=10590].)

zipfile

ZIP output can now be written to unseekable streams. (Contributed by Serhiy Storchaka in bpo-23252 [https://bugs.python.org/issue?@action=redirect&bpo=23252].)

The *mode* argument of **`ZipFile.open()`** method now accepts `"x"` to request exclusive creation. (Contributed by Serhiy Storchaka in bpo-21717 [https://bugs.python.org/issue?@action=redirect&bpo=21717].)

Other module-level changes

Many functions in the **`mmap`**, **`ossaudiodev`**, **`socket`**, **`ssl`**, and **`codecs`** modules now accept writable bytes-like objects.

(Contributed by Serhiy Storchaka in bpo-23001 [https://bugs.python.org/issue?@action=redirect&bpo=23001].)

Optimizations

The **`os.walk()`** function has been sped up by 3 to 5 times on POSIX systems, and by 7 to 20 times on Windows. This was done using the new **`os.scandir()`** function, which exposes file information from the underlying `readdir` or `FindFirstFile`/`FindNextFile` system calls. (Contributed by Ben Hoyt with help from Victor Stinner in bpo-23605 [https://bugs.python.org/issue?@action=redirect&bpo=23605].)

Construction of `bytes(int)` (filled by zero bytes) is faster and uses less memory for large objects. `calloc()` is used instead of `malloc()` to allocate memory for these objects. (Contributed by Victor Stinner in bpo-21233 [https://bugs.python.org/issue?@action=redirect&bpo=21233].)

Some operations on **`ipaddress`** **`IPv4Network`** and **`IPv6Network`** have been massively sped up, such as **`subnets()`**, **`supernet()`**, **`summarize_address_range()`**, **`collapse_addresses()`**. The speed up can range from 3 to 15 times. (Contributed by Antoine Pitrou, Michel Albert, and Markus in bpo-21486 [https://bugs.python.org/issue?@action=redirect&bpo=21486], bpo-21487 [https://bugs.python.org/issue?@action=redirect&bpo=21487], bpo-20826 [https://bugs.python.org/issue?@action=redirect&bpo=20826], bpo-23266 [https://bugs.python.org/issue?@action=redirect&bpo=23266].)

Pickling of **`ipaddress`** objects was optimized to produce significantly smaller output. (Contributed by Serhiy Storchaka in bpo-23133 [https://bugs.python.org/issue?@action=redirect&bpo=23133].)

Many operations on **`io.BytesIO`** are now 50% to 100% faster. (Contributed by Serhiy Storchaka in bpo-15381 [https://bugs.python.org/issue?@action=redirect&bpo=15381] and David Wilson in bpo-22003 [https://bugs.python.org/issue?@action=redirect&bpo=22003].)

The **`marshal.dumps()`** function is now faster: 65–85% with versions 3 and 4, 20–25% with versions 0 to 2 on typical data, and up to 5 times in best cases. (Contributed by Serhiy Storchaka in bpo-20416 [https://bugs.python.org/issue?@action=redirect&bpo=20416] and bpo-23344 [https://bugs.python.org/issue?@action=redirect&bpo=23344].)

The UTF-32 encoder is now 3 to 7 times faster. (Contributed by Serhiy Storchaka in bpo-15027 [https://bugs.python.org/issue?@action=redirect&bpo=15027].)

Regular expressions are now parsed up to 10% faster. (Contributed by Serhiy Storchaka in bpo-19380 [https://bugs.python.org/issue?@action=redirect&bpo=19380].)

The **`json.dumps()`** function was optimized to run with `ensure_ascii=False` as fast as with `ensure_ascii=True`. (Contributed by Naoki Inada in bpo-23206 [https://bugs.python.org/issue?@action=redirect&bpo=23206].)

The **`PyObject_IsInstance()`** and **`PyObject_IsSubclass()`** functions have been sped up in the common case that the second argument has **`type`** as its metaclass. (Contributed Georg Brandl by in bpo-22540 [https://bugs.python.org/issue?@action=redirect&bpo=22540].)

Method caching was slightly improved, yielding up to 5% performance improvement in some benchmarks. (Contributed by Antoine Pitrou in bpo-22847 [https://bugs.python.org/issue?@action=redirect&bpo=22847].)

Objects from the **`random`** module now use 50% less memory on 64-bit builds. (Contributed by Serhiy Storchaka in bpo-

23488 [https://bugs.python.org/issue?@action=redirect&bpo=23488].)

The **`property()`** getter calls are up to 25% faster. (Contributed by Joe Jevnik in bpo-23910 [https://bugs.python.org/issue?@action=redirect&bpo=23910].)

Instantiation of **`fractions.Fraction`** is now up to 30% faster. (Contributed by Stefan Behnel in bpo-22464 [https://bugs.python.org/issue?@action=redirect&bpo=22464].)

String methods **`find()`**, **`rfind()`**, **`split()`**, **`partition()`** and the **`in`** string operator are now significantly faster for searching 1-character substrings. (Contributed by Serhiy Storchaka in bpo-23573 [https://bugs.python.org/issue?@action=redirect&bpo=23573].)

Build and C API Changes

New `calloc` functions were added:

- **`PyMem_RawCalloc()`**,
- **`PyMem_Calloc()`**,
- **`PyObject_Calloc()`**.

(Contributed by Victor Stinner in bpo-21233 [https://bugs.python.org/issue?@action=redirect&bpo=21233].)

New encoding/decoding helper functions:

- **`Py_DecodeLocale()`** (replaced `_Py_char2wchar()`),
- **`Py_EncodeLocale()`** (replaced `_Py_wchar2char()`).

(Contributed by Victor Stinner in bpo-18395 [https://bugs.python.org/issue?@action=redirect&bpo=18395].)

A new **`PyCodec_NameReplaceErrors()`** function to replace the unicode encode error with `\N{...}` escapes. (Contributed by

Serhiy Storchaka in bpo-19676 [https://bugs.python.org/issue?@action=redirect&bpo=19676].)

A new `PyErr_FormatV()` function similar to `PyErr_Format()`, but accepts a `va_list` argument. (Contributed by Antoine Pitrou in bpo-18711 [https://bugs.python.org/issue?@action=redirect&bpo=18711].)

A new `PyExc_RecursionError` exception. (Contributed by Georg Brandl in bpo-19235 [https://bugs.python.org/issue?@action=redirect&bpo=19235].)

New `PyModule_FromDefAndSpec()`, `PyModule_FromDefAndSpec2()`, and `PyModule_ExecDef()` functions introduced by **PEP 489** [https://peps.python.org/pep-0489/] – multi-phase extension module initialization. (Contributed by Petr Viktorin in bpo-24268 [https://bugs.python.org/issue?@action=redirect&bpo=24268].)

New `PyNumber_MatrixMultiply()` and `PyNumber_InPlaceMatrixMultiply()` functions to perform matrix multiplication. (Contributed by Benjamin Peterson in bpo-21176 [https://bugs.python.org/issue?@action=redirect&bpo=21176]. See also **PEP 465** [https://peps.python.org/pep-0465/] for details.)

The `PyTypeObject.tp_finalize` slot is now part of the stable ABI.

Windows builds now require Microsoft Visual C++ 14.0, which is available as part of Visual Studio 2015 [https://visualstudio.microsoft.com/en/vs/older-downloads/#visual-studio-2015-and-other-products].

Extension modules now include a platform information tag in their filename on some platforms (the tag is optional, and CPython will import extensions without it, although if the tag is present and mismatched, the extension won’t be loaded):

- On Linux, extension module filenames end with `.cpython-<major><minor>m-<architecture>-<os>.pyd`:
    - `<major>` is the major number of the Python version; for Python 3.5 this is `3`.
    - `<minor>` is the minor number of the Python version; for Python 3.5 this is `5`.
    - `<architecture>` is the hardware architecture the extension module was built to run on. It's most commonly either `i386` for 32-bit Intel platforms or `x86_64` for 64-bit Intel (and AMD) platforms.
    - `<os>` is always `linux-gnu`, except for extensions built to talk to the 32-bit ABI on 64-bit platforms, in which case it is `linux-gnu32` (and `<architecture>` will be `x86_64`).
- On Windows, extension module filenames end with `<debug>.cp<major><minor>-<platform>.pyd`:
    - `<major>` is the major number of the Python version; for Python 3.5 this is `3`.
    - `<minor>` is the minor number of the Python version; for Python 3.5 this is `5`.
    - `<platform>` is the platform the extension module was built for, either `win32` for Win32, `win_amd64` for Win64, `win_ia64` for Windows Itanium 64, and `win_arm` for Windows on ARM.
    - If built in debug mode, `<debug>` will be `_d`, otherwise it will be blank.
- On OS X platforms, extension module filenames now end with `-darwin.so`.
- On all other platforms, extension module filenames are the same as they were with Python 3.4.

Deprecated

New Keywords

`async` and `await` are not recommended to be used as variable, class, function or module names. Introduced by **PEP 492** [https://peps.python.org/pep-0492/] in Python 3.5, they will become proper keywords in Python 3.7.

Deprecated Python Behavior

Raising the **`StopIteration`** exception inside a generator will now generate a silent **`PendingDeprecationWarning`**, which will become a non-silent deprecation warning in Python 3.6 and will trigger a **`RuntimeError`** in Python 3.7. See PEP 479: Change StopIteration handling inside generators for details.

Unsupported Operating Systems

Windows XP is no longer supported by Microsoft, thus, per **PEP 11** [https://peps.python.org/pep-0011/], CPython 3.5 is no longer officially supported on this OS.

Deprecated Python modules, functions and methods

The **`formatter`** module has now graduated to full deprecation and is still slated for removal in Python 3.6.

The **`asyncio.async()`** function is deprecated in favor of **`ensure_future()`**.

The **`smtpd`** module has in the past always decoded the DATA portion of email messages using the `utf-8` codec. This can now be controlled by the new *decode_data* keyword to **`SMTPServer`**. The default value is `True`, but this default is deprecated. Specify the *decode_data* keyword with an appropriate value to avoid the deprecation warning.

Directly assigning values to the `key`, `value` and `coded_value` of `http.cookies.Morsel` objects is deprecated. Use the `set()` method instead. In addition, the undocumented *LegalChars* parameter of `set()` is deprecated, and is now ignored.

Passing a format string as keyword argument *format_string* to the `format()` method of the `string.Formatter` class has been deprecated. (Contributed by Serhiy Storchaka in bpo-23671 [https://bugs.python.org/issue?@action=redirect&bpo=23671].)

The **`platform.dist()`** and **`platform.linux_distribution()`** functions are now deprecated. Linux distributions use too many different ways of describing themselves, so the functionality is left to a package. (Contributed by Vajrasky Kok and Berker Peksag in bpo-1322 [https://bugs.python.org/issue?@action=redirect&bpo=1322].)

The previously undocumented `from_function` and `from_builtin` methods of `inspect.Signature` are deprecated. Use the new `Signature.from_callable()` method instead. (Contributed by Yury Selivanov in bpo-24248 [https://bugs.python.org/issue?@action=redirect&bpo=24248].)

The **`inspect.getargspec()`** function is deprecated and scheduled to be removed in Python 3.6. (See bpo-20438 [https://bugs.python.org/issue?@action=redirect&bpo=20438] for details.)

The `inspect` `getfullargspec()`, `getcallargs()`, and **`formatargspec()`** functions are deprecated in favor of the `inspect.signature()` API. (Contributed by Yury Selivanov in bpo-20438 [https://bugs.python.org/issue?@action=redirect&bpo=20438].)

`getargvalues()` and `formatargvalues()` functions were inadvertently marked as deprecated with the release of Python 3.5.0.

Use of **`re.LOCALE`** flag with str patterns or **`re.ASCII`** is now deprecated. (Contributed by Serhiy Storchaka in bpo-22407 [https://bugs.python.org/issue?@action=redirect&bpo=22407].)

Use of unrecognized special sequences consisting of `'\'` and an ASCII letter in regular expression patterns and replacement patterns now raises a deprecation warning and will be forbidden in Python 3.6. (Contributed by Serhiy Storchaka in bpo-23622 [https://bugs.python.org/issue?@action=redirect&bpo=23622].)

The undocumented and unofficial *use_load_tests* default argument of the **`unittest.TestLoader.loadTestsFromModule()`** method now is deprecated and ignored. (Contributed by Robert Collins and Barry A. Warsaw in bpo-16662 [https://bugs.python.org/issue?@action=redirect&bpo=16662].)

Removed

API and Feature Removals

The following obsolete and previously deprecated APIs and features have been removed:

- The `__version__` attribute has been dropped from the email package. The email code hasn't been shipped separately from the stdlib for a long time, and the `__version__` string was not updated in the last few releases.
- The internal `Netrc` class in the **`ftplib`** module was deprecated in 3.4, and has now been removed. (Contributed by Matt Chaput in bpo-6623 [https://bugs.python.org/issue?@action=redirect&bpo=6623].)
- The concept of `.pyo` files has been removed.
- The JoinableQueue class in the provisional **`asyncio`** module was deprecated in 3.4.4 and is now removed.

(Contributed by A. Jesse Jiryu Davis in bpo-23464 [https://bugs.python.org/issue?@action=redirect&bpo=23464].)

Porting to Python 3.5

This section lists previously described changes and other bugfixes that may require changes to your code.

Changes in Python behavior

- Due to an oversight, earlier Python versions erroneously accepted the following syntax:

  ```
  f(1 for x in [1], *args)
  f(1 for x in [1], **kwargs)
  ```

  Python 3.5 now correctly raises a `SyntaxError`, as generator expressions must be put in parentheses if not a sole argument to a function.

Changes in the Python API

- **PEP 475** [https://peps.python.org/pep-0475/]: System calls are now retried when interrupted by a signal instead of raising `InterruptedError` if the Python signal handler does not raise an exception.
- Before Python 3.5, a `datetime.time` object was considered to be false if it represented midnight in UTC. This behavior was considered obscure and error-prone and has been removed in Python 3.5. See bpo-13936 [https://bugs.python.org/issue?@action=redirect&bpo=13936] for full details.
- The **`ssl.SSLSocket.send()`** method now raises either `ssl.SSLWantReadError` or `ssl.SSLWantWriteError` on a non-blocking socket if the operation would block. Previously,

it would return `0`. (Contributed by Nikolaus Rath in bpo-20951 [https://bugs.python.org/issue?@action=redirect&bpo=20951].)

- The `__name__` attribute of generators is now set from the function name, instead of being set from the code name. Use `gen.gi_code.co_name` to retrieve the code name. Generators also have a new `__qualname__` attribute, the qualified name, which is now used for the representation of a generator (`repr(gen)`). (Contributed by Victor Stinner in bpo-21205 [https://bugs.python.org/issue?@action=redirect&bpo=21205].)
- The deprecated "strict" mode and argument of **`HTMLParser`**, **`HTMLParser.error()`**, and the **`HTMLParserError`** exception have been removed. (Contributed by Ezio Melotti in bpo-15114 [https://bugs.python.org/issue?@action=redirect&bpo=15114].) The *convert_charrefs* argument of **`HTMLParser`** is now `True` by default. (Contributed by Berker Peksag in bpo-21047 [https://bugs.python.org/issue?@action=redirect&bpo=21047].)
- Although it is not formally part of the API, it is worth noting for porting purposes (ie: fixing tests) that error messages that were previously of the form "'sometype' does not support the buffer protocol" are now of the form "a bytes-like object is required, not 'sometype'". (Contributed by Ezio Melotti in bpo-16518 [https://bugs.python.org/issue?@action=redirect&bpo=16518].)
- If the current directory is set to a directory that no longer exists then **`FileNotFoundError`** will no longer be raised and instead **`find_spec()`** will return `None` **without** caching `None` in **`sys.path_importer_cache`**, which is different than the typical case (bpo-22834 [https://bugs.python.org/issue?@action=redirect&bpo=22834]).
- HTTP status code and messages from **`http.client`** and **`http.server`** were refactored into a common **`HTTPStatus`** enum. The values in **`http.client`** and **`http.server`** remain available for backwards compatibility. (Contributed by

Demian Brecht in bpo-21793 [https://bugs.python.org/issue?@action=redirect&bpo=21793].)

- When an import loader defines **`exec_module()`** it is now expected to also define **`create_module()`** (raises a **`DeprecationWarning`** now, will be an error in Python 3.6). If the loader inherits from **`importlib.abc.Loader`** then there is nothing to do, else simply define **`create_module()`** to return `None`. (Contributed by Brett Cannon in bpo-23014 [https://bugs.python.org/issue?@action=redirect&bpo=23014].)
- The **`re.split()`** function always ignored empty pattern matches, so the `"x*"` pattern worked the same as `"x+"`, and the `"\b"` pattern never worked. Now **`re.split()`** raises a warning if the pattern could match an empty string. For compatibility, use patterns that never match an empty string (e.g. `"x+"` instead of `"x*"`). Patterns that could only match an empty string (such as `"\b"`) now raise an error. (Contributed by Serhiy Storchaka in bpo-22818 [https://bugs.python.org/issue?@action=redirect&bpo=22818].)
- The **`http.cookies.Morsel`** dict-like interface has been made self consistent: morsel comparison now takes the **`key`** and **`value`** into account, **`copy()`** now results in a **`Morsel`** instance rather than a **`dict`**, and **`update()`** will now raise an exception if any of the keys in the update dictionary are invalid. In addition, the undocumented *LegalChars* parameter of **`set()`** is deprecated and is now ignored. (Contributed by Demian Brecht in bpo-2211 [https://bugs.python.org/issue?@action=redirect&bpo=2211].)
- **PEP 488** [https://peps.python.org/pep-0488/] has removed `.pyo` files from Python and introduced the optional `opt-` tag in `.pyc` file names. The **`importlib.util.cache_from_source()`** has gained an *optimization* parameter to help control the `opt-` tag. Because of this, the *debug_override* parameter of the function is now deprecated. `.pyo` files are also no longer supported as a file argument to the Python interpreter and thus serve no purpose when distributed on their own (i.e. sourceless code

distribution). Due to the fact that the magic number for bytecode has changed in Python 3.5, all old `.pyo` files from previous versions of Python are invalid regardless of this PEP.

- The `socket` module now exports the `CAN_RAW_FD_FRAMES` constant on linux 3.6 and greater.
- The `ssl.cert_time_to_seconds()` function now interprets the input time as UTC and not as local time, per **RFC 5280** [https://datatracker.ietf.org/doc/html/rfc5280.html]. Additionally, the return value is always an `int`. (Contributed by Akira Li in bpo-19940 [https://bugs.python.org/issue?@action=redirect&bpo=19940].)
- The `pygettext.py` Tool now uses the standard +NNNN format for timezones in the POT-Creation-Date header.
- The `smtplib` module now uses `sys.stderr` instead of the previous module-level **`stderr`** variable for debug output. If your (test) program depends on patching the module-level variable to capture the debug output, you will need to update it to capture sys.stderr instead.
- The `str.startswith()` and `str.endswith()` methods no longer return `True` when finding the empty string and the indexes are completely out of range. (Contributed by Serhiy Storchaka in bpo-24284 [https://bugs.python.org/issue?@action=redirect&bpo=24284].)
- The `inspect.getdoc()` function now returns documentation strings inherited from base classes. Documentation strings no longer need to be duplicated if the inherited documentation is appropriate. To suppress an inherited string, an empty string must be specified (or the documentation may be filled in). This change affects the output of the `pydoc` module and the `help()` function. (Contributed by Serhiy Storchaka in bpo-15582 [https://bugs.python.org/issue?@action=redirect&bpo=15582].)
- Nested `functools.partial()` calls are now flattened. If you were relying on the previous behavior, you can now either add an attribute to a `functools.partial()` object or

you can create a subclass of **`functools.partial()`**. (Contributed by Alexander Belopolsky in bpo-7830 [https://bugs.python.org/issue?@action=redirect&bpo=7830].)

Changes in the C API

- The undocumented **`format`** member of the (non-public) **`PyMemoryViewObject`** structure has been removed. All extensions relying on the relevant parts in `memoryobject.h` must be rebuilt.
- The **`PyMemAllocator`** structure was renamed to **`PyMemAllocatorEx`** and a new `calloc` field was added.
- Removed non-documented macro **`PyObject_REPR()`** which leaked references. Use format character `%R` in **`PyUnicode_FromFormat()`**-like functions to format the **`repr()`** of the object. (Contributed by Serhiy Storchaka in bpo-22453 [https://bugs.python.org/issue?@action=redirect&bpo=22453].)
- Because the lack of the **`__module__`** attribute breaks pickling and introspection, a deprecation warning is now raised for builtin types without the **`__module__`** attribute. This will be an **`AttributeError`** in the future. (Contributed by Serhiy Storchaka in bpo-20204 [https://bugs.python.org/issue?@action=redirect&bpo=20204].)
- As part of the **PEP 492** [https://peps.python.org/pep-0492/] implementation, the `tp_reserved` slot of **`PyTypeObject`** was replaced with a **`tp_as_async`** slot. Refer to Coroutine Objects for new types, structures and functions.

Notable changes in Python 3.5.4

New `make regen-all` build target

To simplify cross-compilation, and to ensure that CPython can reliably be compiled without requiring an existing version of Python to already be available, the autotools-based build system no longer attempts to implicitly recompile generated files based on file modification times.

Instead, a new `make regen-all` command has been added to force regeneration of these files when desired (e.g. after an initial version of Python has already been built based on the pregenerated versions).

More selective regeneration targets are also defined - see Makefile.pre.in [https://github.com/python/cpython/tree/3.14/Makefile.pre.in] for details.

(Contributed by Victor Stinner in bpo-23404 [https://bugs.python.org/issue?@action=redirect&bpo=23404].)

Added in version 3.5.4.

Removal of `make touch` build target

The `make touch` build target previously used to request implicit regeneration of generated files by updating their modification times has been removed.

It has been replaced by the new `make regen-all` target.

(Contributed by Victor Stinner in bpo-23404 [https://bugs.python.org/issue?@action=redirect&bpo=23404].)

Changed in version 3.5.4.

What’s New In Python 3.4

Author:: R. David Murray <rdmurray@bitdance.com> (Editor)

This article explains the new features in Python 3.4, compared to 3.3. Python 3.4 was released on March 16, 2014. For full details, see the changelog [https://docs.python.org/3.4/whatsnew/changelog.html].

See also

PEP 429 [https://peps.python.org/pep-0429/] – Python 3.4 Release Schedule

Summary – Release Highlights

New syntax features:

- No new syntax features were added in Python 3.4.

Other new features:

- pip should always be available (**PEP 453** [https://peps.python.org/pep-0453/]).
- Newly created file descriptors are non-inheritable (**PEP 446** [https://peps.python.org/pep-0446/]).
- command line option for isolated mode (bpo-16499 [https://bugs.python.org/issue?@action=redirect&bpo=16499]).
- improvements in the handling of codecs that are not text encodings (multiple issues).

- A ModuleSpec Type for the Import System (**PEP 451** [https://peps.python.org/pep-0451/]). (Affects importer authors.)
- The **`marshal`** format has been made more compact and efficient (bpo-16475 [https://bugs.python.org/issue?@action=redirect&bpo=16475]).

New library modules:

- **`asyncio`**: New provisional API for asynchronous IO (**PEP 3156** [https://peps.python.org/pep-3156/]).
- **`ensurepip`**: Bootstrapping the pip installer (**PEP 453** [https://peps.python.org/pep-0453/]).
- **`enum`**: Support for enumeration types (**PEP 435** [https://peps.python.org/pep-0435/]).
- **`pathlib`**: Object-oriented filesystem paths (**PEP 428** [https://peps.python.org/pep-0428/]).
- **`selectors`**: High-level and efficient I/O multiplexing, built upon the **`select`** module primitives (part of **PEP 3156** [https://peps.python.org/pep-3156/]).
- **`statistics`**: A basic numerically stable statistics library (**PEP 450** [https://peps.python.org/pep-0450/]).
- **`tracemalloc`**: Trace Python memory allocations (**PEP 454** [https://peps.python.org/pep-0454/]).

Significantly improved library modules:

- Single-dispatch generic functions in **`functools`** (**PEP 443** [https://peps.python.org/pep-0443/]).
- New **`pickle`** protocol 4 (**PEP 3154** [https://peps.python.org/pep-3154/]).
- **`multiprocessing`** now has an option to avoid using os.fork on Unix (bpo-8713 [https://bugs.python.org/issue?@action=redirect&bpo=8713]).
- **`email`** has a new submodule, **`contentmanager`**, and a new **`Message`** subclass (**`EmailMessage`**) that simplify MIME

handling (bpo-18891 [https://bugs.python.org/issue?@action=redirect&bpo=18891]).
- The **`inspect`** and **`pydoc`** modules are now capable of correct introspection of a much wider variety of callable objects, which improves the output of the Python **`help()`** system.
- The **`ipaddress`** module API has been declared stable

Security improvements:

- Secure and interchangeable hash algorithm (**PEP 456** [https://peps.python.org/pep-0456/]).
- Make newly created file descriptors non-inheritable (**PEP 446** [https://peps.python.org/pep-0446/]) to avoid leaking file descriptors to child processes.
- New command line option for isolated mode, (bpo-16499 [https://bugs.python.org/issue?@action=redirect&bpo=16499]).
- **`multiprocessing`** now has an option to avoid using os.fork on Unix. *spawn* and *forkserver* are more secure because they avoid sharing data with child processes.
- **`multiprocessing`** child processes on Windows no longer inherit all of the parent's inheritable handles, only the necessary ones.
- A new **`hashlib.pbkdf2_hmac()`** function provides the PKCS#5 password-based key derivation function 2 [https://en.wikipedia.org/wiki/PBKDF2].
- TLSv1.1 and TLSv1.2 support for **`ssl`**.
- Retrieving certificates from the Windows system cert store support for **`ssl`**.
- Server-side SNI (Server Name Indication) support for **`ssl`**.
- The **`ssl.SSLContext`** class has a lot of improvements.
- All modules in the standard library that support SSL now support server certificate verification, including hostname matching (**`ssl.match_hostname()`**) and CRLs (Certificate Revocation lists, see **`ssl.SSLContext.load_verify_locations()`**).

CPython implementation improvements:

- Safe object finalization (**PEP 442** [https://peps.python.org/pep-0442/]).
- Leveraging **PEP 442** [https://peps.python.org/pep-0442/], in most cases module globals are no longer set to None during finalization (bpo-18214 [https://bugs.python.org/issue?@action=redirect&bpo=18214]).
- Configurable memory allocators (**PEP 445** [https://peps.python.org/pep-0445/]).
- Argument Clinic (**PEP 436** [https://peps.python.org/pep-0436/]).

Please read on for a comprehensive list of user-facing changes, including many other smaller improvements, CPython optimizations, deprecations, and potential porting issues.

New Features

PEP 453: Explicit Bootstrapping of PIP in Python Installations

Bootstrapping pip By Default

The new **`ensurepip`** module (defined in **PEP 453** [https://peps.python.org/pep-0453/]) provides a standard cross-platform mechanism to bootstrap the pip installer into Python installations and virtual environments. The version of `pip` included with Python 3.4.0 is `pip` 1.5.4, and future 3.4.x maintenance releases will update the bundled version to the latest version of `pip` that is available at the time of creating the release candidate.

By default, the commands `pipX` and `pipX.Y` will be installed on all platforms (where X.Y stands for the version of the Python installation), along with the `pip` Python package and its dependencies. On Windows and in virtual environments on all platforms, the unversioned `pip` command will also be installed. On other platforms, the system wide unversioned `pip` command typically refers to the separately installed Python 2 version.

The `pyvenv` command line utility and the **`venv`** module make use of the **`ensurepip`** module to make `pip` readily available in virtual environments. When using the command line utility, `pip` is installed by default, while when using the **`venv`** module API installation of `pip` must be requested explicitly.

For CPython source builds on POSIX systems, the `make install` and `make altinstall` commands bootstrap `pip` by default. This behaviour can be controlled through configure options, and overridden through Makefile options.

On Windows and Mac OS X, the CPython installers now default to installing `pip` along with CPython itself (users may opt out of installing it during the installation process). Window users will need to opt in to the automatic `PATH` modifications to have `pip` available from the command line by default, otherwise it can still be accessed through the Python launcher for Windows as `py -m pip`.

As **discussed in the PEP** [https://peps.python.org/pep-0453/#recommendations-for-downstream-distributors] platform packagers may choose not to install these commands by default, as long as, when invoked, they provide clear and simple directions on how to install them on that platform (usually using the system package manager).

Note

To avoid conflicts between parallel Python 2 and Python 3 installations, only the versioned `pip3` and `pip3.4` commands are bootstrapped by default when `ensurepip` is invoked directly - the `--default-pip` option is needed to also request the unversioned `pip` command. `pyvenv` and the Windows installer ensure that the unqualified `pip` command is made available in those environments, and `pip` can always be invoked via the `-m` switch rather than directly to avoid ambiguity on systems with multiple Python installations.

Documentation Changes

As part of this change, the Installing Python Modules and Distributing Python Modules sections of the documentation have been completely redesigned as short getting started and FAQ documents. Most packaging documentation has now been moved out to the Python Packaging Authority maintained Python Packaging User Guide [https://packaging.python.org] and the documentation of the individual projects.

However, as this migration is currently still incomplete, the legacy versions of those guides remaining available as Building C and C++ Extensions with setuptools and Building C and C++ Extensions with setuptools.

See also

PEP 453 [https://peps.python.org/pep-0453/] – Explicit bootstrapping of pip in Python installations

PEP written by Donald Stufft and Nick Coghlan, implemented by Donald Stufft, Nick Coghlan, Martin von Löwis and Ned Deily.

PEP 446: Newly Created File Descriptors Are Non-Inheritable

PEP 446 [https://peps.python.org/pep-0446/] makes newly created file descriptors non-inheritable. In general, this is the behavior an application will want: when launching a new process, having currently open files also open in the new process can lead to all sorts of hard to find bugs, and potentially to security issues.

However, there are occasions when inheritance is desired. To support these cases, the following new functions and methods are available:

- `os.get_inheritable()`, `os.set_inheritable()`
- `os.get_handle_inheritable()`, `os.set_handle_inheritable()`
- `socket.socket.get_inheritable()`, `socket.socket.set_inheritable()`

See also

PEP 446 [https://peps.python.org/pep-0446/] – Make newly created file descriptors non-inheritable

PEP written and implemented by Victor Stinner.

Improvements to Codec Handling

Since it was first introduced, the `codecs` module has always been intended to operate as a type-neutral dynamic encoding and decoding system. However, its close coupling with the Python text model, especially the type restricted convenience methods on the builtin `str`, `bytes` and `bytearray` types, has historically obscured that fact.

As a key step in clarifying the situation, the `codecs.encode()` and `codecs.decode()` convenience functions are now properly documented in Python 2.7, 3.3 and 3.4. These functions have existed in the `codecs` module (and have been covered by the regression test suite) since Python 2.4, but were previously only discoverable through runtime introspection.

Unlike the convenience methods on `str`, `bytes` and `bytearray`, the `codecs` convenience functions support arbitrary codecs in both Python 2 and Python 3, rather than being limited to Unicode text encodings (in Python 3) or `basestring` <-> `basestring` conversions (in Python 2).

In Python 3.4, the interpreter is able to identify the known non-text encodings provided in the standard library and direct users towards these general purpose convenience functions when appropriate:

```
>>> b"abcdef".decode("hex")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
LookupError: 'hex' is not a text encoding; use codecs.decode()
to handle arbitrary codecs

>>> "hello".encode("rot13")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
LookupError: 'rot13' is not a text encoding; use
codecs.encode() to handle arbitrary codecs

>>> open("foo.txt", encoding="hex")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
LookupError: 'hex' is not a text encoding; use codecs.open() to
handle arbitrary codecs
```

In a related change, whenever it is feasible without breaking backwards compatibility, exceptions raised during encoding and decoding operations are wrapped in a chained exception of the same type that mentions the name of the codec responsible for producing the error:

```
>>> import codecs

>>> codecs.decode(b"abcdefgh", "hex")
Traceback (most recent call last):
  File "/usr/lib/python3.4/encodings/hex_codec.py", line 20, in
hex_decode
    return (binascii.a2b_hex(input), len(input))
binascii.Error: Non-hexadecimal digit found

The above exception was the direct cause of the following
exception:

Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
binascii.Error: decoding with 'hex' codec failed (Error: Non-
hexadecimal digit found)

>>> codecs.encode("hello", "bz2")
Traceback (most recent call last):
  File "/usr/lib/python3.4/encodings/bz2_codec.py", line 17, in
bz2_encode
    return (bz2.compress(input), len(input))
  File "/usr/lib/python3.4/bz2.py", line 498, in compress
    return comp.compress(data) + comp.flush()
TypeError: 'str' does not support the buffer interface

The above exception was the direct cause of the following
exception:

Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: encoding with 'bz2' codec failed (TypeError: 'str'
does not support the buffer interface)
```

Finally, as the examples above show, these improvements have permitted the restoration of the convenience aliases for the non-Unicode codecs that were themselves restored in Python 3.2. This means that encoding binary data to and from its hexadecimal representation (for example) can now be written as:

```
>>> from codecs import encode, decode
>>> encode(b"hello", "hex")
b'68656c6c6f'
>>> decode(b"68656c6c6f", "hex")
b'hello'
```

The binary and text transforms provided in the standard library are detailed in Binary Transforms and Text Transforms.

(Contributed by Nick Coghlan in bpo-7475 [https://bugs.python.org/issue?@action=redirect&bpo=7475], bpo-17827 [https://bugs.python.org/issue?@action=redirect&bpo=17827], bpo-17828 [https://bugs.python.org/issue?@action=redirect&bpo=17828] and bpo-19619 [https://bugs.python.org/issue?@action=redirect&bpo=19619].)

PEP 451: A ModuleSpec Type for the Import System

PEP 451 [https://peps.python.org/pep-0451/] provides an encapsulation of the information about a module that the import machinery will use to load it (that is, a module specification). This helps simplify both the import implementation and several import-related APIs. The change is also a stepping stone for several future import-related improvements [https://mail.python.org/pipermail/python-dev/2013-November/130111.html].

The public-facing changes from the PEP are entirely backward-compatible. Furthermore, they should be transparent to everyone but importer authors. Key finder and loader methods have been deprecated, but they will continue working. New importers should use the new methods described in the PEP. Existing importers should be updated to implement the new methods. See the Deprecated section for a list of methods that should be replaced and their replacements.

Other Language Changes

Some smaller changes made to the core Python language are:

- Unicode database updated to UCD version 6.3.
- `min()` and `max()` now accept a *default* keyword-only argument that can be used to specify the value they return if the iterable they are evaluating has no elements. (Contributed by Julian Berman in bpo-18111 [https://bugs.python.org/issue?@action=redirect&bpo=18111].)
- Module objects are now weakly referenceable.
- Module `__file__` attributes (and related values) should now always contain absolute paths by default, with the sole exception of `__main__.__file__` when a script has been executed directly using a relative path. (Contributed by Brett Cannon in bpo-18416 [https://bugs.python.org/issue?@action=redirect&bpo=18416].)
- All the UTF-* codecs (except UTF-7) now reject surrogates during both encoding and decoding unless the `surrogatepass` error handler is used, with the exception of the UTF-16 decoder (which accepts valid surrogate pairs) and the UTF-16 encoder (which produces them while encoding non-BMP characters). (Contributed by Victor Stinner, Kang-Hao (Kenny) Lu and Serhiy Storchaka in bpo-12892 [https://bugs.python.org/issue?@action=redirect&bpo=12892].)
- New German EBCDIC codec `cp273`. (Contributed by Michael Bierenfeld and Andrew Kuchling in bpo-1097797 [https://bugs.python.org/issue?@action=redirect&bpo=1097797].)
- New Ukrainian codec `cp1125`. (Contributed by Serhiy Storchaka in bpo-19668 [https://bugs.python.org/issue?

@action=redirect&bpo=19668].)

- **bytes**.join() and **bytearray**.join() now accept arbitrary buffer objects as arguments. (Contributed by Antoine Pitrou in bpo-15958 [https://bugs.python.org/issue?@action=redirect&bpo=15958].)
- The **int** constructor now accepts any object that has an `__index__` method for its *base* argument. (Contributed by Mark Dickinson in bpo-16772 [https://bugs.python.org/issue?@action=redirect&bpo=16772].)
- Frame objects now have a **clear()** method that clears all references to local variables from the frame. (Contributed by Antoine Pitrou in bpo-17934 [https://bugs.python.org/issue?@action=redirect&bpo=17934].)
- **memoryview** is now registered as a **Sequence**, and supports the **reversed()** builtin. (Contributed by Nick Coghlan and Claudiu Popa in bpo-18690 [https://bugs.python.org/issue?@action=redirect&bpo=18690] and bpo-19078 [https://bugs.python.org/issue?@action=redirect&bpo=19078].)
- Signatures reported by **help()** have been modified and improved in several cases as a result of the introduction of Argument Clinic and other changes to the **inspect** and **pydoc** modules.
- **__length_hint__()** is now part of the formal language specification (see **PEP 424** [https://peps.python.org/pep-0424/]). (Contributed by Armin Ronacher in bpo-16148 [https://bugs.python.org/issue?@action=redirect&bpo=16148].)

New Modules

asyncio

The new **asyncio** module (defined in **PEP 3156** [https://peps.python.org/pep-3156/]) provides a standard pluggable event loop model for Python, providing solid asynchronous IO support in the standard library, and making it easier for other event loop implementations to interoperate with the standard library and each other.

For Python 3.4, this module is considered a provisional API.

See also

PEP 3156 [https://peps.python.org/pep-3156/] – Asynchronous IO Support Rebooted: the "asyncio" Module

PEP written and implementation led by Guido van Rossum.

ensurepip

The new `ensurepip` module is the primary infrastructure for the **PEP 453** [https://peps.python.org/pep-0453/] implementation. In the normal course of events end users will not need to interact with this module, but it can be used to manually bootstrap `pip` if the automated bootstrapping into an installation or virtual environment was declined.

`ensurepip` includes a bundled copy of `pip`, up-to-date as of the first release candidate of the release of CPython with which it ships (this applies to both maintenance releases and feature releases). `ensurepip` does not access the internet. If the installation has internet access, after `ensurepip` is run the bundled `pip` can be used to upgrade `pip` to a more recent release than the bundled one. (Note that such an upgraded version of `pip` is considered to be a separately installed package and will not be removed if Python is uninstalled.)

The module is named *ensure*pip because if called when `pip` is already installed, it does nothing. It also has an `--upgrade` option that will cause it to install the bundled copy of `pip` if the existing installed version of `pip` is older than the bundled copy.

enum

The new `enum` module (defined in **PEP 435** [https://peps.python.org/pep-0435/]) provides a standard implementation of enumeration types, allowing other modules (such as `socket`) to provide more informative error messages and better debugging support by replacing opaque integer constants with backwards compatible enumeration values.

See also

PEP 435 [https://peps.python.org/pep-0435/] – Adding an Enum type to the Python standard library

PEP written by Barry Warsaw, Eli Bendersky and Ethan Furman, implemented by Ethan Furman.

pathlib

The new `pathlib` module offers classes representing filesystem paths with semantics appropriate for different operating systems. Path classes are divided between *pure paths*, which provide purely computational operations without I/O, and *concrete paths*, which inherit from pure paths but also provide I/O operations.

For Python 3.4, this module is considered a provisional API.

See also

PEP 428 [https://peps.python.org/pep-0428/] – The pathlib module – object-oriented filesystem paths

PEP written and implemented by Antoine Pitrou.

selectors

The new `selectors` module (created as part of implementing **PEP 3156** [https://peps.python.org/pep-3156/]) allows high-level and efficient I/O multiplexing, built upon the `select` module primitives.

statistics

The new `statistics` module (defined in **PEP 450** [https://peps.python.org/pep-0450/]) offers some core statistics functionality directly in the standard library. This module supports calculation of the mean, median, mode, variance and standard deviation of a data series.

See also

PEP 450 [https://peps.python.org/pep-0450/] – Adding A Statistics Module To The Standard Library

PEP written and implemented by Steven D'Aprano

tracemalloc

The new `tracemalloc` module (defined in **PEP 454** [https://peps.python.org/pep-0454/]) is a debug tool to trace memory blocks allocated by Python. It provides the following information:

- Trace where an object was allocated
- Statistics on allocated memory blocks per filename and per line number: total size, number and average size of allocated memory blocks

- Compute the differences between two snapshots to detect memory leaks

See also

PEP 454 [https://peps.python.org/pep-0454/] – Add a new tracemalloc module to trace Python memory allocations

PEP written and implemented by Victor Stinner

Improved Modules

abc

New function **`abc.get_cache_token()`** can be used to know when to invalidate caches that are affected by changes in the object graph. (Contributed by Łukasz Langa in bpo-16832 [https://bugs.python.org/issue?@action=redirect&bpo=16832].)

New class **`ABC`** has **`ABCMeta`** as its meta class. Using `ABC` as a base class has essentially the same effect as specifying `metaclass=abc.ABCMeta`, but is simpler to type and easier to read. (Contributed by Bruno Dupuis in bpo-16049 [https://bugs.python.org/issue?@action=redirect&bpo=16049].)

aifc

The **`getparams()`** method now returns a namedtuple rather than a plain tuple. (Contributed by Claudiu Popa in bpo-17818 [https://bugs.python.org/issue?@action=redirect&bpo=17818].)

`aifc.open()` now supports the context management protocol: when used in a **`with`** block, the **`close()`** method of the returned object will be called automatically at the end of the block. (Contributed by Serhiy Storchacha in bpo-16486 [https://bugs.python.org/issue?@action=redirect&bpo=16486].)

The **writeframesraw()** and **writeframes()** methods now accept any bytes-like object. (Contributed by Serhiy Storchaka in bpo-8311 [https://bugs.python.org/issue?@action=redirect&bpo=8311].)

argparse

The **FileType** class now accepts *encoding* and *errors* arguments, which are passed through to **open()**. (Contributed by Lucas Maystre in bpo-11175 [https://bugs.python.org/issue?@action=redirect&bpo=11175].)

audioop

audioop now supports 24-bit samples. (Contributed by Serhiy Storchaka in bpo-12866 [https://bugs.python.org/issue?@action=redirect&bpo=12866].)

New **byteswap()** function converts big-endian samples to little-endian and vice versa. (Contributed by Serhiy Storchaka in bpo-19641 [https://bugs.python.org/issue?@action=redirect&bpo=19641].)

All **audioop** functions now accept any bytes-like object. Strings are not accepted: they didn't work before, now they raise an error right away. (Contributed by Serhiy Storchaka in bpo-16685 [https://bugs.python.org/issue?@action=redirect&bpo=16685].)

base64

The encoding and decoding functions in **base64** now accept any bytes-like object in cases where it previously required a `bytes` or `bytearray` instance. (Contributed by Nick Coghlan in bpo-17839 [https://bugs.python.org/issue?@action=redirect&bpo=17839].)

New functions `a85encode()`, `a85decode()`, `b85encode()`, and `b85decode()` provide the ability to encode and decode binary data from and to `Ascii85` and the git/mercurial `Base85` formats, respectively. The `a85` functions have options that can be used to make them compatible with the variants of the `Ascii85` encoding, including the Adobe variant. (Contributed by Martin Morrison, the Mercurial project, Serhiy Storchaka, and Antoine Pitrou in bpo-17618 [https://bugs.python.org/issue?@action=redirect&bpo=17618].)

collections

The `ChainMap.new_child()` method now accepts an *m* argument specifying the child map to add to the chain. This allows an existing mapping and/or a custom mapping type to be used for the child. (Contributed by Vinay Sajip in bpo-16613 [https://bugs.python.org/issue?@action=redirect&bpo=16613].)

colorsys

The number of digits in the coefficients for the RGB — YIQ conversions have been expanded so that they match the FCC NTSC versions. The change in results should be less than 1% and may better match results found elsewhere. (Contributed by Brian Landers and Serhiy Storchaka in bpo-14323 [https://bugs.python.org/issue?@action=redirect&bpo=14323].)

contextlib

The new `contextlib.suppress` context manager helps to clarify the intent of code that deliberately suppresses exceptions from a single statement. (Contributed by Raymond Hettinger in bpo-15806 [https://bugs.python.org/issue?

@action=redirect&bpo=15806] and Zero Piraeus in bpo-19266 [https://bugs.python.org/issue?@action=redirect&bpo=19266].)

The new **`contextlib.redirect_stdout()`** context manager makes it easier for utility scripts to handle inflexible APIs that write their output to **`sys.stdout`** and don't provide any options to redirect it. Using the context manager, the **`sys.stdout`** output can be redirected to any other stream or, in conjunction with **`io.StringIO`**, to a string. The latter can be especially useful, for example, to capture output from a function that was written to implement a command line interface. It is recommended only for utility scripts because it affects the global state of **`sys.stdout`**. (Contributed by Raymond Hettinger in bpo-15805 [https://bugs.python.org/issue?@action=redirect&bpo=15805].)

The **`contextlib`** documentation has also been updated to include a discussion of the differences between single use, reusable and reentrant context managers.

dbm

`dbm.open()` objects now support the context management protocol. When used in a **`with`** statement, the `close` method of the database object will be called automatically at the end of the block. (Contributed by Claudiu Popa and Nick Coghlan in bpo-19282 [https://bugs.python.org/issue?@action=redirect&bpo=19282].)

dis

Functions **`show_code()`**, **`dis()`**, **`distb()`**, and **`disassemble()`** now accept a keyword-only *file* argument that controls where they write their output.

The `dis` module is now built around an `Instruction` class that provides object oriented access to the details of each individual bytecode operation.

A new method, `get_instructions()`, provides an iterator that emits the Instruction stream for a given piece of Python code. Thus it is now possible to write a program that inspects and manipulates a bytecode object in ways different from those provided by the `dis` module itself. For example:

```
>>> import dis
>>> for instr in dis.get_instructions(lambda x: x + 1):
...     print(instr.opname)
LOAD_FAST
LOAD_CONST
BINARY_ADD
RETURN_VALUE
```

The various display tools in the `dis` module have been rewritten to use these new components.

In addition, a new application-friendly class `Bytecode` provides an object-oriented API for inspecting bytecode in both in human-readable form and for iterating over instructions. The `Bytecode` constructor takes the same arguments that `get_instructions()` does (plus an optional *current_offset*), and the resulting object can be iterated to produce `Instruction` objects. But it also has a `dis` method, equivalent to calling `dis` on the constructor argument, but returned as a multi-line string:

```
>>> bytecode = dis.Bytecode(lambda x: x + 1, current_offset=3)
>>> for instr in bytecode:
...     print('{} ({})'.format(instr.opname, instr.opcode))
LOAD_FAST (124)
LOAD_CONST (100)
BINARY_ADD (23)
RETURN_VALUE (83)
```

```
>>> bytecode.dis().splitlines()
['  1           0 LOAD_FAST                0 (x)',
 '      -->     3 LOAD_CONST               1 (1)',
 '              6 BINARY_ADD',
 '              7 RETURN_VALUE']
```

`Bytecode` also has a class method, **`from_traceback()`**, that provides the ability to manipulate a traceback (that is, `print(Bytecode.from_traceback(tb).dis())` is equivalent to `distb(tb)`).

(Contributed by Nick Coghlan, Ryan Kelly and Thomas Kluyver in bpo-11816 [https://bugs.python.org/issue?@action=redirect&bpo=11816] and Claudiu Popa in bpo-17916 [https://bugs.python.org/issue?@action=redirect&bpo=17916].)

New function **`stack_effect()`** computes the effect on the Python stack of a given opcode and argument, information that is not otherwise available. (Contributed by Larry Hastings in bpo-19722 [https://bugs.python.org/issue?@action=redirect&bpo=19722].)

doctest

A new option flag, **`FAIL_FAST`**, halts test running as soon as the first failure is detected. (Contributed by R. David Murray and Daniel Urban in bpo-16522 [https://bugs.python.org/issue?@action=redirect&bpo=16522].)

The **`doctest`** command line interface now uses **`argparse`**, and has two new options, `-o` and `-f`. `-o` allows doctest options to be specified on the command line, and `-f` is a shorthand for `-o FAIL_FAST` (to parallel the similar option supported by the **`unittest`** CLI). (Contributed by R. David Murray in bpo-11390 [https://bugs.python.org/issue?@action=redirect&bpo=11390].)

`doctest` will now find doctests in extension module `__doc__` strings. (Contributed by Zachary Ware in bpo-3158 [https://bugs.python.org/issue?@action=redirect&bpo=3158].)

email

`as_string()` now accepts a *policy* argument to override the default policy of the message when generating a string representation of it. This means that `as_string` can now be used in more circumstances, instead of having to create and use a **`generator`** in order to pass formatting parameters to its `flatten` method. (Contributed by R. David Murray in bpo-18600 [https://bugs.python.org/issue?@action=redirect&bpo=18600].)

New method **`as_bytes()`** added to produce a bytes representation of the message in a fashion similar to how `as_string` produces a string representation. It does not accept the *maxheaderlen* argument, but does accept the *unixfrom* and *policy* arguments. The **`Message __bytes__()`** method calls it, meaning that `bytes(mymsg)` will now produce the intuitive result: a bytes object containing the fully formatted message. (Contributed by R. David Murray in bpo-18600 [https://bugs.python.org/issue?@action=redirect&bpo=18600].)

The **`Message.set_param()`** message now accepts a *replace* keyword argument. When specified, the associated header will be updated without changing its location in the list of headers. For backward compatibility, the default is `False`. (Contributed by R. David Murray in bpo-18891 [https://bugs.python.org/issue?@action=redirect&bpo=18891].)

A pair of new subclasses of **`Message`** have been added (**`EmailMessage`** and **`MIMEPart`**), along with a new sub-module, **`contentmanager`** and a new **`policy`** attribute **`content_manager`**. All documentation is currently in the new module, which is being added as part of email's new provisional API. These classes provide a number of new methods that make extracting content from and inserting content into email messages much easier. For details, see the `contentmanager` documentation and the email: Examples. These API additions complete the bulk of the work that was planned as part of the email6 project. The currently provisional API is scheduled to become final in Python 3.5 (possibly with a few minor additions in the area of error handling). (Contributed by R. David Murray in bpo-18891 [https://bugs.python.org/issue?@action=redirect&bpo=18891].)

filecmp

A new `clear_cache()` function provides the ability to clear the `filecmp` comparison cache, which uses `os.stat()` information to determine if the file has changed since the last compare. This can be used, for example, if the file might have been changed and re-checked in less time than the resolution of a particular filesystem's file modification time field. (Contributed by Mark Levitt in bpo-18149 [https://bugs.python.org/issue?@action=redirect&bpo=18149].)

New module attribute `DEFAULT_IGNORES` provides the list of directories that are used as the default value for the *ignore* parameter of the `dircmp()` function. (Contributed by Eli Bendersky in bpo-15442 [https://bugs.python.org/issue?@action=redirect&bpo=15442].)

functools

The new `partialmethod()` descriptor brings partial argument application to descriptors, just as `partial()` provides for normal callables. The new descriptor also makes it easier to get arbitrary callables (including `partial()` instances) to behave like normal instance methods when included in a class definition. (Contributed by Alon Horev and Nick

Coghlan in bpo-4331 [https://bugs.python.org/issue?@action=redirect&bpo=4331].)

The new `singledispatch()` decorator brings support for single-dispatch generic functions to the Python standard library. Where object oriented programming focuses on grouping multiple operations on a common set of data into a class, a generic function focuses on grouping multiple implementations of an operation that allows it to work with *different* kinds of data.

See also

PEP 443 [https://peps.python.org/pep-0443/] – Single-dispatch generic functions

PEP written and implemented by Łukasz Langa.

`total_ordering()` now supports a return value of `NotImplemented` from the underlying comparison function. (Contributed by Katie Miller in bpo-10042 [https://bugs.python.org/issue?@action=redirect&bpo=10042].)

A pure-python version of the `partial()` function is now in the stdlib; in CPython it is overridden by the C accelerated version, but it is available for other implementations to use. (Contributed by Brian Thorne in bpo-12428 [https://bugs.python.org/issue?@action=redirect&bpo=12428].)

gc

New function `get_stats()` returns a list of three per-generation dictionaries containing the collections statistics since interpreter startup. (Contributed by Antoine Pitrou in bpo-16351 [https://bugs.python.org/issue?@action=redirect&bpo=16351].)

glob

A new function **`escape()`** provides a way to escape special characters in a filename so that they do not become part of the globbing expansion but are instead matched literally. (Contributed by Serhiy Storchaka in bpo-8402 [https://bugs.python.org/issue?@action=redirect&bpo=8402].)

hashlib

A new **`hashlib.pbkdf2_hmac()`** function provides the PKCS#5 password-based key derivation function 2 [https://en.wikipedia.org/wiki/PBKDF2]. (Contributed by Christian Heimes in bpo-18582 [https://bugs.python.org/issue?@action=redirect&bpo=18582].)

The **`name`** attribute of **`hashlib`** hash objects is now a formally supported interface. It has always existed in CPython's **`hashlib`** (although it did not return lower case names for all supported hashes), but it was not a public interface and so some other Python implementations have not previously supported it. (Contributed by Jason R. Coombs in bpo-18532 [https://bugs.python.org/issue?@action=redirect&bpo=18532].)

hmac

`hmac` now accepts `bytearray` as well as `bytes` for the *key* argument to the **`new()`** function, and the *msg* parameter to both the **`new()`** function and the **`update()`** method now accepts any type supported by the **`hashlib`** module. (Contributed by Jonas Borgström in bpo-18240 [https://bugs.python.org/issue?@action=redirect&bpo=18240].)

The *digestmod* argument to the **`hmac.new()`** function may now be any hash digest name recognized by **`hashlib`**. In addition, the current behavior in which the value of *digestmod* defaults to `MD5` is deprecated: in a future version of Python there will be no default value. (Contributed by Christian Heimes in bpo-17276 [https://bugs.python.org/issue?@action=redirect&bpo=17276].)

With the addition of **`block_size`** and **`name`** attributes (and the formal documentation of the **`digest_size`** attribute), the **`hmac`** module now conforms fully to the **PEP 247** [https://peps.python.org/pep-0247/] API. (Contributed by Christian Heimes in bpo-18775 [https://bugs.python.org/issue?@action=redirect&bpo=18775].)

html

New function **`unescape()`** function converts HTML5 character references to the corresponding Unicode characters. (Contributed by Ezio Melotti in bpo-2927 [https://bugs.python.org/issue?@action=redirect&bpo=2927].)

`HTMLParser` accepts a new keyword argument *convert_charrefs* that, when `True`, automatically converts all character references. For backward-compatibility, its value defaults to `False`, but it will change to `True` in a future version of Python, so you are invited to set it explicitly and update your code to use this new feature. (Contributed by Ezio Melotti in bpo-13633 [https://bugs.python.org/issue?@action=redirect&bpo=13633].)

The *strict* argument of **`HTMLParser`** is now deprecated. (Contributed by Ezio Melotti in bpo-15114 [https://bugs.python.org/issue?@action=redirect&bpo=15114].)

http

`send_error()` now accepts an optional additional *explain* parameter which can be used to provide an extended error description, overriding the hardcoded default if there is one. This extended error description will be formatted using the **`error_message_format`** attribute and sent as the body of the error response. (Contributed by Karl Cow in bpo-12921 [https://bugs.python.org/issue?@action=redirect&bpo=12921].)

The **`http.server`** command line interface now has a `-b/--bind` option that causes the server to listen on a specific address. (Contributed by Malte Swart in bpo-17764 [https://bugs.python.org/issue?@action=redirect&bpo=17764].)

idlelib and IDLE

Since idlelib implements the IDLE shell and editor and is not intended for import by other programs, it gets improvements with every release. See `Lib/idlelib/NEWS.txt` for a cumulative list of changes since 3.3.0, as well as changes made in future 3.4.x releases. This file is also available from the IDLE *Help* ‣ *About IDLE* dialog.

importlib

The **`InspectLoader`** ABC defines a new method, **`source_to_code()`** that accepts source data and a path and returns a code object. The default implementation is equivalent to `compile(data, path, 'exec', dont_inherit=True)`. (Contributed by Eric Snow and Brett Cannon in bpo-15627 [https://bugs.python.org/issue?@action=redirect&bpo=15627].)

`InspectLoader` also now has a default implementation for the **`get_code()`** method. However, it will normally be desirable to override the default implementation for performance reasons. (Contributed by Brett Cannon in bpo-18072 [https://bugs.python.org/issue?@action=redirect&bpo=18072].)

The `reload()` function has been moved from **`imp`** to `importlib` as part of the **`imp`** module deprecation. (Contributed by Berker Peksag in bpo-18193 [https://bugs.python.org/issue?@action=redirect&bpo=18193].)

`importlib.util` now has a `MAGIC_NUMBER` attribute providing access to the bytecode version number. This replaces the **`get_magic()`** function in the deprecated **`imp`** module. (Contributed by Brett Cannon in bpo-18192 [https://bugs.python.org/issue?@action=redirect&bpo=18192].)

New `importlib.util` functions `cache_from_source()` and `source_from_cache()` replace the same-named functions in the deprecated **`imp`** module. (Contributed by Brett Cannon in bpo-18194 [https://bugs.python.org/issue?@action=redirect&bpo=18194].)

The `importlib` bootstrap `NamespaceLoader` now conforms to the `InspectLoader` ABC, which means that `runpy` and `python -m` can now be used with namespace packages. (Contributed by Brett Cannon in bpo-18058 [https://bugs.python.org/issue?@action=redirect&bpo=18058].)

`importlib.util` has a new function `decode_source()` that decodes source from bytes using universal newline processing. This is useful for implementing `InspectLoader.get_source()` methods.

`importlib.machinery.ExtensionFileLoader` now has a `get_filename()` method. This was inadvertently omitted in the original implementation. (Contributed by Eric Snow in bpo-19152 [https://bugs.python.org/issue?@action=redirect&bpo=19152].)

inspect

The `inspect` module now offers a basic command line interface to quickly display source code and other information for modules, classes and functions. (Contributed by Claudiu Popa and Nick Coghlan in bpo-18626 [https://bugs.python.org/issue?@action=redirect&bpo=18626].)

`unwrap()` makes it easy to unravel wrapper function chains created by **`functools.wraps()`** (and any other API that sets the `__wrapped__` attribute on a wrapper function). (Contributed by Daniel Urban, Aaron Iles and Nick Coghlan in bpo-13266 [https://bugs.python.org/issue?@action=redirect&bpo=13266].)

As part of the implementation of the new **`enum`** module, the **`inspect`** module now has substantially better support for custom `__dir__` methods and dynamic class attributes provided through metaclasses. (Contributed by Ethan Furman in bpo-18929 [https://bugs.python.org/issue?@action=redirect&bpo=18929] and bpo-19030 [https://bugs.python.org/issue?@action=redirect&bpo=19030].)

`getfullargspec()` and **`getargspec()`** now use the **`signature()`** API. This allows them to support a much broader range of callables, including those with `__signature__` attributes, those with metadata provided by argument clinic, **`functools.partial()`** objects and more. Note that, unlike **`signature()`**, these functions still ignore `__wrapped__` attributes, and report the already bound first argument for bound methods, so it is still necessary to update your code to use **`signature()`** directly if those features are desired. (Contributed by Yury Selivanov in bpo-17481 [https://bugs.python.org/issue?@action=redirect&bpo=17481].)

`signature()` now supports duck types of CPython functions, which adds support for functions compiled with Cython. (Contributed by Stefan Behnel and Yury Selivanov in bpo-17159 [https://bugs.python.org/issue?@action=redirect&bpo=17159].)

ipaddress

`ipaddress` was added to the standard library in Python 3.3 as a provisional API. With the release of Python 3.4, this qualification has been removed: **`ipaddress`** is now considered a stable API, covered by the normal standard library requirements to maintain backwards compatibility.

A new **`is_global`** property is `True` if an address is globally routeable. (Contributed by Peter Moody in bpo-17400 [https://bugs.python.org/issue?@action=redirect&bpo=17400].)

logging

The **`TimedRotatingFileHandler`** has a new *atTime* parameter that can be used to specify the time of day when rollover should happen. (Contributed by Ronald Oussoren in bpo-9556 [https://bugs.python.org/issue?@action=redirect&bpo=9556].)

`SocketHandler` and **`DatagramHandler`** now support Unix domain sockets (by setting *port* to `None`). (Contributed by Vinay Sajip in commit ce46195b56a9.)

`fileConfig()` now accepts a **`configparser.RawConfigParser`** subclass instance for the *fname* parameter. This facilitates using a configuration file when logging configuration is just a part of the overall application configuration, or where the application modifies the configuration before passing it to **`fileConfig()`**. (Contributed by Vinay Sajip in bpo-16110 [https://bugs.python.org/issue?@action=redirect&bpo=16110].)

Logging configuration data received from a socket via the **`logging.config.listen()`** function can now be validated before being processed by supplying a verification function as the argument to the new *verify* keyword argument. (Contributed by Vinay Sajip in bpo-15452 [https://bugs.python.org/issue?@action=redirect&bpo=15452].)

marshal

The default `marshal` version has been bumped to 3. The code implementing the new version restores the Python2 behavior of recording only one copy of interned strings and preserving the interning on deserialization, and extends this “one copy” ability to any object type (including handling recursive references). This reduces both the size of `.pyc` files and the amount of memory a module occupies in memory when it is loaded from a `.pyc` (or `.pyo`) file. (Contributed by Kristján Valur Jónsson in bpo-16475 [https://bugs.python.org/issue?@action=redirect&bpo=16475], with additional speedups by Antoine Pitrou in bpo-19219 [https://bugs.python.org/issue?@action=redirect&bpo=19219].)

mmap

mmap objects are now weakly referenceable. (Contributed by Valerie Lambert in bpo-4885 [https://bugs.python.org/issue?@action=redirect&bpo=4885].)

multiprocessing

On Unix two new start methods, `spawn` and `forkserver`, have been added for starting processes using `multiprocessing`. These make the mixing of processes with threads more robust, and the `spawn` method matches the semantics that multiprocessing has always used on Windows. New function `get_all_start_methods()` reports all start methods available on the platform, `get_start_method()` reports the current start method, and `set_start_method()` sets the start method. (Contributed by Richard Oudkerk in bpo-8713 [https://bugs.python.org/issue?@action=redirect&bpo=8713].)

`multiprocessing` also now has the concept of a `context`, which determines how child processes are created. New function `get_context()` returns a context that uses a specified start method. It has the same API as the `multiprocessing` module itself, so you can use it to create `Pool`s and other objects that will operate within that context. This allows a framework and an application or different parts of the same application to use multiprocessing without interfering with each other. (Contributed by Richard Oudkerk in bpo-18999 [https://bugs.python.org/issue?@action=redirect&bpo=18999].)

Except when using the old *fork* start method, child processes no longer inherit unneeded handles/file descriptors from their parents (part of bpo-8713 [https://bugs.python.org/issue?@action=redirect&bpo=8713]).

`multiprocessing` now relies on `runpy` (which implements the `-m` switch) to initialise `__main__` appropriately in child processes when using the `spawn` or `forkserver` start methods. This resolves some edge cases where combining multiprocessing, the `-m` command line switch, and explicit relative imports could cause obscure failures in child processes. (Contributed by Nick Coghlan in bpo-19946 [https://bugs.python.org/issue?@action=redirect&bpo=19946].)

operator

New function `length_hint()` provides an implementation of the specification for how the `__length_hint__()` special method should be used, as part of the **PEP 424** [https://peps.python.org/pep-0424/] formal specification of this language feature. (Contributed by Armin Ronacher in bpo-16148 [https://bugs.python.org/issue?@action=redirect&bpo=16148].)

There is now a pure-python version of the `operator` module available for reference and for use by alternate implementations of Python. (Contributed by Zachary Ware in bpo-16694 [https://bugs.python.org/issue?@action=redirect&bpo=16694].)

os

There are new functions to get and set the inheritable flag of a file descriptor (`os.get_inheritable()`, `os.set_inheritable()`) or a Windows handle (`os.get_handle_inheritable()`, `os.set_handle_inheritable()`).

New function `cpu_count()` reports the number of CPUs available on the platform on which Python is running (or `None` if the count can't be determined). The `multiprocessing.cpu_count()` function is now implemented in terms of this function). (Contributed by Trent Nelson, Yogesh Chaudhari, Victor Stinner, and Charles-François Natali in bpo-17914 [https://bugs.python.org/issue?@action=redirect&bpo=17914].)

`os.path.samestat()` is now available on the Windows platform (and the `os.path.samefile()` implementation is now shared between Unix and Windows). (Contributed by Brian Curtin in bpo-11939 [https://bugs.python.org/issue?@action=redirect&bpo=11939].)

`os.path.ismount()` now recognizes volumes mounted below a drive root on Windows. (Contributed by Tim Golden in bpo-9035 [https://bugs.python.org/issue?@action=redirect&bpo=9035].)

`os.open()` supports two new flags on platforms that provide them, `O_PATH` (un-opened file descriptor), and `O_TMPFILE` (unnamed temporary file; as of 3.4.0 release available only on Linux systems with a kernel version of 3.11 or newer that have uapi headers). (Contributed by Christian Heimes in bpo-18673 [https://bugs.python.org/issue?@action=redirect&bpo=18673] and Benjamin Peterson, respectively.)

pdb

`pdb` has been enhanced to handle generators, `yield`, and `yield from` in a more useful fashion. This is especially helpful when debugging `asyncio` based programs. (Contributed by Andrew Svetlov and Xavier de Gaye in bpo-16596 [https://bugs.python.org/issue?@action=redirect&bpo=16596].)

The `print` command has been removed from `pdb`, restoring access to the Python `print()` function from the pdb command line. Python2's `pdb` did not have a `print` command; instead, entering `print` executed the `print` statement. In Python3 `print` was mistakenly made an alias for the pdb `p` command. `p`, however, prints the `repr` of its argument, not the `str` like the Python2 `print` command did. Worse, the Python3 `pdb print` command shadowed the Python3 `print` function, making it inaccessible at the `pdb` prompt. (Contributed by Connor Osborn in bpo-18764 [https://bugs.python.org/issue?@action=redirect&bpo=18764].)

pickle

`pickle` now supports (but does not use by default) a new pickle protocol, protocol 4. This new protocol addresses a number of issues that were present in previous protocols, such as the serialization of nested classes, very large strings and containers, and classes whose `__new__()` method takes keyword-only arguments. It also provides some efficiency improvements.

See also

PEP 3154 [https://peps.python.org/pep-3154/] – Pickle protocol 4

PEP written by Antoine Pitrou and implemented by Alexandre Vassalotti.

plistlib

`plistlib` now has an API that is similar to the standard pattern for stdlib serialization protocols, with new `load()`, `dump()`, `loads()`, and `dumps()` functions. (The older API is now deprecated.) In addition to the already supported XML plist format (`FMT_XML`), it also now supports the binary plist format (`FMT_BINARY`). (Contributed by Ronald Oussoren and others in bpo-14455 [https://bugs.python.org/issue?@action=redirect&bpo=14455].)

poplib

Two new methods have been added to `poplib`: `capa()`, which returns the list of capabilities advertised by the POP server, and `stls()`, which switches a clear-text POP3 session into an encrypted POP3 session if the POP server supports it. (Contributed by Lorenzo Catucci in bpo-4473 [https://bugs.python.org/issue?@action=redirect&bpo=4473].)

pprint

The `pprint` module's `PrettyPrinter` class and its `pformat()`, and `pprint()` functions have a new option, *compact*, that controls how the output is formatted. Currently setting *compact* to `True` means that sequences will be printed with as many sequence elements as will fit within *width* on each (indented) line. (Contributed by Serhiy Storchaka in bpo-19132 [https://bugs.python.org/issue?@action=redirect&bpo=19132].)

Long strings are now wrapped using Python's normal line continuation syntax. (Contributed by Antoine Pitrou in bpo-17150 [https://bugs.python.org/issue?@action=redirect&bpo=17150].)

pty

`pty.spawn()` now returns the status value from `os.waitpid()` on the child process, instead of `None`. (Contributed by Gregory P. Smith.)

pydoc

The **`pydoc`** module is now based directly on the **`inspect.signature()`** introspection API, allowing it to provide signature information for a wider variety of callable objects. This change also means that `__wrapped__` attributes are now taken into account when displaying help information. (Contributed by Larry Hastings in bpo-19674 [https://bugs.python.org/issue?@action=redirect&bpo=19674].)

The **`pydoc`** module no longer displays the `self` parameter for already bound methods. Instead, it aims to always display the exact current signature of the supplied callable. (Contributed by Larry Hastings in bpo-20710 [https://bugs.python.org/issue?@action=redirect&bpo=20710].)

In addition to the changes that have been made to **`pydoc`** directly, its handling of custom `__dir__` methods and various descriptor behaviours has also been improved substantially by the underlying changes in the **`inspect`** module.

As the **`help()`** builtin is based on **`pydoc`**, the above changes also affect the behaviour of **`help()`**.

re

New **`fullmatch()`** function and **`Pattern.fullmatch()`** method anchor the pattern at both ends of the string to match. This provides a way to be explicit about the goal of the match, which avoids a class of subtle bugs where `$` characters get lost during code changes or the addition of alternatives to an existing regular expression. (Contributed by Matthew Barnett in bpo-16203 [https://bugs.python.org/issue?@action=redirect&bpo=16203].)

The repr of regex objects now includes the pattern and the flags; the repr of match objects now includes the start, end, and the part of the string that matched. (Contributed by Hugo Lopes Tavares and Serhiy Storchaka in bpo-13592 [https://bugs.python.org/issue?@action=redirect&bpo=13592] and bpo-17087 [https://bugs.python.org/issue?@action=redirect&bpo=17087].)

resource

New `prlimit()` function, available on Linux platforms with a kernel version of 2.6.36 or later and glibc of 2.13 or later, provides the ability to query or set the resource limits for processes other than the one making the call. (Contributed by Christian Heimes in bpo-16595 [https://bugs.python.org/issue?@action=redirect&bpo=16595].)

On Linux kernel version 2.6.36 or later, there are also some new Linux specific constants: `RLIMIT_MSGQUEUE`, `RLIMIT_NICE`, `RLIMIT_RTPRIO`, `RLIMIT_RTTIME`, and `RLIMIT_SIGPENDING`. (Contributed by Christian Heimes in bpo-19324 [https://bugs.python.org/issue?@action=redirect&bpo=19324].)

On FreeBSD version 9 and later, there some new FreeBSD specific constants: `RLIMIT_SBSIZE`, `RLIMIT_SWAP`, and `RLIMIT_NPTS`. (Contributed by Claudiu Popa in bpo-19343 [https://bugs.python.org/issue?@action=redirect&bpo=19343].)

select

`epoll` objects now support the context management protocol. When used in a `with` statement, the `close()` method will be called automatically at the end of the block. (Contributed by Serhiy Storchaka in bpo-16488 [https://bugs.python.org/issue?@action=redirect&bpo=16488].)

`devpoll` objects now have `fileno()` and `close()` methods, as well as a new attribute `closed`. (Contributed by Victor Stinner in bpo-18794 [https://bugs.python.org/issue?@action=redirect&bpo=18794].)

shelve

`Shelf` instances may now be used in `with` statements, and will be automatically closed at the end of the **`with`** block. (Contributed by Filip Gruszczyński in bpo-13896 [https://bugs.python.org/issue?@action=redirect&bpo=13896].)

shutil

`copyfile()` now raises a specific `Error` subclass, `SameFileError`, when the source and destination are the same file, which allows an application to take appropriate action on this specific error. (Contributed by Atsuo Ishimoto and Hynek Schlawack in bpo-1492704 [https://bugs.python.org/issue?@action=redirect&bpo=1492704].)

smtpd

The **`SMTPServer`** and **`SMTPChannel`** classes now accept a *map* keyword argument which, if specified, is passed in to **`asynchat.async_chat`** as its *map* argument. This allows an application to avoid affecting the global socket map. (Contributed by Vinay Sajip in bpo-11959 [https://bugs.python.org/issue?@action=redirect&bpo=11959].)

smtplib

`SMTPException` is now a subclass of `OSError`, which allows both socket level errors and SMTP protocol level errors to be caught in one try/except statement by code that only cares whether or not an error occurred. (Contributed by Ned

Jackson Lovely in bpo-2118 [https://bugs.python.org/issue?@action=redirect&bpo=2118].)

socket

The socket module now supports the **`CAN_BCM`** protocol on platforms that support it. (Contributed by Brian Thorne in bpo-15359 [https://bugs.python.org/issue?@action=redirect&bpo=15359].)

Socket objects have new methods to get or set their inheritable flag, **`get_inheritable()`** and **`set_inheritable()`**.

The `socket.AF_*` and `socket.SOCK_*` constants are now enumeration values using the new **`enum`** module. This allows meaningful names to be printed during debugging, instead of integer “magic numbers”.

The **`AF_LINK`** constant is now available on BSD and OSX.

`inet_pton()` and **`inet_ntop()`** are now supported on Windows. (Contributed by Atsuo Ishimoto in bpo-7171 [https://bugs.python.org/issue?@action=redirect&bpo=7171].)

sqlite3

A new boolean parameter to the **`connect()`** function, *uri*, can be used to indicate that the *database* parameter is a `uri` (see the SQLite URI documentation [https://www.sqlite.org/uri.html]). (Contributed by poq in bpo-13773 [https://bugs.python.org/issue?@action=redirect&bpo=13773].)

ssl

`PROTOCOL_TLSv1_1` and **`PROTOCOL_TLSv1_2`** (TLSv1.1 and TLSv1.2 support) have been added; support for these protocols is only available if Python is linked with OpenSSL 1.0.1 or later.

(Contributed by Michele Orrù and Antoine Pitrou in bpo-16692 [https://bugs.python.org/issue?@action=redirect&bpo=16692].)

New function `create_default_context()` provides a standard way to obtain an `SSLContext` whose settings are intended to be a reasonable balance between compatibility and security. These settings are more stringent than the defaults provided by the `SSLContext` constructor, and may be adjusted in the future, without prior deprecation, if best-practice security requirements change. The new recommended best practice for using stdlib libraries that support SSL is to use `create_default_context()` to obtain an `SSLContext` object, modify it if needed, and then pass it as the *context* argument of the appropriate stdlib API. (Contributed by Christian Heimes in bpo-19689 [https://bugs.python.org/issue?@action=redirect&bpo=19689].)

`SSLContext` method `load_verify_locations()` accepts a new optional argument *cadata*, which can be used to provide PEM or DER encoded certificates directly via strings or bytes, respectively. (Contributed by Christian Heimes in bpo-18138 [https://bugs.python.org/issue?@action=redirect&bpo=18138].)

New function `get_default_verify_paths()` returns a named tuple of the paths and environment variables that the `set_default_verify_paths()` method uses to set OpenSSL's default `cafile` and `capath`. This can be an aid in debugging default verification issues. (Contributed by Christian Heimes in bpo-18143 [https://bugs.python.org/issue?@action=redirect&bpo=18143].)

`SSLContext` has a new method, `cert_store_stats()`, that reports the number of loaded `X.509` certs, `X.509 CA` certs, and certificate revocation lists (`crls`), as well as a `get_ca_certs()` method that returns a list of the loaded `CA` certificates. (Contributed by Christian Heimes in bpo-18147 [https://bugs.python.org/issue?@action=redirect&bpo=18147].)

If OpenSSL 0.9.8 or later is available, `SSLContext` has a new attribute `verify_flags` that can be used to control the certificate verification process by setting it to some combination of the new constants `VERIFY_DEFAULT`, `VERIFY_CRL_CHECK_LEAF`, `VERIFY_CRL_CHECK_CHAIN`, or `VERIFY_X509_STRICT`. OpenSSL does not do any CRL verification by default. (Contributed by Christien Heimes in bpo-8813 [https://bugs.python.org/issue?@action=redirect&bpo=8813].)

New `SSLContext` method `load_default_certs()` loads a set of default "certificate authority" (CA) certificates from default locations, which vary according to the platform. It can be used to load both TLS web server authentication certificates (`purpose=SERVER_AUTH`) for a client to use to verify a server, and certificates for a server to use in verifying client certificates (`purpose=CLIENT_AUTH`). (Contributed by Christian Heimes in bpo-19292 [https://bugs.python.org/issue?@action=redirect&bpo=19292].)

Two new windows-only functions, `enum_certificates()` and `enum_crls()` provide the ability to retrieve certificates, certificate information, and CRLs from the Windows cert store. (Contributed by Christian Heimes in bpo-17134 [https://bugs.python.org/issue?@action=redirect&bpo=17134].)

Support for server-side SNI (Server Name Indication) using the new `ssl.SSLContext.set_servername_callback()` method. (Contributed by Daniel Black in bpo-8109 [https://bugs.python.org/issue?@action=redirect&bpo=8109].)

The dictionary returned by `SSLSocket.getpeercert()` contains additional `X509v3` extension items: `crlDistributionPoints`, `calIssuers`, and `OCSP` URIs. (Contributed by Christian Heimes in bpo-18379 [https://bugs.python.org/issue?@action=redirect&bpo=18379].)

stat

The **`stat`** module is now backed by a C implementation in **`_stat`**. A C implementation is required as most of the values aren't standardized and are platform-dependent. (Contributed by Christian Heimes in bpo-11016 [https://bugs.python.org/issue?@action=redirect&bpo=11016].)

The module supports new **`ST_MODE`** flags, **`S_IFDOOR`**, **`S_IFPORT`**, and **`S_IFWHT`**. (Contributed by Christian Hiemes in bpo-11016 [https://bugs.python.org/issue?@action=redirect&bpo=11016].)

struct

New function **`iter_unpack`** and a new **`struct.Struct.iter_unpack()`** method on compiled formats provide streamed unpacking of a buffer containing repeated instances of a given format of data. (Contributed by Antoine Pitrou in bpo-17804 [https://bugs.python.org/issue?@action=redirect&bpo=17804].)

subprocess

`check_output()` now accepts an *input* argument that can be used to provide the contents of `stdin` for the command that is run. (Contributed by Zack Weinberg in bpo-16624 [https://bugs.python.org/issue?@action=redirect&bpo=16624].)

`getoutput()` and **`getstatusoutput()`** now work on Windows. This change was actually inadvertently made in 3.3.4. (Contributed by Tim Golden in bpo-10197 [https://bugs.python.org/issue?@action=redirect&bpo=10197].)

sunau

The **`getparams()`** method now returns a namedtuple rather than a plain tuple. (Contributed by Claudiu Popa in bpo-18901 [https://bugs.python.org/issue?@action=redirect&bpo=18901].)

`sunau.open()` now supports the context management protocol: when used in a **`with`** block, the `close` method of the returned object will be called automatically at the end of the block. (Contributed by Serhiy Storchaka in bpo-18878 [https://bugs.python.org/issue?@action=redirect&bpo=18878].)

`AU_write.setsampwidth()` now supports 24 bit samples, thus adding support for writing 24 sample using the module. (Contributed by Serhiy Storchaka in bpo-19261 [https://bugs.python.org/issue?@action=redirect&bpo=19261].)

The **`writeframesraw()`** and **`writeframes()`** methods now accept any bytes-like object. (Contributed by Serhiy Storchaka in bpo-8311 [https://bugs.python.org/issue?@action=redirect&bpo=8311].)

sys

New function **`sys.getallocatedblocks()`** returns the current number of blocks allocated by the interpreter. (In CPython with the default `--with-pymalloc` setting, this is allocations made through the **`PyObject_Malloc()`** API.) This can be useful for tracking memory leaks, especially if automated via a test suite. (Contributed by Antoine Pitrou in bpo-13390 [https://bugs.python.org/issue?@action=redirect&bpo=13390].)

When the Python interpreter starts in interactive mode, it checks for an **`__interactivehook__`** attribute on the **`sys`** module. If the attribute exists, its value is called with no arguments just before interactive mode is started. The check is made after the **`PYTHONSTARTUP`** file is read, so it can be set there. The **`site`** module sets it to a function that enables tab completion and history saving (in `~/.python-history`) if the platform supports **`readline`**. If you do not want this (new) behavior, you can override it in **`PYTHONSTARTUP`**, **`sitecustomize`**, or **`usercustomize`** by deleting this attribute from **`sys`** (or setting it to some other callable). (Contributed by Éric Araujo and

Antoine Pitrou in bpo-5845 [https://bugs.python.org/issue?@action=redirect&bpo=5845].)

tarfile

The `tarfile` module now supports a simple Command-Line Interface when called as a script directly or via `-m`. This can be used to create and extract tarfile archives. (Contributed by Berker Peksag in bpo-13477 [https://bugs.python.org/issue?@action=redirect&bpo=13477].)

textwrap

The `TextWrapper` class has two new attributes/constructor arguments: `max_lines`, which limits the number of lines in the output, and `placeholder`, which is a string that will appear at the end of the output if it has been truncated because of *max_lines*. Building on these capabilities, a new convenience function `shorten()` collapses all of the whitespace in the input to single spaces and produces a single line of a given *width* that ends with the *placeholder* (by default, `[...]`). (Contributed by Antoine Pitrou and Serhiy Storchaka in bpo-18585 [https://bugs.python.org/issue?@action=redirect&bpo=18585] and bpo-18725 [https://bugs.python.org/issue?@action=redirect&bpo=18725].)

threading

The `Thread` object representing the main thread can be obtained from the new `main_thread()` function. In normal conditions this will be the thread from which the Python interpreter was started. (Contributed by Andrew Svetlov in bpo-18882 [https://bugs.python.org/issue?@action=redirect&bpo=18882].)

traceback

A new **`traceback.clear_frames()`** function takes a traceback object and clears the local variables in all of the frames it references, reducing the amount of memory consumed. (Contributed by Andrew Kuchling in bpo-1565525 [https://bugs.python.org/issue?@action=redirect&bpo=1565525].)

types

A new **`DynamicClassAttribute()`** descriptor provides a way to define an attribute that acts normally when looked up through an instance object, but which is routed to the *class* `__getattr__` when looked up through the class. This allows one to have properties active on a class, and have virtual attributes on the class with the same name (see **`enum`** for an example). (Contributed by Ethan Furman in bpo-19030 [https://bugs.python.org/issue?@action=redirect&bpo=19030].)

urllib

`urllib.request` now supports `data:` URLs via the **`DataHandler`** class. (Contributed by Mathias Panzenböck in bpo-16423 [https://bugs.python.org/issue?@action=redirect&bpo=16423].)

The http method that will be used by a **`Request`** class can now be specified by setting a **`method`** class attribute on the subclass. (Contributed by Jason R Coombs in bpo-18978 [https://bugs.python.org/issue?@action=redirect&bpo=18978].)

`Request` objects are now reusable: if the **`full_url`** or **`data`** attributes are modified, all relevant internal properties are updated. This means, for example, that it is now possible to use the same **`Request`** object in more than one **`OpenerDirector.open()`** call with different *data* arguments, or to modify a **`Request`**‘s `url` rather than recomputing it from scratch. There is also a new **`remove_header()`** method that can be used to remove headers from a **`Request`**. (Contributed by

Alexey Kachayev in bpo-16464 [https://bugs.python.org/issue?@action=redirect&bpo=16464], Daniel Wozniak in bpo-17485 [https://bugs.python.org/issue?@action=redirect&bpo=17485], and Damien Brecht and Senthil Kumaran in bpo-17272 [https://bugs.python.org/issue?@action=redirect&bpo=17272].)

`HTTPError` objects now have a **`headers`** attribute that provides access to the HTTP response headers associated with the error. (Contributed by Berker Peksag in bpo-15701 [https://bugs.python.org/issue?@action=redirect&bpo=15701].)

unittest

The **`TestCase`** class has a new method, **`subTest()`**, that produces a context manager whose **`with`** block becomes a "sub-test". This context manager allows a test method to dynamically generate subtests by, say, calling the `subTest` context manager inside a loop. A single test method can thereby produce an indefinite number of separately identified and separately counted tests, all of which will run even if one or more of them fail. For example:

```
class NumbersTest(unittest.TestCase):
    def test_even(self):
        for i in range(6):
            with self.subTest(i=i):
                self.assertEqual(i % 2, 0)
```

will result in six subtests, each identified in the unittest verbose output with a label consisting of the variable name `i` and a particular value for that variable (`i=0`, `i=1`, etc). See Distinguishing test iterations using subtests for the full version of this example. (Contributed by Antoine Pitrou in bpo-16997 [https://bugs.python.org/issue?@action=redirect&bpo=16997].)

`unittest.main()` now accepts an iterable of test names for *defaultTest*, where previously it only accepted a single test name as a string. (Contributed by Jyrki Pulliainen in bpo-15132 [https://bugs.python.org/issue?@action=redirect&bpo=15132].)

If **`SkipTest`** is raised during test discovery (that is, at the module level in the test file), it is now reported as a skip instead of an error. (Contributed by Zach Ware in bpo-16935 [https://bugs.python.org/issue?@action=redirect&bpo=16935].)

`discover()` now sorts the discovered files to provide consistent test ordering. (Contributed by Martin Melin and Jeff Ramnani in bpo-16709 [https://bugs.python.org/issue?@action=redirect&bpo=16709].)

`TestSuite` now drops references to tests as soon as the test has been run, if the test is successful. On Python interpreters that do garbage collection, this allows the tests to be garbage collected if nothing else is holding a reference to the test. It is possible to override this behavior by creating a **`TestSuite`** subclass that defines a custom `_removeTestAtIndex` method. (Contributed by Tom Wardill, Matt McClure, and Andrew Svetlov in bpo-11798 [https://bugs.python.org/issue?@action=redirect&bpo=11798].)

A new test assertion context-manager, **`assertLogs()`**, will ensure that a given block of code emits a log message using the **`logging`** module. By default the message can come from any logger and have a priority of `INFO` or higher, but both the logger name and an alternative minimum logging level may be specified. The object returned by the context manager can be queried for the **`LogRecord`**s and/or formatted messages that were logged. (Contributed by Antoine Pitrou in bpo-18937 [https://bugs.python.org/issue?@action=redirect&bpo=18937].)

Test discovery now works with namespace packages (Contributed by Claudiu Popa in bpo-17457 [https://bugs.python.org/issue?@action=redirect&bpo=17457].)

`unittest.mock` objects now inspect their specification signatures when matching calls, which means an argument can now be matched by either position or name, instead of only by position. (Contributed by Antoine Pitrou in bpo-17015 [https://bugs.python.org/issue?@action=redirect&bpo=17015].)

`mock_open()` objects now have `readline` and `readlines` methods. (Contributed by Toshio Kuratomi in bpo-17467 [https://bugs.python.org/issue?@action=redirect&bpo=17467].)

venv

`venv` now includes activation scripts for the `csh` and `fish` shells. (Contributed by Andrew Svetlov in bpo-15417 [https://bugs.python.org/issue?@action=redirect&bpo=15417].)

`EnvBuilder` and the **`create()`** convenience function take a new keyword argument *with_pip*, which defaults to `False`, that controls whether or not **`EnvBuilder`** ensures that `pip` is installed in the virtual environment. (Contributed by Nick Coghlan in bpo-19552 [https://bugs.python.org/issue?@action=redirect&bpo=19552] as part of the **PEP 453** [https://peps.python.org/pep-0453/] implementation.)

wave

The **`getparams()`** method now returns a namedtuple rather than a plain tuple. (Contributed by Claudiu Popa in bpo-17487 [https://bugs.python.org/issue?@action=redirect&bpo=17487].)

`wave.open()` now supports the context management protocol. (Contributed by Claudiu Popa in bpo-17616 [https://bugs.python.org/issue?@action=redirect&bpo=17616].)

`wave` can now write output to unseekable files. (Contributed by David Jones, Guilherme Polo, and Serhiy Storchaka in

bpo-5202 [https://bugs.python.org/issue?@action=redirect&bpo=5202].)

The `writeframesraw()` and `writeframes()` methods now accept any bytes-like object. (Contributed by Serhiy Storchaka in bpo-8311 [https://bugs.python.org/issue?@action=redirect&bpo=8311].)

weakref

New `WeakMethod` class simulates weak references to bound methods. (Contributed by Antoine Pitrou in bpo-14631 [https://bugs.python.org/issue?@action=redirect&bpo=14631].)

New `finalize` class makes it possible to register a callback to be invoked when an object is garbage collected, without needing to carefully manage the lifecycle of the weak reference itself. (Contributed by Richard Oudkerk in bpo-15528 [https://bugs.python.org/issue?@action=redirect&bpo=15528].)

The callback, if any, associated with a `ref` is now exposed via the `__callback__` attribute. (Contributed by Mark Dickinson in bpo-17643 [https://bugs.python.org/issue?@action=redirect&bpo=17643].)

xml.etree

A new parser, `XMLPullParser`, allows a non-blocking applications to parse XML documents. An example can be seen at Pull API for non-blocking parsing. (Contributed by Antoine Pitrou in bpo-17741 [https://bugs.python.org/issue?@action=redirect&bpo=17741].)

The `xml.etree.ElementTree` `tostring()` and `tostringlist()` functions, and the `ElementTree` `write()` method, now have a *short_empty_elements* keyword-only parameter providing control over whether elements with no content are written in abbreviated (`<tag />`) or expanded (`<tag></tag>`) form.

(Contributed by Ariel Poliak and Serhiy Storchaka in bpo-14377 [https://bugs.python.org/issue?@action=redirect&bpo=14377].)

zipfile

The **`writepy()`** method of the **`PyZipFile`** class has a new *filterfunc* option that can be used to control which directories and files are added to the archive. For example, this could be used to exclude test files from the archive. (Contributed by Christian Tismer in bpo-19274 [https://bugs.python.org/issue?@action=redirect&bpo=19274].)

The *allowZip64* parameter to **`ZipFile`** and **`PyZipFile`** is now `True` by default. (Contributed by William Mallard in bpo-17201 [https://bugs.python.org/issue?@action=redirect&bpo=17201].)

CPython Implementation Changes

PEP 445: Customization of CPython Memory Allocators

PEP 445 [https://peps.python.org/pep-0445/] adds new C level interfaces to customize memory allocation in the CPython interpreter.

See also

PEP 445 [https://peps.python.org/pep-0445/] – Add new APIs to customize Python memory allocators
PEP written and implemented by Victor Stinner.

PEP 442: Safe Object Finalization

PEP 442 [https://peps.python.org/pep-0442/] removes the current limitations and quirks of object finalization in CPython. With it, objects with `__del__()` methods, as well as generators with `finally` clauses, can be finalized when they are part of a reference cycle.

As part of this change, module globals are no longer forcibly set to `None` during interpreter shutdown in most cases, instead relying on the normal operation of the cyclic garbage collector. This avoids a whole class of interpreter-shutdown-time errors, usually involving `__del__` methods, that have plagued Python since the cyclic GC was first introduced.

See also

PEP 442 [https://peps.python.org/pep-0442/] – Safe object finalization

PEP written and implemented by Antoine Pitrou.

PEP 456: Secure and Interchangeable Hash Algorithm

PEP 456 [https://peps.python.org/pep-0456/] follows up on earlier security fix work done on Python’s hash algorithm to address certain DOS attacks to which public facing APIs backed by dictionary lookups may be subject. (See bpo-14621 [https://bugs.python.org/issue?@action=redirect&bpo=14621] for the start of the current round of improvements.) The PEP unifies CPython’s hash code to make it easier for a packager to substitute a different hash algorithm, and switches Python’s default implementation to a SipHash implementation on platforms that have a 64 bit data type.

Any performance differences in comparison with the older FNV algorithm are trivial.

The PEP adds additional fields to the `sys.hash_info` named tuple to describe the hash algorithm in use by the currently executing binary. Otherwise, the PEP does not alter any existing CPython APIs.

PEP 436: Argument Clinic

“Argument Clinic” (**PEP 436** [https://peps.python.org/pep-0436/]) is now part of the CPython build process and can be used to simplify the process of defining and maintaining accurate signatures for builtins and standard library extension modules implemented in C.

Some standard library extension modules have been converted to use Argument Clinic in Python 3.4, and `pydoc` and `inspect` have been updated accordingly.

It is expected that signature metadata for programmatic introspection will be added to additional callables implemented in C as part of Python 3.4 maintenance releases.

Note

The Argument Clinic PEP is not fully up to date with the state of the implementation. This has been deemed acceptable by the release manager and core development team in this case, as Argument Clinic will not be made available as a public API for third party use in Python 3.4.

See also

PEP 436 [https://peps.python.org/pep-0436/] – The Argument Clinic DSL

PEP written and implemented by Larry Hastings.

Other Build and C API Changes

- The new `PyType_GetSlot()` function has been added to the stable ABI, allowing retrieval of function pointers from named type slots when using the limited API. (Contributed by Martin von Löwis in bpo-17162 [https://bugs.python.org/issue?@action=redirect&bpo=17162].)
- The new **`Py_SetStandardStreamEncoding()`** pre-initialization API allows applications embedding the CPython interpreter to reliably force a particular encoding and error handler for the standard streams. (Contributed by Bastien Montagne and Nick Coghlan in bpo-16129 [https://bugs.python.org/issue?@action=redirect&bpo=16129].)
- Most Python C APIs that don’t mutate string arguments are now correctly marked as accepting `const char *` rather than `char *`. (Contributed by Serhiy Storchaka in bpo-1772673 [https://bugs.python.org/issue?@action=redirect&bpo=1772673].)
- A new shell version of `python-config` can be used even when a python interpreter is not available (for example, in cross compilation scenarios).
- `PyUnicode_FromFormat()` now supports width and precision specifications for `%s`, `%A`, `%U`, `%V`, `%S`, and `%R`. (Contributed by Ysj Ray and Victor Stinner in bpo-7330 [https://bugs.python.org/issue?@action=redirect&bpo=7330].)
- New function `PyStructSequence_InitType2()` supplements the existing `PyStructSequence_InitType()` function. The difference is that it returns `0` on success and `-1` on failure.

- The CPython source can now be compiled using the address sanity checking features of recent versions of GCC and clang: the false alarms in the small object allocator have been silenced. (Contributed by Dhiru Kholia in bpo-18596 [https://bugs.python.org/issue?@action=redirect&bpo=18596].)
- The Windows build now uses Address Space Layout Randomization [https://en.wikipedia.org/wiki/Address_space_layout_randomization] and Data Execution Prevention [https://en.wikipedia.org/wiki/Data_Execution_Prevention]. (Contributed by Christian Heimes in bpo-16632 [https://bugs.python.org/issue?@action=redirect&bpo=16632].)
- New function **`PyObject_LengthHint()`** is the C API equivalent of **`operator.length_hint()`**. (Contributed by Armin Ronacher in bpo-16148 [https://bugs.python.org/issue?@action=redirect&bpo=16148].)

Other Improvements

- The python command has a new option, `-I`, which causes it to run in “isolated mode”, which means that **`sys.path`** contains neither the script’s directory nor the user’s `site-packages` directory, and all **`PYTHON*`** environment variables are ignored (it implies both `-s` and `-E`). Other restrictions may also be applied in the future, with the goal being to isolate the execution of a script from the user’s environment. This is appropriate, for example, when Python is used to run a system script. On most POSIX systems it can and should be used in the `#!` line of system scripts. (Contributed by Christian Heimes in bpo-16499 [https://bugs.python.org/issue?@action=redirect&bpo=16499].)
- Tab-completion is now enabled by default in the interactive interpreter on systems that support **`readline`**. History is also enabled by default, and is written to (and

read from) the file `~/.python-history`. (Contributed by Antoine Pitrou and Éric Araujo in bpo-5845 [https://bugs.python.org/issue?@action=redirect&bpo=5845].)

- Invoking the Python interpreter with `--version` now outputs the version to standard output instead of standard error (bpo-18338 [https://bugs.python.org/issue?@action=redirect&bpo=18338]). Similar changes were made to **`argparse`** (bpo-18920 [https://bugs.python.org/issue?@action=redirect&bpo=18920]) and other modules that have script-like invocation capabilities (bpo-18922 [https://bugs.python.org/issue?@action=redirect&bpo=18922]).
- The CPython Windows installer now adds `.py` to the **PATHEXT** variable when extensions are registered, allowing users to run a python script at the windows command prompt by just typing its name without the `.py` extension. (Contributed by Paul Moore in bpo-18569 [https://bugs.python.org/issue?@action=redirect&bpo=18569].)
- A new `make` target coverage-report [https://devguide.python.org/coverage/#measuring-coverage-of-c-code-with-gcov-and-lcov] will build python, run the test suite, and generate an HTML coverage report for the C codebase using `gcov` and lcov [https://github.com/linux-test-project/lcov].
- The `-R` option to the python regression test suite now also checks for memory allocation leaks, using **`sys.getallocatedblocks()`**. (Contributed by Antoine Pitrou in bpo-13390 [https://bugs.python.org/issue?@action=redirect&bpo=13390].)
- `python -m` now works with namespace packages.
- The **`stat`** module is now implemented in C, which means it gets the values for its constants from the C header files, instead of having the values hard-coded in the python module as was previously the case.
- Loading multiple python modules from a single OS module (`.so`, `.dll`) now works correctly (previously it silently returned the first python module in the file).

(Contributed by Václav Šmilauer in bpo-16421 [https://bugs.python.org/issue?@action=redirect&bpo=16421].)

- A new opcode, **`LOAD_CLASSDEREF`**, has been added to fix a bug in the loading of free variables in class bodies that could be triggered by certain uses of __prepare__. (Contributed by Benjamin Peterson in bpo-17853 [https://bugs.python.org/issue?@action=redirect&bpo=17853].)
- A number of MemoryError-related crashes were identified and fixed by Victor Stinner using his **PEP 445** [https://peps.python.org/pep-0445/]-based `pyfailmalloc` tool (bpo-18408 [https://bugs.python.org/issue?@action=redirect&bpo=18408], bpo-18520 [https://bugs.python.org/issue?@action=redirect&bpo=18520]).
- The `pyvenv` command now accepts a `--copies` option to use copies rather than symlinks even on systems where symlinks are the default. (Contributed by Vinay Sajip in bpo-18807 [https://bugs.python.org/issue?@action=redirect&bpo=18807].)
- The `pyvenv` command also accepts a `--without-pip` option to suppress the otherwise-automatic bootstrapping of pip into the virtual environment. (Contributed by Nick Coghlan in bpo-19552 [https://bugs.python.org/issue?@action=redirect&bpo=19552] as part of the **PEP 453** [https://peps.python.org/pep-0453/] implementation.)
- The encoding name is now optional in the value set for the **`PYTHONIOENCODING`** environment variable. This makes it possible to set just the error handler, without changing the default encoding. (Contributed by Serhiy Storchaka in bpo-18818 [https://bugs.python.org/issue?@action=redirect&bpo=18818].)
- The **`bz2`**, **`lzma`**, and **`gzip`** module `open` functions now support `x` (exclusive creation) mode. (Contributed by Tim Heaney and Vajrasky Kok in bpo-19201 [https://bugs.python.org/issue?@action=redirect&bpo=19201], bpo-19222 [https://bugs.python.org/issue?@action=redirect&bpo=19222],

and bpo-19223 [https://bugs.python.org/issue?@action=redirect&bpo=19223].)

Significant Optimizations

- The UTF-32 decoder is now 3x to 4x faster. (Contributed by Serhiy Storchaka in bpo-14625 [https://bugs.python.org/issue?@action=redirect&bpo=14625].)
- The cost of hash collisions for sets is now reduced. Each hash table probe now checks a series of consecutive, adjacent key/hash pairs before continuing to make random probes through the hash table. This exploits cache locality to make collision resolution less expensive. The collision resolution scheme can be described as a hybrid of linear probing and open addressing. The number of additional linear probes defaults to nine. This can be changed at compile-time by defining LINEAR_PROBES to be any value. Set LINEAR_PROBES=0 to turn-off linear probing entirely. (Contributed by Raymond Hettinger in bpo-18771 [https://bugs.python.org/issue?@action=redirect&bpo=18771].)
- The interpreter starts about 30% faster. A couple of measures lead to the speedup. The interpreter loads fewer modules on startup, e.g. the `re`, `collections` and `locale` modules and their dependencies are no longer imported by default. The marshal module has been improved to load compiled Python code faster. (Contributed by Antoine Pitrou, Christian Heimes and Victor Stinner in bpo-19219 [https://bugs.python.org/issue?@action=redirect&bpo=19219], bpo-19218 [https://bugs.python.org/issue?@action=redirect&bpo=19218], bpo-19209 [https://bugs.python.org/issue?@action=redirect&bpo=19209], bpo-19205 [https://bugs.python.org/issue?@action=redirect&bpo=19205] and bpo-9548 [https://bugs.python.org/issue?@action=redirect&bpo=9548].)

- `bz2.BZ2File` is now as fast or faster than the Python2 version for most cases. `lzma.LZMAFile` has also been optimized. (Contributed by Serhiy Storchaka and Nadeem Vawda in bpo-16034 [https://bugs.python.org/issue?@action=redirect&bpo=16034].)
- `random.getrandbits()` is 20%-40% faster for small integers (the most common use case). (Contributed by Serhiy Storchaka in bpo-16674 [https://bugs.python.org/issue?@action=redirect&bpo=16674].)
- By taking advantage of the new storage format for strings, pickling of strings is now significantly faster. (Contributed by Victor Stinner and Antoine Pitrou in bpo-15596 [https://bugs.python.org/issue?@action=redirect&bpo=15596].)
- A performance issue in `io.FileIO.readall()` has been solved. This particularly affects Windows, and significantly speeds up the case of piping significant amounts of data through `subprocess`. (Contributed by Richard Oudkerk in bpo-15758 [https://bugs.python.org/issue?@action=redirect&bpo=15758].)
- `html.escape()` is now 10x faster. (Contributed by Matt Bryant in bpo-18020 [https://bugs.python.org/issue?@action=redirect&bpo=18020].)
- On Windows, the native `VirtualAlloc` is now used instead of the CRT `malloc` in `obmalloc`. Artificial benchmarks show about a 3% memory savings.
- `os.urandom()` now uses a lazily opened persistent file descriptor so as to avoid using many file descriptors when run in parallel from multiple threads. (Contributed by Antoine Pitrou in bpo-18756 [https://bugs.python.org/issue?@action=redirect&bpo=18756].)

Deprecated

This section covers various APIs and other features that have been deprecated in Python 3.4, and will be removed in

Python 3.5 or later. In most (but not all) cases, using the deprecated APIs will produce a **`DeprecationWarning`** when the interpreter is run with deprecation warnings enabled (for example, by using `-Wd`).

Deprecations in the Python API

- As mentioned in PEP 451: A ModuleSpec Type for the Import System, a number of **`importlib`** methods and functions are deprecated: **`importlib.find_loader()`** is replaced by **`importlib.util.find_spec()`**; **`importlib.machinery.PathFinder.find_module()`** is replaced by **`importlib.machinery.PathFinder.find_spec()`**; **`importlib.abc.MetaPathFinder.find_module()`** is replaced by **`importlib.abc.MetaPathFinder.find_spec()`**; **`importlib.abc.PathEntryFinder.find_loader()`** and **`find_module()`** are replaced by **`importlib.abc.PathEntryFinder.find_spec()`**; all of the *xxx*`Loader` ABC `load_module` methods (**`importlib.abc.Loader.load_module()`**, **`importlib.abc.InspectLoader.load_module()`**, **`importlib.abc.FileLoader.load_module()`**, **`importlib.abc.SourceLoader.load_module()`**) should no longer be implemented, instead loaders should implement an `exec_module` method (**`importlib.abc.Loader.exec_module()`**, **`importlib.abc.InspectLoader.exec_module()`** **`importlib.abc.SourceLoader.exec_module()`**) and let the import system take care of the rest; and **`importlib.abc.Loader.module_repr()`**, **`importlib.util.module_for_loader()`**, **`importlib.util.set_loader()`**, and **`importlib.util.set_package()`** are no longer needed because their functions are now handled automatically by the import system.
- The **`imp`** module is pending deprecation. To keep compatibility with Python 2/3 code bases, the module’s

removal is currently not scheduled.

- The **`formatter`** module is pending deprecation and is slated for removal in Python 3.6.
- `MD5` as the default *digestmod* for the `hmac.new()` function is deprecated. Python 3.6 will require an explicit digest name or constructor as *digestmod* argument.
- The internal `Netrc` class in the `ftplib` module has been documented as deprecated in its docstring for quite some time. It now emits a `DeprecationWarning` and will be removed completely in Python 3.5.
- The undocumented *endtime* argument to `subprocess.Popen.wait()` should not have been exposed and is hopefully not in use; it is deprecated and will mostly likely be removed in Python 3.5.
- The *strict* argument of `HTMLParser` is deprecated.
- The `plistlib` **`readPlist()`**, **`writePlist()`**, **`readPlistFromBytes()`**, and **`writePlistToBytes()`** functions are deprecated in favor of the corresponding new functions `load()`, `dump()`, `loads()`, and `dumps()`. **`Data()`** is deprecated in favor of just using the `bytes` constructor.
- The `sysconfig` key `SO` is deprecated, it has been replaced by `EXT_SUFFIX`.
- The `U` mode accepted by various `open` functions is deprecated. In Python3 it does not do anything useful, and should be replaced by appropriate uses of `io.TextIOWrapper` (if needed) and its *newline* argument.
- The *parser* argument of `xml.etree.ElementTree.iterparse()` has been deprecated, as has the *html* argument of `XMLParser()`. To prepare for the removal of the latter, all arguments to `XMLParser` should be passed by keyword.

Deprecated Features

- Running IDLE — Python editor and shell with the `-n` flag (no subprocess) is deprecated. However, the feature will

not be removed until bpo-18823 [https://bugs.python.org/issue?@action=redirect&bpo=18823] is resolved.
- The site module adding a “site-python” directory to sys.path, if it exists, is deprecated (bpo-19375 [https://bugs.python.org/issue?@action=redirect&bpo=19375]).

Removed

Operating Systems No Longer Supported

Support for the following operating systems has been removed from the source and build tools:

- OS/2 (bpo-16135 [https://bugs.python.org/issue?@action=redirect&bpo=16135]).
- Windows 2000 (changeset e52df05b496a).
- Windows systems where `COMSPEC` points to `command.com` (bpo-14470 [https://bugs.python.org/issue?@action=redirect&bpo=14470]).
- VMS (bpo-16136 [https://bugs.python.org/issue?@action=redirect&bpo=16136]).

API and Feature Removals

The following obsolete and previously deprecated APIs and features have been removed:

- The unmaintained `Misc/TextMate` and `Misc/vim` directories have been removed (see the devguide [https://devguide.python.org] for suggestions on what to use instead).
- The `SO` makefile macro is removed (it was replaced by the `SHLIB_SUFFIX` and `EXT_SUFFIX` macros) (bpo-16754 [https://bugs.python.org/issue?@action=redirect&bpo=16754]).

- The `PyThreadState.tick_counter` field has been removed; its value has been meaningless since Python 3.2, when the “new GIL” was introduced (bpo-19199 [https://bugs.python.org/issue?@action=redirect&bpo=19199]).
- `PyLoader` and `PyPycLoader` have been removed from **`importlib`**. (Contributed by Taras Lyapun in bpo-15641 [https://bugs.python.org/issue?@action=redirect&bpo=15641].)
- The *strict* argument to **`HTTPConnection`** and **`HTTPSConnection`** has been removed. HTTP 0.9-style “Simple Responses” are no longer supported.
- The deprecated **`urllib.request.Request`** getter and setter methods `add_data`, `has_data`, `get_data`, `get_type`, `get_host`, `get_selector`, `set_proxy`, `get_origin_req_host`, and `is_unverifiable` have been removed (use direct attribute access instead).
- Support for loading the deprecated `TYPE_INT64` has been removed from **`marshal`**. (Contributed by Dan Riti in bpo-15480 [https://bugs.python.org/issue?@action=redirect&bpo=15480].)
- **`inspect.Signature`**: positional-only parameters are now required to have a valid name.
- **`object.__format__()`** no longer accepts non-empty format strings, it now raises a **`TypeError`** instead. Using a non-empty string has been deprecated since Python 3.2. This change has been made to prevent a situation where previously working (but incorrect) code would start failing if an object gained a __format__ method, which means that your code may now raise a **`TypeError`** if you are using an `'s'` format code with objects that do not have a __format__ method that handles it. See bpo-7994 [https://bugs.python.org/issue?@action=redirect&bpo=7994] for background.
- **`difflib.SequenceMatcher.isbjunk()`** and **`difflib.SequenceMatcher.isbpopular()`** were deprecated in 3.2, and have now been removed: use `x in sm.bjunk` and `x in sm.bpopular`, where *sm* is a **`SequenceMatcher`** object

(bpo-13248 [https://bugs.python.org/issue?@action=redirect&bpo=13248]).

Code Cleanups

- The unused and undocumented internal `Scanner` class has been removed from the **`pydoc`** module.
- The private and effectively unused `_gestalt` module has been removed, along with the private **`platform`** functions `_mac_ver_lookup`, `_mac_ver_gstalt`, and `_bcd2str`, which would only have ever been called on badly broken OSX systems (see bpo-18393 [https://bugs.python.org/issue?@action=redirect&bpo=18393]).
- The hardcoded copies of certain **`stat`** constants that were included in the **`tarfile`** module namespace have been removed.

Porting to Python 3.4

This section lists previously described changes and other bugfixes that may require changes to your code.

Changes in 'python' Command Behavior

- In a posix shell, setting the **`PATH`** environment variable to an empty value is equivalent to not setting it at all. However, setting **`PYTHONPATH`** to an empty value was *not* equivalent to not setting it at all: setting **`PYTHONPATH`** to an empty value was equivalent to setting it to `.`, which leads to confusion when reasoning by analogy to how **`PATH`** works. The behavior now conforms to the posix convention for **`PATH`**.
- The [X refs, Y blocks] output of a debug (`--with-pydebug`) build of the CPython interpreter is now off by default. It can be re-enabled using the `-X showrefcount` option.

(Contributed by Ezio Melotti in bpo-17323 [https://bugs.python.org/issue?@action=redirect&bpo=17323].)

- The python command and most stdlib scripts (as well as **`argparse`**) now output `--version` information to `stdout` instead of `stderr` (for issue list see Other Improvements above).

Changes in the Python API

- The ABCs defined in **`importlib.abc`** now either raise the appropriate exception or return a default value instead of raising **`NotImplementedError`** blindly. This will only affect code calling **`super()`** and falling through all the way to the ABCs. For compatibility, catch both **`NotImplementedError`** or the appropriate exception as needed.
- The module type now initializes the **`__package__`** and **`__loader__`** attributes to `None` by default. To determine if these attributes were set in a backwards-compatible fashion, use e.g. `getattr(module, '__loader__', None) is not None`. (bpo-17115 [https://bugs.python.org/issue?@action=redirect&bpo=17115].)
- **`importlib.util.module_for_loader()`** now sets `__loader__` and `__package__` unconditionally to properly support reloading. If this is not desired then you will need to set these attributes manually. You can use **`importlib.util.module_to_load()`** for module management.
- Import now resets relevant attributes (e.g. `__name__`, `__loader__`, `__package__`, `__file__`, `__cached__`) unconditionally when reloading. Note that this restores a pre-3.3 behavior in that it means a module is re-found when re-loaded (bpo-19413 [https://bugs.python.org/issue?@action=redirect&bpo=19413]).
- Frozen packages no longer set `__path__` to a list containing the package name, they now set it to an empty list. The previous behavior could cause the

import system to do the wrong thing on submodule imports if there was also a directory with the same name as the frozen package. The correct way to determine if a module is a package or not is to use `hasattr(module, '__path__')` (bpo-18065 [https://bugs.python.org/issue?@action=redirect&bpo=18065]).

- Frozen modules no longer define a `__file__` attribute. It's semantically incorrect for frozen modules to set the attribute as they are not loaded from any explicit location. If you must know that a module comes from frozen code then you can see if the module's `__spec__.location` is set to `'frozen'`, check if the loader is a subclass of `importlib.machinery.FrozenImporter`, or if Python 2 compatibility is necessary you can use **`imp.is_frozen()`**.
- `py_compile.compile()` now raises `FileExistsError` if the file path it would write to is a symlink or a non-regular file. This is to act as a warning that import will overwrite those files with a regular file regardless of what type of file path they were originally.
- `importlib.abc.SourceLoader.get_source()` no longer raises `ImportError` when the source code being loaded triggers a `SyntaxError` or `UnicodeDecodeError`. As `ImportError` is meant to be raised only when source code cannot be found but it should, it was felt to be over-reaching/overloading of that meaning when the source code is found but improperly structured. If you were catching ImportError before and wish to continue to ignore syntax or decoding issues, catch all three exceptions now.
- `functools.update_wrapper()` and `functools.wraps()` now correctly set the `__wrapped__` attribute to the function being wrapped, even if that function also had its `__wrapped__` attribute set. This means `__wrapped__` attributes now correctly link a stack of decorated functions rather than every `__wrapped__` attribute in the

chain referring to the innermost function. Introspection libraries that assumed the previous behaviour was intentional can use `inspect.unwrap()` to access the first function in the chain that has no `__wrapped__` attribute.

- `inspect.getfullargspec()` has been reimplemented on top of `inspect.signature()` and hence handles a much wider variety of callable objects than it did in the past. It is expected that additional builtin and extension module callables will gain signature metadata over the course of the Python 3.4 series. Code that assumes that `inspect.getfullargspec()` will fail on non-Python callables may need to be adjusted accordingly.
- `importlib.machinery.PathFinder` now passes on the current working directory to objects in `sys.path_hooks` for the empty string. This results in `sys.path_importer_cache` never containing `''`, thus iterating through `sys.path_importer_cache` based on `sys.path` will not find all keys. A module's `__file__` when imported in the current working directory will also now have an absolute path, including when using `-m` with the interpreter (except for `__main__.__file__` when a script has been executed directly using a relative path) (Contributed by Brett Cannon in bpo-18416 [https://bugs.python.org/issue?@action=redirect&bpo=18416]). is specified on the command-line) (bpo-18416 [https://bugs.python.org/issue?@action=redirect&bpo=18416]).
- The removal of the *strict* argument to `HTTPConnection` and `HTTPSConnection` changes the meaning of the remaining arguments if you are specifying them positionally rather than by keyword. If you've been paying attention to deprecation warnings your code should already be specifying any additional arguments via keywords.
- Strings between `from __future__ import ...` statements now *always* raise a `SyntaxError`. Previously if there was no leading docstring, an interstitial string would sometimes be ignored. This brings CPython into

compliance with the language spec; Jython and PyPy already were. (bpo-17434 [https://bugs.python.org/issue?@action=redirect&bpo=17434]).

- `ssl.SSLSocket.getpeercert()` and `ssl.SSLSocket.do_handshake()` now raise an `OSError` with `ENOTCONN` when the `SSLSocket` is not connected, instead of the previous behavior of raising an `AttributeError`. In addition, `getpeercert()` will raise a `ValueError` if the handshake has not yet been done.
- `base64.b32decode()` now raises a `binascii.Error` when the input string contains non-b32-alphabet characters, instead of a `TypeError`. This particular `TypeError` was missed when the other `TypeError`s were converted. (Contributed by Serhiy Storchaka in bpo-18011 [https://bugs.python.org/issue?@action=redirect&bpo=18011].) Note: this change was also inadvertently applied in Python 3.3.3.
- The **`file`** attribute is now automatically closed when the creating **`cgi.FieldStorage`** instance is garbage collected. If you were pulling the file object out separately from the **`cgi.FieldStorage`** instance and not keeping the instance alive, then you should either store the entire **`cgi.FieldStorage`** instance or read the contents of the file before the **`cgi.FieldStorage`** instance is garbage collected.
- Calling `read` or `write` on a closed SSL socket now raises an informative `ValueError` rather than the previous more mysterious `AttributeError` (bpo-9177 [https://bugs.python.org/issue?@action=redirect&bpo=9177]).
- `slice.indices()` no longer produces an `OverflowError` for huge values. As a consequence of this fix, `slice.indices()` now raises a `ValueError` if given a negative length; previously it returned nonsense values (bpo-14794 [https://bugs.python.org/issue?@action=redirect&bpo=14794]).
- The `complex` constructor, unlike the `cmath` functions, was incorrectly accepting `float` values if an object's

`__complex__` special method returned one. This now raises a **`TypeError`**. (bpo-16290 [https://bugs.python.org/issue?@action=redirect&bpo=16290].)

- The **`int`** constructor in 3.2 and 3.3 erroneously accepts **`float`** values for the *base* parameter. It is unlikely anyone was doing this, but if so, it will now raise a **`TypeError`** (bpo-16772 [https://bugs.python.org/issue?@action=redirect&bpo=16772]).
- Defaults for keyword-only arguments are now evaluated *after* defaults for regular keyword arguments, instead of before. Hopefully no one wrote any code that depends on the previous buggy behavior (bpo-16967 [https://bugs.python.org/issue?@action=redirect&bpo=16967]).
- Stale thread states are now cleared after **`fork()`**. This may cause some system resources to be released that previously were incorrectly kept perpetually alive (for example, database connections kept in thread-local storage). (bpo-17094 [https://bugs.python.org/issue?@action=redirect&bpo=17094].)
- Parameter names in `__annotations__` dicts are now mangled properly, similarly to **`__kwdefaults__`**. (Contributed by Yury Selivanov in bpo-20625 [https://bugs.python.org/issue?@action=redirect&bpo=20625].)
- **`hashlib.hash.name`** now always returns the identifier in lower case. Previously some builtin hashes had uppercase names, but now that it is a formal public interface the naming has been made consistent (bpo-18532 [https://bugs.python.org/issue?@action=redirect&bpo=18532]).
- Because **`unittest.TestSuite`** now drops references to tests after they are run, test harnesses that reuse a **`TestSuite`** to re-run a set of tests may fail. Test suites should not be re-used in this fashion since it means state is retained between test runs, breaking the test isolation that **`unittest`** is designed to provide. However, if the lack of isolation is considered acceptable, the old behavior can be restored by creating a **`TestSuite`** subclass that

defines a `_removeTestAtIndex` method that does nothing (see **`TestSuite.__iter__()`**) (bpo-11798 [https://bugs.python.org/issue?@action=redirect&bpo=11798]).

- **`unittest`** now uses **`argparse`** for command line parsing. There are certain invalid command forms that used to work that are no longer allowed; in theory this should not cause backward compatibility issues since the disallowed command forms didn't make any sense and are unlikely to be in use.
- The **`re.split()`**, **`re.findall()`**, and **`re.sub()`** functions, and the **`group()`** and **`groups()`** methods of `match` objects now always return a *bytes* object when the string to be matched is a bytes-like object. Previously the return type matched the input type, so if your code was depending on the return value being, say, a `bytearray`, you will need to change your code.
- **`audioop`** functions now raise an error immediately if passed string input, instead of failing randomly later on (bpo-16685 [https://bugs.python.org/issue?@action=redirect&bpo=16685]).
- The new *convert_charrefs* argument to **`HTMLParser`** currently defaults to `False` for backward compatibility, but will eventually be changed to default to `True`. It is recommended that you add this keyword, with the appropriate value, to any **`HTMLParser`** calls in your code (bpo-13633 [https://bugs.python.org/issue?@action=redirect&bpo=13633]).
- Since the *digestmod* argument to the **`hmac.new()`** function will in the future have no default, all calls to **`hmac.new()`** should be changed to explicitly specify a *digestmod* (bpo-17276 [https://bugs.python.org/issue?@action=redirect&bpo=17276]).
- Calling **`sysconfig.get_config_var()`** with the `SO` key, or looking `SO` up in the results of a call to **`sysconfig.get_config_vars()`** is deprecated. This key should be replaced by `EXT_SUFFIX` or `SHLIB_SUFFIX`, depending on

the context (bpo-19555 [https://bugs.python.org/issue?@action=redirect&bpo=19555]).

- Any calls to `open` functions that specify `U` should be modified. `U` is ineffective in Python3 and will eventually raise an error if used. Depending on the function, the equivalent of its old Python2 behavior can be achieved using either a *newline* argument, or if necessary by wrapping the stream in **`TextIOWrapper`** to use its *newline* argument (bpo-15204 [https://bugs.python.org/issue?@action=redirect&bpo=15204]).
- If you use `pyvenv` in a script and desire that pip *not* be installed, you must add `--without-pip` to your command invocation.
- The default behavior of **`json.dump()`** and **`json.dumps()`** when an indent is specified has changed: it no longer produces trailing spaces after the item separating commas at the ends of lines. This will matter only if you have tests that are doing white-space-sensitive comparisons of such output (bpo-16333 [https://bugs.python.org/issue?@action=redirect&bpo=16333]).
- **`doctest`** now looks for doctests in extension module `__doc__` strings, so if your doctest test discovery includes extension modules that have things that look like doctests in them you may see test failures you’ve never seen before when running your tests (bpo-3158 [https://bugs.python.org/issue?@action=redirect&bpo=3158]).
- The **`collections.abc`** module has been slightly refactored as part of the Python startup improvements. As a consequence of this, it is no longer the case that importing **`collections`** automatically imports **`collections.abc`**. If your program depended on the (undocumented) implicit import, you will need to add an explicit `import collections.abc` (bpo-20784 [https://bugs.python.org/issue?@action=redirect&bpo=20784]).

Changes in the C API

- `PyEval_EvalFrameEx()`, `PyObject_Repr()`, and `PyObject_Str()`, along with some other internal C APIs, now include a debugging assertion that ensures they are not used in situations where they may silently discard a currently active exception. In cases where discarding the active exception is expected and desired (for example, because it has already been saved locally with `PyErr_Fetch()` or is being deliberately replaced with a different exception), an explicit `PyErr_Clear()` call will be needed to avoid triggering the assertion when invoking these operations (directly or indirectly) and running against a version of Python that is compiled with assertions enabled.
- `PyErr_SetImportError()` now sets `TypeError` when its **msg** argument is not set. Previously only `NULL` was returned with no exception set.
- The result of the `PyOS_ReadlineFunctionPointer` callback must now be a string allocated by `PyMem_RawMalloc()` or `PyMem_RawRealloc()`, or `NULL` if an error occurred, instead of a string allocated by `PyMem_Malloc()` or `PyMem_Realloc()` (bpo-16742 [https://bugs.python.org/issue?@action=redirect&bpo=16742])
- `PyThread_set_key_value()` now always set the value. In Python 3.3, the function did nothing if the key already exists (if the current value is a non-`NULL` pointer).
- The `f_tstate` (thread state) field of the `PyFrameObject` structure has been removed to fix a bug: see bpo-14432 [https://bugs.python.org/issue?@action=redirect&bpo=14432] for the rationale.

Changed in 3.4.3

PEP 476: Enabling certificate verification by default for stdlib http clients

`http.client` and modules which use it, such as `urllib.request` and `xmlrpc.client`, will now verify that the server presents a certificate which is signed by a CA in the platform trust store and whose hostname matches the hostname being requested by default, significantly improving security for many applications.

For applications which require the old previous behavior, they can pass an alternate context:

```
import urllib.request
import ssl

# This disables all verification
context = ssl._create_unverified_context()

# This allows using a specific certificate for the host, which doesn't need
# to be in the trust store
context = ssl.create_default_context(cafile="/path/to/file.crt")

urllib.request.urlopen("https://invalid-cert", context=context)
```

What’s New In Python 3.3

This article explains the new features in Python 3.3, compared to 3.2. Python 3.3 was released on September 29, 2012. For full details, see the changelog [https://docs.python.org/3.3/whatsnew/changelog.html].

See also

PEP 398 [https://peps.python.org/pep-0398/] - Python 3.3 Release Schedule

Summary – Release highlights

New syntax features:

- New `yield from` expression for generator delegation.
- The `u'unicode'` syntax is accepted again for **`str`** objects.

New library modules:

- **`faulthandler`** (helps debugging low-level crashes)
- **`ipaddress`** (high-level objects representing IP addresses and masks)
- **`lzma`** (compress data using the XZ / LZMA algorithm)
- **`unittest.mock`** (replace parts of your system under test with mock objects)
- **`venv`** (Python virtual environments, as in the popular `virtualenv` package)

New built-in features:

- Reworked I/O exception hierarchy.

Implementation improvements:

- Rewritten import machinery based on **`importlib`**.
- More compact unicode strings.
- More compact attribute dictionaries.

Significantly Improved Library Modules:

- C Accelerator for the decimal module.
- Better unicode handling in the email module (provisional).

Security improvements:

- Hash randomization is switched on by default.

Please read on for a comprehensive list of user-facing changes.

PEP 405: Virtual Environments

Virtual environments help create separate Python setups while sharing a system-wide base install, for ease of maintenance. Virtual environments have their own set of private site packages (i.e. locally installed libraries), and are optionally segregated from the system-wide site packages. Their concept and implementation are inspired by the popular `virtualenv` third-party package, but benefit from tighter integration with the interpreter core.

This PEP adds the **`venv`** module for programmatic access, and the `pyvenv` script for command-line access and administration. The Python interpreter checks for a `pyvenv.cfg`, file whose existence signals the base of a virtual environment’s directory tree.

See also

[PEP 405](https://peps.python.org/pep-0405/) [https://peps.python.org/pep-0405/] - Python Virtual Environments
PEP written by Carl Meyer; implementation by Carl Meyer and Vinay Sajip

PEP 420: Implicit Namespace Packages

Native support for package directories that don’t require `__init__.py` marker files and can automatically span multiple path segments (inspired by various third party approaches to namespace packages, as described in **PEP 420** [https://peps.python.org/pep-0420/])

See also

PEP 420 [https://peps.python.org/pep-0420/] - Implicit Namespace Packages
PEP written by Eric V. Smith; implementation by Eric V. Smith and Barry Warsaw

PEP 3118: New memoryview implementation and buffer protocol documentation

The implementation of **PEP 3118** [https://peps.python.org/pep-3118/] has been significantly improved.

The new memoryview implementation comprehensively fixes all ownership and lifetime issues of dynamically allocated fields in the Py_buffer struct that led to multiple crash reports. Additionally, several functions that crashed or returned incorrect results for non-contiguous or multi-dimensional input have been fixed.

The memoryview object now has a PEP-3118 compliant getbufferproc() that checks the consumer’s request type. Many new features have been added, most of them work in full generality for non-contiguous arrays and arrays with suboffsets.

The documentation has been updated, clearly spelling out responsibilities for both exporters and consumers. Buffer request flags are grouped into basic and compound flags. The memory layout of non-contiguous and multi-dimensional NumPy-style arrays is explained.

Features

- All native single character format specifiers in struct module syntax (optionally prefixed with ‘@’) are now supported.
- With some restrictions, the cast() method allows changing of format and shape of C-contiguous arrays.
- Multi-dimensional list representations are supported for any array type.
- Multi-dimensional comparisons are supported for any array type.
- One-dimensional memoryviews of hashable (read-only) types with formats B, b or c are now hashable. (Contributed by Antoine Pitrou in bpo-13411 [https://bugs.python.org/issue?@action=redirect&bpo=13411].)
- Arbitrary slicing of any 1-D arrays type is supported. For example, it is now possible to reverse a memoryview in

O(1) by using a negative step.

API changes

- The maximum number of dimensions is officially limited to 64.
- The representation of empty shape, strides and suboffsets is now an empty tuple instead of `None`.
- Accessing a memoryview element with format 'B' (unsigned bytes) now returns an integer (in accordance with the struct module syntax). For returning a bytes object the view must be cast to 'c' first.
- memoryview comparisons now use the logical structure of the operands and compare all array elements by value. All format strings in struct module syntax are supported. Views with unrecognised format strings are still permitted, but will always compare as unequal, regardless of view contents.
- For further changes see Build and C API Changes and Porting C code.

(Contributed by Stefan Krah in bpo-10181 [https://bugs.python.org/issue?@action=redirect&bpo=10181].)

See also

PEP 3118 [https://peps.python.org/pep-3118/] - Revising the Buffer Protocol

PEP 393: Flexible String Representation

The Unicode string type is changed to support multiple internal representations, depending on the character with the largest Unicode ordinal (1, 2, or 4 bytes) in the represented string. This allows a space-efficient representation in common cases, but gives access to full UCS-4 on all systems. For compatibility with existing APIs, several representations may exist in parallel; over time, this compatibility should be phased out.

On the Python side, there should be no downside to this change.

On the C API side, **PEP 393** [https://peps.python.org/pep-0393/] is fully backward compatible. The legacy API should remain available at least five years. Applications using the legacy API will not fully benefit of the memory reduction, or - worse - may use a bit more memory, because Python may have to maintain two versions of each string (in the legacy format and in the new efficient storage).

Functionality

Changes introduced by **PEP 393** [https://peps.python.org/pep-0393/] are the following:

- Python now always supports the full range of Unicode code points, including non-BMP ones (i.e. from `U+0000` to `U+10FFFF`). The distinction between narrow and wide builds no longer exists and Python now behaves like a wide build, even under Windows.
- With the death of narrow builds, the problems specific to narrow builds have also been fixed, for example:
    - `len()` now always returns 1 for non-BMP characters, so `len('\U0010FFFF') == 1`;
    - surrogate pairs are not recombined in string literals, so `'\uDBFF\uDFFF' != '\U0010FFFF'`;
    - indexing or slicing non-BMP characters returns the expected value, so `'\U0010FFFF'[0]` now returns

    `'\U0010FFFF'` and not `'\uDBFF'`;
    - all other functions in the standard library now correctly handle non-BMP code points.
- The value of **`sys.maxunicode`** is now always `1114111` (`0x10FFFF` in hexadecimal). The **`PyUnicode_GetMax()`** function still returns either `0xFFFF` or `0x10FFFF` for backward compatibility, and it should not be used with the new Unicode API (see bpo-13054 [https://bugs.python.org/issue?@action=redirect&bpo=13054]).
- The `./configure` flag `--with-wide-unicode` has been removed.

Performance and resource usage

The storage of Unicode strings now depends on the highest code point in the string:

- pure ASCII and Latin1 strings (`U+0000-U+00FF`) use 1 byte per code point;
- BMP strings (`U+0000-U+FFFF`) use 2 bytes per code point;
- non-BMP strings (`U+10000-U+10FFFF`) use 4 bytes per code point.

The net effect is that for most applications, memory usage of string storage should decrease significantly - especially compared to former wide unicode builds - as, in many cases, strings will be pure ASCII even in international contexts (because many strings store non-human language data, such as XML fragments, HTTP headers, JSON-encoded data, etc.). We also hope that it will, for the same reasons, increase CPU cache efficiency on non-trivial applications. The memory usage of Python 3.3 is two to three times smaller than Python 3.2, and a little bit better than Python 2.7, on a Django benchmark (see the PEP for details).

See also

PEP 393 [https://peps.python.org/pep-0393/] - Flexible String Representation

PEP written by Martin von Löwis; implementation by Torsten Becker and Martin von Löwis.

PEP 397: Python Launcher for Windows

The Python 3.3 Windows installer now includes a `py` launcher application that can be used to launch Python applications in a version independent fashion.

This launcher is invoked implicitly when double-clicking `*.py` files. If only a single Python version is installed on the system, that version will be used to run the file. If multiple versions are installed, the most recent version is used by default, but this can be overridden by including a Unix-style “shebang line” in the Python script.

The launcher can also be used explicitly from the command line as the `py` application. Running `py` follows the same version selection rules as implicitly launching scripts, but a more specific version can be selected by passing appropriate arguments (such as `-3` to request Python 3 when Python 2 is also installed, or `-2.6` to specifically request an earlier Python version when a more recent version is installed).

In addition to the launcher, the Windows installer now includes an option to add the newly installed Python to the system PATH. (Contributed by Brian Curtin in bpo-3561 [https://bugs.python.org/issue?@action=redirect&bpo=3561].)

See also

PEP 397 [https://peps.python.org/pep-0397/] - Python Launcher for Windows

PEP written by Mark Hammond and Martin v. Löwis; implementation by Vinay Sajip.

Launcher documentation: Python install manager

Installer PATH modification: Python install manager

PEP 3151: Reworking the OS and IO exception hierarchy

The hierarchy of exceptions raised by operating system errors is now both simplified and finer-grained.

You don’t have to worry anymore about choosing the appropriate exception type between `OSError`, `IOError`, `EnvironmentError`, `WindowsError`, `mmap.error`, `socket.error` or `select.error`. All these exception types are now only one: `OSError`. The other names are kept as aliases for compatibility reasons.

Also, it is now easier to catch a specific error condition. Instead of inspecting the `errno` attribute (or `args[0]`) for a particular constant from the `errno` module, you can catch the adequate `OSError` subclass. The available subclasses are the following:

- `BlockingIOError`
- `ChildProcessError`
- `ConnectionError`
- `FileExistsError`
- `FileNotFoundError`
- `InterruptedError`

- `IsADirectoryError`
- `NotADirectoryError`
- `PermissionError`
- `ProcessLookupError`
- `TimeoutError`

And the `ConnectionError` itself has finer-grained subclasses:

- `BrokenPipeError`
- `ConnectionAbortedError`
- `ConnectionRefusedError`
- `ConnectionResetError`

Thanks to the new exceptions, common usages of the `errno` can now be avoided. For example, the following code written for Python 3.2:

```
from errno import ENOENT, EACCES, EPERM

try:
    with open("document.txt") as f:
        content = f.read()
except IOError as err:
    if err.errno == ENOENT:
        print("document.txt file is missing")
    elif err.errno in (EACCES, EPERM):
        print("You are not allowed to read document.txt")
    else:
        raise
```

can now be written without the `errno` import and without manual inspection of exception attributes:

```
try:
    with open("document.txt") as f:
        content = f.read()
except FileNotFoundError:
    print("document.txt file is missing")
except PermissionError:
    print("You are not allowed to read document.txt")
```

See also

PEP 3151 [https://peps.python.org/pep-3151/] - Reworking the OS and IO Exception Hierarchy
PEP written and implemented by Antoine Pitrou

PEP 380: Syntax for Delegating to a Subgenerator

PEP 380 adds the `yield from` expression, allowing a generator to delegate part of its operations to another generator. This allows a section of code containing `yield` to be factored out and placed in another generator. Additionally, the subgenerator is allowed to return with a value, and the value is made available to the delegating generator.

While designed primarily for use in delegating to a subgenerator, the `yield from` expression actually allows delegation to arbitrary subiterators.

For simple iterators, `yield from iterable` is essentially just a shortened form of `for item in iterable: yield item`:

```
>>> def g(x):
...     yield from range(x, 0, -1)
...     yield from range(x)
...
>>> list(g(5))
[5, 4, 3, 2, 1, 0, 1, 2, 3, 4]
```

However, unlike an ordinary loop, `yield from` allows subgenerators to receive sent and thrown values directly from the calling scope, and return a final value to the outer generator:

```
>>> def accumulate():
...     tally = 0
...     while 1:
...         next = yield
...         if next is None:
...             return tally
...         tally += next
...
>>> def gather_tallies(tallies):
...     while 1:
...         tally = yield from accumulate()
...         tallies.append(tally)
...
>>> tallies = []
>>> acc = gather_tallies(tallies)
>>> next(acc)  # Ensure the accumulator is ready to accept
values
>>> for i in range(4):
...     acc.send(i)
...
>>> acc.send(None)  # Finish the first tally
>>> for i in range(5):
...     acc.send(i)
...
>>> acc.send(None)  # Finish the second tally
>>> tallies
[6, 10]
```

The main principle driving this change is to allow even generators that are designed to be used with the `send` and `throw` methods to be split into multiple subgenerators as easily as a single large function can be split into multiple subfunctions.

See also

PEP 380 [https://peps.python.org/pep-0380/] - Syntax for Delegating to a Subgenerator

PEP written by Greg Ewing; implementation by Greg Ewing, integrated into 3.3 by Renaud Blanch, Ryan Kelly and Nick Coghlan; documentation by Zbigniew Jędrzejewski-Szmek and Nick Coghlan

PEP 409: Suppressing exception context

PEP 409 introduces new syntax that allows the display of the chained exception context to be disabled. This allows cleaner error messages in applications that convert between exception types:

```
>>> class D:
...     def __init__(self, extra):
...         self._extra_attributes = extra
...     def __getattr__(self, attr):
...         try:
...             return self._extra_attributes[attr]
...         except KeyError:
...             raise AttributeError(attr) from None
...
>>> D({}).x
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 8, in __getattr__
AttributeError: x
```

Without the `from None` suffix to suppress the cause, the original exception would be displayed by default:

```
>>> class C:
...     def __init__(self, extra):
...         self._extra_attributes = extra
...     def __getattr__(self, attr):
...         try:
...             return self._extra_attributes[attr]
...         except KeyError:
...             raise AttributeError(attr)
...
>>> C({}).x
```

```
Traceback (most recent call last):
  File "<stdin>", line 6, in __getattr__
KeyError: 'x'

During handling of the above exception, another exception
occurred:

Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 8, in __getattr__
AttributeError: x
```

No debugging capability is lost, as the original exception context remains available if needed (for example, if an intervening library has incorrectly suppressed valuable underlying details):

```
>>> try:
...     D({}).x
... except AttributeError as exc:
...     print(repr(exc.__context__))
...
KeyError('x',)
```

See also

PEP 409 [https://peps.python.org/pep-0409/] - Suppressing exception context
PEP written by Ethan Furman; implemented by Ethan Furman and Nick Coghlan.

PEP 414: Explicit Unicode literals

To ease the transition from Python 2 for Unicode aware Python applications that make heavy use of Unicode literals, Python 3.3 once again supports the “`u`” prefix for string literals. This prefix has no semantic significance in Python 3, it is provided solely to reduce the number of purely mechanical changes in migrating to Python 3, making it easier for developers to focus on the more significant semantic changes (such as the stricter default separation of binary and text data).

See also

PEP 414 [https://peps.python.org/pep-0414/] - Explicit Unicode literals
PEP written by Armin Ronacher.

PEP 3155: Qualified name for classes and functions

Functions and class objects have a new `__qualname__` attribute representing the “path” from the module top-level to their definition. For global functions and classes, this is the same as `__name__`. For other functions and classes, it provides better information about where they were actually defined, and how they might be accessible from the global scope.

Example with (non-bound) methods:

```
>>> class C:
...     def meth(self):
...         pass
...
>>> C.meth.__name__
'meth'
>>> C.meth.__qualname__
'C.meth'
```

Example with nested classes:

```
>>> class C:
...     class D:
...         def meth(self):
...             pass
...
>>> C.D.__name__
'D'
>>> C.D.__qualname__
'C.D'
>>> C.D.meth.__name__
'meth'
>>> C.D.meth.__qualname__
'C.D.meth'
```

Example with nested functions:

```
>>> def outer():
...     def inner():
...         pass
...     return inner
...
>>> outer().__name__
'inner'
>>> outer().__qualname__
'outer.<locals>.inner'
```

The string representation of those objects is also changed to include the new, more precise information:

```
>>> str(C.D)
"<class '__main__.C.D'>"
>>> str(C.D.meth)
'<function C.D.meth at 0x7f46b9fe31e0>'
```

See also

PEP 3155 [https://peps.python.org/pep-3155/] - Qualified name for classes and functions

PEP written and implemented by Antoine Pitrou.

PEP 412: Key-Sharing Dictionary

Dictionaries used for the storage of objects’ attributes are now able to share part of their internal storage between each other (namely, the part which stores the keys and their respective hashes). This reduces the memory consumption of programs creating many instances of non-builtin types.

See also

PEP 412 [https://peps.python.org/pep-0412/] - Key-Sharing Dictionary

PEP written and implemented by Mark Shannon.

PEP 362: Function Signature Object

A new function `inspect.signature()` makes introspection of python callables easy and straightforward. A broad range of callables is supported: python functions, decorated or not, classes, and `functools.partial()` objects. New classes `inspect.Signature`, `inspect.Parameter` and `inspect.BoundArguments` hold information about the call signatures, such as, annotations, default values, parameters kinds, and bound arguments, which considerably simplifies writing decorators and any code that validates or amends calling signatures or arguments.

See also

PEP 362 [https://peps.python.org/pep-0362/]: - Function Signature Object

PEP written by Brett Cannon, Yury Selivanov, Larry Hastings, Jiwon Seo; implemented by Yury Selivanov.

PEP 421: Adding sys.implementation

A new attribute on the `sys` module exposes details specific to the implementation of the currently running interpreter. The initial set of attributes on `sys.implementation` are `name`, `version`, `hexversion`, and `cache_tag`.

The intention of `sys.implementation` is to consolidate into one namespace the implementation-specific data used by the standard library. This allows different Python implementations to share a single standard library code base much more easily. In its initial state, `sys.implementation` holds only a small portion of the implementation-specific data. Over time that ratio will shift in order to make the standard library more portable.

One example of improved standard library portability is `cache_tag`. As of Python 3.3, `sys.implementation.cache_tag` is used by `importlib` to support **PEP 3147** [https://peps.python.org/pep-3147/] compliance. Any Python implementation that uses `importlib` for its built-in import system may use `cache_tag` to control the caching behavior for modules.

SimpleNamespace

The implementation of `sys.implementation` also introduces a new type to Python: `types.SimpleNamespace`. In contrast to a mapping-based namespace, like `dict`, `SimpleNamespace` is attribute-based, like `object`. However, unlike `object`, `SimpleNamespace` instances are writable. This means that you can add, remove, and modify the namespace through normal attribute access.

See also

PEP 421 [https://peps.python.org/pep-0421/] - Adding sys.implementation

PEP written and implemented by Eric Snow.

Using importlib as the Implementation of Import

bpo-2377 [https://bugs.python.org/issue?@action=redirect&bpo=2377] - Replace __import__ w/ importlib.__import__ bpo-13959 [https://bugs.python.org/issue?@action=redirect&bpo=13959] - Re-implement parts of **`imp`** in pure Python bpo-14605 [https://bugs.python.org/issue?@action=redirect&bpo=14605] - Make import machinery explicit bpo-14646 [https://bugs.python.org/issue?@action=redirect&bpo=14646] - Require loaders set __loader__ and __package__

The `__import__()` function is now powered by `importlib.__import__()`. This work leads to the completion of “phase 2” of **PEP 302** [https://peps.python.org/pep-0302/]. There are multiple benefits to this change. First, it has allowed for more of the machinery powering import to be exposed instead of being implicit and hidden within the C code. It also provides a single implementation for all Python VMs supporting Python 3.3 to use, helping to end any VM-specific deviations in import semantics. And finally it eases the maintenance of import, allowing for future growth to occur.

For the common user, there should be no visible change in semantics. For those whose code currently manipulates import or calls import programmatically, the code changes that might possibly be required are covered in the Porting Python code section of this document.

New APIs

One of the large benefits of this work is the exposure of what goes into making the import statement work. That means the various importers that were once implicit are now fully exposed as part of the `importlib` package.

The abstract base classes defined in `importlib.abc` have been expanded to properly delineate between meta path finders and path entry finders by introducing `importlib.abc.MetaPathFinder` and `importlib.abc.PathEntryFinder`, respectively. The old ABC of `importlib.abc.Finder` is now only provided for backwards-compatibility and does not enforce any method requirements.

In terms of finders, `importlib.machinery.FileFinder` exposes the mechanism used to search for source and bytecode files of a module. Previously this class was an implicit member of `sys.path_hooks`.

For loaders, the new abstract base class `importlib.abc.FileLoader` helps write a loader that uses the file system as the storage mechanism for a module's code. The loader for source files (`importlib.machinery.SourceFileLoader`), sourceless bytecode files (`importlib.machinery.SourcelessFileLoader`), and extension modules (`importlib.machinery.ExtensionFileLoader`) are now available for direct use.

`ImportError` now has `name` and `path` attributes which are set when there is relevant data to provide. The message for failed imports will also provide the full name of the module now instead of just the tail end of the module's name.

The `importlib.invalidate_caches()` function will now call the method with the same name on all finders cached in `sys.path_importer_cache` to help clean up any stored state as necessary.

Visible Changes

For potential required changes to code, see the Porting Python code section.

Beyond the expanse of what `importlib` now exposes, there are other visible changes to import. The biggest is that `sys.meta_path` and `sys.path_hooks` now store all of the meta path finders and path entry hooks used by import. Previously the finders were implicit and hidden within the C code of import instead of being directly exposed. This means that one can now easily remove or change the order of the various finders to fit one's needs.

Another change is that all modules have a `__loader__` attribute, storing the loader used to create the module. **PEP 302** [https://peps.python.org/pep-0302/] has been updated to make this attribute mandatory for loaders to implement, so in the future once 3rd-party loaders have been updated people will be able to rely on the existence of the attribute. Until such time, though, import is setting the module post-load.

Loaders are also now expected to set the `__package__` attribute from **PEP 366** [https://peps.python.org/pep-0366/]. Once again, import itself is already setting this on all loaders from `importlib` and import itself is setting the attribute post-load.

`None` is now inserted into `sys.path_importer_cache` when no finder can be found on `sys.path_hooks`. Since **`imp.NullImporter`** is not directly exposed on `sys.path_hooks` it could no longer be relied upon to always be available to use as a value representing no finder found.

All other changes relate to semantic changes which should be taken into consideration when updating code for Python

3.3, and thus should be read about in the Porting Python code section of this document.

(Implementation by Brett Cannon)

Other Language Changes

Some smaller changes made to the core Python language are:

- Added support for Unicode name aliases and named sequences. Both **`unicodedata.lookup()`** and `'\N{...}'` now resolve name aliases, and **`unicodedata.lookup()`** resolves named sequences too.

  (Contributed by Ezio Melotti in bpo-12753 [https://bugs.python.org/issue?@action=redirect&bpo=12753].)
- Unicode database updated to UCD version 6.1.0
- Equality comparisons on **`range()`** objects now return a result reflecting the equality of the underlying sequences generated by those range objects. (bpo-13201 [https://bugs.python.org/issue?@action=redirect&bpo=13201])
- The `count()`, `find()`, `rfind()`, `index()` and `rindex()` methods of **`bytes`** and **`bytearray`** objects now accept an integer between 0 and 255 as their first argument.

  (Contributed by Petri Lehtinen in bpo-12170 [https://bugs.python.org/issue?@action=redirect&bpo=12170].)
- The `rjust()`, `ljust()`, and `center()` methods of **`bytes`** and **`bytearray`** now accept a **`bytearray`** for the `fill` argument. (Contributed by Petri Lehtinen in bpo-12380 [https://bugs.python.org/issue?@action=redirect&bpo=12380].)

- New methods have been added to **`list`** and **`bytearray`**: `copy()` and `clear()` (bpo-10516 [https://bugs.python.org/issue?@action=redirect&bpo=10516]). Consequently, **`MutableSequence`** now also defines a **`clear()`** method (bpo-11388 [https://bugs.python.org/issue?@action=redirect&bpo=11388]).
- Raw bytes literals can now be written `rb"..."` as well as `br"..."`.

  (Contributed by Antoine Pitrou in bpo-13748 [https://bugs.python.org/issue?@action=redirect&bpo=13748].)
- **`dict.setdefault()`** now does only one lookup for the given key, making it atomic when used with built-in types.

  (Contributed by Filip Gruszczyński in bpo-13521 [https://bugs.python.org/issue?@action=redirect&bpo=13521].)
- The error messages produced when a function call does not match the function signature have been significantly improved.

  (Contributed by Benjamin Peterson.)

A Finer-Grained Import Lock

Previous versions of CPython have always relied on a global import lock. This led to unexpected annoyances, such as deadlocks when importing a module would trigger code execution in a different thread as a side-effect. Clumsy workarounds were sometimes employed, such as the **`PyImport_ImportModuleNoBlock()`** C API function.

In Python 3.3, importing a module takes a per-module lock. This correctly serializes importation of a given module from multiple threads (preventing the exposure of incompletely initialized modules), while eliminating the aforementioned annoyances.

(Contributed by Antoine Pitrou in bpo-9260 [https://bugs.python.org/issue?@action=redirect&bpo=9260].)

Builtin functions and types

- **`open()`** gets a new *opener* parameter: the underlying file descriptor for the file object is then obtained by calling *opener* with (*file*, *flags*). It can be used to use custom flags like **`os.O_CLOEXEC`** for example. The `'x'` mode was added: open for exclusive creation, failing if the file already exists.
- **`print()`**: added the *flush* keyword argument. If the *flush* keyword argument is true, the stream is forcibly flushed.
- **`hash()`**: hash randomization is enabled by default, see **`object.__hash__()`** and **`PYTHONHASHSEED`**.
- The **`str`** type gets a new **`casefold()`** method: return a casefolded copy of the string, casefolded strings may be used for caseless matching. For example, `'ß'.casefold()` returns `'ss'`.
- The sequence documentation has been substantially rewritten to better explain the binary/text sequence distinction and to provide specific documentation sections for the individual builtin sequence types (bpo-4966 [https://bugs.python.org/issue?@action=redirect&bpo=4966]).

New Modules

faulthandler

This new debug module **`faulthandler`** contains functions to dump Python tracebacks explicitly, on a fault (a crash like a segmentation fault), after a timeout, or on a user signal. Call

`faulthandler.enable()` to install fault handlers for the **`SIGSEGV`**, **`SIGFPE`**, **`SIGABRT`**, **`SIGBUS`**, and **`SIGILL`** signals. You can also enable them at startup by setting the **`PYTHONFAULTHANDLER`** environment variable or by using **`-X`** `faulthandler` command line option.

Example of a segmentation fault on Linux:

```
$ python -q -X faulthandler
>>> import ctypes
>>> ctypes.string_at(0)
Fatal Python error: Segmentation fault

Current thread 0x00007fb899f39700:
  File "/home/python/cpython/Lib/ctypes/__init__.py", line 486
in string_at
  File "<stdin>", line 1 in <module>
Segmentation fault
```

ipaddress

The new **`ipaddress`** module provides tools for creating and manipulating objects representing IPv4 and IPv6 addresses, networks and interfaces (i.e. an IP address associated with a specific IP subnet).

(Contributed by Google and Peter Moody in **PEP 3144** [https://peps.python.org/pep-3144/].)

lzma

The newly added **`lzma`** module provides data compression and decompression using the LZMA algorithm, including support for the `.xz` and `.lzma` file formats.

(Contributed by Nadeem Vawda and Per Øyvind Karlsen in bpo-6715 [https://bugs.python.org/issue?@action=redirect&bpo=6715].)

Improved Modules

abc

Improved support for abstract base classes containing descriptors composed with abstract methods. The recommended approach to declaring abstract descriptors is now to provide `__isabstractmethod__` as a dynamically updated property. The built-in descriptors have been updated accordingly.

- `abc.abstractproperty` has been deprecated, use `property` with `abc.abstractmethod()` instead.
- `abc.abstractclassmethod` has been deprecated, use `classmethod` with `abc.abstractmethod()` instead.
- `abc.abstractstaticmethod` has been deprecated, use `staticmethod` with `abc.abstractmethod()` instead.

(Contributed by Darren Dale in bpo-11610 [https://bugs.python.org/issue?@action=redirect&bpo=11610].)

`abc.ABCMeta.register()` now returns the registered subclass, which means it can now be used as a class decorator (bpo-10868 [https://bugs.python.org/issue?@action=redirect&bpo=10868]).

array

The `array` module supports the long long type using `q` and `Q` type codes.

(Contributed by Oren Tirosh and Hirokazu Yamamoto in bpo-1172711 [https://bugs.python.org/issue?@action=redirect&bpo=1172711].)

base64

ASCII-only Unicode strings are now accepted by the decoding functions of the **`base64`** modern interface. For example, `base64.b64decode('YWJj')` returns `b'abc'`. (Contributed by Catalin Iacob in bpo-13641 [https://bugs.python.org/issue?@action=redirect&bpo=13641].)

binascii

In addition to the binary objects they normally accept, the `a2b_` functions now all also accept ASCII-only strings as input. (Contributed by Antoine Pitrou in bpo-13637 [https://bugs.python.org/issue?@action=redirect&bpo=13637].)

bz2

The **`bz2`** module has been rewritten from scratch. In the process, several new features have been added:

- New **`bz2.open()`** function: open a bzip2-compressed file in binary or text mode.
- **`bz2.BZ2File`** can now read from and write to arbitrary file-like objects, by means of its constructor's *fileobj* argument.

  (Contributed by Nadeem Vawda in bpo-5863 [https://bugs.python.org/issue?@action=redirect&bpo=5863].)
- **`bz2.BZ2File`** and **`bz2.decompress()`** can now decompress multi-stream inputs (such as those produced by the **pbzip2** tool). **`bz2.BZ2File`** can now also be used to create this type of file, using the `'a'` (append) mode.

  (Contributed by Nir Aides in bpo-1625 [https://bugs.python.org/issue?@action=redirect&bpo=1625].)

- **`bz2.BZ2File`** now implements all of the **`io.BufferedIOBase`** API, except for the **`detach()`** and **`truncate()`** methods.

codecs

The **`mbcs`** codec has been rewritten to handle correctly `replace` and `ignore` error handlers on all Windows versions. The **`mbcs`** codec now supports all error handlers, instead of only `replace` to encode and `ignore` to decode.

A new Windows-only codec has been added: `cp65001` (bpo-13216 [https://bugs.python.org/issue?@action=redirect&bpo=13216]). It is the Windows code page 65001 (Windows UTF-8, `CP_UTF8`). For example, it is used by `sys.stdout` if the console output code page is set to cp65001 (e.g., using `chcp 65001` command).

Multibyte CJK decoders now resynchronize faster. They only ignore the first byte of an invalid byte sequence. For example, `b'\xff\n'.decode('gb2312', 'replace')` now returns a `\n` after the replacement character.

(bpo-12016 [https://bugs.python.org/issue?@action=redirect&bpo=12016])

Incremental CJK codec encoders are no longer reset at each call to their encode() methods. For example:

```
>>> import codecs
>>> encoder = codecs.getincrementalencoder('hz')('strict')
>>> b''.join(encoder.encode(x) for x in
'\u52ff\u65bd\u65bc\u4eba\u3002 Bye.')
b'~{NpJ)l6HK!#~} Bye.'
```

This example gives `b'~{Np~}~{J)~}~{l6~}~{HK~}~{!#~} Bye.'` with older Python versions.

(bpo-12100 [https://bugs.python.org/issue?@action=redirect&bpo=12100])

The `unicode_internal` codec has been deprecated.

collections

Addition of a new **`ChainMap`** class to allow treating a number of mappings as a single unit. (Written by Raymond Hettinger for bpo-11089 [https://bugs.python.org/issue?@action=redirect&bpo=11089], made public in bpo-11297 [https://bugs.python.org/issue?@action=redirect&bpo=11297].)

The abstract base classes have been moved in a new **`collections.abc`** module, to better differentiate between the abstract and the concrete collections classes. Aliases for ABCs are still present in the **`collections`** module to preserve existing imports. (bpo-11085 [https://bugs.python.org/issue?@action=redirect&bpo=11085])

The **`Counter`** class now supports the unary `+` and `-` operators, as well as the in-place operators `+=`, `-=`, `|=`, and `&=`. (Contributed by Raymond Hettinger in bpo-13121 [https://bugs.python.org/issue?@action=redirect&bpo=13121].)

contextlib

`ExitStack` now provides a solid foundation for programmatic manipulation of context managers and similar cleanup functionality. Unlike the previous `contextlib.nested` API (which was deprecated and removed), the new API is designed to work correctly regardless of whether context managers acquire their resources in their `__init__` method (for example, file objects) or in their `__enter__` method (for example, synchronisation objects from the **`threading`** module).

(bpo-13585 [https://bugs.python.org/issue?@action=redirect&bpo=13585])

crypt

Addition of salt and modular crypt format (hashing method) and the `mksalt()` function to the `crypt` module.

(bpo-10924 [https://bugs.python.org/issue?@action=redirect&bpo=10924])

curses

- If the `curses` module is linked to the ncursesw library, use Unicode functions when Unicode strings or characters are passed (e.g. `waddwstr()`), and bytes functions otherwise (e.g. `waddstr()`).
- Use the locale encoding instead of `utf-8` to encode Unicode strings.
- `curses.window` has a new `curses.window.encoding` attribute.
- The `curses.window` class has a new `get_wch()` method to get a wide character
- The `curses` module has a new `unget_wch()` function to push a wide character so the next `get_wch()` will return it

(Contributed by Iñigo Serna in bpo-6755 [https://bugs.python.org/issue?@action=redirect&bpo=6755].)

datetime

- Equality comparisons between naive and aware `datetime` instances now return `False` instead of raising `TypeError` (bpo-15006 [https://bugs.python.org/issue?@action=redirect&bpo=15006]).
- New `datetime.datetime.timestamp()` method: Return POSIX timestamp corresponding to the `datetime` instance.
- The `datetime.datetime.strftime()` method supports formatting years older than 1000.
- The `datetime.datetime.astimezone()` method can now be called without arguments to convert datetime instance to the system timezone.

decimal

bpo-7652 [https://bugs.python.org/issue?@action=redirect&bpo=7652] - integrate fast native decimal arithmetic.
C-module and libmpdec written by Stefan Krah.

The new C version of the decimal module integrates the high speed libmpdec library for arbitrary precision correctly rounded decimal floating-point arithmetic. libmpdec conforms to IBM's General Decimal Arithmetic Specification.

Performance gains range from 10x for database applications to 100x for numerically intensive applications. These numbers are expected gains for standard precisions used in decimal floating-point arithmetic. Since the precision is user configurable, the exact figures may vary. For example, in integer bignum arithmetic the differences can be significantly higher.

The following table is meant as an illustration. Benchmarks are available at https://www.bytereef.org/mpdecimal/quickstart.html.

	decimal.py	_decimal	speedup
pi	42.02s	0.345s	120x
telco	172.19s	5.68s	30x
psycopg	3.57s	0.29s	12x

Features

- The `FloatOperation` signal optionally enables stricter semantics for mixing floats and Decimals.
- If Python is compiled without threads, the C version automatically disables the expensive thread local context machinery. In this case, the variable `HAVE_THREADS` is set to `False`.

API changes

- The C module has the following context limits, depending on the machine architecture:

	32-bit	64-bit
`MAX_PREC`	`425000000`	`999999999999999999`
`MAX_EMAX`	`425000000`	`999999999999999999`
`MIN_EMIN`	`-425000000`	`-999999999999999999`

- In the context templates (`DefaultContext`, `BasicContext` and `ExtendedContext`) the magnitude of `Emax` and `Emin` has changed to `999999`.

- The `Decimal` constructor in decimal.py does not observe the context limits and converts values with arbitrary exponents or precision exactly. Since the C version has internal limits, the following scheme is used: If possible, values are converted exactly, otherwise `InvalidOperation` is raised and the result is NaN. In the latter case it is always possible to use `create_decimal()` in order to obtain a rounded or inexact value.

- The power function in decimal.py is always correctly rounded. In the C version, it is defined in terms of the correctly rounded `exp()` and `ln()` functions, but the final result is only “almost always correctly rounded”.

- In the C version, the context dictionary containing the signals is a `MutableMapping`. For speed reasons, `flags` and `traps` always refer to the same `MutableMapping` that the context was initialized with. If a new signal dictionary is assigned, `flags` and `traps` are updated with the new values, but they do not reference the RHS dictionary.

- Pickling a `Context` produces a different output in order to have a common interchange format for the Python and C versions.

- The order of arguments in the `Context` constructor has been changed to match the order displayed by `repr()`.

- The `watchexp` parameter in the `quantize()` method is deprecated.

email

Policy Framework

The email package now has a `policy` framework. A `Policy` is an object with several methods and properties that control how the email package behaves. The primary policy for Python 3.3 is the `Compat32` policy, which provides backward compatibility with the email package in Python 3.2. A `policy` can be specified when an email message is parsed by a `parser`, or when a `Message` object is created, or when an email is serialized using a `generator`. Unless overridden, a policy passed to a `parser` is inherited by all the `Message` object and sub-objects created by the `parser`. By default a `generator` will use the policy of the `Message` object it is serializing. The default policy is [`compat32`](#).

The minimum set of controls implemented by all `policy` objects are:

max_line_length	The maximum length, excluding the linesep character(s), individual lines may have when a `Message` is serialized. Defaults to 78.
linesep	The character used to separate individual lines when a `Message` is serialized. Defaults to `\n`.
cte_type	`7bit` or `8bit`. `8bit` applies only to a `Bytes generator`, and means that non-ASCII may be used where allowed by the protocol (or where it exists in the original input).
raise_on_defect	Causes a `parser` to raise error when defects are encountered instead of adding them to the `Message` object's `defects` list.

A new policy instance, with new settings, is created using the **`clone()`** method of policy objects. `clone` takes any of the above controls as keyword arguments. Any control not specified in the call retains its default value. Thus you can create a policy that uses `\r\n` linesep characters like this:

```
mypolicy = compat32.clone(linesep='\r\n')
```

Policies can be used to make the generation of messages in the format needed by your application simpler. Instead of having to remember to specify `linesep='\r\n'` in all the places you call a `generator`, you can specify it once, when you set the policy used by the `parser` or the `Message`, whichever your program uses to create `Message` objects. On the other hand, if you need to generate messages in multiple forms, you can still specify the parameters in the appropriate `generator` call. Or you can have custom policy instances for your different cases, and pass those in when you create the `generator`.

Provisional Policy with New Header API

While the policy framework is worthwhile all by itself, the main motivation for introducing it is to allow the creation of new policies that implement new features for the email package in a way that maintains backward compatibility for those who do not use the new policies. Because the new policies introduce a new API, we are releasing them in Python 3.3 as a provisional policy. Backwards incompatible changes (up to and including removal of the code) may occur if deemed necessary by the core developers.

The new policies are instances of **`EmailPolicy`**, and add the following additional controls:

refold_source	Controls whether or not headers parsed by a **`parser`** are refolded by the **`generator`**. It can be `none`, `long`, or `all`. The default is `long`, which means that source headers with a line longer than `max_line_length` get refolded. `none` means no line get refolded, and `all` means that all lines get refolded.

header_factory	A callable that take a `name` and `value` and produces a custom header object.

The `header_factory` is the key to the new features provided by the new policies. When one of the new policies is used, any header retrieved from a `Message` object is an object produced by the `header_factory`, and any time you set a header on a `Message` it becomes an object produced by `header_factory`. All such header objects have a `name` attribute equal to the header name. Address and Date headers have additional attributes that give you access to the parsed data of the header. This means you can now do things like this:

```
>>> m = Message(policy=SMTP)
>>> m['To'] = 'Éric <foo@example.com>'
>>> m['to']
'Éric <foo@example.com>'
>>> m['to'].addresses
(Address(display_name='Éric', username='foo',
domain='example.com'),)
>>> m['to'].addresses[0].username
'foo'
>>> m['to'].addresses[0].display_name
'Éric'
>>> m['Date'] = email.utils.localtime()
>>> m['Date'].datetime
datetime.datetime(2012, 5, 25, 21, 39, 24, 465484,
tzinfo=datetime.timezone(datetime.timedelta(-1, 72000), 'EDT'))
>>> m['Date']
'Fri, 25 May 2012 21:44:27 -0400'
>>> print(m)
To: =?utf-8?q?=C3=89ric?= <foo@example.com>
Date: Fri, 25 May 2012 21:44:27 -0400
```

You will note that the unicode display name is automatically encoded as `utf-8` when the message is serialized, but that when the header is accessed directly, you get the unicode version. This eliminates any need to deal with the `email.header` `decode_header()` or `make_header()` functions.

You can also create addresses from parts:

```
>>> m['cc'] = [Group('pals', [Address('Bob', 'bob',
'example.com'),
...                               Address('Sally', 'sally',
'example.com')]),
...               Address('Bonzo', addr_spec='bonz@laugh.com')]
>>> print(m)
To: =?utf-8?q?=C3=89ric?= <foo@example.com>
Date: Fri, 25 May 2012 21:44:27 -0400
cc: pals: Bob <bob@example.com>, Sally <sally@example.com>;,
Bonzo <bonz@laugh.com>
```

Decoding to unicode is done automatically:

```
>>> m2 = message_from_string(str(m))
>>> m2['to']
'Éric <foo@example.com>'
```

When you parse a message, you can use the `addresses` and `groups` attributes of the header objects to access the groups and individual addresses:

```
>>> m2['cc'].addresses
(Address(display_name='Bob', username='bob',
domain='example.com'), Address(display_name='Sally',
username='sally', domain='example.com'),
Address(display_name='Bonzo', username='bonz',
domain='laugh.com'))
>>> m2['cc'].groups
(Group(display_name='pals', addresses=
(Address(display_name='Bob', username='bob',
domain='example.com'), Address(display_name='Sally',
username='sally', domain='example.com')),
Group(display_name=None, addresses=
(Address(display_name='Bonzo', username='bonz',
domain='laugh.com'),))
```

In summary, if you use one of the new policies, header manipulation works the way it ought to: your application works with unicode strings, and the email package transparently encodes and decodes the unicode to and from the RFC standard Content Transfer Encodings.

Other API Changes

New `BytesHeaderParser`, added to the `parser` module to complement `HeaderParser` and complete the Bytes API.

New utility functions:

- `format_datetime()`: given a `datetime`, produce a string formatted for use in an email header.
- `parsedate_to_datetime()`: given a date string from an email header, convert it into an aware `datetime`, or a naive `datetime` if the offset is `-0000`.
- `localtime()`: With no argument, returns the current local time as an aware `datetime` using the local `timezone`. Given an aware `datetime`, converts it into an aware `datetime` using the local `timezone`.

ftplib

- `ftplib.FTP` now accepts a `source_address` keyword argument to specify the `(host, port)` to use as the source address in the bind call when creating the outgoing socket. (Contributed by Giampaolo Rodolà in bpo-8594 [https://bugs.python.org/issue?@action=redirect&bpo=8594].)
- The `FTP_TLS` class now provides a new `ccc()` function to revert control channel back to plaintext. This can be useful to take advantage of firewalls that know how to handle NAT with non-secure FTP without opening fixed

ports. (Contributed by Giampaolo Rodolà in bpo-12139 [https://bugs.python.org/issue?@action=redirect&bpo=12139].)

- Added `ftplib.FTP.mlsd()` method which provides a parsable directory listing format and deprecates `ftplib.FTP.nlst()` and `ftplib.FTP.dir()`. (Contributed by Giampaolo Rodolà in bpo-11072 [https://bugs.python.org/issue?@action=redirect&bpo=11072].)

functools

The `functools.lru_cache()` decorator now accepts a `typed` keyword argument (that defaults to `False` to ensure that it caches values of different types that compare equal in separate cache slots. (Contributed by Raymond Hettinger in bpo-13227 [https://bugs.python.org/issue?@action=redirect&bpo=13227].)

gc

It is now possible to register callbacks invoked by the garbage collector before and after collection using the new `callbacks` list.

hmac

A new `compare_digest()` function has been added to prevent side channel attacks on digests through timing analysis. (Contributed by Nick Coghlan and Christian Heimes in bpo-15061 [https://bugs.python.org/issue?@action=redirect&bpo=15061].)

http

`http.server.BaseHTTPRequestHandler` now buffers the headers and writes them all at once when `end_headers()` is called. A new method `flush_headers()` can be used to directly manage when the accumulated headers are sent. (Contributed by

Andrew Schaaf in bpo-3709 [https://bugs.python.org/issue?@action=redirect&bpo=3709].)

`http.server` now produces valid `HTML 4.01 strict` output. (Contributed by Ezio Melotti in bpo-13295 [https://bugs.python.org/issue?@action=redirect&bpo=13295].)

`http.client.HTTPResponse` now has a **`readinto()`** method, which means it can be used as an **`io.RawIOBase`** class. (Contributed by John Kuhn in bpo-13464 [https://bugs.python.org/issue?@action=redirect&bpo=13464].)

html

`html.parser.HTMLParser` is now able to parse broken markup without raising errors, therefore the *strict* argument of the constructor and the **`HTMLParseError`** exception are now deprecated. The ability to parse broken markup is the result of a number of bug fixes that are also available on the latest bug fix releases of Python 2.7/3.2. (Contributed by Ezio Melotti in bpo-15114 [https://bugs.python.org/issue?@action=redirect&bpo=15114], and bpo-14538 [https://bugs.python.org/issue?@action=redirect&bpo=14538], bpo-13993 [https://bugs.python.org/issue?@action=redirect&bpo=13993], bpo-13960 [https://bugs.python.org/issue?@action=redirect&bpo=13960], bpo-13358 [https://bugs.python.org/issue?@action=redirect&bpo=13358], bpo-1745761 [https://bugs.python.org/issue?@action=redirect&bpo=1745761], bpo-755670 [https://bugs.python.org/issue?@action=redirect&bpo=755670], bpo-13357 [https://bugs.python.org/issue?@action=redirect&bpo=13357], bpo-12629 [https://bugs.python.org/issue?@action=redirect&bpo=12629], bpo-1200313 [https://bugs.python.org/issue?@action=redirect&bpo=1200313], bpo-670664 [https://bugs.python.org/issue?@action=redirect&bpo=670664], bpo-13273 [https://bugs.python.org/issue?@action=redirect&bpo=13273], bpo-12888 [https://bugs.python.org/issue?@action=redirect&bpo=12888], bpo-7311 [https://bugs.python.org/issue?@action=redirect&bpo=7311].)

A new **`html5`** dictionary that maps HTML5 named character references to the equivalent Unicode character(s) (e.g. `html5['gt;'] == '>'`) has been added to the **`html.entities`** module. The dictionary is now also used by **`HTMLParser`**. (Contributed by Ezio Melotti in bpo-11113 [https://bugs.python.org/issue?@action=redirect&bpo=11113] and bpo-15156 [https://bugs.python.org/issue?@action=redirect&bpo=15156].)

imaplib

The **`IMAP4_SSL`** constructor now accepts an SSLContext parameter to control parameters of the secure channel.

(Contributed by Sijin Joseph in bpo-8808 [https://bugs.python.org/issue?@action=redirect&bpo=8808].)

inspect

A new **`getclosurevars()`** function has been added. This function reports the current binding of all names referenced from the function body and where those names were resolved, making it easier to verify correct internal state when testing code that relies on stateful closures.

(Contributed by Meador Inge and Nick Coghlan in bpo-13062 [https://bugs.python.org/issue?@action=redirect&bpo=13062].)

A new **`getgeneratorlocals()`** function has been added. This function reports the current binding of local variables in the generator's stack frame, making it easier to verify correct internal state when testing generators.

(Contributed by Meador Inge in bpo-15153 [https://bugs.python.org/issue?@action=redirect&bpo=15153].)

io

The `open()` function has a new `'x'` mode that can be used to exclusively create a new file, and raise a `FileExistsError` if the file already exists. It is based on the C11 ‘x’ mode to fopen().

(Contributed by David Townshend in bpo-12760 [https://bugs.python.org/issue?@action=redirect&bpo=12760].)

The constructor of the `TextIOWrapper` class has a new *write_through* optional argument. If *write_through* is `True`, calls to **`write()`** are guaranteed not to be buffered: any data written on the `TextIOWrapper` object is immediately handled to its underlying binary buffer.

itertools

`accumulate()` now takes an optional `func` argument for providing a user-supplied binary function.

logging

The `basicConfig()` function now supports an optional `handlers` argument taking an iterable of handlers to be added to the root logger.

A class level attribute **`append_nul`** has been added to `SysLogHandler` to allow control of the appending of the `NUL` (`\000`) byte to syslog records, since for some daemons it is required while for others it is passed through to the log.

math

The `math` module has a new function, `log2()`, which returns the base-2 logarithm of *x*.

(Written by Mark Dickinson in bpo-11888 [https://bugs.python.org/issue?@action=redirect&bpo=11888].)

mmap

The `read()` method is now more compatible with other file-like objects: if the argument is omitted or specified as `None`, it returns the bytes from the current file position to the end of the mapping. (Contributed by Petri Lehtinen in bpo-12021 [https://bugs.python.org/issue?@action=redirect&bpo=12021].)

multiprocessing

The new `multiprocessing.connection.wait()` function allows polling multiple objects (such as connections, sockets and pipes) with a timeout. (Contributed by Richard Oudkerk in bpo-12328 [https://bugs.python.org/issue?@action=redirect&bpo=12328].)

`multiprocessing.connection.Connection` objects can now be transferred over multiprocessing connections. (Contributed by Richard Oudkerk in bpo-4892 [https://bugs.python.org/issue?@action=redirect&bpo=4892].)

`multiprocessing.Process` now accepts a `daemon` keyword argument to override the default behavior of inheriting the `daemon` flag from the parent process (bpo-6064 [https://bugs.python.org/issue?@action=redirect&bpo=6064]).

New attribute `multiprocessing.Process.sentinel` allows a program to wait on multiple `Process` objects at one time using the appropriate OS primitives (for example, `select` on posix systems).

New methods `multiprocessing.pool.Pool.starmap()` and `starmap_async()` provide `itertools.starmap()` equivalents to the existing `multiprocessing.pool.Pool.map()` and `map_async()`

functions. (Contributed by Hynek Schlawack in bpo-12708 [https://bugs.python.org/issue?@action=redirect&bpo=12708].)

nntplib

The **`nntplib.NNTP`** class now supports the context management protocol to unconditionally consume **`socket.error`** exceptions and to close the NNTP connection when done:

```
>>> from nntplib import NNTP
>>> with NNTP('news.gmane.org') as n:
...     n.group('gmane.comp.python.committers')
...
('211 1755 1 1755 gmane.comp.python.committers', 1755, 1, 1755,
'gmane.comp.python.committers')
>>>
```

(Contributed by Giampaolo Rodolà in bpo-9795 [https://bugs.python.org/issue?@action=redirect&bpo=9795].)

os

- The **`os`** module has a new **`pipe2()`** function that makes it possible to create a pipe with **`O_CLOEXEC`** or **`O_NONBLOCK`** flags set atomically. This is especially useful to avoid race conditions in multi-threaded programs.

- The **`os`** module has a new **`sendfile()`** function which provides an efficient “zero-copy” way for copying data from one file (or socket) descriptor to another. The phrase “zero-copy” refers to the fact that all of the copying of data between the two descriptors is done entirely by the kernel, with no copying of data into userspace buffers. **`sendfile()`** can be used to efficiently copy data from a file on disk to a network socket, e.g. for downloading a file.

(Patch submitted by Ross Lagerwall and Giampaolo Rodolà in bpo-10882 [https://bugs.python.org/issue?@action=redirect&bpo=10882].)

- To avoid race conditions like symlink attacks and issues with temporary files and directories, it is more reliable (and also faster) to manipulate file descriptors instead of file names. Python 3.3 enhances existing functions and introduces new functions to work on file descriptors (bpo-4761 [https://bugs.python.org/issue?@action=redirect&bpo=4761], bpo-10755 [https://bugs.python.org/issue?@action=redirect&bpo=10755] and bpo-14626 [https://bugs.python.org/issue?@action=redirect&bpo=14626]).
  - The **`os`** module has a new **`fwalk()`** function similar to **`walk()`** except that it also yields file descriptors referring to the directories visited. This is especially useful to avoid symlink races.
  - The following functions get new optional *dir_fd* (paths relative to directory descriptors) and/or *follow_symlinks* (not following symlinks): **`access()`**, **`chflags()`**, **`chmod()`**, **`chown()`**, **`link()`**, **`lstat()`**, **`mkdir()`**, **`mkfifo()`**, **`mknod()`**, **`open()`**, **`readlink()`**, **`remove()`**, **`rename()`**, **`replace()`**, **`rmdir()`**, **`stat()`**, **`symlink()`**, **`unlink()`**, **`utime()`**. Platform support for using these parameters can be checked via the sets **`os.supports_dir_fd`** and **`os.supports_follow_symlinks`**.
  - The following functions now support a file descriptor for their path argument: **`chdir()`**, **`chmod()`**, **`chown()`**, **`execve()`**, **`listdir()`**, **`pathconf()`**, **`exists()`**, **`stat()`**, **`statvfs()`**, **`utime()`**. Platform support for this can be checked via the **`os.supports_fd`** set.

- **`access()`** accepts an `effective_ids` keyword argument to turn on using the effective uid/gid rather than the real uid/gid in the access check. Platform support for this can be checked via the `supports_effective_ids` set.

- The `os` module has two new functions: `getpriority()` and `setpriority()`. They can be used to get or set process niceness/priority in a fashion similar to `os.nice()` but extended to all processes instead of just the current one.

  (Patch submitted by Giampaolo Rodolà in bpo-10784 [https://bugs.python.org/issue?@action=redirect&bpo=10784].)

- The new `os.replace()` function allows cross-platform renaming of a file with overwriting the destination. With `os.rename()`, an existing destination file is overwritten under POSIX, but raises an error under Windows. (Contributed by Antoine Pitrou in bpo-8828 [https://bugs.python.org/issue?@action=redirect&bpo=8828].)

- The stat family of functions (`stat()`, `fstat()`, and `lstat()`) now support reading a file's timestamps with nanosecond precision. Symmetrically, `utime()` can now write file timestamps with nanosecond precision. (Contributed by Larry Hastings in bpo-14127 [https://bugs.python.org/issue?@action=redirect&bpo=14127].)

- The new `os.get_terminal_size()` function queries the size of the terminal attached to a file descriptor. See also `shutil.get_terminal_size()`. (Contributed by Zbigniew Jędrzejewski-Szmek in bpo-13609 [https://bugs.python.org/issue?@action=redirect&bpo=13609].)

- New functions to support Linux extended attributes (bpo-12720 [https://bugs.python.org/issue?@action=redirect&bpo=12720]): `getxattr()`, `listxattr()`, `removexattr()`, `setxattr()`.

- New interface to the scheduler. These functions control how a process is allocated CPU time by the operating system. New functions: `sched_get_priority_max()`, `sched_get_priority_min()`, `sched_getaffinity()`, `sched_getparam()`, `sched_getscheduler()`, `sched_rr_get_interval()`, `sched_setaffinity()`, `sched_setparam()`, `sched_setscheduler()`, `sched_yield()`,
- New functions to control the file system:
  - `posix_fadvise()`: Announces an intention to access data in a specific pattern thus allowing the kernel to make optimizations.
  - `posix_fallocate()`: Ensures that enough disk space is allocated for a file.
  - `sync()`: Force write of everything to disk.
- Additional new posix functions:
  - `lockf()`: Apply, test or remove a POSIX lock on an open file descriptor.
  - `pread()`: Read from a file descriptor at an offset, the file offset remains unchanged.
  - `pwrite()`: Write to a file descriptor from an offset, leaving the file offset unchanged.
  - `readv()`: Read from a file descriptor into a number of writable buffers.
  - `truncate()`: Truncate the file corresponding to *path*, so that it is at most *length* bytes in size.
  - `waitid()`: Wait for the completion of one or more child processes.
  - `writev()`: Write the contents of *buffers* to a file descriptor, where *buffers* is an arbitrary sequence of buffers.
  - `getgrouplist()` (bpo-9344 [https://bugs.python.org/issue?@action=redirect&bpo=9344]): Return list of group ids that specified user belongs to.
- `times()` and `uname()`: Return type changed from a tuple to a tuple-like object with named attributes.

- Some platforms now support additional constants for the **`lseek()`** function, such as `os.SEEK_HOLE` and `os.SEEK_DATA`.
- New constants **`RTLD_LAZY`**, **`RTLD_NOW`**, **`RTLD_GLOBAL`**, **`RTLD_LOCAL`**, **`RTLD_NODELETE`**, **`RTLD_NOLOAD`**, and **`RTLD_DEEPBIND`** are available on platforms that support them. These are for use with the **`sys.setdlopenflags()`** function, and supersede the similar constants defined in **`ctypes`** and **`DLFCN`**. (Contributed by Victor Stinner in bpo-13226 [https://bugs.python.org/issue?@action=redirect&bpo=13226].)
- **`os.symlink()`** now accepts (and ignores) the `target_is_directory` keyword argument on non-Windows platforms, to ease cross-platform support.

pdb

Tab-completion is now available not only for command names, but also their arguments. For example, for the `break` command, function and file names are completed.

(Contributed by Georg Brandl in bpo-14210 [https://bugs.python.org/issue?@action=redirect&bpo=14210])

pickle

`pickle.Pickler` objects now have an optional **`dispatch_table`** attribute allowing per-pickler reduction functions to be set.

(Contributed by Richard Oudkerk in bpo-14166 [https://bugs.python.org/issue?@action=redirect&bpo=14166].)

pydoc

The Tk GUI and the **`serve()`** function have been removed from the **`pydoc`** module: `pydoc -g` and **`serve()`** have been deprecated in Python 3.2.

re

`str` regular expressions now support `\u` and `\U` escapes.

(Contributed by Serhiy Storchaka in bpo-3665 [https://bugs.python.org/issue?@action=redirect&bpo=3665].)

sched

- `run()` now accepts a *blocking* parameter which when set to false makes the method execute the scheduled events due to expire soonest (if any) and then return immediately. This is useful in case you want to use the `scheduler` in non-blocking applications. (Contributed by Giampaolo Rodolà in bpo-13449 [https://bugs.python.org/issue?@action=redirect&bpo=13449].)
- `scheduler` class can now be safely used in multi-threaded environments. (Contributed by Josiah Carlson and Giampaolo Rodolà in bpo-8684 [https://bugs.python.org/issue?@action=redirect&bpo=8684].)
- *timefunc* and *delayfunct* parameters of `scheduler` class constructor are now optional and defaults to `time.time()` and `time.sleep()` respectively. (Contributed by Chris Clark in bpo-13245 [https://bugs.python.org/issue?@action=redirect&bpo=13245].)
- `enter()` and `enterabs()` *argument* parameter is now optional. (Contributed by Chris Clark in bpo-13245 [https://bugs.python.org/issue?@action=redirect&bpo=13245].)
- `enter()` and `enterabs()` now accept a *kwargs* parameter. (Contributed by Chris Clark in bpo-13245 [https://bugs.python.org/issue?@action=redirect&bpo=13245].)

select

Solaris and derivative platforms have a new class `select.devpoll` for high performance asynchronous sockets via `/dev/poll`. (Contributed by Jesús Cea Avión in bpo-6397 [https://bugs.python.org/issue?@action=redirect&bpo=6397].)

shlex

The previously undocumented helper function `quote` from the **`pipes`** modules has been moved to the **`shlex`** module and documented. **`quote()`** properly escapes all characters in a string that might be otherwise given special meaning by the shell.

shutil

- New functions:
    - **`disk_usage()`**: provides total, used and free disk space statistics. (Contributed by Giampaolo Rodolà in bpo-12442 [https://bugs.python.org/issue?@action=redirect&bpo=12442].)
    - **`chown()`**: allows one to change user and/or group of the given path also specifying the user/group names and not only their numeric ids. (Contributed by Sandro Tosi in bpo-12191 [https://bugs.python.org/issue?@action=redirect&bpo=12191].)
    - **`shutil.get_terminal_size()`**: returns the size of the terminal window to which the interpreter is attached. (Contributed by Zbigniew Jędrzejewski-Szmek in bpo-13609 [https://bugs.python.org/issue?@action=redirect&bpo=13609].)
- **`copy2()`** and **`copystat()`** now preserve file timestamps with nanosecond precision on platforms that support it. They also preserve file “extended attributes” on Linux. (Contributed by Larry Hastings in bpo-14127 [https://bugs.python.org/issue?@action=redirect&bpo=14127] and bpo-15238 [https://bugs.python.org/issue?@action=redirect&bpo=15238].)
- Several functions now take an optional `symlinks` argument: when that parameter is true, symlinks aren’t

dereferenced and the operation instead acts on the symlink itself (or creates one, if relevant). (Contributed by Hynek Schlawack in bpo-12715 [https://bugs.python.org/issue?@action=redirect&bpo=12715].)

- When copying files to a different file system, `move()` now handles symlinks the way the posix `mv` command does, recreating the symlink rather than copying the target file contents. (Contributed by Jonathan Niehof in bpo-9993 [https://bugs.python.org/issue?@action=redirect&bpo=9993].) `move()` now also returns the `dst` argument as its result.
- `rmtree()` is now resistant to symlink attacks on platforms which support the new `dir_fd` parameter in `os.open()` and `os.unlink()`. (Contributed by Martin von Löwis and Hynek Schlawack in bpo-4489 [https://bugs.python.org/issue?@action=redirect&bpo=4489].)

signal

- The `signal` module has new functions:
    - `pthread_sigmask()`: fetch and/or change the signal mask of the calling thread (Contributed by Jean-Paul Calderone in bpo-8407 [https://bugs.python.org/issue?@action=redirect&bpo=8407]);
    - `pthread_kill()`: send a signal to a thread;
    - `sigpending()`: examine pending functions;
    - `sigwait()`: wait a signal;
    - `sigwaitinfo()`: wait for a signal, returning detailed information about it;
    - `sigtimedwait()`: like `sigwaitinfo()` but with a timeout.
- The signal handler writes the signal number as a single byte instead of a nul byte into the wakeup file descriptor. So it is possible to wait more than one signal and know which signals were raised.
- `signal.signal()` and `signal.siginterrupt()` raise an OSError, instead of a RuntimeError: OSError has an errno attribute.

smtpd

The **`smtpd`** module now supports **RFC 5321** [https://datatracker.ietf.org/doc/html/rfc5321.html] (extended SMTP) and **RFC 1870** [https://datatracker.ietf.org/doc/html/rfc1870.html] (size extension). Per the standard, these extensions are enabled if and only if the client initiates the session with an `EHLO` command.

(Initial `ELHO` support by Alberto Trevino. Size extension by Juhana Jauhiainen. Substantial additional work on the patch contributed by Michele Orrù and Dan Boswell. bpo-8739 [https://bugs.python.org/issue?@action=redirect&bpo=8739])

smtplib

The **`SMTP`**, **`SMTP_SSL`**, and **`LMTP`** classes now accept a `source_address` keyword argument to specify the `(host, port)` to use as the source address in the bind call when creating the outgoing socket. (Contributed by Paulo Scardine in bpo-11281 [https://bugs.python.org/issue?@action=redirect&bpo=11281].)

`SMTP` now supports the context management protocol, allowing an `SMTP` instance to be used in a `with` statement. (Contributed by Giampaolo Rodolà in bpo-11289 [https://bugs.python.org/issue?@action=redirect&bpo=11289].)

The **`SMTP_SSL`** constructor and the **`starttls()`** method now accept an SSLContext parameter to control parameters of the secure channel. (Contributed by Kasun Herath in bpo-8809 [https://bugs.python.org/issue?@action=redirect&bpo=8809].)

socket

- The **`socket`** class now exposes additional methods to process ancillary data when supported by the underlying platform:

  - `sendmsg()`
  - `recvmsg()`
  - `recvmsg_into()`

  (Contributed by David Watson in bpo-6560 [https://bugs.python.org/issue?@action=redirect&bpo=6560], based on an earlier patch by Heiko Wundram)

- The `socket` class now supports the PF_CAN protocol family (https://en.wikipedia.org/wiki/Socketcan), on Linux (https://lwn.net/Articles/253425).

  (Contributed by Matthias Fuchs, updated by Tiago Gonçalves in bpo-10141 [https://bugs.python.org/issue?@action=redirect&bpo=10141].)

- The `socket` class now supports the PF_RDS protocol family (https://en.wikipedia.org/wiki/Reliable_Datagram_Sockets and https://oss.oracle.com/projects/rds [https://web.archive.org/web/20130115155505/https://oss.oracle.com/projects/rds/]).

- The `socket` class now supports the `PF_SYSTEM` protocol family on OS X. (Contributed by Michael Goderbauer in bpo-13777 [https://bugs.python.org/issue?@action=redirect&bpo=13777].)

- New function `sethostname()` allows the hostname to be set on Unix systems if the calling process has sufficient privileges. (Contributed by Ross Lagerwall in bpo-10866 [https://bugs.python.org/issue?@action=redirect&bpo=10866].)

socketserver

`BaseServer` now has an overridable method `service_actions()` that is called by the `serve_forever()` method in the service loop. `ForkingMixIn` now uses this to clean up zombie child processes. (Contributed by Justin Warkentin in bpo-11109 [https://bugs.python.org/issue?@action=redirect&bpo=11109].)

sqlite3

New `sqlite3.Connection` method `set_trace_callback()` can be used to capture a trace of all sql commands processed by sqlite. (Contributed by Torsten Landschoff in bpo-11688 [https://bugs.python.org/issue?@action=redirect&bpo=11688].)

ssl

- The `ssl` module has two new random generation functions:
  - `RAND_bytes()`: generate cryptographically strong pseudo-random bytes.
  - **`RAND_pseudo_bytes()`**: generate pseudo-random bytes.

  (Contributed by Victor Stinner in bpo-12049 [https://bugs.python.org/issue?@action=redirect&bpo=12049].)
- The `ssl` module now exposes a finer-grained exception hierarchy in order to make it easier to inspect the various kinds of errors. (Contributed by Antoine Pitrou in bpo-11183 [https://bugs.python.org/issue?@action=redirect&bpo=11183].)
- `load_cert_chain()` now accepts a *password* argument to be used if the private key is encrypted. (Contributed by Adam Simpkins in bpo-12803 [https://bugs.python.org/issue?@action=redirect&bpo=12803].)

- Diffie-Hellman key exchange, both regular and Elliptic Curve-based, is now supported through the `load_dh_params()` and `set_ecdh_curve()` methods. (Contributed by Antoine Pitrou in bpo-13626 [https://bugs.python.org/issue?@action=redirect&bpo=13626] and bpo-13627 [https://bugs.python.org/issue?@action=redirect&bpo=13627].)
- SSL sockets have a new `get_channel_binding()` method allowing the implementation of certain authentication mechanisms such as SCRAM-SHA-1-PLUS. (Contributed by Jacek Konieczny in bpo-12551 [https://bugs.python.org/issue?@action=redirect&bpo=12551].)
- You can query the SSL compression algorithm used by an SSL socket, thanks to its new `compression()` method. The new attribute `OP_NO_COMPRESSION` can be used to disable compression. (Contributed by Antoine Pitrou in bpo-13634 [https://bugs.python.org/issue?@action=redirect&bpo=13634].)
- Support has been added for the Next Protocol Negotiation extension using the `ssl.SSLContext.set_npn_protocols()` method. (Contributed by Colin Marc in bpo-14204 [https://bugs.python.org/issue?@action=redirect&bpo=14204].)
- SSL errors can now be introspected more easily thanks to `library` and `reason` attributes. (Contributed by Antoine Pitrou in bpo-14837 [https://bugs.python.org/issue?@action=redirect&bpo=14837].)
- The `get_server_certificate()` function now supports IPv6. (Contributed by Charles-François Natali in bpo-11811 [https://bugs.python.org/issue?@action=redirect&bpo=11811].)
- New attribute `OP_CIPHER_SERVER_PREFERENCE` allows setting SSLv3 server sockets to use the server’s cipher ordering

preference rather than the client’s (bpo-13635 [https://bugs.python.org/issue?@action=redirect&bpo=13635]).

stat

The undocumented tarfile.filemode function has been moved to `stat.filemode()`. It can be used to convert a file’s mode to a string of the form ‘-rwxrwxrwx’.

(Contributed by Giampaolo Rodolà in bpo-14807 [https://bugs.python.org/issue?@action=redirect&bpo=14807].)

struct

The `struct` module now supports `ssize_t` and `size_t` via the new codes `n` and `N`, respectively. (Contributed by Antoine Pitrou in bpo-3163 [https://bugs.python.org/issue?@action=redirect&bpo=3163].)

subprocess

Command strings can now be bytes objects on posix platforms. (Contributed by Victor Stinner in bpo-8513 [https://bugs.python.org/issue?@action=redirect&bpo=8513].)

A new constant `DEVNULL` allows suppressing output in a platform-independent fashion. (Contributed by Ross Lagerwall in bpo-5870 [https://bugs.python.org/issue?@action=redirect&bpo=5870].)

sys

The `sys` module has a new `thread_info` named tuple holding information about the thread implementation (bpo-11223 [https://bugs.python.org/issue?@action=redirect&bpo=11223]).

tarfile

`tarfile` now supports `lzma` encoding via the **`lzma`** module. (Contributed by Lars Gustäbel in bpo-5689 [https://bugs.python.org/issue?@action=redirect&bpo=5689].)

tempfile

`tempfile.SpooledTemporaryFile`'s **`truncate()`** method now accepts a `size` parameter. (Contributed by Ryan Kelly in bpo-9957 [https://bugs.python.org/issue?@action=redirect&bpo=9957].)

textwrap

The **`textwrap`** module has a new **`indent()`** that makes it straightforward to add a common prefix to selected lines in a block of text (bpo-13857 [https://bugs.python.org/issue?@action=redirect&bpo=13857]).

threading

`threading.Condition`, **`threading.Semaphore`**, **`threading.BoundedSemaphore`**, **`threading.Event`**, and **`threading.Timer`**, all of which used to be factory functions returning a class instance, are now classes and may be subclassed. (Contributed by Éric Araujo in bpo-10968 [https://bugs.python.org/issue?@action=redirect&bpo=10968].)

The **`threading.Thread`** constructor now accepts a `daemon` keyword argument to override the default behavior of inheriting the `daemon` flag value from the parent thread (bpo-6064 [https://bugs.python.org/issue?@action=redirect&bpo=6064]).

The formerly private function `_thread.get_ident` is now available as the public function **`threading.get_ident()`**. This eliminates several cases of direct access to the `_thread` module in the stdlib. Third party code that used `_thread.get_ident` should likewise be changed to use the new public interface.

time

The **PEP 418** [https://peps.python.org/pep-0418/] added new functions to the `time` module:

- `get_clock_info()`: Get information on a clock.
- `monotonic()`: Monotonic clock (cannot go backward), not affected by system clock updates.
- `perf_counter()`: Performance counter with the highest available resolution to measure a short duration.
- `process_time()`: Sum of the system and user CPU time of the current process.

Other new functions:

- `clock_getres()`, `clock_gettime()` and `clock_settime()` functions with `CLOCK_xxx` constants. (Contributed by Victor Stinner in bpo-10278 [https://bugs.python.org/issue?@action=redirect&bpo=10278].)

To improve cross platform consistency, `sleep()` now raises a `ValueError` when passed a negative sleep value. Previously this was an error on posix, but produced an infinite sleep on Windows.

types

Add a new `types.MappingProxyType` class: Read-only proxy of a mapping. (bpo-14386 [https://bugs.python.org/issue?@action=redirect&bpo=14386])

The new functions `types.new_class()` and `types.prepare_class()` provide support for **PEP 3115** [https://peps.python.org/pep-3115/] compliant dynamic type creation. (bpo-14588 [https://bugs.python.org/issue?@action=redirect&bpo=14588])

unittest

`assertRaises()`, `assertRaisesRegex()`, `assertWarns()`, and `assertWarnsRegex()` now accept a keyword argument *msg* when used as context managers. (Contributed by Ezio Melotti and Winston Ewert in bpo-10775 [https://bugs.python.org/issue?@action=redirect&bpo=10775].)

`unittest.TestCase.run()` now returns the `TestResult` object.

urllib

The `Request` class, now accepts a *method* argument used by `get_method()` to determine what HTTP method should be used. For example, this will send a `'HEAD'` request:

```
>>> urlopen(Request('https://www.python.org', method='HEAD'))
```

(bpo-1673007 [https://bugs.python.org/issue?@action=redirect&bpo=1673007])

webbrowser

The `webbrowser` module supports more “browsers”: Google Chrome (named **chrome**, **chromium**, **chrome-browser** or **chromium-browser** depending on the version and operating system), and the generic launchers **xdg-open**, from the FreeDesktop.org project, and **gvfs-open**, which is the default URI handler for GNOME 3. (The former contributed by Arnaud Calmettes in bpo-13620 [https://bugs.python.org/issue?@action=redirect&bpo=13620], the latter by

Matthias Klose in bpo-14493 [https://bugs.python.org/issue?@action=redirect&bpo=14493].)

xml.etree.ElementTree

The **`xml.etree.ElementTree`** module now imports its C accelerator by default; there is no longer a need to explicitly import **`xml.etree.cElementTree`** (this module stays for backwards compatibility, but is now deprecated). In addition, the `iter` family of methods of **`Element`** has been optimized (rewritten in C). The module's documentation has also been greatly improved with added examples and a more detailed reference.

zlib

New attribute **`zlib.Decompress.eof`** makes it possible to distinguish between a properly formed compressed stream and an incomplete or truncated one. (Contributed by Nadeem Vawda in bpo-12646 [https://bugs.python.org/issue?@action=redirect&bpo=12646].)

New attribute **`zlib.ZLIB_RUNTIME_VERSION`** reports the version string of the underlying `zlib` library that is loaded at runtime. (Contributed by Torsten Landschoff in bpo-12306 [https://bugs.python.org/issue?@action=redirect&bpo=12306].)

Optimizations

Major performance enhancements have been added:

- Thanks to **PEP 393** [https://peps.python.org/pep-0393/], some operations on Unicode strings have been optimized:
    - the memory footprint is divided by 2 to 4 depending on the text

  - encode an ASCII string to UTF-8 doesn’t need to encode characters anymore, the UTF-8 representation is shared with the ASCII representation
  - the UTF-8 encoder has been optimized
  - repeating a single ASCII letter and getting a substring of an ASCII string is 4 times faster

- UTF-8 is now 2x to 4x faster. UTF-16 encoding is now up to 10x faster.

  (Contributed by Serhiy Storchaka, bpo-14624 [https://bugs.python.org/issue?@action=redirect&bpo=14624], bpo-14738 [https://bugs.python.org/issue?@action=redirect&bpo=14738] and bpo-15026 [https://bugs.python.org/issue?@action=redirect&bpo=15026].)

Build and C API Changes

Changes to Python’s build process and to the C API include:

- New **PEP 3118** [https://peps.python.org/pep-3118/] related function:
  - `PyMemoryView_FromMemory()`
- **PEP 393** [https://peps.python.org/pep-0393/] added new Unicode types, macros and functions:
  - High-level API:
    - `PyUnicode_CopyCharacters()`
    - `PyUnicode_FindChar()`
    - `PyUnicode_GetLength()`, `PyUnicode_GET_LENGTH`
    - `PyUnicode_New()`
    - `PyUnicode_Substring()`
    - `PyUnicode_ReadChar()`, `PyUnicode_WriteChar()`
  - Low-level API:
    - `Py_UCS1`, `Py_UCS2`, `Py_UCS4` types

    - PyASCIIObject and PyCompactUnicodeObject structures
    - PyUnicode_READY
    - PyUnicode_FromKindAndData()
    - PyUnicode_AsUCS4(), PyUnicode_AsUCS4Copy()
    - PyUnicode_DATA, PyUnicode_1BYTE_DATA, PyUnicode_2BYTE_DATA, PyUnicode_4BYTE_DATA
    - PyUnicode_KIND with **PyUnicode_Kind** enum: **PyUnicode_WCHAR_KIND**, PyUnicode_1BYTE_KIND, PyUnicode_2BYTE_KIND, PyUnicode_4BYTE_KIND
    - PyUnicode_READ, PyUnicode_READ_CHAR, PyUnicode_WRITE
    - PyUnicode_MAX_CHAR_VALUE
- PyArg_ParseTuple now accepts a bytearray for the `c` format (bpo-12380 [https://bugs.python.org/issue?@action=redirect&bpo=12380]).

Deprecated

Unsupported Operating Systems

OS/2 and VMS are no longer supported due to the lack of a maintainer.

Windows 2000 and Windows platforms which set `COMSPEC` to `command.com` are no longer supported due to maintenance burden.

OSF support, which was deprecated in 3.2, has been completely removed.

Deprecated Python modules, functions and methods

- Passing a non-empty string to `object.__format__()` is deprecated, and will produce a TypeError in Python 3.4

(bpo-9856 [https://bugs.python.org/issue?@action=redirect&bpo=9856]).

- The `unicode_internal` codec has been deprecated because of the **PEP 393** [https://peps.python.org/pep-0393/], use UTF-8, UTF-16 (`utf-16-le` or `utf-16-be`), or UTF-32 (`utf-32-le` or `utf-32-be`)
- `ftplib.FTP.nlst()` and `ftplib.FTP.dir()`: use `ftplib.FTP.mlsd()`
- `platform.popen()`: use the `subprocess` module. Check especially the Replacing Older Functions with the subprocess Module section (bpo-11377 [https://bugs.python.org/issue?@action=redirect&bpo=11377]).
- bpo-13374 [https://bugs.python.org/issue?@action=redirect&bpo=13374]: The Windows bytes API has been deprecated in the `os` module. Use Unicode filenames, instead of bytes filenames, to not depend on the ANSI code page anymore and to support any filename.
- bpo-13988 [https://bugs.python.org/issue?@action=redirect&bpo=13988]: The `xml.etree.cElementTree` module is deprecated. The accelerator is used automatically whenever available.
- The behaviour of `time.clock()` depends on the platform: use the new `time.perf_counter()` or `time.process_time()` function instead, depending on your requirements, to have a well defined behaviour.
- The `os.stat_float_times()` function is deprecated.
- `abc` module:
  - `abc.abstractproperty` has been deprecated, use `property` with `abc.abstractmethod()` instead.
  - `abc.abstractclassmethod` has been deprecated, use `classmethod` with `abc.abstractmethod()` instead.
  - `abc.abstractstaticmethod` has been deprecated, use `staticmethod` with `abc.abstractmethod()` instead.
- `importlib` package:

  - `importlib.abc.SourceLoader.path_mtime()` is now deprecated in favour of `importlib.abc.SourceLoader.path_stats()` as bytecode files now store both the modification time and size of the source file the bytecode file was compiled from.

Deprecated functions and types of the C API

The `Py_UNICODE` has been deprecated by **PEP 393** [https://peps.python.org/pep-0393/] and will be removed in Python 4. All functions using this type are deprecated:

Unicode functions and methods using `Py_UNICODE` and Py_UNICODE* types:

- **`PyUnicode_FromUnicode`**: use `PyUnicode_FromWideChar()` or `PyUnicode_FromKindAndData()`
- **`PyUnicode_AS_UNICODE`**, **`PyUnicode_AsUnicode()`**, **`PyUnicode_AsUnicodeAndSize()`**: use `PyUnicode_AsWideCharString()`
- **`PyUnicode_AS_DATA`**: use `PyUnicode_DATA` with `PyUnicode_READ` and `PyUnicode_WRITE`
- **`PyUnicode_GET_SIZE`**, **`PyUnicode_GetSize()`**: use `PyUnicode_GET_LENGTH` or `PyUnicode_GetLength()`
- **`PyUnicode_GET_DATA_SIZE`**: use `PyUnicode_GET_LENGTH(str) * PyUnicode_KIND(str)` (only work on ready strings)
- **`PyUnicode_AsUnicodeCopy()`**: use `PyUnicode_AsUCS4Copy()` or `PyUnicode_AsWideCharString()`
- **`PyUnicode_GetMax()`**

Functions and macros manipulating Py_UNICODE* strings:

- **`Py_UNICODE_strlen()`**: use `PyUnicode_GetLength()` or `PyUnicode_GET_LENGTH`

- **`Py_UNICODE_strcat()`**: use `PyUnicode_CopyCharacters()` or `PyUnicode_FromFormat()`
- **`Py_UNICODE_strcpy()`**, **`Py_UNICODE_strncpy()`**, **`Py_UNICODE_COPY()`**: use `PyUnicode_CopyCharacters()` or `PyUnicode_Substring()`
- **`Py_UNICODE_strcmp()`**: use `PyUnicode_Compare()`
- **`Py_UNICODE_strncmp()`**: use `PyUnicode_Tailmatch()`
- **`Py_UNICODE_strchr()`**, **`Py_UNICODE_strrchr()`**: use `PyUnicode_FindChar()`
- **`Py_UNICODE_FILL()`**: use `PyUnicode_Fill()`
- **`Py_UNICODE_MATCH`**

Encoders:

- **`PyUnicode_Encode()`**: use **`PyUnicode_AsEncodedObject()`**
- **`PyUnicode_EncodeUTF7()`**
- **`PyUnicode_EncodeUTF8()`**: use `PyUnicode_AsUTF8()` or `PyUnicode_AsUTF8String()`
- **`PyUnicode_EncodeUTF32()`**
- **`PyUnicode_EncodeUTF16()`**
- **`PyUnicode_EncodeUnicodeEscape()`** use `PyUnicode_AsUnicodeEscapeString()`
- **`PyUnicode_EncodeRawUnicodeEscape()`** use `PyUnicode_AsRawUnicodeEscapeString()`
- **`PyUnicode_EncodeLatin1()`**: use `PyUnicode_AsLatin1String()`
- **`PyUnicode_EncodeASCII()`**: use `PyUnicode_AsASCIIString()`
- **`PyUnicode_EncodeCharmap()`**
- **`PyUnicode_TranslateCharmap()`**
- **`PyUnicode_EncodeMBCS()`**: use `PyUnicode_AsMBCSString()` or `PyUnicode_EncodeCodePage()` (with `CP_ACP` code_page)
- **`PyUnicode_EncodeDecimal()`**, **`PyUnicode_TransformDecimalToASCII()`**

Deprecated features

The `array` module's `'u'` format code is now deprecated and will be removed in Python 4 together with the rest of the (`Py_UNICODE`) API.

Porting to Python 3.3

This section lists previously described changes and other bugfixes that may require changes to your code.

Porting Python code

- Hash randomization is enabled by default. Set the `PYTHONHASHSEED` environment variable to `0` to disable hash randomization. See also the `object.__hash__()` method.
- bpo-12326 [https://bugs.python.org/issue?@action=redirect&bpo=12326]: On Linux, sys.platform doesn't contain the major version anymore. It is now always 'linux', instead of 'linux2' or 'linux3' depending on the Linux version used to build Python. Replace sys.platform == 'linux2' with sys.platform.startswith('linux'), or directly sys.platform == 'linux' if you don't need to support older Python versions.
- bpo-13847 [https://bugs.python.org/issue?@action=redirect&bpo=13847], bpo-14180 [https://bugs.python.org/issue?@action=redirect&bpo=14180]: `time` and `datetime`: `OverflowError` is now raised instead of `ValueError` if a timestamp is out of range. `OSError` is now raised if C functions **`gmtime()`** or **`localtime()`** failed.
- The default finders used by import now utilize a cache of what is contained within a specific directory. If you create a Python source file or sourceless bytecode file, make sure to call `importlib.invalidate_caches()` to clear out the cache for the finders to notice the new file.
- `ImportError` now uses the full name of the module that was attempted to be imported. Doctests that check

ImportErrors' message will need to be updated to use the full name of the module instead of just the tail of the name.

- The *index* argument to `__import__()` now defaults to 0 instead of -1 and no longer support negative values. It was an oversight when **PEP 328** [https://peps.python.org/pep-0328/] was implemented that the default value remained -1. If you need to continue to perform a relative import followed by an absolute import, then perform the relative import using an index of 1, followed by another import using an index of 0. It is preferred, though, that you use `importlib.import_module()` rather than call `__import__()` directly.
- `__import__()` no longer allows one to use an index value other than 0 for top-level modules. E.g. `__import__('sys', level=1)` is now an error.
- Because `sys.meta_path` and `sys.path_hooks` now have finders on them by default, you will most likely want to use `list.insert()` instead of `list.append()` to add to those lists.
- Because `None` is now inserted into `sys.path_importer_cache`, if you are clearing out entries in the dictionary of paths that do not have a finder, you will need to remove keys paired with values of `None` **and** **`imp.NullImporter`** to be backwards-compatible. This will lead to extra overhead on older versions of Python that re-insert `None` into `sys.path_importer_cache` where it represents the use of implicit finders, but semantically it should not change anything.
- **`importlib.abc.Finder`** no longer specifies a `find_module()` abstract method that must be implemented. If you were relying on subclasses to implement that method, make sure to check for the method's existence first. You will probably want to check for `find_loader()` first, though, in the case of working with path entry finders.

- **`pkgutil`** has been converted to use **`importlib`** internally. This eliminates many edge cases where the old behaviour of the **PEP 302** [https://peps.python.org/pep-0302/] import emulation failed to match the behaviour of the real import system. The import emulation itself is still present, but is now deprecated. The **`pkgutil.iter_importers()`** and **`pkgutil.walk_packages()`** functions special case the standard import hooks so they are still supported even though they do not provide the non-standard `iter_modules()` method.
- A longstanding RFC-compliance bug (bpo-1079 [https://bugs.python.org/issue?@action=redirect&bpo=1079]) in the parsing done by **`email.header.decode_header()`** has been fixed. Code that uses the standard idiom to convert encoded headers into unicode (`str(make_header(decode_header(h)))`) will see no change, but code that looks at the individual tuples returned by decode_header will see that whitespace that precedes or follows `ASCII` sections is now included in the `ASCII` section. Code that builds headers using `make_header` should also continue to work without change, since `make_header` continues to add whitespace between `ASCII` and non-`ASCII` sections if it is not already present in the input strings.
- **`email.utils.formataddr()`** now does the correct content transfer encoding when passed non-`ASCII` display names. Any code that depended on the previous buggy behavior that preserved the non-`ASCII` unicode in the formatted output string will need to be changed (bpo-1690608 [https://bugs.python.org/issue?@action=redirect&bpo=1690608]).
- **`poplib.POP3.quit()`** may now raise protocol errors like all other `poplib` methods. Code that assumes `quit` does not raise **`poplib.error_proto`** errors may need to be changed if errors on `quit` are encountered by a particular

application (bpo-11291 [https://bugs.python.org/issue?@action=redirect&bpo=11291]).

- The `strict` argument to **`email.parser.Parser`**, deprecated since Python 2.4, has finally been removed.
- The deprecated method `unittest.TestCase.assertSameElements` has been removed.
- The deprecated variable `time.accept2dyear` has been removed.
- The deprecated `Context._clamp` attribute has been removed from the **`decimal`** module. It was previously replaced by the public attribute **`clamp`**. (See bpo-8540 [https://bugs.python.org/issue?@action=redirect&bpo=8540].)
- The undocumented internal helper class `SSLFakeFile` has been removed from **`smtplib`**, since its functionality has long been provided directly by **`socket.socket.makefile()`**.
- Passing a negative value to **`time.sleep()`** on Windows now raises an error instead of sleeping forever. It has always raised an error on posix.
- The `ast.__version__` constant has been removed. If you need to make decisions affected by the AST version, use **`sys.version_info`** to make the decision.
- Code that used to work around the fact that the **`threading`** module used factory functions by subclassing the private classes will need to change to subclass the now-public classes.
- The undocumented debugging machinery in the threading module has been removed, simplifying the code. This should have no effect on production code, but is mentioned here in case any application debug frameworks were interacting with it (bpo-13550 [https://bugs.python.org/issue?@action=redirect&bpo=13550]).

Porting C code

- In the course of changes to the buffer API the undocumented **`smalltable`** member of the **`Py_buffer`** structure has been removed and the layout of the **`PyMemoryViewObject`** has changed.

All extensions relying on the relevant parts in `memoryobject.h` or `object.h` must be rebuilt.

- Due to PEP 393, the **`Py_UNICODE`** type and all functions using this type are deprecated (but will stay available for at least five years). If you were using low-level Unicode APIs to construct and access unicode objects and you want to benefit of the memory footprint reduction provided by **PEP 393** [https://peps.python.org/pep-0393/], you have to convert your code to the new Unicode API.

  However, if you only have been using high-level functions such as **`PyUnicode_Concat()`**, **`PyUnicode_Join()`** or **`PyUnicode_FromFormat()`**, your code will automatically take advantage of the new unicode representations.

- **`PyImport_GetMagicNumber()`** now returns `-1` upon failure.

- As a negative value for the *level* argument to **`__import__()`** is no longer valid, the same now holds for **`PyImport_ImportModuleLevel()`**. This also means that the value of *level* used by **`PyImport_ImportModuleEx()`** is now `0` instead of `-1`.

Building C extensions

- The range of possible file names for C extensions has been narrowed. Very rarely used spellings have been suppressed: under POSIX, files named `xxxmodule.so`, `xxxmodule.abi3.so` and `xxxmodule.cpython-*.so` are no longer recognized as implementing the `xxx` module. If you had been generating such files, you have to switch to the other spellings (i.e., remove the `module` string from the file names).

(implemented in bpo-14040 [https://bugs.python.org/issue?@action=redirect&bpo=14040].)

Command Line Switch Changes

- The -Q command-line flag and related artifacts have been removed. Code checking sys.flags.division_warning will need updating.

  (bpo-10998 [https://bugs.python.org/issue?@action=redirect&bpo=10998], contributed by Éric Araujo.)

- When **python** is started with **`-S`**, `import site` will no longer add site-specific paths to the module search paths. In previous versions, it did.

  (bpo-11591 [https://bugs.python.org/issue?@action=redirect&bpo=11591], contributed by Carl Meyer with editions by Éric Araujo.)

What’s New In Python 3.2

Author:: Raymond Hettinger

This article explains the new features in Python 3.2 as compared to 3.1. Python 3.2 was released on February 20, 2011. It focuses on a few highlights and gives a few examples. For full details, see the Misc/NEWS [https://github.com/python/cpython/blob/076ca6c3c8df3030307e548d9be792ce3c1c6eea/Misc/NEWS] file.

See also

PEP 392 [https://peps.python.org/pep-0392/] - Python 3.2 Release Schedule

PEP 384: Defining a Stable ABI

In the past, extension modules built for one Python version were often not usable with other Python versions. Particularly on Windows, every feature release of Python required rebuilding all extension modules that one wanted to use. This requirement was the result of the free access to Python interpreter internals that extension modules could use.

With Python 3.2, an alternative approach becomes available: extension modules which restrict themselves to a limited API (by defining Py_LIMITED_API) cannot use many of the internals, but are constrained to a set of API functions that are promised to be stable for several releases. As a consequence, extension modules built for 3.2 in that mode will also work with 3.3, 3.4, and so on. Extension modules that make use of details of memory structures can still be built, but will need to be recompiled for every feature release.

See also

PEP 384 [https://peps.python.org/pep-0384/] - Defining a Stable ABI
PEP written by Martin von Löwis.

PEP 389: Argparse Command Line Parsing Module

A new module for command line parsing, `argparse`, was introduced to overcome the limitations of `optparse` which did not provide support for positional arguments (not just options), subcommands, required options and other common patterns of specifying and validating options.

This module has already had widespread success in the community as a third-party module. Being more fully featured than its predecessor, the `argparse` module is now the preferred module for command-line processing. The older module is still being kept available because of the substantial amount of legacy code that depends on it.

Here’s an annotated example parser showing features like limiting results to a set of choices, specifying a *metavar* in the help screen, validating that one or more positional arguments is present, and making a required option:

```
import argparse
parser = argparse.ArgumentParser(
            description = 'Manage servers',         # main
description for help
            epilog = 'Tested on Solaris and Linux') # displayed
after help
parser.add_argument('action',                       # argument
name
            choices = ['deploy', 'start', 'stop'],  # three
allowed values
            help = 'action on each target')         # help msg
parser.add_argument('targets',
            metavar = 'HOSTNAME',                   # var name
used in help msg
            nargs = '+',                            # require
one or more targets
            help = 'url for target machines')       # help msg
explanation
parser.add_argument('-u', '--user',                 # -u or --
user option
            required = True,                        # make it a
required argument
            help = 'login as user')
```

Example of calling the parser on a command string:

```
>>> cmd = 'deploy sneezy.example.com sleepy.example.com -u
skycaptain'
>>> result = parser.parse_args(cmd.split())
>>> result.action
'deploy'
>>> result.targets
['sneezy.example.com', 'sleepy.example.com']
>>> result.user
'skycaptain'
```

Example of the parser’s automatically generated help:

```
>>> parser.parse_args('-h'.split())

usage: manage_cloud.py [-h] -u USER
                       {deploy,start,stop} HOSTNAME [HOSTNAME
...]

Manage servers
```

```
positional arguments:
  {deploy,start,stop}   action on each target
  HOSTNAME              url for target machines

optional arguments:
  -h, --help            show this help message and exit
  -u USER, --user USER  login as user

Tested on Solaris and Linux
```

An especially nice argparse feature is the ability to define subparsers, each with their own argument patterns and help displays:

```
import argparse
parser = argparse.ArgumentParser(prog='HELM')
subparsers = parser.add_subparsers()

parser_l = subparsers.add_parser('launch', help='Launch
Control')   # first subgroup
parser_l.add_argument('-m', '--missiles', action='store_true')
parser_l.add_argument('-t', '--torpedos', action='store_true')

parser_m = subparsers.add_parser('move', help='Move Vessel',
# second subgroup
                                 aliases=('steer', 'turn'))
# equivalent names
parser_m.add_argument('-c', '--course', type=int,
required=True)
parser_m.add_argument('-s', '--speed', type=int, default=0)
```

```
$ ./helm.py --help                         # top level help
(launch and move)
$ ./helm.py launch --help                  # help for launch
options
$ ./helm.py launch --missiles              # set missiles=True
and torpedos=False
$ ./helm.py steer --course 180 --speed 5   # set movement
parameters
```

See also

PEP 389 [https://peps.python.org/pep-0389/] - New Command Line Parsing Module

PEP written by Steven Bethard.

Migrating optparse code to argparse for details on the differences from `optparse`.

PEP 391: Dictionary Based Configuration for Logging

The `logging` module provided two kinds of configuration, one style with function calls for each option or another style driven by an external file saved in a `configparser` format. Those options did not provide the flexibility to create configurations from JSON or YAML files, nor did they support incremental configuration, which is needed for specifying logger options from a command line.

To support a more flexible style, the module now offers `logging.config.dictConfig()` for specifying logging configuration with plain Python dictionaries. The configuration options include formatters, handlers, filters, and loggers. Here's a working example of a configuration dictionary:

```
{"version": 1,
 "formatters": {"brief": {"format": "%(levelname)-8s: %
(name)-15s: %(message)s"},
                "full": {"format": "%(asctime)s %(name)-15s %
(levelname)-8s %(message)s"}
                },
 "handlers": {"console": {
                   "class": "logging.StreamHandler",
                   "formatter": "brief",
                   "level": "INFO",
                   "stream": "ext://sys.stdout"},
              "console_priority": {
                   "class": "logging.StreamHandler",
                   "formatter": "full",
                   "level": "ERROR",
                   "stream": "ext://sys.stderr"}
              },
 "root": {"level": "DEBUG", "handlers": ["console",
"console_priority"]}}
```

If that dictionary is stored in a file called `conf.json`, it can be loaded and called with code like this:

```
>>> import json, logging.config
>>> with open('conf.json') as f:
...     conf = json.load(f)
...
>>> logging.config.dictConfig(conf)
>>> logging.info("Transaction completed normally")
INFO    : root           : Transaction completed normally
>>> logging.critical("Abnormal termination")
2011-02-17 11:14:36,694 root            CRITICAL Abnormal
termination
```

See also

PEP 391 [https://peps.python.org/pep-0391/] - Dictionary Based Configuration for Logging
PEP written by Vinay Sajip.

PEP 3148: The `concurrent.futures` module

Code for creating and managing concurrency is being collected in a new top-level namespace, *concurrent*. Its first member is a *futures* package which provides a uniform high-level interface for managing threads and processes.

The design for `concurrent.futures` was inspired by the *java.util.concurrent* package. In that model, a running call and its result are represented by a `Future` object that abstracts features common to threads, processes, and remote procedure calls. That object supports status checks (running or done), timeouts, cancellations, adding callbacks, and access to results or exceptions.

The primary offering of the new module is a pair of executor classes for launching and managing calls. The goal of the executors is to make it easier to use existing tools for making parallel calls. They save the effort needed to setup a pool of resources, launch the calls, create a results queue, add time-out handling, and limit the total number of threads, processes, or remote procedure calls.

Ideally, each application should share a single executor across multiple components so that process and thread limits can be centrally managed. This solves the design challenge that arises when each component has its own competing strategy for resource management.

Both classes share a common interface with three methods: `submit()` for scheduling a callable and returning a `Future` object; `map()` for scheduling many asynchronous calls at a time, and `shutdown()` for freeing resources. The class is a context manager and can be used in a `with` statement to assure that resources are automatically released when currently pending futures are done executing.

A simple of example of `ThreadPoolExecutor` is a launch of four parallel threads for copying files:

```
import concurrent.futures, shutil
with concurrent.futures.ThreadPoolExecutor(max_workers=4) as e:
    e.submit(shutil.copy, 'src1.txt', 'dest1.txt')
    e.submit(shutil.copy, 'src2.txt', 'dest2.txt')
    e.submit(shutil.copy, 'src3.txt', 'dest3.txt')
    e.submit(shutil.copy, 'src3.txt', 'dest4.txt')
```

See also

PEP 3148 [https://peps.python.org/pep-3148/] - Futures – Execute Computations Asynchronously
PEP written by Brian Quinlan.

Code for Threaded Parallel URL reads, an example using threads to fetch multiple web pages in parallel.

Code for computing prime numbers in parallel, an example demonstrating `ProcessPoolExecutor`.

PEP 3147: PYC Repository Directories

Python’s scheme for caching bytecode in *.pyc* files did not work well in environments with multiple Python interpreters. If one interpreter encountered a cached file created by another interpreter, it would recompile the source and overwrite the cached file, thus losing the benefits of caching.

The issue of “pyc fights” has become more pronounced as it has become commonplace for Linux distributions to ship with multiple versions of Python. These conflicts also arise with CPython alternatives such as Unladen Swallow.

To solve this problem, Python’s import machinery has been extended to use distinct filenames for each interpreter. Instead of Python 3.2 and Python 3.3 and Unladen Swallow each competing for a file called “mymodule.pyc”, they will now look for “mymodule.cpython-32.pyc”, “mymodule.cpython-33.pyc”, and “mymodule.unladen10.pyc”. And to prevent all of these new files from cluttering source directories, the *pyc* files are now collected in a “__pycache__” directory stored under the package directory.

Aside from the filenames and target directories, the new scheme has a few aspects that are visible to the programmer:

- Imported modules now have a `__cached__` attribute which stores the name of the actual file that was imported:

  ```
  >>> import collections
  >>> collections.__cached__
  'c:/py32/lib/__pycache__/collections.cpython-32.pyc'
  ```

- The tag that is unique to each interpreter is accessible from the **`imp`** module:

  ```
  >>> import imp
  >>> imp.get_tag()
  'cpython-32'
  ```

- Scripts that try to deduce source filename from the imported file now need to be smarter. It is no longer sufficient to simply strip the “c” from a “.pyc” filename. Instead, use the new functions in the **`imp`** module:

  ```
  >>>
  imp.source_from_cache('c:/py32/lib/__pycache__/collections.
  cpython-32.pyc')
  'c:/py32/lib/collections.py'
  >>> imp.cache_from_source('c:/py32/lib/collections.py')
  'c:/py32/lib/__pycache__/collections.cpython-32.pyc'
  ```

- The `py_compile` and `compileall` modules have been updated to reflect the new naming convention and target directory. The command-line invocation of *compileall* has new options: `-i` for specifying a list of files and directories to compile and `-b` which causes bytecode files to be written to their legacy location rather than *__pycache__*.

- The **`importlib.abc`** module has been updated with new abstract base classes for loading bytecode files. The obsolete ABCs, **`PyLoader`** and **`PyPycLoader`**, have been deprecated (instructions on how to stay Python 3.1 compatible are included with the documentation).

See also

PEP 3147 [https://peps.python.org/pep-3147/] - PYC Repository Directories
PEP written by Barry Warsaw.

PEP 3149: ABI Version Tagged .so Files

The PYC repository directory allows multiple bytecode cache files to be co-located. This PEP implements a similar mechanism for shared object files by giving them a common directory and distinct names for each version.

The common directory is “pyshared” and the file names are made distinct by identifying the Python implementation (such as CPython, PyPy, Jython, etc.), the major and minor version numbers, and optional build flags (such as “d” for debug, “m” for pymalloc, “u” for wide-unicode). For an arbitrary package “foo”, you may see these files when the distribution package is installed:

```
/usr/share/pyshared/foo.cpython-32m.so
/usr/share/pyshared/foo.cpython-33md.so
```

In Python itself, the tags are accessible from functions in the `sysconfig` module:

```
>>> import sysconfig
>>> sysconfig.get_config_var('SOABI')       # find the version
tag
'cpython-32mu'
>>> sysconfig.get_config_var('EXT_SUFFIX')  # find the full
filename extension
'.cpython-32mu.so'
```

See also

PEP 3149 [https://peps.python.org/pep-3149/] - ABI Version Tagged .so Files
PEP written by Barry Warsaw.

PEP 3333: Python Web Server Gateway Interface v1.0.1

This informational PEP clarifies how bytes/text issues are to be handled by the WSGI protocol. The challenge is that string handling in Python 3 is most conveniently handled with the `str` type even though the HTTP protocol is itself bytes oriented.

The PEP differentiates so-called *native strings* that are used for request/response headers and metadata versus *byte strings* which are used for the bodies of requests and responses.

The *native strings* are always of type `str` but are restricted to code points between *U+0000* through *U+00FF* which are translatable to bytes using *Latin-1* encoding. These strings are used for the keys and values in the environment dictionary and for response headers and statuses in the **`start_response()`** function. They must follow **RFC 2616** [https://datatracker.ietf.org/doc/html/rfc2616.html] with respect to encoding. That is, they must either be *ISO-8859-1* characters or use **RFC 2047** [https://datatracker.ietf.org/doc/html/rfc2047.html] MIME encoding.

For developers porting WSGI applications from Python 2, here are the salient points:

- If the app already used strings for headers in Python 2, no change is needed.
- If instead, the app encoded output headers or decoded input headers, then the headers will need to be re-encoded to Latin-1. For example, an output header encoded in utf-8 was using `h.encode('utf-8')` now needs to convert from bytes to native strings using `h.encode('utf-8').decode('latin-1')`.
- Values yielded by an application or sent using the **`write()`** method must be byte strings. The **`start_response()`** function and environ must use native strings. The two cannot be mixed.

For server implementers writing CGI-to-WSGI pathways or other CGI-style protocols, the users must to be able access the environment using native strings even though the underlying platform may have a different convention. To bridge this gap, the `wsgiref` module has a new function, `wsgiref.handlers.read_environ()` for transcoding CGI variables from `os.environ` into native strings and returning a new dictionary.

See also

PEP 3333 [https://peps.python.org/pep-3333/] - Python Web Server Gateway Interface v1.0.1

PEP written by Phillip Eby.

Other Language Changes

Some smaller changes made to the core Python language are:

- String formatting for `format()` and `str.format()` gained new capabilities for the format character **#**. Previously, for integers in binary, octal, or hexadecimal, it caused the output to be prefixed with ‘0b’, ‘0o’, or ‘0x’ respectively. Now it can also handle floats, complex, and Decimal, causing the output to always have a decimal point even when no digits follow it.

  ```
  >>> format(20, '#o')
  '0o24'
  >>> format(12.34, '#5.0f')
  '  12.'
  ```

  (Suggested by Mark Dickinson and implemented by Eric Smith in bpo-7094 [https://bugs.python.org/issue?@action=redirect&bpo=7094].)

- There is also a new `str.format_map()` method that extends the capabilities of the existing `str.format()` method by accepting arbitrary mapping objects. This new method makes it possible to use string formatting with any of Python’s many dictionary-like objects such as `defaultdict`, `Shelf`, `ConfigParser`, or `dbm`. It is also useful with custom `dict` subclasses that normalize keys before look-up or that supply a `__missing__()` method for unknown keys:

  ```
  >>> import shelve
  >>> d = shelve.open('tmp.shl')
  >>> 'The {project_name} status is {status} as of
  ```

```
{date}'.format_map(d)
'The testing project status is green as of February 15,
2011'

>>> class LowerCasedDict(dict):
...     def __getitem__(self, key):
...         return dict.__getitem__(self, key.lower())
...
>>> lcd = LowerCasedDict(part='widgets', quantity=10)
>>> 'There are {QUANTITY} {Part} in stock'.format_map(lcd)
'There are 10 widgets in stock'

>>> class PlaceholderDict(dict):
...     def __missing__(self, key):
...         return '<{}>'.format(key)
...
>>> 'Hello {name}, welcome to
{location}'.format_map(PlaceholderDict())
'Hello <name>, welcome to <location>'
```

(Suggested by Raymond Hettinger and implemented by Eric Smith in bpo-6081 [https://bugs.python.org/issue?@action=redirect&bpo=6081].)

- The interpreter can now be started with a quiet option, `-q`, to prevent the copyright and version information from being displayed in the interactive mode. The option can be introspected using the **`sys.flags`** attribute:

  ```
  $ python -q
  >>> sys.flags
  sys.flags(debug=0, division_warning=0, inspect=0,
  interactive=0,
  optimize=0, dont_write_bytecode=0, no_user_site=0,
  no_site=0,
  ignore_environment=0, verbose=0, bytes_warning=0, quiet=1)
  ```

  (Contributed by Marcin Wojdyr in bpo-1772833 [https://bugs.python.org/issue?@action=redirect&bpo=1772833]).

- The **`hasattr()`** function works by calling **`getattr()`** and detecting whether an exception is raised. This technique allows it to detect methods created dynamically by **`__getattr__()`** or **`__getattribute__()`** which would otherwise be absent from the class dictionary. Formerly, *hasattr* would catch any exception, possibly masking genuine errors. Now, *hasattr* has been tightened to only catch **`AttributeError`** and let other exceptions pass through:

  ```
  >>> class A:
  ...     @property
  ...     def f(self):
  ...         return 1 // 0
  ...
  >>> a = A()
  >>> hasattr(a, 'f')
  Traceback (most recent call last):
    ...
  ZeroDivisionError: integer division or modulo by zero
  ```

  (Discovered by Yury Selivanov and fixed by Benjamin Peterson; bpo-9666 [https://bugs.python.org/issue?@action=redirect&bpo=9666].)

- The **`str()`** of a float or complex number is now the same as its **`repr()`**. Previously, the **`str()`** form was shorter but that just caused confusion and is no longer needed now that the shortest possible **`repr()`** is displayed by default:

  ```
  >>> import math
  >>> repr(math.pi)
  '3.141592653589793'
  >>> str(math.pi)
  '3.141592653589793'
  ```

  (Proposed and implemented by Mark Dickinson; bpo-9337 [https://bugs.python.org/issue?@action=redirect&bpo=9337].)

- **`memoryview`** objects now have a **`release()`** method and they also now support the context management protocol. This allows timely release of any resources that were acquired when requesting a buffer from the original object.

  ```
  >>> with memoryview(b'abcdefgh') as v:
  ...     print(v.tolist())
  [97, 98, 99, 100, 101, 102, 103, 104]
  ```

  (Added by Antoine Pitrou; bpo-9757 [https://bugs.python.org/issue?@action=redirect&bpo=9757].)

- Previously it was illegal to delete a name from the local namespace if it occurs as a free variable in a nested block:

  ```
  def outer(x):
      def inner():
          return x
      inner()
      del x
  ```

  This is now allowed. Remember that the target of an **`except`** clause is cleared, so this code which used to work with Python 2.6, raised a **`SyntaxError`** with Python 3.1 and now works again:

  ```
  def f():
      def print_error():
          print(e)
      try:
          something
      except Exception as e:
          print_error()
          # implicit "del e" here
  ```

  (See bpo-4617 [https://bugs.python.org/issue?@action=redirect&bpo=4617].)

- Struct sequence types are now subclasses of tuple. This means that C structures like those returned by `os.stat()`, `time.gmtime()`, and `sys.version_info` now work like a named tuple and now work with functions and methods that expect a tuple as an argument. This is a big step forward in making the C structures as flexible as their pure Python counterparts:

  ```
  >>> import sys
  >>> isinstance(sys.version_info, tuple)
  True
  >>> 'Version %d.%d.%d %s(%d)' % sys.version_info
  'Version 3.2.0 final(0)'
  ```

  (Suggested by Arfrever Frehtes Taifersar Arahesis and implemented by Benjamin Peterson in bpo-8413 [https://bugs.python.org/issue?@action=redirect&bpo=8413].)

- Warnings are now easier to control using the `PYTHONWARNINGS` environment variable as an alternative to using `-W` at the command line:

  ```
  $ export
  PYTHONWARNINGS='ignore::RuntimeWarning::,once::UnicodeWarni
  ng::'
  ```

  (Suggested by Barry Warsaw and implemented by Philip Jenvey in bpo-7301 [https://bugs.python.org/issue?@action=redirect&bpo=7301].)

- A new warning category, `ResourceWarning`, has been added. It is emitted when potential issues with resource consumption or cleanup are detected. It is silenced by default in normal release builds but can be enabled through the means provided by the `warnings` module, or on the command line.

A **`ResourceWarning`** is issued at interpreter shutdown if the **`gc.garbage`** list isn't empty, and if **`gc.DEBUG_UNCOLLECTABLE`** is set, all uncollectable objects are printed. This is meant to make the programmer aware that their code contains object finalization issues.

A **`ResourceWarning`** is also issued when a file object is destroyed without having been explicitly closed. While the deallocator for such object ensures it closes the underlying operating system resource (usually, a file descriptor), the delay in deallocating the object could produce various issues, especially under Windows. Here is an example of enabling the warning from the command line:

```
$ python -q -Wdefault
>>> f = open("foo", "wb")
>>> del f
__main__:1: ResourceWarning: unclosed file
<_io.BufferedWriter name='foo'>
```

(Added by Antoine Pitrou and Georg Brandl in bpo-10093 [https://bugs.python.org/issue?@action=redirect&bpo=10093] and bpo-477863 [https://bugs.python.org/issue?@action=redirect&bpo=477863].)

- **`range`** objects now support *index* and *count* methods. This is part of an effort to make more objects fully implement the **`collections.Sequence`** abstract base class. As a result, the language will have a more uniform API. In addition, **`range`** objects now support slicing and negative indices, even with values larger than **`sys.maxsize`**. This makes *range* more interoperable with lists:

  ```
  >>> range(0, 100, 2).count(10)
  1
  >>> range(0, 100, 2).index(10)
  ```

```
5
>>> range(0, 100, 2)[5]
10
>>> range(0, 100, 2)[0:5]
range(0, 10, 2)
```

(Contributed by Daniel Stutzbach in bpo-9213 [https://bugs.python.org/issue?@action=redirect&bpo=9213], by Alexander Belopolsky in bpo-2690 [https://bugs.python.org/issue?@action=redirect&bpo=2690], and by Nick Coghlan in bpo-10889 [https://bugs.python.org/issue?@action=redirect&bpo=10889].)

- The **`callable()`** builtin function from Py2.x was resurrected. It provides a concise, readable alternative to using an abstract base class in an expression like `isinstance(x, collections.Callable)`:

  ```
  >>> callable(max)
  True
  >>> callable(20)
  False
  ```

  (See bpo-10518 [https://bugs.python.org/issue?@action=redirect&bpo=10518].)

- Python’s import mechanism can now load modules installed in directories with non-ASCII characters in the path name. This solved an aggravating problem with home directories for users with non-ASCII characters in their usernames.

(Required extensive work by Victor Stinner in bpo-9425 [https://bugs.python.org/issue?@action=redirect&bpo=9425].)

New, Improved, and Deprecated Modules

Python’s standard library has undergone significant maintenance efforts and quality improvements.

The biggest news for Python 3.2 is that the `email` package, `mailbox` module, and **`nntplib`** modules now work correctly with the bytes/text model in Python 3. For the first time, there is correct handling of messages with mixed encodings.

Throughout the standard library, there has been more careful attention to encodings and text versus bytes issues. In particular, interactions with the operating system are now better able to exchange non-ASCII data using the Windows MBCS encoding, locale-aware encodings, or UTF-8.

Another significant win is the addition of substantially better support for *SSL* connections and security certificates.

In addition, more classes now implement a context manager to support convenient and reliable resource clean-up using a `with` statement.

email

The usability of the `email` package in Python 3 has been mostly fixed by the extensive efforts of R. David Murray. The problem was that emails are typically read and stored in the form of `bytes` rather than `str` text, and they may contain multiple encodings within a single email. So, the email package had to be extended to parse and generate email messages in bytes format.

- New functions `message_from_bytes()` and `message_from_binary_file()`, and new classes `BytesFeedParser` and `BytesParser` allow binary message data to be parsed into model objects.

- Given bytes input to the model, `get_payload()` will by default decode a message body that has a *Content-Transfer-Encoding* of *8bit* using the charset specified in the MIME headers and return the resulting string.
- Given bytes input to the model, `Generator` will convert message bodies that have a *Content-Transfer-Encoding* of *8bit* to instead have a *7bit Content-Transfer-Encoding*.

  Headers with unencoded non-ASCII bytes are deemed to be **RFC 2047** [https://datatracker.ietf.org/doc/html/rfc2047.html]-encoded using the *unknown-8bit* character set.
- A new class `BytesGenerator` produces bytes as output, preserving any unchanged non-ASCII data that was present in the input used to build the model, including message bodies with a *Content-Transfer-Encoding* of *8bit*.
- The `smtplib` `SMTP` class now accepts a byte string for the *msg* argument to the `sendmail()` method, and a new method, `send_message()` accepts a `Message` object and can optionally obtain the *from_addr* and *to_addrs* addresses directly from the object.

(Proposed and implemented by R. David Murray, bpo-4661 [https://bugs.python.org/issue?@action=redirect&bpo=4661] and bpo-10321 [https://bugs.python.org/issue?@action=redirect&bpo=10321].)

elementtree

The `xml.etree.ElementTree` package and its **`xml.etree.cElementTree`** counterpart have been updated to version 1.3.

Several new and useful functions and methods have been added:

- `xml.etree.ElementTree.fromstringlist()` which builds an XML document from a sequence of fragments
- `xml.etree.ElementTree.register_namespace()` for registering a global namespace prefix
- `xml.etree.ElementTree.tostringlist()` for string representation including all sublists
- `xml.etree.ElementTree.Element.extend()` for appending a sequence of zero or more elements
- `xml.etree.ElementTree.Element.iterfind()` searches an element and subelements
- `xml.etree.ElementTree.Element.itertext()` creates a text iterator over an element and its subelements
- `xml.etree.ElementTree.TreeBuilder.end()` closes the current element
- `xml.etree.ElementTree.TreeBuilder.doctype()` handles a doctype declaration

Two methods have been deprecated:

- **`xml.etree.ElementTree.getchildren()`** use `list(elem)` instead.
- **`xml.etree.ElementTree.getiterator()`** use `Element.iter` instead.

For details of the update, see Introducing ElementTree [https://web.archive.org/web/20200703234532/http://effbot.org/zone/elementtree-13-intro.htm] on Fredrik Lundh’s website.

(Contributed by Florent Xicluna and Fredrik Lundh, bpo-6472 [https://bugs.python.org/issue?@action=redirect&bpo=6472].)

functools

- The `functools` module includes a new decorator for caching function calls. `functools.lru_cache()` can save repeated queries to an external resource whenever the results are expected to be the same.

For example, adding a caching decorator to a database query function can save database accesses for popular searches:

```
>>> import functools
>>> @functools.lru_cache(maxsize=300)
... def get_phone_number(name):
...     c = conn.cursor()
...     c.execute('SELECT phonenumber FROM phonelist WHERE
name=?', (name,))
...     return c.fetchone()[0]

>>> for name in user_requests:
...     get_phone_number(name)        # cached lookup
```

To help with choosing an effective cache size, the wrapped function is instrumented for tracking cache statistics:

```
>>> get_phone_number.cache_info()
CacheInfo(hits=4805, misses=980, maxsize=300, currsize=300)
```

If the phonelist table gets updated, the outdated contents of the cache can be cleared with:

```
>>> get_phone_number.cache_clear()
```

(Contributed by Raymond Hettinger and incorporating design ideas from Jim Baker, Miki Tebeka, and Nick Coghlan; see recipe 498245 [https://code.activestate.com/recipes/498245-lru-and-lfu-cache-decorators/], recipe 577479 [https://code.activestate.com/recipes/577479-simple-caching-decorator/], bpo-10586 [https://bugs.python.org/issue?@action=redirect&bpo=10586], and bpo-10593 [https://bugs.python.org/issue?@action=redirect&bpo=10593].)

- The `functools.wraps()` decorator now adds a `__wrapped__` attribute pointing to the original callable function. This allows wrapped functions to be introspected. It also copies **`__annotations__`** if defined. And now it also gracefully skips over missing attributes such as **`__doc__`** which might not be defined for the wrapped callable.

In the above example, the cache can be removed by recovering the original function:

```
>>> get_phone_number = get_phone_number.__wrapped__    # 
uncached function
```

(By Nick Coghlan and Terrence Cole; bpo-9567 [https://bugs.python.org/issue?@action=redirect&bpo=9567], bpo-3445 [https://bugs.python.org/issue?@action=redirect&bpo=3445], and bpo-8814 [https://bugs.python.org/issue?@action=redirect&bpo=8814].)

- To help write classes with rich comparison methods, a new decorator **`functools.total_ordering()`** will use existing equality and inequality methods to fill in the remaining methods.

  For example, supplying *__eq__* and *__lt__* will enable **`total_ordering()`** to fill-in *__le__*, *__gt__* and *__ge__*:

```
@total_ordering
class Student:
    def __eq__(self, other):
        return ((self.lastname.lower(), 
self.firstname.lower()) ==
                (other.lastname.lower(), 
other.firstname.lower()))

    def __lt__(self, other):
        return ((self.lastname.lower(), 
self.firstname.lower()) <
                (other.lastname.lower(), 
other.firstname.lower()))
```

With the *total_ordering* decorator, the remaining comparison methods are filled in automatically.

(Contributed by Raymond Hettinger.)

- To aid in porting programs from Python 2, the **`functools.cmp_to_key()`** function converts an old-style comparison function to modern key function:

  ```
  >>> # locale-aware sort order
  >>> sorted(iterable, key=cmp_to_key(locale.strcoll))
  ```

  For sorting examples and a brief sorting tutorial, see the Sorting HowTo [https://wiki.python.org/moin/HowTo/Sorting/] tutorial.

  (Contributed by Raymond Hettinger.)

itertools

- The **`itertools`** module has a new **`accumulate()`** function modeled on APL's *scan* operator and Numpy's *accumulate* function:

  ```
  >>> from itertools import accumulate
  >>> list(accumulate([8, 2, 50]))
  [8, 10, 60]

  >>> prob_dist = [0.1, 0.4, 0.2, 0.3]
  >>> list(accumulate(prob_dist))      # cumulative
  probability distribution
  [0.1, 0.5, 0.7, 1.0]
  ```

  For an example using **`accumulate()`**, see the examples for the random module.

  (Contributed by Raymond Hettinger and incorporating design suggestions from Mark Dickinson.)

collections

- The `collections.Counter` class now has two forms of in-place subtraction, the existing -= operator for saturating subtraction [https://en.wikipedia.org/wiki/Saturation_arithmetic] and the new `subtract()` method for regular subtraction. The former is suitable for multisets [https://en.wikipedia.org/wiki/Multiset] which only have positive counts, and the latter is more suitable for use cases that allow negative counts:

  ```
  >>> from collections import Counter
  >>> tally = Counter(dogs=5, cats=3)
  >>> tally -= Counter(dogs=2, cats=8)    # saturating
  subtraction
  >>> tally
  Counter({'dogs': 3})
  ```

  ```
  >>> tally = Counter(dogs=5, cats=3)
  >>> tally.subtract(dogs=2, cats=8)      # regular
  subtraction
  >>> tally
  Counter({'dogs': 3, 'cats': -5})
  ```

  (Contributed by Raymond Hettinger.)

- The `collections.OrderedDict` class has a new method `move_to_end()` which takes an existing key and moves it to either the first or last position in the ordered sequence.

  The default is to move an item to the last position. This is equivalent of renewing an entry with `od[k] = od.pop(k)`.

  A fast move-to-end operation is useful for resequencing entries. For example, an ordered dictionary can be used to track order of access by aging entries from the oldest to the most recently accessed.

```
>>> from collections import OrderedDict
>>> d = OrderedDict.fromkeys(['a', 'b', 'X', 'd', 'e'])
>>> list(d)
['a', 'b', 'X', 'd', 'e']
>>> d.move_to_end('X')
>>> list(d)
['a', 'b', 'd', 'e', 'X']
```

(Contributed by Raymond Hettinger.)

- The `collections.deque` class grew two new methods `count()` and `reverse()` that make them more substitutable for `list` objects:

  ```
  >>> from collections import deque
  >>> d = deque('simsalabim')
  >>> d.count('s')
  2
  >>> d.reverse()
  >>> d
  deque(['m', 'i', 'b', 'a', 'l', 'a', 's', 'm', 'i', 's'])
  ```

  (Contributed by Raymond Hettinger.)

threading

The `threading` module has a new `Barrier` synchronization class for making multiple threads wait until all of them have reached a common barrier point. Barriers are useful for making sure that a task with multiple preconditions does not run until all of the predecessor tasks are complete.

Barriers can work with an arbitrary number of threads. This is a generalization of a Rendezvous [https://en.wikipedia.org/wiki/Synchronous_rendezvous] which is defined for only two threads.

Implemented as a two-phase cyclic barrier, `Barrier` objects are suitable for use in loops. The separate *filling* and *draining* phases assure that all threads get released (drained) before any one of them can loop back and re-enter the barrier. The barrier fully resets after each cycle.

Example of using barriers:

```
from threading import Barrier, Thread

def get_votes(site):
    ballots = conduct_election(site)
    all_polls_closed.wait()        # do not count until all
polls are closed
    totals = summarize(ballots)
    publish(site, totals)

all_polls_closed = Barrier(len(sites))
for site in sites:
    Thread(target=get_votes, args=(site,)).start()
```

In this example, the barrier enforces a rule that votes cannot be counted at any polling site until all polls are closed. Notice how a solution with a barrier is similar to one with **`threading.Thread.join()`**, but the threads stay alive and continue to do work (summarizing ballots) after the barrier point is crossed.

If any of the predecessor tasks can hang or be delayed, a barrier can be created with an optional *timeout* parameter. Then if the timeout period elapses before all the predecessor tasks reach the barrier point, all waiting threads are released and a **`BrokenBarrierError`** exception is raised:

```
def get_votes(site):
    ballots = conduct_election(site)
    try:
        all_polls_closed.wait(timeout=midnight - time.now())
    except BrokenBarrierError:
        lockbox = seal_ballots(ballots)
        queue.put(lockbox)
    else:
        totals = summarize(ballots)
        publish(site, totals)
```

In this example, the barrier enforces a more robust rule. If some election sites do not finish before midnight, the barrier times-out and the ballots are sealed and deposited in a queue for later handling.

See Barrier Synchronization Patterns [https://osl.cs.illinois.edu/media/papers/karmani-2009-barrier_synchronization_pattern.pdf] for more examples of how barriers can be used in parallel computing. Also, there is a simple but thorough explanation of barriers in The Little Book of Semaphores [https://greenteapress.com/semaphores/LittleBookOfSemaphores.pdf], *section 3.6*.

(Contributed by Kristján Valur Jónsson with an API review by Jeffrey Yasskin in bpo-8777 [https://bugs.python.org/issue?@action=redirect&bpo=8777].)

datetime and time

- The **`datetime`** module has a new type **`timezone`** that implements the **`tzinfo`** interface by returning a fixed UTC offset and timezone name. This makes it easier to create timezone-aware datetime objects:

  ```
  >>> from datetime import datetime, timezone

  >>> datetime.now(timezone.utc)
  datetime.datetime(2010, 12, 8, 21, 4, 2, 923754,
  tzinfo=datetime.timezone.utc)

  >>> datetime.strptime("01/01/2000 12:00 +0000", "%m/%d/%Y
  %H:%M %z")
  ```

```
datetime.datetime(2000, 1, 1, 12, 0,
tzinfo=datetime.timezone.utc)
```

- Also, **`timedelta`** objects can now be multiplied by **`float`** and divided by **`float`** and **`int`** objects. And **`timedelta`** objects can now divide one another.
- The **`datetime.date.strftime()`** method is no longer restricted to years after 1900. The new supported year range is from 1000 to 9999 inclusive.
- Whenever a two-digit year is used in a time tuple, the interpretation has been governed by **`time.accept2dyear`**. The default is `True` which means that for a two-digit year, the century is guessed according to the POSIX rules governing the `%y` strptime format.

  Starting with Py3.2, use of the century guessing heuristic will emit a **`DeprecationWarning`**. Instead, it is recommended that **`time.accept2dyear`** be set to `False` so that large date ranges can be used without guesswork:

  ```
  >>> import time, warnings
  >>> warnings.resetwarnings()      # remove the default
  warning filters

  >>> time.accept2dyear = True      # guess whether 11 means
  11 or 2011
  >>> time.asctime((11, 1, 1, 12, 34, 56, 4, 1, 0))
  Warning (from warnings module):
    ...
  DeprecationWarning: Century info guessed for a 2-digit
  year.
  'Fri Jan  1 12:34:56 2011'

  >>> time.accept2dyear = False     # use the full range of
  allowable dates
  >>> time.asctime((11, 1, 1, 12, 34, 56, 4, 1, 0))
  'Fri Jan  1 12:34:56 11'
  ```

> Several functions now have significantly expanded date ranges. When **`time.accept2dyear`** is false, the **`time.asctime()`** function will accept any year that fits in a C int, while the **`time.mktime()`** and **`time.strftime()`** functions will accept the full range supported by the corresponding operating system functions.

(Contributed by Alexander Belopolsky and Victor Stinner in bpo-1289118 [https://bugs.python.org/issue?@action=redirect&bpo=1289118], bpo-5094 [https://bugs.python.org/issue?@action=redirect&bpo=5094], bpo-6641 [https://bugs.python.org/issue?@action=redirect&bpo=6641], bpo-2706 [https://bugs.python.org/issue?@action=redirect&bpo=2706], bpo-1777412 [https://bugs.python.org/issue?@action=redirect&bpo=1777412], bpo-8013 [https://bugs.python.org/issue?@action=redirect&bpo=8013], and bpo-10827 [https://bugs.python.org/issue?@action=redirect&bpo=10827].)

math

The **`math`** module has been updated with six new functions inspired by the C99 standard.

The **`isfinite()`** function provides a reliable and fast way to detect special values. It returns `True` for regular numbers and `False` for *Nan* or *Infinity*:

```
>>> from math import isfinite
>>> [isfinite(x) for x in (123, 4.56, float('Nan'),
float('Inf'))]
[True, True, False, False]
```

The **`expm1()`** function computes `e**x-1` for small values of *x* without incurring the loss of precision that usually accompanies the subtraction of nearly equal quantities:

```
>>> from math import expm1
>>> expm1(0.013671875)   # more accurate way to compute e**x-1
```

```
for a small x
0.013765762467652909
```

The **`erf()`** function computes a probability integral or Gaussian error function [https://en.wikipedia.org/wiki/Error_function]. The complementary error function, **`erfc()`**, is `1 - erf(x)`:

```
>>> from math import erf, erfc, sqrt
>>> erf(1.0/sqrt(2.0))   # portion of normal distribution
within 1 standard deviation
0.682689492137086
>>> erfc(1.0/sqrt(2.0))  # portion of normal distribution
outside 1 standard deviation
0.31731050786291404
>>> erf(1.0/sqrt(2.0)) + erfc(1.0/sqrt(2.0))
1.0
```

The **`gamma()`** function is a continuous extension of the factorial function. See https://en.wikipedia.org/wiki/Gamma_function for details. Because the function is related to factorials, it grows large even for small values of *x*, so there is also a **`lgamma()`** function for computing the natural logarithm of the gamma function:

```
>>> from math import gamma, lgamma
>>> gamma(7.0)           # six factorial
720.0
>>> lgamma(801.0)        # log(800 factorial)
4551.950730698041
```

(Contributed by Mark Dickinson.)

abc

The **`abc`** module now supports **`abstractclassmethod()`** and **`abstractstaticmethod()`**.

These tools make it possible to define an abstract base class that requires a particular `classmethod()` or `staticmethod()` to be implemented:

```
class Temperature(metaclass=abc.ABCMeta):
    @abc.abstractclassmethod
    def from_fahrenheit(cls, t):
        ...
    @abc.abstractclassmethod
    def from_celsius(cls, t):
        ...
```

(Patch submitted by Daniel Urban; bpo-5867 [https://bugs.python.org/issue?@action=redirect&bpo=5867].)

io

The `io.BytesIO` has a new method, `getbuffer()`, which provides functionality similar to `memoryview()`. It creates an editable view of the data without making a copy. The buffer's random access and support for slice notation are well-suited to in-place editing:

```
>>> REC_LEN, LOC_START, LOC_LEN = 34, 7, 11

>>> def change_location(buffer, record_number, location):
...     start = record_number * REC_LEN + LOC_START
...     buffer[start: start+LOC_LEN] = location

>>> import io

>>> byte_stream = io.BytesIO(
...     b'G3805  storeroom  Main chassis    '
...     b'X7899  shipping   Reserve cog     '
...     b'L6988  receiving  Primary sprocket'
... )
>>> buffer = byte_stream.getbuffer()
>>> change_location(buffer, 1, b'warehouse  ')
>>> change_location(buffer, 0, b'showroom   ')
>>> print(byte_stream.getvalue())
```

```
b'G3805  showroom   Main chassis    '
b'X7899  warehouse  Reserve cog     '
b'L6988  receiving  Primary sprocket'
```

(Contributed by Antoine Pitrou in bpo-5506 [https://bugs.python.org/issue?@action=redirect&bpo=5506].)

reprlib

When writing a `__repr__()` method for a custom container, it is easy to forget to handle the case where a member refers back to the container itself. Python's builtin objects such as `list` and `set` handle self-reference by displaying "…" in the recursive part of the representation string.

To help write such `__repr__()` methods, the `reprlib` module has a new decorator, `recursive_repr()`, for detecting recursive calls to `__repr__()` and substituting a placeholder string instead:

```
>>> class MyList(list):
...     @recursive_repr()
...     def __repr__(self):
...         return '<' + '|'.join(map(repr, self)) + '>'
...
>>> m = MyList('abc')
>>> m.append(m)
>>> m.append('x')
>>> print(m)
<'a'|'b'|'c'|...|'x'>
```

(Contributed by Raymond Hettinger in bpo-9826 [https://bugs.python.org/issue?@action=redirect&bpo=9826] and bpo-9840 [https://bugs.python.org/issue?@action=redirect&bpo=9840].)

logging

In addition to dictionary-based configuration described above, the **`logging`** package has many other improvements.

The logging documentation has been augmented by a basic tutorial, an advanced tutorial, and a cookbook of logging recipes. These documents are the fastest way to learn about logging.

The **`logging.basicConfig()`** set-up function gained a *style* argument to support three different types of string formatting. It defaults to "%" for traditional %-formatting, can be set to "{" for the new **`str.format()`** style, or can be set to "$" for the shell-style formatting provided by **`string.Template`**. The following three configurations are equivalent:

```
>>> from logging import basicConfig
>>> basicConfig(style='%', format="%(name)s -> %(levelname)s: %
(message)s")
>>> basicConfig(style='{', format="{name} -> {levelname}
{message}")
>>> basicConfig(style='$', format="$name -> $levelname:
$message")
```

If no configuration is set-up before a logging event occurs, there is now a default configuration using a **`StreamHandler`** directed to **`sys.stderr`** for events of `WARNING` level or higher. Formerly, an event occurring before a configuration was set-up would either raise an exception or silently drop the event depending on the value of **`logging.raiseExceptions`**. The new default handler is stored in **`logging.lastResort`**.

The use of filters has been simplified. Instead of creating a **`Filter`** object, the predicate can be any Python callable that returns `True` or `False`.

There were a number of other improvements that add flexibility and simplify configuration. See the module documentation for a full listing of changes in Python 3.2.

csv

The `csv` module now supports a new dialect, `unix_dialect`, which applies quoting for all fields and a traditional Unix style with `'\n'` as the line terminator. The registered dialect name is `unix`.

The `csv.DictWriter` has a new method, `writeheader()` for writing-out an initial row to document the field names:

```
>>> import csv, sys
>>> w = csv.DictWriter(sys.stdout, ['name', 'dept'],
dialect='unix')
>>> w.writeheader()
"name","dept"
>>> w.writerows([
...     {'name': 'tom', 'dept': 'accounting'},
...     {'name': 'susan', 'dept': 'Salesl'}])
"tom","accounting"
"susan","sales"
```

(New dialect suggested by Jay Talbot in bpo-5975 [https://bugs.python.org/issue?@action=redirect&bpo=5975], and the new method suggested by Ed Abraham in bpo-1537721 [https://bugs.python.org/issue?@action=redirect&bpo=1537721].)

contextlib

There is a new and slightly mind-blowing tool `ContextDecorator` that is helpful for creating a context manager that does double duty as a function decorator.

As a convenience, this new functionality is used by `contextmanager()` so that no extra effort is needed to support both roles.

The basic idea is that both context managers and function decorators can be used for pre-action and post-action wrappers. Context managers wrap a group of statements using a `with` statement, and function decorators wrap a group of statements enclosed in a function. So, occasionally there is a need to write a pre-action or post-action wrapper that can be used in either role.

For example, it is sometimes useful to wrap functions or groups of statements with a logger that can track the time of entry and time of exit. Rather than writing both a function decorator and a context manager for the task, the `contextmanager()` provides both capabilities in a single definition:

```
from contextlib import contextmanager
import logging

logging.basicConfig(level=logging.INFO)

@contextmanager
def track_entry_and_exit(name):
    logging.info('Entering: %s', name)
    yield
    logging.info('Exiting: %s', name)
```

Formerly, this would have only been usable as a context manager:

```
with track_entry_and_exit('widget loader'):
    print('Some time consuming activity goes here')
    load_widget()
```

Now, it can be used as a decorator as well:

```
@track_entry_and_exit('widget loader')
def activity():
    print('Some time consuming activity goes here')
    load_widget()
```

Trying to fulfill two roles at once places some limitations on the technique. Context managers normally have the flexibility to return an argument usable by a `with` statement, but there is no parallel for function decorators.

In the above example, there is not a clean way for the *track_entry_and_exit* context manager to return a logging instance for use in the body of enclosed statements.

(Contributed by Michael Foord in bpo-9110 [https://bugs.python.org/issue?@action=redirect&bpo=9110].)

decimal and fractions

Mark Dickinson crafted an elegant and efficient scheme for assuring that different numeric datatypes will have the same hash value whenever their actual values are equal (bpo-8188 [https://bugs.python.org/issue?@action=redirect&bpo=8188]):

```
assert hash(Fraction(3, 2)) == hash(1.5) == \
       hash(Decimal("1.5")) == hash(complex(1.5, 0))
```

Some of the hashing details are exposed through a new attribute, `sys.hash_info`, which describes the bit width of the hash value, the prime modulus, the hash values for *infinity* and *nan*, and the multiplier used for the imaginary part of a number:

```
>>> sys.hash_info
sys.hash_info(width=64, modulus=2305843009213693951,
inf=314159, nan=0, imag=1000003)
```

An early decision to limit the interoperability of various numeric types has been relaxed. It is still unsupported (and ill-advised) to have implicit mixing in arithmetic expressions such as `Decimal('1.1') + float('1.1')` because the latter loses information in the process of constructing the binary float.

However, since existing floating-point value can be converted losslessly to either a decimal or rational representation, it makes sense to add them to the constructor and to support mixed-type comparisons.

- The **`decimal.Decimal`** constructor now accepts **`float`** objects directly so there in no longer a need to use the **`from_float()`** method (bpo-8257 [https://bugs.python.org/issue?@action=redirect&bpo=8257]).
- Mixed type comparisons are now fully supported so that **`Decimal`** objects can be directly compared with **`float`** and **`fractions.Fraction`** (bpo-2531 [https://bugs.python.org/issue?@action=redirect&bpo=2531] and bpo-8188 [https://bugs.python.org/issue?@action=redirect&bpo=8188]).

Similar changes were made to **`fractions.Fraction`** so that the **`from_float()`** and **`from_decimal()`** methods are no longer needed (bpo-8294 [https://bugs.python.org/issue?@action=redirect&bpo=8294]):

```
>>> from decimal import Decimal
>>> from fractions import Fraction
>>> Decimal(1.1)
Decimal('1.100000000000000088817841970012523233890533447265625'
)
>>> Fraction(1.1)
Fraction(2476979795053773, 2251799813685248)
```

Another useful change for the **`decimal`** module is that the **`Context.clamp`** attribute is now public. This is useful in creating contexts that correspond to the decimal interchange formats specified in IEEE 754 (see bpo-8540 [https://bugs.python.org/issue?@action=redirect&bpo=8540]).

(Contributed by Mark Dickinson and Raymond Hettinger.)

ftp

The `ftplib.FTP` class now supports the context management protocol to unconditionally consume `socket.error` exceptions and to close the FTP connection when done:

```
>>> from ftplib import FTP
>>> with FTP("ftp1.at.proftpd.org") as ftp:
        ftp.login()
        ftp.dir()

'230 Anonymous login ok, restrictions apply.'
dr-xr-xr-x   9 ftp      ftp           154 May  6 10:43 .
dr-xr-xr-x   9 ftp      ftp           154 May  6 10:43 ..
dr-xr-xr-x   5 ftp      ftp          4096 May  6 10:43 CentOS
dr-xr-xr-x   3 ftp      ftp            18 Jul 10  2008 Fedora
```

Other file-like objects such as `mmap.mmap` and `fileinput.input()` also grew auto-closing context managers:

```
with fileinput.input(files=('log1.txt', 'log2.txt')) as f:
    for line in f:
        process(line)
```

(Contributed by Tarek Ziadé and Giampaolo Rodolà in bpo-4972 [https://bugs.python.org/issue?@action=redirect&bpo=4972], and by Georg Brandl in bpo-8046 [https://bugs.python.org/issue?@action=redirect&bpo=8046] and bpo-1286 [https://bugs.python.org/issue?@action=redirect&bpo=1286].)

The `FTP_TLS` class now accepts a *context* parameter, which is a `ssl.SSLContext` object allowing bundling SSL configuration options, certificates and private keys into a single (potentially long-lived) structure.

(Contributed by Giampaolo Rodolà; bpo-8806 [https://bugs.python.org/issue?@action=redirect&bpo=8806].)

popen

The **`os.popen()`** and **`subprocess.Popen()`** functions now support **`with`** statements for auto-closing of the file descriptors.

(Contributed by Antoine Pitrou and Brian Curtin in bpo-7461 [https://bugs.python.org/issue?@action=redirect&bpo=7461] and bpo-10554 [https://bugs.python.org/issue?@action=redirect&bpo=10554].)

select

The **`select`** module now exposes a new, constant attribute, **`PIPE_BUF`**, which gives the minimum number of bytes which are guaranteed not to block when **`select.select()`** says a pipe is ready for writing.

```
>>> import select
>>> select.PIPE_BUF
512
```

(Available on Unix systems. Patch by Sébastien Sablé in bpo-9862 [https://bugs.python.org/issue?@action=redirect&bpo=9862])

gzip and zipfile

`gzip.GzipFile` now implements the **`io.BufferedIOBase`** abstract base class (except for `truncate()`). It also has a **`peek()`** method and supports unseekable as well as zero-padded file objects.

The **`gzip`** module also gains the **`compress()`** and **`decompress()`** functions for easier in-memory compression and decompression. Keep in mind that text needs to be encoded as **`bytes`** before compressing and decompressing:

```
>>> import gzip
>>> s = 'Three shall be the number thou shalt count, '
>>> s += 'and the number of the counting shall be three'
>>> b = s.encode()                        # convert to utf-8
>>> len(b)
```

```
89
>>> c = gzip.compress(b)
>>> len(c)
77
>>> gzip.decompress(c).decode()[:42]      # decompress and
convert to text
'Three shall be the number thou shalt count'
```

(Contributed by Anand B. Pillai in bpo-3488 [https://bugs.python.org/issue?@action=redirect&bpo=3488]; and by Antoine Pitrou, Nir Aides and Brian Curtin in bpo-9962 [https://bugs.python.org/issue?@action=redirect&bpo=9962], bpo-1675951 [https://bugs.python.org/issue?@action=redirect&bpo=1675951], bpo-7471 [https://bugs.python.org/issue?@action=redirect&bpo=7471] and bpo-2846 [https://bugs.python.org/issue?@action=redirect&bpo=2846].)

Also, the `zipfile.ZipExtFile` class was reworked internally to represent files stored inside an archive. The new implementation is significantly faster and can be wrapped in an `io.BufferedReader` object for more speedups. It also solves an issue where interleaved calls to *read* and *readline* gave the wrong results.

(Patch submitted by Nir Aides in bpo-7610 [https://bugs.python.org/issue?@action=redirect&bpo=7610].)

tarfile

The `TarFile` class can now be used as a context manager. In addition, its `add()` method has a new option, *filter*, that controls which files are added to the archive and allows the file metadata to be edited.

The new *filter* option replaces the older, less flexible *exclude* parameter which is now deprecated. If specified, the optional *filter* parameter needs to be a keyword argument. The user-supplied filter function accepts a `TarInfo` object and returns an updated **`TarInfo`** object, or if it wants the file to be excluded, the function can return `None`:

```
>>> import tarfile, glob

>>> def myfilter(tarinfo):
...     if tarinfo.isfile():             # only save real files
...         tarinfo.uname = 'monty'      # redact the user name
...         return tarinfo

>>> with tarfile.open(name='myarchive.tar.gz', mode='w:gz') as
tf:
...     for filename in glob.glob('*.txt'):
...         tf.add(filename, filter=myfilter)
...     tf.list()
-rw-r--r-- monty/501        902 2011-01-26 17:59:11
annotations.txt
-rw-r--r-- monty/501        123 2011-01-26 17:59:11
general_questions.txt
-rw-r--r-- monty/501       3514 2011-01-26 17:59:11 prion.txt
-rw-r--r-- monty/501        124 2011-01-26 17:59:11 py_todo.txt
-rw-r--r-- monty/501       1399 2011-01-26 17:59:11
semaphore_notes.txt
```

(Proposed by Tarek Ziadé and implemented by Lars Gustäbel in bpo-6856 [https://bugs.python.org/issue?@action=redirect&bpo=6856].)

hashlib

The **`hashlib`** module has two new constant attributes listing the hashing algorithms guaranteed to be present in all implementations and those available on the current implementation:

```
>>> import hashlib

>>> hashlib.algorithms_guaranteed
{'sha1', 'sha224', 'sha384', 'sha256', 'sha512', 'md5'}
```

```
>>> hashlib.algorithms_available
{'md2', 'SHA256', 'SHA512', 'dsaWithSHA', 'mdc2', 'SHA224',
'MD4', 'sha256',
'sha512', 'ripemd160', 'SHA1', 'MDC2', 'SHA', 'SHA384', 'MD2',
'ecdsa-with-SHA1','md4', 'md5', 'sha1', 'DSA-SHA', 'sha224',
'dsaEncryption', 'DSA', 'RIPEMD160', 'sha', 'MD5', 'sha384'}
```

(Suggested by Carl Chenet in bpo-7418 [https://bugs.python.org/issue?@action=redirect&bpo=7418].)

ast

The **`ast`** module has a wonderful a general-purpose tool for safely evaluating expression strings using the Python literal syntax. The **`ast.literal_eval()`** function serves as a secure alternative to the builtin **`eval()`** function which is easily abused. Python 3.2 adds **`bytes`** and **`set`** literals to the list of supported types: strings, bytes, numbers, tuples, lists, dicts, sets, booleans, and `None`.

```
>>> from ast import literal_eval

>>> request = "{'req': 3, 'func': 'pow', 'args': (2, 0.5)}"
>>> literal_eval(request)
{'args': (2, 0.5), 'req': 3, 'func': 'pow'}

>>> request = "os.system('do something harmful')"
>>> literal_eval(request)
Traceback (most recent call last):
  ...
ValueError: malformed node or string: <_ast.Call object at
0x101739a10>
```

(Implemented by Benjamin Peterson and Georg Brandl.)

os

Different operating systems use various encodings for filenames and environment variables. The **`os`** module provides two new functions, **`fsencode()`** and **`fsdecode()`**, for encoding and decoding filenames:

```
>>> import os
>>> filename = 'Sehenswürdigkeiten'
>>> os.fsencode(filename)
b'Sehensw\xc3\xbcrdigkeiten'
```

Some operating systems allow direct access to encoded bytes in the environment. If so, the **`os.supports_bytes_environ`** constant will be true.

For direct access to encoded environment variables (if available), use the new **`os.getenvb()`** function or use **`os.environb`** which is a bytes version of **`os.environ`**.

(Contributed by Victor Stinner.)

shutil

The **`shutil.copytree()`** function has two new options:

- *ignore_dangling_symlinks*: when `symlinks=False` so that the function copies a file pointed to by a symlink, not the symlink itself. This option will silence the error raised if the file doesn’t exist.
- *copy_function*: is a callable that will be used to copy files. **`shutil.copy2()`** is used by default.

(Contributed by Tarek Ziadé.)

In addition, the **`shutil`** module now supports archiving operations for zipfiles, uncompressed tarfiles, gzipped tarfiles, and bzipped tarfiles. And there are functions for registering additional archiving file formats (such as xz compressed tarfiles or custom formats).

The principal functions are `make_archive()` and `unpack_archive()`. By default, both operate on the current directory (which can be set by `os.chdir()`) and on any sub-directories. The archive filename needs to be specified with a full pathname. The archiving step is non-destructive (the original files are left unchanged).

```
>>> import shutil, pprint

>>> os.chdir('mydata')  # change to the source directory
>>> f = shutil.make_archive('/var/backup/mydata',
...                          'zip')      # archive the current
directory
>>> f                                    # show the name of
archive
'/var/backup/mydata.zip'
>>> os.chdir('tmp')                      # change to an
unpacking
>>> shutil.unpack_archive('/var/backup/mydata.zip')  # recover
the data

>>> pprint.pprint(shutil.get_archive_formats())  # display
known formats
[('bztar', "bzip2'ed tar-file"),
 ('gztar', "gzip'ed tar-file"),
 ('tar', 'uncompressed tar file'),
 ('zip', 'ZIP file')]

>>> shutil.register_archive_format(     # register a new
archive format
...     name='xz',
...     function=xz.compress,           # callable archiving
function
...     extra_args=[('level', 8)],      # arguments to the
function
...     description='xz compression'
... )
```

(Contributed by Tarek Ziadé.)

sqlite3

The **`sqlite3`** module was updated to pysqlite version 2.6.0. It has two new capabilities.

- The **`sqlite3.Connection.in_transit`** attribute is true if there is an active transaction for uncommitted changes.
- The **`sqlite3.Connection.enable_load_extension()`** and **`sqlite3.Connection.load_extension()`** methods allows you to load SQLite extensions from ".so" files. One well-known extension is the fulltext-search extension distributed with SQLite.

(Contributed by R. David Murray and Shashwat Anand; bpo-8845 [https://bugs.python.org/issue?@action=redirect&bpo=8845].)

html

A new **`html`** module was introduced with only a single function, **`escape()`**, which is used for escaping reserved characters from HTML markup:

```
>>> import html
>>> html.escape('x > 2 && x < 7')
'x &gt; 2 &amp;&amp; x &lt; 7'
```

socket

The **`socket`** module has two new improvements.

- Socket objects now have a **`detach()`** method which puts the socket into closed state without actually closing the underlying file descriptor. The latter can then be reused for other purposes. (Added by Antoine Pitrou; bpo-8524 [https://bugs.python.org/issue?@action=redirect&bpo=8524].)
- **`socket.create_connection()`** now supports the context management protocol to unconditionally consume **`socket.error`** exceptions and to close the socket when

done. (Contributed by Giampaolo Rodolà; bpo-9794 [https://bugs.python.org/issue?@action=redirect&bpo=9794].)

ssl

The `ssl` module added a number of features to satisfy common requirements for secure (encrypted, authenticated) internet connections:

- A new class, `SSLContext`, serves as a container for persistent SSL data, such as protocol settings, certificates, private keys, and various other options. It includes a `wrap_socket()` for creating an SSL socket from an SSL context.
- A new function, `ssl.match_hostname()`, supports server identity verification for higher-level protocols by implementing the rules of HTTPS (from **RFC 2818** [https://datatracker.ietf.org/doc/html/rfc2818.html]) which are also suitable for other protocols.
- The `ssl.wrap_socket()` constructor function now takes a *ciphers* argument. The *ciphers* string lists the allowed encryption algorithms using the format described in the OpenSSL documentation [https://docs.openssl.org/1.0.2/man1/ciphers/#cipher-list-format].
- When linked against recent versions of OpenSSL, the `ssl` module now supports the Server Name Indication extension to the TLS protocol, allowing multiple “virtual hosts” using different certificates on a single IP port. This extension is only supported in client mode, and is activated by passing the *server_hostname* argument to `ssl.SSLContext.wrap_socket()`.
- Various options have been added to the `ssl` module, such as `OP_NO_SSLv2` which disables the insecure and obsolete SSLv2 protocol.
- The extension now loads all the OpenSSL ciphers and digest algorithms. If some SSL certificates cannot be

verified, they are reported as an “unknown algorithm” error.

- The version of OpenSSL being used is now accessible using the module attributes `ssl.OPENSSL_VERSION` (a string), `ssl.OPENSSL_VERSION_INFO` (a 5-tuple), and `ssl.OPENSSL_VERSION_NUMBER` (an integer).

(Contributed by Antoine Pitrou in bpo-8850 [https://bugs.python.org/issue?@action=redirect&bpo=8850], bpo-1589 [https://bugs.python.org/issue?@action=redirect&bpo=1589], bpo-8322 [https://bugs.python.org/issue?@action=redirect&bpo=8322], bpo-5639 [https://bugs.python.org/issue?@action=redirect&bpo=5639], bpo-4870 [https://bugs.python.org/issue?@action=redirect&bpo=4870], bpo-8484 [https://bugs.python.org/issue?@action=redirect&bpo=8484], and bpo-8321 [https://bugs.python.org/issue?@action=redirect&bpo=8321].)

nntp

The `nntplib` module has a revamped implementation with better bytes and text semantics as well as more practical APIs. These improvements break compatibility with the nntplib version in Python 3.1, which was partly dysfunctional in itself.

Support for secure connections through both implicit (using `nntplib.NNTP_SSL`) and explicit (using `nntplib.NNTP.starttls()`) TLS has also been added.

(Contributed by Antoine Pitrou in bpo-9360 [https://bugs.python.org/issue?@action=redirect&bpo=9360] and Andrew Vant in bpo-1926 [https://bugs.python.org/issue?@action=redirect&bpo=1926].)

certificates

`http.client.HTTPSConnection`, `urllib.request.HTTPSHandler` and `urllib.request.urlopen()` now take optional arguments to allow for server certificate checking against a set of Certificate Authorities, as recommended in public uses of HTTPS.

(Added by Antoine Pitrou, bpo-9003 [https://bugs.python.org/issue?@action=redirect&bpo=9003].)

imaplib

Support for explicit TLS on standard IMAP4 connections has been added through the new `imaplib.IMAP4.starttls` method.

(Contributed by Lorenzo M. Catucci and Antoine Pitrou, bpo-4471 [https://bugs.python.org/issue?@action=redirect&bpo=4471].)

http.client

There were a number of small API improvements in the `http.client` module. The old-style HTTP 0.9 simple responses are no longer supported and the *strict* parameter is deprecated in all classes.

The `HTTPConnection` and `HTTPSConnection` classes now have a *source_address* parameter for a (host, port) tuple indicating where the HTTP connection is made from.

Support for certificate checking and HTTPS virtual hosts were added to `HTTPSConnection`.

The `request()` method on connection objects allowed an optional *body* argument so that a file object could be used to supply the content of the request. Conveniently, the *body* argument now also accepts an iterable object so long as it includes an explicit `Content-Length` header. This extended interface is much more flexible than before.

To establish an HTTPS connection through a proxy server, there is a new `set_tunnel()` method that sets the host and port for HTTP Connect tunneling.

To match the behavior of `http.server`, the HTTP client library now also encodes headers with ISO-8859-1 (Latin-1) encoding. It was already doing that for incoming headers, so now the behavior is consistent for both incoming and outgoing traffic. (See work by Armin Ronacher in bpo-10980 [https://bugs.python.org/issue?@action=redirect&bpo=10980].)

unittest

The unittest module has a number of improvements supporting test discovery for packages, easier experimentation at the interactive prompt, new testcase methods, improved diagnostic messages for test failures, and better method names.

- The command-line call `python -m unittest` can now accept file paths instead of module names for running specific tests (bpo-10620 [https://bugs.python.org/issue?@action=redirect&bpo=10620]). The new test discovery can find tests within packages, locating any test importable from the top-level directory. The top-level directory can be specified with the `-t` option, a pattern for matching files with `-p`, and a directory to start discovery with `-s`:

  ```
  $ python -m unittest discover -s my_proj_dir -p _test.py
  ```

  (Contributed by Michael Foord.)

- Experimentation at the interactive prompt is now easier because the `unittest.TestCase` class can now be instantiated without arguments:

```
>>> from unittest import TestCase
>>> TestCase().assertEqual(pow(2, 3), 8)
```

(Contributed by Michael Foord.)

- The **`unittest`** module has two new methods, **`assertWarns()`** and **`assertWarnsRegex()`** to verify that a given warning type is triggered by the code under test:

  ```
  with self.assertWarns(DeprecationWarning):
      legacy_function('XYZ')
  ```

  (Contributed by Antoine Pitrou, bpo-9754 [https://bugs.python.org/issue?@action=redirect&bpo=9754].)

  Another new method, **`assertCountEqual()`** is used to compare two iterables to determine if their element counts are equal (whether the same elements are present with the same number of occurrences regardless of order):

  ```
  def test_anagram(self):
      self.assertCountEqual('algorithm', 'logarithm')
  ```

  (Contributed by Raymond Hettinger.)

- A principal feature of the unittest module is an effort to produce meaningful diagnostics when a test fails. When possible, the failure is recorded along with a diff of the output. This is especially helpful for analyzing log files of failed test runs. However, since diffs can sometime be voluminous, there is a new **`maxDiff`** attribute that sets maximum length of diffs displayed.

- In addition, the method names in the module have undergone a number of clean-ups.

For example, `assertRegex()` is the new name for **`assertRegexpMatches()`** which was misnamed because the test uses `re.search()`, not `re.match()`. Other methods using regular expressions are now named using short form “Regex” in preference to “Regexp” – this matches the names used in other unittest implementations, matches Python’s old name for the `re` module, and it has unambiguous camel-casing.

(Contributed by Raymond Hettinger and implemented by Ezio Melotti.)

- To improve consistency, some long-standing method aliases are being deprecated in favor of the preferred names:

Old Name	**Preferred Name**
`assert_()`	`assertTrue()`
`assertEquals()`	`assertEqual()`
`assertNotEquals()`	`assertNotEqual()`
`assertAlmostEquals()`	`assertAlmostEqual()`
`assertNotAlmostEquals()`	`assertNotAlmostEqual()`

Likewise, the `TestCase.fail*` methods deprecated in Python 3.1 are expected to be removed in Python 3.3.

(Contributed by Ezio Melotti; bpo-9424 [https://bugs.python.org/issue?@action=redirect&bpo=9424].)

- The **`assertDictContainsSubset()`** method was deprecated because it was misimplemented with the arguments in the wrong order. This created hard-to-debug optical illusions where tests like `TestCase().assertDictContainsSubset({'a':1, 'b':2}, {'a':1})` would fail.

  (Contributed by Raymond Hettinger.)

random

The integer methods in the **`random`** module now do a better job of producing uniform distributions. Previously, they computed selections with `int(n*random())` which had a slight bias whenever *n* was not a power of two. Now, multiple selections are made from a range up to the next power of two and a selection is kept only when it falls within the range `0 <= x < n`. The functions and methods affected are **`randrange()`**, **`randint()`**, **`choice()`**, **`shuffle()`** and **`sample()`**.

(Contributed by Raymond Hettinger; bpo-9025 [https://bugs.python.org/issue?@action=redirect&bpo=9025].)

poplib

`POP3_SSL` class now accepts a *context* parameter, which is a **`ssl.SSLContext`** object allowing bundling SSL configuration options, certificates and private keys into a single (potentially long-lived) structure.

(Contributed by Giampaolo Rodolà; bpo-8807 [https://bugs.python.org/issue?@action=redirect&bpo=8807].)

asyncore

`asyncore.dispatcher` now provides a **`handle_accepted()`** method returning a `(sock, addr)` pair which is called when a connection has actually been established with a new remote endpoint. This is supposed to be used as a replacement for old **`handle_accept()`** and avoids the user to call **`accept()`** directly.

(Contributed by Giampaolo Rodolà; bpo-6706 [https://bugs.python.org/issue?@action=redirect&bpo=6706].)

tempfile

The **`tempfile`** module has a new context manager, **`TemporaryDirectory`** which provides easy deterministic cleanup of temporary directories:

```
with tempfile.TemporaryDirectory() as tmpdirname:
    print('created temporary dir:', tmpdirname)
```

(Contributed by Neil Schemenauer and Nick Coghlan; bpo-5178 [https://bugs.python.org/issue?@action=redirect&bpo=5178].)

inspect

- The **`inspect`** module has a new function **`getgeneratorstate()`** to easily identify the current state of a generator-iterator:

  ```
  >>> from inspect import getgeneratorstate
  >>> def gen():
  ...     yield 'demo'
  ...
  >>> g = gen()
  >>> getgeneratorstate(g)
  'GEN_CREATED'
  >>> next(g)
  ```

```
'demo'
>>> getgeneratorstate(g)
'GEN_SUSPENDED'
>>> next(g, None)
>>> getgeneratorstate(g)
'GEN_CLOSED'
```

(Contributed by Rodolpho Eckhardt and Nick Coghlan, bpo-10220 [https://bugs.python.org/issue?@action=redirect&bpo=10220].)

- To support lookups without the possibility of activating a dynamic attribute, the **`inspect`** module has a new function, **`getattr_static()`**. Unlike **`hasattr()`**, this is a true read-only search, guaranteed not to change state while it is searching:

```
>>> class A:
...     @property
...     def f(self):
...         print('Running')
...         return 10
...
>>> a = A()
>>> getattr(a, 'f')
Running
10
>>> inspect.getattr_static(a, 'f')
<property object at 0x1022bd788>
```

(Contributed by Michael Foord.)

pydoc

The **`pydoc`** module now provides a much-improved web server interface, as well as a new command-line option `-b` to automatically open a browser window to display that server:

```
$ pydoc3.2 -b
```

(Contributed by Ron Adam; bpo-2001 [https://bugs.python.org/issue?@action=redirect&bpo=2001].)

dis

The **`dis`** module gained two new functions for inspecting code, **`code_info()`** and **`show_code()`**. Both provide detailed code object information for the supplied function, method, source code string or code object. The former returns a string and the latter prints it:

```
>>> import dis, random
>>> dis.show_code(random.choice)
Name:              choice
Filename:
/Library/Frameworks/Python.framework/Versions/3.2/lib/python3.2
/random.py
Argument count:    2
Kw-only arguments: 0
Number of locals:  3
Stack size:        11
Flags:             OPTIMIZED, NEWLOCALS, NOFREE
Constants:
   0: 'Choose a random element from a non-empty sequence.'
   1: 'Cannot choose from an empty sequence'
Names:
   0: _randbelow
   1: len
   2: ValueError
   3: IndexError
Variable names:
   0: self
   1: seq
   2: i
```

In addition, the **`dis()`** function now accepts string arguments so that the common idiom `dis(compile(s, '', 'eval'))` can be shortened to `dis(s)`:

```
>>> dis('3*x+1 if x%2==1 else x//2')
  1           0 LOAD_NAME                0 (x)
              3 LOAD_CONST               0 (2)
              6 BINARY_MODULO
              7 LOAD_CONST               1 (1)
             10 COMPARE_OP               2 (==)
             13 POP_JUMP_IF_FALSE       28
             16 LOAD_CONST               2 (3)
             19 LOAD_NAME                0 (x)
             22 BINARY_MULTIPLY
             23 LOAD_CONST               1 (1)
             26 BINARY_ADD
             27 RETURN_VALUE
        >>   28 LOAD_NAME                0 (x)
             31 LOAD_CONST               0 (2)
             34 BINARY_FLOOR_DIVIDE
             35 RETURN_VALUE
```

Taken together, these improvements make it easier to explore how CPython is implemented and to see for yourself what the language syntax does under-the-hood.

(Contributed by Nick Coghlan in bpo-9147 [https://bugs.python.org/issue?@action=redirect&bpo=9147].)

dbm

All database modules now support the **`get()`** and **`setdefault()`** methods.

(Suggested by Ray Allen in bpo-9523 [https://bugs.python.org/issue?@action=redirect&bpo=9523].)

ctypes

A new type, `ctypes.c_ssize_t` represents the C **`ssize_t`** datatype.

site

The `site` module has three new functions useful for reporting on the details of a given Python installation.

- `getsitepackages()` lists all global site-packages directories.
- `getuserbase()` reports on the user's base directory where data can be stored.
- `getusersitepackages()` reveals the user-specific site-packages directory path.

```
>>> import site
>>> site.getsitepackages()
['/Library/Frameworks/Python.framework/Versions/3.2/lib/python3
.2/site-packages',
 '/Library/Frameworks/Python.framework/Versions/3.2/lib/site-
python',
 '/Library/Python/3.2/site-packages']
>>> site.getuserbase()
'/Users/raymondhettinger/Library/Python/3.2'
>>> site.getusersitepackages()
'/Users/raymondhettinger/Library/Python/3.2/lib/python/site-
packages'
```

Conveniently, some of site's functionality is accessible directly from the command-line:

```
$ python -m site --user-base
/Users/raymondhettinger/.local
$ python -m site --user-site
/Users/raymondhettinger/.local/lib/python3.2/site-packages
```

(Contributed by Tarek Ziadé in bpo-6693 [https://bugs.python.org/issue?@action=redirect&bpo=6693].)

sysconfig

The new `sysconfig` module makes it straightforward to discover installation paths and configuration variables that vary across platforms and installations.

The module offers access simple access functions for platform and version information:

- `get_platform()` returning values like *linux-i586* or *macosx-10.6-ppc*.
- `get_python_version()` returns a Python version string such as “3.2”.

It also provides access to the paths and variables corresponding to one of seven named schemes used by `distutils`. Those include *posix_prefix*, *posix_home*, *posix_user*, *nt*, *nt_user*, *os2*, *os2_home*:

- `get_paths()` makes a dictionary containing installation paths for the current installation scheme.
- `get_config_vars()` returns a dictionary of platform specific variables.

There is also a convenient command-line interface:

```
C:\Python32>python -m sysconfig
Platform: "win32"
Python version: "3.2"
Current installation scheme: "nt"

Paths:
        data = "C:\Python32"
        include = "C:\Python32\Include"
        platinclude = "C:\Python32\Include"
        platlib = "C:\Python32\Lib\site-packages"
        platstdlib = "C:\Python32\Lib"
        purelib = "C:\Python32\Lib\site-packages"
        scripts = "C:\Python32\Scripts"
        stdlib = "C:\Python32\Lib"

Variables:
        BINDIR = "C:\Python32"
        BINLIBDEST = "C:\Python32\Lib"
        EXE = ".exe"
        INCLUDEPY = "C:\Python32\Include"
        LIBDEST = "C:\Python32\Lib"
        SO = ".pyd"
        VERSION = "32"
        abiflags = ""
        base = "C:\Python32"
        exec_prefix = "C:\Python32"
        platbase = "C:\Python32"
        prefix = "C:\Python32"
        projectbase = "C:\Python32"
        py_version = "3.2"
        py_version_nodot = "32"
        py_version_short = "3.2"
        srcdir = "C:\Python32"
        userbase = "C:\Documents and
Settings\Raymond\Application Data\Python"
```

(Moved out of Distutils by Tarek Ziadé.)

pdb

The **`pdb`** debugger module gained a number of usability improvements:

- `pdb.py` now has a `-c` option that executes commands as given in a `.pdbrc` script file.
- A `.pdbrc` script file can contain `continue` and `next` commands that continue debugging.
- The **`Pdb`** class constructor now accepts a *nosigint* argument.
- New commands: `l(list)`, `ll(long list)` and `source` for listing source code.
- New commands: `display` and `undisplay` for showing or hiding the value of an expression if it has changed.
- New command: `interact` for starting an interactive interpreter containing the global and local names found in the current scope.
- Breakpoints can be cleared by breakpoint number.

(Contributed by Georg Brandl, Antonio Cuni and Ilya Sandler.)

configparser

The `configparser` module was modified to improve usability and predictability of the default parser and its supported INI syntax. The old **`ConfigParser`** class was removed in favor of **`SafeConfigParser`** which has in turn been renamed to `ConfigParser`. Support for inline comments is now turned off by default and section or option duplicates are not allowed in a single configuration source.

Config parsers gained a new API based on the mapping protocol:

```
>>> parser = ConfigParser()
>>> parser.read_string("""
... [DEFAULT]
... location = upper left
... visible = yes
... editable = no
... color = blue
...
... [main]
... title = Main Menu
... color = green
...
... [options]
... title = Options
... """)
>>> parser['main']['color']
'green'
>>> parser['main']['editable']
'no'
>>> section = parser['options']
>>> section['title']
'Options'
>>> section['title'] = 'Options (editable: %(editable)s)'
>>> section['title']
'Options (editable: no)'
```

The new API is implemented on top of the classical API, so custom parser subclasses should be able to use it without modifications.

The INI file structure accepted by config parsers can now be customized. Users can specify alternative option/value delimiters and comment prefixes, change the name of the *DEFAULT* section or switch the interpolation syntax.

There is support for pluggable interpolation including an additional interpolation handler **`ExtendedInterpolation`**:

```
>>> parser =
ConfigParser(interpolation=ExtendedInterpolation())
>>> parser.read_dict({'buildout': {'directory':
'/home/ambv/zope9'},
...                    'custom': {'prefix': '/usr/local'}})
>>> parser.read_string("""
... [buildout]
... parts =
...   zope9
...   instance
... find-links =
...   ${buildout:directory}/downloads/dist
...
... [zope9]
... recipe = plone.recipe.zope9install
... location = /opt/zope
...
... [instance]
... recipe = plone.recipe.zope9instance
... zope9-location = ${zope9:location}
... zope-conf = ${custom:prefix}/etc/zope.conf
... """)
>>> parser['buildout']['find-links']
'\n/home/ambv/zope9/downloads/dist'
>>> parser['instance']['zope-conf']
'/usr/local/etc/zope.conf'
```

```
>>> instance = parser['instance']
>>> instance['zope-conf']
'/usr/local/etc/zope.conf'
>>> instance['zope9-location']
'/opt/zope'
```

A number of smaller features were also introduced, like support for specifying encoding in read operations, specifying fallback values for get-functions, or reading directly from dictionaries and strings.

(All changes contributed by Łukasz Langa.)

urllib.parse

A number of usability improvements were made for the `urllib.parse` module.

The `urlparse()` function now supports IPv6 [https://en.wikipedia.org/wiki/IPv6] addresses as described in **RFC 2732** [https://datatracker.ietf.org/doc/html/rfc2732.html]:

```
>>> import urllib.parse
>>>
urllib.parse.urlparse('http://[dead:beef:cafe:5417:affe:8FA3:de
af:feed]/foo/')
ParseResult(scheme='http',
            netloc='[dead:beef:cafe:5417:affe:8FA3:deaf:feed]',
            path='/foo/',
            params='',
            query='',
            fragment='')
```

The `urldefrag()` function now returns a named tuple:

```
>>> r =
urllib.parse.urldefrag('http://python.org/about/#target')
>>> r
DefragResult(url='http://python.org/about/', fragment='target')
>>> r[0]
```

```
'http://python.org/about/'
>>> r.fragment
'target'
```

And, the **`urlencode()`** function is now much more flexible, accepting either a string or bytes type for the *query* argument. If it is a string, then the *safe*, *encoding*, and *error* parameters are sent to **`quote_plus()`** for encoding:

```
>>> urllib.parse.urlencode([
...      ('type', 'telenovela'),
...      ('name', '¿Dónde Está Elisa?')],
...      encoding='latin-1')
'type=telenovela&name=%BFD%F3nde+Est%E1+Elisa%3F'
```

As detailed in Parsing ASCII Encoded Bytes, all the **`urllib.parse`** functions now accept ASCII-encoded byte strings as input, so long as they are not mixed with regular strings. If ASCII-encoded byte strings are given as parameters, the return types will also be an ASCII-encoded byte strings:

```
>>> urllib.parse.urlparse(b'http://www.python.org:80/about/')
ParseResultBytes(scheme=b'http', netloc=b'www.python.org:80',
                 path=b'/about/', params=b'', query=b'',
fragment=b'')
```

(Work by Nick Coghlan, Dan Mahn, and Senthil Kumaran in bpo-2987 [https://bugs.python.org/issue?@action=redirect&bpo=2987], bpo-5468 [https://bugs.python.org/issue?@action=redirect&bpo=5468], and bpo-9873 [https://bugs.python.org/issue?@action=redirect&bpo=9873].)

mailbox

Thanks to a concerted effort by R. David Murray, the **`mailbox`** module has been fixed for Python 3.2. The challenge was that mailbox had been originally designed with a text interface, but email messages are best represented with `bytes` because various parts of a message may have different encodings.

The solution harnessed the `email` package’s binary support for parsing arbitrary email messages. In addition, the solution required a number of API changes.

As expected, the `add()` method for `mailbox.Mailbox` objects now accepts binary input.

`StringIO` and text file input are deprecated. Also, string input will fail early if non-ASCII characters are used. Previously it would fail when the email was processed in a later step.

There is also support for binary output. The `get_file()` method now returns a file in the binary mode (where it used to incorrectly set the file to text-mode). There is also a new `get_bytes()` method that returns a `bytes` representation of a message corresponding to a given *key*.

It is still possible to get non-binary output using the old API’s `get_string()` method, but that approach is not very useful. Instead, it is best to extract messages from a `Message` object or to load them from binary input.

(Contributed by R. David Murray, with efforts from Steffen Daode Nurpmeso and an initial patch by Victor Stinner in bpo-9124 [https://bugs.python.org/issue?@action=redirect&bpo=9124].)

turtledemo

The demonstration code for the `turtle` module was moved from the *Demo* directory to main library. It includes over a dozen sample scripts with lively displays. Being on `sys.path`, it can now be run directly from the command-line:

```
$ python -m turtledemo
```

(Moved from the Demo directory by Alexander Belopolsky in bpo-10199 [https://bugs.python.org/issue?@action=redirect&bpo=10199].)

Multi-threading

- The mechanism for serializing execution of concurrently running Python threads (generally known as the GIL or Global Interpreter Lock) has been rewritten. Among the objectives were more predictable switching intervals and reduced overhead due to lock contention and the number of ensuing system calls. The notion of a “check interval” to allow thread switches has been abandoned and replaced by an absolute duration expressed in seconds. This parameter is tunable through `sys.setswitchinterval()`. It currently defaults to 5 milliseconds.

  Additional details about the implementation can be read from a python-dev mailing-list message [https://mail.python.org/pipermail/python-dev/2009-October/093321.html] (however, “priority requests” as exposed in this message have not been kept for inclusion).

  (Contributed by Antoine Pitrou.)

- Regular and recursive locks now accept an optional *timeout* argument to their `acquire()` method. (Contributed by Antoine Pitrou; bpo-7316 [https://bugs.python.org/issue?@action=redirect&bpo=7316].)

- Similarly, `threading.Semaphore.acquire()` also gained a *timeout* argument. (Contributed by Torsten Landschoff; bpo-850728 [https://bugs.python.org/issue?@action=redirect&bpo=850728].)

- Regular and recursive lock acquisitions can now be interrupted by signals on platforms using Pthreads. This means that Python programs that deadlock while acquiring locks can be successfully killed by repeatedly sending SIGINT to the process (by pressing `Ctrl`+`C` in most shells). (Contributed by Reid Kleckner; bpo-8844 [https://bugs.python.org/issue?@action=redirect&bpo=8844].)

Optimizations

A number of small performance enhancements have been added:

- Python's peephole optimizer now recognizes patterns such `x in {1, 2, 3}` as being a test for membership in a set of constants. The optimizer recasts the **`set`** as a **`frozenset`** and stores the pre-built constant.

  Now that the speed penalty is gone, it is practical to start writing membership tests using set-notation. This style is both semantically clear and operationally fast:

  ```
  extension = name.rpartition('.')[2]
  if extension in {'xml', 'html', 'xhtml', 'css'}:
      handle(name)
  ```

  (Patch and additional tests contributed by Dave Malcolm; bpo-6690 [https://bugs.python.org/issue?@action=redirect&bpo=6690]).

- Serializing and unserializing data using the **`pickle`** module is now several times faster.

  (Contributed by Alexandre Vassalotti, Antoine Pitrou and the Unladen Swallow team in bpo-9410 [https://bugs.python.org/issue?@action=redirect&bpo=9410] and bpo-3873 [https://bugs.python.org/issue?@action=redirect&bpo=3873].)

- The Timsort algorithm [https://en.wikipedia.org/wiki/Timsort] used in **`list.sort()`** and **`sorted()`** now runs faster and uses less memory when called with a key function. Previously, every element of a list was wrapped with a temporary object that remembered the key value associated with each element. Now, two arrays of keys and values are sorted in parallel. This saves the memory consumed by the sort wrappers, and it saves time lost to delegating comparisons.

  (Patch by Daniel Stutzbach in bpo-9915 [https://bugs.python.org/issue?@action=redirect&bpo=9915].)

- JSON decoding performance is improved and memory consumption is reduced whenever the same string is repeated for multiple keys. Also, JSON encoding now uses the C speedups when the `sort_keys` argument is true.

  (Contributed by Antoine Pitrou in bpo-7451 [https://bugs.python.org/issue?@action=redirect&bpo=7451] and by Raymond Hettinger and Antoine Pitrou in bpo-10314 [https://bugs.python.org/issue?@action=redirect&bpo=10314].)

- Recursive locks (created with the **`threading.RLock()`** API) now benefit from a C implementation which makes them as fast as regular locks, and between 10x and 15x faster than their previous pure Python implementation.

  (Contributed by Antoine Pitrou; bpo-3001 [https://bugs.python.org/issue?@action=redirect&bpo=3001].)

- The fast-search algorithm in stringlib is now used by the **`split()`**, **`rsplit()`**, **`splitlines()`** and **`replace()`** methods on **`bytes`**, **`bytearray`** and **`str`** objects. Likewise, the algorithm is also used by **`rfind()`**, **`rindex()`**, **`rsplit()`** and **`rpartition()`**.

(Patch by Florent Xicluna in bpo-7622 [https://bugs.python.org/issue?@action=redirect&bpo=7622] and bpo-7462 [https://bugs.python.org/issue?@action=redirect&bpo=7462].)

- Integer to string conversions now work two “digits” at a time, reducing the number of division and modulo operations.

  (bpo-6713 [https://bugs.python.org/issue?@action=redirect&bpo=6713] by Gawain Bolton, Mark Dickinson, and Victor Stinner.)

There were several other minor optimizations. Set differencing now runs faster when one operand is much larger than the other (patch by Andress Bennetts in bpo-8685 [https://bugs.python.org/issue?@action=redirect&bpo=8685]). The `array.repeat()` method has a faster implementation (bpo-1569291 [https://bugs.python.org/issue?@action=redirect&bpo=1569291] by Alexander Belopolsky). The `BaseHTTPRequestHandler` has more efficient buffering (bpo-3709 [https://bugs.python.org/issue?@action=redirect&bpo=3709] by Andrew Schaaf). The `operator.attrgetter()` function has been sped-up (bpo-10160 [https://bugs.python.org/issue?@action=redirect&bpo=10160] by Christos Georgiou). And `ConfigParser` loads multi-line arguments a bit faster (bpo-7113 [https://bugs.python.org/issue?@action=redirect&bpo=7113] by Łukasz Langa).

Unicode

Python has been updated to Unicode 6.0.0 [https://unicode.org/versions/Unicode6.0.0/]. The update to the standard adds over 2,000 new characters including emoji [https://en.wikipedia.org/wiki/Emoji] symbols which are important for mobile phones.

In addition, the updated standard has altered the character properties for two Kannada characters (U+0CF1, U+0CF2) and one New Tai Lue numeric character (U+19DA), making the former eligible for use in identifiers while disqualifying the latter. For more information, see Unicode Character Database Changes [https://www.unicode.org/versions/Unicode6.0.0/#Database_Changes].

Codecs

Support was added for *cp720* Arabic DOS encoding (bpo-1616979 [https://bugs.python.org/issue?@action=redirect&bpo=1616979]).

MBCS encoding no longer ignores the error handler argument. In the default strict mode, it raises an **`UnicodeDecodeError`** when it encounters an undecodable byte sequence and an **`UnicodeEncodeError`** for an unencodable character.

The MBCS codec supports `'strict'` and `'ignore'` error handlers for decoding, and `'strict'` and `'replace'` for encoding.

To emulate Python3.1 MBCS encoding, select the `'ignore'` handler for decoding and the `'replace'` handler for encoding.

On Mac OS X, Python decodes command line arguments with `'utf-8'` rather than the locale encoding.

By default, **`tarfile`** uses `'utf-8'` encoding on Windows (instead of `'mbcs'`) and the `'surrogateescape'` error handler on all operating systems.

Documentation

The documentation continues to be improved.

- A table of quick links has been added to the top of lengthy sections such as Built-in Functions. In the case of **`itertools`**, the links are accompanied by tables of cheatsheet-style summaries to provide an overview and memory jog without having to read all of the docs.

- In some cases, the pure Python source code can be a helpful adjunct to the documentation, so now many modules now feature quick links to the latest version of the source code. For example, the **`functools`** module documentation has a quick link at the top labeled:

  > **Source code** Lib/functools.py [https://github.com/python/cpython/tree/3.14/Lib/functools.py].

  (Contributed by Raymond Hettinger; see rationale [https://rhettinger.wordpress.com/2011/01/28/open-your-source-more/].)

- The docs now contain more examples and recipes. In particular, **`re`** module has an extensive section, Regular Expression Examples. Likewise, the **`itertools`** module continues to be updated with new Itertools Recipes.

- The **`datetime`** module now has an auxiliary implementation in pure Python. No functionality was changed. This just provides an easier-to-read alternate implementation.

  (Contributed by Alexander Belopolsky in bpo-9528 [https://bugs.python.org/issue?@action=redirect&bpo=9528].)

- The unmaintained `Demo` directory has been removed. Some demos were integrated into the documentation, some were moved to the `Tools/demo` directory, and others were removed altogether.

(Contributed by Georg Brandl in bpo-7962 [https://bugs.python.org/issue?@action=redirect&bpo=7962].)

IDLE

- The format menu now has an option to clean source files by stripping trailing whitespace.

  (Contributed by Raymond Hettinger; bpo-5150 [https://bugs.python.org/issue?@action=redirect&bpo=5150].)

- IDLE on Mac OS X now works with both Carbon AquaTk and Cocoa AquaTk.

  (Contributed by Kevin Walzer, Ned Deily, and Ronald Oussoren; bpo-6075 [https://bugs.python.org/issue?@action=redirect&bpo=6075].)

Code Repository

In addition to the existing Subversion code repository at https://svn.python.org there is now a Mercurial [https://www.mercurial-scm.org/] repository at https://hg.python.org/.

After the 3.2 release, there are plans to switch to Mercurial as the primary repository. This distributed version control system should make it easier for members of the community to create and share external changesets. See **PEP 385** [https://peps.python.org/pep-0385/] for details.

To learn to use the new version control system, see the Quick Start [https://www.mercurial-scm.org/wiki/QuickStart] or the Guide to Mercurial Workflows [https://www.mercurial-scm.org/guide].

Build and C API Changes

Changes to Python’s build process and to the C API include:

- The *idle*, *pydoc* and *2to3* scripts are now installed with a version-specific suffix on `make altinstall` (bpo-10679 [https://bugs.python.org/issue?@action=redirect&bpo=10679]).

- The C functions that access the Unicode Database now accept and return characters from the full Unicode range, even on narrow unicode builds (Py_UNICODE_TOLOWER, Py_UNICODE_ISDECIMAL, and others). A visible difference in Python is that **`unicodedata.numeric()`** now returns the correct value for large code points, and **`repr()`** may consider more characters as printable.

  (Reported by Bupjoe Lee and fixed by Amaury Forgeot D’Arc; bpo-5127 [https://bugs.python.org/issue?@action=redirect&bpo=5127].)

- Computed gotos are now enabled by default on supported compilers (which are detected by the configure script). They can still be disabled selectively by specifying `--without-computed-gotos`.

  (Contributed by Antoine Pitrou; bpo-9203 [https://bugs.python.org/issue?@action=redirect&bpo=9203].)

- The option `--with-wctype-functions` was removed. The built-in unicode database is now used for all functions.

  (Contributed by Amaury Forgeot D’Arc; bpo-9210 [https://bugs.python.org/issue?@action=redirect&bpo=9210].)

- Hash values are now values of a new type, `Py_hash_t`, which is defined to be the same size as a pointer. Previously they were of type long, which on some 64-bit operating systems is still only 32 bits long. As a result of this fix, `set` and `dict` can now hold more than `2**32` entries on builds with 64-bit pointers (previously, they could grow to that size but their performance degraded catastrophically).

  (Suggested by Raymond Hettinger and implemented by Benjamin Peterson; bpo-9778 [https://bugs.python.org/issue?@action=redirect&bpo=9778].)

- A new macro **`Py_VA_COPY`** copies the state of the variable argument list. It is equivalent to C99 *va_copy* but available on all Python platforms (bpo-2443 [https://bugs.python.org/issue?@action=redirect&bpo=2443]).

- A new C API function **`PySys_SetArgvEx()`** allows an embedded interpreter to set `sys.argv` without also modifying `sys.path` (bpo-5753 [https://bugs.python.org/issue?@action=redirect&bpo=5753]).

- **`PyEval_CallObject()`** is now only available in macro form. The function declaration, which was kept for backwards compatibility reasons, is now removed – the macro was introduced in 1997 (bpo-8276 [https://bugs.python.org/issue?@action=redirect&bpo=8276]).

- There is a new function `PyLong_AsLongLongAndOverflow()` which is analogous to `PyLong_AsLongAndOverflow()`. They both serve to convert Python `int` into a native fixed-width type while providing detection of cases where the conversion won’t fit (bpo-7767 [https://bugs.python.org/issue?@action=redirect&bpo=7767]).

- The **`PyUnicode_CompareWithASCIIString()`** function now returns *not equal* if the Python string is *NUL* terminated.

- There is a new function **`PyErr_NewExceptionWithDoc()`** that is like **`PyErr_NewException()`** but allows a docstring to be specified. This lets C exceptions have the same self-documenting capabilities as their pure Python counterparts (bpo-7033 [https://bugs.python.org/issue?@action=redirect&bpo=7033]).

- When compiled with the `--with-valgrind` option, the pymalloc allocator will be automatically disabled when running under Valgrind. This gives improved memory leak detection when running under Valgrind, while taking advantage of pymalloc at other times (bpo-2422 [https://bugs.python.org/issue?@action=redirect&bpo=2422]).

- Removed the `O?` format from the *PyArg_Parse* functions. The format is no longer used and it had never been documented (bpo-8837 [https://bugs.python.org/issue?@action=redirect&bpo=8837]).

There were a number of other small changes to the C-API. See the Misc/NEWS [https://github.com/python/cpython/blob/v3.2.6/Misc/NEWS] file for a complete list.

Also, there were a number of updates to the Mac OS X build, see Mac/BuildScript/README.txt [https://github.com/python/cpython/blob/v3.2.6/Mac/BuildScript/README.txt] for details. For users running a 32/64-bit build, there is a known problem with the default Tcl/Tk on Mac OS X 10.6. Accordingly, we recommend installing an updated alternative such as ActiveState Tcl/Tk 8.5.9 [https://web.archive.org/web/20101208191259/https://www.activestate.com/activetcl/downloads]. See https://www.python.org/download/mac/tcltk/ for additional details.

Porting to Python 3.2

This section lists previously described changes and other bugfixes that may require changes to your code:

- The `configparser` module has a number of clean-ups. The major change is to replace the old **`ConfigParser`** class with long-standing preferred alternative **`SafeConfigParser`**. In addition there are a number of smaller incompatibilities:
    - The interpolation syntax is now validated on `get()` and `set()` operations. In the default interpolation scheme, only two tokens with percent signs are valid: `%(name)s` and `%%`, the latter being an escaped percent sign.
    - The `set()` and `add_section()` methods now verify that values are actual strings. Formerly, unsupported types could be introduced unintentionally.
    - Duplicate sections or options from a single source now raise either `DuplicateSectionError` or `DuplicateOptionError`. Formerly, duplicates would silently overwrite a previous entry.
    - Inline comments are now disabled by default so now the **`;`** character can be safely used in values.
    - Comments now can be indented. Consequently, for **`;`** or **`#`** to appear at the start of a line in multiline values, it has to be interpolated. This keeps comment prefix characters in values from being mistaken as comments.
    - `""` is now a valid value and is no longer automatically converted to an empty string. For empty strings, use `"option ="` in a line.

- The **`nntplib`** module was reworked extensively, meaning that its APIs are often incompatible with the 3.1 APIs.
- **`bytearray`** objects can no longer be used as filenames; instead, they should be converted to **`bytes`**.
- The **`array.tostring()`** and **`array.fromstring()`** have been renamed to **`array.tobytes()`** and **`array.frombytes()`** for clarity. The old names have been deprecated. (See bpo-8990 [https://bugs.python.org/issue?@action=redirect&bpo=8990].)
- `PyArg_Parse*()` functions:
  - "t#" format has been removed: use "s#" or "s*" instead
  - "w" and "w#" formats has been removed: use "w*" instead
- The **`PyCObject`** type, deprecated in 3.1, has been removed. To wrap opaque C pointers in Python objects, the **`PyCapsule`** API should be used instead; the new type has a well-defined interface for passing typing safety information and a less complicated signature for calling a destructor.
- The **`sys.setfilesystemencoding()`** function was removed because it had a flawed design.
- The **`random.seed()`** function and method now salt string seeds with an sha512 hash function. To access the previous version of *seed* in order to reproduce Python 3.1 sequences, set the *version* argument to *1*, `random.seed(s, version=1)`.
- The previously deprecated **`string.maketrans()`** function has been removed in favor of the static methods **`bytes.maketrans()`** and **`bytearray.maketrans()`**. This change

solves the confusion around which types were supported by the `string` module. Now, `str`, `bytes`, and `bytearray` each have their own **maketrans** and **translate** methods with intermediate translation tables of the appropriate type.

(Contributed by Georg Brandl; bpo-5675 [https://bugs.python.org/issue?@action=redirect&bpo=5675].)

- The previously deprecated **`contextlib.nested()`** function has been removed in favor of a plain `with` statement which can accept multiple context managers. The latter technique is faster (because it is built-in), and it does a better job finalizing multiple context managers when one of them raises an exception:

```
with open('mylog.txt') as infile, open('a.out', 'w') as
outfile:
    for line in infile:
        if '<critical>' in line:
            outfile.write(line)
```

(Contributed by Georg Brandl and Mattias Brändström; appspot issue 53094 [https://codereview.appspot.com/53094].)

- `struct.pack()` now only allows bytes for the `s` string pack code. Formerly, it would accept text arguments and implicitly encode them to bytes using UTF-8. This was problematic because it made assumptions about the correct encoding and because a variable-length encoding can fail when writing to fixed length segment of a structure.

Code such as `struct.pack('<6sHHBBB', 'GIF87a', x, y)` should be rewritten with to use bytes instead of text, `struct.pack('<6sHHBBB', b'GIF87a', x, y)`.

(Discovered by David Beazley and fixed by Victor Stinner; bpo-10783 [https://bugs.python.org/issue?@action=redirect&bpo=10783].)

- The **`xml.etree.ElementTree`** class now raises an **`xml.etree.ElementTree.ParseError`** when a parse fails. Previously it raised an **`xml.parsers.expat.ExpatError`**.
- The new, longer **`str()`** value on floats may break doctests which rely on the old output format.
- In **`subprocess.Popen`**, the default value for *close_fds* is now `True` under Unix; under Windows, it is `True` if the three standard streams are set to `None`, `False` otherwise. Previously, *close_fds* was always `False` by default, which produced difficult to solve bugs or race conditions when open file descriptors would leak into the child process.
- Support for legacy HTTP 0.9 has been removed from **`urllib.request`** and **`http.client`**. Such support is still present on the server side (in **`http.server`**).

  (Contributed by Antoine Pitrou, bpo-10711 [https://bugs.python.org/issue?@action=redirect&bpo=10711].)
- SSL sockets in timeout mode now raise **`socket.timeout`** when a timeout occurs, rather than a generic **`SSLError`**.

  (Contributed by Antoine Pitrou, bpo-10272 [https://bugs.python.org/issue?@action=redirect&bpo=10272].)
- The misleading functions **`PyEval_AcquireLock()`** and **`PyEval_ReleaseLock()`** have been officially deprecated. The thread-state aware APIs (such as **`PyEval_SaveThread()`** and **`PyEval_RestoreThread()`**) should be used instead.

- Due to security risks, **`asyncore.handle_accept()`** has been deprecated, and a new function, **`asyncore.handle_accepted()`**, was added to replace it.

  (Contributed by Giampaolo Rodola in bpo-6706 [https://bugs.python.org/issue?@action=redirect&bpo=6706].)

- Due to the new GIL implementation, **`PyEval_InitThreads()`** cannot be called before **`Py_Initialize()`** anymore.

What’s New In Python 3.1

Author:: Raymond Hettinger

This article explains the new features in Python 3.1, compared to 3.0. Python 3.1 was released on June 27, 2009.

PEP 372: Ordered Dictionaries

Regular Python dictionaries iterate over key/value pairs in arbitrary order. Over the years, a number of authors have written alternative implementations that remember the order that the keys were originally inserted. Based on the experiences from those implementations, a new `collections.OrderedDict` class has been introduced.

The OrderedDict API is substantially the same as regular dictionaries but will iterate over keys and values in a guaranteed order depending on when a key was first inserted. If a new entry overwrites an existing entry, the original insertion position is left unchanged. Deleting an entry and reinserting it will move it to the end.

The standard library now supports use of ordered dictionaries in several modules. The `configparser` module uses them by default. This lets configuration files be read, modified, and then written back in their original order. The *_asdict()* method for `collections.namedtuple()` now returns an ordered dictionary with the values appearing in the same order as the underlying tuple indices. The `json` module is being built-out with an *object_pairs_hook* to allow OrderedDicts to be built by the decoder. Support was also added for third-party tools like PyYAML [https://pyyaml.org/].

See also

[PEP 372](https://peps.python.org/pep-0372/) [https://peps.python.org/pep-0372/] - Ordered Dictionaries
PEP written by Armin Ronacher and Raymond Hettinger. Implementation written by Raymond Hettinger.

Since an ordered dictionary remembers its insertion order, it can be used in conjunction with sorting to make a sorted dictionary:

```
>>> # regular unsorted dictionary
>>> d = {'banana': 3, 'apple':4, 'pear': 1, 'orange': 2}

>>> # dictionary sorted by key
>>> OrderedDict(sorted(d.items(), key=lambda t: t[0]))
OrderedDict([('apple', 4), ('banana', 3), ('orange', 2),
('pear', 1)])

>>> # dictionary sorted by value
>>> OrderedDict(sorted(d.items(), key=lambda t: t[1]))
OrderedDict([('pear', 1), ('orange', 2), ('banana', 3),
('apple', 4)])

>>> # dictionary sorted by length of the key string
>>> OrderedDict(sorted(d.items(), key=lambda t: len(t[0])))
OrderedDict([('pear', 1), ('apple', 4), ('orange', 2),
('banana', 3)])
```

The new sorted dictionaries maintain their sort order when entries are deleted. But when new keys are added, the keys are appended to the end and the sort is not maintained.

PEP 378: Format Specifier for Thousands Separator

The built-in `format()` function and the `str.format()` method use a mini-language that now includes a simple, non-locale aware way to format a number with a thousands separator. That provides a way to humanize a program's output, improving its professional appearance and readability:

```
>>> format(1234567, ',d')
'1,234,567'
>>> format(1234567.89, ',.2f')
'1,234,567.89'
>>> format(12345.6 + 8901234.12j, ',f')
'12,345.600000+8,901,234.120000j'
>>> format(Decimal('1234567.89'), ',f')
'1,234,567.89'
```

The supported types are `int`, `float`, `complex` and `decimal.Decimal`.

Discussions are underway about how to specify alternative separators like dots, spaces, apostrophes, or underscores. Locale-aware applications should use the existing *n* format specifier which already has some support for thousands separators.

See also

PEP 378 [https://peps.python.org/pep-0378/] - Format Specifier for Thousands Separator

PEP written by Raymond Hettinger and implemented by Eric Smith and Mark Dickinson.

Other Language Changes

Some smaller changes made to the core Python language are:

- Directories and zip archives containing a `__main__.py` file can now be executed directly by passing their name to the interpreter. The directory/zipfile is automatically inserted as the first entry in sys.path. (Suggestion and initial patch by Andy Chu; revised patch by Phillip J. Eby and Nick Coghlan; bpo-1739468 [https://bugs.python.org/issue?@action=redirect&bpo=1739468].)

- The **`int()`** type gained a `bit_length` method that returns the number of bits necessary to represent its argument in binary:

  ```
  >>> n = 37
  >>> bin(37)
  '0b100101'
  >>> n.bit_length()
  6
  >>> n = 2**123-1
  >>> n.bit_length()
  123
  >>> (n+1).bit_length()
  124
  ```

  (Contributed by Fredrik Johansson, Victor Stinner, Raymond Hettinger, and Mark Dickinson; bpo-3439 [https://bugs.python.org/issue?@action=redirect&bpo=3439].)

- The fields in **`format()`** strings can now be automatically numbered:

  ```
  >>> 'Sir {} of {}'.format('Gallahad', 'Camelot')
  'Sir Gallahad of Camelot'
  ```

  Formerly, the string would have required numbered fields such as: `'Sir {0} of {1}'`.

  (Contributed by Eric Smith; bpo-5237 [https://bugs.python.org/issue?@action=redirect&bpo=5237].)

- The **`string.maketrans()`** function is deprecated and is replaced by new static methods, `bytes.maketrans()` and `bytearray.maketrans()`. This change solves the confusion around which types were supported by the `string` module. Now, `str`, `bytes`, and `bytearray` each have their own **maketrans** and **translate** methods with intermediate translation tables of the appropriate type.

  (Contributed by Georg Brandl; bpo-5675 [https://bugs.python.org/issue?@action=redirect&bpo=5675].)

- The syntax of the `with` statement now allows multiple context managers in a single statement:

  ```
  >>> with open('mylog.txt') as infile, open('a.out', 'w') as
  outfile:
  ...     for line in infile:
  ...         if '<critical>' in line:
  ...             outfile.write(line)
  ```

  With the new syntax, the **`contextlib.nested()`** function is no longer needed and is now deprecated.

  (Contributed by Georg Brandl and Mattias Brändström; appspot issue 53094 [https://codereview.appspot.com/53094].)

- `round(x, n)` now returns an integer if *x* is an integer. Previously it returned a float:

  ```
  >>> round(1123, -2)
  1100
  ```

  (Contributed by Mark Dickinson; bpo-4707 [https://bugs.python.org/issue?@action=redirect&bpo=4707].)

- Python now uses David Gay’s algorithm for finding the shortest floating-point representation that doesn’t change its value. This should help mitigate some of the confusion surrounding binary floating-point numbers.

The significance is easily seen with a number like `1.1` which does not have an exact equivalent in binary floating point. Since there is no exact equivalent, an expression like `float('1.1')` evaluates to the nearest representable value which is `0x1.199999999999ap+0` in hex or `1.100000000000000088817841970012523233890533447265625` in decimal. That nearest value was and still is used in subsequent floating-point calculations.

What is new is how the number gets displayed. Formerly, Python used a simple approach. The value of `repr(1.1)` was computed as `format(1.1, '.17g')` which evaluated to `'1.1000000000000001'`. The advantage of using 17 digits was that it relied on IEEE-754 guarantees to assure that `eval(repr(1.1))` would round-trip exactly to its original value. The disadvantage is that many people found the output to be confusing (mistaking intrinsic limitations of binary floating-point representation as being a problem with Python itself).

The new algorithm for `repr(1.1)` is smarter and returns `'1.1'`. Effectively, it searches all equivalent string representations (ones that get stored with the same underlying float value) and returns the shortest representation.

The new algorithm tends to emit cleaner representations when possible, but it does not change the underlying values. So, it is still the case that `1.1 + 2.2 != 3.3` even though the representations may suggest otherwise.

The new algorithm depends on certain features in the underlying floating-point implementation. If the required features are not found, the old algorithm will continue to be used. Also, the text pickle protocols assure cross-platform portability by using the old algorithm.

(Contributed by Eric Smith and Mark Dickinson; bpo-1580 [https://bugs.python.org/issue?@action=redirect&bpo=1580])

New, Improved, and Deprecated Modules

- Added a **`collections.Counter`** class to support convenient counting of unique items in a sequence or iterable:

  ```
  >>> Counter(['red', 'blue', 'red', 'green', 'blue',
  'blue'])
  Counter({'blue': 3, 'red': 2, 'green': 1})
  ```

  (Contributed by Raymond Hettinger; bpo-1696199 [https://bugs.python.org/issue?@action=redirect&bpo=1696199].)

- Added a new module, **`tkinter.ttk`** for access to the Tk themed widget set. The basic idea of ttk is to separate, to the extent possible, the code implementing a widget's behavior from the code implementing its appearance.

  (Contributed by Guilherme Polo; bpo-2983 [https://bugs.python.org/issue?@action=redirect&bpo=2983].)

- The **`gzip.GzipFile`** and **`bz2.BZ2File`** classes now support the context management protocol:

  ```
  >>> # Automatically close file after writing
  >>> with gzip.GzipFile(filename, "wb") as f:
  ...     f.write(b"xxx")
  ```

(Contributed by Antoine Pitrou.)

- The **`decimal`** module now supports methods for creating a decimal object from a binary **`float`**. The conversion is exact but can sometimes be surprising:

  ```
  >>> Decimal.from_float(1.1)
  Decimal('1.100000000000000088817841970012523233890533447265
  625')
  ```

  The long decimal result shows the actual binary fraction being stored for *1.1*. The fraction has many digits because *1.1* cannot be exactly represented in binary.

  (Contributed by Raymond Hettinger and Mark Dickinson.)

- The **`itertools`** module grew two new functions. The **`itertools.combinations_with_replacement()`** function is one of four for generating combinatorics including permutations and Cartesian products. The **`itertools.compress()`** function mimics its namesake from APL. Also, the existing **`itertools.count()`** function now has an optional *step* argument and can accept any type of counting sequence including **`fractions.Fraction`** and **`decimal.Decimal`**:

  ```
  >>> [p+q for p,q in combinations_with_replacement('LOVE',
  2)]
  ['LL', 'LO', 'LV', 'LE', 'OO', 'OV', 'OE', 'VV', 'VE',
  'EE']

  >>> list(compress(data=range(10), selectors=
  [0,0,1,1,0,1,0,1,0,0]))
  [2, 3, 5, 7]

  >>> c = count(start=Fraction(1,2), step=Fraction(1,6))
  >>> [next(c), next(c), next(c), next(c)]
  ```

```
[Fraction(1, 2), Fraction(2, 3), Fraction(5, 6),
Fraction(1, 1)]
```

(Contributed by Raymond Hettinger.)

- `collections.namedtuple()` now supports a keyword argument *rename* which lets invalid fieldnames be automatically converted to positional names in the form _0, _1, etc. This is useful when the field names are being created by an external source such as a CSV header, SQL field list, or user input:

  ```
  >>> query = input()
  SELECT region, dept, count(*) FROM main GROUPBY region,
  dept

  >>> cursor.execute(query)
  >>> query_fields = [desc[0] for desc in cursor.description]
  >>> UserQuery = namedtuple('UserQuery', query_fields,
  rename=True)
  >>> pprint.pprint([UserQuery(*row) for row in cursor])
  [UserQuery(region='South', dept='Shipping', _2=185),
   UserQuery(region='North', dept='Accounting', _2=37),
   UserQuery(region='West', dept='Sales', _2=419)]
  ```

  (Contributed by Raymond Hettinger; bpo-1818 [https://bugs.python.org/issue?@action=redirect&bpo=1818].)

- The `re.sub()`, `re.subn()` and `re.split()` functions now accept a flags parameter.

  (Contributed by Gregory Smith.)

- The `logging` module now implements a simple `logging.NullHandler` class for applications that are not using logging but are calling library code that does. Setting-up a null handler will suppress spurious warnings such as “No handlers could be found for logger foo”:

```
>>> h = logging.NullHandler()
>>> logging.getLogger("foo").addHandler(h)
```

(Contributed by Vinay Sajip; bpo-4384 [https://bugs.python.org/issue?@action=redirect&bpo=4384]).

- The **`runpy`** module which supports the `-m` command line switch now supports the execution of packages by looking for and executing a `__main__` submodule when a package name is supplied.

  (Contributed by Andi Vajda; bpo-4195 [https://bugs.python.org/issue?@action=redirect&bpo=4195].)

- The **`pdb`** module can now access and display source code loaded via **`zipimport`** (or any other conformant **PEP 302** [https://peps.python.org/pep-0302/] loader).

  (Contributed by Alexander Belopolsky; bpo-4201 [https://bugs.python.org/issue?@action=redirect&bpo=4201].)

- **`functools.partial`** objects can now be pickled.

(Suggested by Antoine Pitrou and Jesse Noller. Implemented by Jack Diederich; bpo-5228 [https://bugs.python.org/issue?@action=redirect&bpo=5228].)

- Add **`pydoc`** help topics for symbols so that `help('@')` works as expected in the interactive environment.

  (Contributed by David Laban; bpo-4739 [https://bugs.python.org/issue?@action=redirect&bpo=4739].)

- The **`unittest`** module now supports skipping individual tests or classes of tests. And it supports marking a test as an expected failure, a test that is known to be broken, but shouldn’t be counted as a failure on a TestResult:

```
class TestGizmo(unittest.TestCase):

    @unittest.skipUnless(sys.platform.startswith("win"),
"requires Windows")
    def test_gizmo_on_windows(self):
        ...

    @unittest.expectedFailure
    def test_gimzo_without_required_library(self):
        ...
```

Also, tests for exceptions have been builtout to work with context managers using the `with` statement:

```
def test_division_by_zero(self):
    with self.assertRaises(ZeroDivisionError):
        x / 0
```

In addition, several new assertion methods were added including `assertSetEqual()`, `assertDictEqual()`, **`assertDictContainsSubset()`**, `assertListEqual()`, `assertTupleEqual()`, `assertSequenceEqual()`, `assertRaisesRegexp()`, `assertIsNone()`, and `assertIsNotNone()`.

(Contributed by Benjamin Peterson and Antoine Pitrou.)

- The `io` module has three new constants for the `seek()` method: `SEEK_SET`, `SEEK_CUR`, and `SEEK_END`.
- The `sys.version_info` tuple is now a named tuple:

  ```
  >>> sys.version_info
  sys.version_info(major=3, minor=1, micro=0,
  releaselevel='alpha', serial=2)
  ```

  (Contributed by Ross Light; bpo-4285 [https://bugs.python.org/issue?@action=redirect&bpo=4285].)
- The **`nntplib`** and `imaplib` modules now support IPv6.

(Contributed by Derek Morr; bpo-1655 [https://bugs.python.org/issue?@action=redirect&bpo=1655] and bpo-1664 [https://bugs.python.org/issue?@action=redirect&bpo=1664].)

- The **`pickle`** module has been adapted for better interoperability with Python 2.x when used with protocol 2 or lower. The reorganization of the standard library changed the formal reference for many objects. For example, `__builtin__.set` in Python 2 is called `builtins.set` in Python 3. This change confounded efforts to share data between different versions of Python. But now when protocol 2 or lower is selected, the pickler will automatically use the old Python 2 names for both loading and dumping. This remapping is turned-on by default but can be disabled with the *fix_imports* option:

  ```
  >>> s = {1, 2, 3}
  >>> pickle.dumps(s, protocol=0)
  b'c__builtin__\nset\np0\n((lp1\nL1L\naL2L\naL3L\natp2\nRp3\
  n.'
  >>> pickle.dumps(s, protocol=0, fix_imports=False)
  b'cbuiltins\nset\np0\n((lp1\nL1L\naL2L\naL3L\natp2\nRp3\n.'
  ```

  An unfortunate but unavoidable side-effect of this change is that protocol 2 pickles produced by Python 3.1 won’t be readable with Python 3.0. The latest pickle protocol, protocol 3, should be used when migrating data between Python 3.x implementations, as it doesn’t attempt to remain compatible with Python 2.x.

  (Contributed by Alexandre Vassalotti and Antoine Pitrou, bpo-6137 [https://bugs.python.org/issue?@action=redirect&bpo=6137].)

- A new module, **`importlib`** was added. It provides a complete, portable, pure Python reference implementation of the **`import`** statement and its counterpart, the `__import__()` function. It represents a substantial step forward in documenting and defining the actions that take place during imports.

(Contributed by Brett Cannon.)

Optimizations

Major performance enhancements have been added:

- The new I/O library (as defined in **PEP 3116** [https://peps.python.org/pep-3116/]) was mostly written in Python and quickly proved to be a problematic bottleneck in Python 3.0. In Python 3.1, the I/O library has been entirely rewritten in C and is 2 to 20 times faster depending on the task at hand. The pure Python version is still available for experimentation purposes through the `_pyio` module.

  (Contributed by Amaury Forgeot d’Arc and Antoine Pitrou.)

- Added a heuristic so that tuples and dicts containing only untrackable objects are not tracked by the garbage collector. This can reduce the size of collections and therefore the garbage collection overhead on long-running programs, depending on their particular use of datatypes.

  (Contributed by Antoine Pitrou, bpo-4688 [https://bugs.python.org/issue?@action=redirect&bpo=4688].)

- Enabling a configure option named `--with-computed-gotos` on compilers that support it (notably: gcc, SunPro, icc), the bytecode evaluation loop is compiled with a new dispatch mechanism which gives speedups of up to

20%, depending on the system, the compiler, and the benchmark.

(Contributed by Antoine Pitrou along with a number of other participants, bpo-4753 [https://bugs.python.org/issue?@action=redirect&bpo=4753]).

- The decoding of UTF-8, UTF-16 and LATIN-1 is now two to four times faster.

  (Contributed by Antoine Pitrou and Amaury Forgeot d'Arc, bpo-4868 [https://bugs.python.org/issue?@action=redirect&bpo=4868].)

- The `json` module now has a C extension to substantially improve its performance. In addition, the API was modified so that json works only with `str`, not with `bytes`. That change makes the module closely match the JSON specification [https://json.org/] which is defined in terms of Unicode.

  (Contributed by Bob Ippolito and converted to Py3.1 by Antoine Pitrou and Benjamin Peterson; bpo-4136 [https://bugs.python.org/issue?@action=redirect&bpo=4136].)

- Unpickling now interns the attribute names of pickled objects. This saves memory and allows pickles to be smaller.

  (Contributed by Jake McGuire and Antoine Pitrou; bpo-5084 [https://bugs.python.org/issue?@action=redirect&bpo=5084].)

IDLE

- IDLE's format menu now provides an option to strip trailing whitespace from a source file.

(Contributed by Roger D. Serwy; bpo-5150 [https://bugs.python.org/issue?@action=redirect&bpo=5150].)

Build and C API Changes

Changes to Python’s build process and to the C API include:

- Integers are now stored internally either in base `2**15` or in base `2**30`, the base being determined at build time. Previously, they were always stored in base `2**15`. Using base `2**30` gives significant performance improvements on 64-bit machines, but benchmark results on 32-bit machines have been mixed. Therefore, the default is to use base `2**30` on 64-bit machines and base `2**15` on 32-bit machines; on Unix, there’s a new configure option `--enable-big-digits` that can be used to override this default.

  Apart from the performance improvements this change should be invisible to end users, with one exception: for testing and debugging purposes there’s a new **`sys.int_info`** that provides information about the internal format, giving the number of bits per digit and the size in bytes of the C type used to store each digit:

  ```
  >>> import sys
  >>> sys.int_info
  sys.int_info(bits_per_digit=30, sizeof_digit=4)
  ```

  (Contributed by Mark Dickinson; bpo-4258 [https://bugs.python.org/issue?@action=redirect&bpo=4258].)

- The **`PyLong_AsUnsignedLongLong()`** function now handles a negative *pylong* by raising **`OverflowError`** instead of **`TypeError`**.

(Contributed by Mark Dickinson and Lisandro Dalcrin; bpo-5175 [https://bugs.python.org/issue?@action=redirect&bpo=5175].)

- Deprecated **`PyNumber_Int()`**. Use **`PyNumber_Long()`** instead.

  (Contributed by Mark Dickinson; bpo-4910 [https://bugs.python.org/issue?@action=redirect&bpo=4910].)

- Added a new **`PyOS_string_to_double()`** function to replace the deprecated functions **`PyOS_ascii_strtod()`** and **`PyOS_ascii_atof()`**.

  (Contributed by Mark Dickinson; bpo-5914 [https://bugs.python.org/issue?@action=redirect&bpo=5914].)

- Added **`PyCapsule`** as a replacement for the **`PyCObject`** API. The principal difference is that the new type has a well defined interface for passing typing safety information and a less complicated signature for calling a destructor. The old type had a problematic API and is now deprecated.

  (Contributed by Larry Hastings; bpo-5630 [https://bugs.python.org/issue?@action=redirect&bpo=5630].)

Porting to Python 3.1

This section lists previously described changes and other bugfixes that may require changes to your code:

- The new floating-point string representations can break existing doctests. For example:

  ```
  def e():
      '''Compute the base of natural logarithms.
  ```

```
    >>> e()
    2.7182818284590451

    '''
    return sum(1/math.factorial(x) for x in
reversed(range(30)))

doctest.testmod()

***********************************************************
***********
Failed example:
    e()
Expected:
    2.7182818284590451
Got:
    2.718281828459045
***********************************************************
***********
```

- The automatic name remapping in the pickle module for protocol 2 or lower can make Python 3.1 pickles unreadable in Python 3.0. One solution is to use protocol 3. Another solution is to set the *fix_imports* option to `False`. See the discussion above for more details.

What’s New In Python 3.0

Author:: Guido van Rossum

This article explains the new features in Python 3.0, compared to 2.6. Python 3.0, also known as “Python 3000” or “Py3K”, is the first ever *intentionally backwards incompatible* Python release. Python 3.0 was released on December 3, 2008. There are more changes than in a typical release, and more that are important for all Python users. Nevertheless, after digesting the changes, you’ll find that Python really hasn’t changed all that much – by and large, we’re mostly fixing well-known annoyances and warts, and removing a lot of old cruft.

This article doesn’t attempt to provide a complete specification of all new features, but instead tries to give a convenient overview. For full details, you should refer to the documentation for Python 3.0, and/or the many PEPs referenced in the text. If you want to understand the complete implementation and design rationale for a particular feature, PEPs usually have more details than the regular documentation; but note that PEPs usually are not kept up-to-date once a feature has been fully implemented.

Due to time constraints this document is not as complete as it should have been. As always for a new release, the `Misc/NEWS` file in the source distribution contains a wealth of detailed information about every small thing that was changed.

Common Stumbling Blocks

This section lists those few changes that are most likely to trip you up if you’re used to Python 2.5.

Print Is A Function

The `print` statement has been replaced with a `print()` function, with keyword arguments to replace most of the special syntax of the old `print` statement (PEP 3105 [https://peps.python.org/pep-3105/]). Examples:

```
Old: print "The answer is", 2*2
New: print("The answer is", 2*2)

Old: print x,           # Trailing comma suppresses newline
New: print(x, end=" ")  # Appends a space instead of a newline

Old: print              # Prints a newline
New: print()            # You must call the function!

Old: print >>sys.stderr, "fatal error"
New: print("fatal error", file=sys.stderr)

Old: print (x, y)       # prints repr((x, y))
New: print((x, y))      # Not the same as print(x, y)!
```

You can also customize the separator between items, e.g.:

```
print("There are <", 2**32, "> possibilities!", sep="")
```

which produces:

```
There are <4294967296> possibilities!
```

Note:

- The `print()` function doesn’t support the “softspace” feature of the old `print` statement. For example, in Python 2.x, `print "A\n", "B"` would write `"A\nB\n"`; but in Python 3.0, `print("A\n", "B")` writes `"A\n B\n"`.

- Initially, you’ll be finding yourself typing the old `print x` a lot in interactive mode. Time to retrain your fingers to type `print(x)` instead!
- When using the `2to3` source-to-source conversion tool, all `print` statements are automatically converted to `print()` function calls, so this is mostly a non-issue for larger projects.

Views And Iterators Instead Of Lists

Some well-known APIs no longer return lists:

- `dict` methods `dict.keys()`, `dict.items()` and `dict.values()` return “views” instead of lists. For example, this no longer works: `k = d.keys(); k.sort()`. Use `k = sorted(d)` instead (this works in Python 2.5 too and is just as efficient).

- Also, the **`dict.iterkeys()`**, **`dict.iteritems()`** and **`dict.itervalues()`** methods are no longer supported.

- `map()` and `filter()` return iterators. If you really need a list and the input sequences are all of equal length, a quick fix is to wrap `map()` in `list()`, e.g. `list(map(...))`, but a better fix is often to use a list comprehension (especially when the original code uses `lambda`), or rewriting the code so it doesn’t need a list at all. Particularly tricky is `map()` invoked for the side effects of the function; the correct transformation is to use a regular `for` loop (since creating a list would just be wasteful).

  If the input sequences are not of equal length, `map()` will stop at the termination of the shortest of the sequences. For full compatibility with `map()` from Python 2.x, also wrap the sequences in `itertools.zip_longest()`, e.g.

`map(func, *sequences)` becomes `list(map(func, itertools.zip_longest(*sequences)))`.

- `range()` now behaves like **`xrange()`** used to behave, except it works with values of arbitrary size. The latter no longer exists.

- `zip()` now returns an iterator.

Ordering Comparisons

Python 3.0 has simplified the rules for ordering comparisons:

- The ordering comparison operators (`<`, `<=`, `>=`, `>`) raise a TypeError exception when the operands don't have a meaningful natural ordering. Thus, expressions like `1 < ''`, `0 > None` or `len <= len` are no longer valid, and e.g. `None < None` raises `TypeError` instead of returning `False`. A corollary is that sorting a heterogeneous list no longer makes sense – all the elements must be comparable to each other. Note that this does not apply to the `==` and `!=` operators: objects of different incomparable types always compare unequal to each other.
- `sorted()` and `list.sort()` no longer accept the *cmp* argument providing a comparison function. Use the *key* argument instead. N.B. the *key* and *reverse* arguments are now "keyword-only".
- The **`cmp()`** function should be treated as gone, and the **`__cmp__()`** special method is no longer supported. Use `__lt__()` for sorting, `__eq__()` with `__hash__()`, and other rich comparisons as needed. (If you really need the **`cmp()`** functionality, you could use the expression `(a > b) - (a < b)` as the equivalent for `cmp(a, b)`.)

Integers

- **PEP 237** [https://peps.python.org/pep-0237/]: Essentially, **`long`** renamed to `int`. That is, there is only one built-in integral type, named `int`; but it behaves mostly like the old **`long`** type.
- **PEP 238** [https://peps.python.org/pep-0238/]: An expression like `1/2` returns a float. Use `1//2` to get the truncating behavior. (The latter syntax has existed for years, at least since Python 2.2.)
- The **`sys.maxint`** constant was removed, since there is no longer a limit to the value of integers. However, `sys.maxsize` can be used as an integer larger than any practical list or string index. It conforms to the implementation's "natural" integer size and is typically the same as **`sys.maxint`** in previous releases on the same platform (assuming the same build options).
- The `repr()` of a long integer doesn't include the trailing `L` anymore, so code that unconditionally strips that character will chop off the last digit instead. (Use `str()` instead.)
- Octal literals are no longer of the form `0720`; use `0o720` instead.

Text Vs. Data Instead Of Unicode Vs. 8-bit

Everything you thought you knew about binary data and Unicode has changed.

- Python 3.0 uses the concepts of *text* and (binary) *data* instead of Unicode strings and 8-bit strings. All text is Unicode; however *encoded* Unicode is represented as binary data. The type used to hold text is `str`, the type used to hold data is `bytes`. The biggest difference with the 2.x situation is that any attempt to mix text and data in Python 3.0 raises `TypeError`, whereas if you were to mix Unicode and 8-bit strings in Python 2.x, it would work if the 8-bit string happened to contain only 7-bit

(ASCII) bytes, but you would get `UnicodeDecodeError` if it contained non-ASCII values. This value-specific behavior has caused numerous sad faces over the years.

- As a consequence of this change in philosophy, pretty much all code that uses Unicode, encodings or binary data most likely has to change. The change is for the better, as in the 2.x world there were numerous bugs having to do with mixing encoded and unencoded text. To be prepared in Python 2.x, start using **`unicode`** for all unencoded text, and `str` for binary or encoded data only. Then the `2to3` tool will do most of the work for you.
- You can no longer use `u"..."` literals for Unicode text. However, you must use `b"..."` literals for binary data.
- As the `str` and `bytes` types cannot be mixed, you must always explicitly convert between them. Use `str.encode()` to go from `str` to `bytes`, and `bytes.decode()` to go from `bytes` to `str`. You can also use `bytes(s, encoding=...)` and `str(b, encoding=...)`, respectively.
- Like `str`, the `bytes` type is immutable. There is a separate *mutable* type to hold buffered binary data, `bytearray`. Nearly all APIs that accept `bytes` also accept `bytearray`. The mutable API is based on `collections.MutableSequence`.
- All backslashes in raw string literals are interpreted literally. This means that `'\U'` and `'\u'` escapes in raw strings are not treated specially. For example, `r'\u20ac'` is a string of 6 characters in Python 3.0, whereas in 2.6, `ur'\u20ac'` was the single “euro” character. (Of course, this change only affects raw string literals; the euro character is `'\u20ac'` in Python 3.0.)
- The built-in **`basestring`** abstract type was removed. Use `str` instead. The `str` and `bytes` types don’t have functionality enough in common to warrant a shared base class. The `2to3` tool (see below) replaces every occurrence of **`basestring`** with `str`.
- Files opened as text files (still the default mode for `open()`) always use an encoding to map between strings

(in memory) and bytes (on disk). Binary files (opened with a `b` in the mode argument) always use bytes in memory. This means that if a file is opened using an incorrect mode or encoding, I/O will likely fail loudly, instead of silently producing incorrect data. It also means that even Unix users will have to specify the correct mode (text or binary) when opening a file. There is a platform-dependent default encoding, which on Unixy platforms can be set with the `LANG` environment variable (and sometimes also with some other platform-specific locale-related environment variables). In many cases, but not all, the system default is UTF-8; you should never count on this default. Any application reading or writing more than pure ASCII text should probably have a way to override the encoding. There is no longer any need for using the encoding-aware streams in the `codecs` module.

- The initial values of `sys.stdin`, `sys.stdout` and `sys.stderr` are now unicode-only text files (i.e., they are instances of `io.TextIOBase`). To read and write bytes data with these streams, you need to use their `io.TextIOBase.buffer` attribute.
- Filenames are passed to and returned from APIs as (Unicode) strings. This can present platform-specific problems because on some platforms filenames are arbitrary byte strings. (On the other hand, on Windows filenames are natively stored as Unicode.) As a work-around, most APIs (e.g. `open()` and many functions in the `os` module) that take filenames accept `bytes` objects as well as strings, and a few APIs have a way to ask for a `bytes` return value. Thus, `os.listdir()` returns a list of `bytes` instances if the argument is a `bytes` instance, and `os.getcwdb()` returns the current working directory as a `bytes` instance. Note that when `os.listdir()` returns a list of strings, filenames that cannot be decoded properly are omitted rather than raising `UnicodeError`.

- Some system APIs like `os.environ` and `sys.argv` can also present problems when the bytes made available by the system is not interpretable using the default encoding. Setting the `LANG` variable and rerunning the program is probably the best approach.
- **PEP 3138** [https://peps.python.org/pep-3138/]: The `repr()` of a string no longer escapes non-ASCII characters. It still escapes control characters and code points with non-printable status in the Unicode standard, however.
- **PEP 3120** [https://peps.python.org/pep-3120/]: The default source encoding is now UTF-8.
- **PEP 3131** [https://peps.python.org/pep-3131/]: Non-ASCII letters are now allowed in identifiers. (However, the standard library remains ASCII-only with the exception of contributor names in comments.)
- The **`StringIO`** and **`cStringIO`** modules are gone. Instead, import the `io` module and use `io.StringIO` or `io.BytesIO` for text and data respectively.
- See also the Unicode HOWTO, which was updated for Python 3.0.

Overview Of Syntax Changes

This section gives a brief overview of every *syntactic* change in Python 3.0.

New Syntax

- **PEP 3107** [https://peps.python.org/pep-3107/]: Function argument and return value annotations. This provides a standardized way of annotating a function’s parameters and return value. There are no semantics attached to such annotations except that they can be introspected at runtime using the `__annotations__` attribute. The intent is to encourage experimentation through metaclasses, decorators or frameworks.

- **PEP 3102** [https://peps.python.org/pep-3102/]: Keyword-only arguments. Named parameters occurring after `*args` in the parameter list *must* be specified using keyword syntax in the call. You can also use a bare `*` in the parameter list to indicate that you don’t accept a variable-length argument list, but you do have keyword-only arguments.

- Keyword arguments are allowed after the list of base classes in a class definition. This is used by the new convention for specifying a metaclass (see next section), but can be used for other purposes as well, as long as the metaclass supports it.

- **PEP 3104** [https://peps.python.org/pep-3104/]: `nonlocal` statement. Using `nonlocal x` you can now assign directly to a variable in an outer (but non-global) scope. **`nonlocal`** is a new reserved word.

- **PEP 3132** [https://peps.python.org/pep-3132/]: Extended Iterable Unpacking. You can now write things like `a, b, *rest = some_sequence`. And even `*rest, a = stuff`. The `rest` object is always a (possibly empty) list; the right-hand side may be any iterable. Example:

  ```
  (a, *rest, b) = range(5)
  ```

  This sets *a* to `0`, *b* to `4`, and *rest* to `[1, 2, 3]`.

- Dictionary comprehensions: `{k: v for k, v in stuff}` means the same thing as `dict(stuff)` but is more flexible. (This is **PEP 274** [https://peps.python.org/pep-0274/] vindicated. :-)

- Set literals, e.g. `{1, 2}`. Note that `{}` is an empty dictionary; use `set()` for an empty set. Set comprehensions are also supported; e.g., `{x for x in stuff}` means the same thing as `set(stuff)` but is more flexible.

- New octal literals, e.g. `0o720` (already in 2.6). The old octal literals (`0720`) are gone.

- New binary literals, e.g. `0b1010` (already in 2.6), and there is a new corresponding built-in function, **`bin()`**.

- Bytes literals are introduced with a leading `b` or `B`, and there is a new corresponding built-in function, **`bytes()`**.

Changed Syntax

- **PEP 3109** [https://peps.python.org/pep-3109/] and **PEP 3134** [https://peps.python.org/pep-3134/]: new **`raise`** statement syntax: `raise [expr [from expr]]`. See below.

- **`as`** and **`with`** are now reserved words. (Since 2.6, actually.)

- `True`, `False`, and `None` are reserved words. (2.6 partially enforced the restrictions on `None` already.)

- Change from **`except`** *exc*, *var* to **`except`** *exc* **`as`** *var*. See **PEP 3110** [https://peps.python.org/pep-3110/].

- **PEP 3115** [https://peps.python.org/pep-3115/]: New Metaclass Syntax. Instead of:

  ```
  class C:
      __metaclass__ = M
      ...
  ```

  you must now use:

```
class C(metaclass=M):
    ...
```

The module-global **`__metaclass__`** variable is no longer supported. (It was a crutch to make it easier to default to new-style classes without deriving every class from **`object`**.)

- List comprehensions no longer support the syntactic form `[... for var in item1, item2, ...]`. Use `[... for var in (item1, item2, ...)]` instead. Also note that list comprehensions have different semantics: they are closer to syntactic sugar for a generator expression inside a **`list()`** constructor, and in particular the loop control variables are no longer leaked into the surrounding scope.

- The *ellipsis* (`...`) can be used as an atomic expression anywhere. (Previously it was only allowed in slices.) Also, it *must* now be spelled as `...`. (Previously it could also be spelled as `. . .`, by a mere accident of the grammar.)

Removed Syntax

- **PEP 3113** [https://peps.python.org/pep-3113/]: Tuple parameter unpacking removed. You can no longer write `def foo(a, (b, c)): ...`. Use `def foo(a, b_c): b, c = b_c` instead.
- Removed backticks (use **`repr()`** instead).
- Removed `<>` (use `!=` instead).
- Removed keyword: **`exec()`** is no longer a keyword; it remains as a function. (Fortunately the function syntax was also accepted in 2.x.) Also note that **`exec()`** no longer takes a stream argument; instead of `exec(f)` you can use `exec(f.read())`.
- Integer literals no longer support a trailing `l` or `L`.

- String literals no longer support a leading `u` or `U`.
- The **`from`** *module* **`import`** `*` syntax is only allowed at the module level, no longer inside functions.
- The only acceptable syntax for relative imports is `from .[module] import name`. All **`import`** forms not starting with `.` are interpreted as absolute imports. (**PEP 328** [https://peps.python.org/pep-0328/])
- Classic classes are gone.

Changes Already Present In Python 2.6

Since many users presumably make the jump straight from Python 2.5 to Python 3.0, this section reminds the reader of new features that were originally designed for Python 3.0 but that were back-ported to Python 2.6. The corresponding sections in What's New in Python 2.6 should be consulted for longer descriptions.

- PEP 343: The 'with' statement. The **`with`** statement is now a standard feature and no longer needs to be imported from the **`__future__`**. Also check out Writing Context Managers and The contextlib module.
- PEP 366: Explicit Relative Imports From a Main Module. This enhances the usefulness of the **`-m`** option when the referenced module lives in a package.
- PEP 370: Per-user site-packages Directory.
- PEP 371: The multiprocessing Package.
- PEP 3101: Advanced String Formatting. Note: the 2.6 description mentions the **`format()`** method for both 8-bit and Unicode strings. In 3.0, only the **`str`** type (text strings with Unicode support) supports this method; the **`bytes`** type does not. The plan is to eventually make this the only API for string formatting, and to start deprecating the `%` operator in Python 3.1.

- PEP 3105: print As a Function. This is now a standard feature and no longer needs to be imported from `__future__`. More details were given above.
- PEP 3110: Exception-Handling Changes. The **`except`** *exc* **`as`** *var* syntax is now standard and **`except`** *exc*, *var* is no longer supported. (Of course, the **`as`** *var* part is still optional.)
- PEP 3112: Byte Literals. The `b"..."` string literal notation (and its variants like `b'...'`, `b"""..."""`, and `br"..."`) now produces a literal of type `bytes`.
- PEP 3116: New I/O Library. The `io` module is now the standard way of doing file I/O. The built-in `open()` function is now an alias for `io.open()` and has additional keyword arguments *encoding*, *errors*, *newline* and *closefd*. Also note that an invalid *mode* argument now raises `ValueError`, not `IOError`. The binary file object underlying a text file object can be accessed as **`f.buffer`** (but beware that the text object maintains a buffer of itself in order to speed up the encoding and decoding operations).
- PEP 3118: Revised Buffer Protocol. The old builtin **`buffer()`** is now really gone; the new builtin `memoryview()` provides (mostly) similar functionality.
- PEP 3119: Abstract Base Classes. The `abc` module and the ABCs defined in the `collections` module plays a somewhat more prominent role in the language now, and built-in collection types like `dict` and `list` conform to the `collections.MutableMapping` and `collections.MutableSequence` ABCs, respectively.
- PEP 3127: Integer Literal Support and Syntax. As mentioned above, the new octal literal notation is the only one supported, and binary literals have been added.
- PEP 3129: Class Decorators.
- PEP 3141: A Type Hierarchy for Numbers. The `numbers` module is another new use of ABCs, defining Python’s

"numeric tower". Also note the new `fractions` module which implements `numbers.Rational`.

Library Changes

Due to time constraints, this document does not exhaustively cover the very extensive changes to the standard library. **PEP 3108** [https://peps.python.org/pep-3108/] is the reference for the major changes to the library. Here's a capsule review:

- Many old modules were removed. Some, like **`gopherlib`** (no longer used) and **`md5`** (replaced by `hashlib`), were already deprecated by **PEP 4** [https://peps.python.org/pep-0004/]. Others were removed as a result of the removal of support for various platforms such as Irix, BeOS and Mac OS 9 (see **PEP 11** [https://peps.python.org/pep-0011/]). Some modules were also selected for removal in Python 3.0 due to lack of use or because a better replacement exists. See **PEP 3108** [https://peps.python.org/pep-3108/] for an exhaustive list.

- The **`bsddb3`** package was removed because its presence in the core standard library has proved over time to be a particular burden for the core developers due to testing instability and Berkeley DB's release schedule. However, the package is alive and well, externally maintained at https://www.jcea.es/programacion/pybsddb.htm.

- Some modules were renamed because their old name disobeyed **PEP 8** [https://peps.python.org/pep-0008/], or for various other reasons. Here's the list:

Old Name	**New Name**
_winreg	winreg
ConfigParser	configparser
copy_reg	copyreg
Queue	queue
SocketServer	socketserver
markupbase	_markupbase
repr	reprlib
test.test_support	test.support

- A common pattern in Python 2.x is to have one version of a module implemented in pure Python, with an optional accelerated version implemented as a C extension; for example, `pickle` and `cPickle`. This places the burden of importing the accelerated version and falling back on the pure Python version on each user of these modules. In Python 3.0, the accelerated versions are considered implementation details of the pure Python versions. Users should always import the standard version, which attempts to import the accelerated version and falls back to the pure Python version. The `pickle` / `cPickle` pair received this treatment. The `profile` module is on the list for 3.1. The `StringIO` module has been turned into a class in the `io` module.

- Some related modules have been grouped into packages, and usually the submodule names have been simplified. The resulting new packages are:
    - `dbm` (`anydbm`, `dbhash`, `dbm`, `dumbdbm`, `gdbm`, `whichdb`).
    - `html` (`HTMLParser`, `htmlentitydefs`).
    - `http` (`httplib`, `BaseHTTPServer`, `CGIHTTPServer`, `SimpleHTTPServer`, `Cookie`, `cookielib`).
    - `tkinter` (all `Tkinter`-related modules except `turtle`). The target audience of `turtle` doesn't really care about `tkinter`. Also note that as of Python 2.6, the functionality of `turtle` has been greatly enhanced.
    - `urllib` (`urllib`, `urllib2`, `urlparse`, `robotparse`).
    - `xmlrpc` (`xmlrpclib`, `DocXMLRPCServer`, `SimpleXMLRPCServer`).

Some other changes to standard library modules, not covered by **PEP 3108** [https://peps.python.org/pep-3108/]:

- Killed `sets`. Use the built-in `set()` class.
- Cleanup of the `sys` module: removed `sys.exitfunc()`, `sys.exc_clear()`, `sys.exc_type`, `sys.exc_value`, `sys.exc_traceback`. (Note that `sys.last_type` etc. remain.)
- Cleanup of the `array.array` type: the `read()` and `write()` methods are gone; use `fromfile()` and `tofile()` instead. Also, the `'c'` typecode for array is gone – use either `'b'` for bytes or `'u'` for Unicode characters.
- Cleanup of the `operator` module: removed `sequenceIncludes()` and `isCallable()`.
- Cleanup of the `thread` module: `acquire_lock()` and `release_lock()` are gone; use `acquire()` and `release()` instead.

- Cleanup of the `random` module: removed the **`jumpahead()`** API.
- The **`new`** module is gone.
- The functions **`os.tmpnam()`**, **`os.tempnam()`** and **`os.tmpfile()`** have been removed in favor of the `tempfile` module.
- The `tokenize` module has been changed to work with bytes. The main entry point is now `tokenize.tokenize()`, instead of generate_tokens.
- **`string.letters`** and its friends (**`string.lowercase`** and **`string.uppercase`**) are gone. Use `string.ascii_letters` etc. instead. (The reason for the removal is that **`string.letters`** and friends had locale-specific behavior, which is a bad idea for such attractively named global "constants".)
- Renamed module **`__builtin__`** to `builtins` (removing the underscores, adding an 's'). The **`__builtins__`** variable found in most global namespaces is unchanged. To modify a builtin, you should use `builtins`, not **`__builtins__`**!

PEP 3101 [https://peps.python.org/pep-3101/]: A New Approach To String Formatting

- A new system for built-in string formatting operations replaces the `%` string formatting operator. (However, the `%` operator is still supported; it will be deprecated in Python 3.1 and removed from the language at some later time.) Read **PEP 3101** [https://peps.python.org/pep-3101/] for the full scoop.

Changes To Exceptions

The APIs for raising and catching exception have been cleaned up and new powerful features added:

- **PEP 352** [https://peps.python.org/pep-0352/]: All exceptions must be derived (directly or indirectly) from `BaseException`. This is the root of the exception hierarchy. This is not new as a recommendation, but the *requirement* to inherit from `BaseException` is new. (Python 2.6 still allowed classic classes to be raised, and placed no restriction on what you can catch.) As a consequence, string exceptions are finally truly and utterly dead.

- Almost all exceptions should actually derive from `Exception`; `BaseException` should only be used as a base class for exceptions that should only be handled at the top level, such as `SystemExit` or `KeyboardInterrupt`. The recommended idiom for handling all exceptions except for this latter category is to use `except Exception`.

- **`StandardError`** was removed.

- Exceptions no longer behave as sequences. Use the `args` attribute instead.

- **PEP 3109** [https://peps.python.org/pep-3109/]: Raising exceptions. You must now use `raise Exception(args)` instead of `raise Exception, args`. Additionally, you can no longer explicitly specify a traceback; instead, if you *have* to do this, you can assign directly to the `__traceback__` attribute (see below).

- **PEP 3110** [https://peps.python.org/pep-3110/]: Catching exceptions. You must now use `except SomeException as variable` instead of `except SomeException, variable`. Moreover, the *variable* is explicitly deleted when the `except` block is left.

- **PEP 3134** [https://peps.python.org/pep-3134/]: Exception chaining. There are two cases: implicit chaining and explicit chaining. Implicit chaining happens when an exception is raised in an `except` or `finally` handler block. This usually happens due to a bug in the handler block; we call this a *secondary* exception. In this case, the original exception (that was being handled) is saved as the `__context__` attribute of the secondary exception. Explicit chaining is invoked with this syntax:

  ```
  raise SecondaryException() from primary_exception
  ```

  (where *primary_exception* is any expression that produces an exception object, probably an exception that was previously caught). In this case, the primary exception is stored on the `__cause__` attribute of the secondary exception. The traceback printed when an unhandled exception occurs walks the chain of `__cause__` and `__context__` attributes and prints a separate traceback for each component of the chain, with the primary exception at the top. (Java users may recognize this behavior.)

- **PEP 3134** [https://peps.python.org/pep-3134/]: Exception objects now store their traceback as the `__traceback__` attribute. This means that an exception object now contains all the information pertaining to an exception, and there are fewer reasons to use `sys.exc_info()` (though the latter is not removed).

- A few exception messages are improved when Windows fails to load an extension module. For example, `error code 193` is now `%1 is not a valid Win32 application`. Strings now deal with non-English locales.

Miscellaneous Other Changes

Operators And Special Methods

- `!=` now returns the opposite of `==`, unless `==` returns **`NotImplemented`**.
- The concept of “unbound methods” has been removed from the language. When referencing a method as a class attribute, you now get a plain function object.
- **`__getslice__()`**, **`__setslice__()`** and **`__delslice__()`** were killed. The syntax `a[i:j]` now translates to `a.__getitem__(slice(i, j))` (or **`__setitem__()`** or **`__delitem__()`**, when used as an assignment or deletion target, respectively).
- **PEP 3114** [https://peps.python.org/pep-3114/]: the standard **`next()`** method has been renamed to **`__next__()`**.
- The **`__oct__()`** and **`__hex__()`** special methods are removed – **`oct()`** and **`hex()`** use **`__index__()`** now to convert the argument to an integer.
- Removed support for **`__members__`** and **`__methods__`**.
- The function attributes named **`func_X`** have been renamed to use the **`__X__`** form, freeing up these names in the function attribute namespace for user-defined attributes. To wit, **`func_closure`**, **`func_code`**, **`func_defaults`**, **`func_dict`**, **`func_doc`**, **`func_globals`**, **`func_name`** were renamed to **`__closure__`**, **`__code__`**, **`__defaults__`**, **`__dict__`**, **`__doc__`**, **`__globals__`**, **`__name__`**, respectively.
- **`__nonzero__()`** is now **`__bool__()`**.

Builtins

- **PEP 3135** [https://peps.python.org/pep-3135/]: New **`super()`**. You can now invoke **`super()`** without arguments and (assuming this is in a regular instance method defined inside a **`class`** statement) the right class and instance will automatically be chosen. With arguments, the behavior of **`super()`** is unchanged.

- **PEP 3111** [https://peps.python.org/pep-3111/]: `raw_input()` was renamed to `input()`. That is, the new `input()` function reads a line from `sys.stdin` and returns it with the trailing newline stripped. It raises `EOFError` if the input is terminated prematurely. To get the old behavior of `input()`, use `eval(input())`.
- A new built-in function `next()` was added to call the `__next__()` method on an object.
- The `round()` function rounding strategy and return type have changed. Exact halfway cases are now rounded to the nearest even result instead of away from zero. (For example, `round(2.5)` now returns `2` rather than `3`.) `round(x[, n])` now delegates to `x.__round__([n])` instead of always returning a float. It generally returns an integer when called with a single argument and a value of the same type as `x` when called with two arguments.
- Moved `intern()` to `sys.intern()`.
- Removed: `apply()`. Instead of `apply(f, args)` use `f(*args)`.
- Removed `callable()`. Instead of `callable(f)` you can use `isinstance(f, collections.Callable)`. The `operator.isCallable()` function is also gone.
- Removed `coerce()`. This function no longer serves a purpose now that classic classes are gone.
- Removed `execfile()`. Instead of `execfile(fn)` use `exec(open(fn).read())`.
- Removed the `file` type. Use `open()`. There are now several different kinds of streams that open can return in the `io` module.
- Removed `reduce()`. Use `functools.reduce()` if you really need it; however, 99 percent of the time an explicit `for` loop is more readable.
- Removed `reload()`. Use `imp.reload()`.
- Removed. `dict.has_key()` – use the `in` operator instead.

Build and C API Changes

Due to time constraints, here is a *very* incomplete list of changes to the C API.

- Support for several platforms was dropped, including but not limited to Mac OS 9, BeOS, RISCOS, Irix, and Tru64.
- **PEP 3118** [https://peps.python.org/pep-3118/]: New Buffer API.
- **PEP 3121** [https://peps.python.org/pep-3121/]: Extension Module Initialization & Finalization.
- **PEP 3123** [https://peps.python.org/pep-3123/]: Making `PyObject_HEAD` conform to standard C.
- No more C API support for restricted execution.
- **`PyNumber_Coerce()`**, **`PyNumber_CoerceEx()`**, **`PyMember_Get()`**, and **`PyMember_Set()`** C APIs are removed.
- New C API `PyImport_ImportModuleNoBlock()`, works like `PyImport_ImportModule()` but won’t block on the import lock (returning an error instead).
- Renamed the boolean conversion C-level slot and method: `nb_nonzero` is now `nb_bool`.
- Removed **`METH_OLDARGS`** and **`WITH_CYCLE_GC`** from the C API.

Performance

The net result of the 3.0 generalizations is that Python 3.0 runs the pystone benchmark around 10% slower than Python 2.5. Most likely the biggest cause is the removal of special-casing for small integers. There’s room for improvement, but it will happen after 3.0 is released!

Porting To Python 3.0

For porting existing Python 2.5 or 2.6 source code to Python 3.0, the best strategy is the following:

0. (Prerequisite:) Start with excellent test coverage.

1. Port to Python 2.6. This should be no more work than the average port from Python 2.x to Python 2.(x+1). Make sure all your tests pass.
2. (Still using 2.6:) Turn on the **`-3`** command line switch. This enables warnings about features that will be removed (or change) in 3.0. Run your test suite again, and fix code that you get warnings about until there are no warnings left, and all your tests still pass.
3. Run the `2to3` source-to-source translator over your source code tree. Run the result of the translation under Python 3.0. Manually fix up any remaining issues, fixing problems until all tests pass again.

It is not recommended to try to write source code that runs unchanged under both Python 2.6 and 3.0; you'd have to use a very contorted coding style, e.g. avoiding `print` statements, metaclasses, and much more. If you are maintaining a library that needs to support both Python 2.6 and Python 3.0, the best approach is to modify step 3 above by editing the 2.6 version of the source code and running the `2to3` translator again, rather than editing the 3.0 version of the source code.

For porting C extensions to Python 3.0, please see Porting Extension Modules to Python 3.

What’s New in Python 2.7

Author:: A.M. Kuchling (amk at amk.ca)

This article explains the new features in Python 2.7. Python 2.7 was released on July 3, 2010.

Numeric handling has been improved in many ways, for both floating-point numbers and for the `Decimal` class. There are some useful additions to the standard library, such as a greatly enhanced `unittest` module, the `argparse` module for parsing command-line options, convenient `OrderedDict` and `Counter` classes in the `collections` module, and many other improvements.

Python 2.7 is planned to be the last of the 2.x releases, so we worked on making it a good release for the long term. To help with porting to Python 3, several new features from the Python 3.x series have been included in 2.7.

This article doesn’t attempt to provide a complete specification of the new features, but instead provides a convenient overview. For full details, you should refer to the documentation for Python 2.7 at https://docs.python.org. If you want to understand the rationale for the design and implementation, refer to the PEP for a particular new feature or the issue on https://bugs.python.org in which a change was discussed. Whenever possible, “What’s New in Python” links to the bug/patch item for each change.

The Future for Python 2.x

Python 2.7 is the last major release in the 2.x series, as the Python maintainers have shifted the focus of their new feature development efforts to the Python 3.x series. This means that while Python 2 continues to receive bug fixes, and to be updated to build correctly on new hardware and versions of supported operated systems, there will be no new full feature releases for the language or standard library.

However, while there is a large common subset between Python 2.7 and Python 3, and many of the changes involved in migrating to that common subset, or directly to Python 3, can be safely automated, some other changes (notably those associated with Unicode handling) may require careful consideration, and preferably robust automated regression test suites, to migrate effectively.

This means that Python 2.7 will remain in place for a long time, providing a stable and supported base platform for production systems that have not yet been ported to Python 3. The full expected lifecycle of the Python 2.7 series is detailed in **PEP 373** [https://peps.python.org/pep-0373/].

Some key consequences of the long-term significance of 2.7 are:

- As noted above, the 2.7 release has a much longer period of maintenance when compared to earlier 2.x versions. Python 2.7 is currently expected to remain supported by the core development team (receiving security updates and other bug fixes) until at least 2020 (10 years after its initial release, compared to the more typical support period of 18–24 months).
- As the Python 2.7 standard library ages, making effective use of the Python Package Index (either directly or via a redistributor) becomes more important for Python 2 users. In addition to a wide variety of third party packages for various tasks, the available packages

include backports of new modules and features from the Python 3 standard library that are compatible with Python 2, as well as various tools and libraries that can make it easier to migrate to Python 3. The Python Packaging User Guide [https://packaging.python.org] provides guidance on downloading and installing software from the Python Package Index.
- While the preferred approach to enhancing Python 2 is now the publication of new packages on the Python Package Index, this approach doesn't necessarily work in all cases, especially those related to network security. In exceptional cases that cannot be handled adequately by publishing new or updated packages on PyPI, the Python Enhancement Proposal process may be used to make the case for adding new features directly to the Python 2 standard library. Any such additions, and the maintenance releases where they were added, will be noted in the New Features Added to Python 2.7 Maintenance Releases section below.

For projects wishing to migrate from Python 2 to Python 3, or for library and framework developers wishing to support users on both Python 2 and Python 3, there are a variety of tools and guides available to help decide on a suitable approach and manage some of the technical details involved. The recommended starting point is the How to port Python 2 Code to Python 3 HOWTO guide.

Changes to the Handling of Deprecation Warnings

For Python 2.7, a policy decision was made to silence warnings only of interest to developers by default. `DeprecationWarning` and its descendants are now ignored unless otherwise requested, preventing users from seeing warnings triggered by an application. This change was also made in the branch that became Python 3.2. (Discussed on stdlib-sig and carried out in bpo-7319 [https://bugs.python.org/issue?@action=redirect&bpo=7319].)

In previous releases, `DeprecationWarning` messages were enabled by default, providing Python developers with a clear indication of where their code may break in a future major version of Python.

However, there are increasingly many users of Python-based applications who are not directly involved in the development of those applications. `DeprecationWarning` messages are irrelevant to such users, making them worry about an application that's actually working correctly and burdening application developers with responding to these concerns.

You can re-enable display of `DeprecationWarning` messages by running Python with the `-Wdefault` (short form: `-Wd`) switch, or by setting the `PYTHONWARNINGS` environment variable to `"default"` (or `"d"`) before running Python. Python code can also re-enable them by calling `warnings.simplefilter('default')`.

The `unittest` module also automatically reenables deprecation warnings when running tests.

Python 3.1 Features

Much as Python 2.6 incorporated features from Python 3.0, version 2.7 incorporates some of the new features in Python 3.1. The 2.x series continues to provide tools for migrating to the 3.x series.

A partial list of 3.1 features that were backported to 2.7:

- The syntax for set literals (`{1,2,3}` is a mutable set).
- Dictionary and set comprehensions (`{i: i*2 for i in range(3)}`).
- Multiple context managers in a single **`with`** statement.
- A new version of the **`io`** library, rewritten in C for performance.
- The ordered-dictionary type described in PEP 372: Adding an Ordered Dictionary to collections.
- The new `","` format specifier described in PEP 378: Format Specifier for Thousands Separator.
- The **`memoryview`** object.
- A small subset of the **`importlib`** module, described below.
- The **`repr()`** of a float `x` is shorter in many cases: it's now based on the shortest decimal string that's guaranteed to round back to `x`. As in previous versions of Python, it's guaranteed that `float(repr(x))` recovers `x`.
- Float-to-string and string-to-float conversions are correctly rounded. The **`round()`** function is also now correctly rounded.
- The **`PyCapsule`** type, used to provide a C API for extension modules.
- The **`PyLong_AsLongAndOverflow()`** C API function.

Other new Python3-mode warnings include:

- **`operator.isCallable()`** and **`operator.sequenceIncludes()`**, which are not supported in 3.x, now trigger warnings.
- The **`-3`** switch now automatically enables the **`-Qwarn`** switch that causes warnings about using classic division with integers and long integers.

PEP 372: Adding an Ordered Dictionary to collections

Regular Python dictionaries iterate over key/value pairs in arbitrary order. Over the years, a number of authors have written alternative implementations that remember the order that the keys were originally inserted. Based on the experiences from those implementations, 2.7 introduces a new **`OrderedDict`** class in the **`collections`** module.

The **`OrderedDict`** API provides the same interface as regular dictionaries but iterates over keys and values in a guaranteed order depending on when a key was first inserted:

```
>>> from collections import OrderedDict
>>> d = OrderedDict([('first', 1),
...                  ('second', 2),
...                  ('third', 3)])
>>> d.items()
[('first', 1), ('second', 2), ('third', 3)]
```

If a new entry overwrites an existing entry, the original insertion position is left unchanged:

```
>>> d['second'] = 4
>>> d.items()
[('first', 1), ('second', 4), ('third', 3)]
```

Deleting an entry and reinserting it will move it to the end:

```
>>> del d['second']
>>> d['second'] = 5
>>> d.items()
[('first', 1), ('third', 3), ('second', 5)]
```

The **`popitem()`** method has an optional *last* argument that defaults to `True`. If *last* is true, the most recently added key is returned and removed; if it’s false, the oldest key is selected:

```
>>> od = OrderedDict([(x,0) for x in range(20)])
>>> od.popitem()
(19, 0)
>>> od.popitem()
(18, 0)
>>> od.popitem(last=False)
(0, 0)
>>> od.popitem(last=False)
(1, 0)
```

Comparing two ordered dictionaries checks both the keys and values, and requires that the insertion order was the same:

```
>>> od1 = OrderedDict([('first', 1),
...                    ('second', 2),
...                    ('third', 3)])
>>> od2 = OrderedDict([('third', 3),
...                    ('first', 1),
...                    ('second', 2)])
>>> od1 == od2
False
>>> # Move 'third' key to the end
>>> del od2['third']; od2['third'] = 3
>>> od1 == od2
True
```

Comparing an `OrderedDict` with a regular dictionary ignores the insertion order and just compares the keys and values.

How does the `OrderedDict` work? It maintains a doubly linked list of keys, appending new keys to the list as they're inserted. A secondary dictionary maps keys to their corresponding list node, so deletion doesn't have to traverse the entire linked list and therefore remains $O(1)$.

The standard library now supports use of ordered dictionaries in several modules.

- The `ConfigParser` module uses them by default, meaning that configuration files can now be read, modified, and then written back in their original order.
- The `_asdict()` method for `collections.namedtuple()` now returns an ordered dictionary with the values appearing in the same order as the underlying tuple indices.
- The `json` module's `JSONDecoder` class constructor was extended with an *object_pairs_hook* parameter to allow `OrderedDict` instances to be built by the decoder. Support was also added for third-party tools like PyYAML [https://pyyaml.org/].

See also

PEP 372 [https://peps.python.org/pep-0372/] - Adding an ordered dictionary to collections

PEP written by Armin Ronacher and Raymond Hettinger; implemented by Raymond Hettinger.

PEP 378: Format Specifier for Thousands Separator

To make program output more readable, it can be useful to add separators to large numbers, rendering them as 18,446,744,073,709,551,616 instead of 18446744073709551616.

The fully general solution for doing this is the `locale` module, which can use different separators (“,” in North America, “.” in Europe) and different grouping sizes, but `locale` is complicated to use and unsuitable for multi-threaded applications where different threads are producing output for different locales.

Therefore, a simple comma-grouping mechanism has been added to the mini-language used by the `str.format()` method. When formatting a floating-point number, simply include a comma between the width and the precision:

```
>>> '{:20,.2f}'.format(18446744073709551616.0)
'18,446,744,073,709,551,616.00'
```

When formatting an integer, include the comma after the width:

```
>>> '{:20,d}'.format(18446744073709551616)
'18,446,744,073,709,551,616'
```

This mechanism is not adaptable at all; commas are always used as the separator and the grouping is always into three-digit groups. The comma-formatting mechanism isn't as general as the `locale` module, but it's easier to use.

See also

PEP 378 [https://peps.python.org/pep-0378/] - Format Specifier for Thousands Separator
PEP written by Raymond Hettinger; implemented by Eric Smith.

PEP 389: The argparse Module for Parsing Command Lines

The `argparse` module for parsing command-line arguments was added as a more powerful replacement for the `optparse` module.

This means Python now supports three different modules for parsing command-line arguments: `getopt`, `optparse`, and `argparse`. The `getopt` module closely resembles the C library’s **`getopt()`** function, so it remains useful if you’re writing a Python prototype that will eventually be rewritten in C. `optparse` becomes redundant, but there are no plans to remove it because there are many scripts still using it, and there’s no automated way to update these scripts. (Making the `argparse` API consistent with `optparse`’s interface was discussed but rejected as too messy and difficult.)

In short, if you’re writing a new script and don’t need to worry about compatibility with earlier versions of Python, use `argparse` instead of `optparse`.

Here’s an example:

```
import argparse

parser = argparse.ArgumentParser(description='Command-line
example.')

# Add optional switches
parser.add_argument('-v', action='store_true',
dest='is_verbose',
                    help='produce verbose output')
parser.add_argument('-o', action='store', dest='output',
                    metavar='FILE',
                    help='direct output to FILE instead of
stdout')
parser.add_argument('-C', action='store', type=int,
dest='context',
                    metavar='NUM', default=0,
                    help='display NUM lines of added context')

# Allow any number of additional arguments.
parser.add_argument(nargs='*', action='store', dest='inputs',
                    help='input filenames (default is stdin)')

args = parser.parse_args()
print args.__dict__
```

Unless you override it, **`-h`** and **`--help`** switches are automatically added, and produce neatly formatted output:

```
-> ./python.exe argparse-example.py --help
usage: argparse-example.py [-h] [-v] [-o FILE] [-C NUM] [inputs
[inputs ...]]

Command-line example.

positional arguments:
  inputs      input filenames (default is stdin)

optional arguments:
  -h, --help  show this help message and exit
  -v          produce verbose output
  -o FILE     direct output to FILE instead of stdout
  -C NUM      display NUM lines of added context
```

As with [`optparse`](), the command-line switches and arguments are returned as an object with attributes named by the *dest* parameters:

```
-> ./python.exe argparse-example.py -v
{'output': None,
 'is_verbose': True,
 'context': 0,
 'inputs': []}

-> ./python.exe argparse-example.py -v -o /tmp/output -C 4
file1 file2
{'output': '/tmp/output',
 'is_verbose': True,
 'context': 4,
 'inputs': ['file1', 'file2']}
```

[`argparse`]() has much fancier validation than [`optparse`](); you can specify an exact number of arguments as an integer, 0 or more arguments by passing `'*'`, 1 or more by passing `'+'`, or an optional argument with `'?'`. A top-level parser can contain sub-parsers to define subcommands that have different sets of switches, as in `svn commit`, `svn checkout`, etc.

You can specify an argument's type as `FileType`, which will automatically open files for you and understands that `'-'` means standard input or output.

See also

`argparse` documentation
The documentation page of the argparse module.

Migrating optparse code to argparse
Part of the Python documentation, describing how to convert code that uses `optparse`.

PEP 389 [https://peps.python.org/pep-0389/] - argparse - New Command Line Parsing Module
PEP written and implemented by Steven Bethard.

PEP 391: Dictionary-Based Configuration For Logging

The `logging` module is very flexible; applications can define a tree of logging subsystems, and each logger in this tree can filter out certain messages, format them differently, and direct messages to a varying number of handlers.

All this flexibility can require a lot of configuration. You can write Python statements to create objects and set their properties, but a complex set-up requires verbose but boring code. `logging` also supports a `fileConfig()` function that parses a file, but the file format doesn't support configuring filters, and it's messier to generate programmatically.

Python 2.7 adds a `dictConfig()` function that uses a dictionary to configure logging. There are many ways to produce a dictionary from different sources: construct one with code; parse a file containing JSON; or use a YAML parsing library if one is installed. For more information see Configuration functions.

The following example configures two loggers, the root logger and a logger named “network”. Messages sent to the root logger will be sent to the system log using the syslog protocol, and messages to the “network” logger will be written to a `network.log` file that will be rotated once the log reaches 1MB.

```
import logging
import logging.config

configdict = {
 'version': 1,    # Configuration schema in use; must be 1 for
now
 'formatters': {
     'standard': {
         'format': ('%(asctime)s %(name)-15s '
                    '%(levelname)-8s %(message)s')}},

 'handlers': {'netlog': {'backupCount': 10,
                     'class':
'logging.handlers.RotatingFileHandler',
                     'filename': '/logs/network.log',
                     'formatter': 'standard',
                     'level': 'INFO',
                     'maxBytes': 1000000},
              'syslog': {'class':
'logging.handlers.SysLogHandler',
                         'formatter': 'standard',
                         'level': 'ERROR'}},

 # Specify all the subordinate loggers
 'loggers': {
             'network': {
```

```
                         'handlers': ['netlog']
             }
 },
 # Specify properties of the root logger
 'root': {
          'handlers': ['syslog']
 },
}

# Set up configuration
logging.config.dictConfig(configdict)

# As an example, log two error messages
logger = logging.getLogger('/')
logger.error('Database not found')

netlogger = logging.getLogger('network')
netlogger.error('Connection failed')
```

Three smaller enhancements to the `logging` module, all implemented by Vinay Sajip, are:

- The `SysLogHandler` class now supports syslogging over TCP. The constructor has a *socktype* parameter giving the type of socket to use, either `socket.SOCK_DGRAM` for UDP or `socket.SOCK_STREAM` for TCP. The default protocol remains UDP.
- `Logger` instances gained a `getChild()` method that retrieves a descendant logger using a relative path. For example, once you retrieve a logger by doing `log = getLogger('app')`, calling `log.getChild('network.listen')` is equivalent to `getLogger('app.network.listen')`.
- The `LoggerAdapter` class gained an `isEnabledFor()` method that takes a *level* and returns whether the underlying logger would process a message of that level of importance.

See also

PEP 391 [https://peps.python.org/pep-0391/] - Dictionary-Based Configuration For Logging

PEP written and implemented by Vinay Sajip.

PEP 3106: Dictionary Views

The dictionary methods `keys()`, `values()`, and `items()` are different in Python 3.x. They return an object called a *view* instead of a fully materialized list.

It’s not possible to change the return values of `keys()`, `values()`, and `items()` in Python 2.7 because too much code would break. Instead the 3.x versions were added under the new names **`viewkeys()`**, **`viewvalues()`**, and **`viewitems()`**.

```
>>> d = dict((i*10, chr(65+i)) for i in range(26))
>>> d
{0: 'A', 130: 'N', 10: 'B', 140: 'O', 20: ..., 250: 'Z'}
>>> d.viewkeys()
dict_keys([0, 130, 10, 140, 20, 150, 30, ..., 250])
```

Views can be iterated over, but the key and item views also behave like sets. The `&` operator performs intersection, and `|` performs a union:

```
>>> d1 = dict((i*10, chr(65+i)) for i in range(26))
>>> d2 = dict((i**.5, i) for i in range(1000))
>>> d1.viewkeys() & d2.viewkeys()
set([0.0, 10.0, 20.0, 30.0])
>>> d1.viewkeys() | range(0, 30)
set([0, 1, 130, 3, 4, 5, 6, ..., 120, 250])
```

The view keeps track of the dictionary and its contents change as the dictionary is modified:

```
>>> vk = d.viewkeys()
>>> vk
```

```
dict_keys([0, 130, 10, ..., 250])
>>> d[260] = '&'
>>> vk
dict_keys([0, 130, 260, 10, ..., 250])
```

However, note that you can't add or remove keys while you're iterating over the view:

```
>>> for k in vk:
...     d[k*2] = k
...
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
RuntimeError: dictionary changed size during iteration
```

You can use the view methods in Python 2.x code, and the 2to3 converter will change them to the standard `keys()`, `values()`, and `items()` methods.

See also

PEP 3106 [https://peps.python.org/pep-3106/] - Revamping dict.keys(), .values() and .items()

PEP written by Guido van Rossum. Backported to 2.7 by Alexandre Vassalotti; bpo-1967 [https://bugs.python.org/issue?@action=redirect&bpo=1967].

PEP 3137: The memoryview Object

The `memoryview` object provides a view of another object's memory content that matches the `bytes` type's interface.

```
>>> import string
>>> m = memoryview(string.letters)
>>> m
<memory at 0x37f850>
>>> len(m)           # Returns length of underlying object
```

```
52
>>> m[0], m[25], m[26]   # Indexing returns one byte
('a', 'z', 'A')
>>> m2 = m[0:26]         # Slicing returns another memoryview
>>> m2
<memory at 0x37f080>
```

The content of the view can be converted to a string of bytes or a list of integers:

```
>>> m2.tobytes()
'abcdefghijklmnopqrstuvwxyz'
>>> m2.tolist()
[97, 98, 99, 100, 101, 102, 103, ... 121, 122]
>>>
```

`memoryview` objects allow modifying the underlying object if it’s a mutable object.

```
>>> m2[0] = 75
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: cannot modify read-only memory
>>> b = bytearray(string.letters)  # Creating a mutable object
>>> b
bytearray(b'abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXY
Z')
>>> mb = memoryview(b)
>>> mb[0] = '*'         # Assign to view, changing the
bytearray.
>>> b[0:5]              # The bytearray has been changed.
bytearray(b'*bcde')
>>>
```

See also

PEP 3137 [https://peps.python.org/pep-3137/] - Immutable Bytes and Mutable Buffer

PEP written by Guido van Rossum. Implemented by Travis Oliphant, Antoine Pitrou and others. Backported to 2.7 by Antoine Pitrou; bpo-2396 [https://bugs.python.org/issue?@action=redirect&bpo=2396].

Other Language Changes

Some smaller changes made to the core Python language are:

- The syntax for set literals has been backported from Python 3.x. Curly brackets are used to surround the contents of the resulting mutable set; set literals are distinguished from dictionaries by not containing colons and values. `{}` continues to represent an empty dictionary; use `set()` for an empty set.

  ```
  >>> {1, 2, 3, 4, 5}
  set([1, 2, 3, 4, 5])
  >>> set() # empty set
  set([])
  >>> {}    # empty dict
  {}
  ```

  Backported by Alexandre Vassalotti; bpo-2335 [https://bugs.python.org/issue?@action=redirect&bpo=2335].

- Dictionary and set comprehensions are another feature backported from 3.x, generalizing list/generator comprehensions to use the literal syntax for sets and dictionaries.

  ```
  >>> {x: x*x for x in range(6)}
  {0: 0, 1: 1, 2: 4, 3: 9, 4: 16, 5: 25}
  >>> {('a'*x) for x in range(6)}
  set(['', 'a', 'aa', 'aaa', 'aaaa', 'aaaaa'])
  ```

  Backported by Alexandre Vassalotti; bpo-2333 [https://bugs.python.org/issue?@action=redirect&bpo=2333].

- The `with` statement can now use multiple context managers in one statement. Context managers are processed from left to right and each one is treated as beginning a new **`with`** statement. This means that:

  ```
  with A() as a, B() as b:
      ... suite of statements ...
  ```

  is equivalent to:

  ```
  with A() as a:
      with B() as b:
          ... suite of statements ...
  ```

  The **`contextlib.nested()`** function provides a very similar function, so it's no longer necessary and has been deprecated.

  (Proposed in https://codereview.appspot.com/53094; implemented by Georg Brandl.)

- Conversions between floating-point numbers and strings are now correctly rounded on most platforms. These conversions occur in many different places: `str()` on floats and complex numbers; the `float` and `complex` constructors; numeric formatting; serializing and deserializing floats and complex numbers using the `marshal`, `pickle` and `json` modules; parsing of float and imaginary literals in Python code; and `Decimal`-to-float conversion.

  Related to this, the `repr()` of a floating-point number *x* now returns a result based on the shortest decimal string that's guaranteed to round back to *x* under correct rounding (with round-half-to-even rounding mode). Previously it gave a string based on rounding x to 17 decimal digits.

The rounding library responsible for this improvement works on Windows and on Unix platforms using the gcc, icc, or suncc compilers. There may be a small number of platforms where correct operation of this code cannot be guaranteed, so the code is not used on such systems. You can find out which code is being used by checking **`sys.float_repr_style`**, which will be `short` if the new code is in use and `legacy` if it isn't.

Implemented by Eric Smith and Mark Dickinson, using David Gay's `dtoa.c` library; bpo-7117 [https://bugs.python.org/issue?@action=redirect&bpo=7117].

- Conversions from long integers and regular integers to floating point now round differently, returning the floating-point number closest to the number. This doesn't matter for small integers that can be converted exactly, but for large numbers that will unavoidably lose precision, Python 2.7 now approximates more closely. For example, Python 2.6 computed the following:

  ```
  >>> n = 295147905179352891391
  >>> float(n)
  2.9514790517935283e+20
  >>> n - long(float(n))
  65535L
  ```

  Python 2.7's floating-point result is larger, but much closer to the true value:

  ```
  >>> n = 295147905179352891391
  >>> float(n)
  2.9514790517935289e+20
  >>> n - long(float(n))
  -1L
  ```

  (Implemented by Mark Dickinson; bpo-3166 [https://bugs.python.org/issue?@action=redirect&bpo=3166].)

Integer division is also more accurate in its rounding behaviours. (Also implemented by Mark Dickinson; bpo-1811 [https://bugs.python.org/issue?@action=redirect&bpo=1811].)

- Implicit coercion for complex numbers has been removed; the interpreter will no longer ever attempt to call a **`__coerce__()`** method on complex objects. (Removed by Meador Inge and Mark Dickinson; bpo-5211 [https://bugs.python.org/issue?@action=redirect&bpo=5211].)
- The **`str.format()`** method now supports automatic numbering of the replacement fields. This makes using **`str.format()`** more closely resemble using `%s` formatting:

```
>>> '{}:{}:{}'.format(2009, 04, 'Sunday')
'2009:4:Sunday'
>>> '{}:{}:{day}'.format(2009, 4, day='Sunday')
'2009:4:Sunday'
```

The auto-numbering takes the fields from left to right, so the first `{...}` specifier will use the first argument to **`str.format()`**, the next specifier will use the next argument, and so on. You can't mix auto-numbering and explicit numbering – either number all of your specifier fields or none of them – but you can mix auto-numbering and named fields, as in the second example above. (Contributed by Eric Smith; bpo-5237 [https://bugs.python.org/issue?@action=redirect&bpo=5237].)

Complex numbers now correctly support usage with **`format()`**, and default to being right-aligned. Specifying a precision or comma-separation applies to both the real and imaginary parts of the number, but a specified field width and alignment is applied to the whole of the resulting `1.5+3j` output. (Contributed by Eric Smith; bpo-1588 [https://bugs.python.org/issue?@action=redirect&bpo=1588] and

bpo-7988 [https://bugs.python.org/issue?@action=redirect&bpo=7988].)

The ‘F’ format code now always formats its output using uppercase characters, so it will now produce ‘INF’ and ‘NAN’. (Contributed by Eric Smith; bpo-3382 [https://bugs.python.org/issue?@action=redirect&bpo=3382].)

A low-level change: the `object.__format__()` method now triggers a `PendingDeprecationWarning` if it’s passed a format string, because the `__format__()` method for `object` converts the object to a string representation and formats that. Previously the method silently applied the format string to the string representation, but that could hide mistakes in Python code. If you’re supplying formatting information such as an alignment or precision, presumably you’re expecting the formatting to be applied in some object-specific way. (Fixed by Eric Smith; bpo-7994 [https://bugs.python.org/issue?@action=redirect&bpo=7994].)

- The `int()` and `long()` types gained a `bit_length` method that returns the number of bits necessary to represent its argument in binary:

```
>>> n = 37
>>> bin(n)
'0b100101'
>>> n.bit_length()
6
>>> n = 2**123-1
>>> n.bit_length()
123
>>> (n+1).bit_length()
124
```

(Contributed by Fredrik Johansson and Victor Stinner; bpo-3439 [https://bugs.python.org/issue?

@action=redirect&bpo=3439].)

- The **`import`** statement will no longer try an absolute import if a relative import (e.g. `from .os import sep`) fails. This fixes a bug, but could possibly break certain **`import`** statements that were only working by accident. (Fixed by Meador Inge; bpo-7902 [https://bugs.python.org/issue?@action=redirect&bpo=7902].)

- It’s now possible for a subclass of the built-in **`unicode`** type to override the **`__unicode__()`** method. (Implemented by Victor Stinner; bpo-1583863 [https://bugs.python.org/issue?@action=redirect&bpo=1583863].)

- The **`bytearray`** type’s **`translate()`** method now accepts `None` as its first argument. (Fixed by Georg Brandl; bpo-4759 [https://bugs.python.org/issue?@action=redirect&bpo=4759].)

- When using **`@classmethod`** and **`@staticmethod`** to wrap methods as class or static methods, the wrapper object now exposes the wrapped function as their **`__func__`** attribute. (Contributed by Amaury Forgeot d’Arc, after a suggestion by George Sakkis; bpo-5982 [https://bugs.python.org/issue?@action=redirect&bpo=5982].)

- When a restricted set of attributes were set using `__slots__`, deleting an unset attribute would not raise **`AttributeError`** as you would expect. Fixed by Benjamin Peterson; bpo-7604 [https://bugs.python.org/issue?@action=redirect&bpo=7604].)

- Two new encodings are now supported: “cp720”, used primarily for Arabic text; and “cp858”, a variant of CP 850 that adds the euro symbol. (CP720 contributed by Alexander Belchenko and Amaury Forgeot d’Arc in bpo-1616979 [https://bugs.python.org/issue?@action=redirect&bpo=1616979]; CP858 contributed by Tim

Hatch in bpo-8016 [https://bugs.python.org/issue?@action=redirect&bpo=8016].)

- The **`file`** object will now set the **`filename`** attribute on the `IOError` exception when trying to open a directory on POSIX platforms (noted by Jan Kaliszewski; bpo-4764 [https://bugs.python.org/issue?@action=redirect&bpo=4764]), and now explicitly checks for and forbids writing to read-only file objects instead of trusting the C library to catch and report the error (fixed by Stefan Krah; bpo-5677 [https://bugs.python.org/issue?@action=redirect&bpo=5677]).

- The Python tokenizer now translates line endings itself, so the `compile()` built-in function now accepts code using any line-ending convention. Additionally, it no longer requires that the code end in a newline.

- Extra parentheses in function definitions are illegal in Python 3.x, meaning that you get a syntax error from `def f((x)): pass`. In Python3-warning mode, Python 2.7 will now warn about this odd usage. (Noted by James Lingard; bpo-7362 [https://bugs.python.org/issue?@action=redirect&bpo=7362].)

- It’s now possible to create weak references to old-style class objects. New-style classes were always weak-referenceable. (Fixed by Antoine Pitrou; bpo-8268 [https://bugs.python.org/issue?@action=redirect&bpo=8268].)

- When a module object is garbage-collected, the module’s dictionary is now only cleared if no one else is holding a reference to the dictionary (bpo-7140 [https://bugs.python.org/issue?@action=redirect&bpo=7140]).

Interpreter Changes

A new environment variable, **`PYTHONWARNINGS`**, allows controlling warnings. It should be set to a string containing warning settings, equivalent to those used with the **`-W`** switch, separated by commas. (Contributed by Brian Curtin; bpo-7301 [https://bugs.python.org/issue?@action=redirect&bpo=7301].)

For example, the following setting will print warnings every time they occur, but turn warnings from the **`Cookie`** module into an error. (The exact syntax for setting an environment variable varies across operating systems and shells.)

```
export PYTHONWARNINGS=all,error:::Cookie:0
```

Optimizations

Several performance enhancements have been added:

- A new opcode was added to perform the initial setup for **`with`** statements, looking up the **`__enter__()`** and **`__exit__()`** methods. (Contributed by Benjamin Peterson.)
- The garbage collector now performs better for one common usage pattern: when many objects are being allocated without deallocating any of them. This would previously take quadratic time for garbage collection, but now the number of full garbage collections is reduced as the number of objects on the heap grows. The new logic only performs a full garbage collection pass when the middle generation has been collected 10 times and when the number of survivor objects from the middle generation exceeds 10% of the number of objects in the oldest generation. (Suggested by Martin von Löwis and implemented by Antoine Pitrou; bpo-4074 [https://bugs.python.org/issue?@action=redirect&bpo=4074].)
- The garbage collector tries to avoid tracking simple containers which can’t be part of a cycle. In Python 2.7,

this is now true for tuples and dicts containing atomic types (such as ints, strings, etc.). Transitively, a dict containing tuples of atomic types won’t be tracked either. This helps reduce the cost of each garbage collection by decreasing the number of objects to be considered and traversed by the collector. (Contributed by Antoine Pitrou; bpo-4688 [https://bugs.python.org/issue?@action=redirect&bpo=4688].)

- Long integers are now stored internally either in base `2**15` or in base `2**30`, the base being determined at build time. Previously, they were always stored in base `2**15`. Using base `2**30` gives significant performance improvements on 64-bit machines, but benchmark results on 32-bit machines have been mixed. Therefore, the default is to use base `2**30` on 64-bit machines and base `2**15` on 32-bit machines; on Unix, there’s a new configure option **`--enable-big-digits`** that can be used to override this default.

  Apart from the performance improvements this change should be invisible to end users, with one exception: for testing and debugging purposes there’s a new structseq **`sys.long_info`** that provides information about the internal format, giving the number of bits per digit and the size in bytes of the C type used to store each digit:

  ```
  >>> import sys
  >>> sys.long_info
  sys.long_info(bits_per_digit=30, sizeof_digit=4)
  ```

  (Contributed by Mark Dickinson; bpo-4258 [https://bugs.python.org/issue?@action=redirect&bpo=4258].)

  Another set of changes made long objects a few bytes smaller: 2 bytes smaller on 32-bit systems and 6 bytes on 64-bit. (Contributed by Mark Dickinson; bpo-5260 [https://bugs.python.org/issue?@action=redirect&bpo=5260].)

- The division algorithm for long integers has been made faster by tightening the inner loop, doing shifts instead of multiplications, and fixing an unnecessary extra iteration. Various benchmarks show speedups of between 50% and 150% for long integer divisions and modulo operations. (Contributed by Mark Dickinson; bpo-5512 [https://bugs.python.org/issue?@action=redirect&bpo=5512].) Bitwise operations are also significantly faster (initial patch by Gregory Smith; bpo-1087418 [https://bugs.python.org/issue?@action=redirect&bpo=1087418]).
- The implementation of `%` checks for the left-side operand being a Python string and special-cases it; this results in a 1–3% performance increase for applications that frequently use `%` with strings, such as templating libraries. (Implemented by Collin Winter; bpo-5176 [https://bugs.python.org/issue?@action=redirect&bpo=5176].)
- List comprehensions with an `if` condition are compiled into faster bytecode. (Patch by Antoine Pitrou, back-ported to 2.7 by Jeffrey Yasskin; bpo-4715 [https://bugs.python.org/issue?@action=redirect&bpo=4715].)
- Converting an integer or long integer to a decimal string was made faster by special-casing base 10 instead of using a generalized conversion function that supports arbitrary bases. (Patch by Gawain Bolton; bpo-6713 [https://bugs.python.org/issue?@action=redirect&bpo=6713].)
- The **`split()`**, **`replace()`**, **`rindex()`**, **`rpartition()`**, and **`rsplit()`** methods of string-like types (strings, Unicode strings, and **`bytearray`** objects) now use a fast reverse-search

algorithm instead of a character-by-character scan. This is sometimes faster by a factor of 10. (Added by Florent Xicluna; bpo-7462 [https://bugs.python.org/issue?@action=redirect&bpo=7462] and bpo-7622 [https://bugs.python.org/issue?@action=redirect&bpo=7622].)

- The **`pickle`** and **`cPickle`** modules now automatically intern the strings used for attribute names, reducing memory usage of the objects resulting from unpickling. (Contributed by Jake McGuire; bpo-5084 [https://bugs.python.org/issue?@action=redirect&bpo=5084].)
- The **`cPickle`** module now special-cases dictionaries, nearly halving the time required to pickle them. (Contributed by Collin Winter; bpo-5670 [https://bugs.python.org/issue?@action=redirect&bpo=5670].)

New and Improved Modules

As in every release, Python’s standard library received a number of enhancements and bug fixes. Here’s a partial list of the most notable changes, sorted alphabetically by module name. Consult the `Misc/NEWS` file in the source tree for a more complete list of changes, or look through the Subversion logs for all the details.

- The **`bdb`** module’s base debugging class **`Bdb`** gained a feature for skipping modules. The constructor now takes an iterable containing glob-style patterns such as `django.*`; the debugger will not step into stack frames from a module that matches one of these patterns. (Contributed by Maru Newby after a suggestion by Senthil Kumaran; bpo-5142 [https://bugs.python.org/issue?@action=redirect&bpo=5142].)

- The **`binascii`** module now supports the buffer API, so it can be used with **`memoryview`** instances and other similar buffer objects. (Backported from 3.x by Florent Xicluna; bpo-7703 [https://bugs.python.org/issue?@action=redirect&bpo=7703].)

- Updated module: the **`bsddb`** module has been updated from 4.7.2devel9 to version 4.8.4 of the pybsddb package [https://www.jcea.es/programacion/pybsddb.htm]. The new version features better Python 3.x compatibility, various bug fixes, and adds several new BerkeleyDB flags and methods. (Updated by Jesús Cea Avión; bpo-8156 [https://bugs.python.org/issue?@action=redirect&bpo=8156]. The pybsddb changelog can be read at https://hg.jcea.es/pybsddb/file/tip/ChangeLog.)

- The **`bz2`** module's **`BZ2File`** now supports the context management protocol, so you can write `with bz2.BZ2File(...) as f:`. (Contributed by Hagen Fürstenau; bpo-3860 [https://bugs.python.org/issue?@action=redirect&bpo=3860].)

- New class: the **`Counter`** class in the **`collections`** module is useful for tallying data. **`Counter`** instances behave mostly like dictionaries but return zero for missing keys instead of raising a **`KeyError`**:

```
>>> from collections import Counter
>>> c = Counter()
>>> for letter in 'here is a sample of english text':
...   c[letter] += 1
...
>>> c
Counter({' ': 6, 'e': 5, 's': 3, 'a': 2, 'i': 2, 'h': 2,
'l': 2, 't': 2, 'g': 1, 'f': 1, 'm': 1, 'o': 1, 'n': 1,
'p': 1, 'r': 1, 'x': 1})
>>> c['e']
5
```

```
>>> c['z']
0
```

There are three additional **`Counter`** methods. **`most_common()`** returns the N most common elements and their counts. **`elements()`** returns an iterator over the contained elements, repeating each element as many times as its count. **`subtract()`** takes an iterable and subtracts one for each element instead of adding; if the argument is a dictionary or another **`Counter`**, the counts are subtracted.

```
>>> c.most_common(5)
[(' ', 6), ('e', 5), ('s', 3), ('a', 2), ('i', 2)]
>>> c.elements() ->
   'a', 'a', ' ', ' ', ' ', ' ', ' ', ' ',
   'e', 'e', 'e', 'e', 'e', 'g', 'f', 'i', 'i',
   'h', 'h', 'm', 'l', 'l', 'o', 'n', 'p', 's',
   's', 's', 'r', 't', 't', 'x'
>>> c['e']
5
>>> c.subtract('very heavy on the letter e')
>>> c['e']    # Count is now lower
-1
```

Contributed by Raymond Hettinger; bpo-1696199 [https://bugs.python.org/issue?@action=redirect&bpo=1696199].

New class: **`OrderedDict`** is described in the earlier section PEP 372: Adding an Ordered Dictionary to collections.

New method: The **`deque`** data type now has a **`count()`** method that returns the number of contained elements equal to the supplied argument *x*, and a **`reverse()`** method that reverses the elements of the deque in-place. **`deque`** also exposes its maximum length as the read-only **`maxlen`** attribute. (Both features added by Raymond Hettinger.)

The **`namedtuple`** class now has an optional *rename* parameter. If *rename* is true, field names that are invalid because they've been repeated or aren't legal Python identifiers will be renamed to legal names that are derived from the field's position within the list of fields:

```
>>> from collections import namedtuple
>>> T = namedtuple('T', ['field1', '$illegal', 'for',
'field2'], rename=True)
>>> T._fields
('field1', '_1', '_2', 'field2')
```

(Added by Raymond Hettinger; bpo-1818 [https://bugs.python.org/issue?@action=redirect&bpo=1818].)

Finally, the **`Mapping`** abstract base class now returns **`NotImplemented`** if a mapping is compared to another type that isn't a **`Mapping`**. (Fixed by Daniel Stutzbach; bpo-8729 [https://bugs.python.org/issue?@action=redirect&bpo=8729].)

- Constructors for the parsing classes in the **`ConfigParser`** module now take an *allow_no_value* parameter, defaulting to false; if true, options without values will be allowed. For example:

```
>>> import ConfigParser, StringIO
>>> sample_config = """
... [mysqld]
... user = mysql
... pid-file = /var/run/mysqld/mysqld.pid
... skip-bdb
... """
>>> config =
ConfigParser.RawConfigParser(allow_no_value=True)
>>> config.readfp(StringIO.StringIO(sample_config))
>>> config.get('mysqld', 'user')
'mysql'
>>> print config.get('mysqld', 'skip-bdb')
None
>>> print config.get('mysqld', 'unknown')
```

```
Traceback (most recent call last):
  ...
NoOptionError: No option 'unknown' in section: 'mysqld'
```

(Contributed by Mats Kindahl; bpo-7005 [https://bugs.python.org/issue?@action=redirect&bpo=7005].)

- Deprecated function: **`contextlib.nested()`**, which allows handling more than one context manager with a single **`with`** statement, has been deprecated, because the **`with`** statement now supports multiple context managers.
- The **`cookielib`** module now ignores cookies that have an invalid version field, one that doesn’t contain an integer value. (Fixed by John J. Lee; bpo-3924 [https://bugs.python.org/issue?@action=redirect&bpo=3924].)
- The **`copy`** module’s **`deepcopy()`** function will now correctly copy bound instance methods. (Implemented by Robert Collins; bpo-1515 [https://bugs.python.org/issue?@action=redirect&bpo=1515].)
- The **`ctypes`** module now always converts `None` to a C `NULL` pointer for arguments declared as pointers. (Changed by Thomas Heller; bpo-4606 [https://bugs.python.org/issue?@action=redirect&bpo=4606].) The underlying libffi library [https://sourceware.org/libffi/] has been updated to version 3.0.9, containing various fixes for different platforms. (Updated by Matthias Klose; bpo-8142 [https://bugs.python.org/issue?@action=redirect&bpo=8142].)
- New method: the **`datetime`** module’s **`timedelta`** class gained a **`total_seconds()`** method that returns the number of seconds in the duration. (Contributed by Brian Quinlan; bpo-5788 [https://bugs.python.org/issue?@action=redirect&bpo=5788].)

- New method: the **`Decimal`** class gained a **`from_float()`** class method that performs an exact conversion of a floating-point number to a **`Decimal`**. This exact conversion strives for the closest decimal approximation to the floating-point representation's value; the resulting decimal value will therefore still include the inaccuracy, if any. For example, `Decimal.from_float(0.1)` returns `Decimal('0.1000000000000000055511151231257827021181583404541015625')`. (Implemented by Raymond Hettinger; bpo-4796 [https://bugs.python.org/issue?@action=redirect&bpo=4796].)

  Comparing instances of **`Decimal`** with floating-point numbers now produces sensible results based on the numeric values of the operands. Previously such comparisons would fall back to Python's default rules for comparing objects, which produced arbitrary results based on their type. Note that you still cannot combine **`Decimal`** and floating point in other operations such as addition, since you should be explicitly choosing how to convert between float and **`Decimal`**. (Fixed by Mark Dickinson; bpo-2531 [https://bugs.python.org/issue?@action=redirect&bpo=2531].)

  The constructor for **`Decimal`** now accepts floating-point numbers (added by Raymond Hettinger; bpo-8257 [https://bugs.python.org/issue?@action=redirect&bpo=8257]) and non-European Unicode characters such as Arabic-Indic digits (contributed by Mark Dickinson; bpo-6595 [https://bugs.python.org/issue?@action=redirect&bpo=6595]).

  Most of the methods of the **`Context`** class now accept integers as well as **`Decimal`** instances; the only exceptions are the **`canonical()`** and **`is_canonical()`** methods. (Patch by Juan José Conti; bpo-7633 [https://bugs.python.org/issue?@action=redirect&bpo=7633].)

When using **`Decimal`** instances with a string's **`format()`** method, the default alignment was previously left-alignment. This has been changed to right-alignment, which is more sensible for numeric types. (Changed by Mark Dickinson; bpo-6857 [https://bugs.python.org/issue?@action=redirect&bpo=6857].)

Comparisons involving a signaling NaN value (or `sNAN`) now signal **`InvalidOperation`** instead of silently returning a true or false value depending on the comparison operator. Quiet NaN values (or `NaN`) are now hashable. (Fixed by Mark Dickinson; bpo-7279 [https://bugs.python.org/issue?@action=redirect&bpo=7279].)

- The **`difflib`** module now produces output that is more compatible with modern **diff**/**patch** tools through one small change, using a tab character instead of spaces as a separator in the header giving the filename. (Fixed by Anatoly Techtonik; bpo-7585 [https://bugs.python.org/issue?@action=redirect&bpo=7585].)

- The Distutils `sdist` command now always regenerates the `MANIFEST` file, since even if the `MANIFEST.in` or `setup.py` files haven't been modified, the user might have created some new files that should be included. (Fixed by Tarek Ziadé; bpo-8688 [https://bugs.python.org/issue?@action=redirect&bpo=8688].)

- The **`doctest`** module's **`IGNORE_EXCEPTION_DETAIL`** flag will now ignore the name of the module containing the exception being tested. (Patch by Lennart Regebro; bpo-7490 [https://bugs.python.org/issue?@action=redirect&bpo=7490].)

- The **`email`** module's **`Message`** class will now accept a Unicode-valued payload, automatically converting the payload to the encoding specified by **`output_charset`**.

(Added by R. David Murray; bpo-1368247 [https://bugs.python.org/issue?@action=redirect&bpo=1368247].)

- The **`Fraction`** class now accepts a single float or **`Decimal`** instance, or two rational numbers, as arguments to its constructor. (Implemented by Mark Dickinson; rationals added in bpo-5812 [https://bugs.python.org/issue?@action=redirect&bpo=5812], and float/decimal in bpo-8294 [https://bugs.python.org/issue?@action=redirect&bpo=8294].)

  Ordering comparisons (<, <=, >, >=) between fractions and complex numbers now raise a **`TypeError`**. This fixes an oversight, making the **`Fraction`** match the other numeric types.

- New class: **`FTP_TLS`** in the **`ftplib`** module provides secure FTP connections using TLS encapsulation of authentication as well as subsequent control and data transfers. (Contributed by Giampaolo Rodola; bpo-2054 [https://bugs.python.org/issue?@action=redirect&bpo=2054].)

  The **`storbinary()`** method for binary uploads can now restart uploads thanks to an added *rest* parameter (patch by Pablo Mouzo; bpo-6845 [https://bugs.python.org/issue?@action=redirect&bpo=6845].)

- New class decorator: **`total_ordering()`** in the **`functools`** module takes a class that defines an **`__eq__()`** method and one of **`__lt__()`**, **`__le__()`**, **`__gt__()`**, or **`__ge__()`**, and generates the missing comparison methods. Since the **`__cmp__()`** method is being deprecated in Python 3.x, this decorator makes it easier to define ordered classes. (Added by Raymond Hettinger; bpo-5479 [https://bugs.python.org/issue?@action=redirect&bpo=5479].)

  New function: **`cmp_to_key()`** will take an old-style comparison function that expects two arguments and return a new callable that can be used as the *key* parameter to functions such as **`sorted()`**, **`min()`** and **`max()`**, etc. The primary intended use is to help with making code compatible with Python 3.x. (Added by Raymond Hettinger.)

- New function: the **`gc`** module's **`is_tracked()`** returns true if a given instance is tracked by the garbage collector, false otherwise. (Contributed by Antoine Pitrou; bpo-4688 [https://bugs.python.org/issue?@action=redirect&bpo=4688].)
- The **`gzip`** module's **`GzipFile`** now supports the context management protocol, so you can write `with gzip.GzipFile(...) as f:` (contributed by Hagen Fürstenau; bpo-3860 [https://bugs.python.org/issue?@action=redirect&bpo=3860]), and it now implements the **`io.BufferedIOBase`** ABC, so you can wrap it with **`io.BufferedReader`** for faster processing (contributed by Nir Aides; bpo-7471 [https://bugs.python.org/issue?@action=redirect&bpo=7471]). It's also now possible to override the modification time recorded in a gzipped file by providing an optional timestamp to the constructor. (Contributed by Jacques Frechet; bpo-4272 [https://bugs.python.org/issue?@action=redirect&bpo=4272].)

  Files in gzip format can be padded with trailing zero bytes; the **`gzip`** module will now consume these trailing bytes. (Fixed by Tadek Pietraszek and Brian Curtin; bpo-2846 [https://bugs.python.org/issue?@action=redirect&bpo=2846].)
- New attribute: the **`hashlib`** module now has an **`algorithms`** attribute containing a tuple naming the supported algorithms. In Python 2.7, `hashlib.algorithms` contains `('md5', 'sha1', 'sha224', 'sha256', 'sha384', 'sha512')`. (Contributed by Carl Chenet; bpo-7418 [https://bugs.python.org/issue?@action=redirect&bpo=7418].)

- The default **HTTPResponse** class used by the **httplib** module now supports buffering, resulting in much faster reading of HTTP responses. (Contributed by Kristján Valur Jónsson; bpo-4879 [https://bugs.python.org/issue?@action=redirect&bpo=4879].)

  The **HTTPConnection** and **HTTPSConnection** classes now support a *source_address* parameter, a `(host, port)` 2-tuple giving the source address that will be used for the connection. (Contributed by Eldon Ziegler; bpo-3972 [https://bugs.python.org/issue?@action=redirect&bpo=3972].)

- The **ihooks** module now supports relative imports. Note that **ihooks** is an older module for customizing imports, superseded by the **imputil** module added in Python 2.0. (Relative import support added by Neil Schemenauer.)

- The **imaplib** module now supports IPv6 addresses. (Contributed by Derek Morr; bpo-1655 [https://bugs.python.org/issue?@action=redirect&bpo=1655].)

- New function: the **inspect** module's **getcallargs()** takes a callable and its positional and keyword arguments, and figures out which of the callable's parameters will receive each argument, returning a dictionary mapping argument names to their values. For example:

```
>>> from inspect import getcallargs
>>> def f(a, b=1, *pos, **named):
...     pass
...
>>> getcallargs(f, 1, 2, 3)
{'a': 1, 'b': 2, 'pos': (3,), 'named': {}}
>>> getcallargs(f, a=2, x=4)
{'a': 2, 'b': 1, 'pos': (), 'named': {'x': 4}}
>>> getcallargs(f)
Traceback (most recent call last):
```

```
...
TypeError: f() takes at least 1 argument (0 given)
```

Contributed by George Sakkis; bpo-3135 [https://bugs.python.org/issue?@action=redirect&bpo=3135].

- Updated module: The **`io`** library has been upgraded to the version shipped with Python 3.1. For 3.1, the I/O library was entirely rewritten in C and is 2 to 20 times faster depending on the task being performed. The original Python version was renamed to the **`_pyio`** module.

  One minor resulting change: the **`io.TextIOBase`** class now has an **`errors`** attribute giving the error setting used for encoding and decoding errors (one of `'strict'`, `'replace'`, `'ignore'`).

  The **`io.FileIO`** class now raises an **`OSError`** when passed an invalid file descriptor. (Implemented by Benjamin Peterson; bpo-4991 [https://bugs.python.org/issue?@action=redirect&bpo=4991].) The **`truncate()`** method now preserves the file position; previously it would change the file position to the end of the new file. (Fixed by Pascal Chambon; bpo-6939 [https://bugs.python.org/issue?@action=redirect&bpo=6939].)

- New function: `itertools.compress(data, selectors)` takes two iterators. Elements of *data* are returned if the corresponding value in *selectors* is true:

  ```
  itertools.compress('ABCDEF', [1,0,1,0,1,1]) =>
    A, C, E, F
  ```

  New function: `itertools.combinations_with_replacement(iter, r)` returns all the possible *r*-length combinations of elements from the iterable *iter*. Unlike **`combinations()`**, individual elements can be repeated in the generated combinations:

```
itertools.combinations_with_replacement('abc', 2) =>
  ('a', 'a'), ('a', 'b'), ('a', 'c'),
  ('b', 'b'), ('b', 'c'), ('c', 'c')
```

Note that elements are treated as unique depending on their position in the input, not their actual values.

The **`itertools.count()`** function now has a *step* argument that allows incrementing by values other than 1. **`count()`** also now allows keyword arguments, and using non-integer values such as floats or **`Decimal`** instances. (Implemented by Raymond Hettinger; bpo-5032 [https://bugs.python.org/issue?@action=redirect&bpo=5032].)

`itertools.combinations()` and **`itertools.product()`** previously raised **`ValueError`** for values of *r* larger than the input iterable. This was deemed a specification error, so they now return an empty iterator. (Fixed by Raymond Hettinger; bpo-4816 [https://bugs.python.org/issue?@action=redirect&bpo=4816].)

- Updated module: The **`json`** module was upgraded to version 2.0.9 of the simplejson package, which includes a C extension that makes encoding and decoding faster. (Contributed by Bob Ippolito; bpo-4136 [https://bugs.python.org/issue?@action=redirect&bpo=4136].)

  To support the new **`collections.OrderedDict`** type, **`json.load()`** now has an optional *object_pairs_hook* parameter that will be called with any object literal that decodes to a list of pairs. (Contributed by Raymond Hettinger; bpo-5381 [https://bugs.python.org/issue?@action=redirect&bpo=5381].)

- The `mailbox` module's `Maildir` class now records the timestamp on the directories it reads, and only re-reads them if the modification time has subsequently changed. This improves performance by avoiding unneeded directory scans. (Fixed by A.M. Kuchling and Antoine Pitrou; bpo-1607951 [https://bugs.python.org/issue?@action=redirect&bpo=1607951], bpo-6896 [https://bugs.python.org/issue?@action=redirect&bpo=6896].)
- New functions: the `math` module gained `erf()` and `erfc()` for the error function and the complementary error function, `expm1()` which computes `e**x - 1` with more precision than using `exp()` and subtracting 1, `gamma()` for the Gamma function, and `lgamma()` for the natural log of the Gamma function. (Contributed by Mark Dickinson and nirinA raseliarison; bpo-3366 [https://bugs.python.org/issue?@action=redirect&bpo=3366].)
- The `multiprocessing` module's `Manager*` classes can now be passed a callable that will be called whenever a subprocess is started, along with a set of arguments that will be passed to the callable. (Contributed by lekma; bpo-5585 [https://bugs.python.org/issue?@action=redirect&bpo=5585].)

  The `Pool` class, which controls a pool of worker processes, now has an optional *maxtasksperchild* parameter. Worker processes will perform the specified number of tasks and then exit, causing the `Pool` to start a new worker. This is useful if tasks may leak memory or other resources, or if some tasks will cause the worker to become very large. (Contributed by Charles Cazabon; bpo-6963 [https://bugs.python.org/issue?@action=redirect&bpo=6963].)

- The **nntplib** module now supports IPv6 addresses. (Contributed by Derek Morr; bpo-1664 [https://bugs.python.org/issue?@action=redirect&bpo=1664].)

- New functions: the `os` module wraps the following POSIX system calls: `getresgid()` and `getresuid()`, which return the real, effective, and saved GIDs and UIDs; `setresgid()` and `setresuid()`, which set real, effective, and saved GIDs and UIDs to new values; `initgroups()`, which initialize the group access list for the current process. (GID/UID functions contributed by Travis H.; bpo-6508 [https://bugs.python.org/issue?@action=redirect&bpo=6508]. Support for initgroups added by Jean-Paul Calderone; bpo-7333 [https://bugs.python.org/issue?@action=redirect&bpo=7333].)

  The `os.fork()` function now re-initializes the import lock in the child process; this fixes problems on Solaris when `fork()` is called from a thread. (Fixed by Zsolt Cserna; bpo-7242 [https://bugs.python.org/issue?@action=redirect&bpo=7242].)

- In the `os.path` module, the `normpath()` and `abspath()` functions now preserve Unicode; if their input path is a Unicode string, the return value is also a Unicode string. (`normpath()` fixed by Matt Giuca in bpo-5827 [https://bugs.python.org/issue?@action=redirect&bpo=5827]; `abspath()` fixed by Ezio Melotti in bpo-3426 [https://bugs.python.org/issue?@action=redirect&bpo=3426].)

- The `pydoc` module now has help for the various symbols that Python uses. You can now do `help('<<')` or `help('@')`, for example. (Contributed by David Laban; bpo-4739 [https://bugs.python.org/issue?@action=redirect&bpo=4739].)

- The `re` module's `split()`, `sub()`, and `subn()` now accept an optional *flags* argument, for consistency with the other functions in the module. (Added by Gregory P. Smith.)

- New function: `run_path()` in the `runpy` module will execute the code at a provided *path* argument. *path* can be the path of a Python source file (`example.py`), a compiled bytecode file (`example.pyc`), a directory (`./package/`), or a zip archive (`example.zip`). If a directory or zip path is provided, it will be added to the front of `sys.path` and the module `__main__` will be imported. It's expected that the directory or zip contains a `__main__.py`; if it doesn't, some other `__main__.py` might be imported from a location later in `sys.path`. This makes more of the machinery of `runpy` available to scripts that want to mimic the way Python's command line processes an explicit path name. (Added by Nick Coghlan; bpo-6816 [https://bugs.python.org/issue?@action=redirect&bpo=6816].)
- New function: in the `shutil` module, `make_archive()` takes a filename, archive type (zip or tar-format), and a directory path, and creates an archive containing the directory's contents. (Added by Tarek Ziadé.)

  `shutil`'s `copyfile()` and `copytree()` functions now raise a `SpecialFileError` exception when asked to copy a named pipe. Previously the code would treat named pipes like a regular file by opening them for reading, and this would block indefinitely. (Fixed by Antoine Pitrou; bpo-3002 [https://bugs.python.org/issue?@action=redirect&bpo=3002].)
- The `signal` module no longer re-installs the signal handler unless this is truly necessary, which fixes a bug that could make it impossible to catch the EINTR signal robustly. (Fixed by Charles-Francois Natali; bpo-8354 [https://bugs.python.org/issue?@action=redirect&bpo=8354].)

- New functions: in the `site` module, three new functions return various site- and user-specific paths. `getsitepackages()` returns a list containing all global site-packages directories, `getusersitepackages()` returns the path of the user's site-packages directory, and `getuserbase()` returns the value of the `USER_BASE` environment variable, giving the path to a directory that can be used to store data. (Contributed by Tarek Ziadé; bpo-6693 [https://bugs.python.org/issue?@action=redirect&bpo=6693].)

  The `site` module now reports exceptions occurring when the `sitecustomize` module is imported, and will no longer catch and swallow the `KeyboardInterrupt` exception. (Fixed by Victor Stinner; bpo-3137 [https://bugs.python.org/issue?@action=redirect&bpo=3137].)

- The `create_connection()` function gained a *source_address* parameter, a `(host, port)` 2-tuple giving the source address that will be used for the connection. (Contributed by Eldon Ziegler; bpo-3972 [https://bugs.python.org/issue?@action=redirect&bpo=3972].)

  The `recv_into()` and `recvfrom_into()` methods will now write into objects that support the buffer API, most usefully the `bytearray` and `memoryview` objects. (Implemented by Antoine Pitrou; bpo-8104 [https://bugs.python.org/issue?@action=redirect&bpo=8104].)

- The `SocketServer` module's `TCPServer` class now supports socket timeouts and disabling the Nagle algorithm. The **`disable_nagle_algorithm`** class attribute defaults to `False`; if overridden to be true, new request connections will have the TCP_NODELAY option set to prevent buffering many small sends into a single TCP packet. The `timeout` class attribute can hold a timeout in seconds that will be applied to the request socket; if no request is received within that time, **`handle_timeout()`** will be called and **`handle_request()`** will return. (Contributed by Kristján Valur Jónsson; bpo-6192 [https://bugs.python.org/issue?@action=redirect&bpo=6192] and bpo-6267 [https://bugs.python.org/issue?@action=redirect&bpo=6267].)

- Updated module: the **`sqlite3`** module has been updated to version 2.6.0 of the pysqlite package [https://github.com/ghaering/pysqlite]. Version 2.6.0 includes a number of bugfixes, and adds the ability to load SQLite extensions from shared libraries. Call the `enable_load_extension(True)` method to enable extensions, and then call **`load_extension()`** to load a particular shared library. (Updated by Gerhard Häring.)

- The **`ssl`** module's **`SSLSocket`** objects now support the buffer API, which fixed a test suite failure (fix by Antoine Pitrou; bpo-7133 [https://bugs.python.org/issue?@action=redirect&bpo=7133]) and automatically set OpenSSL's **`SSL_MODE_AUTO_RETRY`**, which will prevent an error code being returned from **`recv()`** operations that trigger an SSL renegotiation (fix by Antoine Pitrou; bpo-8222 [https://bugs.python.org/issue?@action=redirect&bpo=8222]).

  The **`wrap_socket()`** constructor function now takes a *ciphers* argument that's a string listing the encryption algorithms to be allowed; the format of the string is described in the OpenSSL documentation [https://docs.openssl.org/1.0.2/man1/ciphers/]. (Added by Antoine Pitrou; bpo-8322 [https://bugs.python.org/issue?@action=redirect&bpo=8322].)

  Another change makes the extension load all of OpenSSL's ciphers and digest algorithms so that they're all available. Some SSL certificates couldn't be verified, reporting an “unknown algorithm” error. (Reported by Beda Kosata, and fixed by Antoine Pitrou; bpo-8484 [https://bugs.python.org/issue?@action=redirect&bpo=8484].)

The version of OpenSSL being used is now available as the module attributes **`ssl.OPENSSL_VERSION`** (a string), **`ssl.OPENSSL_VERSION_INFO`** (a 5-tuple), and **`ssl.OPENSSL_VERSION_NUMBER`** (an integer). (Added by Antoine Pitrou; bpo-8321 [https://bugs.python.org/issue?@action=redirect&bpo=8321].)

- The **`struct`** module will no longer silently ignore overflow errors when a value is too large for a particular integer format code (one of `bBhHiIlLqQ`); it now always raises a **`struct.error`** exception. (Changed by Mark Dickinson; bpo-1523 [https://bugs.python.org/issue?@action=redirect&bpo=1523].) The **`pack()`** function will also attempt to use **`__index__()`** to convert and pack non-integers before trying the **`__int__()`** method or reporting an error. (Changed by Mark Dickinson; bpo-8300 [https://bugs.python.org/issue?@action=redirect&bpo=8300].)

- New function: the **`subprocess`** module’s **`check_output()`** runs a command with a specified set of arguments and returns the command’s output as a string when the command runs without error, or raises a **`CalledProcessError`** exception otherwise.

  ```
  >>> subprocess.check_output(['df', '-h', '.'])
  'Filesystem     Size   Used  Avail Capacity  Mounted on\n
  /dev/disk0s2    52G    49G   3.0G    94%    /\n'

  >>> subprocess.check_output(['df', '-h', '/bogus'])
    ...
  subprocess.CalledProcessError: Command '['df', '-h',
  '/bogus']' returned non-zero exit status 1
  ```

  (Contributed by Gregory P. Smith.)

The **`subprocess`** module will now retry its internal system calls on receiving an **`EINTR`** signal. (Reported by several people; final patch by Gregory P. Smith in bpo-1068268 [https://bugs.python.org/issue?@action=redirect&bpo=1068268].)

- New function: **`is_declared_global()`** in the **`symtable`** module returns true for variables that are explicitly declared to be global, false for ones that are implicitly global. (Contributed by Jeremy Hylton.)
- The **`syslog`** module will now use the value of `sys.argv[0]` as the identifier instead of the previous default value of `'python'`. (Changed by Sean Reifschneider; bpo-8451 [https://bugs.python.org/issue?@action=redirect&bpo=8451].)
- The **`sys.version_info`** value is now a named tuple, with attributes named **`major`**, **`minor`**, **`micro`**, **`releaselevel`**, and **`serial`**. (Contributed by Ross Light; bpo-4285 [https://bugs.python.org/issue?@action=redirect&bpo=4285].)

  `sys.getwindowsversion()` also returns a named tuple, with attributes named **`major`**, **`minor`**, **`build`**, **`platform`**, **`service_pack`**, **`service_pack_major`**, **`service_pack_minor`**, **`suite_mask`**, and **`product_type`**. (Contributed by Brian Curtin; bpo-7766 [https://bugs.python.org/issue?@action=redirect&bpo=7766].)
- The **`tarfile`** module's default error handling has changed, to no longer suppress fatal errors. The default error level was previously 0, which meant that errors would only result in a message being written to the debug log, but because the debug log is not activated by default, these errors go unnoticed. The default error level is now 1, which raises an exception if there's an error. (Changed by Lars Gustäbel; bpo-7357 [https://bugs.python.org/issue?@action=redirect&bpo=7357].)

`tarfile` now supports filtering the **`TarInfo`** objects being added to a tar file. When you call **`add()`**, you may supply an optional *filter* argument that's a callable. The *filter* callable will be passed the **`TarInfo`** for every file being added, and can modify and return it. If the callable returns `None`, the file will be excluded from the resulting archive. This is more powerful than the existing *exclude* argument, which has therefore been deprecated. (Added by Lars Gustäbel; bpo-6856 [https://bugs.python.org/issue?@action=redirect&bpo=6856].) The **`TarFile`** class also now supports the context management protocol. (Added by Lars Gustäbel; bpo-7232 [https://bugs.python.org/issue?@action=redirect&bpo=7232].)

- The **`wait()`** method of the **`threading.Event`** class now returns the internal flag on exit. This means the method will usually return true because **`wait()`** is supposed to block until the internal flag becomes true. The return value will only be false if a timeout was provided and the operation timed out. (Contributed by Tim Lesher; bpo-1674032 [https://bugs.python.org/issue?@action=redirect&bpo=1674032].)

- The Unicode database provided by the **`unicodedata`** module is now used internally to determine which characters are numeric, whitespace, or represent line breaks. The database also includes information from the `Unihan.txt` data file (patch by Anders Chrigström and Amaury Forgeot d'Arc; bpo-1571184 [https://bugs.python.org/issue?@action=redirect&bpo=1571184]) and has been updated to version 5.2.0 (updated by Florent Xicluna; bpo-8024 [https://bugs.python.org/issue?@action=redirect&bpo=8024]).

- The **`urlparse`** module's **`urlsplit()`** now handles unknown URL schemes in a fashion compliant with **RFC 3986** [https://datatracker.ietf.org/doc/html/rfc3986.html]: if the URL is of the form `"<something>://..."`, the text before the `://` is treated as the scheme, even if it's a made-up scheme that the module doesn't know about. This change may break code that worked around the old behaviour. For example, Python 2.6.4 or 2.5 will return the following:

```
>>> import urlparse
>>> urlparse.urlsplit('invented://host/filename?query')
('invented', '', '//host/filename?query', '', '')
```

Python 2.7 (and Python 2.6.5) will return:

```
>>> import urlparse
>>> urlparse.urlsplit('invented://host/filename?query')
('invented', 'host', '/filename?query', '', '')
```

(Python 2.7 actually produces slightly different output, since it returns a named tuple instead of a standard tuple.)

The **`urlparse`** module also supports IPv6 literal addresses as defined by **RFC 2732** [https://datatracker.ietf.org/doc/html/rfc2732.html] (contributed by Senthil Kumaran; bpo-2987 [https://bugs.python.org/issue?@action=redirect&bpo=2987]).

```
>>> urlparse.urlparse('http://[1080::8:800:200C:417A]/foo')
ParseResult(scheme='http',
netloc='[1080::8:800:200C:417A]',
            path='/foo', params='', query='', fragment='')
```

- New class: the **`WeakSet`** class in the **`weakref`** module is a set that only holds weak references to its elements; elements will be removed once there are no references pointing to them. (Originally implemented in Python 3.x by Raymond Hettinger, and backported to 2.7 by Michael Foord.)

- The `xml.etree.ElementTree` library, no longer escapes ampersands and angle brackets when outputting an XML processing instruction (which looks like `<?xml-stylesheet href="#style1"?>`) or comment (which looks like `<!-- comment -->`). (Patch by Neil Muller; bpo-2746 [https://bugs.python.org/issue?@action=redirect&bpo=2746].)
- The XML-RPC client and server, provided by the `xmlrpclib` and `SimpleXMLRPCServer` modules, have improved performance by supporting HTTP/1.1 keep-alive and by optionally using gzip encoding to compress the XML being exchanged. The gzip compression is controlled by the **`encode_threshold`** attribute of `SimpleXMLRPCRequestHandler`, which contains a size in bytes; responses larger than this will be compressed. (Contributed by Kristján Valur Jónsson; bpo-6267 [https://bugs.python.org/issue?@action=redirect&bpo=6267].)
- The `zipfile` module's `ZipFile` now supports the context management protocol, so you can write `with zipfile.ZipFile(...) as f:`. (Contributed by Brian Curtin; bpo-5511 [https://bugs.python.org/issue?@action=redirect&bpo=5511].)

  `zipfile` now also supports archiving empty directories and extracts them correctly. (Fixed by Kuba Wieczorek; bpo-4710 [https://bugs.python.org/issue?@action=redirect&bpo=4710].) Reading files out of an archive is faster, and interleaving `read()` and `readline()` now works correctly. (Contributed by Nir Aides; bpo-7610 [https://bugs.python.org/issue?@action=redirect&bpo=7610].)

  The `is_zipfile()` function now accepts a file object, in addition to the path names accepted in earlier versions. (Contributed by Gabriel Genellina; bpo-4756 [https://bugs.python.org/issue?@action=redirect&bpo=4756].)

The `writestr()` method now has an optional *compress_type* parameter that lets you override the default compression method specified in the `ZipFile` constructor. (Contributed by Ronald Oussoren; bpo-6003 [https://bugs.python.org/issue?@action=redirect&bpo=6003].)

New module: importlib

Python 3.1 includes the `importlib` package, a re-implementation of the logic underlying Python's `import` statement. `importlib` is useful for implementers of Python interpreters and to users who wish to write new importers that can participate in the import process. Python 2.7 doesn't contain the complete `importlib` package, but instead has a tiny subset that contains a single function, `import_module()`.

`import_module(name, package=None)` imports a module. *name* is a string containing the module or package's name. It's possible to do relative imports by providing a string that begins with a `.` character, such as `..utils.errors`. For relative imports, the *package* argument must be provided and is the name of the package that will be used as the anchor for the relative import. `import_module()` both inserts the imported module into `sys.modules` and returns the module object.

Here are some examples:

```
>>> from importlib import import_module
>>> anydbm = import_module('anydbm')  # Standard absolute
import
>>> anydbm
<module 'anydbm' from '/p/python/Lib/anydbm.py'>
>>> # Relative import
>>> file_util = import_module('..file_util',
'distutils.command')
>>> file_util
```

```
<module 'distutils.file_util' from
'/python/Lib/distutils/file_util.pyc'>
```

`importlib` was implemented by Brett Cannon and introduced in Python 3.1.

New module: sysconfig

The `sysconfig` module has been pulled out of the Distutils package, becoming a new top-level module in its own right. `sysconfig` provides functions for getting information about Python’s build process: compiler switches, installation paths, the platform name, and whether Python is running from its source directory.

Some of the functions in the module are:

- `get_config_var()` returns variables from Python’s Makefile and the `pyconfig.h` file.
- `get_config_vars()` returns a dictionary containing all of the configuration variables.
- `get_path()` returns the configured path for a particular type of module: the standard library, site-specific modules, platform-specific modules, etc.
- `is_python_build()` returns true if you’re running a binary from a Python source tree, and false otherwise.

Consult the `sysconfig` documentation for more details and for a complete list of functions.

The Distutils package and `sysconfig` are now maintained by Tarek Ziadé, who has also started a Distutils2 package (source repository at https://hg.python.org/distutils2/) for developing a next-generation version of Distutils.

ttk: Themed Widgets for Tk

Tcl/Tk 8.5 includes a set of themed widgets that re-implement basic Tk widgets but have a more customizable appearance and can therefore more closely resemble the native platform's widgets. This widget set was originally called Tile, but was renamed to Ttk (for "themed Tk") on being added to Tcl/Tck release 8.5.

To learn more, read the **`ttk`** module documentation. You may also wish to read the Tcl/Tk manual page describing the Ttk theme engine, available at https://www.tcl.tk/man/tcl8.5/TkCmd/ttk_intro.html. Some screenshots of the Python/Ttk code in use are at https://code.google.com/archive/p/python-ttk/wikis/Screenshots.wiki.

The **`tkinter.ttk`** module was written by Guilherme Polo and added in bpo-2983 [https://bugs.python.org/issue?@action=redirect&bpo=2983]. An alternate version called `Tile.py`, written by Martin Franklin and maintained by Kevin Walzer, was proposed for inclusion in bpo-2618 [https://bugs.python.org/issue?@action=redirect&bpo=2618], but the authors argued that Guilherme Polo's work was more comprehensive.

Updated module: unittest

The **`unittest`** module was greatly enhanced; many new features were added. Most of these features were implemented by Michael Foord, unless otherwise noted. The enhanced version of the module is downloadable separately for use with Python versions 2.4 to 2.6, packaged as the **`unittest2`** package, from unittest2 [https://pypi.org/project/unittest2/].

When used from the command line, the module can automatically discover tests. It's not as fancy as py.test [https://pytest.org] or nose [https://nose.readthedocs.io/], but provides a simple way to run tests kept within a set of package directories. For example, the following command will search the `test/` subdirectory for any importable test files named `test*.py`:

```
python -m unittest discover -s test
```

Consult the **`unittest`** module documentation for more details. (Developed in bpo-6001 [https://bugs.python.org/issue?@action=redirect&bpo=6001].)

The **`main()`** function supports some other new options:

- **`-b`** or **`--buffer`** will buffer the standard output and standard error streams during each test. If the test passes, any resulting output will be discarded; on failure, the buffered output will be displayed.
- **`-c`** or **`--catch`** will cause the control-C interrupt to be handled more gracefully. Instead of interrupting the test process immediately, the currently running test will be completed and then the partial results up to the interruption will be reported. If you're impatient, a second press of control-C will cause an immediate interruption.

  This control-C handler tries to avoid causing problems when the code being tested or the tests being run have defined a signal handler of their own, by noticing that a signal handler was already set and calling it. If this doesn't work for you, there's a **`removeHandler()`** decorator that can be used to mark tests that should have the control-C handling disabled.
- **`-f`** or **`--failfast`** makes test execution stop immediately when a test fails instead of continuing to execute further tests. (Suggested by Cliff Dyer and implemented by

Michael Foord; bpo-8074 [https://bugs.python.org/issue?@action=redirect&bpo=8074].)

The progress messages now show ‘x’ for expected failures and ‘u’ for unexpected successes when run in verbose mode. (Contributed by Benjamin Peterson.)

Test cases can raise the `SkipTest` exception to skip a test (bpo-1034053 [https://bugs.python.org/issue?@action=redirect&bpo=1034053]).

The error messages for `assertEqual()`, `assertTrue()`, and `assertFalse()` failures now provide more information. If you set the `longMessage` attribute of your `TestCase` classes to true, both the standard error message and any additional message you provide will be printed for failures. (Added by Michael Foord; bpo-5663 [https://bugs.python.org/issue?@action=redirect&bpo=5663].)

The `assertRaises()` method now returns a context handler when called without providing a callable object to run. For example, you can write this:

```
with self.assertRaises(KeyError):
    {}['foo']
```

(Implemented by Antoine Pitrou; bpo-4444 [https://bugs.python.org/issue?@action=redirect&bpo=4444].)

Module- and class-level setup and teardown fixtures are now supported. Modules can contain `setUpModule()` and `tearDownModule()` functions. Classes can have `setUpClass()` and `tearDownClass()` methods that must be defined as class methods (using `@classmethod` or equivalent). These functions and methods are invoked when the test runner switches to a test case in a different module or class.

The methods `addCleanup()` and `doCleanups()` were added. `addCleanup()` lets you add cleanup functions that will be called unconditionally (after `setUp()` if `setUp()` fails, otherwise after `tearDown()`). This allows for much simpler resource allocation and deallocation during tests (bpo-5679 [https://bugs.python.org/issue?@action=redirect&bpo=5679]).

A number of new methods were added that provide more specialized tests. Many of these methods were written by Google engineers for use in their test suites; Gregory P. Smith, Michael Foord, and GvR worked on merging them into Python's version of `unittest`.

- `assertIsNone()` and `assertIsNotNone()` take one expression and verify that the result is or is not `None`.
- `assertIs()` and `assertIsNot()` take two values and check whether the two values evaluate to the same object or not. (Added by Michael Foord; bpo-2578 [https://bugs.python.org/issue?@action=redirect&bpo=2578].)
- `assertIsInstance()` and `assertNotIsInstance()` check whether the resulting object is an instance of a particular class, or of one of a tuple of classes. (Added by Georg Brandl; bpo-7031 [https://bugs.python.org/issue?@action=redirect&bpo=7031].)
- `assertGreater()`, `assertGreaterEqual()`, `assertLess()`, and `assertLessEqual()` compare two quantities.
- `assertMultiLineEqual()` compares two strings, and if they're not equal, displays a helpful comparison that highlights the differences in the two strings. This comparison is now used by default when Unicode strings are compared with `assertEqual()`.
- `assertRegexpMatches()` and `assertNotRegexpMatches()` checks whether the first argument is a string matching or not matching the regular expression provided as the second argument (bpo-8038 [https://bugs.python.org/issue?@action=redirect&bpo=8038]).

- `assertRaisesRegexp()` checks whether a particular exception is raised, and then also checks that the string representation of the exception matches the provided regular expression.
- `assertIn()` and `assertNotIn()` tests whether *first* is or is not in *second*.
- `assertItemsEqual()` tests whether two provided sequences contain the same elements.
- `assertSetEqual()` compares whether two sets are equal, and only reports the differences between the sets in case of error.
- Similarly, `assertListEqual()` and `assertTupleEqual()` compare the specified types and explain any differences without necessarily printing their full values; these methods are now used by default when comparing lists and tuples using `assertEqual()`. More generally, `assertSequenceEqual()` compares two sequences and can optionally check whether both sequences are of a particular type.
- `assertDictEqual()` compares two dictionaries and reports the differences; it's now used by default when you compare two dictionaries using `assertEqual()`. **`assertDictContainsSubset()`** checks whether all of the key/value pairs in *first* are found in *second*.
- `assertAlmostEqual()` and `assertNotAlmostEqual()` test whether *first* and *second* are approximately equal. This method can either round their difference to an optionally specified number of *places* (the default is 7) and compare it to zero, or require the difference to be smaller than a supplied *delta* value.
- `loadTestsFromName()` properly honors the `suiteClass` attribute of the `TestLoader`. (Fixed by Mark Roddy; bpo-6866 [https://bugs.python.org/issue?@action=redirect&bpo=6866].)
- A new hook lets you extend the `assertEqual()` method to handle new data types. The `addTypeEqualityFunc()` method takes a type object and a function. The function

will be used when both of the objects being compared are of the specified type. This function should compare the two objects and raise an exception if they don’t match; it’s a good idea for the function to provide additional information about why the two objects aren’t matching, much as the new sequence comparison methods do.

`unittest.main()` now takes an optional `exit` argument. If false, `main()` doesn’t call `sys.exit()`, allowing `main()` to be used from the interactive interpreter. (Contributed by J. Pablo Fernández; bpo-3379 [https://bugs.python.org/issue?@action=redirect&bpo=3379].)

`TestResult` has new `startTestRun()` and `stopTestRun()` methods that are called immediately before and after a test run. (Contributed by Robert Collins; bpo-5728 [https://bugs.python.org/issue?@action=redirect&bpo=5728].)

With all these changes, the `unittest.py` was becoming awkwardly large, so the module was turned into a package and the code split into several files (by Benjamin Peterson). This doesn’t affect how the module is imported or used.

See also

https://web.archive.org/web/20210619163128/http://www.voidspace.org.uk/python/articles/unittest2.shtml

Describes the new features, how to use them, and the rationale for various design decisions. (By Michael Foord.)

Updated module: ElementTree 1.3

The version of the ElementTree library included with Python was updated to version 1.3. Some of the new features are:

- The various parsing functions now take a *parser* keyword argument giving an `XMLParser` instance that will be used. This makes it possible to override the file's internal encoding:

  ```
  p = ET.XMLParser(encoding='utf-8')
  t = ET.XML("""<root/>""", parser=p)
  ```

  Errors in parsing XML now raise a `ParseError` exception, whose instances have a **`position`** attribute containing a (*line*, *column*) tuple giving the location of the problem.

- ElementTree's code for converting trees to a string has been significantly reworked, making it roughly twice as fast in many cases. The `ElementTree.write()` and **`Element.write()`** methods now have a *method* parameter that can be "xml" (the default), "html", or "text". HTML mode will output empty elements as `<empty></empty>` instead of `<empty/>`, and text mode will skip over elements and only output the text chunks. If you set the `tag` attribute of an element to `None` but leave its children in place, the element will be omitted when the tree is written out, so you don't need to do more extensive rearrangement to remove a single element.

  Namespace handling has also been improved. All `xmlns:<whatever>` declarations are now output on the root element, not scattered throughout the resulting XML. You can set the default namespace for a tree by setting the **`default_namespace`** attribute and can register new prefixes with `register_namespace()`. In XML mode, you can use the true/false *xml_declaration* parameter to suppress the XML declaration.

- New `Element` method: `extend()` appends the items from a sequence to the element’s children. Elements themselves behave like sequences, so it’s easy to move children from one element to another:

  ```
  from xml.etree import ElementTree as ET

  t = ET.XML("""<list>
    <item>1</item> <item>2</item>  <item>3</item>
  </list>""")
  new = ET.XML('<root/>')
  new.extend(t)

  # Outputs <root><item>1</item>...</root>
  print ET.tostring(new)
  ```

- New `Element` method: `iter()` yields the children of the element as a generator. It’s also possible to write `for child in elem:` to loop over an element’s children. The existing method **`getiterator()`** is now deprecated, as is **`getchildren()`** which constructs and returns a list of children.

- New `Element` method: `itertext()` yields all chunks of text that are descendants of the element. For example:

  ```
  t = ET.XML("""<list>
    <item>1</item> <item>2</item>  <item>3</item>
  </list>""")

  # Outputs ['\n  ', '1', ' ', '2', '  ', '3', '\n']
  print list(t.itertext())
  ```

- Deprecated: using an element as a Boolean (i.e., `if elem:`) would return true if the element had any children, or false if there were no children. This behaviour is confusing – `None` is false, but so is a childless element? – so it will now trigger a `FutureWarning`. In your code, you should be explicit: write `len(elem) != 0` if you’re interested in the number of children, or `elem is not None`.

Fredrik Lundh develops ElementTree and produced the 1.3 version; you can read his article describing 1.3 at https://web.archive.org/web/20200703234532/http://effbot.org/zone/elementtree-13-intro.htm. Florent Xicluna updated the version included with Python, after discussions on python-dev and in bpo-6472 [https://bugs.python.org/issue?@action=redirect&bpo=6472].)

Build and C API Changes

Changes to Python’s build process and to the C API include:

- The latest release of the GNU Debugger, GDB 7, can be scripted using Python [https://web.archive.org/web/20110715084810/http://sourceware.org/gdb/current/onlinedocs/gdb/Python.html]. When you begin debugging an executable program P, GDB will look for a file named `P-gdb.py` and automatically read it. Dave Malcolm contributed a `python-gdb.py` that adds a number of commands useful when debugging Python itself. For example, `py-up` and `py-down` go up or down one Python stack frame, which usually corresponds to several C stack frames. `py-print` prints the value of a Python variable, and `py-bt` prints the Python stack trace. (Added as a result of bpo-8032 [https://bugs.python.org/issue?@action=redirect&bpo=8032].)

- If you use the `.gdbinit` file provided with Python, the “pyo” macro in the 2.7 version now works correctly when the thread being debugged doesn’t hold the GIL; the macro now acquires it before printing. (Contributed by Victor Stinner; bpo-3632 [https://bugs.python.org/issue?@action=redirect&bpo=3632].)

- **`Py_AddPendingCall()`** is now thread-safe, letting any worker thread submit notifications to the main Python thread. This is particularly useful for asynchronous IO operations. (Contributed by Kristján Valur Jónsson; bpo-4293 [https://bugs.python.org/issue?@action=redirect&bpo=4293].)

- New function: **`PyCode_NewEmpty()`** creates an empty code object; only the filename, function name, and first line number are required. This is useful for extension modules that are attempting to construct a more useful traceback stack. Previously such extensions needed to call **`PyCode_New()`**, which had many more arguments. (Added by Jeffrey Yasskin.)

- New function: **`PyErr_NewExceptionWithDoc()`** creates a new exception class, just as the existing **`PyErr_NewException()`** does, but takes an extra `char *` argument containing the docstring for the new exception class. (Added by 'lekma' on the Python bug tracker; bpo-7033 [https://bugs.python.org/issue?@action=redirect&bpo=7033].)

- New function: **`PyFrame_GetLineNumber()`** takes a frame object and returns the line number that the frame is currently executing. Previously code would need to get the index of the bytecode instruction currently executing, and then look up the line number corresponding to that address. (Added by Jeffrey Yasskin.)

- New functions: **`PyLong_AsLongAndOverflow()`** and **`PyLong_AsLongLongAndOverflow()`** approximates a Python long integer as a C long or long long. If the number is too large to fit into the output type, an *overflow* flag is set and returned to the caller. (Contributed by Case Van Horsen; bpo-7528 [https://bugs.python.org/issue?

@action=redirect&bpo=7528] and bpo-7767 [https://bugs.python.org/issue?@action=redirect&bpo=7767].)

- New function: stemming from the rewrite of string-to-float conversion, a new **`PyOS_string_to_double()`** function was added. The old **`PyOS_ascii_strtod()`** and **`PyOS_ascii_atof()`** functions are now deprecated.

- New function: **`PySys_SetArgvEx()`** sets the value of `sys.argv` and can optionally update `sys.path` to include the directory containing the script named by `sys.argv[0]` depending on the value of an *updatepath* parameter.

  This function was added to close a security hole for applications that embed Python. The old function, **`PySys_SetArgv()`**, would always update `sys.path`, and sometimes it would add the current directory. This meant that, if you ran an application embedding Python in a directory controlled by someone else, attackers could put a Trojan-horse module in the directory (say, a file named `os.py`) that your application would then import and run.

  If you maintain a C/C++ application that embeds Python, check whether you’re calling **`PySys_SetArgv()`** and carefully consider whether the application should be using **`PySys_SetArgvEx()`** with *updatepath* set to false.

  Security issue reported as **CVE 2008-5983** [https://www.cve.org/CVERecord?id=CVE-2008-5983]; discussed in bpo-5753 [https://bugs.python.org/issue?@action=redirect&bpo=5753], and fixed by Antoine Pitrou.

- New macros: the Python header files now define the following macros: **`Py_ISALNUM`**, **`Py_ISALPHA`**, **`Py_ISDIGIT`**, **`Py_ISLOWER`**, **`Py_ISSPACE`**, **`Py_ISUPPER`**, **`Py_ISXDIGIT`**, **`Py_TOLOWER`**, and **`Py_TOUPPER`**. All of these functions are analogous to the C standard macros for classifying characters, but ignore the current locale setting, because in several places Python needs to analyze characters in a locale-independent way. (Added by Eric Smith; bpo-5793 [https://bugs.python.org/issue?@action=redirect&bpo=5793].)

- Removed function: **`PyEval_CallObject()`** is now only available as a macro. A function version was being kept around to preserve ABI linking compatibility, but that was in 1997; it can certainly be deleted by now. (Removed by Antoine Pitrou; bpo-8276 [https://bugs.python.org/issue?@action=redirect&bpo=8276].)

- New format codes: the **`PyString_FromFormat()`**, **`PyString_FromFormatV()`**, and **`PyErr_Format()`** functions now accept `%lld` and `%llu` format codes for displaying C's long long types. (Contributed by Mark Dickinson; bpo-7228 [https://bugs.python.org/issue?@action=redirect&bpo=7228].)

- The complicated interaction between threads and process forking has been changed. Previously, the child process created by **`os.fork()`** might fail because the child is created with only a single thread running, the thread performing the **`os.fork()`**. If other threads were holding a lock, such as Python's import lock, when the fork was performed, the lock would still be marked as "held" in the new process. But in the child process nothing would ever release the lock, since the other threads weren't replicated, and the child process would no longer be able to perform imports.

  Python 2.7 acquires the import lock before performing an **`os.fork()`**, and will also clean up any locks created using the **`threading`** module. C extension modules that have internal locks, or that call **`fork()`** themselves, will not benefit from this clean-up.

(Fixed by Thomas Wouters; bpo-1590864 [https://bugs.python.org/issue?@action=redirect&bpo=1590864].)

- The **`Py_Finalize()`** function now calls the internal **`threading._shutdown()`** function; this prevents some exceptions from being raised when an interpreter shuts down. (Patch by Adam Olsen; bpo-1722344 [https://bugs.python.org/issue?@action=redirect&bpo=1722344].)
- When using the **`PyMemberDef`** structure to define attributes of a type, Python will no longer let you try to delete or set a **`T_STRING_INPLACE`** attribute.
- Global symbols defined by the **`ctypes`** module are now prefixed with `Py`, or with `_ctypes`. (Implemented by Thomas Heller; bpo-3102 [https://bugs.python.org/issue?@action=redirect&bpo=3102].)
- New configure option: the **`--with-system-expat`** switch allows building the **`pyexpat`** module to use the system Expat library. (Contributed by Arfrever Frehtes Taifersar Arahesis; bpo-7609 [https://bugs.python.org/issue?@action=redirect&bpo=7609].)
- New configure option: the **`--with-valgrind`** option will now disable the pymalloc allocator, which is difficult for the Valgrind memory-error detector to analyze correctly. Valgrind will therefore be better at detecting memory leaks and overruns. (Contributed by James Henstridge; bpo-2422 [https://bugs.python.org/issue?@action=redirect&bpo=2422].)
- New configure option: you can now supply an empty string to **`--with-dbmliborder=`** in order to disable all of the various DBM modules. (Added by Arfrever Frehtes Taifersar Arahesis; bpo-6491 [https://bugs.python.org/issue?@action=redirect&bpo=6491].)

- The **configure** script now checks for floating-point rounding bugs on certain 32-bit Intel chips and defines a `X87_DOUBLE_ROUNDING` preprocessor definition. No code currently uses this definition, but it's available if anyone wishes to use it. (Added by Mark Dickinson; bpo-2937 [https://bugs.python.org/issue?@action=redirect&bpo=2937].)

  configure also now sets a `LDCXXSHARED` Makefile variable for supporting C++ linking. (Contributed by Arfrever Frehtes Taifersar Arahesis; bpo-1222585 [https://bugs.python.org/issue?@action=redirect&bpo=1222585].)

- The build process now creates the necessary files for pkg-config support. (Contributed by Clinton Roy; bpo-3585 [https://bugs.python.org/issue?@action=redirect&bpo=3585].)

- The build process now supports Subversion 1.7. (Contributed by Arfrever Frehtes Taifersar Arahesis; bpo-6094 [https://bugs.python.org/issue?@action=redirect&bpo=6094].)

Capsules

Python 3.1 adds a new C datatype, `PyCapsule`, for providing a C API to an extension module. A capsule is essentially the holder of a C `void *` pointer, and is made available as a module attribute; for example, the `socket` module's API is exposed as `socket.CAPI`, and `unicodedata` exposes `ucnhash_CAPI`. Other extensions can import the module, access its dictionary to get the capsule object, and then get the `void *` pointer, which will usually point to an array of pointers to the module's various API functions.

There is an existing data type already used for this, `PyCObject`, but it doesn't provide type safety. Evil code written in pure Python could cause a segmentation fault by taking a `PyCObject` from module A and somehow substituting it for the **`PyCObject`** in module B. Capsules know their own name, and getting the pointer requires providing the name:

```
void *vtable;

if (!PyCapsule_IsValid(capsule, "mymodule.CAPI") {
        PyErr_SetString(PyExc_ValueError, "argument type
invalid");
        return NULL;
}

vtable = PyCapsule_GetPointer(capsule, "mymodule.CAPI");
```

You are assured that `vtable` points to whatever you're expecting. If a different capsule was passed in, **`PyCapsule_IsValid()`** would detect the mismatched name and return false. Refer to Providing a C API for an Extension Module for more information on using these objects.

Python 2.7 now uses capsules internally to provide various extension-module APIs, but the **`PyCObject_AsVoidPtr()`** was modified to handle capsules, preserving compile-time compatibility with the **`PyCObject`** interface. Use of **`PyCObject_AsVoidPtr()`** will signal a **`PendingDeprecationWarning`**, which is silent by default.

Implemented in Python 3.1 and backported to 2.7 by Larry Hastings; discussed in bpo-5630 [https://bugs.python.org/issue?@action=redirect&bpo=5630].

Port-Specific Changes: Windows

- The **`msvcrt`** module now contains some constants from the `crtassem.h` header file: **`CRT_ASSEMBLY_VERSION`**, **`VC_ASSEMBLY_PUBLICKEYTOKEN`**, and **`LIBRARIES_ASSEMBLY_NAME_PREFIX`**. (Contributed by David Cournapeau; bpo-4365 [https://bugs.python.org/issue?@action=redirect&bpo=4365].)

- The `_winreg` module for accessing the registry now implements the `CreateKeyEx()` and `DeleteKeyEx()` functions, extended versions of previously supported functions that take several extra arguments. The `DisableReflectionKey()`, `EnableReflectionKey()`, and `QueryReflectionKey()` were also tested and documented. (Implemented by Brian Curtin: bpo-7347 [https://bugs.python.org/issue?@action=redirect&bpo=7347].)
- The new `_beginthreadex()` API is used to start threads, and the native thread-local storage functions are now used. (Contributed by Kristján Valur Jónsson; bpo-3582 [https://bugs.python.org/issue?@action=redirect&bpo=3582].)
- The `os.kill()` function now works on Windows. The signal value can be the constants `CTRL_C_EVENT`, `CTRL_BREAK_EVENT`, or any integer. The first two constants will send `Control-C` and `Control-Break` keystroke events to subprocesses; any other value will use the `TerminateProcess()` API. (Contributed by Miki Tebeka; bpo-1220212 [https://bugs.python.org/issue?@action=redirect&bpo=1220212].)
- The `os.listdir()` function now correctly fails for an empty path. (Fixed by Hirokazu Yamamoto; bpo-5913 [https://bugs.python.org/issue?@action=redirect&bpo=5913].)
- The `mimetypes` module will now read the MIME database from the Windows registry when initializing. (Patch by Gabriel Genellina; bpo-4969 [https://bugs.python.org/issue?@action=redirect&bpo=4969].)

Port-Specific Changes: Mac OS X

- The path `/Library/Python/2.7/site-packages` is now appended to `sys.path`, in order to share added packages between the system installation and a user-installed copy of the same version. (Changed by Ronald Oussoren; bpo-4865 [https://bugs.python.org/issue?@action=redirect&bpo=4865].)

Changed in version 2.7.13: As of 2.7.13, this change was removed. `/Library/Python/2.7/site-packages`, the site-packages directory used by the Apple-supplied system Python 2.7 is no longer appended to `sys.path` for user-installed Pythons such as from the python.org installers. As of macOS 10.12, Apple changed how the system site-packages directory is configured, which could cause installation of pip components, like setuptools, to fail. Packages installed for the system Python will no longer be shared with user-installed Pythons. (bpo-28440 [https://bugs.python.org/issue?@action=redirect&bpo=28440])

Port-Specific Changes: FreeBSD

- FreeBSD 7.1's **`SO_SETFIB`** constant, used with the **`socket()`** methods **`getsockopt()`**/**`setsockopt()`** to select an alternate routing table, is now available in the **`socket`** module. (Added by Kyle VanderBeek; bpo-8235 [https://bugs.python.org/issue?@action=redirect&bpo=8235].)

Other Changes and Fixes

- Two benchmark scripts, `iobench` and `ccbench`, were added to the `Tools` directory. `iobench` measures the speed of the built-in file I/O objects returned by **`open()`** while performing various operations, and `ccbench` is a concurrency benchmark that tries to measure computing throughput, thread switching latency, and IO processing bandwidth when performing several tasks using a varying number of threads.
- The `Tools/i18n/msgfmt.py` script now understands plural forms in `.po` files. (Fixed by Martin von Löwis; bpo-5464 [https://bugs.python.org/issue?@action=redirect&bpo=5464].)
- When importing a module from a `.pyc` or `.pyo` file with an existing `.py` counterpart, the **`co_filename`** attributes of the

resulting code objects are overwritten when the original filename is obsolete. This can happen if the file has been renamed, moved, or is accessed through different paths. (Patch by Ziga Seilnacht and Jean-Paul Calderone; bpo-1180193 [https://bugs.python.org/issue?@action=redirect&bpo=1180193].)

- The `regrtest.py` script now takes a **`--randseed=`** switch that takes an integer that will be used as the random seed for the **`-r`** option that executes tests in random order. The **`-r`** option also reports the seed that was used (Added by Collin Winter.)
- Another `regrtest.py` switch is **`-j`**, which takes an integer specifying how many tests run in parallel. This allows reducing the total runtime on multi-core machines. This option is compatible with several other options, including the **`-R`** switch which is known to produce long runtimes. (Added by Antoine Pitrou, bpo-6152 [https://bugs.python.org/issue?@action=redirect&bpo=6152].) This can also be used with a new **`-F`** switch that runs selected tests in a loop until they fail. (Added by Antoine Pitrou; bpo-7312 [https://bugs.python.org/issue?@action=redirect&bpo=7312].)
- When executed as a script, the `py_compile.py` module now accepts `'-'` as an argument, which will read standard input for the list of filenames to be compiled. (Contributed by Piotr Ożarowski; bpo-8233 [https://bugs.python.org/issue?@action=redirect&bpo=8233].)

Porting to Python 2.7

This section lists previously described changes and other bugfixes that may require changes to your code:

- The **`range()`** function processes its arguments more consistently; it will now call **`__int__()`** on non-float, non-integer arguments that are supplied to it. (Fixed by Alexander Belopolsky; bpo-1533 [https://bugs.python.org/issue?@action=redirect&bpo=1533].)

- The string **`format()`** method changed the default precision used for floating-point and complex numbers from 6 decimal places to 12, which matches the precision used by **`str()`**. (Changed by Eric Smith; bpo-5920 [https://bugs.python.org/issue?@action=redirect&bpo=5920].)
- Because of an optimization for the **`with`** statement, the special methods **`__enter__()`** and **`__exit__()`** must belong to the object's type, and cannot be directly attached to the object's instance. This affects new-style classes (derived from **`object`**) and C extension types. (bpo-6101 [https://bugs.python.org/issue?@action=redirect&bpo=6101].)
- Due to a bug in Python 2.6, the *exc_value* parameter to **`__exit__()`** methods was often the string representation of the exception, not an instance. This was fixed in 2.7, so *exc_value* will be an instance as expected. (Fixed by Florent Xicluna; bpo-7853 [https://bugs.python.org/issue?@action=redirect&bpo=7853].)
- When a restricted set of attributes were set using `__slots__`, deleting an unset attribute would not raise **`AttributeError`** as you would expect. Fixed by Benjamin Peterson; bpo-7604 [https://bugs.python.org/issue?@action=redirect&bpo=7604].)

In the standard library:

- Operations with **`datetime`** instances that resulted in a year falling outside the supported range didn't always raise **`OverflowError`**. Such errors are now checked more carefully and will now raise the exception. (Reported by Mark Leander, patch by Anand B. Pillai and Alexander Belopolsky; bpo-7150 [https://bugs.python.org/issue?@action=redirect&bpo=7150].)

- When using **`Decimal`** instances with a string's **`format()`** method, the default alignment was previously left-alignment. This has been changed to right-alignment, which might change the output of your programs. (Changed by Mark Dickinson; bpo-6857 [https://bugs.python.org/issue?@action=redirect&bpo=6857].)

  Comparisons involving a signaling NaN value (or `sNAN`) now signal **`InvalidOperation`** instead of silently returning a true or false value depending on the comparison operator. Quiet NaN values (or `NaN`) are now hashable. (Fixed by Mark Dickinson; bpo-7279 [https://bugs.python.org/issue?@action=redirect&bpo=7279].)

- The **`xml.etree.ElementTree`** library no longer escapes ampersands and angle brackets when outputting an XML processing instruction (which looks like `<?xml-stylesheet href="#style1"?>`) or comment (which looks like `<!-- comment -->`). (Patch by Neil Muller; bpo-2746 [https://bugs.python.org/issue?@action=redirect&bpo=2746].)

- The **`readline()`** method of **`StringIO`** objects now does nothing when a negative length is requested, as other file-like objects do. (bpo-7348 [https://bugs.python.org/issue?@action=redirect&bpo=7348]).

- The **`syslog`** module will now use the value of `sys.argv[0]` as the identifier instead of the previous default value of `'python'`. (Changed by Sean Reifschneider; bpo-8451 [https://bugs.python.org/issue?@action=redirect&bpo=8451].)

- The **`tarfile`** module's default error handling has changed, to no longer suppress fatal errors. The default error level was previously 0, which meant that errors would only result in a message being written to the debug log, but because the debug log is not activated by default, these errors go unnoticed. The default error level is now 1, which raises an exception if there's an error. (Changed by Lars Gustäbel; bpo-7357 [https://bugs.python.org/issue?@action=redirect&bpo=7357].)

- The `urlparse` module's `urlsplit()` now handles unknown URL schemes in a fashion compliant with **RFC 3986** [https://datatracker.ietf.org/doc/html/rfc3986.html]: if the URL is of the form `"<something>://..."`, the text before the `://` is treated as the scheme, even if it's a made-up scheme that the module doesn't know about. This change may break code that worked around the old behaviour. For example, Python 2.6.4 or 2.5 will return the following:

  ```
  >>> import urlparse
  >>> urlparse.urlsplit('invented://host/filename?query')
  ('invented', '', '//host/filename?query', '', '')
  ```

  Python 2.7 (and Python 2.6.5) will return:

  ```
  >>> import urlparse
  >>> urlparse.urlsplit('invented://host/filename?query')
  ('invented', 'host', '/filename?query', '', '')
  ```

  (Python 2.7 actually produces slightly different output, since it returns a named tuple instead of a standard tuple.)

For C extensions:

- C extensions that use integer format codes with the `PyArg_Parse*` family of functions will now raise a `TypeError` exception instead of triggering a `DeprecationWarning` (bpo-5080 [https://bugs.python.org/issue?@action=redirect&bpo=5080]).
- Use the new `PyOS_string_to_double()` function instead of the old **`PyOS_ascii_strtod()`** and **`PyOS_ascii_atof()`** functions, which are now deprecated.

For applications that embed Python:

- The **`PySys_SetArgvEx()`** function was added, letting applications close a security hole when the existing **`PySys_SetArgv()`** function was used. Check whether you're calling **`PySys_SetArgv()`** and carefully consider whether the application should be using **`PySys_SetArgvEx()`** with *updatepath* set to false.

New Features Added to Python 2.7 Maintenance Releases

New features may be added to Python 2.7 maintenance releases when the situation genuinely calls for it. Any such additions must go through the Python Enhancement Proposal process, and make a compelling case for why they can't be adequately addressed by either adding the new feature solely to Python 3, or else by publishing it on the Python Package Index.

In addition to the specific proposals listed below, there is a general exemption allowing new `-3` warnings to be added in any Python 2.7 maintenance release.

Two new environment variables for debug mode

In debug mode, the `[xxx refs]` statistic is not written by default, the **`PYTHONSHOWREFCOUNT`** environment variable now must also be set. (Contributed by Victor Stinner; bpo-31733 [https://bugs.python.org/issue?@action=redirect&bpo=31733].)

When Python is compiled with `COUNT_ALLOC` defined, allocation counts are no longer dumped by default anymore: the **`PYTHONSHOWALLOCCOUNT`** environment variable must now also be set. Moreover, allocation counts are now dumped into stderr, rather than stdout. (Contributed by Victor Stinner; bpo-31692 [https://bugs.python.org/issue?@action=redirect&bpo=31692].)

Added in version 2.7.15.

PEP 434: IDLE Enhancement Exception for All Branches

PEP 434 [https://peps.python.org/pep-0434/] describes a general exemption for changes made to the IDLE development environment shipped along with Python. This exemption makes it possible for the IDLE developers to provide a more consistent user experience across all supported versions of Python 2 and 3.

For details of any IDLE changes, refer to the NEWS file for the specific release.

PEP 466: Network Security Enhancements for Python 2.7

PEP 466 [https://peps.python.org/pep-0466/] describes a number of network security enhancement proposals that have been approved for inclusion in Python 2.7 maintenance releases, with the first of those changes appearing in the Python 2.7.7 release.

PEP 466 [https://peps.python.org/pep-0466/] related features added in Python 2.7.7:

- `hmac.compare_digest()` was backported from Python 3 to make a timing attack resistant comparison operation available to Python 2 applications. (Contributed by Alex Gaynor; bpo-21306 [https://bugs.python.org/issue?@action=redirect&bpo=21306].)

- OpenSSL 1.0.1g was upgraded in the official Windows installers published on python.org. (Contributed by Zachary Ware; bpo-21462 [https://bugs.python.org/issue?@action=redirect&bpo=21462].)

PEP 466 [https://peps.python.org/pep-0466/] related features added in Python 2.7.8:

- `hashlib.pbkdf2_hmac()` was backported from Python 3 to make a hashing algorithm suitable for secure password storage broadly available to Python 2 applications. (Contributed by Alex Gaynor; bpo-21304 [https://bugs.python.org/issue?@action=redirect&bpo=21304].)
- OpenSSL 1.0.1h was upgraded for the official Windows installers published on python.org. (Contributed by Zachary Ware in bpo-21671 [https://bugs.python.org/issue?@action=redirect&bpo=21671] for **CVE 2014-0224** [https://www.cve.org/CVERecord?id=CVE-2014-0224].)

PEP 466 [https://peps.python.org/pep-0466/] related features added in Python 2.7.9:

- Most of Python 3.4's `ssl` module was backported. This means `ssl` now supports Server Name Indication, TLS1.x settings, access to the platform certificate store, the `SSLContext` class, and other features. (Contributed by Alex Gaynor and David Reid; bpo-21308 [https://bugs.python.org/issue?@action=redirect&bpo=21308].)

  Refer to the “Version added: 2.7.9” notes in the module documentation for specific details.

- `os.urandom()` was changed to cache a file descriptor to `/dev/urandom` instead of reopening `/dev/urandom` on every call. (Contributed by Alex Gaynor; bpo-21305 [https://bugs.python.org/issue?@action=redirect&bpo=21305].)

- **`hashlib.algorithms_guaranteed`** and **`hashlib.algorithms_available`** were backported from Python 3 to make it easier for Python 2 applications to select the strongest available hash algorithm. (Contributed by Alex Gaynor in bpo-21307 [https://bugs.python.org/issue?@action=redirect&bpo=21307])

PEP 477: Backport ensurepip (PEP 453) to Python 2.7

PEP 477 [https://peps.python.org/pep-0477/] approves the inclusion of the **PEP 453** [https://peps.python.org/pep-0453/] ensurepip module and the improved documentation that was enabled by it in the Python 2.7 maintenance releases, appearing first in the Python 2.7.9 release.

Bootstrapping pip By Default

The new **`ensurepip`** module (defined in **PEP 453** [https://peps.python.org/pep-0453/]) provides a standard cross-platform mechanism to bootstrap the pip installer into Python installations. The version of `pip` included with Python 2.7.9 is `pip` 1.5.6, and future 2.7.x maintenance releases will update the bundled version to the latest version of `pip` that is available at the time of creating the release candidate.

By default, the commands `pip`, `pipX` and `pipX.Y` will be installed on all platforms (where X.Y stands for the version of the Python installation), along with the `pip` Python package and its dependencies.

For CPython source builds on POSIX systems, the `make install` and `make altinstall` commands do not bootstrap `pip` by default. This behaviour can be controlled through configure options, and overridden through Makefile options.

On Windows and Mac OS X, the CPython installers now default to installing `pip` along with CPython itself (users may opt out of installing it during the installation process). Window users will need to opt in to the automatic `PATH` modifications to have `pip` available from the command line by default, otherwise it can still be accessed through the Python launcher for Windows as `py -m pip`.

As **discussed in the PEP** [https://peps.python.org/pep-0477/#disabling-ensurepip-by-downstream-distributors], platform packagers may choose not to install these commands by default, as long as, when invoked, they provide clear and simple directions on how to install them on that platform (usually using the system package manager).

Documentation Changes

As part of this change, the Installing Python Modules and Distributing Python Modules sections of the documentation have been completely redesigned as short getting started and FAQ documents. Most packaging documentation has now been moved out to the Python Packaging Authority maintained Python Packaging User Guide [https://packaging.python.org] and the documentation of the individual projects.

However, as this migration is currently still incomplete, the legacy versions of those guides remaining available as Building C and C++ Extensions with setuptools and Building C and C++ Extensions with setuptools.

See also

PEP 453 [https://peps.python.org/pep-0453/] – Explicit bootstrapping of pip in Python installations

> PEP written by Donald Stufft and Nick Coghlan, implemented by Donald Stufft, Nick Coghlan, Martin von Löwis and Ned Deily.

PEP 476: Enabling certificate verification by default for stdlib http clients

PEP 476 [https://peps.python.org/pep-0476/] updated **`httplib`** and modules which use it, such as **`urllib2`** and **`xmlrpclib`**, to now verify that the server presents a certificate which is signed by a Certificate Authority in the platform trust store and whose hostname matches the hostname being requested by default, significantly improving security for many applications. This change was made in the Python 2.7.9 release.

For applications which require the old previous behavior, they can pass an alternate context:

```
import urllib2
import ssl

# This disables all verification
context = ssl._create_unverified_context()

# This allows using a specific certificate for the host, which
doesn't need
# to be in the trust store
context =
ssl.create_default_context(cafile="/path/to/file.crt")

urllib2.urlopen("https://invalid-cert", context=context)
```

PEP 493: HTTPS verification migration tools for Python 2.7

PEP 493 [https://peps.python.org/pep-0493/] provides additional migration tools to support a more incremental infrastructure upgrade process for environments containing applications and services relying on the historically permissive processing of server certificates when establishing client HTTPS connections. These additions were made in the Python 2.7.12 release.

These tools are intended for use in cases where affected applications and services can't be modified to explicitly pass a more permissive SSL context when establishing the connection.

For applications and services which can't be modified at all, the new `PYTHONHTTPSVERIFY` environment variable may be set to `0` to revert an entire Python process back to the default permissive behaviour of Python 2.7.8 and earlier.

For cases where the connection establishment code can't be modified, but the overall application can be, the new **`ssl._https_verify_certificates()`** function can be used to adjust the default behaviour at runtime.

New `make regen-all` build target

To simplify cross-compilation, and to ensure that CPython can reliably be compiled without requiring an existing version of Python to already be available, the autotools-based build system no longer attempts to implicitly recompile generated files based on file modification times.

Instead, a new `make regen-all` command has been added to force regeneration of these files when desired (e.g. after an initial version of Python has already been built based on the pregenerated versions).

More selective regeneration targets are also defined - see Makefile.pre.in [https://github.com/python/cpython/tree/3.14/Makefile.pre.in] for details.

(Contributed by Victor Stinner in bpo-23404 [https://bugs.python.org/issue?@action=redirect&bpo=23404].)

Added in version 2.7.14.

Removal of `make touch` build target

The `make touch` build target previously used to request implicit regeneration of generated files by updating their modification times has been removed.

It has been replaced by the new `make regen-all` target.

(Contributed by Victor Stinner in bpo-23404 [https://bugs.python.org/issue?@action=redirect&bpo=23404].)

Changed in version 2.7.14.

Acknowledgements

The author would like to thank the following people for offering suggestions, corrections and assistance with various drafts of this article: Nick Coghlan, Philip Jenvey, Ryan Lovett, R. David Murray, Hugh Secker-Walker.

What’s New in Python 2.6

Author:: A.M. Kuchling (amk at amk.ca)

This article explains the new features in Python 2.6, released on October 1, 2008. The release schedule is described in **PEP 361** [https://peps.python.org/pep-0361/].

The major theme of Python 2.6 is preparing the migration path to Python 3.0, a major redesign of the language. Whenever possible, Python 2.6 incorporates new features and syntax from 3.0 while remaining compatible with existing code by not removing older features or syntax. When it’s not possible to do that, Python 2.6 tries to do what it can, adding compatibility functions in a `future_builtins` module and a `-3` switch to warn about usages that will become unsupported in 3.0.

Some significant new packages have been added to the standard library, such as the `multiprocessing` and `json` modules, but there aren’t many new features that aren’t related to Python 3.0 in some way.

Python 2.6 also sees a number of improvements and bugfixes throughout the source. A search through the change logs finds there were 259 patches applied and 612 bugs fixed between Python 2.5 and 2.6. Both figures are likely to be underestimates.

This article doesn’t attempt to provide a complete specification of the new features, but instead provides a convenient overview. For full details, you should refer to the documentation for Python 2.6. If you want to understand the rationale for the design and implementation, refer to the

PEP for a particular new feature. Whenever possible, “What’s New in Python” links to the bug/patch item for each change.

Python 3.0

The development cycle for Python versions 2.6 and 3.0 was synchronized, with the alpha and beta releases for both versions being made on the same days. The development of 3.0 has influenced many features in 2.6.

Python 3.0 is a far-ranging redesign of Python that breaks compatibility with the 2.x series. This means that existing Python code will need some conversion in order to run on Python 3.0. However, not all the changes in 3.0 necessarily break compatibility. In cases where new features won’t cause existing code to break, they’ve been backported to 2.6 and are described in this document in the appropriate place. Some of the 3.0-derived features are:

- A **`__complex__()`** method for converting objects to a complex number.
- Alternate syntax for catching exceptions: `except TypeError as exc`.
- The addition of **`functools.reduce()`** as a synonym for the built-in **`reduce()`** function.

Python 3.0 adds several new built-in functions and changes the semantics of some existing builtins. Functions that are new in 3.0 such as **`bin()`** have simply been added to Python 2.6, but existing builtins haven’t been changed; instead, the **`future_builtins`** module has versions with the new 3.0 semantics. Code written to be compatible with 3.0 can do `from future_builtins import hex, map` as necessary.

A new command-line switch, **`-3`**, enables warnings about features that will be removed in Python 3.0. You can run code with this switch to see how much work will be necessary to port code to 3.0. The value of this switch is available to Python code as the boolean variable **`sys.py3kwarning`**, and to C extension code as **`Py_Py3kWarningFlag`**.

See also

The 3*xxx* series of PEPs, which contains proposals for Python 3.0. **PEP 3000** [https://peps.python.org/pep-3000/] describes the development process for Python 3.0. Start with **PEP 3100** [https://peps.python.org/pep-3100/] that describes the general goals for Python 3.0, and then explore the higher-numbered PEPs that propose specific features.

Changes to the Development Process

While 2.6 was being developed, the Python development process underwent two significant changes: we switched from SourceForge’s issue tracker to a customized Roundup installation, and the documentation was converted from LaTeX to reStructuredText.

New Issue Tracker: Roundup

For a long time, the Python developers had been growing increasingly annoyed by SourceForge’s bug tracker. SourceForge’s hosted solution doesn’t permit much customization; for example, it wasn’t possible to customize the life cycle of issues.

The infrastructure committee of the Python Software Foundation therefore posted a call for issue trackers, asking volunteers to set up different products and import some of the bugs and patches from SourceForge. Four different trackers were examined: Jira [https://www.atlassian.com/software/jira/], Launchpad [https://launchpad.net/], Roundup [https://roundup.sourceforge.io/], and Trac [https://trac.edgewall.org/]. The committee eventually settled on Jira and Roundup as the two candidates. Jira is a commercial product that offers no-cost hosted instances to free-software projects; Roundup is an open-source project that requires volunteers to administer it and a server to host it.

After posting a call for volunteers, a new Roundup installation was set up at https://bugs.python.org. One installation of Roundup can host multiple trackers, and this server now also hosts issue trackers for Jython and for the Python web site. It will surely find other uses in the future. Where possible, this edition of “What’s New in Python” links to the bug/patch item for each change.

Hosting of the Python bug tracker is kindly provided by Upfront Systems [https://upfrontsoftware.co.za] of Stellenbosch, South Africa. Martin von Löwis put a lot of effort into importing existing bugs and patches from SourceForge; his scripts for this import operation are at `https://svn.python.org/view/tracker/importer/` and may be useful to other projects wishing to move from SourceForge to Roundup.

See also

https://bugs.python.org
The Python bug tracker.

https://bugs.jython.org:

The Jython bug tracker.

https://roundup.sourceforge.io/
Roundup downloads and documentation.

https://svn.python.org/view/tracker/importer/
Martin von Löwis’s conversion scripts.

New Documentation Format: reStructuredText Using Sphinx

The Python documentation was written using LaTeX since the project started around 1989. In the 1980s and early 1990s, most documentation was printed out for later study, not viewed online. LaTeX was widely used because it provided attractive printed output while remaining straightforward to write once the basic rules of the markup were learned.

Today LaTeX is still used for writing publications destined for printing, but the landscape for programming tools has shifted. We no longer print out reams of documentation; instead, we browse through it online and HTML has become the most important format to support. Unfortunately, converting LaTeX to HTML is fairly complicated and Fred L. Drake Jr., the long-time Python documentation editor, spent a lot of time maintaining the conversion process. Occasionally people would suggest converting the documentation into SGML and later XML, but performing a good conversion is a major task and no one ever committed the time required to finish the job.

During the 2.6 development cycle, Georg Brandl put a lot of effort into building a new toolchain for processing the documentation. The resulting package is called Sphinx, and is available from https://www.sphinx-doc.org/.

Sphinx concentrates on HTML output, producing attractively styled and modern HTML; printed output is still supported through conversion to LaTeX. The input format is reStructuredText, a markup syntax supporting custom extensions and directives that is commonly used in the Python community.

Sphinx is a standalone package that can be used for writing, and almost two dozen other projects (listed on the Sphinx web site [https://www.sphinx-doc.org/en/master/examples.html]) have adopted Sphinx as their documentation tool.

See also

Documenting Python [https://devguide.python.org/documenting/]
Describes how to write for Python’s documentation.

Sphinx [https://www.sphinx-doc.org/]
Documentation and code for the Sphinx toolchain.

Docutils [https://docutils.sourceforge.io]
The underlying reStructuredText parser and toolset.

PEP 343: The ‘with’ statement

The previous version, Python 2.5, added the ‘`with`’ statement as an optional feature, to be enabled by a `from __future__ import with_statement` directive. In 2.6 the statement no longer needs to be specially enabled; this means that **`with`** is now always a keyword. The rest of this section is a copy of the corresponding section from the “What’s New in Python 2.5” document; if you’re familiar with the ‘**`with`**’ statement from Python 2.5, you can skip this section.

The ‘`with`’ statement clarifies code that previously would use `try...finally` blocks to ensure that clean-up code is executed. In this section, I’ll discuss the statement as it will commonly be used. In the next section, I’ll examine the implementation details and show how to write objects for use with this statement.

The ‘`with`’ statement is a control-flow structure whose basic structure is:

```
with expression [as variable]:
    with-block
```

The expression is evaluated, and it should result in an object that supports the context management protocol (that is, has `__enter__()` and `__exit__()` methods).

The object’s `__enter__()` is called before *with-block* is executed and therefore can run set-up code. It also may return a value that is bound to the name *variable*, if given. (Note carefully that *variable* is *not* assigned the result of *expression*.)

After execution of the *with-block* is finished, the object’s `__exit__()` method is called, even if the block raised an exception, and can therefore run clean-up code.

Some standard Python objects now support the context management protocol and can be used with the ‘`with`’ statement. File objects are one example:

```
with open('/etc/passwd', 'r') as f:
    for line in f:
        print line
        ... more processing code ...
```

After this statement has executed, the file object in *f* will have been automatically closed, even if the `for` loop raised an exception part-way through the block.

Note

In this case, *f* is the same object created by `open()`, because `__enter__()` returns *self*.

The `threading` module’s locks and condition variables also support the ‘`with`’ statement:

```
lock = threading.Lock()
with lock:
    # Critical section of code
    ...
```

The lock is acquired before the block is executed and always released once the block is complete.

The `localcontext()` function in the `decimal` module makes it easy to save and restore the current decimal context, which encapsulates the desired precision and rounding characteristics for computations:

```
from decimal import Decimal, Context, localcontext

# Displays with default precision of 28 digits
v = Decimal('578')
print v.sqrt()

with localcontext(Context(prec=16)):
    # All code in this block uses a precision of 16 digits.
    # The original context is restored on exiting the block.
    print v.sqrt()
```

Writing Context Managers

Under the hood, the ‘`with`’ statement is fairly complicated. Most people will only use ‘**`with`**’ in company with existing objects and don’t need to know these details, so you can skip the rest of this section if you like. Authors of new objects will need to understand the details of the underlying implementation and should keep reading.

A high-level explanation of the context management protocol is:

- The expression is evaluated and should result in an object called a “context manager”. The context manager must have `__enter__()` and `__exit__()` methods.
- The context manager’s `__enter__()` method is called. The value returned is assigned to *VAR*. If no `as VAR` clause is present, the value is simply discarded.
- The code in *BLOCK* is executed.
- If *BLOCK* raises an exception, the context manager’s `__exit__()` method is called with three arguments, the exception details (`type, value, traceback`, the same values returned by `sys.exc_info()`, which can also be `None` if no exception occurred). The method’s return value controls whether an exception is re-raised: any false value re-raises the exception, and `True` will result in suppressing it. You’ll only rarely want to suppress the exception, because if you do the author of the code containing the ‘`with`’ statement will never realize anything went wrong.
- If *BLOCK* didn’t raise an exception, the `__exit__()` method is still called, but *type*, *value*, and *traceback* are all `None`.

Let’s think through an example. I won’t present detailed code but will only sketch the methods necessary for a database that supports transactions.

(For people unfamiliar with database terminology: a set of changes to the database are grouped into a transaction.

Transactions can be either committed, meaning that all the changes are written into the database, or rolled back, meaning that the changes are all discarded and the database is unchanged. See any database textbook for more information.)

Let’s assume there’s an object representing a database connection. Our goal will be to let the user write code like this:

```
db_connection = DatabaseConnection()
with db_connection as cursor:
    cursor.execute('insert into ...')
    cursor.execute('delete from ...')
    # ... more operations ...
```

The transaction should be committed if the code in the block runs flawlessly or rolled back if there’s an exception. Here’s the basic interface for **`DatabaseConnection`** that I’ll assume:

```
class DatabaseConnection:
    # Database interface
    def cursor(self):
        "Returns a cursor object and starts a new transaction"
    def commit(self):
        "Commits current transaction"
    def rollback(self):
        "Rolls back current transaction"
```

The **`__enter__()`** method is pretty easy, having only to start a new transaction. For this application the resulting cursor object would be a useful result, so the method will return it. The user can then add `as cursor` to their ‘**`with`**’ statement to bind the cursor to a variable name.

```
class DatabaseConnection:
    ...
    def __enter__(self):
        # Code to start a new transaction
        cursor = self.cursor()
        return cursor
```

The `__exit__()` method is the most complicated because it's where most of the work has to be done. The method has to check if an exception occurred. If there was no exception, the transaction is committed. The transaction is rolled back if there was an exception.

In the code below, execution will just fall off the end of the function, returning the default value of `None`. `None` is false, so the exception will be re-raised automatically. If you wished, you could be more explicit and add a `return` statement at the marked location.

```
class DatabaseConnection:
    ...
    def __exit__(self, type, value, tb):
        if tb is None:
            # No exception, so commit
            self.commit()
        else:
            # Exception occurred, so rollback.
            self.rollback()
            # return False
```

The contextlib module

The `contextlib` module provides some functions and a decorator that are useful when writing objects for use with the '`with`' statement.

The decorator is called `contextmanager()`, and lets you write a single generator function instead of defining a new class. The generator should yield exactly one value. The code up to the `yield` will be executed as the `__enter__()` method, and the value yielded will be the method's return value that will get bound to the variable in the ‘`with`’ statement’s **`as`** clause, if any. The code after the **`yield`** will be executed in the `__exit__()` method. Any exception raised in the block will be raised by the **`yield`** statement.

Using this decorator, our database example from the previous section could be written as:

```
from contextlib import contextmanager

@contextmanager
def db_transaction(connection):
    cursor = connection.cursor()
    try:
        yield cursor
    except:
        connection.rollback()
        raise
    else:
        connection.commit()

db = DatabaseConnection()
with db_transaction(db) as cursor:
    ...
```

The `contextlib` module also has a `nested(mgr1, mgr2, ...)` function that combines a number of context managers so you don’t need to write nested ‘`with`’ statements. In this example, the single ‘**`with`**’ statement both starts a database transaction and acquires a thread lock:

```
lock = threading.Lock()
with nested (db_transaction(db), lock) as (cursor, locked):
    ...
```

Finally, the `closing()` function returns its argument so that it can be bound to a variable, and calls the argument’s `.close()` method at the end of the block.

```
import urllib, sys
from contextlib import closing

with closing(urllib.urlopen('http://www.yahoo.com')) as f:
    for line in f:
        sys.stdout.write(line)
```

See also

PEP 343 [https://peps.python.org/pep-0343/] - The “with” statement

PEP written by Guido van Rossum and Nick Coghlan; implemented by Mike Bland, Guido van Rossum, and Neal Norwitz. The PEP shows the code generated for a ‘`with`’ statement, which can be helpful in learning how the statement works.

The documentation for the `contextlib` module.

PEP 366: Explicit Relative Imports From a Main Module

Python’s `-m` switch allows running a module as a script. When you ran a module that was located inside a package, relative imports didn’t work correctly.

The fix for Python 2.6 adds a `module.__package__` attribute. When this attribute is present, relative imports will be relative to the value of this attribute instead of the `__name__` attribute.

PEP 302-style importers can then set `__package__` as necessary. The `runpy` module that implements the `-m` switch now does this, so relative imports will now work correctly in scripts running from inside a package.

PEP 370: Per-user `site-packages` Directory

When you run Python, the module search path `sys.path` usually includes a directory whose path ends in `"site-packages"`. This directory is intended to hold locally installed packages available to all users using a machine or a particular site installation.

Python 2.6 introduces a convention for user-specific site directories. The directory varies depending on the platform:

- Unix and Mac OS X: `~/.local/`
- Windows: `%APPDATA%/Python`

Within this directory, there will be version-specific subdirectories, such as `lib/python2.6/site-packages` on Unix/Mac OS and `Python26/site-packages` on Windows.

If you don’t like the default directory, it can be overridden by an environment variable. **`PYTHONUSERBASE`** sets the root directory used for all Python versions supporting this feature. On Windows, the directory for application-specific data can be changed by setting the **`APPDATA`** environment variable. You can also modify the `site.py` file for your Python installation.

The feature can be disabled entirely by running Python with the **`-s`** option or setting the **`PYTHONNOUSERSITE`** environment variable.

See also

PEP 370 [https://peps.python.org/pep-0370/] - Per-user `site-packages` Directory

PEP written and implemented by Christian Heimes.

PEP 371: The `multiprocessing` Package

The new `multiprocessing` package lets Python programs create new processes that will perform a computation and return a result to the parent. The parent and child processes can communicate using queues and pipes, synchronize their operations using locks and semaphores, and can share simple arrays of data.

The `multiprocessing` module started out as an exact emulation of the `threading` module using processes instead of threads. That goal was discarded along the path to Python 2.6, but the general approach of the module is still similar. The fundamental class is the `Process`, which is passed a callable object and a collection of arguments. The `start()` method sets the callable running in a subprocess, after which you can call the `is_alive()` method to check whether the subprocess is still running and the `join()` method to wait for the process to exit.

Here’s a simple example where the subprocess will calculate a factorial. The function doing the calculation is written strangely so that it takes significantly longer when the input argument is a multiple of 4.

```
import time
from multiprocessing import Process, Queue


def factorial(queue, N):
    "Compute a factorial."
    # If N is a multiple of 4, this function will take much
longer.
    if (N % 4) == 0:
        time.sleep(.05 * N/4)
    # Calculate the result
    fact = 1L
    for i in range(1, N+1):
        fact = fact * i

    # Put the result on the queue
    queue.put(fact)

if __name__ == '__main__':
    queue = Queue()

    N = 5

    p = Process(target=factorial, args=(queue, N))
    p.start()
    p.join()

    result = queue.get()
    print 'Factorial', N, '=', result
```

A **`Queue`** is used to communicate the result of the factorial. The **`Queue`** object is stored in a global variable. The child process will use the value of the variable when the child was created; because it's a **`Queue`**, parent and child can use the object to communicate. (If the parent were to change the value of the global variable, the child's value would be unaffected, and vice versa.)

Two other classes, **`Pool`** and **`Manager`**, provide higher-level interfaces. **`Pool`** will create a fixed number of worker processes, and requests can then be distributed to the workers by calling **`apply()`** or **`apply_async()`** to add a single request, and **`map()`** or **`map_async()`** to add a number of requests. The following code uses a **`Pool`** to spread requests across 5 worker processes and retrieve a list of results:

```
from multiprocessing import Pool

def factorial(N, dictionary):
    "Compute a factorial."
    ...
p = Pool(5)
result = p.map(factorial, range(1, 1000, 10))
for v in result:
    print v
```

This produces the following output:

```
1
39916800
51090942171709440000
8222838654177922817725562880000000
33452526613163807108170062053440751665152000000000
...
```

The other high-level interface, the `Manager` class, creates a separate server process that can hold master copies of Python data structures. Other processes can then access and modify these data structures using proxy objects. The following example creates a shared dictionary by calling the `dict()` method; the worker processes then insert values into the dictionary. (Locking is not done for you automatically, which doesn't matter in this example. `Manager`'s methods also include `Lock()`, `RLock()`, and `Semaphore()` to create shared locks.)

```
import time
from multiprocessing import Pool, Manager

def factorial(N, dictionary):
    "Compute a factorial."
    # Calculate the result
    fact = 1L
    for i in range(1, N+1):
        fact = fact * i

    # Store result in dictionary
    dictionary[N] = fact

if __name__ == '__main__':
    p = Pool(5)
    mgr = Manager()
    d = mgr.dict()         # Create shared dictionary

    # Run tasks using the pool
    for N in range(1, 1000, 10):
        p.apply_async(factorial, (N, d))

    # Mark pool as closed -- no more tasks can be added.
    p.close()

    # Wait for tasks to exit
    p.join()

    # Output results
    for k, v in sorted(d.items()):
        print k, v
```

This will produce the output:

```
1 1
11 39916800
21 51090942171709440000
31 8222838654177922817725562880000000
41 33452526613163807108170062053440751665152000000000
51
15511187532873822802242430164693032110632597200169861120000...
```

See also

The documentation for the `multiprocessing` module.

PEP 371 [https://peps.python.org/pep-0371/] - Addition of the multiprocessing package

PEP written by Jesse Noller and Richard Oudkerk; implemented by Richard Oudkerk and Jesse Noller.

PEP 3101: Advanced String Formatting

In Python 3.0, the `%` operator is supplemented by a more powerful string formatting method, `format()`. Support for the `str.format()` method has been backported to Python 2.6.

In 2.6, both 8-bit and Unicode strings have a `.format()` method that treats the string as a template and takes the arguments to be formatted. The formatting template uses curly brackets (`{`, `}`) as special characters:

```
>>> # Substitute positional argument 0 into the string.
>>> "User ID: {0}".format("root")
'User ID: root'
>>> # Use the named keyword arguments
>>> "User ID: {uid}   Last seen: {last_login}".format(
...    uid="root",
...    last_login = "5 Mar 2008 07:20")
'User ID: root   Last seen: 5 Mar 2008 07:20'
```

Curly brackets can be escaped by doubling them:

```
>>> "Empty dict: {{}}".format()
"Empty dict: {}"
```

Field names can be integers indicating positional arguments, such as `{0}`, `{1}`, etc. or names of keyword arguments. You can also supply compound field names that read attributes or access dictionary keys:

```
>>> import sys
>>> print 'Platform: {0.platform}\nPython version:
{0.version}'.format(sys)
Platform: darwin
Python version: 2.6a1+ (trunk:61261M, Mar  5 2008, 20:29:41)
[GCC 4.0.1 (Apple Computer, Inc. build 5367)]'

>>> import mimetypes
>>> 'Content-type: {0[.mp4]}'.format(mimetypes.types_map)
'Content-type: video/mp4'
```

Note that when using dictionary-style notation such as `[.mp4]`, you don’t need to put any quotation marks around the string; it will look up the value using `.mp4` as the key. Strings beginning with a number will be converted to an integer. You can’t write more complicated expressions inside a format string.

So far we’ve shown how to specify which field to substitute into the resulting string. The precise formatting used is also controllable by adding a colon followed by a format specifier. For example:

```
>>> # Field 0: left justify, pad to 15 characters
>>> # Field 1: right justify, pad to 6 characters
>>> fmt = '{0:15} ${1:>6}'
>>> fmt.format('Registration', 35)
'Registration    $    35'
>>> fmt.format('Tutorial', 50)
'Tutorial        $    50'
>>> fmt.format('Banquet', 125)
'Banquet         $   125'
```

Format specifiers can reference other fields through nesting:

```
>>> fmt = '{0:{1}}'
>>> width = 15
>>> fmt.format('Invoice #1234', width)
'Invoice #1234  '
>>> width = 35
>>> fmt.format('Invoice #1234', width)
'Invoice #1234                      '
```

The alignment of a field within the desired width can be specified:

Character	Effect
< (default)	Left-align
>	Right-align
^	Center
=	(For numeric types only) Pad after the sign.

Format specifiers can also include a presentation type, which controls how the value is formatted. For example, floating-point numbers can be formatted as a general number or in exponential notation:

```
>>> '{0:g}'.format(3.75)
'3.75'
>>> '{0:e}'.format(3.75)
'3.750000e+00'
```

A variety of presentation types are available. Consult the 2.6 documentation for a complete list; here’s a sample:

`b`	Binary. Outputs the number in base 2.
`c`	Character. Converts the integer to the corresponding Unicode character before printing.
`d`	Decimal Integer. Outputs the number in base 10.

`o`	Octal format. Outputs the number in base 8.
`x`	Hex format. Outputs the number in base 16, using lower-case letters for the digits above 9.
`e`	Exponent notation. Prints the number in scientific notation using the letter ‘e’ to indicate the exponent.
`g`	General format. This prints the number as a fixed-point number, unless the number is too large, in which case it switches to ‘e’ exponent notation.
`n`	Number. This is the same as ‘g’ (for floats) or ‘d’ (for integers), except that it uses the current locale setting to insert the appropriate number separator characters.
`%`	Percentage. Multiplies the number by 100 and displays in fixed (‘f’) format, followed by a percent sign.

Classes and types can define a `__format__()` method to control how they’re formatted. It receives a single argument, the format specifier:

```
def __format__(self, format_spec):
    if isinstance(format_spec, unicode):
        return unicode(str(self))
    else:
        return str(self)
```

There’s also a `format()` builtin that will format a single value. It calls the type’s `__format__()` method with the provided specifier:

```
>>> format(75.6564, '.2f')
'75.66'
```

See also

Format String Syntax
The reference documentation for format fields.

PEP 3101 [https://peps.python.org/pep-3101/] - Advanced String Formatting
PEP written by Talin. Implemented by Eric Smith.

PEP 3105: `print` As a Function

The `print` statement becomes the `print()` function in Python 3.0. Making `print()` a function makes it possible to replace the function by doing `def print(...)` or importing a new function from somewhere else.

Python 2.6 has a `__future__` import that removes `print` as language syntax, letting you use the functional form instead. For example:

```
>>> from __future__ import print_function
>>> print('# of entries', len(dictionary), file=sys.stderr)
```

The signature of the new function is:

```
def print(*args, sep=' ', end='\n', file=None)
```

The parameters are:

- *args*: positional arguments whose values will be printed out.

- *sep*: the separator, which will be printed between arguments.
- *end*: the ending text, which will be printed after all of the arguments have been output.
- *file*: the file object to which the output will be sent.

See also

PEP 3105 [https://peps.python.org/pep-3105/] - Make print a function
PEP written by Georg Brandl.

PEP 3110: Exception-Handling Changes

One error that Python programmers occasionally make is writing the following code:

```
try:
    ...
except TypeError, ValueError:  # Wrong!
    ...
```

The author is probably trying to catch both `TypeError` and `ValueError` exceptions, but this code actually does something different: it will catch `TypeError` and bind the resulting exception object to the local name `"ValueError"`. The `ValueError` exception will not be caught at all. The correct code specifies a tuple of exceptions:

```
try:
    ...
except (TypeError, ValueError):
    ...
```

This error happens because the use of the comma here is ambiguous: does it indicate two different nodes in the parse tree, or a single node that's a tuple?

Python 3.0 makes this unambiguous by replacing the comma with the word "as". To catch an exception and store the exception object in the variable `exc`, you must write:

```
try:
    ...
except TypeError as exc:
    ...
```

Python 3.0 will only support the use of "as", and therefore interprets the first example as catching two different exceptions. Python 2.6 supports both the comma and "as", so existing code will continue to work. We therefore suggest using "as" when writing new Python code that will only be executed with 2.6.

See also

PEP 3110 [https://peps.python.org/pep-3110/] - Catching Exceptions in Python 3000
PEP written and implemented by Collin Winter.

PEP 3112: Byte Literals

Python 3.0 adopts Unicode as the language's fundamental string type and denotes 8-bit literals differently, either as `b'string'` or using a `bytes` constructor. For future compatibility, Python 2.6 adds `bytes` as a synonym for the `str` type, and it also supports the `b''` notation.

The 2.6 `str` differs from 3.0’s `bytes` type in various ways; most notably, the constructor is completely different. In 3.0, `bytes([65, 66, 67])` is 3 elements long, containing the bytes representing `ABC`; in 2.6, `bytes([65, 66, 67])` returns the 12-byte string representing the `str()` of the list.

The primary use of `bytes` in 2.6 will be to write tests of object type such as `isinstance(x, bytes)`. This will help the 2to3 converter, which can’t tell whether 2.x code intends strings to contain either characters or 8-bit bytes; you can now use either `bytes` or `str` to represent your intention exactly, and the resulting code will also be correct in Python 3.0.

There’s also a `__future__` import that causes all string literals to become Unicode strings. This means that `\u` escape sequences can be used to include Unicode characters:

```
from __future__ import unicode_literals

s = ('\u751f\u3080\u304e\u3000\u751f\u3054'
     '\u3081\u3000\u751f\u305f\u307e\u3054')

print len(s)               # 12 Unicode characters
```

At the C level, Python 3.0 will rename the existing 8-bit string type, called **`PyStringObject`** in Python 2.x, to `PyBytesObject`. Python 2.6 uses `#define` to support using the names `PyBytesObject()`, `PyBytes_Check()`, `PyBytes_FromStringAndSize()`, and all the other functions and macros used with strings.

Instances of the `bytes` type are immutable just as strings are. A new `bytearray` type stores a mutable sequence of bytes:

```
>>> bytearray([65, 66, 67])
bytearray(b'ABC')
>>> b = bytearray(u'\u21ef\u3244', 'utf-8')
>>> b
```

```
bytearray(b'\xe2\x87\xaf\xe3\x89\x84')
>>> b[0] = '\xe3'
>>> b
bytearray(b'\xe3\x87\xaf\xe3\x89\x84')
>>> unicode(str(b), 'utf-8')
u'\u31ef \u3244'
```

Byte arrays support most of the methods of string types, such as `startswith()`/`endswith()`, `find()`/`rfind()`, and some of the methods of lists, such as `append()`, `pop()`, and `reverse()`.

```
>>> b = bytearray('ABC')
>>> b.append('d')
>>> b.append(ord('e'))
>>> b
bytearray(b'ABCde')
```

There's also a corresponding C API, with `PyByteArray_FromObject()`, `PyByteArray_FromStringAndSize()`, and various other functions.

See also

PEP 3112 [https://peps.python.org/pep-3112/] - Bytes literals in Python 3000

PEP written by Jason Orendorff; backported to 2.6 by Christian Heimes.

PEP 3116: New I/O Library

Python's built-in file objects support a number of methods, but file-like objects don't necessarily support all of them. Objects that imitate files usually support **`read()`** and **`write()`**, but they may not support **`readline()`**, for example. Python 3.0 introduces a layered I/O library in the `io` module that separates buffering and text-handling features from the fundamental read and write operations.

There are three levels of abstract base classes provided by the `io` module:

- `RawIOBase` defines raw I/O operations: `read()`, `readinto()`, `write()`, `seek()`, `tell()`, `truncate()`, and `close()`. Most of the methods of this class will often map to a single system call. There are also `readable()`, `writable()`, and `seekable()` methods for determining what operations a given object will allow.

  Python 3.0 has concrete implementations of this class for files and sockets, but Python 2.6 hasn't restructured its file and socket objects in this way.

- `BufferedIOBase` is an abstract base class that buffers data in memory to reduce the number of system calls used, making I/O processing more efficient. It supports all of the methods of `RawIOBase`, and adds a `raw` attribute holding the underlying raw object.

  There are five concrete classes implementing this ABC. `BufferedWriter` and `BufferedReader` are for objects that support write-only or read-only usage that have a `seek()` method for random access. `BufferedRandom` objects support read and write access upon the same underlying stream, and `BufferedRWPair` is for objects such as TTYs that have both read and write operations acting upon unconnected streams of data. The `BytesIO` class supports reading, writing, and seeking over an in-memory buffer.

- `TextIOBase`: Provides functions for reading and writing strings (remember, strings will be Unicode in Python 3.0), and supporting universal newlines. `TextIOBase` defines the `readline()` method and supports iteration upon objects.

There are two concrete implementations. `TextIOWrapper` wraps a buffered I/O object, supporting all of the methods for text I/O and adding a `buffer` attribute for access to the underlying object. `StringIO` simply buffers everything in memory without ever writing anything to disk.

(In Python 2.6, `io.StringIO` is implemented in pure Python, so it's pretty slow. You should therefore stick with the existing `StringIO` module or `cStringIO` for now. At some point Python 3.0's `io` module will be rewritten into C for speed, and perhaps the C implementation will be backported to the 2.x releases.)

In Python 2.6, the underlying implementations haven't been restructured to build on top of the `io` module's classes. The module is being provided to make it easier to write code that's forward-compatible with 3.0, and to save developers the effort of writing their own implementations of buffering and text I/O.

See also

PEP 3116 [https://peps.python.org/pep-3116/] - New I/O
PEP written by Daniel Stutzbach, Mike Verdone, and Guido van Rossum. Code by Guido van Rossum, Georg Brandl, Walter Doerwald, Jeremy Hylton, Martin von Löwis, Tony Lownds, and others.

PEP 3118: Revised Buffer Protocol

The buffer protocol is a C-level API that lets Python types exchange pointers into their internal representations. A memory-mapped file can be viewed as a buffer of characters, for example, and this lets another module such as `re` treat memory-mapped files as a string of characters to be searched.

The primary users of the buffer protocol are numeric-processing packages such as NumPy, which expose the internal representation of arrays so that callers can write data directly into an array instead of going through a slower API. This PEP updates the buffer protocol in light of experience from NumPy development, adding a number of new features such as indicating the shape of an array or locking a memory region.

The most important new C API function is `PyObject_GetBuffer(PyObject *obj, Py_buffer *view, int flags)`, which takes an object and a set of flags, and fills in the `Py_buffer` structure with information about the object's memory representation. Objects can use this operation to lock memory in place while an external caller could be modifying the contents, so there's a corresponding `PyBuffer_Release(Py_buffer *view)` to indicate that the external caller is done.

The *flags* argument to `PyObject_GetBuffer()` specifies constraints upon the memory returned. Some examples are:

- `PyBUF_WRITABLE` indicates that the memory must be writable.
- **`PyBUF_LOCK`** requests a read-only or exclusive lock on the memory.
- `PyBUF_C_CONTIGUOUS` and `PyBUF_F_CONTIGUOUS` requests a C-contiguous (last dimension varies the fastest) or

Fortran-contiguous (first dimension varies the fastest) array layout.

Two new argument codes for `PyArg_ParseTuple()`, `s*` and `z*`, return locked buffer objects for a parameter.

See also

PEP 3118 [https://peps.python.org/pep-3118/] - Revising the buffer protocol
PEP written by Travis Oliphant and Carl Banks; implemented by Travis Oliphant.

PEP 3119: Abstract Base Classes

Some object-oriented languages such as Java support interfaces, declaring that a class has a given set of methods or supports a given access protocol. Abstract Base Classes (or ABCs) are an equivalent feature for Python. The ABC support consists of an `abc` module containing a metaclass called `ABCMeta`, special handling of this metaclass by the `isinstance()` and `issubclass()` builtins, and a collection of basic ABCs that the Python developers think will be widely useful. Future versions of Python will probably add more ABCs.

Let’s say you have a particular class and wish to know whether it supports dictionary-style access. The phrase “dictionary-style” is vague, however. It probably means that accessing items with `obj[1]` works. Does it imply that setting items with `obj[2] = value` works? Or that the object will have **`keys()`**, **`values()`**, and **`items()`** methods? What about the iterative variants such as **`iterkeys()`**? **`copy`and :meth: ()`**!update`? Iterating over the object with **`iter()`**?

The Python 2.6 **`collections`** module includes a number of different ABCs that represent these distinctions. **`Iterable`** indicates that a class defines **`__iter__()`**, and **`Container`** means the class defines a **`__contains__()`** method and therefore supports `x in y` expressions. The basic dictionary interface of getting items, setting items, and **`keys()`**, **`values()`**, and **`items()`**, is defined by the **`MutableMapping`** ABC.

You can derive your own classes from a particular ABC to indicate they support that ABC’s interface:

```
import collections

class Storage(collections.MutableMapping):
    ...
```

Alternatively, you could write the class without deriving from the desired ABC and instead register the class by calling the ABC’s **`register()`** method:

```
import collections

class Storage:
    ...

collections.MutableMapping.register(Storage)
```

For classes that you write, deriving from the ABC is probably clearer. The **`register()`** method is useful when you’ve written a new ABC that can describe an existing type or class, or if you want to declare that some third-party class implements an ABC. For example, if you defined a **`PrintableType`** ABC, it’s legal to do:

```
# Register Python's types
PrintableType.register(int)
PrintableType.register(float)
PrintableType.register(str)
```

Classes should obey the semantics specified by an ABC, but Python can’t check this; it’s up to the class author to understand the ABC’s requirements and to implement the code accordingly.

To check whether an object supports a particular interface, you can now write:

```
def func(d):
    if not isinstance(d, collections.MutableMapping):
        raise ValueError("Mapping object expected, not %r" % d)
```

Don’t feel that you must now begin writing lots of checks as in the above example. Python has a strong tradition of duck-typing, where explicit type-checking is never done and code simply calls methods on an object, trusting that those methods will be there and raising an exception if they aren’t. Be judicious in checking for ABCs and only do it where it’s absolutely necessary.

You can write your own ABCs by using `abc.ABCMeta` as the metaclass in a class definition:

```
from abc import ABCMeta, abstractmethod

class Drawable():
    __metaclass__ = ABCMeta

    @abstractmethod
    def draw(self, x, y, scale=1.0):
        pass

    def draw_doubled(self, x, y):
        self.draw(x, y, scale=2.0)


class Square(Drawable):
    def draw(self, x, y, scale):
        ...
```

In the **`Drawable`** ABC above, the **`draw_doubled()`** method renders the object at twice its size and can be implemented in terms of other methods described in **`Drawable`**. Classes implementing this ABC therefore don’t need to provide their own implementation of **`draw_doubled()`**, though they can do so. An implementation of **`draw()`** is necessary, though; the ABC can’t provide a useful generic implementation.

You can apply the **`@~abc.abstractmethod`** decorator to methods such as **`draw()`** that must be implemented; Python will then raise an exception for classes that don’t define the method. Note that the exception is only raised when you actually try to create an instance of a subclass lacking the method:

```
>>> class Circle(Drawable):
...     pass
...
>>> c = Circle()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: Can't instantiate abstract class Circle with
abstract methods draw
>>>
```

Abstract data attributes can be declared using the `@abstractproperty` decorator:

```
from abc import abstractproperty
...

@abstractproperty
def readonly(self):
   return self._x
```

Subclasses must then define a `readonly` property.

See also

PEP 3119 [https://peps.python.org/pep-3119/] - Introducing Abstract Base Classes

PEP written by Guido van Rossum and Talin. Implemented by Guido van Rossum. Backported to 2.6 by Benjamin Aranguren, with Alex Martelli.

PEP 3127: Integer Literal Support and Syntax

Python 3.0 changes the syntax for octal (base-8) integer literals, prefixing them with “0o” or “0O” instead of a leading zero, and adds support for binary (base-2) integer literals, signalled by a “0b” or “0B” prefix.

Python 2.6 doesn’t drop support for a leading 0 signalling an octal number, but it does add support for “0o” and “0b”:

```
>>> 0o21, 2*8 + 1
(17, 17)
>>> 0b101111
47
```

The `oct()` builtin still returns numbers prefixed with a leading zero, and a new `bin()` builtin returns the binary representation for a number:

```
>>> oct(42)
'052'
>>> future_builtins.oct(42)
'0o52'
>>> bin(173)
'0b10101101'
```

The `int()` and **`long()`** builtins will now accept the “0o” and “0b” prefixes when base-8 or base-2 are requested, or when the *base* argument is zero (signalling that the base used should be determined from the string):

```
>>> int ('0o52', 0)
42
>>> int('1101', 2)
13
>>> int('0b1101', 2)
13
>>> int('0b1101', 0)
13
```

See also

[PEP 3127](https://peps.python.org/pep-3127/) [https://peps.python.org/pep-3127/] - Integer Literal Support and Syntax

PEP written by Patrick Maupin; backported to 2.6 by Eric Smith.

PEP 3129: Class Decorators

Decorators have been extended from functions to classes. It’s now legal to write:

```
@foo
@bar
class A:
  pass
```

This is equivalent to:

```
class A:
  pass

A = foo(bar(A))
```

See also

[PEP 3129](https://peps.python.org/pep-3129/) [https://peps.python.org/pep-3129/] - Class Decorators

PEP written by Collin Winter.

PEP 3141: A Type Hierarchy for Numbers

Python 3.0 adds several abstract base classes for numeric types inspired by Scheme’s numeric tower. These classes were backported to 2.6 as the `numbers` module.

The most general ABC is **`Number`**. It defines no operations at all, and only exists to allow checking if an object is a number by doing `isinstance(obj, Number)`.

`Complex` is a subclass of **`Number`**. Complex numbers can undergo the basic operations of addition, subtraction, multiplication, division, and exponentiation, and you can retrieve the real and imaginary parts and obtain a number’s conjugate. Python’s built-in complex type is an implementation of **`Complex`**.

`Real` further derives from **`Complex`**, and adds operations that only work on real numbers: **`floor()`**, **`trunc()`**, rounding, taking the remainder mod N, floor division, and comparisons.

`Rational` numbers derive from **`Real`**, have **`numerator`** and **`denominator`** properties, and can be converted to floats. Python 2.6 adds a simple rational-number class, **`Fraction`**, in the `fractions` module. (It’s called **`Fraction`** instead of **`Rational`** to avoid a name clash with `numbers.Rational`.)

`Integral` numbers derive from **`Rational`**, and can be shifted left and right with << and >>, combined using bitwise operations such as `&` and `|`, and can be used as array indexes and slice boundaries.

In Python 3.0, the PEP slightly redefines the existing builtins `round()`, `math.floor()`, `math.ceil()`, and adds a new one, `math.trunc()`, that’s been backported to Python 2.6. `math.trunc()` rounds toward zero, returning the closest **`Integral`** that’s between the function’s argument and zero.

See also

PEP 3141 [https://peps.python.org/pep-3141/] - A Type Hierarchy for Numbers
PEP written by Jeffrey Yasskin.

Scheme’s numerical tower [https://www.gnu.org/software/guile/manual/html_node/Numerical-Tower.html#Numerical-Tower], from the Guile manual.

Scheme’s number datatypes [https://conservatory.scheme.org/schemers/Documents/Standards/R5RS/HTML/r5rs-Z-H-9.html#%_sec_6.2] from the R5RS Scheme specification.

The `fractions` Module

To fill out the hierarchy of numeric types, the `fractions` module provides a rational-number class. Rational numbers store their values as a numerator and denominator forming a fraction, and can exactly represent numbers such as `2/3` that floating-point numbers can only approximate.

The **`Fraction`** constructor takes two **`Integral`** values that will be the numerator and denominator of the resulting fraction.

```
>>> from fractions import Fraction
>>> a = Fraction(2, 3)
>>> b = Fraction(2, 5)
>>> float(a), float(b)
(0.66666666666666663, 0.40000000000000002)
>>> a+b
Fraction(16, 15)
>>> a/b
Fraction(5, 3)
```

For converting floating-point numbers to rationals, the float type now has an **`as_integer_ratio()`** method that returns the numerator and denominator for a fraction that evaluates to the same floating-point value:

```
>>> (2.5) .as_integer_ratio()
(5, 2)
>>> (3.1415) .as_integer_ratio()
(7074029114692207L, 2251799813685248L)
>>> (1./3) .as_integer_ratio()
(6004799503160661L, 18014398509481984L)
```

Note that values that can only be approximated by floating-point numbers, such as 1./3, are not simplified to the number being approximated; the fraction attempts to match the floating-point value **exactly**.

The **`fractions`** module is based upon an implementation by Sjoerd Mullender that was in Python’s `Demo/classes/` directory for a long time. This implementation was significantly updated by Jeffrey Yasskin.

Other Language Changes

Some smaller changes made to the core Python language are:

- Directories and zip archives containing a `__main__.py` file can now be executed directly by passing their name to the interpreter. The directory or zip archive is automatically inserted as the first entry in sys.path. (Suggestion and initial patch by Andy Chu, subsequently revised by Phillip J. Eby and Nick Coghlan; bpo-1739468 [https://bugs.python.org/issue?@action=redirect&bpo=1739468].)

- The `hasattr()` function was catching and ignoring all errors, under the assumption that they meant a **`__getattr__()`** method was failing somehow and the return value of `hasattr()` would therefore be `False`. This logic shouldn't be applied to `KeyboardInterrupt` and `SystemExit`, however; Python 2.6 will no longer discard such exceptions when `hasattr()` encounters them. (Fixed by Benjamin Peterson; bpo-2196 [https://bugs.python.org/issue?@action=redirect&bpo=2196].)
- When calling a function using the `**` syntax to provide keyword arguments, you are no longer required to use a Python dictionary; any mapping will now work:

  ```
  >>> def f(**kw):
  ...    print sorted(kw)
  ...
  >>> ud=UserDict.UserDict()
  >>> ud['a'] = 1
  >>> ud['b'] = 'string'
  >>> f(**ud)
  ['a', 'b']
  ```

  (Contributed by Alexander Belopolsky; bpo-1686487 [https://bugs.python.org/issue?@action=redirect&bpo=1686487].)

  It's also become legal to provide keyword arguments after a `*args` argument to a function call.

  ```
  >>> def f(*args, **kw):
  ...     print args, kw
  ...
  >>> f(1,2,3, *(4,5,6), keyword=13)
  (1, 2, 3, 4, 5, 6) {'keyword': 13}
  ```

Previously this would have been a syntax error. (Contributed by Amaury Forgeot d'Arc; bpo-3473 [https://bugs.python.org/issue?@action=redirect&bpo=3473].)

- A new builtin, `next(iterator, [default])` returns the next item from the specified iterator. If the *default* argument is supplied, it will be returned if *iterator* has been exhausted; otherwise, the **`StopIteration`** exception will be raised. (Backported in bpo-2719 [https://bugs.python.org/issue?@action=redirect&bpo=2719].)
- Tuples now have **`index()`** and **`count()`** methods matching the list type's **`index()`** and **`count()`** methods:

  ```
  >>> t = (0,1,2,3,4,0,1,2)
  >>> t.index(3)
  3
  >>> t.count(0)
  2
  ```

  (Contributed by Raymond Hettinger)
- The built-in types now have improved support for extended slicing syntax, accepting various combinations of `(start, stop, step)`. Previously, the support was partial and certain corner cases wouldn't work. (Implemented by Thomas Wouters.)
- Properties now have three attributes, **`getter`**, **`setter`** and **`deleter`**, that are decorators providing useful shortcuts for adding a getter, setter or deleter function to an existing property. You would use them like this:

  ```
  class C(object):
      @property
      def x(self):
          return self._x

      @x.setter
  ```

```
    def x(self, value):
        self._x = value

    @x.deleter
    def x(self):
        del self._x

class D(C):
    @C.x.getter
    def x(self):
        return self._x * 2

    @x.setter
    def x(self, value):
        self._x = value / 2
```

- Several methods of the built-in set types now accept multiple iterables: **`intersection()`**, **`intersection_update()`**, **`union()`**, **`update()`**, **`difference()`** and **`difference_update()`**.

  ```
  >>> s=set('1234567890')
  >>> s.intersection('abc123', 'cdf246')  # Intersection
  between all inputs
  set(['2'])
  >>> s.difference('246', '789')
  set(['1', '0', '3', '5'])
  ```

  (Contributed by Raymond Hettinger.)

- Many floating-point features were added. The **`float()`** function will now turn the string `nan` into an IEEE 754 Not A Number value, and `+inf` and `-inf` into positive or negative infinity. This works on any platform with IEEE 754 semantics. (Contributed by Christian Heimes; bpo-1635 [https://bugs.python.org/issue?@action=redirect&bpo=1635].)

  Other functions in the **`math`** module, **`isinf()`** and **`isnan()`**, return true if their floating-point argument is infinite or Not A Number. (bpo-1640 [https://bugs.python.org/issue?@action=redirect&bpo=1640])

Conversion functions were added to convert floating-point numbers into hexadecimal strings (bpo-3008 [https://bugs.python.org/issue?@action=redirect&bpo=3008]). These functions convert floats to and from a string representation without introducing rounding errors from the conversion between decimal and binary. Floats have a **`hex()`** method that returns a string representation, and the `float.fromhex()` method converts a string back into a number:

```
>>> a = 3.75
>>> a.hex()
'0x1.e000000000000p+1'
>>> float.fromhex('0x1.e000000000000p+1')
3.75
>>> b=1./3
>>> b.hex()
'0x1.5555555555555p-2'
```

- A numerical nicety: when creating a complex number from two floats on systems that support signed zeros (-0 and +0), the **`complex()`** constructor will now preserve the sign of the zero. (Fixed by Mark T. Dickinson; bpo-1507 [https://bugs.python.org/issue?@action=redirect&bpo=1507].)

- Classes that inherit a **`__hash__()`** method from a parent class can set `__hash__ = None` to indicate that the class isn’t hashable. This will make `hash(obj)` raise a **`TypeError`** and the class will not be indicated as implementing the **`Hashable`** ABC.

  You should do this when you’ve defined a **`__cmp__()`** or **`__eq__()`** method that compares objects by their value rather than by identity. All objects have a default hash method that uses `id(obj)` as the hash value. There’s no tidy way to remove the **`__hash__()`** method inherited from a parent class, so assigning `None` was implemented as an override. At the C level, extensions can set `tp_hash` to

`PyObject_HashNotImplemented()`. (Fixed by Nick Coghlan and Amaury Forgeot d'Arc; bpo-2235 [https://bugs.python.org/issue?@action=redirect&bpo=2235].)

- The **`GeneratorExit`** exception now subclasses **`BaseException`** instead of **`Exception`**. This means that an exception handler that does `except Exception:` will not inadvertently catch **`GeneratorExit`**. (Contributed by Chad Austin; bpo-1537 [https://bugs.python.org/issue?@action=redirect&bpo=1537].)
- Generator objects now have a **`gi_code`** attribute that refers to the original code object backing the generator. (Contributed by Collin Winter; bpo-1473257 [https://bugs.python.org/issue?@action=redirect&bpo=1473257].)
- The **`compile()`** built-in function now accepts keyword arguments as well as positional parameters. (Contributed by Thomas Wouters; bpo-1444529 [https://bugs.python.org/issue?@action=redirect&bpo=1444529].)
- The **`complex()`** constructor now accepts strings containing parenthesized complex numbers, meaning that `complex(repr(cplx))` will now round-trip values. For example, `complex('(3+4j)')` now returns the value (3+4j). (bpo-1491866 [https://bugs.python.org/issue?@action=redirect&bpo=1491866])
- The string **`translate()`** method now accepts `None` as the translation table parameter, which is treated as the identity transformation. This makes it easier to carry out operations that only delete characters. (Contributed by Bengt Richter and implemented by Raymond Hettinger; bpo-1193128 [https://bugs.python.org/issue?@action=redirect&bpo=1193128].)
- The built-in **`dir()`** function now checks for a **`__dir__()`** method on the objects it receives. This method must

return a list of strings containing the names of valid attributes for the object, and lets the object control the value that `dir()` produces. Objects that have `__getattr__()` or `__getattribute__()` methods can use this to advertise pseudo-attributes they will honor. (bpo-1591665 [https://bugs.python.org/issue?@action=redirect&bpo=1591665])

- Instance method objects have new attributes for the object and function comprising the method; the new synonym for `im_self` is `__self__`, and `im_func` is also available as `__func__`. The old names are still supported in Python 2.6, but are gone in 3.0.

- An obscure change: when you use the `locals()` function inside a `class` statement, the resulting dictionary no longer returns free variables. (Free variables, in this case, are variables referenced in the `class` statement that aren’t attributes of the class.)

Optimizations

- The `warnings` module has been rewritten in C. This makes it possible to invoke warnings from the parser, and may also make the interpreter’s startup faster. (Contributed by Neal Norwitz and Brett Cannon; bpo-1631171 [https://bugs.python.org/issue?@action=redirect&bpo=1631171].)

- Type objects now have a cache of methods that can reduce the work required to find the correct method implementation for a particular class; once cached, the interpreter doesn’t need to traverse base classes to figure out the right method to call. The cache is cleared if a base class or the class itself is modified, so the cache should remain correct even in the face of Python’s dynamic nature. (Original optimization implemented by Armin Rigo, updated for Python 2.6 by Kevin Jacobs; bpo-1700288 [https://bugs.python.org/issue?@action=redirect&bpo=1700288].)

By default, this change is only applied to types that are included with the Python core. Extension modules may not necessarily be compatible with this cache, so they must explicitly add **`Py_TPFLAGS_HAVE_VERSION_TAG`** to the module's `tp_flags` field to enable the method cache. (To be compatible with the method cache, the extension module's code must not directly access and modify the `tp_dict` member of any of the types it implements. Most modules don't do this, but it's impossible for the Python interpreter to determine that. See bpo-1878 [https://bugs.python.org/issue?@action=redirect&bpo=1878] for some discussion.)

- Function calls that use keyword arguments are significantly faster by doing a quick pointer comparison, usually saving the time of a full string comparison. (Contributed by Raymond Hettinger, after an initial implementation by Antoine Pitrou; bpo-1819 [https://bugs.python.org/issue?@action=redirect&bpo=1819].)
- All of the functions in the **`struct`** module have been rewritten in C, thanks to work at the Need For Speed sprint. (Contributed by Raymond Hettinger.)
- Some of the standard built-in types now set a bit in their type objects. This speeds up checking whether an object is a subclass of one of these types. (Contributed by Neal Norwitz.)
- Unicode strings now use faster code for detecting whitespace and line breaks; this speeds up the **`split()`** method by about 25% and **`splitlines()`** by 35%.

(Contributed by Antoine Pitrou.) Memory usage is reduced by using pymalloc for the Unicode string's data.

- The `with` statement now stores the `__exit__()` method on the stack, producing a small speedup. (Implemented by Jeffrey Yasskin.)
- To reduce memory usage, the garbage collector will now clear internal free lists when garbage-collecting the highest generation of objects. This may return memory to the operating system sooner.

Interpreter Changes

Two command-line options have been reserved for use by other Python implementations. The **`-J`** switch has been reserved for use by Jython for Jython-specific options, such as switches that are passed to the underlying JVM. **`-X`** has been reserved for options specific to a particular implementation of Python such as CPython, Jython, or IronPython. If either option is used with Python 2.6, the interpreter will report that the option isn't currently used.

Python can now be prevented from writing `.pyc` or `.pyo` files by supplying the **`-B`** switch to the Python interpreter, or by setting the **`PYTHONDONTWRITEBYTECODE`** environment variable before running the interpreter. This setting is available to Python programs as the `sys.dont_write_bytecode` variable, and Python code can change the value to modify the interpreter's behaviour. (Contributed by Neal Norwitz and Georg Brandl.)

The encoding used for standard input, output, and standard error can be specified by setting the **`PYTHONIOENCODING`** environment variable before running the interpreter. The value should be a string in the form `<encoding>` or `<encoding>:`

`<errorhandler>`. The *encoding* part specifies the encoding's name, e.g. `utf-8` or `latin-1`; the optional *errorhandler* part specifies what to do with characters that can't be handled by the encoding, and should be one of "error", "ignore", or "replace". (Contributed by Martin von Löwis.)

New and Improved Modules

As in every release, Python's standard library received a number of enhancements and bug fixes. Here's a partial list of the most notable changes, sorted alphabetically by module name. Consult the `Misc/NEWS` file in the source tree for a more complete list of changes, or look through the Subversion logs for all the details.

- The **`asyncore`** and **`asynchat`** modules are being actively maintained again, and a number of patches and bugfixes were applied. (Maintained by Josiah Carlson; see bpo-1736190 [https://bugs.python.org/issue?@action=redirect&bpo=1736190] for one patch.)
- The **`bsddb`** module also has a new maintainer, Jesús Cea Avión, and the package is now available as a standalone package. The web page for the package is www.jcea.es/programacion/pybsddb.htm [https://www.jcea.es/programacion/pybsddb.htm]. The plan is to remove the package from the standard library in Python 3.0, because its pace of releases is much more frequent than Python's.

  The **`bsddb.dbshelve`** module now uses the highest pickling protocol available, instead of restricting itself to protocol 1. (Contributed by W. Barnes.)
- The **`cgi`** module will now read variables from the query string of an HTTP POST request. This makes it possible

to use form actions with URLs that include query strings such as “/cgi-bin/add.py?category=1”. (Contributed by Alexandre Fiori and Nubis; bpo-1817 [https://bugs.python.org/issue?@action=redirect&bpo=1817].)

The `parse_qs()` and `parse_qsl()` functions have been relocated from the `cgi` module to the `urlparse` module. The versions still available in the `cgi` module will trigger `PendingDeprecationWarning` messages in 2.6 (bpo-600362 [https://bugs.python.org/issue?@action=redirect&bpo=600362]).

- The `cmath` module underwent extensive revision, contributed by Mark Dickinson and Christian Heimes. Five new functions were added:

  - `polar()` converts a complex number to polar form, returning the modulus and argument of the complex number.
  - `rect()` does the opposite, turning a modulus, argument pair back into the corresponding complex number.
  - `phase()` returns the argument (also called the angle) of a complex number.
  - `isnan()` returns True if either the real or imaginary part of its argument is a NaN.
  - `isinf()` returns True if either the real or imaginary part of its argument is infinite.

  The revisions also improved the numerical soundness of the `cmath` module. For all functions, the real and imaginary parts of the results are accurate to within a few units of least precision (ulps) whenever possible. See bpo-1381 [https://bugs.python.org/issue?@action=redirect&bpo=1381] for the details. The branch cuts for `asinh()`, `atanh()`: and `atan()` have also been corrected.

The tests for the module have been greatly expanded; nearly 2000 new test cases exercise the algebraic functions.

On IEEE 754 platforms, the `cmath` module now handles IEEE 754 special values and floating-point exceptions in a manner consistent with Annex ‘G’ of the C99 standard.

- A new data type in the `collections` module: `namedtuple(typename, fieldnames)` is a factory function that creates subclasses of the standard tuple whose fields are accessible by name as well as index. For example:

```
>>> var_type = collections.namedtuple('variable',
...             'id name type size')
>>> # Names are separated by spaces or commas.
>>> # 'id, name, type, size' would also work.
>>> var_type._fields
('id', 'name', 'type', 'size')

>>> var = var_type(1, 'frequency', 'int', 4)
>>> print var[0], var.id    # Equivalent
1 1
>>> print var[2], var.type  # Equivalent
int int
>>> var._asdict()
{'size': 4, 'type': 'int', 'id': 1, 'name': 'frequency'}
>>> v2 = var._replace(name='amplitude')
>>> v2
variable(id=1, name='amplitude', type='int', size=4)
```

Several places in the standard library that returned tuples have been modified to return **`namedtuple()`** instances. For example, the **`Decimal.as_tuple()`** method now returns a named tuple with **`sign`**, **`digits`**, and **`exponent`** fields.

(Contributed by Raymond Hettinger.)

- Another change to the **`collections`** module is that the **`deque`** type now supports an optional *maxlen* parameter; if supplied, the deque's size will be restricted to no more than *maxlen* items. Adding more items to a full deque causes old items to be discarded.

  ```
  >>> from collections import deque
  >>> dq=deque(maxlen=3)
  >>> dq
  deque([], maxlen=3)
  >>> dq.append(1); dq.append(2); dq.append(3)
  >>> dq
  deque([1, 2, 3], maxlen=3)
  >>> dq.append(4)
  >>> dq
  deque([2, 3, 4], maxlen=3)
  ```

  (Contributed by Raymond Hettinger.)

- The **`Cookie`** module's **`Morsel`** objects now support an **`httponly`** attribute. In some browsers. cookies with this attribute set cannot be accessed or manipulated by JavaScript code. (Contributed by Arvin Schnell; bpo-1638033 [https://bugs.python.org/issue?@action=redirect&bpo=1638033].)

- A new window method in the **`curses`** module, **`chgat()`**, changes the display attributes for a certain number of characters on a single line. (Contributed by Fabian Kreutz.)

  ```
  # Boldface text starting at y=0,x=21
  # and affecting the rest of the line.
  stdscr.chgat(0, 21, curses.A_BOLD)
  ```

  The **`Textbox`** class in the **`curses.textpad`** module now supports editing in insert mode as well as overwrite mode. Insert mode is enabled by supplying a true value for the *insert_mode* parameter when creating the **`Textbox`** instance.

- The **`datetime`** module's **`strftime()`** methods now support a `%f` format code that expands to the number of microseconds in the object, zero-padded on the left to six places. (Contributed by Skip Montanaro; bpo-1158 [https://bugs.python.org/issue?@action=redirect&bpo=1158].)

- The **`decimal`** module was updated to version 1.66 of the General Decimal Specification [https://speleotrove.com/decimal/decarith.html]. New features include some methods for some basic mathematical functions such as **`exp()`** and **`log10()`**:

  ```
  >>> Decimal(1).exp()
  Decimal("2.718281828459045235360287471")
  >>> Decimal("2.7182818").ln()
  Decimal("0.9999999895305022877376682436")
  >>> Decimal(1000).log10()
  Decimal("3")
  ```

  The **`as_tuple()`** method of **`Decimal`** objects now returns a named tuple with **`sign`**, **`digits`**, and **`exponent`** fields.

  (Implemented by Facundo Batista and Mark Dickinson. Named tuple support added by Raymond Hettinger.)

- The **`difflib`** module's **`SequenceMatcher`** class now returns named tuples representing matches, with **`a`**, **`b`**, and **`size`** attributes. (Contributed by Raymond Hettinger.)

- An optional `timeout` parameter, specifying a timeout measured in seconds, was added to the **`ftplib.FTP`** class constructor as well as the **`connect()`** method. (Added by Facundo Batista.) Also, the **`FTP`** class's **`storbinary()`** and **`storlines()`** now take an optional *callback* parameter that will be called with each block of data after the data has been sent. (Contributed by Phil Schwartz; bpo-1221598 [https://bugs.python.org/issue?@action=redirect&bpo=1221598].)

- The **`reduce()`** built-in function is also available in the **`functools`** module. In Python 3.0, the builtin has been dropped and **`reduce()`** is only available from **`functools`**; currently there are no plans to drop the builtin in the 2.x series. (Patched by Christian Heimes; bpo-1739906 [https://bugs.python.org/issue?@action=redirect&bpo=1739906].)
- When possible, the **`getpass`** module will now use `/dev/tty` to print a prompt message and read the password, falling back to standard error and standard input. If the password may be echoed to the terminal, a warning is printed before the prompt is displayed. (Contributed by Gregory P. Smith.)
- The **`glob.glob()`** function can now return Unicode filenames if a Unicode path was used and Unicode filenames are matched within the directory. (bpo-1001604 [https://bugs.python.org/issue?@action=redirect&bpo=1001604])
- A new function in the **`heapq`** module, `merge(iter1, iter2, ...)`, takes any number of iterables returning data in sorted order, and returns a new generator that returns the contents of all the iterators, also in sorted order. For example:

  ```
  >>> list(heapq.merge([1, 3, 5, 9], [2, 8, 16]))
  [1, 2, 3, 5, 8, 9, 16]
  ```

  Another new function, `heappushpop(heap, item)`, pushes *item* onto *heap*, then pops off and returns the smallest item. This is more efficient than making a call to **`heappush()`** and then **`heappop()`**.

heapq is now implemented to only use less-than comparison, instead of the less-than-or-equal comparison it previously used. This makes **heapq**’s usage of a type match the **`list.sort()`** method. (Contributed by Raymond Hettinger.)

- An optional `timeout` parameter, specifying a timeout measured in seconds, was added to the **`httplib.HTTPConnection`** and **`HTTPSConnection`** class constructors. (Added by Facundo Batista.)
- Most of the **`inspect`** module’s functions, such as **`getmoduleinfo()`** and **`getargs()`**, now return named tuples. In addition to behaving like tuples, the elements of the return value can also be accessed as attributes. (Contributed by Raymond Hettinger.)

  Some new functions in the module include **`isgenerator()`**, **`isgeneratorfunction()`**, and **`isabstract()`**.
- The **`itertools`** module gained several new functions.

  `izip_longest(iter1, iter2, ...[, fillvalue])` makes tuples from each of the elements; if some of the iterables are shorter than others, the missing values are set to *fillvalue*. For example:

  ```
  >>> tuple(itertools.izip_longest([1,2,3], [1,2,3,4,5]))
  ((1, 1), (2, 2), (3, 3), (None, 4), (None, 5))
  ```

  `product(iter1, iter2, ..., [repeat=N])` returns the Cartesian product of the supplied iterables, a set of tuples containing every possible combination of the elements returned from each iterable.

  ```
  >>> list(itertools.product([1,2,3], [4,5,6]))
  [(1, 4), (1, 5), (1, 6),
  ```

```
 (2, 4), (2, 5), (2, 6),
 (3, 4), (3, 5), (3, 6)]
```

The optional *repeat* keyword argument is used for taking the product of an iterable or a set of iterables with themselves, repeated *N* times. With a single iterable argument, *N*-tuples are returned:

```
>>> list(itertools.product([1,2], repeat=3))
[(1, 1, 1), (1, 1, 2), (1, 2, 1), (1, 2, 2),
 (2, 1, 1), (2, 1, 2), (2, 2, 1), (2, 2, 2)]
```

With two iterables, *2N*-tuples are returned.

```
>>> list(itertools.product([1,2], [3,4], repeat=2))
[(1, 3, 1, 3), (1, 3, 1, 4), (1, 3, 2, 3), (1, 3, 2, 4),
 (1, 4, 1, 3), (1, 4, 1, 4), (1, 4, 2, 3), (1, 4, 2, 4),
 (2, 3, 1, 3), (2, 3, 1, 4), (2, 3, 2, 3), (2, 3, 2, 4),
 (2, 4, 1, 3), (2, 4, 1, 4), (2, 4, 2, 3), (2, 4, 2, 4)]
```

`combinations(iterable, r)` returns sub-sequences of length *r* from the elements of *iterable*.

```
>>> list(itertools.combinations('123', 2))
[('1', '2'), ('1', '3'), ('2', '3')]
>>> list(itertools.combinations('123', 3))
[('1', '2', '3')]
>>> list(itertools.combinations('1234', 3))
[('1', '2', '3'), ('1', '2', '4'),
 ('1', '3', '4'), ('2', '3', '4')]
```

`permutations(iter[, r])` returns all the permutations of length *r* of the iterable’s elements. If *r* is not specified, it will default to the number of elements produced by the iterable.

```
>>> list(itertools.permutations([1,2,3,4], 2))
[(1, 2), (1, 3), (1, 4),
 (2, 1), (2, 3), (2, 4),
 (3, 1), (3, 2), (3, 4),
 (4, 1), (4, 2), (4, 3)]
```

`itertools.chain(*iterables)` is an existing function in **itertools** that gained a new constructor in Python 2.6. `itertools.chain.from_iterable(iterable)` takes a single iterable that should return other iterables. **`chain()`** will then return all the elements of the first iterable, then all the elements of the second, and so on.

```
>>> list(itertools.chain.from_iterable([[1,2,3], [4,5,6]]))
[1, 2, 3, 4, 5, 6]
```

(All contributed by Raymond Hettinger.)

- The **logging** module’s **FileHandler** class and its subclasses **WatchedFileHandler**, **RotatingFileHandler**, and **TimedRotatingFileHandler** now have an optional *delay* parameter to their constructors. If *delay* is true, opening of the log file is deferred until the first **`emit()`** call is made. (Contributed by Vinay Sajip.)

  TimedRotatingFileHandler also has a *utc* constructor parameter. If the argument is true, UTC time will be used in determining when midnight occurs and in generating filenames; otherwise local time will be used.

- Several new functions were added to the **math** module:
  - **isinf()** and **isnan()** determine whether a given float is a (positive or negative) infinity or a NaN (Not a Number), respectively.
  - **copysign()** copies the sign bit of an IEEE 754 number, returning the absolute value of *x* combined with the sign bit of *y*. For example, `math.copysign(1, -0.0)` returns -1.0. (Contributed by Christian Heimes.)
  - **factorial()** computes the factorial of a number. (Contributed by Raymond Hettinger; bpo-2138 [https://bugs.python.org/issue?@action=redirect&bpo=2138].)

  - **`fsum()`** adds up the stream of numbers from an iterable, and is careful to avoid loss of precision through using partial sums. (Contributed by Jean Brouwers, Raymond Hettinger, and Mark Dickinson; bpo-2819 [https://bugs.python.org/issue?@action=redirect&bpo=2819].)
  - **`acosh()`**, **`asinh()`** and **`atanh()`** compute the inverse hyperbolic functions.
  - **`log1p()`** returns the natural logarithm of *1+x* (base *e*).
  - **`trunc()`** rounds a number toward zero, returning the closest **`Integral`** that's between the function's argument and zero. Added as part of the backport of PEP 3141's type hierarchy for numbers.

- The **`math`** module has been improved to give more consistent behaviour across platforms, especially with respect to handling of floating-point exceptions and IEEE 754 special values.

  Whenever possible, the module follows the recommendations of the C99 standard about 754's special values. For example, `sqrt(-1.)` should now give a **`ValueError`** across almost all platforms, while `sqrt(float('NaN'))` should return a NaN on all IEEE 754 platforms. Where Annex 'F' of the C99 standard recommends signaling 'divide-by-zero' or 'invalid', Python will raise **`ValueError`**. Where Annex 'F' of the C99 standard recommends signaling 'overflow', Python will raise **`OverflowError`**. (See bpo-711019 [https://bugs.python.org/issue?@action=redirect&bpo=711019] and bpo-1640 [https://bugs.python.org/issue?@action=redirect&bpo=1640].)

  (Contributed by Christian Heimes and Mark Dickinson.)

- `mmap` objects now have a **`rfind()`** method that searches for a substring beginning at the end of the string and searching backwards. The **`find()`** method also gained an *end* parameter giving an index at which to stop searching. (Contributed by John Lenton.)

- The `operator` module gained a **`methodcaller()`** function that takes a name and an optional set of arguments, returning a callable that will call the named function on any arguments passed to it. For example:

  ```
  >>> # Equivalent to lambda s: s.replace('old', 'new')
  >>> replacer = operator.methodcaller('replace', 'old',
  'new')
  >>> replacer('old wine in old bottles')
  'new wine in new bottles'
  ```

  (Contributed by Georg Brandl, after a suggestion by Gregory Petrosyan.)

  The **`attrgetter()`** function now accepts dotted names and performs the corresponding attribute lookups:

  ```
  >>> inst_name = operator.attrgetter(
  ...        '__class__.__name__')
  >>> inst_name('')
  'str'
  >>> inst_name(help)
  '_Helper'
  ```

  (Contributed by Georg Brandl, after a suggestion by Barry Warsaw.)

- The `os` module now wraps several new system calls. `fchmod(fd, mode)` and `fchown(fd, uid, gid)` change the mode and ownership of an opened file, and `lchmod(path, mode)` changes the mode of a symlink. (Contributed by Georg Brandl and Christian Heimes.)

`chflags()` and **`lchflags()`** are wrappers for the corresponding system calls (where they’re available), changing the flags set on a file. Constants for the flag values are defined in the **`stat`** module; some possible values include **`UF_IMMUTABLE`** to signal the file may not be changed and **`UF_APPEND`** to indicate that data can only be appended to the file. (Contributed by M. Levinson.)

`os.closerange(low, high)` efficiently closes all file descriptors from *low* to *high*, ignoring any errors and not including *high* itself. This function is now used by the **`subprocess`** module to make starting processes faster. (Contributed by Georg Brandl; bpo-1663329 [https://bugs.python.org/issue?@action=redirect&bpo=1663329].)

- The `os.environ` object’s **`clear()`** method will now unset the environment variables using **`os.unsetenv()`** in addition to clearing the object’s keys. (Contributed by Martin Horcicka; bpo-1181 [https://bugs.python.org/issue?@action=redirect&bpo=1181].)
- The **`os.walk()`** function now has a `followlinks` parameter. If set to True, it will follow symlinks pointing to directories and visit the directory’s contents. For backward compatibility, the parameter’s default value is false. Note that the function can fall into an infinite recursion if there’s a symlink that points to a parent directory. (bpo-1273829 [https://bugs.python.org/issue?@action=redirect&bpo=1273829])
- In the **`os.path`** module, the **`splitext()`** function has been changed to not split on leading period characters. This produces better results when operating on Unix’s dot-files. For example, `os.path.splitext('.ipython')` now returns `('.ipython', '')` instead of `('', '.ipython')`. (bpo-

1115886 [https://bugs.python.org/issue?@action=redirect&bpo=1115886])

A new function, `os.path.relpath(path, start='.')`, returns a relative path from the `start` path, if it's supplied, or from the current working directory to the destination `path`. (Contributed by Richard Barran; bpo-1339796 [https://bugs.python.org/issue?@action=redirect&bpo=1339796].)

On Windows, **`os.path.expandvars()`** will now expand environment variables given in the form “%var%”, and “~user” will be expanded into the user's home directory path. (Contributed by Josiah Carlson; bpo-957650 [https://bugs.python.org/issue?@action=redirect&bpo=957650].)

- The Python debugger provided by the **`pdb`** module gained a new command: “run” restarts the Python program being debugged and can optionally take new command-line arguments for the program. (Contributed by Rocky Bernstein; bpo-1393667 [https://bugs.python.org/issue?@action=redirect&bpo=1393667].)

- The **`pdb.post_mortem()`** function, used to begin debugging a traceback, will now use the traceback returned by **`sys.exc_info()`** if no traceback is supplied. (Contributed by Facundo Batista; bpo-1106316 [https://bugs.python.org/issue?@action=redirect&bpo=1106316].)

- The **`pickletools`** module now has an **`optimize()`** function that takes a string containing a pickle and removes some unused opcodes, returning a shorter pickle that contains the same data structure. (Contributed by Raymond Hettinger.)

- A **`get_data()`** function was added to the **`pkgutil`** module that returns the contents of resource files included with an installed Python package. For example:

```
>>> import pkgutil
>>> print pkgutil.get_data('test',
'exception_hierarchy.txt')
BaseException
 +-- SystemExit
 +-- KeyboardInterrupt
 +-- GeneratorExit
 +-- Exception
      +-- StopIteration
      +-- StandardError
 ...
```

(Contributed by Paul Moore; bpo-2439 [https://bugs.python.org/issue?@action=redirect&bpo=2439].)

- The **pyexpat** module's **Parser** objects now allow setting their **buffer_size** attribute to change the size of the buffer used to hold character data. (Contributed by Achim Gaedke; bpo-1137 [https://bugs.python.org/issue?@action=redirect&bpo=1137].)
- The **Queue** module now provides queue variants that retrieve entries in different orders. The **PriorityQueue** class stores queued items in a heap and retrieves them in priority order, and **LifoQueue** retrieves the most recently added entries first, meaning that it behaves like a stack. (Contributed by Raymond Hettinger.)
- The **random** module's **Random** objects can now be pickled on a 32-bit system and unpickled on a 64-bit system, and vice versa. Unfortunately, this change also means that Python 2.6's **Random** objects can't be unpickled correctly on earlier versions of Python. (Contributed by Shawn Ligocki; bpo-1727780 [https://bugs.python.org/issue?@action=redirect&bpo=1727780].)

  The new `triangular(low, high, mode)` function returns random numbers following a triangular distribution. The returned values are between *low* and *high*, not including *high* itself, and with *mode* as the most frequently occurring value in the distribution. (Contributed by Wladmir van der Laan and Raymond Hettinger; bpo-1681432 [https://bugs.python.org/issue?@action=redirect&bpo=1681432].)

- Long regular expression searches carried out by the **`re`** module will check for signals being delivered, so time-consuming searches can now be interrupted. (Contributed by Josh Hoyt and Ralf Schmitt; bpo-846388 [https://bugs.python.org/issue?@action=redirect&bpo=846388].)

  The regular expression module is implemented by compiling bytecodes for a tiny regex-specific virtual machine. Untrusted code could create malicious strings of bytecode directly and cause crashes, so Python 2.6 includes a verifier for the regex bytecode. (Contributed by Guido van Rossum from work for Google App Engine; bpo-3487 [https://bugs.python.org/issue?@action=redirect&bpo=3487].)

- The **`rlcompleter`** module's **`Completer.complete()`** method will now ignore exceptions triggered while evaluating a name. (Fixed by Lorenz Quack; bpo-2250 [https://bugs.python.org/issue?@action=redirect&bpo=2250].)

- The **`sched`** module's **`scheduler`** instances now have a read-only **`queue`** attribute that returns the contents of the scheduler's queue, represented as a list of named tuples with the fields `(time, priority, action, argument)`. (Contributed by Raymond Hettinger; bpo-1861 [https://bugs.python.org/issue?@action=redirect&bpo=1861].)

- The **`select`** module now has wrapper functions for the Linux **`epoll()`** and BSD **`kqueue()`** system calls. **`modify()`**

method was added to the existing **`poll`** objects; `pollobj.modify(fd, eventmask)` takes a file descriptor or file object and an event mask, modifying the recorded event mask for that file. (Contributed by Christian Heimes; bpo-1657 [https://bugs.python.org/issue?@action=redirect&bpo=1657].)

- The **`shutil.copytree()`** function now has an optional *ignore* argument that takes a callable object. This callable will receive each directory path and a list of the directory's contents, and returns a list of names that will be ignored, not copied.

  The **`shutil`** module also provides an **`ignore_patterns()`** function for use with this new parameter. **`ignore_patterns()`** takes an arbitrary number of glob-style patterns and returns a callable that will ignore any files and directories that match any of these patterns. The following example copies a directory tree, but skips both `.svn` directories and Emacs backup files, which have names ending with '~':

  ```
  shutil.copytree('Doc/library', '/tmp/library',
                  ignore=shutil.ignore_patterns('*~',
  '.svn'))
  ```

  (Contributed by Tarek Ziadé; bpo-2663 [https://bugs.python.org/issue?@action=redirect&bpo=2663].)

- Integrating signal handling with GUI handling event loops like those used by Tkinter or GTk+ has long been a problem; most software ends up polling, waking up every fraction of a second to check if any GUI events have occurred. The **`signal`** module can now make this more efficient. Calling `signal.set_wakeup_fd(fd)` sets a file descriptor to be used; when a signal is received, a byte is written to that file descriptor. There's also a C-level function, **`PySignal_SetWakeupFd()`**, for setting the descriptor.

Event loops will use this by opening a pipe to create two descriptors, one for reading and one for writing. The writable descriptor will be passed to **`set_wakeup_fd()`**, and the readable descriptor will be added to the list of descriptors monitored by the event loop via **`select()`** or **`poll()`**. On receiving a signal, a byte will be written and the main event loop will be woken up, avoiding the need to poll.

(Contributed by Adam Olsen; bpo-1583 [https://bugs.python.org/issue?@action=redirect&bpo=1583].)

The **`siginterrupt()`** function is now available from Python code, and allows changing whether signals can interrupt system calls or not. (Contributed by Ralf Schmitt.)

The **`setitimer()`** and **`getitimer()`** functions have also been added (where they're available). **`setitimer()`** allows setting interval timers that will cause a signal to be delivered to the process after a specified time, measured in wall-clock time, consumed process time, or combined process+system time. (Contributed by Guilherme Polo; bpo-2240 [https://bugs.python.org/issue?@action=redirect&bpo=2240].)

- The **`smtplib`** module now supports SMTP over SSL thanks to the addition of the **`SMTP_SSL`** class. This class supports an interface identical to the existing **`SMTP`** class. (Contributed by Monty Taylor.) Both class constructors also have an optional `timeout` parameter that specifies a timeout for the initial connection attempt, measured in seconds. (Contributed by Facundo Batista.)

An implementation of the LMTP protocol (**RFC 2033** [https://datatracker.ietf.org/doc/html/rfc2033.html]) was also added to the module. LMTP is used in place of SMTP when transferring e-mail between agents that don't manage a mail queue. (LMTP implemented by Leif Hedstrom; bpo-957003 [https://bugs.python.org/issue?@action=redirect&bpo=957003].)

`SMTP.starttls()` now complies with **RFC 3207** [https://datatracker.ietf.org/doc/html/rfc3207.html] and forgets any knowledge obtained from the server not obtained from the TLS negotiation itself. (Patch contributed by Bill Fenner; bpo-829951 [https://bugs.python.org/issue?@action=redirect&bpo=829951].)

- The `socket` module now supports TIPC (https://tipc.sourceforge.net/), a high-performance non-IP-based protocol designed for use in clustered environments. TIPC addresses are 4- or 5-tuples. (Contributed by Alberto Bertogli; bpo-1646 [https://bugs.python.org/issue?@action=redirect&bpo=1646].)

  A new function, `create_connection()`, takes an address and connects to it using an optional timeout value, returning the connected socket object. This function also looks up the address's type and connects to it using IPv4 or IPv6 as appropriate. Changing your code to use `create_connection()` instead of `socket(socket.AF_INET, ...)` may be all that's required to make your code work with IPv6.

- The base classes in the `SocketServer` module now support calling a `handle_timeout()` method after a span of inactivity specified by the server's `timeout` attribute. (Contributed by Michael Pomraning.) The `serve_forever()` method now takes an optional poll interval measured in seconds, controlling how often the server will check for a shutdown request. (Contributed by Pedro Werneck and Jeffrey Yasskin; bpo-742598 [https://bugs.python.org/issue?@action=redirect&bpo=742598], bpo-1193577 [https://bugs.python.org/issue?@action=redirect&bpo=1193577].)

- The **`sqlite3`** module, maintained by Gerhard Häring, has been updated from version 2.3.2 in Python 2.5 to version 2.4.1.

- The **`struct`** module now supports the C99 _Bool type, using the format character `'?'`. (Contributed by David Remahl.)

- The **`Popen`** objects provided by the **`subprocess`** module now have **`terminate()`**, **`kill()`**, and **`send_signal()`** methods. On Windows, **`send_signal()`** only supports the **`SIGTERM`** signal, and all these methods are aliases for the Win32 API function **`TerminateProcess()`**. (Contributed by Christian Heimes.)

- A new variable in the **`sys`** module, **`float_info`**, is an object containing information derived from the `float.h` file about the platform’s floating-point support. Attributes of this object include **`mant_dig`** (number of digits in the mantissa), **`epsilon`** (smallest difference between 1.0 and the next largest value representable), and several others. (Contributed by Christian Heimes; bpo-1534 [https://bugs.python.org/issue?@action=redirect&bpo=1534].)

  Another new variable, **`dont_write_bytecode`**, controls whether Python writes any `.pyc` or `.pyo` files on importing a module. If this variable is true, the compiled files are not written. The variable is initially set on start-up by supplying the **`-B`** switch to the Python interpreter, or by setting the **`PYTHONDONTWRITEBYTECODE`** environment variable before running the interpreter. Python code can subsequently change the value of this variable to control whether bytecode files are written or not. (Contributed by Neal Norwitz and Georg Brandl.)

Information about the command-line arguments supplied to the Python interpreter is available by reading attributes of a named tuple available as `sys.flags`. For example, the **`verbose`** attribute is true if Python was executed in verbose mode, **`debug`** is true in debugging mode, etc. These attributes are all read-only. (Contributed by Christian Heimes.)

A new function, **`getsizeof()`**, takes a Python object and returns the amount of memory used by the object, measured in bytes. Built-in objects return correct results; third-party extensions may not, but can define a **`__sizeof__()`** method to return the object's size. (Contributed by Robert Schuppenies; bpo-2898 [https://bugs.python.org/issue?@action=redirect&bpo=2898].)

It's now possible to determine the current profiler and tracer functions by calling **`sys.getprofile()`** and **`sys.gettrace()`**. (Contributed by Georg Brandl; bpo-1648 [https://bugs.python.org/issue?@action=redirect&bpo=1648].)

- The **`tarfile`** module now supports POSIX.1-2001 (pax) tarfiles in addition to the POSIX.1-1988 (ustar) and GNU tar formats that were already supported. The default format is GNU tar; specify the `format` parameter to open a file using a different format:

  ```
  tar = tarfile.open("output.tar", "w",
                     format=tarfile.PAX_FORMAT)
  ```

  The new `encoding` and `errors` parameters specify an encoding and an error handling scheme for character conversions. `'strict'`, `'ignore'`, and `'replace'` are the three standard ways Python can handle errors,; `'utf-8'` is a special value that replaces bad characters with their UTF-8 representation. (Character conversions occur because the PAX format supports Unicode filenames, defaulting to UTF-8 encoding.)

The **`TarFile.add()`** method now accepts an `exclude` argument that's a function that can be used to exclude certain filenames from an archive. The function must take a filename and return true if the file should be excluded or false if it should be archived. The function is applied to both the name initially passed to **`add()`** and to the names of files in recursively added directories.

(All changes contributed by Lars Gustäbel).

- An optional `timeout` parameter was added to the **`telnetlib.Telnet`** class constructor, specifying a timeout measured in seconds. (Added by Facundo Batista.)

- The **`tempfile.NamedTemporaryFile`** class usually deletes the temporary file it created when the file is closed. This behaviour can now be changed by passing `delete=False` to the constructor. (Contributed by Damien Miller; bpo-1537850 [https://bugs.python.org/issue?@action=redirect&bpo=1537850].)

  A new class, **`SpooledTemporaryFile`**, behaves like a temporary file but stores its data in memory until a maximum size is exceeded. On reaching that limit, the contents will be written to an on-disk temporary file. (Contributed by Dustin J. Mitchell.)

  The **`NamedTemporaryFile`** and **`SpooledTemporaryFile`** classes both work as context managers, so you can write `with tempfile.NamedTemporaryFile() as tmp: ...`. (Contributed by

Alexander Belopolsky; bpo-2021 [https://bugs.python.org/issue?@action=redirect&bpo=2021].)

- The `test.test_support` module gained a number of context managers useful for writing tests. `EnvironmentVarGuard()` is a context manager that temporarily changes environment variables and automatically restores them to their old values.

  Another context manager, **`TransientResource`**, can surround calls to resources that may or may not be available; it will catch and ignore a specified list of exceptions. For example, a network test may ignore certain failures when connecting to an external web site:

  ```
  with test_support.TransientResource(IOError,
                                  errno=errno.ETIMEDOUT):
      f = urllib.urlopen('https://sf.net')
      ...
  ```

  Finally, **`check_warnings()`** resets the **`warning`** module's warning filters and returns an object that will record all warning messages triggered (bpo-3781 [https://bugs.python.org/issue?@action=redirect&bpo=3781]):

  ```
  with test_support.check_warnings() as wrec:
      warnings.simplefilter("always")
      # ... code that triggers a warning ...
      assert str(wrec.message) == "function is outdated"
      assert len(wrec.warnings) == 1, "Multiple warnings
  raised"
  ```

  (Contributed by Brett Cannon.)

- The `textwrap` module can now preserve existing whitespace at the beginnings and ends of the newly created lines by specifying `drop_whitespace=False` as an argument:

```
>>> S = """This  sentence  has a bunch   of
...   extra   whitespace."""
>>> print textwrap.fill(S, width=15)
This  sentence
has a bunch
of    extra
whitespace.
>>> print textwrap.fill(S, drop_whitespace=False, width=15)
This  sentence
  has a bunch
   of    extra
   whitespace.
>>>
```

(Contributed by Dwayne Bailey; bpo-1581073 [https://bugs.python.org/issue?@action=redirect&bpo=1581073].)

- The `threading` module API is being changed to use properties such as **`daemon`** instead of **`setDaemon()`** and **`isDaemon()`** methods, and some methods have been renamed to use underscores instead of camel-case; for example, the **`activeCount()`** method is renamed to **`active_count()`**. Both the 2.6 and 3.0 versions of the module support the same properties and renamed methods, but don't remove the old methods. No date has been set for the deprecation of the old APIs in Python 3.x; the old APIs won't be removed in any 2.x version. (Carried out by several people, most notably Benjamin Peterson.)

  The `threading` module's **`Thread`** objects gained an **`ident`** property that returns the thread's identifier, a nonzero integer. (Contributed by Gregory P. Smith; bpo-2871 [https://bugs.python.org/issue?@action=redirect&bpo=2871].)

- The `timeit` module now accepts callables as well as strings for the statement being timed and for the setup code. Two convenience functions were added for creating **`Timer`** instances: `repeat(stmt, setup, time, repeat, number)` and `timeit(stmt, setup, time, number)` create an instance and call the corresponding method. (Contributed by Erik Demaine; bpo-1533909 [https://bugs.python.org/issue?@action=redirect&bpo=1533909].)

- The **`Tkinter`** module now accepts lists and tuples for options, separating the elements by spaces before passing the resulting value to Tcl/Tk. (Contributed by Guilherme Polo; bpo-2906 [https://bugs.python.org/issue?@action=redirect&bpo=2906].)
- The **`turtle`** module for turtle graphics was greatly enhanced by Gregor Lingl. New features in the module include:
  - Better animation of turtle movement and rotation.
  - Control over turtle movement using the new **`delay()`**, **`tracer()`**, and **`speed()`** methods.
  - The ability to set new shapes for the turtle, and to define a new coordinate system.
  - Turtles now have an **`undo()`** method that can roll back actions.
  - Simple support for reacting to input events such as mouse and keyboard activity, making it possible to write simple games.
  - A `turtle.cfg` file can be used to customize the starting appearance of the turtle’s screen.
  - The module’s docstrings can be replaced by new docstrings that have been translated into another language.

  (bpo-1513695 [https://bugs.python.org/issue?@action=redirect&bpo=1513695])

- An optional `timeout` parameter was added to the **`urllib.urlopen`** function and the **`urllib.ftpwrapper`** class constructor, as well as the **`urllib2.urlopen`** function. The parameter specifies a timeout measured in seconds. For example:

  ```
  >>> u = urllib2.urlopen("http://slow.example.com",
                          timeout=3)
  Traceback (most recent call last):
    ...
  urllib2.URLError: <urlopen error timed out>
  >>>
  ```

  (Added by Facundo Batista.)

- The Unicode database provided by the **`unicodedata`** module has been updated to version 5.1.0. (Updated by Martin von Löwis; bpo-3811 [https://bugs.python.org/issue?@action=redirect&bpo=3811].)

- The **`warnings`** module's **`formatwarning()`** and **`showwarning()`** gained an optional *line* argument that can be used to supply the line of source code. (Added as part of bpo-1631171 [https://bugs.python.org/issue?@action=redirect&bpo=1631171], which re-implemented part of the **`warnings`** module in C code.)

  A new function, **`catch_warnings()`**, is a context manager intended for testing purposes that lets you temporarily modify the warning filters and then restore their original values (bpo-3781 [https://bugs.python.org/issue?@action=redirect&bpo=3781]).

- The XML-RPC **`SimpleXMLRPCServer`** and **`DocXMLRPCServer`** classes can now be prevented from immediately opening and binding to their socket by passing `False` as the *bind_and_activate* constructor parameter. This can be used to modify the instance's **`allow_reuse_address`** attribute before calling the **`server_bind()`** and **`server_activate()`** methods to open the socket and begin listening for connections. (Contributed by Peter Parente; bpo-1599845 [https://bugs.python.org/issue?@action=redirect&bpo=1599845].)

`SimpleXMLRPCServer` also has a **`_send_traceback_header`** attribute; if true, the exception and formatted traceback are returned as HTTP headers "X-Exception" and "X-Traceback". This feature is for debugging purposes only and should not be used on production servers because the tracebacks might reveal passwords or other sensitive information. (Contributed by Alan McIntyre as part of his project for Google's Summer of Code 2007.)

- The **`xmlrpclib`** module no longer automatically converts **`datetime.date`** and **`datetime.time`** to the **`xmlrpclib.DateTime`** type; the conversion semantics were not necessarily correct for all applications. Code using **`xmlrpclib`** should convert **`date`** and **`time`** instances. (bpo-1330538 [https://bugs.python.org/issue?@action=redirect&bpo=1330538]) The code can also handle dates before 1900 (contributed by Ralf Schmitt; bpo-2014 [https://bugs.python.org/issue?@action=redirect&bpo=2014]) and 64-bit integers represented by using `<i8>` in XML-RPC responses (contributed by Riku Lindblad; bpo-2985 [https://bugs.python.org/issue?@action=redirect&bpo=2985]).

- The **`zipfile`** module's **`ZipFile`** class now has **`extract()`** and **`extractall()`** methods that will unpack a single file or all the files in the archive to the current directory, or to a specified directory:

```
z = zipfile.ZipFile('python-251.zip')

# Unpack a single file, writing it relative
# to the /tmp directory.
```

```
z.extract('Python/sysmodule.c', '/tmp')

# Unpack all the files in the archive.
z.extractall()
```

(Contributed by Alan McIntyre; bpo-467924 [https://bugs.python.org/issue?@action=redirect&bpo=467924].)

The **open()**, **read()** and **extract()** methods can now take either a filename or a **ZipInfo** object. This is useful when an archive accidentally contains a duplicated filename. (Contributed by Graham Horler; bpo-1775025 [https://bugs.python.org/issue?@action=redirect&bpo=1775025].)

Finally, **zipfile** now supports using Unicode filenames for archived files. (Contributed by Alexey Borzenkov; bpo-1734346 [https://bugs.python.org/issue?@action=redirect&bpo=1734346].)

The ast module

The **ast** module provides an Abstract Syntax Tree representation of Python code, and Armin Ronacher contributed a set of helper functions that perform a variety of common tasks. These will be useful for HTML templating packages, code analyzers, and similar tools that process Python code.

The **parse()** function takes an expression and returns an AST. The **dump()** function outputs a representation of a tree, suitable for debugging:

```
import ast

t = ast.parse("""
d = {}
for i in 'abcdefghijklm':
    d[i + i] = ord(i) - ord('a') + 1
print d
""")
print ast.dump(t)
```

This outputs a deeply nested tree:

```
Module(body=[
  Assign(targets=[
    Name(id='d', ctx=Store())
   ], value=Dict(keys=[], values=[]))
  For(target=Name(id='i', ctx=Store()),
      iter=Str(s='abcdefghijklm'), body=[
    Assign(targets=[
      Subscript(value=
        Name(id='d', ctx=Load()),
          slice=
          Index(value=
            BinOp(left=Name(id='i', ctx=Load()), op=Add(),
             right=Name(id='i', ctx=Load()))), ctx=Store())
     ], value=
     BinOp(left=
      BinOp(left=
       Call(func=
        Name(id='ord', ctx=Load()), args=[
          Name(id='i', ctx=Load())
         ], keywords=[], starargs=None, kwargs=None),
       op=Sub(), right=Call(func=
        Name(id='ord', ctx=Load()), args=[
          Str(s='a')
         ], keywords=[], starargs=None, kwargs=None)),
       op=Add(), right=Num(n=1)))
    ], orelse=[])
   Print(dest=None, values=[
     Name(id='d', ctx=Load())
   ], nl=True)
 ])
```

The **`literal_eval()`** method takes a string or an AST representing a literal expression, parses and evaluates it, and returns the resulting value. A literal expression is a Python expression containing only strings, numbers, dictionaries, etc. but no statements or function calls. If you need to evaluate an expression but cannot accept the security risk of using an `eval()` call, **`literal_eval()`** will handle it safely:

```
>>> literal = '("a", "b", {2:4, 3:8, 1:2})'
>>> print ast.literal_eval(literal)
('a', 'b', {1: 2, 2: 4, 3: 8})
>>> print ast.literal_eval('"a" + "b"')
Traceback (most recent call last):
  ...
ValueError: malformed string
```

The module also includes **`NodeVisitor`** and **`NodeTransformer`** classes for traversing and modifying an AST, and functions for common transformations such as changing line numbers.

The `future_builtins` module

Python 3.0 makes many changes to the repertoire of built-in functions, and most of the changes can't be introduced in the Python 2.x series because they would break compatibility. The **`future_builtins`** module provides versions of these built-in functions that can be imported when writing 3.0-compatible code.

The functions in this module currently include:

- `ascii(obj)`: equivalent to `repr()`. In Python 3.0, `repr()` will return a Unicode string, while `ascii()` will return a pure ASCII bytestring.
- `filter(predicate, iterable)`, `map(func, iterable1, ...)`: the 3.0 versions return iterators, unlike the 2.x builtins which return lists.
- `hex(value)`, `oct(value)`: instead of calling the **`__hex__()`** or **`__oct__()`** methods, these versions will call the **`__index__()`**

method and convert the result to hexadecimal or octal. `oct()` will use the new `0o` notation for its result.

The `json` module: JavaScript Object Notation

The new `json` module supports the encoding and decoding of Python types in JSON (Javascript Object Notation). JSON is a lightweight interchange format often used in web applications. For more information about JSON, see http://www.json.org.

`json` comes with support for decoding and encoding most built-in Python types. The following example encodes and decodes a dictionary:

```
>>> import json
>>> data = {"spam": "foo", "parrot": 42}
>>> in_json = json.dumps(data) # Encode the data
>>> in_json
'{"parrot": 42, "spam": "foo"}'
>>> json.loads(in_json) # Decode into a Python object
{"spam": "foo", "parrot": 42}
```

It’s also possible to write your own decoders and encoders to support more types. Pretty-printing of the JSON strings is also supported.

`json` (originally called simplejson) was written by Bob Ippolito.

The `plistlib` module: A Property-List Parser

The `.plist` format is commonly used on Mac OS X to store basic data types (numbers, strings, lists, and dictionaries) by serializing them into an XML-based format. It resembles the XML-RPC serialization of data types.

Despite being primarily used on Mac OS X, the format has nothing Mac-specific about it and the Python implementation works on any platform that Python supports, so the **`plistlib`** module has been promoted to the standard library.

Using the module is simple:

```
import sys
import plistlib
import datetime

# Create data structure
data_struct = dict(lastAccessed=datetime.datetime.now(),
                   version=1,
                   categories=('Personal','Shared','Private'))

# Create string containing XML.
plist_str = plistlib.writePlistToString(data_struct)
new_struct = plistlib.readPlistFromString(plist_str)
print data_struct
print new_struct

# Write data structure to a file and read it back.
plistlib.writePlist(data_struct, '/tmp/customizations.plist')
new_struct = plistlib.readPlist('/tmp/customizations.plist')

# read/writePlist accepts file-like objects as well as paths.
plistlib.writePlist(data_struct, sys.stdout)
```

ctypes Enhancements

Thomas Heller continued to maintain and enhance the **`ctypes`** module.

`ctypes` now supports a **`c_bool`** datatype that represents the C99 `bool` type. (Contributed by David Remahl; bpo-1649190 [https://bugs.python.org/issue?@action=redirect&bpo=1649190].)

The `ctypes` string, buffer and array types have improved support for extended slicing syntax, where various combinations of `(start, stop, step)` are supplied. (Implemented by Thomas Wouters.)

All `ctypes` data types now support **`from_buffer()`** and **`from_buffer_copy()`** methods that create a ctypes instance based on a provided buffer object. **`from_buffer_copy()`** copies the contents of the object, while **`from_buffer()`** will share the same memory area.

A new calling convention tells `ctypes` to clear the `errno` or Win32 LastError variables at the outset of each wrapped call. (Implemented by Thomas Heller; bpo-1798 [https://bugs.python.org/issue?@action=redirect&bpo=1798].)

You can now retrieve the Unix `errno` variable after a function call. When creating a wrapped function, you can supply `use_errno=True` as a keyword parameter to the **`DLL()`** function and then call the module-level methods **`set_errno()`** and **`get_errno()`** to set and retrieve the error value.

The Win32 LastError variable is similarly supported by the **`DLL()`**, **`OleDLL()`**, and **`WinDLL()`** functions. You supply `use_last_error=True` as a keyword parameter and then call the module-level methods **`set_last_error()`** and **`get_last_error()`**.

The **`byref()`** function, used to retrieve a pointer to a ctypes instance, now has an optional *offset* parameter that is a byte count that will be added to the returned pointer.

Improved SSL Support

Bill Janssen made extensive improvements to Python 2.6’s support for the Secure Sockets Layer by adding a new module, `ssl`, that’s built atop the OpenSSL [https://www.openssl.org/] library. This new module provides more control over the protocol negotiated, the X.509 certificates used, and has better support for writing SSL servers (as opposed to clients) in Python. The existing SSL support in the `socket` module hasn't been removed and continues to work, though it will be removed in Python 3.0.

To use the new module, you must first create a TCP connection in the usual way and then pass it to the `ssl.wrap_socket()` function. It's possible to specify whether a certificate is required, and to obtain certificate info by calling the `getpeercert()` method.

See also

The documentation for the `ssl` module.

Deprecations and Removals

- String exceptions have been removed. Attempting to use them raises a `TypeError`.
- Changes to the `Exception` interface as dictated by **PEP 352** [https://peps.python.org/pep-0352/] continue to be made. For 2.6, the `message` attribute is being deprecated in favor of the `args` attribute.
- (3.0-warning mode) Python 3.0 will feature a reorganized standard library that will drop many outdated modules and rename others. Python 2.6 running in 3.0-warning mode will warn about these modules when they are imported.

  The list of deprecated modules is: `audiodev`, `bgenlocations`, `buildtools`, `bundlebuilder`, `Canvas`, `compiler`, `dircache`, `dl`,

`fpformat`, **`gensuitemodule`**, **`ihooks`**, **`imageop`**, **`imgfile`**, **`linuxaudiodev`**, **`mhlib`**, **`mimetools`**, **`multifile`**, **`new`**, **`pure`**, **`statvfs`**, **`sunaudiodev`**, **`test.testall`**, and **`toaiff`**.

- The **`gopherlib`** module has been removed.
- The **`MimeWriter`** module and **`mimify`** module have been deprecated; use the **`email`** package instead.
- The **`md5`** module has been deprecated; use the **`hashlib`** module instead.
- The **`posixfile`** module has been deprecated; **`fcntl.lockf()`** provides better locking.
- The **`popen2`** module has been deprecated; use the **`subprocess`** module.
- The **`rgbimg`** module has been removed.
- The **`sets`** module has been deprecated; it’s better to use the built-in **`set`** and **`frozenset`** types.
- The **`sha`** module has been deprecated; use the **`hashlib`** module instead.

Build and C API Changes

Changes to Python’s build process and to the C API include:

- Python now must be compiled with C89 compilers (after 19 years!). This means that the Python source tree has dropped its own implementations of **`memmove()`** and **`strerror()`**, which are in the C89 standard library.
- Python 2.6 can be built with Microsoft Visual Studio 2008 (version 9.0), and this is the new default compiler.

See the `PCbuild` directory for the build files. (Implemented by Christian Heimes.)

- On Mac OS X, Python 2.6 can be compiled as a 4-way universal build. The **configure** script can take a **`--with-universal-archs=[32-bit|64-bit|all]`** switch, controlling whether the binaries are built for 32-bit architectures (x86, PowerPC), 64-bit (x86-64 and PPC-64), or both. (Contributed by Ronald Oussoren.)
- A new function added in Python 2.6.6, **`PySys_SetArgvEx()`**, sets the value of `sys.argv` and can optionally update `sys.path` to include the directory containing the script named by `sys.argv[0]` depending on the value of an *updatepath* parameter.

  This function was added to close a security hole for applications that embed Python. The old function, **`PySys_SetArgv()`**, would always update `sys.path`, and sometimes it would add the current directory. This meant that, if you ran an application embedding Python in a directory controlled by someone else, attackers could put a Trojan-horse module in the directory (say, a file named `os.py`) that your application would then import and run.

  If you maintain a C/C++ application that embeds Python, check whether you're calling **`PySys_SetArgv()`** and carefully consider whether the application should be using **`PySys_SetArgvEx()`** with *updatepath* set to false. Note that using this function will break compatibility with Python versions 2.6.5 and earlier; if you have to continue working with earlier versions, you can leave the call to **`PySys_SetArgv()`** alone and call `PyRun_SimpleString("sys.path.pop(0)\n")` afterwards to discard the first `sys.path` component.

Security issue reported as **CVE 2008-5983** [https://www.cve.org/CVERecord?id=CVE-2008-5983]; discussed in gh-50003 [https://github.com/python/cpython/issues/50003], and fixed by Antoine Pitrou.

- The BerkeleyDB module now has a C API object, available as `bsddb.db.api`. This object can be used by other C extensions that wish to use the **`bsddb`** module for their own purposes. (Contributed by Duncan Grisby.)

- The new buffer interface, previously described in the PEP 3118 section, adds **`PyObject_GetBuffer()`** and **`PyBuffer_Release()`**, as well as a few other functions.

- Python's use of the C stdio library is now thread-safe, or at least as thread-safe as the underlying library is. A long-standing potential bug occurred if one thread closed a file object while another thread was reading from or writing to the object. In 2.6 file objects have a reference count, manipulated by the **`PyFile_IncUseCount()`** and **`PyFile_DecUseCount()`** functions. File objects can't be closed unless the reference count is zero. **`PyFile_IncUseCount()`** should be called while the GIL is still held, before carrying out an I/O operation using the `FILE *` pointer, and **`PyFile_DecUseCount()`** should be called immediately after the GIL is re-acquired. (Contributed by Antoine Pitrou and Gregory P. Smith.)

- Importing modules simultaneously in two different threads no longer deadlocks; it will now raise an **`ImportError`**. A new API function, **`PyImport_ImportModuleNoBlock()`**, will look for a module in `sys.modules` first, then try to import it after acquiring an import lock. If the import lock is held by another thread, an **`ImportError`** is raised. (Contributed by Christian Heimes.)

- Several functions return information about the platform’s floating-point support. `PyFloat_GetMax()` returns the maximum representable floating-point value, and `PyFloat_GetMin()` returns the minimum positive value. `PyFloat_GetInfo()` returns an object containing more information from the `float.h` file, such as `"mant_dig"` (number of digits in the mantissa), `"epsilon"` (smallest difference between 1.0 and the next largest value representable), and several others. (Contributed by Christian Heimes; bpo-1534 [https://bugs.python.org/issue?@action=redirect&bpo=1534].)

- C functions and methods that use `PyComplex_AsCComplex()` will now accept arguments that have a **`__complex__()`** method. In particular, the functions in the `cmath` module will now accept objects with this method. This is a backport of a Python 3.0 change. (Contributed by Mark Dickinson; bpo-1675423 [https://bugs.python.org/issue?@action=redirect&bpo=1675423].)

- Python’s C API now includes two functions for case-insensitive string comparisons, `PyOS_stricmp(char*, char*)` and `PyOS_strnicmp(char*, char*, Py_ssize_t)`. (Contributed by Christian Heimes; bpo-1635 [https://bugs.python.org/issue?@action=redirect&bpo=1635].)

- Many C extensions define their own little macro for adding integers and strings to the module’s dictionary in the `init*` function. Python 2.6 finally defines standard macros for adding values to a module, `PyModule_AddStringMacro` and `PyModule_AddIntMacro()`. (Contributed by Christian Heimes.)

- Some macros were renamed in both 3.0 and 2.6 to make it clearer that they are macros, not functions. **`Py_Size()`** became `Py_SIZE()`, **`Py_Type()`** became `Py_TYPE()`, and **`Py_Refcnt()`** became **`Py_REFCNT()`**. The mixed-case macros are still available in Python 2.6 for backward compatibility. (bpo-1629 [https://bugs.python.org/issue?@action=redirect&bpo=1629])

- Distutils now places C extensions it builds in a different directory when running on a debug version of Python. (Contributed by Collin Winter; bpo-1530959 [https://bugs.python.org/issue?@action=redirect&bpo=1530959].)

- Several basic data types, such as integers and strings, maintain internal free lists of objects that can be re-used. The data structures for these free lists now follow a naming convention: the variable is always named `free_list`, the counter is always named `numfree`, and a macro `Py<typename>_MAXFREELIST` is always defined.

- A new Makefile target, “make patchcheck”, prepares the Python source tree for making a patch: it fixes trailing whitespace in all modified `.py` files, checks whether the documentation has been changed, and reports whether the `Misc/ACKS` and `Misc/NEWS` files have been updated. (Contributed by Brett Cannon.)

  Another new target, “make profile-opt”, compiles a Python binary using GCC’s profile-guided optimization. It compiles Python with profiling enabled, runs the test suite to obtain a set of profiling results, and then compiles using these results for optimization. (Contributed by Gregory P. Smith.)

Port-Specific Changes: Windows

- The support for Windows 95, 98, ME and NT4 has been dropped. Python 2.6 requires at least Windows 2000 SP4.

- The new default compiler on Windows is Visual Studio 2008 (version 9.0). The build directories for Visual Studio 2003 (version 7.1) and 2005 (version 8.0) were moved into the PC/ directory. The new `PCbuild` directory supports cross compilation for X64, debug builds and Profile Guided Optimization (PGO). PGO builds are roughly 10% faster than normal builds. (Contributed by Christian Heimes with help from Amaury Forgeot d'Arc and Martin von Löwis.)

- The `msvcrt` module now supports both the normal and wide char variants of the console I/O API. The `getwch()` function reads a keypress and returns a Unicode value, as does the `getwche()` function. The `putwch()` function takes a Unicode character and writes it to the console. (Contributed by Christian Heimes.)

- `os.path.expandvars()` will now expand environment variables in the form "%var%", and "~user" will be expanded into the user's home directory path. (Contributed by Josiah Carlson; bpo-957650 [https://bugs.python.org/issue?@action=redirect&bpo=957650].)

- The `socket` module's socket objects now have an `ioctl()` method that provides a limited interface to the **`WSAIoctl()`** system interface.

- The `_winreg` module now has a function, `ExpandEnvironmentStrings()`, that expands environment variable references such as `%NAME%` in an input string. The handle objects provided by this module now support the context protocol, so they can be used in `with` statements. (Contributed by Christian Heimes.)

  `_winreg` also has better support for x64 systems, exposing the `DisableReflectionKey()`, `EnableReflectionKey()`, and `QueryReflectionKey()` functions, which enable and disable registry reflection for 32-bit processes running on 64-bit systems. (bpo-1753245 [https://bugs.python.org/issue?@action=redirect&bpo=1753245])

- The `msilib` module's `Record` object gained `GetInteger()` and `GetString()` methods that return field values as an integer or a string. (Contributed by Floris Bruynooghe; bpo-2125 [https://bugs.python.org/issue?@action=redirect&bpo=2125].)

Port-Specific Changes: Mac OS X

- When compiling a framework build of Python, you can now specify the framework name to be used by providing the `--with-framework-name=` option to the **configure** script.
- The `macfs` module has been removed. This in turn required the `macostools.touched()` function to be removed because it depended on the `macfs` module. (bpo-1490190 [https://bugs.python.org/issue?@action=redirect&bpo=1490190])
- Many other Mac OS modules have been deprecated and will be removed in Python 3.0: `_builtinSuites`, `aepack`, `aetools`, `aetypes`, `applesingle`, `appletrawmain`, `appletrunner`, `argvemulator`, `Audio_mac`, `autoGIL`, `Carbon`, `cfmfile`, `CodeWarrior`, `ColorPicker`, `EasyDialogs`, `Explorer`, `Finder`, `FrameWork`, `findertools`, `ic`, `icglue`, `icopen`, `macerrors`, `MacOS`, `macfs`, `macostools`, `macresource`, `MiniAEFrame`, `Nav`, `Netscape`, `OSATerminology`, `pimp`, `PixMapWrapper`, `StdSuites`, `SystemEvents`, `Terminal`, and `terminalcommand`.

Port-Specific Changes: IRIX

A number of old IRIX-specific modules were deprecated and will be removed in Python 3.0: `al` and `AL`, `cd`, `cddb`, `cdplayer`, `CL` and **`cl`**, **`DEVICE`**, **`ERRNO`**, **`FILE`**, **`FL`** and **`fl`**, **`flp`**, **`fm`**, **`GET`**, **`GLWS`**, **`GL`** and **`gl`**, **`IN`**, **`IOCTL`**, **`jpeg`**, **`panelparser`**, **`readcd`**, **`SV`** and **`sv`**, **`torgb`**, **`videoreader`**, and **`WAIT`**.

Porting to Python 2.6

This section lists previously described changes and other bugfixes that may require changes to your code:

- Classes that aren't supposed to be hashable should set `__hash__ = None` in their definitions to indicate the fact.
- String exceptions have been removed. Attempting to use them raises a **`TypeError`**.
- The **`__init__()`** method of **`collections.deque`** now clears any existing contents of the deque before adding elements from the iterable. This change makes the behavior match `list.__init__()`.
- **`object.__init__()`** previously accepted arbitrary arguments and keyword arguments, ignoring them. In Python 2.6, this is no longer allowed and will result in a **`TypeError`**. This will affect **`__init__()`** methods that end up calling the corresponding method on **`object`** (perhaps through using **`super()`**). See bpo-1683368 [https://bugs.python.org/issue?@action=redirect&bpo=1683368] for discussion.
- The **`Decimal`** constructor now accepts leading and trailing whitespace when passed a string. Previously it would raise an **`InvalidOperation`** exception. On the other hand, the **`create_decimal()`** method of **`Context`** objects now explicitly disallows extra whitespace, raising a **`ConversionSyntax`** exception.

- Due to an implementation accident, if you passed a file path to the built-in `__import__()` function, it would actually import the specified file. This was never intended to work, however, and the implementation now explicitly checks for this case and raises an `ImportError`.

- C API: the `PyImport_Import()` and `PyImport_ImportModule()` functions now default to absolute imports, not relative imports. This will affect C extensions that import other modules.

- C API: extension data types that shouldn't be hashable should define their `tp_hash` slot to `PyObject_HashNotImplemented()`.

- The `socket` module exception `socket.error` now inherits from `IOError`. Previously it wasn't a subclass of **`StandardError`** but now it is, through `IOError`. (Implemented by Gregory P. Smith; bpo-1706815 [https://bugs.python.org/issue?@action=redirect&bpo=1706815].)

- The `xmlrpclib` module no longer automatically converts `datetime.date` and `datetime.time` to the `xmlrpclib.DateTime` type; the conversion semantics were not necessarily correct for all applications. Code using **`xmlrpclib`** should convert **`date`** and `time` instances. (bpo-1330538 [https://bugs.python.org/issue?@action=redirect&bpo=1330538])

- (3.0-warning mode) The `Exception` class now warns when accessed using slicing or index access; having `Exception` behave like a tuple is being phased out.

- (3.0-warning mode) inequality comparisons between two dictionaries or two objects that don't implement comparison methods are reported as warnings. `dict1 == dict2` still works, but `dict1 < dict2` is being phased out.

Comparisons between cells, which are an implementation detail of Python’s scoping rules, also cause warnings because such comparisons are forbidden entirely in 3.0.

For applications that embed Python:

- The **`PySys_SetArgvEx()`** function was added in Python 2.6.6, letting applications close a security hole when the existing **`PySys_SetArgv()`** function was used. Check whether you’re calling **`PySys_SetArgv()`** and carefully consider whether the application should be using **`PySys_SetArgvEx()`** with *updatepath* set to false.

Acknowledgements

The author would like to thank the following people for offering suggestions, corrections and assistance with various drafts of this article: Georg Brandl, Steve Brown, Nick Coghlan, Ralph Corderoy, Jim Jewett, Kent Johnson, Chris Lambacher, Martin Michlmayr, Antoine Pitrou, Brian Warner.

What’s New in Python 2.5

Author:: A.M. Kuchling

This article explains the new features in Python 2.5. The final release of Python 2.5 is scheduled for August 2006; **PEP 356** [https://peps.python.org/pep-0356/] describes the planned release schedule. Python 2.5 was released on September 19, 2006.

The changes in Python 2.5 are an interesting mix of language and library improvements. The library enhancements will be more important to Python’s user community, I think, because several widely useful packages were added. New modules include ElementTree for XML processing (`xml.etree`), the SQLite database module (`sqlite`), and the `ctypes` module for calling C functions.

The language changes are of middling significance. Some pleasant new features were added, but most of them aren’t features that you’ll use every day. Conditional expressions were finally added to the language using a novel syntax; see section PEP 308: Conditional Expressions. The new ‘`with`’ statement will make writing cleanup code easier (section PEP 343: The ‘with’ statement). Values can now be passed into generators (section PEP 342: New Generator Features). Imports are now visible as either absolute or relative (section PEP 328: Absolute and Relative Imports). Some corner cases of exception handling are handled better (section PEP 341: Unified try/except/finally). All these improvements are worthwhile, but they’re improvements to one specific language feature or another; none of them are broad modifications to Python’s semantics.

As well as the language and library additions, other improvements and bugfixes were made throughout the source tree. A search through the SVN change logs finds there were 353 patches applied and 458 bugs fixed between Python 2.4 and 2.5. (Both figures are likely to be underestimates.)

This article doesn’t try to be a complete specification of the new features; instead changes are briefly introduced using helpful examples. For full details, you should always refer to the documentation for Python 2.5 at https://docs.python.org. If you want to understand the complete implementation and design rationale, refer to the PEP for a particular new feature.

Comments, suggestions, and error reports for this document are welcome; please e-mail them to the author or open a bug in the Python bug tracker.

PEP 308: Conditional Expressions

For a long time, people have been requesting a way to write conditional expressions, which are expressions that return value A or value B depending on whether a Boolean value is true or false. A conditional expression lets you write a single assignment statement that has the same effect as the following:

```
if condition:
    x = true_value
else:
    x = false_value
```

There have been endless tedious discussions of syntax on both python-dev and comp.lang.python. A vote was even held that found the majority of voters wanted conditional expressions in some form, but there was no syntax that was preferred by a clear majority. Candidates included C's `cond ? true_v : false_v`, `if cond then true_v else false_v`, and 16 other variations.

Guido van Rossum eventually chose a surprising syntax:

```
x = true_value if condition else false_value
```

Evaluation is still lazy as in existing Boolean expressions, so the order of evaluation jumps around a bit. The *condition* expression in the middle is evaluated first, and the *true_value* expression is evaluated only if the condition was true. Similarly, the *false_value* expression is only evaluated when the condition is false.

This syntax may seem strange and backwards; why does the condition go in the *middle* of the expression, and not in the front as in C's `c ? x : y`? The decision was checked by applying the new syntax to the modules in the standard library and seeing how the resulting code read. In many cases where a conditional expression is used, one value seems to be the 'common case' and one value is an 'exceptional case', used only on rarer occasions when the condition isn't met. The conditional syntax makes this pattern a bit more obvious:

```
contents = ((doc + '\n') if doc else '')
```

I read the above statement as meaning "here *contents* is usually assigned a value of `doc+'\n'`; sometimes *doc* is empty, in which special case an empty string is returned." I doubt I will use conditional expressions very often where there isn't a clear common and uncommon case.

There was some discussion of whether the language should require surrounding conditional expressions with parentheses. The decision was made to *not* require parentheses in the Python language’s grammar, but as a matter of style I think you should always use them. Consider these two statements:

```
# First version -- no parens
level = 1 if logging else 0

# Second version -- with parens
level = (1 if logging else 0)
```

In the first version, I think a reader’s eye might group the statement into ‘level = 1’, ‘if logging’, ‘else 0’, and think that the condition decides whether the assignment to *level* is performed. The second version reads better, in my opinion, because it makes it clear that the assignment is always performed and the choice is being made between two values.

Another reason for including the brackets: a few odd combinations of list comprehensions and lambdas could look like incorrect conditional expressions. See **PEP 308** [https://peps.python.org/pep-0308/] for some examples. If you put parentheses around your conditional expressions, you won’t run into this case.

See also

PEP 308 [https://peps.python.org/pep-0308/] - Conditional Expressions

PEP written by Guido van Rossum and Raymond D. Hettinger; implemented by Thomas Wouters.

PEP 309: Partial Function Application

The **`functools`** module is intended to contain tools for functional-style programming.

One useful tool in this module is the **`partial()`** function. For programs written in a functional style, you’ll sometimes want to construct variants of existing functions that have some of the parameters filled in. Consider a Python function `f(a, b, c)`; you could create a new function `g(b, c)` that was equivalent to `f(1, b, c)`. This is called “partial function application”.

`partial()` takes the arguments `(function, arg1, arg2, ... kwarg1=value1, kwarg2=value2)`. The resulting object is callable, so you can just call it to invoke *function* with the filled-in arguments.

Here’s a small but realistic example:

```
import functools

def log (message, subsystem):
    "Write the contents of 'message' to the specified
subsystem."
    print '%s: %s' % (subsystem, message)
    ...

server_log = functools.partial(log, subsystem='server')
server_log('Unable to open socket')
```

Here’s another example, from a program that uses PyGTK. Here a context-sensitive pop-up menu is being constructed dynamically. The callback provided for the menu option is a partially applied version of the **`open_item()`** method, where the first argument has been provided.

```
...
class Application:
    def open_item(self, path):
       ...
    def init (self):
        open_func = functools.partial(self.open_item,
item_path)
        popup_menu.append( ("Open", open_func, 1) )
```

Another function in the `functools` module is the `update_wrapper(wrapper, wrapped)` function that helps you write well-behaved decorators. **`update_wrapper()`** copies the name, module, and docstring attribute to a wrapper function so that tracebacks inside the wrapped function are easier to understand. For example, you might write:

```
def my_decorator(f):
    def wrapper(*args, **kwds):
        print 'Calling decorated function'
        return f(*args, **kwds)
    functools.update_wrapper(wrapper, f)
    return wrapper
```

`wraps()` is a decorator that can be used inside your own decorators to copy the wrapped function’s information. An alternate version of the previous example would be:

```
def my_decorator(f):
    @functools.wraps(f)
    def wrapper(*args, **kwds):
        print 'Calling decorated function'
        return f(*args, **kwds)
    return wrapper
```

See also

PEP 309 [https://peps.python.org/pep-0309/] - Partial Function Application

PEP proposed and written by Peter Harris; implemented by Hye-Shik Chang and Nick Coghlan, with adaptations by Raymond Hettinger.

PEP 314: Metadata for Python Software Packages v1.1

Some simple dependency support was added to Distutils. The **`setup()`** function now has `requires`, `provides`, and `obsoletes` keyword parameters. When you build a source distribution using the `sdist` command, the dependency information will be recorded in the `PKG-INFO` file.

Another new keyword parameter is `download_url`, which should be set to a URL for the package’s source code. This means it’s now possible to look up an entry in the package index, determine the dependencies for a package, and download the required packages.

```
VERSION = '1.0'
setup(name='PyPackage',
      version=VERSION,
      requires=['numarray', 'zlib (>=1.1.4)'],
      obsoletes=['OldPackage']
      download_url=('http://www.example.com/pypackage/dist/pkg-
%s.tar.gz'
                    % VERSION),
     )
```

Another new enhancement to the Python package index at https://pypi.org is storing source and binary archives for a package. The new **upload** Distutils command will upload a package to the repository.

Before a package can be uploaded, you must be able to build a distribution using the **sdist** Distutils command. Once that works, you can run `python setup.py upload` to add your package to the PyPI archive. Optionally you can GPG-sign the package by supplying the **`--sign`** and **`--identity`** options.

Package uploading was implemented by Martin von Löwis and Richard Jones.

See also

PEP 314 [https://peps.python.org/pep-0314/] - Metadata for Python Software Packages v1.1

PEP proposed and written by A.M. Kuchling, Richard Jones, and Fred Drake; implemented by Richard Jones and Fred Drake.

PEP 328: Absolute and Relative Imports

The simpler part of **PEP 328** [https://peps.python.org/pep-0328/] was implemented in Python 2.4: parentheses could now be used to enclose the names imported from a module using the `from ... import ...` statement, making it easier to import many different names.

The more complicated part has been implemented in Python 2.5: importing a module can be specified to use absolute or package-relative imports. The plan is to move toward making absolute imports the default in future versions of Python.

Let’s say you have a package directory like this:

```
pkg/
pkg/__init__.py
pkg/main.py
pkg/string.py
```

This defines a package named **`pkg`** containing the **`pkg.main`** and **`pkg.string`** submodules.

Consider the code in the `main.py` module. What happens if it executes the statement `import string`? In Python 2.4 and earlier, it will first look in the package's directory to perform a relative import, finds `pkg/string.py`, imports the contents of that file as the **`pkg.string`** module, and that module is bound to the name `string` in the **`pkg.main`** module's namespace.

That's fine if **`pkg.string`** was what you wanted. But what if you wanted Python's standard **`string`** module? There's no clean way to ignore **`pkg.string`** and look for the standard module; generally you had to look at the contents of `sys.modules`, which is slightly unclean. Holger Krekel's **`py.std`** package provides a tidier way to perform imports from the standard library, `import py; py.std.string.join()`, but that package isn't available on all Python installations.

Reading code which relies on relative imports is also less clear, because a reader may be confused about which module, **`string`** or **`pkg.string`**, is intended to be used. Python users soon learned not to duplicate the names of standard library modules in the names of their packages' submodules, but you can't protect against having your submodule's name being used for a new module added in a future version of Python.

In Python 2.5, you can switch **`import`**'s behaviour to absolute imports using a `from __future__ import absolute_import` directive. This absolute-import behaviour will become the default in a future version (probably Python 2.7). Once absolute imports are the default, `import string` will always find the standard library's version. It's suggested that users should begin using absolute imports as much as possible, so it's preferable to begin writing `from pkg import string` in your code.

Relative imports are still possible by adding a leading period to the module name when using the `from ... import` form:

```
# Import names from pkg.string
from .string import name1, name2
# Import pkg.string
from . import string
```

This imports the [`string`] module relative to the current package, so in **`pkg.main`** this will import *name1* and *name2* from **`pkg.string`**. Additional leading periods perform the relative import starting from the parent of the current package. For example, code in the **`A.B.C`** module can do:

```
from . import D                 # Imports A.B.D
from .. import E                # Imports A.E
from ..F import G               # Imports A.F.G
```

Leading periods cannot be used with the `import modname` form of the import statement, only the `from ... import` form.

See also

PEP 328 [https://peps.python.org/pep-0328/] - Imports: Multi-Line and Absolute/Relative

PEP written by Aahz; implemented by Thomas Wouters.

https://pylib.readthedocs.io/

The py library by Holger Krekel, which contains the **`py.std`** package.

PEP 338: Executing Modules as Scripts

The `-m` switch added in Python 2.4 to execute a module as a script gained a few more abilities. Instead of being implemented in C code inside the Python interpreter, the switch now uses an implementation in a new module, `runpy`.

The `runpy` module implements a more sophisticated import mechanism so that it's now possible to run modules in a package such as **`pychecker.checker`**. The module also supports alternative import mechanisms such as the `zipimport` module. This means you can add a .zip archive's path to `sys.path` and then use the `-m` switch to execute code from the archive.

See also

PEP 338 [https://peps.python.org/pep-0338/] - Executing modules as scripts
PEP written and implemented by Nick Coghlan.

PEP 341: Unified try/except/finally

Until Python 2.5, the `try` statement came in two flavours. You could use a `finally` block to ensure that code is always executed, or one or more `except` blocks to catch specific exceptions. You couldn't combine both **`except`** blocks and a **`finally`** block, because generating the right bytecode for the combined version was complicated and it wasn't clear what the semantics of the combined statement should be.

Guido van Rossum spent some time working with Java, which does support the equivalent of combining `except` blocks and a `finally` block, and this clarified what the statement should mean. In Python 2.5, you can now write:

```
try:
    block-1 ...
except Exception1:
    handler-1 ...
except Exception2:
    handler-2 ...
else:
    else-block
finally:
    final-block
```

The code in *block-1* is executed. If the code raises an exception, the various `except` blocks are tested: if the exception is of class **`Exception1`**, *handler-1* is executed; otherwise if it’s of class **`Exception2`**, *handler-2* is executed, and so forth. If no exception is raised, the *else-block* is executed.

No matter what happened previously, the *final-block* is executed once the code block is complete and any raised exceptions handled. Even if there’s an error in an exception handler or the *else-block* and a new exception is raised, the code in the *final-block* is still run.

See also

PEP 341 [https://peps.python.org/pep-0341/] - Unifying try-except and try-finally
PEP written by Georg Brandl; implementation by Thomas Lee.

PEP 342: New Generator Features

Python 2.5 adds a simple way to pass values *into* a generator. As introduced in Python 2.3, generators only produce output; once a generator’s code was invoked to create an iterator, there was no way to pass any new information into the function when its execution is resumed. Sometimes the ability to pass in some information would be useful. Hackish solutions to this include making the generator's code look at a global variable and then changing the global variable's value, or passing in some mutable object that callers then modify.

To refresh your memory of basic generators, here's a simple example:

```
def counter (maximum):
    i = 0
    while i < maximum:
        yield i
        i += 1
```

When you call `counter(10)`, the result is an iterator that returns the values from 0 up to 9. On encountering the **`yield`** statement, the iterator returns the provided value and suspends the function's execution, preserving the local variables. Execution resumes on the following call to the iterator's **`next()`** method, picking up after the **`yield`** statement.

In Python 2.3, **`yield`** was a statement; it didn't return any value. In 2.5, **`yield`** is now an expression, returning a value that can be assigned to a variable or otherwise operated on:

```
val = (yield i)
```

I recommend that you always put parentheses around a **`yield`** expression when you're doing something with the returned value, as in the above example. The parentheses aren't always necessary, but it's easier to always add them instead of having to remember when they're needed.

(**PEP 342** [https://peps.python.org/pep-0342/] explains the exact rules, which are that a **`yield`**-expression must always be parenthesized except when it occurs at the top-level expression on the right-hand side of an assignment. This means you can write `val = yield i` but have to use parentheses when there's an operation, as in `val = (yield i) + 12`.)

Values are sent into a generator by calling its `send(value)` method. The generator's code is then resumed and the **`yield`** expression returns the specified *value*. If the regular **`next()`** method is called, the **`yield`** returns **`None`**.

Here's the previous example, modified to allow changing the value of the internal counter.

```
def counter (maximum):
    i = 0
    while i < maximum:
        val = (yield i)
        # If value provided, change counter
        if val is not None:
            i = val
        else:
            i += 1
```

And here's an example of changing the counter:

```
>>> it = counter(10)
>>> print it.next()
0
>>> print it.next()
1
>>> print it.send(8)
8
>>> print it.next()
9
>>> print it.next()
Traceback (most recent call last):
  File "t.py", line 15, in ?
    print it.next()
StopIteration
```

`yield` will usually return `None`, so you should always check for this case. Don't just use its value in expressions unless you're sure that the **`send()`** method will be the only method used to resume your generator function.

In addition to **`send()`**, there are two other new methods on generators:

- `throw(type, value=None, traceback=None)` is used to raise an exception inside the generator; the exception is raised by the `yield` expression where the generator's execution is paused.

- **`close()`** raises a new `GeneratorExit` exception inside the generator to terminate the iteration. On receiving this exception, the generator's code must either raise `GeneratorExit` or `StopIteration`. Catching the `GeneratorExit` exception and returning a value is illegal and will trigger a `RuntimeError`; if the function raises some other exception, that exception is propagated to the caller. **`close()`** will also be called by Python's garbage collector when the generator is garbage-collected.

  If you need to run cleanup code when a `GeneratorExit` occurs, I suggest using a `try: ... finally:` suite instead of catching `GeneratorExit`.

The cumulative effect of these changes is to turn generators from one-way producers of information into both producers and consumers.

Generators also become *coroutines*, a more generalized form of subroutines. Subroutines are entered at one point and exited at another point (the top of the function, and a `return` statement), but coroutines can be entered, exited, and resumed at many different points (the `yield` statements). We’ll have to figure out patterns for using coroutines effectively in Python.

The addition of the **`close()`** method has one side effect that isn’t obvious. **`close()`** is called when a generator is garbage-collected, so this means the generator’s code gets one last chance to run before the generator is destroyed. This last chance means that `try...finally` statements in generators can now be guaranteed to work; the **`finally`** clause will now always get a chance to run. The syntactic restriction that you couldn’t mix **`yield`** statements with a `try...finally` suite has therefore been removed. This seems like a minor bit of language trivia, but using generators and `try...finally` is actually necessary in order to implement the **`with`** statement described by **PEP 343** [https://peps.python.org/pep-0343/]. I’ll look at this new statement in the following section.

Another even more esoteric effect of this change: previously, the **`gi_frame`** attribute of a generator was always a frame object. It’s now possible for **`gi_frame`** to be `None` once the generator has been exhausted.

See also

PEP 342 [https://peps.python.org/pep-0342/] - Coroutines via Enhanced Generators

PEP written by Guido van Rossum and Phillip J. Eby; implemented by Phillip J. Eby. Includes examples of some fancier uses of generators as coroutines.

Earlier versions of these features were proposed in **PEP 288** [https://peps.python.org/pep-0288/] by Raymond Hettinger and **PEP 325** [https://peps.python.org/pep-0325/] by Samuele Pedroni.

https://en.wikipedia.org/wiki/Coroutine

The Wikipedia entry for coroutines.

https://web.archive.org/web/20160321211320/http://www.sidhe.org/~dan/blog/archives/000178.html

An explanation of coroutines from a Perl point of view, written by Dan Sugalski.

PEP 343: The ‘with’ statement

The ‘`with`’ statement clarifies code that previously would use `try...finally` blocks to ensure that clean-up code is executed. In this section, I’ll discuss the statement as it will commonly be used. In the next section, I’ll examine the implementation details and show how to write objects for use with this statement.

The ‘`with`’ statement is a new control-flow structure whose basic structure is:

```
with expression [as variable]:
    with-block
```

The expression is evaluated, and it should result in an object that supports the context management protocol (that is, has `__enter__()` and `__exit__()` methods.

The object’s `__enter__()` is called before *with-block* is executed and therefore can run set-up code. It also may return a value that is bound to the name *variable*, if given. (Note carefully that *variable* is *not* assigned the result of *expression*.)

After execution of the *with-block* is finished, the object’s `__exit__()` method is called, even if the block raised an exception, and can therefore run clean-up code.

To enable the statement in Python 2.5, you need to add the following directive to your module:

```
from __future__ import with_statement
```

The statement will always be enabled in Python 2.6.

Some standard Python objects now support the context management protocol and can be used with the ‘`with`’ statement. File objects are one example:

```
with open('/etc/passwd', 'r') as f:
    for line in f:
        print line
        ... more processing code ...
```

After this statement has executed, the file object in *f* will have been automatically closed, even if the `for` loop raised an exception part-way through the block.

Note

In this case, *f* is the same object created by `open()`, because `__enter__()` returns *self*.

The `threading` module’s locks and condition variables also support the ‘`with`’ statement:

```
lock = threading.Lock()
with lock:
    # Critical section of code
    ...
```

The lock is acquired before the block is executed and always released once the block is complete.

The new **`localcontext()`** function in the `decimal` module makes it easy to save and restore the current decimal context, which encapsulates the desired precision and rounding characteristics for computations:

```
from decimal import Decimal, Context, localcontext

# Displays with default precision of 28 digits
v = Decimal('578')
print v.sqrt()

with localcontext(Context(prec=16)):
    # All code in this block uses a precision of 16 digits.
    # The original context is restored on exiting the block.
    print v.sqrt()
```

Writing Context Managers

Under the hood, the ‘`with`’ statement is fairly complicated. Most people will only use ‘**`with`**’ in company with existing objects and don’t need to know these details, so you can skip the rest of this section if you like. Authors of new objects will need to understand the details of the underlying implementation and should keep reading.

A high-level explanation of the context management protocol is:

- The expression is evaluated and should result in an object called a “context manager”. The context manager must have `__enter__()` and `__exit__()` methods.
- The context manager’s `__enter__()` method is called. The value returned is assigned to *VAR*. If no `'as VAR'` clause is present, the value is simply discarded.
- The code in *BLOCK* is executed.
- If *BLOCK* raises an exception, the `__exit__(type, value, traceback)` is called with the exception details, the same values returned by `sys.exc_info()`. The method’s return

value controls whether the exception is re-raised: any false value re-raises the exception, and `True` will result in suppressing it. You'll only rarely want to suppress the exception, because if you do the author of the code containing the '`with`' statement will never realize anything went wrong.

- If *BLOCK* didn't raise an exception, the `__exit__()` method is still called, but *type*, *value*, and *traceback* are all `None`.

Let's think through an example. I won't present detailed code but will only sketch the methods necessary for a database that supports transactions.

(For people unfamiliar with database terminology: a set of changes to the database are grouped into a transaction. Transactions can be either committed, meaning that all the changes are written into the database, or rolled back, meaning that the changes are all discarded and the database is unchanged. See any database textbook for more information.)

Let's assume there's an object representing a database connection. Our goal will be to let the user write code like this:

```
db_connection = DatabaseConnection()
with db_connection as cursor:
    cursor.execute('insert into ...')
    cursor.execute('delete from ...')
    # ... more operations ...
```

The transaction should be committed if the code in the block runs flawlessly or rolled back if there's an exception. Here's the basic interface for **`DatabaseConnection`** that I'll assume:

```
class DatabaseConnection:
    # Database interface
    def cursor (self):
        "Returns a cursor object and starts a new transaction"
    def commit (self):
        "Commits current transaction"
    def rollback (self):
        "Rolls back current transaction"
```

The `__enter__()` method is pretty easy, having only to start a new transaction. For this application the resulting cursor object would be a useful result, so the method will return it. The user can then add `as cursor` to their '`with`' statement to bind the cursor to a variable name.

```
class DatabaseConnection:
    ...
    def __enter__ (self):
        # Code to start a new transaction
        cursor = self.cursor()
        return cursor
```

The `__exit__()` method is the most complicated because it's where most of the work has to be done. The method has to check if an exception occurred. If there was no exception, the transaction is committed. The transaction is rolled back if there was an exception.

In the code below, execution will just fall off the end of the function, returning the default value of `None`. `None` is false, so the exception will be re-raised automatically. If you wished, you could be more explicit and add a `return` statement at the marked location.

```
class DatabaseConnection:
    ...
    def __exit__ (self, type, value, tb):
        if tb is None:
            # No exception, so commit
            self.commit()
        else:
            # Exception occurred, so rollback.
            self.rollback()
            # return False
```

The contextlib module

The new `contextlib` module provides some functions and a decorator that are useful for writing objects for use with the '`with`' statement.

The decorator is called **`contextmanager()`**, and lets you write a single generator function instead of defining a new class. The generator should yield exactly one value. The code up to the `yield` will be executed as the `__enter__()` method, and the value yielded will be the method's return value that will get bound to the variable in the '`with`' statement's **`as`** clause, if any. The code after the `yield` will be executed in the `__exit__()` method. Any exception raised in the block will be raised by the **`yield`** statement.

Our database example from the previous section could be written using this decorator as:

```
from contextlib import contextmanager

@contextmanager
def db_transaction (connection):
    cursor = connection.cursor()
    try:
        yield cursor
    except:
        connection.rollback()
        raise
    else:
        connection.commit()

db = DatabaseConnection()
```

```
with db_transaction(db) as cursor:
    ...
```

The `contextlib` module also has a `nested(mgr1, mgr2, ...)` function that combines a number of context managers so you don't need to write nested '`with`' statements. In this example, the single '**`with`**' statement both starts a database transaction and acquires a thread lock:

```
lock = threading.Lock()
with nested (db_transaction(db), lock) as (cursor, locked):
    ...
```

Finally, the `closing(object)` function returns *object* so that it can be bound to a variable, and calls `object.close` at the end of the block.

```
import urllib, sys
from contextlib import closing

with closing(urllib.urlopen('http://www.yahoo.com')) as f:
    for line in f:
        sys.stdout.write(line)
```

See also

PEP 343 [https://peps.python.org/pep-0343/] - The "with" statement

PEP written by Guido van Rossum and Nick Coghlan; implemented by Mike Bland, Guido van Rossum, and Neal Norwitz. The PEP shows the code generated for a '`with`' statement, which can be helpful in learning how the statement works.

The documentation for the `contextlib` module.

PEP 352: Exceptions as New-Style Classes

Exception classes can now be new-style classes, not just classic classes, and the built-in **`Exception`** class and all the standard built-in exceptions (**`NameError`**, **`ValueError`**, etc.) are now new-style classes.

The inheritance hierarchy for exceptions has been rearranged a bit. In 2.5, the inheritance relationships are:

```
BaseException       # New in Python 2.5
|- KeyboardInterrupt
|- SystemExit
|- Exception
   |- (all other current built-in exceptions)
```

This rearrangement was done because people often want to catch all exceptions that indicate program errors. **`KeyboardInterrupt`** and **`SystemExit`** aren't errors, though, and usually represent an explicit action such as the user hitting `Control-C` or code calling **`sys.exit()`**. A bare `except:` will catch all exceptions, so you commonly need to list **`KeyboardInterrupt`** and **`SystemExit`** in order to re-raise them. The usual pattern is:

```
try:
    ...
except (KeyboardInterrupt, SystemExit):
    raise
except:
    # Log error...
    # Continue running program...
```

In Python 2.5, you can now write `except Exception` to achieve the same result, catching all the exceptions that usually indicate errors but leaving **`KeyboardInterrupt`** and **`SystemExit`** alone. As in previous versions, a bare `except:` still catches all exceptions.

The goal for Python 3.0 is to require any class raised as an exception to derive from `BaseException` or some descendant of `BaseException`, and future releases in the Python 2.x series may begin to enforce this constraint. Therefore, I suggest you begin making all your exception classes derive from `Exception` now. It's been suggested that the bare `except:` form should be removed in Python 3.0, but Guido van Rossum hasn't decided whether to do this or not.

Raising of strings as exceptions, as in the statement `raise "Error occurred"`, is deprecated in Python 2.5 and will trigger a warning. The aim is to be able to remove the string-exception feature in a few releases.

See also

PEP 352 [https://peps.python.org/pep-0352/] - Required Superclass for Exceptions

PEP written by Brett Cannon and Guido van Rossum; implemented by Brett Cannon.

PEP 353: Using ssize_t as the index type

A wide-ranging change to Python's C API, using a new `Py_ssize_t` type definition instead of int, will permit the interpreter to handle more data on 64-bit platforms. This change doesn't affect Python's capacity on 32-bit platforms.

Various pieces of the Python interpreter used C's int type to store sizes or counts; for example, the number of items in a list or tuple were stored in an int. The C compilers for most 64-bit platforms still define int as a 32-bit type, so that meant that lists could only hold up to `2**31 - 1` = 2147483647 items. (There are actually a few different programming models that 64-bit C compilers can use – see https://unix.org/version2/whatsnew/lp64_wp.html for a discussion – but the most commonly available model leaves int as 32 bits.)

A limit of 2147483647 items doesn't really matter on a 32-bit platform because you'll run out of memory before hitting the length limit. Each list item requires space for a pointer, which is 4 bytes, plus space for a **`PyObject`** representing the item. 2147483647*4 is already more bytes than a 32-bit address space can contain.

It's possible to address that much memory on a 64-bit platform, however. The pointers for a list that size would only require 16 GiB of space, so it's not unreasonable that Python programmers might construct lists that large. Therefore, the Python interpreter had to be changed to use some type other than int, and this will be a 64-bit type on 64-bit platforms. The change will cause incompatibilities on 64-bit machines, so it was deemed worth making the transition now, while the number of 64-bit users is still relatively small. (In 5 or 10 years, we may *all* be on 64-bit machines, and the transition would be more painful then.)

This change most strongly affects authors of C extension modules. Python strings and container types such as lists and tuples now use **`Py_ssize_t`** to store their size. Functions such as **`PyList_Size()`** now return **`Py_ssize_t`**. Code in extension modules may therefore need to have some variables changed to **`Py_ssize_t`**.

The `PyArg_ParseTuple()` and `Py_BuildValue()` functions have a new conversion code, `n`, for `Py_ssize_t`. `PyArg_ParseTuple()`'s `s#` and `t#` still output int by default, but you can define the macro **`PY_SSIZE_T_CLEAN`** before including `Python.h` to make them return `Py_ssize_t`.

PEP 353 [https://peps.python.org/pep-0353/] has a section on conversion guidelines that extension authors should read to learn about supporting 64-bit platforms.

See also

PEP 353 [https://peps.python.org/pep-0353/] - Using ssize_t as the index type
PEP written and implemented by Martin von Löwis.

PEP 357: The '__index__' method

The NumPy developers had a problem that could only be solved by adding a new special method, **`__index__()`**. When using slice notation, as in `[start:stop:step]`, the values of the *start*, *stop*, and *step* indexes must all be either integers or long integers. NumPy defines a variety of specialized integer types corresponding to unsigned and signed integers of 8, 16, 32, and 64 bits, but there was no way to signal that these types could be used as slice indexes.

Slicing can't just use the existing **`__int__()`** method because that method is also used to implement coercion to integers. If slicing used **`__int__()`**, floating-point numbers would also become legal slice indexes and that's clearly an undesirable behaviour.

Instead, a new special method called **`__index__()`** was added. It takes no arguments and returns an integer giving the slice index to use. For example:

```
class C:
    def __index__ (self):
        return self.value
```

The return value must be either a Python integer or long integer. The interpreter will check that the type returned is correct, and raises a `TypeError` if this requirement isn't met.

A corresponding `nb_index` slot was added to the C-level `PyNumberMethods` structure to let C extensions implement this protocol. `PyNumber_Index(obj)` can be used in extension code to call the **`__index__()`** function and retrieve its result.

See also

PEP 357 [https://peps.python.org/pep-0357/] - Allowing Any Object to be Used for Slicing
PEP written and implemented by Travis Oliphant.

Other Language Changes

Here are all of the changes that Python 2.5 makes to the core Python language.

- The `dict` type has a new hook for letting subclasses provide a default value when a key isn't contained in the dictionary. When a key isn't found, the dictionary's `__missing__(key)` method will be called. This hook is used to implement the new **`defaultdict`** class in the `collections` module. The following example defines a dictionary that returns zero for any missing key:

```
class zerodict (dict):
    def __missing__ (self, key):
        return 0

d = zerodict({1:1, 2:2})
print d[1], d[2]   # Prints 1, 2
print d[3], d[4]   # Prints 0, 0
```

- Both 8-bit and Unicode strings have new `partition(sep)` and `rpartition(sep)` methods that simplify a common use case.

  The `find(S)` method is often used to get an index which is then used to slice the string and obtain the pieces that are before and after the separator. `partition(sep)` condenses this pattern into a single method call that returns a 3-tuple containing the substring before the separator, the separator itself, and the substring after the separator. If the separator isn’t found, the first element of the tuple is the entire string and the other two elements are empty. `rpartition(sep)` also returns a 3-tuple but starts searching from the end of the string; the `r` stands for ‘reverse’.

  Some examples:

  ```
  >>> ('http://www.python.org').partition('://')
  ('http', '://', 'www.python.org')
  >>> ('file:/usr/share/doc/index.html').partition('://')
  ('file:/usr/share/doc/index.html', '', '')
  >>> (u'Subject: a quick question').partition(':')
  (u'Subject', u':', u' a quick question')
  >>> 'www.python.org'.rpartition('.')
  ('www.python', '.', 'org')
  >>> 'www.python.org'.rpartition(':')
  ('', '', 'www.python.org')
  ```

  (Implemented by Fredrik Lundh following a suggestion by Raymond Hettinger.)

- The **`startswith()`** and **`endswith()`** methods of string types now accept tuples of strings to check for.

  ```
  def is_image_file (filename):
      return filename.endswith(('.gif', '.jpg', '.tiff'))
  ```

  (Implemented by Georg Brandl following a suggestion by Tom Lynn.)

- The **`min()`** and **`max()`** built-in functions gained a `key` keyword parameter analogous to the `key` argument for **`sort()`**. This parameter supplies a function that takes a single argument and is called for every value in the list; **`min()`**/**`max()`** will return the element with the smallest/largest return value from this function. For example, to find the longest string in a list, you can do:

  ```
  L = ['medium', 'longest', 'short']
  # Prints 'longest'
  print max(L, key=len)
  # Prints 'short', because lexicographically 'short' has the
  largest value
  print max(L)
  ```

  (Contributed by Steven Bethard and Raymond Hettinger.)

- Two new built-in functions, **`any()`** and **`all()`**, evaluate whether an iterator contains any true or false values. **`any()`** returns **`True`** if any value returned by the iterator is true; otherwise it will return **`False`**. **`all()`** returns **`True`** only if all of the values returned by the iterator evaluate as true. (Suggested by Guido van Rossum, and implemented by Raymond Hettinger.)

- The result of a class's **`__hash__()`** method can now be either a long integer or a regular integer. If a long integer is returned, the hash of that value is taken. In earlier versions the hash value was required to be a regular integer, but in 2.5 the `id()` built-in was changed to always return non-negative numbers, and users often seem to use `id(self)` in **`__hash__()`** methods (though this is discouraged).

- ASCII is now the default encoding for modules. It's now a syntax error if a module contains string literals with 8-bit characters but doesn't have an encoding declaration. In Python 2.4 this triggered a warning, not a syntax error. See **PEP 263** [https://peps.python.org/pep-0263/] for how to declare a module's encoding; for example, you might add a line like this near the top of the source file:

  ```
  # -*- coding: latin1 -*-
  ```

- A new warning, `UnicodeWarning`, is triggered when you attempt to compare a Unicode string and an 8-bit string that can't be converted to Unicode using the default ASCII encoding. The result of the comparison is false:

  ```
  >>> chr(128) == unichr(128)   # Can't convert chr(128) to Unicode
  __main__:1: UnicodeWarning: Unicode equal comparison failed
    to convert both arguments to Unicode - interpreting them
    as being unequal
  False
  >>> chr(127) == unichr(127)   # chr(127) can be converted
  True
  ```

  Previously this would raise a `UnicodeDecodeError` exception, but in 2.5 this could result in puzzling problems when accessing a dictionary. If you looked up `unichr(128)` and `chr(128)` was being used as a key, you'd get a `UnicodeDecodeError` exception. Other changes in 2.5 resulted in this exception being raised instead of suppressed by the code in `dictobject.c` that implements dictionaries.

Raising an exception for such a comparison is strictly correct, but the change might have broken code, so instead `UnicodeWarning` was introduced.

(Implemented by Marc-André Lemburg.)

- One error that Python programmers sometimes make is forgetting to include an `__init__.py` module in a package directory. Debugging this mistake can be confusing, and usually requires running Python with the `-v` switch to log all the paths searched. In Python 2.5, a new `ImportWarning` warning is triggered when an import would have picked up a directory as a package but no `__init__.py` was found. This warning is silently ignored by default; provide the `-Wd` option when running the Python executable to display the warning message. (Implemented by Thomas Wouters.)
- The list of base classes in a class definition can now be empty. As an example, this is now legal:

  ```
  class C():
      pass
  ```

  (Implemented by Brett Cannon.)

Interactive Interpreter Changes

In the interactive interpreter, `quit` and `exit` have long been strings so that new users get a somewhat helpful message when they try to quit:

```
>>> quit
'Use Ctrl-D (i.e. EOF) to exit.'
```

In Python 2.5, `quit` and `exit` are now objects that still produce string representations of themselves, but are also callable. Newbies who try `quit()` or `exit()` will now exit the interpreter as they expect. (Implemented by Georg Brandl.)

The Python executable now accepts the standard long options `--help` and `--version`; on Windows, it also accepts the `/?` option for displaying a help message. (Implemented by Georg Brandl.)

Optimizations

Several of the optimizations were developed at the NeedForSpeed sprint, an event held in Reykjavik, Iceland, from May 21–28 2006. The sprint focused on speed enhancements to the CPython implementation and was funded by EWT LLC with local support from CCP Games. Those optimizations added at this sprint are specially marked in the following list.

- When they were introduced in Python 2.4, the built-in `set` and `frozenset` types were built on top of Python’s dictionary type. In 2.5 the internal data structure has been customized for implementing sets, and as a result sets will use a third less memory and are somewhat faster. (Implemented by Raymond Hettinger.)
- The speed of some Unicode operations, such as finding substrings, string splitting, and character map encoding and decoding, has been improved. (Substring search and splitting improvements were added by Fredrik Lundh and Andrew Dalke at the NeedForSpeed sprint. Character maps were improved by Walter Dörwald and Martin von Löwis.)
- The `long(str, base)` function is now faster on long digit strings because fewer intermediate results are calculated. The peak is for strings of around 800–1000 digits where the function is 6 times faster. (Contributed

by Alan McIntyre and committed at the NeedForSpeed sprint.)

- It’s now illegal to mix iterating over a file with `for line in file` and calling the file object’s **`read()`**/**`readline()`**/**`readlines()`** methods. Iteration uses an internal buffer and the **`read*()`** methods don’t use that buffer. Instead they would return the data following the buffer, causing the data to appear out of order. Mixing iteration and these methods will now trigger a **`ValueError`** from the **`read*()`** method. (Implemented by Thomas Wouters.)
- The **`struct`** module now compiles structure format strings into an internal representation and caches this representation, yielding a 20% speedup. (Contributed by Bob Ippolito at the NeedForSpeed sprint.)
- The **`re`** module got a 1 or 2% speedup by switching to Python’s allocator functions instead of the system’s **`malloc()`** and **`free()`**. (Contributed by Jack Diederich at the NeedForSpeed sprint.)
- The code generator’s peephole optimizer now performs simple constant folding in expressions. If you write something like `a = 2+3`, the code generator will do the arithmetic and produce code corresponding to `a = 5`. (Proposed and implemented by Raymond Hettinger.)
- Function calls are now faster because code objects now keep the most recently finished frame (a “zombie frame”) in an internal field of the code object, reusing it the next time the code object is invoked. (Original patch by Michael Hudson, modified by Armin Rigo and Richard Jones; committed at the NeedForSpeed sprint.) Frame objects are also slightly smaller, which may improve cache locality and reduce memory usage a bit. (Contributed by Neal Norwitz.)
- Python’s built-in exceptions are now new-style classes, a change that speeds up instantiation considerably. Exception handling in Python 2.5 is therefore about 30%

faster than in 2.4. (Contributed by Richard Jones, Georg Brandl and Sean Reifschneider at the NeedForSpeed sprint.)

- Importing now caches the paths tried, recording whether they exist or not so that the interpreter makes fewer **`open()`** and **`stat()`** calls on startup. (Contributed by Martin von Löwis and Georg Brandl.)

New, Improved, and Removed Modules

The standard library received many enhancements and bug fixes in Python 2.5. Here’s a partial list of the most notable changes, sorted alphabetically by module name. Consult the `Misc/NEWS` file in the source tree for a more complete list of changes, or look through the SVN logs for all the details.

- The **`audioop`** module now supports the a-LAW encoding, and the code for u-LAW encoding has been improved. (Contributed by Lars Immisch.)

- The **`codecs`** module gained support for incremental codecs. The **`codec.lookup()`** function now returns a **`CodecInfo`** instance instead of a tuple. **`CodecInfo`** instances behave like a 4-tuple to preserve backward compatibility but also have the attributes **`encode`**, **`decode`**, **`incrementalencoder`**, **`incrementaldecoder`**, **`streamwriter`**, and **`streamreader`**. Incremental codecs can receive input and produce output in multiple chunks; the output is the same as if the entire input was fed to the non-incremental codec. See the **`codecs`** module documentation for details. (Designed and implemented by Walter Dörwald.)

- The `collections` module gained a new type, **`defaultdict`**, that subclasses the standard `dict` type. The new type mostly behaves like a dictionary but constructs a default value when a key isn’t present, automatically adding it to the dictionary for the requested key value.

  The first argument to **`defaultdict`**’s constructor is a factory function that gets called whenever a key is requested but not found. This factory function receives no arguments, so you can use built-in type constructors such as `list()` or `int()`. For example, you can make an index of words based on their initial letter like this:

  ```
  words = """Nel mezzo del cammin di nostra vita
  mi ritrovai per una selva oscura
  che la diritta via era smarrita""".lower().split()

  index = defaultdict(list)

  for w in words:
      init_letter = w[0]
      index[init_letter].append(w)
  ```

  Printing `index` results in the following output:

  ```
  defaultdict(<type 'list'>, {'c': ['cammin', 'che'], 'e':
  ['era'],
          'd': ['del', 'di', 'diritta'], 'm': ['mezzo',
  'mi'],
          'l': ['la'], 'o': ['oscura'], 'n': ['nel',
  'nostra'],
          'p': ['per'], 's': ['selva', 'smarrita'],
          'r': ['ritrovai'], 'u': ['una'], 'v': ['vita',
  'via']}
  ```

  (Contributed by Guido van Rossum.)

- The **`deque`** double-ended queue type supplied by the `collections` module now has a `remove(value)` method that removes the first occurrence of *value* in the queue, raising `ValueError` if the value isn't found. (Contributed by Raymond Hettinger.)

- New module: The `contextlib` module contains helper functions for use with the new '`with`' statement. See section The contextlib module for more about this module.

- New module: The `cProfile` module is a C implementation of the existing `profile` module that has much lower overhead. The module's interface is the same as `profile`: you run `cProfile.run('main()')` to profile a function, can save profile data to a file, etc. It's not yet known if the Hotshot profiler, which is also written in C but doesn't match the `profile` module's interface, will continue to be maintained in future versions of Python. (Contributed by Armin Rigo.)

  Also, the `pstats` module for analyzing the data measured by the profiler now supports directing the output to any file object by supplying a *stream* argument to the **`Stats`** constructor. (Contributed by Skip Montanaro.)

- The `csv` module, which parses files in comma-separated value format, received several enhancements and a number of bugfixes. You can now set the maximum size in bytes of a field by calling the `csv.field_size_limit(new_limit)` function; omitting the *new_limit* argument will return the currently set limit. The **`reader`** class now has a **`line_num`** attribute that counts the number of physical lines read from the source; records can span multiple physical lines, so **`line_num`** is not the same as the number of records read.

  The CSV parser is now stricter about multi-line quoted fields. Previously, if a line ended within a quoted field without a terminating newline character, a newline would be inserted into the returned field. This behavior caused problems when reading files that contained carriage return characters within fields, so the code was changed to return the field without inserting newlines. As a consequence, if newlines embedded within fields are important, the input should be split into lines in a manner that preserves the newline characters.

(Contributed by Skip Montanaro and Andrew McNamara.)

- The **`datetime`** class in the **`datetime`** module now has a `strptime(string, format)` method for parsing date strings, contributed by Josh Spoerri. It uses the same format characters as **`time.strptime()`** and **`time.strftime()`**:

```
from datetime import datetime

ts = datetime.strptime('10:13:15 2006-03-07',
                       '%H:%M:%S %Y-%m-%d')
```

- The **`SequenceMatcher.get_matching_blocks()`** method in the **`difflib`** module now guarantees to return a minimal list of blocks describing matching subsequences. Previously, the algorithm would occasionally break a block of matching elements into two list entries. (Enhancement by Tim Peters.)

- The **`doctest`** module gained a `SKIP` option that keeps an example from being executed at all. This is intended for code snippets that are usage examples intended for the reader and aren’t actually test cases.

  An *encoding* parameter was added to the **`testfile()`** function and the **`DocFileSuite`** class to specify the file’s encoding. This makes it easier to use non-ASCII characters in tests contained within a docstring. (Contributed by Bjorn Tillenius.)

- The `email` package has been updated to version 4.0. (Contributed by Barry Warsaw.)

- The `fileinput` module was made more flexible. Unicode filenames are now supported, and a *mode* parameter that defaults to `"r"` was added to the `input()` function to allow opening files in binary or universal newlines mode. Another new parameter, *openhook*, lets you use a function other than `open()` to open the input files. Once you're iterating over the set of files, the **`FileInput`** object's new `fileno()` returns the file descriptor for the currently opened file. (Contributed by Georg Brandl.)

- In the `gc` module, the new **`get_count()`** function returns a 3-tuple containing the current collection counts for the three GC generations. This is accounting information for the garbage collector; when these counts reach a specified threshold, a garbage collection sweep will be made. The existing `gc.collect()` function now takes an optional *generation* argument of 0, 1, or 2 to specify which generation to collect. (Contributed by Barry Warsaw.)

- The **`nsmallest()`** and **`nlargest()`** functions in the `heapq` module now support a `key` keyword parameter similar to the one provided by the `min()`/`max()` functions and the **`sort()`** methods. For example:

```
>>> import heapq
>>> L = ["short", 'medium', 'longest', 'longer still']
>>> heapq.nsmallest(2, L)  # Return two lowest elements,
lexicographically
['longer still', 'longest']
>>> heapq.nsmallest(2, L, key=len)   # Return two shortest
```

```
elements
['short', 'medium']
```

(Contributed by Raymond Hettinger.)

- The `itertools.islice()` function now accepts `None` for the start and step arguments. This makes it more compatible with the attributes of slice objects, so that you can now write the following:

  ```
  s = slice(5)     # Create slice object
  itertools.islice(iterable, s.start, s.stop, s.step)
  ```

  (Contributed by Raymond Hettinger.)

- The `format()` function in the `locale` module has been modified and two new functions were added, **`format_string()`** and **`currency()`**.

  The `format()` function's *val* parameter could previously be a string as long as no more than one %char specifier appeared; now the parameter must be exactly one %char specifier with no surrounding text. An optional *monetary* parameter was also added which, if `True`, will use the locale's rules for formatting currency in placing a separator between groups of three digits.

  To format strings with multiple %char specifiers, use the new **`format_string()`** function that works like `format()` but also supports mixing %char specifiers with arbitrary text.

  A new **`currency()`** function was also added that formats a number according to the current locale's settings.

  (Contributed by Georg Brandl.)

- The [`mailbox`] module underwent a massive rewrite to add the capability to modify mailboxes in addition to reading them. A new set of classes that include **`mbox`**, **`MH`**, and **`Maildir`** are used to read mailboxes, and have an `add(message)` method to add messages, `remove(key)` to remove messages, and **`lock()`**/**`unlock()`** to lock/unlock the mailbox. The following example converts a maildir-format mailbox into an mbox-format one:

  ```
  import mailbox

  # 'factory=None' uses email.Message.Message as the class
  representing
  # individual messages.
  src = mailbox.Maildir('maildir', factory=None)
  dest = mailbox.mbox('/tmp/mbox')

  for msg in src:
      dest.add(msg)
  ```

  (Contributed by Gregory K. Johnson. Funding was provided by Google’s 2005 Summer of Code.)

- New module: the **`msilib`** module allows creating Microsoft Installer `.msi` files and CAB files. Some support for reading the `.msi` database is also included. (Contributed by Martin von Löwis.)

- The **`nis`** module now supports accessing domains other than the system default domain by supplying a *domain* argument to the **`nis.match()`** and **`nis.maps()`** functions. (Contributed by Ben Bell.)

- The [`operator`] module’s **`itemgetter()`** and **`attrgetter()`** functions now support multiple fields. A call such as `operator.attrgetter('a', 'b')` will return a function that retrieves the **`a`** and **`b`** attributes. Combining this new feature with the **`sort()`** method’s `key` parameter lets you easily sort lists using multiple fields. (Contributed by Raymond Hettinger.)

- The `optparse` module was updated to version 1.5.1 of the Optik library. The **`OptionParser`** class gained an **`epilog`** attribute, a string that will be printed after the help message, and a **`destroy()`** method to break reference cycles created by the object. (Contributed by Greg Ward.)

- The `os` module underwent several changes. The **`stat_float_times`** variable now defaults to true, meaning that `os.stat()` will now return time values as floats. (This doesn't necessarily mean that `os.stat()` will return times that are precise to fractions of a second; not all systems support such precision.)

  Constants named `os.SEEK_SET`, `os.SEEK_CUR`, and `os.SEEK_END` have been added; these are the parameters to the `os.lseek()` function. Two new constants for locking are `os.O_SHLOCK` and `os.O_EXLOCK`.

  Two new functions, **`wait3()`** and **`wait4()`**, were added. They're similar the **`waitpid()`** function which waits for a child process to exit and returns a tuple of the process ID and its exit status, but **`wait3()`** and **`wait4()`** return additional information. **`wait3()`** doesn't take a process ID as input, so it waits for any child process to exit and returns a 3-tuple of *process-id*, *exit-status*, *resource-usage* as returned from the `resource.getrusage()` function. `wait4(pid)` does take a process ID. (Contributed by Chad J. Schroeder.)

  On FreeBSD, the `os.stat()` function now returns times with nanosecond resolution, and the returned object now has **`st_gen`** and **`st_birthtime`**. The **`st_flags`** attribute is also available, if the platform supports it. (Contributed by Antti Louko and Diego Pettenò.)

- The Python debugger provided by the `pdb` module can now store lists of commands to execute when a breakpoint is reached and execution stops. Once breakpoint #1 has been created, enter `commands 1` and enter a series of commands to be executed, finishing the list with `end`. The command list can include commands that resume execution, such as `continue` or `next`. (Contributed by Grégoire Dooms.)

- The `pickle` and **`cPickle`** modules no longer accept a return value of `None` from the `__reduce__()` method; the method must return a tuple of arguments instead. The ability to return `None` was deprecated in Python 2.4, so this completes the removal of the feature.

- The `pkgutil` module, containing various utility functions for finding packages, was enhanced to support **PEP 302** [https://peps.python.org/pep-0302/]’s import hooks and now also works for packages stored in ZIP-format archives. (Contributed by Phillip J. Eby.)

- The pybench benchmark suite by Marc-André Lemburg is now included in the `Tools/pybench` directory. The pybench suite is an improvement on the commonly used `pystone.py` program because pybench provides a more detailed measurement of the interpreter’s speed. It times particular operations such as function calls, tuple slicing, method lookups, and numeric operations, instead of performing many different operations and reducing the result to a single number as `pystone.py` does.

- The **`pyexpat`** module now uses version 2.0 of the Expat parser. (Contributed by Trent Mick.)

- The **`Queue`** class provided by the **`Queue`** module gained two new methods. **`join()`** blocks until all items in the queue have been retrieved and all processing work on the items have been completed. Worker threads call the other new method, **`task_done()`**, to signal that processing for an item has been completed. (Contributed by Raymond Hettinger.)

- The old **`regex`** and **`regsub`** modules, which have been deprecated ever since Python 2.0, have finally been deleted. Other deleted modules: **`statcache`**, **`tzparse`**, **`whrandom`**.

- Also deleted: the `lib-old` directory, which includes ancient modules such as **`dircmp`** and **`ni`**, was removed. `lib-old` wasn't on the default `sys.path`, so unless your programs explicitly added the directory to `sys.path`, this removal shouldn't affect your code.

- The **`rlcompleter`** module is no longer dependent on importing the **`readline`** module and therefore now works on non-Unix platforms. (Patch from Robert Kiendl.)

- The **`SimpleXMLRPCServer`** and **`DocXMLRPCServer`** classes now have a **`rpc_paths`** attribute that constrains XML-RPC operations to a limited set of URL paths; the default is to allow only `'/'` and `'/RPC2'`. Setting **`rpc_paths`** to `None` or an empty tuple disables this path checking.

- The **`socket`** module now supports **`AF_NETLINK`** sockets on Linux, thanks to a patch from Philippe Biondi. Netlink sockets are a Linux-specific mechanism for communications between a user-space process and kernel code; an introductory article about them is at https://www.linuxjournal.com/article/7356. In Python code, netlink addresses are represented as a tuple of 2 integers, `(pid, group_mask)`.

Two new methods on socket objects, `recv_into(buffer)` and `recvfrom_into(buffer)`, store the received data in an object that supports the buffer protocol instead of returning the data as a string. This means you can put the data directly into an array or a memory-mapped file.

Socket objects also gained **`getfamily()`**, **`gettype()`**, and **`getproto()`** accessor methods to retrieve the family, type, and protocol values for the socket.

- New module: the **`spwd`** module provides functions for accessing the shadow password database on systems that support shadow passwords.

- The **`struct`** is now faster because it compiles format strings into **`Struct`** objects with **`pack()`** and **`unpack()`** methods. This is similar to how the **`re`** module lets you create compiled regular expression objects. You can still use the module-level **`pack()`** and **`unpack()`** functions; they'll create **`Struct`** objects and cache them. Or you can use **`Struct`** instances directly:

  ```
  s = struct.Struct('ih3s')

  data = s.pack(1972, 187, 'abc')
  year, number, name = s.unpack(data)
  ```

  You can also pack and unpack data to and from buffer objects directly using the `pack_into(buffer, offset, v1, v2, ...)` and `unpack_from(buffer, offset)` methods. This lets you store data directly into an array or a memory-mapped file.

(**`Struct`** objects were implemented by Bob Ippolito at the NeedForSpeed sprint. Support for buffer objects was added by Martin Blais, also at the NeedForSpeed sprint.)

- The Python developers switched from CVS to Subversion during the 2.5 development process. Information about the exact build version is available as the `sys.subversion` variable, a 3-tuple of `(interpreter-name, branch-name, revision-range)`. For example, at the time of writing my copy of 2.5 was reporting `('CPython', 'trunk', '45313:45315')`.

  This information is also available to C extensions via the **`Py_GetBuildInfo()`** function that returns a string of build information like this: `"trunk:45355:45356M, Apr 13 2006, 07:42:19"`. (Contributed by Barry Warsaw.)

- Another new function, **`sys._current_frames()`**, returns the current stack frames for all running threads as a dictionary mapping thread identifiers to the topmost stack frame currently active in that thread at the time the function is called. (Contributed by Tim Peters.)

- The **`TarFile`** class in the **`tarfile`** module now has an **`extractall()`** method that extracts all members from the archive into the current working directory. It’s also possible to set a different directory as the extraction target, and to unpack only a subset of the archive’s members.

  The compression used for a tarfile opened in stream mode can now be autodetected using the mode `'r|*'`. (Contributed by Lars Gustäbel.)

- The **`threading`** module now lets you set the stack size used when new threads are created. The `stack_size([*size*])` function returns the currently configured stack size, and supplying the optional *size* parameter sets a new value. Not all platforms support changing the stack size, but Windows, POSIX threading, and OS/2 all do. (Contributed by Andrew MacIntyre.)

- The `unicodedata` module has been updated to use version 4.1.0 of the Unicode character database. Version 3.2.0 is required by some specifications, so it’s still available as `unicodedata.ucd_3_2_0`.
- New module: the `uuid` module generates universally unique identifiers (UUIDs) according to **RFC 4122** [https://datatracker.ietf.org/doc/html/rfc4122.html]. The RFC defines several different UUID versions that are generated from a starting string, from system properties, or purely randomly. This module contains a **`UUID`** class and functions named **`uuid1()`**, **`uuid3()`**, **`uuid4()`**, and **`uuid5()`** to generate different versions of UUID. (Version 2 UUIDs are not specified in **RFC 4122** [https://datatracker.ietf.org/doc/html/rfc4122.html] and are not supported by this module.)

```
>>> import uuid
>>> # make a UUID based on the host ID and current time
>>> uuid.uuid1()
UUID('a8098c1a-f86e-11da-bd1a-00112444be1e')

>>> # make a UUID using an MD5 hash of a namespace UUID and
a name
>>> uuid.uuid3(uuid.NAMESPACE_DNS, 'python.org')
UUID('6fa459ea-ee8a-3ca4-894e-db77e160355e')

>>> # make a random UUID
>>> uuid.uuid4()
UUID('16fd2706-8baf-433b-82eb-8c7fada847da')

>>> # make a UUID using a SHA-1 hash of a namespace UUID
and a name
```

```
>>> uuid.uuid5(uuid.NAMESPACE_DNS, 'python.org')
UUID('886313e1-3b8a-5372-9b90-0c9aee199e5d')
```

(Contributed by Ka-Ping Yee.)

- The `weakref` module's **`WeakKeyDictionary`** and **`WeakValueDictionary`** types gained new methods for iterating over the weak references contained in the dictionary. **`iterkeyrefs()`** and **`keyrefs()`** methods were added to **`WeakKeyDictionary`**, and **`itervaluerefs()`** and **`valuerefs()`** were added to **`WeakValueDictionary`**. (Contributed by Fred L. Drake, Jr.)

- The `webbrowser` module received a number of enhancements. It's now usable as a script with `python -m webbrowser`, taking a URL as the argument; there are a number of switches to control the behaviour (**`-n`** for a new browser window, **`-t`** for a new tab). New module-level functions, **`open_new()`** and **`open_new_tab()`**, were added to support this. The module's `open()` function supports an additional feature, an *autoraise* parameter that signals whether to raise the open window when possible. A number of additional browsers were added to the supported list such as Firefox, Opera, Konqueror, and elinks. (Contributed by Oleg Broytmann and Georg Brandl.)

- The `xmlrpclib` module now supports returning `datetime` objects for the XML-RPC date type. Supply `use_datetime=True` to the `loads()` function or the **`Unmarshaller`** class to enable this feature. (Contributed by Skip Montanaro.)

- The `zipfile` module now supports the ZIP64 version of the format, meaning that a .zip archive can now be larger than 4 GiB and can contain individual files larger than 4 GiB. (Contributed by Ronald Oussoren.)

- The `zlib` module’s **`Compress`** and **`Decompress`** objects now support a `copy()` method that makes a copy of the object’s internal state and returns a new **`Compress`** or **`Decompress`** object. (Contributed by Chris AtLee.)

The ctypes package

The `ctypes` package, written by Thomas Heller, has been added to the standard library. `ctypes` lets you call arbitrary functions in shared libraries or DLLs. Long-time users may remember the **`dl`** module, which provides functions for loading shared libraries and calling functions in them. The `ctypes` package is much fancier.

To load a shared library or DLL, you must create an instance of the **`CDLL`** class and provide the name or path of the shared library or DLL. Once that’s done, you can call arbitrary functions by accessing them as attributes of the **`CDLL`** object.

```
import ctypes

libc = ctypes.CDLL('libc.so.6')
result = libc.printf("Line of output\n")
```

Type constructors for the various C types are provided: **`c_int()`**, **`c_float()`**, **`c_double()`**, **`c_char_p()`** (equivalent to char*), and so forth. Unlike Python’s types, the C versions are all mutable; you can assign to their **`value`** attribute to change the wrapped value. Python integers and strings will be automatically converted to the corresponding C types, but for other types you must call the correct type constructor. (And I mean *must*; getting it wrong will often result in the interpreter crashing with a segmentation fault.)

You shouldn’t use **`c_char_p()`** with a Python string when the C function will be modifying the memory area, because Python strings are supposed to be immutable; breaking this rule will cause puzzling bugs. When you need a modifiable memory area, use **`create_string_buffer()`**:

```
s = "this is a string"
buf = ctypes.create_string_buffer(s)
libc.strfry(buf)
```

C functions are assumed to return integers, but you can set the **`restype`** attribute of the function object to change this:

```
>>> libc.atof('2.71828')
-1783957616
>>> libc.atof.restype = ctypes.c_double
>>> libc.atof('2.71828')
2.71828
```

`ctypes` also provides a wrapper for Python's C API as the `ctypes.pythonapi` object. This object does *not* release the global interpreter lock before calling a function, because the lock must be held when calling into the interpreter's code. There's a **`py_object`** type constructor that will create a PyObject* pointer. A simple usage:

```
import ctypes

d = {}
ctypes.pythonapi.PyObject_SetItem(ctypes.py_object(d),
          ctypes.py_object("abc"),  ctypes.py_object(1))
# d is now {'abc', 1}.
```

Don't forget to use **`py_object()`**; if it's omitted you end up with a segmentation fault.

`ctypes` has been around for a while, but people still write and distribution hand-coded extension modules because you can't rely on **`ctypes`** being present. Perhaps developers will begin to write Python wrappers atop a library accessed through **`ctypes`** instead of extension modules, now that **`ctypes`** is included with core Python.

See also

https://web.archive.org/web/20180410025338/http://starship.python.net/crew/theller/ctypes/
: The pre-stdlib ctypes web page, with a tutorial, reference, and FAQ.

The documentation for the `ctypes` module.

The ElementTree package

A subset of Fredrik Lundh’s ElementTree library for processing XML has been added to the standard library as `xml.etree`. The available modules are `ElementTree`, `ElementPath`, and `ElementInclude` from ElementTree 1.2.6. The `cElementTree` accelerator module is also included.

The rest of this section will provide a brief overview of using ElementTree. Full documentation for ElementTree is available at https://web.archive.org/web/20201124024954/http://effbot.org/zone/element-index.htm.

ElementTree represents an XML document as a tree of element nodes. The text content of the document is stored as the `text` and `tail` attributes of (This is one of the major differences between ElementTree and the Document Object Model; in the DOM there are many different types of node, including `TextNode`.)

The most commonly used parsing function is `parse()`, that takes either a string (assumed to contain a filename) or a file-like object and returns an `ElementTree` instance:

```
from xml.etree import ElementTree as ET

tree = ET.parse('ex-1.xml')

feed = urllib.urlopen(
          'http://planet.python.org/rss10.xml')
tree = ET.parse(feed)
```

Once you have an **`ElementTree`** instance, you can call its **`getroot()`** method to get the root **`Element`** node.

There’s also an **`XML()`** function that takes a string literal and returns an **`Element`** node (not an **`ElementTree`**). This function provides a tidy way to incorporate XML fragments, approaching the convenience of an XML literal:

```
svg = ET.XML("""<svg width="10px" version="1.0">
             </svg>""")
svg.set('height', '320px')
svg.append(elem1)
```

Each XML element supports some dictionary-like and some list-like access methods. Dictionary-like operations are used to access attribute values, and list-like operations are used to access child nodes.

Operation	Result
`elem[n]`	Returns n’th child element.
`elem[m:n]`	Returns list of m’th through n’th child elements.
`len(elem)`	Returns number of child elements.

Operation	**Result**
`list(elem)`	Returns list of child elements.
`elem.append(elem2)`	Adds *elem2* as a child.
`elem.insert(index, elem2)`	Inserts *elem2* at the specified location.
`del elem[n]`	Deletes n’th child element.
`elem.keys()`	Returns list of attribute names.
`elem.get(name)`	Returns value of attribute *name*.
`elem.set(name, value)`	Sets new value for attribute *name*.
`elem.attrib`	Retrieves the dictionary containing attributes.
`del elem.attrib[name]`	Deletes attribute *name*.

Comments and processing instructions are also represented as **`Element`** nodes. To check if a node is a comment or processing instructions:

```
if elem.tag is ET.Comment:
    ...
elif elem.tag is ET.ProcessingInstruction:
    ...
```

To generate XML output, you should call the **`ElementTree.write()`** method. Like **`parse()`**, it can take either a string or a file-like object:

```
# Encoding is US-ASCII
tree.write('output.xml')

# Encoding is UTF-8
f = open('output.xml', 'w')
tree.write(f, encoding='utf-8')
```

(Caution: the default encoding used for output is ASCII. For general XML work, where an element’s name may contain arbitrary Unicode characters, ASCII isn’t a very useful encoding because it will raise an exception if an element’s name contains any characters with values greater than 127. Therefore, it’s best to specify a different encoding such as UTF-8 that can handle any Unicode character.)

This section is only a partial description of the ElementTree interfaces. Please read the package’s official documentation for more details.

See also

https://web.archive.org/web/20201124024954/http://effbot.org/zone/element-index.htm
Official documentation for ElementTree.

The hashlib package

A new `hashlib` module, written by Gregory P. Smith, has been added to replace the **md5** and **sha** modules. `hashlib` adds support for additional secure hashes (SHA-224, SHA-256, SHA-384, and SHA-512). When available, the module uses OpenSSL for fast platform optimized implementations of algorithms.

The old **md5** and **sha** modules still exist as wrappers around hashlib to preserve backwards compatibility. The new module's interface is very close to that of the old modules, but not identical. The most significant difference is that the constructor functions for creating new hashing objects are named differently.

```
# Old versions
h = md5.md5()
h = md5.new()

# New version
h = hashlib.md5()

# Old versions
h = sha.sha()
h = sha.new()

# New version
h = hashlib.sha1()

# Hash that weren't previously available
h = hashlib.sha224()
h = hashlib.sha256()
h = hashlib.sha384()
h = hashlib.sha512()

# Alternative form
h = hashlib.new('md5')          # Provide algorithm as a string
```

Once a hash object has been created, its methods are the same as before: `update(string)` hashes the specified string into the current digest state, **`digest()`** and **`hexdigest()`** return the digest value as a binary string or a string of hex digits, and `copy()` returns a new hashing object with the same digest state.

See also

The documentation for the `hashlib` module.

The sqlite3 package

The pysqlite module (https://www.pysqlite.org), a wrapper for the SQLite embedded database, has been added to the standard library under the package name `sqlite3`.

SQLite is a C library that provides a lightweight disk-based database that doesn’t require a separate server process and allows accessing the database using a nonstandard variant of the SQL query language. Some applications can use SQLite for internal data storage. It’s also possible to prototype an application using SQLite and then port the code to a larger database such as PostgreSQL or Oracle.

pysqlite was written by Gerhard Häring and provides a SQL interface compliant with the DB-API 2.0 specification described by **PEP 249** [https://peps.python.org/pep-0249/].

If you’re compiling the Python source yourself, note that the source tree doesn’t include the SQLite code, only the wrapper module. You’ll need to have the SQLite libraries and headers installed before compiling Python, and the build process will compile the module when the necessary headers are available.

To use the module, you must first create a `Connection` object that represents the database. Here the data will be stored in the `/tmp/example` file:

```
conn = sqlite3.connect('/tmp/example')
```

You can also supply the special name `:memory:` to create a database in RAM.

Once you have a **`Connection`**, you can create a **`Cursor`** object and call its **`execute()`** method to perform SQL commands:

```
c = conn.cursor()

# Create table
c.execute('''create table stocks
(date text, trans text, symbol text,
 qty real, price real)''')

# Insert a row of data
c.execute("""insert into stocks
          values ('2006-01-05','BUY','RHAT',100,35.14)""")
```

Usually your SQL operations will need to use values from Python variables. You shouldn’t assemble your query using Python’s string operations because doing so is insecure; it makes your program vulnerable to an SQL injection attack.

Instead, use the DB-API’s parameter substitution. Put `?` as a placeholder wherever you want to use a value, and then provide a tuple of values as the second argument to the cursor’s **`execute()`** method. (Other database modules may use a different placeholder, such as `%s` or `:1`.) For example:

```
# Never do this -- insecure!
symbol = 'IBM'
c.execute("... where symbol = '%s'" % symbol)

# Do this instead
t = (symbol,)
c.execute('select * from stocks where symbol=?', t)
```

```
# Larger example
for t in (('2006-03-28', 'BUY', 'IBM', 1000, 45.00),
          ('2006-04-05', 'BUY', 'MSOFT', 1000, 72.00),
          ('2006-04-06', 'SELL', 'IBM', 500, 53.00),
         ):
    c.execute('insert into stocks values (?,?,?,?,?)', t)
```

To retrieve data after executing a SELECT statement, you can either treat the cursor as an iterator, call the cursor's **`fetchone()`** method to retrieve a single matching row, or call **`fetchall()`** to get a list of the matching rows.

This example uses the iterator form:

```
>>> c = conn.cursor()
>>> c.execute('select * from stocks order by price')
>>> for row in c:
...    print row
...
(u'2006-01-05', u'BUY', u'RHAT', 100, 35.140000000000001)
(u'2006-03-28', u'BUY', u'IBM', 1000, 45.0)
(u'2006-04-06', u'SELL', u'IBM', 500, 53.0)
(u'2006-04-05', u'BUY', u'MSOFT', 1000, 72.0)
>>>
```

For more information about the SQL dialect supported by SQLite, see https://www.sqlite.org.

See also

https://www.pysqlite.org
The pysqlite web page.

https://www.sqlite.org
The SQLite web page; the documentation describes the syntax and the available data types for the supported SQL dialect.

The documentation for the `sqlite3` module.

PEP 249 **[https://peps.python.org/pep-0249/] - Database API Specification 2.0**
PEP written by Marc-André Lemburg.

The wsgiref package

The Web Server Gateway Interface (WSGI) v1.0 defines a standard interface between web servers and Python web applications and is described in **PEP 333** [https://peps.python.org/pep-0333/]. The `wsgiref` package is a reference implementation of the WSGI specification.

The package includes a basic HTTP server that will run a WSGI application; this server is useful for debugging but isn’t intended for production use. Setting up a server takes only a few lines of code:

```
from wsgiref import simple_server

wsgi_app = ...

host = ''
port = 8000
httpd = simple_server.make_server(host, port, wsgi_app)
httpd.serve_forever()
```

See also

https://web.archive.org/web/20160331090247/http://wsgi.readthedocs.org/en/latest/
A central web site for WSGI-related resources.

PEP 333 **[https://peps.python.org/pep-0333/] - Python Web Server Gateway Interface v1.0**
PEP written by Phillip J. Eby.

Build and C API Changes

Changes to Python’s build process and to the C API include:

- The Python source tree was converted from CVS to Subversion, in a complex migration procedure that was supervised and flawlessly carried out by Martin von Löwis. The procedure was developed as **PEP 347** [https://peps.python.org/pep-0347/].

- Coverity, a company that markets a source code analysis tool called Prevent, provided the results of their examination of the Python source code. The analysis found about 60 bugs that were quickly fixed. Many of the bugs were refcounting problems, often occurring in error-handling code. See https://scan.coverity.com for the statistics.

- The largest change to the C API came from **PEP 353** [https://peps.python.org/pep-0353/], which modifies the interpreter to use a **`Py_ssize_t`** type definition instead of int. See the earlier section PEP 353: Using ssize_t as the index type for a discussion of this change.

- The design of the bytecode compiler has changed a great deal, no longer generating bytecode by traversing the parse tree. Instead the parse tree is converted to an abstract syntax tree (or AST), and it is the abstract syntax tree that’s traversed to produce the bytecode.

  It’s possible for Python code to obtain AST objects by using the **`compile()`** built-in and specifying `_ast.PyCF_ONLY_AST` as the value of the *flags* parameter:

  ```
  from _ast import PyCF_ONLY_AST
  ast = compile("""a=0
  for i in range(10):
  ```

```
    a += i
""", "<string>", 'exec', PyCF_ONLY_AST)

assignment = ast.body[0]
for_loop = ast.body[1]
```

No official documentation has been written for the AST code yet, but **PEP 339** [https://peps.python.org/pep-0339/] discusses the design. To start learning about the code, read the definition of the various AST nodes in `Parser/Python.asdl`. A Python script reads this file and generates a set of C structure definitions in `Include/Python-ast.h`. The **`PyParser_ASTFromString()`** and **`PyParser_ASTFromFile()`**, defined in `Include/pythonrun.h`, take Python source as input and return the root of an AST representing the contents. This AST can then be turned into a code object by **`PyAST_Compile()`**. For more information, read the source code, and then ask questions on python-dev.

The AST code was developed under Jeremy Hylton's management, and implemented by (in alphabetical order) Brett Cannon, Nick Coghlan, Grant Edwards, John Ehresman, Kurt Kaiser, Neal Norwitz, Tim Peters, Armin Rigo, and Neil Schemenauer, plus the participants in a number of AST sprints at conferences such as PyCon.

- Evan Jones's patch to obmalloc, first described in a talk at PyCon DC 2005, was applied. Python 2.4 allocated small objects in 256K-sized arenas, but never freed arenas. With this patch, Python will free arenas when they're empty. The net effect is that on some platforms, when you allocate many objects, Python's memory usage may actually drop when you delete them and the memory may be returned to the operating system. (Implemented by Evan Jones, and reworked by Tim Peters.)

Note that this change means extension modules must be more careful when allocating memory. Python's API has many different functions for allocating memory that are grouped into families. For example, `PyMem_Malloc()`, `PyMem_Realloc()`, and `PyMem_Free()` are one family that allocates raw memory, while `PyObject_Malloc()`, `PyObject_Realloc()`, and `PyObject_Free()` are another family that's supposed to be used for creating Python objects.

Previously these different families all reduced to the platform's **`malloc()`** and **`free()`** functions. This meant it didn't matter if you got things wrong and allocated memory with the `PyMem` function but freed it with the `PyObject` function. With 2.5's changes to obmalloc, these families now do different things and mismatches will probably result in a segfault. You should carefully test your C extension modules with Python 2.5.

- The built-in set types now have an official C API. Call `PySet_New()` and `PyFrozenSet_New()` to create a new set, `PySet_Add()` and `PySet_Discard()` to add and remove elements, and `PySet_Contains()` and `PySet_Size()` to examine the set's state. (Contributed by Raymond Hettinger.)
- C code can now obtain information about the exact revision of the Python interpreter by calling the `Py_GetBuildInfo()` function that returns a string of build information like this: `"trunk:45355:45356M, Apr 13 2006, 07:42:19"`. (Contributed by Barry Warsaw.)
- Two new macros can be used to indicate C functions that are local to the current file so that a faster calling convention can be used. `Py_LOCAL` declares the function as returning a value of the specified *type* and uses a fast-calling qualifier. `Py_LOCAL_INLINE` does the same thing

and also requests the function be inlined. If macro **`PY_LOCAL_AGGRESSIVE`** is defined before `python.h` is included, a set of more aggressive optimizations are enabled for the module; you should benchmark the results to find out if these optimizations actually make the code faster. (Contributed by Fredrik Lundh at the NeedForSpeed sprint.)

- `PyErr_NewException(name, base, dict)` can now accept a tuple of base classes as its *base* argument. (Contributed by Georg Brandl.)

- The **`PyErr_Warn()`** function for issuing warnings is now deprecated in favour of `PyErr_WarnEx(category, message, stacklevel)` which lets you specify the number of stack frames separating this function and the caller. A *stacklevel* of 1 is the function calling **`PyErr_WarnEx()`**, 2 is the function above that, and so forth. (Added by Neal Norwitz.)

- The CPython interpreter is still written in C, but the code can now be compiled with a C++ compiler without errors. (Implemented by Anthony Baxter, Martin von Löwis, Skip Montanaro.)

- The **`PyRange_New()`** function was removed. It was never documented, never used in the core code, and had dangerously lax error checking. In the unlikely case that your extensions were using it, you can replace it by something like the following:

```
range = PyObject_CallFunction((PyObject*) &PyRange_Type,
"lll",
                              start, stop, step);
```

Port-Specific Changes

- MacOS X (10.3 and higher): dynamic loading of modules now uses the **`dlopen()`** function instead of MacOS-specific functions.
- MacOS X: an **`--enable-universalsdk`** switch was added to the **configure** script that compiles the interpreter as a universal binary able to run on both PowerPC and Intel processors. (Contributed by Ronald Oussoren; bpo-2573 [https://bugs.python.org/issue?@action=redirect&bpo=2573].)
- Windows: `.dll` is no longer supported as a filename extension for extension modules. `.pyd` is now the only filename extension that will be searched for.

Porting to Python 2.5

This section lists previously described changes that may require changes to your code:

- ASCII is now the default encoding for modules. It's now a syntax error if a module contains string literals with 8-bit characters but doesn't have an encoding declaration. In Python 2.4 this triggered a warning, not a syntax error.
- Previously, the **`gi_frame`** attribute of a generator was always a frame object. Because of the **PEP 342** [https://peps.python.org/pep-0342/] changes described in section PEP 342: New Generator Features, it's now possible for **`gi_frame`** to be `None`.
- A new warning, **`UnicodeWarning`**, is triggered when you attempt to compare a Unicode string and an 8-bit string that can't be converted to Unicode using the default ASCII encoding. Previously such comparisons would raise a **`UnicodeDecodeError`** exception.
- Library: the **`csv`** module is now stricter about multi-line quoted fields. If your files contain newlines embedded

within fields, the input should be split into lines in a manner which preserves the newline characters.
- Library: the `locale` module’s `format()` function’s would previously accept any string as long as no more than one %char specifier appeared. In Python 2.5, the argument must be exactly one %char specifier with no surrounding text.
- Library: The `pickle` and **`cPickle`** modules no longer accept a return value of `None` from the `__reduce__()` method; the method must return a tuple of arguments instead. The modules also no longer accept the deprecated *bin* keyword parameter.
- Library: The `SimpleXMLRPCServer` and `DocXMLRPCServer` classes now have a **`rpc_paths`** attribute that constrains XML-RPC operations to a limited set of URL paths; the default is to allow only `'/'` and `'/RPC2'`. Setting **`rpc_paths`** to `None` or an empty tuple disables this path checking.
- C API: Many functions now use `Py_ssize_t` instead of int to allow processing more data on 64-bit machines. Extension code may need to make the same change to avoid warnings and to support 64-bit machines. See the earlier section PEP 353: Using ssize_t as the index type for a discussion of this change.
- C API: The obmalloc changes mean that you must be careful to not mix usage of the `PyMem_*` and `PyObject_*` families of functions. Memory allocated with one family’s `*_Malloc` must be freed with the corresponding family’s `*_Free` function.

Acknowledgements

The author would like to thank the following people for offering suggestions, corrections and assistance with various drafts of this article: Georg Brandl, Nick Coghlan, Phillip J. Eby, Lars Gustäbel, Raymond Hettinger, Ralf W.

Grosse-Kunstleve, Kent Johnson, Iain Lowe, Martin von Löwis, Fredrik Lundh, Andrew McNamara, Skip Montanaro, Gustavo Niemeyer, Paul Prescod, James Pryor, Mike Rovner, Scott Weikart, Barry Warsaw, Thomas Wouters.

What’s New in Python 2.4

Author:: A.M. Kuchling

This article explains the new features in Python 2.4.1, released on March 30, 2005.

Python 2.4 is a medium-sized release. It doesn’t introduce as many changes as the radical Python 2.2, but introduces more features than the conservative 2.3 release. The most significant new language features are function decorators and generator expressions; most other changes are to the standard library.

According to the CVS change logs, there were 481 patches applied and 502 bugs fixed between Python 2.3 and 2.4. Both figures are likely to be underestimates.

This article doesn’t attempt to provide a complete specification of every single new feature, but instead provides a brief introduction to each feature. For full details, you should refer to the documentation for Python 2.4, such as the Python Library Reference and the Python Reference Manual. Often you will be referred to the PEP for a particular new feature for explanations of the implementation and design rationale.

PEP 218: Built-In Set Objects

Python 2.3 introduced the **`sets`** module. C implementations of set data types have now been added to the Python core as two new built-in types, `set(iterable)` and `frozenset(iterable)`. They provide high speed operations for membership testing, for eliminating duplicates from sequences, and for mathematical operations like unions, intersections, differences, and symmetric differences.

```
>>> a = set('abracadabra')              # form a set from a
string
>>> 'z' in a                            # fast membership
testing
False
>>> a                                   # unique letters in a
set(['a', 'r', 'b', 'c', 'd'])
>>> ''.join(a)                          # convert back into a
string
'arbcd'

>>> b = set('alacazam')                 # form a second set
>>> a - b                               # letters in a but not
in b
set(['r', 'd', 'b'])
>>> a | b                               # letters in either a
or b
set(['a', 'c', 'r', 'd', 'b', 'm', 'z', 'l'])
>>> a & b                               # letters in both a and
b
set(['a', 'c'])
>>> a ^ b                               # letters in a or b but
not both
set(['r', 'd', 'b', 'm', 'z', 'l'])

>>> a.add('z')                          # add a new element
>>> a.update('wxy')                     # add multiple new
elements
>>> a
set(['a', 'c', 'b', 'd', 'r', 'w', 'y', 'x', 'z'])
>>> a.remove('x')                       # take one element out
>>> a
set(['a', 'c', 'b', 'd', 'r', 'w', 'y', 'z'])
```

The `frozenset()` type is an immutable version of `set()`. Since it is immutable and hashable, it may be used as a dictionary key or as a member of another set.

The `sets` module remains in the standard library, and may be useful if you wish to subclass the `Set` or `ImmutableSet` classes. There are currently no plans to deprecate the module.

See also

PEP 218 [https://peps.python.org/pep-0218/] - Adding a Built-In Set Object Type
Originally proposed by Greg Wilson and ultimately implemented by Raymond Hettinger.

PEP 237: Unifying Long Integers and Integers

The lengthy transition process for this PEP, begun in Python 2.2, takes another step forward in Python 2.4. In 2.3, certain integer operations that would behave differently after int/long unification triggered `FutureWarning` warnings and returned values limited to 32 or 64 bits (depending on your platform). In 2.4, these expressions no longer produce a warning and instead produce a different result that's usually a long integer.

The problematic expressions are primarily left shifts and lengthy hexadecimal and octal constants. For example, `2 << 32` results in a warning in 2.3, evaluating to 0 on 32-bit platforms. In Python 2.4, this expression now returns the correct answer, 8589934592.

See also

PEP 237 [https://peps.python.org/pep-0237/] - Unifying Long Integers and Integers

> Original PEP written by Moshe Zadka and GvR. The changes for 2.4 were implemented by Kalle Svensson.

PEP 289: Generator Expressions

The iterator feature introduced in Python 2.2 and the `itertools` module make it easier to write programs that loop through large data sets without having the entire data set in memory at one time. List comprehensions don’t fit into this picture very well because they produce a Python list object containing all of the items. This unavoidably pulls all of the objects into memory, which can be a problem if your data set is very large. When trying to write a functionally styled program, it would be natural to write something like:

```
links = [link for link in get_all_links() if not link.followed]
for link in links:
    ...
```

instead of

```
for link in get_all_links():
    if link.followed:
        continue
    ...
```

The first form is more concise and perhaps more readable, but if you’re dealing with a large number of link objects you’d have to write the second form to avoid having all link objects in memory at the same time.

Generator expressions work similarly to list comprehensions but don’t materialize the entire list; instead they create a generator that will return elements one by one. The above example could be written as:

```
links = (link for link in get_all_links() if not link.followed)
for link in links:
    ...
```

Generator expressions always have to be written inside parentheses, as in the above example. The parentheses signalling a function call also count, so if you want to create an iterator that will be immediately passed to a function you could write:

```
print sum(obj.count for obj in list_all_objects())
```

Generator expressions differ from list comprehensions in various small ways. Most notably, the loop variable (*obj* in the above example) is not accessible outside of the generator expression. List comprehensions leave the variable assigned to its last value; future versions of Python will change this, making list comprehensions match generator expressions in this respect.

See also

PEP 289 [https://peps.python.org/pep-0289/] - Generator Expressions

Proposed by Raymond Hettinger and implemented by Jiwon Seo with early efforts steered by Hye-Shik Chang.

PEP 292: Simpler String Substitutions

Some new classes in the standard library provide an alternative mechanism for substituting variables into strings; this style of substitution may be better for applications where untrained users need to edit templates.

The usual way of substituting variables by name is the `%` operator:

```
>>> '%(page)i: %(title)s' % {'page':2, 'title': 'The Best of
Times'}
'2: The Best of Times'
```

When writing the template string, it can be easy to forget the `i` or `s` after the closing parenthesis. This isn’t a big problem if the template is in a Python module, because you run the code, get an “Unsupported format character” `ValueError`, and fix the problem. However, consider an application such as Mailman where template strings or translations are being edited by users who aren’t aware of the Python language. The format string’s syntax is complicated to explain to such users, and if they make a mistake, it’s difficult to provide helpful feedback to them.

PEP 292 adds a **`Template`** class to the `string` module that uses `$` to indicate a substitution:

```
>>> import string
>>> t = string.Template('$page: $title')
>>> t.substitute({'page':2, 'title': 'The Best of Times'})
'2: The Best of Times'
```

If a key is missing from the dictionary, the **`substitute()`** method will raise a `KeyError`. There’s also a **`safe_substitute()`** method that ignores missing keys:

```
>>> t = string.Template('$page: $title')
>>> t.safe_substitute({'page':3})
'3: $title'
```

See also

PEP 292 [https://peps.python.org/pep-0292/] - Simpler String Substitutions

Written and implemented by Barry Warsaw.

PEP 318: Decorators for Functions and Methods

Python 2.2 extended Python’s object model by adding static methods and class methods, but it didn’t extend Python’s syntax to provide any new way of defining static or class methods. Instead, you had to write a `def` statement in the usual way, and pass the resulting method to a `staticmethod()` or `classmethod()` function that would wrap up the function as a method of the new type. Your code would look like this:

```
class C:
   def meth (cls):
       ...

   meth = classmethod(meth)   # Rebind name to wrapped-up class
method
```

If the method was very long, it would be easy to miss or forget the `classmethod()` invocation after the function body.

The intention was always to add some syntax to make such definitions more readable, but at the time of 2.2’s release a good syntax was not obvious. Today a good syntax *still* isn’t obvious but users are asking for easier access to the feature; a new syntactic feature has been added to meet this need.

The new feature is called “function decorators”. The name comes from the idea that `classmethod()`, `staticmethod()`, and friends are storing additional information on a function object; they’re *decorating* functions with more details.

The notation borrows from Java and uses the `'@'` character as an indicator. Using the new syntax, the example above would be written:

```
class C:

   @classmethod
   def meth (cls):
       ...
```

The `@classmethod` is shorthand for the `meth=classmethod(meth)` assignment. More generally, if you have the following:

```
@A
@B
@C
def f ():
    ...
```

It’s equivalent to the following pre-decorator code:

```
def f(): ...
f = A(B(C(f)))
```

Decorators must come on the line before a function definition, one decorator per line, and can’t be on the same line as the def statement, meaning that `@A def f(): ...` is illegal. You can only decorate function definitions, either at the module level or inside a class; you can’t decorate class definitions.

A decorator is just a function that takes the function to be decorated as an argument and returns either the same function or some new object. The return value of the decorator need not be callable (though it typically is), unless further decorators will be applied to the result. It’s easy to write your own decorators. The following simple example just sets an attribute on the function object:

```
>>> def deco(func):
...    func.attr = 'decorated'
...    return func
...
>>> @deco
... def f(): pass
...
>>> f
<function f at 0x402ef0d4>
>>> f.attr
'decorated'
>>>
```

As a slightly more realistic example, the following decorator checks that the supplied argument is an integer:

```
def require_int (func):
    def wrapper (arg):
        assert isinstance(arg, int)
        return func(arg)

    return wrapper

@require_int
def p1 (arg):
    print arg

@require_int
def p2(arg):
    print arg*2
```

An example in **PEP 318** [https://peps.python.org/pep-0318/] contains a fancier version of this idea that lets you both specify the required type and check the returned type.

Decorator functions can take arguments. If arguments are supplied, your decorator function is called with only those arguments and must return a new decorator function; this function must take a single function and return a function, as previously described. In other words, `@A @B @C(args)` becomes:

```
def f(): ...
_deco = C(args)
f = A(B(_deco(f)))
```

Getting this right can be slightly brain-bending, but it’s not too difficult.

A small related change makes the `func_name` attribute of functions writable. This attribute is used to display function names in tracebacks, so decorators should change the name of any new function that’s constructed and returned.

See also

PEP 318 [https://peps.python.org/pep-0318/] - Decorators for Functions, Methods and Classes

Written by Kevin D. Smith, Jim Jewett, and Skip Montanaro. Several people wrote patches implementing function decorators, but the one that was actually checked in was patch #979728, written by Mark Russell.

https://wiki.python.org/moin/PythonDecoratorLibrary

This Wiki page contains several examples of decorators.

PEP 322: Reverse Iteration

A new built-in function, `reversed(seq)`, takes a sequence and returns an iterator that loops over the elements of the sequence in reverse order.

```
>>> for i in reversed(xrange(1,4)):
...    print i
...
3
```

```
2
1
```

Compared to extended slicing, such as `range(1,4)[::-1]`, **`reversed()`** is easier to read, runs faster, and uses substantially less memory.

Note that **`reversed()`** only accepts sequences, not arbitrary iterators. If you want to reverse an iterator, first convert it to a list with **`list()`**.

```
>>> input = open('/etc/passwd', 'r')
>>> for line in reversed(list(input)):
...   print line
...
root:*:0:0:System Administrator:/var/root:/bin/tcsh
  ...
```

See also

PEP 322 [https://peps.python.org/pep-0322/] - Reverse Iteration
Written and implemented by Raymond Hettinger.

PEP 324: New subprocess Module

The standard library provides a number of ways to execute a subprocess, offering different features and different levels of complexity. `os.system(command)` is easy to use, but slow (it runs a shell process which executes the command) and dangerous (you have to be careful about escaping the shell’s metacharacters). The **`popen2`** module offers classes that can capture standard output and standard error from the subprocess, but the naming is confusing. The **`subprocess`** module cleans this up, providing a unified interface that offers all the features you might need.

Instead of **`popen2`**'s collection of classes, [**`subprocess`**] contains a single class called [**`subprocess.Popen`**] whose constructor supports a number of different keyword arguments.

```
class Popen(args, bufsize=0, executable=None,
            stdin=None, stdout=None, stderr=None,
            preexec_fn=None, close_fds=False, shell=False,
            cwd=None, env=None, universal_newlines=False,
            startupinfo=None, creationflags=0):
```

args is commonly a sequence of strings that will be the arguments to the program executed as the subprocess. (If the *shell* argument is true, *args* can be a string which will then be passed on to the shell for interpretation, just as **`os.system()`** does.)

stdin, *stdout*, and *stderr* specify what the subprocess's input, output, and error streams will be. You can provide a file object or a file descriptor, or you can use the constant `subprocess.PIPE` to create a pipe between the subprocess and the parent.

The constructor has a number of handy options:

- *close_fds* requests that all file descriptors be closed before running the subprocess.
- *cwd* specifies the working directory in which the subprocess will be executed (defaulting to whatever the parent's working directory is).
- *env* is a dictionary specifying environment variables.
- *preexec_fn* is a function that gets called before the child is started.
- *universal_newlines* opens the child's input and output using Python's universal newlines feature.

Once you've created the **`Popen`** instance, you can call its **`wait()`** method to pause until the subprocess has exited,

`poll()` to check if it’s exited without pausing, or `communicate(data)` to send the string *data* to the subprocess’s standard input. `communicate(data)` then reads any data that the subprocess has sent to its standard output or standard error, returning a tuple `(stdout_data, stderr_data)`.

`call()` is a shortcut that passes its arguments along to the **`Popen`** constructor, waits for the command to complete, and returns the status code of the subprocess. It can serve as a safer analog to **`os.system()`**:

```
sts = subprocess.call(['dpkg', '-i', '/tmp/new-package.deb'])
if sts == 0:
    # Success
    ...
else:
    # dpkg returned an error
    ...
```

The command is invoked without use of the shell. If you really do want to use the shell, you can add `shell=True` as a keyword argument and provide a string instead of a sequence:

```
sts = subprocess.call('dpkg -i /tmp/new-package.deb',
shell=True)
```

The PEP takes various examples of shell and Python code and shows how they’d be translated into Python code that uses **`subprocess`**. Reading this section of the PEP is highly recommended.

See also

PEP 324 [https://peps.python.org/pep-0324/] - subprocess - New process module

> Written and implemented by Peter Åstrand, with assistance from Fredrik Lundh and others.

PEP 327: Decimal Data Type

Python has always supported floating-point (FP) numbers, based on the underlying C double type, as a data type. However, while most programming languages provide a floating-point type, many people (even programmers) are unaware that floating-point numbers don’t represent certain decimal fractions accurately. The new **`Decimal`** type can represent these fractions accurately, up to a user-specified precision limit.

Why is Decimal needed?

The limitations arise from the representation used for floating-point numbers. FP numbers are made up of three components:

- The sign, which is positive or negative.
- The mantissa, which is a single-digit binary number followed by a fractional part. For example, `1.01` in base-2 notation is `1 + 0/2 + 1/4`, or 1.25 in decimal notation.
- The exponent, which tells where the decimal point is located in the number represented.

For example, the number 1.25 has positive sign, a mantissa value of 1.01 (in binary), and an exponent of 0 (the decimal point doesn’t need to be shifted). The number 5 has the same sign and mantissa, but the exponent is 2 because the mantissa is multiplied by 4 (2 to the power of the exponent 2); 1.25 * 4 equals 5.

Modern systems usually provide floating-point support that conforms to a standard called IEEE 754. C’s double type is usually implemented as a 64-bit IEEE 754 number, which uses 52 bits of space for the mantissa. This means that numbers can only be specified to 52 bits of precision. If you’re trying to represent numbers whose expansion repeats endlessly, the expansion is cut off after 52 bits. Unfortunately, most software needs to produce output in base 10, and common fractions in base 10 are often repeating decimals in binary. For example, 1.1 decimal is binary `1.0001100110011` ...; .1 = 1/16 + 1/32 + 1/256 plus an infinite number of additional terms. IEEE 754 has to chop off that infinitely repeated decimal after 52 digits, so the representation is slightly inaccurate.

Sometimes you can see this inaccuracy when the number is printed:

```
>>> 1.1
1.1000000000000001
```

The inaccuracy isn’t always visible when you print the number because the FP-to-decimal-string conversion is provided by the C library, and most C libraries try to produce sensible output. Even if it’s not displayed, however, the inaccuracy is still there and subsequent operations can magnify the error.

For many applications this doesn’t matter. If I’m plotting points and displaying them on my monitor, the difference between 1.1 and 1.1000000000000001 is too small to be visible. Reports often limit output to a certain number of decimal places, and if you round the number to two or three or even eight decimal places, the error is never apparent. However, for applications where it does matter, it’s a lot of work to implement your own custom arithmetic routines.

Hence, the **`Decimal`** type was created.

The `Decimal` type

A new module, [`decimal`], was added to Python’s standard library. It contains two classes, **`Decimal`** and **`Context`**. **`Decimal`** instances represent numbers, and **`Context`** instances are used to wrap up various settings such as the precision and default rounding mode.

`Decimal` instances are immutable, like regular Python integers and FP numbers; once it’s been created, you can’t change the value an instance represents. **`Decimal`** instances can be created from integers or strings:

```
>>> import decimal
>>> decimal.Decimal(1972)
Decimal("1972")
>>> decimal.Decimal("1.1")
Decimal("1.1")
```

You can also provide tuples containing the sign, the mantissa represented as a tuple of decimal digits, and the exponent:

```
>>> decimal.Decimal((1, (1, 4, 7, 5), -2))
Decimal("-14.75")
```

Cautionary note: the sign bit is a Boolean value, so 0 is positive and 1 is negative.

Converting from floating-point numbers poses a bit of a problem: should the FP number representing 1.1 turn into the decimal number for exactly 1.1, or for 1.1 plus whatever inaccuracies are introduced? The decision was to dodge the issue and leave such a conversion out of the API. Instead, you should convert the floating-point number into a string using the desired precision and pass the string to the **`Decimal`** constructor:

```
>>> f = 1.1
>>> decimal.Decimal(str(f))
Decimal("1.1")
>>> decimal.Decimal('%.12f' % f)
Decimal("1.100000000000")
```

Once you have **`Decimal`** instances, you can perform the usual mathematical operations on them. One limitation: exponentiation requires an integer exponent:

```
>>> a = decimal.Decimal('35.72')
>>> b = decimal.Decimal('1.73')
>>> a+b
Decimal("37.45")
>>> a-b
Decimal("33.99")
>>> a*b
Decimal("61.7956")
>>> a/b
Decimal("20.64739884393063583815028902")
>>> a ** 2
Decimal("1275.9184")
>>> a**b
Traceback (most recent call last):
  ...
decimal.InvalidOperation: x ** (non-integer)
```

You can combine **`Decimal`** instances with integers, but not with floating-point numbers:

```
>>> a + 4
Decimal("39.72")
>>> a + 4.5
Traceback (most recent call last):
  ...
TypeError: You can interact Decimal only with int, long or
Decimal data types.
>>>
```

`Decimal` numbers can be used with the `math` and `cmath` modules, but note that they’ll be immediately converted to floating-point numbers before the operation is performed, resulting in a possible loss of precision and accuracy. You’ll also get back a regular floating-point number and not a **`Decimal`**.

```
>>> import math, cmath
>>> d = decimal.Decimal('123456789012.345')
>>> math.sqrt(d)
351364.18288201344
>>> cmath.sqrt(-d)
351364.18288201344j
```

`Decimal` instances have a **`sqrt()`** method that returns a **`Decimal`**, but if you need other things such as trigonometric functions you’ll have to implement them.

```
>>> d.sqrt()
Decimal("351364.1828820134592177245001")
```

The `Context` type

Instances of the **`Context`** class encapsulate several settings for decimal operations:

- **`prec`** is the precision, the number of decimal places.
- **`rounding`** specifies the rounding mode. The `decimal` module has constants for the various possibilities: **`ROUND_DOWN`**, **`ROUND_CEILING`**, **`ROUND_HALF_EVEN`**, and various others.
- **`traps`** is a dictionary specifying what happens on encountering certain error conditions: either an exception is raised or a value is returned. Some examples of error conditions are division by zero, loss of precision, and overflow.

There’s a thread-local default context available by calling **`getcontext()`**; you can change the properties of this context to alter the default precision, rounding, or trap handling. The following example shows the effect of changing the precision of the default context:

```
>>> decimal.getcontext().prec
28
>>> decimal.Decimal(1) / decimal.Decimal(7)
Decimal("0.1428571428571428571428571429")
>>> decimal.getcontext().prec = 9
>>> decimal.Decimal(1) / decimal.Decimal(7)
Decimal("0.142857143")
```

The default action for error conditions is selectable; the module can either return a special value such as infinity or not-a-number, or exceptions can be raised:

```
>>> decimal.Decimal(1) / decimal.Decimal(0)
Traceback (most recent call last):
  ...
decimal.DivisionByZero: x / 0
>>> decimal.getcontext().traps[decimal.DivisionByZero] = False
>>> decimal.Decimal(1) / decimal.Decimal(0)
Decimal("Infinity")
>>>
```

The **`Context`** instance also has various methods for formatting numbers such as **`to_eng_string()`** and **`to_sci_string()`**.

For more information, see the documentation for the **`decimal`** module, which includes a quick-start tutorial and a reference.

See also

PEP 327 [https://peps.python.org/pep-0327/] - Decimal Data Type

Written by Facundo Batista and implemented by Facundo Batista, Eric Price, Raymond Hettinger, Aahz, and Tim Peters.

http://www.lahey.com/float.htm

[https://web.archive.org/web/20230604072523/http://www.lahey.com/float.htm]

The article uses Fortran code to illustrate many of the problems that floating-point inaccuracy can cause.

https://speleotrove.com/decimal/

A description of a decimal-based representation. This representation is being proposed as a standard, and underlies the new Python decimal type. Much of this material was written by Mike Cowlishaw, designer of the Rexx language.

PEP 328: Multi-line Imports

One language change is a small syntactic tweak aimed at making it easier to import many names from a module. In a `from module import names` statement, *names* is a sequence of names separated by commas. If the sequence is very long, you can either write multiple imports from the same module, or you can use backslashes to escape the line endings like this:

```
from SimpleXMLRPCServer import SimpleXMLRPCServer,\
            SimpleXMLRPCRequestHandler,\
            CGIXMLRPCRequestHandler,\
            resolve_dotted_attribute
```

The syntactic change in Python 2.4 simply allows putting the names within parentheses. Python ignores newlines within a parenthesized expression, so the backslashes are no longer needed:

```
from SimpleXMLRPCServer import (SimpleXMLRPCServer,
                                SimpleXMLRPCRequestHandler,
                                CGIXMLRPCRequestHandler,
                                resolve_dotted_attribute)
```

The PEP also proposes that all `import` statements be absolute imports, with a leading `.` character to indicate a relative import. This part of the PEP was not implemented for Python 2.4, but was completed for Python 2.5.

See also

PEP 328 [https://peps.python.org/pep-0328/] - Imports: Multi-Line and Absolute/Relative
Written by Aahz. Multi-line imports were implemented by Dima Dorfman.

PEP 331: Locale-Independent Float/String Conversions

The `locale` modules lets Python software select various conversions and display conventions that are localized to a particular country or language. However, the module was careful to not change the numeric locale because various functions in Python’s implementation required that the numeric locale remain set to the `'C'` locale. Often this was because the code was using the C library’s **`atof()`** function.

Not setting the numeric locale caused trouble for extensions that used third-party C libraries, however, because they wouldn’t have the correct locale set. The motivating example was GTK+, whose user interface widgets weren’t displaying numbers in the current locale.

The solution described in the PEP is to add three new functions to the Python API that perform ASCII-only conversions, ignoring the locale setting:

- `PyOS_ascii_strtod(str, ptr)` and `PyOS_ascii_atof(str, ptr)` both convert a string to a C double.
- `PyOS_ascii_formatd(buffer, buf_len, format, d)` converts a double to an ASCII string.

The code for these functions came from the GLib library (https://developer-old.gnome.org/glib/2.26/ [http://web.archive.org/web/20210306104320/https://developer.gnome.org/glib/2.26/]), whose developers kindly relicensed the relevant functions and donated them to the Python Software Foundation. The **`locale`** module can now change the numeric locale, letting extensions such as GTK+ produce the correct results.

See also

PEP 331 [https://peps.python.org/pep-0331/] - Locale-Independent Float/String Conversions
Written by Christian R. Reis, and implemented by Gustavo Carneiro.

Other Language Changes

Here are all of the changes that Python 2.4 makes to the core Python language.

- Decorators for functions and methods were added (**PEP 318** [https://peps.python.org/pep-0318/]).

- Built-in **`set()`** and **`frozenset()`** types were added (**PEP 218** [https://peps.python.org/pep-0218/]). Other new built-ins include the `reversed(seq)` function (**PEP 322** [https://peps.python.org/pep-0322/]).

- Generator expressions were added (**PEP 289** [https://peps.python.org/pep-0289/]).
- Certain numeric expressions no longer return values restricted to 32 or 64 bits (**PEP 237** [https://peps.python.org/pep-0237/]).
- You can now put parentheses around the list of names in a `from module import names` statement (**PEP 328** [https://peps.python.org/pep-0328/]).
- The **`dict.update()`** method now accepts the same argument forms as the **`dict`** constructor. This includes any mapping, any iterable of key/value pairs, and keyword arguments. (Contributed by Raymond Hettinger.)
- The string methods **`ljust()`**, **`rjust()`**, and **`center()`** now take an optional argument for specifying a fill character other than a space. (Contributed by Raymond Hettinger.)
- Strings also gained an **`rsplit()`** method that works like the **`split()`** method but splits from the end of the string. (Contributed by Sean Reifschneider.)

  ```
  >>> 'www.python.org'.split('.', 1)
  ['www', 'python.org']
  'www.python.org'.rsplit('.', 1)
  ['www.python', 'org']
  ```

- Three keyword parameters, *cmp*, *key*, and *reverse*, were added to the **`sort()`** method of lists. These parameters make some common usages of **`sort()`** simpler. All of these parameters are optional.

For the *cmp* parameter, the value should be a comparison function that takes two parameters and returns -1, 0, or +1 depending on how the parameters compare. This function will then be used to sort the list. Previously this was the only parameter that could be provided to **`sort()`**.

key should be a single-parameter function that takes a list element and returns a comparison key for the element. The list is then sorted using the comparison keys. The following example sorts a list case-insensitively:

```
>>> L = ['A', 'b', 'c', 'D']
>>> L.sort()                 # Case-sensitive sort
>>> L
['A', 'D', 'b', 'c']
>>> # Using 'key' parameter to sort list
>>> L.sort(key=lambda x: x.lower())
>>> L
['A', 'b', 'c', 'D']
>>> # Old-fashioned way
>>> L.sort(cmp=lambda x,y: cmp(x.lower(), y.lower()))
>>> L
['A', 'b', 'c', 'D']
```

The last example, which uses the *cmp* parameter, is the old way to perform a case-insensitive sort. It works but is slower than using a *key* parameter. Using *key* calls **`lower()`** method once for each element in the list while using *cmp* will call it twice for each comparison, so using *key* saves on invocations of the **`lower()`** method.

For simple key functions and comparison functions, it is often possible to avoid a **`lambda`** expression by using an unbound method instead. For example, the above case-insensitive sort is best written as:

```
>>> L.sort(key=str.lower)
>>> L
['A', 'b', 'c', 'D']
```

Finally, the *reverse* parameter takes a Boolean value. If the value is true, the list will be sorted into reverse order. Instead of `L.sort(); L.reverse()`, you can now write `L.sort(reverse=True)`.

The results of sorting are now guaranteed to be stable. This means that two entries with equal keys will be returned in the same order as they were input. For example, you can sort a list of people by name, and then sort the list by age, resulting in a list sorted by age where people with the same age are in name-sorted order.

(All changes to **`sort()`** contributed by Raymond Hettinger.)

- There is a new built-in function `sorted(iterable)` that works like the in-place `list.sort()` method but can be used in expressions. The differences are:
- the input may be any iterable;
- a newly formed copy is sorted, leaving the original intact; and
- the expression returns the new sorted copy

```
>>> L = [9,7,8,3,2,4,1,6,5]
>>> [10+i for i in sorted(L)]       # usable in a list
comprehension
[11, 12, 13, 14, 15, 16, 17, 18, 19]
>>> L                               # original is left
unchanged
[9,7,8,3,2,4,1,6,5]
>>> sorted('Monty Python')          # any iterable may be
```

```
an input
[' ', 'M', 'P', 'h', 'n', 'n', 'o', 'o', 't', 't', 'y',
'y']

>>> # List the contents of a dict sorted by key values
>>> colormap = dict(red=1, blue=2, green=3, black=4,
yellow=5)
>>> for k, v in sorted(colormap.iteritems()):
...     print k, v
...
black 4
blue 2
green 3
red 1
yellow 5
```

(Contributed by Raymond Hettinger.)

- Integer operations will no longer trigger an **`OverflowWarning`**. The **`OverflowWarning`** warning will disappear in Python 2.5.

- The interpreter gained a new switch, `-m`, that takes a name, searches for the corresponding module on `sys.path`, and runs the module as a script. For example, you can now run the Python profiler with `python -m profile`. (Contributed by Nick Coghlan.)

- The `eval(expr, globals, locals)` and `execfile(filename, globals, locals)` functions and the `exec` statement now accept any mapping type for the *locals* parameter. Previously this had to be a regular Python dictionary. (Contributed by Raymond Hettinger.)

- The `zip()` built-in function and **`itertools.izip()`** now return an empty list if called with no arguments. Previously they raised a `TypeError` exception. This makes them more suitable for use with variable length argument lists:

```
>>> def transpose(array):
...     return zip(*array)
...
>>> transpose([(1,2,3), (4,5,6)])
[(1, 4), (2, 5), (3, 6)]
>>> transpose([])
[]
```

(Contributed by Raymond Hettinger.)

- Encountering a failure while importing a module no longer leaves a partially initialized module object in `sys.modules`. The incomplete module object left behind would fool further imports of the same module into succeeding, leading to confusing errors. (Fixed by Tim Peters.)

- `None` is now a constant; code that binds a new value to the name `None` is now a syntax error. (Contributed by Raymond Hettinger.)

Optimizations

- The inner loops for list and tuple slicing were optimized and now run about one-third faster. The inner loops for dictionaries were also optimized, resulting in performance boosts for **`keys()`**, **`values()`**, **`items()`**, **`iterkeys()`**, **`itervalues()`**, and **`iteritems()`**. (Contributed by Raymond Hettinger.)
- The machinery for growing and shrinking lists was optimized for speed and for space efficiency. Appending and popping from lists now runs faster due to more efficient code paths and less frequent use of the underlying system **`realloc()`**. List comprehensions also benefit. `list.extend()` was also optimized and no longer converts its argument into a temporary list before

extending the base list. (Contributed by Raymond Hettinger.)

- `list()`, `tuple()`, `map()`, `filter()`, and `zip()` now run several times faster with non-sequence arguments that supply a **`__len__()`** method. (Contributed by Raymond Hettinger.)
- The methods **`list.__getitem__()`**, **`dict.__getitem__()`**, and **`dict.__contains__()`** are now implemented as **`method_descriptor`** objects rather than **`wrapper_descriptor`** objects. This form of access doubles their performance and makes them more suitable for use as arguments to functionals: `map(mydict.__getitem__, keylist)`. (Contributed by Raymond Hettinger.)
- Added a new opcode, `LIST_APPEND`, that simplifies the generated bytecode for list comprehensions and speeds them up by about a third. (Contributed by Raymond Hettinger.)
- The peephole bytecode optimizer has been improved to produce shorter, faster bytecode; remarkably, the resulting bytecode is more readable. (Enhanced by Raymond Hettinger.)
- String concatenations in statements of the form `s = s + "abc"` and `s += "abc"` are now performed more efficiently in certain circumstances. This optimization won’t be present in other Python implementations such as Jython, so you shouldn’t rely on it; using the **`join()`** method of strings is still recommended when you want to efficiently glue a large number of strings together. (Contributed by Armin Rigo.)

The net result of the 2.4 optimizations is that Python 2.4 runs the pystone benchmark around 5% faster than Python 2.3 and 35% faster than Python 2.2. (pystone is not a particularly good benchmark, but it’s the most commonly used measurement of Python’s performance. Your own applications may show greater or smaller benefits from Python 2.4.)

New, Improved, and Deprecated Modules

As usual, Python's standard library received a number of enhancements and bug fixes. Here's a partial list of the most notable changes, sorted alphabetically by module name. Consult the `Misc/NEWS` file in the source tree for a more complete list of changes, or look through the CVS logs for all the details.

- The **`asyncore`** module's **`loop()`** function now has a *count* parameter that lets you perform a limited number of passes through the polling loop. The default is still to loop forever.
- The **`base64`** module now has more complete **RFC 3548** [https://datatracker.ietf.org/doc/html/rfc3548.html] support for Base64, Base32, and Base16 encoding and decoding, including optional case folding and optional alternative alphabets. (Contributed by Barry Warsaw.)
- The **`bisect`** module now has an underlying C implementation for improved performance. (Contributed by Dmitry Vasiliev.)
- The CJKCodecs collections of East Asian codecs, maintained by Hye-Shik Chang, was integrated into 2.4. The new encodings are:
- Chinese (PRC): gb2312, gbk, gb18030, big5hkscs, hz
- Chinese (ROC): big5, cp950
- Japanese: cp932, euc-jis-2004, euc-jp, euc-jisx0213, iso-2022-jp,

  iso-2022-jp-1, iso-2022-jp-2, iso-2022-jp-3, iso-2022-jp-ext, iso-2022-jp-2004, shift-jis, shift-jisx0213, shift-jis-2004

- Korean: cp949, euc-kr, johab, iso-2022-kr
- Some other new encodings were added: HP Roman8, ISO_8859-11, ISO_8859-16, PCTP-154, and TIS-620.
- The UTF-8 and UTF-16 codecs now cope better with receiving partial input. Previously the **`StreamReader`** class would try to read more data, making it impossible to resume decoding from the stream. The **`read()`** method will now return as much data as it can and future calls will resume decoding where previous ones left off. (Implemented by Walter Dörwald.)
- There is a new `collections` module for various specialized collection datatypes. Currently it contains just one type, **`deque`**, a double-ended queue that supports efficiently adding and removing elements from either end:

```
>>> from collections import deque
>>> d = deque('ghi')        # make a new deque with three
items
>>> d.append('j')           # add a new entry to the right
side
>>> d.appendleft('f')       # add a new entry to the left
side
>>> d                       # show the representation of
the deque
deque(['f', 'g', 'h', 'i', 'j'])
>>> d.pop()                 # return and remove the
rightmost item
'j'
>>> d.popleft()             # return and remove the
leftmost item
'f'
>>> list(d)                 # list the contents of the
```

```
deque
['g', 'h', 'i']
>>> 'h' in d                # search the deque
True
```

Several modules, such as the **`Queue`** and [`threading`] modules, now take advantage of [`collections.deque`] for improved performance. (Contributed by Raymond Hettinger.)

- The `ConfigParser` classes have been enhanced slightly. The `read()` method now returns a list of the files that were successfully parsed, and the `set()` method raises `TypeError` if passed a *value* argument that isn't a string. (Contributed by John Belmonte and David Goodger.)

- The `curses` module now supports the ncurses extension **`use_default_colors()`**. On platforms where the terminal supports transparency, this makes it possible to use a transparent background. (Contributed by Jörg Lehmann.)

- The `difflib` module now includes an **`HtmlDiff`** class that creates an HTML table showing a side by side comparison of two versions of a text. (Contributed by Dan Gass.)

- The `email` package was updated to version 3.0, which dropped various deprecated APIs and removes support for Python versions earlier than 2.3. The 3.0 version of the package uses a new incremental parser for MIME messages, available in the **`email.FeedParser`** module. The new parser doesn't require reading the entire message into memory, and doesn't raise exceptions if a message is malformed; instead it records any problems in the **`defect`** attribute of the message. (Developed by Anthony Baxter, Barry Warsaw, Thomas Wouters, and others.)

- The **heapq** module has been converted to C. The resulting tenfold improvement in speed makes the module suitable for handling high volumes of data. In addition, the module has two new functions **`nlargest()`** and **`nsmallest()`** that use heaps to find the N largest or smallest values in a dataset without the expense of a full sort. (Contributed by Raymond Hettinger.)

- The **httplib** module now contains constants for HTTP status codes defined in various HTTP-related RFC documents. Constants have names such as **`OK`**, **`CREATED`**, **`CONTINUE`**, and **`MOVED_PERMANENTLY`**; use pydoc to get a full list. (Contributed by Andrew Eland.)

- The **imaplib** module now supports IMAP's THREAD command (contributed by Yves Dionne) and new **`deleteacl()`** and **`myrights()`** methods (contributed by Arnaud Mazin).

- The **itertools** module gained a `groupby(iterable[, *func*])` function. *iterable* is something that can be iterated over to return a stream of elements, and the optional *func* parameter is a function that takes an element and returns a key value; if omitted, the key is simply the element itself. **`groupby()`** then groups the elements into subsequences which have matching values of the key, and returns a series of 2-tuples containing the key value and an iterator over the subsequence.

  Here's an example to make this clearer. The *key* function simply returns whether a number is even or odd, so the result of **`groupby()`** is to return consecutive runs of odd or even numbers.

  ```
  >>> import itertools
  >>> L = [2, 4, 6, 7, 8, 9, 11, 12, 14]
  ```

```
>>> for key_val, it in itertools.groupby(L, lambda x: x %
2):
...     print key_val, list(it)
...
0 [2, 4, 6]
1 [7]
0 [8]
1 [9, 11]
0 [12, 14]
>>>
```

`groupby()` is typically used with sorted input. The logic for **`groupby()`** is similar to the Unix `uniq` filter which makes it handy for eliminating, counting, or identifying duplicate elements:

```
>>> word = 'abracadabra'
>>> letters = sorted(word)   # Turn string into a sorted
list of letters
>>> letters
['a', 'a', 'a', 'a', 'a', 'b', 'b', 'c', 'd', 'r', 'r']
>>> for k, g in itertools.groupby(letters):
...     print k, list(g)
...
a ['a', 'a', 'a', 'a', 'a']
b ['b', 'b']
c ['c']
d ['d']
r ['r', 'r']
>>> # List unique letters
>>> [k for k, g in groupby(letters)]
['a', 'b', 'c', 'd', 'r']
>>> # Count letter occurrences
>>> [(k, len(list(g))) for k, g in groupby(letters)]
[('a', 5), ('b', 2), ('c', 1), ('d', 1), ('r', 2)]
```

(Contributed by Hye-Shik Chang.)

- **`itertools`** also gained a function named `tee(iterator, N)` that returns *N* independent iterators that replicate *iterator*. If *N* is omitted, the default is 2.

```
>>> L = [1,2,3]
>>> i1, i2 = itertools.tee(L)
>>> i1,i2
(<itertools.tee object at 0x402c2080>, <itertools.tee
object at 0x402c2090>)
>>> list(i1)               # Run the first iterator to
exhaustion
[1, 2, 3]
>>> list(i2)               # Run the second iterator to
exhaustion
[1, 2, 3]
```

Note that **`tee()`** has to keep copies of the values returned by the iterator; in the worst case, it may need to keep all of them. This should therefore be used carefully if the leading iterator can run far ahead of the trailing iterator in a long stream of inputs. If the separation is large, then you might as well use **`list()`** instead. When the iterators track closely with one another, **`tee()`** is ideal. Possible applications include bookmarking, windowing, or lookahead iterators. (Contributed by Raymond Hettinger.)

- A number of functions were added to the **`locale`** module, such as **`bind_textdomain_codeset()`** to specify a particular encoding and a family of **`l*gettext()`** functions that return messages in the chosen encoding. (Contributed by Gustavo Niemeyer.)

- Some keyword arguments were added to the **`logging`** package’s **`basicConfig()`** function to simplify log configuration. The default behavior is to log messages to standard error, but various keyword arguments can be specified to log to a particular file, change the logging format, or set the logging level. For example:

```
import logging
logging.basicConfig(filename='/var/log/application.log',
    level=0,  # Log all messages
    format='%(levelname):%(process):%(thread):%(message)')
```

Other additions to the `logging` package include a `log(level, msg)` convenience method, as well as a **`TimedRotatingFileHandler`** class that rotates its log files at a timed interval. The module already had **`RotatingFileHandler`**, which rotated logs once the file exceeded a certain size. Both classes derive from a new **`BaseRotatingHandler`** class that can be used to implement other rotating handlers.

(Changes implemented by Vinay Sajip.)

- The `marshal` module now shares interned strings on unpacking a data structure. This may shrink the size of certain pickle strings, but the primary effect is to make `.pyc` files significantly smaller. (Contributed by Martin von Löwis.)

- The **`nntplib`** module’s **`NNTP`** class gained **`description()`** and **`descriptions()`** methods to retrieve newsgroup descriptions for a single group or for a range of groups. (Contributed by Jürgen A. Erhard.)

- Two new functions were added to the `operator` module, `attrgetter(attr)` and `itemgetter(index)`. Both functions return callables that take a single argument and return the corresponding attribute or item; these callables make excellent data extractors when used with `map()` or `sorted()`. For example:

```
>>> L = [('c', 2), ('d', 1), ('a', 4), ('b', 3)]
>>> map(operator.itemgetter(0), L)
['c', 'd', 'a', 'b']
>>> map(operator.itemgetter(1), L)
[2, 1, 4, 3]
>>> sorted(L, key=operator.itemgetter(1)) # Sort list by
```

```
second tuple item
[('d', 1), ('c', 2), ('b', 3), ('a', 4)]
```

(Contributed by Raymond Hettinger.)

- The **`optparse`** module was updated in various ways. The module now passes its messages through **`gettext.gettext()`**, making it possible to internationalize Optik's help and error messages. Help messages for options can now include the string `'%default'`, which will be replaced by the option's default value. (Contributed by Greg Ward.)

- The long-term plan is to deprecate the **`rfc822`** module in some future Python release in favor of the **`email`** package. To this end, the **`email.Utils.formatdate`** function has been changed to make it usable as a replacement for **`rfc822.formatdate()`**. You may want to write new e-mail processing code with this in mind. (Change implemented by Anthony Baxter.)

- A new `urandom(n)` function was added to the **`os`** module, returning a string containing *n* bytes of random data. This function provides access to platform-specific sources of randomness such as `/dev/urandom` on Linux or the Windows CryptoAPI. (Contributed by Trevor Perrin.)

- Another new function: `os.path.lexists(path)` returns true if the file specified by *path* exists, whether or not it's a symbolic link. This differs from the existing `os.path.exists(path)` function, which returns false if *path* is a symlink that points to a destination that doesn't exist. (Contributed by Beni Cherniavsky.)

- A new **`getsid()`** function was added to the **`posix`** module that underlies the **`os`** module. (Contributed by J. Raynor.)

- The `poplib` module now supports POP over SSL. (Contributed by Hector Urtubia.)

- The `profile` module can now profile C extension functions. (Contributed by Nick Bastin.)

- The `random` module has a new method called `getrandbits(N)` that returns a long integer *N* bits in length. The existing **`randrange()`** method now uses **`getrandbits()`** where appropriate, making generation of arbitrarily large random numbers more efficient. (Contributed by Raymond Hettinger.)

- The regular expression language accepted by the `re` module was extended with simple conditional expressions, written as `(?(group)A|B)`. *group* is either a numeric group ID or a group name defined with `(?P<group>...)` earlier in the expression. If the specified group matched, the regular expression pattern *A* will be tested against the string; if the group didn't match, the pattern *B* will be used instead. (Contributed by Gustavo Niemeyer.)

- The `re` module is also no longer recursive, thanks to a massive amount of work by Gustavo Niemeyer. In a recursive regular expression engine, certain patterns result in a large amount of C stack space being consumed, and it was possible to overflow the stack. For example, if you matched a 30000-byte string of `a` characters against the expression `(a|b)+`, one stack frame was consumed per character. Python 2.3 tried to check for stack overflow and raise a `RuntimeError` exception, but certain patterns could sidestep the checking and if you were unlucky Python could segfault. Python 2.4's regular expression engine can match this pattern without problems.

- The `signal` module now performs tighter error-checking on the parameters to the `signal.signal()` function. For example, you can't set a handler on the **`SIGKILL`** signal; previous versions of Python would quietly accept this, but 2.4 will raise a `RuntimeError` exception.

- Two new functions were added to the `socket` module. **`socketpair()`** returns a pair of connected sockets and `getservbyport(port)` looks up the service name for a given port number. (Contributed by Dave Cole and Barry Warsaw.)

- The **`sys.exitfunc()`** function has been deprecated. Code should be using the existing `atexit` module, which correctly handles calling multiple exit functions. Eventually **`sys.exitfunc()`** will become a purely internal interface, accessed only by `atexit`.

- The `tarfile` module now generates GNU-format tar files by default. (Contributed by Lars Gustäbel.)

- The `threading` module now has an elegantly simple way to support thread-local data. The module contains a **`local`** class whose attribute values are local to different threads.

  ```
  import threading

  data = threading.local()
  data.number = 42
  data.url = ('www.python.org', 80)
  ```

  Other threads can assign and retrieve their own values for the **`number`** and **`url`** attributes. You can subclass **`local`** to initialize attributes or to add methods. (Contributed by Jim Fulton.)

- The `timeit` module now automatically disables periodic garbage collection during the timing loop. This change makes consecutive timings more comparable. (Contributed by Raymond Hettinger.)
- The `weakref` module now supports a wider variety of objects including Python functions, class instances, sets, frozensets, deques, arrays, files, sockets, and regular expression pattern objects. (Contributed by Raymond Hettinger.)
- The `xmlrpclib` module now supports a multi-call extension for transmitting multiple XML-RPC calls in a single HTTP operation. (Contributed by Brian Quinlan.)
- The `mpz`, `rotor`, and `xreadlines` modules have been removed.

cookielib

The `cookielib` library supports client-side handling for HTTP cookies, mirroring the `Cookie` module’s server-side cookie support. Cookies are stored in cookie jars; the library transparently stores cookies offered by the web server in the cookie jar, and fetches the cookie from the jar when connecting to the server. As in web browsers, policy objects control whether cookies are accepted or not.

In order to store cookies across sessions, two implementations of cookie jars are provided: one that stores cookies in the Netscape format so applications can use the Mozilla or Lynx cookie files, and one that stores cookies in the same format as the Perl libwww library.

`urllib2` has been changed to interact with `cookielib`: `HTTPCookieProcessor` manages a cookie jar that is used when accessing URLs.

This module was contributed by John J. Lee.

doctest

The `doctest` module underwent considerable refactoring thanks to Edward Loper and Tim Peters. Testing can still be as simple as running `doctest.testmod()`, but the refactorings allow customizing the module's operation in various ways

The new **`DocTestFinder`** class extracts the tests from a given object's docstrings:

```
def f (x, y):
    """>>> f(2,2)
4
>>> f(3,2)
6
    """
    return x*y

finder = doctest.DocTestFinder()

# Get list of DocTest instances
tests = finder.find(f)
```

The new **`DocTestRunner`** class then runs individual tests and can produce a summary of the results:

```
runner = doctest.DocTestRunner()
for t in tests:
    tried, failed = runner.run(t)

runner.summarize(verbose=1)
```

The above example produces the following output:

```
1 items passed all tests:
   2 tests in f
```

```
2 tests in 1 items.
2 passed and 0 failed.
Test passed.
```

`DocTestRunner` uses an instance of the **`OutputChecker`** class to compare the expected output with the actual output. This class takes a number of different flags that customize its behaviour; ambitious users can also write a completely new subclass of **`OutputChecker`**.

The default output checker provides a number of handy features. For example, with the **`doctest.ELLIPSIS`** option flag, an ellipsis (`...`) in the expected output matches any substring, making it easier to accommodate outputs that vary in minor ways:

```
def o (n):
    """>>> o(1)
<__main__.C instance at 0x...>
>>>
"""
```

Another special string, `<BLANKLINE>`, matches a blank line:

```
def p (n):
    """>>> p(1)
<BLANKLINE>
>>>
"""
```

Another new capability is producing a diff-style display of the output by specifying the **`doctest.REPORT_UDIFF`** (unified diffs), **`doctest.REPORT_CDIFF`** (context diffs), or **`doctest.REPORT_NDIFF`** (delta-style) option flags. For example:

```
def g (n):
    """>>> g(4)
here
is
a
```

```
lengthy
>>>"""
    L = 'here is a rather lengthy list of words'.split()
    for word in L[:n]:
        print word
```

Running the above function’s tests with [`doctest.REPORT_UDIFF`](#) specified, you get the following output:

```
**************************************************************
*******
File "t.py", line 15, in g
Failed example:
    g(4)
Differences (unified diff with -expected +actual):
    @@ -2,3 +2,3 @@
     is
     a
    -lengthy
    +rather
**************************************************************
*******
```

Build and C API Changes

Some of the changes to Python’s build process and to the C API are:

- Three new convenience macros were added for common return values from extension functions: **`Py_RETURN_NONE`**, **`Py_RETURN_TRUE`**, and **`Py_RETURN_FALSE`**. (Contributed by Brett Cannon.)
- Another new macro, **`Py_CLEAR`**, decreases the reference count of *obj* and sets *obj* to the null pointer. (Contributed by Jim Fulton.)
- A new function, `PyTuple_Pack(N, obj1, obj2, ..., objN)`, constructs tuples from a variable length argument list of Python objects. (Contributed by Raymond Hettinger.)

- A new function, `PyDict_Contains(d, k)`, implements fast dictionary lookups without masking exceptions raised during the look-up process. (Contributed by Raymond Hettinger.)
- The Py_IS_NAN(X) macro returns 1 if its float or double argument *X* is a NaN. (Contributed by Tim Peters.)
- C code can avoid unnecessary locking by using the new **`PyEval_ThreadsInitialized()`** function to tell if any thread operations have been performed. If this function returns false, no lock operations are needed. (Contributed by Nick Coghlan.)
- A new function, **`PyArg_VaParseTupleAndKeywords()`**, is the same as **`PyArg_ParseTupleAndKeywords()`** but takes a **`va_list`** instead of a number of arguments. (Contributed by Greg Chapman.)
- A new method flag, **`METH_COEXIST`**, allows a function defined in slots to co-exist with a **`PyCFunction`** having the same name. This can halve the access time for a method such as **`set.__contains__()`**. (Contributed by Raymond Hettinger.)
- Python can now be built with additional profiling for the interpreter itself, intended as an aid to people developing the Python core. Providing **`--enable-profiling`** to the **configure** script will let you profile the interpreter with **gprof**, and providing the **`--with-tsc`** switch enables profiling using the Pentium’s Time-Stamp-Counter register. Note that the **`--with-tsc`** switch is slightly misnamed, because the profiling feature also works on the PowerPC platform, though that processor architecture doesn’t call that register “the TSC register”. (Contributed by Jeremy Hylton.)
- The **`tracebackobject`** type has been renamed to **`PyTracebackObject`**.

Port-Specific Changes

- The Windows port now builds under MSVC++ 7.1 as well as version 6. (Contributed by Martin von Löwis.)

Porting to Python 2.4

This section lists previously described changes that may require changes to your code:

- Left shifts and hexadecimal/octal constants that are too large no longer trigger a `FutureWarning` and return a value limited to 32 or 64 bits; instead they return a long integer.
- Integer operations will no longer trigger an `OverflowWarning`. The `OverflowWarning` warning will disappear in Python 2.5.
- The `zip()` built-in function and `itertools.izip()` now return an empty list instead of raising a `TypeError` exception if called with no arguments.
- You can no longer compare the `date` and `datetime` instances provided by the `datetime` module. Two instances of different classes will now always be unequal, and relative comparisons (`<`, `>`) will raise a `TypeError`.
- `dircache.listdir()` now passes exceptions to the caller instead of returning empty lists.
- `LexicalHandler.startDTD()` used to receive the public and system IDs in the wrong order. This has been corrected; applications relying on the wrong order need to be fixed.
- `fcntl.ioctl()` now warns if the *mutate* argument is omitted and relevant.
- The `tarfile` module now generates GNU-format tar files by default.
- Encountering a failure while importing a module no longer leaves a partially initialized module object in `sys.modules`.

- `None` is now a constant; code that binds a new value to the name `None` is now a syntax error.
- The **`signals.signal()`** function now raises a `RuntimeError` exception for certain illegal values; previously these errors would pass silently. For example, you can no longer set a handler on the **`SIGKILL`** signal.

Acknowledgements

The author would like to thank the following people for offering suggestions, corrections and assistance with various drafts of this article: Koray Can, Hye-Shik Chang, Michael Dyck, Raymond Hettinger, Brian Hurt, Hamish Lawson, Fredrik Lundh, Sean Reifschneider, Sadruddin Rejeb.

What’s New in Python 2.3

Author:: A.M. Kuchling

This article explains the new features in Python 2.3. Python 2.3 was released on July 29, 2003.

The main themes for Python 2.3 are polishing some of the features added in 2.2, adding various small but useful enhancements to the core language, and expanding the standard library. The new object model introduced in the previous version has benefited from 18 months of bugfixes and from optimization efforts that have improved the performance of new-style classes. A few new built-in functions have been added such as `sum()` and `enumerate()`. The `in` operator can now be used for substring searches (e.g. `"ab" in "abc"` returns `True`).

Some of the many new library features include Boolean, set, heap, and date/time data types, the ability to import modules from ZIP-format archives, metadata support for the long-awaited Python catalog, an updated version of IDLE, and modules for logging messages, wrapping text, parsing CSV files, processing command-line options, using BerkeleyDB databases… the list of new and enhanced modules is lengthy.

This article doesn’t attempt to provide a complete specification of the new features, but instead provides a convenient overview. For full details, you should refer to the documentation for Python 2.3, such as the Python Library Reference and the Python Reference Manual. If you want to understand the complete implementation and design rationale, refer to the PEP for a particular new feature.

PEP 218: A Standard Set Datatype

The new **`sets`** module contains an implementation of a set datatype. The **`Set`** class is for mutable sets, sets that can have members added and removed. The **`ImmutableSet`** class is for sets that can’t be modified, and instances of **`ImmutableSet`** can therefore be used as dictionary keys. Sets are built on top of dictionaries, so the elements within a set must be hashable.

Here’s a simple example:

```
>>> import sets
>>> S = sets.Set([1,2,3])
>>> S
Set([1, 2, 3])
>>> 1 in S
True
>>> 0 in S
False
>>> S.add(5)
>>> S.remove(3)
>>> S
Set([1, 2, 5])
>>>
```

The union and intersection of sets can be computed with the **`union()`** and **`intersection()`** methods; an alternative notation uses the bitwise operators `&` and `|`. Mutable sets also have in-place versions of these methods, **`union_update()`** and **`intersection_update()`**.

```
>>> S1 = sets.Set([1,2,3])
>>> S2 = sets.Set([4,5,6])
>>> S1.union(S2)
Set([1, 2, 3, 4, 5, 6])
>>> S1 | S2                  # Alternative notation
Set([1, 2, 3, 4, 5, 6])
>>> S1.intersection(S2)
```

```
Set([])
>>> S1 & S2                  # Alternative notation
Set([])
>>> S1.union_update(S2)
>>> S1
Set([1, 2, 3, 4, 5, 6])
>>>
```

It’s also possible to take the symmetric difference of two sets. This is the set of all elements in the union that aren’t in the intersection. Another way of putting it is that the symmetric difference contains all elements that are in exactly one set. Again, there’s an alternative notation (^), and an in-place version with the ungainly name `symmetric_difference_update()`.

```
>>> S1 = sets.Set([1,2,3,4])
>>> S2 = sets.Set([3,4,5,6])
>>> S1.symmetric_difference(S2)
Set([1, 2, 5, 6])
>>> S1 ^ S2
Set([1, 2, 5, 6])
>>>
```

There are also **`issubset()`** and **`issuperset()`** methods for checking whether one set is a subset or superset of another:

```
>>> S1 = sets.Set([1,2,3])
>>> S2 = sets.Set([2,3])
>>> S2.issubset(S1)
True
>>> S1.issubset(S2)
False
>>> S1.issuperset(S2)
True
>>>
```

See also

PEP 218 [https://peps.python.org/pep-0218/] - Adding a Built-In Set Object Type

PEP written by Greg V. Wilson. Implemented by Greg V. Wilson, Alex Martelli, and GvR.

PEP 255: Simple Generators

In Python 2.2, generators were added as an optional feature, to be enabled by a `from __future__ import generators` directive. In 2.3 generators no longer need to be specially enabled, and are now always present; this means that `yield` is now always a keyword. The rest of this section is a copy of the description of generators from the “What’s New in Python 2.2” document; if you read it back when Python 2.2 came out, you can skip the rest of this section.

You’re doubtless familiar with how function calls work in Python or C. When you call a function, it gets a private namespace where its local variables are created. When the function reaches a `return` statement, the local variables are destroyed and the resulting value is returned to the caller. A later call to the same function will get a fresh new set of local variables. But, what if the local variables weren’t thrown away on exiting a function? What if you could later resume the function where it left off? This is what generators provide; they can be thought of as resumable functions.

Here’s the simplest example of a generator function:

```
def generate_ints(N):
    for i in range(N):
        yield i
```

A new keyword, `yield`, was introduced for generators. Any function containing a **`yield`** statement is a generator function; this is detected by Python's bytecode compiler which compiles the function specially as a result.

When you call a generator function, it doesn't return a single value; instead it returns a generator object that supports the iterator protocol. On executing the `yield` statement, the generator outputs the value of `i`, similar to a `return` statement. The big difference between **`yield`** and a **`return`** statement is that on reaching a **`yield`** the generator's state of execution is suspended and local variables are preserved. On the next call to the generator's `.next()` method, the function will resume executing immediately after the **`yield`** statement. (For complicated reasons, the **`yield`** statement isn't allowed inside the `try` block of a **`try`**... **`finally`** statement; read **PEP 255** [https://peps.python.org/pep-0255/] for a full explanation of the interaction between **`yield`** and exceptions.)

Here's a sample usage of the **`generate_ints()`** generator:

```
>>> gen = generate_ints(3)
>>> gen
<generator object at 0x8117f90>
>>> gen.next()
0
>>> gen.next()
1
>>> gen.next()
2
>>> gen.next()
Traceback (most recent call last):
  File "stdin", line 1, in ?
  File "stdin", line 2, in generate_ints
StopIteration
```

You could equally write `for i in generate_ints(5)`, or `a,b,c = generate_ints(3)`.

Inside a generator function, the `return` statement can only be used without a value, and signals the end of the procession of values; afterwards the generator cannot return any further values. **`return`** with a value, such as `return 5`, is a syntax error inside a generator function. The end of the generator's results can also be indicated by raising `StopIteration` manually, or by just letting the flow of execution fall off the bottom of the function.

You could achieve the effect of generators manually by writing your own class and storing all the local variables of the generator as instance variables. For example, returning a list of integers could be done by setting `self.count` to 0, and having the `next()` method increment `self.count` and return it. However, for a moderately complicated generator, writing a corresponding class would be much messier. `Lib/test/test_generators.py` contains a number of more interesting examples. The simplest one implements an in-order traversal of a tree using generators recursively.

```
# A recursive generator that generates Tree leaves in in-order.
def inorder(t):
    if t:
        for x in inorder(t.left):
            yield x
        yield t.label
        for x in inorder(t.right):
            yield x
```

Two other examples in `Lib/test/test_generators.py` produce solutions for the N-Queens problem (placing N queens on an NxN chess board so that no queen threatens another) and the Knight's Tour (a route that takes a knight to every square of an NxN chessboard without visiting any square twice).

The idea of generators comes from other programming languages, especially Icon (https://www2.cs.arizona.edu/icon/), where the idea of generators is central. In Icon, every expression and function call behaves like a generator. One example from “An Overview of the Icon Programming Language” at https://www2.cs.arizona.edu/icon/docs/ipd266.htm gives an idea of what this looks like:

```
sentence := "Store it in the neighboring harbor"
if (i := find("or", sentence)) > 5 then write(i)
```

In Icon the **`find()`** function returns the indexes at which the substring “or” is found: 3, 23, 33. In the **`if`** statement, `i` is first assigned a value of 3, but 3 is less than 5, so the comparison fails, and Icon retries it with the second value of 23. 23 is greater than 5, so the comparison now succeeds, and the code prints the value 23 to the screen.

Python doesn’t go nearly as far as Icon in adopting generators as a central concept. Generators are considered part of the core Python language, but learning or using them isn’t compulsory; if they don’t solve any problems that you have, feel free to ignore them. One novel feature of Python’s interface as compared to Icon’s is that a generator’s state is represented as a concrete object (the iterator) that can be passed around to other functions or stored in a data structure.

See also

PEP 255 [https://peps.python.org/pep-0255/] - Simple Generators

Written by Neil Schemenauer, Tim Peters, Magnus Lie Hetland. Implemented mostly by Neil Schemenauer and Tim Peters, with other fixes from the Python Labs crew.

PEP 263: Source Code Encodings

Python source files can now be declared as being in different character set encodings. Encodings are declared by including a specially formatted comment in the first or second line of the source file. For example, a UTF-8 file can be declared with:

```
#!/usr/bin/env python
# -*- coding: UTF-8 -*-
```

Without such an encoding declaration, the default encoding used is 7-bit ASCII. Executing or importing modules that contain string literals with 8-bit characters and have no encoding declaration will result in a `DeprecationWarning` being signalled by Python 2.3; in 2.4 this will be a syntax error.

The encoding declaration only affects Unicode string literals, which will be converted to Unicode using the specified encoding. Note that Python identifiers are still restricted to ASCII characters, so you can’t have variable names that use characters outside of the usual alphanumerics.

See also

PEP 263 [https://peps.python.org/pep-0263/] - Defining Python Source Code Encodings

Written by Marc-André Lemburg and Martin von Löwis; implemented by Suzuki Hisao and Martin von Löwis.

PEP 273: Importing Modules from ZIP Archives

The new `zipimport` module adds support for importing modules from a ZIP-format archive. You don’t need to import the module explicitly; it will be automatically imported if a ZIP archive’s filename is added to `sys.path`. For example:

```
amk@nyman:~/src/python$ unzip -l /tmp/example.zip
Archive:  /tmp/example.zip
  Length     Date   Time    Name
 --------    ----   ----    ----
     8467  11-26-02 22:30   jwzthreading.py
 --------                   -------
     8467                   1 file
amk@nyman:~/src/python$ ./python
Python 2.3 (#1, Aug 1 2003, 19:54:32)
>>> import sys
>>> sys.path.insert(0, '/tmp/example.zip')  # Add .zip file to
front of path
>>> import jwzthreading
>>> jwzthreading.__file__
'/tmp/example.zip/jwzthreading.py'
>>>
```

An entry in `sys.path` can now be the filename of a ZIP archive. The ZIP archive can contain any kind of files, but only files named `*.py`, `*.pyc`, or `*.pyo` can be imported. If an archive only contains `*.py` files, Python will not attempt to modify the archive by adding the corresponding `*.pyc` file, meaning that if a ZIP archive doesn’t contain `*.pyc` files, importing may be rather slow.

A path within the archive can also be specified to only import from a subdirectory; for example, the path `/tmp/example.zip/lib/` would only import from the `lib/` subdirectory within the archive.

See also

PEP 273 [https://peps.python.org/pep-0273/] - Import Modules from Zip Archives

> Written by James C. Ahlstrom, who also provided an implementation. Python 2.3 follows the specification in **PEP 273** [https://peps.python.org/pep-0273/], but uses an implementation written by Just van Rossum that uses the import hooks described in **PEP 302** [https://peps.python.org/pep-0302/]. See section PEP 302: New Import Hooks for a description of the new import hooks.

PEP 277: Unicode file name support for Windows NT

On Windows NT, 2000, and XP, the system stores file names as Unicode strings. Traditionally, Python has represented file names as byte strings, which is inadequate because it renders some file names inaccessible.

Python now allows using arbitrary Unicode strings (within the limitations of the file system) for all functions that expect file names, most notably the `open()` built-in function. If a Unicode string is passed to `os.listdir()`, Python now returns a list of Unicode strings. A new function, **`os.getcwdu()`**, returns the current directory as a Unicode string.

Byte strings still work as file names, and on Windows Python will transparently convert them to Unicode using the `mbcs` encoding.

Other systems also allow Unicode strings as file names but convert them to byte strings before passing them to the system, which can cause a `UnicodeError` to be raised. Applications can test whether arbitrary Unicode strings are supported as file names by checking `os.path.supports_unicode_filenames`, a Boolean value.

Under MacOS, `os.listdir()` may now return Unicode filenames.

See also

PEP 277 [https://peps.python.org/pep-0277/] - Unicode file name support for Windows NT

Written by Neil Hodgson; implemented by Neil Hodgson, Martin von Löwis, and Mark Hammond.

PEP 278: Universal Newline Support

The three major operating systems used today are Microsoft Windows, Apple’s Macintosh OS, and the various Unix derivatives. A minor irritation of cross-platform work is that these three platforms all use different characters to mark the ends of lines in text files. Unix uses the linefeed (ASCII character 10), MacOS uses the carriage return (ASCII character 13), and Windows uses a two-character sequence of a carriage return plus a newline.

Python’s file objects can now support end of line conventions other than the one followed by the platform on which Python is running. Opening a file with the mode `'U'` or `'rU'` will open a file for reading in universal newlines mode. All three line ending conventions will be translated to a `'\n'` in the strings returned by the various file methods such as **`read()`** and **`readline()`**.

Universal newline support is also used when importing modules and when executing a file with the **`execfile()`** function. This means that Python modules can be shared between all three operating systems without needing to convert the line-endings.

This feature can be disabled when compiling Python by specifying the **`--without-universal-newlines`** switch when running Python’s **configure** script.

See also

PEP 278 [https://peps.python.org/pep-0278/] - Universal Newline Support
Written and implemented by Jack Jansen.

PEP 279: enumerate()

A new built-in function, `enumerate()`, will make certain loops a bit clearer. `enumerate(thing)`, where *thing* is either an iterator or a sequence, returns an iterator that will return `(0, thing[0])`, `(1, thing[1])`, `(2, thing[2])`, and so forth.

A common idiom to change every element of a list looks like this:

```
for i in range(len(L)):
    item = L[i]
    # ... compute some result based on item ...
    L[i] = result
```

This can be rewritten using `enumerate()` as:

```
for i, item in enumerate(L):
    # ... compute some result based on item ...
    L[i] = result
```

See also

PEP 279 [https://peps.python.org/pep-0279/] - The enumerate() built-in function

Written and implemented by Raymond D. Hettinger.

PEP 282: The logging Package

A standard package for writing logs, **`logging`**, has been added to Python 2.3. It provides a powerful and flexible mechanism for generating logging output which can then be filtered and processed in various ways. A configuration file written in a standard format can be used to control the logging behavior of a program. Python includes handlers that will write log records to standard error or to a file or socket, send them to the system log, or even e-mail them to a particular address; of course, it’s also possible to write your own handler classes.

The **`Logger`** class is the primary class. Most application code will deal with one or more **`Logger`** objects, each one used by a particular subsystem of the application. Each **`Logger`** is identified by a name, and names are organized into a hierarchy using `.` as the component separator. For example, you might have **`Logger`** instances named `server`, `server.auth` and `server.network`. The latter two instances are below `server` in the hierarchy. This means that if you turn up the verbosity for `server` or direct `server` messages to a different handler, the changes will also apply to records logged to `server.auth` and `server.network`. There’s also a root **`Logger`** that’s the parent of all other loggers.

For simple uses, the **`logging`** package contains some convenience functions that always use the root log:

```
import logging

logging.debug('Debugging information')
logging.info('Informational message')
logging.warning('Warning:config file %s not found',
'server.conf')
logging.error('Error occurred')
logging.critical('Critical error -- shutting down')
```

This produces the following output:

```
WARNING:root:Warning:config file server.conf not found
ERROR:root:Error occurred
CRITICAL:root:Critical error -- shutting down
```

In the default configuration, informational and debugging messages are suppressed and the output is sent to standard error. You can enable the display of informational and debugging messages by calling the **`setLevel()`** method on the root logger.

Notice the **`warning()`** call's use of string formatting operators; all of the functions for logging messages take the arguments `(msg, arg1, arg2, ...)` and log the string resulting from `msg % (arg1, arg2, ...)`.

There's also an **`exception()`** function that records the most recent traceback. Any of the other functions will also record the traceback if you specify a true value for the keyword argument *exc_info*.

```
def f():
    try:    1/0
    except: logging.exception('Problem recorded')

f()
```

This produces the following output:

```
ERROR:root:Problem recorded
Traceback (most recent call last):
  File "t.py", line 6, in f
    1/0
ZeroDivisionError: integer division or modulo by zero
```

Slightly more advanced programs will use a logger other than the root logger. The `getLogger(name)` function is used to get a particular log, creating it if it doesn't exist yet. `getLogger(None)` returns the root logger.

```
log = logging.getLogger('server')
 ...
log.info('Listening on port %i', port)
 ...
log.critical('Disk full')
 ...
```

Log records are usually propagated up the hierarchy, so a message logged to `server.auth` is also seen by `server` and `root`, but a **`Logger`** can prevent this by setting its **`propagate`** attribute to **`False`**.

There are more classes provided by the **`logging`** package that can be customized. When a **`Logger`** instance is told to log a message, it creates a **`LogRecord`** instance that is sent to any number of different **`Handler`** instances. Loggers and handlers can also have an attached list of filters, and each filter can cause the **`LogRecord`** to be ignored or can modify the record before passing it along. When they're finally output, **`LogRecord`** instances are converted to text by a **`Formatter`** class. All of these classes can be replaced by your own specially written classes.

With all of these features the **`logging`** package should provide enough flexibility for even the most complicated applications. This is only an incomplete overview of its features, so please see the package's reference documentation for all of the details. Reading **PEP 282** [https://peps.python.org/pep-0282/] will also be helpful.

See also

PEP 282 [https://peps.python.org/pep-0282/] - A Logging System

Written by Vinay Sajip and Trent Mick; implemented by Vinay Sajip.

PEP 285: A Boolean Type

A Boolean type was added to Python 2.3. Two new constants were added to the `__builtin__` module, `True` and `False`. (`True` and `False` constants were added to the built-ins in Python 2.2.1, but the 2.2.1 versions are simply set to integer values of 1 and 0 and aren't a different type.)

The type object for this new type is named `bool`; the constructor for it takes any Python value and converts it to `True` or `False`.

```
>>> bool(1)
True
>>> bool(0)
False
>>> bool([])
False
>>> bool( (1,) )
True
```

Most of the standard library modules and built-in functions have been changed to return Booleans.

```
>>> obj = []
>>> hasattr(obj, 'append')
True
>>> isinstance(obj, list)
True
>>> isinstance(obj, tuple)
False
```

Python’s Booleans were added with the primary goal of making code clearer. For example, if you’re reading a function and encounter the statement `return 1`, you might wonder whether the `1` represents a Boolean truth value, an index, or a coefficient that multiplies some other quantity. If the statement is `return True`, however, the meaning of the return value is quite clear.

Python’s Booleans were *not* added for the sake of strict type-checking. A very strict language such as Pascal would also prevent you performing arithmetic with Booleans, and would require that the expression in an `if` statement always evaluate to a Boolean result. Python is not this strict and never will be, as **PEP 285** [https://peps.python.org/pep-0285/] explicitly says. This means you can still use any expression in an **`if`** statement, even ones that evaluate to a list or tuple or some random object. The Boolean type is a subclass of the `int` class so that arithmetic using a Boolean still works.

```
>>> True + 1
2
>>> False + 1
1
>>> False * 75
0
>>> True * 75
75
```

To sum up `True` and `False` in a sentence: they’re alternative ways to spell the integer values 1 and 0, with the single difference that `str()` and `repr()` return the strings `'True'` and `'False'` instead of `'1'` and `'0'`.

See also

PEP 285 [https://peps.python.org/pep-0285/] - Adding a bool type

Written and implemented by GvR.

PEP 293: Codec Error Handling Callbacks

When encoding a Unicode string into a byte string, unencodable characters may be encountered. So far, Python has allowed specifying the error processing as either “strict” (raising `UnicodeError`), “ignore” (skipping the character), or “replace” (using a question mark in the output string), with “strict” being the default behavior. It may be desirable to specify alternative processing of such errors, such as inserting an XML character reference or HTML entity reference into the converted string.

Python now has a flexible framework to add different processing strategies. New error handlers can be added with `codecs.register_error()`, and codecs then can access the error handler with `codecs.lookup_error()`. An equivalent C API has been added for codecs written in C. The error handler gets the necessary state information such as the string being converted, the position in the string where the error was detected, and the target encoding. The handler can then either raise an exception or return a replacement string.

Two additional error handlers have been implemented using this framework: “backslashreplace” uses Python backslash quoting to represent unencodable characters and “xmlcharrefreplace” emits XML character references.

See also

PEP 293 [https://peps.python.org/pep-0293/] - Codec Error Handling Callbacks

Written and implemented by Walter Dörwald.

PEP 301: Package Index and Metadata for Distutils

Support for the long-requested Python catalog makes its first appearance in 2.3.

The heart of the catalog is the new Distutils **register** command. Running `python setup.py register` will collect the metadata describing a package, such as its name, version, maintainer, description, &c., and send it to a central catalog server. The resulting catalog is available from https://pypi.org.

To make the catalog a bit more useful, a new optional *classifiers* keyword argument has been added to the Distutils **`setup()`** function. A list of Trove [http://catb.org/~esr/trove/]-style strings can be supplied to help classify the software.

Here's an example `setup.py` with classifiers, written to be compatible with older versions of the Distutils:

```
from distutils import core
kw = {'name': "Quixote",
      'version': "0.5.1",
      'description': "A highly Pythonic Web application
framework",
      # ...
      }

if (hasattr(core, 'setup_keywords') and
    'classifiers' in core.setup_keywords):
    kw['classifiers'] = \
        ['Topic :: Internet :: WWW/HTTP :: Dynamic Content',
         'Environment :: No Input/Output (Daemon)',
        'Intended Audience :: Developers'],

core.setup(**kw)
```

The full list of classifiers can be obtained by running `python setup.py register --list-classifiers`.

See also

PEP 301 [https://peps.python.org/pep-0301/] - Package Index and Metadata for Distutils
Written and implemented by Richard Jones.

PEP 302: New Import Hooks

While it’s been possible to write custom import hooks ever since the **`ihooks`** module was introduced in Python 1.3, no one has ever been really happy with it because writing new import hooks is difficult and messy. There have been various proposed alternatives such as the **`imputil`** and **`iu`** modules, but none of them has ever gained much acceptance, and none of them were easily usable from C code.

PEP 302 [https://peps.python.org/pep-0302/] borrows ideas from its predecessors, especially from Gordon McMillan’s **`iu`** module. Three new items are added to the `sys` module:

- `sys.path_hooks` is a list of callable objects; most often they’ll be classes. Each callable takes a string containing a path and either returns an importer object that will handle imports from this path or raises an `ImportError` exception if it can’t handle this path.
- `sys.path_importer_cache` caches importer objects for each path, so `sys.path_hooks` will only need to be traversed once for each path.

- `sys.meta_path` is a list of importer objects that will be traversed before `sys.path` is checked. This list is initially empty, but user code can add objects to it. Additional built-in and frozen modules can be imported by an object added to this list.

Importer objects must have a single method, `find_module(fullname, path=None)`. *fullname* will be a module or package name, e.g. `string` or `distutils.core`. **`find_module()`** must return a loader object that has a single method, `load_module(fullname)`, that creates and returns the corresponding module object.

Pseudo-code for Python’s new import logic, therefore, looks something like this (simplified a bit; see **PEP 302** [https://peps.python.org/pep-0302/] for the full details):

```
for mp in sys.meta_path:
    loader = mp(fullname)
    if loader is not None:
        <module> = loader.load_module(fullname)

for path in sys.path:
    for hook in sys.path_hooks:
        try:
            importer = hook(path)
        except ImportError:
            # ImportError, so try the other path hooks
            pass
        else:
            loader = importer.find_module(fullname)
            <module> = loader.load_module(fullname)

# Not found!
raise ImportError
```

See also

PEP 302 [https://peps.python.org/pep-0302/] - New Import Hooks

Written by Just van Rossum and Paul Moore. Implemented by Just van Rossum.

PEP 305: Comma-separated Files

Comma-separated files are a format frequently used for exporting data from databases and spreadsheets. Python 2.3 adds a parser for comma-separated files.

Comma-separated format is deceptively simple at first glance:

```
Costs,150,200,3.95
```

Read a line and call `line.split(',')`: what could be simpler? But toss in string data that can contain commas, and things get more complicated:

```
"Costs",150,200,3.95,"Includes taxes, shipping, and sundry items"
```

A big ugly regular expression can parse this, but using the new `csv` package is much simpler:

```
import csv

input = open('datafile', 'rb')
reader = csv.reader(input)
for line in reader:
    print line
```

The `reader()` function takes a number of different options. The field separator isn't limited to the comma and can be changed to any character, and so can the quoting and line-ending characters.

Different dialects of comma-separated files can be defined and registered; currently there are two dialects, both used by Microsoft Excel. A separate `csv.writer` class will generate comma-separated files from a succession of tuples or lists, quoting strings that contain the delimiter.

See also

PEP 305 [https://peps.python.org/pep-0305/] - CSV File API
Written and implemented by Kevin Altis, Dave Cole, Andrew McNamara, Skip Montanaro, Cliff Wells.

PEP 307: Pickle Enhancements

The `pickle` and `cPickle` modules received some attention during the 2.3 development cycle. In 2.2, new-style classes could be pickled without difficulty, but they weren’t pickled very compactly; PEP 307 [https://peps.python.org/pep-0307/] quotes a trivial example where a new-style class results in a pickled string three times longer than that for a classic class.

The solution was to invent a new pickle protocol. The `pickle.dumps()` function has supported a text-or-binary flag for a long time. In 2.3, this flag is redefined from a Boolean to an integer: 0 is the old text-mode pickle format, 1 is the old binary format, and now 2 is a new 2.3-specific format. A new constant, `pickle.HIGHEST_PROTOCOL`, can be used to select the fanciest protocol available.

Unpickling is no longer considered a safe operation. 2.2’s `pickle` provided hooks for trying to prevent unsafe classes from being unpickled (specifically, a `__safe_for_unpickling__` attribute), but none of this code was ever audited and therefore it's all been ripped out in 2.3. You should not unpickle untrusted data in any version of Python.

To reduce the pickling overhead for new-style classes, a new interface for customizing pickling was added using three special methods: `__getstate__()`, `__setstate__()`, and `__getnewargs__()`. Consult PEP 307 [https://peps.python.org/pep-0307/] for the full semantics of these methods.

As a way to compress pickles yet further, it's now possible to use integer codes instead of long strings to identify pickled classes. The Python Software Foundation will maintain a list of standardized codes; there's also a range of codes for private use. Currently no codes have been specified.

See also

PEP 307 [https://peps.python.org/pep-0307/] - Extensions to the pickle protocol

Written and implemented by Guido van Rossum and Tim Peters.

Extended Slices

Ever since Python 1.4, the slicing syntax has supported an optional third "step" or "stride" argument. For example, these are all legal Python syntax: `L[1:10:2]`, `L[:-1:1]`, `L[::-1]`. This was added to Python at the request of the developers of Numerical Python, which uses the third argument extensively. However, Python's built-in list, tuple, and string sequence types have never supported this feature, raising a `TypeError` if you tried it. Michael Hudson contributed a patch to fix this shortcoming.

For example, you can now easily extract the elements of a list that have even indexes:

```
>>> L = range(10)
>>> L[::2]
[0, 2, 4, 6, 8]
```

Negative values also work to make a copy of the same list in reverse order:

```
>>> L[::-1]
[9, 8, 7, 6, 5, 4, 3, 2, 1, 0]
```

This also works for tuples, arrays, and strings:

```
>>> s='abcd'
>>> s[::2]
'ac'
>>> s[::-1]
'dcba'
```

If you have a mutable sequence such as a list or an array you can assign to or delete an extended slice, but there are some differences between assignment to extended and regular slices. Assignment to a regular slice can be used to change the length of the sequence:

```
>>> a = range(3)
>>> a
[0, 1, 2]
>>> a[1:3] = [4, 5, 6]
>>> a
[0, 4, 5, 6]
```

Extended slices aren’t this flexible. When assigning to an extended slice, the list on the right hand side of the statement must contain the same number of items as the slice it is replacing:

```
>>> a = range(4)
>>> a
[0, 1, 2, 3]
>>> a[::2]
[0, 2]
>>> a[::2] = [0, -1]
>>> a
[0, 1, -1, 3]
>>> a[::2] = [0,1,2]
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
ValueError: attempt to assign sequence of size 3 to extended
slice of size 2
```

Deletion is more straightforward:

```
>>> a = range(4)
>>> a
[0, 1, 2, 3]
>>> a[::2]
[0, 2]
>>> del a[::2]
>>> a
[1, 3]
```

One can also now pass slice objects to the `__getitem__()` methods of the built-in sequences:

```
>>> range(10).__getitem__(slice(0, 5, 2))
[0, 2, 4]
```

Or use slice objects directly in subscripts:

```
>>> range(10)[slice(0, 5, 2)]
[0, 2, 4]
```

To simplify implementing sequences that support extended slicing, slice objects now have a method `indices(length)` which, given the length of a sequence, returns a `(start, stop, step)` tuple that can be passed directly to `range()`. **`indices()`** handles omitted and out-of-bounds indices in a manner consistent with regular slices (and this innocuous phrase hides a welter of confusing details!). The method is intended to be used like this:

```
class FakeSeq:
    ...
    def calc_item(self, i):
        ...
    def __getitem__(self, item):
        if isinstance(item, slice):
            indices = item.indices(len(self))
            return FakeSeq([self.calc_item(i) for i in
range(*indices)])
        else:
            return self.calc_item(i)
```

From this example you can also see that the built-in **`slice`** object is now the type object for the slice type, and is no longer a function. This is consistent with Python 2.2, where **`int`**, **`str`**, etc., underwent the same change.

Other Language Changes

Here are all of the changes that Python 2.3 makes to the core Python language.

- The **`yield`** statement is now always a keyword, as described in section PEP 255: Simple Generators of this document.

- A new built-in function **`enumerate()`** was added, as described in section PEP 279: enumerate() of this document.

- Two new constants, **`True`** and **`False`** were added along with the built-in **`bool`** type, as described in section PEP 285: A Boolean Type of this document.

- The **`int()`** type constructor will now return a long integer instead of raising an **`OverflowError`** when a string or floating-point number is too large to fit into an integer. This can lead to the paradoxical result that `isinstance(int(expression), int)` is false, but that seems unlikely to cause problems in practice.

- Built-in types now support the extended slicing syntax, as described in section Extended Slices of this document.

- A new built-in function, `sum(iterable, start=0)`, adds up the numeric items in the iterable object and returns their sum. **`sum()`** only accepts numbers, meaning that you can't use it to concatenate a bunch of strings. (Contributed by Alex Martelli.)

- `list.insert(pos, value)` used to insert *value* at the front of the list when *pos* was negative. The behaviour has now been changed to be consistent with slice indexing, so when *pos* is -1 the value will be inserted before the last element, and so forth.

- `list.index(value)`, which searches for *value* within the list and returns its index, now takes optional *start* and *stop* arguments to limit the search to only part of the list.

- Dictionaries have a new method, `pop(key[, *default*])`, that returns the value corresponding to *key* and removes that key/value pair from the dictionary. If the requested key isn't present in the dictionary, *default* is returned if it's specified and **`KeyError`** raised if it isn't.

  ```
  >>> d = {1:2}
  >>> d
  {1: 2}
  >>> d.pop(4)
  ```

```
Traceback (most recent call last):
  File "stdin", line 1, in ?
KeyError: 4
>>> d.pop(1)
2
>>> d.pop(1)
Traceback (most recent call last):
  File "stdin", line 1, in ?
KeyError: 'pop(): dictionary is empty'
>>> d
{}
>>>
```

There's also a new class method, `dict.fromkeys(iterable, value)`, that creates a dictionary with keys taken from the supplied iterator *iterable* and all values set to *value*, defaulting to `None`.

(Patches contributed by Raymond Hettinger.)

Also, the `dict()` constructor now accepts keyword arguments to simplify creating small dictionaries:

```
>>> dict(red=1, blue=2, green=3, black=4)
{'blue': 2, 'black': 4, 'green': 3, 'red': 1}
```

(Contributed by Just van Rossum.)

- The `assert` statement no longer checks the `__debug__` flag, so you can no longer disable assertions by assigning to `__debug__`. Running Python with the `-O` switch will still generate code that doesn't execute any assertions.

- Most type objects are now callable, so you can use them to create new objects such as functions, classes, and modules. (This means that the **`new`** module can be deprecated in a future Python version, because you can now use the type objects available in the `types` module.)

For example, you can create a new module object with the following code:

```
>>> import types
>>> m = types.ModuleType('abc','docstring')
>>> m
<module 'abc' (built-in)>
>>> m.__doc__
'docstring'
```

- A new warning, `PendingDeprecationWarning` was added to indicate features which are in the process of being deprecated. The warning will *not* be printed by default. To check for use of features that will be deprecated in the future, supply `-Walways::PendingDeprecationWarning::` on the command line or use `warnings.filterwarnings()`.
- The process of deprecating string-based exceptions, as in `raise "Error occurred"`, has begun. Raising a string will now trigger `PendingDeprecationWarning`.
- Using `None` as a variable name will now result in a `SyntaxWarning` warning. In a future version of Python, `None` may finally become a keyword.
- The **`xreadlines()`** method of file objects, introduced in Python 2.1, is no longer necessary because files now behave as their own iterator. **`xreadlines()`** was originally introduced as a faster way to loop over all the lines in a file, but now you can simply write `for line in file_obj`. File objects also have a new read-only **`encoding`** attribute that gives the encoding used by the file; Unicode strings written to the file will be automatically converted to bytes using the given encoding.
- The method resolution order used by new-style classes has changed, though you’ll only notice the difference if you have a really complicated inheritance hierarchy.

Classic classes are unaffected by this change. Python 2.2 originally used a topological sort of a class's ancestors, but 2.3 now uses the C3 algorithm as described in the paper "A Monotonic Superclass Linearization for Dylan" [https://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.19.3910]. To understand the motivation for this change, read Michele Simionato's article The Python 2.3 Method Resolution Order, or read the thread on python-dev starting with the message at https://mail.python.org/pipermail/python-dev/2002-October/029035.html. Samuele Pedroni first pointed out the problem and also implemented the fix by coding the C3 algorithm.

- Python runs multithreaded programs by switching between threads after executing N bytecodes. The default value for N has been increased from 10 to 100 bytecodes, speeding up single-threaded applications by reducing the switching overhead. Some multithreaded applications may suffer slower response time, but that's easily fixed by setting the limit back to a lower number using `sys.setcheckinterval(N)`. The limit can be retrieved with the new **`sys.getcheckinterval()`** function.

- One minor but far-reaching change is that the names of extension types defined by the modules included with Python now contain the module and a `'.'` in front of the type name. For example, in Python 2.2, if you created a socket and printed its **`__class__`**, you'd get this output:

```
>>> s = socket.socket()
>>> s.__class__
<type 'socket'>
```

  In 2.3, you get this:

```
>>> s.__class__
<type '_socket.socket'>
```

- One of the noted incompatibilities between old- and new-style classes has been removed: you can now assign to the `__name__` and `__bases__` attributes of new-style classes. There are some restrictions on what can be assigned to `__bases__` along the lines of those relating to assigning to an instance's `__class__` attribute.

String Changes

- The `in` operator now works differently for strings. Previously, when evaluating `X in Y` where *X* and *Y* are strings, *X* could only be a single character. That's now changed; *X* can be a string of any length, and `X in Y` will return `True` if *X* is a substring of *Y*. If *X* is the empty string, the result is always `True`.

  ```
  >>> 'ab' in 'abcd'
  True
  >>> 'ad' in 'abcd'
  False
  >>> '' in 'abcd'
  True
  ```

  Note that this doesn't tell you where the substring starts; if you need that information, use the `find()` string method.

- The `strip()`, `lstrip()`, and `rstrip()` string methods now have an optional argument for specifying the characters to strip. The default is still to remove all whitespace characters:

  ```
  >>> '   abc '.strip()
  'abc'
  >>> '><><abc<><><>'.strip('<>')
  ```

```
'abc'
>>> '><><abc<><><>\n'.strip('<>')
'abc<><><>\n'
>>> u'\u4000\u4001abc\u4000'.strip(u'\u4000')
u'\u4001abc'
>>>
```

(Suggested by Simon Brunning and implemented by Walter Dörwald.)

- The **`startswith()`** and **`endswith()`** string methods now accept negative numbers for the *start* and *end* parameters.

- Another new string method is **`zfill()`**, originally a function in the **`string`** module. **`zfill()`** pads a numeric string with zeros on the left until it's the specified width. Note that the `%` operator is still more flexible and powerful than **`zfill()`**.

```
>>> '45'.zfill(4)
'0045'
>>> '12345'.zfill(4)
'12345'
>>> 'goofy'.zfill(6)
'0goofy'
```

(Contributed by Walter Dörwald.)

- A new type object, **`basestring`**, has been added. Both 8-bit strings and Unicode strings inherit from this type, so `isinstance(obj, basestring)` will return **`True`** for either kind of string. It's a completely abstract type, so you can't create **`basestring`** instances.

- Interned strings are no longer immortal and will now be garbage-collected in the usual way when the only reference to them is from the internal dictionary of interned strings. (Implemented by Oren Tirosh.)

Optimizations

- The creation of new-style class instances has been made much faster; they’re now faster than classic classes!
- The `sort()` method of list objects has been extensively rewritten by Tim Peters, and the implementation is significantly faster.
- Multiplication of large long integers is now much faster thanks to an implementation of Karatsuba multiplication, an algorithm that scales better than the $O(n^2)$ required for the grade-school multiplication algorithm. (Original patch by Christopher A. Craig, and significantly reworked by Tim Peters.)
- The `SET_LINENO` opcode is now gone. This may provide a small speed increase, depending on your compiler’s idiosyncrasies. See section Other Changes and Fixes for a longer explanation. (Removed by Michael Hudson.)
- **`xrange()`** objects now have their own iterator, making `for i in xrange(n)` slightly faster than `for i in range(n)`. (Patch by Raymond Hettinger.)
- A number of small rearrangements have been made in various hotspots to improve performance, such as inlining a function or removing some code. (Implemented mostly by GvR, but lots of people have contributed single changes.)

The net result of the 2.3 optimizations is that Python 2.3 runs the pystone benchmark around 25% faster than Python 2.2.

New, Improved, and Deprecated Modules

As usual, Python’s standard library received a number of enhancements and bug fixes. Here’s a partial list of the most notable changes, sorted alphabetically by module name. Consult the `Misc/NEWS` file in the source tree for a more complete list of changes, or look through the CVS logs for all the details.

- The **`array`** module now supports arrays of Unicode characters using the `'u'` format character. Arrays also now support using the `+=` assignment operator to add another array’s contents, and the `*=` assignment operator to repeat an array. (Contributed by Jason Orendorff.)
- The **`bsddb`** module has been replaced by version 4.1.6 of the PyBSDDB [https://pybsddb.sourceforge.net] package, providing a more complete interface to the transactional features of the BerkeleyDB library.

  The old version of the module has been renamed to **`bsddb185`** and is no longer built automatically; you’ll have to edit `Modules/Setup` to enable it. Note that the new **`bsddb`** package is intended to be compatible with the old module, so be sure to file bugs if you discover any incompatibilities. When upgrading to Python 2.3, if the new interpreter is compiled with a new version of the underlying BerkeleyDB library, you will almost certainly have to convert your database files to the new version. You can do this fairly easily with the new scripts `db2pickle.py` and `pickle2db.py` which you will find in the distribution’s `Tools/scripts` directory. If you’ve already been using the PyBSDDB package and importing it as **`bsddb3`**, you will have to change your `import` statements to import it as **`bsddb`**.

- The new `bz2` module is an interface to the bz2 data compression library. bz2-compressed data is usually smaller than corresponding `zlib`-compressed data. (Contributed by Gustavo Niemeyer.)

- A set of standard date/time types has been added in the new `datetime` module. See the following section for more details.

- The Distutils **`Extension`** class now supports an extra constructor argument named *depends* for listing additional source files that an extension depends on. This lets Distutils recompile the module if any of the dependency files are modified. For example, if `sampmodule.c` includes the header file `sample.h`, you would create the **`Extension`** object like this:

  ```
  ext = Extension("samp",
                  sources=["sampmodule.c"],
                  depends=["sample.h"])
  ```

  Modifying `sample.h` would then cause the module to be recompiled. (Contributed by Jeremy Hylton.)

- Other minor changes to Distutils: it now checks for the `CC`, `CFLAGS`, **`CPP`**, `LDFLAGS`, and `CPPFLAGS` environment variables, using them to override the settings in Python’s configuration (contributed by Robert Weber).

- Previously the `doctest` module would only search the docstrings of public methods and functions for test cases, but it now also examines private ones as well. The `DocTestSuite()` function creates a `unittest.TestSuite` object from a set of `doctest` tests.

- The new `gc.get_referents(object)` function returns a list of all the objects referenced by *object*.

- The `getopt` module gained a new function, `gnu_getopt()`, that supports the same arguments as the existing `getopt()` function but uses GNU-style scanning mode. The existing `getopt()` stops processing options as soon as a non-option argument is encountered, but in GNU-style mode processing continues, meaning that options and arguments can be mixed. For example:

  ```
  >>> getopt.getopt(['-f', 'filename', 'output', '-v'],
  'f:v')
  ([('-f', 'filename')], ['output', '-v'])
  >>> getopt.gnu_getopt(['-f', 'filename', 'output', '-v'],
  'f:v')
  ([('-f', 'filename'), ('-v', '')], ['output'])
  ```

  (Contributed by Peter Åstrand.)

- The `grp`, `pwd`, and `resource` modules now return enhanced tuples:

  ```
  >>> import grp
  >>> g = grp.getgrnam('amk')
  >>> g.gr_name, g.gr_gid
  ('amk', 500)
  ```

- The `gzip` module can now handle files exceeding 2 GiB.

- The new `heapq` module contains an implementation of a heap queue algorithm. A heap is an array-like data structure that keeps items in a partially sorted order such that, for every index *k*, `heap[k] <= heap[2*k+1]` and `heap[k] <= heap[2*k+2]`. This makes it quick to remove the smallest item, and inserting a new item while maintaining the heap property is *O*(log *n*). (See https://xlinux.nist.gov/dads//HTML/priorityque.html for more information about the priority queue data structure.)

The **`heapq`** module provides **`heappush()`** and **`heappop()`** functions for adding and removing items while maintaining the heap property on top of some other mutable Python sequence type. Here’s an example that uses a Python list:

```
>>> import heapq
>>> heap = []
>>> for item in [3, 7, 5, 11, 1]:
...    heapq.heappush(heap, item)
...
>>> heap
[1, 3, 5, 11, 7]
>>> heapq.heappop(heap)
1
>>> heapq.heappop(heap)
3
>>> heap
[5, 7, 11]
```

(Contributed by Kevin O’Connor.)

- The IDLE integrated development environment has been updated using the code from the IDLEfork project (https://idlefork.sourceforge.net). The most notable feature is that the code being developed is now executed in a subprocess, meaning that there’s no longer any need for manual `reload()` operations. IDLE’s core code has been incorporated into the standard library as the **`idlelib`** package.
- The **`imaplib`** module now supports IMAP over SSL. (Contributed by Piers Lauder and Tino Lange.)
- The **`itertools`** contains a number of useful functions for use with iterators, inspired by various functions provided by the ML and Haskell languages. For example, `itertools.ifilter(predicate, iterator)` returns all elements in the iterator for which the function

`predicate()` returns **`True`**, and `itertools.repeat(obj, N)` returns `obj` *N* times. There are a number of other functions in the module; see the package's reference documentation for details. (Contributed by Raymond Hettinger.)

- Two new functions in the **`math`** module, `degrees(rads)` and `radians(degs)`, convert between radians and degrees. Other functions in the **`math`** module such as **`math.sin()`** and **`math.cos()`** have always required input values measured in radians. Also, an optional *base* argument was added to **`math.log()`** to make it easier to compute logarithms for bases other than `e` and `10`. (Contributed by Raymond Hettinger.)

- Several new POSIX functions (**`getpgid()`**, **`killpg()`**, **`lchown()`**, **`loadavg()`**, **`major()`**, **`makedev()`**, **`minor()`**, and **`mknod()`**) were added to the **`posix`** module that underlies the **`os`** module. (Contributed by Gustavo Niemeyer, Geert Jansen, and Denis S. Otkidach.)

- In the **`os`** module, the **`*stat()`** family of functions can now report fractions of a second in a timestamp. Such time stamps are represented as floats, similar to the value returned by **`time.time()`**.

  During testing, it was found that some applications will break if time stamps are floats. For compatibility, when using the tuple interface of the **`stat_result`** time stamps will be represented as integers. When using named fields (a feature first introduced in Python 2.2), time stamps are still represented as integers, unless **`os.stat_float_times()`** is invoked to enable float return values:

  ```
  >>> os.stat("/tmp").st_mtime
  1034791200
  ```

```
>>> os.stat_float_times(True)
>>> os.stat("/tmp").st_mtime
1034791200.6335014
```

In Python 2.4, the default will change to always returning floats.

Application developers should enable this feature only if all their libraries work properly when confronted with floating-point time stamps, or if they use the tuple API. If used, the feature should be activated on an application level instead of trying to enable it on a per-use basis.

- The `optparse` module contains a new parser for command-line arguments that can convert option values to a particular Python type and will automatically generate a usage message. See the following section for more details.

- The old and never-documented **`linuxaudiodev`** module has been deprecated, and a new version named **`ossaudiodev`** has been added. The module was renamed because the OSS sound drivers can be used on platforms other than Linux, and the interface has also been tidied and brought up to date in various ways. (Contributed by Greg Ward and Nicholas FitzRoy-Dale.)

- The new `platform` module contains a number of functions that try to determine various properties of the platform you’re running on. There are functions for getting the architecture, CPU type, the Windows OS version, and even the Linux distribution version. (Contributed by Marc-André Lemburg.)

- The parser objects provided by the `pyexpat` module can now optionally buffer character data, resulting in fewer calls to your character data handler and therefore faster performance. Setting the parser object’s `buffer_text` attribute to `True` will enable buffering.

- The `sample(population, k)` function was added to the `random` module. *population* is a sequence or **`xrange`** object containing the elements of a population, and `sample()` chooses *k* elements from the population without replacing chosen elements. *k* can be any value up to `len(population)`. For example:

  ```
  >>> days = ['Mo', 'Tu', 'We', 'Th', 'Fr', 'St', 'Sn']
  >>> random.sample(days, 3)      # Choose 3 elements
  ['St', 'Sn', 'Th']
  >>> random.sample(days, 7)      # Choose 7 elements
  ['Tu', 'Th', 'Mo', 'We', 'St', 'Fr', 'Sn']
  >>> random.sample(days, 7)      # Choose 7 again
  ['We', 'Mo', 'Sn', 'Fr', 'Tu', 'St', 'Th']
  >>> random.sample(days, 8)      # Can't choose eight
  Traceback (most recent call last):
    File "<stdin>", line 1, in ?
    File "random.py", line 414, in sample
        raise ValueError, "sample larger than population"
  ValueError: sample larger than population
  >>> random.sample(xrange(1,10000,2), 10)   # Choose ten odd
  nos. under 10000
  [3407, 3805, 1505, 7023, 2401, 2267, 9733, 3151, 8083,
  9195]
  ```

  The `random` module now uses a new algorithm, the Mersenne Twister, implemented in C. It’s faster and more extensively studied than the previous algorithm.

  (All changes contributed by Raymond Hettinger.)

- The `readline` module also gained a number of new functions: `get_history_item()`, `get_current_history_length()`, and `redisplay()`.

- The **`rexec`** and **`Bastion`** modules have been declared dead, and attempts to import them will fail with a `RuntimeError`. New-style classes provide new ways to break out of the restricted execution environment provided by **`rexec`**, and no one has interest in fixing them or time to do so. If you have applications using **`rexec`**, rewrite them to use something else.

  (Sticking with Python 2.2 or 2.1 will not make your applications any safer because there are known bugs in the **`rexec`** module in those versions. To repeat: if you're using **`rexec`**, stop using it immediately.)

- The **`rotor`** module has been deprecated because the algorithm it uses for encryption is not believed to be secure. If you need encryption, use one of the several AES Python modules that are available separately.

- The `shutil` module gained a `move(src, dest)` function that recursively moves a file or directory to a new location.

- Support for more advanced POSIX signal handling was added to the `signal` but then removed again as it proved impossible to make it work reliably across platforms.

- The `socket` module now supports timeouts. You can call the `settimeout(t)` method on a socket object to set a timeout of *t* seconds. Subsequent socket operations that take longer than *t* seconds to complete will abort and raise a `socket.timeout` exception.

  The original timeout implementation was by Tim O'Malley. Michael Gilfix integrated it into the Python `socket` module and shepherded it through a lengthy review. After the code was checked in, Guido van Rossum rewrote parts of it. (This is a good example of a collaborative development process in action.)

- On Windows, the `socket` module now ships with Secure Sockets Layer (SSL) support.
- The value of the C **`PYTHON_API_VERSION`** macro is now exposed at the Python level as `sys.api_version`. The current exception can be cleared by calling the new **`sys.exc_clear()`** function.
- The new `tarfile` module allows reading from and writing to **tar**-format archive files. (Contributed by Lars Gustäbel.)
- The new `textwrap` module contains functions for wrapping strings containing paragraphs of text. The `wrap(text, width)` function takes a string and returns a list containing the text split into lines of no more than the chosen width. The `fill(text, width)` function returns a single string, reformatted to fit into lines no longer than the chosen width. (As you can guess, `fill()` is built on top of `wrap()`. For example:

```
>>> import textwrap
>>> paragraph = "Not a whit, we defy augury: ... more text
..."
>>> textwrap.wrap(paragraph, 60)
["Not a whit, we defy augury: there's a special providence
in",
 "the fall of a sparrow. If it be now, 'tis not to come; if
it",
 ...]
>>> print textwrap.fill(paragraph, 35)
Not a whit, we defy augury: there's
a special providence in the fall of
a sparrow. If it be now, 'tis not
to come; if it be not to come, it
will be now; if it be not now, yet
it will come: the readiness is all.
>>>
```

The module also contains a `TextWrapper` class that actually implements the text wrapping strategy. Both the `TextWrapper` class and the `wrap()` and `fill()` functions support a number of additional keyword arguments for fine-tuning the formatting; consult the module's documentation for details. (Contributed by Greg Ward.)

- The **`thread`** and `threading` modules now have companion modules, **`dummy_thread`** and **`dummy_threading`**, that provide a do-nothing implementation of the **`thread`** module's interface for platforms where threads are not supported. The intention is to simplify thread-aware modules (ones that *don't* rely on threads to run) by putting the following code at the top:

  ```
  try:
      import threading as _threading
  except ImportError:
      import dummy_threading as _threading
  ```

  In this example, **`_threading`** is used as the module name to make it clear that the module being used is not necessarily the actual `threading` module. Code can call functions and use classes in **`_threading`** whether or not threads are supported, avoiding an `if` statement and making the code slightly clearer. This module will not magically make multithreaded code run without threads; code that waits for another thread to return or to do something will simply hang forever.

- The `time` module's `strptime()` function has long been an annoyance because it uses the platform C library's `strptime()` implementation, and different platforms sometimes have odd bugs. Brett Cannon contributed a portable implementation that's written in pure Python and should behave identically on all platforms.

- The new `timeit` module helps measure how long snippets of Python code take to execute. The `timeit.py` file can be run directly from the command line, or the module's `Timer` class can be imported and used directly. Here's a short example that figures out whether it's faster to convert an 8-bit string to Unicode by appending an empty Unicode string to it or by using the **`unicode()`** function:

  ```
  import timeit

  timer1 = timeit.Timer('unicode("abc")')
  timer2 = timeit.Timer('"abc" + u""')

  # Run three trials
  print timer1.repeat(repeat=3, number=100000)
  print timer2.repeat(repeat=3, number=100000)

  # On my laptop this outputs:
  # [0.36831796169281006, 0.37441694736480713,
  0.35304892063140869]
  # [0.17574405670166016, 0.18193507194519043,
  0.17565798759460449]
  ```

- The **`Tix`** module has received various bug fixes and updates for the current version of the Tix package.

- The **`Tkinter`** module now works with a thread-enabled version of Tcl. Tcl's threading model requires that widgets only be accessed from the thread in which they're created; accesses from another thread can cause Tcl to panic. For certain Tcl interfaces, **`Tkinter`** will now automatically avoid this when a widget is accessed from a different thread by marshalling a command, passing it to the correct thread, and waiting for the results. Other interfaces can't be handled automatically but **`Tkinter`** will now raise an exception on such an access so that you can at least find out about the problem. See https://mail.python.org/pipermail/python-dev/2002-December/031107.html for a more detailed explanation of this change. (Implemented by Martin von Löwis.)

- Calling Tcl methods through `_tkinter` no longer returns only strings. Instead, if Tcl returns other objects those objects are converted to their Python equivalent, if one exists, or wrapped with a `_tkinter.Tcl_Obj` object if no Python equivalent exists. This behavior can be controlled through the `wantobjects()` method of `tkapp` objects.

  When using `_tkinter` through the `Tkinter` module (as most Tkinter applications will), this feature is always activated. It should not cause compatibility problems, since Tkinter would always convert string results to Python types where possible.

  If any incompatibilities are found, the old behavior can be restored by setting the `wantobjects` variable in the `Tkinter` module to false before creating the first `tkapp` object.

  ```
  import Tkinter
  Tkinter.wantobjects = 0
  ```

  Any breakage caused by this change should be reported as a bug.

- The `UserDict` module has a new `DictMixin` class which defines all dictionary methods for classes that already have a minimum mapping interface. This greatly simplifies writing classes that need to be substitutable for dictionaries, such as the classes in the `shelve` module.

Adding the mix-in as a superclass provides the full dictionary interface whenever the class defines `__getitem__()`, `__setitem__()`, `__delitem__()`, and **`keys()`**. For example:

```
>>> import UserDict
>>> class SeqDict(UserDict.DictMixin):
...     """Dictionary lookalike implemented with lists."""
...     def __init__(self):
...         self.keylist = []
...         self.valuelist = []
...     def __getitem__(self, key):
...         try:
...             i = self.keylist.index(key)
...         except ValueError:
...             raise KeyError
...         return self.valuelist[i]
...     def __setitem__(self, key, value):
...         try:
...             i = self.keylist.index(key)
...             self.valuelist[i] = value
...         except ValueError:
...             self.keylist.append(key)
...             self.valuelist.append(value)
...     def __delitem__(self, key):
...         try:
...             i = self.keylist.index(key)
...         except ValueError:
...             raise KeyError
...         self.keylist.pop(i)
...         self.valuelist.pop(i)
...     def keys(self):
...         return list(self.keylist)
...
>>> s = SeqDict()
>>> dir(s)      # See that other dictionary methods are
implemented
['__cmp__', '__contains__', '__delitem__', '__doc__',
'__getitem__',
 '__init__', '__iter__', '__len__', '__module__',
'__repr__',
 '__setitem__', 'clear', 'get', 'has_key', 'items',
'iteritems',
 'iterkeys', 'itervalues', 'keylist', 'keys', 'pop',
'popitem',
 'setdefault', 'update', 'valuelist', 'values']
```

(Contributed by Raymond Hettinger.)

- The DOM implementation in `xml.dom.minidom` can now generate XML output in a particular encoding by providing an optional encoding argument to the `toxml()` and `toprettyxml()` methods of DOM nodes.

- The **`xmlrpclib`** module now supports an XML-RPC extension for handling nil data values such as Python’s `None`. Nil values are always supported on unmarshalling an XML-RPC response. To generate requests containing `None`, you must supply a true value for the *allow_none* parameter when creating a **`Marshaller`** instance.

- The new **`DocXMLRPCServer`** module allows writing self-documenting XML-RPC servers. Run it in demo mode (as a program) to see it in action. Pointing the web browser to the RPC server produces pydoc-style documentation; pointing xmlrpclib to the server allows invoking the actual methods. (Contributed by Brian Quinlan.)

- Support for internationalized domain names (RFCs 3454, 3490, 3491, and 3492) has been added. The “idna” encoding can be used to convert between a Unicode domain name and the ASCII-compatible encoding (ACE) of that name.

```
>{}>{}> u"www.Alliancefrançaise.nu".encode("idna")
'www.xn--alliancefranaise-npb.nu'
```

  The `socket` module has also been extended to transparently convert Unicode hostnames to the ACE version before passing them to the C library. Modules that deal with hostnames such as **`httplib`** and **`ftplib`**) also support Unicode host names; **`httplib`** also sends HTTP `Host` headers using the ACE version of the domain name. **`urllib`** supports Unicode URLs with non-ASCII host names as long as the `path` part of the URL is ASCII only.

To implement this change, the **`stringprep`** module, the `mkstringprep` tool and the `punycode` encoding have been added.

Date/Time Type

Date and time types suitable for expressing timestamps were added as the **`datetime`** module. The types don't support different calendars or many fancy features, and just stick to the basics of representing time.

The three primary types are: **`date`**, representing a day, month, and year; **`time`**, consisting of hour, minute, and second; and **`datetime`**, which contains all the attributes of both **`date`** and **`time`**. There's also a **`timedelta`** class representing differences between two points in time, and time zone logic is implemented by classes inheriting from the abstract **`tzinfo`** class.

You can create instances of **`date`** and **`time`** by either supplying keyword arguments to the appropriate constructor, e.g. `datetime.date(year=1972, month=10, day=15)`, or by using one of a number of class methods. For example, the **`today()`** class method returns the current local date.

Once created, instances of the date/time classes are all immutable. There are a number of methods for producing formatted strings from objects:

```
>>> import datetime
>>> now = datetime.datetime.now()
>>> now.isoformat()
'2002-12-30T21:27:03.994956'
>>> now.ctime()  # Only available on date, datetime
'Mon Dec 30 21:27:03 2002'
>>> now.strftime('%Y %d %b')
'2002 30 Dec'
```

The `replace()` method allows modifying one or more fields of a `date` or `datetime` instance, returning a new instance:

```
>>> d = datetime.datetime.now()
>>> d
datetime.datetime(2002, 12, 30, 22, 15, 38, 827738)
>>> d.replace(year=2001, hour = 12)
datetime.datetime(2001, 12, 30, 12, 15, 38, 827738)
>>>
```

Instances can be compared, hashed, and converted to strings (the result is the same as that of `isoformat()`). `date` and `datetime` instances can be subtracted from each other, and added to `timedelta` instances. The largest missing feature is that there's no standard library support for parsing strings and getting back a `date` or `datetime`.

For more information, refer to the module's reference documentation. (Contributed by Tim Peters.)

The optparse Module

The `getopt` module provides simple parsing of command-line arguments. The new `optparse` module (originally named Optik) provides more elaborate command-line parsing that follows the Unix conventions, automatically creates the output for **`--help`**, and can perform different actions for different options.

You start by creating an instance of `OptionParser` and telling it what your program's options are.

```
import sys
from optparse import OptionParser

op = OptionParser()
op.add_option('-i', '--input',
              action='store', type='string', dest='input',
              help='set input filename')
op.add_option('-l', '--length',
              action='store', type='int', dest='length',
              help='set maximum length of output')
```

Parsing a command line is then done by calling the `parse_args()` method.

```
options, args = op.parse_args(sys.argv[1:])
print options
print args
```

This returns an object containing all of the option values, and a list of strings containing the remaining arguments.

Invoking the script with the various arguments now works as you’d expect it to. Note that the length argument is automatically converted to an integer.

```
$ ./python opt.py -i data arg1
<Values at 0x400cad4c: {'input': 'data', 'length': None}>
['arg1']
$ ./python opt.py --input=data --length=4
<Values at 0x400cad2c: {'input': 'data', 'length': 4}>
[]
$
```

The help message is automatically generated for you:

```
$ ./python opt.py --help
usage: opt.py [options]

options:
  -h, --help            show this help message and exit
  -iINPUT, --input=INPUT
```

```
                        set input filename
  -lLENGTH, --length=LENGTH
                        set maximum length of output
$
```

See the module’s documentation for more details.

Optik was written by Greg Ward, with suggestions from the readers of the Getopt SIG.

Pymalloc: A Specialized Object Allocator

Pymalloc, a specialized object allocator written by Vladimir Marangozov, was a feature added to Python 2.1. Pymalloc is intended to be faster than the system `malloc()` and to have less memory overhead for allocation patterns typical of Python programs. The allocator uses C’s `malloc()` function to get large pools of memory and then fulfills smaller memory requests from these pools.

In 2.1 and 2.2, pymalloc was an experimental feature and wasn’t enabled by default; you had to explicitly enable it when compiling Python by providing the `--with-pymalloc` option to the **configure** script. In 2.3, pymalloc has had further enhancements and is now enabled by default; you’ll have to supply `--without-pymalloc` to disable it.

This change is transparent to code written in Python; however, pymalloc may expose bugs in C extensions. Authors of C extension modules should test their code with pymalloc enabled, because some incorrect code may cause core dumps at runtime.

There’s one particularly common error that causes problems. There are a number of memory allocation functions in Python’s C API that have previously just been aliases for the C library’s `malloc()` and `free()`, meaning that if you accidentally called mismatched functions the error wouldn’t be noticeable. When the object allocator is enabled, these functions aren’t aliases of `malloc()` and `free()` any more, and calling the wrong function to free memory may get you a core dump. For example, if memory was allocated using `PyObject_Malloc()`, it has to be freed using `PyObject_Free()`, not `free()`. A few modules included with Python fell afoul of this and had to be fixed; doubtless there are more third-party modules that will have the same problem.

As part of this change, the confusing multiple interfaces for allocating memory have been consolidated down into two API families. Memory allocated with one family must not be manipulated with functions from the other family. There is one family for allocating chunks of memory and another family of functions specifically for allocating Python objects.

- To allocate and free an undistinguished chunk of memory use the “raw memory” family: `PyMem_Malloc()`, `PyMem_Realloc()`, and `PyMem_Free()`.
- The “object memory” family is the interface to the pymalloc facility described above and is biased towards a large number of “small” allocations: `PyObject_Malloc()`, `PyObject_Realloc()`, and `PyObject_Free()`.
- To allocate and free Python objects, use the “object” family `PyObject_New`, `PyObject_NewVar`, and `PyObject_Del()`.

Thanks to lots of work by Tim Peters, pymalloc in 2.3 also provides debugging features to catch memory overwrites and doubled frees in both extension modules and in the interpreter itself. To enable this support, compile a debugging version of the Python interpreter by running **configure** with `--with-pydebug`.

To aid extension writers, a header file `Misc/pymemcompat.h` is distributed with the source to Python 2.3 that allows Python extensions to use the 2.3 interfaces to memory allocation while compiling against any version of Python since 1.5.2. You would copy the file from Python’s source distribution and bundle it with the source of your extension.

See also

https://hg.python.org/cpython/file/default/Objects/obmalloc.c

For the full details of the pymalloc implementation, see the comments at the top of the file `Objects/obmalloc.c` in the Python source code. The above link points to the file within the python.org SVN browser.

Build and C API Changes

Changes to Python’s build process and to the C API include:

- The cycle detection implementation used by the garbage collection has proven to be stable, so it’s now been made mandatory. You can no longer compile Python without it, and the **`--with-cycle-gc`** switch to **configure** has been removed.
- Python can now optionally be built as a shared library (`libpython2.3.so`) by supplying **`--enable-shared`** when running Python’s **configure** script. (Contributed by Ondrej Palkovsky.)
- The **`DL_EXPORT`** and **`DL_IMPORT`** macros are now deprecated. Initialization functions for Python extension modules should now be declared using the new macro `PyMODINIT_FUNC`, while the Python core will generally use the **`PyAPI_FUNC`** and **`PyAPI_DATA`** macros.

- The interpreter can be compiled without any docstrings for the built-in functions and modules by supplying **`--without-doc-strings`** to the **configure** script. This makes the Python executable about 10% smaller, but will also mean that you can’t get help for Python’s built-ins. (Contributed by Gustavo Niemeyer.)
- The **`PyArg_NoArgs()`** macro is now deprecated, and code that uses it should be changed. For Python 2.2 and later, the method definition table can specify the **`METH_NOARGS`** flag, signalling that there are no arguments, and the argument checking can then be removed. If compatibility with pre-2.2 versions of Python is important, the code could use `PyArg_ParseTuple(args, "")` instead, but this will be slower than using **`METH_NOARGS`**.
- **`PyArg_ParseTuple()`** accepts new format characters for various sizes of unsigned integers: `B` for unsigned char, `H` for unsigned short int, `I` for unsigned int, and `K` for unsigned long long.
- A new function, `PyObject_DelItemString(mapping, char *key)` was added as shorthand for `PyObject_DelItem(mapping, PyString_New(key))`.
- File objects now manage their internal string buffer differently, increasing it exponentially when needed. This results in the benchmark tests in `Lib/test/test_bufio.py` speeding up considerably (from 57 seconds to 1.7 seconds, according to one measurement).
- It’s now possible to define class and static methods for a C extension type by setting either the **`METH_CLASS`** or **`METH_STATIC`** flags in a method’s **`PyMethodDef`** structure.
- Python now includes a copy of the Expat XML parser’s source code, removing any dependence on a system version or local installation of Expat.
- If you dynamically allocate type objects in your extension, you should be aware of a change in the rules relating to the **`__module__`** and **`__name__`** attributes. In

summary, you will want to ensure the type’s dictionary contains a `'__module__'` key; making the module name the part of the type name leading up to the final period will no longer have the desired effect. For more detail, read the API reference documentation or the source.

Port-Specific Changes

Support for a port to IBM’s OS/2 using the EMX runtime environment was merged into the main Python source tree. EMX is a POSIX emulation layer over the OS/2 system APIs. The Python port for EMX tries to support all the POSIX-like capability exposed by the EMX runtime, and mostly succeeds; **`fork()`** and **`fcntl()`** are restricted by the limitations of the underlying emulation layer. The standard OS/2 port, which uses IBM’s Visual Age compiler, also gained support for case-sensitive import semantics as part of the integration of the EMX port into CVS. (Contributed by Andrew MacIntyre.)

On MacOS, most toolbox modules have been weaklinked to improve backward compatibility. This means that modules will no longer fail to load if a single routine is missing on the current OS version. Instead calling the missing routine will raise an exception. (Contributed by Jack Jansen.)

The RPM spec files, found in the `Misc/RPM/` directory in the Python source distribution, were updated for 2.3. (Contributed by Sean Reifschneider.)

Other new platforms now supported by Python include AtheOS (http://www.atheos.cx/), GNU/Hurd, and OpenVMS.

Other Changes and Fixes

As usual, there were a bunch of other improvements and bugfixes scattered throughout the source tree. A search through the CVS change logs finds there were 523 patches applied and 514 bugs fixed between Python 2.2 and 2.3. Both figures are likely to be underestimates.

Some of the more notable changes are:

- If the **`PYTHONINSPECT`** environment variable is set, the Python interpreter will enter the interactive prompt after running a Python program, as if Python had been invoked with the **`-i`** option. The environment variable can be set before running the Python interpreter, or it can be set by the Python program as part of its execution.

- The `regrtest.py` script now provides a way to allow “all resources except *foo*.” A resource name passed to the **`-u`** option can now be prefixed with a hyphen (`'-'`) to mean “remove this resource.” For example, the option ‘`-uall,-bsddb`’ could be used to enable the use of all resources except `bsddb`.

- The tools used to build the documentation now work under Cygwin as well as Unix.

- The `SET_LINENO` opcode has been removed. Back in the mists of time, this opcode was needed to produce line numbers in tracebacks and support trace functions (for, e.g., **`pdb`**). Since Python 1.5, the line numbers in tracebacks have been computed using a different mechanism that works with “python -O”. For Python 2.3 Michael Hudson implemented a similar scheme to determine when to call the trace function, removing the need for `SET_LINENO` entirely.

It would be difficult to detect any resulting difference from Python code, apart from a slight speed up when Python is run without `-O`.

C extensions that access the `f_lineno` field of frame objects should instead call `PyCode_Addr2Line(f->f_code, f->f_lasti)`. This will have the added effect of making the code work as desired under “python -O” in earlier versions of Python.

A nifty new feature is that trace functions can now assign to the `f_lineno` attribute of frame objects, changing the line that will be executed next. A `jump` command has been added to the `pdb` debugger taking advantage of this new feature. (Implemented by Richie Hindle.)

Porting to Python 2.3

This section lists previously described changes that may require changes to your code:

- `yield` is now always a keyword; if it’s used as a variable name in your code, a different name must be chosen.
- For strings *X* and *Y*, `X in Y` now works if *X* is more than one character long.
- The `int()` type constructor will now return a long integer instead of raising an `OverflowError` when a string or floating-point number is too large to fit into an integer.
- If you have Unicode strings that contain 8-bit characters, you must declare the file’s encoding (UTF-8, Latin-1, or whatever) by adding a comment to the top of

the file. See section PEP 263: Source Code Encodings for more information.

- Calling Tcl methods through **`_tkinter`** no longer returns only strings. Instead, if Tcl returns other objects those objects are converted to their Python equivalent, if one exists, or wrapped with a **`_tkinter.Tcl_Obj`** object if no Python equivalent exists.

- Large octal and hex literals such as `0xffffffff` now trigger a **`FutureWarning`**. Currently they're stored as 32-bit numbers and result in a negative value, but in Python 2.4 they'll become positive long integers.

  There are a few ways to fix this warning. If you really need a positive number, just add an `L` to the end of the literal. If you're trying to get a 32-bit integer with low bits set and have previously used an expression such as `~(1 << 31)`, it's probably clearest to start with all bits set and clear the desired upper bits. For example, to clear just the top bit (bit 31), you could write `0xffffffffL &~(1L<<31)`.

- You can no longer disable assertions by assigning to `__debug__`.

- The Distutils **`setup()`** function has gained various new keyword arguments such as *depends*. Old versions of the Distutils will abort if passed unknown keywords. A solution is to check for the presence of the new **`get_distutil_options()`** function in your `setup.py` and only uses the new keywords with a version of the Distutils that supports them:

  ```
  from distutils import core

  kw = {'sources': 'foo.c', ...}
  ```

```
if hasattr(core, 'get_distutil_options'):
    kw['depends'] = ['foo.h']
ext = Extension(**kw)
```

- Using `None` as a variable name will now result in a `SyntaxWarning` warning.
- Names of extension types defined by the modules included with Python now contain the module and a `'.'` in front of the type name.

Acknowledgements

The author would like to thank the following people for offering suggestions, corrections and assistance with various drafts of this article: Jeff Bauer, Simon Brunning, Brett Cannon, Michael Chermside, Andrew Dalke, Scott David Daniels, Fred L. Drake, Jr., David Fraser, Kelly Gerber, Raymond Hettinger, Michael Hudson, Chris Lambert, Detlef Lannert, Martin von Löwis, Andrew MacIntyre, Lalo Martins, Chad Netzer, Gustavo Niemeyer, Neal Norwitz, Hans Nowak, Chris Reedy, Francesco Ricciardi, Vinay Sajip, Neil Schemenauer, Roman Suzi, Jason Tishler, Just van Rossum.

What’s New in Python 2.2

Author:: A.M. Kuchling

Introduction

This article explains the new features in Python 2.2.2, released on October 14, 2002. Python 2.2.2 is a bugfix release of Python 2.2, originally released on December 21, 2001.

Python 2.2 can be thought of as the “cleanup release”. There are some features such as generators and iterators that are completely new, but most of the changes, significant and far-reaching though they may be, are aimed at cleaning up irregularities and dark corners of the language design.

This article doesn’t attempt to provide a complete specification of the new features, but instead provides a convenient overview. For full details, you should refer to the documentation for Python 2.2, such as the Python Library Reference [https://docs.python.org/2.2/lib/lib.html] and the Python Reference Manual [https://docs.python.org/2.2/ref/ref.html]. If you want to understand the complete implementation and design rationale for a change, refer to the PEP for a particular new feature.

PEPs 252 and 253: Type and Class Changes

The largest and most far-reaching changes in Python 2.2 are to Python's model of objects and classes. The changes should be backward compatible, so it's likely that your code will continue to run unchanged, but the changes provide some amazing new capabilities. Before beginning this, the longest and most complicated section of this article, I'll provide an overview of the changes and offer some comments.

A long time ago I wrote a web page listing flaws in Python's design. One of the most significant flaws was that it's impossible to subclass Python types implemented in C. In particular, it's not possible to subclass built-in types, so you can't just subclass, say, lists in order to add a single useful method to them. The **`UserList`** module provides a class that supports all of the methods of lists and that can be subclassed further, but there's lots of C code that expects a regular Python list and won't accept a `UserList` instance.

Python 2.2 fixes this, and in the process adds some exciting new capabilities. A brief summary:

- You can subclass built-in types such as lists and even integers, and your subclasses should work in every place that requires the original type.
- It's now possible to define static and class methods, in addition to the instance methods available in previous versions of Python.
- It's also possible to automatically call methods on accessing or setting an instance attribute by using a new mechanism called *properties*. Many uses of `__getattr__()` can be rewritten to use properties instead, making the resulting code simpler and faster. As a small side benefit, attributes can now have docstrings, too.
- The list of legal attributes for an instance can be limited to a particular set using *slots*, making it possible to

safeguard against typos and perhaps make more optimizations possible in future versions of Python.

Some users have voiced concern about all these changes. Sure, they say, the new features are neat and lend themselves to all sorts of tricks that weren’t possible in previous versions of Python, but they also make the language more complicated. Some people have said that they’ve always recommended Python for its simplicity, and feel that its simplicity is being lost.

Personally, I think there’s no need to worry. Many of the new features are quite esoteric, and you can write a lot of Python code without ever needed to be aware of them. Writing a simple class is no more difficult than it ever was, so you don’t need to bother learning or teaching them unless they’re actually needed. Some very complicated tasks that were previously only possible from C will now be possible in pure Python, and to my mind that’s all for the better.

I’m not going to attempt to cover every single corner case and small change that were required to make the new features work. Instead this section will paint only the broad strokes. See section Related Links, “Related Links”, for further sources of information about Python 2.2’s new object model.

Old and New Classes

First, you should know that Python 2.2 really has two kinds of classes: classic or old-style classes, and new-style classes. The old-style class model is exactly the same as the class model in earlier versions of Python. All the new features described in this section apply only to new-style classes. This divergence isn’t intended to last forever;

eventually old-style classes will be dropped, possibly in Python 3.0.

So how do you define a new-style class? You do it by subclassing an existing new-style class. Most of Python's built-in types, such as integers, lists, dictionaries, and even files, are new-style classes now. A new-style class named `object`, the base class for all built-in types, has also been added so if no built-in type is suitable, you can just subclass `object`:

```
class C(object):
    def __init__ (self):
        ...
    ...
```

This means that `class` statements that don't have any base classes are always classic classes in Python 2.2. (Actually you can also change this by setting a module-level variable named **`__metaclass__`** — see **PEP 253** [https://peps.python.org/pep-0253/] for the details — but it's easier to just subclass `object`.)

The type objects for the built-in types are available as built-ins, named using a clever trick. Python has always had built-in functions named `int()`, `float()`, and `str()`. In 2.2, they aren't functions any more, but type objects that behave as factories when called.

```
>>> int
<type 'int'>
>>> int('123')
123
```

To make the set of types complete, new type objects such as `dict()` and **`file()`** have been added. Here's a more interesting example, adding a **`lock()`** method to file objects:

```
class LockableFile(file):
    def lock (self, operation, length=0, start=0, whence=0):
        import fcntl
        return fcntl.lockf(self.fileno(), operation,
                           length, start, whence)
```

The now-obsolete **`posixfile`** module contained a class that emulated all of a file object's methods and also added a **`lock()`** method, but this class couldn't be passed to internal functions that expected a built-in file, something which is possible with our new **`LockableFile`**.

Descriptors

In previous versions of Python, there was no consistent way to discover what attributes and methods were supported by an object. There were some informal conventions, such as defining **`__members__`** and **`__methods__`** attributes that were lists of names, but often the author of an extension type or a class wouldn't bother to define them. You could fall back on inspecting the `__dict__` of an object, but when class inheritance or an arbitrary **`__getattr__()`** hook were in use this could still be inaccurate.

The one big idea underlying the new class model is that an API for describing the attributes of an object using *descriptors* has been formalized. Descriptors specify the value of an attribute, stating whether it's a method or a field. With the descriptor API, static methods and class methods become possible, as well as more exotic constructs.

Attribute descriptors are objects that live inside class objects, and have a few attributes of their own:

- `__name__` is the attribute's name.
- `__doc__` is the attribute's docstring.

- `__get__(object)` is a method that retrieves the attribute value from *object*.
- `__set__(object, value)` sets the attribute on *object* to *value*.
- `__delete__(object, value)` deletes the *value* attribute of *object*.

For example, when you write `obj.x`, the steps that Python actually performs are:

```
descriptor = obj.__class__.x
descriptor.__get__(obj)
```

For methods, **`descriptor.__get__`** returns a temporary object that's callable, and wraps up the instance and the method to be called on it. This is also why static methods and class methods are now possible; they have descriptors that wrap up just the method, or the method and the class. As a brief explanation of these new kinds of methods, static methods aren't passed the instance, and therefore resemble regular functions. Class methods are passed the class of the object, but not the object itself. Static and class methods are defined like this:

```
class C(object):
    def f(arg1, arg2):
        ...
    f = staticmethod(f)

    def g(cls, arg1, arg2):
        ...
    g = classmethod(g)
```

The **`staticmethod()`** function takes the function **`f()`**, and returns it wrapped up in a descriptor so it can be stored in the class object. You might expect there to be special syntax for creating such methods (`def static f`, `defstatic f()`, or something like that) but no such syntax has been defined yet; that’s been left for future versions of Python.

More new features, such as slots and properties, are also implemented as new kinds of descriptors, and it’s not difficult to write a descriptor class that does something novel. For example, it would be possible to write a descriptor class that made it possible to write Eiffel-style preconditions and postconditions for a method. A class that used this feature might be defined like this:

```
from eiffel import eiffelmethod

class C(object):
    def f(self, arg1, arg2):
        # The actual function
        ...
    def pre_f(self):
        # Check preconditions
        ...
    def post_f(self):
        # Check postconditions
        ...

    f = eiffelmethod(f, pre_f, post_f)
```

Note that a person using the new **`eiffelmethod()`** doesn’t have to understand anything about descriptors. This is why I think the new features don’t increase the basic complexity of the language. There will be a few wizards who need to know about it in order to write **`eiffelmethod()`** or the ZODB or whatever, but most users will just write code on top of the resulting libraries and ignore the implementation details.

Multiple Inheritance: The Diamond Rule

Multiple inheritance has also been made more useful through changing the rules under which names are resolved. Consider this set of classes (diagram taken from **PEP 253** [https://peps.python.org/pep-0253/] by Guido van Rossum):

```
      class A:
        ^ ^  def save(self): ...
       /   \
      /     \
     /       \
    /         \
class B     class C:
    ^         ^  def save(self): ...
     \       /
      \     /
       \   /
        \ /
      class D
```

The lookup rule for classic classes is simple but not very smart; the base classes are searched depth-first, going from left to right. A reference to **`D.save()`** will search the classes **`D`**, **`B`**, and then **`A`**, where **`save()`** would be found and returned. **`C.save()`** would never be found at all. This is bad, because if **`C`**’s **`save()`** method is saving some internal state specific to **`C`**, not calling it will result in that state never getting saved.

New-style classes follow a different algorithm that’s a bit more complicated to explain, but does the right thing in this situation. (Note that Python 2.3 changes this algorithm to one that produces the same results in most cases, but produces more useful results for really complicated inheritance graphs.)

1. List all the base classes, following the classic lookup rule and include a class multiple times if it’s visited repeatedly. In the above example, the list of visited classes is [**`D`**, **`B`**, **`A`**, **`C`**, **`A`**].
2. Scan the list for duplicated classes. If any are found, remove all but one occurrence, leaving the *last* one in

the list. In the above example, the list becomes [**`D`**, **`B`**, **`C`**, **`A`**] after dropping duplicates.

Following this rule, referring to **`D.save()`** will return **`C.save()`**, which is the behaviour we’re after. This lookup rule is the same as the one followed by Common Lisp. A new built-in function, `super()`, provides a way to get at a class’s superclasses without having to reimplement Python’s algorithm. The most commonly used form will be `super(class, obj)`, which returns a bound superclass object (not the actual class object). This form will be used in methods to call a method in the superclass; for example, **`D`**’s **`save()`** method would look like this:

```
class D (B,C):
    def save (self):
        # Call superclass .save()
        super(D, self).save()
        # Save D's private information here
        ...
```

`super()` can also return unbound superclass objects when called as `super(class)` or `super(class1, class2)`, but this probably won’t often be useful.

Attribute Access

A fair number of sophisticated Python classes define hooks for attribute access using `__getattr__()`; most commonly this is done for convenience, to make code more readable by automatically mapping an attribute access such as `obj.parent` into a method call such as `obj.get_parent`. Python 2.2 adds some new ways of controlling attribute access.

First, `__getattr__(attr_name)` is still supported by new-style classes, and nothing about it has changed. As before, it will be called when an attempt is made to access `obj.foo` and no attribute named `foo` is found in the instance's dictionary.

New-style classes also support a new method, `__getattribute__(attr_name)`. The difference between the two methods is that **`__getattribute__()`** is *always* called whenever any attribute is accessed, while the old **`__getattr__()`** is only called if `foo` isn't found in the instance's dictionary.

However, Python 2.2's support for *properties* will often be a simpler way to trap attribute references. Writing a **`__getattr__()`** method is complicated because to avoid recursion you can't use regular attribute accesses inside them, and instead have to mess around with the contents of **`__dict__`**. **`__getattr__()`** methods also end up being called by Python when it checks for other methods such as **`__repr__()`** or **`__coerce__()`**, and so have to be written with this in mind. Finally, calling a function on every attribute access results in a sizable performance loss.

`property` is a new built-in type that packages up three functions that get, set, or delete an attribute, and a docstring. For example, if you want to define a **`size`** attribute that's computed, but also settable, you could write:

```
class C(object):
    def get_size (self):
        result = ... computation ...
        return result
    def set_size (self, size):
        ... compute something based on the size
        and set internal state appropriately ...

    # Define a property.  The 'delete this attribute'
    # method is defined as None, so the attribute
    # can't be deleted.
    size = property(get_size, set_size,
                    None,
                    "Storage size of this instance")
```

That is certainly clearer and easier to write than a pair of **`__getattr__()`**/**`__setattr__()`** methods that check for the **`size`** attribute and handle it specially while retrieving all other attributes from the instance's [`__dict__`]. Accesses to **`size`** are also the only ones which have to perform the work of calling a function, so references to other attributes run at their usual speed.

Finally, it's possible to constrain the list of attributes that can be referenced on an object using the new [`__slots__`] class attribute. Python objects are usually very dynamic; at any time it's possible to define a new attribute on an instance by just doing `obj.new_attr=1`. A new-style class can define a class attribute named [`__slots__`] to limit the legal attributes to a particular set of names. An example will make this clear:

```
>>> class C(object):
...     __slots__ = ('template', 'name')
...
>>> obj = C()
>>> print obj.template
None
>>> obj.template = 'Test'
>>> print obj.template
Test
>>> obj.newattr = None
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
AttributeError: 'C' object has no attribute 'newattr'
```

Note how you get an [`AttributeError`] on the attempt to assign to an attribute not listed in [`__slots__`].

Related Links

This section has just been a quick overview of the new features, giving enough of an explanation to start you programming, but many details have been simplified or ignored. Where should you go to get a more complete picture?

The Descriptor Guide is a lengthy tutorial introduction to the descriptor features, written by Guido van Rossum. If my description has whetted your appetite, go read this tutorial next, because it goes into much more detail about the new features while still remaining quite easy to read.

Next, there are two relevant PEPs, **PEP 252** [https://peps.python.org/pep-0252/] and **PEP 253** [https://peps.python.org/pep-0253/]. **PEP 252** [https://peps.python.org/pep-0252/] is titled “Making Types Look More Like Classes”, and covers the descriptor API. **PEP 253** [https://peps.python.org/pep-0253/] is titled “Subtyping Built-in Types”, and describes the changes to type objects that make it possible to subtype built-in objects. **PEP 253** [https://peps.python.org/pep-0253/] is the more complicated PEP of the two, and at a few points the necessary explanations of types and meta-types may cause your head to explode. Both PEPs were written and implemented by Guido van Rossum, with substantial assistance from the rest of the Zope Corp. team.

Finally, there’s the ultimate authority: the source code. Most of the machinery for the type handling is in `Objects/typeobject.c`, but you should only resort to it after all other avenues have been exhausted, including posting a question to python-list or python-dev.

PEP 234: Iterators

Another significant addition to 2.2 is an iteration interface at both the C and Python levels. Objects can define how they can be looped over by callers.

In Python versions up to 2.1, the usual way to make `for item in obj` work is to define a `__getitem__()` method that looks something like this:

```
def __getitem__(self, index):
    return <next item>
```

`__getitem__()` is more properly used to define an indexing operation on an object so that you can write `obj[5]` to retrieve the sixth element. It’s a bit misleading when you’re using this only to support `for` loops. Consider some file-like object that wants to be looped over; the *index* parameter is essentially meaningless, as the class probably assumes that a series of `__getitem__()` calls will be made with *index* incrementing by one each time. In other words, the presence of the `__getitem__()` method doesn’t mean that using `file[5]` to randomly access the sixth element will work, though it really should.

In Python 2.2, iteration can be implemented separately, and `__getitem__()` methods can be limited to classes that really do support random access. The basic idea of iterators is simple. A new built-in function, `iter(obj)` or `iter(C, sentinel)`, is used to get an iterator. `iter(obj)` returns an iterator for the object *obj*, while `iter(C, sentinel)` returns an iterator that will invoke the callable object *C* until it returns *sentinel* to signal that the iterator is done.

Python classes can define an **`__iter__()`** method, which should create and return a new iterator for the object; if the object is its own iterator, this method can just return `self`. In particular, iterators will usually be their own iterators. Extension types implemented in C can implement a `tp_iter` function in order to return an iterator, and extension types that want to behave as iterators can define a `tp_iternext` function.

So, after all this, what do iterators actually do? They have one required method, **`next()`**, which takes no arguments and returns the next value. When there are no more values to be returned, calling **`next()`** should raise the **`StopIteration`** exception.

```
>>> L = [1,2,3]
>>> i = iter(L)
>>> print i
<iterator object at 0x8116870>
>>> i.next()
1
>>> i.next()
2
>>> i.next()
3
>>> i.next()
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
StopIteration
>>>
```

In 2.2, Python’s **`for`** statement no longer expects a sequence; it expects something for which **`iter()`** will return an iterator. For backward compatibility and convenience, an iterator is automatically constructed for sequences that don’t implement **`__iter__()`** or a **`tp_iter`** slot, so `for i in [1,2,3]` will still work. Wherever the Python interpreter loops over a sequence, it’s been changed to use the iterator protocol. This means you can do things like this:

```
>>> L = [1,2,3]
>>> i = iter(L)
>>> a,b,c = i
>>> a,b,c
(1, 2, 3)
```

Iterator support has been added to some of Python’s basic types. Calling **`iter()`** on a dictionary will return an iterator which loops over its keys:

```
>>> m = {'Jan': 1, 'Feb': 2, 'Mar': 3, 'Apr': 4, 'May': 5,
'Jun': 6,
...      'Jul': 7, 'Aug': 8, 'Sep': 9, 'Oct': 10, 'Nov': 11,
'Dec': 12}
>>> for key in m: print key, m[key]
...
Mar 3
Feb 2
Aug 8
Sep 9
May 5
Jun 6
Jul 7
Jan 1
Apr 4
Nov 11
Dec 12
Oct 10
```

That's just the default behaviour. If you want to iterate over keys, values, or key/value pairs, you can explicitly call the **`iterkeys()`**, **`itervalues()`**, or **`iteritems()`** methods to get an appropriate iterator. In a minor related change, the **`in`** operator now works on dictionaries, so `key in dict` is now equivalent to `dict.has_key(key)`.

Files also provide an iterator, which calls the **`readline()`** method until there are no more lines in the file. This means you can now read each line of a file using code like this:

```
for line in file:
    # do something for each line
    ...
```

Note that you can only go forward in an iterator; there's no way to get the previous element, reset the iterator, or make a copy of it. An iterator object could provide such additional capabilities, but the iterator protocol only requires a **`next()`** method.

See also

PEP 234 [https://peps.python.org/pep-0234/] - Iterators
Written by Ka-Ping Yee and GvR; implemented by the Python Labs crew, mostly by GvR and Tim Peters.

PEP 255: Simple Generators

Generators are another new feature, one that interacts with the introduction of iterators.

You’re doubtless familiar with how function calls work in Python or C. When you call a function, it gets a private namespace where its local variables are created. When the function reaches a `return` statement, the local variables are destroyed and the resulting value is returned to the caller. A later call to the same function will get a fresh new set of local variables. But, what if the local variables weren’t thrown away on exiting a function? What if you could later resume the function where it left off? This is what generators provide; they can be thought of as resumable functions.

Here’s the simplest example of a generator function:

```
def generate_ints(N):
    for i in range(N):
        yield i
```

A new keyword, `yield`, was introduced for generators. Any function containing a **`yield`** statement is a generator function; this is detected by Python’s bytecode compiler which compiles the function specially as a result. Because a new keyword was introduced, generators must be explicitly enabled in a module by including a `from __future__ import` `generators` statement near the top of the module's source code. In Python 2.3 this statement will become unnecessary.

When you call a generator function, it doesn't return a single value; instead it returns a generator object that supports the iterator protocol. On executing the `yield` statement, the generator outputs the value of `i`, similar to a `return` statement. The big difference between **`yield`** and a **`return`** statement is that on reaching a **`yield`** the generator's state of execution is suspended and local variables are preserved. On the next call to the generator's `next()` method, the function will resume executing immediately after the **`yield`** statement. (For complicated reasons, the **`yield`** statement isn't allowed inside the **`try`** block of a `try`...`finally` statement; read **PEP 255** [https://peps.python.org/pep-0255/] for a full explanation of the interaction between **`yield`** and exceptions.)

Here's a sample usage of the **`generate_ints()`** generator:

```
>>> gen = generate_ints(3)
>>> gen
<generator object at 0x8117f90>
>>> gen.next()
0
>>> gen.next()
1
>>> gen.next()
2
>>> gen.next()
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
  File "<stdin>", line 2, in generate_ints
StopIteration
```

You could equally write `for i in generate_ints(5)`, or `a,b,c = generate_ints(3)`.

Inside a generator function, the **`return`** statement can only be used without a value, and signals the end of the procession of values; afterwards the generator cannot return any further values. **`return`** with a value, such as `return 5`, is a syntax error inside a generator function. The end of the generator's results can also be indicated by raising **`StopIteration`** manually, or by just letting the flow of execution fall off the bottom of the function.

You could achieve the effect of generators manually by writing your own class and storing all the local variables of the generator as instance variables. For example, returning a list of integers could be done by setting `self.count` to 0, and having the **`next()`** method increment `self.count` and return it. However, for a moderately complicated generator, writing a corresponding class would be much messier. `Lib/test/test_generators.py` contains a number of more interesting examples. The simplest one implements an in-order traversal of a tree using generators recursively.

```
# A recursive generator that generates Tree leaves in in-order.
def inorder(t):
    if t:
        for x in inorder(t.left):
            yield x
        yield t.label
        for x in inorder(t.right):
            yield x
```

Two other examples in `Lib/test/test_generators.py` produce solutions for the N-Queens problem (placing N queens on an NxN chess board so that no queen threatens another) and the Knight's Tour (a route that takes a knight to every square of an NxN chessboard without visiting any square twice).

The idea of generators comes from other programming languages, especially Icon (https://www2.cs.arizona.edu/icon/), where the idea of generators is central. In Icon, every expression and function call behaves like a generator. One example from “An Overview of the Icon Programming Language” at https://www2.cs.arizona.edu/icon/docs/ipd266.htm gives an idea of what this looks like:

```
sentence := "Store it in the neighboring harbor"
if (i := find("or", sentence)) > 5 then write(i)
```

In Icon the `find()` function returns the indexes at which the substring “or” is found: 3, 23, 33. In the `if` statement, `i` is first assigned a value of 3, but 3 is less than 5, so the comparison fails, and Icon retries it with the second value of 23. 23 is greater than 5, so the comparison now succeeds, and the code prints the value 23 to the screen.

Python doesn’t go nearly as far as Icon in adopting generators as a central concept. Generators are considered a new part of the core Python language, but learning or using them isn’t compulsory; if they don’t solve any problems that you have, feel free to ignore them. One novel feature of Python’s interface as compared to Icon’s is that a generator’s state is represented as a concrete object (the iterator) that can be passed around to other functions or stored in a data structure.

See also

PEP 255 [https://peps.python.org/pep-0255/] - Simple Generators

Written by Neil Schemenauer, Tim Peters, Magnus Lie Hetland. Implemented mostly by Neil Schemenauer and Tim Peters, with other fixes from the Python Labs crew.

PEP 237: Unifying Long Integers and Integers

In recent versions, the distinction between regular integers, which are 32-bit values on most machines, and long integers, which can be of arbitrary size, was becoming an annoyance. For example, on platforms that support files larger than `2**32` bytes, the **`tell()`** method of file objects has to return a long integer. However, there were various bits of Python that expected plain integers and would raise an error if a long integer was provided instead. For example, in Python 1.5, only regular integers could be used as a slice index, and `'abc'[1L:]` would raise a **`TypeError`** exception with the message ‘slice index must be int’.

Python 2.2 will shift values from short to long integers as required. The ‘L’ suffix is no longer needed to indicate a long integer literal, as now the compiler will choose the appropriate type. (Using the ‘L’ suffix will be discouraged in future 2.x versions of Python, triggering a warning in Python 2.4, and probably dropped in Python 3.0.) Many operations that used to raise an **`OverflowError`** will now return a long integer as their result. For example:

```
>>> 1234567890123
1234567890123L
>>> 2 ** 64
18446744073709551616L
```

In most cases, integers and long integers will now be treated identically. You can still distinguish them with the **`type()`** built-in function, but that’s rarely needed.

See also

PEP 237 [https://peps.python.org/pep-0237/] - Unifying Long Integers and Integers

Written by Moshe Zadka and Guido van Rossum. Implemented mostly by Guido van Rossum.

PEP 238: Changing the Division Operator

The most controversial change in Python 2.2 heralds the start of an effort to fix an old design flaw that's been in Python from the beginning. Currently Python's division operator, `/`, behaves like C's division operator when presented with two integer arguments: it returns an integer result that's truncated down when there would be a fractional part. For example, `3/2` is 1, not 1.5, and `(-1)/2` is -1, not -0.5. This means that the results of division can vary unexpectedly depending on the type of the two operands and because Python is dynamically typed, it can be difficult to determine the possible types of the operands.

(The controversy is over whether this is *really* a design flaw, and whether it's worth breaking existing code to fix this. It's caused endless discussions on python-dev, and in July 2001 erupted into a storm of acidly sarcastic postings on *comp.lang.python*. I won't argue for either side here and will stick to describing what's implemented in 2.2. Read **PEP 238** [https://peps.python.org/pep-0238/] for a summary of arguments and counter-arguments.)

Because this change might break code, it's being introduced very gradually. Python 2.2 begins the transition, but the switch won't be complete until Python 3.0.

First, I’ll borrow some terminology from **PEP 238** [https://peps.python.org/pep-0238/]. “True division” is the division that most non-programmers are familiar with: 3/2 is 1.5, 1/4 is 0.25, and so forth. “Floor division” is what Python’s `/` operator currently does when given integer operands; the result is the floor of the value returned by true division. “Classic division” is the current mixed behaviour of `/`; it returns the result of floor division when the operands are integers, and returns the result of true division when one of the operands is a floating-point number.

Here are the changes 2.2 introduces:

- A new operator, `//`, is the floor division operator. (Yes, we know it looks like C++’s comment symbol.) `//` *always* performs floor division no matter what the types of its operands are, so `1 // 2` is 0 and `1.0 // 2.0` is also 0.0.

  `//` is always available in Python 2.2; you don’t need to enable it using a `__future__` statement.

- By including a `from __future__ import division` in a module, the `/` operator will be changed to return the result of true division, so `1/2` is 0.5. Without the `__future__` statement, `/` still means classic division. The default meaning of `/` will not change until Python 3.0.

- Classes can define methods called **`__truediv__()`** and **`__floordiv__()`** to overload the two division operators. At the C level, there are also slots in the **`PyNumberMethods`** structure so extension types can define the two operators.

- Python 2.2 supports some command-line arguments for testing whether code will work with the changed division semantics. Running python with **`-Q warn`** will cause a warning to be issued whenever division is applied to two integers. You can use this to find code that's affected by the change and fix it. By default, Python 2.2 will simply perform classic division without a warning; the warning will be turned on by default in Python 2.3.

See also

PEP 238 [https://peps.python.org/pep-0238/] - Changing the Division Operator

Written by Moshe Zadka and Guido van Rossum. Implemented by Guido van Rossum..

Unicode Changes

Python's Unicode support has been enhanced a bit in 2.2. Unicode strings are usually stored as UCS-2, as 16-bit unsigned integers. Python 2.2 can also be compiled to use UCS-4, 32-bit unsigned integers, as its internal encoding by supplying `--enable-unicode=ucs4` to the configure script. (It's also possible to specify `--disable-unicode` to completely disable Unicode support.)

When built to use UCS-4 (a “wide Python”), the interpreter can natively handle Unicode characters from U+000000 to U+110000, so the range of legal values for the `unichr()` function is expanded accordingly. Using an interpreter compiled to use UCS-2 (a “narrow Python”), values greater than 65535 will still cause `unichr()` to raise a `ValueError` exception. This is all described in **PEP 261** [https://peps.python.org/pep-0261/], “Support for ‘wide' Unicode characters”; consult it for further details.

Another change is simpler to explain. Since their introduction, Unicode strings have supported an **`encode()`** method to convert the string to a selected encoding such as UTF-8 or Latin-1. A symmetric `decode([*encoding*])` method has been added to 8-bit strings (though not to Unicode strings) in 2.2. **`decode()`** assumes that the string is in the specified encoding and decodes it, returning whatever is returned by the codec.

Using this new feature, codecs have been added for tasks not directly related to Unicode. For example, codecs have been added for uu-encoding, MIME's base64 encoding, and compression with the **`zlib`** module:

```
>>> s = """Here is a lengthy piece of redundant, overly
verbose,
... and repetitive text.
... """
>>> data = s.encode('zlib')
>>> data
'x\x9c\r\xc9\xc1\r\x80 \x10\x04\xc0?Ul...'
>>> data.decode('zlib')
'Here is a lengthy piece of redundant, overly verbose,\nand
repetitive text.\n'
>>> print s.encode('uu')
begin 666 <data>
M2&5R92!I<R!A(&QE;F=T:'D@<&EE8V4@;V8@<F5D=6YD86YT+"!O=F5R;'D@
>=F5R8F]S92P*86YD(')E<&5T:71I=F4@=&5X="X*

end
>>> "sheesh".encode('rot-13')
'furrfu'
```

To convert a class instance to Unicode, a **`__unicode__()`** method can be defined by a class, analogous to **`__str__()`**.

`encode()`, **`decode()`**, and **`__unicode__()`** were implemented by Marc-André Lemburg. The changes to support using UCS-4 internally were implemented by Fredrik Lundh and Martin von Löwis.

See also

[PEP 261](https://peps.python.org/pep-0261/) [https://peps.python.org/pep-0261/] - Support for 'wide' Unicode characters
Written by Paul Prescod.

PEP 227: Nested Scopes

In Python 2.1, statically nested scopes were added as an optional feature, to be enabled by a `from __future__ import nested_scopes` directive. In 2.2 nested scopes no longer need to be specially enabled, and are now always present. The rest of this section is a copy of the description of nested scopes from my "What's New in Python 2.1" document; if you read it when 2.1 came out, you can skip the rest of this section.

The largest change introduced in Python 2.1, and made complete in 2.2, is to Python's scoping rules. In Python 2.0, at any given time there are at most three namespaces used to look up variable names: local, module-level, and the built-in namespace. This often surprised people because it didn't match their intuitive expectations. For example, a nested recursive function definition doesn't work:

```
def f():
    ...
    def g(value):
        ...
        return g(value-1) + 1
    ...
```

The function **`g()`** will always raise a `NameError` exception, because the binding of the name `g` isn't in either its local namespace or in the module-level namespace. This isn't much of a problem in practice (how often do you recursively define interior functions like this?), but this also made using the `lambda` expression clumsier, and this was a problem in practice. In code which uses **`lambda`** you can often find local variables being copied by passing them as the default values of arguments.

```
def find(self, name):
    "Return list of any entries equal to 'name'"
    L = filter(lambda x, name=name: x == name,
               self.list_attribute)
    return L
```

The readability of Python code written in a strongly functional style suffers greatly as a result.

The most significant change to Python 2.2 is that static scoping has been added to the language to fix this problem. As a first effect, the `name=name` default argument is now unnecessary in the above example. Put simply, when a given variable name is not assigned a value within a function (by an assignment, or the `def`, `class`, or `import` statements), references to the variable will be looked up in the local namespace of the enclosing scope. A more detailed explanation of the rules, and a dissection of the implementation, can be found in the PEP.

This change may cause some compatibility problems for code where the same variable name is used both at the module level and as a local variable within a function that contains further function definitions. This seems rather unlikely though, since such code would have been pretty confusing to read in the first place.

One side effect of the change is that the `from module import *` and `exec` statements have been made illegal inside a function scope under certain conditions. The Python reference manual has said all along that `from module import *` is only legal at the top level of a module, but the CPython interpreter has never enforced this before. As part of the implementation of nested scopes, the compiler which turns Python source into bytecodes has to generate different code to access variables in a containing scope. `from module import *` and `exec` make it impossible for the compiler to figure this out, because they add names to the local namespace that are unknowable at compile time. Therefore, if a function contains function definitions or **`lambda`** expressions with free variables, the compiler will flag this by raising a **`SyntaxError`** exception.

To make the preceding explanation a bit clearer, here’s an example:

```
x = 1
def f():
    # The next line is a syntax error
    exec 'x=2'
    def g():
        return x
```

Line 4 containing the `exec` statement is a syntax error, since `exec` would define a new local variable named `x` whose value should be accessed by **`g()`**.

This shouldn’t be much of a limitation, since `exec` is rarely used in most Python code (and when it is used, it’s often a sign of a poor design anyway).

See also

PEP 227 [https://peps.python.org/pep-0227/] - Statically Nested Scopes

Written and implemented by Jeremy Hylton.

New and Improved Modules

- The `xmlrpclib` module was contributed to the standard library by Fredrik Lundh, providing support for writing XML-RPC clients. XML-RPC is a simple remote procedure call protocol built on top of HTTP and XML. For example, the following snippet retrieves a list of RSS channels from the O’Reilly Network, and then lists the recent headlines for one channel:

  ```
  import xmlrpclib
  s = xmlrpclib.Server(
        'http://www.oreillynet.com/meerkat/xml-rpc/server.php')
  channels = s.meerkat.getChannels()
  # channels is a list of dictionaries, like this:
  # [{'id': 4, 'title': 'Freshmeat Daily News'}
  #  {'id': 190, 'title': '32Bits Online'},
  #  {'id': 4549, 'title': '3DGamers'}, ... ]

  # Get the items for one channel
  items = s.meerkat.getItems( {'channel': 4} )

  # 'items' is another list of dictionaries, like this:
  # [{'link': 'http://freshmeat.net/releases/52719/',
  #   'description': 'A utility which converts HTML to XSL FO.',
  #   'title': 'html2fo 0.3 (Default)'}, ... ]
  ```

  The `SimpleXMLRPCServer` module makes it easy to create straightforward XML-RPC servers. See http://xmlrpc.scripting.com/ for more information about XML-RPC.

- The new `hmac` module implements the HMAC algorithm described by **RFC 2104** [https://datatracker.ietf.org/doc/html/rfc2104.html]. (Contributed by Gerhard Häring.)
- Several functions that originally returned lengthy tuples now return pseudo-sequences that still behave like tuples but also have mnemonic attributes such as **`memberst_mtime`** or `tm_year`. The enhanced functions include `stat()`, `fstat()`, `statvfs()`, and `fstatvfs()` in the `os` module, and `localtime()`, `gmtime()`, and `strptime()` in the `time` module.

  For example, to obtain a file's size using the old tuples, you'd end up writing something like `file_size = os.stat(filename)[stat.ST_SIZE]`, but now this can be written more clearly as `file_size = os.stat(filename).st_size`.

  The original patch for this feature was contributed by Nick Mathewson.
- The Python profiler has been extensively reworked and various errors in its output have been corrected. (Contributed by Fred L. Drake, Jr. and Tim Peters.)
- The `socket` module can be compiled to support IPv6; specify the **`--enable-ipv6`** option to Python's configure script. (Contributed by Jun-ichiro "itojun" Hagino.)
- Two new format characters were added to the `struct` module for 64-bit integers on platforms that support the C long long type. `q` is for a signed 64-bit integer, and `Q` is for an unsigned one. The value is returned in Python's long integer type. (Contributed by Tim Peters.)

- In the interpreter's interactive mode, there's a new built-in function **`help()`** that uses the **`pydoc`** module introduced in Python 2.1 to provide interactive help. `help(object)` displays any available help text about *object*. **`help()`** with no argument puts you in an online help utility, where you can enter the names of functions, classes, or modules to read their help text. (Contributed by Guido van Rossum, using Ka-Ping Yee's **`pydoc`** module.)

- Various bugfixes and performance improvements have been made to the SRE engine underlying the **`re`** module. For example, the **`re.sub()`** and **`re.split()`** functions have been rewritten in C. Another contributed patch speeds up certain Unicode character ranges by a factor of two, and a new **`finditer()`** method that returns an iterator over all the non-overlapping matches in a given string. (SRE is maintained by Fredrik Lundh. The BIGCHARSET patch was contributed by Martin von Löwis.)

- The **`smtplib`** module now supports **RFC 2487** [https://datatracker.ietf.org/doc/html/rfc2487.html], "Secure SMTP over TLS", so it's now possible to encrypt the SMTP traffic between a Python program and the mail transport agent being handed a message. **`smtplib`** also supports SMTP authentication. (Contributed by Gerhard Häring.)

- The **`imaplib`** module, maintained by Piers Lauder, has support for several new extensions: the NAMESPACE extension defined in **RFC 2342** [https://datatracker.ietf.org/doc/html/rfc2342.html], SORT, GETACL and SETACL. (Contributed by Anthony Baxter and Michel Pelletier.)

- The **`rfc822`** module's parsing of email addresses is now compliant with **RFC 2822** [https://datatracker.ietf.org/doc/html/rfc2822.html], an update to **RFC 822** [https://datatracker.ietf.org/doc/html/rfc822.html]. (The module’s name is *not* going to be changed to `rfc2822`.) A new package, `email`, has also been added for parsing and generating e-mail messages. (Contributed by Barry Warsaw, and arising out of his work on Mailman.)

- The `difflib` module now contains a new `Differ` class for producing human-readable lists of changes (a “delta”) between two sequences of lines of text. There are also two generator functions, `ndiff()` and `restore()`, which respectively return a delta from two sequences, or one of the original sequences from a delta. (Grunt work contributed by David Goodger, from ndiff.py code by Tim Peters who then did the generatorization.)

- New constants `ascii_letters`, `ascii_lowercase`, and `ascii_uppercase` were added to the `string` module. There were several modules in the standard library that used `string.letters` to mean the ranges A-Za-z, but that assumption is incorrect when locales are in use, because `string.letters` varies depending on the set of legal characters defined by the current locale. The buggy modules have all been fixed to use `ascii_letters` instead. (Reported by an unknown person; fixed by Fred L. Drake, Jr.)

- The `mimetypes` module now makes it easier to use alternative MIME-type databases by the addition of a `MimeTypes` class, which takes a list of filenames to be parsed. (Contributed by Fred L. Drake, Jr.)

- A `Timer` class was added to the `threading` module that allows scheduling an activity to happen at some future time. (Contributed by Itamar Shtull-Trauring.)

Interpreter Changes and Fixes

Some of the changes only affect people who deal with the Python interpreter at the C level because they're writing Python extension modules, embedding the interpreter, or just hacking on the interpreter itself. If you only write Python code, none of the changes described here will affect you very much.

- Profiling and tracing functions can now be implemented in C, which can operate at much higher speeds than Python-based functions and should reduce the overhead of profiling and tracing. This will be of interest to authors of development environments for Python. Two new C functions were added to Python's API, `PyEval_SetProfile()` and `PyEval_SetTrace()`. The existing `sys.setprofile()` and `sys.settrace()` functions still exist, and have simply been changed to use the new C-level interface. (Contributed by Fred L. Drake, Jr.)

- Another low-level API, primarily of interest to implementers of Python debuggers and development tools, was added. `PyInterpreterState_Head()` and `PyInterpreterState_Next()` let a caller walk through all the existing interpreter objects; `PyInterpreterState_ThreadHead()` and `PyThreadState_Next()` allow looping over all the thread states for a given interpreter. (Contributed by David Beazley.)

- The C-level interface to the garbage collector has been changed to make it easier to write extension types that support garbage collection and to debug misuses of the functions. Various functions have slightly different semantics, so a bunch of functions had to be renamed. Extensions that use the old API will still compile but will *not* participate in garbage collection, so updating them for 2.2 should be considered fairly high priority.

To upgrade an extension module to the new API, perform the following steps:

- Rename **`Py_TPFLAGS_GC`** to `Py_TPFLAGS_HAVE_GC`.
- Use `PyObject_GC_New()` or `PyObject_GC_NewVar()` to allocate objects, and `PyObject_GC_Del()` to deallocate them.
- Rename **`PyObject_GC_Init()`** to `PyObject_GC_Track()` and **`PyObject_GC_Fini()`** to `PyObject_GC_UnTrack()`.
- Remove **`PyGC_HEAD_SIZE`** from object size calculations.
- Remove calls to **`PyObject_AS_GC()`** and **`PyObject_FROM_GC()`**.
- A new `et` format sequence was added to `PyArg_ParseTuple()`; `et` takes both a parameter and an encoding name, and converts the parameter to the given encoding if the parameter turns out to be a Unicode string, or leaves it alone if it’s an 8-bit string, assuming it to already be in the desired encoding. This differs from the `es` format character, which assumes that 8-bit strings are in Python’s default ASCII encoding and converts them to the specified new encoding. (Contributed by M.-A. Lemburg, and used for the MBCS support on Windows described in the following section.)
- A different argument parsing function, `PyArg_UnpackTuple()`, has been added that’s simpler and presumably faster. Instead of specifying a format string, the caller simply gives the minimum and maximum number of arguments expected, and a set of pointers to PyObject* variables that will be filled in with argument values.

- Two new flags `METH_NOARGS` and `METH_O` are available in method definition tables to simplify implementation of methods with no arguments or a single untyped argument. Calling such methods is more efficient than calling a corresponding method that uses `METH_VARARGS`. Also, the old `METH_OLDARGS` style of writing C methods is now officially deprecated.

- Two new wrapper functions, `PyOS_snprintf()` and `PyOS_vsnprintf()` were added to provide cross-platform implementations for the relatively new `snprintf()` and `vsnprintf()` C lib APIs. In contrast to the standard `sprintf()` and `vsprintf()` functions, the Python versions check the bounds of the buffer used to protect against buffer overruns. (Contributed by M.-A. Lemburg.)

- The `_PyTuple_Resize()` function has lost an unused parameter, so now it takes 2 parameters instead of 3. The third argument was never used, and can simply be discarded when porting code from earlier versions to Python 2.2.

Other Changes and Fixes

As usual there were a bunch of other improvements and bugfixes scattered throughout the source tree. A search through the CVS change logs finds there were 527 patches applied and 683 bugs fixed between Python 2.1 and 2.2; 2.2.1 applied 139 patches and fixed 143 bugs; 2.2.2 applied 106 patches and fixed 82 bugs. These figures are likely to be underestimates.

Some of the more notable changes are:

- The code for the MacOS port for Python, maintained by Jack Jansen, is now kept in the main Python CVS tree, and many changes have been made to support MacOS X.

The most significant change is the ability to build Python as a framework, enabled by supplying the **`--enable-framework`** option to the configure script when compiling Python. According to Jack Jansen, “This installs a self-contained Python installation plus the OS X framework “glue” into `/Library/Frameworks/Python.framework` (or another location of choice). For now there is little immediate added benefit to this (actually, there is the disadvantage that you have to change your PATH to be able to find Python), but it is the basis for creating a full-blown Python application, porting the MacPython IDE, possibly using Python as a standard OSA scripting language and much more.”

Most of the MacPython toolbox modules, which interface to MacOS APIs such as windowing, QuickTime, scripting, etc. have been ported to OS X, but they’ve been left commented out in `setup.py`. People who want to experiment with these modules can uncomment them manually.

- Keyword arguments passed to built-in functions that don’t take them now cause a `TypeError` exception to be raised, with the message “*function* takes no keyword arguments”.

- Weak references, added in Python 2.1 as an extension module, are now part of the core because they’re used in the implementation of new-style classes. The `ReferenceError` exception has therefore moved from the `weakref` module to become a built-in exception.

- A new script, `Tools/scripts/cleanfuture.py` by Tim Peters, automatically removes obsolete `__future__` statements from Python source code.

- An additional *flags* argument has been added to the built-in function **`compile()`**, so the behaviour of `__future__` statements can now be correctly observed in simulated shells, such as those presented by IDLE and other development environments. This is described in **PEP 264** [https://peps.python.org/pep-0264/]. (Contributed by Michael Hudson.)

- The new license introduced with Python 1.6 wasn’t GPL-compatible. This is fixed by some minor textual changes to the 2.2 license, so it’s now legal to embed Python inside a GPLed program again. Note that Python itself is not GPLed, but instead is under a license that’s essentially equivalent to the BSD license, same as it always was. The license changes were also applied to the Python 2.0.1 and 2.1.1 releases.

- When presented with a Unicode filename on Windows, Python will now convert it to an MBCS encoded string, as used by the Microsoft file APIs. As MBCS is explicitly used by the file APIs, Python’s choice of ASCII as the default encoding turns out to be an annoyance. On Unix, the locale’s character set is used if `locale.nl_langinfo(CODESET)` is available. (Windows support was contributed by Mark Hammond with assistance from Marc-André Lemburg. Unix support was added by Martin von Löwis.)

- Large file support is now enabled on Windows. (Contributed by Tim Peters.)

- The `Tools/scripts/ftpmirror.py` script now parses a `.netrc` file, if you have one. (Contributed by Mike Romberg.)

- Some features of the object returned by the **`xrange()`** function are now deprecated, and trigger warnings when they're accessed; they'll disappear in Python 2.3. **`xrange`** objects tried to pretend they were full sequence types by supporting slicing, sequence multiplication, and the **`in`** operator, but these features were rarely used and therefore buggy. The **`tolist()`** method and the **`start`**, **`stop`**, and **`step`** attributes are also being deprecated. At the C level, the fourth argument to the **`PyRange_New()`** function, `repeat`, has also been deprecated.

- There were a bunch of patches to the dictionary implementation, mostly to fix potential core dumps if a dictionary contains objects that sneakily changed their hash value, or mutated the dictionary they were contained in. For a while python-dev fell into a gentle rhythm of Michael Hudson finding a case that dumped core, Tim Peters fixing the bug, Michael finding another case, and round and round it went.

- On Windows, Python can now be compiled with Borland C thanks to a number of patches contributed by Stephen Hansen, though the result isn't fully functional yet. (But this *is* progress…)

- Another Windows enhancement: Wise Solutions generously offered PythonLabs use of their InstallerMaster 8.1 system. Earlier PythonLabs Windows installers used Wise 5.0a, which was beginning to show its age. (Packaged up by Tim Peters.)

- Files ending in `.pyw` can now be imported on Windows. `.pyw` is a Windows-only thing, used to indicate that a script needs to be run using PYTHONW.EXE instead of PYTHON.EXE in order to prevent a DOS console from popping up to display the output. This patch makes it possible to import such scripts, in case they're also usable as modules. (Implemented by David Bolen.)

- On platforms where Python uses the C **`dlopen()`** function to load extension modules, it's now possible to set the flags used by **`dlopen()`** using the `sys.getdlopenflags()` and `sys.setdlopenflags()` functions. (Contributed by Bram Stolk.)

- The `pow()` built-in function no longer supports 3 arguments when floating-point numbers are supplied. `pow(x, y, z)` returns `(x**y) % z`, but this is never useful for floating-point numbers, and the final result varies unpredictably depending on the platform. A call such as `pow(2.0, 8.0, 7.0)` will now raise a `TypeError` exception.

Acknowledgements

The author would like to thank the following people for offering suggestions, corrections and assistance with various drafts of this article: Fred Bremmer, Keith Briggs, Andrew Dalke, Fred L. Drake, Jr., Carel Fellinger, David Goodger, Mark Hammond, Stephen Hansen, Michael Hudson, Jack Jansen, Marc-André Lemburg, Martin von Löwis, Fredrik Lundh, Michael McLay, Nick Mathewson, Paul Moore, Gustavo Niemeyer, Don O'Donnell, Joonas Paalasma, Tim Peters, Jens Quade, Tom Reinhardt, Neil Schemenauer, Guido van Rossum, Greg Ward, Edward Welbourne.

What’s New in Python 2.1

Author:: A.M. Kuchling

Introduction

This article explains the new features in Python 2.1. While there aren’t as many changes in 2.1 as there were in Python 2.0, there are still some pleasant surprises in store. 2.1 is the first release to be steered through the use of Python Enhancement Proposals, or PEPs, so most of the sizable changes have accompanying PEPs that provide more complete documentation and a design rationale for the change. This article doesn’t attempt to document the new features completely, but simply provides an overview of the new features for Python programmers. Refer to the Python 2.1 documentation, or to the specific PEP, for more details about any new feature that particularly interests you.

One recent goal of the Python development team has been to accelerate the pace of new releases, with a new release coming every 6 to 9 months. 2.1 is the first release to come out at this faster pace, with the first alpha appearing in January, 3 months after the final version of 2.0 was released.

The final release of Python 2.1 was made on April 17, 2001.

PEP 227: Nested Scopes

The largest change in Python 2.1 is to Python’s scoping rules. In Python 2.0, at any given time there are at most three namespaces used to look up variable names: local, module-level, and the built-in namespace. This often surprised people because it didn’t match their intuitive expectations. For example, a nested recursive function definition doesn’t work:

```
def f():
    ...
    def g(value):
        ...
        return g(value-1) + 1
    ...
```

The function **`g()`** will always raise a **`NameError`** exception, because the binding of the name `g` isn’t in either its local namespace or in the module-level namespace. This isn’t much of a problem in practice (how often do you recursively define interior functions like this?), but this also made using the **`lambda`** expression clumsier, and this was a problem in practice. In code which uses **`lambda`** you can often find local variables being copied by passing them as the default values of arguments.

```
def find(self, name):
    "Return list of any entries equal to 'name'"
    L = filter(lambda x, name=name: x == name,
               self.list_attribute)
    return L
```

The readability of Python code written in a strongly functional style suffers greatly as a result.

The most significant change to Python 2.1 is that static scoping has been added to the language to fix this problem. As a first effect, the `name=name` default argument is now unnecessary in the above example. Put simply, when a given variable name is not assigned a value within a function (by an assignment, or the **`def`**, **`class`**, or **`import`** statements), references to the variable will be looked up in the local namespace of the enclosing scope. A more detailed explanation of the rules, and a dissection of the implementation, can be found in the PEP.

This change may cause some compatibility problems for code where the same variable name is used both at the module level and as a local variable within a function that contains further function definitions. This seems rather unlikely though, since such code would have been pretty confusing to read in the first place.

One side effect of the change is that the `from module import *` and `exec` statements have been made illegal inside a function scope under certain conditions. The Python reference manual has said all along that `from module import *` is only legal at the top level of a module, but the CPython interpreter has never enforced this before. As part of the implementation of nested scopes, the compiler which turns Python source into bytecodes has to generate different code to access variables in a containing scope. `from module import *` and `exec` make it impossible for the compiler to figure this out, because they add names to the local namespace that are unknowable at compile time. Therefore, if a function contains function definitions or **`lambda`** expressions with free variables, the compiler will flag this by raising a **`SyntaxError`** exception.

To make the preceding explanation a bit clearer, here’s an example:

```
x = 1
def f():
    # The next line is a syntax error
    exec 'x=2'
    def g():
        return x
```

Line 4 containing the `exec` statement is a syntax error, since `exec` would define a new local variable named `x` whose value should be accessed by **`g()`**.

This shouldn't be much of a limitation, since `exec` is rarely used in most Python code (and when it is used, it's often a sign of a poor design anyway).

Compatibility concerns have led to nested scopes being introduced gradually; in Python 2.1, they aren't enabled by default, but can be turned on within a module by using a future statement as described in **PEP 236** [https://peps.python.org/pep-0236/]. (See the following section for further discussion of **PEP 236** [https://peps.python.org/pep-0236/].) In Python 2.2, nested scopes will become the default and there will be no way to turn them off, but users will have had all of 2.1's lifetime to fix any breakage resulting from their introduction.

See also

PEP 227 [https://peps.python.org/pep-0227/] - Statically Nested Scopes

Written and implemented by Jeremy Hylton.

PEP 236: __future__ Directives

The reaction to nested scopes was widespread concern about the dangers of breaking code with the 2.1 release, and it was strong enough to make the Pythoneers take a more conservative approach. This approach consists of introducing a convention for enabling optional functionality in release N that will become compulsory in release N+1.

The syntax uses a `from...import` statement using the reserved module name `__future__`. Nested scopes can be enabled by the following statement:

```
from __future__ import nested_scopes
```

While it looks like a normal `import` statement, it’s not; there are strict rules on where such a future statement can be put. They can only be at the top of a module, and must precede any Python code or regular **`import`** statements. This is because such statements can affect how the Python bytecode compiler parses code and generates bytecode, so they must precede any statement that will result in bytecodes being produced.

See also

PEP 236 [https://peps.python.org/pep-0236/] - Back to the `__future__`

Written by Tim Peters, and primarily implemented by Jeremy Hylton.

PEP 207: Rich Comparisons

In earlier versions, Python’s support for implementing comparisons on user-defined classes and extension types was quite simple. Classes could implement a **`__cmp__()`** method that was given two instances of a class, and could only return 0 if they were equal or +1 or -1 if they weren’t; the method couldn’t raise an exception or return anything other than a Boolean value. Users of Numeric Python often found this model too weak and restrictive, because in the number-crunching programs that numeric Python is used for, it would be more useful to be able to perform elementwise comparisons of two matrices, returning a matrix containing the results of a given comparison for each element. If the two matrices are of different sizes, then the compare has to be able to raise an exception to signal the error.

In Python 2.1, rich comparisons were added in order to support this need. Python classes can now individually overload each of the `<`, `<=`, `>`, `>=`, `==`, and `!=` operations. The new magic method names are:

Operation	**Method name**
`<`	`__lt__()`
`<=`	`__le__()`
`>`	`__gt__()`
`>=`	`__ge__()`
`==`	`__eq__()`
`!=`	`__ne__()`

(The magic methods are named after the corresponding Fortran operators `.LT.`. `.LE.`, &c. Numeric programmers are almost certainly quite familiar with these names and will find them easy to remember.)

Each of these magic methods is of the form `method(self, other)`, where `self` will be the object on the left-hand side of the operator, while `other` will be the object on the right-hand side. For example, the expression `A < B` will cause `A.__lt__(B)` to be called.

Each of these magic methods can return anything at all: a Boolean, a matrix, a list, or any other Python object. Alternatively they can raise an exception if the comparison is impossible, inconsistent, or otherwise meaningless.

The built-in `cmp(A,B)` function can use the rich comparison machinery, and now accepts an optional argument specifying which comparison operation to use; this is given as one of the strings `"<"`, `"<="`, `">"`, `">="`, `"=="`, or `"!="`. If called without the optional third argument, **`cmp()`** will only return -1, 0, or +1 as in previous versions of Python; otherwise it will call the appropriate method and can return any Python object.

There are also corresponding changes of interest to C programmers; there's a new slot `tp_richcmp` in type objects and an API for performing a given rich comparison. I won't cover the C API here, but will refer you to **PEP 207** [https://peps.python.org/pep-0207/], or to 2.1's C API documentation, for the full list of related functions.

See also

PEP 207 [https://peps.python.org/pep-0207/] - Rich Comparisons

Written by Guido van Rossum, heavily based on earlier work by David Ascher, and implemented by Guido van Rossum.

PEP 230: Warning Framework

Over its 10 years of existence, Python has accumulated a certain number of obsolete modules and features along the way. It’s difficult to know when a feature is safe to remove, since there’s no way of knowing how much code uses it — perhaps no programs depend on the feature, or perhaps many do. To enable removing old features in a more structured way, a warning framework was added. When the Python developers want to get rid of a feature, it will first trigger a warning in the next version of Python. The following Python version can then drop the feature, and users will have had a full release cycle to remove uses of the old feature.

Python 2.1 adds the warning framework to be used in this scheme. It adds a `warnings` module that provide functions to issue warnings, and to filter out warnings that you don’t want to be displayed. Third-party modules can also use this framework to deprecate old features that they no longer wish to support.

For example, in Python 2.1 the **`regex`** module is deprecated, so importing it causes a warning to be printed:

```
>>> import regex
__main__:1: DeprecationWarning: the regex module
         is deprecated; please use the re module
>>>
```

Warnings can be issued by calling the `warnings.warn()` function:

```
warnings.warn("feature X no longer supported")
```

The first parameter is the warning message; an additional optional parameters can be used to specify a particular warning category.

Filters can be added to disable certain warnings; a regular expression pattern can be applied to the message or to the module name in order to suppress a warning. For example, you may have a program that uses the **`regex`** module and not want to spare the time to convert it to use the `re` module right now. The warning can be suppressed by calling

```
import warnings
warnings.filterwarnings(action = 'ignore',
                        message='.*regex module is deprecated',
                        category=DeprecationWarning,
                        module = '__main__')
```

This adds a filter that will apply only to warnings of the class `DeprecationWarning` triggered in the `__main__` module, and applies a regular expression to only match the message about the **`regex`** module being deprecated, and will cause such warnings to be ignored. Warnings can also be printed only once, printed every time the offending code is executed, or turned into exceptions that will cause the program to stop (unless the exceptions are caught in the usual way, of course).

Functions were also added to Python’s C API for issuing warnings; refer to PEP 230 or to Python’s API documentation for the details.

See also

PEP 5 [https://peps.python.org/pep-0005/] - Guidelines for Language Evolution

Written by Paul Prescod, to specify procedures to be followed when removing old features from Python. The policy described in this PEP hasn’t been officially adopted, but the eventual policy probably won’t be too different from Prescod’s proposal.

PEP 230 [https://peps.python.org/pep-0230/] - Warning Framework

Written and implemented by Guido van Rossum.

PEP 229: New Build System

When compiling Python, the user had to go in and edit the `Modules/Setup` file in order to enable various additional modules; the default set is relatively small and limited to modules that compile on most Unix platforms. This means that on Unix platforms with many more features, most notably Linux, Python installations often don’t contain all useful modules they could.

Python 2.0 added the Distutils, a set of modules for distributing and installing extensions. In Python 2.1, the Distutils are used to compile much of the standard library of extension modules, autodetecting which ones are supported on the current machine. It’s hoped that this will make Python installations easier and more featureful.

Instead of having to edit the `Modules/Setup` file in order to enable modules, a `setup.py` script in the top directory of the Python source distribution is run at build time, and attempts to discover which modules can be enabled by examining the modules and header files on the system. If a module is configured in `Modules/Setup`, the `setup.py` script won’t attempt to compile that module and will defer to the `Modules/Setup` file’s contents. This provides a way to specific any strange command-line flags or libraries that are required for a specific platform.

In another far-reaching change to the build mechanism, Neil Schemenauer restructured things so Python now uses a single makefile that isn’t recursive, instead of makefiles in the top directory and in each of the `Python/`, `Parser/`, `Objects/`, and `Modules/` subdirectories. This makes building Python faster and also makes hacking the Makefiles clearer and simpler.

See also

PEP 229 [https://peps.python.org/pep-0229/] - Using Distutils to Build Python
Written and implemented by A.M. Kuchling.

PEP 205: Weak References

Weak references, available through the `weakref` module, are a minor but useful new data type in the Python programmer’s toolbox.

Storing a reference to an object (say, in a dictionary or a list) has the side effect of keeping that object alive forever. There are a few specific cases where this behaviour is undesirable, object caches being the most common one, and another being circular references in data structures such as trees.

For example, consider a memoizing function that caches the results of another function `f(x)` by storing the function’s argument and its result in a dictionary:

```
_cache = {}
def memoize(x):
    if _cache.has_key(x):
        return _cache[x]
    retval = f(x)

    # Cache the returned object
    _cache[x] = retval

    return retval
```

This version works for simple things such as integers, but it has a side effect; the `_cache` dictionary holds a reference to the return values, so they'll never be deallocated until the Python process exits and cleans up. This isn't very noticeable for integers, but if **`f()`** returns an object, or a data structure that takes up a lot of memory, this can be a problem.

Weak references provide a way to implement a cache that won't keep objects alive beyond their time. If an object is only accessible through weak references, the object will be deallocated and the weak references will now indicate that the object it referred to no longer exists. A weak reference to an object *obj* is created by calling `wr = weakref.ref(obj)`. The object being referred to is returned by calling the weak reference as if it were a function: `wr()`. It will return the referenced object, or `None` if the object no longer exists.

This makes it possible to write a **`memoize()`** function whose cache doesn't keep objects alive, by storing weak references in the cache.

```
_cache = {}
def memoize(x):
    if _cache.has_key(x):
        obj = _cache[x]()
        # If weak reference object still exists,
        # return it
        if obj is not None: return obj

    retval = f(x)
    # Cache a weak reference
    _cache[x] = weakref.ref(retval)

    return retval
```

The `weakref` module also allows creating proxy objects which behave like weak references — an object referenced only by proxy objects is deallocated – but instead of requiring an explicit call to retrieve the object, the proxy transparently forwards all operations to the object as long as the object still exists. If the object is deallocated, attempting to use a proxy will cause a **`weakref.ReferenceError`** exception to be raised.

```
proxy = weakref.proxy(obj)
proxy.attr   # Equivalent to obj.attr
proxy.meth() # Equivalent to obj.meth()
del obj
proxy.attr   # raises weakref.ReferenceError
```

See also

PEP 205 [https://peps.python.org/pep-0205/] - Weak References
Written and implemented by Fred L. Drake, Jr.

PEP 232: Function Attributes

In Python 2.1, functions can now have arbitrary information attached to them. People were often using docstrings to hold information about functions and methods, because the `__doc__` attribute was the only way of attaching any information to a function. For example, in the Zope web application server, functions are marked as safe for public access by having a docstring, and in John Aycock’s SPARK parsing framework, docstrings hold parts of the BNF grammar to be parsed. This overloading is unfortunate, since docstrings are really intended to hold a function's documentation; for example, it means you can't properly document functions intended for private use in Zope.

Arbitrary attributes can now be set and retrieved on functions using the regular Python syntax:

```
def f(): pass

f.publish = 1
f.secure = 1
f.grammar = "A ::= B (C D)*"
```

The dictionary containing attributes can be accessed as the function's `__dict__`. Unlike the `__dict__` attribute of class instances, in functions you can actually assign a new dictionary to `__dict__`, though the new value is restricted to a regular Python dictionary; you *can't* be tricky and set it to a **`UserDict`** instance, or any other random object that behaves like a mapping.

See also

PEP 232 [https://peps.python.org/pep-0232/] - Function Attributes
Written and implemented by Barry Warsaw.

PEP 235: Importing Modules on Case-Insensitive Platforms

Some operating systems have filesystems that are case-insensitive, MacOS and Windows being the primary examples; on these systems, it's impossible to distinguish the filenames `FILE.PY` and `file.py`, even though they do store the file’s name in its original case (they’re case-preserving, too).

In Python 2.1, the `import` statement will work to simulate case-sensitivity on case-insensitive platforms. Python will now search for the first case-sensitive match by default, raising an `ImportError` if no such file is found, so `import file` will not import a module named `FILE.PY`. Case-insensitive matching can be requested by setting the `PYTHONCASEOK` environment variable before starting the Python interpreter.

PEP 217: Interactive Display Hook

When using the Python interpreter interactively, the output of commands is displayed using the built-in `repr()` function. In Python 2.1, the variable `sys.displayhook()` can be set to a callable object which will be called instead of `repr()`. For example, you can set it to a special pretty-printing function:

```
>>> # Create a recursive data structure
... L = [1,2,3]
>>> L.append(L)
>>> L # Show Python's default output
[1, 2, 3, [...]]
>>> # Use pprint.pprint() as the display function
... import sys, pprint
>>> sys.displayhook = pprint.pprint
>>> L
[1, 2, 3,  <Recursion on list with id=135143996>]
>>>
```

See also

PEP 217 [https://peps.python.org/pep-0217/] - Display Hook for Interactive Use

Written and implemented by Moshe Zadka.

PEP 208: New Coercion Model

How numeric coercion is done at the C level was significantly modified. This will only affect the authors of C extensions to Python, allowing them more flexibility in writing extension types that support numeric operations.

Extension types can now set the type flag `Py_TPFLAGS_CHECKTYPES` in their `PyTypeObject` structure to indicate that they support the new coercion model. In such extension types, the numeric slot functions can no longer assume that they'll be passed two arguments of the same type; instead they may be passed two arguments of differing types, and can then perform their own internal coercion. If the slot function is passed a type it can't handle, it can indicate the failure by returning a reference to the `Py_NotImplemented` singleton value. The numeric functions of the other type will then be tried, and perhaps they can handle the operation; if the other type also returns `Py_NotImplemented`, then a `TypeError` will be raised. Numeric methods written in Python can also return `Py_NotImplemented`, causing the interpreter to act as if the method did not exist (perhaps raising a `TypeError`, perhaps trying another object's numeric methods).

See also

PEP 208 [https://peps.python.org/pep-0208/] - Reworking the Coercion Model

Written and implemented by Neil Schemenauer, heavily based upon earlier work by Marc-André Lemburg. Read this to understand the fine points of how numeric operations will now be processed at the C level.

PEP 241: Metadata in Python Packages

A common complaint from Python users is that there’s no single catalog of all the Python modules in existence. T. Middleton’s Vaults of Parnassus at `www.vex.net/parnassus/` (retired in February 2009, available in the Internet Archive Wayback Machine [https://web.archive.org/web/20090130140102/http://www.vex.net/parnassus/]) was the largest catalog of Python modules, but registering software at the Vaults is optional, and many people did not bother.

As a first small step toward fixing the problem, Python software packaged using the Distutils **sdist** command will include a file named `PKG-INFO` containing information about the package such as its name, version, and author (metadata, in cataloguing terminology). **PEP 241** [https://peps.python.org/pep-0241/] contains the full list of fields that can be present in the `PKG-INFO` file. As people began to package their software using Python 2.1, more and more packages will include metadata, making it possible to build automated cataloguing systems and experiment with them. With the result experience, perhaps it’ll be possible to design a really good catalog and then build support for it into Python 2.2. For example, the Distutils **sdist** and **bdist_*** commands could support an `upload` option that would automatically upload your package to a catalog server.

You can start creating packages containing `PKG-INFO` even if you’re not using Python 2.1, since a new release of the Distutils will be made for users of earlier Python versions.

Version 1.0.2 of the Distutils includes the changes described in **PEP 241** [https://peps.python.org/pep-0241/], as well as various bugfixes and enhancements. It will be available from the Distutils SIG at https://www.python.org/community/sigs/current/distutils-sig/.

See also

PEP 241 [https://peps.python.org/pep-0241/] - Metadata for Python Software Packages
Written and implemented by A.M. Kuchling.

PEP 243 [https://peps.python.org/pep-0243/] - Module Repository Upload Mechanism
Written by Sean Reifschneider, this draft PEP describes a proposed mechanism for uploading Python packages to a central server.

New and Improved Modules

- Ka-Ping Yee contributed two new modules: **`inspect.py`**, a module for getting information about live Python code, and **`pydoc.py`**, a module for interactively converting docstrings to HTML or text. As a bonus, `Tools/scripts/pydoc`, which is now automatically installed, uses **`pydoc.py`** to display documentation given a Python module, package, or class name. For example, `pydoc xml.dom` displays the following:

```
Python Library Documentation: package xml.dom in xml

NAME
    xml.dom - W3C Document Object Model implementation for
Python.
```

```
FILE
    /usr/local/lib/python2.1/xml/dom/__init__.pyc

DESCRIPTION
    The Python mapping of the Document Object Model is
documented in the
    Python Library Reference in the section on the xml.dom
package.

    This package contains the following modules:
      ...
```

`pydoc` also includes a Tk-based interactive help browser. `pydoc` quickly becomes addictive; try it out!

- Two different modules for unit testing were added to the standard library. The **`doctest`** module, contributed by Tim Peters, provides a testing framework based on running embedded examples in docstrings and comparing the results against the expected output. PyUnit, contributed by Steve Purcell, is a unit testing framework inspired by JUnit, which was in turn an adaptation of Kent Beck's Smalltalk testing framework. See https://pyunit.sourceforge.net/ for more information about PyUnit.

- The **`difflib`** module contains a class, **`SequenceMatcher`**, which compares two sequences and computes the changes required to transform one sequence into the other. For example, this module can be used to write a tool similar to the Unix **diff** program, and in fact the sample program `Tools/scripts/ndiff.py` demonstrates how to write such a script.

- **`curses.panel`**, a wrapper for the panel library, part of ncurses and of SYSV curses, was contributed by Thomas Gellekum. The panel library provides windows with the additional feature of depth. Windows can be moved higher or lower in the depth ordering, and the panel library figures out where panels overlap and which sections are visible.

- The PyXML package has gone through a few releases since Python 2.0, and Python 2.1 includes an updated version of the `xml` package. Some of the noteworthy changes include support for Expat 1.2 and later versions, the ability for Expat parsers to handle files in any encoding supported by Python, and various bugfixes for SAX, DOM, and the **`minidom`** module.

- Ping also contributed another hook for handling uncaught exceptions. `sys.excepthook()` can be set to a callable object. When an exception isn't caught by any `try`…`except` blocks, the exception will be passed to `sys.excepthook()`, which can then do whatever it likes. At the Ninth Python Conference, Ping demonstrated an application for this hook: printing an extended traceback that not only lists the stack frames, but also lists the function arguments and the local variables for each frame.

- Various functions in the `time` module, such as `asctime()` and `localtime()`, require a floating-point argument containing the time in seconds since the epoch. The most common use of these functions is to work with the current time, so the floating-point argument has been made optional; when a value isn't provided, the current time will be used. For example, log file entries usually need a string containing the current time; in Python 2.1, `time.asctime()` can be used, instead of the lengthier `time.asctime(time.localtime(time.time()))` that was previously required.

This change was proposed and implemented by Thomas Wouters.

- The **`ftplib`** module now defaults to retrieving files in passive mode, because passive mode is more likely to work from behind a firewall. This request came from the Debian bug tracking system, since other Debian packages use **`ftplib`** to retrieve files and then don't work from behind a firewall. It's deemed unlikely that this will cause problems for anyone, because Netscape defaults to passive mode and few people complain, but if passive mode is unsuitable for your application or network setup, call `set_pasv(0)` on FTP objects to disable passive mode.

- Support for raw socket access has been added to the **`socket`** module, contributed by Grant Edwards.

- The **`pstats`** module now contains a simple interactive statistics browser for displaying timing profiles for Python programs, invoked when the module is run as a script. Contributed by Eric S. Raymond.

- A new implementation-dependent function, `sys._getframe([depth])`, has been added to return a given frame object from the current call stack. **`sys._getframe()`** returns the frame at the top of the call stack; if the optional integer argument *depth* is supplied, the function returns the frame that is *depth* calls below the top of the stack. For example, `sys._getframe(1)` returns the caller's frame object.

  This function is only present in CPython, not in Jython or the .NET implementation. Use it for debugging, and resist the temptation to put it into production code.

Other Changes and Fixes

There were relatively few smaller changes made in Python 2.1 due to the shorter release cycle. A search through the CVS change logs turns up 117 patches applied, and 136 bugs fixed; both figures are likely to be underestimates. Some of the more notable changes are:

- A specialized object allocator is now optionally available, that should be faster than the system `malloc()` and have less memory overhead. The allocator uses C's `malloc()` function to get large pools of memory, and then fulfills smaller memory requests from these pools. It can be enabled by providing the `--with-pymalloc` option to the **configure** script; see `Objects/obmalloc.c` for the implementation details.

  Authors of C extension modules should test their code with the object allocator enabled, because some incorrect code may break, causing core dumps at runtime. There are a bunch of memory allocation functions in Python's C API that have previously been just aliases for the C library's `malloc()` and `free()`, meaning that if you accidentally called mismatched functions, the error wouldn't be noticeable. When the object allocator is enabled, these functions aren't aliases of `malloc()` and `free()` any more, and calling the wrong function to free memory will get you a core dump. For example, if memory was allocated using `PyMem_New`, it has to be freed using `PyMem_Del()`, not `free()`. A few modules included with Python fell afoul of this and had to be fixed; doubtless there are more third-party modules that will have the same problem.

  The object allocator was contributed by Vladimir Marangozov.

- The speed of line-oriented file I/O has been improved because people often complain about its lack of speed, and because it’s often been used as a naïve benchmark. The `readline()` method of file objects has therefore been rewritten to be much faster. The exact amount of the speedup will vary from platform to platform depending on how slow the C library’s **`getc()`** was, but is around 66%, and potentially much faster on some particular operating systems. Tim Peters did much of the benchmarking and coding for this change, motivated by a discussion in comp.lang.python.

  A new module and method for file objects was also added, contributed by Jeff Epler. The new method, **`xreadlines()`**, is similar to the existing **`xrange()`** built-in. **`xreadlines()`** returns an opaque sequence object that only supports being iterated over, reading a line on every iteration but not reading the entire file into memory as the existing **`readlines()`** method does. You’d use it like this:

  ```
  for line in sys.stdin.xreadlines():
      # ... do something for each line ...
      ...
  ```

  For a fuller discussion of the line I/O changes, see the python-dev summary for January 1–15, 2001 at https://mail.python.org/pipermail/python-dev/2001-January/.

- A new method, `popitem()`, was added to dictionaries to enable destructively iterating through the contents of a dictionary; this can be faster for large dictionaries because there’s no need to construct a list containing all the keys or values. `D.popitem()` removes a random `(key, value)` pair from the dictionary `D` and returns it as a 2-tuple. This was implemented mostly by Tim Peters and

Guido van Rossum, after a suggestion and preliminary patch by Moshe Zadka.

- Modules can now control which names are imported when `from module import *` is used, by defining an `__all__` attribute containing a list of names that will be imported. One common complaint is that if the module imports other modules such as **sys** or **string**, `from module import *` will add them to the importing module's namespace. To fix this, simply list the public names in `__all__`:

  ```
  # List public names
  __all__ = ['Database', 'open']
  ```

  A stricter version of this patch was first suggested and implemented by Ben Wolfson, but after some python-dev discussion, a weaker final version was checked in.

- Applying **`repr()`** to strings previously used octal escapes for non-printable characters; for example, a newline was `'\012'`. This was a vestigial trace of Python's C ancestry, but today octal is of very little practical use. Ka-Ping Yee suggested using hex escapes instead of octal ones, and using the `\n`, `\t`, `\r` escapes for the appropriate characters, and implemented this new formatting.

- Syntax errors detected at compile-time can now raise exceptions containing the filename and line number of the error, a pleasant side effect of the compiler reorganization done by Jeremy Hylton.

- C extensions which import other modules have been changed to use **`PyImport_ImportModule()`**, which means that they will use any import hooks that have been installed. This is also encouraged for third-party extensions that need to import some other module from C code.

- The size of the Unicode character database was shrunk by another 340K thanks to Fredrik Lundh.
- Some new ports were contributed: MacOS X (by Steven Majewski), Cygwin (by Jason Tishler); RISCOS (by Dietmar Schwertberger); Unixware 7 (by Billy G. Allie).

And there’s the usual list of minor bugfixes, minor memory leaks, docstring edits, and other tweaks, too lengthy to be worth itemizing; see the CVS logs for the full details if you want them.

Acknowledgements

The author would like to thank the following people for offering suggestions on various drafts of this article: Graeme Cross, David Goodger, Jay Graves, Michael Hudson, Marc-André Lemburg, Fredrik Lundh, Neil Schemenauer, Thomas Wouters.

What’s New in Python 2.0

Author:: A.M. Kuchling and Moshe Zadka

Introduction

A new release of Python, version 2.0, was released on October 16, 2000. This article covers the exciting new features in 2.0, highlights some other useful changes, and points out a few incompatible changes that may require rewriting code.

Python’s development never completely stops between releases, and a steady flow of bug fixes and improvements are always being submitted. A host of minor fixes, a few optimizations, additional docstrings, and better error messages went into 2.0; to list them all would be impossible, but they’re certainly significant. Consult the publicly available CVS logs if you want to see the full list. This progress is due to the five developers working for PythonLabs are now getting paid to spend their days fixing bugs, and also due to the improved communication resulting from moving to SourceForge.

What About Python 1.6?

Python 1.6 can be thought of as the Contractual Obligations Python release. After the core development team left CNRI in May 2000, CNRI requested that a 1.6 release be created, containing all the work on Python that had been performed at CNRI. Python 1.6 therefore represents the state of the CVS tree as of May 2000, with the most significant new feature being Unicode support. Development continued after May, of course, so the 1.6 tree received a few fixes to ensure that it's forward-compatible with Python 2.0. 1.6 is therefore part of Python's evolution, and not a side branch.

So, should you take much interest in Python 1.6? Probably not. The 1.6final and 2.0beta1 releases were made on the same day (September 5, 2000), the plan being to finalize Python 2.0 within a month or so. If you have applications to maintain, there seems little point in breaking things by moving to 1.6, fixing them, and then having another round of breakage within a month by moving to 2.0; you're better off just going straight to 2.0. Most of the really interesting features described in this document are only in 2.0, because a lot of work was done between May and September.

New Development Process

The most important change in Python 2.0 may not be to the code at all, but to how Python is developed: in May 2000 the Python developers began using the tools made available by SourceForge for storing source code, tracking bug reports, and managing the queue of patch submissions. To report bugs or submit patches for Python 2.0, use the bug tracking and patch manager tools available from Python's project page, located at https://sourceforge.net/projects/python/.

The most important of the services now hosted at SourceForge is the Python CVS tree, the version-controlled repository containing the source code for Python. Previously, there were roughly 7 or so people who had write access to the CVS tree, and all patches had to be inspected and checked in by one of the people on this short list. Obviously, this wasn't very scalable. By moving the CVS tree to SourceForge, it became possible to grant write access to more people; as of September 2000 there were 27 people able to check in changes, a fourfold increase. This makes possible large-scale changes that wouldn’t be attempted if they’d have to be filtered through the small group of core developers. For example, one day Peter Schneider-Kamp took it into his head to drop K&R C compatibility and convert the C source for Python to ANSI C. After getting approval on the python-dev mailing list, he launched into a flurry of checkins that lasted about a week, other developers joined in to help, and the job was done. If there were only 5 people with write access, probably that task would have been viewed as “nice, but not worth the time and effort needed” and it would never have gotten done.

The shift to using SourceForge’s services has resulted in a remarkable increase in the speed of development. Patches now get submitted, commented on, revised by people other than the original submitter, and bounced back and forth between people until the patch is deemed worth checking in. Bugs are tracked in one central location and can be assigned to a specific person for fixing, and we can count the number of open bugs to measure progress. This didn’t come without a cost: developers now have more e-mail to deal with, more mailing lists to follow, and special tools had to be written for the new environment. For example, SourceForge sends default patch and bug notification e-mail messages that are completely unhelpful, so Ka-Ping Yee wrote an HTML screen-scraper that sends more useful messages.

The ease of adding code caused a few initial growing pains, such as code was checked in before it was ready or without getting clear agreement from the developer group. The approval process that has emerged is somewhat similar to that used by the Apache group. Developers can vote +1, +0, -0, or -1 on a patch; +1 and -1 denote acceptance or rejection, while +0 and -0 mean the developer is mostly indifferent to the change, though with a slight positive or negative slant. The most significant change from the Apache model is that the voting is essentially advisory, letting Guido van Rossum, who has Benevolent Dictator For Life status, know what the general opinion is. He can still ignore the result of a vote, and approve or reject a change even if the community disagrees with him.

Producing an actual patch is the last step in adding a new feature, and is usually easy compared to the earlier task of coming up with a good design. Discussions of new features can often explode into lengthy mailing list threads, making the discussion hard to follow, and no one can read every posting to python-dev. Therefore, a relatively formal process has been set up to write Python Enhancement Proposals (PEPs), modelled on the internet RFC process. PEPs are draft documents that describe a proposed new feature, and are continually revised until the community reaches a consensus, either accepting or rejecting the proposal. Quoting from the introduction to **PEP 1** [https://peps.python.org/pep-0001/], “PEP Purpose and Guidelines”:

> PEP stands for Python Enhancement Proposal. A PEP is a design document providing information to the Python community, or describing a new feature for Python. The PEP should provide a concise technical specification of the feature and a rationale for the feature.
>
> We intend PEPs to be the primary mechanisms for proposing new features, for collecting community input on an issue, and for documenting the design decisions that have gone into Python. The PEP author is responsible for building consensus within the community and documenting dissenting opinions.

Read the rest of **PEP 1** [https://peps.python.org/pep-0001/] for the details of the PEP editorial process, style, and format. PEPs are kept in the Python CVS tree on SourceForge, though they're not part of the Python 2.0 distribution, and are also available in HTML form from https://peps.python.org/. As of September 2000, there are 25 PEPs, ranging from **PEP 201** [https://peps.python.org/pep-0201/], "Lockstep Iteration", to PEP 225, "Elementwise/Objectwise Operators".

Unicode

The largest new feature in Python 2.0 is a new fundamental data type: Unicode strings. Unicode uses 16-bit numbers to represent characters instead of the 8-bit number used by ASCII, meaning that 65,536 distinct characters can be supported.

The final interface for Unicode support was arrived at through countless often-stormy discussions on the python-dev mailing list, and mostly implemented by Marc-André Lemburg, based on a Unicode string type implementation by Fredrik Lundh. A detailed explanation of the interface was written up as **PEP 100** [https://peps.python.org/pep-0100/], "Python Unicode Integration". This article will simply cover the most significant points about the Unicode interfaces.

In Python source code, Unicode strings are written as `u"string"`. Arbitrary Unicode characters can be written using a new escape sequence, `\u`*HHHH*, where *HHHH* is a 4-digit hexadecimal number from 0000 to FFFF. The existing `\x`*HH* escape sequence can also be used, and octal escapes can be used for characters up to U+01FF, which is represented by `\777`.

Unicode strings, just like regular strings, are an immutable sequence type. They can be indexed and sliced, but not modified in place. Unicode strings have an `encode( [encoding] )` method that returns an 8-bit string in the desired encoding. Encodings are named by strings, such as `'ascii'`, `'utf-8'`, `'iso-8859-1'`, or whatever. A codec API is defined for implementing and registering new encodings that are then available throughout a Python program. If an encoding isn't specified, the default encoding is usually 7-bit ASCII, though it can be changed for your Python installation by calling the `sys.setdefaultencoding(encoding)` function in a customized version of `site.py`.

Combining 8-bit and Unicode strings always coerces to Unicode, using the default ASCII encoding; the result of `'a' + u'bc'` is `u'abc'`.

New built-in functions have been added, and existing built-ins modified to support Unicode:

- `unichr(ch)` returns a Unicode string 1 character long, containing the character *ch*.
- `ord(u)`, where *u* is a 1-character regular or Unicode string, returns the number of the character as an integer.
- `unicode(string [, encoding]  [, errors] )` creates a Unicode string from an 8-bit string. `encoding` is a string naming the encoding to use. The `errors` parameter specifies the treatment of characters that are invalid for the current encoding; passing `'strict'` as the value causes an exception to be raised on any encoding error, while `'ignore'` causes errors to be silently ignored and `'replace'` uses U+FFFD, the official replacement character, in case of any problems.
- The `exec` statement, and various built-ins such as `eval()`, `getattr()`, and `setattr()` will also accept Unicode strings as well as regular strings. (It's possible that the process of fixing this missed some built-ins; if you find a built-in

function that accepts strings but doesn't accept Unicode strings at all, please report it as a bug.)

A new module, **`unicodedata`**, provides an interface to Unicode character properties. For example, `unicodedata.category(u'A')` returns the 2-character string 'Lu', the 'L' denoting it's a letter, and 'u' meaning that it's uppercase. `unicodedata.bidirectional(u'\u0660')` returns 'AN', meaning that U+0660 is an Arabic number.

The **`codecs`** module contains functions to look up existing encodings and register new ones. Unless you want to implement a new encoding, you'll most often use the `codecs.lookup(encoding)` function, which returns a 4-element tuple: `(encode_func, decode_func, stream_reader, stream_writer)`.

- *encode_func* is a function that takes a Unicode string, and returns a 2-tuple `(string, length)`. *string* is an 8-bit string containing a portion (perhaps all) of the Unicode string converted into the given encoding, and *length* tells you how much of the Unicode string was converted.
- *decode_func* is the opposite of *encode_func*, taking an 8-bit string and returning a 2-tuple `(ustring, length)`, consisting of the resulting Unicode string *ustring* and the integer *length* telling how much of the 8-bit string was consumed.
- *stream_reader* is a class that supports decoding input from a stream. *stream_reader(file_obj)* returns an object that supports the **`read()`**, **`readline()`**, and **`readlines()`** methods. These methods will all translate from the given encoding and return Unicode strings.
- *stream_writer*, similarly, is a class that supports encoding output to a stream. *stream_writer(file_obj)* returns an object that supports the **`write()`** and **`writelines()`** methods. These methods expect Unicode

strings, translating them to the given encoding on output.

For example, the following code writes a Unicode string into a file, encoding it as UTF-8:

```
import codecs

unistr = u'\u0660\u2000ab ...'

(UTF8_encode, UTF8_decode,
 UTF8_streamreader, UTF8_streamwriter) = codecs.lookup('UTF-8')

output = UTF8_streamwriter( open( '/tmp/output', 'wb') )
output.write( unistr )
output.close()
```

The following code would then read UTF-8 input from the file:

```
input = UTF8_streamreader( open( '/tmp/output', 'rb') )
print repr(input.read())
input.close()
```

Unicode-aware regular expressions are available through the `re` module, which has a new underlying implementation called SRE written by Fredrik Lundh of Secret Labs AB.

A `-U` command line option was added which causes the Python compiler to interpret all string literals as Unicode string literals. This is intended to be used in testing and future-proofing your Python code, since some future version of Python may drop support for 8-bit strings and provide only Unicode strings.

List Comprehensions

Lists are a workhorse data type in Python, and many programs manipulate a list at some point. Two common operations on lists are to loop over them, and either pick out the elements that meet a certain criterion, or apply some function to each element. For example, given a list of strings, you might want to pull out all the strings containing a given substring, or strip off trailing whitespace from each line.

The existing `map()` and `filter()` functions can be used for this purpose, but they require a function as one of their arguments. This is fine if there's an existing built-in function that can be passed directly, but if there isn't, you have to create a little function to do the required work, and Python's scoping rules make the result ugly if the little function needs additional information. Take the first example in the previous paragraph, finding all the strings in the list containing a given substring. You could write the following to do it:

```
# Given the list L, make a list of all strings
# containing the substring S.
sublist = filter( lambda s, substring=S:
                     string.find(s, substring) != -1,
                  L)
```

Because of Python's scoping rules, a default argument is used so that the anonymous function created by the `lambda` expression knows what substring is being searched for. List comprehensions make this cleaner:

```
sublist = [ s for s in L if string.find(s, S) != -1 ]
```

List comprehensions have the form:

```
[ expression for expr in sequence1
             for expr2 in sequence2 ...
             for exprN in sequenceN
             if condition ]
```

The **for**...**in** clauses contain the sequences to be iterated over. The sequences do not have to be the same length, because they are *not* iterated over in parallel, but from left to right; this is explained more clearly in the following paragraphs. The elements of the generated list will be the successive values of *expression*. The final **if** clause is optional; if present, *expression* is only evaluated and added to the result if *condition* is true.

To make the semantics very clear, a list comprehension is equivalent to the following Python code:

```
for expr1 in sequence1:
    for expr2 in sequence2:
    ...
        for exprN in sequenceN:
             if (condition):
                  # Append the value of
                  # the expression to the
                  # resulting list.
```

This means that when there are multiple **for**...**in** clauses, the resulting list will be equal to the product of the lengths of all the sequences. If you have two lists of length 3, the output list is 9 elements long:

```
seq1 = 'abc'
seq2 = (1,2,3)
>>> [ (x,y) for x in seq1 for y in seq2]
[('a', 1), ('a', 2), ('a', 3), ('b', 1), ('b', 2), ('b', 3),
('c', 1),
('c', 2), ('c', 3)]
```

To avoid introducing an ambiguity into Python’s grammar, if *expression* is creating a tuple, it must be surrounded with parentheses. The first list comprehension below is a syntax error, while the second one is correct:

```
# Syntax error
[ x,y for x in seq1 for y in seq2]
# Correct
[ (x,y) for x in seq1 for y in seq2]
```

The idea of list comprehensions originally comes from the functional programming language Haskell (https://www.haskell.org). Greg Ewing argued most effectively for adding them to Python and wrote the initial list comprehension patch, which was then discussed for a seemingly endless time on the python-dev mailing list and kept up-to-date by Skip Montanaro.

Augmented Assignment

Augmented assignment operators, another long-requested feature, have been added to Python 2.0. Augmented assignment operators include `+=`, `-=`, `*=`, and so forth. For example, the statement `a += 2` increments the value of the variable `a` by 2, equivalent to the slightly lengthier `a = a + 2`.

The full list of supported assignment operators is `+=`, `-=`, `*=`, `/=`, `%=`, `**=`, `&=`, `|=`, `^=`, `>>=`, and `<<=`. Python classes can override the augmented assignment operators by defining methods named **`__iadd__()`**, **`__isub__()`**, etc. For example, the following **`Number`** class stores a number and supports using += to create a new instance with an incremented value.

```
class Number:
    def __init__(self, value):
        self.value = value
    def __iadd__(self, increment):
        return Number( self.value + increment)

n = Number(5)
```

```
n += 3
print n.value
```

The **`__iadd__()`** special method is called with the value of the increment, and should return a new instance with an appropriately modified value; this return value is bound as the new value of the variable on the left-hand side.

Augmented assignment operators were first introduced in the C programming language, and most C-derived languages, such as **awk**, C++, Java, Perl, and PHP also support them. The augmented assignment patch was implemented by Thomas Wouters.

String Methods

Until now string-manipulation functionality was in the **`string`** module, which was usually a front-end for the **`strop`** module written in C. The addition of Unicode posed a difficulty for the **`strop`** module, because the functions would all need to be rewritten in order to accept either 8-bit or Unicode strings. For functions such as **`string.replace()`**, which takes 3 string arguments, that means eight possible permutations, and correspondingly complicated code.

Instead, Python 2.0 pushes the problem onto the string type, making string manipulation functionality available through methods on both 8-bit strings and Unicode strings.

```
>>> 'andrew'.capitalize()
'Andrew'
>>> 'hostname'.replace('os', 'linux')
'hlinuxtname'
>>> 'moshe'.find('sh')
2
```

One thing that hasn’t changed, a noteworthy April Fools’ joke notwithstanding, is that Python strings are immutable. Thus, the string methods return new strings, and do not modify the string on which they operate.

The old `string` module is still around for backwards compatibility, but it mostly acts as a front-end to the new string methods.

Two methods which have no parallel in pre-2.0 versions, although they did exist in JPython for quite some time, are **`startswith()`** and **`endswith()`**. `s.startswith(t)` is equivalent to `s[:len(t)] == t`, while `s.endswith(t)` is equivalent to `s[-len(t):] == t`.

One other method which deserves special mention is **`join()`**. The **`join()`** method of a string receives one parameter, a sequence of strings, and is equivalent to the **`string.join()`** function from the old `string` module, with the arguments reversed. In other words, `s.join(seq)` is equivalent to the old `string.join(seq, s)`.

Garbage Collection of Cycles

The C implementation of Python uses reference counting to implement garbage collection. Every Python object maintains a count of the number of references pointing to itself, and adjusts the count as references are created or destroyed. Once the reference count reaches zero, the object is no longer accessible, since you need to have a reference to an object to access it, and if the count is zero, no references exist any longer.

Reference counting has some pleasant properties: it’s easy to understand and implement, and the resulting implementation is portable, fairly fast, and reacts well with other libraries that implement their own memory handling schemes. The major problem with reference counting is that it sometimes doesn’t realise that objects are no longer accessible, resulting in a memory leak. This happens when there are cycles of references.

Consider the simplest possible cycle, a class instance which has a reference to itself:

```
instance = SomeClass()
instance.myself = instance
```

After the above two lines of code have been executed, the reference count of `instance` is 2; one reference is from the variable named `'instance'`, and the other is from the `myself` attribute of the instance.

If the next line of code is `del instance`, what happens? The reference count of `instance` is decreased by 1, so it has a reference count of 1; the reference in the `myself` attribute still exists. Yet the instance is no longer accessible through Python code, and it could be deleted. Several objects can participate in a cycle if they have references to each other, causing all of the objects to be leaked.

Python 2.0 fixes this problem by periodically executing a cycle detection algorithm which looks for inaccessible cycles and deletes the objects involved. A new `gc` module provides functions to perform a garbage collection, obtain debugging statistics, and tuning the collector’s parameters.

Running the cycle detection algorithm takes some time, and therefore will result in some additional overhead. It is hoped that after we’ve gotten experience with the cycle collection from using 2.0, Python 2.1 will be able to minimize the overhead with careful tuning. It’s not yet obvious how much performance is lost, because benchmarking this is tricky and depends crucially on how often the program creates and destroys objects. The detection of cycles can be disabled when Python is compiled, if you can't afford even a tiny speed penalty or suspect that the cycle collection is buggy, by specifying the **`--without-cycle-gc`** switch when running the **configure** script.

Several people tackled this problem and contributed to a solution. An early implementation of the cycle detection approach was written by Toby Kelsey. The current algorithm was suggested by Eric Tiedemann during a visit to CNRI, and Guido van Rossum and Neil Schemenauer wrote two different implementations, which were later integrated by Neil. Lots of other people offered suggestions along the way; the March 2000 archives of the python-dev mailing list contain most of the relevant discussion, especially in the threads titled “Reference cycle collection for Python” and “Finalization again”.

Other Core Changes

Various minor changes have been made to Python's syntax and built-in functions. None of the changes are very far-reaching, but they're handy conveniences.

Minor Language Changes

A new syntax makes it more convenient to call a given function with a tuple of arguments and/or a dictionary of keyword arguments. In Python 1.5 and earlier, you'd use the **`apply()`** built-in function: `apply(f, args, kw)` calls the function **`f()`** with the argument tuple *args* and the keyword arguments in the dictionary *kw*. **`apply()`** is the same in 2.0, but thanks to a patch from Greg Ewing, `f(*args, **kw)` is a shorter and clearer way to achieve the same effect. This syntax is symmetrical with the syntax for defining functions:

```
def f(*args, **kw):
    # args is a tuple of positional args,
    # kw is a dictionary of keyword args
    ...
```

The `print` statement can now have its output directed to a file-like object by following the `print` with `>> file`, similar to the redirection operator in Unix shells. Previously you'd either have to use the **`write()`** method of the file-like object, which lacks the convenience and simplicity of `print`, or you could assign a new value to `sys.stdout` and then restore the old value. For sending output to standard error, it's much easier to write this:

```
print >> sys.stderr, "Warning: action field not supplied"
```

Modules can now be renamed on importing them, using the syntax `import module as name` or `from module import name as othername`. The patch was submitted by Thomas Wouters.

A new format style is available when using the `%` operator; '%r' will insert the **`repr()`** of its argument. This was also added from symmetry considerations, this time for symmetry with the existing '%s' format style, which inserts the **`str()`** of its argument. For example, `'%r %s' % ('abc', 'abc')` returns a string containing `'abc' abc`.

Previously there was no way to implement a class that overrode Python's built-in **`in`** operator and implemented a custom version. `obj in seq` returns true if *obj* is present in the sequence *seq*; Python computes this by simply trying every index of the sequence until either *obj* is found or an **`IndexError`** is encountered. Moshe Zadka contributed a patch which adds a **`__contains__()`** magic method for providing a custom implementation for **`in`**. Additionally, new built-in objects written in C can define what **`in`** means for them via a new slot in the sequence protocol.

Earlier versions of Python used a recursive algorithm for deleting objects. Deeply nested data structures could cause the interpreter to fill up the C stack and crash; Christian Tismer rewrote the deletion logic to fix this problem. On a related note, comparing recursive objects recursed infinitely and crashed; Jeremy Hylton rewrote the code to no longer crash, producing a useful result instead. For example, after this code:

```
a = []
b = []
a.append(a)
b.append(b)
```

The comparison `a==b` returns true, because the two recursive data structures are isomorphic. See the thread “trashcan and PR#7” in the April 2000 archives of the python-dev mailing list for the discussion leading up to this implementation, and some useful relevant links. Note that comparisons can now also raise exceptions. In earlier versions of Python, a comparison operation such as `cmp(a,b)` would always produce an answer, even if a user-defined **`__cmp__()`** method encountered an error, since the resulting exception would simply be silently swallowed.

Work has been done on porting Python to 64-bit Windows on the Itanium processor, mostly by Trent Mick of ActiveState. (Confusingly, `sys.platform` is still `'win32'` on Win64 because it seems that for ease of porting, MS Visual C++ treats code as 32 bit on Itanium.) PythonWin also supports Windows CE; see the Python CE page at https://pythonce.sourceforge.net/ for more information.

Another new platform is Darwin/MacOS X; initial support for it is in Python 2.0. Dynamic loading works, if you specify “configure –with-dyld –with-suffix=.x”. Consult the README in the Python source distribution for more instructions.

An attempt has been made to alleviate one of Python’s warts, the often-confusing `NameError` exception when code refers to a local variable before the variable has been assigned a value. For example, the following code raises an exception on the `print` statement in both 1.5.2 and 2.0; in 1.5.2 a `NameError` exception is raised, while 2.0 raises a new `UnboundLocalError` exception. `UnboundLocalError` is a subclass of `NameError`, so any existing code that expects `NameError` to be raised should still work.

```
def f():
    print "i=",i
    i = i + 1
f()
```

Two new exceptions, `TabError` and `IndentationError`, have been introduced. They’re both subclasses of `SyntaxError`, and are raised when Python code is found to be improperly indented.

Changes to Built-in Functions

A new built-in, `zip(seq1, seq2, ...)`, has been added. `zip()` returns a list of tuples where each tuple contains the i-th element from each of the argument sequences. The difference between `zip()` and `map(None, seq1, seq2)` is that `map()` pads the sequences with `None` if the sequences aren’t all of the same length, while `zip()` truncates the returned list to the length of the shortest argument sequence.

The `int()` and **`long()`** functions now accept an optional “base” parameter when the first argument is a string.

`int('123', 10)` returns 123, while `int('123', 16)` returns 291. `int(123, 16)` raises a `TypeError` exception with the message “can’t convert non-string with explicit base”.

A new variable holding more detailed version information has been added to the `sys` module. `sys.version_info` is a tuple `(major, minor, micro, level, serial)` For example, in a hypothetical 2.0.1beta1, `sys.version_info` would be `(2, 0, 1, 'beta', 1)`. *level* is a string such as `"alpha"`, `"beta"`, or `"final"` for a final release.

Dictionaries have an odd new method, `setdefault(key, default)`, which behaves similarly to the existing **`get()`** method. However, if the key is missing, **`setdefault()`** both returns the value of *default* as **`get()`** would do, and also inserts it into the dictionary as the value for *key*. Thus, the following lines of code:

```
if dict.has_key( key ): return dict[key]
else:
    dict[key] = []
    return dict[key]
```

can be reduced to a single `return dict.setdefault(key, [])` statement.

The interpreter sets a maximum recursion depth in order to catch runaway recursion before filling the C stack and causing a core dump or GPF.. Previously this limit was fixed when you compiled Python, but in 2.0 the maximum recursion depth can be read and modified using `sys.getrecursionlimit()` and `sys.setrecursionlimit()`. The default value is 1000, and a rough maximum value for a given platform can be found by running a new script, `Misc/find_recursionlimit.py`.

Porting to 2.0

New Python releases try hard to be compatible with previous releases, and the record has been pretty good. However, some changes are considered useful enough, usually because they fix initial design decisions that turned out to be actively mistaken, that breaking backward compatibility can't always be avoided. This section lists the changes in Python 2.0 that may cause old Python code to break.

The change which will probably break the most code is tightening up the arguments accepted by some methods. Some methods would take multiple arguments and treat them as a tuple, particularly various list methods such as **`append()`** and **`insert()`**. In earlier versions of Python, if `L` is a list, `L.append( 1,2 )` appends the tuple `(1,2)` to the list. In Python 2.0 this causes a **`TypeError`** exception to be raised, with the message: 'append requires exactly 1 argument; 2 given'. The fix is to simply add an extra set of parentheses to pass both values as a tuple: `L.append( (1,2) )`.

The earlier versions of these methods were more forgiving because they used an old function in Python's C interface to parse their arguments; 2.0 modernizes them to use **`PyArg_ParseTuple()`**, the current argument parsing function, which provides more helpful error messages and treats multi-argument calls as errors. If you absolutely must use 2.0 but can't fix your code, you can edit `Objects/listobject.c` and define the preprocessor symbol `NO_STRICT_LIST_APPEND` to preserve the old behaviour; this isn't recommended.

Some of the functions in the **`socket`** module are still forgiving in this way. For example, `socket.connect( ('hostname', 25) )` is the correct form, passing a tuple representing an IP address, but `socket.connect('hostname', 25)` also works. **`socket.connect_ex`** and **`socket.bind`** are similarly easy-going. 2.0alpha1 tightened these functions up, but because the documentation actually used the erroneous multiple argument form, many people wrote code which would break with the stricter checking. GvR backed out the changes in the face of public reaction, so for the `socket` module, the documentation was fixed and the multiple argument form is simply marked as deprecated; it *will* be tightened up again in a future Python version.

The `\x` escape in string literals now takes exactly 2 hex digits. Previously it would consume all the hex digits following the ‘x’ and take the lowest 8 bits of the result, so `\x123456` was equivalent to `\x56`.

The `AttributeError` and `NameError` exceptions have a more friendly error message, whose text will be something like `'Spam' instance has no attribute 'eggs'` or `name 'eggs' is not defined`. Previously the error message was just the missing attribute name `eggs`, and code written to take advantage of this fact will break in 2.0.

Some work has been done to make integers and long integers a bit more interchangeable. In 1.5.2, large-file support was added for Solaris, to allow reading files larger than 2 GiB; this made the **`tell()`** method of file objects return a long integer instead of a regular integer. Some code would subtract two file offsets and attempt to use the result to multiply a sequence or slice a string, but this raised a `TypeError`. In 2.0, long integers can be used to multiply or slice a sequence, and it’ll behave as you’d intuitively expect it to; `3L * 'abc'` produces ‘abcabcabc’, and `(0,1,2,3)[2L:4L]` produces (2,3). Long integers can also be used in various contexts where previously only integers were accepted, such as in the **`seek()`** method of file objects, and in the formats supported by the `%` operator (`%d`, `%i`, `%x`, etc.). For example, `"%d" % 2L**64` will produce the string `18446744073709551616`.

The subtlest long integer change of all is that the `str()` of a long integer no longer has a trailing ‘L’ character, though `repr()` still includes it. The ‘L’ annoyed many people who wanted to print long integers that looked just like regular integers, since they had to go out of their way to chop off the character. This is no longer a problem in 2.0, but code which does `str(longval)[:-1]` and assumes the ‘L’ is there, will now lose the final digit.

Taking the `repr()` of a float now uses a different formatting precision than `str()`. `repr()` uses `%.17g` format string for C’s **`sprintf()`**, while `str()` uses `%.12g` as before. The effect is that `repr()` may occasionally show more decimal places than `str()`, for certain numbers. For example, the number 8.1 can’t be represented exactly in binary, so `repr(8.1)` is `'8.0999999999999996'`, while str(8.1) is `'8.1'`.

The `-X` command-line option, which turned all standard exceptions into strings instead of classes, has been removed; the standard exceptions will now always be classes. The **`exceptions`** module containing the standard exceptions was translated from Python to a built-in C module, written by Barry Warsaw and Fredrik Lundh.

Extending/Embedding Changes

Some of the changes are under the covers, and will only be apparent to people writing C extension modules or embedding a Python interpreter in a larger application. If you aren’t dealing with Python’s C API, you can safely skip this section.

The version number of the Python C API was incremented, so C extensions compiled for 1.5.2 must be recompiled in order to work with 2.0. On Windows, it’s not possible for Python 2.0 to import a third party extension built for Python

1.5.x due to how Windows DLLs work, so Python will raise an exception and the import will fail.

Users of Jim Fulton’s ExtensionClass module will be pleased to find out that hooks have been added so that ExtensionClasses are now supported by **`isinstance()`** and **`issubclass()`**. This means you no longer have to remember to write code such as `if type(obj) == myExtensionClass`, but can use the more natural `if isinstance(obj, myExtensionClass)`.

The `Python/importdl.c` file, which was a mass of #ifdefs to support dynamic loading on many different platforms, was cleaned up and reorganised by Greg Stein. `importdl.c` is now quite small, and platform-specific code has been moved into a bunch of `Python/dynload_*.c` files. Another cleanup: there were also a number of `my*.h` files in the Include/ directory that held various portability hacks; they’ve been merged into a single file, `Include/pyport.h`.

Vladimir Marangozov’s long-awaited malloc restructuring was completed, to make it easy to have the Python interpreter use a custom allocator instead of C’s standard **`malloc()`**. For documentation, read the comments in `Include/pymem.h` and `Include/objimpl.h`. For the lengthy discussions during which the interface was hammered out, see the web archives of the ‘patches’ and ‘python-dev’ lists at python.org.

Recent versions of the GUSI development environment for MacOS support POSIX threads. Therefore, Python’s POSIX threading support now works on the Macintosh. Threading support using the user-space GNU `pth` library was also contributed.

Threading support on Windows was enhanced, too. Windows supports thread locks that use kernel objects only in case of contention; in the common case when there’s no contention, they use simpler functions which are an order of magnitude faster. A threaded version of Python 1.5.2 on NT is twice as slow as an unthreaded version; with the 2.0 changes, the difference is only 10%. These improvements were contributed by Yakov Markovitch.

Python 2.0’s source now uses only ANSI C prototypes, so compiling Python now requires an ANSI C compiler, and can no longer be done using a compiler that only supports K&R C.

Previously the Python virtual machine used 16-bit numbers in its bytecode, limiting the size of source files. In particular, this affected the maximum size of literal lists and dictionaries in Python source; occasionally people who are generating Python code would run into this limit. A patch by Charles G. Waldman raises the limit from `2**16` to `2**32`.

Three new convenience functions intended for adding constants to a module’s dictionary at module initialization time were added: `PyModule_AddObject()`, `PyModule_AddIntConstant()`, and `PyModule_AddStringConstant()`. Each of these functions takes a module object, a null-terminated C string containing the name to be added, and a third argument for the value to be assigned to the name. This third argument is, respectively, a Python object, a C long, or a C string.

A wrapper API was added for Unix-style signal handlers. `PyOS_getsig()` gets a signal handler and `PyOS_setsig()` will set a new handler.

Distutils: Making Modules Easy to Install

Before Python 2.0, installing modules was a tedious affair – there was no way to figure out automatically where Python is installed, or what compiler options to use for extension modules. Software authors had to go through an arduous ritual of editing Makefiles and configuration files, which only really work on Unix and leave Windows and MacOS unsupported. Python users faced wildly differing installation instructions which varied between different extension packages, which made administering a Python installation something of a chore.

The SIG for distribution utilities, shepherded by Greg Ward, has created the Distutils, a system to make package installation much easier. They form the `distutils` package, a new part of Python's standard library. In the best case, installing a Python module from source will require the same steps: first you simply mean unpack the tarball or zip archive, and the run "`python setup.py install`". The platform will be automatically detected, the compiler will be recognized, C extension modules will be compiled, and the distribution installed into the proper directory. Optional command-line arguments provide more control over the installation process, the distutils package offers many places to override defaults – separating the build from the install, building or installing in non-default directories, and more.

In order to use the Distutils, you need to write a `setup.py` script. For the simple case, when the software contains only .py files, a minimal `setup.py` can be just a few lines long:

```
from distutils.core import setup
setup (name = "foo", version = "1.0",
       py_modules = ["module1", "module2"])
```

The `setup.py` file isn't much more complicated if the software consists of a few packages:

```
from distutils.core import setup
setup (name = "foo", version = "1.0",
       packages = ["package", "package.subpackage"])
```

A C extension can be the most complicated case; here’s an example taken from the PyXML package:

```
from distutils.core import setup, Extension

expat_extension = Extension('xml.parsers.pyexpat',
     define_macros = [('XML_NS', None)],
     include_dirs = [ 'extensions/expat/xmltok',
                      'extensions/expat/xmlparse' ],
     sources = [ 'extensions/pyexpat.c',
                 'extensions/expat/xmltok/xmltok.c',
                 'extensions/expat/xmltok/xmlrole.c', ]
       )
setup (name = "PyXML", version = "0.5.4",
       ext_modules =[ expat_extension ] )
```

The Distutils can also take care of creating source and binary distributions. The “sdist” command, run by “`python setup.py sdist`’, builds a source distribution such as `foo-1.0.tar.gz`. Adding new commands isn’t difficult, “bdist_rpm” and “bdist_wininst” commands have already been contributed to create an RPM distribution and a Windows installer for the software, respectively. Commands to create other distribution formats such as Debian packages and Solaris `.pkg` files are in various stages of development.

All this is documented in a new manual, *Distributing Python Modules*, that joins the basic set of Python documentation.

XML Modules

Python 1.5.2 included a simple XML parser in the form of the **`xmllib`** module, contributed by Sjoerd Mullender. Since 1.5.2’s release, two different interfaces for processing XML have become common: SAX2 (version 2 of the Simple API for XML) provides an event-driven interface with some similarities to **`xmllib`**, and the DOM (Document Object Model) provides a tree-based interface, transforming an XML document into a tree of nodes that can be traversed and modified. Python 2.0 includes a SAX2 interface and a stripped-down DOM interface as part of the **`xml`** package. Here we will give a brief overview of these new interfaces; consult the Python documentation or the source code for complete details. The Python XML SIG is also working on improved documentation.

SAX2 Support

SAX defines an event-driven interface for parsing XML. To use SAX, you must write a SAX handler class. Handler classes inherit from various classes provided by SAX, and override various methods that will then be called by the XML parser. For example, the **`startElement()`** and **`endElement()`** methods are called for every starting and end tag encountered by the parser, the **`characters()`** method is called for every chunk of character data, and so forth.

The advantage of the event-driven approach is that the whole document doesn’t have to be resident in memory at any one time, which matters if you are processing really huge documents. However, writing the SAX handler class can get very complicated if you’re trying to modify the document structure in some elaborate way.

For example, this little example program defines a handler that prints a message for every starting and ending tag, and then parses the file `hamlet.xml` using it:

```
from xml import sax

class SimpleHandler(sax.ContentHandler):
    def startElement(self, name, attrs):
        print 'Start of element:', name, attrs.keys()

    def endElement(self, name):
        print 'End of element:', name

# Create a parser object
parser = sax.make_parser()

# Tell it what handler to use
handler = SimpleHandler()
parser.setContentHandler( handler )

# Parse a file!
parser.parse( 'hamlet.xml' )
```

For more information, consult the Python documentation, or the XML HOWTO at https://pyxml.sourceforge.net/topics/howto/xml-howto.html.

DOM Support

The Document Object Model is a tree-based representation for an XML document. A top-level **`Document`** instance is the root of the tree, and has a single child which is the top-level **`Element`** instance. This **`Element`** has children nodes representing character data and any sub-elements, which may have further children of their own, and so forth. Using the DOM you can traverse the resulting tree any way you like, access element and attribute values, insert and delete nodes, and convert the tree back into XML.

The DOM is useful for modifying XML documents, because you can create a DOM tree, modify it by adding new nodes or rearranging subtrees, and then produce a new XML document as output. You can also construct a DOM tree manually and convert it to XML, which can be a more flexible way of producing XML output than simply writing `<tag1>`...`</tag1>` to a file.

The DOM implementation included with Python lives in the [`xml.dom.minidom`] module. It’s a lightweight implementation of the Level 1 DOM with support for XML namespaces. The **`parse()`** and **`parseString()`** convenience functions are provided for generating a DOM tree:

```
from xml.dom import minidom
doc = minidom.parse('hamlet.xml')
```

`doc` is a **`Document`** instance. **`Document`**, like all the other DOM classes such as **`Element`** and **`Text`**, is a subclass of the **`Node`** base class. All the nodes in a DOM tree therefore support certain common methods, such as **`toxml()`** which returns a string containing the XML representation of the node and its children. Each class also has special methods of its own; for example, **`Element`** and **`Document`** instances have a method to find all child elements with a given tag name. Continuing from the previous 2-line example:

```
perslist = doc.getElementsByTagName( 'PERSONA' )
print perslist[0].toxml()
print perslist[1].toxml()
```

For the *Hamlet* XML file, the above few lines output:

```
<PERSONA>CLAUDIUS, king of Denmark. </PERSONA>
<PERSONA>HAMLET, son to the late, and nephew to the present
king.</PERSONA>
```

The root element of the document is available as `doc.documentElement`, and its children can be easily modified by deleting, adding, or removing nodes:

```
root = doc.documentElement

# Remove the first child
```

```
root.removeChild( root.childNodes[0] )

# Move the new first child to the end
root.appendChild( root.childNodes[0] )

# Insert the new first child (originally,
# the third child) before the 20th child.
root.insertBefore( root.childNodes[0], root.childNodes[20] )
```

Again, I will refer you to the Python documentation for a complete listing of the different **`Node`** classes and their various methods.

Relationship to PyXML

The XML Special Interest Group has been working on XML-related Python code for a while. Its code distribution, called PyXML, is available from the SIG's web pages at https://www.python.org/community/sigs/current/xml-sig. The PyXML distribution also used the package name `xml`. If you've written programs that used PyXML, you're probably wondering about its compatibility with the 2.0 **`xml`** package.

The answer is that Python 2.0's **`xml`** package isn't compatible with PyXML, but can be made compatible by installing a recent version PyXML. Many applications can get by with the XML support that is included with Python 2.0, but more complicated applications will require that the full PyXML package will be installed. When installed, PyXML versions 0.6.0 or greater will replace the **`xml`** package shipped with Python, and will be a strict superset of the standard package, adding a bunch of additional features. Some of the additional features in PyXML include:

- 4DOM, a full DOM implementation from FourThought, Inc.

- The xmlproc validating parser, written by Lars Marius Garshol.
- The **`sgmlop`** parser accelerator module, written by Fredrik Lundh.

Module changes

Lots of improvements and bugfixes were made to Python’s extensive standard library; some of the affected modules include **`readline`**, **`ConfigParser`**, **`cgi`**, **`calendar`**, **`posix`**, **`readline`**, **`xmllib`**, **`aifc`**, **`chunk`**, **`wave`**, **`random`**, **`shelve`**, and **`nntplib`**. Consult the CVS logs for the exact patch-by-patch details.

Brian Gallew contributed OpenSSL support for the **`socket`** module. OpenSSL is an implementation of the Secure Socket Layer, which encrypts the data being sent over a socket. When compiling Python, you can edit `Modules/Setup` to include SSL support, which adds an additional function to the **`socket`** module: `socket.ssl(socket, keyfile, certfile)`, which takes a socket object and returns an SSL socket. The **`httplib`** and **`urllib`** modules were also changed to support `https://` URLs, though no one has implemented FTP or SMTP over SSL.

The **`httplib`** module has been rewritten by Greg Stein to support HTTP/1.1.

Backward compatibility with the 1.5 version of **`httplib`** is provided, though using HTTP/1.1 features such as pipelining will require rewriting code to use a different set of interfaces.

The **`Tkinter`** module now supports Tcl/Tk version 8.1, 8.2, or 8.3, and support for the older 7.x versions has been dropped. The Tkinter module now supports displaying Unicode strings in Tk widgets. Also, Fredrik Lundh contributed an optimization which makes operations like `create_line` and `create_polygon` much faster, especially when using lots of coordinates.

The **`curses`** module has been greatly extended, starting from Oliver Andrich's enhanced version, to provide many additional functions from ncurses and SYSV curses, such as colour, alternative character set support, pads, and mouse support. This means the module is no longer compatible with operating systems that only have BSD curses, but there don't seem to be any currently maintained OSes that fall into this category.

As mentioned in the earlier discussion of 2.0's Unicode support, the underlying implementation of the regular expressions provided by the **`re`** module has been changed. SRE, a new regular expression engine written by Fredrik Lundh and partially funded by Hewlett Packard, supports matching against both 8-bit strings and Unicode strings.

New modules

A number of new modules were added. We'll simply list them with brief descriptions; consult the 2.0 documentation for the details of a particular module.

- **`atexit`**: For registering functions to be called before the Python interpreter exits. Code that currently sets `sys.exitfunc` directly should be changed to use the **`atexit`** module instead, importing **`atexit`** and calling **`atexit.register()`** with the function to be called on exit. (Contributed by Skip Montanaro.)
- **`codecs`**, **`encodings`**, **`unicodedata`**: Added as part of the new Unicode support.
- **`filecmp`**: Supersedes the old **`cmp`**, **`cmpcache`** and **`dircmp`** modules, which have now become deprecated. (Contributed by Gordon MacMillan and Moshe Zadka.)

- **`gettext`**: This module provides internationalization (I18N) and localization (L10N) support for Python programs by providing an interface to the GNU gettext message catalog library. (Integrated by Barry Warsaw, from separate contributions by Martin von Löwis, Peter Funk, and James Henstridge.)
- **`linuxaudiodev`**: Support for the `/dev/audio` device on Linux, a twin to the existing **`sunaudiodev`** module. (Contributed by Peter Bosch, with fixes by Jeremy Hylton.)
- **`mmap`**: An interface to memory-mapped files on both Windows and Unix. A file's contents can be mapped directly into memory, at which point it behaves like a mutable string, so its contents can be read and modified. They can even be passed to functions that expect ordinary strings, such as the **`re`** module. (Contributed by Sam Rushing, with some extensions by A.M. Kuchling.)
- **`pyexpat`**: An interface to the Expat XML parser. (Contributed by Paul Prescod.)
- **`robotparser`**: Parse a `robots.txt` file, which is used for writing web spiders that politely avoid certain areas of a web site. The parser accepts the contents of a `robots.txt` file, builds a set of rules from it, and can then answer questions about the fetchability of a given URL. (Contributed by Skip Montanaro.)
- **`tabnanny`**: A module/script to check Python source code for ambiguous indentation. (Contributed by Tim Peters.)
- **`UserString`**: A base class useful for deriving objects that behave like strings.
- **`webbrowser`**: A module that provides a platform independent way to launch a web browser on a specific URL. For each platform, various browsers are tried in a specific order. The user can alter which browser is launched by setting the *BROWSER* environment variable. (Originally inspired by Eric S. Raymond's patch to **`urllib`** which added similar functionality, but the final

module comes from code originally implemented by Fred Drake as `Tools/idle/BrowserControl.py`, and adapted for the standard library by Fred.)
- `_winreg`: An interface to the Windows registry. `_winreg` is an adaptation of functions that have been part of PythonWin since 1995, but has now been added to the core distribution, and enhanced to support Unicode. `_winreg` was written by Bill Tutt and Mark Hammond.
- `zipfile`: A module for reading and writing ZIP-format archives. These are archives produced by **PKZIP** on DOS/Windows or **zip** on Unix, not to be confused with **gzip**-format files (which are supported by the `gzip` module) (Contributed by James C. Ahlstrom.)
- `imputil`: A module that provides a simpler way for writing customized import hooks, in comparison to the existing `ihooks` module. (Implemented by Greg Stein, with much discussion on python-dev along the way.)

IDLE Improvements

IDLE is the official Python cross-platform IDE, written using Tkinter. Python 2.0 includes IDLE 0.6, which adds a number of new features and improvements. A partial list:

- UI improvements and optimizations, especially in the area of syntax highlighting and auto-indentation.
- The class browser now shows more information, such as the top level functions in a module.
- Tab width is now a user settable option. When opening an existing Python file, IDLE automatically detects the indentation conventions, and adapts.
- There is now support for calling browsers on various platforms, used to open the Python documentation in a browser.

- IDLE now has a command line, which is largely similar to the vanilla Python interpreter.
- Call tips were added in many places.
- IDLE can now be installed as a package.
- In the editor window, there is now a line/column bar at the bottom.
- Three new keystroke commands: Check module (`Alt-F5`), Import module (`F5`) and Run script (`Ctrl-F5`).

Deleted and Deprecated Modules

A few modules have been dropped because they're obsolete, or because there are now better ways to do the same thing. The **`stdwin`** module is gone; it was for a platform-independent windowing toolkit that's no longer developed.

A number of modules have been moved to the `lib-old` subdirectory: **`cmp`**, **`cmpcache`**, **`dircmp`**, **`dump`**, **`find`**, **`grep`**, **`packmail`**, **`poly`**, **`util`**, **`whatsound`**, **`zmod`**. If you have code which relies on a module that's been moved to `lib-old`, you can simply add that directory to `sys.path` to get them back, but you're encouraged to update any code that uses these modules.

Acknowledgements

The authors would like to thank the following people for offering suggestions on various drafts of this article: David Bolen, Mark Hammond, Gregg Hauser, Jeremy Hylton, Fredrik Lundh, Detlef Lannert, Aahz Maruch, Skip Montanaro, Vladimir Marangozov, Tobias Polzin, Guido van Rossum, Neil Schemenauer, and Russ Schmidt.

Changelog

Python next

Release date: XXXX-XX-XX

Windows

- gh-128067 [https://github.com/python/cpython/issues/128067]: Fix a bug in PyREPL on Windows where output without a trailing newline was overwritten by the next prompt.

Tools/Demos

- gh-142095 [https://github.com/python/cpython/issues/142095]: Make gdb ‘py-bt’ command use frame from thread local state when available. Patch by Sam Gross and Victor Stinner.

Tests

- gh-143460 [https://github.com/python/cpython/issues/143460]: Skip tests relying on infinite recusion if stack size is unlimited.
- gh-65784 [https://github.com/python/cpython/issues/65784]: Add support for parametrized resource `wantobjects` in regrtests, which allows to run Tkinter tests with the specified value of **`tkinter.wantobjects`**, for example `-u wantobjects=0`.
- gh-143553 [https://github.com/python/cpython/issues/143553]: Add support for parametrized resources, such as `-u xpickle=2.7`.
- gh-142836 [https://github.com/python/cpython/issues/142836]: Accommodated Solaris in

`test_pdb.test_script_target_anonymous_pipe`.
- bpo-31391 [https://bugs.python.org/issue?@action=redirect&bpo=31391]: Forward-port test_xpickle from Python 2 to Python 3 and add the resource back to test's command line.

Security

- gh-143916 [https://github.com/python/cpython/issues/143916]: Reject C0 control characters within wsgiref.headers.Headers fields, values, and parameters.

Library

- gh-143831 [https://github.com/python/cpython/issues/143831]: **`annotationlib.ForwardRef`** objects are now hashable when created from annotation scopes with closures. Previously, hashing such objects would throw an exception. Patch by Bartosz Sławecki.
- gh-143880 [https://github.com/python/cpython/issues/143880]: Fix data race in **`functools.partial()`** in the free threading build.
- gh-143706 [https://github.com/python/cpython/issues/143706]: Fix **`multiprocessing`** forkserver so that **`sys.argv`** is correctly set before `__main__` is preloaded. Previously, **`sys.argv`** was empty during main module import in forkserver child processes. This fixes a regression introduced in 3.13.8 and 3.14.1. Root caused by Aaron Wieczorek, test provided by Thomas Watson, thanks!
- gh-143638 [https://github.com/python/cpython/issues/143638]: Forbid reentrant calls of the **`pickle.Pickler`** and **`pickle.Unpickler`** methods for the C implementation. Previously, this could cause crash or data corruption, now concurrent calls of methods of the same object raise **`RuntimeError`**.

- gh-78724 [https://github.com/python/cpython/issues/78724]: Raise **`RuntimeError`**’s when user attempts to call methods on half-initialized **`Struct`** objects, For example, created by `Struct.__new__(Struct)`. Patch by Sergey B Kirpichev.
- gh-143196 [https://github.com/python/cpython/issues/143196]: Fix crash when the internal encoder object returned by undocumented function **`json.encoder.c_make_encoder()`** was called with non-zero second (*_current_indent_level*) argument.
- gh-143191 [https://github.com/python/cpython/issues/143191]: **`_thread.stack_size()`** now raises **`ValueError`** if the stack size is too small. Patch by Victor Stinner.
- gh-143602 [https://github.com/python/cpython/issues/143602]: Fix a inconsistency issue in **`write()`** that leads to unexpected buffer overwrite by deduplicating the buffer exports.
- gh-143547 [https://github.com/python/cpython/issues/143547]: Fix **`sys.unraisablehook()`** when the hook raises an exception and changes **`sys.unraisablehook()`**: hold a strong reference to the old hook. Patch by Victor Stinner.
- gh-143517 [https://github.com/python/cpython/issues/143517]: **`annotationlib.get_annotations()`** no longer raises a **`SyntaxError`** when evaluating a stringified starred annotation that starts with one or more whitespace characters followed by a `*`. Patch by Bartosz Sławecki.
- gh-143378 [https://github.com/python/cpython/issues/143378]: Fix use-after-free crashes when a **`BytesIO`** object is concurrently mutated during **`write()`** or **`writelines()`**.
- gh-143346 [https://github.com/python/cpython/issues/143346]: Fix incorrect wrapping of the Base64 data in **`plistlib._PlistWriter`** when the indent contains a mix of tabs and spaces.
- gh-143310 [https://github.com/python/cpython/issues/143310]: **`tkinter`**: fix a crash when a Python **`list`** is mutated during the conversion to a Tcl object (e.g., when setting a Tcl variable). Patch by Bénédikt Tran.

- gh-143309 [https://github.com/python/cpython/issues/143309]: Fix a crash in `os.execve()` on non-Windows platforms when given a custom environment mapping which is then mutated during parsing. Patch by Bénédikt Tran.
- gh-143308 [https://github.com/python/cpython/issues/143308]: `pickle`: fix use-after-free crashes when a `PickleBuffer` is concurrently mutated by a custom buffer callback during pickling. Patch by Bénédikt Tran and Aaron Wieczorek.
- gh-143249 [https://github.com/python/cpython/issues/143249]: Fix possible buffer leaks in Windows overlapped I/O on error handling.
- gh-143241 [https://github.com/python/cpython/issues/143241]: `zoneinfo`: fix infinite loop in `ZoneInfo.from_file` when parsing a malformed TZif file. Patch by Fatih Celik.
- gh-142830 [https://github.com/python/cpython/issues/142830]: `sqlite3`: fix use-after-free crashes when the connection's callbacks are mutated during a callback execution. Patch by Bénédikt Tran.
- gh-143200 [https://github.com/python/cpython/issues/143200]: `xml.etree.ElementTree`: fix use-after-free crashes in `__getitem__()` and `__setitem__()` methods of `Element` when the element is concurrently mutated. Patch by Bénédikt Tran.
- gh-142195 [https://github.com/python/cpython/issues/142195]: Updated timeout evaluation logic in `subprocess` to be compatible with deterministic environments like Shadow where time moves exactly as requested.
- gh-142164 [https://github.com/python/cpython/issues/142164]: Fix the ctypes bitfield overflow error message to report the correct offset and size calculation.
- gh-143145 [https://github.com/python/cpython/issues/143145]: Fixed a possible reference leak in ctypes when constructing results with multiple output parameters on error.
- gh-122431 [https://github.com/python/cpython/issues/122431]: Corrected the error message in

`readline.append_history_file()` to state that `nelements` must be non-negative instead of positive.

- gh-143004 [https://github.com/python/cpython/issues/143004]: Fix a potential use-after-free in `collections.Counter.update()` when user code mutates the Counter during an update.
- gh-143046 [https://github.com/python/cpython/issues/143046]: The `asyncio` REPL no longer prints copyright and version messages in the quiet mode (`-q`). Patch by Bartosz Sławecki.
- gh-140648 [https://github.com/python/cpython/issues/140648]: The `asyncio` REPL now respects the `-I` flag (isolated mode). Previously, it would load and execute `PYTHONSTARTUP` even if the flag was set. Contributed by Bartosz Sławecki.
- gh-142991 [https://github.com/python/cpython/issues/142991]: Fixed socket operations such as recvfrom() and sendto() for FreeBSD divert(4) socket.
- gh-143010 [https://github.com/python/cpython/issues/143010]: Fixed a bug in `mailbox` where the precise timing of an external event could result in the library opening an existing file instead of a file it expected to create.
- gh-142881 [https://github.com/python/cpython/issues/142881]: Fix concurrent and reentrant call of `atexit.unregister()`.
- gh-112127 [https://github.com/python/cpython/issues/112127]: Fix possible use-after-free in `atexit.unregister()` when the callback is unregistered during comparison.
- gh-142783 [https://github.com/python/cpython/issues/142783]: Fix zoneinfo use-after-free with descriptor _weak_cache. a descriptor as _weak_cache could cause crashes during object creation. The fix ensures proper reference counting for descriptor-provided objects.
- gh-142754 [https://github.com/python/cpython/issues/142754]: Add the *ownerDocument* attribute to `xml.dom.minidom` elements and attributes created by directly instantiating the `Element` or `Attr` class. Note that this way of creating nodes is not supported; creator functions like

`xml.dom.Document.documentElement()` should be used instead.

- gh-142784 [https://github.com/python/cpython/issues/142784]: The **`asyncio`** REPL now properly closes the loop upon the end of interactive session. Previously, it could cause surprising warnings. Contributed by Bartosz Sławecki.
- gh-142594 [https://github.com/python/cpython/issues/142594]: Fix crash in `TextIOWrapper.close()` when the underlying buffer's `closed` property calls **`detach()`**.
- gh-142451 [https://github.com/python/cpython/issues/142451]: **`hmac`**: Ensure that the **`HMAC.block_size`** attribute is correctly copied by **`HMAC.copy`**. Patch by Bénédikt Tran.
- gh-142495 [https://github.com/python/cpython/issues/142495]: **`collections.defaultdict`** now prioritizes **`__setitem__()`** when inserting default values from `default_factory`. This prevents race conditions where a default value would overwrite a value set before `default_factory` returns.
- gh-142651 [https://github.com/python/cpython/issues/142651]: **`unittest.mock`**: fix a thread safety issue where **`Mock.call_count`** may return inaccurate values when the mock is called concurrently from multiple threads.
- gh-142595 [https://github.com/python/cpython/issues/142595]: Added type check during initialization of the **`decimal`** module to prevent a crash in case of broken stdlib. Patch by Sergey B Kirpichev.
- gh-142556 [https://github.com/python/cpython/issues/142556]: Fix crash when a task gets re-registered during finalization in **`asyncio`**. Patch by Kumar Aditya.
- gh-123241 [https://github.com/python/cpython/issues/123241]: Avoid reference count operations in garbage collection of **`ctypes`** objects.
- gh-142517 [https://github.com/python/cpython/issues/142517]: The non-`compat32` **`email`** policies now correctly handle refolding encoded words that contain bytes that can not be decoded in their specified character set. Previously this resulted in an encoding exception during folding.

- gh-112527 [https://github.com/python/cpython/issues/112527]: The help text for required options in `argparse` no longer extended with " (default: None)".
- gh-142346 [https://github.com/python/cpython/issues/142346]: Fix usage formatting for mutually exclusive groups in `argparse` when they are preceded by positional arguments or followed or intermixed with other optional arguments.
- gh-142315 [https://github.com/python/cpython/issues/142315]: Pdb can now run scripts from anonymous pipes used in process substitution. Patch by Bartosz Sławecki.
- gh-142332 [https://github.com/python/cpython/issues/142332]: Fix usage formatting for positional arguments in mutually exclusive groups in `argparse`. in `argparse`.
- gh-142282 [https://github.com/python/cpython/issues/142282]: Fix `winreg.QueryValueEx()` to not accidentally read garbage buffer under race condition.
- gh-75949 [https://github.com/python/cpython/issues/75949]: Fix `argparse` to preserve `|` separators in mutually exclusive groups when the usage line wraps due to length.
- gh-142267 [https://github.com/python/cpython/issues/142267]: Improve `argparse` performance by caching the formatter used for argument validation.
- gh-68552 [https://github.com/python/cpython/issues/68552]: `MisplacedEnvelopeHeaderDefect` and `Missing header name` defects are now correctly passed to the `handle_defect` method of `policy` in `FeedParser`.
- gh-142006 [https://github.com/python/cpython/issues/142006]: Fix a bug in the `email.policy.default` folding algorithm which incorrectly resulted in a doubled newline when a line ending at exactly max_line_length was followed by an unfoldable token.
- gh-105836 [https://github.com/python/cpython/issues/105836]: Fix `asyncio.run_coroutine_threadsafe()` leaving underlying cancelled asyncio task running.

- gh-139971 [https://github.com/python/cpython/issues/139971]: `pydoc`: Ensure that the link to the online documentation of a stdlib module is correct.
- gh-139262 [https://github.com/python/cpython/issues/139262]: Some keystrokes can be swallowed in the new `PyREPL` on Windows, especially when used together with the ALT key. Fix by Chris Eibl.
- gh-138897 [https://github.com/python/cpython/issues/138897]: Improved `license`/`copyright`/`credits` display in the REPL: now uses a pager.
- gh-79986 [https://github.com/python/cpython/issues/79986]: Add parsing for `References` and `In-Reply-To` headers to the `email` library that parses the header content as lists of message id tokens. This prevents them from being folded incorrectly.
- gh-136282 [https://github.com/python/cpython/issues/136282]: Add support for `UNNAMED_SECTION` when creating a section via the mapping protocol access
- gh-109263 [https://github.com/python/cpython/issues/109263]: Starting a process from spawn context in `multiprocessing` no longer sets the start method globally.
- gh-133253 [https://github.com/python/cpython/issues/133253]: Fix thread-safety issues in `linecache`.
- gh-132715 [https://github.com/python/cpython/issues/132715]: Skip writing objects during marshalling once a failure has occurred.

Documentation

- gh-140806 [https://github.com/python/cpython/issues/140806]: Add documentation for `enum.bin()`.

Core and Builtins

- gh-141805 [https://github.com/python/cpython/issues/141805]: Fix crash in `set` when objects with the same hash are concurrently added to the set after removing an element with the same hash while the set still contains elements with the same hash.

- gh-143670 [https://github.com/python/cpython/issues/143670]: Fixes a crash in `ga_repr_items_list` function.
- gh-143377 [https://github.com/python/cpython/issues/143377]: Fix a crash in **`_interpreters.capture_exception()`** when the exception is incorrectly formatted. Patch by Bénédikt Tran.
- gh-136924 [https://github.com/python/cpython/issues/136924]: The interactive help mode in the REPL no longer incorrectly syntax highlights text input as Python code. Contributed by Olga Matoula.
- gh-143189 [https://github.com/python/cpython/issues/143189]: Fix crash when inserting a non-**`str`** key into a split table dictionary when the key matches an existing key in the split table but has no corresponding value in the dict.
- gh-143228 [https://github.com/python/cpython/issues/143228]: Fix use-after-free in perf trampoline when toggling profiling while threads are running or during interpreter finalization with daemon threads active. The fix uses reference counting to ensure trampolines are not freed while any code object could still reference them. Pach by Pablo Galindo
- gh-142664 [https://github.com/python/cpython/issues/142664]: Fix a use-after-free crash in **`memoryview.__hash__`** when the `__hash__` method of the referenced object mutates that object or the view. Patch by Bénédikt Tran.
- gh-142557 [https://github.com/python/cpython/issues/142557]: Fix a use-after-free crash in bytearray.__mod__ when the **`bytearray`** is mutated while formatting the `%`-style arguments. Patch by Bénédikt Tran.
- gh-143195 [https://github.com/python/cpython/issues/143195]: Fix use-after-free crashes in **`bytearray.hex()`** and **`memoryview.hex()`** when the separator’s **`__len__()`** mutates the original object. Patch by Bénédikt Tran.

- gh-142975 [https://github.com/python/cpython/issues/142975]: Fix crash after unfreezing all objects tracked by the garbage collector on the free threaded build.
- gh-143135 [https://github.com/python/cpython/issues/143135]: Set **`sys.flags.inspect`** to `1` when **`PYTHONINSPECT`** is `0`. Previously, it was set to `0` in this case.
- gh-143003 [https://github.com/python/cpython/issues/143003]: Fix an overflow of the shared empty buffer in **`bytearray.extend()`** when `__length_hint__()` returns 0 for non-empty iterator.
- gh-143006 [https://github.com/python/cpython/issues/143006]: Fix a possible assertion error when comparing negative non-integer `float` and `int` with the same number of bits in the integer part.
- gh-143057 [https://github.com/python/cpython/issues/143057]: Avoid locking in **`PyTraceMalloc_Track()`** and **`PyTraceMalloc_Untrack()`** when **`tracemalloc`** is not enabled.
- gh-142776 [https://github.com/python/cpython/issues/142776]: Fix a file descriptor leak in import.c
- gh-142829 [https://github.com/python/cpython/issues/142829]: Fix a use-after-free crash in **`contextvars.Context`** comparison when a custom `__eq__` method modifies the context via **`set()`**.
- gh-142766 [https://github.com/python/cpython/issues/142766]: Clear the frame of a generator when **`generator.close()`** is called.
- gh-142737 [https://github.com/python/cpython/issues/142737]: Tracebacks will be displayed in fallback mode even if **`io.open()`** is lost. Previously, this would crash the interpreter. Patch by Bartosz Sławecki.
- gh-142554 [https://github.com/python/cpython/issues/142554]: Fix a crash in **`divmod()`** when **`_pylong.int_divmod()`** does not return a tuple of length two exactly. Patch by Bénédikt Tran.
- gh-142560 [https://github.com/python/cpython/issues/142560]: Fix use-after-free in **`bytearray`** search-like methods (**`find()`**,

`count()`, `index()`, `rindex()`, and `rfind()`) by marking the storage as exported which causes reallocation attempts to raise `BufferError`. For `contains()`, `split()`, and `rsplit()` the buffer protocol is used for this.

- gh-142531 [https://github.com/python/cpython/issues/142531]: Fix a free-threaded GC performance regression. If there are many untracked tuples, the GC will run too often, resulting in poor performance. The fix is to include untracked tuples in the “long lived” object count. The number of frozen objects is also now included since the free-threaded GC must scan those too.
- gh-142402 [https://github.com/python/cpython/issues/142402]: Fix reference counting when adjacent literal parts are merged while constructing `string.templatelib.Template`, preventing the displaced string object from leaking.
- gh-133932 [https://github.com/python/cpython/issues/133932]: Fix crash in the free threading build when clearing frames that hold tagged integers.
- gh-142343 [https://github.com/python/cpython/issues/142343]: Fix SIGILL crash on m68k due to incorrect assembly constraint.
- gh-100964 [https://github.com/python/cpython/issues/100964]: Fix reference cycle in exhausted generator frames. Patch by Savannah Ostrowski.
- gh-69605 [https://github.com/python/cpython/issues/69605]: Fix edge-cases around already imported modules in the REPL auto-completion of imports.
- gh-138568 [https://github.com/python/cpython/issues/138568]: Adjusted the built-in `help()` function so that empty inputs are ignored in interactive mode.
- gh-137007 [https://github.com/python/cpython/issues/137007]: Fix a bug during JIT compilation failure which caused garbage collection debug assertions to fail.

C API

- gh-142589 [https://github.com/python/cpython/issues/142589]: Fix **`PyUnstable_Object_IsUniqueReferencedTemporary()`** handling of tagged ints on the interpreter stack.
- gh-142571 [https://github.com/python/cpython/issues/142571]: **`PyUnstable_CopyPerfMapFile()`** now checks that opening the file succeeded before flushing.

Build

- gh-142454 [https://github.com/python/cpython/issues/142454]: When calculating the digest of the JIT stencils input, sort the hashed files by filenames before adding their content to the hasher. This ensures deterministic hash input and hence deterministic hash, independent on filesystem order.
- gh-141808 [https://github.com/python/cpython/issues/141808]: When running `make clean-retain-profile`, keep the generated JIT stencils. That way, the stencils are not generated twice when Profile-guided optimization (PGO) is used. It also allows distributors to supply their own pre-built JIT stencils.
- gh-138061 [https://github.com/python/cpython/issues/138061]: Ensure reproducible builds by making JIT stencil header generation deterministic.

Python 3.14.2 final

Release date: 2025-12-05

Security

- gh-142145 [https://github.com/python/cpython/issues/142145]: Remove quadratic behavior in `xml.minidom` node ID cache clearing.

- gh-119452 [https://github.com/python/cpython/issues/119452]: Fix a potential memory denial of service in the `http.server` module. When a malicious user is connected to the CGI server on Windows, it could cause an arbitrary amount of memory to be allocated. This could have led to symptoms including a `MemoryError`, swapping, out of memory (OOM) killed processes or containers, or even system crashes.

Library

- gh-140797 [https://github.com/python/cpython/issues/140797]: Revert changes to the undocumented **`re.Scanner`** class. Capturing groups are still allowed for backward compatibility, although using them can lead to incorrect result. They will be forbidden in future Python versions.

- gh-142206 [https://github.com/python/cpython/issues/142206]: The resource tracker in the `multiprocessing` module now uses the original communication protocol, as in Python 3.14.0 and below, by default. This avoids issues with upgrading Python while it is running. (Note that such ‘in-place’ upgrades are not tested.) The tracker remains compatible with subprocesses that use new protocol (that is, subprocesses using Python 3.13.10, 3.14.1 and 3.15).

- gh-142214 [https://github.com/python/cpython/issues/142214]: Fix two regressions in `dataclasses` in Python 3.14.1 related to annotations.

    - An exception is no longer raised if `slots=True` is used and the `__init__` method does not have an `__annotate__` attribute (likely because `init=False` was used).

  - An exception is no longer raised if annotations are requested on the `__init__` method and one of the fields is not present in the class annotations. This can occur in certain dynamic scenarios.

  Patch by Jelle Zijlstra.

Core and Builtins

- gh-142218 [https://github.com/python/cpython/issues/142218]: Fix crash when inserting into a split table dictionary with a non **`str`** key that matches an existing key.

Library

- gh-116738 [https://github.com/python/cpython/issues/116738]: Fix **`cmath`** data race when initializing trigonometric tables with subinterpreters.

Python 3.14.1 final

Release date: 2025-12-02

Windows

- gh-139810 [https://github.com/python/cpython/issues/139810]: Installing with `py install 3[.x]-dev` will now select final versions as well as prereleases.

Tools/Demos

- gh-141692 [https://github.com/python/cpython/issues/141692]: Each slice of an iOS XCframework now contains a `lib` folder that contains a symlink to the libpython dylib. This allows binary modules to be compiled for iOS using dynamic libreary linking, rather than Framework linking.

- gh-141442 [https://github.com/python/cpython/issues/141442]: The iOS testbed now correctly handles test arguments that contain spaces.
- gh-140702 [https://github.com/python/cpython/issues/140702]: The iOS testbed app will now expose the `GITHUB_ACTIONS` environment variable to iOS apps being tested.
- gh-137484 [https://github.com/python/cpython/issues/137484]: Have `Tools/wasm/wasi` put the build Python into a directory named after the build triple instead of “build”.
- gh-137248 [https://github.com/python/cpython/issues/137248]: Add a `--logdir` option to `Tools/wasm/wasi` for specifying where to write log files.
- gh-137243 [https://github.com/python/cpython/issues/137243]: Have Tools/wasm/wasi detect a WASI SDK install in /opt when it was directly extracted from a release tarball.

Tests

- gh-140482 [https://github.com/python/cpython/issues/140482]: Preserve and restore the state of `stty echo` as part of the test environment.
- gh-140082 [https://github.com/python/cpython/issues/140082]: Update `python -m test` to set `FORCE_COLOR=1` when being run with color enabled so that **`unittest`** which is run by it with redirected output will output in color.
- gh-139208 [https://github.com/python/cpython/issues/139208]: Fix regrtest `--fast-ci --verbose`: don’t ignore the `--verbose` option anymore. Patch by Victor Stinner.
- gh-136442 [https://github.com/python/cpython/issues/136442]: Use exitcode `1` instead of `5` if **`unittest.TestCase.setUpClass()`** raises an exception

Security

- gh-139700 [https://github.com/python/cpython/issues/139700]: Check consistency of the zip64 end of central directory record. Support records with “zip64 extensible data” if there are no bytes prepended to the ZIP file.
- gh-139283 [https://github.com/python/cpython/issues/139283]: `sqlite3`: correctly handle maximum number of rows to fetch in `Cursor.fetchmany` and reject negative values for `Cursor.arraysize`. Patch by Bénédikt Tran.
- gh-137836 [https://github.com/python/cpython/issues/137836]: Add support of the “plaintext” element, RAWTEXT elements “xmp”, “iframe”, “noembed” and “noframes”, and optionally RAWTEXT element “noscript” in `html.parser.HTMLParser`.
- gh-136063 [https://github.com/python/cpython/issues/136063]: `email.message`: ensure linear complexity for legacy HTTP parameters parsing. Patch by Bénédikt Tran.
- gh-136065 [https://github.com/python/cpython/issues/136065]: Fix quadratic complexity in `os.path.expandvars()`.
- gh-119451 [https://github.com/python/cpython/issues/119451]: Fix a potential memory denial of service in the `http.client` module. When connecting to a malicious server, it could cause an arbitrary amount of memory to be allocated. This could have led to symptoms including a `MemoryError`, swapping, out of memory (OOM) killed processes or containers, or even system crashes.
- gh-119342 [https://github.com/python/cpython/issues/119342]: Fix a potential memory denial of service in the `plistlib` module. When reading a Plist file received from untrusted source, it could cause an arbitrary amount of memory to be allocated. This could have led to symptoms including a `MemoryError`, swapping, out of memory (OOM) killed processes or containers, or even system crashes.

Library

- gh-74389 [https://github.com/python/cpython/issues/74389]: When the stdin being used by a **`subprocess.Popen`** instance is closed, this is now ignored in **`subprocess.Popen.communicate()`** instead of leaving the class in an inconsistent state.

- gh-87512 [https://github.com/python/cpython/issues/87512]: Fix **`subprocess.Popen.communicate()`** timeout handling on Windows when writing large input. Previously, the timeout was ignored during stdin writing, causing the method to block indefinitely if the child process did not consume input quickly. The stdin write is now performed in a background thread, allowing the timeout to be properly enforced.

- gh-141473 [https://github.com/python/cpython/issues/141473]: When **`subprocess.Popen.communicate()`** was called with *input* and a *timeout* and is called for a second time after a **`TimeoutExpired`** exception before the process has died, it should no longer hang.

- gh-59000 [https://github.com/python/cpython/issues/59000]: Fix **`pdb`** breakpoint resolution for class methods when the module defining the class is not imported.

- gh-141570 [https://github.com/python/cpython/issues/141570]: Support file-like object raising **`OSError`** from **`fileno()`** in color detection (`_colorize.can_colorize()`). This can occur when `sys.stdout` is redirected.

- gh-141659 [https://github.com/python/cpython/issues/141659]: Fix bad file descriptor errors from `_posixsubprocess` on AIX.

- gh-141600 [https://github.com/python/cpython/issues/141600]: Fix musl version detection on Void Linux.

- gh-141497 [https://github.com/python/cpython/issues/141497]: **`ipaddress`**: ensure that the methods **`IPv4Network.hosts()`** and **`IPv6Network.hosts()`** always return an iterator.

- gh-140938 [https://github.com/python/cpython/issues/140938]: The **`statistics.stdev()`** and **`statistics.pstdev()`** functions now raise a **`ValueError`** when the input contains an infinity or a NaN.

- gh-124111 [https://github.com/python/cpython/issues/124111]: Updated Tcl threading configuration in **`_tkinter`** to assume that threads are always available in Tcl 9 and later.

- gh-137109 [https://github.com/python/cpython/issues/137109]: The **`os.fork`** and related forking APIs will no longer warn in the common case where Linux or macOS platform APIs return the number of threads in a process and find the answer to be 1 even when a **`os.register_at_fork()`** `after_in_parent=` callback (re)starts a thread.

- gh-141314 [https://github.com/python/cpython/issues/141314]: Fix assertion failure in **`io.TextIOWrapper.tell()`** when reading files with standalone carriage return (`\r`) line endings.

- gh-141311 [https://github.com/python/cpython/issues/141311]: Fix assertion failure in **`io.BytesIO.readinto()`** and undefined behavior arising when read position is above capcity in **`io.BytesIO`**.

- gh-141141 [https://github.com/python/cpython/issues/141141]: Fix a thread safety issue with **`base64.b85decode()`**. Contributed by Benel Tayar.

- gh-137969 [https://github.com/python/cpython/issues/137969]: Fix **`annotationlib.ForwardRef.evaluate()`** returning **`ForwardRef`** objects which don’t update with new globals.

- gh-140911 [https://github.com/python/cpython/issues/140911]: **`collections`**: Ensure that the methods `UserString.rindex()` and `UserString.index()` accept **`collections.UserString`** instances as the sub argument.

- gh-140797 [https://github.com/python/cpython/issues/140797]: The undocumented **`re.Scanner`** class now forbids regular expressions containing capturing groups in its lexicon patterns. Patterns using capturing groups could previously lead to crashes with segmentation fault. Use non-capturing groups (?:…) instead.

- gh-125115 [https://github.com/python/cpython/issues/125115]: Refactor the **`pdb`** parsing issue so positional arguments can pass through intuitively.

- gh-140815 [https://github.com/python/cpython/issues/140815]: **`faulthandler`** now detects if a frame or a code object is invalid or freed. Patch by Victor Stinner.

- gh-100218 [https://github.com/python/cpython/issues/100218]: Correctly set **`errno`** when **`socket.if_nametoindex()`** or **`socket.if_indextoname()`** raise an **`OSError`**. Patch by Bénédikt Tran.

- gh-140875 [https://github.com/python/cpython/issues/140875]: Fix handling of unclosed character references (named and numerical) followed by the end of file in **`html.parser.HTMLParser`** with `convert_charrefs=False`.

- gh-140734 [https://github.com/python/cpython/issues/140734]: **`multiprocessing`**: fix off-by-one error when checking the length of a temporary socket file path. Patch by Bénédikt Tran.

- gh-140874 [https://github.com/python/cpython/issues/140874]: Bump the version of pip bundled in ensurepip to version

25.3

- gh-140691 [https://github.com/python/cpython/issues/140691]: In **`urllib.request`**, when opening a FTP URL fails because a data connection cannot be made, the control connection’s socket is now closed to avoid a **`ResourceWarning`**.

- gh-103847 [https://github.com/python/cpython/issues/103847]: Fix hang when cancelling process created by **`asyncio.create_subprocess_exec()`** or **`asyncio.create_subprocess_shell()`**. Patch by Kumar Aditya.

- gh-120057 [https://github.com/python/cpython/issues/120057]: Add **`os.reload_environ()`** to `os.__all__`.

- gh-140228 [https://github.com/python/cpython/issues/140228]: Avoid making unnecessary filesystem calls for frozen modules in **`linecache`** when the global module cache is not present.

- gh-140590 [https://github.com/python/cpython/issues/140590]: Fix arguments checking for the **`functools.partial.__setstate__()`** that may lead to internal state corruption and crash. Patch by Sergey Miryanov.

- gh-125434 [https://github.com/python/cpython/issues/125434]: Display thread name in **`faulthandler`** on Windows. Patch by Victor Stinner.

- gh-140634 [https://github.com/python/cpython/issues/140634]: Fix a reference counting bug in **`os.sched_param.__reduce__()`**.

- gh-140633 [https://github.com/python/cpython/issues/140633]: Ignore **`AttributeError`** when setting a module’s `__file__` attribute when loading an extension module packaged as Apple Framework.

- gh-140593 [https://github.com/python/cpython/issues/140593]: **`xml.parsers.expat`**: Fix a memory leak that could affect users with **`ElementDeclHandler()`** set to a custom element declaration handler. Patch by Sebastian Pipping.

- gh-140607 [https://github.com/python/cpython/issues/140607]: Inside **`io.RawIOBase.read()`**, validate that the count of bytes returned by **`io.RawIOBase.readinto()`** is valid (inside the provided buffer).

- gh-138162 [https://github.com/python/cpython/issues/138162]: Fix **`logging.LoggerAdapter`** with `merge_extra=True` and without the *extra* argument.

- gh-138774 [https://github.com/python/cpython/issues/138774]: **`ast.unparse()`** now generates full source code when handling **`ast.Interpolation`** nodes that do not have a specified source.

- gh-140474 [https://github.com/python/cpython/issues/140474]: Fix memory leak in **`array.array`** when creating arrays from an empty **`str`** and the `u` type code.

- gh-137530 [https://github.com/python/cpython/issues/137530]: **`dataclasses`** Fix annotations for generated `__init__` methods by replacing the annotations that were in-line in the generated source code with `__annotate__` functions attached to the methods.

- gh-140348 [https://github.com/python/cpython/issues/140348]: Fix regression in Python 3.14.0 where using the `|` operator on a **`typing.Union`** object combined with an object that is not a type would raise an error.

- gh-140272 [https://github.com/python/cpython/issues/140272]: Fix memory leak in the **`clear()`** method of the **`dbm.gnu`** database.

- gh-140041 [https://github.com/python/cpython/issues/140041]: Fix import of **`ctypes`** on Android and Cygwin when ABI flags are present.

- gh-140120 [https://github.com/python/cpython/issues/140120]: Fixed a memory leak in **`hmac`** when it was using the hacl-star backend. Discovered by `@ashm-dev` using AddressSanitizer.

- gh-139905 [https://github.com/python/cpython/issues/139905]: Add suggestion to error message for **`typing.Generic`** subclasses when `cls.__parameters__` is missing due to a parent class failing to call **`super().__init_subclass__()`** in its `__init_subclass__`.

- gh-139894 [https://github.com/python/cpython/issues/139894]: Fix incorrect sharing of current task with the child process while forking in **`asyncio`**. Patch by Kumar Aditya.

- gh-139845 [https://github.com/python/cpython/issues/139845]: Fix to not print KeyboardInterrupt twice in default asyncio REPL.

- gh-139783 [https://github.com/python/cpython/issues/139783]: Fix **`inspect.getsourcelines()`** for the case when a decorator is followed by a comment or an empty line.

- gh-139809 [https://github.com/python/cpython/issues/139809]: Prevent premature colorization of subparser `prog` in **`argparse.ArgumentParser.add_subparsers()`** to respect color environment variable changes after parser creation.

- gh-139736 [https://github.com/python/cpython/issues/139736]: Fix excessive indentation in the default **`argparse`** **`HelpFormatter`**. Patch by Alexander Edland.

- gh-70765 [https://github.com/python/cpython/issues/70765]: **`http.server`**: fix default handling of HTTP/0.9 requests in **`BaseHTTPRequestHandler`**. Previously, **`BaseHTTPRequestHandler.parse_request()`** incorrectly waited for headers in the request although those are not supported in HTTP/0.9. Patch by Bénédikt Tran.

- gh-63161 [https://github.com/python/cpython/issues/63161]: Fix **`tokenize.detect_encoding()`**. Support non-UTF-8 shebang and comments if non-UTF-8 encoding is specified. Detect decoding error for non-UTF-8 encoding. Detect null bytes in source code.

- gh-139391 [https://github.com/python/cpython/issues/139391]: Fix an issue when, on non-Windows platforms, it was not possible to gracefully exit a `python -m asyncio` process suspended by Ctrl+Z and later resumed by *fg* *[https://manpages.debian.org/fg]* other than with *kill* *[https://manpages.debian.org/kill]*.

- gh-101828 [https://github.com/python/cpython/issues/101828]: Fix `'shift_jisx0213'`, `'shift_jis_2004'`, `'euc_jisx0213'` and `'euc_jis_2004'` codecs truncating null chars as they were treated as part of multi-character sequences.

- gh-139289 [https://github.com/python/cpython/issues/139289]: Do a real lazy-import on **`rlcompleter`** in **`pdb`** and restore the existing completer after importing **`rlcompleter`**.

- gh-139246 [https://github.com/python/cpython/issues/139246]: fix: paste zero-width in default repl width is wrong.

- gh-90949 [https://github.com/python/cpython/issues/90949]: Add **`SetAllocTrackerActivationThreshold()`** and **`SetAllocTrackerMaximumAmplification()`** to xmlparser objects to prevent use of disproportional amounts of dynamic memory from within an Expat parser. Patch by Bénédikt Tran.

- gh-139210 [https://github.com/python/cpython/issues/139210]: Fix use-after-free when reporting unknown event in `xml.etree.ElementTree.iterparse()`. Patch by Ken Jin.

- gh-138860 [https://github.com/python/cpython/issues/138860]: Lazy import `rlcompleter` in `pdb` to avoid deadlock in subprocess.

- gh-112729 [https://github.com/python/cpython/issues/112729]: Fix crash when calling `concurrent.interpreters.create()` when the process is out of memory.

- gh-135729 [https://github.com/python/cpython/issues/135729]: Fix unraisable exception during finalization when using `concurrent.interpreters` in the REPL.

- gh-139076 [https://github.com/python/cpython/issues/139076]: Fix a bug in the `pydoc` module that was hiding functions in a Python module if they were implemented in an extension module and the module did not have `__all__`.

- gh-139065 [https://github.com/python/cpython/issues/139065]: Fix trailing space before a wrapped long word if the line length is exactly *width* in `textwrap`.

- gh-139001 [https://github.com/python/cpython/issues/139001]: Fix race condition in `pathlib.Path` on the internal `_raw_paths` field.

- gh-138813 [https://github.com/python/cpython/issues/138813]: **`multiprocessing.BaseProcess`** defaults `kwargs` to `None` instead of a shared dictionary.

- gh-138993 [https://github.com/python/cpython/issues/138993]: Dedent `credits` text.

- gh-138891 [https://github.com/python/cpython/issues/138891]: Fix `SyntaxError` when `inspect.get_annotations(f, eval_str=True)` is called on a function annotated with a **PEP 646** [https://peps.python.org/pep-0646/] `star_expression`

- gh-130567 [https://github.com/python/cpython/issues/130567]: Fix possible crash in `locale.strxfrm()` due to a platform bug on macOS.

- gh-138859 [https://github.com/python/cpython/issues/138859]: Fix generic type parameterization raising a `TypeError` when omitting a **`ParamSpec`** that has a default which is not a list of types.

- gh-138764 [https://github.com/python/cpython/issues/138764]: Prevent `annotationlib.call_annotate_function()` from calling `__annotate__` functions that don’t support `VALUE_WITH_FAKE_GLOBALS` in a fake globals namespace with empty globals.

  Make `FORWARDREF` and `STRING` annotations fall back to using `VALUE` annotations in the case that neither their own format, nor `VALUE_WITH_FAKE_GLOBALS` are supported.

- gh-138775 [https://github.com/python/cpython/issues/138775]: Use of `python -m` with `base64` has been fixed to detect input from a terminal so that it properly notices EOF.

- gh-138779 [https://github.com/python/cpython/issues/138779]: Support device numbers larger than `2**63-1` for the `st_rdev` field of the `os.stat_result` structure.

- gh-137706 [https://github.com/python/cpython/issues/137706]: Fix the partial evaluation of annotations that use `typing.Annotated[T, x]` where `T` is a forward reference.

- gh-88375 [https://github.com/python/cpython/issues/88375]: Fix normalization of the `robots.txt` rules and URLs in the **`urllib.robotparser`** module. No longer ignore trailing `?`. Distinguish raw special characters `?`, `=` and `&` from the percent-encoded ones.

- gh-111788 [https://github.com/python/cpython/issues/111788]: Fix parsing errors in the **`urllib.robotparser`** module. Don't fail trying to parse weird paths. Don't fail trying to decode non-UTF-8 `robots.txt` files.

- gh-98896 [https://github.com/python/cpython/issues/98896]: Fix a failure in multiprocessing resource_tracker when SharedMemory names contain colons. Patch by Rani Pinchuk.

- gh-138425 [https://github.com/python/cpython/issues/138425]: Fix partial evaluation of **`annotationlib.ForwardRef`** objects which rely on names defined as globals.

- gh-138432 [https://github.com/python/cpython/issues/138432]: **`zoneinfo.reset_tzpath()`** will now convert any **`os.PathLike`** objects it receives into strings before adding them to `TZPATH`. It will raise `TypeError` if anything other than a string is found after this conversion. If given an **`os.PathLike`** object that represents a relative path, it will now raise `ValueError` instead of `TypeError`, and present a more informative error message.

- gh-138008 [https://github.com/python/cpython/issues/138008]: Fix segmentation faults in the **`ctypes`** module due to invalid **`argtypes`**. Patch by Dung Nguyen.

- gh-60462 [https://github.com/python/cpython/issues/60462]: Fix **`locale.strxfrm()`** on Solaris (and possibly other platforms).

- gh-138239 [https://github.com/python/cpython/issues/138239]: The REPL now highlights `type` as a soft keyword in type statements.
- gh-138204 [https://github.com/python/cpython/issues/138204]: Forbid expansion of shared anonymous `memory maps` on Linux, which caused a bus error.
- gh-138010 [https://github.com/python/cpython/issues/138010]: Fix an issue where defining a class with an `@warnings.deprecated`-decorated base class may not invoke the correct `__init_subclass__()` method in cases involving multiple inheritance. Patch by Brian Schubert.
- gh-138151 [https://github.com/python/cpython/issues/138151]: In `annotationlib`, improve evaluation of forward references to nonlocal variables that are not yet defined when the annotations are initially evaluated.
- gh-137317 [https://github.com/python/cpython/issues/137317]: `inspect.signature()` now correctly handles classes that use a descriptor on a wrapped `__init__()` or `__new__()` method. Contributed by Yongyu Yan.
- gh-137754 [https://github.com/python/cpython/issues/137754]: Fix import of the `zoneinfo` module if the C implementation of the `datetime` module is not available.
- gh-137490 [https://github.com/python/cpython/issues/137490]: Handle `ECANCELED` in the same way as `EINTR` in `signal.sigwaitinfo()` on NetBSD.
- gh-137477 [https://github.com/python/cpython/issues/137477]: Fix `inspect.getblock()`, `inspect.getsourcelines()` and `inspect.getsource()` for generator expressions.

- gh-137044 [https://github.com/python/cpython/issues/137044]: Return large limit values as positive integers instead of negative integers in `resource.getrlimit()`. Accept large values and reject negative values (except `RLIM_INFINITY`) for limits in `resource.setrlimit()`.

- gh-75989 [https://github.com/python/cpython/issues/75989]: `tarfile.TarFile.extractall()` and `tarfile.TarFile.extract()` now overwrite symlinks when extracting hardlinks. (Contributed by Alexander Enrique Urieles Nieto in gh-75989 [https://github.com/python/cpython/issues/75989].)

- gh-137017 [https://github.com/python/cpython/issues/137017]: Fix `threading.Thread.is_alive` to remain `True` until the underlying OS thread is fully cleaned up. This avoids false negatives in edge cases involving thread monitoring or premature `threading.Thread.is_alive` calls.

- gh-137273 [https://github.com/python/cpython/issues/137273]: Fix debug assertion failure in `locale.setlocale()` on Windows.

- gh-137239 [https://github.com/python/cpython/issues/137239]: `heapq`: Update **`heapq.__all__`** with `*_max` functions.

- gh-81325 [https://github.com/python/cpython/issues/81325]: `tarfile.TarFile` now accepts a path-like when working on a tar archive. (Contributed by Alexander Enrique Urieles Nieto in gh-81325 [https://github.com/python/cpython/issues/81325].)

- gh-137185 [https://github.com/python/cpython/issues/137185]: Fix a potential async-signal-safety issue in `faulthandler` when printing C stack traces.

- gh-136914 [https://github.com/python/cpython/issues/136914]: Fix retrieval of `doctest.DocTest.lineno` for objects decorated with `functools.cache()` or `functools.cached_property`.

- gh-136912 [https://github.com/python/cpython/issues/136912]: **`hmac.digest()`** now properly handles large keys and messages by falling back to the pure Python implementation when necessary. Patch by Bénédikt Tran.

- gh-83424 [https://github.com/python/cpython/issues/83424]: Allows creating a **`ctypes.CDLL`** without name when passing a handle as an argument.

- gh-136234 [https://github.com/python/cpython/issues/136234]: Fix **`asyncio.WriteTransport.writelines()`** to be robust to connection failure, by using the same behavior as **`write()`**.

- gh-136507 [https://github.com/python/cpython/issues/136507]: Fix mimetypes CLI to handle multiple file parameters.

- gh-136057 [https://github.com/python/cpython/issues/136057]: Fixed the bug in **`pdb`** and **`bdb`** where `next` and `step` can’t go over the line if a loop exists in the line.

- gh-135386 [https://github.com/python/cpython/issues/135386]: Fix opening a **`dbm.sqlite3`** database for reading from read-only file or directory.

- gh-135444 [https://github.com/python/cpython/issues/135444]: Fix **`asyncio.DatagramTransport.sendto()`** to account for datagram header size when data cannot be sent.

- gh-126631 [https://github.com/python/cpython/issues/126631]: Fix **`multiprocessing`** `forkserver` bug which prevented `__main__` from being preloaded.

- gh-135307 [https://github.com/python/cpython/issues/135307]: **`email`**: Fix exception in `set_content()` when encoding text and max_line_length is set to `0` or `None` (unlimited).

- gh-134453 [https://github.com/python/cpython/issues/134453]: Fixed **`subprocess.Popen.communicate()`** `input=` handling of **`memoryview`** instances that were non-byte shaped on POSIX platforms. Those are now properly cast to a byte shaped view instead of truncating the input. Windows platforms did not have this bug.

- gh-134698 [https://github.com/python/cpython/issues/134698]: Fix a crash when calling methods of **`ssl.SSLContext`** or **`ssl.SSLSocket`** across multiple threads.

- gh-125996 [https://github.com/python/cpython/issues/125996]: Fix thread safety of **`collections.OrderedDict`**. Patch by Kumar Aditya.

- gh-133789 [https://github.com/python/cpython/issues/133789]: Fix unpickling of **`pathlib`** objects that were pickled in Python 3.13.

- gh-127081 [https://github.com/python/cpython/issues/127081]: Fix libc thread safety issues with **`dbm`** by performing stateful operations in critical sections.

- gh-132551 [https://github.com/python/cpython/issues/132551]: Make **`io.BytesIO`** safe in free-threaded build.

- gh-131788 [https://github.com/python/cpython/issues/131788]: Make `ResourceTracker.send` from **`multiprocessing`** re-entrant safe

- gh-118981 [https://github.com/python/cpython/issues/118981]: Fix potential hang in `multiprocessing.popen_spawn_posix` that can happen when the child proc dies early by closing the child fds right away.

- gh-102431 [https://github.com/python/cpython/issues/102431]: Clarify constraints for “logical” arguments in methods of **`decimal.Context`**.

- gh-78319 [https://github.com/python/cpython/issues/78319]: UTF8 support for the IMAP APPEND command has been made RFC compliant.

- bpo-38735 [https://bugs.python.org/issue?@action=redirect&bpo=38735]: Fix failure when importing a module from the root directory on unix-like platforms with sys.pycache_prefix set.

- bpo-41839 [https://bugs.python.org/issue?@action=redirect&bpo=41839]: Allow negative priority values from `os.sched_get_priority_min()` and `os.sched_get_priority_max()` functions.

IDLE

- gh-96491 [https://github.com/python/cpython/issues/96491]: Deduplicate version number in IDLE shell title bar after saving to a file.
- gh-139742 [https://github.com/python/cpython/issues/139742]: Colorize t-string prefixes for template strings in IDLE, as done for f-string prefixes.

Documentation

- gh-141994 [https://github.com/python/cpython/issues/141994]: `xml.sax.handler`: Make Documentation of `xml.sax.handler.feature_external_ges` warn of opening up to external entity attacks [https://en.wikipedia.org/wiki/XML_external_entity_attack]. Patch by Sebastian Pipping.
- gh-140578 [https://github.com/python/cpython/issues/140578]: Remove outdated sencence in the documentation for `multiprocessing`, that implied that `concurrent.futures.ThreadPoolExecutor` did not exist.

Core and Builtins

- gh-142048 [https://github.com/python/cpython/issues/142048]: Fix quadratically increasing garbage collection delays in free-threaded build.

Library

- gh-116738 [https://github.com/python/cpython/issues/116738]: Fix thread safety issue with **`re`** scanner objects in free-threaded builds.

Core and Builtins

- gh-141930 [https://github.com/python/cpython/issues/141930]: When importing a module, use Python’s regular file object to ensure that writes to `.pyc` files are complete or an appropriate error is raised.
- gh-120158 [https://github.com/python/cpython/issues/120158]: Fix inconsistent state when enabling or disabling monitoring events too many times.
- gh-139653 [https://github.com/python/cpython/issues/139653]: Only raise a `RecursionError` or trigger a fatal error if the stack pointer is both below the limit pointer *and* above the stack base. If outside of these bounds assume that it is OK. This prevents false positives when user-space threads swap stacks.
- gh-139103 [https://github.com/python/cpython/issues/139103]: Improve multithreaded scaling of dataclasses on the free-threaded build.
- gh-141579 [https://github.com/python/cpython/issues/141579]: Fix **`sys.activate_stack_trampoline()`** to properly support the `perf_jit` backend. Patch by Pablo Galindo.
- gh-114203 [https://github.com/python/cpython/issues/114203]: Skip locking if object is already locked by two-mutex critical

section.

- gh-141528 [https://github.com/python/cpython/issues/141528]: Suggest using **`concurrent.interpreters.Interpreter.close()`** instead of the private `_interpreters.destroy` function when warning about remaining subinterpreters. Patch by Sergey Miryanov.
- gh-141312 [https://github.com/python/cpython/issues/141312]: Fix the assertion failure in the `__setstate__` method of the range iterator when a non-integer argument is passed. Patch by Sergey Miryanov.

Library

- gh-116738 [https://github.com/python/cpython/issues/116738]: Make csv module thread-safe on the free threaded build.

Core and Builtins

- gh-140939 [https://github.com/python/cpython/issues/140939]: Fix memory leak when **`bytearray`** or **`bytes`** is formated with the `%*b` format with a large width that results in a **`MemoryError`**.

Library

- gh-140260 [https://github.com/python/cpython/issues/140260]: Fix **`struct`** data race in endian table initialization with subinterpreters. Patch by Shamil Abdulaev.

Core and Builtins

- gh-140530 [https://github.com/python/cpython/issues/140530]: Fix a reference leak when `raise exc from cause` fails. Patch by Bénédikt Tran.
- gh-140373 [https://github.com/python/cpython/issues/140373]: Correctly emit `PY_UNWIND` event when generator object is

closed. Patch by Mikhail Efimov.

- gh-140576 [https://github.com/python/cpython/issues/140576]: Fixed crash in **`tokenize.generate_tokens()`** in case of specific incorrect input. Patch by Mikhail Efimov.
- gh-140551 [https://github.com/python/cpython/issues/140551]: Fixed crash in **`dict`** if **`dict.clear()`** is called at the lookup stage. Patch by Mikhail Efimov and Inada Naoki.
- gh-140517 [https://github.com/python/cpython/issues/140517]: Fixed a reference leak when iterating over the result of **`map()`** with `strict=True` when the input iterables have different lengths. Patch by Mikhail Efimov.
- gh-140471 [https://github.com/python/cpython/issues/140471]: Fix potential buffer overflow in **`ast.AST`** node initialization when encountering malformed **`_fields`** containing non-**`str`**.
- gh-140431 [https://github.com/python/cpython/issues/140431]: Fix a crash in Python's garbage collector due to partially initialized coroutine objects when coroutine origin tracking depth is enabled (**`sys.set_coroutine_origin_tracking_depth()`**).

Library

- gh-140398 [https://github.com/python/cpython/issues/140398]: Fix memory leaks in **`readline`** functions **`read_init_file()`**, **`read_history_file()`**, **`write_history_file()`**, and **`append_history_file()`** when **`PySys_Audit()`** fails.

Core and Builtins

- gh-140406 [https://github.com/python/cpython/issues/140406]: Fix memory leak when an object's **`__hash__()`** method returns an object that isn't an **`int`**.
- gh-140358 [https://github.com/python/cpython/issues/140358]: Restore elapsed time and unreachable object count in GC debug output. These were inadvertently removed

during a refactor of `gc.c`. The debug log now again reports elapsed collection time and the number of unreachable objects. Contributed by Pål Grønås Drange.

- gh-140306 [https://github.com/python/cpython/issues/140306]: Fix memory leaks in cross-interpreter channel operations and shared namespace handling.
- gh-140301 [https://github.com/python/cpython/issues/140301]: Fix memory leak of `PyConfig` in subinterpreters.
- gh-140257 [https://github.com/python/cpython/issues/140257]: Fix data race between interpreter_clear() and take_gil() on eval_breaker during finalization with daemon threads.
- gh-139951 [https://github.com/python/cpython/issues/139951]: Fixes a regression in GC performance for a growing heap composed mostly of small tuples.
  - Counts number of actually tracked objects, instead of trackable objects. This ensures that untracking tuples has the desired effect of reducing GC overhead.
  - Does not track most untrackable tuples during creation. This prevents large numbers of small tuples causing excessive GCs.
- gh-140104 [https://github.com/python/cpython/issues/140104]: Fix a bug with exception handling in the JIT. Patch by Ken Jin. Bug reported by Daniel Diniz.
- gh-140061 [https://github.com/python/cpython/issues/140061]: Fixing the checking of whether an object is uniquely referenced to ensure free-threaded compatibility. Patch by Sergey Miryanov.
- gh-140067 [https://github.com/python/cpython/issues/140067]: Fix memory leak in sub-interpreter creation.
- gh-140000 [https://github.com/python/cpython/issues/140000]: Fix potential memory leak when a reference cycle exists between an instance of **`typing.TypeAliasType`**, **`typing.TypeVar`**, **`typing.ParamSpec`**, or **`typing.TypeVarTuple`** and its `__name__` attribute. Patch by Mikhail Efimov.

- gh-139914 [https://github.com/python/cpython/issues/139914]: Restore support for HP PA-RISC, which has an upwards-growing stack.
- gh-139988 [https://github.com/python/cpython/issues/139988]: Fix a memory leak when failing to create a **`Union`** type. Patch by Bénédikt Tran.
- gh-139748 [https://github.com/python/cpython/issues/139748]: Fix reference leaks in error branches of functions accepting path strings or bytes such as **`compile()`** and **`os.system()`**. Patch by Bénédikt Tran.
- gh-139516 [https://github.com/python/cpython/issues/139516]: Fix lambda colon erroneously start format spec in f-string in tokenizer.
- gh-139640 [https://github.com/python/cpython/issues/139640]: **`ast.parse()`** no longer emits syntax warnings for `return`/`break`/`continue` in `finally` (see **PEP 765** [https://peps.python.org/pep-0765/]) – they are only emitted during compilation.
- gh-139640 [https://github.com/python/cpython/issues/139640]: Fix swallowing some syntax warnings in different modules if they accidentally have the same message and are emitted from the same line. Fix duplicated warnings in the `finally` block.
- gh-63161 [https://github.com/python/cpython/issues/63161]: Support non-UTF-8 shebang and comments in Python source files if non-UTF-8 encoding is specified. Detect decoding error in comments for default (UTF-8) encoding. Show the line and position of decoding error for default encoding in a traceback. Show the line containing the coding cookie when it conflicts with the BOM in a traceback.

Library

- gh-116738 [https://github.com/python/cpython/issues/116738]: Make **`mmap`** thread-safe on the free threaded build.

Core and Builtins

- gh-138558 [https://github.com/python/cpython/issues/138558]: Fix handling of unusual t-string annotations in annotationlib. Patch by Dave Peck.
- gh-134466 [https://github.com/python/cpython/issues/134466]: Don’t run PyREPL in a degraded environment where setting termios attributes is not allowed.
- gh-138944 [https://github.com/python/cpython/issues/138944]: Fix **`SyntaxError`** message when invalid syntax appears on the same line as a valid `import ... as ...` or `from ... import ... as ...` statement. Patch by Brian Schubert.
- gh-105487 [https://github.com/python/cpython/issues/105487]: Remove non-existent **`__copy__()`**, **`__deepcopy__()`**, and **`__bases__`** from the **`__dir__()`** entries of **`types.GenericAlias`**.
- gh-69605 [https://github.com/python/cpython/issues/69605]: Fix some standard library submodules missing from the REPL auto-completion of imports.

Library

- gh-116738 [https://github.com/python/cpython/issues/116738]: Make **`cProfile`** thread-safe on the free threaded build.
- gh-138004 [https://github.com/python/cpython/issues/138004]: On Solaris/Illumos platforms, thread names are now encoded as ASCII to avoid errors on systems (e.g. OpenIndiana) that don’t support non-ASCII names.

Core and Builtins

- gh-137433 [https://github.com/python/cpython/issues/137433]: Fix a potential deadlock in the free threading build when daemon threads enable or disable profiling or tracing while the main thread is shutting down the interpreter.

- gh-137400 [https://github.com/python/cpython/issues/137400]: Fix a crash in the free threading build when disabling profiling or tracing across all threads with `PyEval_SetProfileAllThreads()` or `PyEval_SetTraceAllThreads()` or their Python equivalents `threading.settrace_all_threads()` and `threading.setprofile_all_threads()`.
- gh-58124 [https://github.com/python/cpython/issues/58124]: Fix name of the Python encoding in Unicode errors of the code page codec: use "cp65000" and "cp65001" instead of "CP_UTF7" and "CP_UTF8" which are not valid Python code names. Patch by Victor Stinner.
- gh-132657 [https://github.com/python/cpython/issues/132657]: Improve performance of `frozenset` by removing locks in the free-threading build.
- gh-133400 [https://github.com/python/cpython/issues/133400]: Fixed Ctrl+D (^D) behavior in _pyrepl module to match old pre-3.13 REPL behavior.
- gh-128640 [https://github.com/python/cpython/issues/128640]: Fix a crash when using threads inside of a subinterpreter.

C API

- gh-137422 [https://github.com/python/cpython/issues/137422]: Fix free threading race condition in `PyImport_AddModuleRef()`. It was previously possible for two calls to the function return two different objects, only one of which was stored in `sys.modules`.
- gh-140042 [https://github.com/python/cpython/issues/140042]: Removed the sqlite3_shutdown call that could cause closing connections for sqlite when used with multiple sub interpreters.
- gh-141042 [https://github.com/python/cpython/issues/141042]: Make qNaN in `PyFloat_Pack2()` and `PyFloat_Pack4()`, if while conversion to a narrower precision floating-point format

— the remaining after truncation payload will be zero. Patch by Sergey B Kirpichev.

- gh-140487 [https://github.com/python/cpython/issues/140487]: Fix **`Py_RETURN_NOTIMPLEMENTED`** in limited C API 3.11 and older: don’t treat `Py_NotImplemented` as immortal. Patch by Victor Stinner.
- gh-140153 [https://github.com/python/cpython/issues/140153]: Fix **`Py_REFCNT()`** definition on limited C API 3.11-3.13. Patch by Victor Stinner.
- gh-139653 [https://github.com/python/cpython/issues/139653]: Add **`PyUnstable_ThreadState_SetStackProtection()`** and **`PyUnstable_ThreadState_ResetStackProtection()`** functions to set the stack protection base address and stack protection size of a Python thread state. Patch by Victor Stinner.

Build

- gh-141808 [https://github.com/python/cpython/issues/141808]: Do not generate the jit stencils twice in case of PGO builds on Windows.

- gh-141784 [https://github.com/python/cpython/issues/141784]: Fix `_remote_debugging_module.c` compilation on 32-bit Linux. Include Python.h before system headers to make sure that `_remote_debugging_module.c` uses the same types (ABI) than Python. Patch by Victor Stinner.

- gh-140768 [https://github.com/python/cpython/issues/140768]: Warn when the WASI SDK version doesn’t match what’s supported.

- gh-140513 [https://github.com/python/cpython/issues/140513]: Generate a clear compilation error when `_Py_TAIL_CALL_INTERP` is enabled but either `preserve_none` or `musttail` is not supported.

- gh-140189 [https://github.com/python/cpython/issues/140189]: iOS builds were added to CI.
- gh-138489 [https://github.com/python/cpython/issues/138489]: When cross-compiling for WASI by `build_wasm` or `build_emscripten`, the `build-details.json` step is now included in the build process, just like with native builds.

  This fixes the `libinstall` task which requires the `build-details.json` file during the process.
- gh-137618 [https://github.com/python/cpython/issues/137618]: `PYTHON_FOR_REGEN` now requires Python 3.10 to Python 3.15. Patch by Adam Turner.
- gh-123681 [https://github.com/python/cpython/issues/123681]: Check the `strftime()` behavior at runtime instead of at the compile time to support cross-compiling. Remove the internal macro `_Py_NORMALIZE_CENTURY`.

Python 3.14.0 final

Release date: 2025-10-07

macOS

- gh-124111 [https://github.com/python/cpython/issues/124111]: Update macOS installer to use Tcl/Tk 8.6.17.
- gh-139573 [https://github.com/python/cpython/issues/139573]: Updated bundled version of OpenSSL to 3.0.18.

Windows

- gh-139573 [https://github.com/python/cpython/issues/139573]: Updated bundled version of OpenSSL to 3.0.18.

Tools/Demos

- gh-139330 [https://github.com/python/cpython/issues/139330]: SBOM generation tool didn't cross-check the version and checksum values against the `Modules/expat/refresh.sh` script, leading to the values becoming out-of-date during routine updates.
- gh-132006 [https://github.com/python/cpython/issues/132006]: XCframeworks now include privacy manifests to satisfy Apple App Store submission requirements.
- gh-138171 [https://github.com/python/cpython/issues/138171]: A script for building an iOS XCframework was added. As part of this change, the top level `iOS` folder has been moved to be a subdirectory of the `Apple` folder.

Security

- gh-139400 [https://github.com/python/cpython/issues/139400]: **`xml.parsers.expat`**: Make sure that parent Expat parsers are only garbage-collected once they are no longer referenced by subparsers created by **`ExternalEntityParserCreate()`**. Patch by Sebastian Pipping.

Library

- gh-139312 [https://github.com/python/cpython/issues/139312]: Upgrade bundled libexpat to 2.7.3

Python 3.14.0 release candidate 3

Release date: 2025-09-18

Windows

- gh-138896 [https://github.com/python/cpython/issues/138896]: Fix error installing C runtime on non-updated Windows machines

Tools/Demos

- gh-137873 [https://github.com/python/cpython/issues/137873]: The iOS test runner has been simplified, resolving some issues that have been observed using the runner in GitHub Actions and Azure Pipelines test environments.

Security

- gh-135661 [https://github.com/python/cpython/issues/135661]: Fix CDATA section parsing in `html.parser.HTMLParser` according to the HTML5 standard: `] ]>` and `]] >` no longer end the CDATA section. Add private method `_set_support_cdata()` which can be used to specify how to parse `<[CDATA[` — as a CDATA section in foreign content (SVG or MathML) or as a bogus comment in the HTML namespace.

Library

- gh-138998 [https://github.com/python/cpython/issues/138998]: Update bundled libexpat to 2.7.2
- gh-118803 [https://github.com/python/cpython/issues/118803]: Add back `collections.abc.ByteString` and `typing.ByteString`. Both had been removed in prior alpha, beta and release candidates for Python 3.14, but their removal has now been postponed to Python 3.17.
- gh-137226 [https://github.com/python/cpython/issues/137226]: Fix `typing.get_type_hints()` calls on generic `typing.TypedDict` classes defined with string annotations.
- gh-138804 [https://github.com/python/cpython/issues/138804]: Raise `TypeError` instead of `AttributeError` when an argument of

incorrect type is passed to `shlex.quote()`. This restores the behavior of the function prior to 3.14.

- gh-128636 [https://github.com/python/cpython/issues/128636]: Fix crash in PyREPL when os.environ is overwritten with an invalid value for mac
- gh-138514 [https://github.com/python/cpython/issues/138514]: Raise `ValueError` when a multi-character string is passed to the *echo_char* parameter of `getpass.getpass()`. Patch by Benjamin Johnson.
- gh-138515 [https://github.com/python/cpython/issues/138515]: `email` is added to Emscripten build.
- gh-99948 [https://github.com/python/cpython/issues/99948]: `ctypes.util.find_library()` now works in Emscripten build.
- gh-138253 [https://github.com/python/cpython/issues/138253]: Add the *block* parameter in the **`put()`** and **`get()`** methods of the `concurrent.interpreters` queues for compatibility with the `queue.Queue` interface.
- gh-138133 [https://github.com/python/cpython/issues/138133]: Prevent infinite traceback loop when sending CTRL^C to Python through `strace`.
- gh-134869 [https://github.com/python/cpython/issues/134869]: Fix an issue where pressing Ctrl+C during tab completion in the REPL would leave the autocompletion menu in a corrupted state.
- gh-90548 [https://github.com/python/cpython/issues/90548]: Fix `musl` detection for `platform.libc_ver()` on Alpine Linux if compiled with –strip-all.
- gh-136134 [https://github.com/python/cpython/issues/136134]: **`SMTP.auth_cram_md5()`** now raises an `SMTPException` instead of a `ValueError` if Python has been built without MD5 support. In particular, `SMTP` clients will not attempt to use this method even if the remote server is assumed to support it. Patch by Bénédikt Tran.
- gh-136134 [https://github.com/python/cpython/issues/136134]: `IMAP4.login_cram_md5` now raises an `IMAP4.error` if CRAM-

MD5 authentication is not supported. Patch by Bénédikt Tran.
- gh-134953 [https://github.com/python/cpython/issues/134953]: Expand `_colorize` theme with `keyword_constant` and implement in repl.

Core and Builtins

- gh-71810 [https://github.com/python/cpython/issues/71810]: Raise **`OverflowError`** for `(-1).to_bytes()` for signed conversions when bytes count is zero. Patch by Sergey B Kirpichev.
- gh-138192 [https://github.com/python/cpython/issues/138192]: Fix **`contextvars`** initialization so that all subinterpreters are assigned the **`MISSING`** value.
- gh-138479 [https://github.com/python/cpython/issues/138479]: Fix a crash when a generic object’s `__typing_subst__` returns an object that isn’t a **`tuple`**.
- gh-138372 [https://github.com/python/cpython/issues/138372]: Fix **`SyntaxWarning`** emitted for erroneous subscript expressions involving template string literals. Patch by Brian Schubert.
- gh-138318 [https://github.com/python/cpython/issues/138318]: The default REPL now avoids highlighting built-in names (for instance **`set`** or **`format()`**) when they are used as attribute names (for instance in `value.set` or `text.format`).
- gh-138349 [https://github.com/python/cpython/issues/138349]: Fix crash in certain cases where a module contains both a module-level annotation and a comprehension.
- gh-137384 [https://github.com/python/cpython/issues/137384]: Fix a crash when using the **`warnings`** module in a finalizer at shutdown. Patch by Kumar Aditya.
- gh-137883 [https://github.com/python/cpython/issues/137883]: Fix runaway recursion when calling a function with keyword arguments.
- gh-137079 [https://github.com/python/cpython/issues/137079]: Fix keyword typo recognition when parsing files. Patch by

Pablo Galindo.

- gh-137728 [https://github.com/python/cpython/issues/137728]: Fix the JIT’s handling of many local variables. This previously caused a segfault.
- gh-137576 [https://github.com/python/cpython/issues/137576]: Fix for incorrect source code being shown in tracebacks from the Basic REPL when **`PYTHONSTARTUP`** is given. Patch by Adam Hartz.

Python 3.14.0 release candidate 2

Release date: 2025-08-14

macOS

- gh-137450 [https://github.com/python/cpython/issues/137450]: macOS installer shell path management improvements: separate the installer `Shell profile updater` postinstall script from the `Update Shell Profile.command` to enable more robust error handling.
- gh-137134 [https://github.com/python/cpython/issues/137134]: Update macOS installer to ship with SQLite version 3.50.4.

Windows

- gh-137134 [https://github.com/python/cpython/issues/137134]: Update Windows installer to ship with SQLite 3.50.4.

Library

- gh-137426 [https://github.com/python/cpython/issues/137426]: Remove the code deprecation of `importlib.abc.ResourceLoader`. It is documented as deprecated, but left for backwards compatibility with other classes in `importlib.abc`.

- gh-137282 [https://github.com/python/cpython/issues/137282]: Fix tab completion and **`dir()`** on **`concurrent.futures`**.
- gh-137257 [https://github.com/python/cpython/issues/137257]: Bump the version of pip bundled in ensurepip to version 25.2
- gh-137226 [https://github.com/python/cpython/issues/137226]: Fix behavior of **`annotationlib.ForwardRef.evaluate()`** when the *type_params* parameter is passed and the name of a type param is also present in an enclosing scope.
- gh-130522 [https://github.com/python/cpython/issues/130522]: Fix unraisable **`TypeError`** raised during interpreter shutdown in the **`threading`** module.
- gh-137059 [https://github.com/python/cpython/issues/137059]: Fix handling of file URLs with a Windows drive letter in the URL authority by **`urllib.request.url2pathname()`**. This fixes a regression in earlier pre-releases of Python 3.14.
- gh-130577 [https://github.com/python/cpython/issues/130577]: **`tarfile`** now validates archives to ensure member offsets are non-negative. (Contributed by Alexander Enrique Urieles Nieto in gh-130577 [https://github.com/python/cpython/issues/130577].)
- gh-135228 [https://github.com/python/cpython/issues/135228]: When **`dataclasses`** replaces a class with a slotted dataclass, the original class can now be garbage collected again. Earlier changes in Python 3.14 caused this class to always remain in existence together with the replacement class synthesized by **`dataclasses`**.

Documentation

- gh-136155 [https://github.com/python/cpython/issues/136155]: We are now checking for fatal errors in EPUB builds in CI.

Core and Builtins

- gh-137400 [https://github.com/python/cpython/issues/137400]: Fix a crash in the free threading build when disabling profiling or tracing across all threads with **`PyEval_SetProfileAllThreads()`** or **`PyEval_SetTraceAllThreads()`** or their Python equivalents **`threading.settrace_all_threads()`** and **`threading.setprofile_all_threads()`**.
- gh-137314 [https://github.com/python/cpython/issues/137314]: Fixed a regression where raw f-strings incorrectly interpreted escape sequences in format specifications. Raw f-strings now properly preserve literal backslashes in format specs, matching the behavior from Python 3.11. For example, `rf"{obj:\xFF}"` now correctly produces `'\\xFF'` instead of `'ÿ'`. Patch by Pablo Galindo.
- gh-137308 [https://github.com/python/cpython/issues/137308]: A standalone docstring in a node body is optimized as a **`pass`** statement to ensure that the node’s body is never empty. There was a **`ValueError`** in **`compile()`** otherwise.
- gh-137288 [https://github.com/python/cpython/issues/137288]: Fix bug where some bytecode instructions of a boolean expression are not associated with the correct exception handler.
- gh-134291 [https://github.com/python/cpython/issues/134291]: Remove some newer macOS API usage from the JIT compiler in order to restore compatibility with older OSX 10.15 deployment targets.
- gh-131338 [https://github.com/python/cpython/issues/131338]: Disable computed stack limit checks on non-glibc linux platforms to fix crashes on deep recursion.
- gh-136870 [https://github.com/python/cpython/issues/136870]: Fix data races while de-instrumenting bytecode of code objects running concurrently in threads.

C API

- gh-137573 [https://github.com/python/cpython/issues/137573]: Mark `_PyOptimizer_Optimize` as **`Py_NO_INLINE`** to prevent stack overflow crashes on macOS.

Build

- gh-132339 [https://github.com/python/cpython/issues/132339]: Add support for OpenSSL 3.5.

Python 3.14.0 release candidate 1

Release date: 2025-07-22

Tools/Demos

- gh-136251 [https://github.com/python/cpython/issues/136251]: Fixes and usability improvements for `Tools/wasm/emscripten/web_example`

Security

- gh-135661 [https://github.com/python/cpython/issues/135661]: Fix parsing attributes with whitespaces around the `=` separator in **`html.parser.HTMLParser`** according to the HTML5 standard.
- gh-118350 [https://github.com/python/cpython/issues/118350]: Fix support of escapable raw text mode (elements “textarea” and “title”) in **`html.parser.HTMLParser`**.

Library

- gh-136170 [https://github.com/python/cpython/issues/136170]: Removed the unreleased `zipfile.ZipFile.data_offset` property added in 3.14.0a7 as it wasn't fully clear which behavior it should have in some situations so the result was not always what a user might expect.

- gh-124621 [https://github.com/python/cpython/issues/124621]: pyrepl now works in Emscripten.
- gh-136874 [https://github.com/python/cpython/issues/136874]: Discard URL query and fragment in **`urllib.request.url2pathname()`**.
- gh-130645 [https://github.com/python/cpython/issues/130645]: Enable color help by default in **`argparse`**.
- gh-136549 [https://github.com/python/cpython/issues/136549]: Fix signature of **`threading.excepthook()`**.
- gh-136523 [https://github.com/python/cpython/issues/136523]: Fix **`wave.Wave_write`** emitting an unraisable when open raises.
- gh-52876 [https://github.com/python/cpython/issues/52876]: Add missing `keepends` (default `True`) parameter to **`codecs.StreamReaderWriter.readline()`** and **`codecs.StreamReaderWriter.readlines()`**.
- gh-136470 [https://github.com/python/cpython/issues/136470]: Correct **`concurrent.futures.InterpreterPoolExecutor`**'s default thread name.
- gh-136476 [https://github.com/python/cpython/issues/136476]: Fix a bug that was causing the `get_async_stack_trace` function to miss some frames in the stack trace.
- gh-136434 [https://github.com/python/cpython/issues/136434]: Fix docs generation of `UnboundItem` in **`concurrent.interpreters`** when running with **`-OO`**.
- gh-136380 [https://github.com/python/cpython/issues/136380]: Raises **`AttributeError`** when accessing **`concurrent.futures.InterpreterPoolExecutor`** and subinterpreters are not available.
- gh-134759 [https://github.com/python/cpython/issues/134759]: Fix **`UnboundLocalError`** in **`email.message.Message.get_payload()`** when the payload to decode is a **`bytes`** object. Patch by Kliment Lamonov.

- gh-134657 [https://github.com/python/cpython/issues/134657]: **`asyncio`**: Remove some private names from `asyncio.__all__`.

Core and Builtins

- gh-136801 [https://github.com/python/cpython/issues/136801]: Fix PyREPL syntax highlighting on match cases after multi-line case. Contributed by Olga Matoula.

Library

- gh-136421 [https://github.com/python/cpython/issues/136421]: Fix crash when initializing **`datetime`** concurrently.

Core and Builtins

- gh-136541 [https://github.com/python/cpython/issues/136541]: Fix some issues with the perf trampolines on x86-64 and aarch64. The trampolines were not being generated correctly for some cases, which could lead to the perf integration not working correctly. Patch by Pablo Galindo.

- gh-136517 [https://github.com/python/cpython/issues/136517]: Fixed a typo that prevented printing of uncollectable objects when the **`gc.DEBUG_UNCOLLECTABLE`** mode was set.

- gh-136525 [https://github.com/python/cpython/issues/136525]: Fix issue where per-thread bytecode was not instrumented for newly created threads.

- gh-132661 [https://github.com/python/cpython/issues/132661]: `Interpolation.expression` now has a default, the empty string.

- gh-132661 [https://github.com/python/cpython/issues/132661]: Reflect recent **PEP 750** [https://peps.python.org/pep-0750/] change.

  Disallow concatenation of `string.templatelib.Template` and **`str`**. Also, disallow implicit concatenation of t-string literals with string or f-string literals.

Library

- gh-116738 [https://github.com/python/cpython/issues/116738]: Make functions in **`grp`** thread-safe on the free threaded build.

Core and Builtins

- gh-135148 [https://github.com/python/cpython/issues/135148]: Fixed a bug where f-string debug expressions (using =) would incorrectly strip out parts of strings containing escaped quotes and # characters. Patch by Pablo Galindo.
- gh-133136 [https://github.com/python/cpython/issues/133136]: Limit excess memory usage in the free threading build when a large dictionary or list is resized and accessed by multiple threads.
- gh-91153 [https://github.com/python/cpython/issues/91153]: Fix a crash when a **`bytearray`** is concurrently mutated during item assignment.
- gh-127971 [https://github.com/python/cpython/issues/127971]: Fix off-by-one read beyond the end of a string in string search.

C API

- gh-112068 [https://github.com/python/cpython/issues/112068]: Revert support of nullable arguments in **`PyArg_Parse()`**.
- gh-133296 [https://github.com/python/cpython/issues/133296]: New variants for the critical section API that accept one or

two `PyMutex` pointers rather than `PyObject` instances are now public in the non-limited C API.
- gh-134009 [https://github.com/python/cpython/issues/134009]: Expose `PyMutex_IsLocked()` as part of the public C API.

Build

- gh-135621 [https://github.com/python/cpython/issues/135621]: PyREPL no longer depends on the `curses` standard library. Contributed by Łukasz Langa.

Python 3.14.0 beta 4

Release date: 2025-07-08

Tools/Demos

- gh-135968 [https://github.com/python/cpython/issues/135968]: Stubs for `strip` are now provided as part of an iOS install.

- gh-133600 [https://github.com/python/cpython/issues/133600]: Backport file reorganization for Tools/wasm/wasi.

  This should make backporting future code changes easier. It also simplifies instructions around how to do WASI builds in the devguide.

Tests

- gh-135966 [https://github.com/python/cpython/issues/135966]: The iOS testbed now handles the `app_packages` folder as a site directory.
- gh-135494 [https://github.com/python/cpython/issues/135494]: Fix regrtest to support excluding tests from `--pgo` tests. Patch by Victor Stinner.

Security

- gh-136053 [https://github.com/python/cpython/issues/136053]: **`marshal`**: fix a possible crash when deserializing **`slice`** objects.
- gh-135661 [https://github.com/python/cpython/issues/135661]: Fix parsing start and end tags in **`html.parser.HTMLParser`** according to the HTML5 standard.
    - Whitespaces no longer accepted between `</` and the tag name. E.g. `</ script>` does not end the script section.
    - Vertical tabulation (`\v`) and non-ASCII whitespaces no longer recognized as whitespaces. The only whitespaces are `\t\n\r\f` and space.
    - Null character (U+0000) no longer ends the tag name.
    - Attributes and slashes after the tag name in end tags are now ignored, instead of terminating after the first `>` in quoted attribute value. E.g. `</script/foo=">"/>`.
    - Multiple slashes and whitespaces between the last attribute and closing `>` are now ignored in both start and end tags. E.g. `<a foo=bar/ //>`.
    - Multiple `=` between attribute name and value are no longer collapsed. E.g. `<a foo==bar>` produces attribute "foo" with value "=bar".
    - [Reverted in gh-136927 [https://github.com/python/cpython/issues/136927]] Whitespaces between the `=` separator and attribute name or value are no longer ignored. E.g. `<a foo =bar>` produces two attributes "foo" and "=bar", both with value None; `<a foo= bar>` produces two attributes: "foo" with value "" and "bar" with value None.
- gh-102555 [https://github.com/python/cpython/issues/102555]: Fix comment parsing in **`html.parser.HTMLParser`** according to

the HTML5 standard. `--!>` now ends the comment. `-- >` no longer ends the comment. Support abnormally ended empty comments `<-->` and `<--->`.

Library

- gh-136286 [https://github.com/python/cpython/issues/136286]: Fix pickling failures for protocols 0 and 1 for many objects realted to subinterpreters.
- gh-136316 [https://github.com/python/cpython/issues/136316]: Improve support for evaluating nested forward references in `typing.evaluate_forward_ref()`.
- gh-85702 [https://github.com/python/cpython/issues/85702]: If `zoneinfo._common.load_tzdata` is given a package without a resource a `zoneinfo.ZoneInfoNotFoundError` is raised rather than a `PermissionError`. Patch by Victor Stinner.
- gh-136028 [https://github.com/python/cpython/issues/136028]: Fix parsing month names containing “İ” (U+0130, LATIN CAPITAL LETTER I WITH DOT ABOVE) in `time.strptime()`. This affects locales az_AZ, ber_DZ, ber_MA and crh_UA.
- gh-135995 [https://github.com/python/cpython/issues/135995]: In the palmos encoding, make byte `0x9b` decode to `›` (U+203A - SINGLE RIGHT-POINTING ANGLE QUOTATION MARK).
- gh-53203 [https://github.com/python/cpython/issues/53203]: Fix `time.strptime()` for `%c` and `%x` formats on locales byn_ER, wal_ET and lzh_TW, and for `%X` format on locales ar_SA, bg_BG and lzh_TW.
- gh-91555 [https://github.com/python/cpython/issues/91555]: An earlier change, which was introduced in 3.14.0b2, has been reverted. It disabled logging for a logger during handling of log messages for that logger. Since the reversion, the behaviour should be as it was before 3.14.0b2.
- gh-135878 [https://github.com/python/cpython/issues/135878]: Fixes a crash of `types.SimpleNamespace` on free threading builds,

when several threads were calling its `__repr__()` method at the same time.

- gh-135836 [https://github.com/python/cpython/issues/135836]: Fix `IndexError` in `asyncio.loop.create_connection()` that could occur when non-`OSError` exception is raised during connection and socket's `close()` raises **`OSError`**.
- gh-135836 [https://github.com/python/cpython/issues/135836]: Fix `IndexError` in `asyncio.loop.create_connection()` that could occur when the Happy Eyeballs algorithm resulted in an empty exceptions list during connection attempts.
- gh-135855 [https://github.com/python/cpython/issues/135855]: Raise `TypeError` instead of `SystemError` when **`_interpreters.set___main___attrs()`** is passed a non-dict object. Patch by Brian Schubert.
- gh-135815 [https://github.com/python/cpython/issues/135815]: `netrc`: skip security checks if `os.getuid()` is missing. Patch by Bénédikt Tran.
- gh-135640 [https://github.com/python/cpython/issues/135640]: Address bug where it was possible to call `xml.etree.ElementTree.ElementTree.write()` on an ElementTree object with an invalid root element. This behavior blanked the file passed to `write` if it already existed.
- gh-135645 [https://github.com/python/cpython/issues/135645]: Added `supports_isolated_interpreters` field to `sys.implementation`.
- gh-135646 [https://github.com/python/cpython/issues/135646]: Raise consistent `NameError` exceptions in `annotationlib.ForwardRef.evaluate()`
- gh-135557 [https://github.com/python/cpython/issues/135557]: Fix races on `heapq` updates and `list` reads on the free threaded build.
- gh-119180 [https://github.com/python/cpython/issues/119180]: Only fetch globals and locals if necessary in `annotationlib.get_annotations()`

- gh-135561 [https://github.com/python/cpython/issues/135561]: Fix a crash on DEBUG builds when an HACL* HMAC routine fails. Patch by Bénédikt Tran.
- gh-135487 [https://github.com/python/cpython/issues/135487]: Fix **`reprlib.Repr.repr_int()`** when given integers with more than **`sys.get_int_max_str_digits()`** digits. Patch by Bénédikt Tran.
- gh-135335 [https://github.com/python/cpython/issues/135335]: **`multiprocessing`**: Flush `stdout` and `stderr` after preloading modules in the `forkserver`.
- gh-135069 [https://github.com/python/cpython/issues/135069]: Fix the “Invalid error handling” exception in **`encodings.idna.IncrementalDecoder`** to correctly replace the ‘errors’ parameter.
- gh-130662 [https://github.com/python/cpython/issues/130662]: +Accept leading zeros in precision and width fields for +:class:**`Decimal`** formatting, for example `format(Decimal(1.25), '.016f')`.
- gh-130662 [https://github.com/python/cpython/issues/130662]: Accept leading zeros in precision and width fields for **`Fraction`** formatting, for example `format(Fraction(1, 3), '.016f')`.
- gh-87790 [https://github.com/python/cpython/issues/87790]: Support underscore and comma as thousands separators in the fractional part for **`Fraction`**’s formatting. Patch by Sergey B Kirpichev.
- gh-87790 [https://github.com/python/cpython/issues/87790]: Support underscore and comma as thousands separators in the fractional part for **`Decimal`**’s formatting. Patch by Sergey B Kirpichev.
- gh-130664 [https://github.com/python/cpython/issues/130664]: Handle corner-case for **`Fraction`**’s formatting: treat zero-padding (preceding the width field by a zero (`'0'`) character) as an equivalent to a fill character of `'0'` with an alignment type of `'='`, just as in case of **`float`**’s.

Documentation

- gh-136155 [https://github.com/python/cpython/issues/136155]: EPUB builds are fixed by excluding non-XHTML-compatible tags.

Core and Builtins

- gh-109700 [https://github.com/python/cpython/issues/109700]: Fix memory error handling in **`PyDict_SetDefault()`**.
- gh-78465 [https://github.com/python/cpython/issues/78465]: Fix error message for `cls.__new__(cls, ...)` where `cls` is not instantiable builtin or extension type (with `tp_new` set to `NULL`).
- gh-129958 [https://github.com/python/cpython/issues/129958]: Differentiate between t-strings and f-strings in syntax error for newlines in format specifiers of single-quoted interpolated strings.
- gh-135871 [https://github.com/python/cpython/issues/135871]: Non-blocking mutex lock attempts now return immediately when the lock is busy instead of briefly spinning in the free threading build.
- gh-135106 [https://github.com/python/cpython/issues/135106]: Restrict the trashcan mechanism to GC’ed objects and untrack them while in the trashcan to prevent the GC and trashcan mechanisms conflicting.
- gh-135607 [https://github.com/python/cpython/issues/135607]: Fix potential **`weakref`** races in an object’s destructor on the free threaded build.
- gh-135608 [https://github.com/python/cpython/issues/135608]: Fix a crash in the JIT involving attributes of modules.
- gh-135543 [https://github.com/python/cpython/issues/135543]: Emit `sys.remote_exec` audit event when **`sys.remote_exec()`** is called and migrate `remote_debugger_script` to `cpython.remote_debugger_script`.

- gh-134280 [https://github.com/python/cpython/issues/134280]: Disable constant folding for `~` with a boolean argument. This moves the deprecation warning from compile time to runtime.

C API

- gh-135906 [https://github.com/python/cpython/issues/135906]: Fix compilation errors when compiling the internal headers with a C++ compiler.

Build

- gh-134273 [https://github.com/python/cpython/issues/134273]: Add support for configuring compiler flags for the JIT with `CFLAGS_JIT`

Python 3.14.0 beta 3

Release date: 2025-06-17

Windows

- gh-135099 [https://github.com/python/cpython/issues/135099]: Fix a crash that could occur on Windows when a background thread waits on a **`PyMutex`** while the main thread is shutting down the interpreter.

Tests

- gh-132815 [https://github.com/python/cpython/issues/132815]: Fix test__opcode: add `JUMP_BACKWARD` to specialization stats.
- gh-135489 [https://github.com/python/cpython/issues/135489]: Show verbose output for failing tests during PGO profiling step with –enable-optimizations.

- gh-135120 [https://github.com/python/cpython/issues/135120]: Add **`test.support.subTests()`**.

Security

- gh-135462 [https://github.com/python/cpython/issues/135462]: Fix quadratic complexity in processing specially crafted input in **`html.parser.HTMLParser`**. End-of-file errors are now handled according to the HTML5 specs – comments and declarations are automatically closed, tags are ignored.

- gh-135034 [https://github.com/python/cpython/issues/135034]: Fixes multiple issues that allowed `tarfile` extraction filters (`filter="data"` and `filter="tar"`) to be bypassed using crafted symlinks and hard links.

  Addresses **CVE 2024-12718** [https://www.cve.org/CVERecord?id=CVE-2024-12718], **CVE 2025-4138** [https://www.cve.org/CVERecord?id=CVE-2025-4138], **CVE 2025-4330** [https://www.cve.org/CVERecord?id=CVE-2025-4330], and **CVE 2025-4517** [https://www.cve.org/CVERecord?id=CVE-2025-4517].

Library

- gh-65697 [https://github.com/python/cpython/issues/65697]: **`configparser`**’s error message when attempting to write an invalid key is now more helpful.
- gh-135497 [https://github.com/python/cpython/issues/135497]: Fix **`os.getlogin()`** failing for longer usernames on BSD-based platforms.
- gh-135429 [https://github.com/python/cpython/issues/135429]: Fix the argument mismatch in `_lsprof` for `PY_THROW` event.
- gh-135368 [https://github.com/python/cpython/issues/135368]: Fix **`unittest.mock.Mock`** generation on **`dataclasses.dataclass()`** objects. Now all special attributes are set as it was

before gh-124429 [https://github.com/python/cpython/issues/124429].

- gh-133967 [https://github.com/python/cpython/issues/133967]: Do not normalize **`locale`** name ‘C.UTF-8’ to ‘en_US.UTF-8’.
- gh-135321 [https://github.com/python/cpython/issues/135321]: Raise a correct exception for values greater than 0x7fffffff for the `BINSTRING` opcode in the C implementation of **`pickle`**.
- gh-135276 [https://github.com/python/cpython/issues/135276]: Backported bugfixes in zipfile.Path from zipp 3.23. Fixed `.name`, `.stem` and other basename-based properties on Windows when working with a zipfile on disk.
- gh-135244 [https://github.com/python/cpython/issues/135244]: **`uuid`**: when the MAC address cannot be determined, the 48-bit node ID is now generated with a cryptographically-secure pseudo-random number generator (CSPRNG) as per **RFC 9562, §6.10.3** [https://datatracker.ietf.org/doc/html/rfc9562.html#section-6.10-3]. This affects **`uuid1()`** and **`uuid6()`**.
- gh-134970 [https://github.com/python/cpython/issues/134970]: Fix the “unknown action” exception in **`argparse.ArgumentParser.add_argument_group()`** to correctly replace the action class.
- gh-134718 [https://github.com/python/cpython/issues/134718]: **`ast.dump()`** now only omits `None` and `[]` values if they are default values.
- gh-134939 [https://github.com/python/cpython/issues/134939]: Add the **`concurrent.interpreters`** module. See **PEP 734** [https://peps.python.org/pep-0734/].
- gh-134885 [https://github.com/python/cpython/issues/134885]: Fix possible crash in the **`compression.zstd`** module related to setting parameter types. Patch by Jelle Zijlstra.
- gh-134857 [https://github.com/python/cpython/issues/134857]: Improve error report for **`doctest`**s run with **`unittest`**. Remove **`doctest`** module frames from tracebacks and redundant newline character from a failure message.

- gh-128840 [https://github.com/python/cpython/issues/128840]: Fix parsing long IPv6 addresses with embedded IPv4 address.
- gh-134637 [https://github.com/python/cpython/issues/134637]: Fix performance regression in calling a **`ctypes`** function pointer in free threading.
- gh-134696 [https://github.com/python/cpython/issues/134696]: Built-in HACL* and OpenSSL implementations of hash function constructors now correctly accept the same *documented* named arguments. For instance, **`md5()`** could be previously invoked as `md5(data=data)` or `md5(string=string)` depending on the underlying implementation but these calls were not compatible. Patch by Bénédikt Tran.
- gh-134151 [https://github.com/python/cpython/issues/134151]: **`email`**: Fix **`TypeError`** in **`email.utils.decode_params()`** when sorting **RFC 2231** [https://datatracker.ietf.org/doc/html/rfc2231.html] continuations that contain an unnumbered section.
- gh-134210 [https://github.com/python/cpython/issues/134210]: **`curses.window.getch()`** now correctly handles signals. Patch by Bénédikt Tran.
- gh-134152 [https://github.com/python/cpython/issues/134152]: **`email`**: Fix parsing of email message ID with invalid domain.
- gh-133489 [https://github.com/python/cpython/issues/133489]: **`random.getrandbits()`** can now generate more that 2^{31} bits. **`random.randbytes()`** can now generate more that 256 MiB.
- gh-132813 [https://github.com/python/cpython/issues/132813]: Improve error messages for incorrect types and values of **`csv.Dialect`** attributes.
- gh-132969 [https://github.com/python/cpython/issues/132969]: Prevent the **`ProcessPoolExecutor`** executor thread, which remains running when **`shutdown(wait=False)`**, from attempting to adjust the pool’s worker processes after the object state has already been reset during

shutdown. A combination of conditions, including a worker process having terminated abormally, resulted in an exception and a potential hang when the still-running executor thread attempted to replace dead workers within the pool.

- gh-127081 [https://github.com/python/cpython/issues/127081]: Fix libc thread safety issues with **`os`** by replacing `getlogin` with `getlogin_r` re-entrant version.
- gh-131884 [https://github.com/python/cpython/issues/131884]: Fix formatting issues in **`json.dump()`** when both *indent* and *skipkeys* are used.
- gh-130999 [https://github.com/python/cpython/issues/130999]: Avoid exiting the new REPL and offer suggestions even if there are non-string candidates when errors occur.

Documentation

- gh-135171 [https://github.com/python/cpython/issues/135171]: Document that the iterator for the leftmost **`for`** clause in the generator expression is created immediately.
- bpo-45210 [https://bugs.python.org/issue?@action=redirect&bpo=45210]: Document that error indicator may be set in tp_dealloc, and how to avoid clobbering it.

Core and Builtins

- gh-135496 [https://github.com/python/cpython/issues/135496]: Fix typo in the f-string conversion type error (“exclamanation” -> “exclamation”).
- gh-135371 [https://github.com/python/cpython/issues/135371]: Fixed **`asyncio`** debugging tools to properly display internal coroutine call stacks alongside external task dependencies. The `python -m asyncio ps` and `python -m asyncio pstree` commands now show complete execution context. Patch by Pablo Galindo.

Library

- gh-127319 [https://github.com/python/cpython/issues/127319]: Set the `allow_reuse_port` class variable to `False` on the XMLRPC, logging, and HTTP servers. This matches the behavior in prior Python releases, which is to not allow port reuse.

Core and Builtins

- gh-135171 [https://github.com/python/cpython/issues/135171]: Reverts the behavior of async generator expressions when created with object w/o __aiter__ method to the pre-3.13 behavior of raising a TypeError.
- gh-130077 [https://github.com/python/cpython/issues/130077]: Properly raise custom syntax errors when incorrect syntax containing names that are prefixes of soft keywords is encountered. Patch by Pablo Galindo.
- gh-135171 [https://github.com/python/cpython/issues/135171]: Reverts the behavior of generator expressions when created with a non-iterable to the pre-3.13 behavior of raising a TypeError. It is no longer possible to cause a crash in the debugger by altering the generator expression’s local variables. This is achieved by moving the `GET_ITER` instruction back to the creation of the generator expression and adding an additional check to `FOR_ITER`.

Library

- gh-116738 [https://github.com/python/cpython/issues/116738]: Make methods in **`heapq`** thread-safe on the free threaded build.

Core and Builtins

- gh-134876 [https://github.com/python/cpython/issues/134876]: Add support to **PEP 768** [https://peps.python.org/pep-0768/] remote debugging for Linux kernels which don't have CONFIG_CROSS_MEMORY_ATTACH configured.
- gh-134889 [https://github.com/python/cpython/issues/134889]: Fix handling of a few opcodes that leave operands on the stack when optimizing `LOAD_FAST`.

Library

- gh-134908 [https://github.com/python/cpython/issues/134908]: Fix crash when iterating over lines in a text file on the free threaded build.

Core and Builtins

- gh-132617 [https://github.com/python/cpython/issues/132617]: Fix **`dict.update()`** modification check that could incorrectly raise a "dict mutated during update" error when a different dictionary was modified that happens to share the same underlying keys object.
- gh-134679 [https://github.com/python/cpython/issues/134679]: Fix crash in the free threading build's QSBR code that could occur when changing an object's `__dict__` attribute.
- gh-127682 [https://github.com/python/cpython/issues/127682]: No longer call `__iter__` twice in list comprehensions. This brings the behavior of list comprehensions in line with other forms of iteration
- gh-133912 [https://github.com/python/cpython/issues/133912]: Fix the C API function `PyObject_GenericSetDict` to handle extension classes with inline values.

C API

- gh-134989 [https://github.com/python/cpython/issues/134989]: Fix `Py_RETURN_NONE`, `Py_RETURN_TRUE` and `Py_RETURN_FALSE` macros in the limited C API 3.11 and older: don’t treat `Py_None`, `Py_True` and `Py_False` as immortal. Patch by Victor Stinner.

- gh-134989 [https://github.com/python/cpython/issues/134989]: Implement **`PyObject_DelAttr()`** and **`PyObject_DelAttrString()`** as macros in the limited C API 3.12 and older. Patch by Victor Stinner.
- gh-133968 [https://github.com/python/cpython/issues/133968]: Add **`PyUnicodeWriter_WriteASCII()`** function to write an ASCII string into a **`PyUnicodeWriter`**. The function is faster than **`PyUnicodeWriter_WriteUTF8()`**, but has an undefined behavior if the input string contains non-ASCII characters. Patch by Victor Stinner.

Build

- gh-119132 [https://github.com/python/cpython/issues/119132]: Remove “experimental” tag from the CPython free-threading build.
- gh-135497 [https://github.com/python/cpython/issues/135497]: Fix the detection of `MAXLOGNAME` in the `configure.ac` script.
- gh-134923 [https://github.com/python/cpython/issues/134923]: Windows builds with profile-guided optimization enabled now use `/GENPROFILE` and `/USEPROFILE` instead of deprecated `/LTCG:` options.
- gh-134774 [https://github.com/python/cpython/issues/134774]: Fix **`Py_DEBUG`** macro redefinition warnings on Windows debug builds. Patch by Chris Eibl.
- gh-134632 [https://github.com/python/cpython/issues/134632]: Fixed `build-details.json` generation to use `INCLUDEPY`, in order to reference the `pythonX.Y` subdirectory of the include directory, as required in **PEP 739** [https://peps.python.org/pep-0739/], instead of the top-level include directory.

Python 3.14.0 beta 2

Release date: 2025-05-26

Windows

- gh-130727 [https://github.com/python/cpython/issues/130727]: Fix a race in internal calls into WMI that can result in an “invalid handle” exception under high load. Patch by Chris Eibl.
- gh-76023 [https://github.com/python/cpython/issues/76023]: Make `os.path.realpath()` ignore Windows error 1005 when in non-strict mode.
- gh-133779 [https://github.com/python/cpython/issues/133779]: Reverts the change to generate different `pyconfig.h` files based on compiler settings, as it was frequently causing extension builds to break. In particular, the `Py_GIL_DISABLED` preprocessor variable must now always be defined explicitly when compiling for the experimental free-threaded runtime. The `sysconfig.get_config_var()` function can be used to determine whether the current runtime was compiled with that flag or not.
- gh-133626 [https://github.com/python/cpython/issues/133626]: Ensures packages are not accidentally bundled into the traditional installer.

Tools/Demos

- gh-134215 [https://github.com/python/cpython/issues/134215]: REPL import autocomplete only suggests private modules when explicitly specified.

Tests

- gh-133744 [https://github.com/python/cpython/issues/133744]: Fix multiprocessing interrupt test. Add an event to synchronize the parent process with the child process:

wait until the child process starts sleeping. Patch by Victor Stinner.
- gh-133682 [https://github.com/python/cpython/issues/133682]: Fixed test case `test.test_annotationlib.TestStringFormat.test_displays` which ensures proper handling of complex data structures (lists, sets, dictionaries, and tuples) in string annotations.
- gh-133639 [https://github.com/python/cpython/issues/133639]: Fix `TestPyReplAutoindent.test_auto_indent_default()` doesn't run `input_code`.

Security

- gh-133767 [https://github.com/python/cpython/issues/133767]: Fix use-after-free in the “unicode-escape” decoder with a non-“strict” error handler.
- gh-128840 [https://github.com/python/cpython/issues/128840]: Short-circuit the processing of long IPv6 addresses early in **`ipaddress`** to prevent excessive memory consumption and a minor denial-of-service.

Library

- gh-132710 [https://github.com/python/cpython/issues/132710]: If possible, ensure that **`uuid.getnode()`** returns the same result even across different processes. Previously, the result was constant only within the same process. Patch by Bénédikt Tran.
- gh-80334 [https://github.com/python/cpython/issues/80334]: **`multiprocessing.freeze_support()`** now checks for work on any “spawn” start method platform rather than only on Windows.
- gh-134582 [https://github.com/python/cpython/issues/134582]: Fix tokenize.untokenize() round-trip errors related to t-strings braces escaping

- gh-134546 [https://github.com/python/cpython/issues/134546]: Ensure **`pdb`** remote debugging script is readable by remote Python process.
- gh-134451 [https://github.com/python/cpython/issues/134451]: Converted `asyncio.tools.CycleFoundException` from dataclass to a regular exception type.
- gh-114177 [https://github.com/python/cpython/issues/114177]: Fix **`asyncio`** to not close subprocess pipes which would otherwise error out when the event loop is already closed.
- gh-90871 [https://github.com/python/cpython/issues/90871]: Fixed an off by one error concerning the backlog parameter in **`create_unix_server()`**. Contributed by Christian Harries.
- gh-134323 [https://github.com/python/cpython/issues/134323]: Fix the **`threading.RLock.locked()`** method.
- gh-86802 [https://github.com/python/cpython/issues/86802]: Fixed asyncio memory leak in cancelled shield tasks. For shielded tasks where the shield was cancelled, log potential exceptions through the exception handler. Contributed by Christian Harries.
- gh-134209 [https://github.com/python/cpython/issues/134209]: **`curses`**: The **`curses.window.instr()`** and **`curses.window.getstr()`** methods now allocate their internal buffer on the heap instead of the stack; in addition, the max buffer size is increased from 1023 to 2047.
- gh-134235 [https://github.com/python/cpython/issues/134235]: Updated tab completion on REPL to include builtin modules. Contributed by Tom Wang, Hunter Young
- gh-134152 [https://github.com/python/cpython/issues/134152]: Fixed **`UnboundLocalError`** that could occur during **`email`** header parsing if an expected trailing delimiter is missing in some contexts.
- gh-134168 [https://github.com/python/cpython/issues/134168]: **`http.server`**: Fix IPv6 address binding and **`--directory`** handling when using HTTPS.

- gh-62184 [https://github.com/python/cpython/issues/62184]: Remove import of C implementation of **`io.FileIO`** from Python implementation which has its own implementation
- gh-133982 [https://github.com/python/cpython/issues/133982]: Emit **`RuntimeWarning`** in the Python implementation of **`io`** when the file-like object is not closed explicitly in the presence of multiple I/O layers.
- gh-133890 [https://github.com/python/cpython/issues/133890]: The **`tarfile`** module now handles **`UnicodeEncodeError`** in the same way as **`OSError`** when cannot extract a member.
- gh-134097 [https://github.com/python/cpython/issues/134097]: Fix interaction of the new REPL and **`-X showrefcount`** command line option.
- gh-133889 [https://github.com/python/cpython/issues/133889]: The generated directory listing page in **`http.server.SimpleHTTPRequestHandler`** now only shows the decoded path component of the requested URL, and not the query and fragment.
- gh-134098 [https://github.com/python/cpython/issues/134098]: Fix handling paths that end with a percent-encoded slash (`%2f` or `%2F`) in **`http.server.SimpleHTTPRequestHandler`**.
- gh-132124 [https://github.com/python/cpython/issues/132124]: On POSIX-compliant systems, **`multiprocessing.util.get_temp_dir()`** now ignores **`TMPDIR`** (and similar environment variables) if the path length of `AF_UNIX` socket files exceeds the platform-specific maximum length when using the forkserver start method. Patch by Bénédikt Tran.
- gh-134062 [https://github.com/python/cpython/issues/134062]: **`ipaddress`**: fix collisions in **`__hash__()`** for **`IPv4Network`** and **`IPv6Network`** objects.
- gh-133970 [https://github.com/python/cpython/issues/133970]: Make **`string.templatelib.Template`** and **`string.templatelib.Interpolation`** generic.

- gh-71253 [https://github.com/python/cpython/issues/71253]: Raise **`ValueError`** in **`open()`** if *opener* returns a negative file-descriptor in the Python implementation of **`io`** to match the C implementation.
- gh-133960 [https://github.com/python/cpython/issues/133960]: Simplify and improve **`typing.evaluate_forward_ref()`**. It now no longer raises errors on certain invalid types. In several situations, it is now able to evaluate forward references that were previously unsupported.
- gh-133925 [https://github.com/python/cpython/issues/133925]: Make the private class `typing._UnionGenericAlias` hashable.
- gh-133653 [https://github.com/python/cpython/issues/133653]: Fix **`argparse.ArgumentParser`** with the *formatter_class* argument. Fix TypeError when *formatter_class* is a custom subclass of **`HelpFormatter`**. Fix TypeError when *formatter_class* is not a subclass of **`HelpFormatter`** and non-standard *prefix_char* is used. Fix support of colorizing when *formatter_class* is not a subclass of **`HelpFormatter`**.
- gh-132641 [https://github.com/python/cpython/issues/132641]: Fixed a race in **`functools.lru_cache()`** under free-threading.
- gh-133783 [https://github.com/python/cpython/issues/133783]: Fix bug with applying **`copy.replace()`** to **`ast`** objects. Attributes that default to `None` were incorrectly treated as required for manually created AST nodes.
- gh-133684 [https://github.com/python/cpython/issues/133684]: Fix bug where **`annotationlib.get_annotations()`** would return the wrong result for certain classes that are part of a class hierarchy where `from __future__ import annotations` is used.
- gh-77057 [https://github.com/python/cpython/issues/77057]: Fix handling of invalid markup declarations in **`html.parser.HTMLParser`**.
- gh-130328 [https://github.com/python/cpython/issues/130328]: Speedup pasting in `PyREPL` on Windows in a legacy console. Patch by Chris Eibl.

- gh-133701 [https://github.com/python/cpython/issues/133701]: Fix bug where `typing.TypedDict` classes defined under `from __future__ import annotations` and inheriting from another `TypedDict` had an incorrect `__annotations__` attribute.
- gh-133581 [https://github.com/python/cpython/issues/133581]: Improve unparsing of t-strings in `ast.unparse()` and `from __future__ import annotations`. Empty t-strings now round-trip correctly and formatting in interpolations is preserved. Patch by Jelle Zijlstra.
- gh-133551 [https://github.com/python/cpython/issues/133551]: Support t-strings (**PEP 750** [https://peps.python.org/pep-0750/]) in `annotationlib`. Patch by Jelle Zijlstra.
- gh-133439 [https://github.com/python/cpython/issues/133439]: Fix dot commands with trailing spaces are mistaken for multi-line SQL statements in the sqlite3 command-line interface.
- gh-132493 [https://github.com/python/cpython/issues/132493]: Avoid accessing `__annotations__` unnecessarily in `inspect.signature()`.
- gh-132876 [https://github.com/python/cpython/issues/132876]: `ldexp()` on Windows doesn't round subnormal results before Windows 11, but should. Python's `math.ldexp()` wrapper now does round them, so results may change slightly, in rare cases of very small results, on Windows versions before 11.
- gh-133009 [https://github.com/python/cpython/issues/133009]: `xml.etree.ElementTree`: Fix a crash in `Element.__deepcopy__` when the element is concurrently mutated. Patch by Bénédikt Tran.
- gh-91555 [https://github.com/python/cpython/issues/91555]: Ignore log messages generated during handling of log messages, to avoid deadlock or infinite recursion. [NOTE: This change has since been reverted.]
- gh-125028 [https://github.com/python/cpython/issues/125028]: `functools.Placeholder` cannot be passed to `functools.partial()` as a keyword argument.

- gh-62824 [https://github.com/python/cpython/issues/62824]: Fix aliases for `iso8859_8` encoding. Patch by Dave Goncalves.
- gh-86155 [https://github.com/python/cpython/issues/86155]: **`html.parser.HTMLParser.close()`** no longer loses data when the `<script>` tag is not closed. Patch by Waylan Limberg.
- gh-69426 [https://github.com/python/cpython/issues/69426]: Fix **`html.parser.HTMLParser`** to not unescape character entities in attribute values if they are followed by an ASCII alphanumeric or an equals sign.
- bpo-28494 [https://bugs.python.org/issue?@action=redirect&bpo=28494]: Improve Zip file validation false positive rate in **`zipfile.is_zipfile()`**.
- gh-134381 [https://github.com/python/cpython/issues/134381]: Fix **`RuntimeError`** when using a not-started **`threading.Thread`** after calling **`os.fork()`**

Core and Builtins

- gh-127960 [https://github.com/python/cpython/issues/127960]: PyREPL interactive shell no longer starts with `__package__` and `__file__` global names set to `_pyrepl` package internals. Contributed by Yuichiro Tachibana.
- gh-130397 [https://github.com/python/cpython/issues/130397]: Remove special-casing for C stack depth limits for WASI. Due to WebAssembly’s built-in stack protection this does not pose a security concern.
- gh-128066 [https://github.com/python/cpython/issues/128066]: Fixes an edge case where PyREPL improperly threw an error when Python is invoked on a read only filesystem while trying to write history file entries.
- gh-69605 [https://github.com/python/cpython/issues/69605]: When auto-completing an import in the REPL, finding no candidates now issues no suggestion, rather than suggestions from the current namespace.
- gh-134158 [https://github.com/python/cpython/issues/134158]: Fix coloring of double braces in f-strings and t-strings in the

REPL.

- gh-134119 [https://github.com/python/cpython/issues/134119]: Fix crash when calling `next()` on an exhausted template string iterator. Patch by Jelle Zijlstra.
- gh-134100 [https://github.com/python/cpython/issues/134100]: Fix a use-after-free bug that occurs when an imported module isn’t in `sys.modules` after its initial import. Patch by Nico-Posada.
- gh-133999 [https://github.com/python/cpython/issues/133999]: Fix `SyntaxError` regression in `except` parsing after gh-123440 [https://github.com/python/cpython/issues/123440].
- gh-133886 [https://github.com/python/cpython/issues/133886]: Fix `sys.remote_exec()` for non-ASCII paths in non-UTF-8 locales and non-UTF-8 paths in UTF-8 locales.
- gh-133703 [https://github.com/python/cpython/issues/133703]: Fix hashtable in dict can be bigger than intended in some situations.
- gh-133778 [https://github.com/python/cpython/issues/133778]: Fix bug where assigning to the `__annotations__` attributes of classes defined under `from __future__ import annotations` had no effect.
- gh-132762 [https://github.com/python/cpython/issues/132762]: `fromkeys()` no longer loops forever when adding a small set of keys to a large base dict. Patch by Angela Liss.
- gh-133541 [https://github.com/python/cpython/issues/133541]: Inconsistent indentation in user input crashed the new REPL when syntax highlighting was active. This is now fixed.
- gh-133516 [https://github.com/python/cpython/issues/133516]: Raise `ValueError` when constants `True`, `False` or `None` are used as an identifier after NFKC normalization.
- gh-132542 [https://github.com/python/cpython/issues/132542]: Update `Thread.native_id` after *fork(2)* *[https://manpages.debian.org/fork(2)]* to ensure accuracy. Patch by Noam Cohen.

C API

- gh-134144 [https://github.com/python/cpython/issues/134144]: Fix crash when calling **`Py_EndInterpreter()`** with a thread state that isn't the initial thread for the interpreter.

Build

- gh-134455 [https://github.com/python/cpython/issues/134455]: Fixed `build-details.json` generation to use the correct `c_api.headers` as defined in **PEP 739** [https://peps.python.org/pep-0739/], instead of `c_api.include`.
- gh-131769 [https://github.com/python/cpython/issues/131769]: Fix detecting when the build Python in a cross-build is a pydebug build.
- gh-117088 [https://github.com/python/cpython/issues/117088]: AIX linker don't support -h option, so avoid it through platform check

Python 3.14.0 beta 1

Release date: 2025-05-06

Windows

- gh-132930 [https://github.com/python/cpython/issues/132930]: Marks the installer for Windows as deprecated and updates documentation to cover the new Python install manager.
- gh-127405 [https://github.com/python/cpython/issues/127405]: Add `ABIFLAGS` to **`sysconfig.get_config_vars()`** on Windows. Patch by Xuehai Pan.

Tools/Demos

- gh-130453 [https://github.com/python/cpython/issues/130453]: Allow passing multiple keyword arguments with the same function name in **pygettext**.
- gh-130195 [https://github.com/python/cpython/issues/130195]: Add warning messages when **pygettext** unimplemented `-a/--extract-all` option is called.

Tests

- gh-133131 [https://github.com/python/cpython/issues/133131]: The iOS testbed will now select the most recently released “SE-class” device for testing if a device isn’t explicitly specified.
- gh-91048 [https://github.com/python/cpython/issues/91048]: Add ability to externally inspect all pending asyncio tasks, even if no task is currently entered on the event loop.
- gh-109981 [https://github.com/python/cpython/issues/109981]: The test helper that counts the list of open file descriptors now uses the optimised `/dev/fd` approach on all Apple platforms, not just macOS. This avoids crashes caused by guarded file descriptors.
- gh-132678 [https://github.com/python/cpython/issues/132678]: Add `--prioritize` to `-m test`. This option allows the user to specify which selected tests should execute first, even if the order is otherwise randomized. This is particularly useful for tests that run the longest.
- gh-131290 [https://github.com/python/cpython/issues/131290]: Tests in `Lib/test` can now be correctly executed as standalone scripts.

Security

- gh-115322 [https://github.com/python/cpython/issues/115322]: The underlying extension modules behind **`readline`**:, **`subprocess`**, and **`ctypes`** now raise audit events on previously uncovered code paths that could lead to file system access related to C function calling and external binary execution. The `ctypes.call_function` audit hook has also been fixed to use an unsigned value for its `function pointer`.

Library

- gh-133490 [https://github.com/python/cpython/issues/133490]: Add color support to PDB in remote mode.

- gh-132493 [https://github.com/python/cpython/issues/132493]: Avoid eagerly evaluating annotations in functions decorated with **`reprlib.recursive_repr()`**.

- gh-130645 [https://github.com/python/cpython/issues/130645]: Add color to stdlib argparse CLIs. Patch by Hugo van Kemenade.

- gh-119180 [https://github.com/python/cpython/issues/119180]: Make **`annotationlib.get_annotations()`** succeed with the `FORWARDREF` format if evaluating the annotations throws an exception other than **`NameError`** or **`AttributeError`**.

- gh-133351 [https://github.com/python/cpython/issues/133351]: Fix remote PDB to correctly request tab completions for Python expressions from the server when completing a continuation line of a multi-line Python block.

- gh-133367 [https://github.com/python/cpython/issues/133367]: Add the `--feature-version`, `--optimize`, and `--show-empty` options to the **`ast`** command-line interface. Patch by Semyon Moroz.

- gh-133363 [https://github.com/python/cpython/issues/133363]: The **`cmd.Cmd`** class has been fixed to reliably call the `completedefault` method whenever the `do_shell` method is not defined and tab completion is requested for a line beginning with `!`.

- gh-113081 [https://github.com/python/cpython/issues/113081]: Highlight syntax on source code in **`pdb`**.
- gh-133349 [https://github.com/python/cpython/issues/133349]: Introduced auto-indent in **`pdb`** multi-line input.
- gh-133306 [https://github.com/python/cpython/issues/133306]: Use `\z` instead of `\Z` in **`fnmatch.translate()`** and **`glob.translate()`**.
- gh-133306 [https://github.com/python/cpython/issues/133306]: Support `\z` as a synonym for `\Z` in **`regular expressions`**.
- gh-133300 [https://github.com/python/cpython/issues/133300]: Make **`argparse.ArgumentParser`**’s `suggest_on_error` a keyword-only parameter. Patch by Hugo van Kemenade.
- gh-133290 [https://github.com/python/cpython/issues/133290]: Fix attribute caching issue when setting **`ctypes._Pointer._type_`** in the undocumented and deprecated **`ctypes.SetPointerType()`** function and the undocumented **`set_type()`** method.
- gh-133223 [https://github.com/python/cpython/issues/133223]: When PDB is attached to a remote process, do a better job of intercepting Ctrl+C and forwarding it to the remote process.
- gh-133153 [https://github.com/python/cpython/issues/133153]: Do not complete **`pdb`** commands in `interact` mode of **`pdb`**.
- gh-133139 [https://github.com/python/cpython/issues/133139]: Add the **`curses.assume_default_colors()`** function, a refinement

of the `curses.use_default_colors()` function which allows to change the color pair `0`.

- gh-133089 [https://github.com/python/cpython/issues/133089]: Use original timeout value for `subprocess.TimeoutExpired` when the func `subprocess.run()` is called with a timeout instead of sometimes a confusing partial remaining time out value used internally on the final `wait()`.
- gh-133036 [https://github.com/python/cpython/issues/133036]: `codecs.open()` is now deprecated. Use `open()` instead. Contributed by Inada Naoki.
- gh-132987 [https://github.com/python/cpython/issues/132987]: Many builtin and extension functions which accept an unsigned integer argument, now use `__index__()` if available.
- gh-124703 [https://github.com/python/cpython/issues/124703]: Set return code to `1` when aborting process from `pdb`.
- gh-133005 [https://github.com/python/cpython/issues/133005]: Support passing `preset` option to `tarfile.open()` when using `'w|xz'` mode.
- gh-115032 [https://github.com/python/cpython/issues/115032]: Support for custom logging handlers with the *strm* argument is deprecated and scheduled for removal in Python 3.16. Define handlers with the *stream* argument instead. Patch by Mariusz Felisiak.
- gh-132991 [https://github.com/python/cpython/issues/132991]: Add **`socket.IP_FREEBIND`** constant on Linux 2.4 and later.
- gh-132995 [https://github.com/python/cpython/issues/132995]: Bump the version of pip bundled in ensurepip to version 25.1.1

- gh-132933 [https://github.com/python/cpython/issues/132933]: The zipapp module now applies the filter when creating the list of files to add, rather than waiting until the file is being added to the archive.

- gh-121249 [https://github.com/python/cpython/issues/121249]: Always support the float complex and double complex C types in the **`struct`** module. Patch by Sergey B Kirpichev.

- gh-132915 [https://github.com/python/cpython/issues/132915]: **`fcntl.fcntl()`** and **`fcntl.ioctl()`** can now detect a buffer overflow and raise **`SystemError`**. The stack and memory can be corrupted in such case, so treat this error as fatal.

- gh-132017 [https://github.com/python/cpython/issues/132017]: Fix error when `pyrepl` is suspended, then resumed and terminated.

- gh-132893 [https://github.com/python/cpython/issues/132893]: Improved **`statistics.NormalDist.cdf()`** accuracy for inputs smaller than the mean.

- gh-130328 [https://github.com/python/cpython/issues/130328]: Speedup pasting in `PyREPL` on Windows. Fix by Chris Eibl.

- gh-132882 [https://github.com/python/cpython/issues/132882]: Fix copying of **`typing.Union`** objects containing objects that do not support the `|` operator.

- gh-93696 [https://github.com/python/cpython/issues/93696]: Fixed the breakpoint display error for frozen modules in **`pdb`**.

- gh-129965 [https://github.com/python/cpython/issues/129965]: Add MIME types for `.7z`, `.apk`, `.deb`, `.glb`, `.gltf`, `.gz`, `.m4v`, `.php`, `.rar`, `.rpm`, `.stl` and `.wmv`. Patch by Hugo van Kemenade.

- gh-132742 [https://github.com/python/cpython/issues/132742]: **`fcntl.fcntl()`** now supports arbitrary bytes-like objects, not only **`bytes`**. **`fcntl.ioctl()`** now automatically retries system calls failing with EINTR and releases the GIL during a system call even for large bytes-like object.

- gh-132451 [https://github.com/python/cpython/issues/132451]: The CLI for the PDB debugger now accepts a `-p PID` argument to allow attaching to a running process. The process must be running the same version of Python as the one running PDB.

- gh-125618 [https://github.com/python/cpython/issues/125618]: Add a *format* parameter to **`annotationlib.ForwardRef.evaluate()`**. Evaluating annotations in the `FORWARDREF` format now succeeds in more cases that would previously have raised an exception.

- gh-132805 [https://github.com/python/cpython/issues/132805]: Fix incorrect handling of nested non-constant values in the FORWARDREF format in **`annotationlib`**.

- gh-132734 [https://github.com/python/cpython/issues/132734]: Add new constants for Bluetooth **`sockets`**.

- gh-132673 [https://github.com/python/cpython/issues/132673]: Fix **`AssertionError`** raised on **`ctypes.Structure`** with `_align_ = 0` and `_fields_ = []`.

- gh-132578 [https://github.com/python/cpython/issues/132578]: Rename the `threading.Thread._handle` field to avoid shadowing methods defined on subclasses of `threading.Thread`.

- gh-132561 [https://github.com/python/cpython/issues/132561]: Fix the public `locked` method of `multiprocessing.SemLock` class.

Also adding 2 tests for the derivated `multiprocessing.Lock` and `multiprocessing.RLock` classes.

- gh-121468 [https://github.com/python/cpython/issues/121468]: Add `pdb.set_trace_async()` function to support `await` statements in `pdb`.
- gh-132493 [https://github.com/python/cpython/issues/132493]: `typing.Protocol` now uses `annotationlib.get_annotations()` when checking whether or not an instance implements the protocol with `isinstance()`. This enables support for `isinstance` checks against classes with deferred annotations.
- gh-132536 [https://github.com/python/cpython/issues/132536]: Do not disable `PY_THROW` event in `bdb` because it can’t be disabled.
- gh-132527 [https://github.com/python/cpython/issues/132527]: Include the valid typecode ‘w’ in the error message when an invalid typecode is passed to `array.array`.
- gh-132099 [https://github.com/python/cpython/issues/132099]: The Bluetooth socket with the `BTPROTO_HCI` protocol on Linux now accepts an address in the format of an integer `device_id`, not only a tuple `(device_id,)`.
- gh-81793 [https://github.com/python/cpython/issues/81793]: Fix `os.link()` on platforms (like Linux) where the system **`link()`** function does not follow symlinks. On Linux, it now follows symlinks by default or if `follow_symlinks=True` is specified. On Windows, it now raises an error if `follow_symlinks=True` is passed. On macOS, it now raises an error if `follow_symlinks=False` is passed and the system **`linkat()`** function is not available at runtime.

- gh-132493 [https://github.com/python/cpython/issues/132493]: Support creation of `typing.Protocol` classes with annotations that cannot be resolved at class creation time.

- gh-132491 [https://github.com/python/cpython/issues/132491]: Rename `annotationlib.value_to_string` to `annotationlib.type_repr()` and provide better handling for function objects.

- gh-132426 [https://github.com/python/cpython/issues/132426]: Add `annotationlib.get_annotate_from_class_namespace()` as a helper for accessing annotations in metaclasses, and remove `annotationlib.get_annotate_function`.

- gh-70145 [https://github.com/python/cpython/issues/70145]: Add support for channels in Bluetooth HCI protocol (`BTPROTO_HCI`).

- gh-131913 [https://github.com/python/cpython/issues/131913]: Add a shortcut function `multiprocessing.Process.interrupt()` alongside the existing `multiprocessing.Process.terminate()` and `multiprocessing.Process.kill()` for an improved control over child process termination.

- gh-132439 [https://github.com/python/cpython/issues/132439]: Fix PyREPL on Windows: characters entered via AltGr are swallowed. Patch by Chris Eibl.

- gh-132429 [https://github.com/python/cpython/issues/132429]: Fix support of Bluetooth sockets on NetBSD and DragonFly BSD. Add support for *cid* and *bdaddr_type* in the BTPROTO_L2CAP address on FreeBSD. Return *cid* in `getsockname()` for BTPROTO_L2CAP if it is not zero.

- gh-132106 [https://github.com/python/cpython/issues/132106]: `QueueListener.start` now raises a `RuntimeError` if the listener is already started.

- gh-132417 [https://github.com/python/cpython/issues/132417]: Fix a `NULL` pointer dereference when a C function called using **`ctypes`** with `restype` **`py_object`** returns `NULL`.

- gh-132385 [https://github.com/python/cpython/issues/132385]: Fix instance error suggestions trigger potential exceptions in **`object.__getattr__()`** in **`traceback`**.

- gh-125866 [https://github.com/python/cpython/issues/125866]: Add optional *add_scheme* argument to **`urllib.request.pathname2url()`**; when set to true, a complete URL is returned. Likewise add optional *require_scheme* argument to **`url2pathname()`**; when set to true, a complete URL is accepted.

- gh-132308 [https://github.com/python/cpython/issues/132308]: A **`traceback.TracebackException`** now correctly renders the `__context__` and `__cause__` attributes from falsey **`Exception`**, and the `exceptions` attribute from falsey **`ExceptionGroup`**.

- gh-130645 [https://github.com/python/cpython/issues/130645]: Add colour to **`argparse`** help output. Patch by Hugo van Kemenade.

- gh-127495 [https://github.com/python/cpython/issues/127495]: In PyREPL, append a new entry to the `PYTHON_HISTORY` file *after* every statement. This should preserve command-line history after interpreter is terminated. Patch by Sergey B Kirpichev.

- gh-129463 [https://github.com/python/cpython/issues/129463]: Comparison of **`annotationlib.ForwardRef`** objects no longer uses the internal `__code__` and `__ast_node__` attributes, which are used as caches.

- gh-132250 [https://github.com/python/cpython/issues/132250]: Fixed the `SystemError` in `cProfile` when locating the actual C function of a method raises an exception.

- gh-132064 [https://github.com/python/cpython/issues/132064]: `annotationlib.get_annotations()` now uses the `__annotate__` attribute if it is present, even if `__annotations__` is not present. Additionally, the function now raises a `TypeError` if it is passed an object that does not have any annotatins.

- gh-130664 [https://github.com/python/cpython/issues/130664]: Support the `'_'` digit separator in formatting of the integral part of `Decimal`’s. Patch by Sergey B Kirpichev.

- gh-131952 [https://github.com/python/cpython/issues/131952]: Add color output to the **json** CLI. Patch by Tomas Roun.

- gh-132063 [https://github.com/python/cpython/issues/132063]: Prevent exceptions that evaluate as falsey (namely, when their `__bool__` method returns `False` or their `__len__` method returns 0) from being ignored by `concurrent.futures.ProcessPoolExecutor` and `concurrent.futures.ThreadPoolExecutor`.

- gh-132106 [https://github.com/python/cpython/issues/132106]: `logging.handlers.QueueListener` now implements the context manager protocol, allowing it to be used in a `with` statement.

- gh-132054 [https://github.com/python/cpython/issues/132054]: The `application/yaml` mime type (**RFC 9512** [https://datatracker.ietf.org/doc/html/rfc9512.html]) is now supported by `mimetypes`. Patch by Sasha “Nelie” Chernykh and Hugo van Kemenade.

- gh-119605 [https://github.com/python/cpython/issues/119605]: Respect `follow_wrapped` for `__init__()` and `__new__()` methods when getting the class signature for a class with `inspect.signature()`. Preserve class signature after wrapping with `warnings.deprecated()`. Patch by Xuehai Pan.

- gh-118761 [https://github.com/python/cpython/issues/118761]: Improve import times by up to 33x for the `shlex` module, and improve the performance of `shlex.quote()` by up to 12x. Patch by Adam Turner.

- gh-85302 [https://github.com/python/cpython/issues/85302]: Add support for `BTPROTO_SCO` in sockets on FreeBSD.

- gh-131757 [https://github.com/python/cpython/issues/131757]: Make `functools.lru_cache()` call the cached function unlocked to allow concurrency.

- gh-131423 [https://github.com/python/cpython/issues/131423]: `ssl` can show descriptions for errors added in OpenSSL 3.4.1. Patch by Bénédikt Tran.

- gh-131434 [https://github.com/python/cpython/issues/131434]: Improve error reporting for incorrect format in `time.strptime()`.

- gh-131524 [https://github.com/python/cpython/issues/131524]: Add help message to `platform` command-line interface. Contributed by Harry Lees.

- gh-100926 [https://github.com/python/cpython/issues/100926]: Move `ctypes.POINTER()` types cache from a global internal cache (`_pointer_type_cache`) to the `ctypes._CData.__pointer_type__` attribute of the corresponding `ctypes` types. This will stop the cache from growing without limits in some situations.

- gh-85702 [https://github.com/python/cpython/issues/85702]: If `zoneinfo._common.load_tzdata` is given a package without a resource a `ZoneInfoNotFoundError` is raised rather than a **`IsADirectoryError`**.

- gh-123471 [https://github.com/python/cpython/issues/123471]: Make concurrent iterations over **`itertools.repeat`** safe under free-threading.

- gh-131127 [https://github.com/python/cpython/issues/131127]: Systems using LibreSSL now successfully build.

- gh-89157 [https://github.com/python/cpython/issues/89157]: Make the pure Python implementation of **`datetime.date.fromisoformat()`**, only accept ASCII strings for consistency with the C implementation.

- gh-130941 [https://github.com/python/cpython/issues/130941]: Fix **`configparser.ConfigParser`** parsing empty interpolation with `allow_no_value` set to `True`.

- gh-110067 [https://github.com/python/cpython/issues/110067]: Make **`heapq`** max-heap functions **`heapq.heapify_max()`**, **`heapq.heappush_max()`**, **`heapq.heappop_max()`**, and **`heapq.heapreplace_max()`** public. Previous underscored naming is kept for backwards compatibility. Additionally, the missing function **`heapq.heappushpop_max()`** has been added to both the C and Python implementations.

- gh-129098 [https://github.com/python/cpython/issues/129098]: Fix REPL traceback reporting when using **`compile()`** with an inexisting file. Patch by Bénédikt Tran.

- gh-130631 [https://github.com/python/cpython/issues/130631]: **`http.cookiejar.join_header_words()`** is now more similar to the original Perl version. It now quotes the same set of characters and always quote values that end with `"\n"`.

- gh-130482 [https://github.com/python/cpython/issues/130482]: Add ability to specify name for **`tkinter.OptionMenu`** and **`tkinter.ttk.OptionMenu`**.

- gh-77065 [https://github.com/python/cpython/issues/77065]: Add keyword-only optional argument *echo_char* for `getpass.getpass()` for optional visual keyboard feedback support. Patch by Semyon Moroz.

- gh-130317 [https://github.com/python/cpython/issues/130317]: Fix `PyFloat_Pack2()` and `PyFloat_Unpack2()` for NaN’s with payload. This corrects round-trip for `struct.unpack()` and `struct.pack()` in case of the IEEE 754 binary16 “half precision” type. Patch by Sergey B Kirpichev.

- gh-130402 [https://github.com/python/cpython/issues/130402]: Joining running daemon threads during interpreter shutdown now raises `PythonFinalizationError`.

- gh-130167 [https://github.com/python/cpython/issues/130167]: Improve speed of `difflib.IS_LINE_JUNK()`. Patch by Semyon Moroz.

- gh-101410 [https://github.com/python/cpython/issues/101410]: Added more detailed messages for domain errors in the `math` module.

- gh-128384 [https://github.com/python/cpython/issues/128384]: Make `warnings.catch_warnings` use a context variable for holding the warning filtering state if the `sys.flags.context_aware_warnings` flag is set to true. This makes using the context manager thread-safe in multi-threaded programs. The flag is true by default in free-threaded builds and is otherwise false. The value of the flag can be overridden by the the `-X context_aware_warnings` command-line option or by the `PYTHON_CONTEXT_AWARE_WARNINGS` environment variable.

- gh-129719 [https://github.com/python/cpython/issues/129719]: Fix missing **`socket.CAN_RAW_ERR_FILTER`** constant in the socket module on Linux systems. It was missing since Python 3.11.

- gh-129027 [https://github.com/python/cpython/issues/129027]: Raise **`DeprecationWarning`** for **`sys._clear_type_cache()`**. This function was deprecated in Python 3.13 but it didn’t raise a runtime warning.

- gh-128307 [https://github.com/python/cpython/issues/128307]: Add `eager_start` keyword argument to **`asyncio.loop.create_task()`**

- gh-127604 [https://github.com/python/cpython/issues/127604]: Add support for printing the C stack trace on systems that support it via **`faulthandler.dump_c_stack()`** or via the *c_stack* argument in **`faulthandler.enable()`**.

- gh-127385 [https://github.com/python/cpython/issues/127385]: Add the `F_DUPFD_QUERY` constant to the **`fcntl`** module.

- gh-126838 [https://github.com/python/cpython/issues/126838]: Add *resolve_host* keyword-only parameter to **`urllib.request.url2pathname()`**, and fix handling of file URLs with authorities.

- gh-82129 [https://github.com/python/cpython/issues/82129]: Fix **`NameError`** when calling **`typing.get_type_hints()`** on a **`dataclasses.dataclass()`** created by **`dataclasses.make_dataclass()`** with un-annotated fields.

- gh-122559 [https://github.com/python/cpython/issues/122559]: Remove **`__reduce__()`** and **`__reduce_ex__()`** methods that always raise **`TypeError`** in the C implementation of **`io.FileIO`**, **`io.BufferedReader`**, **`io.BufferedWriter`** and **`io.BufferedRandom`** and replace them with default `__getstate__()` methods that raise `TypeError`. This restores fine details of behavior of Python 3.11 and older versions.

- gh-122179 [https://github.com/python/cpython/issues/122179]: `hashlib.file_digest()` now raises `BlockingIOError` when no data is available during non-blocking I/O. Before, it added spurious null bytes to the digest.

- gh-53032 [https://github.com/python/cpython/issues/53032]: Expose `decimal.IEEEContext()` to support creation of contexts corresponding to the IEEE 754 (2008) decimal interchange formats. Patch by Sergey B Kirpichev.

- gh-120220 [https://github.com/python/cpython/issues/120220]: Deprecate the `tkinter.Variable` methods `trace_variable()`, `trace_vdelete()` and `trace_vinfo()`. Methods `trace_add()`, `trace_remove()` and `trace_info()` can be used instead.

- gh-113539 [https://github.com/python/cpython/issues/113539]: `webbrowser`: Names in the `BROWSER` environment variable can now refer to already registered web browsers, instead of always generating a new browser command.

  This makes it possible to set `BROWSER` to the value of one of the supported browsers on macOS.

- bpo-44172 [https://bugs.python.org/issue?@action=redirect&bpo=44172]: Keep a reference to original `curses` windows in subwindows so that the original window does not get deleted before subwindows.

- gh-75223 [https://github.com/python/cpython/issues/75223]: Deprecate undotted extensions in `mimetypes.MimeTypes.add_type()`. Patch by Hugo van Kemenade.

IDLE

- gh-112936 [https://github.com/python/cpython/issues/112936]: fix IDLE: no Shell menu item in single-process mode.

Documentation

- gh-107006 [https://github.com/python/cpython/issues/107006]: Move documentation and example code for **`threading.local`** from its docstring to the official docs.
- gh-125142 [https://github.com/python/cpython/issues/125142]: As part of the builtin help intro text, show the keyboard shortcuts for the new, non-basic REPL (F1, F2, and F3).

Core and Builtins

- gh-133336 [https://github.com/python/cpython/issues/133336]: **`-J`** is no longer reserved for use by Jython. Patch by Adam Turner.
- gh-133261 [https://github.com/python/cpython/issues/133261]: Fix bug where the cycle GC could untrack objects in the trashcan because they looked like they were immortal. When objects are added to the trashcan, we take care to ensure they keep a mortal reference count.
- gh-133346 [https://github.com/python/cpython/issues/133346]: Added experimental color theming support to the `_colorize` module.
- gh-132917 [https://github.com/python/cpython/issues/132917]: For the free-threaded build, check the process memory usage increase before triggering a full automatic garbage collection. If the memory used has not increased 10% since the last collection then defer it.
- gh-91048 [https://github.com/python/cpython/issues/91048]: Add a new `python -m asyncio ps PID` command-line interface to inspect asyncio tasks in a running Python process.

Displays a flat table of await relationships. A variant showing a tree view is also available as `python -m asyncio pstree PID`. Both are useful for debugging async code. Patch by Pablo Galindo, Łukasz Langa, Yury Selivanov, and Marta Gomez Macias.

- gh-133304 [https://github.com/python/cpython/issues/133304]: Workaround NaN's "canonicalization" in **`PyFloat_Pack4()`** and **`PyFloat_Unpack4()`** on RISC-V.
- gh-133197 [https://github.com/python/cpython/issues/133197]: Improve **`SyntaxError`** error messages for incompatible string / bytes prefixes.
- gh-133231 [https://github.com/python/cpython/issues/133231]: Add new utilities of observing JIT compilation: **`sys._jit.is_available()`**, **`sys._jit.is_enabled()`**, and **`sys._jit.is_active()`**.

Library

- gh-133194 [https://github.com/python/cpython/issues/133194]: **`ast.parse()`** will no longer parse new **PEP 758** [https://peps.python.org/pep-0758/] syntax with older *feature_version* passed.

Core and Builtins

- gh-131798 [https://github.com/python/cpython/issues/131798]: Split `CALL_LEN` into several uops allowing the JIT to remove them when optimizing. Patch by Diego Russo.
- gh-131798 [https://github.com/python/cpython/issues/131798]: Use `sym_new_type` instead of `sym_new_not_null` for _BUILD_STRING, _BUILD_SET
- gh-132942 [https://github.com/python/cpython/issues/132942]: Fix two races in the type lookup cache. This affected the free-threaded build and could cause crashes (apparently quite difficult to trigger).

- gh-131798 [https://github.com/python/cpython/issues/131798]: Propagate the return type of `_BINARY_OP_SUBSCR_TUPLE_INT` in JIT. Patch by Tomas Roun
- gh-132952 [https://github.com/python/cpython/issues/132952]: Speed up startup with the `-S` argument by importing the private `_io` module instead of **`io`**. This fixes a performance regression introduced earlier in Python 3.14 development and restores performance to the level of Python 3.13.
- gh-131798 [https://github.com/python/cpython/issues/131798]: Allow the JIT to remove int guards after `_CALL_LEN` by setting the return type to int. Patch by Diego Russo
- gh-131798 [https://github.com/python/cpython/issues/131798]: Split `CALL_TUPLE_1` into several uops allowing the JIT to remove some of them. Patch by Tomas Roun
- gh-131798 [https://github.com/python/cpython/issues/131798]: Split `CALL_STR_1` into several uops allowing the JIT to remove some of them. Patch by Tomas Roun
- gh-132825 [https://github.com/python/cpython/issues/132825]: Enhance unhashable key/element error messages for **`dict`** and **`set`**. Patch by Victor Stinner.
- gh-131591 [https://github.com/python/cpython/issues/131591]: Reset any **PEP 768** [https://peps.python.org/pep-0768/] remote debugging pending call in children after **`os.fork()`** calls.
- gh-132713 [https://github.com/python/cpython/issues/132713]: Fix `repr(list)` race condition: hold a strong reference to the item while calling `repr(item)`. Patch by Victor Stinner.
- gh-132661 [https://github.com/python/cpython/issues/132661]: Implement **PEP 750** [https://peps.python.org/pep-0750/] (Template Strings). Add new syntax for t-strings and implement new internal **`string.templatelib.Template`** and **`string.templatelib.Interpolation`** types.
- gh-132479 [https://github.com/python/cpython/issues/132479]: Fix compiler crash in certain circumstances where multiple module-level annotations include comprehensions and other nested scopes.

- gh-132747 [https://github.com/python/cpython/issues/132747]: Fix a crash when calling `__get__()` of a method with a `None` second argument.
- gh-132744 [https://github.com/python/cpython/issues/132744]: Certain calls now check for runaway recursion and respect the system recursion limit.
- gh-132449 [https://github.com/python/cpython/issues/132449]: Syntax errors that look like misspellings of Python keywords now provide a helpful fix suggestion for the typo. Contributed by Pablo Galindo Salgado.

Library

- gh-132737 [https://github.com/python/cpython/issues/132737]: Support profiling code that requires `__main__`, such as `pickle`.

Core and Builtins

- gh-132639 [https://github.com/python/cpython/issues/132639]: Added `PyLong_AsNativeBytes()`, `PyLong_FromNativeBytes()` and `PyLong_FromUnsignedNativeBytes()` to the limited C API.
- gh-100239 [https://github.com/python/cpython/issues/100239]: Add specialisation for `BINARY_OP/SUBSCR` on list and slice.
- gh-132508 [https://github.com/python/cpython/issues/132508]: Uses tagged integers on the evaluation stack to represent the instruction offsets when reraising an exception. This avoids the need to box the integer which could fail in low memory conditions.
- gh-124476 [https://github.com/python/cpython/issues/124476]: Fix decoding from the locale encoding in the C.UTF-8 locale.
- gh-131927 [https://github.com/python/cpython/issues/131927]: Compiler warnings originating from the same module and line number are now only emitted once, matching the behaviour of warnings emitted from user code. This can also be configured with `warnings` filters.

- gh-132457 [https://github.com/python/cpython/issues/132457]: Make **`staticmethod()`** and **`classmethod()`** generic.
- gh-131798 [https://github.com/python/cpython/issues/131798]: Use `sym_new_type` instead of `sym_new_not_null` for _BUILD_LIST, _BUILD_SET, _BUILD_MAP
- gh-131798 [https://github.com/python/cpython/issues/131798]: Split `CALL_TYPE_1` into several uops allowing the JIT to remove some of them.
- gh-132386 [https://github.com/python/cpython/issues/132386]: Fix crash when passing a dict subclass as the `globals` parameter to **`exec()`**.
- gh-127682 [https://github.com/python/cpython/issues/127682]: No longer call `__iter__` twice when creating and executing a generator expression. Creating a generator expression from a non-interable will raise only when the generator expression is executed. This brings the behavior of generator expressions in line with other generators.
- gh-132261 [https://github.com/python/cpython/issues/132261]: The internal storage for annotations and annotate functions on classes now uses different keys in the class dictionary. This eliminates various edge cases where access to the `__annotate__` and `__annotations__` attributes would behave unpredictably.
- gh-132284 [https://github.com/python/cpython/issues/132284]: Don’t wrap base `PyCFunction` slots on class creation if not overridden.
- gh-130415 [https://github.com/python/cpython/issues/130415]: Improve the JIT’s ability to remove unused constant and local variable loads, and fix an issue where deallocating unused values could cause JIT code to crash or behave incorrectly.
- gh-126703 [https://github.com/python/cpython/issues/126703]: Fix possible use after free in cases where a method’s definition has the same lifetime as its `self`.
- gh-132286 [https://github.com/python/cpython/issues/132286]: Fix that **`type.__annotate__`** was not deleted, when

`type.__annotations__` was deleted.

- gh-131798 [https://github.com/python/cpython/issues/131798]: Allow the JIT to remove an extra `_TO_BOOL_BOOL` instruction after `_CONTAINS_OP_DICT` by setting the return type to bool.
- gh-124715 [https://github.com/python/cpython/issues/124715]: Prevents against stack overflows when calling **`Py_DECREF()`**. Third-party extension objects no longer need to use the “trashcan” mechanism, as protection is now built into the **`Py_DECREF()`** macro.
- gh-131798 [https://github.com/python/cpython/issues/131798]: Allow the JIT compiler to remove some type checks for operations on lists, tuples, dictionaries, and sets.
- gh-128398 [https://github.com/python/cpython/issues/128398]: Improve error message when an object supporting the synchronous (resp. asynchronous) context manager protocol is entered using **`async with`** (resp. **`with`**) instead of **`with`** (resp. **`async with`**). Patch by Bénédikt Tran.
- gh-131798 [https://github.com/python/cpython/issues/131798]: Allow the JIT to remove unicode guards after `_BINARY_OP_SUBSCR_STR_INT` by setting the return type to string.
- gh-131878 [https://github.com/python/cpython/issues/131878]: Handle uncaught exceptions in the main input loop for the new REPL.
- gh-131878 [https://github.com/python/cpython/issues/131878]: Fix support of unicode characters with two or more codepoints on Windows in the new REPL.
- gh-126835 [https://github.com/python/cpython/issues/126835]: Move constant folding to the peephole optimizer. Rename AST optimization related files (Python/ast_opt.c -> Python/ast_preprocess.c), structs (_PyASTOptimizeState -> _PyASTPreprocessState) and functions (_PyAST_Optimize -> _PyAST_Preprocess, _PyCompile_AstOptimize -> _PyCompile_AstPreprocess).
- gh-114809 [https://github.com/python/cpython/issues/114809]: Add support for macOS multi-arch builds with the JIT enabled

- gh-131507 [https://github.com/python/cpython/issues/131507]: PyREPL now supports syntax highlighting. Contributed by Łukasz Langa.
- gh-130907 [https://github.com/python/cpython/issues/130907]: If the `__annotations__` of a module object are accessed while the module is executing, return the annotations that have been defined so far, without caching them.
- gh-130104 [https://github.com/python/cpython/issues/130104]: Three-argument `pow()` now try calling `__rpow__()` if necessary. Previously it was only called in two-argument **`pow()`** and the binary power operator.
- gh-130070 [https://github.com/python/cpython/issues/130070]: Fixed an assertion error for `exec()` passed a string `source` and a non-`None` `closure`. Patch by Bartosz Sławecki.
- gh-129958 [https://github.com/python/cpython/issues/129958]: Fix a bug that was allowing newlines inconsistently in format specifiers for single-quoted f-strings. Patch by Pablo Galindo.
- gh-129858 [https://github.com/python/cpython/issues/129858]: `elif` statements that follow an `else` block now have a specific error message.
- gh-69605 [https://github.com/python/cpython/issues/69605]: Add module autocomplete to PyREPL.
- gh-128555 [https://github.com/python/cpython/issues/128555]: Add the `sys.flags.thread_inherit_context` flag.
    - This flag is set to true by default on the free-threaded build and false otherwise. If the flag is true, starting a new thread using `threading.Thread` will, by default, use a copy of the `contextvars.Context` from the caller of `threading.Thread.start()` rather than using an empty context.
    - Add the `-X thread_inherit_context` command-line option and `PYTHON_THREAD_INHERIT_CONTEXT` environment variable, which set the `thread_inherit_context` flag.
    - Add the `context` keyword parameter to `Thread`. It can be used to explicitly pass a context value to be used

  by a new thread.
  - Make the `_contextvars` module built-in.
- gh-123539 [https://github.com/python/cpython/issues/123539]: Improve **`SyntaxError`** message for using `import ... as` and `from ... import ... as` with not a name.
- gh-102567 [https://github.com/python/cpython/issues/102567]: **`-X importtime`** now accepts value `2`, which indicates that an `importtime` entry should also be printed if an imported module has already been loaded. Patch by Noah Kim and Adam Turner.
- gh-116436 [https://github.com/python/cpython/issues/116436]: Improve error message when **`TypeError`** occurs during **`dict.update()`**
- gh-103997 [https://github.com/python/cpython/issues/103997]: String arguments passed to "-c" are now automatically dedented as if by **`textwrap.dedent()`**. This allows "python -c" invocations to be indented in shell scripts without causing indentation errors. (Patch by Jon Crall and Steven Sun)

Library

- gh-89562 [https://github.com/python/cpython/issues/89562]: Remove `hostflags` member from `PySSLContext` struct.

C API

- gh-133166 [https://github.com/python/cpython/issues/133166]: Fix regression where **`PyType_GetModuleByDef()`** returns NULL without setting **`TypeError`** when a static type is passed.
- gh-133164 [https://github.com/python/cpython/issues/133164]: Add **`PyUnstable_Object_IsUniqueReferencedTemporary()`** function for determining if an object exists as a unique temporary variable on the interpreter's stack. This is a replacement for some cases where checking that **`Py_REFCNT()`** is one is no longer sufficient to determine if

it’s safe to modify a Python object in-place with no visible side effects.

- gh-133140 [https://github.com/python/cpython/issues/133140]: Add **`PyUnstable_Object_IsUniquelyReferenced()`** as a replacement for `Py_REFNCT(op) == 1` on free threaded builds of Python.
- gh-131747 [https://github.com/python/cpython/issues/131747]: On non-Windows platforms, deprecate using **`ctypes.Structure._pack_`** to use a Windows-compatible layout on non-Windows platforms. The layout should be specified explicitly by setting **`ctypes.Structure._layout_`** to `'ms'`.
- gh-128972 [https://github.com/python/cpython/issues/128972]: For non-free-threaded builds, the memory layout of **`PyASCIIObject`** is reverted to match Python 3.13. (Note that the structure is not part of stable ABI and so its memory layout is *guaranteed* to remain stable.)
- gh-133079 [https://github.com/python/cpython/issues/133079]: The undocumented APIs **`Py_C_RECURSION_LIMIT`** and **`PyThreadState.c_recursion_remaining`**, added in 3.13, are removed without a deprecation period.
- gh-132987 [https://github.com/python/cpython/issues/132987]: The `k` and `K` formats in **`PyArg_Parse()`** now support the **`__index__()`** special method, like all other integer formats.
- gh-132909 [https://github.com/python/cpython/issues/132909]: Fix an overflow when handling the K format in **`Py_BuildValue()`**. Patch by Bénédikt Tran.
- gh-132798 [https://github.com/python/cpython/issues/132798]: Deprecated and undocumented functions **`PyUnicode_AsEncodedObject()`**, **`PyUnicode_AsDecodedObject()`**, **`PyUnicode_AsEncodedUnicode()`** and **`PyUnicode_AsDecodedUnicode()`** are scheduled for removal in 3.15.
- gh-132470 [https://github.com/python/cpython/issues/132470]: Creating a **`ctypes.CField`** with a *byte_size* that does not match the actual type size now raises a **`ValueError`** instead of crashing the interpreter.

- gh-112068 [https://github.com/python/cpython/issues/112068]: [Reverted in gh-136991 [https://github.com/python/cpython/issues/136991]] Add support of nullable arguments in **`PyArg_Parse()`** and similar functions. Adding `?` after any format unit makes `None` be accepted as a value.
- gh-50333 [https://github.com/python/cpython/issues/50333]: Non-tuple sequences are deprecated as argument for the `(items)` format unit in **`PyArg_ParseTuple()`** and other argument parsing functions if *items* contains format units which store a borrowed buffer or a borrowed reference.

Build

- gh-113464 [https://github.com/python/cpython/issues/113464]: Use the cpython-bin-deps “externals” repository for Windows LLVM dependency management. Installing LLVM manually is no longer necessary for Windows JIT builds.
- gh-133183 [https://github.com/python/cpython/issues/133183]: iOS compiler shims now include `IPHONEOS_DEPLOYMENT_TARGET` in target triples, ensuring that SDK version minimums are honored.
- gh-133167 [https://github.com/python/cpython/issues/133167]: Fix compilation process with `--enable-optimizations` and `--without-docstrings`.
- gh-133171 [https://github.com/python/cpython/issues/133171]: Since free-threaded builds do not support the experimental JIT compiler, prevent these configurations from being combined.
- gh-132758 [https://github.com/python/cpython/issues/132758]: Fix building with tail call interpreter and pystats.
- gh-132649 [https://github.com/python/cpython/issues/132649]: The `PClayout` script now allows passing `--include-tcltk` on Windows ARM64.

- gh-132257 [https://github.com/python/cpython/issues/132257]: Change the default LTO flags on GCC to not pass `-flto-partition=none`, and allow parallelization of LTO. For newer GNU makes and GCC, this has a multiple factor speedup for LTO build times, with no noticeable loss in performance.
- gh-132026 [https://github.com/python/cpython/issues/132026]: Fix use of undefined identifiers in platform triplet detection on MIPS Linux platforms.

Python 3.14.0 alpha 7

Release date: 2025-04-08

macOS

- gh-124111 [https://github.com/python/cpython/issues/124111]: Update macOS installer to use Tcl/Tk 8.6.16.
- gh-131423 [https://github.com/python/cpython/issues/131423]: Update macOS installer to use OpenSSL 3.0.16. Patch by Bénédikt Tran.
- gh-131025 [https://github.com/python/cpython/issues/131025]: Update macOS installer to ship with SQLite 3.49.1.

Windows

- gh-131423 [https://github.com/python/cpython/issues/131423]: Update bundled version of OpenSSL to 3.0.16. The new build also disables uplink support, which may be relevant to embedders but has no impact on normal use.
- gh-131453 [https://github.com/python/cpython/issues/131453]: Some **`SND_*`** and **`MB_*`** constants are added to **`winsound`**.
- gh-91349 [https://github.com/python/cpython/issues/91349]: Replaces our copy of `zlib` with `zlib-ng`, for performance

improvements in `zlib`.
- gh-131025 [https://github.com/python/cpython/issues/131025]: Update Windows installer to ship with SQLite 3.49.1.

Tools/Demos

- gh-132121 [https://github.com/python/cpython/issues/132121]: Always escape non-printable Unicode characters in **pygettext**.
- gh-131852 [https://github.com/python/cpython/issues/131852]: **msgfmt** no longer adds the `POT-Creation-Date` to generated `.mo` files for consistency with GNU `msgfmt`.

Tests

- gh-131277 [https://github.com/python/cpython/issues/131277]: Allow to unset one or more environment variables at once via `EnvironmentVarGuard.unset()`. Patch by Bénédikt Tran.
- gh-131050 [https://github.com/python/cpython/issues/131050]: `test_ssl.test_dh_params` is skipped if the underlying TLS library does not support finite-field ephemeral Diffie-Hellman.

Security

- gh-131809 [https://github.com/python/cpython/issues/131809]: Update bundled libexpat to 2.7.1
- gh-131261 [https://github.com/python/cpython/issues/131261]: Upgrade to libexpat 2.7.0
- gh-121284 [https://github.com/python/cpython/issues/121284]: Fix bug in the folding of rfc2047 encoded-words when flattening an email message using a modern email policy. Previously when an encoded-word was too long for a line, it would be decoded, split across lines, and re-encoded. But commas and other special characters in the original text could be left unencoded and unquoted.

This could theoretically be used to spoof header lines using a carefully constructed encoded-word if the resulting rendered email was transmitted or re-parsed.

Library

- gh-132174 [https://github.com/python/cpython/issues/132174]: Fix function name in error message of `_interpreters.run_string`.
- gh-132171 [https://github.com/python/cpython/issues/132171]: Fix crash of `_interpreters.run_string` on string subclasses.
- gh-129204 [https://github.com/python/cpython/issues/129204]: Introduce new `_PYTHON_SUBPROCESS_USE_POSIX_SPAWN` environment variable knob in **`subprocess`** to control the use of **`os.posix_spawn()`**.
- gh-132159 [https://github.com/python/cpython/issues/132159]: Do not shadow user arguments in generated **`__new__()`** by decorator **`warnings.deprecated`**. Patch by Xuehai Pan.
- gh-132168 [https://github.com/python/cpython/issues/132168]: The **`ctypes.py_object`** type now supports subscription, making it a generic type.
- gh-84481 [https://github.com/python/cpython/issues/84481]: Add the **`zipfile.ZipFile.data_offset`** attribute, which stores the offset to the beginning of ZIP data in a file when available. When the **`zipfile.ZipFile`** is opened in either mode `'w'` or `'x'` and the underlying file does not support `tell()`, the value will be `None` instead.
- gh-132075 [https://github.com/python/cpython/issues/132075]: Fix possible use of **`socket`** address structures with uninitialized members. Now all structure members are initialized with zeroes by default.
- gh-118761 [https://github.com/python/cpython/issues/118761]: Improve import times by up to 27x for the **`string`** module. Patch by Adam Turner.
- gh-125434 [https://github.com/python/cpython/issues/125434]: Display thread name in **`faulthandler`**. Patch by Victor

Stinner.

- gh-132002 [https://github.com/python/cpython/issues/132002]: Fix crash when deallocating `contextvars.ContextVar` with weird unahashable string names.
- gh-131938 [https://github.com/python/cpython/issues/131938]: `xml.etree.ElementTree`: update the error message when an element to remove via `Element.remove` is not found. Patch by Bénédikt Tran.
- gh-115942 [https://github.com/python/cpython/issues/115942]: Add `threading.RLock.locked()`, `multiprocessing.Lock.locked()`, `multiprocessing.RLock.locked()`, and allow `multiprocessing.managers.SyncManager.Lock()` and `multiprocessing.managers.SyncManager.RLock()` to proxy `locked()` call.
- gh-131974 [https://github.com/python/cpython/issues/131974]: Fix several thread-safety issues in `ctypes` on the free threaded build.
- gh-118761 [https://github.com/python/cpython/issues/118761]: Improve the import time of the `ast` module by extracting the `unparse()` function to a helper module.
- gh-107369 [https://github.com/python/cpython/issues/107369]: Improved performance of `textwrap.indent()` by an average of ~1.3x. Patch by Adam Turner.
- gh-131792 [https://github.com/python/cpython/issues/131792]: Improved performance of `textwrap.dedent()` by an average of ~2.4x, (with improvements of up to 4x for large inputs), and fixed a bug where blank lines with whitespace characters other than space or horizontal tab were not normalised to the newline. Patch by Adam Turner, Marius Juston, and Pieter Eendebak.
- gh-131668 [https://github.com/python/cpython/issues/131668]: `socket`: Fix code parsing AF_BLUETOOTH socket addresses.
- gh-60115 [https://github.com/python/cpython/issues/60115]: Support frozen modules for `linecache.getline()`.

- gh-131492 [https://github.com/python/cpython/issues/131492]: Fix a resource leak when constructing a **`gzip.GzipFile`** with a filename fails, for example when passing an invalid `compresslevel`.
- gh-131435 [https://github.com/python/cpython/issues/131435]: 10-20% performance improvement of **`random.randint()`**.
- gh-131461 [https://github.com/python/cpython/issues/131461]: Fix **`ResourceWarning`** when constructing a **`gzip.GzipFile`** in write mode with a broken file object.
- gh-125866 [https://github.com/python/cpython/issues/125866]: Deprecate the **`nturl2path`** module. Call **`urllib.request.url2pathname()`** and **`pathname2url()`** instead.
- gh-126367 [https://github.com/python/cpython/issues/126367]: Fix issue where **`urllib.request.url2pathname()`** raised **`OSError`** when given a Windows URI containing a colon character not following a drive letter, such as before an NTFS alternate data stream.
- gh-120144 [https://github.com/python/cpython/issues/120144]: Disable `CALL` event in **`bdb`** in `monitoring` backend when we don't need any new events on the code object to get a better performance.
- gh-131358 [https://github.com/python/cpython/issues/131358]: Register `cseuckr` as an encoding alias for `euc_kr`.
- gh-131325 [https://github.com/python/cpython/issues/131325]: Fix sendfile fallback implementation to drain data after writing to transport in **`asyncio`**.
- gh-90548 [https://github.com/python/cpython/issues/90548]: **`platform.libc_ver()`** can now detect and report the version of `musl` on Alpine Linux.
- gh-129843 [https://github.com/python/cpython/issues/129843]: Fix incorrect argument passing in **`warnings.warn_explicit()`**.
- gh-70647 [https://github.com/python/cpython/issues/70647]: When creating a **`datetime`** object with an out of range date a more informative error is raised.
- gh-130914 [https://github.com/python/cpython/issues/130914]: Allow **`graphlib.TopologicalSorter.prepare()`** to be called more

than once as long as sorting has not started. Patch by Daniel Pope.

- gh-131236 [https://github.com/python/cpython/issues/131236]: Allow to generate multiple UUIDs at once via **`python -m uuid --count`**.
- gh-126895 [https://github.com/python/cpython/issues/126895]: Fix **`readline`** in free-threaded build.
- gh-121468 [https://github.com/python/cpython/issues/121468]: `$_asynctask` is added as a **`pdb`** convenience variable to access the current asyncio task if applicable.
- gh-118761 [https://github.com/python/cpython/issues/118761]: Improve import time of **`locale`** using lazy import `re`. Patch by Semyon Moroz.
- gh-129598 [https://github.com/python/cpython/issues/129598]: Fix **`ast.unparse()`** when **`ast.Interactive`** contains multiple statements.
- gh-85162 [https://github.com/python/cpython/issues/85162]: The **`http.server`** module now includes built-in support for HTTPS servers exposed by **`http.server.HTTPSServer`**. This functionality is exposed by the command-line interface (`python -m http.server`) through the `--tls-cert`, `--tls-key` and `--tls-password-file` options. Patch by Semyon Moroz.
- gh-129463 [https://github.com/python/cpython/issues/129463]: The implementations of equality and hashing for **`annotationlib.ForwardRef`** now use all attributes on the object. Two **`ForwardRef`** objects are equal only if all attributes are equal.
- gh-128593 [https://github.com/python/cpython/issues/128593]: **`annotationlib.ForwardRef`** objects no longer cache their value when they are successfully evaluated. Successive calls to **`annotationlib.ForwardRef.evaluate()`** may return different values.
- gh-117779 [https://github.com/python/cpython/issues/117779]: Fix reading duplicated entries in **`zipfile`** by name. Reading duplicated entries (except the last one) by `ZipInfo` now emits a warning instead of raising an exception.

- gh-128715 [https://github.com/python/cpython/issues/128715]: The class of **`Structure`**/**`Union`** field descriptors is now available as **`CField`**, and has new attributes to aid debugging and introspection.
- gh-128055 [https://github.com/python/cpython/issues/128055]: Fix `test.test_sysconfig.test_sysconfigdata_json` when running outside the build directory (eg. after installing).
- gh-126037 [https://github.com/python/cpython/issues/126037]: **`xml.etree.ElementTree`**: Fix a crash in **`Element.find`**, **`Element.findtext`** and **`Element.findall`** when the tag to find implements an **`__eq__()`** method mutating the element being queried. Patch by Bénédikt Tran.
- gh-127794 [https://github.com/python/cpython/issues/127794]: When headers are added to **`email.message.Message`** objects, either through **`email.message.Message.__setitem__()`** or **`email.message.Message.add_header()`**, the field name is now validated according to **RFC 5322, Section 2.2** [https://datatracker.ietf.org/doc/html/rfc5322.html#section-2.2] and a **`ValueError`** is raised if the field name contains any invalid characters.
- gh-123599 [https://github.com/python/cpython/issues/123599]: Deprecate **`pathlib.PurePath.as_uri()`**; use **`pathlib.Path.as_uri()`** instead.
- gh-126033 [https://github.com/python/cpython/issues/126033]: **`xml.etree.ElementTree`**: Fix a crash in **`Element.remove`** when the element is concurrently mutated. Patch by Bénédikt Tran.
- gh-120144 [https://github.com/python/cpython/issues/120144]: Add the optional backend of `sys.monitoring` to **`bdb`** and use it for **`pdb`**.
- gh-74598 [https://github.com/python/cpython/issues/74598]: Add **`fnmatch.filterfalse()`** for excluding names matching a pattern. Patch by Bénédikt Tran.
- gh-114917 [https://github.com/python/cpython/issues/114917]: Add support for AI_NUMERICSERV in getaddrinfo emulation

- bpo-17254 [https://bugs.python.org/issue?@action=redirect&bpo=17254]: Added aliases for Thai Language using Microsoft Code Pages.

Documentation

- gh-131417 [https://github.com/python/cpython/issues/131417]: Mention **`asyncio.Future`** and **`asyncio.Task`** in generic classes list.

Core and Builtins

- gh-131798 [https://github.com/python/cpython/issues/131798]: Allow the JIT to remove an extra `_TO_BOOL_BOOL` instruction after `_CONTAINS_OP_SET` by setting the return type to bool.
- gh-132011 [https://github.com/python/cpython/issues/132011]: Fix crash when calling **`list.append()`** as an unbound method.
- gh-131998 [https://github.com/python/cpython/issues/131998]: Fix a crash when using an unbound method descriptor object in a function where a bound method descriptor was used.
- gh-131591 [https://github.com/python/cpython/issues/131591]: Implement **PEP 768** [https://peps.python.org/pep-0768/] (Safe external debugger interface for CPython). Add a new **`sys.remote_exec()`** function to the **`sys`** module. This function schedules the execution of a Python file in a separate process. Patch by Pablo Galindo, Matt Wozniski and Ivona Stojanovic.
- gh-131798 [https://github.com/python/cpython/issues/131798]: Allow JIT to omit str guard in truthiness test when str type is known.
- gh-131833 [https://github.com/python/cpython/issues/131833]: Add support for optionally dropping grouping parentheses when using multiple exception types as per **PEP 758** [https://peps.python.org/pep-0758/]. Patch by Pablo Galindo

- gh-130924 [https://github.com/python/cpython/issues/130924]: Usage of a name in a function-scope annotation no longer triggers creation of a cell for that variable. This fixes a regression in earlier alphas of Python 3.14.
- gh-131800 [https://github.com/python/cpython/issues/131800]: Improve the experimental JIT’s ability to remove type checks for certain subscripting operations.
- gh-131738 [https://github.com/python/cpython/issues/131738]: Compiler emits optimized code for builtin any/all/tuple calls over a generator expression.
- gh-131719 [https://github.com/python/cpython/issues/131719]: Fix missing NULL check in `_PyMem_FreeDelayed` in free-threaded build.
- gh-131670 [https://github.com/python/cpython/issues/131670]: Fix **`anext()`** failing on sync **`__anext__()`** raising an exception.
- gh-131666 [https://github.com/python/cpython/issues/131666]: Fix signature of `anext_awaitable.close` objects. Patch by Bénédikt Tran.
- gh-130415 [https://github.com/python/cpython/issues/130415]: Optimize comparison of two constants in JIT builds
- gh-129149 [https://github.com/python/cpython/issues/129149]: Add fast path for small and medium-size integers in **`PyLong_FromInt32()`**, **`PyLong_FromUInt32()`**, **`PyLong_FromInt64()`** and **`PyLong_FromUInt64()`**. Patch by Chris Eibl.
- gh-130887 [https://github.com/python/cpython/issues/130887]: Optimize the AArch64 code generation for the JIT. Patch by Diego Russo
- gh-130956 [https://github.com/python/cpython/issues/130956]: Optimize the AArch64 code generation for the JIT. Patch by Diego Russo
- gh-130928 [https://github.com/python/cpython/issues/130928]: Fix error message when formatting bytes using the `'i'` flag. Patch by Maxim Ageev.
- gh-130935 [https://github.com/python/cpython/issues/130935]: Annotations at the class and module level that are conditionally defined are now only reflected in

`__annotations__` if the block they are in is executed. Patch by Jelle Zijlstra.

- gh-130775 [https://github.com/python/cpython/issues/130775]: Do not crash on negative `column` and `end_column` in **`ast`** locations.
- gh-130704 [https://github.com/python/cpython/issues/130704]: Optimize `LOAD_FAST` and its superinstruction form to reduce reference counting overhead. These instructions are replaced with faster variants that load borrowed references onto the operand stack when we can prove that the reference in the frame outlives the reference loaded onto the stack.
- gh-88887 [https://github.com/python/cpython/issues/88887]: Fixing multiprocessing Resource Tracker process leaking, usually observed when running Python as PID 1.
- gh-130115 [https://github.com/python/cpython/issues/130115]: Fix an issue with thread identifiers being sign-extended on some platforms.

Library

- gh-99108 [https://github.com/python/cpython/issues/99108]: Add support for built-in implementation of HMAC (**RFC 2104** [https://datatracker.ietf.org/doc/html/rfc2104.html]) based on HACL*. Patch by Bénédikt Tran.

Core and Builtins

- gh-130080 [https://github.com/python/cpython/issues/130080]: Implement PEP 765: Disallow return/break/continue that exit a finally block.
- gh-129900 [https://github.com/python/cpython/issues/129900]: Fix return codes inside **`SystemExit`** not getting returned by the REPL.
- gh-128632 [https://github.com/python/cpython/issues/128632]: Disallow `__classdict__` as the name of a type parameter.

Using this name would previously crash the interpreter in some circumstances.

- gh-126703 [https://github.com/python/cpython/issues/126703]: Improve performance of builtin methods by using a freelist.
- gh-126703 [https://github.com/python/cpython/issues/126703]: Improve performance of `range` by using a freelist.

C API

- gh-131740 [https://github.com/python/cpython/issues/131740]: Update PyUnstable_GC_VisitObjects to traverse perm gen.
- gh-131525 [https://github.com/python/cpython/issues/131525]: The PyTupleObject now caches the computed hash value in the new field ob_hash.

Build

- gh-131865 [https://github.com/python/cpython/issues/131865]: The DTrace build now properly passes the `CC` and `CFLAGS` variables to the `dtrace` command when utilizing SystemTap on Linux.
- gh-131675 [https://github.com/python/cpython/issues/131675]: Fix mimalloc library builds for 32-bit ARM targets.
- gh-131691 [https://github.com/python/cpython/issues/131691]: clang-cl on Windows needs option `/EHa` to support SEH (structured exception handling) correctly. Fix by Chris Eibl.
- gh-131278 [https://github.com/python/cpython/issues/131278]: Add optimizing flag `WITH_COMPUTED_GOTOS` to Windows builds for when using a compiler that supports it (currently clang-cl). Patch by Chris Eibl.
- gh-130213 [https://github.com/python/cpython/issues/130213]: Update the vendored HACL* library to fix build issues with older clang compilers.

- gh-130673 [https://github.com/python/cpython/issues/130673]: Fix potential `KeyError` when handling object sections during JIT building process.

Python 3.14.0 alpha 6

Release date: 2025-03-14

macOS

- gh-128540 [https://github.com/python/cpython/issues/128540]: Ensure web browser is launched by `webbrowser.open()` on macOS, even for `file://` URLs.

Windows

- gh-131020 [https://github.com/python/cpython/issues/131020]: pylauncher [https://github.com/python/cpython/tree/3.14/PC/launcher2.c] correctly detects a BOM when searching for the shebang. Fix by Chris Eibl.

Tools/Demos

- gh-130453 [https://github.com/python/cpython/issues/130453]: Make it possible to override default keywords in **pygettext**.
- gh-85012 [https://github.com/python/cpython/issues/85012]: Correctly reset `msgctxt` when compiling messages in **msgfmt**.
- gh-130453 [https://github.com/python/cpython/issues/130453]: Extend support for specifying custom keywords in **pygettext**.
- gh-130195 [https://github.com/python/cpython/issues/130195]: Add warning messages when `pygettext` unimplemented `-a/--extract-all` option is called.

- gh-130057 [https://github.com/python/cpython/issues/130057]: Add support for translator comments in **pygettext.py**.
- gh-130025 [https://github.com/python/cpython/issues/130025]: The iOS testbed now correctly handles symlinks used as Python framework references.
- gh-129911 [https://github.com/python/cpython/issues/129911]: Fix the keyword entry in the help output of **pygettext**.

Tests

- gh-129200 [https://github.com/python/cpython/issues/129200]: Multiple iOS testbed runners can now be started at the same time without introducing an ambiguity over simulator ownership.
- gh-130292 [https://github.com/python/cpython/issues/130292]: The iOS testbed will now run successfully on a machine that has not previously run Xcode tests (such as CI configurations).
- gh-130293 [https://github.com/python/cpython/issues/130293]: The tests of terminal colorization are no longer sensitive to the value of the `TERM` variable in the testing environment.
- gh-129401 [https://github.com/python/cpython/issues/129401]: Fix a flaky test in `test_repr_rlock` that checks the representation of **`multiprocessing.RLock`**.
- gh-126332 [https://github.com/python/cpython/issues/126332]: Add unit tests for pyrepl.

Security

- gh-127371 [https://github.com/python/cpython/issues/127371]: Avoid unbounded buffering for **`tempfile.SpooledTemporaryFile.writelines()`**. Previously, disk spillover was only checked after the lines iterator had been exhausted. This is now done after each line is written.

Library

- gh-131204 [https://github.com/python/cpython/issues/131204]: Use monospace font from System Font Stack for cross-platform support in **`difflib.HtmlDiff`**.
- gh-131196 [https://github.com/python/cpython/issues/131196]: Improve performance of **`uuid.UUID.hex`** and **`uuid.UUID.__str__`**.
- gh-130940 [https://github.com/python/cpython/issues/130940]: The behavior of `PyConfig.use_system_logger` was modified to be enabled by default on iOS. It remains disabled by default on macOS.
- gh-131123 [https://github.com/python/cpython/issues/131123]: Supported completions for attributes of convenience variables in **`pdb`**.
- gh-93096 [https://github.com/python/cpython/issues/93096]: Removed undocumented CLI `python -m difflib`. Use `python -m doctest Lib/difflib.py -v` instead. Patch by Semyon Moroz.
- gh-93096 [https://github.com/python/cpython/issues/93096]: Removed undocumented `-t` and `-v` arguments of `python -m pickle`. Use `python -m doctest Lib/pickle.py -v` instead. Patch by Semyon Moroz.
- gh-81267 [https://github.com/python/cpython/issues/81267]: Correct **`time.sleep()`** error message when an object that cannot be interpreted as an integer or float is provided.
- gh-93096 [https://github.com/python/cpython/issues/93096]: Removed undocumented `-t` and `-v` arguments of `python -m pickletools`. Use `python -m doctest Lib/pickletools.py -v` instead. Patch by Semyon Moroz.
- gh-131045 [https://github.com/python/cpython/issues/131045]: Fix issue with `__contains__`, values, and pseudo-members for **`enum.Flag`**.
- gh-130959 [https://github.com/python/cpython/issues/130959]: Fix pure-Python implementation of

`datetime.time.fromisoformat()` to reject times with spaces in fractional part (for example, `12:34:56.400 +02:00`), matching the C implementation. Patch by Michał Gorny.

- gh-130806 [https://github.com/python/cpython/issues/130806]: Deleting **`gzip.GzipFile`** before it is closed now emits a **`ResourceWarning`**.
- gh-130637 [https://github.com/python/cpython/issues/130637]: Add validation for numeric response data in poplib.POP3.stat() method
- gh-130665 [https://github.com/python/cpython/issues/130665]: Only apply locale to calendar CLI when set via `--locale` and not via `LANG` environment variable.
- gh-130660 [https://github.com/python/cpython/issues/130660]: `sys.ps1` and `sys.ps2` are now restored after **`code.interact()`** call.
- gh-130608 [https://github.com/python/cpython/issues/130608]: Remove *dirs_exist_ok* argument from **`pathlib.Path.copy()`** and **`copy_into()`**. These methods are new in Python 3.14.
- gh-130461 [https://github.com/python/cpython/issues/130461]: Remove `.. index::` directives from the **`uuid`** module documentation. These directives previously created entries in the general index for **`getnode()`** as well as the **`uuid1()`**, **`uuid3()`**, **`uuid4()`**, **`uuid5()`**, and **`uuid8()`** constructor functions.
- gh-130379 [https://github.com/python/cpython/issues/130379]: The zipapp module now calculates the list of files to be added to the archive before creating the archive. This avoids accidentally including the target when it is being created in the source directory.
- gh-82987 [https://github.com/python/cpython/issues/82987]: Inline breakpoints like **`breakpoint()`** or **`pdb.set_trace()`** will always stop the program at calling frame, ignoring the `skip` pattern (if any).
- gh-125377 [https://github.com/python/cpython/issues/125377]: `<tab>` at the beginning of the line in **`pdb`** multi-line input will fill

in a 4-space indentation now, instead of inserting a `\t` character.

- gh-125413 [https://github.com/python/cpython/issues/125413]: Ensure the path returned from **`pathlib.Path.copy()`** or **`move()`** has fresh **`info`**.
- gh-65697 [https://github.com/python/cpython/issues/65697]: stdlib configparser will now attempt to validate that keys it writes will not result in file corruption (creating a file unable to be accurately parsed by a future read() call from the same parser). Attempting a corrupting write() will raise an InvalidWriteError.
- gh-125413 [https://github.com/python/cpython/issues/125413]: Speed up **`Path.copy`** by making better use of **`info`** internally.
- gh-130285 [https://github.com/python/cpython/issues/130285]: Fix corner case for **`random.sample()`** allowing the *counts* parameter to specify an empty population. So now, `sample([], 0, counts=[])` and `sample('abc', k=0, counts=[0, 0, 0])` both give the same result as `sample([], 0)`.
- gh-124703 [https://github.com/python/cpython/issues/124703]: Executing `quit` command in **`pdb`** will raise **`bdb.BdbQuit`** when **`pdb`** is started from an interactive console using **`breakpoint()`** or **`pdb.set_trace()`**.
- gh-107773 [https://github.com/python/cpython/issues/107773]: Make **`datetime`** subclass **`__repr__()`** consistent both implementations. Patch by Semyon Moroz.
- gh-130250 [https://github.com/python/cpython/issues/130250]: Fix regression in `traceback.print_last()`.
- gh-123471 [https://github.com/python/cpython/issues/123471]: Make concurrent iterations over **`itertools.batched`** safe under free-threading.
- gh-57537 [https://github.com/python/cpython/issues/57537]: Support breakpoints for **`zipimport`** modules on **`pdb`**
- gh-130230 [https://github.com/python/cpython/issues/130230]: Fix crash in **`pow()`** with only **`Decimal`** third argument.

- gh-126944 [https://github.com/python/cpython/issues/126944]: Show explicit errors when required arguments of **`pdb`** commands are missing
- gh-127750 [https://github.com/python/cpython/issues/127750]: Improve repr of **`functools.singledispatchmethod`** methods and descriptors.
- gh-128520 [https://github.com/python/cpython/issues/128520]: Apply type conversion consistently in **`pathlib.PurePath`** and **`Path`** methods can accept a path object as an argument, such as **`match()`** and **`rename()`**. The argument is now converted to path object if it lacks a **`with_segments()`** attribute, and not otherwise.
- gh-118761 [https://github.com/python/cpython/issues/118761]: Reverts a change in the previous release attempting to make some stdlib imports used within the **`subprocess`** module lazy as this was causing errors during `__del__` finalizers calling methods such as `terminate`, or `kill`, or `send_signal`.
- gh-130164 [https://github.com/python/cpython/issues/130164]: Fixed failure to raise **`TypeError`** in **`inspect.Signature.bind()`** for positional-only arguments provided by keyword when a variadic keyword argument (e.g. `**kwargs`) is present.
- gh-130151 [https://github.com/python/cpython/issues/130151]: Fix reference leaks in **`_hashlib.hmac_new()`** and **`_hashlib.hmac_digest()`**. Patch by Bénédikt Tran.
- gh-130145 [https://github.com/python/cpython/issues/130145]: Fix **`asyncio.AbstractEventloop.run_forever()`** when another loop is already running.
- gh-130139 [https://github.com/python/cpython/issues/130139]: Fix bug where **`ast.parse()`** did not error on AST input which is not of the correct type, when called with optimize=False.
- gh-127260 [https://github.com/python/cpython/issues/127260]: Forbid the use of colon (“:”) as a fractional component separator and other improvements to the consistency of error raising between the C and Python implementations

of **`datetime.time.fromisoformat()`** and **`datetime.datetime.fromisoformat()`**. Patch by Semyon Moroz.

- gh-85795 [https://github.com/python/cpython/issues/85795]: Using **`super()`** and `__class__` closure variable in user-defined methods of **`typing.NamedTuple`** subclasses is now explicitly prohibited at runtime. Contributed by Bartosz Sławecki in gh-130082 [https://github.com/python/cpython/issues/130082].
- gh-118761 [https://github.com/python/cpython/issues/118761]: Improve import time of **`cmd`** by lazy importing **`inspect`** and removing **`string`**. Patch by Semyon Moroz.
- gh-129726 [https://github.com/python/cpython/issues/129726]: Fix **`gzip.GzipFile`** raising an unraisable exception during garbage collection when referring to a temporary object by breaking the reference loop with **`weakref`**.
- gh-127750 [https://github.com/python/cpython/issues/127750]: Remove broken **`functools.singledispatchmethod()`** caching introduced in gh-85160 [https://github.com/python/cpython/issues/85160]. Achieve the same performance using different optimization.
- gh-129948 [https://github.com/python/cpython/issues/129948]: Add support for shared **`set`** to **`multiprocessing.managers.SyncManager`** via **`SyncManager.set()`**.
- gh-129965 [https://github.com/python/cpython/issues/129965]: Update MIME types for `.avi` and `.wav`. Add MIME types for `.docx`, `.pptx`, `.xlsx`, `.epub`, `.flac`, `.m4a`, `.odg`, `.odp`, `.ods`, `.odt`, `.oga`, `.ogg`, `.ogx` and `.weba`. Patch by Hugo van Kemenade.
- gh-129889 [https://github.com/python/cpython/issues/129889]: Support context manager protocol by **`contextvars.Token`**. Patch by Andrew Svetlov.
- gh-97850 [https://github.com/python/cpython/issues/97850]: Update the deprecation warning of **`importlib.abc.Loader.load_module()`**.
- gh-129678 [https://github.com/python/cpython/issues/129678]: **`configparser.ConfigParser`**: do not write an empty unnamed section

- gh-128641 [https://github.com/python/cpython/issues/128641]: Restore `configparser.ConfigParser.read()` performance.
- gh-129569 [https://github.com/python/cpython/issues/129569]: Fix `unicodedata.normalize()` to always return a built-in `str` object when given an input of a `str` subclass, regardless of whether the string is already normalized.
- gh-128231 [https://github.com/python/cpython/issues/128231]: Execution of multiple statements in the new REPL now stops immediately upon the first exception encountered. Patch by Bartosz Sławecki.
- gh-96092 [https://github.com/python/cpython/issues/96092]: Fix bug in `traceback.walk_stack()` called with None where it was skipping more frames than in prior versions. This bug fix also changes walk_stack to walk the stack in the frame where it was called rather than where it first gets used.
- gh-129288 [https://github.com/python/cpython/issues/129288]: Add optional `l2_cid` and `l2_bdaddr_type` fields to `socket` `BTPROTO_L2CAP` sockaddr tuple.
- gh-128703 [https://github.com/python/cpython/issues/128703]: Fix `mimetypes.guess_type()` to use default mapping for empty `Content-Type` in registry.
- gh-128647 [https://github.com/python/cpython/issues/128647]: Eagerly write to buffers passed to `gzip.GzipFile`’s `readinto()` and `readinto1()` implementations, avoiding unnecessary allocations. Patch by Chris Markiewicz.
- gh-128184 [https://github.com/python/cpython/issues/128184]: Improve display of `annotationlib.ForwardRef` object within `inspect.Signature` representations. This also fixes a `NameError` that was raised when using `dataclasses.dataclass()` on classes with unresolvable forward references.
- gh-128041 [https://github.com/python/cpython/issues/128041]: Add `concurrent.futures.ProcessPoolExecutor.terminate_workers()` and `concurrent.futures.ProcessPoolExecutor.kill_workers()` as ways to terminate or kill all living worker processes in

the given pool. (Contributed by Charles Machalow in gh-130849 [https://github.com/python/cpython/issues/130849].)

- gh-127647 [https://github.com/python/cpython/issues/127647]: Add protocols **`io.Reader`** and **`io.Writer`** as alternatives to **`typing.IO`**, **`typing.TextIO`**, and **`typing.BinaryIO`**.
- gh-109798 [https://github.com/python/cpython/issues/109798]: Added additional information into error messages in **`datetime`**, and made the messages more consistent between the C and Python implementations. Patch by Semyon Moroz.
- gh-125746 [https://github.com/python/cpython/issues/125746]: Delay deprecated **`zipimport.zipimporter.load_module()`** removal time to 3.15. Use **`zipimport.zipimporter.exec_module()`** instead.
- gh-74028 [https://github.com/python/cpython/issues/74028]: Add the optional `buffersize` parameter to **`concurrent.futures.Executor.map()`** to limit the number of submitted tasks whose results have not yet been yielded. If the buffer is full, iteration over the *iterables* pauses until a result is yielded from the buffer.
- gh-124927 [https://github.com/python/cpython/issues/124927]: Non-printing characters are now properly handled in the new REPL.
- gh-124096 [https://github.com/python/cpython/issues/124096]: Turn on virtual terminal mode and enable bracketed paste in REPL on Windows console. (If the terminal does not support bracketed paste, enabling it does nothing.)
- gh-89083 [https://github.com/python/cpython/issues/89083]: Add **`uuid.uuid7()`** for generating UUIDv7 objects as specified in **RFC 9562** [https://datatracker.ietf.org/doc/html/rfc9562.html]. Patch by Bénédikt Tran.
- gh-89083 [https://github.com/python/cpython/issues/89083]: Add **`uuid.uuid6()`** for generating UUIDv6 objects as specified in **RFC 9562** [https://datatracker.ietf.org/doc/html/rfc9562.html]. Patch by Bénédikt Tran.

- gh-117151 [https://github.com/python/cpython/issues/117151]: Increase `io.DEFAULT_BUFFER_SIZE` from 8k to 128k and adjust `open()` on platforms where `os.fstat()` provides a `st_blksize` field (such as Linux) to use `max(min(blocksize, 8 MiB), io.DEFAULT_BUFFER_SIZE)` rather than always using the device block size. This should improve I/O performance. Patch by Romain Morotti.
- gh-105499 [https://github.com/python/cpython/issues/105499]: Make `types.UnionType` an alias for `typing.Union`. Both `int | str` and `Union[int, str]` now create instances of the same type. Patch by Jelle Zijlstra.
- gh-93096 [https://github.com/python/cpython/issues/93096]: Document the command-line for `mimetypes`. It now exits with `1` on failure instead of `0` and `2` on incorrect command-line parameters instead of `1`. Also, errors are printed to stderr instead of stdout and their text is made tighter. Patch by Oleg Iarygin and Hugo van Kemenade.

Documentation

- gh-125722 [https://github.com/python/cpython/issues/125722]: Require Sphinx 8.2.0 or later to build the Python documentation. Patch by Adam Turner.
- gh-129712 [https://github.com/python/cpython/issues/129712]: The wheel tags supported by each macOS universal SDK option are now documented.
- gh-46236 [https://github.com/python/cpython/issues/46236]: C API: Document `PyUnicode_RSplit()`, `PyUnicode_Partition()` and `PyUnicode_RPartition()`.

Core and Builtins

- gh-131141 [https://github.com/python/cpython/issues/131141]: Fix data race in `sys.monitoring` instrumentation while registering callback.

- gh-130804 [https://github.com/python/cpython/issues/130804]: Fix support of unicode characters on Windows in the new REPL.
- gh-130932 [https://github.com/python/cpython/issues/130932]: Fix incorrect exception handling in `_PyModule_IsPossiblyShadowing`
- gh-122029 [https://github.com/python/cpython/issues/122029]: **`sys.setprofile()`** and **`sys.settrace()`** will not generate a `c_call` event for `INSTRUMENTED_CALL_FUNCTION_EX` if the callable is a method with a C function wrapped, because we do not generate `c_return` event in such case.
- gh-129964 [https://github.com/python/cpython/issues/129964]: Fix JIT crash on Windows on Arm. Patch by Diego Russo and Brandt Bucher.
- gh-130851 [https://github.com/python/cpython/issues/130851]: Fix a crash in the free threading build when constructing a **`code`** object with **`co_consts`** that contains instances of types that are not otherwise generated by the bytecode compiler.
- gh-128534 [https://github.com/python/cpython/issues/128534]: Ensure that both left and right branches have the same source for `async for` loops. Add these branches to the `co_branches()` iterator.
- gh-130794 [https://github.com/python/cpython/issues/130794]: Fix memory leak in the free threaded build when resizing a shared list or dictionary from multiple short-lived threads.
- gh-130415 [https://github.com/python/cpython/issues/130415]: Improve JIT understanding of integers in boolean context.
- gh-130382 [https://github.com/python/cpython/issues/130382]: Fix `PyRefTracer_DESTROY` not being sent from `Python/ceval.c` `Py_DECREF()`.
- gh-130574 [https://github.com/python/cpython/issues/130574]: Renumber **`RESUME`** from 149 to 128.

- gh-124878 [https://github.com/python/cpython/issues/124878]: Fix race conditions during runtime finalization that could lead to accessing freed memory.
- gh-130415 [https://github.com/python/cpython/issues/130415]: Improve the experimental JIT's ability to narrow boolean values based on the results of truthiness tests.
- gh-130618 [https://github.com/python/cpython/issues/130618]: Fix a bug that was causing `UnicodeDecodeError` or `SystemError` to be raised when using f-strings with `lambda` expressions with non-ASCII characters. Patch by Pablo Galindo
- gh-123044 [https://github.com/python/cpython/issues/123044]: Make sure that the location of branch targets in `match` cases is in the body, not the pattern.
- gh-128534 [https://github.com/python/cpython/issues/128534]: Add branch monitoring (`BRANCH_LEFT` and `BRANCH_RIGHT` events) for `async for` loops.
- gh-130163 [https://github.com/python/cpython/issues/130163]: Fix possible crashes related to concurrent change and use of the **`sys`** module attributes.
- gh-122029 [https://github.com/python/cpython/issues/122029]: `INSTRUMENTED_CALL_KW` will expand the method before monitoring to reflect the actual behavior more accurately.
- gh-130415 [https://github.com/python/cpython/issues/130415]: Improve JIT's ability to optimize strings in boolean contexts.
- gh-130396 [https://github.com/python/cpython/issues/130396]: Use actual stack limits (from *pthread_getattr_np(3)* *[https://manpages.debian.org/pthread_getattr_np(3)]*) for linux, and other systems with `_GNU_SOURCE` defined, when determining limits for C stack protection.
- gh-128396 [https://github.com/python/cpython/issues/128396]: Fix a crash that occurs when calling **`locals()`** inside an inline comprehension that uses the same local variable as the outer frame scope where the variable is a free or cell var.

- gh-129107 [https://github.com/python/cpython/issues/129107]: Fix two more **`bytearray`** functions for free threading.
- gh-127705 [https://github.com/python/cpython/issues/127705]: Use tagged references (`_PyStackRef`) for the default build as well as for the free-threading build. This has a small negative performance impact short-term but will enable larger speedups in the future and significantly reduce maintenance costs by allowing a single implementation of tagged references in the future.
- gh-130094 [https://github.com/python/cpython/issues/130094]: Fix two race conditions involving concurrent imports that could lead to spurious failures with **`ModuleNotFoundError`**.
- gh-129107 [https://github.com/python/cpython/issues/129107]: Make **`bytearray`** iterator safe under free threading.
- gh-115802 [https://github.com/python/cpython/issues/115802]: Use the more efficient "medium" code model for JIT-compiled code on supported platforms.
- gh-107956 [https://github.com/python/cpython/issues/107956]: A `build-details.json` file is now install in the platform-independent standard library directory (**PEP 739** [https://peps.python.org/pep-0739/] implementation).
- gh-116042 [https://github.com/python/cpython/issues/116042]: Fix location for SyntaxErrors of invalid escapes in the tokenizer. Patch by Pablo Galindo
- gh-91079 [https://github.com/python/cpython/issues/91079]: Change C stack overflow protection to consider the amount of stack consumed, rather than a counter. This allows deeper recursion in many cases, but remains safe.
- gh-129715 [https://github.com/python/cpython/issues/129715]: Improve the experimental JIT's handling of returns to unknown callers.

Library

- gh-129983 [https://github.com/python/cpython/issues/129983]: Fix data race in compile_template in `sre.c`.

Core and Builtins

- gh-129967 [https://github.com/python/cpython/issues/129967]: Fix a race condition in the free threading build when `repr(set)` is called concurrently with `set.clear()`.
- gh-129953 [https://github.com/python/cpython/issues/129953]: The internal (evaluation) stack is now spilled to memory whenever execution escapes from the interpreter or JIT compiled code. This should have no observable effect in either Python or builtin extensions, but will allow various important optimizations in the future.
- gh-129515 [https://github.com/python/cpython/issues/129515]: Clarify syntax error messages for conditional expressions when a statement is specified before an **`if`** or after an **`else`** keyword.
- gh-129349 [https://github.com/python/cpython/issues/129349]: **`bytes.fromhex()`** and **`bytearray.fromhex()`** now accepts ASCII **`bytes`** and bytes-like objects.
- gh-129149 [https://github.com/python/cpython/issues/129149]: Add fast path for medium-size integers in **`PyLong_FromSsize_t()`**. Patch by Chris Eibl.
- gh-129107 [https://github.com/python/cpython/issues/129107]: Make the **`bytearray`** safe under free threading.
- gh-128974 [https://github.com/python/cpython/issues/128974]: Fix a crash in **`UnicodeError.__str__`** when custom attributes implement **`__str__()`** with side-effects. Patch by Bénédikt Tran.
- gh-126085 [https://github.com/python/cpython/issues/126085]: **`typing.TypeAliasType`** now supports star unpacking.
- gh-125331 [https://github.com/python/cpython/issues/125331]: `from __future__ import barry_as_FLUFL` now works in more contexts, including when it is used in files, with the `-c` flag, and in the REPL when there are multiple statements on the same line. Previously, it worked only on subsequent lines in the REPL, and when the

appropriate flags were passed directly to **`compile()`**. Patch by Pablo Galindo.

- gh-121464 [https://github.com/python/cpython/issues/121464]: Make concurrent iterations over the same **`enumerate()`** iterator safe under free-threading. See Strategy for Iterators in Free Threading [https://github.com/python/cpython/issues/124397].
- gh-87790 [https://github.com/python/cpython/issues/87790]: Support underscore and comma as thousands separators in the fractional part for floating-point presentation types of the new-style string formatting (with **`format()`** or f-strings). Patch by Sergey B Kirpichev.
- gh-124445 [https://github.com/python/cpython/issues/124445]: Fix specialization of generic aliases that are generic over a **`typing.ParamSpec`** and have been specialized with a nested type variable.
- gh-120608 [https://github.com/python/cpython/issues/120608]: Adapt **`reversed()`** for use in the free-threading build. The **`reversed()`** is still not thread-safe in the sense that concurrent iterations may see the same object, but they will not corrupt the interpreter state.

Library

- gh-100388 [https://github.com/python/cpython/issues/100388]: Fix the `platform._sys_version()` method when `__DATE__` is undefined at buildtime by changing default buildtime datetime string to the UNIX epoch.

Core and Builtins

- bpo-44369 [https://bugs.python.org/issue?@action=redirect&bpo=44369]: Improve syntax errors for incorrectly closed strings. Patch by Pablo Galindo

C API

- gh-111178 [https://github.com/python/cpython/issues/111178]: Fix `PyCMethod` API: replace `size_t nargs` with `Py_ssize_t nargs` in `PyCMethod`. Patch by Victor Stinner.

- gh-130947 [https://github.com/python/cpython/issues/130947]: Add again `PySequence_Fast()` to the limited C API. Patch by Victor Stinner.

- gh-128863 [https://github.com/python/cpython/issues/128863]: The following private functions are deprecated and planned for removal in Python 3.18:

    - **`_PyUnicodeWriter_Init()`**: replace `_PyUnicodeWriter_Init(&writer)` with `writer = PyUnicodeWriter_Create(0)`.
    - **`_PyUnicodeWriter_Finish()`**: replace `_PyUnicodeWriter_Finish(&writer)` with `PyUnicodeWriter_Finish(writer)`.
    - **`_PyUnicodeWriter_Dealloc()`**: replace `_PyUnicodeWriter_Dealloc(&writer)` with `PyUnicodeWriter_Discard(writer)`.
    - **`_PyUnicodeWriter_WriteChar()`**: replace `_PyUnicodeWriter_WriteChar(&writer, ch)` with `PyUnicodeWriter_WriteChar(writer, ch)`.
    - **`_PyUnicodeWriter_WriteStr()`**: replace `_PyUnicodeWriter_WriteStr(&writer, str)` with `PyUnicodeWriter_WriteStr(writer, str)`.
    - **`_PyUnicodeWriter_WriteSubstring()`**: replace `_PyUnicodeWriter_WriteSubstring(&writer, str, start, end)` with `PyUnicodeWriter_WriteSubstring(writer, str, start, end)`.
    - **`_PyUnicodeWriter_WriteASCIIString()`**: replace `_PyUnicodeWriter_WriteASCIIString(&writer, str)` with `PyUnicodeWriter_WriteUTF8(writer, str)`.
    - **`_PyUnicodeWriter_WriteLatin1String()`**: replace `_PyUnicodeWriter_WriteLatin1String(&writer, str)` with

  `PyUnicodeWriter_WriteUTF8(writer, str)`.
  - **`_PyUnicodeWriter_Prepare()`**: (no replacement).
  - **`_PyUnicodeWriter_PrepareKind()`**: (no replacement).

  The pythoncapi-compat project [https://github.com/python/pythoncapi-compat/] can be used to get these new public functions on Python 3.13 and older.

  Patch by Victor Stinner.

- gh-45325 [https://github.com/python/cpython/issues/45325]: Add a new `p` format parameter to **`Py_BuildValue()`** that allows to take a C integer and produce a Python **`bool`** object. Patch by Pablo Galindo.

Build

- gh-131035 [https://github.com/python/cpython/issues/131035]: Use `-flto=thin` for faster build times using clang-cl on Windows. Patch by Chris Eibl.
- gh-130740 [https://github.com/python/cpython/issues/130740]: Ensure that `Python.h` is included before `stdbool.h` unless `pyconfig.h` is included before or in some platform-specific contexts.
- gh-130090 [https://github.com/python/cpython/issues/130090]: Building with `PlatformToolset=ClangCL` on Windows now supports PGO (profile guided optimization). Patch by Chris Eibl with invaluable support from Steve Dover.
- gh-129819 [https://github.com/python/cpython/issues/129819]: Allow building the JIT with the tailcall interpreter.
- gh-129989 [https://github.com/python/cpython/issues/129989]: Fix a bug where the tailcall interpreter was enabled when `--without-tail-call-interp` was provided to the configure script.
- gh-129838 [https://github.com/python/cpython/issues/129838]: Don’t redefine `_Py_NO_SANITIZE_UNDEFINED` when compiling with a

recent GCC version and undefined sanitizer enabled.

- gh-82909 [https://github.com/python/cpython/issues/82909]: `#pragma`-based linking with `python3*.lib` can now be switched off with Py_NO_LINK_LIB. Patch by Jean-Christophe Fillion-Robin.

Python 3.14.0 alpha 5

Release date: 2025-02-11

macOS

- gh-91132 [https://github.com/python/cpython/issues/91132]: Update macOS installer to use ncurses 6.5.

Tools/Demos

- gh-129248 [https://github.com/python/cpython/issues/129248]: The iOS test runner now strips the log prefix from each line output by the test suite.
- gh-104400 [https://github.com/python/cpython/issues/104400]: Fix several bugs in extraction by switching to an AST parser in **pygettext**.

Tests

- gh-129386 [https://github.com/python/cpython/issues/129386]: Add `test.support.reset_code`, which can be used to reset various bytecode-level optimizations and local instrumentation for a function.
- gh-128474 [https://github.com/python/cpython/issues/128474]: Disable `test_embed` test cases that segfault on BOLT instrument binaries. The tests are only disabled when BOLT is enabled.

- gh-128003 [https://github.com/python/cpython/issues/128003]: Add an option `--parallel-threads=N` to the regression test runner that runs individual tests in multiple threads in parallel in order to find concurrency bugs. Note that most of the test suite is not yet reviewed for thread-safety or annotated with `@thread_unsafe` when necessary.

Security

- gh-105704 [https://github.com/python/cpython/issues/105704]: When using **`urllib.parse.urlsplit()`** and **`urllib.parse.urlparse()`** host parsing would not reject domain names containing square brackets (`[` and `]`). Square brackets are only valid for IPv6 and IPvFuture hosts according to RFC 3986 Section 3.2.2 [https://www.rfc-editor.org/rfc/rfc3986#section-3.2.2].
- gh-126108 [https://github.com/python/cpython/issues/126108]: Fix a possible `NULL` pointer dereference in **`PySys_AddWarnOptionUnicode()`**.
- gh-80222 [https://github.com/python/cpython/issues/80222]: Fix bug in the folding of quoted strings when flattening an email message using a modern email policy. Previously when a quoted string was folded so that it spanned more than one line, the surrounding quotes and internal escapes would be omitted. This could theoretically be used to spoof header lines using a carefully constructed quoted string if the resulting rendered email was transmitted or re-parsed.
- gh-119511 [https://github.com/python/cpython/issues/119511]: Fix a potential denial of service in the **`imaplib`** module. When connecting to a malicious server, it could cause an arbitrary amount of memory to be allocated. On many systems this is harmless as unused virtual memory is only a mapping, but if this hit a virtual address size limit it could lead to a **`MemoryError`** or other process crash. On unusual systems or builds where all allocated memory is

touched and backed by actual ram or storage it could’ve consumed resources doing so until similarly crashing.

Library

- gh-129939 [https://github.com/python/cpython/issues/129939]: Comparison pages with highlighted changes generated by the `difflib.HtmlDiff` class now support dark mode.
- gh-129928 [https://github.com/python/cpython/issues/129928]: Raise `sqlite3.ProgrammingError` if a user-defined SQL function with invalid number of parameters is created. Patch by Erlend Aasland.
- gh-129583 [https://github.com/python/cpython/issues/129583]: Update bundled pip to 25.0.1
- gh-129766 [https://github.com/python/cpython/issues/129766]: Fix crash in `warnings`, when calling `_release_lock()` with no existing lock.
- gh-129005 [https://github.com/python/cpython/issues/129005]: `_pyio.FileIO.readall()` now allocates, resizes, and fills a data buffer using the same algorithm `_io.FileIO.readall()` uses.
- gh-129646 [https://github.com/python/cpython/issues/129646]: Update the locale alias mapping in the `locale` module to match the latest X Org locale alias mapping and support new locales in Glibc 2.41.
- gh-128317 [https://github.com/python/cpython/issues/128317]: Put CLI calendar highlighting in private class, removing `highlight_day` from public `calendar.TextCalendar` API. Patch by Hugo van Kemenade.
- gh-129603 [https://github.com/python/cpython/issues/129603]: Fix bugs where `sqlite3.Row` objects could segfault if their inherited `description` was set to `None`. Patch by Erlend Aasland.
- gh-129559 [https://github.com/python/cpython/issues/129559]: Add `bytearray.resize()` method so `bytearray` can be efficiently resized in place.

- gh-129502 [https://github.com/python/cpython/issues/129502]: Unlikely errors in preparing arguments for **`ctypes`** callback are now handled in the same way as errors raised in the callback of in converting the result of the callback – using **`sys.unraisablehook()`** instead of **`sys.excepthook()`** and not setting **`sys.last_exc`** and other variables.
- gh-129403 [https://github.com/python/cpython/issues/129403]: Corrected **`ValueError`** message for **`asyncio.Barrier`** and **`threading.Barrier`**.
- gh-129409 [https://github.com/python/cpython/issues/129409]: Fix an integer overflow in the **`csv`** module when writing a data field larger than 2GB.
- gh-126400 [https://github.com/python/cpython/issues/126400]: Add a socket *timeout* keyword argument to **`logging.handlers.SysLogHandler`**.
- gh-118761 [https://github.com/python/cpython/issues/118761]: Always lazy import `warnings` in **`threading`**. Patch by Taneli Hukkinen.
- gh-118761 [https://github.com/python/cpython/issues/118761]: Improve import time of **`subprocess`** by lazy importing `locale` and `signal`. Patch by Taneli Hukkinen.
- gh-129346 [https://github.com/python/cpython/issues/129346]: In **`sqlite3`**, handle out-of-memory when creating user-defined SQL functions.
- gh-129005 [https://github.com/python/cpython/issues/129005]: Optimize `_pyio.FileIO.readinto` by avoiding unnecessary objects and copies using **`os.readinto()`**.
- gh-129195 [https://github.com/python/cpython/issues/129195]: Support reporting call graph information from **`asyncio.staggered.staggered_race()`**.
- gh-129205 [https://github.com/python/cpython/issues/129205]: Add **`os.readinto()`** to read into a buffer object from a file descriptor.
- gh-128772 [https://github.com/python/cpython/issues/128772]: Fix **`pydoc`** for methods with the `__module__` attribute equal to

`None`.

- gh-129061 [https://github.com/python/cpython/issues/129061]: Fix FORCE_COLOR and NO_COLOR when empty strings. Patch by Hugo van Kemenade.
- gh-92897 [https://github.com/python/cpython/issues/92897]: Scheduled the deprecation of the `check_home` argument of `sysconfig.is_python_build()` to Python 3.15.
- gh-129064 [https://github.com/python/cpython/issues/129064]: Deprecate **`sysconfig.expand_makefile_vars()`**, in favor of using `sysconfig.get_paths()` with the `vars` argument.
- gh-128550 [https://github.com/python/cpython/issues/128550]: Removed an incorrect optimization relating to eager tasks in `asyncio.TaskGroup` that resulted in cancellations being missed.
- gh-128991 [https://github.com/python/cpython/issues/128991]: Release the enter frame reference within `bdb` callback
- gh-118761 [https://github.com/python/cpython/issues/118761]: Reduce import time of `pstats` and `zipfile` by up to 20%, by removing unnecessary imports to `typing`. Patch by Bénédikt Tran.
- gh-128978 [https://github.com/python/cpython/issues/128978]: Fix a `NameError` in **`sysconfig.expand_makefile_vars()`**. Patch by Bénédikt Tran.
- gh-128961 [https://github.com/python/cpython/issues/128961]: Fix a crash when setting state on an exhausted `array.array` iterator.
- gh-128894 [https://github.com/python/cpython/issues/128894]: Fix `traceback.TracebackException._format_syntax_error` not to fail on exceptions with custom metadata.
- gh-128916 [https://github.com/python/cpython/issues/128916]: Do not attempt to set `SO_REUSEPORT` on sockets of address families other than `AF_INET` and `AF_INET6`, as it is meaningless with these address families, and the call with fail with Linux kernel 6.12.9 and newer.
- gh-118761 [https://github.com/python/cpython/issues/118761]: Improve import time of `tomllib` by removing `typing`,

`string`, and `tomllib._types` imports. Patch by Taneli Hukkinen.

- gh-128679 [https://github.com/python/cpython/issues/128679]: **`tracemalloc`**: Fix race conditions when **`tracemalloc.stop()`** is called by a thread, while other threads are tracing memory allocations. Patch by Victor Stinner.
- gh-128891 [https://github.com/python/cpython/issues/128891]: Add specialized opcodes to `opcode.opname`.
- gh-118761 [https://github.com/python/cpython/issues/118761]: Reduce import time of **`gettext`** by up to ten times, by importing **`re`** on demand. In particular, `re` is no longer implicitly exposed as `gettext.re`. Patch by Eli Schwartz.
- gh-118761 [https://github.com/python/cpython/issues/118761]: Reduce the import time of **`optparse`** when no help text is printed. Patch by Eli Schwartz.
- gh-128657 [https://github.com/python/cpython/issues/128657]: Fix possible extra reference when using objects returned by **`hashlib.sha256()`** under free threading.
- gh-118761 [https://github.com/python/cpython/issues/118761]: Reduce the import time of **`csv`** by up to five times, by importing **`re`** on demand. In particular, `re` is no more implicitly exposed as `csv.re`. Patch by Bénédikt Tran.
- gh-128308 [https://github.com/python/cpython/issues/128308]: Support the *name* keyword argument for eager tasks in **`asyncio.loop.create_task()`**, **`asyncio.create_task()`** and **`asyncio.TaskGroup.create_task()`**, by passing on all *kwargs* to the task factory set by **`asyncio.loop.set_task_factory()`**.
- gh-118761 [https://github.com/python/cpython/issues/118761]: Improve the performance of **`base64.b16decode()`** by up to ten times by more efficiently checking the byte-string for hexadecimal digits. Reduce the import time of **`base64`** by up to six times, by no longer importing **`re`**. Patch by Bénédikt Tran, Chris Markiewicz, and Adam Turner.
- gh-128156 [https://github.com/python/cpython/issues/128156]: When using macOS system `libffi`, support for complex types in **`ctypes`** is now checked at runtime (macOS 10.15

or newer). The types must also be available at build time.

- gh-128636 [https://github.com/python/cpython/issues/128636]: Fix PyREPL failure when **`os.environ`** is overwritten with an invalid value.
- gh-128498 [https://github.com/python/cpython/issues/128498]: Default to stdout isatty for color detection instead of stderr. Patch by Hugo van Kemenade.
- gh-128384 [https://github.com/python/cpython/issues/128384]: Add locking to **`warnings`** to avoid some data races when free-threading is used. Change `_warnings_runtime_state.mutex` to be a recursive mutex and expose it to **`warnings`**, via the **`_acquire_lock()`** and **`_release_lock()`** functions. The lock is held when `filters` and `_filters_version` are updated.
- gh-128509 [https://github.com/python/cpython/issues/128509]: Add **`sys._is_immortal()`** for identifying immortal objects at runtime.
- gh-128479 [https://github.com/python/cpython/issues/128479]: Fix **`asyncio.staggered.staggered_race()`** leaking tasks and issuing an unhandled exception.
- gh-128427 [https://github.com/python/cpython/issues/128427]: **`uuid.NIL`** and **`uuid.MAX`** are now available to represent the Nil and Max UUID formats as defined by **RFC 9562** [https://datatracker.ietf.org/doc/html/rfc9562.html].
- gh-91279 [https://github.com/python/cpython/issues/91279]: **`zipfile.ZipFile.writestr()`** now respect `SOURCE_DATE_EPOCH` that distributions can set centrally and have build tools consume this in order to produce reproducible output.
- gh-112064 [https://github.com/python/cpython/issues/112064]: Fix incorrect handling of negative read sizes in **`HTTPResponse.read`**. Patch by Yury Manushkin.
- gh-58956 [https://github.com/python/cpython/issues/58956]: Fixed a frame reference leak in **`bdb`**.
- gh-128131 [https://github.com/python/cpython/issues/128131]: Completely support random access of uncompressed

unencrypted read-only zip files obtained by `ZipFile.open`.

- gh-127975 [https://github.com/python/cpython/issues/127975]: Avoid reusing quote types in `ast.unparse()` if not needed.
- gh-115514 [https://github.com/python/cpython/issues/115514]: Fix exceptions and incomplete writes after **`asyncio._SelectorTransport`** is closed before writes are completed.
- gh-121604 [https://github.com/python/cpython/issues/121604]: Add missing Deprecation warnings for `importlib.machinery.DEBUG_BYTECODE_SUFFIXES`, `importlib.machinery.OPTIMIZED_BYTECODE_SUFFIXES`, `importlib.machinery.WindowsRegistryFinder`, `importlib.abc.ResourceLoader`, `importlib.abc.SourceLoader.path_mtime()`.
- gh-127873 [https://github.com/python/cpython/issues/127873]: When `-E` is set, only ignore `PYTHON_COLORS` and not `FORCE_COLOR`/`NO_COLOR`/`TERM` when colourising output. Patch by Hugo van Kemenade.
- gh-125413 [https://github.com/python/cpython/issues/125413]: Add `pathlib.Path.info` attribute, which stores an object implementing the `pathlib.types.PathInfo` protocol (also new). The object supports querying the file type and internally caching `stat()` results. Path objects generated by `iterdir()` are initialized with file type information gleaned from scanning the parent directory.
- gh-127712 [https://github.com/python/cpython/issues/127712]: Fix handling of the `secure` argument of `logging.handlers.SMTPHandler`.
- gh-127096 [https://github.com/python/cpython/issues/127096]: Do not recreate unnamed section on every read in `configparser.ConfigParser`. Patch by Andrey Efremov.
- gh-124369 [https://github.com/python/cpython/issues/124369]: Deprecate `pdb.Pdb.curframe_locals`
- gh-126332 [https://github.com/python/cpython/issues/126332]: Fix _pyrepl crash when entering a double CTRL-Z on an overflowing line.

- gh-125553 [https://github.com/python/cpython/issues/125553]: Fix round-trip invariance for backslash continuations in `tokenize.untokenize()`.
- gh-91048 [https://github.com/python/cpython/issues/91048]: Add `asyncio.capture_call_graph()` and `asyncio.print_call_graph()` functions.
- gh-124703 [https://github.com/python/cpython/issues/124703]: Quitting `pdb` in `inline` mode will emit a confirmation prompt and exit gracefully now, instead of printing an exception traceback.
- gh-123987 [https://github.com/python/cpython/issues/123987]: Fixed issue in NamespaceReader where a non-path item in a namespace path, such as a sentinel added by an editable installer, would break resource loading.
- gh-119349 [https://github.com/python/cpython/issues/119349]: Add the `ctypes.util.dllist()` function to list the loaded shared libraries for the current process.
- gh-55454 [https://github.com/python/cpython/issues/55454]: Add IMAP4 `IDLE` support to the `imaplib` module. Patch by Forest.
- gh-119257 [https://github.com/python/cpython/issues/119257]: Show tab completions menu below the current line, which results in less janky behaviour, and fixes a cursor movement bug. Patch by Daniel Hollas
- gh-101410 [https://github.com/python/cpython/issues/101410]: Support custom messages for domain errors in the `math` module (`math.sqrt()`, `math.log()` and `math.atanh()` were modified as examples). Patch by Charlie Zhao and Sergey B Kirpichev.
- gh-81340 [https://github.com/python/cpython/issues/81340]: Use `os.copy_file_range()` in `shutil.copy()`, `shutil.copy2()`, and `shutil.copyfile()` functions by default. An underlying Linux system call gives filesystems an opportunity to implement the use of copy-on-write (in case of btrfs and XFS) or server-side copy (in the case of NFS.) Patch by Illia Volochii.

- bpo-27307 [https://bugs.python.org/issue?@action=redirect&bpo=27307]: Add attribute and item access support to **`string.Formatter`** in auto-numbering mode, which allows format strings like '{.name}' and '{[1]}'.

IDLE

- gh-129873 [https://github.com/python/cpython/issues/129873]: Simplify displaying the IDLE doc by only copying the text section of idle.html to idlelib/help.html. Patch by Stan Ulbrych.

Documentation

- gh-125722 [https://github.com/python/cpython/issues/125722]: Require Sphinx 8.1.3 or later to build the Python documentation. Patch by Adam Turner.
- gh-67206 [https://github.com/python/cpython/issues/67206]: Document that **`string.printable`** is not printable in the POSIX sense. In particular, **`string.printable.isprintable()`** returns **`False`**. Patch by Bénédikt Tran.

Core and Builtins

- gh-100239 [https://github.com/python/cpython/issues/100239]: Replace the opcode BINARY_SUBSCR and its family by BINARY_OP with oparg NB_SUBSCR.
- gh-129732 [https://github.com/python/cpython/issues/129732]: Fixed a race in `_Py_qsbr_reserve` in the free threading build.
- gh-129763 [https://github.com/python/cpython/issues/129763]: Remove the internal `LLTRACE` macro (use **`Py_DEBUG`** instead).
- gh-129715 [https://github.com/python/cpython/issues/129715]: Improve JIT performance for generators.
- gh-129643 [https://github.com/python/cpython/issues/129643]: Fix thread safety of **`PyList_Insert()`** in free-threading builds.

- gh-129668 [https://github.com/python/cpython/issues/129668]: Fix race condition when raising `MemoryError` in the free threaded build.
- gh-129643 [https://github.com/python/cpython/issues/129643]: Fix thread safety of `PyList_SetItem()` in free-threading builds. Patch by Kumar Aditya.
- gh-128563 [https://github.com/python/cpython/issues/128563]: Fix an issue where the “lltrace” debug feature could have been incorrectly enabled for some frames.
- gh-129393 [https://github.com/python/cpython/issues/129393]: On FreeBSD, `sys.platform` doesn’t contain the major version anymore. It is always `'freebsd'`, instead of `'freebsd13'` or `'freebsd14'`.

Library

- gh-129345 [https://github.com/python/cpython/issues/129345]: Fix null pointer dereference in `syslog.openlog()` when an audit hook raises an exception.

Core and Builtins

- gh-129231 [https://github.com/python/cpython/issues/129231]: Improve memory layout of JIT traces. Patch by Diego Russo
- gh-129149 [https://github.com/python/cpython/issues/129149]: Add fast path for medium-size integers in `PyLong_FromUnsignedLong()`, `PyLong_FromUnsignedLongLong()` and `PyLong_FromSize_t()`.
- gh-129201 [https://github.com/python/cpython/issues/129201]: The free-threaded version of the cyclic garbage collector has been optimized to conditionally use CPU prefetch instructions during the collection. This can reduce collection times by making it more likely that data is in the CPU cache when it is needed. The prefetch instructions are enabled if the number of long-lived

objects (objects surviving a full collection) exceeds a threshold.

- gh-129093 [https://github.com/python/cpython/issues/129093]: Fix f-strings such as `f'{expr=}'` sometimes not displaying the full expression when the expression contains `!=`.
- gh-124363 [https://github.com/python/cpython/issues/124363]: Treat debug expressions in f-string as raw strings. Patch by Pablo Galindo
- gh-128714 [https://github.com/python/cpython/issues/128714]: Fix the potential races in get/set dunder methods `__annotations__`, `__annotate__` and `__type_params__` for function object, and add related tests.
- gh-128799 [https://github.com/python/cpython/issues/128799]: Add frame of `except*` to traceback when it wraps a naked exception.
- gh-128842 [https://github.com/python/cpython/issues/128842]: Collect JIT memory stats using pystats. Patch by Diego Russo.
- gh-100239 [https://github.com/python/cpython/issues/100239]: Specialize `BINARY_OP` for bitwise logical operations on compact ints.
- gh-128910 [https://github.com/python/cpython/issues/128910]: Undocumented and unused private C-API functions `_PyTrash_begin` and `_PyTrash_end` are removed.
- gh-128807 [https://github.com/python/cpython/issues/128807]: Add a marking phase to the free-threaded GC. This is similar to what was done in gh-126491 [https://github.com/python/cpython/issues/126491]. Since the free-threaded GC does not have generations and is not incremental, the marking phase looks for all objects reachable from known roots. The roots are objects known to not be garbage, like the module dictionary for **`sys`**. For most programs, this marking phase should make the GC a bit faster since typically less work is done per object.

- gh-100239 [https://github.com/python/cpython/issues/100239]: Add opcode `BINARY_OP_EXTEND` which executes a pair of functions (guard and specialization functions) accessed from the inline cache.
- gh-128563 [https://github.com/python/cpython/issues/128563]: A new type of interpreter has been added to CPython. This interpreter uses tail calls for its instruction handlers. Preliminary benchmark results suggest 7-11% geometric mean faster on pyperformance (depending on platform), and up to 30% faster on Python-intensive workloads. This interpreter currently only works on newer compilers, such as `clang-19`. Other compilers will continue using the old interpreter. Patch by Ken Jin, with ideas on how to implement this in CPython by Mark Shannon, Garret Gu, Haoran Xu, and Josh Haberman.
- gh-126703 [https://github.com/python/cpython/issues/126703]: Improve performance of iterating over lists and tuples by using a freelist for the iterator objects.
- gh-127953 [https://github.com/python/cpython/issues/127953]: The time to handle a `LINE` event in sys.monitoring (and sys.settrace) is now independent of the number of lines in the code object.
- gh-128330 [https://github.com/python/cpython/issues/128330]: Restore terminal control characters on REPL exit.
- gh-128016 [https://github.com/python/cpython/issues/128016]: Improved the `SyntaxWarning` message for invalid escape sequences to clarify that such sequences will raise a `SyntaxError` in future Python releases. The new message also suggests a potential fix, i.e., `Did you mean "\\e"?`.
- gh-126004 [https://github.com/python/cpython/issues/126004]: Fix handling of **`UnicodeError.start`** and **`UnicodeError.end`** values in the **`codecs.replace_errors()`** error handler. Patch by Bénédikt Tran.
- gh-126004 [https://github.com/python/cpython/issues/126004]: Fix handling of **`UnicodeError.start`** and **`UnicodeError.end`** values

in the `codecs.backslashreplace_errors()` error handler. Patch by Bénédikt Tran.
- gh-126004 [https://github.com/python/cpython/issues/126004]: Fix handling of `UnicodeError.start` and `UnicodeError.end` values in the `codecs.xmlcharrefreplace_errors()` error handler. Patch by Bénédikt Tran.
- gh-127119 [https://github.com/python/cpython/issues/127119]: Slightly optimize the `int` deallocator.
- gh-127349 [https://github.com/python/cpython/issues/127349]: Fixed the error when resizing terminal in Python REPL. Patch by Semyon Moroz.
- gh-125723 [https://github.com/python/cpython/issues/125723]: Fix crash with `gi_frame.f_locals` when generator frames outlive their generator. Patch by Mikhail Efimov.

Library

- gh-126349 [https://github.com/python/cpython/issues/126349]: Add `turtle.fill()`, `turtle.poly()` and `turtle.no_animation()` context managers. Patch by Marie Roald and Yngve Mardal Moe.

Core and Builtins

- gh-115911 [https://github.com/python/cpython/issues/115911]: If the current working directory cannot be determined due to permissions, then import will no longer raise `PermissionError`. Patch by Alex Willmer.

Library

- gh-112713 [https://github.com/python/cpython/issues/112713]: Added support for the `Partitioned` cookie flag in `http.cookies`.

C API

- gh-129533 [https://github.com/python/cpython/issues/129533]: Update **`PyGC_Enable()`**, **`PyGC_Disable()`**, **`PyGC_IsEnabled()`** to use atomic operation for thread-safety at free-threading build. Patch by Donghee Na.

- gh-89188 [https://github.com/python/cpython/issues/89188]: Implement **`PyUnicode_KIND()`** and **`PyUnicode_DATA()`** as function, in addition to the macros with the same names. The macros rely on C bit fields which have compiler-specific layout. Patch by Victor Stinner.

- gh-91417 [https://github.com/python/cpython/issues/91417]: Remove **`PySequence_Fast()`** from the limited C API, since this function has to be used with **`PySequence_Fast_GET_ITEM`** which never worked in the limited C API. Patch by Victor Stinner.

- gh-128509 [https://github.com/python/cpython/issues/128509]: Add **`PyUnstable_IsImmortal()`** for determining whether an object is immortal.

- gh-129033 [https://github.com/python/cpython/issues/129033]: Remove `_PyInterpreterState_GetConfigCopy()` and `_PyInterpreterState_SetConfig()` private functions. Use instead **`PyConfig_Get()`** and **`PyConfig_Set()`**, public C API added by **PEP 741** [https://peps.python.org/pep-0741/] “Python Configuration C API”. Patch by Victor Stinner.

- gh-129033 [https://github.com/python/cpython/issues/129033]: Remove the private `_Py_InitializeMain()` function. It was a provisional API added to Python 3.8 by **PEP 587** [https://peps.python.org/pep-0587/]. Patch by Victor Stinner.

- gh-128844 [https://github.com/python/cpython/issues/128844]: Add **`PyUnstable_TryIncRef()`** and **`PyUnstable_EnableTryIncRef()`** unstable APIs. These are helpers for dealing with unowned references in a thread-safe way, particularly in the free threading build.

- gh-128911 [https://github.com/python/cpython/issues/128911]: Add **`PyImport_ImportModuleAttr()`** and **`PyImport_ImportModuleAttrString()`** helper functions to import a module and get an attribute of the module. Patch by Victor Stinner.

- gh-128863 [https://github.com/python/cpython/issues/128863]: The following private functions are deprecated and planned for removal in Python 3.18:

  - **`_PyBytes_Join()`**: use **`PyBytes_Join()`**.
  - **`_PyDict_GetItemStringWithError()`**: use **`PyDict_GetItemStringRef()`**.
  - **`_PyDict_Pop()`**: use **`PyDict_Pop()`**.
  - **`_PyLong_Sign()`**: use **`PyLong_GetSign()`**.
  - **`_PyLong_FromDigits()`** and **`_PyLong_New()`**: use **`PyLongWriter_Create()`**.
  - **`_PyThreadState_UncheckedGet()`**: use **`PyThreadState_GetUnchecked()`**.
  - **`_PyUnicode_AsString()`**: use **`PyUnicode_AsUTF8()`**.
  - **`_Py_HashPointer()`**: use **`Py_HashPointer()`**.
  - **`_Py_fopen_obj()`**: use **`Py_fopen()`**.

  The pythoncapi-compat project [https://github.com/python/pythoncapi-compat/] can be used to get these new public functions on Python 3.13 and older.

  Patch by Victor Stinner.

- gh-126599 [https://github.com/python/cpython/issues/126599]: Remove some internal test APIs for the experimental JIT compiler.

- gh-127925 [https://github.com/python/cpython/issues/127925]: Convert the **`decimal`** module to use **PEP 757** [https://peps.python.org/pep-0757/] C API (export-import integers), offering some speed-up if the integer part of the **`Decimal`** instance is small. Patch by Sergey B Kirpichev.

Build

- gh-129660 [https://github.com/python/cpython/issues/129660]: Drop `test_embed` from PGO training, whose contribution in recent versions is considered to be ignorable.
- gh-128902 [https://github.com/python/cpython/issues/128902]: Fix compile errors with Clang 9 and older due to lack of `__attribute__((fallthrough))` support.

Python 3.14.0 alpha 4

Release date: 2025-01-14

macOS

- gh-127592 [https://github.com/python/cpython/issues/127592]: Usage of the unified Apple System Log APIs was disabled when the minimum macOS version is earlier than 10.12.

Tools/Demos

- gh-128152 [https://github.com/python/cpython/issues/128152]: Fix a bug where Argument Clinic’s C pre-processor parser tried to parse pre-processor directives inside C comments. Patch by Erlend Aasland.

Tests

- gh-128690 [https://github.com/python/cpython/issues/128690]: Temporarily do not use test_embed in PGO profile builds until the problem with test_init_pyvenv_cfg failing in some configurations is resolved.

Library

- gh-128731 [https://github.com/python/cpython/issues/128731]: Fix `ResourceWarning` in `urllib.robotparser.RobotFileParser.read()`.
- gh-71339 [https://github.com/python/cpython/issues/71339]: Add new assertion methods for `unittest`: `assertHasAttr()`, `assertNotHasAttr()`, `assertIsSubclass()`, `assertNotIsSubclass()` `assertStartsWith()`, `assertNotStartsWith()`, `assertEndsWith()` and `assertNotEndsWith()`.
- gh-118761 [https://github.com/python/cpython/issues/118761]: Improve import time of `pickle` by 25% by removing an unnecessary regular expression. As such, `re` is no more implicitly available as `pickle.re`. Patch by Bénédikt Tran.
- gh-128661 [https://github.com/python/cpython/issues/128661]: Fixes `typing.evaluate_forward_ref()` not showing deprecation when `type_params` arg is not passed.
- gh-128562 [https://github.com/python/cpython/issues/128562]: Fix possible conflicts in generated `tkinter` widget names if the widget class name ends with a digit.
- gh-128559 [https://github.com/python/cpython/issues/128559]: Improved import time of `asyncio`.
- gh-128552 [https://github.com/python/cpython/issues/128552]: Fix cyclic garbage introduced by `asyncio.loop.create_task()` and `asyncio.TaskGroup.create_task()` holding a reference to the created task if it is eager.

- gh-128340 [https://github.com/python/cpython/issues/128340]: Add internal thread safe handle to be used in `asyncio.loop.call_soon_threadsafe()` for thread safe cancellation.
- gh-128182 [https://github.com/python/cpython/issues/128182]: Fix crash when using `ctypes` pointers concurrently on the free threaded build.
- gh-128400 [https://github.com/python/cpython/issues/128400]: Only show the current thread in `faulthandler` on the free threaded build to prevent races.
- gh-128400 [https://github.com/python/cpython/issues/128400]: Fix crash when using `faulthandler.dump_traceback()` while other threads are active on the free threaded build.
- gh-128388 [https://github.com/python/cpython/issues/128388]: Fix `PyREPL` on Windows to support more keybindings, like the `Control-←` and `Control-→` word-skipping keybindings and those with meta (i.e. `Alt`), e.g. `Alt-d` to `kill-word` or `Alt-Backspace` `backward-kill-word`.
- gh-88834 [https://github.com/python/cpython/issues/88834]: Unify the instance check for `typing.Union` and `types.UnionType`: **`Union`** now uses the instance checks against its parameters instead of the subclass checks.
- gh-128302 [https://github.com/python/cpython/issues/128302]: Fix **`xml.dom.xmlbuilder.DOMEntityResolver.resolveEntity()`**, which was broken by the Python 3.0 transition.
- gh-128317 [https://github.com/python/cpython/issues/128317]: Highlight today in colour in `calendar`'s CLI output. Patch by Hugo van Kemenade.
- gh-128302 [https://github.com/python/cpython/issues/128302]: Allow **`xml.dom.xmlbuilder.DOMParser.parse()`** to correctly handle **`xml.dom.xmlbuilder.DOMInputSource`** instances that only have a **`systemId`** attribute set.
- gh-128151 [https://github.com/python/cpython/issues/128151]: Improve generation of `UUID` objects version 3, 4, 5, and 8 via their dedicated functions by 30%. Patch by Bénédikt Tran.

- gh-128118 [https://github.com/python/cpython/issues/128118]: Improve performance of `copy.copy()` by 30% via a fast path for atomic types and container types.
- gh-127946 [https://github.com/python/cpython/issues/127946]: Fix crash when modifying `ctypes._CFuncPtr` objects concurrently on the free threaded build.
- gh-128062 [https://github.com/python/cpython/issues/128062]: Revert the font of `turtledemo`'s menu bar to its default value and display the shortcut keys in the correct position.
- gh-128014 [https://github.com/python/cpython/issues/128014]: Fix resetting the default window icon by passing `default=''` to the `tkinter` method **`wm_iconbitmap()`**.
- gh-41872 [https://github.com/python/cpython/issues/41872]: Fix quick extraction of module docstrings from a file in `pydoc`. It now supports docstrings with single quotes, escape sequences, raw string literals, and other Python syntax.
- gh-127060 [https://github.com/python/cpython/issues/127060]: Set TERM environment variable to “dumb” to disable traceback colors in IDLE, since IDLE doesn’t understand ANSI escape sequences. Patch by Victor Stinner.
- gh-126742 [https://github.com/python/cpython/issues/126742]: Fix support of localized error messages reported by *dlerror(3) [https://manpages.debian.org/dlerror(3)]* and *gdbm_strerror [https://manpages.debian.org/gdbm(3)]* in `ctypes` and `dbm.gnu` functions respectively. Patch by Bénédikt Tran.
- gh-122548 [https://github.com/python/cpython/issues/122548]: Adds two new local events to sys.monitoring, `BRANCH_LEFT` and `BRANCH_RIGHT`. This allows the two arms of the branch to be disabled independently, which should hugely improve performance of branch-level coverage tools. The old branch event, `BRANCH` is now deprecated.
- gh-127847 [https://github.com/python/cpython/issues/127847]: Fix the position when doing interleaved seeks and reads in

uncompressed, unencrypted zip files returned by **`zipfile.ZipFile.open()`**.

- gh-127688 [https://github.com/python/cpython/issues/127688]: Add the **`SCHED_DEADLINE`** and **`SCHED_NORMAL`** constants to the **`os`** module.
- gh-83662 [https://github.com/python/cpython/issues/83662]: Add missing `__class_getitem__` method to the Python implementation of **`functools.partial()`**, to make it compatible with the C version. This is mainly relevant for alternative Python implementations like PyPy and GraalPy, because CPython will usually use the C-implementation of that function.
- gh-127586 [https://github.com/python/cpython/issues/127586]: **`multiprocessing.pool.Pool`** now properly restores blocked signal handlers of the parent thread when creating processes via either *spawn* or *forkserver*.
- gh-98188 [https://github.com/python/cpython/issues/98188]: Fix an issue in **`email.message.Message.get_payload()`** where data cannot be decoded if the Content Transfer Encoding mechanism contains trailing whitespaces or additional junk text. Patch by Hui Liu.
- gh-127529 [https://github.com/python/cpython/issues/127529]: Correct behavior of **`asyncio.selector_events.BaseSelectorEventLoop._accept_connection()`** in handling **`ConnectionAbortedError`** in a loop. This improves performance on OpenBSD.
- gh-127360 [https://github.com/python/cpython/issues/127360]: When a descriptive error message cannot be provided for an **`ssl.SSLError`**, the “unknown error” message now shows the internal error code (as retrieved by `ERR_get_error` and similar OpenSSL functions).
- gh-127196 [https://github.com/python/cpython/issues/127196]: Fix crash when dict with keys in invalid encoding were passed to several functions in `_interpreters` module.
- gh-124130 [https://github.com/python/cpython/issues/124130]: Fix a bug in matching regular expression `\B` in empty input

string. Now it is always the opposite of `\b`. To get an old behavior, use `(?!\A\Z)\B`. To get a new behavior in old Python versions, use `(?!\b)`.

- gh-126639 [https://github.com/python/cpython/issues/126639]: **`tempfile.NamedTemporaryFile`** will now issue a **`ResourceWarning`** when it is finalized by the garbage collector without being explicitly closed.
- gh-126624 [https://github.com/python/cpython/issues/126624]: Expose error code **`XML_ERROR_NOT_STARTED`** of Expat >=2.6.4 in **`xml.parsers.expat.errors`**.
- gh-126225 [https://github.com/python/cpython/issues/126225]: **`getopt`** and **`optparse`** are no longer marked as deprecated. There are legitimate reasons to use one of these modules in preference to **`argparse`**, and none of these modules are at risk of being removed from the standard library. Of the three, `argparse` remains the recommended default choice, *unless* one of the concerns noted at the top of the `optparse` module documentation applies.
- gh-124761 [https://github.com/python/cpython/issues/124761]: Add **`SO_REUSEPORT_LB`** constant to **`socket`** for FreeBSD.
- gh-121720 [https://github.com/python/cpython/issues/121720]: **`enum.EnumDict`** can now be used without resorting to private API.
- gh-123424 [https://github.com/python/cpython/issues/123424]: Add **`zipfile.ZipInfo._for_archive()`** setting default properties on **`ZipInfo`** objects. Patch by Bénédikt Tran and Jason R. Coombs.
- gh-121676 [https://github.com/python/cpython/issues/121676]: Deprecate calling the Python implementation of **`functools.reduce()`** with a `function` or `sequence` as a keyword argument. This will be forbidden in Python 3.16 in order to match the C implementation.
- gh-112015 [https://github.com/python/cpython/issues/112015]: **`ctypes.memoryview_at()`** now exists to create a **`memoryview`** object that refers to the supplied pointer and length. This works like **`ctypes.string_at()`** except it avoids a

buffer copy, and is typically useful when implementing pure Python callback functions that are passed dynamically-sized buffers.

- gh-95371 [https://github.com/python/cpython/issues/95371]: Added support for other image formats (PNG, PGM, and PPM) to the turtle module. Patch by Shin-myoung-serp.

Core and Builtins

- gh-128078 [https://github.com/python/cpython/issues/128078]: Fix a **`SystemError`** when using **`anext()`** with a default tuple value. Patch by Bénédikt Tran.
- gh-128717 [https://github.com/python/cpython/issues/128717]: Fix a crash when setting the recursion limit while other threads are active on the free threaded build.
- gh-124483 [https://github.com/python/cpython/issues/124483]: Treat `Py_DECREF` and variants as escaping when generating opcode and uop metadata. This prevents the possibility of a `__del__` method causing the JIT to behave incorrectly.
- gh-126703 [https://github.com/python/cpython/issues/126703]: Improve performance of class methods by using a freelist.
- gh-128137 [https://github.com/python/cpython/issues/128137]: Update **`PyASCIIObject`** layout to handle interned field with the atomic operation. Patch by Donghee Na.

Library

- gh-128192 [https://github.com/python/cpython/issues/128192]: Upgrade HTTP digest authentication algorithm for **`urllib.request`** by supporting SHA-256 digest authentication as specified in **RFC 7616** [https://datatracker.ietf.org/doc/html/rfc7616.html].

Core and Builtins

- gh-126868 [https://github.com/python/cpython/issues/126868]: Increase usage of freelist for `int` allocation.
- gh-114203 [https://github.com/python/cpython/issues/114203]: Optimize `Py_BEGIN_CRITICAL_SECTION` for simple recursive calls.
- gh-127705 [https://github.com/python/cpython/issues/127705]: Adds stackref debugging when `Py_STACKREF_DEBUG` is set. Finds all double-closes and leaks, logging the origin and last borrow.

  Inspired by HPy's debug mode. https://docs.hpyproject.org/en/latest/debug-mode.html
- gh-128079 [https://github.com/python/cpython/issues/128079]: Fix a bug where **`except*`** does not properly check the return value of an **`ExceptionGroup`**'s **`split()`** function, leading to a crash in some cases. Now when **`split()`** returns an invalid object, **`except*`** raises a **`TypeError`** with the original raised **`ExceptionGroup`** object chained to it.
- gh-128030 [https://github.com/python/cpython/issues/128030]: Avoid error from calling `PyModule_GetFilenameObject` on a non-module object when importing a non-existent symbol from a non-module object.

Library

- gh-128035 [https://github.com/python/cpython/issues/128035]: Indicate through **`ssl.HAS_PHA`** whether the **`ssl`** module supports TLSv1.3 post-handshake client authentication (PHA). Patch by Will Childs-Klein.

Core and Builtins

- gh-127274 [https://github.com/python/cpython/issues/127274]: Add a new flag, `CO_METHOD`, to **`co_flags`** that indicates whether the code object belongs to a function defined in class scope.
- gh-66409 [https://github.com/python/cpython/issues/66409]: During the path initialization, we now check if `base_exec_prefix` is the same as `base_prefix` before falling back to searching the Python interpreter directory.
- gh-127970 [https://github.com/python/cpython/issues/127970]: We now use the location of the `libpython` runtime library used in the current process to determine **`sys.base_prefix`** on all platforms implementing the dladdr [https://pubs.opengroup.org/onlinepubs/9799919799/functions/dladdr.html] function defined by the UNIX standard — this includes Linux, Android, macOS, iOS, FreeBSD, etc. This was already the case on Windows and macOS Framework builds.
- gh-127773 [https://github.com/python/cpython/issues/127773]: Do not use the type attribute cache for types with incompatible MRO.
- gh-127903 [https://github.com/python/cpython/issues/127903]: `Objects/unicodeobject.c`: fix a crash on DEBUG builds in `_copy_characters` when there is nothing to copy.
- gh-127809 [https://github.com/python/cpython/issues/127809]: Fix an issue where the experimental JIT may infer an incorrect result type for exponentiation (`**` and `**=`), leading to bugs or crashes.
- gh-126862 [https://github.com/python/cpython/issues/126862]: Fix a possible overflow when a class inherits from an absurd number of super-classes. Reported by Valery Fedorenko. Patch by Bénédikt Tran.

C API

- gh-128400 [https://github.com/python/cpython/issues/128400]: `Py_FatalError()` no longer shows all threads on the free threaded build to prevent crashes.
- gh-128629 [https://github.com/python/cpython/issues/128629]: Add macros `Py_PACK_VERSION()` and `Py_PACK_FULL_VERSION()` for bit-packing Python version numbers.
- gh-128008 [https://github.com/python/cpython/issues/128008]: Add `PyWeakref_IsDead()` function, which tests if a weak reference is dead.
- gh-127350 [https://github.com/python/cpython/issues/127350]: Add `Py_fopen()` function to open a file. Similar to the **`fopen()`** function, but the *path* parameter is a Python object and an exception is set on error. Add also `Py_fclose()` function to close a file, function needed for Windows support. Patch by Victor Stinner.

Build

- gh-128627 [https://github.com/python/cpython/issues/128627]: For Emscripten builds the function pointer cast call trampoline now uses the wasm-gc ref.test instruction if it’s available instead of Wasm JS type reflection.
- gh-128472 [https://github.com/python/cpython/issues/128472]: Skip BOLT optimization of functions using computed gotos, fixing errors on build with LLVM 19.
- gh-115765 [https://github.com/python/cpython/issues/115765]: GNU Autoconf 2.72 is now required to generate `configure`. Patch by Erlend Aasland.
- gh-123925 [https://github.com/python/cpython/issues/123925]: Fix building the `curses` module on platforms with libncurses but without libncursesw.
- gh-90905 [https://github.com/python/cpython/issues/90905]: Add support for cross-compiling to x86_64 on aarch64/arm64 macOS.
- gh-128321 [https://github.com/python/cpython/issues/128321]: Set `LIBS` instead of `LDFLAGS` when checking if `sqlite3` library

functions are available. This fixes the ordering of linked libraries during checks, which was incorrect when using a statically linked `libsqlite3`.

- gh-100384 [https://github.com/python/cpython/issues/100384]: Error on `unguarded-availability` in macOS builds, preventing invalid use of symbols that are not available in older versions of the OS.
- gh-128104 [https://github.com/python/cpython/issues/128104]: Remove `Py_STRFTIME_C99_SUPPORT` conditions in favor of requiring C99 *strftime(3)* *[https://manpages.debian.org/strftime(3)]* specifier support at build time. When cross-compiling, there is no build time check and support is assumed.
- gh-127951 [https://github.com/python/cpython/issues/127951]: Add option `--pystats` to the Windows build to enable performance statistics collection.

Python 3.14.0 alpha 3

Release date: 2024-12-17

Windows

- gh-127353 [https://github.com/python/cpython/issues/127353]: Allow to force color output on Windows using environment variables. Patch by Andrey Efremov.
- gh-125729 [https://github.com/python/cpython/issues/125729]: Makes the presence of the **`turtle`** module dependent on the Tcl/Tk installer option. Previously, the module was always installed but would be unusable without Tcl/Tk.

Tools/Demos

- gh-126700 [https://github.com/python/cpython/issues/126700]: Add support for multi-argument **`gettext`** functions in **pygettext.py**.

Tests

- gh-127906 [https://github.com/python/cpython/issues/127906]: Test the limited C API in test_cppext. Patch by Victor Stinner.
- gh-127637 [https://github.com/python/cpython/issues/127637]: Add tests for the **`dis`** command-line interface. Patch by Bénédikt Tran.
- gh-126925 [https://github.com/python/cpython/issues/126925]: iOS test results are now streamed during test execution, and the deprecated xcresulttool is no longer used.
- gh-127076 [https://github.com/python/cpython/issues/127076]: Disable strace based system call tests when LD_PRELOAD is set.
- gh-127076 [https://github.com/python/cpython/issues/127076]: Filter out memory-related `mmap`, `munmap`, and `mprotect` calls from file-related ones when testing **`io`** behavior using strace.

Security

- gh-127655 [https://github.com/python/cpython/issues/127655]: Fixed the **`asyncio.selector_events._SelectorSocketTransport`** transport not pausing writes for the protocol when the buffer reaches the high water mark when using **`asyncio.WriteTransport.writelines()`**.

Library

- gh-126907 [https://github.com/python/cpython/issues/126907]: Fix crash when using **`atexit`** concurrently on the free-threaded build.

- gh-127870 [https://github.com/python/cpython/issues/127870]: Detect recursive calls in ctypes `_as_parameter_` handling. Patch by Victor Stinner.

- gh-127732 [https://github.com/python/cpython/issues/127732]: The **`platform`** module now correctly detects Windows Server 2025.

- gh-126789 [https://github.com/python/cpython/issues/126789]: Fixed **`sysconfig.get_config_vars()`**, **`sysconfig.get_paths()`**, and siblings, returning outdated cached data if the value of **`sys.prefix`** or **`sys.exec_prefix`** changes. Overwriting **`sys.prefix`** or **`sys.exec_prefix`** still is discouraged, as that might break other parts of the code.

- gh-127718 [https://github.com/python/cpython/issues/127718]: Add colour to **`test.regrtest`** output. Patch by Hugo van Kemenade.

- gh-127610 [https://github.com/python/cpython/issues/127610]: Added validation for more than one var-positional or var-keyword parameters in **`inspect.Signature`**. Patch by Maxim Ageev.

- gh-127627 [https://github.com/python/cpython/issues/127627]: Added `posix._emscripten_debugger()` to help with debugging the test suite on the Emscripten target.

- gh-126821 [https://github.com/python/cpython/issues/126821]: macOS and iOS apps can now choose to redirect stdout and stderr to the system log during interpreter configuration.

- gh-93312 [https://github.com/python/cpython/issues/93312]: Include `<sys/pidfd.h>` to get `os.PIDFD_NONBLOCK` constant. Patch by Victor Stinner.

- gh-127481 [https://github.com/python/cpython/issues/127481]: Add the `EPOLLWAKEUP` constant to the **`select`** module.

- gh-127065 [https://github.com/python/cpython/issues/127065]: Make `operator.methodcaller()` thread-safe and re-entrant safe.
- gh-127321 [https://github.com/python/cpython/issues/127321]: `pdb.set_trace()` will not stop at an opcode that does not have an associated line number anymore.
- gh-127429 [https://github.com/python/cpython/issues/127429]: Fixed bug where, on cross-builds, the `sysconfig` POSIX data was being generated with the host Python’s `Makefile`. The data is now generated from current build’s `Makefile`.
- gh-127413 [https://github.com/python/cpython/issues/127413]: Add the `dis --specialized` command-line option to show specialized bytecode. Patch by Bénédikt Tran.
- gh-125413 [https://github.com/python/cpython/issues/125413]: Revert addition of `pathlib.Path.scandir()`. This method was added in 3.14.0a2. The optimizations remain for file system paths, but other subclasses should only have to implement `pathlib.Path.iterdir()`.
- gh-127257 [https://github.com/python/cpython/issues/127257]: In `ssl`, system call failures that OpenSSL reports using ERR_LIB_SYS are now raised as `OSError`.
- gh-59705 [https://github.com/python/cpython/issues/59705]: On Linux, `threading.Thread` now sets the thread name to the operating system. Patch by Victor Stinner.
- gh-127303 [https://github.com/python/cpython/issues/127303]: Publicly expose `EXACT_TOKEN_TYPES` in `token.__all__`.
- gh-127331 [https://github.com/python/cpython/issues/127331]: `ssl` can show descriptions for errors added in OpenSSL 3.4.

- gh-123967 [https://github.com/python/cpython/issues/123967]: Fix faulthandler for trampoline frames. If the top-most frame is a trampoline frame, skip it. Patch by Victor Stinner.

- gh-127178 [https://github.com/python/cpython/issues/127178]: A `_sysconfig_vars_(...).json` file is now shipped in the standard library directory. It contains the output of **`sysconfig.get_config_vars()`** on the default environment encoded as JSON data. This is an implementation detail, and may change at any time.

- gh-127072 [https://github.com/python/cpython/issues/127072]: Remove outdated `socket.NETLINK_*` constants not present in Linux kernels beyond 2.6.17.

- gh-127255 [https://github.com/python/cpython/issues/127255]: The **`CopyComPointer()`** function is now public. Previously, this was private and only available in `_ctypes`.

- gh-127182 [https://github.com/python/cpython/issues/127182]: Fix **`io.StringIO.__setstate__()`** crash, when **`None`** was passed as the first value.

- gh-127217 [https://github.com/python/cpython/issues/127217]: Fix **`urllib.request.pathname2url()`** for paths starting with multiple slashes on Posix.

- gh-125866 [https://github.com/python/cpython/issues/125866]: **`urllib.request.pathname2url()`** now adds an empty authority when generating a URL for a path that begins with exactly one slash. For example, the path `/etc/hosts` is converted to the scheme-less URL `///etc/hosts`. As a result of this change, URLs without authorities are only generated for relative paths.

- gh-127221 [https://github.com/python/cpython/issues/127221]: Add colour to `unittest` output. Patch by Hugo van Kemenade.

- gh-127035 [https://github.com/python/cpython/issues/127035]: Fix `shutil.which` on Windows. Now it looks at direct match if and only if the command ends with a PATHEXT extension or X_OK is not in mode. Support extensionless files if "." is in PATHEXT. Support PATHEXT extensions that end with a dot.

- gh-122273 [https://github.com/python/cpython/issues/122273]: Support PyREPL history on Windows. Patch by devdanzin and Victor Stinner.

- gh-125866 [https://github.com/python/cpython/issues/125866]: `urllib.request.pathname2url()` and `url2pathname()` no longer convert Windows drive letters to uppercase.

- gh-127078 [https://github.com/python/cpython/issues/127078]: Fix issue where `urllib.request.url2pathname()` failed to discard an extra slash before a UNC drive in the URL path on Windows.

- gh-126766 [https://github.com/python/cpython/issues/126766]: Fix issue where `urllib.request.url2pathname()` failed to discard any 'localhost' authority present in the URL.

- gh-127065 [https://github.com/python/cpython/issues/127065]: Fix crash when calling a `operator.methodcaller()` instance from multiple threads in the free threading build.

- gh-127090 [https://github.com/python/cpython/issues/127090]: Fix value of `urllib.response.addinfourl.url` for `file:` URLs that express relative paths and absolute Windows paths. The canonical URL generated by `urllib.request.pathname2url()` is now used.

- gh-126992 [https://github.com/python/cpython/issues/126992]: Fix LONG and INT opcodes to only use base 10 for string to integer conversion in **`pickle`**.

- gh-126997 [https://github.com/python/cpython/issues/126997]: Fix support of STRING and GLOBAL opcodes with non-ASCII arguments in **`pickletools`**. **`pickletools.dis()`** now outputs non-ASCII bytes in STRING, BINSTRING and SHORT_BINSTRING arguments as escaped (`\xXX`).

- gh-126316 [https://github.com/python/cpython/issues/126316]: **`grp`**: Make **`grp.getgrall()`** thread-safe by adding a mutex. Patch by Victor Stinner.

- gh-126618 [https://github.com/python/cpython/issues/126618]: Fix the representation of **`itertools.count`** objects when the count value is **`sys.maxsize`**.

- gh-126615 [https://github.com/python/cpython/issues/126615]: The **`COMError`** exception is now public. Previously, this was private and only available in `_ctypes`.

- gh-126985 [https://github.com/python/cpython/issues/126985]: When running under a virtual environment with the **`site`** disabled (see **`-S`**), **`sys.prefix`** and **`sys.base_prefix`** will now point to the virtual environment, instead of the base installation.

- gh-112192 [https://github.com/python/cpython/issues/112192]: In the **`trace`** module, increase the coverage precision (`cov%`) to one decimal.

- gh-118761 [https://github.com/python/cpython/issues/118761]: Improve import time of **`mimetypes`** by around 11-16 times. Patch by Hugo van Kemenade.

- gh-126947 [https://github.com/python/cpython/issues/126947]: Raise `TypeError` in `_pydatetime.timedelta.__new__()` if the passed arguments are not `int` or `float`, so that the Python implementation is in line with the C implementation.

- gh-126946 [https://github.com/python/cpython/issues/126946]: Improve the `GetoptError` error message when a long option prefix matches multiple accepted options in `getopt.getopt()` and `getopt.gnu_getopt()`.

- gh-126899 [https://github.com/python/cpython/issues/126899]: Make tkinter widget methods `after()` and `after_idle()` accept arguments passed by keyword.

- gh-85168 [https://github.com/python/cpython/issues/85168]: Fix issue where `urllib.request.url2pathname()` and `pathname2url()` always used UTF-8 when quoting and unquoting file URIs. They now use the filesystem encoding and error handler.

- gh-126780 [https://github.com/python/cpython/issues/126780]: Fix `os.path.normpath()` for drive-relative paths on Windows.

- gh-126775 [https://github.com/python/cpython/issues/126775]: Make `linecache.checkcache()` thread safe and GC re-entrancy safe.

- gh-126601 [https://github.com/python/cpython/issues/126601]: Fix issue where `urllib.request.pathname2url()` raised `OSError` when given a Windows path containing a colon character not following a drive letter, such as before an NTFS alternate data stream.

- gh-126727 [https://github.com/python/cpython/issues/126727]: `locale.nl_langinfo(locale.ERA)` now returns multiple era description segments separated by semicolons.

Previously it only returned the first segment on platforms with Glibc.

- gh-85046 [https://github.com/python/cpython/issues/85046]: Add `EHWPOISON` error code to `errno`.
- gh-118201 [https://github.com/python/cpython/issues/118201]: Fixed intermittent failures of `os.confstr`, `os.pathconf` and `os.sysconf` on iOS and Android.
- gh-86463 [https://github.com/python/cpython/issues/86463]: The `usage` parameter of `argparse.ArgumentParser` no longer affects the default value of the `prog` parameter in subparsers.
- gh-124008 [https://github.com/python/cpython/issues/124008]: Fix possible crash (in debug build), incorrect output or returning incorrect value from raw binary `write()` when writing to console on Windows.
- gh-123401 [https://github.com/python/cpython/issues/123401]: The `http.cookies` module now supports parsing obsolete **RFC 850** [https://datatracker.ietf.org/doc/html/rfc850.html] date formats, in accordance with **RFC 9110** [https://datatracker.ietf.org/doc/html/rfc9110.html] requirements. Patch by Nano Zheng.
- gh-122431 [https://github.com/python/cpython/issues/122431]: `readline.append_history_file()` now raises a `ValueError` when given a negative value.
- gh-122356 [https://github.com/python/cpython/issues/122356]: Guarantee that the position of a file-like object passed to `zipfile.is_zipfile()` is left untouched after the call. Patch by Bénédikt Tran.

- gh-122288 [https://github.com/python/cpython/issues/122288]: Improve the performances of `fnmatch.translate()` by a factor 1.7. Patch by Bénédikt Tran.

- gh-88110 [https://github.com/python/cpython/issues/88110]: Fixed `multiprocessing.Process` reporting a `.exitcode` of 1 even on success when using the `"fork"` start method while using a `concurrent.futures.ThreadPoolExecutor`.

- gh-97514 [https://github.com/python/cpython/issues/97514]: Authentication was added to the `multiprocessing` forkserver start method control socket so that only processes with the authentication key generated by the process that spawned the forkserver can control it. This is an enhancement over the other gh-97514 [https://github.com/python/cpython/issues/97514] fixes so that access is no longer limited only by filesystem permissions.

  The file descriptor exchange of control pipes with the forked worker process now requires an explicit acknowledgement byte to be sent over the socket after the exchange on all forkserver supporting platforms. That makes testing the above much easier.

Documentation

- gh-127347 [https://github.com/python/cpython/issues/127347]: Publicly expose `traceback.print_list()` in **`traceback.__all__`**.

Core and Builtins

- gh-127740 [https://github.com/python/cpython/issues/127740]: Fix error message in `bytes.fromhex()` when given an odd number of digits to properly indicate that an even number of hexadecimal digits is required.

- gh-127058 [https://github.com/python/cpython/issues/127058]: `PySequence_Tuple` now creates the resulting tuple atomically, preventing partially created tuples being visible to the garbage collector or through `gc.get_referrers()`
- gh-127599 [https://github.com/python/cpython/issues/127599]: Fix statistics for increments of object reference counts (in particular, when a reference count was increased by more than 1 in a single operation).
- gh-127651 [https://github.com/python/cpython/issues/127651]: When raising **`ImportError`** for missing symbols in `from` imports, use `__file__` in the error message if `__spec__.origin` is not a location
- gh-127582 [https://github.com/python/cpython/issues/127582]: Fix non-thread-safe object resurrection when calling finalizers and watcher callbacks in the free threading build.
- gh-127434 [https://github.com/python/cpython/issues/127434]: The iOS compiler shims can now accept arguments with spaces.
- gh-127536 [https://github.com/python/cpython/issues/127536]: Add missing locks around some list assignment operations in the free threading build.
- gh-127085 [https://github.com/python/cpython/issues/127085]: Fix race when exporting a buffer from a **`memoryview`** object on the free-threaded build.
- gh-127238 [https://github.com/python/cpython/issues/127238]: Correct error message for **`sys.set_int_max_str_digits()`**.
- gh-113841 [https://github.com/python/cpython/issues/113841]: Fix possible undefined behavior division by zero in **`complex`**’s **`_Py_c_pow()`**.

Library

- gh-127133 [https://github.com/python/cpython/issues/127133]: Calling `argparse.ArgumentParser.add_argument_group()` on an argument group, and calling `argparse.ArgumentParser.add_argument_group()` or `argparse.ArgumentParser.add_mutually_exclusive_group()` on a mutually exclusive group now raise exceptions. This nesting was never supported, often failed to work correctly, and was unintentionally exposed through inheritance. This functionality has been deprecated since Python 3.11.

Core and Builtins

- gh-126491 [https://github.com/python/cpython/issues/126491]: Add a marking phase to the GC. All objects that can be transitively reached from builtin modules or the stacks are marked as reachable before cycle detection. This reduces the amount of work done by the GC by approximately half.
- gh-127020 [https://github.com/python/cpython/issues/127020]: Fix a crash in the free threading build when `PyCode_GetCode()`, `PyCode_GetVarnames()`, `PyCode_GetCellvars()`, or `PyCode_GetFreevars()` were called from multiple threads at the same time.
- gh-127010 [https://github.com/python/cpython/issues/127010]: Simplify GC tracking of dictionaries. All dictionaries are tracked when created, rather than being lazily tracked when a trackable object was added to them. This simplifies the code considerably and results in a slight speedup.
- gh-126980 [https://github.com/python/cpython/issues/126980]: Fix `__buffer__()` of `bytearray` crashing when `READ` or `WRITE` are passed as flags.
- gh-126937 [https://github.com/python/cpython/issues/126937]: Fix `TypeError` when a `ctypes.Structure` has a field size that

doesn’t fit into an unsigned 16-bit integer. Instead, the maximum number of *bits* is `sys.maxsize`.

- gh-126868 [https://github.com/python/cpython/issues/126868]: Increase performance of `int` by adding a freelist for compact ints.
- gh-126881 [https://github.com/python/cpython/issues/126881]: Fix crash in finalization of dtoa state. Patch by Kumar Aditya.
- gh-126892 [https://github.com/python/cpython/issues/126892]: Require cold or invalidated code to “warm up” before being JIT compiled again.
- gh-126091 [https://github.com/python/cpython/issues/126091]: Ensure stack traces are complete when throwing into a generator chain that ends in a custom generator.
- gh-126024 [https://github.com/python/cpython/issues/126024]: Optimize decoding of short UTF-8 sequences containing non-ASCII characters by approximately 15%.
- gh-125420 [https://github.com/python/cpython/issues/125420]: Add `memoryview.index()` to `memoryview` objects. Patch by Bénédikt Tran.
- gh-125420 [https://github.com/python/cpython/issues/125420]: Add `memoryview.count()` to `memoryview` objects. Patch by Bénédikt Tran.
- gh-124470 [https://github.com/python/cpython/issues/124470]: Fix crash in free-threaded builds when replacing object dictionary while reading attribute on another thread
- gh-69639 [https://github.com/python/cpython/issues/69639]: Implement mixed-mode arithmetic rules combining real and complex numbers as specified by C standards since C99. Patch by Sergey B Kirpichev.
- gh-120010 [https://github.com/python/cpython/issues/120010]: Correct invalid corner cases which resulted in `(nan+nanj)` output in complex multiplication, e.g., `(1e300+1j)*(nan+infj)`. Patch by Sergey B Kirpichev.
- gh-109746 [https://github.com/python/cpython/issues/109746]: If **`_thread.start_new_thread()`** fails to start a new thread, it

deletes its state from interpreter and thus avoids its repeated cleanup on finalization.

C API

- gh-127896 [https://github.com/python/cpython/issues/127896]: The previously undocumented function **`PySequence_In()`** is soft deprecated. Use **`PySequence_Contains()`** instead.

- gh-127791 [https://github.com/python/cpython/issues/127791]: Fix loss of callbacks after more than one call to **`PyUnstable_AtExit()`**.

- gh-127691 [https://github.com/python/cpython/issues/127691]: The Unicode Exception Objects C API now raises a **`TypeError`** if its exception argument is not a **`UnicodeError`** object. Patch by Bénédikt Tran.

- gh-123378 [https://github.com/python/cpython/issues/123378]: Ensure that the value of **`UnicodeEncodeError.end`** retrieved by **`PyUnicodeEncodeError_GetEnd()`** lies in `[min(1, objlen), max(min(1, objlen), objlen)]` where *objlen* is the length of **`UnicodeEncodeError.object`**. Similar arguments apply to **`UnicodeDecodeError`** and **`UnicodeTranslateError`** and their corresponding C interface. Patch by Bénédikt Tran.

- gh-127314 [https://github.com/python/cpython/issues/127314]: Improve error message when calling the C API without an active thread state on the free-threaded build.

- gh-123378 [https://github.com/python/cpython/issues/123378]: Ensure that the value of **`UnicodeEncodeError.start`** retrieved by **`PyUnicodeEncodeError_GetStart()`** lies in `[0, max(0, objlen - 1)]` where *objlen* is the length of **`UnicodeEncodeError.object`**. Similar arguments apply to

`UnicodeDecodeError` and `UnicodeTranslateError` and their corresponding C interface. Patch by Bénédikt Tran.

- gh-109523 [https://github.com/python/cpython/issues/109523]: Reading text from a non-blocking stream with `read` may now raise a `BlockingIOError` if the operation cannot immediately return bytes.
- gh-102471 [https://github.com/python/cpython/issues/102471]: Add a new import and export API for Python `int` objects (**PEP 757** [https://peps.python.org/pep-0757/]):

  - `PyLong_GetNativeLayout()`;
  - `PyLong_Export()`;
  - `PyLong_FreeExport()`;
  - `PyLongWriter_Create()`;
  - `PyLongWriter_Finish()`;
  - `PyLongWriter_Discard()`.

  Patch by Victor Stinner.
- gh-121058 [https://github.com/python/cpython/issues/121058]: `PyThreadState_Clear()` now warns (and calls `sys.excepthook`) if the thread state still has an active exception.

Build

- gh-127865 [https://github.com/python/cpython/issues/127865]: Fix build failure on systems without thread-locals support.
- gh-127629 [https://github.com/python/cpython/issues/127629]: Emscripten builds now include ctypes support.
- gh-127111 [https://github.com/python/cpython/issues/127111]: Updated the Emscripten web example to use ES6 modules and be built into a distinct `web_example` subfolder.

- gh-115869 [https://github.com/python/cpython/issues/115869]: Make `jit_stencils.h` (which is produced during JIT builds) reproducible.
- gh-126898 [https://github.com/python/cpython/issues/126898]: The Emscripten build of Python is now based on ES6 modules.

Python 3.14.0 alpha 2

Release date: 2024-11-19

Windows

- gh-126911 [https://github.com/python/cpython/issues/126911]: Update credits command output.
- gh-118973 [https://github.com/python/cpython/issues/118973]: Ensures the experimental free-threaded install includes the `_tkinter` module. The optional Tcl/Tk component must also be installed in order for the module to work.
- gh-126497 [https://github.com/python/cpython/issues/126497]: Fixes venv failure due to missing redirector executables in experimental free-threaded installs.
- gh-126074 [https://github.com/python/cpython/issues/126074]: Removed unnecessary DLLs from Windows embeddable package
- gh-125315 [https://github.com/python/cpython/issues/125315]: Avoid crashing in **`platform`** due to slow WMI calls on some Windows machines.
- gh-126084 [https://github.com/python/cpython/issues/126084]: Fix venvwlauncher to launch pythonw instead of python so no extra console window is created.
- gh-125842 [https://github.com/python/cpython/issues/125842]: Fix a **`SystemError`** when **`sys.exit()`** is called with `0xffffffff` on Windows.

- gh-125550 [https://github.com/python/cpython/issues/125550]: Enable the Python install manager to detect Python 3.14 installs from the Windows Store.
- gh-123803 [https://github.com/python/cpython/issues/123803]: All Windows code pages are now supported as “cpXXX” codecs on Windows.

Tools/Demos

- gh-126807 [https://github.com/python/cpython/issues/126807]: Fix extraction warnings in **pygettext.py** caused by mistaking function definitions for function calls.
- gh-126167 [https://github.com/python/cpython/issues/126167]: The iOS testbed was modified so that it can be used by third-party projects for testing purposes.

Tests

- gh-126909 [https://github.com/python/cpython/issues/126909]: Fix test_os extended attribute tests to work on filesystems with 1 KiB xattr size limit.
- gh-125730 [https://github.com/python/cpython/issues/125730]: Change `make test` to not run GUI tests by default. Use `make ci` to run tests with GUI tests instead.
- gh-124295 [https://github.com/python/cpython/issues/124295]: Add translation tests to the **`argparse`** module.

Security

- gh-126623 [https://github.com/python/cpython/issues/126623]: Upgrade libexpat to 2.6.4

Library

- gh-85957 [https://github.com/python/cpython/issues/85957]: Add missing MIME types for images with RFCs: emf, fits, g3fax, jp2, jpm, jpx, t38, tiff-fx and wmf. Patch by Hugo van Kemenade.

- gh-126920 [https://github.com/python/cpython/issues/126920]: Fix the `prefix` and `exec_prefix` keys from **`sysconfig.get_config_vars()`** incorrectly having the same value as **`sys.base_prefix`** and **`sys.base_exec_prefix`**, respectively, inside virtual environments. They now accurately reflect **`sys.prefix`** and **`sys.exec_prefix`**.

- gh-67877 [https://github.com/python/cpython/issues/67877]: Fix memory leaks when **`regular expression`** matching terminates abruptly, either because of a signal or because memory allocation fails.

- gh-125063 [https://github.com/python/cpython/issues/125063]: **`marshal`** now supports **`slice`** objects. The marshal format version was increased to 5.

- gh-126789 [https://github.com/python/cpython/issues/126789]: Fixed the values of **`sysconfig.get_config_vars()`**, **`sysconfig.get_paths()`**, and their siblings when the **`site`** initialization happens after **`sysconfig`** has built a cache for **`sysconfig.get_config_vars()`**.

- gh-126188 [https://github.com/python/cpython/issues/126188]: Update bundled pip to 24.3.1

- gh-126766 [https://github.com/python/cpython/issues/126766]: Fix issue where **`urllib.request.url2pathname()`** failed to discard two leading slashes introducing an empty authority section.

- gh-126705 [https://github.com/python/cpython/issues/126705]: Allow **`os.PathLike`** to be a base for Protocols.

- gh-126699 [https://github.com/python/cpython/issues/126699]: Allow `collections.abc.AsyncIterator` to be a base for Protocols.
- gh-126654 [https://github.com/python/cpython/issues/126654]: Fix crash when non-dict was passed to several functions in `_interpreters` module.
- gh-104745 [https://github.com/python/cpython/issues/104745]: Limit starting a patcher (from `unittest.mock.patch()` or `unittest.mock.patch.object()`) more than once without stopping it
- gh-126595 [https://github.com/python/cpython/issues/126595]: Fix a crash when instantiating `itertools.count` with an initial count of `sys.maxsize` on debug builds. Patch by Bénédikt Tran.
- gh-120423 [https://github.com/python/cpython/issues/120423]: Fix issue where `urllib.request.pathname2url()` mishandled Windows paths with embedded forward slashes.
- gh-126565 [https://github.com/python/cpython/issues/126565]: Improve performances of `zipfile.Path.open()` for non-reading modes.
- gh-126505 [https://github.com/python/cpython/issues/126505]: Fix bugs in compiling case-insensitive `regular expressions` with character classes containing non-BMP characters: upper-case non-BMP character did was ignored and the ASCII flag was ignored when matching a character range whose upper bound is beyond the BMP region.
- gh-117378 [https://github.com/python/cpython/issues/117378]: Fixed the `multiprocessing` `"forkserver"` start method forkserver process to correctly inherit the parent’s `sys.path` during the importing of `multiprocessing.set_forkserver_preload()`

modules in the same manner as `sys.path` is configured in workers before executing work items.

This bug caused some forkserver module preloading to silently fail to preload. This manifested as a performance degradation in child processes when the `sys.path` was required due to additional repeated work in every worker.

It could also have a side effect of `""` remaining in `sys.path` during forkserver preload imports instead of the absolute path from `os.getcwd()` at multiprocessing import time used in the worker `sys.path`.

The `sys.path` differences between phases in the child process could potentially have caused preload to import incorrect things from the wrong location. We are unaware of that actually having happened in practice.

- gh-125679 [https://github.com/python/cpython/issues/125679]: The `multiprocessing.Lock` and `multiprocessing.RLock` `repr` values no longer say “unknown” on macOS.
- gh-126476 [https://github.com/python/cpython/issues/126476]: Raise `calendar.IllegalMonthError` (now a subclass of `IndexError`) for `calendar.month()` when the input month is not correct.
- gh-126489 [https://github.com/python/cpython/issues/126489]: The Python implementation of `pickle` no longer calls `pickle.Pickler.persistent_id()` for the result of **`persistent_id()`**.
- gh-126451 [https://github.com/python/cpython/issues/126451]: Register the `contextvars.Context` type to `collections.abc.Mapping`.

- gh-126175 [https://github.com/python/cpython/issues/126175]: Add `msg`, `doc`, `pos`, `lineno` and `colno` attributes to **`tomllib.TOMLDecodeError`**. Deprecate instantiating with free-form arguments.

- gh-89416 [https://github.com/python/cpython/issues/89416]: Add **RFC 9559** [https://datatracker.ietf.org/doc/html/rfc9559.html] MIME types for Matroska audiovisual container formats. Patch by Hugo van Kemenade.

- gh-126417 [https://github.com/python/cpython/issues/126417]: Register the **`multiprocessing.managers.DictProxy`** and **`multiprocessing.managers.ListProxy`** types in **`multiprocessing.managers`** to **`collections.abc.MutableMapping`** and **`collections.abc.MutableSequence`**, respectively.

- gh-126390 [https://github.com/python/cpython/issues/126390]: Add support for returning intermixed options and non-option arguments in order in **`getopt.gnu_getopt()`**.

- gh-126374 [https://github.com/python/cpython/issues/126374]: Add support for options with optional arguments in the **`getopt`** module.

- gh-126363 [https://github.com/python/cpython/issues/126363]: Speed up pattern parsing in **`pathlib.Path.glob()`** by skipping creation of a **`pathlib.Path`** object for the pattern.

- gh-126353 [https://github.com/python/cpython/issues/126353]: **`asyncio.get_event_loop()`** now does not implicitly creates an event loop. It now raises a **`RuntimeError`** if there is no set event loop. Patch by Kumar Aditya.

- gh-126313 [https://github.com/python/cpython/issues/126313]: Fix an issue in **`curses.napms()`** when **`curses.initscr()`** has not yet been called. Patch by Bénédikt Tran.

- gh-126303 [https://github.com/python/cpython/issues/126303]: Fix pickling and copying of `os.sched_param` objects.
- gh-126138 [https://github.com/python/cpython/issues/126138]: Fix a use-after-free crash on `asyncio.Task` objects whose underlying coroutine yields an object that implements an evil `__getattribute__()`. Patch by Nico Posada.
- gh-120057 [https://github.com/python/cpython/issues/120057]: Replace the `os.environ.refresh()` method with a new `os.reload_environ()` function. Patch by Victor Stinner.
- gh-126220 [https://github.com/python/cpython/issues/126220]: Fix crash in `cProfile.Profile` and `_lsprof.Profiler` when their callbacks were directly called with 0 arguments.
- gh-126212 [https://github.com/python/cpython/issues/126212]: Fix issue where `urllib.request.pathname2url()` and `url2pathname()` removed slashes from Windows DOS drive paths and URLs.
- gh-126223 [https://github.com/python/cpython/issues/126223]: Raise a `UnicodeEncodeError` instead of a `SystemError` upon calling `_interpreters.create()` with an invalid Unicode character.
- gh-126205 [https://github.com/python/cpython/issues/126205]: Fix issue where `urllib.request.pathname2url()` generated URLs beginning with four slashes (rather than two) when given a Windows UNC path.
- gh-126156 [https://github.com/python/cpython/issues/126156]: Improved performances of creating `Morsel` objects by a factor of 3.8x.
- gh-126105 [https://github.com/python/cpython/issues/126105]: Fix a crash in `ast` when the `ast.AST._fields` attribute is deleted.

- gh-126106 [https://github.com/python/cpython/issues/126106]: Fixes a possible `NULL` pointer dereference in `ssl`.

- gh-126080 [https://github.com/python/cpython/issues/126080]: Fix a use-after-free crash on `asyncio.Task` objects for which the underlying event loop implements an evil `__getattribute__()`. Reported by Nico-Posada. Patch by Bénédikt Tran.

- gh-125322 [https://github.com/python/cpython/issues/125322]: Correct detection of complex numbers support in libffi.

- gh-126083 [https://github.com/python/cpython/issues/126083]: Fixed a reference leak in `asyncio.Task` objects when reinitializing the same object with a non-`None` context. Patch by Nico Posada.

- gh-126068 [https://github.com/python/cpython/issues/126068]: Fix exceptions in the `argparse` module so that only error messages for ArgumentError and ArgumentTypeError are now translated. ArgumentError is now only used for command line errors, not for logical errors in the program. TypeError is now raised instead of ValueError for some logical errors.

- gh-125413 [https://github.com/python/cpython/issues/125413]: Add **`pathlib.Path.scandir()`** method to efficiently fetch directory children and their file attributes. This is a trivial wrapper of `os.scandir()`.

- gh-125984 [https://github.com/python/cpython/issues/125984]: Fix use-after-free crashes on `asyncio.Future` objects for which the underlying event loop implements an evil `__getattribute__()`. Reported by Nico-Posada. Patch by Bénédikt Tran.

- gh-125926 [https://github.com/python/cpython/issues/125926]: Fix `urllib.parse.urljoin()` for base URI with undefined authority. Although **RFC 3986** [https://datatracker.ietf.org/doc/html/rfc3986.html] only specify reference resolution for absolute base URI, `urljoin()` should continue to return sensible result for relative base URI.

- gh-125969 [https://github.com/python/cpython/issues/125969]: Fix an out-of-bounds crash when an evil `asyncio.loop.call_soon()` mutates the length of the internal callbacks list. Patch by Bénédikt Tran.

- gh-125966 [https://github.com/python/cpython/issues/125966]: Fix a use-after-free crash in `asyncio.Future.remove_done_callback()`. Patch by Bénédikt Tran.

- gh-125789 [https://github.com/python/cpython/issues/125789]: Fix possible crash when mutating list of callbacks returned by `asyncio.Future._callbacks`. It now always returns a new copy in C implementation `_asyncio`. Patch by Kumar Aditya.

- gh-126916 [https://github.com/python/cpython/issues/126916]: Allow the *initial* parameter of `functools.reduce()` to be passed as a keyword argument. Patch by Sayandip Dutta.

- gh-124452 [https://github.com/python/cpython/issues/124452]: Fix an issue in `email.policy.EmailPolicy.header_source_parse()` and `email.policy.Compat32.header_source_parse()` that introduced spurious leading whitespaces into header values when the header includes a newline character after the header name delimiter (`:`) and before the value.

- gh-117941 [https://github.com/python/cpython/issues/117941]: **`argparse.BooleanOptionalAction`** now rejects option names starting with `--no-`.

- gh-125884 [https://github.com/python/cpython/issues/125884]: Fixed the bug for **`pdb`** where it can't set breakpoints on functions with certain annotations.

- gh-125355 [https://github.com/python/cpython/issues/125355]: Fix several bugs in **`argparse.ArgumentParser.parse_intermixed_args()`**.
  - The parser no longer changes temporarily during parsing.
  - Default values are not processed twice.
  - Required mutually exclusive groups containing positional arguments are now supported.
  - The missing arguments report now includes the names of all required optional and positional arguments.
  - Unknown options can be intermixed with positional arguments in parse_known_intermixed_args().

- gh-125767 [https://github.com/python/cpython/issues/125767]: **`super`** objects are now **`pickleable`** and **`copyable`**.

- gh-124969 [https://github.com/python/cpython/issues/124969]: `locale.nl_langinfo(locale.ALT_DIGITS)` now returns a string again. The returned value consists of up to 100 semicolon-separated symbols.

- gh-84850 [https://github.com/python/cpython/issues/84850]: Remove **`URLopener`** and **`FancyURLopener`** classes from **`urllib.request`**. They had previously raised **`DeprecationWarning`** since Python 3.3.

- gh-125666 [https://github.com/python/cpython/issues/125666]: Avoid the exiting the interpreter if a null byte is given as input in the new REPL.

- gh-125710 [https://github.com/python/cpython/issues/125710]: [Enum] fix hashable<->nonhashable comparisons for member values

- gh-125631 [https://github.com/python/cpython/issues/125631]: Restore ability to set `persistent_id` and `persistent_load` attributes of instances of the `Pickler` and `Unpickler` classes in the `pickle` module.

- gh-125378 [https://github.com/python/cpython/issues/125378]: Fixed the bug in `pdb` where after a multi-line command, an empty line repeats the first line of the multi-line command, instead of the full command.

- gh-125682 [https://github.com/python/cpython/issues/125682]: Reject non-ASCII digits in the Python implementation of `json.loads()` conforming to the JSON specification.

- gh-125660 [https://github.com/python/cpython/issues/125660]: Reject invalid unicode escapes for Python implementation of `json.loads()`.

- gh-52551 [https://github.com/python/cpython/issues/52551]: Use `wcsftime()` to implement `time.strftime()` on Windows.

- gh-125259 [https://github.com/python/cpython/issues/125259]: Fix the notes removal logic for errors thrown in enum initialization.

- gh-125633 [https://github.com/python/cpython/issues/125633]: Add function `inspect.ispackage()` to determine whether an object is a package or not.

- gh-125614 [https://github.com/python/cpython/issues/125614]: In the `FORWARDREF` format of `annotationlib`, fix bug where nested expressions were not returned as `annotationlib.ForwardRef` format.

- gh-125590 [https://github.com/python/cpython/issues/125590]: Allow `FrameLocalsProxy` to delete and pop if the key is not a fast variable.

- gh-125600 [https://github.com/python/cpython/issues/125600]: Only show stale code warning in `pdb` when we display source code.

- gh-125542 [https://github.com/python/cpython/issues/125542]: Deprecate passing keyword-only *prefix_chars* argument to `argparse.ArgumentParser.add_argument_group()`.

- gh-125541 [https://github.com/python/cpython/issues/125541]: Pressing `Ctrl-C` while blocked in `threading.Lock.acquire()`, `threading.RLock.acquire()`, and `threading.Thread.join()` now interrupts the function call and raises a `KeyboardInterrupt` exception on Windows, similar to how those functions behave on macOS and Linux.

- gh-125519 [https://github.com/python/cpython/issues/125519]: Improve traceback if `importlib.reload()` is called with an object that is not a module. Patch by Alex Waygood.

- gh-125451 [https://github.com/python/cpython/issues/125451]: Fix deadlock when `concurrent.futures.ProcessPoolExecutor` shuts down concurrently with an error when feeding a job to a worker process.

- gh-125115 [https://github.com/python/cpython/issues/125115]: Fixed a bug in `pdb` where arguments starting with `-` can’t be passed to the debugged script.

- gh-125398 [https://github.com/python/cpython/issues/125398]: Fix the conversion of the `VIRTUAL_ENV` path in the activate script in `venv` when running in Git Bash for Windows.

- gh-125245 [https://github.com/python/cpython/issues/125245]: Fix race condition when importing `collections.abc`, which could incorrectly return an empty module.

- gh-52551 [https://github.com/python/cpython/issues/52551]: Fix encoding issues in `time.strftime()`, the `strftime()` method of the `datetime` classes `datetime`, `date` and `time` and formatting of these classes. Characters not encodable in the current locale are now acceptable in the format string. Surrogate pairs and sequence of surrogatescape-encoded bytes are no longer recombinated. Embedded null character no longer terminates the format string.

- gh-124984 [https://github.com/python/cpython/issues/124984]: Fixed thread safety in `ssl` in the free-threaded build. OpenSSL operations are now protected by a per-object lock.

- gh-124651 [https://github.com/python/cpython/issues/124651]: Properly quote template strings in `venv` activation scripts.

- gh-124694 [https://github.com/python/cpython/issues/124694]: We've added `concurrent.futures.InterpreterPoolExecutor`, which allows you to run code in multiple isolated interpreters. This allows you to circumvent the limitations of CPU-bound threads (due to the GIL). Patch by Eric Snow.

  This addition is unrelated to **PEP 734** [https://peps.python.org/pep-0734/].

- gh-58032 [https://github.com/python/cpython/issues/58032]: Deprecate the `argparse.FileType` type converter.

- gh-99749 [https://github.com/python/cpython/issues/99749]: Adds a feature to optionally enable suggestions for argument choices and subparser names if mistyped by the user.

- gh-58956 [https://github.com/python/cpython/issues/58956]: Fixed a bug in `pdb` where sometimes the breakpoint won’t trigger if it was set on a function which is already in the call stack.

- gh-124111 [https://github.com/python/cpython/issues/124111]: The tkinter module can now be built to use either the new version 9.0.0 of Tcl/Tk or the latest release 8.6.15 of Tcl/Tk 8. Tcl/Tk 9 includes many improvements, both to the Tcl language and to the appearance and utility of the graphical user interface provided by Tk.

- gh-80958 [https://github.com/python/cpython/issues/80958]: unittest discovery supports PEP 420 namespace packages as start directory again.

- gh-123370 [https://github.com/python/cpython/issues/123370]: Fix the canvas not clearing after running turtledemo clock.

- gh-89083 [https://github.com/python/cpython/issues/89083]: Add `uuid.uuid8()` for generating UUIDv8 objects as specified in **RFC 9562** [https://datatracker.ietf.org/doc/html/rfc9562.html]. Patch by Bénédikt Tran

- gh-122549 [https://github.com/python/cpython/issues/122549]: Add `platform.invalidate_caches()` to invalidate cached results.

- gh-120754 [https://github.com/python/cpython/issues/120754]: Update unbounded `read` calls in `zipfile` to specify an explicit `size` putting a limit on how much data they may read. This also updates handling around ZIP max comment size to match the standard instead of reading comments that are one byte too long.

- gh-121267 [https://github.com/python/cpython/issues/121267]: Improve the performance of `tarfile` when writing files, by caching user names and group names.

- gh-70764 [https://github.com/python/cpython/issues/70764]: Fixed an issue where `inspect.getclosurevars()` would incorrectly classify an attribute name as a global variable when the name exists both as an attribute name and a global variable.

- gh-118289 [https://github.com/python/cpython/issues/118289]: **`posixpath.realpath()`** now raises `NotADirectoryError` when *strict* mode is enabled and a non-directory path with a trailing slash is supplied.

- gh-119826 [https://github.com/python/cpython/issues/119826]: Always return an absolute path for `os.path.abspath()` on Windows.

- gh-97850 [https://github.com/python/cpython/issues/97850]: Remove deprecated **`pkgutil.get_loader()`** and **`pkgutil.find_loader()`**.

- gh-118986 [https://github.com/python/cpython/issues/118986]: Add **`socket.IPV6_RECVERR`** constant (available since Linux 2.2).

- gh-116897 [https://github.com/python/cpython/issues/116897]: Accepting objects with false values (like `0` and `[]`) except empty strings, byte-like objects and `None` in `urllib.parse` functions `parse_qsl()` and `parse_qs()` is now deprecated.

- gh-101955 [https://github.com/python/cpython/issues/101955]: Fix SystemError when match regular expression pattern containing some combination of possessive quantifier, alternative and capture group.

- gh-71936 [https://github.com/python/cpython/issues/71936]: Fix a race condition in `multiprocessing.pool.Pool`.

- bpo-46128 [https://bugs.python.org/issue?@action=redirect&bpo=46128]: Strip `unittest.IsolatedAsyncioTestCase` stack frames from reported stacktraces.

- gh-84852 [https://github.com/python/cpython/issues/84852]: Add MIME types for MS Embedded OpenType, OpenType Layout, TrueType, WOFF 1.0 and 2.0 fonts. Patch by Sahil Prajapati and Hugo van Kemenade.

Documentation

- gh-126622 [https://github.com/python/cpython/issues/126622]: Added stub pages for removed modules explaining their removal, where to find replacements, and linking to the last Python version that supported them. Contributed by Ned Batchelder.
- gh-125277 [https://github.com/python/cpython/issues/125277]: Require Sphinx 7.2.6 or later to build the Python documentation. Patch by Adam Turner.
- gh-60712 [https://github.com/python/cpython/issues/60712]: Include the `object` type in the lists of documented types. Change by Furkan Onder and Martin Panter.

Core and Builtins

- gh-126795 [https://github.com/python/cpython/issues/126795]: Increase the threshold for JIT code warmup. Depending on platform and workload, this can result in performance gains of 1-9% and memory savings of 3-5%.
- gh-126341 [https://github.com/python/cpython/issues/126341]: Now `ValueError` is raised instead of `SystemError` when trying to

iterate over a released `memoryview` object.
- gh-126688 [https://github.com/python/cpython/issues/126688]: Fix a crash when calling `os.fork()` on some operating systems, including SerenityOS.

Library

- gh-126066 [https://github.com/python/cpython/issues/126066]: Fix `importlib` to not write an incomplete .pyc files when a ulimit or some other operating system mechanism is preventing the write to go through fully.

Core and Builtins

- gh-126222 [https://github.com/python/cpython/issues/126222]: Do not include count of “peek” items in `_PyUop_num_popped`. This ensures that the correct number of items are popped from the stack when a micro-op exits with an error.
- gh-126366 [https://github.com/python/cpython/issues/126366]: Fix crash when using `yield from` on an object that raises an exception in its `__iter__`.

Library

- gh-126209 [https://github.com/python/cpython/issues/126209]: Fix an issue with `skip_file_prefixes` parameter which resulted in an inconsistent behaviour between the C and Python implementations of `warnings.warn()`. Patch by Daehee Kim.

Core and Builtins

- gh-126312 [https://github.com/python/cpython/issues/126312]: Fix crash during garbage collection on an object frozen by `gc.freeze()` on the free-threaded build.

- gh-103951 [https://github.com/python/cpython/issues/103951]: Relax optimization requirements to allow fast attribute access to module subclasses.
- gh-126072 [https://github.com/python/cpython/issues/126072]: Following gh-126101 [https://github.com/python/cpython/issues/126101], for Code Objects like lambda, annotation and type alias, we no longer add `None` to its **`co_consts`**.
- gh-126195 [https://github.com/python/cpython/issues/126195]: Improve JIT performance by 1.4% on macOS Apple Silicon by using platform-specific memory protection APIs. Patch by Diego Russo.
- gh-126139 [https://github.com/python/cpython/issues/126139]: Provide better error location when attempting to use a future statement with an unknown future feature.
- gh-126072 [https://github.com/python/cpython/issues/126072]: Add a new attribute in **`co_flags`** to indicate whether the first item in **`co_consts`** is the docstring. If a code object has no docstring, `None` will **NOT** be inserted.
- gh-126076 [https://github.com/python/cpython/issues/126076]: Relocated objects such as `tuple`, `bytes` and `str` objects are properly tracked by **`tracemalloc`** and its associated hooks. Patch by Pablo Galindo.
- gh-90370 [https://github.com/python/cpython/issues/90370]: Avoid temporary tuple creation for vararg in argument passing with Argument Clinic generated code (if arguments either vararg or positional-only).
- gh-126018 [https://github.com/python/cpython/issues/126018]: Fix a crash in **`sys.audit()`** when passing a non-string as first argument and Python was compiled in debug mode.
- gh-126012 [https://github.com/python/cpython/issues/126012]: The **`memoryview`** type now supports subscription, making it a generic type.
- gh-125837 [https://github.com/python/cpython/issues/125837]: Adds **`LOAD_SMALL_INT`** and **`LOAD_CONST_IMMORTAL`** instructions. `LOAD_SMALL_INT` pushes a small integer equal to the `oparg`

to the stack. `LOAD_CONST_IMMORTAL` does the same as `LOAD_CONST` but is more efficient for immortal objects. Removes `RETURN_CONST` instruction.

- gh-125942 [https://github.com/python/cpython/issues/125942]: On Android, the `errors` setting of **`sys.stdout`** was changed from `surrogateescape` to `backslashreplace`.
- gh-125859 [https://github.com/python/cpython/issues/125859]: Fix a crash in the free threading build when **`gc.get_objects()`** or **`gc.get_referrers()`** is called during an in-progress garbage collection.
- gh-125868 [https://github.com/python/cpython/issues/125868]: It was possible in 3.14.0a1 only for attribute lookup to give the wrong value. This was due to an incorrect specialization in very specific circumstances. This is fixed in 3.14.0a2.
- gh-125498 [https://github.com/python/cpython/issues/125498]: The JIT has been updated to leverage Clang 19’s new `preserve_none` attribute, which supports more platforms and is more useful than LLVM’s existing `ghccc` calling convention. This also removes the need to manually patch the calling convention in LLVM IR, simplifying the JIT compilation process.
- gh-125703 [https://github.com/python/cpython/issues/125703]: Correctly honour **`tracemalloc`** hooks in specialized `Py_DECREF` paths. Patch by Pablo Galindo
- gh-125593 [https://github.com/python/cpython/issues/125593]: Use color to highlight error locations in traceback from exception group
- gh-125017 [https://github.com/python/cpython/issues/125017]: Fix crash on certain accesses to the `__annotations__` of **`staticmethod`** and **`classmethod`** objects.
- gh-125588 [https://github.com/python/cpython/issues/125588]: The Python PEG generator can now use f-strings in the grammar actions. Patch by Pablo Galindo
- gh-125444 [https://github.com/python/cpython/issues/125444]: Fix illegal instruction for older Arm architectures. Patch by

Diego Russo, testing by Ross Burton.

- gh-118423 [https://github.com/python/cpython/issues/118423]: Add a new `INSTRUCTION_SIZE` macro to the cases generator which returns the current instruction size.
- gh-125038 [https://github.com/python/cpython/issues/125038]: Fix crash when iterating over a generator expression after direct changes on `gi_frame.f_locals`. Patch by Mikhail Efimov.
- gh-124855 [https://github.com/python/cpython/issues/124855]: Don’t allow the JIT and perf support to be active at the same time. Patch by Pablo Galindo
- gh-123714 [https://github.com/python/cpython/issues/123714]: Update JIT compilation to use LLVM 19
- gh-123930 [https://github.com/python/cpython/issues/123930]: Improve the error message when a script shadowing a module from the standard library causes **`ImportError`** to be raised during a “from” import. Similarly, improve the error message when a script shadowing a third party module attempts to “from” import an attribute from that third party module while still initialising.
- gh-119793 [https://github.com/python/cpython/issues/119793]: The **`map()`** built-in now has an optional keyword-only *strict* flag like **`zip()`** to check that all the iterables are of equal length. Patch by Wannes Boeykens.

Library

- gh-118950 [https://github.com/python/cpython/issues/118950]: Fix bug where SSLProtocol.connection_lost wasn’t getting called when OSError was thrown on writing to socket.
- gh-113570 [https://github.com/python/cpython/issues/113570]: Fixed a bug in `reprlib.repr` where it incorrectly called the repr method on shadowed Python built-in types.

C API

- gh-126554 [https://github.com/python/cpython/issues/126554]: Fix error handling in `ctypes.CDLL` objects which could result in a crash in rare situations.
- gh-126061 [https://github.com/python/cpython/issues/126061]: Add `PyLong_IsPositive()`, `PyLong_IsNegative()` and `PyLong_IsZero()` for checking if a `PyLongObject` is positive, negative, or zero, respectively.
- gh-125608 [https://github.com/python/cpython/issues/125608]: Fix a bug where dictionary watchers (e.g., `PyDict_Watch()`) on an object's attribute dictionary (`__dict__`) were not triggered when the object's attributes were modified.
- gh-123619 [https://github.com/python/cpython/issues/123619]: Added the `PyUnstable_Object_EnableDeferredRefcount()` function for enabling **PEP 703** [https://peps.python.org/pep-0703/] deferred reference counting.
- gh-121654 [https://github.com/python/cpython/issues/121654]: Add `PyType_Freeze()` function to make a type immutable. Patch by Victor Stinner.
- gh-120026 [https://github.com/python/cpython/issues/120026]: The `Py_HUGE_VAL` macro is soft deprecated.

Build

- gh-126691 [https://github.com/python/cpython/issues/126691]: Removed the `--with-emscripten-target` configure flag. We unified the `node` and `browser` options and the same build can now be used, independent of target runtime.
- gh-123877 [https://github.com/python/cpython/issues/123877]: Use `wasm32-wasip1` as the target triple for WASI instead of `wasm32-wasi`. The latter will eventually be reclaimed for WASI 1.0 while CPython currently only supports WASI preview1.
- gh-126458 [https://github.com/python/cpython/issues/126458]: Disable SIMD support for HACL under WASI.
- gh-89640 [https://github.com/python/cpython/issues/89640]: Hard-code float word ordering as little endian on WASM.

- gh-126206 [https://github.com/python/cpython/issues/126206]: `make clinic` now runs Argument Clinic using the `--force` option, thus forcefully regenerating generated code.
- gh-126187 [https://github.com/python/cpython/issues/126187]: Introduced `Tools/wasm/emscripten.py` to simplify doing Emscripten builds.
- gh-124932 [https://github.com/python/cpython/issues/124932]: For cross builds, there is now support for having a different install `prefix` than the `host_prefix` used by `getpath.py`. This is set to `/` by default for Emscripten, on other platforms the default behavior is the same as before.
- gh-125946 [https://github.com/python/cpython/issues/125946]: The minimum supported Android version is now 7.0 (API level 24).
- gh-125940 [https://github.com/python/cpython/issues/125940]: The Android build now supports 16 KB page sizes [https://developer.android.com/guide/practices/page-sizes].
- gh-89640 [https://github.com/python/cpython/issues/89640]: Improve detection of float word ordering on Linux when link-time optimizations are enabled.
- gh-124928 [https://github.com/python/cpython/issues/124928]: Emscripten builds now require node >= 18.
- gh-115382 [https://github.com/python/cpython/issues/115382]: Fix cross compile failures when the host and target SOABIs match.

Python 3.14.0 alpha 1

Release date: 2024-10-15

macOS

- gh-124448 [https://github.com/python/cpython/issues/124448]: Update bundled Tcl/Tk in macOS installer to 8.6.15.

- gh-123797 [https://github.com/python/cpython/issues/123797]: Check for runtime availability of `ptsname_r` function on macos.
- gh-123418 [https://github.com/python/cpython/issues/123418]: Updated macOS installer build to use OpenSSL 3.0.15.

Windows

- gh-124487 [https://github.com/python/cpython/issues/124487]: Increases Windows required OS and API level to Windows 10.
- gh-124609 [https://github.com/python/cpython/issues/124609]: Fix `_Py_ThreadId` for Windows builds using MinGW. Patch by Tony Roberts.
- gh-124448 [https://github.com/python/cpython/issues/124448]: Updated bundled Tcl/Tk to 8.6.15.
- gh-124254 [https://github.com/python/cpython/issues/124254]: Ensures experimental free-threaded binaries remain installed when updating.
- gh-123915 [https://github.com/python/cpython/issues/123915]: Ensure that `Tools\msi\buildrelease.bat` uses different directories for AMD64 and ARM64 builds.
- gh-123418 [https://github.com/python/cpython/issues/123418]: Updated Windows build to use OpenSSL 3.0.15.
- gh-123476 [https://github.com/python/cpython/issues/123476]: Add support for `socket.TCP_QUICKACK` on Windows platforms.
- gh-122573 [https://github.com/python/cpython/issues/122573]: The Windows build of CPython now requires 3.10 or newer.
- gh-100256 [https://github.com/python/cpython/issues/100256]: **`mimetypes`** no longer fails when it encounters an inaccessible registry key.
- gh-119679 [https://github.com/python/cpython/issues/119679]: Ensures correct import libraries are included in Windows installs.
- gh-119690 [https://github.com/python/cpython/issues/119690]: Adds Unicode support and fixes audit events for

`_winapi.CreateNamedPipe`.
- gh-111201 [https://github.com/python/cpython/issues/111201]: Add support for new pyrepl on Windows
- gh-119070 [https://github.com/python/cpython/issues/119070]: Fixes `py.exe` handling of shebangs like `/usr/bin/env python3.12`, which were previously interpreted as `python3.exe` instead of `python3.12.exe`.
- gh-117505 [https://github.com/python/cpython/issues/117505]: Fixes an issue with the Windows installer not running ensurepip in a fully isolated environment. This could cause unexpected interactions with the user site-packages.
- gh-118209 [https://github.com/python/cpython/issues/118209]: Avoid crashing in **`mmap`** on Windows when the mapped memory is inaccessible due to file system errors or access violations.
- gh-79846 [https://github.com/python/cpython/issues/79846]: Makes `ssl.create_default_context()` ignore invalid certificates in the Windows certificate store

Tools/Demos

- gh-123418 [https://github.com/python/cpython/issues/123418]: Update GitHub CI workflows to use OpenSSL 3.0.15 and multissltests to use 3.0.15, 3.1.7, and 3.2.3.

Tests

- gh-125041 [https://github.com/python/cpython/issues/125041]: Re-enable skipped tests for **`zlib`** on the s390x architecture: only skip checks of the compressed bytes, which can be different between zlib’s software implementation and the hardware-accelerated implementation.
- gh-124378 [https://github.com/python/cpython/issues/124378]: Updated `test_ttk` to pass with Tcl/Tk 8.6.15.

- gh-124213 [https://github.com/python/cpython/issues/124213]: Detect whether the test suite is running inside a systemd-nspawn container with `--suppress-sync=true` option, and skip the `test_os` and `test_mmap` tests that are failing in this scenario.
- gh-124190 [https://github.com/python/cpython/issues/124190]: Add capability to ignore entire files or directories in check warning CI tool
- gh-121921 [https://github.com/python/cpython/issues/121921]: Update `Lib/test/crashers/bogus_code_obj.py` so that it crashes properly again.
- gh-112301 [https://github.com/python/cpython/issues/112301]: Add tooling to check for changes in compiler warnings. Patch by Nate Ohlson.
- gh-59022 [https://github.com/python/cpython/issues/59022]: Add tests for **`pkgutil.extend_path()`**. Patch by Andreas Stocker.
- gh-99242 [https://github.com/python/cpython/issues/99242]: **`os.getloadavg()`** may throw **`OSError`** when running regression tests under certain conditions (e.g. chroot). This error is now caught and ignored, since reporting load average is optional.
- gh-121084 [https://github.com/python/cpython/issues/121084]: Fix test_typing random leaks. Clear typing ABC caches when running tests for refleaks (`-R` option): call `_abc_caches_clear()` on typing abstract classes and their subclasses. Patch by Victor Stinner.
- gh-121160 [https://github.com/python/cpython/issues/121160]: Add a test for **`readline.set_history_length()`**. Note that this test may fail on readline libraries.
- gh-121200 [https://github.com/python/cpython/issues/121200]: Fix `test_expanduser_pwd2()` of `test_posixpath`. Call `getpwnam()` to get `pw_dir`, since it can be different than `getpwall()` `pw_dir`. Patch by Victor Stinner.
- gh-121188 [https://github.com/python/cpython/issues/121188]: When creating the JUnit XML file, regrtest now escapes characters which are invalid in XML, such as the chr(27)

control character used in ANSI escape sequences. Patch by Victor Stinner.

- gh-120801 [https://github.com/python/cpython/issues/120801]: Cleaned up fixtures for importlib.metadata tests and consolidated behavior with ‘test.support.os_helper’.
- gh-119727 [https://github.com/python/cpython/issues/119727]: Add `--single-process` command line option to Python test runner (regrtest). Patch by Victor Stinner.
- gh-119273 [https://github.com/python/cpython/issues/119273]: Python test runner no longer runs tests using TTY (ex: test_ioctl) in a process group (using `setsid()`). Previously, tests using TTY were skipped. Patch by Victor Stinner.
- gh-119050 [https://github.com/python/cpython/issues/119050]: regrtest test runner: Add XML support to the refleak checker (-R option). Patch by Victor Stinner.
- gh-101525 [https://github.com/python/cpython/issues/101525]: Skip `test_gdb` if the binary is relocated by BOLT. Patch by Donghee Na.
- gh-107562 [https://github.com/python/cpython/issues/107562]: Test certificates have been updated to expire far in the future. This allows testing Y2038 with system time set to after that, so that actual Y2038 issues can be exposed, and not masked by expired certificate errors.

Security

- gh-125140 [https://github.com/python/cpython/issues/125140]: Remove the current directory from `sys.path` when using PyREPL.

- gh-123678 [https://github.com/python/cpython/issues/123678]: Upgrade libexpat to 2.6.3

- gh-112301 [https://github.com/python/cpython/issues/112301]: Enable compiler options that warn of potential security vulnerabilities.

- gh-122792 [https://github.com/python/cpython/issues/122792]: Changed IPv4-mapped `ipaddress.IPv6Address` to consistently use the mapped IPv4 address value for deciding properties. Properties which have their behavior fixed are `is_multicast`, `is_reserved`, `is_link_local`, `is_global`, and `is_unspecified`.

- gh-112301 [https://github.com/python/cpython/issues/112301]: Add ability to ignore warnings per file with warning count in warning checking tooling. Patch by Nate Ohlson.

- gh-112301 [https://github.com/python/cpython/issues/112301]: Add macOS warning tracking to warning check tooling. Patch by Nate Ohlson.

- gh-122133 [https://github.com/python/cpython/issues/122133]: Authenticate the socket connection for the `socket.socketpair()` fallback on platforms where `AF_UNIX` is not available like Windows.

  Patch by Gregory P. Smith <greg@krypto.org> and Seth Larson <seth@python.org>. Reported by Ellie <el@horse64.org>

- gh-121957 [https://github.com/python/cpython/issues/121957]: Fixed missing audit events around interactive use of Python, now also properly firing for `python -i`, as well as for `python -m asyncio`. The events in question are `cpython.run_stdin` and `cpython.run_startup`.

- gh-112301 [https://github.com/python/cpython/issues/112301]: Enable runtime protections for glibc to abort execution when unsafe behavior is encountered, for all platforms except Windows.

- gh-121285 [https://github.com/python/cpython/issues/121285]: Remove backtracking from tarfile header parsing for `hdrcharset`, PAX, and GNU sparse headers.

- gh-112301 [https://github.com/python/cpython/issues/112301]: Add default compiler options to improve security. Enable -Wimplicit-fallthrough, -fstack-protector-strong, -Wtrampolines.

- gh-118773 [https://github.com/python/cpython/issues/118773]: Fixes creation of ACLs in **`os.mkdir()`** on Windows to work correctly on non-English machines.

- gh-118486 [https://github.com/python/cpython/issues/118486]: **`os.mkdir()`** on Windows now accepts *mode* of `0o700` to restrict the new directory to the current user. This fixes **CVE 2024-4030** [https://www.cve.org/CVERecord?id=CVE-2024-4030] affecting **`tempfile.mkdtemp()`** in scenarios where the base temporary directory is more permissive than the default.

Library

- gh-125422 [https://github.com/python/cpython/issues/125422]: Fixed the bug where **`pdb`** and **`bdb`** can step into the bottom caller frame.

- gh-100141 [https://github.com/python/cpython/issues/100141]: Fixed the bug where **`pdb`** will be stuck in an infinite loop when debugging an empty file.

- gh-53203 [https://github.com/python/cpython/issues/53203]: Fix **`time.strptime()`** for `%c`, `%x` and `%X` formats in many locales that use non-ASCII digits, like Persian, Burmese, Odia and Shan.

- gh-125206 [https://github.com/python/cpython/issues/125206]: Workaround for old libffi versions is added. Module ctypes supports double complex only with libffi >= 3.3.0. Patch by Mikhail Efimov.

- gh-89967 [https://github.com/python/cpython/issues/89967]: Make `WeakKeyDictionary` and `WeakValueDictionary` safe against concurrent mutations from other threads. Patch by Kumar Aditya.

- gh-125260 [https://github.com/python/cpython/issues/125260]: The `gzip.compress()` *mtime* parameter now defaults to 0 for reproducible output. Patch by Bernhard M. Wiedemann and Adam Turner.

- gh-125243 [https://github.com/python/cpython/issues/125243]: Fix data race when creating `zoneinfo.ZoneInfo` objects in the free threading build.

- gh-125254 [https://github.com/python/cpython/issues/125254]: Fix a bug where ArgumentError includes the incorrect ambiguous option in `argparse`.

- gh-125235 [https://github.com/python/cpython/issues/125235]: Keep `tkinter` TCL paths in venv pointing to base installation on Windows.

- gh-61011 [https://github.com/python/cpython/issues/61011]: Fix inheritance of nested mutually exclusive groups from parent parser in `argparse.ArgumentParser`. Previously, all nested mutually exclusive groups lost their connection to the group containing them and were displayed as belonging directly to the parser.

- gh-125118 [https://github.com/python/cpython/issues/125118]: Don’t copy arbitrary values to _Bool in the `struct` module.

- gh-125069 [https://github.com/python/cpython/issues/125069]: Fix an issue where providing a `pathlib.PurePath` object as an initializer argument to a second `PurePath` object with a different `parser` resulted in arguments to the former object's initializer being joined by the latter object's parser.

- gh-125096 [https://github.com/python/cpython/issues/125096]: If the `PYTHON_BASIC_REPL` environment variable is set, the `site` module no longer imports the **`_pyrepl`** module. Moreover, the `site` module now respects `-E` and `-I` command line options: ignore `PYTHON_BASIC_REPL` in this case. Patch by Victor Stinner.

- gh-124969 [https://github.com/python/cpython/issues/124969]: Fix `locale.nl_langinfo(locale.ALT_DIGITS)`. Now it returns a tuple of up to 100 strings (an empty tuple on most locales). Previously it returned the first item of that tuple or an empty string.

- gh-124960 [https://github.com/python/cpython/issues/124960]: Fix support for the `barry_as_FLUFL` future flag in the new REPL.

- gh-69998 [https://github.com/python/cpython/issues/69998]: Fix `locale.nl_langinfo()` in case when different categories have different locales. The function now sets temporarily the `LC_CTYPE` locale in some cases. This temporary change affects other threads.

- gh-124958 [https://github.com/python/cpython/issues/124958]: Fix refcycles in exceptions raised from `asyncio.TaskGroup` and the python implementation of `asyncio.Future`

- gh-53203 [https://github.com/python/cpython/issues/53203]: Fix `time.strptime()` for `%c` and `%x` formats in many locales: Arabic, Bislama, Breton, Bodo, Kashubian, Chuvash, Estonian, French, Irish, Ge'ez, Gurajati, Manx Gaelic, Hebrew, Hindi, Chhattisgarhi, Haitian Kreyol, Japanese, Kannada, Korean, Marathi, Malay, Norwegian, Nynorsk, Punjabi, Rajasthani, Tok Pisin, Yoruba, Yue Chinese, Yau/Nungon and Chinese.

- gh-123961 [https://github.com/python/cpython/issues/123961]: Convert **`curses`** to multi-phase initialization (**PEP 489** [https://peps.python.org/pep-0489/]), thereby fixing reference leaks at interpreter shutdown. Patch by Bénédikt Tran.

- gh-124944 [https://github.com/python/cpython/issues/124944]: Add `SO_ORIGINAL_DST` to the **`socket`** module.

- gh-117151 [https://github.com/python/cpython/issues/117151]: The default buffer size used by **`shutil.copyfileobj()`** has been increased from 64k to 256k on non-Windows platforms. It was already larger on Windows.

- gh-90102 [https://github.com/python/cpython/issues/90102]: Skip the `isatty` system call during open() when the file is known to not be a character device. This provides a slight performance improvement when reading whole files.

- gh-124917 [https://github.com/python/cpython/issues/124917]: Allow calling **`os.path.exists()`** and **`os.path.lexists()`** with keyword arguments on Windows. Fixes a regression in 3.13.0.

- gh-65865 [https://github.com/python/cpython/issues/65865]: **`argparse`** now raises early error for invalid `help` arguments to **`add_argument()`**, **`add_subparsers()`** and **`add_parser()`**.

- gh-124653 [https://github.com/python/cpython/issues/124653]: Fix detection of the minimal Queue API needed by the **`logging`** module. Patch by Bénédikt Tran.

- gh-91818 [https://github.com/python/cpython/issues/91818]: The CLI of many modules (**`ast`**, **`ensurepip`**, **`json`**, **`pdb`**, **`sqlite3`**, **`tokenize`**, **`venv`**) now uses the actual executable name instead of simply “python” to display in the usage message.

- gh-124858 [https://github.com/python/cpython/issues/124858]: Fix reference cycles left in tracebacks in **`asyncio.open_connection()`** when used with `happy_eyeballs_delay`

- gh-124390 [https://github.com/python/cpython/issues/124390]: Fixed **`AssertionError`** when using **`asyncio.staggered.staggered_race()`** with **`asyncio.eager_task_factory`**.

- gh-85935 [https://github.com/python/cpython/issues/85935]: **`argparse.ArgumentParser.add_argument()`** now raises an exception if an action that does not consume arguments (like ‘store_const’ or ‘store_true’) or explicit `nargs=0` are specified for positional arguments.

- gh-124835 [https://github.com/python/cpython/issues/124835]: Make **`tomllib.loads()`** raise **`TypeError`** not **`AttributeError`** on bad input types that do not have the `replace` attribute. Improve error message when **`bytes`** is received.

- gh-124693 [https://github.com/python/cpython/issues/124693]: Fix a bug where **`argparse`** doesn’t recognize negative complex numbers or negative numbers using scientific notation.

- gh-124787 [https://github.com/python/cpython/issues/124787]: Fix **`typing.TypeAliasType`** with incorrect `type_params` argument. Now it raises a **`TypeError`** when a type parameter without a default follows one with a default, and when an entry in the `type_params` tuple is not a type parameter object.

- gh-66436 [https://github.com/python/cpython/issues/66436]: Improved prog default value for **`argparse.ArgumentParser`**. It will now include the name of the Python executable along with the module or package name, or the path to a directory, ZIP file, or directory within a ZIP file if the code was run that way.

- gh-116850 [https://github.com/python/cpython/issues/116850]: Fix **`argparse`** for namespaces with not directly writable dict (e.g. classes).

- gh-101552 [https://github.com/python/cpython/issues/101552]: Add an *annoation_format* parameter to **`inspect.signature()`**. Add an *quote_annotation_strings* parameter to **`inspect.Signature.format()`**. Use the new functionality to improve the display of annotations in signatures in **`pydoc`**. Patch by Jelle Zijlstra.

- gh-58573 [https://github.com/python/cpython/issues/58573]: Fix conflicts between abbreviated long options in the parent parser and subparsers in **`argparse`**.

- gh-124594 [https://github.com/python/cpython/issues/124594]: All **`asyncio`** REPL prompts run in the same **`context`**. Contributed by Bartosz Sławecki.

- gh-61181 [https://github.com/python/cpython/issues/61181]: Fix support of choices with string value in **`argparse`**. Substrings of the specified string no longer considered valid values.

- gh-116750 [https://github.com/python/cpython/issues/116750]: Provide **`sys.monitoring.clear_tool_id()`** to unregister all events and callbacks set by the tool.

- gh-124552 [https://github.com/python/cpython/issues/124552]: Improve the accuracy of **`bdb`**’s check for the possibility of breakpoint in a frame. This makes it possible to disable unnecessary events in functions.

- gh-124538 [https://github.com/python/cpython/issues/124538]: Fixed crash when using **`gc.get_referents()`** on a capsule object.
- gh-80259 [https://github.com/python/cpython/issues/80259]: Fix **`argparse`** support of positional arguments with `nargs='?'`, `default=argparse.SUPPRESS` and specified `type`.
- gh-120378 [https://github.com/python/cpython/issues/120378]: Fix a crash related to an integer overflow in **`curses.resizeterm()`** and **`curses.resize_term()`**.
- gh-124498 [https://github.com/python/cpython/issues/124498]: Fix **`typing.TypeAliasType`** not to be generic, when `type_params` is an empty tuple.
- gh-53834 [https://github.com/python/cpython/issues/53834]: Fix support of arguments with choices in **`argparse`**. Positional arguments with nargs equal to `'?'` or `'*'` no longer check default against `choices`. Optional arguments with `nargs` equal to `'?'` no longer check const against `choices`.
- gh-123884 [https://github.com/python/cpython/issues/123884]: Fixed bug in itertools.tee() handling of other tee inputs (a tee in a tee). The output now has the promised *n* independent new iterators. Formerly, the first iterator was identical (not independent) to the input iterator. This would sometimes give surprising results.
- gh-123017 [https://github.com/python/cpython/issues/123017]: Due to unreliable results on some devices, **`time.strftime()`** no longer accepts negative years on Android.
- gh-123014 [https://github.com/python/cpython/issues/123014]: **`os.pidfd_open()`** and **`signal.pidfd_send_signal()`** are now

unavailable when building against Android API levels older than 31, since the underlying system calls may cause a crash.

- gh-124176 [https://github.com/python/cpython/issues/124176]: Add support for **`dataclasses.dataclass()`** in **`unittest.mock.create_autospec()`**. Now `create_autospec` will check for potential dataclasses and use **`dataclasses.fields()`** function to retrieve the spec information.

- gh-124345 [https://github.com/python/cpython/issues/124345]: **`argparse`** supports abbreviated single-dash long options separated by `=` from its value.

- gh-124400 [https://github.com/python/cpython/issues/124400]: Fixed a **`pdb`** bug where `until` has no effect when it appears in a `commands` sequence. Also avoid printing the frame information at a breakpoint that has a command list containing a command that resumes execution.

- gh-90562 [https://github.com/python/cpython/issues/90562]: Modify dataclasses to support zero-argument super() when `slots=True` is specified. This works by modifying all references to `__class__` to point to the newly created class.

- gh-104860 [https://github.com/python/cpython/issues/104860]: Fix disallowing abbreviation of single-dash long options in **`argparse`** with `allow_abbrev=False`.

- gh-63143 [https://github.com/python/cpython/issues/63143]: Fix parsing mutually exclusive arguments in **`argparse`**. Arguments with the value identical to the default value (e.g. booleans, small integers, empty or 1-character strings) are no longer considered “not present”.

- gh-72795 [https://github.com/python/cpython/issues/72795]: Positional arguments with nargs equal to `'*'` or **`argparse.REMAINDER`** are no longer required. This allows to use positional argument with `nargs='*'` and without `default` in mutually exclusive group and improves error message about required arguments.

- gh-59317 [https://github.com/python/cpython/issues/59317]: Fix parsing positional argument with nargs equal to `'?'` or `'*'` if it is preceded by an option and another positional argument.

- gh-100980 [https://github.com/python/cpython/issues/100980]: The **`_fields_`** attribute of **`ctypes.Structure`** and **`Union`** is no longer set if the setattr operation raises an error.

- gh-53780 [https://github.com/python/cpython/issues/53780]: **`argparse`** now ignores the first `"--"` (double dash) between an option and command.

- gh-124217 [https://github.com/python/cpython/issues/124217]: Add RFC 9637 reserved IPv6 block `3fff::/20` in **`ipaddress`** module.

- gh-111513 [https://github.com/python/cpython/issues/111513]: Improve the error message that may be raised by **`datetime.date.fromtimestamp()`**.

- gh-124248 [https://github.com/python/cpython/issues/124248]: Fixed potential crash when using **`struct`** to process zero-width ‘Pascal string’ fields (`0p`).

- gh-81691 [https://github.com/python/cpython/issues/81691]: Fix handling of multiple `"--"` (double dashes) in **`argparse`**. Only the first one has now been removed, all subsequent ones are now taken literally.

- gh-87041 [https://github.com/python/cpython/issues/87041]: Fix a bug in **`argparse`** where lengthy subparser argument help is incorrectly indented.

- gh-84559 [https://github.com/python/cpython/issues/84559]: The default **`multiprocessing`** start method on Linux and other POSIX systems has been changed away from often unsafe `"fork"` to `"forkserver"` (when the platform supports sending file handles over pipes as most do) or `"spawn"`. Mac and Windows are unchanged as they already default to `"spawn"`.

- gh-124212 [https://github.com/python/cpython/issues/124212]: Fix invalid variable in **`venv`** handling of failed symlink on Windows

- gh-124171 [https://github.com/python/cpython/issues/124171]: Add workaround for broken **`fmod()`** implementations on Windows, that loose zero sign (e.g. `fmod(-10, 1)` returns `0.0`). Patch by Sergey B Kirpichev.

- gh-123978 [https://github.com/python/cpython/issues/123978]: Remove broken **`time.thread_time()`** and **`time.thread_time_ns()`** on NetBSD.

- gh-123934 [https://github.com/python/cpython/issues/123934]: Fix **`unittest.mock.MagicMock`** resetting magic methods return values after `.reset_mock(return_value=True)` was called.

- gh-124016 [https://github.com/python/cpython/issues/124016]: Update **`unicodedata`** database to Unicode 16.0.0.

- gh-123968 [https://github.com/python/cpython/issues/123968]: Fix the command-line interface for the **`random`** module to select floats between 0 and N, not 1 and N.

- gh-123945 [https://github.com/python/cpython/issues/123945]: Fix a bug where **`argparse`** doesn’t recognize negative numbers with underscores

- gh-123935 [https://github.com/python/cpython/issues/123935]: Fix parent slots detection for dataclasses that inherit from classes with `__dictoffset__`.

- gh-123892 [https://github.com/python/cpython/issues/123892]: Add `"_wmi"` to **`sys.stdlib_module_names`**. Patch by Victor Stinner.

- gh-84808 [https://github.com/python/cpython/issues/84808]: Fix error handling in **`socket`** method **`connect_ex()`** on platforms where **`errno`** can be negative.

- gh-123756 [https://github.com/python/cpython/issues/123756]: Added a new argument `mode` to **`pdb.Pdb`**. Only allow **`pdb`** from command line to use `restart` command.

- gh-122765 [https://github.com/python/cpython/issues/122765]: Fix unbalanced quote errors occurring when activate.csh in **`venv`** was sourced with a custom prompt containing unpaired quotes or newlines.

- gh-123657 [https://github.com/python/cpython/issues/123657]: Fix crash and memory leak in **`decimal.getcontext()`**. It crashed when using a thread-local context by `--with-decimal-contextvar=no`.

- gh-123339 [https://github.com/python/cpython/issues/123339]: Fix **`inspect.getsource()`** for classes in **`collections.abc`** and **`decimal`** (for pure Python implementation) modules. **`inspect.getcomments()`** now raises OSError instead of IndexError if the `__firstlineno__` value for a class is out of bound.

- gh-123374 [https://github.com/python/cpython/issues/123374]: Remove check for redefined memo entry in `pickletools.dis()`.

- gh-123504 [https://github.com/python/cpython/issues/123504]: Fixed reference leak in the finalization of `tkinter`.

- gh-123430 [https://github.com/python/cpython/issues/123430]: Pages generated by the `http.server` module allow the browser to apply its default dark mode.

- gh-123446 [https://github.com/python/cpython/issues/123446]: Fix empty function name in `TypeError` when `csv.reader()`, `csv.writer()`, or `csv.register_dialect()` are used without the required args.

- gh-123448 [https://github.com/python/cpython/issues/123448]: Fixed memory leak of `typing.NoDefault` by moving it to the static types array.

- gh-123409 [https://github.com/python/cpython/issues/123409]: Fix `ipaddress.IPv6Address.reverse_pointer` output according to **RFC 3596, §2.5** [https://datatracker.ietf.org/doc/html/rfc3596.html#section-2.5]. Patch by Bénédikt Tran.

- gh-123089 [https://github.com/python/cpython/issues/123089]: Make `weakref.WeakSet` safe against concurrent mutations while it is being iterated. Patch by Kumar Aditya.

- gh-123363 [https://github.com/python/cpython/issues/123363]: Show string value of `CONTAINS_OP` oparg in `dis` output. Patch by Alexandr153.

- gh-123341 [https://github.com/python/cpython/issues/123341]: Add `__class_getitem__()` to `tkinter.Event` for type subscript support at runtime. Patch by Adonis Rakateli.

- gh-123340 [https://github.com/python/cpython/issues/123340]: Show string value of `IS_OP` oparg in `dis` output.

- gh-119518 [https://github.com/python/cpython/issues/119518]: Speed up normalization of `pathlib.PurePath` and `Path` objects by not interning string parts.

- gh-123270 [https://github.com/python/cpython/issues/123270]: Applied a more surgical fix for malformed payloads in `zipfile.Path` causing infinite loops (gh-122905 [https://github.com/python/cpython/issues/122905]) without breaking contents using legitimate characters.

- gh-73991 [https://github.com/python/cpython/issues/73991]: Add `pathlib.Path.copy_into()` and `move_into()`, which copy and move files and directories into *existing* directories.

- gh-123228 [https://github.com/python/cpython/issues/123228]: Fix return type for `_pyrepl.readline._ReadlineWrapper.get_line_buffer()` to be `str()`. Patch by Sergey B Kirpichev.

- gh-123240 [https://github.com/python/cpython/issues/123240]: Raise audit events for the `input()` in the new REPL.

- gh-76960 [https://github.com/python/cpython/issues/76960]: Fix `urllib.parse.urljoin()` and `urllib.parse.urldefrag()` for URIs containing empty components. For example, `urljoin()` with relative reference “?” now sets empty query and removes fragment. Preserve empty components (authority, params, query, fragment) in `urljoin()`. Preserve empty components (authority, params, query) in `urldefrag()`.

- gh-116810 [https://github.com/python/cpython/issues/116810]: Resolve a memory leak introduced in CPython 3.10’s `ssl` when the `ssl.SSLSocket.session` property was accessed.

Speeds up read and write access to said property by no longer unnecessarily cloning session objects via serialization.

- gh-123243 [https://github.com/python/cpython/issues/123243]: Fix memory leak in **`_decimal`**.
- gh-122546 [https://github.com/python/cpython/issues/122546]: Consistently use same file name for different exceptions in the new repl. Patch by Sergey B Kirpichev.
- gh-123213 [https://github.com/python/cpython/issues/123213]: **`xml.etree.ElementTree.Element.extend()`** and **`Element`** assignment no longer hide the internal exception if an erroneous generator is passed. Patch by Bar Harel.
- gh-85110 [https://github.com/python/cpython/issues/85110]: Preserve relative path in URL without netloc in **`urllib.parse.urlunsplit()`** and **`urllib.parse.urlunparse()`**.
- gh-123165 [https://github.com/python/cpython/issues/123165]: Add support for rendering **`Positions`** in **`dis`**.
- gh-122909 [https://github.com/python/cpython/issues/122909]: In urllib.request when URLError is raised opening an ftp URL, the exception argument is now consistently a string. Earlier versions passed either a string or an ftplib exception instance as the argument to URLError.
- gh-123084 [https://github.com/python/cpython/issues/123084]: Deprecate **`shutil.ExecError`**, which hasn’t been raised by any **`shutil`** function since Python 3.4. It’s now an alias for **`RuntimeError`**.
- gh-123085 [https://github.com/python/cpython/issues/123085]: In a bare call to **`importlib.resources.files()`**, ensure the caller’s frame is properly detected when

`importlib.resources` is itself available as a compiled module only (no source).

- gh-123067 [https://github.com/python/cpython/issues/123067]: Fix quadratic complexity in parsing `"`-quoted cookie values with backslashes by **`http.cookies`**.
- gh-123049 [https://github.com/python/cpython/issues/123049]: Add support for **`UNNAMED_SECTION`** in **`configparser.ConfigParser.add_section()`**.
- gh-121735 [https://github.com/python/cpython/issues/121735]: When working with zip archives, importlib.resources now properly honors module-adjacent references (e.g. `files(pkg.mod)` and not just `files(pkg)`).
- gh-122981 [https://github.com/python/cpython/issues/122981]: Fix **`inspect.getsource()`** for generated classes with Python base classes (e.g. enums).
- gh-122903 [https://github.com/python/cpython/issues/122903]: `zipfile.Path.glob` now correctly matches directories instead of silently omitting them.
- gh-122905 [https://github.com/python/cpython/issues/122905]: **`zipfile.Path`** objects now sanitize names from the zipfile.
- gh-122873 [https://github.com/python/cpython/issues/122873]: Enable **`json`** module to work as a script using the **`-m`** switch: `python -m json`. See the JSON command-line interface documentation. Patch by Trey Hunner.
- gh-122858 [https://github.com/python/cpython/issues/122858]: Deprecate **`asyncio.iscoroutinefunction()`** in favor of **`inspect.iscoroutinefunction()`**.

- gh-116263 [https://github.com/python/cpython/issues/116263]: **`logging.handlers.RotatingFileHandler`** no longer rolls over empty log files.

- gh-105376 [https://github.com/python/cpython/issues/105376]: Restore the deprecated **`logging`** `warn()` method. It was removed in Python 3.13 alpha 1. Keep the deprecated `warn()` method in Python 3.13. Patch by Victor Stinner.

- gh-122311 [https://github.com/python/cpython/issues/122311]: Improve errors in the **`pickle`** module. **`PicklingError`** is now raised more often instead of **`UnicodeEncodeError`**, **`ValueError`** and **`AttributeError`**, and the original exception is chained to it. Improve and unify error messages in Python and C implementations.

- gh-122744 [https://github.com/python/cpython/issues/122744]: Bump the version of pip bundled in ensurepip to version 24.2.

- gh-118761 [https://github.com/python/cpython/issues/118761]: Improve import time of **`pprint`** by around seven times. Patch by Hugo van Kemenade.

- gh-118974 [https://github.com/python/cpython/issues/118974]: Add `decorator` parameter to **`dataclasses.make_dataclass()`** to customize the functional creation of dataclasses.

- gh-118814 [https://github.com/python/cpython/issues/118814]: Fix the **`typing.TypeVar`** constructor when name is passed by keyword.

- gh-122637 [https://github.com/python/cpython/issues/122637]: Adjust `cmath.tanh(nanj)` and `cmath.tanh(infj)` for recent C standards.

- gh-122478 [https://github.com/python/cpython/issues/122478]: Remove internal frames from tracebacks shown in `code.InteractiveInterpreter` with non-default `sys.excepthook()`. Save correct tracebacks in `sys.last_traceback` and update `__traceback__` attribute of `sys.last_value` and `sys.last_exc`.

- gh-116622 [https://github.com/python/cpython/issues/116622]: On Android, the `FICLONE` and `FICLONERANGE` constants are no longer exposed by `fcntl`, as these ioctls are blocked by SELinux.

- gh-82378 [https://github.com/python/cpython/issues/82378]: Make sure that the new REPL interprets `sys.tracebacklimit` in the same way that the classic REPL did.

- gh-122334 [https://github.com/python/cpython/issues/122334]: Fix crash when importing `ssl` after the main interpreter restarts.

- gh-122459 [https://github.com/python/cpython/issues/122459]: Optimize `pickling` by name objects without the `__module__` attribute.

- gh-87320 [https://github.com/python/cpython/issues/87320]: In `code.InteractiveInterpreter`, handle exceptions caused by calling a non-default `sys.excepthook()`. Before, the exception bubbled up to the caller, ending the REPL.

- gh-122272 [https://github.com/python/cpython/issues/122272]: On some platforms such as Linux, year with century was not 0-padded when formatted by `strftime()` with C99-specific specifiers `'%C'` or `'%F'`. The 0-padding behavior is now guaranteed when the format specifiers `'%C'` and `'%F'` are supported by the C library. Patch by Ben Hsing

- gh-122400 [https://github.com/python/cpython/issues/122400]: Handle `ValueError`s raised by `os.stat()` in `filecmp.dircmp` and `filecmp.cmpfiles()`. Patch by Bénédikt Tran.

- gh-122311 [https://github.com/python/cpython/issues/122311]: Fix some error messages in `pickle`.

- gh-121650 [https://github.com/python/cpython/issues/121650]: `email` headers with embedded newlines are now quoted on output. The `generator` will now refuse to serialize (write) headers that are unsafely folded or delimited; see `verify_generated_headers`. (Contributed by Bas Bloemsaat and Petr Viktorin in gh-121650 [https://github.com/python/cpython/issues/121650].)

- gh-122332 [https://github.com/python/cpython/issues/122332]: Fixed segfault with `asyncio.Task.get_coro()` when using an eager task factory.

- gh-105733 [https://github.com/python/cpython/issues/105733]: `ctypes.ARRAY()` is now soft deprecated: it no longer emits deprecation warnings and is not scheduled for removal.

- gh-122213 [https://github.com/python/cpython/issues/122213]: Add notes for pickle serialization errors that allow to identify the source of the error.

- gh-119180 [https://github.com/python/cpython/issues/119180]: As part of **PEP 749** [https://peps.python.org/pep-0749/], add the following attributes for customizing evaluation of annotation scopes:

  - `evaluate_value` on `typing.TypeAliasType`
  - `evaluate_bound`, `evaluate_constraints`, and `evaluate_default` on `typing.TypeVar`
  - `evaluate_default` on `typing.ParamSpec`
  - `evaluate_default` on `typing.TypeVarTuple`

- gh-119180 [https://github.com/python/cpython/issues/119180]: Fix handling of classes with custom metaclasses in `annotationlib.get_annotations`.

- gh-122170 [https://github.com/python/cpython/issues/122170]: Handle **`ValueError`**s raised by **`os.stat()`** in **`linecache`**. Patch by Bénédikt Tran.

- gh-122163 [https://github.com/python/cpython/issues/122163]: Add notes for JSON serialization errors that allow to identify the source of the error.

- gh-122129 [https://github.com/python/cpython/issues/122129]: Improve support of method descriptors and wrappers in the help title.

- gh-122145 [https://github.com/python/cpython/issues/122145]: Fix an issue when reporting tracebacks corresponding to Python code emitting an empty AST body. Patch by Nikita Sobolev and Bénédikt Tran.

- gh-121723 [https://github.com/python/cpython/issues/121723]: Make **`logging.config.dictConfig()`** accept any object implementing the Queue public API. See the queue configuration section for details. Patch by Bénédikt Tran.

- gh-82951 [https://github.com/python/cpython/issues/82951]: Serializing objects with complex `__qualname__` (such as unbound methods and nested classes) by name no longer involves serializing parent objects by value in pickle protocols < 4.

- gh-120754 [https://github.com/python/cpython/issues/120754]: `Pathlib.read_bytes` no longer opens the file in Python’s buffered I/O mode. This reduces overheads as the code reads a file in whole leading to a modest speedup.

- gh-113785 [https://github.com/python/cpython/issues/113785]: `csv` now correctly parses numeric fields (when used with `csv.QUOTE_NONNUMERIC` or `csv.QUOTE_STRINGS`) which start with an escape character.

- gh-122088 [https://github.com/python/cpython/issues/122088]: `@warnings.deprecated` now copies the coroutine status of functions and methods so that `inspect.iscoroutinefunction()` returns the correct result.

- gh-122081 [https://github.com/python/cpython/issues/122081]: Fix a crash in the **`decimal.IEEEContext()`** optional function available via the `EXTRA_FUNCTIONALITY` configuration flag.

- gh-73991 [https://github.com/python/cpython/issues/73991]: Add `pathlib.Path.move()`, which moves a file or directory tree.

- gh-121268 [https://github.com/python/cpython/issues/121268]: Remove workarounds for non-IEEE 754 systems in `cmath`.

- gh-119698 [https://github.com/python/cpython/issues/119698]: Due to the lack of interest for `symtable.Class.get_methods()`, the method is marked as deprecated and will be removed in Python 3.16. Patch by Bénédikt Tran.

- gh-121889 [https://github.com/python/cpython/issues/121889]: Adjusts `cmath.acosh(complex('0+nanj'))` for recent C standards.

- gh-121804 [https://github.com/python/cpython/issues/121804]: Correctly show error locations, when `SyntaxError` raised in new repl. Patch by Sergey B Kirpichev.

- gh-121797 [https://github.com/python/cpython/issues/121797]: Add alternative `Fraction` constructor `Fraction.from_number()`.

- gh-121798 [https://github.com/python/cpython/issues/121798]: Add alternative **`Decimal`** constructor **`Decimal.from_number()`**.

- gh-120930 [https://github.com/python/cpython/issues/120930]: Fixed a bug introduced by gh-92081 [https://github.com/python/cpython/issues/92081] that added an incorrect extra blank to encoded words occurring in wrapped headers.

- gh-57141 [https://github.com/python/cpython/issues/57141]: The *shallow* argument to **`filecmp.dircmp`** (new in Python 3.13) is now keyword-only.

- gh-121245 [https://github.com/python/cpython/issues/121245]: Simplify handling of the history file in `site.register_readline()` helper. The `CAN_USE_PYREPL` variable now will be initialized, when imported. Patch by Sergey B Kirpichev.

- gh-121249 [https://github.com/python/cpython/issues/121249]: Support the float complex and double complex C types in the **`struct`** module if the compiler has C11 complex arithmetic. Patch by Sergey B Kirpichev.

- gh-121486 [https://github.com/python/cpython/issues/121486]: **`math`** functions **`isqrt()`**, **`log()`**, **`log2()`** and **`log10()`** now support integers larger than `2**2**32` on 32-bit platforms.

- gh-121474 [https://github.com/python/cpython/issues/121474]: Fix missing sanity check for `parties` arg in **`threading.Barrier`** constructor. Patch by Clinton Christian (pygeek).

- gh-121450 [https://github.com/python/cpython/issues/121450]: Hard-coded breakpoints (**`breakpoint()`** and **`pdb.set_trace()`**) now reuse the most recent `Pdb` instance that calls `Pdb.set_trace()`, instead of creating a new one each time.

As a result, all the instance specific data like `display` and `commands` are preserved across Hard-coded breakpoints.

- gh-119169 [https://github.com/python/cpython/issues/119169]: Slightly speed up **`os.walk()`** by simplifying exception handling.
- gh-121423 [https://github.com/python/cpython/issues/121423]: Improve import time of **`socket`** by lazy importing modules and writing **`socket.errorTab`** as a constant.
- gh-59110 [https://github.com/python/cpython/issues/59110]: **`zipimport`** supports now namespace packages when no directory entry exists.
- gh-119004 [https://github.com/python/cpython/issues/119004]: Fix a crash in OrderedDict.__eq__ when operands are mutated during the check. Patch by Bénédikt Tran.
- gh-121313 [https://github.com/python/cpython/issues/121313]: Limit the reading size in the **`multiprocessing.connection.Connection`** class to 64 KiB to prevent memory overallocation and unnecessary memory management system calls.
- gh-121332 [https://github.com/python/cpython/issues/121332]: Fix constructor of **`ast`** nodes with custom `_attributes`. Previously, passing custom attributes would raise a **`DeprecationWarning`**. Passing arguments to the constructor that are not in `_fields` or `_attributes` remains deprecated. Patch by Jelle Zijlstra.
- gh-121245 [https://github.com/python/cpython/issues/121245]: Fix a bug in the handling of the command history of the new REPL that caused the history file to be wiped at REPL exit.

- gh-121210 [https://github.com/python/cpython/issues/121210]: Handle AST nodes with missing runtime fields or attributes in `ast.compare()`. Patch by Bénédikt Tran.

- gh-121163 [https://github.com/python/cpython/issues/121163]: Add support for `all` as an valid `action` for `warnings.simplefilter()` and `warnings.filterwarnings()`.

- gh-121151 [https://github.com/python/cpython/issues/121151]: Fix wrapping of long usage text of arguments inside a mutually exclusive group in `argparse`.

- gh-121141 [https://github.com/python/cpython/issues/121141]: Add support for `copy.replace()` to AST nodes. Patch by Bénédikt Tran.

- gh-87744 [https://github.com/python/cpython/issues/87744]: Fix waitpid race while calling `send_signal()` in asyncio. Patch by Kumar Aditya.

- gh-121027 [https://github.com/python/cpython/issues/121027]: Add a future warning in `functools.partial.__get__()`. In future Python versions `functools.partial` will be a method descriptor.

- gh-121027 [https://github.com/python/cpython/issues/121027]: Make the `functools.partial` object a method descriptor.

- gh-117784 [https://github.com/python/cpython/issues/117784]: CPython now detects whether its linked TLS library supports TLSv1.3 post-handshake authentication and disables that feature if support is lacking.

- gh-121025 [https://github.com/python/cpython/issues/121025]: Improve the `__repr__()` of `functools.partialmethod`. Patch by Bénédikt Tran.

- gh-121018 [https://github.com/python/cpython/issues/121018]: Fixed issues where **`argparse.ArgumentParser.parse_args()`** did not honor `exit_on_error=False`. Based on patch by Ben Hsing.

- gh-119614 [https://github.com/python/cpython/issues/119614]: Fix truncation of strings with embedded null characters in some internal operations in **`tkinter`**.

- gh-120910 [https://github.com/python/cpython/issues/120910]: When reading installed files from an egg, use `relative_to(walk_up=True)` to honor files installed outside of the installation root.

- gh-61103 [https://github.com/python/cpython/issues/61103]: Support float complex, double complex and long double complex C types in **`ctypes`** as **`c_float_complex`**, **`c_double_complex`** and **`c_longdouble_complex`** if the compiler has C11 complex arithmetic. Patch by Sergey B Kirpichev.

- gh-120888 [https://github.com/python/cpython/issues/120888]: Upgrade pip wheel bundled with ensurepip (pip 24.1.1)

- gh-101830 [https://github.com/python/cpython/issues/101830]: Accessing the **`tkinter`** object's string representation no longer converts the underlying Tcl object to a string on Windows.

- gh-120678 [https://github.com/python/cpython/issues/120678]: Fix regression in the new REPL that meant that globals from files passed using the `-i` argument would not be included in the REPL's global namespace. Patch by Alex Waygood.

- gh-120811 [https://github.com/python/cpython/issues/120811]: Fix possible memory leak in **`contextvars.Context.run()`**.

- gh-120782 [https://github.com/python/cpython/issues/120782]: Fix wrong references of the `datetime` types after reloading the module.

- gh-120713 [https://github.com/python/cpython/issues/120713]: `datetime.datetime.strftime()` now 0-pads years with less than four digits for the format specifiers `%Y` and `%G` on Linux. Patch by Ben Hsing

- gh-120769 [https://github.com/python/cpython/issues/120769]: Make empty line in `pdb` repeats the last command even when the command is from `cmdqueue`.

- gh-120780 [https://github.com/python/cpython/issues/120780]: Show string value of LOAD_SPECIAL oparg in `dis` output.

- gh-41431 [https://github.com/python/cpython/issues/41431]: Add `datetime.time.strptime()` and `datetime.date.strptime()`. Contributed by Wannes Boeykens.

- gh-120743 [https://github.com/python/cpython/issues/120743]: Soft deprecate `os.popen()` and `os.spawn*` functions. They should no longer be used to write new code. The `subprocess` module is recommended instead. Patch by Victor Stinner.

- gh-120732 [https://github.com/python/cpython/issues/120732]: Fix `name` passing to `unittest.mock.Mock` object when using `unittest.mock.create_autospec()`.

- gh-111259 [https://github.com/python/cpython/issues/111259]: `re` now handles patterns like `"[\s\S]"` or `"\s|\S"` which match any character as effectively as a dot with the `DOTALL` modifier (`"(?s:.)"`).

- gh-120683 [https://github.com/python/cpython/issues/120683]: Fix an error in `logging.LogRecord`, when the integer part of the timestamp is rounded up, while the millisecond calculation truncates, causing the log timestamp to be wrong by up to 999 ms (affected roughly 1 in 8 million timestamps).

- gh-118710 [https://github.com/python/cpython/issues/118710]: **`ipaddress.IPv4Address`** and **`ipaddress.IPv6Address`** attributes `version` and `max_prefixlen` are now available on the class.
- gh-120633 [https://github.com/python/cpython/issues/120633]: Move scrollbar and remove tear-off menus in turtledemo.
- gh-120606 [https://github.com/python/cpython/issues/120606]: Allow users to use EOF to exit `commands` definition in **`pdb`**
- gh-120284 [https://github.com/python/cpython/issues/120284]: Allow **`asyncio.Runner.run()`** to accept awaitable objects instead of simply coroutines.
- gh-120541 [https://github.com/python/cpython/issues/120541]: Improve the prompt in the “less” pager when **`help()`** is called with non-string argument.
- gh-120495 [https://github.com/python/cpython/issues/120495]: Fix incorrect exception handling in Tab Nanny. Patch by Wulian233.
- gh-120388 [https://github.com/python/cpython/issues/120388]: Improve a warning message when a test method in **`unittest`** returns something other than `None`. Now we show the returned object type and optional asyncio-related tip.
- gh-120381 [https://github.com/python/cpython/issues/120381]: Correct **`inspect.ismethoddescriptor()`** to check also for the lack of **`__delete__()`**. Patch by Jan Kaliszewski.

- gh-90425 [https://github.com/python/cpython/issues/90425]: The OS byte in gzip headers is now always set to 255 when using `gzip.compress()`.

- gh-120343 [https://github.com/python/cpython/issues/120343]: Fix column offset reporting for tokens that come after multiline f-strings in the `tokenize` module.

- gh-119180 [https://github.com/python/cpython/issues/119180]: As part of implementing **PEP 649** [https://peps.python.org/pep-0649/] and **PEP 749** [https://peps.python.org/pep-0749/], add a new module `annotationlib`. Add support for unresolved forward references in annotations to `dataclasses`, `typing.TypedDict`, and `typing.NamedTuple`.

- gh-119600 [https://github.com/python/cpython/issues/119600]: Fix `unittest.mock.patch()` to not read attributes of the target when `new_callable` is set. Patch by Robert Collins.

- gh-120289 [https://github.com/python/cpython/issues/120289]: Fixed the use-after-free issue in `cProfile` by disallowing `disable()` and `clear()` in external timers.

- gh-82017 [https://github.com/python/cpython/issues/82017]: Added support for converting any objects that have the **`as_integer_ratio()`** method to a `Fraction`.

- gh-114053 [https://github.com/python/cpython/issues/114053]: Fix edge-case bug where `typing.get_type_hints()` would produce incorrect results if type parameters in a class scope were overridden by assignments in a class scope and `from __future__ import annotations` semantics were enabled. Patch by Alex Waygood.

- gh-114053 [https://github.com/python/cpython/issues/114053]: Fix erroneous `NameError` when calling `inspect.get_annotations()` with `eval_str=True`` on a class that made use of **PEP 695** [https://peps.python.org/pep-0695/] type parameters in a module that had `from __future__ import annotations` at the top of the file. Patch by Alex Waygood.

- gh-120268 [https://github.com/python/cpython/issues/120268]: Prohibit passing `None` to pure-Python **`datetime.date.fromtimestamp()`** to achieve consistency with C-extension implementation.
- gh-120244 [https://github.com/python/cpython/issues/120244]: Fix memory leak in **`re.sub()`** when the replacement string contains backreferences.
- gh-120254 [https://github.com/python/cpython/issues/120254]: Added `commands` argument to **`pdb.set_trace()`** which allows users to send debugger commands from the source file.
- gh-120211 [https://github.com/python/cpython/issues/120211]: Fix **`tkinter.ttk`** with Tcl/Tk 9.0.
- gh-71587 [https://github.com/python/cpython/issues/71587]: Fix crash in C version of **`datetime.datetime.strptime()`** when called again on the restarted interpreter.
- gh-117983 [https://github.com/python/cpython/issues/117983]: Defer the `threading` import in `importlib.util` until lazy loading is used.
- gh-120157 [https://github.com/python/cpython/issues/120157]: Remove unused constant `concurrent.futures._base._FUTURE_STATES` in **`concurrent.futures`**. Patch by Clinton Christian (pygeek).
- gh-120161 [https://github.com/python/cpython/issues/120161]: **`datetime`** no longer crashes in certain complex reference cycle situations.

- gh-119698 [https://github.com/python/cpython/issues/119698]: Fix `symtable.Class.get_methods()` and document its behaviour. Patch by Bénédikt Tran.

- gh-120121 [https://github.com/python/cpython/issues/120121]: Add `concurrent.futures.InvalidStateError` to module's `__all__`.

- gh-119933 [https://github.com/python/cpython/issues/119933]: Add the `symtable.SymbolTableType` enumeration to represent the possible outputs of the `symtable.SymbolTable.get_type` method. Patch by Bénédikt Tran.

- gh-120029 [https://github.com/python/cpython/issues/120029]: Expose `symtable.Symbol` methods `is_free_class()`, `is_comp_iter()` and `is_comp_cell()`. Patch by Bénédikt Tran.

- gh-120108 [https://github.com/python/cpython/issues/120108]: Fix calling `copy.deepcopy()` on `ast` trees that have been modified to have references to parent nodes. Patch by Jelle Zijlstra.

- gh-120056 [https://github.com/python/cpython/issues/120056]: Add **`socket.IP_RECVERR`** and **`socket.IP_RECVTTL`** constants (both available since Linux 2.2). And **`socket.IP_RECVORIGDSTADDR`** constant (available since Linux 2.6.29).

- gh-112672 [https://github.com/python/cpython/issues/112672]: Support building `tkinter` with Tcl 9.0.

- gh-120057 [https://github.com/python/cpython/issues/120057]: Added the `os.environ.refresh()` method to update `os.environ` with changes to the environment made by `os.putenv()`, by `os.unsetenv()`, or made outside Python in the same process. Patch by Victor Stinner.

- gh-120029 [https://github.com/python/cpython/issues/120029]: Expose `symtable.Symbol.is_type_parameter()` in the `symtable` module. Patch by Bénédikt Tran.

- gh-119819 [https://github.com/python/cpython/issues/119819]: Fix regression to allow logging configuration with multiprocessing queue types.
- gh-65454 [https://github.com/python/cpython/issues/65454]: **`unittest.mock.Mock.attach_mock()`** no longer triggers a call to a `PropertyMock` being attached.
- gh-117142 [https://github.com/python/cpython/issues/117142]: The **`ctypes`** module may now be imported in all subinterpreters, including those that have their own GIL.
- gh-118835 [https://github.com/python/cpython/issues/118835]: Fix _pyrepl crash when using custom prompt with ANSI escape codes.
- gh-81936 [https://github.com/python/cpython/issues/81936]: **`help()`** and **`showtopic()`** methods now respect a configured *output* argument to **`pydoc.Helper`** and not use the pager in such cases. Patch by Enrico Tröger.
- gh-117398 [https://github.com/python/cpython/issues/117398]: The `_datetime` module (C implementation for **`datetime`**) now supports being imported in multiple interpreters.
- gh-119824 [https://github.com/python/cpython/issues/119824]: Print stack entry in **`pdb`** when and only when user input is needed.
- gh-119838 [https://github.com/python/cpython/issues/119838]: In mixed arithmetic operations with **`Fraction`** and complex, the fraction is now converted to **`float`** instead of **`complex`**.
- gh-119770 [https://github.com/python/cpython/issues/119770]: Make **`termios`** `ioctl()` constants positive. Patch by Victor

Stinner.

- gh-89727 [https://github.com/python/cpython/issues/89727]: Fix issue with **`shutil.rmtree()`** where a **`RecursionError`** is raised on deep directory trees.

- gh-119577 [https://github.com/python/cpython/issues/119577]: The **`DeprecationWarning`** emitted when testing the truth value of an **`xml.etree.ElementTree.Element`** now describes unconditionally returning `True` in a future version rather than raising an exception in Python 3.14.

- gh-89727 [https://github.com/python/cpython/issues/89727]: Partially fix issue with **`shutil.rmtree()`** where a **`RecursionError`** is raised on deep directory trees. A recursion error is no longer raised when **`rmtree.avoids_symlink_attacks`** is false.

- gh-93963 [https://github.com/python/cpython/issues/93963]: Remove deprecated names from `importlib.abc` as found in `importlib.resources.abc`.

- gh-119118 [https://github.com/python/cpython/issues/119118]: Fix performance regression in the **`tokenize`** module by caching the `line` token attribute and calculating the column offset more efficiently.

- gh-89727 [https://github.com/python/cpython/issues/89727]: Fix issue with **`os.fwalk()`** where a **`RecursionError`** was raised on deep directory trees by adjusting the implementation to be iterative instead of recursive.

- gh-119594 [https://github.com/python/cpython/issues/119594]: If one calls pow(fractions.Fraction, x, module) with modulo not None, the error message now says that the types are incompatible rather than saying pow only takes 2 arguments. Patch by Wim Jeantine-Glenn and Mark Dickinson.

- gh-119588 [https://github.com/python/cpython/issues/119588]: `zipfile.Path.is_symlink` now assesses if the given path is a symlink.
- gh-119562 [https://github.com/python/cpython/issues/119562]: Remove **`ast.Num`**, **`ast.Str`**, **`ast.Bytes`**, **`ast.NameConstant`** and **`ast.Ellipsis`**. They had all emitted deprecation warnings since Python 3.12. Patch by Alex Waygood.
- gh-119555 [https://github.com/python/cpython/issues/119555]: Catch **`SyntaxError`** from **`compile()`** in the runsource() method of the InteractiveColoredConsole. Patch by Sergey B Kirpichev.
- gh-118908 [https://github.com/python/cpython/issues/118908]: Limit exposed globals from internal imports and definitions on new REPL startup. Patch by Eugene Triguba and Pablo Galindo.
- gh-117865 [https://github.com/python/cpython/issues/117865]: Improve the import time of the **`ast`** module by deferring the import of **`re`**. Patch by Jelle Zijlstra.
- gh-119127 [https://github.com/python/cpython/issues/119127]: Positional arguments of **`functools.partial()`** objects now support placeholders via **`functools.Placeholder`**.
- gh-113892 [https://github.com/python/cpython/issues/113892]: Now, the method `sock_connect` of **`asyncio.ProactorEventLoop`** raises a **`ValueError`** if given socket is not in non-blocking mode, as well as in other loop implementations.
- gh-119506 [https://github.com/python/cpython/issues/119506]: Fix **`io.TextIOWrapper.write()`** method breaks internal buffer

when the method is called again during flushing internal buffer.

- gh-69214 [https://github.com/python/cpython/issues/69214]: Fix `fcntl.ioctl()` *request* parameter: use an `unsigned long` instead of an `unsigned int` for the *request* parameter of **`fcntl.ioctl()`** to support requests larger than `UINT_MAX`. Patch by Victor Stinner.

- gh-119105 [https://github.com/python/cpython/issues/119105]: `difflib`’s `DIffer.compare()` (and so also `ndiff`) can no longer be provoked into cubic-time behavior, or into unbounded recursion, and should generally be faster in ordinary cases too. Results may change in some cases, although that should be rare. Correctness of diffs is not affected. Some similar lines far apart may be reported as deleting one and adding the other, where before they were displayed on adjacent output lines with markup showing the intraline differences.

- gh-119443 [https://github.com/python/cpython/issues/119443]: The interactive REPL no longer runs with `from __future__ import annotations` enabled. Patch by Jelle Zijlstra.

- gh-119461 [https://github.com/python/cpython/issues/119461]: Add `socket.VMADDR_CID_LOCAL` constant. Patch by Victor Stinner.

- gh-117398 [https://github.com/python/cpython/issues/117398]: Objects in the datetime C-API are now all statically allocated, which means better memory safety, especially when the module is reloaded. This should be transparent to users.

- gh-118894 [https://github.com/python/cpython/issues/118894]: **`asyncio`** REPL now has the same capabilities as PyREPL.

- gh-118830 [https://github.com/python/cpython/issues/118830]: Bump **`pickle`** default protocol to `5`.

- gh-118911 [https://github.com/python/cpython/issues/118911]: In PyREPL, updated `maybe-accept`’s logic so that if the user hits `Enter` twice, they are able to terminate the block even if there’s trailing whitespace. Also, now when the user hits arrow up, the cursor is on the last functional line. This matches IPython’s behavior. Patch by Aya Elsayed.

- gh-115225 [https://github.com/python/cpython/issues/115225]: Raise error on certain technically valid but pathological ISO 8601 strings passed to **`datetime.time.fromisoformat()`** that were previously parsed incorrectly.

- gh-111201 [https://github.com/python/cpython/issues/111201]: Remove dependency to **`readline`** from the new Python REPL.

- gh-119189 [https://github.com/python/cpython/issues/119189]: When using the `**` operator or **`pow()`** with **`Fraction`** as the base and an exponent that is not rational, a float, or a complex, the fraction is no longer converted to a float.

- gh-119174 [https://github.com/python/cpython/issues/119174]: Fix high DPI causes turtledemo(turtle-graphics examples) windows blurry Patch by Wulian233 and Terry Jan Reedy

- gh-119121 [https://github.com/python/cpython/issues/119121]: Fix a NameError happening in `asyncio.staggered.staggered_race`. This function is now tested.

- gh-119105 [https://github.com/python/cpython/issues/119105]: `difflib.Differ` is much faster for some cases of diffs where many pairs of lines are equally similar.

- gh-119113 [https://github.com/python/cpython/issues/119113]: Fix issue where **`pathlib.PurePath.with_suffix()`** didn’t raise **`TypeError`** when given `None` as a suffix.

- gh-118643 [https://github.com/python/cpython/issues/118643]: Fix an AttributeError in the **`email`** module when re-fold a long address list. Also fix more cases of incorrect encoding of the address separator in the address list.

- gh-73991 [https://github.com/python/cpython/issues/73991]: Add **`pathlib.Path.copy()`**, which copies a file or directory to another.

- gh-58933 [https://github.com/python/cpython/issues/58933]: Make **`pdb`** return to caller frame correctly when `f_trace` of the caller frame is not set

- gh-82805 [https://github.com/python/cpython/issues/82805]: Support single-dot file extensions in **`pathlib.PurePath.suffix`** and related attributes and methods. For example, the **`suffixes`** of `PurePath('foo.bar.')` are now `['.bar', '.']` rather than `[]`. This brings file extension splitting in line with **`os.path.splitext()`**.

- gh-118924 [https://github.com/python/cpython/issues/118924]: Remove **`version`** and **`version_info`** from **`sqlite3`**. Patch by Hugo van Kemenade.

- gh-118928 [https://github.com/python/cpython/issues/118928]: Disallow using a sequence of parameters with named placeholders in **`sqlite3`** queries. Patch by Erlend E. Aasland.

- gh-118895 [https://github.com/python/cpython/issues/118895]: Setting attributes on **`typing.NoDefault`** now raises **`AttributeError`** instead of **`TypeError`**.

- gh-118868 [https://github.com/python/cpython/issues/118868]: Fixed issue where kwargs were no longer passed to the logging handler QueueHandler

- gh-118827 [https://github.com/python/cpython/issues/118827]: Remove deprecated `Quoter` class from `urllib.parse`. It had previously raised a `DeprecationWarning` since Python 3.11. Patch by Nikita Sobolev.

- gh-118824 [https://github.com/python/cpython/issues/118824]: Remove deprecated `pty.master_open()` and `pty.slave_open()`. Use `pty.openpty()` instead. Patch by Nikita Sobolev.

- gh-118851 [https://github.com/python/cpython/issues/118851]: `ctx` arguments to the constructors of `ast` node classes now default to `ast.Load()`. Patch by Jelle Zijlstra.

- gh-101588 [https://github.com/python/cpython/issues/101588]: Remove copy, deepcopy, and pickle from itertools. These had previously raised a DeprecationWarning since Python 3.12.

- gh-118805 [https://github.com/python/cpython/issues/118805]: Remove *type*, *choices*, and *metavar* parameters of `argparse.BooleanOptionalAction`. They were deprecated since Python 3.12.

- gh-118803 [https://github.com/python/cpython/issues/118803]: `typing.ByteString` and `collections.abc.ByteString` are removed. They had previously raised a `DeprecationWarning` since Python 3.12.

- gh-118798 [https://github.com/python/cpython/issues/118798]: The *isdst* parameter has been removed from `email.utils.localtime()`. Patch by Hugo van Kemenade.

- gh-118760 [https://github.com/python/cpython/issues/118760]: Restore the default value of `tkiter.wantobjects` to `1`.

- gh-118760 [https://github.com/python/cpython/issues/118760]: Fix errors in calling Tkinter bindings on Windows.

- gh-74033 [https://github.com/python/cpython/issues/74033]: Drop support for passing keyword arguments to `pathlib.Path`.

- gh-101357 [https://github.com/python/cpython/issues/101357]: Suppress all `OSError` exceptions from `pathlib.Path.exists()` and `is_*()` methods, rather than a selection of more common errors. The new behaviour is consistent with `os.path.exists()`, `os.path.isdir()`, etc. Use `pathlib.Path.stat()` to retrieve the file status without suppressing exceptions.

- gh-78707 [https://github.com/python/cpython/issues/78707]: Drop support for passing additional positional arguments to `pathlib.PurePath.relative_to()` and `is_relative_to()`.

- gh-118507 [https://github.com/python/cpython/issues/118507]: Fix `os.path.isfile()` on Windows for pipes. Speedup `os.path.isjunction()` and `os.path.lexists()` on Windows with a native implementation.

- gh-118772 [https://github.com/python/cpython/issues/118772]: Allow `typing.TypeVar` instances without a default to follow instances without a default in some cases. Patch by Jelle Zijlstra.

- gh-118714 [https://github.com/python/cpython/issues/118714]: Allow `restart` in post-mortem debugging of `pdb`. Removed restart message when the user quits pdb from post-mortem mode.

- gh-118673 [https://github.com/python/cpython/issues/118673]: Removed executable bits and shebang from stdlib modules.

- gh-110863 [https://github.com/python/cpython/issues/110863]: **`os.path.realpath()`** now suppresses any **`OSError`** from **`os.readlink()`** when *strict* mode is disabled (the default).

- gh-118263 [https://github.com/python/cpython/issues/118263]: Speed up **`os.path.splitroot()`** & **`os.path.normpath()`** with a direct C call.

- gh-110190 [https://github.com/python/cpython/issues/110190]: Fix ctypes structs with array on SPARC by setting `MAX_STRUCT_SIZE` to 32 in stgdict. Patch by Jakub Kulik

- gh-118033 [https://github.com/python/cpython/issues/118033]: Fix **`dataclasses.dataclass()`** not creating a `__weakref__` slot when subclassing **`typing.Generic`**.

- gh-117766 [https://github.com/python/cpython/issues/117766]: Always use **`str()`** to print `choices` in **`argparse`**.

- gh-106531 [https://github.com/python/cpython/issues/106531]: In **`importlib.resources`**, sync with importlib_resources 6.3.2 [https://importlib-resources.readthedocs.io/en/latest/history.html#v6-3-2], including: `MultiplexedPath` now expects `Traversable` paths, deprecating string arguments to `MultiplexedPath`; Enabled support for resources in namespace packages in zip files; Fixed `NotADirectoryError` when calling files on a subdirectory of a namespace package.

- gh-115937 [https://github.com/python/cpython/issues/115937]: Removed extra preprocessing for the `__signature__` attribute: the code just check if it’s a **`inspect.Signature`** instance. Patch by Sergey B Kirpichev.

- gh-115808 [https://github.com/python/cpython/issues/115808]: Add **`operator.is_none()`** and **`operator.is_not_none()`** functions.

- gh-114264 [https://github.com/python/cpython/issues/114264]: Improve performance of **`copy.deepcopy()`** by adding a fast path for atomic types.

- gh-113878 [https://github.com/python/cpython/issues/113878]: Add *doc* parameter to **`dataclasses.field()`**, so it can be stored and shown as a documentation / metadata. If `@dataclass(slots=True)` is used, then the supplied string is available in the **`__slots__`** dict. Otherwise, the supplied string is only available in the corresponding **`dataclasses.Field`** object.

  In order to support this feature we are changing the `__slots__` format in dataclasses from **`tuple`** to **`dict`** when documentation / metadata is present.

- gh-113978 [https://github.com/python/cpython/issues/113978]: Ignore warnings on text completion inside REPL.

- gh-113008 [https://github.com/python/cpython/issues/113008]: Correct argparse usage output for required, mutually exclusive groups containing a positional argument

- gh-108172 [https://github.com/python/cpython/issues/108172]: `webbrowser` honors OS preferred browser on Linux when its desktop entry name contains the text of a known browser name.

- gh-109109 [https://github.com/python/cpython/issues/109109]: You can now get the raw TLS certificate chains from TLS connections via **`ssl.SSLSocket.get_verified_chain()`** and **`ssl.SSLSocket.get_unverified_chain()`** methods.

  Contributed by Mateusz Nowak.

- gh-105623 [https://github.com/python/cpython/issues/105623]: Fix performance degradation in **`logging.handlers.RotatingFileHandler`**. Patch by Craig Robson.

- gh-102450 [https://github.com/python/cpython/issues/102450]: Add missing ISO-8601 24:00 alternative to midnight of next day to **`datetime.datetime.fromisoformat()`** and **`datetime.time.fromisoformat()`**. Patch by Izan “TizzySaurus” Robinson (tizzysaurus@gmail.com)

- gh-103956 [https://github.com/python/cpython/issues/103956]: Fix lack of newline characters in **`trace`** module output when line tracing is enabled but source code line for current frame is not available.

- gh-92081 [https://github.com/python/cpython/issues/92081]: Fix missing spaces in email headers when the spaces are mixed with encoded 8-bit characters.

- gh-103194 [https://github.com/python/cpython/issues/103194]: Prepare Tkinter for C API changes in Tcl 8.7/9.0 to avoid **`_tkinter.Tcl_Obj`** being unexpectedly returned instead of **`bool`**, **`str`**, **`bytearray`**, or **`int`**.

- gh-87106 [https://github.com/python/cpython/issues/87106]: Fixed handling in **`inspect.Signature.bind()`** of keyword arguments having the same name as positional-only arguments when a variadic keyword argument (e.g. `**kwargs`) is present.

- gh-103134 [https://github.com/python/cpython/issues/103134]: Add additional methods to proxy objects in the **`multiprocessing`** module:

    - **`clear()`** and **`copy()`** for proxies of **`list`**

  - `fromkeys()`, `reversed(d)`, `d | {}`, `{} | d`, `d |= {'b': 2}` for proxies of `dict`

- bpo-45767 [https://bugs.python.org/issue?@action=redirect&bpo=45767]: Fix integer conversion in `os.major()`, `os.minor()`, and `os.makedev()`. Support device numbers larger than `2**63-1`. Support non-existent device number (`NODEV`).

- bpo-44864 [https://bugs.python.org/issue?@action=redirect&bpo=44864]: Do not translate user-provided strings in `argparse.ArgumentParser`.

- bpo-41843 [https://bugs.python.org/issue?@action=redirect&bpo=41843]: Solaris now uses `os.sendfile()` fast-copy syscall for more efficient `shutil` file copy related functions.

- bpo-15987 [https://bugs.python.org/issue?@action=redirect&bpo=15987]: Implemented `ast.compare()` for comparing two ASTs. Patch by Batuhan Taskaya with some help from Jeremy Hylton.

- gh-67693 [https://github.com/python/cpython/issues/67693]: Fix `urllib.parse.urlunparse()` and `urllib.parse.urlunsplit()` for URIs with path starting with multiple slashes and no authority. Based on patch by Ashwin Ramaswami.

- bpo-14074 [https://bugs.python.org/issue?@action=redirect&bpo=14074]: Fix `argparse` metavar processing to allow positional arguments to have a tuple metavar.

IDLE

- gh-122392 [https://github.com/python/cpython/issues/122392]: Increase currently inadequate vertical spacing for the

IDLE browsers (path, module, and stack) on high-resolution monitors.

- gh-112938 [https://github.com/python/cpython/issues/112938]: Fix uninteruptable hang when Shell gets rapid continuous output.
- gh-122482 [https://github.com/python/cpython/issues/122482]: Change About IDLE to direct users to discuss.python.org instead of the now unused idle-dev email and mailing list.
- gh-78889 [https://github.com/python/cpython/issues/78889]: Stop Shell freezes by blocking user access to non-method sys.stdout.shell attributes, which are all private.
- gh-120083 [https://github.com/python/cpython/issues/120083]: Add explicit black IDLE Hovertip foreground color needed for recent macOS. Fixes Sonoma showing unreadable white on pale yellow. Patch by John Riggles.
- gh-120104 [https://github.com/python/cpython/issues/120104]: Fix padding in config and search dialog windows in IDLE.

Documentation

- gh-124872 [https://github.com/python/cpython/issues/124872]: Added definitions for context, current context, and context management protocol, updated related definitions to be consistent, and expanded the documentation for `contextvars.Context`.

- gh-125018 [https://github.com/python/cpython/issues/125018]: The `importlib.metadata` documentation now includes semantic cross-reference targets for the significant documented APIs. This means intersphinx references like `importlib.metadata.version()` will now work as expected.

- gh-124720 [https://github.com/python/cpython/issues/124720]: Update “Using Python on a Mac” section of the “Python

Setup and Usage” document and include information on installing free-threading support.

- gh-124457 [https://github.com/python/cpython/issues/124457]: Remove coverity scan from the CPython repo. It has not been used since 2020 and is currently unmaintained.

- gh-116622 [https://github.com/python/cpython/issues/116622]: Add an Android platform guide, and flag modules not available on Android.

- gh-123976 [https://github.com/python/cpython/issues/123976]: Refresh docs around custom providers.

- gh-70870 [https://github.com/python/cpython/issues/70870]: Clarified the dual usage of the term “free variable” (both the formal meaning of any reference to names defined outside the local scope, and the narrower pragmatic meaning of nonlocal variables named in `co_freevars`).

- gh-121277 [https://github.com/python/cpython/issues/121277]: Writers of CPython’s documentation can now use `next` as the version for the `versionchanged`, `versionadded`, `deprecated` directives.

- gh-117765 [https://github.com/python/cpython/issues/117765]: Improved documentation for `unittest.mock.patch.dict()`

- gh-121749 [https://github.com/python/cpython/issues/121749]: Fix documentation for `PyModule_AddObjectRef()`.

- gh-120012 [https://github.com/python/cpython/issues/120012]: Clarify the behaviours of `multiprocessing.Queue.empty()` and `multiprocessing.SimpleQueue.empty()` on closed queues. Patch by Bénédikt Tran.

- gh-119574 [https://github.com/python/cpython/issues/119574]: Added some missing environment variables to the output of `--help-env`.

- bpo-34008 [https://bugs.python.org/issue?@action=redirect&bpo=34008]: The `Py_Main()` documentation moved from the “Very High Level API” section to the “Initialization and Finalization” section.

  Also make it explicit that we expect `Py_Main` to typically be called instead of `Py_Initialize` rather than after it (since `Py_Main` makes its own call to `Py_Initialize`). Document that calling both is supported but is version dependent on which settings will be applied correctly.

Core and Builtins

- gh-124375 [https://github.com/python/cpython/issues/124375]: Fix a crash in the free threading build when the GC runs concurrently with a new thread starting.
- gh-125221 [https://github.com/python/cpython/issues/125221]: Fix possible race condition when calling `__reduce_ex__()` for the first time in the free threading build.
- gh-125174 [https://github.com/python/cpython/issues/125174]: Make the handling of reference counts of immortal objects more robust. Immortal objects with reference counts that deviate from their original reference count by up to a billion (half a billion on 32 bit builds) are still counted as immortal.
- gh-125039 [https://github.com/python/cpython/issues/125039]: Make `this_instr` and `prev_instr` const in cases generator.

Library

- gh-125008 [https://github.com/python/cpython/issues/125008]: Fix `tokenize.untokenize()` producing invalid syntax for double braces preceded by certain escape characters.

Core and Builtins

- gh-124871 [https://github.com/python/cpython/issues/124871]: Fix compiler bug (in some versions of 3.13) where an assertion fails during reachability analysis.
- gh-123378 [https://github.com/python/cpython/issues/123378]: Fix a crash in the `__str__()` method of `UnicodeError` objects when the `UnicodeError.start` and `UnicodeError.end` values are invalid or out-of-range. Patch by Bénédikt Tran.
- gh-118093 [https://github.com/python/cpython/issues/118093]: Improve the experimental JIT compiler's ability to stay "on trace" when encountering highly-biased branches.
- gh-124642 [https://github.com/python/cpython/issues/124642]: Fixed scalability issue in free-threaded builds for lock-free reads from dictionaries in multi-threaded scenarios
- gh-116510 [https://github.com/python/cpython/issues/116510]: Fix a crash caused by immortal interned strings being shared between sub-interpreters that use basic single-phase init. In that case, the string can be used by an interpreter that outlives the interpreter that created and interned it. For interpreters that share obmalloc state, also share the interned dict with the main interpreter.
- gh-116510 [https://github.com/python/cpython/issues/116510]: Fix a bug that can cause a crash when sub-interpreters use "basic" single-phase extension modules. Shared objects could refer to PyGC_Head nodes that had been freed as part of interpreter cleanup.
- gh-119180 [https://github.com/python/cpython/issues/119180]: The `__main__` module no longer always contains an `__annotations__` dictionary in its global namespace.
- gh-124547 [https://github.com/python/cpython/issues/124547]: When deallocating an object with inline values whose `__dict__` is still live: if memory allocation for the inline

values fails, clear the dictionary. Prevents an interpreter crash.

- gh-124513 [https://github.com/python/cpython/issues/124513]: Fix a crash in FrameLocalsProxy constructor: check the number of arguments. Patch by Victor Stinner.
- gh-124442 [https://github.com/python/cpython/issues/124442]: Fix nondeterminism in compilation by sorting the value of `__static_attributes__`. Patch by kp2pml30.
- gh-124285 [https://github.com/python/cpython/issues/124285]: Fix bug where `bool(a)` can be invoked more than once during the evaluation of a compound boolean expression.
- gh-123856 [https://github.com/python/cpython/issues/123856]: Fix PyREPL failure when a keyboard interrupt is triggered after using a history search
- gh-65961 [https://github.com/python/cpython/issues/65961]: Deprecate the setting and using `__package__` and `__cached__`.
- gh-119726 [https://github.com/python/cpython/issues/119726]: The JIT now generates more efficient code for calls to C functions resulting in up to 0.8% memory savings and 1.5% speed improvement on AArch64. Patch by Diego Russo.
- gh-122878 [https://github.com/python/cpython/issues/122878]: Use the `pager` binary, if available (e.g. on Debian and derivatives), to display REPL `help()`.
- gh-124188 [https://github.com/python/cpython/issues/124188]: Fix reading and decoding a line from the source file with non-UTF-8 encoding for syntax errors raised in the compiler.
- gh-124027 [https://github.com/python/cpython/issues/124027]: Support `<page up>`, `<page down>`, and `<delete>` keys in the Python REPL when `$TERM` is set to `vt100`.
- gh-124022 [https://github.com/python/cpython/issues/124022]: Fix bug where docstring is removed from classes in interactive mode.

- gh-123958 [https://github.com/python/cpython/issues/123958]: docstrings are now removed from the optimized AST in optimization level 2.
- gh-123923 [https://github.com/python/cpython/issues/123923]: The `f_executable` field in the internal **`_PyInterpreterFrame`** struct now uses a tagged pointer. Profilers and debuggers that uses this field should clear the least significant bit to recover the PyObject* pointer.
- gh-77894 [https://github.com/python/cpython/issues/77894]: Fix possible crash in the garbage collector when it tries to break a reference loop containing a **`memoryview`** object. Now a **`memoryview`** object can only be cleared if there are no buffers that refer it.
- gh-120221 [https://github.com/python/cpython/issues/120221]: asyncio REPL is now again properly recognizing KeyboardInterrupts. Display of exceptions raised in secondary threads is fixed.
- gh-119310 [https://github.com/python/cpython/issues/119310]: Allow the new interactive shell to read history files written with the editline library that use unicode-escaped entries. Patch by aorcajo and Łukasz Langa.
- gh-123572 [https://github.com/python/cpython/issues/123572]: Fix key mappings for various F-keys in Windows for the new REPL. Patch by devdanzin

Library

- gh-123614 [https://github.com/python/cpython/issues/123614]: Add **`turtle.save()`** to easily save Turtle drawings as PostScript files. Patch by Marie Roald and Yngve Mardal Moe.

Core and Builtins

- gh-123339 [https://github.com/python/cpython/issues/123339]: Setting the **`__module__`** attribute for a class now removes the `__firstlineno__` item from the type’s dict, so they will no longer be inconsistent.

- gh-119034 [https://github.com/python/cpython/issues/119034]: Change `<page up>` and `<page down>` keys of the Python REPL to history search forward/backward. Patch by Victor Stinner.

- gh-123562 [https://github.com/python/cpython/issues/123562]: Improve **`SyntaxError`** message for using `case ... as ...` with not a name.

- gh-123545 [https://github.com/python/cpython/issues/123545]: Fix a double decref in rare cases on experimental JIT builds.

- gh-123484 [https://github.com/python/cpython/issues/123484]: Fix `_Py_DebugOffsets` for long objects to be relative to the start of the object rather than the start of a subobject.

- gh-123446 [https://github.com/python/cpython/issues/123446]: Fix empty function name in **`TypeError`** when builtin magic methods are used without the required args.

- gh-123440 [https://github.com/python/cpython/issues/123440]: Improve **`SyntaxError`** message for using `except as` with not a name.

- gh-116017 [https://github.com/python/cpython/issues/116017]: Improved JIT memory consumption by periodically freeing memory used by infrequently-executed code. This change is especially likely to improve the memory footprint of long-running programs.

- gh-123344 [https://github.com/python/cpython/issues/123344]: Add AST optimizations for type parameter defaults.

- gh-123321 [https://github.com/python/cpython/issues/123321]: Prevent Parser/myreadline race condition from segfaulting on multi-threaded use. Patch by Bar Harel and Amit Wienner.

- gh-123177 [https://github.com/python/cpython/issues/123177]: Fix a bug causing stray prompts to appear in the middle of wrapped lines in the new REPL.

- gh-122982 [https://github.com/python/cpython/issues/122982]: Extend the deprecation period for bool inversion (~) by two years.

- gh-123271 [https://github.com/python/cpython/issues/123271]: Make concurrent iterations over the same **`zip()`** iterator safe under free-threading.

- gh-123275 [https://github.com/python/cpython/issues/123275]: Support **`-X gil=1`** and **`PYTHON_GIL=1`** on non-free-threaded builds.

- gh-123177 [https://github.com/python/cpython/issues/123177]: Deactivate line wrap in the Apple Terminal via a ANSI escape code. Patch by Pablo Galindo

- gh-123229 [https://github.com/python/cpython/issues/123229]: Fix valgrind warning by initializing the f-string buffers to 0 in the tokenizer. Patch by Pablo Galindo

- gh-122298 [https://github.com/python/cpython/issues/122298]: Restore printout of GC stats when `gc.set_debug(gc.DEBUG_STATS)` is called. This feature was accidentally removed when implementing incremental GC.

- gh-121804 [https://github.com/python/cpython/issues/121804]: Correctly show error locations when a **`SyntaxError`** is raised in the basic REPL. Patch by Sergey B Kirpichev.

- gh-115776 [https://github.com/python/cpython/issues/115776]: Enables inline values (Python’s equivalent of hidden classes) on any class who’s instances are of a fixed size.
- gh-123142 [https://github.com/python/cpython/issues/123142]: Fix too-wide source location in exception tracebacks coming from broken iterables in comprehensions.
- gh-123048 [https://github.com/python/cpython/issues/123048]: Fix a bug where pattern matching code could emit a **`JUMP_FORWARD`** with no source location.
- gh-118093 [https://github.com/python/cpython/issues/118093]: Break up `CALL_ALLOC_AND_ENTER_INIT` into micro-ops and relax requirement for exact args, in order to increase the amount of code supported by tier 2.
- gh-123123 [https://github.com/python/cpython/issues/123123]: Fix displaying **`SyntaxError`** exceptions covering multiple lines. Patch by Pablo Galindo
- gh-123083 [https://github.com/python/cpython/issues/123083]: Fix a potential use-after-free in `STORE_ATTR_WITH_HINT`.
- gh-123022 [https://github.com/python/cpython/issues/123022]: Fix crash in free-threaded build when calling **`Py_Initialize()`** from a non-main thread.
- gh-118093 [https://github.com/python/cpython/issues/118093]: Add three specializations for **`CALL_KW`**:
    - **`CALL_KW_PY`** for calls to Python functions
    - **`CALL_KW_BOUND_METHOD`** for calls to bound methods
    - **`CALL_KW_NON_PY`** for all other calls

- gh-122821 [https://github.com/python/cpython/issues/122821]: Make sure that branches in `while` statements have consistent offsets for `sys.monitoring`. `while` statements are now compiled with a simple jump at the end of the body, instead of duplicating the test.

- gh-122907 [https://github.com/python/cpython/issues/122907]: Building with `HAVE_DYNAMIC_LOADING` now works as well as it did in 3.12. Existing deficiences will be addressed separately. (See https://github.com/python/cpython/issues/122950.)

- gh-122888 [https://github.com/python/cpython/issues/122888]: Fix crash on certain calls to `str()` with positional arguments of the wrong type. Patch by Jelle Zijlstra.

- gh-118093 [https://github.com/python/cpython/issues/118093]: Improve the experimental JIT's handling of polymorphic code.

- gh-122697 [https://github.com/python/cpython/issues/122697]: Fixed memory leaks at interpreter shutdown in the free-threaded build, and also reporting of leaked memory blocks via `-X showrefcount`.

- gh-116622 [https://github.com/python/cpython/issues/116622]: Fix Android stdout and stderr messages being truncated or lost.

- gh-122527 [https://github.com/python/cpython/issues/122527]: Fix a crash that occurred when a `PyStructSequence` was deallocated after its type's dictionary was cleared by the GC. The type's `tp_basicsize` now accounts for non-sequence fields that aren't included in the `Py_SIZE` of the sequence.

- gh-122445 [https://github.com/python/cpython/issues/122445]: Add only fields which are modified via self.* to `__static_attributes__`.

- gh-122417 [https://github.com/python/cpython/issues/122417]: In the free-threaded build, the reference counts for heap type objects are now partially stored in a distributed manner in per-thread arrays. This reduces contention on the heap type's reference count fields when creating or destroying instances of the same type from multiple threads concurrently.

- gh-116090 [https://github.com/python/cpython/issues/116090]: Fix an issue in JIT builds that prevented some `for` loops from correctly firing `RAISE` monitoring events.

- gh-122300 [https://github.com/python/cpython/issues/122300]: Preserve AST nodes for f-string with single-element format specifiers. Patch by Pablo Galindo

- gh-120906 [https://github.com/python/cpython/issues/120906]: `frame.f_locals` now supports arbitrary hashable objects as keys.

- gh-122239 [https://github.com/python/cpython/issues/122239]: When a `list`, `tuple` or `dict` with too many elements is unpacked, show the actual length in the error message.

- gh-122245 [https://github.com/python/cpython/issues/122245]: Detection of writes to `__debug__` is moved from the compiler's codegen stage to the symtable. This means that these errors are now detected even in code that is optimized away before codegen (such as assertions with the `-O` command line option).

- gh-122234 [https://github.com/python/cpython/issues/122234]: Specializations for sums with float and complex inputs in `sum()` now always use compensated summation. Also, for integer items in above specializations: `PyLong_AsDouble()` is used, instead of `PyLong_AsLongAndOverflow()`. Patch by Sergey B Kirpichev.

- gh-122208 [https://github.com/python/cpython/issues/122208]: Dictionary watchers now only deliver the PyDict_EVENT_ADDED event when the insertion is in a known good state to succeed.
- gh-122160 [https://github.com/python/cpython/issues/122160]: Remove the `BUILD_CONST_KEY_MAP` opcode. Use `BUILD_MAP` instead.
- gh-122029 [https://github.com/python/cpython/issues/122029]: Emit `c_call` events in `sys.setprofile()` when a `PyMethodObject` pointing to a `PyCFunction` is called.
- gh-122026 [https://github.com/python/cpython/issues/122026]: Fix a bug that caused the tokenizer to not correctly identify mismatched parentheses inside f-strings in some situations. Patch by Pablo Galindo
- gh-99108 [https://github.com/python/cpython/issues/99108]: Python’s hashlib now unconditionally uses the vendored HACL* library for Blake2. Python no longer accepts libb2 as an optional dependency for Blake2.

  We refreshed HACL* to the latest version, and now vendor HACL*’s 128-bit and 256-bit wide vector implementations for Blake2, which are used on x86/x64 toolchains when the required CPU features are available at runtime.

  HACL*’s 128-bit wide vector implementation of Blake2 can also run on ARM NEON and Power8, but lacking evidence of a performance gain, these are not enabled (yet).

Library

- gh-121999 [https://github.com/python/cpython/issues/121999]: The default extraction filter for the **`tarfile`** module is now set to **`'data'`**.

Core and Builtins

- gh-121860 [https://github.com/python/cpython/issues/121860]: Fix crash when rematerializing a managed dictionary after it was deleted.
- gh-121795 [https://github.com/python/cpython/issues/121795]: Improve performance of set membership testing, `set.remove()` and `set.discard()` when the argument is a set.
- gh-121814 [https://github.com/python/cpython/issues/121814]: Fixed the SegFault when **`PyEval_SetTrace()`** is used with no Python frame on stack.
- gh-121295 [https://github.com/python/cpython/issues/121295]: Fix PyREPL console getting into a blocked state after interrupting a long paste
- gh-121794 [https://github.com/python/cpython/issues/121794]: Fix bug in free-threaded Python where a resurrected object could lead to a negative ref count assertion failure.
- gh-121657 [https://github.com/python/cpython/issues/121657]: Improve the **`SyntaxError`** message if the user tries to use **`yield from`** outside a function.
- gh-121609 [https://github.com/python/cpython/issues/121609]: Fix pasting of characters containing unicode character joiners in the new REPL. Patch by Marta Gomez Macias
- gh-121297 [https://github.com/python/cpython/issues/121297]: Previously, incorrect usage of **`await`** or asynchronous comprehensions in code removed by the **`-O`** option was

not flagged by the Python compiler. Now, such code raises `SyntaxError`. Patch by Jelle Zijlstra.

- gh-117482 [https://github.com/python/cpython/issues/117482]: Unexpected slot wrappers are no longer created for builtin static types in subinterpreters.
- gh-121562 [https://github.com/python/cpython/issues/121562]: Optimized performance of hex_from_char by replacing switch-case with a lookup table
- gh-121499 [https://github.com/python/cpython/issues/121499]: Fix a bug affecting how multi-line history was being rendered in the new REPL after interacting with the new screen cache. Patch by Pablo Galindo
- gh-121497 [https://github.com/python/cpython/issues/121497]: Fix a bug that was preventing the REPL to correctly respect the history when an input hook was set. Patch by Pablo Galindo
- gh-121012 [https://github.com/python/cpython/issues/121012]: Tier 2 execution now ensures that list iterators remain exhausted, once they become exhausted.
- gh-121439 [https://github.com/python/cpython/issues/121439]: Allow tuples of length 20 in the freelist to be reused.
- gh-121288 [https://github.com/python/cpython/issues/121288]: `ValueError` messages for `list.index()`, `range.index()`, **`deque.index()`**, **`deque.remove()`** and **`ShareableList.index()`** no longer contain the repr of the searched value (which can be arbitrary large) and are consistent with error messages for other `index()` and `remove()` methods.
- gh-121368 [https://github.com/python/cpython/issues/121368]: Fix race condition in `_PyType_Lookup` in the free-threaded build due to a missing memory fence. This could lead to `_PyType_Lookup` returning incorrect results on arm64.
- gh-121149 [https://github.com/python/cpython/issues/121149]: Added specialization for summation of complexes, this also improves accuracy of builtin `sum()` for such inputs. Patch by Sergey B Kirpichev.

- gh-121130 [https://github.com/python/cpython/issues/121130]: Fix f-strings with debug expressions in format specifiers. Patch by Pablo Galindo

Library

- gh-121381 [https://github.com/python/cpython/issues/121381]: Remove `subprocess._USE_VFORK` escape hatch code and documentation. It was added just in case, and doesn't have any known cases that require it.

Core and Builtins

- gh-119726 [https://github.com/python/cpython/issues/119726]: Optimize code layout for calls to C functions from the JIT on AArch64. Patch by Diego Russo.
- gh-121115 [https://github.com/python/cpython/issues/121115]: **`PyLong_AsNativeBytes()`** no longer uses **`__index__()`** methods by default. The `Py_ASNATIVEBYTES_ALLOW_INDEX` flag has been added to allow it.
- gh-120838 [https://github.com/python/cpython/issues/120838]: **`Py_Finalize()`** and **`Py_FinalizeEx()`** now always run with the main interpreter active.
- gh-113433 [https://github.com/python/cpython/issues/113433]: Subinterpreters now get cleaned up automatically during runtime finalization.
- gh-119726 [https://github.com/python/cpython/issues/119726]: Improve the speed and memory use of C function calls from JIT code on AArch64. Patch by Diego Russo
- gh-116017 [https://github.com/python/cpython/issues/116017]: Simplify the warmup mechanism used for "side exits" in JIT code, resulting in slightly better performance and slightly lower memory usage for most platforms.
- gh-98442 [https://github.com/python/cpython/issues/98442]: Fix too wide source locations of the cleanup instructions of a with statement.

- gh-120754 [https://github.com/python/cpython/issues/120754]: Reduce the number of system calls invoked when reading a whole file (ex. `open('a.txt').read()`). For a sample program that reads the contents of the 400+ `.rst` files in the cpython repository `Doc` folder, there is an over 10% reduction in system call count.
- gh-119462 [https://github.com/python/cpython/issues/119462]: Make sure that invariants of type versioning are maintained: * Superclasses always have their version number assigned before subclasses * The version tag is always zero if the tag is not valid. * The version tag is always non-if the tag is valid.
- gh-120437 [https://github.com/python/cpython/issues/120437]: Fix `_CHECK_STACK_SPACE` optimization problems introduced in gh-118322 [https://github.com/python/cpython/issues/118322].
- gh-120722 [https://github.com/python/cpython/issues/120722]: Correctly set the bytecode position on return instructions within lambdas. Patch by Jelle Zijlstra.
- gh-120367 [https://github.com/python/cpython/issues/120367]: Fix bug where compiler creates a redundant jump during pseudo-op replacement. Can only happen with a synthetic AST that has a try on the same line as the instruction following the exception handler.
- gh-120507 [https://github.com/python/cpython/issues/120507]: Remove the `BEFORE_WITH` and `BEFORE_ASYNC_WITH` instructions. Add the new **`LOAD_SPECIAL`** instruction. Generate code for `with` and `async with` statements using the new instruction.
- gh-113993 [https://github.com/python/cpython/issues/113993]: Strings interned with **`sys.intern()`** are again garbage-collected when no longer used, as per the documentation. Strings interned with the C function **`PyUnicode_InternInPlace()`** are still immortal. Internals of the string interning mechanism have been changed. This may affect performance and identities of **`str`** objects.

Library

- gh-120485 [https://github.com/python/cpython/issues/120485]: Add an override of `allow_reuse_port` on classes subclassing `socketserver.TCPServer` where `allow_reuse_address` is also overridden.

Core and Builtins

- gh-120384 [https://github.com/python/cpython/issues/120384]: Fix an array out of bounds crash in `list_ass_subscript`, which could be invoked via some specifically tailored input: including concurrent modification of a list object, where one thread assigns a slice and another clears it.
- gh-120367 [https://github.com/python/cpython/issues/120367]: Fix crash in compiler on code with redundant NOPs and JUMPs which show up after exception handlers are moved to the end of the code.

Library

- gh-120380 [https://github.com/python/cpython/issues/120380]: Fix Python implementation of **`pickle.Pickler`** for **`bytes`** and **`bytearray`** objects when using protocol version 5. Patch by Bénédikt Tran.

Core and Builtins

- gh-120397 [https://github.com/python/cpython/issues/120397]: Improve the throughput by up to two times for the **`str.count()`**, **`bytes.count()`** and **`bytearray.count()`** methods for counting single characters.

- gh-120221 [https://github.com/python/cpython/issues/120221]: Deliver real signals on Ctrl-C and Ctrl-Z in the new REPL. Patch by Pablo Galindo

- gh-120346 [https://github.com/python/cpython/issues/120346]: Respect `PYTHON_BASIC_REPL` when running in interactive inspect mode (`python -i`). Patch by Pablo Galindo

- gh-93691 [https://github.com/python/cpython/issues/93691]: Fix source locations of instructions generated for the iterator of a for statement.

- gh-120198 [https://github.com/python/cpython/issues/120198]: Fix a crash when multiple threads read and write to the same `__class__` of an object concurrently.

- gh-120298 [https://github.com/python/cpython/issues/120298]: Fix use-after free in `list_richcompare_impl` which can be invoked via some specifically tailored evil input.

- gh-119666 [https://github.com/python/cpython/issues/119666]: Fix a compiler crash in the case where two comprehensions in class scope both reference `__class__`.

- gh-119726 [https://github.com/python/cpython/issues/119726]: JIT: Re-use trampolines on AArch64 when creating stencils. Patch by Diego Russo

- gh-120225 [https://github.com/python/cpython/issues/120225]: Fix crash in compiler on empty block at end of exception handler.

- gh-93691 [https://github.com/python/cpython/issues/93691]: Fix source locations of instructions generated for with statements.

- gh-120097 [https://github.com/python/cpython/issues/120097]: `FrameLocalsProxy` now subclasses `collections.abc.Mapping` and can be matched as a mapping in `match` statements

- gh-120080 [https://github.com/python/cpython/issues/120080]: Direct call to the **`int.__round__()`** now accepts `None` as a valid argument.

- gh-119933 [https://github.com/python/cpython/issues/119933]: Improve **`SyntaxError`** messages for invalid expressions in a type parameters bound, a type parameter constraint tuple or a default type parameter. Patch by Bénédikt Tran.

- gh-119724 [https://github.com/python/cpython/issues/119724]: Reverted improvements to error messages for `elif`/`else` statements not matching any valid statements, which made in hard to locate the syntax errors inside those `elif`/`else` blocks.

- gh-119879 [https://github.com/python/cpython/issues/119879]: String search is now slightly faster for certain cases. It now utilizes last character gap (good suffix rule) for two-way periodic needles.

- gh-119842 [https://github.com/python/cpython/issues/119842]: Honor **`PyOS_InputHook()`** in the new REPL. Patch by Pablo Galindo

- gh-119180 [https://github.com/python/cpython/issues/119180]: **`classmethod()`** and **`staticmethod()`** now wrap the **`__annotations__`** and **`__annotate__`** attributes of their underlying callable lazily. See **PEP 649** [https://peps.python.org/pep-0649/]. Patch by Jelle Zijlstra.

- gh-119821 [https://github.com/python/cpython/issues/119821]: Fix execution of annotation scopes within classes when `globals` is set to a non-dict. Patch by Jelle Zijlstra.

- gh-118934 [https://github.com/python/cpython/issues/118934]: Make `PyEval_GetLocals` return borrowed reference

- gh-119740 [https://github.com/python/cpython/issues/119740]: Remove the previously-deprecated delegation of **`int()`** to **`__trunc__()`**.

- gh-119689 [https://github.com/python/cpython/issues/119689]: Generate stack effect metadata for pseudo instructions from bytecodes.c.

- gh-109218 [https://github.com/python/cpython/issues/109218]: **`complex()`** accepts now a string only as a positional argument. Passing a complex number as the “real” or “imag” argument is deprecated; it should only be passed as a single positional argument.

- gh-119548 [https://github.com/python/cpython/issues/119548]: Add a `clear` command to the REPL. Patch by Pablo Galindo

- gh-111999 [https://github.com/python/cpython/issues/111999]: Fix the signature of **`str.format_map()`**.

- gh-119560 [https://github.com/python/cpython/issues/119560]: An invalid assert in beta 1 has been removed. The assert would fail if `PyState_FindModule()` was used in an extension module’s init function before the module def had been initialized.

- gh-119369 [https://github.com/python/cpython/issues/119369]: Fix deadlock during thread deletion in free-threaded build, which could occur when the GIL was enabled at runtime.

- gh-119525 [https://github.com/python/cpython/issues/119525]: Fix deadlock involving `_PyType_Lookup()` cache in the free-threaded build when the GIL is dynamically enabled at runtime.

- gh-119258 [https://github.com/python/cpython/issues/119258]: Eliminate type version guards in the tier two interpreter.

Note that setting the `tp_version_tag` manually (which has never been supported) may result in crashes.

- gh-119311 [https://github.com/python/cpython/issues/119311]: Fix bug where names are unexpectedly mangled in the bases of generic classes.

- gh-119395 [https://github.com/python/cpython/issues/119395]: Fix bug where names appearing after a generic class are mangled as if they are in the generic class.

- gh-119372 [https://github.com/python/cpython/issues/119372]: Correct invalid corner cases in complex division (resulted in `(nan+nanj)` output), e.g. `1/complex('(inf+infj)')`. Patch by Sergey B Kirpichev.

- gh-119180 [https://github.com/python/cpython/issues/119180]: Evaluation of annotations is now deferred. See **PEP 649** [https://peps.python.org/pep-0649/] for details.

- gh-119180 [https://github.com/python/cpython/issues/119180]: Replace **`LOAD_ASSERTION_ERROR`** opcode with **`LOAD_COMMON_CONSTANT`** and add support for **`NotImplementedError`**.

- gh-119213 [https://github.com/python/cpython/issues/119213]: Non-builtin modules built with argument clinic were crashing if used in a subinterpreter before the main interpreter. The objects that were causing the problem by leaking between interpreters carelessly have been fixed.

- gh-119011 [https://github.com/python/cpython/issues/119011]: Fixes `type.__type_params__` to return an empty tuple instead of a descriptor.

- gh-118692 [https://github.com/python/cpython/issues/118692]: Avoid creating unnecessary **`StopIteration`** instances for monitoring.

- gh-119180 [https://github.com/python/cpython/issues/119180]: Add an `__annotate__` attribute to functions, classes, and modules as part of **PEP 649** [https://peps.python.org/pep-0649/]. Patch by Jelle Zijlstra.

- gh-119049 [https://github.com/python/cpython/issues/119049]: Fix displaying the source line for warnings created by the C API if the **`warnings`** module had not yet been imported.

- gh-119057 [https://github.com/python/cpython/issues/119057]: Improve **`ZeroDivisionError`** error message. Now, all error messages are harmonized: all `/`, `//`, and `%` operations just use “division by zero” message. And `0 ** -1` operation uses “zero to a negative power”.

- gh-118844 [https://github.com/python/cpython/issues/118844]: Fix build failures when configuring with both `--disable-gil` and `--enable-experimental-jit`.

- gh-118921 [https://github.com/python/cpython/issues/118921]: Add `copy()` method for `FrameLocalsProxy` which returns a snapshot `dict` for local variables.

- gh-117657 [https://github.com/python/cpython/issues/117657]: Fix data races on the field that stores a pointer to the interpreter’s main thread that occur in free-threaded builds.

- gh-118750 [https://github.com/python/cpython/issues/118750]: If the C version of the `decimal` module is available, `int(str)` now uses it to supply an asymptotically much faster conversion. However, this only applies if the string contains over about 2 million digits.

- gh-118767 [https://github.com/python/cpython/issues/118767]: Using **`NotImplemented`** in a boolean context now raises **`TypeError`**. Contributed by Jelle Zijlstra.

- gh-118561 [https://github.com/python/cpython/issues/118561]: Fix race condition in free-threaded build where **`list.extend()`** could expose uninitialised memory to concurrent readers.

- gh-117139 [https://github.com/python/cpython/issues/117139]: Convert the Python evaluation stack to use internal stack references. The purpose is to support tagged pointers. In **PEP 703** [https://peps.python.org/pep-0703/], this will allow for its form of deferred reference counting. For both the default and free-threaded builds, this sets up the infrastructure for unboxed integers in the future.

Library

- gh-115801 [https://github.com/python/cpython/issues/115801]: Raise `TypeError` when passing a string to **`difflib.unified_diff()`** and **`difflib.context_diff()`**.

Core and Builtins

- gh-117558 [https://github.com/python/cpython/issues/117558]: Improve error messages when a string, bytes or bytearray object of length 1 is expected.
- gh-117195 [https://github.com/python/cpython/issues/117195]: Avoid assertion failure for debug builds when calling `object.__sizeof__(1)`
- gh-116022 [https://github.com/python/cpython/issues/116022]: Improve the **`__repr__()`** output of **`AST`** nodes.
- gh-114091 [https://github.com/python/cpython/issues/114091]: Changed the error message for awaiting something that can’t be awaited from “object <type> can’t be used in

an await expression” to “’<type>’ object can’t be awaited”.

- gh-113190 [https://github.com/python/cpython/issues/113190]: `Py_Finalize()` now deletes all interned strings.
- gh-84978 [https://github.com/python/cpython/issues/84978]: Add class methods `float.from_number()` and `complex.from_number()`.
- gh-95144 [https://github.com/python/cpython/issues/95144]: Improve the error message from `a in b` when `b` is not a container to mention the term “container”.
- bpo-24766 [https://bugs.python.org/issue?@action=redirect&bpo=24766]: Fix handling of `doc` argument to subclasses of `property`.

C API

- gh-123961 [https://github.com/python/cpython/issues/123961]: Convert the curses.window static type exposed by the **`PyCursesWindow_Type`** macro in `Include/py_curses.h` to a heap type. Patch by Bénédikt Tran.

- gh-124502 [https://github.com/python/cpython/issues/124502]: Add `PyUnicode_Equal()` function to the limited C API: test if two strings are equal. Patch by Victor Stinner.

- gh-124296 [https://github.com/python/cpython/issues/124296]: `PyDictObject` no longer maintains a private version tag field `ma_version_tag` per **PEP 699** [https://peps.python.org/pep-0699/]. This field was originally added in Python 3.6 (**PEP 509** [https://peps.python.org/pep-0509/]) and deprecated in Python 3.12.

- gh-124160 [https://github.com/python/cpython/issues/124160]: Fix crash when importing modules containing state and single-phase initialization in a subinterpreter.

- gh-124153 [https://github.com/python/cpython/issues/124153]: Add `PyType_GetBaseByToken()` and `Py_tp_token` slot for easier type checking, related to **PEP 489** [https://peps.python.org/pep-0489/] and **PEP 630** [https://peps.python.org/pep-0630/].

- gh-124127 [https://github.com/python/cpython/issues/124127]: In the limited C API 3.14 and newer, `Py_REFCNT()` is now implemented as an opaque function call to hide implementation details. Patch by Victor Stinner.

- gh-123880 [https://github.com/python/cpython/issues/123880]: Fixed a bug that prevented circular imports of extension modules that use single-phase initialization.

- gh-123909 [https://github.com/python/cpython/issues/123909]: `PyType_FromSpec()`, `PyType_FromSpecWithBases()` and `PyType_FromModuleAndSpec()` will now fail if the metaclass of the new type has custom `tp_new`.

- gh-107954 [https://github.com/python/cpython/issues/107954]: Add functions to configure the Python initialization (**PEP 741** [https://peps.python.org/pep-0741/]):

  - `PyInitConfig_Create()`
  - `PyInitConfig_Free()`
  - `PyInitConfig_GetError()`
  - `PyInitConfig_GetExitCode()`
  - `PyInitConfig_HasOption()`
  - `PyInitConfig_GetInt()`
  - `PyInitConfig_GetStr()`
  - `PyInitConfig_GetStrList()`
  - `PyInitConfig_FreeStrList()`
  - `PyInitConfig_SetInt()`
  - `PyInitConfig_SetStr()`
  - `PyInitConfig_SetStrList()`

    - `PyInitConfig_AddModule()`
    - `Py_InitializeFromInitConfig()`

  Patch by Victor Stinner.

- gh-107954 [https://github.com/python/cpython/issues/107954]: Add functions to get and set the current runtime Python configuration:

    - `PyConfig_Get()`
    - `PyConfig_GetInt()`
    - `PyConfig_Set()`
    - `PyConfig_Names()`

  Patch by Victor Stinner.

- gh-123465 [https://github.com/python/cpython/issues/123465]: `Py_RELATIVE_OFFSET` is now allowed in `PyMemberDef` for the special offset member `"__vectorcalloffset__"`, as well as the discouraged special offset members `"__dictoffset__"` and `"__weaklistoffset__"`

- gh-100554 [https://github.com/python/cpython/issues/100554]: Added a slot `Py_tp_vectorcall` to set `tp_vectorcall` via the `PyType_FromSpec()` function family. Limited API extensions can use this feature to provide more efficient vector call-based implementation of `__new__` and `__init__`.

- gh-122854 [https://github.com/python/cpython/issues/122854]: Add `Py_HashBuffer()` to compute and return the hash value of a buffer. Patch by Antoine Pitrou and Victor Stinner.

- gh-122728 [https://github.com/python/cpython/issues/122728]: Fix `PyEval_GetLocals()` to avoid `SystemError` ("bad argument to internal function"). Patch by Victor Stinner.

- gh-116622 [https://github.com/python/cpython/issues/116622]: Make **`PyObject_Print`** work around a bug in Android and OpenBSD which prevented it from throwing an exception when trying to write to a read-only stream.

- gh-105201 [https://github.com/python/cpython/issues/105201]: Add **`PyIter_NextItem()`** to replace **`PyIter_Next()`**, which has an ambiguous return value. Patch by Irit Katriel and Erlend Aasland.

- gh-121489 [https://github.com/python/cpython/issues/121489]: Export private **`_PyBytes_Join()`** again.

- gh-121645 [https://github.com/python/cpython/issues/121645]: Add **`PyBytes_Join(sep, iterable)`** function, similar to `sep.join(iterable)` in Python. Patch by Victor Stinner.

- gh-89364 [https://github.com/python/cpython/issues/89364]: Export the **`PySignal_SetWakeupFd()`** function. Previously, the function was documented but it couldn’t be used in 3rd party code. Patch by Victor Stinner.

- gh-113993 [https://github.com/python/cpython/issues/113993]: **`PyUnicode_InternInPlace()`** no longer prevents its argument from being garbage collected.

  Several functions that take `char *` are now documented as possibly preventing string objects from being garbage collected; refer to their documentation for details: **`PyUnicode_InternFromString()`**, **`PyDict_SetItemString()`**, **`PyObject_SetAttrString()`**, **`PyObject_DelAttrString()`**, **`PyUnicode_InternFromString()`**, and `PyModule_Add*` convenience functions.

- gh-113601 [https://github.com/python/cpython/issues/113601]: Removed debug build assertions related to interning strings, which were falsely triggered by stable ABI extensions.

- gh-112136 [https://github.com/python/cpython/issues/112136]: Restore the private `_PyArg_Parser` structure and the private `_PyArg_ParseTupleAndKeywordsFast()` function, previously removed in Python 3.13 alpha 1. Patch by Victor Stinner.

- gh-120642 [https://github.com/python/cpython/issues/120642]: Remove the private `_Py_CODEUNIT` type from the public C API. The internal `pycore_code.h` header should now be used to get this internal type. Patch by Victor Stinner.

- gh-120858 [https://github.com/python/cpython/issues/120858]: **`PyDict_Next()`** no longer locks the dictionary in the free-threaded build. The locking needs to be done by the caller around the entire iteration loop.

- gh-120642 [https://github.com/python/cpython/issues/120642]: Remove the following unstable functions:

    - `PyUnstable_Replace_Executor()`
    - `PyUnstable_SetOptimizer()`
    - `PyUnstable_GetOptimizer()`
    - `PyUnstable_GetExecutor()`
    - `PyUnstable_Optimizer_NewCounter()`
    - `PyUnstable_Optimizer_NewUOpOptimizer()`

  Patch by Victor Stinner.

- gh-120389 [https://github.com/python/cpython/issues/120389]: Add new functions to convert C `<stdint.h>` numbers from/to Python **`int`**:

    - **`PyLong_FromInt32()`**
    - **`PyLong_FromUInt32()`**

  - `PyLong_FromInt64()`
  - `PyLong_FromUInt64()`
  - `PyLong_AsInt32()`
  - `PyLong_AsUInt32()`
  - `PyLong_AsInt64()`
  - `PyLong_AsUInt64()`

  Patch by Victor Stinner.

- gh-120600 [https://github.com/python/cpython/issues/120600]: In the limited C API 3.14 and newer, `Py_TYPE()` is now implemented as an opaque function call to hide implementation details. Patch by Victor Stinner.

- gh-70278 [https://github.com/python/cpython/issues/70278]: `PyUnicode_FromFormat()` no longer produces the ending `\ufffd` character for truncated C string when use precision with `%s` and `%V`. It now truncates the string before the start of truncated multibyte sequences.

- gh-119771 [https://github.com/python/cpython/issues/119771]: Set `errno` in `_Py_c_pow()` on overflows. Patch by Sergey B Kirpichev.

- gh-119182 [https://github.com/python/cpython/issues/119182]: Add a new `PyUnicodeWriter` API to create a Python `str` object:

  - `PyUnicodeWriter_Create()`.
  - `PyUnicodeWriter_Discard()`.
  - `PyUnicodeWriter_Finish()`.
  - `PyUnicodeWriter_WriteChar()`.
  - `PyUnicodeWriter_WriteUTF8()`.
  - `PyUnicodeWriter_WriteUCS4()`.
  - `PyUnicodeWriter_WriteWideChar()`.
  - `PyUnicodeWriter_WriteStr()`.
  - `PyUnicodeWriter_WriteRepr()`.
  - `PyUnicodeWriter_WriteSubstring()`.

  - `PyUnicodeWriter_Format()`.
  - `PyUnicodeWriter_DecodeUTF8Stateful()`.

  Patch by Victor Stinner.

- gh-119613 [https://github.com/python/cpython/issues/119613]: Soft deprecate the `Py_MEMCPY` macro: use directly `memcpy()` instead. Patch by Victor Stinner.

- gh-119775 [https://github.com/python/cpython/issues/119775]: Creating `immutable types` with mutable bases was deprecated since 3.12 and now raises a `TypeError`.

- gh-119585 [https://github.com/python/cpython/issues/119585]: Fix crash when a thread state that was created by `PyGILState_Ensure()` calls a destructor that during `PyThreadState_Clear()` that calls back into `PyGILState_Ensure()` and `PyGILState_Release()`. This might occur when in the free-threaded build or when using thread-local variables whose destructors call `PyGILState_Ensure()`.

- gh-119613 [https://github.com/python/cpython/issues/119613]: Macros `Py_IS_NAN`, `Py_IS_INFINITY` and `Py_IS_FINITE` are soft deprecated.

- gh-119336 [https://github.com/python/cpython/issues/119336]: Restore the removed `_PyLong_NumBits()` function. It is used by the pywin32 project. Patch by Ethan Smith

- gh-119344 [https://github.com/python/cpython/issues/119344]: The critical section API is now public as part of the non-limited C API.

- gh-119333 [https://github.com/python/cpython/issues/119333]: Add `PyContext_AddWatcher()` and `PyContext_ClearWatcher()` APIs to register callbacks to receive notification on enter and exit of context objects.

- gh-119247 [https://github.com/python/cpython/issues/119247]: Added `Py_BEGIN_CRITICAL_SECTION_SEQUENCE_FAST` and `Py_END_CRITICAL_SECTION_SEQUENCE_FAST` macros to make it possible to use PySequence_Fast APIs safely when free-threaded, and update str.join to work without the GIL using them.

- gh-111389 [https://github.com/python/cpython/issues/111389]: Add **`PyHASH_MULTIPLIER`** constant: prime multiplier used in string and various other hashes. Patch by Victor Stinner.

- gh-118771 [https://github.com/python/cpython/issues/118771]: Several C declarations with names that didn't start with the `Py` or `_Py` prefixes, which were added by mistake in 3.13 alpha and beta releases, were moved to internal headers.

- gh-116984 [https://github.com/python/cpython/issues/116984]: Make mimalloc includes relative to the current file to avoid embedders or extensions needing to include `Internal/mimalloc` if they are already including internal CPython headers.

- gh-118789 [https://github.com/python/cpython/issues/118789]: Add **`PyUnstable_Object_ClearWeakRefsNoCallbacks()`**, which clears weakrefs without calling their callbacks.

- gh-118789 [https://github.com/python/cpython/issues/118789]: Restore `_PyWeakref_ClearRef` that was previously removed in Python 3.13 alpha 1.

- gh-117511 [https://github.com/python/cpython/issues/117511]: Make the **`PyMutex`** public in the non-limited C API.

- gh-116560 [https://github.com/python/cpython/issues/116560]: Add **`PyLong_GetSign()`** function. Patch by Sergey B Kirpichev.

- gh-97588 [https://github.com/python/cpython/issues/97588]: Fix creating bitfields in **`ctypes`** structures and unions. Fields no longer overlap.

- gh-87135 [https://github.com/python/cpython/issues/87135]: Attempting to acquire the GIL after runtime finalization has begun in a different thread now causes the thread to hang rather than terminate, which avoids potential crashes or memory corruption caused by attempting to terminate a thread that is running code not specifically designed to support termination. In most cases this hanging is harmless since the process will soon exit anyway.

  The `PyThread_exit_thread` function is now deprecated. Its behavior is inconsistent across platforms, and it can only be used safely in the unlikely case that every function in the entire call stack has been designed to support the platform-dependent termination mechanism. It is recommended that users of this function change their design to not require thread termination. In the unlikely case that thread termination is needed and can be done safely, users may migrate to calling platform-specific APIs such as `pthread_exit` (POSIX) or `_endthreadex` (Windows) directly.

- bpo-34008 [https://bugs.python.org/issue?@action=redirect&bpo=34008]: Added `Py_IsInitialized` to the list of APIs that are safe to call before the interpreter is initialized, and updated the embedding tests to cover it.

Build

- gh-125269 [https://github.com/python/cpython/issues/125269]: Fix detection of whether `-latomic` is needed when cross-compiling CPython using the configure script.
- gh-123990 [https://github.com/python/cpython/issues/123990]: Remove `WITH_FREELISTS` macro and `--without-freelists` build configuration
- gh-124102 [https://github.com/python/cpython/issues/124102]: Update internal documentation under PCbuild, so it now correctly states that Windows requires VS2017 or later and Python 3.10 or later
- gh-124043 [https://github.com/python/cpython/issues/124043]: Building using **`--with-trace-refs`** is (temporarily) disallowed when the GIL is disabled.
- gh-123418 [https://github.com/python/cpython/issues/123418]: Updated Android build to use OpenSSL 3.0.15.
- gh-123297 [https://github.com/python/cpython/issues/123297]: Propagate the value of `LDFLAGS` to `LDCXXSHARED` in **`sysconfig`**. Patch by Pablo Galindo
- gh-121634 [https://github.com/python/cpython/issues/121634]: Allow for specifying the target compile triple for WASI.
- gh-122578 [https://github.com/python/cpython/issues/122578]: Use WASI SDK 24 for testing.
- gh-116622 [https://github.com/python/cpython/issues/116622]: Rename build variable `MODULE_LDFLAGS` back to `LIBPYTHON`, as it’s used by package build systems (e.g. Meson).
- gh-118943 [https://github.com/python/cpython/issues/118943]: Fix an issue where the experimental JIT could be built several times by the `make regen-all` target, leading to possible race conditions on heavily parallelized builds.
- gh-121996 [https://github.com/python/cpython/issues/121996]: Introduce ./configure –disable-safety and –enable-slower-safety options. Patch by Donghee Na.
- gh-120522 [https://github.com/python/cpython/issues/120522]: Added a **`--with-app-store-compliance`** option to patch out known issues with macOS/iOS App Store review processes.

- gh-120371 [https://github.com/python/cpython/issues/120371]: Support WASI SDK 22 by explicitly skipping functions that are just stubs in wasi-libc.
- gh-121731 [https://github.com/python/cpython/issues/121731]: Fix mimalloc compile error on GNU/Hurd
- gh-121487 [https://github.com/python/cpython/issues/121487]: Fix deprecation warning for ATOMIC_VAR_INIT in mimalloc.
- gh-121467 [https://github.com/python/cpython/issues/121467]: Fix a Makefile bug that prevented mimalloc header files from being installed.
- gh-121103 [https://github.com/python/cpython/issues/121103]: On POSIX systems, excluding macOS framework installs, the lib directory for the free-threaded build now includes a “t” suffix to avoid conflicts with a co-located default build installation.
- gh-120831 [https://github.com/python/cpython/issues/120831]: The default minimum iOS version was increased to 13.0.
- gh-121082 [https://github.com/python/cpython/issues/121082]: Fix build failure when the developer use `--enable-pystats` arguments in configuration command after #118450.
- gh-120671 [https://github.com/python/cpython/issues/120671]: Fix failing configure tests due to a missing space when appending to CFLAGS.
- gh-120602 [https://github.com/python/cpython/issues/120602]: Correctly handle LLVM installs with `LLVM_VERSION_SUFFIX` when building with `--enable-experimental-jit`.
- gh-120688 [https://github.com/python/cpython/issues/120688]: On WASI in debug mode, Python is now built with compiler flag `-O3` instead of `-Og`, to support more recursive calls. Patch by Victor Stinner.
- gh-118943 [https://github.com/python/cpython/issues/118943]: Fix a possible race condition affecting parallel builds configured with `--enable-experimental-jit`, in which **`FileNotFoundError`** could be caused by another process already moving `jit_stencils.h.new` to `jit_stencils.h`.

- gh-120326 [https://github.com/python/cpython/issues/120326]: On Windows, fix build error when `--disable-gil` and `--experimental-jit` options are combined.
- gh-120291 [https://github.com/python/cpython/issues/120291]: Make the `python-config` shell script compatible with non-bash shells.
- gh-113565 [https://github.com/python/cpython/issues/113565]: Improve **`curses`** and **`curses.panel`** dependency checks in **configure**.
- gh-119729 [https://github.com/python/cpython/issues/119729]: On POSIX systems, the pkg-config (`.pc`) filenames now include the ABI flags, which may include debug (“d”) and free-threaded (“t”). For example: * `python-3.14.pc` (default, non-debug build) * `python-3.14d.pc` (default, debug build) * `python-3.14t.pc` (free-threaded build)
- gh-119400 [https://github.com/python/cpython/issues/119400]: `make_ssl_certs`, the script that prepares certificate data for the test suite, now allows specifying expiration dates.
- gh-115119 [https://github.com/python/cpython/issues/115119]: Fall back to the bundled libmpdec if a system version cannot be found.
- gh-119132 [https://github.com/python/cpython/issues/119132]: Update **`sys.version`** to identify whether the build is default build or free-threading build. Patch By Donghee Na.
- gh-118836 [https://github.com/python/cpython/issues/118836]: Fix an `AssertionError` when building with `--enable-experimental-jit` and the compiler emits a `SHT_NOTE` section.
- gh-118943 [https://github.com/python/cpython/issues/118943]: Fix a possible race condition affecting parallel builds configured with `--enable-experimental-jit`, in which compilation errors could be caused by an incompletely-generated header file.

Python 3.13.0 beta 1

Release date: 2024-05-08

Security

- gh-116741 [https://github.com/python/cpython/issues/116741]: Update bundled libexpat to 2.6.2
- gh-117233 [https://github.com/python/cpython/issues/117233]: Detect BLAKE2, SHA3, Shake, & truncated SHA512 support in the OpenSSL-ish libcrypto library at build time. This allows `hashlib` to be used with libraries that do not to support every algorithm that upstream OpenSSL does.

Core and Builtins

- gh-118414 [https://github.com/python/cpython/issues/118414]: Add instrumented opcodes to YIELD_VALUE assertion for tracing cases.

- gh-117953 [https://github.com/python/cpython/issues/117953]: When a builtin or extension module is imported for the first time, while a subinterpreter is active, the module’s init function is now run by the main interpreter first before import continues in the subinterpreter. Consequently, single-phase init modules now fail in an isolated subinterpreter without the init function running under that interpreter, whereas before it would run under the subinterpreter *before* failing, potentially leaving behind global state and callbacks and otherwise leaving the module in an inconsistent state.

- gh-117549 [https://github.com/python/cpython/issues/117549]: Don’t use designated initializer syntax in inline functions in internal headers. They cause problems for C++ or MSVC users who aren’t yet using the latest C++ standard (C++20). While internal, pycore_backoff.h, is included (indirectly, via pycore_code.h) by some key 3rd party software that does so for speed.

- gh-95382 [https://github.com/python/cpython/issues/95382]: Improve performance of **`json.dumps()`** and **`json.dump()`** when using the argument *indent*. Depending on the data the encoding using **`json.dumps()`** with *indent* can be up to 2 to 3 times faster.

- gh-116322 [https://github.com/python/cpython/issues/116322]: In `--disable-gil` builds, the GIL will be enabled while loading C extension modules. If the module indicates that it supports running without the GIL, the GIL will be disabled once loading is complete. Otherwise, the GIL will remain enabled for the remainder of the interpreter’s lifetime. This behavior does not apply if the GIL has been explicitly enabled or disabled with `PYTHON_GIL` or `-Xgil`.

- gh-118513 [https://github.com/python/cpython/issues/118513]: Fix incorrect **`UnboundLocalError`** when two comprehensions in the same function both reference the same name, and in one comprehension the name is bound while in the other it’s an implicit global.

- gh-118518 [https://github.com/python/cpython/issues/118518]: Allow the Linux perf support to work without frame pointers using perf’s advanced JIT support. The feature is activated when using the `PYTHON_PERF_JIT_SUPPORT` environment variable or when running Python with `-Xperf_jit`. Patch by Pablo Galindo.

- gh-117514 [https://github.com/python/cpython/issues/117514]: Add `sys._is_gil_enabled()` function that returns whether the GIL is currently enabled. In the default build it always returns `True` because the GIL is always enabled. In the free-threaded build, it may return `True` or `False`.

- gh-118164 [https://github.com/python/cpython/issues/118164]: Break a loop between the Python implementation of the **`decimal`** module and the Python code for integer to string conversion. Also optimize integer to string conversion for values in the range from 9_000 to 135_000 decimal digits.

- gh-118473 [https://github.com/python/cpython/issues/118473]: Fix **`sys.set_asyncgen_hooks()`** not to be partially set when raising **`TypeError`**.

- gh-118465 [https://github.com/python/cpython/issues/118465]: Compiler populates the new `__firstlineno__` field on a class with the line number of the first line of the class definition.

- gh-118492 [https://github.com/python/cpython/issues/118492]: Fix an issue where the type cache can expose a previously accessed attribute when a finalizer is run.

- gh-117714 [https://github.com/python/cpython/issues/117714]: update `async_generator.athrow().close()` and `async_generator.asend().close()` to close their section of the underlying async generator

- gh-111201 [https://github.com/python/cpython/issues/111201]: The interactive interpreter is now implemented in Python, which allows for a number of new features like colors, multiline input, history viewing, and paste mode. Contributed by Pablo Galindo, Łukasz Langa and

Lysandros Nikolaou based on code from the PyPy project.

- gh-74929 [https://github.com/python/cpython/issues/74929]: Implement PEP 667: converted **`FrameType.f_locals`** and **`PyFrame_GetLocals()`** to return a write-through proxy object when the frame refers to a function or comprehension.

- gh-116767 [https://github.com/python/cpython/issues/116767]: Fix crash in compiler on ‘async with’ that has many context managers.

- gh-118335 [https://github.com/python/cpython/issues/118335]: Change how to use the tier 2 interpreter. Instead of running Python with `-X uops` or setting the environment variable `PYTHON_UOPS=1`, this choice is now made at build time by configuring with `--enable-experimental-jit=interpreter`.

  Beware! This changes the environment variable to enable or disable micro-ops to `PYTHON_JIT`. The old `PYTHON_UOPS` is no longer used.

- gh-118306 [https://github.com/python/cpython/issues/118306]: Update JIT compilation to use LLVM 18

- gh-118160 [https://github.com/python/cpython/issues/118160]: Annotation scopes within classes can now contain comprehensions. However, such comprehensions are not inlined into their parent scope at runtime. Patch by Jelle Zijlstra.

- gh-118272 [https://github.com/python/cpython/issues/118272]: Fix bug where `generator.close` does not free the generator frame’s locals.

- gh-118216 [https://github.com/python/cpython/issues/118216]: Don’t consider **`__future__`** imports with dots before the module name.

- gh-118074 [https://github.com/python/cpython/issues/118074]: Make sure that the Executor objects in the COLD_EXITS array aren’t assumed to be GC-able (which would access bytes outside the object).

- gh-107674 [https://github.com/python/cpython/issues/107674]: Lazy load frame line number to improve performance of tracing

- gh-118082 [https://github.com/python/cpython/issues/118082]: Improve **`SyntaxError`** message for imports without names, like in `from x import` and `import` cases. It now points out to users that **`import`** expects at least one name after it.

- gh-118090 [https://github.com/python/cpython/issues/118090]: Improve **`SyntaxError`** message for empty type param brackets.

- gh-117958 [https://github.com/python/cpython/issues/117958]: Added a `get_jit_code()` method to access JIT compiled machine code from the UOp Executor when the experimental JIT is enabled. Patch by Anthony Shaw.

- gh-117901 [https://github.com/python/cpython/issues/117901]: Add option for compiler’s codegen to save nested instruction sequences for introspection.

- gh-116622 [https://github.com/python/cpython/issues/116622]: Redirect stdout and stderr to system log when embedded in an Android app.

- gh-109118 [https://github.com/python/cpython/issues/109118]: annotation scope within class scopes can now contain lambdas.

- gh-117894 [https://github.com/python/cpython/issues/117894]: Prevent `agen.aclose()` objects being re-used after `.throw()`.
- gh-117881 [https://github.com/python/cpython/issues/117881]: prevent concurrent access to an async generator via athrow().throw() or asend().throw()
- gh-117536 [https://github.com/python/cpython/issues/117536]: Fix a **`RuntimeWarning`** when calling `agen.aclose().throw(Exception)`.
- gh-117755 [https://github.com/python/cpython/issues/117755]: Fix mimalloc allocator for huge memory allocation (around 8,589,934,592 GiB) on s390x. Patch by Victor Stinner.
- gh-117750 [https://github.com/python/cpython/issues/117750]: Fix issue where an object's dict would get out of sync with the object's internal values when being cleared. `obj.__dict__.clear()` now clears the internal values, but leaves the dict attached to the object.
- gh-117431 [https://github.com/python/cpython/issues/117431]: Improve the performance of the following **`bytes`** and **`bytearray`** methods by adapting them to the **`METH_FASTCALL`** calling convention:
  - **`count()`**
  - **`find()`**
  - **`index()`**
  - **`rfind()`**
  - **`rindex()`**
- gh-117709 [https://github.com/python/cpython/issues/117709]: Speed up calls to **`str()`** with positional-only argument, by

using the **PEP 590** [https://peps.python.org/pep-0590/] `vectorcall` calling convention. Patch by Erlend Aasland.

- gh-117680 [https://github.com/python/cpython/issues/117680]: Give `_PyInstructionSequence` a Python interface and use it in tests.

- gh-115776 [https://github.com/python/cpython/issues/115776]: Statically allocated objects are, by definition, immortal so must be marked as such regardless of whether they are in extension modules or not.

- gh-117385 [https://github.com/python/cpython/issues/117385]: Remove unhandled `PY_MONITORING_EVENT_BRANCH` and `PY_MONITORING_EVENT_EXCEPTION_HANDLED` events from **`sys.settrace()`**.

- gh-116322 [https://github.com/python/cpython/issues/116322]: Extension modules may indicate to the runtime that they can run without the GIL. Multi-phase init modules do so by calling providing `Py_MOD_GIL_NOT_USED` for the `Py_mod_gil` slot, while single-phase init modules call `PyUnstable_Module_SetGIL(mod, Py_MOD_GIL_NOT_USED)` from their init function.

- gh-116129 [https://github.com/python/cpython/issues/116129]: Implement **PEP 696** [https://peps.python.org/pep-0696/], adding support for defaults on type parameters. Patch by Jelle Zijlstra.

- gh-93502 [https://github.com/python/cpython/issues/93502]: Add two new functions to the C-API, **`PyRefTracer_SetTracer()`** and **`PyRefTracer_GetTracer()`**, that allows to track object creation and destruction the same way the **`tracemalloc`** module does. Patch by Pablo Galindo

- gh-107674 [https://github.com/python/cpython/issues/107674]: Improved the performance of `sys.settrace()` significantly

- gh-95754 [https://github.com/python/cpython/issues/95754]: Improve the error message when a script shadowing a module from the standard library causes `AttributeError` to be raised. Similarly, improve the error message when a script shadowing a third party module attempts to access an attribute from that third party module while still initialising.

- gh-99180 [https://github.com/python/cpython/issues/99180]: Elide uninformative traceback indicators in `return` and simple `assignment` statements. Patch by Pablo Galindo.

- gh-105879 [https://github.com/python/cpython/issues/105879]: Allow the *globals* and *locals* arguments to `exec()` and `eval()` to be passed as keywords.

Library

- gh-118418 [https://github.com/python/cpython/issues/118418]: A `DeprecationWarning` is now emitted if you fail to pass a value to the new *type_params* parameter of `typing._eval_type()` or `typing.ForwardRef._evaluate()`. (Using either of these private and undocumented functions is discouraged to begin with, but failing to pass a value to the `type_params` parameter may lead to incorrect behaviour on Python 3.12 or newer.)

- gh-118660 [https://github.com/python/cpython/issues/118660]: Add an optional second type parameter to `typing.ContextManager` and `typing.AsyncContextManager`, representing the return types of `__exit__()` and `__aexit__()` respectively. This parameter defaults to `bool | None`.

- gh-118650 [https://github.com/python/cpython/issues/118650]: The `enum` module allows method named `_repr_*` to be defined on `Enum` types.

- gh-118648 [https://github.com/python/cpython/issues/118648]: Add type parameter defaults to **`typing.Generator`** and **`typing.AsyncGenerator`**.

- gh-101137 [https://github.com/python/cpython/issues/101137]: Mime type `text/x-rst` is now supported by **`mimetypes`**.

- gh-118164 [https://github.com/python/cpython/issues/118164]: The Python implementation of the `decimal` module could appear to hang in relatively small power cases (like `2**117`) if context precision was set to a very high value. A different method to check for exactly representable results is used now that doesn’t rely on computing `10**precision` (which could be effectively too large to compute).

- gh-111744 [https://github.com/python/cpython/issues/111744]: `breakpoint()` and `pdb.set_trace()` now enter the debugger immediately after the call rather than before the next line is executed.

- gh-118500 [https://github.com/python/cpython/issues/118500]: Add **`pdb`** support for zipapps

- gh-118406 [https://github.com/python/cpython/issues/118406]: Add signature for **`sqlite3.Connection`** objects.

- gh-101732 [https://github.com/python/cpython/issues/101732]: Use a Y2038 compatible openssl time function when available.

- gh-118404 [https://github.com/python/cpython/issues/118404]: Fix **`inspect.signature()`** for non-comparable callables.

- gh-118402 [https://github.com/python/cpython/issues/118402]: Fix `inspect.signature()` for the result of the `functools.cmp_to_key()` call.

- gh-116622 [https://github.com/python/cpython/issues/116622]: On Android, `sysconfig.get_platform` now returns the format specified by **PEP 738** [https://peps.python.org/pep-0738/].

- gh-118285 [https://github.com/python/cpython/issues/118285]: Allow to specify the signature of custom callable instances of extension type by the `__text_signature__` attribute. Specify signatures of `operator.attrgetter`, `operator.itemgetter`, and `operator.methodcaller` instances.

- gh-118314 [https://github.com/python/cpython/issues/118314]: Fix an edge case in `binascii.a2b_base64()` strict mode, where excessive padding is not detected when no padding is necessary.

- gh-118271 [https://github.com/python/cpython/issues/118271]: Add the **`PhotoImage`** methods **`read()`** to read an image from a file and **`data()`** to get the image data. Add *background* and *grayscale* parameters to **`PhotoImage`** method **`write()`**.

- gh-118225 [https://github.com/python/cpython/issues/118225]: Add the **`PhotoImage`** method **`copy_replace()`** to copy a region from one image to other image, possibly with pixel zooming and/or subsampling. Add *from_coords* parameter to **`PhotoImage`** methods **`copy()`**, **`zoom()`** and **`subsample()`**. Add *zoom* and *subsample* parameters to **`PhotoImage`** method **`copy()`**.

- gh-118221 [https://github.com/python/cpython/issues/118221]: Fix a bug where `sqlite3.Connection.iterdump()` could fail if a custom `row_factory` was used. Patch by Erlend Aasland.

- gh-118013 [https://github.com/python/cpython/issues/118013]: Fix regression introduced in gh-103193 [https://github.com/python/cpython/issues/103193] that meant that calling **`inspect.getattr_static()`** on an instance would cause a strong reference to that instance's class to persist in an internal cache in the **`inspect`** module. This caused unexpected memory consumption if the class was dynamically created, the class held strong references to other objects which took up a significant amount of memory, and the cache contained the sole strong reference to the class. The fix for the regression leads to a slowdown in **`getattr_static()`**, but the function should still be significantly faster than it was in Python 3.11. Patch by Alex Waygood.

- gh-118218 [https://github.com/python/cpython/issues/118218]: Speed up **`itertools.pairwise()`** in the common case by up to 1.8x.

- gh-117486 [https://github.com/python/cpython/issues/117486]: Improve the behavior of user-defined subclasses of **`ast.AST`**. Such classes will now require no changes in the usual case to conform with the behavior changes of the **`ast`** module in Python 3.13. Patch by Jelle Zijlstra.

- gh-90848 [https://github.com/python/cpython/issues/90848]: Fixed **`unittest.mock.create_autospec()`** to configure parent mock with keyword arguments.

- gh-118168 [https://github.com/python/cpython/issues/118168]: Fix incorrect argument substitution when **`typing.Unpack`** is used with the builtin **`tuple`**. **`typing.Unpack`** now raises **`TypeError`** when used with certain invalid types. Patch by Jelle Zijlstra.

- gh-118131 [https://github.com/python/cpython/issues/118131]: Add command-line interface for the `random` module. Patch by Hugo van Kemenade.

- gh-118107 [https://github.com/python/cpython/issues/118107]: Fix `zipimport` reading of ZIP64 files with file entries that are too big or offset too far.

- gh-102511 [https://github.com/python/cpython/issues/102511]: Fix `os.path.normpath()` for UNC paths on Windows. Speed up `os.path.splitroot()` with a native implementation.

- gh-117535 [https://github.com/python/cpython/issues/117535]: Change the unknown filename of `warnings` from `sys` to `<sys>` to clarify that it's not a real filename.

- gh-114053 [https://github.com/python/cpython/issues/114053]: Fix erroneous `NameError` when calling `typing.get_type_hints()` on a class that made use of **PEP 695** [https://peps.python.org/pep-0695/] type parameters in a module that had `from __future__ import annotations` at the top of the file. Patch by Alex Waygood.

- gh-116931 [https://github.com/python/cpython/issues/116931]: Add parameter *fileobj* check for `tarfile.TarFile.addfile()`

- gh-117995 [https://github.com/python/cpython/issues/117995]: Don't raise `DeprecationWarning` when a sequence of parameters is used to bind indexed, nameless placeholders. See also gh-100668 [https://github.com/python/cpython/issues/100668].

- gh-80361 [https://github.com/python/cpython/issues/80361]: Fix TypeError in `email.message.Message.get_payload()` when the charset is **RFC 2231** [https://datatracker.ietf.org/doc/html/rfc2231.html] encoded.

- gh-86650 [https://github.com/python/cpython/issues/86650]: Fix IndexError when parse some emails with invalid Message-ID (including one-off addresses generated by Microsoft Outlook).

- gh-117691 [https://github.com/python/cpython/issues/117691]: Improve the error messages emitted by **`tarfile`** deprecation warnings relating to PEP 706. If a `filter` argument is not provided to `extract()` or `extractall`, the deprecation warning now points to the line in the user's code where the relevant function was called. Patch by Alex Waygood.

- gh-115874 [https://github.com/python/cpython/issues/115874]: Fixed a possible segfault during garbage collection of `_asyncio.FutureIter` objects. Patch by Savannah Ostrowski.

- gh-115060 [https://github.com/python/cpython/issues/115060]: Speed up **`pathlib.Path.glob()`** by omitting an initial **`is_dir()`** call. As a result of this change, **`glob()`** can no longer raise **`OSError`**.

- gh-77102 [https://github.com/python/cpython/issues/77102]: **`site`** module now parses `.pth` file with UTF-8 first, and locale encoding if `UnicodeDecodeError` happened. It supported only locale encoding before.

- gh-76785 [https://github.com/python/cpython/issues/76785]: We've exposed the low-level **`_interpreters`** module for the sake of the PyPI implementation of **PEP 734** [https://peps.python.org/pep-0734/]. It was sometimes available as the **`_xxsubinterpreters`** module and was formerly used only for testing. For the most part, it should be considered an internal module, like **`_thread`** and **`_imp`**. See https://discuss.python.org/t/pep-734-multiple-interpreters-in-the-stdlib/41147/26.

- gh-115060 [https://github.com/python/cpython/issues/115060]: Speed up `pathlib.Path.glob()` by not scanning directories for non-wildcard pattern segments.

- gh-117727 [https://github.com/python/cpython/issues/117727]: Speed up `pathlib.Path.iterdir()` by using `os.scandir()` internally.

- gh-117586 [https://github.com/python/cpython/issues/117586]: Speed up `pathlib.Path.walk()` by working with strings internally.

- gh-117722 [https://github.com/python/cpython/issues/117722]: Change the new multi-separator support in `asyncio.StreamReader.readuntil()` to only accept tuples of separators rather than arbitrary iterables.

- gh-117692 [https://github.com/python/cpython/issues/117692]: Fixes a bug when `doctest.DocTestFinder` was failing on wrapped `builtin_function_or_method`.

- gh-117348 [https://github.com/python/cpython/issues/117348]: Largely restored import time performance of configparser by avoiding dataclasses.

- gh-117641 [https://github.com/python/cpython/issues/117641]: Speedup `os.path.commonpath()` on Unix.

- gh-117663 [https://github.com/python/cpython/issues/117663]: Fix `_simple_enum` to detect aliases when multiple arguments are present but only one is the member value.

- gh-117636 [https://github.com/python/cpython/issues/117636]: Speedup `os.path.join()`.

- gh-117618 [https://github.com/python/cpython/issues/117618]: Support `package.module` as `filename` for `break` command of **`pdb`**

- gh-102247 [https://github.com/python/cpython/issues/102247]: the status codes enum with constants in http.HTTPStatus are updated to include the names from RFC9110. This RFC includes some HTTP statuses previously only used for WEBDAV and assigns more generic names to them.

  The old constants are preserved for backwards compatibility.

- gh-117607 [https://github.com/python/cpython/issues/117607]: Speedup **`os.path.relpath()`**.

- gh-117586 [https://github.com/python/cpython/issues/117586]: Speed up **`pathlib.Path.glob()`** by working with strings internally.

- gh-117225 [https://github.com/python/cpython/issues/117225]: Add colour to doctest output. Patch by Hugo van Kemenade.

- gh-117566 [https://github.com/python/cpython/issues/117566]: **`ipaddress.IPv6Address.is_loopback()`** will now return `True` for IPv4-mapped loopback addresses, i.e. addresses in the `::ffff:127.0.0.0/104` address space.

- gh-117546 [https://github.com/python/cpython/issues/117546]: Fix issue where **`os.path.realpath()`** stopped resolving symlinks after encountering a symlink loop on POSIX.

- gh-116720 [https://github.com/python/cpython/issues/116720]: Improved behavior of **`asyncio.TaskGroup`** when an external cancellation collides with an internal cancellation. For example, when two task groups are nested and both experience an exception in a child task simultaneously, it was possible that the outer task group would misbehave, because its internal cancellation was swallowed by the inner task group.

In the case where a task group is cancelled externally and also must raise an `ExceptionGroup`, it will now call the parent task's `cancel()` method. This ensures that a `asyncio.CancelledError` will be raised at the next `await`, so the cancellation is not lost.

An added benefit of these changes is that task groups now preserve the cancellation count (`asyncio.Task.cancelling()`).

In order to handle some corner cases, `asyncio.Task.uncancel()` may now reset the undocumented `_must_cancel` flag when the cancellation count reaches zero.

- gh-117516 [https://github.com/python/cpython/issues/117516]: Add `typing.TypeIs`, implementing **PEP 742** [https://peps.python.org/pep-0742/]. Patch by Jelle Zijlstra.
- gh-117503 [https://github.com/python/cpython/issues/117503]: Fix support of non-ASCII user names in bytes paths in `os.path.expanduser()` on Posix.
- gh-117394 [https://github.com/python/cpython/issues/117394]: `os.path.ismount()` is now 2-3 times faster if the user has permissions.
- gh-117313 [https://github.com/python/cpython/issues/117313]: Only treat `'\n'`, `'\r'` and `'\r\n'` as line separators in re-folding the `email` messages. Preserve control characters `'\v'`, `'\f'`, `'\x1c'`, `'\x1d'` and `'\x1e'` and Unicode line separators `'\x85'`, `'\u2028'` and `'\u2029'` as is.

- gh-117142 [https://github.com/python/cpython/issues/117142]: Convert **`_ctypes`** to multi-phase initialisation (**PEP 489** [https://peps.python.org/pep-0489/]).

- gh-66543 [https://github.com/python/cpython/issues/66543]: Add the **`mimetypes.guess_file_type()`** function which works with file path. Passing file path instead of URL in **`guess_type()`** is soft deprecated.

- gh-68583 [https://github.com/python/cpython/issues/68583]: webbrowser CLI: replace getopt with argparse, add long options. Patch by Hugo van Kemenade.

- gh-116871 [https://github.com/python/cpython/issues/116871]: Name suggestions for **`AttributeError`** and **`ImportError`** now only include underscored names if the original name was underscored.

- gh-116023 [https://github.com/python/cpython/issues/116023]: Don’t show empty fields (value `None` or `[]`) in **`ast.dump()`** by default. Add `show_empty=False` parameter to optionally show them.

- gh-115961 [https://github.com/python/cpython/issues/115961]: Added **`name`** and **`mode`** attributes for compressed and archived file-like objects in modules **`bz2`**, **`lzma`**, **`tarfile`** and **`zipfile`**. The value of the **`mode`** attribute of **`gzip.GzipFile`** was changed from integer (`1` or `2`) to string (`'rb'` or `'wb'`). The value of the **`mode`** attribute of the readable file-like object returned by **`zipfile.ZipFile.open()`** was changed from `'r'` to `'rb'`.

- gh-82062 [https://github.com/python/cpython/issues/82062]: Fix **`inspect.signature()`** to correctly handle parameter defaults on methods in extension modules that use names defined in the module namespace.

- gh-83856 [https://github.com/python/cpython/issues/83856]: Honor **`atexit`** for all **`multiprocessing`** start methods

- gh-113081 [https://github.com/python/cpython/issues/113081]: Print colorized exception just like built-in traceback in **`pdb`**

- gh-112855 [https://github.com/python/cpython/issues/112855]: Speed up pickling of **`pathlib.PurePath`** objects. Patch by Barney Gale.

- gh-111744 [https://github.com/python/cpython/issues/111744]: Support opcode events in **`bdb`**

- gh-109617 [https://github.com/python/cpython/issues/109617]: **`ncurses`**: fixed a crash that could occur on macOS 13 or earlier when Python was built with Apple Xcode 15’s SDK.

- gh-83151 [https://github.com/python/cpython/issues/83151]: Enabled arbitrary statements and evaluations in **`pdb`** shell to access the local variables of the current frame, which made it possible for multi-scope code like generators or nested function to work.

- gh-110209 [https://github.com/python/cpython/issues/110209]: Add **`__class_getitem__()`** to **`types.GeneratorType`** and **`types.CoroutineType`** for type hinting purposes. Patch by James Hilton-Balfe.

- gh-108191 [https://github.com/python/cpython/issues/108191]: The **`types.SimpleNamespace`** now accepts an optional positional argument which specifies initial values of attributes as a dict or an iterable of key-value pairs.

- gh-62090 [https://github.com/python/cpython/issues/62090]: Fix assertion errors caused by whitespace in metavars or

`SUPPRESS`-ed groups in **`argparse`** by simplifying usage formatting. Patch by Ali Hamdan.

- gh-102402 [https://github.com/python/cpython/issues/102402]: Adjust `logging.LogRecord` to use `time.time_ns()` and fix minor bug related to floating-point math.

- gh-100242 [https://github.com/python/cpython/issues/100242]: Bring pure Python implementation `functools.partial.__new__` more in line with the C-implementation by not just always checking for the presence of the attribute `'func'` on the first argument of `partial`. Instead, both the Python version and the C version perform an `isinstance(func, partial)` check on the first argument of `partial`.

- gh-99730 [https://github.com/python/cpython/issues/99730]: HEAD requests are no longer upgraded to GET request during redirects in urllib.

- gh-66410 [https://github.com/python/cpython/issues/66410]: Setting the **`tkinter`** module global **`wantobjects`** to `2` before creating the **`Tk`** object or call the **`wantobjects()`** method of the **`Tk`** object with argument `2` makes now arguments to callbacks registered in the **`tkinter`** module to be passed as various Python objects (`int`, `float`, `bytes`, `tuple`), depending on their internal representation in Tcl, instead of always `str`. **`tkinter.wantobjects`** is now set to `2` by default.

- bpo-40943 [https://bugs.python.org/issue?@action=redirect&bpo=40943]: Fix several IndexError when parse emails with truncated Message-ID, address, routes, etc, e.g. `example@`.

- bpo-39324 [https://bugs.python.org/issue?@action=redirect&bpo=39324]: Add mime type mapping for

.md <-> text/markdown

- bpo-18108 [https://bugs.python.org/issue?@action=redirect&bpo=18108]: **`shutil.chown()`** now supports *dir_fd* and *follow_symlinks* keyword arguments.

- bpo-30988 [https://bugs.python.org/issue?@action=redirect&bpo=30988]: Fix parsing of emails with invalid address headers having a leading or trailing dot. Patch by tsufeki.

- bpo-32839 [https://bugs.python.org/issue?@action=redirect&bpo=32839]: Add the **`after_info()`** method for Tkinter widgets.

Documentation

- gh-117928 [https://github.com/python/cpython/issues/117928]: The minimum Sphinx version required for the documentation is now 6.2.1.

Build

- gh-118734 [https://github.com/python/cpython/issues/118734]: Fixes Windows build when invoked directly (not through the `build.bat` script) without specifying a value for `UseTIER2`.
- gh-115119 [https://github.com/python/cpython/issues/115119]: The `configure` option **`--with-system-libmpdec`** now defaults to `yes`. The bundled copy of `libmpdecimal` will be removed in Python 3.15.
- gh-117845 [https://github.com/python/cpython/issues/117845]: Fix building against recent libedit versions by detecting readline hook signatures in **configure**.
- gh-116622 [https://github.com/python/cpython/issues/116622]: A testbed project was added to run the test suite on Android.

- gh-117645 [https://github.com/python/cpython/issues/117645]: Increase WASI stack size from 512 KiB to 8 MiB and the initial memory from 10 MiB to 20 MiB. Patch by Victor Stinner.
- gh-115119 [https://github.com/python/cpython/issues/115119]: **configure** now uses **pkg-config** to detect `decimal` dependencies if the `--with-system-libmpdec` option is given.

Windows

- gh-115119 [https://github.com/python/cpython/issues/115119]: Update Windows installer to use libmpdecimal 4.0.0.
- gh-118486 [https://github.com/python/cpython/issues/118486]: `os.mkdir()` now accepts *mode* of `0o700` to restrict the new directory to the current user.
- gh-118347 [https://github.com/python/cpython/issues/118347]: Fixes launcher updates not being installed.
- gh-118293 [https://github.com/python/cpython/issues/118293]: The `multiprocessing` module now passes the `STARTF_FORCEOFFFEEDBACK` flag when spawning processes to tell Windows not to change the mouse cursor.
- gh-115009 [https://github.com/python/cpython/issues/115009]: Update Windows installer to use SQLite 3.45.3.
- gh-90329 [https://github.com/python/cpython/issues/90329]: Suppress the warning displayed on virtual environment creation when the requested and created paths differ only by a short (8.3 style) name. Warnings will continue to be shown if a junction or symlink in the path caused the venv to be created in a different location than originally requested.
- gh-117786 [https://github.com/python/cpython/issues/117786]: Fixes virtual environments not correctly launching when created from a Store install.

macOS

- gh-115119 [https://github.com/python/cpython/issues/115119]: Update macOS installer to use libmpdecimal 4.0.0.
- gh-114099 [https://github.com/python/cpython/issues/114099]: iOS preprocessor symbol usage was made compatible with older macOS SDKs.
- gh-115009 [https://github.com/python/cpython/issues/115009]: Update macOS installer to use SQLite 3.45.3.
- gh-91629 [https://github.com/python/cpython/issues/91629]: Use `~/.config/fish/conf.d` configs and **fish_add_path** to set **`PATH`** when installing for the Fish shell.

IDLE

- bpo-34774 [https://bugs.python.org/issue?@action=redirect&bpo=34774]: Use user-selected color theme for Help => IDLE Doc.

C API

- gh-118124 [https://github.com/python/cpython/issues/118124]: Fix `Py_BUILD_ASSERT` and `Py_BUILD_ASSERT_EXPR` for non-constant expressions: use `static_assert()` on C11 and newer. Patch by Victor Stinner.

- gh-110850 [https://github.com/python/cpython/issues/110850]: Add “Raw” variant of PyTime functions

  - `PyTime_MonotonicRaw()`
  - `PyTime_PerfCounterRaw()`
  - `PyTime_TimeRaw()`

  Patch by Victor Stinner.

- gh-117987 [https://github.com/python/cpython/issues/117987]: Restore functions removed in Python 3.13 alpha 1:

  - `Py_SetPythonHome()`
  - `Py_SetProgramName()`
  - `PySys_SetArgvEx()`
  - `PySys_SetArgv()`

  Patch by Victor Stinner.

- gh-117929 [https://github.com/python/cpython/issues/117929]: Restore removed `PyEval_InitThreads()` function. Patch by Victor Stinner.

- gh-117534 [https://github.com/python/cpython/issues/117534]: Improve validation logic in the C implementation of `datetime.datetime.fromisoformat()` to better handle invalid years. Patch by Vlad Efanov.

- gh-68114 [https://github.com/python/cpython/issues/68114]: Fixed skipitem()’s handling of the old ‘w’ and ‘w#’ formatters. These are no longer supported and now raise an exception if used.

- gh-111997 [https://github.com/python/cpython/issues/111997]: Add a C-API for firing monitoring events.

Python 3.13.0 alpha 6

Release date: 2024-04-09

Core and Builtins

- gh-117494 [https://github.com/python/cpython/issues/117494]: Refactored the instruction sequence data structure out of compile.c into instruction_sequence.c.

- gh-116968 [https://github.com/python/cpython/issues/116968]: Introduce a unified 16-bit backoff counter type (`_Py_BackoffCounter`), shared between the Tier 1 adaptive specializer and the Tier 2 optimizer. The API used for adaptive specialization counters is changed but the behavior is (supposed to be) identical.

  The behavior of the Tier 2 counters is changed:

  - There are no longer dynamic thresholds (we never varied these).
  - All counters now use the same exponential backoff.
  - The counter for `JUMP_BACKWARD` starts counting down from 16.
  - The `temperature` in side exits starts counting down from 64.

- gh-117431 [https://github.com/python/cpython/issues/117431]: Improve the performance of the following **`bytes`** and **`bytearray`** methods by adapting them to the **`METH_FASTCALL`** calling convention:

  - **`endswith()`**
  - **`startswith()`**

- gh-117431 [https://github.com/python/cpython/issues/117431]: Improve the performance of the following **`str`** methods by adapting them to the **`METH_FASTCALL`** calling convention:

  - **`count()`**
  - **`endswith()`**
  - **`find()`**
  - **`index()`**
  - **`rfind()`**
  - **`rindex()`**
  - **`startswith()`**

- gh-117411 [https://github.com/python/cpython/issues/117411]: Move `PyFutureFeatures` to an internal header and make it private.

- gh-109120 [https://github.com/python/cpython/issues/109120]: Added handle of incorrect star expressions, e.g `f(3, *)`. Patch by Grigoryev Semyon

- gh-117266 [https://github.com/python/cpython/issues/117266]: Fix crashes for certain user-created subclasses of **`ast.AST`**. Such classes are now expected to set the `_field_types` attribute.

- gh-99108 [https://github.com/python/cpython/issues/99108]: Updated the **`hashlib`** built-in HACL* project [https://github.com/hacl-star/hacl-star] C code from upstream that we use for many implementations when they are not present via OpenSSL in a given build. This also avoids the rare potential for a C symbol name one definition rule linking issue.

- gh-117108 [https://github.com/python/cpython/issues/117108]: Change the old space bit of objects in the young generation from 0 to gcstate->visited, so that any objects created during GC will have the old bit set correctly if they get moved into the old generation.

- gh-117108 [https://github.com/python/cpython/issues/117108]: The cycle GC now chooses the size of increments based on the total heap size, instead of the rate of object creation. This ensures that it can keep up with growing heaps.

- gh-116735 [https://github.com/python/cpython/issues/116735]: For `INSTRUMENTED_CALL_FUNCTION_EX`, set `arg0` to `sys.monitoring.MISSING` instead of `None` for **`CALL`** event.

- gh-113964 [https://github.com/python/cpython/issues/113964]: Starting new threads and process creation through `os.fork()` are now only prevented once all non-daemon threads exit.

- gh-116626 [https://github.com/python/cpython/issues/116626]: Ensure `INSTRUMENTED_CALL_FUNCTION_EX` always emits `CALL`

- gh-116554 [https://github.com/python/cpython/issues/116554]: `list.sort()` now exploits more cases of partial ordering, particularly those with long descending runs with sub-runs of equal values. Those are recognized as single runs now (previously, each block of repeated values caused a new run to be created).

- gh-114099 [https://github.com/python/cpython/issues/114099]: Added a Loader that can discover extension modules in an iOS-style Frameworks folder.

- gh-115775 [https://github.com/python/cpython/issues/115775]: Compiler populates the new `__static_attributes__` field on a class with the names of attributes of this class which are accessed through self.X from any function in its body.

- gh-115776 [https://github.com/python/cpython/issues/115776]: The array of values, the `PyDictValues` struct is now embedded in the object during allocation. This provides better performance in the common case, and does not degrade as much when the object’s `__dict__` is materialized.

- gh-108362 [https://github.com/python/cpython/issues/108362]: Implement an incremental cyclic garbage collector. By collecting the old generation in increments, there is no need for a full heap scan. This can hugely reduce maximum pause time for programs with large heaps.

Reduce the number of generations from three to two. The old generation is split into two spaces, “visited” and “pending”.

Collection happens in two steps:: * An increment is formed from the young generation and a small part of the pending space. * This increment is scanned and the survivors moved to the end of the visited space.

When the collecting space becomes empty, the two spaces are swapped.

- gh-109870 [https://github.com/python/cpython/issues/109870]: Dataclasses now calls `exec()` once per dataclass, instead of once per method being added. This can speed up dataclass creation by up to 20%.
- gh-97901 [https://github.com/python/cpython/issues/97901]: Mime type `text/rtf` is now supported by `mimetypes`.
- bpo-24612 [https://bugs.python.org/issue?@action=redirect&bpo=24612]: Improve the `SyntaxError` that happens when ‘not’ appears after an operator. Patch by Pablo Galindo

Library

- gh-117648 [https://github.com/python/cpython/issues/117648]: Improve performance of `os.path.join()` and `os.path.expanduser()`.
- gh-117584 [https://github.com/python/cpython/issues/117584]: Raise `TypeError` for non-paths in `posixpath.relpath`.
- gh-117467 [https://github.com/python/cpython/issues/117467]: Preserve mailbox ownership when rewriting in `mailbox.mbox.flush`. Patch by Tony Mountifield.

- gh-114848 [https://github.com/python/cpython/issues/114848]: Raise **`FileNotFoundError`** when `getcwd()` returns '(unreachable)', which can happen on Linux >= 2.6.36 with glibc < 2.27.

- gh-117459 [https://github.com/python/cpython/issues/117459]: **`asyncio.asyncio.run_coroutine_threadsafe()`** now keeps the traceback of **`CancelledError`**, **`TimeoutError`** and **`InvalidStateError`** which are raised in the coroutine.

- gh-117381 [https://github.com/python/cpython/issues/117381]: Fix error message for **`ntpath.commonpath()`**.

- gh-117337 [https://github.com/python/cpython/issues/117337]: Deprecate undocumented **`glob.glob0()`** and **`glob.glob1()`** functions. Use **`glob.glob()`** and pass a directory to its *root_dir* argument instead.

- gh-117349 [https://github.com/python/cpython/issues/117349]: Optimise several functions in **`os.path`**.

- gh-117348 [https://github.com/python/cpython/issues/117348]: Refactored **`configparser.RawConfigParser._read()`** to reduce cyclometric complexity and improve comprehensibility.

- gh-117335 [https://github.com/python/cpython/issues/117335]: Raise TypeError for non-sequences for **`ntpath.commonpath()`**.

- gh-66449 [https://github.com/python/cpython/issues/66449]: **`configparser.ConfigParser`** now accepts unnamed sections before named ones, if configured to do so.

- gh-88014 [https://github.com/python/cpython/issues/88014]: In documentation of **`gzip.GzipFile`** in module gzip, explain data type of optional constructor argument *mtime*, and recommend `mtime = 0` for generating deterministic streams.

- gh-117310 [https://github.com/python/cpython/issues/117310]: Fixed an unlikely early & extra `Py_DECREF` triggered crash in **`ssl`** when creating a new `_ssl._SSLContext` if CPython was built implausibly such that the default cipher list is empty **or** the SSL library it was linked against reports a failure from its C `SSL_CTX_set_cipher_list()` API.

- gh-117294 [https://github.com/python/cpython/issues/117294]: A `DocTestCase` now reports as skipped if all examples in the doctest are skipped.

- gh-98966 [https://github.com/python/cpython/issues/98966]: In **`subprocess`**, raise a more informative message when `stdout=STDOUT`.

- gh-117225 [https://github.com/python/cpython/issues/117225]: doctest: only print “and X failed” when non-zero, don’t pluralise “1 items”. Patch by Hugo van Kemenade.

- gh-117205 [https://github.com/python/cpython/issues/117205]: Speed up **`compileall.compile_dir()`** by 20% when using multiprocessing by increasing `chunksize`.

- gh-117178 [https://github.com/python/cpython/issues/117178]: Fix regression in lazy loading of self-referential modules, introduced in gh-114781 [https://github.com/python/cpython/issues/114781].

- gh-112383 [https://github.com/python/cpython/issues/112383]: Fix **`dis`** module’s handling of `ENTER_EXECUTOR` instructions.

- gh-117182 [https://github.com/python/cpython/issues/117182]: Lazy-loading of modules that modify their own `__class__` no longer reverts the `__class__` to **`types.ModuleType`**.

- gh-117084 [https://github.com/python/cpython/issues/117084]: Fix **`zipfile`** extraction for directory entries with the name containing backslashes on Windows.

- gh-117114 [https://github.com/python/cpython/issues/117114]: Make **`os.path.isdevdrive()`** available on all platforms. For those that do not offer Dev Drives, it will always return `False`.

- gh-117110 [https://github.com/python/cpython/issues/117110]: Fix a bug that prevents subclasses of **`typing.Any`** to be instantiated with arguments. Patch by Chris Fu.

- gh-109653 [https://github.com/python/cpython/issues/109653]: Deferred select imports in importlib.metadata and importlib.resources for a 14% speedup.

- gh-70647 [https://github.com/python/cpython/issues/70647]: Start the deprecation period for the current behavior of **`datetime.datetime.strptime()`** and **`time.strptime()`** which always fails to parse a date string with a **`ValueError`** involving a day of month such as `strptime("02-29", "%m-%d")` when a year is **not** specified and the date happen to be February 29th. This should help avoid users finding new bugs every four years due to a natural mistaken assumption about the API when parsing partial date values.

- gh-116987 [https://github.com/python/cpython/issues/116987]: Fixed **`inspect.findsource()`** for class code objects.

- gh-114099 [https://github.com/python/cpython/issues/114099]: Modify standard library to allow for iOS platform differences.

- gh-90872 [https://github.com/python/cpython/issues/90872]: On Windows, **`subprocess.Popen.wait()`** no longer calls `WaitForSingleObject()` with a negative timeout: pass `0` ms if the timeout is negative. Patch by Victor Stinner.

- gh-116957 [https://github.com/python/cpython/issues/116957]: configparser: Don’t leave ConfigParser values in an invalid state (stored as a list instead of a str) after an earlier read raised DuplicateSectionError or DuplicateOptionError.

- gh-115538 [https://github.com/python/cpython/issues/115538]: **`_io.WindowsConsoleIO`** now emit a warning if a boolean value is passed as a filedescriptor argument.

- gh-90095 [https://github.com/python/cpython/issues/90095]: Ignore empty lines and comments in `.pdbrc`

- gh-106531 [https://github.com/python/cpython/issues/106531]: Refreshed zipfile._path from zipp 3.18 [https://zipp.readthedocs.io/en/latest/history.html#v3-18-0], providing better compatibility for PyPy, better glob performance for deeply nested zipfiles, and providing internal access to `CompleteDirs.inject` for use in other tests (like importlib.resources).

- gh-63207 [https://github.com/python/cpython/issues/63207]: On Windows, **`time.time()`** now uses the `GetSystemTimePreciseAsFileTime()` clock to have a resolution better than 1 us, instead of the `GetSystemTimeAsFileTime()` clock which has a resolution of 15.6 ms. Patch by Victor Stinner.

- gh-116764 [https://github.com/python/cpython/issues/116764]: Restore support of `None` and other false values in **`urllib.parse`** functions **`parse_qs()`** and **`parse_qsl()`**. Also, they now raise a TypeError for non-zero integers and non-empty sequences.

- gh-116811 [https://github.com/python/cpython/issues/116811]: In `PathFinder.invalidate_caches`, delegate to `MetadataPathFinder.invalidate_caches`.

- gh-116647 [https://github.com/python/cpython/issues/116647]: Fix recursive child in dataclasses

- gh-113171 [https://github.com/python/cpython/issues/113171]: Fixed various false positives and false negatives in

  - `ipaddress.IPv4Address.is_private` (see these docs for details)
  - `ipaddress.IPv4Address.is_global`
  - `ipaddress.IPv6Address.is_private`
  - `ipaddress.IPv6Address.is_global`

  Also in the corresponding `ipaddress.IPv4Network` and `ipaddress.IPv6Network` attributes.

- gh-63283 [https://github.com/python/cpython/issues/63283]: In `encodings.idna`, any capitalization of the ACE prefix (`xn--`) is now acceptable. Patch by Pepijn de Vos and Zackery Spytz.

- gh-71042 [https://github.com/python/cpython/issues/71042]: Add `platform.android_ver()`, which provides device and OS information on Android.

- gh-73468 [https://github.com/python/cpython/issues/73468]: Added new `math.fma()` function, wrapping C99's `fma()` operation: fused multiply-add function. Patch by Mark Dickinson and Victor Stinner.

- gh-116608 [https://github.com/python/cpython/issues/116608]: The `importlib.resources` functions `is_resource()`, `open_binary()`, `open_text()`, `path()`, `read_binary()`, and `read_text()` are undeprecated, and support subdirectories via multiple positional arguments. The `contents()` function also allows subdirectories, but remains deprecated.

- gh-116484 [https://github.com/python/cpython/issues/116484]: Change automatically generated `tkinter.Checkbutton` widget names to avoid collisions with automatically generated `tkinter.ttk.Checkbutton` widget names within the same parent widget.

- gh-114314 [https://github.com/python/cpython/issues/114314]: In `ctypes`, ctype data is now stored in type objects directly rather than in a dict subclass. This is an internal change that should not affect usage.

- gh-116401 [https://github.com/python/cpython/issues/116401]: Fix blocking `os.fwalk()` and `shutil.rmtree()` on opening named pipe.

- gh-71052 [https://github.com/python/cpython/issues/71052]: Implement `ctypes.util.find_library()` on Android.

- gh-90535 [https://github.com/python/cpython/issues/90535]: Fix support of *interval* values > 1 in `logging.TimedRotatingFileHandler` for `when='MIDNIGHT'` and `when='Wx'`.

- gh-113308 [https://github.com/python/cpython/issues/113308]: Remove some internal protected parts from `uuid`: `_has_uuid_generate_time_safe`, `_netbios_getnode`, `_ipconfig_getnode`, and `_load_system_functions`. They were unused.

- gh-115627 [https://github.com/python/cpython/issues/115627]: Fix the `ssl` module error handling of connection terminate by peer. It now throws an OSError with the appropriate error code instead of an EOFError.

- gh-114847 [https://github.com/python/cpython/issues/114847]: Speed up `os.path.realpath()` on non-Windows platforms.

- gh-114271 [https://github.com/python/cpython/issues/114271]: Fix a race in `threading.Thread.join()`.

  `threading._MainThread` now always represents the main thread of the main interpreter.

  `PyThreadState.on_delete` and `PyThreadState.on_delete_data` have been removed.

- gh-113538 [https://github.com/python/cpython/issues/113538]: Add **`asyncio.Server.close_clients()`** and **`asyncio.Server.abort_clients()`** methods which allow to more forcefully close an asyncio server.

- gh-85287 [https://github.com/python/cpython/issues/85287]: Changes Unicode codecs to return UnicodeEncodeError or UnicodeDecodeError, rather than just UnicodeError.

- gh-113548 [https://github.com/python/cpython/issues/113548]: **`pdb`** now allows CLI arguments to `pdb -m`.

- gh-112948 [https://github.com/python/cpython/issues/112948]: Make completion of **`pdb`** similar to Python REPL

- gh-105866 [https://github.com/python/cpython/issues/105866]: Fixed `_get_slots` bug which caused error when defining dataclasses with slots and a weakref_slot.

- gh-96471 [https://github.com/python/cpython/issues/96471]: Add **`asyncio.Queue`** termination with **`shutdown()`** method.

- gh-89739 [https://github.com/python/cpython/issues/89739]: The **`zipimport`** module can now read ZIP64 files.

- bpo-33533 [https://bugs.python.org/issue?@action=redirect&bpo=33533]: **`asyncio.as_completed()`** now returns an object that is both an asynchronous iterator and plain iterator. The new asynchronous iteration pattern allows for easier correlation between prior tasks and their completed results. This is a closer match to `concurrent.futures.as_completed()`'s iteration pattern. Patch by Justin Arthur.

- bpo-27578 [https://bugs.python.org/issue?@action=redirect&bpo=27578]: `inspect.getsource()` (and related functions) work with empty module files, returning `'\n'` (or reasonable equivalent) instead of raising `OSError`. Patch by Kernc.

- bpo-37141 [https://bugs.python.org/issue?@action=redirect&bpo=37141]: Accept an iterable of separators in `asyncio.StreamReader.readuntil()`, stopping when one of them is encountered.

- gh-66543 [https://github.com/python/cpython/issues/66543]: Make `mimetypes.guess_type()` properly parsing of URLs with only a host name, URLs containing fragment or query, and filenames with only a UNC sharepoint on Windows. Based on patch by Dong-hee Na.

- bpo-15010 [https://bugs.python.org/issue?@action=redirect&bpo=15010]: `unittest.TestLoader.discover()` now saves the original value of `unittest.TestLoader._top_level_dir` and restores it at the end of the call.

Documentation

- gh-115977 [https://github.com/python/cpython/issues/115977]: Remove compatibility references to Emscripten.
- gh-114099 [https://github.com/python/cpython/issues/114099]: Add an iOS platform guide, and flag modules not available on iOS.

- gh-91565 [https://github.com/python/cpython/issues/91565]: Changes to documentation files and config outputs to reflect the new location for reporting bugs - i.e. GitHub rather than bugs.python.org.

Tests

- gh-83434 [https://github.com/python/cpython/issues/83434]: Disable JUnit XML output (`--junit-xml=FILE` command line option) in regrtest when hunting for reference leaks (`-R` option). Patch by Victor Stinner.
- gh-117187 [https://github.com/python/cpython/issues/117187]: Fix XML tests for vanilla Expat <2.6.0.
- gh-116333 [https://github.com/python/cpython/issues/116333]: Tests of TLS related things (error codes, etc) were updated to be more lenient about specific error message strings and behaviors as seen in the BoringSSL and AWS-LC forks of OpenSSL.
- gh-117089 [https://github.com/python/cpython/issues/117089]: Consolidated tests for importlib.metadata in their own `metadata` package.
- gh-115979 [https://github.com/python/cpython/issues/115979]: Update test_importlib so that it passes under WASI SDK 21.
- gh-112536 [https://github.com/python/cpython/issues/112536]: Add –tsan to test.regrtest for running TSAN tests in reasonable execution times. Patch by Donghee Na.
- gh-116307 [https://github.com/python/cpython/issues/116307]: Added import helper `isolated_modules` as `CleanImport` does not remove modules imported during the context. Use it in importlib.resources tests to avoid leaving `mod` around to impede importlib.metadata tests.

Build

- gh-114736 [https://github.com/python/cpython/issues/114736]: Have WASI builds use WASI SDK 21.
- gh-115983 [https://github.com/python/cpython/issues/115983]: Skip building test modules that must be built as shared under WASI.
- gh-71052 [https://github.com/python/cpython/issues/71052]: Add Android build script and instructions.

Windows

- gh-117267 [https://github.com/python/cpython/issues/117267]: Ensure `DirEntry.stat().st_ctime` behaves consistently with **`os.stat()`** during the deprecation period of `st_ctime` by containing the same value as `st_birthtime`. After the deprecation period, `st_ctime` will be the metadata change time (or unavailable through `DirEntry`), and only `st_birthtime` will contain the creation time.
- gh-116195 [https://github.com/python/cpython/issues/116195]: Improves performance of **`os.getppid()`** by using an alternate system API when available. Contributed by vxiiduu.
- gh-88494 [https://github.com/python/cpython/issues/88494]: On Windows, **`time.monotonic()`** now uses the `QueryPerformanceCounter()` clock to have a resolution better than 1 us, instead of the `GetTickCount64()` clock which has a resolution of 15.6 ms. Patch by Victor Stinner.
- gh-116773 [https://github.com/python/cpython/issues/116773]: Fix instances of `<_overlapped.Overlapped object at 0xXXX> still has pending operation at deallocation, the process may crash`.
- gh-91227 [https://github.com/python/cpython/issues/91227]: Fix the asyncio ProactorEventLoop implementation so that sending a datagram to an address that is not listening does not prevent receiving any more datagrams.

- gh-115119 [https://github.com/python/cpython/issues/115119]: Switched from vendored `libmpdecimal` code to a separately-hosted external package in the `cpython-source-deps` repository when building the `_decimal` module.

C API

- gh-117642 [https://github.com/python/cpython/issues/117642]: Fix **PEP 737** [https://peps.python.org/pep-0737/] implementation for `%#T` and `%#N`.
- gh-87193 [https://github.com/python/cpython/issues/87193]: **`_PyBytes_Resize()`** can now be called for bytes objects with reference count > 1, including 1-byte bytes objects. It creates a new bytes object and destroys the old one if it has reference count > 1.
- gh-117021 [https://github.com/python/cpython/issues/117021]: Fix integer overflow in **`PyLong_AsPid()`** on non-Windows 64-bit platforms.
- gh-115756 [https://github.com/python/cpython/issues/115756]: **`PyCode_GetFirstFree()`** is an ustable API now and has been renamed to **`PyUnstable_Code_GetFirstFree()`**. (Contributed by Bogdan Romanyuk in gh-115781 [https://github.com/python/cpython/issues/115781])
- gh-116869 [https://github.com/python/cpython/issues/116869]: Add `test_cext` test: build a C extension to check if the Python C API emits C compiler warnings. Patch by Victor Stinner.
- gh-116869 [https://github.com/python/cpython/issues/116869]: Make the C API compatible with `-Werror=declaration-after-statement` compiler flag again. Patch by Victor Stinner.
- gh-116936 [https://github.com/python/cpython/issues/116936]: Add **`PyType_GetModuleByDef()`** to the limited C API. Patch by Victor Stinner.
- gh-116809 [https://github.com/python/cpython/issues/116809]: Restore removed private `_PyErr_ChainExceptions1()` function. Patch by Victor Stinner.

- gh-115754 [https://github.com/python/cpython/issues/115754]: In the limited C API version 3.13, getting `Py_None`, `Py_False`, `Py_True`, `Py_Ellipsis` and `Py_NotImplemented` singletons is now implemented as function calls at the stable ABI level to hide implementation details. Getting these constants still return borrowed references. Patch by Victor Stinner.
- gh-115754 [https://github.com/python/cpython/issues/115754]: Add **`Py_GetConstant()`** and **`Py_GetConstantBorrowed()`** functions to get constants. For example, `Py_GetConstant(Py_CONSTANT_ZERO)` returns a strong reference to the constant zero. Patch by Victor Stinner.
- gh-111696 [https://github.com/python/cpython/issues/111696]: Add support for `%T`, `%T#`, `%N` and `%N#` formats to **`PyUnicode_FromFormat()`**: format the fully qualified name of an object type and of a type: call **`PyType_GetModuleName()`**. See **PEP 737** [https://peps.python.org/pep-0737/] for more information. Patch by Victor Stinner.
- gh-111696 [https://github.com/python/cpython/issues/111696]: Add **`PyType_GetModuleName()`** function to get the type's module name. Equivalent to getting the `type.__module__` attribute. Patch by Eric Snow and Victor Stinner.
- gh-111696 [https://github.com/python/cpython/issues/111696]: Add **`PyType_GetFullyQualifiedName()`** function to get the type's fully qualified name. Equivalent to `f"{type.__module__}.{type.__qualname__}"`, or `type.__qualname__` if `type.__module__` is not a string or is equal to `"builtins"`. Patch by Victor Stinner.
- gh-85283 [https://github.com/python/cpython/issues/85283]: The `fcntl`, `grp`, `pwd`, `termios`, `_statistics` and `_testconsole` C extensions are now built with the limited C API. Patch by Victor Stinner.
- gh-111140 [https://github.com/python/cpython/issues/111140]: Add additional flags to **`PyLong_AsNativeBytes()`** and **`PyLong_FromNativeBytes()`** to allow the caller to determine

how to handle edge cases around values that fill the entire buffer.

- gh-113024 [https://github.com/python/cpython/issues/113024]: Add `PyObject_GenericHash()` function.

Python 3.13.0 alpha 5

Release date: 2024-03-12

Security

- gh-115398 [https://github.com/python/cpython/issues/115398]: Allow controlling Expat >=2.6.0 reparse deferral (**CVE 2023-52425** [https://www.cve.org/CVERecord?id=CVE-2023-52425]) by adding five new methods:
  - `xml.etree.ElementTree.XMLParser.flush()`
  - `xml.etree.ElementTree.XMLPullParser.flush()`
  - `xml.parsers.expat.xmlparser.GetReparseDeferralEnabled()`
  - `xml.parsers.expat.xmlparser.SetReparseDeferralEnabled()`
  - `xml.sax.expatreader.ExpatParser.flush()`
- gh-114572 [https://github.com/python/cpython/issues/114572]: `ssl.SSLContext.cert_store_stats()` and `ssl.SSLContext.get_ca_certs()` now correctly lock access to the certificate store, when the `ssl.SSLContext` is shared across multiple threads.

Core and Builtins

- gh-116604 [https://github.com/python/cpython/issues/116604]: Respect the status of the garbage collector when indirect calls are made via `PyErr_CheckSignals()` and the evaluation breaker. Patch by Pablo Galindo
- gh-112087 [https://github.com/python/cpython/issues/112087]: `list` is now compatible with the implementation of **PEP 703** [https://peps.python.org/pep-0703/].

- gh-116381 [https://github.com/python/cpython/issues/116381]: Add specialization for `CONTAINS_OP`.
- gh-116296 [https://github.com/python/cpython/issues/116296]: Fix possible refleak in **`object.__reduce__()`** internal error handling.
- gh-115823 [https://github.com/python/cpython/issues/115823]: Properly calculate error ranges in the parser when raising **`SyntaxError`** exceptions caused by invalid byte sequences. Patch by Pablo Galindo
- gh-115778 [https://github.com/python/cpython/issues/115778]: Add `tierN` annotation for instruction definition in interpreter DSL.
- gh-115733 [https://github.com/python/cpython/issues/115733]: Fix crash when calling `next()` on exhausted list iterators.
- gh-115700 [https://github.com/python/cpython/issues/115700]: The regen-cases build stage now works on Windows.
- gh-115347 [https://github.com/python/cpython/issues/115347]: Fix bug where docstring was replaced by a redundant NOP when Python is run with `-OO`.
- gh-115323 [https://github.com/python/cpython/issues/115323]: Make error message more meaningful for when **`bytearray.extend()`** is called with a **`str`** object.
- gh-112175 [https://github.com/python/cpython/issues/112175]: Every `PyThreadState` now has its own `eval_breaker`, allowing specific threads to be interrupted.
- gh-115154 [https://github.com/python/cpython/issues/115154]: Fix a bug that was causing the **`tokenize.untokenize()`** function to handle unicode named literals incorrectly. Patch by Pablo Galindo
- gh-112433 [https://github.com/python/cpython/issues/112433]: Add ability to force alignment of **`ctypes.Structure`** by way of the new `_align_` attribute on the class.
- gh-104090 [https://github.com/python/cpython/issues/104090]: The multiprocessing resource tracker now exits with non-zero status code if a resource leak was detected. It still exits with status code 0 otherwise.

- gh-105858 [https://github.com/python/cpython/issues/105858]: Improve the constructors for **`ast`** nodes. Arguments of list types now default to an empty list if omitted, and optional fields default to `None`. AST nodes now have an `__annotations__` attribute with the expected types of their attributes. Passing unrecognized extra arguments to AST nodes is deprecated and will become an error in Python 3.15. Omitting a required argument to an AST node is deprecated and will become an error in Python 3.15. Patch by Jelle Zijlstra.
- gh-101860 [https://github.com/python/cpython/issues/101860]: Expose `__name__` attribute on property.
- gh-96497 [https://github.com/python/cpython/issues/96497]: Fix incorrect resolution of mangled class variables used in assignment expressions in comprehensions.

Library

- gh-116600 [https://github.com/python/cpython/issues/116600]: Fix **`repr()`** for global **`Flag`** members.

- gh-116349 [https://github.com/python/cpython/issues/116349]: **`platform.java_ver()`** is deprecated and will be removed in 3.15. It was largely untested, had a confusing API, and was only useful for Jython support.

- gh-116143 [https://github.com/python/cpython/issues/116143]: Fix a race in pydoc `_start_server`, eliminating a window in which `_start_server` can return a thread that is “serving” but without a `docserver` set.

- gh-116127 [https://github.com/python/cpython/issues/116127]: **`typing`**: implement **PEP 705** [https://peps.python.org/pep-0705/] which adds **`typing.ReadOnly`** support to **`typing.TypedDict`**.

- gh-116325 [https://github.com/python/cpython/issues/116325]: `typing`: raise `SyntaxError` instead of `AttributeError` on forward references as empty strings.

- gh-115957 [https://github.com/python/cpython/issues/115957]: When `asyncio.TaskGroup.create_task` is called on an inactive `asyncio.TaskGroup`, the given coroutine will be closed (which prevents a `RuntimeWarning`).

- gh-115978 [https://github.com/python/cpython/issues/115978]: Disable preadv(), readv(), pwritev(), and writev() on WASI.

  Under wasmtime for WASI 0.2, these functions don’t pass test_posix (https://github.com/bytecodealliance/wasmtime/issues/7830).

- gh-88352 [https://github.com/python/cpython/issues/88352]: Fix the computation of the next rollover time in the **`logging.TimedRotatingFileHandler`** handler. **`computeRollover()`** now always returns a timestamp larger than the specified time and works correctly during the DST change. **`doRollover()`** no longer overwrite the already rolled over file, saving from data loss when run at midnight or during repeated time at the DST change.

- gh-87115 [https://github.com/python/cpython/issues/87115]: Set `__main__.__spec__` to `None` when running a script with `pdb`

- gh-76511 [https://github.com/python/cpython/issues/76511]: Fix UnicodeEncodeError in **`email.Message.as_string()`** that results when a message that claims to be in the ascii character set actually has non-ascii characters. Non-ascii characters are now replaced with the U+FFFD replacement character, like in the `replace` error handler.

- gh-89547 [https://github.com/python/cpython/issues/89547]: Add support for nested typing special forms like Final[ClassVar[int]].

- gh-65824 [https://github.com/python/cpython/issues/65824]: Improve the `less` prompt in **`pydoc`**.

- gh-116040 [https://github.com/python/cpython/issues/116040]: [Enum] fix by-value calls when second value is falsey; e.g. Cardinal(1, 0)

- gh-115821 [https://github.com/python/cpython/issues/115821]: [Enum] Improve error message when calling super().__new__() in custom __new__.

- gh-85644 [https://github.com/python/cpython/issues/85644]: Use the `XDG_CURRENT_DESKTOP` environment variable in **`webbrowser`** to check desktop. Prefer it to the deprecated `GNOME_DESKTOP_SESSION_ID` for GNOME detection.

- gh-75988 [https://github.com/python/cpython/issues/75988]: Fixed **`unittest.mock.create_autospec()`** to pass the call through to the wrapped object to return the real result.

- gh-115881 [https://github.com/python/cpython/issues/115881]: Fix issue where **`ast.parse()`** would incorrectly flag conditional context managers (such as `with (x() if y else z()): ...`) as invalid syntax if `feature_version=(3, 8)` was passed. This reverts changes to the grammar made as part of gh-94949 [https://github.com/python/cpython/issues/94949].

- gh-115886 [https://github.com/python/cpython/issues/115886]: Fix silent truncation of the name with an embedded null character in **`multiprocessing.shared_memory.SharedMemory`**.

- gh-115532 [https://github.com/python/cpython/issues/115532]: Add kernel density estimation to the statistics module.

- gh-115714 [https://github.com/python/cpython/issues/115714]: On WASI, the **`time`** module no longer get process time using `times()` or `CLOCK_PROCESS_CPUTIME_ID`, system API is that is unreliable and is likely to be removed from WASI. The affected clock functions fall back to calling `clock()`.

- gh-115809 [https://github.com/python/cpython/issues/115809]: Improve algorithm for computing which rolled-over log files to delete in **`logging.TimedRotatingFileHandler`**. It is now reliable for handlers without `namer` and with arbitrary deterministic `namer` that leaves the datetime part in the file name unmodified.

- gh-74668 [https://github.com/python/cpython/issues/74668]: **`urllib.parse`** functions **`parse_qs()`** and **`parse_qsl()`** now support bytes arguments containing raw and percent-encoded non-ASCII data.

- gh-67044 [https://github.com/python/cpython/issues/67044]: **`csv.writer()`** now always quotes or escapes `'\r'` and `'\n'`, regardless of *lineterminator* value.

- gh-115712 [https://github.com/python/cpython/issues/115712]: Restore support of space delimiter with `skipinitialspace=True` in **`csv`**. **`csv.writer()`** now quotes empty fields if delimiter is a space and skipinitialspace is true and raises exception if quoting is not possible.

- gh-112364 [https://github.com/python/cpython/issues/112364]: Fixed **`ast.unparse()`** to handle format_spec with `"`, `'` or `\\`. Patched by Frank Hoffmann.

- gh-112997 [https://github.com/python/cpython/issues/112997]: Stop logging potentially sensitive callback arguments in **`asyncio`** unless debug mode is active.

- gh-114914 [https://github.com/python/cpython/issues/114914]: Fix an issue where an abandoned **`StreamWriter`** would not be garbage collected.
- gh-111358 [https://github.com/python/cpython/issues/111358]: Fix a bug in **`asyncio.BaseEventLoop.shutdown_default_executor()`** to ensure the timeout passed to the coroutine behaves as expected.
- gh-115618 [https://github.com/python/cpython/issues/115618]: Fix improper decreasing the reference count for `None` argument in **`property`** methods **`getter()`**, **`setter()`** and **`deleter()`**.
- gh-112720 [https://github.com/python/cpython/issues/112720]: Refactor **`dis.ArgResolver`** to make it possible to subclass and change the way jump args are interpreted.
- gh-112006 [https://github.com/python/cpython/issues/112006]: Fix **`inspect.unwrap()`** for types with the `__wrapper__` data descriptor. Fix **`inspect.Signature.from_callable()`** for builtins **`classmethod()`** and **`staticmethod()`**.
- gh-101293 [https://github.com/python/cpython/issues/101293]: Support callables with the `__call__()` method and types with `__new__()` and `__init__()` methods set to class methods, static methods, bound methods, partial functions, and other types of methods and descriptors in **`inspect.Signature.from_callable()`**.
- gh-103092 [https://github.com/python/cpython/issues/103092]: Isolate **`_lsprof`** (apply **PEP 687** [https://peps.python.org/pep-0687/]).
- gh-113942 [https://github.com/python/cpython/issues/113942]: **`pydoc`** no longer skips global functions implemented as builtin

methods, such as **`MethodDescriptorType`** and **`WrapperDescriptorType`**.

- gh-115256 [https://github.com/python/cpython/issues/115256]: Added DeprecationWarning when accessing the tarfile attribute of TarInfo objects. The attribute is never used internally and is only attached to TarInfos when the tarfile is opened in write-mode, not read-mode. The attribute creates an unnecessary reference cycle which may cause corruption when not closing the handle after writing a tarfile.

- gh-115197 [https://github.com/python/cpython/issues/115197]: `urllib.request` no longer resolves the hostname before checking it against the system's proxy bypass list on macOS and Windows.

- gh-113812 [https://github.com/python/cpython/issues/113812]: **`DatagramTransport.sendto()`** will now send zero-length datagrams if called with an empty bytes object. The transport flow control also now accounts for the datagram header when calculating the buffer size.

- gh-114763 [https://github.com/python/cpython/issues/114763]: Protect modules loaded with **`importlib.util.LazyLoader`** from race conditions when multiple threads try to access attributes before the loading is complete.

- gh-114709 [https://github.com/python/cpython/issues/114709]: **`posixpath.commonpath()`** now raises a **`ValueError`** exception when passed an empty iterable. Previously, **`IndexError`** was raised.

  `posixpath.commonpath()` now raises a **`TypeError`** exception when passed `None`. Previously, **`ValueError`** was raised.

- gh-114610 [https://github.com/python/cpython/issues/114610]: Fix bug where `pathlib.PurePath.with_stem()` converted a non-empty path suffix to a stem when given an empty *stem* argument. It now raises `ValueError`, just like `pathlib.PurePath.with_suffix()` does when called on a path with an empty stem, given a non-empty *suffix* argument.

- gh-107361 [https://github.com/python/cpython/issues/107361]: Add `ssl.VERIFY_X509_PARTIAL_CHAIN` and **`VERIFY_X509_STRICT`** to the default SSL context created with `ssl.create_default_context()`.

- gh-112281 [https://github.com/python/cpython/issues/112281]: Allow creating union of types for `typing.Annotated` with unhashable metadata.

- gh-111775 [https://github.com/python/cpython/issues/111775]: Fix **`importlib.resources.simple.ResourceHandle.open()`** for text mode, added missed `stream` argument.

- gh-90095 [https://github.com/python/cpython/issues/90095]: Make .pdbrc and -c work with any valid pdb commands.

- gh-107625 [https://github.com/python/cpython/issues/107625]: Raise `configparser.ParsingError` from `read()` and `read_file()` methods of `configparser.ConfigParser` if a key without a corresponding value is continued (that is, followed by an indented line).

- gh-107155 [https://github.com/python/cpython/issues/107155]: Fix incorrect output of `help(x)` where `x` is a `lambda` function, which has an `__annotations__` dictionary attribute with a `"return"` key.

- gh-57141 [https://github.com/python/cpython/issues/57141]: Add option for *non-shallow* comparisons to `filecmp.dircmp` like

`filecmp.cmp()`. Original patch by Steven Ward. Enhanced by Tobias Rautenkranz

- gh-69990 [https://github.com/python/cpython/issues/69990]: **`Profile.print_stats()`** has been improved to accept multiple sort arguments. Patched by Chiu-Hsiang Hsu and Furkan Onder.

- gh-104061 [https://github.com/python/cpython/issues/104061]: Add **`socket.SO_BINDTOIFINDEX`** constant.

- gh-60346 [https://github.com/python/cpython/issues/60346]: Fix ArgumentParser inconsistent with parse_known_args.

- gh-102389 [https://github.com/python/cpython/issues/102389]: Add `windows_31j` to aliases for `cp932` codec

- gh-72249 [https://github.com/python/cpython/issues/72249]: Always include the module name in the **`repr()`** of **`functools.partial()`** objects. Patch by Furkan Onder and Anilyka Barry.

- gh-100985 [https://github.com/python/cpython/issues/100985]: Update HTTPSConnection to consistently wrap IPv6 Addresses when using a proxy.

- gh-100884 [https://github.com/python/cpython/issues/100884]: email: fix misfolding of comma in address-lists over multiple lines in combination with unicode encoding.

- gh-95782 [https://github.com/python/cpython/issues/95782]: Fix **`io.BufferedReader.tell()`**, **`io.BufferedReader.seek()`**, **`_pyio.BufferedReader.tell()`**, **`io.BufferedRandom.tell()`**, **`io.BufferedRandom.seek()`** and **`_pyio.BufferedRandom.tell()`** being able to return negative offsets.

- gh-96310 [https://github.com/python/cpython/issues/96310]: Fix a traceback in `argparse` when all options in a mutually exclusive group are suppressed.

- gh-93205 [https://github.com/python/cpython/issues/93205]: Fixed a bug in `logging.handlers.TimedRotatingFileHandler` where multiple rotating handler instances pointing to files with the same name but different extensions would conflict and not delete the correct files.

- bpo-31116 [https://bugs.python.org/issue?@action=redirect&bpo=31116]: Add Z85 encoding to `base64`.

- bpo-44865 [https://bugs.python.org/issue?@action=redirect&bpo=44865]: Add missing call to localization function in `argparse`.

- bpo-43952 [https://bugs.python.org/issue?@action=redirect&bpo=43952]: Fix `multiprocessing.connection.Listener.accept()` to accept empty bytes as authkey. Not accepting empty bytes as key causes it to hang indefinitely.

- bpo-42125 [https://bugs.python.org/issue?@action=redirect&bpo=42125]: linecache: get module name from `__spec__` if available. This allows getting source code for the `__main__` module when a custom loader is used.

- bpo-41122 [https://bugs.python.org/issue?@action=redirect&bpo=41122]: Failing to pass arguments properly to `functools.singledispatchmethod()` now throws a TypeError instead of hitting an index out of bounds internally.

- bpo-40818 [https://bugs.python.org/issue?@action=redirect&bpo=40818]: The asyncio REPL now runs

`sys.__interactivehook__` on startup. The default implementation of **`sys.__interactivehook__`** provides auto-completion to the asyncio REPL. Patch contributed by Rémi Lapeyre.

- bpo-33775 [https://bugs.python.org/issue?@action=redirect&bpo=33775]: Add ‘default’ and ‘version’ help text for localization in argparse.

Documentation

- gh-115399 [https://github.com/python/cpython/issues/115399]: Document **CVE 2023-52425** [https://www.cve.org/CVERecord?id=CVE-2023-52425] of Expat <2.6.0 under “XML vulnerabilities”.
- gh-109653 [https://github.com/python/cpython/issues/109653]: Improve import time of **`uuid`** on Linux.

Tests

- gh-71052 [https://github.com/python/cpython/issues/71052]: Add test exclusions to support running the test suite on Android.
- gh-71052 [https://github.com/python/cpython/issues/71052]: Enable `test_concurrent_futures` on platforms that support threading but not multiprocessing.
- gh-115796 [https://github.com/python/cpython/issues/115796]: Make ‘_testinternalcapi.assemble_code_object’ construct the exception table for the code object.
- gh-115720 [https://github.com/python/cpython/issues/115720]: Leak tests (`-R`, `--huntrleaks`) now show a summary of the number of leaks found in each iteration.
- gh-115122 [https://github.com/python/cpython/issues/115122]: Add `--bisect` option to regrtest test runner: run failed tests with `test.bisect_cmd` to identify failing tests. Patch by Victor Stinner.

- gh-115596 [https://github.com/python/cpython/issues/115596]: Fix `ProgramPriorityTests` in `test_os` permanently changing the process priority.
- gh-115556 [https://github.com/python/cpython/issues/115556]: On Windows, commas passed in arguments to `Tools\buildbot\test.bat` and `PCbuild\\rt.bat` are now properly handled.
- gh-115420 [https://github.com/python/cpython/issues/115420]: Fix translation of exception handler targets by `_testinternalcapi.optimize_cfg`.
- gh-115376 [https://github.com/python/cpython/issues/115376]: Fix segfault in `_testinternalcapi.compiler_codegen` on bad input.

Build

- gh-116313 [https://github.com/python/cpython/issues/116313]: Get WASI builds to work under wasmtime 18 w/ WASI 0.2/preview2 primitives.
- gh-71052 [https://github.com/python/cpython/issues/71052]: Change Android's **`sys.platform`** from `"linux"` to `"android"`.
- gh-116117 [https://github.com/python/cpython/issues/116117]: Backport `libb2`'s PR #42 to fix compiling CPython on 32-bit Windows with `clang-cl`.
- gh-71052 [https://github.com/python/cpython/issues/71052]: Fix several Android build issues
- gh-114099 [https://github.com/python/cpython/issues/114099]: A testbed project was added to run the test suite on iOS.
- gh-115350 [https://github.com/python/cpython/issues/115350]: Fix building ctypes module with -DWIN32_LEAN_AND_MEAN defined
- gh-111225 [https://github.com/python/cpython/issues/111225]: Link extension modules against libpython on Android.
- gh-115737 [https://github.com/python/cpython/issues/115737]: The install name for libPython is now correctly set for non-framework macOS builds.

- gh-114099 [https://github.com/python/cpython/issues/114099]: Makefile targets were added to support compiling an iOS-compatible framework build.

Windows

- gh-116012 [https://github.com/python/cpython/issues/116012]: Ensure the value of `GetLastError()` is preserved across GIL operations.
- gh-115582 [https://github.com/python/cpython/issues/115582]: Building extensions intended for free-threaded builds of CPython now require compiling with `/DPy_GIL_DISABLED` manually when using a regular install. This is expected to change in future releases.
- gh-115554 [https://github.com/python/cpython/issues/115554]: The installer now has more strict rules about updating the Python install manager. In general, most users only have a single launcher installed and will see no difference. When multiple launchers have been installed, the option to install the launcher is disabled until all but one have been removed. Downgrading the launcher (which was never allowed) is now more obviously blocked.
- gh-115543 [https://github.com/python/cpython/issues/115543]: Python install manager can now detect Python 3.13 when installed from the Microsoft Store, and will install Python 3.12 by default when **`PYLAUNCHER_ALLOW_INSTALL`** is set.

macOS

- gh-116145 [https://github.com/python/cpython/issues/116145]: Update macOS installer to Tcl/Tk 8.6.14.

IDLE

- gh-88516 [https://github.com/python/cpython/issues/88516]: On macOS show a proxy icon in the title bar of editor windows to match platform behaviour.

Tools/Demos

- gh-100176 [https://github.com/python/cpython/issues/100176]: Remove outdated Tools/{io,cc,string}bench
- bpo-45101 [https://bugs.python.org/issue?@action=redirect&bpo=45101]: Add consistency in usage message IO between 2 versions of python-config.

C API

- gh-114626 [https://github.com/python/cpython/issues/114626]: Add again `_PyCFunctionFastWithKeywords` name, removed in Python 3.13 alpha 4 by mistake. Keep the old private `_PyCFunctionFastWithKeywords` name (Python 3.7) as an alias to the new public name `PyCFunctionFastWithKeywords` (Python 3.13a4). Patch by Victor Stinner.
- gh-111418 [https://github.com/python/cpython/issues/111418]: Add **`PyHASH_MODULUS`**, **`PyHASH_BITS`**, **`PyHASH_INF`** and **`PyHASH_IMAG`** C macros. Patch by Sergey B Kirpichev.

Python 3.13.0 alpha 4

Release date: 2024-02-15

Security

- gh-115399 [https://github.com/python/cpython/issues/115399]: Update bundled libexpat to 2.6.0
- gh-115243 [https://github.com/python/cpython/issues/115243]: Fix possible crashes in **`collections.deque.index()`** when the deque is concurrently modified.

Core and Builtins

- gh-112087 [https://github.com/python/cpython/issues/112087]: For an empty reverse iterator for list will be reduced to `reversed()`. Patch by Donghee Na
- gh-114570 [https://github.com/python/cpython/issues/114570]: Add `PythonFinalizationError` exception. This exception derived from `RuntimeError` is raised when an operation is blocked during the Python finalization. Patch by Victor Stinner.
- gh-114695 [https://github.com/python/cpython/issues/114695]: Add `sys._clear_internal_caches()`, which clears all internal performance-related caches (and deprecate the less-general `sys._clear_type_cache()` function).
- gh-114828 [https://github.com/python/cpython/issues/114828]: Fix compilation crashes in uncommon code examples using `super()` inside a comprehension in a class body.
- gh-112069 [https://github.com/python/cpython/issues/112069]: Adapt `set` and `frozenset` methods to Argument Clinic.
- gh-115011 [https://github.com/python/cpython/issues/115011]: Setters for members with an unsigned integer type now support the same range of valid values for objects that has a `__index__()` method as for `int`.
- gh-114887 [https://github.com/python/cpython/issues/114887]: Changed socket type validation in `create_datagram_endpoint()` to accept all non-stream sockets. This fixes a regression in compatibility with raw sockets.
- gh-114944 [https://github.com/python/cpython/issues/114944]: Fixes a race between `PyParkingLot_Park` and `_PyParkingLot_UnparkAll`.
- gh-113462 [https://github.com/python/cpython/issues/113462]: Limit the number of versions that a single class can use. Prevents a few wayward classes using up all the version numbers.

- gh-76763 [https://github.com/python/cpython/issues/76763]: The `chr()` builtin function now always raises `ValueError` for values outside the valid range. Previously it raised `OverflowError` for very large or small values.
- gh-114806 [https://github.com/python/cpython/issues/114806]: No longer specialize calls to classes, if those classes have metaclasses. Fixes bug where the `__call__` method of the metaclass was not being called.
- gh-107944 [https://github.com/python/cpython/issues/107944]: Improve error message for function calls with bad keyword arguments via getargs
- gh-112529 [https://github.com/python/cpython/issues/112529]: The free-threaded build no longer allocates space for the `PyGC_Head` structure in objects that support cyclic garbage collection. A number of other fields and data structures are used as replacements, including `ob_gc_bits`, `ob_tid`, and mimalloc internal data structures.
- gh-114456 [https://github.com/python/cpython/issues/114456]: Lower the recursion limit under a debug build of WASI.
- gh-114083 [https://github.com/python/cpython/issues/114083]: Compiler applies folding of LOAD_CONST with following instruction in a separate pass before other optimisations. This enables jump threading in certain circumstances.
- gh-114388 [https://github.com/python/cpython/issues/114388]: Fix a `RuntimeWarning` emitted when assign an integer-like value that is not an instance of `int` to an attribute that corresponds to a C struct member of type T_UINT and T_ULONG. Fix a double `RuntimeWarning` emitted when assign a negative integer value to an attribute that corresponds to a C struct member of type T_UINT.
- gh-114265 [https://github.com/python/cpython/issues/114265]: Compiler propagates line numbers before optimization, leading to more optimization opportunities and removing the need for the `guarantee_lineno_for_exits` hack.

- gh-112529 [https://github.com/python/cpython/issues/112529]: The free-threaded build now has its own thread-safe GC implementation that uses mimalloc to find GC tracked objects. It is non-generational, unlike the existing GC implementation.
- gh-114050 [https://github.com/python/cpython/issues/114050]: Fix segmentation fault caused by an incorrect format string in `TypeError` exception when more than two arguments are passed to `int`.
- gh-112354 [https://github.com/python/cpython/issues/112354]: The `END_FOR` instruction now pops only one value. This is to better support side exits in loops.
- gh-113884 [https://github.com/python/cpython/issues/113884]: Make **`queue.SimpleQueue`** thread safe when the GIL is disabled.
- gh-114058 [https://github.com/python/cpython/issues/114058]: Implement the foundations of the Tier 2 redundancy eliminator.
- gh-113939 [https://github.com/python/cpython/issues/113939]: frame.clear(): Clear frame.f_locals as well, and not only the fast locals. This is relevant once frame.f_locals was accessed, which would contain also references to all the locals.
- gh-112050 [https://github.com/python/cpython/issues/112050]: Convert **`collections.deque`** to use Argument Clinic.
- gh-112050 [https://github.com/python/cpython/issues/112050]: Make methods on **`collections.deque`** thread-safe when the GIL is disabled.
- gh-113464 [https://github.com/python/cpython/issues/113464]: Add an option (`--enable-experimental-jit` for `configure`-based builds or `--experimental-jit` for `PCbuild`-based ones) to build an *experimental* just-in-time compiler, based on copy-and-patch [https://fredrikbk.com/publications/copy-and-patch.pdf]
- gh-113055 [https://github.com/python/cpython/issues/113055]: Make interp->obmalloc a pointer. For interpreters that share state with the main interpreter, this points to the same

static memory structure. For interpreters with their own obmalloc state, it is heap allocated. Add free_obmalloc_arenas() which will free the obmalloc arenas and radix tree structures for interpreters with their own obmalloc state.

- gh-55664 [https://github.com/python/cpython/issues/55664]: Add warning when creating **`type`** using a namespace dictionary with non-string keys. Patched by Daniel Urban and Furkan Onder.
- gh-104530 [https://github.com/python/cpython/issues/104530]: Use native Win32 condition variables.

Library

- gh-115392 [https://github.com/python/cpython/issues/115392]: Fix a bug in **`doctest`** where incorrect line numbers would be reported for decorated functions.

- gh-114563 [https://github.com/python/cpython/issues/114563]: Fix several **`format()`** bugs when using the C implementation of **`Decimal`**: * memory leak in some rare cases when using the `z` format option (coerce negative 0) * incorrect output when applying the `z` format option to type `F` (fixed-point with capital `NAN` / `INF`) * incorrect output when applying the `#` format option (alternate form)

- gh-102840 [https://github.com/python/cpython/issues/102840]: Fix confused traceback when floordiv, mod, or divmod operations happens between instances of **`fractions.Fraction`** and **`complex`**.

- gh-115165 [https://github.com/python/cpython/issues/115165]: Most exceptions are now ignored when attempting to set the `__orig_class__` attribute on objects returned when calling **`typing`** generic aliases (including generic aliases created using `typing.Annotated`). Previously only `AttributeError` was ignored. Patch by Dave Shawley.

- gh-112903 [https://github.com/python/cpython/issues/112903]: Fix “issubclass() arg 1 must be a class” errors in certain cases of multiple inheritance with generic aliases (regression in early 3.13 alpha releases).

- gh-115133 [https://github.com/python/cpython/issues/115133]: Fix tests for `XMLPullParser` with Expat 2.6.0.

- gh-115059 [https://github.com/python/cpython/issues/115059]: **`io.BufferedRandom.read1()`** now flushes the underlying write buffer.

- gh-79382 [https://github.com/python/cpython/issues/79382]: Trailing `**` no longer allows to match files and non-existing paths in recursive `glob()`.

- gh-67837 [https://github.com/python/cpython/issues/67837]: Avoid race conditions in the creation of directories during concurrent extraction in `tarfile` and `zipfile`.

- gh-115060 [https://github.com/python/cpython/issues/115060]: Speed up `pathlib.Path.glob()` by removing redundant regex matching.

- gh-97928 [https://github.com/python/cpython/issues/97928]: Partially revert the behavior of **`tkinter.Text.count()`**. By default it preserves the behavior of older Python versions, except that setting `wantobjects` to 0 no longer has effect. Add a new parameter *return_ints*: specifying `return_ints=True` makes `Text.count()` always returning the single count as an integer instead of a 1-tuple or `None`.

- gh-114628 [https://github.com/python/cpython/issues/114628]: When csv.Error is raised when handling TypeError, do not print the TypeError traceback.

- gh-85984 [https://github.com/python/cpython/issues/85984]: Added `_POSIX_VDISABLE` from C’s `<unistd.h>` to **`termios`**.

- gh-114965 [https://github.com/python/cpython/issues/114965]: Update bundled pip to 24.0

- gh-114959 [https://github.com/python/cpython/issues/114959]: **`tarfile`** no longer ignores errors when trying to extract a directory on top of a file.

- gh-114894 [https://github.com/python/cpython/issues/114894]: Add **`array.array.clear()`**.

- gh-114071 [https://github.com/python/cpython/issues/114071]: Support tuple subclasses using auto() for enum member value.

- gh-109475 [https://github.com/python/cpython/issues/109475]: Fix support of explicit option value “–” in **`argparse`** (e.g. `--option=--`).

- gh-49766 [https://github.com/python/cpython/issues/49766]: Fix **`date`**-**`datetime`** comparison. Now the special comparison methods like `__eq__` and `__lt__` return **`NotImplemented`** if one of comparands is **`date`** and other is **`datetime`** instead of ignoring the time part and the time zone or forcefully return “not equal” or raise **`TypeError`**. It makes comparison of **`date`** and **`datetime`** subclasses more symmetric and allows to change the default behavior by overriding the special comparison methods in subclasses.

- gh-110190 [https://github.com/python/cpython/issues/110190]: Fix ctypes structs with array on Windows ARM64 platform by setting `MAX_STRUCT_SIZE` to 32 in stgdict. Patch by Diego Russo

- gh-114678 [https://github.com/python/cpython/issues/114678]: Ensure that deprecation warning for ‘N’ specifier in **`Decimal`** format is not raised for cases where ‘N’ appears in other places in the format specifier. Based on patch by Stefan Krah.

- gh-70303 [https://github.com/python/cpython/issues/70303]: Return both files and directories from **`pathlib.Path.glob()`** if a pattern ends with “`**`”. Previously only directories were returned.

- gh-109653 [https://github.com/python/cpython/issues/109653]: Improve import time of **`importlib.metadata`** and **`email.utils`**.

- gh-113280 [https://github.com/python/cpython/issues/113280]: Fix a leak of open socket in rare cases when error occurred in **`ssl.SSLSocket`** creation.

- gh-77749 [https://github.com/python/cpython/issues/77749]: **`email.policy.EmailPolicy.fold()`** now always encodes non-ASCII characters in headers if **`utf8`** is false.

- gh-83383 [https://github.com/python/cpython/issues/83383]: Synchronization of the **`dbm.dumb`** database is now no-op if there was no modification since opening or last synchronization. The directory file for a newly created empty **`dbm.dumb`** database is now created immediately after opening instead of deferring this until synchronizing or closing.

- gh-91602 [https://github.com/python/cpython/issues/91602]: Add *filter* keyword-only parameter to [`sqlite3.Connection.iterdump()`] for filtering database objects to dump. Patch by Mariusz Felisiak.

- gh-112451 [https://github.com/python/cpython/issues/112451]: Prohibit subclassing pure-Python **`datetime.timezone`**. This is consistent with C-extension implementation. Patch by Mariusz Felisiak.

- gh-69893 [https://github.com/python/cpython/issues/69893]: Add the **`close()`** method for the iterator returned by **`xml.etree.ElementTree.iterparse()`**.

- gh-109653 [https://github.com/python/cpython/issues/109653]: Reduce the import time of **`threading`** module by ~50%. Patch by Daniel Hollas.

- gh-114492 [https://github.com/python/cpython/issues/114492]: Make the result of **`termios.tcgetattr()`** reproducible on Alpine Linux. Previously it could leave a random garbage in some fields.

- gh-114315 [https://github.com/python/cpython/issues/114315]: Make **`threading.Lock`** a real class, not a factory function. Add `__new__` to `_thread.lock` type.

- gh-100414 [https://github.com/python/cpython/issues/100414]: Add **`dbm.sqlite3`** as a backend to **`dbm`**, and make it the new default **`dbm`** backend. Patch by Raymond Hettinger and Erlend E. Aasland.

- gh-113267 [https://github.com/python/cpython/issues/113267]: Revert changes in gh-106584 [https://github.com/python/cpython/issues/106584] which made calls of `TestResult` methods `startTest()` and `stopTest()` unbalanced.

- gh-75128 [https://github.com/python/cpython/issues/75128]: Ignore an `OSError` in **`asyncio.BaseEventLoop.create_server()`** when IPv6 is available but the interface cannot actually support it.

- gh-114423 [https://github.com/python/cpython/issues/114423]: `_DummyThread` entries in `threading._active` are now automatically removed when the related thread dies.

- gh-114257 [https://github.com/python/cpython/issues/114257]: Dismiss the **`FileNotFound`** error in `ctypes.util.find_library()` and just return `None` on Linux.

- gh-114321 [https://github.com/python/cpython/issues/114321]: Expose more platform specific constants in the `fcntl` module on Linux, macOS, FreeBSD and NetBSD.

- gh-114328 [https://github.com/python/cpython/issues/114328]: The `tty.setcbreak()` and new `tty.cfmakecbreak()` no longer clears the terminal input ICRLF flag. This fixes a regression introduced in 3.12 that no longer matched how OSes define cbreak mode in their `stty(1)` manual pages.

- gh-114281 [https://github.com/python/cpython/issues/114281]: Remove type hints from `Lib/asyncio/staggered.py`. The annotations in the typeshed [https://github.com/python/typeshed] project should be used instead.

- gh-101438 [https://github.com/python/cpython/issues/101438]: Avoid reference cycle in ElementTree.iterparse. The iterator returned by `ElementTree.iterparse` may hold on to a file descriptor. The reference cycle prevented prompt clean-up of the file descriptor if the returned iterator was not exhausted.

- gh-114198 [https://github.com/python/cpython/issues/114198]: The signature for the `__replace__` method on **`dataclasses`** now has the first argument named `self`, rather than `obj`.

- gh-104522 [https://github.com/python/cpython/issues/104522]: **`OSError`** raised when run a subprocess now only has *filename* attribute set to *cwd* if the error was caused by a failed attempt to change the current directory.

- gh-114149 [https://github.com/python/cpython/issues/114149]: Enum: correctly handle tuple subclasses in custom `__new__`.

- gh-83648 [https://github.com/python/cpython/issues/83648]: Support deprecation of options, positional arguments and subcommands in **`argparse`**.

- gh-114087 [https://github.com/python/cpython/issues/114087]: Speed up `dataclasses.asdict` up to 1.35x.

- gh-109534 [https://github.com/python/cpython/issues/109534]: Fix a reference leak in **`asyncio.selector_events.BaseSelectorEventLoop`** when SSL handshakes fail. Patch contributed by Jamie Phan.

- gh-79634 [https://github.com/python/cpython/issues/79634]: Accept path-like objects as patterns in **`pathlib.Path.glob()`** and **`rglob()`**.

- gh-112202 [https://github.com/python/cpython/issues/112202]: Ensure that a **`asyncio.Condition.notify()`** call does not get lost if the awakened `Task` is simultaneously cancelled or encounters any other error.

- gh-113951 [https://github.com/python/cpython/issues/113951]: Fix the behavior of `tag_unbind()` methods of **`tkinter.Text`** and **`tkinter.Canvas`** classes with three arguments. Previously, `widget.tag_unbind(tag, sequence, funcid)` destroyed the current binding for *sequence*, leaving *sequence* unbound, and deleted the *funcid* command. Now it removes only *funcid* from the binding for *sequence*, keeping other commands, and deletes the *funcid* command. It leaves *sequence* unbound only if *funcid* was the last bound command.

- gh-97959 [https://github.com/python/cpython/issues/97959]: Fix rendering class methods, bound methods, method and function aliases in **`pydoc`**. Class methods no longer have “method of builtins.type instance” note. Corresponding notes are now added for class and unbound methods. Method and function aliases now have references to the module or the class where the origin was defined if it differs from the current. Bound methods are now listed in the static methods section. Methods of builtin classes are now supported as well as methods of Python classes.

- gh-113796 [https://github.com/python/cpython/issues/113796]: Add more validation checks in the **`csv.Dialect`** constructor. **`ValueError`** is now raised if the same character is used in different roles.

- gh-113732 [https://github.com/python/cpython/issues/113732]: Fix support of **`QUOTE_NOTNULL`** and **`QUOTE_STRINGS`** in **`csv.reader()`**.

- gh-113225 [https://github.com/python/cpython/issues/113225]: Speed up **`pathlib.Path.walk()`** by using **`os.DirEntry.path`** where possible.

- gh-89039 [https://github.com/python/cpython/issues/89039]: When replace() method is called on a subclass of datetime, date or time, properly call derived constructor. Previously, only the base class’s constructor was called.

Also, make sure to pass non-zero fold values when creating subclasses in various methods. Previously, fold was silently ignored.

- gh-112919 [https://github.com/python/cpython/issues/112919]: Speed-up **`datetime.datetime.replace()`**, **`datetime.date.replace()`** and **`datetime.time.replace()`**.

- gh-59013 [https://github.com/python/cpython/issues/59013]: Set breakpoint on the first executable line of the function, instead of the line of function definition when the user do `break func` using **`pdb`**

- gh-112343 [https://github.com/python/cpython/issues/112343]: Improve handling of pdb convenience variables to avoid replacing string contents.

- gh-112240 [https://github.com/python/cpython/issues/112240]: Add option to calendar module CLI to specify the weekday to start each week. Patch by Steven Ward.

- gh-111741 [https://github.com/python/cpython/issues/111741]: Recognise `image/webp` as a standard format in the **`mimetypes`** module.

- gh-43457 [https://github.com/python/cpython/issues/43457]: Fix the **`tkinter`** widget method **`wm_attributes()`**. It now accepts the attribute name without the minus prefix to get window attributes and allows to specify attributes and values to set as keyword arguments. Add new optional keyword argument *return_python_dict*: calling `w.wm_attributes(return_python_dict=True)` returns the attributes as a dict instead of a tuple. Calling `w.wm_attributes()` now returns a tuple instead of string if *wantobjects* was set to 0.

- gh-82626 [https://github.com/python/cpython/issues/82626]: Many functions now emit a warning if a boolean value is passed as a file descriptor argument.

- gh-111051 [https://github.com/python/cpython/issues/111051]: Added check for file modification during debugging with **`pdb`**

- gh-110345 [https://github.com/python/cpython/issues/110345]: Show the Tcl/Tk patchlevel (rather than version) in **`tkinter._test()`**.

- gh-38807 [https://github.com/python/cpython/issues/38807]: Fix race condition in **`trace`**. Instead of checking if a directory exists and creating it, directly call **`os.makedirs()`** with the kwarg `exist_ok=True`.

- gh-75705 [https://github.com/python/cpython/issues/75705]: Set unixfrom envelope in **`mailbox.mbox`** and **`mailbox.MMDF`**.

- gh-106233 [https://github.com/python/cpython/issues/106233]: Fix stacklevel in `InvalidTZPathWarning` during **`zoneinfo`** module import.

- gh-105102 [https://github.com/python/cpython/issues/105102]: Allow **`ctypes.Union`** to be nested in **`ctypes.Structure`** when the system endianness is the opposite of the classes.

- gh-104282 [https://github.com/python/cpython/issues/104282]: Fix null pointer dereference in **`lzma._decode_filter_properties()`** due to improper handling of BCJ filters with properties of zero length. Patch by Radislav Chugunov.

- gh-96471 [https://github.com/python/cpython/issues/96471]: Add **`queue.Queue`** termination with **`shutdown()`**.

- gh-101599 [https://github.com/python/cpython/issues/101599]: Changed argparse flag options formatting to remove redundancy.

- gh-85984 [https://github.com/python/cpython/issues/85984]: Add POSIX pseudo-terminal functions **`os.posix_openpt()`**, **`os.grantpt()`**, **`os.unlockpt()`**, and **`os.ptsname()`**.

- gh-102512 [https://github.com/python/cpython/issues/102512]: When **`os.fork()`** is called from a foreign thread (aka `_DummyThread`), the type of the thread in a child process is changed to `_MainThread`. Also changed its name and daemonic status, it can be now joined.

- gh-88569 [https://github.com/python/cpython/issues/88569]: Add **`os.path.isreserved()`**, which identifies reserved pathnames such as “NUL”, “AUX” and “CON”. This function is only available on Windows.

  Deprecate **`pathlib.PurePath.is_reserved()`**.

- bpo-38364 [https://bugs.python.org/issue?@action=redirect&bpo=38364]: The `inspect` functions `isgeneratorfunction`, `iscoroutinefunction`, `isasyncgenfunction` now support `functools.partialmethod` wrapped functions the same way they support `functools.partial`.

Documentation

- gh-115233 [https://github.com/python/cpython/issues/115233]: Fix an example for **`LoggerAdapter`** in the Logging Cookbook.

- gh-114123 [https://github.com/python/cpython/issues/114123]: Move the **`csv`** module docstring to the **`csv`** module instead of reexporting it from the internal **`_csv`** module, and remove `__doc__` from `csv.__all__`.

Move **`csv.__version__`** to the **`csv`** module instead of reexporting it from the internal **`_csv`** module, and remove `__version__` from `csv.__all__`.

Tests

- gh-114099 [https://github.com/python/cpython/issues/114099]: Added test exclusions required to run the test suite on iOS.
- gh-105089 [https://github.com/python/cpython/issues/105089]: Fix `test.test_zipfile.test_core.TestWithDirectory.test_create_directory_with_write` test in AIX by doing a bitwise AND of 0xFFFF on mode , so that it will be in sync with `zinfo.external_attr`

Build

- gh-115167 [https://github.com/python/cpython/issues/115167]: Avoid vendoring `vcruntime140_threads.dll` when building with Visual Studio 2022 version 17.8.
- gh-113632 [https://github.com/python/cpython/issues/113632]: Promote WASI to a tier 2 platform and drop Emscripten from tier 3 in configure.ac.
- gh-114099 [https://github.com/python/cpython/issues/114099]: configure and Makefile were refactored to accommodate framework builds on Apple platforms other than macOS.
- gh-114875 [https://github.com/python/cpython/issues/114875]: Add **`getgrent()`** as a prerequisite for building the **`grp`** module.

Windows

- gh-115049 [https://github.com/python/cpython/issues/115049]: Fixes `py.exe` launcher failing when run as users without user profiles.
- gh-115009 [https://github.com/python/cpython/issues/115009]: Update Windows installer to use SQLite 3.45.1.

- gh-109991 [https://github.com/python/cpython/issues/109991]: Update Windows build to use OpenSSL 3.0.13.
- gh-111239 [https://github.com/python/cpython/issues/111239]: Update Windows builds to use zlib v1.3.1.
- gh-100107 [https://github.com/python/cpython/issues/100107]: The `py.exe` launcher will no longer attempt to run the Microsoft Store redirector when launching a script containing a `/usr/bin/env` shebang
- gh-112984 [https://github.com/python/cpython/issues/112984]: Adds free-threaded binaries to Windows installer as an optional component.
- gh-89240 [https://github.com/python/cpython/issues/89240]: Allows **`multiprocessing`** to create pools of greater than 62 processes.

macOS

- gh-115009 [https://github.com/python/cpython/issues/115009]: Update macOS installer to use SQLite 3.45.1.
- gh-109991 [https://github.com/python/cpython/issues/109991]: Update macOS installer to use OpenSSL 3.0.13.
- gh-114490 [https://github.com/python/cpython/issues/114490]: Add Mach-O linkage support for **`platform.architecture()`**.
- gh-87804 [https://github.com/python/cpython/issues/87804]: On macOS the result of `os.statvfs` and `os.fstatvfs` now correctly report the size of very large disks, in previous versions the reported number of blocks was wrong for disks with at least 2**32 blocks.

IDLE

- gh-96905 [https://github.com/python/cpython/issues/96905]: In idlelib code, stop redefining built-ins ‘dict’ and ‘object’.
- gh-103820 [https://github.com/python/cpython/issues/103820]: Revise IDLE bindings so that events from mouse button

4/5 on non-X11 windowing systems (i.e. Win32 and Aqua) are not mistaken for scrolling.

Tools/Demos

- gh-113516 [https://github.com/python/cpython/issues/113516]: Don’t set `LDSHARED` when building for WASI.
- gh-109991 [https://github.com/python/cpython/issues/109991]: Update GitHub CI workflows to use OpenSSL 3.0.13 and multissltests to use 1.1.1w, 3.0.13, 3.1.5, and 3.2.1.
- gh-115015 [https://github.com/python/cpython/issues/115015]: Fix a bug in Argument Clinic that generated incorrect code for methods with no parameters that use the METH_METHOD | METH_FASTCALL | METH_KEYWORDS calling convention. Only the positional parameter count was checked; any keyword argument passed would be silently accepted.

C API

- gh-111140 [https://github.com/python/cpython/issues/111140]: Adds **`PyLong_AsNativeBytes()`**, **`PyLong_FromNativeBytes()`** and **`PyLong_FromUnsignedNativeBytes()`** functions.

- gh-114685 [https://github.com/python/cpython/issues/114685]: **`PyBuffer_FillInfo()`** now raises a **`SystemError`** if called with **`PyBUF_READ`** or **`PyBUF_WRITE`** as flags. These flags should only be used with the `PyMemoryView_*` C API.

- gh-114685 [https://github.com/python/cpython/issues/114685]: **`PyObject_GetBuffer()`** now raises a **`SystemError`** if called with **`PyBUF_READ`** or **`PyBUF_WRITE`** as flags. These flags should only be used with the `PyMemoryView_*` C API.

- gh-114626 [https://github.com/python/cpython/issues/114626]: Add `PyCFunctionFast` and `PyCFunctionFastWithKeywords` typedefs (identical to the existing `_PyCFunctionFast` and `_PyCFunctionFastWithKeywords` typedefs, just without a leading _ prefix).

- gh-114329 [https://github.com/python/cpython/issues/114329]: Add **`PyList_GetItemRef()`**, which is similar to **`PyList_GetItem()`** but returns a strong reference instead of a borrowed reference.
- gh-110850 [https://github.com/python/cpython/issues/110850]: Add PyTime C API:
  - **`PyTime_t`** type.
  - **`PyTime_MIN`** and **`PyTime_MAX`** constants.
  - **`PyTime_AsSecondsDouble()`**, **`PyTime_Monotonic()`**, **`PyTime_PerfCounter()`**, and **`PyTime_Time()`** functions.

  Patch by Victor Stinner.
- gh-112066 [https://github.com/python/cpython/issues/112066]: Add **`PyDict_SetDefaultRef()`**: insert a key and value into a dictionary if the key is not already present. This is similar to **`dict.setdefault()`**, but returns an integer value indicating if the key was already present. It is also similar to **`PyDict_SetDefault()`**, but returns a strong reference instead of a borrowed reference.

Python 3.13.0 alpha 3

Release date: 2024-01-17

Security

- gh-113659 [https://github.com/python/cpython/issues/113659]: Skip `.pth` files with names starting with a dot or hidden file attribute.

- gh-112302 [https://github.com/python/cpython/issues/112302]: Created a Software Bill-of-Materials document and tooling for tracking dependencies.

Core and Builtins

- gh-107901 [https://github.com/python/cpython/issues/107901]: Compiler duplicates basic blocks that have an eval breaker check, no line number, and multiple predecessors.
- gh-107901 [https://github.com/python/cpython/issues/107901]: A jump leaving an exception handler back to normal code no longer checks the eval breaker.
- gh-113655 [https://github.com/python/cpython/issues/113655]: Set the C recursion limit to 4000 on Windows, and 10000 on Linux/OSX. This seems to be near the sweet spot to maintain safety, but not compromise backwards compatibility.
- gh-113710 [https://github.com/python/cpython/issues/113710]: Add typed stack effects to the interpreter DSL, along with various instruction annotations.
- gh-77046 [https://github.com/python/cpython/issues/77046]: On Windows, file descriptors wrapping Windows handles are now created non inheritable by default (**PEP 446** [https://peps.python.org/pep-0446/]). Patch by Zackery Spytz and Victor Stinner.
- gh-113853 [https://github.com/python/cpython/issues/113853]: Guarantee that all executors make progress. This then guarantees that tier 2 execution always makes progress.
- gh-113753 [https://github.com/python/cpython/issues/113753]: Fix an issue where the finalizer of `PyAsyncGenASend` objects

might not be called if they were allocated from a free list.

- gh-107901 [https://github.com/python/cpython/issues/107901]: Compiler changed so that synthetic jumps which are not at loop end no longer check the eval breaker.
- gh-113703 [https://github.com/python/cpython/issues/113703]: Fix a regression in the **`codeop`** module that was causing it to incorrectly identify incomplete f-strings. Patch by Pablo Galindo
- gh-89811 [https://github.com/python/cpython/issues/89811]: Check for a valid `tp_version_tag` before performing bytecode specializations that rely on this value being usable.
- gh-111488 [https://github.com/python/cpython/issues/111488]: Changed error message in case of no 'in' keyword after 'for' in list comprehensions
- gh-113657 [https://github.com/python/cpython/issues/113657]: Fix an issue that caused important instruction pointer updates to be optimized out of tier two traces.
- gh-113603 [https://github.com/python/cpython/issues/113603]: Fixed bug where a redundant NOP is not removed, causing an assertion to fail in the compiler in debug mode.
- gh-113602 [https://github.com/python/cpython/issues/113602]: Fix an error that was causing the parser to try to overwrite existing errors and crashing in the process. Patch by Pablo Galindo
- gh-113486 [https://github.com/python/cpython/issues/113486]: No longer issue spurious `PY_UNWIND` events for optimized calls to classes.

- gh-113297 [https://github.com/python/cpython/issues/113297]: Fix segfault in the compiler on with statement with 19 context managers.

- gh-113212 [https://github.com/python/cpython/issues/113212]: Improve **`super`** error messages.

- gh-111375 [https://github.com/python/cpython/issues/111375]: Only use `NULL` in the exception stack to indicate an exception was handled. Patch by Carey Metcalfe.

- gh-112215 [https://github.com/python/cpython/issues/112215]: Increase the C recursion limit by a factor of 3 for non-debug builds, except for webassembly and s390 platforms which are unchanged. This mitigates some regressions in 3.12 with deep recursion mixing builtin (C) and Python code.

- gh-113054 [https://github.com/python/cpython/issues/113054]: Fixed bug where a redundant NOP is not removed, causing an assertion to fail in the compiler in debug mode.

- gh-106905 [https://github.com/python/cpython/issues/106905]: Use per AST-parser state rather than global state to track recursion depth within the AST parser to prevent potential race condition due to simultaneous parsing.

  The issue primarily showed up in 3.11 by multithreaded users of **`ast.parse()`**. In 3.12 a change to when garbage collection can be triggered prevented the race condition from occurring.

- gh-108866 [https://github.com/python/cpython/issues/108866]: Change the API and contract of `_PyExecutorObject` to return the next_instr pointer, instead of the frame, and to always execute at least one instruction.

- gh-90350 [https://github.com/python/cpython/issues/90350]: Optimize builtin functions **`min()`** and **`max()`**.

- gh-112943 [https://github.com/python/cpython/issues/112943]: Correctly compute end column offsets for multiline tokens in the **`tokenize`** module. Patch by Pablo Galindo

- gh-112125 [https://github.com/python/cpython/issues/112125]: Fix `None.__ne__(None)` returning `NotImplemented` instead of `False`.

- gh-74616 [https://github.com/python/cpython/issues/74616]: **`input()`** now raises a ValueError when output on the terminal if the prompt contains embedded null characters instead of silently truncating it.

- gh-112716 [https://github.com/python/cpython/issues/112716]: Fix SystemError in the `import` statement and in `__reduce__()` methods of builtin types when `__builtins__` is not a dict.

- gh-112730 [https://github.com/python/cpython/issues/112730]: Use color to highlight error locations in tracebacks. Patch by Pablo Galindo

- gh-112625 [https://github.com/python/cpython/issues/112625]: Fixes a bug where a bytearray object could be cleared while iterating over an argument in the `bytearray.join()` method that could result in reading memory after it was freed.

- gh-112660 [https://github.com/python/cpython/issues/112660]: Do not clear unexpected errors during formatting error messages for ImportError and AttributeError for modules.

- gh-105967 [https://github.com/python/cpython/issues/105967]: Workaround a bug in Apple’s macOS platform zlib library where **`zlib.crc32()`** and **`binascii.crc32()`** could produce incorrect results on multi-gigabyte inputs. Including when using `zipfile` on zips containing large data.

- gh-95754 [https://github.com/python/cpython/issues/95754]: Provide a better error message when accessing invalid attributes on partially initialized modules. The origin of the module being accessed is now included in the message to help with the common issue of shadowing other modules.

- gh-112217 [https://github.com/python/cpython/issues/112217]: Add check for the type of `__cause__` returned from calling the type `T` in `raise from T`.

- gh-111058 [https://github.com/python/cpython/issues/111058]: Change coro.cr_frame/gen.gi_frame to return `None` after the coroutine/generator has been closed. This fixes a bug where `getcoroutinestate()` and `getgeneratorstate()` return the wrong state for a closed coroutine/generator.

- gh-112388 [https://github.com/python/cpython/issues/112388]: Fix an error that was causing the parser to try to overwrite tokenizer errors. Patch by pablo Galindo

- gh-112387 [https://github.com/python/cpython/issues/112387]: Fix error positions for decoded strings with backwards tokenize errors. Patch by Pablo Galindo

- gh-99606 [https://github.com/python/cpython/issues/99606]: Make code generated for an empty f-string identical to the code of an empty normal string.

- gh-112367 [https://github.com/python/cpython/issues/112367]: Avoid undefined behaviour when using the perf trampolines by not freeing the code arenas until shutdown. Patch by Pablo Galindo

- gh-112320 [https://github.com/python/cpython/issues/112320]: The Tier 2 translator now tracks the confidence level for staying “on trace” (i.e. not exiting back to the Tier 1 interpreter) for branch instructions based on the number of bits set in the branch “counter”. Trace translation ends when the confidence drops below 1/3rd.

- gh-109598 [https://github.com/python/cpython/issues/109598]: `PyComplex_RealAsDouble()`/`PyComplex_ImagAsDouble()` now tries to convert an object to a `complex` instance using its `__complex__()` method before falling back to the `__float__()` method. Patch by Sergey B Kirpichev.

- gh-94606 [https://github.com/python/cpython/issues/94606]: Fix UnicodeEncodeError when **`email.message.get_payload()`** reads a message with a Unicode surrogate character and the message content is not well-formed for surrogateescape encoding. Patch by Sidney Markowitz.

- bpo-21861 [https://bugs.python.org/issue?@action=redirect&bpo=21861]: Use the object’s actual class name in **`_io.FileIO.__repr__()`**, **`_io._WindowsConsoleIO()`** and **`_io.TextIOWrapper.__repr__()`**, to make these methods subclass friendly.

- bpo-45369 [https://bugs.python.org/issue?@action=redirect&bpo=45369]: Remove LibreSSL workarounds as per **PEP 644** [https://peps.python.org/pep-0644/].

- bpo-34392 [https://bugs.python.org/issue?@action=redirect&bpo=34392]: Added `sys._is_interned()`.

Library

- gh-114077 [https://github.com/python/cpython/issues/114077]: Fix possible `OverflowError` in `socket.socket.sendfile()` when pass *count* larger than 2 GiB on 32-bit platform.

- gh-111803 [https://github.com/python/cpython/issues/111803]: **`plistlib`** now supports loading more deeply nested lists in binary format.

- gh-114014 [https://github.com/python/cpython/issues/114014]: Fixed a bug in **`fractions.Fraction`** where an invalid string using `d` in the decimals part creates a different error compared to other invalid letters/characters. Patch by Jeremiah Gabriel Pascual.

- gh-108364 [https://github.com/python/cpython/issues/108364]: **`sqlite3.Connection.iterdump()`** now ensures that foreign key support is disabled before dumping the database schema, if there is any foreign key violation. Patch by Erlend E. Aasland and Mariusz Felisiak.

- gh-113971 [https://github.com/python/cpython/issues/113971]: The **`zipfile.ZipInfo`** previously protected `._compresslevel` attribute has been made public as `.compress_level` with the old `_compresslevel` name remaining available as a property to retain compatibility.

- gh-113877 [https://github.com/python/cpython/issues/113877]: Fix **`tkinter`** method `winfo_pathname()` on 64-bit Windows.

- gh-113868 [https://github.com/python/cpython/issues/113868]: Added **`mmap.MAP_NORESERVE`**, **`mmap.MAP_NOEXTEND`**, **`mmap.MAP_HASSEMAPHORE`**, **`mmap.MAP_NOCACHE`**, **`mmap.MAP_JIT`**, **`mmap.MAP_RESILIENT_CODESIGN`**, **`mmap.MAP_RESILIENT_MEDIA`**, **`mmap.MAP_32BIT`**, **`mmap.MAP_TRANSLATED_ALLOW_EXECUTE`**, **`mmap.MAP_UNIX03`** and **`mmap.MAP_TPRO`**. All of them are `mmap(2)` flags on macOS.

- gh-113848 [https://github.com/python/cpython/issues/113848]: **`asyncio.TaskGroup()`** and **`asyncio.timeout()`** context managers now handle **`CancelledError`** subclasses as well as exact **`CancelledError`**.

- gh-113661 [https://github.com/python/cpython/issues/113661]: unittest runner: Don’t exit 5 if tests were skipped. The intention of exiting 5 was to detect issues where the test suite wasn’t discovered at all. If we skipped tests, it was correctly discovered.

- gh-96037 [https://github.com/python/cpython/issues/96037]: Insert **`TimeoutError`** in the context of the exception that was raised during exiting an expired **`asyncio.timeout()`** block.

- gh-113781 [https://github.com/python/cpython/issues/113781]: Silence unraisable AttributeError when warnings are emitted during Python finalization.

- gh-113238 [https://github.com/python/cpython/issues/113238]: Add `Anchor` to `importlib.resources` (in order for the code to comply with the documentation)

- gh-111693 [https://github.com/python/cpython/issues/111693]: **`asyncio.Condition.wait()`** now re-raises the same **`CancelledError`** instance that may have caused it to be interrupted. Fixed race condition in **`asyncio.Semaphore.acquire()`** when interrupted with a **`CancelledError`**.

- gh-113791 [https://github.com/python/cpython/issues/113791]: Add `CLOCK_MONOTONIC_RAW_APPROX` and `CLOCK_UPTIME_RAW_APPROX` to **`time`** on macOS. These are clocks available on macOS 10.12 or later.

- gh-112932 [https://github.com/python/cpython/issues/112932]: Restore the ability for **`zipfile`** to `extractall` from zip files with a “/” directory entry in them as is commonly added to zips by some wiki or bug tracker data exporters.

- gh-113568 [https://github.com/python/cpython/issues/113568]: Raise deprecation warnings from **`pathlib.PurePath`** and not its private base class `PurePathBase`.

- gh-113594 [https://github.com/python/cpython/issues/113594]: Fix **`UnicodeEncodeError`** in **`email`** when re-fold lines that contain unknown-8bit encoded part followed by non-unknown-8bit encoded part.

- gh-113538 [https://github.com/python/cpython/issues/113538]: In **`asyncio.StreamReaderProtocol.connection_made()`**, there is callback that logs an error if the task wrapping the “connected callback” fails. This callback would itself fail if the task was cancelled. Prevent this by checking whether the task was cancelled first. If so, close the transport but don’t log an error.

- gh-113626 [https://github.com/python/cpython/issues/113626]: Add support for the *allow_code* argument in the **`marshal`** module. Passing `allow_code=False` prevents serialization and de-serialization of code objects which is incompatible between Python versions.

- gh-85567 [https://github.com/python/cpython/issues/85567]: Fix resource warnings for unclosed files in **`pickle`** and **`pickletools`** command line interfaces.

- gh-113537 [https://github.com/python/cpython/issues/113537]: Support loads `str` in **`plistlib.loads()`**.

- gh-89850 [https://github.com/python/cpython/issues/89850]: Add default implementations of **`pickle.Pickler.persistent_id()`** and **`pickle.Unpickler.persistent_load()`** methods in the C implementation. Calling `super().persistent_id()` and `super().persistent_load()` in subclasses of the C implementation of **`pickle.Pickler`** and **`pickle.Unpickler`** classes no longer causes infinite recursion.

- gh-113569 [https://github.com/python/cpython/issues/113569]: Indicate if there were no actual calls in unittest `assert_has_calls()` failure.

- gh-101225 [https://github.com/python/cpython/issues/101225]: Increase the backlog for `multiprocessing.connection.Listener` objects created by **`multiprocessing.manager`** and **`multiprocessing.resource_sharer`** to significantly reduce the risk of getting a connection refused error when creating a `multiprocessing.connection.Connection` to them.

- gh-113568 [https://github.com/python/cpython/issues/113568]: Raise audit events from `pathlib.Path` and not its private base class `PathBase`.

- gh-113543 [https://github.com/python/cpython/issues/113543]: Make sure that `webbrowser.MacOSXOSAScript` sends `webbrowser.open` audit event.

- gh-113028 [https://github.com/python/cpython/issues/113028]: When a second reference to a string appears in the input to `pickle`, and the Python implementation is in use, we are guaranteed that a single copy gets pickled and a single object is shared when reloaded. Previously, in protocol 0, when a string contained certain characters (e.g. newline) it resulted in duplicate objects.

- gh-113421 [https://github.com/python/cpython/issues/113421]: Fix multiprocessing logger for `%(filename)s`.

- gh-111784 [https://github.com/python/cpython/issues/111784]: Fix segfaults in the `_elementtree` module. Fix first segfault during deallocation of `_elementtree.XMLParser` instances by keeping strong reference to `pyexpat` module in module state for capsule lifetime. Fix second segfault which happens in the same deallocation process by keeping strong reference to `_elementtree` module in `XMLParser` structure for `_elementtree` module lifetime.

- gh-113407 [https://github.com/python/cpython/issues/113407]: Fix import of **`unittest.mock`** when CPython is built without docstrings.

- gh-113320 [https://github.com/python/cpython/issues/113320]: Fix regression in Python 3.12 where **`Protocol`** classes that were not marked as **`runtime-checkable`** would be unnecessarily introspected, potentially causing exceptions to be raised if the protocol had problematic members. Patch by Alex Waygood.

- gh-53502 [https://github.com/python/cpython/issues/53502]: Add a new option `aware_datetime` in **`plistlib`** to loads or dumps aware datetime.

- gh-113358 [https://github.com/python/cpython/issues/113358]: Fix rendering tracebacks with exceptions with a broken __getattr__

- gh-113214 [https://github.com/python/cpython/issues/113214]: Fix an `AttributeError` during asyncio SSL protocol aborts in SSL-over-SSL scenarios.

- gh-113246 [https://github.com/python/cpython/issues/113246]: Update bundled pip to 23.3.2.

- gh-87264 [https://github.com/python/cpython/issues/87264]: Fixed tarfile list() method to show file type.

- gh-112182 [https://github.com/python/cpython/issues/112182]: **`asyncio.futures.Future.set_exception()`** now transforms **`StopIteration`** into **`RuntimeError`** instead of hanging or other misbehavior. Patch contributed by Jamie Phan.

- gh-113225 [https://github.com/python/cpython/issues/113225]: Speed up **`pathlib.Path.glob()`** by using **`os.DirEntry.path`** where possible.

- gh-113149 [https://github.com/python/cpython/issues/113149]: Improve error message when a JSON array or object contains a trailing comma. Patch by Carson Radtke.

- gh-113117 [https://github.com/python/cpython/issues/113117]: The **`subprocess`** module can now use the **`os.posix_spawn()`** function with `close_fds=True` on platforms where `posix_spawn_file_actions_addclosefrom_np` is available. Patch by Jakub Kulik.

- gh-113199 [https://github.com/python/cpython/issues/113199]: Make `http.client.HTTPResponse.read1` and `http.client.HTTPResponse.readline` close IO after reading all data when content length is known. Patch by Illia Volochii.

- gh-113191 [https://github.com/python/cpython/issues/113191]: Add support of **`os.fchmod()`** and a file descriptor in **`os.chmod()`** on Windows.

- gh-113188 [https://github.com/python/cpython/issues/113188]: Fix **`shutil.copymode()`** and **`shutil.copystat()`** on Windows. Previously they worked differently if *dst* is a symbolic link: they modified the permission bits of *dst* itself rather than the file it points to if *follow_symlinks* is true or *src* is not a symbolic link, and did not modify the permission bits if *follow_symlinks* is false and *src* is a symbolic link.

- gh-113119 [https://github.com/python/cpython/issues/113119]: **`os.posix_spawn()`** now accepts `env=None`, which makes the newly spawned process use the current process environment. Patch by Jakub Kulik.

- gh-113202 [https://github.com/python/cpython/issues/113202]: Add a `strict` option to `batched()` in the `itertools` module.

- gh-61648 [https://github.com/python/cpython/issues/61648]: Detect line numbers of properties in doctests.

- gh-113175 [https://github.com/python/cpython/issues/113175]: Sync with importlib_metadata 7.0, including improved type annotations, fixed issue with symlinked packages in `package_distributions`, added `EntryPoints.__repr__`, introduced the `diagnose` script, added `Distribution.origin` property, and removed deprecated `EntryPoint` access by numeric index (tuple behavior).

- gh-59616 [https://github.com/python/cpython/issues/59616]: Add support of **`os.lchmod()`** and the *follow_symlinks* argument in **`os.chmod()`** on Windows. Note that the default value of *follow_symlinks* in **`os.lchmod()`** is `False` on Windows.

- gh-112559 [https://github.com/python/cpython/issues/112559]: **`signal.signal()`** and **`signal.getsignal()`** no longer call `repr` on callable handlers. **`asyncio.run()`** and **`asyncio.Runner.run()`** no longer call `repr` on the task results. Patch by Yilei Yang.

- gh-112962 [https://github.com/python/cpython/issues/112962]: **`dis`** module functions add cache information to the **`Instruction`** instance rather than creating fake **`Instruction`** instances to represent the cache entries.

- gh-112989 [https://github.com/python/cpython/issues/112989]: Reduce overhead to connect sockets with **`asyncio`** SelectorEventLoop.

- gh-112970 [https://github.com/python/cpython/issues/112970]: Use **`closefrom()`** on Linux where available (e.g. glibc-2.34), rather than only FreeBSD.

- gh-110190 [https://github.com/python/cpython/issues/110190]: Fix ctypes structs with array on PPC64LE platform by setting `MAX_STRUCT_SIZE` to 64 in stgdict. Patch by Diego Russo.

- gh-112540 [https://github.com/python/cpython/issues/112540]: The statistics.geometric_mean() function now returns zero for datasets containing a zero. Formerly, it would raise an exception.

- gh-87286 [https://github.com/python/cpython/issues/87286]: Added **`LOG_FTP`**, **`LOG_NETINFO`**, **`LOG_REMOTEAUTH`**, **`LOG_INSTALL`**, **`LOG_RAS`**, and **`LOG_LAUNCHD`** tot the **`syslog`** module, all of them constants on used on macOS.

- gh-112800 [https://github.com/python/cpython/issues/112800]: Fix **`asyncio`** `SubprocessTransport.close()` not to throw `PermissionError` when used with setuid executables.

- gh-51944 [https://github.com/python/cpython/issues/51944]: Add the following constants to the **`termios`** module. These values are present in macOS system headers: `ALTWERASE`, `B14400`, `B28800`, `B7200`, `B76800`, `CCAR_OFLOW`, `CCTS_OFLOW`, `CDSR_OFLOW`, `CDTR_IFLOW`, `CIGNORE`, `CRTS_IFLOW`, `EXTPROC`, `IUTF8`, `MDMBUF`, `NL2`, `NL3`, `NOKERNINFO`, `ONOEOT`, `OXTABS`, `VDSUSP`, `VSTATUS`.

- gh-79325 [https://github.com/python/cpython/issues/79325]: Fix an infinite recursion error in **`tempfile.TemporaryDirectory()`** cleanup on Windows.

- gh-94692 [https://github.com/python/cpython/issues/94692]: **`shutil.rmtree()`** now only catches OSError exceptions. Previously a symlink attack resistant version of `shutil.rmtree()` could ignore or pass to the error handler arbitrary exception when invalid arguments were provided.

- gh-112736 [https://github.com/python/cpython/issues/112736]: The use of del-safe symbols in `subprocess` was refactored to allow for use in cross-platform build environments.

- gh-112727 [https://github.com/python/cpython/issues/112727]: Speed up `pathlib.Path.absolute()`. Patch by Barney Gale.

- gh-74690 [https://github.com/python/cpython/issues/74690]: Speedup `issubclass()` checks against simple `runtime-checkable protocols` by around 6%. Patch by Alex Waygood.

- gh-74690 [https://github.com/python/cpython/issues/74690]: Speedup `isinstance()` checks by roughly 20% for `runtime-checkable protocols` that only have one callable member. Speedup `issubclass()` checks for these protocols by roughly 10%. Patch by Alex Waygood.

- gh-112645 [https://github.com/python/cpython/issues/112645]: Remove deprecation error on passing `onerror` to `shutil.rmtree()`.

- gh-112640 [https://github.com/python/cpython/issues/112640]: Add `kwdefaults` parameter to `types.FunctionType` to set default keyword argument values.

- gh-112622 [https://github.com/python/cpython/issues/112622]: Ensure `name` parameter is passed to event loop in `asyncio.create_task()`.

- gh-112618 [https://github.com/python/cpython/issues/112618]: Fix a caching bug relating to `typing.Annotated`. `Annotated[str, True]` is no longer identical to `Annotated[str, 1]`.

- gh-112334 [https://github.com/python/cpython/issues/112334]: Fixed a performance regression in 3.12’s `subprocess` on Linux where it would no longer use the fast-path `vfork()`

system call when it could have due to a logic bug, instead falling back to the safe but slower `fork()`.

Also fixed a second 3.12.0 potential security bug. If a value of `extra_groups=[]` was passed to **`subprocess.Popen`** or related APIs, the underlying `setgroups(0, NULL)` system call to clear the groups list would not be made in the child process prior to `exec()`.

This was identified via code inspection in the process of fixing the first bug.

- gh-110190 [https://github.com/python/cpython/issues/110190]: Fix ctypes structs with array on Arm platform by setting `MAX_STRUCT_SIZE` to 32 in stgdict. Patch by Diego Russo.
- gh-81194 [https://github.com/python/cpython/issues/81194]: Fix a crash in **`socket.if_indextoname()`** with specific value (UINT_MAX). Fix an integer overflow in **`socket.if_indextoname()`** on 64-bit non-Windows platforms.
- gh-112578 [https://github.com/python/cpython/issues/112578]: Fix a spurious **`RuntimeWarning`** when executing the **`zipfile`** module.
- gh-112516 [https://github.com/python/cpython/issues/112516]: Update the bundled copy of pip to version 23.3.1.
- gh-112510 [https://github.com/python/cpython/issues/112510]: Add **`readline.backend`** for the backend readline uses (`editline` or `readline`)
- gh-112328 [https://github.com/python/cpython/issues/112328]: [Enum] Make `EnumDict`, `EnumDict.member_names`, `EnumType._add_alias_` and `EnumType._add_value_alias_` public.

- gh-112509 [https://github.com/python/cpython/issues/112509]: Fix edge cases that could cause a key to be present in both the `__required_keys__` and `__optional_keys__` attributes of a **`typing.TypedDict`**. Patch by Jelle Zijlstra.

- gh-101336 [https://github.com/python/cpython/issues/101336]: Add `keep_alive` keyword parameter for **`AbstractEventLoop.create_server()`** and **`BaseEventLoop.create_server()`**.

- gh-63284 [https://github.com/python/cpython/issues/63284]: Added support for TLS-PSK (pre-shared key) mode to the **`ssl`** module.

- gh-112414 [https://github.com/python/cpython/issues/112414]: Fix regression in Python 3.12 where calling **`repr()`** on a module that had been imported using a custom loader could fail with **`AttributeError`**. Patch by Alex Waygood.

- gh-112358 [https://github.com/python/cpython/issues/112358]: Revert change to **`struct.Struct`** initialization that broke some cases of subclassing.

- gh-112405 [https://github.com/python/cpython/issues/112405]: Optimize **`pathlib.PurePath.relative_to()`**. Patch by Alex Waygood.

- gh-94722 [https://github.com/python/cpython/issues/94722]: Fix bug where comparison between instances of **`DocTest`** fails if one of them has `None` as its lineno.

- gh-112361 [https://github.com/python/cpython/issues/112361]: Speed up a small handful of **`pathlib`** methods by removing some temporary objects.

- gh-112345 [https://github.com/python/cpython/issues/112345]: Improve error message when trying to call **`issubclass()`**

against a **`typing.Protocol`** that has non-method members. Patch by Randolf Scholz.

- gh-112137 [https://github.com/python/cpython/issues/112137]: Change **`dis`** output to display no-lineno as "–" instead of "None".

- gh-112332 [https://github.com/python/cpython/issues/112332]: Deprecate the `exc_type` field of **`traceback.TracebackException`**. Add `exc_type_str` to replace it.

- gh-81620 [https://github.com/python/cpython/issues/81620]: Add extra tests for **`random.binomialvariate()`**

- gh-112292 [https://github.com/python/cpython/issues/112292]: Fix a crash in **`readline`** when imported from a sub interpreter. Patch by Anthony Shaw

- gh-77621 [https://github.com/python/cpython/issues/77621]: Slightly improve the import time of the **`pathlib`** module by deferring some imports. Patch by Barney Gale.

- gh-112137 [https://github.com/python/cpython/issues/112137]: Change **`dis`** output to display logical labels for jump targets instead of offsets.

- gh-112139 [https://github.com/python/cpython/issues/112139]: Add **`Signature.format()`** to format signatures to string with extra options. And use it in **`pydoc`** to render more readable signatures that have new lines between parameters.

- gh-112105 [https://github.com/python/cpython/issues/112105]: Make **`readline.set_completer_delims()`** work with libedit

- gh-106922 [https://github.com/python/cpython/issues/106922]: Display multiple lines with `traceback` when errors span multiple lines.

- gh-111874 [https://github.com/python/cpython/issues/111874]: When creating a **`typing.NamedTuple`** class, ensure **`__set_name__()`** is called on all objects that define `__set_name__` and exist in the values of the `NamedTuple` class’s class dictionary. Patch by Alex Waygood.

- gh-68166 [https://github.com/python/cpython/issues/68166]: Add support of the “vsapi” element type in **`tkinter.ttk.Style.element_create()`**.

- gh-110275 [https://github.com/python/cpython/issues/110275]: Named tuple’s methods `_replace()` and `__replace__()` now raise TypeError instead of ValueError for invalid keyword arguments.

- gh-99367 [https://github.com/python/cpython/issues/99367]: Do not mangle `sys.path[0]` in **`pdb`** if safe_path is set

- gh-111615 [https://github.com/python/cpython/issues/111615]: Fix a regression caused by a fix to gh-93162 [https://github.com/python/cpython/issues/93162] whereby you couldn’t configure a **`QueueHandler`** without specifying handlers.

- gh-75666 [https://github.com/python/cpython/issues/75666]: Fix the behavior of **`tkinter`** widget’s `unbind()` method with two arguments. Previously, `widget.unbind(sequence, funcid)` destroyed the current binding for *sequence*, leaving *sequence* unbound, and deleted the *funcid* command. Now it removes only *funcid* from the binding for *sequence*, keeping other commands, and deletes the *funcid* command. It leaves *sequence* unbound only if *funcid* was the last bound command.

- gh-67790 [https://github.com/python/cpython/issues/67790]: Implement basic formatting support (minimum width, alignment, fill) for **`fractions.Fraction`**.

- gh-111049 [https://github.com/python/cpython/issues/111049]: Fix crash during garbage collection of the **`io.BytesIO`** buffer object.

- gh-102980 [https://github.com/python/cpython/issues/102980]: Redirect the output of `interact` command of **`pdb`** to the same channel as the debugger. Add tests and improve docs.

- gh-102988 [https://github.com/python/cpython/issues/102988]: **`email.utils.getaddresses()`** and **`email.utils.parseaddr()`** now return `('', '')` 2-tuples in more situations where invalid email addresses are encountered instead of potentially inaccurate values. Add optional *strict* parameter to these two functions: use `strict=False` to get the old behavior, accept malformed inputs. `getattr(email.utils, 'supports_strict_parsing', False)` can be use to check if the *strict* parameter is available. Patch by Thomas Dwyer and Victor Stinner to improve the **CVE 2023-27043** [https://www.cve.org/CVERecord?id=CVE-2023-27043] fix.

- gh-52161 [https://github.com/python/cpython/issues/52161]: **`cmd.Cmd.do_help()`** now cleans docstrings with **`inspect.cleandoc()`** before writing them. Patch by Filip Łapkiewicz.

- gh-82300 [https://github.com/python/cpython/issues/82300]: Add `track` parameter to **`multiprocessing.shared_memory.SharedMemory`** that allows using shared memory blocks without having to register with the POSIX resource tracker that automatically releases them upon process exit.

- gh-110109 [https://github.com/python/cpython/issues/110109]: Add private `pathlib._PurePathBase` class: a base class for **`pathlib.PurePath`** that omits certain magic methods. It may be made public (along with `_PathBase`) in future.

- gh-109858 [https://github.com/python/cpython/issues/109858]: Protect **`zipfile`** from “quoted-overlap” zipbomb. It now raises BadZipFile when try to read an entry that overlaps with other entry or central directory.

- gh-109786 [https://github.com/python/cpython/issues/109786]: Fix possible reference leaks and crash when re-enter the `__next__()` method of **`itertools.pairwise`**.

- gh-91539 [https://github.com/python/cpython/issues/91539]: Small (10 - 20%) and trivial performance improvement of **`urllib.request.getproxies_environment()`**, typically useful when there are many environment variables to go over.

- gh-103363 [https://github.com/python/cpython/issues/103363]: Add *follow_symlinks* keyword-only argument to **`pathlib.Path.owner()`** and **`group()`**, defaulting to `True`.

- gh-102130 [https://github.com/python/cpython/issues/102130]: Support tab completion in **`cmd`** for `editline`.

- gh-99437 [https://github.com/python/cpython/issues/99437]: **`runpy.run_path()`** now decodes path-like objects, making sure __file__ and sys.argv[0] of the module being run are always strings.

- gh-104003 [https://github.com/python/cpython/issues/104003]: Add **`warnings.deprecated()`**, a decorator to mark deprecated functions to static type checkers and to warn on usage of deprecated classes and functions. See **PEP 702** [https://peps.python.org/pep-0702/]. Patch by Jelle Zijlstra.

- gh-103708 [https://github.com/python/cpython/issues/103708]: Make hardcoded python name, a configurable parameter so that different implementations of python can override it instead of making huge diffs in sysconfig.py

- gh-66515 [https://github.com/python/cpython/issues/66515]: **`mailbox.MH`** now supports folders that do not contain a `.mh_sequences` file (e.g. Claws Mail IMAP-cache folders). Patch by Serhiy Storchaka.

- gh-83162 [https://github.com/python/cpython/issues/83162]: Renamed **`re.error`** to **`PatternError`** for clarity, and kept **`re.error`** for backward compatibility. Patch by Matthias Bussonnier and Adam Chhina.

- gh-91133 [https://github.com/python/cpython/issues/91133]: Fix a bug in **`tempfile.TemporaryDirectory`** cleanup, which now no longer dereferences symlinks when working around file system permission errors.

- bpo-43153 [https://bugs.python.org/issue?@action=redirect&bpo=43153]: On Windows, `tempfile.TemporaryDirectory` previously masked a `PermissionError` with `NotADirectoryError` during directory cleanup. It now correctly raises `PermissionError` if errors are not ignored. Patch by Andrei Kulakov and Ken Jin.

- bpo-32731 [https://bugs.python.org/issue?@action=redirect&bpo=32731]: **`getpass.getuser()`** now raises **`OSError`** for all failures rather than **`ImportError`** on systems lacking the **`pwd`** module or **`KeyError`** if the password database is empty.

- bpo-34321 [https://bugs.python.org/issue?@action=redirect&bpo=34321]: **`mmap.mmap`** now has a *trackfd* parameter on Unix; if it is `False`, the file descriptor specified by *fileno* will not be duplicated.

- bpo-35332 [https://bugs.python.org/issue?@action=redirect&bpo=35332]: The `shutil.rmtree()` function now ignores errors when calling `os.close()` when *ignore_errors* is `True`, and `os.close()` no longer retried after error.

- bpo-35928 [https://bugs.python.org/issue?@action=redirect&bpo=35928]: `io.TextIOWrapper` now correctly handles the decoding buffer after `read()` and `write()`.

- bpo-26791 [https://bugs.python.org/issue?@action=redirect&bpo=26791]: `shutil.move()` now moves a symlink into a directory when that directory is the target of the symlink. This provides the same behavior as the mv shell command. The previous behavior raised an exception. Patch by Jeffrey Kintscher.

- bpo-41422 [https://bugs.python.org/issue?@action=redirect&bpo=41422]: Fixed memory leaks of `pickle.Pickler` and `pickle.Unpickler` involving cyclic references via the internal memo mapping.

- bpo-19821 [https://bugs.python.org/issue?@action=redirect&bpo=19821]: The **`pydoc.ispackage()`** function has been deprecated.

- bpo-40262 [https://bugs.python.org/issue?@action=redirect&bpo=40262]: The **`ssl.SSLSocket.recv_into()`** method no longer requires the *buffer* argument to implement `__len__` and supports buffers with arbitrary item size.

- bpo-39912 [https://bugs.python.org/issue?@action=redirect&bpo=39912]: `warnings.filterwarnings()` and `warnings.simplefilter()` now raise appropriate exceptions instead of `AssertionError`. Patch contributed by Rémi Lapeyre.

- bpo-37260 [https://bugs.python.org/issue?@action=redirect&bpo=37260]: Fixed a race condition in **`shutil.rmtree()`** in which directory entries removed by another process or thread while `shutil.rmtree()` is running can cause it to raise FileNotFoundError. Patch by Jeffrey Kintscher.

- bpo-36959 [https://bugs.python.org/issue?@action=redirect&bpo=36959]: Fix some error messages for invalid ISO format string combinations in `strptime()` that referred to directives not contained in the format string. Patch by Gordon P. Hemsley.

- bpo-18060 [https://bugs.python.org/issue?@action=redirect&bpo=18060]: Fixed a class inheritance issue that can cause segfaults when deriving two or more levels of subclasses from a base class of Structure or Union.

- bpo-29779 [https://bugs.python.org/issue?@action=redirect&bpo=29779]: Add a new **`PYTHON_HISTORY`** environment variable to set the location of a `.python_history` file.

- bpo-21360 [https://bugs.python.org/issue?@action=redirect&bpo=21360]: **`mailbox.Maildir`** now ignores files with a leading dot.

Documentation

- gh-111699 [https://github.com/python/cpython/issues/111699]: Relocate `smtpd` deprecation notice to its own section rather than under `locale` in What’s New in Python 3.12 document
- gh-110746 [https://github.com/python/cpython/issues/110746]: Improved markup for valid options/values for methods

ttk.treeview.column and ttk.treeview.heading, and for Layouts.

- gh-95649 [https://github.com/python/cpython/issues/95649]: Document that the **`asyncio`** module contains code taken from v0.16.0 of the uvloop project [https://github.com/MagicStack/uvloop/tree/v0.16.0], as well as the required MIT licensing information.

Tests

- gh-111798 [https://github.com/python/cpython/issues/111798]: Disable `test_super_deep()` from `test_call` under pydebug builds on WASI; the stack depth is too small to make the test useful.

- gh-111801 [https://github.com/python/cpython/issues/111801]: Lower the recursion limit in `test_isinstance` for `test_infinitely_many_bases()`. This prevents a stack overflow on a pydebug build of WASI.

- gh-111802 [https://github.com/python/cpython/issues/111802]: Specify a low recursion depth for `test_bad_getattr()` in `test.pickletester` to avoid exhausting the stack under a pydebug build for WASI.

- gh-44626 [https://github.com/python/cpython/issues/44626]: Fix **`os.path.isabs()`** incorrectly returning `True` when given a path that starts with exactly one (back)slash on Windows.

  Fix **`pathlib.PureWindowsPath.is_absolute()`** incorrectly returning `False` for some paths beginning with two (back)slashes.

- gh-113633 [https://github.com/python/cpython/issues/113633]: Use module state for the _testcapi extension module.

- gh-109980 [https://github.com/python/cpython/issues/109980]: Fix `test_tarfile_vs_tar` in `test_shutil` for macOS, where system tar can include more information in the archive than **`shutil.make_archive`**.

- gh-112769 [https://github.com/python/cpython/issues/112769]: The tests now correctly compare zlib version when **`zlib.ZLIB_RUNTIME_VERSION`** contains non-integer suffixes. For example zlib-ng defines the version as `1.3.0.zlib-ng`.

- gh-112334 [https://github.com/python/cpython/issues/112334]: Adds a regression test to verify that `vfork()` is used when expected by **`subprocess`** on vfork enabled POSIX systems (Linux).

- gh-108927 [https://github.com/python/cpython/issues/108927]: Fixed order dependence in running tests in the same process when a test that has submodules (e.g. test_importlib) follows a test that imports its submodule (e.g. test_importlib.util) and precedes a test (e.g. test_unittest or test_compileall) that uses that submodule.

- bpo-40648 [https://bugs.python.org/issue?@action=redirect&bpo=40648]: Test modes that file can get with chmod() on Windows.

Build

- gh-114013 [https://github.com/python/cpython/issues/114013]: Fix `Tools/wasm/wasi.py` to not include the path to `python.wasm` as part of `HOSTRUNNER`. The environment variable is meant to specify how to run the WASI host only, having `python.wasm` and relevant flags appended to the `HOSTRUNNER`. This fixes `make test` work.

- gh-113258 [https://github.com/python/cpython/issues/113258]: Changed the Windows build to write out generated frozen modules into the build tree instead of the source tree.
- gh-112305 [https://github.com/python/cpython/issues/112305]: Fixed the `check-clean-src` step performed on out of tree builds to detect errant `$(srcdir)/Python/frozen_modules/*.h` files and recommend appropriate source tree cleanup steps to get a working build again.
- gh-112536 [https://github.com/python/cpython/issues/112536]: Add support for thread sanitizer (TSAN)
- gh-112867 [https://github.com/python/cpython/issues/112867]: Fix the build for the case that WITH_PYMALLOC_RADIX_TREE=0 set.
- gh-103065 [https://github.com/python/cpython/issues/103065]: Introduce `Tools/wasm/wasi.py` to simplify doing a WASI build.
- bpo-11102 [https://bugs.python.org/issue?@action=redirect&bpo=11102]: The **`os.major()`**, **`os.makedev()`**, and **`os.minor()`** functions are now available on HP-UX v3.
- bpo-36351 [https://bugs.python.org/issue?@action=redirect&bpo=36351]: Do not set ipv6type when cross-compiling.

Windows

- gh-114096 [https://github.com/python/cpython/issues/114096]: Process privileges that are activated for creating directory junctions are now restored afterwards, avoiding behaviour changes in other parts of the program.
- gh-111877 [https://github.com/python/cpython/issues/111877]: **`os.stat()`** calls were returning incorrect time values for files that could not be accessed directly.
- gh-111973 [https://github.com/python/cpython/issues/111973]: Update Windows installer to use SQLite 3.44.2.

- gh-113009 [https://github.com/python/cpython/issues/113009]: **`multiprocessing`**: On Windows, fix a race condition in `Process.terminate()`: no longer set the `returncode` attribute to always call `WaitForSingleObject()` in `Process.wait()`. Previously, sometimes the process was still running after `TerminateProcess()` even if `GetExitCodeProcess()` is not `STILL_ACTIVE`. Patch by Victor Stinner.
- gh-86179 [https://github.com/python/cpython/issues/86179]: Fixes path calculations when launching Python on Windows through a symlink.
- gh-71383 [https://github.com/python/cpython/issues/71383]: Update Tcl/Tk in Windows installer to 8.6.13 with a patch to suppress incorrect ThemeChanged warnings.
- gh-111650 [https://github.com/python/cpython/issues/111650]: Ensures the `Py_GIL_DISABLED` preprocessor variable is defined in `pyconfig.h` so that extension modules written in C are able to use it.
- gh-112278 [https://github.com/python/cpython/issues/112278]: Reduce the time cost for some functions in **`platform`** on Windows if current user has no permission to the WMI.
- gh-73427 [https://github.com/python/cpython/issues/73427]: Deprecate **`sys._enablelegacywindowsfsencoding()`**. Use **`PYTHONLEGACYWINDOWSFSENCODING`** instead. Patch by Inada Naoki.
- gh-87868 [https://github.com/python/cpython/issues/87868]: Correctly sort and remove duplicate environment variables in **`_winapi.CreateProcess()`**.
- bpo-37308 [https://bugs.python.org/issue?@action=redirect&bpo=37308]: Fix mojibake in **`mmap.mmap`** when using a non-ASCII *tagname* argument on Windows.

macOS

- gh-113666 [https://github.com/python/cpython/issues/113666]: Add the following constants to module **`stat`**: `UF_SETTABLE`, `UF_TRACKED`, `UF_DATAVAULT`, `SF_SUPPORTED`, `SF_SETTABLE`, `SF_SYNTHETIC`, `SF_RESTRICTED`, `SF_FIRMLINK` and `SF_DATALESS`. The values `UF_SETTABLE`, `SF_SUPPORTED`, `SF_SETTABLE` and `SF_SYNTHETIC` are only available on macOS.

- gh-113536 [https://github.com/python/cpython/issues/113536]: `os.waitid()` is now available on macOS
- gh-110459 [https://github.com/python/cpython/issues/110459]: Running `configure ... --with-openssl-rpath=X/Y/Z` no longer fails to detect OpenSSL on macOS.
- gh-74573 [https://github.com/python/cpython/issues/74573]: Document that `dbm.ndbm` can silently corrupt DBM files on updates when exceeding undocumented platform limits, and can crash (segmentation fault) when reading such a corrupted file. (FB8919203)
- gh-65701 [https://github.com/python/cpython/issues/65701]: The **freeze** tool doesn’t work with framework builds of Python. Document this and bail out early when running the tool with such a build.
- gh-87277 [https://github.com/python/cpython/issues/87277]: webbrowser: Don’t look for X11 browsers on macOS. Those are generally not used and probing for them can result in starting XQuartz even if it isn’t used otherwise.
- gh-111973 [https://github.com/python/cpython/issues/111973]: Update macOS installer to use SQLite 3.44.2.
- gh-108269 [https://github.com/python/cpython/issues/108269]: Set `CFBundleAllowMixedLocalizations` to true in the Info.plist for the framework, embedded Python.app and IDLE.app with framework installs on macOS. This allows applications to pick up the user’s preferred locale when that’s different from english.
- gh-102362 [https://github.com/python/cpython/issues/102362]: Make sure the result of **`sysconfig.get_plaform()`** includes at least a major and minor versions, even if `MACOSX_DEPLOYMENT_TARGET` is set to only a major version during build to match the format expected by pip.
- gh-110017 [https://github.com/python/cpython/issues/110017]: Disable a signal handling stress test on macOS due to a

bug in macOS (FB13453490).
- gh-110820 [https://github.com/python/cpython/issues/110820]: Make sure the preprocessor definitions for `ALIGNOF_MAX_ALIGN_T`, `SIZEOF_LONG_DOUBLE` and `HAVE_GCC_ASM_FOR_X64` are correct for Universal 2 builds on macOS.
- gh-109981 [https://github.com/python/cpython/issues/109981]: Use `/dev/fd` on macOS to determine the number of open files in `test.support.os_helper.fd_count` to avoid a crash with “guarded” file descriptors when probing for open files.

IDLE

- gh-72284 [https://github.com/python/cpython/issues/72284]: Improve the lists of features, editor key bindings, and shell key bingings in the IDLE doc.
- gh-113903 [https://github.com/python/cpython/issues/113903]: Fix rare failure of test.test_idle, in test_configdialog.
- gh-113729 [https://github.com/python/cpython/issues/113729]: Fix the “Help -> IDLE Doc” menu bug in 3.11.7 and 3.12.1.
- gh-113269 [https://github.com/python/cpython/issues/113269]: Fix test_editor hang on macOS Catalina.
- gh-112898 [https://github.com/python/cpython/issues/112898]: Fix processing unsaved files when quitting IDLE on macOS.
- bpo-13586 [https://bugs.python.org/issue?@action=redirect&bpo=13586]: Enter the selected text when opening the “Replace” dialog.

C API

- gh-106560 [https://github.com/python/cpython/issues/106560]: Fix redundant declarations in the public C API. Declare PyBool_Type, PyLong_Type and PySys_Audit() only once. Patch by Victor Stinner.
- gh-112438 [https://github.com/python/cpython/issues/112438]: Fix support of format units “es”, “et”, “es#”, and “et#” in nested tuples in `PyArg_ParseTuple()`-like functions.

- gh-111545 [https://github.com/python/cpython/issues/111545]: Add `Py_HashPointer()` function to hash a pointer. Patch by Victor Stinner.
- gh-65210 [https://github.com/python/cpython/issues/65210]: Change the declaration of the *keywords* parameter of `PyArg_ParseTupleAndKeywords()` and `PyArg_VaParseTupleAndKeywords()` for better compatibility with C++.

Python 3.13.0 alpha 2

Release date: 2023-11-22

Core and Builtins

- gh-112243 [https://github.com/python/cpython/issues/112243]: Don’t include comments in f-string debug expressions. Patch by Pablo Galindo

- gh-112287 [https://github.com/python/cpython/issues/112287]: Slightly optimize the Tier 2 (uop) interpreter by only loading `oparg` and `operand` when needed. Also double the trace size limit again, to 512 this time.

- gh-112266 [https://github.com/python/cpython/issues/112266]: Change docstrings of `__dict__` and `__weakref__`.

- gh-111807 [https://github.com/python/cpython/issues/111807]: Lower the max parser stack depth to 1000 under WASI debug builds.

- gh-111798 [https://github.com/python/cpython/issues/111798]: When Python is built in debug mode, set the C recursion limit to 500 instead of 1500. A debug build is likely built with low optimization level which implies higher stack memory usage than a release build. Patch by Victor Stinner.

- gh-106529 [https://github.com/python/cpython/issues/106529]: Enable translating unspecialized `FOR_ITER` to Tier 2.
- gh-111916 [https://github.com/python/cpython/issues/111916]: Make hashlib related modules thread-safe without the GIL
- gh-81137 [https://github.com/python/cpython/issues/81137]: Deprecate assignment to a function’s `__code__` field when the new code object is of a mismatched type (e.g., from a generator to a plain function).
- gh-79932 [https://github.com/python/cpython/issues/79932]: Raise exception if **`frame.clear()`** is called on a suspended frame.
- gh-81925 [https://github.com/python/cpython/issues/81925]: Implement native thread ids for GNU KFreeBSD.
- gh-111843 [https://github.com/python/cpython/issues/111843]: Use exponential backoff to reduce the number of failed tier 2 optimization attempts by over 99%.
- gh-110829 [https://github.com/python/cpython/issues/110829]: Joining a thread now ensures the underlying OS thread has exited. This is required for safer fork() in multi-threaded processes.
- gh-109369 [https://github.com/python/cpython/issues/109369]: Make sure that tier 2 traces are de-optimized if the code is instrumented
- gh-111772 [https://github.com/python/cpython/issues/111772]: Specialize slot loads and stores for _Py_T_OBJECT as well as Py_T_OBJECT_EX

- gh-111666 [https://github.com/python/cpython/issues/111666]: Speed up `BaseExceptionGroup.derive()`, `BaseExceptionGroup.subgroup()`, and `BaseExceptionGroup.split()` by changing how they parse passed arguments.

- gh-111654 [https://github.com/python/cpython/issues/111654]: Fix runtime crash when some error happens in opcode `LOAD_FROM_DICT_OR_DEREF`.

- gh-111623 [https://github.com/python/cpython/issues/111623]: Add support for sharing tuples between interpreters using the cross-interpreter API. Patch by Anthony Shaw.

- gh-111354 [https://github.com/python/cpython/issues/111354]: The oparg of `YIELD_VALUE` is now `1` if the instruction is part of a yield-from or await, and `0` otherwise.

  The SUSPENDED frame state is now split into `SUSPENDED` and `SUSPENDED_YIELD_FROM`. This simplifies the code in `_PyGen_yf`.

- gh-111520 [https://github.com/python/cpython/issues/111520]: Merge the Tier 1 (bytecode) and Tier 2 (micro-ops) interpreters together, moving the Tier 2 interpreter loop and switch into `_PyEval_EvalFrameDefault()` in `Python/ceval.c`. The `Python/executor.c` file is gone. Also the `TIER_ONE` and `TIER_TWO` macros are now handled by the code generator.

  Beware! This changes the environment variables to enable micro-ops and their debugging to `PYTHON_UOPS` and `PYTHON_LLTRACE`.

- gh-109181 [https://github.com/python/cpython/issues/109181]: Speed up **`Traceback`** object creation by lazily compute the line number. Patch by Pablo Galindo

- gh-111420 [https://github.com/python/cpython/issues/111420]: Allow type comments in parenthesized `with` statements

- gh-111438 [https://github.com/python/cpython/issues/111438]: Add support for sharing floats between interpreters using the cross-interpreter API. Patch by Anthony Shaw.

- gh-111435 [https://github.com/python/cpython/issues/111435]: Add support for sharing of `True` and `False` between interpreters using the cross-interpreter API. Patch by Anthony Shaw.

- gh-102388 [https://github.com/python/cpython/issues/102388]: Fix a bug where `iso2022_jp_3` and `iso2022_jp_2004` codecs read out of bounds

- gh-111366 [https://github.com/python/cpython/issues/111366]: Fix an issue in the **`codeop`** that was causing **`SyntaxError`** exceptions raised in the presence of invalid syntax to not contain precise error messages. Patch by Pablo Galindo

- gh-111380 [https://github.com/python/cpython/issues/111380]: Fix a bug that was causing **`SyntaxWarning`** to appear twice when parsing if invalid syntax is encountered later. Patch by Pablo galindo

- gh-111374 [https://github.com/python/cpython/issues/111374]: Added a new environment variable **`PYTHON_FROZEN_MODULES`**. It determines whether or not frozen modules are ignored by the import machinery, equivalent of the **`-X frozen_modules`** command-line option.

- gh-111354 [https://github.com/python/cpython/issues/111354]: Remove `oparg` from **`YIELD_VALUE`**. Change `oparg` of **`RESUME`** to include information about the except-depth. These changes make it possible to simplify the code in generator close.

- gh-94438 [https://github.com/python/cpython/issues/94438]: Fix a regression that prevented jumping across `is None` and `is not None` when debugging. Patch by Savannah Ostrowski.

- gh-67224 [https://github.com/python/cpython/issues/67224]: Show source lines in tracebacks when using the `-c` option when running Python. Patch by Pablo Galindo

- gh-111123 [https://github.com/python/cpython/issues/111123]: Fix a bug where a **`global`** declaration in an **`except`** block is rejected when the global is used in the **`else`** block.

- gh-110938 [https://github.com/python/cpython/issues/110938]: Fix error messages for indented blocks with functions and classes with generic type parameters. Patch by Pablo Galindo

- gh-109214 [https://github.com/python/cpython/issues/109214]: Remove unnecessary instruction pointer updates before returning from frames.

- gh-110912 [https://github.com/python/cpython/issues/110912]: Correctly display the traceback for **`MemoryError`** exceptions using the **`traceback`** module. Patch by Pablo Galindo

- gh-109894 [https://github.com/python/cpython/issues/109894]: Fixed crash due to improperly initialized static **`MemoryError`** in subinterpreter.

- gh-110892 [https://github.com/python/cpython/issues/110892]: Return `NULL` for `PyTrace_RETURN` events caused by an exception

- gh-110864 [https://github.com/python/cpython/issues/110864]: Fix argument parsing by `_PyArg_UnpackKeywordsWithVararg` for functions defining pos-or-keyword, vararg, and kw-only parameters.

- gh-109094 [https://github.com/python/cpython/issues/109094]: Replace `prev_instr` on the interpreter frame by `instr_ptr` which points to the beginning of the instruction that is currently executing (or will execute once the frame resumes).

- gh-110805 [https://github.com/python/cpython/issues/110805]: Allow the repl to show source code and complete tracebacks. Patch by Pablo Galindo

- gh-110722 [https://github.com/python/cpython/issues/110722]: Add **`PYTHON_PRESITE=package.module`** to import a module early in the interpreter lifecycle before `site.py` is executed. Python needs to be built in debug mode for this option to exist.

- gh-110481 [https://github.com/python/cpython/issues/110481]: Implement biased reference counting in `--disable-gil` builds.

- gh-110543 [https://github.com/python/cpython/issues/110543]: Fix regression in Python 3.12 where **`types.CodeType.replace()`** would produce a broken code object if called on a module or class code object that contains a comprehension. Patch by Jelle Zijlstra.

- gh-89519 [https://github.com/python/cpython/issues/89519]: Removed chained **`classmethod`** descriptors (introduced in bpo-19072 [https://bugs.python.org/issue?@action=redirect&bpo=19072]). This can no longer be used to wrap other descriptors such as **`property`**. The core design of this feature was flawed and caused a number of downstream problems. To “pass-through” a `classmethod`, consider using the `__wrapped__` attribute that was added in Python 3.10.

- gh-103615 [https://github.com/python/cpython/issues/103615]: Use local events for opcode tracing
- bpo-46657 [https://bugs.python.org/issue?@action=redirect&bpo=46657]: Add mimalloc memory allocator support.
- gh-106718 [https://github.com/python/cpython/issues/106718]: When PyConfig.stdlib_dir is explicitly set, it’s now respected and won’t be overridden by PyConfig.home.
- gh-106905 [https://github.com/python/cpython/issues/106905]: Fix incorrect SystemError about AST constructor recursion depth mismatch.
- gh-100445 [https://github.com/python/cpython/issues/100445]: Improve error message for unterminated strings with escapes.
- bpo-45759 [https://bugs.python.org/issue?@action=redirect&bpo=45759]: Improved error messages for `elif`/`else` statements not matching any valid statements. Patch by Jeremiah Vivian.

Library

- gh-111942 [https://github.com/python/cpython/issues/111942]: Fix SystemError in the TextIOWrapper constructor with non-encodable “errors” argument in non-debug mode.
- gh-111995 [https://github.com/python/cpython/issues/111995]: Added the `NI_IDN` constant to the `socket` module when

present in C at build time for use with `socket.getnameinfo()`.

- gh-109538 [https://github.com/python/cpython/issues/109538]: Issue warning message instead of having `RuntimeError` be displayed when event loop has already been closed at `StreamWriter.__del__()`.
- gh-111942 [https://github.com/python/cpython/issues/111942]: Fix crashes in `io.TextIOWrapper.reconfigure()` when pass invalid arguments, e.g. non-string encoding.
- gh-111460 [https://github.com/python/cpython/issues/111460]: `curses`: restore wide character support (including `curses.unget_wch()` and `get_wch()`) on macOS, which was unavailable due to a regression in Python 3.12.
- gh-103791 [https://github.com/python/cpython/issues/103791]: `contextlib.suppress` now supports suppressing exceptions raised as part of a `BaseExceptionGroup`, in addition to the recent support for `ExceptionGroup`.
- gh-111835 [https://github.com/python/cpython/issues/111835]: The `mmap.mmap` class now has an `seekable()` method that can be used when a seekable file-like object is required. The `seek()` method now returns the new absolute position. Patch by Donghee Na.
- gh-111804 [https://github.com/python/cpython/issues/111804]: Remove posix.fallocate() under WASI as the underlying posix_fallocate() is not available in WASI preview2.
- gh-111841 [https://github.com/python/cpython/issues/111841]: Fix truncating arguments on an embedded null character in `os.putenv()` and `os.unsetenv()` on Windows.

- gh-111768 [https://github.com/python/cpython/issues/111768]: **`wsgiref.util.is_hop_by_hop()`** is now exposed correctly in `__all__`.

- gh-80731 [https://github.com/python/cpython/issues/80731]: Avoid executing the default function in **`cmd.Cmd`** in an except block

- gh-111541 [https://github.com/python/cpython/issues/111541]: Fix **`doctest`** for **`SyntaxError`** not-builtin subclasses.

- gh-111719 [https://github.com/python/cpython/issues/111719]: Add extra argument validation for `alias` command in **`pdb`**

- gh-111482 [https://github.com/python/cpython/issues/111482]: **`time`**: Make **`time.clock_gettime()`** and **`time.clock_gettime_ns()`** functions up to 2x faster by faster calling convention. Patch by Victor Stinner.

- gh-110894 [https://github.com/python/cpython/issues/110894]: Call loop exception handler for exceptions in `client_connected_cb` of **`asyncio.start_server()`** so that applications can handle it. Patch by Kumar Aditya.

- gh-111531 [https://github.com/python/cpython/issues/111531]: Fix reference leaks in `bind_class()` and `bind_all()` methods of **`tkinter`** widgets.

- gh-111246 [https://github.com/python/cpython/issues/111246]: **`asyncio.loop.create_unix_server()`** will now automatically remove the Unix socket when the server is closed.

- gh-111356 [https://github.com/python/cpython/issues/111356]: Added **`io.text_encoding()`**, **`io.DEFAULT_BUFFER_SIZE`**, and **`io.IncrementalNewlineDecoder`** to `io.__all__`.

- gh-66425 [https://github.com/python/cpython/issues/66425]: Remove the code to set the REMOTE_HOST header from wsgiref module, as it is unreachable. This header is used for performance reasons, which is not necessary in the wsgiref module.

- gh-111429 [https://github.com/python/cpython/issues/111429]: Speed up **`pathlib.PurePath.relative_to()`** and **`is_relative_to()`**.

- gh-111342 [https://github.com/python/cpython/issues/111342]: Fixed typo in **`math.sumprod()`**.

- gh-68166 [https://github.com/python/cpython/issues/68166]: Remove mention of not supported “vsapi” element type in **`tkinter.ttk.Style.element_create()`**. Add tests for `element_create()` and other `ttk.Style` methods. Add examples for `element_create()` in the documentation.

- gh-111388 [https://github.com/python/cpython/issues/111388]: Add `show_group` parameter to **`traceback.format_exception_only()`**, which allows to format **`ExceptionGroup`** instances.

- gh-79033 [https://github.com/python/cpython/issues/79033]: Another attempt at fixing **`asyncio.Server.wait_closed()`**. It now blocks until both conditions are true: the server is closed, *and* there are no more active connections. (This means that in some cases where in 3.12.0 this function would *incorrectly* have returned immediately, it will now block; in particular, when there are no active connections but the server hasn’t been closed yet.)

- gh-111259 [https://github.com/python/cpython/issues/111259]: Optimize recursive wildcards in **`pathlib`**.

- gh-111295 [https://github.com/python/cpython/issues/111295]: Fix **`time`** not checking for errors when initializing.

- gh-111253 [https://github.com/python/cpython/issues/111253]: Add error checking during **`_socket`** module init.

- gh-111251 [https://github.com/python/cpython/issues/111251]: Fix **`_blake2`** not checking for errors when initializing.

- gh-111233 [https://github.com/python/cpython/issues/111233]: Fix **`select`** not checking for errors when initializing.

- gh-111230 [https://github.com/python/cpython/issues/111230]: Fix **`ssl`** not checking for errors when initializing.

- gh-111174 [https://github.com/python/cpython/issues/111174]: Fix crash in **`io.BytesIO.getbuffer()`** called repeatedly for empty BytesIO.

- gh-111187 [https://github.com/python/cpython/issues/111187]: Postpone removal version for locale.getdefaultlocale() to Python 3.15.

- gh-111159 [https://github.com/python/cpython/issues/111159]: Fix **`doctest`** output comparison for exceptions with notes.

- gh-110910 [https://github.com/python/cpython/issues/110910]: Fix invalid state handling in **`asyncio.TaskGroup`** and **`asyncio.Timeout`**. They now raise proper RuntimeError if they are improperly used and are left in consistent state after this.

- gh-111092 [https://github.com/python/cpython/issues/111092]: Make turtledemo run without default root enabled.

- gh-110944 [https://github.com/python/cpython/issues/110944]: Support alias and convenience vars for **`pdb`** completion

- gh-110745 [https://github.com/python/cpython/issues/110745]: Added *newline* parameter to **`pathlib.Path.read_text()`**.

Patch by Junya Okabe.

- gh-84583 [https://github.com/python/cpython/issues/84583]: Make **`pdb`** enter post-mortem mode even for **`SyntaxError`**
- gh-80675 [https://github.com/python/cpython/issues/80675]: Set `f_trace_lines = True` on all frames upon **`pdb.set_trace()`**
- gh-110771 [https://github.com/python/cpython/issues/110771]: Expose the setup and cleanup portions of `asyncio.run_forever()` as the standalone methods `asyncio.run_forever_setup()` and `asyncio.run_forever_cleanup()`. This allows for tighter integration with GUI event loops.
- gh-110774 [https://github.com/python/cpython/issues/110774]: Support setting the **`asyncio.Runner`** loop_factory kwarg in **`unittest.IsolatedAsyncioTestCase`**
- gh-110392 [https://github.com/python/cpython/issues/110392]: Fix **`tty.setraw()`** and **`tty.setcbreak()`**: previously they returned partially modified list of the original tty attributes. **`tty.cfmakeraw()`** and **`tty.cfmakecbreak()`** now make a copy of the list of special characters before modifying it.
- gh-59013 [https://github.com/python/cpython/issues/59013]: Make line number of function breakpoint more precise in **`pdb`**
- gh-88434 [https://github.com/python/cpython/issues/88434]: Emit deprecation warning for non-integer numbers in **`gettext`** functions and methods that consider plural forms even if the translation was not found.
- gh-110395 [https://github.com/python/cpython/issues/110395]: Ensure that **`select.kqueue()`** objects correctly appear as closed in forked children, to prevent operations on an invalid file descriptor.

- gh-110196 [https://github.com/python/cpython/issues/110196]: Add `__reduce__` method to **`IPv6Address`** in order to keep `scope_id`

- gh-109747 [https://github.com/python/cpython/issues/109747]: Improve errors for unsupported look-behind patterns. Now re.error is raised instead of OverflowError or RuntimeError for too large width of look-behind pattern.

- gh-109466 [https://github.com/python/cpython/issues/109466]: Add the **`ipaddress.IPv4Address.ipv6_mapped`** property, which returns the IPv4-mapped IPv6 address.

- gh-85098 [https://github.com/python/cpython/issues/85098]: Implement the CLI of the **`symtable`** module and improve the repr of **`Symbol`**.

- gh-108791 [https://github.com/python/cpython/issues/108791]: Improved error handling in **`pdb`** command line interface, making it produce more concise error messages.

- gh-105931 [https://github.com/python/cpython/issues/105931]: Change **`compileall`** to only strip the stripdir prefix from the full path recorded in the compiled `.pyc` file, when the prefix matches the start of the full path in its entirety. When the prefix does not match, no stripping is performed and a warning to this effect is displayed.

  Previously all path components of the stripdir prefix that matched the full path were removed, while those that did not match were left alone (including ones interspersed between matching components).

- gh-107431 [https://github.com/python/cpython/issues/107431]: Make the `DictProxy` and `ListProxy` types in **`multiprocessing.managers`** Generic Alias Types for `[]` use in typing contexts.

- gh-72904 [https://github.com/python/cpython/issues/72904]: Add `glob.translate()`. This function converts a pathname with shell-style wildcards to a regular expression.

- gh-90026 [https://github.com/python/cpython/issues/90026]: Define `USE_XATTRS` on Cygwin so that XATTR-related functions in the `os` module become available.

- gh-90890 [https://github.com/python/cpython/issues/90890]: New methods `mailbox.Maildir.get_info()`, `mailbox.Maildir.set_info()`, `mailbox.Maildir.get_flags()`, `mailbox.Maildir.set_flags()`, `mailbox.Maildir.add_flag()`, `mailbox.Maildir.remove_flag()`. These methods speed up accessing a message’s info and/or flags and are useful when it is not necessary to access the message’s contents, as when iterating over a Maildir to find messages with specific flags.

- gh-102956 [https://github.com/python/cpython/issues/102956]: Fix returning of empty byte strings after seek in zipfile module

- gh-102895 [https://github.com/python/cpython/issues/102895]: Added a parameter `local_exit` for `code.interact()` to prevent `exit()` and `quit` from closing `sys.stdin` and raise `SystemExit`.

- gh-97928 [https://github.com/python/cpython/issues/97928]: Change the behavior of **`tkinter.Text.count()`**. It now always returns an integer if one or less counting options are specified. Previously it could return a single count as a 1-tuple, an integer (only if option `"update"` was specified) or `None` if no items found. The result is now the same if `wantobjects` is set to `0`.

- gh-96954 [https://github.com/python/cpython/issues/96954]: Switch the storage of the unicode codepoint names to use a different data-structure, a directed acyclic word graph [https://en.wikipedia.org/wiki/Deterministic_acyclic_finite_state_automaton] . This makes the unicodedata shared library about 440 KiB smaller. Contributed by Carl Friedrich Bolz-Tereick using code from the PyPy project.

- gh-73561 [https://github.com/python/cpython/issues/73561]: Omit the interface scope from an IPv6 address when used as Host header by **`http.client`**.

- gh-86826 [https://github.com/python/cpython/issues/86826]: **`zipinfo`** now supports the full range of values in the TZ string determined by RFC 8536 and detects all invalid formats. Both Python and C implementations now raise exceptions of the same type on invalid data.

Tests

- gh-111808 [https://github.com/python/cpython/issues/111808]: Make the default value of `test.support.infinite_recursion()` to be conditional based on whether optimizations were used when compiling the interpreter. This helps with platforms like WASI whose stack size is greatly restricted in debug builds.
- gh-110722 [https://github.com/python/cpython/issues/110722]: Gathering line coverage of standard libraries within the regression test suite is now precise, as well as much faster. Patch by Łukasz Langa.
- gh-110367 [https://github.com/python/cpython/issues/110367]: Make regrtest `--verbose3` option compatible with `--huntrleaks -jN` options. The `./python -m test -j1 -R 3:3 --verbose3` command now works as expected. Patch by Victor Stinner.
- gh-111165 [https://github.com/python/cpython/issues/111165]: Remove no longer used functions `run_unittest()` and `run_doctest()` from the **`test.support`** module.

- gh-110932 [https://github.com/python/cpython/issues/110932]: Fix regrtest if the `SOURCE_DATE_EPOCH` environment variable is defined: use the variable value as the random seed. Patch by Victor Stinner.
- gh-110995 [https://github.com/python/cpython/issues/110995]: test_gdb: Fix detection of gdb built without Python scripting support. Patch by Victor Stinner.
- gh-110918 [https://github.com/python/cpython/issues/110918]: Test case matching patterns specified by options `--match`, `--ignore`, `--matchfile` and `--ignorefile` are now tested in the order of specification, and the last match determines whether the test case be run or ignored.
- gh-108747 [https://github.com/python/cpython/issues/108747]: Add unit test for `usercustomize` and `sitecustomize` hooks from **`site`**.

Build

- gh-96954 [https://github.com/python/cpython/issues/96954]: Make `make regen-unicodedata` work for out-of-tree builds of CPython.
- gh-112088 [https://github.com/python/cpython/issues/112088]: Add `Tools/build/regen-configure.sh` script to regenerate the `configure` with an Ubuntu container image. The `quay.io/tiran/cpython_autoconf:271` container image (tiran/cpython_autoconf [https://github.com/tiran/cpython_autoconf]) is no longer used. Patch by Victor Stinner.
- gh-111046 [https://github.com/python/cpython/issues/111046]: For wasi-threads, memory is now exported to fix compatibility issues with some wasm runtimes.
- gh-110828 [https://github.com/python/cpython/issues/110828]: AIX 32bit needs `-latomic` to build the **`_testcapi`** extension module.
- gh-85283 [https://github.com/python/cpython/issues/85283]: The `errno`, `md5`, `resource`, `winsound`, `_ctypes_test`,

`_multiprocessing.posixshmem`, `_scproxy`, `_stat`, `_testimportmultiple` and `_uuid` C extensions are now built with the limited C API. Patch by Victor Stinner.

Windows

- gh-111856 [https://github.com/python/cpython/issues/111856]: Fixes **`fstat()`** on file systems that do not support file ID requests. This includes FAT32 and exFAT.
- gh-111293 [https://github.com/python/cpython/issues/111293]: Fix **`os.DirEntry.inode`** dropping higher 64 bits of a file id on some filesystems on Windows.
- gh-110913 [https://github.com/python/cpython/issues/110913]: WindowsConsoleIO now correctly chunks large buffers without splitting up UTF-8 sequences.

macOS

- gh-59703 [https://github.com/python/cpython/issues/59703]: For macOS framework builds, in `getpath.c` use the system `dladdr` function to find the path to the shared library rather than depending on deprecated macOS APIs.
- gh-110950 [https://github.com/python/cpython/issues/110950]: Update macOS installer to include an upstream Tcl/Tk fix for the `Secure coding is not enabled for restorable state!` warning encountered in Tkinter on macOS 14 Sonoma.
- gh-111015 [https://github.com/python/cpython/issues/111015]: Ensure that IDLE.app and Python Launcher.app are installed with appropriate permissions on macOS builds.
- gh-71383 [https://github.com/python/cpython/issues/71383]: Update macOS installer to include an upstream Tcl/Tk fix for the `ttk::ThemeChanged` error encountered in Tkinter.
- gh-92603 [https://github.com/python/cpython/issues/92603]: Update macOS installer to include a fix accepted by upstream Tcl/Tk for a crash encountered after the first **`tkinter.Tk()`** instance is destroyed.

IDLE

- bpo-35668 [https://bugs.python.org/issue?@action=redirect&bpo=35668]: Add docstrings to the IDLE debugger module. Fix two bugs: initialize `Idb.botframe` (should be in Bdb); in `Idb.in_rpc_code`, check whether `prev_frame` is `None` before trying to use it. Greatly expand test_debugger.

Tools/Demos

- gh-111903 [https://github.com/python/cpython/issues/111903]: Argument Clinic now supports the `@critical_section` directive that instructs Argument Clinic to generate a critical section around the function call, which locks the `self` object in `--disable-gil` builds. Patch by Sam Gross.

C API

- gh-112026 [https://github.com/python/cpython/issues/112026]: Add again the private `_PyThreadState_UncheckedGet()` function as an alias to the new public **`PyThreadState_GetUnchecked()`** function. Patch by Victor Stinner.

- gh-112026 [https://github.com/python/cpython/issues/112026]: Restore the removed `_PyDict_GetItemStringWithError()` function. It is used by numpy. Patch by Victor Stinner.

- gh-112026 [https://github.com/python/cpython/issues/112026]: Restore removed private C API functions, macros and structures which have no simple replacement for now:
  - _PyDict_GetItem_KnownHash()
  - _PyDict_NewPresized()
  - _PyHASH_BITS
  - _PyHASH_IMAG

- _PyHASH_INF
- _PyHASH_MODULUS
- _PyHASH_MULTIPLIER
- _PyLong_Copy()
- _PyLong_FromDigits()
- _PyLong_New()
- _PyLong_Sign()
- _PyObject_CallMethodId()
- _PyObject_CallMethodNoArgs()
- _PyObject_CallMethodOneArg()
- _PyObject_CallOneArg()
- _PyObject_EXTRA_INIT
- _PyObject_FastCallDict()
- _PyObject_GetAttrId()
- _PyObject_Vectorcall()
- _PyObject_VectorcallMethod()
- _PyStack_AsDict()
- _PyThread_CurrentFrames()
- _PyUnicodeWriter structure
- _PyUnicodeWriter_Dealloc()
- _PyUnicodeWriter_Finish()
- _PyUnicodeWriter_Init()
- _PyUnicodeWriter_Prepare()
- _PyUnicodeWriter_PrepareKind()
- _PyUnicodeWriter_WriteASCIIString()
- _PyUnicodeWriter_WriteChar()
- _PyUnicodeWriter_WriteLatin1String()
- _PyUnicodeWriter_WriteStr()
- _PyUnicodeWriter_WriteSubstring()
- _PyUnicode_AsString()
- _PyUnicode_FromId()
- _PyVectorcall_Function()
- _Py_IDENTIFIER()
- _Py_c_abs()
- _Py_c_diff()
- _Py_c_neg()

  - _Py_c_pow()
  - _Py_c_prod()
  - _Py_c_quot()
  - _Py_c_sum()
  - _Py_static_string()
  - _Py_static_string_init()

  Patch by Victor Stinner.

- gh-112026 [https://github.com/python/cpython/issues/112026]: Add again `<ctype.h>` and `<unistd.h>` includes in `Python.h`, but don’t include them in the limited C API version 3.13 and newer. Patch by Victor Stinner.

- gh-111956 [https://github.com/python/cpython/issues/111956]: Add internal-only one-time initialization API: `_PyOnceFlag` and `_PyOnceFlag_CallOnce`.

- gh-111262 [https://github.com/python/cpython/issues/111262]: Add **`PyDict_Pop()`** and **`PyDict_PopString()`** functions: remove a key from a dictionary and optionally return the removed value. This is similar to **`dict.pop()`**, but without the default value and not raising **`KeyError`** if the key missing. Patch by Stefan Behnel and Victor Stinner.

- gh-111863 [https://github.com/python/cpython/issues/111863]: Rename `Py_NOGIL` to `Py_GIL_DISABLED`. Patch by Hugo van Kemenade.

- gh-111138 [https://github.com/python/cpython/issues/111138]: Add **`PyList_Extend()`** and **`PyList_Clear()`** functions: similar to Python `list.extend()` and `list.clear()` methods. Patch by Victor Stinner.

- gh-108765 [https://github.com/python/cpython/issues/108765]: On Windows, `Python.h` no longer includes the `<stddef.h>`

standard header file. If needed, it should now be included explicitly. Patch by Victor Stinner.

- gh-111569 [https://github.com/python/cpython/issues/111569]: Implement “Python Critical Sections” from **PEP 703** [https://peps.python.org/pep-0703/]. These are macros to help replace the GIL with per-object locks in the `--disable-gil` build of CPython. The macros are no-ops in the default build.

- gh-111506 [https://github.com/python/cpython/issues/111506]: In the limited C API version 3.13, **`Py_SET_REFCNT()`** function is now implemented as an opaque function call. Patch by Victor Stinner.

- gh-108082 [https://github.com/python/cpython/issues/108082]: Add **`PyErr_FormatUnraisable()`** function.

- gh-110964 [https://github.com/python/cpython/issues/110964]: Move the undocumented private _PyArg functions and _PyArg_Parser structure to internal C API (`pycore_modsupport.h`). Patch by Victor Stinner.

- gh-110815 [https://github.com/python/cpython/issues/110815]: Support non-ASCII keyword names in **`PyArg_ParseTupleAndKeywords()`**.

- gh-109587 [https://github.com/python/cpython/issues/109587]: Introduced **`PyUnstable_PerfTrampoline_CompileCode()`**, **`PyUnstable_PerfTrampoline_SetPersistAfterFork()`** and **`PyUnstable_CopyPerfMapFile()`**. These functions allow extension modules to initialize trampolines eagerly, after the application is “warmed up”. This makes it possible to have perf-trampolines running in an always-enabled fashion.

- gh-85283 [https://github.com/python/cpython/issues/85283]: Add the `PySys_Audit()` function to the limited C API. Patch by Victor Stinner.

- gh-85283 [https://github.com/python/cpython/issues/85283]: Add `PyMem_RawMalloc()`, `PyMem_RawCalloc()`, `PyMem_RawRealloc()` and `PyMem_RawFree()` to the limited C API. Patch by Victor Stinner.

- gh-106672 [https://github.com/python/cpython/issues/106672]: Functions `PyDict_GetItem()`, `PyDict_GetItemString()`, `PyMapping_HasKey()`, `PyMapping_HasKeyString()`, `PyObject_HasAttr()`, `PyObject_HasAttrString()`, and `PySys_GetObject()`, which clear all errors occurred during calling the function, report now them using `sys.unraisablehook()`.

- gh-67565 [https://github.com/python/cpython/issues/67565]: Remove redundant C-contiguity check in `getargs.c`, `binascii`, `ssl` and Argument Clinic. Patched by Stefan Krah and Furkan Onder

Python 3.13.0 alpha 1

Release date: 2023-10-13

Security

- gh-108310 [https://github.com/python/cpython/issues/108310]: Fixed an issue where instances of `ssl.SSLSocket` were vulnerable to a bypass of the TLS handshake and included protections (like certificate verification) and treating sent unencrypted data as if it were post-handshake TLS encrypted data. Security issue reported as **CVE 2023-40217** [https://www.cve.org/CVERecord?id=CVE-2023-40217] by Aapo Oksman. Patch by Gregory P. Smith.

- gh-107774 [https://github.com/python/cpython/issues/107774]: PEP 669 specifies that `sys.monitoring.register_callback` will generate an audit event. Pre-releases of Python 3.12 did not generate the audit event. This is now fixed.
- gh-102988 [https://github.com/python/cpython/issues/102988]: Reverted the **`email.utils`** security improvement change released in 3.12beta4 that unintentionally caused **`email.utils.getaddresses`** to fail to parse email addresses with a comma in the quoted name field. See gh-106669 [https://github.com/python/cpython/issues/106669].
- gh-99108 [https://github.com/python/cpython/issues/99108]: Refresh our new HACL* built-in **`hashlib`** code from upstream. Built-in SHA2 should be faster and an issue with SHA3 on 32-bit platforms is fixed.
- gh-102509 [https://github.com/python/cpython/issues/102509]: Start initializing `ob_digit` during creation of **`PyLongObject`** objects. Patch by Illia Volochii.

Core and Builtins

- gh-110782 [https://github.com/python/cpython/issues/110782]: Fix crash when **`typing.TypeVar`** is constructed with a keyword argument. Patch by Jelle Zijlstra.

- gh-110752 [https://github.com/python/cpython/issues/110752]: Reset `ceval.eval_breaker` in **`interpreter_clear()`**

- gh-110721 [https://github.com/python/cpython/issues/110721]: Use the **`traceback`** implementation for the default **`PyErr_Display()`** functionality. Patch by Pablo Galindo

- gh-110696 [https://github.com/python/cpython/issues/110696]: Fix incorrect error message for invalid argument unpacking. Patch by Pablo Galindo

- gh-104169 [https://github.com/python/cpython/issues/104169]: Split the tokenizer into two separate directories: - One part includes the actual lexeme producing logic and lives in `Parser/lexer`. - The second part wraps the lexer according to the different tokenization modes we have (string, utf-8, file, interactive, readline) and lives in `Parser/tokenizer`.

- gh-110688 [https://github.com/python/cpython/issues/110688]: Remove undocumented `test_c_api` method from **`set`**, which was only defined for testing purposes under `Py_DEBUG`. Now we have proper CAPI tests.

- gh-104584 [https://github.com/python/cpython/issues/104584]: Fix a reference leak when running with **`PYTHONUOPS`** or **`-X uops`** enabled.

- gh-110514 [https://github.com/python/cpython/issues/110514]: Add `PY_THROW` to **`sys.setprofile()`** events

- gh-110489 [https://github.com/python/cpython/issues/110489]: Optimise **`math.ceil()`** when the input is exactly a float, resulting in about a 10% improvement.

- gh-110455 [https://github.com/python/cpython/issues/110455]: Guard `assert(tstate->thread_id > 0)` with `#ifndef HAVE_PTHREAD_STUBS`. This allows for for pydebug builds to work under WASI which (currently) lacks thread support.

- gh-110309 [https://github.com/python/cpython/issues/110309]: Remove unnecessary empty constant nodes in the ast of f-string specs.

- gh-110259 [https://github.com/python/cpython/issues/110259]: Correctly identify the format spec in f-strings (with single or triple quotes) that have multiple lines in the expression part and include a formatting spec. Patch by Pablo Galindo

- gh-110237 [https://github.com/python/cpython/issues/110237]: Fix missing error checks for calls to `PyList_Append` in `_PyEval_MatchClass`.

- gh-110164 [https://github.com/python/cpython/issues/110164]: regrtest: If the `SOURCE_DATE_EPOCH` environment variable is defined, regrtest now disables tests randomization. Patch by Victor Stinner.

- gh-109889 [https://github.com/python/cpython/issues/109889]: Fix the compiler’s redundant NOP detection algorithm to skip over NOPs with no line number when looking for the next instruction’s lineno.

- gh-109853 [https://github.com/python/cpython/issues/109853]: `sys.path[0]` is now set correctly for subinterpreters.

- gh-109923 [https://github.com/python/cpython/issues/109923]: Set line number on the `POP_TOP` that follows a `RETURN_GENERATOR`.

- gh-105716 [https://github.com/python/cpython/issues/105716]: Subinterpreters now correctly handle the case where they have threads running in the background. Before, such threads would interfere with cleaning up and destroying them, as well as prevent running another script.

- gh-109369 [https://github.com/python/cpython/issues/109369]: The internal eval_breaker and supporting flags, plus the monitoring version have been merged into a single atomic integer to speed up checks.

- gh-109823 [https://github.com/python/cpython/issues/109823]: Fix bug where compiler does not adjust labels when removing an empty basic block which is a jump target.

- gh-109793 [https://github.com/python/cpython/issues/109793]: The main thread no longer exits prematurely when a subinterpreter is cleaned up during runtime finalization. The bug was a problem particularly because, when triggered, the Python process would always return with a 0 exitcode, even if it failed.

- gh-109719 [https://github.com/python/cpython/issues/109719]: Fix missing jump target labels when compiler reorders cold/warm blocks.

- gh-109595 [https://github.com/python/cpython/issues/109595]: Add **`-X cpu_count`** command line option to override return results of **`os.cpu_count()`** and **`os.process_cpu_count()`**. This option is useful for users who need to limit CPU resources of a container system without having to modify the container (application code). Patch by Donghee Na.

- gh-109627 [https://github.com/python/cpython/issues/109627]: Fix bug where the compiler does not assign a new jump target label to a duplicated small exit block.

- gh-109596 [https://github.com/python/cpython/issues/109596]: Fix some tokens in the grammar that were incorrectly marked as soft keywords. Also fix some repeated rule names and ensure that repeated rules are not allowed. Patch by Pablo Galindo

- gh-109496 [https://github.com/python/cpython/issues/109496]: On a Python built in debug mode, **`Py_DECREF()`** now calls `_Py_NegativeRefcount()` if the object is a dangling pointer to deallocated memory: memory filled with `0xDD` “dead byte” by the debug hook on memory allocators. The fix is to check the reference count *before* checking for `_Py_IsImmortal()`. Patch by Victor Stinner.

- gh-107265 [https://github.com/python/cpython/issues/107265]: Deopt opcodes hidden by the executor when base opcode is needed

- gh-109371 [https://github.com/python/cpython/issues/109371]: Deopted instructions correctly for tool initialization and modified the incorrect assertion in instrumentation, when a previous tool already sets INSTRUCTION events

- gh-105658 [https://github.com/python/cpython/issues/105658]: Fix bug where the line trace of an except block ending with a conditional includes an excess event with the line of the conditional expression.

- gh-109219 [https://github.com/python/cpython/issues/109219]: Fix compiling type param scopes that use a name which is also free in an inner scope.

- gh-109351 [https://github.com/python/cpython/issues/109351]: Fix crash when compiling an invalid AST involving a named (walrus) expression.

- gh-109341 [https://github.com/python/cpython/issues/109341]: Fix crash when compiling an invalid AST involving a **`ast.TypeAlias`**.

- gh-109195 [https://github.com/python/cpython/issues/109195]: Fix source location for the `LOAD_*` instruction preceding a `LOAD_SUPER_ATTR` to load the `super` global (or shadowing variable) so that it encompasses only the name `super` and not the following parentheses.

- gh-109256 [https://github.com/python/cpython/issues/109256]: Opcode IDs for specialized opcodes are allocated in their own range to improve stability of the IDs for the ‘real’ opcodes.

- gh-109216 [https://github.com/python/cpython/issues/109216]: Fix possible memory leak in **`BUILD_MAP`**.

- gh-109207 [https://github.com/python/cpython/issues/109207]: Fix a SystemError in `__repr__` of symtable entry object.

- gh-109179 [https://github.com/python/cpython/issues/109179]: Fix bug where the C traceback display drops notes from **`SyntaxError`**.

- gh-109118 [https://github.com/python/cpython/issues/109118]: Disallow nested scopes (lambdas, generator expressions, and comprehensions) within PEP 695 annotation scopes that are nested within classes.

- gh-109156 [https://github.com/python/cpython/issues/109156]: Add tests for de-instrumenting instructions while keeping the instrumentation for lines

- gh-109114 [https://github.com/python/cpython/issues/109114]: Relax the detection of the error message for invalid lambdas inside f-strings to not search for arbitrary replacement fields to avoid false positives. Patch by Pablo Galindo

- gh-105848 [https://github.com/python/cpython/issues/105848]: Add a new **`CALL_KW`** opcode, used for calls containing keyword arguments. Also, fix a possible crash when jumping over method calls in a debugger.

- gh-109052 [https://github.com/python/cpython/issues/109052]: Use the base opcode when comparing code objects to avoid interference from instrumentation

- gh-109118 [https://github.com/python/cpython/issues/109118]: Fix interpreter crash when a NameError is raised inside the type parameters of a generic class.

- gh-88943 [https://github.com/python/cpython/issues/88943]: Improve syntax error for non-ASCII character that follows a numerical literal. It now points on the invalid non-ASCII character, not on the valid numerical literal.

- gh-108976 [https://github.com/python/cpython/issues/108976]: Fix crash that occurs after de-instrumenting a code object in a monitoring callback.

- gh-108732 [https://github.com/python/cpython/issues/108732]: Make iteration variables of module- and class-scoped comprehensions visible to pdb and other tools that use `frame.f_locals` again.

- gh-108959 [https://github.com/python/cpython/issues/108959]: Fix caret placement for error locations for subscript and binary operations that involve non-semantic parentheses and spaces. Patch by Pablo Galindo

- gh-104584 [https://github.com/python/cpython/issues/104584]: Fix a crash when running with **`PYTHONUOPS`** or **`-X uops`** enabled and an error occurs during optimization.

- gh-108727 [https://github.com/python/cpython/issues/108727]: Define `tp_dealloc` for `CounterOptimizer_Type`. This fixes a segfault on deallocation.

- gh-108520 [https://github.com/python/cpython/issues/108520]: Fix **`multiprocessing.synchronize.SemLock.__setstate__()`** to properly initialize **`multiprocessing.synchronize.SemLock._is_fork_ctx`**. This fixes a regression when passing a SemLock across nested processes.

  Rename **`multiprocessing.synchronize.SemLock.is_fork_ctx`** to **`multiprocessing.synchronize.SemLock._is_fork_ctx`** to avoid exposing it as public API.

- gh-108654 [https://github.com/python/cpython/issues/108654]: Restore locals shadowed by an inlined comprehension if the comprehension raises an exception.

- gh-108488 [https://github.com/python/cpython/issues/108488]: Change the initialization of inline cache entries so that the cache entry for `JUMP_BACKWARD` is initialized to zero, instead of the `adaptive_counter_warmup()` value used for all other instructions. This counter, unique among instructions, counts up from zero.

- gh-108716 [https://github.com/python/cpython/issues/108716]: Turn off deep-freezing of code objects. Modules are still frozen, so that a file system search is not needed for common modules.

- gh-108614 [https://github.com/python/cpython/issues/108614]: Add RESUME_CHECK instruction, to avoid having to handle instrumentation, signals, and contexts switches in the tier 2 execution engine.

- gh-108487 [https://github.com/python/cpython/issues/108487]: Move an assert that would cause a spurious crash in a devious case that should only trigger deoptimization.

- gh-106176 [https://github.com/python/cpython/issues/106176]: Use a `WeakValueDictionary` to track the lists containing the modules each thread is currently importing. This helps avoid a reference leak from keeping the list around longer than necessary. Weakrefs are used as GC can’t interrupt the cleanup.

- gh-105481 [https://github.com/python/cpython/issues/105481]: The regen-opcode build stage was removed and its work is now done in regen-cases.

- gh-107901 [https://github.com/python/cpython/issues/107901]: Fix missing line number on `JUMP_BACKWARD` at the end of a for loop.

- gh-108113 [https://github.com/python/cpython/issues/108113]: The `compile()` built-in can now accept a new flag, `ast.PyCF_OPTIMIZED_AST`, which is similar to `ast.PyCF_ONLY_AST` except that the returned `AST` is optimized according to the value of the `optimize` argument.

  `ast.parse()` now accepts an optional argument `optimize` which is passed on to the `compile()` built-in. This makes it possible to obtain an optimized `AST`.

- gh-107971 [https://github.com/python/cpython/issues/107971]: Opcode IDs are generated from bytecodes.c instead of being hard coded in opcode.py.

- gh-107944 [https://github.com/python/cpython/issues/107944]: Improve error message for function calls with bad keyword arguments. Patch by Pablo Galindo

- gh-108390 [https://github.com/python/cpython/issues/108390]: Raise an exception when setting a non-local event (`RAISE`, `EXCEPTION_HANDLED`, etc.) in `sys.monitoring.set_local_events`.

  Fixes crash when tracing in recursive calls to Python classes.

- gh-108035 [https://github.com/python/cpython/issues/108035]: Remove the `_PyCFrame` struct, moving the pointer to the current interpreter frame back to the threadstate, as it was for 3.10 and earlier. The `_PyCFrame` existed as a performance optimization for tracing. Since PEP 669 has been implemented, this optimization no longer applies.

- gh-91051 [https://github.com/python/cpython/issues/91051]: Fix abort / segfault when using all eight type watcher slots, on platforms where `char` is signed by default.
- gh-106581 [https://github.com/python/cpython/issues/106581]: Fix possible assertion failures and missing instrumentation events when **`PYTHONUOPS`** or **`-X uops`** is enabled.
- gh-107526 [https://github.com/python/cpython/issues/107526]: Revert converting `vars`, `dir`, `next`, `getattr`, and `iter` to argument clinic.
- gh-84805 [https://github.com/python/cpython/issues/84805]: Autogenerate signature for **`METH_NOARGS`** and **`METH_O`** extension functions.
- gh-107758 [https://github.com/python/cpython/issues/107758]: Make the `dump_stack()` routine used by the `lltrace` feature (low-level interpreter debugging) robust against recursion by ensuring that it never calls a `__repr__` method implemented in Python. Also make the similar output for Tier-2 uops appear on `stdout` (instead of `stderr`), to match the `lltrace` code in ceval.c.
- gh-107659 [https://github.com/python/cpython/issues/107659]: Add docstrings for **`ctypes.pointer()`** and **`ctypes.POINTER()`**.
- gh-105848 [https://github.com/python/cpython/issues/105848]: Modify the bytecode so that the actual callable for a **`CALL`** is at a consistent position on the stack (regardless of whether or not bound-method-calling optimizations are active).
- gh-107674 [https://github.com/python/cpython/issues/107674]: Fixed performance regression in `sys.settrace`.

- gh-107724 [https://github.com/python/cpython/issues/107724]: In pre-release versions of 3.12, up to rc1, the sys.monitoring callback function for the `PY_THROW` event was missing the third, exception argument. That is now fixed.

- gh-84436 [https://github.com/python/cpython/issues/84436]: Skip reference count modifications for many known immortal objects.

- gh-107596 [https://github.com/python/cpython/issues/107596]: Specialize subscripting **`str`** objects by **`int`** indexes.

- gh-107080 [https://github.com/python/cpython/issues/107080]: Trace refs builds (`--with-trace-refs`) were crashing when used with isolated subinterpreters. The problematic global state has been isolated to each interpreter. Other fixing the crashes, this change does not affect users.

- gh-107557 [https://github.com/python/cpython/issues/107557]: Generate the cases needed for the barebones tier 2 abstract interpreter for optimization passes in CPython.

- gh-106608 [https://github.com/python/cpython/issues/106608]: Make `_PyUOpExecutorObject` variable length.

- gh-100964 [https://github.com/python/cpython/issues/100964]: Clear generators’ exception state after `return` to break reference cycles.

- gh-107455 [https://github.com/python/cpython/issues/107455]: Improve error messages when converting an incompatible type to **`ctypes.c_char_p`**, **`ctypes.c_wchar_p`** and **`ctypes.c_void_p`**.

- gh-107263 [https://github.com/python/cpython/issues/107263]: Increase C recursion limit for functions other than the main interpreter from 800 to 1500. This should allow functions like `list.__repr__` and `json.dumps` to handle all the inputs that they could prior to 3.12

- gh-104584 [https://github.com/python/cpython/issues/104584]: Fix an issue which caused incorrect inline caches to be read when running with **`PYTHONUOPS`** or **`-X uops`** enabled.

- gh-104432 [https://github.com/python/cpython/issues/104432]: Fix potential unaligned memory access on C APIs involving returned sequences of `char *` pointers within the **`grp`** and **`socket`** modules. These were revealed using a `-fsaniziter=alignment` build on ARM macOS. Patch by Christopher Chavez.

- gh-106078 [https://github.com/python/cpython/issues/106078]: Isolate **`_decimal`** (apply **PEP 687** [https://peps.python.org/pep-0687/]). Patch by Charlie Zhao.

- gh-106898 [https://github.com/python/cpython/issues/106898]: Add the exception as the third argument to `PY_UNIND` callbacks in `sys.monitoring`. This makes the `PY_UNWIND` callback consistent with the other exception handling callbacks.

- gh-106895 [https://github.com/python/cpython/issues/106895]: Raise a `ValueError` when a monitoring callback function returns `DISABLE` for events that cannot be disabled locally.

- gh-106897 [https://github.com/python/cpython/issues/106897]: Add a `RERAISE` event to `sys.monitoring`, which occurs when an exception is reraise, either explicitly by a plain `raise` statement, or implicitly in an `except` or `finally` block.

- gh-77377 [https://github.com/python/cpython/issues/77377]: Ensure that multiprocessing synchronization objects created in a fork context are not sent to a different process created in a spawn context. This changes a segfault into an actionable RuntimeError in the parent process.

- gh-106931 [https://github.com/python/cpython/issues/106931]: Statically allocated string objects are now interned globally instead of per-interpreter. This fixes a situation where such a string would only be interned in a single interpreter. Normal string objects are unaffected.

- gh-104621 [https://github.com/python/cpython/issues/104621]: Unsupported modules now always fail to be imported.

- gh-107122 [https://github.com/python/cpython/issues/107122]: Add **`dbm.ndbm.ndbm.clear()`** to `dbm.ndbm`. Patch By Donghee Na.

- gh-107122 [https://github.com/python/cpython/issues/107122]: Add **`dbm.gnu.gdbm.clear()`** to `dbm.gnu`. Patch By Donghee Na.

- gh-107015 [https://github.com/python/cpython/issues/107015]: The ASYNC and AWAIT tokens are removed from the Grammar, which removes the possibility of making `async` and `await` soft keywords when using `feature_version<7` in `ast.parse()`.

- gh-106917 [https://github.com/python/cpython/issues/106917]: Fix classmethod-style `super()` method calls (i.e., where the second argument to `super()`, or the implied second argument drawn from `self/cls` in the case of zero-arg super, is a type) when the target of the call is not a classmethod.

- gh-105699 [https://github.com/python/cpython/issues/105699]: Python no longer crashes due an infrequent race when initializing per-interpreter interned strings. The crash would manifest when the interpreter was finalized.

- gh-105699 [https://github.com/python/cpython/issues/105699]: Python no longer crashes due to an infrequent race in setting `Py_FileSystemDefaultEncoding` and `Py_FileSystemDefaultEncodeErrors` (both deprecated), when simultaneously initializing two isolated subinterpreters. Now they are only set during runtime initialization.

- gh-106908 [https://github.com/python/cpython/issues/106908]: Fix various hangs, reference leaks, test failures, and tracing/introspection bugs when running with **`PYTHONUOPS`** or **`-X uops`** enabled.

- gh-106092 [https://github.com/python/cpython/issues/106092]: Fix a segmentation fault caused by a use-after-free bug in `frame_dealloc` when the trashcan delays the deallocation of a `PyFrameObject`.

- gh-106485 [https://github.com/python/cpython/issues/106485]: Reduce the number of materialized instances dictionaries by dematerializing them when possible.

- gh-106719 [https://github.com/python/cpython/issues/106719]: No longer suppress arbitrary errors in the `__annotations__` getter and setter in the type and module types.

- gh-106723 [https://github.com/python/cpython/issues/106723]: Propagate `frozen_modules` to multiprocessing spawned process interpreters.

- gh-104909 [https://github.com/python/cpython/issues/104909]: Split **`LOAD_ATTR_INSTANCE_VALUE`** into micro-ops.

- gh-104909 [https://github.com/python/cpython/issues/104909]: Split **`LOAD_GLOBAL`** specializations into micro-ops.

- gh-106597 [https://github.com/python/cpython/issues/106597]: A new debug structure of offsets has been added to the

`_PyRuntimeState` that will help out-of-process debuggers and profilers to obtain the offsets to relevant interpreter structures in a way that is agnostic of how Python was compiled and that doesn't require copying the headers. Patch by Pablo Galindo

- gh-106487 [https://github.com/python/cpython/issues/106487]: Allow the *count* argument of `str.replace()` to be a keyword. Patch by Hugo van Kemenade.

- gh-96844 [https://github.com/python/cpython/issues/96844]: Improve error message of `list.remove()`. Patch by Donghee Na.

- gh-81283 [https://github.com/python/cpython/issues/81283]: Compiler now strips indents from docstrings. It reduces `pyc` file size 5% when the module is heavily documented. This change affects to `__doc__` so tools like doctest will be affected.

- gh-106396 [https://github.com/python/cpython/issues/106396]: When the format specification of an f-string expression is empty, the parser now generates an empty `ast.JoinedStr` node for it instead of an one-element `ast.JoinedStr` with an empty string `ast.Constant`.

- gh-100288 [https://github.com/python/cpython/issues/100288]: Specialize `LOAD_ATTR` for non-descriptors on the class. Adds **`LOAD_ATTR_NONDESCRIPTOR_WITH_VALUES`** and **`LOAD_ATTR_NONDESCRIPTOR_NO_DICT`**.

- gh-106008 [https://github.com/python/cpython/issues/106008]: Fix possible reference leaks when failing to optimize comparisons with `None` in the bytecode compiler.

- gh-106145 [https://github.com/python/cpython/issues/106145]: Make `end_lineno` and `end_col_offset` required on `type_param` ast nodes.

- gh-106213 [https://github.com/python/cpython/issues/106213]: Changed the way that Emscripten call trampolines work for compatibility with Wasm/JS Promise integration.

- gh-106182 [https://github.com/python/cpython/issues/106182]: `sys.getfilesystemencoding()` and `sys.getfilesystemencodeerrors` now return interned Unicode object.

- gh-106210 [https://github.com/python/cpython/issues/106210]: Removed Emscripten import trampoline as it was no longer necessary for Pyodide.

- gh-104584 [https://github.com/python/cpython/issues/104584]: Added a new, experimental, tracing optimizer and interpreter (a.k.a. “tier 2”). This currently pessimizes, so don’t use yet – this is infrastructure so we can experiment with optimizing passes. To enable it, pass `-Xuops` or set `PYTHONUOPS=1`. To get debug output, set `PYTHONUOPSDEBUG=N` where `N` is a debug level (0-4, where 0 is no debug output and 4 is excessively verbose).

- gh-105775 [https://github.com/python/cpython/issues/105775]: `LOAD_CLOSURE` is now a pseudo-op.

- gh-105730 [https://github.com/python/cpython/issues/105730]: Allow any callable other than type objects as the condition predicate in `BaseExceptionGroup.split()` and `BaseExceptionGroup.subgroup()`.

- gh-105979 [https://github.com/python/cpython/issues/105979]: Fix crash in `_imp.get_frozen_object()` due to improper exception handling.

- gh-106003 [https://github.com/python/cpython/issues/106003]: Add a new **`TO_BOOL`** instruction, which performs boolean conversions for **`POP_JUMP_IF_TRUE`**, **`POP_JUMP_IF_FALSE`**, and **`UNARY_NOT`** (which all expect exact **`bool`** values now). Also, modify the oparg of **`COMPARE_OP`** to include an optional “boolean conversion” flag.

- gh-98931 [https://github.com/python/cpython/issues/98931]: Ensure custom **`SyntaxError`** error messages are raised for invalid imports with multiple targets. Patch by Pablo Galindo

- gh-105724 [https://github.com/python/cpython/issues/105724]: Improve `assert` error messages by providing exact error range.

- gh-105908 [https://github.com/python/cpython/issues/105908]: Fixed bug where gh-99111 [https://github.com/python/cpython/issues/99111] breaks future import `barry_as_FLUFL` in the Python REPL.

- gh-105840 [https://github.com/python/cpython/issues/105840]: Fix possible crashes when specializing function calls with too many `__defaults__`.

- gh-105831 [https://github.com/python/cpython/issues/105831]: Fix an f-string bug, where using a debug expression (the `=` sign) that appears in the last line of a file results to the debug buffer that holds the expression text being one character too small.

- gh-105800 [https://github.com/python/cpython/issues/105800]: Correctly issue **`SyntaxWarning`** in f-strings if invalid sequences are used. Patch by Pablo Galindo

- gh-105340 [https://github.com/python/cpython/issues/105340]: Include the comprehension iteration variable in `locals()` inside a module- or class-scope comprehension.

- gh-105331 [https://github.com/python/cpython/issues/105331]: Raise `ValueError` if the `delay` argument to `asyncio.sleep()` is a NaN (matching `time.sleep()`).

- gh-105587 [https://github.com/python/cpython/issues/105587]: The runtime can’t guarantee that immortal objects will not be mutated by Extensions. Thus, this modifies _PyStaticObject_CheckRefcnt to warn instead of asserting.

- gh-105564 [https://github.com/python/cpython/issues/105564]: Don’t include artificil newlines in the `line` attribute of tokens in the APIs of the `tokenize` module. Patch by Pablo Galindo

- gh-105549 [https://github.com/python/cpython/issues/105549]: Tokenize separately `NUMBER` and `NAME` tokens that are not ambiguous. Patch by Pablo Galindo.

- gh-105588 [https://github.com/python/cpython/issues/105588]: Fix an issue that could result in crashes when compiling malformed `ast` nodes.

- gh-100987 [https://github.com/python/cpython/issues/100987]: Allow objects other than code objects as the “executable” in internal frames. In the long term, this can help tools like Cython and PySpy interact more efficiently. In the shorter term, it allows us to perform some optimizations more simply.

- gh-105375 [https://github.com/python/cpython/issues/105375]: Fix bugs in the `builtins` module where exceptions could end up being overwritten.

- gh-105375 [https://github.com/python/cpython/issues/105375]: Fix bug in the compiler where an exception could end up being overwritten.

- gh-105375 [https://github.com/python/cpython/issues/105375]: Improve error handling in `PyUnicode_BuildEncodingMap()` where an exception could end up being overwritten.

- gh-105486 [https://github.com/python/cpython/issues/105486]: Change the repr of `ParamSpec` list of args in `types.GenericAlias`.

- gh-105678 [https://github.com/python/cpython/issues/105678]: Break the `MAKE_FUNCTION` instruction into two parts, `MAKE_FUNCTION` which makes the function and `SET_FUNCTION_ATTRIBUTE` which sets the attributes on the function. This makes the stack effect of `MAKE_FUNCTION` regular to ease optimization and code generation.

- gh-105435 [https://github.com/python/cpython/issues/105435]: Fix spurious newline character if file ends on a comment without a newline. Patch by Pablo Galindo

- gh-105390 [https://github.com/python/cpython/issues/105390]: Correctly raise `tokenize.TokenError` exceptions instead of `SyntaxError` for tokenize errors such as incomplete input. Patch by Pablo Galindo

- gh-105259 [https://github.com/python/cpython/issues/105259]: Don’t include newline character for trailing `NEWLINE` tokens emitted in the `tokenize` module. Patch by Pablo Galindo

- gh-104635 [https://github.com/python/cpython/issues/104635]: Eliminate redundant `STORE_FAST` instructions in the compiler. Patch by Donghee Na and Carl Meyer.

- gh-105324 [https://github.com/python/cpython/issues/105324]: Fix the main function of the `tokenize` module when reading from `sys.stdin`. Patch by Pablo Galindo

- gh-33092 [https://github.com/python/cpython/issues/33092]: Simplify and speed up interpreter for f-strings. Removes `FORMAT_VALUE` opcode. Add `CONVERT_VALUE`, `FORMAT_SIMPLE` and `FORMAT_WITH_SPEC` opcode. Compiler emits more efficient sequence for each format expression.

- gh-105229 [https://github.com/python/cpython/issues/105229]: Remove remaining two-codeunit superinstructions. All remaining superinstructions only take a single codeunit, simplifying instrumentation and quickening.

- gh-105235 [https://github.com/python/cpython/issues/105235]: Prevent out-of-bounds memory access during `mmap.find()` calls.

- gh-98963 [https://github.com/python/cpython/issues/98963]: Restore the ability for a subclass of **`property`** to define `__slots__` or otherwise be dict-less by ignoring failures to set a docstring on such a class. This behavior had regressed in 3.12beta1. An **`AttributeError`** where there had not previously been one was disruptive to existing code.

- gh-104812 [https://github.com/python/cpython/issues/104812]: The “pending call” machinery now works for all interpreters, not just the main interpreter, and runs in all threads, not just the main thread. Some calls are still only done in the main thread, ergo in the main interpreter. This change does not affect signal handling nor the existing public C-API (`Py_AddPendingCall()`), which both still only target the main thread. The new functionality is meant strictly for internal use for now, since consequences of its use are not well understood yet outside some very restricted cases. This change brings the capability in line with the intention when the state was made per-interpreter several years ago.

- gh-105194 [https://github.com/python/cpython/issues/105194]: Do not escape with backslashes f-string format specifiers. Patch by Pablo Galindo

- gh-105229 [https://github.com/python/cpython/issues/105229]: Replace some dynamic superinstructions with single instruction equivalents.

- gh-105162 [https://github.com/python/cpython/issues/105162]: Fixed bug in generator.close()/throw() where an inner iterator would be ignored when the outer iterator was instrumented.

- gh-105164 [https://github.com/python/cpython/issues/105164]: Ensure annotations are set up correctly if the only annotation in a block is within a **`match`** block. Patch by Jelle Zijlstra.

- gh-105148 [https://github.com/python/cpython/issues/105148]: Make `_PyASTOptimizeState` internal to ast_opt.c. Make `_PyAST_Optimize` take two integers instead of a pointer to this struct. This avoids the need to include pycore_compile.h in ast_opt.c.

- gh-104799 [https://github.com/python/cpython/issues/104799]: Attributes of **`ast`** nodes that are lists now default to the empty list if omitted. This means that some code that previously raised **`TypeError`** when the AST node was used will now proceed with the empty list instead. Patch by Jelle Zijlstra.

- gh-105111 [https://github.com/python/cpython/issues/105111]: Remove the old trashcan macros `Py_TRASHCAN_SAFE_BEGIN` and `Py_TRASHCAN_SAFE_END`. They should be replaced by the new macros `Py_TRASHCAN_BEGIN` and `Py_TRASHCAN_END`.

- gh-105035 [https://github.com/python/cpython/issues/105035]: Fix `super()` calls on types with custom `tp_getattro` implementation (e.g. meta-types.)

- gh-105017 [https://github.com/python/cpython/issues/105017]: Show CRLF lines in the tokenize string attribute in both NL and NEWLINE tokens. Patch by Marta Gómez.

- gh-105013 [https://github.com/python/cpython/issues/105013]: Fix handling of multiline parenthesized lambdas in `inspect.getsource()`. Patch by Pablo Galindo

- gh-105017 [https://github.com/python/cpython/issues/105017]: Do not include an additional final `NL` token when parsing files having CRLF lines. Patch by Marta Gómez.

- gh-104976 [https://github.com/python/cpython/issues/104976]: Ensure that trailing `DEDENT` **`tokenize.TokenInfo`** objects emitted by the `tokenize` module are reported as in Python 3.11. Patch by Pablo Galindo

- gh-104972 [https://github.com/python/cpython/issues/104972]: Ensure that the `line` attribute in **`tokenize.TokenInfo`** objects in the `tokenize` module are always correct. Patch by Pablo Galindo

- gh-104955 [https://github.com/python/cpython/issues/104955]: Fix signature for the new `__release_buffer__()` slot. Patch by Jelle Zijlstra.

- gh-104690 [https://github.com/python/cpython/issues/104690]: Starting new threads and process creation through `os.fork()` during interpreter shutdown (such as from `atexit` handlers) is no longer supported. It can lead to race condition between the main Python runtime thread freeing thread states while internal `threading` routines are trying to allocate and use the state of just created threads. Or forked children trying to use the mid-shutdown runtime and thread state in the child process.

- gh-104879 [https://github.com/python/cpython/issues/104879]: Fix crash when accessing the `__module__` attribute of type aliases defined outside a module. Patch by Jelle Zijlstra.
- gh-104825 [https://github.com/python/cpython/issues/104825]: Tokens emitted by the **`tokenize`** module do not include an implicit `\n` character in the `line` attribute anymore. Patch by Pablo Galindo
- gh-104770 [https://github.com/python/cpython/issues/104770]: If a generator returns a value upon being closed, the value is now returned by **`generator.close()`**.
- gh-89091 [https://github.com/python/cpython/issues/89091]: Raise **`RuntimeWarning`** for unawaited async generator methods like **`asend()`**, **`athrow()`** and **`aclose()`**. Patch by Kumar Aditya.
- gh-96663 [https://github.com/python/cpython/issues/96663]: Add a better, more introspect-able error message when setting attributes on classes without a `__dict__` and no slot member for the attribute.
- gh-93627 [https://github.com/python/cpython/issues/93627]: Update the Python pickle module implementation to match the C implementation of the pickle module. For objects setting reduction methods like **`__reduce_ex__()`** or **`__reduce__()`** to `None`, pickling will result in a **`TypeError`**.
- gh-101006 [https://github.com/python/cpython/issues/101006]: Improve error handling when read **`marshal`** data.
- gh-91095 [https://github.com/python/cpython/issues/91095]: Specializes calls to most Python classes. Specifically,

any class that inherits from `object`, or another Python class, and does not override `__new__`.

The specialized instruction does the following:

1. Creates the object (by calling `object.__new__`)
2. Pushes a shim frame to the frame stack (to cleanup after `__init__`)
3. Pushes the frame for `__init__` to the frame stack

Speeds up the instantiation of most Python classes.

Library

- gh-110786 [https://github.com/python/cpython/issues/110786]: **`sysconfig`**'s CLI now ignores **`BrokenPipeError`**, making it exit normally if its output is being piped and the pipe closes.

- gh-103480 [https://github.com/python/cpython/issues/103480]: The **`sysconfig`** module is now a package, instead of a single-file module.

- gh-110733 [https://github.com/python/cpython/issues/110733]: Micro-optimization: Avoid calling `min()`, `max()` in **`BaseEventLoop._run_once()`**.

- gh-94597 [https://github.com/python/cpython/issues/94597]: Added **`asyncio.EventLoop`** for use with the **`asyncio.run()`** *loop_factory* kwarg to avoid calling the asyncio policy system.

- gh-110682 [https://github.com/python/cpython/issues/110682]: **`runtime-checkable protocols`** used to consider `__match_args__` a protocol member in `__instancecheck__` if it was present on the protocol. Now, this attribute is ignored if it is present.

- gh-110488 [https://github.com/python/cpython/issues/110488]: Fix a couple of issues in `pathlib.PurePath.with_name()`: a single dot was incorrectly considered a valid name, and in **`PureWindowsPath`**, a name with an NTFS alternate data stream, like `a:b`, was incorrectly considered invalid.

- gh-110590 [https://github.com/python/cpython/issues/110590]: Fix a bug in **`_sre.compile()`** where `TypeError` would be overwritten by `OverflowError` when the *code* argument was a list of non-ints.

- gh-65052 [https://github.com/python/cpython/issues/65052]: Prevent `pdb` from crashing when trying to display undisplayable objects

- gh-110519 [https://github.com/python/cpython/issues/110519]: Deprecation warning about non-integer number in `gettext` now always refers to the line in the user code where gettext function or method is used. Previously it could refer to a line in `gettext` code.

- gh-89902 [https://github.com/python/cpython/issues/89902]: Deprecate non-standard format specifier "N" for `decimal.Decimal`. It was not documented and only supported in the C implementation.

- gh-110378 [https://github.com/python/cpython/issues/110378]: `contextmanager()` and `asynccontextmanager()` context managers now close an invalid underlying generator object that yields more then one value.

- gh-106670 [https://github.com/python/cpython/issues/106670]: In `pdb`, set convenience variable `$_exception` for post mortem debugging.

- gh-110365 [https://github.com/python/cpython/issues/110365]: Fix `termios.tcsetattr()` bug that was overwriting existing errors during parsing integers from `term` list.

- gh-109653 [https://github.com/python/cpython/issues/109653]: Slightly improve the import time of several standard-library modules by deferring imports of **`warnings`** within those modules. Patch by Alex Waygood.
- gh-110273 [https://github.com/python/cpython/issues/110273]: **`dataclasses.replace()`** now raises TypeError instead of ValueError if specify keyword argument for a field declared with init=False or miss keyword argument for required InitVar field.
- gh-110249 [https://github.com/python/cpython/issues/110249]: Add `--inline-caches` flag to `dis` command line.
- gh-109653 [https://github.com/python/cpython/issues/109653]: Fix a Python 3.12 regression in the import time of **`random`**. Patch by Alex Waygood.
- gh-110222 [https://github.com/python/cpython/issues/110222]: Add support of struct sequence objects in **`copy.replace()`**. Patched by Xuehai Pan.
- gh-109649 [https://github.com/python/cpython/issues/109649]: **`multiprocessing`**, **`concurrent.futures`**, **`compileall`**: Replace **`os.cpu_count()`** with **`os.process_cpu_count()`** to select the default number of worker threads and processes. Get the CPU affinity if supported. Patch by Victor Stinner.
- gh-110150 [https://github.com/python/cpython/issues/110150]: Fix base case handling in statistics.quantiles. Now allows a single data point.
- gh-110036 [https://github.com/python/cpython/issues/110036]: On Windows, multiprocessing `Popen.terminate()` now catches **`PermissionError`** and get the process exit code. If the

process is still running, raise again the `PermissionError`. Otherwise, the process terminated as expected: store its exit code. Patch by Victor Stinner.

- gh-110038 [https://github.com/python/cpython/issues/110038]: Fixed an issue that caused **`KqueueSelector.select()`** to not return all the ready events in some cases when a file descriptor is registered for both read and write.

- gh-110045 [https://github.com/python/cpython/issues/110045]: Update the `symtable` module to support the new scopes introduced by **PEP 695** [https://peps.python.org/pep-0695/].

- gh-88402 [https://github.com/python/cpython/issues/88402]: Add new variables to `sysconfig.get_config_vars()` on Windows: `LIBRARY`, `LDLIBRARY`, `LIBDIR`, `SOABI`, and `Py_NOGIL`.

- gh-109631 [https://github.com/python/cpython/issues/109631]: `re` functions such as `re.findall()`, `re.split()`, `re.search()` and `re.sub()` which perform short repeated matches can now be interrupted by user.

- gh-109653 [https://github.com/python/cpython/issues/109653]: Reduce the import time of `email.utils` by around 43%. This results in the import time of `email.message` falling by around 18%, which in turn reduces the import time of `importlib.metadata` by around 6%. Patch by Alex Waygood.

- gh-109818 [https://github.com/python/cpython/issues/109818]: Fix `reprlib.recursive_repr()` not copying `__type_params__` from decorated function.

- gh-109047 [https://github.com/python/cpython/issues/109047]: `concurrent.futures`: The *executor manager thread* now catches exceptions when adding an item to the *call queue*. During Python finalization, creating a new thread can now raise `RuntimeError`. Catch the exception and call `terminate_broken()` in this case. Patch by Victor Stinner.

- gh-109782 [https://github.com/python/cpython/issues/109782]: Ensure the signature of `os.path.isdir()` is identical on all platforms. Patch by Amin Alaee.
- gh-109653 [https://github.com/python/cpython/issues/109653]: Improve import time of `functools` by around 13%. Patch by Alex Waygood.
- gh-109590 [https://github.com/python/cpython/issues/109590]: `shutil.which()` will prefer files with an extension in `PATHEXT` if the given mode includes `os.X_OK` on win32. If no `PATHEXT` match is found, a file without an extension in `PATHEXT` can be returned. This change will have `shutil.which()` act more similarly to previous behavior in Python 3.11.
- gh-109653 [https://github.com/python/cpython/issues/109653]: Reduce the import time of `enum` by over 50%. Patch by Alex Waygood.
- gh-109593 [https://github.com/python/cpython/issues/109593]: Avoid deadlocking on a reentrant call to the multiprocessing resource tracker. Such a reentrant call, though unlikely, can happen if a GC pass invokes the finalizer for a multiprocessing object such as SemLock.
- gh-109653 [https://github.com/python/cpython/issues/109653]: Reduce the import time of `typing` by around a third. Patch by Alex Waygood.
- gh-109649 [https://github.com/python/cpython/issues/109649]: Add `os.process_cpu_count()` function to get the number of logical CPUs usable by the calling thread of the current process. Patch by Victor Stinner.

- gh-74481 [https://github.com/python/cpython/issues/74481]: Add `set_error_mode` related constants in `msvcrt` module in Python debug build.

- gh-109613 [https://github.com/python/cpython/issues/109613]: Fix **`os.stat()`** and **`os.DirEntry.stat()`**: check for exceptions. Previously, on Python built in debug mode, these functions could trigger a fatal Python error (and abort the process) when a function succeeded with an exception set. Patch by Victor Stinner.

- gh-109599 [https://github.com/python/cpython/issues/109599]: Expose the type of PyCapsule objects as `types.CapsuleType`.

- gh-109109 [https://github.com/python/cpython/issues/109109]: You can now get the raw TLS certificate chains from TLS connections via **`ssl.SSLSocket.get_verified_chain()`** and **`ssl.SSLSocket.get_unverified_chain()`** methods.

  Contributed by Mateusz Nowak.

- gh-109559 [https://github.com/python/cpython/issues/109559]: Update **`unicodedata`** database to Unicode 15.1.0.

- gh-109543 [https://github.com/python/cpython/issues/109543]: Remove unnecessary **`hasattr()`** check during **`typing.TypedDict`** creation.

- gh-109495 [https://github.com/python/cpython/issues/109495]: Remove unnecessary extra `__slots__` in **`datetime`**'s pure python implementation to reduce memory size, as they are defined in the superclass. Patch by James Hilton-Balfe

- gh-109461 [https://github.com/python/cpython/issues/109461]: **`logging`**: Use a context manager for lock acquisition.

- gh-109096 [https://github.com/python/cpython/issues/109096]: **`http.server.CGIHTTPRequestHandler`** has been deprecated for removal in 3.15. Its design is old and the web world has long since moved beyond CGI.

- gh-109409 [https://github.com/python/cpython/issues/109409]: Fix error when it was possible to inherit a frozen dataclass from multiple parents some of which were possibly not frozen.

- gh-109375 [https://github.com/python/cpython/issues/109375]: The **`pdb`** `alias` command now prevents registering aliases without arguments.

- gh-109319 [https://github.com/python/cpython/issues/109319]: Deprecate the `dis.HAVE_ARGUMENT` field in favour of `dis.hasarg`.

- gh-107219 [https://github.com/python/cpython/issues/107219]: Fix a race condition in `concurrent.futures`. When a process in the process pool was terminated abruptly (while the future was running or pending), close the connection write end. If the call queue is blocked on sending bytes to a worker process, closing the connection write end interrupts the send, so the queue can be closed. Patch by Victor Stinner.

- gh-66143 [https://github.com/python/cpython/issues/66143]: The **`codecs.CodecInfo`** object has been made copyable and pickleable. Patched by Robert Lehmann and Furkan Onder.

- gh-109187 [https://github.com/python/cpython/issues/109187]: **`pathlib.Path.resolve()`** now treats symlink loops like other errors: in strict mode, **`OSError`** is raised, and in non-strict mode, no exception is raised.

- gh-50644 [https://github.com/python/cpython/issues/50644]: Attempts to pickle or create a shallow or deep copy of **`codecs`** streams now raise a TypeError. Previously, copying failed with a RecursionError, while pickling produced wrong results that eventually caused unpickling to fail with a RecursionError.

- gh-109174 [https://github.com/python/cpython/issues/109174]: Add support of **`types.SimpleNamespace`** in **`copy.replace()`**.

- gh-109164 [https://github.com/python/cpython/issues/109164]: **`pdb`**: Replace **`getopt`** with **`argparse`** for parsing command line arguments.

- gh-109151 [https://github.com/python/cpython/issues/109151]: Enable `readline` editing features in the sqlite3 command-line interface (`python -m sqlite3`).

- gh-108987 [https://github.com/python/cpython/issues/108987]: Fix **`_thread.start_new_thread()`** race condition. If a thread is created during Python finalization, the newly spawned thread now exits immediately instead of trying to access freed memory and lead to a crash. Patch by Victor Stinner.

- gh-108682 [https://github.com/python/cpython/issues/108682]: Enum: require `names=()` or `type=...` to create an empty enum using the functional syntax.

- gh-109033 [https://github.com/python/cpython/issues/109033]: Exceptions raised by os.utime builtin function now include the related filename

- gh-108843 [https://github.com/python/cpython/issues/108843]: Fix an issue in **`ast.unparse()`** when unparsing f-strings containing many quote types.

- gh-108469 [https://github.com/python/cpython/issues/108469]: `ast.unparse()` now supports new f-string syntax introduced in Python 3.12. Note that the f-string quotes are reselected for simplicity under the new syntax. (Patch by Steven Sun)

- gh-108751 [https://github.com/python/cpython/issues/108751]: Add `copy.replace()` function which allows to create a modified copy of an object. It supports named tuples, dataclasses, and many other objects.

- gh-108682 [https://github.com/python/cpython/issues/108682]: Enum: raise `TypeError` if `super().__new__()` is called from a custom `__new__`.

- gh-108278 [https://github.com/python/cpython/issues/108278]: Deprecate passing the callback callable by keyword for the following `sqlite3.Connection` APIs:

  - `set_authorizer()`
  - `set_progress_handler()`
  - `set_trace_callback()`

  The affected parameters will become positional-only in Python 3.15.

  Patch by Erlend E. Aasland.

- gh-105829 [https://github.com/python/cpython/issues/105829]: Fix concurrent.futures.ProcessPoolExecutor deadlock

- gh-108295 [https://github.com/python/cpython/issues/108295]: Fix crashes related to use of weakrefs on `typing.TypeVar`.

- gh-108463 [https://github.com/python/cpython/issues/108463]: Make expressions/statements work as expected in pdb

- gh-108277 [https://github.com/python/cpython/issues/108277]: Add `os.timerfd_create()`, `os.timerfd_settime()`, `os.timerfd_gettime()`, `os.timerfd_settime_ns()`, and `os.timerfd_gettime_ns()` to provide a low level interface for Linux’s timer notification file descriptor.

- gh-107811 [https://github.com/python/cpython/issues/107811]: `tarfile`: extraction of members with overly large UID or GID (e.g. on an OS with 32-bit `id_t`) now fails in the same way as failing to set the ID.

- gh-64662 [https://github.com/python/cpython/issues/64662]: Fix support for virtual tables in `sqlite3.Connection.iterdump()`. Patch by Aviv Palivoda.

- gh-108111 [https://github.com/python/cpython/issues/108111]: Fix a regression introduced in gh-101251 [https://github.com/python/cpython/issues/101251] for 3.12, resulting in an incorrect offset calculation in `gzip.GzipFile.seek()`.

- gh-108294 [https://github.com/python/cpython/issues/108294]: `time.sleep()` now raises an auditing event.

- gh-108278 [https://github.com/python/cpython/issues/108278]: Deprecate passing name, number of arguments, and the callable as keyword arguments, for the following `sqlite3.Connection` APIs:

  - `create_function()`
  - `create_aggregate()`

  The affected parameters will become positional-only in Python 3.15.

  Patch by Erlend E. Aasland.

- gh-108322 [https://github.com/python/cpython/issues/108322]: Speed-up NormalDist.samples() by using the inverse CDF method instead of calling random.gauss().

- gh-83417 [https://github.com/python/cpython/issues/83417]: Add the ability for venv to create a `.gitignore` file which causes the created environment to be ignored by Git. It is on by default when venv is called via its CLI.

- gh-105736 [https://github.com/python/cpython/issues/105736]: Harmonized the pure Python version of **`OrderedDict`** with the C version. Now, both versions set up their internal state in `__new__`. Formerly, the pure Python version did the set up in `__init__`.

- gh-108083 [https://github.com/python/cpython/issues/108083]: Fix bugs in the constructor of **`sqlite3.Connection`** and **`sqlite3.Connection.close()`** where exceptions could be leaked. Patch by Erlend E. Aasland.

- gh-107932 [https://github.com/python/cpython/issues/107932]: Fix `dis` module to properly report and display bytecode that do not have source lines.

- gh-105539 [https://github.com/python/cpython/issues/105539]: **`sqlite3`** now emits an **`ResourceWarning`** if a **`sqlite3.Connection`** object is not **`closed`** explicitly. Patch by Erlend E. Aasland.

- gh-107995 [https://github.com/python/cpython/issues/107995]: The `__module__` attribute on instances of **`functools.cached_property`** is now set to the name of the module in which the cached_property is defined, rather than “functools”. This means that doctests in `cached_property` docstrings are now properly collected by the **`doctest`** module. Patch by Tyler Smart.

- gh-107963 [https://github.com/python/cpython/issues/107963]: Fix `multiprocessing.set_forkserver_preload()` to check the given list of modules names. Patch by Donghee Na.

- gh-106242 [https://github.com/python/cpython/issues/106242]: Fixes `os.path.normpath()` to handle embedded null characters without truncating the path.

- gh-81555 [https://github.com/python/cpython/issues/81555]: `xml.dom.minidom` now only quotes `"` in attributes.

- gh-50002 [https://github.com/python/cpython/issues/50002]: `xml.dom.minidom` now preserves whitespaces in attributes.

- gh-93057 [https://github.com/python/cpython/issues/93057]: Passing more than one positional argument to `sqlite3.connect()` and the `sqlite3.Connection` constructor is deprecated. The remaining parameters will become keyword-only in Python 3.15. Patch by Erlend E. Aasland.

- gh-76913 [https://github.com/python/cpython/issues/76913]: Add *merge_extra* parameter/feature to `logging.LoggerAdapter`

- gh-107913 [https://github.com/python/cpython/issues/107913]: Fix possible losses of `errno` and `winerror` values in `OSError` exceptions if they were cleared or modified by the cleanup code before creating the exception object.

- gh-107845 [https://github.com/python/cpython/issues/107845]: `tarfile.data_filter()` now takes the location of symlinks into account when determining their target, so it will no longer reject some valid tarballs with `LinkOutsideDestinationError`.

- gh-107812 [https://github.com/python/cpython/issues/107812]: Extend socket’s netlink support to the FreeBSD platform.

- gh-107805 [https://github.com/python/cpython/issues/107805]: Fix signatures of module-level generated functions in **`turtle`**.
- gh-107782 [https://github.com/python/cpython/issues/107782]: **`pydoc`** is now able to show signatures which are not representable in Python, e.g. for `getattr` and `dict.pop`.
- gh-56166 [https://github.com/python/cpython/issues/56166]: Deprecate passing optional arguments *maxsplit*, *count* and *flags* in module-level functions **`re.split()`**, **`re.sub()`** and **`re.subn()`** as positional. They should only be passed by keyword.
- gh-107710 [https://github.com/python/cpython/issues/107710]: Speed up **`logging.getHandlerNames()`**.
- gh-107715 [https://github.com/python/cpython/issues/107715]: Fix **`doctest.DocTestFinder.find()`** in presence of class names with special characters. Patch by Gertjan van Zwieten.
- gh-100814 [https://github.com/python/cpython/issues/100814]: Passing a callable object as an option value to a Tkinter image now raises the expected TclError instead of an AttributeError.
- gh-72684 [https://github.com/python/cpython/issues/72684]: Add **`tkinter`** widget methods: **`tk_busy_hold()`**, **`tk_busy_configure()`**, **`tk_busy_cget()`**, **`tk_busy_forget()`**, **`tk_busy_current()`**, and **`tk_busy_status()`**.
- gh-106684 [https://github.com/python/cpython/issues/106684]: Raise **`ResourceWarning`** when **`asyncio.StreamWriter`** is not closed leading to memory leaks. Patch by Kumar Aditya.
- gh-107465 [https://github.com/python/cpython/issues/107465]: Add **`pathlib.Path.from_uri()`** classmethod.

- gh-107077 [https://github.com/python/cpython/issues/107077]: Seems that in some conditions, OpenSSL will return `SSL_ERROR_SYSCALL` instead of `SSL_ERROR_SSL` when a certification verification has failed, but the error parameters will still contain `ERR_LIB_SSL` and `SSL_R_CERTIFICATE_VERIFY_FAILED`. We are now detecting this situation and raising the appropriate `ssl.SSLCertVerificationError`. Patch by Pablo Galindo

- gh-107576 [https://github.com/python/cpython/issues/107576]: Fix **`types.get_original_bases()`** to only return **`__orig_bases__`** if it is present on `cls` directly. Patch by James Hilton-Balfe.

- gh-105481 [https://github.com/python/cpython/issues/105481]: Remove `opcode.is_pseudo`, `opcode.MIN_PSEUDO_OPCODE` and `opcode.MAX_PSEUDO_OPCODE`, which were added in 3.12, were never documented and were not intended to be used externally.

- gh-105481 [https://github.com/python/cpython/issues/105481]: **`opcode.ENABLE_SPECIALIZATION`** (which was added in 3.12 but never documented or intended for external usage) is moved to **`_opcode.ENABLE_SPECIALIZATION`** where tests can access it.

- gh-107396 [https://github.com/python/cpython/issues/107396]: tarfiles; Fixed use before assignment of self.exception for gzip decompression

- gh-107409 [https://github.com/python/cpython/issues/107409]: Set **`__wrapped__`** attribute in **`reprlib.recursive_repr()`**.

- gh-107406 [https://github.com/python/cpython/issues/107406]: Implement new **`__repr__()`** method for **`struct.Struct`**. Now it returns `Struct(<format repr>)`.

- gh-107369 [https://github.com/python/cpython/issues/107369]: Optimize `textwrap.indent()`. It is ~30% faster for large input. Patch by Inada Naoki.

- gh-78722 [https://github.com/python/cpython/issues/78722]: Fix issue where `pathlib.Path.iterdir()` did not raise `OSError` until iterated.

- gh-105578 [https://github.com/python/cpython/issues/105578]: Deprecate `typing.AnyStr` in favor of the new Type Parameter syntax. See PEP 695.

- gh-62519 [https://github.com/python/cpython/issues/62519]: Make `gettext.pgettext()` search plural definitions when translation is not found.

- gh-107089 [https://github.com/python/cpython/issues/107089]: Shelves opened with `shelve.open()` have a much faster **`clear()`** method. Patch by James Cave.

- gh-82500 [https://github.com/python/cpython/issues/82500]: Fix overflow on 32-bit systems with `asyncio` `os.sendfile()` implementation.

- gh-83006 [https://github.com/python/cpython/issues/83006]: Document behavior of `shutil.disk_usage()` for non-mounted filesystems on Unix.

- gh-65495 [https://github.com/python/cpython/issues/65495]: Use lowercase `mail from` and `rcpt to` in **`smptlib.SMTP`**.

- gh-106186 [https://github.com/python/cpython/issues/106186]: Do not report `MultipartInvariantViolationDefect` defect when the `email.parser.Parser` class is used to parse emails with `headersonly=True`.

- gh-105002 [https://github.com/python/cpython/issues/105002]: Fix invalid result from **`PurePath.relative_to()`** method when attempting to walk a “`..`” segment in *other* with *walk_up* enabled. A **`ValueError`** exception is now raised in this case.

- gh-106739 [https://github.com/python/cpython/issues/106739]: Add the `rtype_cache` to the warning message (as an addition to the type of leaked objects and the number of leaked objects already included in the message) to make debugging leaked objects easier when the multiprocessing resource tracker process finds leaked objects at shutdown. This helps more quickly identify what was leaked and/or why the leaked object was not properly cleaned up.

- gh-106751 [https://github.com/python/cpython/issues/106751]: Optimize **`SelectSelector.select()`** for many iteration case. Patch By Donghee Na.

- gh-106751 [https://github.com/python/cpython/issues/106751]: Optimize **`_PollLikeSelector.select()`** for many iteration case.

- gh-106751 [https://github.com/python/cpython/issues/106751]: Optimize **`KqueueSelector.select()`** for many iteration case. Patch By Donghee Na.

- gh-106831 [https://github.com/python/cpython/issues/106831]: Fix potential missing `NULL` check of `d2i_SSL_SESSION` result in `_ssl.c`.

- gh-105481 [https://github.com/python/cpython/issues/105481]: The various opcode lists in the **`dis`** module are now generated from bytecodes.c instead of explicitly constructed in opcode.py.

- gh-106727 [https://github.com/python/cpython/issues/106727]: Make `inspect.getsource()` smarter for class for same name definitions

- gh-106789 [https://github.com/python/cpython/issues/106789]: Remove import of `pprint` from `sysconfig`.

- gh-105726 [https://github.com/python/cpython/issues/105726]: Added `__slots__` to `contextlib.AbstractContextManager` and `contextlib.AbstractAsyncContextManager` so that child classes can use `__slots__`.

- gh-106774 [https://github.com/python/cpython/issues/106774]: Update the bundled copy of pip to version 23.2.1.

- gh-106751 [https://github.com/python/cpython/issues/106751]: `selectors`: Optimize `EpollSelector.select()` code by moving some code outside of the loop.

- gh-106752 [https://github.com/python/cpython/issues/106752]: Fixed several bugs in zipfile.Path, including: in **`zipfile.Path.match()`**, Windows separators are no longer honored (and never were meant to be); Fixed `name`/`suffix`/`suffixes`/`stem` operations when no filename is present and the Path is not at the root of the zipfile; Reworked glob for performance and more correct matching behavior.

- gh-105293 [https://github.com/python/cpython/issues/105293]: Remove call to `SSL_CTX_set_session_id_context` during client side context creation in the `ssl` module.

- gh-106734 [https://github.com/python/cpython/issues/106734]: Disable tab completion in multiline mode of `pdb`

- gh-105481 [https://github.com/python/cpython/issues/105481]: Expose opcode metadata through **`_opcode`**.

- gh-106670 [https://github.com/python/cpython/issues/106670]: Add the new `exceptions` command to the Pdb debugger. It makes it possible to move between chained exceptions when using post mortem debugging.

- gh-106602 [https://github.com/python/cpython/issues/106602]: Add __copy__ and __deepcopy__ in **`enum`**

- gh-106664 [https://github.com/python/cpython/issues/106664]: **`selectors`**: Add `_SelectorMapping.get()` method and optimize `_SelectorMapping.__getitem__()`.

- gh-106628 [https://github.com/python/cpython/issues/106628]: Speed up parsing of emails by about 20% by not compiling a new regular expression for every single email.

- gh-89427 [https://github.com/python/cpython/issues/89427]: Set the environment variable `VIRTUAL_ENV_PROMPT` at **`venv`** activation, even when `VIRTUAL_ENV_DISABLE_PROMPT` is set.

- gh-106530 [https://github.com/python/cpython/issues/106530]: Revert a change to **`colorsys.rgb_to_hls()`** that caused division by zero for certain almost-white inputs. Patch by Terry Jan Reedy.

- gh-106584 [https://github.com/python/cpython/issues/106584]: Fix exit code for `unittest` if all tests are skipped. Patch by Egor Eliseev.

- gh-106566 [https://github.com/python/cpython/issues/106566]: Optimize `(?!)` (pattern which always fails) in regular expressions.

- gh-106554 [https://github.com/python/cpython/issues/106554]: **`selectors`**: Reduce Selector overhead by using a `dict.get()` to lookup file descriptors.

- gh-106558 [https://github.com/python/cpython/issues/106558]: Remove ref cycle in callers of **`convert_to_error()`** by deleting `result` from scope in a `finally` block.
- gh-100502 [https://github.com/python/cpython/issues/100502]: Add **`pathlib.PurePath.pathmod`** class attribute that stores the implementation of **`os.path`** used for low-level path operations: either `posixpath` or `ntpath`.
- gh-106527 [https://github.com/python/cpython/issues/106527]: Reduce overhead to add and remove **`asyncio`** readers and writers.
- gh-106524 [https://github.com/python/cpython/issues/106524]: Fix crash in **`_sre.template()`** with templates containing invalid group indices.
- gh-106531 [https://github.com/python/cpython/issues/106531]: Removed `_legacy` and the names it provided from `importlib.resources`: `Resource`, `contents`, `is_resource`, `open_binary`, `open_text`, `path`, `read_binary`, and `read_text`.
- gh-106052 [https://github.com/python/cpython/issues/106052]: **`re`** module: fix the matching of possessive quantifiers in the case of a subpattern containing backtracking.
- gh-106510 [https://github.com/python/cpython/issues/106510]: Improve debug output for atomic groups in regular expressions.
- gh-106503 [https://github.com/python/cpython/issues/106503]: Fix ref cycle in **`asyncio._SelectorSocketTransport`** by removing `_write_ready` in `close`.
- gh-105497 [https://github.com/python/cpython/issues/105497]: Fix flag mask inversion when unnamed flags exist.

- gh-90876 [https://github.com/python/cpython/issues/90876]: Prevent **`multiprocessing.spawn`** from failing to *import* in environments where `sys.executable` is `None`. This regressed in 3.11 with the addition of support for path-like objects in multiprocessing.

- gh-106403 [https://github.com/python/cpython/issues/106403]: Instances of **`typing.TypeVar`**, **`typing.ParamSpec`**, **`typing.ParamSpecArgs`**, **`typing.ParamSpecKwargs`**, and **`typing.TypeVarTuple`** once again support weak references, fixing a regression introduced in Python 3.12.0 beta 1. Patch by Jelle Zijlstra.

- gh-89812 [https://github.com/python/cpython/issues/89812]: Add private `pathlib._PathBase` class, which provides experimental support for virtual filesystems, and may be made public in a future version of Python.

- gh-106292 [https://github.com/python/cpython/issues/106292]: Check for an instance-dict cached value in the **`__get__()`** method of **`functools.cached_property()`**. This better matches the pre-3.12 behavior and improves compatibility for users subclassing **`functools.cached_property()`** and adding a **`__set__()`** method.

- gh-106350 [https://github.com/python/cpython/issues/106350]: Detect possible memory allocation failure in the libtommath function **`mp_init()`** used by the `_tkinter` module.

- gh-106330 [https://github.com/python/cpython/issues/106330]: Fix incorrect matching of empty paths in **`pathlib.PurePath.match()`**. This bug was introduced in Python 3.12.0 beta 1.

- gh-106309 [https://github.com/python/cpython/issues/106309]: Deprecate **`typing.no_type_check_decorator()`**. No major type checker ever added support for this decorator. Patch by Alex Waygood.

- gh-102541 [https://github.com/python/cpython/issues/102541]: Make **`pydoc.doc()`** catch bad module **`ImportError`** when output stream is not `None`.

- gh-106263 [https://github.com/python/cpython/issues/106263]: Fix crash when calling `repr` with a manually constructed SignalDict object. Patch by Charlie Zhao.

- gh-106236 [https://github.com/python/cpython/issues/106236]: Replace `assert` statements with `raise RuntimeError` in **`threading`**, so that `_DummyThread` cannot be joined even with `-OO`.

- gh-106238 [https://github.com/python/cpython/issues/106238]: Fix rare concurrency bug in lock acquisition by the logging package.

- gh-106152 [https://github.com/python/cpython/issues/106152]: Added PY_THROW event hook for **`cProfile`** for generators

- gh-106075 [https://github.com/python/cpython/issues/106075]: Added `asyncio.taskgroups.__all__` to `asyncio.__all__` for export in star imports.

- gh-104527 [https://github.com/python/cpython/issues/104527]: Zipapp will now skip over appending an archive to itself.

- gh-106046 [https://github.com/python/cpython/issues/106046]: Improve the error message from **`os.fspath()`** if called on an object where `__fspath__` is set to `None`. Patch by Alex Waygood.

- gh-105987 [https://github.com/python/cpython/issues/105987]: Fix crash due to improper reference counting in **`asyncio`** eager task factory internal routines.

- gh-105974 [https://github.com/python/cpython/issues/105974]: Fix bug where a **`typing.Protocol`** class that had one or more non-callable members would raise **`TypeError`** when **`issubclass()`** was called against it, even if it defined a custom `__subclasshook__` method. The behaviour in Python 3.11 and lower – which has now been restored – was not to raise **`TypeError`** in these situations if a custom `__subclasshook__` method was defined. Patch by Alex Waygood.

- gh-96145 [https://github.com/python/cpython/issues/96145]: Reverted addition of `json.AttrDict`.

- gh-89812 [https://github.com/python/cpython/issues/89812]: Add **`pathlib.UnsupportedOperation`**, which is raised instead of **`NotImplementedError`** when a path operation isn't supported.

- gh-105808 [https://github.com/python/cpython/issues/105808]: Fix a regression introduced in gh-101251 [https://github.com/python/cpython/issues/101251] for 3.12, causing **`gzip.GzipFile.flush()`** to not flush the compressor (nor pass along the `zip_mode` argument).

- gh-105481 [https://github.com/python/cpython/issues/105481]: **`stack_effect()`** no longer raises an exception if an `oparg` is provided for an `opcode` that doesn't use its arg, or when it is not provided for an `opcode` that does use it. In the latter case, the stack effect is returned for `oparg=0`.

- gh-104799 [https://github.com/python/cpython/issues/104799]: Enable **`ast.unparse()`** to unparse function and class definitions created without the new `type_params` field from

PEP 695 [https://peps.python.org/pep-0695/]. Patch by Jelle Zijlstra.

- gh-105793 [https://github.com/python/cpython/issues/105793]: Add *follow_symlinks* keyword-only argument to **`pathlib.Path.is_dir()`** and **`is_file()`**, defaulting to `True`.
- gh-105570 [https://github.com/python/cpython/issues/105570]: Deprecate two methods of creating **`typing.TypedDict`** classes with 0 fields using the functional syntax: `TD = TypedDict("TD")` and `TD = TypedDict("TD", None)`. Both will be disallowed in Python 3.15. To create a `TypedDict` class with 0 fields, either use `class TD(TypedDict): pass` or `TD = TypedDict("TD", {})`.
- gh-105745 [https://github.com/python/cpython/issues/105745]: Fix `webbrowser.Konqueror.open` method.
- gh-105733 [https://github.com/python/cpython/issues/105733]: **`ctypes`**: Deprecate undocumented **`ctypes.SetPointerType()`** and **`ctypes.ARRAY()`** functions. Patch by Victor Stinner.
- gh-105687 [https://github.com/python/cpython/issues/105687]: Remove deprecated `re.template`, `re.T`, `re.TEMPLATE`, `sre_constans.SRE_FLAG_TEMPLATE`.
- gh-105684 [https://github.com/python/cpython/issues/105684]: Supporting **`asyncio.Task.set_name()`** is now mandatory for third party task implementations. The undocumented **`_set_task_name()`** function (deprecated since 3.8) has been removed. Patch by Kumar Aditya.
- gh-105375 [https://github.com/python/cpython/issues/105375]: Fix a bug in **`_Unpickler_SetInputStream()`** where an exception could end up being overwritten in case of failure.

- gh-105626 [https://github.com/python/cpython/issues/105626]: Change the default return value of `http.client.HTTPConnection.get_proxy_response_headers()` to be `None` and not `{}`.

- gh-105375 [https://github.com/python/cpython/issues/105375]: Fix bugs in `sys` where exceptions could end up being overwritten because of deferred error handling.

- gh-105605 [https://github.com/python/cpython/issues/105605]: Harden **`pyexpat`** error handling during module initialisation to prevent exceptions from possibly being overwritten, and objects from being dereferenced twice.

- gh-105375 [https://github.com/python/cpython/issues/105375]: Fix bug in `decimal` where an exception could end up being overwritten.

- gh-105375 [https://github.com/python/cpython/issues/105375]: Fix bugs in **`_datetime`** where exceptions could be overwritten in case of module initialisation failure.

- gh-105375 [https://github.com/python/cpython/issues/105375]: Fix bugs in **`_ssl`** initialisation which could lead to leaked references and overwritten exceptions.

- gh-105375 [https://github.com/python/cpython/issues/105375]: Fix a bug in `array.array` where an exception could end up being overwritten.

- gh-105375 [https://github.com/python/cpython/issues/105375]: Fix bugs in **`_ctypes`** where exceptions could end up being overwritten.

- gh-105375 [https://github.com/python/cpython/issues/105375]: Fix a bug in the `posix` module where an exception could be overwritten.

- gh-105375 [https://github.com/python/cpython/issues/105375]: Fix bugs in **`_elementtree`** where exceptions could be overwritten.

- gh-105375 [https://github.com/python/cpython/issues/105375]: Fix bugs in **`zoneinfo`** where exceptions could be overwritten.

- gh-105375 [https://github.com/python/cpython/issues/105375]: Fix bugs in **`errno`** where exceptions could be overwritten.

- gh-105566 [https://github.com/python/cpython/issues/105566]: Deprecate creating a **`typing.NamedTuple`** class using keyword arguments to denote the fields (`NT = NamedTuple("NT", x=int, y=str)`). This will be disallowed in Python 3.15. Use the class-based syntax or the functional syntax instead.

  Two methods of creating `NamedTuple` classes with 0 fields using the functional syntax are also deprecated, and will be disallowed in Python 3.15: `NT = NamedTuple("NT")` and `NT = NamedTuple("NT", None)`. To create a `NamedTuple` class with 0 fields, either use `class NT(NamedTuple): pass` or `NT = NamedTuple("NT", [])`.

- gh-105545 [https://github.com/python/cpython/issues/105545]: Remove deprecated in 3.11 `webbrowser.MacOSXOSAScript._name` attribute.

- gh-105497 [https://github.com/python/cpython/issues/105497]: Fix flag inversion when alias/mask members exist.

- gh-105509 [https://github.com/python/cpython/issues/105509]: **`typing.Annotated`** is now implemented as an instance of `typing._SpecialForm` rather than a class. This should have no user-facing impact for users of the **`typing`** module public API.

- gh-105375 [https://github.com/python/cpython/issues/105375]: Fix bugs in `pickle` where exceptions could be overwritten.

- gh-70303 [https://github.com/python/cpython/issues/70303]: Emit `FutureWarning` from `pathlib.Path.glob()` and `rglob()` if the given pattern ends with "`**`". In a future Python release, patterns with this ending will match both files and directories. Add a trailing slash to only match directories.

- gh-105375 [https://github.com/python/cpython/issues/105375]: Fix a bug in `sqlite3` where an exception could be overwritten in the `collation` callback.

- gh-105382 [https://github.com/python/cpython/issues/105382]: Remove *cafile*, *capath* and *cadefault* parameters of the `urllib.request.urlopen()` function, deprecated in Python 3.6. Patch by Victor Stinner.

- gh-105376 [https://github.com/python/cpython/issues/105376]: `logging`: Remove undocumented and untested `Logger.warn()` and `LoggerAdapter.warn()` methods and `logging.warn()` function. Deprecated since Python 3.3, they were aliases to the `logging.Logger.warning()` method, **`logging.LoggerAdapter.warning()`** method and `logging.warning()` function. Patch by Victor Stinner.

- gh-105332 [https://github.com/python/cpython/issues/105332]: Revert pickling method from by-name back to by-value.

- gh-104554 [https://github.com/python/cpython/issues/104554]: Add RTSPS scheme support in urllib.parse

- gh-105292 [https://github.com/python/cpython/issues/105292]: Add option to `traceback.format_exception_only()` to recurse into the nested exception of a `BaseExceptionGroup`.

- gh-105280 [https://github.com/python/cpython/issues/105280]: Fix bug where `isinstance([], collections.abc.Mapping)` could evaluate to `True` if garbage collection happened at the wrong time. The bug was caused by changes to the implementation of **`typing.Protocol`** in Python 3.12.

- gh-80480 [https://github.com/python/cpython/issues/80480]: **`array`**: Add `'w'` typecode that represents `Py_UCS4`.

- gh-105239 [https://github.com/python/cpython/issues/105239]: Fix longstanding bug where `issubclass(object, typing.Protocol)` would evaluate to `True` in some edge cases. Patch by Alex Waygood.

- gh-104310 [https://github.com/python/cpython/issues/104310]: In the beta 1 release we added a utility function for extension module authors, to use when testing their module for support in multiple interpreters or under a per-interpreter GIL. The name of that function has changed from `allowing_all_extensions` to `_incompatible_extension_module_restrictions`. The default for the "disable_check" argument has change from `True` to `False`, to better match the new function name.

- gh-105080 [https://github.com/python/cpython/issues/105080]: Fixed inconsistent signature on derived classes for **`inspect.signature()`**

- gh-105144 [https://github.com/python/cpython/issues/105144]: Fix a recent regression in the **`typing`** module. The regression meant that doing `class Foo(X, typing.Protocol)`, where `X` was a class that had **`abc.ABCMeta`** as its metaclass, would then cause subsequent `isinstance(1, X)` calls to erroneously raise **`TypeError`**. Patch by Alex Waygood.

- gh-62948 [https://github.com/python/cpython/issues/62948]: The **`io.IOBase`** finalizer now logs the `close()` method errors with **`sys.unraisablehook`**. Previously, errors were ignored silently by default, and only logged in Python Development Mode or on Python built on debug mode. Patch by Victor Stinner.

- gh-105096 [https://github.com/python/cpython/issues/105096]: **`wave`**: Deprecate the `getmark()`, `setmark()` and `getmarkers()` methods of the **`wave.Wave_read`** and **`wave.Wave_write`** classes. They will be removed in Python 3.15. Patch by Victor Stinner.

- gh-104992 [https://github.com/python/cpython/issues/104992]: Remove the untested and undocumented **`unittest.TestProgram.usageExit()`** method, deprecated in Python 3.11. Patch by Hugo van Kemenade.

- gh-104996 [https://github.com/python/cpython/issues/104996]: Improve performance of **`pathlib.PurePath`** initialisation by deferring joining of paths when multiple arguments are given.

- gh-101588 [https://github.com/python/cpython/issues/101588]: Deprecate undocumented copy/deepcopy/pickle support for itertools.

- gh-103631 [https://github.com/python/cpython/issues/103631]: Fix `pathlib.PurePosixPath(pathlib.PureWindowsPath(...))` not converting path separators to restore 3.11 compatible behavior.

- gh-104947 [https://github.com/python/cpython/issues/104947]: Make comparisons between **`pathlib.PureWindowsPath`** objects consistent across Windows and Posix to match 3.11 behavior.

- gh-104773 [https://github.com/python/cpython/issues/104773]: **PEP 594** [https://peps.python.org/pep-0594/]: Remove the **`audioop`** module, deprecated in Python 3.11. Patch by Victor Stinner.

- gh-104773 [https://github.com/python/cpython/issues/104773]: **PEP 594** [https://peps.python.org/pep-0594/]: Remove the `aifc` module, deprecated in Python 3.11. Patch by Victor Stinner.

- gh-104773 [https://github.com/python/cpython/issues/104773]: **PEP 594** [https://peps.python.org/pep-0594/]: Remove the `uu` module, deprecated in Python 3.11. Patch by Victor Stinner.

- gh-104935 [https://github.com/python/cpython/issues/104935]: Fix bugs with the interaction between `typing.runtime_checkable()` and `typing.Generic` that were introduced by the **PEP 695** [https://peps.python.org/pep-0695/] implementation. Patch by Jelle Zijlstra.

- gh-104773 [https://github.com/python/cpython/issues/104773]: **PEP 594** [https://peps.python.org/pep-0594/]: Remove the `crypt` module and its private `_crypt` extension, deprecated in Python 3.11. Patch by Victor Stinner.

- gh-104773 [https://github.com/python/cpython/issues/104773]: **PEP 594** [https://peps.python.org/pep-0594/]: Remove the `nis` module, deprecated in Python 3.11. Patch by Victor Stinner.

- gh-104898 [https://github.com/python/cpython/issues/104898]: Add missing `__slots__` to `os.PathLike`.

- gh-104773 [https://github.com/python/cpython/issues/104773]: **PEP 594** [https://peps.python.org/pep-0594/]: Remove the `xdrlib` module, deprecated in Python 3.11. Patch by Victor Stinner.

- gh-104773 [https://github.com/python/cpython/issues/104773]: **PEP 594** [https://peps.python.org/pep-0594/]: Remove the **`nntplib`** module, deprecated in Python 3.11. Patch by Victor Stinner.

- gh-104886 [https://github.com/python/cpython/issues/104886]: Remove the undocumented **`configparser.LegacyInterpolation`** class, deprecated in the docstring since Python 3.2, and with a deprecation warning since Python 3.11. Patch by Hugo van Kemenade.

- gh-104786 [https://github.com/python/cpython/issues/104786]: Remove kwargs-based **`typing.TypedDict`** creation

- gh-104876 [https://github.com/python/cpython/issues/104876]: Remove the **`turtle.RawTurtle.settiltangle()`** method, deprecated in docs since Python 3.1 and with a deprecation warning since Python 3.11. Patch by Hugo van Kemenade.

- gh-104773 [https://github.com/python/cpython/issues/104773]: **PEP 594** [https://peps.python.org/pep-0594/]: Removed the **`msilib`** package, deprecated in Python 3.11.

- gh-104773 [https://github.com/python/cpython/issues/104773]: **PEP 594** [https://peps.python.org/pep-0594/]: Remove the **`spwd`** module, deprecated in Python 3.11: the python-pam [https://pypi.org/project/python-pam/] project can be used instead. Patch by Victor Stinner.

- gh-75552 [https://github.com/python/cpython/issues/75552]: Removed the `tkinter.tix` module, deprecated since Python 3.6.

- gh-104773 [https://github.com/python/cpython/issues/104773]: **PEP 594** [https://peps.python.org/pep-0594/]: Remove the **`chunk`** module, deprecated in Python 3.11. Patch by Victor Stinner.

- gh-104773 [https://github.com/python/cpython/issues/104773]: **PEP 594** [https://peps.python.org/pep-0594/]: Remove the **`mailcap`** module, deprecated in Python 3.11. Patch by Victor Stinner.

- gh-104773 [https://github.com/python/cpython/issues/104773]: **PEP 594** [https://peps.python.org/pep-0594/]: Remove the **`sunau`** module, deprecated in Python 3.11. Patch by Victor Stinner.

- gh-104780 [https://github.com/python/cpython/issues/104780]: **PEP 594** [https://peps.python.org/pep-0594/]: Remove the **`ossaudiodev`** module, deprecated in Python 3.11. Patch Victor Stinner.

- gh-104773 [https://github.com/python/cpython/issues/104773]: **PEP 594** [https://peps.python.org/pep-0594/]: Remove the **`pipes`** module, deprecated in Python 3.11. Patch by Victor Stinner.

- gh-104873 [https://github.com/python/cpython/issues/104873]: Add **`typing.get_protocol_members()`** to return the set of members defining a **`typing.Protocol`**. Add **`typing.is_protocol()`** to check whether a class is a **`typing.Protocol`**. Patch by Jelle Zijlstra.

- gh-104874 [https://github.com/python/cpython/issues/104874]: Document the `__name__` and `__supertype__` attributes of **`typing.NewType`**. Patch by Jelle Zijlstra.

- gh-104835 [https://github.com/python/cpython/issues/104835]: Removed the following **`unittest`** functions, deprecated in Python 3.11:

  - **`unittest.findTestCases()`**

  - **`unittest.makeSuite()`**
  - **`unittest.getTestCaseNames()`**

  Use `TestLoader` methods instead:

  - `unittest.TestLoader.loadTestsFromModule()`
  - `unittest.TestLoader.loadTestsFromTestCase()`
  - `unittest.TestLoader.getTestCaseNames()`

  Patch by Hugo van Kemenade.

- gh-104804 [https://github.com/python/cpython/issues/104804]: Remove the untested and undocumented `webbrowser` **`MacOSX`** class, deprecated in Python 3.11. Patch by Hugo van Kemenade.

- gh-83863 [https://github.com/python/cpython/issues/83863]: Support for using `pathlib.Path` objects as context managers has been removed. Before Python 3.9, exiting the context manager marked a path as “closed”, which caused some (but not all!) methods to raise when called. Since Python 3.9, using a path as a context manager does nothing.

- gh-104799 [https://github.com/python/cpython/issues/104799]: Adjust the location of the (see **PEP 695** [https://peps.python.org/pep-0695/]) `type_params` field on `ast.ClassDef`, `ast.AsyncFunctionDef`, and `ast.FunctionDef` to better preserve backward compatibility. Patch by Jelle Zijlstra

- gh-104797 [https://github.com/python/cpython/issues/104797]: Allow `typing.Protocol` classes to inherit from `collections.abc.Buffer`. Patch by Jelle Zijlstra.

- gh-104783 [https://github.com/python/cpython/issues/104783]: Remove `locale.resetlocale()` function deprecated in

Python 3.11. Patch by Victor Stinner.

- gh-104780 [https://github.com/python/cpython/issues/104780]: Remove the `2to3` program and the **`lib2to3`** module, deprecated in Python 3.11. Patch by Victor Stinner.

- gh-104773 [https://github.com/python/cpython/issues/104773]: **PEP 594** [https://peps.python.org/pep-0594/]: Remove the **`telnetlib`** module, deprecated in Python 3.11. Patch by Victor Stinner.

- gh-104773 [https://github.com/python/cpython/issues/104773]: **PEP 594** [https://peps.python.org/pep-0594/]: Remove the **`imghdr`** module, deprecated in Python 3.11. Patch by Victor Stinner.

- gh-104773 [https://github.com/python/cpython/issues/104773]: **PEP 594** [https://peps.python.org/pep-0594/]: Remove the **`cgi`** and **`cgitb`** modules, deprecated in Python 3.11. Patch by Victor Stinner.

- gh-104773 [https://github.com/python/cpython/issues/104773]: **PEP 594** [https://peps.python.org/pep-0594/]: Remove the **`sndhdr`** module, deprecated in Python 3.11. Patch by Victor Stinner.

- gh-104372 [https://github.com/python/cpython/issues/104372]: On Linux where **`subprocess`** can use the `vfork()` syscall for faster spawning, prevent the parent process from blocking other threads by dropping the GIL while it waits for the vfork’ed child process `exec()` outcome. This prevents spawning a binary from a slow filesystem from blocking the rest of the application.

- gh-99108 [https://github.com/python/cpython/issues/99108]: We now release the GIL around built-in **`hashlib`** computations of reasonable size for the SHA families and MD5 hash functions, matching what our OpenSSL backed hash computations already does.

- gh-102613 [https://github.com/python/cpython/issues/102613]: Improve performance of **`pathlib.Path.glob()`** when expanding a pattern with a non-terminal "`**`" component by filtering walked paths through a regular expression, rather than calling **`os.scandir()`** more than once on each directory.

- gh-104399 [https://github.com/python/cpython/issues/104399]: Prepare the `_tkinter` module for building with Tcl 9.0 and future libtommath by replacing usage of deprecated functions **`mp_to_unsigned_bin_n()`** and **`mp_unsigned_bin_size()`** when necessary.

- gh-102676 [https://github.com/python/cpython/issues/102676]: Add fields `start_offset`, `cache_offset`, `end_offset`, `baseopname`, `baseopcode`, `jump_target` and `oparg` to **`dis.Instruction`**.

- gh-103558 [https://github.com/python/cpython/issues/103558]: Fixed `parent` argument validation mechanism of **`argparse`**. Improved test coverage.

- gh-103464 [https://github.com/python/cpython/issues/103464]: Provide helpful usage messages when parsing incorrect **`pdb`** commands.

- gh-103384 [https://github.com/python/cpython/issues/103384]: Generalize the regex pattern `BaseConfigurator.INDEX_PATTERN` to allow spaces and non-alphanumeric characters in keys.

- gh-103124 [https://github.com/python/cpython/issues/103124]: Added multiline statement support for **`pdb`**

- gh-101162 [https://github.com/python/cpython/issues/101162]: Forbid using **`builtins.issubclass()`** with **`types.GenericAlias`** as the first argument.

- gh-103200 [https://github.com/python/cpython/issues/103200]: Fix cache repopulation semantics of zipimport.invalidate_caches(). The cache is now repopulated upon retrieving files with an invalid cache, not when the cache is invalidated.

- gh-100061 [https://github.com/python/cpython/issues/100061]: Fix a bug that causes wrong matches for regular expressions with possessive qualifier.

- gh-77609 [https://github.com/python/cpython/issues/77609]: Add *follow_symlinks* argument to **`pathlib.Path.glob()`** and **`rglob()`**, defaulting to false.

- gh-102541 [https://github.com/python/cpython/issues/102541]: Hide traceback in **`help()`** prompt, when import failed.

- gh-102120 [https://github.com/python/cpython/issues/102120]: Added a stream mode to `tarfile` that allows for reading archives without caching info about the inner files.

- gh-102029 [https://github.com/python/cpython/issues/102029]: Deprecate passing any arguments to **`threading.RLock()`**.

- gh-88233 [https://github.com/python/cpython/issues/88233]: Refactored `zipfile._strip_extra` to use higher level abstractions for extras instead of a heavy-state loop.

- gh-102024 [https://github.com/python/cpython/issues/102024]: Reduce calls of `_idle_semaphore.release()` in **`concurrent.futures.thread._worker()`**.

- gh-73435 [https://github.com/python/cpython/issues/73435]: Add support for recursive wildcards in **`pathlib.PurePath.match()`**.

- gh-84867 [https://github.com/python/cpython/issues/84867]: **`unittest.TestLoader`** no longer loads test cases from exact **`unittest.TestCase`** and **`unittest.FunctionTestCase`** classes.

- gh-99203 [https://github.com/python/cpython/issues/99203]: Restore following CPython <= 3.10.5 behavior of **`shutil.make_archive()`**: do not create an empty archive if `root_dir` is not a directory, and, in that case, raise **`FileNotFoundError`** or **`NotADirectoryError`** regardless of `format` choice. Beyond the brought-back behavior, the function may now also raise these exceptions in `dry_run` mode.

- gh-80480 [https://github.com/python/cpython/issues/80480]: Emit **`DeprecationWarning`** for **`array`**'s `'u'` type code, deprecated in docs since Python 3.3.

- gh-94924 [https://github.com/python/cpython/issues/94924]: **`unittest.mock.create_autospec()`** now properly returns coroutine functions compatible with **`inspect.iscoroutinefunction()`**

- gh-94777 [https://github.com/python/cpython/issues/94777]: Fix hanging **`multiprocessing`** `ProcessPoolExecutor` when a child process crashes while data is being written in the call queue.

- gh-92871 [https://github.com/python/cpython/issues/92871]: Remove the `typing.io` and `typing.re` namespaces, deprecated since Python 3.8. All items are still available from the main **`typing`** module.

- bpo-43633 [https://bugs.python.org/issue?@action=redirect&bpo=43633]: Improve the textual representation of IPv4-mapped IPv6 addresses (**RFC 4291** [https://datatracker.ietf.org/doc/html/rfc4291.html] Sections 2.2, 2.5.5.2) in **`ipaddress`**. Patch by Oleksandr Pavliuk.

- bpo-44850 [https://bugs.python.org/issue?@action=redirect&bpo=44850]: Improve performance of **`operator.methodcaller()`** using the **PEP 590** [https://peps.python.org/pep-0590/] `vectorcall` convention. Patch by Anthony Lee and Pieter Eendebak.

- bpo-44185 [https://bugs.python.org/issue?@action=redirect&bpo=44185]: **`unittest.mock.mock_open()`** will call the **`close()`** method of the file handle mock when it is exiting from the context manager. Patch by Samet Yaslan.

- bpo-40988 [https://bugs.python.org/issue?@action=redirect&bpo=40988]: Improve performance of **`functools.singledispatchmethod`** by caching the generated dispatch wrapper. Optimization suggested by frederico. Patch by @mental32, Alex Waygood and Pieter Eendebak.

- bpo-41768 [https://bugs.python.org/issue?@action=redirect&bpo=41768]: **`unittest.mock`** speccing no longer calls class properties. Patch by Melanie Witt.

- bpo-18319 [https://bugs.python.org/issue?@action=redirect&bpo=18319]: Ensure `gettext(msg)` retrieve translations even if a plural form exists. In other words: `gettext(msg) == ngettext(msg, '', 1)`.

- bpo-17013 [https://bugs.python.org/issue?@action=redirect&bpo=17013]: Add `ThreadingMock` to **`unittest.mock`** that can be used to create Mock objects that can wait until they are called. Patch by Karthikeyan Singaravelan and Mario Corchero.

Documentation

- gh-109209 [https://github.com/python/cpython/issues/109209]: The minimum Sphinx version required for the documentation is now 4.2.
- gh-108826 [https://github.com/python/cpython/issues/108826]: **`dis`** module command-line interface is now mentioned in documentation.
- gh-107305 [https://github.com/python/cpython/issues/107305]: Add documentation for **`PyInterpreterConfig`** and **`Py_NewInterpreterFromConfig()`**. Also clarify some of the nearby docs relative to per-interpreter GIL.
- gh-107008 [https://github.com/python/cpython/issues/107008]: Document the **`curses`** module variables **`LINES`** and **`COLS`**.
- gh-106948 [https://github.com/python/cpython/issues/106948]: Add a number of standard external names to `nitpick_ignore`.
- gh-106232 [https://github.com/python/cpython/issues/106232]: Make timeit doc command lines compatible with Windows by using double quotes for arguments. This works on linux and macOS also.
- gh-105172 [https://github.com/python/cpython/issues/105172]: Fixed **`functools.lru_cache()`** docstring accounting for `typed` argument’s different handling of str and int. Patch by Bar Harel.
- gh-105052 [https://github.com/python/cpython/issues/105052]: Update `timeit` doc to specify that time in seconds is just the default.
- gh-89455 [https://github.com/python/cpython/issues/89455]: Add missing documentation for the `max_group_depth` and `max_group_width` parameters and the `exceptions` attribute of the **`traceback.TracebackException`** class.
- gh-89412 [https://github.com/python/cpython/issues/89412]: Add missing documentation for the `end_lineno` and `end_offset`

attributes of the `traceback.TracebackException` class.

- gh-104943 [https://github.com/python/cpython/issues/104943]: Remove mentions of old Python versions in `typing.NamedTuple`.
- gh-54738 [https://github.com/python/cpython/issues/54738]: Add documentation on how to localize the `argparse` module.
- gh-102823 [https://github.com/python/cpython/issues/102823]: Document the return type of `x // y` when `x` and `y` have type `float`.
- gh-102759 [https://github.com/python/cpython/issues/102759]: Align function signature for `functools.reduce` in documentation and docstring with the C implementation.

Tests

- gh-110647 [https://github.com/python/cpython/issues/110647]: Fix test_stress_modifying_handlers() of test_signal. Patch by Victor Stinner.
- gh-103053 [https://github.com/python/cpython/issues/103053]: Fix test_tools.test_freeze on FreeBSD: run “make distclean” instead of “make clean” in the copied source directory to remove also the “python” program. Patch by Victor Stinner.
- gh-110167 [https://github.com/python/cpython/issues/110167]: Fix a deadlock in test_socket when server fails with a timeout but the client is still running in its thread. Don’t hold a lock to call cleanup functions in doCleanups(). One of the cleanup function waits until the client completes, whereas the client could deadlock if it called addCleanup() in such situation. Patch by Victor Stinner.
- gh-110388 [https://github.com/python/cpython/issues/110388]: Add tests for `tty`.
- gh-81002 [https://github.com/python/cpython/issues/81002]: Add tests for `termios`.
- gh-110367 [https://github.com/python/cpython/issues/110367]: regrtest: When using worker processes (-jN) with –

verbose3 option, regrtest can now display the worker output even if a worker process does crash. Previously, sys.stdout and sys.stderr were replaced and so the worker output was lost on a crash. Patch by Victor Stinner.

- gh-110267 [https://github.com/python/cpython/issues/110267]: Add tests for pickling and copying PyStructSequence objects. Patched by Xuehai Pan.
- gh-110171 [https://github.com/python/cpython/issues/110171]: `libregrtest` now always sets and shows `random.seed`, so tests are more reproducible. Use `--randseed` flag to pass the explicit random seed for tests.
- gh-110152 [https://github.com/python/cpython/issues/110152]: Remove `Tools/scripts/run_tests.py` and `make hostrunnertest`. Just run `./python -m test --slow-ci`, `make buildbottest` or `make test` instead. Python test runner (regrtest) now handles cross-compilation and HOSTRUNNER. It also adds options to Python such fast `-u -E -W default -bb` when `--fast-ci` or `--slow-ci` option is used. Patch by Victor Stinner.
- gh-110031 [https://github.com/python/cpython/issues/110031]: Skip test_threading tests using thread+fork if Python is built with Address Sanitizer (ASAN). Patch by Victor Stinner.
- gh-110088 [https://github.com/python/cpython/issues/110088]: Fix test_asyncio timeouts: don’t measure the maximum duration, a test should not measure a CI performance. Only measure the minimum duration when a task has a timeout or delay. Add `CLOCK_RES` to `test_asyncio.utils`. Patch by Victor Stinner.
- gh-109974 [https://github.com/python/cpython/issues/109974]: Fix race conditions in test_threading lock tests. Wait until a condition is met rather than using **`time.sleep()`** with a hardcoded number of seconds. Patch by Victor Stinner.
- gh-110033 [https://github.com/python/cpython/issues/110033]: Fix `test_interprocess_signal()` of `test_signal`. Make sure that the `subprocess.Popen` object is deleted before the test

raising an exception in a signal handler. Otherwise, `Popen.__del__()` can get the exception which is logged as `Exception ignored in: ...` and the test fails. Patch by Victor Stinner.

- gh-109594 [https://github.com/python/cpython/issues/109594]: Fix test_timeout() of test_concurrent_futures.test_wait. Remove the future which may or may not complete depending if it takes longer than the timeout or not. Keep the second future which does not complete before wait() timeout. Patch by Victor Stinner.
- gh-109972 [https://github.com/python/cpython/issues/109972]: Split test_gdb.py file into a test_gdb package made of multiple tests, so tests can now be run in parallel. Patch by Victor Stinner.
- gh-109566 [https://github.com/python/cpython/issues/109566]: regrtest: When `--fast-ci` or `--slow-ci` option is used, regrtest now replaces the current process with a new process to add `-u -W default -bb -E` options to Python. Patch by Victor Stinner.
- gh-109748 [https://github.com/python/cpython/issues/109748]: Fix `test_zippath_from_non_installed_posix()` of test_venv: don't copy `__pycache__/` sub-directories, because they can be modified by other Python tests running in parallel. Patch by Victor Stinner.
- gh-109739 [https://github.com/python/cpython/issues/109739]: regrtest: Fix reference leak check on Windows. Disable the load tracker on Windows in the reference leak check mode (-R option). Patch by Victor Stinner.
- gh-109276 [https://github.com/python/cpython/issues/109276]: regrtest: When a test fails with "env changed" and the –rerun option is used, the test is now re-run in verbose mode in a fresh process. Patch by Victor Stinner.
- gh-103053 [https://github.com/python/cpython/issues/103053]: Skip test_freeze_simple_script() of test_tools.test_freeze if Python is built with `./configure --enable-optimizations`, which means with Profile Guided Optimization (PGO): it

just makes the test too slow. The freeze tool is tested by many other CIs with other (faster) compiler flags. Patch by Victor Stinner.

- gh-109580 [https://github.com/python/cpython/issues/109580]: Skip `test_perf_profiler` if Python is built with ASAN, MSAN or UBSAN sanitizer. Python does crash randomly in this test on such build. Patch by Victor Stinner.
- gh-109566 [https://github.com/python/cpython/issues/109566]: regrtest: Add `--fast-ci` and `--slow-ci` options. `--fast-ci` uses a default timeout of 10 minutes and `-u all,-cpu` (skip slowest tests). `--slow-ci` uses a default timeout of 20 minutes and `-u all` (run all tests). Patch by Victor Stinner.
- gh-109425 [https://github.com/python/cpython/issues/109425]: libregrtest now decodes stdout of test worker processes with the “backslashreplace” error handler to log corrupted stdout, instead of failing with an error and not logging the stdout. Patch by Victor Stinner.
- gh-109396 [https://github.com/python/cpython/issues/109396]: Fix `test_socket.test_hmac_sha1()` in FIPS mode. Use a longer key: FIPS mode requires at least of at least 112 bits. The previous key was only 32 bits. Patch by Victor Stinner.
- gh-104736 [https://github.com/python/cpython/issues/104736]: Fix test_gdb on Python built with LLVM clang 16 on Linux ppc64le (ex: Fedora 38). Search patterns in gdb “bt” command output to detect when gdb fails to retrieve the traceback. For example, skip a test if `Backtrace stopped: frame did not save the PC` is found. Patch by Victor Stinner.
- gh-109276 [https://github.com/python/cpython/issues/109276]: libregrtest now calls **`random.seed()`** before running each test file when `-r/--randomize` command line option is used. Moreover, it’s also called in worker processes. It should help to make tests more deterministic. Previously, it was only called once in the main process

before running all test files and it was not called in worker processes. Patch by Victor Stinner.

- gh-109276 [https://github.com/python/cpython/issues/109276]: libregrtest now uses a separated file descriptor to write test result as JSON. Previously, if a test wrote debug messages late around the JSON, the main test process failed to parse JSON. Patch by Victor Stinner.
- gh-108996 [https://github.com/python/cpython/issues/108996]: Fix and enable `test_msvcrt`.
- gh-109237 [https://github.com/python/cpython/issues/109237]: Fix `test_site.test_underpth_basic()` when the working directory contains at least one non-ASCII character: encode the `._pth` file to UTF-8 and enable the UTF-8 Mode to use UTF-8 for the child process stdout. Patch by Victor Stinner.
- gh-109230 [https://github.com/python/cpython/issues/109230]: Fix `test_pyexpat.test_exception()`: it can now be run from a directory different than Python source code directory. Before, the test failed in this case. Skip the test if Modules/pyexpat.c source is not available. Skip also the test on Python implementations other than CPython. Patch by Victor Stinner.
- gh-108996 [https://github.com/python/cpython/issues/108996]: Add tests for `msvcrt`.
- gh-109015 [https://github.com/python/cpython/issues/109015]: Fix test_asyncio, test_imaplib and test_socket tests on FreeBSD if the TCP blackhole is enabled (`sysctl net.inet.tcp.blackhole`). Skip the few tests which failed with `ETIMEDOUT` which such non standard configuration. Currently, the FreeBSD GCP image enables TCP and UDP blackhole [https://reviews.freebsd.org/D41751] (`sysctl net.inet.tcp.blackhole=2` and `sysctl net.inet.udp.blackhole=1`). Patch by Victor Stinner.
- gh-91960 [https://github.com/python/cpython/issues/91960]: Skip `test_gdb` if gdb is unable to retrieve Python frame objects: if a frame is `<optimized out>`. When Python is

built with “clang -Og”, gdb can fail to retrieve the *frame* parameter of `_PyEval_EvalFrameDefault()`. In this case, tests like `py_bt()` are likely to fail. Without getting access to Python frames, `python-gdb.py` is mostly clueless on retrieving the Python traceback. Moreover, `test_gdb` is no longer skipped on macOS if Python is built with Clang. Patch by Victor Stinner.

- gh-108962 [https://github.com/python/cpython/issues/108962]: Skip `test_tempfile.test_flags()` if `chflags()` fails with “OSError: [Errno 45] Operation not supported” (ex: on FreeBSD 13). Patch by Victor Stinner.
- gh-91960 [https://github.com/python/cpython/issues/91960]: FreeBSD 13.2 CI coverage for pull requests is now provided by Cirrus-CI (a hosted CI service that supports Linux, macOS, Windows, and FreeBSD).
- gh-89392 [https://github.com/python/cpython/issues/89392]: Removed support of `test_main()` function in tests. They now always use normal unittest test runner.
- gh-108851 [https://github.com/python/cpython/issues/108851]: Fix `test_tomllib` recursion tests for WASI buildbots: reduce the recursion limit and compute the maximum nested array/dict depending on the current available recursion limit. Patch by Victor Stinner.
- gh-108851 [https://github.com/python/cpython/issues/108851]: Add `get_recursion_available()` and `get_recursion_depth()` functions to the **`test.support`** module. Patch by Victor Stinner.
- gh-108834 [https://github.com/python/cpython/issues/108834]: Add `--fail-rerun option` option to regrtest: if a test failed when then passed when rerun in verbose mode, exit the process with exit code 2 (error), instead of exit code 0 (success). Patch by Victor Stinner.
- gh-108834 [https://github.com/python/cpython/issues/108834]: Rename regrtest `--verbose2` option (`-w`) to `--rerun`. Keep `--verbose2` as a deprecated alias. Patch by Victor Stinner.

- gh-108834 [https://github.com/python/cpython/issues/108834]: When regrtest reruns failed tests in verbose mode (`./python -m test --rerun`), tests are now rerun in fresh worker processes rather than being executed in the main process. If a test does crash or is killed by a timeout, the main process can detect and handle the killed worker process. Tests are rerun in parallel if the `-jN` option is used to run tests in parallel. Patch by Victor Stinner.
- gh-108822 [https://github.com/python/cpython/issues/108822]: `regrtest` now computes statistics on all tests: successes, failures and skipped. `test_netrc`, `test_pep646_syntax` and `test_xml_etree` now return results in their `test_main()` function. Patch by Victor Stinner and Alex Waygood.
- gh-108794 [https://github.com/python/cpython/issues/108794]: The **`doctest.DocTestRunner.run()`** method now counts the number of skipped tests. Add **`doctest.DocTestRunner.skips`** and **`doctest.TestResults.skipped`** attributes. Patch by Victor Stinner.
- gh-108388 [https://github.com/python/cpython/issues/108388]: Convert test_concurrent_futures to a package of 7 sub-tests. Patch by Victor Stinner.
- gh-108388 [https://github.com/python/cpython/issues/108388]: Split test_multiprocessing_fork, test_multiprocessing_forkserver and test_multiprocessing_spawn into test packages. Each package is made of 4 sub-tests: processes, threads, manager and misc. It allows running more tests in parallel and so reduce the total test duration. Patch by Victor Stinner.
- gh-105776 [https://github.com/python/cpython/issues/105776]: Fix test_cppext when the C compiler command `-std=c11` option: remove `-std=` options from the compiler command. Patch by Victor Stinner.
- gh-107652 [https://github.com/python/cpython/issues/107652]: Set up CIFuzz to run fuzz targets in GitHub Actions. Patch by

Illia Volochii.

- gh-107237 [https://github.com/python/cpython/issues/107237]: `test_logging`: Fix `test_udp_reconnection()` by increasing the timeout from 100 ms to 5 minutes (LONG_TIMEOUT). Patch by Victor Stinner.
- gh-107178 [https://github.com/python/cpython/issues/107178]: Add the C API test for functions in the Mapping Protocol, the Sequence Protocol and some functions in the Object Protocol.
- gh-106714 [https://github.com/python/cpython/issues/106714]: test_capi: Fix test_no_FatalError_infinite_loop() to no longer write a coredump, by using test.support.SuppressCrashReport. Patch by Victor Stinner.
- gh-104090 [https://github.com/python/cpython/issues/104090]: Avoid creating a reference to the test object in **`collectedDurations()`**.
- gh-106752 [https://github.com/python/cpython/issues/106752]: Moved tests for `zipfile.Path` into `Lib/test/test_zipfile/_path`. Made `zipfile._path` a package.
- gh-106690 [https://github.com/python/cpython/issues/106690]: Add .coveragerc to cpython repository for use with coverage package.
- gh-101634 [https://github.com/python/cpython/issues/101634]: When running the Python test suite with `-jN` option, if a worker stdout cannot be decoded from the locale encoding report a failed testn so the exitcode is non-zero. Patch by Victor Stinner.
- gh-105084 [https://github.com/python/cpython/issues/105084]: When the Python build is configured `--with-wheel-pkg-dir`, tests requiring the `setuptools` and `wheel` wheels will search for the wheels in `WHEEL_PKG_DIR`.
- gh-81005 [https://github.com/python/cpython/issues/81005]: String tests are modified to reflect that `str` and `unicode` are merged in Python 3. Patch by Daniel Fortunov.

- gh-103186 [https://github.com/python/cpython/issues/103186]: Suppress and assert expected RuntimeWarnings in test_sys_settrace.py
- gh-69714 [https://github.com/python/cpython/issues/69714]: Add additional tests to **`calendar`** to achieve full test coverage.

Build

- gh-103053 [https://github.com/python/cpython/issues/103053]: “make check-clean-src” now also checks if the “python” program is found in the source directory: fail with an error if it does exist. Patch by Victor Stinner.

- gh-109191 [https://github.com/python/cpython/issues/109191]: Fix compile error when building with recent versions of libedit.

- gh-110276 [https://github.com/python/cpython/issues/110276]: No longer ignore **`PROFILE_TASK`** failure silently: command used by Profile Guided Optimization (PGO). Patch by Victor Stinner.

- gh-109566 [https://github.com/python/cpython/issues/109566]: Remove `make testall` target: use `make buildbottest` instead. Patch by Victor Stinner.

- gh-109740 [https://github.com/python/cpython/issues/109740]: The experimental `--disable-gil` configure flag now includes “t” (for “threaded”) in extension ABI tags.

- gh-109054 [https://github.com/python/cpython/issues/109054]: Fix building the `_testcapi` extension on Linux AArch64 which requires linking to libatomic when `<cpython/pyatomic.h>` is used: the `_Py_atomic_or_uint64()` function requires libatomic `__atomic_fetch_or_8()` on this platform. The configure script now checks if linking to libatomic is needed and generates a new LIBATOMIC variable used to build the _testcapi extension. Patch by Victor Stinner.

- gh-63760 [https://github.com/python/cpython/issues/63760]: Fix Solaris build: no longer redefine the `gethostname()` function. Solaris defines the function since 2005. Patch by Victor Stinner, original patch by Jakub Kulík.

- gh-108740 [https://github.com/python/cpython/issues/108740]: Fix a race condition in `make regen-all`. The `deepfreeze.c` source and files generated by Argument Clinic are now generated or updated before generating “global objects”. Previously, some identifiers may miss depending on the order in which these files were generated. Patch by Victor Stinner.

- gh-108634 [https://github.com/python/cpython/issues/108634]: Python built with `configure` **`--with-trace-refs`** (tracing references) is now ABI compatible with Python release build and debug build. Patch by Victor Stinner.

- gh-85283 [https://github.com/python/cpython/issues/85283]: The `_stat` C extension is now built with the limited C API. Patch by Victor Stinner.

- gh-108447 [https://github.com/python/cpython/issues/108447]: Fix x86_64 GNU/Hurd build

- gh-107814 [https://github.com/python/cpython/issues/107814]: When calling `find_python.bat` with `-q` it did not properly silence the output of nuget. That is now fixed.

- gh-105481 [https://github.com/python/cpython/issues/105481]: Remove the make target `regen-opcode-targets`, merge its work into `regen-opcode` which repeats most of the calculation. This simplifies the code for the build and reduces code duplication.

- gh-106881 [https://github.com/python/cpython/issues/106881]: Check for `linux/limits.h` before including it in `Modules/posixmodule.c`.

- gh-95855 [https://github.com/python/cpython/issues/95855]: Refactor platform triplet detection code and add detection for MIPS soft float and musl libc.

- gh-106962 [https://github.com/python/cpython/issues/106962]: Detect MPI compilers in `configure`.

- gh-106118 [https://github.com/python/cpython/issues/106118]: Fix compilation for platforms without **`O_CLOEXEC`**. The issue was introduced with Python 3.12b1 in gh-103295 [https://github.com/python/cpython/issues/103295]. Patch by Erlend Aasland.

- gh-105875 [https://github.com/python/cpython/issues/105875]: SQLite 3.15.2 or newer is required to build the **`sqlite3`** extension module. Patch by Erlend Aasland.

- gh-90005 [https://github.com/python/cpython/issues/90005]: Fix a regression in `configure` where we could end up unintentionally linking with `libbsd`.

- gh-102404 [https://github.com/python/cpython/issues/102404]: Document how to perform a WASI build on Linux. Also add Tools/wasm/build_wasi.sh as a reference implementation of the docs.

- gh-89886 [https://github.com/python/cpython/issues/89886]: Autoconf 2.71 and aclocal 1.16.4 is now required to regenerate `configure`.

- gh-104692 [https://github.com/python/cpython/issues/104692]: Include `commoninstall` as a prerequisite for `bininstall`

This ensures that `commoninstall` is completed before `bininstall` is started when parallel builds are used (`make -j install`), and so the `python3` symlink is only installed after all standard library modules are installed.

- gh-101538 [https://github.com/python/cpython/issues/101538]: Add experimental wasi-threads support. Patch by Takashi Yamamoto.

Windows

- gh-110437 [https://github.com/python/cpython/issues/110437]: Allows overriding the source of VC redistributables so that releases can be guaranteed to never downgrade between updates.
- gh-109286 [https://github.com/python/cpython/issues/109286]: Update Windows installer to use SQLite 3.43.1.
- gh-82367 [https://github.com/python/cpython/issues/82367]: **`os.path.realpath()`** now resolves MS-DOS style file names even if the file is not accessible. Patch by Moonsik Park.
- gh-109991 [https://github.com/python/cpython/issues/109991]: Update Windows build to use OpenSSL 3.0.11.
- gh-106242 [https://github.com/python/cpython/issues/106242]: Fixes **`realpath()`** to behave consistently when passed a path containing an embedded null character on Windows. In strict mode, it now raises **`OSError`** instead of the unexpected **`ValueError`**, and in non-strict mode will make the path absolute.
- gh-83180 [https://github.com/python/cpython/issues/83180]: Changes the Python install manager to prefer an active virtual environment when the launched script has a shebang line using a Unix-like virtual command, even if the command requests a specific version of Python.
- gh-106844 [https://github.com/python/cpython/issues/106844]: Fix integer overflow and truncating by the null character in **`_winapi.LCMapStringEx()`** which affects **`ntpath.normcase()`**.

- gh-105436 [https://github.com/python/cpython/issues/105436]: Ensure that an empty environment block is terminated by two null characters, as is required by Windows.
- gh-105146 [https://github.com/python/cpython/issues/105146]: Updated the links at the end of the installer to point to Discourse rather than the mailing lists.
- gh-103646 [https://github.com/python/cpython/issues/103646]: When installed from the Microsoft Store, `pip` no longer defaults to per-user installs. However, as the install directory is unwritable, it should automatically decide to do a per-user install anyway. This should resolve issues when `pip` is passed an option that conflicts with `--user`.
- gh-88745 [https://github.com/python/cpython/issues/88745]: Improve performance of **`shutil.copy2()`** by using the operating system's `CopyFile2` function. This may result in subtle changes to metadata copied along with some files, bringing them in line with normal OS behavior.
- gh-104820 [https://github.com/python/cpython/issues/104820]: Fixes **`stat()`** and related functions on file systems that do not support file ID requests. This includes FAT32 and exFAT.
- gh-104803 [https://github.com/python/cpython/issues/104803]: Add **`os.path.isdevdrive()`** to detect whether a path is on a Windows Dev Drive. Returns `False` on platforms that do not support Dev Drive, and is absent on non-Windows platforms.

macOS

- gh-109286 [https://github.com/python/cpython/issues/109286]: Update macOS installer to use SQLite 3.43.1.
- gh-109991 [https://github.com/python/cpython/issues/109991]: Update macOS installer to use OpenSSL 3.0.11.
- gh-99079 [https://github.com/python/cpython/issues/99079]: Update macOS installer to use OpenSSL 3.0.9.

IDLE

- gh-104719 [https://github.com/python/cpython/issues/104719]: Remove IDLE’s modification of tokenize.tabsize and test other uses of tokenize data and methods.

Tools/Demos

- gh-109991 [https://github.com/python/cpython/issues/109991]: Update GitHub CI workflows to use OpenSSL 3.0.11 and multissltests to use 1.1.1w, 3.0.11, and 3.1.3.
- gh-108494 [https://github.com/python/cpython/issues/108494]: Argument Clinic [https://devguide.python.org/development-tools/clinic/] now has a partial support of the Limited API: see documentation in the Python Developer’s Guide [https://devguide.python.org/development-tools/clinic/#clinic-howto-limited-capi] Patch by Victor Stinner.
- gh-107704 [https://github.com/python/cpython/issues/107704]: It is now possible to deprecate passing keyword arguments for keyword-or-positional parameters with Argument Clinic, using the new `/ [from X.Y]` syntax. (To be read as *“positional-only from Python version X.Y”*.) See documentation in the Python Developer’s Guide [https://devguide.python.org/development-tools/clinic/#clinic-howto-deprecate-keyword] for more information.
- gh-107880 [https://github.com/python/cpython/issues/107880]: Argument Clinic can now clone **`__init__()`** and **`__new__()`** methods.
- gh-104683 [https://github.com/python/cpython/issues/104683]: Add `--exclude` option to Argument Clinic CLI.
- gh-95065 [https://github.com/python/cpython/issues/95065]: Argument Clinic now supports overriding automatically generated signature by using directive `@text_signature`. See documentation in the Python Developer’s Guide

[https://devguide.python.org/development-tools/clinic/#clinic-howto-override-signature]

- gh-107609 [https://github.com/python/cpython/issues/107609]: Fix duplicate module check in Argument Clinic. Previously, a duplicate definition would incorrectly be silently accepted. Patch by Erlend E. Aasland.
- gh-107467 [https://github.com/python/cpython/issues/107467]: The Argument Clinic command-line tool now prints to stderr instead of stdout on failure.
- gh-106970 [https://github.com/python/cpython/issues/106970]: Fix bugs in the Argument Clinic `destination <name> clear` command; the destination buffers would never be cleared, and the `destination` directive parser would simply continue to the fault handler after processing the command. Patch by Erlend E. Aasland.
- gh-106706 [https://github.com/python/cpython/issues/106706]: Change bytecode syntax for families to remove redundant name matching pseudo syntax.
- gh-106359 [https://github.com/python/cpython/issues/106359]: Argument Clinic now explicitly forbids “kwarg splats” in function calls used as annotations.
- gh-103186 [https://github.com/python/cpython/issues/103186]: `freeze` now fetches `CONFIG_ARGS` from the original CPython instance the Makefile uses to call utility scripts. Patch by Ijtaba Hussain.
- gh-95065 [https://github.com/python/cpython/issues/95065]: It is now possible to deprecate passing parameters positionally with Argument Clinic, using the new `* [from X.Y]` syntax. (To be read as *"keyword-only from Python version X.Y"*.) See documentation in the Python Developer's Guide [https://devguide.python.org/development-tools/clinic/#clinic-howto-deprecate-positional] for more information. Patch by Erlend E. Aasland with help from Alex Waygood, Nikita Sobolev, and Serhiy Storchaka.

C API

- gh-85283 [https://github.com/python/cpython/issues/85283]: If the **`Py_LIMITED_API`** macro is defined, **`Py_BUILD_CORE`**, **`Py_BUILD_CORE_BUILTIN`** and **`Py_BUILD_CORE_MODULE`** macros are now undefined by `<Python.h>`. Patch by Victor Stinner.

- gh-110289 [https://github.com/python/cpython/issues/110289]: Add **`PyUnicode_EqualToUTF8AndSize()`** and **`PyUnicode_EqualToUTF8()`** functions.

- gh-110235 [https://github.com/python/cpython/issues/110235]: Raise **`TypeError`** for duplicate/unknown fields in `PyStructSequence` constructor. Patched by Xuehai Pan.

- gh-110014 [https://github.com/python/cpython/issues/110014]: Remove undocumented `PY_TIMEOUT_MAX` constant from the limited C API. Patch by Victor Stinner.

- gh-109521 [https://github.com/python/cpython/issues/109521]: **`PyImport_GetImporter()`** now sets RuntimeError if it fails to get **`sys.path_hooks`** or **`sys.path_importer_cache`** or they are not list and dict correspondingly. Previously it could return NULL without setting error in obscure cases, crash or raise SystemError if these attributes have wrong type.

- gh-108724 [https://github.com/python/cpython/issues/108724]: Add **`PyMutex`** internal-only lightweight locking API.

- gh-85283 [https://github.com/python/cpython/issues/85283]: Add **`PySys_AuditTuple()`** function: similar to **`PySys_Audit()`**, but pass event arguments as a Python **`tuple`** object. Patch by Victor Stinner.

- gh-108867 [https://github.com/python/cpython/issues/108867]: Add **`PyThreadState_GetUnchecked()`** function: similar to

`PyThreadState_Get()`, but don't kill the process with a fatal error if it is NULL. The caller is responsible to check if the result is NULL. Previously, the function was private and known as `_PyThreadState_UncheckedGet()`. Patch by Victor Stinner.

- gh-108765 [https://github.com/python/cpython/issues/108765]: `Python.h` no longer includes the `<ctype.h>` standard header file. If needed, it should now be included explicitly. For example, it provides `isalpha()` and `tolower()` functions which are locale dependent. Python provides locale independent functions, like **`Py_ISALPHA()`** and **`Py_TOLOWER()`**. Patch by Victor Stinner.

- gh-108765 [https://github.com/python/cpython/issues/108765]: `Python.h` no longer includes the `<unistd.h>` standard header file. If needed, it should now be included explicitly. For example, it provides the functions: `close()`, `getpagesize()`, `getpid()` and `sysconf()`. Patch by Victor Stinner.

- gh-108765 [https://github.com/python/cpython/issues/108765]: `Python.h` no longer includes the `<ieeefp.h>` standard header. It was included for the `finite()` function which is now provided by the `<math.h>` header. It should now be included explicitly if needed. Remove also the `HAVE_IEEEFP_H` macro. Patch by Victor Stinner.

- gh-108765 [https://github.com/python/cpython/issues/108765]: `Python.h` no longer includes these standard header files: `<time.h>`, `<sys/select.h>` and `<sys/time.h>`. If needed, they should now be included explicitly. For example, `<time.h>` provides the `clock()` and `gmtime()` functions, `<sys/select.h>` provides the `select()` function, and `<sys/time.h>` provides the `futimes()`, `gettimeofday()` and `setitimer()` functions. Patch by Victor Stinner.

- gh-108511 [https://github.com/python/cpython/issues/108511]: Add functions **`PyObject_HasAttrWithError()`**, **`PyObject_HasAttrStringWithError()`**, **`PyMapping_HasKeyWithError()`** and **`PyMapping_HasKeyStringWithError()`**.

- gh-107073 [https://github.com/python/cpython/issues/107073]: Add **`PyObject_VisitManagedDict()`** and **`PyObject_ClearManagedDict()`** functions which must be called by the traverse and clear functions of a type using **`Py_TPFLAGS_MANAGED_DICT`** flag. Patch by Victor Stinner.

- gh-108634 [https://github.com/python/cpython/issues/108634]: Python built with `configure` **`--with-trace-refs`** (tracing references) now supports the Limited API. Patch by Victor Stinner.

- gh-108014 [https://github.com/python/cpython/issues/108014]: Add **`PyLong_AsInt()`** function: similar to **`PyLong_AsLong()`**, but store the result in a C int instead of a C long. Previously, it was known as the private function **`_PyLong_AsInt()`** (with an underscore prefix). Patch by Victor Stinner.

- gh-108314 [https://github.com/python/cpython/issues/108314]: Add **`PyDict_ContainsString()`** function: same as **`PyDict_Contains()`**, but *key* is specified as a const char* UTF-8 encoded bytes string, rather than a PyObject*. Patch by Victor Stinner.

- gh-108337 [https://github.com/python/cpython/issues/108337]: Add atomic operations on additional data types in pyatomic.h.

- gh-108014 [https://github.com/python/cpython/issues/108014]: Add **`Py_IsFinalizing()`** function: check if the main Python interpreter is shutting down. Patch by Victor Stinner.

- gh-107916 [https://github.com/python/cpython/issues/107916]: C API functions `PyErr_SetFromErrnoWithFilename()`, `PyErr_SetExcFromWindowsErrWithFilename()` and `PyErr_SetFromWindowsErrWithFilename()` save now the error code before calling `PyUnicode_DecodeFSDefault()`.

- gh-107915 [https://github.com/python/cpython/issues/107915]: Such C API functions as `PyErr_SetString()`, `PyErr_Format()`, `PyErr_SetFromErrnoWithFilename()` and many others no longer crash or ignore errors if it failed to format the error message or decode the filename. Instead, they keep a corresponding error.

- gh-107810 [https://github.com/python/cpython/issues/107810]: Improve `DeprecationWarning` for uses of `PyType_Spec` with metaclasses that have custom `tp_new`.

- gh-107249 [https://github.com/python/cpython/issues/107249]: Implement the `Py_UNUSED` macro for Windows MSVC compiler. Patch by Victor Stinner.

- gh-107226 [https://github.com/python/cpython/issues/107226]: `PyModule_AddObjectRef()` is now only available in the limited API version 3.10 or later.

- gh-106320 [https://github.com/python/cpython/issues/106320]: Remove private `_PyUnicode_AsString()` alias to `PyUnicode_AsUTF8()`. It was kept for backward compatibility with Python 3.0 - 3.2. The `PyUnicode_AsUTF8()` is available since Python 3.3. The `PyUnicode_AsUTF8String()` function can be used to keep compatibility with Python 3.2 and older. Patch by Victor Stinner.

- gh-106572 [https://github.com/python/cpython/issues/106572]: Convert `PyObject_DelAttr()` and `PyObject_DelAttrString()` macros to functions. Patch by Victor Stinner.

- gh-106307 [https://github.com/python/cpython/issues/106307]: Add **`PyMapping_GetOptionalItem()`** function.

- gh-106521 [https://github.com/python/cpython/issues/106521]: Add **`PyObject_GetOptionalAttr()`** and **`PyObject_GetOptionalAttrString()`** functions.

- gh-106320 [https://github.com/python/cpython/issues/106320]: Remove `_PyInterpreterState_Get()` alias to **`PyInterpreterState_Get()`** which was kept for backward compatibility with Python 3.8. Patch by Victor Stinner.

- gh-106316 [https://github.com/python/cpython/issues/106316]: Remove `cpython/pytime.h` header file: it only contained private functions. Patch by Victor Stinner.

- gh-106023 [https://github.com/python/cpython/issues/106023]: Remove private `_PyObject_FastCall()` function: use `PyObject_Vectorcall()` which is available since Python 3.8 (**PEP 590** [https://peps.python.org/pep-0590/]). Patch by Victor Stinner.

- gh-106168 [https://github.com/python/cpython/issues/106168]: If Python is built in debug mode or **`with assertions`**, **`PyTuple_SET_ITEM()`** and **`PyList_SET_ITEM()`** now check the index argument with an assertion. If the assertion fails, make sure that the size is set before. Patch by Victor Stinner.

- gh-106084 [https://github.com/python/cpython/issues/106084]: Remove the old aliases to functions calling functions which were kept for backward compatibility with Python 3.8 provisional API:

    - `_PyObject_CallMethodNoArgs()`: use `PyObject_CallMethodNoArgs()`

  - `_PyObject_CallMethodOneArg()`: use `PyObject_CallMethodOneArg()`
  - `_PyObject_CallOneArg()`: use `PyObject_CallOneArg()`
  - `_PyObject_FastCallDict()`: use `PyObject_VectorcallDict()`
  - `_PyObject_Vectorcall()`: use `PyObject_Vectorcall()`
  - `_PyObject_VectorcallMethod()`: use `PyObject_VectorcallMethod()`
  - `_PyVectorcall_Function()`: use `PyVectorcall_Function()`

  Just remove the underscore prefix to update your code. Patch by Victor Stinner.

- gh-106004 [https://github.com/python/cpython/issues/106004]: Adds **`PyDict_GetItemRef()`** and **`PyDict_GetItemStringRef()`** functions: similar to **`PyDict_GetItemWithError()`** but returning a strong reference instead of a borrowed reference. Patch by Victor Stinner.

- gh-105927 [https://github.com/python/cpython/issues/105927]: Deprecate the **`PyWeakref_GetObject()`** and **`PyWeakref_GET_OBJECT()`** functions: use the new **`PyWeakref_GetRef()`** function instead. Patch by Victor Stinner.

- gh-105927 [https://github.com/python/cpython/issues/105927]: Add **`PyWeakref_GetRef()`** function: similar to **`PyWeakref_GetObject()`** but returns a strong reference, or `NULL` if the referent is no longer live. Patch by Victor Stinner.

- gh-105922 [https://github.com/python/cpython/issues/105922]: Add **`PyImport_AddModuleRef()`**: similar to **`PyImport_AddModule()`**, but return a strong reference instead of a borrowed reference. Patch by Victor Stinner.

- gh-105227 [https://github.com/python/cpython/issues/105227]: The new **`PyType_GetDict()`** provides the dictionary for the given type object that is normally exposed by `cls.__dict__`. Normally it’s sufficient to use **`tp_dict`**, but for the static builtin types **`tp_dict`** is now always `NULL`. **`PyType_GetDict()`** provides the correct dict object instead.

- gh-105375 [https://github.com/python/cpython/issues/105375]: Fix a bug in **`PyErr_WarnExplicit()`** where an exception could end up being overwritten if the API failed internally.

- gh-105603 [https://github.com/python/cpython/issues/105603]: We’ve renamed the new (in 3.12) `PyInterpreterConfig.own_gil` to `PyInterpreterConfig.gil` and changed the meaning of the value from “bool” to an integer with supported values of `PyInterpreterConfig_DEFAULT_GIL`, `PyInterpreterConfig_SHARED_GIL`, and `PyInterpreterConfig_OWN_GIL`. The default is “shared”.

- gh-105387 [https://github.com/python/cpython/issues/105387]: In the limited C API version 3.12, **`Py_INCREF()`** and **`Py_DECREF()`** functions are now implemented as opaque function calls to hide implementation details. Patch by Victor Stinner.

- gh-105396 [https://github.com/python/cpython/issues/105396]: Deprecate the **`PyImport_ImportModuleNoBlock()`** function which is just an alias to **`PyImport_ImportModule()`** since Python 3.3. Patch by Victor Stinner.

- gh-103968 [https://github.com/python/cpython/issues/103968]: **`PyType_FromMetaclass()`** now allows metaclasses with `tp_new` set to `NULL`.

- gh-105268 [https://github.com/python/cpython/issues/105268]: Remove the old private, undocumented and untested `_PyGC_FINALIZED()` macro which was kept for backward compatibility with Python 3.8 and older. Patch by Victor Stinner.

- gh-105182 [https://github.com/python/cpython/issues/105182]: Remove `PyEval_AcquireLock()` and `PyEval_ReleaseLock()` functions, deprecated in Python 3.2. Patch by Victor Stinner.

- gh-105182 [https://github.com/python/cpython/issues/105182]: Remove `PyEval_InitThreads()` and `PyEval_ThreadsInitialized()` functions, deprecated in Python 3.9. Patch by Victor Stinner.

- gh-105145 [https://github.com/python/cpython/issues/105145]: Deprecate old Python initialization functions:

  - **`PySys_ResetWarnOptions()`**
  - **`Py_GetExecPrefix()`**
  - **`Py_GetPath()`**
  - **`Py_GetPrefix()`**
  - **`Py_GetProgramFullPath()`**
  - **`Py_GetProgramName()`**
  - **`Py_GetPythonHome()`**

  Patch by Victor Stinner.

- gh-85275 [https://github.com/python/cpython/issues/85275]: `PyObject_AsCharBuffer()`, `PyObject_AsReadBuffer()`, `PyObject_CheckReadBuffer()`, and `PyObject_AsWriteBuffer()` are removed. Please migrate to new buffer protocol; **`PyObject_GetBuffer()`** and **`PyBuffer_Release()`**.

- gh-105156 [https://github.com/python/cpython/issues/105156]: Deprecate the old `Py_UNICODE` and `PY_UNICODE_TYPE` types: use directly the **`wchar_t`** type instead. Since Python 3.3, `Py_UNICODE` and `PY_UNICODE_TYPE` are just aliases to **`wchar_t`**. Patch by Victor Stinner.

- gh-105145 [https://github.com/python/cpython/issues/105145]: Remove the following old functions to configure the Python initialization, deprecated in Python 3.11:
    - `PySys_AddWarnOptionUnicode()`
    - `PySys_AddWarnOption()`
    - `PySys_AddXOption()`
    - `PySys_HasWarnOptions()`
    - `PySys_SetArgvEx()`
    - `PySys_SetArgv()`
    - `PySys_SetPath()`
    - `Py_SetPath()`
    - `Py_SetProgramName()`
    - `Py_SetPythonHome()`
    - `Py_SetStandardStreamEncoding()`
    - `_Py_SetProgramFullPath()`

  Patch by Victor Stinner.

- gh-105107 [https://github.com/python/cpython/issues/105107]: Remove functions deprecated in Python 3.9.
    - `PyEval_CallObject()`, `PyEval_CallObjectWithKeywords()`: use **`PyObject_CallNoArgs()`** and **`PyObject_Call()`** (positional arguments must not be *NULL*) instead.
    - `PyEval_CallFunction()`: use **`PyObject_CallFunction()`** instead.
    - `PyEval_CallMethod()`: use **`PyObject_CallMethod()`** instead.
    - `PyCFunction_Call()`: use **`PyObject_Call()`** instead.

  Patch by Victor Stinner.

- gh-105115 [https://github.com/python/cpython/issues/105115]: `PyTypeObject.tp_bases` (and `tp_mro`) for builtin static types are now shared by all interpreters, whereas in 3.12-beta1 they were stored on `PyInterpreterState`. Also note that now the tuples are immortal objects.

- gh-105071 [https://github.com/python/cpython/issues/105071]: Add `PyUnstable_Exc_PrepReraiseStar` to the unstable C api to expose the implementation of **`except*`**.

- gh-104922 [https://github.com/python/cpython/issues/104922]: `PY_SSIZE_T_CLEAN` is no longer required to use `'#'` formats in APIs like **`PyArg_ParseTuple()`** and **`Py_BuildValue()`**. They uses `Py_ssize_t` for `'#'` regardless `PY_SSIZE_T_CLEAN`.

- gh-104584 [https://github.com/python/cpython/issues/104584]: Add an unstable C API for hooking in an optimizer. This is mainly internal, but marked "unstable" to allow third-party experimentation.

- gh-104668 [https://github.com/python/cpython/issues/104668]: Don't call **`PyOS_InputHook`** or **`PyOS_ReadlineFunctionPointer`** in subinterpreters, since it's generally difficult to avoid using global state in their registered callbacks. This also avoids situations where extensions may find themselves running in a subinterpreter they don't support (or haven't yet been loaded in).

- bpo-42327 [https://bugs.python.org/issue?@action=redirect&bpo=42327]: Add **`PyModule_Add()`** function: similar to **`PyModule_AddObjectRef()`** and **`PyModule_AddObject()`**, but always steals a reference to the value.

- bpo-40309 [https://bugs.python.org/issue?@action=redirect&bpo=40309]: Properly handle trailing spaces before closing parenthesis in **`Py_BuildValue()`** format strings.

Python 3.12.0 beta 1

Release date: 2023-05-22

Security

- gh-99889 [https://github.com/python/cpython/issues/99889]: Fixed a security in flaw in **`uu.decode()`** that could allow for directory traversal based on the input if no `out_file` was specified.
- gh-104049 [https://github.com/python/cpython/issues/104049]: Do not expose the local on-disk location in directory indexes produced by **`http.client.SimpleHTTPRequestHandler`**.
- gh-99108 [https://github.com/python/cpython/issues/99108]: Upgrade built-in **`hashlib`** SHA3 implementation to a verified implementation from the `HACL*` project. Used when OpenSSL is not present or lacks SHA3.
- gh-102153 [https://github.com/python/cpython/issues/102153]: **`urllib.parse.urlsplit()`** now strips leading C0 control and space characters following the specification for URLs defined by WHATWG in response to **CVE 2023-24329** [https://www.cve.org/CVERecord?id=CVE-2023-24329]. Patch by Illia Volochii.

Core and Builtins

- gh-102856 [https://github.com/python/cpython/issues/102856]: Implement PEP 701 changes in the **`tokenize`** module. Patch by Marta Gómez Macías and Pablo Galindo Salgado

- gh-104615 [https://github.com/python/cpython/issues/104615]: Fix wrong ordering of assignments in code like `a, a = x, y`. Contributed by Carl Meyer.

- gh-104572 [https://github.com/python/cpython/issues/104572]: Improve syntax error message for invalid constructs in **PEP 695** [https://peps.python.org/pep-0695/] contexts and in annotations when `from __future__ import annotations` is active.

- gh-104482 [https://github.com/python/cpython/issues/104482]: Fix three error handling bugs in ast.c's validation of pattern matching statements.

- gh-102818 [https://github.com/python/cpython/issues/102818]: Do not add a frame to the traceback in the `sys.setprofile` and `sys.settrace` trampoline functions. This ensures that frames are not duplicated if an exception is raised in the callback function, and ensures that frames are not omitted if a C callback is used and that does not add the frame.

- gh-104405 [https://github.com/python/cpython/issues/104405]: Fix an issue where some bytecode instructions could ignore **PEP 523** [https://peps.python.org/pep-0523/] when "inlining" calls.

- gh-103082 [https://github.com/python/cpython/issues/103082]: Change behavior of `sys.monitoring.events.LINE` events in `sys.monitoring`: Line events now occur when a new line is reached dynamically, instead of using a static approximation, as before. This makes the behavior very similar to that of "line" events in `sys.settrace`. This should ease porting of tools from 3.11 to 3.12.

- gh-104263 [https://github.com/python/cpython/issues/104263]: Fix `float("nan")` to produce a quiet NaN on platforms (like MIPS) where the meaning of the signalling / quiet bit is inverted from its usual meaning. Also introduce a new macro `Py_INFINITY` matching C99's `INFINITY`, and refactor internals to rely on C99's `NAN` and `INFINITY` macros instead of hard-coding bit patterns for infinities and NaNs. Thanks Sebastian Berg.

- gh-99113 [https://github.com/python/cpython/issues/99113]: Multi-phase init extension modules may now indicate that they support running in subinterpreters that have their own GIL. This is done by using `Py_MOD_PER_INTERPRETER_GIL_SUPPORTED` as the value for the `Py_mod_multiple_interpreters` module def slot. Otherwise the module, by default, cannot be imported in such subinterpreters. (This does not affect the main interpreter or subinterpreters that do not have their own GIL.) In addition to the isolation that multi-phase init already normally requires, support for per-interpreter GIL involves one additional constraint: thread-safety. If the module has external (linked) dependencies and those libraries have any state that isn't thread-safe then the module must do the additional work to add thread-safety. This should be an uncommon case.

- gh-99113 [https://github.com/python/cpython/issues/99113]: The GIL is now (optionally) per-interpreter. This is the fundamental change for PEP 684. This is all made possible by virtue of the isolated state of each interpreter in the process. The behavior of the main interpreter remains unchanged. Likewise, interpreters created using `Py_NewInterpreter()` are not affected. To get an interpreter with its own GIL, call `Py_NewInterpreterFromConfig()`.

- gh-104108 [https://github.com/python/cpython/issues/104108]: Multi-phase init extension modules may now indicate whether or not they actually support multiple interpreters. By default such modules are expected to support use in multiple interpreters. In the uncommon case that one does not, it may use the new `Py_mod_multiple_interpreters` module def slot. A value of `0` means the module does not support them. `1` means it does. The default is `1`.

- gh-104142 [https://github.com/python/cpython/issues/104142]: Fix an issue where **`list`** or **`tuple`** repetition could fail to respect **PEP 683** [https://peps.python.org/pep-0683/].

- gh-104078 [https://github.com/python/cpython/issues/104078]: Improve the performance of **`PyObject_HasAttrString()`**

- gh-104066 [https://github.com/python/cpython/issues/104066]: Improve the performance of **`hasattr()`** for module objects with a missing attribute.

- gh-104028 [https://github.com/python/cpython/issues/104028]: Reduce object creation while calling callback function from gc. Patch by Donghee Na.

- gh-104018 [https://github.com/python/cpython/issues/104018]: Disallow the "z" format specifier in %-format of bytes objects.

- gh-102213 [https://github.com/python/cpython/issues/102213]: Fix performance loss when accessing an object's attributes with `__getattr__` defined.

- gh-103895 [https://github.com/python/cpython/issues/103895]: Improve handling of edge cases in showing `Exception.__notes__`. Ensures that the messages always end with a newline and that string/bytes are not exploded over multiple lines. Patch by Carey Metcalfe.

- gh-103907 [https://github.com/python/cpython/issues/103907]: Don't modify the refcounts of known immortal objects (**`True`**, **`False`**, and **`None`**) in the main interpreter loop.

- gh-103899 [https://github.com/python/cpython/issues/103899]: Provide a helpful hint in the **`TypeError`** message when accidentally calling a module object that has a callable attribute of the same name (such as **`dis.dis()`** or **`datetime.datetime`**).

- gh-103845 [https://github.com/python/cpython/issues/103845]: Remove both line and instruction instrumentation before adding new ones for monitoring, to avoid newly added instrumentation being removed immediately.
- gh-103763 [https://github.com/python/cpython/issues/103763]: Implement **PEP 695** [https://peps.python.org/pep-0695/], adding syntactic support for generic classes, generic functions, and type aliases.

  A new `type X = ...` syntax is added for type aliases, which resolves at runtime to an instance of the new class `typing.TypeAliasType`. The value is lazily evaluated and is accessible through the `.__value__` attribute. This is implemented as a new AST node `ast.TypeAlias`.

  New syntax (`class X[T]: ...`, `def func[T](): ...`) is added for defining generic functions and classes. This is implemented as a new `type_params` attribute on the AST nodes for classes and functions. This node holds instances of the new AST classes `ast.TypeVar`, `ast.ParamSpec`, and `ast.TypeVarTuple`.

  `typing.TypeVar`, `typing.ParamSpec`, `typing.ParamSpecArgs`, `typing.ParamSpecKwargs`, `typing.TypeVarTuple`, and `typing.Generic` are now implemented in C rather than Python.

  There are new bytecode instructions `LOAD_LOCALS`, `LOAD_CLASSDICT_OR_GLOBAL`, and `LOAD_CLASSDICT_OR_DEREF` to support correct resolution of names in class namespaces.

  Patch by Eric Traut, Larry Hastings, and Jelle Zijlstra.
- gh-103801 [https://github.com/python/cpython/issues/103801]: Adds three minor linting fixes to the wasm module caught

that were caught by ruff.

- gh-103793 [https://github.com/python/cpython/issues/103793]: Optimized asyncio Task creation by deferring expensive string formatting (task name generation) from Task creation to the first time `get_name` is called. This makes asyncio benchmarks up to 5% faster.
- gh-102310 [https://github.com/python/cpython/issues/102310]: Change the error range for invalid bytes literals.
- gh-103590 [https://github.com/python/cpython/issues/103590]: Do not wrap a single exception raised from a `try-except*` construct in an **`ExceptionGroup`**.
- gh-103650 [https://github.com/python/cpython/issues/103650]: Change the perf map format to remove the ‘0x’ prefix from the addresses
- gh-102856 [https://github.com/python/cpython/issues/102856]: Implement the required C tokenizer changes for PEP 701. Patch by Pablo Galindo Salgado, Lysandros Nikolaou, Batuhan Taskaya, Marta Gómez Macías and sunmy2019.
- gh-100530 [https://github.com/python/cpython/issues/100530]: Clarify the error message raised when the called part of a class pattern isn’t actually a class.
- gh-101517 [https://github.com/python/cpython/issues/101517]: Fix bug in line numbers of instructions emitted for **`except*`**.
- gh-103492 [https://github.com/python/cpython/issues/103492]: Clarify **`SyntaxWarning`** with literal `is` comparison by specifying which literal is problematic, since comparisons using `is` with e.g. `None` and bool literals are idiomatic.

- gh-87729 [https://github.com/python/cpython/issues/87729]: Add **`LOAD_SUPER_ATTR`** (and a specialization for `super().method()`) to speed up `super().method()` and `super().attr`. This makes `super().method()` roughly 2.3x faster and brings it within 20% of the performance of a simple method call. Patch by Vladimir Matveev and Carl Meyer.

- gh-103488 [https://github.com/python/cpython/issues/103488]: Change the internal offset distinguishing yield and return target addresses, so that the instruction pointer is correct for exception handling and other stack unwinding.

- gh-82012 [https://github.com/python/cpython/issues/82012]: The bitwise inversion operator (~) on bool is deprecated. It returns the bitwise inversion of the underlying `int` representation such that `bool(~True) == True`, which can be confusing. Use `not` for logical negation of bools. In the rare case that you really need the bitwise inversion of the underlying `int`, convert to int explicitly `~int(x)`.

- gh-77757 [https://github.com/python/cpython/issues/77757]: Exceptions raised in a typeobject's `__set_name__` method are no longer wrapped by a **`RuntimeError`**. Context information is added to the exception as a **PEP 678** [https://peps.python.org/pep-0678/] note.

- gh-103333 [https://github.com/python/cpython/issues/103333]: **`AttributeError`** now retains the `name` attribute when pickled and unpickled.

- gh-103242 [https://github.com/python/cpython/issues/103242]: Migrate **`set_ecdh_curve()`** method not to use deprecated OpenSSL APIs. Patch by Donghee Na.

- gh-103323 [https://github.com/python/cpython/issues/103323]: We've replaced our use of `_PyRuntime.tstate_current` with a thread-local variable. This is a fairly low-level implementation detail, and there should be no change in behavior.

- gh-84436 [https://github.com/python/cpython/issues/84436]: The implementation of PEP-683 which adds Immortal Objects by using a fixed reference count that skips reference counting to make objects truly immutable.

- gh-102700 [https://github.com/python/cpython/issues/102700]: Allow built-in modules to be submodules. This allows submodules to be statically linked into a CPython binary.

- gh-103082 [https://github.com/python/cpython/issues/103082]: Implement **PEP 669** [https://peps.python.org/pep-0669/] Low Impact Monitoring for CPython.

- gh-88691 [https://github.com/python/cpython/issues/88691]: Reduce the number of inline **`CACHE`** entries for **`CALL`**.

- gh-102500 [https://github.com/python/cpython/issues/102500]: Make the buffer protocol accessible in Python code using the new `__buffer__` and `__release_buffer__` magic methods. See **PEP 688** [https://peps.python.org/pep-0688/] for details. Patch by Jelle Zijlstra.

- gh-97933 [https://github.com/python/cpython/issues/97933]: **PEP 709** [https://peps.python.org/pep-0709/]: inline list, dict and set comprehensions to improve performance and reduce bytecode size.

- gh-99184 [https://github.com/python/cpython/issues/99184]: Bypass instance attribute access of `__name__` in `repr` of **`weakref.ref`**.

- gh-98003 [https://github.com/python/cpython/issues/98003]: Complex function calls are now faster and consume no C stack space.

- bpo-39610 [https://bugs.python.org/issue?@action=redirect&bpo=39610]: `len()` for 0-dimensional **`memoryview`** objects (such as `memoryview(ctypes.c_uint8(42))`) now raises a **`TypeError`**. Previously this returned `1`, which was not consistent with `mem_0d[0]` raising an **`IndexError`**.

- bpo-31821 [https://bugs.python.org/issue?@action=redirect&bpo=31821]: Fix **`pause_reading()`** to work when called from **`connection_made()`** in **`asyncio`**.

Library

- gh-104600 [https://github.com/python/cpython/issues/104600]: **`functools.update_wrapper()`** now sets the `__type_params__` attribute (added by **PEP 695** [https://peps.python.org/pep-0695/]).
- gh-104340 [https://github.com/python/cpython/issues/104340]: When an `asyncio` pipe protocol loses its connection due to an error, and the caller doesn't await `wait_closed()` on the corresponding `StreamWriter`, don't log a warning about an exception that was never retrieved. After all, according to the `StreamWriter.close()` docs, the `wait_closed()` call is optional ("not mandatory").
- gh-104555 [https://github.com/python/cpython/issues/104555]: Fix issue where an **`issubclass()`** check comparing a class `X` against a **`runtime-checkable protocol`** `Y` with non-callable members would not cause **`TypeError`** to be raised if an **`isinstance()`** call had previously been made comparing an instance of `X` to `Y`. This issue was present in edge cases on Python 3.11, but became more prominent in 3.12 due to some unrelated changes that were made to runtime-checkable protocols. Patch by Alex Waygood.
- gh-104372 [https://github.com/python/cpython/issues/104372]: Refactored the `_posixsubprocess` internals to avoid Python

C API usage between fork and exec when marking `pass_fds=` file descriptors inheritable.

- gh-104484 [https://github.com/python/cpython/issues/104484]: Added *case_sensitive* argument to **`pathlib.PurePath.match()`**
- gh-75367 [https://github.com/python/cpython/issues/75367]: Fix data descriptor detection in **`inspect.getattr_static()`**.
- gh-104536 [https://github.com/python/cpython/issues/104536]: Fix a race condition in the internal **`multiprocessing.process`** cleanup logic that could manifest as an unintended `AttributeError` when calling `process.close()`.
- gh-103857 [https://github.com/python/cpython/issues/103857]: Update datetime deprecations' stracktrace to point to the calling line
- gh-101520 [https://github.com/python/cpython/issues/101520]: Move the core functionality of the `tracemalloc` module in the `Python/` folder, leaving just the module wrapper in `Modules/`.
- gh-104392 [https://github.com/python/cpython/issues/104392]: Remove undocumented and unused `_paramspec_tvars` attribute from some classes in **`typing`**.
- gh-102613 [https://github.com/python/cpython/issues/102613]: Fix issue where **`pathlib.Path.glob()`** raised **`RecursionError`** when walking deep directory trees.
- gh-103000 [https://github.com/python/cpython/issues/103000]: Improve performance of **`dataclasses.asdict()`** for the common case where *dict_factory* is `dict`. Patch by David C Ellis.
- gh-104301 [https://github.com/python/cpython/issues/104301]: Allow leading whitespace in disambiguated statements in **`pdb`**.
- gh-104139 [https://github.com/python/cpython/issues/104139]: Teach **`urllib.parse.unsplit()`** to retain the `"//"` when assembling `itms-services://?action=generate-bugs` style Apple Platform Deployment [https://support.apple.com/en-gb/guide/deployment/depce7cefc4d/web] URLs.

- gh-104307 [https://github.com/python/cpython/issues/104307]: **`socket.getnameinfo()`** now releases the GIL while contacting the DNS server
- gh-104310 [https://github.com/python/cpython/issues/104310]: Users may now use `importlib.util.allowing_all_extensions()` (a context manager) to temporarily disable the strict compatibility checks for importing extension modules in subinterpreters.
- gh-87695 [https://github.com/python/cpython/issues/87695]: Fix issue where **`pathlib.Path.glob()`** raised **`OSError`** when it encountered a symlink to an overly long path.
- gh-104265 [https://github.com/python/cpython/issues/104265]: Prevent possible crash by disallowing instantiation of the **`_csv.Reader`** and **`_csv.Writer`** types. The regression was introduced in 3.10.0a4 with PR 23224 (bpo-14935 [https://bugs.python.org/issue?@action=redirect&bpo=14935]). Patch by Radislav Chugunov.
- gh-102613 [https://github.com/python/cpython/issues/102613]: Improve performance of **`pathlib.Path.glob()`** when expanding recursive wildcards ("**") by merging adjacent wildcards and de-duplicating results only when necessary.
- gh-65772 [https://github.com/python/cpython/issues/65772]: Remove unneeded comments and code in turtle.py.
- gh-90208 [https://github.com/python/cpython/issues/90208]: Fixed issue where **`pathlib.Path.glob()`** returned incomplete results when it encountered a **`PermissionError`**. This method now suppresses all **`OSError`** exceptions, except those raised from calling **`is_dir()`** on the top-level path.
- gh-104144 [https://github.com/python/cpython/issues/104144]: Optimize **`asyncio.TaskGroup`** when using **`asyncio.eager_task_factory()`**. Skip scheduling a done callback if a TaskGroup task completes eagerly.
- gh-104144 [https://github.com/python/cpython/issues/104144]: Optimize **`asyncio.gather()`** when using **`asyncio.eager_task_factory()`** to complete eagerly if all

fututres completed eagerly. Avoid scheduling done callbacks for futures that complete eagerly.

- gh-104114 [https://github.com/python/cpython/issues/104114]: Fix issue where `pathlib.Path.glob()` returns paths using the case of non-wildcard segments for corresponding path segments, rather than the real filesystem case.
- gh-104104 [https://github.com/python/cpython/issues/104104]: Improve performance of `pathlib.Path.glob()` by using `re.IGNORECASE` to implement case-insensitive matching.
- gh-104102 [https://github.com/python/cpython/issues/104102]: Improve performance of `pathlib.Path.glob()` when evaluating patterns that contain `'../'` segments.
- gh-103822 [https://github.com/python/cpython/issues/103822]: Update the return type of `weekday` to the newly added Day attribute
- gh-103629 [https://github.com/python/cpython/issues/103629]: Update the `repr` of `typing.Unpack` according to **PEP 692** [https://peps.python.org/pep-0692/].
- gh-103963 [https://github.com/python/cpython/issues/103963]: Make `dis` display the names of the args for **`CALL_INTRINSIC_*`**.
- gh-104035 [https://github.com/python/cpython/issues/104035]: Do not ignore user-defined `__getstate__` and `__setstate__` methods for slotted frozen dataclasses.
- gh-103987 [https://github.com/python/cpython/issues/103987]: In `mmap`, fix several bugs that could lead to access to memory-mapped files after they have been invalidated.
- gh-103977 [https://github.com/python/cpython/issues/103977]: Improve import time of `platform` module.
- gh-88773 [https://github.com/python/cpython/issues/88773]: Added `turtle.teleport()` to the `turtle` module to move a turtle to a new point without tracing a line, visible or invisible. Patch by Liam Gersten.
- gh-103935 [https://github.com/python/cpython/issues/103935]: Use `io.open_code()` for files to be executed instead of raw `open()`

- gh-68968 [https://github.com/python/cpython/issues/68968]: Fixed garbled output of `assertEqual()` when an input lacks final newline.
- gh-100370 [https://github.com/python/cpython/issues/100370]: Fix potential `OverflowError` in `sqlite3.Connection.blobopen()` for 32-bit builds. Patch by Erlend E. Aasland.
- gh-102628 [https://github.com/python/cpython/issues/102628]: Substitute CTRL-D with CTRL-Z in `sqlite3` CLI banner when running on Windows.
- gh-103636 [https://github.com/python/cpython/issues/103636]: Module-level attributes `January` and `February` are deprecated from `calendar`.
- gh-103583 [https://github.com/python/cpython/issues/103583]: Isolate **`_multibytecodec`** and codecs extension modules. Patches by Erlend E. Aasland.
- gh-103848 [https://github.com/python/cpython/issues/103848]: Add checks to ensure that `[` bracketed `]` hosts found by `urllib.parse.urlsplit()` are of IPv6 or IPvFuture format.
- gh-103872 [https://github.com/python/cpython/issues/103872]: Update the bundled copy of pip to version 23.1.2.
- gh-99944 [https://github.com/python/cpython/issues/99944]: Make `dis` display the value of oparg of **`KW_NAMES`**.
- gh-74940 [https://github.com/python/cpython/issues/74940]: The C.UTF-8 locale is no longer converted to en_US.UTF-8, enabling the use of UTF-8 encoding on systems which have no locales installed.
- gh-103861 [https://github.com/python/cpython/issues/103861]: Fix `zipfile.Zipfile` creating invalid zip files when `force_zip64` was used to add files to them. Patch by Carey Metcalfe.
- gh-103857 [https://github.com/python/cpython/issues/103857]: Deprecated `datetime.datetime.utcnow()` and `datetime.datetime.utcfromtimestamp()`. (Patch by Paul Ganssle)
- gh-103839 [https://github.com/python/cpython/issues/103839]: Avoid compilation error due to tommath.h not being found

when building Tkinter against Tcl 8.7 built with bundled libtommath.

- gh-103791 [https://github.com/python/cpython/issues/103791]: `contextlib.suppress` now supports suppressing exceptions raised as part of an `ExceptionGroup`. If other exceptions exist on the group, they are re-raised in a group that does not contain the suppressed exceptions.
- gh-90750 [https://github.com/python/cpython/issues/90750]: Use `datetime.datetime.fromisocalendar()` in the implementation of `datetime.datetime.strptime()`, which should now accept only valid ISO dates. (Patch by Paul Ganssle)
- gh-103685 [https://github.com/python/cpython/issues/103685]: Prepare **`tkinter.Menu.index()`** for Tk 8.7 so that it does not raise `TclError: expected integer but got ""` when it should return `None`.
- gh-81403 [https://github.com/python/cpython/issues/81403]: `urllib.request.CacheFTPHandler` no longer raises **`URLError`** if a cached FTP instance is reused. ftplib’s endtransfer method calls voidresp to drain the connection to handle FTP instance reuse properly.
- gh-103699 [https://github.com/python/cpython/issues/103699]: Add `__orig_bases__` to non-generic TypedDicts, call-based TypedDicts, and call-based NamedTuples. Other TypedDicts and NamedTuples already had the attribute.
- gh-103693 [https://github.com/python/cpython/issues/103693]: Add convenience variable feature to `pdb`
- gh-92248 [https://github.com/python/cpython/issues/92248]: Deprecate `type`, `choices`, and `metavar` parameters of `argparse.BooleanOptionalAction`.
- gh-89415 [https://github.com/python/cpython/issues/89415]: Add `socket` constants for source-specific multicast. Patch by Reese Hyde.
- gh-103673 [https://github.com/python/cpython/issues/103673]: `socketserver` gains `ForkingUnixStreamServer` and `ForkingUnixDatagramServer` classes. Patch by Jay Berry.

- gh-103636 [https://github.com/python/cpython/issues/103636]: Added Enum for months and days in the calendar module.
- gh-84976 [https://github.com/python/cpython/issues/84976]: Create a new `Lib/_pydatetime.py` file that defines the Python version of the `datetime` module, and make `datetime` import the contents of the new library only if the C implementation is missing. Currently, the full Python implementation is defined and then deleted if the C implementation is not available, slowing down `import datetime` unnecessarily.
- gh-103596 [https://github.com/python/cpython/issues/103596]: Attributes/methods are no longer shadowed by same-named enum members, although they may be shadowed by enum.property’s.
- gh-103584 [https://github.com/python/cpython/issues/103584]: Updated `importlib.metadata` with changes from `importlib_metadata` 5.2 through 6.5.0, including: Support `installed-files.txt` for `Distribution.files` when present. `PackageMetadata` now stipulates an additional `get` method allowing for easy querying of metadata keys that may not be present. `packages_distributions` now honors packages and modules with Python modules that not `.py` sources (e.g. `.pyc`, `.so`). Expand protocol for `PackageMetadata.get_all` to match the upstream implementation of `email.message.Message.get_all` in python/typeshed#9620. Deprecated use of `Distribution` without defining abstract methods. Deprecated expectation that `PackageMetadata.__getitem__` will return `None` for missing keys. In the future, it will raise a `KeyError`.
- gh-103578 [https://github.com/python/cpython/issues/103578]: Fixed a bug where **`pdb`** crashes when reading source file with different encoding by replacing **`io.open()`** with **`io.open_code()`**. The new method would also call into the hook set by **`PyFile_SetOpenCodeHook()`**.

- gh-103556 [https://github.com/python/cpython/issues/103556]: Now creating **`inspect.Signature`** objects with positional-only parameter with a default followed by a positional-or-keyword parameter without one is impossible.
- gh-103559 [https://github.com/python/cpython/issues/103559]: Update the bundled copy of pip to version 23.1.1.
- gh-103548 [https://github.com/python/cpython/issues/103548]: Improve performance of **`pathlib.Path.absolute()`** and **`cwd()`** by joining paths only when necessary. Also improve performance of **`pathlib.PurePath.is_absolute()`** on Posix by skipping path parsing and normalization.
- gh-103538 [https://github.com/python/cpython/issues/103538]: Remove `_tkinter` module code guarded by definition of the `TK_AQUA` macro which was only needed for Tk 8.4.7 or earlier and was never actually defined by any build system or documented for manual use.
- gh-103533 [https://github.com/python/cpython/issues/103533]: Update **`cProfile`** to use PEP 669 API
- gh-103525 [https://github.com/python/cpython/issues/103525]: Fix misleading exception message when mixed `str` and `bytes` arguments are supplied to **`pathlib.PurePath`** and **`Path`**.
- gh-103489 [https://github.com/python/cpython/issues/103489]: Add **`getconfig()`** and **`setconfig()`** to **`Connection`** to make configuration changes to a database connection. Patch by Erlend E. Aasland.
- gh-103365 [https://github.com/python/cpython/issues/103365]: Set default Flag boundary to `STRICT` and fix bitwise operations.
- gh-103472 [https://github.com/python/cpython/issues/103472]: Avoid a potential **`ResourceWarning`** in **`http.client.HTTPConnection`** by closing the proxy / tunnel’s CONNECT response explicitly.
- gh-103462 [https://github.com/python/cpython/issues/103462]: Fixed an issue with using **`writelines()`** in **`asyncio`** to send very large payloads that exceed the amount of data that can

be written in one call to `socket.socket.send()` or `socket.socket.sendmsg()`, resulting in the remaining buffer being left unwritten.

- gh-103449 [https://github.com/python/cpython/issues/103449]: Fix a bug in doc string generation in `dataclasses.dataclass()`.
- gh-103092 [https://github.com/python/cpython/issues/103092]: Isolate `_collections` (apply **PEP 687** [https://peps.python.org/pep-0687/]). Patch by Erlend E. Aasland.
- gh-103357 [https://github.com/python/cpython/issues/103357]: Added support for `logging.Formatter` `defaults` parameter to `logging.config.dictConfig()` and `logging.config.fileConfig()`. Patch by Bar Harel.
- gh-103092 [https://github.com/python/cpython/issues/103092]: Adapt the `winreg` extension module to **PEP 687** [https://peps.python.org/pep-0687/].
- gh-74690 [https://github.com/python/cpython/issues/74690]: The performance of `isinstance()` checks against **runtime-checkable protocols** has been considerably improved for protocols that only have a few members. To achieve this improvement, several internal implementation details of the `typing` module have been refactored, including `typing._ProtocolMeta.__instancecheck__`, `typing._is_callable_members_only`, and `typing._get_protocol_attrs`. Patches by Alex Waygood.
- gh-74690 [https://github.com/python/cpython/issues/74690]: The members of a runtime-checkable protocol are now considered "frozen" at runtime as soon as the class has been created. See "What's new in Python 3.12" for more details.
- gh-103256 [https://github.com/python/cpython/issues/103256]: Fixed a bug that caused `hmac` to raise an exception when the requested hash algorithm was not available in OpenSSL despite being available separately as part of `hashlib` itself. It now falls back properly to the built-in. This could happen when, for example, your OpenSSL does not

include SHA3 support and you want to compute `hmac.digest(b'K', b'M', 'sha3_256')`.

- gh-102778 [https://github.com/python/cpython/issues/102778]: Support `sys.last_exc` in **`idlelib`**.
- gh-103285 [https://github.com/python/cpython/issues/103285]: Improve performance of **`ast.get_source_segment()`**.
- gh-103225 [https://github.com/python/cpython/issues/103225]: Fix a bug in **`pdb`** when displaying line numbers of module-level source code.
- gh-103092 [https://github.com/python/cpython/issues/103092]: Adapt the **`msvcrt`** extension module to **PEP 687** [https://peps.python.org/pep-0687/].
- gh-103092 [https://github.com/python/cpython/issues/103092]: Adapt the **`winsound`** extension module to **PEP 687** [https://peps.python.org/pep-0687/].
- gh-93910 [https://github.com/python/cpython/issues/93910]: Remove deprecation of enum `member.member` access.
- gh-102978 [https://github.com/python/cpython/issues/102978]: Fixes **`unittest.mock.patch()`** not enforcing function signatures for methods decorated with `@classmethod` or `@staticmethod` when patch is called with `autospec=True`.
- gh-103092 [https://github.com/python/cpython/issues/103092]: Isolate **`_socket`** (apply **PEP 687** [https://peps.python.org/pep-0687/]). Patch by Erlend E. Aasland.
- gh-100479 [https://github.com/python/cpython/issues/100479]: Add **`pathlib.PurePath.with_segments()`**, which creates a path object from arguments. This method is called whenever a derivative path is created, such as from **`pathlib.PurePath.parent`**. Subclasses may override this method to share information between path objects.
- gh-103220 [https://github.com/python/cpython/issues/103220]: Fix issue where **`os.path.join()`** added a slash when joining onto an incomplete UNC drive with a trailing slash on Windows.
- gh-103204 [https://github.com/python/cpython/issues/103204]: Fixes **`http.server`** accepting HTTP requests with HTTP version

numbers preceded by ‘+’, or ‘-’, or with digit-separating ‘_’ characters. The length of the version numbers is also constrained.

- gh-75586 [https://github.com/python/cpython/issues/75586]: Fix various Windows-specific issues with `shutil.which`.
- gh-103193 [https://github.com/python/cpython/issues/103193]: Improve performance of **`inspect.getattr_static()`**. Patch by Alex Waygood.
- gh-103176 [https://github.com/python/cpython/issues/103176]: **`sys._current_exceptions()`** now returns a mapping from thread-id to an exception instance, rather than to a `(typ, exc, tb)` tuple.
- gh-103143 [https://github.com/python/cpython/issues/103143]: Polish the help messages and docstrings of **`pdb`**.
- gh-103015 [https://github.com/python/cpython/issues/103015]: Add *entrypoint* keyword-only parameter to **`sqlite3.Connection.load_extension()`**, for overriding the SQLite extension entry point. Patch by Erlend E. Aasland.
- gh-103000 [https://github.com/python/cpython/issues/103000]: Improve performance of **`dataclasses.astuple()`** and **`dataclasses.asdict()`** in cases where the contents are common Python types.
- gh-102953 [https://github.com/python/cpython/issues/102953]: The extraction methods in **`tarfile`**, and **`shutil.unpack_archive()`**, have a new a *filter* argument that allows limiting tar features than may be surprising or dangerous, such as creating files outside the destination directory. See Extraction filters for details.
- gh-97696 [https://github.com/python/cpython/issues/97696]: Implemented an eager task factory in asyncio. When used as a task factory on an event loop, it performs eager execution of coroutines. Coroutines that are able to complete synchronously (e.g. return or raise without blocking) are returned immediately as a finished task, and the task is never scheduled to the event loop. If the

coroutine blocks, the (pending) task is scheduled and returned.

- gh-81079 [https://github.com/python/cpython/issues/81079]: Add *case_sensitive* keyword-only argument to `pathlib.Path.glob()` and `rglob()`.
- gh-101819 [https://github.com/python/cpython/issues/101819]: Isolate the `io` extension module by applying **PEP 687** [https://peps.python.org/pep-0687/]. Patch by Kumar Aditya, Victor Stinner, and Erlend E. Aasland.
- gh-91896 [https://github.com/python/cpython/issues/91896]: Deprecate `collections.abc.ByteString`
- gh-101362 [https://github.com/python/cpython/issues/101362]: Speed up `pathlib.Path` construction by omitting the path anchor from the internal list of path parts.
- gh-102114 [https://github.com/python/cpython/issues/102114]: Functions in the `dis` module that accept a source code string as argument now print a more concise traceback when the string contains a syntax or indentation error.
- gh-62432 [https://github.com/python/cpython/issues/62432]: The `unittest` runner will now exit with status code 5 if no tests were run. It is common for test runner misconfiguration to fail to find any tests, this should be an error.
- gh-78079 [https://github.com/python/cpython/issues/78079]: Fix incorrect normalization of UNC device path roots, and partial UNC share path roots, in `pathlib.PurePath`. Pathlib no longer appends a trailing slash to such paths.
- gh-85984 [https://github.com/python/cpython/issues/85984]: Add `tty.cfmakeraw()` and `tty.cfmakecbreak()` to `tty` and modernize, the behavior of `tty.setraw()` and `tty.setcbreak()` to use POSIX.1-2017 Chapter 11 “General Terminal Interface” flag masks by default.
- gh-101688 [https://github.com/python/cpython/issues/101688]: Implement `types.get_original_bases()` to provide further introspection for types.

- gh-101640 [https://github.com/python/cpython/issues/101640]: `argparse.ArgumentParser` now catches errors when writing messages, such as when `sys.stderr` is `None`. Patch by Oleg Iarygin.
- gh-83861 [https://github.com/python/cpython/issues/83861]: Fix datetime.astimezone method return value when invoked on a naive datetime instance that represents local time falling in a timezone transition gap. PEP 495 requires that instances with fold=1 produce earlier times than those with fold=0 in this case.
- gh-89550 [https://github.com/python/cpython/issues/89550]: Decrease execution time of some `gzip` file writes by 15% by adding more appropriate buffering.
- gh-95299 [https://github.com/python/cpython/issues/95299]: Remove the bundled setuptools wheel from `ensurepip`, and stop installing setuptools in environments created by `venv`.
- gh-99353 [https://github.com/python/cpython/issues/99353]: Respect the `http.client.HTTPConnection` `.debuglevel` flag in **`urllib.request.AbstractHTTPHandler`** when its constructor parameter `debuglevel` is not set. And do the same for `*HTTPS*`.
- gh-98040 [https://github.com/python/cpython/issues/98040]: Remove the long-deprecated `imp` module.
- gh-97850 [https://github.com/python/cpython/issues/97850]: Deprecate **`pkgutil.find_loader()`** and **`pkgutil.get_loader()`** in favor of `importlib.util.find_spec()`.
- gh-94473 [https://github.com/python/cpython/issues/94473]: Flatten arguments in **`tkinter.Canvas.coords()`**. It now accepts not only `x1, y1, x2, y2, ...` and `[x1, y1, x2, y2, ...]`, but also `(x1, y1), (x2, y2), ...` and `[(x1, y1), (x2, y2), ...]`.
- gh-98040 [https://github.com/python/cpython/issues/98040]: Remove more deprecated importlib APIs: `find_loader()`, `find_module()`, `importlib.abc.Finder`, `pkgutil.ImpImporter`, `pkgutil.ImpLoader`.

- gh-96522 [https://github.com/python/cpython/issues/96522]: Fix potential deadlock in pty.spawn()
- gh-96534 [https://github.com/python/cpython/issues/96534]: Support divert(4) added in FreeBSD 14.
- gh-87474 [https://github.com/python/cpython/issues/87474]: Fix potential file descriptor leaks in **`subprocess.Popen`**.
- gh-94906 [https://github.com/python/cpython/issues/94906]: Support multiple steps in **`math.nextafter()`**. Patch by Shantanu Jain and Matthias Gorgens.
- gh-51574 [https://github.com/python/cpython/issues/51574]: Make **`tempfile.mkdtemp()`** return absolute paths when its *dir* parameter is relative.
- gh-94518 [https://github.com/python/cpython/issues/94518]: Convert private **`_posixsubprocess.fork_exec()`** to use Argument Clinic.
- gh-92184 [https://github.com/python/cpython/issues/92184]: When creating zip files using **`zipfile`**, `os.altsep`, if not `None`, will always be treated as a path separator even when it is not `/`. Patch by Carey Metcalfe.
- bpo-46797 [https://bugs.python.org/issue?@action=redirect&bpo=46797]: Deprecation warnings are now emitted for **`ast.Num`**, **`ast.Bytes`**, **`ast.Str`**, **`ast.NameConstant`** and **`ast.Ellipsis`**. These have been documented as deprecated since Python 3.8, and will be removed in Python 3.14.
- bpo-44844 [https://bugs.python.org/issue?@action=redirect&bpo=44844]: Enables **`webbrowser`** to detect and launch Microsoft Edge browser.
- bpo-45606 [https://bugs.python.org/issue?@action=redirect&bpo=45606]: Fixed the bug in **`pathlib.Path.glob()`** – previously a dangling symlink would not be found by this method when the pattern is an exact match, but would be found when the pattern contains a wildcard or the recursive wildcard (`**`). With this change, a dangling symlink will be found in both cases.

- bpo-23041 [https://bugs.python.org/issue?@action=redirect&bpo=23041]: Add **`QUOTE_STRINGS`** and **`QUOTE_NOTNULL`** to the suite of **`csv`** module quoting styles.
- bpo-24964 [https://bugs.python.org/issue?@action=redirect&bpo=24964]: Added **`http.client.HTTPConnection.get_proxy_response_headers()`** that provides access to the HTTP headers on a proxy server response to the `CONNECT` request.
- bpo-17258 [https://bugs.python.org/issue?@action=redirect&bpo=17258]: **`multiprocessing`** now supports stronger HMAC algorithms for inter-process connection authentication rather than only HMAC-MD5.
- bpo-39744 [https://bugs.python.org/issue?@action=redirect&bpo=39744]: Make **`asyncio.subprocess.Process.communicate()`** close the subprocess's stdin even when called with `input=None`.
- bpo-22708 [https://bugs.python.org/issue?@action=redirect&bpo=22708]: http.client CONNECT method tunnel improvements: Use HTTP 1.1 protocol; send a matching Host: header with CONNECT, if one is not provided; convert IDN domain names to Punycode. Patch by Michael Handler.

Documentation

- gh-67056 [https://github.com/python/cpython/issues/67056]: Document that the effect of registering or unregistering an **`atexit`** cleanup function from within a registered cleanup function is undefined.
- gh-103629 [https://github.com/python/cpython/issues/103629]: Mention the new way of typing `**kwargs` with `Unpack` and `TypedDict` introduced in **PEP 692** [https://peps.python.org/pep-0692/].
- gh-48241 [https://github.com/python/cpython/issues/48241]: Clarifying documentation about the url parameter to

urllib.request.urlopen and urllib.request.Request needing to be encoded properly.

- gh-86094 [https://github.com/python/cpython/issues/86094]: Add support for Unicode Path Extra Field in ZipFile. Patch by Yeojin Kim and Andrea Giudiceandrea
- gh-99202 [https://github.com/python/cpython/issues/99202]: Fix extension type from documentation for compiling in C++20 mode

Tests

- gh-104494 [https://github.com/python/cpython/issues/104494]: Update `test_pack_configure_in` and `test_place_configure_in` for changes to error message formatting in Tk 8.7.
- gh-104461 [https://github.com/python/cpython/issues/104461]: Run test_configure_screen on X11 only, since the `DISPLAY` environment variable and `-screen` option for toplevels are not useful on Tk for Win32 or Aqua.
- gh-86275 [https://github.com/python/cpython/issues/86275]: Added property-based tests to the **`zoneinfo`** tests, along with stubs for the `hypothesis` interface. (Patch by Paul Ganssle)
- gh-103329 [https://github.com/python/cpython/issues/103329]: Regression tests for the behaviour of `unittest.mock.PropertyMock` were added.
- gh-102795 [https://github.com/python/cpython/issues/102795]: fix use of poll in test_epoll’s test_control_and_wait
- gh-75729 [https://github.com/python/cpython/issues/75729]: Fix the **`os.spawn*`** tests failing on Windows when the working directory or interpreter path contains spaces.

Build

- gh-101282 [https://github.com/python/cpython/issues/101282]: BOLT optimization is now applied to the libpython shared library if building a shared library. BOLT instrumentation and application settings can now be influenced via the `BOLT_INSTRUMENT_FLAGS` and `BOLT_APPLY_FLAGS` configure variables.

- gh-99017 [https://github.com/python/cpython/issues/99017]: `PYTHON_FOR_REGEN` now require Python 3.10 or newer.

- gh-104490 [https://github.com/python/cpython/issues/104490]: Define `.PHONY` / virtual make targets consistently and properly.

- gh-104106 [https://github.com/python/cpython/issues/104106]: Add gcc fallback of mkfifoat/mknodat for macOS. Patch by Donghee Na.

- gh-103532 [https://github.com/python/cpython/issues/103532]: The `TKINTER_PROTECT_LOADTK` macro is no longer defined or used in the `_tkinter` module. It was previously only defined when building against Tk 8.4.13 and older, but Tk older than 8.5.12 has been unsupported since gh-91152 [https://github.com/python/cpython/issues/91152].

- gh-99069 [https://github.com/python/cpython/issues/99069]: Extended workaround defining `static_assert` when missing from the libc headers to all clang and gcc builds. In particular, this fixes building on macOS <= 10.10.

- gh-100220 [https://github.com/python/cpython/issues/100220]: Changed the default value of the `SHELL` Makefile variable from `/bin/sh` to `/bin/sh -e` to ensure that complex recipes correctly fail after an error. Previously, `make install` could fail to install some files and yet return a successful result.

- gh-90656 [https://github.com/python/cpython/issues/90656]: Add platform triplets for 64-bit LoongArch:

  - loongarch64-linux-gnusf
  - loongarch64-linux-gnuf32
  - loongarch64-linux-gnu

  Patch by Zhang Na.

Windows

- gh-104623 [https://github.com/python/cpython/issues/104623]: Update Windows installer to use SQLite 3.42.0.
- gh-82814 [https://github.com/python/cpython/issues/82814]: Fix a potential `[Errno 13] Permission denied` when using **`shutil.copystat()`** within Windows Subsystem for Linux (WSL) on a mounted filesystem by adding `errno.EACCES` to the list of ignored errors within the internal implementation.
- gh-103088 [https://github.com/python/cpython/issues/103088]: Fix virtual environment `activate` script having incorrect line endings for Cygwin.
- gh-103088 [https://github.com/python/cpython/issues/103088]: Fixes venvs not working in bash on Windows across different disks
- gh-102997 [https://github.com/python/cpython/issues/102997]: Update Windows installer to use SQLite 3.41.2.
- gh-88013 [https://github.com/python/cpython/issues/88013]: Fixed a bug where **`TypeError`** was raised when calling **`ntpath.realpath()`** with a bytes parameter in some cases.

macOS

- gh-99834 [https://github.com/python/cpython/issues/99834]: Update macOS installer to Tcl/Tk 8.6.13.
- gh-104623 [https://github.com/python/cpython/issues/104623]: Update macOS installer to SQLite 3.42.0.
- gh-103545 [https://github.com/python/cpython/issues/103545]: Add `os.PRIO_DARWIN_THREAD`, `os.PRIO_DARWIN_PROCESS`,

`os.PRIO_DARWIN_BG` and `os.PRIO_DARWIN_NONUI`. These can be used with `os.setpriority` to run the process at a lower priority and make use of the efficiency cores on Apple Silicon systems.
- gh-104180 [https://github.com/python/cpython/issues/104180]: Support reading SOCKS proxy configuration from macOS System Configuration. Patch by Sam Schott.
- gh-60436 [https://github.com/python/cpython/issues/60436]: update curses textbox to additionally handle backspace using the `curses.ascii.DEL` key press.
- gh-102997 [https://github.com/python/cpython/issues/102997]: Update macOS installer to SQLite 3.41.2.

IDLE

- gh-104499 [https://github.com/python/cpython/issues/104499]: Fix completions for Tk Aqua 8.7 (currently blank).
- gh-104496 [https://github.com/python/cpython/issues/104496]: About prints both tcl and tk versions if different (expected someday).
- gh-88496 [https://github.com/python/cpython/issues/88496]: Fix IDLE test hang on macOS.

Tools/Demos

- gh-104389 [https://github.com/python/cpython/issues/104389]: Argument Clinic C converters now accept the `unused` keyword, for wrapping a parameter with **`Py_UNUSED`**. Patch by Erlend E. Aasland.

C API

- gh-101291 [https://github.com/python/cpython/issues/101291]: Added unstable C API for extracting the value of “compact” integers: **`PyUnstable_Long_IsCompact()`** and **`PyUnstable_Long_CompactValue()`**.

- gh-104109 [https://github.com/python/cpython/issues/104109]: We’ve added `Py_NewInterpreterFromConfig()` and `PyInterpreterConfig` to the public C-API (but not the stable ABI; not yet at least). The new function may be used to create a new interpreter with various features configured. The function was added to support PEP 684 (per-interpreter GIL).
- gh-103968 [https://github.com/python/cpython/issues/103968]: **`PyType_FromSpec()`** and its variants now allow creating classes whose metaclass overrides **`tp_new`**. The `tp_new` is ignored. This behavior is deprecated and will be disallowed in 3.14+. The new **`PyType_FromMetaclass()`** already disallows it.
- gh-103743 [https://github.com/python/cpython/issues/103743]: Add **`PyUnstable_Object_GC_NewWithExtraData()`** function that can be used to allocate additional memory after an object for data not managed by Python.
- gh-103295 [https://github.com/python/cpython/issues/103295]: Introduced **`PyUnstable_WritePerfMapEntry()`**, **`PyUnstable_PerfMapState_Init()`** and **`PyUnstable_PerfMapState_Fini()`**. These allow extension modules (JIT compilers in particular) to write to perf-map files in a thread safe manner. The Python support for the Linux perf profiler also uses these APIs to write entries in the perf-map file.
- gh-103509 [https://github.com/python/cpython/issues/103509]: Added C API for extending types whose instance memory layout is opaque: **`PyType_Spec.basicsize`** can now be zero or negative, **`PyObject_GetTypeData()`** can be used to get subclass-specific data, and **`Py_TPFLAGS_ITEMS_AT_END`** can be used to safely extend variable-size objects. See **PEP 697** [https://peps.python.org/pep-0697/] for details.
- gh-103091 [https://github.com/python/cpython/issues/103091]: Add a new C-API function to eagerly assign a version tag to a PyTypeObject: `PyUnstable_Type_AssignVersionTag()`.

- gh-101408 [https://github.com/python/cpython/issues/101408]: **`PyObject_GC_Resize`** should calculate preheader size if needed. Patch by Donghee Na.
- gh-98836 [https://github.com/python/cpython/issues/98836]: Add support of more formatting options (left aligning, octals, uppercase hexadecimals, **`intmax_t`**, **`ptrdiff_t`**, **`wchar_t`** C strings, variable width and precision) in **`PyUnicode_FromFormat()`** and **`PyUnicode_FromFormatV()`**.
- gh-96803 [https://github.com/python/cpython/issues/96803]: Add unstable C-API functions to get the code object, lasti and line number from the internal `_PyInterpreterFrame` in the limited API. The functions are:
  - `PyCodeObject * PyUnstable_InterpreterFrame_GetCode(struct _PyInterpreterFrame *frame)`
  - `int PyUnstable_InterpreterFrame_GetLasti(struct _PyInterpreterFrame *frame)`
  - `int PyUnstable_InterpreterFrame_GetLine(struct _PyInterpreterFrame *frame)`

Python 3.12.0 alpha 7

Release date: 2023-04-04

Core and Builtins

- gh-102192 [https://github.com/python/cpython/issues/102192]: Deprecated `_PyErr_ChainExceptions` in favour of `_PyErr_ChainExceptions1`.

- gh-89987 [https://github.com/python/cpython/issues/89987]: Reduce the number of inline **`CACHE`** entries for **`BINARY_SUBSCR`**.

- gh-102859 [https://github.com/python/cpython/issues/102859]: Removed **`JUMP_IF_FALSE_OR_POP`** and **`JUMP_IF_TRUE_OR_POP`** instructions.

- gh-101975 [https://github.com/python/cpython/issues/101975]: Fixed `stacktop` value on tracing entries to avoid corruption on garbage collection.
- gh-102778 [https://github.com/python/cpython/issues/102778]: Add **`sys.last_exc`** and deprecate **`sys.last_type`**, **`sys.last_value`** and **`sys.last_traceback`**, which hold the same information in its legacy form.
- gh-100982 [https://github.com/python/cpython/issues/100982]: Replace all occurrences of `COMPARE_AND_BRANCH` with **`COMPARE_OP`**.
- gh-102701 [https://github.com/python/cpython/issues/102701]: Fix overflow when creating very large dict.
- gh-102755 [https://github.com/python/cpython/issues/102755]: Add **`PyErr_DisplayException()`** which takes just an exception instance, to replace the legacy **`PyErr_Display()`** which takes the `(typ, exc, tb)` triplet.
- gh-102594 [https://github.com/python/cpython/issues/102594]: Add note to exception raised in `PyErr_SetObject` when normalization fails.
- gh-90997 [https://github.com/python/cpython/issues/90997]: Shrink the number of inline **`CACHE`** entries used by **`LOAD_GLOBAL`**.
- gh-102491 [https://github.com/python/cpython/issues/102491]: Improve import time of `platform` by removing IronPython version parsing. The IronPython version parsing was not functional (see https://github.com/IronLanguages/ironpython3/issues/1667).

- gh-101291 [https://github.com/python/cpython/issues/101291]: Rearrage bits in first field (after header) of PyLongObject. * Bits 0 and 1: 1 - sign. I.e. 0 for positive numbers, 1 for zero and 2 for negative numbers. * Bit 2 reserved (probably for the immortal bit) * Bits 3+ the unsigned size.

  This makes a few operations slightly more efficient, and will enable a more compact and faster 2s-complement representation of most ints in future.

- gh-102397 [https://github.com/python/cpython/issues/102397]: Fix segfault from race condition in signal handling during garbage collection. Patch by Kumar Aditya.

- gh-102406 [https://github.com/python/cpython/issues/102406]: **`codecs`** encoding/decoding errors now get the context information (which operation and which codecs) attached as **PEP 678** [https://peps.python.org/pep-0678/] notes instead of through chaining a new instance of the exception.

- gh-102281 [https://github.com/python/cpython/issues/102281]: Fix potential nullptr dereference and use of uninitialized memory in fileutils. Patch by Max Bachmann.

- gh-102300 [https://github.com/python/cpython/issues/102300]: Reuse operands with refcount of 1 in float specializations of BINARY_OP.

- gh-102213 [https://github.com/python/cpython/issues/102213]: Fix performance loss when accessing an object’s attributes with `__getattr__` defined.

- gh-102255 [https://github.com/python/cpython/issues/102255]: Improve build support for the Xbox. Patch by Max Bachmann.

- gh-102027 [https://github.com/python/cpython/issues/102027]: Fix SSE2 and SSE3 detection in `_blake2` internal module. Patch by Max Bachmann.

- gh-101865 [https://github.com/python/cpython/issues/101865]: Deprecate `co_lnotab` in code objects, schedule it for removal in Python 3.14

- bpo-1635741 [https://bugs.python.org/issue?@action=redirect&bpo=1635741]: Adapt **`_pickle`** to **PEP 687** [https://peps.python.org/pep-0687/]. Patch by Mohamed Koubaa and Erlend Aasland.

Library

- gh-103085 [https://github.com/python/cpython/issues/103085]: Pure python **`locale.getencoding()`** will not warn deprecation.

- gh-103068 [https://github.com/python/cpython/issues/103068]: It’s no longer possible to register conditional breakpoints in **`Pdb`** that raise **`SyntaxError`**. Patch by Tian Gao.

- gh-102549 [https://github.com/python/cpython/issues/102549]: Don’t ignore exceptions in member type creation.

- gh-103056 [https://github.com/python/cpython/issues/103056]: Ensure final `_generate_next_value_` is a `staticmethod`.

- gh-103046 [https://github.com/python/cpython/issues/103046]: Display current line label correctly in **`dis`** when `show_caches` is False and `lasti` points to a CACHE entry.

- gh-102433 [https://github.com/python/cpython/issues/102433]: **`isinstance()`** checks against **`runtime-checkable protocols`** now use **`inspect.getattr_static()`** rather than **`hasattr()`** to lookup whether attributes exist. This means that descriptors and **`__getattr__()`** methods are no longer unexpectedly evaluated during `isinstance()` checks against runtime-checkable protocols. However, it may also mean that some objects which used to be considered instances of a runtime-checkable protocol may no longer be considered instances of that protocol on Python 3.12+, and vice versa. Most users are unlikely to be affected by this change. Patch by Alex Waygood.

- gh-103023 [https://github.com/python/cpython/issues/103023]: It’s no longer possible to register expressions to display in **`Pdb`** that raise **`SyntaxError`**. Patch by Tian Gao.

- gh-102947 [https://github.com/python/cpython/issues/102947]: Improve traceback when **`dataclasses.fields()`** is called on a non-dataclass. Patch by Alex Waygood

- gh-102780 [https://github.com/python/cpython/issues/102780]: The **`asyncio.Timeout`** context manager now works reliably even when performing cleanup due to task cancellation. Previously it could raise a **`CancelledError`** instead of an **`TimeoutError`** in such cases.

- gh-102871 [https://github.com/python/cpython/issues/102871]: Remove support for obsolete browsers from **`webbrowser`**. Removed browsers include Grail, Mosaic, Netscape, Galeon, Skipstone, Iceape, Firebird, and Firefox versions 35 and below.

- gh-102839 [https://github.com/python/cpython/issues/102839]: Improve performance of **`math.log()`** arguments handling by removing the argument clinic.

- gh-102828 [https://github.com/python/cpython/issues/102828]: Add the `onexc` arg to **`shutil.rmtree()`**, which is like `onerror` but expects an exception instance rather than an exc_info tuple. Deprecate `onerror`.

- gh-88965 [https://github.com/python/cpython/issues/88965]: typing: Fix a bug relating to substitution in custom classes generic over a **`ParamSpec`**. Previously, if the `ParamSpec` was substituted with a parameters list that itself contained a **`TypeVar`**, the `TypeVar` in the parameters list could not be subsequently substituted. This is now fixed.

  Patch by Nikita Sobolev.

- gh-76846 [https://github.com/python/cpython/issues/76846]: Fix issue where `__new__()` and `__init__()` methods of **`pathlib.PurePath`** and **`Path`** subclasses were not called in some circumstances.

- gh-78530 [https://github.com/python/cpython/issues/78530]: **`asyncio.wait()`** now accepts generators yielding tasks. Patch by Kumar Aditya.

- gh-102748 [https://github.com/python/cpython/issues/102748]: **`asyncio.iscoroutine()`** now returns `False` for generators as **`asyncio`** does not support legacy generator-based coroutines. Patch by Kumar Aditya.

- gh-102670 [https://github.com/python/cpython/issues/102670]: Optimized fmean(), correlation(), covariance(), and linear_regression() using the new math.sumprod() function.

- gh-102615 [https://github.com/python/cpython/issues/102615]: Typing: Improve the `repr` of generic aliases for classes generic over a **`ParamSpec`**. (Use square brackets to represent a parameter list.)

- gh-100112 [https://github.com/python/cpython/issues/100112]: **`asyncio.Task.get_coro()`** now always returns a coroutine when wrapping an awaitable object. Patch by Kumar Aditya.

- gh-102578 [https://github.com/python/cpython/issues/102578]: Speed up setting or deleting mutable attributes on non-dataclass subclasses of frozen dataclasses. Due to the implementation of `__setattr__` and `__delattr__` for frozen dataclasses, this previously had a time complexity of $O(n)$. It now has a time complexity of $O(1)$.

- gh-102519 [https://github.com/python/cpython/issues/102519]: Add **`os.listdrives()`**, **`os.listvolumes()`** and **`os.listmounts()`** functions on Windows for enumerating drives, volumes and mount points

- gh-74468 [https://github.com/python/cpython/issues/74468]: Attribute name of the extracted **`tarfile`** file object now holds filename of itself rather than of the archive it is contained in. Patch by Oleg Iarygin.

- gh-102378 [https://github.com/python/cpython/issues/102378]: Private helper method `inspect._signature_strip_non_python_syntax` will no longer strip `/` from the input string.

- gh-79940 [https://github.com/python/cpython/issues/79940]: Add **`inspect.getasyncgenstate()`** and **`inspect.getasyncgenlocals()`**. Patch by Thomas Krennwallner.

- gh-102103 [https://github.com/python/cpython/issues/102103]: Add `module` argument to **`dataclasses.make_dataclass()`** and make classes produced by it pickleable.

- gh-102069 [https://github.com/python/cpython/issues/102069]: Fix `__weakref__` descriptor generation for custom dataclasses.

- gh-102038 [https://github.com/python/cpython/issues/102038]: Skip a `stat` in **`site`** if we have already found a `pyvenv.cfg`

- gh-98886 [https://github.com/python/cpython/issues/98886]: Fix issues when defining dataclasses that have fields with specific underscore names that aren’t clearly reserved by **`dataclasses`**.

- gh-101673 [https://github.com/python/cpython/issues/101673]: Fix a **`pdb`** bug where `ll` clears the changes to local variables.

- gh-101313 [https://github.com/python/cpython/issues/101313]: Added -h and –help arguments to the webbrowser CLI

- gh-100372 [https://github.com/python/cpython/issues/100372]: **`ssl.SSLContext.load_verify_locations()`** no longer incorrectly accepts some cases of trailing data when parsing DER.

- gh-89727 [https://github.com/python/cpython/issues/89727]: Fix pathlib.Path.walk RecursionError on deep directory trees by rewriting it using iteration instead of recursion.

- gh-100131 [https://github.com/python/cpython/issues/100131]: Added an optional `delete` keyword argument to **`tempfile.TemporaryDirectory`**.

- gh-48330 [https://github.com/python/cpython/issues/48330]: Added `--durations` command line option, showing the N slowest test cases. **`unittest.TextTestRunner`** and **`unittest.TextTestResult`** constructors accept a new *durations* keyword argument. Subclasses should take this into account or accept `**kwargs`. Added **`unittest.TestResult.addDuration()`** method and **`unittest.TestResult.collectedDurations`** attribute.

  (Contributed by Giampaolo Rodola)

- gh-98169 [https://github.com/python/cpython/issues/98169]: Fix **`dataclasses.astuple()`** crash when **`collections.defaultdict`** is present in the attributes.

- gh-96931 [https://github.com/python/cpython/issues/96931]: Fix incorrect results from **`ssl.SSLSocket.shared_ciphers()`**

- gh-95495 [https://github.com/python/cpython/issues/95495]: When built against OpenSSL 3.0, the **`ssl`** module had a bug where it reported unauthenticated EOFs (i.e. without close_notify) as a clean TLS-level EOF. It now raises **`SSLEOFError`**, matching the behavior in previous versions of OpenSSL. The **`options`** attribute on **`SSLContext`** also no longer includes **`OP_IGNORE_UNEXPECTED_EOF`** by default. This option may be set to specify the previous OpenSSL 3.0 behavior.

- gh-94684 [https://github.com/python/cpython/issues/94684]: Now **`uuid.uuid3()`** and **`uuid.uuid5()`** functions support **`bytes`** objects as their *name* argument.

- gh-94440 [https://github.com/python/cpython/issues/94440]: Fix a **`concurrent.futures.process`** bug where `ProcessPoolExecutor` shutdown could hang after a future has been quickly submitted and canceled.

- gh-72346 [https://github.com/python/cpython/issues/72346]: Added deprecation warning to *isdst* parameter of **`email.utils.localtime()`**.

- bpo-36305 [https://bugs.python.org/issue?@action=redirect&bpo=36305]: Fix handling of Windows filenames that resemble drives, such as `./a:b`, in **`pathlib`**.

Documentation

- gh-103112 [https://github.com/python/cpython/issues/103112]: Add docstring to **`http.client.HTTPResponse.read()`** to fix `pydoc` output.

Tests

- gh-102980 [https://github.com/python/cpython/issues/102980]: Improve test coverage on `pdb`.
- gh-102537 [https://github.com/python/cpython/issues/102537]: Adjust the error handling strategy in `test_zoneinfo.TzPathTest.python_tzpath_context`. Patch by Paul Ganssle.
- gh-101377 [https://github.com/python/cpython/issues/101377]: Improved test_locale_calendar_formatweekday of calendar.

Build

- gh-102973 [https://github.com/python/cpython/issues/102973]: Add a dev container (along with accompanying Dockerfile) for development purposes.
- gh-102711 [https://github.com/python/cpython/issues/102711]: Fix `-Wstrict-prototypes` compiler warnings.

Windows

- gh-102690 [https://github.com/python/cpython/issues/102690]: Update `webbrowser` to fall back to Microsoft Edge instead of Internet Explorer.
- gh-99726 [https://github.com/python/cpython/issues/99726]: Improves correctness of stat results for Windows, and uses faster API when available

Tools/Demos

- gh-102809 [https://github.com/python/cpython/issues/102809]: `Misc/gdbinit` was removed.

C API

- gh-102013 [https://github.com/python/cpython/issues/102013]: Add a new (unstable) C-API function for iterating over GC'able objects using a callback: `PyUnstable_VisitObjects`.

Python 3.12.0 alpha 6

Release date: 2023-03-07

Security

- gh-99108 [https://github.com/python/cpython/issues/99108]: Replace builtin hashlib implementations of MD5 and SHA1 with verified ones from the HACL* project.
- gh-101727 [https://github.com/python/cpython/issues/101727]: Updated the OpenSSL version used in Windows and macOS binary release builds to 1.1.1t to address **CVE 2023-0286** [https://www.cve.org/CVERecord?id=CVE-2023-0286], **CVE 2022-4303** [https://www.cve.org/CVERecord?id=CVE-2022-4303], and **CVE 2022-4303** [https://www.cve.org/CVERecord?id=CVE-2022-4303] per the OpenSSL 2023-02-07 security advisory [https://openssl-library.org/news/secadv/20230207.txt].
- gh-99108 [https://github.com/python/cpython/issues/99108]: Replace the builtin **`hashlib`** implementations of SHA2-384 and SHA2-512 originally from LibTomCrypt with formally verified, side-channel resistant code from the HACL* [https://github.com/hacl-star/hacl-star/] project. The builtins remain a fallback only used when OpenSSL does not provide them.
- gh-101283 [https://github.com/python/cpython/issues/101283]: **`subprocess.Popen`** now uses a safer approach to find `cmd.exe` when launching with `shell=True`. Patch by Eryk Sun, based on a patch by Oleg Iarygin.

Core and Builtins

- gh-102493 [https://github.com/python/cpython/issues/102493]: Fix regression in semantics of normalisation in `PyErr_SetObject`.
- gh-102416 [https://github.com/python/cpython/issues/102416]: Do not memoize incorrectly automatically generated loop rules in the parser. Patch by Pablo Galindo.
- gh-102356 [https://github.com/python/cpython/issues/102356]: Fix a bug that caused a crash when deallocating deeply nested filter objects. Patch by Marta Gómez Macías.
- gh-102336 [https://github.com/python/cpython/issues/102336]: Cleanup Windows 7 specific special handling. Patch by Max Bachmann.
- gh-102250 [https://github.com/python/cpython/issues/102250]: Fixed a segfault occurring when the interpreter calls a `__bool__` method that raises.
- gh-102126 [https://github.com/python/cpython/issues/102126]: Fix deadlock at shutdown when clearing thread states if any finalizer tries to acquire the runtime head lock. Patch by Kumar Aditya.
- gh-102027 [https://github.com/python/cpython/issues/102027]: Use `GetCurrentProcessId` on Windows when `getpid` is unavailable. Patch by Max Bachmann.
- gh-102056 [https://github.com/python/cpython/issues/102056]: Fix error handling bugs in interpreter’s exception printing code, which could cause a crash on infinite recursion.
- gh-100982 [https://github.com/python/cpython/issues/100982]: Restrict the scope of the **`FOR_ITER_RANGE`** instruction to the scope of the original **`FOR_ITER`** instruction, to allow instrumentation.
- gh-101967 [https://github.com/python/cpython/issues/101967]: Fix possible segfault in `positional_only_passed_as_keyword` function, when new list created.
- gh-101952 [https://github.com/python/cpython/issues/101952]: Fix possible segfault in `BUILD_SET` opcode, when new set created.

- gh-74895 [https://github.com/python/cpython/issues/74895]: **`socket.getaddrinfo`** no longer raises **`OverflowError`** for **`int`** **port** values outside of the C long range. Out of range values are left up to the underlying string based C library API to report. A **`socket.gaierror`** `SAI_SERVICE` may occur instead, or no error at all as not all platform C libraries generate an error.
- gh-101799 [https://github.com/python/cpython/issues/101799]: Add **`CALL_INTRINSIC_2`** and use it instead of **`PREP_RERAISE_STAR`**.
- gh-101857 [https://github.com/python/cpython/issues/101857]: Fix xattr support detection on Linux systems by widening the check to linux, not just glibc. This fixes support for musl.
- gh-84783 [https://github.com/python/cpython/issues/84783]: Make the slice object hashable. Patch by Will Bradshaw and Furkan Onder.
- gh-87849 [https://github.com/python/cpython/issues/87849]: Change the `SEND` instruction to leave the receiver on the stack. This allows the specialized form of `SEND` to skip the chain of C calls and jump directly to the `RESUME` in the generator or coroutine.
- gh-101765 [https://github.com/python/cpython/issues/101765]: Fix SystemError / segmentation fault in iter `__reduce__` when internal access of `builtins.__dict__` keys mutates the iter object.
- gh-101430 [https://github.com/python/cpython/issues/101430]: Update **`tracemalloc`** to handle presize of object properly. Patch by Donghee Na.
- gh-101696 [https://github.com/python/cpython/issues/101696]: Invalidate type version tag in `_PyStaticType_Dealloc` for static types, avoiding bug where a false cache hit could crash the interpreter. Patch by Kumar Aditya.
- gh-101632 [https://github.com/python/cpython/issues/101632]: Adds a new **`RETURN_CONST`** instruction.
- gh-100719 [https://github.com/python/cpython/issues/100719]: Remove gi_code field from generator (and coroutine and

async generator) objects as it is redundant. The frame already includes a reference to the code object.

- gh-98627 [https://github.com/python/cpython/issues/98627]: When an interpreter is configured to check (and only then), importing an extension module will now fail when the extension does not support multiple interpreters (i.e. doesn't implement PEP 489 multi-phase init). This does not apply to the main interpreter, nor to subinterpreters created with `Py_NewInterpreter()`.

Library

- gh-102302 [https://github.com/python/cpython/issues/102302]: Micro-optimise hashing of **`inspect.Parameter`**, reducing the time it takes to hash an instance by around 40%.

- gh-101979 [https://github.com/python/cpython/issues/101979]: Fix a bug where parentheses in the `metavar` argument to **`argparse.ArgumentParser.add_argument()`** were dropped. Patch by Yeojin Kim.

- gh-91038 [https://github.com/python/cpython/issues/91038]: **`platform.platform()`** now has boolean default arguments.

- gh-81652 [https://github.com/python/cpython/issues/81652]: Add **`mmap.MAP_ALIGNED_SUPER`** FreeBSD and **`mmap.MAP_CONCEAL`** OpenBSD constants to **`mmap`**. Patch by Yeojin Kim.

- gh-102179 [https://github.com/python/cpython/issues/102179]: Fix **`os.dup2()`** error message for negative fds.

- gh-101961 [https://github.com/python/cpython/issues/101961]: For the binary mode, **`fileinput.hookcompressed()`** doesn't set the `encoding` value even if the value is `None`. Patch by Gihwan Kim.

- gh-101936 [https://github.com/python/cpython/issues/101936]: The default value of `fp` becomes **`io.BytesIO`** if **`HTTPError`** is initialized without a designated `fp` parameter. Patch by Long Vo.

- gh-101566 [https://github.com/python/cpython/issues/101566]: In zipfile, sync Path with zipp 3.14 [https://zipp.readthedocs.io/en/latest/history.html#v3-14-0], including fix for extractall on the underlying zipfile after being wrapped in `Path`.

- gh-97930 [https://github.com/python/cpython/issues/97930]: Apply changes from importlib_resources 5.12 [https://importlib-resources.readthedocs.io/en/latest/history.html#v5-12-0], including fix for `MultiplexedPath` to support directories in multiple namespaces (python/importlib_resources#265).

- gh-101997 [https://github.com/python/cpython/issues/101997]: Upgrade pip wheel bundled with ensurepip (pip 23.0.1)

- gh-99108 [https://github.com/python/cpython/issues/99108]: The built-in extension modules for **`hashlib`** SHA2 algorithms, used when OpenSSL does not provide them, now live in a single internal `_sha2` module instead of separate `_sha256` and `_sha512` modules.

- gh-101892 [https://github.com/python/cpython/issues/101892]: Callable iterators no longer raise **`SystemError`** when the callable object exhausts the iterator but forgets to either return a sentinel value or raise **`StopIteration`**.

- gh-87634 [https://github.com/python/cpython/issues/87634]: Remove locking behavior from **`functools.cached_property()`**.

- gh-97786 [https://github.com/python/cpython/issues/97786]: Fix potential undefined behaviour in corner cases of floating-point-to-time conversions.

- gh-101517 [https://github.com/python/cpython/issues/101517]: Fixed bug where **`bdb`** looks up the source line with **`linecache`** with a `lineno=None`, which causes it to fail with an unhandled exception.
- gh-101773 [https://github.com/python/cpython/issues/101773]: Optimize **`fractions.Fraction`** for small components. The private argument `_normalize` of the **`fractions.Fraction`** constructor has been removed.
- gh-101693 [https://github.com/python/cpython/issues/101693]: In **`sqlite3.Cursor.execute()`**, **`DeprecationWarning`** is now emitted when named placeholders are used together with parameters supplied as a sequence instead of as a **`dict`**. Starting from Python 3.14, using named placeholders with parameters supplied as a sequence will raise a **`ProgrammingError`**. Patch by Erlend E. Aasland.
- gh-101446 [https://github.com/python/cpython/issues/101446]: Change repr of **`collections.OrderedDict`** to use regular dictionary formatting instead of pairs of keys and values.
- gh-101362 [https://github.com/python/cpython/issues/101362]: Speed up **`pathlib.PurePath`** construction by handling arguments more uniformly. When a **`pathlib.Path`** argument is supplied, we use its string representation rather than joining its parts with **`os.path.join()`**.
- gh-101362 [https://github.com/python/cpython/issues/101362]: Speed up **`pathlib.PurePath`** construction by calling **`os.path.join()`** only when two or more arguments are given.

- gh-101362 [https://github.com/python/cpython/issues/101362]: Speed up **`pathlib.Path`** construction by running the path flavour compatibility check only when pathlib is imported.

- gh-85984 [https://github.com/python/cpython/issues/85984]: Refactored the implementation of **`pty.fork()`** to use **`os.login_tty()`**.

  A **`DeprecationWarning`** is now raised by `pty.master_open()` and `pty.slave_open()`. They were undocumented and deprecated long long ago in the docstring in favor of **`pty.openpty()`**.

- gh-101561 [https://github.com/python/cpython/issues/101561]: Add a new decorator **`typing.override()`**. See **PEP 698** [https://peps.python.org/pep-0698/] for details. Patch by Steven Troxler.

- gh-63301 [https://github.com/python/cpython/issues/63301]: Set exit code when **`tabnanny`** CLI exits on error.

- gh-101360 [https://github.com/python/cpython/issues/101360]: Fix anchor matching in **`pathlib.PureWindowsPath.match()`**. Path and pattern anchors are now matched with **`fnmatch`**, just like other path parts. This allows patterns such as `"*:/Users/*"` to be matched.

- gh-101277 [https://github.com/python/cpython/issues/101277]: Remove global state from **`itertools`** module (**PEP 687** [https://peps.python.org/pep-0687/]). Patches by Erlend E. Aasland.

- gh-100809 [https://github.com/python/cpython/issues/100809]: Fix handling of drive-relative paths (like ‘C:’ and ‘C:foo’) in **`pathlib.Path.absolute()`**. This method now uses the OS API to retrieve the correct current working directory for the drive.

- gh-99138 [https://github.com/python/cpython/issues/99138]: Apply **PEP 687** [https://peps.python.org/pep-0687/] to `zoneinfo`. Patch by Erlend E. Aasland.

- gh-96764 [https://github.com/python/cpython/issues/96764]: `asyncio.wait_for()` now uses `asyncio.timeout()` as its underlying implementation. Patch by Kumar Aditya.

- gh-88233 [https://github.com/python/cpython/issues/88233]: Correctly preserve “extra” fields in `zipfile` regardless of their ordering relative to a zip64 “extra.”

- bpo-23224 [https://bugs.python.org/issue?@action=redirect&bpo=23224]: Fix segfaults when creating `lzma.LZMADecompressor` and `bz2.BZ2Decompressor` objects without calling `__init__()`, and fix leakage of locks and internal buffers when calling the `__init__()` methods of `lzma.LZMADecompressor`, `lzma.LZMACompressor`, `bz2.BZ2Compressor`, and `bz2.BZ2Decompressor` objects multiple times.

Documentation

- gh-85417 [https://github.com/python/cpython/issues/85417]: Update `cmath` documentation to clarify behaviour on branch cuts.
- gh-97725 [https://github.com/python/cpython/issues/97725]: Fix `asyncio.Task.print_stack()` description for `file=None`. Patch by Oleg Iarygin.

Tests

- gh-102019 [https://github.com/python/cpython/issues/102019]: Fix deadlock on shutdown if `test_current_{exception,frames}` fails. Patch by Jacob Bower.

- gh-85984 [https://github.com/python/cpython/issues/85984]: Utilize new “winsize” functions from termios in pty tests.
- gh-89792 [https://github.com/python/cpython/issues/89792]: `test_tools` now copies up to 10x less source data to a temporary directory during the `freeze` test by ignoring git metadata and other artifacts. It also limits its python build parallelism based on os.cpu_count instead of hard coding it as 8 cores.

Build

- gh-99942 [https://github.com/python/cpython/issues/99942]: On Android, in a static build, python-config in embed mode no longer incorrectly reports a library to link to.
- gh-99942 [https://github.com/python/cpython/issues/99942]: On Android, python.pc now correctly reports the library to link to, the same as python-config.sh.
- gh-100221 [https://github.com/python/cpython/issues/100221]: Fix creating install directories in `make sharedinstall` if they exist outside `DESTDIR` already.
- gh-96821 [https://github.com/python/cpython/issues/96821]: Explicitly mark C extension modules that need defined signed integer overflow, and add a configure option **`--with-strict-overflow`**. Patch by Matthias Görgens and Shantanu Jain.

Windows

- gh-102344 [https://github.com/python/cpython/issues/102344]: Implement `winreg.QueryValue` using `QueryValueEx` and `winreg.SetValue` using `SetValueEx`. Patch by Max Bachmann.
- gh-101881 [https://github.com/python/cpython/issues/101881]: Handle read and write operations on non-blocking pipes properly on Windows.

- gh-101881 [https://github.com/python/cpython/issues/101881]: Add support for the os.get_blocking() and os.set_blocking() functions on Windows.
- gh-101849 [https://github.com/python/cpython/issues/101849]: Ensures installer will correctly upgrade existing `py.exe` launcher installs.
- gh-101763 [https://github.com/python/cpython/issues/101763]: Updates copy of libffi bundled with Windows installs to 3.4.4.
- gh-101759 [https://github.com/python/cpython/issues/101759]: Update Windows installer to SQLite 3.40.1.
- gh-101614 [https://github.com/python/cpython/issues/101614]: Correctly handle extensions built against debug binaries that reference `python3_d.dll`.
- gh-101196 [https://github.com/python/cpython/issues/101196]: The functions `os.path.isdir`, `os.path.isfile`, `os.path.islink` and `os.path.exists` are now 13% to 28% faster on Windows, by making fewer Win32 API calls.

macOS

- gh-101759 [https://github.com/python/cpython/issues/101759]: Update macOS installer to SQLite 3.40.1.

C API

- gh-101907 [https://github.com/python/cpython/issues/101907]: Removes use of non-standard C++ extension in public header files.

- gh-99293 [https://github.com/python/cpython/issues/99293]: Document that the Py_TPFLAGS_VALID_VERSION_TAG is an internal feature, should not be used, and will be removed.

- gh-101578 [https://github.com/python/cpython/issues/101578]: Add `PyErr_GetRaisedException()` and `PyErr_SetRaisedException()` for saving and restoring the current exception. These functions return and accept a single exception object, rather than the triple arguments of the now-deprecated `PyErr_Fetch()` and `PyErr_Restore()`. This is less error prone and a bit more efficient.

  Add `PyException_GetArgs()` and `PyException_SetArgs()` as convenience functions for retrieving and modifying the `args` passed to the exception's constructor.

- gh-91744 [https://github.com/python/cpython/issues/91744]: Introduced the *Unstable C API tier*, marking APi that is allowed to change in minor releases without a deprecation period. See **PEP 689** [https://peps.python.org/pep-0689/] for details.

Python 3.12.0 alpha 5

Release date: 2023-02-07

Security

- gh-99108 [https://github.com/python/cpython/issues/99108]: Replace the builtin `hashlib` implementations of SHA2-224 and SHA2-256 originally from LibTomCrypt with formally verified, side-channel resistant code from the HACL* [https://github.com/hacl-star/hacl-star/] project. The builtins remain a fallback only used when OpenSSL does not provide them.

Core and Builtins

- gh-92173 [https://github.com/python/cpython/issues/92173]: Fix the `defs` and `kwdefs` arguments to **`PyEval_EvalCodeEx()`** and a reference leak in that function.
- gh-59956 [https://github.com/python/cpython/issues/59956]: The GILState API is now partially compatible with subinterpreters. Previously, `PyThreadState_GET()` and `PyGILState_GetThisThreadState()` would get out of sync, causing inconsistent behavior and crashes.
- gh-101400 [https://github.com/python/cpython/issues/101400]: Fix wrong lineno in exception message on **`continue`** or **`break`** which are not in a loop. Patch by Donghee Na.
- gh-101372 [https://github.com/python/cpython/issues/101372]: Fix **`is_normalized()`** to properly handle the UCD 3.2.0 cases. Patch by Donghee Na.
- gh-101266 [https://github.com/python/cpython/issues/101266]: Fix **`sys.getsizeof()`** reporting for **`int`** subclasses.
- gh-101291 [https://github.com/python/cpython/issues/101291]: Refactor the `PyLongObject` struct into a normal Python object header and a `PyLongValue` struct.
- gh-101046 [https://github.com/python/cpython/issues/101046]: Fix a possible memory leak in the parser when raising **`MemoryError`**. Patch by Pablo Galindo
- gh-101037 [https://github.com/python/cpython/issues/101037]: Fix potential memory underallocation issue for instances of **`int`** subclasses with value zero.
- gh-100762 [https://github.com/python/cpython/issues/100762]: Record the (virtual) exception block depth in the oparg of **`YIELD_VALUE`**. Use this to avoid the expensive `throw()` when closing generators (and coroutines) that can be closed trivially.
- gh-100982 [https://github.com/python/cpython/issues/100982]: Adds a new **`COMPARE_AND_BRANCH`** instruction. This is a bit more efficient when performing a comparison immediately followed by a branch, and restores the design intent of PEP 659 that specializations are local to a single instruction.

- gh-100942 [https://github.com/python/cpython/issues/100942]: Fixed segfault in property.getter/setter/deleter that occurred when a property subclass overrode the `__new__` method to return a non-property instance.
- gh-100923 [https://github.com/python/cpython/issues/100923]: Remove the `mask` cache entry for the **`COMPARE_OP`** instruction and embed the mask into the oparg.
- gh-100892 [https://github.com/python/cpython/issues/100892]: Fix race while iterating over thread states in clearing **`threading.local`**. Patch by Kumar Aditya.
- gh-91351 [https://github.com/python/cpython/issues/91351]: Fix a case where re-entrant imports could corrupt the import deadlock detection code and cause a **`KeyError`** to be raised out of **`importlib/_bootstrap`**. In addition to the straightforward cases, this could also happen when garbage collection leads to a warning being emitted – as happens when it collects an open socket or file)
- gh-100726 [https://github.com/python/cpython/issues/100726]: Optimize construction of `range` object for medium size integers.
- gh-100712 [https://github.com/python/cpython/issues/100712]: Added option to build cpython with specialization disabled, by setting `ENABLE_SPECIALIZATION=False` in **`opcode`**, followed by `make regen-all`.
- bpo-32780 [https://bugs.python.org/issue?@action=redirect&bpo=32780]: Inter-field padding is now inserted into the PEP3118 format strings obtained from **`ctypes.Structure`** objects, reflecting their true representation in memory.

Library

- gh-101541 [https://github.com/python/cpython/issues/101541]: [Enum] - fix psuedo-flag creation
- gh-101570 [https://github.com/python/cpython/issues/101570]: Upgrade pip wheel bundled with ensurepip (pip 23.0)

- gh-101323 [https://github.com/python/cpython/issues/101323]: Fix a bug where errors where not thrown by zlib._ZlibDecompressor if encountered during decompressing.
- gh-101317 [https://github.com/python/cpython/issues/101317]: Add *ssl_shutdown_timeout* parameter for **`asyncio.StreamWriter.start_tls()`**.
- gh-101326 [https://github.com/python/cpython/issues/101326]: Fix regression when passing `None` as second or third argument to `FutureIter.throw`.
- gh-92123 [https://github.com/python/cpython/issues/92123]: Adapt the `_elementtree` extension module to multi-phase init (**PEP 489** [https://peps.python.org/pep-0489/]). Patches by Erlend E. Aasland.
- gh-100795 [https://github.com/python/cpython/issues/100795]: Avoid potential unexpected `freeaddrinfo` call (double free) in **`socket`** when when a libc `getaddrinfo()` implementation leaves garbage in an output pointer when returning an error. Original patch by Sergey G. Brester.
- gh-101143 [https://github.com/python/cpython/issues/101143]: Remove unused references to **`TimerHandle`** in `asyncio.base_events.BaseEventLoop._add_callback`.
- gh-101144 [https://github.com/python/cpython/issues/101144]: Make **`zipfile.Path.open()`** and **`zipfile.Path.read_text()`** also accept `encoding` as a positional argument. This was the behavior in Python 3.9 and earlier. 3.10 introduced a regression where supplying it as a positional argument would lead to a **`TypeError`**.
- gh-94518 [https://github.com/python/cpython/issues/94518]: Group-related variables of `_posixsubprocess` module are renamed to stress that supplementary group affinity is added to a fork, not replace the inherited ones. Patch by Oleg Iarygin.
- gh-101015 [https://github.com/python/cpython/issues/101015]: Fix **`typing.get_type_hints()`** on `'*tuple[...]'` and `*tuple[...]`. It must not drop the `Unpack` part.

- gh-101000 [https://github.com/python/cpython/issues/101000]: Add **`os.path.splitroot()`**, which splits a path into a 3-item tuple `(drive, root, tail)`. This new function is used by **`pathlib`** to improve the performance of path construction by up to a third.
- gh-100573 [https://github.com/python/cpython/issues/100573]: Fix a Windows **`asyncio`** bug with named pipes where a client doing `os.stat()` on the pipe would cause an error in the server that disabled serving future requests.
- gh-39615 [https://github.com/python/cpython/issues/39615]: **`warnings.warn()`** now has the ability to skip stack frames based on code filename prefix rather than only a numeric `stacklevel` via the new `skip_file_prefixes` keyword argument.
- gh-100750 [https://github.com/python/cpython/issues/100750]: pass encoding kwarg to subprocess in platform
- gh-100160 [https://github.com/python/cpython/issues/100160]: Emit a deprecation warning in **`asyncio.DefaultEventLoopPolicy.get_event_loop()`** if there is no current event loop set and it decides to create one.
- gh-96290 [https://github.com/python/cpython/issues/96290]: Fix handling of partial and invalid UNC drives in `ntpath.splitdrive()`, and in `ntpath.normpath()` on non-Windows systems. Paths such as ‘\server’ and ‘\’ are now considered by `splitdrive()` to contain only a drive, and consequently are not modified by `normpath()` on non-Windows systems. The behaviour of `normpath()` on Windows systems is unaffected, as native OS APIs are used. Patch by Eryk Sun, with contributions by Barney Gale.
- gh-99952 [https://github.com/python/cpython/issues/99952]: Fix a reference undercounting issue in **`ctypes.Structure`** with `from_param()` results larger than a C pointer.
- gh-67790 [https://github.com/python/cpython/issues/67790]: Add float-style formatting support for **`fractions.Fraction`** instances.

- gh-99266 [https://github.com/python/cpython/issues/99266]: Preserve more detailed error messages in **`ctypes`**.
- gh-86682 [https://github.com/python/cpython/issues/86682]: Ensure runtime-created collections have the correct module name using the newly added (internal) **`sys._getframemodulename()`**.
- gh-88597 [https://github.com/python/cpython/issues/88597]: **`uuid`** now has a command line interface. Try `python -m uuid -h`.
- gh-60580 [https://github.com/python/cpython/issues/60580]: **`ctypes.wintypes.BYTE`** definition changed from **`c_byte`** to **`c_ubyte`** to match Windows SDK. Patch by Anatoly Techtonik and Oleg Iarygin.
- gh-94518 [https://github.com/python/cpython/issues/94518]: `_posixsubprocess` now initializes all UID and GID variables using a reserved `-1` value instead of a separate flag. Patch by Oleg Iarygin.
- bpo-38941 [https://bugs.python.org/issue?@action=redirect&bpo=38941]: The **`xml.etree.ElementTree`** module now emits **`DeprecationWarning`** when testing the truth value of an **`xml.etree.ElementTree.Element`**. Before, the Python implementation emitted **`FutureWarning`**, and the C implementation emitted nothing.
- bpo-40077 [https://bugs.python.org/issue?@action=redirect&bpo=40077]: Convert **`elementtree`** types to heap types. Patch by Erlend E. Aasland.
- bpo-29847 [https://bugs.python.org/issue?@action=redirect&bpo=29847]: Fix a bug where **`pathlib.Path`** accepted and ignored keyword arguments. Patch provided by Yurii Karabas.
- gh-77772 [https://github.com/python/cpython/issues/77772]: **`ctypes.CDLL`**, **`ctypes.OleDLL`**, **`ctypes.WinDLL`**, and **`ctypes.PyDLL`** now accept path-like objects as their `name` argument. Patch by Robert Hoelzl.

Documentation

- gh-88324 [https://github.com/python/cpython/issues/88324]: Reword **`subprocess`** to emphasize default behavior of *stdin*, *stdout*, and *stderr* arguments. Remove inaccurate statement about child file handle inheritance.

Tests

- gh-101334 [https://github.com/python/cpython/issues/101334]: `test_tarfile` has been updated to pass when run as a high UID.

Build

- gh-101282 [https://github.com/python/cpython/issues/101282]: Update BOLT configuration not to use deprecated usage of `--split functions`. Patch by Donghee Na.
- gh-101522 [https://github.com/python/cpython/issues/101522]: Allow overriding Windows dependencies versions and paths using MSBuild properties.
- gh-77532 [https://github.com/python/cpython/issues/77532]: Minor fixes to allow building with `PlatformToolset=ClangCL` on Windows.
- gh-101152 [https://github.com/python/cpython/issues/101152]: In accordance with **PEP 699** [https://peps.python.org/pep-0699/], the `ma_version_tag` field in **`PyDictObject`** is deprecated for extension modules. Accessing this field will generate a compiler warning at compile time. This field will be removed in Python 3.14.
- gh-100340 [https://github.com/python/cpython/issues/100340]: Allows -Wno-int-conversion for wasm-sdk 17 and onwards, thus enables building WASI builds once against the latest sdk.
- gh-101060 [https://github.com/python/cpython/issues/101060]: Conditionally add `-fno-reorder-blocks-and-partition` in configure. Effectively fixes `--enable-bolt` when using Clang, as this appears to be a GCC-only flag.

- gh-98705 [https://github.com/python/cpython/issues/98705]: `__bool__` is defined in AIX system header files which breaks the build in AIX, so undefine it.
- gh-98636 [https://github.com/python/cpython/issues/98636]: Fix a regression in detecting `gdbm_compat` library for the `_gdbm` module build.
- gh-96305 [https://github.com/python/cpython/issues/96305]: `_aix_support` now uses a simple code to get platform details rather than the now non-existent `_bootsubprocess` during bootstrap.

Windows

- gh-101543 [https://github.com/python/cpython/issues/101543]: Ensure the install path in the registry is only used when the standard library hasn't been located in any other way.
- gh-101467 [https://github.com/python/cpython/issues/101467]: The `py.exe` launcher now correctly filters when only a single runtime is installed. It also correctly handles prefix matches on tags so that `-3.1` does not match `3.11`, but would still match `3.1-32`.
- gh-99834 [https://github.com/python/cpython/issues/99834]: Updates bundled copy of Tcl/Tk to 8.6.13.0
- gh-101135 [https://github.com/python/cpython/issues/101135]: Restore ability to launch older 32-bit versions from the `py.exe` launcher when both 32-bit and 64-bit installs of the same version are available.
- gh-82052 [https://github.com/python/cpython/issues/82052]: Fixed an issue where writing more than 32K of Unicode output to the console screen in one go can result in mojibake.
- gh-100320 [https://github.com/python/cpython/issues/100320]: Ensures the `PythonPath` registry key from an install is used when launching from a different copy of Python that relies on an existing install to provide a copy of its modules and standard library.

- gh-100247 [https://github.com/python/cpython/issues/100247]: Restores support for the `py.exe` launcher finding shebang commands in its configuration file using the full command name.

Python 3.12.0 alpha 4

Release date: 2023-01-10

Core and Builtins

- gh-100776 [https://github.com/python/cpython/issues/100776]: Fix misleading default value in **`input()`**'s `__text_signature__`.

- gh-99005 [https://github.com/python/cpython/issues/99005]: Remove **`UNARY_POSITIVE`**, **`ASYNC_GEN_WRAP`** and **`LIST_TO_TUPLE`**, replacing them with intrinsics.

- gh-99005 [https://github.com/python/cpython/issues/99005]: Add new **`CALL_INTRINSIC_1`** instruction. Remove **`IMPORT_STAR`**, **`PRINT_EXPR`** and **`STOPITERATION_ERROR`**, replacing them with the **`CALL_INTRINSIC_1`** instruction.

- gh-100288 [https://github.com/python/cpython/issues/100288]: Remove the LOAD_ATTR_METHOD_WITH_DICT specialized instruction. Stats show it is not useful.

- gh-100720 [https://github.com/python/cpython/issues/100720]: Added `_PyFrame_NumSlotsForCodeObject`, which returns the number of slots needed in a frame for a given code object.

- gh-100719 [https://github.com/python/cpython/issues/100719]: Removed the co_nplaincellvars field from the code object, as it is redundant.

- gh-100637 [https://github.com/python/cpython/issues/100637]: Fix **`int.__sizeof__()`** calculation to include the 1-element `ob_digit` array for `0` and `False`.

- gh-100649 [https://github.com/python/cpython/issues/100649]: Update the native_thread_id field of PyThreadState after fork.

- gh-100126 [https://github.com/python/cpython/issues/100126]: Fix an issue where “incomplete” frames could be briefly visible to C code while other frames are being torn down, possibly resulting in corruption or hard crashes of the interpreter while running finalizers.

- gh-87447 [https://github.com/python/cpython/issues/87447]: Fix **`SyntaxError`** on comprehension rebind checking with names that are not actually redefined.

  Now reassigning `b` in `[(b := 1) for a, b.prop in some_iter]` is allowed. Reassigning `a` is still disallowed as per **PEP 572** [https://peps.python.org/pep-0572/].

- gh-100268 [https://github.com/python/cpython/issues/100268]: Add **`int.is_integer()`** to improve duck type compatibility between **`int`** and **`float`**.

- gh-100425 [https://github.com/python/cpython/issues/100425]: Improve the accuracy of `sum()` with compensated summation.

- gh-100374 [https://github.com/python/cpython/issues/100374]: Fix incorrect result and delay in **`socket.getfqdn()`**. Patch by Dominic Socular.

- gh-100357 [https://github.com/python/cpython/issues/100357]: Convert `vars`, `dir`, `next`, `getattr`, and `iter` to argument clinic.

- gh-100117 [https://github.com/python/cpython/issues/100117]: Improve the output of `codeobject.co_lines()` by emitting only one entry for each line range.

- gh-90043 [https://github.com/python/cpython/issues/90043]: Handle NaNs when specializing `COMPARE_OP` for `float` values.

- gh-100222 [https://github.com/python/cpython/issues/100222]: Redefine the `_Py_CODEUNIT` typedef as a union to describe its layout to the C compiler, avoiding type punning and improving clarity.

- gh-99955 [https://github.com/python/cpython/issues/99955]: Internal compiler functions (in compile.c) now consistently return -1 on error and 0 on success.

- gh-100188 [https://github.com/python/cpython/issues/100188]: The `BINARY_SUBSCR_LIST_INT` and `BINARY_SUBSCR_TUPLE_INT` instructions are no longer used for negative integers because those instructions always miss when encountering negative integers.

- gh-99110 [https://github.com/python/cpython/issues/99110]: Initialize frame->previous in frameobject.c to fix a segmentation fault when accessing frames created by `PyFrame_New()`.

- gh-94155 [https://github.com/python/cpython/issues/94155]: Improved the hashing algorithm for code objects, mitigating some hash collisions.

- gh-99540 [https://github.com/python/cpython/issues/99540]: `None` now hashes to a constant value. This is not a requirements change.

- gh-100143 [https://github.com/python/cpython/issues/100143]: When built with `--enable-pystats`, stats collection is now off by default. To enable it early at startup, pass the `-Xpystats` flag. Stats are now always dumped, even if switched off.

- gh-100146 [https://github.com/python/cpython/issues/100146]: Improve `BUILD_LIST` opcode so that it works similarly to the `BUILD_TUPLE` opcode, by stealing references from the stack rather than repeatedly using stack operations to set list elements. Implementation details are in a new private API **`_PyList_FromArraySteal()`**.
- gh-100110 [https://github.com/python/cpython/issues/100110]: Specialize `FOR_ITER` for tuples.
- gh-100050 [https://github.com/python/cpython/issues/100050]: Honor existing errors obtained when searching for mismatching parentheses in the tokenizer. Patch by Pablo Galindo
- gh-92216 [https://github.com/python/cpython/issues/92216]: Improve the performance of **`hasattr()`** for type objects with a missing attribute.
- gh-99582 [https://github.com/python/cpython/issues/99582]: Freeze **`zipimport`** module into `_bootstrap_python`.
- gh-99554 [https://github.com/python/cpython/issues/99554]: Pack debugging location tables more efficiently during bytecode compilation.
- gh-98522 [https://github.com/python/cpython/issues/98522]: Add an internal version number to code objects, to give better versioning of inner functions and comprehensions, and thus better specialization of those functions. This change is invisible to both Python and C extensions.

- gh-94603 [https://github.com/python/cpython/issues/94603]: Improve performance of `list.pop` for small lists.

- gh-89051 [https://github.com/python/cpython/issues/89051]: Add **`ssl.OP_LEGACY_SERVER_CONNECT`**

- bpo-32782 [https://bugs.python.org/issue?@action=redirect&bpo=32782]: `ctypes` arrays of length 0 now report a correct itemsize when a `memoryview` is constructed from them, rather than always giving a value of 0.

Library

- gh-100833 [https://github.com/python/cpython/issues/100833]: Speed up **`math.fsum()`** by removing defensive `volatile` qualifiers.
- gh-100805 [https://github.com/python/cpython/issues/100805]: Modify **`random.choice()`** implementation to once again work with NumPy arrays.
- gh-100813 [https://github.com/python/cpython/issues/100813]: Add **`socket.IP_PKTINFO`** constant.
- gh-100792 [https://github.com/python/cpython/issues/100792]: Make **`email.message.Message.__contains__()`** twice as fast.
- gh-91851 [https://github.com/python/cpython/issues/91851]: Microoptimizations for **`fractions.Fraction.__round__()`**, **`fractions.Fraction.__ceil__()`** and **`fractions.Fraction.__floor__()`**.
- gh-90104 [https://github.com/python/cpython/issues/90104]: Avoid RecursionError on `repr` if a dataclass field definition has a cyclic reference.
- gh-100689 [https://github.com/python/cpython/issues/100689]: Fix crash in **`pyexpat`** by statically allocating `PyExpat_CAPI` capsule.
- gh-100740 [https://github.com/python/cpython/issues/100740]: Fix `unittest.mock.Mock` not respecting the spec for attribute

names prefixed with `assert`.

- gh-91219 [https://github.com/python/cpython/issues/91219]: Change `SimpleHTTPRequestHandler` to support subclassing to provide a different set of index file names instead of using `__init__` parameters.
- gh-100690 [https://github.com/python/cpython/issues/100690]: `Mock` objects which are not unsafe will now raise an `AttributeError` when accessing an attribute that matches the name of an assertion but without the prefix `assert_`, e.g. accessing `called_once` instead of `assert_called_once`. This is in addition to this already happening for accessing attributes with prefixes `assert`, `assret`, `asert`, `aseert`, and `assrt`.
- gh-89727 [https://github.com/python/cpython/issues/89727]: Simplify and optimize **`os.walk()`** by using **`isinstance()`** checks to check the top of the stack.
- gh-100485 [https://github.com/python/cpython/issues/100485]: Add math.sumprod() to compute the sum of products.
- gh-86508 [https://github.com/python/cpython/issues/86508]: Fix **`asyncio.open_connection()`** to skip binding to local addresses of different family. Patch by Kumar Aditya.
- gh-97930 [https://github.com/python/cpython/issues/97930]: `importlib.resources.files` now accepts a module as an anchor instead of only accepting packages. If a module is passed, resources are resolved adjacent to that module (in the same package or at the package root). The parameter was renamed from `package` to `anchor` with a compatibility shim for those passing by keyword. Additionally, the new `anchor` parameter is now optional and will default to the caller’s module.
- gh-100585 [https://github.com/python/cpython/issues/100585]: Fixed a bug where importlib.resources.as_file was leaving file pointers open
- gh-100562 [https://github.com/python/cpython/issues/100562]: Improve performance of **`pathlib.Path.absolute()`** by nearly 2x. This comes at the cost of a performance regression

in `pathlib.Path.cwd()`, which is generally used less frequently in user code.

- gh-100519 [https://github.com/python/cpython/issues/100519]: Small simplification of **`http.cookiejar.eff_request_host()`** that improves readability and better matches the RFC wording.
- gh-100287 [https://github.com/python/cpython/issues/100287]: Fix the interaction of `unittest.mock.seal()` with `unittest.mock.AsyncMock`.
- gh-100488 [https://github.com/python/cpython/issues/100488]: Add **`Fraction.is_integer()`** to check whether a `fractions.Fraction` is an integer. This improves duck type compatibility with `float` and `int`.
- gh-100474 [https://github.com/python/cpython/issues/100474]: `http.server` now checks that an index page is actually a regular file before trying to serve it. This avoids issues with directories named `index.html`.
- gh-100363 [https://github.com/python/cpython/issues/100363]: Speed up `asyncio.get_running_loop()` by removing redundant `getpid` checks. Patch by Kumar Aditya.
- gh-78878 [https://github.com/python/cpython/issues/78878]: Fix crash when creating an instance of **`_ctypes.CField`**.
- gh-100348 [https://github.com/python/cpython/issues/100348]: Fix ref cycle in **`asyncio._SelectorSocketTransport`** by removing `_read_ready_cb` in `close`.
- gh-100344 [https://github.com/python/cpython/issues/100344]: Provide C implementation for `asyncio.current_task()` for a 4x-6x speedup.
- gh-100272 [https://github.com/python/cpython/issues/100272]: Fix JSON serialization of OrderedDict. It now preserves the order of keys.
- gh-83076 [https://github.com/python/cpython/issues/83076]: Instantiation of `Mock()` and `AsyncMock()` is now 3.8x faster.
- gh-100234 [https://github.com/python/cpython/issues/100234]: Set a default value of 1.0 for the `lambd` parameter in random.expovariate().

- gh-100228 [https://github.com/python/cpython/issues/100228]: A **`DeprecationWarning`** may be raised when **`os.fork()`** or **`os.forkpty()`** is called from multi-threaded processes. Forking with threads is unsafe and can cause deadlocks, crashes and subtle problems. Lack of a warning does not indicate that the fork call was actually safe, as Python may not be aware of all threads.
- gh-100039 [https://github.com/python/cpython/issues/100039]: Improve signatures for enums and flags.
- gh-100133 [https://github.com/python/cpython/issues/100133]: Fix regression in **`asyncio`** where a subprocess would sometimes lose data received from pipe.
- bpo-44592 [https://bugs.python.org/issue?@action=redirect&bpo=44592]: Fixes inconsistent handling of case sensitivity of *extrasaction* arg in **`csv.DictWriter`**.
- gh-100098 [https://github.com/python/cpython/issues/100098]: Fix `tuple` subclasses being cast to `tuple` when used as enum values.
- gh-85432 [https://github.com/python/cpython/issues/85432]: Rename the *fmt* parameter of the pure-Python implementation of **`datetime.time.strftime()`** to *format*. Rename the *t* parameter of **`datetime.datetime.fromtimestamp()`** to *timestamp*. These changes mean the parameter names in the pure-Python implementation now match the parameter names in the C implementation. Patch by Alex Waygood.
- gh-98778 [https://github.com/python/cpython/issues/98778]: Update **`HTTPError`** to be initialized properly, even if the `fp` is `None`. Patch by Donghee Na.
- gh-99925 [https://github.com/python/cpython/issues/99925]: Unify error messages in JSON serialization between `json.dumps(float('nan'), allow_nan=False)` and `json.dumps(float('nan'), allow_nan=False, indent=<SOMETHING>)`. Now both include the representation of the value that could not be serialized.

- gh-89727 [https://github.com/python/cpython/issues/89727]: Fix issue with `os.walk()` where a `RecursionError` would occur on deep directory structures by adjusting the implementation of `os.walk()` to be iterative instead of recursive.
- gh-94943 [https://github.com/python/cpython/issues/94943]: Add Dataclass support to the `Enum` `__repr__()`. When inheriting from a `dataclass`, only show the field names in the value section of the member `repr()`, and not the dataclass' class name.
- gh-83035 [https://github.com/python/cpython/issues/83035]: Fix `inspect.getsource()` handling of decorator calls with nested parentheses.
- gh-99576 [https://github.com/python/cpython/issues/99576]: Fix `.save()` method for `LWPCookieJar` and `MozillaCookieJar`: saved file was not truncated on repeated save.
- gh-94912 [https://github.com/python/cpython/issues/94912]: Add `inspect.markcoroutinefunction()` decorator which manually marks a function as a coroutine for the benefit of **`iscoroutinefunction()`**.
- gh-99509 [https://github.com/python/cpython/issues/99509]: Add **PEP 585** [https://peps.python.org/pep-0585/] support for **`multiprocessing.queues.Queue`**.
- gh-99482 [https://github.com/python/cpython/issues/99482]: Remove `Jython` partial compatibility code from several stdlib modules.
- gh-99433 [https://github.com/python/cpython/issues/99433]: Fix `doctest` failure on `types.MethodWrapperType` in modules.
- gh-85267 [https://github.com/python/cpython/issues/85267]: Several improvements to `inspect.signature()`'s handling of `__text_signature`. - Fixes a case where `inspect.signature()` dropped parameters - Fixes a case where `inspect.signature()` raised `tokenize.TokenError` - Allows `inspect.signature()` to understand defaults involving binary operations of constants - `inspect.signature()` is documented as only raising `TypeError` or `ValueError`, but

sometimes raised **`RuntimeError`**. These cases now raise **`ValueError`** - Removed a dead code path

- gh-91166 [https://github.com/python/cpython/issues/91166]: **`asyncio`** is optimized to avoid excessive copying when writing to socket and use **`sendmsg()`** if the platform supports it. Patch by Kumar Aditya.
- gh-98030 [https://github.com/python/cpython/issues/98030]: Add missing TCP socket options from Linux: `TCP_MD5SIG`, `TCP_THIN_LINEAR_TIMEOUTS`, `TCP_THIN_DUPACK`, `TCP_REPAIR`, `TCP_REPAIR_QUEUE`, `TCP_QUEUE_SEQ`, `TCP_REPAIR_OPTIONS`, `TCP_TIMESTAMP`, `TCP_CC_INFO`, `TCP_SAVE_SYN`, `TCP_SAVED_SYN`, `TCP_REPAIR_WINDOW`, `TCP_FASTOPEN_CONNECT`, `TCP_ULP`, `TCP_MD5SIG_EXT`, `TCP_FASTOPEN_KEY`, `TCP_FASTOPEN_NO_COOKIE`, `TCP_ZEROCOPY_RECEIVE`, `TCP_INQ`, `TCP_TX_DELAY`.
- gh-88500 [https://github.com/python/cpython/issues/88500]: Reduced the memory usage of **`urllib.parse.unquote()`** and **`urllib.parse.unquote_to_bytes()`** on large values.
- gh-96127 [https://github.com/python/cpython/issues/96127]: `inspect.signature` was raising `TypeError` on call with mock objects. Now it correctly returns `(*args, **kwargs)` as inferred signature.
- gh-95882 [https://github.com/python/cpython/issues/95882]: Fix a 3.11 regression in **`asynccontextmanager()`**, which caused it to propagate exceptions with incorrect tracebacks and fix a 3.11 regression in **`contextmanager()`**, which caused it to propagate exceptions with incorrect tracebacks for **`StopIteration`**.
- gh-78707 [https://github.com/python/cpython/issues/78707]: Deprecate passing more than one positional argument to **`pathlib.PurePath.relative_to()`** and **`is_relative_to()`**.
- gh-92122 [https://github.com/python/cpython/issues/92122]: Fix reStructuredText syntax errors in docstrings in the **`enum`** module.
- gh-91851 [https://github.com/python/cpython/issues/91851]: Optimize the **`Fraction`** arithmetics for small components.

- bpo-24132 [https://bugs.python.org/issue?@action=redirect&bpo=24132]: Make **`pathlib.PurePath`** and **`Path`** subclassable (private to start). Previously, attempting to instantiate a subclass resulted in an **`AttributeError`** being raised. Patch by Barney Gale.
- bpo-40447 [https://bugs.python.org/issue?@action=redirect&bpo=40447]: Accept **`os.PathLike`** (such as **`pathlib.Path`**) in the `stripdir` arguments of **`compileall.compile_file()`** and **`compileall.compile_dir()`**.
- bpo-36880 [https://bugs.python.org/issue?@action=redirect&bpo=36880]: Fix a reference counting issue when a **`ctypes`** callback with return type **`py_object`** returns `None`, which could cause crashes.

Documentation

- gh-100616 [https://github.com/python/cpython/issues/100616]: Document existing `attr` parameter to **`curses.window.vline()`** function in **`curses`**.
- gh-100472 [https://github.com/python/cpython/issues/100472]: Remove claim in documentation that the `stripdir`, `prependdir` and `limit_sl_dest` parameters of **`compileall.compile_dir()`** and **`compileall.compile_file()`** could be **`bytes`**.
- bpo-25377 [https://bugs.python.org/issue?@action=redirect&bpo=25377]: Clarify use of octal format of mode argument in help(os.chmod) as well as help(os.fchmod)

Tests

- gh-100454 [https://github.com/python/cpython/issues/100454]: Start running SSL tests with OpenSSL 3.1.0-beta1.
- gh-100086 [https://github.com/python/cpython/issues/100086]: The Python test runner (libregrtest) now logs Python build

information like “debug” vs “release” build, or LTO and PGO optimizations. Patch by Victor Stinner.
- gh-93018 [https://github.com/python/cpython/issues/93018]: Make two tests forgiving towards host system libexpat with backported security fixes applied.

Build

- gh-100540 [https://github.com/python/cpython/issues/100540]: Removed the `--with-system-ffi` `configure` option; `libffi` must now always be supplied by the system on all non-Windows platforms. The option has had no effect on non-Darwin platforms for several releases, and in 3.11 only had the non-obvious effect of invoking `pkg-config` to find `libffi` and never setting `-DUSING_APPLE_OS_LIBFFI`. Now on Darwin platforms `configure` will first check for the OS `libffi` and then fall back to the same processing as other platforms if it is not found.
- gh-88267 [https://github.com/python/cpython/issues/88267]: Avoid exporting Python symbols in linked Windows applications when the core is built as static.
- bpo-41916 [https://bugs.python.org/issue?@action=redirect&bpo=41916]: Allow override of ac_cv_cxx_thread so that cross compiled python can set -pthread for CXX.

Windows

- gh-100180 [https://github.com/python/cpython/issues/100180]: Update Windows installer to OpenSSL 1.1.1s
- gh-99191 [https://github.com/python/cpython/issues/99191]: Use `_MSVC_LANG >= 202002L` instead of less-precise `_MSC_VER >=1929` to more accurately test for C++20 support in `PC/_wmimodule.cpp`.
- gh-79218 [https://github.com/python/cpython/issues/79218]: Define `MS_WIN64` for Mingw-w64 64bit, fix cython compilation

failure.

- gh-99941 [https://github.com/python/cpython/issues/99941]: Ensure that `asyncio.Protocol.data_received()` receives an immutable `bytes` object (as documented), instead of `bytearray`.
- bpo-43984 [https://bugs.python.org/issue?@action=redirect&bpo=43984]: `winreg.SetValueEx()` now leaves the target value untouched in the case of conversion errors. Previously, `-1` would be written in case of such errors.
- bpo-34816 [https://bugs.python.org/issue?@action=redirect&bpo=34816]: `hasattr(ctypes.windll, 'nonexistant')` now returns `False` instead of raising `OSError`.

macOS

- gh-100180 [https://github.com/python/cpython/issues/100180]: Update macOS installer to OpenSSL 1.1.1s
- gh-100540 [https://github.com/python/cpython/issues/100540]: Removed obsolete `dlfcn.h` shim from the `_ctypes` extension module, which has not been necessary since Mac OS X 10.2.

Tools/Demos

- bpo-45256 [https://bugs.python.org/issue?@action=redirect&bpo=45256]: Fix a bug that caused an `AttributeError` to be raised in `python-gdb.py` when `py-locals` is used without a frame.
- gh-100342 [https://github.com/python/cpython/issues/100342]: Add missing `NULL` check for possible allocation failure in `*args` parsing in Argument Clinic.

C API

- gh-99947 [https://github.com/python/cpython/issues/99947]: Raising SystemError on import will now have its cause be set to the original unexpected exception.
- gh-99240 [https://github.com/python/cpython/issues/99240]: In argument parsing, after deallocating newly allocated memory, reset its pointer to NULL.
- gh-98724 [https://github.com/python/cpython/issues/98724]: The **`Py_CLEAR`**, **`Py_SETREF`** and **`Py_XSETREF`** macros now only evaluate their arguments once. If an argument has side effects, these side effects are no longer duplicated. Patch by Victor Stinner.

Python 3.12.0 alpha 3

Release date: 2022-12-06

Security

- gh-100001 [https://github.com/python/cpython/issues/100001]: `python -m http.server` no longer allows terminal control characters sent within a garbage request to be printed to the stderr server log.

  This is done by changing the **`http.server`** **`BaseHTTPRequestHandler`** `.log_message` method to replace control characters with a `\x`*HH* hex escape before printing.

- gh-87604 [https://github.com/python/cpython/issues/87604]: Avoid publishing list of active per-interpreter audit hooks via the **`gc`** module

Core and Builtins

- gh-99891 [https://github.com/python/cpython/issues/99891]: Fix a bug in the tokenizer that could cause infinite recursion when showing syntax warnings that happen in the first line of the source. Patch by Pablo Galindo
- gh-91054 [https://github.com/python/cpython/issues/91054]: Add **`PyCode_AddWatcher()`** and **`PyCode_ClearWatcher()`** APIs to register callbacks to receive notification on creation and destruction of code objects.
- gh-99729 [https://github.com/python/cpython/issues/99729]: Fix an issue that could cause frames to be visible to Python code as they are being torn down, possibly leading to memory corruption or hard crashes of the interpreter.
- gh-99708 [https://github.com/python/cpython/issues/99708]: Fix bug where compiler crashes on an if expression with an empty body block.
- gh-99578 [https://github.com/python/cpython/issues/99578]: Fix a reference bug in **`_imp.create_builtin()`** after the creation of the first sub-interpreter for modules `builtins` and `sys`. Patch by Victor Stinner.
- gh-99581 [https://github.com/python/cpython/issues/99581]: Fixed a bug that was causing a buffer overflow if the tokenizer copies a line missing the newline character from a file that is as long as the available tokenizer buffer. Patch by Pablo galindo
- gh-99553 [https://github.com/python/cpython/issues/99553]: Fix bug where an **`ExceptionGroup`** subclass can wrap a **`BaseException`**.
- gh-99547 [https://github.com/python/cpython/issues/99547]: Add a function to os.path to check if a path is a junction: isjunction. Add similar functionality to pathlib.Path as is_junction.
- gh-99370 [https://github.com/python/cpython/issues/99370]: Fix zip path for venv created from a non-installed python on POSIX platforms.
- gh-99377 [https://github.com/python/cpython/issues/99377]: Add audit events for thread creation and clear operations.

- gh-98686 [https://github.com/python/cpython/issues/98686]: Remove the `BINARY_OP_GENERIC` and `COMPARE_OP_GENERIC` “specializations”.
- gh-99298 [https://github.com/python/cpython/issues/99298]: Remove the remaining error paths for attribute specializations, and refuse to specialize attribute accesses on types that haven’t had **`PyType_Ready()`** called on them yet.
- gh-99127 [https://github.com/python/cpython/issues/99127]: Allow some features of **`syslog`** to the main interpreter only. Patch by Donghee Na.
- gh-91053 [https://github.com/python/cpython/issues/91053]: Optimizing interpreters and JIT compilers may need to invalidate internal metadata when functions are modified. This change adds the ability to provide a callback that will be invoked each time a function is created, modified, or destroyed.
- gh-90994 [https://github.com/python/cpython/issues/90994]: Improve error messages when there’s a syntax error with call arguments. The following three cases are covered: - No value is assigned to a named argument, eg `foo(a=)`. - A value is assigned to a star argument, eg `foo(*args=[0])`. - A value is assigned to a double-star keyword argument, eg `foo(**kwarg={'a': 0})`.
- bpo-45026 [https://bugs.python.org/issue?@action=redirect&bpo=45026]: Optimize the **`range`** object iterator. It is now smaller, faster iteration of ranges containing large numbers. Smaller pickles, faster unpickling.
- bpo-31718 [https://bugs.python.org/issue?@action=redirect&bpo=31718]: Raise **`ValueError`** instead of **`SystemError`** when methods of uninitialized **`io.IncrementalNewlineDecoder`** objects are called. Patch by Oren Milman.
- bpo-38031 [https://bugs.python.org/issue?@action=redirect&bpo=38031]: Fix a possible assertion failure

in **`io.FileIO`** when the opener returns an invalid file descriptor.

Library

- gh-100001 [https://github.com/python/cpython/issues/100001]: Also escape s in the http.server BaseHTTPRequestHandler.log_message so that it is technically possible to parse the line and reconstruct what the original data was. Without this a xHH is ambiguous as to if it is a hex replacement we put in or the characters r”x” came through in the original request line.

- gh-99957 [https://github.com/python/cpython/issues/99957]: Add `frozen_default` parameter to **`typing.dataclass_transform()`**.

- gh-79033 [https://github.com/python/cpython/issues/79033]: Fix **`asyncio.Server.wait_closed()`** to actually do what the docs promise – wait for all existing connections to complete, after closing the server.

- gh-51524 [https://github.com/python/cpython/issues/51524]: Fix bug when calling trace.CoverageResults with valid infile.

- gh-99645 [https://github.com/python/cpython/issues/99645]: Fix a bug in handling class cleanups in **`unittest.TestCase`**. Now `addClassCleanup()` uses separate lists for different `TestCase` subclasses, and `doClassCleanups()` only cleans up the particular class.

- gh-99508 [https://github.com/python/cpython/issues/99508]: Fix `TypeError` in `Lib/importlib/_bootstrap_external.py` while calling `_imp.source_hash()`.

- gh-66285 [https://github.com/python/cpython/issues/66285]: Fix **`asyncio`** to not share event loop and signal wakeupfd in forked processes. Patch by Kumar Aditya.

- gh-97001 [https://github.com/python/cpython/issues/97001]: Release the GIL when calling termios APIs to avoid blocking threads.

- gh-92647 [https://github.com/python/cpython/issues/92647]: Use final status of an enum to determine lookup or creation branch of functional API.

- gh-99388 [https://github.com/python/cpython/issues/99388]: Add *loop_factory* parameter to **`asyncio.run()`** to allow specifying a custom event loop factory. Patch by Kumar Aditya.

- gh-99341 [https://github.com/python/cpython/issues/99341]: Fix **`ast.increment_lineno()`** to also cover **`ast.TypeIgnore`** when changing line numbers.

- gh-99382 [https://github.com/python/cpython/issues/99382]: Check the number of arguments in substitution in user generics containing a **`TypeVarTuple`** and one or more **`TypeVar`**.

- gh-99379 [https://github.com/python/cpython/issues/99379]: Fix substitution of **`ParamSpec`** followed by **`TypeVarTuple`** in generic aliases.

- gh-99344 [https://github.com/python/cpython/issues/99344]: Fix substitution of **`TypeVarTuple`** and **`ParamSpec`** together in user generics.

- gh-99284 [https://github.com/python/cpython/issues/99284]: Remove `_use_broken_old_ctypes_structure_semantics_` old untested and undocumented hack from **`ctypes`**.

- gh-99201 [https://github.com/python/cpython/issues/99201]: Fix `IndexError` when initializing the config variables on Windows if `HAVE_DYNAMIC_LOADING` is not set.

- gh-99240 [https://github.com/python/cpython/issues/99240]: Fix double-free bug in Argument Clinic `str_converter` by extracting memory clean up to a new `post_parsing` section.

- gh-64490 [https://github.com/python/cpython/issues/64490]: Fix refcount error when arguments are packed to tuple in Argument Clinic.

- gh-99029 [https://github.com/python/cpython/issues/99029]: `pathlib.PurePath.relative_to()` now treats naked Windows drive paths as relative. This brings its behaviour in line with other parts of pathlib.

- gh-98253 [https://github.com/python/cpython/issues/98253]: The implementation of the typing module is now more resilient to reference leaks in binary extension modules.

  Previously, a reference leak in a typed C API-based extension module could leak internals of the typing module, which could in turn introduce leaks in essentially any other package with typed function signatures. Although the typing package is not the original source of the problem, such non-local dependences exacerbate debugging of large-scale projects, and the implementation was therefore changed to reduce harm by providing better isolation.

- gh-98458 [https://github.com/python/cpython/issues/98458]: Fix infinite loop in unittest when a self-referencing chained exception is raised

- gh-93453 [https://github.com/python/cpython/issues/93453]: **`asyncio.get_event_loop()`** and many other **`asyncio`** functions like **`asyncio.ensure_future()`**, **`asyncio.shield()`** or **`asyncio.gather()`**, and also the **`get_event_loop()`** method of **`asyncio.BaseDefaultEventLoopPolicy`** now raise a **`RuntimeError`** if called when there is no running event loop and the current event loop was not set. Previously they implicitly created and set a new current event loop. **`DeprecationWarning`** is no longer emitted if there is no running event loop but the current event loop was set.

- gh-97966 [https://github.com/python/cpython/issues/97966]: On `uname_result`, restored expectation that `_fields` and `_asdict` would include all six properties including `processor`.

- gh-98248 [https://github.com/python/cpython/issues/98248]: Provide informative error messages in **`struct.pack()`** when its integral arguments are not in range.

- gh-98108 [https://github.com/python/cpython/issues/98108]: `zipfile.Path` is now pickleable if its initialization parameters were pickleable (e.g. for file system paths).

- gh-98098 [https://github.com/python/cpython/issues/98098]: Created packages from zipfile and test_zipfile modules, separating `zipfile.Path` functionality.

- gh-82836 [https://github.com/python/cpython/issues/82836]: Fix **`is_private`** properties in the **`ipaddress`** module. Previously non-private networks (0.0.0.0/0) would return `True` from this method; now they correctly return `False`.

- gh-96828 [https://github.com/python/cpython/issues/96828]: Add an **`OP_ENABLE_KTLS`** option for enabling the use of the kernel TLS (kTLS). Patch by Illia Volochii.

- gh-88863 [https://github.com/python/cpython/issues/88863]: To avoid apparent memory leaks when `asyncio.open_connection()` raises, break reference cycles generated by local exception and future instances (which has exception instance as its member var). Patch by Dong Uk, Kang.
- gh-91078 [https://github.com/python/cpython/issues/91078]: **`TarFile.next()`** now returns `None` when called on an empty tarfile.
- bpo-47220 [https://bugs.python.org/issue?@action=redirect&bpo=47220]: Document the optional *callback* parameter of **`WeakMethod`**. Patch by Géry Ogam.
- bpo-44817 [https://bugs.python.org/issue?@action=redirect&bpo=44817]: Ignore WinError 53 (ERROR_BAD_NETPATH), 65 (ERROR_NETWORK_ACCESS_DENIED) and 161 (ERROR_BAD_PATHNAME) when using ntpath.realpath().
- bpo-41260 [https://bugs.python.org/issue?@action=redirect&bpo=41260]: Rename the *fmt* parameter of the pure Python implementation of `datetime.date.strftime()` to *format*.
- bpo-15999 [https://bugs.python.org/issue?@action=redirect&bpo=15999]: All built-in functions now accept arguments of any type instead of just `bool` and `int` for boolean parameters.

Documentation

- gh-99931 [https://github.com/python/cpython/issues/99931]: Use sphinxext-opengraph [https://sphinxext-opengraph.readthedocs.io/] to generate OpenGraph metadata [https://ogp.me/].

- gh-89682 [https://github.com/python/cpython/issues/89682]: Reworded docstring of the default `__contains__` to clarify that it returns a **`bool`**.
- gh-88330 [https://github.com/python/cpython/issues/88330]: Improved the description of what a resource is in importlib.resources docs.
- gh-92892 [https://github.com/python/cpython/issues/92892]: Document that calling variadic functions with ctypes requires special care on macOS/arm64 (and possibly other platforms).
- bpo-41825 [https://bugs.python.org/issue?@action=redirect&bpo=41825]: Restructured the documentation for the **`os.wait*`** family of functions, and improved the docs for **`os.waitid()`** with more explanation of the possible argument constants.

Tests

- gh-99892 [https://github.com/python/cpython/issues/99892]: Skip test_normalization() of test_unicodedata if it fails to download NormalizationTest.txt file from pythontest.net. Patch by Victor Stinner.
- gh-99934 [https://github.com/python/cpython/issues/99934]: Correct test_marsh on (32 bit) x86: test_deterministic sets was failing.
- gh-99741 [https://github.com/python/cpython/issues/99741]: We've implemented multi-phase init (PEP 489/630/687) for the internal (for testing) _xxsubinterpreters module.
- gh-99659 [https://github.com/python/cpython/issues/99659]: Optional big memory tests in `test_sqlite3` now catch the correct **`sqlite.DataError`** exception type in case of too large strings and/or blobs passed.
- gh-99593 [https://github.com/python/cpython/issues/99593]: Cover the Unicode C API with tests.
- gh-96002 [https://github.com/python/cpython/issues/96002]: Add functional test for Argument Clinic.

Build

- gh-99086 [https://github.com/python/cpython/issues/99086]: Fix `-Wimplicit-int`, `-Wstrict-prototypes`, and `-Wimplicit-function-declaration` compiler warnings in **configure** checks.
- gh-99337 [https://github.com/python/cpython/issues/99337]: Fix a compilation issue with GCC 12 on macOS.
- gh-99289 [https://github.com/python/cpython/issues/99289]: Add a `COMPILEALL_OPTS` variable in Makefile to override `compileall` options (default: `-j0`) in `make install`. Also merged the `compileall` commands into a single command building .pyc files for the all optimization levels (0, 1, 2) at once. Patch by Victor Stinner.
- gh-98872 [https://github.com/python/cpython/issues/98872]: Fix a possible fd leak in `Programs/_freeze_module.c` introduced in Python 3.11.
- gh-88226 [https://github.com/python/cpython/issues/88226]: Always define `TARGET_*` labels in `Python/ceval.c`, even if `USE_COMPUTED_GOTOS` is disabled. This allows breakpoints to be set at those labels in (for instance) `gdb`.

Windows

- gh-99345 [https://github.com/python/cpython/issues/99345]: Use faster initialization functions to detect install location for Windows Store package
- gh-98629 [https://github.com/python/cpython/issues/98629]: Fix initialization of `sys.version` and `sys._git` on Windows
- gh-99442 [https://github.com/python/cpython/issues/99442]: Fix handling in Python install manager when `argv[0]` does not include a file extension.
- bpo-40882 [https://bugs.python.org/issue?@action=redirect&bpo=40882]: Fix a memory leak in `multiprocessing.shared_memory.SharedMemory` on Windows.

macOS

- gh-87235 [https://github.com/python/cpython/issues/87235]: On macOS `python3 /dev/fd/9 9</path/to/script.py` failed for any script longer than a couple of bytes.
- gh-98940 [https://github.com/python/cpython/issues/98940]: Fix `Mac/Extras.install.py` file filter bug.

Tools/Demos

- gh-64490 [https://github.com/python/cpython/issues/64490]: Argument Clinic varargs bugfixes
  - Fix out-of-bounds error in **`_PyArg_UnpackKeywordsWithVararg()`**.
  - Fix incorrect check which allowed more than one varargs in clinic.py.
  - Fix miscalculation of `noptargs` in generated code.
  - Do not generate `noptargs` when there is a vararg argument and no optional argument.

C API

- gh-98680 [https://github.com/python/cpython/issues/98680]: PyBUF_* constants were marked as part of Limited API of Python 3.11+. These were available in 3.11.0 with **`Py_LIMITED_API`** defined for 3.11, and are necessary to use the buffer API.
- gh-99612 [https://github.com/python/cpython/issues/99612]: Fix **`PyUnicode_DecodeUTF8Stateful()`** for ASCII-only data: `*consumed` was not set.
- gh-47146 [https://github.com/python/cpython/issues/47146]: The `structmember.h` header is deprecated. Its non-deprecated contents are now available just by including `Python.h`, with a `Py_` prefix added if it was missing. (Deprecated contents are **`T_OBJECT`**, **`T_NONE`**, and no-op flags.) Patch by

Petr Viktorin, based on earlier work by Alexander Belopolsky and Matthias Braun.

Python 3.12.0 alpha 2

Release date: 2022-11-14

Security

- gh-98433 [https://github.com/python/cpython/issues/98433]: The IDNA codec decoder used on DNS hostnames by **`socket`** or **`asyncio`** related name resolution functions no longer involves a quadratic algorithm. This prevents a potential CPU denial of service if an out-of-spec excessive length hostname involving bidirectional characters were decoded. Some protocols such as **`urllib`** http *3xx* redirects potentially allow for an attacker to supply such a name.

  Individual labels within an IDNA encoded DNS name will now raise an error early during IDNA decoding if they are longer than 1024 unicode characters given that each decoded DNS label must be 63 or fewer characters and the entire decoded DNS name is limited to 255. Only an application presenting a hostname or label consisting primarily of **RFC 3454** [https://datatracker.ietf.org/doc/html/rfc3454.html] section 3.1 “Nothing” characters to be removed would run into of this new limit. See also **RFC 5894** [https://datatracker.ietf.org/doc/html/rfc5894.html] section 6 and **RFC 3491** [https://datatracker.ietf.org/doc/html/rfc3491.html].

- gh-98739 [https://github.com/python/cpython/issues/98739]: Update bundled libexpat to 2.5.0

Core and Builtins

- gh-81057 [https://github.com/python/cpython/issues/81057]: The docs clearly say that `PyImport_Inittab`, `PyImport_AppendInittab()`, and `PyImport_ExtendInittab()` should not be used after `Py_Initialize()` has been called. We now enforce this for the two functions. Additionally, the runtime now uses an internal copy of `PyImport_Inittab`, to guard against modification.

- gh-99298 [https://github.com/python/cpython/issues/99298]: Fix an issue that could potentially cause incorrect error handling for some bytecode instructions.

- gh-99254 [https://github.com/python/cpython/issues/99254]: The compiler now removes all unused constants from code objects (except the first one, which may be a docstring).

- gh-99205 [https://github.com/python/cpython/issues/99205]: Fix an issue that prevented `PyThreadState` and `PyInterpreterState` memory from being freed properly.

- gh-81057 [https://github.com/python/cpython/issues/81057]: The 18 global C variables holding the state of the allocators have been moved to `_PyRuntimeState`. This is a strictly internal change with no change in behavior.

- gh-99181 [https://github.com/python/cpython/issues/99181]: Fix failure in `except*` with unhashable exceptions.

- gh-99204 [https://github.com/python/cpython/issues/99204]: Fix calculation of **`sys._base_executable`** when inside a POSIX virtual environment using copies of the python binary when the base installation does not provide the executable name used by the venv. Calculation will fall back to alternative names (“python<MAJOR>”, “python<MAJOR>.<MINOR>”).

- gh-96055 [https://github.com/python/cpython/issues/96055]: Update **`faulthandler`** to emit an error message with the proper unexpected signal number. Patch by Donghee Na.

- gh-99153 [https://github.com/python/cpython/issues/99153]: Fix location of **`SyntaxError`** for a **`try`** block with both **`except`** and **`except*`**.

- gh-98686 [https://github.com/python/cpython/issues/98686]: Merge the adaptive opcode logic into each instruction’s unquickened variant, and merge the logic in `EXTENDED_ARG_QUICK` into **`EXTENDED_ARG`**. With these changes, the quickening that happens at code object creation is now only responsible for initializing warmup counters and inserting superinstructions.

- gh-99103 [https://github.com/python/cpython/issues/99103]: Fix the error reporting positions of specialized traceback anchors when the source line contains Unicode characters.

- gh-99139 [https://github.com/python/cpython/issues/99139]: Improve the error suggestion for **`NameError`** exceptions for instances. Now if a **`NameError`** is raised in a method and the instance has an attribute that’s exactly equal to the name in the exception, the suggestion will include `self.<NAME>` instead of the closest match in the method scope. Patch by Pablo Galindo

- gh-98401 [https://github.com/python/cpython/issues/98401]: Octal escapes with value larger than `0o377` (ex: `"\477"`), deprecated in Python 3.11, now produce a **`SyntaxWarning`**, instead of **`DeprecationWarning`**. In a future Python version they will be eventually a **`SyntaxError`**. Patch by Victor Stinner.

- gh-98401 [https://github.com/python/cpython/issues/98401]: A backslash-character pair that is not a valid escape sequence now generates a **`SyntaxWarning`**, instead of **`DeprecationWarning`**. For example, `re.compile("\d+\.\d+")` now emits a **`SyntaxWarning`** (`"\d"` is an invalid escape sequence), use raw strings for regular expression: `re.compile(r"\d+\.\d+")`. In a future Python version, **`SyntaxError`** will eventually be raised, instead of **`SyntaxWarning`**. Patch by Victor Stinner.

- gh-96793 [https://github.com/python/cpython/issues/96793]: Handle StopIteration and StopAsyncIteration raised in generator or coroutines in the bytecode, rather than in wrapping C code.

- gh-98931 [https://github.com/python/cpython/issues/98931]: Improve the **`SyntaxError`** error message when the user types `import x from y` instead of `from y import x`. Patch by Pablo Galindo

- gh-98852 [https://github.com/python/cpython/issues/98852]: Fix subscription of type aliases containing bare generic types or types like **`TypeVar`**: for example `tuple[A, T][int]` and `tuple[TypeVar, T][int]`, where `A` is a generic type, and `T` is a type variable.

- gh-98925 [https://github.com/python/cpython/issues/98925]: Lower the recursion depth for marshal on WASI to support (in-development) wasmtime 2.0.

- gh-98783 [https://github.com/python/cpython/issues/98783]: Fix multiple crashes in debug mode when `str` subclasses are used instead of `str` itself.

- gh-98811 [https://github.com/python/cpython/issues/98811]: Use complete source locations to simplify detection of `__future__` imports which are not at the beginning of the file. Also corrects the offset in the exception raised in one case, which was off by one and impeded highlighting.

- gh-96793 [https://github.com/python/cpython/issues/96793]: Add specialization of **`FOR_ITER`** for generators. Saves multiple layers of dispatch and checking to get from the **`FOR_ITER`** instruction in the caller to the **`RESUME`** in the generator.

- gh-98762 [https://github.com/python/cpython/issues/98762]: Fix source locations of **`match`** sub-patterns.

- gh-98586 [https://github.com/python/cpython/issues/98586]: Added the methods **`PyObject_Vectorcall()`** and **`PyObject_VectorcallMethod()`** to the Limited API along with the auxiliary macro constant **`PY_VECTORCALL_ARGUMENTS_OFFSET`**.

  The availability of these functions enables more efficient **PEP 590** [https://peps.python.org/pep-0590/] vector calls from binary extension modules that avoid argument boxing/unboxing overheads.

- gh-99257 [https://github.com/python/cpython/issues/99257]: Fix an issue where member descriptors (such as those for **`__slots__`**) could behave incorrectly or crash instead of raising a **`TypeError`** when accessed via an instance of an invalid type.

- gh-93143 [https://github.com/python/cpython/issues/93143]: Rather than changing **`co_code`**, the interpreter will now display a **`RuntimeWarning`** and assign **`None`** to any fast locals that are left unbound after jumps or **`del`** statements executed while tracing.

- gh-96421 [https://github.com/python/cpython/issues/96421]: When calling into Python code from C code, through **`PyEval_EvalFrameEx()`** or a related C-API function, a shim frame in inserted into the call stack. This occurs in the `_PyEval_EvalFrameDefault()` function. The extra frame should be invisible to all Python and most C extensions, but out-of-process profilers and debuggers need to be aware of it. These shim frames can be detected by checking `frame->owner == FRAME_OWNED_BY_CSTACK`.

  Extensions implementing their own interpreters using PEP 523 need to be aware of this shim frame and the changes to the semantics of **`RETURN_VALUE`**, **`YIELD_VALUE`**, and **`RETURN_GENERATOR`**, which now clear the frame.

- gh-98415 [https://github.com/python/cpython/issues/98415]: Fix detection of MAC addresses for **`uuid`** on certain OSs. Patch by Chaim Sanders

- gh-98686 [https://github.com/python/cpython/issues/98686]: Quicken all code objects, and specialize adaptive bytecode instructions more aggressively.

- gh-92119 [https://github.com/python/cpython/issues/92119]: Print exception class name instead of its string representation when raising errors from **`ctypes`** calls.

- gh-91058 [https://github.com/python/cpython/issues/91058]: **`ImportError`** raised from failed `from <module> import <name>` now include suggestions for the value of `<name>` based on the available names in `<module>`. Patch by Pablo Galindo

- gh-96793 [https://github.com/python/cpython/issues/96793]: The **`FOR_ITER`** now leaves the iterator on the stack on termination of the loop. This is to assist specialization of loops for generators.

- gh-90716 [https://github.com/python/cpython/issues/90716]: Add _pylong.py module. It includes asymptotically faster algorithms that can be used for operations on integers with many digits. It is used by longobject.c to speed up some operations.

- gh-95389 [https://github.com/python/cpython/issues/95389]: Expose **`ETH_P_ALL`** and some of the ETHERTYPE_* constants in **`socket`**. Patch by Noam Cohen.

- gh-93696 [https://github.com/python/cpython/issues/93696]: Allow **`pdb`** to locate source for frozen modules in the standard library.

Library

- gh-99418 [https://github.com/python/cpython/issues/99418]: Fix bug in **`urllib.parse.urlparse()`** that causes URL schemes that begin with a digit, a plus sign, or a minus sign to be parsed incorrectly.
- gh-94597 [https://github.com/python/cpython/issues/94597]: Deprecate **`asyncio.AbstractChildWatcher`** to be removed in Python 3.14. Patch by Kumar Aditya.
- gh-99305 [https://github.com/python/cpython/issues/99305]: Improve performance of **`secrets.token_hex()`**.
- gh-74044 [https://github.com/python/cpython/issues/74044]: Fixed bug where **`inspect.signature()`** reported incorrect arguments for decorated methods.
- gh-99275 [https://github.com/python/cpython/issues/99275]: Fix `SystemError` in **`ctypes`** when exception was not set during `__initsubclass__`.
- gh-99277 [https://github.com/python/cpython/issues/99277]: Remove older version of `_SSLProtocolTransport.get_write_buffer_limits` in **`asyncio.sslproto`**

- gh-99248 [https://github.com/python/cpython/issues/99248]: fix negative numbers failing in verify()
- gh-99155 [https://github.com/python/cpython/issues/99155]: Fix **`statistics.NormalDist`** pickle with `0` and `1` protocols.
- gh-93464 [https://github.com/python/cpython/issues/93464]: `enum.auto()` is now correctly activated when combined with other assignment values. E.g. `ONE = auto(), 'some text'` will now evaluate as `(1, 'some text')`.
- gh-99134 [https://github.com/python/cpython/issues/99134]: Update the bundled copy of pip to version 22.3.1.
- gh-92584 [https://github.com/python/cpython/issues/92584]: Remove the `distutils` package. It was deprecated in Python 3.10 by **PEP 632** [https://peps.python.org/pep-0632/] “Deprecate distutils module”. For projects still using `distutils` and cannot be updated to something else, the `setuptools` project can be installed: it still provides `distutils`. Patch by Victor Stinner.
- gh-98999 [https://github.com/python/cpython/issues/98999]: Now **`_pyio`** is consistent with **`_io`** in raising `ValueError` when executing methods over closed buffers.
- gh-83004 [https://github.com/python/cpython/issues/83004]: Clean up refleak on failed module initialisation in **`_zoneinfo`**
- gh-83004 [https://github.com/python/cpython/issues/83004]: Clean up refleaks on failed module initialisation in **`_pickle`**
- gh-83004 [https://github.com/python/cpython/issues/83004]: Clean up refleak on failed module initialisation in **`_io`**.
- gh-98897 [https://github.com/python/cpython/issues/98897]: Fix memory leak in **`math.dist()`** when both points don’t have the same dimension. Patch by Kumar Aditya.
- gh-98878 [https://github.com/python/cpython/issues/98878]: Use the frame bound builtins when offering a name suggestion in **`traceback`** to prevent crashing when `__builtins__` is not a dict.
- gh-98139 [https://github.com/python/cpython/issues/98139]: In **`importlib._bootstrap`**, enhance namespace package repr to `<module 'x' (namespace) from ['path']>`.

- gh-90352 [https://github.com/python/cpython/issues/90352]: Fix `_SelectorDatagramTransport` to inherit from **`DatagramTransport`** in **`asyncio`**. Patch by Kumar Aditya.
- gh-98793 [https://github.com/python/cpython/issues/98793]: Fix argument typechecks in **`_overlapped.WSAConnect()`** and **`_overlapped.Overlapped.WSASendTo()`** functions.
- gh-98744 [https://github.com/python/cpython/issues/98744]: Prevent crashing in **`traceback`** when retrieving the byte-offset for some source files that contain certain unicode characters.
- gh-98740 [https://github.com/python/cpython/issues/98740]: Fix internal error in the **`re`** module which in very rare circumstances prevented compilation of a regular expression containing a conditional expression without the "else" branch.
- gh-98703 [https://github.com/python/cpython/issues/98703]: Fix **`asyncio.StreamWriter.drain()`** to call `protocol.connection_lost` callback only once on Windows.
- gh-98624 [https://github.com/python/cpython/issues/98624]: Add a mutex to unittest.mock.NonCallableMock to protect concurrent access to mock attributes.
- gh-98658 [https://github.com/python/cpython/issues/98658]: The **`array.array`** class now supports subscripting, making it a generic type.
- gh-98284 [https://github.com/python/cpython/issues/98284]: Improved **`TypeError`** message for undefined abstract methods of a **`abc.ABC`** instance. The names of the missing methods are surrounded by single-quotes to highlight them.
- gh-96151 [https://github.com/python/cpython/issues/96151]: Allow `BUILTINS` to be a valid field name for frozen dataclasses.
- gh-98086 [https://github.com/python/cpython/issues/98086]: Make sure `patch.dict()` can be applied on async functions.
- gh-72719 [https://github.com/python/cpython/issues/72719]: Remove modules **`asyncore`** and **`asynchat`**, which were deprecated by **PEP 594** [https://peps.python.org/pep-0594/].

- gh-96192 [https://github.com/python/cpython/issues/96192]: Fix handling of `bytes` path-like objects in **`os.ismount()`**.
- gh-94172 [https://github.com/python/cpython/issues/94172]: **`ftplib`**: Remove the `FTP_TLS.ssl_version` class attribute: use the *context* parameter instead. Patch by Victor Stinner
- gh-94172 [https://github.com/python/cpython/issues/94172]: Remove the *keyfile* and *certfile* parameters from the **`ftplib`**, **`imaplib`**, **`poplib`** and **`smtplib`** modules, and the *key_file*, *cert_file* and *check_hostname* parameters from the **`http.client`** module, all deprecated since Python 3.6. Use the *context* parameter (*ssl_context* in **`imaplib`**) instead. Patch by Victor Stinner.
- gh-83638 [https://github.com/python/cpython/issues/83638]: Add the **`autocommit`** attribute to **`sqlite3.Connection`** and the *autocommit* parameter to **`sqlite3.connect()`** to control **PEP 249** [https://peps.python.org/pep-0249/]-compliant transaction handling. Patch by Erlend E. Aasland.
- gh-92452 [https://github.com/python/cpython/issues/92452]: Fixed a race condition that could cause **`sysconfig.get_config_var()`** to incorrectly return **`None`** in multi-threaded programs.
- gh-91803 [https://github.com/python/cpython/issues/91803]: Fix an error when using a method of objects mocked with **`unittest.mock.create_autospec()`** after it was sealed with **`unittest.mock.seal()`** function.
- bpo-38523 [https://bugs.python.org/issue?@action=redirect&bpo=38523]: **`shutil.copytree()`** now applies the *ignore_dangling_symlinks* argument recursively.
- bpo-40358 [https://bugs.python.org/issue?@action=redirect&bpo=40358]: Add walk_up argument in **`pathlib.PurePath.relative_to()`**.
- bpo-36267 [https://bugs.python.org/issue?@action=redirect&bpo=36267]: Fix IndexError in **`argparse.ArgumentParser`** when a `store_true` action is given an explicit argument.

Documentation

- gh-98832 [https://github.com/python/cpython/issues/98832]: Changes wording of docstring for `pathlib.Path.iterdir()`.
- gh-97966 [https://github.com/python/cpython/issues/97966]: Update uname docs to clarify the special nature of the platform attribute and to indicate when it became late-bound.

Tests

- gh-98903 [https://github.com/python/cpython/issues/98903]: The Python test suite now fails with exit code 4 if no tests ran. It should help detecting typos in test names and test methods.
- gh-98713 [https://github.com/python/cpython/issues/98713]: Fix a bug in the `typing` tests where a test relying on CPython-specific implementation details was not decorated with `@cpython_only` and was not skipped on other implementations.
- gh-87390 [https://github.com/python/cpython/issues/87390]: Add tests for star-unpacking with PEP 646, and some other miscellaneous PEP 646 tests.
- gh-96853 [https://github.com/python/cpython/issues/96853]: Added explicit coverage of `Py_Initialize` (and hence `Py_InitializeEx`) back to the embedding tests (all other embedding tests migrated to `Py_InitializeFromConfig` in Python 3.11)
- bpo-34272 [https://bugs.python.org/issue?@action=redirect&bpo=34272]: Some C API tests were moved into the new Lib/test/test_capi/ directory.

Build

- gh-99086 [https://github.com/python/cpython/issues/99086]: Fix `-Wimplicit-int` compiler warning in **configure** check for

`PTHREAD_SCOPE_SYSTEM`.

- gh-99016 [https://github.com/python/cpython/issues/99016]: Fix build with `PYTHON_FOR_REGEN=python3.8`.
- gh-97731 [https://github.com/python/cpython/issues/97731]: Specify the full path to the source location for `make docclean` (needed for cross-builds).
- gh-98949 [https://github.com/python/cpython/issues/98949]: Drop unused build dependency on `readelf`.
- gh-98989 [https://github.com/python/cpython/issues/98989]: Use `python3.11`, if available, for regeneration and freezing.
- gh-98831 [https://github.com/python/cpython/issues/98831]: Add new tooling, in `Tools/cases_generator`, to generate the interpreter switch statement from a list of opcode definitions. This only affects adding, modifying or removing instruction definitions. The instruction definitions now live in `Python/bytecodes.c`, in the form of a custom DSL (under development) [https://github.com/faster-cpython/ideas/blob/main/3.12/interpreter_definition.md]. The tooling reads this file and writes `Python/generated_cases.c.h`, which is then included by `Python/ceval.c` to provide most of the cases of the main interpreter switch.
- gh-98817 [https://github.com/python/cpython/issues/98817]: Remove PCbuild/lib.pyproj: it’s not used for anything, is only a minor convenience for Visual Studio users (who probably mostly don’t even know about it), and it takes a lot of maintenance effort to keep updated.
- gh-98776 [https://github.com/python/cpython/issues/98776]: Fix `make regen-test-levenshtein` for out-of-tree builds.
- gh-98707 [https://github.com/python/cpython/issues/98707]: Don’t use vendored `libmpdec` headers if **`--with-system-libmpdec`** is passed to **configure**. Don’t use vendored `libexpat` headers if **`--with-system-expat`** is passed to **configure**.

Windows

- gh-98689 [https://github.com/python/cpython/issues/98689]: Update Windows builds to zlib v1.2.13. v1.2.12 has **CVE 2022-37434** [https://www.cve.org/CVERecord?id=CVE-2022-37434], but the vulnerable `inflateGetHeader` API is not used by Python.
- gh-98790 [https://github.com/python/cpython/issues/98790]: Assumes that a missing `DLLs` directory means that standard extension modules are in the executable’s directory.
- gh-98745 [https://github.com/python/cpython/issues/98745]: Update `py.exe` launcher to install 3.11 by default and 3.12 on request.
- gh-98692 [https://github.com/python/cpython/issues/98692]: Fix the Python install manager ignoring unrecognized shebang lines instead of treating them as local paths
- gh-94328 [https://github.com/python/cpython/issues/94328]: Update Windows installer to use SQLite 3.39.4.

macOS

- gh-94328 [https://github.com/python/cpython/issues/94328]: Update macOS installer to SQLite 3.39.4.

C API

- gh-98724 [https://github.com/python/cpython/issues/98724]: The **`Py_CLEAR`**, **`Py_SETREF`** and **`Py_XSETREF`** macros now only evaluate their argument once. If the argument has side effects, these side effects are no longer duplicated. Patch by Victor Stinner.
- gh-98978 [https://github.com/python/cpython/issues/98978]: Fix use-after-free in `Py_SetPythonHome(NULL)`, `Py_SetProgramName(NULL)` and `_Py_SetProgramFullPath(NULL)` function calls. Issue reported by Benedikt Reinartz. Patch by Victor Stinner.
- gh-98410 [https://github.com/python/cpython/issues/98410]: Add `getbufferproc` and `releasebufferproc` to the stable API.

- gh-98610 [https://github.com/python/cpython/issues/98610]: Some configurable capabilities of sub-interpreters have changed. They always allow subprocesses (**`subprocess`**) now, whereas before subprocesses could be optionally disallowed for a sub-interpreter. Instead **`os.exec()`** can now be disallowed. Disallowing daemon threads is now supported. Disallowing all threads is still allowed, but is never done by default. Note that the optional restrictions are only available through `_Py_NewInterpreterFromConfig()`, which isn't a public API. They do not affect the main interpreter, nor **`Py_NewInterpreter()`**.
- gh-98608 [https://github.com/python/cpython/issues/98608]: A `_PyInterpreterConfig` has been added and `_Py_NewInterpreter()` has been renamed to `_Py_NewInterpreterFromConfig()`. The "isolated_subinterpreters" argument is now a granular config that captures the previous behavior. Note that this is all "private" API.
- gh-96853 [https://github.com/python/cpython/issues/96853]: `Py_InitializeEx` now correctly calls `PyConfig_Clear` after initializing the interpreter (the omission didn't cause a memory leak only because none of the dynamically allocated config fields are populated by the wrapper function)
- gh-91248 [https://github.com/python/cpython/issues/91248]: Add **`PyFrame_GetVar()`** and **`PyFrame_GetVarString()`** functions to get a frame variable by its name. Patch by Victor Stinner.

Python 3.12.0 alpha 1

Release date: 2022-10-25

Security

- gh-97616 [https://github.com/python/cpython/issues/97616]: Fix multiplying a list by an integer (`list *= int`): detect the integer overflow when the new allocated length is close to the maximum size. Issue reported by Jordan Limor. Patch by Victor Stinner.

- gh-97514 [https://github.com/python/cpython/issues/97514]: On Linux the **`multiprocessing`** module returns to using filesystem backed unix domain sockets for communication with the *forkserver* process instead of the Linux abstract socket namespace. Only code that chooses to use the "forkserver" start method is affected.

  Abstract sockets have no permissions and could allow any user on the system in the same network namespace [https://man7.org/linux/man-pages/man7/network_namespaces.7.html] (often the whole system) to inject code into the multiprocessing *forkserver* process. This was a potential privilege escalation. Filesystem based socket permissions restrict this to the *forkserver* process user as was the default in Python 3.8 and earlier.

  This prevents Linux **CVE 2022-42919** [https://www.cve.org/CVERecord?id=CVE-2022-42919].

- gh-87389 [https://github.com/python/cpython/issues/87389]: **`http.server`**: Fix an open redirection vulnerability in the HTTP server when an URI path starts with `//`. Vulnerability discovered, and initial fix proposed, by Hamza Avvan.

- gh-79096 [https://github.com/python/cpython/issues/79096]: LWPCookieJar and MozillaCookieJar create files with file mode 600 instead of 644 (Microsoft Windows is not affected)

- gh-92888 [https://github.com/python/cpython/issues/92888]: Fix `memoryview` use after free when accessing the backing buffer in certain cases.

- gh-68966 [https://github.com/python/cpython/issues/68966]: The deprecated mailcap module now refuses to inject unsafe text (filenames, MIME types, parameters) into shell commands. Instead of using such text, it will warn and act as if a match was not found (or for test commands, as if the test failed).

Core and Builtins

- gh-98374 [https://github.com/python/cpython/issues/98374]: Suppress ImportError for invalid query for help() command. Patch by Donghee Na.

- gh-98461 [https://github.com/python/cpython/issues/98461]: Fix source location in bytecode for list, set and dict comprehensions as well as generator expressions.

- gh-98354 [https://github.com/python/cpython/issues/98354]: Added unicode check for `name` attribute of `spec` argument passed in **`_imp.create_builtin()`** function.

- gh-98398 [https://github.com/python/cpython/issues/98398]: Fix source location of ‘assert’ bytecodes.

- gh-98390 [https://github.com/python/cpython/issues/98390]: Fix location of sub-expressions of boolean expressions, by reducing their scope to that of the sub-expression.

- gh-98254 [https://github.com/python/cpython/issues/98254]: Modules from the standard library are now potentially suggested as part of the error messages displayed by the interpreter when an **`NameError`** is raised to the top level. Patch by Pablo Galindo

- gh-97997 [https://github.com/python/cpython/issues/97997]: Add running column offset to the tokenizer state to avoid calculating AST column information with pointer arithmetic.

- gh-97973 [https://github.com/python/cpython/issues/97973]: Modify the tokenizer to return all necessary information the parser needs to set location information in the AST nodes, so that the parser does not have to calculate those doing pointer arithmetic.

- gh-96078 [https://github.com/python/cpython/issues/96078]: `os.sched_yield()` now release the GIL while calling sched_yield(2). Patch by Donghee Na.

- gh-97955 [https://github.com/python/cpython/issues/97955]: Migrate `zoneinfo` to Argument Clinic.

- gh-97912 [https://github.com/python/cpython/issues/97912]: The compiler now avoids quadratic behavior when finding which instructions should use the `LOAD_FAST_CHECK` opcode.

- gh-97002 [https://github.com/python/cpython/issues/97002]: Fix an issue where several frame objects could be backed by the same interpreter frame, possibly leading to corrupted memory and hard crashes of the interpreter.

- gh-97943 [https://github.com/python/cpython/issues/97943]: Bugfix: `PyFunction_GetAnnotations()` should return a borrowed reference. It was returning a new reference.

- gh-97922 [https://github.com/python/cpython/issues/97922]: The Garbage Collector now runs only on the eval breaker mechanism of the Python bytecode evaluation loop instead on object allocations. The GC can also run when `PyErr_CheckSignals()` is called so C extensions that need to run for a long time without executing any Python code also have a chance to execute the GC periodically.

- gh-65961 [https://github.com/python/cpython/issues/65961]: When `__package__` is different than `__spec__.parent`, raise a `DeprecationWarning` instead of `ImportWarning`.

  Also remove `importlib.util.set_package()` which was scheduled for removal.

- gh-97850 [https://github.com/python/cpython/issues/97850]: Long deprecated, `module_repr()` should now be completely eradicated.
- gh-86298 [https://github.com/python/cpython/issues/86298]: In cases where `warnings.warn_explicit()` consults the module's loader, an `DeprecationWarning` is issued when `m.__loader__` differs from `m.__spec__.loader`.
- gh-97779 [https://github.com/python/cpython/issues/97779]: Ensure that all Python frame objects are backed by "complete" frames.
- gh-91052 [https://github.com/python/cpython/issues/91052]: Add API for subscribing to modification events on selected dictionaries.
- gh-97752 [https://github.com/python/cpython/issues/97752]: Fix possible data corruption or crashes when accessing the `f_back` member of newly-created generator or coroutine frames.
- gh-97591 [https://github.com/python/cpython/issues/97591]: Fixed a missing incref/decref pair in `Exception.__setstate__()`. Patch by Ofey Chan.

- gh-97670 [https://github.com/python/cpython/issues/97670]: Remove the **`sys.getdxp()`** function and the `Tools/scripts/analyze_dxp.py` script. DXP stands for “dynamic execution pairs”. They were related to `DYNAMIC_EXECUTION_PROFILE` and `DXPAIRS` macros which have been removed in Python 3.11. Python can now be built with **`./configure --enable-pystats`** to gather statistics on Python opcodes. Patch by Victor Stinner.

- gh-94526 [https://github.com/python/cpython/issues/94526]: Fix the Python path configuration used to initialized **`sys.path`** at Python startup. Paths are no longer encoded to UTF-8/strict to avoid encoding errors if it contains surrogate characters (bytes paths are decoded with the surrogateescape error handler). Patch by Victor Stinner.

- gh-96670 [https://github.com/python/cpython/issues/96670]: The parser now raises **`SyntaxError`** when parsing source code containing null bytes. Patch by Pablo Galindo

- gh-96975 [https://github.com/python/cpython/issues/96975]: Fix a crash occurring when **`PyEval_GetFrame()`** is called while the topmost Python frame is in a partially-initialized state.

- gh-96848 [https://github.com/python/cpython/issues/96848]: Fix command line parsing: reject **`-X int_max_str_digits`** option with no value (invalid) when the **`PYTHONINTMAXSTRDIGITS`** environment variable is set to a valid limit. Patch by Victor Stinner.

- gh-95921 [https://github.com/python/cpython/issues/95921]: Fix overly-broad source position information for chained comparisons used as branching conditions.

- gh-96821 [https://github.com/python/cpython/issues/96821]: Fix undefined behaviour in `audioop.c`.

- gh-96821 [https://github.com/python/cpython/issues/96821]: Fix undefined behaviour in `_testcapimodule.c`.

- gh-95778 [https://github.com/python/cpython/issues/95778]: When **`ValueError`** is raised if an integer is larger than the limit, mention the **`sys.set_int_max_str_digits()`** function in the error message. Patch by Victor Stinner.

- gh-96387 [https://github.com/python/cpython/issues/96387]: At Python exit, sometimes a thread holding the GIL can wait forever for a thread (usually a daemon thread) which requested to drop the GIL, whereas the thread already exited. To fix the race condition, the thread which requested the GIL drop now resets its request before exiting. Issue discovered and analyzed by Mingliang ZHAO. Patch by Victor Stinner.

- gh-96864 [https://github.com/python/cpython/issues/96864]: Fix a possible assertion failure, fatal error, or **`SystemError`** if a line tracing event raises an exception while opcode tracing is enabled.

- gh-95778 [https://github.com/python/cpython/issues/95778]: The `PyLong_FromString` function was refactored to make it more maintainable and extensible.

- gh-96678 [https://github.com/python/cpython/issues/96678]: Fix undefined behaviour in C code of null pointer arithmetic.

- gh-96754 [https://github.com/python/cpython/issues/96754]: Make sure that all frame objects created are created from valid interpreter frames. Prevents the possibility of invalid frames in backtraces and signal handlers.

- gh-90997 [https://github.com/python/cpython/issues/90997]: Improve the performance of reading and writing inline bytecode caches on some platforms.

- gh-96751 [https://github.com/python/cpython/issues/96751]: Remove dead code from `CALL_FUNCTION_EX` opcode.

- gh-90751 [https://github.com/python/cpython/issues/90751]: **`memoryview`** now supports half-floats. Patch by Donghee Na and Antoine Pitrou.

- gh-96678 [https://github.com/python/cpython/issues/96678]: Fix case of undefined behavior in ceval.c

- gh-64373 [https://github.com/python/cpython/issues/64373]: Convert **`_functools`** to argument clinic.

- gh-96641 [https://github.com/python/cpython/issues/96641]: Do not expose `KeyWrapper` in **`_functools`**.

- gh-96636 [https://github.com/python/cpython/issues/96636]: Ensure that tracing, `sys.setrace()`, is turned on immediately. In pre-release versions of 3.11, some tracing events might have been lost when turning on tracing in a `__del__` method or interrupt.

- gh-96572 [https://github.com/python/cpython/issues/96572]: Fix use after free in trace refs build mode. Patch by Kumar Aditya.

- gh-96611 [https://github.com/python/cpython/issues/96611]: When loading a file with invalid UTF-8 inside a multi-line string, a correct SyntaxError is emitted.

- gh-96612 [https://github.com/python/cpython/issues/96612]: Make sure that incomplete frames do not show up in tracemalloc traces.

- gh-90230 [https://github.com/python/cpython/issues/90230]: Fix compiler warnings and test failures when building with `--enable-pystats`.

- gh-96587 [https://github.com/python/cpython/issues/96587]: Correctly raise `SyntaxError` on exception groups (**PEP 654** [https://peps.python.org/pep-0654/]) on python versions prior to 3.11

- gh-96569 [https://github.com/python/cpython/issues/96569]: Remove two cases of undefined behavior, by adding NULL checks.

- gh-96582 [https://github.com/python/cpython/issues/96582]: Fix possible `NULL` pointer dereference in `_PyThread_CurrentFrames`. Patch by Kumar Aditya.

- gh-91079 [https://github.com/python/cpython/issues/91079]: Separate Python recursion checking from C recursion checking which reduces the chance of C stack overflow and allows the recursion limit to be increased safely.

- gh-93911 [https://github.com/python/cpython/issues/93911]: Fix an issue that could prevent `LOAD_ATTR` from specializing properly when accessing properties.

- gh-96348 [https://github.com/python/cpython/issues/96348]: Emit a DeprecationWarning when `throw()`, `throw()` or `athrow()` are called with more than one argument.

- gh-95196 [https://github.com/python/cpython/issues/95196]: Disable incorrect pickling of the C implemented classmethod descriptors.

- gh-96364 [https://github.com/python/cpython/issues/96364]: Fix text signatures of `list.__getitem__` and `dict.__getitem__`.

- gh-96352 [https://github.com/python/cpython/issues/96352]: Fix `AttributeError` missing `name` and `obj` attributes in `object.__getattribute__()`. Patch by Philip Georgi.

- gh-93554 [https://github.com/python/cpython/issues/93554]: Change the jump opcodes so that all conditional jumps are forward jumps. Backward jumps are converted by the assembler into a conditional forward jump whose target is the fallthrough block (and with a reversed condition), followed by an unconditional backward jump. For example:

  `POP_JUMP_IF_TRUE BACKWARD_TARGET` becomes `POP_JUMP_IF_FALSE NEXT_BLOCK; JUMP BACKWARD_TARGET`.

  All the directed conditional jump opcodes were removed: `POP_JUMP_FORWARD_IF_TRUE`, `POP_JUMP_BACKWARD_IF_TRUE`, `POP_JUMP_FORWARD_IF_FALSE`, `POP_JUMP_BACKWARD_IF_FALSE`, `POP_JUMP_FORWARD_IF_NONE`, `POP_JUMP_BACKWARD_IF_NONE`, `POP_JUMP_FORWARD_IF_NOT_NONE`, `POP_JUMP_BACKWARD_IF_NOT_NONE`.

  The corresponding opcodes without direction are no longer pseudo-instructions, and they implement the forward conditional jumps.

- gh-96268 [https://github.com/python/cpython/issues/96268]: Loading a file with invalid UTF-8 will now report the broken character at the correct location.

- gh-96237 [https://github.com/python/cpython/issues/96237]: The internal field `_PyInterpreterFrame.f_func` is renamed to `_PyInterpreterFrame.f_funcobj` and may be any object. The `f_globals` and `f_builtin` fields may hold junk values.

  It is safest to treat the `_PyInterpreterFrame` struct as opaque.

- gh-96187 [https://github.com/python/cpython/issues/96187]: Fixed a bug that caused `_PyCode_GetExtra` to return garbage for negative indexes. Patch by Pablo Galindo

- gh-96143 [https://github.com/python/cpython/issues/96143]: Add a new `-X perf` Python command line option as well as **`sys.activate_stack_trampoline()`** and **`sys.deactivate_stack_trampoline()`** function in the **`sys`** module that allows to set/unset the interpreter in a way that the Linux `perf` profiler can detect Python calls. The new **`sys.is_stack_trampoline_active()`** function allows to query the state of the perf trampoline. Design by Pablo Galindo. Patch by Pablo Galindo and Christian Heimes with contributions from Gregory P. Smith [Google] and Mark Shannon.

- gh-96071 [https://github.com/python/cpython/issues/96071]: Fix a deadlock in **`PyGILState_Ensure()`** when allocating new thread state. Patch by Kumar Aditya.

- gh-96046 [https://github.com/python/cpython/issues/96046]: **`PyType_Ready()`** now initializes `ht_cached_keys` and performs additional checks to ensure that type objects are properly configured. This avoids crashes in 3rd party packages that don’t use regular API to create new types.

- gh-96005 [https://github.com/python/cpython/issues/96005]: On WASI **`ENOTCAPABLE`** is now mapped to **`PermissionError`**. The **`errno`** modules exposes the new error number. `getpath.py` now ignores **`PermissionError`** when it cannot open landmark files `pybuilddir.txt` and `pyenv.cfg`.

- gh-93678 [https://github.com/python/cpython/issues/93678]: Added test a harness for direct unit tests of the compiler’s optimization stage. The `_testinternalcapi.optimize_cfg()` function runs the optimiser on a sequence of instructions. The `CfgOptimizationTestCase` class in `test.support` has utilities for invoking the optimizer and checking the output.

- gh-95245 [https://github.com/python/cpython/issues/95245]: Reduces the size of a “simple” Python object from 8 to 6 words by moving the weakreflist pointer into the pre-header directly before the object’s dict/values pointer.

- gh-90997 [https://github.com/python/cpython/issues/90997]: Compile virtual `try`/`except` blocks to handle exceptions raised during `close()` or `throw()` calls through a suspended frame.

- gh-95977 [https://github.com/python/cpython/issues/95977]: Optimized calling `__get__()` with vectorcall. Patch by Kumar Aditya.

- gh-91210 [https://github.com/python/cpython/issues/91210]: Improve error message when a parameter without a default value follows one with a default value, and show the same message, even when the non-default/default sequence is preceded by positional-only parameters.

- gh-95922 [https://github.com/python/cpython/issues/95922]: Fixed bug where the compiler’s `eliminate_empty_basic_blocks` function ignores the last block of the code unit.

- gh-95818 [https://github.com/python/cpython/issues/95818]: Skip over incomplete frames in `PyThreadState_GetFrame()`.

- gh-95876 [https://github.com/python/cpython/issues/95876]: Fix format string in `_PyPegen_raise_error_known_location` that can lead to memory corruption on some 64bit systems. The function was building a tuple with `i` (int) instead of `n` (Py_ssize_t) for Py_ssize_t arguments.

- gh-95605 [https://github.com/python/cpython/issues/95605]: Fix misleading contents of error message when converting an all-whitespace string to `float`.

- gh-95150 [https://github.com/python/cpython/issues/95150]: Update code object hashing and equality to consider all debugging and exception handling tables. This fixes an issue where certain non-identical code objects could be “deduplicated” during compilation.

- gh-91146 [https://github.com/python/cpython/issues/91146]: Reduce allocation size of **`list`** from **`str.split()`** and **`str.rsplit()`**. Patch by Donghee Na and Inada Naoki.

- gh-87092 [https://github.com/python/cpython/issues/87092]: Create a ‘jump target label’ abstraction in the compiler so that the compiler’s codegen stage does not work directly with basic blocks. This prepares the code for changes to the underlying CFG generation mechanism.

- gh-95355 [https://github.com/python/cpython/issues/95355]: `_PyPegen_Parser_New` now properly detects token memory allocation errors. Patch by Honglin Zhu.

- gh-90081 [https://github.com/python/cpython/issues/90081]: Run Python code in tracer/profiler function at full speed. Fixes slowdown in earlier versions of 3.11.

- gh-95324 [https://github.com/python/cpython/issues/95324]: Emit a warning in debug mode if an object does not call **`PyObject_GC_UnTrack()`** before deallocation. Patch by Pablo Galindo.

- gh-95245 [https://github.com/python/cpython/issues/95245]: Merge managed dict and values pointer into a single tagged pointer to save one word in the pre-header.

- gh-93678 [https://github.com/python/cpython/issues/93678]: Add cfg_builder struct and refactor the relevant code so that a cfg can be constructed without an instance of the compiler struct.

- gh-95185 [https://github.com/python/cpython/issues/95185]: Prevented crashes in the AST constructor when compiling some absurdly long expressions like `"+0"*1000000`. **`RecursionError`** is now raised instead. Patch by Pablo Galindo

- gh-93351 [https://github.com/python/cpython/issues/93351]: **`ast.AST`** node positions are now validated when provided to **`compile()`** and other related functions. If invalid positions are detected, a **`ValueError`** will be raised.

- gh-94438 [https://github.com/python/cpython/issues/94438]: Fix an issue that caused extended opcode arguments and some conditional pops to be ignored when calculating valid jump targets for assignments to the `f_lineno` attribute of frame objects. In some cases, this could cause inconsistent internal state, resulting in a hard crash of the interpreter.

- gh-95060 [https://github.com/python/cpython/issues/95060]: Undocumented `PyCode_Addr2Location` function now properly returns when `addrq` argument is less than zero.

- gh-95113 [https://github.com/python/cpython/issues/95113]: Replace all `EXTENDED_ARG_QUICK` instructions with basic **`EXTENDED_ARG`** instructions in unquickened code. Consumers of non-adaptive bytecode should be able to handle extended arguments the same way they were handled in CPython 3.10 and older.

- gh-91409 [https://github.com/python/cpython/issues/91409]: Fix incorrect source location info caused by certain optimizations in the bytecode compiler.

- gh-95023 [https://github.com/python/cpython/issues/95023]: Implement **`os.setns()`** and **`os.unshare()`** for Linux. Patch by Noam Cohen.

- gh-94036 [https://github.com/python/cpython/issues/94036]: Fix incorrect source location info for some multi-line attribute accesses and method calls.

- gh-94938 [https://github.com/python/cpython/issues/94938]: Fix error detection in some builtin functions when keyword argument name is an instance of a str subclass with overloaded `__eq__` and `__hash__`. Previously it could cause SystemError or other undesired behavior.

- gh-94996 [https://github.com/python/cpython/issues/94996]: `ast.parse()` will no longer parse function definitions with positional-only params when passed `feature_version` less than `(3, 8)`. Patch by Shantanu Jain.

- gh-94739 [https://github.com/python/cpython/issues/94739]: Allow jumping within, out of, and across exception handlers in the debugger.

- gh-94949 [https://github.com/python/cpython/issues/94949]: `ast.parse()` will no longer parse parenthesized context managers when passed `feature_version` less than `(3, 9)`. Patch by Shantanu Jain.

- gh-94947 [https://github.com/python/cpython/issues/94947]: `ast.parse()` will no longer parse assignment expressions when passed `feature_version` less than `(3, 8)`. Patch by Shantanu Jain.

- gh-91256 [https://github.com/python/cpython/issues/91256]: Ensures the program name is known for help text during interpreter startup.

- gh-94869 [https://github.com/python/cpython/issues/94869]: Fix the column offsets for some expressions in multi-line f-strings `ast` nodes. Patch by Pablo Galindo.

- gh-94893 [https://github.com/python/cpython/issues/94893]: Fix an issue where frame object manipulations could corrupt inline bytecode caches.

- gh-94822 [https://github.com/python/cpython/issues/94822]: Fix an issue where lookups of metaclass descriptors may be ignored when an identically-named attribute also exists on the class itself.

- gh-91153 [https://github.com/python/cpython/issues/91153]: Fix an issue where a `bytearray` item assignment could crash if it's resized by the new value's `__index__()` method.

- gh-90699 [https://github.com/python/cpython/issues/90699]: Fix reference counting bug in `bool.__repr__()`. Patch by Kumar Aditya.

- gh-94694 [https://github.com/python/cpython/issues/94694]: Fix an issue that could cause code with multi-line method lookups to have misleading or incorrect column offset information. In some cases (when compiling a hand-built AST) this could have resulted in a hard crash of the interpreter.

- gh-93252 [https://github.com/python/cpython/issues/93252]: Fix an issue that caused internal frames to outlive failed Python function calls, possibly resulting in memory leaks or hard interpreter crashes.

- gh-94215 [https://github.com/python/cpython/issues/94215]: Fix an issue where exceptions raised by line-tracing events would cause frames to be left in an invalid state, possibly resulting in a hard crash of the interpreter.

- gh-92228 [https://github.com/python/cpython/issues/92228]: Disable the compiler's inline-small-exit-blocks optimization for exit blocks that are associated with source code lines.

This fixes a bug where the debugger cannot tell where an exception handler ends and the following code block begins.

- gh-94485 [https://github.com/python/cpython/issues/94485]: Line number of a module’s `RESUME` instruction is set to 0 as specified in **PEP 626** [https://peps.python.org/pep-0626/].

- gh-94438 [https://github.com/python/cpython/issues/94438]: Account for instructions that can push NULL to the stack when setting line number in a frame. Prevents some (unlikely) crashes.

- gh-91719 [https://github.com/python/cpython/issues/91719]: Reload `opcode` when raising `unknown opcode error` in the interpreter main loop, for C compilers to generate dispatching code independently.

- gh-94329 [https://github.com/python/cpython/issues/94329]: Compile and run code with unpacking of extremely large sequences (1000s of elements). Such code failed to compile. It now compiles and runs correctly.

- gh-94360 [https://github.com/python/cpython/issues/94360]: Fixed a tokenizer crash when reading encoded files with syntax errors from `stdin` with non utf-8 encoded text. Patch by Pablo Galindo

- gh-88116 [https://github.com/python/cpython/issues/88116]: Fix an issue when reading line numbers from code objects if the encoded line numbers are close to `INT_MIN`. Patch by Pablo Galindo

- gh-94262 [https://github.com/python/cpython/issues/94262]: Don’t create frame objects for incomplete frames. Prevents the creation of generators and closures from being observable to Python and C extensions, restoring the behavior of 3.10 and earlier.

- gh-94192 [https://github.com/python/cpython/issues/94192]: Fix error for dictionary literals with invalid expression as value.
- gh-87995 [https://github.com/python/cpython/issues/87995]: `types.MappingProxyType` instances are now hashable if the underlying mapping is hashable.
- gh-93883 [https://github.com/python/cpython/issues/93883]: Revise the display strategy of traceback enhanced error locations. The indicators are only shown when the location doesn’t span the whole line.
- gh-94163 [https://github.com/python/cpython/issues/94163]: Add **`BINARY_SLICE`** and **`STORE_SLICE`** instructions for more efficient handling and better specialization of slicing operations, where the slice is explicit in the source code.
- gh-94021 [https://github.com/python/cpython/issues/94021]: Fix unreachable code warning in `Python/specialize.c`.
- gh-93911 [https://github.com/python/cpython/issues/93911]: Specialize `LOAD_ATTR` for objects with custom `__getattribute__`.
- gh-93955 [https://github.com/python/cpython/issues/93955]: Improve performance of attribute lookups on objects with custom `__getattribute__` and `__getattr__`. Patch by Ken Jin.
- gh-93911 [https://github.com/python/cpython/issues/93911]: Specialize `LOAD_ATTR` for `property()` attributes.

- gh-93678 [https://github.com/python/cpython/issues/93678]: Refactor compiler optimisation code so that it no longer needs the `struct assembler` and `struct compiler` passed around. Instead, each function takes the CFG and other data that it actually needs. This will make it possible to test this code directly.

- gh-93841 [https://github.com/python/cpython/issues/93841]: When built with `-enable-pystats`, `sys._stats_on()`, `sys._stats_off()`, `sys._stats_clear()` and `sys._stats_dump()` functions have been added to enable gathering stats for parts of programs.

- gh-93516 [https://github.com/python/cpython/issues/93516]: Store offset of first traceable instruction in code object to avoid having to recompute it for each instruction when tracing.

- gh-93516 [https://github.com/python/cpython/issues/93516]: Lazily create a table mapping bytecode offsets to line numbers to speed up calculation of line numbers when tracing.

- gh-89828 [https://github.com/python/cpython/issues/89828]: **`types.GenericAlias`** no longer relays the `__class__` attribute. For example, `isinstance(list[int], type)` no longer returns `True`.

- gh-93678 [https://github.com/python/cpython/issues/93678]: Refactor the compiler to reduce boilerplate and repetition.

- gh-93671 [https://github.com/python/cpython/issues/93671]: Fix some exponential backtrace case happening with deeply nested sequence patterns in match statements. Patch by Pablo Galindo

- gh-93662 [https://github.com/python/cpython/issues/93662]: Make sure that the end column offsets are correct in multi-line method calls. Previously, the end column could precede the column offset.

- gh-93461 [https://github.com/python/cpython/issues/93461]: `importlib.invalidate_caches()` now drops entries from `sys.path_importer_cache` with a relative path as name. This solves a caching issue when a process changes its current working directory.

  `FileFinder` no longer inserts a dot in the path, e.g. `/egg/./spam` is now `/egg/spam`.

- gh-93621 [https://github.com/python/cpython/issues/93621]: Change order of bytecode instructions emitted for `with` and `async with` to reduce the number of entries in the exception table.

- gh-93533 [https://github.com/python/cpython/issues/93533]: Reduce the size of the inline cache for `LOAD_METHOD` by 2 bytes.

- gh-93444 [https://github.com/python/cpython/issues/93444]: Removed redundant fields from the compiler's basicblock struct: `b_nofallthrough`, `b_exit`, `b_return`. They can be easily calculated from the opcode of the last instruction of the block.

- gh-93429 [https://github.com/python/cpython/issues/93429]: `LOAD_METHOD` instruction has been removed. It was merged back into `LOAD_ATTR`.

- gh-93418 [https://github.com/python/cpython/issues/93418]: Fixed an assert where an f-string has an equal sign '=' following an expression, but there's no trailing brace. For example, f"{i=".

- gh-93382 [https://github.com/python/cpython/issues/93382]: Cache the result of **`PyCode_GetCode()`** function to restore the $O(1)$ lookup of the **`co_code`** attribute.

- gh-93359 [https://github.com/python/cpython/issues/93359]: Ensure that custom **`ast`** nodes without explicit end positions can be compiled. Patch by Pablo Galindo.

- gh-93356 [https://github.com/python/cpython/issues/93356]: Code for exception handlers is emitted at the end of the code unit’s bytecode. This avoids one jump when no exception is raised.

- gh-93354 [https://github.com/python/cpython/issues/93354]: Use exponential backoff for specialization counters in the interpreter. Can reduce the number of failed specializations significantly and avoid slowdown for those parts of a program that are not suitable for specialization.

- gh-93283 [https://github.com/python/cpython/issues/93283]: Improve error message for invalid syntax of conversion character in f-string expressions.

- gh-93345 [https://github.com/python/cpython/issues/93345]: Fix a crash in substitution of a `TypeVar` in nested generic alias after `TypeVarTuple`.

- gh-93223 [https://github.com/python/cpython/issues/93223]: When a bytecode instruction jumps to an unconditional jump instruction, the first instruction can often be optimized to target the unconditional jump’s target directly. For tracing reasons, this would previously only occur if both instructions have the same line number. This also now occurs if the unconditional jump is artificial, i.e., if it has no associated line number.

- gh-84694 [https://github.com/python/cpython/issues/84694]: The `--experimental-isolated-subinterpreters` configure option and `EXPERIMENTAL_ISOLATED_SUBINTERPRETERS` macro have been removed.

- gh-91924 [https://github.com/python/cpython/issues/91924]: Fix `__lltrace__` debug feature if the stdout encoding is not UTF-8. Patch by Victor Stinner.

- gh-93040 [https://github.com/python/cpython/issues/93040]: Wraps unused parameters in `Objects/obmalloc.c` with `Py_UNUSED`.

- gh-93143 [https://github.com/python/cpython/issues/93143]: Avoid `NULL` checks for uninitialized local variables by determining at compile time which variables must be initialized.

- gh-93061 [https://github.com/python/cpython/issues/93061]: Backward jumps after `async for` loops are no longer given dubious line numbers.

- gh-93065 [https://github.com/python/cpython/issues/93065]: Fix contextvars HAMT implementation to handle iteration over deep trees.

  The bug was discovered and fixed by Eli Libman. See MagicStack/immutables#84 [https://github.com/MagicStack/immutables/issues/84] for more details.

- gh-93012 [https://github.com/python/cpython/issues/93012]: Added the new function **`PyType_FromMetaclass()`**, which generalizes the existing **`PyType_FromModuleAndSpec()`** using an additional metaclass argument. This is useful for language binding tools, where it can be used to intercept type-related operations like subclassing or static attribute access by specifying a metaclass with custom slots.

Importantly, `PyType_FromMetaclass()` is available in the Limited API, which provides a path towards migrating more binding tools onto the Stable ABI.

- gh-93021 [https://github.com/python/cpython/issues/93021]: Fix the **`__text_signature__`** for **`__get__()`** methods implemented in C. Patch by Jelle Zijlstra.
- gh-89914 [https://github.com/python/cpython/issues/89914]: The operand of the `YIELD_VALUE` instruction is set to the stack depth. This is done to help frame handling on `yield` and may assist debuggers.
- gh-92955 [https://github.com/python/cpython/issues/92955]: Fix memory leak in code object's lines and positions iterators as they were not finalized at exit. Patch by Kumar Aditya.
- gh-92930 [https://github.com/python/cpython/issues/92930]: Fixed a crash in `_pickle.c` from mutating collections during `__reduce__` or `persistent_id`.
- gh-90690 [https://github.com/python/cpython/issues/90690]: The PRECALL instruction has been removed. It offered only a small advantage for specialization and is not needed in the vast majority of cases.
- gh-92914 [https://github.com/python/cpython/issues/92914]: Always round the allocated size for lists up to the nearest even number.
- gh-92858 [https://github.com/python/cpython/issues/92858]: Improve error message for some suites with syntax error before ':'

- gh-90473 [https://github.com/python/cpython/issues/90473]: Decrease default recursion limit on WASI to address limited call stack size.

- gh-92804 [https://github.com/python/cpython/issues/92804]: Fix memory leak in `memoryview` iterator as it was not finalized at exit. Patch by Kumar Aditya.

- gh-92777 [https://github.com/python/cpython/issues/92777]: Specialize `LOAD_METHOD` for objects with lazy dictionaries. Patch by Ken Jin.

- gh-92658 [https://github.com/python/cpython/issues/92658]: Add support for connecting and binding to Hyper-V sockets on Windows Hyper-V hosts and guests.

- gh-92236 [https://github.com/python/cpython/issues/92236]: Remove spurious “LINE” event when starting a generator or coroutine, visible tracing functions implemented in C.

- gh-91102 [https://github.com/python/cpython/issues/91102]: **`_warnings.warn_explicit()`** is ported to Argument Clinic.

- gh-92619 [https://github.com/python/cpython/issues/92619]: Make the compiler duplicate an exit block only if none of its instructions have a lineno (previously only the first instruction in the block was checked, leading to unnecessarily duplicated blocks).

- gh-88750 [https://github.com/python/cpython/issues/88750]: The deprecated debug build only `PYTHONTHREADDEBUG` environment variable no longer does anything.

- gh-92261 [https://github.com/python/cpython/issues/92261]: Fix hang when trying to iterate over a `typing.Union`.

- gh-91432 [https://github.com/python/cpython/issues/91432]: Specialized the **`FOR_ITER`** opcode using the PEP 659 machinery

- gh-91399 [https://github.com/python/cpython/issues/91399]: Removed duplicate '{0, 0, 0, 0, 0, 0}' entry in 'Objects/unicodetype_db.h'.

- gh-91578 [https://github.com/python/cpython/issues/91578]: Updates the error message for abstract class.

- bpo-47091 [https://bugs.python.org/issue?@action=redirect&bpo=47091]: Improve performance of repetition of **`list`** and **`tuple`** by using `memcpy` to copy data and performing the reference increments in one step.

- bpo-46142 [https://bugs.python.org/issue?@action=redirect&bpo=46142]: Make `--help` output shorter by moving some info to the new `--help-env` and `--help-xoptions` command-line options. Also add `--help-all` option to print complete usage.

- bpo-42316 [https://bugs.python.org/issue?@action=redirect&bpo=42316]: Document some places where an assignment expression needs parentheses.

Library

- gh-89237 [https://github.com/python/cpython/issues/89237]: Fix hang on Windows in `subprocess.wait_closed()` in **`asyncio`** with **`ProactorEventLoop`**. Patch by Kumar Aditya.

- gh-97928 [https://github.com/python/cpython/issues/97928]: **`tkinter.Text.count()`** raises now an exception for options starting with "-" instead of silently ignoring them.

- gh-98393 [https://github.com/python/cpython/issues/98393]: The **`os`** module no longer accepts bytes-like paths, like **`bytearray`** and **`memoryview`** types: only the exact **`bytes`** type is accepted for bytes strings. Patch by Victor Stinner.

- gh-98363 [https://github.com/python/cpython/issues/98363]: Added itertools.batched() to batch data into lists of a given length with the last list possibly being shorter than the others.

- gh-98331 [https://github.com/python/cpython/issues/98331]: Update the bundled copies of pip and setuptools to versions 22.3 and 65.5.0 respectively.

- gh-98307 [https://github.com/python/cpython/issues/98307]: A **`createSocket()`** method was added to **`SysLogHandler`**.

- gh-96035 [https://github.com/python/cpython/issues/96035]: Fix bug in **`urllib.parse.urlparse()`** that causes certain port numbers containing whitespace, underscores, plus and minus signs, or non-ASCII digits to be incorrectly accepted.

- gh-98257 [https://github.com/python/cpython/issues/98257]: Make **`sys.setprofile()`** and **`sys.settrace()`** functions reentrant. They can no long fail with: `RuntimeError("Cannot install a trace function while another trace function is being installed")`. Patch by Victor Stinner.

- gh-98251 [https://github.com/python/cpython/issues/98251]: Allow **`venv`** to pass along **`PYTHON*`** variables to `ensurepip` and `pip` when they do not impact path resolution

- gh-94597 [https://github.com/python/cpython/issues/94597]: Deprecated **`asyncio.AbstractEventLoopPolicy.get_child_watcher()`** and **`asyncio.AbstractEventLoopPolicy.set_child_watcher()`**

methods to be removed in Python 3.14. Patch by Kumar Aditya.

- gh-98178 [https://github.com/python/cpython/issues/98178]: On macOS, fix a crash in **`syslog.syslog()`** in multi-threaded applications. On macOS, the libc `syslog()` function is not thread-safe, so **`syslog.syslog()`** no longer releases the GIL to call it. Patch by Victor Stinner.

- gh-44098 [https://github.com/python/cpython/issues/44098]: Release the GIL when creating **`mmap.mmap`** objects on Unix.

- gh-87730 [https://github.com/python/cpython/issues/87730]: Wrap network errors consistently in urllib FTP support, so the test suite doesn’t fail when a network is available but the public internet is not reachable.

- gh-94597 [https://github.com/python/cpython/issues/94597]: The child watcher classes **`MultiLoopChildWatcher`**, **`FastChildWatcher`** and **`SafeChildWatcher`** are deprecated and will be removed in Python 3.14. Patch by Kumar Aditya.

- gh-98023 [https://github.com/python/cpython/issues/98023]: Change default child watcher to **`PidfdChildWatcher`** on Linux systems which supports it. Patch by Kumar Aditya.

- gh-90985 [https://github.com/python/cpython/issues/90985]: Earlier in 3.11 we deprecated `asyncio.Task.cancel("message")`. We realized we were too harsh, and have undeprecated it.

- gh-65961 [https://github.com/python/cpython/issues/65961]: Do not rely solely on `__cached__` on modules; code will also support `__spec__.cached`.

- gh-97646 [https://github.com/python/cpython/issues/97646]: Replace deprecated `application/javascript` with `text/javascript` in **`mimetypes`**. See **RFC 9239** [https://datatracker.ietf.org/doc/html/rfc9239.html]. Patch by Noam Cohen.

- gh-97930 [https://github.com/python/cpython/issues/97930]: Apply changes from importlib_resources 5.8 and 5.9: `Traversable.joinpath` provides a concrete implementation. `as_file` now supports directories of resources.

- gh-97850 [https://github.com/python/cpython/issues/97850]: Remove deprecated **`importlib.util.set_loader()`** and **`importlib.util.module_for_loader()`** from **`importlib.util`**.

- gh-97837 [https://github.com/python/cpython/issues/97837]: Change deprecate warning message in **`unittest`** from

  ```
  It is deprecated to return a value!=None
  ```

  to

  ```
  It is deprecated to return a value that is not None from a
  test case
  ```

- gh-97825 [https://github.com/python/cpython/issues/97825]: Fixes **`AttributeError`** when **`subprocess.check_output()`** is used with argument `input=None` and either of the arguments *encoding* or *errors* are used.

- gh-97008 [https://github.com/python/cpython/issues/97008]: **`NameError`** and **`AttributeError`** spelling suggestions provided since gh-82711 [https://github.com/python/cpython/issues/82711] are now also emitted by the pure Python **`traceback`** module. Tests for those suggestions now exercise both implementations to ensure they are equivalent. Patch by Carl Friedrich Bolz-Tereick and Łukasz Langa.

- gh-97799 [https://github.com/python/cpython/issues/97799]: **`dataclass`** now uses **`inspect.get_annotations()`** to examine the annotations on class objects.

- gh-97781 [https://github.com/python/cpython/issues/97781]: Removed deprecated interfaces in `importlib.metadata` (entry points accessed as dictionary, implicit dictionary construction of sequence of `EntryPoint` objects, mutablility of `EntryPoints` result, access of entry point by index). `entry_points` now has a simpler, more straightforward API (returning `EntryPoints`).

- gh-96827 [https://github.com/python/cpython/issues/96827]: Avoid spurious tracebacks from **`asyncio`** when default executor cleanup is delayed until after the event loop is closed (e.g. as the result of a keyboard interrupt).

- gh-95534 [https://github.com/python/cpython/issues/95534]: **`gzip.GzipFile.read()`** reads 10% faster.

- gh-97592 [https://github.com/python/cpython/issues/97592]: Avoid a crash in the C version of **`asyncio.Future.remove_done_callback()`** when an evil argument is passed.

- gh-97639 [https://github.com/python/cpython/issues/97639]: Remove `tokenize.NL` check from **`tabnanny`**.

- gh-97545 [https://github.com/python/cpython/issues/97545]: Make Semaphore run faster.

- gh-73588 [https://github.com/python/cpython/issues/73588]: Fix generation of the default name of **`tkinter.Checkbutton`**. Previously, checkbuttons in different parent widgets could have the same short name and share the same state if arguments “name” and “variable” are not specified. Now they are globally unique.

- gh-96865 [https://github.com/python/cpython/issues/96865]: fix Flag to use boundary CONFORM

  This restores previous Flag behavior of allowing flags with non-sequential values to be combined; e.g.

  class Skip(Flag): TWO = 2 EIGHT = 8

  Skip.TWO | Skip.EIGHT -> <Skip.TWO|EIGHT: 10>

- gh-97005 [https://github.com/python/cpython/issues/97005]: Update bundled libexpat to 2.4.9

- gh-85760 [https://github.com/python/cpython/issues/85760]: Fix race condition in **`asyncio`** where **`process_exited()`** called before the **`pipe_data_received()`** leading to inconsistent output. Patch by Kumar Aditya.

- gh-96704 [https://github.com/python/cpython/issues/96704]: Pass the correct `contextvars.Context` when a `asyncio` exception handler is called on behalf of a task or callback handle. This adds a new `Task` method, `get_context`, and also a new `Handle` method with the same name. If this method is not found on a task object (perhaps because it is a third-party library that does not yet provide this method), the context prevailing at the time the exception handler is called is used.

- gh-96819 [https://github.com/python/cpython/issues/96819]: Fixed check in **`multiprocessing.resource_tracker`** that guarantees that the length of a write to a pipe is not greater than `PIPE_BUF`.

- gh-95865 [https://github.com/python/cpython/issues/95865]: Reduce **`urllib.parse.quote_from_bytes()`** memory use on large values.

Contributed by Dennis Sweeney.

- gh-96741 [https://github.com/python/cpython/issues/96741]: Corrected type annotation for dataclass attribute `pstats.FunctionProfile.ncalls` to be `str`.

- gh-96734 [https://github.com/python/cpython/issues/96734]: Update **`unicodedata`** database to Unicode 15.0.0.

- gh-96735 [https://github.com/python/cpython/issues/96735]: Fix undefined behaviour in **`struct.unpack()`**.

- gh-46412 [https://github.com/python/cpython/issues/46412]: Improve performance of `bool(db)` for large ndb/gdb databases. Previously this would call `len(db)` which would iterate over all keys – the answer (empty or not) is known after the first key.

- gh-96652 [https://github.com/python/cpython/issues/96652]: Fix the faulthandler implementation of `faulthandler.register(signal, chain=True)` if the `sigaction()` function is not available: don’t call the previous signal handler if it’s NULL. Patch by Victor Stinner.

- gh-68163 [https://github.com/python/cpython/issues/68163]: Correct conversion of **`numbers.Rational`**’s to **`float`**.

- gh-96538 [https://github.com/python/cpython/issues/96538]: Speed up `bisect.bisect()` functions by taking advantage of type-stability.

- gh-96465 [https://github.com/python/cpython/issues/96465]: Fraction hashes are now cached.

- gh-96079 [https://github.com/python/cpython/issues/96079]: In **`typing`**, fix missing field `name` and incorrect `__module__` in _AnnotatedAlias.

- gh-96415 [https://github.com/python/cpython/issues/96415]: Remove `types._cell_factory` from module namespace.

- gh-95987 [https://github.com/python/cpython/issues/95987]: Fix `repr` of `Any` subclasses.

- gh-96388 [https://github.com/python/cpython/issues/96388]: Work around missing socket functions in **`socket`**'s `__repr__`.

- gh-96385 [https://github.com/python/cpython/issues/96385]: Fix `TypeVarTuple.__typing_prepare_subst__`. `TypeError` was not raised when using more than one `TypeVarTuple`, like `[*T, *V]` in type alias substitutions.

- gh-96142 [https://github.com/python/cpython/issues/96142]: Add `match_args`, `kw_only`, `slots`, and `weakref_slot` to `_DataclassParams`.

- gh-96073 [https://github.com/python/cpython/issues/96073]: In **`inspect`**, fix overeager replacement of "`typing.`" in formatting annotations.

- gh-89258 [https://github.com/python/cpython/issues/89258]: Added a **`getChildren()`** method to **`logging.Logger`**, to get the immediate child loggers of a logger.

- gh-96346 [https://github.com/python/cpython/issues/96346]: Use double caching for compiled RE patterns.

- gh-96349 [https://github.com/python/cpython/issues/96349]: Fixed a minor performance regression in **`threading.Event.__init__()`**

- gh-90467 [https://github.com/python/cpython/issues/90467]: Fix **`asyncio.streams.StreamReaderProtocol`** to keep a strong reference to the created task, so that it's not garbage collected

- gh-96172 [https://github.com/python/cpython/issues/96172]: Fix a bug in `unicodedata`: `east_asian_width` used to return the wrong value for unassigned characters; and for yet unassigned, but reserved characters.

- gh-96159 [https://github.com/python/cpython/issues/96159]: Fix a performance regression in logging TimedRotatingFileHandler. Only check for special files when the rollover time has passed.

- gh-96175 [https://github.com/python/cpython/issues/96175]: Fix unused `localName` parameter in the `Attr` class in **`xml.dom.minidom`**.

- gh-96145 [https://github.com/python/cpython/issues/96145]: Add AttrDict to JSON module for use with object_hook.

- gh-96052 [https://github.com/python/cpython/issues/96052]: Fix handling compiler warnings (SyntaxWarning and DeprecationWarning) in **`codeop.compile_command()`** when checking for incomplete input. Previously it emitted warnings and raised a SyntaxError. Now it always returns `None` for incomplete input without emitting any warnings.

- gh-96125 [https://github.com/python/cpython/issues/96125]: Fix incorrect condition that causes `sys.thread_info.name` to be wrong on pthread platforms.

- gh-96019 [https://github.com/python/cpython/issues/96019]: Fix a bug in the `makeunicodedata.py` script leading to about 13 KiB of space saving in the `unicodedata` module, specifically the character decomposition data.

- gh-95463 [https://github.com/python/cpython/issues/95463]: Remove an incompatible change from bpo-28080 [https://bugs.python.org/issue?@action=redirect&bpo=28080] that caused a regression that ignored the utf8 in `ZipInfo.flag_bits`. Patch by Pablo Galindo.

- gh-69142 [https://github.com/python/cpython/issues/69142]: Add `%:z` strftime format code (generates tzoffset with colons as separator), see strftime() and strptime() Behavior.
- gh-95899 [https://github.com/python/cpython/issues/95899]: Fix **`asyncio.Runner`** to call **`asyncio.set_event_loop()`** only once to avoid calling **`attach_loop()`** multiple times on child watchers. Patch by Kumar Aditya.
- gh-95736 [https://github.com/python/cpython/issues/95736]: Fix **`unittest.IsolatedAsyncioTestCase`** to set event loop before calling setup functions. Patch by Kumar Aditya.
- gh-95865 [https://github.com/python/cpython/issues/95865]: Speed up **`urllib.parse.quote_from_bytes()`** by replacing a list comprehension with `map()`.
- gh-95861 [https://github.com/python/cpython/issues/95861]: Add support for computing Spearman’s correlation coefficient to the existing statistics.correlation() function.
- gh-95804 [https://github.com/python/cpython/issues/95804]: Fix `logging` shutdown handler so it respects `MemoryHandler.flushOnClose`.
- gh-95704 [https://github.com/python/cpython/issues/95704]: When a task catches **`asyncio.CancelledError`** and raises some other error, the other error should generally not silently be suppressed.
- gh-95149 [https://github.com/python/cpython/issues/95149]: The **`HTTPStatus`** enum offers a couple of properties to indicate the HTTP status category e.g. `HTTPStatus.OK.is_success`.

- gh-95609 [https://github.com/python/cpython/issues/95609]: Update bundled pip to 22.2.2.

- gh-95289 [https://github.com/python/cpython/issues/95289]: Fix **`asyncio.TaskGroup`** to propagate exception when **`asyncio.CancelledError`** was replaced with another exception by a context manager. Patch by Kumar Aditya and Guido van Rossum.

- gh-94909 [https://github.com/python/cpython/issues/94909]: Fix incorrect joining of relative Windows paths with drives in **`pathlib.PurePath`** initializer.

- gh-95385 [https://github.com/python/cpython/issues/95385]: Faster `json.dumps()` when sorting of keys is not requested (default).

- gh-83901 [https://github.com/python/cpython/issues/83901]: Improve **`Signature.bind`** error message for missing keyword-only arguments.

- gh-95339 [https://github.com/python/cpython/issues/95339]: Update bundled pip to 22.2.1.

- gh-95045 [https://github.com/python/cpython/issues/95045]: Fix GC crash when deallocating `_lsprof.Profiler` by untracking it before calling any callbacks. Patch by Kumar Aditya.

- gh-95231 [https://github.com/python/cpython/issues/95231]: Fail gracefully if **`EPERM`** or **`ENOSYS`** is raised when loading **`crypt`** methods. This may happen when trying to load `MD5` on a Linux kernel with FIPS enabled.

- gh-95097 [https://github.com/python/cpython/issues/95097]: Fix **`asyncio.run()`** for **`asyncio.Task`** implementations without **`uncancel()`** method. Patch by Kumar Aditya.

- gh-95087 [https://github.com/python/cpython/issues/95087]: Fix IndexError in parsing invalid date in the **`email`** module.

- gh-95199 [https://github.com/python/cpython/issues/95199]: Upgrade bundled setuptools to 63.2.0.

- gh-95194 [https://github.com/python/cpython/issues/95194]: Upgrade bundled pip to 22.2.

- gh-93899 [https://github.com/python/cpython/issues/93899]: Fix check for existence of **`os.EFD_CLOEXEC`**, **`os.EFD_NONBLOCK`** and **`os.EFD_SEMAPHORE`** flags on older kernel versions where these flags are not present. Patch by Kumar Aditya.

- gh-95166 [https://github.com/python/cpython/issues/95166]: Fix **`concurrent.futures.Executor.map()`** to cancel the currently waiting on future on an error - e.g. TimeoutError or KeyboardInterrupt.

- gh-95132 [https://github.com/python/cpython/issues/95132]: Fix a **`sqlite3`** regression where `*args` and `**kwds` were incorrectly relayed from **`connect()`** to the **`Connection`** factory. The regression was introduced in 3.11a1 with PR 24421 (gh-85128 [https://github.com/python/cpython/issues/85128]). Patch by Erlend E. Aasland.

- gh-93157 [https://github.com/python/cpython/issues/93157]: Fix **`fileinput`** module didn’t support `errors` option when `inplace` is true.

- gh-91212 [https://github.com/python/cpython/issues/91212]: Fixed flickering of the turtle window when the tracer is turned off. Patch by Shin-myoung-serp.

- gh-95077 [https://github.com/python/cpython/issues/95077]: Add deprecation warning for enum `member.member` access (e.g.

`Color.RED.BLUE`). Remove `EnumMeta.__getattr__`.

- gh-95109 [https://github.com/python/cpython/issues/95109]: Ensure that timeouts scheduled with **`asyncio.Timeout`** that have already expired are delivered promptly.
- gh-95105 [https://github.com/python/cpython/issues/95105]: **`wsgiref.types.InputStream.__iter__()`** should return `Iterator[bytes]`, not `Iterable[bytes]`. Patch by Shantanu Jain.
- gh-95066 [https://github.com/python/cpython/issues/95066]: Replaced assert with exception in **`ast.parse()`**, when `feature_version` has an invalid major version. Patch by Shantanu Jain.
- gh-77617 [https://github.com/python/cpython/issues/77617]: Add **`sqlite3`** command-line interface. Patch by Erlend Aasland.
- gh-95005 [https://github.com/python/cpython/issues/95005]: Replace _PyAccu with _PyUnicodeWriter in JSON encoder and StringIO and remove the _PyAccu implementation.
- gh-90085 [https://github.com/python/cpython/issues/90085]: Remove `-c/--clock` and `-t/--time` CLI options of **`timeit`**. The options had been deprecated since Python 3.3 and the functionality was removed in Python 3.7. Patch by Shantanu Jain.
- gh-94857 [https://github.com/python/cpython/issues/94857]: Fix refleak in `_io.TextIOWrapper.reconfigure`. Patch by Kumar Aditya.
- gh-94821 [https://github.com/python/cpython/issues/94821]: Fix binding of unix socket to empty address on Linux to use

an available address from the abstract namespace, instead of “0”.

- gh-94736 [https://github.com/python/cpython/issues/94736]: Fix crash when deallocating an instance of a subclass of `_multiprocessing.SemLock`. Patch by Kumar Aditya.

- gh-81620 [https://github.com/python/cpython/issues/81620]: Add random.binomialvariate().

- gh-74116 [https://github.com/python/cpython/issues/74116]: Allow **`asyncio.StreamWriter.drain()`** to be awaited concurrently by multiple tasks. Patch by Kumar Aditya.

- gh-87822 [https://github.com/python/cpython/issues/87822]: When called with `capture_locals=True`, the **`traceback`** module functions swallow exceptions raised from calls to `repr()` on local variables of frames. This is in order to prioritize the original exception over rendering errors. An indication of the failure is printed in place of the missing value. (Patch by Simon-Martin Schroeder).

- gh-88050 [https://github.com/python/cpython/issues/88050]: Fix **`asyncio`** subprocess transport to kill process cleanly when process is blocked and avoid `RuntimeError` when loop is closed. Patch by Kumar Aditya.

- gh-94637 [https://github.com/python/cpython/issues/94637]: **`SSLContext.set_default_verify_paths()`** now releases the GIL around `SSL_CTX_set_default_verify_paths` call. The function call performs I/O and CPU intensive work.

- gh-94309 [https://github.com/python/cpython/issues/94309]: Deprecate aliases **`typing.Hashable`** and **`typing.Sized`**

- gh-92546 [https://github.com/python/cpython/issues/92546]: An undocumented `python -m pprint` benchmark is moved into `pprint` suite of pyperformance. Patch by Oleg Iarygin.

- gh-94607 [https://github.com/python/cpython/issues/94607]: Fix subclassing complex generics with type variables in **`typing`**. Previously an error message saying `Some type variables ... are not listed in Generic[...]` was shown. **`typing`** no longer populates `__parameters__` with the `__parameters__` of a Python class.

- gh-94619 [https://github.com/python/cpython/issues/94619]: Remove the long-deprecated `module_repr()` from **`importlib`**.

- gh-93910 [https://github.com/python/cpython/issues/93910]: The ability to access the other values of an enum on an enum (e.g. `Color.RED.BLUE`) has been restored in order to fix a performance regression.

- gh-93896 [https://github.com/python/cpython/issues/93896]: Fix **`asyncio.run()`** and **`unittest.IsolatedAsyncioTestCase`** to always the set event loop as it was done in Python 3.10 and earlier. Patch by Kumar Aditya.

- gh-94343 [https://github.com/python/cpython/issues/94343]: Allow setting the attributes of `reprlib.Repr` during object initialization

- gh-94382 [https://github.com/python/cpython/issues/94382]: Port static types of `_multiprocessing` module to heap types. Patch by Kumar Aditya.

- gh-78724 [https://github.com/python/cpython/issues/78724]: Fix crash in **`struct.Struct`** when it was not completely initialized by initializing it in **`__new__()`**. Patch by Kumar Aditya.

- gh-94510 [https://github.com/python/cpython/issues/94510]: Re-entrant calls to `sys.setprofile()` and `sys.settrace()` now raise `RuntimeError`. Patch by Pablo Galindo.

- gh-92336 [https://github.com/python/cpython/issues/92336]: Fix bug where `linecache.getline()` fails on bad files with `UnicodeDecodeError` or `SyntaxError`. It now returns an empty string as per the documentation.

- gh-94398 [https://github.com/python/cpython/issues/94398]: Once a `asyncio.TaskGroup` has started shutting down (i.e., at least one task has failed and the task group has started cancelling the remaining tasks), it should not be possible to add new tasks to the task group.

- gh-94383 [https://github.com/python/cpython/issues/94383]: **`xml.etree`**: Remove the `ElementTree.Element.copy()` method of the pure Python implementation, deprecated in Python 3.10, use the `copy.copy()` function instead. The C implementation of **`xml.etree`** has no `copy()` method, only a `__copy__()` method. Patch by Victor Stinner.

- gh-94379 [https://github.com/python/cpython/issues/94379]: `zipimport`: Remove `find_loader()` and `find_module()` methods, deprecated in Python 3.10: use the `find_spec()` method instead. See **PEP 451** [https://peps.python.org/pep-0451/] for the rationale. Patch by Victor Stinner.

- gh-94352 [https://github.com/python/cpython/issues/94352]: `shlex.split()`: Passing `None` for *s* argument now raises an exception, rather than reading `sys.stdin`. The feature was deprecated in Python 3.9. Patch by Victor Stinner.

- gh-94318 [https://github.com/python/cpython/issues/94318]: Strip trailing spaces in `pydoc` text output.

- gh-89988 [https://github.com/python/cpython/issues/89988]: Fix memory leak in `pickle.Pickler` when looking up **`dispatch_table`**. Patch by Kumar Aditya.

- gh-90016 [https://github.com/python/cpython/issues/90016]: Deprecate `sqlite3` default adapters and converters. Patch by Erlend E. Aasland.

- gh-94254 [https://github.com/python/cpython/issues/94254]: Fixed types of `struct` module to be immutable. Patch by Kumar Aditya.

- gh-93259 [https://github.com/python/cpython/issues/93259]: Now raise `ValueError` when `None` or an empty string are passed to `Distribution.from_name` (and other callers).

- gh-74696 [https://github.com/python/cpython/issues/74696]: `shutil.make_archive()` now passes the *root_dir* argument to custom archivers which support it.

- gh-94216 [https://github.com/python/cpython/issues/94216]: The `dis` module now has the opcodes for pseudo instructions (those which are used by the compiler during code generation but then removed or replaced by real opcodes before the final bytecode is emitted).

- gh-93096 [https://github.com/python/cpython/issues/93096]: Removed undocumented `python -m codecs`. Use `python -m unittest test.test_codecs.EncodedFileTest` instead.

- gh-94207 [https://github.com/python/cpython/issues/94207]: Made **`_struct.Struct`** GC-tracked in order to fix a reference leak in the **`_struct`** module.

- gh-93096 [https://github.com/python/cpython/issues/93096]: Removed undocumented `-t` argument of `python -m base64`. Use `python -m unittest`

`test.test_base64.LegacyBase64TestCase.test_encodebytes` instead.

- gh-94226 [https://github.com/python/cpython/issues/94226]: Remove the **`locale.format()`** function, deprecated in Python 3.7: use **`locale.format_string()`** instead. Patch by Victor Stinner.

- gh-94199 [https://github.com/python/cpython/issues/94199]: Remove the **`ssl.match_hostname()`** function. The **`ssl.match_hostname()`** was deprecated in Python 3.7. OpenSSL performs hostname matching since Python 3.7, Python no longer uses the **`ssl.match_hostname()`** function. Patch by Victor Stinner.

- gh-94214 [https://github.com/python/cpython/issues/94214]: Document the `context` object used in the `venv.EnvBuilder` class, and add the new environment's library path to it.

- gh-94199 [https://github.com/python/cpython/issues/94199]: Remove the **`ssl.wrap_socket()`** function, deprecated in Python 3.7: instead, create a **`ssl.SSLContext`** object and call its **`ssl.SSLContext.wrap_socket`** method. Any package that still uses **`ssl.wrap_socket()`** is broken and insecure. The function neither sends a SNI TLS extension nor validates server hostname. Code is subject to **CWE 295** [https://cwe.mitre.org/data/definitions/295.html] Improper Certificate Validation. Patch by Victor Stinner.

- gh-94199 [https://github.com/python/cpython/issues/94199]: Remove the **`ssl.RAND_pseudo_bytes()`** function, deprecated in Python 3.6: use **`os.urandom()`** or **`ssl.RAND_bytes()`** instead. Patch by Victor Stinner.

- gh-94199 [https://github.com/python/cpython/issues/94199]: **`hashlib`**: Remove the pure Python implementation of **`hashlib.pbkdf2_hmac()`**, deprecated in Python 3.10. Python

3.10 and newer requires OpenSSL 1.1.1 (**PEP 644** [https://peps.python.org/pep-0644/]): this OpenSSL version provides a C implementation of **`pbkdf2_hmac()`** which is faster. Patch by Victor Stinner.

- gh-94196 [https://github.com/python/cpython/issues/94196]: **`gzip`**: Remove the `filename` attribute of **`gzip.GzipFile`**, deprecated since Python 2.6, use the **`name`** attribute instead. In write mode, the `filename` attribute added `'.gz'` file extension if it was not present. Patch by Victor Stinner.

- gh-94182 [https://github.com/python/cpython/issues/94182]: run the **`asyncio.PidfdChildWatcher`** on the running loop, this allows event loops to run subprocesses when there is no default event loop running on the main thread

- gh-94169 [https://github.com/python/cpython/issues/94169]: Remove `io.OpenWrapper` and `_pyio.OpenWrapper`, deprecated in Python 3.10: just use **`open()`** instead. The **`open()`** (**`io.open()`**) function is a built-in function. Since Python 3.10, **`_pyio.open()`** is also a static method. Patch by Victor Stinner.

- gh-91742 [https://github.com/python/cpython/issues/91742]: Fix **`pdb`** crash after jump caused by a null pointer dereference. Patch by Kumar Aditya.

- gh-94101 [https://github.com/python/cpython/issues/94101]: Manual instantiation of **`ssl.SSLSession`** objects is no longer allowed as it lead to misconfigured instances that crashed the interpreter when attributes where accessed on them.

- gh-84753 [https://github.com/python/cpython/issues/84753]: **`inspect.iscoroutinefunction()`**, **`inspect.isgeneratorfunction()`**, and

`inspect.isasyncgenfunction()` now properly return `True` for duck-typed function-like objects like instances of **`unittest.mock.AsyncMock`**.

This makes **`inspect.iscoroutinefunction()`** consistent with the behavior of **`asyncio.iscoroutinefunction()`**. Patch by Mehdi ABAAKOUK.

- gh-94028 [https://github.com/python/cpython/issues/94028]: Fix a regression in the **`sqlite3`** where statement objects were not properly cleared and reset after use in cursor iters. The regression was introduced by PR 27884 in Python 3.11a1. Patch by Erlend E. Aasland.

- gh-93973 [https://github.com/python/cpython/issues/93973]: Add keyword argument `all_errors` to `asyncio.create_connection` so that multiple connection errors can be raised as an `ExceptionGroup`.

- gh-93963 [https://github.com/python/cpython/issues/93963]: Officially deprecate from `importlib.abc` classes moved to `importlib.resources.abc`.

- gh-93858 [https://github.com/python/cpython/issues/93858]: Prevent error when activating venv in nested fish instances.

- gh-93820 [https://github.com/python/cpython/issues/93820]: Pickle **`enum.Flag`** by name.

- gh-93847 [https://github.com/python/cpython/issues/93847]: Fix repr of enum of generic aliases.

- gh-91404 [https://github.com/python/cpython/issues/91404]: Revert the **`re`** memory leak when a match is terminated by a signal or memory allocation failure as the implemented fix caused a major performance regression.

- gh-83499 [https://github.com/python/cpython/issues/83499]: Fix double closing of file description in **`tempfile`**.

- gh-93820 [https://github.com/python/cpython/issues/93820]: Fixed a regression when **`copy.copy()`**-ing **`enum.Flag`** with multiple flag members.

- gh-79512 [https://github.com/python/cpython/issues/79512]: Fixed names and `__module__` value of **`weakref`** classes **`ReferenceType`**, **`ProxyType`**, **`CallableProxyType`**. It makes them pickleable.

- gh-91389 [https://github.com/python/cpython/issues/91389]: Fix an issue where **`dis`** utilities could report missing or incorrect position information in the presence of `CACHE` entries.

- gh-93626 [https://github.com/python/cpython/issues/93626]: Set `__future__.annotations` to have a `None` mandatoryRelease to indicate that it is currently ‘TBD’.

- gh-90473 [https://github.com/python/cpython/issues/90473]: Emscripten and WASI have no home directory and cannot provide **PEP 370** [https://peps.python.org/pep-0370/] user site directory.

- gh-90494 [https://github.com/python/cpython/issues/90494]: **`copy.copy()`** and **`copy.deepcopy()`** now always raise a TypeError if `__reduce__()` returns a tuple with length 6 instead of silently ignore the 6th item or produce incorrect result.

- gh-90549 [https://github.com/python/cpython/issues/90549]: Fix a multiprocessing bug where a global named resource (such as a semaphore) could leak when a child process is spawned (as opposed to forked).

- gh-93521 [https://github.com/python/cpython/issues/93521]: Fixed a case where dataclasses would try to add `__weakref__` into the `__slots__` for a dataclass that specified `weakref_slot=True` when it was already defined in one of its bases. This resulted in a `TypeError` upon the new class being created.

- gh-79579 [https://github.com/python/cpython/issues/79579]: **`sqlite3`** now correctly detects DML queries with leading comments. Patch by Erlend E. Aasland.

- gh-93421 [https://github.com/python/cpython/issues/93421]: Update **`sqlite3.Cursor.rowcount`** when a DML statement has run to completion. This fixes the row count for SQL queries like `UPDATE ... RETURNING`. Patch by Erlend E. Aasland.

- gh-93475 [https://github.com/python/cpython/issues/93475]: Expose `FICLONE` and `FICLONERANGE` constants in **`fcntl`**. Patch by Illia Volochii.

- gh-93370 [https://github.com/python/cpython/issues/93370]: Deprecate **`version`** and **`version_info`**.

- gh-91810 [https://github.com/python/cpython/issues/91810]: Suppress writing an XML declaration in open files in `ElementTree.write()` with `encoding='unicode'` and `xml_declaration=None`.

- gh-91162 [https://github.com/python/cpython/issues/91162]: Support splitting of unpacked arbitrary-length tuple over `TypeVar` and `TypeVarTuple` parameters. For example:

    - `A[T, *Ts][*tuple[int, ...]]` -> `A[int, *tuple[int, ...]]`
    - `A[*Ts, T][*tuple[int, ...]]` -> `A[*tuple[int, ...], int]`

- gh-93353 [https://github.com/python/cpython/issues/93353]: Fix the **`importlib.resources.as_file()`** context manager to remove the temporary file if destroyed late during Python finalization: keep a local reference to the `os.remove()` function. Patch by Victor Stinner.

- gh-83658 [https://github.com/python/cpython/issues/83658]: Make **`multiprocessing.Pool`** raise an exception if `maxtasksperchild` is not `None` or a positive int.

- gh-93312 [https://github.com/python/cpython/issues/93312]: Add `os.PIDFD_NONBLOCK` flag to open a file descriptor for a process with `os.pidfd_open()` in non-blocking mode. Patch by Kumar Aditya.

- gh-88123 [https://github.com/python/cpython/issues/88123]: Implement `Enum.__contains__` that returns `True` or `False` to replace the deprecated behaviour that would sometimes raise a `TypeError`.

- gh-93297 [https://github.com/python/cpython/issues/93297]: Make asyncio task groups prevent child tasks from being GCed

- gh-85308 [https://github.com/python/cpython/issues/85308]: Changed `argparse.ArgumentParser` to use filesystem encoding and error handler instead of default text encoding to read arguments from file (e.g. `fromfile_prefix_chars` option). This change affects Windows; argument file should be encoded with UTF-8 instead of ANSI Codepage.

- gh-93156 [https://github.com/python/cpython/issues/93156]: Accessing the `pathlib.PurePath.parents` sequence of an absolute path using negative index values produced incorrect results.

- gh-93162 [https://github.com/python/cpython/issues/93162]: Add the ability for `logging.config.dictConfig()` to usefully configure `QueueHandler` and `QueueListener` as a pair, and add `logging.getHandlerByName()` and `logging.getHandlerNames()` APIs to allow access to handlers by name.

- gh-93243 [https://github.com/python/cpython/issues/93243]: The **`smtpd`** module was removed per the schedule in **PEP 594** [https://peps.python.org/pep-0594/].

- gh-92886 [https://github.com/python/cpython/issues/92886]: Replace `assert` statements with `raise AssertionError()` in **`BaseHandler`** so that the tested behaviour is maintained running with optimizations `(-O)`.

- gh-90155 [https://github.com/python/cpython/issues/90155]: Fix broken `asyncio.Semaphore` when acquire is cancelled.

- gh-90817 [https://github.com/python/cpython/issues/90817]: The **`locale.resetlocale()`** function is deprecated and will be removed in Python 3.13. Use `locale.setlocale(locale.LC_ALL, "")` instead. Patch by Victor Stinner.

- gh-91513 [https://github.com/python/cpython/issues/91513]: Added `taskName` attribute to `logging` module for use with `asyncio` tasks.

- gh-74696 [https://github.com/python/cpython/issues/74696]: `shutil.make_archive()` no longer temporarily changes the current working directory during creation of standard `.zip` or tar archives.

- gh-92728 [https://github.com/python/cpython/issues/92728]: The **`re.template()`** function and the corresponding **`re.TEMPLATE`** and **`re.T`** flags are restored after they were removed in 3.11.0b1, but they are now deprecated, so they might be removed from Python 3.13.

- gh-93033 [https://github.com/python/cpython/issues/93033]: Search in some strings (platform dependent i.e [U+0xFFFF, U+0x0100] on Windows or [U+0xFFFFFFFF, U+0x00010000] on Linux 64-bit) are now up to 10 times faster.

- gh-89973 [https://github.com/python/cpython/issues/89973]: Fix **`re.error`** raised in **`fnmatch`** if the pattern contains a character range with upper bound lower than lower bound (e.g. `[c-a]`). Now such ranges are interpreted as empty ranges.

- gh-93044 [https://github.com/python/cpython/issues/93044]: No longer convert the database argument of **`sqlite3.connect()`** to bytes before passing it to the factory.

- gh-93010 [https://github.com/python/cpython/issues/93010]: In a very special case, the email package tried to append the nonexistent `InvalidHeaderError` to the defect list. It should have been `InvalidHeaderDefect`.

- gh-92986 [https://github.com/python/cpython/issues/92986]: Fix **`ast.unparse()`** when `ImportFrom.level` is `None`

- gh-92932 [https://github.com/python/cpython/issues/92932]: Now **`dis()`** and **`get_instructions()`** handle operand values for instructions prefixed by `EXTENDED_ARG_QUICK`. Patch by Sam Gross and Donghee Na.

- gh-92675 [https://github.com/python/cpython/issues/92675]: Fix **`venv.ensure_directories()`** to accept **`pathlib.Path`** arguments in addition to **`str`** paths. Patch by David Foster.

- gh-87901 [https://github.com/python/cpython/issues/87901]: Removed the `encoding` argument from **`os.popen()`** that was added in 3.11b1.

- gh-91922 [https://github.com/python/cpython/issues/91922]: Fix function **`sqlite.connect()`** and the **`sqlite.Connection`** constructor on non-UTF-8 locales. Also, they now support bytes paths non-decodable with the current FS encoding.

- gh-92869 [https://github.com/python/cpython/issues/92869]: Added **`c_time_t`** to **`ctypes`**, which has the same size as the **`time_t`** type in C.

- gh-92839 [https://github.com/python/cpython/issues/92839]: Fixed crash resulting from calling bisect.insort() or bisect.insort_left() with the key argument not equal to `None`.

- gh-90473 [https://github.com/python/cpython/issues/90473]: **`subprocess`** now fails early on Emscripten and WASI platforms to work around missing **`os.pipe()`** on WASI.

- gh-89325 [https://github.com/python/cpython/issues/89325]: Removed many old deprecated **`unittest`** features: **`TestCase`** method aliases, undocumented and broken **`TestCase`** method `assertDictContainsSubset`, undocumented **`TestLoader.loadTestsFromModule`** parameter *use_load_tests*, and an underscored alias of the **`TextTestResult`** class.

- gh-92734 [https://github.com/python/cpython/issues/92734]: Allow multi-element reprs emitted by **`reprlib`** to be pretty-printed using configurable indentation.

- gh-92671 [https://github.com/python/cpython/issues/92671]: Fixed **`ast.unparse()`** for empty tuples in the assignment target context.

- gh-91581 [https://github.com/python/cpython/issues/91581]: **`utcfromtimestamp()`** no longer attempts to resolve `fold` in the pure Python implementation, since the fold is never 1 in UTC. In addition to being slightly faster in the common case, this also prevents some errors when the timestamp is close to **`datetime.min`**. Patch by Paul Ganssle.

- gh-86388 [https://github.com/python/cpython/issues/86388]: Removed randrange() functionality deprecated since Python 3.10. Formerly, randrange(10.0) losslessly converted to randrange(10). Now, it raises a TypeError. Also, the exception raised for non-integral values such as randrange(10.5) or randrange(‘10’) has been changed from ValueError to TypeError.

- gh-90385 [https://github.com/python/cpython/issues/90385]: Add **`pathlib.Path.walk()`** as an alternative to **`os.walk()`**.

- gh-92550 [https://github.com/python/cpython/issues/92550]: Fix **`pathlib.Path.rglob()`** for empty pattern.

- gh-92591 [https://github.com/python/cpython/issues/92591]: Allow **`logging`** filters to return a **`logging.LogRecord`** instance so that filters attached to **`logging.Handler`**s can enrich records without side effects on other handlers.

- gh-92445 [https://github.com/python/cpython/issues/92445]: Fix a bug in **`argparse`** where `nargs="*"` would raise an error instead of returning an empty list when 0 arguments were supplied if choice was also defined in `parser.add_argument`.

- gh-92547 [https://github.com/python/cpython/issues/92547]: Remove undocumented **`sqlite3`** features deprecated in Python 3.10:

  - `sqlite3.enable_shared_cache()`
  - `sqlite3.OptimizedUnicode`

  Patch by Erlend E. Aasland.

- gh-92530 [https://github.com/python/cpython/issues/92530]: Fix an issue that occurred after interrupting **`threading.Condition.notify()`**.

- gh-92531 [https://github.com/python/cpython/issues/92531]: The statistics.median_grouped() function now always return a float. Formerly, it did not convert the input type when for sequences of length one.

- gh-84131 [https://github.com/python/cpython/issues/84131]: The **`pathlib.Path`** deprecated method `link_to` has been removed. Use 3.10’s **`hardlink_to()`** method instead as its semantics are consistent with that of **`symlink_to()`**.

- gh-89336 [https://github.com/python/cpython/issues/89336]: Removed **`configparser`** module APIs: the `SafeConfigParser` class alias, the `ParsingError.filename` property and parameter, and the `ConfigParser.readfp` method, all of which were deprecated since Python 3.2.

- gh-92391 [https://github.com/python/cpython/issues/92391]: Add **`__class_getitem__()`** to **`csv.DictReader`** and **`csv.DictWriter`**, allowing them to be parameterized at runtime. Patch by Marc Mueller.

- gh-91968 [https://github.com/python/cpython/issues/91968]: Add `SO_RTABLE` and `SO_USER_COOKIE` constants to **`socket`**.

- gh-91810 [https://github.com/python/cpython/issues/91810]: **`ElementTree`** method **`write()`** and function **`tostring()`** now use the text file’s encoding (“UTF-8” if not available) instead of locale encoding in XML declaration when `encoding="unicode"` is specified.

- gh-81790 [https://github.com/python/cpython/issues/81790]: `os.path.splitdrive()` now understands DOS device paths with UNC links (beginning `\\?\UNC\`). Contributed by Barney Gale.

- gh-91760 [https://github.com/python/cpython/issues/91760]: Apply more strict rules for numerical group references and group names in regular expressions. Only sequence of ASCII digits is now accepted as a numerical reference. The group name in bytes patterns and replacement strings can now only contain ASCII letters and digits and underscore.

- gh-90622 [https://github.com/python/cpython/issues/90622]: Worker processes for `concurrent.futures.ProcessPoolExecutor` are no longer spawned on demand (a feature added in 3.9) when the multiprocessing context start method is `"fork"` as that can lead to deadlocks in the child processes due to a fork happening while threads are running.

- gh-91577 [https://github.com/python/cpython/issues/91577]: Move imports in **`SharedMemory`** methods to module level so that they can be executed late in python finalization.

- gh-91581 [https://github.com/python/cpython/issues/91581]: Remove an unhandled error case in the C implementation of calls to `datetime.fromtimestamp` with no time zone (i.e. getting a local time from an epoch timestamp). This should have no user-facing effect other than giving a possibly more accurate error message when called with timestamps that fall on 10000-01-01 in the local time. Patch by Paul Ganssle.

- gh-91539 [https://github.com/python/cpython/issues/91539]: Improve performance of `urllib.request.getproxies_environment` when there are many environment variables

- gh-91524 [https://github.com/python/cpython/issues/91524]: Speed up the regular expression substitution (functions `re.sub()` and `re.subn()` and corresponding `re.Pattern` methods) for replacement strings containing group references by 2–3 times.

- gh-91447 [https://github.com/python/cpython/issues/91447]: Fix findtext in the xml module to only give an empty string when the text attribute is set to `None`.

- gh-91456 [https://github.com/python/cpython/issues/91456]: Deprecate current default auto() behavior: In 3.13 the default will be for for auto() to always return the largest member value incremented by 1, and to raise if incompatible value types are used.

- bpo-47231 [https://bugs.python.org/issue?@action=redirect&bpo=47231]: Fixed an issue with inconsistent trailing slashes in tarfile longname directories.

- bpo-39064 [https://bugs.python.org/issue?@action=redirect&bpo=39064]: `zipfile.ZipFile` now raises `zipfile.BadZipFile` instead of `ValueError` when reading a corrupt zip file in which the central directory offset is negative.

- bpo-41287 [https://bugs.python.org/issue?@action=redirect&bpo=41287]: Fix handling of the `doc` argument in subclasses of `property()`.

- gh-90005 [https://github.com/python/cpython/issues/90005]: `ctypes` dependency `libffi` is now detected with `pkg-config`.

- bpo-32547 [https://bugs.python.org/issue?@action=redirect&bpo=32547]: The constructors for `DictWriter` and `DictReader` now coerce the `fieldnames` argument to a `list` if it is an iterator.

- bpo-35540 [https://bugs.python.org/issue?@action=redirect&bpo=35540]: Fix `dataclasses.asdict()` crash when `collections.defaultdict` is present in the attributes.

- bpo-47063 [https://bugs.python.org/issue?@action=redirect&bpo=47063]: Add an index_pages parameter to support using non-default index page names.

- bpo-47025 [https://bugs.python.org/issue?@action=redirect&bpo=47025]: Drop support for `bytes` on `sys.path`.

- bpo-46951 [https://bugs.python.org/issue?@action=redirect&bpo=46951]: Order the contents of zipapp archives, to make builds more reproducible.

- bpo-42777 [https://bugs.python.org/issue?@action=redirect&bpo=42777]: Implement `pathlib.Path.is_mount()` for Windows paths.

- bpo-46755 [https://bugs.python.org/issue?@action=redirect&bpo=46755]: In **`QueueHandler`**, clear `stack_info` from **`LogRecord`** to prevent stack trace from being written twice.

- bpo-45393 [https://bugs.python.org/issue?@action=redirect&bpo=45393]: Fix the formatting for `await x` and `not x` in the operator precedence table when using the `help()` system.

- bpo-46642 [https://bugs.python.org/issue?@action=redirect&bpo=46642]: Improve error message when trying to subclass an instance of **`typing.TypeVar`**, **`typing.ParamSpec`**, **`typing.TypeVarTuple`**, etc. Based on patch by Gregory Beauregard.

- bpo-46364 [https://bugs.python.org/issue?@action=redirect&bpo=46364]: Restrict use of sockets instead of pipes for stdin of subprocesses created by **`asyncio`** to AIX platform only.
- bpo-28249 [https://bugs.python.org/issue?@action=redirect&bpo=28249]: Set **`doctest.DocTest.lineno`** to `None` when an object does not have **`__doc__`**.
- bpo-46197 [https://bugs.python.org/issue?@action=redirect&bpo=46197]: Fix **`ensurepip`** environment isolation for subprocess running `pip`.
- bpo-45924 [https://bugs.python.org/issue?@action=redirect&bpo=45924]: Fix **`asyncio`** incorrect traceback when future’s exception is raised multiple times. Patch by Kumar Aditya.
- bpo-45046 [https://bugs.python.org/issue?@action=redirect&bpo=45046]: Add support of context managers in **`unittest`**: methods **`enterContext()`** and **`enterClassContext()`** of class **`TestCase`**, method **`enterAsyncContext()`** of class **`IsolatedAsyncioTestCase`** and function **`unittest.enterModuleContext()`**.
- bpo-44173 [https://bugs.python.org/issue?@action=redirect&bpo=44173]: Enable fast seeking of uncompressed unencrypted **`zipfile.ZipExtFile`**
- bpo-42627 [https://bugs.python.org/issue?@action=redirect&bpo=42627]: Fix incorrect parsing of Windows registry proxy settings

- bpo-42047 [https://bugs.python.org/issue?@action=redirect&bpo=42047]: Add `threading.get_native_id()` support for DragonFly BSD. Patch by David Carlier.

- bpo-14243 [https://bugs.python.org/issue?@action=redirect&bpo=14243]: The `tempfile.NamedTemporaryFile` function has a new optional parameter *delete_on_close*

- bpo-41246 [https://bugs.python.org/issue?@action=redirect&bpo=41246]: Give the same callback function for when the overlapped operation is done to the functions `recv`, `recv_into`, `recvfrom`, `sendto`, `send` and `sendfile` inside `IocpProactor`.

- bpo-39264 [https://bugs.python.org/issue?@action=redirect&bpo=39264]: Fixed **`collections.UserDict.get()`** to not call `__missing__()` when a value is not found. This matches the behavior of `dict`. Patch by Bar Harel.

- bpo-38693 [https://bugs.python.org/issue?@action=redirect&bpo=38693]: `importlib` now uses f-strings internally instead of `str.format`.

- bpo-38267 [https://bugs.python.org/issue?@action=redirect&bpo=38267]: Add *timeout* parameter to `asyncio.loop.shutdown_default_executor()`. The default value is `None`, which means the executor will be given an unlimited amount of time. When called from `asyncio.Runner` or `asyncio.run()`, the default timeout is 5 minutes.

- bpo-34828 [https://bugs.python.org/issue?@action=redirect&bpo=34828]: `sqlite3.Connection.iterdump()` now handles databases that use `AUTOINCREMENT` in one or more tables.

- bpo-32990 [https://bugs.python.org/issue?@action=redirect&bpo=32990]: Support reading wave files with the `WAVE_FORMAT_EXTENSIBLE` format in the **`wave`** module.

- bpo-26253 [https://bugs.python.org/issue?@action=redirect&bpo=26253]: Allow adjustable compression level for tarfile streams in **`tarfile.open()`**.

Documentation

- gh-85525 [https://github.com/python/cpython/issues/85525]: Remove extra row
- gh-86404 [https://github.com/python/cpython/issues/86404]: Deprecated tools `make suspicious` and `rstlint.py` are now removed. They have been replaced by sphinx-lint [https://pypi.org/project/sphinx-lint/].
- gh-97741 [https://github.com/python/cpython/issues/97741]: Fix `!` in c domain ref target syntax via a `conf.py` patch, so it works as intended to disable ref target resolution.
- gh-96432 [https://github.com/python/cpython/issues/96432]: Fraction literals now support whitespace around the forward slash, `Fraction('2 / 3')`.
- gh-96098 [https://github.com/python/cpython/issues/96098]: Improve discoverability of the higher level concurrent.futures module by providing clearer links from the lower level threading and multiprocessing modules.
- gh-95957 [https://github.com/python/cpython/issues/95957]: What’s New 3.11 now has instructions for how to provide compiler and linker flags for Tcl/Tk and OpenSSL on RHEL 7 and CentOS 7.
- gh-95588 [https://github.com/python/cpython/issues/95588]: Clarified the conflicting advice given in the **`ast`** documentation about **`ast.literal_eval()`** being “safe” for use on untrusted input while at the same time warning that it can crash the process. The latter statement is

true and is deemed unfixable without a large amount of work unsuitable for a bugfix. So we keep the warning and no longer claim that `literal_eval` is safe.

- gh-91207 [https://github.com/python/cpython/issues/91207]: Fix stylesheet not working in Windows CHM htmlhelp docs and add warning that they are deprecated. Contributed by C.A.M. Gerlach.
- gh-95454 [https://github.com/python/cpython/issues/95454]: Replaced incorrectly written true/false values in documentation. Patch by Robert O’Shea
- gh-95451 [https://github.com/python/cpython/issues/95451]: Update library documentation with availability information on WebAssembly platforms `wasm32-emscripten` and `wasm32-wasi`.
- gh-95415 [https://github.com/python/cpython/issues/95415]: Use consistent syntax for platform availability. The directive now supports a content body and emits a warning when it encounters an unknown platform.
- gh-94321 [https://github.com/python/cpython/issues/94321]: Document the **PEP 246** [https://peps.python.org/pep-0246/] style protocol type **`sqlite3.PrepareProtocol`**.
- gh-86128 [https://github.com/python/cpython/issues/86128]: Document a limitation in ThreadPoolExecutor where its exit handler is executed before any handlers in atexit.
- gh-61162 [https://github.com/python/cpython/issues/61162]: Clarify **`sqlite3`** behavior when How to use the connection context manager.
- gh-87260 [https://github.com/python/cpython/issues/87260]: Align **`sqlite3`** argument specs with the actual implementation.
- gh-86986 [https://github.com/python/cpython/issues/86986]: The minimum Sphinx version required to build the documentation is now 3.2.
- gh-88831 [https://github.com/python/cpython/issues/88831]: Augmented documentation of asyncio.create_task(). Clarified the need to keep strong references to tasks and added a code snippet detailing how to do this.

- gh-86438 [https://github.com/python/cpython/issues/86438]: Clarify that **`-W`** and **`PYTHONWARNINGS`** are matched literally and case-insensitively, rather than as regular expressions, in **`warnings`**.
- gh-93031 [https://github.com/python/cpython/issues/93031]: Update tutorial introduction output to use 3.10+ SyntaxError invalid range.
- gh-92240 [https://github.com/python/cpython/issues/92240]: Added release dates for "What's New in Python 3.X" for 3.0, 3.1, 3.2, 3.8 and 3.10
- bpo-47161 [https://bugs.python.org/issue?@action=redirect&bpo=47161]: Document that **`pathlib.PurePath`** does not collapse initial double slashes because they denote UNC paths.
- bpo-40838 [https://bugs.python.org/issue?@action=redirect&bpo=40838]: Document that **`inspect.getdoc()`**, **`inspect.getmodule()`**, and **`inspect.getsourcefile()`** might return `None`.
- bpo-43689 [https://bugs.python.org/issue?@action=redirect&bpo=43689]: The `Differ` documentation now also mentions other whitespace characters, which make it harder to understand the diff output.
- bpo-38056 [https://bugs.python.org/issue?@action=redirect&bpo=38056]: Overhaul the Error Handlers documentation in **`codecs`**.
- bpo-13553 [https://bugs.python.org/issue?@action=redirect&bpo=13553]: Document tkinter.Tk args.

Tests

- gh-95027 [https://github.com/python/cpython/issues/95027]: On Windows, when the Python test suite is run with the `-jN` option, the ANSI code page is now used as the encoding for the stdout temporary file, rather than using UTF-8 which can lead to decoding errors. Patch by Victor Stinner.

- gh-96624 [https://github.com/python/cpython/issues/96624]: Fixed the failure of repeated runs of `test.test_unittest` caused by side effects in `test_dotted_but_module_not_loaded`.
- gh-95243 [https://github.com/python/cpython/issues/95243]: Mitigate the inherent race condition from using find_unused_port() in testSockName() by trying to find an unused port a few times before failing. Patch by Ross Burton.
- gh-95573 [https://github.com/python/cpython/issues/95573]: Lib/test/test_asyncio/test_ssl.py [https://github.com/python/cpython/tree/3.14/Lib/test/test_asyncio/test_ssl.py] exposed a bug in the macOS kernel where intense concurrent load on non-blocking sockets occasionally causes **`errno.ENOBUFS`** (“No buffer space available”) to be emitted. FB11063974 filed with Apple, in the mean time as a workaround buffer size used in tests on macOS is decreased to avoid intermittent failures. Patch by Fantix King.
- gh-95280 [https://github.com/python/cpython/issues/95280]: Fix problem with `test_ssl` `test_get_ciphers` on systems that require perfect forward secrecy (PFS) ciphers.
- gh-95212 [https://github.com/python/cpython/issues/95212]: Make multiprocessing test case `test_shared_memory_recreate` parallel-safe.
- gh-95218 [https://github.com/python/cpython/issues/95218]: Move tests for importlib.resources into test_importlib.resources.
- gh-93963 [https://github.com/python/cpython/issues/93963]: Updated tests to use preferred location for `importlib.resources` ABCs.
- gh-94675 [https://github.com/python/cpython/issues/94675]: Add a regression test for **`re`** exponentional slowdown when using rjsmin.
- gh-91330 [https://github.com/python/cpython/issues/91330]: Added more tests for **`dataclasses`** to cover behavior with data descriptor-based fields.

- gh-94208 [https://github.com/python/cpython/issues/94208]: `test_ssl` is now checking for supported TLS version and protocols in more tests.
- gh-94315 [https://github.com/python/cpython/issues/94315]: Tests now check for DAC override capability instead of relying on **`os.geteuid()`**.
- gh-54781 [https://github.com/python/cpython/issues/54781]: Rename test_tk to test_tkinter, and rename test_ttk_guionly to test_ttk. Patch by Victor Stinner.
- gh-93839 [https://github.com/python/cpython/issues/93839]: Move `Lib/ctypes/test/` to `Lib/test/test_ctypes/`. Patch by Victor Stinner.
- gh-93951 [https://github.com/python/cpython/issues/93951]: In test_bdb.StateTestCase.test_skip, avoid including auxiliary importers.
- gh-93957 [https://github.com/python/cpython/issues/93957]: Provide nicer error reporting from subprocesses in test_venv.EnsurePipTest.test_with_pip.
- gh-93884 [https://github.com/python/cpython/issues/93884]: Add test cases for **`PyNumber_ToBase()`** that take a large number or a non-int object as parameter.
- gh-93852 [https://github.com/python/cpython/issues/93852]: test_asyncio, test_logging, test_socket and test_socketserver now create AF_UNIX domains in the current directory to no longer fail with `OSError("AF_UNIX path too long")` if the temporary directory (the **`TMPDIR`** environment variable) is too long. Patch by Victor Stinner.
- gh-93353 [https://github.com/python/cpython/issues/93353]: regrtest now checks if a test leaks temporary files or directories if run with -jN option. Patch by Victor Stinner.
- gh-84461 [https://github.com/python/cpython/issues/84461]: `run_tests.py` now handles cross compiling env vars correctly and pass `HOSTRUNNER` to regression tests.
- gh-93616 [https://github.com/python/cpython/issues/93616]: `test_modulefinder` now creates a temporary directory in

`ModuleFinderTest.setUp()` instead of module scope.

- gh-93575 [https://github.com/python/cpython/issues/93575]: Fix issue with test_unicode test_raiseMemError. The test case now use `test.support.calcobjsize` to calculate size of PyUnicode structs. **`sys.getsizeof()`** may return different size when string has UTF-8 memory.
- gh-90473 [https://github.com/python/cpython/issues/90473]: WASI does not have a `chmod(2)` syscall. **`os.chmod()`** is now a dummy function on WASI. Skip all tests that depend on working **`os.chmod()`**.
- gh-90473 [https://github.com/python/cpython/issues/90473]: Skip tests on WASI that require symlinks with absolute paths.
- gh-57539 [https://github.com/python/cpython/issues/57539]: Increase calendar test coverage for **`calendar.LocaleTextCalendar.formatweekday()`**.
- gh-90473 [https://github.com/python/cpython/issues/90473]: Skip symlink tests on WASI. wasmtime uses `openat2(2)` with `RESOLVE_BENEATH` flag, which prevents symlinks with absolute paths.
- gh-89858 [https://github.com/python/cpython/issues/89858]: Fix `test_embed` for out-of-tree builds. Patch by Kumar Aditya.
- gh-92886 [https://github.com/python/cpython/issues/92886]: Fixing tests that fail when running with optimizations (`-O`) in `test_imaplib.py`.
- gh-92886 [https://github.com/python/cpython/issues/92886]: Fixing tests that fail when running with optimizations (`-O`) in `test_zipimport.py`
- gh-92886 [https://github.com/python/cpython/issues/92886]: Fixing tests that fail when running with optimizations (`-O`) in `test_py_compile.py`
- gh-92886 [https://github.com/python/cpython/issues/92886]: Fixing tests that fail when running with optimizations (`-O`) in `test_sys_settrace.py`.
- gh-92886 [https://github.com/python/cpython/issues/92886]: Fixing tests that fail when running with optimizations (`-O`) in `_test_multiprocessing.py`

- gh-92670 [https://github.com/python/cpython/issues/92670]: Skip `test_shutil.TestCopy.test_copyfile_nonexistent_dir` test on AIX as the test uses a trailing slash to force the OS consider the path as a directory, but on AIX the trailing slash has no effect and is considered as a file.
- gh-92514 [https://github.com/python/cpython/issues/92514]: Remove unused `test.support.BasicTestRunner`. Patch by Jelle Zijlstra.
- bpo-47016 [https://bugs.python.org/issue?@action=redirect&bpo=47016]: Create a GitHub Actions workflow for verifying bundled pip and setuptools. Patch by Illia Volochii and Adam Turner.

Build

- gh-96761 [https://github.com/python/cpython/issues/96761]: Fix the build process of clang compiler for **_bootstrap_python** if LTO optimization is applied. Patch by Matthias Görgens and Donghee Na.
- gh-96883 [https://github.com/python/cpython/issues/96883]: `wasm32-emscripten` builds for browsers now include **`concurrent.futures`** for **`asyncio`** and **`unittest.mock`**.
- gh-85936 [https://github.com/python/cpython/issues/85936]: CPython now uses the ThinLTO option as the default policy if the Clang compiler accepts the flag. Patch by Donghee Na.
- gh-96729 [https://github.com/python/cpython/issues/96729]: Ensure that Windows releases built with `Tools\msi\buildrelease.bat` are upgradable to and from official Python releases.
- gh-96269 [https://github.com/python/cpython/issues/96269]: Shared module targets now depend on new `MODULE_DEPS` variable, which includes `EXPORTSYMS`. This fixes a build order issue on unsupported AIX platform.
- gh-84461 [https://github.com/python/cpython/issues/84461]: `wasm32-emscripten` platform no longer builds **`resource`** module,

`getresuid()`, **`getresgid()`**, and their setters. The APIs are stubs and not functional.

- gh-95973 [https://github.com/python/cpython/issues/95973]: Add a new `--with-dsymutil` configure option to link debug information in macOS. Patch by Pablo Galindo.
- gh-90536 [https://github.com/python/cpython/issues/90536]: Use the BOLT post-link optimizer to improve performance, particularly on medium-to-large applications.
- gh-93744 [https://github.com/python/cpython/issues/93744]: Remove the `configure --with-cxx-main` build option: it didn’t work for many years. Remove the `MAINCC` variable from `configure` and `Makefile`. Patch by Victor Stinner.
- gh-94801 [https://github.com/python/cpython/issues/94801]: Fix a regression in `configure` script that caused some header checks to ignore custom `CPPFLAGS`. The regression was introduced in gh-94802 [https://github.com/python/cpython/issues/94802].
- gh-95145 [https://github.com/python/cpython/issues/95145]: wasm32-wasi builds no longer depend on WASIX’s pthread stubs. Python now has its own stubbed pthread API.
- gh-95174 [https://github.com/python/cpython/issues/95174]: Python now detects missing `dup` function in WASI and works around some missing **`errno`**, **`select`**, and **`socket`** constants.
- gh-95174 [https://github.com/python/cpython/issues/95174]: Python now skips missing **`socket`** functions and methods on WASI. WASI can only create sockets from existing fd / accept and has no netdb.
- gh-95085 [https://github.com/python/cpython/issues/95085]: Platforms `wasm32-unknown-emscripten` and `wasm32-unknown-wasi` have been promoted to **PEP 11** [https://peps.python.org/pep-0011/] tier 3 platform support.
- gh-94847 [https://github.com/python/cpython/issues/94847]: Fixed `_decimal` module build issue on GCC when compiling with LTO and pydebug. Debug builds no longer force inlining of functions.

- gh-94841 [https://github.com/python/cpython/issues/94841]: Fix the possible performance regression of **`PyObject_Free()`** compiled with MSVC version 1932.
- gh-94801 [https://github.com/python/cpython/issues/94801]: `configure` now uses custom flags like `ZLIB_CFLAGS` and `ZLIB_LIBS` when searching for headers and libraries.
- gh-94773 [https://github.com/python/cpython/issues/94773]: `deepfreeze.py` now supports code object with frozensets that contain incompatible, unsortable types.
- gh-94682 [https://github.com/python/cpython/issues/94682]: Build and test with OpenSSL 1.1.1q
- gh-90005 [https://github.com/python/cpython/issues/90005]: Dependencies of **`readline`** and **`curses`** module are now detected in `configure` script with `pkg-config`. Only `ncurses` / `ncursesw` are detected automatically. The old `curses` library is not configured automatically. Workaround for missing `termcap` or `tinfo` library has been removed.
- gh-90005 [https://github.com/python/cpython/issues/90005]: Fix building `_ctypes` extension without `pkg-config`.
- gh-90005 [https://github.com/python/cpython/issues/90005]: `_dbm` module dependencies are now detected by configure.
- gh-94404 [https://github.com/python/cpython/issues/94404]: `makesetup` now works around an issue with sed on macOS and uses correct CFLAGS for object files that end up in a shared extension. Module CFLAGS are used before PY_STDMODULE_CFLAGS to avoid clashes with system headers.
- gh-93939 [https://github.com/python/cpython/issues/93939]: C extension modules are now built by `configure` and `make` instead of `distutils` and `setup.py`.
- gh-93939 [https://github.com/python/cpython/issues/93939]: The `2to3`, `idle`, and `pydoc` scripts are now generated and installed by `Makefile` instead of `setup.py`.
- gh-94280 [https://github.com/python/cpython/issues/94280]: Updated pegen regeneration script on Windows to find and use Python 3.9 or higher. Prior to this, pegen

regeneration already required 3.9 or higher, but the script may have used lower versions of Python.

- gh-93584 [https://github.com/python/cpython/issues/93584]: Address race condition in `Makefile` when installing a PGO build. All `test` and `install` targets now depend on `all` target.
- gh-93491 [https://github.com/python/cpython/issues/93491]: `configure` now detects and reports **PEP 11** [https://peps.python.org/pep-0011/] support tiers.
- gh-69093 [https://github.com/python/cpython/issues/69093]: Fix `Modules/Setup.stdlib.in` rule for `_sqlite3` extension.
- gh-93207 [https://github.com/python/cpython/issues/93207]: `va_start()` with two parameters, like `va_start(args, format)`, is now required to build Python. `va_start()` is no longer called with a single parameter. Patch by Kumar Aditya.
- gh-93202 [https://github.com/python/cpython/issues/93202]: Python now always use the `%zu` and `%zd` printf formats to format a **`size_t`** or `Py_ssize_t` number. Building Python 3.12 requires a C11 compiler, so these printf formats are now always supported. Patch by Victor Stinner.
- gh-90473 [https://github.com/python/cpython/issues/90473]: Disable pymalloc and increase stack size on `wasm32-wasi`.
- bpo-34449 [https://bugs.python.org/issue?@action=redirect&bpo=34449]: Drop invalid compiler switch `-fPIC` for HP aCC on HP-UX. Patch by Michael Osipov.

Windows

- gh-98360 [https://github.com/python/cpython/issues/98360]: Fixes **`multiprocessing`** spawning child processes on Windows from a virtual environment to ensure that child processes that also use **`multiprocessing`** to spawn more children will recognize that they are in a virtual environment.

- gh-98414 [https://github.com/python/cpython/issues/98414]: Fix `py.exe` launcher handling of `-V:<company>/` option when default preferences have been set in environment variables or configuration files.
- gh-97728 [https://github.com/python/cpython/issues/97728]: Fix possible crashes caused by the use of uninitialized variables when pass invalid arguments in **`os.system()`** on Windows and in Windows-specific modules (like `winreg`).
- gh-90989 [https://github.com/python/cpython/issues/90989]: Made Python install manager install per-user by default (unless an all users install already exists), and clarify some text in the installer.
- gh-97649 [https://github.com/python/cpython/issues/97649]: The `Tools` directory is no longer installed on Windows
- gh-96965 [https://github.com/python/cpython/issues/96965]: Update libffi to 3.4.3
- gh-96577 [https://github.com/python/cpython/issues/96577]: Fixes a potential buffer overrun in **`msilib`**.
- gh-96559 [https://github.com/python/cpython/issues/96559]: Fixes the Windows launcher not using the compatible interpretation of default tags found in configuration files when no tag was passed to the command.
- gh-94781 [https://github.com/python/cpython/issues/94781]: Fix `pcbuild.proj` to clean previous instances of output files in `Python\deepfreeze` and `Python\frozen_modules` directories on Windows. Patch by Charlie Zhao.
- gh-89545 [https://github.com/python/cpython/issues/89545]: Updates **`platform`** code getting the Windows version to use native Windows Management Instrumentation (WMI) queries to determine OS version, type, and architecture.
- gh-95733 [https://github.com/python/cpython/issues/95733]: Make certain requirements of the Windows Store package optional to allow installing on earlier updates of Windows.
- gh-95656 [https://github.com/python/cpython/issues/95656]: Enable the **`enable_load_extension()`** **`sqlite3`** API.

- gh-95587 [https://github.com/python/cpython/issues/95587]: Fixes some issues where the Windows installer would incorrectly detect certain features of an existing install when upgrading.
- gh-94399 [https://github.com/python/cpython/issues/94399]: Restores the behaviour of Python install manager for `/usr/bin/env` shebang lines, which will now search **`PATH`** for an executable matching the given command. If none is found, the usual search process is used.
- gh-95445 [https://github.com/python/cpython/issues/95445]: Fixes the unsuccessful removal of the HTML document directory when uninstalling with Windows msi.
- gh-95359 [https://github.com/python/cpython/issues/95359]: Fix Python install manager handling of `py.ini` commands (it was incorrectly expecting a `py_` prefix on keys) and crashes when reading per-user configuration file.
- gh-95285 [https://github.com/python/cpython/issues/95285]: Fix Python install manager handling of command lines where it is only passed a short executable name.
- gh-90844 [https://github.com/python/cpython/issues/90844]: Allow virtual environments to correctly launch when they have spaces in the path.
- gh-94772 [https://github.com/python/cpython/issues/94772]: Fix incorrect handling of shebang lines in py.exe launcher
- gh-94018 [https://github.com/python/cpython/issues/94018]: **`zipfile`** will now remove trailing spaces from path components when extracting files on Windows.
- gh-93824 [https://github.com/python/cpython/issues/93824]: Drag and drop of files onto Python files in Windows Explorer has been enabled for Windows ARM64.
- gh-43414 [https://github.com/python/cpython/issues/43414]: **`os.get_terminal_size()`** now attempts to read the size from any provided handle, rather than only supporting file descriptors 0, 1 and 2.
- gh-92817 [https://github.com/python/cpython/issues/92817]: Ensures that `py.exe` will prefer an active virtual

environment over default tags specified with environment variables or through a `py.ini` file.

- gh-92984 [https://github.com/python/cpython/issues/92984]: Explicitly disable incremental linking for non-Debug builds
- gh-92841 [https://github.com/python/cpython/issues/92841]: **`asyncio`** no longer throws `RuntimeError: Event loop is closed` on interpreter exit after asynchronous socket activity. Patch by Oleg Iarygin.
- bpo-46907 [https://bugs.python.org/issue?@action=redirect&bpo=46907]: Update Windows installer to use SQLite 3.38.4.
- gh-91061 [https://github.com/python/cpython/issues/91061]: Accept os.PathLike for the argument to winsound.PlaySound
- bpo-42658 [https://bugs.python.org/issue?@action=redirect&bpo=42658]: Support native Windows case-insensitive path comparisons by using `LCMapStringEx` instead of **`str.lower()`** in **`ntpath.normcase()`**. Add `LCMapStringEx` to the **`_winapi`** module.
- bpo-38704 [https://bugs.python.org/issue?@action=redirect&bpo=38704]: Prevent installation on unsupported Windows versions.

macOS

- gh-97897 [https://github.com/python/cpython/issues/97897]: The macOS 13 SDK includes support for the `mkfifoat` and `mknodat` system calls. Using the `dir_fd` option with either **`os.mkfifo()`** or **`os.mknod()`** could result in a segfault if cpython is built with the macOS 13 SDK but run on an earlier version of macOS. Prevent this by adding runtime support for detection of these system calls (“weaklinking”) as is done for other newer syscalls on macOS.

IDLE

- gh-97527 [https://github.com/python/cpython/issues/97527]: Fix a bug in the previous bugfix that caused IDLE to not start when run with 3.10.8, 3.12.0a1, and at least Microsoft Python 3.10.2288.0 installed without the Lib/test package. 3.11.0 was never affected.
- gh-65802 [https://github.com/python/cpython/issues/65802]: Document handling of extensions in Save As dialogs.
- gh-95191 [https://github.com/python/cpython/issues/95191]: Include prompts when saving Shell (interactive input and output).
- gh-95511 [https://github.com/python/cpython/issues/95511]: Fix the Shell context menu copy-with-prompts bug of copying an extra line when one selects whole lines.
- gh-95471 [https://github.com/python/cpython/issues/95471]: In the Edit menu, move `Select All` and add a new separator.
- gh-95411 [https://github.com/python/cpython/issues/95411]: Enable using IDLE's module browser with .pyw files.
- gh-89610 [https://github.com/python/cpython/issues/89610]: Add .pyi as a recognized extension for IDLE on macOS. This allows opening stub files by double clicking on them in the Finder.

Tools/Demos

- gh-68686 [https://github.com/python/cpython/issues/68686]: Remove ptags and eptags scripts.
- gh-97681 [https://github.com/python/cpython/issues/97681]: Remove the `Tools/demo/` directory which contained old demo scripts. A copy can be found in the old-demos project [https://github.com/gvanrossum/old-demos]. Patch by Victor Stinner.
- gh-97669 [https://github.com/python/cpython/issues/97669]: Remove outdated example scripts of the `Tools/scripts/`

directory. A copy can be found in the old-demos project [https://github.com/gvanrossum/old-demos]. Patch by Victor Stinner.

- gh-95853 [https://github.com/python/cpython/issues/95853]: The `wasm_build.py` script now pre-builds Emscripten ports, checks for broken EMSDK versions, and warns about pkg-config env vars.
- gh-95853 [https://github.com/python/cpython/issues/95853]: The new tool `Tools/wasm/wasm_builder.py` automates configure, compile, and test steps for building CPython on WebAssembly platforms.
- gh-95731 [https://github.com/python/cpython/issues/95731]: Fix handling of module docstrings in `Tools/i18n/pygettext.py`.
- gh-93939 [https://github.com/python/cpython/issues/93939]: Add script `Tools/scripts/check_modules.py` to check and validate builtin and shared extension modules. The script also handles `Modules/Setup` and will eventually replace `setup.py`.
- gh-94538 [https://github.com/python/cpython/issues/94538]: Fix Argument Clinic output to custom file destinations. Patch by Erlend E. Aasland.
- gh-94430 [https://github.com/python/cpython/issues/94430]: Allow parameters named `module` and `self` with custom C names in Argument Clinic. Patch by Erlend E. Aasland
- gh-86087 [https://github.com/python/cpython/issues/86087]: The `Tools/scripts/parseentities.py` script used to parse HTML4 entities has been removed.

C API

- gh-98393 [https://github.com/python/cpython/issues/98393]: The **`PyUnicode_FSDecoder()`** function no longer accepts bytes-like paths, like **`bytearray`** and **`memoryview`** types: only the exact **`bytes`** type is accepted for bytes strings. Patch by Victor Stinner.

- gh-91051 [https://github.com/python/cpython/issues/91051]: Add `PyType_Watch()` and related APIs to allow callbacks on `PyType_Modified()`.

- gh-95756 [https://github.com/python/cpython/issues/95756]: Lazily create and cache `co_` attributes for better performance for code getters.

- gh-96512 [https://github.com/python/cpython/issues/96512]: Configuration for the integer string conversion length limitation now lives in the PyConfig C API struct.

- gh-95589 [https://github.com/python/cpython/issues/95589]: Extensions classes that set `tp_dictoffset` and `tp_weaklistoffset` lose the support for multiple inheritance, but are now safe. Extension classes should use `Py_TPFLAGS_MANAGED_DICT` and `Py_TPFLAGS_MANAGED_WEAKREF` instead.

- gh-95781 [https://github.com/python/cpython/issues/95781]: An unrecognized format character in `PyUnicode_FromFormat()` and `PyUnicode_FromFormatV()` now sets a `SystemError`. In previous versions it caused all the rest of the format string to be copied as-is to the result string, and any extra arguments discarded.

- gh-92678 [https://github.com/python/cpython/issues/92678]: Restore the 3.10 behavior for multiple inheritance of C extension classes that store their dictionary at the end of the struct.

- gh-92678 [https://github.com/python/cpython/issues/92678]: Support C extensions using managed dictionaries by setting the `Py_TPFLAGS_MANAGED_DICT` flag.

- gh-93274 [https://github.com/python/cpython/issues/93274]: API for implementing vectorcall (`Py_TPFLAGS_HAVE_VECTORCALL`, `PyVectorcall_NARGS()` and `PyVectorcall_Call()`) was added to the limited API and stable ABI.

- gh-95504 [https://github.com/python/cpython/issues/95504]: Fix sign placement when specifying width or precision in `PyUnicode_FromFormat()` and `PyUnicode_FromFormatV()`. Patch by Philip Georgi.

- gh-93012 [https://github.com/python/cpython/issues/93012]: The `Py_TPFLAGS_HAVE_VECTORCALL` flag is now removed from a class when the class's `__call__()` method is reassigned. This makes vectorcall safe to use with mutable types (i.e. heap types without the **`immutable`** flag). Mutable types that do not override `tp_call` now inherit the `Py_TPFLAGS_HAVE_VECTORCALL` flag.

- gh-95388 [https://github.com/python/cpython/issues/95388]: Creating `immutable types` with mutable bases is deprecated and is planned to be disabled in Python 3.14.

- gh-92678 [https://github.com/python/cpython/issues/92678]: Adds unstable C-API functions `_PyObject_VisitManagedDict` and `_PyObject_ClearManagedDict` to allow C extensions to allow the VM to manage their object's dictionaries.

- gh-94936 [https://github.com/python/cpython/issues/94936]: Added `PyCode_GetVarnames()`, `PyCode_GetCellvars()` and `PyCode_GetFreevars()` for accessing `co_varnames`, `co_cellvars` and `co_freevars` respectively via the C API.

- gh-94930 [https://github.com/python/cpython/issues/94930]: Fix `SystemError` raised when `PyArg_ParseTupleAndKeywords()` is used with `#` in `(...)` but without `PY_SSIZE_T_CLEAN` defined.

- gh-94731 [https://github.com/python/cpython/issues/94731]: Python again uses C-style casts for most casting operations when compiled with C++. This may trigger compiler warnings, if they are enabled with e.g. `-Wold-style-cast` or `-Wzero-as-null-pointer-constant` options for `g++`.

- gh-93937 [https://github.com/python/cpython/issues/93937]: The following frame functions and type are now directly available with `#include <Python.h>`, it's no longer needed to add `#include <frameobject.h>`:
  - **`PyFrame_Check()`**
  - **`PyFrame_GetBack()`**
  - **`PyFrame_GetBuiltins()`**
  - **`PyFrame_GetGenerator()`**
  - **`PyFrame_GetGlobals()`**
  - **`PyFrame_GetLasti()`**
  - **`PyFrame_GetLocals()`**
  - **`PyFrame_Type`**

  Patch by Victor Stinner.

- gh-91321 [https://github.com/python/cpython/issues/91321]: Fix the compatibility of the Python C API with C++ older than C++11. Patch by Victor Stinner.

- gh-91731 [https://github.com/python/cpython/issues/91731]: Avoid defining the `static_assert` when compiling with C++ 11, where this is a keyword and redefining it can lead to undefined behavior. Patch by Pablo Galindo

- gh-89546 [https://github.com/python/cpython/issues/89546]: **`PyType_FromMetaclass()`** (and other `PyType_From*` functions) now check that offsets and the base class's **`tp_basicsize`** fit in the new class's `tp_basicsize`.

- gh-93503 [https://github.com/python/cpython/issues/93503]: Add two new public functions to the public C-API, **`PyEval_SetProfileAllThreads()`** and

`PyEval_SetTraceAllThreads()`, that allow to set tracing and profiling functions in all running threads in addition to the calling one. Also, two analogous functions to the `threading` module (`threading.setprofile_all_threads()` and `threading.settrace_all_threads()`) that allow to do the same from Python. Patch by Pablo Galindo

- gh-93442 [https://github.com/python/cpython/issues/93442]: Add C++ overloads for _Py_CAST_impl() to handle 0/NULL. This will allow C++ extensions that pass 0 or NULL to macros using _Py_CAST() to continue to compile.

- gh-93466 [https://github.com/python/cpython/issues/93466]: Slot IDs in PyType_Spec may not be repeated. The documentation was updated to mention this. For some cases of repeated slots, PyType_FromSpec and related functions will now raise an exception.

- gh-92898 [https://github.com/python/cpython/issues/92898]: Fix C++ compiler warnings when casting function arguments to `PyObject*`. Patch by Serge Guelton.

- gh-93103 [https://github.com/python/cpython/issues/93103]: Deprecate global configuration variables, like `Py_IgnoreEnvironmentFlag`, in the documentation: the `Py_InitializeFromConfig()` API should be instead. Patch by Victor Stinner.

- gh-77782 [https://github.com/python/cpython/issues/77782]: Deprecate global configuration variable like `Py_IgnoreEnvironmentFlag`: the `Py_InitializeFromConfig()` API should be instead. Patch by Victor Stinner.

- gh-92913 [https://github.com/python/cpython/issues/92913]: Ensures changes to `PyConfig.module_search_paths` are ignored unless `PyConfig.module_search_paths_set` is set

- gh-92781 [https://github.com/python/cpython/issues/92781]: Avoid mixing declarations and code in the C API to fix the compiler warning: “ISO C90 forbids mixed declarations and code” [-Werror=declaration-after-statement]. Patch by Victor Stinner.
- gh-92651 [https://github.com/python/cpython/issues/92651]: Remove the `token.h` header file. There was never any public tokenizer C API. The `token.h` header file was only designed to be used by Python internals. Patch by Victor Stinner.
- gh-92536 [https://github.com/python/cpython/issues/92536]: Remove legacy Unicode APIs based on `Py_UNICODE*`.
- gh-85858 [https://github.com/python/cpython/issues/85858]: Remove the `PyUnicode_InternImmortal()` function and the `SSTATE_INTERNED_IMMORTAL` macro. Patch by Victor Stinner.
- gh-92193 [https://github.com/python/cpython/issues/92193]: Add new function **`PyFunction_SetVectorcall()`** to the C API which sets the vectorcall field of a given **`PyFunctionObject`**.

  Warning: extensions using this API must preserve the behavior of the unaltered function!
- gh-59121 [https://github.com/python/cpython/issues/59121]: Fixed an assert that prevented `PyRun_InteractiveOne` from providing tracebacks when parsing from the provided FD.
- bpo-45383 [https://bugs.python.org/issue?@action=redirect&bpo=45383]: The **`PyType_FromSpec()`** API will now find and use a metaclass based on the provided bases. An error will be raised if there is a metaclass conflict.

Python 3.11.0 beta 1

Release date: 2022-05-06

Security

- gh-57684 [https://github.com/python/cpython/issues/57684]: Add the **`-P`** command line option and the **`PYTHONSAFEPATH`** environment variable to not prepend a potentially unsafe path to **`sys.path`**. Patch by Victor Stinner.

Core and Builtins

- gh-89519 [https://github.com/python/cpython/issues/89519]: Chaining classmethod descriptors (introduced in bpo-19072 [https://bugs.python.org/issue?@action=redirect&bpo=19072]) is deprecated. It can no longer be used to wrap other descriptors such as property(). The core design of this feature was flawed, and it caused a number of downstream problems.
- gh-92345 [https://github.com/python/cpython/issues/92345]: `pymain_run_python()` now imports `readline` and `rlcompleter` before sys.path is extended to include the current working directory of an interactive interpreter. Non-interactive interpreters are not affected.
- bpo-43857 [https://bugs.python.org/issue?@action=redirect&bpo=43857]: Improve the **`AttributeError`** message when deleting a missing attribute. Patch by Géry Ogam.
- gh-92245 [https://github.com/python/cpython/issues/92245]: Make sure that PEP 523 is respected in all cases. In 3.11a7, specialization may have prevented Python-to-Python calls respecting PEP 523.
- gh-92203 [https://github.com/python/cpython/issues/92203]: Add a closure keyword-only parameter to **`exec()`**. It can only be

specified when exec-ing a code object that uses free variables. When specified, it must be a tuple, with exactly the number of cell variables referenced by the code object. closure has a default value of `None`, and it must be `None` if the code object doesn’t refer to any free variables.

- gh-91173 [https://github.com/python/cpython/issues/91173]: Disable frozen modules in debug builds. Patch by Kumar Aditya.
- gh-92114 [https://github.com/python/cpython/issues/92114]: Improve error message when subscript a type with `__class_getitem__` set to `None`.
- gh-92112 [https://github.com/python/cpython/issues/92112]: Fix crash triggered by an evil custom `mro()` on a metaclass.
- gh-92063 [https://github.com/python/cpython/issues/92063]: The `PRECALL_METHOD_DESCRIPTOR_FAST_WITH_KEYWORDS` instruction now ensures methods are called only on objects of the correct type.
- gh-92031 [https://github.com/python/cpython/issues/92031]: Deoptimize statically allocated code objects during `Py_FINALIZE()` so that future `_PyCode_Quicken` calls always start with unquickened code.
- gh-92036 [https://github.com/python/cpython/issues/92036]: Fix a crash in subinterpreters related to the garbage collector. When a subinterpreter is deleted, untrack all objects tracked by its GC. To prevent a crash in deallocator functions expecting objects to be tracked by the GC, leak a strong reference to these objects on purpose, so they are never deleted and their deallocator functions are not called. Patch by Victor Stinner.
- gh-92032 [https://github.com/python/cpython/issues/92032]: The interpreter can now autocomplete soft keywords, as of now `match`, `case`, and `_` (wildcard pattern) from **PEP 634** [https://peps.python.org/pep-0634/].
- gh-87999 [https://github.com/python/cpython/issues/87999]: The warning emitted by the Python parser for a numeric

literal immediately followed by keyword has been changed from deprecation warning to syntax warning.

- gh-91869 [https://github.com/python/cpython/issues/91869]: Fix an issue where specialized opcodes with extended arguments could produce incorrect tracing output or lead to assertion failures.
- gh-91603 [https://github.com/python/cpython/issues/91603]: Speed up `types.UnionType` instantiation. Based on patch provided by Yurii Karabas.
- gh-89373 [https://github.com/python/cpython/issues/89373]: If Python is built in debug mode, Python now ensures that deallocator functions leave the current exception unchanged. Patch by Victor Stinner.
- gh-91632 [https://github.com/python/cpython/issues/91632]: Fix a minor memory leak at exit: release the memory of the **`generic_alias_iterator`** type. Patch by Donghee Na.
- gh-81548 [https://github.com/python/cpython/issues/81548]: Octal escapes with value larger than `0o377` now produce a `DeprecationWarning`. In a future Python version they will be a `SyntaxWarning` and eventually a `SyntaxError`.
- bpo-43950 [https://bugs.python.org/issue?@action=redirect&bpo=43950]: Use a single compact table for line starts, ends and column offsets. Reduces memory consumption for location info by half
- gh-91102 [https://github.com/python/cpython/issues/91102]: Use Argument Clinic for **`EncodingMap`**. Patch by Oleg Iarygin.
- gh-91636 [https://github.com/python/cpython/issues/91636]: Fixed a crash in a garbage-collection edge-case, in which a `PyFunction_Type.tp_clear` function could leave a python function object in an inconsistent state.
- gh-91603 [https://github.com/python/cpython/issues/91603]: Speed up `isinstance()` and `issubclass()` checks for `types.UnionType`. Patch by Yurii Karabas.
- gh-91625 [https://github.com/python/cpython/issues/91625]: Fixed a bug in which adaptive opcodes ignored any preceding `EXTENDED_ARG`s on specialization failure.

- gh-78607 [https://github.com/python/cpython/issues/78607]: The LLTRACE special build now looks for the name `__lltrace__` defined in module globals, rather than the name `__ltrace__`, which had been introduced as a typo.
- gh-91576 [https://github.com/python/cpython/issues/91576]: Speed up iteration of ascii strings by 50%. Patch by Kumar Aditya.
- gh-89279 [https://github.com/python/cpython/issues/89279]: Improve interpreter performance on Windows by inlining a few specific macros.
- gh-91502 [https://github.com/python/cpython/issues/91502]: Add a new **`_PyFrame_IsEntryFrame()`** API function, to check if a **`PyFrameObject`** is an entry frame. Patch by Pablo Galindo.
- gh-91266 [https://github.com/python/cpython/issues/91266]: Refactor the `bytearray` strip methods `strip`, `lstrip` and `rstrip` to use a common implementation.
- gh-91479 [https://github.com/python/cpython/issues/91479]: Replaced the `__note__` field of **`BaseException`** (added in an earlier version of 3.11) with the final design of **PEP 678** [https://peps.python.org/pep-0678/]. Namely, **`BaseException`** gets an **`add_note()`** method, and its `__notes__` field is created when necessary.
- gh-46055 [https://github.com/python/cpython/issues/46055]: Speed up right shift of negative integers, by removing unnecessary creation of temporaries. Original patch by Xinhang Xu, reworked by Mark Dickinson.
- gh-91462 [https://github.com/python/cpython/issues/91462]: Make the interpreter’s low-level tracing (lltrace) feature output more readable by displaying opcode names (rather than just numbers), and by displaying stack contents before each opcode.
- gh-89455 [https://github.com/python/cpython/issues/89455]: Fixed an uninitialized bool value in the traceback printing code path that was introduced by the initial bpo-45292 [https://bugs.python.org/issue?@action=redirect&bpo=45292] exception groups work.

- gh-91421 [https://github.com/python/cpython/issues/91421]: Fix a potential integer overflow in _Py_DecodeUTF8Ex.
- gh-91428 [https://github.com/python/cpython/issues/91428]: Add `static const char *const _PyOpcode_OpName[256] = {...};` to `opcode.h` for debug builds to assist in debugging the Python interpreter. It is now more convenient to make various forms of debugging output more human-readable by including opcode names rather than just the corresponding decimal digits.
- bpo-47120 [https://bugs.python.org/issue?@action=redirect&bpo=47120]: Make **`POP_JUMP_IF_TRUE`**, **`POP_JUMP_IF_FALSE`**, **`POP_JUMP_IF_NONE`** and **`POP_JUMP_IF_NOT_NONE`** virtual, mapping to new relative jump opcodes.
- bpo-45317 [https://bugs.python.org/issue?@action=redirect&bpo=45317]: Add internal documentation explaining design of new (for 3.11) frame stack.

Library

- bpo-47197 [https://bugs.python.org/issue?@action=redirect&bpo=47197]: ctypes used to mishandle `void` return types, so that for instance a function declared like `ctypes.CFUNCTYPE(None, ctypes.c_int)` would be called with signature `int f(int)` instead of `void f(int)`. Wasm targets require function pointers to be called with the correct signatures so this led to crashes. The problem is now fixed.

Core and Builtins

- bpo-47120 [https://bugs.python.org/issue?@action=redirect&bpo=47120]: Make opcodes **`JUMP_IF_TRUE_OR_POP`** and **`JUMP_IF_FALSE_OR_POP`** relative rather than absolute.

- bpo-47177 [https://bugs.python.org/issue?@action=redirect&bpo=47177]: Replace the `f_lasti` member of the internal `_PyInterpreterFrame` structure with a `prev_instr` pointer, which reduces overhead in the main interpreter loop. The `f_lasti` attribute of Python-layer frame objects is preserved for backward-compatibility.
- bpo-46961 [https://bugs.python.org/issue?@action=redirect&bpo=46961]: Integer mod/remainder operations, including the three-argument form of **`pow()`**, now consistently return ints from the global small integer cache when applicable.
- bpo-46962 [https://bugs.python.org/issue?@action=redirect&bpo=46962]: Classes and functions that unconditionally declared their docstrings ignoring the `--without-doc-strings` compilation flag no longer do so.

  The classes affected are **`ctypes.UnionType`**, **`pickle.PickleBuffer`**, **`testcapi.RecursingInfinitelyError`**, and **`types.GenericAlias`**.

  The functions affected are 24 methods in **`ctypes`**.

  Patch by Oleg Iarygin.
- bpo-46942 [https://bugs.python.org/issue?@action=redirect&bpo=46942]: Use Argument Clinic for the **`types.MethodType`** constructor. Patch by Oleg Iarygin.
- bpo-46764 [https://bugs.python.org/issue?@action=redirect&bpo=46764]: Fix wrapping bound methods with @classmethod
- bpo-43464 [https://bugs.python.org/issue?@action=redirect&bpo=43464]: Optimize **`set.intersection()`** for non-set arguments.

- bpo-46721 [https://bugs.python.org/issue?@action=redirect&bpo=46721]: Optimize `set.issuperset()` for non-set argument.

- bpo-46509 [https://bugs.python.org/issue?@action=redirect&bpo=46509]: Add type-specialized versions of the `Py_DECREF()`, and use them for `float`, `int`, `str`, `bool`, and `None` to avoid pointer-chasing at runtime where types are known at C compile time.

Build

- bpo-46045 [https://bugs.python.org/issue?@action=redirect&bpo=46045]: Do not use POSIX semaphores on NetBSD

Core and Builtins

- bpo-36819 [https://bugs.python.org/issue?@action=redirect&bpo=36819]: Fix crashes in built-in encoders with error handlers that return position less or equal than the starting position of non-encodable characters.

- bpo-34093 [https://bugs.python.org/issue?@action=redirect&bpo=34093]: `marshal.dumps()` uses `FLAG_REF` for all interned strings. This makes output more deterministic and helps reproducible build.

- bpo-26579 [https://bugs.python.org/issue?@action=redirect&bpo=26579]: Added `object.__getstate__` which provides the default implementation of the `__getstate__()` method.

  Copying and pickling instances of subclasses of builtin types bytearray, set, frozenset, collections.OrderedDict, collections.deque, weakref.WeakSet, and datetime.tzinfo now copies and pickles instance attributes implemented as slots.

Library

- gh-87901 [https://github.com/python/cpython/issues/87901]: Add the *encoding* parameter to **`os.popen()`**.
- gh-90997 [https://github.com/python/cpython/issues/90997]: Fix an issue where **`dis`** utilities may interpret populated inline cache entries as valid instructions.
- gh-92332 [https://github.com/python/cpython/issues/92332]: Deprecate **`typing.Text`** (removal of the class is currently not planned). Patch by Alex Waygood.
- gh-78157 [https://github.com/python/cpython/issues/78157]: Deprecate nested classes in enum definitions becoming members – in 3.13 they will be normal classes; add `member` and `nonmember` functions to allow control over results now.
- gh-92356 [https://github.com/python/cpython/issues/92356]: Fixed a performance regression in ctypes function calls.
- gh-90997 [https://github.com/python/cpython/issues/90997]: Show the actual named values stored in inline caches when `show_caches=True` is passed to **`dis`** utilities.
- gh-92301 [https://github.com/python/cpython/issues/92301]: Prefer `close_range()` to iterating over procfs for file descriptor closing in **`subprocess`** for better performance.
- gh-67248 [https://github.com/python/cpython/issues/67248]: Sort the miscellaneous topics in Cmd.do_help()

- gh-92210 [https://github.com/python/cpython/issues/92210]: Port `socket.__init__` to Argument Clinic. Patch by Cinder.
- gh-80010 [https://github.com/python/cpython/issues/80010]: Add support for generalized ISO 8601 parsing to **`datetime.datetime.fromisoformat()`**, **`datetime.date.fromisoformat()`** and **`datetime.time.fromisoformat()`**. Patch by Paul Ganssle.
- gh-92118 [https://github.com/python/cpython/issues/92118]: Fix a 3.11 regression in **`contextmanager()`**, which caused it to propagate exceptions with incorrect tracebacks.
- gh-90887 [https://github.com/python/cpython/issues/90887]: Adding `COPYFILE_STAT`, `COPYFILE_ACL` and `COPYFILE_XATTR` constants for **`os.fcopyfile()`** available in macOs.
- gh-91215 [https://github.com/python/cpython/issues/91215]: For **`@~dataclasses.dataclass`**, add *weakref_slot*. The new parameter defaults to `False`. If true, and if `slots=True`, add a slot named `"__weakref__"`, which will allow instances to be weakref'd. Contributed by Eric V. Smith
- gh-85984 [https://github.com/python/cpython/issues/85984]: New function os.login_tty() for Unix.
- gh-92128 [https://github.com/python/cpython/issues/92128]: Add **`__class_getitem__()`** to **`logging.LoggerAdapter`** and **`logging.StreamHandler`**, allowing them to be parameterized at runtime. Patch by Alex Waygood.
- gh-92049 [https://github.com/python/cpython/issues/92049]: Forbid pickling constants `re._constants.SUCCESS` etc. Previously, pickling did not fail, but the result could not be unpickled.

- gh-92062 [https://github.com/python/cpython/issues/92062]: **`inspect.Parameter`** now raises **`ValueError`** if `name` is a keyword, in addition to the existing check that it is an identifier.

- gh-87390 [https://github.com/python/cpython/issues/87390]: Add an `__unpacked__` attribute to **`types.GenericAlias`**. Patch by Jelle Zijlstra.

- gh-88089 [https://github.com/python/cpython/issues/88089]: Add support for generic **`typing.NamedTuple`**.

- gh-91996 [https://github.com/python/cpython/issues/91996]: New http.HTTPMethod enum to represent all the available HTTP request methods in a convenient way

- gh-91984 [https://github.com/python/cpython/issues/91984]: Modified test strings in test_argparse.py to not contain trailing spaces before end of line.

- gh-91952 [https://github.com/python/cpython/issues/91952]: Add `encoding="locale"` support to **`TextIOWrapper.reconfigure()`**.

- gh-91954 [https://github.com/python/cpython/issues/91954]: Add *encoding* and *errors* arguments to **`subprocess.getoutput()`** and **`subprocess.getstatusoutput()`**.

- bpo-47029 [https://bugs.python.org/issue?@action=redirect&bpo=47029]: Always close the read end of the pipe used by **`multiprocessing.Queue`** *after* the last write of buffered data to the write end of the pipe to avoid **`BrokenPipeError`** at garbage collection and at **`multiprocessing.Queue.close()`** calls. Patch by Géry Ogam.

- gh-91928 [https://github.com/python/cpython/issues/91928]: Add `datetime.UTC` alias for `datetime.timezone.utc`.

Patch by Kabir Kwatra.

- gh-68966 [https://github.com/python/cpython/issues/68966]: The **`mailcap`** module is now deprecated and will be removed in Python 3.13. See **PEP 594** [https://peps.python.org/pep-0594/] for the rationale and the **`mimetypes`** module for an alternative. Patch by Victor Stinner.

- gh-91401 [https://github.com/python/cpython/issues/91401]: Provide a way to disable **`subprocess`** use of `vfork()` just in case it is ever needed and document the existing mechanism for `posix_spawn()`.

- gh-64783 [https://github.com/python/cpython/issues/64783]: Fix **`signal.NSIG`** value on FreeBSD to accept signal numbers greater than 32, like **`signal.SIGRTMIN`** and **`signal.SIGRTMAX`**. Patch by Victor Stinner.

- gh-91910 [https://github.com/python/cpython/issues/91910]: Add missing f prefix to f-strings in error messages from the **`multiprocessing`** and **`asyncio`** modules.

- gh-91860 [https://github.com/python/cpython/issues/91860]: Add **`typing.dataclass_transform()`**, implementing **PEP 681** [https://peps.python.org/pep-0681/]. Patch by Jelle Zijlstra.

- gh-91832 [https://github.com/python/cpython/issues/91832]: Add `required` attribute to **`argparse.Action`** repr output.

- gh-91827 [https://github.com/python/cpython/issues/91827]: In the **`tkinter`** module add method `info_patchlevel()` which returns the exact version of the Tcl library as a named tuple similar to **`sys.version_info`**.

- gh-84461 [https://github.com/python/cpython/issues/84461]: Add **`--enable-wasm-pthreads`** to enable pthreads support for

WASM builds. `Emscripten/node` no longer has threading enabled by default. Include additional file systems.

- gh-91821 [https://github.com/python/cpython/issues/91821]: Fix unstable `test_from_tuple` test in `test_decimal.py`.
- gh-91217 [https://github.com/python/cpython/issues/91217]: Deprecate the xdrlib module.
- gh-91217 [https://github.com/python/cpython/issues/91217]: Deprecate the uu module.
- gh-91760 [https://github.com/python/cpython/issues/91760]: More strict rules will be applied for numerical group references and group names in regular expressions. For now, a deprecation warning is emitted for group references and group names which will be errors in future Python versions.
- gh-84461 [https://github.com/python/cpython/issues/84461]: Add provisional `sys._emscripten_info` named tuple with build-time and run-time information about Emscripten platform.
- gh-90623 [https://github.com/python/cpython/issues/90623]: `signal.raise_signal()` and `os.kill()` now check immediately for pending signals. Patch by Victor Stinner.
- gh-91734 [https://github.com/python/cpython/issues/91734]: Fix OSS audio support on Solaris.
- gh-90633 [https://github.com/python/cpython/issues/90633]: Include the passed value in the exception thrown by `typing.assert_never()`. Patch by Jelle Zijlstra.
- gh-91700 [https://github.com/python/cpython/issues/91700]: Compilation of regular expression containing a

conditional expression `(?(group)...)` now raises an appropriate **`re.error`** if the group number refers to not defined group. Previously an internal RuntimeError was raised.

- gh-91231 [https://github.com/python/cpython/issues/91231]: Add an optional keyword *shutdown_timeout* parameter to the **`multiprocessing.BaseManager`** constructor. Kill the process if terminate() takes longer than the timeout. Patch by Victor Stinner.
- gh-91621 [https://github.com/python/cpython/issues/91621]: Fix **`typing.get_type_hints()`** for **`collections.abc.Callable`**. Patch by Shantanu Jain.
- gh-90568 [https://github.com/python/cpython/issues/90568]: Parsing `\N` escapes of Unicode Named Character Sequences in a **`regular expression`** raises now **`re.error`** instead of `TypeError`.
- gh-91670 [https://github.com/python/cpython/issues/91670]: Remove deprecated `SO` config variable in **`sysconfig`**.
- gh-91217 [https://github.com/python/cpython/issues/91217]: Deprecate the telnetlib module.
- gh-91217 [https://github.com/python/cpython/issues/91217]: Deprecate the sunau module.
- gh-91217 [https://github.com/python/cpython/issues/91217]: Deprecate the spwd module.
- gh-91217 [https://github.com/python/cpython/issues/91217]: Deprecate the sndhdr module, as well as inline needed functionality for `email.mime.MIMEAudio`.

- gh-91616 [https://github.com/python/cpython/issues/91616]: `re` module, fix `fullmatch()` mismatch when using Atomic Grouping or Possessive Quantifiers.

- gh-91217 [https://github.com/python/cpython/issues/91217]: Deprecate the ‘pipes’ module.

- gh-91217 [https://github.com/python/cpython/issues/91217]: Deprecate the ossaudiodev module.

- bpo-47256 [https://bugs.python.org/issue?@action=redirect&bpo=47256]: `re` module, limit the maximum capturing group to 1,073,741,823 in 64-bit build, this increases the depth of backtracking.

- gh-91217 [https://github.com/python/cpython/issues/91217]: Deprecate the nis module.

- gh-91595 [https://github.com/python/cpython/issues/91595]: Fix the comparison of character and integer inside `Tools.gdb.libpython.write_repr()`. Patch by Yu Liu.

- gh-74166 [https://github.com/python/cpython/issues/74166]: Add option to raise all errors from `create_connection()` in an `ExceptionGroup` when it fails to create a connection. The default remains to raise only the last error that had occurred when multiple addresses were tried.

- gh-91487 [https://github.com/python/cpython/issues/91487]: Optimize asyncio UDP speed, over 100 times faster when transferring a large file.

- gh-91575 [https://github.com/python/cpython/issues/91575]: Update case-insensitive matching in the `re` module to the latest Unicode version.

- gh-90622 [https://github.com/python/cpython/issues/90622]: In `concurrent.futures.process.ProcessPoolExecutor` disallow the “fork” multiprocessing start method when the new `max_tasks_per_child` feature is used as the mix of threads+fork can hang the child processes. Default to using the safe “spawn” start method in that circumstance if no `mp_context` was supplied.

- gh-89022 [https://github.com/python/cpython/issues/89022]: In **`sqlite3`**, `SQLITE_MISUSE` result codes are now mapped to **`InterfaceError`** instead of **`ProgrammingError`**. Also, more accurate exceptions are raised when binding parameters fail. Patch by Erlend E. Aasland.

- gh-91526 [https://github.com/python/cpython/issues/91526]: Stop calling `os.device_encoding(file.fileno())` in **`TextIOWrapper`**. It was complex, never documented, and didn’t work for most cases. (Patch by Inada Naoki.)

- gh-88116 [https://github.com/python/cpython/issues/88116]: Change the frame-related functions in the **`inspect`** module to return a regular object (that is backwards compatible with the old tuple-like interface) that include the extended **PEP 657** [https://peps.python.org/pep-0657/] position information (end line number, column and end column). The affected functions are: **`inspect.getframeinfo()`**, **`inspect.getouterframes()`**, **`inspect.getinnerframes()`**, **`inspect.stack()`** and **`inspect.trace()`**. Patch by Pablo Galindo.

- gh-69093 [https://github.com/python/cpython/issues/69093]: Add indexing and slicing support to **`sqlite3.Blob`**. Patch by Aviv Palivoda and Erlend E. Aasland.

- gh-69093 [https://github.com/python/cpython/issues/69093]: Add context manager support to **`sqlite3.Blob`**. Patch by Aviv

Palivoda and Erlend E. Aasland.

- gh-91217 [https://github.com/python/cpython/issues/91217]: Deprecate nntplib.
- gh-91217 [https://github.com/python/cpython/issues/91217]: Deprecate msilib.
- gh-91404 [https://github.com/python/cpython/issues/91404]: Improve the performance of **`re`** matching by using computed gotos (or “threaded code”) on supported platforms and removing expensive pointer indirections.
- gh-91217 [https://github.com/python/cpython/issues/91217]: Deprecate the imghdr module.
- gh-91217 [https://github.com/python/cpython/issues/91217]: Deprecate the crypt module.
- gh-91276 [https://github.com/python/cpython/issues/91276]: Make space for longer opcodes in **`dis`** output.
- bpo-47000 [https://bugs.python.org/issue?@action=redirect&bpo=47000]: Make **`TextIOWrapper`** uses locale encoding when `encoding="locale"` is specified even in UTF-8 mode.
- gh-91230 [https://github.com/python/cpython/issues/91230]: **`warnings.catch_warnings()`** now accepts arguments for **`warnings.simplefilter()`**, providing a more concise way to locally ignore warnings or convert them to errors.
- gh-91217 [https://github.com/python/cpython/issues/91217]: Deprecate the chunk module.
- gh-91498 [https://github.com/python/cpython/issues/91498]: Add the `TCP_CONNECTION_INFO` option (available on macOS) to **`socket`**.

- bpo-47260 [https://bugs.python.org/issue?@action=redirect&bpo=47260]: Fix `os.closerange()` potentially being a no-op in a Linux seccomp sandbox.

- bpo-47087 [https://bugs.python.org/issue?@action=redirect&bpo=47087]: Implement `typing.Required` and `typing.NotRequired` (**PEP 655** [https://peps.python.org/pep-0655/]). Patch by David Foster and Jelle Zijlstra.

- bpo-47061 [https://bugs.python.org/issue?@action=redirect&bpo=47061]: Deprecate cgi and cgitb.

- bpo-47061 [https://bugs.python.org/issue?@action=redirect&bpo=47061]: Deprecate audioop.

- bpo-47000 [https://bugs.python.org/issue?@action=redirect&bpo=47000]: Add **`locale.getencoding()`** to get the current locale encoding. It is similar to `locale.getpreferredencoding(False)` but ignores the Python UTF-8 Mode.

- bpo-42012 [https://bugs.python.org/issue?@action=redirect&bpo=42012]: Add **`wsgiref.types`**, containing WSGI-specific types for static type checking.

- bpo-47227 [https://bugs.python.org/issue?@action=redirect&bpo=47227]: Suppress expression chaining for more **`re`** parsing errors.

- bpo-47211 [https://bugs.python.org/issue?@action=redirect&bpo=47211]: Remove undocumented and never working function `re.template()` and flag `re.TEMPLATE`. This was later reverted in 3.11.0b2 and deprecated instead.

- bpo-47135 [https://bugs.python.org/issue?@action=redirect&bpo=47135]: **`decimal.localcontext()`** now accepts context attributes via keyword arguments

- bpo-43323 [https://bugs.python.org/issue?@action=redirect&bpo=43323]: Fix errors in the `email` module if the charset itself contains undecodable/unencodable characters.

- bpo-46841 [https://bugs.python.org/issue?@action=redirect&bpo=46841]: Disassembly of quickened code.

- bpo-46681 [https://bugs.python.org/issue?@action=redirect&bpo=46681]: Forward gzip.compress() compresslevel to zlib.

- bpo-45100 [https://bugs.python.org/issue?@action=redirect&bpo=45100]: Add `typing.get_overloads()` and `typing.clear_overloads()`. Patch by Jelle Zijlstra.

- bpo-44807 [https://bugs.python.org/issue?@action=redirect&bpo=44807]: `typing.Protocol` no longer silently replaces `__init__()` methods defined on subclasses. Patch by Adrian Garcia Badaracco.

- bpo-46787 [https://bugs.python.org/issue?@action=redirect&bpo=46787]: Fix `concurrent.futures.ProcessPoolExecutor` exception memory leak

- bpo-46720 [https://bugs.python.org/issue?@action=redirect&bpo=46720]: Add support for path-like objects to `multiprocessing.set_executable()` for Windows to be on a par with Unix-like systems. Patch by Géry Ogam.

- bpo-46696 [https://bugs.python.org/issue?@action=redirect&bpo=46696]: Add `SO_INCOMING_CPU` constant to `socket`.

- bpo-46053 [https://bugs.python.org/issue?@action=redirect&bpo=46053]: Fix OSS audio support on NetBSD.

- bpo-45639 [https://bugs.python.org/issue?@action=redirect&bpo=45639]: `image/avif` and `image/webp` were added to **`mimetypes`**.

- bpo-46285 [https://bugs.python.org/issue?@action=redirect&bpo=46285]: Add command-line option `-p/--protocol` to module **`http.server`** which specifies the HTTP version to which the server is conformant (HTTP/1.1 conformant servers can now be run from the command-line interface of module **`http.server`**). Patch by Géry Ogam.

- bpo-44791 [https://bugs.python.org/issue?@action=redirect&bpo=44791]: Accept ellipsis as the last argument of **`typing.Concatenate`**.

- bpo-46547 [https://bugs.python.org/issue?@action=redirect&bpo=46547]: Remove variables leaking into `pydoc.Helper` class namespace.

- bpo-46415 [https://bugs.python.org/issue?@action=redirect&bpo=46415]: Fix ipaddress.ip_{address,interface,network} raising TypeError instead of ValueError if given invalid tuple as address parameter.

- bpo-46075 [https://bugs.python.org/issue?@action=redirect&bpo=46075]: `CookieJar` with `DefaultCookiePolicy` now can process cookies from localhost with domain=localhost explicitly specified in Set-Cookie header.

- bpo-45995 [https://bugs.python.org/issue?@action=redirect&bpo=45995]: Add a “z” option to the string formatting specification that coerces negative zero floating-point values to positive zero after rounding to the format precision. Contributed by John Belmonte.

- bpo-26175 [https://bugs.python.org/issue?@action=redirect&bpo=26175]: Fully implement the **`io.BufferedIOBase`** or **`io.TextIOBase`** interface for **`tempfile.SpooledTemporaryFile`** objects. This lets them work correctly with higher-level layers (like compression modules). Patch by Carey Metcalfe.

- bpo-45138 [https://bugs.python.org/issue?@action=redirect&bpo=45138]: Fix a regression in the **`sqlite3`** trace callback where bound parameters were not expanded in the passed statement string. The regression was introduced in Python 3.10 by bpo-40318 [https://bugs.python.org/issue?@action=redirect&bpo=40318]. Patch by Erlend E. Aasland.

- bpo-44863 [https://bugs.python.org/issue?@action=redirect&bpo=44863]: Allow **`TypedDict`** subclasses to also include **`Generic`** as a base class in class based syntax. Thereby allowing the user to define a generic `TypedDict`, just like a user-defined generic but with `TypedDict` semantics.

- bpo-44587 [https://bugs.python.org/issue?@action=redirect&bpo=44587]: Fix BooleanOptionalAction to not automatically add a default string. If a default string is desired, use a formatter to add it.

- bpo-43827 [https://bugs.python.org/issue?@action=redirect&bpo=43827]: All positional-or-keyword parameters to `ABCMeta.__new__` are now positional-only to avoid conflicts with keyword arguments to be passed to **`__init_subclass__()`**.

- bpo-43218 [https://bugs.python.org/issue?@action=redirect&bpo=43218]: Prevent creation of a venv whose path contains the PATH separator. This could affect the usage of the activate script. Patch by Dustin Rodrigues.
- bpo-38435 [https://bugs.python.org/issue?@action=redirect&bpo=38435]: Add a `process_group` parameter to **`subprocess.Popen`** to help move more things off of the unsafe `preexec_fn` parameter.
- bpo-42066 [https://bugs.python.org/issue?@action=redirect&bpo=42066]: Fix cookies getting sorted in **`CookieJar.__iter__()`** which is an extra behavior and not mentioned in RFC 2965 or Netscape cookie protocol. Now the cookies in `CookieJar` follows the order of the `Set-Cookie` header. Patch by Iman Kermani.
- bpo-40617 [https://bugs.python.org/issue?@action=redirect&bpo=40617]: Add **`create_window_function()`** to **`sqlite3.Connection`** for creating aggregate window functions. Patch by Erlend E. Aasland.
- bpo-40676 [https://bugs.python.org/issue?@action=redirect&bpo=40676]: Convert **`csv`** to use Argument Clinic for **`csv.field_size_limit()`**, **`csv.get_dialect()`**, **`csv.unregister_dialect()`** and **`csv.list_dialects()`**.
- bpo-39716 [https://bugs.python.org/issue?@action=redirect&bpo=39716]: Raise an `ArgumentError` when the same subparser name is added twice to an `argparse.ArgumentParser`. This is consistent with the (default) behavior when the same option string is added twice to an `ArgumentParser`.

- bpo-36073 [https://bugs.python.org/issue?@action=redirect&bpo=36073]: Raise **`ProgrammingError`** instead of segfaulting on recursive usage of cursors in **`sqlite3`** converters. Patch by Sergey Fedoseev.

- bpo-34975 [https://bugs.python.org/issue?@action=redirect&bpo=34975]: Adds a `start_tls()` method to **`StreamWriter`**, which upgrades the connection with TLS using the given **`SSLContext`**.

- bpo-22276 [https://bugs.python.org/issue?@action=redirect&bpo=22276]: **`Path`** methods **`glob()`** and **`rglob()`** return only directories if *pattern* ends with a pathname components separator (`/` or **`sep`**). Patch by Eisuke Kawashima.

- bpo-24905 [https://bugs.python.org/issue?@action=redirect&bpo=24905]: Add **`blobopen()`** to **`sqlite3.Connection`**. **`sqlite3.Blob`** allows incremental I/O operations on blobs. Patch by Aviv Palivoda and Erlend E. Aasland.

Documentation

- gh-91888 [https://github.com/python/cpython/issues/91888]: Add a new `gh` role to the documentation to link to GitHub issues.
- gh-91783 [https://github.com/python/cpython/issues/91783]: Document security issues concerning the use of the function **`shutil.unpack_archive()`**
- gh-91547 [https://github.com/python/cpython/issues/91547]: Remove “Undocumented modules” page.
- gh-91298 [https://github.com/python/cpython/issues/91298]: In `importlib.resources.abc`, refined the documentation of the Traversable Protocol, applying changes from importlib_resources 5.7.1.

- bpo-44347 [https://bugs.python.org/issue?@action=redirect&bpo=44347]: Clarify the meaning of *dirs_exist_ok*, a kwarg of **`shutil.copytree()`**.
- bpo-36329 [https://bugs.python.org/issue?@action=redirect&bpo=36329]: Remove ‘make -C Doc serve’ in favour of ‘make -C Doc htmlview’
- bpo-47189 [https://bugs.python.org/issue?@action=redirect&bpo=47189]: Add a What’s New in Python 3.11 entry for the Faster CPython project. Documentation by Ken Jin and Kumar Aditya.
- bpo-38668 [https://bugs.python.org/issue?@action=redirect&bpo=38668]: Update the introduction to documentation for **`os.path`** to remove warnings that became irrelevant after the implementations of **PEP 383** [https://peps.python.org/pep-0383/] and **PEP 529** [https://peps.python.org/pep-0529/].
- bpo-47115 [https://bugs.python.org/issue?@action=redirect&bpo=47115]: The documentation now lists which members of C structs are part of the Limited API/Stable ABI.
- bpo-46962 [https://bugs.python.org/issue?@action=redirect&bpo=46962]: All docstrings in code snippets are now wrapped into **`PyDoc_STR`** to follow the guideline of **PEP 7’s Documentation Strings paragraph** [https://peps.python.org/pep-0007/#documentation-strings]. Patch by Oleg Iarygin.
- bpo-26792 [https://bugs.python.org/issue?@action=redirect&bpo=26792]: Improve the docstrings of **`runpy.run_module()`** and **`runpy.run_path()`**. Original patch by Andrew Brezovsky.

Tests

- gh-92169 [https://github.com/python/cpython/issues/92169]: Use `warnings_helper.import_deprecated()` to import deprecated modules uniformly in tests. Patch by Hugo van Kemenade.

- gh-84461 [https://github.com/python/cpython/issues/84461]: When multiprocessing is enabled, libregrtest can now use a Python executable other than `sys.executable` via the `--python` flag.
- gh-91904 [https://github.com/python/cpython/issues/91904]: Fix initialization of **`PYTHONREGRTEST_UNICODE_GUARD`** which prevented running regression tests on non-UTF-8 locale.
- gh-91752 [https://github.com/python/cpython/issues/91752]: Added @requires_zlib to test.test_tools.test_freeze.TestFreeze.
- gh-91607 [https://github.com/python/cpython/issues/91607]: Fix `test_concurrent_futures` to test the correct multiprocessing start method context in several cases where the test logic mixed this up.
- bpo-40280 [https://bugs.python.org/issue?@action=redirect&bpo=40280]: Threading tests are now skipped on WASM targets without pthread support.
- bpo-47109 [https://bugs.python.org/issue?@action=redirect&bpo=47109]: Test for **`ctypes.macholib.dyld`**, **`ctypes.macholib.dylib`**, and **`ctypes.macholib.framework`** are brought from manual pre-**`unittest`** times to **`ctypes.test`** location and structure. Patch by Oleg Iarygin.
- bpo-29890 [https://bugs.python.org/issue?@action=redirect&bpo=29890]: Add tests for **`ipaddress.IPv4Interface`** and **`ipaddress.IPv6Interface`** construction with tuple arguments. Original patch and tests by louisom.

Build

- gh-89452 [https://github.com/python/cpython/issues/89452]: gdbm-compat is now preferred over ndbm if both are available on the system. This allows avoiding the problematic ndbm.h on macOS.

- gh-91731 [https://github.com/python/cpython/issues/91731]: Python is now built with `-std=c11` compiler option, rather than `-std=c99`. Patch by Victor Stinner.
- bpo-47152 [https://bugs.python.org/issue?@action=redirect&bpo=47152]: Add script and make target for generating `sre_constants.h`.
- bpo-47103 [https://bugs.python.org/issue?@action=redirect&bpo=47103]: Windows `PGInstrument` builds now copy a required DLL into the output directory, making it easier to run the profile stage of a PGO build.

Windows

- bpo-46907 [https://bugs.python.org/issue?@action=redirect&bpo=46907]: Update Windows installer to use SQLite 3.38.3.
- bpo-47239 [https://bugs.python.org/issue?@action=redirect&bpo=47239]: Fixed –list and –list-paths output for Python install manager when used in an active virtual environment.
- bpo-46907 [https://bugs.python.org/issue?@action=redirect&bpo=46907]: Update Windows installer to use SQLite 3.38.2.
- bpo-46785 [https://bugs.python.org/issue?@action=redirect&bpo=46785]: Fix race condition between **`os.stat()`** and unlinking a file on Windows, by using errors codes returned by `FindFirstFileW()` when appropriate in `win32_xstat_impl`.
- bpo-40859 [https://bugs.python.org/issue?@action=redirect&bpo=40859]: Update Windows build to use xz-5.2.5

macOS

- bpo-46907 [https://bugs.python.org/issue?@action=redirect&bpo=46907]: Update macOS installer to

SQLite 3.38.4.

Tools/Demos

- gh-91583 [https://github.com/python/cpython/issues/91583]: Fix regression in the code generated by Argument Clinic for functions with the `defining_class` parameter.
- gh-91575 [https://github.com/python/cpython/issues/91575]: Add script `Tools/scripts/generate_re_casefix.py` and the make target `regen-re` for generating additional data for case-insensitive matching according to the current Unicode version.
- gh-91551 [https://github.com/python/cpython/issues/91551]: Remove the ancient Pynche color editor. It has moved to https://gitlab.com/warsaw/pynche

C API

- gh-88279 [https://github.com/python/cpython/issues/88279]: Deprecate the C functions: **`PySys_SetArgv()`**, **`PySys_SetArgvEx()`**, **`PySys_SetPath()`**. Patch by Victor Stinner.

- gh-92154 [https://github.com/python/cpython/issues/92154]: Added the **`PyCode_GetCode()`** function. This function does the equivalent of the Python code `getattr(code_object, 'co_code')`.

- gh-92173 [https://github.com/python/cpython/issues/92173]: Fix the `closure` argument to **`PyEval_EvalCodeEx()`**.

- gh-91320 [https://github.com/python/cpython/issues/91320]: Fix C++ compiler warnings about “old-style cast” (`g++ -Wold-style-cast`) in the Python C API. Use C++ `reinterpret_cast<>` and `static_cast<>` casts when the Python C API is used in C++. Patch by Victor Stinner.

- gh-80527 [https://github.com/python/cpython/issues/80527]: Mark functions as deprecated by **PEP 623** [https://peps.python.org/pep-0623/]: **`PyUnicode_AS_DATA()`**, **`PyUnicode_AS_UNICODE()`**, **`PyUnicode_GET_DATA_SIZE()`**, **`PyUnicode_GET_SIZE()`**. Patch by Victor Stinner.

- gh-91768 [https://github.com/python/cpython/issues/91768]: **`Py_REFCNT()`**, **`Py_TYPE()`**, **`Py_SIZE()`** and **`Py_IS_TYPE()`** functions argument type is now `PyObject*`, rather than `const PyObject*`. Patch by Victor Stinner.

- gh-91020 [https://github.com/python/cpython/issues/91020]: Add `PyBytes_Type.tp_alloc` to initialize `PyBytesObject.ob_shash` for bytes subclasses.

- bpo-40421 [https://bugs.python.org/issue?@action=redirect&bpo=40421]: Add `PyFrame_GetLasti` C-API function to access frame object's `f_lasti` attribute safely from C code.

- bpo-35134 [https://bugs.python.org/issue?@action=redirect&bpo=35134]: Remove the `Include/code.h` header file. C extensions should only include the main `<Python.h>` header file. Patch by Victor Stinner.

- bpo-47169 [https://bugs.python.org/issue?@action=redirect&bpo=47169]: **`PyOS_CheckStack()`** is now exported in the Stable ABI on Windows.

- bpo-47169 [https://bugs.python.org/issue?@action=redirect&bpo=47169]: **`PyThread_get_thread_native_id()`** is excluded from the stable ABI on platforms where it doesn't exist (like Solaris).

- bpo-46343 [https://bugs.python.org/issue?@action=redirect&bpo=46343]: Added **`PyErr_GetHandledException()`** and **`PyErr_SetHandledException()`**

as simpler alternatives to `PyErr_GetExcInfo()` and `PyErr_SetExcInfo()`.

They are included in the stable ABI.

Python 3.11.0 alpha 7

Release date: 2022-04-05

Core and Builtins

- bpo-47212 [https://bugs.python.org/issue?@action=redirect&bpo=47212]: Raise `IndentationError` instead of `SyntaxError` for a bare `except` with no following indent. Improve `SyntaxError` locations for an un-parenthesized generator used as arguments. Patch by Matthieu Dartiailh.
- bpo-47186 [https://bugs.python.org/issue?@action=redirect&bpo=47186]: Replace **`JUMP_IF_NOT_EG_MATCH`** by `CHECK_EG_MATCH` + jump.
- bpo-47176 [https://bugs.python.org/issue?@action=redirect&bpo=47176]: Emscripten builds cannot handle signals in the usual way due to platform limitations. Python can now handle signals. To use, set Module.Py_EmscriptenSignalBuffer to be a single byte SharedArrayBuffer and set Py_EMSCRIPTEN_SIGNAL_HANDLING to 1. Writing a number into the SharedArrayBuffer will cause the corresponding signal to be raised into the Python thread.
- bpo-47186 [https://bugs.python.org/issue?@action=redirect&bpo=47186]: Replace **`JUMP_IF_NOT_EXC_MATCH`** by `CHECK_EXC_MATCH` + jump.
- bpo-47120 [https://bugs.python.org/issue?@action=redirect&bpo=47120]: Replace the absolute jump

opcode `JUMP_NO_INTERRUPT` by the relative `JUMP_BACKWARD_NO_INTERRUPT`.

- bpo-46841 [https://bugs.python.org/issue?@action=redirect&bpo=46841]: Avoid unnecessary allocations when comparing code objects.
- bpo-47182 [https://bugs.python.org/issue?@action=redirect&bpo=47182]: Fix a crash when using a named unicode character like `"\N{digit nine}"` after the main interpreter has been initialized a second time.
- bpo-47162 [https://bugs.python.org/issue?@action=redirect&bpo=47162]: WebAssembly cannot deal with bad function pointer casts (different count or types of arguments). Python can now use call trampolines to mitigate the problem. Define **`PY_CALL_TRAMPOLINE`** to enable call trampolines.
- bpo-46775 [https://bugs.python.org/issue?@action=redirect&bpo=46775]: Some Windows system error codes(>= 10000) are now mapped into the correct errno and may now raise a subclass of `OSError`. Patch by Donghee Na.
- bpo-47129 [https://bugs.python.org/issue?@action=redirect&bpo=47129]: Improve error messages in f-string syntax errors concerning empty expressions.
- bpo-47117 [https://bugs.python.org/issue?@action=redirect&bpo=47117]: Fix a crash if we fail to decode characters in interactive mode if the tokenizer buffers are uninitialized. Patch by Pablo Galindo.
- bpo-47127 [https://bugs.python.org/issue?@action=redirect&bpo=47127]: Speed up calls to c functions with keyword arguments by 25% with specialization. Patch by Kumar Aditya.
- bpo-47120 [https://bugs.python.org/issue?@action=redirect&bpo=47120]: Replaced **`JUMP_ABSOLUTE`** by the relative jump `JUMP_BACKWARD`.
- bpo-42197 [https://bugs.python.org/issue?@action=redirect&bpo=42197]: **`PyFrame_FastToLocalsWithError()`**

and **`PyFrame_LocalsToFast()`** are no longer called during profiling nor tracing. C code can access the `f_locals` attribute of **`PyFrameObject`** by calling **`PyFrame_GetLocals()`**.

- bpo-47070 [https://bugs.python.org/issue?@action=redirect&bpo=47070]: Improve performance of `array_inplace_repeat` by reducing the number of invocations of `memcpy`. Refactor the `repeat` and inplace `repeat` methods of `array`, `bytes`, `bytearray` and `unicodeobject` to use the common `_PyBytes_Repeat`.
- bpo-47053 [https://bugs.python.org/issue?@action=redirect&bpo=47053]: Reduce de-optimization in the specialized `BINARY_OP_INPLACE_ADD_UNICODE` opcode.
- bpo-47045 [https://bugs.python.org/issue?@action=redirect&bpo=47045]: Remove the `f_state` field from the _PyInterpreterFrame struct. Add the `owner` field to the _PyInterpreterFrame struct to make ownership explicit to simplify clearing and deallocing frames and generators.

Library

- bpo-46968 [https://bugs.python.org/issue?@action=redirect&bpo=46968]: Check for the existence of the “sys/auxv.h” header in **`faulthandler`** to avoid compilation problems in systems where this header doesn’t exist. Patch by Pablo Galindo

Core and Builtins

- bpo-46329 [https://bugs.python.org/issue?@action=redirect&bpo=46329]: Use low bit of `LOAD_GLOBAL` to indicate whether to push a `NULL` before the global. Helps streamline the call sequence a bit.
- bpo-46841 [https://bugs.python.org/issue?@action=redirect&bpo=46841]: Quicken bytecode in-place by storing it as part of the corresponding `PyCodeObject`.

- bpo-47012 [https://bugs.python.org/issue?@action=redirect&bpo=47012]: Speed up iteration of **`bytes`** and **`bytearray`** by 30%. Patch by Kumar Aditya.
- bpo-47009 [https://bugs.python.org/issue?@action=redirect&bpo=47009]: Improved the performance of **`list.append()`** and list comprehensions by optimizing for the common case, where no resize is needed. Patch by Dennis Sweeney.
- bpo-47005 [https://bugs.python.org/issue?@action=redirect&bpo=47005]: Improve performance of `bytearray_repeat` and `bytearray_irepeat` by reducing the number of invocations of `memcpy`.

Library

- bpo-46829 [https://bugs.python.org/issue?@action=redirect&bpo=46829]: Deprecate passing a message into **`asyncio.Future.cancel()`** and **`asyncio.Task.cancel()`**

Core and Builtins

- bpo-46993 [https://bugs.python.org/issue?@action=redirect&bpo=46993]: Speed up **`bytearray`** creation from **`list`** and **`tuple`** by 40%. Patch by Kumar Aditya.
- bpo-39829 [https://bugs.python.org/issue?@action=redirect&bpo=39829]: Removed the `__len__()` call when initializing a list and moved initializing to `list_extend`. Patch by Jeremiah Pascual.
- bpo-46944 [https://bugs.python.org/issue?@action=redirect&bpo=46944]: Speed up throwing exception in generator with **`METH_FASTCALL`** calling convention. Patch by Kumar Aditya.
- bpo-46841 [https://bugs.python.org/issue?@action=redirect&bpo=46841]: Modify **`STORE_SUBSCR`** to use an inline cache entry (rather than its oparg) as an adaptive counter.

- bpo-46841 [https://bugs.python.org/issue?@action=redirect&bpo=46841]: Use inline caching for **`PRECALL`** and **`CALL`**, and remove the internal machinery for managing the (now unused) non-inline caches.
- bpo-46881 [https://bugs.python.org/issue?@action=redirect&bpo=46881]: Statically allocate and initialize the latin1 characters.
- bpo-46838 [https://bugs.python.org/issue?@action=redirect&bpo=46838]: Improve syntax errors for incorrect function definitions. Patch by Pablo Galindo
- bpo-43721 [https://bugs.python.org/issue?@action=redirect&bpo=43721]: Fix docstrings of **`getter`**, **`setter`**, and **`deleter`** to clarify that they create a new copy of the property.
- bpo-43224 [https://bugs.python.org/issue?@action=redirect&bpo=43224]: Make grammar changes required for PEP 646.

Library

- bpo-47208 [https://bugs.python.org/issue?@action=redirect&bpo=47208]: Allow vendors to override **`CTYPES_MAX_ARGCOUNT`**.

- bpo-23689 [https://bugs.python.org/issue?@action=redirect&bpo=23689]: **`re`** module: fix memory leak when a match is terminated by a signal or memory allocation failure. Patch by Ma Lin.

- bpo-47167 [https://bugs.python.org/issue?@action=redirect&bpo=47167]: Allow overriding a future compliance check in **`asyncio.Task`**.

- bpo-47151 [https://bugs.python.org/issue?@action=redirect&bpo=47151]: When subprocess tries to use vfork, it now falls back to fork if vfork returns an error.

This allows use in situations where vfork isn't allowed by the OS kernel.

- bpo-47152 [https://bugs.python.org/issue?@action=redirect&bpo=47152]: Convert the `re` module into a package. Deprecate modules `sre_compile`, `sre_constants` and `sre_parse`.

- bpo-4833 [https://bugs.python.org/issue?@action=redirect&bpo=4833]: Add **`ZipFile.mkdir()`**

- bpo-27929 [https://bugs.python.org/issue?@action=redirect&bpo=27929]: Fix `asyncio.loop.sock_connect()` to only resolve names for `socket.AF_INET` or `socket.AF_INET6` families. Resolution may not make sense for other families, like `socket.AF_BLUETOOTH` and `socket.AF_UNIX`.

- bpo-14265 [https://bugs.python.org/issue?@action=redirect&bpo=14265]: Adds the fully qualified test name to unittest output

- bpo-47061 [https://bugs.python.org/issue?@action=redirect&bpo=47061]: Deprecate the aifc module.

- bpo-39622 [https://bugs.python.org/issue?@action=redirect&bpo=39622]: Handle Ctrl+C in asyncio programs to interrupt the main task.

- bpo-47101 [https://bugs.python.org/issue?@action=redirect&bpo=47101]: `hashlib.algorithms_available` now lists only algorithms that are provided by activated crypto providers on OpenSSL 3.0. Legacy algorithms are not listed unless the legacy provider has been loaded into the default OSSL context.

- bpo-47099 [https://bugs.python.org/issue?@action=redirect&bpo=47099]: All **`URLError`** exception messages raised in **`urllib.request.URLopener`** now contain a colon between `ftp error` and the rest of the message. Previously, **`open_ftp()`** missed the colon. Patch by Oleg Iarygin.

- bpo-47099 [https://bugs.python.org/issue?@action=redirect&bpo=47099]: Exception chaining is changed from **`Exception.with_traceback()`**/**`sys.exc_info()`** to **PEP 3134** [https://peps.python.org/pep-3134/]. Patch by Oleg Iarygin.

- bpo-47095 [https://bugs.python.org/issue?@action=redirect&bpo=47095]: **`hashlib`**’s internal `_blake2` module now prefers `libb2` from https://www.blake2.net/ over Python’s vendored copy of blake2.

- bpo-47098 [https://bugs.python.org/issue?@action=redirect&bpo=47098]: The Keccak Code Package for **`hashlib`**’s internal `_sha3` module has been replaced with tiny_sha3. The module is used as fallback when Python is built without OpenSSL.

- bpo-47088 [https://bugs.python.org/issue?@action=redirect&bpo=47088]: Implement **`typing.LiteralString`**, part of **PEP 675** [https://peps.python.org/pep-0675/]. Patch by Jelle Zijlstra.

- bpo-42885 [https://bugs.python.org/issue?@action=redirect&bpo=42885]: Optimize **`re.search()`**, **`re.split()`**, **`re.findall()`**, **`re.finditer()`** and **`re.sub()`** for regular expressions starting with `\A` or `^`.

- bpo-23691 [https://bugs.python.org/issue?@action=redirect&bpo=23691]: Protect the **`re.finditer()`** iterator from re-entering.

- bpo-47067 [https://bugs.python.org/issue?@action=redirect&bpo=47067]: Optimize calling `GenericAlias` objects by using **PEP 590** [https://peps.python.org/pep-0590/] `vectorcall` and by replacing `PyObject_SetAttrString` with `PyObject_SetAttr`.

- bpo-28080 [https://bugs.python.org/issue?@action=redirect&bpo=28080]: Add the *metadata_encoding* parameter in the **`zipfile.ZipFile`** constructor and the `--metadata-encoding` option in the **`zipfile`** CLI to allow reading zipfiles using non-standard codecs to encode the filenames within the archive.

- bpo-47000 [https://bugs.python.org/issue?@action=redirect&bpo=47000]: Make **`io.text_encoding()`** returns "utf-8" when UTF-8 mode is enabled.

- bpo-42369 [https://bugs.python.org/issue?@action=redirect&bpo=42369]: Fix thread safety of **`zipfile._SharedFile.tell()`** to avoid a "zipfile.BadZipFile: Bad CRC-32 for file" exception when reading a **`ZipFile`** from multiple threads.

- bpo-38256 [https://bugs.python.org/issue?@action=redirect&bpo=38256]: Fix **`binascii.crc32()`** when it is compiled to use zlib'c crc32 to work properly on inputs 4+GiB in length instead of returning the wrong result. The workaround prior to this was to always feed the function data in increments smaller than 4GiB or to just call the zlib module function.

  We also have **`binascii.crc32()`** release the GIL when computing on larger inputs as **`zlib.crc32()`** and **`hashlib`** do.

  This also boosts performance on Windows as it now uses the zlib crc32 implementation for **`binascii.crc32()`** for a

2-3x speedup.

That the stdlib has a crc32 API in two modules is a known historical oddity. This moves us closer to a single implementation behind them.

- bpo-47066 [https://bugs.python.org/issue?@action=redirect&bpo=47066]: Global inline flags (e.g. `(?i)`) can now only be used at the start of the regular expressions. Using them not at the start of expression was deprecated since Python 3.6.
- bpo-39394 [https://bugs.python.org/issue?@action=redirect&bpo=39394]: A warning about inline flags not at the start of the regular expression now contains the position of the flag.
- bpo-433030 [https://bugs.python.org/issue?@action=redirect&bpo=433030]: Add support of atomic grouping (`(?>...)`) and possessive quantifiers (`*+`, `++`, `?+`, `{m,n}+`) in **`regular expressions`**.
- bpo-47062 [https://bugs.python.org/issue?@action=redirect&bpo=47062]: Implement **`asyncio.Runner`** context manager.
- bpo-46382 [https://bugs.python.org/issue?@action=redirect&bpo=46382]: **`dataclass()`** `slots=True` now correctly omits slots already defined in base classes. Patch by Arie Bovenberg.
- bpo-47057 [https://bugs.python.org/issue?@action=redirect&bpo=47057]: Use FASTCALL convention for `FutureIter.throw()`
- bpo-47061 [https://bugs.python.org/issue?@action=redirect&bpo=47061]: Deprecate the various modules

listed by **PEP 594** [https://peps.python.org/pep-0594/]:

aifc, asynchat, asyncore, audioop, cgi, cgitb, chunk, crypt, imghdr, msilib, nntplib, nis, ossaudiodev, pipes, smtpd, sndhdr, spwd, sunau, telnetlib, uu, xdrlib

- bpo-34790 [https://bugs.python.org/issue?@action=redirect&bpo=34790]: Remove passing coroutine objects to **`asyncio.wait()`**.
- bpo-47039 [https://bugs.python.org/issue?@action=redirect&bpo=47039]: Normalize `repr()` of asyncio future and task objects.
- bpo-2604 [https://bugs.python.org/issue?@action=redirect&bpo=2604]: Fix bug where doctests using globals would fail when run multiple times.
- bpo-45150 [https://bugs.python.org/issue?@action=redirect&bpo=45150]: Add **`hashlib.file_digest()`** helper for efficient hashing of file object.
- bpo-34861 [https://bugs.python.org/issue?@action=redirect&bpo=34861]: Made cumtime the default sorting key for cProfile
- bpo-45997 [https://bugs.python.org/issue?@action=redirect&bpo=45997]: Fix **`asyncio.Semaphore`** re-aquiring FIFO order.
- bpo-47022 [https://bugs.python.org/issue?@action=redirect&bpo=47022]: The **`asynchat`**, **`asyncore`** and **`smtpd`** modules have been deprecated since at least Python 3.6. Their documentation and deprecation warnings and have now been updated to note they will removed in Python 3.12 (**PEP 594** [https://peps.python.org/pep-0594/]).

- bpo-43253 [https://bugs.python.org/issue?@action=redirect&bpo=43253]: Fix a crash when closing transports where the underlying socket handle is already invalid on the Proactor event loop.
- bpo-40280 [https://bugs.python.org/issue?@action=redirect&bpo=40280]: **`select.select()`** now passes `NULL` to `select` for each empty fdset.
- bpo-47004 [https://bugs.python.org/issue?@action=redirect&bpo=47004]: Apply bugfixes from importlib_metadata 4.11.3, including bugfix for EntryPoint.extras, which was returning match objects and not the extras strings.
- bpo-46998 [https://bugs.python.org/issue?@action=redirect&bpo=46998]: Allow subclassing of **`typing.Any`**. Patch by Shantanu Jain.
- bpo-46995 [https://bugs.python.org/issue?@action=redirect&bpo=46995]: Deprecate missing **`asyncio.Task.set_name()`** for third-party task implementations, schedule making it mandatory in Python 3.13.
- bpo-46994 [https://bugs.python.org/issue?@action=redirect&bpo=46994]: Accept explicit contextvars.Context in **`asyncio.create_task()`** and **`asyncio.loop.create_task()`**.
- bpo-46981 [https://bugs.python.org/issue?@action=redirect&bpo=46981]: `typing.get_args(typing.Tuple[()])` now returns `()` instead of `((),)`.
- bpo-46968 [https://bugs.python.org/issue?@action=redirect&bpo=46968]: Add `os.sysconf_names['SC_MINSIGSTKSZ']`.

- bpo-46985 [https://bugs.python.org/issue?@action=redirect&bpo=46985]: Upgrade pip wheel bundled with ensurepip (pip 22.0.4)

- bpo-46968 [https://bugs.python.org/issue?@action=redirect&bpo=46968]: **`faulthandler`**: On Linux 5.14 and newer, dynamically determine size of signal handler stack size CPython allocates using `getauxval(AT_MINSIGSTKSZ)`. This changes allows for Python extension's request to Linux kernel to use AMX_TILE instruction set on Sapphire Rapids Xeon processor to succeed, unblocking use of the ISA in frameworks.

- bpo-46917 [https://bugs.python.org/issue?@action=redirect&bpo=46917]: The **`math.nan`** value is now always available. Patch by Victor Stinner.

- bpo-46955 [https://bugs.python.org/issue?@action=redirect&bpo=46955]: Expose **`asyncio.base_events.Server`** as **`asyncio.Server`**. Patch by Stefan Zabka.

- bpo-23325 [https://bugs.python.org/issue?@action=redirect&bpo=23325]: The **`signal`** module no longer assumes that **`SIG_IGN`** and **`SIG_DFL`** are small int singletons.

- bpo-46932 [https://bugs.python.org/issue?@action=redirect&bpo=46932]: Update bundled libexpat to 2.4.7

- bpo-46933 [https://bugs.python.org/issue?@action=redirect&bpo=46933]: The **`pwd`** module is now optional. **`os.path.expanduser()`** returns the path when the **`pwd`** module is not available.

- bpo-40059 [https://bugs.python.org/issue?@action=redirect&bpo=40059]: **PEP 680** [https://peps.python.org/pep-0680/], the **`tomllib`** module. Adds support for parsing TOML.

- bpo-464471 [https://bugs.python.org/issue?@action=redirect&bpo=464471]: **`asyncio.timeout()`** and **`asyncio.timeout_at()`** context managers added. Patch by Tin Tvrtković and Andrew Svetlov.

- bpo-46805 [https://bugs.python.org/issue?@action=redirect&bpo=46805]: Added raw datagram socket functions for asyncio: **`sock_sendto()`**, **`sock_recvfrom()`** and **`sock_recvfrom_into()`**.

- bpo-46644 [https://bugs.python.org/issue?@action=redirect&bpo=46644]: No longer require valid typeforms to be callable. This allows **`typing.Annotated`** to wrap **`typing.ParamSpecArgs`** and **`dataclasses.InitVar`**. Patch by Gregory Beauregard.

- bpo-46581 [https://bugs.python.org/issue?@action=redirect&bpo=46581]: Brings **`ParamSpec`** propagation for **`GenericAlias`** in line with **`Concatenate`** (and others).

- bpo-45413 [https://bugs.python.org/issue?@action=redirect&bpo=45413]: Define *posix_venv* and *nt_venv* sysconfig installation schemes to be used for bootstrapping new virtual environments. Add *venv* sysconfig installation scheme to get the appropriate one of the above. The schemes are identical to the pre-existing *posix_prefix* and *nt* install schemes. The **`venv`** module now uses the *venv* scheme to create new virtual environments instead of hardcoding the paths depending only on the platform. Downstream Python distributors customizing the *posix_prefix* or *nt* install scheme in a way that is not compatible with the install scheme used in virtual environments are encouraged not to customize the *venv* schemes. When Python itself runs in a virtual environment, `sysconfig.get_default_scheme()` and `sysconfig.get_preferred_scheme()` with `key="prefix"` returns *venv*.

- bpo-43224 [https://bugs.python.org/issue?@action=redirect&bpo=43224]: Implement support for PEP 646 in typing.py.
- bpo-43224 [https://bugs.python.org/issue?@action=redirect&bpo=43224]: Allow unpacking types.GenericAlias objects, e.g. `*tuple[int, str]`.
- bpo-46557 [https://bugs.python.org/issue?@action=redirect&bpo=46557]: Warnings captured by the logging module are now logged without a format string to prevent systems that group logs by the msg argument from grouping captured warnings together.
- bpo-41370 [https://bugs.python.org/issue?@action=redirect&bpo=41370]: `typing.get_type_hints()` now supports evaluating strings as forward references in PEP 585 generic aliases.
- bpo-46607 [https://bugs.python.org/issue?@action=redirect&bpo=46607]: Add `DeprecationWarning` to **`LegacyInterpolation`**, deprecated in the docstring since Python 3.2. Will be removed in Python 3.13. Use **`BasicInterpolation`** or **`ExtendedInterpolation`** instead.
- bpo-26120 [https://bugs.python.org/issue?@action=redirect&bpo=26120]: `pydoc` now excludes __future__ imports from the module’s data items.

- bpo-46480 [https://bugs.python.org/issue?@action=redirect&bpo=46480]: Add `typing.assert_type()`. Patch by Jelle Zijlstra.

- bpo-46421 [https://bugs.python.org/issue?@action=redirect&bpo=46421]: Fix a unittest issue where if the command was invoked as `python -m unittest` and the filename(s) began with a dot (.), a `ValueError` is returned.

- bpo-46245 [https://bugs.python.org/issue?@action=redirect&bpo=46245]: Add optional parameter *dir_fd* in `shutil.rmtree()`.

- bpo-22859 [https://bugs.python.org/issue?@action=redirect&bpo=22859]: **`unittest.TestProgram.usageExit()`** is marked as deprecated, to be removed in Python 3.13.

- bpo-46170 [https://bugs.python.org/issue?@action=redirect&bpo=46170]: Improve the error message when you try to subclass an instance of `typing.NewType`.

- bpo-40296 [https://bugs.python.org/issue?@action=redirect&bpo=40296]: Fix supporting generic aliases in `pydoc`.

- bpo-20392 [https://bugs.python.org/issue?@action=redirect&bpo=20392]: Fix inconsistency with uppercase file extensions in **`MimeTypes.guess_type()`**. Patch by Kumar Aditya.

- bpo-46030 [https://bugs.python.org/issue?@action=redirect&bpo=46030]: Add `LOCAL_CREDS`, `LOCAL_CREDS_PERSISTENT` and `SCM_CREDS2` FreeBSD constants to the socket module.

- bpo-44439 [https://bugs.python.org/issue?@action=redirect&bpo=44439]: Fix `.write()` method of a member file in `ZipFile`, when the input data is an object that supports the buffer protocol, the file length may be wrong.

- bpo-45171 [https://bugs.python.org/issue?@action=redirect&bpo=45171]: Fix handling of the `stacklevel` argument to logging functions in the **`logging`** module so that it is consistent across all logging functions and, as advertised, similar to the `stacklevel` argument used in **`warn()`**.
- bpo-24959 [https://bugs.python.org/issue?@action=redirect&bpo=24959]: Fix bug where **`unittest`** sometimes drops frames from tracebacks of exceptions raised in tests.
- bpo-44859 [https://bugs.python.org/issue?@action=redirect&bpo=44859]: Raise more accurate and **PEP 249** [https://peps.python.org/pep-0249/] compatible exceptions in **`sqlite3`**.
    - Raise **`InterfaceError`** instead of **`ProgrammingError`** for `SQLITE_MISUSE` errors.
    - Don't overwrite **`BufferError`** with **`ValueError`** when conversion to BLOB fails.
    - Raise **`ProgrammingError`** instead of **`Warning`** if user tries to **`execute()`** more than one SQL statement.
    - Raise **`ProgrammingError`** instead of **`ValueError`** if an SQL query contains null characters.
- bpo-44493 [https://bugs.python.org/issue?@action=redirect&bpo=44493]: Add missing terminated NUL in sockaddr_un's length

  This was potentially observable when using non-abstract AF_UNIX datagram sockets to processes written in another programming language.

- bpo-41930 [https://bugs.python.org/issue?@action=redirect&bpo=41930]: Add `serialize()` and `deserialize()` support to `sqlite3`. Patch by Erlend E. Aasland.

- bpo-33178 [https://bugs.python.org/issue?@action=redirect&bpo=33178]: Added `ctypes.BigEndianUnion` and `ctypes.LittleEndianUnion` classes, as originally documented in the library docs but not yet implemented.

- bpo-43352 [https://bugs.python.org/issue?@action=redirect&bpo=43352]: Add an Barrier object in synchronization primitives of *asyncio* Lib in order to be consistent with Barrier from *threading* and *multiprocessing* libs*

- bpo-35859 [https://bugs.python.org/issue?@action=redirect&bpo=35859]: `re` module, fix a few bugs about capturing group. In rare cases, capturing group gets an incorrect string. Patch by Ma Lin.

Documentation

- bpo-45099 [https://bugs.python.org/issue?@action=redirect&bpo=45099]: Document internal `asyncio` API.
- bpo-47126 [https://bugs.python.org/issue?@action=redirect&bpo=47126]: Update PEP URLs to **PEP 676** [https://peps.python.org/pep-0676/]’s new canonical form.
- bpo-47040 [https://bugs.python.org/issue?@action=redirect&bpo=47040]: Clarified the old Python versions compatibility note of `binascii.crc32()` / `zlib.adler32()` / `zlib.crc32()` functions.
- bpo-46033 [https://bugs.python.org/issue?@action=redirect&bpo=46033]: Clarify `for` statement execution in its doc.

- bpo-45790 [https://bugs.python.org/issue?@action=redirect&bpo=45790]: Adjust inaccurate phrasing in Defining Extension Types: Tutorial about the `ob_base` field and the macros used to access its contents.
- bpo-42340 [https://bugs.python.org/issue?@action=redirect&bpo=42340]: Document that in some circumstances **`KeyboardInterrupt`** may cause the code to enter an inconsistent state. Provided a sample workaround to avoid it if needed.
- bpo-41233 [https://bugs.python.org/issue?@action=redirect&bpo=41233]: Link the errnos referenced in `Doc/library/exceptions.rst` to their respective section in `Doc/library/errno.rst`, and vice versa. Previously this was only done for EINTR and InterruptedError. Patch by Yan “yyyyyyyan” Orestes.

Tests

- bpo-47205 [https://bugs.python.org/issue?@action=redirect&bpo=47205]: Skip test for **`sched_getaffinity()`** and **`sched_setaffinity()`** error case on FreeBSD.
- bpo-46126 [https://bugs.python.org/issue?@action=redirect&bpo=46126]: Restore ‘descriptions’ when running tests internally.
- bpo-47104 [https://bugs.python.org/issue?@action=redirect&bpo=47104]: Rewrite **`asyncio.to_thread()`** tests to use **`unittest.IsolatedAsyncioTestCase`**.
- bpo-40280 [https://bugs.python.org/issue?@action=redirect&bpo=40280]: The test suite is now passing on the Emscripten platform. All fork, socket, and subprocess-based tests are skipped.
- bpo-47037 [https://bugs.python.org/issue?@action=redirect&bpo=47037]: Skip `strftime("%4Y")` feature test on Windows. It can cause an assertion error in debug builds.

- bpo-46587 [https://bugs.python.org/issue?@action=redirect&bpo=46587]: Skip tests if platform's `strftime` does not support non-portable glibc extensions.
- bpo-47015 [https://bugs.python.org/issue?@action=redirect&bpo=47015]: A test case for `os.sendfile()` is converted from deprecated **`asyncore`** (see **PEP 594** [https://peps.python.org/pep-0594/]) to `asyncio`. Patch by Oleg Iarygin.

Build

- bpo-40280 [https://bugs.python.org/issue?@action=redirect&bpo=40280]: Add configure option `--enable-wasm-dynamic-linking` to enable `dlopen` and MAIN_MODULE / SIDE_MODULE on `wasm32-emscripten`.
- bpo-46023 [https://bugs.python.org/issue?@action=redirect&bpo=46023]: `makesetup` now detects and skips all duplicated module definitions. The first entry wins.
- bpo-40280 [https://bugs.python.org/issue?@action=redirect&bpo=40280]: Add SOABI `wasm32-emscripten` for Emscripten and `wasm32-wasi` for WASI on 32bit WASM as well as `wasm64` counter parts.
- bpo-47032 [https://bugs.python.org/issue?@action=redirect&bpo=47032]: Ensure Windows install builds fail correctly with a non-zero exit code when part of the build fails.
- bpo-47024 [https://bugs.python.org/issue?@action=redirect&bpo=47024]: Update OpenSSL to 1.1.1n for macOS installers and all Windows builds.
- bpo-46996 [https://bugs.python.org/issue?@action=redirect&bpo=46996]: The `tkinter` package now requires Tcl/Tk version 8.5.12 or newer.
- bpo-46973 [https://bugs.python.org/issue?@action=redirect&bpo=46973]: Add `regen-configure` make target to regenerate configure script with Christian's container image `quay.io/tiran/cpython_autoconf:269`.

- bpo-46917 [https://bugs.python.org/issue?@action=redirect&bpo=46917]: Building Python now requires support of IEEE 754 floating-point numbers. Patch by Victor Stinner.
- bpo-45774 [https://bugs.python.org/issue?@action=redirect&bpo=45774]: `configure` now verifies that all SQLite C APIs needed for the **`sqlite3`** extension module are found.

Windows

- bpo-47194 [https://bugs.python.org/issue?@action=redirect&bpo=47194]: Update `zlib` to v1.2.12 to resolve **CVE 2018-25032** [https://www.cve.org/CVERecord?id=CVE-2018-25032].
- bpo-47171 [https://bugs.python.org/issue?@action=redirect&bpo=47171]: Enables installing the `py.exe` launcher on Windows ARM64.
- bpo-46566 [https://bugs.python.org/issue?@action=redirect&bpo=46566]: Upgraded Python install manager to support a new `-V:company/tag` argument for full **PEP 514** [https://peps.python.org/pep-0514/] support and to detect ARM64 installs. The `-64` suffix on arguments is deprecated, but still selects any non-32-bit install. Setting **`PYLAUNCHER_ALLOW_INSTALL`** and specifying a version that is not installed will attempt to install the requested version from the Microsoft Store.
- bpo-47086 [https://bugs.python.org/issue?@action=redirect&bpo=47086]: The installer for Windows now includes documentation as loose HTML files rather than a single compiled `.chm` file.
- bpo-46907 [https://bugs.python.org/issue?@action=redirect&bpo=46907]: Update Windows installer to use SQLite 3.38.1.
- bpo-44549 [https://bugs.python.org/issue?@action=redirect&bpo=44549]: Update bzip2 to 1.0.8 in

Windows builds to mitigate **CVE 2016-3189** [https://www.cve.org/CVERecord?id=CVE-2016-3189] and **CVE 2019-12900** [https://www.cve.org/CVERecord?id=CVE-2019-12900].
- bpo-46948 [https://bugs.python.org/issue?@action=redirect&bpo=46948]: Prevent **CVE 2022-26488** [https://www.cve.org/CVERecord?id=CVE-2022-26488] by ensuring the Add to PATH option in the Windows installer uses the correct path when being repaired.

macOS

- bpo-46890 [https://bugs.python.org/issue?@action=redirect&bpo=46890]: Fix a regression in the setting of `sys._base_executable` in framework builds, and thereby fix a regression in **`venv`** virtual environments with such builds.
- bpo-46907 [https://bugs.python.org/issue?@action=redirect&bpo=46907]: Update macOS installer to SQLite 3.38.1.

Tools/Demos

- bpo-40280 [https://bugs.python.org/issue?@action=redirect&bpo=40280]: Replace Emscripten’s limited shell with Katie Bell’s browser-ui REPL from python-wasm project.

C API

- bpo-40421 [https://bugs.python.org/issue?@action=redirect&bpo=40421]: Add `PyFrame_GetBuiltins`, `PyFrame_GetGenerator` and `PyFrame_GetGlobals` C-API functions to access frame object attributes safely from C code.

- bpo-46850 [https://bugs.python.org/issue?@action=redirect&bpo=46850]: Move the private `_PyFrameEvalFunction` type, and private `_PyInterpreterState_GetEvalFrameFunc()` and `_PyInterpreterState_SetEvalFrameFunc()` functions to the internal C API. The `_PyFrameEvalFunction` callback function type now uses the `_PyInterpreterFrame` type which is part of the internal C API. Patch by Victor Stinner.

- bpo-46850 [https://bugs.python.org/issue?@action=redirect&bpo=46850]: Move the private undocumented `_PyEval_EvalFrameDefault()` function to the internal C API. The function now uses the `_PyInterpreterFrame` type which is part of the internal C API. Patch by Victor Stinner.

- bpo-46850 [https://bugs.python.org/issue?@action=redirect&bpo=46850]: Remove the private undocumented function `_PyEval_CallTracing()` from the C API. Call the public **`sys.call_tracing()`** function instead. Patch by Victor Stinner.

- bpo-46850 [https://bugs.python.org/issue?@action=redirect&bpo=46850]: Remove the private undocumented function `_PyEval_GetCoroutineOriginTrackingDepth()` from the C API. Call the public **`sys.get_coroutine_origin_tracking_depth()`** function instead. Patch by Victor Stinner.

- bpo-46850 [https://bugs.python.org/issue?@action=redirect&bpo=46850]: Remove the following private undocumented functions from the C API:

  - `_PyEval_GetAsyncGenFirstiter()`
  - `_PyEval_GetAsyncGenFinalizer()`
  - `_PyEval_SetAsyncGenFirstiter()`

  - `_PyEval_SetAsyncGenFinalizer()`

  Call the public **`sys.get_asyncgen_hooks()`** and **`sys.set_asyncgen_hooks()`** functions instead. Patch by Victor Stinner.

- bpo-46987 [https://bugs.python.org/issue?@action=redirect&bpo=46987]: Remove private functions `_PySys_GetObjectId()` and `_PySys_SetObjectId()`. Patch by Donghee Na.

- bpo-46906 [https://bugs.python.org/issue?@action=redirect&bpo=46906]: Add new functions to pack and unpack C double (serialize and deserialize): **`PyFloat_Pack2()`**, **`PyFloat_Pack4()`**, **`PyFloat_Pack8()`**, **`PyFloat_Unpack2()`**, **`PyFloat_Unpack4()`** and **`PyFloat_Unpack8()`**. Patch by Victor Stinner.

Python 3.11.0 alpha 6

Release date: 2022-03-07

Core and Builtins

- bpo-46940 [https://bugs.python.org/issue?@action=redirect&bpo=46940]: Avoid overriding **`AttributeError`** metadata information for nested attribute access calls. Patch by Pablo Galindo.
- bpo-46927 [https://bugs.python.org/issue?@action=redirect&bpo=46927]: Include the type’s name in the error message for subscripting non-generic types.
- bpo-46921 [https://bugs.python.org/issue?@action=redirect&bpo=46921]: Support vectorcall for `super()`. Patch by Ken Jin.
- bpo-46841 [https://bugs.python.org/issue?@action=redirect&bpo=46841]: Fix incorrect handling of inline

cache entries when specializing **`BINARY_OP`**.

- bpo-46841 [https://bugs.python.org/issue?@action=redirect&bpo=46841]: Use an oparg to simplify the construction of helpful error messages in **`GET_AWAITABLE`**.
- bpo-46903 [https://bugs.python.org/issue?@action=redirect&bpo=46903]: Make sure that str subclasses can be used as attribute names for instances with virtual dictionaries. Fixes regression in 3.11alpha
- bpo-46841 [https://bugs.python.org/issue?@action=redirect&bpo=46841]: Add more detailed specialization failure stats for **`COMPARE_OP`** followed by **`EXTENDED_ARG`**.
- bpo-46891 [https://bugs.python.org/issue?@action=redirect&bpo=46891]: Fix bug introduced during 3.11alpha where subclasses of `types.ModuleType` with `__slots__` were not initialized correctly, resulting in an interpreter crash.
- bpo-46841 [https://bugs.python.org/issue?@action=redirect&bpo=46841]: Use inline caching for **`LOAD_ATTR`**, **`LOAD_METHOD`**, and **`STORE_ATTR`**.
- bpo-46841 [https://bugs.python.org/issue?@action=redirect&bpo=46841]: Use inline cache for **`BINARY_SUBSCR`**.
- bpo-46841 [https://bugs.python.org/issue?@action=redirect&bpo=46841]: Use inline caching for **`COMPARE_OP`**.
- bpo-46864 [https://bugs.python.org/issue?@action=redirect&bpo=46864]: Deprecate `PyBytesObject.ob_shash`. It will be removed in Python 3.13.
- bpo-46841 [https://bugs.python.org/issue?@action=redirect&bpo=46841]: Use inline caching for **`UNPACK_SEQUENCE`**.
- bpo-46845 [https://bugs.python.org/issue?@action=redirect&bpo=46845]: Reduces dict size by removing hash value from hash table when all inserted keys are Unicode. For example,

`sys.getsizeof(dict.fromkeys("abcdefg"))` becomes 272 bytes from 352 bytes on 64bit platform.
- bpo-46841 [https://bugs.python.org/issue?@action=redirect&bpo=46841]: Use inline cache for **`LOAD_GLOBAL`**.
- bpo-46852 [https://bugs.python.org/issue?@action=redirect&bpo=46852]: Rename the private undocumented `float.__set_format__()` method to `float.__setformat__()` to fix a typo introduced in Python 3.7. The method is only used by test_float. Patch by Victor Stinner.
- bpo-46852 [https://bugs.python.org/issue?@action=redirect&bpo=46852]: Remove the undocumented private `float.__set_format__()` method, previously known as `float.__setformat__()` in Python 3.7. Its docstring said: “You probably don’t want to use this function. It exists mainly to be used in Python’s test suite.” Patch by Victor Stinner.
- bpo-40116 [https://bugs.python.org/issue?@action=redirect&bpo=40116]: Fix regression that dict.update(other) may don’t respect iterate order of other when other is key sharing dict.
- bpo-46712 [https://bugs.python.org/issue?@action=redirect&bpo=46712]: Share global string identifiers in deep-frozen modules.
- bpo-46430 [https://bugs.python.org/issue?@action=redirect&bpo=46430]: Fix memory leak in interned strings of deep-frozen modules.
- bpo-46841 [https://bugs.python.org/issue?@action=redirect&bpo=46841]: Store **`BINARY_OP`** caches inline using a new **`CACHE`** instruction.
- bpo-45107 [https://bugs.python.org/issue?@action=redirect&bpo=45107]: Specialize `LOAD_METHOD` for instances with a dict.
- bpo-44337 [https://bugs.python.org/issue?@action=redirect&bpo=44337]: Reduce the memory usage of specialized **`LOAD_ATTR`** and **`STORE_ATTR`** instructions.

- bpo-46729 [https://bugs.python.org/issue?@action=redirect&bpo=46729]: Add number of sub-exceptions to **`BaseException.__str__()`**.
- bpo-45885 [https://bugs.python.org/issue?@action=redirect&bpo=45885]: Don’t un-adapt **`COMPARE_OP`** when collecting specialization stats.
- bpo-46329 [https://bugs.python.org/issue?@action=redirect&bpo=46329]: Fix specialization stats gathering for **`PRECALL`** instructions.

Library

- bpo-46794 [https://bugs.python.org/issue?@action=redirect&bpo=46794]: Bump up the libexpat version into 2.4.6

Core and Builtins

- bpo-46823 [https://bugs.python.org/issue?@action=redirect&bpo=46823]: Implement a specialized combined opcode `LOAD_FAST__LOAD_ATTR_INSTANCE_VALUE`. Patch by Dennis Sweeney.

- bpo-46820 [https://bugs.python.org/issue?@action=redirect&bpo=46820]: Fix parsing a numeric literal immediately (without spaces) followed by “not in” keywords, like in `1not in x`. Now the parser only emits a warning, not a syntax error.

- bpo-46329 [https://bugs.python.org/issue?@action=redirect&bpo=46329]: Move `KW_NAMES` before `PRECALL` instruction in call sequence. Change `operand` of `CALL` to match `PRECALL` for easier specialization.

- bpo-46808 [https://bugs.python.org/issue?@action=redirect&bpo=46808]: Remove the `NEXT_BLOCK` macro from compile.c, and make the compiler automatically generate implicit blocks when they are needed.

- bpo-46329 [https://bugs.python.org/issue?@action=redirect&bpo=46329]: Add `PUSH_NULL` instruction. This is used as a prefix when evaluating a callable, so that the stack has the same shape for methods and other calls. `PRECALL_FUNCTION` and `PRECALL_METHOD` are merged into a single `PRECALL` instruction.

  There is no change in semantics.

- bpo-46762 [https://bugs.python.org/issue?@action=redirect&bpo=46762]: Fix an assert failure in debug builds when a '<', '>', or '=' is the last character in an f-string that's missing a closing right brace.

- bpo-46730 [https://bugs.python.org/issue?@action=redirect&bpo=46730]: Message of AttributeError caused by getting, setting or deleting a property without the corresponding function now mentions that the attribute is in fact a property and also specifies type of the class that it belongs to.

- bpo-46724 [https://bugs.python.org/issue?@action=redirect&bpo=46724]: Make sure that all backwards jumps use the `JUMP_ABSOLUTE` instruction, rather than `JUMP_FORWARD` with an argument of `(2**32)+offset`.

- bpo-46732 [https://bugs.python.org/issue?@action=redirect&bpo=46732]: Correct the docstring for the **`__bool__()`** method. Patch by Jelle Zijlstra.

- bpo-46072 [https://bugs.python.org/issue?@action=redirect&bpo=46072]: Add more detailed specialization failure statistics for **`BINARY_OP`**.

- bpo-46707 [https://bugs.python.org/issue?@action=redirect&bpo=46707]: Avoid potential exponential backtracking when producing some syntax errors involving lots of brackets. Patch by Pablo Galindo.

Library

- bpo-46323 [https://bugs.python.org/issue?@action=redirect&bpo=46323]: **`ctypes`** now allocates memory on the stack instead of on the heap to pass arguments while calling a Python callback function. Patch by Donghee Na.

Core and Builtins

- bpo-45923 [https://bugs.python.org/issue?@action=redirect&bpo=45923]: Add a quickened form of **`RESUME`** that skips quickening checks.
- bpo-46702 [https://bugs.python.org/issue?@action=redirect&bpo=46702]: Specialize **`UNPACK_SEQUENCE`** for **`tuple`** and **`list`** unpackings.
- bpo-46072 [https://bugs.python.org/issue?@action=redirect&bpo=46072]: Opcode pair stats are now gathered with `--enable-pystats`. Defining `DYNAMIC_EXECUTION_PROFILE` or `DXPAIRS` no longer has any effect.
- bpo-46675 [https://bugs.python.org/issue?@action=redirect&bpo=46675]: Allow more than 16 items in a split dict before it is combined. The limit is now 254.

Library

- bpo-40479 [https://bugs.python.org/issue?@action=redirect&bpo=40479]: Add a missing call to `va_end()` in `Modules/_hashopenssl.c`.

- bpo-46323 [https://bugs.python.org/issue?@action=redirect&bpo=46323]: Use `PyObject_Vectorcall()` while calling ctypes callback function. Patch by Donghee Na.

Core and Builtins

- bpo-46615 [https://bugs.python.org/issue?@action=redirect&bpo=46615]: When iterating over sets internally in `setobject.c`, acquire strong references to the resulting items from the set. This prevents crashes in corner-cases of various set operations where the set gets mutated.
- bpo-45828 [https://bugs.python.org/issue?@action=redirect&bpo=45828]: The bytecode compiler now attempts to apply runtime stack manipulations at compile-time (whenever it is feasible to do so).
- bpo-30496 [https://bugs.python.org/issue?@action=redirect&bpo=30496]: Fixed a minor portability issue in the implementation of `PyLong_FromLong()`, and added a fast path for single-digit integers to `PyLong_FromLongLong()`.

Library

- bpo-25707 [https://bugs.python.org/issue?@action=redirect&bpo=25707]: Fixed a file leak in `xml.etree.ElementTree.iterparse()` when the iterator is not exhausted. Patch by Jacob Walls.

- bpo-46877 [https://bugs.python.org/issue?@action=redirect&bpo=46877]: Export `unittest.doModuleCleanups()` in `unittest`. Patch by Kumar Aditya.

- bpo-46848 [https://bugs.python.org/issue?@action=redirect&bpo=46848]: For performance, use the optimized string-searching implementations from `find()` and `rfind()` for **`find()`** and **`rfind()`**.

- bpo-46736 [https://bugs.python.org/issue?@action=redirect&bpo=46736]: `SimpleHTTPRequestHandler` now uses HTML5 grammar. Patch by Donghee Na.
- bpo-44886 [https://bugs.python.org/issue?@action=redirect&bpo=44886]: Inherit asyncio proactor datagram transport from `asyncio.DatagramTransport`.
- bpo-46827 [https://bugs.python.org/issue?@action=redirect&bpo=46827]: Support UDP sockets in `asyncio.loop.sock_connect()` for selector-based event loops. Patch by Thomas Grainger.
- bpo-46811 [https://bugs.python.org/issue?@action=redirect&bpo=46811]: Make test suite support Expat >=2.4.5
- bpo-46252 [https://bugs.python.org/issue?@action=redirect&bpo=46252]: Raise `TypeError` if `ssl.SSLSocket` is passed to transport-based APIs.
- bpo-46784 [https://bugs.python.org/issue?@action=redirect&bpo=46784]: Fix libexpat symbols collisions with user dynamically loaded or statically linked libexpat in embedded Python.
- bpo-46786 [https://bugs.python.org/issue?@action=redirect&bpo=46786]: The HTML serialisation in xml.etree.ElementTree now writes `embed`, `source`, `track` and `wbr` as empty tags, as defined in HTML 5.
- bpo-39327 [https://bugs.python.org/issue?@action=redirect&bpo=39327]: `shutil.rmtree()` can now work

with VirtualBox shared folders when running from the guest operating-system.

- bpo-45390 [https://bugs.python.org/issue?@action=redirect&bpo=45390]: Propagate `asyncio.CancelledError` message from inner task to outer awaiter.
- bpo-46756 [https://bugs.python.org/issue?@action=redirect&bpo=46756]: Fix a bug in `urllib.request.HTTPPasswordMgr.find_user_password()` and `urllib.request.HTTPPasswordMgrWithPriorAuth.is_authenticated()` which allowed to bypass authorization. For example, access to URI `example.org/foobar` was allowed if the user was authorized for URI `example.org/foo`.
- bpo-46737 [https://bugs.python.org/issue?@action=redirect&bpo=46737]: `random.gauss()` and `random.normalvariate()` now have default arguments.
- bpo-46752 [https://bugs.python.org/issue?@action=redirect&bpo=46752]: Add task groups to asyncio (structured concurrency, inspired by Trio’s nurseries). This also introduces a change to task cancellation, where a cancelled task can’t be cancelled again until it calls .uncancel().
- bpo-46724 [https://bugs.python.org/issue?@action=redirect&bpo=46724]: Fix `dis` behavior on negative jump offsets.
- bpo-46333 [https://bugs.python.org/issue?@action=redirect&bpo=46333]: The `__repr__()` method of `typing.ForwardRef` now includes the `module` parameter of `typing.ForwardRef` when it is set.

- bpo-46643 [https://bugs.python.org/issue?@action=redirect&bpo=46643]: In `typing.get_type_hints()`, support evaluating stringified `ParamSpecArgs` and `ParamSpecKwargs` annotations. Patch by Gregory Beauregard.

- bpo-45863 [https://bugs.python.org/issue?@action=redirect&bpo=45863]: When the `tarfile` module creates a pax format archive, it will put an integer representation of timestamps in the ustar header (if possible) for the benefit of older unarchivers, in addition to the existing full-precision timestamps in the pax extended header.

- bpo-46066 [https://bugs.python.org/issue?@action=redirect&bpo=46066]: Deprecate kwargs-based syntax for `typing.TypedDict` definitions. It had confusing semantics when specifying totality, and was largely unused. Patch by Jingchen Ye.

- bpo-46676 [https://bugs.python.org/issue?@action=redirect&bpo=46676]: Make `typing.ParamSpec` args and kwargs equal to themselves. Patch by Gregory Beauregard.

- bpo-46323 [https://bugs.python.org/issue?@action=redirect&bpo=46323]: `ctypes.CFUNCTYPE()` and `ctypes.WINFUNCTYPE()` now fail to create the type if its `_argtypes_` member contains too many arguments. Previously, the error was only raised when calling a function. Patch by Victor Stinner.

- bpo-46672 [https://bugs.python.org/issue?@action=redirect&bpo=46672]: Fix `NameError` in `asyncio.gather()` when initial type check fails.

- bpo-46659 [https://bugs.python.org/issue?@action=redirect&bpo=46659]: The `calendar.LocaleTextCalendar` and `calendar.LocaleHTMLCalendar` classes now use `locale.getlocale()`, instead of using `locale.getdefaultlocale()`, if no locale is specified. Patch by Victor Stinner.

- bpo-46659 [https://bugs.python.org/issue?@action=redirect&bpo=46659]: The `locale.getdefaultlocale()` function is deprecated and will be removed in Python 3.13. Use `locale.setlocale()`, `locale.getpreferredencoding(False)` and `locale.getlocale()` functions instead. Patch by Victor Stinner.

- bpo-46655 [https://bugs.python.org/issue?@action=redirect&bpo=46655]: In `typing.get_type_hints()`, support evaluating bare stringified `TypeAlias` annotations. Patch by Gregory Beauregard.

- bpo-45948 [https://bugs.python.org/issue?@action=redirect&bpo=45948]: Fixed a discrepancy in the C implementation of the `xml.etree.ElementTree` module. Now, instantiating an `xml.etree.ElementTree.XMLParser` with a `target=None` keyword provides a default `xml.etree.ElementTree.TreeBuilder` target as the Python implementation does.

- bpo-46626 [https://bugs.python.org/issue?@action=redirect&bpo=46626]: Expose Linux's `IP_BIND_ADDRESS_NO_PORT` option in `socket`.

- bpo-46521 [https://bugs.python.org/issue?@action=redirect&bpo=46521]: Fix a bug in the `codeop` module that was incorrectly identifying invalid code involving string quotes as valid code.

- bpo-46571 [https://bugs.python.org/issue?@action=redirect&bpo=46571]: Improve `typing.no_type_check()`.

  Now it does not modify external classes and functions. We also now correctly mark classmethods as not to be type checked.

- bpo-46400 [https://bugs.python.org/issue?@action=redirect&bpo=46400]: expat: Update libexpat from 2.4.1 to 2.4.4

- bpo-46556 [https://bugs.python.org/issue?@action=redirect&bpo=46556]: Deprecate undocumented support for using a `pathlib.Path` object as a context manager.

- bpo-46534 [https://bugs.python.org/issue?@action=redirect&bpo=46534]: Implement **PEP 673** [https://peps.python.org/pep-0673/] `typing.Self`. Patch by James Hilton-Balfe.

- bpo-46522 [https://bugs.python.org/issue?@action=redirect&bpo=46522]: Make various module `__getattr__` AttributeErrors more closely match a typical AttributeError

- bpo-46475 [https://bugs.python.org/issue?@action=redirect&bpo=46475]: Add `typing.Never` and `typing.assert_never()`. Patch by Jelle Zijlstra.

- bpo-46333 [https://bugs.python.org/issue?@action=redirect&bpo=46333]: The **`__eq__()`** and **`__hash__()`** methods of `typing.ForwardRef` now honor the `module` parameter of `typing.ForwardRef`. Forward references from different modules are now differentiated.

- bpo-46246 [https://bugs.python.org/issue?@action=redirect&bpo=46246]: Add missing `__slots__` to `importlib.metadata.DeprecatedList`. Patch by Arie Bovenberg.

- bpo-46232 [https://bugs.python.org/issue?@action=redirect&bpo=46232]: The **`ssl`** module now handles certificates with bit strings in DN correctly.

- bpo-46195 [https://bugs.python.org/issue?@action=redirect&bpo=46195]: **`typing.get_type_hints()`** no longer adds `Optional` to parameters with `None` as a default. This aligns to changes to PEP 484 in https://github.com/python/peps/pull/689

- bpo-31369 [https://bugs.python.org/issue?@action=redirect&bpo=31369]: Add **`RegexFlag`** to `re.__all__` and documented it. Add **`NOFLAG`** to indicate no flags being set.

- bpo-45898 [https://bugs.python.org/issue?@action=redirect&bpo=45898]: **`ctypes`** no longer defines `ffi_type_*` symbols in `cfield.c`. The symbols have been provided by libffi for over a decade.

- bpo-44953 [https://bugs.python.org/issue?@action=redirect&bpo=44953]: Calling `operator.itemgetter` objects and `operator.attrgetter` objects is now faster due to use of the vectorcall calling convention.

- bpo-44289 [https://bugs.python.org/issue?@action=redirect&bpo=44289]: Fix an issue with **`is_tarfile()`** method when using *fileobj* argument: position in the *fileobj* was advanced forward which made it unreadable with **`tarfile.TarFile.open()`**.

- bpo-44011 [https://bugs.python.org/issue?@action=redirect&bpo=44011]: Reimplement SSL/TLS support in asyncio, borrow the implementation from uvloop library.

- bpo-41086 [https://bugs.python.org/issue?@action=redirect&bpo=41086]: Make the `configparser.ConfigParser` constructor raise `TypeError` if the `interpolation` parameter is not of type **`configparser.Interpolation`**
- bpo-29418 [https://bugs.python.org/issue?@action=redirect&bpo=29418]: Implement `inspect.ismethodwrapper()` and fix `inspect.isroutine()` for cases where methodwrapper is given. Patch by Hakan Çelik.
- bpo-14156 [https://bugs.python.org/issue?@action=redirect&bpo=14156]: argparse.FileType now supports an argument of '-' in binary mode, returning the .buffer attribute of sys.stdin/sys.stdout as appropriate. Modes including 'x' and 'a' are treated equivalently to 'w' when argument is '-'. Patch contributed by Josh Rosenberg

Documentation

- bpo-42238 [https://bugs.python.org/issue?@action=redirect&bpo=42238]: `Doc/tools/rstlint.py` has moved to its own repository and is now packaged on PyPI as `sphinx-lint`.

Tests

- bpo-46913 [https://bugs.python.org/issue?@action=redirect&bpo=46913]: Fix test_faulthandler.test_sigfpe() if Python is built with undefined behavior sanitizer (UBSAN): disable UBSAN on the faulthandler_sigfpe() function. Patch by Victor Stinner.

- bpo-46760 [https://bugs.python.org/issue?@action=redirect&bpo=46760]: Remove bytecode offsets from expected values in test.test_dis module. Reduces the obstacles to modifying the VM or compiler.
- bpo-46708 [https://bugs.python.org/issue?@action=redirect&bpo=46708]: Prevent default asyncio event loop policy modification warning after `test_asyncio` execution.
- bpo-46678 [https://bugs.python.org/issue?@action=redirect&bpo=46678]: The function `make_legacy_pyc` in `Lib/test/support/import_helper.py` no longer fails when `PYTHONPYCACHEPREFIX` is set to a directory on a different device from where tempfiles are stored.
- bpo-46623 [https://bugs.python.org/issue?@action=redirect&bpo=46623]: Skip test_pair() and test_speech128() of test_zlib on s390x since they fail if zlib uses the s390x hardware accelerator. Patch by Victor Stinner.

Build

- bpo-46860 [https://bugs.python.org/issue?@action=redirect&bpo=46860]: Respect `--with-suffix` when building on case-insensitive file systems.
- bpo-46656 [https://bugs.python.org/issue?@action=redirect&bpo=46656]: Building Python now requires a C11 compiler. Optional C11 features are not required. Patch by Victor Stinner.
- bpo-46656 [https://bugs.python.org/issue?@action=redirect&bpo=46656]: Building Python now requires support for floating-point Not-a-Number (NaN): remove the `Py_NO_NAN` macro. Patch by Victor Stinner.
- bpo-46640 [https://bugs.python.org/issue?@action=redirect&bpo=46640]: Building Python now requires a

C99 `<math.h>` header file providing a `NAN` constant, or the `__builtin_nan()` built-in function. Patch by Victor Stinner.

- bpo-46608 [https://bugs.python.org/issue?@action=redirect&bpo=46608]: Exclude marshalled-frozen data if deep-freezing to save 300 KB disk space. This includes adding a new `is_package` field to `_frozen`. Patch by Kumar Aditya.
- bpo-40280 [https://bugs.python.org/issue?@action=redirect&bpo=40280]: Fix wasm32-emscripten test failures and platform issues. - Disable syscalls that are not supported or don't work, e.g. wait, getrusage, prlimit, mkfifo, mknod, setres[gu]id, setgroups. - Use fd_count to count open fds. - Add more checks for subprocess and fork. - Add workarounds for missing _multiprocessing and failing socket.accept(). - Enable bzip2. - Disable large file support. - Disable signal.alarm.
- bpo-46430 [https://bugs.python.org/issue?@action=redirect&bpo=46430]: Intern strings in deep-frozen modules. Patch by Kumar Aditya.

Windows

- bpo-46744 [https://bugs.python.org/issue?@action=redirect&bpo=46744]: The default all users install directory for ARM64 is now under the native `Program Files` folder, rather than `Program Files (Arm)` which is intended for ARM (32-bit) files.
- bpo-46567 [https://bugs.python.org/issue?@action=redirect&bpo=46567]: Adds Tcl and Tk support for Windows ARM64. This also adds IDLE to the installation.
- bpo-46638 [https://bugs.python.org/issue?@action=redirect&bpo=46638]: Ensures registry virtualization is consistently disabled. For 3.10 and earlier, it remains enabled (some registry writes are protected), while for 3.11 and later it is disabled (registry modifications affect all applications).

IDLE

- bpo-46630 [https://bugs.python.org/issue?@action=redirect&bpo=46630]: Make query dialogs on Windows start with a cursor in the entry box.
- bpo-45447 [https://bugs.python.org/issue?@action=redirect&bpo=45447]: Apply IDLE syntax highlighting to `.pyi` files. Patch by Alex Waygood and Terry Jan Reedy.

C API

- bpo-46748 [https://bugs.python.org/issue?@action=redirect&bpo=46748]: Python’s public headers no longer import `<stdbool.h>`, leaving code that embeds/extends Python free to define `bool`, `true` and `false`.
- bpo-46836 [https://bugs.python.org/issue?@action=redirect&bpo=46836]: Move the **`PyFrameObject`** type definition (`struct _frame`) to the internal C API `pycore_frame.h` header file. Patch by Victor Stinner.
- bpo-45459 [https://bugs.python.org/issue?@action=redirect&bpo=45459]: Rename `Include/buffer.h` header file to `Include/pybuffer.h` to avoid conflicts with projects having an existing `buffer.h` header file. Patch by Victor Stinner.
- bpo-45412 [https://bugs.python.org/issue?@action=redirect&bpo=45412]: Remove the `HAVE_PY_SET_53BIT_PRECISION` macro (moved to the internal C API). Patch by Victor Stinner.
- bpo-46613 [https://bugs.python.org/issue?@action=redirect&bpo=46613]: Added function **`PyType_GetModuleByDef()`**, which allows access to module state when a method’s defining class is not available.

Python 3.11.0 alpha 5

Release date: 2022-02-03

Core and Builtins

- bpo-45773 [https://bugs.python.org/issue?@action=redirect&bpo=45773]: Remove two invalid "peephole" optimizations from the bytecode compiler.

- bpo-46564 [https://bugs.python.org/issue?@action=redirect&bpo=46564]: Do not create frame objects when creating **`super`** object. Patch by Kumar Aditya.

- bpo-45885 [https://bugs.python.org/issue?@action=redirect&bpo=45885]: Added more fined-grained specialization failure stats regarding the `COMPARE_OP` bytecode.

- bpo-44977 [https://bugs.python.org/issue?@action=redirect&bpo=44977]: The delegation of **`int()`** to **`__trunc__()`** is now deprecated. Calling `int(a)` when `type(a)` implements **`__trunc__()`** but not **`__int__()`** or **`__index__()`** now raises a **`DeprecationWarning`**.

- bpo-46458 [https://bugs.python.org/issue?@action=redirect&bpo=46458]: Reorder code emitted by the compiler for a **`try`**-**`except`** block so that the **`else`** block's code immediately follows the **`try`** body (without a jump). This is more optimal for the happy path.

- bpo-46527 [https://bugs.python.org/issue?@action=redirect&bpo=46527]: Allow passing `iterable` as a keyword argument to **`enumerate()`** again. Patch by Jelle Zijlstra.

- bpo-46528 [https://bugs.python.org/issue?@action=redirect&bpo=46528]: Replace several stack manipulation instructions (`DUP_TOP`, `DUP_TOP_TWO`, `ROT_TWO`, `ROT_THREE`, `ROT_FOUR`, and `ROT_N`) with new **`COPY`** and **`SWAP`** instructions.

- bpo-46329 [https://bugs.python.org/issue?@action=redirect&bpo=46329]: Use two or three bytecodes to implement most calls.

  Calls without named arguments are implemented as a sequence of two instructions: `PRECALL; CALL`. Calls with named arguments are implemented as a sequence of three instructions: `PRECALL; KW_NAMES; CALL`. There are two different `PRECALL` instructions: `PRECALL_FUNTION` and `PRECALL_METHOD`. The latter pairs with `LOAD_METHOD`.

  This partition into pre-call and call allows better specialization, and thus better performance ultimately.

  There is no change in semantics.

- bpo-46503 [https://bugs.python.org/issue?@action=redirect&bpo=46503]: Fix an assert when parsing some invalid N escape sequences in f-strings.

- bpo-46431 [https://bugs.python.org/issue?@action=redirect&bpo=46431]: Improve error message on invalid calls to **`BaseExceptionGroup.__new__()`**.

- bpo-46476 [https://bugs.python.org/issue?@action=redirect&bpo=46476]: Fix memory leak in code objects generated by deepfreeze. Patch by Kumar Aditya.

- bpo-46481 [https://bugs.python.org/issue?@action=redirect&bpo=46481]: Speed up calls to **`weakref.ref.__call__()`** by using the **PEP 590** [https://peps.python.org/pep-0590/] `vectorcall` calling convention. Patch by Donghee Na.

- bpo-46417 [https://bugs.python.org/issue?@action=redirect&bpo=46417]: Fix a race condition on setting a type `__bases__` attribute: the internal function `add_subclass()` now gets the `PyTypeObject.tp_subclasses` member after calling `PyWeakref_NewRef()` which can trigger a garbage collection which can indirectly modify `PyTypeObject.tp_subclasses`. Patch by Victor Stinner.

- bpo-46417 [https://bugs.python.org/issue?@action=redirect&bpo=46417]: `python -X showrefcount` now shows the total reference count after clearing and destroyed the main Python interpreter. Previously, it was shown before. Patch by Victor Stinner.

- bpo-43683 [https://bugs.python.org/issue?@action=redirect&bpo=43683]: Add ASYNC_GEN_WRAP opcode to wrap the value to be yielded in async generators. Removes the need to special case async generators in the `YIELD_VALUE` instruction.

- bpo-46407 [https://bugs.python.org/issue?@action=redirect&bpo=46407]: Optimize some modulo operations in `Objects/longobject.c`. Patch by Jeremiah Vivian.

- bpo-46409 [https://bugs.python.org/issue?@action=redirect&bpo=46409]: Add new `RETURN_GENERATOR` bytecode to make generators. Simplifies calling Python functions in the VM, as they no longer any need to special case generator functions.

  Also add `JUMP_NO_INTERRUPT` bytecode that acts like `JUMP_ABSOLUTE`, but does not check for interrupts.

- bpo-46406 [https://bugs.python.org/issue?@action=redirect&bpo=46406]: The integer division `//` implementation has been optimized to better let the compiler understand its constraints. It can be 20% faster on the amd64 platform when dividing an int by a value smaller than `2**30`.

Library

- bpo-46383 [https://bugs.python.org/issue?@action=redirect&bpo=46383]: Fix invalid signature of `_zoneinfo`'s `module_free` function to resolve a crash on wasm32-emscripten platform.

Core and Builtins

- bpo-46361 [https://bugs.python.org/issue?@action=redirect&bpo=46361]: Ensure that "small" integers created by `int.from_bytes()` and `decimal.Decimal` are properly cached.
- bpo-46161 [https://bugs.python.org/issue?@action=redirect&bpo=46161]: Fix the class building error when the arguments are constants and CALL_FUNCTION_EX is used.
- bpo-46028 [https://bugs.python.org/issue?@action=redirect&bpo=46028]: Fixes calculation of **`sys._base_executable`** when inside a virtual environment that uses symlinks with different binary names than the base environment provides.
- bpo-46091 [https://bugs.python.org/issue?@action=redirect&bpo=46091]: Correctly calculate indentation levels for lines with whitespace character that are ended by line continuation characters. Patch by Pablo Galindo

Library

- bpo-30512 [https://bugs.python.org/issue?@action=redirect&bpo=30512]: Add CAN Socket support for NetBSD.

Build

- bpo-46045 [https://bugs.python.org/issue?@action=redirect&bpo=46045]: Do not use POSIX semaphores on NetBSD

Core and Builtins

- bpo-44024 [https://bugs.python.org/issue?@action=redirect&bpo=44024]: Improve the **`TypeError`** message for non-string second arguments passed to the built-in functions **`getattr()`** and **`hasattr()`**. Patch by Géry Ogam.

Library

- bpo-46624 [https://bugs.python.org/issue?@action=redirect&bpo=46624]: Restore support for non-integer arguments of **`random.randrange()`** and **`random.randint()`**.
- bpo-46591 [https://bugs.python.org/issue?@action=redirect&bpo=46591]: Make the IDLE doc URL on the About IDLE dialog clickable.
- bpo-46565 [https://bugs.python.org/issue?@action=redirect&bpo=46565]: Remove loop variables that are leaking into modules’ namespaces.
- bpo-46553 [https://bugs.python.org/issue?@action=redirect&bpo=46553]: In **`typing.get_type_hints()`**, support evaluating bare stringified `ClassVar` annotations. Patch by Gregory Beauregard.
- bpo-46544 [https://bugs.python.org/issue?@action=redirect&bpo=46544]: Don’t leak `x` & `uspace` intermediate vars in **`textwrap.TextWrapper`**.
- bpo-46487 [https://bugs.python.org/issue?@action=redirect&bpo=46487]: Add the `get_write_buffer_limits` method to **`asyncio.transports.WriteTransport`** and to the SSL transport.

- bpo-45173 [https://bugs.python.org/issue?@action=redirect&bpo=45173]: Note the configparser deprecations will be removed in Python 3.12.
- bpo-45162 [https://bugs.python.org/issue?@action=redirect&bpo=45162]: The deprecated `unittest` APIs removed in 3.11a1 have been temporarily restored to be removed in 3.12 while cleanups in external projects go in.
- bpo-46539 [https://bugs.python.org/issue?@action=redirect&bpo=46539]: In `typing.get_type_hints()`, support evaluating stringified `ClassVar` and `Final` annotations inside `Annotated`. Patch by Gregory Beauregard.
- bpo-46510 [https://bugs.python.org/issue?@action=redirect&bpo=46510]: Add missing test for `types.TracebackType` and `types.FrameType`. Calculate them directly from the caught exception without calling `sys.exc_info()`.
- bpo-46491 [https://bugs.python.org/issue?@action=redirect&bpo=46491]: Allow `typing.Annotated` to wrap `typing.Final` and `typing.ClassVar`. Patch by Gregory Beauregard.
- bpo-46483 [https://bugs.python.org/issue?@action=redirect&bpo=46483]: Remove `__class_getitem__()` from `pathlib.PurePath` as this class was not supposed to be generic.
- bpo-46436 [https://bugs.python.org/issue?@action=redirect&bpo=46436]: Fix command-line option `-d/--directory` in module `http.server` which is ignored when combined with command-line option `--cgi`. Patch by Géry Ogam.
- bpo-41403 [https://bugs.python.org/issue?@action=redirect&bpo=41403]: Make **`mock.patch()`** raise a `TypeError` with a relevant error message on invalid arg. Previously it allowed a cryptic `AttributeError` to escape.

- bpo-46474 [https://bugs.python.org/issue?@action=redirect&bpo=46474]: In `importlib.metadata.EntryPoint.pattern`, avoid potential REDoS by limiting ambiguity in consecutive whitespace.
- bpo-46474 [https://bugs.python.org/issue?@action=redirect&bpo=46474]: Removed private method from `importlib.metadata.Path`. Sync with importlib_metadata 4.10.0.
- bpo-46470 [https://bugs.python.org/issue?@action=redirect&bpo=46470]: Remove unused branch from `typing._remove_dups_flatten`
- bpo-46469 [https://bugs.python.org/issue?@action=redirect&bpo=46469]: **`asyncio`** generic classes now return **`types.GenericAlias`** in `__class_getitem__` instead of the same class.
- bpo-41906 [https://bugs.python.org/issue?@action=redirect&bpo=41906]: Support passing filter instances in the `filters` values of `handlers` and `loggers` in the dictionary passed to **`logging.config.dictConfig()`**.
- bpo-46422 [https://bugs.python.org/issue?@action=redirect&bpo=46422]: Use `dis.Positions` in `dis.Instruction` instead of a regular `tuple`.
- bpo-46434 [https://bugs.python.org/issue?@action=redirect&bpo=46434]: **`pdb`** now gracefully handles `help` when **`__doc__`** is missing, for example when run with pregenerated optimized `.pyc` files.
- bpo-43869 [https://bugs.python.org/issue?@action=redirect&bpo=43869]: Python uses the same time Epoch on all platforms. Add an explicit unit test to ensure that it’s the case. Patch by Victor Stinner.
- bpo-46414 [https://bugs.python.org/issue?@action=redirect&bpo=46414]: Add **`typing.reveal_type()`**. Patch by Jelle Zijlstra.
- bpo-40280 [https://bugs.python.org/issue?@action=redirect&bpo=40280]: **`subprocess`** now imports Windows-specific imports when `msvcrt` module is

available, and POSIX-specific imports on all other platforms. This gives a clean exception when `_posixsubprocess` is not available (e.g. Emscripten browser target).

- bpo-40066 [https://bugs.python.org/issue?@action=redirect&bpo=40066]: `IntEnum`, `IntFlag`, and `StrEnum` use the mixed-in type for their `str()` and `format()` output.
- bpo-46316 [https://bugs.python.org/issue?@action=redirect&bpo=46316]: Optimize `pathlib.Path.iterdir()` by removing an unnecessary check for special entries.
- bpo-29688 [https://bugs.python.org/issue?@action=redirect&bpo=29688]: Document `pathlib.Path.absolute()` (which has always existed).
- bpo-43012 [https://bugs.python.org/issue?@action=redirect&bpo=43012]: The pathlib module’s obsolete and internal `_Accessor` class has been removed to prepare the terrain for upcoming enhancements to the module.
- bpo-46258 [https://bugs.python.org/issue?@action=redirect&bpo=46258]: Speed up `math.isqrt()` for small positive integers by replacing two division steps with a lookup table.
- bpo-46242 [https://bugs.python.org/issue?@action=redirect&bpo=46242]: Improve error message when creating a new `enum.Enum` type subclassing an existing `Enum` with `_member_names_` using **`enum.Enum.__call__()`**.
- bpo-43118 [https://bugs.python.org/issue?@action=redirect&bpo=43118]: Fix a bug in `inspect.signature()` that was causing it to fail on some subclasses of classes with a `__text_signature__` referencing module globals. Patch by Weipeng Hong.
- bpo-26552 [https://bugs.python.org/issue?@action=redirect&bpo=26552]: Fixed case where failing `asyncio.ensure_future()` did not close the coroutine. Patch by Kumar Aditya.

- bpo-21987 [https://bugs.python.org/issue?@action=redirect&bpo=21987]: Fix an issue with `tarfile.TarFile.getmember()` getting a directory name with a trailing slash.
- bpo-46124 [https://bugs.python.org/issue?@action=redirect&bpo=46124]: Update `zoneinfo` to rely on importlib.resources traversable API.
- bpo-46103 [https://bugs.python.org/issue?@action=redirect&bpo=46103]: Now `inspect.getmembers()` only gets `__bases__` attribute from class type. Patch by Weipeng Hong.
- bpo-46080 [https://bugs.python.org/issue?@action=redirect&bpo=46080]: Fix exception in argparse help text generation if a `argparse.BooleanOptionalAction` argument's default is `argparse.SUPPRESS` and it has `help` specified. Patch by Felix Fontein.
- bpo-44791 [https://bugs.python.org/issue?@action=redirect&bpo=44791]: Fix substitution of `ParamSpec` in `Concatenate` with different parameter expressions. Substitution with a list of types returns now a tuple of types. Substitution with `Concatenate` returns now a `Concatenate` with concatenated lists of arguments.

Documentation

- bpo-46463 [https://bugs.python.org/issue?@action=redirect&bpo=46463]: Fixes `escape4chm.py` script used when building the CHM documentation file

Tests

- bpo-43478 [https://bugs.python.org/issue?@action=redirect&bpo=43478]: Mocks can no longer be provided as the specs for other Mocks. As a result, an already-mocked object cannot be passed to `mock.Mock()`. This can uncover bugs in tests since these Mock-derived

Mocks will always pass certain tests (e.g. isinstance) and builtin assert functions (e.g. assert_called_once_with) will unconditionally pass.

- bpo-46616 [https://bugs.python.org/issue?@action=redirect&bpo=46616]: Ensures `test_importlib.test_windows` cleans up registry keys after completion.
- bpo-44359 [https://bugs.python.org/issue?@action=redirect&bpo=44359]: test_ftplib now silently ignores socket errors to prevent logging unhandled threading exceptions. Patch by Victor Stinner.
- bpo-46600 [https://bugs.python.org/issue?@action=redirect&bpo=46600]: Fix test_gdb.test_pycfunction() for Python built with `clang -Og`. Tolerate inlined functions in the gdb traceback. Patch by Victor Stinner.
- bpo-46542 [https://bugs.python.org/issue?@action=redirect&bpo=46542]: Fix a Python crash in test_lib2to3 when using Python built in debug mode: limit the recursion limit. Patch by Victor Stinner.
- bpo-46576 [https://bugs.python.org/issue?@action=redirect&bpo=46576]: test_peg_generator now disables compiler optimization when testing compilation of its own C extensions to significantly speed up the testing on non-debug builds of CPython.
- bpo-46542 [https://bugs.python.org/issue?@action=redirect&bpo=46542]: Fix `test_json` tests checking for **`RecursionError`**: modify these tests to use `support.infinite_recursion()`. Patch by Victor Stinner.
- bpo-13886 [https://bugs.python.org/issue?@action=redirect&bpo=13886]: Skip test_builtin PTY tests on non-ASCII characters if the readline module is loaded. The readline module changes input() behavior, but test_builtin is not intended to test the readline module. Patch by Victor Stinner.
- bpo-40280 [https://bugs.python.org/issue?@action=redirect&bpo=40280]: Add **`test.support.requires_fork()`**

decorators to mark tests that require a working `os.fork()`.

- bpo-40280 [https://bugs.python.org/issue?@action=redirect&bpo=40280]: Add **`test.support.requires_subprocess()`** decorator to mark tests which require working `subprocess` module or `os.spawn*`. The wasm32-emscripten platform has no support for processes.
- bpo-46126 [https://bugs.python.org/issue?@action=redirect&bpo=46126]: Disable ‘descriptions’ when running tests internally.

Build

- bpo-46602 [https://bugs.python.org/issue?@action=redirect&bpo=46602]: Tidied up configure.ac so that conftest.c is truncated rather than appended. This assists in the case where the ‘rm’ of conftest.c fails to happen between tests. Downstream issues such as a clobbered SOABI can result.
- bpo-46600 [https://bugs.python.org/issue?@action=redirect&bpo=46600]: Fix the test checking if the C compiler supports `-Og` option in the `./configure` script to also use `-Og` on clang which supports it. Patch by Victor Stinner.
- bpo-38472 [https://bugs.python.org/issue?@action=redirect&bpo=38472]: Fix GCC detection in setup.py when cross-compiling. The C compiler is now run with LC_ALL=C. Previously, the detection failed with a German locale.
- bpo-46513 [https://bugs.python.org/issue?@action=redirect&bpo=46513]: **configure** no longer uses `AC_C_CHAR_UNSIGNED` macro and `pyconfig.h` no longer defines reserved symbol `__CHAR_UNSIGNED__`.
- bpo-46471 [https://bugs.python.org/issue?@action=redirect&bpo=46471]: Use global singletons for single

byte bytes objects in deepfreeze.

- bpo-46443 [https://bugs.python.org/issue?@action=redirect&bpo=46443]: Deepfreeze now uses cached small integers as it saves some space for common small integers.
- bpo-46429 [https://bugs.python.org/issue?@action=redirect&bpo=46429]: Merge all deep-frozen files into one for space savings. Patch by Kumar Aditya.
- bpo-45569 [https://bugs.python.org/issue?@action=redirect&bpo=45569]: The build now defaults to using 30-bit digits for Python integers. Previously either 15-bit or 30-bit digits would be selected, depending on the platform. 15-bit digits may still be selected using the `--enable-big-digits=15` option to the `configure` script, or by defining `PYLONG_BITS_IN_DIGIT` in `pyconfig.h`.
- bpo-45925 [https://bugs.python.org/issue?@action=redirect&bpo=45925]: Update Windows installer to use SQLite 3.37.2.
- bpo-43112 [https://bugs.python.org/issue?@action=redirect&bpo=43112]: Detect musl libc as a separate SOABI (tagged as `linux-musl`).

Windows

- bpo-33125 [https://bugs.python.org/issue?@action=redirect&bpo=33125]: The traditional EXE/MSI based installer for Windows is now available for ARM64
- bpo-46362 [https://bugs.python.org/issue?@action=redirect&bpo=46362]: os.path.abspath(“C:CON”) is now fixed to return “\.CON”, not the same path. The regression was true of all legacy DOS devices such as COM1, LPT1, or NUL.
- bpo-44934 [https://bugs.python.org/issue?@action=redirect&bpo=44934]: The installer now offers a command-line only option to add the installation directory to the end of **`PATH`** instead of at the start.

macOS

- bpo-45925 [https://bugs.python.org/issue?@action=redirect&bpo=45925]: Update macOS installer to SQLite 3.37.2.

IDLE

- bpo-45296 [https://bugs.python.org/issue?@action=redirect&bpo=45296]: Clarify close, quit, and exit in IDLE. In the File menu, ‘Close’ and ‘Exit’ are now ‘Close Window’ (the current one) and ‘Exit’ is now ‘Exit IDLE’ (by closing all windows). In Shell, ‘quit()’ and ‘exit()’ mean ‘close Shell’. If there are no other windows, this also exits IDLE.

C API

- bpo-40170 [https://bugs.python.org/issue?@action=redirect&bpo=40170]: Remove the `PyHeapType_GET_MEMBERS()` macro. It was exposed in the public C API by mistake, it must only be used by Python internally. Use the `PyTypeObject.tp_members` member instead. Patch by Victor Stinner.
- bpo-40170 [https://bugs.python.org/issue?@action=redirect&bpo=40170]: Move _Py_GetAllocatedBlocks() and _PyObject_DebugMallocStats() private functions to the internal C API. Patch by Victor Stinner.
- bpo-46433 [https://bugs.python.org/issue?@action=redirect&bpo=46433]: The internal function _PyType_GetModuleByDef now correctly handles inheritance patterns involving static types.
- bpo-45459 [https://bugs.python.org/issue?@action=redirect&bpo=45459]: **`Py_buffer`** and various `Py_buffer`

related functions are now part of the limited API and stable ABI.

- bpo-14916 [https://bugs.python.org/issue?@action=redirect&bpo=14916]: Fixed bug in the tokenizer that prevented `PyRun_InteractiveOne` from parsing from the provided FD.

Python 3.11.0 alpha 4

Release date: 2022-01-13

Core and Builtins

- bpo-46070 [https://bugs.python.org/issue?@action=redirect&bpo=46070]: **`Py_EndInterpreter()`** now explicitly untracks all objects currently tracked by the GC. Previously, if an object was used later by another interpreter, calling **`PyObject_GC_UnTrack()`** on the object crashed if the previous or the next object of the **`PyGC_Head`** structure became a dangling pointer. Patch by Victor Stinner.
- bpo-46347 [https://bugs.python.org/issue?@action=redirect&bpo=46347]: Fix memory leak in PyEval_EvalCodeEx.
- bpo-46339 [https://bugs.python.org/issue?@action=redirect&bpo=46339]: Fix a crash in the parser when retrieving the error text for multi-line f-strings expressions that do not start in the first line of the string. Patch by Pablo Galindo
- bpo-46331 [https://bugs.python.org/issue?@action=redirect&bpo=46331]: Do not set line number of instruction storing doc-string. Fixes regression introduced in 3.11 alpha.
- bpo-46314 [https://bugs.python.org/issue?@action=redirect&bpo=46314]: Remove spurious “call” event

when creating a lambda function that was accidentally introduced in 3.11a4.

- bpo-46289 [https://bugs.python.org/issue?@action=redirect&bpo=46289]: ASDL declaration of `FormattedValue` has changed to reflect `conversion` field is not optional.
- bpo-46297 [https://bugs.python.org/issue?@action=redirect&bpo=46297]: Fixed an interpreter crash on bootup with multiple PythonPaths set in the Windows registry. Patch by Derzsi Dániel.
- bpo-46237 [https://bugs.python.org/issue?@action=redirect&bpo=46237]: Fix the line number of tokenizer errors inside f-strings. Patch by Pablo Galindo.
- bpo-46263 [https://bugs.python.org/issue?@action=redirect&bpo=46263]: We always expect the “use_frozen_modules” config to be set, now that getpath.c was rewritten in pure Python and the logic improved.
- bpo-46006 [https://bugs.python.org/issue?@action=redirect&bpo=46006]: Fix a regression when a type method like `__init__()` is modified in a subinterpreter. Fix a regression in `_PyUnicode_EqualToASCIIId()` and type `update_slot()`. Revert the change which made the Unicode dictionary of interned strings compatible with subinterpreters: the internal interned dictionary is shared again by all interpreters. Patch by Victor Stinner.
- bpo-45923 [https://bugs.python.org/issue?@action=redirect&bpo=45923]: Add RESUME opcode. This is a logical no-op. It is emitted by the compiler anywhere a Python function can be entered. It is used by the interpreter to perform tracing and optimizer checks.

Library

- bpo-46208 [https://bugs.python.org/issue?@action=redirect&bpo=46208]: Fix the regression of os.path.normpath("A/../../B") not returning expected "../B" but "B".

Core and Builtins

- bpo-46240 [https://bugs.python.org/issue?@action=redirect&bpo=46240]: Correct the error message for unclosed parentheses when the tokenizer doesn't reach the end of the source when the error is reported. Patch by Pablo Galindo
- bpo-46009 [https://bugs.python.org/issue?@action=redirect&bpo=46009]: Remove the `GEN_START` opcode.
- bpo-46235 [https://bugs.python.org/issue?@action=redirect&bpo=46235]: Certain sequence multiplication operations like `[0] * 1_000` are now faster due to reference-counting optimizations. Patch by Dennis Sweeney.
- bpo-46221 [https://bugs.python.org/issue?@action=redirect&bpo=46221]: **`PREP_RERAISE_STAR`** no longer pushes `lasti` to the stack.
- bpo-46202 [https://bugs.python.org/issue?@action=redirect&bpo=46202]: Remove **`POP_EXCEPT_AND_RERAISE`** and replace it by an equivalent sequence of other opcodes.

Library

- bpo-46085 [https://bugs.python.org/issue?@action=redirect&bpo=46085]: Fix iterator cache mechanism of **`OrderedDict`**.

Core and Builtins

- bpo-46055 [https://bugs.python.org/issue?@action=redirect&bpo=46055]: Speed up shifting operation involving integers less than **`PyLong_BASE`**. Patch by Xinhang Xu.

- bpo-46110 [https://bugs.python.org/issue?@action=redirect&bpo=46110]: Add a maximum recursion check to the PEG parser to avoid stack overflow. Patch by Pablo Galindo

- bpo-46107 [https://bugs.python.org/issue?@action=redirect&bpo=46107]: Fix bug where **`ExceptionGroup.split()`** and **`ExceptionGroup.subgroup()`** did not copy the exception group's `__note__` field to the parts.

- bpo-45711 [https://bugs.python.org/issue?@action=redirect&bpo=45711]: The interpreter state's representation of handled exceptions (a.k.a exc_info, or _PyErr_StackItem) now has only the `exc_value` field, `exc_type` and `exc_traceback` have been removed as their values can be derived from `exc_value`.

- bpo-44525 [https://bugs.python.org/issue?@action=redirect&bpo=44525]: Replace the four call bytecode instructions which one pre-call instruction and two call instructions.

  Removes `CALL_FUNCTION`, `CALL_FUNCTION_KW`, `CALL_METHOD` and `CALL_METHOD_KW`.

  Adds `CALL_NO_KW` and `CALL_KW` call instructions, and `PRECALL_METHOD` prefix for pairing with `LOAD_METHOD`.

- bpo-46039 [https://bugs.python.org/issue?@action=redirect&bpo=46039]: Remove the `YIELD_FROM` instruction and replace it with the `SEND` instruction which performs the same operation, but without the loop.

- bpo-45635 [https://bugs.python.org/issue?@action=redirect&bpo=45635]: The code called from **`_PyErr_Display()`** was refactored to improve error handling. It now exits immediately upon an unrecoverable error.

- bpo-46054 [https://bugs.python.org/issue?@action=redirect&bpo=46054]: Fix parser error when parsing non-utf8 characters in source files. Patch by Pablo Galindo.

- bpo-46042 [https://bugs.python.org/issue?@action=redirect&bpo=46042]: Improve the location of the caret in **`SyntaxError`** exceptions emitted by the symbol table. Patch by Pablo Galindo.

- bpo-46049 [https://bugs.python.org/issue?@action=redirect&bpo=46049]: Ensure `._pth` files work as intended on platforms other than Windows.

- bpo-46048 [https://bugs.python.org/issue?@action=redirect&bpo=46048]: Fixes parsing of `._pth` files on startup so that single-character paths are correctly read.

- bpo-37971 [https://bugs.python.org/issue?@action=redirect&bpo=37971]: Fix a bug where the line numbers given in a traceback when a decorator application raised an exception were wrong.

- bpo-46031 [https://bugs.python.org/issue?@action=redirect&bpo=46031]: Add **`POP_JUMP_IF_NOT_NONE`** and **`POP_JUMP_IF_NONE`** opcodes to speed up conditional jumps.

- bpo-45654 [https://bugs.python.org/issue?@action=redirect&bpo=45654]: Deepfreeze **`runpy`**, patch by Kumar Aditya.

- bpo-46025 [https://bugs.python.org/issue?@action=redirect&bpo=46025]: Fix a crash in the **`atexit`** module involving functions that unregister themselves before raising exceptions. Patch by Pablo Galindo.

Library

- bpo-46000 [https://bugs.python.org/issue?@action=redirect&bpo=46000]: Improve compatibility of the **`curses`** module with NetBSD curses.

Core and Builtins

- bpo-44525 [https://bugs.python.org/issue?@action=redirect&bpo=44525]: Specialize the CALL_FUNCTION instruction for calls to builtin types with a single argument. Speeds up `range(x)`, `list(x)`, and specifically `type(obj)`.
- bpo-42918 [https://bugs.python.org/issue?@action=redirect&bpo=42918]: Fix bug where the built-in **`compile()`** function did not always raise a **`SyntaxError`** when passed multiple statements in ‘single’ mode. Patch by Weipeng Hong.
- bpo-45953 [https://bugs.python.org/issue?@action=redirect&bpo=45953]: The main interpreter in _PyRuntimeState.interpreters is now statically allocated (as part of _PyRuntime). Likewise for the initial thread state of each interpreter. This means less allocation during runtime init, as well as better memory locality for these key state objects.
- bpo-45292 [https://bugs.python.org/issue?@action=redirect&bpo=45292]: Complete the **PEP 654** [https://peps.python.org/pep-0654/] implementation: add `except*`.
- bpo-43413 [https://bugs.python.org/issue?@action=redirect&bpo=43413]: Revert changes in `set.__init__`. Subclass of **`set`** needs to define a `__init__()` method if it

defines a `__new__()` method with additional keyword parameters.
- bpo-43931 [https://bugs.python.org/issue?@action=redirect&bpo=43931]: Added the **`Py_Version`** constant which bears the same value as **`PY_VERSION_HEX`**. Patch by Gabriele N. Tornetta.

Library

- bpo-46342 [https://bugs.python.org/issue?@action=redirect&bpo=46342]: The `@typing.final` decorator now sets the `__final__` attribute on the decorated object to allow runtime introspection. Patch by Jelle Zijlstra.

- bpo-46328 [https://bugs.python.org/issue?@action=redirect&bpo=46328]: Added the **`sys.exception()`** method which returns the active exception instance.

- bpo-46307 [https://bugs.python.org/issue?@action=redirect&bpo=46307]: Add **`string.Template.is_valid()`** and **`string.Template.get_identifiers()`** methods.

- bpo-46306 [https://bugs.python.org/issue?@action=redirect&bpo=46306]: Assume that **`types.CodeType`** always has **`types.CodeType.co_firstlineno`** in **`doctest`**.

- bpo-40479 [https://bugs.python.org/issue?@action=redirect&bpo=40479]: Fix **`hashlib`** *usedforsecurity* option to work correctly with OpenSSL 3.0.0 in FIPS mode.

- bpo-46070 [https://bugs.python.org/issue?@action=redirect&bpo=46070]: Fix possible segfault when importing the **`asyncio`** module from different sub-interpreters in parallel. Patch by Erlend E. Aasland.

- bpo-46244 [https://bugs.python.org/issue?@action=redirect&bpo=46244]: Removed `__slots__` from **`typing.ParamSpec`** and **`typing.TypeVar`**. They served no purpose. Patch by Arie Bovenberg.

- bpo-46278 [https://bugs.python.org/issue?@action=redirect&bpo=46278]: Reflect `context` argument in `AbstractEventLoop.call_*()` methods. Loop implementations already support it.

- bpo-46269 [https://bugs.python.org/issue?@action=redirect&bpo=46269]: Remove special-casing of `__new__` in **`enum.Enum.__dir__()`**.

- bpo-46266 [https://bugs.python.org/issue?@action=redirect&bpo=46266]: Improve day constants in **`calendar`**.

  Now all constants (`MONDAY` ... `SUNDAY`) are documented, tested, and added to `__all__`.

- bpo-46257 [https://bugs.python.org/issue?@action=redirect&bpo=46257]: Optimized the mean, variance, and stdev functions in the statistics module. If the input is an iterator, it is consumed in a single pass rather than eating memory by conversion to a list. The single pass algorithm is about twice as fast as the previous two pass code.

- bpo-41011 [https://bugs.python.org/issue?@action=redirect&bpo=41011]: Added two new variables to *pyvenv.cfg* which is generated by **`venv`** module: *executable* for the executable and *command* for the command line used to create the environment.

- bpo-46239 [https://bugs.python.org/issue?@action=redirect&bpo=46239]: Improve error message when importing **`asyncio.windows_events`** on non-Windows.

- bpo-46238 [https://bugs.python.org/issue?@action=redirect&bpo=46238]: Reuse `_winapi` constants in `asyncio.windows_events`.
- bpo-46222 [https://bugs.python.org/issue?@action=redirect&bpo=46222]: Adding `SF_NOCACHE` sendfile constant for FreeBSD for the posixmodule.
- bpo-37295 [https://bugs.python.org/issue?@action=redirect&bpo=37295]: Add fast path for `0 <= k <= n <= 67` for **`math.comb()`**.
- bpo-46176 [https://bugs.python.org/issue?@action=redirect&bpo=46176]: Adding the `MAP_STACK` constant for the mmap module.
- bpo-43424 [https://bugs.python.org/issue?@action=redirect&bpo=43424]: Deprecate **`webbrowser.MacOSXOSAScript._name`** and use `name` instead.
- bpo-45321 [https://bugs.python.org/issue?@action=redirect&bpo=45321]: Added missing error codes to module `xml.parsers.expat.errors`.
- bpo-46125 [https://bugs.python.org/issue?@action=redirect&bpo=46125]: Refactor tests to test traversable API directly. Includes changes from importlib 5.4.0.
- bpo-46118 [https://bugs.python.org/issue?@action=redirect&bpo=46118]: Moved importlib.resources and its related functionality to a package.
- bpo-37578 [https://bugs.python.org/issue?@action=redirect&bpo=37578]: Add *include_hidden* parameter

to `glob()` and `iglob()` to match hidden files and directories when using special characters like `*`, `**`, `?` and `[]`.

- bpo-20369 [https://bugs.python.org/issue?@action=redirect&bpo=20369]: `concurrent.futures.wait()` no longer blocks forever when given duplicate Futures. Patch by Kumar Aditya.

- bpo-46105 [https://bugs.python.org/issue?@action=redirect&bpo=46105]: Honor spec when generating requirement specs with urls and extras (importlib_metadata 4.8.3).

- bpo-44893 [https://bugs.python.org/issue?@action=redirect&bpo=44893]: EntryPoint objects are no longer tuples. Recommended means to access is by attribute (‘.name’, ‘.group’) or accessor (‘.load()’). Access by index is deprecated and will raise deprecation warning.

- bpo-22815 [https://bugs.python.org/issue?@action=redirect&bpo=22815]: Print unexpected successes together with failures and errors in summary in `unittest.TextTestResult`.

- bpo-22047 [https://bugs.python.org/issue?@action=redirect&bpo=22047]: Calling **`add_argument_group()`** on an argument group is deprecated. Calling **`add_argument_group()`** or **`add_mutually_exclusive_group()`** on a mutually exclusive group is deprecated.

  These features were never supported and do not always work correctly. The functions exist on the API by accident through inheritance and will be removed in the future.

- bpo-26952 [https://bugs.python.org/issue?@action=redirect&bpo=26952]: **`argparse`** raises **`ValueError`** with clear message when trying to render usage for an empty mutually exclusive group. Previously it raised a cryptic **`IndexError`**.

- bpo-45615 [https://bugs.python.org/issue?@action=redirect&bpo=45615]: Functions in the **`traceback`** module raise **`TypeError`** rather than **`AttributeError`** when an exception argument is not of type **`BaseException`**.

- bpo-16594 [https://bugs.python.org/issue?@action=redirect&bpo=16594]: Add allow allow_reuse_port flag in socketserver.

- bpo-27718 [https://bugs.python.org/issue?@action=redirect&bpo=27718]: Fix help for the **`signal`** module. Some functions (e.g. `signal()` and `getsignal()`) were omitted.

- bpo-46032 [https://bugs.python.org/issue?@action=redirect&bpo=46032]: The `registry()` method of **`functools.singledispatch()`** functions checks now the first argument or the first parameter annotation and raises a TypeError if it is not supported. Previously unsupported “types” were ignored (e.g. `typing.List[int]`) or caused an error at calling time (e.g. `list[int]`).

- bpo-46014 [https://bugs.python.org/issue?@action=redirect&bpo=46014]: Add ability to use `typing.Union` and `types.UnionType` as dispatch argument to `functools.singledispatch`. Patch provided by Yurii Karabas.

- bpo-27062 [https://bugs.python.org/issue?@action=redirect&bpo=27062]: Add **`__all__`** to **`inspect`**, patch by Kumar Aditya.

- bpo-46018 [https://bugs.python.org/issue?@action=redirect&bpo=46018]: Ensure that `math.expm1()` does not raise on underflow.

- bpo-46016 [https://bugs.python.org/issue?@action=redirect&bpo=46016]: Adding **`F_DUP2FD`** and **`F_DUP2FD_CLOEXEC`** constants from FreeBSD into the `fcntl` module.

- bpo-45755 [https://bugs.python.org/issue?@action=redirect&bpo=45755]: `typing` generic aliases now reveal the class attributes of the original generic class when passed to `dir()`. This was the behavior up to Python 3.6, but was changed in 3.7-3.9.

- bpo-45874 [https://bugs.python.org/issue?@action=redirect&bpo=45874]: The empty query string, consisting of no query arguments, is now handled correctly in `urllib.parse.parse_qsl`. This caused problems before when strict parsing was enabled.

- bpo-44674 [https://bugs.python.org/issue?@action=redirect&bpo=44674]: Change how dataclasses disallows mutable default values. It used to use a list of known types (list, dict, set). Now it disallows unhashable objects to be defaults. It’s using unhashability as a proxy for mutability. Patch by Eric V. Smith, idea by Raymond Hettinger.

- bpo-23882 [https://bugs.python.org/issue?@action=redirect&bpo=23882]: Remove namespace package (PEP 420) support from unittest discovery. It was introduced in Python 3.4 but has been broken since Python 3.7.

- bpo-25066 [https://bugs.python.org/issue?@action=redirect&bpo=25066]: Added a **`__repr__()`** method to

`multiprocessing.Event` objects, patch by Kumar Aditya.

- bpo-45643 [https://bugs.python.org/issue?@action=redirect&bpo=45643]: Added **`signal.SIGSTKFLT`** on platforms where this signal is defined.

- bpo-44092 [https://bugs.python.org/issue?@action=redirect&bpo=44092]: Fetch across rollback no longer raises **`InterfaceError`**. Instead we leave it to the SQLite library to handle these cases. Patch by Erlend E. Aasland.

- bpo-42413 [https://bugs.python.org/issue?@action=redirect&bpo=42413]: Replace `concurrent.futures.TimeoutError` and `asyncio.TimeoutError` with builtin **`TimeoutError`**, keep these names as deprecated aliases.

Documentation

- bpo-46196 [https://bugs.python.org/issue?@action=redirect&bpo=46196]: Document method **`cmd.Cmd.columnize()`**.
- bpo-46120 [https://bugs.python.org/issue?@action=redirect&bpo=46120]: State that `|` is preferred for readability over `Union` in the **`typing`** docs.
- bpo-46109 [https://bugs.python.org/issue?@action=redirect&bpo=46109]: Extracted `importlib.resources` and `importlib.resources.abc` documentation into separate files.
- bpo-19737 [https://bugs.python.org/issue?@action=redirect&bpo=19737]: Update the documentation for the **`globals()`** function.

Tests

- bpo-46296 [https://bugs.python.org/issue?@action=redirect&bpo=46296]: Add a test case for **`enum`** with `_use_args_ == True` and `_member_type_ == object`.
- bpo-46205 [https://bugs.python.org/issue?@action=redirect&bpo=46205]: Fix hang in runtest_mp due to race condition
- bpo-46263 [https://bugs.python.org/issue?@action=redirect&bpo=46263]: Fix test_capi on FreeBSD 14-dev: instruct jemalloc to not fill freed memory with junk byte.
- bpo-46262 [https://bugs.python.org/issue?@action=redirect&bpo=46262]: Cover `ValueError` path in tests for **`enum.Flag._missing_()`**.
- bpo-46150 [https://bugs.python.org/issue?@action=redirect&bpo=46150]: Now `fakename` in `test_pathlib.PosixPathTest.test_expanduser` is checked to be non-existent.
- bpo-46129 [https://bugs.python.org/issue?@action=redirect&bpo=46129]: Rewrite `asyncio.locks` tests with **`unittest.IsolatedAsyncioTestCase`** usage.
- bpo-23819 [https://bugs.python.org/issue?@action=redirect&bpo=23819]: Fixed **`asyncio`** tests in python optimized mode. Patch by Kumar Aditya.
- bpo-46114 [https://bugs.python.org/issue?@action=redirect&bpo=46114]: Fix test case for OpenSSL 3.0.1 version. OpenSSL 3.0 uses `0xMNN00PP0L`.

Build

- bpo-44133 [https://bugs.python.org/issue?@action=redirect&bpo=44133]: When Python is configured with **`--without-static-libpython`**, the Python static library (libpython.a) is no longer built. Patch by Victor Stinner.
- bpo-44133 [https://bugs.python.org/issue?@action=redirect&bpo=44133]: When Python is built without **`--enable-shared`**, the `python` program is now linked to object

files, rather than being linked to the Python static library (libpython.a), to make sure that all symbols are exported. Previously, the linker omitted some symbols like the **`Py_FrozenMain()`** function. Patch by Victor Stinner.

- bpo-40280 [https://bugs.python.org/issue?@action=redirect&bpo=40280]: The `configure` script has a new option `--with-emscripten-target` to select browser or node as Emscripten build target.
- bpo-46315 [https://bugs.python.org/issue?@action=redirect&bpo=46315]: Added and fixed `#ifdef HAVE_FEATURE` checks for functionality that is not available on WASI platform.
- bpo-45723 [https://bugs.python.org/issue?@action=redirect&bpo=45723]: Fixed a regression in `configure` check for **`select.epoll()`**.
- bpo-46263 [https://bugs.python.org/issue?@action=redirect&bpo=46263]: `configure` no longer sets `MULTIARCH` on FreeBSD platforms.
- bpo-46106 [https://bugs.python.org/issue?@action=redirect&bpo=46106]: Updated OpenSSL to 1.1.1m in Windows builds, macOS installer builds, and CI. Patch by Kumar Aditya.
- bpo-46088 [https://bugs.python.org/issue?@action=redirect&bpo=46088]: Automatically detect or install bootstrap Python runtime when building from Visual Studio.
- bpo-46072 [https://bugs.python.org/issue?@action=redirect&bpo=46072]: Add a –with-pystats configure option to turn on internal statistics gathering.
- bpo-40280 [https://bugs.python.org/issue?@action=redirect&bpo=40280]: A new directory `Tools/wasm` contains WebAssembly-related helpers like `config.site` override for wasm32-emscripten, wasm assets generator to bundle the stdlib, and a README.
- bpo-46023 [https://bugs.python.org/issue?@action=redirect&bpo=46023]: **makesetup** no longer builds

extensions that have been marked as *disabled*. This allows users to disable modules in `Modules/Setup.local`.
- bpo-45949 [https://bugs.python.org/issue?@action=redirect&bpo=45949]: Use pure Python `freeze_module` for all but importlib bootstrap files. `--with-freeze-module` **configure** option is no longer needed for cross builds.

Windows

- bpo-46217 [https://bugs.python.org/issue?@action=redirect&bpo=46217]: Removed parameter that is unsupported on Windows 8.1 and early Windows 10 and may have caused build or runtime failures.

macOS

- bpo-40477 [https://bugs.python.org/issue?@action=redirect&bpo=40477]: The Python Launcher app for macOS now properly launches scripts and, if necessary, the Terminal app when running on recent macOS releases.

C API

- bpo-46236 [https://bugs.python.org/issue?@action=redirect&bpo=46236]: Fix a bug in **`PyFunction_GetAnnotations()`** that caused it to return a `tuple` instead of a `dict`.
- bpo-46140 [https://bugs.python.org/issue?@action=redirect&bpo=46140]: **`PyBuffer_GetPointer()`**, **`PyBuffer_FromContiguous()`**, **`PyBuffer_ToContiguous()`** and **`PyMemoryView_FromBuffer()`** now take buffer info by `const Py_buffer *` instead of `Py_buffer *`, as they do not need mutability. **`PyBuffer_FromContiguous()`** also now takes the source buffer as `const void *`, and similarly

`PyBuffer_GetPointer()` takes the strides as `const Py_ssize_t *`.

- bpo-45855 [https://bugs.python.org/issue?@action=redirect&bpo=45855]: Document that the *no_block* argument to `PyCapsule_Import()` is a no-op now.
- bpo-45855 [https://bugs.python.org/issue?@action=redirect&bpo=45855]: Replaced deprecated usage of `PyImport_ImportModuleNoBlock()` with `PyImport_ImportModule()` in stdlib modules. Patch by Kumar Aditya.
- bpo-46007 [https://bugs.python.org/issue?@action=redirect&bpo=46007]: The **`PyUnicode_CHECK_INTERNED()`** macro has been excluded from the limited C API. It was never usable there, because it used internal structures which are not available in the limited C API. Patch by Victor Stinner.

Python 3.11.0 alpha 3

Release date: 2021-12-08

Core and Builtins

- bpo-46009 [https://bugs.python.org/issue?@action=redirect&bpo=46009]: Restore behavior from 3.9 and earlier when sending non-None to newly started generator. In 3.9 this did not affect the state of the generator. In 3.10.0 and 3.10.1 `gen_func().send(0)` is equivalent to `gen_func().throw(TypeError(...)` which exhausts the generator. In 3.10.2 onward, the behavior has been reverted to that of 3.9.
- bpo-46004 [https://bugs.python.org/issue?@action=redirect&bpo=46004]: Fix the `SyntaxError` location for errors involving for loops with invalid targets. Patch by Pablo Galindo

- bpo-45711 [https://bugs.python.org/issue?@action=redirect&bpo=45711]: **`_PyErr_ChainStackItem()`** no longer normalizes `exc_info` (including setting the traceback on the exception instance) because `exc_info` is always normalized.
- bpo-45607 [https://bugs.python.org/issue?@action=redirect&bpo=45607]: The `__note__` field was added to **`BaseException`**. It is `None` by default but can be set to a string which is added to the exception's traceback.
- bpo-45947 [https://bugs.python.org/issue?@action=redirect&bpo=45947]: Place pointers to dict and values immediately before GC header. This reduces number of dependent memory loads to access either dict or values from 3 to 1.
- bpo-45915 [https://bugs.python.org/issue?@action=redirect&bpo=45915]: `is_valid_fd` now uses faster `fcntl(fd, F_GETFD)` on Linux, macOS, and Windows.
- bpo-44530 [https://bugs.python.org/issue?@action=redirect&bpo=44530]: Reverts a change to the `code.__new__` audit event from an earlier prerelease.
- bpo-42268 [https://bugs.python.org/issue?@action=redirect&bpo=42268]: Fail the configure step if the selected compiler doesn't support memory sanitizer. Patch by Pablo Galindo
- bpo-45711 [https://bugs.python.org/issue?@action=redirect&bpo=45711]: The three values of `exc_info` are now always consistent with each other. In particular, the `type` and `traceback` fields are now derived from the exception instance. This impacts the return values of **`sys.exc_info()`** and **`PyErr_GetExcInfo()`** if the exception instance is modified while the exception is handled, as well as **`PyErr_SetExcInfo()`**, which now ignores the `type` and `traceback` arguments provided to it.
- bpo-45727 [https://bugs.python.org/issue?@action=redirect&bpo=45727]: Refine the custom syntax error that suggests that a comma may be missing to trigger

only when the expressions are detected between parentheses or brackets. Patch by Pablo Galindo

- bpo-45885 [https://bugs.python.org/issue?@action=redirect&bpo=45885]: Specialized the `COMPARE_OP` opcode using the PEP 659 machinery.
- bpo-45786 [https://bugs.python.org/issue?@action=redirect&bpo=45786]: Allocate space for the interpreter frame in the frame object, to avoid an additional allocation when the frame object outlives the frame activation.

Library

- bpo-45614 [https://bugs.python.org/issue?@action=redirect&bpo=45614]: Fix **`traceback`** display for exceptions with invalid module name.

Core and Builtins

- bpo-45813 [https://bugs.python.org/issue?@action=redirect&bpo=45813]: Fix crash when calling coro.cr_frame.clear() after coroutine has been freed.

- bpo-45811 [https://bugs.python.org/issue?@action=redirect&bpo=45811]: Improve the tokenizer errors when encountering invisible control characters in the parser. Patch by Pablo Galindo

- bpo-45848 [https://bugs.python.org/issue?@action=redirect&bpo=45848]: Allow the parser to obtain error lines directly from encoded files. Patch by Pablo Galindo

- bpo-45709 [https://bugs.python.org/issue?@action=redirect&bpo=45709]: Restore behavior from 3.10 when tracing an exception raised within a with statement.

- bpo-44525 [https://bugs.python.org/issue?@action=redirect&bpo=44525]: Adds new **`COPY_FREE_VARS`** opcode, to make copying of free variables from function to frame explicit. Helps optimization of calls to Python function.

- bpo-45829 [https://bugs.python.org/issue?@action=redirect&bpo=45829]: Specialize **`BINARY_SUBSCR`** for classes with a `__getitem__` method implemented in Python

- bpo-45826 [https://bugs.python.org/issue?@action=redirect&bpo=45826]: Fixed a crash when calling `.with_traceback(None)` on `NameError`. This occurs internally in `unittest.TestCase.assertRaises()`.

- bpo-45822 [https://bugs.python.org/issue?@action=redirect&bpo=45822]: Fixed a bug in the parser that was causing it to not respect **PEP 263** [https://peps.python.org/pep-0263/] coding cookies when no flags are provided. Patch by Pablo Galindo

- bpo-45820 [https://bugs.python.org/issue?@action=redirect&bpo=45820]: Fix a segfault when the parser fails without reading any input. Patch by Pablo Galindo

- bpo-45636 [https://bugs.python.org/issue?@action=redirect&bpo=45636]: Simplify the implementation of **`BINARY_OP`** by indexing into an array of function pointers (rather than switching on the oparg).

- bpo-42540 [https://bugs.python.org/issue?@action=redirect&bpo=42540]: Fix crash when **`os.fork()`** is called with an active non-default memory allocator.

- bpo-45738 [https://bugs.python.org/issue?@action=redirect&bpo=45738]: Fix computation of error location for invalid continuation characters in the parser. Patch by Pablo Galindo.

- bpo-45636 [https://bugs.python.org/issue?@action=redirect&bpo=45636]: Remove an existing “fast path” for old-style string formatting, since it no longer appears to have any measurable impact.
- bpo-45753 [https://bugs.python.org/issue?@action=redirect&bpo=45753]: Make recursion checks a bit more efficient by tracking amount of calls left before overflow.
- bpo-45773 [https://bugs.python.org/issue?@action=redirect&bpo=45773]: Fix a compiler hang when attempting to optimize certain jump patterns.
- bpo-45764 [https://bugs.python.org/issue?@action=redirect&bpo=45764]: The parser now gives a better error message when leaving out the opening parenthesis `(` after a `def`-statement:

  ```
  >>> def f:
    File "<stdin>", line 1
      def f:
           ^
  SyntaxError: expected '('
  ```

- bpo-45609 [https://bugs.python.org/issue?@action=redirect&bpo=45609]: Specialized the `STORE_SUBSCR` opcode using the PEP 659 machinery.
- bpo-45636 [https://bugs.python.org/issue?@action=redirect&bpo=45636]: Replace all numeric `BINARY_*` and `INPLACE_*` instructions with a single **`BINARY_OP`** implementation.

- bpo-45582 [https://bugs.python.org/issue?@action=redirect&bpo=45582]: Path calculation (known as `getpath`) has been reimplemented as a frozen Python module. This should have no visible impact, but may affect calculation of all paths referenced in **`sys`** and **`sysconfig`**.

- bpo-45450 [https://bugs.python.org/issue?@action=redirect&bpo=45450]: Improve the syntax error message for parenthesized arguments. Patch by Pablo Galindo.

Library

- bpo-27946 [https://bugs.python.org/issue?@action=redirect&bpo=27946]: Fix possible crash when getting an attribute of **`xml.etree.ElementTree.Element`** simultaneously with replacing the `attrib` dict.

- bpo-45711 [https://bugs.python.org/issue?@action=redirect&bpo=45711]: Make **`asyncio`** normalize exceptions as soon as they are captured with **`PyErr_Fetch()`**, and before they are stored as an exc_info triplet. This brings **`asyncio`** in line with the rest of the codebase, where an exc_info triplet is always normalized.

- bpo-23819 [https://bugs.python.org/issue?@action=redirect&bpo=23819]: Replaced asserts with exceptions in asyncio, patch by Kumar Aditya.

- bpo-13236 [https://bugs.python.org/issue?@action=redirect&bpo=13236]: **`unittest.TextTestResult`** and **`unittest.TextTestRunner`** flush now the output stream more often.

- bpo-45917 [https://bugs.python.org/issue?@action=redirect&bpo=45917]: Added `math.exp2()`:, which returns 2 raised to the power of x.
- bpo-37658 [https://bugs.python.org/issue?@action=redirect&bpo=37658]: Fix issue when on certain conditions `asyncio.wait_for()` may allow a coroutine to complete successfully, but fail to return the result, potentially causing memory leaks or other issues.
- bpo-45876 [https://bugs.python.org/issue?@action=redirect&bpo=45876]: Improve the accuracy of stdev() and pstdev() in the statistics module. When the inputs are floats or fractions, the output is a correctly rounded float
- bpo-44649 [https://bugs.python.org/issue?@action=redirect&bpo=44649]: Handle dataclass(slots=True) with a field that has default a default value, but for which init=False.
- bpo-45803 [https://bugs.python.org/issue?@action=redirect&bpo=45803]: Added missing kw_only parameter to dataclasses.make_dataclass().
- bpo-45837 [https://bugs.python.org/issue?@action=redirect&bpo=45837]: The **`turtle.RawTurtle.settiltangle()`** is deprecated since Python 3.1, it now emits a deprecation warning and will be removed in Python 3.13.

  Use **`turtle.RawTurtle.tiltangle()`** instead.

  `turtle.RawTurtle.tiltangle()` was earlier incorrectly marked as deprecated, its docstring has been corrected.

  Patch by Hugo van Kemenade.

- bpo-45831 [https://bugs.python.org/issue?@action=redirect&bpo=45831]: **`faulthandler`** can now write ASCII-only strings (like filenames and function names) with a single write() syscall when dumping a traceback. It reduces the risk of getting an unreadable dump when two threads or two processes dump a traceback to the same file (like stderr) at the same time. Patch by Victor Stinner.

- bpo-45828 [https://bugs.python.org/issue?@action=redirect&bpo=45828]: **`sqlite`** C callbacks now use unraisable exceptions if callback tracebacks are enabled. Patch by Erlend E. Aasland.

- bpo-41735 [https://bugs.python.org/issue?@action=redirect&bpo=41735]: Fix thread lock in `zlib.Decompress.flush()` method before `PyObject_GetBuffer`.

- bpo-45235 [https://bugs.python.org/issue?@action=redirect&bpo=45235]: Reverted an argparse bugfix that caused regression in the handling of default arguments for subparsers. This prevented leaf level arguments from taking precedence over root level arguments.

- bpo-45754 [https://bugs.python.org/issue?@action=redirect&bpo=45754]: Fix a regression in Python 3.11a1 and 3.11a2 where **`sqlite3`** incorrectly would use `SQLITE_LIMIT_LENGTH` when checking SQL statement lengths. Now, `SQLITE_LIMIT_SQL_LENGTH` is used. Patch by Erlend E. Aasland.

- bpo-45766 [https://bugs.python.org/issue?@action=redirect&bpo=45766]: Added *proportional* option to **`statistics.linear_regression()`**.

- bpo-45765 [https://bugs.python.org/issue?@action=redirect&bpo=45765]: In importlib.metadata, fix distribution discovery for an empty path.

- bpo-45757 [https://bugs.python.org/issue?@action=redirect&bpo=45757]: Fix bug where `dis` produced an incorrect oparg when `EXTENDED_ARG` is followed by an opcode that does not use its argument.

- bpo-45644 [https://bugs.python.org/issue?@action=redirect&bpo=45644]: In-place JSON file formatting using `python3 -m json.tool infile infile` now works correctly, previously it left the file empty. Patch by Chris Wesseling.

- bpo-45703 [https://bugs.python.org/issue?@action=redirect&bpo=45703]: When a namespace package is imported before another module from the same namespace is created/installed in a different `sys.path` location while the program is running, calling the `importlib.invalidate_caches()` function will now also guarantee the new module is noticed.

- bpo-45535 [https://bugs.python.org/issue?@action=redirect&bpo=45535]: Improve output of `dir()` with Enums.

- bpo-45664 [https://bugs.python.org/issue?@action=redirect&bpo=45664]: Fix `types.resolve_bases()` and `types.new_class()` for `types.GenericAlias` instance as a base.

- bpo-45663 [https://bugs.python.org/issue?@action=redirect&bpo=45663]: Fix `dataclasses.is_dataclass()` for dataclasses which are subclasses of `types.GenericAlias`.

- bpo-45662 [https://bugs.python.org/issue?@action=redirect&bpo=45662]: Fix the repr of `dataclasses.InitVar` with a type alias to the built-in class, e.g. `InitVar[list[int]]`.

- bpo-43137 [https://bugs.python.org/issue?@action=redirect&bpo=43137]: Launch GNOME web browsers via gio tool instead of obsolete gvfs-open

- bpo-45429 [https://bugs.python.org/issue?@action=redirect&bpo=45429]: On Windows, `time.sleep()` now uses a waitable timer which supports high-resolution timers. Patch by Donghee Na and Eryk Sun.

- bpo-37295 [https://bugs.python.org/issue?@action=redirect&bpo=37295]: Optimize `math.comb()` and `math.perm()`.

- bpo-45514 [https://bugs.python.org/issue?@action=redirect&bpo=45514]: Deprecated legacy functions in `importlib.resources`.

- bpo-45507 [https://bugs.python.org/issue?@action=redirect&bpo=45507]: Add tests for truncated/missing trailers in gzip.decompress implementation.

- bpo-45359 [https://bugs.python.org/issue?@action=redirect&bpo=45359]: Implement **PEP 585** [https://peps.python.org/pep-0585/] for `graphlib.TopologicalSorter`.

- bpo-44733 [https://bugs.python.org/issue?@action=redirect&bpo=44733]: Add `max_tasks_per_child` to `concurrent.futures.ProcessPoolExecutor`. This allows users to specify the maximum number of tasks a single process should execute before the process needs to be restarted.

- bpo-28806 [https://bugs.python.org/issue?@action=redirect&bpo=28806]: Improve netrc library. netrc file no longer needs to contain all tokens. And if the login name is anonymous, security check is no longer need.

- bpo-43498 [https://bugs.python.org/issue?@action=redirect&bpo=43498]: Avoid a possible *"RuntimeError: dictionary changed size during iteration"* when adjusting the process count of **`ProcessPoolExecutor`**.

- bpo-42158 [https://bugs.python.org/issue?@action=redirect&bpo=42158]: Add MIME types for N-quads, N-triples, Notation3 and TriG to `mimetypes`.

- bpo-30533 [https://bugs.python.org/issue?@action=redirect&bpo=30533]: Add **`inspect.getmembers_static()`** , it return all members without triggering dynamic lookup via the descriptor protocol. Patch by Weipeng Hong.

Documentation

- bpo-42238 [https://bugs.python.org/issue?@action=redirect&bpo=42238]: `make -C Doc suspicious` will be removed soon in favor of `make -C Doc check`, mark it as deprecated.
- bpo-45840 [https://bugs.python.org/issue?@action=redirect&bpo=45840]: Improve cross-references in the documentation for the data model.
- bpo-45640 [https://bugs.python.org/issue?@action=redirect&bpo=45640]: Properly marked-up grammar tokens in the documentation are now clickable and take you to the definition of a given piece of grammar. Patch by Arthur Milchior.
- bpo-45788 [https://bugs.python.org/issue?@action=redirect&bpo=45788]: Link doc for sys.prefix to sysconfig doc on installation paths.

- bpo-45772 [https://bugs.python.org/issue?@action=redirect&bpo=45772]: `socket.socket` documentation is corrected to a class from a function.
- bpo-45392 [https://bugs.python.org/issue?@action=redirect&bpo=45392]: Update the docstring of the **`type`** built-in to remove a redundant line and to mention keyword arguments for the constructor.
- bpo-45250 [https://bugs.python.org/issue?@action=redirect&bpo=45250]: Update the documentation to note that CPython does not consistently require iterators to define `__iter__`.
- bpo-25381 [https://bugs.python.org/issue?@action=redirect&bpo=25381]: In the extending chapter of the extending doc, update a paragraph about the global variables containing exception information.
- bpo-43905 [https://bugs.python.org/issue?@action=redirect&bpo=43905]: Expanded **`astuple()`** and **`asdict()`** docs, warning about deepcopy being applied and providing a workaround.

Tests

- bpo-45695 [https://bugs.python.org/issue?@action=redirect&bpo=45695]: Out-of-tree builds with a read-only source directory are now tested by CI.
- bpo-19460 [https://bugs.python.org/issue?@action=redirect&bpo=19460]: Add new Test for `Lib/email/mime/nonmultipart.py::MIMENonMultipart`.
- bpo-45835 [https://bugs.python.org/issue?@action=redirect&bpo=45835]: Fix race condition in test_queue tests with multiple “feeder” threads.
- bpo-45783 [https://bugs.python.org/issue?@action=redirect&bpo=45783]: The test for the freeze tool now handles file moves and deletions.
- bpo-45745 [https://bugs.python.org/issue?@action=redirect&bpo=45745]: Remove the `--findleaks`

command line option of regrtest: use the `--fail-env-changed` option instead. Since Python 3.7, it was a deprecated alias to the `--fail-env-changed` option.
- bpo-45701 [https://bugs.python.org/issue?@action=redirect&bpo=45701]: Add tests with `tuple` type with **`functools.lru_cache()`** to `test_functools`.

Build

- bpo-44035 [https://bugs.python.org/issue?@action=redirect&bpo=44035]: CI now verifies that autoconf files have been regenerated with a current and unpatched autoconf package.

- bpo-45950 [https://bugs.python.org/issue?@action=redirect&bpo=45950]: The build system now uses a **_bootstrap_python** interpreter for freezing and deepfreezing again. To speed up build process the build tools **_bootstrap_python** and **_freeze_module** are no longer build with LTO.

- bpo-45881 [https://bugs.python.org/issue?@action=redirect&bpo=45881]: The **configure** script now accepts `--with-build-python` and `--with-freeze-module` options to make cross compiling easier.

- bpo-40280 [https://bugs.python.org/issue?@action=redirect&bpo=40280]: Emscripten platform now uses `.wasm` suffix by default.

- bpo-40280 [https://bugs.python.org/issue?@action=redirect&bpo=40280]: Disable unusable core extension modules on WASM/Emscripten targets.

- bpo-40280 [https://bugs.python.org/issue?@action=redirect&bpo=40280]: `configure` now checks for socket `shutdown` function. The check makes it possible to disable `SYS_shutdown` with `ac_cv_func_shutdown=no` in CONFIG_SITE.

- bpo-40280 [https://bugs.python.org/issue?@action=redirect&bpo=40280]: `configure` now checks for functions `fork1, getegid, geteuid, getgid, getppid, getuid, opendir, pipe, system, wait, ttyname`.

- bpo-33393 [https://bugs.python.org/issue?@action=redirect&bpo=33393]: Update `config.guess` to 2021-06-03 and `config.sub` to 2021-08-14. `Makefile` now has an `update-config` target to make updating more convenient.

- bpo-45866 [https://bugs.python.org/issue?@action=redirect&bpo=45866]: `make regen-all` now produces the same output when run from a directory other than the source tree: when building Python out of the source tree. pegen now strips directory of the “generated by pygen from <FILENAME>” header Patch by Victor Stinner.

- bpo-40280 [https://bugs.python.org/issue?@action=redirect&bpo=40280]: `configure` now accepts machine `wasm32` or `wasm64` and OS `wasi` or `emscripten` for cross building, e.g. `wasm32-unknown-emscripten`, `wasm32-wasi`, or `wasm32-unknown-wasi`.

- bpo-41498 [https://bugs.python.org/issue?@action=redirect&bpo=41498]: Python now compiles on platforms without `sigset_t`. Several functions in **`signal`** are not available when `sigset_t` is missing.

  Based on patch by Roman Yurchak for pyodide.

- bpo-45881 [https://bugs.python.org/issue?@action=redirect&bpo=45881]: `setup.py` now uses `CC` from environment first to discover multiarch and cross compile paths.

- bpo-45886 [https://bugs.python.org/issue?@action=redirect&bpo=45886]: The `_freeze_module` program path can now be overridden on the command line, e.g. `make FREEZE_MODULE=../x86_64/Program/_freeze_module`.

- bpo-45873 [https://bugs.python.org/issue?@action=redirect&bpo=45873]: Get rid of the `_bootstrap_python` build step. The deepfreeze.py script is now run using `$(PYTHON_FOR_REGEN)` which can be Python 3.7 or newer (on Windows, 3.8 or newer).

- bpo-45847 [https://bugs.python.org/issue?@action=redirect&bpo=45847]: Port builtin hashlib extensions to `PY_STDLIB_MOD` macro and `addext()`.

- bpo-45723 [https://bugs.python.org/issue?@action=redirect&bpo=45723]: Add `autoconf` helpers for saving and restoring environment variables:

    - `SAVE_ENV`: Save `$CFLAGS`, `$LDFLAGS`, `$LIBS`, and `$CPPFLAGS`.
    - `RESTORE_ENV`: Restore `$CFLAGS`, `$LDFLAGS`, `$LIBS`, and `$CPPFLAGS`.
    - `WITH_SAVE_ENV([SCRIPT])`: Run `SCRIPT` wrapped with `SAVE_ENV` and `RESTORE_ENV`.

  Patch by Erlend E. Aasland.

- bpo-45573 [https://bugs.python.org/issue?@action=redirect&bpo=45573]: Mandatory core modules, that are required to bootstrap Python, are now in `Modules/Setup.bootstrap`.

- bpo-45573 [https://bugs.python.org/issue?@action=redirect&bpo=45573]: `configure` now creates

`Modules/Setup.stdlib` with conditionally enabled/disabled extension module lines. The file is not used, yet.

- bpo-45573 [https://bugs.python.org/issue?@action=redirect&bpo=45573]: `configure` now uses a unified format to set state, compiler flags, and linker flags in Makefile. The new macro `PY_STDLIB_MOD` sets three variables that are consumed by `Modules/Setup` and `setup.py`.

- bpo-45816 [https://bugs.python.org/issue?@action=redirect&bpo=45816]: Python now supports building with Visual Studio 2022 (MSVC v143, VS Version 17.0). Patch by Jeremiah Vivian.

- bpo-45800 [https://bugs.python.org/issue?@action=redirect&bpo=45800]: Settings for **`pyexpat`** C extension are now detected by `configure`. The bundled `expat` library is built in `Makefile`.

- bpo-45798 [https://bugs.python.org/issue?@action=redirect&bpo=45798]: Settings for **`decimal`** internal C extension are now detected by `configure`. The bundled `libmpdec` library is built in `Makefile`.

- bpo-45723 [https://bugs.python.org/issue?@action=redirect&bpo=45723]: **configure** has a new option `--with-pkg-config` to disable or require pkg-config.

- bpo-45774 [https://bugs.python.org/issue?@action=redirect&bpo=45774]: The build dependencies for **`sqlite3`** are now detected by `configure` and `pkg-config`. Patch by Erlend E. Aasland.

- bpo-45763 [https://bugs.python.org/issue?@action=redirect&bpo=45763]: The build dependencies for **`zlib`**, **`bz2`**, and **`lzma`** are now detected by `configure`.

- bpo-45747 [https://bugs.python.org/issue?@action=redirect&bpo=45747]: gdbm and dbm build dependencies are now detected by `configure`.

- bpo-45743 [https://bugs.python.org/issue?@action=redirect&bpo=45743]: On macOS, the build system no longer passes `search_paths_first` to the linker. The flag has been the default since Xcode 4 / macOS 10.6.

- bpo-45723 [https://bugs.python.org/issue?@action=redirect&bpo=45723]: `configure.ac` is now compatible with autoconf 2.71. Deprecated checks `STDC_HEADERS` and `AC_HEADER_TIME` have been removed.

- bpo-45723 [https://bugs.python.org/issue?@action=redirect&bpo=45723]: `configure` now prints a warning when pkg-config is missing.

- bpo-45731 [https://bugs.python.org/issue?@action=redirect&bpo=45731]: `configure --enable-loadable-sqlite-extensions` is now handled by new `PY_SQLITE_ENABLE_LOAD_EXTENSION` macro instead of logic in setup.py.

- bpo-45723 [https://bugs.python.org/issue?@action=redirect&bpo=45723]: configure.ac now uses custom helper macros and `AC_CACHE_CHECK` to simplify and speed up configure runs.

- bpo-45696 [https://bugs.python.org/issue?@action=redirect&bpo=45696]: Skip the marshal step for frozen modules by generating C code that produces a set of ready-to-use code objects. This speeds up startup time by another 10% or more.

- bpo-45561 [https://bugs.python.org/issue?@action=redirect&bpo=45561]: Run smelly.py tool from

$(srcdir).

Windows

- bpo-46105 [https://bugs.python.org/issue?@action=redirect&bpo=46105]: Fixed calculation of `sys.path` in a venv on Windows.
- bpo-45901 [https://bugs.python.org/issue?@action=redirect&bpo=45901]: When installed through the Microsoft Store and set as the default app for `*.py` files, command line arguments will now be passed to Python when invoking a script without explicitly launching Python (that is, `script.py args` rather than `python script.py args`).
- bpo-45616 [https://bugs.python.org/issue?@action=redirect&bpo=45616]: Fix Python Launcher’s ability to distinguish between versions 3.1 and 3.10 when either one is explicitly requested. Previously, 3.1 would be used if 3.10 was requested but not installed, and 3.10 would be used if 3.1 was requested but 3.10 was installed.
- bpo-45850 [https://bugs.python.org/issue?@action=redirect&bpo=45850]: Implement changes to build with deep-frozen modules on Windows. Note that we now require Python 3.10 as the “bootstrap” or “host” Python.
- bpo-45732 [https://bugs.python.org/issue?@action=redirect&bpo=45732]: Updates bundled Tcl/Tk to 8.6.12.
- bpo-45720 [https://bugs.python.org/issue?@action=redirect&bpo=45720]: Internal reference to `shlwapi.dll` was dropped to help improve startup time. This DLL will no longer be loaded at the start of every Python process.

macOS

- bpo-45732 [https://bugs.python.org/issue?@action=redirect&bpo=45732]: Update python.org macOS installer to use Tcl/Tk 8.6.12.

C API

- bpo-39026 [https://bugs.python.org/issue?@action=redirect&bpo=39026]: Fix Python.h to build C extensions with Xcode: remove a relative include from `Include/cpython/pystate.h`.

Python 3.11.0 alpha 2

Release date: 2021-11-05

Core and Builtins

- bpo-45716 [https://bugs.python.org/issue?@action=redirect&bpo=45716]: Improve the **`SyntaxError`** message when using `True`, `None` or `False` as keywords in a function call. Patch by Pablo Galindo.

- bpo-45688 [https://bugs.python.org/issue?@action=redirect&bpo=45688]: **`sys.stdlib_module_names`** now contains the macOS-specific module **`_scproxy`**.

- bpo-45379 [https://bugs.python.org/issue?@action=redirect&bpo=45379]: Clarify **`ImportError`** message when we try to explicitly import a frozen module but frozen modules are disabled.

- bpo-44525 [https://bugs.python.org/issue?@action=redirect&bpo=44525]: Specialize simple calls to Python functions (no starargs, keyword dict, or closure)

- bpo-45530 [https://bugs.python.org/issue?@action=redirect&bpo=45530]: Cases of sorting using tuples as keys may now be significantly faster in some cases. Patch by Tim Peters.

  The order of the result may differ from earlier releases if the tuple elements don't define a total ordering (see Value comparisons for information on total ordering). It's generally true that the result of sorting simply isn't well-defined in the absence of a total ordering on list elements.

- bpo-45526 [https://bugs.python.org/issue?@action=redirect&bpo=45526]: In obmalloc, set ADDRESS_BITS to not ignore any bits (ignored 16 before). That is safer in the case that the kernel gives user-space virtual addresses that span a range greater than 48 bits.

- bpo-30570 [https://bugs.python.org/issue?@action=redirect&bpo=30570]: Fixed a crash in `issubclass()` from infinite recursion when searching pathological `__bases__` tuples.

- bpo-45521 [https://bugs.python.org/issue?@action=redirect&bpo=45521]: Fix a bug in the obmalloc radix tree code. On 64-bit machines, the bug causes the tree to hold 46-bits of virtual addresses, rather than the intended 48-bits.

- bpo-45494 [https://bugs.python.org/issue?@action=redirect&bpo=45494]: Fix parser crash when reporting errors involving invalid continuation characters. Patch by Pablo Galindo.

- bpo-45445 [https://bugs.python.org/issue?@action=redirect&bpo=45445]: Python now fails to initialize if it finds an invalid `-X` option in the command line. Patch by Pablo Galindo.

- bpo-45340 [https://bugs.python.org/issue?@action=redirect&bpo=45340]: Object attributes are held in an array instead of a dictionary. An object's dictionary are created lazily, only when needed. Reduces the memory consumption of a typical Python object by about 30%. Patch by Mark Shannon.

- bpo-45408 [https://bugs.python.org/issue?@action=redirect&bpo=45408]: Fix a crash in the parser when reporting tokenizer errors that occur at the same time unclosed parentheses are detected. Patch by Pablo Galindo.

- bpo-29410 [https://bugs.python.org/issue?@action=redirect&bpo=29410]: Add SipHash13 for string hash algorithm and use it by default.

- bpo-45385 [https://bugs.python.org/issue?@action=redirect&bpo=45385]: Fix reference leak from descr_check. Patch by Donghee Na.

- bpo-45367 [https://bugs.python.org/issue?@action=redirect&bpo=45367]: Specialized the `BINARY_MULTIPLY` opcode to `BINARY_MULTIPLY_INT` and `BINARY_MULTIPLY_FLOAT` using the PEP 659 machinery.

- bpo-21736 [https://bugs.python.org/issue?@action=redirect&bpo=21736]: Frozen stdlib modules now have `__file__` to the .py file they would otherwise be loaded from, if possible. For packages, `__path__` now has the correct entry instead of being an empty list, which allows unfrozen submodules to be imported. These are set only if the stdlib directory is known when the runtime is initialized. Note that the file at `__file__` is not guaranteed to exist. None of this affects non-stdlib frozen modules nor, for now, frozen modules imported using `PyImport_ImportFrozenModule()`. Also, at the moment `co_filename` is not updated for the module.

- bpo-45020 [https://bugs.python.org/issue?@action=redirect&bpo=45020]: For frozen stdlib modules, record the original module name as `module.__spec__.loader_state.origname`. If the value is different than `module.__spec__.name` then the module was defined as an alias in Tools/scripts/freeze_modules.py. If it is `None` then the module comes from a source file outside the stdlib.

- bpo-45324 [https://bugs.python.org/issue?@action=redirect&bpo=45324]: In FrozenImporter.find_spec(), we now preserve the information needed in exec_module() to load the module. This change mostly impacts internal details, rather than changing the importer's behavior.

- bpo-45292 [https://bugs.python.org/issue?@action=redirect&bpo=45292]: Implement **PEP 654** [https://peps.python.org/pep-0654/]. Add **`ExceptionGroup`** and **`BaseExceptionGroup`**. Update traceback display code.

- bpo-40116 [https://bugs.python.org/issue?@action=redirect&bpo=40116]: Change to the implementation of split dictionaries. Classes where the instances differ either in the exact set of attributes, or in the order in which those attributes are set, can still share keys. This should have no observable effect on users of Python or the C-API. Patch by Mark Shannon.

- bpo-44050 [https://bugs.python.org/issue?@action=redirect&bpo=44050]: Extensions that indicate they use global state (by setting `m_size` to -1) can again be used in multiple interpreters. This reverts to behavior of Python 3.8.

- bpo-44525 [https://bugs.python.org/issue?@action=redirect&bpo=44525]: Setup initial specialization infrastructure for the `CALL_FUNCTION` opcode. Implemented initial specializations for C function calls:
  - `CALL_FUNCTION_BUILTIN_O` for `METH_O` flag.
  - `CALL_FUNCTION_BUILTIN_FAST` for `METH_FASTCALL` flag without keywords.
  - `CALL_FUNCTION_LEN` for `len(o)`.
  - `CALL_FUNCTION_ISINSTANCE` for `isinstance(o, t)`.
- bpo-44511 [https://bugs.python.org/issue?@action=redirect&bpo=44511]: Improve the generated bytecode for class and mapping patterns.
- bpo-43706 [https://bugs.python.org/issue?@action=redirect&bpo=43706]: Speed up calls to `enumerate()` by using the **PEP 590** [https://peps.python.org/pep-0590/] `vectorcall` calling convention. Patch by Donghee Na.

Library

- bpo-45679 [https://bugs.python.org/issue?@action=redirect&bpo=45679]: Fix caching of multi-value **`typing.Literal`**. `Literal[True, 2]` is no longer equal to `Literal[1, 2]`.
- bpo-42064 [https://bugs.python.org/issue?@action=redirect&bpo=42064]: Convert **`sqlite3`** to multi-phase initialisation (PEP 489). Patches by Erlend E. Aasland.
- bpo-45438 [https://bugs.python.org/issue?@action=redirect&bpo=45438]: Fix typing.Signature string

representation for generic builtin types.

- bpo-45613 [https://bugs.python.org/issue?@action=redirect&bpo=45613]: **`sqlite3`** now sets **`sqlite3.threadsafety`** based on the default threading mode the underlying SQLite library has been compiled with. Patch by Erlend E. Aasland.

- bpo-45574 [https://bugs.python.org/issue?@action=redirect&bpo=45574]: Fix warning about `print_escape` being unused.

- bpo-45581 [https://bugs.python.org/issue?@action=redirect&bpo=45581]: **`sqlite3.connect()`** now correctly raises **`MemoryError`** if the underlying SQLite API signals memory error. Patch by Erlend E. Aasland.

- bpo-45557 [https://bugs.python.org/issue?@action=redirect&bpo=45557]: **`pprint.pprint()`** now handles *underscore_numbers* correctly. Previously it was always setting it to `False`.

- bpo-44019 [https://bugs.python.org/issue?@action=redirect&bpo=44019]: Add **`operator.call()`** to `operator.__all__`. Patch by Kreusada.

- bpo-42174 [https://bugs.python.org/issue?@action=redirect&bpo=42174]: **`shutil.get_terminal_size()`** now falls back to sane values if the column or line count are 0.

- bpo-35673 [https://bugs.python.org/issue?@action=redirect&bpo=35673]: Improve the introspectability of the `__loader__` attribute for namespace packages. **`importlib.machinery.NamespaceLoader`** is now public, and implements the **`importlib.abc.InspectLoader`** interface. `_NamespaceLoader` is kept for backward compatibility.

- bpo-45515 [https://bugs.python.org/issue?@action=redirect&bpo=45515]: Add references to **`zoneinfo`** in the **`datetime`** documentation, mostly replacing outdated references to `dateutil.tz`. Change by Paul Ganssle.

- bpo-45475 [https://bugs.python.org/issue?@action=redirect&bpo=45475]: Reverted optimization of iterating **`gzip.GzipFile`**, **`bz2.BZ2File`**, and **`lzma.LZMAFile`** (see bpo-43787 [https://bugs.python.org/issue?@action=redirect&bpo=43787]) because it caused regression when user iterate them without having reference of them. Patch by Inada Naoki.

- bpo-45489 [https://bugs.python.org/issue?@action=redirect&bpo=45489]: Update **`ForwardRef`** to support `|` operator. Patch by Donghee Na.

- bpo-42222 [https://bugs.python.org/issue?@action=redirect&bpo=42222]: Removed deprecated support for float arguments in *randrange()*.

- bpo-45428 [https://bugs.python.org/issue?@action=redirect&bpo=45428]: Fix a regression in py_compile when reading filenames from standard input.

- bpo-45467 [https://bugs.python.org/issue?@action=redirect&bpo=45467]: Fix incremental decoder and stream reader in the “raw-unicode-escape” codec. Previously they failed if the escape sequence was split.

- bpo-45461 [https://bugs.python.org/issue?@action=redirect&bpo=45461]: Fix incremental decoder and stream reader in the “unicode-escape” codec. Previously they failed if the escape sequence was split.

- bpo-45239 [https://bugs.python.org/issue?@action=redirect&bpo=45239]: Fixed **`email.utils.parsedate_tz()`**

crashing with `UnboundLocalError` on certain invalid input instead of returning `None`. Patch by Ben Hoyt.

- bpo-45417 [https://bugs.python.org/issue?@action=redirect&bpo=45417]: Fix quadratic behaviour in the enum module: Creation of enum classes with a lot of entries was quadratic.
- bpo-45249 [https://bugs.python.org/issue?@action=redirect&bpo=45249]: Fix the behaviour of `traceback.print_exc()` when displaying the caret when the `end_offset` in the exception is set to 0. Patch by Pablo Galindo
- bpo-45416 [https://bugs.python.org/issue?@action=redirect&bpo=45416]: Fix use of `asyncio.Condition` with explicit `asyncio.Lock` objects, which was a regression due to removal of explicit loop arguments. Patch by Joongi Kim.
- bpo-20028 [https://bugs.python.org/issue?@action=redirect&bpo=20028]: Empty escapechar/quotechar is not allowed when initializing `csv.Dialect`. Patch by Vajrasky Kok and Donghee Na.
- bpo-44904 [https://bugs.python.org/issue?@action=redirect&bpo=44904]: Fix bug in the `doctest` module that caused it to fail if a docstring included an example with a `classmethod` `property`. Patch by Alex Waygood.
- bpo-45406 [https://bugs.python.org/issue?@action=redirect&bpo=45406]: Make `inspect.getmodule()` catch `FileNotFoundError` raised by **`inspect.getabsfile()`**, and return `None` to indicate that the module could not be determined.

- bpo-45411 [https://bugs.python.org/issue?@action=redirect&bpo=45411]: Add extensions for files containing subtitles - .srt & .vtt - to the mimetypes.py module.
- bpo-10716 [https://bugs.python.org/issue?@action=redirect&bpo=10716]: Migrated pydoc to HTML5 (without changing the look of it). Side effect is to update xmlrpc's `ServerHTMLDoc` which now uses the CSS too. cgitb now relies less on pydoc (as it can't use the CSS file).
- bpo-27580 [https://bugs.python.org/issue?@action=redirect&bpo=27580]: Add support of null characters in **`csv`**.
- bpo-45262 [https://bugs.python.org/issue?@action=redirect&bpo=45262]: Prevent use-after-free in asyncio. Make sure the cached running loop holder gets cleared on dealloc to prevent use-after-free in get_running_loop
- bpo-45386 [https://bugs.python.org/issue?@action=redirect&bpo=45386]: Make **`xmlrpc.client`** more robust to C runtimes where the underlying C `strftime` function results in a `ValueError` when testing for year formatting options.
- bpo-20028 [https://bugs.python.org/issue?@action=redirect&bpo=20028]: Improve error message of **`csv.Dialect`** when initializing. Patch by Vajrasky Kok and Donghee Na.
- bpo-45343 [https://bugs.python.org/issue?@action=redirect&bpo=45343]: Update bundled pip to 21.2.4 and setuptools to 58.1.0

- bpo-45328 [https://bugs.python.org/issue?@action=redirect&bpo=45328]: Fixed **`http.client.HTTPConnection`** to work properly in OSs that don’t support the `TCP_NODELAY` socket option.

- bpo-45243 [https://bugs.python.org/issue?@action=redirect&bpo=45243]: Add **`setlimit()`** and **`getlimit()`** to **`sqlite3.Connection`** for setting and getting SQLite limits by connection basis. Patch by Erlend E. Aasland.

- bpo-45320 [https://bugs.python.org/issue?@action=redirect&bpo=45320]: Removed from the **`inspect`** module:

    - the `getargspec` function, deprecated since Python 3.0; use **`inspect.signature()`** or **`inspect.getfullargspec()`** instead.
    - the `formatargspec` function, deprecated since Python 3.5; use the **`inspect.signature()`** function and **`Signature`** object directly.
    - the undocumented `Signature.from_builtin` and `Signature.from_function` functions, deprecated since Python 3.5; use the **`Signature.from_callable()`** method instead.

  Patch by Hugo van Kemenade.

- bpo-45192 [https://bugs.python.org/issue?@action=redirect&bpo=45192]: Fix the `tempfile._infer_return_type` function so that the `dir` argument of the **`tempfile`** functions accepts an object implementing the `os.PathLike` protocol.

  Patch by Kyungmin Lee.

- bpo-45160 [https://bugs.python.org/issue?@action=redirect&bpo=45160]: When tracing a tkinter variable used by a ttk OptionMenu, callbacks are no longer made twice.

- bpo-25625 [https://bugs.python.org/issue?@action=redirect&bpo=25625]: Added non parallel-safe **`chdir()`** context manager to change the current working directory and then restore it on exit. Simple wrapper around **`chdir()`**.

- bpo-24139 [https://bugs.python.org/issue?@action=redirect&bpo=24139]: Add support for SQLite extended result codes in **`sqlite3.Error`**. Patch by Erlend E. Aasland.

- bpo-24444 [https://bugs.python.org/issue?@action=redirect&bpo=24444]: Fixed an error raised in **`argparse`** help display when help for an option is set to 1+ blank spaces or when *choices* arg is an empty container.

- bpo-44547 [https://bugs.python.org/issue?@action=redirect&bpo=44547]: Implement `Fraction.__int__`, so that a **`fractions.Fraction`** instance `f` passes an `isinstance(f, typing.SupportsInt)` check.

- bpo-40321 [https://bugs.python.org/issue?@action=redirect&bpo=40321]: Adds support for HTTP 308 redirects to **`urllib`**. See **RFC 7538** [https://datatracker.ietf.org/doc/html/rfc7538.html] for details. Patch by Jochem Schulenklopper.

- bpo-41374 [https://bugs.python.org/issue?@action=redirect&bpo=41374]: Ensure that `socket.TCP_*` constants are exposed on Cygwin 3.1.6 and greater.

- bpo-35970 [https://bugs.python.org/issue?@action=redirect&bpo=35970]: Add help flag to the base64 module’s command line interface. Patch contributed by Robert Kuska.

Documentation

- bpo-45726 [https://bugs.python.org/issue?@action=redirect&bpo=45726]: Improve documentation for **`functools.singledispatch()`** and **`functools.singledispatchmethod`**.
- bpo-45680 [https://bugs.python.org/issue?@action=redirect&bpo=45680]: Amend the docs on `GenericAlias` objects to clarify that non-container classes can also implement `__class_getitem__`. Patch contributed by Alex Waygood.
- bpo-45618 [https://bugs.python.org/issue?@action=redirect&bpo=45618]: Update Sphinx version used to build the documentation to 4.2.0. Patch by Maciej Olko.
- bpo-45655 [https://bugs.python.org/issue?@action=redirect&bpo=45655]: Add a new “relevant PEPs” section to the top of the documentation for the `typing` module. Patch by Alex Waygood.
- bpo-45604 [https://bugs.python.org/issue?@action=redirect&bpo=45604]: Add `level` argument to `multiprocessing.log_to_stderr` function docs.
- bpo-45516 [https://bugs.python.org/issue?@action=redirect&bpo=45516]: Add protocol description to the **`importlib.abc.TraversableResources`** documentation.
- bpo-45464 [https://bugs.python.org/issue?@action=redirect&bpo=45464]: Mention in the documentation of Built-in Exceptions that inheriting from multiple exception types in a single subclass is not recommended due to possible memory layout incompatibility.

- bpo-45449 [https://bugs.python.org/issue?@action=redirect&bpo=45449]: Add note about **PEP 585** [https://peps.python.org/pep-0585/] in **`collections.abc`**.
- bpo-45516 [https://bugs.python.org/issue?@action=redirect&bpo=45516]: Add protocol description to the **`importlib.abc.Traversable`** documentation.
- bpo-20692 [https://bugs.python.org/issue?@action=redirect&bpo=20692]: Add Programming FAQ entry explaining that int literal attribute access requires either a space after or parentheses around the literal.

Tests

- bpo-45678 [https://bugs.python.org/issue?@action=redirect&bpo=45678]: Add tests for scenarios in which **`functools.singledispatchmethod`** is stacked on top of a method that has already been wrapped by two other decorators. Patch by Alex Waygood.
- bpo-45578 [https://bugs.python.org/issue?@action=redirect&bpo=45578]: Add tests for **`dis.distb()`**
- bpo-45678 [https://bugs.python.org/issue?@action=redirect&bpo=45678]: Add tests to ensure that `functools.singledispatchmethod` correctly wraps the attributes of the target function.
- bpo-45668 [https://bugs.python.org/issue?@action=redirect&bpo=45668]: PGO tests now pass when Python is built without test extension modules.
- bpo-45577 [https://bugs.python.org/issue?@action=redirect&bpo=45577]: Add subtests for all `pickle` protocols in `test_zoneinfo`.
- bpo-45566 [https://bugs.python.org/issue?@action=redirect&bpo=45566]: Fix `test_frozen_pickle` in `test_dataclasses` to check all `pickle` versions.
- bpo-43592 [https://bugs.python.org/issue?@action=redirect&bpo=43592]: **`test.libregrtest`** now raises the soft resource limit for the maximum number of file

descriptors when the default is too low for our test suite as was often the case on macOS.

- bpo-39679 [https://bugs.python.org/issue?@action=redirect&bpo=39679]: Add more test cases for `@functools.singledispatchmethod` when combined with `@classmethod` or `@staticmethod`.
- bpo-45410 [https://bugs.python.org/issue?@action=redirect&bpo=45410]: When libregrtest spawns a worker process, stderr is now written into stdout to keep messages order. Use a single pipe for stdout and stderr, rather than two pipes. Previously, messages were out of order which made analysis of buildbot logs harder Patch by Victor Stinner.
- bpo-45402 [https://bugs.python.org/issue?@action=redirect&bpo=45402]: Fix test_tools.test_sundry() when Python is built out of tree: fix how the freeze_modules.py tool locates the _freeze_module program. Patch by Victor Stinner.
- bpo-45403 [https://bugs.python.org/issue?@action=redirect&bpo=45403]: Fix test_sys.test_stdlib_dir() when Python is built outside the source tree: compare normalized paths. Patch by Victor Stinner.
- bpo-45400 [https://bugs.python.org/issue?@action=redirect&bpo=45400]: Fix test_name_error_suggestions_do_not_trigger_for_too_many_locals() of test_exceptions if a directory name contains “a1” (like “Python-3.11.0a1”): use a stricter regular expression. Patch by Victor Stinner.
- bpo-10572 [https://bugs.python.org/issue?@action=redirect&bpo=10572]: Rename **`sqlite3`** tests from `test_sqlite` to `test_sqlite3`, and relocate them to `Lib/test/test_sqlite3`. Patch by Erlend E. Aasland.

Build

- bpo-43158 [https://bugs.python.org/issue?@action=redirect&bpo=43158]: `setup.py` now uses values from configure script to build the `_uuid` extension module. Configure now detects util-linux's `libuuid`, too.
- bpo-45666 [https://bugs.python.org/issue?@action=redirect&bpo=45666]: Fix warning of `swprintf` and `%s` usage in `_testembed.c`
- bpo-45548 [https://bugs.python.org/issue?@action=redirect&bpo=45548]: `Modules/Setup` and `Modules/makesetup` have been improved. The `Setup` file now contains working rules for all extensions. Outdated comments have been removed. Rules defined by `makesetup` track dependencies correctly.
- bpo-45548 [https://bugs.python.org/issue?@action=redirect&bpo=45548]: The **`math`** and **`cmath`** implementation now require a C99 compatible `libm` and no longer ship with workarounds for missing acosh, asinh, atanh, expm1, and log1p functions.
- bpo-45595 [https://bugs.python.org/issue?@action=redirect&bpo=45595]: `setup.py` and `makesetup` now track build dependencies on all Python header files and module specific header files.
- bpo-45571 [https://bugs.python.org/issue?@action=redirect&bpo=45571]: `Modules/Setup` now use `PY_CFLAGS_NODIST` instead of `PY_CFLAGS` to compile shared modules.
- bpo-45570 [https://bugs.python.org/issue?@action=redirect&bpo=45570]: **`pyexpat`** and **`_elementtree`** no longer define obsolete macros `HAVE_EXPAT_CONFIG_H` and `USE_PYEXPAT_CAPI`. `XML_POOR_ENTROPY` is now defined in `expat_config.h`.
- bpo-43974 [https://bugs.python.org/issue?@action=redirect&bpo=43974]: `setup.py` no longer defines `Py_BUILD_CORE_MODULE`. Instead every module, that uses the internal API, defines the macro.

- bpo-45548 [https://bugs.python.org/issue?@action=redirect&bpo=45548]: Fill in missing entries in Modules/Setup.
- bpo-45532 [https://bugs.python.org/issue?@action=redirect&bpo=45532]: Update `sys.version` to use `main` as fallback information. Patch by Jeong YunWon.
- bpo-45536 [https://bugs.python.org/issue?@action=redirect&bpo=45536]: The `configure` script now checks whether OpenSSL headers and libraries provide required APIs. Most common APIs are verified. The check detects outdated or missing OpenSSL. Failures do not stop configure.
- bpo-45221 [https://bugs.python.org/issue?@action=redirect&bpo=45221]: Fixed regression in handling of `LDFLAGS` and `CPPFLAGS` options where **`argparse.parse_known_args()`** could interpret an option as one of the built-in command line argument, for example `-h` for help.
- bpo-45440 [https://bugs.python.org/issue?@action=redirect&bpo=45440]: Building Python now requires a C99 `<math.h>` header file providing the following functions: `copysign()`, `hypot()`, `isfinite()`, `isinf()`, `isnan()`, `round()`. Patch by Victor Stinner.
- bpo-45405 [https://bugs.python.org/issue?@action=redirect&bpo=45405]: Prevent `internal configure error` when running `configure` with recent versions of non-Apple clang. Patch by David Bohman.
- bpo-45433 [https://bugs.python.org/issue?@action=redirect&bpo=45433]: Avoid linking libpython with libcrypt.

Windows

- bpo-43652 [https://bugs.python.org/issue?@action=redirect&bpo=43652]: Update Tcl/Tk to 8.6.11, actually this time. The previous update incorrectly included 8.6.10.

- bpo-45337 [https://bugs.python.org/issue?@action=redirect&bpo=45337]: venv now warns when the created environment may need to be accessed at a different path, due to redirections, links or junctions. It also now correctly installs or upgrades components when the alternate path is required.
- bpo-43851 [https://bugs.python.org/issue?@action=redirect&bpo=43851]: Build SQLite `SQLITE_OMIT_AUTOINIT` on Windows. Patch by Erlend E. Aasland.

macOS

- bpo-44828 [https://bugs.python.org/issue?@action=redirect&bpo=44828]: Avoid tkinter file dialog failure on macOS 12 Monterey when using the Tk 8.6.11 provided by python.org macOS installers. Patch by Marc Culler of the Tk project.

IDLE

- bpo-45495 [https://bugs.python.org/issue?@action=redirect&bpo=45495]: Add context keywords ‘case’ and ‘match’ to completions list.

C API

- bpo-29103 [https://bugs.python.org/issue?@action=redirect&bpo=29103]: **`PyType_FromSpec*`** now copies the class name from the spec to a buffer owned by the class, so the original can be safely deallocated. Patch by Petr Viktorin.

- bpo-45522 [https://bugs.python.org/issue?@action=redirect&bpo=45522]: The internal freelists for frame, float, list, dict, async generators, and context objects can now be disabled.

- bpo-35134 [https://bugs.python.org/issue?@action=redirect&bpo=35134]: Exclude **`PyWeakref_GET_OBJECT()`** from the limited C API. It never worked since the **`PyWeakReference`** structure is opaque in the limited C API.

- bpo-35081 [https://bugs.python.org/issue?@action=redirect&bpo=35081]: Move the `interpreteridobject.h` header file from `Include/` to `Include/internal/`. It only provides private functions. Patch by Victor Stinner.

- bpo-35134 [https://bugs.python.org/issue?@action=redirect&bpo=35134]: The non-limited API files `cellobject.h`, `classobject.h`, `context.h`, `funcobject.h`, `genobject.h` and `longintrepr.h` have been moved to the `Include/cpython` directory. Moreover, the `eval.h` header file was removed. These files must not be included directly, as they are already included in `Python.h`: Include Files. If they have been included directly, consider including `Python.h` instead. Patch by Victor Stinner.

- bpo-45474 [https://bugs.python.org/issue?@action=redirect&bpo=45474]: The following items are no longer available when `Py_LIMITED_API` is defined:

    - **`PyMarshal_WriteLongToFile()`**
    - **`PyMarshal_WriteObjectToFile()`**
    - **`PyMarshal_ReadObjectFromString()`**
    - **`PyMarshal_WriteObjectToString()`**
    - the `Py_MARSHAL_VERSION` macro

  These are not part of the limited API.

Patch by Victor Stinner.

- bpo-45434 [https://bugs.python.org/issue?@action=redirect&bpo=45434]: Remove the `pystrhex.h` header file. It only contains private functions. C extensions should only include the main `<Python.h>` header file. Patch by Victor Stinner.

- bpo-45440 [https://bugs.python.org/issue?@action=redirect&bpo=45440]: Remove the `Py_FORCE_DOUBLE()` macro. It was used by the `Py_IS_INFINITY()` macro. Patch by Victor Stinner.

- bpo-45434 [https://bugs.python.org/issue?@action=redirect&bpo=45434]: `<Python.h>` no longer includes the header files `<stdlib.h>`, `<stdio.h>`, `<errno.h>` and `<string.h>` when the `Py_LIMITED_API` macro is set to `0x030b0000` (Python 3.11) or higher. C extensions should explicitly include the header files after `#include <Python.h>`. Patch by Victor Stinner.

- bpo-41123 [https://bugs.python.org/issue?@action=redirect&bpo=41123]: Remove `Py_UNICODE_COPY()` and `Py_UNICODE_FILL()` macros, deprecated since Python 3.3. Use `PyUnicode_CopyCharacters()` or `memcpy()` (`wchar_t*` string), and `PyUnicode_Fill()` functions instead. Patch by Victor Stinner.

- bpo-45412 [https://bugs.python.org/issue?@action=redirect&bpo=45412]: Remove the following math macros using the `errno` variable:

    - `Py_ADJUST_ERANGE1()`
    - `Py_ADJUST_ERANGE2()`
    - `Py_OVERFLOWED()`
    - `Py_SET_ERANGE_IF_OVERFLOW()`
    - `Py_SET_ERRNO_ON_MATH_ERROR()`

Patch by Victor Stinner.

- bpo-45395 [https://bugs.python.org/issue?@action=redirect&bpo=45395]: Custom frozen modules (the array set to `PyImport_FrozenModules`) are now treated as additions, rather than replacing all the default frozen modules. Frozen stdlib modules can still be disabled by setting the “code” field of the custom array entry to NULL.

- bpo-43760 [https://bugs.python.org/issue?@action=redirect&bpo=43760]: Add new **`PyThreadState_EnterTracing()`**, and **`PyThreadState_LeaveTracing()`** functions to the limited C API to suspend and resume tracing and profiling. Patch by Victor Stinner.

- bpo-44220 [https://bugs.python.org/issue?@action=redirect&bpo=44220]: **`PyStructSequence_UnnamedField`** is added to the Stable ABI.

Python 3.11.0 alpha 1

Release date: 2021-10-05

Security

- bpo-42278 [https://bugs.python.org/issue?@action=redirect&bpo=42278]: Replaced usage of **`tempfile.mktemp()`** with **`TemporaryDirectory`** to avoid a potential race condition.
- bpo-44600 [https://bugs.python.org/issue?@action=redirect&bpo=44600]: Fix incorrect line numbers while tracing some failed patterns in match statements. Patch by Charles Burkland.

- bpo-41180 [https://bugs.python.org/issue?@action=redirect&bpo=41180]: Add auditing events to the **`marshal`** module, and stop raising `code.__init__` events for every unmarshalled code object. Directly instantiated code objects will continue to raise an event, and audit event handlers should inspect or collect the raw marshal data. This reduces a significant performance overhead when loading from `.pyc` files.
- bpo-44394 [https://bugs.python.org/issue?@action=redirect&bpo=44394]: Update the vendored copy of libexpat to 2.4.1 (from 2.2.8) to get the fix for the **CVE 2013-0340** [https://www.cve.org/CVERecord?id=CVE-2013-0340] “Billion Laughs” vulnerability. This copy is most used on Windows and macOS.
- bpo-43124 [https://bugs.python.org/issue?@action=redirect&bpo=43124]: Made the internal `putcmd` function in **`smtplib`** sanitize input for presence of `\r` and `\n` characters to avoid (unlikely) command injection.
- bpo-44022 [https://bugs.python.org/issue?@action=redirect&bpo=44022]: **`http.client`** now avoids infinitely reading potential HTTP headers after a `100 Continue` status response from the server.

Core and Builtins

- bpo-43760 [https://bugs.python.org/issue?@action=redirect&bpo=43760]: The number of hardware branches per instruction dispatch is reduced from two to one by adding a special instruction for tracing. Patch by Mark Shannon.
- bpo-45061 [https://bugs.python.org/issue?@action=redirect&bpo=45061]: Add a deallocator to the bool type to detect refcount bugs in C extensions which call Py_DECREF(Py_True) or Py_DECREF(Py_False) by mistake. Detect also refcount bugs when the empty

tuple singleton or the Unicode empty string singleton is destroyed by mistake. Patch by Victor Stinner.

- bpo-24076 [https://bugs.python.org/issue?@action=redirect&bpo=24076]: sum() was further optimised for summing up single digit integers.
- bpo-45190 [https://bugs.python.org/issue?@action=redirect&bpo=45190]: Update Unicode databases to Unicode 14.0.0.
- bpo-45167 [https://bugs.python.org/issue?@action=redirect&bpo=45167]: Fix deepcopying of **`types.GenericAlias`** objects.
- bpo-45155 [https://bugs.python.org/issue?@action=redirect&bpo=45155]: **`int.to_bytes()`** and **`int.from_bytes()`** now take a default value of `"big"` for the `byteorder` argument. **`int.to_bytes()`** also takes a default value of `1` for the `length` argument.

Library

- bpo-44219 [https://bugs.python.org/issue?@action=redirect&bpo=44219]: Release the GIL while performing `isatty` system calls on arbitrary file descriptors. In particular, this affects **`os.isatty()`**, **`os.device_encoding()`** and **`io.TextIOWrapper`**. By extension, **`io.open()`** in text mode is also affected. This change solves a deadlock in **`os.isatty()`**. Patch by Vincent Michel in bpo-44219 [https://bugs.python.org/issue?@action=redirect&bpo=44219].

Core and Builtins

- bpo-44959 [https://bugs.python.org/issue?@action=redirect&bpo=44959]: Added fallback to extension modules with ‘.sl’ suffix on HP-UX

Library

- bpo-45121 [https://bugs.python.org/issue?@action=redirect&bpo=45121]: Fix issue where `Protocol.__init__` raises `RecursionError` when it's called directly or via `super()`. Patch provided by Yurii Karabas.

Core and Builtins

- bpo-44348 [https://bugs.python.org/issue?@action=redirect&bpo=44348]: The deallocator function of the **`BaseException`** type now uses the trashcan mechanism to prevent stack overflow. For example, when a **`RecursionError`** instance is raised, it can be linked to another RecursionError through the `__context__` attribute or the `__traceback__` attribute, and then a chain of exceptions is created. When the chain is destroyed, nested deallocator function calls can crash with a stack overflow if the chain is too long compared to the available stack memory. Patch by Victor Stinner.

- bpo-45123 [https://bugs.python.org/issue?@action=redirect&bpo=45123]: Fix PyAiter_Check to only check for the __anext__ presence (not for __aiter__). Rename PyAiter_Check to PyAIter_Check, PyObject_GetAiter -> PyObject_GetAIter.

- bpo-1514420 [https://bugs.python.org/issue?@action=redirect&bpo=1514420]: Interpreter no longer attempts to open files with names in angle brackets (like "<string>" or "<stdin>") when formatting an exception.

- bpo-41031 [https://bugs.python.org/issue?@action=redirect&bpo=41031]: Match C and Python code formatting of unprintable exceptions and exceptions in the **`__main__`** module.

- bpo-37330 [https://bugs.python.org/issue?@action=redirect&bpo=37330]: **`open()`**, **`io.open()`**, **`codecs.open()`** and **`fileinput.FileInput`** no longer accept `'U'` (“universal newline”) in the file mode. This flag was deprecated since Python 3.3. Patch by Victor Stinner.
- bpo-45083 [https://bugs.python.org/issue?@action=redirect&bpo=45083]: When the interpreter renders an exception, its name now has a complete qualname. Previously only the class name was concatenated to the module name, which sometimes resulted in an incorrect full name being displayed.

  (This issue impacted only the C code exception rendering, the **`traceback`** module was using qualname already).
- bpo-34561 [https://bugs.python.org/issue?@action=redirect&bpo=34561]: List sorting now uses the merge-ordering strategy from Munro and Wild’s `powersort()`. Unlike the former strategy, this is provably near-optimal in the entropy of the distribution of run lengths. Most uses of `list.sort()` probably won’t see a significant time difference, but may see significant improvements in cases where the former strategy was exceptionally poor. However, as these are all fast linear-time approximations to a problem that’s inherently at best quadratic-time to solve truly optimally, it’s also possible to contrive cases where the former strategy did better.
- bpo-45056 [https://bugs.python.org/issue?@action=redirect&bpo=45056]: Compiler now removes trailing unused constants from co_consts.

- bpo-45020 [https://bugs.python.org/issue?@action=redirect&bpo=45020]: Add a new command line option, "-X frozen_modules=[on|off]" to opt out of (or into) using optional frozen modules. This defaults to "on" (or "off" if it's running out of the source tree).

Library

- bpo-45012 [https://bugs.python.org/issue?@action=redirect&bpo=45012]: In **`posix`**, release GIL during `stat()`, `lstat()`, and `fstatat()` syscalls made by **`os.DirEntry.stat()`**. Patch by Stanisław Skonieczny.

Core and Builtins

- bpo-45018 [https://bugs.python.org/issue?@action=redirect&bpo=45018]: Fixed pickling of range iterators that iterated for over `2**32` times.
- bpo-45000 [https://bugs.python.org/issue?@action=redirect&bpo=45000]: A **`SyntaxError`** is now raised when trying to delete **`__debug__`**. Patch by Donghee Na.
- bpo-44963 [https://bugs.python.org/issue?@action=redirect&bpo=44963]: Implement `send()` and `throw()` methods for `anext_awaitable` objects. Patch by Pablo Galindo.

Library

- bpo-44962 [https://bugs.python.org/issue?@action=redirect&bpo=44962]: Fix a race in WeakKeyDictionary, WeakValueDictionary and WeakSet when two threads attempt to commit the last pending removal. This fixes asyncio.create_task and fixes a data loss in asyncio.run where shutdown_asyncgens is not run

Core and Builtins

- bpo-24234 [https://bugs.python.org/issue?@action=redirect&bpo=24234]: Implement the **`__bytes__()`** special method on the **`bytes`** type, so a bytes object `b` passes an `isinstance(b, typing.SupportsBytes)` check.
- bpo-24234 [https://bugs.python.org/issue?@action=redirect&bpo=24234]: Implement the **`__complex__()`** special method on the **`complex`** type, so a complex number `z` passes an `isinstance(z, typing.SupportsComplex)` check.
- bpo-44954 [https://bugs.python.org/issue?@action=redirect&bpo=44954]: Fixed a corner case bug where the result of `float.fromhex('0x.8p-1074')` was rounded the wrong way.
- bpo-44947 [https://bugs.python.org/issue?@action=redirect&bpo=44947]: Refine the syntax error for trailing commas in import statements. Patch by Pablo Galindo.
- bpo-44945 [https://bugs.python.org/issue?@action=redirect&bpo=44945]: Specialize the BINARY_ADD instruction using the PEP 659 machinery. Adds five new instructions:
    - BINARY_ADD_ADAPTIVE
    - BINARY_ADD_FLOAT
    - BINARY_ADD_INT
    - BINARY_ADD_UNICODE
    - BINARY_ADD_UNICODE_INPLACE_FAST

Library

- bpo-44929 [https://bugs.python.org/issue?@action=redirect&bpo=44929]: Fix some edge cases of `enum.Flag` string representation in the REPL. Patch by Pablo Galindo.

Core and Builtins

- bpo-44914 [https://bugs.python.org/issue?@action=redirect&bpo=44914]: Class version tags are no longer recycled.

  This means that a version tag serves as a unique identifier for the state of a class. We rely on this for effective specialization of the LOAD_ATTR and other instructions.

- bpo-44698 [https://bugs.python.org/issue?@action=redirect&bpo=44698]: Restore behaviour of complex exponentiation with integer-valued exponent of type **`float`** or **`complex`**.

- bpo-44895 [https://bugs.python.org/issue?@action=redirect&bpo=44895]: A debug variable **`PYTHONDUMPREFSFILE`** is added for creating a dump file which is generated by **`--with-trace-refs`**. Patch by Donghee Na.

- bpo-44900 [https://bugs.python.org/issue?@action=redirect&bpo=44900]: Add five superinstructions for PEP 659 quickening:

  - LOAD_FAST LOAD_FAST
  - STORE_FAST LOAD_FAST
  - LOAD_FAST LOAD_CONST
  - LOAD_CONST LOAD_FAST
  - STORE_FAST STORE_FAST

- bpo-44889 [https://bugs.python.org/issue?@action=redirect&bpo=44889]: Initial implementation of adaptive specialization of `LOAD_METHOD`. The following specialized forms were added:

  - `LOAD_METHOD_CACHED`
  - `LOAD_METHOD_MODULE`
  - `LOAD_METHOD_CLASS`

- bpo-44890 [https://bugs.python.org/issue?@action=redirect&bpo=44890]: Specialization stats are always collected in debug builds.

- bpo-44885 [https://bugs.python.org/issue?@action=redirect&bpo=44885]: Correct the ast locations of f-strings with format specs and repeated expressions. Patch by Pablo Galindo

- bpo-44878 [https://bugs.python.org/issue?@action=redirect&bpo=44878]: Remove the loop from the bytecode interpreter. All instructions end with a DISPATCH macro, so the loop is now redundant.

- bpo-44878 [https://bugs.python.org/issue?@action=redirect&bpo=44878]: Remove switch statement for interpreter loop when using computed gotos. This makes sure that we only have one dispatch table in the interpreter.

- bpo-44874 [https://bugs.python.org/issue?@action=redirect&bpo=44874]: Deprecate the old trashcan macros (`Py_TRASHCAN_SAFE_BEGIN`/`Py_TRASHCAN_SAFE_END`). They should be replaced by the new macros `Py_TRASHCAN_BEGIN` and `Py_TRASHCAN_END`.

- bpo-44872 [https://bugs.python.org/issue?@action=redirect&bpo=44872]: Use new trashcan macros (Py_TRASHCAN_BEGIN/END) in frameobject.c instead of the old ones (Py_TRASHCAN_SAFE_BEGIN/END).

- bpo-33930 [https://bugs.python.org/issue?@action=redirect&bpo=33930]: Fix segmentation fault with deep recursion when cleaning method objects. Patch by Augusto Goulart and Pablo Galindo.

- bpo-25782 [https://bugs.python.org/issue?@action=redirect&bpo=25782]: Fix bug where `PyErr_SetObject` hangs when the current exception has a cycle in its context chain.

- bpo-44856 [https://bugs.python.org/issue?@action=redirect&bpo=44856]: Fix reference leaks in the error paths of `update_bases()` and `__build_class__`. Patch by Pablo Galindo.

- bpo-44826 [https://bugs.python.org/issue?@action=redirect&bpo=44826]: Initial implementation of adaptive specialization of STORE_ATTR

  Three specialized forms of STORE_ATTR are added:

  - STORE_ATTR_SLOT
  - STORE_ATTR_SPLIT_KEYS
  - STORE_ATTR_WITH_HINT

- bpo-44838 [https://bugs.python.org/issue?@action=redirect&bpo=44838]: Fixed a bug that was causing the parser to raise an incorrect custom **`SyntaxError`** for invalid ‘if’ expressions. Patch by Pablo Galindo.

- bpo-44821 [https://bugs.python.org/issue?@action=redirect&bpo=44821]: Create instance dictionaries (__dict__) eagerly, to improve regularity of object layout and assist specialization.

- bpo-44792 [https://bugs.python.org/issue?@action=redirect&bpo=44792]: Improve syntax errors for if expressions. Patch by Miguel Brito

- bpo-34013 [https://bugs.python.org/issue?@action=redirect&bpo=34013]: Generalize the invalid legacy statement custom error message (like the one generated when “print” is called without parentheses) to include more generic expressions. Patch by Pablo Galindo
- bpo-44732 [https://bugs.python.org/issue?@action=redirect&bpo=44732]: Rename `types.Union` to `types.UnionType`.
- bpo-44725 [https://bugs.python.org/issue?@action=redirect&bpo=44725]: Expose specialization stats in python via **`_opcode.get_specialization_stats()`**.
- bpo-44717 [https://bugs.python.org/issue?@action=redirect&bpo=44717]: Improve AttributeError on circular imports of submodules.
- bpo-44698 [https://bugs.python.org/issue?@action=redirect&bpo=44698]: Fix undefined behaviour in complex object exponentiation.
- bpo-44653 [https://bugs.python.org/issue?@action=redirect&bpo=44653]: Support **`typing`** types in parameter substitution in the union type.
- bpo-44676 [https://bugs.python.org/issue?@action=redirect&bpo=44676]: Add ability to serialise `types.Union` objects. Patch provided by Yurii Karabas.
- bpo-44633 [https://bugs.python.org/issue?@action=redirect&bpo=44633]: Parameter substitution of the union type with wrong types now raises `TypeError` instead of returning **`NotImplemented`**.

- bpo-44661 [https://bugs.python.org/issue?@action=redirect&bpo=44661]: Update `property_descr_set` to use vectorcall if possible. Patch by Donghee Na.

- bpo-44662 [https://bugs.python.org/issue?@action=redirect&bpo=44662]: Add `__module__` to `types.Union`. This also fixes `types.Union` issues with `typing.Annotated`. Patch provided by Yurii Karabas.

- bpo-44655 [https://bugs.python.org/issue?@action=redirect&bpo=44655]: Include the name of the type in unset __slots__ attribute errors. Patch by Pablo Galindo

- bpo-44655 [https://bugs.python.org/issue?@action=redirect&bpo=44655]: Don’t include a missing attribute with the same name as the failing one when offering suggestions for missing attributes. Patch by Pablo Galindo

- bpo-44646 [https://bugs.python.org/issue?@action=redirect&bpo=44646]: Fix the hash of the union type: it no longer depends on the order of arguments.

- bpo-44636 [https://bugs.python.org/issue?@action=redirect&bpo=44636]: Collapse union of equal types. E.g. the result of `int | int` is now `int`. Fix comparison of the union type with non-hashable objects. E.g. `int | str == {}` no longer raises a TypeError.

Library

- bpo-44611 [https://bugs.python.org/issue?@action=redirect&bpo=44611]: On Windows, `os.urandom()`: uses BCryptGenRandom API instead of CryptGenRandom API which is deprecated from Microsoft Windows API. Patch by Donghee Na.

Core and Builtins

- bpo-44635 [https://bugs.python.org/issue?@action=redirect&bpo=44635]: Convert `None` to `type(None)` in the union type constructor.
- bpo-26280 [https://bugs.python.org/issue?@action=redirect&bpo=26280]: Implement adaptive specialization for BINARY_SUBSCR

  Three specialized forms of BINARY_SUBSCR are added:

  - BINARY_SUBSCR_LIST_INT
  - BINARY_SUBSCR_TUPLE_INT
  - BINARY_SUBSCR_DICT
- bpo-44589 [https://bugs.python.org/issue?@action=redirect&bpo=44589]: Mapping patterns in `match` statements with two or more equal literal keys will now raise a **`SyntaxError`** at compile-time.
- bpo-44606 [https://bugs.python.org/issue?@action=redirect&bpo=44606]: Fix `__instancecheck__` and `__subclasscheck__` for the union type.
- bpo-42073 [https://bugs.python.org/issue?@action=redirect&bpo=42073]: The `@classmethod` decorator can now wrap other classmethod-like descriptors.
- bpo-41972 [https://bugs.python.org/issue?@action=redirect&bpo=41972]: Tuned the string-searching algorithm of fastsearch.h to have a shorter inner loop for most cases.
- bpo-44590 [https://bugs.python.org/issue?@action=redirect&bpo=44590]: All necessary data for executing a Python function (local variables, stack, etc)

is now kept in a per-thread stack. Frame objects are lazily allocated on demand. This increases performance by about 7% on the standard benchmark suite. Introspection and debugging are unaffected as frame objects are always available when needed. Patch by Mark Shannon.

- bpo-44584 [https://bugs.python.org/issue?@action=redirect&bpo=44584]: The threading debug (**`PYTHONTHREADDEBUG`** environment variable) is deprecated in Python 3.10 and will be removed in Python 3.12. This feature requires a debug build of Python. Patch by Victor Stinner.

- bpo-43895 [https://bugs.python.org/issue?@action=redirect&bpo=43895]: An obsolete internal cache of shared object file handles added in 1995 that attempted, but did not guarantee, that a .so would not be dlopen’ed twice to work around flaws in mid-1990s posix-ish operating systems has been removed from dynload_shlib.c.

Library

- bpo-44490 [https://bugs.python.org/issue?@action=redirect&bpo=44490]: **`typing`** now searches for type parameters in `types.Union` objects. `get_type_hints` will also properly resolve annotations with nested `types.Union` objects. Patch provided by Yurii Karabas.

Core and Builtins

- bpo-43950 [https://bugs.python.org/issue?@action=redirect&bpo=43950]: Code objects can now provide the column information for instructions when available.

This is levaraged during traceback printing to show the expressions responsible for errors.

Contributed by Pablo Galindo, Batuhan Taskaya and Ammar Askar as part of **PEP 657** [https://peps.python.org/pep-0657/].

- bpo-44562 [https://bugs.python.org/issue?@action=redirect&bpo=44562]: Remove uses of `PyObject_GC_Del()` in error path when initializing `types.GenericAlias`.

Library

- bpo-41486 [https://bugs.python.org/issue?@action=redirect&bpo=41486]: Fix a memory consumption and copying performance regression in earlier 3.10 beta releases if someone used an output buffer larger than 4GiB with zlib.decompress on input data that expands that large.

Core and Builtins

- bpo-43908 [https://bugs.python.org/issue?@action=redirect&bpo=43908]: Heap types with the `Py_TPFLAGS_IMMUTABLETYPE` flag can now inherit the **PEP 590** [https://peps.python.org/pep-0590/] vectorcall protocol. Previously, this was only possible for static types. Patch by Erlend E. Aasland.

- bpo-44553 [https://bugs.python.org/issue?@action=redirect&bpo=44553]: Implement GC methods for `types.Union` to break reference cycles and prevent memory leaks.

- bpo-44490 [https://bugs.python.org/issue?@action=redirect&bpo=44490]: Add `__parameters__` attribute and `__getitem__` operator to `types.Union`. Patch provided by Yurii Karabas.

- bpo-44523 [https://bugs.python.org/issue?@action=redirect&bpo=44523]: Remove the pass-through for **`hash()`** of **`weakref.proxy`** objects to prevent unintended consequences when the original referred object dies while the proxy is part of a hashable object. Patch by Pablo Galindo.

- bpo-44483 [https://bugs.python.org/issue?@action=redirect&bpo=44483]: Fix a crash in `types.Union` objects when creating a union of an object with bad `__module__` field.

- bpo-44486 [https://bugs.python.org/issue?@action=redirect&bpo=44486]: Modules will always have a dictionary, even when created by `types.ModuleType.__new__()`

- bpo-44472 [https://bugs.python.org/issue?@action=redirect&bpo=44472]: Fix ltrace functionality when exceptions are raised. Patch by Pablo Galindo

- bpo-12022 [https://bugs.python.org/issue?@action=redirect&bpo=12022]: A **`TypeError`** is now raised instead of an **`AttributeError`** in **`with`** and **`async with`** statements for objects which do not support the context manager or asynchronous context manager protocols correspondingly.

- bpo-44297 [https://bugs.python.org/issue?@action=redirect&bpo=44297]: Make sure that the line number is set when entering a comprehension scope. This ensures that backtraces including generator expressions show the correct line number.

- bpo-44456 [https://bugs.python.org/issue?@action=redirect&bpo=44456]: Improve the syntax error when mixing positional and keyword patterns. Patch by Pablo Galindo.

- bpo-44409 [https://bugs.python.org/issue?@action=redirect&bpo=44409]: Fix error location information for tokenizer errors raised on initialization of the tokenizer. Patch by Pablo Galindo.

- bpo-44396 [https://bugs.python.org/issue?@action=redirect&bpo=44396]: Fix a possible crash in the tokenizer when raising syntax errors for unclosed strings. Patch by Pablo Galindo.

- bpo-44376 [https://bugs.python.org/issue?@action=redirect&bpo=44376]: Exact integer exponentiation (like `i**2` or `pow(i, 2)`) with a small exponent is much faster, due to reducing overhead in such cases.

- bpo-44313 [https://bugs.python.org/issue?@action=redirect&bpo=44313]: Directly imported objects and modules (through import and from import statements) don’t generate `LOAD_METHOD`/`CALL_METHOD` for directly accessed objects on their namespace. They now use the regular `LOAD_ATTR`/`CALL_FUNCTION`.

- bpo-44338 [https://bugs.python.org/issue?@action=redirect&bpo=44338]: Implement adaptive specialization for LOAD_GLOBAL

  Two specialized forms of LOAD_GLOBAL are added:

  - LOAD_GLOBAL_MODULE

  - LOAD_GLOBAL_BUILTIN

- bpo-44368 [https://bugs.python.org/issue?@action=redirect&bpo=44368]: Improve syntax errors for invalid “as” targets. Patch by Pablo Galindo

- bpo-44349 [https://bugs.python.org/issue?@action=redirect&bpo=44349]: Fix an edge case when displaying text from files with encoding in syntax errors. Patch by Pablo Galindo.

- bpo-44337 [https://bugs.python.org/issue?@action=redirect&bpo=44337]: Initial implementation of adaptive specialization of LOAD_ATTR

  Four specialized forms of LOAD_ATTR are added:

  - LOAD_ATTR_SLOT
  - LOAD_ATTR_SPLIT_KEYS
  - LOAD_ATTR_WITH_HINT
  - LOAD_ATTR_MODULE

- bpo-44335 [https://bugs.python.org/issue?@action=redirect&bpo=44335]: Fix a regression when identifying incorrect characters in syntax errors. Patch by Pablo Galindo

- bpo-43693 [https://bugs.python.org/issue?@action=redirect&bpo=43693]: Computation of the offsets of cell variables is done in the compiler instead of at runtime. This reduces the overhead of handling cell and free variables, especially in the case where a variable is both an argument and cell variable.

- bpo-44317 [https://bugs.python.org/issue?@action=redirect&bpo=44317]: Improve tokenizer error with improved locations. Patch by Pablo Galindo.

Library

- bpo-44304 [https://bugs.python.org/issue?@action=redirect&bpo=44304]: Fix a crash in the **`sqlite3`** module that happened when the garbage collector clears **`sqlite.Statement`** objects. Patch by Pablo Galindo

Core and Builtins

- bpo-44305 [https://bugs.python.org/issue?@action=redirect&bpo=44305]: Improve error message for `try` blocks without `except` or `finally` blocks. Patch by Pablo Galindo.

- bpo-43413 [https://bugs.python.org/issue?@action=redirect&bpo=43413]: Constructors of subclasses of some builtin classes (e.g. **`tuple`**, **`list`**, **`frozenset`**) no longer accept arbitrary keyword arguments. [reverted in 3.11a4] Subclass of **`set`** can now define a `__new__()` method with additional keyword parameters without overriding also `__init__()`.

- bpo-43667 [https://bugs.python.org/issue?@action=redirect&bpo=43667]: Improve Unicode support in non-UTF locales on Oracle Solaris. This issue does not affect other Solaris systems.

- bpo-43693 [https://bugs.python.org/issue?@action=redirect&bpo=43693]: A new opcode MAKE_CELL has been added that effectively moves some of the work done on function entry into the compiler and into the eval loop. In addition to creating the required cell objects, the new opcode converts relevant arguments (and other locals) to cell variables on function entry.

- bpo-44232 [https://bugs.python.org/issue?@action=redirect&bpo=44232]: Fix a regression in `type()` when a metaclass raises an exception. The C function **`type_new()`** must properly report the exception when a metaclass constructor raises an exception and the winner class is not the metaclass. Patch by Victor Stinner.

- bpo-44201 [https://bugs.python.org/issue?@action=redirect&bpo=44201]: Avoid side effects of checking for specialized syntax errors in the REPL that was causing it to ask for extra tokens after a syntax error had been detected. Patch by Pablo Galindo

- bpo-43693 [https://bugs.python.org/issue?@action=redirect&bpo=43693]: `PyCodeObject` gained `co_fastlocalnames` and `co_fastlocalkinds` as the authoritative source of fast locals info. Marshaled code objects have changed accordingly.

- bpo-44184 [https://bugs.python.org/issue?@action=redirect&bpo=44184]: Fix a crash at Python exit when a deallocator function removes the last strong reference to a heap type. Patch by Victor Stinner.

- bpo-44187 [https://bugs.python.org/issue?@action=redirect&bpo=44187]: Implement quickening in the interpreter. This offers no advantages as yet, but is an enabler of future optimizations. See PEP 659 for full explanation.

- bpo-44180 [https://bugs.python.org/issue?@action=redirect&bpo=44180]: The parser doesn’t report generic syntax errors that happen in a position further away that the one it reached in the first pass. Patch by Pablo Galindo

- bpo-44168 [https://bugs.python.org/issue?@action=redirect&bpo=44168]: Fix error message in the parser involving keyword arguments with invalid expressions. Patch by Pablo Galindo

- bpo-44156 [https://bugs.python.org/issue?@action=redirect&bpo=44156]: String caches in `compile.c` are now subinterpreter compatible.

- bpo-44143 [https://bugs.python.org/issue?@action=redirect&bpo=44143]: Fixed a crash in the parser that manifest when raising tokenizer errors when an existing exception was present. Patch by Pablo Galindo.

- bpo-44032 [https://bugs.python.org/issue?@action=redirect&bpo=44032]: Move ‘fast’ locals and other variables from the frame object to a per-thread datastack.

- bpo-44114 [https://bugs.python.org/issue?@action=redirect&bpo=44114]: Fix incorrect dictkeys_reversed and dictitems_reversed function signatures in C code, which broke webassembly builds.

- bpo-44110 [https://bugs.python.org/issue?@action=redirect&bpo=44110]: Improve **`str.__getitem__()`** error message

- bpo-26110 [https://bugs.python.org/issue?@action=redirect&bpo=26110]: Add `CALL_METHOD_KW` opcode to speed up method calls with keyword arguments. Idea originated from PyPy. A side effect is executing `CALL_METHOD` is now branchless in the evaluation loop.

- bpo-28307 [https://bugs.python.org/issue?@action=redirect&bpo=28307]: Compiler now optimizes simple C-style formatting with literal format containing only format codes %s, %r and %a by converting them to f-string expressions.

- bpo-43149 [https://bugs.python.org/issue?@action=redirect&bpo=43149]: Correct the syntax error message regarding multiple exception types to not refer to “exception groups”. Patch by Pablo Galindo
- bpo-43822 [https://bugs.python.org/issue?@action=redirect&bpo=43822]: The parser will prioritize tokenizer errors over custom syntax errors when raising exceptions. Patch by Pablo Galindo.
- bpo-40222 [https://bugs.python.org/issue?@action=redirect&bpo=40222]: “Zero cost” exception handling.
    - Uses a lookup table to determine how to handle exceptions.
    - Removes SETUP_FINALLY and POP_TOP block instructions, eliminating the runtime overhead of try statements.
    - Reduces the size of the frame object by about 60%.

  Patch by Mark Shannon
- bpo-43918 [https://bugs.python.org/issue?@action=redirect&bpo=43918]: Document the signature and `default` argument in the docstring of the new `anext` builtin.
- bpo-43833 [https://bugs.python.org/issue?@action=redirect&bpo=43833]: Emit a deprecation warning if the numeric literal is immediately followed by one of keywords: and, else, for, if, in, is, or. Raise a syntax error with more informative message if it is immediately followed by other keyword or identifier.

- bpo-43879 [https://bugs.python.org/issue?@action=redirect&bpo=43879]: Add native_thread_id to PyThreadState. Patch by Gabriele N. Tornetta.
- bpo-43693 [https://bugs.python.org/issue?@action=redirect&bpo=43693]: Compute cell offsets relative to locals in compiler. Allows the interpreter to treats locals and cells a single array, which is slightly more efficient. Also make the LOAD_CLOSURE opcode an alias for LOAD_FAST. Preserving LOAD_CLOSURE helps keep bytecode a bit more readable.
- bpo-17792 [https://bugs.python.org/issue?@action=redirect&bpo=17792]: More accurate error messages for access of unbound locals or free vars.
- bpo-28146 [https://bugs.python.org/issue?@action=redirect&bpo=28146]: Fix a confusing error message in `str.format()`.
- bpo-11105 [https://bugs.python.org/issue?@action=redirect&bpo=11105]: When compiling `ast.AST` objects with recursive references through `compile()`, the interpreter doesn’t crash anymore instead it raises a `RecursionError`.
- bpo-39091 [https://bugs.python.org/issue?@action=redirect&bpo=39091]: Fix crash when using passing a non-exception to a generator’s `throw()` method. Patch by Noah Oxer
- bpo-33346 [https://bugs.python.org/issue?@action=redirect&bpo=33346]: Asynchronous comprehensions are now allowed inside comprehensions in asynchronous functions. Outer comprehensions implicitly become asynchronous.

Library

- bpo-45371 [https://bugs.python.org/issue?@action=redirect&bpo=45371]: Fix clang rpath issue in `distutils`. The UnixCCompiler now uses correct clang option to add a runtime library directory (rpath) to a shared library.

- bpo-45329 [https://bugs.python.org/issue?@action=redirect&bpo=45329]: Fix freed memory access in **`pyexpat.xmlparser`** when building it with an installed expat library <= 2.2.0.

- bpo-41710 [https://bugs.python.org/issue?@action=redirect&bpo=41710]: On Unix, if the `sem_clockwait()` function is available in the C library (glibc 2.30 and newer), the **`threading.Lock.acquire()`** method now uses the monotonic clock (**`time.CLOCK_MONOTONIC`**) for the timeout, rather than using the system clock (**`time.CLOCK_REALTIME`**), to not be affected by system clock changes. Patch by Victor Stinner.

- bpo-1596321 [https://bugs.python.org/issue?@action=redirect&bpo=1596321]: Fix the **`threading._shutdown()`** function when the **`threading`** module was imported first from a thread different than the main thread: no longer log an error at Python exit.

- bpo-45274 [https://bugs.python.org/issue?@action=redirect&bpo=45274]: Fix a race condition in the **`Thread.join()`** method of the **`threading`** module. If the function is interrupted by a signal and the signal handler raises an exception, make sure that the thread remains in a consistent state to prevent a deadlock. Patch by Victor Stinner.

- bpo-21302 [https://bugs.python.org/issue?@action=redirect&bpo=21302]: In Unix operating systems, **`time.sleep()`** now uses the `nanosleep()` function, if `clock_nanosleep()` is not available but `nanosleep()` is available. `nanosleep()` allows to sleep with nanosecond precision.

- bpo-21302 [https://bugs.python.org/issue?@action=redirect&bpo=21302]: On Windows, **`time.sleep()`** now uses a waitable timer which has a resolution of 100 nanoseconds (10^{-7} seconds). Previously, it had a resolution of 1 millisecond (10^{-3} seconds). Patch by Benjamin Szőke and Victor Stinner.

- bpo-45238 [https://bugs.python.org/issue?@action=redirect&bpo=45238]: Fix **`unittest.IsolatedAsyncioTestCase.debug()`**: it runs now asynchronous methods and callbacks.

- bpo-36674 [https://bugs.python.org/issue?@action=redirect&bpo=36674]: **`unittest.TestCase.debug()`** raises now a **`unittest.SkipTest`** if the class or the test method are decorated with the skipping decorator.

- bpo-45235 [https://bugs.python.org/issue?@action=redirect&bpo=45235]: Fix an issue where argparse would not preserve values in a provided namespace when using a subparser with defaults.

- bpo-45183 [https://bugs.python.org/issue?@action=redirect&bpo=45183]: Have zipimport.zipimporter.find_spec() not raise an exception when the underlying zip file has been deleted and the internal cache has been reset via invalidate_cache().

- bpo-45234 [https://bugs.python.org/issue?@action=redirect&bpo=45234]: Fixed a regression in `copyfile()`, `copy()`, `copy2()` raising `FileNotFoundError` when source is a directory, which should raise `IsADirectoryError`
- bpo-45228 [https://bugs.python.org/issue?@action=redirect&bpo=45228]: Fix stack buffer overflow in parsing J1939 network address.
- bpo-45225 [https://bugs.python.org/issue?@action=redirect&bpo=45225]: use map function instead of genexpr in capwords.
- bpo-42135 [https://bugs.python.org/issue?@action=redirect&bpo=42135]: Fix typo: `importlib.find_loader` is really slated for removal in Python 3.12 not 3.10, like the others in PR 25169.

  Patch by Hugo van Kemenade.
- bpo-20524 [https://bugs.python.org/issue?@action=redirect&bpo=20524]: Improves error messages on `.format()` operation for `str`, `float`, `int`, and `complex`. New format now shows the problematic pattern and the object type.
- bpo-45168 [https://bugs.python.org/issue?@action=redirect&bpo=45168]: Change `dis.dis()` output to omit op arg values that cannot be resolved due to `co_consts`, `co_names` etc not being provided. Previously the oparg itself was repeated in the value field, which is not useful and can be confusing.
- bpo-21302 [https://bugs.python.org/issue?@action=redirect&bpo=21302]: In Unix operating systems, `time.sleep()` now uses the `clock_nanosleep()` function, if

available, which allows to sleep for an interval specified with nanosecond precision.

- bpo-45173 [https://bugs.python.org/issue?@action=redirect&bpo=45173]: Remove from the **`configparser`** module: the **`SafeConfigParser`** class, the **`filename`** property of the **`ParsingError`** class, the **`readfp()`** method of the **`ConfigParser`** class, deprecated since Python 3.2.

  Patch by Hugo van Kemenade.

- bpo-44987 [https://bugs.python.org/issue?@action=redirect&bpo=44987]: Pure ASCII strings are now normalized in constant time by **`unicodedata.normalize()`**. Patch by Donghee Na.

- bpo-35474 [https://bugs.python.org/issue?@action=redirect&bpo=35474]: Calling **`mimetypes.guess_all_extensions()`** with `strict=False` no longer affects the result of the following call with `strict=True`. Also, mutating the returned list no longer affects the global state.

- bpo-45166 [https://bugs.python.org/issue?@action=redirect&bpo=45166]: **`typing.get_type_hints()`** now works with **`Final`** wrapped in **`ForwardRef`**.

- bpo-45162 [https://bugs.python.org/issue?@action=redirect&bpo=45162]: Remove many old deprecated **`unittest`** features:

  - “`fail*`” and “`assert*`” aliases of **`TestCase`** methods.
  - Broken from start **`TestCase`** method `assertDictContainsSubset()`.
  - Ignored **`<unittest.TestLoader.loadTestsFromModule> TestLoader.loadTestsFromModule()`** parameter *use_load_tests*.

  - Old alias `_TextTestResult` of **`TextTestResult`**.

- bpo-38371 [https://bugs.python.org/issue?@action=redirect&bpo=38371]: Remove the deprecated `split()` method of **`_tkinter.TkappType`**. Patch by Erlend E. Aasland.

- bpo-20499 [https://bugs.python.org/issue?@action=redirect&bpo=20499]: Improve the speed and accuracy of statistics.pvariance().

- bpo-45132 [https://bugs.python.org/issue?@action=redirect&bpo=45132]: Remove **`__getitem__()`** methods of **`xml.dom.pulldom.DOMEventStream`**, **`wsgiref.util.FileWrapper`** and **`fileinput.FileInput`**, deprecated since Python 3.9.

  Patch by Hugo van Kemenade.

- bpo-45129 [https://bugs.python.org/issue?@action=redirect&bpo=45129]: Due to significant security concerns, the *reuse_address* parameter of **`asyncio.loop.create_datagram_endpoint()`**, disabled in Python 3.9, is now entirely removed. This is because of the behavior of the socket option `SO_REUSEADDR` in UDP.

  Patch by Hugo van Kemenade.

- bpo-45124 [https://bugs.python.org/issue?@action=redirect&bpo=45124]: The `bdist_msi` command, deprecated in Python 3.9, is now removed.

  Use `bdist_wheel` (wheel packages) instead.

  Patch by Hugo van Kemenade.

- bpo-30856 [https://bugs.python.org/issue?@action=redirect&bpo=30856]: **`unittest.TestResult`** methods **`addFailure()`**, **`addError()`**, **`addSkip()`** and **`addSubTest()`** are now called immediately after raising an exception in test or finishing a subtest. Previously they were called only after finishing the test clean up.

- bpo-45034 [https://bugs.python.org/issue?@action=redirect&bpo=45034]: Changes how error is formatted for `struct.pack` with `'H'` and `'h'` modes and too large / small numbers. Now it shows the actual numeric limits, while previously it was showing arithmetic expressions.

- bpo-25894 [https://bugs.python.org/issue?@action=redirect&bpo=25894]: **`unittest`** now always reports skipped and failed subtests separately: separate characters in default mode and separate lines in verbose mode. Also the test description is now output for errors in test method, class and module cleanups.

- bpo-45081 [https://bugs.python.org/issue?@action=redirect&bpo=45081]: Fix issue when dataclasses that inherit from `typing.Protocol` subclasses have wrong `__init__`. Patch provided by Yurii Karabas.

- bpo-45085 [https://bugs.python.org/issue?@action=redirect&bpo=45085]: The `binhex` module, deprecated in Python 3.9, is now removed. The following **`binascii`** functions, deprecated in Python 3.9, are now also removed:

    - `a2b_hqx()`, `b2a_hqx()`;
    - `rlecode_hqx()`, `rledecode_hqx()`.

  The **`binascii.crc_hqx()`** function remains available.

  Patch by Victor Stinner.

- bpo-40360 [https://bugs.python.org/issue?@action=redirect&bpo=40360]: The **`lib2to3`** package is now deprecated and may not be able to parse Python 3.10 or newer. See the **PEP 617** [https://peps.python.org/pep-0617/] (New PEG parser for CPython). Patch by Victor Stinner.

- bpo-45075 [https://bugs.python.org/issue?@action=redirect&bpo=45075]: Rename `traceback.StackSummary.format_frame()` to `traceback.StackSummary.format_frame_summary()`. This method was added for 3.11 so it was not released yet.

  Updated code and docs to better distinguish frame and FrameSummary.

- bpo-31299 [https://bugs.python.org/issue?@action=redirect&bpo=31299]: Add option to completely drop frames from a traceback by returning `None` from a `format_frame()` override.

- bpo-41620 [https://bugs.python.org/issue?@action=redirect&bpo=41620]: `run()` now always return a `TestResult` instance. Previously it returned `None` if the test class or method was decorated with a skipping decorator.

- bpo-45021 [https://bugs.python.org/issue?@action=redirect&bpo=45021]: Fix a potential deadlock at shutdown of forked children when using `concurrent.futures` module

- bpo-43913 [https://bugs.python.org/issue?@action=redirect&bpo=43913]: Fix bugs in cleaning up classes and modules in `unittest`:

  - Functions registered with `addModuleCleanup()` were not called unless the user defines `tearDownModule()` in their test module.

  - Functions registered with `addClassCleanup()` were not called if `tearDownClass` is set to `None`.
  - Buffering in `TestResult` did not work with functions registered with `addClassCleanup()` and `addModuleCleanup()`.
  - Errors in functions registered with `addClassCleanup()` and `addModuleCleanup()` were not handled correctly in buffered and debug modes.
  - Errors in `setUpModule()` and functions registered with `addModuleCleanup()` were reported in wrong order.
  - And several lesser bugs.

- bpo-45030 [https://bugs.python.org/issue?@action=redirect&bpo=45030]: Fix integer overflow in pickling and copying the range iterator.

- bpo-45001 [https://bugs.python.org/issue?@action=redirect&bpo=45001]: Made email date parsing more robust against malformed input, namely a whitespace-only `Date:` header. Patch by Wouter Bolsterlee.

- bpo-45010 [https://bugs.python.org/issue?@action=redirect&bpo=45010]: Remove support of special method `__div__` in `unittest.mock`. It is not used in Python 3.

- bpo-39218 [https://bugs.python.org/issue?@action=redirect&bpo=39218]: Improve accuracy of variance calculations by using `x*x` instead of `x**2`.

- bpo-43613 [https://bugs.python.org/issue?@action=redirect&bpo=43613]: Improve the speed of `gzip.compress()` and `gzip.decompress()` by compressing and decompressing at once in memory instead of in a streamed fashion.

- bpo-37596 [https://bugs.python.org/issue?@action=redirect&bpo=37596]: Ensure that `set` and `frozenset` objects are always `marshalled` reproducibly.

- bpo-44019 [https://bugs.python.org/issue?@action=redirect&bpo=44019]: A new function `operator.call` has been added, such that `operator.call(obj, *args, **kwargs) == obj(*args, **kwargs)`.

- bpo-42255 [https://bugs.python.org/issue?@action=redirect&bpo=42255]: **`webbrowser.MacOSX`** is deprecated and will be removed in Python 3.13. It is untested and undocumented and also not used by `webbrowser` itself. Patch by Donghee Na.

- bpo-44955 [https://bugs.python.org/issue?@action=redirect&bpo=44955]: Method `stopTestRun()` is now always called in pair with method `startTestRun()` for `TestResult` objects implicitly created in `run()`. Previously it was not called for test methods and classes decorated with a skipping decorator.

- bpo-39039 [https://bugs.python.org/issue?@action=redirect&bpo=39039]: tarfile.open raises `ReadError` when a zlib error occurs during file extraction.

- bpo-44935 [https://bugs.python.org/issue?@action=redirect&bpo=44935]: `subprocess` on Solaris now also uses `os.posix_spawn()` for better performance.

- bpo-44911 [https://bugs.python.org/issue?@action=redirect&bpo=44911]: `IsolatedAsyncioTestCase` will no longer throw an exception while cancelling leaked tasks. Patch by Bar Harel.

- bpo-41322 [https://bugs.python.org/issue?@action=redirect&bpo=41322]: Added `DeprecationWarning` for tests and async tests that return a value!=None (as this may indicate an improperly written test, for example a test written as a generator function).

- bpo-44524 [https://bugs.python.org/issue?@action=redirect&bpo=44524]: Make exception message more useful when subclass from typing special form alias. Patch provided by Yurii Karabas.

- bpo-38956 [https://bugs.python.org/issue?@action=redirect&bpo=38956]: **`argparse.BooleanOptionalAction`**’s default value is no longer printed twice when used with **`argparse.ArgumentDefaultsHelpFormatter`**.

- bpo-44860 [https://bugs.python.org/issue?@action=redirect&bpo=44860]: Fix the `posix_user` scheme in **`sysconfig`** to not depend on **`sys.platlibdir`**.

- bpo-44859 [https://bugs.python.org/issue?@action=redirect&bpo=44859]: Improve error handling in **`sqlite3`** and raise more accurate exceptions.

    - **`MemoryError`** is now raised instead of **`sqlite3.Warning`** when memory is not enough for encoding a statement to UTF-8 in `Connection.__call__()` and `Cursor.execute()`.
    - **`UnicodEncodeError`** is now raised instead of **`sqlite3.Warning`** when the statement contains surrogate characters in `Connection.__call__()` and `Cursor.execute()`.
    - **`TypeError`** is now raised instead of **`ValueError`** for non-string script argument in `Cursor.executescript()`.
    - **`ValueError`** is now raised for script containing the null character instead of truncating it in `Cursor.executescript()`.

  - Correctly handle exceptions raised when getting boolean value of the result of the progress handler.
  - Add many tests covering different corner cases.

- bpo-44581 [https://bugs.python.org/issue?@action=redirect&bpo=44581]: Upgrade bundled pip to 21.2.3 and setuptools to 57.4.0

- bpo-44849 [https://bugs.python.org/issue?@action=redirect&bpo=44849]: Fix the **`os.set_inheritable()`** function on FreeBSD 14 for file descriptor opened with the **`O_PATH`** flag: ignore the **`EBADF`** error on `ioctl()`, fallback on the `fcntl()` implementation. Patch by Victor Stinner.

- bpo-44605 [https://bugs.python.org/issue?@action=redirect&bpo=44605]: The @functools.total_ordering() decorator now works with metaclasses.

- bpo-44524 [https://bugs.python.org/issue?@action=redirect&bpo=44524]: Fixed an issue wherein the `__name__` and `__qualname__` attributes of subscribed specialforms could be `None`.

- bpo-44839 [https://bugs.python.org/issue?@action=redirect&bpo=44839]: **`MemoryError`** raised in user-defined functions will now produce a `MemoryError` in **`sqlite3`**. **`OverflowError`** will now be converted to **`DataError`**. Previously **`OperationalError`** was produced in these cases.

- bpo-44822 [https://bugs.python.org/issue?@action=redirect&bpo=44822]: **`sqlite3`** user-defined functions and aggregators returning **`strings`** with embedded NUL characters are no longer truncated. Patch by Erlend E. Aasland.

- bpo-44801 [https://bugs.python.org/issue?@action=redirect&bpo=44801]: Ensure that the **`ParamSpec`** variable in Callable can only be substituted with a parameters expression (a list of types, an ellipsis, ParamSpec or Concatenate).

- bpo-44806 [https://bugs.python.org/issue?@action=redirect&bpo=44806]: Non-protocol subclasses of **`typing.Protocol`** ignore now the `__init__` method inherited from protocol base classes.

- bpo-27275 [https://bugs.python.org/issue?@action=redirect&bpo=27275]: **`collections.OrderedDict.popitem()`** and **`collections.OrderedDict.pop()`** no longer call `__getitem__` and `__delitem__` methods of the OrderedDict subclasses.

- bpo-44793 [https://bugs.python.org/issue?@action=redirect&bpo=44793]: Fix checking the number of arguments when subscribe a generic type with `ParamSpec` parameter.

- bpo-44784 [https://bugs.python.org/issue?@action=redirect&bpo=44784]: In importlib.metadata tests, override warnings behavior under expected DeprecationWarnings (importlib_metadata 4.6.3).

- bpo-44667 [https://bugs.python.org/issue?@action=redirect&bpo=44667]: The **`tokenize.tokenize()`** doesn’t incorrectly generate a `NEWLINE` token if the source doesn’t end with a new line character but the last line is a comment, as the function is already generating a `NL` token. Patch by Pablo Galindo

- bpo-44771 [https://bugs.python.org/issue?@action=redirect&bpo=44771]: Added `importlib.simple` module implementing adapters from a low-level resources reader interface to a `TraversableResources` interface. Legacy API (`path`, `contents`, …) is now supported entirely by the `.files()` API with a compatibility shim supplied for resource loaders without that functionality. Feature parity with `importlib_resources` 5.2.

- bpo-44752 [https://bugs.python.org/issue?@action=redirect&bpo=44752]: **`rcompleter`** does not call **`getattr()`** on **`property`** objects to avoid the side-effect of evaluating the corresponding method.

- bpo-44747 [https://bugs.python.org/issue?@action=redirect&bpo=44747]: Refactor usage of `sys._getframe` in `typing` module. Patch provided by Yurii Karabas.

- bpo-42378 [https://bugs.python.org/issue?@action=redirect&bpo=42378]: Fixes the issue with log file being overwritten when **`logging.FileHandler`** is used in **`atexit`** with *filemode* set to `'w'`. Note this will cause the message in *atexit* not being logged if the log stream is already closed due to shutdown of logging.

- bpo-44720 [https://bugs.python.org/issue?@action=redirect&bpo=44720]: `weakref.proxy` objects referencing non-iterators now raise `TypeError` rather than dereferencing the null `tp_iternext` slot and crashing.

- bpo-44704 [https://bugs.python.org/issue?@action=redirect&bpo=44704]: The implementation of `collections.abc.Set._hash()` now matches that of `frozenset.__hash__()`.

- bpo-44666 [https://bugs.python.org/issue?@action=redirect&bpo=44666]: Fixed issue in **`compileall.compile_file()`** when `sys.stdout` is redirected. Patch by Stefan Hölzl.

- bpo-44688 [https://bugs.python.org/issue?@action=redirect&bpo=44688]:

`sqlite3.Connection.create_collation()` now accepts non-ASCII collation names. Patch by Erlend E. Aasland.

- bpo-44690 [https://bugs.python.org/issue?@action=redirect&bpo=44690]: Adopt *binacii.a2b_base64*'s strict mode in *base64.b64decode*.
- bpo-42854 [https://bugs.python.org/issue?@action=redirect&bpo=42854]: Fixed a bug in the **`_ssl`** module that was throwing `OverflowError` when using **`_ssl._SSLSocket.write()`** and **`_ssl._SSLSocket.read()`** for a big value of the `len` parameter. Patch by Pablo Galindo.
- bpo-44686 [https://bugs.python.org/issue?@action=redirect&bpo=44686]: Replace `unittest.mock._importer` with `pkgutil.resolve_name`.
- bpo-44353 [https://bugs.python.org/issue?@action=redirect&bpo=44353]: Make `NewType.__call__` faster by implementing it in C. Patch provided by Yurii Karabas.
- bpo-44682 [https://bugs.python.org/issue?@action=redirect&bpo=44682]: Change the `pdb` *commands* directive to disallow setting commands for an invalid breakpoint and to display an appropriate error.
- bpo-44353 [https://bugs.python.org/issue?@action=redirect&bpo=44353]: Refactor `typing.NewType` from function into callable class. Patch provided by Yurii Karabas.
- bpo-44678 [https://bugs.python.org/issue?@action=redirect&bpo=44678]: Added a separate error message for discontinuous padding in *binascii.a2b_base64* strict mode.

- bpo-44524 [https://bugs.python.org/issue?@action=redirect&bpo=44524]: Add missing `__name__` and `__qualname__` attributes to `typing` module classes. Patch provided by Yurii Karabas.

- bpo-40897 [https://bugs.python.org/issue?@action=redirect&bpo=40897]: Give priority to using the current class constructor in **`inspect.signature()`**. Patch by Weipeng Hong.

- bpo-44638 [https://bugs.python.org/issue?@action=redirect&bpo=44638]: Add a reference to the zipp project and hint as to how to use it.

- bpo-44648 [https://bugs.python.org/issue?@action=redirect&bpo=44648]: Fixed wrong error being thrown by **`inspect.getsource()`** when examining a class in the interactive session. Instead of **`TypeError`**, it should be **`OSError`** with appropriate error message.

- bpo-44608 [https://bugs.python.org/issue?@action=redirect&bpo=44608]: Fix memory leak in **`_tkinter._flatten()`** if it is called with a sequence or set, but not list or tuple.

- bpo-44594 [https://bugs.python.org/issue?@action=redirect&bpo=44594]: Fix an edge case of **`ExitStack`** and **`AsyncExitStack`** exception chaining. They will now match `with` block behavior when `__context__` is explicitly set to `None` when the exception is in flight.

- bpo-42799 [https://bugs.python.org/issue?@action=redirect&bpo=42799]: In **`fnmatch`**, the cache size for compiled regex patterns (**`functools.lru_cache()`**) was bumped up from 256 to 32768, affecting functions: **`fnmatch.fnmatch()`**, **`fnmatch.fnmatchcase()`**, **`fnmatch.filter()`**.

- bpo-41928 [https://bugs.python.org/issue?@action=redirect&bpo=41928]: Update `shutil.copyfile()` to raise `FileNotFoundError` instead of confusing `IsADirectoryError` when a path ending with a `os.path.sep` does not exist; `shutil.copy()` and `shutil.copy2()` are also affected.

- bpo-44569 [https://bugs.python.org/issue?@action=redirect&bpo=44569]: Added the `StackSummary.format_frame()` function in `traceback`. This allows users to customize the way individual lines are formatted in tracebacks without re-implementing logic to handle recursive tracebacks.

- bpo-44566 [https://bugs.python.org/issue?@action=redirect&bpo=44566]: handle StopIteration subclass raised from @contextlib.contextmanager generator

- bpo-44558 [https://bugs.python.org/issue?@action=redirect&bpo=44558]: Make the implementation consistency of `indexOf()` between C and Python versions. Patch by Donghee Na.

- bpo-41249 [https://bugs.python.org/issue?@action=redirect&bpo=41249]: Fixes `TypedDict` to work with `typing.get_type_hints()` and postponed evaluation of annotations across modules.

- bpo-44554 [https://bugs.python.org/issue?@action=redirect&bpo=44554]: Refactor argument processing in `pdb.main()` to simplify detection of errors in input loading and clarify behavior around module or script invocation.

- bpo-34798 [https://bugs.python.org/issue?@action=redirect&bpo=34798]: Break up paragraph about `pprint.PrettyPrinter` construction parameters to make it easier to read.

- bpo-44539 [https://bugs.python.org/issue?@action=redirect&bpo=44539]: Added support for recognizing JPEG files without JFIF or Exif markers.

- bpo-44461 [https://bugs.python.org/issue?@action=redirect&bpo=44461]: Fix bug with `pdb`'s handling of import error due to a package which does not have a `__main__` module

- bpo-43625 [https://bugs.python.org/issue?@action=redirect&bpo=43625]: Fix a bug in the detection of CSV file headers by `csv.Sniffer.has_header()` and improve documentation of same.

- bpo-44516 [https://bugs.python.org/issue?@action=redirect&bpo=44516]: Update vendored pip to 21.1.3

- bpo-42892 [https://bugs.python.org/issue?@action=redirect&bpo=42892]: Fixed an exception thrown while parsing a malformed multipart email by `email.message.EmailMessage`.

- bpo-44468 [https://bugs.python.org/issue?@action=redirect&bpo=44468]: `typing.get_type_hints()` now finds annotations in classes and base classes with unexpected `__module__`. Previously, it skipped those MRO elements.

- bpo-44491 [https://bugs.python.org/issue?@action=redirect&bpo=44491]: Allow clearing the `sqlite3` authorizer callback by passing `None` to `set_authorizer()`. Patch by Erlend E. Aasland.

- bpo-43977 [https://bugs.python.org/issue?@action=redirect&bpo=43977]: Set the proper **`Py_TPFLAGS_MAPPING`** and **`Py_TPFLAGS_SEQUENCE`** flags for subclasses created before a parent has been registered as a **`collections.abc.Mapping`** or **`collections.abc.Sequence`**.

- bpo-44482 [https://bugs.python.org/issue?@action=redirect&bpo=44482]: Fix very unlikely resource leak in **`glob`** in alternate Python implementations.

- bpo-44466 [https://bugs.python.org/issue?@action=redirect&bpo=44466]: The **`faulthandler`** module now detects if a fatal error occurs during a garbage collector collection. Patch by Victor Stinner.

- bpo-44471 [https://bugs.python.org/issue?@action=redirect&bpo=44471]: A **`TypeError`** is now raised instead of an **`AttributeError`** in **`contextlib.ExitStack.enter_context()`** and **`contextlib.AsyncExitStack.enter_async_context()`** for objects which do not support the context manager or asynchronous context manager protocols correspondingly.

- bpo-44404 [https://bugs.python.org/issue?@action=redirect&bpo=44404]: **`tkinter`**’s `after()` method now supports callables without the `__name__` attribute.

- bpo-41546 [https://bugs.python.org/issue?@action=redirect&bpo=41546]: Make **`pprint`** (like the builtin `print`) not attempt to write to `stdout` when it is `None`.

- bpo-44458 [https://bugs.python.org/issue?@action=redirect&bpo=44458]: `BUFFER_BLOCK_SIZE` is now declared static, to avoid linking collisions when bz2, lmza or zlib are statically linked.

- bpo-44464 [https://bugs.python.org/issue?@action=redirect&bpo=44464]: Remove exception for flake8 in deprecated importlib.metadata interfaces. Sync with importlib_metadata 4.6.

- bpo-44446 [https://bugs.python.org/issue?@action=redirect&bpo=44446]: Take into account that `lineno` might be `None` in **`traceback.FrameSummary`**.

- bpo-44439 [https://bugs.python.org/issue?@action=redirect&bpo=44439]: Fix in **`bz2.BZ2File.write()`** / **`lzma.LZMAFile.write()`** methods, when the input data is an object that supports the buffer protocol, the file length may be wrong.

- bpo-44434 [https://bugs.python.org/issue?@action=redirect&bpo=44434]: _thread.start_new_thread() no longer calls PyThread_exit_thread() explicitly at the thread exit, the call was redundant. On Linux with the glibc, pthread_exit() aborts the whole process if dlopen() fails to open libgcc_s.so file (ex: EMFILE error). Patch by Victor Stinner.

- bpo-42972 [https://bugs.python.org/issue?@action=redirect&bpo=42972]: The _thread.RLock type now fully implement the GC protocol: add a traverse function and the **`Py_TPFLAGS_HAVE_GC`** flag. Patch by Victor Stinner.

- bpo-44422 [https://bugs.python.org/issue?@action=redirect&bpo=44422]: The **`threading.enumerate()`** function now uses a reentrant lock to prevent a hang on reentrant call. Patch by Victor Stinner.

- bpo-38291 [https://bugs.python.org/issue?@action=redirect&bpo=38291]: Importing typing.io or typing.re now prints a `DeprecationWarning`.

- bpo-37880 [https://bugs.python.org/issue?@action=redirect&bpo=37880]: argparse actions store_const and append_const each receive a default value of `None` when the `const` kwarg is not provided. Previously, this raised a `TypeError`.

- bpo-44389 [https://bugs.python.org/issue?@action=redirect&bpo=44389]: Fix deprecation of **`ssl.OP_NO_TLSv1_3`**

- bpo-27827 [https://bugs.python.org/issue?@action=redirect&bpo=27827]: **`pathlib.PureWindowsPath.is_reserved()`** now identifies a greater range of reserved filenames, including those with trailing spaces or colons.

- bpo-44395 [https://bugs.python.org/issue?@action=redirect&bpo=44395]: Fix **`as_string()`** to pass unixfrom properly. Patch by Donghee Na.

- bpo-34266 [https://bugs.python.org/issue?@action=redirect&bpo=34266]: Handle exceptions from parsing the arg of **`pdb`**’s run/restart command.

- bpo-44362 [https://bugs.python.org/issue?@action=redirect&bpo=44362]: Improve **`ssl`** module’s deprecation messages, error reporting, and documentation for deprecations.

- bpo-44342 [https://bugs.python.org/issue?@action=redirect&bpo=44342]: [Enum] Change pickling from by-value to by-name.

- bpo-44356 [https://bugs.python.org/issue?@action=redirect&bpo=44356]: [Enum] Allow multiple data-type mixins if they are all the same.

- bpo-44351 [https://bugs.python.org/issue?@action=redirect&bpo=44351]: Restore back **`parse_makefile()`** in `distutils.sysconfig` because it behaves differently than the similar implementation in **`sysconfig`**.

- bpo-35800 [https://bugs.python.org/issue?@action=redirect&bpo=35800]: **`smtpd.MailmanProxy`** is now removed as it is unusable without an external module, `mailman`. Patch by Donghee Na.

- bpo-44357 [https://bugs.python.org/issue?@action=redirect&bpo=44357]: Added a function that returns cube root of the given number **`math.cbrt()`**

- bpo-44339 [https://bugs.python.org/issue?@action=redirect&bpo=44339]: Change `math.pow(±0.0, -math.inf)` to return `inf` instead of raising `ValueError`. This brings the special-case handling of `math.pow` into compliance with the IEEE 754 standard.

- bpo-44242 [https://bugs.python.org/issue?@action=redirect&bpo=44242]: Remove missing flag check from Enum creation and move into a `verify` decorator.

- bpo-44246 [https://bugs.python.org/issue?@action=redirect&bpo=44246]: In `importlib.metadata`, restore compatibility in the result from `Distribution.entry_points` (`EntryPoints`) to honor expectations in older implementations and issuing deprecation warnings for these cases: A. `EntryPoints` objects are once again mutable, allowing for `sort()` and other list-based mutation operations. Avoid deprecation warnings by casting to a mutable sequence (e.g. `list(dist.entry_points).sort()`). B. `EntryPoints` results once again allow for access by index. To avoid deprecation warnings, cast the result to a Sequence first (e.g. `tuple(dist.entry_points)[0]`).

- bpo-44246 [https://bugs.python.org/issue?@action=redirect&bpo=44246]: In importlib.metadata.entry_points, de-duplication of distributions no longer requires loading the full metadata for PathDistribution objects, improving entry point loading performance by ~10x.

- bpo-43858 [https://bugs.python.org/issue?@action=redirect&bpo=43858]: Added a function that returns a copy of a dict of logging levels: **`logging.getLevelNamesMapping()`**

- bpo-44260 [https://bugs.python.org/issue?@action=redirect&bpo=44260]: The **`random.Random`** constructor no longer reads system entropy without need.

- bpo-44254 [https://bugs.python.org/issue?@action=redirect&bpo=44254]: On Mac, give turtledemo button text a color that works on both light or dark background. Programmers cannot control the latter.

- bpo-44258 [https://bugs.python.org/issue?@action=redirect&bpo=44258]: Support PEP 515 for Fraction’s initialization from string.

- bpo-44235 [https://bugs.python.org/issue?@action=redirect&bpo=44235]: Remove deprecated functions in the **`gettext`**. Patch by Donghee Na.

- bpo-38693 [https://bugs.python.org/issue?@action=redirect&bpo=38693]: Prefer f-strings to `.format` in importlib.resources.

- bpo-33693 [https://bugs.python.org/issue?@action=redirect&bpo=33693]: Importlib.metadata now prefers f-strings to .format.

- bpo-44241 [https://bugs.python.org/issue?@action=redirect&bpo=44241]: Incorporate minor tweaks from importlib_metadata 4.1: SimplePath protocol, support for Metadata 2.2.

- bpo-43216 [https://bugs.python.org/issue?@action=redirect&bpo=43216]: Remove the **`@asyncio.coroutine`** decorator enabling legacy generator-based coroutines to be compatible with async/await code; remove **`asyncio.coroutines.CoroWrapper`** used for wrapping legacy coroutine objects in the debug mode. The decorator has been deprecated since Python 3.8 and the removal was initially scheduled for Python 3.10. Patch by Illia Volochii.

- bpo-44210 [https://bugs.python.org/issue?@action=redirect&bpo=44210]: Make importlib.metadata._meta.PackageMetadata public.

- bpo-43643 [https://bugs.python.org/issue?@action=redirect&bpo=43643]: Declare readers.MultiplexedPath.name as a property per the spec.

- bpo-27334 [https://bugs.python.org/issue?@action=redirect&bpo=27334]: The **`sqlite3`** context manager now performs a rollback (thus releasing the database lock) if commit failed. Patch by Luca Citi and Erlend E. Aasland.

- bpo-4928 [https://bugs.python.org/issue?@action=redirect&bpo=4928]: Documented existing behavior on POSIX:

NamedTemporaryFiles are not deleted when creating process is killed with SIGKILL

- bpo-44154 [https://bugs.python.org/issue?@action=redirect&bpo=44154]: Optimize **`fractions.Fraction`** pickling for large components.

- bpo-33433 [https://bugs.python.org/issue?@action=redirect&bpo=33433]: For IPv4 mapped IPv6 addresses (**RFC 4291** [https://datatracker.ietf.org/doc/html/rfc4291.html] Section 2.5.5.2), the **`ipaddress.IPv6Address.is_private`** check is deferred to the mapped IPv4 address. This solves a bug where public mapped IPv4 addresses were considered private by the IPv6 check.

- bpo-44150 [https://bugs.python.org/issue?@action=redirect&bpo=44150]: Add optional *weights* argument to statistics.fmean().

- bpo-44142 [https://bugs.python.org/issue?@action=redirect&bpo=44142]: **`ast.unparse()`** will now drop the redundant parentheses when tuples used as assignment targets (e.g in for loops).

- bpo-44145 [https://bugs.python.org/issue?@action=redirect&bpo=44145]: **`hmac`** computations were not releasing the GIL while calling the OpenSSL `HMAC_Update` C API (a new feature in 3.9). This unintentionally prevented parallel computation as other **`hashlib`** algorithms support.

- bpo-44095 [https://bugs.python.org/issue?@action=redirect&bpo=44095]: **`zipfile.Path`** now supports **`zipfile.Path.stem`**, **`zipfile.Path.suffixes`**, and **`zipfile.Path.suffix`** attributes.

- bpo-44077 [https://bugs.python.org/issue?@action=redirect&bpo=44077]: It’s now possible to receive the type of service (ToS), a.k.a. differentiated services (DS), a.k.a. differentiated services code point (DSCP) and explicit congestion notification (ECN) IP header fields with `socket.IP_RECVTOS`.

- bpo-37788 [https://bugs.python.org/issue?@action=redirect&bpo=37788]: Fix a reference leak when a Thread object is never joined.

- bpo-38908 [https://bugs.python.org/issue?@action=redirect&bpo=38908]: Subclasses of `typing.Protocol` which only have data variables declared will now raise a `TypeError` when checked with `isinstance` unless they are decorated with **`runtime_checkable()`**. Previously, these checks passed silently. Patch provided by Yurii Karabas.

- bpo-44098 [https://bugs.python.org/issue?@action=redirect&bpo=44098]: `typing.ParamSpec` will no longer be found in the `__parameters__` of most **`typing`** generics except in valid use locations specified by **PEP 612** [https://peps.python.org/pep-0612/]. This prevents incorrect usage like `typing.List[P][int]`. This change means incorrect usage which may have passed silently in 3.10 beta 1 and earlier will now error.

- bpo-44089 [https://bugs.python.org/issue?@action=redirect&bpo=44089]: Allow subclassing `csv.Error` in 3.10 (it was allowed in 3.9 and earlier but was disallowed in early versions of 3.10).

- bpo-44081 [https://bugs.python.org/issue?@action=redirect&bpo=44081]: **`ast.unparse()`** now doesn’t use redundant spaces to separate `lambda` and the `:` if there are no parameters.

- bpo-44061 [https://bugs.python.org/issue?@action=redirect&bpo=44061]: Fix regression in previous release when calling **`pkgutil.iter_modules()`** with a list of **`pathlib.Path`** objects

- bpo-44059 [https://bugs.python.org/issue?@action=redirect&bpo=44059]: Register the SerenityOS Browser in the **`webbrowser`** module.

- bpo-36515 [https://bugs.python.org/issue?@action=redirect&bpo=36515]: The **`hashlib`** module no longer does unaligned memory accesses when compiled for ARM platforms.

- bpo-40465 [https://bugs.python.org/issue?@action=redirect&bpo=40465]: Remove random module features deprecated in Python 3.9.

- bpo-44018 [https://bugs.python.org/issue?@action=redirect&bpo=44018]: random.seed() no longer mutates bytearray inputs.

- bpo-38352 [https://bugs.python.org/issue?@action=redirect&bpo=38352]: Add `IO`, `BinaryIO`, `TextIO`, `Match`, and `Pattern` to `typing.__all__`. Patch by Jelle Zijlstra.

- bpo-44002 [https://bugs.python.org/issue?@action=redirect&bpo=44002]: **`urllib.parse`** now uses **`functool.lru_cache()`** for its internal URL splitting and quoting caches instead of rolling its own like its the ’90s.

  The undocumented internal **`urllib.parse`** `Quoted` class API is now deprecated, for removal in 3.14.

- bpo-43972 [https://bugs.python.org/issue?@action=redirect&bpo=43972]: When **`http.server.SimpleHTTPRequestHandler`** sends a `301 (Moved`

`Permanently)` for a directory path not ending with `/`, add a `Content-Length: 0` header. This improves the behavior for certain clients.

- bpo-28528 [https://bugs.python.org/issue?@action=redirect&bpo=28528]: Fix a bug in **`pdb`** where **`checkline()`** raises **`AttributeError`** if it is called after **`reset()`**.

- bpo-43853 [https://bugs.python.org/issue?@action=redirect&bpo=43853]: Improved string handling for **`sqlite3`** user-defined functions and aggregates:

  - It is now possible to pass strings with embedded null characters to UDFs
  - Conversion failures now correctly raise **`MemoryError`**

  Patch by Erlend E. Aasland.

- bpo-43666 [https://bugs.python.org/issue?@action=redirect&bpo=43666]: AIX: `Lib/_aix_support.get_platform()` may fail in an AIX WPAR. The fileset bos.rte appears to have a builddate in both LPAR and WPAR so this fileset is queried rather than bos.mp64. To prevent a similar situation (no builddate in ODM) a value (9988) sufficient for completing a build is provided. Patch by M Felt.

- bpo-43650 [https://bugs.python.org/issue?@action=redirect&bpo=43650]: Fix **`MemoryError`** in **`shutil.unpack_archive()`** which fails inside **`shutil._unpack_zipfile()`** on large files. Patch by Igor Bolshakov.

- bpo-43612 [https://bugs.python.org/issue?@action=redirect&bpo=43612]: **`zlib.compress()`** now accepts a wbits parameter which allows users to compress data as a raw deflate block without zlib headers and trailers in one go. Previously this required instantiating a `zlib.compressobj`. It also provides a faster alternative to `gzip.compress` when wbits=31 is used.

- bpo-43392 [https://bugs.python.org/issue?@action=redirect&bpo=43392]: **`importlib._bootstrap._find_and_load()`** now implements a two-step check to avoid locking when modules have been already imported and are ready. This improves performance of repeated calls to **`importlib.import_module()`** and **`importlib.__import__()`**.

- bpo-43318 [https://bugs.python.org/issue?@action=redirect&bpo=43318]: Fix a bug where **`pdb`** does not always echo cleared breakpoints.

- bpo-43234 [https://bugs.python.org/issue?@action=redirect&bpo=43234]: Prohibit passing non-**`concurrent.futures.ThreadPoolExecutor`** executors to **`loop.set_default_executor()`** following a deprecation in Python 3.8. Patch by Illia Volochii.

- bpo-43232 [https://bugs.python.org/issue?@action=redirect&bpo=43232]: Prohibit previously deprecated potentially disruptive operations on **`asyncio.trsock.TransportSocket`**. Patch by Illia Volochii.

- bpo-30077 [https://bugs.python.org/issue?@action=redirect&bpo=30077]: Added support for Apple’s aifc/sowt pseudo-compression

- bpo-42971 [https://bugs.python.org/issue?@action=redirect&bpo=42971]: Add definition of `errno.EQFULL` for platforms that define this constant (such as macOS).

- bpo-43086 [https://bugs.python.org/issue?@action=redirect&bpo=43086]: Added a new optional `strict_mode` parameter to *binascii.a2b_base64*. When `scrict_mode` is set to `True`, the *a2b_base64* function will accept only valid base64 content. More details about what “valid base64 content” is, can be found in the function’s documentation.

- bpo-43024 [https://bugs.python.org/issue?@action=redirect&bpo=43024]: Improve the help signature of **`traceback.print_exception()`**, **`traceback.format_exception()`** and **`traceback.format_exception_only()`**.

- bpo-33809 [https://bugs.python.org/issue?@action=redirect&bpo=33809]: Add the **`traceback.TracebackException.print()`** method which prints the formatted exception information.

- bpo-42862 [https://bugs.python.org/issue?@action=redirect&bpo=42862]: **`sqlite3`** now utilizes **`functools.lru_cache()`** to implement the connection statement cache. As a small optimisation, the default statement cache size has been increased from 100 to 128. Patch by Erlend E. Aasland.

- bpo-41818 [https://bugs.python.org/issue?@action=redirect&bpo=41818]: Soumendra Ganguly: add termios.tcgetwinsize(), termios.tcsetwinsize().

- bpo-40497 [https://bugs.python.org/issue?@action=redirect&bpo=40497]: **`subprocess.check_output()`** now raises **`ValueError`** when the invalid keyword argument *check* is passed by user code. Previously such use would fail later with a **`TypeError`**. Patch by Rémi Lapeyre.

- bpo-37449 [https://bugs.python.org/issue?@action=redirect&bpo=37449]: `ensurepip` now uses `importlib.resources.files()` traversable APIs

- bpo-40956 [https://bugs.python.org/issue?@action=redirect&bpo=40956]: Use Argument Clinic in **`sqlite3`**. Patches by Erlend E. Aasland.

- bpo-41730 [https://bugs.python.org/issue?@action=redirect&bpo=41730]: `DeprecationWarning` is now raised when importing **`tkinter.tix`**, which has been deprecated in documentation since Python 3.6.

- bpo-20684 [https://bugs.python.org/issue?@action=redirect&bpo=20684]: Remove unused `_signature_get_bound_param` function from **`inspect`** - by Anthony Sottile.

- bpo-41402 [https://bugs.python.org/issue?@action=redirect&bpo=41402]: Fix **`email.message.EmailMessage.set_content()`** when called with binary data and `7bit` content transfer encoding.

- bpo-32695 [https://bugs.python.org/issue?@action=redirect&bpo=32695]: The *compresslevel* and *preset* keyword arguments of **`tarfile.open()`** are now both documented and tested.

- bpo-41137 [https://bugs.python.org/issue?@action=redirect&bpo=41137]: Use utf-8 encoding while reading .pdbrc files. Patch by Srinivas Reddy Thatiparthy

- bpo-24391 [https://bugs.python.org/issue?@action=redirect&bpo=24391]: Improved reprs of **`threading`** synchronization objects: **`Semaphore`**, **`BoundedSemaphore`**, **`Event`** and **`Barrier`**.

- bpo-5846 [https://bugs.python.org/issue?@action=redirect&bpo=5846]: Deprecated the following **`unittest`** functions, scheduled for removal in Python 3.13:

  - **`findTestCases()`**
  - **`makeSuite()`**
  - **`getTestCaseNames()`**

  Use **`TestLoader`** methods instead:

  - **`unittest.TestLoader.loadTestsFromModule()`**
  - **`unittest.TestLoader.loadTestsFromTestCase()`**
  - **`unittest.TestLoader.getTestCaseNames()`**

  Patch by Erlend E. Aasland.

- bpo-40563 [https://bugs.python.org/issue?@action=redirect&bpo=40563]: Support pathlike objects on dbm/shelve. Patch by Hakan Çelik and Henry-Joseph Audéoud.

- bpo-34990 [https://bugs.python.org/issue?@action=redirect&bpo=34990]: Fixed a Y2k38 bug in the compileall module where it would fail to compile files with a modification time after the year 2038.

- bpo-39549 [https://bugs.python.org/issue?@action=redirect&bpo=39549]: Whereas the code for reprlib.Repr had previously used a hardcoded string value of '…', this PR updates it to use of a "fillvalue" attribute, whose value defaults to '…' and can be reset in either individual reprlib.Repr instances or in subclasses thereof.

- bpo-37022 [https://bugs.python.org/issue?@action=redirect&bpo=37022]: **`pdb`** now displays exceptions from `repr()` with its `p` and `pp` commands.

- bpo-38840 [https://bugs.python.org/issue?@action=redirect&bpo=38840]: Fix `test___all__` on platforms lacking a shared memory implementation.

- bpo-39359 [https://bugs.python.org/issue?@action=redirect&bpo=39359]: Add one missing check that the password is a bytes object for an encrypted zipfile.

- bpo-38741 [https://bugs.python.org/issue?@action=redirect&bpo=38741]: `configparser`: using ‘]’ inside a section header will no longer cut the section name short at the ‘]’

- bpo-38415 [https://bugs.python.org/issue?@action=redirect&bpo=38415]: Added missing behavior to `contextlib.asynccontextmanager()` to match `contextlib.contextmanager()` so decorated functions can themselves be decorators.

- bpo-30256 [https://bugs.python.org/issue?@action=redirect&bpo=30256]: Pass multiprocessing BaseProxy argument `manager_owned` through AutoProxy.

- bpo-27513 [https://bugs.python.org/issue?@action=redirect&bpo=27513]: `email.utils.getaddresses()` now accepts `email.header.Header` objects along with string values. Patch by Zackery Spytz.

- bpo-16379 [https://bugs.python.org/issue?@action=redirect&bpo=16379]: Add SQLite error code and name to `sqlite3` exceptions. Patch by Aviv Palivoda, Daniel Shahaf, and Erlend E. Aasland.

- bpo-26228 [https://bugs.python.org/issue?@action=redirect&bpo=26228]: pty.spawn no longer hangs on FreeBSD, macOS, and Solaris.

- bpo-33349 [https://bugs.python.org/issue?@action=redirect&bpo=33349]: lib2to3 now recognizes async generators everywhere.

- bpo-29298 [https://bugs.python.org/issue?@action=redirect&bpo=29298]: Fix `TypeError` when required subparsers without `dest` do not receive arguments. Patch by Anthony Sottile.

Documentation

- bpo-45216 [https://bugs.python.org/issue?@action=redirect&bpo=45216]: Remove extra documentation listing methods in `difflib`. It was rendering twice in pydoc and was outdated in some places.
- bpo-45024 [https://bugs.python.org/issue?@action=redirect&bpo=45024]: **`collections.abc`** documentation has been expanded to explicitly cover how instance and subclass checks work, with additional doctest examples and an exhaustive list of ABCs which test membership purely by presence of the right special methods. Patch by Raymond Hettinger.
- bpo-44957 [https://bugs.python.org/issue?@action=redirect&bpo=44957]: Promote PEP 604 union syntax by using it where possible. Also, mention `X | Y` more prominently in section about `Union` and mention `X | None` at all in section about `Optional`.
- bpo-16580 [https://bugs.python.org/issue?@action=redirect&bpo=16580]: Added code equivalents for the **`int.to_bytes()`** and **`int.from_bytes()`** methods, as well as tests ensuring that these code equivalents are valid.
- bpo-44903 [https://bugs.python.org/issue?@action=redirect&bpo=44903]: Removed the `othergui.rst` file, any references to it, and the list of GUI frameworks in the FAQ. In their place I’ve added links to the Python

Wiki page on GUI frameworks [https://wiki.python.org/moin/GuiProgramming].

- bpo-33479 [https://bugs.python.org/issue?@action=redirect&bpo=33479]: Tkinter documentation has been greatly expanded with new “Architecture” and “Threading model” sections.
- bpo-36700 [https://bugs.python.org/issue?@action=redirect&bpo=36700]: **`base64`** RFC references were updated to point to **RFC 4648** [https://datatracker.ietf.org/doc/html/rfc4648.html]; a section was added to point users to the new “security considerations” section of the RFC.
- bpo-44740 [https://bugs.python.org/issue?@action=redirect&bpo=44740]: Replaced occurrences of uppercase “Web” and “Internet” with lowercase versions per the 2016 revised Associated Press Style Book.
- bpo-44693 [https://bugs.python.org/issue?@action=redirect&bpo=44693]: Update the definition of __future__ in the glossary by replacing the confusing word “pseudo-module” with a more accurate description.
- bpo-35183 [https://bugs.python.org/issue?@action=redirect&bpo=35183]: Add typical examples to os.path.splitext docs
- bpo-30511 [https://bugs.python.org/issue?@action=redirect&bpo=30511]: Clarify that **`shutil.make_archive()`** is not thread-safe due to reliance on changing the current working directory.
- bpo-44561 [https://bugs.python.org/issue?@action=redirect&bpo=44561]: Update of three expired hyperlinks in Doc/distributing/index.rst: “Project structure”, “Building and packaging the project”, and “Uploading the project to the Python Packaging Index”.
- bpo-44651 [https://bugs.python.org/issue?@action=redirect&bpo=44651]: Delete entry “coercion” in Doc/glossary.rst for its outdated definition.

- bpo-42958 [https://bugs.python.org/issue?@action=redirect&bpo=42958]: Updated the docstring and docs of **`filecmp.cmp()`** to be more accurate and less confusing especially in respect to *shallow* arg.
- bpo-44631 [https://bugs.python.org/issue?@action=redirect&bpo=44631]: Refactored the `repr()` code of the `_Environ` (os module).
- bpo-44613 [https://bugs.python.org/issue?@action=redirect&bpo=44613]: importlib.metadata is no longer provisional.
- bpo-44558 [https://bugs.python.org/issue?@action=redirect&bpo=44558]: Match the docstring and python implementation of **`countOf()`** to the behavior of its c implementation.
- bpo-44544 [https://bugs.python.org/issue?@action=redirect&bpo=44544]: List all kwargs for **`textwrap.wrap()`**, **`textwrap.fill()`**, and **`textwrap.shorten()`**. Now, there are nav links to attributes of **`TextWrap`**, which makes navigation much easier while minimizing duplication in the documentation.
- bpo-38062 [https://bugs.python.org/issue?@action=redirect&bpo=38062]: Clarify that atexit uses equality comparisons internally.
- bpo-40620 [https://bugs.python.org/issue?@action=redirect&bpo=40620]: Convert examples in tutorial controlflow.rst section 4.3 to be interpreter-demo style.
- bpo-43066 [https://bugs.python.org/issue?@action=redirect&bpo=43066]: Added a warning to **`zipfile`** docs: filename arg with a leading slash may cause archive to be un-openable on Windows systems.
- bpo-39452 [https://bugs.python.org/issue?@action=redirect&bpo=39452]: Rewrote `Doc/library/__main__.rst`. Broadened scope of the document to explicitly discuss and differentiate between `__main__.py` in packages versus the `__name__ == '__main__'` expression (and the idioms that surround it).

- bpo-13814 [https://bugs.python.org/issue?@action=redirect&bpo=13814]: In the Design FAQ, answer “Why don’t generators support the with statement?”
- bpo-27752 [https://bugs.python.org/issue?@action=redirect&bpo=27752]: Documentation of csv.Dialect is more descriptive.
- bpo-44453 [https://bugs.python.org/issue?@action=redirect&bpo=44453]: Fix documentation for the return type of **`sysconfig.get_path()`**.
- bpo-44392 [https://bugs.python.org/issue?@action=redirect&bpo=44392]: Added a new section in the C API documentation for types used in type hinting. Documented `Py_GenericAlias` and `Py_GenericAliasType`.
- bpo-38291 [https://bugs.python.org/issue?@action=redirect&bpo=38291]: Mark `typing.io` and `typing.re` as deprecated since Python 3.8 in the documentation. They were never properly supported by type checkers.
- bpo-44322 [https://bugs.python.org/issue?@action=redirect&bpo=44322]: Document that SyntaxError args have a details tuple and that details are adjusted for errors in f-string field replacement expressions.
- bpo-42392 [https://bugs.python.org/issue?@action=redirect&bpo=42392]: Document the deprecation and removal of the `loop` parameter for many functions and classes in **`asyncio`**.
- bpo-44195 [https://bugs.python.org/issue?@action=redirect&bpo=44195]: Corrected references to `TraversableResources` in docs. There is no `TraversableReader`.
- bpo-41963 [https://bugs.python.org/issue?@action=redirect&bpo=41963]: Document that `ConfigParser` strips off comments when reading configuration files.
- bpo-44072 [https://bugs.python.org/issue?@action=redirect&bpo=44072]: Correct where in the numeric ABC hierarchy `**` support is added, i.e., in numbers.Complex, not numbers.Integral.

- bpo-43558 [https://bugs.python.org/issue?@action=redirect&bpo=43558]: Add the remark to **`dataclasses`** documentation that the **`__init__()`** of any base class has to be called in **`__post_init__()`**, along with a code example.
- bpo-44025 [https://bugs.python.org/issue?@action=redirect&bpo=44025]: Clarify when '_' in match statements is a keyword, and when not.
- bpo-41706 [https://bugs.python.org/issue?@action=redirect&bpo=41706]: Fix docs about how methods like `__add__` are invoked when evaluating operator expressions.
- bpo-41621 [https://bugs.python.org/issue?@action=redirect&bpo=41621]: Document that **`collections.defaultdict`** parameter `default_factory` defaults to `None` and is positional-only.
- bpo-41576 [https://bugs.python.org/issue?@action=redirect&bpo=41576]: document BaseException in favor of bare except
- bpo-21760 [https://bugs.python.org/issue?@action=redirect&bpo=21760]: The description for __file__ fixed. Patch by Furkan Onder
- bpo-39498 [https://bugs.python.org/issue?@action=redirect&bpo=39498]: Add a "Security Considerations" index which links to standard library modules that have explicitly documented security considerations.
- bpo-33479 [https://bugs.python.org/issue?@action=redirect&bpo=33479]: Remove the unqualified claim that tkinter is threadsafe. It has not been true for several years and likely never was. An explanation of what is true may be added later, after more discussion, and possibly after patching _tkinter.c,

Tests

- bpo-40173 [https://bugs.python.org/issue?@action=redirect&bpo=40173]: Fix `test.support.import_helper.import_fresh_module()`.
- bpo-45280 [https://bugs.python.org/issue?@action=redirect&bpo=45280]: Add a test case for empty `typing.NamedTuple`.
- bpo-45269 [https://bugs.python.org/issue?@action=redirect&bpo=45269]: Cover case when invalid `markers` type is supplied to `c_make_encoder`.
- bpo-45128 [https://bugs.python.org/issue?@action=redirect&bpo=45128]: Fix `test_multiprocessing_fork` failure due to `test_logging` and `sys.modules` manipulation.
- bpo-45209 [https://bugs.python.org/issue?@action=redirect&bpo=45209]: Fix `UserWarning: resource_tracker` warning in `_test_multiprocessing._TestSharedMemory.test_shared_memory_cleaned_after_process_termination`
- bpo-45185 [https://bugs.python.org/issue?@action=redirect&bpo=45185]: Enables `TestEnumerations` test cases in `test_ssl` suite.
- bpo-45195 [https://bugs.python.org/issue?@action=redirect&bpo=45195]: Fix test_readline.test_nonascii(): sometimes, the newline character is not written at the end, so don’t expect it in the output. Patch by Victor Stinner.
- bpo-45156 [https://bugs.python.org/issue?@action=redirect&bpo=45156]: Fixes infinite loop on `unittest.mock.seal()` of mocks created by **`create_autospec()`**.

- bpo-45125 [https://bugs.python.org/issue?@action=redirect&bpo=45125]: Improves pickling tests and docs of `SharedMemory` and `SharableList` objects.

- bpo-44860 [https://bugs.python.org/issue?@action=redirect&bpo=44860]: Update `test_sysconfig.test_user_similar()` for the posix_user scheme: `platlib` doesn't use **`sys.platlibdir`**. Patch by Victor Stinner.

- bpo-45052 [https://bugs.python.org/issue?@action=redirect&bpo=45052]: `WithProcessesTestSharedMemory.test_shared_memory_basics` test was ignored, because `self.assertEqual(sms.size, sms2.size)` line was failing. It is now removed and test is unskipped.

  The main motivation for this line to be removed from the test is that the `size` of `SharedMemory` is not ever guaranteed to be the same. It is decided by the platform.

- bpo-44895 [https://bugs.python.org/issue?@action=redirect&bpo=44895]: libregrtest now clears the type cache later to reduce the risk of false alarm when checking for reference leaks. Previously, the type cache was cleared too early and libregrtest raised a false alarm about reference leaks under very specific conditions. Patch by Irit Katriel and Victor Stinner.

- bpo-45042 [https://bugs.python.org/issue?@action=redirect&bpo=45042]: Fixes that test classes decorated with `@hashlib_helper.requires_hashdigest` were skipped all the time.

- bpo-25130 [https://bugs.python.org/issue?@action=redirect&bpo=25130]: Add calls of **`gc.collect()`** in tests to support PyPy.

- bpo-45011 [https://bugs.python.org/issue?@action=redirect&bpo=45011]: Made tests relying on the **`_asyncio`** C extension module optional to allow running on alternative Python implementations. Patch by Serhiy Storchaka.
- bpo-44949 [https://bugs.python.org/issue?@action=redirect&bpo=44949]: Fix auto history tests of test_readline: sometimes, the newline character is not written at the end, so don’t expect it in the output.
- bpo-44891 [https://bugs.python.org/issue?@action=redirect&bpo=44891]: Tests were added to clarify **`id()`** is preserved when `obj * 1` is used on **`str`** and **`bytes`** objects. Patch by Nikita Sobolev.
- bpo-44852 [https://bugs.python.org/issue?@action=redirect&bpo=44852]: Add ability to wholesale silence DeprecationWarnings while running the regression test suite.
- bpo-40928 [https://bugs.python.org/issue?@action=redirect&bpo=40928]: Notify users running test_decimal regression tests on macOS of potential harmless “malloc can’t allocate region” messages spewed by test_decimal.
- bpo-44734 [https://bugs.python.org/issue?@action=redirect&bpo=44734]: Fixed floating-point precision issue in turtle tests.
- bpo-44708 [https://bugs.python.org/issue?@action=redirect&bpo=44708]: Regression tests, when run with -w, are now re-running only the affected test methods instead of re-running the entire test file.

- bpo-42095 [https://bugs.python.org/issue?@action=redirect&bpo=42095]: Added interop tests for Apple plists: generate plist files with Python plistlib and parse with Apple plutil; and the other way round.

- bpo-44647 [https://bugs.python.org/issue?@action=redirect&bpo=44647]: Added a permanent Unicode-valued environment variable to regression tests to ensure they handle this use case in the future. If your test environment breaks because of that, report a bug to us, and temporarily set PYTHONREGRTEST_UNICODE_GUARD=0 in your test environment.

- bpo-44515 [https://bugs.python.org/issue?@action=redirect&bpo=44515]: Adjust recently added contextlib tests to avoid assuming the use of a refcounted GC

- bpo-44287 [https://bugs.python.org/issue?@action=redirect&bpo=44287]: Fix asyncio test_popen() of test_windows_utils by using a longer timeout. Use military grade battle-tested **`test.support.SHORT_TIMEOUT`** timeout rather than a hardcoded timeout of 10 seconds: it’s 30 seconds by default, but it is made longer on slow buildbots. Patch by Victor Stinner.

- bpo-44451 [https://bugs.python.org/issue?@action=redirect&bpo=44451]: Reset `DeprecationWarning` filters in `test.test_importlib.test_metadata_api.APITests.test_entry_points_by_index` to avoid `StopIteration` error if `DeprecationWarnings` are ignored.

- bpo-44363 [https://bugs.python.org/issue?@action=redirect&bpo=44363]: Account for address sanitizer in test_capi. test_capi now passes when run GCC address sanitizer.

- bpo-44364 [https://bugs.python.org/issue?@action=redirect&bpo=44364]: Add non integral tests for `math.sqrt()` function.
- bpo-43921 [https://bugs.python.org/issue?@action=redirect&bpo=43921]: Fix test_ssl.test_wrong_cert_tls13(): use `suppress_ragged_eofs=False`, since `read()` can raise `ssl.SSLEOFError` on Windows. Patch by Victor Stinner.
- bpo-43921 [https://bugs.python.org/issue?@action=redirect&bpo=43921]: Fix test_pha_required_nocert() of test_ssl: catch two more EOF cases (when the `recv()` method returns an empty string). Patch by Victor Stinner.
- bpo-44131 [https://bugs.python.org/issue?@action=redirect&bpo=44131]: Add test_frozenmain to test_embed to test the **`Py_FrozenMain()`** C function. Patch by Victor Stinner.
- bpo-31904 [https://bugs.python.org/issue?@action=redirect&bpo=31904]: Ignore error string case in test_file_not_exists().
- bpo-42083 [https://bugs.python.org/issue?@action=redirect&bpo=42083]: Add test to check that `PyStructSequence_NewType` accepts a `PyStructSequence_Desc` with `doc` field set to `NULL`.
- bpo-35753 [https://bugs.python.org/issue?@action=redirect&bpo=35753]: Fix crash in doctest when doctest parses modules that include unwrappable functions by skipping those functions.

- bpo-30256 [https://bugs.python.org/issue?@action=redirect&bpo=30256]: Add test for nested queues when using `multiprocessing` shared objects `AutoProxy[Queue]` inside `ListProxy` and `DictProxy`

Build

- bpo-45220 [https://bugs.python.org/issue?@action=redirect&bpo=45220]: Avoid building with the Windows 11 SDK previews automatically. This may be overridden by setting the `DefaultWindowsSDKVersion` environment variable before building.

- bpo-45020 [https://bugs.python.org/issue?@action=redirect&bpo=45020]: Freeze stdlib modules that are imported during startup. This provides significant performance improvements to startup. If necessary, use the previously added "-X frozen_modules=off" commandline option to force importing the source modules.

- bpo-45188 [https://bugs.python.org/issue?@action=redirect&bpo=45188]: Windows builds now regenerate frozen modules as the first part of the build. Previously the regeneration was later in the build, which would require it to be restarted if any modules had changed.

- bpo-45163 [https://bugs.python.org/issue?@action=redirect&bpo=45163]: Fixes Haiku platform build.

- bpo-45067 [https://bugs.python.org/issue?@action=redirect&bpo=45067]: The ncurses function extended_color_content was introduced in 2017 (https://invisible-island.net/ncurses/NEWS.html#index-t20170401). The

ncurses-devel package in CentOS 7 had a older version ncurses resulted in compilation error. For compiling ncurses with extended color support, we verify the version of the ncurses library >= 20170401.

- bpo-45019 [https://bugs.python.org/issue?@action=redirect&bpo=45019]: Generate lines in relevant files for frozen modules. Up until now each of the files had to be edited manually. This change makes it easier to add to and modify the frozen modules.
- bpo-44340 [https://bugs.python.org/issue?@action=redirect&bpo=44340]: Add support for building with clang thin lto via –with-lto=thin/full. Patch by Donghee Na and Brett Holman.
- bpo-44535 [https://bugs.python.org/issue?@action=redirect&bpo=44535]: Enable building using a Visual Studio 2022 install on Windows.
- bpo-43298 [https://bugs.python.org/issue?@action=redirect&bpo=43298]: Improved error message when building without a Windows SDK installed.
- bpo-44381 [https://bugs.python.org/issue?@action=redirect&bpo=44381]: The Windows build now accepts **`EnableControlFlowGuard`** set to `guard` to enable CFG.
- bpo-41282 [https://bugs.python.org/issue?@action=redirect&bpo=41282]: Fix broken `make install` that caused standard library extension modules to be unnecessarily and incorrectly rebuilt during the install phase of cpython.

Windows

- bpo-45375 [https://bugs.python.org/issue?@action=redirect&bpo=45375]: Fixes an assertion failure due to searching for the standard library in unnormalised paths.
- bpo-45022 [https://bugs.python.org/issue?@action=redirect&bpo=45022]: Update Windows release to include libffi 3.4.2
- bpo-45007 [https://bugs.python.org/issue?@action=redirect&bpo=45007]: Update to OpenSSL 1.1.1l in Windows build
- bpo-44848 [https://bugs.python.org/issue?@action=redirect&bpo=44848]: Upgrade Windows installer to use SQLite 3.36.0.
- bpo-44572 [https://bugs.python.org/issue?@action=redirect&bpo=44572]: Avoid consuming standard input in the `platform` module
- bpo-44582 [https://bugs.python.org/issue?@action=redirect&bpo=44582]: Accelerate speed of `mimetypes` initialization using a native implementation of the registry scan.
- bpo-41299 [https://bugs.python.org/issue?@action=redirect&bpo=41299]: Fix 16 milliseconds jitter when using timeouts in `threading`, such as with `threading.Lock.acquire()` or `threading.Condition.wait()`.
- bpo-42686 [https://bugs.python.org/issue?@action=redirect&bpo=42686]: Build `sqlite3` with math functions enabled. Patch by Erlend E. Aasland.
- bpo-40263 [https://bugs.python.org/issue?@action=redirect&bpo=40263]: This is a follow-on bug from https://bugs.python.org/issue26903. Once that is applied we run into an off-by-one assertion problem. The assert was not correct.

macOS

- bpo-45007 [https://bugs.python.org/issue?@action=redirect&bpo=45007]: Update macOS installer builds to use OpenSSL 1.1.1l.
- bpo-34602 [https://bugs.python.org/issue?@action=redirect&bpo=34602]: When building CPython on macOS with `./configure --with-undefined-behavior-sanitizer --with-pydebug`, the stack size is now quadrupled to allow for the entire test suite to pass.
- bpo-44848 [https://bugs.python.org/issue?@action=redirect&bpo=44848]: Update macOS installer to use SQLite 3.36.0.
- bpo-44689 [https://bugs.python.org/issue?@action=redirect&bpo=44689]: **`ctypes.util.find_library()`** now works correctly on macOS 11 Big Sur even if Python is built on an older version of macOS. Previously, when built on older macOS systems, `find_library` was not able to find macOS system libraries when running on Big Sur due to changes in how system libraries are stored.
- bpo-41972 [https://bugs.python.org/issue?@action=redirect&bpo=41972]: The framework build’s user header path in sysconfig is changed to add a ‘pythonX.Y’ component to match distutils’s behavior.
- bpo-43109 [https://bugs.python.org/issue?@action=redirect&bpo=43109]: Allow –with-lto configure option to work with Apple-supplied Xcode or Command Line Tools.
- bpo-34932 [https://bugs.python.org/issue?@action=redirect&bpo=34932]: Add socket.TCP_KEEPALIVE support for macOS. Patch by Shane Harvey.

IDLE

- bpo-45296 [https://bugs.python.org/issue?@action=redirect&bpo=45296]: On Windows, change exit/quit message to suggest Ctrl-D, which works, instead of <Ctrl-Z Return>, which does not work in IDLE.
- bpo-45193 [https://bugs.python.org/issue?@action=redirect&bpo=45193]: Make completion boxes appear on Ubuntu again.
- bpo-40128 [https://bugs.python.org/issue?@action=redirect&bpo=40128]: Mostly fix completions on macOS when not using tcl/tk 8.6.11 (as with 3.9). The added update_idletask call should be harmless and possibly helpful otherwise.
- bpo-33962 [https://bugs.python.org/issue?@action=redirect&bpo=33962]: Move the indent space setting from the Font tab to the new Windows tab. Patch by Mark Roseman and Terry Jan Reedy.
- bpo-40468 [https://bugs.python.org/issue?@action=redirect&bpo=40468]: Split the settings dialog General tab into Windows and Shell/ED tabs. Move help sources, which extend the Help menu, to the Extensions tab. Make space for new options and shorten the dialog. The latter makes the dialog better fit small screens.
- bpo-41611 [https://bugs.python.org/issue?@action=redirect&bpo=41611]: Avoid uncaught exceptions in `AutoCompleteWindow.winconfig_event()`.
- bpo-41611 [https://bugs.python.org/issue?@action=redirect&bpo=41611]: Fix IDLE sometimes freezing upon tab-completion on macOS.
- bpo-44010 [https://bugs.python.org/issue?@action=redirect&bpo=44010]: Highlight the new match statement’s soft keywords: **`match`**, **`case`**, and **`_`**. However, this highlighting is not perfect and will be incorrect in some rare cases, including some `_`-s in `case` patterns.
- bpo-44026 [https://bugs.python.org/issue?@action=redirect&bpo=44026]: Include interpreter’s typo fix

suggestions in message line for NameErrors and AttributeErrors. Patch by E. Paine.

Tools/Demos

- bpo-44786 [https://bugs.python.org/issue?@action=redirect&bpo=44786]: Fix a warning in regular expression in the c-analyzer script.
- bpo-44967 [https://bugs.python.org/issue?@action=redirect&bpo=44967]: pydoc now returns a non-zero status code when a module cannot be found.
- bpo-44978 [https://bugs.python.org/issue?@action=redirect&bpo=44978]: Allow the Argument Clinic tool to handle `__complex__` special methods.
- bpo-43425 [https://bugs.python.org/issue?@action=redirect&bpo=43425]: Removed the ‘test2to3’ demo project that demonstrated using lib2to3 to support Python 2.x and Python 3.x from a single source in a distutils package. Patch by Donghee Na
- bpo-44074 [https://bugs.python.org/issue?@action=redirect&bpo=44074]: Make patchcheck automatically detect the correct base branch name (previously it was hardcoded to ‘master’)
- bpo-20291 [https://bugs.python.org/issue?@action=redirect&bpo=20291]: Added support for variadic positional parameters in Argument Clinic.

C API

- bpo-41710 [https://bugs.python.org/issue?@action=redirect&bpo=41710]: The PyThread_acquire_lock_timed() function now clamps the timeout if it is too large, rather than aborting the process. Patch by Victor Stinner.

- bpo-44687 [https://bugs.python.org/issue?@action=redirect&bpo=44687]: **`BufferedReader.peek()`** no longer raises `ValueError` when the entire file has already been buffered.

- bpo-45116 [https://bugs.python.org/issue?@action=redirect&bpo=45116]: Add the `Py_ALWAYS_INLINE` macro to ask the compiler to always inline a static inline function. The compiler can ignore it and decides to not inline the function. Patch by Victor Stinner.

- bpo-45094 [https://bugs.python.org/issue?@action=redirect&bpo=45094]: Add the `Py_NO_INLINE` macro to disable inlining on a function. Patch by Victor Stinner.

- bpo-45061 [https://bugs.python.org/issue?@action=redirect&bpo=45061]: Add a deallocator to the `bool` type to detect refcount bugs in C extensions which call `Py_DECREF(Py_True);` or `Py_DECREF(Py_False);` by mistake. Patch by Victor Stinner.

- bpo-42035 [https://bugs.python.org/issue?@action=redirect&bpo=42035]: Add a new `PyType_GetQualName()` function to get type’s qualified name.

- bpo-41103 [https://bugs.python.org/issue?@action=redirect&bpo=41103]: Reverts removal of the old buffer protocol because they are part of stable ABI.

- bpo-44751 [https://bugs.python.org/issue?@action=redirect&bpo=44751]: Remove `crypt.h` include from the public `Python.h` header.

- bpo-42747 [https://bugs.python.org/issue?@action=redirect&bpo=42747]: The `Py_TPFLAGS_HAVE_VERSION_TAG` type flag now does nothing. The `Py_TPFLAGS_HAVE_AM_SEND` flag (which was added in 3.10) is removed. Both were unnecessary because it is not possible to have type objects with the relevant fields missing.

- bpo-44530 [https://bugs.python.org/issue?@action=redirect&bpo=44530]: Added the `co_qualname` to the `PyCodeObject` structure to propagate the qualified name from the compiler to code objects.

  Patch by Gabriele N. Tornetta

- bpo-44441 [https://bugs.python.org/issue?@action=redirect&bpo=44441]: **`Py_RunMain()`** now resets **`PyImport_Inittab`** to its initial value at exit. It must be possible to call **`PyImport_AppendInittab()`** or **`PyImport_ExtendInittab()`** at each Python initialization. Patch by Victor Stinner.

- bpo-39947 [https://bugs.python.org/issue?@action=redirect&bpo=39947]: Remove 4 private trashcan C API functions which were only kept for the backward compatibility of the stable ABI with Python 3.8 and older, since the trashcan API was not usable with the limited C API on Python 3.8 and older. The trashcan API was excluded from the limited C API in Python 3.9.

  Removed functions:

  - _PyTrash_deposit_object()
  - _PyTrash_destroy_chain()
  - _PyTrash_thread_deposit_object()
  - _PyTrash_thread_destroy_chain()

  The trashcan C API was never usable with the limited C API, since old trashcan macros accessed directly **`PyThreadState`** members like `_tstate->trash_delete_nesting`, whereas the **`PyThreadState`** structure is opaque in the limited C API.

Exclude also the `PyTrash_UNWIND_LEVEL` constant from the C API.

Patch by Victor Stinner.

- bpo-40939 [https://bugs.python.org/issue?@action=redirect&bpo=40939]: Removed documentation for the removed `PyParser_*` C API.
- bpo-43795 [https://bugs.python.org/issue?@action=redirect&bpo=43795]: The list in Contents of Limited API now shows the public name **`PyFrameObject`** rather than `_frame`. The non-existing entry `_node` no longer appears in the list.
- bpo-44378 [https://bugs.python.org/issue?@action=redirect&bpo=44378]: **`Py_IS_TYPE()`** no longer uses **`Py_TYPE()`** to avoid a compiler warning: no longer cast `const PyObject*` to `PyObject*`. Patch by Victor Stinner.
- bpo-39573 [https://bugs.python.org/issue?@action=redirect&bpo=39573]: Convert the **`Py_TYPE()`** and **`Py_SIZE()`** macros to static inline functions. The **`Py_SET_TYPE()`** and **`Py_SET_SIZE()`** functions must now be used to set an object type and size. Patch by Victor Stinner.
- bpo-44263 [https://bugs.python.org/issue?@action=redirect&bpo=44263]: The **`PyType_Ready()`** function now raises an error if a type is defined with the **`Py_TPFLAGS_HAVE_GC`** flag set but has no traverse function (**`PyTypeObject.tp_traverse`**). Patch by Victor Stinner.
- bpo-43795 [https://bugs.python.org/issue?@action=redirect&bpo=43795]: The undocumented function **`Py_FrozenMain()`** is removed from the Limited API.

- bpo-44113 [https://bugs.python.org/issue?@action=redirect&bpo=44113]: Deprecate the following functions to configure the Python initialization:
  - **`PySys_AddWarnOptionUnicode()`**
  - **`PySys_AddWarnOption()`**
  - **`PySys_AddXOption()`**
  - **`PySys_HasWarnOptions()`**
  - **`Py_SetPath()`**
  - **`Py_SetProgramName()`**
  - **`Py_SetPythonHome()`**
  - **`Py_SetStandardStreamEncoding()`**
  - **`_Py_SetProgramFullPath()`**

  Use the new **`PyConfig`** API of the Python Initialization Configuration instead (**PEP 587** [https://peps.python.org/pep-0587/]).

- bpo-44094 [https://bugs.python.org/issue?@action=redirect&bpo=44094]: Remove `PyErr_SetFromErrnoWithUnicodeFilename()`, `PyErr_SetFromWindowsErrWithUnicodeFilename()`, and `PyErr_SetExcFromWindowsErrWithUnicodeFilename()`. They are not documented and have been deprecated since Python 3.3.

- bpo-43795 [https://bugs.python.org/issue?@action=redirect&bpo=43795]: **`PyCodec_Unregister()`** is now properly exported as a function in the Windows Stable ABI DLL.

- bpo-44029 [https://bugs.python.org/issue?@action=redirect&bpo=44029]: Remove deprecated `Py_UNICODE` APIs: `PyUnicode_Encode`, `PyUnicode_EncodeUTF7`, `PyUnicode_EncodeUTF8`, `PyUnicode_EncodeUTF16`, `PyUnicode_EncodeUTF32`, `PyUnicode_EncodeLatin1`, `PyUnicode_EncodeMBCS`, `PyUnicode_EncodeDecimal`, `PyUnicode_EncodeRawUnicodeEscape`, `PyUnicode_EncodeCharmap`, `PyUnicode_EncodeUnicodeEscape`, `PyUnicode_TransformDecimalToASCII`, `PyUnicode_TranslateCharmap`, `PyUnicodeEncodeError_Create`, `PyUnicodeTranslateError_Create`. See **PEP 393** [https://peps.python.org/pep-0393/] and **PEP 624** [https://peps.python.org/pep-0624/] for reference.

- bpo-42035 [https://bugs.python.org/issue?@action=redirect&bpo=42035]: Add a new **`PyType_GetName()`** function to get type's short name.

Python 3.10.0 beta 1

Release date: 2021-05-03

Security

- bpo-43434 [https://bugs.python.org/issue?@action=redirect&bpo=43434]: Creating **`sqlite3.Connection`** objects now also produces `sqlite3.connect` and `sqlite3.connect/handle` auditing events. Previously these events were only produced by **`sqlite3.connect()`** calls. Patch by Erlend E. Aasland.

- bpo-43998 [https://bugs.python.org/issue?@action=redirect&bpo=43998]: The **`ssl`** module sets more secure cipher suites defaults. Ciphers without forward secrecy and with SHA-1 MAC are disabled by default. Security level 2 prohibits weak RSA, DH, and ECC keys with less than 112 bits of security. **`SSLContext`** defaults to minimum protocol version TLS 1.2. Settings are based on Hynek Schlawack's research.

- bpo-43882 [https://bugs.python.org/issue?@action=redirect&bpo=43882]: The presence of newline or tab characters in parts of a URL could allow some forms of attacks.

  Following the controlling specification for URLs defined by WHATWG `urllib.parse()` now removes ASCII newlines and tabs from URLs, preventing such attacks.

- bpo-43472 [https://bugs.python.org/issue?@action=redirect&bpo=43472]: Ensures interpreter-level audit hooks receive the `cpython.PyInterpreterState_New` event when called through the `_xxsubinterpreters` module.

- bpo-43362 [https://bugs.python.org/issue?@action=redirect&bpo=43362]: Fix invalid free in _sha3 module. The issue was introduced in 3.10.0a1. Python 3.9 and earlier are not affected.

- bpo-43762 [https://bugs.python.org/issue?@action=redirect&bpo=43762]: Add audit events for **`sqlite3.connect/handle()`**, `sqlite3.Connection.enable_load_extension()`, and `sqlite3.Connection.load_extension()`. Patch by Erlend E. Aasland.

- bpo-43756 [https://bugs.python.org/issue?@action=redirect&bpo=43756]: Add new audit event `glob.glob/2` to incorporate the new *root_dir* and *dir_fd* arguments added to `glob.glob()` and `glob.iglob()`.

- bpo-36384 [https://bugs.python.org/issue?@action=redirect&bpo=36384]: `ipaddress` module no longer accepts any leading zeros in IPv4 address strings. Leading zeros are ambiguous and interpreted as octal notation by some libraries. For example the legacy function `socket.inet_aton()` treats leading zeros as octal notation. glibc implementation of modern `inet_pton()` does not accept any leading zeros. For a while the `ipaddress` module used to accept ambiguous leading zeros.

- bpo-43075 [https://bugs.python.org/issue?@action=redirect&bpo=43075]: Fix Regular Expression Denial of Service (ReDoS) vulnerability in `urllib.request.AbstractBasicAuthHandler`. The ReDoS-vulnerable regex has quadratic worst-case complexity and it allows cause a denial of service when identifying crafted invalid RFCs. This ReDoS issue is on the client side and needs remote attackers to control the HTTP server.

- bpo-42800 [https://bugs.python.org/issue?@action=redirect&bpo=42800]: Audit hooks are now fired for frame.f_code, traceback.tb_frame, and generator code/frame attribute access.

- bpo-37363 [https://bugs.python.org/issue?@action=redirect&bpo=37363]: Add audit events to the `http.client` module.

Core and Builtins

- bpo-43977 [https://bugs.python.org/issue?@action=redirect&bpo=43977]: Prevent classes being both a sequence and a mapping when pattern matching.
- bpo-43977 [https://bugs.python.org/issue?@action=redirect&bpo=43977]: Use `tp_flags` on the class object to determine if the subject is a sequence or mapping when pattern matching. Avoids the need to import `collections.abc` when pattern matching.
- bpo-43892 [https://bugs.python.org/issue?@action=redirect&bpo=43892]: Restore proper validation of

complex literal value patterns when parsing **`match`** blocks.
- bpo-43933 [https://bugs.python.org/issue?@action=redirect&bpo=43933]: Set frame.f_lineno to the line number of the 'with' kweyword when executing the call to `__exit__`.
- bpo-43933 [https://bugs.python.org/issue?@action=redirect&bpo=43933]: If the current position in a frame has no line number then set the f_lineno attribute to None, instead of -1, to conform to PEP 626. This should not normally be possible, but might occur in some unusual circumstances.
- bpo-43963 [https://bugs.python.org/issue?@action=redirect&bpo=43963]: Importing the **`_signal`** module in a subinterpreter has no longer side effects.
- bpo-42739 [https://bugs.python.org/issue?@action=redirect&bpo=42739]: The internal representation of line number tables is changed to not use sentinels, and an explicit length parameter is added to the out of process API function `PyLineTable_InitAddressRange`. This makes the handling of line number tables more robust in some circumstances.

Library

- bpo-43908 [https://bugs.python.org/issue?@action=redirect&bpo=43908]: Make **`re`** types immutable. Patch by Erlend E. Aasland.
- bpo-43908 [https://bugs.python.org/issue?@action=redirect&bpo=43908]: Make the **`array.array`** type immutable. Patch by Erlend E. Aasland.

Core and Builtins

- bpo-43901 [https://bugs.python.org/issue?@action=redirect&bpo=43901]: Change class and module objects to lazy-create empty annotations dicts on demand. The annotations dicts are stored in the object’s __dict__ for backwards compatibility.

- bpo-43892 [https://bugs.python.org/issue?@action=redirect&bpo=43892]: Match patterns now use new dedicated AST nodes (`MatchValue`, `MatchSingleton`, `MatchSequence`, `MatchStar`, `MatchMapping`, `MatchClass`) rather than reusing expression AST nodes. `MatchAs` and `MatchOr` are now defined as pattern nodes rather than as expression nodes. Patch by Nick Coghlan.
- bpo-42725 [https://bugs.python.org/issue?@action=redirect&bpo=42725]: Usage of `await`/`yield`/`yield from` and named expressions within an annotation is now forbidden when PEP 563 is activated.
- bpo-43754 [https://bugs.python.org/issue?@action=redirect&bpo=43754]: When performing structural pattern matching (**PEP 634** [https://peps.python.org/pep-0634/]), captured names are now left unbound until the *entire* pattern has matched successfully.
- bpo-42737 [https://bugs.python.org/issue?@action=redirect&bpo=42737]: Annotations for complex targets (everything beside simple names) no longer cause any runtime effects with `from __future__ import annotations`.
- bpo-43914 [https://bugs.python.org/issue?@action=redirect&bpo=43914]: **`SyntaxError`** exceptions raised by the interpreter will highlight the full error range of the expression that constitutes the syntax error itself, instead of just where the problem is detected. Patch by Pablo Galindo.
- bpo-38605 [https://bugs.python.org/issue?@action=redirect&bpo=38605]: Revert making `from __future__ import annotations` the default. This follows the Steering Council decision to postpone PEP 563 changes to at least Python 3.11. See the original email for more information regarding the decision:

https://mail.python.org/archives/list/python-dev@python.org/thread/CLVXXPQ2T2LQ5MP2Y53VVQFCXYWQJHKZ/. Patch by Pablo Galindo.

- bpo-43475 [https://bugs.python.org/issue?@action=redirect&bpo=43475]: Hashes of NaN values now depend on object identity. Formerly, they always hashed to 0 even though NaN values are not equal to one another. Having the same hash for unequal values caused pile-ups in hash tables.
- bpo-43859 [https://bugs.python.org/issue?@action=redirect&bpo=43859]: Improve the error message for **`IndentationError`** exceptions. Patch by Pablo Galindo
- bpo-41323 [https://bugs.python.org/issue?@action=redirect&bpo=41323]: Constant tuple folding in bytecode optimizer now reuses tuple in constant table.
- bpo-43846 [https://bugs.python.org/issue?@action=redirect&bpo=43846]: Data stack usage is much reduced for large literal and call expressions.
- bpo-38530 [https://bugs.python.org/issue?@action=redirect&bpo=38530]: When printing **`NameError`** raised by the interpreter, **`PyErr_Display()`** will offer suggestions of similar variable names in the function that the exception was raised from. Patch by Pablo Galindo
- bpo-43823 [https://bugs.python.org/issue?@action=redirect&bpo=43823]: Improve syntax errors for invalid dictionary literals. Patch by Pablo Galindo.
- bpo-43822 [https://bugs.python.org/issue?@action=redirect&bpo=43822]: Improve syntax errors in the parser for missing commas between expressions. Patch by Pablo Galindo.
- bpo-43798 [https://bugs.python.org/issue?@action=redirect&bpo=43798]: **`ast.alias`** nodes now include source location metadata attributes e.g. lineno, col_offset.
- bpo-43797 [https://bugs.python.org/issue?@action=redirect&bpo=43797]: Improve `SyntaxError` error

messages for invalid comparisons. Patch by Pablo Galindo.

- bpo-43760 [https://bugs.python.org/issue?@action=redirect&bpo=43760]: Move the flag for checking whether tracing is enabled to the C stack, from the heap. Should speed up dispatch in the interpreter.
- bpo-43682 [https://bugs.python.org/issue?@action=redirect&bpo=43682]: Static methods (**`@staticmethod`**) and class methods (**`@classmethod`**) now inherit the method attributes (`__module__`, `__name__`, `__qualname__`, `__doc__`, `__annotations__`) and have a new `__wrapped__` attribute. Patch by Victor Stinner.
- bpo-43751 [https://bugs.python.org/issue?@action=redirect&bpo=43751]: Fixed a bug where `anext(ait, default)` would erroneously return None.
- bpo-42128 [https://bugs.python.org/issue?@action=redirect&bpo=42128]: **`__match_args__`** is no longer allowed to be a list.
- bpo-43683 [https://bugs.python.org/issue?@action=redirect&bpo=43683]: Add GEN_START opcode. Marks start of generator, including async, or coroutine and handles sending values to a newly created generator or coroutine.

Library

- bpo-43105 [https://bugs.python.org/issue?@action=redirect&bpo=43105]: Importlib now resolves relative paths when creating module spec objects from file locations.

Core and Builtins

- bpo-43682 [https://bugs.python.org/issue?@action=redirect&bpo=43682]: Static methods (**`@staticmethod`**) are now callable as regular functions. Patch by Victor Stinner.

- bpo-42609 [https://bugs.python.org/issue?@action=redirect&bpo=42609]: Prevented crashes in the AST validator and optimizer when compiling some absurdly long expressions like `"+0"*1000000`. **`RecursionError`** is now raised instead.
- bpo-38530 [https://bugs.python.org/issue?@action=redirect&bpo=38530]: When printing **`AttributeError`**, **`PyErr_Display()`** will offer suggestions of similar attribute names in the object that the exception was raised from. Patch by Pablo Galindo

Library

- bpo-44015 [https://bugs.python.org/issue?@action=redirect&bpo=44015]: In @dataclass(), raise a TypeError if KW_ONLY is specified more than once.

- bpo-25478 [https://bugs.python.org/issue?@action=redirect&bpo=25478]: Added a *total()* method to collections.Counter() to compute the sum of the counts.

- bpo-43733 [https://bugs.python.org/issue?@action=redirect&bpo=43733]: Change **`netrc.netrc`** to use UTF-8 encoding before using locale encoding.

- bpo-43979 [https://bugs.python.org/issue?@action=redirect&bpo=43979]: Removed an unnecessary list comprehension before looping from **`urllib.parse.parse_qsl()`**. Patch by Christoph Zwerschke and Donghee Na.

- bpo-43993 [https://bugs.python.org/issue?@action=redirect&bpo=43993]: Update bundled pip to 21.1.1.

- bpo-43957 [https://bugs.python.org/issue?@action=redirect&bpo=43957]: [Enum] Deprecate `TypeError` when non-member is used in a containment check; In 3.12 `True` or `False` will be returned instead, and containment will return `True` if the value is either a member of that enum or one of its members’ value.

- bpo-42904 [https://bugs.python.org/issue?@action=redirect&bpo=42904]: For backwards compatibility with previous minor versions of Python, if **`typing.get_type_hints()`** receives no namespace dictionary arguments, **`typing.get_type_hints()`** will search through the global then local namespaces during evaluation of stringized type annotations (string forward references) inside a class.

- bpo-43945 [https://bugs.python.org/issue?@action=redirect&bpo=43945]: [Enum] Deprecate non-standard mixin format() behavior: in 3.12 the enum member, not the member’s value, will be used for format() calls.

- bpo-41139 [https://bugs.python.org/issue?@action=redirect&bpo=41139]: Deprecate undocumented `cgi.log()` API.

- bpo-43937 [https://bugs.python.org/issue?@action=redirect&bpo=43937]: Fixed the **`turtle`** module working with non-default root window.

- bpo-43930 [https://bugs.python.org/issue?@action=redirect&bpo=43930]: Update bundled pip to 21.1 and setuptools to 56.0.0

- bpo-43907 [https://bugs.python.org/issue?@action=redirect&bpo=43907]: Fix a bug in the pure-Python pickle implementation when using protocol 5, where bytearray instances that occur several time in the pickled object graph would incorrectly unpickle into repeated copies of the bytearray object.

- bpo-43926 [https://bugs.python.org/issue?@action=redirect&bpo=43926]: In `importlib.metadata`, provide a uniform interface to `Description`, allow for any field to be encoded with multiline values, remove continuation lines from multiline values, and add a `.json` property for easy access to the PEP 566 JSON-compatible form. Sync with `importlib_metadata 4.0`.

- bpo-43920 [https://bugs.python.org/issue?@action=redirect&bpo=43920]: OpenSSL 3.0.0: **`load_verify_locations()`** now returns a consistent error message when cadata contains no valid certificate.

- bpo-43607 [https://bugs.python.org/issue?@action=redirect&bpo=43607]: **`urllib`** can now convert Windows paths with `\\?\` prefixes into URL paths.

- bpo-43817 [https://bugs.python.org/issue?@action=redirect&bpo=43817]: Add **`inspect.get_annotations()`**, which safely computes the annotations defined on an object. It works around the quirks of accessing the annotations from various types of objects, and makes very few assumptions about the object passed in. **`inspect.get_annotations()`** can also correctly un-stringize stringized annotations.

  `inspect.signature()`, **`inspect.from_callable()`**, and **`inspect.from_function()`** now call **`inspect.get_annotations()`** to retrieve annotations. This means **`inspect.signature()`** and **`inspect.from_callable()`** can now un-stringize stringized annotations, too.

- bpo-43284 [https://bugs.python.org/issue?@action=redirect&bpo=43284]: platform.win32_ver derives the windows version from sys.getwindowsversion().platform_version which in turn derives the version from kernel32.dll (which can be of a different version than Windows itself). Therefore change the platform.win32_ver to determine the version using the platform module's _syscmd_ver private function to return an accurate version.

- bpo-42854 [https://bugs.python.org/issue?@action=redirect&bpo=42854]: The **`ssl`** module now uses `SSL_read_ex` and `SSL_write_ex` internally. The functions support reading and writing of data larger than 2 GB. Writing zero-length data no longer fails with a protocol violation error.

- bpo-42333 [https://bugs.python.org/issue?@action=redirect&bpo=42333]: Port `_ssl` extension module to multiphase initialization.

- bpo-43880 [https://bugs.python.org/issue?@action=redirect&bpo=43880]: **`ssl`** now raises DeprecationWarning for OP_NO_SSL/TLS* options, old TLS versions, old protocols, and other features that have been deprecated since Python 3.6, 3.7, or OpenSSL 1.1.0.

- bpo-41559 [https://bugs.python.org/issue?@action=redirect&bpo=41559]: **PEP 612** [https://peps.python.org/pep-0612/] is now implemented purely in Python; builtin `types.GenericAlias` objects no longer include `typing.ParamSpec` in `__parameters__` (with the exception of `collections.abc.Callable`'s `GenericAlias`). This means previously invalid uses of `ParamSpec` (such as `list[P]`) which worked in earlier versions of Python 3.10 alpha, will now raise `TypeError` during substitution.

- bpo-43867 [https://bugs.python.org/issue?@action=redirect&bpo=43867]: The **`multiprocessing`** `Server` class now explicitly catches **`SystemExit`** and closes the client connection in this case. It happens when the `Server.serve_client()` method reaches the end of file (EOF).

- bpo-40443 [https://bugs.python.org/issue?@action=redirect&bpo=40443]: Remove unused imports: pyclbr no longer uses copy, and typing no longer uses ast. Patch by Victor Stinner.

- bpo-43820 [https://bugs.python.org/issue?@action=redirect&bpo=43820]: Remove an unneeded copy of the namespace passed to dataclasses.make_dataclass().

- bpo-43787 [https://bugs.python.org/issue?@action=redirect&bpo=43787]: Add `__iter__()` method to **`bz2.BZ2File`**, **`gzip.GzipFile`**, and **`lzma.LZMAFile`**. It makes iterating them about 2x faster. Patch by Inada Naoki.

- bpo-43680 [https://bugs.python.org/issue?@action=redirect&bpo=43680]: Deprecate io.OpenWrapper and _pyio.OpenWrapper: use io.open and _pyio.open instead. Until Python 3.9, _pyio.open was not a static method and builtins.open was set to OpenWrapper to not become a bound method when set to a class variable. _io.open is a built-in function whereas _pyio.open is a Python function. In Python 3.10, _pyio.open() is now a static method, and builtins.open() is now io.open().

- bpo-43680 [https://bugs.python.org/issue?@action=redirect&bpo=43680]: The Python **`_pyio.open()`** function becomes a static method to behave as

`io.open()` built-in function: don’t become a bound method when stored as a class variable. It becomes possible since static methods are now callable in Python 3.10. Moreover, **`_pyio.OpenWrapper()`** becomes a simple alias to **`_pyio.open()`**. Patch by Victor Stinner.

- bpo-41515 [https://bugs.python.org/issue?@action=redirect&bpo=41515]: Fix **`KeyError`** raised in **`typing.get_type_hints()`** due to synthetic modules that don’t appear in `sys.modules`.
- bpo-43776 [https://bugs.python.org/issue?@action=redirect&bpo=43776]: When **`subprocess.Popen`** args are provided as a string or as **`pathlib.Path`**, the Popen instance repr now shows the right thing.
- bpo-42248 [https://bugs.python.org/issue?@action=redirect&bpo=42248]: [Enum] ensure exceptions raised in `_missing__` are released
- bpo-43744 [https://bugs.python.org/issue?@action=redirect&bpo=43744]: fix issue with enum member name matching the start of a private variable name
- bpo-43772 [https://bugs.python.org/issue?@action=redirect&bpo=43772]: Fixed the return value of `TypeVar.__ror__`. Patch by Jelle Zijlstra.
- bpo-43764 [https://bugs.python.org/issue?@action=redirect&bpo=43764]: Add match_args parameter to @dataclass decorator to allow suppression of __match_args__ generation.
- bpo-43799 [https://bugs.python.org/issue?@action=redirect&bpo=43799]: OpenSSL 3.0.0: define `OPENSSL_API_COMPAT` 1.1.1 to suppress deprecation warnings. Python requires OpenSSL 1.1.1 APIs.

- bpo-43478 [https://bugs.python.org/issue?@action=redirect&bpo=43478]: Mocks can no longer be used as the specs for other Mocks. As a result, an already-mocked object cannot have an attribute mocked using `autospec=True` or be the subject of a `create_autospec(...)` call. This can uncover bugs in tests since these Mock-derived Mocks will always pass certain tests (e.g. **`isinstance()`**) and builtin assert functions (e.g. assert_called_once_with) will unconditionally pass.

- bpo-43794 [https://bugs.python.org/issue?@action=redirect&bpo=43794]: Add **`ssl.OP_IGNORE_UNEXPECTED_EOF`** constants (OpenSSL 3.0.0)

- bpo-43785 [https://bugs.python.org/issue?@action=redirect&bpo=43785]: Improve `bz2.BZ2File` performance by removing the RLock from BZ2File. This makes BZ2File thread unsafe in the face of multiple simultaneous readers or writers, just like its equivalent classes in **`gzip`** and **`lzma`** have always been. Patch by Inada Naoki.

- bpo-43789 [https://bugs.python.org/issue?@action=redirect&bpo=43789]: OpenSSL 3.0.0: Don’t call the password callback function a second time when first call has signaled an error condition.

- bpo-43788 [https://bugs.python.org/issue?@action=redirect&bpo=43788]: The header files for **`ssl`** error codes are now OpenSSL version-specific. Exceptions will now show correct reason and library codes. The `make_ssl_data.py` script has been rewritten to use OpenSSL’s text file with error codes.

- bpo-43766 [https://bugs.python.org/issue?@action=redirect&bpo=43766]: Implement **PEP 647**

[https://peps.python.org/pep-0647/] in the `typing` module by adding **`TypeGuard`**.

- bpo-25264 [https://bugs.python.org/issue?@action=redirect&bpo=25264]: `os.path.realpath()` now accepts a *strict* keyword-only argument. When set to `True`, `OSError` is raised if a path doesn't exist or a symlink loop is encountered.

- bpo-43780 [https://bugs.python.org/issue?@action=redirect&bpo=43780]: In `importlib.metadata`, incorporate changes from importlib_metadata 3.10: Add mtime-based caching during distribution discovery. Flagged use of dict result from `entry_points()` as deprecated.

- gh-47383 [https://github.com/python/cpython/issues/47383]: The `P.args` and `P.kwargs` attributes of `typing.ParamSpec` are now instances of the new classes `typing.ParamSpecArgs` and `typing.ParamSpecKwargs`, which enables a more useful `repr()`. Patch by Jelle Zijlstra.

- bpo-43731 [https://bugs.python.org/issue?@action=redirect&bpo=43731]: Add an `encoding` parameter **`logging.fileConfig()`**.

- bpo-43712 [https://bugs.python.org/issue?@action=redirect&bpo=43712]: Add `encoding` and `errors` parameters to `fileinput.input()` and `fileinput.FileInput`.

- bpo-38659 [https://bugs.python.org/issue?@action=redirect&bpo=38659]: A `simple_enum` decorator is added to the `enum` module to convert a normal class into an Enum. `test_simple_enum` added to test simple enums against a corresponding normal Enum. Standard library modules updated to use `simple_enum`.

- bpo-43764 [https://bugs.python.org/issue?@action=redirect&bpo=43764]: Fix an issue where `__match_args__` generation could fail for some `dataclasses`.

- bpo-43752 [https://bugs.python.org/issue?@action=redirect&bpo=43752]: Fix `sqlite3` regression for zero-sized blobs with converters, where `b""` was returned instead of `None`. The regression was introduced by PR 24723. Patch by Erlend E. Aasland.

- bpo-43655 [https://bugs.python.org/issue?@action=redirect&bpo=43655]: `tkinter` dialog windows are now recognized as dialogs by window managers on macOS and X Window.

- bpo-43723 [https://bugs.python.org/issue?@action=redirect&bpo=43723]: The following `threading` methods are now deprecated and should be replaced:

  - `currentThread` => `threading.current_thread()`
  - `activeCount` => `threading.active_count()`
  - `Condition.notifyAll` => `threading.Condition.notify_all()`
  - `Event.isSet` => `threading.Event.is_set()`
  - `Thread.setName` => `threading.Thread.name`
  - `thread.getName` => `threading.Thread.name`
  - `Thread.isDaemon` => `threading.Thread.daemon`
  - `Thread.setDaemon` => `threading.Thread.daemon`

  Patch by Jelle Zijlstra.

- bpo-2135 [https://bugs.python.org/issue?@action=redirect&bpo=2135]: Deprecate find_module() and find_loader() implementations in importlib and zipimport.

- bpo-43534 [https://bugs.python.org/issue?@action=redirect&bpo=43534]: `turtle.textinput()` and

`turtle.numinput()` create now a transient window working on behalf of the canvas window.

- bpo-43532 [https://bugs.python.org/issue?@action=redirect&bpo=43532]: Add the ability to specify keyword-only fields to dataclasses. These fields will become keyword-only arguments to the generated __init__.

- bpo-43522 [https://bugs.python.org/issue?@action=redirect&bpo=43522]: Fix problem with **`hostname_checks_common_name`**. OpenSSL does not copy hostflags from *struct SSL_CTX* to *struct SSL*.

- bpo-8978 [https://bugs.python.org/issue?@action=redirect&bpo=8978]: Improve error message for **`tarfile.open()`** when **`lzma`** / **`bz2`** are unavailable. Patch by Anthony Sottile.

- bpo-42967 [https://bugs.python.org/issue?@action=redirect&bpo=42967]: Allow **`bytes`** `separator` argument in `urllib.parse.parse_qs` and `urllib.parse.parse_qsl` when parsing **`str`** query strings. Previously, this raised a `TypeError`.

- bpo-43296 [https://bugs.python.org/issue?@action=redirect&bpo=43296]: Improve **`sqlite3`** error handling: `sqlite3_value_blob()` errors that set `SQLITE_NOMEM` now raise **`MemoryError`**. Patch by Erlend E. Aasland.

- bpo-43312 [https://bugs.python.org/issue?@action=redirect&bpo=43312]: New functions **`sysconfig.get_preferred_scheme()`** and **`sysconfig.get_default_scheme()`** are added to query a platform for its preferred “user”, “home”, and “prefix” (default) scheme names.

- bpo-43265 [https://bugs.python.org/issue?@action=redirect&bpo=43265]: Improve `sqlite3.Connection.backup()` error handling. The error message for non-existent target database names is now `unknown database <database name>` instead of `SQL logic error`. Patch by Erlend E. Aasland.

- bpo-41282 [https://bugs.python.org/issue?@action=redirect&bpo=41282]: Install schemes in `distutils.command.install` are now loaded from `sysconfig`.

- bpo-41282 [https://bugs.python.org/issue?@action=redirect&bpo=41282]: `distutils.sysconfig` has been merged to `sysconfig`.

- bpo-43176 [https://bugs.python.org/issue?@action=redirect&bpo=43176]: Fixed processing of a dataclass that inherits from a frozen dataclass with no fields. It is now correctly detected as an error.

- bpo-43080 [https://bugs.python.org/issue?@action=redirect&bpo=43080]: `pprint` now has support for `dataclasses.dataclass`. Patch by Lewis Gaul.

- bpo-39950 [https://bugs.python.org/issue?@action=redirect&bpo=39950]: Add `pathlib.Path.hardlink_to()` method that supersedes `link_to()`. The new method has the same argument order as `symlink_to()`.

- bpo-42904 [https://bugs.python.org/issue?@action=redirect&bpo=42904]: `typing.get_type_hints()` now checks the local namespace of a class when evaluating **PEP 563** [https://peps.python.org/pep-0563/] annotations inside said class.

- bpo-42269 [https://bugs.python.org/issue?@action=redirect&bpo=42269]: Add `slots` parameter to `dataclasses.dataclass` decorator to automatically generate `__slots__` for class. Patch provided by Yurii Karabas.

- bpo-39529 [https://bugs.python.org/issue?@action=redirect&bpo=39529]: Deprecated use of **`asyncio.get_event_loop()`** without running event loop. Emit deprecation warning for **`asyncio`** functions which implicitly create a **`Future`** or **`Task`** objects if there is no running event loop and no explicit *loop* argument is passed: **`ensure_future()`**, **`wrap_future()`**, **`gather()`**, **`shield()`**, **`as_completed()`** and constructors of **`Future`**, **`Task`**, **`StreamReader`**, **`StreamReaderProtocol`**.

- bpo-18369 [https://bugs.python.org/issue?@action=redirect&bpo=18369]: Certificate and PrivateKey classes were added to the ssl module. Certificates and keys can now be loaded from memory buffer, too.

- bpo-41486 [https://bugs.python.org/issue?@action=redirect&bpo=41486]: Use a new output buffer management code for **`bz2`** / **`lzma`** / **`zlib`** modules, and add `.readall()` function to `_compression.DecompressReader` class. These bring some performance improvements. Patch by Ma Lin.

- bpo-31870 [https://bugs.python.org/issue?@action=redirect&bpo=31870]: The **`ssl.get_server_certificate()`** function now has a *timeout* parameter.

- bpo-41735 [https://bugs.python.org/issue?@action=redirect&bpo=41735]: Fix thread locks in zlib module may go wrong in rare case. Patch by Ma Lin.

- bpo-36470 [https://bugs.python.org/issue?@action=redirect&bpo=36470]: Fix dataclasses with `InitVar`s and **`replace()`**. Patch by Claudiu Popa.

- bpo-40849 [https://bugs.python.org/issue?@action=redirect&bpo=40849]: Expose X509_V_FLAG_PARTIAL_CHAIN ssl flag

- bpo-35114 [https://bugs.python.org/issue?@action=redirect&bpo=35114]: **`ssl.RAND_status()`** now returns a boolean value (as documented) instead of `1` or `0`.

- bpo-39906 [https://bugs.python.org/issue?@action=redirect&bpo=39906]: **`pathlib.Path.stat()`** and **`chmod()`** now accept a *follow_symlinks* keyword-only argument for consistency with corresponding functions in the **`os`** module.

- bpo-39899 [https://bugs.python.org/issue?@action=redirect&bpo=39899]: **`os.path.expanduser()`** now refuses to guess Windows home directories if the basename of current user's home directory does not match their username.

  `pathlib.Path.expanduser()` and **`home()`** now consistently raise **`RuntimeError`** exception when a home directory cannot be resolved. Previously a **`KeyError`** exception could be raised on Windows when the `"USERNAME"` environment variable was unset.

- bpo-36076 [https://bugs.python.org/issue?@action=redirect&bpo=36076]: Added SNI support to **`ssl.get_server_certificate()`**.

- bpo-38490 [https://bugs.python.org/issue?@action=redirect&bpo=38490]: Covariance, Pearson's correlation, and simple linear regression functionality was added to statistics module. Patch by Tymoteusz Wołodźko.

- bpo-33731 [https://bugs.python.org/issue?@action=redirect&bpo=33731]: Provide a locale.localize() function, which converts a normalized number string into a locale format.

- bpo-32745 [https://bugs.python.org/issue?@action=redirect&bpo=32745]: Fix a regression in the handling of ctypes' `ctypes.c_wchar_p` type: embedded null characters would cause a `ValueError` to be raised. Patch by Zackery Spytz.

Documentation

- bpo-43987 [https://bugs.python.org/issue?@action=redirect&bpo=43987]: Add "Annotations Best Practices" document as a new HOWTO.
- bpo-43977 [https://bugs.python.org/issue?@action=redirect&bpo=43977]: Document the new `Py_TPFLAGS_MAPPING` and `Py_TPFLAGS_SEQUENCE` type flags.
- bpo-43959 [https://bugs.python.org/issue?@action=redirect&bpo=43959]: The documentation on the PyContextVar C-API was clarified.
- bpo-43938 [https://bugs.python.org/issue?@action=redirect&bpo=43938]: Update dataclasses documentation to express that FrozenInstanceError is derived from AttributeError.
- bpo-43778 [https://bugs.python.org/issue?@action=redirect&bpo=43778]: Fix the Sphinx glossary_search extension: create the _static/ sub-directory if it doesn't exist.
- bpo-43755 [https://bugs.python.org/issue?@action=redirect&bpo=43755]: Update documentation to reflect that unparenthesized lambda expressions can no longer be the expression part in an `if` clause in comprehensions and generator expressions since Python 3.9.

- bpo-43739 [https://bugs.python.org/issue?@action=redirect&bpo=43739]: Fixing the example code in Doc/extending/extending.rst to declare and initialize the pmodule variable to be of the right type.

Tests

- bpo-43961 [https://bugs.python.org/issue?@action=redirect&bpo=43961]: Fix test_logging.test_namer_rotator_inheritance() on Windows: use `os.replace()` rather than `os.rename()`. Patch by Victor Stinner.
- bpo-43842 [https://bugs.python.org/issue?@action=redirect&bpo=43842]: Fix a race condition in the SMTP test of test_logging. Don’t close a file descriptor (socket) from a different thread while asyncore.loop() is polling the file descriptor. Patch by Victor Stinner.
- bpo-43843 [https://bugs.python.org/issue?@action=redirect&bpo=43843]: **`test.libregrtest`** now marks a test as ENV_CHANGED (altered the execution environment) if a thread raises an exception but does not catch it. It sets a hook on `threading.excepthook()`. Use `--fail-env-changed` option to mark the test as failed. Patch by Victor Stinner.
- bpo-43811 [https://bugs.python.org/issue?@action=redirect&bpo=43811]: Tests multiple OpenSSL versions on GitHub Actions. Use ccache to speed up testing.
- bpo-43791 [https://bugs.python.org/issue?@action=redirect&bpo=43791]: OpenSSL 3.0.0: Disable testing of legacy protocols TLS 1.0 and 1.1. Tests are failing with TLSV1_ALERT_INTERNAL_ERROR.

Build

- bpo-43567 [https://bugs.python.org/issue?@action=redirect&bpo=43567]: Improved generated code refresh (AST/tokens/opcodes/keywords) on Windows.
- bpo-43669 [https://bugs.python.org/issue?@action=redirect&bpo=43669]: Implement **PEP 644** [https://peps.python.org/pep-0644/]. Python now requires OpenSSL 1.1.1 or newer.

Windows

- bpo-35306 [https://bugs.python.org/issue?@action=redirect&bpo=35306]: Adds additional arguments to `os.startfile()` function.
- bpo-43538 [https://bugs.python.org/issue?@action=redirect&bpo=43538]: Avoid raising errors from `pathlib.Path.exists()` when passed an invalid filename.
- bpo-38822 [https://bugs.python.org/issue?@action=redirect&bpo=38822]: Fixed `os.stat()` failing on inaccessible directories with a trailing slash, rather than falling back to the parent directory's metadata. This implicitly affected `os.path.exists()` and `os.path.isdir()`.
- bpo-26227 [https://bugs.python.org/issue?@action=redirect&bpo=26227]: Fixed decoding of host names in `socket.gethostbyaddr()` and `socket.gethostbyname_ex()`.
- bpo-40432 [https://bugs.python.org/issue?@action=redirect&bpo=40432]: Updated pegen regeneration script on Windows to find and use Python 3.8 or higher. Prior to this, pegen regeneration already required 3.8 or higher, but the script may have used lower versions of Python.
- bpo-43745 [https://bugs.python.org/issue?@action=redirect&bpo=43745]: Actually updates Windows release to OpenSSL 1.1.1k. Earlier releases were mislabelled and actually included 1.1.1i again.
- bpo-43652 [https://bugs.python.org/issue?@action=redirect&bpo=43652]: Update Tcl and Tk to 8.6.11 in

Windows installer.

- bpo-43492 [https://bugs.python.org/issue?@action=redirect&bpo=43492]: Upgrade Windows installer to use SQLite 3.35.5.
- bpo-30555 [https://bugs.python.org/issue?@action=redirect&bpo=30555]: Fix `WindowsConsoleIO` errors in the presence of fd redirection. Patch by Segev Finer.

macOS

- bpo-42119 [https://bugs.python.org/issue?@action=redirect&bpo=42119]: Fix check for macOS SDK paths when building Python. Narrow search to match contents of SDKs, namely only files in `/System/Library`, `/System/IOSSupport`, and `/usr` other than `/usr/local`. Previously, anything under `/System` was assumed to be in an SDK which causes problems with the new file system layout in 10.15+ where user file systems may appear to be mounted under `/System`. Paths in `/Library` were also incorrectly treated as SDK locations.
- bpo-43568 [https://bugs.python.org/issue?@action=redirect&bpo=43568]: Drop support for MACOSX_DEPLOYMENT_TARGET < 10.3
- bpo-44009 [https://bugs.python.org/issue?@action=redirect&bpo=44009]: Provide “python3.x-intel64” executable to allow reliably forcing macOS universal2 framework builds to run under Rosetta 2 Intel-64 emulation on Apple Silicon Macs. This can be useful for testing or when universal2 wheels are not yet available.
- bpo-43851 [https://bugs.python.org/issue?@action=redirect&bpo=43851]: Build SQLite with `SQLITE_OMIT_AUTOINIT` on macOS. Patch by Erlend E. Aasland.
- bpo-43492 [https://bugs.python.org/issue?@action=redirect&bpo=43492]: Update macOS installer to use SQLite 3.35.4.

- bpo-42235 [https://bugs.python.org/issue?@action=redirect&bpo=42235]: `Mac/BuildScript/build-installer.py` will now use "–enable-optimizations" and `--with-lto` when building on macOS 10.15 or later.

IDLE

- bpo-37903 [https://bugs.python.org/issue?@action=redirect&bpo=37903]: Add mouse actions to the shell sidebar. Left click and optional drag selects one or more lines, as with the editor line number sidebar. Right click after selecting raises a context menu with 'copy with prompts'. This zips together prompts from the sidebar with lines from the selected text.
- bpo-43981 [https://bugs.python.org/issue?@action=redirect&bpo=43981]: Fix reference leak in test_sidebar and test_squeezer. Patches by Terry Jan Reedy and Pablo Galindo
- bpo-37892 [https://bugs.python.org/issue?@action=redirect&bpo=37892]: Indent IDLE Shell input with spaces instead of tabs
- bpo-43655 [https://bugs.python.org/issue?@action=redirect&bpo=43655]: IDLE dialog windows are now recognized as dialogs by window managers on macOS and X Window.
- bpo-37903 [https://bugs.python.org/issue?@action=redirect&bpo=37903]: IDLE's shell now shows prompts in a separate side-bar.

C API

- bpo-43916 [https://bugs.python.org/issue?@action=redirect&bpo=43916]: Add a new `Py_TPFLAGS_DISALLOW_INSTANTIATION` type flag to disallow creating type instances. Patch by Victor Stinner.

- bpo-43774 [https://bugs.python.org/issue?@action=redirect&bpo=43774]: Remove the now unused `PYMALLOC_DEBUG` macro. Debug hooks on memory allocators are now installed by default if Python is built in debug mode (if `Py_DEBUG` macro is defined). Moreover, they can now be used on Python build in release mode (ex: using `PYTHONMALLOC=debug` environment variable).
- bpo-43962 [https://bugs.python.org/issue?@action=redirect&bpo=43962]: _PyInterpreterState_IDIncref() now calls _PyInterpreterState_IDInitref() and always increments id_refcount. Previously, calling _xxsubinterpreters.get_current() could create an id_refcount inconsistency when a _xxsubinterpreters.InterpreterID object was deallocated. Patch by Victor Stinner.
- bpo-28254 [https://bugs.python.org/issue?@action=redirect&bpo=28254]: Add new C-API functions to control the state of the garbage collector: **`PyGC_Enable()`**, **`PyGC_Disable()`**, **`PyGC_IsEnabled()`**, corresponding to the functions in the **`gc`** module.
- bpo-43908 [https://bugs.python.org/issue?@action=redirect&bpo=43908]: Introduce **`Py_TPFLAGS_IMMUTABLETYPE`** flag for immutable type objects, and modify **`PyType_Ready()`** to set it for static types. Patch by Erlend E. Aasland.
- bpo-43795 [https://bugs.python.org/issue?@action=redirect&bpo=43795]: **`PyMem_Calloc()`** is now available in the limited C API (`Py_LIMITED_API`).
- bpo-43868 [https://bugs.python.org/issue?@action=redirect&bpo=43868]: **`PyOS_ReadlineFunctionPointer()`** is no longer exported by limited C API headers and by `python3.dll` on Windows. Like any function that takes `FILE*`, it is not part of the stable ABI.
- bpo-43795 [https://bugs.python.org/issue?@action=redirect&bpo=43795]: Stable ABI and limited API

definitions are generated from a central manifest (**PEP 652** [https://peps.python.org/pep-0652/]).

- bpo-43753 [https://bugs.python.org/issue?@action=redirect&bpo=43753]: Add the `Py_Is(x, y)` function to test if the *x* object is the *y* object, the same as `x is y` in Python. Add also the `Py_IsNone()`, `Py_IsTrue()`, `Py_IsFalse()` functions to test if an object is, respectively, the `None` singleton, the `True` singleton or the `False` singleton. Patch by Victor Stinner.

Python 3.10.0 alpha 7

Release date: 2021-04-05

Security

- bpo-42988 [https://bugs.python.org/issue?@action=redirect&bpo=42988]: **CVE 2021-3426** [https://www.cve.org/CVERecord?id=CVE-2021-3426]: Remove the `getfile` feature of the `pydoc` module which could be abused to read arbitrary files on the disk (directory traversal vulnerability). Moreover, even source code of Python modules can contain sensitive data like passwords. Vulnerability reported by David Schwörer.

- bpo-43285 [https://bugs.python.org/issue?@action=redirect&bpo=43285]: `ftplib` no longer trusts the IP address value returned from the server in response to the PASV command by default. This prevents a malicious FTP server from using the response to probe IPv4 address and port combinations on the client network.

  Code that requires the former vulnerable behavior may set a `trust_server_pasv_ipv4_address` attribute on their `ftplib.FTP` instances to `True` to re-enable it.

- bpo-43439 [https://bugs.python.org/issue?@action=redirect&bpo=43439]: Add audit hooks for `gc.get_objects()`, `gc.get_referrers()` and `gc.get_referents()`. Patch by Pablo Galindo.

Core and Builtins

- bpo-27129 [https://bugs.python.org/issue?@action=redirect&bpo=27129]: Update CPython bytecode magic number.
- bpo-43672 [https://bugs.python.org/issue?@action=redirect&bpo=43672]: Raise ImportWarning when calling find_loader().
- bpo-43660 [https://bugs.python.org/issue?@action=redirect&bpo=43660]: Fix crash that happens when replacing `sys.stderr` with a callable that can remove the object while an exception is being printed. Patch by Pablo Galindo.
- bpo-27129 [https://bugs.python.org/issue?@action=redirect&bpo=27129]: The bytecode interpreter uses instruction, rather byte, offsets internally. This reduces the number of EXTENDED_ARG instructions needed and streamlines instruction dispatch a bit.

Library

- bpo-40645 [https://bugs.python.org/issue?@action=redirect&bpo=40645]: Fix reference leak in the **`_hashopenssl`** extension. Patch by Pablo Galindo.

Core and Builtins

- bpo-42134 [https://bugs.python.org/issue?@action=redirect&bpo=42134]: Calls to find_module() by the import system now raise ImportWarning.

- bpo-41064 [https://bugs.python.org/issue?@action=redirect&bpo=41064]: Improve the syntax error for invalid usage of double starred elements (‘**’) in f-strings. Patch by Pablo Galindo.
- bpo-43575 [https://bugs.python.org/issue?@action=redirect&bpo=43575]: Speed up calls to `map()` by using the **PEP 590** [https://peps.python.org/pep-0590/] `vectorcall` calling convention. Patch by Donghee Na.
- bpo-42137 [https://bugs.python.org/issue?@action=redirect&bpo=42137]: The import system now prefers using `__spec__` for `ModuleType.__repr__` over `module_repr()`.
- bpo-43452 [https://bugs.python.org/issue?@action=redirect&bpo=43452]: Added micro-optimizations to `_PyType_Lookup()` to improve cache lookup performance in the common case of cache hits.
- bpo-43555 [https://bugs.python.org/issue?@action=redirect&bpo=43555]: Report the column offset for **`SyntaxError`** for invalid line continuation characters. Patch by Pablo Galindo.
- bpo-43517 [https://bugs.python.org/issue?@action=redirect&bpo=43517]: Fix misdetection of circular imports when using `from pkg.mod import attr`, which caused false positives in non-trivial multi-threaded code.
- bpo-43497 [https://bugs.python.org/issue?@action=redirect&bpo=43497]: Emit SyntaxWarnings for assertions with tuple constants, this is a regression introduced in python3.7

- bpo-39316 [https://bugs.python.org/issue?@action=redirect&bpo=39316]: Tracing now has correct line numbers for attribute accesses when the attribute is on a different line from the object. Improves debugging and profiling for multi-line method chains.

- bpo-35883 [https://bugs.python.org/issue?@action=redirect&bpo=35883]: Python no longer fails at startup with a fatal error if a command line argument contains an invalid Unicode character. The **`Py_DecodeLocale()`** function now escapes byte sequences which would be decoded as Unicode characters outside the [U+0000; U+10ffff] range.

- bpo-43410 [https://bugs.python.org/issue?@action=redirect&bpo=43410]: Fix a bug that was causing the parser to crash when emitting syntax errors when reading input from stdin. Patch by Pablo Galindo

- bpo-43406 [https://bugs.python.org/issue?@action=redirect&bpo=43406]: Fix a possible race condition where `PyErr_CheckSignals` tries to execute a non-Python signal handler.

- bpo-42128 [https://bugs.python.org/issue?@action=redirect&bpo=42128]: Add `__match_args__` to struct sequence objects. Patch by Pablo Galindo.

- bpo-43390 [https://bugs.python.org/issue?@action=redirect&bpo=43390]: CPython now sets the `SA_ONSTACK` flag in `PyOS_setsig` for the VM’s default signal handlers. This is friendlier to other in-process code that an extension module or embedding use could pull in (such as Golang’s cgo) where tiny thread stacks are the norm and `sigaltstack()` has been used to provide for signal handlers. This is a no-op change for the vast majority of processes that don't use sigaltstack.

- bpo-43287 [https://bugs.python.org/issue?@action=redirect&bpo=43287]: Speed up calls to `filter()` by using the **PEP 590** [https://peps.python.org/pep-0590/] `vectorcall` calling convention. Patch by Donghee Na.

- bpo-37448 [https://bugs.python.org/issue?@action=redirect&bpo=37448]: Add a radix tree based memory map to track in-use obmalloc arenas. Use to replace the old implementation of address_in_range(). The radix tree approach makes it easy to increase pool sizes beyond the OS page size. Boosting the pool and arena size allows obmalloc to handle a significantly higher percentage of requests from its ultra-fast paths.

  It also has the advantage of eliminating the memory unsanitary behavior of the previous address_in_range(). The old address_in_range() was marked with the annotations _Py_NO_SANITIZE_ADDRESS, _Py_NO_SANITIZE_THREAD, and _Py_NO_SANITIZE_MEMORY. Those annotations are no longer needed.

  To disable the radix tree map, set a preprocessor flag as follows: `-DWITH_PYMALLOC_RADIX_TREE=0`.

  Co-authored-by: Tim Peters <tim.peters@gmail.com>

- bpo-29988 [https://bugs.python.org/issue?@action=redirect&bpo=29988]: Only handle asynchronous exceptions and requests to drop the GIL when returning from a call or on the back edges of loops. Makes sure that **`__exit__()`** is always called in with statements, even for interrupts.

Library

- bpo-43720 [https://bugs.python.org/issue?@action=redirect&bpo=43720]: Document various stdlib deprecations in imp, pkgutil, and importlib.util for removal in Python 3.12.

- bpo-43433 [https://bugs.python.org/issue?@action=redirect&bpo=43433]: **`xmlrpc.client.ServerProxy`** no longer ignores query and fragment in the URL of the server.

- bpo-31956 [https://bugs.python.org/issue?@action=redirect&bpo=31956]: The **`index()`** method of **`array.array`** now has optional *start* and *stop* parameters.

- bpo-40066 [https://bugs.python.org/issue?@action=redirect&bpo=40066]: Enum: adjust `repr()` to show only enum and member name (not value, nor angle brackets) and `str()` to show only member name. Update and improve documentation to match.

- bpo-42136 [https://bugs.python.org/issue?@action=redirect&bpo=42136]: Deprecate all module_repr() methods found in importlib as their use is being phased out by Python 3.12.

- bpo-35930 [https://bugs.python.org/issue?@action=redirect&bpo=35930]: Raising an exception raised in a “future” instance will create reference cycles.

- bpo-41369 [https://bugs.python.org/issue?@action=redirect&bpo=41369]: Finish updating the vendored libmpdec to version 2.5.1. Patch by Stefan Krah.

- bpo-43422 [https://bugs.python.org/issue?@action=redirect&bpo=43422]: Revert the _decimal C API which was added in bpo-41324 [https://bugs.python.org/issue?@action=redirect&bpo=41324].

- bpo-43577 [https://bugs.python.org/issue?@action=redirect&bpo=43577]: Fix deadlock when using **`ssl.SSLContext`** debug callback with **`ssl.SSLContext.sni_callback()`**.
- bpo-43571 [https://bugs.python.org/issue?@action=redirect&bpo=43571]: It’s now possible to create MPTCP sockets with IPPROTO_MPTCP
- bpo-43542 [https://bugs.python.org/issue?@action=redirect&bpo=43542]: `image/heic` and `image/heif` were added to **`mimetypes`**.
- bpo-40645 [https://bugs.python.org/issue?@action=redirect&bpo=40645]: The **`hmac`** module now uses OpenSSL’s HMAC implementation when digestmod argument is a hash name or builtin hash function.
- bpo-43510 [https://bugs.python.org/issue?@action=redirect&bpo=43510]: Implement **PEP 597** [https://peps.python.org/pep-0597/]: Add `EncodingWarning` warning, `-X warn_default_encoding` option, **`PYTHONWARNDEFAULTENCODING`** environment variable and `encoding="locale"` argument value.
- bpo-43521 [https://bugs.python.org/issue?@action=redirect&bpo=43521]: `ast.unparse` can now render NaNs and empty sets.
- bpo-42914 [https://bugs.python.org/issue?@action=redirect&bpo=42914]: **`pprint.pprint()`** gains a new boolean `underscore_numbers` optional argument to emit integers with thousands separated by an underscore

character for improved readability (for example `1_000_000` instead of `1000000`).

- bpo-41361 [https://bugs.python.org/issue?@action=redirect&bpo=41361]: **`rotate()`** calls are now slightly faster due to faster argument parsing.
- bpo-43423 [https://bugs.python.org/issue?@action=redirect&bpo=43423]: **`subprocess.communicate()`** no longer raises an IndexError when there is an empty stdout or stderr IO buffer during a timeout on Windows.
- bpo-27820 [https://bugs.python.org/issue?@action=redirect&bpo=27820]: Fixed long-standing bug of smtplib.SMTP where doing AUTH LOGIN with initial_response_ok=False will fail.

  The cause is that SMTP.auth_login _always_ returns a password if provided with a challenge string, thus non-compliant with the standard for AUTH LOGIN.

  Also fixes bug with the test for smtpd.
- bpo-43445 [https://bugs.python.org/issue?@action=redirect&bpo=43445]: Add frozen modules to **`sys.stdlib_module_names`**. For example, add `"_frozen_importlib"` and `"_frozen_importlib_external"` names.
- bpo-43245 [https://bugs.python.org/issue?@action=redirect&bpo=43245]: Add keyword arguments support to `ChainMap.new_child()`.
- bpo-29982 [https://bugs.python.org/issue?@action=redirect&bpo=29982]: Add optional parameter *ignore_cleanup_errors* to **`tempfile.TemporaryDirectory()`**

and allow multiple **`cleanup()`** attempts. Contributed by C.A.M. Gerlach.

- bpo-43428 [https://bugs.python.org/issue?@action=redirect&bpo=43428]: Include changes from importlib_metadata 3.7 [https://importlib-metadata.readthedocs.io/en/latest/history.html#v3-7-0]:

  Performance enhancements to distribution discovery.

  `entry_points` only returns unique distributions.

  Introduces new `EntryPoints` object for containing a set of entry points with convenience methods for selecting entry points by group or name. `entry_points` now returns this object if selection parameters are supplied but continues to return a dict object for compatibility. Users are encouraged to rely on the selection interface. The dict object result is likely to be deprecated in the future.

  Added packages_distributions function to return a mapping of packages to the distributions that provide them.

- bpo-43332 [https://bugs.python.org/issue?@action=redirect&bpo=43332]: Improves the networking efficiency of **`http.client`** when using a proxy via **`set_tunnel()`**. Fewer small send calls are made during connection setup.

- bpo-43420 [https://bugs.python.org/issue?@action=redirect&bpo=43420]: Improve performance of **`fractions.Fraction`** arithmetics for large components. Contributed by Sergey B. Kirpichev.

- bpo-43356 [https://bugs.python.org/issue?@action=redirect&bpo=43356]: Allow passing a signal number to `_thread.interrupt_main()`.

- bpo-43399 [https://bugs.python.org/issue?@action=redirect&bpo=43399]: Fix `ElementTree.extend` not working on iterators when using the Python implementation
- bpo-43369 [https://bugs.python.org/issue?@action=redirect&bpo=43369]: Improve **`sqlite3`** error handling: If `sqlite3_column_text()` and `sqlite3_column_blob()` set `SQLITE_NOMEM`, **`MemoryError`** is now raised. Patch by Erlend E. Aasland.
- bpo-43368 [https://bugs.python.org/issue?@action=redirect&bpo=43368]: Fix a regression introduced in PR 24562, where an empty bytestring was fetched as `None` instead of `b''` in **`sqlite3`**. Patch by Mariusz Felisiak.
- bpo-41282 [https://bugs.python.org/issue?@action=redirect&bpo=41282]: Fixed stacklevel of `DeprecationWarning` emitted from `import distutils`.
- bpo-42129 [https://bugs.python.org/issue?@action=redirect&bpo=42129]: `importlib.resources` now honors namespace packages, merging resources from each location in the namespace as introduced in `importlib_resources` 3.2 and including incidental changes through 5.0.3.
- bpo-43295 [https://bugs.python.org/issue?@action=redirect&bpo=43295]: **`datetime.datetime.strptime()`** now raises `ValueError` instead of `IndexError` when matching `'z'` with the `%z` format specifier.
- bpo-43125 [https://bugs.python.org/issue?@action=redirect&bpo=43125]: Return empty string if base64mime.body_encode receive empty bytes

- bpo-43084 [https://bugs.python.org/issue?@action=redirect&bpo=43084]: `curses.window.enclose()` returns now `True` or `False` (as was documented) instead of `1` or `0`.

- bpo-42994 [https://bugs.python.org/issue?@action=redirect&bpo=42994]: Add MIME types for opus, AAC, 3gpp and 3gpp2

- bpo-14678 [https://bugs.python.org/issue?@action=redirect&bpo=14678]: Add an invalidate_caches() method to the zipimport.zipimporter class to support importlib.invalidate_caches(). Patch by Desmond Cheong.

- bpo-42782 [https://bugs.python.org/issue?@action=redirect&bpo=42782]: Fail fast in `shutil.move()` to avoid creating destination directories on failure.

- bpo-40066 [https://bugs.python.org/issue?@action=redirect&bpo=40066]: Enum's `repr()` and `str()` have changed: `repr()` is now *EnumClass.MemberName* and `str()` is *MemberName*. Additionally, stdlib Enum's whose contents are available as module attributes, such as `RegexFlag.IGNORECASE`, have their `repr()` as *module.name*, e.g. `re.IGNORECASE`.

- bpo-26053 [https://bugs.python.org/issue?@action=redirect&bpo=26053]: Fixed bug where the `pdb` interactive run command echoed the args from the shell command line, even if those have been overridden at the pdb prompt.

- bpo-24160 [https://bugs.python.org/issue?@action=redirect&bpo=24160]: Fixed bug where breakpoints did not persist across multiple debugger sessions in `pdb`'s interactive mode.

- bpo-40701 [https://bugs.python.org/issue?@action=redirect&bpo=40701]: When the `tempfile.tempdir` global variable is set to a value of type bytes, it is now handled consistently. Previously exceptions could be raised from some tempfile APIs when the directory did not already exist in this situation. Also ensures that the `tempfile.gettempdir()` and `tempfile.gettempdirb()` functions *always* return `str` and `bytes` respectively.

- bpo-39342 [https://bugs.python.org/issue?@action=redirect&bpo=39342]: Expose `X509_V_FLAG_ALLOW_PROXY_CERTS` as `VERIFY_ALLOW_PROXY_CERTS` to allow proxy certificate validation as explained in https://docs.openssl.org/1.1.1/man7/proxy-certificates/.

- bpo-31861 [https://bugs.python.org/issue?@action=redirect&bpo=31861]: Add builtins.aiter and builtins.anext. Patch by Joshua Bronson (@jab), Daniel Pope (@lordmauve), and Justin Wang (@justin39).

Documentation

- bpo-43199 [https://bugs.python.org/issue?@action=redirect&bpo=43199]: Answer “Why is there no goto?” in the Design and History FAQ.
- bpo-43407 [https://bugs.python.org/issue?@action=redirect&bpo=43407]: Clarified that a result from `time.monotonic()`, `time.perf_counter()`, `time.process_time()`, or `time.thread_time()` can be compared with the result from any following call to the same function - not just the next immediate call.
- bpo-43354 [https://bugs.python.org/issue?@action=redirect&bpo=43354]: Fix type documentation for `Fault.faultCode`; the type has to be `int` instead of `str`.
- bpo-41933 [https://bugs.python.org/issue?@action=redirect&bpo=41933]: Clarified wording of s * n in the

Common Sequence Operations

Tests

- bpo-37945 [https://bugs.python.org/issue?@action=redirect&bpo=37945]: Fix test_getsetlocale_issue1813() of test_locale: skip the test if `setlocale()` fails. Patch by Victor Stinner.
- bpo-41561 [https://bugs.python.org/issue?@action=redirect&bpo=41561]: Add workaround for Ubuntu's custom OpenSSL security level policy.

Build

- bpo-43179 [https://bugs.python.org/issue?@action=redirect&bpo=43179]: Introduce and correctly use ALIGNOF_X in place of SIZEOF_X for alignment-related code in optimized string routines. Patch by Jessica Clarke.
- bpo-43631 [https://bugs.python.org/issue?@action=redirect&bpo=43631]: Update macOS, Windows, and CI to OpenSSL 1.1.1k.
- bpo-43617 [https://bugs.python.org/issue?@action=redirect&bpo=43617]: Improve configure.ac: Check for presence of autoconf-archive package and remove our copies of M4 macros.
- bpo-43466 [https://bugs.python.org/issue?@action=redirect&bpo=43466]: The `configure` script now supports `--with-openssl-rpath` option.
- bpo-43372 [https://bugs.python.org/issue?@action=redirect&bpo=43372]: Use `_freeze_importlib` to generate code for the `__hello__` module. This approach ensures the code matches the interpreter version. Previously, PYTHON_FOR_REGEN was used to generate the code, which might be wrong. The marshal format for code objects has changed with bpo-42246

[https://bugs.python.org/issue?@action=redirect&bpo=42246], commit 877df851. Update the code and the expected code sizes in ctypes test_frozentable.

Windows

- bpo-43440 [https://bugs.python.org/issue?@action=redirect&bpo=43440]: Build **`sqlite3`** with the `R*Tree` module enabled. Patch by Erlend E. Aasland.

IDLE

- bpo-42225 [https://bugs.python.org/issue?@action=redirect&bpo=42225]: Document that IDLE can fail on Unix either from misconfigured IP masquerade rules or failure displaying complex colored (non-ascii) characters.

C API

- bpo-43688 [https://bugs.python.org/issue?@action=redirect&bpo=43688]: The limited C API is now supported if Python is built in debug mode (if the `Py_DEBUG` macro is defined). In the limited C API, the **`Py_INCREF()`** and **`Py_DECREF()`** functions are now implemented as opaque function calls, rather than accessing directly the **`PyObject.ob_refcnt`** member, if Python is built in debug mode and the `Py_LIMITED_API` macro targets Python 3.10 or newer. It became possible to support the limited C API in debug mode because the **`PyObject`** structure is the same in release and debug mode since Python 3.8 (see bpo-36465 [https://bugs.python.org/issue?@action=redirect&bpo=36465]).

  The limited C API is still not supported in the `--with-trace-refs` special build (`Py_TRACE_REFS` macro).

Patch by Victor Stinner.

- bpo-43244 [https://bugs.python.org/issue?@action=redirect&bpo=43244]: Remove the `pyarena.h` header file with functions:

    - `PyArena_New()`
    - `PyArena_Free()`
    - `PyArena_Malloc()`
    - `PyArena_AddPyObject()`

  These functions were undocumented, excluded from the limited C API, and were only used internally by the compiler. Patch by Victor Stinner.

- bpo-43244 [https://bugs.python.org/issue?@action=redirect&bpo=43244]: Remove the compiler and parser functions using `struct _mod` type, because the public AST C API was removed:

    - `PyAST_Compile()`
    - `PyAST_CompileEx()`
    - `PyAST_CompileObject()`
    - `PyFuture_FromAST()`
    - `PyFuture_FromASTObject()`
    - `PyParser_ASTFromFile()`
    - `PyParser_ASTFromFileObject()`
    - `PyParser_ASTFromFilename()`
    - `PyParser_ASTFromString()`
    - `PyParser_ASTFromStringObject()`

  These functions were undocumented and excluded from the limited C API. Patch by Victor Stinner.

- bpo-43244 [https://bugs.python.org/issue?@action=redirect&bpo=43244]: Remove `ast.h`, `asdl.h`, and `Python-ast.h` header files. These functions were undocumented and excluded from the limited C API. Most names defined by these header files were not prefixed by `Py` and so could create names conflicts. For example, `Python-ast.h` defined a `Yield` macro which was conflict with the `Yield` name used by the Windows `<winbase.h>` header. Use the Python **`ast`** module instead. Patch by Victor Stinner.

- bpo-43541 [https://bugs.python.org/issue?@action=redirect&bpo=43541]: Fix a `PyEval_EvalCodeEx()` regression: fix reference counting on builtins. Patch by Victor Stinner.
- bpo-43244 [https://bugs.python.org/issue?@action=redirect&bpo=43244]: Remove the `symtable.h` header file and the undocumented functions:

    - `PyST_GetScope()`
    - `PySymtable_Build()`
    - `PySymtable_BuildObject()`
    - `PySymtable_Free()`
    - `Py_SymtableString()`
    - `Py_SymtableStringObject()`

  The `Py_SymtableString()` function was part the stable ABI by mistake but it could not be used, because the `symtable.h` header file was excluded from the limited C API.

  The Python **`symtable`** module remains available and is unchanged.

  Patch by Victor Stinner.

- bpo-43244 [https://bugs.python.org/issue?@action=redirect&bpo=43244]: Remove the `PyAST_Validate()` function. It is no longer possible to build a AST object (`mod_ty` type) with the public C API. The function was already excluded from the limited C API (**PEP 384** [https://peps.python.org/pep-0384/]). Patch by Victor Stinner.

Python 3.10.0 alpha 6

Release date: 2021-03-01

Security

- bpo-42967 [https://bugs.python.org/issue?@action=redirect&bpo=42967]: Fix web cache poisoning vulnerability by defaulting the query args separator to `&`, and allowing the user to choose a custom separator.

Core and Builtins

- bpo-43321 [https://bugs.python.org/issue?@action=redirect&bpo=43321]: Fix `SystemError` raised when `PyArg_Parse*()` is used with `#` but without `PY_SSIZE_T_CLEAN` defined.
- bpo-36346 [https://bugs.python.org/issue?@action=redirect&bpo=36346]: `PyArg_Parse*()` functions now emits `DeprecationWarning` when `u` or `Z` format is used. See **PEP 623** [https://peps.python.org/pep-0623/] for detail.
- bpo-43277 [https://bugs.python.org/issue?@action=redirect&bpo=43277]: Add a new **`PySet_CheckExact()`** function to the C-API to check if an object is an instance of **`set`** but not an instance of a subtype. Patch by Pablo Galindo.
- bpo-42990 [https://bugs.python.org/issue?@action=redirect&bpo=42990]: The **`types.FunctionType`** constructor now inherits the current builtins if the *globals* dictionary has no `"__builtins__"` key, rather than using `{"None": None}` as builtins: same behavior as **`eval()`**

and **`exec()`** functions. Defining a function with `def function(...): ...` in Python is not affected, globals cannot be overridden with this syntax: it also inherits the current builtins. Patch by Victor Stinner.

- bpo-42990 [https://bugs.python.org/issue?@action=redirect&bpo=42990]: Functions have a new `__builtins__` attribute which is used to look for builtin symbols when a function is executed, instead of looking into `__globals__['__builtins__']`. Patch by Mark Shannon and Victor Stinner.
- bpo-43149 [https://bugs.python.org/issue?@action=redirect&bpo=43149]: Improve the error message in the parser for exception groups without parentheses. Patch by Pablo Galindo.
- bpo-43121 [https://bugs.python.org/issue?@action=redirect&bpo=43121]: Fixed an incorrect **`SyntaxError`** message for missing comma in literals. Patch by Pablo Galindo.

Library

- bpo-42819 [https://bugs.python.org/issue?@action=redirect&bpo=42819]: **`readline`**: Explicitly disable bracketed paste in the interactive interpreter, even if it’s set in the inputrc, is enabled by default (eg GNU Readline 8.1), or a user calls `readline.read_init_file()`. The Python REPL has not implemented bracketed paste support. Also, bracketed mode writes the `"\x1b[?2004h"` escape sequence into stdout which causes test failures in applications that don’t support it. It can still be explicitly enabled by calling `readline.parse_and_bind("set enable-bracketed-paste on")`. Patch by Dustin Rodrigues.

Core and Builtins

- bpo-42808 [https://bugs.python.org/issue?@action=redirect&bpo=42808]: Simple calls to `type(object)` are now faster due to the `vectorcall` calling convention. Patch by Dennis Sweeney.
- bpo-42217 [https://bugs.python.org/issue?@action=redirect&bpo=42217]: Make the compiler merges same co_code and co_linetable objects in a module like already did for co_consts.
- bpo-41972 [https://bugs.python.org/issue?@action=redirect&bpo=41972]: Substring search functions such as `str1 in str2` and `str2.find(str1)` now sometimes use the “Two-Way” string comparison algorithm to avoid quadratic behavior on long strings.
- bpo-42128 [https://bugs.python.org/issue?@action=redirect&bpo=42128]: Implement **PEP 634** [https://peps.python.org/pep-0634/] (structural pattern matching). Patch by Brandt Bucher.

Library

- bpo-40692 [https://bugs.python.org/issue?@action=redirect&bpo=40692]: In the `concurrent.futures.ProcessPoolExecutor`, validate that **`multiprocess.synchronize()`** is available on a given platform and rely on that check in the `concurrent.futures` test suite so we can run tests that are unrelated to **`ProcessPoolExecutor`** on those platforms.

Core and Builtins

- bpo-38302 [https://bugs.python.org/issue?@action=redirect&bpo=38302]: If `object.__ipow__()` returns `NotImplemented`, the operator will correctly fall back to `object.__pow__()` and `object.__rpow__()` as expected.

Library

- bpo-43316 [https://bugs.python.org/issue?@action=redirect&bpo=43316]: The `python -m gzip` command line application now properly fails when detecting an unsupported extension. It exits with a non-zero exit code and prints an error message to stderr.
- bpo-43317 [https://bugs.python.org/issue?@action=redirect&bpo=43317]: Set the chunk size for the `gzip` module main function to io.DEFAULT_BUFFER_SIZE. This is slightly faster than the 1024 bytes constant that was used previously.
- bpo-43146 [https://bugs.python.org/issue?@action=redirect&bpo=43146]: Handle None in single-arg versions of **`print_exception()`** and **`format_exception()`**.
- bpo-43260 [https://bugs.python.org/issue?@action=redirect&bpo=43260]: Fix TextIOWrapper can not flush internal buffer forever after very large text is written.
- bpo-43258 [https://bugs.python.org/issue?@action=redirect&bpo=43258]: Prevent needless allocation of **`sqlite3`** aggregate function context when no rows match an aggregate query. Patch by Erlend E. Aasland.
- bpo-43251 [https://bugs.python.org/issue?@action=redirect&bpo=43251]: Improve **`sqlite3`** error handling: `sqlite3_column_name()` failures now result in **`MemoryError`**. Patch by Erlend E. Aasland.
- bpo-40956 [https://bugs.python.org/issue?@action=redirect&bpo=40956]: Fix segfault in **`sqlite3.Connection.backup()`** if no argument was provided. The regression was introduced by PR 23838. Patch by Erlend E. Aasland.
- bpo-43172 [https://bugs.python.org/issue?@action=redirect&bpo=43172]: The readline module now passes its tests when built directly against libedit. Existing irreconcilable API differences remain in

`readline.get_begidx()` and `readline.get_endidx()` behavior based on libreadline vs libedit use.

- bpo-43163 [https://bugs.python.org/issue?@action=redirect&bpo=43163]: Fix a bug in `codeop` that was causing it to not ask for more input when multi-line snippets have unclosed parentheses. Patch by Pablo Galindo
- bpo-43162 [https://bugs.python.org/issue?@action=redirect&bpo=43162]: deprecate unsupported ability to access enum members as attributes of other enum members
- bpo-43146 [https://bugs.python.org/issue?@action=redirect&bpo=43146]: Fix recent regression in None argument handling in `traceback` module functions.
- bpo-43102 [https://bugs.python.org/issue?@action=redirect&bpo=43102]: The namedtuple __new__ method had its __builtins__ set to None instead of an actual dictionary. This created problems for introspection tools.
- bpo-43106 [https://bugs.python.org/issue?@action=redirect&bpo=43106]: Added `O_EVTONLY`, `O_FSYNC`, `O_SYMLINK` and `O_NOFOLLOW_ANY` for macOS. Patch by Donghee Na.
- bpo-42960 [https://bugs.python.org/issue?@action=redirect&bpo=42960]: Adds `resource.RLIMIT_KQUEUES` constant from FreeBSD to the `resource` module.
- bpo-42151 [https://bugs.python.org/issue?@action=redirect&bpo=42151]: Make the pure Python implementation of `xml.etree.ElementTree` behave the same as the C implementation (`_elementree`) regarding default attribute values (by not setting `specified_attributes=1`).
- bpo-29753 [https://bugs.python.org/issue?@action=redirect&bpo=29753]: In ctypes, now packed bitfields are calculated properly and the first item of packed bitfields is now shrank correctly.

Documentation

- bpo-27646 [https://bugs.python.org/issue?@action=redirect&bpo=27646]: Clarify that 'yield from <expr>' works with any iterable, not just iterators.
- bpo-36346 [https://bugs.python.org/issue?@action=redirect&bpo=36346]: Update some deprecated unicode APIs which are documented as "will be removed in 4.0" to "3.12". See **PEP 623** [https://peps.python.org/pep-0623/] for detail.

Tests

- bpo-43288 [https://bugs.python.org/issue?@action=redirect&bpo=43288]: Fix test_importlib to correctly skip Unicode file tests if the filesystem does not support them.

Build

- bpo-43174 [https://bugs.python.org/issue?@action=redirect&bpo=43174]: Windows build now uses `/utf-8` compiler option.

- bpo-43103 [https://bugs.python.org/issue?@action=redirect&bpo=43103]: Add a new configure `--without-static-libpython` option to not build the `libpythonMAJOR.MINOR.a` static library and not install the `python.o` object file.

- bpo-13501 [https://bugs.python.org/issue?@action=redirect&bpo=13501]: The configure script can now use *libedit* instead of *readline* with the command line option `--with-readline=editline`.

- bpo-42603 [https://bugs.python.org/issue?@action=redirect&bpo=42603]: Make configure script use pkg-config to detect the location of Tcl/Tk headers and libraries, used to build tkinter.

  On macOS, a Tcl/Tk configuration provided by pkg-config will be preferred over Tcl/Tk frameworks installed in `/{System/,}Library/Frameworks`. If both exist and the latter is preferred, the appropriate `--with-tcltk-*` configuration options need to be explicitly set.

- bpo-39448 [https://bugs.python.org/issue?@action=redirect&bpo=39448]: Add the "regen-frozen" makefile target that regenerates the code for the frozen `__hello__` module.

Windows

- bpo-43155 [https://bugs.python.org/issue?@action=redirect&bpo=43155]: **`PyCMethod_New()`** is now present in `python3.lib`.

macOS

- bpo-41837 [https://bugs.python.org/issue?@action=redirect&bpo=41837]: Update macOS installer build to use OpenSSL 1.1.1j.

IDLE

- bpo-43283 [https://bugs.python.org/issue?@action=redirect&bpo=43283]: Document why printing to IDLE's Shell is often slower than printing to a system terminal and that it can be made faster by pre-formatting a single string before printing.

C API

- bpo-43278 [https://bugs.python.org/issue?@action=redirect&bpo=43278]: Always put compiler and system information on the first line of the REPL welcome message.
- bpo-43270 [https://bugs.python.org/issue?@action=redirect&bpo=43270]: Remove the private `_PyErr_OCCURRED()` macro: use the public **`PyErr_Occurred()`** function instead.
- bpo-35134 [https://bugs.python.org/issue?@action=redirect&bpo=35134]: Move odictobject.h, parser_interface.h, picklebufobject.h, pydebug.h, and pyfpe.h into the cpython/ directory. They must not be included directly, as they are already included by Python.h: Include Files.
- bpo-35134 [https://bugs.python.org/issue?@action=redirect&bpo=35134]: Move pyarena.h, pyctype.h, and pytime.h into the cpython/ directory. They must not be included directly, as they are already included by Python.h: Include Files.
- bpo-40170 [https://bugs.python.org/issue?@action=redirect&bpo=40170]: **`PyExceptionClass_Name()`** is now always declared as a function, in order to hide implementation details. The macro accessed **`PyTypeObject.tp_name`** directly. Patch by Erlend E. Aasland.
- bpo-43239 [https://bugs.python.org/issue?@action=redirect&bpo=43239]: The **`PyCFunction_New()`** function is now exported in the ABI when compiled with `-fvisibility=hidden`.
- bpo-40170 [https://bugs.python.org/issue?@action=redirect&bpo=40170]: **`PyIter_Check()`** is now always declared as a function, in order to hide implementation details. The macro accessed **`PyTypeObject.tp_iternext`** directly. Patch by Erlend E. Aasland.

- bpo-40170 [https://bugs.python.org/issue?@action=redirect&bpo=40170]: Convert `PyDescr_IsData()` macro to a function to hide implementation details: The macro accessed `PyTypeObject.tp_descr_set` directly. Patch by Erlend E. Aasland.
- bpo-43181 [https://bugs.python.org/issue?@action=redirect&bpo=43181]: Convert `PyObject_TypeCheck()` macro to a static inline function. Patch by Erlend E. Aasland.

Python 3.10.0 alpha 5

Release date: 2021-02-02

Security

- bpo-42938 [https://bugs.python.org/issue?@action=redirect&bpo=42938]: Avoid static buffers when computing the repr of `ctypes.c_double` and `ctypes.c_longdouble` values.

Core and Builtins

- bpo-42990 [https://bugs.python.org/issue?@action=redirect&bpo=42990]: Refactor the `PyEval_` family of functions.
    - An new function `_PyEval_Vector` is added to simplify calls to Python from C.
    - `_PyEval_EvalCodeWithName` is removed
    - `PyEval_EvalCodeEx` is retained as part of the API, but is not used internally
- bpo-38631 [https://bugs.python.org/issue?@action=redirect&bpo=38631]: Replace `Py_FatalError()` calls in the compiler with regular `SystemError` exceptions. Patch by Victor Stinner.

- bpo-42997 [https://bugs.python.org/issue?@action=redirect&bpo=42997]: Improve error message for missing ":" before blocks. Patch by Pablo Galindo.
- bpo-43017 [https://bugs.python.org/issue?@action=redirect&bpo=43017]: Improve error message in the parser when using un-parenthesised tuples in comprehensions. Patch by Pablo Galindo.
- bpo-42986 [https://bugs.python.org/issue?@action=redirect&bpo=42986]: Fix parser crash when reporting syntax errors in f-string with newlines. Patch by Pablo Galindo.
- bpo-40176 [https://bugs.python.org/issue?@action=redirect&bpo=40176]: Syntax errors for unterminated string literals now point to the start of the string instead of reporting EOF/EOL.
- bpo-42927 [https://bugs.python.org/issue?@action=redirect&bpo=42927]: The inline cache for `LOAD_ATTR` now also optimizes access to attributes defined by `__slots__`. This makes reading such attribute up to 30% faster.
- bpo-42864 [https://bugs.python.org/issue?@action=redirect&bpo=42864]: Improve error messages in the parser when parentheses are not closed. Patch by Pablo Galindo.
- bpo-42924 [https://bugs.python.org/issue?@action=redirect&bpo=42924]: Fix `bytearray` repetition incorrectly copying data from the start of the buffer, even if the data is offset within the buffer (e.g. after reassigning a slice at the start of the `bytearray` to a shorter byte string).
- bpo-42882 [https://bugs.python.org/issue?@action=redirect&bpo=42882]: Fix the **`_PyUnicode_FromId()`** function (_Py_IDENTIFIER(var) API) when **`Py_Initialize()`** / **`Py_Finalize()`** is called multiple times: preserve `_PyRuntime.unicode_ids.next_index` value.

- bpo-42827 [https://bugs.python.org/issue?@action=redirect&bpo=42827]: Fix a crash when working out the error line of a **`SyntaxError`** in some multi-line expressions.
- bpo-42823 [https://bugs.python.org/issue?@action=redirect&bpo=42823]: frame.f_lineno is correct even if frame.f_trace is set to True

Library

- bpo-37324 [https://bugs.python.org/issue?@action=redirect&bpo=37324]: Remove deprecated aliases to Collections Abstract Base Classes from the **`collections`** module.

Core and Builtins

- bpo-41994 [https://bugs.python.org/issue?@action=redirect&bpo=41994]: Fixed possible leak in `import` when `sys.modules` is not a `dict`.
- bpo-27772 [https://bugs.python.org/issue?@action=redirect&bpo=27772]: In string formatting, preceding the *width* field by `'0'` no longer affects the default alignment for strings.

Library

- bpo-43108 [https://bugs.python.org/issue?@action=redirect&bpo=43108]: Fixed a reference leak in the **`curses`** module. Patch by Pablo Galindo
- bpo-43077 [https://bugs.python.org/issue?@action=redirect&bpo=43077]: Update the bundled pip to 21.0.1 and setuptools to 52.0.0.
- bpo-41282 [https://bugs.python.org/issue?@action=redirect&bpo=41282]: Deprecate `distutils` in documentation and add warning on import.

- bpo-43014 [https://bugs.python.org/issue?@action=redirect&bpo=43014]: Improve performance of **`tokenize`** by 20-30%. Patch by Anthony Sottile.
- bpo-42323 [https://bugs.python.org/issue?@action=redirect&bpo=42323]: Fix **`math.nextafter()`** for NaN on AIX.
- bpo-42955 [https://bugs.python.org/issue?@action=redirect&bpo=42955]: Add **`sys.stdlib_module_names`**, containing the list of the standard library module names. Patch by Victor Stinner.
- bpo-42944 [https://bugs.python.org/issue?@action=redirect&bpo=42944]: Fix `random.Random.sample` when `counts` argument is not `None`.
- bpo-42934 [https://bugs.python.org/issue?@action=redirect&bpo=42934]: Use **`TracebackException`**’s new `compact` param in **`TestResult`** to reduce time and memory consumed by traceback formatting.
- bpo-42931 [https://bugs.python.org/issue?@action=redirect&bpo=42931]: Add **`randbytes()`** to `random.__all__`.
- bpo-38250 [https://bugs.python.org/issue?@action=redirect&bpo=38250]: [Enum] Flags consisting of a single bit are now considered canonical, and will be the only flags returned from listing and iterating over a Flag class or a Flag member. Multi-bit flags are considered aliases; they will be returned from lookups and operations that result in their value. Iteration for both Flag and Flag members is in definition order.
- bpo-42877 [https://bugs.python.org/issue?@action=redirect&bpo=42877]: Added the `compact` parameter to the constructor of **`traceback.TracebackException`** to reduce time and memory for use cases that only need to call **`TracebackException.format()`** and **`TracebackException.format_exception_only()`**.
- bpo-42923 [https://bugs.python.org/issue?@action=redirect&bpo=42923]: The **`Py_FatalError()`** function

and the **`faulthandler`** module now dump the list of extension modules on a fatal error.

- bpo-42848 [https://bugs.python.org/issue?@action=redirect&bpo=42848]: Removed recursion from **`TracebackException`** to allow it to handle long exception chains.
- bpo-42901 [https://bugs.python.org/issue?@action=redirect&bpo=42901]: [Enum] move member creation from `EnumMeta.__new__` to `_proto_member.__set_name__`, allowing members to be created and visible in `__init_subclass__`.
- bpo-42780 [https://bugs.python.org/issue?@action=redirect&bpo=42780]: Fix os.set_inheritable() for O_PATH file descriptors on Linux.
- bpo-42866 [https://bugs.python.org/issue?@action=redirect&bpo=42866]: Fix a reference leak in the `getcodec()` function of CJK codecs. Patch by Victor Stinner.
- bpo-42846 [https://bugs.python.org/issue?@action=redirect&bpo=42846]: Convert the 6 CJK codec extension modules (_codecs_cn, _codecs_hk, _codecs_iso2022, _codecs_jp, _codecs_kr and _codecs_tw) to the multiphase initialization API (**PEP 489** [https://peps.python.org/pep-0489/]). Patch by Victor Stinner.
- bpo-42851 [https://bugs.python.org/issue?@action=redirect&bpo=42851]: remove __init_subclass__ support for Enum members
- bpo-42834 [https://bugs.python.org/issue?@action=redirect&bpo=42834]: Make internal caches of the `_json` module compatible with subinterpreters.
- bpo-41748 [https://bugs.python.org/issue?@action=redirect&bpo=41748]: Fix HTMLParser parsing rules for element attributes containing commas with spaces. Patch by Karl Dubost.
- bpo-40810 [https://bugs.python.org/issue?@action=redirect&bpo=40810]: Require SQLite 3.7.15 or newer.

Patch by Erlend E. Aasland.

- bpo-1635741 [https://bugs.python.org/issue?@action=redirect&bpo=1635741]: Convert the _multibytecodec extension module (CJK codecs) to multi-phase initialization (**PEP 489** [https://peps.python.org/pep-0489/]). Patch by Erlend E. Aasland.
- bpo-42802 [https://bugs.python.org/issue?@action=redirect&bpo=42802]: The distutils `bdist_wininst` command deprecated in Python 3.8 has been removed. The distutils `bdist_wheel` command is now recommended to distribute binary packages on Windows.
- bpo-24464 [https://bugs.python.org/issue?@action=redirect&bpo=24464]: The undocumented built-in function `sqlite3.enable_shared_cache` is now deprecated, scheduled for removal in Python 3.12. Its use is strongly discouraged by the SQLite3 documentation. Patch by Erlend E. Aasland.
- bpo-42384 [https://bugs.python.org/issue?@action=redirect&bpo=42384]: Make pdb populate sys.path[0] exactly the same as regular python execution.
- bpo-42383 [https://bugs.python.org/issue?@action=redirect&bpo=42383]: Fix pdb: previously pdb would fail to restart the debugging target if it was specified using a relative path and the current directory changed.
- bpo-42005 [https://bugs.python.org/issue?@action=redirect&bpo=42005]: Fix CLI of **`cProfile`** and **`profile`** to catch **`BrokenPipeError`**.
- bpo-41604 [https://bugs.python.org/issue?@action=redirect&bpo=41604]: Don’t decrement the reference count of the previous user_ptr when set_panel_userptr fails.
- bpo-41149 [https://bugs.python.org/issue?@action=redirect&bpo=41149]: Allow executing callables that have a boolean value of `False` when passed to **`Threading.thread`** as the target. Patch contributed by Barney Stratford.

- bpo-38307 [https://bugs.python.org/issue?@action=redirect&bpo=38307]: Add an ‘end_lineno’ attribute to the Class and Function objects that appear in the tree returned by pyclbr functions. This and the existing ‘lineno’ attribute define the extent of class and def statements. Patch by Aviral Srivastava.
- bpo-39273 [https://bugs.python.org/issue?@action=redirect&bpo=39273]: The `BUTTON5_*` constants are now exposed in the **`curses`** module if available.
- bpo-33289 [https://bugs.python.org/issue?@action=redirect&bpo=33289]: Correct call to **`tkinter.colorchooser`** to return RGB triplet of ints instead of floats. Patch by Cheryl Sabella.

Documentation

- bpo-40304 [https://bugs.python.org/issue?@action=redirect&bpo=40304]: Fix doc for type(name, bases, dict). Patch by Boris Verkhovskiy and Éric Araujo.
- bpo-42811 [https://bugs.python.org/issue?@action=redirect&bpo=42811]: Updated importlib.util.resolve_name() doc to use __spec__.parent instead of __package__. (Thanks Yair Frid.)

Tests

- bpo-40823 [https://bugs.python.org/issue?@action=redirect&bpo=40823]: Use **`unittest.TestLoader().loadTestsFromTestCase()`** instead of **`unittest.makeSuite()`** in **`sqlite3`** tests. Patch by Erlend E. Aasland.
- bpo-40810 [https://bugs.python.org/issue?@action=redirect&bpo=40810]: In **`sqlite3`**, fix `CheckTraceCallbackContent` for SQLite pre 3.7.15.

Build

- bpo-43031 [https://bugs.python.org/issue?@action=redirect&bpo=43031]: Pass `--timeout=$(TESTTIMEOUT)` option to the default profile task `./python -m test --pgo` command.

- bpo-36143 [https://bugs.python.org/issue?@action=redirect&bpo=36143]: `make regen-all` now also runs `regen-keyword`. Patch by Victor Stinner.

- bpo-42874 [https://bugs.python.org/issue?@action=redirect&bpo=42874]: Removed the grep -q and -E flags in the tzpath validation section of the configure script to better accommodate users of some platforms (specifically Solaris 10).

- bpo-31904 [https://bugs.python.org/issue?@action=redirect&bpo=31904]: Add library search path by wr-cc in add_cross_compiling_paths() for VxWorks.

- bpo-42856 [https://bugs.python.org/issue?@action=redirect&bpo=42856]: Add `--with-wheel-pkg-dir=PATH` option to the `./configure` script. If specified, the **`ensurepip`** module looks for `setuptools` and `pip` wheel packages in this directory: if both are present, these wheel packages are used instead of ensurepip bundled wheel packages.

  Some Linux distribution packaging policies recommend against bundling dependencies. For example, Fedora installs wheel packages in the `/usr/share/python-wheels/` directory and don’t install the `ensurepip._bundled` package.

Windows

- bpo-41837 [https://bugs.python.org/issue?@action=redirect&bpo=41837]: Updated Windows installer to include OpenSSL 1.1.1i
- bpo-42584 [https://bugs.python.org/issue?@action=redirect&bpo=42584]: Upgrade Windows installer to use SQLite 3.34.0.

macOS

- bpo-42504 [https://bugs.python.org/issue?@action=redirect&bpo=42504]: Ensure that the value of sysconfig.get_config_var('MACOSX_DEPLOYMENT_TARGET') is always a string, even in when the value is parsable as an integer.

IDLE

- bpo-43008 [https://bugs.python.org/issue?@action=redirect&bpo=43008]: Make IDLE invoke `sys.excepthook()` in normal, 2-process mode. Patch by Ken Hilton.
- bpo-33065 [https://bugs.python.org/issue?@action=redirect&bpo=33065]: Fix problem debugging user classes with __repr__ method.
- bpo-23544 [https://bugs.python.org/issue?@action=redirect&bpo=23544]: Disable Debug=>Stack Viewer when user code is running or Debugger is active, to prevent hang or crash. Patch by Zackery Spytz.
- bpo-32631 [https://bugs.python.org/issue?@action=redirect&bpo=32631]: Finish zzdummy example extension module: make menu entries work; add docstrings and tests with 100% coverage.

C API

- bpo-42979 [https://bugs.python.org/issue?@action=redirect&bpo=42979]: When Python is built in debug mode (with C assertions), calling a type slot like `sq_length` (`__len__()` in Python) now fails with a fatal error if the slot succeeded with an exception set, or failed with no exception set. The error message contains the slot, the type name, and the current exception (if an exception is set). Patch by Victor Stinner.
- bpo-43030 [https://bugs.python.org/issue?@action=redirect&bpo=43030]: Fixed a compiler warning in **`Py_UNICODE_ISSPACE()`** on platforms with signed **`wchar_t`**.

Python 3.10.0 alpha 4

Release date: 2021-01-04

Core and Builtins

- bpo-42814 [https://bugs.python.org/issue?@action=redirect&bpo=42814]: Fix undefined behavior in `Objects/genericaliasobject.c`.
- bpo-42806 [https://bugs.python.org/issue?@action=redirect&bpo=42806]: Fix the column offsets for f-strings **`ast`** nodes surrounded by parentheses and for nodes that spawn multiple lines. Patch by Pablo Galindo.
- bpo-40631 [https://bugs.python.org/issue?@action=redirect&bpo=40631]: Fix regression where a single parenthesized starred expression was a valid assignment target.
- bpo-27794 [https://bugs.python.org/issue?@action=redirect&bpo=27794]: Improve the error message for failed writes/deletes to property objects. When possible, the attribute name is now shown. Patch provided by Yurii Karabas.

- bpo-42745 [https://bugs.python.org/issue?@action=redirect&bpo=42745]: Make the type attribute lookup cache per-interpreter. Patch by Victor Stinner.
- bpo-42246 [https://bugs.python.org/issue?@action=redirect&bpo=42246]: Jumps to jumps are not eliminated when it would break PEP 626.
- bpo-42246 [https://bugs.python.org/issue?@action=redirect&bpo=42246]: Make sure that the `f_lasti` and `f_lineno` attributes of a frame are set correctly when an exception is raised or re-raised. Required for PEP 626.
- bpo-32381 [https://bugs.python.org/issue?@action=redirect&bpo=32381]: The coding cookie (ex: `# coding: latin1`) is now ignored in the command passed to the **`-c`** command line option. Patch by Victor Stinner.
- bpo-30858 [https://bugs.python.org/issue?@action=redirect&bpo=30858]: Improve error location in expressions that contain assignments. Patch by Pablo Galindo and Lysandros Nikolaou.
- bpo-42615 [https://bugs.python.org/issue?@action=redirect&bpo=42615]: Remove jump commands made redundant by the deletion of unreachable bytecode blocks

Library

- bpo-42639 [https://bugs.python.org/issue?@action=redirect&bpo=42639]: Make the **`atexit`** module state per-interpreter. It is now safe have more than one **`atexit`** module instance. Patch by Donghee Na and Victor Stinner.

Core and Builtins

- bpo-32381 [https://bugs.python.org/issue?@action=redirect&bpo=32381]: Fix encoding name when running a `.pyc` file on Windows: **`PyRun_SimpleFileExFlags()`** now uses the correct encoding to decode the filename.

Library

- bpo-42195 [https://bugs.python.org/issue?@action=redirect&bpo=42195]: The `__args__` of the parameterized generics for **`typing.Callable`** and **`collections.abc.Callable`** are now consistent. The `__args__` for **`collections.abc.Callable`** are now flattened while **`typing.Callable`**'s have not changed. To allow this change, **`types.GenericAlias`** can now be subclassed and `collections.abc.Callable`'s `__class_getitem__` will now return a subclass of `types.GenericAlias`. Tests for typing were also updated to not subclass things like `Callable[..., T]` as that is not a valid base class. Finally, both `Callable`s no longer validate their `argtypes`, in `Callable[[argtypes], resulttype]` to prepare for **PEP 612** [https://peps.python.org/pep-0612/]. Patch by Ken Jin.
- bpo-40137 [https://bugs.python.org/issue?@action=redirect&bpo=40137]: Convert functools module to use **`PyType_FromModuleAndSpec()`**.
- bpo-40077 [https://bugs.python.org/issue?@action=redirect&bpo=40077]: Convert **`array`** to use heap types, and establish module state for these.
- bpo-42008 [https://bugs.python.org/issue?@action=redirect&bpo=42008]: Fix _random.Random() seeding.
- bpo-1635741 [https://bugs.python.org/issue?@action=redirect&bpo=1635741]: Port the **`pyexpat`** extension module to multi-phase initialization (**PEP 489** [https://peps.python.org/pep-0489/]).

Core and Builtins

- bpo-40521 [https://bugs.python.org/issue?@action=redirect&bpo=40521]: Make the Unicode dictionary of interned strings compatible with subinterpreters. Patch by Victor Stinner.

- bpo-39465 [https://bugs.python.org/issue?@action=redirect&bpo=39465]: Make **`_PyUnicode_FromId()`** function compatible with subinterpreters. Each interpreter now has an array of identifier objects (interned strings decoded from UTF-8). Patch by Victor Stinner.

Library

- bpo-42257 [https://bugs.python.org/issue?@action=redirect&bpo=42257]: Handle empty string in variable executable in platform.libc_ver()

- bpo-42772 [https://bugs.python.org/issue?@action=redirect&bpo=42772]: randrange() now raises a TypeError when step is specified without a stop argument. Formerly, it silently ignored the step argument.

- bpo-42759 [https://bugs.python.org/issue?@action=redirect&bpo=42759]: Fixed equality comparison of **`tkinter.Variable`** and **`tkinter.font.Font`**. Objects which belong to different Tcl interpreters are now always different, even if they have the same name.

- bpo-42756 [https://bugs.python.org/issue?@action=redirect&bpo=42756]: Configure LMTP Unix-domain socket to use socket global default timeout when a timeout is not explicitly provided.

- bpo-23328 [https://bugs.python.org/issue?@action=redirect&bpo=23328]: Allow / character in username, password fields on _PROXY envars.

- bpo-42740 [https://bugs.python.org/issue?@action=redirect&bpo=42740]: **`typing.get_args()`** and **`typing.get_origin()`** now support **PEP 604** [https://peps.python.org/pep-0604/] union types and **PEP 612** [https://peps.python.org/pep-0612/] additions to `Callable`.

- bpo-42655 [https://bugs.python.org/issue?@action=redirect&bpo=42655]: **`subprocess`** *extra_groups* is now correctly passed into setgroups() system call.

- bpo-42727 [https://bugs.python.org/issue?@action=redirect&bpo=42727]: `EnumMeta.__prepare__` now accepts `**kwds` to properly support `__init_subclass__`

- bpo-38308 [https://bugs.python.org/issue?@action=redirect&bpo=38308]: Add optional *weights* to *statistics.harmonic_mean()*.

- bpo-42721 [https://bugs.python.org/issue?@action=redirect&bpo=42721]: When simple query dialogs (**`tkinter.simpledialog`**), message boxes (**`tkinter.messagebox`**) or color choose dialog (**`tkinter.colorchooser`**) are created without arguments *master* and *parent*, and the default root window is not yet created, and **`NoDefaultRoot()`** was not called, a new temporal hidden root window will be created automatically. It will not be set as the default root window and will be destroyed right after closing the dialog window. It will help to use these simple dialog windows in programs which do not need other GUI.

- bpo-25246 [https://bugs.python.org/issue?@action=redirect&bpo=25246]: Optimized **`collections.deque.remove()`**.

- bpo-35728 [https://bugs.python.org/issue?@action=redirect&bpo=35728]: Added a root parameter to `tkinter.font.nametofont()`.

- bpo-15303 [https://bugs.python.org/issue?@action=redirect&bpo=15303]: `tkinter` supports now widgets with boolean value False.
- bpo-42681 [https://bugs.python.org/issue?@action=redirect&bpo=42681]: Fixed range checks for color and pair numbers in `curses`.
- bpo-42685 [https://bugs.python.org/issue?@action=redirect&bpo=42685]: Improved placing of simple query windows in Tkinter (such as `tkinter.simpledialog.askinteger()`). They are now centered at the center of the parent window if it is specified and shown, otherwise at the center of the screen.
- bpo-9694 [https://bugs.python.org/issue?@action=redirect&bpo=9694]: Argparse help no longer uses the confusing phrase, “optional arguments”. It uses “options” instead.
- bpo-1635741 [https://bugs.python.org/issue?@action=redirect&bpo=1635741]: Port the `_thread` extension module to the multiphase initialization API (**PEP 489** [https://peps.python.org/pep-0489/]) and convert its static types to heap types.
- bpo-37961 [https://bugs.python.org/issue?@action=redirect&bpo=37961]: Fix crash in `tracemalloc.Traceback.__repr__()` (regressed in Python 3.9).
- bpo-42630 [https://bugs.python.org/issue?@action=redirect&bpo=42630]: `tkinter` functions and constructors which need a default root window raise now `RuntimeError` with descriptive message instead of

obscure **`AttributeError`** or **`NameError`** if it is not created yet or cannot be created automatically.

- bpo-42639 [https://bugs.python.org/issue?@action=redirect&bpo=42639]: **`atexit._run_exitfuncs()`** now logs callback exceptions using **`sys.unraisablehook`**, rather than logging them directly into **`sys.stderr`** and raise the last exception.

- bpo-42644 [https://bugs.python.org/issue?@action=redirect&bpo=42644]: `logging.disable` will now validate the types and value of its parameter. It also now accepts strings representing the levels (as does `logging.setLevel`) instead of only the numerical values.

- bpo-42639 [https://bugs.python.org/issue?@action=redirect&bpo=42639]: At Python exit, if a callback registered with **`atexit.register()`** fails, its exception is now logged. Previously, only some exceptions were logged, and the last exception was always silently ignored.

- bpo-36541 [https://bugs.python.org/issue?@action=redirect&bpo=36541]: Fixed lib2to3.pgen2 to be able to parse PEP-570 positional only argument syntax.

- bpo-42382 [https://bugs.python.org/issue?@action=redirect&bpo=42382]: In `importlib.metadata`: - `EntryPoint` objects now expose a `.dist` object referencing the `Distribution` when constructed from a `Distribution`. - Add support for package discovery under package normalization rules. - The object returned by `metadata()` now has a formally defined protocol called `PackageMetadata` with declared support for the `.get_all()` method. - Synced with importlib_metadata 3.3.

- bpo-41877 [https://bugs.python.org/issue?@action=redirect&bpo=41877]: A check is added against misspellings of autospect, auto_spec and set_spec being passed as arguments to patch, patch.object and create_autospec.

- bpo-39717 [https://bugs.python.org/issue?@action=redirect&bpo=39717]: [tarfile] update nested exception raising to use `from None` or `from e`

- bpo-41877 [https://bugs.python.org/issue?@action=redirect&bpo=41877]: AttributeError for suspected misspellings of assertions on mocks are now pointing out that the cause are misspelled assertions and also what to do if the misspelling is actually an intended attribute name. The unittest.mock document is also updated to reflect the current set of recognised misspellings.

- bpo-41559 [https://bugs.python.org/issue?@action=redirect&bpo=41559]: Implemented **PEP 612** [https://peps.python.org/pep-0612/]: added `ParamSpec` and `Concatenate` to **`typing`**. Patch by Ken Jin.

- bpo-42385 [https://bugs.python.org/issue?@action=redirect&bpo=42385]: StrEnum: fix _generate_next_value_ to return a str

- bpo-31904 [https://bugs.python.org/issue?@action=redirect&bpo=31904]: Define THREAD_STACK_SIZE for VxWorks.

- bpo-34750 [https://bugs.python.org/issue?@action=redirect&bpo=34750]: [Enum] `_EnumDict.update()` is now supported.

- bpo-42517 [https://bugs.python.org/issue?@action=redirect&bpo=42517]: Enum: private names do not become members / do not generate errors – they remain normal attributes

- bpo-42678 [https://bugs.python.org/issue?@action=redirect&bpo=42678]: `Enum`: call `__init_subclass__` after members have been added

- bpo-28964 [https://bugs.python.org/issue?@action=redirect&bpo=28964]: **`ast.literal_eval()`** adds line number information (if available) in error message for malformed nodes.

- bpo-42470 [https://bugs.python.org/issue?@action=redirect&bpo=42470]: **`random.sample()`** no longer warns on a sequence which is also a set.

- bpo-31904 [https://bugs.python.org/issue?@action=redirect&bpo=31904]: **`posixpath.expanduser()`** returns the input *path* unchanged if user home directory is None on VxWorks.

- bpo-42388 [https://bugs.python.org/issue?@action=redirect&bpo=42388]: Fix subprocess.check_output(…, input=None) behavior when text=True to be consistent with that of the documentation and universal_newlines=True.

- bpo-34463 [https://bugs.python.org/issue?@action=redirect&bpo=34463]: Fixed discrepancy between **`traceback`** and the interpreter in formatting of SyntaxError with lineno not set (**`traceback`** was changed to match interpreter).

- bpo-42393 [https://bugs.python.org/issue?@action=redirect&bpo=42393]: Raise **`OverflowError`** instead of silent truncation in `socket.ntohs()` and `socket.htons()`. Silent truncation was deprecated in Python 3.7. Patch by Erlend E. Aasland

- bpo-42222 [https://bugs.python.org/issue?@action=redirect&bpo=42222]: Harmonized `random.randrange()` argument handling to match `range()`.

  - The integer test and conversion in `randrange()` now uses `operator.index()`.
  - Non-integer arguments to `randrange()` are deprecated.
  - The `ValueError` is deprecated in favor of a `TypeError`.
  - It now runs a little faster than before.

  (Contributed by Raymond Hettinger and Serhiy Storchaka.)

- bpo-42163 [https://bugs.python.org/issue?@action=redirect&bpo=42163]: Restore compatibility for `uname_result` around deepcopy and _replace.

- bpo-42090 [https://bugs.python.org/issue?@action=redirect&bpo=42090]: `zipfile.Path.joinpath` now accepts arbitrary arguments, same as `pathlib.Path.joinpath`.

- bpo-1635741 [https://bugs.python.org/issue?@action=redirect&bpo=1635741]: Port the _csv module to the multi-phase initialization API (**PEP 489** [https://peps.python.org/pep-0489/]).

- bpo-42059 [https://bugs.python.org/issue?@action=redirect&bpo=42059]: `typing.TypedDict` types created using the alternative call-style syntax now correctly respect the `total` keyword argument when setting their `__required_keys__` and `__optional_keys__` class attributes.

- bpo-41960 [https://bugs.python.org/issue?@action=redirect&bpo=41960]: Add `globalns` and `localns` parameters to the `inspect.signature()` and `inspect.Signature.from_callable()`.
- bpo-41907 [https://bugs.python.org/issue?@action=redirect&bpo=41907]: fix `format()` behavior for `IntFlag`
- bpo-41891 [https://bugs.python.org/issue?@action=redirect&bpo=41891]: Ensure asyncio.wait_for waits for task completion
- bpo-24792 [https://bugs.python.org/issue?@action=redirect&bpo=24792]: Fixed bug where **`zipimporter`** sometimes reports an incorrect cause of import errors.
- bpo-31904 [https://bugs.python.org/issue?@action=redirect&bpo=31904]: Fix site and sysconfig modules for VxWorks RTOS which has no home directories.
- bpo-41462 [https://bugs.python.org/issue?@action=redirect&bpo=41462]: Add `os.set_blocking()` support for VxWorks RTOS.
- bpo-40219 [https://bugs.python.org/issue?@action=redirect&bpo=40219]: Lowered **`tkinter.ttk.LabeledScale`** dummy widget to prevent hiding part of the content label.
- bpo-37193 [https://bugs.python.org/issue?@action=redirect&bpo=37193]: Fixed memory leak in `socketserver.ThreadingMixIn` introduced in Python 3.7.
- bpo-39068 [https://bugs.python.org/issue?@action=redirect&bpo=39068]: Fix initialization race condition in **`a85encode()`** and **`b85encode()`** in `base64`. Patch by Brandon Stansbury.

Documentation

- bpo-17140 [https://bugs.python.org/issue?@action=redirect&bpo=17140]: Add documentation for the `multiprocessing.pool.ThreadPool` class.
- bpo-34398 [https://bugs.python.org/issue?@action=redirect&bpo=34398]: Prominently feature listings from the glossary in documentation search results. Patch by Ammar Askar.

Tests

- bpo-42794 [https://bugs.python.org/issue?@action=redirect&bpo=42794]: Update test_nntplib to use official group name of news.aioe.org for testing. Patch by Donghee Na.
- bpo-31904 [https://bugs.python.org/issue?@action=redirect&bpo=31904]: Skip some asyncio tests on VxWorks.
- bpo-42641 [https://bugs.python.org/issue?@action=redirect&bpo=42641]: Enhance `test_select.test_select()`: it now takes 500 milliseconds rather than 10 seconds. Use Python rather than a shell to make the test more portable.
- bpo-31904 [https://bugs.python.org/issue?@action=redirect&bpo=31904]: Skip some tests in _test_all_chown_common() on VxWorks.
- bpo-42199 [https://bugs.python.org/issue?@action=redirect&bpo=42199]: Fix bytecode helper assertNotInBytecode.
- bpo-41443 [https://bugs.python.org/issue?@action=redirect&bpo=41443]: Add more attribute checking in test_posix.py
- bpo-31904 [https://bugs.python.org/issue?@action=redirect&bpo=31904]: Disable os.popen and impacted

tests on VxWorks
- bpo-41439 [https://bugs.python.org/issue?@action=redirect&bpo=41439]: Port test_ssl and test_uuid to VxWorks RTOS.

Build

- bpo-42692 [https://bugs.python.org/issue?@action=redirect&bpo=42692]: Fix __builtin_available check on older compilers. Patch by Joshua Root.
- bpo-27640 [https://bugs.python.org/issue?@action=redirect&bpo=27640]: Added `--disable-test-modules` option to the `configure` script: don't build nor install test modules. Patch by Xavier de Gaye, Thomas Petazzoni and Peixing Xin.
- bpo-42604 [https://bugs.python.org/issue?@action=redirect&bpo=42604]: Now all platforms use a value for the "EXT_SUFFIX" build variable derived from SOABI (for instance in freeBSD, "EXT_SUFFIX" is now ".cpython-310d.so" instead of ".so"). Previously only Linux, Mac and VxWorks were using a value for "EXT_SUFFIX" that included "SOABI".
- bpo-42598 [https://bugs.python.org/issue?@action=redirect&bpo=42598]: Fix implicit function declarations in configure which could have resulted in incorrect configuration checks. Patch contributed by Joshua Root.
- bpo-31904 [https://bugs.python.org/issue?@action=redirect&bpo=31904]: Enable libpython3.so for VxWorks.
- bpo-29076 [https://bugs.python.org/issue?@action=redirect&bpo=29076]: Add fish shell support to macOS installer.

macOS

- bpo-42361 [https://bugs.python.org/issue?@action=redirect&bpo=42361]: Update macOS installer build to use Tcl/Tk 8.6.11 (rc2, expected to be final release).
- bpo-41837 [https://bugs.python.org/issue?@action=redirect&bpo=41837]: Update macOS installer build to use OpenSSL 1.1.1i.
- bpo-42584 [https://bugs.python.org/issue?@action=redirect&bpo=42584]: Update macOS installer to use SQLite 3.34.0.

Tools/Demos

- bpo-42726 [https://bugs.python.org/issue?@action=redirect&bpo=42726]: Fixed Python 3 compatibility issue with gdb/libpython.py handling of attribute dictionaries.
- bpo-42613 [https://bugs.python.org/issue?@action=redirect&bpo=42613]: Fix `freeze.py` tool to use the prope config and library directories. Patch by Victor Stinner.

C API

- bpo-42591 [https://bugs.python.org/issue?@action=redirect&bpo=42591]: Export the **`Py_FrozenMain()`** function: fix a Python 3.9.0 regression. Python 3.9 uses `-fvisibility=hidden` and the function was not exported explicitly and so not exported.
- bpo-32381 [https://bugs.python.org/issue?@action=redirect&bpo=32381]: Remove the private **`_Py_fopen()`** function which is no longer needed. Use **`_Py_wfopen()`** or **`_Py_fopen_obj()`** instead. Patch by Victor Stinner.
- bpo-1635741 [https://bugs.python.org/issue?@action=redirect&bpo=1635741]: Port **`resource`** extension module to module state

- bpo-42111 [https://bugs.python.org/issue?@action=redirect&bpo=42111]: Update the `xxlimited` module to be a better example of how to use the limited C API.
- bpo-40052 [https://bugs.python.org/issue?@action=redirect&bpo=40052]: Fix an alignment build warning/error in function `PyVectorcall_Function()`. Patch by Andreas Schneider, Antoine Pitrou and Petr Viktorin.

Python 3.10.0 alpha 3

Release date: 2020-12-07

Security

- bpo-40791 [https://bugs.python.org/issue?@action=redirect&bpo=40791]: Add `volatile` to the accumulator variable in `hmac.compare_digest`, making constant-time-defeating optimizations less likely.

Core and Builtins

- bpo-42576 [https://bugs.python.org/issue?@action=redirect&bpo=42576]: `types.GenericAlias` will now raise a `TypeError` when attempting to initialize with a keyword argument. Previously, this would cause the interpreter to crash if the interpreter was compiled with debug symbols. This does not affect interpreters compiled for release. Patch by Ken Jin.

- bpo-42536 [https://bugs.python.org/issue?@action=redirect&bpo=42536]: Several built-in and standard library types now ensure that their internal result tuples are always tracked by the garbage collector:

    - **`collections.OrderedDict.items()`**

  - `dict.items()`
  - `enumerate()`
  - `functools.reduce()`
  - `itertools.combinations()`
  - `itertools.combinations_with_replacement()`
  - `itertools.permutations()`
  - `itertools.product()`
  - `itertools.zip_longest()`
  - `zip()`

  Previously, they could have become untracked by a prior garbage collection. Patch by Brandt Bucher.

- bpo-42500 [https://bugs.python.org/issue?@action=redirect&bpo=42500]: Improve handling of exceptions near recursion limit. Converts a number of Fatal Errors in RecursionErrors.

- bpo-42246 [https://bugs.python.org/issue?@action=redirect&bpo=42246]: PEP 626: After a return, the f_lineno attribute of a frame is always the last line executed.

- bpo-42435 [https://bugs.python.org/issue?@action=redirect&bpo=42435]: Speed up comparison of bytes objects with non-bytes objects when option `-b` is specified. Speed up comparison of bytarray objects with non-buffer object.

Library

- bpo-1635741 [https://bugs.python.org/issue?@action=redirect&bpo=1635741]: Port the `_warnings` extension module to the multi-phase initialization API (**PEP 489** [https://peps.python.org/pep-0489/]). Patch by Victor Stinner.

Core and Builtins

- bpo-41686 [https://bugs.python.org/issue?@action=redirect&bpo=41686]: On Windows, the `SIGINT` event, `_PyOS_SigintEvent()`, is now created even if Python is configured to not install signal handlers (if `PyConfig.install_signal_handlers` equals to 0, or `Py_InitializeEx(0)`).
- bpo-42381 [https://bugs.python.org/issue?@action=redirect&bpo=42381]: Allow assignment expressions in set literals and set comprehensions as per PEP 572. Patch by Pablo Galindo.
- bpo-42202 [https://bugs.python.org/issue?@action=redirect&bpo=42202]: Change function parameters annotations internal representation to tuple of strings. Patch provided by Yurii Karabas.
- bpo-42374 [https://bugs.python.org/issue?@action=redirect&bpo=42374]: Fix a regression introduced by the new parser, where an unparenthesized walrus operator was not allowed within generator expressions.
- bpo-42316 [https://bugs.python.org/issue?@action=redirect&bpo=42316]: Allow an unparenthesized walrus in subscript indexes.
- bpo-42349 [https://bugs.python.org/issue?@action=redirect&bpo=42349]: Make sure that the compiler front-end produces a well-formed control flow graph. Be more aggressive in the compiler back-end, as it is now safe to do so.
- bpo-42296 [https://bugs.python.org/issue?@action=redirect&bpo=42296]: On Windows, fix a regression in signal handling which prevented to interrupt a program using CTRL+C. The signal handler can be run in a thread different than the Python thread, in which case the test deciding if the thread can handle signals is wrong.

- bpo-42332 [https://bugs.python.org/issue?@action=redirect&bpo=42332]: `types.GenericAlias` objects can now be the targets of weakrefs.
- bpo-42282 [https://bugs.python.org/issue?@action=redirect&bpo=42282]: Optimise constant subexpressions that appear as part of named expressions (previously the AST optimiser did not descend into named expressions). Patch by Nick Coghlan.
- bpo-42266 [https://bugs.python.org/issue?@action=redirect&bpo=42266]: Fixed a bug with the LOAD_ATTR opcode cache that was not respecting monkey-patching a class-level attribute to make it a descriptor. Patch by Pablo Galindo.

Library

- bpo-40077 [https://bugs.python.org/issue?@action=redirect&bpo=40077]: Convert `queue` to use heap types.

Core and Builtins

- bpo-42246 [https://bugs.python.org/issue?@action=redirect&bpo=42246]: Improved accuracy of line tracing events and f_lineno attribute of Frame objects. See PEP 626 for details.

Library

- bpo-40077 [https://bugs.python.org/issue?@action=redirect&bpo=40077]: Convert `mmap` to use heap types.

Core and Builtins

- bpo-42233 [https://bugs.python.org/issue?@action=redirect&bpo=42233]: Allow `GenericAlias` objects to use union type expressions. This allows expressions like `list[int] | dict[float, str]` where previously a `TypeError` would have been thrown. This also fixes union type expressions not de-duplicating `GenericAlias` objects. (Contributed by Ken Jin in bpo-42233 [https://bugs.python.org/issue?@action=redirect&bpo=42233].)
- bpo-26131 [https://bugs.python.org/issue?@action=redirect&bpo=26131]: The import system triggers a `ImportWarning` when it falls back to using `load_module()`.

Library

- bpo-5054 [https://bugs.python.org/issue?@action=redirect&bpo=5054]: CGIHTTPRequestHandler.run_cgi() HTTP_ACCEPT improperly parsed. Replace the special purpose getallmatchingheaders with generic get_all method and add relevant tests.

  Original Patch by Martin Panter. Modified by Senthil Kumaran.

- bpo-42562 [https://bugs.python.org/issue?@action=redirect&bpo=42562]: Fix issue when dis failed to parse function that has no line numbers. Patch provided by Yurii Karabas.

- bpo-17735 [https://bugs.python.org/issue?@action=redirect&bpo=17735]: **`inspect.findsource()`** now raises **`OSError`** instead of **`IndexError`** when **`co_lineno`** of a code object is greater than the file length. This can happen, for example, when a file is edited after it was imported. PR by Irit Katriel.

- bpo-42116 [https://bugs.python.org/issue?@action=redirect&bpo=42116]: Fix handling of trailing comments by `inspect.getsource()`.

- bpo-42532 [https://bugs.python.org/issue?@action=redirect&bpo=42532]: Remove unexpected call of `__bool__` when passing a `spec_arg` argument to a Mock.

- bpo-38200 [https://bugs.python.org/issue?@action=redirect&bpo=38200]: Added itertools.pairwise()

- bpo-41818 [https://bugs.python.org/issue?@action=redirect&bpo=41818]: Fix test_master_read() so that it succeeds on all platforms that either raise OSError or return b"" upon reading from master.

- bpo-42487 [https://bugs.python.org/issue?@action=redirect&bpo=42487]: ChainMap.__iter__ no longer calls __getitem__ on underlying maps

- bpo-42482 [https://bugs.python.org/issue?@action=redirect&bpo=42482]: `TracebackException` no longer holds a reference to the exception's traceback object. Consequently, instances of TracebackException for equivalent but non-equal exceptions now compare as equal.

- bpo-41818 [https://bugs.python.org/issue?@action=redirect&bpo=41818]: Make test_openpty() avoid unexpected success due to number of rows and/or number of columns being == 0.

- bpo-42392 [https://bugs.python.org/issue?@action=redirect&bpo=42392]: Remove loop parameter from `asyncio.subprocess` and `asyncio.tasks` functions. Patch provided by Yurii Karabas.

- bpo-42392 [https://bugs.python.org/issue?@action=redirect&bpo=42392]: Remove loop parameter from `asyncio.open_connection` and `asyncio.start_server` functions. Patch provided by Yurii Karabas.
- bpo-28468 [https://bugs.python.org/issue?@action=redirect&bpo=28468]: Add **`platform.freedesktop_os_release()`** function to parse freedesktop.org `os-release` files.
- bpo-42299 [https://bugs.python.org/issue?@action=redirect&bpo=42299]: Removed the `formatter` module, which was deprecated in Python 3.4. It is somewhat obsolete, little used, and not tested. It was originally scheduled to be removed in Python 3.6, but such removals were delayed until after Python 2.7 EOL. Existing users should copy whatever classes they use into their code. Patch by Donghee Na and and Terry J. Reedy.
- bpo-26131 [https://bugs.python.org/issue?@action=redirect&bpo=26131]: Deprecate zipimport.zipimporter.load_module() in favour of exec_module().
- bpo-41818 [https://bugs.python.org/issue?@action=redirect&bpo=41818]: Updated tests for the pty library. test_basic() has been changed to test_openpty(); this additionally checks if slave termios and slave winsize are being set properly by pty.openpty(). In order to add support for FreeBSD, NetBSD, OpenBSD, and Darwin, this also adds test_master_read(), which demonstrates that pty.spawn() should not depend on an OSError to exit from its copy loop.

- bpo-42392 [https://bugs.python.org/issue?@action=redirect&bpo=42392]: Remove loop parameter from `__init__` in all `asyncio.locks` and `asyncio.Queue` classes. Patch provided by Yurii Karabas.

- bpo-15450 [https://bugs.python.org/issue?@action=redirect&bpo=15450]: Make `filecmp.dircmp` respect subclassing. Now the `filecmp.dircmp.subdirs` behaves as expected when subclassing dircmp.

- bpo-42413 [https://bugs.python.org/issue?@action=redirect&bpo=42413]: The exception `socket.timeout` is now an alias of `TimeoutError`.

- bpo-31904 [https://bugs.python.org/issue?@action=redirect&bpo=31904]: Support signal module on VxWorks.

- bpo-42406 [https://bugs.python.org/issue?@action=redirect&bpo=42406]: We fixed an issue in `pickle.whichmodule` in which importing `multiprocessing` could change the how pickle identifies which module an object belongs to, potentially breaking the unpickling of those objects.

- bpo-42403 [https://bugs.python.org/issue?@action=redirect&bpo=42403]: Simplify the `importlib` external bootstrap code: `importlib._bootstrap_external` now uses regular imports to import builtin modules. When it is imported, the builtin `__import__()` function is already fully working and so can be used to import builtin modules like `sys`. Patch by Victor Stinner.

- bpo-1635741 [https://bugs.python.org/issue?@action=redirect&bpo=1635741]: Convert _sre module types to heap types (PEP 384). Patch by Erlend E. Aasland.

- bpo-42375 [https://bugs.python.org/issue?@action=redirect&bpo=42375]: subprocess module update for DragonFlyBSD support.

- bpo-41713 [https://bugs.python.org/issue?@action=redirect&bpo=41713]: Port the `_signal` extension module to the multi-phase initialization API (**PEP 489** [https://peps.python.org/pep-0489/]). Patch by Victor Stinner and Mohamed Koubaa.

- bpo-37205 [https://bugs.python.org/issue?@action=redirect&bpo=37205]: `time.time()`, `time.perf_counter()` and `time.monotonic()` functions can no longer fail with a Python fatal error, instead raise a regular Python exception on failure.

- bpo-42328 [https://bugs.python.org/issue?@action=redirect&bpo=42328]: Fixed `tkinter.ttk.Style.map()`. The function accepts now the representation of the default state as empty sequence (as returned by `Style.map()`). The structure of the result is now the same on all platform and does not depend on the value of `wantobjects`.

- bpo-42345 [https://bugs.python.org/issue?@action=redirect&bpo=42345]: Fix various issues with `typing.Literal` parameter handling (flatten, deduplicate, use type to cache key). Patch provided by Yurii Karabas.

- bpo-37205 [https://bugs.python.org/issue?@action=redirect&bpo=37205]: `time.perf_counter()` on Windows and `time.monotonic()` on macOS are now system-wide. Previously, they used an offset computed at startup to reduce the precision loss caused by the float type. Use `time.perf_counter_ns()` and `time.monotonic_ns()` added in Python 3.7 to avoid this precision loss.

- bpo-42318 [https://bugs.python.org/issue?@action=redirect&bpo=42318]: Fixed support of non-BMP characters in **`tkinter`** on macOS.

- bpo-42350 [https://bugs.python.org/issue?@action=redirect&bpo=42350]: Fix the **`threading.Thread`** class at fork: do nothing if the thread is already stopped (ex: fork called at Python exit). Previously, an error was logged in the child process.

- bpo-42333 [https://bugs.python.org/issue?@action=redirect&bpo=42333]: Port _ssl extension module to heap types.

- bpo-42014 [https://bugs.python.org/issue?@action=redirect&bpo=42014]: The `onerror` callback from `shutil.rmtree` now receives correct function when `os.open` fails.

- bpo-42237 [https://bugs.python.org/issue?@action=redirect&bpo=42237]: Fix `os.sendfile()` on illumos.

- bpo-42308 [https://bugs.python.org/issue?@action=redirect&bpo=42308]: Add **`threading.__excepthook__`** to allow retrieving the original value of **`threading.excepthook()`** in case it is set to a broken or a different value. Patch by Mario Corchero.

- bpo-42131 [https://bugs.python.org/issue?@action=redirect&bpo=42131]: Implement PEP 451/spec methods on zipimport.zipimporter: find_spec(), create_module(), and exec_module().

  This also allows for the documented deprecation of find_loader(), find_module(), and load_module().

- bpo-41877 [https://bugs.python.org/issue?@action=redirect&bpo=41877]: Mock objects which are not unsafe will now raise an AttributeError if an attribute with the prefix asert, aseert, or assrt is accessed, in addition to this already happening for the prefixes assert or assret.

- bpo-42264 [https://bugs.python.org/issue?@action=redirect&bpo=42264]: `sqlite3.OptimizedUnicode` has been undocumented and obsolete since Python 3.3, when it was made an alias to **`str`**. It is now deprecated, scheduled for removal in Python 3.12.

- bpo-42251 [https://bugs.python.org/issue?@action=redirect&bpo=42251]: Added **`threading.gettrace()`** and **`threading.getprofile()`** to retrieve the functions set by **`threading.settrace()`** and **`threading.setprofile()`** respectively. Patch by Mario Corchero.

- bpo-42249 [https://bugs.python.org/issue?@action=redirect&bpo=42249]: Fixed writing binary Plist files larger than 4 GiB.

- bpo-42236 [https://bugs.python.org/issue?@action=redirect&bpo=42236]: On Unix, the **`os.device_encoding()`** function now returns `'UTF-8'` rather than the device encoding if the Python UTF-8 Mode is enabled.

- bpo-41754 [https://bugs.python.org/issue?@action=redirect&bpo=41754]: webbrowser: Ignore *NotADirectoryError* when calling `xdg-settings`.

- bpo-42183 [https://bugs.python.org/issue?@action=redirect&bpo=42183]: Fix a stack overflow error for asyncio Task or Future repr().

The overflow occurs under some circumstances when a Task or Future recursively returns itself.

- bpo-42140 [https://bugs.python.org/issue?@action=redirect&bpo=42140]: Improve asyncio.wait function to create the futures set just one time.

- bpo-42133 [https://bugs.python.org/issue?@action=redirect&bpo=42133]: Update various modules in the stdlib to fall back on `__spec__.loader` when `__loader__` isn’t defined on a module.

- bpo-26131 [https://bugs.python.org/issue?@action=redirect&bpo=26131]: The `load_module()` methods found in `importlib` now trigger a `DeprecationWarning`.

- bpo-39825 [https://bugs.python.org/issue?@action=redirect&bpo=39825]: Windows: Change `sysconfig.get_config_var('EXT_SUFFIX')` to the expected full `platform_tag.extension` format. Previously it was hard-coded to `.pyd`, now it is compatible with `distutils.sysconfig` and will result in something like `.cp38-win_amd64.pyd`. This brings windows into conformance with the other platforms.

- bpo-26389 [https://bugs.python.org/issue?@action=redirect&bpo=26389]: The **`traceback.format_exception()`**, **`traceback.format_exception_only()`**, and **`traceback.print_exception()`** functions can now take an exception object as a positional-only argument.

- bpo-41889 [https://bugs.python.org/issue?@action=redirect&bpo=41889]: Enum: fix regression involving inheriting a multiply inherited enum

- bpo-41861 [https://bugs.python.org/issue?@action=redirect&bpo=41861]: Convert **`sqlite3`** to use heap types (PEP 384). Patch by Erlend E. Aasland.

- bpo-40624 [https://bugs.python.org/issue?@action=redirect&bpo=40624]: Added support for the XPath `!=` operator in xml.etree

- bpo-28850 [https://bugs.python.org/issue?@action=redirect&bpo=28850]: Fix **`pprint.PrettyPrinter.format()`** overrides being ignored for contents of small containers. The **`pprint._safe_repr()`** function was removed.

- bpo-41625 [https://bugs.python.org/issue?@action=redirect&bpo=41625]: Expose the **`splice()`** as **`os.splice()`** in the **`os`** module. Patch by Pablo Galindo

- bpo-34215 [https://bugs.python.org/issue?@action=redirect&bpo=34215]: Clarify the error message for **`asyncio.IncompleteReadError`** when `expected` is `None`.

- bpo-41543 [https://bugs.python.org/issue?@action=redirect&bpo=41543]: Add async context manager support for contextlib.nullcontext.

- bpo-21041 [https://bugs.python.org/issue?@action=redirect&bpo=21041]: **`pathlib.PurePath.parents`** now supports negative indexing. Patch contributed by Yaroslav Pankovych.

- bpo-41332 [https://bugs.python.org/issue?@action=redirect&bpo=41332]: Added missing connect_accepted_socket() method to `asyncio.AbstractEventLoop`.

- bpo-12800 [https://bugs.python.org/issue?@action=redirect&bpo=12800]: Extracting a symlink from a tarball should succeed and overwrite the symlink if it already exists. The fix is to remove the existing file or symlink before extraction. Based on patch by Chris AtLee, Jeffrey Kintscher, and Senthil Kumaran.

- bpo-40968 [https://bugs.python.org/issue?@action=redirect&bpo=40968]: **`urllib.request`** and **`http.client`** now send `http/1.1` ALPN extension during TLS handshake when no custom context is supplied.

- bpo-41001 [https://bugs.python.org/issue?@action=redirect&bpo=41001]: Add **`os.eventfd()`** to provide a low level interface for Linux’s event notification file descriptor.

- bpo-40816 [https://bugs.python.org/issue?@action=redirect&bpo=40816]: Add AsyncContextDecorator to contextlib to support async context manager as a decorator.

- bpo-40550 [https://bugs.python.org/issue?@action=redirect&bpo=40550]: Fix time-of-check/time-of-action issue in subprocess.Popen.send_signal.

- bpo-39411 [https://bugs.python.org/issue?@action=redirect&bpo=39411]: Add an `is_async` identifier to **`pyclbr`**’s `Function` objects. Patch by Batuhan Taskaya

- bpo-35498 [https://bugs.python.org/issue?@action=redirect&bpo=35498]: Add slice support to **`pathlib.PurePath.parents`**.

Documentation

- bpo-42238 [https://bugs.python.org/issue?@action=redirect&bpo=42238]: Tentative to deprecate `make suspicious` by first removing it from the CI and documentation builds, but keeping it around for manual uses.

- bpo-42153 [https://bugs.python.org/issue?@action=redirect&bpo=42153]: Fix the URL for the IMAP protocol documents.
- bpo-41028 [https://bugs.python.org/issue?@action=redirect&bpo=41028]: Language and version switchers, previously maintained in every cpython branches, are now handled by docsbuild-script.

Tests

- bpo-41473 [https://bugs.python.org/issue?@action=redirect&bpo=41473]: Re-enable test_gdb on gdb 9.2 and newer: https://bugzilla.redhat.com/show_bug.cgi?id=1866884 bug is fixed in gdb 10.1.
- bpo-42553 [https://bugs.python.org/issue?@action=redirect&bpo=42553]: Fix `test_asyncio.test_call_later()` race condition: don't measure asyncio performance in the `call_later()` unit test. The test failed randomly on the CI.
- bpo-31904 [https://bugs.python.org/issue?@action=redirect&bpo=31904]: Fix test_netrc on VxWorks: create temporary directories using temp_cwd().
- bpo-31904 [https://bugs.python.org/issue?@action=redirect&bpo=31904]: skip test_getaddrinfo_ipv6_scopeid_symbolic and test_getnameinfo_ipv6_scopeid_symbolic on VxWorks
- bpo-31904 [https://bugs.python.org/issue?@action=redirect&bpo=31904]: skip test_test of test_mailcap on VxWorks
- bpo-31904 [https://bugs.python.org/issue?@action=redirect&bpo=31904]: add shell requirement for test_pipes
- bpo-31904 [https://bugs.python.org/issue?@action=redirect&bpo=31904]: skip some tests related to fifo on VxWorks

- bpo-31904 [https://bugs.python.org/issue?@action=redirect&bpo=31904]: Fix test_doctest.py failures for VxWorks.
- bpo-40754 [https://bugs.python.org/issue?@action=redirect&bpo=40754]: Include `_testinternalcapi` module in Windows installer for test suite
- bpo-41561 [https://bugs.python.org/issue?@action=redirect&bpo=41561]: test_ssl: skip test_min_max_version_mismatch when TLS 1.0 is not available
- bpo-31904 [https://bugs.python.org/issue?@action=redirect&bpo=31904]: Fix os module failures for VxWorks RTOS.
- bpo-31904 [https://bugs.python.org/issue?@action=redirect&bpo=31904]: Fix fifo test cases for VxWorks RTOS.

Build

- bpo-31904 [https://bugs.python.org/issue?@action=redirect&bpo=31904]: remove libnet dependency from detect_socket() for VxWorks
- bpo-42398 [https://bugs.python.org/issue?@action=redirect&bpo=42398]: Fix a race condition in “make regen-all” when make -jN option is used to run jobs in parallel. The clinic.py script now only use atomic write to write files. Moveover, generated files are now left unchanged if the content does not change, to not change the file modification time.
- bpo-41617 [https://bugs.python.org/issue?@action=redirect&bpo=41617]: Fix building `pycore_bitutils.h` internal header on old clang version without `__builtin_bswap16()` (ex: Xcode 4.6.3 on Mac OS X 10.7). Patch by Joshua Root and Victor Stinner.
- bpo-38823 [https://bugs.python.org/issue?@action=redirect&bpo=38823]: It is no longer possible to build

the `_ctypes` extension module without **`wchar_t`** type: remove `CTYPES_UNICODE` macro. Anyway, the **`wchar_t`** type is required to build Python. Patch by Victor Stinner.

- bpo-42087 [https://bugs.python.org/issue?@action=redirect&bpo=42087]: Support was removed for AIX 5.3 and below. See bpo-40680 [https://bugs.python.org/issue?@action=redirect&bpo=40680].
- bpo-40998 [https://bugs.python.org/issue?@action=redirect&bpo=40998]: Addressed three compiler warnings found by undefined behavior sanitizer (ubsan).

Windows

- bpo-42120 [https://bugs.python.org/issue?@action=redirect&bpo=42120]: Remove macro definition of `copysign` (to `_copysign`) in headers.
- bpo-38506 [https://bugs.python.org/issue?@action=redirect&bpo=38506]: The Windows launcher now properly handles Python 3.10 when listing installed Python versions.

macOS

- bpo-42504 [https://bugs.python.org/issue?@action=redirect&bpo=42504]: Fix build on macOS Big Sur when MACOSX_DEPLOYMENT_TARGET=11

- bpo-41116 [https://bugs.python.org/issue?@action=redirect&bpo=41116]: Ensure distutils.unixxcompiler.find_library_file can find system provided libraries on macOS 11.

- bpo-41100 [https://bugs.python.org/issue?@action=redirect&bpo=41100]: Add support for macOS 11 and Apple Silicon systems.

It is now possible to build “Universal 2” binaries using “–enable-universalsdk –with-universal-archs=universal2”.

Binaries build on later macOS versions can be deployed back to older versions (tested up to macOS 10.9), when using the correct deployment target. This is tested using Xcode 11 and later.

- bpo-42232 [https://bugs.python.org/issue?@action=redirect&bpo=42232]: Added Darwin specific madvise options to mmap module.

- bpo-38443 [https://bugs.python.org/issue?@action=redirect&bpo=38443]: The `--enable-universalsdk` and `--with-universal-archs` options for the configure script now check that the specified architectures can be used.

IDLE

- bpo-42508 [https://bugs.python.org/issue?@action=redirect&bpo=42508]: Keep IDLE running on macOS. Remove obsolete workaround that prevented running files with shortcuts when using new universal2 installers built on macOS 11.
- bpo-42426 [https://bugs.python.org/issue?@action=redirect&bpo=42426]: Fix reporting offset of the RE error in searchengine.
- bpo-42415 [https://bugs.python.org/issue?@action=redirect&bpo=42415]: Get docstrings for IDLE calltips more often by using inspect.getdoc.

Tools/Demos

- bpo-42212 [https://bugs.python.org/issue?@action=redirect&bpo=42212]: The smelly.py script now also checks the Python dynamic library and extension modules, not only the Python static library. Make also the script more verbose: explain what it does.

- bpo-36310 [https://bugs.python.org/issue?@action=redirect&bpo=36310]: Allow `Tools/i18n/pygettext.py` to detect calls to `gettext` in f-strings.

C API

- bpo-42423 [https://bugs.python.org/issue?@action=redirect&bpo=42423]: The **`PyType_FromSpecWithBases()`** and **`PyType_FromModuleAndSpec()`** functions now accept a single class as the *bases* argument.
- bpo-1635741 [https://bugs.python.org/issue?@action=redirect&bpo=1635741]: Port **`select`** extension module to multiphase initialization (**PEP 489** [https://peps.python.org/pep-0489/]).
- bpo-1635741 [https://bugs.python.org/issue?@action=redirect&bpo=1635741]: Port _posixsubprocess extension module to multiphase initialization (**PEP 489** [https://peps.python.org/pep-0489/]).
- bpo-1635741 [https://bugs.python.org/issue?@action=redirect&bpo=1635741]: Port _posixshmem extension module to multiphase initialization (**PEP 489** [https://peps.python.org/pep-0489/])
- bpo-1635741 [https://bugs.python.org/issue?@action=redirect&bpo=1635741]: Port _struct extension module to multiphase initialization (**PEP 489** [https://peps.python.org/pep-0489/])
- bpo-1635741 [https://bugs.python.org/issue?@action=redirect&bpo=1635741]: Port **`spwd`** extension module to multiphase initialization (**PEP 489** [https://peps.python.org/pep-0489/])
- bpo-1635741 [https://bugs.python.org/issue?@action=redirect&bpo=1635741]: Port **`gc`** extension module to multiphase initialization (**PEP 489** [https://peps.python.org/pep-0489/])

- bpo-1635741 [https://bugs.python.org/issue?@action=redirect&bpo=1635741]: Port _queue extension module to multiphase initialization (**PEP 489** [https://peps.python.org/pep-0489/])
- bpo-39573 [https://bugs.python.org/issue?@action=redirect&bpo=39573]: Convert **`Py_TYPE()`** and **`Py_SIZE()`** back to macros to allow using them as an l-value. Many third party C extension modules rely on the ability of using Py_TYPE() and Py_SIZE() to set an object type and size: `Py_TYPE(obj) = type;` and `Py_SIZE(obj) = size;`.
- bpo-1635741 [https://bugs.python.org/issue?@action=redirect&bpo=1635741]: Port **`symtable`** extension module to multiphase initialization (**PEP 489** [https://peps.python.org/pep-0489/])
- bpo-1635741 [https://bugs.python.org/issue?@action=redirect&bpo=1635741]: Port **`grp`** and **`pwd`** extension modules to multiphase initialization (**PEP 489** [https://peps.python.org/pep-0489/])
- bpo-1635741 [https://bugs.python.org/issue?@action=redirect&bpo=1635741]: Port _random extension module to multiphase initialization (**PEP 489** [https://peps.python.org/pep-0489/])
- bpo-1635741 [https://bugs.python.org/issue?@action=redirect&bpo=1635741]: Port _hashlib extension module to multiphase initialization (**PEP 489** [https://peps.python.org/pep-0489/])
- bpo-41713 [https://bugs.python.org/issue?@action=redirect&bpo=41713]: Removed the undocumented `PyOS_InitInterrupts()` function. Initializing Python already implicitly installs signal handlers: see **`PyConfig.install_signal_handlers`**. Patch by Victor Stinner.
- bpo-40170 [https://bugs.python.org/issue?@action=redirect&bpo=40170]: The `Py_TRASHCAN_BEGIN` macro no longer accesses PyTypeObject attributes, but now can get the condition by calling the new private

`_PyTrash_cond()` function which hides implementation details.

- bpo-42260 [https://bugs.python.org/issue?@action=redirect&bpo=42260]: **`Py_GetPath()`**, **`Py_GetPrefix()`**, **`Py_GetExecPrefix()`**, **`Py_GetProgramFullPath()`**, **`Py_GetPythonHome()`** and **`Py_GetProgramName()`** functions now return `NULL` if called before **`Py_Initialize()`** (before Python is initialized). Use the new Python Initialization Configuration API to get the Python Path Configuration.. Patch by Victor Stinner.
- bpo-42260 [https://bugs.python.org/issue?@action=redirect&bpo=42260]: The **`PyConfig_Read()`** function now only parses **`PyConfig.argv`** arguments once: **`PyConfig.parse_argv`** is set to `2` after arguments are parsed. Since Python arguments are strippped from **`PyConfig.argv`**, parsing arguments twice would parse the application options as Python options.
- bpo-42262 [https://bugs.python.org/issue?@action=redirect&bpo=42262]: Added **`Py_NewRef()`** and **`Py_XNewRef()`** functions to increment the reference count of an object and return the object. Patch by Victor Stinner.
- bpo-42260 [https://bugs.python.org/issue?@action=redirect&bpo=42260]: When **`Py_Initialize()`** is called twice, the second call now updates more **`sys`** attributes for the configuration, rather than only **`sys.argv`**. Patch by Victor Stinner.
- bpo-41832 [https://bugs.python.org/issue?@action=redirect&bpo=41832]: The **`PyType_FromModuleAndSpec()`** function now accepts NULL `tp_doc` slot.
- bpo-1635741 [https://bugs.python.org/issue?@action=redirect&bpo=1635741]: Added **`PyModule_AddObjectRef()`** function: similar to **`PyModule_AddObject()`** but don’t steal a reference to the value on success. Patch by Victor Stinner.

- bpo-42171 [https://bugs.python.org/issue?@action=redirect&bpo=42171]: The `METH_FASTCALL` calling convention is added to the limited API. The functions `PyModule_AddType()`, `PyType_FromModuleAndSpec()`, `PyType_GetModule()` and `PyType_GetModuleState()` are added to the limited API on Windows.
- bpo-42085 [https://bugs.python.org/issue?@action=redirect&bpo=42085]: Add dedicated entry to PyAsyncMethods for sending values
- bpo-41073 [https://bugs.python.org/issue?@action=redirect&bpo=41073]: `PyType_GetSlot()` can now accept static types.
- bpo-30459 [https://bugs.python.org/issue?@action=redirect&bpo=30459]: `PyList_SET_ITEM()`, `PyTuple_SET_ITEM()` and `PyCell_SET()` macros can no longer be used as l-value or r-value. For example, `x = PyList_SET_ITEM(a, b, c)` and `PyList_SET_ITEM(a, b, c) = x` now fail with a compiler error. It prevents bugs like `if (PyList_SET_ITEM (a, b, c) < 0) ...` test. Patch by Zackery Spytz and Victor Stinner.

Python 3.10.0 alpha 2

Release date: 2020-11-03

Security

- bpo-42103 [https://bugs.python.org/issue?@action=redirect&bpo=42103]: Prevented potential DoS attack via CPU and RAM exhaustion when processing malformed Apple Property List files in binary format.
- bpo-42051 [https://bugs.python.org/issue?@action=redirect&bpo=42051]: The `plistlib` module no longer accepts entity declarations in XML plist files to avoid

XML vulnerabilities. This should not affect users as entity declarations are not used in regular plist files.

Core and Builtins

- bpo-42236 [https://bugs.python.org/issue?@action=redirect&bpo=42236]: If the `nl_langinfo(CODESET)` function returns an empty string, Python now uses UTF-8 as the filesystem encoding. Patch by Victor Stinner.
- bpo-42218 [https://bugs.python.org/issue?@action=redirect&bpo=42218]: Fixed a bug in the PEG parser that was causing crashes in debug mode. Now errors are checked in left-recursive rules to avoid cases where such errors do not get handled in time and appear as long-distance crashes in other places.
- bpo-42214 [https://bugs.python.org/issue?@action=redirect&bpo=42214]: Fixed a possible crash in the PEG parser when checking for the '!=' token in the `barry_as_flufl` rule. Patch by Pablo Galindo.
- bpo-42206 [https://bugs.python.org/issue?@action=redirect&bpo=42206]: Propagate and raise the errors caused by **`PyAST_Validate()`** in the parser.
- bpo-41796 [https://bugs.python.org/issue?@action=redirect&bpo=41796]: The **`ast`** module internal state is now per interpreter. Patch by Victor Stinner.
- bpo-42143 [https://bugs.python.org/issue?@action=redirect&bpo=42143]: Fix handling of errors during creation of `PyFunctionObject`, which resulted in operations on uninitialized memory. Patch by Yonatan Goldschmidt.
- bpo-41659 [https://bugs.python.org/issue?@action=redirect&bpo=41659]: Fix a bug in the parser, where a curly brace following a `primary` didn't fail immediately. This led to invalid expressions like `a {b}` to throw a **`SyntaxError`** with a wrong offset, or invalid expressions ending with a curly brace like `a {` to not fail immediately in the REPL.

- bpo-42150 [https://bugs.python.org/issue?@action=redirect&bpo=42150]: Fix possible buffer overflow in the new parser when checking for continuation lines. Patch by Pablo Galindo.
- bpo-42123 [https://bugs.python.org/issue?@action=redirect&bpo=42123]: Run the parser two times. On the first run, disable all the rules that only generate better error messages to gain performance. If there's a parse failure, run the parser a second time with those enabled.
- bpo-42093 [https://bugs.python.org/issue?@action=redirect&bpo=42093]: The `LOAD_ATTR` instruction now uses new "per opcode cache" mechanism and it is about 36% faster now. Patch by Pablo Galindo and Yury Selivanov.
- bpo-42030 [https://bugs.python.org/issue?@action=redirect&bpo=42030]: Support for the legacy AIX-specific shared library loading support has been removed. All versions of AIX since 4.3 have supported and defaulted to using the common Unix mechanism instead.
- bpo-41984 [https://bugs.python.org/issue?@action=redirect&bpo=41984]: The garbage collector now tracks all user-defined classes. Patch by Brandt Bucher.
- bpo-41993 [https://bugs.python.org/issue?@action=redirect&bpo=41993]: Fixed potential issues with removing not completely initialized module from `sys.modules` when import fails.
- bpo-41979 [https://bugs.python.org/issue?@action=redirect&bpo=41979]: Star-unpacking is now allowed for with item's targets in the PEG parser.
- bpo-41974 [https://bugs.python.org/issue?@action=redirect&bpo=41974]: Removed special methods `__int__`, `__float__`, `__floordiv__`, `__mod__`, `__divmod__`, `__rfloordiv__`, `__rmod__` and `__rdivmod__` of the **`complex`** class. They always raised a **`TypeError`**.

- bpo-41902 [https://bugs.python.org/issue?@action=redirect&bpo=41902]: Micro optimization when compute **`sq_item`** and **`mp_subscript`** of **`range`**. Patch by Donghee Na.
- bpo-41894 [https://bugs.python.org/issue?@action=redirect&bpo=41894]: When loading a native module and a load failure occurs, prevent a possible UnicodeDecodeError when not running in a UTF-8 locale by decoding the load error message using the current locale's encoding.
- bpo-41902 [https://bugs.python.org/issue?@action=redirect&bpo=41902]: Micro optimization for range.index if step is 1. Patch by Donghee Na.

Library

- bpo-41435 [https://bugs.python.org/issue?@action=redirect&bpo=41435]: Add `sys._current_exceptions()` function to retrieve a dictionary mapping each thread's identifier to the topmost exception currently active in that thread at the time the function is called.

Core and Builtins

- bpo-38605 [https://bugs.python.org/issue?@action=redirect&bpo=38605]: Enable `from __future__ import annotations` (**PEP 563** [https://peps.python.org/pep-0563/]) by default. The values found in **`__annotations__`** dicts are now strings, for example `{"x": "int"}` instead of `{"x": int}`.

Library

- bpo-35455 [https://bugs.python.org/issue?@action=redirect&bpo=35455]: On Solaris, **`thread_time()`** is now implemented with `gethrvtime()` because `clock_gettime(CLOCK_THREAD_CPUTIME_ID)` is not always available. Patch by Jakub Kulik.

- bpo-42233 [https://bugs.python.org/issue?@action=redirect&bpo=42233]: The **`repr()`** of **`typing`** types containing Generic Alias Types previously did not show the parameterized types in the `GenericAlias`. They have now been changed to do so.

- bpo-29566 [https://bugs.python.org/issue?@action=redirect&bpo=29566]: `binhex.binhex()` consistently writes macOS 9 line endings.

- bpo-26789 [https://bugs.python.org/issue?@action=redirect&bpo=26789]: The **`logging.FileHandler`** class now keeps a reference to the builtin **`open()`** function to be able to open or reopen the file during Python finalization. Fix errors like: `NameError: name 'open' is not defined`. Patch by Victor Stinner.

- bpo-42157 [https://bugs.python.org/issue?@action=redirect&bpo=42157]: Removed the `unicodedata.ucnhash_CAPI` attribute which was an internal PyCapsule object. The related private `_PyUnicode_Name_CAPI` structure was moved to the internal C API. Patch by Victor Stinner.

- bpo-42157 [https://bugs.python.org/issue?@action=redirect&bpo=42157]: Convert the **`unicodedata`** extension module to the multiphase initialization API (**PEP 489** [https://peps.python.org/pep-0489/]) and convert the `unicodedata.UCD` static type to a heap type. Patch by Mohamed Koubaa and Victor Stinner.

- bpo-42146 [https://bugs.python.org/issue?@action=redirect&bpo=42146]: Fix memory leak in

`subprocess.Popen()` in case an uid (gid) specified in `user` (`group`, `extra_groups`) overflows `uid_t` (`gid_t`).

- bpo-42103 [https://bugs.python.org/issue?@action=redirect&bpo=42103]: **`InvalidFileException`** and **`RecursionError`** are now the only errors caused by loading malformed binary Plist file (previously ValueError and TypeError could be raised in some specific cases).
- bpo-41490 [https://bugs.python.org/issue?@action=redirect&bpo=41490]: In `importlib.resources`, `.path` method is more aggressive about releasing handles to zipfile objects early, enabling use-cases like certifi to leave the context open but delete the underlying zip file.
- bpo-41052 [https://bugs.python.org/issue?@action=redirect&bpo=41052]: Pickling heap types implemented in C with protocols 0 and 1 raises now an error instead of producing incorrect data.
- bpo-42089 [https://bugs.python.org/issue?@action=redirect&bpo=42089]: In `importlib.metadata.PackageNotFoundError`, make reference to the package metadata being missing to improve the user experience.
- bpo-41491 [https://bugs.python.org/issue?@action=redirect&bpo=41491]: plistlib: fix parsing XML plists with hexadecimal integer values
- bpo-42065 [https://bugs.python.org/issue?@action=redirect&bpo=42065]: Fix an incorrectly formatted error from **`_codecs.charmap_decode()`** when called with a mapped value outside the range of valid Unicode code points. PR by Max Bernstein.

- bpo-41966 [https://bugs.python.org/issue?@action=redirect&bpo=41966]: Fix pickling pure Python **`datetime.time`** subclasses. Patch by Dean Inwood.

- bpo-19270 [https://bugs.python.org/issue?@action=redirect&bpo=19270]: **`sched.scheduler.cancel()`** will now cancel the correct event, if two events with same priority are scheduled for the same time. Patch by Bar Harel.

- bpo-28660 [https://bugs.python.org/issue?@action=redirect&bpo=28660]: **`textwrap.wrap()`** now attempts to break long words after hyphens when `break_long_words=True` and `break_on_hyphens=True`.

- bpo-35823 [https://bugs.python.org/issue?@action=redirect&bpo=35823]: Use `vfork()` instead of `fork()` for **`subprocess.Popen()`** on Linux to improve performance in cases where it is deemed safe.

- bpo-42043 [https://bugs.python.org/issue?@action=redirect&bpo=42043]: Add support for `zipfile.Path` inheritance. `zipfile.Path.is_file()` now returns False for non-existent names. `zipfile.Path` objects now expose a `.filename` attribute and rely on that to resolve `.name` and `.parent` when the `Path` object is at the root of the zipfile.

- bpo-42021 [https://bugs.python.org/issue?@action=redirect&bpo=42021]: Fix possible ref leaks in **`sqlite3`** module init.

- bpo-39101 [https://bugs.python.org/issue?@action=redirect&bpo=39101]: Fixed tests using IsolatedAsyncioTestCase from hanging on BaseExceptions.

- bpo-41976 [https://bugs.python.org/issue?@action=redirect&bpo=41976]: Fixed a bug that was causing **`ctypes.util.find_library()`** to return `None` when triying to locate a library in an environment when gcc>=9 is available and `ldconfig` is not. Patch by Pablo Galindo

- bpo-41943 [https://bugs.python.org/issue?@action=redirect&bpo=41943]: Fix bug where TestCase.assertLogs doesn't correctly filter messages by level.

- bpo-41923 [https://bugs.python.org/issue?@action=redirect&bpo=41923]: Implement **PEP 613** [https://peps.python.org/pep-0613/], introducing **`typing.TypeAlias`** annotation.

- bpo-41905 [https://bugs.python.org/issue?@action=redirect&bpo=41905]: A new function in abc: *update_abstractmethods* to re-calculate an abstract class's abstract status. In addition, *dataclass* has been changed to call this function.

- bpo-23706 [https://bugs.python.org/issue?@action=redirect&bpo=23706]: Added *newline* parameter to `pathlib.Path.write_text()`.

- bpo-41876 [https://bugs.python.org/issue?@action=redirect&bpo=41876]: Tkinter font class repr uses font name

- bpo-41831 [https://bugs.python.org/issue?@action=redirect&bpo=41831]: `str()` for the `type` attribute of the `tkinter.Event` object always returns now the numeric code returned by Tk instead of the name of the event type.

- bpo-39337 [https://bugs.python.org/issue?@action=redirect&bpo=39337]: `encodings.normalize_encoding()` now ignores non-ASCII characters.

- bpo-41747 [https://bugs.python.org/issue?@action=redirect&bpo=41747]: Ensure all methods that generated from `dataclasses.dataclass()` objects now have the proper `__qualname__` attribute referring to the class they belong to. Patch by Batuhan Taskaya.

- bpo-30681 [https://bugs.python.org/issue?@action=redirect&bpo=30681]: Handle exceptions caused by unparsable date headers when using email “default” policy. Patch by Tim Bell, Georges Toth

- bpo-41586 [https://bugs.python.org/issue?@action=redirect&bpo=41586]: Add F_SETPIPE_SZ and F_GETPIPE_SZ to fcntl module. Allow setting pipesize on subprocess.Popen.

- bpo-41229 [https://bugs.python.org/issue?@action=redirect&bpo=41229]: Add `contextlib.aclosing` for deterministic cleanup of async generators which is analogous to `contextlib.closing` for non-async generators. Patch by Joongi Kim and John Belmonte.

- bpo-16396 [https://bugs.python.org/issue?@action=redirect&bpo=16396]: Allow `ctypes.wintypes` to be imported on non-Windows systems.

- bpo-4356 [https://bugs.python.org/issue?@action=redirect&bpo=4356]: Add a key function to the bisect module.

- bpo-40592 [https://bugs.python.org/issue?@action=redirect&bpo=40592]: `shutil.which()` now ignores empty entries in **`PATHEXT`** instead of treating them as a match.

- bpo-40492 [https://bugs.python.org/issue?@action=redirect&bpo=40492]: Fix `--outfile` for **`cProfile`** / **`profile`** not writing the output file in the original directory when the program being profiled changes the working directory. PR by Anthony Sottile.

- bpo-34204 [https://bugs.python.org/issue?@action=redirect&bpo=34204]: The **`shelve`** module now uses **`pickle.DEFAULT_PROTOCOL`** by default instead of **`pickle`** protocol `3`.

- bpo-27321 [https://bugs.python.org/issue?@action=redirect&bpo=27321]: Fixed KeyError exception when flattening an email to a string attempts to replace a non-existent Content-Transfer-Encoding header.

- bpo-38976 [https://bugs.python.org/issue?@action=redirect&bpo=38976]: The **`http.cookiejar`** module now supports the parsing of cookies in CURL-style cookiejar files through MozillaCookieJar on all platforms. Previously, such cookie entries would be silently ignored when loading a cookiejar with such entries.

  Additionally, the HTTP Only attribute is persisted in the object, and will be correctly written to file if the MozillaCookieJar object is subsequently dumped.

Documentation

- bpo-42061 [https://bugs.python.org/issue?@action=redirect&bpo=42061]: Document __format__ functionality for IP addresses.
- bpo-41910 [https://bugs.python.org/issue?@action=redirect&bpo=41910]: Document the default implementation of `object.__eq__`.

- bpo-42010 [https://bugs.python.org/issue?@action=redirect&bpo=42010]: Clarify that subscription expressions are also valid for certain classes and types in the standard library, and for user-defined classes and types if the classmethod **`__class_getitem__()`** is provided.
- bpo-41805 [https://bugs.python.org/issue?@action=redirect&bpo=41805]: Documented generic alias type and **`types.GenericAlias`**. Also added an entry in glossary for generic types.
- bpo-39693 [https://bugs.python.org/issue?@action=redirect&bpo=39693]: Fix tarfile's extractfile documentation
- bpo-39416 [https://bugs.python.org/issue?@action=redirect&bpo=39416]: Document some restrictions on the default string representations of numeric classes.

Tests

- bpo-41739 [https://bugs.python.org/issue?@action=redirect&bpo=41739]: Fix test_logging.test_race_between_set_target_and_flush(): the test now waits until all threads complete to avoid leaking running threads.
- bpo-41970 [https://bugs.python.org/issue?@action=redirect&bpo=41970]: Avoid a test failure in `test_lib2to3` if the module has already imported at the time the test executes. Patch by Pablo Galindo.
- bpo-41944 [https://bugs.python.org/issue?@action=redirect&bpo=41944]: Tests for CJK codecs no longer call `eval()` on content received via HTTP.
- bpo-41306 [https://bugs.python.org/issue?@action=redirect&bpo=41306]: Fixed a failure in `test_tk.test_widgets.ScaleTest` happening when executing the test with Tk 8.6.10.

Build

- bpo-38980 [https://bugs.python.org/issue?@action=redirect&bpo=38980]: Add `-fno-semantic-interposition` to both the compile and link line when building with `--enable-optimizations`. Patch by Victor Stinner and Pablo Galindo.

Windows

- bpo-38439 [https://bugs.python.org/issue?@action=redirect&bpo=38439]: Updates the icons for IDLE in the Windows Store package.
- bpo-38252 [https://bugs.python.org/issue?@action=redirect&bpo=38252]: Use 8-byte step to detect ASCII sequence in 64-bit Windows build.
- bpo-39107 [https://bugs.python.org/issue?@action=redirect&bpo=39107]: Update Tcl and Tk to 8.6.10 in Windows installer.
- bpo-41557 [https://bugs.python.org/issue?@action=redirect&bpo=41557]: Update Windows installer to use SQLite 3.33.0.
- bpo-38324 [https://bugs.python.org/issue?@action=redirect&bpo=38324]: Avoid Unicode errors when accessing certain locale data on Windows.

macOS

- bpo-41471 [https://bugs.python.org/issue?@action=redirect&bpo=41471]: Ignore invalid prefix lengths in system proxy excludes.

IDLE

- bpo-33987 [https://bugs.python.org/issue?@action=redirect&bpo=33987]: Mostly finish using ttk widgets, mainly for editor, settings, and searches. Some patches by Mark Roseman.
- bpo-40511 [https://bugs.python.org/issue?@action=redirect&bpo=40511]: Typing opening and closing parentheses inside the parentheses of a function call will no longer cause unnecessary “flashing” off and on of an existing open call-tip, e.g. when typed in a string literal.
- bpo-38439 [https://bugs.python.org/issue?@action=redirect&bpo=38439]: Add a 256×256 pixel IDLE icon to the Windows .ico file. Created by Andrew Clover. Remove the low-color gif variations from the .ico file.

C API

- bpo-42157 [https://bugs.python.org/issue?@action=redirect&bpo=42157]: The private `_PyUnicode_Name_CAPI` structure of the PyCapsule API `unicodedata.ucnhash_CAPI` has been moved to the internal C API. Patch by Victor Stinner.
- bpo-42015 [https://bugs.python.org/issue?@action=redirect&bpo=42015]: Fix potential crash in deallocating method objects when dynamically allocated **`PyMethodDef`**’s lifetime is managed through the `self` argument of a **`PyCFunction`**.
- bpo-40423 [https://bugs.python.org/issue?@action=redirect&bpo=40423]: The **`subprocess`** module and `os.closerange` will now use the `close_range(low, high, flags)` syscall when it is available for more efficient closing of ranges of descriptors.
- bpo-41845 [https://bugs.python.org/issue?@action=redirect&bpo=41845]: **`PyObject_GenericGetDict()`** is available again in the limited API when targeting 3.10 or later.

- bpo-40422 [https://bugs.python.org/issue?@action=redirect&bpo=40422]: Add `_Py_closerange` function to provide performant closing of a range of file descriptors.
- bpo-41986 [https://bugs.python.org/issue?@action=redirect&bpo=41986]: **`Py_FileSystemDefaultEncodeErrors`** and **`Py_UTF8Mode`** are available again in limited API.
- bpo-41756 [https://bugs.python.org/issue?@action=redirect&bpo=41756]: Add `PyIter_Send` function to allow sending value into generator/coroutine/iterator without raising StopIteration exception to signal return.
- bpo-41784 [https://bugs.python.org/issue?@action=redirect&bpo=41784]: Added `PyUnicode_AsUTF8AndSize` to the limited C API.

Python 3.10.0 alpha 1

Release date: 2020-10-05

Security

- bpo-41304 [https://bugs.python.org/issue?@action=redirect&bpo=41304]: Fixes `python3x._pth` being ignored on Windows, caused by the fix for bpo-29778 [https://bugs.python.org/issue?@action=redirect&bpo=29778] (**CVE 2020-15801** [https://www.cve.org/CVERecord?id=CVE-2020-15801]).
- bpo-41162 [https://bugs.python.org/issue?@action=redirect&bpo=41162]: Audit hooks are now cleared later during finalization to avoid missing events.
- bpo-29778 [https://bugs.python.org/issue?@action=redirect&bpo=29778]: Ensure `python3.dll` is loaded from correct locations when Python is embedded (**CVE 2020-15523** [https://www.cve.org/CVERecord?id=CVE-2020-15523]).
- bpo-41004 [https://bugs.python.org/issue?@action=redirect&bpo=41004]: The __hash__() methods of ipaddress.IPv4Interface and ipaddress.IPv6Interface

incorrectly generated constant hash values of 32 and 128 respectively. This resulted in always causing hash collisions. The fix uses hash() to generate hash values for the tuple of (address, mask length, network address).
- bpo-39603 [https://bugs.python.org/issue?@action=redirect&bpo=39603]: Prevent http header injection by rejecting control characters in http.client.putrequest(…).

Core and Builtins

- bpo-41909 [https://bugs.python.org/issue?@action=redirect&bpo=41909]: Fixed stack overflow in **`issubclass()`** and **`isinstance()`** when getting the `__bases__` attribute leads to infinite recursion.
- bpo-41922 [https://bugs.python.org/issue?@action=redirect&bpo=41922]: Speed up calls to `reversed()` by using the **PEP 590** [https://peps.python.org/pep-0590/] `vectorcall` calling convention. Patch by Donghee Na.
- bpo-41873 [https://bugs.python.org/issue?@action=redirect&bpo=41873]: Calls to `float()` are now faster due to the `vectorcall` calling convention. Patch by Dennis Sweeney.
- bpo-41870 [https://bugs.python.org/issue?@action=redirect&bpo=41870]: Speed up calls to `bool()` by using the **PEP 590** [https://peps.python.org/pep-0590/] `vectorcall` calling convention. Patch by Donghee Na.

Library

- bpo-1635741 [https://bugs.python.org/issue?@action=redirect&bpo=1635741]: Port the **`_bisect`** module to the multi-phase initialization API (**PEP 489** [https://peps.python.org/pep-0489/]).

Core and Builtins

- bpo-39934 [https://bugs.python.org/issue?@action=redirect&bpo=39934]: Correctly count control blocks in ‘except’ in compiler. Ensures that a syntax error, rather a fatal error, occurs for deeply nested, named exception handlers.
- bpo-41780 [https://bugs.python.org/issue?@action=redirect&bpo=41780]: Fix `__dir__()` of `types.GenericAlias`. Patch by Batuhan Taskaya.

Library

- bpo-1635741 [https://bugs.python.org/issue?@action=redirect&bpo=1635741]: Port the `_lsprof` extension module to multi-phase initialization (**PEP 489** [https://peps.python.org/pep-0489/]).
- bpo-1635741 [https://bugs.python.org/issue?@action=redirect&bpo=1635741]: Port the `cmath` extension module to multi-phase initialization (**PEP 489** [https://peps.python.org/pep-0489/]).
- bpo-1635741 [https://bugs.python.org/issue?@action=redirect&bpo=1635741]: Port the `_scproxy` extension module to multi-phase initialization (**PEP 489** [https://peps.python.org/pep-0489/]).
- bpo-1635741 [https://bugs.python.org/issue?@action=redirect&bpo=1635741]: Port the `termios` extension module to multi-phase initialization (**PEP 489** [https://peps.python.org/pep-0489/]).
- bpo-1635741 [https://bugs.python.org/issue?@action=redirect&bpo=1635741]: Convert the `_sha256` extension module types to heap types.

Core and Builtins

- bpo-41690 [https://bugs.python.org/issue?@action=redirect&bpo=41690]: Fix a possible stack overflow in the parser when parsing functions and classes with a huge amount of arguments. Patch by Pablo Galindo.

Library

- bpo-1635741 [https://bugs.python.org/issue?@action=redirect&bpo=1635741]: Port the `_overlapped` extension module to multi-phase initialization (**PEP 489** [https://peps.python.org/pep-0489/]).
- bpo-1635741 [https://bugs.python.org/issue?@action=redirect&bpo=1635741]: Port the `_curses_panel` extension module to multi-phase initialization (**PEP 489** [https://peps.python.org/pep-0489/]).
- bpo-1635741 [https://bugs.python.org/issue?@action=redirect&bpo=1635741]: Port the `_opcode` extension module to multi-phase initialization (**PEP 489** [https://peps.python.org/pep-0489/]).

Core and Builtins

- bpo-41681 [https://bugs.python.org/issue?@action=redirect&bpo=41681]: Fixes the wrong error description in the error raised by using 2 , in format string in f-string and `str.format()`.

Library

- bpo-41675 [https://bugs.python.org/issue?@action=redirect&bpo=41675]: The implementation of `signal.siginterrupt()` now uses `sigaction()` (if it is available in the system) instead of the deprecated `siginterrupt()`. Patch by Pablo Galindo.

Core and Builtins

- bpo-41670 [https://bugs.python.org/issue?@action=redirect&bpo=41670]: Prevent line trace being skipped on platforms not compiled with `USE_COMPUTED_GOTOS`. Fixes issue where some lines nested within a try-except block were not being traced on Windows.
- bpo-41654 [https://bugs.python.org/issue?@action=redirect&bpo=41654]: Fix a crash that occurred when destroying subclasses of **`MemoryError`**. Patch by Pablo Galindo.

Library

- bpo-1635741 [https://bugs.python.org/issue?@action=redirect&bpo=1635741]: Port the **`zlib`** extension module to multi-phase initialization (**PEP 489** [https://peps.python.org/pep-0489/]).

Core and Builtins

- bpo-41631 [https://bugs.python.org/issue?@action=redirect&bpo=41631]: The `_ast` module uses again a global state. Using a module state per module instance is causing subtle practical problems. For example, the Mercurial project replaces the `__import__()` function to implement lazy import, whereas Python expected that `import _ast` always return a fully initialized `_ast` module.

Library

- bpo-40077 [https://bugs.python.org/issue?@action=redirect&bpo=40077]: Convert **`_operator`** to use **`PyType_FromSpec()`**.

Core and Builtins

- bpo-1653741 [https://bugs.python.org/issue?@action=redirect&bpo=1653741]: Port **`_sha3`** to multi-phase init. Convert static types to heap types.

Library

- bpo-1635741 [https://bugs.python.org/issue?@action=redirect&bpo=1635741]: Port the **`_blake2`** extension module to the multi-phase initialization API (**PEP 489** [https://peps.python.org/pep-0489/]).

Core and Builtins

- bpo-41533 [https://bugs.python.org/issue?@action=redirect&bpo=41533]: Free the stack allocated in `va_build_stack` if `do_mkstack` fails and the stack is not a `small_stack`.
- bpo-41531 [https://bugs.python.org/issue?@action=redirect&bpo=41531]: Fix a bug that was dropping keys when compiling dict literals with more than 0xFFFF elements. Patch by Pablo Galindo.
- bpo-41525 [https://bugs.python.org/issue?@action=redirect&bpo=41525]: The output of `python --help` contains now only ASCII characters.

Library

- bpo-1635741 [https://bugs.python.org/issue?@action=redirect&bpo=1635741]: Port the **`_sha1`**, **`_sha512`**, and **`_md5`** extension modules to multi-phase initialization API (**PEP 489** [https://peps.python.org/pep-0489/]).

Core and Builtins

- bpo-41431 [https://bugs.python.org/issue?@action=redirect&bpo=41431]: Optimize `dict_merge()` for copying dict (e.g. `dict(d)` and `{}.update(d)`).
- bpo-41428 [https://bugs.python.org/issue?@action=redirect&bpo=41428]: Implement PEP 604. This supports (int | str) etc. in place of Union[str, int].
- bpo-41340 [https://bugs.python.org/issue?@action=redirect&bpo=41340]: Removed fallback implementation for `strdup`.
- bpo-38156 [https://bugs.python.org/issue?@action=redirect&bpo=38156]: Handle interrupts that come after EOF correctly in `PyOS_StdioReadline`.
- bpo-41342 [https://bugs.python.org/issue?@action=redirect&bpo=41342]: **`round()`** with integer argument is now faster (9–60%).
- bpo-41334 [https://bugs.python.org/issue?@action=redirect&bpo=41334]: Constructors **`str()`**, **`bytes()`** and **`bytearray()`** are now faster (around 30–40% for small objects).
- bpo-41295 [https://bugs.python.org/issue?@action=redirect&bpo=41295]: Resolve a regression in CPython 3.8.4 where defining “__setattr__” in a multi-inheritance setup and calling up the hierarchy chain could fail if builtins/extension types were involved in the base types.
- bpo-41323 [https://bugs.python.org/issue?@action=redirect&bpo=41323]: Bytecode optimizations are performed directly on the control flow graph. This will result in slightly more compact code objects in some circumstances.
- bpo-41247 [https://bugs.python.org/issue?@action=redirect&bpo=41247]: Always cache the running loop holder when running `asyncio.set_running_loop`.
- bpo-41252 [https://bugs.python.org/issue?@action=redirect&bpo=41252]: Fix incorrect refcounting in _ssl.c’s `_servername_callback()`.

- bpo-1635741 [https://bugs.python.org/issue?@action=redirect&bpo=1635741]: Port **`multiprocessing`** to multi-phase initialization

Library

- bpo-1635741 [https://bugs.python.org/issue?@action=redirect&bpo=1635741]: Port **`winapi`** to multiphase initialization

Core and Builtins

- bpo-41215 [https://bugs.python.org/issue?@action=redirect&bpo=41215]: Use non-NULL default values in the PEG parser keyword list to overcome a bug that was preventing Python from being properly compiled when using the XLC compiler. Patch by Pablo Galindo.
- bpo-41218 [https://bugs.python.org/issue?@action=redirect&bpo=41218]: Python 3.8.3 had a regression where compiling with ast.PyCF_ALLOW_TOP_LEVEL_AWAIT would aggressively mark list comprehension with CO_COROUTINE. Now only list comprehension making use of async/await will tagged as so.

Library

- bpo-1635741 [https://bugs.python.org/issue?@action=redirect&bpo=1635741]: Port **`faulthandler`** to multiphase initialization.
- bpo-1635741 [https://bugs.python.org/issue?@action=redirect&bpo=1635741]: Port **`sha256`** to multiphase initialization

Core and Builtins

- bpo-41175 [https://bugs.python.org/issue?@action=redirect&bpo=41175]: Guard against a NULL pointer dereference within bytearrayobject triggered by the `bytearray() + bytearray()` operation.
- bpo-41100 [https://bugs.python.org/issue?@action=redirect&bpo=41100]: add arm64 to the allowable Mac OS arches in mpdecimal.h
- bpo-41094 [https://bugs.python.org/issue?@action=redirect&bpo=41094]: Fix decoding errors with audit when open files with non-ASCII names on non-UTF-8 locale.
- bpo-39960 [https://bugs.python.org/issue?@action=redirect&bpo=39960]: The “hackcheck” that prevents sneaking around a type’s __setattr__() by calling the superclass method was rewritten to allow C implemented heap types.
- bpo-41084 [https://bugs.python.org/issue?@action=redirect&bpo=41084]: Prefix the error message with ‘f-string: ‘, when parsing an f-string expression which throws a **`SyntaxError`**.
- bpo-40521 [https://bugs.python.org/issue?@action=redirect&bpo=40521]: Empty frozensets are no longer singletons.
- bpo-41076 [https://bugs.python.org/issue?@action=redirect&bpo=41076]: Pre-feed the parser with the location of the f-string expression, not the f-string itself, which allows us to skip the shifting of the AST node locations after the parsing is completed.
- bpo-41056 [https://bugs.python.org/issue?@action=redirect&bpo=41056]: Fixes a reference to deallocated stack space during startup when constructing sys.path involving a relative symlink when code was supplied via -c. (discovered via Coverity)
- bpo-41061 [https://bugs.python.org/issue?@action=redirect&bpo=41061]: Fix incorrect expressions and asserts in hashtable code and tests.

- bpo-41052 [https://bugs.python.org/issue?@action=redirect&bpo=41052]: Opt out serialization/deserialization for _random.Random
- bpo-40939 [https://bugs.python.org/issue?@action=redirect&bpo=40939]: Rename `PyPegen*` functions to `PyParser*`, so that we can remove the old set of `PyParser*` functions that were using the old parser, but keep everything backwards-compatible.
- bpo-35975 [https://bugs.python.org/issue?@action=redirect&bpo=35975]: Stefan Behnel reported that cf_feature_version is used even when PyCF_ONLY_AST is not set. This is against the intention and against the documented behavior, so it's been fixed.
- bpo-40939 [https://bugs.python.org/issue?@action=redirect&bpo=40939]: Remove the remaining files from the old parser and the **`symbol`** module.

Library

- bpo-40077 [https://bugs.python.org/issue?@action=redirect&bpo=40077]: Convert **`_bz2`** to use **`PyType_FromSpec()`**.

Core and Builtins

- bpo-41006 [https://bugs.python.org/issue?@action=redirect&bpo=41006]: The `encodings.latin_1` module is no longer imported at startup. Now it is only imported when it is the filesystem encoding or the stdio encoding.
- bpo-40636 [https://bugs.python.org/issue?@action=redirect&bpo=40636]: **`zip()`** now supports **PEP 618** [https://peps.python.org/pep-0618/]'s `strict` parameter, which raises a **`ValueError`** if the arguments are exhausted at different lengths. Patch by Brandt Bucher.
- bpo-1635741 [https://bugs.python.org/issue?@action=redirect&bpo=1635741]: Port **`_gdbm`** to multiphase

initialization.

- bpo-40985 [https://bugs.python.org/issue?@action=redirect&bpo=40985]: Fix a bug that caused the `SyntaxError` text to be empty when a file ends with a line ending in a line continuation character (i.e. backslash). The error text should contain the text of the last line.
- bpo-40958 [https://bugs.python.org/issue?@action=redirect&bpo=40958]: Fix a possible buffer overflow in the PEG parser when gathering information for emitting syntax errors. Patch by Pablo Galindo.
- bpo-1635741 [https://bugs.python.org/issue?@action=redirect&bpo=1635741]: Port `_dbm` to multiphase initialization.
- bpo-40957 [https://bugs.python.org/issue?@action=redirect&bpo=40957]: Fix refleak in _Py_fopen_obj() when PySys_Audit() fails

Library

- bpo-40950 [https://bugs.python.org/issue?@action=redirect&bpo=40950]: Add a state to the `nis` module (**PEP 3121** [https://peps.python.org/pep-3121/]) and apply the multiphase initialization. Patch by Donghee Na.

Core and Builtins

- bpo-40947 [https://bugs.python.org/issue?@action=redirect&bpo=40947]: The Python Path Configuration now takes `PyConfig.platlibdir` in account.

- bpo-40939 [https://bugs.python.org/issue?@action=redirect&bpo=40939]: Remove the old parser, the `parser` module and all associated support code, command-line options and environment variables. Patch by Pablo Galindo.

- bpo-40847 [https://bugs.python.org/issue?@action=redirect&bpo=40847]: Fix a bug where a line with only a line continuation character is not considered a blank line at tokenizer level. In such cases, more than a single `NEWLINE` token was emitted. The old parser was working around the issue, but the new parser threw a **`SyntaxError`** for valid input due to this. For example, an empty line following a line continuation character was interpreted as a **`SyntaxError`**.

- bpo-40890 [https://bugs.python.org/issue?@action=redirect&bpo=40890]: Each dictionary view now has a `mapping` attribute that provides a **`types.MappingProxyType`** wrapping the original dictionary. Patch contributed by Dennis Sweeney.

- bpo-40889 [https://bugs.python.org/issue?@action=redirect&bpo=40889]: Improved the performance of symmetric difference operations on dictionary item views. Patch by Dennis Sweeney.

- bpo-40904 [https://bugs.python.org/issue?@action=redirect&bpo=40904]: Fix possible segfault in the new PEG parser when parsing f-string containing yield statements with no value (`f"{yield}"`). Patch by Pablo Galindo

- bpo-40903 [https://bugs.python.org/issue?@action=redirect&bpo=40903]: Fixed a possible segfault in the new PEG parser when producing error messages for invalid assignments of the form `p=p=`. Patch by Pablo Galindo

- bpo-40880 [https://bugs.python.org/issue?@action=redirect&bpo=40880]: Fix invalid memory read in the new parser when checking newlines in string literals. Patch by Pablo Galindo.

- bpo-40883 [https://bugs.python.org/issue?@action=redirect&bpo=40883]: Fix memory leak in when parsing f-strings in the new parser. Patch by Pablo Galindo
- bpo-40870 [https://bugs.python.org/issue?@action=redirect&bpo=40870]: Raise `ValueError` when validating custom AST’s where the constants `True`, `False` and `None` are used within a `ast.Name` node.
- bpo-40854 [https://bugs.python.org/issue?@action=redirect&bpo=40854]: Allow overriding `sys.platlibdir` via a new `PYTHONPLATLIBDIR` environment variable.
- bpo-40826 [https://bugs.python.org/issue?@action=redirect&bpo=40826]: Fix GIL usage in **`PyOS_Readline()`**: lock the GIL to set an exception and pass the Python thread state when checking if there is a pending signal.
- bpo-1635741 [https://bugs.python.org/issue?@action=redirect&bpo=1635741]: Port `fcntl` to multiphase initialization.
- bpo-19468 [https://bugs.python.org/issue?@action=redirect&bpo=19468]: Delete unnecessary instance check in importlib.reload(). Patch by Furkan Önder.
- bpo-40824 [https://bugs.python.org/issue?@action=redirect&bpo=40824]: Unexpected errors in calling the `__iter__` method are no longer masked by `TypeError` in the `in` operator and functions `contains()`, `indexOf()` and `countOf()` of the `operator` module.

- bpo-40792 [https://bugs.python.org/issue?@action=redirect&bpo=40792]: Attributes `start`, `stop` and `step` of the **`range`** object now always has exact type **`int`**. Previously, they could have been an instance of a subclass of `int`.

- bpo-40780 [https://bugs.python.org/issue?@action=redirect&bpo=40780]: Fix a corner case where g-style string formatting of a float failed to remove trailing zeros.

- bpo-38964 [https://bugs.python.org/issue?@action=redirect&bpo=38964]: When there's a **`SyntaxError`** in the expression part of an fstring, the filename attribute of the **`SyntaxError`** gets correctly set to the name of the file the fstring resides in.

- bpo-40750 [https://bugs.python.org/issue?@action=redirect&bpo=40750]: Support the "-d" debug flag in the new PEG parser. Patch by Pablo Galindo

- bpo-40217 [https://bugs.python.org/issue?@action=redirect&bpo=40217]: Instances of types created with **`PyType_FromSpecWithBases()`** will no longer automatically visit their class object when traversing references in the garbage collector. The user is expected to manually visit the object's class. Patch by Pablo Galindo.

- bpo-39573 [https://bugs.python.org/issue?@action=redirect&bpo=39573]: **`Py_TYPE()`** is changed to the inline static function. Patch by Donghee Na.

- bpo-40696 [https://bugs.python.org/issue?@action=redirect&bpo=40696]: Fix a hang that can arise after **`generator.throw()`** due to a cycle in the exception context chain.

- bpo-40521 [https://bugs.python.org/issue?@action=redirect&bpo=40521]: Each interpreter now its has own free lists, singletons and caches:

  - Free lists: float, tuple, list, dict, frame, context, asynchronous generator, MemoryError.
  - Singletons: empty tuple, empty bytes string, empty Unicode string, single byte character, single Unicode (latin1) character.
  - Slice cache.

  They are no longer shared by all interpreters.

- bpo-40679 [https://bugs.python.org/issue?@action=redirect&bpo=40679]: Certain **`TypeError`** messages about missing or extra arguments now include the function’s qualified name. Patch by Dennis Sweeney.

- bpo-29590 [https://bugs.python.org/issue?@action=redirect&bpo=29590]: Make the stack trace correct after calling **`generator.throw()`** on a generator that has yielded from a `yield from`.

- bpo-4022 [https://bugs.python.org/issue?@action=redirect&bpo=4022]: Improve performance of generators by not raising internal StopIteration.

Library

- bpo-1635741 [https://bugs.python.org/issue?@action=redirect&bpo=1635741]: Port **`mmap`** to multiphase initialization.
- bpo-1635741 [https://bugs.python.org/issue?@action=redirect&bpo=1635741]: Port **`_lzma`** to multiphase initialization.

Core and Builtins

- bpo-37999 [https://bugs.python.org/issue?@action=redirect&bpo=37999]: Builtin and extension functions that take integer arguments no longer accept `Decimal`s, `Fraction`s and other objects that can be converted to integers only with a loss (e.g. that have the `__int__()` method but do not have the `__index__()` method).
- bpo-29882 [https://bugs.python.org/issue?@action=redirect&bpo=29882]: Add `int.bit_count()`, counting the number of ones in the binary representation of an integer. Patch by Niklas Fiekas.

Library

- bpo-36982 [https://bugs.python.org/issue?@action=redirect&bpo=36982]: Use ncurses extended color functions when available to support terminals with 256 colors, and add the new function `curses.has_extended_color_support()` to indicate whether extended color support is provided by the underlying ncurses library.

Core and Builtins

- bpo-19569 [https://bugs.python.org/issue?@action=redirect&bpo=19569]: Add the private macros `_Py_COMP_DIAG_PUSH`, `_Py_COMP_DIAG_IGNORE_DEPR_DECLS`, and `_Py_COMP_DIAG_POP`.
- bpo-26680 [https://bugs.python.org/issue?@action=redirect&bpo=26680]: The int type now supports the x.is_integer() method for compatibility with float.

Library

- bpo-41900 [https://bugs.python.org/issue?@action=redirect&bpo=41900]: C14N 2.0 serialisation in xml.etree.ElementTree failed for unprefixed attributes when a default namespace was defined.

- bpo-41887 [https://bugs.python.org/issue?@action=redirect&bpo=41887]: Strip leading spaces and tabs on `ast.literal_eval()`. Also document stripping of spaces and tabs for `eval()`.

- bpo-41773 [https://bugs.python.org/issue?@action=redirect&bpo=41773]: Note in documentation that `random.choices()` doesn’t support non-finite weights, raise `ValueError` when given non-finite weights.

- bpo-41840 [https://bugs.python.org/issue?@action=redirect&bpo=41840]: Fix a bug in the `symtable` module that was causing module-scope global variables to not be reported as both local and global. Patch by Pablo Galindo.

- bpo-41842 [https://bugs.python.org/issue?@action=redirect&bpo=41842]: Add `codecs.unregister()` function to unregister a codec search function.

- bpo-40564 [https://bugs.python.org/issue?@action=redirect&bpo=40564]: In `zipfile.Path`, mutate the passed ZipFile object type instead of making a copy. Prevents issues when both the local copy and the caller’s copy attempt to close the same file handle.

- bpo-40670 [https://bugs.python.org/issue?@action=redirect&bpo=40670]: More reliable validation of statements in `timeit.Timer`. It now accepts “empty” statements (only whitespaces and comments) and rejects misindentent statements.

- bpo-41833 [https://bugs.python.org/issue?@action=redirect&bpo=41833]: The **`threading.Thread`** constructor now uses the target name if the *target* argument is specified but the *name* argument is omitted.

- bpo-41817 [https://bugs.python.org/issue?@action=redirect&bpo=41817]: fix `tkinter.EventType` Enum so all members are strings, and none are tuples

- bpo-41810 [https://bugs.python.org/issue?@action=redirect&bpo=41810]: **`types.EllipsisType`**, **`types.NotImplementedType`** and **`types.NoneType`** have been reintroduced, providing a new set of types readily interpretable by static type checkers.

- bpo-41815 [https://bugs.python.org/issue?@action=redirect&bpo=41815]: Fix SQLite3 segfault when backing up closed database. Patch contributed by Peter David McCormick.

- bpo-41816 [https://bugs.python.org/issue?@action=redirect&bpo=41816]: StrEnum added: it ensures that all members are already strings or string candidates

- bpo-41517 [https://bugs.python.org/issue?@action=redirect&bpo=41517]: fix bug allowing Enums to be extended via multiple inheritance

- bpo-39587 [https://bugs.python.org/issue?@action=redirect&bpo=39587]: use the correct mix-in data type when constructing Enums

- bpo-41792 [https://bugs.python.org/issue?@action=redirect&bpo=41792]: Add is_typeddict function to typing.py to check if a type is a TypedDict class

Previously there was no way to check that without using private API. See the relevant issue in python/typing [https://github.com/python/typing/issues/751].

- bpo-41789 [https://bugs.python.org/issue?@action=redirect&bpo=41789]: Honor `object` overrides in `Enum` class creation (specifically, `__str__`, `__repr__`, `__format__`, and `__reduce_ex__`).

- bpo-32218 [https://bugs.python.org/issue?@action=redirect&bpo=32218]: `enum.Flag` and `enum.IntFlag` members are now iterable.

- bpo-39651 [https://bugs.python.org/issue?@action=redirect&bpo=39651]: Fix a race condition in the `call_soon_threadsafe()` method of `asyncio.ProactorEventLoop`: do nothing if the self-pipe socket has been closed.

- bpo-1635741 [https://bugs.python.org/issue?@action=redirect&bpo=1635741]: Port the `mashal` extension module to the multi-phase initialization API (**PEP 489** [https://peps.python.org/pep-0489/]).

- bpo-1635741 [https://bugs.python.org/issue?@action=redirect&bpo=1635741]: Port the `_string` extension module to the multi-phase initialization API (**PEP 489** [https://peps.python.org/pep-0489/]).

- bpo-41732 [https://bugs.python.org/issue?@action=redirect&bpo=41732]: Added an iterator to **`memoryview`**.

- bpo-41720 [https://bugs.python.org/issue?@action=redirect&bpo=41720]: Fixed **`turtle.Vec2D.__rmul__()`** for arguments which are not int or float.

- bpo-41696 [https://bugs.python.org/issue?@action=redirect&bpo=41696]: Fix handling of debug mode in **`asyncio.run()`**. This allows setting `PYTHONASYNCIODEBUG` or `-X dev` to enable asyncio debug mode when using **`asyncio.run()`**.

- bpo-41687 [https://bugs.python.org/issue?@action=redirect&bpo=41687]: Fix implementation of sendfile to be compatible with Solaris.

- bpo-41662 [https://bugs.python.org/issue?@action=redirect&bpo=41662]: No longer override exceptions raised in `__len__()` of a sequence of parameters in **`sqlite3`** with **`ProgrammingError`**.

- bpo-39010 [https://bugs.python.org/issue?@action=redirect&bpo=39010]: Restarting a `ProactorEventLoop` on Windows no longer logs spurious `ConnectionResetErrors`.

- bpo-41638 [https://bugs.python.org/issue?@action=redirect&bpo=41638]: **`ProgrammingError`** message for absent parameter in **`sqlite3`** contains now the name of the parameter instead of its index when parameters are supplied as a dict.

- bpo-41662 [https://bugs.python.org/issue?@action=redirect&bpo=41662]: Fixed crash when mutate list of parameters during iteration in **`sqlite3`**.

- bpo-41513 [https://bugs.python.org/issue?@action=redirect&bpo=41513]: Improved the accuracy of math.hypot(). Internally, each step is computed with extra precision so that the result is now almost always correctly rounded.

- bpo-41609 [https://bugs.python.org/issue?@action=redirect&bpo=41609]: The pdb whatis command correctly reports instance methods as ‘Method’ rather than ‘Function’.

- bpo-39994 [https://bugs.python.org/issue?@action=redirect&bpo=39994]: Fixed pprint’s handling of dict subclasses that override __repr__.

- bpo-32751 [https://bugs.python.org/issue?@action=redirect&bpo=32751]: When cancelling the task due to a timeout, **`asyncio.wait_for()`** will now wait until the cancellation is complete also in the case when *timeout* is <= 0, like it does with positive timeouts.

- bpo-37658 [https://bugs.python.org/issue?@action=redirect&bpo=37658]: **`asyncio.wait_for()`** now properly handles races between cancellation of itself and the completion of the wrapped awaitable.

- bpo-40782 [https://bugs.python.org/issue?@action=redirect&bpo=40782]: Change the method asyncio.AbstractEventLoop.run_in_executor to not be a coroutine.

- bpo-41520 [https://bugs.python.org/issue?@action=redirect&bpo=41520]: Fix **`codeop`** regression that prevented turning compile warnings into errors.

- bpo-41528 [https://bugs.python.org/issue?@action=redirect&bpo=41528]: turtle uses math module functions to convert degrees to radians and vice versa and to calculate vector norm

- bpo-41513 [https://bugs.python.org/issue?@action=redirect&bpo=41513]: Minor algorithmic improvement to math.hypot() and math.dist() giving small gains in speed and accuracy.

- bpo-41503 [https://bugs.python.org/issue?@action=redirect&bpo=41503]: Fixed a race between setTarget and flush in logging.handlers.MemoryHandler.

- bpo-41497 [https://bugs.python.org/issue?@action=redirect&bpo=41497]: Fix potential UnicodeDecodeError in dis module.

- bpo-41467 [https://bugs.python.org/issue?@action=redirect&bpo=41467]: On Windows, fix asyncio `recv_into()` return value when the socket/pipe is closed (**`BrokenPipeError`**): return `0` rather than an empty byte string (`b''`).

- bpo-41425 [https://bugs.python.org/issue?@action=redirect&bpo=41425]: Make tkinter doc example runnable.

- bpo-41421 [https://bugs.python.org/issue?@action=redirect&bpo=41421]: Make an algebraic simplification to random.paretovariate(). It now is slightly less subject to round-off error and is slightly faster. Inputs that used to cause ZeroDivisionError now cause an OverflowError instead.

- bpo-41440 [https://bugs.python.org/issue?@action=redirect&bpo=41440]: Add **`os.cpu_count()`** support for VxWorks RTOS.

- bpo-41316 [https://bugs.python.org/issue?@action=redirect&bpo=41316]: Fix the **`tarfile`** module to write only basename of TAR file to GZIP compression header.

- bpo-41384 [https://bugs.python.org/issue?@action=redirect&bpo=41384]: Raise TclError instead of TypeError when an unknown option is passed to tkinter.OptionMenu.

- bpo-41317 [https://bugs.python.org/issue?@action=redirect&bpo=41317]: Use add_done_callback() in asyncio.loop.sock_accept() to unsubscribe reader early on cancellation.

- bpo-41364 [https://bugs.python.org/issue?@action=redirect&bpo=41364]: Reduce import overhead of **`uuid`**.

- bpo-35328 [https://bugs.python.org/issue?@action=redirect&bpo=35328]: Set the environment variable `VIRTUAL_ENV_PROMPT` at **`venv`** activation.

- bpo-41341 [https://bugs.python.org/issue?@action=redirect&bpo=41341]: Recursive evaluation of `typing.ForwardRef` in `get_type_hints`.

- bpo-41344 [https://bugs.python.org/issue?@action=redirect&bpo=41344]: Prevent creating **`shared_memory.SharedMemory`** objects with `size=0`.

- bpo-41333 [https://bugs.python.org/issue?@action=redirect&bpo=41333]: **`collections.OrderedDict.pop()`** is now 2 times faster.

- bpo-41288 [https://bugs.python.org/issue?@action=redirect&bpo=41288]: Unpickling invalid NEWOBJ_EX opcode with the C implementation raises now UnpicklingError instead of crashing.

- bpo-39017 [https://bugs.python.org/issue?@action=redirect&bpo=39017]: Avoid infinite loop when reading specially crafted TAR files using the tarfile module (**CVE 2019-20907** [https://www.cve.org/CVERecord?id=CVE-2019-20907]).

- bpo-41273 [https://bugs.python.org/issue?@action=redirect&bpo=41273]: Speed up any transport using `_ProactorReadPipeTransport` by calling `recv_into` instead of `recv`, thus not creating a new buffer for each `recv` call in the transport’s read loop.

- bpo-41235 [https://bugs.python.org/issue?@action=redirect&bpo=41235]: Fix the error handling in **`ssl.SSLContext.load_dh_params()`**.

- bpo-41207 [https://bugs.python.org/issue?@action=redirect&bpo=41207]: In distutils.spawn, restore expectation that DistutilsExecError is raised when the command is not found.

- bpo-29727 [https://bugs.python.org/issue?@action=redirect&bpo=29727]: Register **`array.array`** as a **`MutableSequence`**. Patch by Pablo Galindo.

- bpo-39168 [https://bugs.python.org/issue?@action=redirect&bpo=39168]: Remove the `__new__` method of **`typing.Generic`**.

- bpo-41194 [https://bugs.python.org/issue?@action=redirect&bpo=41194]: Fix a crash in the `_ast` module: it can no longer be loaded more than once. It now uses a global state rather than a module state.

- bpo-41195 [https://bugs.python.org/issue?@action=redirect&bpo=41195]: Add read-only ssl.SSLContext.security_level attribute to retrieve the context’s security level.

- bpo-41193 [https://bugs.python.org/issue?@action=redirect&bpo=41193]: The `write_history()` atexit function of the readline completer now ignores any **`OSError`** to ignore error if the filesystem is read-only, instead of only ignoring **`FileNotFoundError`** and **`PermissionError`**.

- bpo-41182 [https://bugs.python.org/issue?@action=redirect&bpo=41182]: selector: use DefaultSelector based upon implementation

- bpo-41161 [https://bugs.python.org/issue?@action=redirect&bpo=41161]: The decimal module now requires libmpdec-2.5.0. Users of –with-system-libmpdec should update their system library.

- bpo-40874 [https://bugs.python.org/issue?@action=redirect&bpo=40874]: The decimal module now requires libmpdec-2.5.0.

- bpo-41138 [https://bugs.python.org/issue?@action=redirect&bpo=41138]: Fixed the **`trace`** module CLI for Python source files with non-UTF-8 encoding.

- bpo-31082 [https://bugs.python.org/issue?@action=redirect&bpo=31082]: Use the term “iterable” in the docstring for **`functools.reduce()`**.

- bpo-40521 [https://bugs.python.org/issue?@action=redirect&bpo=40521]: Remove freelist from collections.deque().

- bpo-31938 [https://bugs.python.org/issue?@action=redirect&bpo=31938]: Fix default-value signatures of several functions in the **`select`** module - by Anthony Sottile.

- bpo-41068 [https://bugs.python.org/issue?@action=redirect&bpo=41068]: Fixed reading files with non-ASCII names from ZIP archive directly after writing them.
- bpo-41058 [https://bugs.python.org/issue?@action=redirect&bpo=41058]: **`pdb.find_function()`** now correctly determines the source file encoding.
- bpo-41056 [https://bugs.python.org/issue?@action=redirect&bpo=41056]: Invalid file descriptor values are now prevented from being passed to os.fpathconf. (discovered by Coverity)
- bpo-41056 [https://bugs.python.org/issue?@action=redirect&bpo=41056]: Fix a NULL pointer dereference within the ssl module during a MemoryError in the keylog callback. (discovered by Coverity)
- bpo-41056 [https://bugs.python.org/issue?@action=redirect&bpo=41056]: Fixed an instance where a MemoryError within the zoneinfo module might not be reported or not reported at its source. (found by Coverity)
- bpo-41048 [https://bugs.python.org/issue?@action=redirect&bpo=41048]: **`mimetypes.read_mime_types()`** function reads the rule file using UTF-8 encoding, not the locale encoding. Patch by Srinivas Reddy Thatiparthy.
- bpo-41043 [https://bugs.python.org/issue?@action=redirect&bpo=41043]: Fixed the use of **`glob()`** in the stdlib: literal part of the path is now always correctly escaped.

- bpo-41025 [https://bugs.python.org/issue?@action=redirect&bpo=41025]: Fixed an issue preventing the C implementation of **`zoneinfo.ZoneInfo`** from being subclassed.

- bpo-35018 [https://bugs.python.org/issue?@action=redirect&bpo=35018]: Add the **`xml.sax.handler.LexicalHandler`** class that is present in other SAX XML implementations.

- bpo-41002 [https://bugs.python.org/issue?@action=redirect&bpo=41002]: Improve performance of HTTPResponse.read with a given amount. Patch by Bruce Merry.

- bpo-40448 [https://bugs.python.org/issue?@action=redirect&bpo=40448]: **`ensurepip`** now disables the use of `pip` cache when installing the bundled versions of `pip` and `setuptools`. Patch by Krzysztof Konopko.

- bpo-40967 [https://bugs.python.org/issue?@action=redirect&bpo=40967]: Removed **`asyncio.Task.current_task()`** and **`asyncio.Task.all_tasks()`**. Patch contributed by Rémi Lapeyre.

- bpo-40924 [https://bugs.python.org/issue?@action=redirect&bpo=40924]: Ensure `importlib.resources.path` returns an extant path for the SourceFileLoader's resource reader. Avoids the regression identified in master while a long-term solution is devised.

- bpo-40955 [https://bugs.python.org/issue?@action=redirect&bpo=40955]: Fix a minor memory leak in **`subprocess`** module when extra_groups was specified.

- bpo-40855 [https://bugs.python.org/issue?@action=redirect&bpo=40855]: The standard deviation and variance functions in the statistics module were ignoring their mu and xbar arguments.

- bpo-40939 [https://bugs.python.org/issue?@action=redirect&bpo=40939]: Use the new PEG parser when generating the stdlib **`keyword`** module.

- bpo-23427 [https://bugs.python.org/issue?@action=redirect&bpo=23427]: Add **`sys.orig_argv`** attribute: the list of the original command line arguments passed to the Python executable.

- bpo-33689 [https://bugs.python.org/issue?@action=redirect&bpo=33689]: Ignore empty or whitespace-only lines in .pth files. This matches the documentated behavior. Before, empty lines caused the site-packages dir to appear multiple times in sys.path. By Ido Michael, contributors Malcolm Smith and Tal Einat.

- bpo-40884 [https://bugs.python.org/issue?@action=redirect&bpo=40884]: Added a `defaults` parameter to **`logging.Formatter`**, to allow specifying default values for custom fields. Patch by Asaf Alon and Bar Harel.

- bpo-40876 [https://bugs.python.org/issue?@action=redirect&bpo=40876]: Clarify error message in the **`csv`** module.

- bpo-39791 [https://bugs.python.org/issue?@action=redirect&bpo=39791]: Refresh importlib.metadata from importlib_metadata 1.6.1.

- bpo-40807 [https://bugs.python.org/issue?@action=redirect&bpo=40807]: Stop codeop._maybe_compile, used by code.InteractiveInterpreter (and IDLE). from emitting each warning three times.

- bpo-32604 [https://bugs.python.org/issue?@action=redirect&bpo=32604]: Fix reference leak in the `select` module when the module is imported in a subinterpreter.

- bpo-39791 [https://bugs.python.org/issue?@action=redirect&bpo=39791]: Built-in loaders (SourceFileLoader and ZipImporter) now supply `TraversableResources` implementations for `ResourceReader`, and the fallback function has been removed.

- bpo-39314 [https://bugs.python.org/issue?@action=redirect&bpo=39314]: `rlcompleter.Completer` and the standard Python shell now close the parenthesis for functions that take no arguments. Patch contributed by Rémi Lapeyre.

- bpo-17005 [https://bugs.python.org/issue?@action=redirect&bpo=17005]: The topological sort functionality that was introduced initially in the `functools` module has been moved to a new `graphlib` module to better accommodate the new tools and keep the original scope of the `functools` module. Patch by Pablo Galindo

- bpo-40834 [https://bugs.python.org/issue?@action=redirect&bpo=40834]: Fix truncate when sending str object with_xxsubinterpreters.channel_send.

- bpo-40755 [https://bugs.python.org/issue?@action=redirect&bpo=40755]: Add rich comparisons to collections.Counter().

- bpo-26407 [https://bugs.python.org/issue?@action=redirect&bpo=26407]: Unexpected errors in calling the `__iter__` method are no longer masked by `TypeError` in `csv.reader()`, **`csv.writer.writerow()`** and **`csv.writer.writerows()`**.

- bpo-39384 [https://bugs.python.org/issue?@action=redirect&bpo=39384]: Fixed email.contentmanager to allow set_content() to set a null string.

- bpo-40744 [https://bugs.python.org/issue?@action=redirect&bpo=40744]: The **`sqlite3`** module uses SQLite API functions that require SQLite v3.7.3 or higher. This patch removes support for older SQLite versions, and explicitly requires SQLite 3.7.3 both at build, compile and runtime. Patch by Sergey Fedoseev and Erlend E. Aasland.

- bpo-40777 [https://bugs.python.org/issue?@action=redirect&bpo=40777]: Initialize PyDateTime_IsoCalendarDateType.tp_base at run-time to avoid errors on some compilers.

- bpo-38488 [https://bugs.python.org/issue?@action=redirect&bpo=38488]: Update ensurepip to install pip 20.1.1 and setuptools 47.1.0.

- bpo-40792 [https://bugs.python.org/issue?@action=redirect&bpo=40792]: The result of **`operator.index()`** now always has exact type **`int`**. Previously, the result could have been an instance of a subclass of `int`.

- bpo-40767 [https://bugs.python.org/issue?@action=redirect&bpo=40767]: **`webbrowser`** now properly finds the default browser in pure Wayland systems by checking the WAYLAND_DISPLAY environment variable. Patch contributed by Jérémy Attali.

- bpo-40791 [https://bugs.python.org/issue?@action=redirect&bpo=40791]: **`hashlib.compare_digest()`** uses OpenSSL’s `CRYPTO_memcmp()` function when OpenSSL is available.

- bpo-40795 [https://bugs.python.org/issue?@action=redirect&bpo=40795]: `ctypes` module: If ctypes fails to convert the result of a callback or if a ctypes callback function raises an exception, sys.unraisablehook is now called with an exception set. Previously, the error was logged into stderr by `PyErr_Print()`.

- bpo-16995 [https://bugs.python.org/issue?@action=redirect&bpo=16995]: Add `base64.b32hexencode()` and `base64.b32hexdecode()` to support the Base32 Encoding with Extended Hex Alphabet.

- bpo-30008 [https://bugs.python.org/issue?@action=redirect&bpo=30008]: Fix `ssl` code to be compatible with OpenSSL 1.1.x builds that use `no-deprecated` and `--api=1.1.0`.

- bpo-30064 [https://bugs.python.org/issue?@action=redirect&bpo=30064]: Fix asyncio `loop.sock_*` race condition issue

- bpo-40759 [https://bugs.python.org/issue?@action=redirect&bpo=40759]: Deprecate the **`symbol`** module.

- bpo-40756 [https://bugs.python.org/issue?@action=redirect&bpo=40756]: The second argument (extra) of `LoggerAdapter.__init__` now defaults to None.

- bpo-37129 [https://bugs.python.org/issue?@action=redirect&bpo=37129]: Add a new `os.RWF_APPEND` flag for `os.pwritev()`.

- bpo-40737 [https://bugs.python.org/issue?@action=redirect&bpo=40737]: Fix possible reference leak for `sqlite3` initialization.

- bpo-40726 [https://bugs.python.org/issue?@action=redirect&bpo=40726]: Handle cases where the `end_lineno` is `None` on **`ast.increment_lineno()`**.

- bpo-40698 [https://bugs.python.org/issue?@action=redirect&bpo=40698]: `distutils` upload creates SHA2-256 and Blake2b-256 digests. MD5 digests is skipped if platform blocks MD5.

- bpo-40695 [https://bugs.python.org/issue?@action=redirect&bpo=40695]: **`hashlib`** no longer falls back to builtin hash implementations when OpenSSL provides a hash digest and the algorithm is blocked by security policy.

- bpo-9216 [https://bugs.python.org/issue?@action=redirect&bpo=9216]: **`hashlib.new()`** passed `usedforsecurity` to OpenSSL EVP constructor `_hashlib.new()`. test_hashlib and test_smtplib handle strict security policy better.

- bpo-40614 [https://bugs.python.org/issue?@action=redirect&bpo=40614]: **`ast.parse()`** will not parse self documenting expressions in f-strings when passed `feature_version` is less than `(3, 8)`.

- bpo-40626 [https://bugs.python.org/issue?@action=redirect&bpo=40626]: Add h5 file extension as MIME Type application/x-hdf5, as per HDF Group recommendation for HDF5 formatted data files. Patch contributed by Mark Schwab.

- bpo-25920 [https://bugs.python.org/issue?@action=redirect&bpo=25920]: On macOS, when building Python for macOS 10.4 and older, which wasn’t the case for python.org macOS installer, **`socket.getaddrinfo()`** no longer uses an internal lock to prevent race conditions when calling `getaddrinfo()` which is thread-safe since macOS 10.5. Python 3.9 requires macOS 10.6 or newer. The internal lock caused random hang on fork when another thread was calling `socket.getaddrinfo()`. The lock was also used on FreeBSD older than 5.3, OpenBSD older than 201311 and NetBSD older than 4.

- bpo-40671 [https://bugs.python.org/issue?@action=redirect&bpo=40671]: Prepare `_hashlib` for **PEP 489** [https://peps.python.org/pep-0489/] and use `PyModule_AddType()`.

- bpo-32309 [https://bugs.python.org/issue?@action=redirect&bpo=32309]: Added a new coroutine `asyncio.to_thread()`. It is mainly used for running IO-bound functions in a separate thread to avoid blocking the event loop, and essentially works as a high-level version of `run_in_executor()` that can directly take keyword arguments.

- bpo-36543 [https://bugs.python.org/issue?@action=redirect&bpo=36543]: Restored the deprecated **`xml.etree.cElementTree`** module.

- bpo-40611 [https://bugs.python.org/issue?@action=redirect&bpo=40611]: `MAP_POPULATE` constant has now been added to the list of exported `mmap` module flags.

- bpo-39881 [https://bugs.python.org/issue?@action=redirect&bpo=39881]: PEP 554 for use in the test suite. (Patch By Joannah Nanjekye)

- bpo-13097 [https://bugs.python.org/issue?@action=redirect&bpo=13097]: `ctypes` now raises an `ArgumentError` when a callback is invoked with more than 1024 arguments.

- bpo-39385 [https://bugs.python.org/issue?@action=redirect&bpo=39385]: A new test assertion contextmanager, **`unittest.assertNoLogs()`** will ensure a given block of code emits no log messages using the logging module. Contributed by Kit Yan Choi.

- bpo-23082 [https://bugs.python.org/issue?@action=redirect&bpo=23082]: Updated the error message and docs of PurePath.relative_to() to better reflect the function behaviour.

- bpo-40318 [https://bugs.python.org/issue?@action=redirect&bpo=40318]: Use SQLite3 trace v2 API, if it is available.

- bpo-40105 [https://bugs.python.org/issue?@action=redirect&bpo=40105]: ZipFile truncates files to avoid corruption when a shorter comment is provided in append (“a”) mode. Patch by Jan Mazur.

- bpo-40084 [https://bugs.python.org/issue?@action=redirect&bpo=40084]: Fix `Enum.__dir__`: dir(Enum.member) now includes attributes as well as methods.

- bpo-31122 [https://bugs.python.org/issue?@action=redirect&bpo=31122]: ssl.wrap_socket() now raises ssl.SSLEOFError rather than OSError when peer closes connection during TLS negotiation

- bpo-39728 [https://bugs.python.org/issue?@action=redirect&bpo=39728]: fix default `_missing_` so a duplicate `ValueError` is not set as the `__context__` of the original `ValueError`.

- bpo-39244 [https://bugs.python.org/issue?@action=redirect&bpo=39244]: Fixed **`multiprocessing.context.get_all_start_methods`** to properly return the default method first on macOS.

- bpo-39040 [https://bugs.python.org/issue?@action=redirect&bpo=39040]: Fix parsing of invalid mime headers parameters by collapsing whitespace between encoded words in a bare-quote-string.

- bpo-38731 [https://bugs.python.org/issue?@action=redirect&bpo=38731]: Add `--quiet` option to command-line interface of **`py_compile`**. Patch by Gregory Schevchenko.

- bpo-35714 [https://bugs.python.org/issue?@action=redirect&bpo=35714]: **`struct.error`** is now raised if there is a null character in a **`struct`** format string.

- bpo-38144 [https://bugs.python.org/issue?@action=redirect&bpo=38144]: Added the *root_dir* and *dir_fd* parameters in **`glob.glob()`**.

- bpo-26543 [https://bugs.python.org/issue?@action=redirect&bpo=26543]: Fix **`IMAP4.noop()`** when debug mode is enabled (ex: `imaplib.Debug = 3`).

- bpo-12178 [https://bugs.python.org/issue?@action=redirect&bpo=12178]: **`csv.writer()`** now correctly escapes *escapechar* when input contains *escapechar*. Patch by Catalin Iacob, Berker Peksag, and Itay Elbirt.

- bpo-36290 [https://bugs.python.org/issue?@action=redirect&bpo=36290]: AST nodes are now raising **`TypeError`** on conflicting keyword arguments. Patch contributed by Rémi Lapeyre.

- bpo-33944 [https://bugs.python.org/issue?@action=redirect&bpo=33944]: Added site.py site-packages tracing in verbose mode.

- bpo-35078 [https://bugs.python.org/issue?@action=redirect&bpo=35078]: Refactor formatweekday, formatmonthname methods in LocaleHTMLCalendar and LocaleTextCalendar classes in calendar module to call the base class methods.This enables customizable CSS classes for LocaleHTMLCalendar. Patch by Srinivas Reddy Thatiparthy

- bpo-29620 [https://bugs.python.org/issue?@action=redirect&bpo=29620]: **`assertWarns()`** no longer raises a `RuntimeException` when accessing a module's `__warningregistry__` causes importation of a new module, or when a new module is imported in another thread. Patch by Kernc.

- bpo-31844 [https://bugs.python.org/issue?@action=redirect&bpo=31844]: Remove `ParserBase.error()` method from the private and undocumented `_markupbase` module. **`html.parser.HTMLParser`** is the only subclass of `ParserBase` and its `error()` implementation was deprecated in Python 3.4 and removed in Python 3.5.

- bpo-34226 [https://bugs.python.org/issue?@action=redirect&bpo=34226]: Fix `cgi.parse_multipart` without content_length. Patch by Roger Duran

- bpo-33660 [https://bugs.python.org/issue?@action=redirect&bpo=33660]: Fix pathlib.PosixPath to resolve a relative path located on the root directory properly.

- bpo-28557 [https://bugs.python.org/issue?@action=redirect&bpo=28557]: Improve the error message for a misbehaving `rawio.readinto`

- bpo-26680 [https://bugs.python.org/issue?@action=redirect&bpo=26680]: The d.is_integer() method is added to the Decimal type, for compatibility with other number types.

- bpo-26680 [https://bugs.python.org/issue?@action=redirect&bpo=26680]: The x.is_integer() method is incorporated into the abstract types of the numeric tower, Real, Rational and Integral, with appropriate default implementations.

Documentation

- bpo-41428 [https://bugs.python.org/issue?@action=redirect&bpo=41428]: Add documentation for **PEP 604** [https://peps.python.org/pep-0604/] (Allow writing union types as `X | Y`).
- bpo-41774 [https://bugs.python.org/issue?@action=redirect&bpo=41774]: In Programming FAQ “Sequences (Tuples/Lists)” section, add “How do you remove multiple items from a list”.
- bpo-35293 [https://bugs.python.org/issue?@action=redirect&bpo=35293]: Fix RemovedInSphinx40Warning when building the documentation. Patch by Donghee Na.
- bpo-37149 [https://bugs.python.org/issue?@action=redirect&bpo=37149]: Change Shipman tkinter doc link from archive.org to TkDocs. (The doc has been removed from the NMT server.) The new link responds much faster and includes a short explanatory note.
- bpo-41726 [https://bugs.python.org/issue?@action=redirect&bpo=41726]: Update the refcounts info of `PyType_FromModuleAndSpec`.
- bpo-41624 [https://bugs.python.org/issue?@action=redirect&bpo=41624]: Fix the signature of **`typing.Coroutine`**.
- bpo-40204 [https://bugs.python.org/issue?@action=redirect&bpo=40204]: Enable Sphinx 3.2

`c_allow_pre_v3` option and disable `c_warn_on_allowed_pre_v3` option to make the documentation compatible with Sphinx 2 and Sphinx 3.
- bpo-41045 [https://bugs.python.org/issue?@action=redirect&bpo=41045]: Add documentation for debug feature of f-strings.
- bpo-41314 [https://bugs.python.org/issue?@action=redirect&bpo=41314]: Changed the release when `from __future__ import annotations` becomes the default from `4.0` to `3.10` (following a change in PEP 563).
- bpo-40979 [https://bugs.python.org/issue?@action=redirect&bpo=40979]: Refactored typing.rst, arranging more than 70 classes, functions, and decorators into new sub-sections.
- bpo-40552 [https://bugs.python.org/issue?@action=redirect&bpo=40552]: Fix in tutorial section 4.2. Code snippet is now correct.
- bpo-39883 [https://bugs.python.org/issue?@action=redirect&bpo=39883]: Make code, examples, and recipes in the Python documentation be licensed under the more permissive BSD0 license in addition to the existing Python 2.0 license.
- bpo-37703 [https://bugs.python.org/issue?@action=redirect&bpo=37703]: Updated Documentation to comprehensively elaborate on the behaviour of gather.cancel()

Tests

- bpo-41939 [https://bugs.python.org/issue?@action=redirect&bpo=41939]: Fix test_site.test_license_exists_at_url(): call `urllib.request.urlcleanup()` to reset the global `urllib.request._opener`. Patch by Victor Stinner.
- bpo-41731 [https://bugs.python.org/issue?@action=redirect&bpo=41731]: Make test_cmd_line_script pass

with option ‘-vv’.

- bpo-41602 [https://bugs.python.org/issue?@action=redirect&bpo=41602]: Add tests for SIGINT handling in the runpy module.
- bpo-41521 [https://bugs.python.org/issue?@action=redirect&bpo=41521]: **`test.support`**: Rename `blacklist` parameter of **`check__all__()`** to `not_exported`.
- bpo-41477 [https://bugs.python.org/issue?@action=redirect&bpo=41477]: Make ctypes optional in test_genericalias.
- bpo-41085 [https://bugs.python.org/issue?@action=redirect&bpo=41085]: Fix integer overflow in the **`array.array.index()`** method on 64-bit Windows for index larger than `2**31`.
- bpo-41069 [https://bugs.python.org/issue?@action=redirect&bpo=41069]: **`test.support.TESTFN`** and the current directory for tests when run via `test.regrtest` contain now non-ascii characters if possible.
- bpo-38377 [https://bugs.python.org/issue?@action=redirect&bpo=38377]: On Linux, skip tests using multiprocessing if the current user cannot create a file in `/dev/shm/` directory. Add the **`skip_if_broken_multiprocessing_synchronize()`** function to the **`test.support`** module.
- bpo-41009 [https://bugs.python.org/issue?@action=redirect&bpo=41009]: Fix use of `support.require_{linux|mac|freebsd}_version()` decorators as class decorator.
- bpo-41003 [https://bugs.python.org/issue?@action=redirect&bpo=41003]: Fix `test_copyreg` when `numpy` is installed: `test.pickletester` now saves/restores warnings filters when importing `numpy`, to ignore filters installed by `numpy`.
- bpo-40964 [https://bugs.python.org/issue?@action=redirect&bpo=40964]: Disable remote **`imaplib`** tests,

host cyrus.andrew.cmu.edu is blocking incoming connections.

- bpo-40927 [https://bugs.python.org/issue?@action=redirect&bpo=40927]: Fix test_binhex when run twice: it now uses import_fresh_module() to ensure that it raises DeprecationWarning each time.
- bpo-17258 [https://bugs.python.org/issue?@action=redirect&bpo=17258]: Skip some **`multiprocessing`** tests when MD5 hash digest is blocked.
- bpo-31904 [https://bugs.python.org/issue?@action=redirect&bpo=31904]: Increase LOOPBACK_TIMEOUT to 10 for VxWorks RTOS.
- bpo-38169 [https://bugs.python.org/issue?@action=redirect&bpo=38169]: Increase code coverage for SharedMemory and ShareableList
- bpo-34401 [https://bugs.python.org/issue?@action=redirect&bpo=34401]: Make test_gdb properly run on HP-UX. Patch by Michael Osipov.

Build

- bpo-38249 [https://bugs.python.org/issue?@action=redirect&bpo=38249]: Update **`Py_UNREACHABLE`** to use __builtin_unreachable() if only the compiler is able to use it. Patch by Donghee Na.
- bpo-41617 [https://bugs.python.org/issue?@action=redirect&bpo=41617]: Fix `pycore_bitutils.h` header file to support old clang versions: `__builtin_bswap16()` is not available in LLVM clang 3.0.
- bpo-40204 [https://bugs.python.org/issue?@action=redirect&bpo=40204]: Pin Sphinx version to 2.3.1 in `Doc/Makefile`.
- bpo-36020 [https://bugs.python.org/issue?@action=redirect&bpo=36020]: The C99 functions **`snprintf()`** and **`vsnprintf()`** are now required to build Python.

- bpo-40684 [https://bugs.python.org/issue?@action=redirect&bpo=40684]: `make install` now uses the `PLATLIBDIR` variable for the destination `lib-dynload/` directory when `./configure --with-platlibdir` is used.
- bpo-40683 [https://bugs.python.org/issue?@action=redirect&bpo=40683]: Fixed an issue where the **`zoneinfo`** module and its tests were not included when Python is installed with `make`.

Windows

- bpo-41744 [https://bugs.python.org/issue?@action=redirect&bpo=41744]: Fixes automatic import of props file when using the Nuget package.
- bpo-41627 [https://bugs.python.org/issue?@action=redirect&bpo=41627]: The user site directory for 32-bit now includes a `-32` suffix to distinguish it from the 64-bit interpreter’s directory.
- bpo-41526 [https://bugs.python.org/issue?@action=redirect&bpo=41526]: Fixed layout of final page of the installer by removing the special thanks to Mark Hammond (with his permission).
- bpo-41492 [https://bugs.python.org/issue?@action=redirect&bpo=41492]: Fixes the description that appears in UAC prompts.
- bpo-40948 [https://bugs.python.org/issue?@action=redirect&bpo=40948]: Improve post-install message to direct people to the “py” command.
- bpo-41412 [https://bugs.python.org/issue?@action=redirect&bpo=41412]: The installer will now fail to install on Windows 7 and Windows 8. Further, the UCRT dependency is now always downloaded on demand.
- bpo-40741 [https://bugs.python.org/issue?@action=redirect&bpo=40741]: Update Windows release to include SQLite 3.32.3.

- bpo-41142 [https://bugs.python.org/issue?@action=redirect&bpo=41142]: **`msilib`** now supports creating CAB files with non-ASCII file path and adding files with non-ASCII file path to them.
- bpo-41074 [https://bugs.python.org/issue?@action=redirect&bpo=41074]: Fixed support of non-ASCII names in functions **`msilib.OpenDatabase()`** and **`msilib.init_database()`** and non-ASCII SQL in method **`msilib.Database.OpenView()`**.
- bpo-41039 [https://bugs.python.org/issue?@action=redirect&bpo=41039]: Stable ABI redirection DLL (python3.dll) now uses `#pragma comment(linker)` for re-exporting.
- bpo-40164 [https://bugs.python.org/issue?@action=redirect&bpo=40164]: Updates Windows OpenSSL to 1.1.1g
- bpo-39631 [https://bugs.python.org/issue?@action=redirect&bpo=39631]: Changes the registered MIME type for `.py` files on Windows to `text/x-python` instead of `text/plain`.
- bpo-40677 [https://bugs.python.org/issue?@action=redirect&bpo=40677]: Manually define IO_REPARSE_TAG_APPEXECLINK in case some old Windows SDK doesn’t have it.
- bpo-37556 [https://bugs.python.org/issue?@action=redirect&bpo=37556]: Extend py.exe help to mention overrides via venv, shebang, environmental variables & ini files.

macOS

- bpo-41557 [https://bugs.python.org/issue?@action=redirect&bpo=41557]: Update macOS installer to use SQLite 3.33.0.
- bpo-39580 [https://bugs.python.org/issue?@action=redirect&bpo=39580]: Avoid opening Finder window if

running installer from the command line. Patch contributed by Rick Heil.

- bpo-41100 [https://bugs.python.org/issue?@action=redirect&bpo=41100]: Fix configure error when building on macOS 11. Note that the current Python release was released shortly after the first developer preview of macOS 11 (Big Sur); there are other known issues with building and running on the developer preview. Big Sur is expected to be fully supported in a future bugfix release of Python 3.8.x and with 3.9.0.
- bpo-40741 [https://bugs.python.org/issue?@action=redirect&bpo=40741]: Update macOS installer to use SQLite 3.32.3.
- bpo-41005 [https://bugs.python.org/issue?@action=redirect&bpo=41005]: fixed an XDG settings issue not allowing macos to open browser in webbrowser.py
- bpo-40741 [https://bugs.python.org/issue?@action=redirect&bpo=40741]: Update macOS installer to use SQLite 3.32.2.

IDLE

- bpo-41775 [https://bugs.python.org/issue?@action=redirect&bpo=41775]: Use ‘IDLE Shell’ as shell title
- bpo-35764 [https://bugs.python.org/issue?@action=redirect&bpo=35764]: Rewrite the Calltips doc section.
- bpo-40181 [https://bugs.python.org/issue?@action=redirect&bpo=40181]: In calltips, stop reminding that ‘/’ marks the end of positional-only arguments.
- bpo-41468 [https://bugs.python.org/issue?@action=redirect&bpo=41468]: Improve IDLE run crash error message (which users should never see).
- bpo-41373 [https://bugs.python.org/issue?@action=redirect&bpo=41373]: Save files loaded with no line ending, as when blank, or different line endings, by

setting its line ending to the system default. Fix regression in 3.8.4 and 3.9.0b4.

- bpo-41300 [https://bugs.python.org/issue?@action=redirect&bpo=41300]: Save files with non-ascii chars. Fix regression released in 3.9.0b4 and 3.8.4.
- bpo-37765 [https://bugs.python.org/issue?@action=redirect&bpo=37765]: Add keywords to module name completion list. Rewrite Completions section of IDLE doc.
- bpo-41152 [https://bugs.python.org/issue?@action=redirect&bpo=41152]: The encoding of `stdin`, `stdout` and `stderr` in IDLE is now always UTF-8.
- bpo-41144 [https://bugs.python.org/issue?@action=redirect&bpo=41144]: Make Open Module open a special module such as os.path.
- bpo-39885 [https://bugs.python.org/issue?@action=redirect&bpo=39885]: Make context menu Cut and Copy work again when right-clicking within a selection.
- bpo-40723 [https://bugs.python.org/issue?@action=redirect&bpo=40723]: Make test_idle pass when run after import.

C API

- bpo-41936 [https://bugs.python.org/issue?@action=redirect&bpo=41936]: Removed undocumented macros `Py_ALLOW_RECURSION` and `Py_END_ALLOW_RECURSION` and the `recursion_critical` field of the **`PyInterpreterState`** structure.
- bpo-41692 [https://bugs.python.org/issue?@action=redirect&bpo=41692]: The `PyUnicode_InternImmortal()` function is now deprecated and will be removed in Python 3.12: use **`PyUnicode_InternInPlace()`** instead. Patch by Victor Stinner.
- bpo-41842 [https://bugs.python.org/issue?@action=redirect&bpo=41842]: Add **`PyCodec_Unregister()`**

function to unregister a codec search function.

- bpo-41834 [https://bugs.python.org/issue?@action=redirect&bpo=41834]: Remove the `_Py_CheckRecursionLimit` variable: it has been replaced by `ceval.recursion_limit` of the **`PyInterpreterState`** structure. Patch by Victor Stinner.
- bpo-41689 [https://bugs.python.org/issue?@action=redirect&bpo=41689]: Types created with **`PyType_FromSpec()`** now make any signature in their `tp_doc` slot accessible from `__text_signature__`.
- bpo-41524 [https://bugs.python.org/issue?@action=redirect&bpo=41524]: Fix bug in **`PyOS_mystrnicmp()`** and **`PyOS_mystricmp()`** that incremented pointers beyond the end of a string.
- bpo-41324 [https://bugs.python.org/issue?@action=redirect&bpo=41324]: Add a minimal decimal capsule API. The API supports fast conversions between Decimals up to 38 digits and their triple representation as a C struct.
- bpo-30155 [https://bugs.python.org/issue?@action=redirect&bpo=30155]: Add **`PyDateTime_DATE_GET_TZINFO()`** and **`PyDateTime_TIME_GET_TZINFO()`** macros for accessing the `tzinfo` attributes of **`datetime.datetime`** and **`datetime.time`** objects.
- bpo-40170 [https://bugs.python.org/issue?@action=redirect&bpo=40170]: Revert **`PyType_HasFeature()`** change: it reads again directly the **`PyTypeObject.tp_flags`** member when the limited C API is not used, rather than always calling **`PyType_GetFlags()`** which hides implementation details.
- bpo-41123 [https://bugs.python.org/issue?@action=redirect&bpo=41123]: Remove `PyUnicode_AsUnicodeCopy`.
- bpo-41123 [https://bugs.python.org/issue?@action=redirect&bpo=41123]: Removed `PyLong_FromUnicode()`.
- bpo-41123 [https://bugs.python.org/issue?@action=redirect&bpo=41123]: Removed `PyUnicode_GetMax()`.

- bpo-41123 [https://bugs.python.org/issue?@action=redirect&bpo=41123]: Removed `Py_UNICODE_str*` functions manipulating `Py_UNICODE*` strings.
- bpo-41103 [https://bugs.python.org/issue?@action=redirect&bpo=41103]: `PyObject_AsCharBuffer()`, `PyObject_AsReadBuffer()`, `PyObject_CheckReadBuffer()`, and `PyObject_AsWriteBuffer()` are removed. Please migrate to new buffer protocol; **`PyObject_GetBuffer()`** and **`PyBuffer_Release()`**.
- bpo-36346 [https://bugs.python.org/issue?@action=redirect&bpo=36346]: Raises DeprecationWarning for `PyUnicode_FromUnicode(NULL, size)` and `PyUnicode_FromStringAndSize(NULL, size)` with `size > 0`.
- bpo-36346 [https://bugs.python.org/issue?@action=redirect&bpo=36346]: Mark `Py_UNICODE_COPY`, `Py_UNICODE_FILL`, `PyUnicode_WSTR_LENGTH`, `PyUnicode_FromUnicode`, `PyUnicode_AsUnicode`, and `PyUnicode_AsUnicodeAndSize` as deprecated in C. Remove `Py_UNICODE_MATCH` which was deprecated and broken since Python 3.3.
- bpo-40989 [https://bugs.python.org/issue?@action=redirect&bpo=40989]: The **`PyObject_INIT()`** and **`PyObject_INIT_VAR()`** macros become aliases to, respectively, **`PyObject_Init()`** and **`PyObject_InitVar()`** functions.
- bpo-36020 [https://bugs.python.org/issue?@action=redirect&bpo=36020]: On Windows, `#include "pyerrors.h"` no longer defines `snprintf` and `vsnprintf` macros.
- bpo-40943 [https://bugs.python.org/issue?@action=redirect&bpo=40943]: The `PY_SSIZE_T_CLEAN` macro must now be defined to use **`PyArg_ParseTuple()`** and **`Py_BuildValue()`** formats which use `#`: `es#`, `et#`, `s#`, `u#`, `y#`, `z#`, `U#` and `Z#`. See Parsing arguments and building values and the **PEP 353** [https://peps.python.org/pep-0353/].

- bpo-40910 [https://bugs.python.org/issue?@action=redirect&bpo=40910]: Export explicitly the **`Py_GetArgcArgv()`** function to the C API and document the function. Previously, it was exported implicitly which no longer works since Python is built with `-fvisibility=hidden`.
- bpo-40724 [https://bugs.python.org/issue?@action=redirect&bpo=40724]: Allow defining buffer slots in type specs.
- bpo-40679 [https://bugs.python.org/issue?@action=redirect&bpo=40679]: Fix a `_PyEval_EvalCode()` crash if *qualname* argument is NULL.
- bpo-40839 [https://bugs.python.org/issue?@action=redirect&bpo=40839]: Calling **`PyDict_GetItem()`** without GIL held had been allowed for historical reason. It is no longer allowed.
- bpo-40826 [https://bugs.python.org/issue?@action=redirect&bpo=40826]: **`PyOS_InterruptOccurred()`** now fails with a fatal error if it is called with the GIL released.
- bpo-40792 [https://bugs.python.org/issue?@action=redirect&bpo=40792]: The result of **`PyNumber_Index()`** now always has exact type **`int`**. Previously, the result could have been an instance of a subclass of `int`.
- bpo-39573 [https://bugs.python.org/issue?@action=redirect&bpo=39573]: Convert **`Py_REFCNT()`** and **`Py_SIZE()`** macros to static inline functions. They cannot be used as l-value anymore: use **`Py_SET_REFCNT()`** and **`Py_SET_SIZE()`** to set an object reference count and size. This change is backward incompatible on purpose, to prepare the C API for an opaque **`PyObject`** structure.
- bpo-40703 [https://bugs.python.org/issue?@action=redirect&bpo=40703]: The PyType_FromSpec*() functions no longer overwrite the type's “__module__” attribute if it is set via “Py_tp_members” or “Py_tp_getset”.

- bpo-39583 [https://bugs.python.org/issue?@action=redirect&bpo=39583]: Remove superfluous “extern C” declarations from `Include/cpython/*.h`.

Python 3.9.0 beta 1

Release date: 2020-05-19

Security

- bpo-40501 [https://bugs.python.org/issue?@action=redirect&bpo=40501]: **`uuid`** no longer uses **`ctypes`** to load `libuuid` or `rpcrt4.dll` at runtime.

Core and Builtins

- bpo-40663 [https://bugs.python.org/issue?@action=redirect&bpo=40663]: Correctly generate annotations where parentheses are omitted but required (e.g: `Type[(str, int, *other))]`.
- bpo-40596 [https://bugs.python.org/issue?@action=redirect&bpo=40596]: Fixed **`str.isidentifier()`** for non-canonicalized strings containing non-BMP characters on Windows.
- bpo-40593 [https://bugs.python.org/issue?@action=redirect&bpo=40593]: Improved syntax errors for invalid characters in source code.
- bpo-40585 [https://bugs.python.org/issue?@action=redirect&bpo=40585]: Fixed a bug when using **`codeop.compile_command()`** that was causing exceptions to be swallowed with the new parser. Patch by Pablo Galindo
- bpo-40566 [https://bugs.python.org/issue?@action=redirect&bpo=40566]: Apply **PEP 573** [https://peps.python.org/pep-0573/] to **`abc`**.

- bpo-40502 [https://bugs.python.org/issue?@action=redirect&bpo=40502]: Initialize `n->n_col_offset`. (Patch by Joannah Nanjekye)
- bpo-40527 [https://bugs.python.org/issue?@action=redirect&bpo=40527]: Fix command line argument parsing: no longer write errors multiple times into stderr.
- bpo-1635741 [https://bugs.python.org/issue?@action=redirect&bpo=1635741]: Port **`errno`** to multiphase initialization (**PEP 489** [https://peps.python.org/pep-0489/]).
- bpo-40523 [https://bugs.python.org/issue?@action=redirect&bpo=40523]: Add pass-throughs for **`hash()`** and **`reversed()`** to **`weakref.proxy`** objects. Patch by Pablo Galindo.
- bpo-1635741 [https://bugs.python.org/issue?@action=redirect&bpo=1635741]: Port **`syslog`** to multiphase initialization (**PEP 489** [https://peps.python.org/pep-0489/]).
- bpo-40246 [https://bugs.python.org/issue?@action=redirect&bpo=40246]: Reporting a specialised error message for invalid string prefixes, which was introduced in bpo-40246 [https://bugs.python.org/issue?@action=redirect&bpo=40246], is being reverted due to backwards compatibility concerns for strings that immediately follow a reserved keyword without whitespace between them. Constructs like `bg="#d00" if clear else"#fca"` were failing to parse, which is not an acceptable breakage on such short notice.
- bpo-40417 [https://bugs.python.org/issue?@action=redirect&bpo=40417]: Fix imp module deprecation warning when PyImport_ReloadModule is called. Patch by Robert Rouhani.
- bpo-40408 [https://bugs.python.org/issue?@action=redirect&bpo=40408]: Fixed support of nested type variables in GenericAlias (e.g. `list[list[T]]`).
- bpo-1635741 [https://bugs.python.org/issue?@action=redirect&bpo=1635741]: Port _stat module to

multiphase initialization (**PEP 489** [https://peps.python.org/pep-0489/]).

- bpo-29587 [https://bugs.python.org/issue?@action=redirect&bpo=29587]: Enable implicit exception chaining when calling **`generator.throw()`**.
- bpo-40328 [https://bugs.python.org/issue?@action=redirect&bpo=40328]: Add tools for generating mappings headers for CJKCodecs.
- bpo-40228 [https://bugs.python.org/issue?@action=redirect&bpo=40228]: Setting frame.f_lineno is now robust w.r.t. changes in the source-to-bytecode compiler
- bpo-38880 [https://bugs.python.org/issue?@action=redirect&bpo=38880]: Added the ability to list interpreters associated with channel ends in the internal subinterpreters module.
- bpo-37986 [https://bugs.python.org/issue?@action=redirect&bpo=37986]: Improve performance of **`PyLong_FromDouble()`** for values that fit into long.

Library

- bpo-40662 [https://bugs.python.org/issue?@action=redirect&bpo=40662]: Fixed **`ast.get_source_segment()`** for ast nodes that have incomplete location information. Patch by Irit Katriel.
- bpo-40665 [https://bugs.python.org/issue?@action=redirect&bpo=40665]: Convert **`bisect`** to use Argument Clinic.
- bpo-40536 [https://bugs.python.org/issue?@action=redirect&bpo=40536]: Added the **`available_timezones()`** function to the **`zoneinfo`** module. Patch by Paul Ganssle.
- bpo-40645 [https://bugs.python.org/issue?@action=redirect&bpo=40645]: The **`hmac.HMAC`** exposes internal implementation details. The attributes `digest_cons`, `inner`, and `outer` are deprecated and will be removed in the future.

- bpo-40645 [https://bugs.python.org/issue?@action=redirect&bpo=40645]: The internal module `_hashlib` wraps and exposes OpenSSL's HMAC API. The new code will be used in Python 3.10 after the internal implementation details of the pure Python HMAC module are no longer part of the public API.
- bpo-40637 [https://bugs.python.org/issue?@action=redirect&bpo=40637]: Builtin hash modules can now be disabled or selectively enabled with `configure --with-builtin-hashlib-hashes=sha3,blake1` or `--without-builtin-hashlib-hashes`.
- bpo-37630 [https://bugs.python.org/issue?@action=redirect&bpo=37630]: The **`hashlib`** module can now use SHA3 hashes and SHAKE XOF from OpenSSL when available.
- bpo-40479 [https://bugs.python.org/issue?@action=redirect&bpo=40479]: The **`hashlib`** now compiles with OpenSSL 3.0.0-alpha2.
- bpo-40257 [https://bugs.python.org/issue?@action=redirect&bpo=40257]: Revert changes to **`inspect.getdoc()`**.
- bpo-40607 [https://bugs.python.org/issue?@action=redirect&bpo=40607]: When cancelling a task due to timeout, **`asyncio.wait_for()`** will now propagate the exception if an error happens during cancellation. Patch by Roman Skurikhin.
- bpo-40612 [https://bugs.python.org/issue?@action=redirect&bpo=40612]: Fix edge cases in SyntaxError formatting. If the offset is <= 0, no caret is printed. If the offset is > line length, the caret is printed pointing just after the last character.
- bpo-40597 [https://bugs.python.org/issue?@action=redirect&bpo=40597]: If text content lines are longer than policy.max_line_length, always use a content-encoding to make sure they are wrapped.

- bpo-40571 [https://bugs.python.org/issue?@action=redirect&bpo=40571]: Added functools.cache() as a simpler, more discoverable way to access the unbounded cache variant of lru_cache(maxsize=None).
- bpo-40503 [https://bugs.python.org/issue?@action=redirect&bpo=40503]: **PEP 615** [https://peps.python.org/pep-0615/], the **`zoneinfo`** module. Adds support for the IANA time zone database.
- bpo-40397 [https://bugs.python.org/issue?@action=redirect&bpo=40397]: Removed attributes `__args__` and `__parameters__` from special generic aliases like `typing.List` (not subscripted).
- bpo-40549 [https://bugs.python.org/issue?@action=redirect&bpo=40549]: Convert posixmodule.c (“posix” or “nt” module) to the multiphase initialization (PEP 489).
- bpo-31033 [https://bugs.python.org/issue?@action=redirect&bpo=31033]: Add a `msg` argument to **`Future.cancel()`** and **`Task.cancel()`**.
- bpo-40541 [https://bugs.python.org/issue?@action=redirect&bpo=40541]: Added an optional *counts* parameter to random.sample().
- bpo-40515 [https://bugs.python.org/issue?@action=redirect&bpo=40515]: The **`ssl`** and **`hashlib`** modules now actively check that OpenSSL is build with thread support. Python 3.7.0 made thread support mandatory and no longer works safely with a no-thread builds.
- bpo-31033 [https://bugs.python.org/issue?@action=redirect&bpo=31033]: When a **`asyncio.Task`** is cancelled, the exception traceback now chains all the way back to where the task was first interrupted.
- bpo-40504 [https://bugs.python.org/issue?@action=redirect&bpo=40504]: **`functools.lru_cache()`** objects can now be the targets of weakrefs.
- bpo-40559 [https://bugs.python.org/issue?@action=redirect&bpo=40559]: Fix possible memory leak in the

C implementation of **`asyncio.Task`**.

- bpo-40480 [https://bugs.python.org/issue?@action=redirect&bpo=40480]: `fnmatch.fnmatch()` could take exponential time in the presence of multiple `*` pattern characters. This was repaired by generating more elaborate regular expressions to avoid futile backtracking.
- bpo-40495 [https://bugs.python.org/issue?@action=redirect&bpo=40495]: **`compileall`** is now able to use hardlinks to prevent duplicates in a case when `.pyc` files for different optimization levels have the same content.
- bpo-40457 [https://bugs.python.org/issue?@action=redirect&bpo=40457]: The ssl module now support OpenSSL builds without TLS 1.0 and 1.1 methods.
- bpo-40355 [https://bugs.python.org/issue?@action=redirect&bpo=40355]: Improve error reporting in **`ast.literal_eval()`** in the presence of malformed **`ast.Dict`** nodes instead of silently ignoring any non-conforming elements. Patch by Curtis Bucher.
- bpo-40465 [https://bugs.python.org/issue?@action=redirect&bpo=40465]: Deprecated the optional *random* argument to *random.shuffle()*.
- bpo-40459 [https://bugs.python.org/issue?@action=redirect&bpo=40459]: **`platform.win32_ver()`** now produces correct *ptype* strings instead of empty strings.
- bpo-39435 [https://bugs.python.org/issue?@action=redirect&bpo=39435]: The first argument of **`pickle.loads()`** is now positional-only.
- bpo-39305 [https://bugs.python.org/issue?@action=redirect&bpo=39305]: Update **`nntplib`** to merge **`nntplib.NNTP`** and **`nntplib._NNTPBase`**. Patch by Donghee Na.
- bpo-32494 [https://bugs.python.org/issue?@action=redirect&bpo=32494]: Update **`dbm.gnu`** to use gdbm_count if possible when calling **`len()`**. Patch by Donghee Na.

- bpo-40453 [https://bugs.python.org/issue?@action=redirect&bpo=40453]: Add `isolated=True` keyword-only parameter to `_xxsubinterpreters.create()`. An isolated subinterpreter cannot spawn threads, spawn a child process or call `os.fork()`.
- bpo-40286 [https://bugs.python.org/issue?@action=redirect&bpo=40286]: Remove `_random.Random.randbytes()`: the C implementation of `randbytes()`. Implement the method in Python to ease subclassing: `randbytes()` now directly reuses `getrandbits()`.
- bpo-40394 [https://bugs.python.org/issue?@action=redirect&bpo=40394]: Added default arguments to **`difflib.SequenceMatcher.find_longest_match()`**.
- bpo-39995 [https://bugs.python.org/issue?@action=redirect&bpo=39995]: Fix a race condition in concurrent.futures._ThreadWakeup: access to _ThreadWakeup is now protected with the shutdown lock.
- bpo-30966 [https://bugs.python.org/issue?@action=redirect&bpo=30966]: `Process.shutdown(wait=True)` of **`concurrent.futures`** now closes explicitly the result queue.
- bpo-30966 [https://bugs.python.org/issue?@action=redirect&bpo=30966]: Add a new **`close()`** method to the **`SimpleQueue`** class to explicitly close the queue.
- bpo-39966 [https://bugs.python.org/issue?@action=redirect&bpo=39966]: Revert bpo-25597 [https://bugs.python.org/issue?@action=redirect&bpo=25597]. **`unittest.mock.MagicMock`** with wraps' set uses default return values for magic methods.
- bpo-39791 [https://bugs.python.org/issue?@action=redirect&bpo=39791]: Added `files()` function to importlib.resources with support for subdirectories in package data, matching backport in importlib_resources 1.5.

- bpo-40375 [https://bugs.python.org/issue?@action=redirect&bpo=40375]: **`imaplib.IMAP4.unselect()`** is added. Patch by Donghee Na.
- bpo-40389 [https://bugs.python.org/issue?@action=redirect&bpo=40389]: `repr()` now returns `typing.Optional[T]` when called for `typing.Union` of two types, one of which is `NoneType`.
- bpo-40291 [https://bugs.python.org/issue?@action=redirect&bpo=40291]: Add support for CAN_J1939 sockets (available on Linux 5.4+)
- bpo-40273 [https://bugs.python.org/issue?@action=redirect&bpo=40273]: **`types.MappingProxyType`** is now reversible.
- bpo-39075 [https://bugs.python.org/issue?@action=redirect&bpo=39075]: The repr for **`types.SimpleNamespace`** is now insertion ordered rather than alphabetical.
- bpo-40192 [https://bugs.python.org/issue?@action=redirect&bpo=40192]: On AIX, **`thread_time()`** is now implemented with `thread_cputime()` which has nanosecond resolution, rather than `clock_gettime(CLOCK_THREAD_CPUTIME_ID)` which has a resolution of 10 milliseconds. Patch by Batuhan Taskaya.
- bpo-40025 [https://bugs.python.org/issue?@action=redirect&bpo=40025]: Raise TypeError when _generate_next_value_ is defined after members. Patch by Ethan Onstott.
- bpo-39058 [https://bugs.python.org/issue?@action=redirect&bpo=39058]: In the argparse module, the repr for Namespace() and other argument holders now displayed in the order attributes were added. Formerly, it displayed in alphabetical order even though argument order is preserved the user visible parts of the module.
- bpo-24416 [https://bugs.python.org/issue?@action=redirect&bpo=24416]: The `isocalendar()` methods of

`datetime.date` and `datetime.datetime` now return a named tuple instead of a `tuple`.

Documentation

- bpo-34790 [https://bugs.python.org/issue?@action=redirect&bpo=34790]: Add version of removal for explicit passing of coros to `asyncio.wait()`’s documentation
- bpo-40561 [https://bugs.python.org/issue?@action=redirect&bpo=40561]: Provide docstrings for webbrowser open functions.
- bpo-40499 [https://bugs.python.org/issue?@action=redirect&bpo=40499]: Mention that `asyncio.wait()` requires a non-empty set of awaitables.
- bpo-39705 [https://bugs.python.org/issue?@action=redirect&bpo=39705]: Tutorial example for sorted() in the Loop Techniques section is given a better explanation. Also a new example is included to explain sorted()’s basic behavior.
- bpo-39435 [https://bugs.python.org/issue?@action=redirect&bpo=39435]: Fix an incorrect signature for `pickle.loads()` in the docs

Tests

- bpo-40055 [https://bugs.python.org/issue?@action=redirect&bpo=40055]: distutils.tests now saves/restores warnings filters to leave them unchanged. Importing tests imports docutils which imports pkg_resources which adds a warnings filter.
- bpo-40436 [https://bugs.python.org/issue?@action=redirect&bpo=40436]: test_gdb and test.pythoninfo now check gdb command exit code.

Build

- bpo-40653 [https://bugs.python.org/issue?@action=redirect&bpo=40653]: Move _dirnameW out of HAVE_SYMLINK to fix a potential compiling issue.
- bpo-40514 [https://bugs.python.org/issue?@action=redirect&bpo=40514]: Add `--with-experimental-isolated-subinterpreters` build option to `configure`: better isolate subinterpreters, experimental build mode.

Windows

- bpo-40650 [https://bugs.python.org/issue?@action=redirect&bpo=40650]: Include winsock2.h in pytime.c for timeval.
- bpo-40458 [https://bugs.python.org/issue?@action=redirect&bpo=40458]: Increase reserved stack space to prevent overflow crash on Windows.
- bpo-39148 [https://bugs.python.org/issue?@action=redirect&bpo=39148]: Add IPv6 support to **`asyncio`** datagram endpoints in ProactorEventLoop. Change the raised exception for unknown address families to ValueError as it’s not coming from Windows API.

macOS

- bpo-34956 [https://bugs.python.org/issue?@action=redirect&bpo=34956]: When building Python on macOS from source, `_tkinter` now links with non-system Tcl and Tk frameworks if they are installed in `/Library/Frameworks`, as had been the case on older releases of macOS. If a macOS SDK is explicitly configured, by using `--enable-universalsdk=` or `-isysroot`, only the SDK itself is searched. The default behavior can

still be overridden with `--with-tcltk-includes` and `--with-tcltk-libs`.
- bpo-35569 [https://bugs.python.org/issue?@action=redirect&bpo=35569]: Expose RFC 3542 IPv6 socket options.

Tools/Demos

- bpo-40479 [https://bugs.python.org/issue?@action=redirect&bpo=40479]: Update multissltest helper to test with latest OpenSSL 1.0.2, 1.1.0, 1.1.1, and 3.0.0-alpha.
- bpo-40431 [https://bugs.python.org/issue?@action=redirect&bpo=40431]: Fix a syntax typo in `turtledemo` that now raises a `SyntaxError`.
- bpo-40163 [https://bugs.python.org/issue?@action=redirect&bpo=40163]: Fix multissltest tool. OpenSSL has changed download URL for old releases. The multissltest tool now tries to download from current and old download URLs.

C API

- bpo-39465 [https://bugs.python.org/issue?@action=redirect&bpo=39465]: Remove the `_PyUnicode_ClearStaticStrings()` function from the C API.
- bpo-38787 [https://bugs.python.org/issue?@action=redirect&bpo=38787]: Add PyCFunction_CheckExact() macro for exact type checks now that we allow subtypes of PyCFunction, as well as PyCMethod_CheckExact() and PyCMethod_Check() for the new PyCMethod subtype.
- bpo-40545 [https://bugs.python.org/issue?@action=redirect&bpo=40545]: Declare `_PyErr_GetTopmostException()` with `PyAPI_FUNC()` to properly export the function in the C API. The function remains private (`_Py`) prefix.

- bpo-40412 [https://bugs.python.org/issue?@action=redirect&bpo=40412]: Nullify inittab_copy during finalization, preventing future interpreter initializations in an embedded situation from crashing. Patch by Gregory Szorc.
- bpo-40429 [https://bugs.python.org/issue?@action=redirect&bpo=40429]: The `PyThreadState_GetFrame()` function now returns a strong reference to the frame.
- bpo-40428 [https://bugs.python.org/issue?@action=redirect&bpo=40428]: Remove the following functions from the C API. Call `PyGC_Collect()` explicitly to free all free lists.
  - `PyAsyncGen_ClearFreeLists()`
  - `PyContext_ClearFreeList()`
  - `PyDict_ClearFreeList()`
  - `PyFloat_ClearFreeList()`
  - `PyFrame_ClearFreeList()`
  - `PyList_ClearFreeList()`
  - `PySet_ClearFreeList()`
  - `PyTuple_ClearFreeList()`
- bpo-40421 [https://bugs.python.org/issue?@action=redirect&bpo=40421]: New `PyFrame_GetBack()` function: get the frame next outer frame.
- bpo-40421 [https://bugs.python.org/issue?@action=redirect&bpo=40421]: New `PyFrame_GetCode()` function: return a borrowed reference to the frame code.
- bpo-40217 [https://bugs.python.org/issue?@action=redirect&bpo=40217]: Ensure that instances of types created with `PyType_FromSpecWithBases()` will visit its class object when traversing references in the garbage collector (implemented as an extension of the provided `tp_traverse`). Patch by Pablo Galindo.
- bpo-38787 [https://bugs.python.org/issue?@action=redirect&bpo=38787]: Module C state is now accessible from C-defined heap type methods (**PEP 573**

[https://peps.python.org/pep-0573/]). Patch by Marcel Plch and Petr Viktorin.

Python 3.9.0 alpha 6

Release date: 2020-04-27

Security

- bpo-40121 [https://bugs.python.org/issue?@action=redirect&bpo=40121]: Fixes audit events raised on creating a new socket.
- bpo-39073 [https://bugs.python.org/issue?@action=redirect&bpo=39073]: Disallow CR or LF in email.headerregistry.Address arguments to guard against header injection attacks.
- bpo-39503 [https://bugs.python.org/issue?@action=redirect&bpo=39503]: **CVE 2020-8492** [https://www.cve.org/CVERecord?id=CVE-2020-8492]: The `AbstractBasicAuthHandler` class of the `urllib.request` module uses an inefficient regular expression which can be exploited by an attacker to cause a denial of service. Fix the regex to prevent the catastrophic backtracking. Vulnerability reported by Ben Caller and Matt Schwager.

Core and Builtins

- bpo-40313 [https://bugs.python.org/issue?@action=redirect&bpo=40313]: Improve the performance of bytes.hex().
- bpo-40334 [https://bugs.python.org/issue?@action=redirect&bpo=40334]: Switch to a new parser, based on PEG. For more details see PEP 617. To temporarily switch back to the old parser, use `-X oldparser` or `PYTHONOLDPARSER=1`. In Python 3.10 we will remove the old

parser completely, including the `parser` module (already deprecated) and anything that depends on it.

- bpo-40267 [https://bugs.python.org/issue?@action=redirect&bpo=40267]: Fix the tokenizer to display the correct error message, when there is a `SyntaxError` on the last input character and no newline follows. It used to be `unexpected EOF while parsing`, while it should be `invalid syntax`.
- bpo-39522 [https://bugs.python.org/issue?@action=redirect&bpo=39522]: Correctly unparse explicit `u` prefix for strings when postponed evaluation for annotations activated. Patch by Batuhan Taskaya.
- bpo-40246 [https://bugs.python.org/issue?@action=redirect&bpo=40246]: Report a specialized error message, `invalid string prefix`, when the tokenizer encounters a string with an invalid prefix.
- bpo-40082 [https://bugs.python.org/issue?@action=redirect&bpo=40082]: Fix the signal handler: it now always uses the main interpreter, rather than trying to get the current Python thread state.
- bpo-37388 [https://bugs.python.org/issue?@action=redirect&bpo=37388]: str.encode() and str.decode() no longer check the encoding and errors in development mode or in debug mode during Python finalization. The codecs machinery can no longer work on very late calls to str.encode() and str.decode().
- bpo-40077 [https://bugs.python.org/issue?@action=redirect&bpo=40077]: Fix possible refleaks in **`_json`**, memo of PyScannerObject should be traversed.
- bpo-37207 [https://bugs.python.org/issue?@action=redirect&bpo=37207]: Speed up calls to `dict()` by using the **PEP 590** [https://peps.python.org/pep-0590/] `vectorcall` calling convention.
- bpo-40141 [https://bugs.python.org/issue?@action=redirect&bpo=40141]: Add column and line information to `ast.keyword` nodes. Patch by Pablo Galindo.

- bpo-1635741 [https://bugs.python.org/issue?@action=redirect&bpo=1635741]: Port **`resource`** to multiphase initialization (**PEP 489** [https://peps.python.org/pep-0489/]).
- bpo-1635741 [https://bugs.python.org/issue?@action=redirect&bpo=1635741]: Port **`math`** to multiphase initialization (**PEP 489** [https://peps.python.org/pep-0489/]).
- bpo-1635741 [https://bugs.python.org/issue?@action=redirect&bpo=1635741]: Port _uuid module to multiphase initialization (**PEP 489** [https://peps.python.org/pep-0489/]).
- bpo-40077 [https://bugs.python.org/issue?@action=redirect&bpo=40077]: Convert json module to use **`PyType_FromSpec()`**.
- bpo-40067 [https://bugs.python.org/issue?@action=redirect&bpo=40067]: Improve the error message for multiple star expressions in an assignment. Patch by Furkan Onder
- bpo-1635741 [https://bugs.python.org/issue?@action=redirect&bpo=1635741]: Port _functools module to multiphase initialization (PEP 489). Patch by Paulo Henrique Silva.
- bpo-1635741 [https://bugs.python.org/issue?@action=redirect&bpo=1635741]: Port operator module to multiphase initialization (PEP 489). Patch by Paulo Henrique Silva.
- bpo-20526 [https://bugs.python.org/issue?@action=redirect&bpo=20526]: Fix **`PyThreadState_Clear()`**. `PyThreadState.frame` is a borrowed reference, not a strong reference: `PyThreadState_Clear()` must not call `Py_CLEAR(tstate->frame)`.
- bpo-1635741 [https://bugs.python.org/issue?@action=redirect&bpo=1635741]: Port time module to multiphase initialization (**PEP 489** [https://peps.python.org/pep-0489/]). Patch by Paulo Henrique Silva.

- bpo-1635741 [https://bugs.python.org/issue?@action=redirect&bpo=1635741]: Port _weakref extension module to multiphase initialization (**PEP 489** [https://peps.python.org/pep-0489/]).
- bpo-40020 [https://bugs.python.org/issue?@action=redirect&bpo=40020]: Fix a leak and subsequent crash in parsetok.c caused by realloc misuse on a rare codepath.
- bpo-39939 [https://bugs.python.org/issue?@action=redirect&bpo=39939]: Added str.removeprefix and str.removesuffix methods and corresponding bytes, bytearray, and collections.UserString methods to remove affixes from a string if present. See **PEP 616** [https://peps.python.org/pep-0616/] for a full description. Patch by Dennis Sweeney.
- bpo-39481 [https://bugs.python.org/issue?@action=redirect&bpo=39481]: Implement PEP 585. This supports list[int], tuple[str, ...] etc.
- bpo-32894 [https://bugs.python.org/issue?@action=redirect&bpo=32894]: Support unparsing of infinity numbers in postponed annotations. Patch by Batuhan Taşkaya.
- bpo-37207 [https://bugs.python.org/issue?@action=redirect&bpo=37207]: Speed up calls to `list()` by using the **PEP 590** [https://peps.python.org/pep-0590/] `vectorcall` calling convention. Patch by Mark Shannon.

Library

- bpo-40398 [https://bugs.python.org/issue?@action=redirect&bpo=40398]: **`typing.get_args()`** now always returns an empty tuple for special generic aliases.

- bpo-40396 [https://bugs.python.org/issue?@action=redirect&bpo=40396]: Functions **`typing.get_origin()`**,

`typing.get_args()` and **`typing.get_type_hints()`** support now generic aliases like `list[int]`.

- bpo-38061 [https://bugs.python.org/issue?@action=redirect&bpo=38061]: Optimize the **`subprocess`** module on FreeBSD using `closefrom()`. A single `close(fd)` syscall is cheap, but when `sysconf(_SC_OPEN_MAX)` is high, the loop calling `close(fd)` on each file descriptor can take several milliseconds.

  The workaround on FreeBSD to improve performance was to load and mount the fdescfs kernel module, but this is not enabled by default.

  Initial patch by Ed Maste (emaste), Conrad Meyer (cem), Kyle Evans (kevans) and Kubilay Kocak (koobs): https://bugs.freebsd.org/bugzilla/show_bug.cgi?id=242274

- bpo-38061 [https://bugs.python.org/issue?@action=redirect&bpo=38061]: On FreeBSD, `os.closerange(fd_low, fd_high)` now calls `closefrom(fd_low)` if *fd_high* is greater than or equal to `sysconf(_SC_OPEN_MAX)`.

  Initial patch by Ed Maste (emaste), Conrad Meyer (cem), Kyle Evans (kevans) and Kubilay Kocak (koobs): https://bugs.freebsd.org/bugzilla/show_bug.cgi?id=242274

- bpo-40360 [https://bugs.python.org/issue?@action=redirect&bpo=40360]: The **`lib2to3`** module is pending deprecation due to **PEP 617** [https://peps.python.org/pep-0617/].

- bpo-40138 [https://bugs.python.org/issue?@action=redirect&bpo=40138]: Fix the Windows implementation of **`os.waitpid()`** for exit code larger than

`INT_MAX >> 8`. The exit status is now interpreted as an unsigned number.

- bpo-39942 [https://bugs.python.org/issue?@action=redirect&bpo=39942]: Set "__main__" as the default module name when "__name__" is missing in **`typing.TypeVar`**. Patch by Weipeng Hong.

- bpo-40275 [https://bugs.python.org/issue?@action=redirect&bpo=40275]: The **`logging`** package is now imported lazily in **`unittest`** only when the **`assertLogs()`** assertion is used.

- bpo-40275 [https://bugs.python.org/issue?@action=redirect&bpo=40275]: The **`asyncio`** package is now imported lazily in **`unittest`** only when the **`IsolatedAsyncioTestCase`** class is used.

- bpo-40330 [https://bugs.python.org/issue?@action=redirect&bpo=40330]: In **`ShareableList.__setitem__()`**, check the size of a new string item after encoding it to utf-8, not before.

- bpo-40148 [https://bugs.python.org/issue?@action=redirect&bpo=40148]: Added **`pathlib.Path.with_stem()`** to create a new Path with the stem replaced.

- bpo-40325 [https://bugs.python.org/issue?@action=redirect&bpo=40325]: Deprecated support for set objects in random.sample().

- bpo-40257 [https://bugs.python.org/issue?@action=redirect&bpo=40257]: Improved help for the **`typing`** module. Docstrings are now shown for all special forms and special generic aliases (like `Union` and `List`). Using `help()` with generic alias like `List[int]` will show the help for the correspondent concrete type (`list` in this case).

- bpo-40257 [https://bugs.python.org/issue?@action=redirect&bpo=40257]: **`inspect.getdoc()`** no longer returns docstring inherited from the type of the object or from parent class if it is a class if it is not defined in the object itself. In **`pydoc`** the documentation string is now shown not only for class, function, method etc, but for any object that has its own `__doc__` attribute.

- bpo-40287 [https://bugs.python.org/issue?@action=redirect&bpo=40287]: Fixed `SpooledTemporaryFile.seek()` to return the position.

- bpo-40290 [https://bugs.python.org/issue?@action=redirect&bpo=40290]: Added zscore() to statistics.NormalDist().

- bpo-40282 [https://bugs.python.org/issue?@action=redirect&bpo=40282]: Allow `random.getrandbits(0)` to succeed and to return 0.

- bpo-40286 [https://bugs.python.org/issue?@action=redirect&bpo=40286]: Add **`random.randbytes()`** function and **`random.Random.randbytes()`** method to generate random bytes.

- bpo-40277 [https://bugs.python.org/issue?@action=redirect&bpo=40277]: **`collections.namedtuple()`** now provides a human-readable repr for its field accessors.

- bpo-40270 [https://bugs.python.org/issue?@action=redirect&bpo=40270]: The included copy of sqlite3 on Windows is now compiled with the json extension. This allows the use of functions such as `json_object`.

- bpo-29255 [https://bugs.python.org/issue?@action=redirect&bpo=29255]: Wait in `KqueueSelector.select` when no fds are registered

- bpo-40260 [https://bugs.python.org/issue?@action=redirect&bpo=40260]: Ensure `modulefinder` uses `io.open_code()` and respects coding comments.

- bpo-40234 [https://bugs.python.org/issue?@action=redirect&bpo=40234]: Allow again to spawn daemon threads in subinterpreters (revert change which denied them).

- bpo-39207 [https://bugs.python.org/issue?@action=redirect&bpo=39207]: Workers in `ProcessPoolExecutor` are now spawned on demand, only when there are no available idle workers to reuse. This optimizes startup overhead and reduces the amount of lost CPU time to idle workers. Patch by Kyle Stanley.

- bpo-40091 [https://bugs.python.org/issue?@action=redirect&bpo=40091]: Fix a hang at fork in the logging module: the new private _at_fork_reinit() method is now used to reinitialize locks at fork in the child process.

- bpo-40149 [https://bugs.python.org/issue?@action=redirect&bpo=40149]: Implement traverse and clear slots in _abc._abc_data type.

- bpo-40208 [https://bugs.python.org/issue?@action=redirect&bpo=40208]: Remove deprecated `symtable.SymbolTable.has_exec()`.

- bpo-40196 [https://bugs.python.org/issue?@action=redirect&bpo=40196]: Fix a bug in the `symtable` module that was causing incorrectly report global variables as local. Patch by Pablo Galindo.

- bpo-40190 [https://bugs.python.org/issue?@action=redirect&bpo=40190]: Add support for `_SC_AIX_REALMEM` to `posix.sysconf()`.

- bpo-40182 [https://bugs.python.org/issue?@action=redirect&bpo=40182]: Removed the `_field_types` attribute of the **`typing.NamedTuple`** class.

- bpo-36517 [https://bugs.python.org/issue?@action=redirect&bpo=36517]: Multiple inheritance with **`typing.NamedTuple`** now raises an error instead of silently ignoring other types.

- bpo-40126 [https://bugs.python.org/issue?@action=redirect&bpo=40126]: Fixed reverting multiple patches in unittest.mock. Patcher's `__exit__()` is now never called if its `__enter__()` is failed. Returning true from `__exit__()` silences now the exception.

- bpo-40094 [https://bugs.python.org/issue?@action=redirect&bpo=40094]: CGIHTTPRequestHandler of http.server now logs the CGI script exit code, rather than the CGI script exit status of os.waitpid(). For example, if the script is killed by signal 11, it now logs: "CGI script exit code -11."

- bpo-40108 [https://bugs.python.org/issue?@action=redirect&bpo=40108]: Improve the error message when triying to import a module using **`runpy`** and incorrectly using the ".py" extension at the end of the module name. Patch by Pablo Galindo.

- bpo-40094 [https://bugs.python.org/issue?@action=redirect&bpo=40094]: Add **`os.waitstatus_to_exitcode()`** function: convert a wait status to an exit code.

- bpo-40089 [https://bugs.python.org/issue?@action=redirect&bpo=40089]: Fix threading._after_fork(): if fork was not called by a thread spawned by threading.Thread, threading._after_fork() now creates a

_MainThread instance for _main_thread, instead of a _DummyThread instance.

- bpo-40089 [https://bugs.python.org/issue?@action=redirect&bpo=40089]: Add a private `_at_fork_reinit()` method to **`_thread.Lock`**, **`_thread.RLock`**, **`threading.RLock`** and **`threading.Condition`** classes: reinitialize the lock at fork in the child process, reset the lock to the unlocked state. Rename also the private `_reset_internal_locks()` method of **`threading.Event`** to `_at_fork_reinit()`.

- bpo-25780 [https://bugs.python.org/issue?@action=redirect&bpo=25780]: Expose **`CAN_RAW_JOIN_FILTERS`** in the **`socket`** module.

- bpo-39503 [https://bugs.python.org/issue?@action=redirect&bpo=39503]: **`AbstractBasicAuthHandler`** of **`urllib.request`** now parses all WWW-Authenticate HTTP headers and accepts multiple challenges per header: use the realm of the first Basic challenge.

- bpo-39812 [https://bugs.python.org/issue?@action=redirect&bpo=39812]: Removed daemon threads from **`concurrent.futures`** by adding an internal `threading._register_atexit()`, which calls registered functions prior to joining all non-daemon threads. This allows for compatibility with subinterpreters, which don’t support daemon threads.

- bpo-40050 [https://bugs.python.org/issue?@action=redirect&bpo=40050]: Fix `importlib._bootstrap_external`: avoid creating a new `winreg` builtin module if it’s already available in **`sys.modules`**, and remove redundant imports.

- bpo-40014 [https://bugs.python.org/issue?@action=redirect&bpo=40014]: Fix `os.getgrouplist()`: if `getgrouplist()` function fails because the group list is too small, retry with a larger group list. On failure, the glibc implementation of `getgrouplist()` sets `ngroups` to the total number of groups. For other implementations, double the group list size.

- bpo-40017 [https://bugs.python.org/issue?@action=redirect&bpo=40017]: Add **`time.CLOCK_TAI`** constant if the operating system support it.
- bpo-40016 [https://bugs.python.org/issue?@action=redirect&bpo=40016]: In re docstring, clarify the relationship between inline and argument compile flags.
- bpo-39953 [https://bugs.python.org/issue?@action=redirect&bpo=39953]: Update internal table of OpenSSL error codes in the `ssl` module.
- bpo-36144 [https://bugs.python.org/issue?@action=redirect&bpo=36144]: Added **PEP 584** [https://peps.python.org/pep-0584/] operators to **`weakref.WeakValueDictionary`**.
- bpo-36144 [https://bugs.python.org/issue?@action=redirect&bpo=36144]: Added **PEP 584** [https://peps.python.org/pep-0584/] operators to **`weakref.WeakKeyDictionary`**.
- bpo-38891 [https://bugs.python.org/issue?@action=redirect&bpo=38891]: Fix linear runtime behaviour of the `__getitem__` and `__setitem__` methods in **`multiprocessing.shared_memory.ShareableList`**. This avoids quadratic performance when iterating a `ShareableList`. Patch by Thomas Krennwallner.
- bpo-39682 [https://bugs.python.org/issue?@action=redirect&bpo=39682]: Remove undocumented

support for *closing* a `pathlib.Path` object via its context manager. The context manager magic methods remain, but they are now a no-op, making `Path` objects immutable.

- bpo-36144 [https://bugs.python.org/issue?@action=redirect&bpo=36144]: Added **PEP 584** [https://peps.python.org/pep-0584/] operators (`|` and `|=`) to **`collections.ChainMap`**.
- bpo-39011 [https://bugs.python.org/issue?@action=redirect&bpo=39011]: Normalization of line endings in ElementTree attributes was removed, as line endings which were replaced by entity numbers should be preserved in original form.
- bpo-38410 [https://bugs.python.org/issue?@action=redirect&bpo=38410]: Properly handle **`sys.audit()`** failures in **`sys.set_asyncgen_hooks()`**.
- bpo-36541 [https://bugs.python.org/issue?@action=redirect&bpo=36541]: lib2to3 now recognizes named assignment expressions (the walrus operator, `:=`)
- bpo-35967 [https://bugs.python.org/issue?@action=redirect&bpo=35967]: In platform, delay the invocation of ‘uname -p’ until the processor attribute is requested.
- bpo-35113 [https://bugs.python.org/issue?@action=redirect&bpo=35113]: **`inspect.getsource()`** now returns correct source code for inner class with same name as module level class. Decorators are also returned as part of source of the class. Patch by Karthikeyan Singaravelan.

- bpo-33262 [https://bugs.python.org/issue?@action=redirect&bpo=33262]: Deprecate passing None as an argument for `shlex.split()`'s `s` parameter. Patch by Zackery Spytz.

- bpo-31758 [https://bugs.python.org/issue?@action=redirect&bpo=31758]: Prevent crashes when using an uninitialized `_elementtree.XMLParser` object. Patch by Oren Milman.

Documentation

- bpo-27635 [https://bugs.python.org/issue?@action=redirect&bpo=27635]: The pickle documentation incorrectly claimed that `__new__` isn't called by default when unpickling.
- bpo-39879 [https://bugs.python.org/issue?@action=redirect&bpo=39879]: Updated Data model docs to include `dict()` insertion order preservation. Patch by Furkan Onder and Samy Lahfa.
- bpo-38387 [https://bugs.python.org/issue?@action=redirect&bpo=38387]: Document `PyDoc_STRVAR` macro in the C-API reference.
- bpo-13743 [https://bugs.python.org/issue?@action=redirect&bpo=13743]: Some methods within xml.dom.minidom.Element class are now better documented.

Tests

- bpo-31904 [https://bugs.python.org/issue?@action=redirect&bpo=31904]: Set expected default encoding in test_c_locale_coercion.py for VxWorks RTOS.
- bpo-40162 [https://bugs.python.org/issue?@action=redirect&bpo=40162]: Update Travis CI configuration to OpenSSL 1.1.1f.

- bpo-40146 [https://bugs.python.org/issue?@action=redirect&bpo=40146]: Update OpenSSL to 1.1.1f in Azure Pipelines.
- bpo-40094 [https://bugs.python.org/issue?@action=redirect&bpo=40094]: Add `test.support.wait_process()` function.
- bpo-40003 [https://bugs.python.org/issue?@action=redirect&bpo=40003]: `test.bisect_cmd` now copies Python command line options like `-O` or `-W`. Moreover, emit a warning if `test.bisect_cmd` is used with `-w`/`--verbose2` option.
- bpo-39380 [https://bugs.python.org/issue?@action=redirect&bpo=39380]: Add the encoding in `ftplib.FTP` and `ftplib.FTP_TLS` to the constructor as keyword-only and change the default from `latin-1` to `utf-8` to follow **RFC 2640** [https://datatracker.ietf.org/doc/html/rfc2640.html].
- bpo-39793 [https://bugs.python.org/issue?@action=redirect&bpo=39793]: Use the same domain when testing `make_msgid`. Patch by Batuhan Taskaya.
- bpo-1812 [https://bugs.python.org/issue?@action=redirect&bpo=1812]: Fix newline handling in doctest.testfile when loading from a package whose loader has a get_data method. Patch by Peter Donis.

Build

- bpo-38360 [https://bugs.python.org/issue?@action=redirect&bpo=38360]: Support single-argument form of macOS -isysroot flag.
- bpo-40158 [https://bugs.python.org/issue?@action=redirect&bpo=40158]: Fix CPython MSBuild Properties in NuGet Package (build/native/python.props)
- bpo-38527 [https://bugs.python.org/issue?@action=redirect&bpo=38527]: Fix configure check on Solaris for “float word ordering”: sometimes, the correct “grep” command was not being used. Patch by Arnon Yaari.

Windows

- bpo-40164 [https://bugs.python.org/issue?@action=redirect&bpo=40164]: Updates Windows to OpenSSL 1.1.1f
- bpo-8901 [https://bugs.python.org/issue?@action=redirect&bpo=8901]: Ignore the Windows registry when the `-E` option is used.

macOS

- bpo-38329 [https://bugs.python.org/issue?@action=redirect&bpo=38329]: python.org macOS installers now update the Current version symlink of /Library/Frameworks/Python.framework/Versions for 3.9 installs. Previously, Current was only updated for Python 2.x installs. This should make it easier to embed Python 3 into other macOS applications.
- bpo-40164 [https://bugs.python.org/issue?@action=redirect&bpo=40164]: Update macOS installer builds to use OpenSSL 1.1.1g.

IDLE

- bpo-38439 [https://bugs.python.org/issue?@action=redirect&bpo=38439]: Add a 256×256 pixel IDLE icon to support more modern environments. Created by Andrew Clover. Delete the unused macOS idle.icns icon file.
- bpo-38689 [https://bugs.python.org/issue?@action=redirect&bpo=38689]: IDLE will no longer freeze when inspect.signature fails when fetching a calltip.

Tools/Demos

- bpo-40385 [https://bugs.python.org/issue?@action=redirect&bpo=40385]: Removed the checkpyc.py tool. Please see compileall without force mode as a potential alternative.
- bpo-40179 [https://bugs.python.org/issue?@action=redirect&bpo=40179]: Fixed translation of `#elif` in Argument Clinic.
- bpo-40094 [https://bugs.python.org/issue?@action=redirect&bpo=40094]: Fix `which.py` script exit code: it now uses `os.waitstatus_to_exitcode()` to convert `os.system()` exit status into an exit code.

C API

- bpo-40241 [https://bugs.python.org/issue?@action=redirect&bpo=40241]: Move the **`PyGC_Head`** structure to the internal C API.
- bpo-40170 [https://bugs.python.org/issue?@action=redirect&bpo=40170]: Convert `PyObject_IS_GC()` macro to a function to hide implementation details.
- bpo-40241 [https://bugs.python.org/issue?@action=redirect&bpo=40241]: Add the functions `PyObject_GC_IsTracked()` and `PyObject_GC_IsFinalized()` to the public API to allow to query if Python objects are being currently tracked or have been already finalized by the garbage collector respectively. Patch by Pablo Galindo.
- bpo-40170 [https://bugs.python.org/issue?@action=redirect&bpo=40170]: The **`PyObject_NEW()`** macro becomes an alias to the `PyObject_New()` macro, and the **`PyObject_NEW_VAR()`** macro becomes an alias to the `PyObject_NewVar()` macro, to hide implementation details. They no longer access directly the `PyTypeObject.tp_basicsize` member.
- bpo-40170 [https://bugs.python.org/issue?@action=redirect&bpo=40170]: `PyType_HasFeature()` now always

calls **`PyType_GetFlags()`** to hide implementation details. Previously, it accessed directly the **`PyTypeObject.tp_flags`** member when the limited C API was not used.

- bpo-40170 [https://bugs.python.org/issue?@action=redirect&bpo=40170]: Convert the **`PyObject_GET_WEAKREFS_LISTPTR()`** macro to a function to hide implementation details: the macro accessed directly to the **`PyTypeObject.tp_weaklistoffset`** member.
- bpo-40170 [https://bugs.python.org/issue?@action=redirect&bpo=40170]: Convert **`PyObject_CheckBuffer()`** macro to a function to hide implementation details: the macro accessed directly the **`PyTypeObject.tp_as_buffer`** member.
- bpo-40170 [https://bugs.python.org/issue?@action=redirect&bpo=40170]: Always declare **`PyIndex_Check()`** as an opaque function to hide implementation details: remove `PyIndex_Check()` macro. The macro accessed directly the **`PyTypeObject.tp_as_number`** member.
- bpo-39947 [https://bugs.python.org/issue?@action=redirect&bpo=39947]: Add **`PyThreadState_GetID()`** function: get the unique identifier of a Python thread state.

Python 3.9.0 alpha 5

Release date: 2020-03-23

Security

- bpo-38576 [https://bugs.python.org/issue?@action=redirect&bpo=38576]: Disallow control characters in hostnames in http.client, addressing **CVE 2019-18348** [https://www.cve.org/CVERecord?id=CVE-2019-18348]. Such potentially malicious header injection URLs now cause a InvalidURL to be raised.

Core and Builtins

- bpo-40010 [https://bugs.python.org/issue?@action=redirect&bpo=40010]: Optimize pending calls in multithreaded applications. If a thread different than the main thread schedules a pending call (`Py_AddPendingCall()`), the bytecode evaluation loop is no longer interrupted at each bytecode instruction to check for pending calls which cannot be executed. Only the main thread can execute pending calls.

  Previously, the bytecode evaluation loop was interrupted at each instruction until the main thread executes pending calls.

- bpo-1635741 [https://bugs.python.org/issue?@action=redirect&bpo=1635741]: Port _weakref extension module to multiphase initialization (**PEP 489** [https://peps.python.org/pep-0489/]).

- bpo-1635741 [https://bugs.python.org/issue?@action=redirect&bpo=1635741]: Port _collections module to multiphase initialization (**PEP 489** [https://peps.python.org/pep-0489/]).

- bpo-40010 [https://bugs.python.org/issue?@action=redirect&bpo=40010]: Optimize signal handling in multithreaded applications. If a thread different than the main thread gets a signal, the bytecode evaluation loop is no longer interrupted at each bytecode instruction to check for pending signals which cannot be handled. Only the main thread of the main interpreter can handle signals.

  Previously, the bytecode evaluation loop was interrupted at each instruction until the main thread handles signals.

- bpo-39984 [https://bugs.python.org/issue?@action=redirect&bpo=39984]: If **`Py_AddPendingCall()`** is called in a subinterpreter, the function is now scheduled to be called from the subinterpreter, rather than being called from the main interpreter. Each subinterpreter now has its own list of scheduled calls.

- bpo-1635741 [https://bugs.python.org/issue?@action=redirect&bpo=1635741]: Port _heapq module to multiphase initialization.

- bpo-1635741 [https://bugs.python.org/issue?@action=redirect&bpo=1635741]: Port itertools module to multiphase initialization (**PEP 489** [https://peps.python.org/pep-0489/]).

- bpo-37207 [https://bugs.python.org/issue?@action=redirect&bpo=37207]: Speed up calls to `frozenset()` by using the **PEP 590** [https://peps.python.org/pep-0590/] `vectorcall` calling convention. Patch by Donghee Na.

- bpo-39984 [https://bugs.python.org/issue?@action=redirect&bpo=39984]: subinterpreters: Move `_PyRuntimeState.ceval.tracing_possible` to `PyInterpreterState.ceval.tracing_possible`: each interpreter now has its own variable.

- bpo-37207 [https://bugs.python.org/issue?@action=redirect&bpo=37207]: Speed up calls to `set()` by using the **PEP 590** [https://peps.python.org/pep-0590/] `vectorcall` calling convention. Patch by Donghee Na.

- bpo-1635741 [https://bugs.python.org/issue?@action=redirect&bpo=1635741]: Port _statistics module to multiphase initialization (**PEP 489** [https://peps.python.org/pep-0489/]).

- bpo-39968 [https://bugs.python.org/issue?@action=redirect&bpo=39968]: Use inline function to replace extension modules' get_module_state macros.
- bpo-39965 [https://bugs.python.org/issue?@action=redirect&bpo=39965]: Correctly raise `SyntaxError` if *await* is used inside non-async functions and `PyCF_ALLOW_TOP_LEVEL_AWAIT` is set (like in the asyncio REPL). Patch by Pablo Galindo.
- bpo-39562 [https://bugs.python.org/issue?@action=redirect&bpo=39562]: Allow executing asynchronous comprehensions on the top level when the `PyCF_ALLOW_TOP_LEVEL_AWAIT` flag is given. Patch by Batuhan Taskaya.
- bpo-37207 [https://bugs.python.org/issue?@action=redirect&bpo=37207]: Speed up calls to `tuple()` by using the **PEP 590** [https://peps.python.org/pep-0590/] `vectorcall` calling convention. Patch by Donghee Na.
- bpo-38373 [https://bugs.python.org/issue?@action=redirect&bpo=38373]: Changed list overallocation strategy. It no longer overallocates if the new size is closer to overallocated size than to the old size and adds padding.
- bpo-39926 [https://bugs.python.org/issue?@action=redirect&bpo=39926]: Update Unicode database to Unicode version 13.0.0.
- bpo-19466 [https://bugs.python.org/issue?@action=redirect&bpo=19466]: Clear the frames of daemon threads earlier during the Python shutdown to call

objects destructors. So “unclosed file” resource warnings are now emitted for daemon threads in a more reliable way.

- bpo-38894 [https://bugs.python.org/issue?@action=redirect&bpo=38894]: Fix a bug that was causing incomplete results when calling `pathlib.Path.glob` in the presence of symlinks that point to files where the user does not have read access. Patch by Pablo Galindo and Matt Wozniski.

- bpo-39877 [https://bugs.python.org/issue?@action=redirect&bpo=39877]: Fix **`PyEval_RestoreThread()`** random crash at exit with daemon threads. It now accesses the `_PyRuntime` variable directly instead of using `tstate->interp->runtime`, since `tstate` can be a dangling pointer after **`Py_Finalize()`** has been called. Moreover, the daemon thread now exits before trying to take the GIL.

- bpo-39871 [https://bugs.python.org/issue?@action=redirect&bpo=39871]: Fix a possible **`SystemError`** in `math.{atan2,copysign,remainder}()` when the first argument cannot be converted to a **`float`**. Patch by Zackery Spytz.

- bpo-39776 [https://bugs.python.org/issue?@action=redirect&bpo=39776]: Fix race condition where threads created by PyGILState_Ensure() could get a duplicate id.

  This affects consumers of tstate->id like the contextvar caching machinery, which could return invalid cached objects under heavy thread load (observed in embedded scenarios).

- bpo-39778 [https://bugs.python.org/issue?@action=redirect&bpo=39778]: Fixed a crash due to incorrect handling of weak references in `collections.OrderedDict` classes. Patch by Pablo Galindo.

- bpo-1635741 [https://bugs.python.org/issue?@action=redirect&bpo=1635741]: Port audioop extension module to multiphase initialization (**PEP 489** [https://peps.python.org/pep-0489/]).
- bpo-39702 [https://bugs.python.org/issue?@action=redirect&bpo=39702]: Relax decorator grammar restrictions to allow any valid expression (**PEP 614** [https://peps.python.org/pep-0614/]).
- bpo-38091 [https://bugs.python.org/issue?@action=redirect&bpo=38091]: Tweak import deadlock detection code to not deadlock itself.
- bpo-1635741 [https://bugs.python.org/issue?@action=redirect&bpo=1635741]: Port _locale extension module to multiphase initialization (**PEP 489** [https://peps.python.org/pep-0489/]).
- bpo-39087 [https://bugs.python.org/issue?@action=redirect&bpo=39087]: Optimize **`PyUnicode_AsUTF8()`** and **`PyUnicode_AsUTF8AndSize()`** slightly when they need to create internal UTF-8 cache.
- bpo-39520 [https://bugs.python.org/issue?@action=redirect&bpo=39520]: Fix unparsing of ext slices with no items (`foo[:,]`). Patch by Batuhan Taskaya.
- bpo-39220 [https://bugs.python.org/issue?@action=redirect&bpo=39220]: Do not optimize annotations if ‘from __future__ import annotations’ is used. Patch by Pablo Galindo.

- bpo-35712 [https://bugs.python.org/issue?@action=redirect&bpo=35712]: Using `NotImplemented` in a boolean context has been deprecated. Patch contributed by Josh Rosenberg.

- bpo-22490 [https://bugs.python.org/issue?@action=redirect&bpo=22490]: Don’t leak environment variable `__PYVENV_LAUNCHER__` into the interpreter session on macOS.

Library

- bpo-39830 [https://bugs.python.org/issue?@action=redirect&bpo=39830]: Add `zipfile.Path` to `__all__` in the `zipfile` module.
- bpo-40000 [https://bugs.python.org/issue?@action=redirect&bpo=40000]: Improved error messages for validation of `ast.Constant` nodes. Patch by Batuhan Taskaya.
- bpo-39999 [https://bugs.python.org/issue?@action=redirect&bpo=39999]: `__module__` of the AST node classes is now set to “ast” instead of “_ast”. Added docstrings for dummy AST node classes and deprecated attributes.
- bpo-39991 [https://bugs.python.org/issue?@action=redirect&bpo=39991]: `uuid.getnode()` now skips IPv6 addresses with the same string length than a MAC address (17 characters): only use MAC addresses.
- bpo-39988 [https://bugs.python.org/issue?@action=redirect&bpo=39988]: Deprecated `ast.AugLoad` and `ast.AugStore` node classes because they are no longer used.
- bpo-39656 [https://bugs.python.org/issue?@action=redirect&bpo=39656]: Ensure `bin/python3.#` is always present in virtual environments on POSIX platforms - by Anthony Sottile.

- bpo-39969 [https://bugs.python.org/issue?@action=redirect&bpo=39969]: Deprecated `ast.Param` node class because it’s no longer used. Patch by Batuhan Taskaya.
- bpo-39360 [https://bugs.python.org/issue?@action=redirect&bpo=39360]: Ensure all workers exit when finalizing a **`multiprocessing.Pool`** implicitly via the module finalization handlers of multiprocessing. This fixes a deadlock situation that can be experienced when the Pool is not properly finalized via the context manager or a call to `multiprocessing.Pool.terminate`. Patch by Batuhan Taskaya and Pablo Galindo.
- bpo-35370 [https://bugs.python.org/issue?@action=redirect&bpo=35370]: sys.settrace(), sys.setprofile() and _lsprof.Profiler.enable() now properly report **`PySys_Audit()`** error if “sys.setprofile” or “sys.settrace” audit event is denied.
- bpo-39936 [https://bugs.python.org/issue?@action=redirect&bpo=39936]: AIX: Fix _aix_support module when the subprocess is not available, when building Python from scratch. It now uses new private _bootsubprocess module, rather than having two implementations depending if subprocess is available or not. So _aix_support.aix_platform() result is now the same if subprocess is available or not.
- bpo-36144 [https://bugs.python.org/issue?@action=redirect&bpo=36144]: **`collections.OrderedDict`** now implements `|` and `|=` (**PEP 584** [https://peps.python.org/pep-0584/]).
- bpo-39652 [https://bugs.python.org/issue?@action=redirect&bpo=39652]: The column name found in `sqlite3.Cursor.description` is now truncated on the first ‘[’ only if the PARSE_COLNAMES option is set.
- bpo-39915 [https://bugs.python.org/issue?@action=redirect&bpo=39915]: Ensure **`unittest.mock.AsyncMock.await_args_list`** has call objects in

the order of awaited arguments instead of using `unittest.mock.Mock.call_args` which has the last value of the call. Patch by Karthikeyan Singaravelan.

- bpo-36144 [https://bugs.python.org/issue?@action=redirect&bpo=36144]: Updated `os.environ` and `os.environb` to support **PEP 584** [https://peps.python.org/pep-0584/]'s merge (`|`) and update (`|=`) operators.
- bpo-38662 [https://bugs.python.org/issue?@action=redirect&bpo=38662]: The `ensurepip` module now invokes `pip` via the `runpy` module. Hence it is no longer tightly coupled with the internal API of the bundled `pip` version, allowing easier updates to a newer `pip` version both internally and for distributors.
- bpo-38075 [https://bugs.python.org/issue?@action=redirect&bpo=38075]: Fix the `random.Random.seed()` method when a `bool` is passed as the seed.
- bpo-39916 [https://bugs.python.org/issue?@action=redirect&bpo=39916]: More reliable use of `os.scandir()` in `Path.glob()`. It no longer emits a ResourceWarning when interrupted.
- bpo-39850 [https://bugs.python.org/issue?@action=redirect&bpo=39850]: `multiprocessing` now supports abstract socket addresses (if abstract sockets are supported in the running platform). When creating arbitrary addresses (like when default-constructing `multiprocessing.connection.Listener` objects) abstract sockets are preferred to avoid the case when the temporary-file-generated address is too large for an AF_UNIX socket address. Patch by Pablo Galindo.
- bpo-36287 [https://bugs.python.org/issue?@action=redirect&bpo=36287]: `ast.dump()` no longer outputs optional fields and attributes with default values. The default values for optional fields and attributes of AST nodes are now set as class attributes (e.g. `Constant.kind` is set to `None`).

- bpo-39889 [https://bugs.python.org/issue?@action=redirect&bpo=39889]: Fixed `ast.unparse()` for extended slices containing a single element (e.g. `a[i:j,]`). Remove redundant tuples when index with a tuple (e.g. `a[i, j]`).
- bpo-39828 [https://bugs.python.org/issue?@action=redirect&bpo=39828]: Fix `json.tool` to catch `BrokenPipeError`. Patch by Donghee Na.
- bpo-13487 [https://bugs.python.org/issue?@action=redirect&bpo=13487]: Avoid a possible *“RuntimeError: dictionary changed size during iteration”* from `inspect.getmodule()` when it tried to loop through `sys.modules`.
- bpo-39674 [https://bugs.python.org/issue?@action=redirect&bpo=39674]: Revert “bpo-37330 [https://bugs.python.org/issue?@action=redirect&bpo=37330]: open() no longer accept ‘U’ in file mode”. The “U” mode of open() is kept in Python 3.9 to ease transition from Python 2.7, but will be removed in Python 3.10.
- bpo-28577 [https://bugs.python.org/issue?@action=redirect&bpo=28577]: The hosts method on 32-bit prefix length IPv4Networks and 128-bit prefix IPv6Networks now returns a list containing the single Address instead of an empty list.
- bpo-39826 [https://bugs.python.org/issue?@action=redirect&bpo=39826]: Add getConnection method to logging HTTPHandler to enable custom connections.
- bpo-39763 [https://bugs.python.org/issue?@action=redirect&bpo=39763]: Reimplement `distutils.spawn.spawn` function with the `subprocess` module.
- bpo-39794 [https://bugs.python.org/issue?@action=redirect&bpo=39794]: Add –without-decimal-contextvar build option. This enables a thread-local rather than a coroutine local context.
- bpo-36144 [https://bugs.python.org/issue?@action=redirect&bpo=36144]: `collections.defaultdict` now implements `|` (**PEP 584** [https://peps.python.org/pep-0584/]).

- bpo-39517 [https://bugs.python.org/issue?@action=redirect&bpo=39517]: Fix runpy.run_path() when using pathlike objects
- bpo-39775 [https://bugs.python.org/issue?@action=redirect&bpo=39775]: Change `inspect.Signature.parameters` back to `collections.OrderedDict`. This was changed to `dict` in Python 3.9.0a4.
- bpo-39678 [https://bugs.python.org/issue?@action=redirect&bpo=39678]: Refactor queue_manager in **`concurrent.futures.ProcessPoolExecutor`** to make it easier to maintain.
- bpo-39764 [https://bugs.python.org/issue?@action=redirect&bpo=39764]: Fix AttributeError when calling get_stack on a PyAsyncGenObject Task
- bpo-39769 [https://bugs.python.org/issue?@action=redirect&bpo=39769]: The **`compileall.compile_dir()`** function's *ddir* parameter and the compileall command line flag `-d` no longer write the wrong pathname to the generated pyc file for submodules beneath the root of the directory tree being compiled. This fixes a regression introduced with Python 3.5.
- bpo-36144 [https://bugs.python.org/issue?@action=redirect&bpo=36144]: **`types.MappingProxyType`** objects now support the merge (`|`) operator from **PEP 584** [https://peps.python.org/pep-0584/].
- bpo-38691 [https://bugs.python.org/issue?@action=redirect&bpo=38691]: The **`importlib`** module now ignores the **`PYTHONCASEOK`** environment variable when the **`-E`** or **`-I`** command line options are being used.
- bpo-39719 [https://bugs.python.org/issue?@action=redirect&bpo=39719]: Remove **`tempfile.SpooledTemporaryFile.softspace()`** as files no longer have the `softspace` attribute in Python 3. Patch by Shantanu.

- bpo-39667 [https://bugs.python.org/issue?@action=redirect&bpo=39667]: Improve pathlib.Path compatibility on zipfile.Path and correct performance degradation as found in zipp 3.0.
- bpo-39638 [https://bugs.python.org/issue?@action=redirect&bpo=39638]: Keep ASDL signatures in the docstrings for `AST` nodes. Patch by Batuhan Taskaya
- bpo-39639 [https://bugs.python.org/issue?@action=redirect&bpo=39639]: Deprecated `ast.Suite` node class because it's no longer used. Patch by Batuhan Taskaya.
- bpo-39609 [https://bugs.python.org/issue?@action=redirect&bpo=39609]: Add thread_name_prefix to default asyncio executor
- bpo-39548 [https://bugs.python.org/issue?@action=redirect&bpo=39548]: Fix handling of header in **`urllib.request.AbstractDigestAuthHandler`** when the optional `qop` parameter is not present.
- bpo-39509 [https://bugs.python.org/issue?@action=redirect&bpo=39509]: HTTP status codes `103 EARLY_HINTS` and `425 TOO_EARLY` are added to **`http.HTTPStatus`**. Patch by Donghee Na.
- bpo-39507 [https://bugs.python.org/issue?@action=redirect&bpo=39507]: Adding HTTP status 418 "I'm a Teapot" to HTTPStatus in http library. Patch by Ross Rhodes.
- bpo-39495 [https://bugs.python.org/issue?@action=redirect&bpo=39495]: Remove default value from *attrs* parameter of **`xml.etree.ElementTree.TreeBuilder.start()`** for consistency between Python and C implementations.
- bpo-38971 [https://bugs.python.org/issue?@action=redirect&bpo=38971]: Open issue in the BPO indicated a desire to make the implementation of codecs.open() at parity with io.open(), which

implements a try/except to assure file stream gets closed before an exception is raised.

- bpo-38641 [https://bugs.python.org/issue?@action=redirect&bpo=38641]: Added starred expressions support to `return` and `yield` statements for `lib2to3`. Patch by Vlad Emelianov.
- bpo-37534 [https://bugs.python.org/issue?@action=redirect&bpo=37534]: When using minidom module to generate XML documents the ability to add Standalone Document Declaration is added. All the changes are made to generate a document in compliance with Extensible Markup Language (XML) 1.0 (Fifth Edition) W3C Recommendation (available here: https://www.w3.org/TR/xml/#sec-prolog-dtd).
- bpo-34788 [https://bugs.python.org/issue?@action=redirect&bpo=34788]: Add support for scoped IPv6 addresses to **`ipaddress`**. Patch by Oleksandr Pavliuk.
- bpo-34822 [https://bugs.python.org/issue?@action=redirect&bpo=34822]: Simplified AST for subscription. Simple indices are now represented by their value, extended slices are represented as tuples. **`ast`** classes `Index` and `ExtSlice` are considered deprecated and will be removed in future Python versions. In the meantime, `Index(value)` now returns a `value` itself, `ExtSlice(slices)` returns `Tuple(slices, Load())`.

Documentation

- bpo-39868 [https://bugs.python.org/issue?@action=redirect&bpo=39868]: Updated the Language Reference for **PEP 572** [https://peps.python.org/pep-0572/].
- bpo-13790 [https://bugs.python.org/issue?@action=redirect&bpo=13790]: Change ‘string’ to ‘specification’ in format doc.
- bpo-17422 [https://bugs.python.org/issue?@action=redirect&bpo=17422]: The language reference no

longer restricts default class namespaces to dicts only.

- bpo-39530 [https://bugs.python.org/issue?@action=redirect&bpo=39530]: Fix misleading documentation about mixed-type numeric comparisons.
- bpo-39718 [https://bugs.python.org/issue?@action=redirect&bpo=39718]: Update **`token`** documentation to reflect additions in Python 3.8
- bpo-39677 [https://bugs.python.org/issue?@action=redirect&bpo=39677]: Changed operand name of **MAKE_FUNCTION** from *argc* to *flags* for module **`dis`**

Tests

- bpo-40019 [https://bugs.python.org/issue?@action=redirect&bpo=40019]: test_gdb now skips tests if it detects that gdb failed to read debug information because the Python binary is optimized.
- bpo-27807 [https://bugs.python.org/issue?@action=redirect&bpo=27807]: `test_site.test_startup_imports()` is now skipped if a path of **`sys.path`** contains a `.pth` file.
- bpo-26067 [https://bugs.python.org/issue?@action=redirect&bpo=26067]: Do not fail test_shutil test_chown test when uid or gid of user cannot be resolved to a name.
- bpo-39855 [https://bugs.python.org/issue?@action=redirect&bpo=39855]: test_subprocess.test_user() now skips the test on an user name if the user name doesn’t exist. For example, skip the test if the user “nobody” doesn’t exist on Linux.

Build

- bpo-39761 [https://bugs.python.org/issue?@action=redirect&bpo=39761]: Fix build with DTrace but without additional DFLAGS.

- bpo-39763 [https://bugs.python.org/issue?@action=redirect&bpo=39763]: setup.py now uses a basic implementation of the **`subprocess`** module if the **`subprocess`** module is not available: before required C extension modules are built.
- bpo-1294959 [https://bugs.python.org/issue?@action=redirect&bpo=1294959]: Add `--with-platlibdir` option to the configure script: name of the platform-specific library directory, stored in the new **`sys.platlibdir`** attribute. It is used to build the path of platform-specific extension modules and the path of the standard library. It is equal to `"lib"` on most platforms. On Fedora and SuSE, it is equal to `"lib64"` on 64-bit platforms. Patch by Jan Matějek, Matěj Cepl, Charalampos Stratakis and Victor Stinner.

Windows

- bpo-39930 [https://bugs.python.org/issue?@action=redirect&bpo=39930]: Ensures the required `vcruntime140.dll` is included in install packages.
- bpo-39847 [https://bugs.python.org/issue?@action=redirect&bpo=39847]: Avoid hang when computer is hibernated whilst waiting for a mutex (for lock-related objects from **`threading`**) around 49-day uptime.
- bpo-38597 [https://bugs.python.org/issue?@action=redirect&bpo=38597]: `distutils` will no longer statically link `vcruntime140.dll` when a redistributable version is unavailable. All future releases of CPython will include a copy of this DLL to ensure distributed extensions can continue to load.
- bpo-38380 [https://bugs.python.org/issue?@action=redirect&bpo=38380]: Update Windows builds to use SQLite 3.31.1
- bpo-39789 [https://bugs.python.org/issue?@action=redirect&bpo=39789]: Update Windows release build

machines to Visual Studio 2019 (MSVC 14.2).

- bpo-34803 [https://bugs.python.org/issue?@action=redirect&bpo=34803]: Package for nuget.org now includes repository reference and bundled icon image.

macOS

- bpo-38380 [https://bugs.python.org/issue?@action=redirect&bpo=38380]: Update macOS builds to use SQLite 3.31.1

IDLE

- bpo-27115 [https://bugs.python.org/issue?@action=redirect&bpo=27115]: For ‘Go to Line’, use a Query box subclass with IDLE standard behavior and improved error checking.
- bpo-39885 [https://bugs.python.org/issue?@action=redirect&bpo=39885]: Since clicking to get an IDLE context menu moves the cursor, any text selection should be and now is cleared.
- bpo-39852 [https://bugs.python.org/issue?@action=redirect&bpo=39852]: Edit “Go to line” now clears any selection, preventing accidental deletion. It also updates Ln and Col on the status bar.
- bpo-39781 [https://bugs.python.org/issue?@action=redirect&bpo=39781]: Selecting code context lines no longer causes a jump.

Tools/Demos

- bpo-36184 [https://bugs.python.org/issue?@action=redirect&bpo=36184]: Port python-gdb.py to FreeBSD. python-gdb.py now checks for “take_gil” function name to check if a frame tries to acquire the GIL, instead of checking for “pthread_cond_timedwait” which is specific to Linux and can be a different condition than the GIL.

- bpo-38080 [https://bugs.python.org/issue?@action=redirect&bpo=38080]: Added support to fix `getproxies` in the **`lib2to3.fixes.fix_urllib`** module. Patch by José Roberto Meza Cabrera.

C API

- bpo-40024 [https://bugs.python.org/issue?@action=redirect&bpo=40024]: Add **`PyModule_AddType()`** helper function: add a type to a module. Patch by Donghee Na.

- bpo-39946 [https://bugs.python.org/issue?@action=redirect&bpo=39946]: Remove `_PyRuntime.getframe` hook and remove `_PyThreadState_GetFrame` macro which was an alias to `_PyRuntime.getframe`. They were only exposed by the internal C API. Remove also `PyThreadFrameGetter` type.

- bpo-39947 [https://bugs.python.org/issue?@action=redirect&bpo=39947]: Add **`PyThreadState_GetFrame()`** function: get the current frame of a Python thread state.

- bpo-37207 [https://bugs.python.org/issue?@action=redirect&bpo=37207]: Add _PyArg_NoKwnames helper function. Patch by Donghee Na.

- bpo-39947 [https://bugs.python.org/issue?@action=redirect&bpo=39947]: Add **`PyThreadState_GetInterpreter()`**: get the interpreter of a Python thread state.

- bpo-39947 [https://bugs.python.org/issue?@action=redirect&bpo=39947]: Add **`PyInterpreterState_Get()`** function to the limited C API.

- bpo-35370 [https://bugs.python.org/issue?@action=redirect&bpo=35370]: If **`PySys_Audit()`** fails in **`PyEval_SetProfile()`** or **`PyEval_SetTrace()`**, log the error as an unraisable exception.

- bpo-39947 [https://bugs.python.org/issue?@action=redirect&bpo=39947]: Move the static inline function flavor of Py_EnterRecursiveCall() and Py_LeaveRecursiveCall() to the internal C API: they access PyThreadState attributes. The limited C API provides regular functions which hide implementation details.

- bpo-39947 [https://bugs.python.org/issue?@action=redirect&bpo=39947]: Py_TRASHCAN_BEGIN_CONDITION and Py_TRASHCAN_END macro no longer access PyThreadState attributes, but call new private _PyTrash_begin() and _PyTrash_end() functions which hide implementation details.

- bpo-39884 [https://bugs.python.org/issue?@action=redirect&bpo=39884]: **`PyDescr_NewMethod()`** and **`PyCFunction_NewEx()`** now include the method name in the SystemError “bad call flags” error message to ease debug.

- bpo-39877 [https://bugs.python.org/issue?@action=redirect&bpo=39877]: Deprecated **`PyEval_InitThreads()`** and **`PyEval_ThreadsInitialized()`**. Calling **`PyEval_InitThreads()`** now does nothing.

- bpo-38249 [https://bugs.python.org/issue?@action=redirect&bpo=38249]: **`Py_UNREACHABLE`** is now implemented with `__builtin_unreachable()` and analogs in release mode.

- bpo-38643 [https://bugs.python.org/issue?@action=redirect&bpo=38643]: **`PyNumber_ToBase()`** now raises a **`SystemError`** instead of crashing when called with invalid base.

- bpo-39882 [https://bugs.python.org/issue?@action=redirect&bpo=39882]: The **`Py_FatalError()`** function is replaced with a macro which logs automatically the name of the current function, unless the `Py_LIMITED_API` macro is defined.

- bpo-39824 [https://bugs.python.org/issue?@action=redirect&bpo=39824]: Extension modules: **`m_traverse`**, **`m_clear`** and **`m_free`** functions of **`PyModuleDef`** are no longer called if the module state was requested but is not allocated yet. This is the case immediately after the module is created and before the module is executed (**`Py_mod_exec`** function). More precisely, these functions are not called if **`m_size`** is greater than 0 and the module state (as returned by **`PyModule_GetState()`**) is `NULL`.

  Extension modules without module state (`m_size <= 0`) are not affected.

- bpo-38913 [https://bugs.python.org/issue?@action=redirect&bpo=38913]: Fixed segfault in `Py_BuildValue()` called with a format containing "#" and undefined PY_SSIZE_T_CLEAN whwn an exception is set.

- bpo-38500 [https://bugs.python.org/issue?@action=redirect&bpo=38500]: Add a private API to get and set the frame evaluation function: add **`_PyInterpreterState_GetEvalFrameFunc()`** and **`_PyInterpreterState_SetEvalFrameFunc()`** C functions. The **`_PyFrameEvalFunction`** function type now takes a *tstate* parameter.

Python 3.9.0 alpha 4

Release date: 2020-02-25

Security

- bpo-39184 [https://bugs.python.org/issue?@action=redirect&bpo=39184]: Add audit events to functions in `fcntl`, `msvcrt`, `os`, `resource`, `shutil`, `signal` and `syslog`.
- bpo-39401 [https://bugs.python.org/issue?@action=redirect&bpo=39401]: Avoid unsafe DLL load at startup on Windows 7 and earlier.
- bpo-39184 [https://bugs.python.org/issue?@action=redirect&bpo=39184]: Add audit events to command execution functions in os and pty modules.

Core and Builtins

- bpo-39382 [https://bugs.python.org/issue?@action=redirect&bpo=39382]: Fix a use-after-free in the single inheritance path of `issubclass()`, when the `__bases__` of an object has a single reference, and so does its first item. Patch by Yonatan Goldschmidt.
- bpo-39573 [https://bugs.python.org/issue?@action=redirect&bpo=39573]: Update clinic tool to use **`Py_IS_TYPE()`**. Patch by Donghee Na.
- bpo-39619 [https://bugs.python.org/issue?@action=redirect&bpo=39619]: Enable use of **`os.chroot()`** on HP-UX systems.
- bpo-39573 [https://bugs.python.org/issue?@action=redirect&bpo=39573]: Add **`Py_IS_TYPE()`** static inline function to check whether the object *o* type is *type*.
- bpo-39606 [https://bugs.python.org/issue?@action=redirect&bpo=39606]: Fix regression caused by fix for bpo-39386 [https://bugs.python.org/issue?

@action=redirect&bpo=39386], that prevented calling `aclose` on an async generator that had already been closed or exhausted.
- bpo-39579 [https://bugs.python.org/issue?@action=redirect&bpo=39579]: Change the ending column offset of `Attribute` nodes constructed in `ast_for_dotted_name` to point at the end of the current node and not at the end of the last `NAME` node.
- bpo-1635741 [https://bugs.python.org/issue?@action=redirect&bpo=1635741]: Port _crypt extension module to multiphase initialization (**PEP 489** [https://peps.python.org/pep-0489/]).
- bpo-1635741 [https://bugs.python.org/issue?@action=redirect&bpo=1635741]: Port _contextvars extension module to multiphase initialization (**PEP 489** [https://peps.python.org/pep-0489/]).
- bpo-39510 [https://bugs.python.org/issue?@action=redirect&bpo=39510]: Fix segfault in `readinto()` method on closed BufferedReader.
- bpo-39502 [https://bugs.python.org/issue?@action=redirect&bpo=39502]: Fix `time.localtime()` on 64-bit AIX to support years before 1902 and after 2038. Patch by M Felt.
- bpo-39492 [https://bugs.python.org/issue?@action=redirect&bpo=39492]: Fix a reference cycle in the C Pickler that was preventing the garbage collection of deleted, pickled objects.
- bpo-39453 [https://bugs.python.org/issue?@action=redirect&bpo=39453]: Fixed a possible crash in **`list.__contains__()`** when a list is changed during comparing items. Patch by Donghee Na.
- bpo-39434 [https://bugs.python.org/issue?@action=redirect&bpo=39434]: floor division of float operation now has a better performance. Also the message of `ZeroDivisionError` for this operation is updated. Patch by Donghee Na.

- bpo-1635741 [https://bugs.python.org/issue?@action=redirect&bpo=1635741]: Port _codecs extension module to multiphase initialization (**PEP 489** [https://peps.python.org/pep-0489/]).
- bpo-1635741 [https://bugs.python.org/issue?@action=redirect&bpo=1635741]: Port _bz2 extension module to multiphase initialization (**PEP 489** [https://peps.python.org/pep-0489/]).
- bpo-1635741 [https://bugs.python.org/issue?@action=redirect&bpo=1635741]: Port _abc extension module to multiphase initialization (**PEP 489** [https://peps.python.org/pep-0489/]).
- bpo-39320 [https://bugs.python.org/issue?@action=redirect&bpo=39320]: Replace two complex bytecodes for building dicts with two simpler ones. The new bytecodes `DICT_MERGE` and `DICT_UPDATE` have been added The old bytecodes `BUILD_MAP_UNPACK` and `BUILD_MAP_UNPACK_WITH_CALL` have been removed.
- bpo-39219 [https://bugs.python.org/issue?@action=redirect&bpo=39219]: Syntax errors raised in the tokenizer now always set correct "text" and "offset" attributes.
- bpo-36051 [https://bugs.python.org/issue?@action=redirect&bpo=36051]: Drop the GIL during large `bytes.join` operations. Patch by Bruce Merry.
- bpo-38960 [https://bugs.python.org/issue?@action=redirect&bpo=38960]: Fix DTrace build issues on FreeBSD. Patch by David Carlier.
- bpo-37207 [https://bugs.python.org/issue?@action=redirect&bpo=37207]: Speed up calls to `range()` by about 30%, by using the PEP 590 `vectorcall` calling convention. Patch by Mark Shannon.
- bpo-36144 [https://bugs.python.org/issue?@action=redirect&bpo=36144]: **`dict`** (and **`collections.UserDict`**) objects now support PEP 584's merge (`|`) and update (`|=`) operators. Patch by Brandt Bucher.

- bpo-32856 [https://bugs.python.org/issue?@action=redirect&bpo=32856]: Optimized the idiom for assignment a temporary variable in comprehensions. Now `for y in [expr]` in comprehensions is as fast as a simple assignment `y = expr`.

Library

- bpo-30566 [https://bugs.python.org/issue?@action=redirect&bpo=30566]: Fix **`IndexError`** when trying to decode an invalid string with punycode codec.
- bpo-39649 [https://bugs.python.org/issue?@action=redirect&bpo=39649]: Remove obsolete check for `__args__` in `bdb.Bdb.format_stack_entry`.
- bpo-39648 [https://bugs.python.org/issue?@action=redirect&bpo=39648]: Expanded **`math.gcd()`** and **`math.lcm()`** to handle multiple arguments.
- bpo-39681 [https://bugs.python.org/issue?@action=redirect&bpo=39681]: Fix a regression where the C pickle module wouldn't allow unpickling from a file-like object that doesn't expose a readinto() method.
- bpo-35950 [https://bugs.python.org/issue?@action=redirect&bpo=35950]: Raise **`io.UnsupportedOperation`** in **`io.BufferedReader.truncate()`** when it is called on a read-only **`io.BufferedReader`** instance.
- bpo-39479 [https://bugs.python.org/issue?@action=redirect&bpo=39479]: Add **`math.lcm()`** function: least common multiple.
- bpo-39674 [https://bugs.python.org/issue?@action=redirect&bpo=39674]: Revert "Do not expose abstract collection classes in the collections module" change (bpo-25988 [https://bugs.python.org/issue?@action=redirect&bpo=25988]). Aliases to ABC like collections.Mapping are kept in Python 3.9 to ease transition from Python 2.7, but will be removed in Python 3.10.

- bpo-39104 [https://bugs.python.org/issue?@action=redirect&bpo=39104]: Fix hanging ProcessPoolExcutor on `shutdown(wait=False)` when a task has failed pickling.
- bpo-39627 [https://bugs.python.org/issue?@action=redirect&bpo=39627]: Fixed TypedDict totality check for inherited keys.
- bpo-39474 [https://bugs.python.org/issue?@action=redirect&bpo=39474]: Fixed starting position of AST for expressions like `(a)(b)`, `(a)[b]` and `(a).b`.
- bpo-21016 [https://bugs.python.org/issue?@action=redirect&bpo=21016]: The **`pydoc`** and **`trace`** modules now use the **`sysconfig`** module to get the path to the Python standard library, to support uncommon installation path like `/usr/lib64/python3.9/` on Fedora. Patch by Jan Matějek.
- bpo-39590 [https://bugs.python.org/issue?@action=redirect&bpo=39590]: Collections.deque now holds strong references during deque.__contains__ and deque.count, fixing crashes.
- bpo-39586 [https://bugs.python.org/issue?@action=redirect&bpo=39586]: The distutils `bdist_msi` command is deprecated in Python 3.9, use `bdist_wheel` (wheel packages) instead.
- bpo-39595 [https://bugs.python.org/issue?@action=redirect&bpo=39595]: Improved performance of zipfile.Path for files with a large number of entries. Also improved performance and fixed minor issue as published with importlib_metadata 1.5 [https://importlib-metadata.readthedocs.io/en/latest/history.html#v1-5-0].
- bpo-39350 [https://bugs.python.org/issue?@action=redirect&bpo=39350]: Fix regression in **`fractions.Fraction`** if the numerator and/or the denominator is an **`int`** subclass. The **`math.gcd()`** function is now used to normalize the *numerator* and *denominator*. **`math.gcd()`** always return a **`int`** type.

Previously, the GCD type depended on *numerator* and *denominator*.

- bpo-39567 [https://bugs.python.org/issue?@action=redirect&bpo=39567]: Added audit for **`os.walk()`**, **`os.fwalk()`**, **`pathlib.Path.glob()`** and **`pathlib.Path.rglob()`**.
- bpo-39559 [https://bugs.python.org/issue?@action=redirect&bpo=39559]: Remove unused, undocumented argument `getters` from **`uuid.getnode()`**
- bpo-38149 [https://bugs.python.org/issue?@action=redirect&bpo=38149]: **`sys.audit()`** is now called only once per call of **`glob.glob()`** and **`glob.iglob()`**.
- bpo-39546 [https://bugs.python.org/issue?@action=redirect&bpo=39546]: Fix a regression in **`ArgumentParser`** where `allow_abbrev=False` was ignored for long options that used a prefix character other than "-".
- bpo-39450 [https://bugs.python.org/issue?@action=redirect&bpo=39450]: Striped whitespace from docstring before returning it from **`unittest.case.shortDescription()`**.
- bpo-12915 [https://bugs.python.org/issue?@action=redirect&bpo=12915]: A new function `resolve_name` has been added to the `pkgutil` module. This resolves a string of the form `'a.b.c.d'` or `'a.b:c.d'` to an object. In the example, `a.b` is a package/module and `c.d` is an object within that package/module reached via recursive attribute access.
- bpo-39353 [https://bugs.python.org/issue?@action=redirect&bpo=39353]: The **`binascii.crc_hqx()`** function is no longer deprecated.
- bpo-39493 [https://bugs.python.org/issue?@action=redirect&bpo=39493]: Mark `typing.IO.closed` as a property
- bpo-39491 [https://bugs.python.org/issue?@action=redirect&bpo=39491]: Add **`typing.Annotated`** and `include_extras` parameter to **`typing.get_type_hints()`** as part of **PEP 593** [https://peps.python.org/pep-0593/]. Patch by

Till Varoquaux, documentation by Till Varoquaux and Konstantin Kashin.

- bpo-39485 [https://bugs.python.org/issue?@action=redirect&bpo=39485]: Fix a bug in **`unittest.mock.create_autospec()`** that would complain about the wrong number of arguments for custom descriptors defined in an extension module returning functions.
- bpo-38932 [https://bugs.python.org/issue?@action=redirect&bpo=38932]: Mock fully resets child objects on reset_mock(). Patch by Vegard Stikbakke
- bpo-39082 [https://bugs.python.org/issue?@action=redirect&bpo=39082]: Allow AsyncMock to correctly patch static/class methods
- bpo-39432 [https://bugs.python.org/issue?@action=redirect&bpo=39432]: Implement PEP-489 algorithm for non-ascii “PyInit_…” symbol names in distutils to make it export the correct init symbol also on Windows.
- bpo-18819 [https://bugs.python.org/issue?@action=redirect&bpo=18819]: Omit `devmajor` and `devminor` fields for non-device files in **`tarfile`** archives, enabling bit-for-bit compatibility with GNU `tar(1)`.
- bpo-39349 [https://bugs.python.org/issue?@action=redirect&bpo=39349]: Added a new *cancel_futures* parameter to **`concurrent.futures.Executor.shutdown()`** that cancels all pending futures which have not started running, instead of waiting for them to complete before shutting down the executor.
- bpo-39274 [https://bugs.python.org/issue?@action=redirect&bpo=39274]: `bool(fraction.Fraction)` now returns a boolean even if (numerator != 0) does not return a boolean (ex: numpy number).
- bpo-34793 [https://bugs.python.org/issue?@action=redirect&bpo=34793]: Remove support for `with (await asyncio.lock):` and `with (yield from asyncio.lock):`. The

same is correct for `asyncio.Condition` and `asyncio.Semaphore`.

- bpo-25597 [https://bugs.python.org/issue?@action=redirect&bpo=25597]: Ensure, if `wraps` is supplied to **`unittest.mock.MagicMock`**, it is used to calculate return values for the magic methods instead of using the default return values. Patch by Karthikeyan Singaravelan.
- bpo-36350 [https://bugs.python.org/issue?@action=redirect&bpo=36350]: `inspect.Signature.parameters` and `inspect.BoundArguments.arguments` are now dicts instead of OrderedDicts. Patch contributed by Rémi Lapeyre.
- bpo-35727 [https://bugs.python.org/issue?@action=redirect&bpo=35727]: Fix sys.exit() and sys.exit(None) exit code propagation when used in multiprocessing.Process.
- bpo-32173 [https://bugs.python.org/issue?@action=redirect&bpo=32173]: * Add `lazycache` function to `__all__`. * Use `dict.clear` to clear the cache. * Refactoring `getline` function and `checkcache` function.

Documentation

- bpo-17422 [https://bugs.python.org/issue?@action=redirect&bpo=17422]: The language reference now specifies restrictions on class namespaces. Adapted from a patch by Ethan Furman.
- bpo-39572 [https://bugs.python.org/issue?@action=redirect&bpo=39572]: Updated documentation of `total` flag of `TypedDict`.
- bpo-39654 [https://bugs.python.org/issue?@action=redirect&bpo=39654]: In pyclbr doc, update ‘class’ to ‘module’ where appropriate and add readmodule comment. Patch by Hakan Çelik.
- bpo-39153 [https://bugs.python.org/issue?@action=redirect&bpo=39153]: Clarify refcounting semantics

for the following functions: - PyObject_SetItem - PyMapping_SetItemString - PyDict_SetItem - PyDict_SetItemString

- bpo-39392 [https://bugs.python.org/issue?@action=redirect&bpo=39392]: Explain that when filling with turtle, overlap regions may be left unfilled.
- bpo-39369 [https://bugs.python.org/issue?@action=redirect&bpo=39369]: Update mmap readline method description. The fact that the readline method does update the file position should not be ignored since this might give the impression for the programmer that it doesn’t update it.
- bpo-9056 [https://bugs.python.org/issue?@action=redirect&bpo=9056]: Include subsection in TOC for PDF version of docs.

Tests

- bpo-38325 [https://bugs.python.org/issue?@action=redirect&bpo=38325]: Skip tests on non-BMP characters of test_winconsoleio.
- bpo-39502 [https://bugs.python.org/issue?@action=redirect&bpo=39502]: Skip test_zipfile.test_add_file_after_2107() if `time.localtime()` fails with `OverflowError`. It is the case on AIX 6.1 for example.

Build

- bpo-39489 [https://bugs.python.org/issue?@action=redirect&bpo=39489]: Remove `COUNT_ALLOCS` special build.

Windows

- bpo-39553 [https://bugs.python.org/issue?@action=redirect&bpo=39553]: Delete unused code related to

SxS manifests.
- bpo-39439 [https://bugs.python.org/issue?@action=redirect&bpo=39439]: Honor the Python path when a virtualenv is active on Windows.
- bpo-39393 [https://bugs.python.org/issue?@action=redirect&bpo=39393]: Improve the error message when attempting to load a DLL with unresolved dependencies.
- bpo-38883 [https://bugs.python.org/issue?@action=redirect&bpo=38883]: `home()` and `expanduser()` on Windows now prefer **`USERPROFILE`** and no longer use **`HOME`**, which is not normally set for regular user accounts. This makes them again behave like `os.path.expanduser()`, which was changed to ignore **`HOME`** in 3.8, see bpo-36264 [https://bugs.python.org/issue?@action=redirect&bpo=36264].
- bpo-39185 [https://bugs.python.org/issue?@action=redirect&bpo=39185]: The build.bat script has additional options for very-quiet output (-q) and very-verbose output (-vv)

IDLE

- bpo-39663 [https://bugs.python.org/issue?@action=redirect&bpo=39663]: Add tests for pyparse find_good_parse_start().
- bpo-39600 [https://bugs.python.org/issue?@action=redirect&bpo=39600]: In the font configuration window, remove duplicated font names.
- bpo-30780 [https://bugs.python.org/issue?@action=redirect&bpo=30780]: Add remaining configdialog tests for buttons and highlights and keys tabs.
- bpo-39388 [https://bugs.python.org/issue?@action=redirect&bpo=39388]: IDLE Settings Cancel button now cancels pending changes
- bpo-38792 [https://bugs.python.org/issue?@action=redirect&bpo=38792]: Close an IDLE shell calltip if a

`KeyboardInterrupt` or shell restart occurs. Patch by Zackery Spytz.

C API

- bpo-35081 [https://bugs.python.org/issue?@action=redirect&bpo=35081]: Move the `bytes_methods.h` header file to the internal C API as `pycore_bytes_methods.h`: it only contains private symbols (prefixed by `_Py`), except of the `PyDoc_STRVAR_shared()` macro.
- bpo-35081 [https://bugs.python.org/issue?@action=redirect&bpo=35081]: Move the `dtoa.h` header file to the internal C API as `pycore_dtoa.h`: it only contains private functions (prefixed by `_Py`). The `math` and `cmath` modules must now be compiled with the `Py_BUILD_CORE` macro defined.
- bpo-39573 [https://bugs.python.org/issue?@action=redirect&bpo=39573]: Add `Py_SET_SIZE()` function to set the size of an object.
- bpo-39500 [https://bugs.python.org/issue?@action=redirect&bpo=39500]: `PyUnicode_IsIdentifier()` does not call `Py_FatalError()` anymore if the string is not ready.
- bpo-39573 [https://bugs.python.org/issue?@action=redirect&bpo=39573]: Add `Py_SET_TYPE()` function to set the type of an object.
- bpo-39573 [https://bugs.python.org/issue?@action=redirect&bpo=39573]: Add a `Py_SET_REFCNT()` function to set the reference counter of an object.
- bpo-39542 [https://bugs.python.org/issue?@action=redirect&bpo=39542]: Convert `PyType_HasFeature()`, `PyType_Check()` and `PyType_CheckExact()` macros to static inline functions.
- bpo-39542 [https://bugs.python.org/issue?@action=redirect&bpo=39542]: In the limited C API, `PyObject_INIT()` and `PyObject_INIT_VAR()` are now defined as aliases to `PyObject_Init()` and `PyObject_InitVar()` to

make their implementation opaque. It avoids to leak implementation details in the limited C API. Exclude the following functions from the limited C API: `_Py_NewReference()`, `_Py_ForgetReference()`, `_PyTraceMalloc_NewReference()` and `_Py_GetRefTotal()`.

- bpo-39542 [https://bugs.python.org/issue?@action=redirect&bpo=39542]: Exclude trashcan mechanism from the limited C API: it requires access to PyTypeObject and PyThreadState structure fields, whereas these structures are opaque in the limited C API.
- bpo-39511 [https://bugs.python.org/issue?@action=redirect&bpo=39511]: The **`PyThreadState_Clear()`** function now calls the **`PyThreadState.on_delete`** callback. Previously, that happened in **`PyThreadState_Delete()`**.
- bpo-38076 [https://bugs.python.org/issue?@action=redirect&bpo=38076]: Fix to clear the interpreter state only after clearing module globals to guarantee module state access from C Extensions during runtime destruction
- bpo-39245 [https://bugs.python.org/issue?@action=redirect&bpo=39245]: The Vectorcall API (PEP 590) was made public, adding the functions `PyObject_Vectorcall`, `PyObject_VectorcallMethod`, `PyVectorcall_Function`, `PyObject_CallOneArg`, `PyObject_CallMethodNoArgs`, `PyObject_CallMethodOneArg`, `PyObject_FastCallDict`, and the flag `Py_TPFLAGS_HAVE_VECTORCALL`.

Python 3.9.0 alpha 3

Release date: 2020-01-24

Core and Builtins

- bpo-39427 [https://bugs.python.org/issue?@action=redirect&bpo=39427]: Document all possibilities for the `-X` options in the command line help section. Patch by Pablo Galindo.

- bpo-39421 [https://bugs.python.org/issue?@action=redirect&bpo=39421]: Fix possible crashes when operating with the functions in the **`heapq`** module and custom comparison operators.

- bpo-39386 [https://bugs.python.org/issue?@action=redirect&bpo=39386]: Prevent double awaiting of async iterator.

- bpo-17005 [https://bugs.python.org/issue?@action=redirect&bpo=17005]: Add **`functools.TopologicalSorter`** to the **`functools`** module to offers functionality to perform topological sorting of graphs. Patch by Pablo Galindo, Tim Peters and Larry Hastings.

- bpo-39320 [https://bugs.python.org/issue?@action=redirect&bpo=39320]: Replace four complex bytecodes for building sequences with three simpler ones.

  The following four bytecodes have been removed:

  - BUILD_LIST_UNPACK
  - BUILD_TUPLE_UNPACK
  - BUILD_SET_UNPACK
  - BUILD_TUPLE_UNPACK_WITH_CALL

  The following three bytecodes have been added:

  - LIST_TO_TUPLE
  - LIST_EXTEND
  - SET_UPDATE

- bpo-39336 [https://bugs.python.org/issue?@action=redirect&bpo=39336]: Import loaders which publish immutable module objects can now publish immutable packages in addition to individual modules.
- bpo-39322 [https://bugs.python.org/issue?@action=redirect&bpo=39322]: Added a new function `gc.is_finalized()` to check if an object has been finalized by the garbage collector. Patch by Pablo Galindo.
- bpo-39048 [https://bugs.python.org/issue?@action=redirect&bpo=39048]: Improve the displayed error message when incorrect types are passed to `async with` statements by looking up the **`__aenter__()`** special method before the **`__aexit__()`** special method when entering an asynchronous context manager. Patch by Géry Ogam.
- bpo-39235 [https://bugs.python.org/issue?@action=redirect&bpo=39235]: Fix AST end location for lone generator expression in function call, e.g. f(i for i in a).
- bpo-39209 [https://bugs.python.org/issue?@action=redirect&bpo=39209]: Correctly handle multi-line tokens in interactive mode. Patch by Pablo Galindo.
- bpo-1635741 [https://bugs.python.org/issue?@action=redirect&bpo=1635741]: Port _json extension module to multiphase initialization (**PEP 489** [https://peps.python.org/pep-0489/]).
- bpo-39216 [https://bugs.python.org/issue?@action=redirect&bpo=39216]: Fix constant folding optimization for positional only arguments - by Anthony Sottile.

- bpo-39215 [https://bugs.python.org/issue?@action=redirect&bpo=39215]: Fix `SystemError` when nested function has annotation on positional-only argument - by Anthony Sottile.

- bpo-39200 [https://bugs.python.org/issue?@action=redirect&bpo=39200]: Correct the error message when calling the `min()` or `max()` with no arguments. Patch by Donghee Na.

- bpo-39200 [https://bugs.python.org/issue?@action=redirect&bpo=39200]: Correct the error message when trying to construct `range` objects with no arguments. Patch by Pablo Galindo.

- bpo-39166 [https://bugs.python.org/issue?@action=redirect&bpo=39166]: Fix incorrect line execution reporting in trace functions when tracing the last iteration of asynchronous for loops. Patch by Pablo Galindo.

- bpo-39114 [https://bugs.python.org/issue?@action=redirect&bpo=39114]: Fix incorrect line execution reporting in trace functions when tracing exception handlers with name binding. Patch by Pablo Galindo.

- bpo-39156 [https://bugs.python.org/issue?@action=redirect&bpo=39156]: Split the COMPARE_OP bytecode instruction into four distinct instructions.

  - COMPARE_OP for rich comparisons
  - IS_OP for ‘is’ and ‘is not’ tests
  - CONTAINS_OP for ‘in’ and ‘is not’ tests
  - JUMP_IF_NOT_EXC_MATCH for checking exceptions in ‘try-except’ statements.

This improves the clarity of the interpreter and should provide a modest speedup.

- bpo-38588 [https://bugs.python.org/issue?@action=redirect&bpo=38588]: Fix possible crashes in dict and list when calling `PyObject_RichCompareBool()`.
- bpo-13601 [https://bugs.python.org/issue?@action=redirect&bpo=13601]: By default, `sys.stderr` is line-buffered now, even if `stderr` is redirected to a file. You can still make `sys.stderr` unbuffered by passing the `-u` command-line option or setting the `PYTHONUNBUFFERED` environment variable.

  (Contributed by Jendrik Seipp in bpo-13601 [https://bugs.python.org/issue?@action=redirect&bpo=13601].)
- bpo-38610 [https://bugs.python.org/issue?@action=redirect&bpo=38610]: Fix possible crashes in several list methods by holding strong references to list elements when calling `PyObject_RichCompareBool()`.
- bpo-32021 [https://bugs.python.org/issue?@action=redirect&bpo=32021]: Include brotli .br encoding in mimetypes encodings_map

Library

- bpo-39430 [https://bugs.python.org/issue?@action=redirect&bpo=39430]: Fixed race condition in lazy imports in `tarfile`.
- bpo-39413 [https://bugs.python.org/issue?@action=redirect&bpo=39413]: The `os.unsetenv()` function is now also available on Windows.
- bpo-39390 [https://bugs.python.org/issue?@action=redirect&bpo=39390]: Fixed a regression with the

`ignore` callback of **`shutil.copytree()`**. The argument types are now `str` and `List[str]` again.

- bpo-39395 [https://bugs.python.org/issue?@action=redirect&bpo=39395]: The **`os.putenv()`** and **`os.unsetenv()`** functions are now always available.
- bpo-39406 [https://bugs.python.org/issue?@action=redirect&bpo=39406]: If `setenv()` C function is available, **`os.putenv()`** is now implemented with `setenv()` instead of `putenv()`, so Python doesn’t have to handle the environment variable memory.
- bpo-39396 [https://bugs.python.org/issue?@action=redirect&bpo=39396]: Fix `math.nextafter(-0.0, +0.0)` on AIX 7.1.
- bpo-29435 [https://bugs.python.org/issue?@action=redirect&bpo=29435]: Allow **`tarfile.is_tarfile()`** to be used with file and file-like objects, like **`zipfile.is_zipfile()`**. Patch by William Woodruff.
- bpo-39377 [https://bugs.python.org/issue?@action=redirect&bpo=39377]: Removed `encoding` option from **`json.loads()`**. It has been deprecated since Python 3.1.
- bpo-39389 [https://bugs.python.org/issue?@action=redirect&bpo=39389]: Write accurate compression level metadata in **`gzip`** archives, rather than always signaling maximum compression.
- bpo-39366 [https://bugs.python.org/issue?@action=redirect&bpo=39366]: The previously deprecated `xpath()` and `xgtitle()` methods of **`nntplib.NNTP`** have been removed.
- bpo-39357 [https://bugs.python.org/issue?@action=redirect&bpo=39357]: Remove the *buffering* parameter of **`bz2.BZ2File`**. Since Python 3.0, it was ignored and using it was emitting **`DeprecationWarning`**. Pass an open file object, to control how the file is opened. The *compresslevel* parameter becomes keyword-only.

- bpo-39353 [https://bugs.python.org/issue?@action=redirect&bpo=39353]: Deprecate binhex4 and hexbin4 standards. Deprecate the **`binhex`** module and the following **`binascii`** functions: **`b2a_hqx()`**, **`a2b_hqx()`**, **`rlecode_hqx()`**, **`rledecode_hqx()`**, **`crc_hqx()`**.
- bpo-39351 [https://bugs.python.org/issue?@action=redirect&bpo=39351]: Remove `base64.encodestring()` and `base64.decodestring()`, aliases deprecated since Python 3.1: use **`base64.encodebytes()`** and **`base64.decodebytes()`** instead.
- bpo-39350 [https://bugs.python.org/issue?@action=redirect&bpo=39350]: Remove `fractions.gcd()` function, deprecated since Python 3.5 (bpo-22486 [https://bugs.python.org/issue?@action=redirect&bpo=22486]): use **`math.gcd()`** instead.
- bpo-39329 [https://bugs.python.org/issue?@action=redirect&bpo=39329]: **`LMTP`** constructor now has an optional *timeout* parameter. Patch by Donghee Na.
- bpo-39313 [https://bugs.python.org/issue?@action=redirect&bpo=39313]: Add a new `exec_function` option (*–exec-function* in the CLI) to `RefactoringTool` for making `exec` a function. Patch by Batuhan Taskaya.
- bpo-39259 [https://bugs.python.org/issue?@action=redirect&bpo=39259]: **`FTP_TLS`** and **`FTP_TLS`** now raise a **`ValueError`** if the given timeout for their constructor is zero to prevent the creation of a non-blocking socket. Patch by Donghee Na.
- bpo-39259 [https://bugs.python.org/issue?@action=redirect&bpo=39259]: **`SMTP`** and **`SMTP_SSL`** now raise a **`ValueError`** if the given timeout for their constructor is zero to prevent the creation of a non-blocking socket. Patch by Donghee Na.
- bpo-39310 [https://bugs.python.org/issue?@action=redirect&bpo=39310]: Add **`math.ulp()`**: return the value of the least significant bit of a float.

- bpo-39297 [https://bugs.python.org/issue?@action=redirect&bpo=39297]: Improved performance of importlib.metadata distribution discovery and resilients to inaccessible sys.path entries (importlib_metadata v1.4.0).
- bpo-39259 [https://bugs.python.org/issue?@action=redirect&bpo=39259]: **`NNTP`** and **`NNTP_SSL`** now raise a **`ValueError`** if the given timeout for their constructor is zero to prevent the creation of a non-blocking socket. Patch by Donghee Na.
- bpo-38901 [https://bugs.python.org/issue?@action=redirect&bpo=38901]: When you specify prompt='.' or equivalently python -m venv –prompt . … the basename of the current directory is used to set the created venv's prompt when it's activated.
- bpo-39288 [https://bugs.python.org/issue?@action=redirect&bpo=39288]: Add **`math.nextafter()`**: return the next floating-point value after *x* towards *y*.
- bpo-39259 [https://bugs.python.org/issue?@action=redirect&bpo=39259]: **`POP3`** and **`POP3_SSL`** now raise a **`ValueError`** if the given timeout for their constructor is zero to prevent the creation of a non-blocking socket. Patch by Donghee Na.
- bpo-39242 [https://bugs.python.org/issue?@action=redirect&bpo=39242]: Updated the Gmane domain from news.gmane.org to news.gmane.io which is used for examples of **`NNTP`** news reader server and nntplib tests.
- bpo-35292 [https://bugs.python.org/issue?@action=redirect&bpo=35292]: Proxy the `SimpleHTTPRequestHandler.guess_type` to `mimetypes.guess_type` so the `mimetypes.init` is called lazily to avoid unnecessary costs when **`http.server`** module is imported.
- bpo-39239 [https://bugs.python.org/issue?@action=redirect&bpo=39239]: The **`select.epoll.unregister()`** method no longer ignores the **`EBADF`** error.

- bpo-38907 [https://bugs.python.org/issue?@action=redirect&bpo=38907]: In http.server script, restore binding to IPv4 on Windows.
- bpo-39152 [https://bugs.python.org/issue?@action=redirect&bpo=39152]: Fix ttk.Scale.configure([name]) to return configuration tuple for name or all options. Giovanni Lombardo contributed part of the patch.
- bpo-39198 [https://bugs.python.org/issue?@action=redirect&bpo=39198]: If an exception were to be thrown in `Logger.isEnabledFor` (say, by asyncio timeouts or stopit) , the `logging` global lock may not be released appropriately, resulting in deadlock. This change wraps that block of code with `try...finally` to ensure the lock is released.
- bpo-39191 [https://bugs.python.org/issue?@action=redirect&bpo=39191]: Perform a check for running loop before starting a new task in `loop.run_until_complete()` to fail fast; it prevents the side effect of new task spawning before exception raising.
- bpo-38871 [https://bugs.python.org/issue?@action=redirect&bpo=38871]: Correctly parenthesize filter-based statements that contain lambda expressions in **`lib2to3`**. Patch by Donghee Na.
- bpo-39142 [https://bugs.python.org/issue?@action=redirect&bpo=39142]: A change was made to logging.config.dictConfig to avoid converting instances of named tuples to ConvertingTuple. It’s assumed that named tuples are too specialised to be treated like ordinary tuples; if a user of named tuples requires ConvertingTuple functionality, they will have to implement that themselves in their named tuple class.
- bpo-39158 [https://bugs.python.org/issue?@action=redirect&bpo=39158]: ast.literal_eval() now supports empty sets.
- bpo-39129 [https://bugs.python.org/issue?@action=redirect&bpo=39129]: Fix import path for

`asyncio.TimeoutError`

- bpo-39057 [https://bugs.python.org/issue?@action=redirect&bpo=39057]: **`urllib.request.proxy_bypass_environment()`** now ignores leading dots and no longer ignores a trailing newline.
- bpo-39056 [https://bugs.python.org/issue?@action=redirect&bpo=39056]: Fixed handling invalid warning category in the -W option. No longer import the re module if it is not needed.
- bpo-39055 [https://bugs.python.org/issue?@action=redirect&bpo=39055]: **`base64.b64decode()`** with `validate=True` raises now a binascii.Error if the input ends with a single `\n`.
- bpo-21600 [https://bugs.python.org/issue?@action=redirect&bpo=21600]: Fix **`mock.patch.stopall()`** to stop active patches that were created with **`mock.patch.dict()`**.
- bpo-39019 [https://bugs.python.org/issue?@action=redirect&bpo=39019]: Implement dummy `__class_getitem__` for **`tempfile.SpooledTemporaryFile`**.
- bpo-39019 [https://bugs.python.org/issue?@action=redirect&bpo=39019]: Implement dummy `__class_getitem__` for `subprocess.Popen`, `subprocess.CompletedProcess`
- bpo-38914 [https://bugs.python.org/issue?@action=redirect&bpo=38914]: Adjusted the wording of the warning issued by distutils' `check` command when the `author` and `maintainer` fields are supplied but no corresponding e-mail field (`author_email` or `maintainer_email`) is found. The wording now reflects the fact that these fields are suggested, but not required. Patch by Juergen Gmach.
- bpo-38878 [https://bugs.python.org/issue?@action=redirect&bpo=38878]: Fixed __subclasshook__ of **`os.PathLike`** to return a correct result upon inheritance. Patch by Bar Harel.

- bpo-38615 [https://bugs.python.org/issue?@action=redirect&bpo=38615]: **`IMAP4`** and **`IMAP4_SSL`** now have an optional *timeout* parameter for their constructors. Also, the **`open()`** method now has an optional *timeout* parameter with this change. The overridden methods of **`IMAP4_SSL`** and **`IMAP4_stream`** were applied to this change. Patch by Donghee Na.
- bpo-35182 [https://bugs.python.org/issue?@action=redirect&bpo=35182]: Fixed **`Popen.communicate()`** subsequent call crash when the child process has already closed any piped standard stream, but still continues to be running. Patch by Andriy Maletsky.
- bpo-38630 [https://bugs.python.org/issue?@action=redirect&bpo=38630]: On Unix, **`subprocess.Popen.send_signal()`** now polls the process status. Polling reduces the risk of sending a signal to the wrong process if the process completed, the **`subprocess.Popen.returncode`** attribute is still `None`, and the pid has been reassigned (recycled) to a new different process.
- bpo-38536 [https://bugs.python.org/issue?@action=redirect&bpo=38536]: Removes trailing space in formatted currency with `international=True` and a locale with symbol following value. E.g. `locale.currency(12.34, international=True)` returned `'12,34 EUR '` instead of `'12,34 EUR'`.
- bpo-38473 [https://bugs.python.org/issue?@action=redirect&bpo=38473]: Use signature from inner mock for autospecced methods attached with **`unittest.mock.attach_mock()`**. Patch by Karthikeyan Singaravelan.
- bpo-38361 [https://bugs.python.org/issue?@action=redirect&bpo=38361]: Fixed an issue where `ident` could include a leading path separator when **`syslog.openlog()`** was called without arguments.

- bpo-38293 [https://bugs.python.org/issue?@action=redirect&bpo=38293]: Add **`copy.copy()`** and **`copy.deepcopy()`** support to **`property()`** objects.
- bpo-37958 [https://bugs.python.org/issue?@action=redirect&bpo=37958]: Added the pstats.Stats.get_profile_dict() method to return the profile data as a StatsProfile instance.
- bpo-28367 [https://bugs.python.org/issue?@action=redirect&bpo=28367]: Termios magic constants for the following baud rates: - B500000 - B576000 - B921600 - B1000000 - B1152000 - B1500000 - B2000000 - B2500000 - B3000000 - B3500000 - B4000000 Patch by Andrey Smirnov

Documentation

- bpo-39381 [https://bugs.python.org/issue?@action=redirect&bpo=39381]: Mention in docs that **`asyncio.get_event_loop()`** implicitly creates new event loop only if called from the main thread.
- bpo-38918 [https://bugs.python.org/issue?@action=redirect&bpo=38918]: Add an entry for `__module__` in the “function” & “method” sections of the **`inspect`** docs’ Types and members table.
- bpo-3530 [https://bugs.python.org/issue?@action=redirect&bpo=3530]: In the **`ast`** module documentation, fix a misleading `NodeTransformer` example and add advice on when to use the `fix_missing_locations` function.

Build

- bpo-39395 [https://bugs.python.org/issue?@action=redirect&bpo=39395]: On non-Windows platforms, the **`setenv()`** and **`unsetenv()`** functions are now required to build Python.

- bpo-39160 [https://bugs.python.org/issue?@action=redirect&bpo=39160]: Updated the documentation in `./configure --help` to show default values, reference documentation where required and add additional explanation where needed.
- bpo-39144 [https://bugs.python.org/issue?@action=redirect&bpo=39144]: The ctags and etags build targets both include Modules/_ctypes and Python standard library source files.

IDLE

- bpo-39050 [https://bugs.python.org/issue?@action=redirect&bpo=39050]: Make IDLE Settings dialog Help button work again.
- bpo-34118 [https://bugs.python.org/issue?@action=redirect&bpo=34118]: Tag memoryview, range, and tuple as classes, the same as list, etcetera, in the library manual built-in functions list.
- bpo-32989 [https://bugs.python.org/issue?@action=redirect&bpo=32989]: Add tests for editor newline_and_indent_event method. Remove dead code from pyparse find_good_parse_start method.

C API

- bpo-39372 [https://bugs.python.org/issue?@action=redirect&bpo=39372]: Clean header files of interfaces defined but with no implementation. The public API symbols being removed are: `_PyBytes_InsertThousandsGroupingLocale`, `_PyBytes_InsertThousandsGrouping`, `_Py_InitializeFromArgs`, `_Py_InitializeFromWideArgs`, `_PyFloat_Repr`, `_PyFloat_Digits`, `_PyFloat_DigitsInit`, `PyFrame_ExtendStack`, `_PyAIterWrapper_Type`, `PyNullImporter_Type`, `PyCmpWrapper_Type`, `PySortWrapper_Type`, `PyNoArgsFunction`.

- bpo-39164 [https://bugs.python.org/issue?@action=redirect&bpo=39164]: Add a private `_PyErr_GetExcInfo()` function to retrieve exception information of the specified Python thread state.

Python 3.9.0 alpha 2

Release date: 2019-12-18

Security

- bpo-38945 [https://bugs.python.org/issue?@action=redirect&bpo=38945]: Newline characters have been escaped when performing uu encoding to prevent them from overflowing into to content section of the encoded file. This prevents malicious or accidental modification of data during the decoding process.
- bpo-37228 [https://bugs.python.org/issue?@action=redirect&bpo=37228]: Due to significant security concerns, the *reuse_address* parameter of **`asyncio.loop.create_datagram_endpoint()`** is no longer supported. This is because of the behavior of `SO_REUSEADDR` in UDP. For more details, see the documentation for `loop.create_datagram_endpoint()`. (Contributed by Kyle Stanley, Antoine Pitrou, and Yury Selivanov in bpo-37228 [https://bugs.python.org/issue?@action=redirect&bpo=37228].)
- bpo-38804 [https://bugs.python.org/issue?@action=redirect&bpo=38804]: Fixes a ReDoS vulnerability in **`http.cookiejar`**. Patch by Ben Caller.

Core and Builtins

- bpo-39028 [https://bugs.python.org/issue?@action=redirect&bpo=39028]: Slightly improve the speed of keyword argument parsing with many kwargs by strengthening the assumption that kwargs are interned strings.

- bpo-39080 [https://bugs.python.org/issue?@action=redirect&bpo=39080]: Fix the value of *end_col_offset* for Starred Expression AST nodes when they are among the elements in the *args* attribute of Call AST nodes.
- bpo-39031 [https://bugs.python.org/issue?@action=redirect&bpo=39031]: When parsing an “elif” node, lineno and col_offset of the node now point to the “elif” keyword and not to its condition, making it consistent with the “if” node. Patch by Lysandros Nikolaou.
- bpo-20443 [https://bugs.python.org/issue?@action=redirect&bpo=20443]: In Python 3.9.0a1, sys.argv[0] was made an absolute path if a filename was specified on the command line. Revert this change, since most users expect sys.argv to be unmodified.
- bpo-39008 [https://bugs.python.org/issue?@action=redirect&bpo=39008]: **`PySys_Audit()`** now requires `Py_ssize_t` to be used for size arguments in the format string, regardless of whether `PY_SSIZE_T_CLEAN` was defined at include time.
- bpo-38673 [https://bugs.python.org/issue?@action=redirect&bpo=38673]: In REPL mode, don’t switch to PS2 if the line starts with comment or whitespace. Based on work by Batuhan Taşkaya.
- bpo-38922 [https://bugs.python.org/issue?@action=redirect&bpo=38922]: Calling `replace` on a code object now raises the `code.__new__` audit event.
- bpo-38920 [https://bugs.python.org/issue?@action=redirect&bpo=38920]: Add audit hooks for when **`sys.excepthook()`** and **`sys.unraisablehook()`** are invoked.
- bpo-38892 [https://bugs.python.org/issue?@action=redirect&bpo=38892]: Improve documentation for audit events table and functions.

- bpo-38852 [https://bugs.python.org/issue?@action=redirect&bpo=38852]: Set the thread stack size to 8 Mb for debug builds on android platforms.
- bpo-38858 [https://bugs.python.org/issue?@action=redirect&bpo=38858]: Each Python subinterpreter now has its own “small integer singletons”: numbers in [-5; 257] range. It is no longer possible to change the number of small integers at build time by overriding `NSMALLNEGINTS` and `NSMALLPOSINTS` macros: macros should now be modified manually in `pycore_pystate.h` header file.
- bpo-36854 [https://bugs.python.org/issue?@action=redirect&bpo=36854]: The garbage collector state becomes per interpreter (`PyInterpreterState.gc`), rather than being global (`_PyRuntimeState.gc`).
- bpo-38835 [https://bugs.python.org/issue?@action=redirect&bpo=38835]: The `PyFPE_START_PROTECT()` and `PyFPE_END_PROTECT()` macros are empty: they have been doing nothing for the last year, so stop using them.
- bpo-38328 [https://bugs.python.org/issue?@action=redirect&bpo=38328]: Sped up the creation time of constant `list` and `set` displays. Patch by Brandt Bucher.
- bpo-38707 [https://bugs.python.org/issue?@action=redirect&bpo=38707]: `MainThread.native_id` is now correctly reset in child processes spawned using `multiprocessing.Process`, instead of retaining the parent’s value.
- bpo-38629 [https://bugs.python.org/issue?@action=redirect&bpo=38629]: Added `__floor__` and `__ceil__` methods to float object. Patch by Batuhan Taşkaya.
- bpo-27145 [https://bugs.python.org/issue?@action=redirect&bpo=27145]: int + int and int - int operators can now return small integer singletons. Patch by hongweipeng.
- bpo-38021 [https://bugs.python.org/issue?@action=redirect&bpo=38021]: Provide a platform tag for AIX

that is sufficient for PEP425 binary distribution identification. Patch by Michael Felt.

- bpo-35409 [https://bugs.python.org/issue?@action=redirect&bpo=35409]: Ignore GeneratorExit exceptions when throwing an exception into the aclose coroutine of an asynchronous generator.
- bpo-33387 [https://bugs.python.org/issue?@action=redirect&bpo=33387]: Removed WITH_CLEANUP_START, WITH_CLEANUP_FINISH, BEGIN_FINALLY, END_FINALLY, CALL_FINALLY and POP_FINALLY bytecodes. Replaced with RERAISE and WITH_EXCEPT_START bytecodes. The compiler now generates different code for exceptional and non-exceptional branches for ‘with’ and ‘try-except’ statements. For ‘try-finally’ statements the ‘finally’ block is replaced for each exit from the ‘try’ body.

Library

- bpo-39033 [https://bugs.python.org/issue?@action=redirect&bpo=39033]: Fix `NameError` in `zipimport`. Patch by Karthikeyan Singaravelan.

- bpo-39022 [https://bugs.python.org/issue?@action=redirect&bpo=39022]: Update importlib.metadata to include improvements from importlib_metadata 1.3 including better serialization of EntryPoints and improved documentation for custom finders.

- bpo-39006 [https://bugs.python.org/issue?@action=redirect&bpo=39006]: Fix asyncio when the ssl module is missing: only check for ssl.SSLSocket instance if the ssl module is available.

- bpo-38708 [https://bugs.python.org/issue?@action=redirect&bpo=38708]: Fix a potential IndexError in email parser when parsing an empty msg-id.

- bpo-38698 [https://bugs.python.org/issue?@action=redirect&bpo=38698]: Add a new `InvalidMessageID` token to email parser to represent invalid Message-ID headers. Also, add defects when there is remaining value after parsing the header.
- bpo-38994 [https://bugs.python.org/issue?@action=redirect&bpo=38994]: Implement `__class_getitem__` for `os.PathLike`, `pathlib.Path`.
- bpo-38979 [https://bugs.python.org/issue?@action=redirect&bpo=38979]: Return class from `ContextVar.__class_getitem__` to simplify subclassing.
- bpo-38978 [https://bugs.python.org/issue?@action=redirect&bpo=38978]: Implement `__class_getitem__` on asyncio objects (Future, Task, Queue). Patch by Batuhan Taskaya.
- bpo-38916 [https://bugs.python.org/issue?@action=redirect&bpo=38916]: **`array.array`**: Remove `tostring()` and `fromstring()` methods. They were aliases to `tobytes()` and `frombytes()`, deprecated since Python 3.2.
- bpo-38986 [https://bugs.python.org/issue?@action=redirect&bpo=38986]: Make repr of C accelerated TaskWakeupMethWrapper the same as of pure Python version.
- bpo-38982 [https://bugs.python.org/issue?@action=redirect&bpo=38982]: Fix asyncio `PidfdChildWatcher`: handle `waitpid()` error. If `waitpid()` is called elsewhere, `waitpid()` call fails with **`ChildProcessError`**: use return code 255 in this case, and log a warning. It ensures that the pidfd file descriptor is closed if this error occurs.

- bpo-38529 [https://bugs.python.org/issue?@action=redirect&bpo=38529]: Drop too noisy asyncio warning about deletion of a stream without explicit `.close()` call.
- bpo-27413 [https://bugs.python.org/issue?@action=redirect&bpo=27413]: Added ability to pass through `ensure_ascii` options to json.dumps in the `json.tool` command-line interface.
- bpo-38634 [https://bugs.python.org/issue?@action=redirect&bpo=38634]: The **`readline`** module now detects if Python is linked to libedit at runtime on all platforms. Previously, the check was only done on macOS.
- bpo-33684 [https://bugs.python.org/issue?@action=redirect&bpo=33684]: Fix `json.tool` failed to read a JSON file with non-ASCII characters when locale encoding is not UTF-8.
- bpo-38698 [https://bugs.python.org/issue?@action=redirect&bpo=38698]: Prevent UnboundLocalError to pop up in parse_message_id.

  parse_message_id() was improperly using a token defined inside an exception handler, which was raising `UnboundLocalError` on parsing an invalid value. Patch by Claudiu Popa.
- bpo-38927 [https://bugs.python.org/issue?@action=redirect&bpo=38927]: Use `python -m pip` instead of `pip` to upgrade dependencies in venv.
- bpo-26730 [https://bugs.python.org/issue?@action=redirect&bpo=26730]: Fix `SpooledTemporaryFile.rollover()` might corrupt the file when it is in text mode. Patch by Serhiy Storchaka.

- bpo-38881 [https://bugs.python.org/issue?@action=redirect&bpo=38881]: random.choices() now raises a ValueError when all the weights are zero.

- bpo-38876 [https://bugs.python.org/issue?@action=redirect&bpo=38876]: Raise pickle.UnpicklingError when loading an item from memo for invalid input.

  The previous code was raising a `KeyError` for both the Python and C implementation. This was caused by the specified index of an invalid input which did not exist in the memo structure, where the pickle stores what objects it has seen. The malformed input would have caused either a `BINGET` or `LONG_BINGET` load from the memo, leading to a `KeyError` as the determined index was bogus. Patch by Claudiu Popa.

- bpo-38688 [https://bugs.python.org/issue?@action=redirect&bpo=38688]: Calling func:`shutil.copytree` to copy a directory tree from one directory to another subdirectory resulted in an endless loop and a RecursionError. A fix was added to consume an iterator and create the list of the entries to be copied, avoiding the recursion for newly created directories. Patch by Bruno P. Kinoshita.

- bpo-38863 [https://bugs.python.org/issue?@action=redirect&bpo=38863]: Improve **`is_cgi()`** function in **`http.server`**, which enables processing the case that cgi directory is a child of another directory other than root.

- bpo-37838 [https://bugs.python.org/issue?@action=redirect&bpo=37838]: **`typing.get_type_hints()`** properly handles functions decorated with **`functools.wraps()`**.

- bpo-38870 [https://bugs.python.org/issue?@action=redirect&bpo=38870]: Expose **`ast.unparse()`** as a function of the **`ast`** module that can be used to unparse an **`ast.AST`** object and produce a string with code that would produce an equivalent **`ast.AST`** object when parsed. Patch by Pablo Galindo and Batuhan Taskaya.

- bpo-38859 [https://bugs.python.org/issue?@action=redirect&bpo=38859]: AsyncMock now returns StopAsyncIteration on the exhaustion of a side_effects iterable. Since PEP-479 its Impossible to raise a StopIteration exception from a coroutine.

- bpo-38857 [https://bugs.python.org/issue?@action=redirect&bpo=38857]: AsyncMock fix for return values that are awaitable types. This also covers side_effect iterable values that happened to be awaitable, and wraps callables that return an awaitable type. Before these awaitables were being awaited instead of being returned as is.

- bpo-38834 [https://bugs.python.org/issue?@action=redirect&bpo=38834]: **`typing.TypedDict`** subclasses now track which keys are optional using the `__required_keys__` and `__optional_keys__` attributes, to enable runtime validation by downstream projects. Patch by Zac Hatfield-Dodds.

- bpo-38821 [https://bugs.python.org/issue?@action=redirect&bpo=38821]: Fix unhandled exceptions in **`argparse`** when internationalizing error messages for arguments with `nargs` set to special (non-integer) values. Patch by Federico Bond.

- bpo-38820 [https://bugs.python.org/issue?@action=redirect&bpo=38820]: Make Python compatible with OpenSSL 3.0.0. **`ssl.SSLSocket.getpeercert()`** no longer returns IPv6 addresses with a trailing new line.

- bpo-38811 [https://bugs.python.org/issue?@action=redirect&bpo=38811]: Fix an unhandled exception in **`pathlib`** when **`os.link()`** is missing. Patch by Toke Høiland-Jørgensen.

- bpo-38686 [https://bugs.python.org/issue?@action=redirect&bpo=38686]: Added support for multiple `qop` values in **`urllib.request.AbstractDigestAuthHandler`**.

- bpo-38712 [https://bugs.python.org/issue?@action=redirect&bpo=38712]: Add the Linux-specific **`signal.pidfd_send_signal()`** function, which allows sending a signal to a process identified by a file descriptor rather than a pid.

- bpo-38348 [https://bugs.python.org/issue?@action=redirect&bpo=38348]: Add `-i` and `--indent` (indentation level), and `--no-type-comments` (type comments) command line options to ast parsing tool.

- bpo-37523 [https://bugs.python.org/issue?@action=redirect&bpo=37523]: Change **`zipfile.ZipExtFile`** to raise `ValueError` when trying to access the underlying file object after it has been closed. This new behavior is consistent with how accessing closed files is handled in other parts of Python.

- bpo-38045 [https://bugs.python.org/issue?@action=redirect&bpo=38045]: Improve the performance of **`enum._decompose()`** in **`enum`**. Patch by hongweipeng.

- bpo-36820 [https://bugs.python.org/issue?@action=redirect&bpo=36820]: Break cycle generated when saving an exception in socket.py, codeop.py and dyld.py as they keep alive not only the exception but user objects through the `__traceback__` attribute. Patch by Mario Corchero.

- bpo-36406 [https://bugs.python.org/issue?@action=redirect&bpo=36406]: Handle namespace packages in **`doctest`**. Patch by Karthikeyan Singaravelan.

- bpo-34776 [https://bugs.python.org/issue?@action=redirect&bpo=34776]: Fix dataclasses to support forward references in type annotations

- bpo-20928 [https://bugs.python.org/issue?@action=redirect&bpo=20928]: ElementTree supports recursive XInclude processing. Patch by Stefan Behnel.

- bpo-29636 [https://bugs.python.org/issue?@action=redirect&bpo=29636]: Add whitespace options for formatting JSON with the `json.tool` CLI. The following mutually exclusive options are now supported: `--indent` for setting the indent level in spaces; `--tab` for indenting with tabs; `--no-indent` for suppressing newlines; and `--compact` for suppressing all whitespace. The default behavior remains the same as `--indent=4`.

Documentation

- bpo-38928 [https://bugs.python.org/issue?@action=redirect&bpo=38928]: Correct when venv's `upgrade_dependencies()` and `--upgrade-deps` are added.
- bpo-38899 [https://bugs.python.org/issue?@action=redirect&bpo=38899]: Update documentation to state that to activate virtual environments under fish one should use `source`, not `.` as documented at https://fishshell.com/docs/current/cmds/source.html.
- bpo-22377 [https://bugs.python.org/issue?@action=redirect&bpo=22377]: Improves documentation of the values that **`datetime.datetime.strptime()`** accepts for `%Z`. Patch by Karl Dubost.

Tests

- bpo-38546 [https://bugs.python.org/issue?@action=redirect&bpo=38546]: Fix test_ressources_gced_in_workers() of test_concurrent_futures: explicitly stop the manager to prevent leaking a child process running in the background after the test completes.
- bpo-38546 [https://bugs.python.org/issue?@action=redirect&bpo=38546]: Multiprocessing and concurrent.futures tests now stop the resource tracker process when tests complete.
- bpo-38614 [https://bugs.python.org/issue?@action=redirect&bpo=38614]: Replace hardcoded timeout constants in tests with new `test.support` constants: `LOOPBACK_TIMEOUT`, `INTERNET_TIMEOUT`, `SHORT_TIMEOUT` and `LONG_TIMEOUT`. It becomes easier to adjust these four timeout constants for all tests at once, rather than having to adjust every single test file.
- bpo-38547 [https://bugs.python.org/issue?@action=redirect&bpo=38547]: Fix test_pty: if the process is the session leader, closing the master file descriptor raises a SIGHUP signal: simply ignore SIGHUP when running the tests.
- bpo-38992 [https://bugs.python.org/issue?@action=redirect&bpo=38992]: Fix a test for `math.fsum()` that was failing due to constant folding.
- bpo-38991 [https://bugs.python.org/issue?@action=redirect&bpo=38991]: `test.support`: **`run_python_until_end()`**, **`assert_python_ok()`** and **`assert_python_failure()`** functions no longer strip whitespaces from stderr. Remove `test.support.strip_python_stderr()` function.
- bpo-38965 [https://bugs.python.org/issue?@action=redirect&bpo=38965]: Fix test_faulthandler on GCC

10. Use the "volatile" keyword in `faulthandler._stack_overflow()` to prevent tail call optimization on any compiler, rather than relying on compiler specific pragma.

- bpo-38875 [https://bugs.python.org/issue?@action=redirect&bpo=38875]: test_capi: trashcan tests now require the test "cpu" resource.
- bpo-38841 [https://bugs.python.org/issue?@action=redirect&bpo=38841]: Skip asyncio test_create_datagram_endpoint_existing_sock_unix on platforms lacking a functional bind() for named unix domain sockets.
- bpo-38692 [https://bugs.python.org/issue?@action=redirect&bpo=38692]: Skip the test_posix.test_pidfd_open() test if `os.pidfd_open()` fails with a **`PermissionError`**. This situation can happen in a Linux sandbox using a syscall whitelist which doesn't allow the `pidfd_open()` syscall yet.
- bpo-38839 [https://bugs.python.org/issue?@action=redirect&bpo=38839]: Fix some unused functions in tests. Patch by Adam Johnson.
- bpo-38669 [https://bugs.python.org/issue?@action=redirect&bpo=38669]: Raise **`TypeError`** when passing target as a string with **`unittest.mock.patch.object()`**.
- bpo-37957 [https://bugs.python.org/issue?@action=redirect&bpo=37957]: test.regrtest now can receive a list of test patterns to ignore (using the -i/–ignore argument) or a file with a list of patterns to ignore (using the –ignore-file argument). Patch by Pablo Galindo.

Build

- bpo-37404 [https://bugs.python.org/issue?@action=redirect&bpo=37404]: **`asyncio`** now raises **`TypeError`** when calling incompatible methods with an `ssl.SSLSocket` socket. Patch by Ido Michael.

- bpo-36500 [https://bugs.python.org/issue?@action=redirect&bpo=36500]: Added an optional "regen" project to the Visual Studio solution that will regenerate all grammar, tokens, and opcodes.

Windows

- bpo-39007 [https://bugs.python.org/issue?@action=redirect&bpo=39007]: Add auditing events to functions in `winreg`.
- bpo-33125 [https://bugs.python.org/issue?@action=redirect&bpo=33125]: Add support for building and releasing Windows ARM64 packages.

macOS

- bpo-37931 [https://bugs.python.org/issue?@action=redirect&bpo=37931]: Fixed a crash on OSX dynamic builds that occurred when re-initializing the posix module after a Py_Finalize if the environment had changed since the previous `import posix`. Patch by Benoît Hudson.

IDLE

- bpo-38944 [https://bugs.python.org/issue?@action=redirect&bpo=38944]: Escape key now closes IDLE completion windows. Patch by Johnny Najera.
- bpo-38943 [https://bugs.python.org/issue?@action=redirect&bpo=38943]: Fix IDLE autocomplete windows not always appearing on some systems. Patch by Johnny Najera.
- bpo-38862 [https://bugs.python.org/issue?@action=redirect&bpo=38862]: 'Strip Trailing Whitespace' on

the Format menu removes extra newlines at the end of non-shell files.
- bpo-38636 [https://bugs.python.org/issue?@action=redirect&bpo=38636]: Fix IDLE Format menu tab toggle and file indent width. These functions (default shortcuts Alt-T and Alt-U) were mistakenly disabled in 3.7.5 and 3.8.0.

C API

- bpo-38896 [https://bugs.python.org/issue?@action=redirect&bpo=38896]: Remove `PyUnicode_ClearFreeList()` function: the Unicode free list has been removed in Python 3.3.
- bpo-37340 [https://bugs.python.org/issue?@action=redirect&bpo=37340]: Remove `PyMethod_ClearFreeList()` and `PyCFunction_ClearFreeList()` functions: the free lists of bound method objects have been removed.
- bpo-38835 [https://bugs.python.org/issue?@action=redirect&bpo=38835]: Exclude `PyFPE_START_PROTECT()` and `PyFPE_END_PROTECT()` macros of `pyfpe.h` from `Py_LIMITED_API` (stable API).

Python 3.9.0 alpha 1

Release date: 2019-11-19

Security

- bpo-38722 [https://bugs.python.org/issue?@action=redirect&bpo=38722]: **`runpy`** now uses **`io.open_code()`** to open code files. Patch by Jason Killen.
- bpo-38622 [https://bugs.python.org/issue?@action=redirect&bpo=38622]: Add additional audit events for the **`ctypes`** module.

- bpo-38418 [https://bugs.python.org/issue?@action=redirect&bpo=38418]: Fixes audit event for **`os.system()`** to be named `os.system`.
- bpo-38243 [https://bugs.python.org/issue?@action=redirect&bpo=38243]: Escape the server title of **`xmlrpc.server.DocXMLRPCServer`** when rendering the document page as HTML. (Contributed by Donghee Na in bpo-38243 [https://bugs.python.org/issue?@action=redirect&bpo=38243].)
- bpo-38174 [https://bugs.python.org/issue?@action=redirect&bpo=38174]: Update vendorized expat library version to 2.2.8, which resolves **CVE 2019-15903** [https://www.cve.org/CVERecord?id=CVE-2019-15903].
- bpo-37764 [https://bugs.python.org/issue?@action=redirect&bpo=37764]: Fixes email._header_value_parser.get_unstructured going into an infinite loop for a specific case in which the email header does not have trailing whitespace, and the case in which it contains an invalid encoded word. Patch by Ashwin Ramaswami.
- bpo-37461 [https://bugs.python.org/issue?@action=redirect&bpo=37461]: Fix an infinite loop when parsing specially crafted email headers. Patch by Abhilash Raj.
- bpo-37363 [https://bugs.python.org/issue?@action=redirect&bpo=37363]: Adds audit events for the range of supported run commands (see Command line and environment).
- bpo-37463 [https://bugs.python.org/issue?@action=redirect&bpo=37463]: ssl.match_hostname() no longer accepts IPv4 addresses with additional text after the address and only quad-dotted notation without trailing whitespaces. Some inet_aton() implementations ignore whitespace and all data after whitespace, e.g. ‘127.0.0.1 whatever’.

- bpo-37363 [https://bugs.python.org/issue?@action=redirect&bpo=37363]: Adds audit events for **`ensurepip`**, **`ftplib`**, **`glob`**, **`imaplib`**, **`nntplib`**, **`pdb`**, **`poplib`**, **`shutil`**, **`smtplib`**, **`sqlite3`**, **`subprocess`**, **`telnetlib`**, **`tempfile`** and **`webbrowser`**, as well as **`os.listdir()`**, **`os.scandir()`** and **`breakpoint()`**.
- bpo-37364 [https://bugs.python.org/issue?@action=redirect&bpo=37364]: **`io.open_code()`** is now used when reading `.pth` files.
- bpo-34631 [https://bugs.python.org/issue?@action=redirect&bpo=34631]: Updated OpenSSL to 1.1.1c in Windows installer
- bpo-34155 [https://bugs.python.org/issue?@action=redirect&bpo=34155]: Fix parsing of invalid email addresses with more than one `@` (e.g. a@b@c.com.) to not return the part before 2nd `@` as valid email address. Patch by maxking & jpic.

Core and Builtins

- bpo-38631 [https://bugs.python.org/issue?@action=redirect&bpo=38631]: Replace `Py_FatalError()` call with a regular **`RuntimeError`** exception in **`float.__getformat__()`**.

- bpo-38639 [https://bugs.python.org/issue?@action=redirect&bpo=38639]: Optimized **`math.floor()`**, **`math.ceil()`** and **`math.trunc()`** for floats.

- bpo-38640 [https://bugs.python.org/issue?@action=redirect&bpo=38640]: Fixed a bug in the compiler that was causing to raise in the presence of break statements and continue statements inside always false while loops. Patch by Pablo Galindo.

- bpo-38613 [https://bugs.python.org/issue?@action=redirect&bpo=38613]: Optimized some set operations (e.g. `|`, `^`, and `-`) of `dict_keys`. `d.keys() | other` was slower than `set(d) | other` but they are almost same performance for now.

- bpo-28029 [https://bugs.python.org/issue?@action=redirect&bpo=28029]: `"".replace("", s, n)` now returns `s` instead of an empty string for all non-zero `n`. There are similar changes for **`bytes`** and **`bytearray`** objects.
- bpo-38535 [https://bugs.python.org/issue?@action=redirect&bpo=38535]: Fixed line numbers and column offsets for AST nodes for calls without arguments in decorators.
- bpo-38525 [https://bugs.python.org/issue?@action=redirect&bpo=38525]: Fix a segmentation fault when using reverse iterators of empty `dict` objects. Patch by Donghee Na and Inada Naoki.
- bpo-38465 [https://bugs.python.org/issue?@action=redirect&bpo=38465]: **`bytearray`**, **`array`** and **`mmap`** objects allow now to export more than `2**31` buffers at a time.
- bpo-38469 [https://bugs.python.org/issue?@action=redirect&bpo=38469]: Fixed a bug where the scope of named expressions was not being resolved correctly in the presence of the *global* keyword. Patch by Pablo Galindo.
- bpo-38437 [https://bugs.python.org/issue?@action=redirect&bpo=38437]: Activate the `GC_DEBUG` macro for debug builds of the interpreter (when `Py_DEBUG` is set). Patch by Pablo Galindo.
- bpo-38379 [https://bugs.python.org/issue?@action=redirect&bpo=38379]: When the garbage collector makes a collection in which some objects resurrect (they are reachable from outside the isolated cycles after the

finalizers have been executed), do not block the collection of all objects that are still unreachable. Patch by Pablo Galindo and Tim Peters.

- bpo-38379 [https://bugs.python.org/issue?@action=redirect&bpo=38379]: When cyclic garbage collection (gc) runs finalizers that resurrect unreachable objects, the current gc run ends, without collecting any cyclic trash. However, the statistics reported by `collect()` and `get_stats()` claimed that all cyclic trash found was collected, and that the resurrected objects were collected. Changed the stats to report that none were collected.

- bpo-38392 [https://bugs.python.org/issue?@action=redirect&bpo=38392]: In debug mode, **`PyObject_GC_Track()`** now calls `tp_traverse()` of the object type to ensure that the object is valid: test that objects visited by `tp_traverse()` are valid.

- bpo-38210 [https://bugs.python.org/issue?@action=redirect&bpo=38210]: Remove unnecessary intersection and update set operation in dictview with empty set. (Contributed by Donghee Na in bpo-38210 [https://bugs.python.org/issue?@action=redirect&bpo=38210].)

- bpo-38402 [https://bugs.python.org/issue?@action=redirect&bpo=38402]: Check the error from the system's underlying `crypt` or `crypt_r`.

- bpo-37474 [https://bugs.python.org/issue?@action=redirect&bpo=37474]: On FreeBSD, Python no longer calls `fedisableexcept()` at startup to control the floating-point control mode. The call became useless since FreeBSD 6: it became the default mode.

- bpo-38006 [https://bugs.python.org/issue?@action=redirect&bpo=38006]: Fix a bug due to the interaction of weakrefs and the cyclic garbage collector. We must clear any weakrefs in garbage in order to prevent their callbacks from executing and causing a crash.

- bpo-38317 [https://bugs.python.org/issue?@action=redirect&bpo=38317]: Fix warnings options priority: `PyConfig.warnoptions` has the highest priority, as stated in the **PEP 587** [https://peps.python.org/pep-0587/].

- bpo-38310 [https://bugs.python.org/issue?@action=redirect&bpo=38310]: Predict `BUILD_MAP_UNPACK_WITH_CALL` -> `CALL_FUNCTION_EX` opcode pairs in the main interpreter loop. Patch by Brandt Bucher.

- bpo-36871 [https://bugs.python.org/issue?@action=redirect&bpo=36871]: Improve error handling for the assert_has_calls and assert_has_awaits methods of mocks. Fixed a bug where any errors encountered while binding the expected calls to the mock's spec were silently swallowed, leading to misleading error output.

- bpo-11410 [https://bugs.python.org/issue?@action=redirect&bpo=11410]: Better control over symbol visibility is provided through use of the visibility attributes available in gcc >= 4.0, provided in a uniform way across POSIX and Windows. The POSIX build files have been updated to compile with -fvisibility=hidden, minimising exported symbols.

- bpo-38219 [https://bugs.python.org/issue?@action=redirect&bpo=38219]: Optimized the `dict` constructor and the `update()` method for the case when the argument is a dict.

- bpo-38236 [https://bugs.python.org/issue?@action=redirect&bpo=38236]: Python now dumps path configuration if it fails to import the Python codecs of the filesystem and stdio encodings.
- bpo-38013 [https://bugs.python.org/issue?@action=redirect&bpo=38013]: Allow to call `async_generator_athrow().throw(...)` even for non-started async generator helper. It fixes annoying warning at the end of `asyncio.run()` call.
- bpo-38124 [https://bugs.python.org/issue?@action=redirect&bpo=38124]: Fix an off-by-one error in PyState_AddModule that could cause out-of-bounds memory access.
- bpo-38116 [https://bugs.python.org/issue?@action=redirect&bpo=38116]: The select module is now PEP-384 compliant and no longer has static state
- bpo-38113 [https://bugs.python.org/issue?@action=redirect&bpo=38113]: ast module updated to PEP-384 and all statics removed
- bpo-38076 [https://bugs.python.org/issue?@action=redirect&bpo=38076]: The struct module is now PEP-384 compatible
- bpo-38075 [https://bugs.python.org/issue?@action=redirect&bpo=38075]: The random module is now PEP-384 compatible
- bpo-38074 [https://bugs.python.org/issue?@action=redirect&bpo=38074]: zlib module made PEP-384 compatible

- bpo-38073 [https://bugs.python.org/issue?@action=redirect&bpo=38073]: Make pwd extension module PEP-384 compatible

- bpo-38072 [https://bugs.python.org/issue?@action=redirect&bpo=38072]: grp module made PEP-384 compatible

- bpo-38069 [https://bugs.python.org/issue?@action=redirect&bpo=38069]: Make _posixsubprocess PEP-384 compatible

- bpo-38071 [https://bugs.python.org/issue?@action=redirect&bpo=38071]: Make termios extension module PEP-384 compatible

- bpo-38005 [https://bugs.python.org/issue?@action=redirect&bpo=38005]: Fixed comparing and creating of InterpreterID and ChannelID.

- bpo-36946 [https://bugs.python.org/issue?@action=redirect&bpo=36946]: Fix possible signed integer overflow when handling slices. Patch by hongweipeng.

- bpo-37994 [https://bugs.python.org/issue?@action=redirect&bpo=37994]: Fixed silencing arbitrary errors if an attribute lookup fails in several sites. Only AttributeError should be silenced.

- bpo-8425 [https://bugs.python.org/issue?@action=redirect&bpo=8425]: Optimize set difference_update for the case when the other set is much larger than the base set. (Suggested by Evgeny Kapun with code contributed by Michele Orrù).

- bpo-37966 [https://bugs.python.org/issue?@action=redirect&bpo=37966]: The implementation of `is_normalized()` has been greatly sped up on strings that aren’t normalized, by implementing the full normalization-quick-check algorithm from the Unicode standard.

- bpo-37947 [https://bugs.python.org/issue?@action=redirect&bpo=37947]: Adjust correctly the recursion level in the symtable generation for named expressions. Patch by Pablo Galindo.
- bpo-37812 [https://bugs.python.org/issue?@action=redirect&bpo=37812]: The `CHECK_SMALL_INT` macro used inside `Object/longobject.c` has been replaced with an explicit `return` at each call site.
- bpo-37751 [https://bugs.python.org/issue?@action=redirect&bpo=37751]: Fix `codecs.lookup()` to normalize the encoding name the same way than `encodings.normalize_encoding()`, except that `codecs.lookup()` also converts the name to lower case.
- bpo-37830 [https://bugs.python.org/issue?@action=redirect&bpo=37830]: Fixed compilation of `break` and `continue` in the `finally` block when the corresponding `try` block contains `return` with a non-constant value.
- bpo-20490 [https://bugs.python.org/issue?@action=redirect&bpo=20490]: Improve import error message for partially initialized module on circular `from` imports - by Anthony Sottile.
- bpo-37840 [https://bugs.python.org/issue?@action=redirect&bpo=37840]: Fix handling of negative indices in `sq_item` of `bytearray`. Patch by Sergey Fedoseev.
- bpo-37802 [https://bugs.python.org/issue?@action=redirect&bpo=37802]: Slightly improve performance

of `PyLong_FromUnsignedLong()`, `PyLong_FromUnsignedLongLong()` and `PyLong_FromSize_t()`. Patch by Sergey Fedoseev.

- bpo-37409 [https://bugs.python.org/issue?@action=redirect&bpo=37409]: Ensure explicit relative imports from interactive sessions and scripts (having no parent package) always raise ImportError, rather than treating the current module as the package. Patch by Ben Lewis.

- bpo-32912 [https://bugs.python.org/issue?@action=redirect&bpo=32912]: Reverted bpo-32912 [https://bugs.python.org/issue?@action=redirect&bpo=32912]: emitting `SyntaxWarning` instead of `DeprecationWarning` for invalid escape sequences in string and bytes literals.

- bpo-37757 [https://bugs.python.org/issue?@action=redirect&bpo=37757]: **PEP 572** [https://peps.python.org/pep-0572/]: As described in the PEP, assignment expressions now raise `SyntaxError` when their interaction with comprehension scoping results in an ambiguous target scope.

  The `TargetScopeError` subclass originally proposed by the PEP has been removed in favour of just raising regular syntax errors for the disallowed cases.

- bpo-36279 [https://bugs.python.org/issue?@action=redirect&bpo=36279]: Fix potential use of uninitialized memory in `os.wait3()`.

- bpo-36311 [https://bugs.python.org/issue?@action=redirect&bpo=36311]: Decoding bytes objects larger than 2GiB is faster and no longer fails when a multibyte characters spans a chunk boundary.

- bpo-34880 [https://bugs.python.org/issue?@action=redirect&bpo=34880]: The `assert` statement now works properly if the `AssertionError` exception is being shadowed. Patch by Zackery Spytz.

- bpo-37340 [https://bugs.python.org/issue?@action=redirect&bpo=37340]: Removed object cache (`free_list`) for bound method objects. Temporary bound method objects are less used than before thanks to the `LOAD_METHOD` opcode and the `_PyObject_VectorcallMethod` C API.

- bpo-37648 [https://bugs.python.org/issue?@action=redirect&bpo=37648]: Fixed minor inconsistency in **`list.__contains__()`**, **`tuple.__contains__()`** and a few other places. The collection's item is now always at the left and the needle is on the right of `==`.

- bpo-37444 [https://bugs.python.org/issue?@action=redirect&bpo=37444]: Update differing exception between **`builtins.__import__()`** and `importlib.__import__()`.

- bpo-37619 [https://bugs.python.org/issue?@action=redirect&bpo=37619]: When adding a wrapper descriptor from one class to a different class (for example, setting `__add__ = str.__add__` on an `int` subclass), an exception is correctly raised when the operator is called.

- bpo-37593 [https://bugs.python.org/issue?@action=redirect&bpo=37593]: Swap the positions of the *posonlyargs* and *args* parameters in the constructor of **`ast.parameters`** nodes.

- bpo-37543 [https://bugs.python.org/issue?@action=redirect&bpo=37543]: Optimized pymalloc for non PGO build.

- bpo-37537 [https://bugs.python.org/issue?@action=redirect&bpo=37537]: Compute allocated pymalloc blocks inside _Py_GetAllocatedBlocks(). This slows down _Py_GetAllocatedBlocks() but gives a small speedup to _PyObject_Malloc() and _PyObject_Free().

- bpo-37467 [https://bugs.python.org/issue?@action=redirect&bpo=37467]: Fix **`sys.excepthook()`** and **`PyErr_Display()`** if a filename is a bytes string. For example, for a SyntaxError exception where the filename attribute is a bytes string.

- bpo-37433 [https://bugs.python.org/issue?@action=redirect&bpo=37433]: Fix `SyntaxError` indicator printing too many spaces for multi-line strings - by Anthony Sottile.

- bpo-37417 [https://bugs.python.org/issue?@action=redirect&bpo=37417]: **`bytearray.extend()`** now correctly handles errors that arise during iteration. Patch by Brandt Bucher.

- bpo-37414 [https://bugs.python.org/issue?@action=redirect&bpo=37414]: The undocumented `sys.callstats()` function has been removed. Since Python 3.7, it was deprecated and always returned `None`. It required a special build option `CALL_PROFILE` which was already removed in Python 3.7.

- bpo-37392 [https://bugs.python.org/issue?@action=redirect&bpo=37392]: Remove `sys.getcheckinterval()` and `sys.setcheckinterval()` functions. They were deprecated since Python 3.2. Use **`sys.getswitchinterval()`** and **`sys.setswitchinterval()`** instead. Remove also `check_interval` field of the `PyInterpreterState` structure.

- bpo-37388 [https://bugs.python.org/issue?@action=redirect&bpo=37388]: In development mode and in debug build, *encoding* and *errors* arguments are now checked on string encoding and decoding operations. Examples: **`open()`**, **`str.encode()`** and **`bytes.decode()`**.

  By default, for best performances, the *errors* argument is only checked at the first encoding/decoding error, and the *encoding* argument is sometimes ignored for empty strings.

- bpo-37348 [https://bugs.python.org/issue?@action=redirect&bpo=37348]: Optimized decoding short ASCII string with UTF-8 and ascii codecs. `b"foo".decode()` is about 15% faster. Patch by Inada Naoki.

- bpo-24214 [https://bugs.python.org/issue?@action=redirect&bpo=24214]: Improved support of the surrogatepass error handler in the UTF-8 and UTF-16 incremental decoders.

- bpo-37330 [https://bugs.python.org/issue?@action=redirect&bpo=37330]: **`open()`**, **`io.open()`**, **`codecs.open()`** and **`fileinput.FileInput`** no longer accept `'U'` (“universal newline”) in the file mode. This flag was deprecated since Python 3.3.

- bpo-35224 [https://bugs.python.org/issue?@action=redirect&bpo=35224]: Reverse evaluation order of key: value in dict comprehensions as proposed in PEP 572. I.e. in `{k: v for ...}`, `k` will be evaluated before `v`.

- bpo-37316 [https://bugs.python.org/issue?@action=redirect&bpo=37316]: Fix the **`PySys_Audit()`** call in **`mmap.mmap`**.

- bpo-37300 [https://bugs.python.org/issue?@action=redirect&bpo=37300]: Remove an unnecessary Py_XINCREF in classobject.c.

- bpo-37269 [https://bugs.python.org/issue?@action=redirect&bpo=37269]: Fix a bug in the peephole optimizer that was not treating correctly constant conditions with binary operators. Patch by Pablo Galindo.

- bpo-20443 [https://bugs.python.org/issue?@action=redirect&bpo=20443]: Python now gets the absolute path of the script filename specified on the command line (ex: “python3 script.py”): the __file__ attribute of the __main__ module and sys.path[0] become an absolute path, rather than a relative path.

- bpo-37257 [https://bugs.python.org/issue?@action=redirect&bpo=37257]: Python’s small object allocator (`obmalloc.c`) now allows (no more than) one empty arena to remain available for immediate reuse, without returning it to the OS. This prevents thrashing in simple loops where an arena could be created and destroyed anew on each iteration.

- bpo-37231 [https://bugs.python.org/issue?@action=redirect&bpo=37231]: The dispatching of type slots to special methods (for example calling `__mul__` when doing `x * y`) has been made faster.

- bpo-36974 [https://bugs.python.org/issue?@action=redirect&bpo=36974]: Implemented separate vectorcall functions for every calling convention of builtin functions and methods. This improves performance for calls.

- bpo-37213 [https://bugs.python.org/issue?@action=redirect&bpo=37213]: Handle correctly negative line offsets in the peephole optimizer. Patch by Pablo Galindo.
- bpo-37219 [https://bugs.python.org/issue?@action=redirect&bpo=37219]: Remove erroneous optimization for empty set differences.
- bpo-15913 [https://bugs.python.org/issue?@action=redirect&bpo=15913]: Implement **`PyBuffer_SizeFromFormat()`** function (previously documented but not implemented): call **`struct.calcsize()`**. Patch by Joannah Nanjekye.
- bpo-36922 [https://bugs.python.org/issue?@action=redirect&bpo=36922]: Slot functions optimize any callable with `Py_TPFLAGS_METHOD_DESCRIPTOR` instead of only instances of `function`.
- bpo-36974 [https://bugs.python.org/issue?@action=redirect&bpo=36974]: The slot `tp_vectorcall_offset` is inherited unconditionally to support `super().__call__()` when the base class uses vectorcall.
- bpo-37160 [https://bugs.python.org/issue?@action=redirect&bpo=37160]: **`threading.get_native_id()`** now also supports NetBSD.
- bpo-37077 [https://bugs.python.org/issue?@action=redirect&bpo=37077]: Add **`threading.get_native_id()`** support for AIX. Patch by M. Felt
- bpo-36781 [https://bugs.python.org/issue?@action=redirect&bpo=36781]: **`sum()`** has been optimized for boolean values.

- bpo-34556 [https://bugs.python.org/issue?@action=redirect&bpo=34556]: Add `--upgrade-deps` to venv module. Patch by Cooper Ry Lees

- bpo-20523 [https://bugs.python.org/issue?@action=redirect&bpo=20523]: `pdb.Pdb` supports ~/.pdbrc in Windows 7. Patch by Tim Hopper and Dan Lidral-Porter.

- bpo-35551 [https://bugs.python.org/issue?@action=redirect&bpo=35551]: Updated encodings: - Removed the “tis260” encoding, which was an alias for the nonexistent “tactis” codec. - Added “mac_centeuro” as an alias for the mac_latin2 encoding.

- bpo-19072 [https://bugs.python.org/issue?@action=redirect&bpo=19072]: The **`classmethod`** decorator can now wrap other descriptors such as property objects. Adapted from a patch written by Graham Dumpleton.

- bpo-27575 [https://bugs.python.org/issue?@action=redirect&bpo=27575]: Improve speed of dictview intersection by directly using set intersection logic. Patch by David Su.

- bpo-30773 [https://bugs.python.org/issue?@action=redirect&bpo=30773]: Prohibit parallel running of aclose() / asend() / athrow(). Fix ag_running to reflect the actual running status of the AG.

Library

- bpo-36589 [https://bugs.python.org/issue?@action=redirect&bpo=36589]: The **`curses.update_lines_cols()`** function now returns `None` instead of `1` on success.

- bpo-38807 [https://bugs.python.org/issue?@action=redirect&bpo=38807]: Update **`TypeError`** messages for `os.path.join()` to include `os.PathLike` objects as acceptable input types.

- bpo-38724 [https://bugs.python.org/issue?@action=redirect&bpo=38724]: Add a repr for `subprocess.Popen` objects. Patch by Andrey Doroschenko.

- bpo-38786 [https://bugs.python.org/issue?@action=redirect&bpo=38786]: pydoc now recognizes and parses HTTPS URLs. Patch by python273.

- bpo-38785 [https://bugs.python.org/issue?@action=redirect&bpo=38785]: Prevent asyncio from crashing if parent `__init__` is not called from a constructor of object derived from `asyncio.Future`.

- bpo-38723 [https://bugs.python.org/issue?@action=redirect&bpo=38723]: `pdb` now uses `io.open_code()` to trigger auditing events.

- bpo-27805 [https://bugs.python.org/issue?@action=redirect&bpo=27805]: Allow opening pipes and other non-seekable files in append mode with `open()`.

- bpo-38438 [https://bugs.python.org/issue?@action=redirect&bpo=38438]: Simplify the `argparse` usage message for `nargs="*"`.

- bpo-38761 [https://bugs.python.org/issue?@action=redirect&bpo=38761]: WeakSet is now registered as a collections.abc.MutableSet.

- bpo-38716 [https://bugs.python.org/issue?@action=redirect&bpo=38716]: logging: change RotatingHandler namer and rotator to class-level attributes. This stops __init__ from setting them to None in the case where a subclass defines them with eponymous methods.

- bpo-38713 [https://bugs.python.org/issue?@action=redirect&bpo=38713]: Add `os.P_PIDFD` constant, which may be passed to `os.waitid()` to wait on a Linux process file descriptor.
- bpo-38692 [https://bugs.python.org/issue?@action=redirect&bpo=38692]: Add **`asyncio.PidfdChildWatcher`**, a Linux-specific child watcher implementation that polls process file descriptors.
- bpo-38692 [https://bugs.python.org/issue?@action=redirect&bpo=38692]: Expose the Linux `pidfd_open` syscall as `os.pidfd_open()`.
- bpo-38602 [https://bugs.python.org/issue?@action=redirect&bpo=38602]: Added constants **`F_OFD_GETLK`**, **`F_OFD_SETLK`** and **`F_OFD_SETLKW`** to the `fcntl` module. Patch by Donghee Na.
- bpo-38334 [https://bugs.python.org/issue?@action=redirect&bpo=38334]: Fixed seeking backward on an encrypted **`zipfile.ZipExtFile`**.
- bpo-38312 [https://bugs.python.org/issue?@action=redirect&bpo=38312]: Add `curses.get_escdelay()`, `curses.set_escdelay()`, `curses.get_tabsize()`, and `curses.set_tabsize()` functions - by Anthony Sottile.
- bpo-38586 [https://bugs.python.org/issue?@action=redirect&bpo=38586]: Now `fileConfig()` correctly sets the .name of handlers loaded.
- bpo-38565 [https://bugs.python.org/issue?@action=redirect&bpo=38565]: Add new cache_parameters()

method for functools.lru_cache() to better support pickling.

- bpo-34679 [https://bugs.python.org/issue?@action=redirect&bpo=34679]: asynci.ProactorEventLoop.close() now only calls signal.set_wakeup_fd() in the main thread.
- bpo-31202 [https://bugs.python.org/issue?@action=redirect&bpo=31202]: The case the result of **`pathlib.WindowsPath.glob()`** matches now the case of the pattern for literal parts.
- bpo-36321 [https://bugs.python.org/issue?@action=redirect&bpo=36321]: Remove misspelled attribute. The 3.8 changelog noted that this would be removed in 3.9.
- bpo-38521 [https://bugs.python.org/issue?@action=redirect&bpo=38521]: Fixed erroneous equality comparison in statistics.NormalDist().
- bpo-38493 [https://bugs.python.org/issue?@action=redirect&bpo=38493]: Added **`CLD_KILLED`** and **`CLD_STOPPED`** for **`si_code`**. Patch by Donghee Na.
- bpo-38478 [https://bugs.python.org/issue?@action=redirect&bpo=38478]: Fixed a bug in **`inspect.signature.bind()`** that was causing it to fail when handling a keyword argument with same name as positional-only parameter. Patch by Pablo Galindo.
- bpo-33604 [https://bugs.python.org/issue?@action=redirect&bpo=33604]: Fixed `hmac.new` and `hmac.HMAC` to raise TypeError instead of ValueError when the digestmod parameter, now required in 3.8, is omitted.

Also clarified the hmac module documentation and docstrings.

- bpo-38378 [https://bugs.python.org/issue?@action=redirect&bpo=38378]: Parameters *out* and *in* of **`os.sendfile()`** was renamed to *out_fd* and *in_fd*.

- bpo-38417 [https://bugs.python.org/issue?@action=redirect&bpo=38417]: Added support for setting the umask in the child process to the subprocess module on POSIX systems.

- bpo-38449 [https://bugs.python.org/issue?@action=redirect&bpo=38449]: Revert PR 15522, which introduces a regression in **`mimetypes.guess_type()`** due to improper handling of filenames as urls.

- bpo-38431 [https://bugs.python.org/issue?@action=redirect&bpo=38431]: Fix `__repr__` method for **`dataclasses.InitVar`** to support typing objects, patch by Samuel Colvin.

- bpo-38109 [https://bugs.python.org/issue?@action=redirect&bpo=38109]: Add missing **`stat.S_IFDOOR`**, **`stat.S_IFPORT`**, **`stat.S_IFWHT`**, **`stat.S_ISDOOR()`**, **`stat.S_ISPORT()`**, and **`stat.S_ISWHT()`** values to the Python implementation of **`stat`**.

- bpo-38422 [https://bugs.python.org/issue?@action=redirect&bpo=38422]: Clarify docstrings of pathlib suffix(es)

- bpo-38405 [https://bugs.python.org/issue?@action=redirect&bpo=38405]: Nested subclasses of **`typing.NamedTuple`** are now pickleable.

- bpo-38332 [https://bugs.python.org/issue?@action=redirect&bpo=38332]: Prevent `KeyError` thrown by **`_encoded_words.decode()`** when given an encoded-word with invalid content-type encoding from propagating all the way to **`email.message.get()`**.

- bpo-38371 [https://bugs.python.org/issue?@action=redirect&bpo=38371]: Deprecated the `split()` method in **`_tkinter.TkappType`** in favour of the `splitlist()` method which has more consistent and predictable behavior.

- bpo-38341 [https://bugs.python.org/issue?@action=redirect&bpo=38341]: Add `smtplib.SMTPNotSupportedError` to the `smtplib` exported names.

- bpo-38319 [https://bugs.python.org/issue?@action=redirect&bpo=38319]: sendfile() used in socket and shutil modules was raising OverflowError for files >= 2GiB on 32-bit architectures. (patch by Giampaolo Rodola)

- bpo-38242 [https://bugs.python.org/issue?@action=redirect&bpo=38242]: Revert the new asyncio Streams API

- bpo-13153 [https://bugs.python.org/issue?@action=redirect&bpo=13153]: OS native encoding is now used for converting between Python strings and Tcl objects. This allows to display, copy and paste to clipboard emoji and other non-BMP characters. Converting strings from Tcl to Python and back now never fails (except MemoryError).

- bpo-38019 [https://bugs.python.org/issue?@action=redirect&bpo=38019]: Correctly handle pause/resume reading of closed asyncio unix pipe.

- bpo-38163 [https://bugs.python.org/issue?@action=redirect&bpo=38163]: Child mocks will now detect their type as either synchronous or asynchronous, asynchronous child mocks will be AsyncMocks and synchronous child mocks will be either MagicMock or Mock (depending on their parent type).

- bpo-38161 [https://bugs.python.org/issue?@action=redirect&bpo=38161]: Removes _AwaitEvent from AsyncMock.

- bpo-38216 [https://bugs.python.org/issue?@action=redirect&bpo=38216]: Allow the rare code that wants to send invalid http requests from the `http.client` library a way to do so. The fixes for bpo-30458 [https://bugs.python.org/issue?@action=redirect&bpo=30458] led to breakage for some projects that were relying on this ability to test their own behavior in the face of bad requests.

- bpo-28286 [https://bugs.python.org/issue?@action=redirect&bpo=28286]: Deprecate opening **`GzipFile`** for writing implicitly. Always specify the *mode* argument for writing.

- bpo-38108 [https://bugs.python.org/issue?@action=redirect&bpo=38108]: Any synchronous magic methods on an AsyncMock now return a MagicMock. Any asynchronous magic methods on a MagicMock now return an AsyncMock.

- bpo-38265 [https://bugs.python.org/issue?@action=redirect&bpo=38265]: Update the *length* parameter of **`os.pread()`** to accept **`Py_ssize_t`** instead of int.

- bpo-38112 [https://bugs.python.org/issue?@action=redirect&bpo=38112]: **`compileall`** has a higher default recursion limit and new command-line arguments for path manipulation, symlinks handling, and multiple optimization levels.

- bpo-38248 [https://bugs.python.org/issue?@action=redirect&bpo=38248]: asyncio: Fix inconsistent immediate Task cancellation
- bpo-38237 [https://bugs.python.org/issue?@action=redirect&bpo=38237]: The arguments for the builtin pow function are more descriptive. They can now also be passed in as keywords.
- bpo-34002 [https://bugs.python.org/issue?@action=redirect&bpo=34002]: Improve efficiency in parts of email package by changing while-pop to a for loop, using isdisjoint instead of set intersections.
- bpo-38191 [https://bugs.python.org/issue?@action=redirect&bpo=38191]: Constructors of **`NamedTuple`** and **`TypedDict`** types now accept arbitrary keyword argument names, including "cls", "self", "typename", "_typename", "fields" and "_fields".
- bpo-38155 [https://bugs.python.org/issue?@action=redirect&bpo=38155]: Add `__all__` to **`datetime`**. Patch by Tahia Khan.
- bpo-38185 [https://bugs.python.org/issue?@action=redirect&bpo=38185]: Fixed case-insensitive string comparison in **`sqlite3.Row`** indexing.
- bpo-38136 [https://bugs.python.org/issue?@action=redirect&bpo=38136]: Changes AsyncMock call count and await count to be two different counters. Now await count only counts when a coroutine has been awaited,

not when it has been called, and vice-versa. Update the documentation around this.

- bpo-37828 [https://bugs.python.org/issue?@action=redirect&bpo=37828]: Fix default mock name in `unittest.mock.Mock.assert_called()` exceptions. Patch by Abraham Toriz Cruz.
- bpo-38175 [https://bugs.python.org/issue?@action=redirect&bpo=38175]: Fix a memory leak in comparison of `sqlite3.Row` objects.
- bpo-33936 [https://bugs.python.org/issue?@action=redirect&bpo=33936]: _hashlib no longer calls obsolete OpenSSL initialization function with OpenSSL 1.1.0+.
- bpo-34706 [https://bugs.python.org/issue?@action=redirect&bpo=34706]: Preserve subclassing in inspect.Signature.from_callable.
- bpo-38153 [https://bugs.python.org/issue?@action=redirect&bpo=38153]: Names of hashing algorithms from OpenSSL are now normalized to follow Python’s naming conventions. For example OpenSSL uses sha3-512 instead of sha3_512 or blake2b512 instead of blake2b.
- bpo-38115 [https://bugs.python.org/issue?@action=redirect&bpo=38115]: Fix a bug in dis.findlinestarts() where it would return invalid bytecode offsets. Document that a code object’s co_lnotab can contain invalid bytecode offsets.
- bpo-38148 [https://bugs.python.org/issue?@action=redirect&bpo=38148]: Add slots to `asyncio` transport classes, which can reduce memory usage.

- bpo-38142 [https://bugs.python.org/issue?@action=redirect&bpo=38142]: The _hashlib OpenSSL wrapper extension module is now PEP-384 compliant.

- bpo-9216 [https://bugs.python.org/issue?@action=redirect&bpo=9216]: hashlib constructors now support usedforsecurity flag to signal that a hashing algorithm is not used in a security context.

- bpo-36991 [https://bugs.python.org/issue?@action=redirect&bpo=36991]: Fixes a potential incorrect AttributeError exception escaping ZipFile.extract() in some unsupported input error situations.

- bpo-38134 [https://bugs.python.org/issue?@action=redirect&bpo=38134]: Remove obsolete copy of PBKDF2_HMAC_fast. All supported OpenSSL versions contain a fast implementation.

- bpo-38132 [https://bugs.python.org/issue?@action=redirect&bpo=38132]: The OpenSSL hashlib wrapper uses a simpler implementation. Several Macros and pointless caches are gone. The hash name now comes from OpenSSL’s EVP. The algorithm name stays the same, except it is now always lower case.

- bpo-38008 [https://bugs.python.org/issue?@action=redirect&bpo=38008]: Fix parent class check in protocols to correctly identify the module that provides a builtin protocol, instead of assuming they all come from the `collections.abc` module

- bpo-34037 [https://bugs.python.org/issue?@action=redirect&bpo=34037]: For `asyncio`, add a new coroutine `loop.shutdown_default_executor()`. The new coroutine provides an API to schedule an executor shutdown that waits on the threadpool to finish closing.

Also, [`asyncio.run()`] has been updated to utilize the new coroutine. Patch by Kyle Stanley.

- bpo-37405 [https://bugs.python.org/issue?@action=redirect&bpo=37405]: Fixed regression bug for socket.getsockname() for non-CAN_ISOTP AF_CAN address family sockets by returning a 1-tuple instead of string.

- bpo-38121 [https://bugs.python.org/issue?@action=redirect&bpo=38121]: Update parameter names on functions in importlib.metadata matching the changes in the 0.22 release of importlib_metadata.

- bpo-38110 [https://bugs.python.org/issue?@action=redirect&bpo=38110]: The os.closewalk() implementation now uses the libc fdwalk() API on platforms where it is available.

- bpo-38093 [https://bugs.python.org/issue?@action=redirect&bpo=38093]: Fixes AsyncMock so it doesn't crash when used with AsyncContextManagers or AsyncIterators.

- bpo-37488 [https://bugs.python.org/issue?@action=redirect&bpo=37488]: Add warning to **`datetime.utctimetuple()`**, **`datetime.utcnow()`** and **`datetime.utcfromtimestamp()`** .

- bpo-35640 [https://bugs.python.org/issue?@action=redirect&bpo=35640]: Allow passing a path-like object as `directory` argument to the **`http.server.SimpleHTTPRequestHandler`** class. Patch by Géry Ogam.

- bpo-38086 [https://bugs.python.org/issue?@action=redirect&bpo=38086]: Update importlib.metadata with changes from importlib_metadata 0.21 [https://gitlab.com/python-devs/importlib_metadata/blob/0.21/importlib_metadata/docs/changelog.rst] .

- bpo-37251 [https://bugs.python.org/issue?@action=redirect&bpo=37251]: Remove `__code__` check in AsyncMock that incorrectly evaluated function specs as async objects but failed to evaluate classes with `__await__` but no `__code__` attribute defined as async objects.

- bpo-38037 [https://bugs.python.org/issue?@action=redirect&bpo=38037]: Fix reference counters in the **`signal`** module.

- bpo-38066 [https://bugs.python.org/issue?@action=redirect&bpo=38066]: Hide internal asyncio.Stream methods: feed_eof(), feed_data(), set_exception() and set_transport().

- bpo-38059 [https://bugs.python.org/issue?@action=redirect&bpo=38059]: inspect.py now uses sys.exit() instead of exit()

- bpo-38049 [https://bugs.python.org/issue?@action=redirect&bpo=38049]: Added command-line interface for the **`ast`** module.

- bpo-37953 [https://bugs.python.org/issue?@action=redirect&bpo=37953]: In **`typing`**, improved the `__hash__` and `__eq__` methods for **`ForwardReferences`**.

- bpo-38026 [https://bugs.python.org/issue?@action=redirect&bpo=38026]: Fixed **`inspect.getattr_static()`** used `isinstance` while it should avoid dynamic lookup.

- bpo-35923 [https://bugs.python.org/issue?@action=redirect&bpo=35923]: Update **`importlib.machinery.BuiltinImporter`** to use `loader._ORIGIN` instead of a hardcoded value. Patch by Donghee Na.

- bpo-38010 [https://bugs.python.org/issue?@action=redirect&bpo=38010]: In `importlib.metadata` sync with `importlib_metadata` 0.20, clarifying behavior of `files()` and fixing issue where only one requirement was returned for `requires()` on `dist-info` packages.

- bpo-38006 [https://bugs.python.org/issue?@action=redirect&bpo=38006]: weakref.WeakValueDictionary defines a local remove() function used as callback for weak references. This function was created with a closure. Modify the implementation to avoid the closure.

- bpo-37995 [https://bugs.python.org/issue?@action=redirect&bpo=37995]: Added the *indent* option to **`ast.dump()`** which allows it to produce a multiline indented output.

- bpo-34410 [https://bugs.python.org/issue?@action=redirect&bpo=34410]: Fixed a crash in the **`tee()`** iterator when re-enter it. RuntimeError is now raised in this case.

- bpo-37140 [https://bugs.python.org/issue?@action=redirect&bpo=37140]: Fix a ctypes regression of Python 3.8. When a ctypes.Structure is passed by copy to a function, ctypes internals created a temporary object which had the side effect of calling the structure finalizer (__del__) twice. The Python semantics requires a finalizer to be called exactly once. Fix ctypes internals to no longer call the finalizer twice.

- bpo-37587 [https://bugs.python.org/issue?@action=redirect&bpo=37587]: `_json.scanstring` is now up to 3x faster when there are many backslash escaped characters in the JSON string.

- bpo-37834 [https://bugs.python.org/issue?@action=redirect&bpo=37834]: Prevent shutil.rmtree exception when built on non-Windows system without fd system call support, like older versions of macOS.

- bpo-10978 [https://bugs.python.org/issue?@action=redirect&bpo=10978]: Semaphores and BoundedSemaphores can now release more than one waiting thread at a time.

- bpo-37972 [https://bugs.python.org/issue?@action=redirect&bpo=37972]: Subscripts to the `unittest.mock.call` objects now receive the same chaining mechanism as any other custom attributes, so that the following usage no longer raises a `TypeError`:

  ```
  call().foo().__getitem__('bar')
  ```

  Patch by blhsing

- bpo-37965 [https://bugs.python.org/issue?@action=redirect&bpo=37965]: Fix C compiler warning caused by distutils.ccompiler.CCompiler.has_function.

- bpo-37964 [https://bugs.python.org/issue?@action=redirect&bpo=37964]: Add `F_GETPATH` command to **`fcntl`**.

- bpo-37960 [https://bugs.python.org/issue?@action=redirect&bpo=37960]: `repr()` of buffered and text streams now silences only expected exceptions when get the value of “name” and “mode” attributes.

- bpo-37961 [https://bugs.python.org/issue?@action=redirect&bpo=37961]: Add a `total_nframe` field to the traces collected by the tracemalloc module. This field indicates the original number of frames before it was truncated.

- bpo-37951 [https://bugs.python.org/issue?@action=redirect&bpo=37951]: Most features of the subprocess module now work again in subinterpreters. Only *preexec_fn* is restricted in subinterpreters.

- bpo-36205 [https://bugs.python.org/issue?@action=redirect&bpo=36205]: Fix the rusage implementation of time.process_time() to correctly report the sum of the system and user CPU time.

- bpo-37950 [https://bugs.python.org/issue?@action=redirect&bpo=37950]: Fix **`ast.dump()`** when call with incompletely initialized node.

- bpo-34679 [https://bugs.python.org/issue?@action=redirect&bpo=34679]: Restores instantiation of Windows IOCP event loops from the non-main thread.

- bpo-36917 [https://bugs.python.org/issue?@action=redirect&bpo=36917]: Add default implementation of the **`ast.NodeVisitor.visit_Constant()`** method which emits a deprecation warning and calls corresponding methods `visit_Num()`, `visit_Str()`, etc.

- bpo-37798 [https://bugs.python.org/issue?@action=redirect&bpo=37798]: Update test_statistics.py to verify that the statistics module works well for both C and Python implementations. Patch by Donghee Na

- bpo-26589 [https://bugs.python.org/issue?@action=redirect&bpo=26589]: Added a new status code to the http module: 451 UNAVAILABLE_FOR_LEGAL_REASONS

- bpo-37915 [https://bugs.python.org/issue?@action=redirect&bpo=37915]: Fix a segmentation fault that appeared when comparing instances of `datetime.timezone` and `datetime.tzinfo` objects. Patch by Pablo Galindo.
- bpo-32554 [https://bugs.python.org/issue?@action=redirect&bpo=32554]: Deprecate having random.seed() call hash on arbitrary types.
- bpo-9938 [https://bugs.python.org/issue?@action=redirect&bpo=9938]: Add optional keyword argument `exit_on_error` for **`ArgumentParser`**.
- bpo-37851 [https://bugs.python.org/issue?@action=redirect&bpo=37851]: The **`faulthandler`** module no longer allocates its alternative stack at Python startup. Now the stack is only allocated at the first faulthandler usage.
- bpo-32793 [https://bugs.python.org/issue?@action=redirect&bpo=32793]: Fix a duplicated debug message when **`smtplib.SMTP.connect()`** is called.
- bpo-37885 [https://bugs.python.org/issue?@action=redirect&bpo=37885]: venv: Don’t generate unset variable warning on deactivate.
- bpo-37868 [https://bugs.python.org/issue?@action=redirect&bpo=37868]: Fix dataclasses.is_dataclass when given an instance that never raises AttributeError in __getattr__. That is, an object that returns something for __dataclass_fields__ even if it’s not a dataclass.
- bpo-37811 [https://bugs.python.org/issue?@action=redirect&bpo=37811]: Fix `socket` module’s

`socket.connect(address)` function being unable to establish connection in case of interrupted system call. The problem was observed on all OSes which `poll(2)` system call can take only non-negative integers and -1 as a timeout value.

- bpo-37863 [https://bugs.python.org/issue?@action=redirect&bpo=37863]: Optimizations for Fraction.__hash__ suggested by Tim Peters.
- bpo-21131 [https://bugs.python.org/issue?@action=redirect&bpo=21131]: Fix `faulthandler.register(chain=True)` stack. faulthandler now allocates a dedicated stack of `SIGSTKSZ*2` bytes, instead of just `SIGSTKSZ` bytes. Calling the previous signal handler in faulthandler signal handler uses more than `SIGSTKSZ` bytes of stack memory on some platforms.
- bpo-37798 [https://bugs.python.org/issue?@action=redirect&bpo=37798]: Add C fastpath for statistics.NormalDist.inv_cdf() Patch by Donghee Na
- bpo-37804 [https://bugs.python.org/issue?@action=redirect&bpo=37804]: Remove the deprecated method `threading.Thread.isAlive()`. Patch by Donghee Na.
- bpo-37819 [https://bugs.python.org/issue?@action=redirect&bpo=37819]: Add Fraction.as_integer_ratio() to match the corresponding methods in bool, int, float, and decimal.
- bpo-14465 [https://bugs.python.org/issue?@action=redirect&bpo=14465]: Add an xml.etree.ElementTree.indent() function for pretty-printing XML trees. Contributed by Stefan Behnel.

- bpo-37810 [https://bugs.python.org/issue?@action=redirect&bpo=37810]: Fix `difflib` `?` hint in diff output when dealing with tabs. Patch by Anthony Sottile.

- bpo-37772 [https://bugs.python.org/issue?@action=redirect&bpo=37772]: In `zipfile.Path`, when adding implicit dirs, ensure that ancestral directories are added and that duplicates are excluded.

- bpo-18578 [https://bugs.python.org/issue?@action=redirect&bpo=18578]: Renamed and documented `test.bytecode_helper` as `test.support.bytecode_helper`. Patch by Joannah Nanjekye.

- bpo-37785 [https://bugs.python.org/issue?@action=redirect&bpo=37785]: Fix xgettext warnings in `argparse`.

- bpo-34488 [https://bugs.python.org/issue?@action=redirect&bpo=34488]: **`writelines()`** method of `io.BytesIO` is now slightly faster when many small lines are passed. Patch by Sergey Fedoseev.

- bpo-37449 [https://bugs.python.org/issue?@action=redirect&bpo=37449]: `ensurepip` now uses `importlib.resources.read_binary()` to read data instead of `pkgutil.get_data()`. Patch by Joannah Nanjekye.

- bpo-28292 [https://bugs.python.org/issue?@action=redirect&bpo=28292]: Mark calendar.py helper functions as being private. The follows PEP 8 guidance to maintain the style conventions in the module and it addresses a known case of user confusion.

- bpo-18049 [https://bugs.python.org/issue?@action=redirect&bpo=18049]: Add definition of THREAD_STACK_SIZE for AIX in Python/thread_pthread.h

The default thread stacksize caused crashes with the default recursion limit Patch by M Felt

- bpo-37742 [https://bugs.python.org/issue?@action=redirect&bpo=37742]: The logging.getLogger() API now returns the root logger when passed the name 'root', whereas previously it returned a non-root logger named 'root'. This could affect cases where user code explicitly wants a non-root logger named 'root', or instantiates a logger using logging.getLogger(__name__) in some top-level module called 'root.py'.

- bpo-37738 [https://bugs.python.org/issue?@action=redirect&bpo=37738]: Fix the implementation of curses `addch(str, color_pair)`: pass the color pair to `setcchar()`, instead of always passing 0 as the color pair.

- bpo-37723 [https://bugs.python.org/issue?@action=redirect&bpo=37723]: Fix performance regression on regular expression parsing with huge character sets. Patch by Yann Vaginay.

- bpo-35943 [https://bugs.python.org/issue?@action=redirect&bpo=35943]: The function **`PyImport_GetModule()`** now ensures any module it returns is fully initialized. Patch by Joannah Nanjekye.

- bpo-32178 [https://bugs.python.org/issue?@action=redirect&bpo=32178]: Fix IndexError in **`email`** package when trying to parse invalid address fields starting with `:`.

- bpo-37268 [https://bugs.python.org/issue?@action=redirect&bpo=37268]: The **`parser`** module is deprecated and will be removed in future versions of Python.

- bpo-11953 [https://bugs.python.org/issue?@action=redirect&bpo=11953]: Completing WSA* error codes in **socket**.
- bpo-37685 [https://bugs.python.org/issue?@action=redirect&bpo=37685]: Fixed comparisons of **`datetime.timedelta`** and **`datetime.timezone`**.
- bpo-37697 [https://bugs.python.org/issue?@action=redirect&bpo=37697]: Synchronize `importlib.metadata` with importlib_metadata 0.19 [https://gitlab.com/python-devs/importlib_metadata/-/milestones/20], improving handling of EGG-INFO files and fixing a crash when entry point names contained colons.
- bpo-37695 [https://bugs.python.org/issue?@action=redirect&bpo=37695]: Correct **`curses.unget_wch()`** error message. Patch by Anthony Sottile.
- bpo-37689 [https://bugs.python.org/issue?@action=redirect&bpo=37689]: Add **`is_relative_to()`** in **`PurePath`** to determine whether or not one path is relative to another.
- bpo-29553 [https://bugs.python.org/issue?@action=redirect&bpo=29553]: Fixed **`argparse.ArgumentParser.format_usage()`** for mutually exclusive groups. Patch by Andrew Nester.
- bpo-37691 [https://bugs.python.org/issue?@action=redirect&bpo=37691]: Let math.dist() accept coordinates as sequences (or iterables) rather than just tuples.
- bpo-37685 [https://bugs.python.org/issue?@action=redirect&bpo=37685]: Fixed `__eq__`, `__lt__` etc implementations in some classes. They now return

`NotImplemented` for unsupported type of the other operand. This allows the other operand to play role (for example the equality comparison with `ANY` will return `True`).

- bpo-37354 [https://bugs.python.org/issue?@action=redirect&bpo=37354]: Make Activate.ps1 Powershell script static to allow for signing it.

- bpo-37664 [https://bugs.python.org/issue?@action=redirect&bpo=37664]: Update wheels bundled with ensurepip (pip 19.2.3 and setuptools 41.2.0)

- bpo-37663 [https://bugs.python.org/issue?@action=redirect&bpo=37663]: Bring consistency to venv shell activation scripts by always using __VENV_PROMPT__.

- bpo-37642 [https://bugs.python.org/issue?@action=redirect&bpo=37642]: Allowed the pure Python implementation of `datetime.timezone` to represent sub-minute offsets close to minimum and maximum boundaries, specifically in the ranges (23:59, 24:00) and (-23:59, 24:00). Patch by Ngalim Siregar

- bpo-36161 [https://bugs.python.org/issue?@action=redirect&bpo=36161]: In `posix`, use `ttyname_r` instead of `ttyname` for thread safety.

- bpo-36324 [https://bugs.python.org/issue?@action=redirect&bpo=36324]: Make internal attributes for statistics.NormalDist() private.

- bpo-37555 [https://bugs.python.org/issue?@action=redirect&bpo=37555]: Fix `NonCallableMock._call_matcher` returning tuple instead of `_Call` object when `self._spec_signature` exists. Patch by Elizabeth Uselton

- bpo-29446 [https://bugs.python.org/issue?@action=redirect&bpo=29446]: Make `from tkinter import *` import only the expected objects.

- bpo-16970 [https://bugs.python.org/issue?@action=redirect&bpo=16970]: Adding a value error when an invalid value in passed to nargs Patch by Robert Leenders

- bpo-34443 [https://bugs.python.org/issue?@action=redirect&bpo=34443]: Exceptions from **`enum`** now use the `__qualname` of the enum class in the exception message instead of the `__name__`.

- bpo-37491 [https://bugs.python.org/issue?@action=redirect&bpo=37491]: Fix `IndexError` when parsing email headers with unexpectedly ending bare-quoted string value. Patch by Abhilash Raj.

- bpo-37587 [https://bugs.python.org/issue?@action=redirect&bpo=37587]: Make json.loads faster for long strings. (Patch by Marco Paolini)

- bpo-18378 [https://bugs.python.org/issue?@action=redirect&bpo=18378]: Recognize “UTF-8” as a valid value for LC_CTYPE in locale._parse_localename.

- bpo-37579 [https://bugs.python.org/issue?@action=redirect&bpo=37579]: Return **`NotImplemented`** in Python implementation of `__eq__` for **`timedelta`** and **`time`** when the other object being compared is not of the same type to match C implementation. Patch by Karthikeyan Singaravelan.

- bpo-21478 [https://bugs.python.org/issue?@action=redirect&bpo=21478]: Record calls to parent when autospecced object is attached to a mock using

`unittest.mock.attach_mock()`. Patch by Karthikeyan Singaravelan.

- bpo-37531 [https://bugs.python.org/issue?@action=redirect&bpo=37531]: “python3 -m test -jN –timeout=TIMEOUT” now kills a worker process if it runs longer than *TIMEOUT* seconds.
- bpo-37482 [https://bugs.python.org/issue?@action=redirect&bpo=37482]: Fix serialization of display name in originator or destination address fields with both encoded words and special chars.
- bpo-36993 [https://bugs.python.org/issue?@action=redirect&bpo=36993]: Improve error reporting for corrupt zip files with bad zip64 extra data. Patch by Daniel Hillier.
- bpo-37502 [https://bugs.python.org/issue?@action=redirect&bpo=37502]: pickle.loads() no longer raises TypeError when the buffers argument is set to None
- bpo-37520 [https://bugs.python.org/issue?@action=redirect&bpo=37520]: Correct behavior for zipfile.Path.parent when the path object identifies a subdirectory.
- bpo-18374 [https://bugs.python.org/issue?@action=redirect&bpo=18374]: Fix the `.col_offset` attribute of nested **`ast.BinOp`** instances which had a too large value in some situations.
- bpo-37424 [https://bugs.python.org/issue?@action=redirect&bpo=37424]: Fixes a possible hang when using a timeout on `subprocess.run()` while capturing output. If the child process spawned its own children or otherwise connected its stdout or stderr handles with

another process, we could hang after the timeout was reached and our child was killed when attempting to read final output from the pipes.

- bpo-37421 [https://bugs.python.org/issue?@action=redirect&bpo=37421]: Fix **`multiprocessing.util.get_temp_dir()`** finalizer: clear also the ‘tempdir’ configuration of the current process, so next call to `get_temp_dir()` will create a new temporary directory, rather than reusing the removed temporary directory.

- bpo-37481 [https://bugs.python.org/issue?@action=redirect&bpo=37481]: The distutils `bdist_wininst` command is deprecated in Python 3.8, use `bdist_wheel` (wheel packages) instead.

- bpo-37479 [https://bugs.python.org/issue?@action=redirect&bpo=37479]: When `Enum.__str__` is overridden in a derived class, the override will be used by `Enum.__format__` regardless of whether mixin classes are present.

- bpo-37440 [https://bugs.python.org/issue?@action=redirect&bpo=37440]: http.client now enables TLS 1.3 post-handshake authentication for default context or if a cert_file is passed to HTTPSConnection.

- bpo-37437 [https://bugs.python.org/issue?@action=redirect&bpo=37437]: Update vendorized expat version to 2.2.7.

- bpo-37428 [https://bugs.python.org/issue?@action=redirect&bpo=37428]: SSLContext.post_handshake_auth = True no longer sets SSL_VERIFY_POST_HANDSHAKE verify flag for client connections. Although the option is documented as ignored for clients, OpenSSL implicitly enables cert chain validation when the flag is set.

- bpo-37420 [https://bugs.python.org/issue?@action=redirect&bpo=37420]: `os.sched_setaffinity()` now correctly handles errors that arise during iteration over its `mask` argument. Patch by Brandt Bucher.

- bpo-37412 [https://bugs.python.org/issue?@action=redirect&bpo=37412]: The `os.getcwdb()` function now uses the UTF-8 encoding on Windows, rather than the ANSI code page: see **PEP 529** [https://peps.python.org/pep-0529/] for the rationale. The function is no longer deprecated on Windows.

- bpo-37406 [https://bugs.python.org/issue?@action=redirect&bpo=37406]: The sqlite3 module now raises TypeError, rather than ValueError, if operation argument type is not str: execute(), executemany() and calling a connection.

- bpo-29412 [https://bugs.python.org/issue?@action=redirect&bpo=29412]: Fix IndexError in parsing a header value ending unexpectedly. Patch by Abhilash Raj.

- bpo-36546 [https://bugs.python.org/issue?@action=redirect&bpo=36546]: The *dist* argument for statistics.quantiles() is now positional only. The current name doesn’t reflect that the argument can be either a dataset or a distribution. Marking the parameter as positional avoids confusion and makes it possible to change the name later.

- bpo-37394 [https://bugs.python.org/issue?@action=redirect&bpo=37394]: Fix a bug that was causing the

`queue` module to fail if the accelerator module was not available. Patch by Pablo Galindo.

- bpo-37376 [https://bugs.python.org/issue?@action=redirect&bpo=37376]: **`pprint`** now has support for **`types.SimpleNamespace`**. Patch by Carl Bordum Hansen.

- bpo-26967 [https://bugs.python.org/issue?@action=redirect&bpo=26967]: An **`ArgumentParser`** with `allow_abbrev=False` no longer disables grouping of short flags, such as `-vv`, but only disables abbreviation of long flags as documented. Patch by Zac Hatfield-Dodds.

- bpo-37212 [https://bugs.python.org/issue?@action=redirect&bpo=37212]: **`unittest.mock.call()`** now preserves the order of keyword arguments in repr output. Patch by Karthikeyan Singaravelan.

- bpo-37372 [https://bugs.python.org/issue?@action=redirect&bpo=37372]: Fix error unpickling datetime.time objects from Python 2 with seconds>=24. Patch by Justin Blanchard.

- bpo-37345 [https://bugs.python.org/issue?@action=redirect&bpo=37345]: Add formal support for UDPLITE sockets. Support was present before, but it is now easier to detect support with `hasattr(socket, 'IPPROTO_UDPLITE')` and there are constants defined for each of the values needed: `socket.IPPROTO_UDPLITE`, `UDPLITE_SEND_CSCOV`, and `UDPLITE_RECV_CSCOV`. Patch by Gabe Appleton.

- bpo-37358 [https://bugs.python.org/issue?@action=redirect&bpo=37358]: Optimized `functools.partial` by using vectorcall.

- bpo-37347 [https://bugs.python.org/issue?@action=redirect&bpo=37347]:

`sqlite3.Connection.create_aggregate()`, `sqlite3.Connection.create_function()`, `sqlite3.Connection.set_authorizer()`, `sqlite3.Connection.set_progress_handler()` `sqlite3.Connection.set_trace_callback()` methods lead to segfaults if some of these methods are called twice with an equal object but not the same. Now callbacks are stored more carefully. Patch by Aleksandr Balezin.

- bpo-37163 [https://bugs.python.org/issue?@action=redirect&bpo=37163]: The *obj* argument of `dataclasses.replace()` is positional-only now.
- bpo-37085 [https://bugs.python.org/issue?@action=redirect&bpo=37085]: Add the optional Linux SocketCAN Broadcast Manager constants, used as flags to configure the BCM behaviour, in the socket module. Patch by Karl Ding.
- bpo-37328 [https://bugs.python.org/issue?@action=redirect&bpo=37328]: `HTMLParser.unescape` is removed. It was undocumented and deprecated since Python 3.4.
- bpo-37305 [https://bugs.python.org/issue?@action=redirect&bpo=37305]: Add .webmanifest -> application/manifest+json to list of recognized file types and content type headers
- bpo-37320 [https://bugs.python.org/issue?@action=redirect&bpo=37320]: `aifc.openfp()` alias to `aifc.open()`, `sunau.openfp()` alias to `sunau.open()`, and `wave.openfp()` alias to `wave.open()` have been removed. They were deprecated since Python 3.7.
- bpo-37315 [https://bugs.python.org/issue?@action=redirect&bpo=37315]: Deprecated accepting floats with integral value (like `5.0`) in `math.factorial()`.

- bpo-37312 [https://bugs.python.org/issue?@action=redirect&bpo=37312]: `_dummy_thread` and `dummy_threading` modules have been removed. These modules were deprecated since Python 3.7 which requires threading support.

- bpo-33972 [https://bugs.python.org/issue?@action=redirect&bpo=33972]: Email with single part but content-type set to `multipart/*` doesn’t raise AttributeError anymore.

- bpo-37280 [https://bugs.python.org/issue?@action=redirect&bpo=37280]: Use threadpool for reading from file for sendfile fallback mode.

- bpo-37279 [https://bugs.python.org/issue?@action=redirect&bpo=37279]: Fix asyncio sendfile support when sendfile sends extra data in fallback mode.

- bpo-19865 [https://bugs.python.org/issue?@action=redirect&bpo=19865]: **`ctypes.create_unicode_buffer()`** now also supports non-BMP characters on platforms with 16-bit **`wchar_t`** (for example, Windows and AIX).

- bpo-37266 [https://bugs.python.org/issue?@action=redirect&bpo=37266]: In a subinterpreter, spawning a daemon thread now raises an exception. Daemon threads were never supported in subinterpreters. Previously, the subinterpreter finalization crashed with a Python fatal error if a daemon thread was still running.

- bpo-37210 [https://bugs.python.org/issue?@action=redirect&bpo=37210]: Allow pure Python implementation of **`pickle`** to work even when the C **`_pickle`** module is unavailable.

- bpo-21872 [https://bugs.python.org/issue?@action=redirect&bpo=21872]: Fix **lzma**: module decompresses data incompletely. When decompressing a FORMAT_ALONE format file, and it doesn't have the end marker, sometimes the last one to dozens bytes can't be output. Patch by Ma Lin.

- bpo-35922 [https://bugs.python.org/issue?@action=redirect&bpo=35922]: Fix **`RobotFileParser.crawl_delay()`** and **`RobotFileParser.request_rate()`** to return `None` rather than raise **`AttributeError`** when no relevant rule is defined in the robots.txt file. Patch by Rémi Lapeyre.

- bpo-35766 [https://bugs.python.org/issue?@action=redirect&bpo=35766]: Change the format of feature_version to be a (major, minor) tuple.

- bpo-36607 [https://bugs.python.org/issue?@action=redirect&bpo=36607]: Eliminate **`RuntimeError`** raised by **`asyncio.all_tasks()`** if internal tasks weak set is changed by another thread during iteration.

- bpo-18748 [https://bugs.python.org/issue?@action=redirect&bpo=18748]: **`_pyio.IOBase`** destructor now does nothing if getting the `closed` attribute fails to better mimic **`_io.IOBase`** finalizer.

- bpo-36402 [https://bugs.python.org/issue?@action=redirect&bpo=36402]: Fix a race condition at Python shutdown when waiting for threads. Wait until the Python thread state of all non-daemon threads get deleted (join all non-daemon threads), rather than just wait until non-daemon Python threads complete.

- bpo-37206 [https://bugs.python.org/issue?@action=redirect&bpo=37206]: Default values which cannot be represented as Python objects no longer improperly represented as `None` in function signatures.

- bpo-37111 [https://bugs.python.org/issue?@action=redirect&bpo=37111]: Added `encoding` and `errors` keyword parameters to `logging.basicConfig`.

- bpo-12144 [https://bugs.python.org/issue?@action=redirect&bpo=12144]: Ensure cookies with `expires` attribute are handled in **`CookieJar.make_cookies()`**.

- bpo-34886 [https://bugs.python.org/issue?@action=redirect&bpo=34886]: Fix an unintended ValueError from **`subprocess.run()`** when checking for conflicting *input* and *stdin* or *capture_output* and *stdout* or *stderr* args when they were explicitly provided but with `None` values within a passed in `**kwargs` dict rather than as passed directly by name. Patch contributed by Rémi Lapeyre.

- bpo-37173 [https://bugs.python.org/issue?@action=redirect&bpo=37173]: The exception message for `inspect.getfile()` now correctly reports the passed class rather than the builtins module.

- bpo-37178 [https://bugs.python.org/issue?@action=redirect&bpo=37178]: Give math.perm() a one argument form that means the same as math.factorial().

- bpo-37178 [https://bugs.python.org/issue?@action=redirect&bpo=37178]: For math.perm(n, k), let k default to n, giving the same result as factorial.

- bpo-37165 [https://bugs.python.org/issue?@action=redirect&bpo=37165]: Converted _collections._count_elements to use the Argument Clinic.

- bpo-34767 [https://bugs.python.org/issue?@action=redirect&bpo=34767]: Do not always create a `collections.deque` in `asyncio.Lock`.

- bpo-37158 [https://bugs.python.org/issue?@action=redirect&bpo=37158]: Speed-up statistics.fmean() by switching from a function to a generator.

- bpo-34282 [https://bugs.python.org/issue?@action=redirect&bpo=34282]: Remove `Enum._convert` method, deprecated in 3.8.

- bpo-37150 [https://bugs.python.org/issue?@action=redirect&bpo=37150]: `argparse._ActionsContainer.add_argument` now throws error, if someone accidentally pass FileType class object instead of instance of FileType as `type` argument.

- bpo-28724 [https://bugs.python.org/issue?@action=redirect&bpo=28724]: The socket module now has the `socket.send_fds()` and **`socket.recv.fds()`** methods. Contributed by Joannah Nanjekye, Shinya Okano and Victor Stinner.

- bpo-35621 [https://bugs.python.org/issue?@action=redirect&bpo=35621]: Support running asyncio subprocesses when execution event loop in a thread on UNIX.

- bpo-36520 [https://bugs.python.org/issue?@action=redirect&bpo=36520]: Lengthy email headers with UTF-8 characters are now properly encoded when they are folded. Patch by Jeffrey Kintscher.

- bpo-30835 [https://bugs.python.org/issue?@action=redirect&bpo=30835]: Fixed a bug in email parsing where a message with invalid bytes in content-transfer-encoding of a multipart message can cause an AttributeError. Patch by Andrew Donnellan.

- bpo-31163 [https://bugs.python.org/issue?@action=redirect&bpo=31163]: pathlib.Path instance's rename and replace methods now return the new Path instance.

- bpo-25068 [https://bugs.python.org/issue?@action=redirect&bpo=25068]: **`urllib.request.ProxyHandler`** now lowercases the keys of the passed dictionary.

- bpo-26185 [https://bugs.python.org/issue?@action=redirect&bpo=26185]: Fix **`repr()`** on empty **`ZipInfo`** object. Patch by Mickaël Schoentgen.

- bpo-21315 [https://bugs.python.org/issue?@action=redirect&bpo=21315]: Email headers containing RFC2047 encoded words are parsed despite the missing whitespace, and a defect registered. Also missing trailing whitespace after encoded words is now registered as a defect.

- bpo-31904 [https://bugs.python.org/issue?@action=redirect&bpo=31904]: Port test_datetime to VxWorks: skip zoneinfo tests on VxWorks

- bpo-35805 [https://bugs.python.org/issue?@action=redirect&bpo=35805]: Add parser for Message-ID header and add it to default HeaderRegistry. This should prevent folding of Message-ID using RFC 2048 encoded words.

- bpo-36871 [https://bugs.python.org/issue?@action=redirect&bpo=36871]: Ensure method signature is used instead of constructor signature of a class while asserting mock object against method calls. Patch by Karthikeyan Singaravelan.

- bpo-35070 [https://bugs.python.org/issue?@action=redirect&bpo=35070]: posix.getgrouplist() now works correctly when the user belongs to NGROUPS_MAX supplemental groups. Patch by Jeffrey Kintscher.
- bpo-31783 [https://bugs.python.org/issue?@action=redirect&bpo=31783]: Fix race condition in ThreadPoolExecutor when worker threads are created during interpreter shutdown.
- bpo-36582 [https://bugs.python.org/issue?@action=redirect&bpo=36582]: Fix `UserString.encode()` to correctly return `bytes` rather than a `UserString` instance.
- bpo-32424 [https://bugs.python.org/issue?@action=redirect&bpo=32424]: Deprecate xml.etree.ElementTree.Element.copy() in favor of copy.copy().

  Patch by Gordon P. Hemsley
- bpo-36564 [https://bugs.python.org/issue?@action=redirect&bpo=36564]: Fix infinite loop in email header folding logic that would be triggered when an email policy’s max_line_length is not long enough to include the required markup and any values in the message. Patch by Paul Ganssle
- bpo-36543 [https://bugs.python.org/issue?@action=redirect&bpo=36543]: Removed methods Element.getchildren(), Element.getiterator() and ElementTree.getiterator() and the xml.etree.cElementTree module.
- bpo-36409 [https://bugs.python.org/issue?@action=redirect&bpo=36409]: Remove the old plistlib API deprecated in Python 3.4

- bpo-36302 [https://bugs.python.org/issue?@action=redirect&bpo=36302]: distutils sorts source file lists so that Extension .so files build more reproducibly by default
- bpo-36250 [https://bugs.python.org/issue?@action=redirect&bpo=36250]: Ignore `ValueError` from `signal` with `interaction` in non-main thread.
- bpo-36046 [https://bugs.python.org/issue?@action=redirect&bpo=36046]: Added `user`, `group` and `extra_groups` parameters to the subprocess.Popen constructor. Patch by Patrick McLean.
- bpo-32627 [https://bugs.python.org/issue?@action=redirect&bpo=32627]: Fix compile error when `_uuid` headers conflicting included.
- bpo-35800 [https://bugs.python.org/issue?@action=redirect&bpo=35800]: Deprecate `smtpd.MailmanProxy` ready for future removal.
- bpo-35168 [https://bugs.python.org/issue?@action=redirect&bpo=35168]: **`shlex.shlex.punctuation_chars`** is now a read-only property.
- bpo-8538 [https://bugs.python.org/issue?@action=redirect&bpo=8538]: Add support for boolean actions like `--foo` and `--no-foo` to argparse. Patch contributed by Rémi Lapeyre.
- bpo-20504 [https://bugs.python.org/issue?@action=redirect&bpo=20504]: Fixes a bug in **`cgi`** module when a multipart/form-data request has no `Content-Length` header.
- bpo-25988 [https://bugs.python.org/issue?@action=redirect&bpo=25988]: The abstract base classes in

`collections.abc` no longer are exposed in the regular `collections` module.

- bpo-11122 [https://bugs.python.org/issue?@action=redirect&bpo=11122]: Distutils won’t check for rpmbuild in specified paths only.
- bpo-34775 [https://bugs.python.org/issue?@action=redirect&bpo=34775]: Division handling of PurePath now returns NotImplemented instead of raising a TypeError when passed something other than an instance of str or PurePath. Patch by Roger Aiudi.
- bpo-34749 [https://bugs.python.org/issue?@action=redirect&bpo=34749]: `binascii.a2b_base64()` is now up to 2 times faster. Patch by Sergey Fedoseev.
- bpo-34519 [https://bugs.python.org/issue?@action=redirect&bpo=34519]: Add additional aliases for HP Roman 8. Patch by Michael Osipov.
- bpo-28009 [https://bugs.python.org/issue?@action=redirect&bpo=28009]: Fix uuid.getnode() on platforms with ‘.’ as MAC Addr delimiter as well fix for MAC Addr format that omits a leading 0 in MAC Addr values. Currently, AIX is the only know platform with these settings. Patch by Michael Felt.
- bpo-30618 [https://bugs.python.org/issue?@action=redirect&bpo=30618]: Add `readlink()`. Patch by Girts Folkmanis.
- bpo-32498 [https://bugs.python.org/issue?@action=redirect&bpo=32498]: Made `urllib.parse.unquote()` accept bytes in addition to strings. Patch by Stein Karlsen.

- bpo-33348 [https://bugs.python.org/issue?@action=redirect&bpo=33348]: lib2to3 now recognizes expressions after `*` and `**` like in `f(*[] or [])`.

- bpo-32689 [https://bugs.python.org/issue?@action=redirect&bpo=32689]: Update `shutil.move()` function to allow for Path objects to be used as source argument. Patch by Emily Morehouse and Maxwell “5.13b” McKinnon.

- bpo-32820 [https://bugs.python.org/issue?@action=redirect&bpo=32820]: Added `__format__` to IPv4 and IPv6 classes. Always outputs a fully zero-padded string. Supports b/x/n modifiers (bin/hex/native format). Native format for IPv4 is bin, native format for IPv6 is hex. Also supports ‘#’ and ‘_’ modifiers.

- bpo-27657 [https://bugs.python.org/issue?@action=redirect&bpo=27657]: Fix urllib.parse.urlparse() with numeric paths. A string like “path:80” is no longer parsed as a path but as a scheme (“path”) and a path (“80”).

- bpo-4963 [https://bugs.python.org/issue?@action=redirect&bpo=4963]: Fixed non-deterministic behavior related to mimetypes extension mapping and module reinitialization.

Documentation

- bpo-21767 [https://bugs.python.org/issue?@action=redirect&bpo=21767]: Explicitly mention abc support in functools.singledispatch

- bpo-38816 [https://bugs.python.org/issue?@action=redirect&bpo=38816]: Provides more details about the interaction between **`fork()`** and CPython’s runtime, focusing just on the C-API. This includes cautions about where `fork()` should and shouldn’t be called.

- bpo-38351 [https://bugs.python.org/issue?@action=redirect&bpo=38351]: Modernize `email` examples from %-formatting to f-strings.

- bpo-38778 [https://bugs.python.org/issue?@action=redirect&bpo=38778]: Document the fact that `RuntimeError` is raised if `os.fork()` is called in a subinterpreter.

- bpo-38592 [https://bugs.python.org/issue?@action=redirect&bpo=38592]: Add Brazilian Portuguese to the language switcher at Python Documentation website.

- bpo-38294 [https://bugs.python.org/issue?@action=redirect&bpo=38294]: Add list of no-longer-escaped chars to re.escape documentation

- bpo-38053 [https://bugs.python.org/issue?@action=redirect&bpo=38053]: Modernized the plistlib documentation

- bpo-26868 [https://bugs.python.org/issue?@action=redirect&bpo=26868]: Fix example usage of `PyModule_AddObject()` to properly handle errors.

- bpo-36797 [https://bugs.python.org/issue?@action=redirect&bpo=36797]: Fix a dead link in the distutils API Reference.

- bpo-37977 [https://bugs.python.org/issue?@action=redirect&bpo=37977]: Warn more strongly and clearly about pickle insecurity

- bpo-37979 [https://bugs.python.org/issue?@action=redirect&bpo=37979]: Added a link to dateutil.parser.isoparse in the datetime.fromisoformat documentation. Patch by Paul Ganssle
- bpo-12707 [https://bugs.python.org/issue?@action=redirect&bpo=12707]: Deprecate info(), geturl(), getcode() methods in favor of the headers, url, and status properties, respectively, for HTTPResponse and addinfourl. Also deprecate the code attribute of addinfourl in favor of the status attribute. Patch by Ashwin Ramaswami
- bpo-37937 [https://bugs.python.org/issue?@action=redirect&bpo=37937]: Mention `frame.f_trace` in **`sys.settrace()`** docs.
- bpo-37878 [https://bugs.python.org/issue?@action=redirect&bpo=37878]: Make **`PyThreadState_DeleteCurrent()`** Internal.
- bpo-37759 [https://bugs.python.org/issue?@action=redirect&bpo=37759]: Beginning edits to Whatsnew 3.8
- bpo-37726 [https://bugs.python.org/issue?@action=redirect&bpo=37726]: Stop recommending getopt in the tutorial for command line argument parsing and promote argparse.
- bpo-32910 [https://bugs.python.org/issue?@action=redirect&bpo=32910]: Remove implementation-specific behaviour of how venv’s Deactivate works.
- bpo-37256 [https://bugs.python.org/issue?@action=redirect&bpo=37256]: Fix wording of arguments for **`Request`** in **`urllib.request`**

- bpo-37284 [https://bugs.python.org/issue?@action=redirect&bpo=37284]: Add a brief note to indicate that any new `sys.implementation` required attributes must go through the PEP process.

- bpo-30088 [https://bugs.python.org/issue?@action=redirect&bpo=30088]: Documented that **`mailbox.Maildir`** constructor doesn't attempt to verify the maildir folder layout correctness. Patch by Sviatoslav Sydorenko.

- bpo-37521 [https://bugs.python.org/issue?@action=redirect&bpo=37521]: Fix `importlib` examples to insert any newly created modules via importlib.util.module_from_spec() immediately into sys.modules instead of after calling loader.exec_module().

  Thanks to Benjamin Mintz for finding the bug.

- bpo-37456 [https://bugs.python.org/issue?@action=redirect&bpo=37456]: Slash ('/') is now part of syntax.

- bpo-37487 [https://bugs.python.org/issue?@action=redirect&bpo=37487]: Fix PyList_GetItem index description to include 0.

- bpo-37149 [https://bugs.python.org/issue?@action=redirect&bpo=37149]: Replace the dead link to the Tkinter 8.5 reference by John Shipman, New Mexico Tech, with a link to the archive.org copy.

- bpo-37478 [https://bugs.python.org/issue?@action=redirect&bpo=37478]: Added possible exceptions to the description of os.chdir().

- bpo-34903 [https://bugs.python.org/issue?@action=redirect&bpo=34903]: Documented that in **`datetime.datetime.strptime()`**, the leading zero in some two-digit formats is optional. Patch by Mike Gleen.

- bpo-36260 [https://bugs.python.org/issue?@action=redirect&bpo=36260]: Add decompression pitfalls to zipfile module documentation.

- bpo-37004 [https://bugs.python.org/issue?@action=redirect&bpo=37004]: In the documentation for difflib, a note was added explicitly warning that the results of SequenceMatcher's ratio method may depend on the order of the input strings.

- bpo-36960 [https://bugs.python.org/issue?@action=redirect&bpo=36960]: Restructured the **`datetime`** docs in the interest of making them more user-friendly and improving readability. Patch by Brad Solomon.

- bpo-36487 [https://bugs.python.org/issue?@action=redirect&bpo=36487]: Make C-API docs clear about what the "main" interpreter is.

- bpo-23460 [https://bugs.python.org/issue?@action=redirect&bpo=23460]: The documentation for decimal string formatting using the `:g` specifier has been updated to reflect the correct exponential notation cutoff point. Original patch contributed by Tuomas Suutari.

- bpo-35803 [https://bugs.python.org/issue?@action=redirect&bpo=35803]: Document and test that `tempfile` functions may accept a path-like object for the `dir` argument. Patch by Anthony Sottile.

- bpo-33944 [https://bugs.python.org/issue?@action=redirect&bpo=33944]: Added a note about the intended use of code in .pth files.

- bpo-34293 [https://bugs.python.org/issue?@action=redirect&bpo=34293]: Fix the Doc/Makefile regarding PAPER environment variable and PDF builds

- bpo-25237 [https://bugs.python.org/issue?@action=redirect&bpo=25237]: Add documentation for tkinter modules

Tests

- bpo-38614 [https://bugs.python.org/issue?@action=redirect&bpo=38614]: Fix test_communicate() of test_asyncio.test_subprocess: use `support.LONG_TIMEOUT` (5 minutes), instead of just 1 minute.

- bpo-38614 [https://bugs.python.org/issue?@action=redirect&bpo=38614]: Add timeout constants to **`test.support`**: **`LOOPBACK_TIMEOUT`**, **`INTERNET_TIMEOUT`**, **`SHORT_TIMEOUT`** and **`LONG_TIMEOUT`**.

- bpo-38502 [https://bugs.python.org/issue?@action=redirect&bpo=38502]: test.regrtest now uses process groups in the multiprocessing mode (-jN command line option) if process groups are available: if **`os.setsid()`** and **`os.killpg()`** functions are available.

- bpo-35998 [https://bugs.python.org/issue?@action=redirect&bpo=35998]: Fix a race condition in test_asyncio.test_start_tls_server_1(). Previously, there was a race condition between the test main() function which replaces the protocol and the test ServerProto protocol which sends ANSWER once it gets HELLO. Now, only the test main() function is responsible to send data, ServerProto no longer sends data.

- bpo-38470 [https://bugs.python.org/issue?@action=redirect&bpo=38470]: Fix `test_compileall.test_compile_dir_maxlevels()` on Windows without long path support: only create 3 subdirectories instead of between 20 and 100 subdirectories.

- bpo-37531 [https://bugs.python.org/issue?@action=redirect&bpo=37531]: On timeout, regrtest no longer attempts to call `popen.communicate()` again: it can hang until all child processes using stdout and stderr pipes completes. Kill the worker process and ignores its output. Change also the faulthandler timeout of the main process from 1 minute to 5 minutes, for Python slowest buildbots.

- bpo-38239 [https://bugs.python.org/issue?@action=redirect&bpo=38239]: Fix test_gdb for Link Time Optimization (LTO) builds.

- bpo-38275 [https://bugs.python.org/issue?@action=redirect&bpo=38275]: test_ssl now handles disabled TLS/SSL versions better. OpenSSL’s crypto policy and run-time settings are recognized and tests for disabled versions are skipped. Tests also accept more TLS minimum_versions for platforms that override OpenSSL’s default with strict settings.

- bpo-38271 [https://bugs.python.org/issue?@action=redirect&bpo=38271]: The private keys for test_ssl were encrypted with 3DES in traditional PKCS#5 format. 3DES and the digest algorithm of PKCS#5 are blocked by some strict crypto policies. Use PKCS#8 format with AES256 encryption instead.

- bpo-38270 [https://bugs.python.org/issue?@action=redirect&bpo=38270]: test.support now has a helper function to check for availability of a hash digest function. Several tests are refactored avoid MD5 and use SHA256 instead. Other tests are marked to use MD5 and skipped when MD5 is disabled.
- bpo-37123 [https://bugs.python.org/issue?@action=redirect&bpo=37123]: Multiprocessing test test_mymanager() now also expects -SIGTERM, not only exitcode 0. BaseManager._finalize_manager() sends SIGTERM to the manager process if it takes longer than 1 second to stop, which happens on slow buildbots.
- bpo-38212 [https://bugs.python.org/issue?@action=redirect&bpo=38212]: Multiprocessing tests: increase test_queue_feeder_donot_stop_onexc() timeout from 1 to 60 seconds.
- bpo-38117 [https://bugs.python.org/issue?@action=redirect&bpo=38117]: Test with OpenSSL 1.1.1d
- bpo-38018 [https://bugs.python.org/issue?@action=redirect&bpo=38018]: Increase code coverage for multiprocessing.shared_memory.
- bpo-37805 [https://bugs.python.org/issue?@action=redirect&bpo=37805]: Add tests for json.dump(…, skipkeys=True). Patch by Donghee Na.
- bpo-37531 [https://bugs.python.org/issue?@action=redirect&bpo=37531]: Enhance regrtest multiprocess timeout: write a message when killing a worker process, catch popen.kill() and popen.wait() exceptions, put a timeout on the second call to popen.communicate().

- bpo-37876 [https://bugs.python.org/issue?@action=redirect&bpo=37876]: Add tests for ROT-13 codec.

- bpo-36833 [https://bugs.python.org/issue?@action=redirect&bpo=36833]: Added tests for `PyDateTime_xxx_GET_xxx()` macros of the C API of the **`datetime`** module. Patch by Joannah Nanjekye.

- bpo-37558 [https://bugs.python.org/issue?@action=redirect&bpo=37558]: Fix test_shared_memory_cleaned_after_process_termination name handling

- bpo-37526 [https://bugs.python.org/issue?@action=redirect&bpo=37526]: Add **`test.support.catch_threading_exception()`**: context manager catching **`threading.Thread`** exception using **`threading.excepthook()`**.

- bpo-37421 [https://bugs.python.org/issue?@action=redirect&bpo=37421]: test_concurrent_futures now explicitly stops the ForkServer instance if it’s running.

- bpo-37421 [https://bugs.python.org/issue?@action=redirect&bpo=37421]: multiprocessing tests now stop the ForkServer instance if it’s running: close the “alive” file descriptor to ask the server to stop and then remove its UNIX address.

- bpo-37421 [https://bugs.python.org/issue?@action=redirect&bpo=37421]: test_distutils.test_build_ext() is now able to remove the temporary directory on Windows: don’t import the newly built C extension (“xx”) in the current process, but test it in a separated process.

- bpo-37421 [https://bugs.python.org/issue?@action=redirect&bpo=37421]: test_concurrent_futures now cleans up multiprocessing to remove immediately temporary directories created by multiprocessing.util.get_temp_dir().

- bpo-37421 [https://bugs.python.org/issue?@action=redirect&bpo=37421]: test_winconsoleio doesn’t leak a temporary file anymore: use tempfile.TemporaryFile() to remove it when the test completes.

- bpo-37421 [https://bugs.python.org/issue?@action=redirect&bpo=37421]: multiprocessing tests now explicitly call `_run_finalizers()` to immediately remove temporary directories created by tests.

- bpo-37421 [https://bugs.python.org/issue?@action=redirect&bpo=37421]: urllib.request tests now call **`urlcleanup()`** to remove temporary files created by `urlretrieve()` tests and to clear the `_opener` global variable set by `urlopen()` and functions calling indirectly `urlopen()`.

- bpo-37472 [https://bugs.python.org/issue?@action=redirect&bpo=37472]: Remove `Lib/test/outstanding_bugs.py`.

- bpo-37199 [https://bugs.python.org/issue?@action=redirect&bpo=37199]: Fix test failures when IPv6 is unavailable or disabled.

- bpo-19696 [https://bugs.python.org/issue?@action=redirect&bpo=19696]: Replace deprecated method “random.choose” with “random.choice” in “test_pkg_import.py”.

- bpo-37335 [https://bugs.python.org/issue?@action=redirect&bpo=37335]: Remove no longer necessary code from c locale coercion tests
- bpo-37421 [https://bugs.python.org/issue?@action=redirect&bpo=37421]: Fix test_shutil to no longer leak temporary files.
- bpo-37411 [https://bugs.python.org/issue?@action=redirect&bpo=37411]: Fix test_wsgiref.testEnviron() to no longer depend on the environment variables (don’t fail if “X” variable is set).
- bpo-37400 [https://bugs.python.org/issue?@action=redirect&bpo=37400]: Fix test_os.test_chown(): use os.getgroups() rather than grp.getgrall() to get groups. Rename also the test to test_chown_gid().
- bpo-37359 [https://bugs.python.org/issue?@action=redirect&bpo=37359]: Add –cleanup option to python3 -m test to remove `test_python_*` directories of previous failed jobs. Add “make cleantest” to run `python3 -m test --cleanup`.
- bpo-37362 [https://bugs.python.org/issue?@action=redirect&bpo=37362]: test_gdb no longer fails if it gets an “unexpected” message on stderr: it now ignores stderr. The purpose of test_gdb is to test that python-gdb.py commands work as expected, not to test gdb.
- bpo-35998 [https://bugs.python.org/issue?@action=redirect&bpo=35998]: Avoid TimeoutError in test_asyncio: test_start_tls_server_1()
- bpo-37278 [https://bugs.python.org/issue?@action=redirect&bpo=37278]: Fix test_asyncio

ProactorLoopCtrlC: join the thread to prevent leaking a running thread and leaking a reference.

- bpo-37261 [https://bugs.python.org/issue?@action=redirect&bpo=37261]: Fix `test.support.catch_unraisable_exception()`: its __exit__() method now ignores unraisable exception raised when clearing its `unraisable` attribute.
- bpo-37069 [https://bugs.python.org/issue?@action=redirect&bpo=37069]: regrtest now uses `sys.unraisablehook()` to mark a test as “environment altered” (ENV_CHANGED) if it emits an “unraisable exception”. Moreover, regrtest logs a warning in this case.

  Use `python3 -m test --fail-env-changed` to catch unraisable exceptions in tests.
- bpo-37252 [https://bugs.python.org/issue?@action=redirect&bpo=37252]: Fix assertions in `test_close` and `test_events_mask_overflow` devpoll tests.
- bpo-37169 [https://bugs.python.org/issue?@action=redirect&bpo=37169]: Rewrite `_PyObject_IsFreed()` unit tests.
- bpo-37153 [https://bugs.python.org/issue?@action=redirect&bpo=37153]: `test_venv.test_multiprocessing()` now explicitly calls `pool.terminate()` to wait until the pool completes.
- bpo-34001 [https://bugs.python.org/issue?@action=redirect&bpo=34001]: Make test_ssl pass with LibreSSL. LibreSSL handles minimum and maximum TLS version differently than OpenSSL.

- bpo-36919 [https://bugs.python.org/issue?@action=redirect&bpo=36919]: Make `test_source_encoding.test_issue2301` implementation independent. The test will work now for both CPython and IronPython.

- bpo-30202 [https://bugs.python.org/issue?@action=redirect&bpo=30202]: Update `test.test_importlib.test_abc` to test `find_spec()`.

- bpo-28009 [https://bugs.python.org/issue?@action=redirect&bpo=28009]: Modify the test_uuid logic to test when a program is available AND can be used to obtain a MACADDR as basis for an UUID. Patch by M. Felt

- bpo-34596 [https://bugs.python.org/issue?@action=redirect&bpo=34596]: Fallback to a default reason when **`unittest.skip()`** is uncalled. Patch by Naitree Zhu.

Build

- bpo-38809 [https://bugs.python.org/issue?@action=redirect&bpo=38809]: On Windows, build scripts will now recognize and use python.exe from an active virtual env.

- bpo-38684 [https://bugs.python.org/issue?@action=redirect&bpo=38684]: Fix _hashlib build when Blake2 is disabled, but OpenSSL supports it.

- bpo-38468 [https://bugs.python.org/issue?@action=redirect&bpo=38468]: Misc/python-config.in now uses `getvar()` for all still existing `sysconfig.get_config_var()` calls. Patch by Joannah Nanjekye.

- bpo-37415 [https://bugs.python.org/issue?@action=redirect&bpo=37415]: Fix stdatomic.h header check for ICC compiler: the ICC implementation lacks atomic_uintptr_t type which is needed by Python.

- bpo-38301 [https://bugs.python.org/issue?@action=redirect&bpo=38301]: In Solaris family, we must be sure to use `-D_REENTRANT`. Patch by Jesús Cea Avión.

- bpo-36002 [https://bugs.python.org/issue?@action=redirect&bpo=36002]: Locate `llvm-profdata` and `llvm-ar` binaries using `AC_PATH_TOOL` rather than `AC_PATH_TARGET_TOOL`.

- bpo-37936 [https://bugs.python.org/issue?@action=redirect&bpo=37936]: The `.gitignore` file systematically keeps “rooted”, with a non-trailing slash, all the rules that are meant to apply to files in a specific place in the repo. Previously, when the intended file to ignore happened to be at the root of the repo, we’d most often accidentally also ignore files and directories with the same name anywhere in the tree.

- bpo-37760 [https://bugs.python.org/issue?@action=redirect&bpo=37760]: The `Tools/unicode/makeunicodedata.py` script, which is used for converting information from the Unicode Character Database into generated code and data used by the methods of **`str`** and by the **`unicodedata`** module, now handles each character’s data as a `dataclass` with named attributes, rather than a length-18 list of different fields.

- bpo-37936 [https://bugs.python.org/issue?@action=redirect&bpo=37936]: The `.gitignore` file no longer applies to any files that are in fact tracked in the Git repository. Patch by Greg Price.

- bpo-37725 [https://bugs.python.org/issue?@action=redirect&bpo=37725]: Change “clean” makefile target to also clean the program guided optimization (PGO) data. Previously you would have to use “make clean” and “make profile-removal”, or “make clobber”.

- bpo-37707 [https://bugs.python.org/issue?@action=redirect&bpo=37707]: Mark some individual tests to skip when –pgo is used. The tests marked increase the PGO task time significantly and likely don’t help improve optimization of the final executable.

- bpo-36044 [https://bugs.python.org/issue?@action=redirect&bpo=36044]: Reduce the number of unit tests run for the PGO generation task. This speeds up the task by a factor of about 15x. Running the full unit test suite is slow. This change may result in a slightly less optimized build since not as many code branches will be executed. If you are willing to wait for the much slower build, the old behavior can be restored using ‘./configure [..] PROFILE_TASK=”-m test –pgo-extended”’. We make no guarantees as to which PGO task set produces a faster build. Users who care should run their own relevant benchmarks as results can depend on the environment, workload, and compiler tool chain.

- bpo-37468 [https://bugs.python.org/issue?@action=redirect&bpo=37468]: `make install` no longer installs `wininst-*.exe` files used by distutils bdist_wininst: bdist_wininst only works on Windows.

- bpo-37189 [https://bugs.python.org/issue?@action=redirect&bpo=37189]: Many `PyRun_XXX()` functions like **`PyRun_String()`** were no longer exported in `libpython38.dll` by mistake. Export them again to fix the ABI compatibility.

- bpo-25361 [https://bugs.python.org/issue?@action=redirect&bpo=25361]: Enables use of SSE2 instructions in Windows 32-bit build.

- bpo-36210 [https://bugs.python.org/issue?@action=redirect&bpo=36210]: Update optional extension module detection for AIX. ossaudiodev and spwd are not applicable for AIX, and are no longer reported as missing. 3rd-party packaging of ncurses (with ASIS support) conflicts with officially supported AIX curses library, so configure AIX to use libcurses.a. However, skip trying to build _curses_panel.

  patch by M Felt

Windows

- bpo-38589 [https://bugs.python.org/issue?@action=redirect&bpo=38589]: Fixes HTML Help shortcut when Windows is not installed to C drive
- bpo-38453 [https://bugs.python.org/issue?@action=redirect&bpo=38453]: Ensure ntpath.realpath() correctly resolves relative paths.
- bpo-38519 [https://bugs.python.org/issue?@action=redirect&bpo=38519]: Restores the internal C headers that were missing from the nuget.org and Microsoft Store packages.
- bpo-38492 [https://bugs.python.org/issue?@action=redirect&bpo=38492]: Remove `pythonw.exe` dependency on the Microsoft C++ runtime.
- bpo-38344 [https://bugs.python.org/issue?@action=redirect&bpo=38344]: Fix error message in activate.bat

- bpo-38359 [https://bugs.python.org/issue?@action=redirect&bpo=38359]: Ensures `pyw.exe` launcher reads correct registry key.
- bpo-38355 [https://bugs.python.org/issue?@action=redirect&bpo=38355]: Fixes `ntpath.realpath` failing on `sys.executable`.
- bpo-38117 [https://bugs.python.org/issue?@action=redirect&bpo=38117]: Update bundled OpenSSL to 1.1.1d
- bpo-38092 [https://bugs.python.org/issue?@action=redirect&bpo=38092]: Reduce overhead when using multiprocessing in a Windows virtual environment.
- bpo-38133 [https://bugs.python.org/issue?@action=redirect&bpo=38133]: Allow py.exe launcher to locate installations from the Microsoft Store and improve display of active virtual environments.
- bpo-38114 [https://bugs.python.org/issue?@action=redirect&bpo=38114]: The `pip.ini` is no longer included in the Nuget package.
- bpo-32592 [https://bugs.python.org/issue?@action=redirect&bpo=32592]: Set Windows 8 as the minimum required version for API support
- bpo-36634 [https://bugs.python.org/issue?@action=redirect&bpo=36634]: **`os.cpu_count()`** now returns active processors rather than maximum processors.
- bpo-36634 [https://bugs.python.org/issue?@action=redirect&bpo=36634]: venv activate.bat now works when the existing variables contain double quote characters.
- bpo-38081 [https://bugs.python.org/issue?@action=redirect&bpo=38081]: Prevent error calling **`os.path.realpath()`** on `'NUL'`.
- bpo-38087 [https://bugs.python.org/issue?@action=redirect&bpo=38087]: Fix case sensitivity in test_pathlib and test_ntpath.

- bpo-38088 [https://bugs.python.org/issue?@action=redirect&bpo=38088]: Fixes distutils not finding vcruntime140.dll with only the v142 toolset installed.
- bpo-37283 [https://bugs.python.org/issue?@action=redirect&bpo=37283]: Ensure command-line and unattend.xml setting override previously detected states in Windows installer.
- bpo-38030 [https://bugs.python.org/issue?@action=redirect&bpo=38030]: Fixes **`os.stat()`** failing for block devices on Windows
- bpo-38020 [https://bugs.python.org/issue?@action=redirect&bpo=38020]: Fixes potential crash when calling **`os.readlink()`** (or indirectly through **`realpath()`**) on a file that is not a supported link.
- bpo-37705 [https://bugs.python.org/issue?@action=redirect&bpo=37705]: Improve the implementation of `winerror_to_errno()`.
- bpo-37549 [https://bugs.python.org/issue?@action=redirect&bpo=37549]: **`os.dup()`** no longer fails for standard streams on Windows 7.
- bpo-1311 [https://bugs.python.org/issue?@action=redirect&bpo=1311]: The `nul` file on Windows now returns True from **`exists()`** and a valid result from **`os.stat()`** with `S_IFCHR` set.
- bpo-9949 [https://bugs.python.org/issue?@action=redirect&bpo=9949]: Enable support for following symlinks in **`os.realpath()`**.
- bpo-37834 [https://bugs.python.org/issue?@action=redirect&bpo=37834]: Treat all name surrogate reparse points on Windows in **`os.lstat()`** and other reparse points as regular files in **`os.stat()`**.
- bpo-36266 [https://bugs.python.org/issue?@action=redirect&bpo=36266]: Add the module name in the formatted error message when DLL load fail happens during module import in `_PyImport_FindSharedFuncptrWindows()`. Patch by Srinivas Nyayapati.

- bpo-25172 [https://bugs.python.org/issue?@action=redirect&bpo=25172]: Trying to import the **`crypt`** module on Windows will result in an **`ImportError`** with a message explaining that the module isn't supported on Windows. On other platforms, if the underlying `_crypt` module is not available, the ImportError will include a message explaining the problem.
- bpo-37778 [https://bugs.python.org/issue?@action=redirect&bpo=37778]: Fixes the icons used for file associations to the Microsoft Store package.
- bpo-37734 [https://bugs.python.org/issue?@action=redirect&bpo=37734]: Fix use of registry values to launch Python from Microsoft Store app.
- bpo-37702 [https://bugs.python.org/issue?@action=redirect&bpo=37702]: Fix memory leak on Windows in creating an SSLContext object or running `urllib.request.urlopen('https://...')`.
- bpo-37672 [https://bugs.python.org/issue?@action=redirect&bpo=37672]: Switch Windows Store package's pip to use bundled `pip.ini` instead of **`PIP_USER`** variable.
- bpo-10945 [https://bugs.python.org/issue?@action=redirect&bpo=10945]: Officially drop support for creating bdist_wininst installers on non-Windows systems.
- bpo-37445 [https://bugs.python.org/issue?@action=redirect&bpo=37445]: Include the `FORMAT_MESSAGE_IGNORE_INSERTS` flag in `FormatMessageW()` calls.
- bpo-37369 [https://bugs.python.org/issue?@action=redirect&bpo=37369]: Fixes path for **`sys.executable`** when running from the Microsoft Store.
- bpo-37380 [https://bugs.python.org/issue?@action=redirect&bpo=37380]: Don't collect unfinished processes with `subprocess._active` on Windows to cleanup later. Patch by Ruslan Kuprieiev.

- bpo-37351 [https://bugs.python.org/issue?@action=redirect&bpo=37351]: Removes libpython38.a from standard Windows distribution.
- bpo-35360 [https://bugs.python.org/issue?@action=redirect&bpo=35360]: Update Windows builds to use SQLite 3.28.0.
- bpo-37267 [https://bugs.python.org/issue?@action=redirect&bpo=37267]: On Windows, **`os.dup()`** no longer creates an inheritable fd when handling a character file.
- bpo-36779 [https://bugs.python.org/issue?@action=redirect&bpo=36779]: Ensure `time.tzname` is correct on Windows when the active code page is set to CP_UTF7 or CP_UTF8.
- bpo-32587 [https://bugs.python.org/issue?@action=redirect&bpo=32587]: Make **`winreg.REG_MULTI_SZ`** support zero-length strings.
- bpo-28269 [https://bugs.python.org/issue?@action=redirect&bpo=28269]: Replace use of **`strcasecmp()`** for the system function **`_stricmp()`**. Patch by Minmin Gong.
- bpo-36590 [https://bugs.python.org/issue?@action=redirect&bpo=36590]: Add native Bluetooth RFCOMM support to socket module.

macOS

- bpo-38117 [https://bugs.python.org/issue?@action=redirect&bpo=38117]: Updated OpenSSL to 1.1.1d in macOS installer.
- bpo-38089 [https://bugs.python.org/issue?@action=redirect&bpo=38089]: Move Azure Pipelines to latest VM versions and make macOS tests optional
- bpo-18049 [https://bugs.python.org/issue?@action=redirect&bpo=18049]: Increase the default stack size of threads from 5MB to 16MB on macOS, to match the stack size of the main thread. This avoids crashes on deep recursion in threads.

- bpo-34602 [https://bugs.python.org/issue?@action=redirect&bpo=34602]: Avoid test suite failures on macOS by no longer calling resource.setrlimit to increase the process stack size limit at runtime. The runtime change is no longer needed since the interpreter is being built with a larger default stack size.
- bpo-35360 [https://bugs.python.org/issue?@action=redirect&bpo=35360]: Update macOS installer to use SQLite 3.28.0.
- bpo-34631 [https://bugs.python.org/issue?@action=redirect&bpo=34631]: Updated OpenSSL to 1.1.1c in macOS installer.

IDLE

- bpo-26353 [https://bugs.python.org/issue?@action=redirect&bpo=26353]: Stop adding newline when saving an IDLE shell window.
- bpo-4630 [https://bugs.python.org/issue?@action=redirect&bpo=4630]: Add an option to toggle IDLE’s cursor blink for shell, editor, and output windows. See Settings, General, Window Preferences, Cursor Blink. Patch by Zackery Spytz.
- bpo-38598 [https://bugs.python.org/issue?@action=redirect&bpo=38598]: Do not try to compile IDLE shell or output windows
- bpo-36698 [https://bugs.python.org/issue?@action=redirect&bpo=36698]: IDLE no longer fails when write non-encodable characters to stderr. It now escapes them with a backslash, as the regular Python interpreter. Added the `errors` field to the standard streams.
- bpo-35379 [https://bugs.python.org/issue?@action=redirect&bpo=35379]: When exiting IDLE, catch any AttributeError. One happens when EditorWindow.close is

called twice. Printing a traceback, when IDLE is run from a terminal, is useless and annoying.

- bpo-38183 [https://bugs.python.org/issue?@action=redirect&bpo=38183]: To avoid problems, test_idle ignores the user config directory. It no longer tries to create or access .idlerc or any files within. Users must run IDLE to discover problems with saving settings.
- bpo-38077 [https://bugs.python.org/issue?@action=redirect&bpo=38077]: IDLE no longer adds 'argv' to the user namespace when initializing it. This bug only affected 3.7.4 and 3.8.0b2 to 3.8.0b4.
- bpo-38041 [https://bugs.python.org/issue?@action=redirect&bpo=38041]: Shell restart lines now fill the window width, always start with '=', and avoid wrapping unnecessarily. The line will still wrap if the included file name is long relative to the width.
- bpo-35771 [https://bugs.python.org/issue?@action=redirect&bpo=35771]: To avoid occasional spurious test_idle failures on slower machines, increase the `hover_delay` in test_tooltip.
- bpo-37824 [https://bugs.python.org/issue?@action=redirect&bpo=37824]: Properly handle user input warnings in IDLE shell. Cease turning SyntaxWarnings into SyntaxErrors.
- bpo-37929 [https://bugs.python.org/issue?@action=redirect&bpo=37929]: IDLE Settings dialog now closes properly when there is no shell window.
- bpo-37902 [https://bugs.python.org/issue?@action=redirect&bpo=37902]: Add mousewheel scrolling for IDLE module, path, and stack browsers. Patch by George Zhang.
- bpo-37849 [https://bugs.python.org/issue?@action=redirect&bpo=37849]: Fixed completions list appearing too high or low when shown above the current line.

- bpo-36419 [https://bugs.python.org/issue?@action=redirect&bpo=36419]: Refactor IDLE autocomplete and improve testing.
- bpo-37748 [https://bugs.python.org/issue?@action=redirect&bpo=37748]: Reorder the Run menu. Put the most common choice, Run Module, at the top.
- bpo-37692 [https://bugs.python.org/issue?@action=redirect&bpo=37692]: Improve highlight config sample with example shell interaction and better labels for shell elements.
- bpo-37628 [https://bugs.python.org/issue?@action=redirect&bpo=37628]: Settings dialog no longer expands with font size.
- bpo-37627 [https://bugs.python.org/issue?@action=redirect&bpo=37627]: Initialize the Customize Run dialog with the command line arguments most recently entered before. The user can optionally edit before submitting them.
- bpo-33610 [https://bugs.python.org/issue?@action=redirect&bpo=33610]: Fix code context not showing the correct context when first toggled on.
- bpo-37530 [https://bugs.python.org/issue?@action=redirect&bpo=37530]: Optimize code context to reduce unneeded background activity. Font and highlight changes now occur along with text changes instead of after a random delay.
- bpo-27452 [https://bugs.python.org/issue?@action=redirect&bpo=27452]: Cleanup `config.py` by inlining `RemoveFile` and simplifying the handling of `file` in `CreateConfigHandlers`.
- bpo-37325 [https://bugs.python.org/issue?@action=redirect&bpo=37325]: Fix tab focus traversal order for help source and custom run dialogs.
- bpo-37321 [https://bugs.python.org/issue?@action=redirect&bpo=37321]: Both subprocess connection

error messages now refer to the ‘Startup failure’ section of the IDLE doc.

- bpo-17535 [https://bugs.python.org/issue?@action=redirect&bpo=17535]: Add optional line numbers for IDLE editor windows. Windows open without line numbers unless set otherwise in the General tab of the configuration dialog.
- bpo-26806 [https://bugs.python.org/issue?@action=redirect&bpo=26806]: To compensate for stack frames added by IDLE and avoid possible problems with low recursion limits, add 30 to limits in the user code execution process. Subtract 30 when reporting recursion limits to make this addition mostly transparent.
- bpo-37177 [https://bugs.python.org/issue?@action=redirect&bpo=37177]: Properly ‘attach’ search dialogs to their main window so that they behave like other dialogs and do not get hidden behind their main window.
- bpo-37039 [https://bugs.python.org/issue?@action=redirect&bpo=37039]: Adjust “Zoom Height” to individual screens by momentarily maximizing the window on first use with a particular screen. Changing screen settings may invalidate the saved height. While a window is maximized, “Zoom Height” has no effect.
- bpo-35763 [https://bugs.python.org/issue?@action=redirect&bpo=35763]: Make calltip reminder about ‘/’ meaning positional-only less obtrusive by only adding it when there is room on the first line.
- bpo-5680 [https://bugs.python.org/issue?@action=redirect&bpo=5680]: Add ‘Run… Customized’ to the Run menu to run a module with customized settings. Any ‘command line arguments’ entered are added to sys.argv. One can suppress the normal Shell main module restart.
- bpo-36390 [https://bugs.python.org/issue?@action=redirect&bpo=36390]: Gather Format menu functions

into format.py. Combine paragraph.py, rstrip.py, and format methods from editor.py.

Tools/Demos

- bpo-38118 [https://bugs.python.org/issue?@action=redirect&bpo=38118]: Update Valgrind suppression file to ignore a false alarm in `PyUnicode_Decode()` when using GCC builtin strcmp().
- bpo-38347 [https://bugs.python.org/issue?@action=redirect&bpo=38347]: pathfix.py: Assume all files that end on '.py' are Python scripts when working recursively.
- bpo-37803 [https://bugs.python.org/issue?@action=redirect&bpo=37803]: pdb's `--help` and `--version` long options now work.
- bpo-37942 [https://bugs.python.org/issue?@action=redirect&bpo=37942]: Improve ArgumentClinic converter for floats.
- bpo-37704 [https://bugs.python.org/issue?@action=redirect&bpo=37704]: Remove `Tools/scripts/h2py.py`: use cffi to access a C API in Python.
- bpo-37675 [https://bugs.python.org/issue?@action=redirect&bpo=37675]: 2to3 now works when run from a zipped standard library.
- bpo-37034 [https://bugs.python.org/issue?@action=redirect&bpo=37034]: Argument Clinic now uses the argument name on errors with keyword-only argument instead of their position. Patch contributed by Rémi Lapeyre.
- bpo-37064 [https://bugs.python.org/issue?@action=redirect&bpo=37064]: Add option -k to pathscript.py script: preserve shebang flags. Add option -a to pathscript.py script: add flags.

C API

- bpo-37633 [https://bugs.python.org/issue?@action=redirect&bpo=37633]: Re-export some function compatibility wrappers for macros in `pythonrun.h`.
- bpo-38644 [https://bugs.python.org/issue?@action=redirect&bpo=38644]: Provide `Py_EnterRecursiveCall()` and `Py_LeaveRecursiveCall()` as regular functions for the limited API. Previously, there were defined as macros, but these macros didn’t work with the limited API which cannot access `PyThreadState.recursion_depth` field. Remove `_Py_CheckRecursionLimit` from the stable ABI.
- bpo-38650 [https://bugs.python.org/issue?@action=redirect&bpo=38650]: The global variable `PyStructSequence_UnnamedField` is now a constant and refers to a constant string.
- bpo-38540 [https://bugs.python.org/issue?@action=redirect&bpo=38540]: Fixed possible leak in `PyArg_Parse()` and similar functions for format units `"es#"` and `"et#"` when the macro **`PY_SSIZE_T_CLEAN`** is not defined.
- bpo-38395 [https://bugs.python.org/issue?@action=redirect&bpo=38395]: Fix a crash in `weakref.proxy` objects due to incorrect lifetime management when calling some associated methods that may delete the last reference to object being referenced by the proxy. Patch by Pablo Galindo.
- bpo-36389 [https://bugs.python.org/issue?@action=redirect&bpo=36389]: The `_PyObject_CheckConsistency()` function is now also available in release mode. For example, it can be used to debug a crash in the `visit_decref()` function of the GC.
- bpo-38266 [https://bugs.python.org/issue?@action=redirect&bpo=38266]: Revert the removal of PyThreadState_DeleteCurrent() with documentation.

- bpo-38303 [https://bugs.python.org/issue?@action=redirect&bpo=38303]: Update audioop extension module to use the stable ABI (PEP-384). Patch by Tyler Kieft.
- bpo-38234 [https://bugs.python.org/issue?@action=redirect&bpo=38234]: **`Py_SetPath()`** now sets **`sys.executable`** to the program full path (**`Py_GetProgramFullPath()`**) rather than to the program name (**`Py_GetProgramName()`**).
- bpo-38234 [https://bugs.python.org/issue?@action=redirect&bpo=38234]: Python ignored arguments passed to **`Py_SetPath()`**, **`Py_SetPythonHome()`** and **`Py_SetProgramName()`**: fix Python initialization to use specified arguments.
- bpo-38205 [https://bugs.python.org/issue?@action=redirect&bpo=38205]: The **`Py_UNREACHABLE()`** macro now calls **`Py_FatalError()`**.
- bpo-38140 [https://bugs.python.org/issue?@action=redirect&bpo=38140]: Make dict and weakref offsets opaque for C heap types by passing the offsets through PyMemberDef
- bpo-15088 [https://bugs.python.org/issue?@action=redirect&bpo=15088]: The C function `PyGen_NeedsFinalizing` has been removed. It was not documented, tested or used anywhere within CPython after the implementation of **PEP 442** [https://peps.python.org/pep-0442/]. Patch by Joannah Nanjekye. (Patch by Joannah Nanjekye)
- bpo-36763 [https://bugs.python.org/issue?@action=redirect&bpo=36763]: Options added by `PySys_AddXOption()` are now handled the same way than `PyConfig.xoptions` and command line `-X` options.
- bpo-37926 [https://bugs.python.org/issue?@action=redirect&bpo=37926]: Fix a crash in `PySys_SetArgvEx(0, NULL, 0)`.

- bpo-37879 [https://bugs.python.org/issue?@action=redirect&bpo=37879]: Fix subtype_dealloc to suppress the type decref when the base type is a C heap type
- bpo-37645 [https://bugs.python.org/issue?@action=redirect&bpo=37645]: Add **`_PyObject_FunctionStr()`** to get a user-friendly string representation of a function-like object. Patch by Jeroen Demeyer.
- bpo-29548 [https://bugs.python.org/issue?@action=redirect&bpo=29548]: The functions `PyEval_CallObject`, `PyEval_CallFunction`, `PyEval_CallMethod` and `PyEval_CallObjectWithKeywords` are deprecated. Use **`PyObject_Call()`** and its variants instead.
- bpo-37151 [https://bugs.python.org/issue?@action=redirect&bpo=37151]: `PyCFunction_Call` is now a deprecated alias of **`PyObject_Call()`**.
- bpo-37540 [https://bugs.python.org/issue?@action=redirect&bpo=37540]: The vectorcall protocol now requires that the caller passes only strings as keyword names.
- bpo-37207 [https://bugs.python.org/issue?@action=redirect&bpo=37207]: The vectorcall protocol is now enabled for `type` objects: set `tp_vectorcall` to a vectorcall function to be used instead of `tp_new` and `tp_init` when calling the class itself.
- bpo-21120 [https://bugs.python.org/issue?@action=redirect&bpo=21120]: Exclude Python-ast.h, ast.h and asdl.h from the limited API.
- bpo-37483 [https://bugs.python.org/issue?@action=redirect&bpo=37483]: Add new function `_PyObject_CallOneArg` for calling an object with one positional argument.
- bpo-36763 [https://bugs.python.org/issue?@action=redirect&bpo=36763]: Add **`PyConfig_SetWideStringList()`** function.
- bpo-37337 [https://bugs.python.org/issue?@action=redirect&bpo=37337]: Add fast functions for calling

methods: **`_PyObject_VectorcallMethod()`**, **`_PyObject_CallMethodNoArgs()`** and **`_PyObject_CallMethodOneArg()`**.

- bpo-28805 [https://bugs.python.org/issue?@action=redirect&bpo=28805]: The **`METH_FASTCALL`** calling convention has been documented.
- bpo-37221 [https://bugs.python.org/issue?@action=redirect&bpo=37221]: The new function **`PyCode_NewWithPosOnlyArgs()`** allows to create code objects like **`PyCode_New()`**, but with an extra *posonlyargcount* parameter for indicating the number of positonal-only arguments.
- bpo-37215 [https://bugs.python.org/issue?@action=redirect&bpo=37215]: Fix dtrace issue introduce by bpo-36842 [https://bugs.python.org/issue?@action=redirect&bpo=36842]
- bpo-37194 [https://bugs.python.org/issue?@action=redirect&bpo=37194]: Add a new public **`PyObject_CallNoArgs()`** function to the C API: call a callable Python object without any arguments. It is the most efficient way to call a callback without any argument. On x86-64, for example, `PyObject_CallFunctionObjArgs(func, NULL)` allocates 960 bytes on the stack per call, whereas `PyObject_CallNoArgs(func)` only allocates 624 bytes per call.
- bpo-37170 [https://bugs.python.org/issue?@action=redirect&bpo=37170]: Fix the cast on error in **`PyLong_AsUnsignedLongLongMask()`**.
- bpo-35381 [https://bugs.python.org/issue?@action=redirect&bpo=35381]: Convert posixmodule.c statically allocated types `DirEntryType` and `ScandirIteratorType` to heap-allocated types.
- bpo-34331 [https://bugs.python.org/issue?@action=redirect&bpo=34331]: Use singular/plural noun in

error message when instantiating an abstract class with non-overridden abstract method(s).

Python 3.8.0 beta 1

Release date: 2019-06-04

Security

- bpo-35907 [https://bugs.python.org/issue?@action=redirect&bpo=35907]: **CVE 2019-9948** [https://www.cve.org/CVERecord?id=CVE-2019-9948]: Avoid file reading by disallowing `local-file://` and `local_file://` URL schemes in `URLopener().open()` and `URLopener().retrieve()` of **`urllib.request`**.
- bpo-33529 [https://bugs.python.org/issue?@action=redirect&bpo=33529]: Prevent fold function used in email header encoding from entering infinite loop when there are too many non-ASCII characters in a header.
- bpo-33164 [https://bugs.python.org/issue?@action=redirect&bpo=33164]: Updated blake2 implementation which uses secure memset implementation provided by platform.

Core and Builtins

- bpo-35814 [https://bugs.python.org/issue?@action=redirect&bpo=35814]: Allow unpacking in the right hand side of annotated assignments. In particular, `t: Tuple[int, ...] = x, y, *z` is now allowed.
- bpo-37126 [https://bugs.python.org/issue?@action=redirect&bpo=37126]: All structseq objects are now tracked by the garbage collector. Patch by Pablo Galindo.

- bpo-37122 [https://bugs.python.org/issue?@action=redirect&bpo=37122]: Make the *co_argcount* attribute of code objects represent the total number of positional arguments (including positional-only arguments). The value of *co_posonlyargcount* can be used to distinguish which arguments are positional only, and the difference (*co_argcount* - *co_posonlyargcount*) is the number of positional-or-keyword arguments. Patch by Pablo Galindo.
- bpo-20092 [https://bugs.python.org/issue?@action=redirect&bpo=20092]: Constructors of **`int`**, **`float`** and **`complex`** will now use the **`__index__()`** special method, if available and the corresponding method **`__int__()`**, **`__float__()`** or **`__complex__()`** is not available.
- bpo-37087 [https://bugs.python.org/issue?@action=redirect&bpo=37087]: Add native thread ID (TID) support to OpenBSD.
- bpo-26219 [https://bugs.python.org/issue?@action=redirect&bpo=26219]: Implemented per opcode cache mechanism and `LOAD_GLOBAL` instruction use it. `LOAD_GLOBAL` is now about 40% faster. Contributed by Yury Selivanov, and Inada Naoki.
- bpo-37072 [https://bugs.python.org/issue?@action=redirect&bpo=37072]: Fix crash in PyAST_FromNodeObject() when flags is NULL.
- bpo-37029 [https://bugs.python.org/issue?@action=redirect&bpo=37029]: Freeing a great many small objects could take time quadratic in the number of arenas, due to using linear search to keep `obmalloc.c`’s list of usable arenas sorted by order of number of free memory pools. This is accomplished without search now, leaving the worst-case time linear in the number of arenas. For programs where this quite visibly matters (typically with more than 100 thousand small objects alive simultaneously), this can greatly reduce the time needed to release their memory.

- bpo-26423 [https://bugs.python.org/issue?@action=redirect&bpo=26423]: Fix possible overflow in `wrap_lenfunc()` when `sizeof(long) < sizeof(Py_ssize_t)` (e.g., 64-bit Windows).
- bpo-37050 [https://bugs.python.org/issue?@action=redirect&bpo=37050]: Improve the AST for “debug” f-strings, which use ‘=’ to print out the source of the expression being evaluated. Delete expr_text from the FormattedValue node, and instead use a Constant string node (possibly merged with adjacent constant expressions inside the f-string).
- bpo-22385 [https://bugs.python.org/issue?@action=redirect&bpo=22385]: The `bytes.hex`, `bytearray.hex`, and `memoryview.hex` methods as well as the `binascii.hexlify` and `b2a_hex` functions now have the ability to include an optional separator between hex bytes. This functionality was inspired by MicroPython’s hexlify implementation.
- bpo-26836 [https://bugs.python.org/issue?@action=redirect&bpo=26836]: Add **`os.memfd_create()`**.
- bpo-37032 [https://bugs.python.org/issue?@action=redirect&bpo=37032]: Added new `replace()` method to the code type (**`types.CodeType`**).
- bpo-37007 [https://bugs.python.org/issue?@action=redirect&bpo=37007]: Implement **`socket.if_nameindex()`**, **`socket.if_nametoindex()`**, and **`socket.if_indextoname()`** on Windows.
- bpo-36829 [https://bugs.python.org/issue?@action=redirect&bpo=36829]: **`PyErr_WriteUnraisable()`** now creates a traceback object if there is no current traceback. Moreover, call **`PyErr_NormalizeException()`** and **`PyException_SetTraceback()`** to normalize the exception value. Ignore any error.
- bpo-36878 [https://bugs.python.org/issue?@action=redirect&bpo=36878]: Only accept text after `# type: ignore` if the first character is ASCII. This is to disallow things like `# type: ignoreé`.

- bpo-36878 [https://bugs.python.org/issue?@action=redirect&bpo=36878]: Store text appearing after a `# type: ignore` comment in the AST. For example a type ignore like `# type: ignore[E1000]` will have the string `"[E1000]"` stored in its AST node.
- bpo-2180 [https://bugs.python.org/issue?@action=redirect&bpo=2180]: Treat line continuation at EOF as a `SyntaxError` by Anthony Sottile.
- bpo-36907 [https://bugs.python.org/issue?@action=redirect&bpo=36907]: Fix a crash when calling a C function with a keyword dict (`f(**kwargs)`) and changing the dict `kwargs` while that function is running.
- bpo-36946 [https://bugs.python.org/issue?@action=redirect&bpo=36946]: Fix possible signed integer overflow when handling slices.
- bpo-36826 [https://bugs.python.org/issue?@action=redirect&bpo=36826]: Add NamedExpression kind support to ast_unparse.c
- bpo-1875 [https://bugs.python.org/issue?@action=redirect&bpo=1875]: A **`SyntaxError`** is now raised if a code blocks that will be optimized away (e.g. if conditions that are always false) contains syntax errors. Patch by Pablo Galindo.
- bpo-36027 [https://bugs.python.org/issue?@action=redirect&bpo=36027]: Allow computation of modular inverses via three-argument `pow`: the second argument is now permitted to be negative in the case where the first and third arguments are relatively prime.
- bpo-36861 [https://bugs.python.org/issue?@action=redirect&bpo=36861]: Update the Unicode database to version 12.1.0.
- bpo-28866 [https://bugs.python.org/issue?@action=redirect&bpo=28866]: Avoid caching attributes of classes which type defines mro() to avoid a hard cache invalidation problem.
- bpo-36851 [https://bugs.python.org/issue?@action=redirect&bpo=36851]: The `FrameType` stack is now

correctly cleaned up if the execution ends with a return and the stack is not empty.

- bpo-34616 [https://bugs.python.org/issue?@action=redirect&bpo=34616]: The `compile()` builtin functions now support the `ast.PyCF_ALLOW_TOP_LEVEL_AWAIT` flag, which allow to compile sources that contains top-level `await`, `async with` or `async for`. This is useful to evaluate async-code from with an already async functions; for example in a custom REPL.
- bpo-36842 [https://bugs.python.org/issue?@action=redirect&bpo=36842]: Implement PEP 578, adding sys.audit, io.open_code and related APIs.
- bpo-27639 [https://bugs.python.org/issue?@action=redirect&bpo=27639]: Correct return type for UserList slicing operations. Patch by Michael Blahay, Erick Cervantes, and vaultah
- bpo-36737 [https://bugs.python.org/issue?@action=redirect&bpo=36737]: Move PyRuntimeState.warnings into per-interpreter state (via "module state").
- bpo-36793 [https://bugs.python.org/issue?@action=redirect&bpo=36793]: Removed `__str__` implementations from builtin types **`bool`**, **`int`**, **`float`**, **`complex`** and few classes from the standard library. They now inherit `__str__()` from **`object`**.
- bpo-36817 [https://bugs.python.org/issue?@action=redirect&bpo=36817]: Add a `=` feature f-strings for debugging. This can precede `!s`, `!r`, or `!a`. It produces the text of the expression, followed by an equal sign, followed by the repr of the value of the expression. So `f'{3*9+15=}'` would be equal to the string `'3*9+15=42'`. If `=` is specified, the default conversion is set to `!r`, unless a format spec is given, in which case the formatting behavior is unchanged, and __format__ will be used.
- bpo-24048 [https://bugs.python.org/issue?@action=redirect&bpo=24048]: Save the live exception during import.c's `remove_module()`.

- bpo-27987 [https://bugs.python.org/issue?@action=redirect&bpo=27987]: pymalloc returns memory blocks aligned by 16 bytes, instead of 8 bytes, on 64-bit platforms to conform x86-64 ABI. Recent compilers assume this alignment more often. Patch by Inada Naoki.
- bpo-36601 [https://bugs.python.org/issue?@action=redirect&bpo=36601]: A long-since-meaningless check for `getpid() == main_pid` was removed from Python's internal C signal handler.
- bpo-36594 [https://bugs.python.org/issue?@action=redirect&bpo=36594]: Fix incorrect use of `%p` in format strings. Patch by Zackery Spytz.
- bpo-36045 [https://bugs.python.org/issue?@action=redirect&bpo=36045]: `builtins.help()` now prefixes `async` for async functions.
- bpo-36084 [https://bugs.python.org/issue?@action=redirect&bpo=36084]: Add native thread ID (TID) to threading.Thread objects (supported platforms: Windows, FreeBSD, Linux, macOS)
- bpo-36035 [https://bugs.python.org/issue?@action=redirect&bpo=36035]: Added fix for broken symlinks in combination with pathlib
- bpo-35983 [https://bugs.python.org/issue?@action=redirect&bpo=35983]: Added new trashcan macros to deal with a double deallocation that could occur when the `tp_dealloc` of a subclass calls the `tp_dealloc` of a base class and that base class uses the trashcan mechanism. Patch by Jeroen Demeyer.
- bpo-20602 [https://bugs.python.org/issue?@action=redirect&bpo=20602]: Do not clear **`sys.flags`** and **`sys.float_info`** during shutdown. Patch by Zackery Spytz.
- bpo-26826 [https://bugs.python.org/issue?@action=redirect&bpo=26826]: Expose **`copy_file_range()`** as a low level API in the **`os`** module.

- bpo-32388 [https://bugs.python.org/issue?@action=redirect&bpo=32388]: Remove cross-version binary compatibility requirement in tp_flags.
- bpo-31862 [https://bugs.python.org/issue?@action=redirect&bpo=31862]: Port binascii to PEP 489 multiphase initialization. Patch by Marcel Plch.

Library

- bpo-37128 [https://bugs.python.org/issue?@action=redirect&bpo=37128]: Added **`math.perm()`**.

- bpo-37120 [https://bugs.python.org/issue?@action=redirect&bpo=37120]: Add SSLContext.num_tickets to control the number of TLSv1.3 session tickets.

- bpo-12202 [https://bugs.python.org/issue?@action=redirect&bpo=12202]: Fix the error handling in **`msilib.SummaryInformation.GetProperty()`**. Patch by Zackery Spytz.

- bpo-26835 [https://bugs.python.org/issue?@action=redirect&bpo=26835]: The fcntl module now contains file sealing constants for sealing of memfds.

- bpo-29262 [https://bugs.python.org/issue?@action=redirect&bpo=29262]: Add `get_origin()` and `get_args()` introspection helpers to `typing` module.

- bpo-12639 [https://bugs.python.org/issue?@action=redirect&bpo=12639]: **`msilib.Directory.start_component()`** no longer fails if *keyfile* is not `None`.

- bpo-36999 [https://bugs.python.org/issue?@action=redirect&bpo=36999]: Add the `asyncio.Task.get_coro()` method to publicly expose the tasks’s coroutine object.

- bpo-35246 [https://bugs.python.org/issue?@action=redirect&bpo=35246]: Make `asyncio.create_subprocess_exec()` accept path-like arguments.

- bpo-35279 [https://bugs.python.org/issue?@action=redirect&bpo=35279]: Change default *max_workers* of `ThreadPoolExecutor` from `cpu_count() * 5` to `min(32, cpu_count() + 4)`. Previous value was unreasonably large on many cores machines.

- bpo-37076 [https://bugs.python.org/issue?@action=redirect&bpo=37076]: `_thread.start_new_thread()` now logs uncaught exception raised by the function using `sys.unraisablehook()`, rather than `sys.excepthook()`, so the hook gets access to the function which raised the exception.

- bpo-33725 [https://bugs.python.org/issue?@action=redirect&bpo=33725]: On macOS, the `multiprocessing` module now uses *spawn* start method by default.

- bpo-37054 [https://bugs.python.org/issue?@action=redirect&bpo=37054]: Fix destructor **`_pyio.BytesIO`** and **`_pyio.TextIOWrapper`**: initialize their `_buffer` attribute as soon as possible (in the class body), because it’s used by `__del__()` which calls `close()`.

- bpo-37058 [https://bugs.python.org/issue?@action=redirect&bpo=37058]: PEP 544: Add `Protocol` and `@runtime_checkable` to the `typing` module.

- bpo-36933 [https://bugs.python.org/issue?@action=redirect&bpo=36933]: The functions `sys.set_coroutine_wrapper` and `sys.get_coroutine_wrapper` that were deprecated and marked for removal in 3.8 have been removed.

- bpo-37047 [https://bugs.python.org/issue?@action=redirect&bpo=37047]: Handle late binding and attribute access in **`unittest.mock.AsyncMock`** setup for autospeccing. Document newly implemented async methods in **`unittest.mock.MagicMock`**.

- bpo-37049 [https://bugs.python.org/issue?@action=redirect&bpo=37049]: PEP 589: Add `TypedDict` to the `typing` module.

- bpo-37046 [https://bugs.python.org/issue?@action=redirect&bpo=37046]: PEP 586: Add `Literal` to the `typing` module.

- bpo-37045 [https://bugs.python.org/issue?@action=redirect&bpo=37045]: PEP 591: Add `Final` qualifier and `@final` decorator to the `typing` module.

- bpo-37035 [https://bugs.python.org/issue?@action=redirect&bpo=37035]: Don’t log OSError based exceptions if a fatal error has occurred in asyncio transport. Peer can generate almost any OSError, user cannot avoid these exceptions by fixing own code. Errors are still propagated to user code, it’s just logging them is pointless and pollute asyncio logs.

- bpo-37001 [https://bugs.python.org/issue?@action=redirect&bpo=37001]: **`symtable.symtable()`** now accepts the same input types for source code as the built-in **`compile()`** function. Patch by Dino Viehland.

- bpo-37028 [https://bugs.python.org/issue?@action=redirect&bpo=37028]: Implement asyncio REPL

- bpo-37027 [https://bugs.python.org/issue?@action=redirect&bpo=37027]: Return safe to use proxy socket object from transport.get_extra_info(‘socket’)

- bpo-32528 [https://bugs.python.org/issue?@action=redirect&bpo=32528]: Make asyncio.CancelledError a BaseException.

  This will address the common mistake many asyncio users make: an “except Exception” clause breaking Tasks cancellation.

  In addition to this change, we stop inheriting asyncio.TimeoutError and asyncio.InvalidStateError from their concurrent.futures.* counterparts. There’s no point for these exceptions to share the inheritance chain.

- bpo-1230540 [https://bugs.python.org/issue?@action=redirect&bpo=1230540]: Add a new `threading.excepthook()` function which handles uncaught `threading.Thread.run()` exception. It can be overridden to control how uncaught `threading.Thread.run()` exceptions are handled.

- bpo-36996 [https://bugs.python.org/issue?@action=redirect&bpo=36996]: Handle `unittest.mock.patch()` used as a decorator on async functions.

- bpo-37008 [https://bugs.python.org/issue?@action=redirect&bpo=37008]: Add support for calling `next()` with the mock resulting from `unittest.mock.mock_open()`

- bpo-27737 [https://bugs.python.org/issue?@action=redirect&bpo=27737]: Allow whitespace only header encoding in `email.header` - by Batuhan Taskaya

- bpo-36969 [https://bugs.python.org/issue?@action=redirect&bpo=36969]: PDB command `args` now display positional only arguments. Patch contributed by Rémi Lapeyre.

- bpo-36969 [https://bugs.python.org/issue?@action=redirect&bpo=36969]: PDB command `args` now display keyword only arguments. Patch contributed by Rémi Lapeyre.

- bpo-36983 [https://bugs.python.org/issue?@action=redirect&bpo=36983]: Add missing names to `typing.__all__`: `ChainMap`, `ForwardRef`, `OrderedDict` - by Anthony Sottile.

- bpo-36972 [https://bugs.python.org/issue?@action=redirect&bpo=36972]: Add SupportsIndex protocol to the typing module to allow type checking to detect classes that can be passed to `hex()`, `oct()` and `bin()`.

- bpo-32972 [https://bugs.python.org/issue?@action=redirect&bpo=32972]: Implement `unittest.IsolatedAsyncioTestCase` to help testing asyncio-based code.

- bpo-36952 [https://bugs.python.org/issue?@action=redirect&bpo=36952]: `fileinput.input()` and `fileinput.FileInput` **bufsize** argument has been removed (was deprecated and ignored since Python 3.6), and as a result the **mode** and **openhook** arguments have been made keyword-only.

- bpo-36952 [https://bugs.python.org/issue?@action=redirect&bpo=36952]: Starting with Python 3.3, importing ABCs from `collections` is deprecated, and import should be done from `collections.abc`. Still being able to import from `collections` was marked for removal in 3.8, but has been delayed to 3.9; documentation and `DeprecationWarning` clarified.

- bpo-36949 [https://bugs.python.org/issue?@action=redirect&bpo=36949]: Implement __repr__ for

WeakSet objects.

- bpo-36948 [https://bugs.python.org/issue?@action=redirect&bpo=36948]: Fix **`NameError`** in **`urllib.request.URLopener.retrieve()`**. Patch by Karthikeyan Singaravelan.
- bpo-33524 [https://bugs.python.org/issue?@action=redirect&bpo=33524]: Fix the folding of email header when the max_line_length is 0 or None and the header contains non-ascii characters. Contributed by Licht Takeuchi (@Licht-T).
- bpo-24564 [https://bugs.python.org/issue?@action=redirect&bpo=24564]: **`shutil.copystat()`** now ignores **`errno.EINVAL`** on **`os.setxattr()`** which may occur when copying files on filesystems without extended attributes support.

  Original patch by Giampaolo Rodola, updated by Ying Wang.
- bpo-36888 [https://bugs.python.org/issue?@action=redirect&bpo=36888]: Python child processes can now access the status of their parent process using multiprocessing.process.parent_process
- bpo-36921 [https://bugs.python.org/issue?@action=redirect&bpo=36921]: Deprecate `@coroutine` for sake of `async def`.
- bpo-25652 [https://bugs.python.org/issue?@action=redirect&bpo=25652]: Fix bug in `__rmod__` of `UserString` - by Batuhan Taskaya.
- bpo-36916 [https://bugs.python.org/issue?@action=redirect&bpo=36916]: Remove a message about an

unhandled exception in a task when writer.write() is used without await and writer.drain() fails with an exception.

- bpo-36889 [https://bugs.python.org/issue?@action=redirect&bpo=36889]: Introduce `asyncio.Stream` class that merges `asyncio.StreamReader` and `asyncio.StreamWriter` functionality. `asyncio.Stream` can work in readonly, writeonly and readwrite modes. Provide `asyncio.connect()`, `asyncio.connect_unix()`, `asyncio.connect_read_pipe()` and `asyncio.connect_write_pipe()` factories to open `asyncio.Stream` connections. Provide `asyncio.StreamServer` and `UnixStreamServer` to serve servers with asyncio.Stream API. Modify `asyncio.create_subprocess_shell()` and `asyncio.create_subprocess_exec()` to use `asyncio.Stream` instead of deprecated `StreamReader` and `StreamWriter`. Deprecate `asyncio.StreamReader` and `asyncio.StreamWriter`. Deprecate usage of private classes, e.g. `asyncio.FlowControlMixing` and `asyncio.StreamReaderProtocol` outside of asyncio package.

- bpo-36845 [https://bugs.python.org/issue?@action=redirect&bpo=36845]: Added validation of integer prefixes to the construction of IP networks and interfaces in the ipaddress module.

- bpo-23378 [https://bugs.python.org/issue?@action=redirect&bpo=23378]: Add an extend action to argparser.

- bpo-36867 [https://bugs.python.org/issue?@action=redirect&bpo=36867]: Fix a bug making a SharedMemoryManager instance and its parent process use two separate resource_tracker processes.

- bpo-23896 [https://bugs.python.org/issue?@action=redirect&bpo=23896]: Adds a grammar to lib2to3.pygram that contains exec as a function not as statement.

- bpo-36895 [https://bugs.python.org/issue?@action=redirect&bpo=36895]: The function `time.clock()` was deprecated in 3.3 in favor of `time.perf_counter()` and marked for removal in 3.8, it has removed.

- bpo-35545 [https://bugs.python.org/issue?@action=redirect&bpo=35545]: Fix asyncio discarding IPv6 scopes when ensuring hostname resolutions internally

- bpo-36887 [https://bugs.python.org/issue?@action=redirect&bpo=36887]: Add new function **`math.isqrt()`** to compute integer square roots.

- bpo-34632 [https://bugs.python.org/issue?@action=redirect&bpo=34632]: Introduce the `importlib.metadata` module with (provisional) support for reading metadata from third-party packages.

- bpo-36878 [https://bugs.python.org/issue?@action=redirect&bpo=36878]: When using `type_comments=True` in `ast.parse`, treat `# type: ignore` followed by a non-alphanumeric character and then arbitrary text as a type ignore, instead of requiring nothing but whitespace or another comment. This is to permit formations such as `# type: ignore[E1000]`.

- bpo-36778 [https://bugs.python.org/issue?@action=redirect&bpo=36778]: `cp65001` encoding (Windows code page 65001) becomes an alias to `utf_8` encoding.

- bpo-36867 [https://bugs.python.org/issue?@action=redirect&bpo=36867]: The multiprocessing.resource_tracker replaces the multiprocessing.semaphore_tracker module. Other than semaphores, resource_tracker also tracks shared_memory segments.

- bpo-30262 [https://bugs.python.org/issue?@action=redirect&bpo=30262]: The `Cache` and `Statement` objects of the **`sqlite3`** module are not exposed to the user. Patch by Aviv Palivoda.
- bpo-24538 [https://bugs.python.org/issue?@action=redirect&bpo=24538]: In `shutil.copystat()`, first copy extended file attributes and then file permissions, since extended attributes can only be set on the destination while it is still writeable.
- bpo-36829 [https://bugs.python.org/issue?@action=redirect&bpo=36829]: Add new **`sys.unraisablehook()`** function which can be overridden to control how “unraisable exceptions” are handled. It is called when an exception has occurred but there is no way for Python to handle it. For example, when a destructor raises an exception or during garbage collection (**`gc.collect()`**).
- bpo-36832 [https://bugs.python.org/issue?@action=redirect&bpo=36832]: Introducing `zipfile.Path`, a pathlib-compatible wrapper for traversing zip files.
- bpo-36814 [https://bugs.python.org/issue?@action=redirect&bpo=36814]: Fix an issue where os.posix_spawnp() would incorrectly raise a TypeError when file_actions is None.
- bpo-33110 [https://bugs.python.org/issue?@action=redirect&bpo=33110]: Handle exceptions raised by functions added by concurrent.futures

add_done_callback correctly when the Future has already completed.

- bpo-26903 [https://bugs.python.org/issue?@action=redirect&bpo=26903]: Limit `max_workers` in `ProcessPoolExecutor` to 61 to work around a WaitForMultipleObjects limitation.

- bpo-36813 [https://bugs.python.org/issue?@action=redirect&bpo=36813]: Fix **`QueueListener`** to call `queue.task_done()` upon stopping. Patch by Bar Harel.

- bpo-36806 [https://bugs.python.org/issue?@action=redirect&bpo=36806]: Forbid creation of asyncio stream objects like StreamReader, StreamWriter, Process, and their protocols outside of asyncio package.

- bpo-36802 [https://bugs.python.org/issue?@action=redirect&bpo=36802]: Provide both sync and async calls for StreamWriter.write() and StreamWriter.close()

- bpo-36801 [https://bugs.python.org/issue?@action=redirect&bpo=36801]: Properly handle SSL connection closing in asyncio StreamWriter.drain() call.

- bpo-36785 [https://bugs.python.org/issue?@action=redirect&bpo=36785]: Implement PEP 574 (pickle protocol 5 with out-of-band buffers).

- bpo-36772 [https://bugs.python.org/issue?@action=redirect&bpo=36772]: functools.lru_cache() can now be used as a straight decorator in addition to its existing usage as a function that returns a decorator.

- bpo-6584 [https://bugs.python.org/issue?@action=redirect&bpo=6584]: Add a **`BadGzipFile`** exception to the **`gzip`** module.

- bpo-36748 [https://bugs.python.org/issue?@action=redirect&bpo=36748]: Optimized write buffering in C implementation of `TextIOWrapper`. Writing ASCII string to `TextIOWrapper` with ascii, latin1, or utf-8 encoding is about 20% faster. Patch by Inada Naoki.

- bpo-8138 [https://bugs.python.org/issue?@action=redirect&bpo=8138]: Don't mark `wsgiref.simple_server.SimpleServer` as multi-threaded since `wsgiref.simple_server.WSGIServer` is single-threaded.

- bpo-22640 [https://bugs.python.org/issue?@action=redirect&bpo=22640]: **`py_compile.compile()`** now supports silent mode. Patch by Joannah Nanjekye

- bpo-29183 [https://bugs.python.org/issue?@action=redirect&bpo=29183]: Fix double exceptions in **`wsgiref.handlers.BaseHandler`** by calling its **`close()`** method only when no exception is raised.

- bpo-36548 [https://bugs.python.org/issue?@action=redirect&bpo=36548]: Improved the repr of regular expression flags.

- bpo-36542 [https://bugs.python.org/issue?@action=redirect&bpo=36542]: The signature of Python functions can now be overridden by specifying the `__text_signature__` attribute.

- bpo-36533 [https://bugs.python.org/issue?@action=redirect&bpo=36533]: Reinitialize logging.Handler locks in forked child processes instead of attempting to acquire them all in the parent before forking only to be released in the child process. The acquire/release pattern was leading to deadlocks in code that has implemented any form of chained logging handlers that depend upon one another as the lock acquisition order cannot be guaranteed.

- bpo-35252 [https://bugs.python.org/issue?@action=redirect&bpo=35252]: Throw a TypeError instead of an AssertionError when using an invalid type annotation with singledispatch.
- bpo-35900 [https://bugs.python.org/issue?@action=redirect&bpo=35900]: Allow reduction methods to return a 6-item tuple where the 6th item specifies a custom state-setting method that’s called instead of the regular `__setstate__` method.
- bpo-35900 [https://bugs.python.org/issue?@action=redirect&bpo=35900]: enable custom reduction callback registration for functions and classes in _pickle.c, using the new Pickler’s attribute `reducer_override`
- bpo-36368 [https://bugs.python.org/issue?@action=redirect&bpo=36368]: Fix a bug crashing SharedMemoryManager instances in interactive sessions after a ctrl-c (KeyboardInterrupt) was sent
- bpo-31904 [https://bugs.python.org/issue?@action=redirect&bpo=31904]: Fix mmap fail for VxWorks
- bpo-27497 [https://bugs.python.org/issue?@action=redirect&bpo=27497]: **`csv.DictWriter.writeheader()`** now returns the return value of the underlying **`csv.Writer.writerow()`** method. Patch contributed by Ashish Nitin Patil.
- bpo-36239 [https://bugs.python.org/issue?@action=redirect&bpo=36239]: Parsing .mo files now ignores comments starting and ending with #-#-#-#-#.

- bpo-26707 [https://bugs.python.org/issue?@action=redirect&bpo=26707]: Enable plistlib to read and write binary plist files that were created as a KeyedArchive file. Specifically, this allows the plistlib to process 0x80 tokens as UID objects.

- bpo-31904 [https://bugs.python.org/issue?@action=redirect&bpo=31904]: Add posix module support for VxWorks.

- bpo-35125 [https://bugs.python.org/issue?@action=redirect&bpo=35125]: Asyncio: Remove inner callback on outer cancellation in shield

- bpo-35721 [https://bugs.python.org/issue?@action=redirect&bpo=35721]: Fix **`asyncio.SelectorEventLoop.subprocess_exec()`** leaks file descriptors if `Popen` fails and called with `stdin=subprocess.PIPE`. Patch by Niklas Fiekas.

- bpo-31855 [https://bugs.python.org/issue?@action=redirect&bpo=31855]: **`unittest.mock.mock_open()`** results now respects the argument of read([size]). Patch contributed by Rémi Lapeyre.

- bpo-35431 [https://bugs.python.org/issue?@action=redirect&bpo=35431]: Implement **`math.comb()`** that returns binomial coefficient, that computes the number of ways to choose k items from n items without repetition and without order. Patch by Yash Aggarwal and Keller Fuchs.

- bpo-26660 [https://bugs.python.org/issue?@action=redirect&bpo=26660]: Fixed permission errors in **`TemporaryDirectory`** clean up. Previously `TemporaryDirectory.cleanup()` failed when non-writeable or non-searchable files or directories were created inside a temporary directory.

- bpo-34271 [https://bugs.python.org/issue?@action=redirect&bpo=34271]: Add debugging helpers to ssl module. It’s now possible to dump key material and to trace TLS protocol. The default and stdlib contexts also support SSLKEYLOGFILE env var.
- bpo-26467 [https://bugs.python.org/issue?@action=redirect&bpo=26467]: Added AsyncMock to support using unittest to mock asyncio coroutines. Patch by Lisa Roach.
- bpo-33569 [https://bugs.python.org/issue?@action=redirect&bpo=33569]: dataclasses.InitVar: Exposes the type used to create the init var.
- bpo-34424 [https://bugs.python.org/issue?@action=redirect&bpo=34424]: Fix serialization of messages containing encoded strings when the policy.linesep is set to a multi-character string. Patch by Jens Troeger.
- bpo-34303 [https://bugs.python.org/issue?@action=redirect&bpo=34303]: Performance of `functools.reduce()` is slightly improved. Patch by Sergey Fedoseev.
- bpo-33361 [https://bugs.python.org/issue?@action=redirect&bpo=33361]: Fix a bug in `codecs.StreamRecoder` where seeking might leave old data in a buffer and break subsequent read calls. Patch by Ammar Askar.
- bpo-22454 [https://bugs.python.org/issue?@action=redirect&bpo=22454]: The `shlex` module now exposes `shlex.join()`, the inverse of `shlex.split()`. Patch by Bo Bayles.

- bpo-31922 [https://bugs.python.org/issue?@action=redirect&bpo=31922]: **`asyncio.AbstractEventLoop.create_datagram_endpoint()`**: Do not connect UDP socket when broadcast is allowed. This allows to receive replies after a UDP broadcast.

- bpo-24882 [https://bugs.python.org/issue?@action=redirect&bpo=24882]: Change ThreadPoolExecutor to use existing idle threads before spinning up new ones.

- bpo-31961 [https://bugs.python.org/issue?@action=redirect&bpo=31961]: Added support for bytes and path-like objects in **`subprocess.Popen()`** on Windows. The *args* parameter now accepts a path-like object if *shell* is `False` and a sequence containing bytes and path-like objects. The *executable* parameter now accepts a bytes and path-like object. The *cwd* parameter now accepts a bytes object. Based on patch by Anders Lorentsen.

- bpo-33123 [https://bugs.python.org/issue?@action=redirect&bpo=33123]: **`pathlib.Path.unlink`** now accepts a *missing_ok* parameter to avoid a **`FileNotFoundError`** from being raised. Patch by Robert Buchholz.

- bpo-32941 [https://bugs.python.org/issue?@action=redirect&bpo=32941]: Allow **`mmap.mmap`** objects to access the madvise() system call (through **`mmap.mmap.madvise()`**).

- bpo-22102 [https://bugs.python.org/issue?@action=redirect&bpo=22102]: Added support for ZIP files with disks set to 0. Such files are commonly created by builtin tools on Windows when use ZIP64 extension. Patch by Francisco Facioni.

- bpo-32515 [https://bugs.python.org/issue?@action=redirect&bpo=32515]: trace.py can now run modules via python3 -m trace -t –module module_name

- bpo-32299 [https://bugs.python.org/issue?@action=redirect&bpo=32299]: Changed **`unittest.mock.patch.dict()`** to return the patched dictionary when used as context manager. Patch by Vadim Tsander.

- bpo-27141 [https://bugs.python.org/issue?@action=redirect&bpo=27141]: Added a `__copy__()` to `collections.UserList` and `collections.UserDict` in order to correctly implement shallow copying of the objects. Patch by Bar Harel.

- bpo-31829 [https://bugs.python.org/issue?@action=redirect&bpo=31829]: `\r`, `\0` and `\x1a` (end-of-file on Windows) are now escaped in protocol 0 pickles of Unicode strings. This allows to load them without loss from files open in text mode in Python 2.

- bpo-23395 [https://bugs.python.org/issue?@action=redirect&bpo=23395]: `_thread.interrupt_main()` now avoids setting the Python error status if the `SIGINT` signal is ignored or not handled by Python.

Documentation

- bpo-36896 [https://bugs.python.org/issue?@action=redirect&bpo=36896]: Clarify that some types have unstable constructor signature between Python versions.

- bpo-36686 [https://bugs.python.org/issue?@action=redirect&bpo=36686]: Improve documentation of the stdin, stdout, and stderr arguments of the `asyncio.subprocess_exec` function to specify which values are supported. Also mention that decoding as text is not supported.

Add a few tests to verify that the various values passed to the std* arguments actually work.

- bpo-36984 [https://bugs.python.org/issue?@action=redirect&bpo=36984]: Improve version added references in `typing` module - by Anthony Sottile.
- bpo-36868 [https://bugs.python.org/issue?@action=redirect&bpo=36868]: What's new now mentions SSLContext.hostname_checks_common_name instead of SSLContext.host_flags.
- bpo-35924 [https://bugs.python.org/issue?@action=redirect&bpo=35924]: Add a note to the `curses.addstr()` documentation to warn that multiline strings can cause segfaults because of an ncurses bug.
- bpo-36783 [https://bugs.python.org/issue?@action=redirect&bpo=36783]: Added C API Documentation for Time_FromTimeAndFold and PyDateTime_FromDateAndTimeAndFold as per PEP 495. Patch by Edison Abahurire.
- bpo-36797 [https://bugs.python.org/issue?@action=redirect&bpo=36797]: More of the legacy distutils documentation has been either pruned, or else more clearly marked as being retained solely until the setuptools documentation covers it independently.
- bpo-22865 [https://bugs.python.org/issue?@action=redirect&bpo=22865]: Add detail to the documentation on the `pty.spawn` function.

- bpo-35397 [https://bugs.python.org/issue?@action=redirect&bpo=35397]: Remove deprecation and document urllib.parse.unwrap(). Patch contributed by Rémi Lapeyre.

- bpo-32995 [https://bugs.python.org/issue?@action=redirect&bpo=32995]: Added the context variable in glossary.

- bpo-33519 [https://bugs.python.org/issue?@action=redirect&bpo=33519]: Clarify that `copy()` is not part of the `MutableSequence` ABC.

- bpo-33482 [https://bugs.python.org/issue?@action=redirect&bpo=33482]: Make `codecs.StreamRecoder.writelines` take a list of bytes.

- bpo-25735 [https://bugs.python.org/issue?@action=redirect&bpo=25735]: Added documentation for func factorial to indicate that returns integer values

- bpo-20285 [https://bugs.python.org/issue?@action=redirect&bpo=20285]: Expand object.__doc__ (docstring) to make it clearer. Modify pydoc.py so that help(object) lists object methods (for other classes, help omits methods of the object base class.)

Tests

- bpo-37069 [https://bugs.python.org/issue?@action=redirect&bpo=37069]: Modify test_coroutines, test_cprofile, test_generators, test_raise, test_ssl and test_yield_from to use **`test.support.catch_unraisable_exception()`** rather than **`test.support.captured_stderr()`**.

- bpo-37098 [https://bugs.python.org/issue?@action=redirect&bpo=37098]: Fix test_memfd_create on older Linux Kernels.
- bpo-37081 [https://bugs.python.org/issue?@action=redirect&bpo=37081]: Test with OpenSSL 1.1.1c
- bpo-36829 [https://bugs.python.org/issue?@action=redirect&bpo=36829]: Add `test.support.catch_unraisable_exception()`: context manager catching unraisable exception using `sys.unraisablehook()`.
- bpo-36915 [https://bugs.python.org/issue?@action=redirect&bpo=36915]: The main regrtest process now always removes all temporary directories of worker processes even if they crash or if they are killed on KeyboardInterrupt (CTRL+c).
- bpo-36719 [https://bugs.python.org/issue?@action=redirect&bpo=36719]: “python3 -m test -jN …” now continues the execution of next tests when a worker process crash (CHILD_ERROR state). Previously, the test suite stopped immediately. Use –failfast to stop at the first error.
- bpo-36816 [https://bugs.python.org/issue?@action=redirect&bpo=36816]: Update Lib/test/selfsigned_pythontestdotnet.pem to match self-signed.pythontest.net’s new TLS certificate.
- bpo-35925 [https://bugs.python.org/issue?@action=redirect&bpo=35925]: Skip httplib and nntplib networking tests when they would otherwise fail due to a modern OS or distro with a default OpenSSL policy of rejecting connections to servers with weak certificates.
- bpo-36782 [https://bugs.python.org/issue?@action=redirect&bpo=36782]: Add tests for several C API functions in the `datetime` module. Patch by Edison Abahurire.
- bpo-36342 [https://bugs.python.org/issue?@action=redirect&bpo=36342]: Fix test_multiprocessing in

test_venv if platform lacks functioning sem_open.

Build

- bpo-36721 [https://bugs.python.org/issue?@action=redirect&bpo=36721]: To embed Python into an application, a new `--embed` option must be passed to `python3-config --libs --embed` to get `-lpython3.8` (link the application to libpython). To support both 3.8 and older, try `python3-config --libs --embed` first and fallback to `python3-config --libs` (without `--embed`) if the previous command fails.

  Add a pkg-config `python-3.8-embed` module to embed Python into an application: `pkg-config python-3.8-embed --libs` includes `-lpython3.8`. To support both 3.8 and older, try `pkg-config python-X.Y-embed --libs` first and fallback to `pkg-config python-X.Y --libs` (without `--embed`) if the previous command fails (replace `X.Y` with the Python version).

  On the other hand, `pkg-config python3.8 --libs` no longer contains `-lpython3.8`. C extensions must not be linked to libpython (except on Android, case handled by the script); this change is backward incompatible on purpose.

- bpo-36786 [https://bugs.python.org/issue?@action=redirect&bpo=36786]: “make install” now runs compileall in parallel.

Windows

- bpo-36965 [https://bugs.python.org/issue?@action=redirect&bpo=36965]: include of

STATUS_CONTROL_C_EXIT without depending on MSC compiler
- bpo-35926 [https://bugs.python.org/issue?@action=redirect&bpo=35926]: Update to OpenSSL 1.1.1b for Windows.
- bpo-29883 [https://bugs.python.org/issue?@action=redirect&bpo=29883]: Add Windows support for UDP transports for the Proactor Event Loop. Patch by Adam Meily.
- bpo-33407 [https://bugs.python.org/issue?@action=redirect&bpo=33407]: The `Py_DEPRECATED()` macro has been implemented for MSVC.

macOS

- bpo-36231 [https://bugs.python.org/issue?@action=redirect&bpo=36231]: Support building Python on macOS without /usr/include installed. As of macOS 10.14, system header files are only available within an SDK provided by either the Command Line Tools or the Xcode app.

IDLE

- bpo-35610 [https://bugs.python.org/issue?@action=redirect&bpo=35610]: Replace now redundant .context_use_ps1 with .prompt_last_line. This finishes change started in bpo-31858 [https://bugs.python.org/issue?@action=redirect&bpo=31858].
- bpo-37038 [https://bugs.python.org/issue?@action=redirect&bpo=37038]: Make idlelib.run runnable; add test clause.
- bpo-36958 [https://bugs.python.org/issue?@action=redirect&bpo=36958]: Print any argument other than None or int passed to SystemExit or sys.exit().

- bpo-36807 [https://bugs.python.org/issue?@action=redirect&bpo=36807]: When saving a file, call os.fsync() so bits are flushed to e.g. USB drive.
- bpo-32411 [https://bugs.python.org/issue?@action=redirect&bpo=32411]: In browser.py, remove extraneous sorting by line number since dictionary was created in line number order.

Tools/Demos

- bpo-37053 [https://bugs.python.org/issue?@action=redirect&bpo=37053]: Handle strings like u"bar" correctly in Tools/parser/unparse.py. Patch by Chih-Hsuan Yen.

C API

- bpo-36763 [https://bugs.python.org/issue?@action=redirect&bpo=36763]: Implement the **PEP 587** [https://peps.python.org/pep-0587/] "Python Initialization Configuration".
- bpo-36379 [https://bugs.python.org/issue?@action=redirect&bpo=36379]: Fix crashes when attempting to use the *modulo* parameter when `__ipow__` is implemented in C.
- bpo-37107 [https://bugs.python.org/issue?@action=redirect&bpo=37107]: Update **`PyObject_CallMethodObjArgs()`** and `_PyObject_CallMethodIdObjArgs` to use `_PyObject_GetMethod` to avoid creating a bound method object in many cases. Patch by Michael J. Sullivan.
- bpo-36974 [https://bugs.python.org/issue?@action=redirect&bpo=36974]: Implement **PEP 590** [https://peps.python.org/pep-0590/]: Vectorcall: a fast calling protocol for CPython. This is a new protocol to optimize calls of custom callable objects.

- bpo-36763 [https://bugs.python.org/issue?@action=redirect&bpo=36763]: `Py_Main()` now returns the exitcode rather than calling `Py_Exit(exitcode)` when calling `PyErr_Print()` if the current exception type is `SystemExit`.
- bpo-36922 [https://bugs.python.org/issue?@action=redirect&bpo=36922]: Add new type flag `Py_TPFLAGS_METHOD_DESCRIPTOR` for objects behaving like unbound methods. These are objects supporting the optimization given by the `LOAD_METHOD`/`CALL_METHOD` opcodes. See PEP 590.
- bpo-36728 [https://bugs.python.org/issue?@action=redirect&bpo=36728]: The **`PyEval_ReInitThreads()`** function has been removed from the C API. It should not be called explicitly: use **`PyOS_AfterFork_Child()`** instead.

Python 3.8.0 alpha 4

Release date: 2019-05-06

Security

- bpo-36742 [https://bugs.python.org/issue?@action=redirect&bpo=36742]: Fixes mishandling of pre-normalization characters in urlsplit().
- bpo-30458 [https://bugs.python.org/issue?@action=redirect&bpo=30458]: Address **CVE 2019-9740** [https://www.cve.org/CVERecord?id=CVE-2019-9740] by disallowing URL paths with embedded whitespace or control characters through into the underlying http client request. Such potentially malicious header injection URLs now cause an http.client.InvalidURL exception to be raised.
- bpo-35755 [https://bugs.python.org/issue?@action=redirect&bpo=35755]: **`shutil.which()`** now uses

`os.confstr("CS_PATH")` if available and if the **PATH** environment variable is not set. Remove also the current directory from **posixpath.defpath**. On Unix, **shutil.which()** and the **subprocess** module no longer search the executable in the current directory if the **PATH** environment variable is not set.

Core and Builtins

- bpo-36722 [https://bugs.python.org/issue?@action=redirect&bpo=36722]: In debug build, import now also looks for C extensions compiled in release mode and for C extensions compiled in the stable ABI.
- bpo-32849 [https://bugs.python.org/issue?@action=redirect&bpo=32849]: Fix Python Initialization code on FreeBSD to detect properly when stdin file descriptor (fd 0) is invalid.
- bpo-36623 [https://bugs.python.org/issue?@action=redirect&bpo=36623]: Remove parser headers and related function declarations that lack implementations after the removal of pgen.
- bpo-20180 [https://bugs.python.org/issue?@action=redirect&bpo=20180]: `dict.pop()` is now up to 33% faster thanks to Argument Clinic. Patch by Inada Naoki.
- bpo-36611 [https://bugs.python.org/issue?@action=redirect&bpo=36611]: Debug memory allocators: disable serialno field by default from debug hooks on Python memory allocators to reduce the memory footprint by 5%. Enable **tracemalloc** to get the traceback where a memory block has been allocated when a fatal memory error is logged to decide where to put a breakpoint. Compile Python with `PYMEM_DEBUG_SERIALNO` defined to get back the field.
- bpo-36588 [https://bugs.python.org/issue?@action=redirect&bpo=36588]: On AIX, **sys.platform** doesn’t contain the major version anymore. Always return `'aix'`,

instead of `'aix3'` .. `'aix7'`. Since older Python versions include the version number, it is recommended to always use `sys.platform.startswith('aix')`. Contributed by M. Felt.

- bpo-36549 [https://bugs.python.org/issue?@action=redirect&bpo=36549]: Change str.capitalize to use titlecase for the first character instead of uppercase.
- bpo-36540 [https://bugs.python.org/issue?@action=redirect&bpo=36540]: Implement **PEP 570** [https://peps.python.org/pep-0570/] (Python positional-only parameters). Patch by Pablo Galindo.
- bpo-36475 [https://bugs.python.org/issue?@action=redirect&bpo=36475]: **`PyEval_AcquireLock()`** and **`PyEval_AcquireThread()`** now terminate the current thread if called while the interpreter is finalizing, making them consistent with `PyEval_RestoreThread()`, `Py_END_ALLOW_THREADS()`, and `PyGILState_Ensure()`.
- bpo-36504 [https://bugs.python.org/issue?@action=redirect&bpo=36504]: Fix signed integer overflow in _ctypes.c's `PyCArrayType_new()`.
- bpo-20844 [https://bugs.python.org/issue?@action=redirect&bpo=20844]: Fix running script with encoding cookie and LF line ending may fail on Windows.
- bpo-24214 [https://bugs.python.org/issue?@action=redirect&bpo=24214]: Fixed support of the surrogatepass error handler in the UTF-8 incremental decoder.
- bpo-36452 [https://bugs.python.org/issue?@action=redirect&bpo=36452]: Changing `dict` keys during iteration of the dict itself, `keys()`, `values()`, or `items()` will now be detected in certain corner cases where keys are deleted/added so that the number of keys isn't changed. A `RuntimeError` will be raised after `len(dict)` iterations. Contributed by Thomas Perl.

- bpo-36459 [https://bugs.python.org/issue?@action=redirect&bpo=36459]: Fix a possible double `PyMem_FREE()` due to tokenizer.c’s `tok_nextc()`.
- bpo-36433 [https://bugs.python.org/issue?@action=redirect&bpo=36433]: Fixed TypeError message in classmethoddescr_call.
- bpo-36430 [https://bugs.python.org/issue?@action=redirect&bpo=36430]: Fix a possible reference leak in **`itertools.count()`**.
- bpo-36440 [https://bugs.python.org/issue?@action=redirect&bpo=36440]: Include node names in `ParserError` messages, instead of numeric IDs. Patch by A. Skrobov.
- bpo-36143 [https://bugs.python.org/issue?@action=redirect&bpo=36143]: Regenerate **`keyword`** from the Grammar and Tokens file using pgen. Patch by Pablo Galindo.
- bpo-18372 [https://bugs.python.org/issue?@action=redirect&bpo=18372]: Add missing **`PyObject_GC_Track()`** calls in the **`pickle`** module. Patch by Zackery Spytz.

Library

- bpo-35952 [https://bugs.python.org/issue?@action=redirect&bpo=35952]: Fix pythoninfo when the compiler is missing.
- bpo-28238 [https://bugs.python.org/issue?@action=redirect&bpo=28238]: The `.find*()` methods of xml.etree.ElementTree can now search for wildcards like `{*}tag` and `{ns}*` that match a tag in any namespace or all tags in a namespace. Patch by Stefan Behnel.
- bpo-26978 [https://bugs.python.org/issue?@action=redirect&bpo=26978]: `pathlib.path.link_to()` is now implemented. It creates a hard link pointing to a path.
- bpo-1613500 [https://bugs.python.org/issue?@action=redirect&bpo=1613500]: **`fileinput.FileInput`** now uses

the input file mode to correctly set the output file mode (previously it was hardcoded to `'w'`) when `inplace=True` is passed to its constructor.

- bpo-36734 [https://bugs.python.org/issue?@action=redirect&bpo=36734]: Fix compilation of `faulthandler.c` on HP-UX. Initialize `stack_t current_stack` to zero using `memset()`.
- bpo-13611 [https://bugs.python.org/issue?@action=redirect&bpo=13611]: The xml.etree.ElementTree packages gained support for C14N 2.0 serialisation. Patch by Stefan Behnel.
- bpo-36669 [https://bugs.python.org/issue?@action=redirect&bpo=36669]: Add missing matrix multiplication operator support to weakref.proxy.
- bpo-36676 [https://bugs.python.org/issue?@action=redirect&bpo=36676]: The XMLParser() in xml.etree.ElementTree provides namespace prefix context to the parser target if it defines the callback methods “start_ns()” and/or “end_ns()”. Patch by Stefan Behnel.
- bpo-36673 [https://bugs.python.org/issue?@action=redirect&bpo=36673]: The TreeBuilder and XMLPullParser in xml.etree.ElementTree gained support for parsing comments and processing instructions. Patch by Stefan Behnel.
- bpo-36650 [https://bugs.python.org/issue?@action=redirect&bpo=36650]: The C version of functools.lru_cache() was treating calls with an empty `**kwargs` dictionary as being distinct from calls with no keywords at all. This did not result in an incorrect answer, but it did trigger an unexpected cache miss.
- bpo-28552 [https://bugs.python.org/issue?@action=redirect&bpo=28552]: Fix `distutils.sysconfig` if **`sys.executable`** is `None` or an empty string: use **`os.getcwd()`** to initialize `project_base`. Fix also the distutils build

command: don’t use **`sys.executable`** if it is `None` or an empty string.

- bpo-35755 [https://bugs.python.org/issue?@action=redirect&bpo=35755]: **`shutil.which()`** and `distutils.spawn.find_executable` now use `os.confstr("CS_PATH")` if available instead of **`os.defpath`**, if the `PATH` environment variable is not set. Moreover, don’t use `os.confstr("CS_PATH")` nor **`os.defpath`** if the `PATH` environment variable is set to an empty string.
- bpo-25430 [https://bugs.python.org/issue?@action=redirect&bpo=25430]: improve performance of `IPNetwork.__contains__()`
- bpo-30485 [https://bugs.python.org/issue?@action=redirect&bpo=30485]: Path expressions in xml.etree.ElementTree can now avoid explicit namespace prefixes for tags (or the “{namespace}tag” notation) by passing a default namespace with an empty string prefix.
- bpo-36613 [https://bugs.python.org/issue?@action=redirect&bpo=36613]: Fix **`asyncio`** wait() not removing callback if exception
- bpo-36598 [https://bugs.python.org/issue?@action=redirect&bpo=36598]: Fix `isinstance` check for Mock objects with spec when the code is executed under tracing. Patch by Karthikeyan Singaravelan.
- bpo-18748 [https://bugs.python.org/issue?@action=redirect&bpo=18748]: In development mode (**`-X`** `dev`) and in debug build, the **`io.IOBase`** destructor now logs `close()` exceptions. These exceptions are silent by default in release mode.
- bpo-36575 [https://bugs.python.org/issue?@action=redirect&bpo=36575]: The `_lsprof` module now uses internal timer same to `time.perf_counter()` by default. `gettimeofday(2)` was used on Unix. New timer has better resolution on most Unix platforms and timings are no

longer impacted by system clock updates since `perf_counter()` is monotonic. Patch by Inada Naoki.
- bpo-33461 [https://bugs.python.org/issue?@action=redirect&bpo=33461]: `json.loads` now emits `DeprecationWarning` when `encoding` option is specified. Patch by Matthias Bussonnier.
- bpo-36559 [https://bugs.python.org/issue?@action=redirect&bpo=36559]: The random module now prefers the lean internal _sha512 module over hashlib for seed(version=2) to optimize import time.
- bpo-17561 [https://bugs.python.org/issue?@action=redirect&bpo=17561]: Set backlog=None as the default for socket.create_server.
- bpo-34373 [https://bugs.python.org/issue?@action=redirect&bpo=34373]: Fix **`time.mktime()`** error handling on AIX for year before 1970.
- bpo-36232 [https://bugs.python.org/issue?@action=redirect&bpo=36232]: Improve error message when trying to open existing DBM database that actually doesn’t exist. Patch by Marco Rougeth.
- bpo-36546 [https://bugs.python.org/issue?@action=redirect&bpo=36546]: Add statistics.quantiles()
- bpo-36050 [https://bugs.python.org/issue?@action=redirect&bpo=36050]: Optimized `http.client.HTTPResponse.read()` for large response. Patch by Inada Naoki.
- bpo-36522 [https://bugs.python.org/issue?@action=redirect&bpo=36522]: If *debuglevel* is set to >0 in **`http.client`**, print all values for headers with multiple values for the same header name. Patch by Matt Houglum.
- bpo-36492 [https://bugs.python.org/issue?@action=redirect&bpo=36492]: Deprecated passing required arguments like *func* as keyword arguments in functions which should accept arbitrary keyword arguments and pass them to other function. Arbitrary keyword

arguments (even with names “self” and “func”) can now be passed to these functions if the required arguments are passed as positional arguments.

- bpo-27181 [https://bugs.python.org/issue?@action=redirect&bpo=27181]: Add statistics.geometric_mean().
- bpo-30427 [https://bugs.python.org/issue?@action=redirect&bpo=30427]: `os.path.normcase()` relies on `os.fspath()` to check the type of its argument. Redundant checks have been removed from its `posixpath.normcase()` and `ntpath.normcase()` implementations. Patch by Wolfgang Maier.
- bpo-36385 [https://bugs.python.org/issue?@action=redirect&bpo=36385]: Stop rejecting IPv4 octets for being ambiguously octal. Leading zeros are ignored, and no longer are assumed to specify octal octets. Octets are always decimal numbers. Octets must still be no more than three digits, including leading zeroes.
- bpo-36434 [https://bugs.python.org/issue?@action=redirect&bpo=36434]: Errors during writing to a ZIP file no longer prevent to properly close it.
- bpo-36407 [https://bugs.python.org/issue?@action=redirect&bpo=36407]: Fixed wrong indentation writing for CDATA section in xml.dom.minidom. Patch by Vladimir Surjaninov.
- bpo-36326 [https://bugs.python.org/issue?@action=redirect&bpo=36326]: inspect.getdoc() can now find docstrings for member objects when __slots__ is a dictionary.
- bpo-36366 [https://bugs.python.org/issue?@action=redirect&bpo=36366]: Calling `stop()` on an unstarted or stopped **`unittest.mock.patch()`** object will now return `None` instead of raising **`RuntimeError`**, making the method idempotent. Patch by Karthikeyan Singaravelan.
- bpo-36348 [https://bugs.python.org/issue?@action=redirect&bpo=36348]: The **`imap.IMAP4.logout()`** method

no longer ignores silently arbitrary exceptions.

- bpo-31904 [https://bugs.python.org/issue?@action=redirect&bpo=31904]: Add time module support and fix test_time failures for VxWorks.
- bpo-36227 [https://bugs.python.org/issue?@action=redirect&bpo=36227]: Added support for keyword arguments `default_namespace` and `xml_declaration` in functions `ElementTree.tostring()` and `ElementTree.tostringlist()`.
- bpo-36004 [https://bugs.python.org/issue?@action=redirect&bpo=36004]: Added new alternate constructors **`datetime.date.fromisocalendar()`** and **`datetime.datetime.fromisocalendar()`**, which construct date objects from ISO year, week number and weekday; these are the inverse of each class’s `isocalendar` method. Patch by Paul Ganssle.
- bpo-35936 [https://bugs.python.org/issue?@action=redirect&bpo=35936]: **`modulefinder`** no longer depends on the deprecated **`imp`** module, and the initializer for **`modulefinder.ModuleFinder`** now has immutable default arguments. Patch by Brandt Bucher.
- bpo-35376 [https://bugs.python.org/issue?@action=redirect&bpo=35376]: **`modulefinder`** correctly handles modules that have the same name as a bad package. Patch by Brandt Bucher.
- bpo-17396 [https://bugs.python.org/issue?@action=redirect&bpo=17396]: **`modulefinder`** no longer crashes when encountering syntax errors in followed imports. Patch by Brandt Bucher.
- bpo-35934 [https://bugs.python.org/issue?@action=redirect&bpo=35934]: Added **`create_server()`** and **`has_dualstack_ipv6()`** convenience functions to automate the necessary tasks usually involved when creating a server socket, including accepting both IPv4 and IPv6 connections on the same socket. (Contributed by

Giampaolo Rodola in bpo-17561 [https://bugs.python.org/issue?@action=redirect&bpo=17561].)

- bpo-23078 [https://bugs.python.org/issue?@action=redirect&bpo=23078]: Add support for `classmethod()` and `staticmethod()` to `unittest.mock.create_autospec()`. Initial patch by Felipe Ochoa.
- bpo-35416 [https://bugs.python.org/issue?@action=redirect&bpo=35416]: Fix potential resource warnings in distutils. Patch by Mickaël Schoentgen.
- bpo-25451 [https://bugs.python.org/issue?@action=redirect&bpo=25451]: Add transparency methods to **`tkinter.PhotoImage`**. Patch by Zackery Spytz.
- bpo-35082 [https://bugs.python.org/issue?@action=redirect&bpo=35082]: Don’t return deleted attributes when calling dir on a `unittest.mock.Mock`.
- bpo-34547 [https://bugs.python.org/issue?@action=redirect&bpo=34547]: `wsgiref.handlers.BaseHandler` now handles abrupt client connection terminations gracefully. Patch by Petter Strandmark.
- bpo-31658 [https://bugs.python.org/issue?@action=redirect&bpo=31658]: `xml.sax.parse()` now supports path-like. Patch by Mickaël Schoentgen.
- bpo-34139 [https://bugs.python.org/issue?@action=redirect&bpo=34139]: Remove stale unix datagram socket before binding
- bpo-33530 [https://bugs.python.org/issue?@action=redirect&bpo=33530]: Implemented Happy Eyeballs in `asyncio.create_connection()`. Added two new arguments, *happy_eyeballs_delay* and *interleave*, to specify Happy Eyeballs behavior.
- bpo-33291 [https://bugs.python.org/issue?@action=redirect&bpo=33291]: Do not raise AttributeError when calling the inspect functions isgeneratorfunction, iscoroutinefunction, isasyncgenfunction on a method created from an arbitrary callable. Instead, return False.

- bpo-31310 [https://bugs.python.org/issue?@action=redirect&bpo=31310]: Fix the multiprocessing.semaphore_tracker so it is reused by child processes
- bpo-31292 [https://bugs.python.org/issue?@action=redirect&bpo=31292]: Fix `setup.py check --restructuredtext` for files containing `include` directives.

Documentation

- bpo-36625 [https://bugs.python.org/issue?@action=redirect&bpo=36625]: Remove obsolete comments from docstrings in fractions.Fraction
- bpo-30840 [https://bugs.python.org/issue?@action=redirect&bpo=30840]: Document relative imports
- bpo-36523 [https://bugs.python.org/issue?@action=redirect&bpo=36523]: Add docstring for io.IOBase.writelines().
- bpo-36425 [https://bugs.python.org/issue?@action=redirect&bpo=36425]: New documentation translation: Simplified Chinese [https://docs.python.org/zh-cn/].
- bpo-36345 [https://bugs.python.org/issue?@action=redirect&bpo=36345]: Avoid the duplication of code from `Tools/scripts/serve.py` in using the **`literalinclude`** directive for the basic wsgiref-based web server in the documentation of **`wsgiref`**. Contributed by Stéphane Wirtel.
- bpo-36345 [https://bugs.python.org/issue?@action=redirect&bpo=36345]: Using the code of the `Tools/scripts/serve.py` script as an example in the **`wsgiref`** documentation. Contributed by Stéphane Wirtel.
- bpo-36157 [https://bugs.python.org/issue?@action=redirect&bpo=36157]: Added documentation for PyInterpreterState_Main().
- bpo-33043 [https://bugs.python.org/issue?@action=redirect&bpo=33043]: Updates the docs.python.org

page with the addition of a ‘Contributing to Docs’ link at the end of the page (between ‘Reporting Bugs’ and ‘About Documentation’). Updates the ‘Found a Bug’ page with additional links and information in the Documentation Bugs section.

- bpo-35581 [https://bugs.python.org/issue?@action=redirect&bpo=35581]: @typing.type_check_only now allows type stubs to mark functions and classes not available during runtime.
- bpo-33832 [https://bugs.python.org/issue?@action=redirect&bpo=33832]: Add glossary entry for ‘magic method’.
- bpo-32913 [https://bugs.python.org/issue?@action=redirect&bpo=32913]: Added re.Match.groupdict example to regex HOWTO.

Tests

- bpo-36719 [https://bugs.python.org/issue?@action=redirect&bpo=36719]: regrtest now always detects uncollectable objects. Previously, the check was only enabled by `--findleaks`. The check now also works with `-jN/--multiprocess N`. `--findleaks` becomes a deprecated alias to `--fail-env-changed`.
- bpo-36725 [https://bugs.python.org/issue?@action=redirect&bpo=36725]: When using multiprocessing mode (-jN), regrtest now better reports errors if a worker process fails, and it exits immediately on a worker thread failure or when interrupted.
- bpo-36454 [https://bugs.python.org/issue?@action=redirect&bpo=36454]: Change test_time.test_monotonic() to test only the lower bound of elapsed time after a sleep command rather than the upper bound. This prevents unnecessary test failures on slow buildbots. Patch by Victor Stinner.

- bpo-32424 [https://bugs.python.org/issue?@action=redirect&bpo=32424]: Improve test coverage for xml.etree.ElementTree. Patch by Gordon P. Hemsley.
- bpo-32424 [https://bugs.python.org/issue?@action=redirect&bpo=32424]: Fix typo in test_cyclic_gc() test for xml.etree.ElementTree. Patch by Gordon P. Hemsley.
- bpo-36635 [https://bugs.python.org/issue?@action=redirect&bpo=36635]: Add a new **`_testinternalcapi`** module to test the internal C API.
- bpo-36629 [https://bugs.python.org/issue?@action=redirect&bpo=36629]: Fix `test_imap4_host_default_value()` of `test_imaplib`: catch also **`errno.ENETUNREACH`** error.
- bpo-36611 [https://bugs.python.org/issue?@action=redirect&bpo=36611]: Fix `test_sys.test_getallocatedblocks()` when **`tracemalloc`** is enabled.
- bpo-36560 [https://bugs.python.org/issue?@action=redirect&bpo=36560]: Fix reference leak hunting in regrtest: compute also deltas (of reference count, allocated memory blocks, file descriptor count) during warmup, to ensure that everything is initialized before starting to hunt reference leaks.
- bpo-36565 [https://bugs.python.org/issue?@action=redirect&bpo=36565]: Fix reference hunting (`python3 -m test -R 3:3`) when Python has no built-in abc module.
- bpo-31904 [https://bugs.python.org/issue?@action=redirect&bpo=31904]: Port test_resource to VxWorks: skip tests cases setting RLIMIT_FSIZE and RLIMIT_CPU.
- bpo-31904 [https://bugs.python.org/issue?@action=redirect&bpo=31904]: Fix test_tabnanny on VxWorks: adjust ENOENT error message.
- bpo-36436 [https://bugs.python.org/issue?@action=redirect&bpo=36436]: Fix `_testcapi.pymem_buffer_overflow()`: handle memory allocation failure.

- bpo-31904 [https://bugs.python.org/issue?@action=redirect&bpo=31904]: Fix test_utf8_mode on VxWorks: Python always use UTF-8 on VxWorks.
- bpo-36341 [https://bugs.python.org/issue?@action=redirect&bpo=36341]: Fix tests that may fail with PermissionError upon calling bind() on AF_UNIX sockets.

Build

- bpo-36747 [https://bugs.python.org/issue?@action=redirect&bpo=36747]: Remove the stale scriptsinstall Makefile target.

- bpo-21536 [https://bugs.python.org/issue?@action=redirect&bpo=21536]: On Unix, C extensions are no longer linked to libpython except on Android and Cygwin.

  It is now possible for a statically linked Python to load a C extension built using a shared library Python.

  When Python is embedded, `libpython` must not be loaded with `RTLD_LOCAL`, but `RTLD_GLOBAL` instead. Previously, using `RTLD_LOCAL`, it was already not possible to load C extensions which were not linked to `libpython`, such as C extensions of the standard library built by the `*shared*` section of `Modules/Setup`.

  distutils, python-config and python-config.py have been modified.

- bpo-36707 [https://bugs.python.org/issue?@action=redirect&bpo=36707]: `./configure --with-pymalloc` no longer adds the `m` flag to SOABI (sys.implementation.cache_tag). Enabling or disabling pymalloc has no impact on the ABI.

- bpo-36635 [https://bugs.python.org/issue?@action=redirect&bpo=36635]: Change `PyAPI_FUNC(type)`, `PyAPI_DATA(type)` and `PyMODINIT_FUNC` macros of `pyport.h` when `Py_BUILD_CORE_MODULE` is defined. The `Py_BUILD_CORE_MODULE` define must be now be used to build a C extension as a dynamic library accessing Python internals: export the `PyInit_xxx()` function in DLL exports on Windows.

- bpo-31904 [https://bugs.python.org/issue?@action=redirect&bpo=31904]: Don’t build the `_crypt` extension on VxWorks.

- bpo-36618 [https://bugs.python.org/issue?@action=redirect&bpo=36618]: Add `-fmax-type-align=8` to CFLAGS when clang compiler is detected. The pymalloc memory allocator aligns memory on 8 bytes. On x86-64, clang expects alignment on 16 bytes by default and so uses MOVAPS instruction which can lead to segmentation fault. Instruct clang that Python is limited to alignment on 8 bytes to use MOVUPS instruction instead: slower but don’t trigger a SIGSEGV if the memory is not aligned on 16 bytes. Sadly, the flag must be added to `CFLAGS` and not just `CFLAGS_NODIST`, since third party C extensions can have the same issue.

- bpo-36605 [https://bugs.python.org/issue?@action=redirect&bpo=36605]: `make tags` and `make TAGS` now also parse `Modules/_io/*.c` and `Modules/_io/*.h`.

- bpo-36465 [https://bugs.python.org/issue?@action=redirect&bpo=36465]: Release builds and debug builds are now ABI compatible: defining the `Py_DEBUG` macro no longer implies the `Py_TRACE_REFS` macro, which introduces the only ABI incompatibility. The `Py_TRACE_REFS` macro, which adds the **`sys.getobjects()`** function and the

`PYTHONDUMPREFS` environment variable, can be set using the new `./configure --with-trace-refs` build option.

- bpo-36577 [https://bugs.python.org/issue?@action=redirect&bpo=36577]: setup.py now correctly reports missing OpenSSL headers and libraries again.

- bpo-36544 [https://bugs.python.org/issue?@action=redirect&bpo=36544]: Fix regression introduced in bpo-36146 [https://bugs.python.org/issue?@action=redirect&bpo=36146] refactoring setup.py

- bpo-36508 [https://bugs.python.org/issue?@action=redirect&bpo=36508]: `python-config --ldflags` no longer includes flags of the `LINKFORSHARED` variable. The `LINKFORSHARED` variable must only be used to build executables.

- bpo-36503 [https://bugs.python.org/issue?@action=redirect&bpo=36503]: Remove references to “aix3” and “aix4”. Patch by M. Felt.

Windows

- bpo-35920 [https://bugs.python.org/issue?@action=redirect&bpo=35920]: Added platform.win32_edition() and platform.win32_is_iot(). Added support for cross-compiling packages for Windows ARM32. Skip tests that are not expected to work on Windows IoT Core ARM32.
- bpo-36649 [https://bugs.python.org/issue?@action=redirect&bpo=36649]: Remove trailing spaces for registry keys when installed via the Store.
- bpo-34144 [https://bugs.python.org/issue?@action=redirect&bpo=34144]: Fixed activate.bat to correctly update codepage when chcp.com returns dots in output. Patch by Lorenz Mende.

- bpo-36509 [https://bugs.python.org/issue?@action=redirect&bpo=36509]: Added preset-iot layout for Windows IoT ARM containers. This layout doesn’t contain UI components like tkinter or IDLE. It also doesn’t contain files to support on-target builds since Windows ARM32 builds must be cross-compiled when using MSVC.
- bpo-35941 [https://bugs.python.org/issue?@action=redirect&bpo=35941]: enum_certificates function of the ssl module now returns certificates from all available certificate stores inside windows in a query instead of returning only certificates from the system wide certificate store. This includes certificates from these certificate stores: local machine, local machine enterprise, local machine group policy, current user, current user group policy, services, users. ssl.enum_crls() function is changed in the same way to return all certificate revocation lists inside the windows certificate revocation list stores.
- bpo-36441 [https://bugs.python.org/issue?@action=redirect&bpo=36441]: Fixes creating a venv when debug binaries are installed.
- bpo-36085 [https://bugs.python.org/issue?@action=redirect&bpo=36085]: Enable better DLL resolution on Windows by using safe DLL search paths and adding **`os.add_dll_directory()`**.
- bpo-36010 [https://bugs.python.org/issue?@action=redirect&bpo=36010]: Add the venv standard library module to the nuget distribution for Windows.
- bpo-29515 [https://bugs.python.org/issue?@action=redirect&bpo=29515]: Add the following socket module constants on Windows: IPPROTO_AH IPPROTO_CBT IPPROTO_DSTOPTS IPPROTO_EGP IPPROTO_ESP IPPROTO_FRAGMENT IPPROTO_GGP IPPROTO_HOPOPTS IPPROTO_ICLFXBM IPPROTO_ICMPV6 IPPROTO_IDP IPPROTO_IGMP IPPROTO_IGP

IPPROTO_IPV4 IPPROTO_IPV6 IPPROTO_L2TP IPPROTO_MAX IPPROTO_ND IPPROTO_NONE IPPROTO_PGM IPPROTO_PIM IPPROTO_PUP IPPROTO_RDP IPPROTO_ROUTING IPPROTO_SCTP IPPROTO_ST

- bpo-35947 [https://bugs.python.org/issue?@action=redirect&bpo=35947]: Added current version of libffi to cpython-source-deps. Change _ctypes to use current version of libffi on Windows.
- bpo-34060 [https://bugs.python.org/issue?@action=redirect&bpo=34060]: Report system load when running test suite on Windows. Patch by Ammar Askar. Based on prior work by Jeremy Kloth.
- bpo-31512 [https://bugs.python.org/issue?@action=redirect&bpo=31512]: With the Windows 10 Creators Update, non-elevated users can now create symlinks as long as the computer has Developer Mode enabled.

macOS

- bpo-34602 [https://bugs.python.org/issue?@action=redirect&bpo=34602]: Avoid failures setting macOS stack resource limit with resource.setrlimit. This reverts an earlier fix for bpo-18075 [https://bugs.python.org/issue?@action=redirect&bpo=18075] which forced a non-default stack size when building the interpreter executable on macOS.

IDLE

- bpo-36429 [https://bugs.python.org/issue?@action=redirect&bpo=36429]: Fix starting IDLE with pyshell. Add idlelib.pyshell alias at top; remove pyshell alias at bottom. Remove obsolete __name__=='__main__' command.

Tools/Demos

- bpo-14546 [https://bugs.python.org/issue?@action=redirect&bpo=14546]: Fix the argument handling in Tools/scripts/lll.py.

C API

- bpo-36763 [https://bugs.python.org/issue?@action=redirect&bpo=36763]: Fix memory leak in **`Py_SetStandardStreamEncoding()`**: release memory if the function is called twice.
- bpo-36641 [https://bugs.python.org/issue?@action=redirect&bpo=36641]: PyDoc_VAR(name) and PyDoc_STRVAR(name, str) now create `static const char name[]` instead of `static char name[]`. Patch by Inada Naoki.
- bpo-36389 [https://bugs.python.org/issue?@action=redirect&bpo=36389]: Change the value of `CLEANBYTE`, `DEADDYTE` and `FORBIDDENBYTE` internal constants used by debug hooks on Python memory allocators (**`PyMem_SetupDebugHooks()`** function). Byte patterns `0xCB`, `0xDB` and `0xFB` have been replaced with `0xCD`, `0xDD` and `0xFD` to use the same values than Windows CRT debug `malloc()` and `free()`.
- bpo-36443 [https://bugs.python.org/issue?@action=redirect&bpo=36443]: Since Python 3.7.0, calling **`Py_DecodeLocale()`** before **`Py_Initialize()`** produces mojibake if the `LC_CTYPE` locale is coerced and/or if the UTF-8 Mode is enabled by the user configuration. The LC_CTYPE coercion and UTF-8 Mode are now disabled by default to fix the mojibake issue. They must now be enabled explicitly (opt-in) using the new **`_Py_PreInitialize()`** API with `_PyPreConfig`.

- bpo-36025 [https://bugs.python.org/issue?@action=redirect&bpo=36025]: Fixed an accidental change to the datetime C API where the arguments to the `PyDate_FromTimestamp()` function were incorrectly interpreted as a single timestamp rather than an arguments tuple, which causes existing code to start raising `TypeError`. The backwards-incompatible change was only present in alpha releases of Python 3.8. Patch by Paul Ganssle.
- bpo-35810 [https://bugs.python.org/issue?@action=redirect&bpo=35810]: Modify `PyObject_Init` to correctly increase the refcount of heap-allocated Type objects. Also fix the refcounts of the heap-allocated types that were either doing this manually or not decreasing the type's refcount in tp_dealloc

Python 3.8.0 alpha 3

Release date: 2019-03-25

Security

- bpo-36216 [https://bugs.python.org/issue?@action=redirect&bpo=36216]: Changes urlsplit() to raise ValueError when the URL contains characters that decompose under IDNA encoding (NFKC-normalization) into characters that affect how the URL is parsed.
- bpo-35121 [https://bugs.python.org/issue?@action=redirect&bpo=35121]: Don't send cookies of domain A without Domain attribute to domain B when domain A is a suffix match of domain B while using a cookiejar with `http.cookiejar.DefaultCookiePolicy` policy. Patch by Karthikeyan Singaravelan.

Core and Builtins

- bpo-36421 [https://bugs.python.org/issue?@action=redirect&bpo=36421]: Fix a possible double decref in _ctypes.c’s `PyCArrayType_new()`.
- bpo-36412 [https://bugs.python.org/issue?@action=redirect&bpo=36412]: Fix a possible crash when creating a new dictionary.
- bpo-36398 [https://bugs.python.org/issue?@action=redirect&bpo=36398]: Fix a possible crash in `structseq_repr()`.
- bpo-36256 [https://bugs.python.org/issue?@action=redirect&bpo=36256]: Fix bug in parsermodule when parsing a state in a DFA that has two or more arcs with labels of the same type. Patch by Pablo Galindo.
- bpo-36365 [https://bugs.python.org/issue?@action=redirect&bpo=36365]: repr(structseq) is no longer limited to 512 bytes.
- bpo-36374 [https://bugs.python.org/issue?@action=redirect&bpo=36374]: Fix a possible null pointer dereference in `merge_consts_recursive()`. Patch by Zackery Spytz.
- bpo-36236 [https://bugs.python.org/issue?@action=redirect&bpo=36236]: At Python initialization, the current directory is no longer prepended to **`sys.path`** if it has been removed.
- bpo-36352 [https://bugs.python.org/issue?@action=redirect&bpo=36352]: Python initialization now fails with an error, rather than silently truncating paths, if a path is too long.
- bpo-36301 [https://bugs.python.org/issue?@action=redirect&bpo=36301]: Python initialization now fails if decoding `pybuilddir.txt` configuration file fails at startup.
- bpo-36333 [https://bugs.python.org/issue?@action=redirect&bpo=36333]: Fix leak in _PyRuntimeState_Fini. Contributed by Stéphane Wirtel.
- bpo-36332 [https://bugs.python.org/issue?@action=redirect&bpo=36332]: The builtin **`compile()`** can now

handle AST objects that contain assignment expressions. Patch by Pablo Galindo.

- bpo-36282 [https://bugs.python.org/issue?@action=redirect&bpo=36282]: Improved error message for too much positional arguments in some builtin functions.
- bpo-30040 [https://bugs.python.org/issue?@action=redirect&bpo=30040]: New empty dict uses fewer memory for now. It used more memory than empty dict created by `dict.clear()`. And empty dict creation and deletion is about 2x faster. Patch by Inada Naoki.
- bpo-36262 [https://bugs.python.org/issue?@action=redirect&bpo=36262]: Fix an unlikely memory leak on conversion from string to float in the function `_Py_dg_strtod()` used by `float(str)`, `complex(str)`, **`pickle.load()`**, **`marshal.load()`**, etc.
- bpo-36252 [https://bugs.python.org/issue?@action=redirect&bpo=36252]: Update Unicode databases to version 12.0.0.
- bpo-36218 [https://bugs.python.org/issue?@action=redirect&bpo=36218]: Fix a segfault occurring when sorting a list of heterogeneous values. Patch contributed by Rémi Lapeyre and Elliot Gorokhovsky.
- bpo-36188 [https://bugs.python.org/issue?@action=redirect&bpo=36188]: Cleaned up left-over vestiges of Python 2 unbound method handling in method objects and documentation. Patch by Martijn Pieters
- bpo-36124 [https://bugs.python.org/issue?@action=redirect&bpo=36124]: Add a new interpreter-specific dict and expose it in the C-API via PyInterpreterState_GetDict(). This parallels PyThreadState_GetDict(). However, extension modules should continue using PyModule_GetState() for their own internal per-interpreter state.
- bpo-35975 [https://bugs.python.org/issue?@action=redirect&bpo=35975]: Add a `feature_version` flag to `ast.parse()` (documented) and `compile()` (hidden) that

allows tweaking the parser to support older versions of the grammar. In particular, if `feature_version` is 5 or 6, the hacks for the `async` and `await` keyword from PEP 492 are reinstated. (For 7 or higher, these are unconditionally treated as keywords, but they are still special tokens rather than `NAME` tokens that the parser driver recognizes.)

- bpo-31904 [https://bugs.python.org/issue?@action=redirect&bpo=31904]: Use UTF-8 as the system encoding on VxWorks.
- bpo-36048 [https://bugs.python.org/issue?@action=redirect&bpo=36048]: The **`__index__()`** special method will be used instead of **`__int__()`** for implicit conversion of Python numbers to C integers. Using the `__int__()` method in implicit conversions has been deprecated.
- bpo-35808 [https://bugs.python.org/issue?@action=redirect&bpo=35808]: Retire pgen and use a modified version of pgen2 to generate the parser. Patch by Pablo Galindo.

Library

- bpo-36401 [https://bugs.python.org/issue?@action=redirect&bpo=36401]: The class documentation created by pydoc now has a separate section for readonly properties.
- bpo-36320 [https://bugs.python.org/issue?@action=redirect&bpo=36320]: The typing.NamedTuple() class has deprecated the _field_types attribute in favor of the __annotations__ attribute which carried the same information. Also, both attributes were converted from OrderedDict to a regular dict.
- bpo-34745 [https://bugs.python.org/issue?@action=redirect&bpo=34745]: Fix **`asyncio`** ssl memory issues caused by circular references

- bpo-36324 [https://bugs.python.org/issue?@action=redirect&bpo=36324]: Add method to statistics.NormalDist for computing the inverse cumulative normal distribution.
- bpo-36321 [https://bugs.python.org/issue?@action=redirect&bpo=36321]: collections.namedtuple() misspelled the name of an attribute. To be consistent with typing.NamedTuple, the attribute name should have been “_field_defaults” instead of “_fields_defaults”. For backwards compatibility, both spellings are now created. The misspelled version may be removed in the future.
- bpo-36297 [https://bugs.python.org/issue?@action=redirect&bpo=36297]: “unicode_internal” codec is removed. It was deprecated since Python 3.3. Patch by Inada Naoki.
- bpo-36298 [https://bugs.python.org/issue?@action=redirect&bpo=36298]: Raise ModuleNotFoundError in pyclbr when a module can’t be found. Thanks to ‘mental’ for the bug report.
- bpo-36268 [https://bugs.python.org/issue?@action=redirect&bpo=36268]: Switch the default format used for writing tars with **`tarfile`** to the modern POSIX.1-2001 pax standard, from the vendor-specific GNU. Contributed by C.A.M. Gerlach.
- bpo-36285 [https://bugs.python.org/issue?@action=redirect&bpo=36285]: Fix integer overflows in the array module. Patch by Stephan Hohe.
- bpo-31904 [https://bugs.python.org/issue?@action=redirect&bpo=31904]: Add _signal module support for VxWorks.
- bpo-36272 [https://bugs.python.org/issue?@action=redirect&bpo=36272]: **`logging`** does not silently ignore RecursionError anymore. Patch contributed by Rémi Lapeyre.

- bpo-36280 [https://bugs.python.org/issue?@action=redirect&bpo=36280]: Add a kind field to ast.Constant. It is ‘u’ if the literal has a ‘u’ prefix (i.e. a Python 2 style unicode literal), else None.
- bpo-35931 [https://bugs.python.org/issue?@action=redirect&bpo=35931]: The **`pdb`** `debug` command now gracefully handles all exceptions.
- bpo-36251 [https://bugs.python.org/issue?@action=redirect&bpo=36251]: Fix format strings used for stderrprinter and re.Match reprs. Patch by Stephan Hohe.
- bpo-36235 [https://bugs.python.org/issue?@action=redirect&bpo=36235]: Fix `CFLAGS` in `customize_compiler()` of `distutils.sysconfig`: when the `CFLAGS` environment variable is defined, don’t override `CFLAGS` variable with the `OPT` variable anymore. Initial patch written by David Malcolm.
- bpo-35807 [https://bugs.python.org/issue?@action=redirect&bpo=35807]: Update ensurepip to install pip 19.0.3 and setuptools 40.8.0.
- bpo-36139 [https://bugs.python.org/issue?@action=redirect&bpo=36139]: Release GIL when closing **`mmap`** objects.
- bpo-36179 [https://bugs.python.org/issue?@action=redirect&bpo=36179]: Fix two unlikely reference leaks in _hashopenssl. The leaks only occur in out-of-memory cases.
- bpo-36169 [https://bugs.python.org/issue?@action=redirect&bpo=36169]: Add overlap() method to statistics.NormalDist. Computes the overlapping coefficient for two normal distributions.
- bpo-36103 [https://bugs.python.org/issue?@action=redirect&bpo=36103]: Default buffer size used by `shutil.copyfileobj()` is changed from 16 KiB to 64 KiB on non-Windows platform to reduce system call overhead. Contributed by Inada Naoki.

- bpo-36130 [https://bugs.python.org/issue?@action=redirect&bpo=36130]: Fix `pdb` with `skip=...` when stepping into a frame without a `__name__` global. Patch by Anthony Sottile.
- bpo-35652 [https://bugs.python.org/issue?@action=redirect&bpo=35652]: shutil.copytree(copy_function=…) erroneously pass DirEntry instead of a path string.
- bpo-35178 [https://bugs.python.org/issue?@action=redirect&bpo=35178]: Ensure custom **`warnings.formatwarning()`** function can receive `line` as positional argument. Based on patch by Tashrif Billah.
- bpo-36106 [https://bugs.python.org/issue?@action=redirect&bpo=36106]: Resolve potential name clash with libm’s sinpi(). Patch by Dmitrii Pasechnik.
- bpo-36091 [https://bugs.python.org/issue?@action=redirect&bpo=36091]: Clean up reference to async generator in Lib/types. Patch by Henry Chen.
- bpo-36043 [https://bugs.python.org/issue?@action=redirect&bpo=36043]: **`FileCookieJar`** supports path-like object. Contributed by Stéphane Wirtel
- bpo-35899 [https://bugs.python.org/issue?@action=redirect&bpo=35899]: Enum has been fixed to correctly handle empty strings and strings with non-Latin characters (ie. ‘α’, ‘א’) without crashing. Original patch contributed by Maxwell. Assisted by Stéphane Wirtel.
- bpo-21269 [https://bugs.python.org/issue?@action=redirect&bpo=21269]: Add `args` and `kwargs` properties to mock call objects. Contributed by Kumar Akshay.
- bpo-30670 [https://bugs.python.org/issue?@action=redirect&bpo=30670]: `pprint.pp` has been added to pretty-print objects with dictionary keys being sorted with their insertion order by default. Parameter *sort_dicts* has been added to `pprint.pprint`, `pprint.pformat` and `pprint.PrettyPrinter`. Contributed by Rémi Lapeyre.

- bpo-35843 [https://bugs.python.org/issue?@action=redirect&bpo=35843]: Implement `__getitem__` for `_NamespacePath`. Patch by Anthony Sottile.
- bpo-35802 [https://bugs.python.org/issue?@action=redirect&bpo=35802]: Clean up code which checked presence of `os.stat` / `os.lstat` / `os.chmod` which are always present. Patch by Anthony Sottile.
- bpo-35715 [https://bugs.python.org/issue?@action=redirect&bpo=35715]: Librates the return value of a ProcessPoolExecutor _process_worker after it’s no longer needed to free memory
- bpo-35493 [https://bugs.python.org/issue?@action=redirect&bpo=35493]: Use `multiprocessing.connection.wait()` instead of polling each 0.2 seconds for worker updates in **`multiprocessing.Pool`**. Patch by Pablo Galindo.
- bpo-35661 [https://bugs.python.org/issue?@action=redirect&bpo=35661]: Store the venv prompt in pyvenv.cfg.
- bpo-35121 [https://bugs.python.org/issue?@action=redirect&bpo=35121]: Don’t set cookie for a request when the request path is a prefix match of the cookie’s path attribute but doesn’t end with “/”. Patch by Karthikeyan Singaravelan.
- bpo-21478 [https://bugs.python.org/issue?@action=redirect&bpo=21478]: Calls to a child function created with `unittest.mock.create_autospec()` should propagate to the parent. Patch by Karthikeyan Singaravelan.
- bpo-35198 [https://bugs.python.org/issue?@action=redirect&bpo=35198]: Fix C++ extension compilation on AIX

Documentation

- bpo-36329 [https://bugs.python.org/issue?@action=redirect&bpo=36329]: Declare the path of the Python

binary for the usage of `Tools/scripts/serve.py` when executing `make -C Doc/ serve`. Contributed by Stéphane Wirtel
- bpo-36138 [https://bugs.python.org/issue?@action=redirect&bpo=36138]: Improve documentation about converting datetime.timedelta to scalars.
- bpo-21314 [https://bugs.python.org/issue?@action=redirect&bpo=21314]: A new entry was added to the Core Language Section of the Programming FAQ, which explaines the usage of slash(/) in the signature of a function. Patch by Lysandros Nikolaou

Tests

- bpo-36234 [https://bugs.python.org/issue?@action=redirect&bpo=36234]: test_posix.PosixUidGidTests: add tests for invalid uid/gid type (str). Initial patch written by David Malcolm.
- bpo-29571 [https://bugs.python.org/issue?@action=redirect&bpo=29571]: Fix `test_re.test_locale_flag()`: use `locale.getpreferredencoding()` rather than `locale.getlocale()` to get the locale encoding. With some locales, `locale.getlocale()` returns the wrong encoding.
- bpo-36123 [https://bugs.python.org/issue?@action=redirect&bpo=36123]: Fix race condition in test_socket.

Build

- bpo-36356 [https://bugs.python.org/issue?@action=redirect&bpo=36356]: Fix leaks that led to build failure when configured with address sanitizer.
- bpo-36146 [https://bugs.python.org/issue?@action=redirect&bpo=36146]: Add `TEST_EXTENSIONS` constant to `setup.py` to allow to not build test extensions like `_testcapi`.

- bpo-36146 [https://bugs.python.org/issue?@action=redirect&bpo=36146]: Fix setup.py on macOS: only add `/usr/include/ffi` to include directories of _ctypes, not for all extensions.
- bpo-31904 [https://bugs.python.org/issue?@action=redirect&bpo=31904]: Enable build system to cross-build for VxWorks RTOS.

Windows

- bpo-36312 [https://bugs.python.org/issue?@action=redirect&bpo=36312]: Fixed decoders for the following code pages: 50220, 50221, 50222, 50225, 50227, 50229, 57002 through 57011, 65000 and 42.
- bpo-36264 [https://bugs.python.org/issue?@action=redirect&bpo=36264]: Don’t honor POSIX `HOME` in `os.path.expanduser` on windows. Patch by Anthony Sottile.
- bpo-24643 [https://bugs.python.org/issue?@action=redirect&bpo=24643]: Fix name collisions due to `#define timezone _timezone` in PC/pyconfig.h.

IDLE

- bpo-36405 [https://bugs.python.org/issue?@action=redirect&bpo=36405]: Use dict unpacking in idlelib.
- bpo-36396 [https://bugs.python.org/issue?@action=redirect&bpo=36396]: Remove fgBg param of idlelib.config.GetHighlight(). This param was only used twice and changed the return type.
- bpo-36176 [https://bugs.python.org/issue?@action=redirect&bpo=36176]: Fix IDLE autocomplete & calltip popup colors. Prevent conflicts with Linux dark themes (and slightly darken calltip background).
- bpo-23205 [https://bugs.python.org/issue?@action=redirect&bpo=23205]: For the grep module, add tests

for findfiles, refactor findfiles to be a module-level function, and refactor findfiles to use os.walk.

- bpo-23216 [https://bugs.python.org/issue?@action=redirect&bpo=23216]: Add docstrings to IDLE search modules.
- bpo-36152 [https://bugs.python.org/issue?@action=redirect&bpo=36152]: Remove colorizer.ColorDelegator.close_when_done and the corresponding argument of .close(). In IDLE, both have always been None or False since 2007.
- bpo-32129 [https://bugs.python.org/issue?@action=redirect&bpo=32129]: Avoid blurry IDLE application icon on macOS with Tk 8.6. Patch by Kevin Walzer.
- bpo-36096 [https://bugs.python.org/issue?@action=redirect&bpo=36096]: Refactor class variables to instance variables in colorizer.
- bpo-30348 [https://bugs.python.org/issue?@action=redirect&bpo=30348]: Increase test coverage of idlelib.autocomplete by 30%. Patch by Louie Lu

Tools/Demos

- bpo-35132 [https://bugs.python.org/issue?@action=redirect&bpo=35132]: Fix py-list and py-bt commands of python-gdb.py on gdb7.
- bpo-32217 [https://bugs.python.org/issue?@action=redirect&bpo=32217]: Fix freeze script on Windows.

C API

- bpo-36381 [https://bugs.python.org/issue?@action=redirect&bpo=36381]: Raise `DeprecationWarning` when ‘#’ formats are used for building or parsing values without `PY_SSIZE_T_CLEAN`.
- bpo-36142 [https://bugs.python.org/issue?@action=redirect&bpo=36142]: The whole coreconfig.h header

is now excluded from Py_LIMITED_API. Move functions definitions into a new internal pycore_coreconfig.h header.

Python 3.8.0 alpha 2

Release date: 2019-02-25

Core and Builtins

- bpo-36052 [https://bugs.python.org/issue?@action=redirect&bpo=36052]: Raise a **`SyntaxError`** when assigning a value to `__debug__` with the Assignment Operator. Contributed by Stéphane Wirtel and Pablo Galindo.
- bpo-36012 [https://bugs.python.org/issue?@action=redirect&bpo=36012]: Doubled the speed of class variable writes. When a non-dunder attribute was updated, there was an unnecessary call to update slots.
- bpo-35942 [https://bugs.python.org/issue?@action=redirect&bpo=35942]: The error message emitted when returning invalid types from `__fspath__` in interfaces that allow passing **`PathLike`** objects has been improved and now it does explain the origin of the error.
- bpo-36016 [https://bugs.python.org/issue?@action=redirect&bpo=36016]: `gc.get_objects` can now receive an optional parameter indicating a generation to get objects from. Patch by Pablo Galindo.
- bpo-1054041 [https://bugs.python.org/issue?@action=redirect&bpo=1054041]: When the main interpreter exits due to an uncaught KeyboardInterrupt, the process now exits in the appropriate manner for its parent process to detect that a SIGINT or ^C terminated the process. This allows shells and batch scripts to understand that the user has asked them to stop.

- bpo-35992 [https://bugs.python.org/issue?@action=redirect&bpo=35992]: Fix `__class_getitem__()` not being called on a class with a custom non-subscriptable metaclass.
- bpo-35993 [https://bugs.python.org/issue?@action=redirect&bpo=35993]: Fix a crash on fork when using subinterpreters. Contributed by Stéphane Wirtel
- bpo-35991 [https://bugs.python.org/issue?@action=redirect&bpo=35991]: Fix a potential double free in Modules/_randommodule.c.
- bpo-35961 [https://bugs.python.org/issue?@action=redirect&bpo=35961]: Fix a crash in slice_richcompare(): use strong references rather than stolen references for the two temporary internal tuples.
- bpo-35911 [https://bugs.python.org/issue?@action=redirect&bpo=35911]: Enable the creation of cell objects by adding a `cell.__new__` method, and expose the type `cell` in `Lib/types.py` under the name CellType. Patch by Pierre Glaser.
- bpo-12822 [https://bugs.python.org/issue?@action=redirect&bpo=12822]: Use monotonic clock for `pthread_cond_timedwait` when `pthread_condattr_setclock` and `CLOCK_MONOTONIC` are available.
- bpo-15248 [https://bugs.python.org/issue?@action=redirect&bpo=15248]: The compiler emits now syntax warnings in the case when a comma is likely missed before tuple or list.
- bpo-35886 [https://bugs.python.org/issue?@action=redirect&bpo=35886]: The implementation of PyInterpreterState has been moved into the internal header files (guarded by Py_BUILD_CORE).
- bpo-31506 [https://bugs.python.org/issue?@action=redirect&bpo=31506]: Clarify the errors reported when `object.__new__` and `object.__init__` receive more than one argument. Contributed by Sanyam Khurana.

- bpo-35724 [https://bugs.python.org/issue?@action=redirect&bpo=35724]: Signal-handling is now guaranteed to happen relative to the main interpreter.
- bpo-33608 [https://bugs.python.org/issue?@action=redirect&bpo=33608]: We added a new internal _Py_AddPendingCall() that operates relative to the provided interpreter. This allows us to use the existing implementation to ask another interpreter to do work that cannot be done in the current interpreter, like decref an object the other interpreter owns. The existing Py_AddPendingCall() only operates relative to the main interpreter.
- bpo-33989 [https://bugs.python.org/issue?@action=redirect&bpo=33989]: Fix a possible crash in **`list.sort()`** when sorting objects with `ob_type->tp_richcompare == NULL`. Patch by Zackery Spytz.

Library

- bpo-35512 [https://bugs.python.org/issue?@action=redirect&bpo=35512]: **`unittest.mock.patch.dict()`** used as a decorator with string target resolves the target during function call instead of during decorator construction. Patch by Karthikeyan Singaravelan.
- bpo-36018 [https://bugs.python.org/issue?@action=redirect&bpo=36018]: Add statistics.NormalDist, a tool for creating and manipulating normal distributions of random variable. Features a composite class that treats the mean and standard deviation of measurement data as single entity.
- bpo-35904 [https://bugs.python.org/issue?@action=redirect&bpo=35904]: Added statistics.fmean() as a faster, floating-point variant of the existing mean() function.
- bpo-35918 [https://bugs.python.org/issue?@action=redirect&bpo=35918]: Removed broken `has_key`

method from multiprocessing.managers.SyncManager.dict. Contributed by Rémi Lapeyre.

- bpo-18283 [https://bugs.python.org/issue?@action=redirect&bpo=18283]: Add support for bytes to **`shutil.which()`**.
- bpo-35960 [https://bugs.python.org/issue?@action=redirect&bpo=35960]: Fix **`dataclasses.field()`** throwing away empty mapping objects passed as metadata.
- bpo-35500 [https://bugs.python.org/issue?@action=redirect&bpo=35500]: Write expected and actual call parameters on separate lines in **`unittest.mock.Mock.assert_called_with()`** assertion errors. Contributed by Susan Su.
- bpo-35931 [https://bugs.python.org/issue?@action=redirect&bpo=35931]: The **`pdb`** `debug` command now gracefully handles syntax errors.
- bpo-24209 [https://bugs.python.org/issue?@action=redirect&bpo=24209]: In http.server script, rely on getaddrinfo to bind to preferred address based on the bind parameter. Now default bind or binding to a name may bind to IPv6 or dual-stack, depending on the environment.
- bpo-35321 [https://bugs.python.org/issue?@action=redirect&bpo=35321]: Set `__spec__.origin` of `_frozen_importlib` to frozen so that it matches the behavior of `_frozen_importlib_external`. Patch by Nina Zakharenko.
- bpo-35378 [https://bugs.python.org/issue?@action=redirect&bpo=35378]: Fix a reference issue inside **`multiprocessing.Pool`** that caused the pool to remain alive if it was deleted without being closed or terminated explicitly. A new strong reference is added to the pool iterators to link the lifetime of the pool to the lifetime of its iterators so the pool does not get destroyed if a pool iterator is still alive.

- bpo-34294 [https://bugs.python.org/issue?@action=redirect&bpo=34294]: re module, fix wrong capturing groups in rare cases. `re.search()`, `re.findall()`, `re.sub()` and other functions that scan through string looking for a match, should reset capturing groups between two match attempts. Patch by Ma Lin.
- bpo-35615 [https://bugs.python.org/issue?@action=redirect&bpo=35615]: `weakref`: Fix a RuntimeError when copying a WeakKeyDictionary or a WeakValueDictionary, due to some keys or values disappearing while iterating.
- bpo-35606 [https://bugs.python.org/issue?@action=redirect&bpo=35606]: Implement `math.prod()` as analogous function to `sum()` that returns the product of a ‘start’ value (default: 1) times an iterable of numbers. Patch by Pablo Galindo.
- bpo-32417 [https://bugs.python.org/issue?@action=redirect&bpo=32417]: Performing arithmetic between `datetime.datetime` subclasses and `datetime.timedelta` now returns an object of the same type as the `datetime.datetime` subclass. As a result, `datetime.datetime.astimezone()` and alternate constructors like `datetime.datetime.now()` and **`datetime.fromtimestamp()`** called with a `tz` argument now *also* retain their subclass.
- bpo-35153 [https://bugs.python.org/issue?@action=redirect&bpo=35153]: Add *headers* optional keyword-only parameter to `xmlrpc.client.ServerProxy`, **`xmlrpc.client.Transport`** and **`xmlrpc.client.SafeTransport`**. Patch by Cédric Krier.
- bpo-34572 [https://bugs.python.org/issue?@action=redirect&bpo=34572]: Fix C implementation of pickle.loads to use importlib’s locking mechanisms, and thereby avoid using partially loaded modules. Patch by Tim Burgess.

Documentation

- bpo-36083 [https://bugs.python.org/issue?@action=redirect&bpo=36083]: Fix formatting of –check-hash-based-pycs options in the manpage Synopsis.
- bpo-36007 [https://bugs.python.org/issue?@action=redirect&bpo=36007]: Bump minimum sphinx version to 1.8. Patch by Anthony Sottile.
- bpo-22062 [https://bugs.python.org/issue?@action=redirect&bpo=22062]: Update documentation and docstrings for pathlib. Original patch by Mike Short.

Tests

- bpo-27313 [https://bugs.python.org/issue?@action=redirect&bpo=27313]: Avoid test_ttk_guionly ComboboxTest failure with macOS Cocoa Tk.

- bpo-36019 [https://bugs.python.org/issue?@action=redirect&bpo=36019]: Add test.support.TEST_HTTP_URL and replace references of http://www.example.com by this new constant. Contributed by Stéphane Wirtel.

- bpo-36037 [https://bugs.python.org/issue?@action=redirect&bpo=36037]: Fix test_ssl for strict OpenSSL configuration like RHEL8 strict crypto policy. Use older TLS version for minimum TLS version of the server SSL context if needed, to test TLS version older than default minimum TLS version.

- bpo-35798 [https://bugs.python.org/issue?@action=redirect&bpo=35798]: Added `test.support.check_syntax_warning()`.

- bpo-35505 [https://bugs.python.org/issue?@action=redirect&bpo=35505]: Make test_imap4_host_default_value independent on whether the local IMAP server is running.

- bpo-35917 [https://bugs.python.org/issue?@action=redirect&bpo=35917]: multiprocessing: provide unit tests for SyncManager and SharedMemoryManager classes + all the shareable types which are supposed to be supported by them. (patch by Giampaolo Rodola)

- bpo-35704 [https://bugs.python.org/issue?@action=redirect&bpo=35704]: Skip `test_shutil.test_unpack_archive_xztar` to prevent a MemoryError on 32-bit AIX when MAXDATA setting is less than 0x20000000.

  Patch by Michael Felt (aixtools)

- bpo-34720 [https://bugs.python.org/issue?@action=redirect&bpo=34720]: Assert m_state != NULL to mimic GC traversal functions that do not correctly handle module creation when the module state has not been created.

Windows

- bpo-35976 [https://bugs.python.org/issue?@action=redirect&bpo=35976]: Added ARM build support to Windows build files in PCBuild.
- bpo-35692 [https://bugs.python.org/issue?@action=redirect&bpo=35692]: `pathlib` no longer raises when checking file and directory existence on drives that are not ready
- bpo-35872 [https://bugs.python.org/issue?@action=redirect&bpo=35872]: Uses the base Python

executable when invoking venv in a virtual environment

- bpo-35873 [https://bugs.python.org/issue?@action=redirect&bpo=35873]: Prevents venv paths being inherited by child processes
- bpo-35299 [https://bugs.python.org/issue?@action=redirect&bpo=35299]: Fix sysconfig detection of the source directory and distutils handling of pyconfig.h during PGO profiling

IDLE

- bpo-24310 [https://bugs.python.org/issue?@action=redirect&bpo=24310]: IDLE – Document settings dialog font tab sample.
- bpo-35833 [https://bugs.python.org/issue?@action=redirect&bpo=35833]: Revise IDLE doc for control codes sent to Shell. Add a code example block.
- bpo-35689 [https://bugs.python.org/issue?@action=redirect&bpo=35689]: Add docstrings and unittests for colorizer.py.

Python 3.8.0 alpha 1

Release date: 2019-02-03

Security

- bpo-35746 [https://bugs.python.org/issue?@action=redirect&bpo=35746]: **CVE 2019-5010** [https://www.cve.org/CVERecord?id=CVE-2019-5010]: Fix a NULL pointer deref in ssl module. The cert parser did not handle CRL distribution points with empty DP or URI correctly. A malicious or buggy certificate can result into segfault. Vulnerability (TALOS-2018-0758) reported by Colin Read and Nicolas Edet of Cisco.

- bpo-34812 [https://bugs.python.org/issue?@action=redirect&bpo=34812]: The **`-I`** command line option (run Python in isolated mode) is now also copied by the **`multiprocessing`** and `distutils` modules when spawning child processes. Previously, only **`-E`** and **`-s`** options (enabled by **`-I`**) were copied.
- bpo-34791 [https://bugs.python.org/issue?@action=redirect&bpo=34791]: The xml.sax and xml.dom.domreg no longer use environment variables to override parser implementations when sys.flags.ignore_environment is set by -E or -I arguments.
- bpo-17239 [https://bugs.python.org/issue?@action=redirect&bpo=17239]: The xml.sax and xml.dom.minidom parsers no longer processes external entities by default. External DTD and ENTITY declarations no longer load files or create network connections.
- bpo-34623 [https://bugs.python.org/issue?@action=redirect&bpo=34623]: **CVE 2018-14647** [https://www.cve.org/CVERecord?id=CVE-2018-14647]: The C accelerated _elementtree module now initializes hash randomization salt from _Py_HashSecret instead of libexpat's default CSPRNG.
- bpo-34405 [https://bugs.python.org/issue?@action=redirect&bpo=34405]: Updated to OpenSSL 1.1.0i for Windows builds.
- bpo-33871 [https://bugs.python.org/issue?@action=redirect&bpo=33871]: Fixed sending the part of the file in **`os.sendfile()`** on macOS. Using the *trailers* argument could cause sending more bytes from the input file than was specified.
- bpo-32533 [https://bugs.python.org/issue?@action=redirect&bpo=32533]: Fixed thread-safety of error handling in _ssl.

- bpo-33136 [https://bugs.python.org/issue?@action=redirect&bpo=33136]: Harden ssl module against LibreSSL **CVE 2018-8970** [https://www.cve.org/CVERecord?id=CVE-2018-8970]. X509_VERIFY_PARAM_set1_host() is called with an explicit namelen. A new test ensures that NULL bytes are not allowed.
- bpo-33001 [https://bugs.python.org/issue?@action=redirect&bpo=33001]: Minimal fix to prevent buffer overrun in os.symlink on Windows
- bpo-32981 [https://bugs.python.org/issue?@action=redirect&bpo=32981]: Regexes in difflib and poplib were vulnerable to catastrophic backtracking. These regexes formed potential DOS vectors (REDOS). They have been refactored. This resolves **CVE 2018-1060** [https://www.cve.org/CVERecord?id=CVE-2018-1060] and **CVE 2018-1061** [https://www.cve.org/CVERecord?id=CVE-2018-1061]. Patch by Jamie Davis.
- bpo-28414 [https://bugs.python.org/issue?@action=redirect&bpo=28414]: The ssl module now allows users to perform their own IDN en/decoding when using SNI.

Core and Builtins

- bpo-35877 [https://bugs.python.org/issue?@action=redirect&bpo=35877]: Make parenthesis optional for named expressions in while statement. Patch by Karthikeyan Singaravelan.

- bpo-35814 [https://bugs.python.org/issue?@action=redirect&bpo=35814]: Allow same right hand side expressions in annotated assignments as in normal ones. In particular, `x: Tuple[int, int] = 1, 2` (without parentheses on the right) is now allowed.

- bpo-35766 [https://bugs.python.org/issue?@action=redirect&bpo=35766]: Add the option to parse PEP 484 type comments in the ast module. (Off by default.) This is merging the key functionality of the third party fork thereof, [typed_ast](https://github.com/python/typed_ast).

- bpo-35713 [https://bugs.python.org/issue?@action=redirect&bpo=35713]: Reorganize Python initialization to get working exceptions and sys.stderr earlier.

- bpo-33416 [https://bugs.python.org/issue?@action=redirect&bpo=33416]: Add end line and end column position information to the Python AST nodes. This is a C-level backwards incompatible change.

- bpo-35720 [https://bugs.python.org/issue?@action=redirect&bpo=35720]: Fixed a minor memory leak in pymain_parse_cmdline_impl function in Modules/main.c

- bpo-35634 [https://bugs.python.org/issue?@action=redirect&bpo=35634]: `func(**kwargs)` will now raise an error when `kwargs` is a mapping containing multiple entries with the same key. An error was already raised when other keyword arguments are passed before `**kwargs` since Python 3.6.

- bpo-35623 [https://bugs.python.org/issue?@action=redirect&bpo=35623]: Fix a crash when sorting very long lists. Patch by Stephan Hohe.

- bpo-35214 [https://bugs.python.org/issue?@action=redirect&bpo=35214]: clang Memory Sanitizer build instrumentation was added to work around false positives from posix, socket, time, test_io, and test_faulthandler.

- bpo-35560 [https://bugs.python.org/issue?@action=redirect&bpo=35560]: Fix an assertion error in `format()` in debug build for floating-point formatting with “n” format, zero padding and small width. Release build is not impacted. Patch by Karthikeyan Singaravelan.

- bpo-35552 [https://bugs.python.org/issue?@action=redirect&bpo=35552]: Format characters `%s` and `%V` in `PyUnicode_FromFormat()` and `%s` in `PyBytes_FromFormat()` no longer read memory past the limit if *precision* is specified.

- bpo-35504 [https://bugs.python.org/issue?@action=redirect&bpo=35504]: Fix segfaults and `SystemError`s when deleting certain attributes. Patch by Zackery Spytz.

- bpo-35504 [https://bugs.python.org/issue?@action=redirect&bpo=35504]: Fixed a SystemError when delete the characters_written attribute of an OSError.

- bpo-35494 [https://bugs.python.org/issue?@action=redirect&bpo=35494]: Improved syntax error messages for unbalanced parentheses in f-string.

- bpo-35444 [https://bugs.python.org/issue?@action=redirect&bpo=35444]: Fixed error handling in pickling methods when fail to look up builtin “getattr”. Sped up pickling iterators.

- bpo-35436 [https://bugs.python.org/issue?@action=redirect&bpo=35436]: Fix various issues with memory allocation error handling. Patch by Zackery Spytz.

- bpo-35423 [https://bugs.python.org/issue?@action=redirect&bpo=35423]: Separate the signal handling trigger in the eval loop from the “pending calls” machinery. There is no semantic change and the difference in performance is insignificant.

- bpo-35357 [https://bugs.python.org/issue?@action=redirect&bpo=35357]: Internal attributes’ names of unittest.mock._Call and unittest.mock.MagicProxy (name, parent & from_kall) are now prefixed with _mock_ in order to prevent clashes with widely used object attributes. Fixed minor typo in test function name.

- bpo-35372 [https://bugs.python.org/issue?@action=redirect&bpo=35372]: Fixed the code page decoder for input longer than 2 GiB containing undecodable bytes.

- bpo-35336 [https://bugs.python.org/issue?@action=redirect&bpo=35336]: Fix PYTHONCOERCECLOCALE=1 environment variable: only coerce the C locale if the LC_CTYPE locale is “C”.

- bpo-31241 [https://bugs.python.org/issue?@action=redirect&bpo=31241]: The *lineno* and *col_offset* attributes of AST nodes for list comprehensions, generator expressions and tuples are now point to the opening parenthesis or square brace. For tuples without parenthesis they point to the position of the first item.

- bpo-33954 [https://bugs.python.org/issue?@action=redirect&bpo=33954]: For **`str.format()`**, **`float.__format__()`** and **`complex.__format__()`** methods for non-ASCII decimal point when using the “n” formatter.

- bpo-35269 [https://bugs.python.org/issue?@action=redirect&bpo=35269]: Fix a possible segfault involving a newly created coroutine. Patch by Zackery Spytz.

- bpo-35224 [https://bugs.python.org/issue?@action=redirect&bpo=35224]: Implement **PEP 572** [https://peps.python.org/pep-0572/] (assignment expressions). Patch by Emily Morehouse.

- bpo-32492 [https://bugs.python.org/issue?@action=redirect&bpo=32492]: Speed up **`namedtuple()`** attribute access by 1.6x using a C fast-path for the name descriptors. Patch by Pablo Galindo.

- bpo-35214 [https://bugs.python.org/issue?@action=redirect&bpo=35214]: Fixed an out of bounds memory access when parsing a truncated unicode escape sequence at the end of a string such as `'\N'`. It would read one byte beyond the end of the memory allocation.

- bpo-35214 [https://bugs.python.org/issue?@action=redirect&bpo=35214]: The interpreter and extension modules have had annotations added so that they work properly under clang's Memory Sanitizer. A new configure flag –with-memory-sanitizer has been added to make test builds of this nature easier to perform.

- bpo-35193 [https://bugs.python.org/issue?@action=redirect&bpo=35193]: Fix an off by one error in the bytecode peephole optimizer where it could read bytes beyond the end of bounds of an array when removing unreachable code. This bug was present in every release of Python 3.6 and 3.7 until now.

- bpo-35169 [https://bugs.python.org/issue?@action=redirect&bpo=35169]: Improved error messages for forbidden assignments.

- bpo-34022 [https://bugs.python.org/issue?@action=redirect&bpo=34022]: Fix handling of hash-based bytecode files in **`zipimport`**. Patch by Elvis Pranskevichus.

- bpo-28401 [https://bugs.python.org/issue?@action=redirect&bpo=28401]: Debug builds will no longer to attempt to import extension modules built for the ABI as they were never compatible to begin with. Patch by Stefano Rivera.

- bpo-29341 [https://bugs.python.org/issue?@action=redirect&bpo=29341]: Clarify in the docstrings of `os` methods that path-like objects are also accepted as input parameters.

- bpo-35050 [https://bugs.python.org/issue?@action=redirect&bpo=35050]: `socket`: Fix off-by-one bug in length check for `AF_ALG` name and type.

- bpo-29743 [https://bugs.python.org/issue?@action=redirect&bpo=29743]: Raise `ValueError` instead of `OverflowError` in case of a negative `_length_` in a `ctypes.Array` subclass. Also raise `TypeError` instead of `AttributeError` for non-integer `_length_`. Original patch by Oren Milman.

- bpo-16806 [https://bugs.python.org/issue?@action=redirect&bpo=16806]: Fix `lineno` and `col_offset` for multi-line string tokens.

- bpo-35029 [https://bugs.python.org/issue?@action=redirect&bpo=35029]: `SyntaxWarning` raised as an exception at code generation time will be now replaced with a `SyntaxError` for better error reporting.

- bpo-34983 [https://bugs.python.org/issue?@action=redirect&bpo=34983]: Expose `symtable.Symbol.is_nonlocal()` in the symtable module. Patch by Pablo Galindo.

- bpo-34974 [https://bugs.python.org/issue?@action=redirect&bpo=34974]: **`bytes`** and **`bytearray`** constructors no longer convert unexpected exceptions (e.g. **`MemoryError`** and **`KeyboardInterrupt`**) to **`TypeError`**.

- bpo-34939 [https://bugs.python.org/issue?@action=redirect&bpo=34939]: Allow annotated names in module namespace that are declared global before the annotation happens. Patch by Pablo Galindo.

- bpo-34973 [https://bugs.python.org/issue?@action=redirect&bpo=34973]: Fixed crash in **`bytes()`** when the **`list`** argument is mutated while it is iterated.

- bpo-34876 [https://bugs.python.org/issue?@action=redirect&bpo=34876]: The *lineno* and *col_offset* attributes of the AST for decorated function and class refer now to the position of the corresponding `def`, `async def` and `class` instead of the position of the first decorator. This leads to more correct line reporting in tracing. This is the only case when the position of child AST nodes can precede the position of the parent AST node.

- bpo-34879 [https://bugs.python.org/issue?@action=redirect&bpo=34879]: Fix a possible null pointer dereference in bytesobject.c. Patch by Zackery Spytz.

- bpo-34784 [https://bugs.python.org/issue?@action=redirect&bpo=34784]: Fix the implementation of PyStructSequence_NewType in order to create heap allocated StructSequences.

- bpo-32912 [https://bugs.python.org/issue?@action=redirect&bpo=32912]: A **`SyntaxWarning`** is now emitted instead of a **`DeprecationWarning`** for invalid escape sequences in string and bytes literals.

- bpo-34854 [https://bugs.python.org/issue?@action=redirect&bpo=34854]: Fixed a crash in compiling string annotations containing a lambda with a keyword-only argument that doesn't have a default value.

- bpo-34850 [https://bugs.python.org/issue?@action=redirect&bpo=34850]: The compiler now produces a `SyntaxWarning` when identity checks (`is` and `is not`) are used with certain types of literals (e.g. strings, ints). These can often work by accident in CPython, but are not guaranteed by the language spec. The warning advises users to use equality tests (`==` and `!=`) instead.

- bpo-34824 [https://bugs.python.org/issue?@action=redirect&bpo=34824]: Fix a possible null pointer dereference in Modules/_ssl.c. Patch by Zackery Spytz.

- bpo-30156 [https://bugs.python.org/issue?@action=redirect&bpo=30156]: The C function `property_descr_get()` uses a "cached" tuple to optimize function calls. But this tuple can be discovered in debug mode with `sys.getobjects()`. Remove the optimization, it's not really worth it and it causes 3 different crashes last years.

- bpo-34762 [https://bugs.python.org/issue?@action=redirect&bpo=34762]: Fix contextvars C API to use PyObject* pointer types.

- bpo-34751 [https://bugs.python.org/issue?@action=redirect&bpo=34751]: The hash function for tuples is now based on xxHash which gives better collision results on (formerly) pathological cases. Additionally, on 64-bit systems it improves tuple hashes in general. Patch by Jeroen Demeyer with substantial contributions by Tim Peters.

- bpo-34735 [https://bugs.python.org/issue?@action=redirect&bpo=34735]: Fix a memory leak in Modules/timemodule.c. Patch by Zackery Spytz.

- bpo-34683 [https://bugs.python.org/issue?@action=redirect&bpo=34683]: Fixed a bug where some SyntaxError error pointed to locations that were off-by-one.

- bpo-34651 [https://bugs.python.org/issue?@action=redirect&bpo=34651]: Only allow the main interpreter to fork. The avoids the possibility of affecting the main interpreter, which is critical to operation of the runtime.

- bpo-34653 [https://bugs.python.org/issue?@action=redirect&bpo=34653]: Remove unused function PyParser_SimpleParseStringFilename.

- bpo-32236 [https://bugs.python.org/issue?@action=redirect&bpo=32236]: Warn that line buffering is not supported if `open()` is called with binary mode and `buffering=1`.

- bpo-34641 [https://bugs.python.org/issue?@action=redirect&bpo=34641]: Further restrict the syntax of the left-hand side of keyword arguments in function calls. In particular, `f((keyword)=arg)` is now disallowed.

- bpo-34637 [https://bugs.python.org/issue?@action=redirect&bpo=34637]: Make the *start* argument to *sum()* visible as a keyword argument.

- bpo-1621 [https://bugs.python.org/issue?@action=redirect&bpo=1621]: Do not assume signed integer overflow behavior (C undefined behavior) when performing set hash table resizing.

- bpo-34588 [https://bugs.python.org/issue?@action=redirect&bpo=34588]: Fix an off-by-one in the recursive call pruning feature of traceback formatting.

- bpo-34485 [https://bugs.python.org/issue?@action=redirect&bpo=34485]: On Windows, the LC_CTYPE is now set to the user preferred locale at startup. Previously, the LC_CTYPE locale was “C” at startup, but changed when calling setlocale(LC_CTYPE, “”) or setlocale(LC_ALL, “”).

- bpo-34485 [https://bugs.python.org/issue?@action=redirect&bpo=34485]: Standard streams like sys.stdout now use the “surrogateescape” error handler, instead of “strict”, on the POSIX locale (when the C locale is not coerced and the UTF-8 Mode is disabled).

- bpo-34485 [https://bugs.python.org/issue?@action=redirect&bpo=34485]: Fix the error handler of standard streams like sys.stdout: PYTHONIOENCODING=”:” is now ignored instead of setting the error handler to “strict”.

- bpo-34485 [https://bugs.python.org/issue?@action=redirect&bpo=34485]: Python now gets the locale encoding with C code to initialize the encoding of standard streams like sys.stdout. Moreover, the encoding is now initialized to the Python codec name to get a normalized encoding name and to ensure that the codec is loaded. The change avoids importing _bootlocale and _locale modules at startup by default.

- bpo-34527 [https://bugs.python.org/issue?@action=redirect&bpo=34527]: On FreeBSD, Py_DecodeLocale() and Py_EncodeLocale() now also forces the ASCII encoding if the LC_CTYPE locale is “POSIX”, not only if the LC_CTYPE locale is “C”.

- bpo-34527 [https://bugs.python.org/issue?@action=redirect&bpo=34527]: The UTF-8 Mode is now also enabled by the “POSIX” locale, not only by the “C” locale.
- bpo-34403 [https://bugs.python.org/issue?@action=redirect&bpo=34403]: On HP-UX with C or POSIX locale, sys.getfilesystemencoding() now returns “ascii” instead of “roman8” (when the UTF-8 Mode is disabled and the C locale is not coerced).
- bpo-34523 [https://bugs.python.org/issue?@action=redirect&bpo=34523]: The Python filesystem encoding is now read earlier during the Python initialization.
- bpo-12458 [https://bugs.python.org/issue?@action=redirect&bpo=12458]: Tracebacks show now correct line number for subexpressions in multiline expressions. Tracebacks show now the line number of the first line for multiline expressions instead of the line number of the last subexpression.
- bpo-34408 [https://bugs.python.org/issue?@action=redirect&bpo=34408]: Prevent a null pointer dereference and resource leakage in `PyInterpreterState_New()`.
- bpo-34400 [https://bugs.python.org/issue?@action=redirect&bpo=34400]: Fix undefined behavior in parsetok.c. Patch by Zackery Spytz.
- bpo-33073 [https://bugs.python.org/issue?@action=redirect&bpo=33073]: Added as_integer_ratio to ints

to make them more interoperable with floats.

- bpo-34377 [https://bugs.python.org/issue?@action=redirect&bpo=34377]: Update valgrind suppression list to use `_PyObject_Free`/`_PyObject_Realloc` instead of `PyObject_Free`/`PyObject_Realloc`.
- bpo-34353 [https://bugs.python.org/issue?@action=redirect&bpo=34353]: Added the “socket” option in the `stat.filemode()` Python implementation to match the C implementation.
- bpo-34320 [https://bugs.python.org/issue?@action=redirect&bpo=34320]: Fix `dict(od)` didn’t copy iteration order of OrderedDict.
- bpo-34113 [https://bugs.python.org/issue?@action=redirect&bpo=34113]: Fixed crash on debug builds when opcode stack was adjusted with negative numbers. Patch by Constantin Petrisor.
- bpo-34100 [https://bugs.python.org/issue?@action=redirect&bpo=34100]: Compiler now merges constants in tuples and frozensets recursively. Code attributes like `co_names` are merged too.
- bpo-34151 [https://bugs.python.org/issue?@action=redirect&bpo=34151]: Performance of list concatenation, repetition and slicing operations is slightly improved. Patch by Sergey Fedoseev.
- bpo-34170 [https://bugs.python.org/issue?@action=redirect&bpo=34170]: -X dev: it is now possible to override the memory allocator using PYTHONMALLOC even if the developer mode is enabled.

- bpo-33237 [https://bugs.python.org/issue?@action=redirect&bpo=33237]: Improved `AttributeError` message for partially initialized module.

- bpo-34149 [https://bugs.python.org/issue?@action=redirect&bpo=34149]: Fix min and max functions to get default behavior when key is None.

- bpo-34125 [https://bugs.python.org/issue?@action=redirect&bpo=34125]: Profiling of unbound built-in methods now works when `**kwargs` is given.

- bpo-34141 [https://bugs.python.org/issue?@action=redirect&bpo=34141]: Optimized pickling atomic types (None, bool, int, float, bytes, str).

- bpo-34126 [https://bugs.python.org/issue?@action=redirect&bpo=34126]: Fix crashes when profiling certain invalid calls of unbound methods. Patch by Jeroen Demeyer.

- bpo-24618 [https://bugs.python.org/issue?@action=redirect&bpo=24618]: Fixed reading invalid memory when create the code object with too small varnames tuple or too large argument counts.

- bpo-34068 [https://bugs.python.org/issue?@action=redirect&bpo=34068]: In `io.IOBase.close()`, ensure that the `closed` attribute is not set with a live exception. Patch by Zackery Spytz and Serhiy Storchaka.

- bpo-34087 [https://bugs.python.org/issue?@action=redirect&bpo=34087]: Fix buffer overflow while converting unicode to numeric values.

- bpo-34080 [https://bugs.python.org/issue?@action=redirect&bpo=34080]: Fixed a memory leak in the compiler when it raised some uncommon errors during tokenizing.

- bpo-34066 [https://bugs.python.org/issue?@action=redirect&bpo=34066]: Disabled interruption by Ctrl-C between calling `open()` and entering a **with** block in `with open()`.
- bpo-34042 [https://bugs.python.org/issue?@action=redirect&bpo=34042]: Fix dict.copy() to maintain correct total refcount (as reported by sys.gettotalrefcount()).
- bpo-33418 [https://bugs.python.org/issue?@action=redirect&bpo=33418]: Fix potential memory leak in function object when it creates reference cycle.
- bpo-33985 [https://bugs.python.org/issue?@action=redirect&bpo=33985]: Implement contextvars.ContextVar.name attribute.
- bpo-33956 [https://bugs.python.org/issue?@action=redirect&bpo=33956]: Update vendored Expat library copy to version 2.2.5.
- bpo-24596 [https://bugs.python.org/issue?@action=redirect&bpo=24596]: Decref the module object in **`PyRun_SimpleFileExFlags()`** before calling **`PyErr_Print()`**. Patch by Zackery Spytz.
- bpo-33451 [https://bugs.python.org/issue?@action=redirect&bpo=33451]: Close directly executed pyc files before calling `PyEval_EvalCode()`.
- bpo-1617161 [https://bugs.python.org/issue?@action=redirect&bpo=1617161]: The hash of **`BuiltinMethodType`** instances (methods of built-in classes) now depends on

the hash of the identity of __*self*__ instead of its value. The hash and equality of **`ModuleType`** and **`MethodWrapperType`** instances (methods of user-defined classes and some methods of built-in classes like `str.__add__`) now depend on the hash and equality of the identity of __*self*__ instead of its value. **`MethodWrapperType`** instances no longer support ordering.

- bpo-33824 [https://bugs.python.org/issue?@action=redirect&bpo=33824]: Fix "LC_ALL=C python3.7 -V": reset properly the command line parser when the encoding changes after reading the Python configuration.

- bpo-33803 [https://bugs.python.org/issue?@action=redirect&bpo=33803]: Fix a crash in hamt.c caused by enabling GC tracking for an object that hadn't all of its fields set to NULL.

- bpo-33738 [https://bugs.python.org/issue?@action=redirect&bpo=33738]: Seven macro incompatibilities with the Limited API were fixed, and the macros **`PyIter_Check()`**, **`PyIndex_Check()`** and **`PyExceptionClass_Name()`** were added as functions. A script for automatic macro checks was added.

- bpo-33786 [https://bugs.python.org/issue?@action=redirect&bpo=33786]: Fix asynchronous generators to handle GeneratorExit in athrow() correctly

- bpo-30167 [https://bugs.python.org/issue?@action=redirect&bpo=30167]: `PyRun_SimpleFileExFlags` removes `__cached__` from module in addition to `__file__`.

- bpo-33706 [https://bugs.python.org/issue?@action=redirect&bpo=33706]: Fix a crash in Python initialization when parsing the command line options. Thanks Christoph Gohlke for the bug report and the fix!

- bpo-33597 [https://bugs.python.org/issue?@action=redirect&bpo=33597]: Reduce `PyGC_Head` size from 3 words to 2 words.

- bpo-30654 [https://bugs.python.org/issue?@action=redirect&bpo=30654]: Fixed reset of the SIGINT handler to SIG_DFL on interpreter shutdown even when there was a custom handler set previously. Patch by Philipp Kerling.

- bpo-33622 [https://bugs.python.org/issue?@action=redirect&bpo=33622]: Fixed a leak when the garbage collector fails to add an object with the `__del__` method or referenced by it into the **`gc.garbage`** list. **`PyGC_Collect()`** can now be called when an exception is set and preserves it.

- bpo-33462 [https://bugs.python.org/issue?@action=redirect&bpo=33462]: Make dict and dict views reversible. Patch by Rémi Lapeyre.

- bpo-23722 [https://bugs.python.org/issue?@action=redirect&bpo=23722]: A **`RuntimeError`** is now raised when the custom metaclass doesn't provide the `__classcell__` entry in the namespace passed to `type.__new__`. A **`DeprecationWarning`** was emitted in Python 3.6–3.7.

- bpo-33499 [https://bugs.python.org/issue?@action=redirect&bpo=33499]: Add **`PYTHONPYCACHEPREFIX`** environment variable and **`-X`** `pycache_prefix` command-line option to set an alternate root directory for writing module bytecode cache files.

- bpo-25711 [https://bugs.python.org/issue?@action=redirect&bpo=25711]: The **`zipimport`** module has been rewritten in pure Python.

- bpo-33509 [https://bugs.python.org/issue?@action=redirect&bpo=33509]: Fix module_globals parameter of warnings.warn_explicit(): don’t crash if module_globals is not a dict.

- bpo-31849 [https://bugs.python.org/issue?@action=redirect&bpo=31849]: Fix signed/unsigned comparison warning in pyhash.c.

- bpo-33475 [https://bugs.python.org/issue?@action=redirect&bpo=33475]: Fixed miscellaneous bugs in converting annotations to strings and optimized parentheses in the string representation.

- bpo-20104 [https://bugs.python.org/issue?@action=redirect&bpo=20104]: Added support for the `setpgroup`, `resetids`, `setsigmask`, `setsigdef` and `scheduler` parameters of `posix_spawn`. Patch by Pablo Galindo.

- bpo-33391 [https://bugs.python.org/issue?@action=redirect&bpo=33391]: Fix a leak in set_symmetric_difference().

- bpo-33363 [https://bugs.python.org/issue?@action=redirect&bpo=33363]: Raise a SyntaxError for `async with` and `async for` statements outside of async functions.

- bpo-28055 [https://bugs.python.org/issue?@action=redirect&bpo=28055]: Fix unaligned accesses in siphash24(). Patch by Rolf Eike Beer.

- bpo-33128 [https://bugs.python.org/issue?@action=redirect&bpo=33128]: Fix a bug that causes

PathFinder to appear twice on sys.meta_path. Patch by Pablo Galindo Salgado.

- bpo-33331 [https://bugs.python.org/issue?@action=redirect&bpo=33331]: Modules imported last are now cleared first at interpreter shutdown.
- bpo-33312 [https://bugs.python.org/issue?@action=redirect&bpo=33312]: Fixed clang ubsan (undefined behavior sanitizer) warnings in dictobject.c by adjusting how the internal struct _dictkeysobject shared keys structure is declared.
- bpo-33305 [https://bugs.python.org/issue?@action=redirect&bpo=33305]: Improved syntax error messages for invalid numerical literals.
- bpo-33306 [https://bugs.python.org/issue?@action=redirect&bpo=33306]: Improved syntax error messages for unbalanced parentheses.
- bpo-33234 [https://bugs.python.org/issue?@action=redirect&bpo=33234]: The list constructor will pre-size and not over-allocate when the input length is known.
- bpo-33270 [https://bugs.python.org/issue?@action=redirect&bpo=33270]: Intern the names for all anonymous code objects. Patch by Zackery Spytz.
- bpo-30455 [https://bugs.python.org/issue?@action=redirect&bpo=30455]: The C and Python code and the documentation related to tokens are now generated from a single source file `Grammar/Tokens`.
- bpo-33176 [https://bugs.python.org/issue?@action=redirect&bpo=33176]: Add a `toreadonly()` method to memoryviews.

- bpo-33231 [https://bugs.python.org/issue?@action=redirect&bpo=33231]: Fix potential memory leak in `normalizestring()`.

- bpo-33205 [https://bugs.python.org/issue?@action=redirect&bpo=33205]: Change dict growth function from `round_up_to_power_2(used*2+hashtable_size/2)` to `round_up_to_power_2(used*3)`. Previously, dict is shrunk only when `used == 0`. Now dict has more chance to be shrunk.

- bpo-29922 [https://bugs.python.org/issue?@action=redirect&bpo=29922]: Improved error messages in 'async with' when `__aenter__()` or `__aexit__()` return non-awaitable object.

- bpo-33199 [https://bugs.python.org/issue?@action=redirect&bpo=33199]: Fix `ma_version_tag` in dict implementation is uninitialized when copying from key-sharing dict.

- bpo-33053 [https://bugs.python.org/issue?@action=redirect&bpo=33053]: When using the -m switch, sys.path[0] is now explicitly expanded as the *starting* working directory, rather than being left as the empty path (which allows imports from the current working directory at the time of the import)

- bpo-33138 [https://bugs.python.org/issue?@action=redirect&bpo=33138]: Changed standard error message for non-pickleable and non-copyable types. It now says "cannot pickle" instead of "can't pickle" or "cannot serialize".

- bpo-33018 [https://bugs.python.org/issue?@action=redirect&bpo=33018]: Improve consistency of errors raised by `issubclass()` when called with a non-class and an abstract base class as the first and second arguments, respectively. Patch by Josh Bronson.

- bpo-33083 [https://bugs.python.org/issue?@action=redirect&bpo=33083]: `math.factorial` no longer accepts arguments that are not int-like. Patch by Pablo Galindo.

- bpo-33041 [https://bugs.python.org/issue?@action=redirect&bpo=33041]: Added new opcode **`END_ASYNC_FOR`** and fixes the following issues:

  - Setting global **`StopAsyncIteration`** no longer breaks `async for` loops.
  - Jumping into an `async for` loop is now disabled.
  - Jumping out of an `async for` loop no longer corrupts the stack.

- bpo-25750 [https://bugs.python.org/issue?@action=redirect&bpo=25750]: Fix rare Python crash due to bad refcounting in `type_getattro()` if a descriptor deletes itself from the class. Patch by Jeroen Demeyer.

- bpo-33041 [https://bugs.python.org/issue?@action=redirect&bpo=33041]: Fixed bytecode generation for “async for” with a complex target. A StopAsyncIteration raised on assigning or unpacking will be now propagated instead of stopping the iteration.

- bpo-33026 [https://bugs.python.org/issue?@action=redirect&bpo=33026]: Fixed jumping out of “with” block by setting f_lineno.

- bpo-33005 [https://bugs.python.org/issue?@action=redirect&bpo=33005]: Fix a crash on fork when using a custom memory allocator (ex: using PYTHONMALLOC env var). _PyGILState_Reinit() and

_PyInterpreterState_Enable() now use the default RAW memory allocator to allocate a new interpreters mutex on fork.

- bpo-32911 [https://bugs.python.org/issue?@action=redirect&bpo=32911]: Due to unexpected compatibility issues discovered during downstream beta testing, reverted bpo-29463 [https://bugs.python.org/issue?@action=redirect&bpo=29463]. `docstring` field is removed from Module, ClassDef, FunctionDef, and AsyncFunctionDef ast nodes which was added in 3.7a1. Docstring expression is restored as a first statement in their body. Based on patch by Inada Naoki.

- bpo-17288 [https://bugs.python.org/issue?@action=redirect&bpo=17288]: Prevent jumps from ‘return’ and ‘exception’ trace events.

- bpo-32946 [https://bugs.python.org/issue?@action=redirect&bpo=32946]: Importing names from already imported module with “from … import …” is now 30% faster if the module is not a package.

- bpo-32932 [https://bugs.python.org/issue?@action=redirect&bpo=32932]: Make error message more revealing when there are non-str objects in `__all__`.

- bpo-32925 [https://bugs.python.org/issue?@action=redirect&bpo=32925]: Optimized iterating and containing test for literal lists consisting of non-constants: `x in [a, b]` and `for x in [a, b]`. The case of all constant elements already was optimized.

- bpo-32889 [https://bugs.python.org/issue?@action=redirect&bpo=32889]: Update Valgrind suppression list to account for the rename of `Py_ADDRESS_IN_RANG` to `address_in_range`.

- bpo-32836 [https://bugs.python.org/issue?@action=redirect&bpo=32836]: Don’t use temporary variables in cases of list/dict/set comprehensions

- bpo-31356 [https://bugs.python.org/issue?@action=redirect&bpo=31356]: Remove the new API added in bpo-31356 [https://bugs.python.org/issue?@action=redirect&bpo=31356] (gc.ensure_disabled() context manager).

- bpo-32305 [https://bugs.python.org/issue?@action=redirect&bpo=32305]: For namespace packages, ensure that both `__file__` and `__spec__.origin` are set to None.

- bpo-32303 [https://bugs.python.org/issue?@action=redirect&bpo=32303]: Make sure `__spec__.loader` matches `__loader__` for namespace packages.

- bpo-32711 [https://bugs.python.org/issue?@action=redirect&bpo=32711]: Fix the warning messages for Python/ast_unparse.c. Patch by Stéphane Wirtel

- bpo-32583 [https://bugs.python.org/issue?@action=redirect&bpo=32583]: Fix possible crashing in builtin Unicode decoders caused by write out-of-bound errors when using customized decode error handlers.

- bpo-32489 [https://bugs.python.org/issue?@action=redirect&bpo=32489]: A **`continue`** statement is now allowed in the **`finally`** clause.

- bpo-17611 [https://bugs.python.org/issue?@action=redirect&bpo=17611]: Simplified the interpreter loop by moving the logic of unrolling the stack of blocks into the compiler. The compiler emits now explicit instructions for adjusting the stack of values and calling the cleaning up code for **`break`**, **`continue`** and **`return`**.

Removed opcodes **`BREAK_LOOP`**, **`CONTINUE_LOOP`**, **`SETUP_LOOP`** and **`SETUP_EXCEPT`**. Added new opcodes **`ROT_FOUR`**, **`BEGIN_FINALLY`** and **`CALL_FINALLY`** and **`POP_FINALLY`**. Changed the behavior of **`END_FINALLY`** and **`WITH_CLEANUP_START`**.

- bpo-32285 [https://bugs.python.org/issue?@action=redirect&bpo=32285]: New function unicodedata.is_normalized, which can check whether a string is in a specific normal form.
- bpo-10544 [https://bugs.python.org/issue?@action=redirect&bpo=10544]: Yield expressions are now disallowed in comprehensions and generator expressions except the expression for the outermost iterable.
- bpo-32117 [https://bugs.python.org/issue?@action=redirect&bpo=32117]: Iterable unpacking is now allowed without parentheses in yield and return statements, e.g. `yield 1, 2, 3, *rest`. Thanks to David Cuthbert for the change and Jordan Chapman for added tests.
- bpo-31902 [https://bugs.python.org/issue?@action=redirect&bpo=31902]: Fix the `col_offset` attribute for ast nodes `ast.AsyncFor`, `ast.AsyncFunctionDef`, and `ast.AsyncWith`. Previously, `col_offset` pointed to the keyword after `async`.
- bpo-25862 [https://bugs.python.org/issue?@action=redirect&bpo=25862]: Fix assertion failures in the `tell()` method of `io.TextIOWrapper`. Patch by Zackery Spytz.

- bpo-21983 [https://bugs.python.org/issue?@action=redirect&bpo=21983]: Fix a crash in `ctypes.cast()` in case the type argument is a ctypes structured data type. Patch by Eryk Sun and Oren Milman.

- bpo-31577 [https://bugs.python.org/issue?@action=redirect&bpo=31577]: Fix a crash in `os.utime()` in case of a bad ns argument. Patch by Oren Milman.

- bpo-29832 [https://bugs.python.org/issue?@action=redirect&bpo=29832]: Remove references to 'getsockaddrarg' from various socket error messages. Patch by Oren Milman.

Library

- bpo-35845 [https://bugs.python.org/issue?@action=redirect&bpo=35845]: Add 'order' parameter to memoryview.tobytes().

- bpo-35864 [https://bugs.python.org/issue?@action=redirect&bpo=35864]: The _asdict() method for collections.namedtuple now returns a regular dict instead of an OrderedDict.

- bpo-35537 [https://bugs.python.org/issue?@action=redirect&bpo=35537]: An ExitStack is now used internally within subprocess.Popen to clean up pipe file handles. No behavior change in normal operation. But if closing one handle were ever to cause an exception, the others will now be closed instead of leaked. (patch by Giampaolo Rodola)

- bpo-35847 [https://bugs.python.org/issue?@action=redirect&bpo=35847]: RISC-V needed the

CTYPES_PASS_BY_REF_HACK. Fixes ctypes Structure test_pass_by_value.

- bpo-35813 [https://bugs.python.org/issue?@action=redirect&bpo=35813]: Shared memory submodule added to multiprocessing to avoid need for serialization between processes

- bpo-35780 [https://bugs.python.org/issue?@action=redirect&bpo=35780]: Fix lru_cache() errors arising in recursive, reentrant, or multi-threaded code. These errors could result in orphan links and in the cache being trapped in a state with fewer than the specified maximum number of links. Fix handling of negative maxsize which should have been treated as zero. Fix errors in toggling the “full” status flag. Fix misordering of links when errors are encountered. Sync-up the C code and pure Python code for the space saving path in functions with a single positional argument. In this common case, the space overhead of an lru cache entry is reduced by almost half. Fix counting of cache misses. In error cases, the miss count was out of sync with the actual number of times the underlying user function was called.

- bpo-35537 [https://bugs.python.org/issue?@action=redirect&bpo=35537]: **`os.posix_spawn()`** and **`os.posix_spawnp()`** now have a *setsid* parameter.

- bpo-23846 [https://bugs.python.org/issue?@action=redirect&bpo=23846]: **`asyncio.ProactorEventLoop`** now catches and logs send errors when the self-pipe is full.

- bpo-34323 [https://bugs.python.org/issue?@action=redirect&bpo=34323]: **`asyncio`**: Enhance `IocpProactor.close()` log: wait 1 second before the first log, then log every second. Log also the number of seconds since `close()` was called.

- bpo-35674 [https://bugs.python.org/issue?@action=redirect&bpo=35674]: Add a new `os.posix_spawnp()` function. Patch by Joannah Nanjekye.
- bpo-35733 [https://bugs.python.org/issue?@action=redirect&bpo=35733]: `ast.Constant(boolean)` no longer an instance of **`ast.Num`**. Patch by Anthony Sottile.
- bpo-35726 [https://bugs.python.org/issue?@action=redirect&bpo=35726]: QueueHandler.prepare() now makes a copy of the record before modifying and enqueueing it, to avoid affecting other handlers in the chain.
- bpo-35719 [https://bugs.python.org/issue?@action=redirect&bpo=35719]: Sped up multi-argument `math` functions atan2(), copysign(), remainder() and hypot() by 1.3–2.5 times.
- bpo-35717 [https://bugs.python.org/issue?@action=redirect&bpo=35717]: Fix KeyError exception raised when using enums and compile. Patch contributed by Rémi Lapeyre.
- bpo-35699 [https://bugs.python.org/issue?@action=redirect&bpo=35699]: Fixed detection of Visual Studio Build Tools 2017 in distutils
- bpo-32710 [https://bugs.python.org/issue?@action=redirect&bpo=32710]: Fix memory leaks in asyncio ProactorEventLoop on overlapped operation failure.
- bpo-35702 [https://bugs.python.org/issue?@action=redirect&bpo=35702]: The `time.CLOCK_UPTIME_RAW`

constant is now available for macOS 10.12.

- bpo-32710 [https://bugs.python.org/issue?@action=redirect&bpo=32710]: Fix a memory leak in asyncio in the ProactorEventLoop when `ReadFile()` or `WSASend()` overlapped operation fail immediately: release the internal buffer.

- bpo-35682 [https://bugs.python.org/issue?@action=redirect&bpo=35682]: Fix `asyncio.ProactorEventLoop.sendfile()`: don’t attempt to set the result of an internal future if it’s already done.

- bpo-35283 [https://bugs.python.org/issue?@action=redirect&bpo=35283]: Add a deprecated warning for the **`threading.Thread.isAlive()`** method. Patch by Donghee Na.

- bpo-35664 [https://bugs.python.org/issue?@action=redirect&bpo=35664]: Improve operator.itemgetter() performance by 33% with optimized argument handling and with adding a fast path for the common case of a single non-negative integer index into a tuple (which is the typical use case in the standard library).

- bpo-35643 [https://bugs.python.org/issue?@action=redirect&bpo=35643]: Fixed a SyntaxWarning: invalid escape sequence in Modules/_sha3/cleanup.py. Patch by Mickaël Schoentgen.

- bpo-35619 [https://bugs.python.org/issue?@action=redirect&bpo=35619]: Improved support of custom data descriptors in **`help()`** and **`pydoc`**.

- bpo-28503 [https://bugs.python.org/issue?@action=redirect&bpo=28503]: The `crypt` module now internally uses the `crypt_r()` library function instead of `crypt()` when available.

- bpo-35614 [https://bugs.python.org/issue?@action=redirect&bpo=35614]: Fixed help() on metaclasses. Patch by Sanyam Khurana.

- bpo-35568 [https://bugs.python.org/issue?@action=redirect&bpo=35568]: Expose `raise(signum)` as `raise_signal`

- bpo-35588 [https://bugs.python.org/issue?@action=redirect&bpo=35588]: The floor division and modulo operations and the **`divmod()`** function on **`fractions.Fraction`** types are 2–4x faster. Patch by Stefan Behnel.

- bpo-35585 [https://bugs.python.org/issue?@action=redirect&bpo=35585]: Speed-up building enums by value, e.g. http.HTTPStatus(200).

- bpo-30561 [https://bugs.python.org/issue?@action=redirect&bpo=30561]: random.gammavariate(1.0, beta) now computes the same result as random.expovariate(1.0 / beta). This synchronizes the two algorithms and eliminates some idiosyncrasies in the old implementation. It does however produce a difference stream of random variables than it used to.

- bpo-35537 [https://bugs.python.org/issue?@action=redirect&bpo=35537]: The **`subprocess`** module can now use the **`os.posix_spawn()`** function in some cases for better performance.

- bpo-35526 [https://bugs.python.org/issue?@action=redirect&bpo=35526]: Delaying the ‘joke’ of barry_as_FLUFL.mandatory to Python version 4.0

- bpo-35523 [https://bugs.python.org/issue?@action=redirect&bpo=35523]: Remove **`ctypes`** callback workaround: no longer create a callback at startup. Avoid SELinux alert on `import ctypes` and `import uuid`.

- bpo-31784 [https://bugs.python.org/issue?@action=redirect&bpo=31784]: **`uuid.uuid1()`** now calls **`time.time_ns()`** rather than `int(time.time() * 1e9)`.

- bpo-35513 [https://bugs.python.org/issue?@action=redirect&bpo=35513]: **`TextTestRunner`** of **`unittest.runner`** now uses **`time.perf_counter()`** rather than **`time.time()`** to measure the execution time of a test: **`time.time()`** can go backwards, whereas **`time.perf_counter()`** is monotonic.

- bpo-35502 [https://bugs.python.org/issue?@action=redirect&bpo=35502]: Fixed reference leaks in **`xml.etree.ElementTree.TreeBuilder`** in case of unfinished building of the tree (in particular when an error was raised during parsing XML).

- bpo-35348 [https://bugs.python.org/issue?@action=redirect&bpo=35348]: Make **`platform.architecture()`** parsing of `file` command output more reliable: add the `-b` option to the `file` command to omit the filename, force the usage of the C locale, and search also the “shared object” pattern.

- bpo-35491 [https://bugs.python.org/issue?@action=redirect&bpo=35491]: **`multiprocessing`**: Add `Pool.__repr__()` and enhance `BaseProcess.__repr__()` (add pid and parent pid) to ease debugging. Pool state constant values are now strings instead of integers, for example `RUN` value becomes `'RUN'` instead of `0`.

- bpo-35477 [https://bugs.python.org/issue?@action=redirect&bpo=35477]: **`multiprocessing.Pool.__enter__()`**

now fails if the pool is not running: `with pool:` fails if used more than once.

- bpo-31446 [https://bugs.python.org/issue?@action=redirect&bpo=31446]: Copy command line that was passed to CreateProcessW since this function can change the content of the input buffer.

- bpo-35471 [https://bugs.python.org/issue?@action=redirect&bpo=35471]: Python 2.4 dropped MacOS 9 support. The macpath module was deprecated in Python 3.7. The module is now removed.

- bpo-23057 [https://bugs.python.org/issue?@action=redirect&bpo=23057]: Unblock Proactor event loop when keyboard interrupt is received on Windows

- bpo-35052 [https://bugs.python.org/issue?@action=redirect&bpo=35052]: Fix xml.dom.minidom cloneNode() on a document with an entity: pass the correct arguments to the user data handler of an entity.

- bpo-20239 [https://bugs.python.org/issue?@action=redirect&bpo=20239]: Allow repeated assignment deletion of **`unittest.mock.Mock`** attributes. Patch by Pablo Galindo.

- bpo-17185 [https://bugs.python.org/issue?@action=redirect&bpo=17185]: Set `__signature__` on mock for **`inspect`** to get signature. Patch by Karthikeyan Singaravelan.

- bpo-35445 [https://bugs.python.org/issue?@action=redirect&bpo=35445]: Memory errors during creating posix.environ no longer ignored.

- bpo-35415 [https://bugs.python.org/issue?@action=redirect&bpo=35415]: Validate fileno= argument to socket.socket().

- bpo-35424 [https://bugs.python.org/issue?@action=redirect&bpo=35424]: **`multiprocessing.Pool`** destructor now emits **`ResourceWarning`** if the pool is still running.

- bpo-35330 [https://bugs.python.org/issue?@action=redirect&bpo=35330]: When a **`Mock`** instance was used to wrap an object, if `side_effect` is used in one of the mocks of it methods, don't call the original implementation and return the result of using the side effect the same way that it is done with return_value.

- bpo-35346 [https://bugs.python.org/issue?@action=redirect&bpo=35346]: Drop Mac OS 9 and Rhapsody support from the **`platform`** module. Rhapsody last release was in 2000. Mac OS 9 last release was in 2001.

- bpo-10496 [https://bugs.python.org/issue?@action=redirect&bpo=10496]: **`check_environ()`** of `distutils.utils` now catches **`KeyError`** on calling **`pwd.getpwuid()`**: don't create the `HOME` environment variable in this case.

- bpo-10496 [https://bugs.python.org/issue?@action=redirect&bpo=10496]: **`posixpath.expanduser()`** now returns the input *path* unchanged if the `HOME` environment variable is not set and the current user has no home directory (if the current user identifier doesn't exist in the password database). This change fix the **`site`** module if the current user doesn't exist in the password database (if the user has no home directory).

- bpo-35389 [https://bugs.python.org/issue?@action=redirect&bpo=35389]: **`platform.libc_ver()`** now uses `os.confstr('CS_GNU_LIBC_VERSION')` if available and the *executable* parameter is not set.

- bpo-35394 [https://bugs.python.org/issue?@action=redirect&bpo=35394]: Add empty slots to asyncio abstract protocols.

- bpo-35310 [https://bugs.python.org/issue?@action=redirect&bpo=35310]: Fix a bug in **`select.select()`** where, in some cases, the file descriptor sequences were returned unmodified after a signal interruption, even though the file descriptors might not be ready yet. **`select.select()`** will now always return empty lists if a timeout has occurred. Patch by Oran Avraham.

- bpo-35380 [https://bugs.python.org/issue?@action=redirect&bpo=35380]: Enable TCP_NODELAY on Windows for proactor asyncio event loop.

- bpo-35341 [https://bugs.python.org/issue?@action=redirect&bpo=35341]: Add generic version of `collections.OrderedDict` to the `typing` module. Patch by Ismo Toijala.

- bpo-35371 [https://bugs.python.org/issue?@action=redirect&bpo=35371]: Fixed possible crash in `os.utime()` on Windows when pass incorrect arguments.

- bpo-35346 [https://bugs.python.org/issue?@action=redirect&bpo=35346]: **`platform.uname()`** now redirects `stderr` to **`os.devnull`** when running external programs like `cmd /c ver`.

- bpo-35066 [https://bugs.python.org/issue?@action=redirect&bpo=35066]: Previously, calling the strftime() method on a datetime object with a trailing ‘%’ in the format string would result in an exception. However, this only occurred when the datetime C module was being used; the python implementation did not match this behavior. Datetime is now PEP-399 compliant, and will not throw an exception on a trailing ‘%’.

- bpo-35345 [https://bugs.python.org/issue?@action=redirect&bpo=35345]: The function `platform.popen` has been removed, it was deprecated since Python 3.3: use **`os.popen()`** instead.

- bpo-35344 [https://bugs.python.org/issue?@action=redirect&bpo=35344]: On macOS, **`platform.platform()`** now uses **`platform.mac_ver()`**, if it returns a non-empty release string, to get the macOS version rather than the darwin version.

- bpo-35312 [https://bugs.python.org/issue?@action=redirect&bpo=35312]: Make `lib2to3.pgen2.parse.ParseError` round-trip pickle-able. Patch by Anthony Sottile.

- bpo-35308 [https://bugs.python.org/issue?@action=redirect&bpo=35308]: Fix regression in `webbrowser` where default browsers may be preferred over browsers in the `BROWSER` environment variable.

- bpo-24746 [https://bugs.python.org/issue?@action=redirect&bpo=24746]: Avoid stripping trailing whitespace in doctest fancy diff. Original patch by R. David Murray & Jairo Trad. Enhanced by Sanyam Khurana.

- bpo-28604 [https://bugs.python.org/issue?@action=redirect&bpo=28604]: **`locale.localeconv()`** now sets temporarily the `LC_CTYPE` locale to the `LC_MONETARY` locale if the two locales are different and monetary strings are non-ASCII. This temporary change affects other threads.

- bpo-35277 [https://bugs.python.org/issue?@action=redirect&bpo=35277]: Update ensurepip to install pip 18.1 and setuptools 40.6.2.

- bpo-24209 [https://bugs.python.org/issue?@action=redirect&bpo=24209]: Adds IPv6 support when invoking http.server directly.

- bpo-35226 [https://bugs.python.org/issue?@action=redirect&bpo=35226]: Recursively check arguments when testing for equality of **`unittest.mock.call`** objects and add note that tracking of parameters used to create ancestors of mocks in `mock_calls` is not possible.

- bpo-29564 [https://bugs.python.org/issue?@action=redirect&bpo=29564]: The warnings module now suggests to enable tracemalloc if the source is specified, the tracemalloc module is available, but tracemalloc is not tracing memory allocations.

- bpo-35189 [https://bugs.python.org/issue?@action=redirect&bpo=35189]: Modify the following fnctl function to retry if interrupted by a signal (EINTR): flock, lockf, fnctl

- bpo-30064 [https://bugs.python.org/issue?@action=redirect&bpo=30064]: Use add_done_callback() in sock_* asyncio API to unsubscribe reader/writer early on calcellation.

- bpo-35186 [https://bugs.python.org/issue?@action=redirect&bpo=35186]: Removed the “built with” comment added when `setup.py upload` is used with either `bdist_rpm` or `bdist_dumb`.

- bpo-35152 [https://bugs.python.org/issue?@action=redirect&bpo=35152]: Allow sending more than 2 GB at once on a multiprocessing connection on non-Windows systems.

- bpo-35062 [https://bugs.python.org/issue?@action=redirect&bpo=35062]: Fix incorrect parsing of **`io.IncrementalNewlineDecoder`**’s *translate* argument.

- bpo-35065 [https://bugs.python.org/issue?@action=redirect&bpo=35065]: Remove `StreamReaderProtocol._untrack_reader`. The call to `_untrack_reader` is currently performed too soon, causing the protocol to forget about the reader before `connection_lost` can run and feed the EOF to the reader.

- bpo-34160 [https://bugs.python.org/issue?@action=redirect&bpo=34160]: ElementTree and minidom now preserve the attribute order specified by the user.

- bpo-35079 [https://bugs.python.org/issue?@action=redirect&bpo=35079]: Improve difflib.SequenceManager.get_matching_blocks doc by adding ‘non-overlapping’ and changing ‘!=’ to ‘<’.

- bpo-33710 [https://bugs.python.org/issue?@action=redirect&bpo=33710]: Deprecated `l*gettext()` functions and methods in the **`gettext`** module. They return encoded bytes instead of Unicode strings and are artifacts from Python 2 times. Also deprecated functions and methods related to setting the charset for `l*gettext()` functions and methods.

- bpo-35017 [https://bugs.python.org/issue?@action=redirect&bpo=35017]: **`socketserver.BaseServer.serve_forever()`** now exits immediately if it’s **`shutdown()`** method is called while it is polling for new events.

- bpo-35024 [https://bugs.python.org/issue?@action=redirect&bpo=35024]: `importlib` no longer logs `wrote <bytecode path>` redundantly after `(created|could not create) <bytecode path>` is already logged. Patch by Quentin Agren.

- bpo-35047 [https://bugs.python.org/issue?@action=redirect&bpo=35047]: `unittest.mock` now includes mock calls in exception messages if `assert_not_called`, `assert_called_once`, or `assert_called_once_with` fails. Patch by Petter Strandmark.

- bpo-31047 [https://bugs.python.org/issue?@action=redirect&bpo=31047]: Fix `ntpath.abspath` regression where it didn't remove a trailing separator on Windows. Patch by Tim Graham.

- bpo-35053 [https://bugs.python.org/issue?@action=redirect&bpo=35053]: tracemalloc now tries to update the traceback when an object is reused from a "free list" (optimization for faster object creation, used by the builtin list type for example).

- bpo-31553 [https://bugs.python.org/issue?@action=redirect&bpo=31553]: Add the –json-lines option to json.tool. Patch by hongweipeng.

- bpo-34794 [https://bugs.python.org/issue?@action=redirect&bpo=34794]: Fixed a leak in Tkinter when pass the Python wrapper around Tcl_Obj back to Tcl/Tk.

- bpo-34909 [https://bugs.python.org/issue?@action=redirect&bpo=34909]: Enum: fix grandchildren subclassing when parent mixed with concrete data types.

- bpo-35022 [https://bugs.python.org/issue?@action=redirect&bpo=35022]: `unittest.mock.MagicMock` now supports the `__fspath__` method (from `os.PathLike`).

- bpo-35008 [https://bugs.python.org/issue?@action=redirect&bpo=35008]: Fixed references leaks when call the `__setstate__()` method of `xml.etree.ElementTree.Element` in the C implementation for already initialized element.

- bpo-23420 [https://bugs.python.org/issue?@action=redirect&bpo=23420]: Verify the value for the parameter '-s' of the cProfile CLI. Patch by Robert Kuska

- bpo-33947 [https://bugs.python.org/issue?@action=redirect&bpo=33947]: dataclasses now handle recursive reprs without raising RecursionError.

- bpo-34890 [https://bugs.python.org/issue?@action=redirect&bpo=34890]: Make `inspect.iscoroutinefunction()`, `inspect.isgeneratorfunction()` and `inspect.isasyncgenfunction()` work with `functools.partial()`. Patch by Pablo Galindo.

- bpo-34521 [https://bugs.python.org/issue?@action=redirect&bpo=34521]: Use `socket.CMSG_SPACE()` to calculate ancillary data size instead of `socket.CMSG_LEN()` in `multiprocessing.reduction.recvfds()` as **RFC 3542** [https://datatracker.ietf.org/doc/html/rfc3542.html] requires the use of the former for portable applications.

- bpo-31522 [https://bugs.python.org/issue?@action=redirect&bpo=31522]: The `mailbox.mbox.get_string` function *from_* parameter can now successfully be set to a non-default value.

- bpo-34970 [https://bugs.python.org/issue?@action=redirect&bpo=34970]: Protect tasks weak set manipulation in `asyncio.all_tasks()`

- bpo-34969 [https://bugs.python.org/issue?@action=redirect&bpo=34969]: gzip: Add –fast, –best on the gzip CLI, these parameters will be used for the fast compression method (quick) or the best method compress (slower, but smaller file). Also, change the default compression level to 6 (tradeoff).

- bpo-16965 [https://bugs.python.org/issue?@action=redirect&bpo=16965]: The 2to3 `execfile` fixer now opens the file with mode `'rb'`. Patch by Zackery Spytz.

- bpo-34966 [https://bugs.python.org/issue?@action=redirect&bpo=34966]: **`pydoc`** now supports aliases not only to methods defined in the end class, but also to inherited methods. The docstring is not duplicated for aliases.

- bpo-34926 [https://bugs.python.org/issue?@action=redirect&bpo=34926]: **`mimetypes.MimeTypes.guess_type()`** now accepts path-like object in addition to url strings. Patch by Mayank Asthana.

- bpo-23831 [https://bugs.python.org/issue?@action=redirect&bpo=23831]: Add `moveto()` method to the `tkinter.Canvas` widget. Patch by Juliette Monsel.

- bpo-34941 [https://bugs.python.org/issue?@action=redirect&bpo=34941]: Methods `find()`, `findtext()` and `findall()` of the `Element` class in the **`xml.etree.ElementTree`** module are now able to find children which are instances of `Element` subclasses.

- bpo-32680 [https://bugs.python.org/issue?@action=redirect&bpo=32680]: `smtplib.SMTP` objects now always have a `sock` attribute present

- bpo-34769 [https://bugs.python.org/issue?@action=redirect&bpo=34769]: Fix for async generators not finalizing when event loop is in debug mode and garbage collector runs in another thread.

- bpo-34936 [https://bugs.python.org/issue?@action=redirect&bpo=34936]: Fix `TclError` in `tkinter.Spinbox.selection_element()`. Patch by Juliette Monsel.

- bpo-34829 [https://bugs.python.org/issue?@action=redirect&bpo=34829]: Add methods `selection_from`, `selection_range`, `selection_present` and `selection_to` to the `tkinter.Spinbox` for consistency with the `tkinter.Entry` widget. Patch by Juliette Monsel.

- bpo-34911 [https://bugs.python.org/issue?@action=redirect&bpo=34911]: Added *secure_protocols* argument to *http.cookiejar.DefaultCookiePolicy* to allow for tweaking of protocols and also to add support by default for *wss*, the secure websocket protocol.

- bpo-34922 [https://bugs.python.org/issue?@action=redirect&bpo=34922]: Fixed integer overflow in the `digest()` and `hexdigest()` methods for the SHAKE algorithm in the `hashlib` module.

- bpo-34925 [https://bugs.python.org/issue?@action=redirect&bpo=34925]: 25% speedup in argument parsing for the functions in the bisect module.

- bpo-34900 [https://bugs.python.org/issue?@action=redirect&bpo=34900]: Fixed `unittest.TestCase.debug()`

when used to call test methods with subtests. Patch by Bruno Oliveira.

- bpo-34844 [https://bugs.python.org/issue?@action=redirect&bpo=34844]: logging.Formatter enhancement - Ensure styles and fmt matches in logging.Formatter - Added validate method in each format style class: StrFormatStyle, PercentStyle, StringTemplateStyle. - This method is called in the constructor of logging.Formatter class - Also re-raise the KeyError in the format method of each style class, so it would a bit clear that it's an error with the invalid format fields.

- bpo-34897 [https://bugs.python.org/issue?@action=redirect&bpo=34897]: Adjust test.support.missing_compiler_executable check so that a nominal command name of "" is ignored. Patch by Michael Felt.

- bpo-34871 [https://bugs.python.org/issue?@action=redirect&bpo=34871]: Fix inspect module polluted `sys.modules` when parsing `__text_signature__` of callable.

- bpo-34898 [https://bugs.python.org/issue?@action=redirect&bpo=34898]: Add `mtime` argument to `gzip.compress` for reproducible output. Patch by Guo Ci Teo.

- bpo-28441 [https://bugs.python.org/issue?@action=redirect&bpo=28441]: On Cygwin and MinGW, ensure that `sys.executable` always includes the full filename in the path, including the `.exe` suffix (unless it is a symbolic link).

- bpo-34866 [https://bugs.python.org/issue?@action=redirect&bpo=34866]: Adding `max_num_fields` to `cgi.FieldStorage` to make DOS attacks harder by limiting the number of `MiniFieldStorage` objects created by `FieldStorage`.

- bpo-34711 [https://bugs.python.org/issue?@action=redirect&bpo=34711]: http.server ensures it reports HTTPStatus.NOT_FOUND when the local path ends with "/" and is not a directory, even if the underlying OS (e.g. AIX) accepts such paths as a valid file reference. Patch by Michael Felt.

- bpo-34872 [https://bugs.python.org/issue?@action=redirect&bpo=34872]: Fix self-cancellation in C implementation of asyncio.Task

- bpo-34849 [https://bugs.python.org/issue?@action=redirect&bpo=34849]: Don't log waiting for `selector.select` in asyncio loop iteration. The waiting is pretty normal for any asyncio program, logging its time just adds a noise to logs without any useful information provided.

- bpo-34022 [https://bugs.python.org/issue?@action=redirect&bpo=34022]: The **`SOURCE_DATE_EPOCH`** environment variable no longer overrides the value of the *invalidation_mode* argument to **`py_compile.compile()`**, and determines its default value instead.

- bpo-34819 [https://bugs.python.org/issue?@action=redirect&bpo=34819]: Use a monotonic clock to compute timeouts in **`Executor.map()`** and **`as_completed()`**, in order to prevent timeouts from deviating when the system clock is adjusted.

- bpo-34758 [https://bugs.python.org/issue?@action=redirect&bpo=34758]: Add .wasm ->

application/wasm to list of recognized file types and content type headers

- bpo-34789 [https://bugs.python.org/issue?@action=redirect&bpo=34789]: **`xml.sax.make_parser()`** now accepts any iterable as its *parser_list* argument. Patch by Andrés Delfino.

- bpo-34334 [https://bugs.python.org/issue?@action=redirect&bpo=34334]: In **`QueueHandler`**, clear `exc_text` from **`LogRecord`** to prevent traceback from being written twice.

- bpo-34687 [https://bugs.python.org/issue?@action=redirect&bpo=34687]: On Windows, asyncio now uses ProactorEventLoop, instead of SelectorEventLoop, by default.

- bpo-5950 [https://bugs.python.org/issue?@action=redirect&bpo=5950]: Support reading zip files with archive comments in **`zipimport`**.

- bpo-32892 [https://bugs.python.org/issue?@action=redirect&bpo=32892]: The parser now represents all constants as **`ast.Constant`** instead of using specific constant AST types (`Num`, `Str`, `Bytes`, `NameConstant` and `Ellipsis`). These classes are considered deprecated and will be removed in future Python versions.

- bpo-34728 [https://bugs.python.org/issue?@action=redirect&bpo=34728]: Add deprecation warning when `loop` is used in methods: `asyncio.sleep`, `asyncio.wait` and `asyncio.wait_for`.

- bpo-34738 [https://bugs.python.org/issue?@action=redirect&bpo=34738]: ZIP files created by `distutils` will now include entries for directories.

- bpo-34659 [https://bugs.python.org/issue?@action=redirect&bpo=34659]: Add an optional *initial* argument to itertools.accumulate().

- bpo-29577 [https://bugs.python.org/issue?@action=redirect&bpo=29577]: Support multiple mixin classes when creating Enums.

- bpo-34670 [https://bugs.python.org/issue?@action=redirect&bpo=34670]: Add SSLContext.post_handshake_auth and SSLSocket.verify_client_post_handshake for TLS 1.3's post handshake authentication feature.

- bpo-32718 [https://bugs.python.org/issue?@action=redirect&bpo=32718]: The Activate.ps1 script from venv works with PowerShell Core 6.1 and is now available under all operating systems.

- bpo-31177 [https://bugs.python.org/issue?@action=redirect&bpo=31177]: Fix bug that prevented using **`reset_mock`** on mock instances with deleted attributes

- bpo-34672 [https://bugs.python.org/issue?@action=redirect&bpo=34672]: Add a workaround, so the `'Z'` **`time.strftime()`** specifier on the musl C library can work in some cases.

- bpo-34666 [https://bugs.python.org/issue?@action=redirect&bpo=34666]: Implement `asyncio.StreamWriter.awrite` and `asyncio.StreamWriter.aclose()` coroutines. Methods are needed for providing a consistent stream API with control flow switched on by default.

- bpo-6721 [https://bugs.python.org/issue?@action=redirect&bpo=6721]: Acquire the logging module's commonly used internal locks while fork()ing to avoid deadlocks in the child process.

- bpo-34658 [https://bugs.python.org/issue?@action=redirect&bpo=34658]: Fix a rare interpreter unhandled exception state SystemError only seen when using subprocess with a preexec_fn while an after_parent handler has been registered with os.register_at_fork and the fork system call fails.

- bpo-34652 [https://bugs.python.org/issue?@action=redirect&bpo=34652]: Ensure **`os.lchmod()`** is never defined on Linux.

- bpo-34638 [https://bugs.python.org/issue?@action=redirect&bpo=34638]: Store a weak reference to stream reader to break strong references loop between reader and protocol. It allows to detect and close the socket if the stream is deleted (garbage collected) without `close()` call.

- bpo-34536 [https://bugs.python.org/issue?@action=redirect&bpo=34536]: `Enum._missing_`: raise `ValueError` if None returned and `TypeError` if non-member is returned.

- bpo-34636 [https://bugs.python.org/issue?@action=redirect&bpo=34636]: Speed up re scanning of many non-matching characters for s w and d within bytes objects. (microoptimization)

- bpo-24412 [https://bugs.python.org/issue?@action=redirect&bpo=24412]: Add **`addModuleCleanup()`** and **`addClassCleanup()`** to unittest to support cleanups for **`setUpModule()`** and **`setUpClass()`**. Patch by Lisa Roach.

- bpo-34630 [https://bugs.python.org/issue?@action=redirect&bpo=34630]: Don’t log SSL certificate errors in asyncio code (connection error logging is skipped already).

- bpo-32490 [https://bugs.python.org/issue?@action=redirect&bpo=32490]: Prevent filename duplication in **`subprocess`** exception messages. Patch by Zackery Spytz.

- bpo-34363 [https://bugs.python.org/issue?@action=redirect&bpo=34363]: dataclasses.asdict() and .astuple() now handle namedtuples correctly.

- bpo-34625 [https://bugs.python.org/issue?@action=redirect&bpo=34625]: Update vendorized expat library version to 2.2.6.

- bpo-32270 [https://bugs.python.org/issue?@action=redirect&bpo=32270]: The subprocess module no longer mistakenly closes redirected fds even when they were in pass_fds when outside of the default {0, 1, 2} set.

- bpo-34622 [https://bugs.python.org/issue?@action=redirect&bpo=34622]: Create a dedicated `asyncio.CancelledError`, `asyncio.InvalidStateError` and `asyncio.TimeoutError` exception classes. Inherit them from corresponding exceptions from `concurrent.futures` package. Extract `asyncio` exceptions into a separate file.

- bpo-34610 [https://bugs.python.org/issue?@action=redirect&bpo=34610]: Fixed iterator of **`multiprocessing.managers.DictProxy`**.

- bpo-34421 [https://bugs.python.org/issue?@action=redirect&bpo=34421]: Fix distutils logging for non-ASCII strings. This caused installation issues on Windows.

- bpo-34604 [https://bugs.python.org/issue?@action=redirect&bpo=34604]: Fix possible mojibake in the error message of `pwd.getpwnam` and `grp.getgrnam` using string representation because of invisible characters or trailing whitespaces. Patch by William Grzybowski.

- bpo-30977 [https://bugs.python.org/issue?@action=redirect&bpo=30977]: Make uuid.UUID use `__slots__` to reduce its memory footprint. Based on original patch by Wouter Bolsterlee.

- bpo-34574 [https://bugs.python.org/issue?@action=redirect&bpo=34574]: OrderedDict iterators are not exhausted during pickling anymore. Patch by Sergey Fedoseev.

- bpo-8110 [https://bugs.python.org/issue?@action=redirect&bpo=8110]: Refactored **`subprocess`** to check for Windows-specific modules rather than `sys.platform == 'win32'`.

- bpo-34530 [https://bugs.python.org/issue?@action=redirect&bpo=34530]: `distutils.spawn.find_executable()` now falls back on **`os.defpath`** if the `PATH` environment variable is not set.

- bpo-34563 [https://bugs.python.org/issue?@action=redirect&bpo=34563]: On Windows, fix multiprocessing.Connection for very large read: fix _winapi.PeekNamedPipe() and _winapi.ReadFile() for read larger than INT_MAX (usually `2**31-1`).

- bpo-34558 [https://bugs.python.org/issue?@action=redirect&bpo=34558]: Correct typo in Lib/ctypes/_aix.py

- bpo-34282 [https://bugs.python.org/issue?@action=redirect&bpo=34282]: Move `Enum._convert` to

`EnumMeta._convert_` and fix enum members getting shadowed by parent attributes.

- bpo-22872 [https://bugs.python.org/issue?@action=redirect&bpo=22872]: When the queue is closed, `ValueError` is now raised by `multiprocessing.Queue.put()` and `multiprocessing.Queue.get()` instead of `AssertionError` and `OSError`, respectively. Patch by Zackery Spytz.

- bpo-34515 [https://bugs.python.org/issue?@action=redirect&bpo=34515]: Fix parsing non-ASCII identifiers in **`lib2to3.pgen2.tokenize`** (**PEP 3131** [https://peps.python.org/pep-3131/]).

- bpo-13312 [https://bugs.python.org/issue?@action=redirect&bpo=13312]: Avoids a possible integer underflow (undefined behavior) in the time module’s year handling code when passed a very low negative year value.

- bpo-34472 [https://bugs.python.org/issue?@action=redirect&bpo=34472]: Improved compatibility for streamed files in `zipfile`. Previously an optional signature was not being written and certain ZIP applications were not supported. Patch by Silas Sewell.

- bpo-34454 [https://bugs.python.org/issue?@action=redirect&bpo=34454]: Fix the .fromisoformat() methods of datetime types crashing when given unicode with non-UTF-8-encodable code points. Specifically, datetime.fromisoformat() now accepts surrogate unicode code points used as the separator. Report and tests by Alexey Izbyshev, patch by Paul Ganssle.

- bpo-6700 [https://bugs.python.org/issue?@action=redirect&bpo=6700]: Fix inspect.getsourcelines for module level frames/tracebacks. Patch by Vladimir Matveev.

- bpo-34171 [https://bugs.python.org/issue?@action=redirect&bpo=34171]: Running the **`trace`** module no longer creates the `trace.cover` file.

- bpo-34441 [https://bugs.python.org/issue?@action=redirect&bpo=34441]: Fix crash when an `ABC`-derived class with invalid `__subclasses__` is passed as the second argument to **`issubclass()`**. Patch by Alexey Izbyshev.

- bpo-34427 [https://bugs.python.org/issue?@action=redirect&bpo=34427]: Fix infinite loop in `a.extend(a)` for `MutableSequence` subclasses.

- bpo-34412 [https://bugs.python.org/issue?@action=redirect&bpo=34412]: Make **`signal.strsignal()`** work on HP-UX. Patch by Michael Osipov.

- bpo-20849 [https://bugs.python.org/issue?@action=redirect&bpo=20849]: shutil.copytree now accepts a new `dirs_exist_ok` keyword argument. Patch by Josh Bronson.

- bpo-31715 [https://bugs.python.org/issue?@action=redirect&bpo=31715]: Associate `.mjs` file extension with `application/javascript` MIME Type.

- bpo-34384 [https://bugs.python.org/issue?@action=redirect&bpo=34384]: **`os.readlink()`** now accepts path-like and **`bytes`** objects on Windows.

- bpo-22602 [https://bugs.python.org/issue?@action=redirect&bpo=22602]: The UTF-7 decoder now raises **`UnicodeDecodeError`** for ill-formed sequences starting with “+” (as specified in RFC 2152). Patch by Zackery Spytz.

- bpo-2122 [https://bugs.python.org/issue?@action=redirect&bpo=2122]: The **`mmap.flush()`** method now returns `None` on success, raises an exception on error under all platforms.

- bpo-34341 [https://bugs.python.org/issue?@action=redirect&bpo=34341]: Appending to the ZIP archive with the ZIP64 extension no longer grows the size of extra fields of existing entries.

- bpo-34333 [https://bugs.python.org/issue?@action=redirect&bpo=34333]: Fix %-formatting in `pathlib.PurePath.with_suffix()` when formatting an error message.

- bpo-18540 [https://bugs.python.org/issue?@action=redirect&bpo=18540]: The `imaplib.IMAP4` and `imaplib.IMAP4_SSL` classes now resolve to the local host IP correctly when the default value of *host* parameter (`''`) is used.

- bpo-26502 [https://bugs.python.org/issue?@action=redirect&bpo=26502]: Implement `traceback.FrameSummary.__len__()` method to preserve compatibility with the old tuple API.

- bpo-34318 [https://bugs.python.org/issue?@action=redirect&bpo=34318]: `assertRaises()`, `assertRaisesRegex()`, `assertWarns()` and `assertWarnsRegex()` no longer success if the passed callable is None. They no longer ignore unknown keyword arguments in the context manager mode. A DeprecationWarning was raised in these cases since Python 3.5.

- bpo-9372 [https://bugs.python.org/issue?@action=redirect&bpo=9372]: Deprecate `__getitem__()` methods of `xml.dom.pulldom.DOMEventStream`, `wsgiref.util.FileWrapper` and `fileinput.FileInput`.

- bpo-33613 [https://bugs.python.org/issue?@action=redirect&bpo=33613]: Fix a race condition in `multiprocessing.semaphore_tracker` when the tracker receives SIGINT before it can register signal handlers for ignoring it.

- bpo-34248 [https://bugs.python.org/issue?@action=redirect&bpo=34248]: Report filename in the exception raised when the database file cannot be opened by **`dbm.gnu.open()`** and **`dbm.ndbm.open()`** due to OS-related error. Patch by Zsolt Cserna.

- bpo-33089 [https://bugs.python.org/issue?@action=redirect&bpo=33089]: Add math.dist() to compute the Euclidean distance between two points.

- bpo-34246 [https://bugs.python.org/issue?@action=redirect&bpo=34246]: **`smtplib.SMTP.send_message()`** no longer modifies the content of the *mail_options* argument. Patch by Pablo S. Blum de Aguiar.

- bpo-31047 [https://bugs.python.org/issue?@action=redirect&bpo=31047]: Fix `ntpath.abspath` for invalid paths on windows. Patch by Franz Woellert.

- bpo-32321 [https://bugs.python.org/issue?@action=redirect&bpo=32321]: Add pure Python fallback for functools.reduce. Patch by Robert Wright.

- bpo-34270 [https://bugs.python.org/issue?@action=redirect&bpo=34270]: The default asyncio task class now always has a name which can be get or set using two new methods (**`get_name()`** and **`set_name()`**) and is visible in the **`repr()`** output. An initial name can also be set using the new `name` keyword argument to **`asyncio.create_task()`** or the **`create_task()`** method of the event loop. If no initial name is set, the default Task implementation generates a name like `Task-1` using a monotonic counter.

- bpo-34263 [https://bugs.python.org/issue?@action=redirect&bpo=34263]: asyncio's event loop will not pass timeouts longer than one day to epoll/select etc.
- bpo-34035 [https://bugs.python.org/issue?@action=redirect&bpo=34035]: Fix several AttributeError in zipfile seek() methods. Patch by Mickaël Schoentgen.
- bpo-32215 [https://bugs.python.org/issue?@action=redirect&bpo=32215]: Fix performance regression in **`sqlite3`** when a DML statement appeared in a different line than the rest of the SQL query.
- bpo-34075 [https://bugs.python.org/issue?@action=redirect&bpo=34075]: Deprecate passing non-ThreadPoolExecutor instances to **`AbstractEventLoop.set_default_executor()`**.
- bpo-34251 [https://bugs.python.org/issue?@action=redirect&bpo=34251]: Restore `msilib.Win64` to preserve backwards compatibility since it's already used by `distutils`' `bdist_msi` command.
- bpo-19891 [https://bugs.python.org/issue?@action=redirect&bpo=19891]: Ignore errors caused by missing / non-writable homedir while writing history during exit of an interactive session. Patch by Anthony Sottile.
- bpo-33089 [https://bugs.python.org/issue?@action=redirect&bpo=33089]: Enhanced math.hypot() to support more than two dimensions.

- bpo-34228 [https://bugs.python.org/issue?@action=redirect&bpo=34228]: tracemalloc: PYTHONTRACEMALLOC=0 environment variable and -X tracemalloc=0 command line option are now allowed to disable explicitly tracemalloc at startup.

- bpo-13041 [https://bugs.python.org/issue?@action=redirect&bpo=13041]: Use **`shutil.get_terminal_size()`** to calculate the terminal width correctly in the `argparse.HelpFormatter` class. Initial patch by Zbyszek Jędrzejewski-Szmek.

- bpo-34213 [https://bugs.python.org/issue?@action=redirect&bpo=34213]: Allow frozen dataclasses to have a field named “object”. Previously this conflicted with an internal use of “object”.

- bpo-34052 [https://bugs.python.org/issue?@action=redirect&bpo=34052]: **`sqlite3.Connection.create_aggregate()`**, **`sqlite3.Connection.create_function()`**, **`sqlite3.Connection.set_authorizer()`**, **`sqlite3.Connection.set_progress_handler()`** methods raises TypeError when unhashable objects are passed as callable. These methods now don’t pass such objects to SQLite API. Previous behavior could lead to segfaults. Patch by Sergey Fedoseev.

- bpo-34197 [https://bugs.python.org/issue?@action=redirect&bpo=34197]: Attributes *skipinitialspace*, *doublequote* and *strict* of the *dialect* attribute of the **`csv`** reader are now **`bool`** instances instead of integers 0 or 1.

- bpo-32788 [https://bugs.python.org/issue?@action=redirect&bpo=32788]: Errors other than **`TypeError`** raised in methods `__adapt__()` and `__conform__()` in the **`sqlite3`** module are now propagated to the user.

- bpo-21446 [https://bugs.python.org/issue?@action=redirect&bpo=21446]: The `reload` fixer now uses **`importlib.reload()`** instead of deprecated **`imp.reload()`**.

- bpo-940286 [https://bugs.python.org/issue?@action=redirect&bpo=940286]: pydoc's `Helper.showtopic()` method now prints the cross references of a topic correctly.

- bpo-34164 [https://bugs.python.org/issue?@action=redirect&bpo=34164]: **`base64.b32decode()`** could raise UnboundLocalError or OverflowError for incorrect padding. Now it always raises **`base64.Error`** in these cases.

- bpo-33729 [https://bugs.python.org/issue?@action=redirect&bpo=33729]: Fixed issues with arguments parsing in **`hashlib`**.

- bpo-34097 [https://bugs.python.org/issue?@action=redirect&bpo=34097]: ZipFile can zip files older than 1980-01-01 and newer than 2107-12-31 using a new `strict_timestamps` parameter at the cost of setting the timestamp to the limit.

- bpo-34108 [https://bugs.python.org/issue?@action=redirect&bpo=34108]: Remove extraneous CR in 2to3 refactor.

- bpo-34070 [https://bugs.python.org/issue?@action=redirect&bpo=34070]: Make sure to only check if the handle is a tty, when opening a file with `buffering=-1`.

- bpo-27494 [https://bugs.python.org/issue?@action=redirect&bpo=27494]: Reverted bpo-27494 [https://bugs.python.org/issue?@action=redirect&bpo=27494]. 2to3 rejects now a trailing comma in generator expressions.
- bpo-33967 [https://bugs.python.org/issue?@action=redirect&bpo=33967]: functools.singledispatch now raises TypeError instead of IndexError when no positional arguments are passed.
- bpo-34041 [https://bugs.python.org/issue?@action=redirect&bpo=34041]: Add the parameter *deterministic* to the **`sqlite3.Connection.create_function()`** method. Patch by Sergey Fedoseev.
- bpo-34056 [https://bugs.python.org/issue?@action=redirect&bpo=34056]: Ensure the loader shim created by `imp.load_module` always returns bytes from its `get_data()` function. This fixes using `imp.load_module` with **PEP 552** [https://peps.python.org/pep-0552/] hash-based pycs.
- bpo-34054 [https://bugs.python.org/issue?@action=redirect&bpo=34054]: The multiprocessing module now uses the monotonic clock **`time.monotonic()`** instead of the system clock **`time.time()`** to implement timeout.
- bpo-34043 [https://bugs.python.org/issue?@action=redirect&bpo=34043]: Optimize tarfile uncompress performance about 15% when gzip is used.
- bpo-34044 [https://bugs.python.org/issue?@action=redirect&bpo=34044]: `subprocess.Popen` now copies the *startupinfo* argument to leave it unchanged: it will modify the copy, so that the same `STARTUPINFO` object can be used multiple times.

- bpo-34010 [https://bugs.python.org/issue?@action=redirect&bpo=34010]: Fixed a performance regression for reading streams with tarfile. The buffered read should use a list, instead of appending to a bytes object.

- bpo-34019 [https://bugs.python.org/issue?@action=redirect&bpo=34019]: webbrowser: Correct the arguments passed to Opera Browser when opening a new URL using the `webbrowser` module. Patch by Bumsik Kim.

- bpo-34003 [https://bugs.python.org/issue?@action=redirect&bpo=34003]: csv.DictReader now creates dicts instead of OrderedDicts. Patch by Michael Selik.

- bpo-33978 [https://bugs.python.org/issue?@action=redirect&bpo=33978]: Closed existing logging handlers before reconfiguration via fileConfig and dictConfig. Patch by Karthikeyan Singaravelan.

- bpo-14117 [https://bugs.python.org/issue?@action=redirect&bpo=14117]: Make minor tweaks to turtledemo. The ‘wikipedia’ example is now ‘rosette’, describing what it draws. The ‘penrose’ print output is reduced. The’1024’ output of ‘tree’ is eliminated.

- bpo-33974 [https://bugs.python.org/issue?@action=redirect&bpo=33974]: Fixed passing lists and tuples of strings containing special characters `"`, `\`, `{`, `}` and `\n` as options to **`ttk`** widgets.

- bpo-27500 [https://bugs.python.org/issue?@action=redirect&bpo=27500]: Fix getaddrinfo to resolve IPv6 addresses correctly.

- bpo-24567 [https://bugs.python.org/issue?@action=redirect&bpo=24567]: Improve random.choices() to handle subnormal input weights that could occasionally trigger an IndexError.

- bpo-33871 [https://bugs.python.org/issue?@action=redirect&bpo=33871]: Fixed integer overflow in `os.readv()`, `os.writev()`, `os.preadv()` and `os.pwritev()` and in `os.sendfile()` with *headers* or *trailers* arguments (on BSD-based OSes and macOS).

- bpo-25007 [https://bugs.python.org/issue?@action=redirect&bpo=25007]: Add `copy.copy()` and `copy.deepcopy()` support to zlib compressors and decompressors. Patch by Zackery Spytz.

- bpo-33929 [https://bugs.python.org/issue?@action=redirect&bpo=33929]: multiprocessing: Fix a race condition in Popen of multiprocessing.popen_spawn_win32. The child process now duplicates the read end of pipe instead of “stealing” it. Previously, the read end of pipe was “stolen” by the child process, but it leaked a handle if the child process had been terminated before it could steal the handle from the parent process.

- bpo-33899 [https://bugs.python.org/issue?@action=redirect&bpo=33899]: Tokenize module now implicitly emits a NEWLINE when provided with input that does not have a trailing new line. This behavior now matches what the C tokenizer does internally. Contributed by Ammar Askar.

- bpo-33897 [https://bugs.python.org/issue?@action=redirect&bpo=33897]: Added a ‘force’ keyword argument to logging.basicConfig().

- bpo-33695 [https://bugs.python.org/issue?@action=redirect&bpo=33695]: `shutil.copytree()` uses `os.scandir()` function and all copy functions depending from it use cached `os.stat()` values. The speedup for copying a directory with 8000 files is around +9% on Linux, +20% on Windows and + 30% on a Windows SMB share. Also the number of `os.stat()` syscalls is reduced by 38% making `shutil.copytree()` especially faster on network filesystems. (Contributed by Giampaolo Rodola' in bpo-33695 [https://bugs.python.org/issue?@action=redirect&bpo=33695].)

- bpo-33916 [https://bugs.python.org/issue?@action=redirect&bpo=33916]: bz2 and lzma: When Decompressor.__init__() is called twice, free the old lock to not leak memory.
- bpo-32568 [https://bugs.python.org/issue?@action=redirect&bpo=32568]: Make select.epoll() and its documentation consistent regarding *sizehint* and *flags*.
- bpo-33833 [https://bugs.python.org/issue?@action=redirect&bpo=33833]: Fixed bug in asyncio where ProactorSocketTransport logs AssertionError if force closed during write.
- bpo-33663 [https://bugs.python.org/issue?@action=redirect&bpo=33663]: Convert content length to string before putting to header.
- bpo-33721 [https://bugs.python.org/issue?@action=redirect&bpo=33721]: `os.path` functions that return a boolean result like `exists()`, `lexists()`, `isdir()`, `isfile()`, `islink()`, and `ismount()`, and `pathlib.Path` methods that return a boolean result like `exists()`, `is_dir()`, `is_file()`, `is_mount()`, `is_symlink()`, `is_block_device()`, `is_char_device()`, `is_fifo()`, `is_socket()` now return `False` instead of raising `ValueError` or its subclasses

`UnicodeEncodeError` and `UnicodeDecodeError` for paths that contain characters or bytes unrepresentable at the OS level.

- bpo-26544 [https://bugs.python.org/issue?@action=redirect&bpo=26544]: Fixed implementation of `platform.libc_ver()`. It almost always returned version ‘2.9’ for glibc.
- bpo-33843 [https://bugs.python.org/issue?@action=redirect&bpo=33843]: Remove deprecated `cgi.escape`, `cgi.parse_qs` and `cgi.parse_qsl`.
- bpo-33842 [https://bugs.python.org/issue?@action=redirect&bpo=33842]: Remove `tarfile.filemode` which is deprecated since Python 3.3.
- bpo-30167 [https://bugs.python.org/issue?@action=redirect&bpo=30167]: Prevent site.main() exception if PYTHONSTARTUP is set. Patch by Steve Weber.
- bpo-33805 [https://bugs.python.org/issue?@action=redirect&bpo=33805]: Improve error message of dataclasses.replace() when an InitVar is not specified
- bpo-33687 [https://bugs.python.org/issue?@action=redirect&bpo=33687]: Fix the call to `os.chmod()` for `uu.decode()` if a mode is given or decoded. Patch by Timo Furrer.
- bpo-33812 [https://bugs.python.org/issue?@action=redirect&bpo=33812]: Datetime instance d with non-None tzinfo, but with d.tzinfo.utcoffset(d) returning None is now treated as naive by the astimezone() method.
- bpo-32108 [https://bugs.python.org/issue?@action=redirect&bpo=32108]: In configparser, don’t clear

section when it is assigned to itself.

- bpo-27397 [https://bugs.python.org/issue?@action=redirect&bpo=27397]: Make email module properly handle invalid-length base64 strings.
- bpo-33578 [https://bugs.python.org/issue?@action=redirect&bpo=33578]: Implement multibyte encoder/decoder state methods
- bpo-30805 [https://bugs.python.org/issue?@action=redirect&bpo=30805]: Avoid race condition with debug logging
- bpo-33476 [https://bugs.python.org/issue?@action=redirect&bpo=33476]: Fix _header_value_parser.py when address group is missing final ‘;’. Contributed by Enrique Perez-Terron
- bpo-33694 [https://bugs.python.org/issue?@action=redirect&bpo=33694]: asyncio: Fix a race condition causing data loss on pause_reading()/resume_reading() when using the ProactorEventLoop.
- bpo-32493 [https://bugs.python.org/issue?@action=redirect&bpo=32493]: Correct test for `uuid_enc_be` availability in `configure.ac`. Patch by Michael Felt.
- bpo-33792 [https://bugs.python.org/issue?@action=redirect&bpo=33792]: Add asyncio.WindowsSelectorEventLoopPolicy and asyncio.WindowsProactorEventLoopPolicy.
- bpo-33274 [https://bugs.python.org/issue?@action=redirect&bpo=33274]: W3C DOM Level 1 specifies return value of Element.removeAttributeNode() as “The

Attr node that was removed.” xml.dom.minidom now complies with this requirement.

- bpo-33778 [https://bugs.python.org/issue?@action=redirect&bpo=33778]: Update `unicodedata`’s database to Unicode version 11.0.0.
- bpo-33165 [https://bugs.python.org/issue?@action=redirect&bpo=33165]: Added a stacklevel parameter to logging calls to allow use of wrapper/helper functions for logging APIs.
- bpo-33770 [https://bugs.python.org/issue?@action=redirect&bpo=33770]: improve base64 exception message for encoded inputs of invalid length
- bpo-33769 [https://bugs.python.org/issue?@action=redirect&bpo=33769]: asyncio/start_tls: Fix error message; cancel callbacks in case of an unhandled error; mark SSLTransport as closed if it is aborted.
- bpo-33767 [https://bugs.python.org/issue?@action=redirect&bpo=33767]: The concatenation (+) and repetition (*) sequence operations now raise **`TypeError`** instead of **`SystemError`** when performed on **`mmap.mmap`** objects. Patch by Zackery Spytz.
- bpo-33734 [https://bugs.python.org/issue?@action=redirect&bpo=33734]: asyncio/ssl: Fix AttributeError, increase default handshake timeout
- bpo-31014 [https://bugs.python.org/issue?@action=redirect&bpo=31014]: Fixed creating a controller for **`webbrowser`** when a user specifies a path to an entry in the BROWSER environment variable. Based on patch by John Still.

- bpo-2504 [https://bugs.python.org/issue?@action=redirect&bpo=2504]: Add gettext.pgettext() and variants.
- bpo-33197 [https://bugs.python.org/issue?@action=redirect&bpo=33197]: Add description property for _ParameterKind
- bpo-32751 [https://bugs.python.org/issue?@action=redirect&bpo=32751]: When cancelling the task due to a timeout, **`asyncio.wait_for()`** will now wait until the cancellation is complete.
- bpo-32684 [https://bugs.python.org/issue?@action=redirect&bpo=32684]: Fix gather to propagate cancellation of itself even with return_exceptions.
- bpo-33654 [https://bugs.python.org/issue?@action=redirect&bpo=33654]: Support protocol type switching in SSLTransport.set_protocol().
- bpo-33674 [https://bugs.python.org/issue?@action=redirect&bpo=33674]: Pause the transport as early as possible to further reduce the risk of data_received() being called before connection_made().
- bpo-33671 [https://bugs.python.org/issue?@action=redirect&bpo=33671]: **`shutil.copyfile()`**, **`shutil.copy()`**, **`shutil.copy2()`**, **`shutil.copytree()`** and **`shutil.move()`** use platform-specific fast-copy syscalls on Linux and macOS in order to copy the file more efficiently. On Windows **`shutil.copyfile()`** uses a bigger default buffer size (1 MiB instead of 16 KiB) and a **`memoryview()`**-based variant of **`shutil.copyfileobj()`** is used. The speedup for copying a 512MiB file is about +26% on Linux, +50% on macOS and +40% on Windows. Also, much less CPU cycles are consumed. (Contributed by Giampaolo Rodola’ in bpo-25427 [https://bugs.python.org/issue?@action=redirect&bpo=25427].)

- bpo-33674 [https://bugs.python.org/issue?@action=redirect&bpo=33674]: Fix a race condition in SSLProtocol.connection_made() of asyncio.sslproto: start immediately the handshake instead of using call_soon(). Previously, data_received() could be called before the handshake started, causing the handshake to hang or fail.

- bpo-31647 [https://bugs.python.org/issue?@action=redirect&bpo=31647]: Fixed bug where calling write_eof() on a _SelectorSocketTransport after it's already closed raises AttributeError.

- bpo-32610 [https://bugs.python.org/issue?@action=redirect&bpo=32610]: Make asyncio.all_tasks() return only pending tasks.

- bpo-32410 [https://bugs.python.org/issue?@action=redirect&bpo=32410]: Avoid blocking on file IO in sendfile fallback code

- bpo-33469 [https://bugs.python.org/issue?@action=redirect&bpo=33469]: Fix RuntimeError after closing loop that used run_in_executor

- bpo-33672 [https://bugs.python.org/issue?@action=redirect&bpo=33672]: Fix Task.__repr__ crash with Cython's bogus coroutines

- bpo-33654 [https://bugs.python.org/issue?@action=redirect&bpo=33654]: Fix transport.set_protocol() to support switching between asyncio.Protocol and asyncio.BufferedProtocol. Fix loop.start_tls() to work with asyncio.BufferedProtocols.

- bpo-33652 [https://bugs.python.org/issue?@action=redirect&bpo=33652]: Pickles of type variables and subscripted generics are now future-proof and compatible with older Python versions.

- bpo-32493 [https://bugs.python.org/issue?@action=redirect&bpo=32493]: Fixed `uuid.uuid1()` on FreeBSD.

- bpo-33238 [https://bugs.python.org/issue?@action=redirect&bpo=33238]: Add `InvalidStateError` to `concurrent.futures`. `Future.set_result` and `Future.set_exception` now raise `InvalidStateError` if the futures are not pending or running. Patch by Jason Haydaman.

- bpo-33618 [https://bugs.python.org/issue?@action=redirect&bpo=33618]: Finalize and document preliminary and experimental TLS 1.3 support with OpenSSL 1.1.1

- bpo-33625 [https://bugs.python.org/issue?@action=redirect&bpo=33625]: Release GIL on `grp.getgrnam`, `grp.getgrgid`, `pwd.getpwnam` and `pwd.getpwuid` if reentrant variants of these functions are available. Patch by William Grzybowski.

- bpo-33623 [https://bugs.python.org/issue?@action=redirect&bpo=33623]: Fix possible SIGSGV when asyncio.Future is created in __del__

- bpo-11874 [https://bugs.python.org/issue?@action=redirect&bpo=11874]: Use a better regex when breaking usage into wrappable parts. Avoids bogus assertion errors from custom metavar strings.

- bpo-30877 [https://bugs.python.org/issue?@action=redirect&bpo=30877]: Fixed a bug in the Python implementation of the JSON decoder that prevented the cache of parsed strings from clearing after finishing the decoding. Based on patch by c-fos.

- bpo-33604 [https://bugs.python.org/issue?@action=redirect&bpo=33604]: Remove HMAC default to md5 marked for removal in 3.8 (removal originally planned in 3.6, bump to 3.8 in PR 7062).
- bpo-33582 [https://bugs.python.org/issue?@action=redirect&bpo=33582]: Emit a deprecation warning for inspect.formatargspec
- bpo-21145 [https://bugs.python.org/issue?@action=redirect&bpo=21145]: Add `functools.cached_property` decorator, for computed properties cached for the life of the instance.
- bpo-33570 [https://bugs.python.org/issue?@action=redirect&bpo=33570]: Change TLS 1.3 cipher suite settings for compatibility with OpenSSL 1.1.1-pre6 and newer. OpenSSL 1.1.1 will have TLS 1.3 ciphers enabled by default.
- bpo-28556 [https://bugs.python.org/issue?@action=redirect&bpo=28556]: Do not simplify arguments to `typing.Union`. Now `Union[Manager, Employee]` is not simplified to `Employee` at runtime. Such simplification previously caused several bugs and limited possibilities for introspection.
- bpo-12486 [https://bugs.python.org/issue?@action=redirect&bpo=12486]: **`tokenize.generate_tokens()`** is now documented as a public API to tokenize unicode strings. It was previously present but undocumented.
- bpo-33540 [https://bugs.python.org/issue?@action=redirect&bpo=33540]: Add a new `block_on_close` class

attribute to `ForkingMixIn` and `ThreadingMixIn` classes of **`socketserver`**.

- bpo-33548 [https://bugs.python.org/issue?@action=redirect&bpo=33548]: tempfile._candidate_tempdir_list should consider common TEMP locations
- bpo-33109 [https://bugs.python.org/issue?@action=redirect&bpo=33109]: argparse subparsers are once again not required by default, reverting the change in behavior introduced by bpo-26510 [https://bugs.python.org/issue?@action=redirect&bpo=26510] in 3.7.0a2.
- bpo-33541 [https://bugs.python.org/issue?@action=redirect&bpo=33541]: Remove unused private method `_strptime.LocaleTime.__pad` (a.k.a. `_LocaleTime__pad`).
- bpo-33536 [https://bugs.python.org/issue?@action=redirect&bpo=33536]: dataclasses.make_dataclass now checks for invalid field names and duplicate fields. Also, added a check for invalid field specifications.
- bpo-33542 [https://bugs.python.org/issue?@action=redirect&bpo=33542]: Prevent `uuid.get_node` from using a DUID instead of a MAC on Windows. Patch by Zvi Effron
- bpo-26819 [https://bugs.python.org/issue?@action=redirect&bpo=26819]: Fix race condition with `ReadTransport.resume_reading` in Windows proactor event loop.
- Fix failure in `typing.get_type_hints()` when ClassVar was provided as a string forward reference.

- bpo-33516 [https://bugs.python.org/issue?@action=redirect&bpo=33516]: **`unittest.mock.MagicMock`** now supports the `__round__` magic method.

- bpo-28612 [https://bugs.python.org/issue?@action=redirect&bpo=28612]: Added support for Site Maps to urllib’s `RobotFileParser` as **`RobotFileParser.site_maps()`**. Patch by Lady Red, based on patch by Peter Wirtz.

- bpo-28167 [https://bugs.python.org/issue?@action=redirect&bpo=28167]: Remove platform.linux_distribution, which was deprecated since 3.5.

- bpo-33504 [https://bugs.python.org/issue?@action=redirect&bpo=33504]: Switch the default dictionary implementation for **`configparser`** from **`collections.OrderedDict`** to the standard **`dict`** type.

- bpo-33505 [https://bugs.python.org/issue?@action=redirect&bpo=33505]: Optimize asyncio.ensure_future() by reordering if checks: 1.17x faster.

- bpo-33497 [https://bugs.python.org/issue?@action=redirect&bpo=33497]: Add errors param to cgi.parse_multipart and make an encoding in FieldStorage use the given errors (needed for Twisted). Patch by Amber Brown.

- bpo-29235 [https://bugs.python.org/issue?@action=redirect&bpo=29235]: The **`cProfile.Profile`** class can now be used as a context manager. Patch by Scott Sanderson.

- bpo-33495 [https://bugs.python.org/issue?@action=redirect&bpo=33495]: Change dataclasses.Fields repr to use the repr of each of its members, instead of str. This makes it more clear what each field actually represents. This is especially true for the ‘type’ member.

- bpo-26103 [https://bugs.python.org/issue?@action=redirect&bpo=26103]: Correct `inspect.isdatadescriptor` to look for `__set__` or `__delete__`. Patch by Aaron Hall.

- bpo-29209 [https://bugs.python.org/issue?@action=redirect&bpo=29209]: Removed the `doctype()` method and the *html* parameter of the constructor of **`XMLParser`**. The `doctype()` method defined in a subclass will no longer be called. Deprecated methods `getchildren()` and `getiterator()` in the **`ElementTree`** module emit now a **`DeprecationWarning`** instead of **`PendingDeprecationWarning`**.

- bpo-33453 [https://bugs.python.org/issue?@action=redirect&bpo=33453]: Fix dataclasses to work if using literal string type annotations or if using PEP 563 “Postponed Evaluation of Annotations”. Only specific string prefixes are detected for both ClassVar (“ClassVar” and “typing.ClassVar”) and InitVar (“InitVar” and “dataclasses.InitVar”).

- bpo-28556 [https://bugs.python.org/issue?@action=redirect&bpo=28556]: Minor fixes in typing module: add annotations to `NamedTuple.__new__`, pass `*args` and `**kwds` in `Generic.__new__`. Original PRs by Paulius Šarka and Chad Dombrova.

- bpo-33365 [https://bugs.python.org/issue?@action=redirect&bpo=33365]: Print the header values besides the header keys instead just the header keys if *debuglevel* is set to >0 in **`http.client`**. Patch by Marco Strigl.

- bpo-20087 [https://bugs.python.org/issue?@action=redirect&bpo=20087]: Updated alias mapping with glibc 2.27 supported locales.

- bpo-33422 [https://bugs.python.org/issue?@action=redirect&bpo=33422]: Fix trailing quotation marks getting deleted when looking up byte/string literals on pydoc. Patch by Andrés Delfino.

- bpo-28167 [https://bugs.python.org/issue?@action=redirect&bpo=28167]: The function `platform.linux_distribution` and `platform.dist` now trigger a `DeprecationWarning` and have been marked for removal in Python 3.8

- bpo-33281 [https://bugs.python.org/issue?@action=redirect&bpo=33281]: Fix ctypes.util.find_library regression on macOS.

- bpo-33311 [https://bugs.python.org/issue?@action=redirect&bpo=33311]: Text and html output generated by cgitb does not display parentheses if the current call is done directly in the module. Patch by Stéphane Blondon.

- bpo-27300 [https://bugs.python.org/issue?@action=redirect&bpo=27300]: The file classes in *tempfile* now accept an *errors* parameter that complements the already existing *encoding*. Patch by Stephan Hohe.

- bpo-32933 [https://bugs.python.org/issue?@action=redirect&bpo=32933]: **`unittest.mock.mock_open()`** now supports iteration over the file contents. Patch by Tony Flury.

- bpo-33217 [https://bugs.python.org/issue?@action=redirect&bpo=33217]: Raise **`TypeError`** when looking up non-Enum objects in Enum classes and Enum members.

- bpo-33197 [https://bugs.python.org/issue?@action=redirect&bpo=33197]: Update error message when constructing invalid inspect.Parameters Patch by Donghee Na.

- bpo-33383 [https://bugs.python.org/issue?@action=redirect&bpo=33383]: Fixed crash in the get() method of the **`dbm.ndbm`** database object when it is called with a single argument.

- bpo-33375 [https://bugs.python.org/issue?@action=redirect&bpo=33375]: The warnings module now finds the Python file associated with a warning from the code object, rather than the frame's global namespace. This is consistent with how tracebacks and pdb find filenames, and should work better for dynamically executed code.

- bpo-33336 [https://bugs.python.org/issue?@action=redirect&bpo=33336]: `imaplib` now allows `MOVE` command in `IMAP4.uid()` (RFC 6851: IMAP MOVE Extension) and potentially as a name of supported method of `IMAP4` object.

- bpo-32455 [https://bugs.python.org/issue?@action=redirect&bpo=32455]: Added *jump* parameter to **`dis.stack_effect()`**.

- bpo-27485 [https://bugs.python.org/issue?@action=redirect&bpo=27485]: Rename and deprecate undocumented functions in **`urllib.parse()`**.

- bpo-33332 [https://bugs.python.org/issue?@action=redirect&bpo=33332]: Add `signal.valid_signals()` to expose the POSIX sigfillset() functionality.

- bpo-33251 [https://bugs.python.org/issue?@action=redirect&bpo=33251]: `ConfigParser.items()` was fixed so that key-value pairs passed in via `vars()` are not included in the resulting output.

- bpo-33329 [https://bugs.python.org/issue?@action=redirect&bpo=33329]: Fix multiprocessing regression on newer glibcs

- bpo-33334 [https://bugs.python.org/issue?@action=redirect&bpo=33334]: `dis.stack_effect()` now supports all defined opcodes including NOP and EXTENDED_ARG.

- bpo-991266 [https://bugs.python.org/issue?@action=redirect&bpo=991266]: Fix quoting of the `Comment` attribute of `http.cookies.SimpleCookie`.

- bpo-33131 [https://bugs.python.org/issue?@action=redirect&bpo=33131]: Upgrade bundled version of pip to 10.0.1.

- bpo-33308 [https://bugs.python.org/issue?@action=redirect&bpo=33308]: Fixed a crash in the **`parser`** module when converting an ST object to a tree of tuples or lists with `line_info=False` and `col_info=True`.

- bpo-23403 [https://bugs.python.org/issue?@action=redirect&bpo=23403]: lib2to3 now uses pickle protocol 4 for pre-computed grammars.

- bpo-33266 [https://bugs.python.org/issue?@action=redirect&bpo=33266]: lib2to3 now recognizes `rf'...'` strings.

- bpo-11594 [https://bugs.python.org/issue?@action=redirect&bpo=11594]: Ensure line-endings are respected when using lib2to3.

- bpo-33254 [https://bugs.python.org/issue?@action=redirect&bpo=33254]: Have **`importlib.resources.contents()`** and **`importlib.abc.ResourceReader.contents()`** return an iterable instead of an iterator.
- bpo-33265 [https://bugs.python.org/issue?@action=redirect&bpo=33265]: `contextlib.ExitStack` and `contextlib.AsyncExitStack` now use a method instead of a wrapper function for exit callbacks.
- bpo-33263 [https://bugs.python.org/issue?@action=redirect&bpo=33263]: Fix FD leak in `_SelectorSocketTransport` Patch by Vlad Starostin.
- bpo-33256 [https://bugs.python.org/issue?@action=redirect&bpo=33256]: Fix display of `<module>` call in the html produced by `cgitb.html()`. Patch by Stéphane Blondon.
- bpo-33144 [https://bugs.python.org/issue?@action=redirect&bpo=33144]: `random.Random()` and its subclassing mechanism got optimized to check only once at class/subclass instantiation time whether its `getrandbits()` method can be relied on by other methods, including `randrange()`, for the generation of arbitrarily large random integers. Patch by Wolfgang Maier.
- bpo-33185 [https://bugs.python.org/issue?@action=redirect&bpo=33185]: Fixed regression when running pydoc with the **`-m`** switch. (The regression was introduced in 3.7.0b3 by the resolution of bpo-33053 [https://bugs.python.org/issue?@action=redirect&bpo=33053])

  This fix also changed pydoc to add `os.getcwd()` to **`sys.path`** when necessary, rather than adding `"."`.

- bpo-29613 [https://bugs.python.org/issue?@action=redirect&bpo=29613]: Added support for the `SameSite` cookie flag to the `http.cookies` module.

- bpo-33169 [https://bugs.python.org/issue?@action=redirect&bpo=33169]: Delete entries of `None` in **`sys.path_importer_cache`** when **`importlib.machinery.invalidate_caches()`** is called.

- bpo-33203 [https://bugs.python.org/issue?@action=redirect&bpo=33203]: `random.Random.choice()` now raises `IndexError` for empty sequences consistently even when called from subclasses without a `getrandbits()` implementation.

- bpo-33224 [https://bugs.python.org/issue?@action=redirect&bpo=33224]: Update difflib.mdiff() for **PEP 479** [https://peps.python.org/pep-0479/]. Convert an uncaught StopIteration in a generator into a return-statement.

- bpo-33209 [https://bugs.python.org/issue?@action=redirect&bpo=33209]: End framing at the end of C implementation of **`pickle.Pickler.dump()`**.

- bpo-32861 [https://bugs.python.org/issue?@action=redirect&bpo=32861]: The urllib.robotparser’s `__str__` representation now includes wildcard entries and the “Crawl-delay” and “Request-rate” fields. Also removes extra newlines that were being appended to the end of the string. Patch by Michael Lazar.

- bpo-23403 [https://bugs.python.org/issue?@action=redirect&bpo=23403]: `DEFAULT_PROTOCOL` in **`pickle`** was bumped to 4. Protocol 4 is described in **PEP 3154** [https://peps.python.org/pep-3154/] and available since Python 3.4. It offers better performance and smaller size compared to protocol 3 introduced in Python 3.0.

- bpo-20104 [https://bugs.python.org/issue?@action=redirect&bpo=20104]: Improved error handling and fixed a reference leak in `os.posix_spawn()`.
- bpo-33106 [https://bugs.python.org/issue?@action=redirect&bpo=33106]: Deleting a key from a read-only dbm database raises module specific error instead of KeyError.
- bpo-33175 [https://bugs.python.org/issue?@action=redirect&bpo=33175]: In dataclasses, Field.__set_name__ now looks up the __set_name__ special method on the class, not the instance, of the default value.
- bpo-32380 [https://bugs.python.org/issue?@action=redirect&bpo=32380]: Create functools.singledispatchmethod to support generic single dispatch on descriptors and methods.
- bpo-33141 [https://bugs.python.org/issue?@action=redirect&bpo=33141]: Have Field objects pass through __set_name__ to their default values, if they have their own __set_name__.
- bpo-33096 [https://bugs.python.org/issue?@action=redirect&bpo=33096]: Allow ttk.Treeview.insert to insert iid that has a false boolean value. Note iid=0 and iid=False would be same. Patch by Garvit Khatri.
- bpo-32873 [https://bugs.python.org/issue?@action=redirect&bpo=32873]: Treat type variables and special typing forms as immutable by copy and pickle. This fixes several minor issues and inconsistencies, and improves backwards compatibility with Python 3.6.

- bpo-33134 [https://bugs.python.org/issue?@action=redirect&bpo=33134]: When computing dataclass’s __hash__, use the lookup table to contain the function which returns the __hash__ value. This is an improvement over looking up a string, and then testing that string to see what to do.

- bpo-33127 [https://bugs.python.org/issue?@action=redirect&bpo=33127]: The ssl module now compiles with LibreSSL 2.7.1.

- bpo-32505 [https://bugs.python.org/issue?@action=redirect&bpo=32505]: Raise TypeError if a member variable of a dataclass is of type Field, but doesn’t have a type annotation.

- bpo-33078 [https://bugs.python.org/issue?@action=redirect&bpo=33078]: Fix the failure on OSX caused by the tests relying on sem_getvalue

- bpo-33116 [https://bugs.python.org/issue?@action=redirect&bpo=33116]: Add ‘Field’ to dataclasses.__all__.

- bpo-32896 [https://bugs.python.org/issue?@action=redirect&bpo=32896]: Fix an error where subclassing a dataclass with a field that uses a default_factory would generate an incorrect class.

- bpo-33100 [https://bugs.python.org/issue?@action=redirect&bpo=33100]: Dataclasses: If a field has a default value that’s a MemberDescriptorType, then it’s from that field being in __slots__, not an actual default value.

- bpo-32953 [https://bugs.python.org/issue?@action=redirect&bpo=32953]: If a non-dataclass inherits from a frozen dataclass, allow attributes to be added to the derived class. Only attributes from the frozen dataclass cannot be assigned to. Require all dataclasses in a hierarchy to be either all frozen or all non-frozen.

- bpo-33097 [https://bugs.python.org/issue?@action=redirect&bpo=33097]: Raise RuntimeError when `executor.submit` is called during interpreter shutdown.
- bpo-32968 [https://bugs.python.org/issue?@action=redirect&bpo=32968]: Modulo and floor division involving Fraction and float should return float.
- bpo-33061 [https://bugs.python.org/issue?@action=redirect&bpo=33061]: Add missing `NoReturn` to `__all__` in typing.py
- bpo-33078 [https://bugs.python.org/issue?@action=redirect&bpo=33078]: Fix the size handling in multiprocessing.Queue when a pickling error occurs.
- bpo-33064 [https://bugs.python.org/issue?@action=redirect&bpo=33064]: lib2to3 now properly supports trailing commas after `*args` and `**kwargs` in function signatures.
- bpo-33056 [https://bugs.python.org/issue?@action=redirect&bpo=33056]: FIX properly close leaking fds in concurrent.futures.ProcessPoolExecutor.
- bpo-33021 [https://bugs.python.org/issue?@action=redirect&bpo=33021]: Release the GIL during fstat() calls, avoiding hang of all threads when calling mmap.mmap(), os.urandom(), and random.seed(). Patch by Nir Soffer.

- bpo-31804 [https://bugs.python.org/issue?@action=redirect&bpo=31804]: Avoid failing in multiprocessing.Process if the standard streams are closed or None at exit.

- bpo-33034 [https://bugs.python.org/issue?@action=redirect&bpo=33034]: Providing an explicit error message when casting the port property to anything that is not an integer value using `urlparse()` and `urlsplit()`. Patch by Matt Eaton.

- bpo-30249 [https://bugs.python.org/issue?@action=redirect&bpo=30249]: Improve struct.unpack_from() exception messages for problems with the buffer size and offset.

- bpo-33037 [https://bugs.python.org/issue?@action=redirect&bpo=33037]: Skip sending/receiving data after SSL transport closing.

- bpo-27683 [https://bugs.python.org/issue?@action=redirect&bpo=27683]: Fix a regression in **`ipaddress`** that result of **`hosts()`** is empty when the network is constructed by a tuple containing an integer mask and only 1 bit left for addresses.

- bpo-22674 [https://bugs.python.org/issue?@action=redirect&bpo=22674]: Add the strsignal() function in the signal module that returns the system description of the given signal, as returned by strsignal(3).

- bpo-32999 [https://bugs.python.org/issue?@action=redirect&bpo=32999]: Fix C implementation of `ABC.__subclasscheck__(cls, subclass)` crashed when `subclass` is not a type object.

- bpo-33009 [https://bugs.python.org/issue?@action=redirect&bpo=33009]: Fix inspect.signature() for single-parameter partialmethods.
- bpo-32969 [https://bugs.python.org/issue?@action=redirect&bpo=32969]: Expose several missing constants in zlib and fix corresponding documentation.
- bpo-32056 [https://bugs.python.org/issue?@action=redirect&bpo=32056]: Improved exceptions raised for invalid number of channels and sample width when read an audio file in modules **`aifc`**, **`wave`** and **`sunau`**.
- bpo-32970 [https://bugs.python.org/issue?@action=redirect&bpo=32970]: Improved disassembly of the MAKE_FUNCTION instruction.
- bpo-32844 [https://bugs.python.org/issue?@action=redirect&bpo=32844]: Fix wrong redirection of a low descriptor (0 or 1) to stderr in subprocess if another low descriptor is closed.
- bpo-32960 [https://bugs.python.org/issue?@action=redirect&bpo=32960]: For dataclasses, disallow inheriting frozen from non-frozen classes, and also disallow inheriting non-frozen from frozen classes. This restriction will be relaxed at a future date.
- bpo-32713 [https://bugs.python.org/issue?@action=redirect&bpo=32713]: Fixed tarfile.itn handling of out-of-bounds float values. Patch by Joffrey Fuhrer.
- bpo-32257 [https://bugs.python.org/issue?@action=redirect&bpo=32257]: The ssl module now contains OP_NO_RENEGOTIATION constant, available with OpenSSL 1.1.0h or 1.1.1.

- bpo-32951 [https://bugs.python.org/issue?@action=redirect&bpo=32951]: Direct instantiation of SSLSocket and SSLObject objects is now prohibited. The constructors were never documented, tested, or designed as public constructors. Users were suppose to use ssl.wrap_socket() or SSLContext.

- bpo-32929 [https://bugs.python.org/issue?@action=redirect&bpo=32929]: Remove the tri-state parameter “hash”, and add the boolean “unsafe_hash”. If unsafe_hash is True, add a __hash__ function, but if a __hash__ exists, raise TypeError. If unsafe_hash is False, add a __hash__ based on the values of eq= and frozen=. The unsafe_hash=False behavior is the same as the old hash=None behavior. unsafe_hash=False is the default, just as hash=None used to be.

- bpo-32947 [https://bugs.python.org/issue?@action=redirect&bpo=32947]: Add OP_ENABLE_MIDDLEBOX_COMPAT and test workaround for TLSv1.3 for future compatibility with OpenSSL 1.1.1.

- bpo-32146 [https://bugs.python.org/issue?@action=redirect&bpo=32146]: Document the interaction between frozen executables and the spawn and forkserver start methods in multiprocessing.

- bpo-30622 [https://bugs.python.org/issue?@action=redirect&bpo=30622]: The ssl module now detects missing NPN support in LibreSSL.

- bpo-32922 [https://bugs.python.org/issue?@action=redirect&bpo=32922]: dbm.open() now encodes filename with the filesystem encoding rather than default encoding.

- bpo-32759 [https://bugs.python.org/issue?@action=redirect&bpo=32759]: Free unused arenas in multiprocessing.heap.

- bpo-32859 [https://bugs.python.org/issue?@action=redirect&bpo=32859]: In `os.dup2`, don’t check every call whether the `dup3` syscall exists or not.

- bpo-32556 [https://bugs.python.org/issue?@action=redirect&bpo=32556]: nt._getfinalpathname, nt._getvolumepathname and nt._getdiskusage now correctly convert from bytes.

- bpo-21060 [https://bugs.python.org/issue?@action=redirect&bpo=21060]: Rewrite confusing message from setup.py upload from “No dist file created in earlier command” to the more helpful “Must create and upload files in one command”.

- bpo-32857 [https://bugs.python.org/issue?@action=redirect&bpo=32857]: In **`tkinter`**, `after_cancel(None)` now raises a **`ValueError`** instead of canceling the first scheduled function. Patch by Cheryl Sabella.

- bpo-32852 [https://bugs.python.org/issue?@action=redirect&bpo=32852]: Make sure sys.argv remains as a list when running trace.

- bpo-31333 [https://bugs.python.org/issue?@action=redirect&bpo=31333]: `_abc` module is added. It is a speedup module with C implementations for various functions and methods in `abc`. Creating an ABC subclass and calling `isinstance` or `issubclass` with an ABC subclass are up to 1.5x faster. In addition, this makes Python start-up up to 10% faster.

Note that the new implementation hides internal registry and caches, previously accessible via private attributes `_abc_registry`, `_abc_cache`, and `_abc_negative_cache`. There are three debugging helper methods that can be used instead `_dump_registry`, `_abc_registry_clear`, and `_abc_caches_clear`.

- bpo-32841 [https://bugs.python.org/issue?@action=redirect&bpo=32841]: Fixed `asyncio.Condition` issue which silently ignored cancellation after notifying and cancelling a conditional lock. Patch by Bar Harel.
- bpo-32819 [https://bugs.python.org/issue?@action=redirect&bpo=32819]: ssl.match_hostname() has been simplified and no longer depends on re and ipaddress module for wildcard and IP addresses. Error reporting for invalid wildcards has been improved.
- bpo-19675 [https://bugs.python.org/issue?@action=redirect&bpo=19675]: `multiprocessing.Pool` no longer leaks processes if its initialization fails.
- bpo-32394 [https://bugs.python.org/issue?@action=redirect&bpo=32394]: socket: Remove TCP_FASTOPEN,TCP_KEEPCNT,TCP_KEEPIDLE,TCP_KEEPINTVL flags on older version Windows during run-time.
- bpo-31787 [https://bugs.python.org/issue?@action=redirect&bpo=31787]: Fixed refleaks of `__init__()` methods in various modules. (Contributed by Oren Milman)
- bpo-30157 [https://bugs.python.org/issue?@action=redirect&bpo=30157]: Fixed guessing quote and delimiter in csv.Sniffer.sniff() when only the last field is quoted. Patch by Jake Davis.

- bpo-30688 [https://bugs.python.org/issue?@action=redirect&bpo=30688]: Added support of `\N{name}` escapes in regular expressions. Based on patch by Jonathan Eunice.

- bpo-32792 [https://bugs.python.org/issue?@action=redirect&bpo=32792]: collections.ChainMap() preserves the order of the underlying mappings.

- bpo-32775 [https://bugs.python.org/issue?@action=redirect&bpo=32775]: **`fnmatch.translate()`** no longer produces patterns which contain set operations. Sets starting with '[' or containing '–', '&&', '~~' or '||' will be interpreted differently in regular expressions in future versions. Currently they emit warnings. fnmatch.translate() now avoids producing patterns containing such sets by accident.

- bpo-32622 [https://bugs.python.org/issue?@action=redirect&bpo=32622]: Implement native fast sendfile for Windows proactor event loop.

- bpo-32777 [https://bugs.python.org/issue?@action=redirect&bpo=32777]: Fix a rare but potential pre-exec child process deadlock in subprocess on POSIX systems when marking file descriptors inheritable on exec in the child process. This bug appears to have been introduced in 3.4.

- bpo-32647 [https://bugs.python.org/issue?@action=redirect&bpo=32647]: The ctypes module used to depend on indirect linking for dlopen. The shared extension is now explicitly linked against libdl on platforms with dl.

- bpo-32749 [https://bugs.python.org/issue?@action=redirect&bpo=32749]: A **`dbm.dumb`** database opened with flags ‘r’ is now read-only. **`dbm.dumb.open()`** with flags ‘r’ and ‘w’ no longer creates a database if it does not exist.

- bpo-32741 [https://bugs.python.org/issue?@action=redirect&bpo=32741]: Implement `asyncio.TimerHandle.when()` method.

- bpo-32691 [https://bugs.python.org/issue?@action=redirect&bpo=32691]: Use mod_spec.parent when running modules with pdb

- bpo-32734 [https://bugs.python.org/issue?@action=redirect&bpo=32734]: Fixed `asyncio.Lock()` safety issue which allowed acquiring and locking the same lock multiple times, without it being free. Patch by Bar Harel.

- bpo-32727 [https://bugs.python.org/issue?@action=redirect&bpo=32727]: Do not include name field in SMTP envelope from address. Patch by Stéphane Wirtel

- bpo-31453 [https://bugs.python.org/issue?@action=redirect&bpo=31453]: Add TLSVersion constants and SSLContext.maximum_version / minimum_version attributes. The new API wraps OpenSSL 1.1 https://web.archive.org/web/20180309043602/https://www.openssl.org/docs/man1.1.0/ssl/SSL_CTX_set_min_proto_version.html feature.

- bpo-24334 [https://bugs.python.org/issue?@action=redirect&bpo=24334]: Internal implementation details of ssl module were cleaned up. The SSLSocket has one less layer of indirection. Owner and session information are now handled by the SSLSocket and SSLObject constructor. Channel binding implementation has been simplified.

- bpo-31848 [https://bugs.python.org/issue?@action=redirect&bpo=31848]: Fix the error handling in Aifc_read.initfp() when the SSND chunk is not found. Patch by Zackery Spytz.

- bpo-32585 [https://bugs.python.org/issue?@action=redirect&bpo=32585]: Add Ttk spinbox widget to **`tkinter.ttk`**. Patch by Alan D Moore.

- bpo-32512 [https://bugs.python.org/issue?@action=redirect&bpo=32512]: **`profile`** CLI accepts `-m module_name` as an alternative to script path.

- bpo-8525 [https://bugs.python.org/issue?@action=redirect&bpo=8525]: help() on a type now displays builtin subclasses. This is intended primarily to help with notification of more specific exception subclasses.

  Patch by Sanyam Khurana.

- bpo-31639 [https://bugs.python.org/issue?@action=redirect&bpo=31639]: http.server now exposes a ThreadingHTTPServer class and uses it when the module is run with `-m` to cope with web browsers pre-opening sockets.

- bpo-29877 [https://bugs.python.org/issue?@action=redirect&bpo=29877]: compileall: import ProcessPoolExecutor only when needed, preventing hangs on low resource platforms

- bpo-32221 [https://bugs.python.org/issue?@action=redirect&bpo=32221]: Various functions returning tuple containing IPv6 addresses now omit `%scope` part since the same information is already encoded in *scopeid* tuple item. Especially this speeds up

`socket.recvfrom()` when it receives multicast packet since useless resolving of network interface name is omitted.

- bpo-32147 [https://bugs.python.org/issue?@action=redirect&bpo=32147]: **`binascii.unhexlify()`** is now up to 2 times faster. Patch by Sergey Fedoseev.

- bpo-30693 [https://bugs.python.org/issue?@action=redirect&bpo=30693]: The TarFile class now recurses directories in a reproducible way.

- bpo-30693 [https://bugs.python.org/issue?@action=redirect&bpo=30693]: The ZipFile class now recurses directories in a reproducible way.

- bpo-31680 [https://bugs.python.org/issue?@action=redirect&bpo=31680]: Added **`curses.ncurses_version`**.

- bpo-31908 [https://bugs.python.org/issue?@action=redirect&bpo=31908]: Fix output of cover files for `trace` module command-line tool. Previously emitted cover files only when `--missing` option was used. Patch by Michael Selik.

- bpo-31608 [https://bugs.python.org/issue?@action=redirect&bpo=31608]: Raise a `TypeError` instead of crashing if a `collections.deque` subclass returns a non-deque from `__new__`. Patch by Oren Milman.

- bpo-31425 [https://bugs.python.org/issue?@action=redirect&bpo=31425]: Add support for sockets of the AF_QIPCRTR address family, supported by the Linux kernel. This is used to communicate with services, such as GPS or radio, running on Qualcomm devices. Patch by Bjorn Andersson.

- bpo-22005 [https://bugs.python.org/issue?@action=redirect&bpo=22005]: Implemented unpickling instances of **`datetime`**, **`date`** and **`time`** pickled by Python 2. `encoding='latin1'` should be used for successful decoding.

- bpo-27645 [https://bugs.python.org/issue?@action=redirect&bpo=27645]: **`sqlite3.Connection`** now exposes a **`backup`** method, if the underlying SQLite library is at version 3.6.11 or higher. Patch by Lele Gaifax.

- bpo-16865 [https://bugs.python.org/issue?@action=redirect&bpo=16865]: Support arrays >=2GiB in **`ctypes`**. Patch by Segev Finer.

- bpo-31508 [https://bugs.python.org/issue?@action=redirect&bpo=31508]: Removed support of arguments in `tkinter.ttk.Treeview.selection`. It was deprecated in 3.6. Use specialized methods like `selection_set` for changing the selection.

- bpo-29456 [https://bugs.python.org/issue?@action=redirect&bpo=29456]: Fix bugs in hangul normalization: u1176, u11a7 and u11c3

Documentation

- bpo-21257 [https://bugs.python.org/issue?@action=redirect&bpo=21257]: Document **`http.client.parse_headers()`**.
- bpo-34764 [https://bugs.python.org/issue?@action=redirect&bpo=34764]: Improve example of iter() with 2nd sentinel argument.
- bpo-35564 [https://bugs.python.org/issue?@action=redirect&bpo=35564]: Explicitly set master_doc variable in conf.py for compliance with Sphinx 2.0

- bpo-35511 [https://bugs.python.org/issue?@action=redirect&bpo=35511]: Specified that profile.Profile class doesn’t not support enable or disable methods. Also, elaborated that Profile object as a context manager is only supported in cProfile module.
- bpo-10536 [https://bugs.python.org/issue?@action=redirect&bpo=10536]: Enhance the gettext docs. Patch by Éric Araujo
- bpo-35089 [https://bugs.python.org/issue?@action=redirect&bpo=35089]: Remove mention of `typing.io` and `typing.re`. Their types should be imported from `typing` directly.
- bpo-35038 [https://bugs.python.org/issue?@action=redirect&bpo=35038]: Fix the documentation about an unexisting `f_restricted` attribute in the frame object. Patch by Stéphane Wirtel
- bpo-35042 [https://bugs.python.org/issue?@action=redirect&bpo=35042]: Replace PEP XYZ by the pep role and allow to use the direct links to the PEPs.
- bpo-35044 [https://bugs.python.org/issue?@action=redirect&bpo=35044]: Fix the documentation with the role `exc` for the appropriated exception. Patch by Stéphane Wirtel
- bpo-35035 [https://bugs.python.org/issue?@action=redirect&bpo=35035]: Rename documentation for **`email.utils`** to `email.utils.rst`.
- bpo-34967 [https://bugs.python.org/issue?@action=redirect&bpo=34967]: Use app.add_object_type() instead of the deprecated Sphinx function app.description_unit()
- bpo-34913 [https://bugs.python.org/issue?@action=redirect&bpo=34913]: Add documentation about the new command line interface of the gzip module.
- bpo-32174 [https://bugs.python.org/issue?@action=redirect&bpo=32174]: chm document displays non-

ASCII characters properly on some MBCS Windows systems.

- bpo-11233 [https://bugs.python.org/issue?@action=redirect&bpo=11233]: Create availability directive for documentation. Original patch by Georg Brandl.
- bpo-34790 [https://bugs.python.org/issue?@action=redirect&bpo=34790]: Document how passing coroutines to asyncio.wait() can be confusing.
- bpo-34552 [https://bugs.python.org/issue?@action=redirect&bpo=34552]: Make clear that `==` operator sometimes is equivalent to `is`. The `<`, `<=`, `>` and `>=` operators are only defined where they make sense.
- bpo-28617 [https://bugs.python.org/issue?@action=redirect&bpo=28617]: Fixed info in the stdtypes docs concerning the types that support membership tests.
- bpo-20177 [https://bugs.python.org/issue?@action=redirect&bpo=20177]: Migrate datetime.date.fromtimestamp to Argument Clinic. Patch by Tim Hoffmann.
- bpo-34065 [https://bugs.python.org/issue?@action=redirect&bpo=34065]: Fix wrongly written basicConfig documentation markup syntax
- bpo-33460 [https://bugs.python.org/issue?@action=redirect&bpo=33460]: replaced ellipsis with correct error codes in tutorial chapter 3.
- bpo-33847 [https://bugs.python.org/issue?@action=redirect&bpo=33847]: Add '@' operator entry to index.
- bpo-33409 [https://bugs.python.org/issue?@action=redirect&bpo=33409]: Clarified the relationship between **PEP 538** [https://peps.python.org/pep-0538/]'s PYTHONCOERCECLOCALE and PEP 540's PYTHONUTF8 mode.
- bpo-33197 [https://bugs.python.org/issue?@action=redirect&bpo=33197]: Add versionadded tag to the documentation of ParameterKind.description

- bpo-17045 [https://bugs.python.org/issue?@action=redirect&bpo=17045]: Improve the C-API doc for PyTypeObject. This includes adding several quick-reference tables and a lot of missing slot/typedef entries. The existing entries were also cleaned up with a slightly more consistent format.
- bpo-33736 [https://bugs.python.org/issue?@action=redirect&bpo=33736]: Improve the documentation of **`asyncio.open_connection()`**, **`asyncio.start_server()`** and their UNIX socket counterparts.
- bpo-23859 [https://bugs.python.org/issue?@action=redirect&bpo=23859]: Document that `asyncio.wait()` does not cancel its futures on timeout.
- bpo-32436 [https://bugs.python.org/issue?@action=redirect&bpo=32436]: Document **PEP 567** [https://peps.python.org/pep-0567/] changes to asyncio.
- bpo-33604 [https://bugs.python.org/issue?@action=redirect&bpo=33604]: Update HMAC md5 default to a DeprecationWarning, bump removal to 3.8.
- bpo-33594 [https://bugs.python.org/issue?@action=redirect&bpo=33594]: Document `getargspec`, `from_function` and `from_builtin` as deprecated in their respective docstring, and include version since deprecation in DeprecationWarning message.
- bpo-33503 [https://bugs.python.org/issue?@action=redirect&bpo=33503]: Fix broken pypi link
- bpo-33421 [https://bugs.python.org/issue?@action=redirect&bpo=33421]: Add missing documentation for `typing.AsyncContextManager`.
- bpo-33487 [https://bugs.python.org/issue?@action=redirect&bpo=33487]: BZ2file now emit a DeprecationWarning when buffering=None is passed, the deprecation message and documentation also now explicitly state it is deprecated since 3.0.
- bpo-33378 [https://bugs.python.org/issue?@action=redirect&bpo=33378]: Add Korean language switcher

for https://docs.python.org/3/

- bpo-33276 [https://bugs.python.org/issue?@action=redirect&bpo=33276]: Clarify that the `__path__` attribute on modules cannot be just any value.
- bpo-33201 [https://bugs.python.org/issue?@action=redirect&bpo=33201]: Modernize documentation for writing C extension types.
- bpo-33195 [https://bugs.python.org/issue?@action=redirect&bpo=33195]: Deprecate `Py_UNICODE` usage in `c-api/arg` document. `Py_UNICODE` related APIs are deprecated since Python 3.3, but it is missed in the document.
- bpo-33126 [https://bugs.python.org/issue?@action=redirect&bpo=33126]: Document PyBuffer_ToContiguous().
- bpo-27212 [https://bugs.python.org/issue?@action=redirect&bpo=27212]: Modify documentation for the **`islice()`** recipe to consume initial values up to the start index.
- bpo-28247 [https://bugs.python.org/issue?@action=redirect&bpo=28247]: Update **`zipapp`** documentation to describe how to make standalone applications.
- bpo-18802 [https://bugs.python.org/issue?@action=redirect&bpo=18802]: Documentation changes for ipaddress. Patch by Jon Foster and Berker Peksag.
- bpo-27428 [https://bugs.python.org/issue?@action=redirect&bpo=27428]: Update documentation to clarify that `WindowsRegistryFinder` implements `MetaPathFinder`. (Patch by Himanshu Lakhara)
- bpo-28124 [https://bugs.python.org/issue?@action=redirect&bpo=28124]: The ssl module function ssl.wrap_socket() has been de-emphasized and deprecated in favor of the more secure and efficient SSLContext.wrap_socket() method.
- bpo-17232 [https://bugs.python.org/issue?@action=redirect&bpo=17232]: Clarify docs for -O and -OO.

Patch by Terry Reedy.

- bpo-32436 [https://bugs.python.org/issue?@action=redirect&bpo=32436]: Add documentation for the contextvars module (PEP 567).
- bpo-32800 [https://bugs.python.org/issue?@action=redirect&bpo=32800]: Update link to w3c doc for xml default namespaces.
- bpo-11015 [https://bugs.python.org/issue?@action=redirect&bpo=11015]: Update **`test.support`** documentation.
- bpo-32613 [https://bugs.python.org/issue?@action=redirect&bpo=32613]: Update the faq/windows.html to use the py command from PEP 397 instead of python.
- bpo-8722 [https://bugs.python.org/issue?@action=redirect&bpo=8722]: Document **`__getattr__()`** behavior when property **`get()`** method raises **`AttributeError`**.
- bpo-32614 [https://bugs.python.org/issue?@action=redirect&bpo=32614]: Modify RE examples in documentation to use raw strings to prevent **`DeprecationWarning`** and add text to REGEX HOWTO to highlight the deprecation.
- bpo-20709 [https://bugs.python.org/issue?@action=redirect&bpo=20709]: Remove the paragraph where we explain that os.utime() does not support a directory as path under Windows. Patch by Jan-Philip Gehrcke
- bpo-32722 [https://bugs.python.org/issue?@action=redirect&bpo=32722]: Remove the bad example in the tutorial of the Generator Expression. Patch by Stéphane Wirtel
- bpo-31972 [https://bugs.python.org/issue?@action=redirect&bpo=31972]: Improve docstrings for `pathlib.PurePath` subclasses.
- bpo-30607 [https://bugs.python.org/issue?@action=redirect&bpo=30607]: Use the externalized `python-docs-theme` package when building the documentation.

- bpo-8243 [https://bugs.python.org/issue?@action=redirect&bpo=8243]: Add a note about curses.addch and curses.addstr exception behavior when writing outside a window, or pad.
- bpo-32337 [https://bugs.python.org/issue?@action=redirect&bpo=32337]: Update documentation related with `dict` order.
- bpo-25041 [https://bugs.python.org/issue?@action=redirect&bpo=25041]: Document `AF_PACKET` in the **`socket`** module.
- bpo-31432 [https://bugs.python.org/issue?@action=redirect&bpo=31432]: Clarify meaning of CERT_NONE, CERT_OPTIONAL, and CERT_REQUIRED flags for ssl.SSLContext.verify_mode.

Tests

- bpo-35772 [https://bugs.python.org/issue?@action=redirect&bpo=35772]: Fix sparse file tests of test_tarfile on ppc64 with the tmpfs filesystem. Fix the function testing if the filesystem supports sparse files: create a file which contains data and “holes”, instead of creating a file which contains no data. tmpfs effective block size is a page size (tmpfs lives in the page cache). RHEL uses 64 KiB pages on aarch64, ppc64, ppc64le, only s390x and x86_64 use 4 KiB pages, whereas the test punch holes of 4 KiB.

- bpo-35045 [https://bugs.python.org/issue?@action=redirect&bpo=35045]: Make ssl tests less strict and also accept TLSv1 as system default. The changes unbreaks test_min_max_version on Fedora 29.

- bpo-32710 [https://bugs.python.org/issue?@action=redirect&bpo=32710]: `test_asyncio/test_sendfile.py` now resets the event loop policy using **`tearDownModule()`**

as done in other tests, to prevent a warning when running tests on Windows.

- bpo-33717 [https://bugs.python.org/issue?@action=redirect&bpo=33717]: test.pythoninfo now logs information of all clocks, not only time.time() and time.perf_counter().

- bpo-35488 [https://bugs.python.org/issue?@action=redirect&bpo=35488]: Add a test to pathlib’s Path.match() to verify it does not support glob-style ** recursive pattern matching.

- bpo-31731 [https://bugs.python.org/issue?@action=redirect&bpo=31731]: Fix a race condition in `check_interrupted_write()` of test_io: create directly the thread with SIGALRM signal blocked, rather than blocking the signal later from the thread. Previously, it was possible that the thread gets the signal before the signal is blocked.

- bpo-35424 [https://bugs.python.org/issue?@action=redirect&bpo=35424]: Fix test_multiprocessing_main_handling: use **`multiprocessing.Pool`** with a context manager and then explicitly join the pool.

- bpo-35519 [https://bugs.python.org/issue?@action=redirect&bpo=35519]: Rename **`test.bisect`** module to **`test.bisect_cmd`** to avoid conflict with **`bisect`** module when running directly a test like `./python Lib/test/test_xmlrpc.py`.

- bpo-35513 [https://bugs.python.org/issue?@action=redirect&bpo=35513]: Replace **`time.time()`** with **`time.monotonic()`** in tests to measure time delta.

- bpo-34279 [https://bugs.python.org/issue?@action=redirect&bpo=34279]: **`test.support.run_unittest()`** no longer raise **`TestDidNotRun`** if the test result contains skipped tests. The exception is now only raised if no test have been run and no test have been skipped.

- bpo-35412 [https://bugs.python.org/issue?@action=redirect&bpo=35412]: Add testcase to `test_future4`: check unicode literal.

- bpo-26704 [https://bugs.python.org/issue?@action=redirect&bpo=26704]: Added test demonstrating double-patching of an instance method. Patch by Anthony Sottile.

- bpo-33725 [https://bugs.python.org/issue?@action=redirect&bpo=33725]: test_multiprocessing_fork may crash on recent versions of macOS. Until the issue is resolved, skip the test on macOS.

- bpo-35352 [https://bugs.python.org/issue?@action=redirect&bpo=35352]: Modify test_asyncio to use the certificate set from the test directory.

- bpo-35317 [https://bugs.python.org/issue?@action=redirect&bpo=35317]: Fix `mktime()` overflow error in `test_email`: run `test_localtime_daylight_true_dst_true()` and `test_localtime_daylight_false_dst_true()` with a specific timezone.

- bpo-21263 [https://bugs.python.org/issue?@action=redirect&bpo=21263]: After several reports that test_gdb does not work properly on macOS and since gdb is not shipped by default anymore, test_gdb is now skipped on macOS when LLVM Clang has been used to compile Python. Patch by Lysandros Nikolaou

- bpo-34279 [https://bugs.python.org/issue?@action=redirect&bpo=34279]: regrtest issue a warning when no tests have been executed in a particular test file. Also, a new final result state is issued if no test have been executed across all test files. Patch by Pablo Galindo.

- bpo-34962 [https://bugs.python.org/issue?@action=redirect&bpo=34962]: make docstest in Doc now passes., and is enforced in CI

- bpo-23596 [https://bugs.python.org/issue?@action=redirect&bpo=23596]: Use argparse for the command line of the gzip module. Patch by Antony Lee

- bpo-34537 [https://bugs.python.org/issue?@action=redirect&bpo=34537]: Fix `test_gdb.test_strings()` when `LC_ALL=C` and GDB was compiled with Python 3.6 or earlier.

- bpo-34587 [https://bugs.python.org/issue?@action=redirect&bpo=34587]: test_socket: Remove RDSTest.testCongestion(). The test tries to fill the receiver's socket buffer and expects an error. But the RDS protocol doesn't require that. Moreover, the Linux implementation of RDS expects that the producer of the messages reduces its rate, it's not the role of the receiver to trigger an error. The test fails on Fedora 28 by design, so just remove it.

- bpo-34661 [https://bugs.python.org/issue?@action=redirect&bpo=34661]: Fix test_shutil if unzip doesn't support -t.

- bpo-34200 [https://bugs.python.org/issue?@action=redirect&bpo=34200]: Fixed non-deterministic flakiness of test_pkg by not using the scary test.support.module_cleanup() logic to save and restore sys.modules contents between test cases.

- bpo-34569 [https://bugs.python.org/issue?@action=redirect&bpo=34569]: The experimental PEP 554 data channels now correctly pass negative PyLong objects between subinterpreters on 32-bit systems. Patch by Michael Felt.
- bpo-34594 [https://bugs.python.org/issue?@action=redirect&bpo=34594]: Fix usage of hardcoded `errno` values in the tests.
- bpo-34579 [https://bugs.python.org/issue?@action=redirect&bpo=34579]: Fix test_embed for AIX Patch by Michael Felt
- bpo-34542 [https://bugs.python.org/issue?@action=redirect&bpo=34542]: Use 3072 RSA keys and SHA-256 signature for test certs and keys.
- bpo-11193 [https://bugs.python.org/issue?@action=redirect&bpo=11193]: Remove special condition for AIX in `test_subprocess.test_undecodable_env`
- bpo-34347 [https://bugs.python.org/issue?@action=redirect&bpo=34347]: Fix `test_utf8_mode.test_cmd_line` for AIX
- bpo-34490 [https://bugs.python.org/issue?@action=redirect&bpo=34490]: On AIX with AF_UNIX family sockets getsockname() does not provide ‘sockname’, so skip calls to transport.get_extra_info(‘sockname’)
- bpo-34391 [https://bugs.python.org/issue?@action=redirect&bpo=34391]: Fix ftplib test for TLS 1.3 by reading from data socket.

- bpo-11192 [https://bugs.python.org/issue?@action=redirect&bpo=11192]: Fix `test_socket` on AIX 6.1 and later IPv6 zone id supports only supported by `inet_pton6_zone()`. Switch to runtime-based `platform.system()` to establish current platform rather than build-time based `sys.platform()`

- bpo-34399 [https://bugs.python.org/issue?@action=redirect&bpo=34399]: Update all RSA keys and DH params to use at least 2048 bits.

- bpo-34373 [https://bugs.python.org/issue?@action=redirect&bpo=34373]: Fix `test_mktime` and `test_pthread_getcpuclickid` tests for AIX Add range checking for `_PyTime_localtime` for AIX Patch by Michael Felt

- bpo-11191 [https://bugs.python.org/issue?@action=redirect&bpo=11191]: Skip the distutils test 'test_search_cpp' when using XLC as compiler patch by aixtools (Michael Felt)

- Improved an error message when mock assert_has_calls fails.

- bpo-33746 [https://bugs.python.org/issue?@action=redirect&bpo=33746]: Fix test_unittest when run in verbose mode.

- bpo-33901 [https://bugs.python.org/issue?@action=redirect&bpo=33901]: Fix test_dbm_gnu on macOS with gdbm 1.15: add a larger value to make sure that the file size changes.

- bpo-33873 [https://bugs.python.org/issue?@action=redirect&bpo=33873]: Fix a bug in `regrtest` that caused an extra test to run if –huntrleaks/-R was used.

Exit with error in case that invalid parameters are specified to –huntrleaks/-R (at least one warmup run and one repetition must be used).

- bpo-33562 [https://bugs.python.org/issue?@action=redirect&bpo=33562]: Check that a global asyncio event loop policy is not left behind by any tests.

- bpo-33655 [https://bugs.python.org/issue?@action=redirect&bpo=33655]: Ignore test_posix_fallocate failures on BSD platforms that might be due to running on ZFS.

- bpo-32962 [https://bugs.python.org/issue?@action=redirect&bpo=32962]: Fixed test_gdb when Python is compiled with flags -mcet -fcf-protection -O0.

- bpo-33358 [https://bugs.python.org/issue?@action=redirect&bpo=33358]: Fix `test_embed.test_pre_initialization_sys_options()` when the interpreter is built with `--enable-shared`.

- bpo-32872 [https://bugs.python.org/issue?@action=redirect&bpo=32872]: Avoid regrtest compatibility issue with namespace packages.

- bpo-32517 [https://bugs.python.org/issue?@action=redirect&bpo=32517]: Fix failing `test_asyncio` on macOS 10.12.2+ due to transport of `KqueueSelector` loop was not being closed.

- bpo-32663 [https://bugs.python.org/issue?@action=redirect&bpo=32663]: Making sure the `SMTPUTF8SimTests` class of tests gets run in `test_smtplib.py`.

- bpo-27643 [https://bugs.python.org/issue?@action=redirect&bpo=27643]: Test_C test case needs “signed short” bitfields, but the IBM XLC compiler (on AIX) does not support this Skip the code and test when AIX and XLC are used

Applicable to Python2-2.7 and later

- bpo-19417 [https://bugs.python.org/issue?@action=redirect&bpo=19417]: Add test_bdb.py.

- bpo-31809 [https://bugs.python.org/issue?@action=redirect&bpo=31809]: Add tests to verify connection with secp ECDH curves.

Build

- bpo-34691 [https://bugs.python.org/issue?@action=redirect&bpo=34691]: The _contextvars module is now built into the core Python library on Windows.
- bpo-35683 [https://bugs.python.org/issue?@action=redirect&bpo=35683]: Improved Azure Pipelines build steps and now verifying layouts correctly
- bpo-35642 [https://bugs.python.org/issue?@action=redirect&bpo=35642]: Remove asynciomodule.c from pythoncore.vcxproj
- bpo-35550 [https://bugs.python.org/issue?@action=redirect&bpo=35550]: Fix incorrect Solaris #ifdef checks to look for __sun && __SVR4 instead of sun when compiling.
- bpo-35499 [https://bugs.python.org/issue?@action=redirect&bpo=35499]: `make profile-opt` no longer replaces `CFLAGS_NODIST` with `CFLAGS`. It now adds profile-guided optimization (PGO) flags to `CFLAGS_NODIST`: existing `CFLAGS_NODIST` flags are kept.
- bpo-35257 [https://bugs.python.org/issue?@action=redirect&bpo=35257]: Avoid leaking the linker flags

from Link Time Optimizations (LTO) into distutils when compiling C extensions.

- bpo-35351 [https://bugs.python.org/issue?@action=redirect&bpo=35351]: When building Python with clang and LTO, LTO flags are no longer passed into CFLAGS to build third-party C extensions through distutils.
- bpo-35139 [https://bugs.python.org/issue?@action=redirect&bpo=35139]: Fix a compiler error when statically linking `pyexpat` in `Modules/Setup`.
- bpo-35059 [https://bugs.python.org/issue?@action=redirect&bpo=35059]: PCbuild: Set InlineFunctionExpansion to OnlyExplicitInline ("/Ob1" option) in pyproject.props in Debug mode to expand functions marked as inline. This change should make Python compiled in Debug mode a little bit faster on Windows.
- bpo-35011 [https://bugs.python.org/issue?@action=redirect&bpo=35011]: Restores the use of pyexpatns.h to isolate our embedded copy of the expat C library so that its symbols do not conflict at link or dynamic loading time with an embedding application or other extension modules with their own version of libexpat.
- bpo-28015 [https://bugs.python.org/issue?@action=redirect&bpo=28015]: Have –with-lto works correctly with clang.
- bpo-34765 [https://bugs.python.org/issue?@action=redirect&bpo=34765]: Update the outdated install-sh file to the latest revision from automake v1.16.1
- bpo-34585 [https://bugs.python.org/issue?@action=redirect&bpo=34585]: Check for floating-point byte order in configure.ac using compilation tests instead of executing code, so that these checks work in cross-compiled builds.

- bpo-34710 [https://bugs.python.org/issue?@action=redirect&bpo=34710]: Fixed SSL module build with OpenSSL & pedantic CFLAGS.
- bpo-34582 [https://bugs.python.org/issue?@action=redirect&bpo=34582]: Add JUnit XML output for regression tests and update Azure DevOps builds.
- bpo-34081 [https://bugs.python.org/issue?@action=redirect&bpo=34081]: Make Sphinx warnings as errors in the Docs Makefile.
- bpo-34555 [https://bugs.python.org/issue?@action=redirect&bpo=34555]: Fix for case where it was not possible to have both `HAVE_LINUX_VM_SOCKETS_H` and `HAVE_SOCKADDR_ALG` be undefined.
- bpo-33015 [https://bugs.python.org/issue?@action=redirect&bpo=33015]: Fix an undefined behaviour in the pthread implementation of **`PyThread_start_new_thread()`**: add a function wrapper to always return `NULL`.
- bpo-34245 [https://bugs.python.org/issue?@action=redirect&bpo=34245]: The Python shared library is now installed with write permission (mode 0755), which is the standard way of installing such libraries.
- bpo-34121 [https://bugs.python.org/issue?@action=redirect&bpo=34121]: Fix detection of C11 atomic support on clang.
- bpo-32430 [https://bugs.python.org/issue?@action=redirect&bpo=32430]: Rename Modules/Setup.dist to Modules/Setup, and remove the necessity to copy the former manually to the latter when updating the local source tree.
- bpo-30345 [https://bugs.python.org/issue?@action=redirect&bpo=30345]: Add -g to LDFLAGS when compiling with LTO to get debug symbols.
- bpo-5755 [https://bugs.python.org/issue?@action=redirect&bpo=5755]: Move `-Wstrict-prototypes` option to `CFLAGS_NODIST` from `OPT`.

This option emitted annoying warnings when building extension modules written in C++.

- bpo-33614 [https://bugs.python.org/issue?@action=redirect&bpo=33614]: Ensures module definition files for the stable ABI on Windows are correctly regenerated.
- bpo-33648 [https://bugs.python.org/issue?@action=redirect&bpo=33648]: The –with-c-locale-warning configuration flag has been removed. It has had no effect for about a year.
- bpo-33522 [https://bugs.python.org/issue?@action=redirect&bpo=33522]: Enable CI builds on Visual Studio Team Services at https://python.visualstudio.com/cpython
- bpo-33512 [https://bugs.python.org/issue?@action=redirect&bpo=33512]: configure’s check for “long double” has been simplified
- bpo-33483 [https://bugs.python.org/issue?@action=redirect&bpo=33483]: C compiler is now correctly detected from the standard environment variables. –without-gcc and –with-icc options have been removed.
- bpo-33394 [https://bugs.python.org/issue?@action=redirect&bpo=33394]: Enable the verbose build for extension modules, when GNU make is passed macros on the command line.
- bpo-33393 [https://bugs.python.org/issue?@action=redirect&bpo=33393]: Update config.guess and config.sub files.
- bpo-33377 [https://bugs.python.org/issue?@action=redirect&bpo=33377]: Add new triplets for mips r6 and riscv variants (used in extension suffixes).
- bpo-32232 [https://bugs.python.org/issue?@action=redirect&bpo=32232]: By default, modules configured in `Modules/Setup` are no longer built with `-DPy_BUILD_CORE`. Instead, modules that specifically need that preprocessor definition include it in their individual entries.

- bpo-33182 [https://bugs.python.org/issue?@action=redirect&bpo=33182]: The embedding tests can once again be built with clang 6.0
- bpo-33163 [https://bugs.python.org/issue?@action=redirect&bpo=33163]: Upgrade pip to 9.0.3 and setuptools to v39.0.1.
- bpo-33012 [https://bugs.python.org/issue?@action=redirect&bpo=33012]: gcc 8 has added a new warning heuristic to detect invalid function casts and a stock python build seems to hit that warning quite often. The most common is the cast of a METH_NOARGS function (that uses just one argument) to a PyCFunction. Fix this by adding a dummy argument to all functions that implement METH_NOARGS.
- bpo-32898 [https://bugs.python.org/issue?@action=redirect&bpo=32898]: Fix the python debug build when using COUNT_ALLOCS.
- bpo-29442 [https://bugs.python.org/issue?@action=redirect&bpo=29442]: Replace optparse with argparse in setup.py

Windows

- bpo-35890 [https://bugs.python.org/issue?@action=redirect&bpo=35890]: Fix API calling consistency of GetVersionEx and wcstok.
- bpo-32560 [https://bugs.python.org/issue?@action=redirect&bpo=32560]: The `py` launcher now forwards its `STARTUPINFO` structure to child processes.
- bpo-35854 [https://bugs.python.org/issue?@action=redirect&bpo=35854]: Fix EnvBuilder and –symlinks in venv on Windows
- bpo-35811 [https://bugs.python.org/issue?@action=redirect&bpo=35811]: Avoid propagating venv settings when launching via py.exe

- bpo-35797 [https://bugs.python.org/issue?@action=redirect&bpo=35797]: Fix default executable used by the multiprocessing module
- bpo-35758 [https://bugs.python.org/issue?@action=redirect&bpo=35758]: Allow building on ARM with MSVC.
- bpo-29734 [https://bugs.python.org/issue?@action=redirect&bpo=29734]: Fix handle leaks in os.stat on Windows.
- bpo-35596 [https://bugs.python.org/issue?@action=redirect&bpo=35596]: Use unchecked PYCs for the embeddable distro to avoid zipimport restrictions.
- bpo-35596 [https://bugs.python.org/issue?@action=redirect&bpo=35596]: Fix vcruntime140.dll being added to embeddable distro multiple times.
- bpo-35402 [https://bugs.python.org/issue?@action=redirect&bpo=35402]: Update Windows build to use Tcl and Tk 8.6.9
- bpo-35401 [https://bugs.python.org/issue?@action=redirect&bpo=35401]: Updates Windows build to OpenSSL 1.1.0j
- bpo-34977 [https://bugs.python.org/issue?@action=redirect&bpo=34977]: venv on Windows will now use a python.exe redirector rather than copying the actual binaries from the base environment.
- bpo-34977 [https://bugs.python.org/issue?@action=redirect&bpo=34977]: Adds support for building a Windows App Store package
- bpo-35067 [https://bugs.python.org/issue?@action=redirect&bpo=35067]: Remove _distutils_findvs module and use vswhere.exe instead.
- bpo-32557 [https://bugs.python.org/issue?@action=redirect&bpo=32557]: Allow shutil.disk_usage to take a file path on Windows
- bpo-34770 [https://bugs.python.org/issue?@action=redirect&bpo=34770]: Fix a possible null pointer

dereference in pyshellext.cpp.

- bpo-34603 [https://bugs.python.org/issue?@action=redirect&bpo=34603]: Fix returning structs from functions produced by MSVC
- bpo-34581 [https://bugs.python.org/issue?@action=redirect&bpo=34581]: Guard MSVC-specific code in socketmodule.c with `#ifdef _MSC_VER`.
- bpo-34532 [https://bugs.python.org/issue?@action=redirect&bpo=34532]: Fixes exit code of list version arguments for py.exe.
- bpo-34062 [https://bugs.python.org/issue?@action=redirect&bpo=34062]: Fixed the ‘–list’ and ‘–list-paths’ arguments for the py.exe launcher
- bpo-34225 [https://bugs.python.org/issue?@action=redirect&bpo=34225]: Ensure INCLUDE and LIB directories do not end with a backslash.
- bpo-34011 [https://bugs.python.org/issue?@action=redirect&bpo=34011]: A suite of code has been changed which copied across DLLs and init.tcl from the running Python location into a venv being created. These copies are needed only when running from a Python source build, and the copying code is now only run when that is the case, rather than whenever a venv is created.
- bpo-34006 [https://bugs.python.org/issue?@action=redirect&bpo=34006]: Revert line length limit for Windows help docs. The line-length limit is not needed because the pages appear in a separate app rather than on a browser tab. It can also interact badly with the DPI setting.
- bpo-31546 [https://bugs.python.org/issue?@action=redirect&bpo=31546]: Restore running PyOS_InputHook while waiting for user input at the prompt. The restores integration of interactive GUI windows (such as Matplotlib figures) with the prompt on Windows.

- bpo-30237 [https://bugs.python.org/issue?@action=redirect&bpo=30237]: Output error when ReadConsole is canceled by CancelSynchronousIo instead of crashing.
- bpo-33895 [https://bugs.python.org/issue?@action=redirect&bpo=33895]: GIL is released while calling functions that acquire Windows loader lock.
- bpo-33720 [https://bugs.python.org/issue?@action=redirect&bpo=33720]: Reduces maximum marshal recursion depth on release builds.
- bpo-29097 [https://bugs.python.org/issue?@action=redirect&bpo=29097]: Fix bug where **`datetime.fromtimestamp()`** erroneously throws an **`OSError`** on Windows for values between 0 and 86400. Patch by Ammar Askar.
- bpo-33316 [https://bugs.python.org/issue?@action=redirect&bpo=33316]: PyThread_release_lock always fails
- bpo-33184 [https://bugs.python.org/issue?@action=redirect&bpo=33184]: Update Windows installer to use OpenSSL 1.1.0h.
- bpo-32890 [https://bugs.python.org/issue?@action=redirect&bpo=32890]: Fix usage of GetLastError() instead of errno in os.execve() and os.truncate().
- bpo-33016 [https://bugs.python.org/issue?@action=redirect&bpo=33016]: Fix potential use of uninitialized memory in nt._getfinalpathname
- bpo-32903 [https://bugs.python.org/issue?@action=redirect&bpo=32903]: Fix a memory leak in os.chdir() on Windows if the current directory is set to a UNC path.
- bpo-32901 [https://bugs.python.org/issue?@action=redirect&bpo=32901]: Update Tcl and Tk versions to 8.6.8
- bpo-31966 [https://bugs.python.org/issue?@action=redirect&bpo=31966]: Fixed WindowsConsoleIO.write() for writing empty data.

- bpo-32409 [https://bugs.python.org/issue?@action=redirect&bpo=32409]: Ensures activate.bat can handle Unicode contents.
- bpo-32457 [https://bugs.python.org/issue?@action=redirect&bpo=32457]: Improves handling of denormalized executable path when launching Python.
- bpo-32370 [https://bugs.python.org/issue?@action=redirect&bpo=32370]: Use the correct encoding for ipconfig output in the uuid module. Patch by Segev Finer.
- bpo-29248 [https://bugs.python.org/issue?@action=redirect&bpo=29248]: Fix `os.readlink()` on Windows, which was mistakenly treating the `PrintNameOffset` field of the reparse data buffer as a number of characters instead of bytes. Patch by Craig Holmquist and SSE4.
- bpo-1104 [https://bugs.python.org/issue?@action=redirect&bpo=1104]: Correctly handle string length in `msilib.SummaryInfo.GetProperty()` to prevent it from truncating the last character.

macOS

- bpo-35401 [https://bugs.python.org/issue?@action=redirect&bpo=35401]: Update macOS installer to use OpenSSL 1.1.0j.
- bpo-35025 [https://bugs.python.org/issue?@action=redirect&bpo=35025]: Properly guard the use of the `CLOCK_GETTIME` et al. macros in `timemodule` on macOS.
- bpo-24658 [https://bugs.python.org/issue?@action=redirect&bpo=24658]: On macOS, fix reading from and writing into a file with a size larger than 2 GiB.
- bpo-34405 [https://bugs.python.org/issue?@action=redirect&bpo=34405]: Update to OpenSSL 1.1.0i for macOS installer builds.
- bpo-33635 [https://bugs.python.org/issue?@action=redirect&bpo=33635]: In macOS stat on some file

descriptors (/dev/fd/3 f.e) will result in bad file descriptor OSError. Guard against this exception was added in is_dir, is_file and similar methods. DirEntry.is_dir can also throw this exception so _RecursiveWildcardSelector._iterate_directories was also extended with the same error ignoring pattern.

- bpo-13631 [https://bugs.python.org/issue?@action=redirect&bpo=13631]: The .editrc file in user's home directory is now processed correctly during the readline initialization through editline emulation on macOS.
- bpo-33184 [https://bugs.python.org/issue?@action=redirect&bpo=33184]: Update macOS installer build to use OpenSSL 1.1.0h.
- bpo-32726 [https://bugs.python.org/issue?@action=redirect&bpo=32726]: Build and link with private copy of Tcl/Tk 8.6 for the macOS 10.6+ installer. The 10.9+ installer variant already does this. This means that the Python 3.7 provided by the python.org macOS installers no longer need or use any external versions of Tcl/Tk, either system-provided or user-installed, such as ActiveTcl.
- bpo-32901 [https://bugs.python.org/issue?@action=redirect&bpo=32901]: Update macOS 10.9+ installer to Tcl/Tk 8.6.8.
- bpo-31903 [https://bugs.python.org/issue?@action=redirect&bpo=31903]: In **`_scproxy`**, drop the GIL when calling into `SystemConfiguration` to avoid deadlocks.

IDLE

- bpo-35770 [https://bugs.python.org/issue?@action=redirect&bpo=35770]: IDLE macosx deletes Options => Configure IDLE. It previously deleted Window => Zoom Height by mistake. (Zoom Height is now on the Options menu). On Mac, the settings dialog is accessed via Preferences on the IDLE menu.

- bpo-35769 [https://bugs.python.org/issue?@action=redirect&bpo=35769]: Change IDLE’s new file name from ‘Untitled’ to ‘untitled’
- bpo-35660 [https://bugs.python.org/issue?@action=redirect&bpo=35660]: Fix imports in idlelib.window.
- bpo-35641 [https://bugs.python.org/issue?@action=redirect&bpo=35641]: Proper format `calltip` when the function has no docstring.
- bpo-33987 [https://bugs.python.org/issue?@action=redirect&bpo=33987]: Use ttk Frame for ttk widgets.
- bpo-34055 [https://bugs.python.org/issue?@action=redirect&bpo=34055]: Fix erroneous ‘smart’ indents and newlines in IDLE Shell.
- bpo-35591 [https://bugs.python.org/issue?@action=redirect&bpo=35591]: Find Selection now works when selection not found.
- bpo-35196 [https://bugs.python.org/issue?@action=redirect&bpo=35196]: Speed up squeezer line counting.
- bpo-35598 [https://bugs.python.org/issue?@action=redirect&bpo=35598]: Update config_key: use PEP 8 names and ttk widgets, make some objects global, and add tests.
- bpo-28097 [https://bugs.python.org/issue?@action=redirect&bpo=28097]: Add Previous/Next History entries to Shell menu.
- bpo-35208 [https://bugs.python.org/issue?@action=redirect&bpo=35208]: Squeezer now properly counts wrapped lines before newlines.
- bpo-35555 [https://bugs.python.org/issue?@action=redirect&bpo=35555]: Gray out Code Context menu entry when it’s not applicable.
- bpo-35521 [https://bugs.python.org/issue?@action=redirect&bpo=35521]: Document the IDLE editor code context feature. Add some internal references within the IDLE doc.

- bpo-22703 [https://bugs.python.org/issue?@action=redirect&bpo=22703]: The Code Context menu label now toggles between Show/Hide Code Context. The Zoom Height menu now toggles between Zoom/Restore Height. Zoom Height has moved from the Window menu to the Options menu.
- bpo-35213 [https://bugs.python.org/issue?@action=redirect&bpo=35213]: Where appropriate, use ‘macOS’ in idlelib.
- bpo-34864 [https://bugs.python.org/issue?@action=redirect&bpo=34864]: On macOS, warn if the system preference “Prefer tabs when opening documents” is set to “Always”.
- bpo-34864 [https://bugs.python.org/issue?@action=redirect&bpo=34864]: Document two IDLE on MacOS issues. The System Preferences Dock “prefer tabs always” setting disables some IDLE features. Menus are a bit different than as described for Windows and Linux.
- bpo-35202 [https://bugs.python.org/issue?@action=redirect&bpo=35202]: Remove unused imports from lib/idlelib
- bpo-33000 [https://bugs.python.org/issue?@action=redirect&bpo=33000]: Document that IDLE’s shell has no line limit. A program that runs indefinitely can overfill memory.
- bpo-23220 [https://bugs.python.org/issue?@action=redirect&bpo=23220]: Explain how IDLE’s Shell displays output.
- bpo-35099 [https://bugs.python.org/issue?@action=redirect&bpo=35099]: Improve the doc about IDLE running user code. The section is renamed from “IDLE – console differences” is renamed “Running user code”. It mostly covers the implications of using custom `sys.stdxxx` objects.
- bpo-35097 [https://bugs.python.org/issue?@action=redirect&bpo=35097]: Add IDLE doc subsection

explaining editor windows. Topics include opening, title and status bar, .py* extension, and running.

- bpo-35093 [https://bugs.python.org/issue?@action=redirect&bpo=35093]: Document the IDLE document viewer in the IDLE doc. Add a paragraph in “Help and preferences”, “Help sources” subsection.
- bpo-35088 [https://bugs.python.org/issue?@action=redirect&bpo=35088]: Update idlelib.help.copy_string docstring. We now use git and backporting instead of hg and forward merging.
- bpo-35087 [https://bugs.python.org/issue?@action=redirect&bpo=35087]: Update idlelib help files for the current doc build. The main change is the elimination of chapter-section numbers.
- bpo-34548 [https://bugs.python.org/issue?@action=redirect&bpo=34548]: Use configured color theme for read-only text views.
- bpo-1529353 [https://bugs.python.org/issue?@action=redirect&bpo=1529353]: Enable “squeezing” of long outputs in the shell, to avoid performance degradation and to clean up the history without losing it. Squeezed outputs may be copied, viewed in a separate window, and “unsqueezed”.
- bpo-34047 [https://bugs.python.org/issue?@action=redirect&bpo=34047]: Fixed mousewheel scrolling direction on macOS.
- bpo-34275 [https://bugs.python.org/issue?@action=redirect&bpo=34275]: Make IDLE calltips always visible on Mac. Some MacOS-tk combinations need .update_idletasks(). Patch by Kevin Walzer.
- bpo-34120 [https://bugs.python.org/issue?@action=redirect&bpo=34120]: Fix unresponsiveness after closing certain windows and dialogs.
- bpo-33975 [https://bugs.python.org/issue?@action=redirect&bpo=33975]: Avoid small type when running htests. Since part of the purpose of human-viewed tests

is to determine that widgets look right, it is important that they look the same for testing as when running IDLE.

- bpo-33905 [https://bugs.python.org/issue?@action=redirect&bpo=33905]: Add test for idlelib.stackview.StackBrowser.
- bpo-33924 [https://bugs.python.org/issue?@action=redirect&bpo=33924]: Change mainmenu.menudefs key ‘windows’ to ‘window’. Every other menudef key is lowercase version of main menu entry.
- bpo-33906 [https://bugs.python.org/issue?@action=redirect&bpo=33906]: Rename idlelib.windows as window Match Window on the main menu and remove last plural module name.
- bpo-33917 [https://bugs.python.org/issue?@action=redirect&bpo=33917]: Fix and document idlelib/idle_test/template.py. The revised file compiles, runs, and tests OK. idle_test/README.txt explains how to use it to create new IDLE test files.
- bpo-33904 [https://bugs.python.org/issue?@action=redirect&bpo=33904]: IDLE: In rstrip, rename class RstripExtension as Rstrip
- bpo-33907 [https://bugs.python.org/issue?@action=redirect&bpo=33907]: For consistency and clarity, rename an IDLE module and classes. Module calltips and its class CallTips are now calltip and Calltip. In module calltip_w, class CallTip is now CalltipWindow.
- bpo-33856 [https://bugs.python.org/issue?@action=redirect&bpo=33856]: Add “help” in the welcome message of IDLE
- bpo-33839 [https://bugs.python.org/issue?@action=redirect&bpo=33839]: IDLE: refactor ToolTip and CallTip and add documentation and tests
- bpo-33855 [https://bugs.python.org/issue?@action=redirect&bpo=33855]: Minimally test all IDLE modules.

Add missing files, import module, instantiate classes, and check coverage. Check existing files.

- bpo-33656 [https://bugs.python.org/issue?@action=redirect&bpo=33656]: On Windows, add API call saying that tk scales for DPI. On Windows 8.1+ or 10, with DPI compatibility properties of the Python binary unchanged, and a monitor resolution greater than 96 DPI, this should make text and lines sharper. It should otherwise have no effect.
- bpo-33768 [https://bugs.python.org/issue?@action=redirect&bpo=33768]: Clicking on a context line moves that line to the top of the editor window.
- bpo-33763 [https://bugs.python.org/issue?@action=redirect&bpo=33763]: IDLE: Use read-only text widget for code context instead of label widget.
- bpo-33664 [https://bugs.python.org/issue?@action=redirect&bpo=33664]: Scroll IDLE editor text by lines. Previously, the mouse wheel and scrollbar slider moved text by a fixed number of pixels, resulting in partial lines at the top of the editor box. The change also applies to the shell and grep output windows, but not to read-only text views.
- bpo-33679 [https://bugs.python.org/issue?@action=redirect&bpo=33679]: Enable theme-specific color configuration for Code Context. Use the Highlights tab to see the setting for built-in themes or add settings to custom themes.
- bpo-33642 [https://bugs.python.org/issue?@action=redirect&bpo=33642]: Display up to maxlines non-blank lines for Code Context. If there is no current context, show a single blank line.
- bpo-33628 [https://bugs.python.org/issue?@action=redirect&bpo=33628]: IDLE: Cleanup codecontext.py and its test.
- bpo-33564 [https://bugs.python.org/issue?@action=redirect&bpo=33564]: IDLE’s code context now

recognizes async as a block opener.

- bpo-21474 [https://bugs.python.org/issue?@action=redirect&bpo=21474]: Update word/identifier definition from ascii to unicode. In text and entry boxes, this affects selection by double-click, movement left/right by control-left/right, and deletion left/right by control-BACKSPACE/DEL.
- bpo-33204 [https://bugs.python.org/issue?@action=redirect&bpo=33204]: IDLE: consistently color invalid string prefixes. A ‘u’ string prefix cannot be paired with either ‘r’ or ‘f’. Consistently color as much of the prefix, starting at the right, as is valid. Revise and extend colorizer test.
- bpo-32984 [https://bugs.python.org/issue?@action=redirect&bpo=32984]: Set `__file__` while running a startup file. Like Python, IDLE optionally runs one startup file in the Shell window before presenting the first interactive input prompt. For IDLE, `-s` runs a file named in environmental variable **`IDLESTARTUP`** or **`PYTHONSTARTUP`**; `-r file` runs `file`. Python sets `__file__` to the startup file name before running the file and unsets it before the first prompt. IDLE now does the same when run normally, without the `-n` option.
- bpo-32940 [https://bugs.python.org/issue?@action=redirect&bpo=32940]: Simplify and rename StringTranslatePseudoMapping in pyparse.
- bpo-32916 [https://bugs.python.org/issue?@action=redirect&bpo=32916]: Change `str` to `code` in pyparse.
- bpo-32905 [https://bugs.python.org/issue?@action=redirect&bpo=32905]: Remove unused code in pyparse module.
- bpo-32874 [https://bugs.python.org/issue?@action=redirect&bpo=32874]: Add tests for pyparse.
- bpo-32837 [https://bugs.python.org/issue?@action=redirect&bpo=32837]: Using the system and place-

dependent default encoding for open() is a bad idea for IDLE’s system and location-independent files.

- bpo-32826 [https://bugs.python.org/issue?@action=redirect&bpo=32826]: Add “encoding=utf-8” to open() in IDLE’s test_help_about. GUI test test_file_buttons() only looks at initial ascii-only lines, but failed on systems where open() defaults to ‘ascii’ because readline() internally reads and decodes far enough ahead to encounter a non-ascii character in CREDITS.txt.
- bpo-32831 [https://bugs.python.org/issue?@action=redirect&bpo=32831]: Add docstrings and tests for codecontext.
- bpo-32765 [https://bugs.python.org/issue?@action=redirect&bpo=32765]: Update configdialog General tab docstring to add new widgets to the widget list.

Tools/Demos

- bpo-35884 [https://bugs.python.org/issue?@action=redirect&bpo=35884]: Add a benchmark script for timing various ways to access variables: `Tools/scripts/var_access_benchmark.py`.
- bpo-34989 [https://bugs.python.org/issue?@action=redirect&bpo=34989]: python-gdb.py now handles errors on computing the line number of a Python frame.
- bpo-20260 [https://bugs.python.org/issue?@action=redirect&bpo=20260]: Argument Clinic now has non-bitwise unsigned int converters.
- bpo-32962 [https://bugs.python.org/issue?@action=redirect&bpo=32962]: python-gdb now catches `UnicodeDecodeError` exceptions when calling `string()`.
- bpo-32962 [https://bugs.python.org/issue?@action=redirect&bpo=32962]: python-gdb now catches ValueError on read_var(): when Python has no debug symbols for example.

- bpo-33189 [https://bugs.python.org/issue?@action=redirect&bpo=33189]: **pygettext.py** now recognizes only literal strings as docstrings and translatable strings, and rejects bytes literals and f-string expressions.
- bpo-31920 [https://bugs.python.org/issue?@action=redirect&bpo=31920]: Fixed handling directories as arguments in the `pygettext` script. Based on patch by Oleg Krasnikov.
- bpo-29673 [https://bugs.python.org/issue?@action=redirect&bpo=29673]: Fix pystackv and pystack gdbinit macros.
- bpo-25427 [https://bugs.python.org/issue?@action=redirect&bpo=25427]: Remove the pyvenv script in favor of `python3 -m venv` in order to lower confusion as to what Python interpreter a virtual environment will be created for.
- bpo-32885 [https://bugs.python.org/issue?@action=redirect&bpo=32885]: Add an `-n` flag for `Tools/scripts/pathfix.py` to disable automatic backup creation (files with `~` suffix).
- bpo-32222 [https://bugs.python.org/issue?@action=redirect&bpo=32222]: Fix pygettext not extracting docstrings for functions with type annotated arguments. Patch by Toby Harradine.
- bpo-31583 [https://bugs.python.org/issue?@action=redirect&bpo=31583]: Fix 2to3 for using with –add-suffix option but without –output-dir option for relative path to files in current directory.

C API

- bpo-35713 [https://bugs.python.org/issue?@action=redirect&bpo=35713]: The **`PyByteArray_Init()`** and **`PyByteArray_Fini()`** functions have been removed. They did nothing since Python 2.7.4 and Python 3.2.0, were excluded from the limited API (stable ABI), and were not documented.

- bpo-33817 [https://bugs.python.org/issue?@action=redirect&bpo=33817]: Fixed `_PyBytes_Resize()` for empty bytes objects.
- bpo-35322 [https://bugs.python.org/issue?@action=redirect&bpo=35322]: Fix memory leak in `PyUnicode_EncodeLocale()` and `PyUnicode_EncodeFSDefault()` on error handling.
- bpo-35059 [https://bugs.python.org/issue?@action=redirect&bpo=35059]: The following C macros have been converted to static inline functions: `Py_INCREF()`, `Py_DECREF()`, `Py_XINCREF()`, `Py_XDECREF()`, `PyObject_INIT()`, `PyObject_INIT_VAR()`.
- bpo-35296 [https://bugs.python.org/issue?@action=redirect&bpo=35296]: `make install` now also installs the internal API: `Include/internal/*.h` header files.
- bpo-35081 [https://bugs.python.org/issue?@action=redirect&bpo=35081]: Internal APIs surrounded by `#ifdef Py_BUILD_CORE` have been moved from `Include/*.h` headers to new header files `Include/internal/pycore_*.h`.
- bpo-35259 [https://bugs.python.org/issue?@action=redirect&bpo=35259]: Conditionally declare `Py_FinalizeEx()` (new in 3.6) based on Py_LIMITED_API. Patch by Arthur Neufeld.
- bpo-35081 [https://bugs.python.org/issue?@action=redirect&bpo=35081]: The **`_PyObject_GC_TRACK()`** and **`_PyObject_GC_UNTRACK()`** macros have been removed from the public C API.
- bpo-35134 [https://bugs.python.org/issue?@action=redirect&bpo=35134]: Creation of a new `Include/cpython/` subdirectory.
- bpo-34725 [https://bugs.python.org/issue?@action=redirect&bpo=34725]: Adds _Py_SetProgramFullPath so embedders may override sys.executable

- bpo-34910 [https://bugs.python.org/issue?@action=redirect&bpo=34910]: Ensure that `PyObject_Print()` always returns `-1` on error. Patch by Zackery Spytz.
- bpo-34523 [https://bugs.python.org/issue?@action=redirect&bpo=34523]: Py_DecodeLocale() and Py_EncodeLocale() now use the UTF-8 encoding on Windows if Py_LegacyWindowsFSEncodingFlag is zero.
- bpo-34193 [https://bugs.python.org/issue?@action=redirect&bpo=34193]: Fix pluralization in TypeError messages in getargs.c and typeobject.c: ‘1 argument’ instead of ‘1 arguments’ and ‘1 element’ instead of ‘1 elements’.
- bpo-34127 [https://bugs.python.org/issue?@action=redirect&bpo=34127]: Return grammatically correct error message based on argument count. Patch by Karthikeyan Singaravelan.
- bpo-23927 [https://bugs.python.org/issue?@action=redirect&bpo=23927]: Fixed `SystemError` in `PyArg_ParseTupleAndKeywords()` when the `w*` format unit is used for optional parameter.
- bpo-32455 [https://bugs.python.org/issue?@action=redirect&bpo=32455]: Added `PyCompile_OpcodeStackEffectWithJump()`.
- bpo-34008 [https://bugs.python.org/issue?@action=redirect&bpo=34008]: Py_Main() can again be called after Py_Initialize(), as in Python 3.6.
- bpo-32500 [https://bugs.python.org/issue?@action=redirect&bpo=32500]: Fixed error messages for `PySequence_Size()`, `PySequence_GetItem()`, `PySequence_SetItem()` and `PySequence_DelItem()` called with a mapping and `PyMapping_Size()` called with a sequence.
- bpo-33818 [https://bugs.python.org/issue?@action=redirect&bpo=33818]: `PyExceptionClass_Name()` will now return `const char *` instead of `char *`.
- bpo-33042 [https://bugs.python.org/issue?@action=redirect&bpo=33042]: Embedding applications may

once again call PySys_ResetWarnOptions, PySys_AddWarnOption, and PySys_AddXOption prior to calling Py_Initialize.

- bpo-32374 [https://bugs.python.org/issue?@action=redirect&bpo=32374]: Document that m_traverse for multi-phase initialized modules can be called with m_state=NULL, and add a sanity check
- bpo-30863 [https://bugs.python.org/issue?@action=redirect&bpo=30863]: **`PyUnicode_AsWideChar()`** and **`PyUnicode_AsWideCharString()`** no longer cache the `wchar_t*` representation of string objects.

Python 3.7.0 final

Release date: 2018-06-27

Library

- bpo-33851 [https://bugs.python.org/issue?@action=redirect&bpo=33851]: Fix **`ast.get_docstring()`** for a node that lacks a docstring.

C API

- bpo-33932 [https://bugs.python.org/issue?@action=redirect&bpo=33932]: Calling Py_Initialize() twice does nothing, instead of failing with a fatal error: restore the Python 3.6 behaviour.

Python 3.7.0 release candidate 1

Release date: 2018-06-12

Core and Builtins

- bpo-33803 [https://bugs.python.org/issue?@action=redirect&bpo=33803]: Fix a crash in hamt.c caused by enabling GC tracking for an object that hadn’t all of its fields set to NULL.
- bpo-33706 [https://bugs.python.org/issue?@action=redirect&bpo=33706]: Fix a crash in Python initialization when parsing the command line options. Thanks Christoph Gohlke for the bug report and the fix!
- bpo-30654 [https://bugs.python.org/issue?@action=redirect&bpo=30654]: Fixed reset of the SIGINT handler to SIG_DFL on interpreter shutdown even when there was a custom handler set previously. Patch by Philipp Kerling.
- bpo-31849 [https://bugs.python.org/issue?@action=redirect&bpo=31849]: Fix signed/unsigned comparison warning in pyhash.c.

Library

- bpo-30167 [https://bugs.python.org/issue?@action=redirect&bpo=30167]: Prevent site.main() exception if PYTHONSTARTUP is set. Patch by Steve Weber.
- bpo-33812 [https://bugs.python.org/issue?@action=redirect&bpo=33812]: Datetime instance d with non-None tzinfo, but with d.tzinfo.utcoffset(d) returning None is now treated as naive by the astimezone() method.
- bpo-30805 [https://bugs.python.org/issue?@action=redirect&bpo=30805]: Avoid race condition with debug logging
- bpo-33694 [https://bugs.python.org/issue?@action=redirect&bpo=33694]: asyncio: Fix a race condition causing data loss on pause_reading()/resume_reading() when using the ProactorEventLoop.
- bpo-32493 [https://bugs.python.org/issue?@action=redirect&bpo=32493]: Correct test for `uuid_enc_be` availability in `configure.ac`. Patch by Michael Felt.

- bpo-33792 [https://bugs.python.org/issue?@action=redirect&bpo=33792]: Add asyncio.WindowsSelectorEventLoopPolicy and asyncio.WindowsProactorEventLoopPolicy.
- bpo-33778 [https://bugs.python.org/issue?@action=redirect&bpo=33778]: Update `unicodedata`'s database to Unicode version 11.0.0.
- bpo-33770 [https://bugs.python.org/issue?@action=redirect&bpo=33770]: improve base64 exception message for encoded inputs of invalid length
- bpo-33769 [https://bugs.python.org/issue?@action=redirect&bpo=33769]: asyncio/start_tls: Fix error message; cancel callbacks in case of an unhandled error; mark SSLTransport as closed if it is aborted.
- bpo-33767 [https://bugs.python.org/issue?@action=redirect&bpo=33767]: The concatenation (+) and repetition (*) sequence operations now raise **`TypeError`** instead of **`SystemError`** when performed on **`mmap.mmap`** objects. Patch by Zackery Spytz.
- bpo-33734 [https://bugs.python.org/issue?@action=redirect&bpo=33734]: asyncio/ssl: Fix AttributeError, increase default handshake timeout
- bpo-11874 [https://bugs.python.org/issue?@action=redirect&bpo=11874]: Use a better regex when breaking usage into wrappable parts. Avoids bogus assertion errors from custom metavar strings.
- bpo-33582 [https://bugs.python.org/issue?@action=redirect&bpo=33582]: Emit a deprecation warning for inspect.formatargspec

Documentation

- bpo-33409 [https://bugs.python.org/issue?@action=redirect&bpo=33409]: Clarified the relationship between **PEP 538** [https://peps.python.org/pep-0538/]'s

PYTHONCOERCECLOCALE and PEP 540’s PYTHONUTF8 mode.
- bpo-33736 [https://bugs.python.org/issue?@action=redirect&bpo=33736]: Improve the documentation of `asyncio.open_connection()`, `asyncio.start_server()` and their UNIX socket counterparts.
- bpo-31432 [https://bugs.python.org/issue?@action=redirect&bpo=31432]: Clarify meaning of CERT_NONE, CERT_OPTIONAL, and CERT_REQUIRED flags for ssl.SSLContext.verify_mode.

Build

- bpo-5755 [https://bugs.python.org/issue?@action=redirect&bpo=5755]: Move `-Wstrict-prototypes` option to `CFLAGS_NODIST` from `OPT`. This option emitted annoying warnings when building extension modules written in C++.

Windows

- bpo-33720 [https://bugs.python.org/issue?@action=redirect&bpo=33720]: Reduces maximum marshal recursion depth on release builds.

IDLE

- bpo-33656 [https://bugs.python.org/issue?@action=redirect&bpo=33656]: On Windows, add API call saying that tk scales for DPI. On Windows 8.1+ or 10, with DPI compatibility properties of the Python binary unchanged, and a monitor resolution greater than 96 DPI, this should make text and lines sharper. It should otherwise have no effect.
- bpo-33768 [https://bugs.python.org/issue?@action=redirect&bpo=33768]: Clicking on a context line moves that line to the top of the editor window.

- bpo-33763 [https://bugs.python.org/issue?@action=redirect&bpo=33763]: IDLE: Use read-only text widget for code context instead of label widget.
- bpo-33664 [https://bugs.python.org/issue?@action=redirect&bpo=33664]: Scroll IDLE editor text by lines. Previously, the mouse wheel and scrollbar slider moved text by a fixed number of pixels, resulting in partial lines at the top of the editor box. The change also applies to the shell and grep output windows, but not to read-only text views.
- bpo-33679 [https://bugs.python.org/issue?@action=redirect&bpo=33679]: Enable theme-specific color configuration for Code Context. Use the Highlights tab to see the setting for built-in themes or add settings to custom themes.
- bpo-33642 [https://bugs.python.org/issue?@action=redirect&bpo=33642]: Display up to maxlines non-blank lines for Code Context. If there is no current context, show a single blank line.

Python 3.7.0 beta 5

Release date: 2018-05-30

Core and Builtins

- bpo-33622 [https://bugs.python.org/issue?@action=redirect&bpo=33622]: Fixed a leak when the garbage collector fails to add an object with the `__del__` method or referenced by it into the **`gc.garbage`** list. **`PyGC_Collect()`** can now be called when an exception is set and preserves it.
- bpo-33509 [https://bugs.python.org/issue?@action=redirect&bpo=33509]: Fix module_globals parameter

of warnings.warn_explicit(): don’t crash if module_globals is not a dict.

- bpo-20104 [https://bugs.python.org/issue?@action=redirect&bpo=20104]: The new `os.posix_spawn` added in 3.7.0b1 was removed as we are still working on what the API should look like. Expect this in 3.8 instead.
- bpo-33475 [https://bugs.python.org/issue?@action=redirect&bpo=33475]: Fixed miscellaneous bugs in converting annotations to strings and optimized parentheses in the string representation.
- bpo-33391 [https://bugs.python.org/issue?@action=redirect&bpo=33391]: Fix a leak in set_symmetric_difference().
- bpo-28055 [https://bugs.python.org/issue?@action=redirect&bpo=28055]: Fix unaligned accesses in siphash24(). Patch by Rolf Eike Beer.
- bpo-32911 [https://bugs.python.org/issue?@action=redirect&bpo=32911]: Due to unexpected compatibility issues discovered during downstream beta testing, reverted bpo-29463 [https://bugs.python.org/issue?@action=redirect&bpo=29463]. `docstring` field is removed from Module, ClassDef, FunctionDef, and AsyncFunctionDef ast nodes which was added in 3.7a1. Docstring expression is restored as a first statement in their body. Based on patch by Inada Naoki.
- bpo-21983 [https://bugs.python.org/issue?@action=redirect&bpo=21983]: Fix a crash in `ctypes.cast()` in case the type argument is a ctypes structured data type. Patch by Eryk Sun and Oren Milman.

Library

- bpo-32751 [https://bugs.python.org/issue?@action=redirect&bpo=32751]: When cancelling the task due to a timeout, **`asyncio.wait_for()`** will now wait until the cancellation is complete.

- bpo-32684 [https://bugs.python.org/issue?@action=redirect&bpo=32684]: Fix gather to propagate cancellation of itself even with return_exceptions.
- bpo-33654 [https://bugs.python.org/issue?@action=redirect&bpo=33654]: Support protocol type switching in SSLTransport.set_protocol().
- bpo-33674 [https://bugs.python.org/issue?@action=redirect&bpo=33674]: Pause the transport as early as possible to further reduce the risk of data_received() being called before connection_made().
- bpo-33674 [https://bugs.python.org/issue?@action=redirect&bpo=33674]: Fix a race condition in SSLProtocol.connection_made() of asyncio.sslproto: start immediately the handshake instead of using call_soon(). Previously, data_received() could be called before the handshake started, causing the handshake to hang or fail.
- bpo-31647 [https://bugs.python.org/issue?@action=redirect&bpo=31647]: Fixed bug where calling write_eof() on a _SelectorSocketTransport after it’s already closed raises AttributeError.
- bpo-32610 [https://bugs.python.org/issue?@action=redirect&bpo=32610]: Make asyncio.all_tasks() return only pending tasks.
- bpo-32410 [https://bugs.python.org/issue?@action=redirect&bpo=32410]: Avoid blocking on file IO in sendfile fallback code
- bpo-33469 [https://bugs.python.org/issue?@action=redirect&bpo=33469]: Fix RuntimeError after closing loop that used run_in_executor
- bpo-33672 [https://bugs.python.org/issue?@action=redirect&bpo=33672]: Fix Task.__repr__ crash with Cython’s bogus coroutines
- bpo-33654 [https://bugs.python.org/issue?@action=redirect&bpo=33654]: Fix transport.set_protocol() to support switching between asyncio.Protocol and

asyncio.BufferedProtocol. Fix loop.start_tls() to work with asyncio.BufferedProtocols.

- bpo-33652 [https://bugs.python.org/issue?@action=redirect&bpo=33652]: Pickles of type variables and subscripted generics are now future-proof and compatible with older Python versions.
- bpo-32493 [https://bugs.python.org/issue?@action=redirect&bpo=32493]: Fixed **`uuid.uuid1()`** on FreeBSD.
- bpo-33618 [https://bugs.python.org/issue?@action=redirect&bpo=33618]: Finalize and document preliminary and experimental TLS 1.3 support with OpenSSL 1.1.1
- bpo-33623 [https://bugs.python.org/issue?@action=redirect&bpo=33623]: Fix possible SIGSGV when asyncio.Future is created in __del__
- bpo-30877 [https://bugs.python.org/issue?@action=redirect&bpo=30877]: Fixed a bug in the Python implementation of the JSON decoder that prevented the cache of parsed strings from clearing after finishing the decoding. Based on patch by c-fos.
- bpo-33570 [https://bugs.python.org/issue?@action=redirect&bpo=33570]: Change TLS 1.3 cipher suite settings for compatibility with OpenSSL 1.1.1-pre6 and newer. OpenSSL 1.1.1 will have TLS 1.3 ciphers enabled by default.
- bpo-28556 [https://bugs.python.org/issue?@action=redirect&bpo=28556]: Do not simplify arguments to `typing.Union`. Now `Union[Manager, Employee]` is not simplified to `Employee` at runtime. Such simplification previously caused several bugs and limited possibilities for introspection.
- bpo-33540 [https://bugs.python.org/issue?@action=redirect&bpo=33540]: Add a new `block_on_close` class attribute to `ForkingMixIn` and `ThreadingMixIn` classes of **`socketserver`**.

- bpo-33548 [https://bugs.python.org/issue?@action=redirect&bpo=33548]: tempfile._candidate_tempdir_list should consider common TEMP locations
- bpo-33109 [https://bugs.python.org/issue?@action=redirect&bpo=33109]: argparse subparsers are once again not required by default, reverting the change in behavior introduced by bpo-26510 [https://bugs.python.org/issue?@action=redirect&bpo=26510] in 3.7.0a2.
- bpo-33536 [https://bugs.python.org/issue?@action=redirect&bpo=33536]: dataclasses.make_dataclass now checks for invalid field names and duplicate fields. Also, added a check for invalid field specifications.
- bpo-33542 [https://bugs.python.org/issue?@action=redirect&bpo=33542]: Prevent `uuid.get_node` from using a DUID instead of a MAC on Windows. Patch by Zvi Effron
- bpo-26819 [https://bugs.python.org/issue?@action=redirect&bpo=26819]: Fix race condition with `ReadTransport.resume_reading` in Windows proactor event loop.
- Fix failure in `typing.get_type_hints()` when ClassVar was provided as a string forward reference.
- bpo-33505 [https://bugs.python.org/issue?@action=redirect&bpo=33505]: Optimize asyncio.ensure_future() by reordering if checks: 1.17x faster.
- bpo-33497 [https://bugs.python.org/issue?@action=redirect&bpo=33497]: Add errors param to cgi.parse_multipart and make an encoding in FieldStorage use the given errors (needed for Twisted). Patch by Amber Brown.
- bpo-33495 [https://bugs.python.org/issue?@action=redirect&bpo=33495]: Change dataclasses.Fields repr to use the repr of each of its members, instead of str.

This makes it more clear what each field actually represents. This is especially true for the ‘type’ member.

- bpo-33453 [https://bugs.python.org/issue?@action=redirect&bpo=33453]: Fix dataclasses to work if using literal string type annotations or if using PEP 563 “Postponed Evaluation of Annotations”. Only specific string prefixes are detected for both ClassVar (“ClassVar” and “typing.ClassVar”) and InitVar (“InitVar” and “dataclasses.InitVar”).
- bpo-28556 [https://bugs.python.org/issue?@action=redirect&bpo=28556]: Minor fixes in typing module: add annotations to `NamedTuple.__new__`, pass `*args` and `**kwds` in `Generic.__new__`. Original PRs by Paulius Šarka and Chad Dombrova.
- bpo-20087 [https://bugs.python.org/issue?@action=redirect&bpo=20087]: Updated alias mapping with glibc 2.27 supported locales.
- bpo-33422 [https://bugs.python.org/issue?@action=redirect&bpo=33422]: Fix trailing quotation marks getting deleted when looking up byte/string literals on pydoc. Patch by Andrés Delfino.
- bpo-28167 [https://bugs.python.org/issue?@action=redirect&bpo=28167]: The function `platform.linux_distribution` and `platform.dist` now trigger a `DeprecationWarning` and have been marked for removal in Python 3.8
- bpo-33197 [https://bugs.python.org/issue?@action=redirect&bpo=33197]: Update error message when constructing invalid inspect.Parameters Patch by Donghee Na.
- bpo-33263 [https://bugs.python.org/issue?@action=redirect&bpo=33263]: Fix FD leak in `_SelectorSocketTransport` Patch by Vlad Starostin.
- bpo-32861 [https://bugs.python.org/issue?@action=redirect&bpo=32861]: The urllib.robotparser’s `__str__` representation now includes wildcard entries and the

“Crawl-delay” and “Request-rate” fields. Patch by Michael Lazar.
- bpo-32257 [https://bugs.python.org/issue?@action=redirect&bpo=32257]: The ssl module now contains OP_NO_RENEGOTIATION constant, available with OpenSSL 1.1.0h or 1.1.1.
- bpo-16865 [https://bugs.python.org/issue?@action=redirect&bpo=16865]: Support arrays >=2GiB in **`ctypes`**. Patch by Segev Finer.

Documentation

- bpo-23859 [https://bugs.python.org/issue?@action=redirect&bpo=23859]: Document that `asyncio.wait()` does not cancel its futures on timeout.
- bpo-32436 [https://bugs.python.org/issue?@action=redirect&bpo=32436]: Document **PEP 567** [https://peps.python.org/pep-0567/] changes to asyncio.
- bpo-33604 [https://bugs.python.org/issue?@action=redirect&bpo=33604]: Update HMAC md5 default to a DeprecationWarning, bump removal to 3.8.
- bpo-33503 [https://bugs.python.org/issue?@action=redirect&bpo=33503]: Fix broken pypi link
- bpo-33421 [https://bugs.python.org/issue?@action=redirect&bpo=33421]: Add missing documentation for `typing.AsyncContextManager`.

Tests

- bpo-33655 [https://bugs.python.org/issue?@action=redirect&bpo=33655]: Ignore test_posix_fallocate failures on BSD platforms that might be due to running on ZFS.
- bpo-32604 [https://bugs.python.org/issue?@action=redirect&bpo=32604]: Remove the _xxsubinterpreters

module (meant for testing) and associated helpers. This module was originally added recently in 3.7b1.

Build

- bpo-33614 [https://bugs.python.org/issue?@action=redirect&bpo=33614]: Ensures module definition files for the stable ABI on Windows are correctly regenerated.
- bpo-33522 [https://bugs.python.org/issue?@action=redirect&bpo=33522]: Enable CI builds on Visual Studio Team Services at https://python.visualstudio.com/cpython
- bpo-33012 [https://bugs.python.org/issue?@action=redirect&bpo=33012]: Add `-Wno-cast-function-type` for gcc 8 for silencing warnings about function casts like casting to PyCFunction in method definition lists.

macOS

- bpo-13631 [https://bugs.python.org/issue?@action=redirect&bpo=13631]: The .editrc file in user’s home directory is now processed correctly during the readline initialization through editline emulation on macOS.

IDLE

- bpo-33628 [https://bugs.python.org/issue?@action=redirect&bpo=33628]: IDLE: Cleanup codecontext.py and its test.
- bpo-33564 [https://bugs.python.org/issue?@action=redirect&bpo=33564]: IDLE’s code context now recognizes async as a block opener.
- bpo-32831 [https://bugs.python.org/issue?@action=redirect&bpo=32831]: Add docstrings and tests for codecontext.

Python 3.7.0 beta 4

Release date: 2018-05-02

Core and Builtins

- bpo-33363 [https://bugs.python.org/issue?@action=redirect&bpo=33363]: Raise a SyntaxError for `async with` and `async for` statements outside of async functions.
- bpo-33128 [https://bugs.python.org/issue?@action=redirect&bpo=33128]: Fix a bug that causes PathFinder to appear twice on sys.meta_path. Patch by Pablo Galindo Salgado.
- bpo-33312 [https://bugs.python.org/issue?@action=redirect&bpo=33312]: Fixed clang ubsan (undefined behavior sanitizer) warnings in dictobject.c by adjusting how the internal struct _dictkeysobject shared keys structure is declared.
- bpo-33231 [https://bugs.python.org/issue?@action=redirect&bpo=33231]: Fix potential memory leak in `normalizestring()`.
- bpo-33205 [https://bugs.python.org/issue?@action=redirect&bpo=33205]: Change dict growth function from `round_up_to_power_2(used*2+hashtable_size/2)` to `round_up_to_power_2(used*3)`. Previously, dict is shrunk only when `used == 0`. Now dict has more chance to be shrunk.
- bpo-29922 [https://bugs.python.org/issue?@action=redirect&bpo=29922]: Improved error messages in ‘async with’ when `__aenter__()` or `__aexit__()` return non-awaitable object.
- bpo-33199 [https://bugs.python.org/issue?@action=redirect&bpo=33199]: Fix `ma_version_tag` in dict implementation is uninitialized when copying from key-sharing dict.

Library

- bpo-33281 [https://bugs.python.org/issue?@action=redirect&bpo=33281]: Fix ctypes.util.find_library regression on macOS.
- bpo-33383 [https://bugs.python.org/issue?@action=redirect&bpo=33383]: Fixed crash in the get() method of the `dbm.ndbm` database object when it is called with a single argument.
- bpo-33329 [https://bugs.python.org/issue?@action=redirect&bpo=33329]: Fix multiprocessing regression on newer glibcs
- bpo-991266 [https://bugs.python.org/issue?@action=redirect&bpo=991266]: Fix quoting of the `Comment` attribute of `http.cookies.SimpleCookie`.
- bpo-33131 [https://bugs.python.org/issue?@action=redirect&bpo=33131]: Upgrade bundled version of pip to 10.0.1.
- bpo-33308 [https://bugs.python.org/issue?@action=redirect&bpo=33308]: Fixed a crash in the **`parser`** module when converting an ST object to a tree of tuples or lists with `line_info=False` and `col_info=True`.
- bpo-33266 [https://bugs.python.org/issue?@action=redirect&bpo=33266]: lib2to3 now recognizes `rf'...'` strings.
- bpo-11594 [https://bugs.python.org/issue?@action=redirect&bpo=11594]: Ensure line-endings are respected when using lib2to3.
- bpo-33254 [https://bugs.python.org/issue?@action=redirect&bpo=33254]: Have `importlib.resources.contents()` and `importlib.abc.ResourceReader.contents()` return an iterable instead of an iterator.
- bpo-33256 [https://bugs.python.org/issue?@action=redirect&bpo=33256]: Fix display of `<module>` call in the

html produced by `cgitb.html()`. Patch by Stéphane Blondon.

- bpo-33185 [https://bugs.python.org/issue?@action=redirect&bpo=33185]: Fixed regression when running pydoc with the **`-m`** switch. (The regression was introduced in 3.7.0b3 by the resolution of bpo-33053 [https://bugs.python.org/issue?@action=redirect&bpo=33053]) This fix also changed pydoc to add `os.getcwd()` to **`sys.path`** when necessary, rather than adding `"."`.
- bpo-33169 [https://bugs.python.org/issue?@action=redirect&bpo=33169]: Delete entries of `None` in **`sys.path_importer_cache`** when **`importlib.machinery.invalidate_caches()`** is called.
- bpo-33217 [https://bugs.python.org/issue?@action=redirect&bpo=33217]: Deprecate looking up non-Enum objects in Enum classes and Enum members (will raise **`TypeError`** in 3.8+).
- bpo-33203 [https://bugs.python.org/issue?@action=redirect&bpo=33203]: `random.Random.choice()` now raises `IndexError` for empty sequences consistently even when called from subclasses without a `getrandbits()` implementation.
- bpo-33224 [https://bugs.python.org/issue?@action=redirect&bpo=33224]: Update difflib.mdiff() for **PEP 479** [https://peps.python.org/pep-0479/]. Convert an uncaught StopIteration in a generator into a return-statement.
- bpo-33209 [https://bugs.python.org/issue?@action=redirect&bpo=33209]: End framing at the end of C implementation of **`pickle.Pickler.dump()`**.
- bpo-20104 [https://bugs.python.org/issue?@action=redirect&bpo=20104]: Improved error handling and fixed a reference leak in **`os.posix_spawn()`**.
- bpo-33175 [https://bugs.python.org/issue?@action=redirect&bpo=33175]: In dataclasses, Field.__set_name__ now looks up the __set_name__

special method on the class, not the instance, of the default value.
- bpo-33097 [https://bugs.python.org/issue?@action=redirect&bpo=33097]: Raise RuntimeError when `executor.submit` is called during interpreter shutdown.
- bpo-31908 [https://bugs.python.org/issue?@action=redirect&bpo=31908]: Fix output of cover files for `trace` module command-line tool. Previously emitted cover files only when `--missing` option was used. Patch by Michael Selik.

Documentation

- bpo-33378 [https://bugs.python.org/issue?@action=redirect&bpo=33378]: Add Korean language switcher for https://docs.python.org/3/
- bpo-33276 [https://bugs.python.org/issue?@action=redirect&bpo=33276]: Clarify that the `__path__` attribute on modules cannot be just any value.
- bpo-33201 [https://bugs.python.org/issue?@action=redirect&bpo=33201]: Modernize documentation for writing C extension types.
- bpo-33195 [https://bugs.python.org/issue?@action=redirect&bpo=33195]: Deprecate `Py_UNICODE` usage in `c-api/arg` document. `Py_UNICODE` related APIs are deprecated since Python 3.3, but it is missed in the document.
- bpo-8243 [https://bugs.python.org/issue?@action=redirect&bpo=8243]: Add a note about curses.addch and curses.addstr exception behavior when writing outside a window, or pad.
- bpo-32337 [https://bugs.python.org/issue?@action=redirect&bpo=32337]: Update documentation related with `dict` order.

Tests

- bpo-33358 [https://bugs.python.org/issue?@action=redirect&bpo=33358]: Fix `test_embed.test_pre_initialization_sys_options()` when the interpreter is built with `--enable-shared`.

Build

- bpo-33394 [https://bugs.python.org/issue?@action=redirect&bpo=33394]: Enable the verbose build for extension modules, when GNU make is passed macros on the command line.
- bpo-33393 [https://bugs.python.org/issue?@action=redirect&bpo=33393]: Update config.guess and config.sub files.
- bpo-33377 [https://bugs.python.org/issue?@action=redirect&bpo=33377]: Add new triplets for mips r6 and riscv variants (used in extension suffixes).
- bpo-32232 [https://bugs.python.org/issue?@action=redirect&bpo=32232]: By default, modules configured in `Modules/Setup` are no longer built with `-DPy_BUILD_CORE`. Instead, modules that specifically need that preprocessor definition include it in their individual entries.
- bpo-33182 [https://bugs.python.org/issue?@action=redirect&bpo=33182]: The embedding tests can once again be built with clang 6.0

Windows

- bpo-33184 [https://bugs.python.org/issue?@action=redirect&bpo=33184]: Update Windows installer to use OpenSSL 1.1.0h.

macOS

- bpo-33184 [https://bugs.python.org/issue?@action=redirect&bpo=33184]: Update macOS installer build to use OpenSSL 1.1.0h.

IDLE

- bpo-21474 [https://bugs.python.org/issue?@action=redirect&bpo=21474]: Update word/identifier definition from ascii to unicode. In text and entry boxes, this affects selection by double-click, movement left/right by control-left/right, and deletion left/right by control-BACKSPACE/DEL.
- bpo-33204 [https://bugs.python.org/issue?@action=redirect&bpo=33204]: IDLE: consistently color invalid string prefixes. A ‘u’ string prefix cannot be paired with either ‘r’ or ‘f’. Consistently color as much of the prefix, starting at the right, as is valid. Revise and extend colorizer test.

Tools/Demos

- bpo-33189 [https://bugs.python.org/issue?@action=redirect&bpo=33189]: **pygettext.py** now recognizes only literal strings as docstrings and translatable strings, and rejects bytes literals and f-string expressions.
- bpo-31920 [https://bugs.python.org/issue?@action=redirect&bpo=31920]: Fixed handling directories as arguments in the `pygettext` script. Based on patch by Oleg Krasnikov.
- bpo-29673 [https://bugs.python.org/issue?@action=redirect&bpo=29673]: Fix pystackv and pystack gdbinit macros.

- bpo-31583 [https://bugs.python.org/issue?@action=redirect&bpo=31583]: Fix 2to3 for using with –add-suffix option but without –output-dir option for relative path to files in current directory.

Python 3.7.0 beta 3

Release date: 2018-03-29

Security

- bpo-33136 [https://bugs.python.org/issue?@action=redirect&bpo=33136]: Harden ssl module against LibreSSL **CVE 2018-8970** [https://www.cve.org/CVERecord?id=CVE-2018-8970]. X509_VERIFY_PARAM_set1_host() is called with an explicit namelen. A new test ensures that NULL bytes are not allowed.
- bpo-33001 [https://bugs.python.org/issue?@action=redirect&bpo=33001]: Minimal fix to prevent buffer overrun in os.symlink on Windows
- bpo-32981 [https://bugs.python.org/issue?@action=redirect&bpo=32981]: Regexes in difflib and poplib were vulnerable to catastrophic backtracking. These regexes formed potential DOS vectors (REDOS). They have been refactored. This resolves **CVE 2018-1060** [https://www.cve.org/CVERecord?id=CVE-2018-1060] and **CVE 2018-1061** [https://www.cve.org/CVERecord?id=CVE-2018-1061]. Patch by Jamie Davis.

Core and Builtins

- bpo-33053 [https://bugs.python.org/issue?@action=redirect&bpo=33053]: When using the -m switch, sys.path[0] is now explicitly expanded as the *starting* working directory, rather than being left as the empty path (which allows imports from the current working directory at the time of the import)

- bpo-33018 [https://bugs.python.org/issue?@action=redirect&bpo=33018]: Improve consistency of errors raised by `issubclass()` when called with a non-class and an abstract base class as the first and second arguments, respectively. Patch by Josh Bronson.
- bpo-33041 [https://bugs.python.org/issue?@action=redirect&bpo=33041]: Fixed jumping when the function contains an `async for` loop.
- bpo-33026 [https://bugs.python.org/issue?@action=redirect&bpo=33026]: Fixed jumping out of "with" block by setting f_lineno.
- bpo-33005 [https://bugs.python.org/issue?@action=redirect&bpo=33005]: Fix a crash on fork when using a custom memory allocator (ex: using PYTHONMALLOC env var). _PyGILState_Reinit() and _PyInterpreterState_Enable() now use the default RAW memory allocator to allocate a new interpreters mutex on fork.
- bpo-17288 [https://bugs.python.org/issue?@action=redirect&bpo=17288]: Prevent jumps from 'return' and 'exception' trace events.
- bpo-32836 [https://bugs.python.org/issue?@action=redirect&bpo=32836]: Don't use temporary variables in cases of list/dict/set comprehensions

Library

- bpo-33141 [https://bugs.python.org/issue?@action=redirect&bpo=33141]: Have Field objects pass through __set_name__ to their default values, if they have their own __set_name__.
- bpo-33096 [https://bugs.python.org/issue?@action=redirect&bpo=33096]: Allow ttk.Treeview.insert to

insert iid that has a false boolean value. Note iid=0 and iid=False would be same. Patch by Garvit Khatri.

- bpo-32873 [https://bugs.python.org/issue?@action=redirect&bpo=32873]: Treat type variables and special typing forms as immutable by copy and pickle. This fixes several minor issues and inconsistencies, and improves backwards compatibility with Python 3.6.
- bpo-33134 [https://bugs.python.org/issue?@action=redirect&bpo=33134]: When computing dataclass’s __hash__, use the lookup table to contain the function which returns the __hash__ value. This is an improvement over looking up a string, and then testing that string to see what to do.
- bpo-33127 [https://bugs.python.org/issue?@action=redirect&bpo=33127]: The ssl module now compiles with LibreSSL 2.7.1.
- bpo-32505 [https://bugs.python.org/issue?@action=redirect&bpo=32505]: Raise TypeError if a member variable of a dataclass is of type Field, but doesn’t have a type annotation.
- bpo-33078 [https://bugs.python.org/issue?@action=redirect&bpo=33078]: Fix the failure on OSX caused by the tests relying on sem_getvalue
- bpo-33116 [https://bugs.python.org/issue?@action=redirect&bpo=33116]: Add ‘Field’ to dataclasses.__all__.
- bpo-32896 [https://bugs.python.org/issue?@action=redirect&bpo=32896]: Fix an error where subclassing a dataclass with a field that uses a default_factory would generate an incorrect class.
- bpo-33100 [https://bugs.python.org/issue?@action=redirect&bpo=33100]: Dataclasses: If a field has a default value that’s a MemberDescriptorType, then it’s from that field being in __slots__, not an actual default value.

- bpo-32953 [https://bugs.python.org/issue?@action=redirect&bpo=32953]: If a non-dataclass inherits from a frozen dataclass, allow attributes to be added to the derived class. Only attributes from the frozen dataclass cannot be assigned to. Require all dataclasses in a hierarchy to be either all frozen or all non-frozen.
- bpo-33061 [https://bugs.python.org/issue?@action=redirect&bpo=33061]: Add missing `NoReturn` to `__all__` in typing.py
- bpo-33078 [https://bugs.python.org/issue?@action=redirect&bpo=33078]: Fix the size handling in multiprocessing.Queue when a pickling error occurs.
- bpo-33064 [https://bugs.python.org/issue?@action=redirect&bpo=33064]: lib2to3 now properly supports trailing commas after `*args` and `**kwargs` in function signatures.
- bpo-33056 [https://bugs.python.org/issue?@action=redirect&bpo=33056]: FIX properly close leaking fds in concurrent.futures.ProcessPoolExecutor.
- bpo-33021 [https://bugs.python.org/issue?@action=redirect&bpo=33021]: Release the GIL during fstat() calls, avoiding hang of all threads when calling mmap.mmap(), os.urandom(), and random.seed(). Patch by Nir Soffer.
- bpo-31804 [https://bugs.python.org/issue?@action=redirect&bpo=31804]: Avoid failing in multiprocessing.Process if the standard streams are closed or None at exit.
- bpo-33037 [https://bugs.python.org/issue?@action=redirect&bpo=33037]: Skip sending/receiving data after SSL transport closing.
- bpo-27683 [https://bugs.python.org/issue?@action=redirect&bpo=27683]: Fix a regression in **`ipaddress`** that result of **`hosts()`** is empty when the network is constructed by a tuple containing an integer mask and only 1 bit left for addresses.

- bpo-32999 [https://bugs.python.org/issue?@action=redirect&bpo=32999]: Fix C implementation of `ABC.__subclasscheck__(cls, subclass)` crashed when `subclass` is not a type object.
- bpo-33009 [https://bugs.python.org/issue?@action=redirect&bpo=33009]: Fix inspect.signature() for single-parameter partialmethods.
- bpo-32969 [https://bugs.python.org/issue?@action=redirect&bpo=32969]: Expose several missing constants in zlib and fix corresponding documentation.
- bpo-32056 [https://bugs.python.org/issue?@action=redirect&bpo=32056]: Improved exceptions raised for invalid number of channels and sample width when read an audio file in modules **`aifc`**, **`wave`** and **`sunau`**.
- bpo-32844 [https://bugs.python.org/issue?@action=redirect&bpo=32844]: Fix wrong redirection of a low descriptor (0 or 1) to stderr in subprocess if another low descriptor is closed.
- bpo-32857 [https://bugs.python.org/issue?@action=redirect&bpo=32857]: In **`tkinter`**, `after_cancel(None)` now raises a **`ValueError`** instead of canceling the first scheduled function. Patch by Cheryl Sabella.
- bpo-31639 [https://bugs.python.org/issue?@action=redirect&bpo=31639]: http.server now exposes a ThreadedHTTPServer class and uses it when the module is run with `-m` to cope with web browsers pre-opening sockets.
- bpo-27645 [https://bugs.python.org/issue?@action=redirect&bpo=27645]: **`sqlite3.Connection`** now exposes a **`backup`** method, if the underlying SQLite library is at version 3.6.11 or higher. Patch by Lele Gaifax.

Documentation

- bpo-33126 [https://bugs.python.org/issue?@action=redirect&bpo=33126]: Document

PyBuffer_ToContiguous().

- bpo-27212 [https://bugs.python.org/issue?@action=redirect&bpo=27212]: Modify documentation for the **`islice()`** recipe to consume initial values up to the start index.
- bpo-28247 [https://bugs.python.org/issue?@action=redirect&bpo=28247]: Update **`zipapp`** documentation to describe how to make standalone applications.
- bpo-18802 [https://bugs.python.org/issue?@action=redirect&bpo=18802]: Documentation changes for ipaddress. Patch by Jon Foster and Berker Peksag.
- bpo-27428 [https://bugs.python.org/issue?@action=redirect&bpo=27428]: Update documentation to clarify that `WindowsRegistryFinder` implements `MetaPathFinder`. (Patch by Himanshu Lakhara)

Tests

- bpo-32872 [https://bugs.python.org/issue?@action=redirect&bpo=32872]: Avoid regrtest compatibility issue with namespace packages.
- bpo-32517 [https://bugs.python.org/issue?@action=redirect&bpo=32517]: Fix failing `test_asyncio` on macOS 10.12.2+ due to transport of `KqueueSelector` loop was not being closed.
- bpo-19417 [https://bugs.python.org/issue?@action=redirect&bpo=19417]: Add test_bdb.py.

Build

- bpo-33163 [https://bugs.python.org/issue?@action=redirect&bpo=33163]: Upgrade pip to 9.0.3 and setuptools to v39.0.1.

Windows

- bpo-33016 [https://bugs.python.org/issue?@action=redirect&bpo=33016]: Fix potential use of uninitialized memory in nt._getfinalpathname
- bpo-32903 [https://bugs.python.org/issue?@action=redirect&bpo=32903]: Fix a memory leak in os.chdir() on Windows if the current directory is set to a UNC path.

macOS

- bpo-32726 [https://bugs.python.org/issue?@action=redirect&bpo=32726]: Build and link with private copy of Tcl/Tk 8.6 for the macOS 10.6+ installer. The 10.9+ installer variant already does this. This means that the Python 3.7 provided by the python.org macOS installers no longer need or use any external versions of Tcl/Tk, either system-provided or user-installed, such as ActiveTcl.

IDLE

- bpo-32984 [https://bugs.python.org/issue?@action=redirect&bpo=32984]: Set `__file__` while running a startup file. Like Python, IDLE optionally runs one startup file in the Shell window before presenting the first interactive input prompt. For IDLE, `-s` runs a file named in environmental variable **`IDLESTARTUP`** or **`PYTHONSTARTUP`**; `-r file` runs `file`. Python sets `__file__` to the startup file name before running the file and unsets it before the first prompt. IDLE now does the same when run normally, without the `-n` option.
- bpo-32940 [https://bugs.python.org/issue?@action=redirect&bpo=32940]: Simplify and rename StringTranslatePseudoMapping in pyparse.

Tools/Demos

- bpo-32885 [https://bugs.python.org/issue?@action=redirect&bpo=32885]: Add an `-n` flag for `Tools/scripts/pathfix.py` to disable automatic backup creation (files with ~ suffix).

C API

- bpo-33042 [https://bugs.python.org/issue?@action=redirect&bpo=33042]: Embedding applications may once again call PySys_ResetWarnOptions, PySys_AddWarnOption, and PySys_AddXOption prior to calling Py_Initialize.
- bpo-32374 [https://bugs.python.org/issue?@action=redirect&bpo=32374]: Document that m_traverse for multi-phase initialized modules can be called with m_state=NULL, and add a sanity check

Python 3.7.0 beta 2

Release date: 2018-02-27

Security

- bpo-28414 [https://bugs.python.org/issue?@action=redirect&bpo=28414]: The ssl module now allows users to perform their own IDN en/decoding when using SNI.

Core and Builtins

- bpo-32889 [https://bugs.python.org/issue?@action=redirect&bpo=32889]: Update Valgrind suppression list to account for the rename of `Py_ADDRESS_IN_RANG` to `address_in_range`.

- bpo-31356 [https://bugs.python.org/issue?@action=redirect&bpo=31356]: Remove the new API added in bpo-31356 [https://bugs.python.org/issue?@action=redirect&bpo=31356] (gc.ensure_disabled() context manager).
- bpo-32305 [https://bugs.python.org/issue?@action=redirect&bpo=32305]: For namespace packages, ensure that both `__file__` and `__spec__.origin` are set to None.
- bpo-32303 [https://bugs.python.org/issue?@action=redirect&bpo=32303]: Make sure `__spec__.loader` matches `__loader__` for namespace packages.
- bpo-32711 [https://bugs.python.org/issue?@action=redirect&bpo=32711]: Fix the warning messages for Python/ast_unparse.c. Patch by Stéphane Wirtel
- bpo-32583 [https://bugs.python.org/issue?@action=redirect&bpo=32583]: Fix possible crashing in builtin Unicode decoders caused by write out-of-bound errors when using customized decode error handlers.

Library

- bpo-32960 [https://bugs.python.org/issue?@action=redirect&bpo=32960]: For dataclasses, disallow inheriting frozen from non-frozen classes, and also disallow inheriting non-frozen from frozen classes. This restriction will be relaxed at a future date.
- bpo-32713 [https://bugs.python.org/issue?@action=redirect&bpo=32713]: Fixed tarfile.itn handling of out-of-bounds float values. Patch by Joffrey Fuhrer.
- bpo-32951 [https://bugs.python.org/issue?@action=redirect&bpo=32951]: Direct instantiation of SSLSocket and SSLObject objects is now prohibited. The constructors were never documented, tested, or designed as public constructors. Users were suppose to use ssl.wrap_socket() or SSLContext.

- bpo-32929 [https://bugs.python.org/issue?@action=redirect&bpo=32929]: Remove the tri-state parameter “hash”, and add the boolean “unsafe_hash”. If unsafe_hash is True, add a __hash__ function, but if a __hash__ exists, raise TypeError. If unsafe_hash is False, add a __hash__ based on the values of eq= and frozen=. The unsafe_hash=False behavior is the same as the old hash=None behavior. unsafe_hash=False is the default, just as hash=None used to be.
- bpo-32947 [https://bugs.python.org/issue?@action=redirect&bpo=32947]: Add OP_ENABLE_MIDDLEBOX_COMPAT and test workaround for TLSv1.3 for future compatibility with OpenSSL 1.1.1.
- bpo-30622 [https://bugs.python.org/issue?@action=redirect&bpo=30622]: The ssl module now detects missing NPN support in LibreSSL.
- bpo-32922 [https://bugs.python.org/issue?@action=redirect&bpo=32922]: dbm.open() now encodes filename with the filesystem encoding rather than default encoding.
- bpo-32859 [https://bugs.python.org/issue?@action=redirect&bpo=32859]: In `os.dup2`, don’t check every call whether the `dup3` syscall exists or not.
- bpo-32556 [https://bugs.python.org/issue?@action=redirect&bpo=32556]: nt._getfinalpathname, nt._getvolumepathname and nt._getdiskusage now correctly convert from bytes.
- bpo-25988 [https://bugs.python.org/issue?@action=redirect&bpo=25988]: Emit a **`DeprecationWarning`** when using or importing an ABC directly from **`collections`** rather than from **`collections.abc`**.
- bpo-21060 [https://bugs.python.org/issue?@action=redirect&bpo=21060]: Rewrite confusing message from setup.py upload from “No dist file created in earlier command” to the more helpful “Must create and upload files in one command”.

- bpo-32852 [https://bugs.python.org/issue?@action=redirect&bpo=32852]: Make sure sys.argv remains as a list when running trace.
- bpo-31333 [https://bugs.python.org/issue?@action=redirect&bpo=31333]: `_abc` module is added. It is a speedup module with C implementations for various functions and methods in `abc`. Creating an ABC subclass and calling `isinstance` or `issubclass` with an ABC subclass are up to 1.5x faster. In addition, this makes Python start-up up to 10% faster. Note that the new implementation hides internal registry and caches, previously accessible via private attributes `_abc_registry`, `_abc_cache`, and `_abc_negative_cache`. There are three debugging helper methods that can be used instead `_dump_registry`, `_abc_registry_clear`, and `_abc_caches_clear`.
- bpo-32841 [https://bugs.python.org/issue?@action=redirect&bpo=32841]: Fixed `asyncio.Condition` issue which silently ignored cancellation after notifying and cancelling a conditional lock. Patch by Bar Harel.
- bpo-32819 [https://bugs.python.org/issue?@action=redirect&bpo=32819]: ssl.match_hostname() has been simplified and no longer depends on re and ipaddress module for wildcard and IP addresses. Error reporting for invalid wildcards has been improved.
- bpo-32394 [https://bugs.python.org/issue?@action=redirect&bpo=32394]: socket: Remove TCP_FASTOPEN,TCP_KEEPCNT,TCP_KEEPIDLE,TCP_KEEPINTVL flags on older version Windows during run-time.
- bpo-31787 [https://bugs.python.org/issue?@action=redirect&bpo=31787]: Fixed refleaks of `__init__()` methods in various modules. (Contributed by Oren Milman)
- bpo-30157 [https://bugs.python.org/issue?@action=redirect&bpo=30157]: Fixed guessing quote and delimiter in csv.Sniffer.sniff() when only the last field is quoted. Patch by Jake Davis.

- bpo-32792 [https://bugs.python.org/issue?@action=redirect&bpo=32792]: collections.ChainMap() preserves the order of the underlying mappings.
- bpo-32775 [https://bugs.python.org/issue?@action=redirect&bpo=32775]: **`fnmatch.translate()`** no longer produces patterns which contain set operations. Sets starting with ‘[’ or containing ‘–’, ‘&&’, ‘~~’ or ‘||’ will be interpreted differently in regular expressions in future versions. Currently they emit warnings. fnmatch.translate() now avoids producing patterns containing such sets by accident.
- bpo-32622 [https://bugs.python.org/issue?@action=redirect&bpo=32622]: Implement native fast sendfile for Windows proactor event loop.
- bpo-32777 [https://bugs.python.org/issue?@action=redirect&bpo=32777]: Fix a rare but potential pre-exec child process deadlock in subprocess on POSIX systems when marking file descriptors inheritable on exec in the child process. This bug appears to have been introduced in 3.4.
- bpo-32647 [https://bugs.python.org/issue?@action=redirect&bpo=32647]: The ctypes module used to depend on indirect linking for dlopen. The shared extension is now explicitly linked against libdl on platforms with dl.
- bpo-32741 [https://bugs.python.org/issue?@action=redirect&bpo=32741]: Implement `asyncio.TimerHandle.when()` method.
- bpo-32691 [https://bugs.python.org/issue?@action=redirect&bpo=32691]: Use mod_spec.parent when running modules with pdb
- bpo-32734 [https://bugs.python.org/issue?@action=redirect&bpo=32734]: Fixed `asyncio.Lock()` safety issue which allowed acquiring and locking the same lock multiple times, without it being free. Patch by Bar Harel.

- bpo-32727 [https://bugs.python.org/issue?@action=redirect&bpo=32727]: Do not include name field in SMTP envelope from address. Patch by Stéphane Wirtel
- bpo-31453 [https://bugs.python.org/issue?@action=redirect&bpo=31453]: Add TLSVersion constants and SSLContext.maximum_version / minimum_version attributes. The new API wraps OpenSSL 1.1 https://web.archive.org/web/20180309043602/https://www.openssl.org/docs/man1.1.0/ssl/SSL_CTX_set_min_proto_version.html feature.
- bpo-24334 [https://bugs.python.org/issue?@action=redirect&bpo=24334]: Internal implementation details of ssl module were cleaned up. The SSLSocket has one less layer of indirection. Owner and session information are now handled by the SSLSocket and SSLObject constructor. Channel binding implementation has been simplified.
- bpo-31848 [https://bugs.python.org/issue?@action=redirect&bpo=31848]: Fix the error handling in Aifc_read.initfp() when the SSND chunk is not found. Patch by Zackery Spytz.
- bpo-32585 [https://bugs.python.org/issue?@action=redirect&bpo=32585]: Add Ttk spinbox widget to **`tkinter.ttk`**. Patch by Alan D Moore.
- bpo-32221 [https://bugs.python.org/issue?@action=redirect&bpo=32221]: Various functions returning tuple containing IPv6 addresses now omit `%scope` part since the same information is already encoded in *scopeid* tuple item. Especially this speeds up **`socket.recvfrom()`** when it receives multicast packet since useless resolving of network interface name is omitted.
- bpo-30693 [https://bugs.python.org/issue?@action=redirect&bpo=30693]: The TarFile class now recurses directories in a reproducible way.
- bpo-30693 [https://bugs.python.org/issue?@action=redirect&bpo=30693]: The ZipFile class now recurses

directories in a reproducible way.

Documentation

- bpo-28124 [https://bugs.python.org/issue?@action=redirect&bpo=28124]: The ssl module function ssl.wrap_socket() has been de-emphasized and deprecated in favor of the more secure and efficient SSLContext.wrap_socket() method.
- bpo-17232 [https://bugs.python.org/issue?@action=redirect&bpo=17232]: Clarify docs for -O and -OO. Patch by Terry Reedy.
- bpo-32436 [https://bugs.python.org/issue?@action=redirect&bpo=32436]: Add documentation for the contextvars module (PEP 567).
- bpo-32800 [https://bugs.python.org/issue?@action=redirect&bpo=32800]: Update link to w3c doc for xml default namespaces.
- bpo-11015 [https://bugs.python.org/issue?@action=redirect&bpo=11015]: Update **`test.support`** documentation.
- bpo-8722 [https://bugs.python.org/issue?@action=redirect&bpo=8722]: Document **`__getattr__()`** behavior when property **`get()`** method raises **`AttributeError`**.
- bpo-32614 [https://bugs.python.org/issue?@action=redirect&bpo=32614]: Modify RE examples in documentation to use raw strings to prevent **`DeprecationWarning`** and add text to REGEX HOWTO to highlight the deprecation.
- bpo-31972 [https://bugs.python.org/issue?@action=redirect&bpo=31972]: Improve docstrings for `pathlib.PurePath` subclasses.

Tests

- bpo-31809 [https://bugs.python.org/issue?@action=redirect&bpo=31809]: Add tests to verify connection with secp ECDH curves.

Build

- bpo-32898 [https://bugs.python.org/issue?@action=redirect&bpo=32898]: Fix the python debug build when using COUNT_ALLOCS.

Windows

- bpo-32901 [https://bugs.python.org/issue?@action=redirect&bpo=32901]: Update Tcl and Tk versions to 8.6.8
- bpo-31966 [https://bugs.python.org/issue?@action=redirect&bpo=31966]: Fixed WindowsConsoleIO.write() for writing empty data.
- bpo-32409 [https://bugs.python.org/issue?@action=redirect&bpo=32409]: Ensures activate.bat can handle Unicode contents.
- bpo-32457 [https://bugs.python.org/issue?@action=redirect&bpo=32457]: Improves handling of denormalized executable path when launching Python.
- bpo-32370 [https://bugs.python.org/issue?@action=redirect&bpo=32370]: Use the correct encoding for ipconfig output in the uuid module. Patch by Segev Finer.
- bpo-29248 [https://bugs.python.org/issue?@action=redirect&bpo=29248]: Fix **`os.readlink()`** on Windows, which was mistakenly treating the `PrintNameOffset` field of the reparse data buffer as a number of characters instead of bytes. Patch by Craig Holmquist and SSE4.

macOS

- bpo-32901 [https://bugs.python.org/issue?@action=redirect&bpo=32901]: Update macOS 10.9+ installer to Tcl/Tk 8.6.8.

IDLE

- bpo-32916 [https://bugs.python.org/issue?@action=redirect&bpo=32916]: Change `str` to `code` in pyparse.
- bpo-32905 [https://bugs.python.org/issue?@action=redirect&bpo=32905]: Remove unused code in pyparse module.
- bpo-32874 [https://bugs.python.org/issue?@action=redirect&bpo=32874]: Add tests for pyparse.
- bpo-32837 [https://bugs.python.org/issue?@action=redirect&bpo=32837]: Using the system and place-dependent default encoding for open() is a bad idea for IDLE’s system and location-independent files.
- bpo-32826 [https://bugs.python.org/issue?@action=redirect&bpo=32826]: Add “encoding=utf-8” to open() in IDLE’s test_help_about. GUI test test_file_buttons() only looks at initial ascii-only lines, but failed on systems where open() defaults to ‘ascii’ because readline() internally reads and decodes far enough ahead to encounter a non-ascii character in CREDITS.txt.
- bpo-32765 [https://bugs.python.org/issue?@action=redirect&bpo=32765]: Update configdialog General tab docstring to add new widgets to the widget list.

Tools/Demos

- bpo-32222 [https://bugs.python.org/issue?@action=redirect&bpo=32222]: Fix pygettext not extracting docstrings for functions with type annotated arguments. Patch by Toby Harradine.

Python 3.7.0 beta 1

Release date: 2018-01-30

Core and Builtins

- bpo-32703 [https://bugs.python.org/issue?@action=redirect&bpo=32703]: Fix coroutine's ResourceWarning when there's an active error set when it's being finalized.
- bpo-32650 [https://bugs.python.org/issue?@action=redirect&bpo=32650]: Pdb and other debuggers dependent on bdb.py will correctly step over (next command) native coroutines. Patch by Pablo Galindo.
- bpo-28685 [https://bugs.python.org/issue?@action=redirect&bpo=28685]: Optimize list.sort() and sorted() by using type specialized comparisons when possible.
- bpo-32685 [https://bugs.python.org/issue?@action=redirect&bpo=32685]: Improve suggestion when the Python 2 form of print statement is either present on the same line as the header of a compound statement or else terminated by a semi-colon instead of a newline. Patch by Nitish Chandra.
- bpo-32697 [https://bugs.python.org/issue?@action=redirect&bpo=32697]: Python now explicitly preserves the definition order of keyword-only parameters. It's always preserved their order, but this behavior was never guaranteed before; this behavior is now guaranteed and tested.
- bpo-32690 [https://bugs.python.org/issue?@action=redirect&bpo=32690]: The locals() dictionary now displays in the lexical order that variables were defined. Previously, the order was reversed.
- bpo-32677 [https://bugs.python.org/issue?@action=redirect&bpo=32677]: Add `.isascii()` method to `str`,

`bytes` and `bytearray`. It can be used to test that string contains only ASCII characters.

- bpo-32670 [https://bugs.python.org/issue?@action=redirect&bpo=32670]: Enforce **PEP 479** [https://peps.python.org/pep-0479/] for all code. This means that manually raising a StopIteration exception from a generator is prohibited for all code, regardless of whether 'from __future__ import generator_stop' was used or not.
- bpo-32591 [https://bugs.python.org/issue?@action=redirect&bpo=32591]: Added built-in support for tracking the origin of coroutine objects; see sys.set_coroutine_origin_tracking_depth and CoroutineType.cr_origin. This replaces the asyncio debug mode's use of coroutine wrapping for native coroutine objects.
- bpo-31368 [https://bugs.python.org/issue?@action=redirect&bpo=31368]: Expose preadv and pwritev system calls in the os module. Patch by Pablo Galindo
- bpo-32544 [https://bugs.python.org/issue?@action=redirect&bpo=32544]: `hasattr(obj, name)` and `getattr(obj, name, default)` are about 4 times faster than before when `name` is not found and `obj` doesn't override `__getattr__` or `__getattribute__`.
- bpo-26163 [https://bugs.python.org/issue?@action=redirect&bpo=26163]: Improved frozenset() hash to create more distinct hash values when faced with datasets containing many similar values.
- bpo-32550 [https://bugs.python.org/issue?@action=redirect&bpo=32550]: Remove the STORE_ANNOTATION bytecode.
- bpo-20104 [https://bugs.python.org/issue?@action=redirect&bpo=20104]: Expose posix_spawn as a low level API in the os module. (removed before 3.7.0rc1)
- bpo-24340 [https://bugs.python.org/issue?@action=redirect&bpo=24340]: Fixed estimation of the code

stack size.

- bpo-32436 [https://bugs.python.org/issue?@action=redirect&bpo=32436]: Implement **PEP 567** [https://peps.python.org/pep-0567/] Context Variables.
- bpo-18533 [https://bugs.python.org/issue?@action=redirect&bpo=18533]: `repr()` on a dict containing its own `values()` or `items()` no longer raises `RecursionError`; OrderedDict similarly. Instead, use `...`, as for other recursive structures. Patch by Ben North.
- bpo-20891 [https://bugs.python.org/issue?@action=redirect&bpo=20891]: Py_Initialize() now creates the GIL. The GIL is no longer created “on demand” to fix a race condition when PyGILState_Ensure() is called in a non-Python thread.
- bpo-32028 [https://bugs.python.org/issue?@action=redirect&bpo=32028]: Leading whitespace is now correctly ignored when generating suggestions for converting Py2 print statements to Py3 builtin print function calls. Patch by Sanyam Khurana.
- bpo-31179 [https://bugs.python.org/issue?@action=redirect&bpo=31179]: Make dict.copy() up to 5.5 times faster.
- bpo-31113 [https://bugs.python.org/issue?@action=redirect&bpo=31113]: Get rid of recursion in the compiler for normal control flow.

Library

- bpo-25988 [https://bugs.python.org/issue?@action=redirect&bpo=25988]: Deprecate exposing the contents of collections.abc in the regular collections module.
- bpo-31429 [https://bugs.python.org/issue?@action=redirect&bpo=31429]: The default cipher suite selection of the ssl module now uses a blacklist approach rather than a hard-coded whitelist. Python no

longer re-enables ciphers that have been blocked by OpenSSL security update. Default cipher suite selection can be configured on compile time.

- bpo-30306 [https://bugs.python.org/issue?@action=redirect&bpo=30306]: contextlib.contextmanager now releases the arguments passed to the underlying generator as soon as the context manager is entered. Previously it would keep them alive for as long as the context manager was alive, even when not being used as a function decorator. Patch by Martin Teichmann.
- bpo-21417 [https://bugs.python.org/issue?@action=redirect&bpo=21417]: Added support for setting the compression level for zipfile.ZipFile.
- bpo-32251 [https://bugs.python.org/issue?@action=redirect&bpo=32251]: Implement asyncio.BufferedProtocol (provisional API).
- bpo-32513 [https://bugs.python.org/issue?@action=redirect&bpo=32513]: In dataclasses, allow easier overriding of dunder methods without specifying decorator parameters.
- bpo-32660 [https://bugs.python.org/issue?@action=redirect&bpo=32660]: **`termios`** makes available `FIONREAD`, `FIONCLEX`, `FIOCLEX`, `FIOASYNC` and `FIONBIO` also under Solaris/derivatives.
- bpo-27931 [https://bugs.python.org/issue?@action=redirect&bpo=27931]: Fix email address header parsing error when the username is an empty quoted string. Patch by Xiang Zhang.
- bpo-32659 [https://bugs.python.org/issue?@action=redirect&bpo=32659]: Under Solaris and derivatives, **`os.stat_result`** provides a st_fstype attribute.
- bpo-32662 [https://bugs.python.org/issue?@action=redirect&bpo=32662]: Implement Server.start_serving(), Server.serve_forever(), and Server.is_serving() methods. Add ‘start_serving’

keyword parameter to loop.create_server() and loop.create_unix_server().

- bpo-32391 [https://bugs.python.org/issue?@action=redirect&bpo=32391]: Implement `asyncio.StreamWriter.wait_closed()` and `asyncio.StreamWriter.is_closing()` methods
- bpo-32643 [https://bugs.python.org/issue?@action=redirect&bpo=32643]: Make Task._step, Task._wakeup and Future._schedule_callbacks methods private.
- bpo-32630 [https://bugs.python.org/issue?@action=redirect&bpo=32630]: Refactor decimal module to use contextvars to store decimal context.
- bpo-32622 [https://bugs.python.org/issue?@action=redirect&bpo=32622]: Add **`asyncio.AbstractEventLoop.sendfile()`** method.
- bpo-32304 [https://bugs.python.org/issue?@action=redirect&bpo=32304]: distutils' upload command no longer corrupts tar files ending with a CR byte, and no longer tries to convert CR to CRLF in any of the upload text fields.
- bpo-32502 [https://bugs.python.org/issue?@action=redirect&bpo=32502]: uuid.uuid1 no longer raises an exception if a 64-bit hardware address is encountered.
- bpo-32596 [https://bugs.python.org/issue?@action=redirect&bpo=32596]: `concurrent.futures` imports `ThreadPoolExecutor` and `ProcessPoolExecutor` lazily (using **PEP 562** [https://peps.python.org/pep-0562/]). It makes `import asyncio` about 15% faster because asyncio uses only `ThreadPoolExecutor` by default.
- bpo-31801 [https://bugs.python.org/issue?@action=redirect&bpo=31801]: Add `_ignore_` to `Enum` so temporary variables can be used during class construction without being turned into members.
- bpo-32576 [https://bugs.python.org/issue?@action=redirect&bpo=32576]: Use queue.SimpleQueue() in places where it can be invoked from a weakref callback.

- bpo-32574 [https://bugs.python.org/issue?@action=redirect&bpo=32574]: Fix memory leak in asyncio.Queue, when the queue has limited size and it is full, the cancellation of queue.put() can cause a memory leak. Patch by: José Melero.
- bpo-32521 [https://bugs.python.org/issue?@action=redirect&bpo=32521]: The nis module is now compatible with new libnsl and headers location.
- bpo-32467 [https://bugs.python.org/issue?@action=redirect&bpo=32467]: collections.abc.ValuesView now inherits from collections.abc.Collection.
- bpo-32473 [https://bugs.python.org/issue?@action=redirect&bpo=32473]: Improve ABCMeta._dump_registry() output readability
- bpo-32102 [https://bugs.python.org/issue?@action=redirect&bpo=32102]: New argument `capture_output` for subprocess.run
- bpo-32521 [https://bugs.python.org/issue?@action=redirect&bpo=32521]: glibc has removed Sun RPC. Use replacement libtirpc headers and library in nis module.
- bpo-32493 [https://bugs.python.org/issue?@action=redirect&bpo=32493]: UUID module fixes build for FreeBSD/OpenBSD
- bpo-32503 [https://bugs.python.org/issue?@action=redirect&bpo=32503]: Pickling with protocol 4 no longer creates too small frames.
- bpo-29237 [https://bugs.python.org/issue?@action=redirect&bpo=29237]: Create enum for pstats sorting options
- bpo-32454 [https://bugs.python.org/issue?@action=redirect&bpo=32454]: Add close(fd) function to the socket module.
- bpo-25942 [https://bugs.python.org/issue?@action=redirect&bpo=25942]: The subprocess module is now more graceful when handling a Ctrl-C KeyboardInterrupt

during subprocess.call, subprocess.run, or a Popen context manager. It now waits a short amount of time for the child (presumed to have also gotten the SIGINT) to exit, before continuing the KeyboardInterrupt exception handling. This still includes a SIGKILL in the call() and run() APIs, but at least the child had a chance first.

- bpo-32433 [https://bugs.python.org/issue?@action=redirect&bpo=32433]: The hmac module now has hmac.digest(), which provides an optimized HMAC digest.
- bpo-28134 [https://bugs.python.org/issue?@action=redirect&bpo=28134]: Sockets now auto-detect family, type and protocol from file descriptor by default.
- bpo-32404 [https://bugs.python.org/issue?@action=redirect&bpo=32404]: Fix bug where **`datetime.datetime.fromtimestamp()`** did not call __new__ in **`datetime.datetime`** subclasses.
- bpo-32403 [https://bugs.python.org/issue?@action=redirect&bpo=32403]: Improved speed of **`datetime.date`** and **`datetime.datetime`** alternate constructors.
- bpo-32228 [https://bugs.python.org/issue?@action=redirect&bpo=32228]: Ensure that `truncate()` preserves the file position (as reported by `tell()`) after writes longer than the buffer size.
- bpo-32410 [https://bugs.python.org/issue?@action=redirect&bpo=32410]: Implement `loop.sock_sendfile` for asyncio event loop.
- bpo-22908 [https://bugs.python.org/issue?@action=redirect&bpo=22908]: Added seek and tell to the ZipExtFile class. This only works if the file object used to open the zipfile is seekable.
- bpo-32373 [https://bugs.python.org/issue?@action=redirect&bpo=32373]: Add socket.getblocking() method.

- bpo-32248 [https://bugs.python.org/issue?@action=redirect&bpo=32248]: Add **`importlib.resources`** and **`importlib.abc.ResourceReader`** as the unified API for reading resources contained within packages. Loaders wishing to support resource reading must implement the **`get_resource_reader()`** method. File-based and zipimport-based loaders both implement these APIs. **`importlib.abc.ResourceLoader`** is deprecated in favor of these new APIs.
- bpo-32320 [https://bugs.python.org/issue?@action=redirect&bpo=32320]: collections.namedtuple() now supports default values.
- bpo-29302 [https://bugs.python.org/issue?@action=redirect&bpo=29302]: Add contextlib.AsyncExitStack. Patch by Alexander Mohr and Ilya Kulakov.
- bpo-31961 [https://bugs.python.org/issue?@action=redirect&bpo=31961]: *Removed in Python 3.7.0b2.* The *args* argument of subprocess.Popen can now be a path-like object. If *args* is given as a sequence, it’s first element can now be a path-like object as well.
- bpo-31900 [https://bugs.python.org/issue?@action=redirect&bpo=31900]: The **`locale.localeconv()`** function now sets temporarily the `LC_CTYPE` locale to the `LC_NUMERIC` locale to decode `decimal_point` and `thousands_sep` byte strings if they are non-ASCII or longer than 1 byte, and the `LC_NUMERIC` locale is different than the `LC_CTYPE` locale. This temporary change affects other threads. Same change for the **`str.format()`** method when formatting a number (**`int`**, **`float`**, **`float`** and subclasses) with the `n` type (ex: `'{:n}'.format(1234)`).
- bpo-31853 [https://bugs.python.org/issue?@action=redirect&bpo=31853]: Use super().method instead of socket.method in SSLSocket. They were there most likely for legacy reasons.
- bpo-31399 [https://bugs.python.org/issue?@action=redirect&bpo=31399]: The ssl module now uses

OpenSSL’s X509_VERIFY_PARAM_set1_host() and X509_VERIFY_PARAM_set1_ip() API to verify hostname and IP addresses. Subject common name fallback can be disabled with SSLContext.hostname_checks_common_name.
- bpo-14976 [https://bugs.python.org/issue?@action=redirect&bpo=14976]: Add a queue.SimpleQueue class, an unbounded FIFO queue with a reentrant C implementation of put().

Documentation

- bpo-32724 [https://bugs.python.org/issue?@action=redirect&bpo=32724]: Add references to some commands in the documentation of Pdb. Patch by Stéphane Wirtel
- bpo-32649 [https://bugs.python.org/issue?@action=redirect&bpo=32649]: Complete the C API documentation, profiling and tracing part with the newly added per-opcode events.
- bpo-17799 [https://bugs.python.org/issue?@action=redirect&bpo=17799]: Explain real behaviour of sys.settrace and sys.setprofile and their C-API counterparts regarding which type of events are received in each function. Patch by Pablo Galindo Salgado.

Tests

- bpo-32721 [https://bugs.python.org/issue?@action=redirect&bpo=32721]: Fix test_hashlib to not fail if the _md5 module is not built.
- bpo-28414 [https://bugs.python.org/issue?@action=redirect&bpo=28414]: Add test cases for IDNA 2003 and 2008 host names. IDNA 2003 internationalized host names are working since bpo-31399

[https://bugs.python.org/issue?@action=redirect&bpo=31399] has landed. IDNA 2008 are still broken.
- bpo-32604 [https://bugs.python.org/issue?@action=redirect&bpo=32604]: Add a new "_xxsubinterpreters" extension module that exposes the existing subinterpreter C-API and a new cross-interpreter data sharing mechanism. The module is primarily intended for more thorough testing of the existing subinterpreter support. Note that the _xxsubinterpreters module has been removed in 3.7.0rc1.
- bpo-32602 [https://bugs.python.org/issue?@action=redirect&bpo=32602]: Add test certs and test for ECDSA cert and EC/RSA dual mode.
- bpo-32549 [https://bugs.python.org/issue?@action=redirect&bpo=32549]: On Travis CI, Python now Compiles and uses a local copy of OpenSSL 1.1.0g for testing.

Build

- bpo-32635 [https://bugs.python.org/issue?@action=redirect&bpo=32635]: Fix segfault of the crypt module when libxcrypt is provided instead of libcrypt at the system.
- bpo-32598 [https://bugs.python.org/issue?@action=redirect&bpo=32598]: Use autoconf to detect OpenSSL libs, headers and supported features. The ax_check_openssl M4 macro uses pkg-config to locate OpenSSL and falls back to manual search.
- bpo-32593 [https://bugs.python.org/issue?@action=redirect&bpo=32593]: Drop support of FreeBSD 9 and older.
- bpo-29708 [https://bugs.python.org/issue?@action=redirect&bpo=29708]: If the **`SOURCE_DATE_EPOCH`** environment variable is set, `py_compile` will always create hash-based `.pyc` files.

Windows

- bpo-32588 [https://bugs.python.org/issue?@action=redirect&bpo=32588]: Create standalone _distutils_findvs module and add missing _queue module to installer.
- bpo-29911 [https://bugs.python.org/issue?@action=redirect&bpo=29911]: Ensure separate Modify and Uninstall buttons are displayed.
- bpo-32507 [https://bugs.python.org/issue?@action=redirect&bpo=32507]: Use app-local UCRT install rather than the proper update for old versions of Windows.

macOS

- bpo-32726 [https://bugs.python.org/issue?@action=redirect&bpo=32726]: Provide an additional, more modern macOS installer variant that supports macOS 10.9+ systems in 64-bit mode only. Upgrade the supplied third-party libraries to OpenSSL 1.1.0g and to SQLite 3.22.0. The 10.9+ installer now links with and supplies its own copy of Tcl/Tk 8.6.
- bpo-28440 [https://bugs.python.org/issue?@action=redirect&bpo=28440]: No longer add /Library/Python/3.x/site-packages to sys.path for macOS framework builds to avoid future conflicts.

C API

- bpo-32681 [https://bugs.python.org/issue?@action=redirect&bpo=32681]: Fix uninitialized variable ‘res’ in the C implementation of os.dup2. Patch by Stéphane Wirtel

- bpo-10381 [https://bugs.python.org/issue?@action=redirect&bpo=10381]: Add C API access to the `datetime.timezone` constructor and `datetime.timezone.UTC` singleton.

Python 3.7.0 alpha 4

Release date: 2018-01-08

Core and Builtins

- bpo-31975 [https://bugs.python.org/issue?@action=redirect&bpo=31975]: The default warning filter list now starts with a “default::DeprecationWarning:__main__” entry, so deprecation warnings are once again shown by default in single-file scripts and at the interactive prompt.
- bpo-32226 [https://bugs.python.org/issue?@action=redirect&bpo=32226]: `__class_getitem__` is now an automatic class method.
- bpo-32399 [https://bugs.python.org/issue?@action=redirect&bpo=32399]: Add AIX uuid library support for RFC4122 using uuid_create() in libc.a
- bpo-32390 [https://bugs.python.org/issue?@action=redirect&bpo=32390]: Fix the compilation failure on AIX after the f_fsid field has been added to the object returned by os.statvfs() (bpo-32143 [https://bugs.python.org/issue?@action=redirect&bpo=32143]). Original patch by Michael Felt.
- bpo-32379 [https://bugs.python.org/issue?@action=redirect&bpo=32379]: Make MRO computation faster when a class inherits from a single base.
- bpo-32259 [https://bugs.python.org/issue?@action=redirect&bpo=32259]: The error message of a

TypeError raised when unpack non-iterable is now more specific.

- bpo-27169 [https://bugs.python.org/issue?@action=redirect&bpo=27169]: The `__debug__` constant is now optimized out at compile time. This fixes also bpo-22091 [https://bugs.python.org/issue?@action=redirect&bpo=22091].
- bpo-32329 [https://bugs.python.org/issue?@action=redirect&bpo=32329]: The **`-R`** option now turns on hash randomization when the **`PYTHONHASHSEED`** environment variable is set to `0`. Previously, the option was ignored. Moreover, `sys.flags.hash_randomization` is now properly set to 0 when hash randomization is turned off by `PYTHONHASHSEED=0`.
- bpo-30416 [https://bugs.python.org/issue?@action=redirect&bpo=30416]: The optimizer is now protected from spending much time doing complex calculations and consuming much memory for creating large constants in constant folding. Increased limits for constants that can be produced in constant folding.
- bpo-32282 [https://bugs.python.org/issue?@action=redirect&bpo=32282]: Fix an unnecessary ifdef in the include of VersionHelpers.h in socketmodule on Windows.
- bpo-30579 [https://bugs.python.org/issue?@action=redirect&bpo=30579]: Implement TracebackType.__new__ to allow Python-level creation of traceback objects, and make TracebackType.tb_next mutable.
- bpo-32260 [https://bugs.python.org/issue?@action=redirect&bpo=32260]: Don’t byte swap the input keys to the SipHash algorithm on big-endian platforms. This should ensure siphash gives consistent results across platforms.
- bpo-31506 [https://bugs.python.org/issue?@action=redirect&bpo=31506]: Improve the error message

logic for object.__new__ and object.__init__. Patch by Sanyam Khurana.

- bpo-20361 [https://bugs.python.org/issue?@action=redirect&bpo=20361]: `-b` and `-bb` now inject `'default::BytesWarning'` and `error::BytesWarning` entries into `sys.warnoptions`, ensuring that they take precedence over any other warning filters configured via the `-W` option or the `PYTHONWARNINGS` environment variable.
- bpo-32230 [https://bugs.python.org/issue?@action=redirect&bpo=32230]: `-X dev` now injects a `'default'` entry into sys.warnoptions, ensuring that it behaves identically to actually passing `-Wdefault` at the command line.
- bpo-29240 [https://bugs.python.org/issue?@action=redirect&bpo=29240]: Add a new UTF-8 mode: implementation of the **PEP 540** [https://peps.python.org/pep-0540/].
- bpo-32226 [https://bugs.python.org/issue?@action=redirect&bpo=32226]: **PEP 560** [https://peps.python.org/pep-0560/]: Add support for `__mro_entries__` and `__class_getitem__`. Implemented by Ivan Levkivskyi.
- bpo-32225 [https://bugs.python.org/issue?@action=redirect&bpo=32225]: **PEP 562** [https://peps.python.org/pep-0562/]: Add support for module `__getattr__` and `__dir__`. Implemented by Ivan Levkivskyi.
- bpo-31901 [https://bugs.python.org/issue?@action=redirect&bpo=31901]: The `atexit` module now has its callback stored per interpreter.
- bpo-31650 [https://bugs.python.org/issue?@action=redirect&bpo=31650]: Implement **PEP 552** [https://peps.python.org/pep-0552/] (Deterministic pycs). Python now supports invalidating bytecode cache files bashed on a source content hash rather than source last-modified time.
- bpo-29469 [https://bugs.python.org/issue?@action=redirect&bpo=29469]: Move constant folding from

bytecode layer to AST layer. Original patch by Eugene Toder.

Library

- bpo-32506 [https://bugs.python.org/issue?@action=redirect&bpo=32506]: Now that dict is defined as keeping insertion order, drop OrderedDict and just use plain dict.
- bpo-32279 [https://bugs.python.org/issue?@action=redirect&bpo=32279]: Add params to dataclasses.make_dataclasses(): init, repr, eq, order, hash, and frozen. Pass them through to dataclass().
- bpo-32278 [https://bugs.python.org/issue?@action=redirect&bpo=32278]: Make type information optional on dataclasses.make_dataclass(). If omitted, the string 'typing.Any' is used.
- bpo-32499 [https://bugs.python.org/issue?@action=redirect&bpo=32499]: Add dataclasses.is_dataclass(obj), which returns True if obj is a dataclass or an instance of one.
- bpo-32468 [https://bugs.python.org/issue?@action=redirect&bpo=32468]: Improve frame repr() to mention filename, code name and current line number.
- bpo-23749 [https://bugs.python.org/issue?@action=redirect&bpo=23749]: asyncio: Implement loop.start_tls()
- bpo-32441 [https://bugs.python.org/issue?@action=redirect&bpo=32441]: Return the new file descriptor (i.e., the second argument) from `os.dup2`. Previously, `None` was always returned.
- bpo-32422 [https://bugs.python.org/issue?@action=redirect&bpo=32422]: `functools.lru_cache` uses less memory (3 words for each cached key) and takes about 1/3 time for cyclic GC.

- bpo-31721 [https://bugs.python.org/issue?@action=redirect&bpo=31721]: Prevent Python crash from happening when Future._log_traceback is set to True manually. Now it can only be set to False, or a ValueError is raised.
- bpo-32415 [https://bugs.python.org/issue?@action=redirect&bpo=32415]: asyncio: Add Task.get_loop() and Future.get_loop()
- bpo-26133 [https://bugs.python.org/issue?@action=redirect&bpo=26133]: Don’t unsubscribe signals in asyncio UNIX event loop on interpreter shutdown.
- bpo-32363 [https://bugs.python.org/issue?@action=redirect&bpo=32363]: Make asyncio.Task.set_exception() and set_result() raise NotImplementedError. Task._step() and Future.__await__() raise proper exceptions when they are in an invalid state, instead of raising an AssertionError.
- bpo-32357 [https://bugs.python.org/issue?@action=redirect&bpo=32357]: Optimize asyncio.iscoroutine() and loop.create_task() for non-native coroutines (e.g. async/await compiled with Cython). ‘loop.create_task(python_coroutine)’ used to be 20% faster than ‘loop.create_task(cython_coroutine)’. Now, the latter is as fast.
- bpo-32356 [https://bugs.python.org/issue?@action=redirect&bpo=32356]: asyncio.transport.resume_reading() and pause_reading() are now idempotent. New transport.is_reading() method is added.
- bpo-32355 [https://bugs.python.org/issue?@action=redirect&bpo=32355]: Optimize asyncio.gather(); now up to 15% faster.
- bpo-32351 [https://bugs.python.org/issue?@action=redirect&bpo=32351]: Use fastpath in asyncio.sleep if delay<0 (2x boost)

- bpo-32348 [https://bugs.python.org/issue?@action=redirect&bpo=32348]: Optimize asyncio.Future schedule/add/remove callback. The optimization shows 3-6% performance improvements of async/await code.
- bpo-32331 [https://bugs.python.org/issue?@action=redirect&bpo=32331]: Fix socket.settimeout() and socket.setblocking() to keep socket.type as is. Fix socket.socket() constructor to reset any bit flags applied to socket's type. This change only affects OSes that have SOCK_NONBLOCK and/or SOCK_CLOEXEC.
- bpo-32248 [https://bugs.python.org/issue?@action=redirect&bpo=32248]: Add **`importlib.abc.ResourceReader`** as an ABC for loaders to provide a unified API for reading resources contained within packages. Also add **`importlib.resources`** as the port of `importlib_resources`.
- bpo-32311 [https://bugs.python.org/issue?@action=redirect&bpo=32311]: Implement asyncio.create_task(coro) shortcut
- bpo-32327 [https://bugs.python.org/issue?@action=redirect&bpo=32327]: Convert asyncio functions that were documented as coroutines to coroutines. Affected functions: loop.sock_sendall, loop.sock_recv, loop.sock_accept, loop.getaddrinfo, loop.getnameinfo.
- bpo-32323 [https://bugs.python.org/issue?@action=redirect&bpo=32323]: **`urllib.parse.urlsplit()`** does not convert zone-id (scope) to lower case for scoped IPv6 addresses in hostnames now.
- bpo-32302 [https://bugs.python.org/issue?@action=redirect&bpo=32302]: Fix bdist_wininst of distutils for CRT v142: it binary compatible with CRT v140.
- bpo-29711 [https://bugs.python.org/issue?@action=redirect&bpo=29711]: Fix `stop_serving` in asyncio proactor loop kill all listening servers
- bpo-32308 [https://bugs.python.org/issue?@action=redirect&bpo=32308]: **`re.sub()`** now replaces empty matches adjacent to a previous non-empty match.

- bpo-29970 [https://bugs.python.org/issue?@action=redirect&bpo=29970]: Abort asyncio SSLProtocol connection if handshake not complete within 10 seconds.
- bpo-32314 [https://bugs.python.org/issue?@action=redirect&bpo=32314]: Implement asyncio.run().
- bpo-17852 [https://bugs.python.org/issue?@action=redirect&bpo=17852]: Revert incorrect fix based on misunderstanding of _Py_PyAtExit() semantics.
- bpo-32296 [https://bugs.python.org/issue?@action=redirect&bpo=32296]: Implement asyncio._get_running_loop() and get_event_loop() in C. This makes them 4x faster.
- bpo-32250 [https://bugs.python.org/issue?@action=redirect&bpo=32250]: Implement `asyncio.current_task()` and `asyncio.all_tasks()`. Add helpers intended to be used by alternative task implementations: `asyncio._register_task`, `asyncio._enter_task`, `asyncio._leave_task` and `asyncio._unregister_task`. Deprecate `asyncio.Task.current_task()` and `asyncio.Task.all_tasks()`.
- bpo-32255 [https://bugs.python.org/issue?@action=redirect&bpo=32255]: A single empty field is now always quoted when written into a CSV file. This allows to distinguish an empty row from a row consisting of a single empty field. Patch by Licht Takeuchi.
- bpo-32277 [https://bugs.python.org/issue?@action=redirect&bpo=32277]: Raise `NotImplementedError` instead of `SystemError` on platforms where `chmod(..., follow_symlinks=False)` is not supported. Patch by Anthony Sottile.
- bpo-30050 [https://bugs.python.org/issue?@action=redirect&bpo=30050]: New argument warn_on_full_buffer to signal.set_wakeup_fd lets you control whether Python prints a warning on stderr when the wakeup fd buffer overflows.

- bpo-29137 [https://bugs.python.org/issue?@action=redirect&bpo=29137]: The `fpectl` library has been removed. It was never enabled by default, never worked correctly on x86-64, and it changed the Python ABI in ways that caused unexpected breakage of C extensions.
- bpo-32273 [https://bugs.python.org/issue?@action=redirect&bpo=32273]: Move asyncio.test_utils to test.test_asyncio.
- bpo-32272 [https://bugs.python.org/issue?@action=redirect&bpo=32272]: Remove asyncio.async() function.
- bpo-32269 [https://bugs.python.org/issue?@action=redirect&bpo=32269]: Add asyncio.get_running_loop() function.
- bpo-32265 [https://bugs.python.org/issue?@action=redirect&bpo=32265]: All class and static methods of builtin types now are correctly classified by inspect.classify_class_attrs() and grouped in pydoc output. Added types.ClassMethodDescriptorType for unbound class methods of builtin types.
- bpo-32253 [https://bugs.python.org/issue?@action=redirect&bpo=32253]: Deprecate `yield from lock`, `await lock`, `with (yield from lock)` and `with await lock` for asyncio synchronization primitives.
- bpo-22589 [https://bugs.python.org/issue?@action=redirect&bpo=22589]: Changed MIME type of .bmp from ‘image/x-ms-bmp’ to ‘image/bmp’
- bpo-32193 [https://bugs.python.org/issue?@action=redirect&bpo=32193]: Convert asyncio to use *async/await* syntax. Old styled `yield from` is still supported too.
- bpo-32206 [https://bugs.python.org/issue?@action=redirect&bpo=32206]: Add support to run modules with pdb
- bpo-32227 [https://bugs.python.org/issue?@action=redirect&bpo=32227]: `functools.singledispatch` now

supports registering implementations using type annotations.

- bpo-15873 [https://bugs.python.org/issue?@action=redirect&bpo=15873]: Added new alternate constructors **`datetime.datetime.fromisoformat()`**, **`datetime.time.fromisoformat()`** and **`datetime.date.fromisoformat()`** as the inverse operation of each classes's respective `isoformat` methods.
- bpo-32199 [https://bugs.python.org/issue?@action=redirect&bpo=32199]: The getnode() ip getter now uses 'ip link' instead of 'ip link list'.
- bpo-32143 [https://bugs.python.org/issue?@action=redirect&bpo=32143]: os.statvfs() includes the f_fsid field from statvfs(2)
- bpo-26439 [https://bugs.python.org/issue?@action=redirect&bpo=26439]: Fix ctypes.util.find_library() for AIX by implementing ctypes._aix.find_library() Patch by: Michael Felt
- bpo-31993 [https://bugs.python.org/issue?@action=redirect&bpo=31993]: The pickler now uses less memory when serializing large bytes and str objects into a file. Pickles created with protocol 4 will require less memory for unpickling large bytes and str objects.
- bpo-27456 [https://bugs.python.org/issue?@action=redirect&bpo=27456]: Ensure TCP_NODELAY is set on Linux. Tests by Victor Stinner.
- bpo-31778 [https://bugs.python.org/issue?@action=redirect&bpo=31778]: ast.literal_eval() is now more strict. Addition and subtraction of arbitrary numbers no longer allowed.
- bpo-31802 [https://bugs.python.org/issue?@action=redirect&bpo=31802]: Importing native path module (`posixpath`, `ntpath`) now works even if the `os` module still is not imported.
- bpo-30241 [https://bugs.python.org/issue?@action=redirect&bpo=30241]: Add

contextlib.AbstractAsyncContextManager. Patch by Jelle Zijlstra.
- bpo-31699 [https://bugs.python.org/issue?@action=redirect&bpo=31699]: Fix deadlocks in `concurrent.futures.ProcessPoolExecutor` when task arguments or results cause pickling or unpickling errors. This should make sure that calls to the **`ProcessPoolExecutor`** API always eventually return.
- bpo-15216 [https://bugs.python.org/issue?@action=redirect&bpo=15216]: `TextIOWrapper.reconfigure()` supports changing *encoding*, *errors*, and *newline*.

Documentation

- bpo-32418 [https://bugs.python.org/issue?@action=redirect&bpo=32418]: Add get_loop() method to Server and AbstractServer classes.

Tests

- bpo-32252 [https://bugs.python.org/issue?@action=redirect&bpo=32252]: Fix faulthandler_suppress_crash_report() used to prevent core dump files when testing crashes. getrlimit() returns zero on success.
- bpo-32002 [https://bugs.python.org/issue?@action=redirect&bpo=32002]: Adjust C locale coercion testing for the empty locale and POSIX locale cases to more readily adjust to platform dependent behaviour.

Windows

- bpo-19764 [https://bugs.python.org/issue?@action=redirect&bpo=19764]: Implement support for `subprocess.Popen(close_fds=True)` on Windows. Patch by Segev Finer.

Tools/Demos

- bpo-24960 [https://bugs.python.org/issue?@action=redirect&bpo=24960]: 2to3 and lib2to3 can now read pickled grammar files using pkgutil.get_data() rather than probing the filesystem. This lets 2to3 and lib2to3 work when run from a zipfile.

C API

- bpo-32030 [https://bugs.python.org/issue?@action=redirect&bpo=32030]: Py_Initialize() doesn’t reset the memory allocators to default if the `PYTHONMALLOC` environment variable is not set.
- bpo-29084 [https://bugs.python.org/issue?@action=redirect&bpo=29084]: Undocumented C API for OrderedDict has been excluded from the limited C API. It was added by mistake and actually never worked in the limited C API.
- bpo-32264 [https://bugs.python.org/issue?@action=redirect&bpo=32264]: Moved the pygetopt.h header into internal/, since it has no public APIs.
- bpo-32241 [https://bugs.python.org/issue?@action=redirect&bpo=32241]: **`Py_SetProgramName()`** and **`Py_SetPythonHome()`** now take the `const wchar *` arguments instead of `wchar *`.

Python 3.7.0 alpha 3

Release date: 2017-12-05

Core and Builtins

- bpo-32176 [https://bugs.python.org/issue?@action=redirect&bpo=32176]: co_flags.CO_NOFREE is now always set correctly by the code object constructor based on freevars and cellvars, rather than needing to be set correctly by the caller. This ensures it will be cleared automatically when additional cell references are injected into a modified code object and function.

- bpo-10544 [https://bugs.python.org/issue?@action=redirect&bpo=10544]: Yield expressions are now deprecated in comprehensions and generator expressions. They are still permitted in the definition of the outermost iterable, as that is evaluated directly in the enclosing scope.
- bpo-32137 [https://bugs.python.org/issue?@action=redirect&bpo=32137]: The repr of deeply nested dict now raises a RecursionError instead of crashing due to a stack overflow.
- bpo-32096 [https://bugs.python.org/issue?@action=redirect&bpo=32096]: Revert memory allocator changes in the C API: move structures back from _PyRuntime to Objects/obmalloc.c. The memory allocators are once again initialized statically, and so PyMem_RawMalloc() and Py_DecodeLocale() can be called before _PyRuntime_Initialize().
- bpo-32043 [https://bugs.python.org/issue?@action=redirect&bpo=32043]: Add a new “developer mode”: new “-X dev” command line option to enable debug checks at runtime.
- bpo-32023 [https://bugs.python.org/issue?@action=redirect&bpo=32023]: SyntaxError is now correctly raised when a generator expression without parenthesis is used instead of an inheritance list in a class definition. The duplication of the parentheses can be omitted only on calls.
- bpo-32012 [https://bugs.python.org/issue?@action=redirect&bpo=32012]: SyntaxError is now correctly raised when a generator expression without parenthesis is passed as an argument, but followed by a trailing

comma. A generator expression always needs to be directly inside a set of parentheses and cannot have a comma on either side.

- bpo-28180 [https://bugs.python.org/issue?@action=redirect&bpo=28180]: A new internal `_Py_SetLocaleFromEnv(category)` helper function has been added in order to improve the consistency of behaviour across different `libc` implementations (e.g. Android doesn't support setting the locale from the environment by default).
- bpo-31949 [https://bugs.python.org/issue?@action=redirect&bpo=31949]: Fixed several issues in printing tracebacks (PyTraceBack_Print()). Setting sys.tracebacklimit to 0 or less now suppresses printing tracebacks. Setting sys.tracebacklimit to None now causes using the default limit. Setting sys.tracebacklimit to an integer larger than LONG_MAX now means using the limit LONG_MAX rather than the default limit. Fixed integer overflows in the case of more than `2**31` traceback items on Windows. Fixed output errors handling.
- bpo-30696 [https://bugs.python.org/issue?@action=redirect&bpo=30696]: Fix the interactive interpreter looping endlessly when no memory.
- bpo-20047 [https://bugs.python.org/issue?@action=redirect&bpo=20047]: Bytearray methods partition() and rpartition() now accept only bytes-like objects as separator, as documented. In particular they now raise TypeError rather of returning a bogus result when an integer is passed as a separator.
- bpo-21720 [https://bugs.python.org/issue?@action=redirect&bpo=21720]: BytesWarning no longer emitted when the *fromlist* argument of `__import__()` or the `__all__` attribute of the module contain bytes instances.

- bpo-31845 [https://bugs.python.org/issue?@action=redirect&bpo=31845]: Environment variables are once more read correctly at interpreter startup.
- bpo-28936 [https://bugs.python.org/issue?@action=redirect&bpo=28936]: Ensure that lexically first syntax error involving a parameter and `global` or `nonlocal` is detected first at a given scope. Patch by Ivan Levkivskyi.
- bpo-31825 [https://bugs.python.org/issue?@action=redirect&bpo=31825]: Fixed OverflowError in the ‘unicode-escape’ codec and in codecs.escape_decode() when decode an escaped non-ascii byte.
- bpo-31618 [https://bugs.python.org/issue?@action=redirect&bpo=31618]: The per-frame tracing logic added in 3.7a1 has been altered so that `frame->f_lineno` is updated before either `"line"` or `"opcode"` events are emitted. Previously, opcode events were emitted first, and therefore would occasionally see stale line numbers on the frame. The behavior of this feature has changed slightly as a result: when both `f_trace_lines` and `f_trace_opcodes` are enabled, line events now occur first.
- bpo-28603 [https://bugs.python.org/issue?@action=redirect&bpo=28603]: Print the full context/cause chain of exceptions on interpreter exit, even if an exception in the chain is unhashable or compares equal to later ones. Patch by Zane Bitter.
- bpo-31786 [https://bugs.python.org/issue?@action=redirect&bpo=31786]: Fix timeout rounding in the select module to round correctly negative timeouts between -1.0 and 0.0. The functions now block waiting for events as expected. Previously, the call was incorrectly non-blocking. Patch by Pablo Galindo.
- bpo-31781 [https://bugs.python.org/issue?@action=redirect&bpo=31781]: Prevent crashes when calling methods of an uninitialized `zipimport.zipimporter` object. Patch by Oren Milman.

- bpo-30399 [https://bugs.python.org/issue?@action=redirect&bpo=30399]: Standard repr() of BaseException with a single argument no longer contains redundant trailing comma.
- bpo-31626 [https://bugs.python.org/issue?@action=redirect&bpo=31626]: Fixed a bug in debug memory allocator. There was a write to freed memory after shrinking a memory block.
- bpo-30817 [https://bugs.python.org/issue?@action=redirect&bpo=30817]: `PyErr_PrintEx()` clears now the ignored exception that may be raised by `_PySys_SetObjectId()`, for example when no memory.

Library

- bpo-28556 [https://bugs.python.org/issue?@action=redirect&bpo=28556]: Two minor fixes for `typing` module: allow shallow copying instances of generic classes, improve interaction of `__init_subclass__` with generics. Original PRs by Ivan Levkivskyi.
- bpo-32214 [https://bugs.python.org/issue?@action=redirect&bpo=32214]: PEP 557, Data Classes. Provides a decorator which adds boilerplate methods to classes which use type annotations so specify fields.
- bpo-27240 [https://bugs.python.org/issue?@action=redirect&bpo=27240]: The header folding algorithm for the new email policies has been rewritten, which also fixes bpo-30788 [https://bugs.python.org/issue?@action=redirect&bpo=30788], bpo-31831 [https://bugs.python.org/issue?@action=redirect&bpo=31831], and bpo-32182 [https://bugs.python.org/issue?@action=redirect&bpo=32182]. In particular, RFC2231 folding is now done correctly.
- bpo-32186 [https://bugs.python.org/issue?@action=redirect&bpo=32186]: io.FileIO.readall() and io.FileIO.read() now release the GIL when getting the file

size. Fixed hang of all threads with inaccessible NFS server. Patch by Nir Soffer.

- bpo-32101 [https://bugs.python.org/issue?@action=redirect&bpo=32101]: Add **`sys.flags.dev_mode`** flag
- bpo-32154 [https://bugs.python.org/issue?@action=redirect&bpo=32154]: The `asyncio.windows_utils.socketpair()` function has been removed: use directly **`socket.socketpair()`** which is available on all platforms since Python 3.5 (before, it wasn't available on Windows). `asyncio.windows_utils.socketpair()` was just an alias to `socket.socketpair` on Python 3.5 and newer.
- bpo-32089 [https://bugs.python.org/issue?@action=redirect&bpo=32089]: warnings: In development (-X dev) and debug mode (pydebug build), use the "default" action for ResourceWarning, rather than the "always" action, in the default warnings filters.
- bpo-32107 [https://bugs.python.org/issue?@action=redirect&bpo=32107]: `uuid.getnode()` now preferentially returns universally administered MAC addresses if available, over locally administered MAC addresses. This makes a better guarantee for global uniqueness of UUIDs returned from `uuid.uuid1()`. If only locally administered MAC addresses are available, the first such one found is returned.
- bpo-23033 [https://bugs.python.org/issue?@action=redirect&bpo=23033]: Wildcard is now supported in hostname when it is one and only character in the left most segment of hostname in second argument of **`ssl.match_hostname()`**. Patch by Mandeep Singh.
- bpo-12239 [https://bugs.python.org/issue?@action=redirect&bpo=12239]: Make **`msilib.SummaryInformation.GetProperty()`** return `None` when the value of property is `VT_EMPTY`. Initial patch by Mark Mc Mahon.

- bpo-28334 [https://bugs.python.org/issue?@action=redirect&bpo=28334]: Use `os.path.expanduser()` to find the `~/.netrc` file in `netrc.netrc`. If it does not exist, `FileNotFoundError` is raised. Patch by Dimitri Merejkowsky.
- bpo-32121 [https://bugs.python.org/issue?@action=redirect&bpo=32121]: Made `tracemalloc.Traceback` behave more like the traceback module, sorting the frames from oldest to most recent. `Traceback.format()` now accepts negative *limit*, truncating the result to the `abs(limit)` oldest frames. To get the old behaviour, one can use the new *most_recent_first* argument to `Traceback.format()`. (Patch by Jesse Bakker.)
- bpo-31325 [https://bugs.python.org/issue?@action=redirect&bpo=31325]: Fix wrong usage of `collections.namedtuple()` in the `RobotFileParser.parse()` method. Initial patch by Robin Wellner.
- bpo-12382 [https://bugs.python.org/issue?@action=redirect&bpo=12382]: **`msilib.OpenDatabase()`** now raises a better exception message when it couldn’t open or create an MSI file. Initial patch by William Tisäter.
- bpo-19610 [https://bugs.python.org/issue?@action=redirect&bpo=19610]: `setup()` now warns about invalid types for some fields. The `distutils.dist.Distribution` class now warns when `classifiers`, `keywords` and `platforms` fields are not specified as a list or a string.
- bpo-32071 [https://bugs.python.org/issue?@action=redirect&bpo=32071]: Added the `-k` command-line option to `python -m unittest` to run only tests that match the given pattern(s).
- bpo-10049 [https://bugs.python.org/issue?@action=redirect&bpo=10049]: Added *nullcontext* no-op context manager to contextlib. This provides a simpler and faster alternative to ExitStack() when handling optional context managers.
- bpo-28684 [https://bugs.python.org/issue?@action=redirect&bpo=28684]: The new

test.support.skip_unless_bind_unix_socket() decorator is used here to skip asyncio tests that fail because the platform lacks a functional bind() function for unix domain sockets (as it is the case for non root users on the recent Android versions that run now SELinux in enforcing mode).

- bpo-32110 [https://bugs.python.org/issue?@action=redirect&bpo=32110]: `codecs.StreamReader.read(n)` now returns not more than *n* characters/bytes for non-negative *n*. This makes it compatible with `read()` methods of other file-like objects.
- bpo-27535 [https://bugs.python.org/issue?@action=redirect&bpo=27535]: The warnings module doesn’t leak memory anymore in the hidden warnings registry for the “ignore” action of warnings filters. warn_explicit() function doesn’t add the warning key to the registry anymore for the “ignore” action.
- bpo-32088 [https://bugs.python.org/issue?@action=redirect&bpo=32088]: warnings: When Python is build is debug mode (`Py_DEBUG`), **`DeprecationWarning`**, **`PendingDeprecationWarning`** and **`ImportWarning`** warnings are now displayed by default.
- bpo-1647489 [https://bugs.python.org/issue?@action=redirect&bpo=1647489]: Fixed searching regular expression patterns that could match an empty string. Non-empty string can now be correctly found after matching an empty string.
- bpo-25054 [https://bugs.python.org/issue?@action=redirect&bpo=25054]: Added support of splitting on a pattern that could match an empty string.
- bpo-32072 [https://bugs.python.org/issue?@action=redirect&bpo=32072]: Fixed issues with binary plists: Fixed saving bytearrays. Identical objects will be saved only once. Equal references will be load as identical objects. Added support for saving and loading recursive data structures.

- bpo-32069 [https://bugs.python.org/issue?@action=redirect&bpo=32069]: Drop legacy SSL transport from asyncio, ssl.MemoryBIO is always used anyway.
- bpo-32066 [https://bugs.python.org/issue?@action=redirect&bpo=32066]: asyncio: Support pathlib.Path in create_unix_connection; sock arg should be optional
- bpo-32046 [https://bugs.python.org/issue?@action=redirect&bpo=32046]: Updates 2to3 to convert from operator.isCallable(obj) to callable(obj). Patch by Donghee Na.
- bpo-32018 [https://bugs.python.org/issue?@action=redirect&bpo=32018]: inspect.signature should follow **PEP 8** [https://peps.python.org/pep-0008/], if the parameter has an annotation and a default value. Patch by Donghee Na.
- bpo-32025 [https://bugs.python.org/issue?@action=redirect&bpo=32025]: Add time.thread_time() and time.thread_time_ns()
- bpo-32037 [https://bugs.python.org/issue?@action=redirect&bpo=32037]: Integers that fit in a signed 32-bit integer will be now pickled with protocol 0 using the INT opcode. This will decrease the size of a pickle, speed up pickling and unpickling, and make these integers be unpickled as int instances in Python 2.
- bpo-32034 [https://bugs.python.org/issue?@action=redirect&bpo=32034]: Make asyncio.IncompleteReadError and LimitOverrunError pickleable.
- bpo-32015 [https://bugs.python.org/issue?@action=redirect&bpo=32015]: Fixed the looping of asyncio in the case of reconnection the socket during waiting async read/write from/to the socket.
- bpo-32011 [https://bugs.python.org/issue?@action=redirect&bpo=32011]: Restored support of loading marshal files with the TYPE_INT64 code. These files can be produced in Python 2.7.

- bpo-28369 [https://bugs.python.org/issue?@action=redirect&bpo=28369]: Enhance add_reader/writer check that socket is not used by some transport. Before, only cases when add_reader/writer were called with an int FD were supported. Now the check is implemented correctly for all file-like objects.
- bpo-31976 [https://bugs.python.org/issue?@action=redirect&bpo=31976]: Fix race condition when flushing a file is slow, which can cause a segfault if closing the file from another thread.
- bpo-31985 [https://bugs.python.org/issue?@action=redirect&bpo=31985]: Formally deprecated aifc.openfp, sunau.openfp, and wave.openfp. Since change 7bc817d5ba917528e8bd07ec461c635291e7b06a in 1993, openfp in each of the three modules had been pointing to that module’s open function as a matter of backwards compatibility, though it had been both untested and undocumented.
- bpo-21862 [https://bugs.python.org/issue?@action=redirect&bpo=21862]: cProfile command line now accepts `-m module_name` as an alternative to script path. Patch by Sanyam Khurana.
- bpo-31970 [https://bugs.python.org/issue?@action=redirect&bpo=31970]: Reduce performance overhead of asyncio debug mode.
- bpo-31843 [https://bugs.python.org/issue?@action=redirect&bpo=31843]: *database* argument of sqlite3.connect() now accepts a path-like object, instead of just a string.
- bpo-31945 [https://bugs.python.org/issue?@action=redirect&bpo=31945]: Add Configurable *blocksize* to `HTTPConnection` and `HTTPSConnection` for improved upload throughput. Patch by Nir Soffer.
- bpo-31943 [https://bugs.python.org/issue?@action=redirect&bpo=31943]: Add a `cancelled()` method to

`asyncio.Handle`. Patch by Marat Sharafutdinov.

- bpo-9678 [https://bugs.python.org/issue?@action=redirect&bpo=9678]: Fixed determining the MAC address in the uuid module: Using ifconfig on NetBSD and OpenBSD. Using arp on Linux, FreeBSD, NetBSD and OpenBSD. Based on patch by Takayuki Shimizukawa.
- bpo-30057 [https://bugs.python.org/issue?@action=redirect&bpo=30057]: Fix potential missed signal in signal.signal().
- bpo-31933 [https://bugs.python.org/issue?@action=redirect&bpo=31933]: Fix Blake2 params leaf_size and node_offset on big endian platforms. Patch by Jack O’Connor.
- bpo-21423 [https://bugs.python.org/issue?@action=redirect&bpo=21423]: Add an initializer argument to {Process,Thread}PoolExecutor
- bpo-31927 [https://bugs.python.org/issue?@action=redirect&bpo=31927]: Fixed compilation of the socket module on NetBSD 8. Fixed assertion failure or reading arbitrary data when parse a AF_BLUETOOTH address on NetBSD and DragonFly BSD.
- bpo-27666 [https://bugs.python.org/issue?@action=redirect&bpo=27666]: Fixed stack corruption in curses.box() and curses.ungetmouse() when the size of types chtype or mmask_t is less than the size of C long. curses.box() now accepts characters as arguments. Based on patch by Steve Fink.
- bpo-31917 [https://bugs.python.org/issue?@action=redirect&bpo=31917]: Add 3 new clock identifiers: `time.CLOCK_BOOTTIME`, `time.CLOCK_PROF` and `time.CLOCK_UPTIME`.
- bpo-31897 [https://bugs.python.org/issue?@action=redirect&bpo=31897]: plistlib now catches more errors when read binary plists and raises InvalidFileException instead of unexpected exceptions.
- bpo-25720 [https://bugs.python.org/issue?@action=redirect&bpo=25720]: Fix the method for checking

pad state of curses WINDOW. Patch by Masayuki Yamamoto.

- bpo-31893 [https://bugs.python.org/issue?@action=redirect&bpo=31893]: Fixed the layout of the kqueue_event structure on OpenBSD and NetBSD. Fixed the comparison of the kqueue_event objects.
- bpo-31891 [https://bugs.python.org/issue?@action=redirect&bpo=31891]: Fixed building the curses module on NetBSD.
- bpo-31884 [https://bugs.python.org/issue?@action=redirect&bpo=31884]: added required constants to subprocess module for setting priority on windows
- bpo-28281 [https://bugs.python.org/issue?@action=redirect&bpo=28281]: Remove year (1-9999) limits on the Calendar.weekday() function. Patch by Mark Gollahon.
- bpo-31702 [https://bugs.python.org/issue?@action=redirect&bpo=31702]: crypt.mksalt() now allows to specify the number of rounds for SHA-256 and SHA-512 hashing.
- bpo-30639 [https://bugs.python.org/issue?@action=redirect&bpo=30639]: **`inspect.getfile()`** no longer computes the repr of unknown objects to display in an error message, to protect against badly behaved custom reprs.
- bpo-30768 [https://bugs.python.org/issue?@action=redirect&bpo=30768]: Fix the pthread+semaphore implementation of PyThread_acquire_lock_timed() when called with timeout > 0 and intr_flag=0: recompute the timeout if sem_timedwait() is interrupted by a signal (EINTR). See also the **PEP 475** [https://peps.python.org/pep-0475/].
- bpo-31854 [https://bugs.python.org/issue?@action=redirect&bpo=31854]: Add `mmap.ACCESS_DEFAULT` constant.

- bpo-31834 [https://bugs.python.org/issue?@action=redirect&bpo=31834]: Use optimized code for BLAKE2 only with SSSE3+. The pure SSE2 implementation is slower than the pure C reference implementation.
- bpo-28292 [https://bugs.python.org/issue?@action=redirect&bpo=28292]: Calendar.itermonthdates() will now consistently raise an exception when a date falls outside of the 0001-01-01 through 9999-12-31 range. To support applications that cannot tolerate such exceptions, the new methods itermonthdays3() and itermonthdays4() are added. The new methods return tuples and are not restricted by the range supported by datetime.date.
- bpo-28564 [https://bugs.python.org/issue?@action=redirect&bpo=28564]: The shutil.rmtree() function has been sped up to 20–40%. This was done using the os.scandir() function.
- bpo-28416 [https://bugs.python.org/issue?@action=redirect&bpo=28416]: Instances of pickle.Pickler subclass with the persistent_id() method and pickle.Unpickler subclass with the persistent_load() method no longer create reference cycles.
- bpo-31653 [https://bugs.python.org/issue?@action=redirect&bpo=31653]: Don’t release the GIL if we can acquire a multiprocessing semaphore immediately.
- bpo-28326 [https://bugs.python.org/issue?@action=redirect&bpo=28326]: Fix multiprocessing.Process when stdout and/or stderr is closed or None.
- bpo-20825 [https://bugs.python.org/issue?@action=redirect&bpo=20825]: Add `subnet_of` and `superset_of` containment tests to **`ipaddress.IPv6Network`** and **`ipaddress.IPv4Network`**. Patch by Michel Albert and Cheryl Sabella.
- bpo-31827 [https://bugs.python.org/issue?@action=redirect&bpo=31827]: Remove the os.stat_float_times() function. It was introduced in

Python 2.3 for backward compatibility with Python 2.2, and was deprecated since Python 3.1.

- bpo-31756 [https://bugs.python.org/issue?@action=redirect&bpo=31756]: Add a `subprocess.Popen(text=False)` keyword argument to `subprocess` functions to be more explicit about when the library should attempt to decode outputs into text. Patch by Andrew Clegg.
- bpo-31819 [https://bugs.python.org/issue?@action=redirect&bpo=31819]: Add AbstractEventLoop.sock_recv_into().
- bpo-31457 [https://bugs.python.org/issue?@action=redirect&bpo=31457]: If nested log adapters are used, the inner `process()` methods are no longer omitted.
- bpo-31457 [https://bugs.python.org/issue?@action=redirect&bpo=31457]: The `manager` property on LoggerAdapter objects is now properly settable.
- bpo-31806 [https://bugs.python.org/issue?@action=redirect&bpo=31806]: Fix timeout rounding in time.sleep(), threading.Lock.acquire() and socket.socket.settimeout() to round correctly negative timeouts between -1.0 and 0.0. The functions now block waiting for events as expected. Previously, the call was incorrectly non-blocking. Patch by Pablo Galindo.
- bpo-31803 [https://bugs.python.org/issue?@action=redirect&bpo=31803]: time.clock() and time.get_clock_info('clock') now emit a DeprecationWarning warning.
- bpo-31800 [https://bugs.python.org/issue?@action=redirect&bpo=31800]: Extended support for parsing UTC offsets. strptime '%z' can now parse the output generated by datetime.isoformat, including seconds and microseconds.
- bpo-28603 [https://bugs.python.org/issue?@action=redirect&bpo=28603]: traceback: Fix a TypeError that occurred during printing of exception tracebacks when

either the current exception or an exception in its context/cause chain is unhashable. Patch by Zane Bitter.

- bpo-30541 [https://bugs.python.org/issue?@action=redirect&bpo=30541]: Add new function to seal a mock and prevent the automatically creation of child mocks. Patch by Mario Corchero.
- bpo-31784 [https://bugs.python.org/issue?@action=redirect&bpo=31784]: Implement the **PEP 564** [https://peps.python.org/pep-0564/], add new 6 new functions with nanosecond resolution to the `time` module: `clock_gettime_ns()`, `clock_settime_ns()`, `monotonic_ns()`, `perf_counter_ns()`, `process_time_ns()`, `time_ns()`.
- bpo-30143 [https://bugs.python.org/issue?@action=redirect&bpo=30143]: 2to3 now generates a code that uses abstract collection classes from collections.abc rather than collections.
- bpo-31770 [https://bugs.python.org/issue?@action=redirect&bpo=31770]: Prevent a crash when calling the `__init__()` method of a `sqlite3.Cursor` object more than once. Patch by Oren Milman.
- bpo-31764 [https://bugs.python.org/issue?@action=redirect&bpo=31764]: Prevent a crash in `sqlite3.Cursor.close()` in case the `Cursor` object is uninitialized. Patch by Oren Milman.
- bpo-31752 [https://bugs.python.org/issue?@action=redirect&bpo=31752]: Fix possible crash in timedelta constructor called with custom integers.
- bpo-31620 [https://bugs.python.org/issue?@action=redirect&bpo=31620]: an empty asyncio.Queue now doesn’t leak memory when queue.get pollers timeout
- bpo-31690 [https://bugs.python.org/issue?@action=redirect&bpo=31690]: Allow the flags re.ASCII, re.LOCALE, and re.UNICODE to be used as group flags for regular expressions.
- bpo-30349 [https://bugs.python.org/issue?@action=redirect&bpo=30349]: FutureWarning is now emitted if

a regular expression contains character set constructs that will change semantically in the future (nested sets and set operations).

- bpo-31664 [https://bugs.python.org/issue?@action=redirect&bpo=31664]: Added support for the Blowfish hashing in the crypt module.
- bpo-31632 [https://bugs.python.org/issue?@action=redirect&bpo=31632]: Fix method set_protocol() of class _SSLProtocolTransport in asyncio module. This method was previously modifying a wrong reference to the protocol.
- bpo-15037 [https://bugs.python.org/issue?@action=redirect&bpo=15037]: Added a workaround for getkey() in curses for ncurses 5.7 and earlier.
- bpo-31307 [https://bugs.python.org/issue?@action=redirect&bpo=31307]: Allow use of bytes objects for arguments to **`configparser.ConfigParser.read()`**. Patch by Vincent Michel.
- bpo-31334 [https://bugs.python.org/issue?@action=redirect&bpo=31334]: Fix `poll.poll([timeout])` in the `select` module for arbitrary negative timeouts on all OSes where it can only be a non-negative integer or -1. Patch by Riccardo Coccioli.
- bpo-31310 [https://bugs.python.org/issue?@action=redirect&bpo=31310]: multiprocessing’s semaphore tracker should be launched again if crashed.
- bpo-31308 [https://bugs.python.org/issue?@action=redirect&bpo=31308]: Make multiprocessing’s forkserver process immune to Ctrl-C and other user interruptions. If it crashes, restart it when necessary.
- bpo-31245 [https://bugs.python.org/issue?@action=redirect&bpo=31245]: Added support for AF_UNIX socket in asyncio `create_datagram_endpoint`.
- bpo-30553 [https://bugs.python.org/issue?@action=redirect&bpo=30553]: Add HTTP/2 status code 421

(Misdirected Request) to `http.HTTPStatus`. Patch by Vitor Pereira.

Documentation

- bpo-32105 [https://bugs.python.org/issue?@action=redirect&bpo=32105]: Added asyncio.BaseEventLoop.connect_accepted_socket versionadded marker.

Tests

- bpo-31380 [https://bugs.python.org/issue?@action=redirect&bpo=31380]: Skip test_httpservers test_undecodable_file on macOS: fails on APFS.
- bpo-31705 [https://bugs.python.org/issue?@action=redirect&bpo=31705]: Skip test_socket.test_sha256() on Linux kernel older than 4.5. The test fails with ENOKEY on kernel 3.10 (on ppc64le). A fix was merged into the kernel 4.5.
- bpo-32138 [https://bugs.python.org/issue?@action=redirect&bpo=32138]: Skip on Android test_faulthandler tests that raise SIGSEGV and remove the test.support.requires_android_level decorator.
- bpo-32136 [https://bugs.python.org/issue?@action=redirect&bpo=32136]: The runtime embedding tests have been split out from `Lib/test/test_capi.py` into a new `Lib/test/test_embed.py` file.
- bpo-28668 [https://bugs.python.org/issue?@action=redirect&bpo=28668]: test.support.requires_multiprocessing_queue is removed. Skip tests with test.support.import_module(‘multiprocessing.synchronize’) instead when the semaphore implementation is broken or missing.

- bpo-32126 [https://bugs.python.org/issue?@action=redirect&bpo=32126]: Skip test_get_event_loop_new_process in test.test_asyncio.test_events when sem_open() is not functional.
- bpo-31174 [https://bugs.python.org/issue?@action=redirect&bpo=31174]: Fix test_tools.test_unparse: DirectoryTestCase now stores the names sample to always test the same files. It prevents false alarms when hunting reference leaks.

Build

- bpo-28538 [https://bugs.python.org/issue?@action=redirect&bpo=28538]: Revert the previous changes, the if_nameindex structure is defined by Unified Headers.
- bpo-28762 [https://bugs.python.org/issue?@action=redirect&bpo=28762]: Revert the last commit, the F_LOCK macro is defined by Android Unified Headers.
- bpo-29040 [https://bugs.python.org/issue?@action=redirect&bpo=29040]: Support building Android with Unified Headers. The first NDK release to support Unified Headers is android-ndk-r14.
- bpo-32059 [https://bugs.python.org/issue?@action=redirect&bpo=32059]: `detect_modules()` in `setup.py` now also searches the sysroot paths when cross-compiling.
- bpo-31957 [https://bugs.python.org/issue?@action=redirect&bpo=31957]: Fixes Windows SDK version detection when building for Windows.
- bpo-31609 [https://bugs.python.org/issue?@action=redirect&bpo=31609]: Fixes quotes in PCbuild/clean.bat
- bpo-31934 [https://bugs.python.org/issue?@action=redirect&bpo=31934]: Abort the build when building out of a not clean source tree.

- bpo-31926 [https://bugs.python.org/issue?@action=redirect&bpo=31926]: Fixed Argument Clinic sometimes causing compilation errors when there was more than one function and/or method in a .c file with the same name.
- bpo-28791 [https://bugs.python.org/issue?@action=redirect&bpo=28791]: Update Windows builds to use SQLite 3.21.0.
- bpo-28791 [https://bugs.python.org/issue?@action=redirect&bpo=28791]: Update OS X installer to use SQLite 3.21.0.
- bpo-28643 [https://bugs.python.org/issue?@action=redirect&bpo=28643]: Record profile-opt build progress with stamp files.
- bpo-31866 [https://bugs.python.org/issue?@action=redirect&bpo=31866]: Finish removing support for AtheOS.

Windows

- bpo-1102 [https://bugs.python.org/issue?@action=redirect&bpo=1102]: Return `None` when `View.Fetch()` returns `ERROR_NO_MORE_ITEMS` instead of raising `MSIError`. Initial patch by Anthony Tuininga.
- bpo-31944 [https://bugs.python.org/issue?@action=redirect&bpo=31944]: Fixes Modify button in Apps and Features dialog.
- bpo-20486 [https://bugs.python.org/issue?@action=redirect&bpo=20486]: Implement the `Database.Close()` method to help closing MSI database objects.
- bpo-31857 [https://bugs.python.org/issue?@action=redirect&bpo=31857]: Make the behavior of USE_STACKCHECK deterministic in a multi-threaded environment.

macOS

- bpo-31392 [https://bugs.python.org/issue?@action=redirect&bpo=31392]: Update macOS installer to use OpenSSL 1.0.2m

IDLE

- bpo-32207 [https://bugs.python.org/issue?@action=redirect&bpo=32207]: Improve tk event exception tracebacks in IDLE. When tk event handling is driven by IDLE’s run loop, a confusing and distracting queue.EMPTY traceback context is no longer added to tk event exception tracebacks. The traceback is now the same as when event handling is driven by user code. Patch based on a suggestion by Serhiy Storchaka.
- bpo-32164 [https://bugs.python.org/issue?@action=redirect&bpo=32164]: Delete unused file idlelib/tabbedpages.py. Use of TabbedPageSet in configdialog was replaced by ttk.Notebook.
- bpo-32100 [https://bugs.python.org/issue?@action=redirect&bpo=32100]: IDLE: Fix old and new bugs in pathbrowser; improve tests. Patch mostly by Cheryl Sabella.
- bpo-31858 [https://bugs.python.org/issue?@action=redirect&bpo=31858]: IDLE – Restrict shell prompt manipulation to the shell. Editor and output windows only see an empty last prompt line. This simplifies the code and fixes a minor bug when newline is inserted. Sys.ps1, if present, is read on Shell start-up, but is not set or changed.
- bpo-31860 [https://bugs.python.org/issue?@action=redirect&bpo=31860]: The font sample in the IDLE configuration dialog is now editable. Changes persist while IDLE remains open

- bpo-31836 [https://bugs.python.org/issue?@action=redirect&bpo=31836]: Test_code_module now passes if run after test_idle, which sets ps1. The code module uses sys.ps1 if present or sets it to '>>> ' if not. Test_code_module now properly tests both behaviors. Ditto for ps2.
- bpo-28603 [https://bugs.python.org/issue?@action=redirect&bpo=28603]: Fix a TypeError that caused a shell restart when printing a traceback that includes an exception that is unhashable. Patch by Zane Bitter.
- bpo-13802 [https://bugs.python.org/issue?@action=redirect&bpo=13802]: Use non-Latin characters in the IDLE's Font settings sample. Even if one selects a font that defines a limited subset of the unicode Basic Multilingual Plane, tcl/tk will use other fonts that define a character. The expanded example give users of non-Latin characters a better idea of what they might see in IDLE's shell and editors. To make room for the expanded sample, frames on the Font tab are re-arranged. The Font/Tabs help explains a bit about the additions.

Tools/Demos

- bpo-32159 [https://bugs.python.org/issue?@action=redirect&bpo=32159]: Remove CVS and Subversion tools: remove svneol.py and treesync.py scripts. CPython migrated from CVS to Subversion, to Mercurial, and then to Git. CVS and Subversion are no longer used to develop CPython.
- bpo-30722 [https://bugs.python.org/issue?@action=redirect&bpo=30722]: Make redemo work with Python 3.6 and newer versions. Also, remove the `LOCALE` option since it doesn't work with string patterns in Python 3. Patch by Christoph Sarnowski.

C API

- bpo-20891 [https://bugs.python.org/issue?@action=redirect&bpo=20891]: Fix PyGILState_Ensure(). When PyGILState_Ensure() is called in a non-Python thread before PyEval_InitThreads(), only call PyEval_InitThreads() after calling PyThreadState_New() to fix a crash.
- bpo-32125 [https://bugs.python.org/issue?@action=redirect&bpo=32125]: The `Py_UseClassExceptionsFlag` flag has been removed. It was deprecated and wasn't used anymore since Python 2.0.
- bpo-25612 [https://bugs.python.org/issue?@action=redirect&bpo=25612]: Move the current exception state from the frame object to the co-routine. This simplifies the interpreter and fixes a couple of obscure bugs caused by having swap exception state when entering or exiting a generator.
- bpo-23699 [https://bugs.python.org/issue?@action=redirect&bpo=23699]: Add Py_RETURN_RICHCOMPARE macro to reduce boilerplate code in rich comparison functions.
- bpo-30697 [https://bugs.python.org/issue?@action=redirect&bpo=30697]: The `PyExc_RecursionErrorInst` singleton is removed and `PyErr_NormalizeException()` does not use it anymore. This singleton is persistent and its members being never cleared may cause a segfault during finalization of the interpreter. See also bpo-22898 [https://bugs.python.org/issue?@action=redirect&bpo=22898].

Python 3.7.0 alpha 2

Release date: 2017-10-16

Core and Builtins

- bpo-31558 [https://bugs.python.org/issue?@action=redirect&bpo=31558]: `gc.freeze()` is a new API that allows for moving all objects currently tracked by the garbage collector to a permanent generation, effectively removing them from future collection events. This can be used to protect those objects from having their PyGC_Head mutated. In effect, this enables great copy-on-write stability at fork().
- bpo-31642 [https://bugs.python.org/issue?@action=redirect&bpo=31642]: Restored blocking “from package import module” by setting sys.modules[“package.module”] to None.
- bpo-31708 [https://bugs.python.org/issue?@action=redirect&bpo=31708]: Allow use of asynchronous generator expressions in synchronous functions.
- bpo-31709 [https://bugs.python.org/issue?@action=redirect&bpo=31709]: Drop support of asynchronous __aiter__.
- bpo-30404 [https://bugs.python.org/issue?@action=redirect&bpo=30404]: The -u option now makes the stdout and stderr streams unbuffered rather than line-buffered.
- bpo-31619 [https://bugs.python.org/issue?@action=redirect&bpo=31619]: Fixed a ValueError when convert a string with large number of underscores to integer with binary base.
- bpo-31602 [https://bugs.python.org/issue?@action=redirect&bpo=31602]: Fix an assertion failure in `zipimporter.get_source()` in case of a bad `zlib.decompress()`. Patch by Oren Milman.
- bpo-31592 [https://bugs.python.org/issue?@action=redirect&bpo=31592]: Fixed an assertion failure in Python parser in case of a bad `unicodedata.normalize()`. Patch by Oren Milman.
- bpo-31588 [https://bugs.python.org/issue?@action=redirect&bpo=31588]: Raise a `TypeError` with a helpful

error message when class creation fails due to a metaclass with a bad `__prepare__()` method. Patch by Oren Milman.

- bpo-31574 [https://bugs.python.org/issue?@action=redirect&bpo=31574]: Importlib was instrumented with two dtrace probes to profile import timing.
- bpo-31566 [https://bugs.python.org/issue?@action=redirect&bpo=31566]: Fix an assertion failure in `_warnings.warn()` in case of a bad `__name__` global. Patch by Oren Milman.
- bpo-31506 [https://bugs.python.org/issue?@action=redirect&bpo=31506]: Improved the error message logic for `object.__new__` and `object.__init__`.
- bpo-31505 [https://bugs.python.org/issue?@action=redirect&bpo=31505]: Fix an assertion failure in `json`, in case `_json.make_encoder()` received a bad `encoder()` argument. Patch by Oren Milman.
- bpo-31492 [https://bugs.python.org/issue?@action=redirect&bpo=31492]: Fix assertion failures in case of failing to import from a module with a bad `__name__` attribute, and in case of failing to access an attribute of such a module. Patch by Oren Milman.
- bpo-31478 [https://bugs.python.org/issue?@action=redirect&bpo=31478]: Fix an assertion failure in `_random.Random.seed()` in case the argument has a bad `__abs__()` method. Patch by Oren Milman.
- bpo-31336 [https://bugs.python.org/issue?@action=redirect&bpo=31336]: Speed up class creation by 10-20% by reducing the overhead in the necessary special method lookups. Patch by Stefan Behnel.
- bpo-31415 [https://bugs.python.org/issue?@action=redirect&bpo=31415]: Add `-X importtime` option to show how long each import takes. It can be used to optimize application’s startup time. Support the **`PYTHONPROFILEIMPORTTIME`** as an equivalent way to enable this.

- bpo-31410 [https://bugs.python.org/issue?@action=redirect&bpo=31410]: Optimized calling wrapper and classmethod descriptors.
- bpo-31353 [https://bugs.python.org/issue?@action=redirect&bpo=31353]: **PEP 553** [https://peps.python.org/pep-0553/] - Add a new built-in called `breakpoint()` which calls `sys.breakpointhook()`. By default this imports `pdb` and calls `pdb.set_trace()`, but users may override `sys.breakpointhook()` to call whatever debugger they want. The original value of the hook is saved in `sys.__breakpointhook__`.
- bpo-17852 [https://bugs.python.org/issue?@action=redirect&bpo=17852]: Maintain a list of open buffered files, flush them before exiting the interpreter. Based on a patch from Armin Rigo.
- bpo-31315 [https://bugs.python.org/issue?@action=redirect&bpo=31315]: Fix an assertion failure in imp.create_dynamic(), when spec.name is not a string. Patch by Oren Milman.
- bpo-31311 [https://bugs.python.org/issue?@action=redirect&bpo=31311]: Fix a crash in the `__setstate__()` method of `ctypes._CData`, in case of a bad `__dict__`. Patch by Oren Milman.
- bpo-31293 [https://bugs.python.org/issue?@action=redirect&bpo=31293]: Fix crashes in true division and multiplication of a timedelta object by a float with a bad as_integer_ratio() method. Patch by Oren Milman.
- bpo-31285 [https://bugs.python.org/issue?@action=redirect&bpo=31285]: Fix an assertion failure in `warnings.warn_explicit`, when the return value of the received loader’s `get_source()` has a bad `splitlines()` method. Patch by Oren Milman.
- bpo-30406 [https://bugs.python.org/issue?@action=redirect&bpo=30406]: Make `async` and `await` proper keywords, as specified in **PEP 492** [https://peps.python.org/pep-0492/].

Library

- bpo-30058 [https://bugs.python.org/issue?@action=redirect&bpo=30058]: Fixed buffer overflow in select.kqueue.control().
- bpo-31672 [https://bugs.python.org/issue?@action=redirect&bpo=31672]: `idpattern` in `string.Template` matched some non-ASCII characters. Now it uses `-i` regular expression local flag to avoid non-ASCII characters.
- bpo-31701 [https://bugs.python.org/issue?@action=redirect&bpo=31701]: On Windows, faulthandler.enable() now ignores MSC and COM exceptions.
- bpo-31728 [https://bugs.python.org/issue?@action=redirect&bpo=31728]: Prevent crashes in `_elementtree` due to unsafe cleanup of `Element.text` and `Element.tail`. Patch by Oren Milman.
- bpo-31671 [https://bugs.python.org/issue?@action=redirect&bpo=31671]: Now `re.compile()` converts passed RegexFlag to normal int object before compiling. bm_regex_compile benchmark shows 14% performance improvements.
- bpo-30397 [https://bugs.python.org/issue?@action=redirect&bpo=30397]: The types of compiled regular objects and match objects are now exposed as `re.Pattern` and `re.Match`. This adds information in pydoc output for the `re` module.
- bpo-31675 [https://bugs.python.org/issue?@action=redirect&bpo=31675]: Fixed memory leaks in Tkinter’s methods splitlist() and split() when pass a string larger than 2 GiB.
- bpo-31673 [https://bugs.python.org/issue?@action=redirect&bpo=31673]: Fixed typo in the name of Tkinter’s method adderrorinfo().

- bpo-31648 [https://bugs.python.org/issue?@action=redirect&bpo=31648]: Improvements to path predicates in ElementTree: Allow whitespace around predicate parts, i.e. “[a = ‘text’]” instead of requiring the less readable “[a=’text’]”. Add support for text comparison of the current node, like “[.=’text’]”. Patch by Stefan Behnel.
- bpo-30806 [https://bugs.python.org/issue?@action=redirect&bpo=30806]: Fix the string representation of a netrc object.
- bpo-31638 [https://bugs.python.org/issue?@action=redirect&bpo=31638]: Add optional argument `compressed` to `zipapp.create_archive`, and add option `--compress` to the command line interface of `zipapp`.
- bpo-25351 [https://bugs.python.org/issue?@action=redirect&bpo=25351]: Avoid venv activate failures with undefined variables
- bpo-20519 [https://bugs.python.org/issue?@action=redirect&bpo=20519]: Avoid ctypes use (if possible) and improve import time for uuid.
- bpo-28293 [https://bugs.python.org/issue?@action=redirect&bpo=28293]: The regular expression cache is no longer completely dumped when it is full.
- bpo-31596 [https://bugs.python.org/issue?@action=redirect&bpo=31596]: Added pthread_getcpuclockid() to the time module
- bpo-27494 [https://bugs.python.org/issue?@action=redirect&bpo=27494]: Make 2to3 accept a trailing comma in generator expressions. For example, `set(x for x in [],)` is now allowed.
- bpo-30347 [https://bugs.python.org/issue?@action=redirect&bpo=30347]: Stop crashes when concurrently iterate over itertools.groupby() iterators.
- bpo-30346 [https://bugs.python.org/issue?@action=redirect&bpo=30346]: An iterator produced by

itertools.groupby() iterator now becomes exhausted after advancing the groupby iterator.

- bpo-31556 [https://bugs.python.org/issue?@action=redirect&bpo=31556]: Cancel asyncio.wait_for future faster if timeout <= 0
- bpo-31540 [https://bugs.python.org/issue?@action=redirect&bpo=31540]: Allow passing a context object in `concurrent.futures.ProcessPoolExecutor` constructor. Also, free job resources in `concurrent.futures.ProcessPoolExecutor` earlier to improve memory usage when a worker waits for new jobs.
- bpo-31516 [https://bugs.python.org/issue?@action=redirect&bpo=31516]: `threading.current_thread()` should not return a dummy thread at shutdown.
- bpo-31525 [https://bugs.python.org/issue?@action=redirect&bpo=31525]: In the sqlite module, require the sqlite3_prepare_v2 API. Thus, the sqlite module now requires sqlite version at least 3.3.9.
- bpo-26510 [https://bugs.python.org/issue?@action=redirect&bpo=26510]: argparse subparsers are now required by default. This matches behaviour in Python 2. For optional subparsers, use the new parameter `add_subparsers(required=False)`. Patch by Anthony Sottile. (As of 3.7.0rc1, the default was changed to not required as had been the case since Python 3.3.)
- bpo-27541 [https://bugs.python.org/issue?@action=redirect&bpo=27541]: Reprs of subclasses of some collection and iterator classes (`bytearray`, `array.array`, `collections.deque`, `collections.defaultdict`, `itertools.count`, `itertools.repeat`) now contain actual type name instead of hardcoded names of the base class.
- bpo-31351 [https://bugs.python.org/issue?@action=redirect&bpo=31351]: python -m ensurepip now exits with non-zero exit code if pip bootstrapping has failed.
- bpo-31389 [https://bugs.python.org/issue?@action=redirect&bpo=31389]: `pdb.set_trace()` now takes an

optional keyword-only argument `header`. If given, this is printed to the console just before debugging begins.

Documentation

- bpo-31537 [https://bugs.python.org/issue?@action=redirect&bpo=31537]: Fix incorrect usage of `get_history_length` in readline documentation example code. Patch by Brad Smith.
- bpo-30085 [https://bugs.python.org/issue?@action=redirect&bpo=30085]: The operator functions without double underscores are preferred for clarity. The one with underscores are only kept for back-compatibility.

Build

- bpo-31696 [https://bugs.python.org/issue?@action=redirect&bpo=31696]: Improve compiler version information in `sys.version` when Python is built with Clang.
- bpo-31625 [https://bugs.python.org/issue?@action=redirect&bpo=31625]: Stop using ranlib on static libraries. Instead, we assume ar supports the ‘s’ flag.
- bpo-31624 [https://bugs.python.org/issue?@action=redirect&bpo=31624]: Remove support for BSD/OS.
- bpo-22140 [https://bugs.python.org/issue?@action=redirect&bpo=22140]: Prevent double substitution of prefix in python-config.sh.
- bpo-31569 [https://bugs.python.org/issue?@action=redirect&bpo=31569]: Correct PCBuild/ case to PCbuild/ in build scripts and documentation.
- bpo-31536 [https://bugs.python.org/issue?@action=redirect&bpo=31536]: Avoid wholesale rebuild after `make regen-all` if nothing changed.

IDLE

- bpo-31460 [https://bugs.python.org/issue?@action=redirect&bpo=31460]: Simplify the API of IDLE's Module Browser. Passing a widget instead of an flist with a root widget opens the option of creating a browser frame that is only part of a window. Passing a full file name instead of pieces assumed to come from a .py file opens the possibility of browsing python files that do not end in .py.
- bpo-31649 [https://bugs.python.org/issue?@action=redirect&bpo=31649]: IDLE - Make _htest, _utest parameters keyword only.
- bpo-31559 [https://bugs.python.org/issue?@action=redirect&bpo=31559]: Remove test order dependence in idle_test.test_browser.
- bpo-31459 [https://bugs.python.org/issue?@action=redirect&bpo=31459]: Rename IDLE's module browser from Class Browser to Module Browser. The original module-level class and method browser became a module browser, with the addition of module-level functions, years ago. Nested classes and functions were added yesterday. For back-compatibility, the virtual event <<open-class-browser>>, which appears on the Keys tab of the Settings dialog, is not changed. Patch by Cheryl Sabella.
- bpo-31500 [https://bugs.python.org/issue?@action=redirect&bpo=31500]: Default fonts now are scaled on HiDPI displays.
- bpo-1612262 [https://bugs.python.org/issue?@action=redirect&bpo=1612262]: IDLE module browser now shows nested classes and functions. Original patches for code and tests by Guilherme Polo and Cheryl Sabella, respectively.

C API

- bpo-28280 [https://bugs.python.org/issue?@action=redirect&bpo=28280]: Make `PyMapping_Keys()`, `PyMapping_Values()` and `PyMapping_Items()` always return a `list` (rather than a `list` or a `tuple`). Patch by Oren Milman.
- bpo-31532 [https://bugs.python.org/issue?@action=redirect&bpo=31532]: Fix memory corruption due to allocator mix in getpath.c between Py_GetPath() and Py_SetPath()
- bpo-25658 [https://bugs.python.org/issue?@action=redirect&bpo=25658]: Implement **PEP 539** [https://peps.python.org/pep-0539/] for Thread Specific Storage (TSS) API: it is a new Thread Local Storage (TLS) API to CPython which would supersede use of the existing TLS API within the CPython interpreter, while deprecating the existing API. PEP written by Erik M. Bray, patch by Masayuki Yamamoto.

Python 3.7.0 alpha 1

Release date: 2017-09-19

Security

- bpo-29781 [https://bugs.python.org/issue?@action=redirect&bpo=29781]: SSLObject.version() now correctly returns None when handshake over BIO has not been performed yet.
- bpo-29505 [https://bugs.python.org/issue?@action=redirect&bpo=29505]: Add fuzz tests for float(str), int(str), unicode(str); for oss-fuzz.
- bpo-30947 [https://bugs.python.org/issue?@action=redirect&bpo=30947]: Upgrade libexpat embedded

copy from version 2.2.1 to 2.2.3 to get security fixes.

- bpo-30730 [https://bugs.python.org/issue?@action=redirect&bpo=30730]: Prevent environment variables injection in subprocess on Windows. Prevent passing other environment variables and command arguments.
- bpo-30694 [https://bugs.python.org/issue?@action=redirect&bpo=30694]: Upgrade expat copy from 2.2.0 to 2.2.1 to get fixes of multiple security vulnerabilities including: **CVE 2017-9233** [https://www.cve.org/CVERecord?id=CVE-2017-9233] (External entity infinite loop DoS), **CVE 2016-9063** [https://www.cve.org/CVERecord?id=CVE-2016-9063] (Integer overflow, re-fix), **CVE 2016-0718** [https://www.cve.org/CVERecord?id=CVE-2016-0718] (Fix regression bugs from 2.2.0's fix to **CVE 2016-0718** [https://www.cve.org/CVERecord?id=CVE-2016-0718]) and **CVE 2012-0876** [https://www.cve.org/CVERecord?id=CVE-2012-0876] (Counter hash flooding with SipHash). Note: the **CVE 2016-5300** [https://www.cve.org/CVERecord?id=CVE-2016-5300] (Use os-specific entropy sources like getrandom) doesn't impact Python, since Python already gets entropy from the OS to set the expat secret using `XML_SetHashSalt()`.
- bpo-30500 [https://bugs.python.org/issue?@action=redirect&bpo=30500]: Fix urllib.parse.splithost() to correctly parse fragments. For example, `splithost('//127.0.0.1#@evil.com/')` now correctly returns the `127.0.0.1` host, instead of treating `@evil.com` as the host in an authentication (`login@host`).
- bpo-29591 [https://bugs.python.org/issue?@action=redirect&bpo=29591]: Update expat copy from 2.1.1 to 2.2.0 to get fixes of **CVE 2016-0718** [https://www.cve.org/CVERecord?id=CVE-2016-0718] and **CVE 2016-4472** [https://www.cve.org/CVERecord?id=CVE-2016-4472]. See https://sourceforge.net/p/expat/bugs/537/ for more information.

Core and Builtins

- bpo-31490 [https://bugs.python.org/issue?@action=redirect&bpo=31490]: Fix an assertion failure in `ctypes` class definition, in case the class has an attribute whose name is specified in `_anonymous_` but not in `_fields_`. Patch by Oren Milman.
- bpo-31471 [https://bugs.python.org/issue?@action=redirect&bpo=31471]: Fix an assertion failure in `subprocess.Popen()` on Windows, in case the env argument has a bad `keys()` method. Patch by Oren Milman.
- bpo-31418 [https://bugs.python.org/issue?@action=redirect&bpo=31418]: Fix an assertion failure in `PyErr_WriteUnraisable()` in case of an exception with a bad `__module__` attribute. Patch by Oren Milman.
- bpo-31416 [https://bugs.python.org/issue?@action=redirect&bpo=31416]: Fix assertion failures in case of a bad warnings.filters or warnings.defaultaction. Patch by Oren Milman.
- bpo-28411 [https://bugs.python.org/issue?@action=redirect&bpo=28411]: Change direct usage of PyInterpreterState.modules to PyImport_GetModuleDict(). Also introduce more uniformity in other code that deals with sys.modules. This helps reduce complications when working on sys.modules.
- bpo-28411 [https://bugs.python.org/issue?@action=redirect&bpo=28411]: Switch to the abstract API when dealing with `PyInterpreterState.modules`. This allows later support for all dict subclasses and other Mapping implementations. Also add a `PyImport_GetModule()` function to reduce a bunch of duplicated code.
- bpo-31411 [https://bugs.python.org/issue?@action=redirect&bpo=31411]: Raise a TypeError instead of

SystemError in case warnings.onceregistry is not a dictionary. Patch by Oren Milman.

- bpo-31344 [https://bugs.python.org/issue?@action=redirect&bpo=31344]: For finer control of tracing behaviour when testing the interpreter, two new frame attributes have been added to control the emission of particular trace events: `f_trace_lines` (`True` by default) to turn off per-line trace events; and `f_trace_opcodes` (`False` by default) to turn on per-opcode trace events.
- bpo-31373 [https://bugs.python.org/issue?@action=redirect&bpo=31373]: Fix several possible instances of undefined behavior due to floating-point demotions.
- bpo-30465 [https://bugs.python.org/issue?@action=redirect&bpo=30465]: Location information (`lineno` and `col_offset`) in f-strings is now (mostly) correct. This fixes tools like flake8 from showing warnings on the wrong line (typically the first line of the file).
- bpo-30860 [https://bugs.python.org/issue?@action=redirect&bpo=30860]: Consolidate CPython’s global runtime state under a single struct. This improves discoverability of the runtime state.
- bpo-31347 [https://bugs.python.org/issue?@action=redirect&bpo=31347]: Fix possible undefined behavior in _PyObject_FastCall_Prepend.
- bpo-31343 [https://bugs.python.org/issue?@action=redirect&bpo=31343]: Include sys/sysmacros.h for major(), minor(), and makedev(). GNU C library plans to remove the functions from sys/types.h.
- bpo-31291 [https://bugs.python.org/issue?@action=redirect&bpo=31291]: Fix an assertion failure in `zipimport.zipimporter.get_data` on Windows, when the return value of `pathname.replace('/','\\')` isn’t a string. Patch by Oren Milman.
- bpo-31271 [https://bugs.python.org/issue?@action=redirect&bpo=31271]: Fix an assertion failure in the

`write()` method of `io.TextIOWrapper`, when the encoder doesn’t return a bytes object. Patch by Oren Milman.

- bpo-31243 [https://bugs.python.org/issue?@action=redirect&bpo=31243]: Fix a crash in some methods of `io.TextIOWrapper`, when the decoder’s state is invalid. Patch by Oren Milman.
- bpo-30721 [https://bugs.python.org/issue?@action=redirect&bpo=30721]: `print` now shows correct usage hint for using Python 2 redirection syntax. Patch by Sanyam Khurana.
- bpo-31070 [https://bugs.python.org/issue?@action=redirect&bpo=31070]: Fix a race condition in importlib _get_module_lock().
- bpo-30747 [https://bugs.python.org/issue?@action=redirect&bpo=30747]: Add a non-dummy implementation of _Py_atomic_store and _Py_atomic_load on MSVC.
- bpo-31095 [https://bugs.python.org/issue?@action=redirect&bpo=31095]: Fix potential crash during GC caused by `tp_dealloc` which doesn’t call `PyObject_GC_UnTrack()`.
- bpo-31071 [https://bugs.python.org/issue?@action=redirect&bpo=31071]: Avoid masking original TypeError in call with * unpacking when other arguments are passed.
- bpo-30978 [https://bugs.python.org/issue?@action=redirect&bpo=30978]: str.format_map() now passes key lookup exceptions through. Previously any exception was replaced with a KeyError exception.
- bpo-30808 [https://bugs.python.org/issue?@action=redirect&bpo=30808]: Use _Py_atomic API for concurrency-sensitive signal state.
- bpo-30876 [https://bugs.python.org/issue?@action=redirect&bpo=30876]: Relative import from unloaded package now reimports the package instead of failing

with SystemError. Relative import from non-package now fails with ImportError rather than SystemError.

- bpo-30703 [https://bugs.python.org/issue?@action=redirect&bpo=30703]: Improve signal delivery. Avoid using Py_AddPendingCall from signal handler, to avoid calling signal-unsafe functions. The tests I'm adding here fail without the rest of the patch, on Linux and OS X. This means our signal delivery logic had defects (some signals could be lost).
- bpo-30765 [https://bugs.python.org/issue?@action=redirect&bpo=30765]: Avoid blocking in pthread_mutex_lock() when PyThread_acquire_lock() is asked not to block.
- bpo-31161 [https://bugs.python.org/issue?@action=redirect&bpo=31161]: Make sure the 'Missing parentheses' syntax error message is only applied to SyntaxError, not to subclasses. Patch by Martijn Pieters.
- bpo-30814 [https://bugs.python.org/issue?@action=redirect&bpo=30814]: Fixed a race condition when import a submodule from a package.
- bpo-30736 [https://bugs.python.org/issue?@action=redirect&bpo=30736]: The internal unicodedata database has been upgraded to Unicode 10.0.
- bpo-30604 [https://bugs.python.org/issue?@action=redirect&bpo=30604]: Move co_extra_freefuncs from per-thread to per-interpreter to avoid crashes.
- bpo-30597 [https://bugs.python.org/issue?@action=redirect&bpo=30597]: `print` now shows expected input in custom error message when used as a Python 2 statement. Patch by Sanyam Khurana.
- bpo-30682 [https://bugs.python.org/issue?@action=redirect&bpo=30682]: Removed a too-strict assertion that failed for certain f-strings, such as eval("f'\n'") and eval("f'\r'").
- bpo-30501 [https://bugs.python.org/issue?@action=redirect&bpo=30501]: The compiler now produces

more optimal code for complex condition expressions in the “if”, “while” and “assert” statement, the “if” expression, and generator expressions and comprehensions.

- bpo-28180 [https://bugs.python.org/issue?@action=redirect&bpo=28180]: Implement **PEP 538** [https://peps.python.org/pep-0538/] (legacy C locale coercion). This means that when a suitable coercion target locale is available, both the core interpreter and locale-aware C extensions will assume the use of UTF-8 as the default text encoding, rather than ASCII.
- bpo-30486 [https://bugs.python.org/issue?@action=redirect&bpo=30486]: Allows setting cell values for __closure__. Patch by Lisa Roach.
- bpo-30537 [https://bugs.python.org/issue?@action=redirect&bpo=30537]: itertools.islice now accepts integer-like objects (having an __index__ method) as start, stop, and slice arguments
- bpo-25324 [https://bugs.python.org/issue?@action=redirect&bpo=25324]: Tokens needed for parsing in Python moved to C. `COMMENT`, `NL` and `ENCODING`. This way the tokens and tok_names in the token module don’t get changed when you import the tokenize module.
- bpo-29104 [https://bugs.python.org/issue?@action=redirect&bpo=29104]: Fixed parsing backslashes in f-strings.
- bpo-27945 [https://bugs.python.org/issue?@action=redirect&bpo=27945]: Fixed various segfaults with dict when input collections are mutated during searching, inserting or comparing. Based on patches by Duane Griffin and Tim Mitchell.
- bpo-25794 [https://bugs.python.org/issue?@action=redirect&bpo=25794]: Fixed type.__setattr__() and type.__delattr__() for non-interned attribute names. Based on patch by Eryk Sun.

- bpo-30039 [https://bugs.python.org/issue?@action=redirect&bpo=30039]: If a KeyboardInterrupt happens when the interpreter is in the middle of resuming a chain of nested ‘yield from’ or ‘await’ calls, it’s now correctly delivered to the innermost frame.
- bpo-28974 [https://bugs.python.org/issue?@action=redirect&bpo=28974]: `object.__format__(x, '')` is now equivalent to `str(x)` rather than `format(str(self), '')`.
- bpo-30024 [https://bugs.python.org/issue?@action=redirect&bpo=30024]: Circular imports involving absolute imports with binding a submodule to a name are now supported.
- bpo-12414 [https://bugs.python.org/issue?@action=redirect&bpo=12414]: sys.getsizeof() on a code object now returns the sizes which includes the code struct and sizes of objects which it references. Patch by Donghee Na.
- bpo-29839 [https://bugs.python.org/issue?@action=redirect&bpo=29839]: len() now raises ValueError rather than OverflowError if __len__() returned a large negative integer.
- bpo-11913 [https://bugs.python.org/issue?@action=redirect&bpo=11913]: README.rst is now included in the list of distutils standard READMEs and therefore included in source distributions.
- bpo-29914 [https://bugs.python.org/issue?@action=redirect&bpo=29914]: Fixed default implementations of __reduce__ and __reduce_ex__(). object.__reduce__() no longer takes arguments, object.__reduce_ex__() now requires one argument.
- bpo-29949 [https://bugs.python.org/issue?@action=redirect&bpo=29949]: Fix memory usage regression of set and frozenset object.
- bpo-29935 [https://bugs.python.org/issue?@action=redirect&bpo=29935]: Fixed error messages in the

index() method of tuple, list and deque when pass indices of wrong type.

- bpo-29816 [https://bugs.python.org/issue?@action=redirect&bpo=29816]: Shift operation now has less opportunity to raise OverflowError. ValueError always is raised rather than OverflowError for negative counts. Shifting zero with non-negative count always returns zero.
- bpo-24821 [https://bugs.python.org/issue?@action=redirect&bpo=24821]: Fixed the slowing down to 25 times in the searching of some unlucky Unicode characters.
- bpo-29102 [https://bugs.python.org/issue?@action=redirect&bpo=29102]: Add a unique ID to PyInterpreterState. This makes it easier to identify each subinterpreter.
- bpo-29894 [https://bugs.python.org/issue?@action=redirect&bpo=29894]: The deprecation warning is emitted if __complex__ returns an instance of a strict subclass of complex. In a future versions of Python this can be an error.
- bpo-29859 [https://bugs.python.org/issue?@action=redirect&bpo=29859]: Show correct error messages when any of the pthread_* calls in thread_pthread.h fails.
- bpo-29849 [https://bugs.python.org/issue?@action=redirect&bpo=29849]: Fix a memory leak when an ImportError is raised during from import.
- bpo-28856 [https://bugs.python.org/issue?@action=redirect&bpo=28856]: Fix an oversight that %b format for bytes should support objects follow the buffer protocol.
- bpo-29723 [https://bugs.python.org/issue?@action=redirect&bpo=29723]: The `sys.path[0]` initialization change for bpo-29139 [https://bugs.python.org/issue?@action=redirect&bpo=29139] caused a regression by

revealing an inconsistency in how sys.path is initialized when executing `__main__` from a zipfile, directory, or other import location. The interpreter now consistently avoids ever adding the import location's parent directory to `sys.path`, and ensures no other `sys.path` entries are inadvertently modified when inserting the import location named on the command line.
- bpo-29568 [https://bugs.python.org/issue?@action=redirect&bpo=29568]: Escaped percent "%%" in the format string for classic string formatting no longer allows any characters between two percents.
- bpo-29714 [https://bugs.python.org/issue?@action=redirect&bpo=29714]: Fix a regression that bytes format may fail when containing zero bytes inside.
- bpo-29695 [https://bugs.python.org/issue?@action=redirect&bpo=29695]: bool(), float(), list() and tuple() no longer take keyword arguments. The first argument of int() can now be passes only as positional argument.
- bpo-28893 [https://bugs.python.org/issue?@action=redirect&bpo=28893]: Set correct __cause__ for errors about invalid awaitables returned from __aiter__ and __anext__.
- bpo-28876 [https://bugs.python.org/issue?@action=redirect&bpo=28876]: `bool(range)` works even if `len(range)` raises **`OverflowError`**.
- bpo-29683 [https://bugs.python.org/issue?@action=redirect&bpo=29683]: Fixes to memory allocation in _PyCode_SetExtra. Patch by Brian Coleman.
- bpo-29684 [https://bugs.python.org/issue?@action=redirect&bpo=29684]: Fix minor regression of PyEval_CallObjectWithKeywords. It should raise TypeError when kwargs is not a dict. But it might cause segv when args=NULL and kwargs is not a dict.
- bpo-28598 [https://bugs.python.org/issue?@action=redirect&bpo=28598]: Support __rmod__ for

subclasses of str being called before str.__mod__. Patch by Martijn Pieters.
- bpo-29607 [https://bugs.python.org/issue?@action=redirect&bpo=29607]: Fix stack_effect computation for CALL_FUNCTION_EX. Patch by Matthieu Dartiailh.
- bpo-29602 [https://bugs.python.org/issue?@action=redirect&bpo=29602]: Fix incorrect handling of signed zeros in complex constructor for complex subclasses and for inputs having a __complex__ method. Patch by Serhiy Storchaka.
- bpo-29347 [https://bugs.python.org/issue?@action=redirect&bpo=29347]: Fixed possibly dereferencing undefined pointers when creating weakref objects.
- bpo-29463 [https://bugs.python.org/issue?@action=redirect&bpo=29463]: Add `docstring` field to Module, ClassDef, FunctionDef, and AsyncFunctionDef ast nodes. docstring is not first stmt in their body anymore. It affects `co_firstlineno` and `co_lnotab` of code object for module and class. (Reverted in bpo-32911 [https://bugs.python.org/issue?@action=redirect&bpo=32911].)
- bpo-29438 [https://bugs.python.org/issue?@action=redirect&bpo=29438]: Fixed use-after-free problem in key sharing dict.
- bpo-29546 [https://bugs.python.org/issue?@action=redirect&bpo=29546]: Set the ‘path’ and ‘name’ attribute on ImportError for `from ... import ...`.
- bpo-29546 [https://bugs.python.org/issue?@action=redirect&bpo=29546]: Improve from-import error message with location
- bpo-29478 [https://bugs.python.org/issue?@action=redirect&bpo=29478]: If max_line_length=None is specified while using the Compat32 policy, it is no longer ignored. Patch by Mircea Cosbuc.
- bpo-29319 [https://bugs.python.org/issue?@action=redirect&bpo=29319]: Prevent RunMainFromImporter overwriting sys.path[0].

- bpo-29337 [https://bugs.python.org/issue?@action=redirect&bpo=29337]: Fixed possible BytesWarning when compare the code objects. Warnings could be emitted at compile time.
- bpo-29327 [https://bugs.python.org/issue?@action=redirect&bpo=29327]: Fixed a crash when pass the iterable keyword argument to sorted().
- bpo-29034 [https://bugs.python.org/issue?@action=redirect&bpo=29034]: Fix memory leak and use-after-free in os module (path_converter).
- bpo-29159 [https://bugs.python.org/issue?@action=redirect&bpo=29159]: Fix regression in bytes(x) when x.__index__() raises Exception.
- bpo-29049 [https://bugs.python.org/issue?@action=redirect&bpo=29049]: Call _PyObject_GC_TRACK() lazily when calling Python function. Calling function is up to 5% faster.
- bpo-28927 [https://bugs.python.org/issue?@action=redirect&bpo=28927]: bytes.fromhex() and bytearray.fromhex() now ignore all ASCII whitespace, not only spaces. Patch by Robert Xiao.
- bpo-28932 [https://bugs.python.org/issue?@action=redirect&bpo=28932]: Do not include <sys/random.h> if it does not exist.
- bpo-25677 [https://bugs.python.org/issue?@action=redirect&bpo=25677]: Correct the positioning of the syntax error caret for indented blocks. Based on patch by Michael Layzell.
- bpo-29000 [https://bugs.python.org/issue?@action=redirect&bpo=29000]: Fixed bytes formatting of octals with zero padding in alternate form.
- bpo-18896 [https://bugs.python.org/issue?@action=redirect&bpo=18896]: Python function can now have more than 255 parameters. collections.namedtuple() now supports tuples with more than 255 elements.

- bpo-28596 [https://bugs.python.org/issue?@action=redirect&bpo=28596]: The preferred encoding is UTF-8 on Android. Patch written by Chi Hsuan Yen.
- bpo-22257 [https://bugs.python.org/issue?@action=redirect&bpo=22257]: Clean up interpreter startup (see **PEP 432** [https://peps.python.org/pep-0432/]).
- bpo-26919 [https://bugs.python.org/issue?@action=redirect&bpo=26919]: On Android, operating system data is now always encoded/decoded to/from UTF-8, instead of the locale encoding to avoid inconsistencies with os.fsencode() and os.fsdecode() which are already using UTF-8.
- bpo-28991 [https://bugs.python.org/issue?@action=redirect&bpo=28991]: functools.lru_cache() was susceptible to an obscure reentrancy bug triggerable by a monkey-patched len() function.
- bpo-28147 [https://bugs.python.org/issue?@action=redirect&bpo=28147]: Fix a memory leak in split-table dictionaries: setattr() must not convert combined table into split table. Patch written by INADA Naoki.
- bpo-28739 [https://bugs.python.org/issue?@action=redirect&bpo=28739]: f-string expressions are no longer accepted as docstrings and by ast.literal_eval() even if they do not include expressions.
- bpo-28512 [https://bugs.python.org/issue?@action=redirect&bpo=28512]: Fixed setting the offset attribute of SyntaxError by PyErr_SyntaxLocationEx() and PyErr_SyntaxLocationObject().
- bpo-28918 [https://bugs.python.org/issue?@action=redirect&bpo=28918]: Fix the cross compilation of xxlimited when Python has been built with Py_DEBUG defined.
- bpo-23722 [https://bugs.python.org/issue?@action=redirect&bpo=23722]: Rather than silently producing a class that doesn’t support zero-argument `super()` in methods, failing to pass the new `__classcell__`

namespace entry up to `type.__new__` now results in a `DeprecationWarning` and a class that supports zero-argument `super()`.

- bpo-28797 [https://bugs.python.org/issue?@action=redirect&bpo=28797]: Modifying the class __dict__ inside the __set_name__ method of a descriptor that is used inside that class no longer prevents calling the __set_name__ method of other descriptors.
- bpo-28799 [https://bugs.python.org/issue?@action=redirect&bpo=28799]: Remove the `PyEval_GetCallStats()` function and deprecate the untested and undocumented `sys.callstats()` function. Remove the `CALL_PROFILE` special build: use the **`sys.setprofile()`** function, **`cProfile`** or **`profile`** to profile function calls.
- bpo-12844 [https://bugs.python.org/issue?@action=redirect&bpo=12844]: More than 255 arguments can now be passed to a function.
- bpo-28782 [https://bugs.python.org/issue?@action=redirect&bpo=28782]: Fix a bug in the implementation `yield from` when checking if the next instruction is YIELD_FROM. Regression introduced by WORDCODE (bpo-26647 [https://bugs.python.org/issue?@action=redirect&bpo=26647]).
- bpo-28774 [https://bugs.python.org/issue?@action=redirect&bpo=28774]: Fix error position of the unicode error in ASCII and Latin1 encoders when a string returned by the error handler contains multiple non-encodable characters (non-ASCII for the ASCII codec, characters out of the U+0000-U+00FF range for Latin1).
- bpo-28731 [https://bugs.python.org/issue?@action=redirect&bpo=28731]: Optimize _PyDict_NewPresized() to create correct size dict. Improve speed of dict literal with constant keys up to 30%.

- bpo-28532 [https://bugs.python.org/issue?@action=redirect&bpo=28532]: Show sys.version when -V option is supplied twice.
- bpo-27100 [https://bugs.python.org/issue?@action=redirect&bpo=27100]: The with-statement now checks for __enter__ before it checks for __exit__. This gives less confusing error messages when both methods are missing. Patch by Jonathan Ellington.
- bpo-28746 [https://bugs.python.org/issue?@action=redirect&bpo=28746]: Fix the set_inheritable() file descriptor method on platforms that do not have the ioctl FIOCLEX and FIONCLEX commands.
- bpo-26920 [https://bugs.python.org/issue?@action=redirect&bpo=26920]: Fix not getting the locale's charset upon initializing the interpreter, on platforms that do not have langinfo.
- bpo-28648 [https://bugs.python.org/issue?@action=redirect&bpo=28648]: Fixed crash in Py_DecodeLocale() in debug build on Mac OS X when decode astral characters. Patch by Xiang Zhang.
- bpo-28665 [https://bugs.python.org/issue?@action=redirect&bpo=28665]: Improve speed of the STORE_DEREF opcode by 40%.
- bpo-19398 [https://bugs.python.org/issue?@action=redirect&bpo=19398]: Extra slash no longer added to sys.path components in case of empty compile-time PYTHONPATH components.
- bpo-28621 [https://bugs.python.org/issue?@action=redirect&bpo=28621]: Sped up converting int to float by reusing faster bits counting implementation. Patch by Adrian Wielgosik.
- bpo-28580 [https://bugs.python.org/issue?@action=redirect&bpo=28580]: Optimize iterating split table values. Patch by Xiang Zhang.
- bpo-28583 [https://bugs.python.org/issue?@action=redirect&bpo=28583]: PyDict_SetDefault didn't

combine split table when needed. Patch by Xiang Zhang.

- bpo-28128 [https://bugs.python.org/issue?@action=redirect&bpo=28128]: Deprecation warning for invalid str and byte escape sequences now prints better information about where the error occurs. Patch by Serhiy Storchaka and Eric Smith.
- bpo-28509 [https://bugs.python.org/issue?@action=redirect&bpo=28509]: dict.update() no longer allocate unnecessary large memory.
- bpo-28426 [https://bugs.python.org/issue?@action=redirect&bpo=28426]: Fixed potential crash in PyUnicode_AsDecodedObject() in debug build.
- bpo-28517 [https://bugs.python.org/issue?@action=redirect&bpo=28517]: Fixed of-by-one error in the peephole optimizer that caused keeping unreachable code.
- bpo-28214 [https://bugs.python.org/issue?@action=redirect&bpo=28214]: Improved exception reporting for problematic __set_name__ attributes.
- bpo-23782 [https://bugs.python.org/issue?@action=redirect&bpo=23782]: Fixed possible memory leak in _PyTraceback_Add() and exception loss in PyTraceBack_Here().
- bpo-28183 [https://bugs.python.org/issue?@action=redirect&bpo=28183]: Optimize and cleanup dict iteration.
- bpo-26081 [https://bugs.python.org/issue?@action=redirect&bpo=26081]: Added C implementation of asyncio.Future. Original patch by Yury Selivanov.
- bpo-28379 [https://bugs.python.org/issue?@action=redirect&bpo=28379]: Added sanity checks and tests for PyUnicode_CopyCharacters(). Patch by Xiang Zhang.
- bpo-28376 [https://bugs.python.org/issue?@action=redirect&bpo=28376]: The type of long range iterator is now registered as Iterator. Patch by Oren Milman.

- bpo-28376 [https://bugs.python.org/issue?@action=redirect&bpo=28376]: Creating instances of range_iterator by calling range_iterator type now is disallowed. Calling iter() on range instance is the only way. Patch by Oren Milman.
- bpo-26906 [https://bugs.python.org/issue?@action=redirect&bpo=26906]: Resolving special methods of uninitialized type now causes implicit initialization of the type instead of a fail.
- bpo-18287 [https://bugs.python.org/issue?@action=redirect&bpo=18287]: PyType_Ready() now checks that tp_name is not NULL. Original patch by Niklas Koep.
- bpo-24098 [https://bugs.python.org/issue?@action=redirect&bpo=24098]: Fixed possible crash when AST is changed in process of compiling it.
- bpo-28201 [https://bugs.python.org/issue?@action=redirect&bpo=28201]: Dict reduces possibility of 2nd conflict in hash table when hashes have same lower bits.
- bpo-28350 [https://bugs.python.org/issue?@action=redirect&bpo=28350]: String constants with null character no longer interned.
- bpo-26617 [https://bugs.python.org/issue?@action=redirect&bpo=26617]: Fix crash when GC runs during weakref callbacks.
- bpo-27942 [https://bugs.python.org/issue?@action=redirect&bpo=27942]: String constants now interned recursively in tuples and frozensets.
- bpo-28289 [https://bugs.python.org/issue?@action=redirect&bpo=28289]: ImportError.__init__ now resets not specified attributes.
- bpo-21578 [https://bugs.python.org/issue?@action=redirect&bpo=21578]: Fixed misleading error message when ImportError called with invalid keyword args.
- bpo-28203 [https://bugs.python.org/issue?@action=redirect&bpo=28203]: Fix incorrect type in

complex(1.0, {2:3}) error message. Patch by Soumya Sharma.
- bpo-28086 [https://bugs.python.org/issue?@action=redirect&bpo=28086]: Single var-positional argument of tuple subtype was passed unscathed to the C-defined function. Now it is converted to exact tuple.
- bpo-28214 [https://bugs.python.org/issue?@action=redirect&bpo=28214]: Now __set_name__ is looked up on the class instead of the instance.
- bpo-27955 [https://bugs.python.org/issue?@action=redirect&bpo=27955]: Fallback on reading /dev/urandom device when the getrandom() syscall fails with EPERM, for example when blocked by SECCOMP.
- bpo-28192 [https://bugs.python.org/issue?@action=redirect&bpo=28192]: Don’t import readline in isolated mode.
- bpo-27441 [https://bugs.python.org/issue?@action=redirect&bpo=27441]: Remove some redundant assignments to ob_size in longobject.c. Thanks Oren Milman.
- bpo-27222 [https://bugs.python.org/issue?@action=redirect&bpo=27222]: Clean up redundant code in long_rshift function. Thanks Oren Milman.
- Upgrade internal unicode databases to Unicode version 9.0.0.
- bpo-28131 [https://bugs.python.org/issue?@action=redirect&bpo=28131]: Fix a regression in zipimport’s compile_source(). zipimport should use the same optimization level as the interpreter.
- bpo-28126 [https://bugs.python.org/issue?@action=redirect&bpo=28126]: Replace Py_MEMCPY with memcpy(). Visual Studio can properly optimize memcpy().
- bpo-28120 [https://bugs.python.org/issue?@action=redirect&bpo=28120]: Fix dict.pop() for split dictionary

when trying to remove a “pending key” (Not yet inserted in split-table). Patch by Xiang Zhang.

- bpo-26182 [https://bugs.python.org/issue?@action=redirect&bpo=26182]: Raise DeprecationWarning when async and await keywords are used as variable/attribute/class/function name.
- bpo-26182 [https://bugs.python.org/issue?@action=redirect&bpo=26182]: Fix a refleak in code that raises DeprecationWarning.
- bpo-28721 [https://bugs.python.org/issue?@action=redirect&bpo=28721]: Fix asynchronous generators aclose() and athrow() to handle StopAsyncIteration propagation properly.
- bpo-26110 [https://bugs.python.org/issue?@action=redirect&bpo=26110]: Speed-up method calls: add LOAD_METHOD and CALL_METHOD opcodes.

Library

- bpo-31499 [https://bugs.python.org/issue?@action=redirect&bpo=31499]: xml.etree: Fix a crash when a parser is part of a reference cycle.
- bpo-31482 [https://bugs.python.org/issue?@action=redirect&bpo=31482]: `random.seed()` now works with bytes in version=1
- bpo-28556 [https://bugs.python.org/issue?@action=redirect&bpo=28556]: typing.get_type_hints now finds the right globalns for classes and modules by default (when no `globalns` was specified by the caller).
- bpo-28556 [https://bugs.python.org/issue?@action=redirect&bpo=28556]: Speed improvements to the `typing` module. Original PRs by Ivan Levkivskyi and Mitar.
- bpo-31544 [https://bugs.python.org/issue?@action=redirect&bpo=31544]: The C accelerator module of ElementTree ignored exceptions raised when looking up TreeBuilder target methods in XMLParser().

- bpo-31234 [https://bugs.python.org/issue?@action=redirect&bpo=31234]: socket.create_connection() now fixes manually a reference cycle: clear the variable storing the last exception on success.
- bpo-31457 [https://bugs.python.org/issue?@action=redirect&bpo=31457]: LoggerAdapter objects can now be nested.
- bpo-31431 [https://bugs.python.org/issue?@action=redirect&bpo=31431]: SSLContext.check_hostname now automatically sets SSLContext.verify_mode to ssl.CERT_REQUIRED instead of failing with a ValueError.
- bpo-31233 [https://bugs.python.org/issue?@action=redirect&bpo=31233]: socketserver.ThreadingMixIn now keeps a list of non-daemonic threads to wait until all these threads complete in server_close().
- bpo-28638 [https://bugs.python.org/issue?@action=redirect&bpo=28638]: Changed the implementation strategy for collections.namedtuple() to substantially reduce the use of exec() in favor of precomputed methods. As a result, the *verbose* parameter and _*source* attribute are no longer supported. The benefits include 1) having a smaller memory footprint for applications using multiple named tuples, 2) faster creation of the named tuple class (approx 4x to 6x depending on how it is measured), and 3) minor speed-ups for instance creation using __new__, _make, and _replace. (The primary patch contributor is Jelle Zijlstra with further improvements by INADA Naoki, Serhiy Storchaka, and Raymond Hettinger.)
- bpo-31400 [https://bugs.python.org/issue?@action=redirect&bpo=31400]: Improves SSL error handling to avoid losing error numbers.
- bpo-27629 [https://bugs.python.org/issue?@action=redirect&bpo=27629]: Make return types of SSLContext.wrap_bio() and SSLContext.wrap_socket() customizable.

- bpo-28958 [https://bugs.python.org/issue?@action=redirect&bpo=28958]: ssl.SSLContext() now uses OpenSSL error information when a context cannot be instantiated.
- bpo-28182 [https://bugs.python.org/issue?@action=redirect&bpo=28182]: The SSL module now raises SSLCertVerificationError when OpenSSL fails to verify the peer’s certificate. The exception contains more information about the error.
- bpo-27340 [https://bugs.python.org/issue?@action=redirect&bpo=27340]: SSLSocket.sendall() now uses memoryview to create slices of data. This fixes support for all bytes-like object. It is also more efficient and avoids costly copies.
- bpo-14191 [https://bugs.python.org/issue?@action=redirect&bpo=14191]: A new function `argparse.ArgumentParser.parse_intermixed_args` provides the ability to parse command lines where there user intermixes options and positional arguments.
- bpo-31178 [https://bugs.python.org/issue?@action=redirect&bpo=31178]: Fix string concatenation bug in rare error path in the subprocess module
- bpo-31350 [https://bugs.python.org/issue?@action=redirect&bpo=31350]: Micro-optimize **`asyncio._get_running_loop()`** to become up to 10% faster.
- bpo-31170 [https://bugs.python.org/issue?@action=redirect&bpo=31170]: expat: Update libexpat from 2.2.3 to 2.2.4. Fix copying of partial characters for UTF-8 input (libexpat bug 115): https://github.com/libexpat/libexpat/issues/115
- bpo-29136 [https://bugs.python.org/issue?@action=redirect&bpo=29136]: Add TLS 1.3 cipher suites and OP_NO_TLSv1_3.
- bpo-1198569 [https://bugs.python.org/issue?@action=redirect&bpo=1198569]: `string.Template` subclasses can optionally define `braceidpattern` if they want to

specify different placeholder patterns inside and outside the braces. If None (the default) it falls back to `idpattern`.

- bpo-31326 [https://bugs.python.org/issue?@action=redirect&bpo=31326]: concurrent.futures.ProcessPoolExecutor.shutdown() now explicitly closes the call queue. Moreover, shutdown(wait=True) now also join the call queue thread, to prevent leaking a dangling thread.
- bpo-27144 [https://bugs.python.org/issue?@action=redirect&bpo=27144]: The `map()` and `as_completed()` iterators in `concurrent.futures` now avoid keeping a reference to yielded objects.
- bpo-31281 [https://bugs.python.org/issue?@action=redirect&bpo=31281]: Fix `fileinput.FileInput(files, inplace=True)` when `files` contain `pathlib.Path` objects.
- bpo-10746 [https://bugs.python.org/issue?@action=redirect&bpo=10746]: Fix ctypes producing wrong **PEP 3118** [https://peps.python.org/pep-3118/] type codes for integer types.
- bpo-27584 [https://bugs.python.org/issue?@action=redirect&bpo=27584]: `AF_VSOCK` has been added to the socket interface which allows communication between virtual machines and their host.
- bpo-22536 [https://bugs.python.org/issue?@action=redirect&bpo=22536]: The subprocess module now sets the filename when FileNotFoundError is raised on POSIX systems due to the executable or cwd not being found.
- bpo-29741 [https://bugs.python.org/issue?@action=redirect&bpo=29741]: Update some methods in the _pyio module to also accept integer types. Patch by Oren Milman.
- bpo-31249 [https://bugs.python.org/issue?@action=redirect&bpo=31249]: concurrent.futures: WorkItem.run() used by ThreadPoolExecutor now breaks

a reference cycle between an exception object and the WorkItem object.

- bpo-31247 [https://bugs.python.org/issue?@action=redirect&bpo=31247]: xmlrpc.server now explicitly breaks reference cycles when using sys.exc_info() in code handling exceptions.
- bpo-23835 [https://bugs.python.org/issue?@action=redirect&bpo=23835]: configparser: reading defaults in the `ConfigParser()` constructor is now using `read_dict()`, making its behavior consistent with the rest of the parser. Non-string keys and values in the defaults dictionary are now being implicitly converted to strings. Patch by James Tocknell.
- bpo-31238 [https://bugs.python.org/issue?@action=redirect&bpo=31238]: pydoc: the stop() method of the private ServerThread class now waits until DocServer.serve_until_quit() completes and then explicitly sets its docserver attribute to None to break a reference cycle.
- bpo-5001 [https://bugs.python.org/issue?@action=redirect&bpo=5001]: Many asserts in `multiprocessing` are now more informative, and some error types have been changed to more specific ones.
- bpo-31109 [https://bugs.python.org/issue?@action=redirect&bpo=31109]: Convert zipimport to use Argument Clinic.
- bpo-30102 [https://bugs.python.org/issue?@action=redirect&bpo=30102]: The ssl and hashlib modules now call OPENSSL_add_all_algorithms_noconf() on OpenSSL < 1.1.0. The function detects CPU features and enables optimizations on some CPU architectures such as POWER8. Patch is based on research from Gustavo Serra Scalet.
- bpo-18966 [https://bugs.python.org/issue?@action=redirect&bpo=18966]: Non-daemonic threads created

by a multiprocessing.Process are now joined on child exit.

- bpo-31183 [https://bugs.python.org/issue?@action=redirect&bpo=31183]: `dis` now works with asynchronous generator and coroutine objects. Patch by George Collins based on diagnosis by Luciano Ramalho.
- bpo-5001 [https://bugs.python.org/issue?@action=redirect&bpo=5001]: There are a number of uninformative asserts in the `multiprocessing` module, as noted in issue 5001. This change fixes two of the most potentially problematic ones, since they are in error-reporting code, in the `multiprocessing.managers.convert_to_error` function. (It also makes more informative a ValueError message.) The only potentially problematic change is that the AssertionError is now a TypeError; however, this should also help distinguish it from an AssertionError being *reported* by the function/its caller (such as in issue 31169). - Patch by Allen W. Smith (drallensmith on github).
- bpo-31185 [https://bugs.python.org/issue?@action=redirect&bpo=31185]: Fixed miscellaneous errors in asyncio speedup module.
- bpo-31151 [https://bugs.python.org/issue?@action=redirect&bpo=31151]: socketserver.ForkingMixIn.server_close() now waits until all child processes completed to prevent leaking zombie processes.
- bpo-31072 [https://bugs.python.org/issue?@action=redirect&bpo=31072]: Add an `include_file` parameter to `zipapp.create_archive()`
- bpo-24700 [https://bugs.python.org/issue?@action=redirect&bpo=24700]: Optimize array.array comparison. It is now from 10x up to 70x faster when comparing arrays holding values of the same integer type.

- bpo-31135 [https://bugs.python.org/issue?@action=redirect&bpo=31135]: ttk: fix the destroy() method of LabeledScale and OptionMenu classes. Call the parent destroy() method even if the used attribute doesn’t exist. The LabeledScale.destroy() method now also explicitly clears label and scale attributes to help the garbage collector to destroy all widgets.
- bpo-31107 [https://bugs.python.org/issue?@action=redirect&bpo=31107]: Fix `copyreg._slotnames()` mangled attribute calculation for classes whose name begins with an underscore. Patch by Shane Harvey.
- bpo-31080 [https://bugs.python.org/issue?@action=redirect&bpo=31080]: Allow `logging.config.fileConfig` to accept kwargs and/or args.
- bpo-30897 [https://bugs.python.org/issue?@action=redirect&bpo=30897]: `pathlib.Path` objects now include an `is_mount()` method (only implemented on POSIX). This is similar to `os.path.ismount(p)`. Patch by Cooper Ry Lees.
- bpo-31061 [https://bugs.python.org/issue?@action=redirect&bpo=31061]: Fixed a crash when using asyncio and threads.
- bpo-30987 [https://bugs.python.org/issue?@action=redirect&bpo=30987]: Added support for CAN ISO-TP protocol in the socket module.
- bpo-30522 [https://bugs.python.org/issue?@action=redirect&bpo=30522]: Added a `setStream` method to `logging.StreamHandler` to allow the stream to be set after creation.
- bpo-30502 [https://bugs.python.org/issue?@action=redirect&bpo=30502]: Fix handling of long oids in ssl. Based on patch by Christian Heimes.
- bpo-5288 [https://bugs.python.org/issue?@action=redirect&bpo=5288]: Support tzinfo objects with sub-minute offsets.
- bpo-30919 [https://bugs.python.org/issue?@action=redirect&bpo=30919]: Fix shared memory

performance regression in multiprocessing in 3.x. Shared memory used anonymous memory mappings in 2.x, while 3.x mmaps actual files. Try to be careful to do as little disk I/O as possible.

- bpo-26732 [https://bugs.python.org/issue?@action=redirect&bpo=26732]: Fix too many fds in processes started with the “forkserver” method. A child process would inherit as many fds as the number of still-running children.
- bpo-29403 [https://bugs.python.org/issue?@action=redirect&bpo=29403]: Fix `unittest.mock`’s autospec to not fail on method-bound builtin functions. Patch by Aaron Gallagher.
- bpo-30961 [https://bugs.python.org/issue?@action=redirect&bpo=30961]: Fix decrementing a borrowed reference in tracemalloc.
- bpo-19896 [https://bugs.python.org/issue?@action=redirect&bpo=19896]: Fix multiprocessing.sharedctypes to recognize typecodes `'q'` and `'Q'`.
- bpo-30946 [https://bugs.python.org/issue?@action=redirect&bpo=30946]: Remove obsolete code in readline module for platforms where GNU readline is older than 2.1 or where select() is not available.
- bpo-25684 [https://bugs.python.org/issue?@action=redirect&bpo=25684]: Change `ttk.OptionMenu` radiobuttons to be unique across instances of `OptionMenu`.
- bpo-30886 [https://bugs.python.org/issue?@action=redirect&bpo=30886]: Fix multiprocessing.Queue.join_thread(): it now waits until the thread completes, even if the thread was started by the same process which created the queue.
- bpo-29854 [https://bugs.python.org/issue?@action=redirect&bpo=29854]: Fix segfault in readline when using readline’s history-size option. Patch by Nir Soffer.

- bpo-30794 [https://bugs.python.org/issue?@action=redirect&bpo=30794]: Added multiprocessing.Process.kill method to terminate using the SIGKILL signal on Unix.
- bpo-30319 [https://bugs.python.org/issue?@action=redirect&bpo=30319]: socket.close() now ignores ECONNRESET error.
- bpo-30828 [https://bugs.python.org/issue?@action=redirect&bpo=30828]: Fix out of bounds write in `asyncio.CFuture.remove_done_callback()`.
- bpo-30302 [https://bugs.python.org/issue?@action=redirect&bpo=30302]: Use keywords in the `repr` of `datetime.timedelta`.
- bpo-30807 [https://bugs.python.org/issue?@action=redirect&bpo=30807]: signal.setitimer() may disable the timer when passed a tiny value. Tiny values (such as 1e-6) are valid non-zero values for setitimer(), which is specified as taking microsecond-resolution intervals. However, on some platform, our conversion routine could convert 1e-6 into a zero interval, therefore disabling the timer instead of (re-)scheduling it.
- bpo-30441 [https://bugs.python.org/issue?@action=redirect&bpo=30441]: Fix bug when modifying os.environ while iterating over it
- bpo-29585 [https://bugs.python.org/issue?@action=redirect&bpo=29585]: Avoid importing `sysconfig` from `site` to improve startup speed. Python startup is about 5% faster on Linux and 30% faster on macOS.
- bpo-29293 [https://bugs.python.org/issue?@action=redirect&bpo=29293]: Add missing parameter “n” on multiprocessing.Condition.notify(). The doc claims multiprocessing.Condition behaves like threading.Condition, but its notify() method lacked the optional “n” argument (to specify the number of sleepers to wake up) that threading.Condition.notify() accepts.

- bpo-30532 [https://bugs.python.org/issue?@action=redirect&bpo=30532]: Fix email header value parser dropping folding white space in certain cases.
- bpo-30596 [https://bugs.python.org/issue?@action=redirect&bpo=30596]: Add a `close()` method to `multiprocessing.Process`.
- bpo-9146 [https://bugs.python.org/issue?@action=redirect&bpo=9146]: Fix a segmentation fault in _hashopenssl when standard hash functions such as md5 are not available in the linked OpenSSL library. As in some special FIPS-140 build environments.
- bpo-29169 [https://bugs.python.org/issue?@action=redirect&bpo=29169]: Update zlib to 1.2.11.
- bpo-30119 [https://bugs.python.org/issue?@action=redirect&bpo=30119]: ftplib.FTP.putline() now throws ValueError on commands that contains CR or LF. Patch by Donghee Na.
- bpo-30879 [https://bugs.python.org/issue?@action=redirect&bpo=30879]: os.listdir() and os.scandir() now emit bytes names when called with bytes-like argument.
- bpo-30746 [https://bugs.python.org/issue?@action=redirect&bpo=30746]: Prohibited the ‘=’ character in environment variable names in `os.putenv()` and `os.spawn*()`.
- bpo-30664 [https://bugs.python.org/issue?@action=redirect&bpo=30664]: The description of a unittest subtest now preserves the order of keyword arguments of TestCase.subTest().
- bpo-21071 [https://bugs.python.org/issue?@action=redirect&bpo=21071]: struct.Struct.format type is now **`str`** instead of **`bytes`**.
- bpo-29212 [https://bugs.python.org/issue?@action=redirect&bpo=29212]: Fix concurrent.futures.thread.ThreadPoolExecutor threads to have a non repr() based thread name by default

when no thread_name_prefix is supplied. They will now identify themselves as “ThreadPoolExecutor-y_n”.

- bpo-29755 [https://bugs.python.org/issue?@action=redirect&bpo=29755]: Fixed the lgettext() family of functions in the gettext module. They now always return bytes.
- bpo-30616 [https://bugs.python.org/issue?@action=redirect&bpo=30616]: Functional API of enum allows to create empty enums. Patched by Donghee Na
- bpo-30038 [https://bugs.python.org/issue?@action=redirect&bpo=30038]: Fix race condition between signal delivery and wakeup file descriptor. Patch by Nathaniel Smith.
- bpo-23894 [https://bugs.python.org/issue?@action=redirect&bpo=23894]: lib2to3 now recognizes `rb'...'` and `f'...'` strings.
- bpo-24744 [https://bugs.python.org/issue?@action=redirect&bpo=24744]: pkgutil.walk_packages function now raises ValueError if *path* is a string. Patch by Sanyam Khurana.
- bpo-24484 [https://bugs.python.org/issue?@action=redirect&bpo=24484]: Avoid race condition in multiprocessing cleanup.
- bpo-30589 [https://bugs.python.org/issue?@action=redirect&bpo=30589]: Fix multiprocessing.Process.exitcode to return the opposite of the signal number when the process is killed by a signal (instead of 255) when using the “forkserver” method.
- bpo-28994 [https://bugs.python.org/issue?@action=redirect&bpo=28994]: The traceback no longer displayed for SystemExit raised in a callback registered by atexit.
- bpo-30508 [https://bugs.python.org/issue?@action=redirect&bpo=30508]: Don’t log exceptions if Task/Future “cancel()” method was called.

- bpo-30645 [https://bugs.python.org/issue?@action=redirect&bpo=30645]: Fix path calculation in `imp.load_package()`, fixing it for cases when a package is only shipped with bytecodes. Patch by Alexandru Ardelean.
- bpo-11822 [https://bugs.python.org/issue?@action=redirect&bpo=11822]: The dis.dis() function now is able to disassemble nested code objects.
- bpo-30624 [https://bugs.python.org/issue?@action=redirect&bpo=30624]: selectors does not take KeyboardInterrupt and SystemExit into account, leaving a fd in a bad state in case of error. Patch by Giampaolo Rodola'.
- bpo-30595 [https://bugs.python.org/issue?@action=redirect&bpo=30595]: multiprocessing.Queue.get() with a timeout now polls its reader in non-blocking mode if it succeeded to acquire the lock but the acquire took longer than the timeout.
- bpo-28556 [https://bugs.python.org/issue?@action=redirect&bpo=28556]: Updates to typing module: Add generic AsyncContextManager, add support for ContextManager on all versions. Original PRs by Jelle Zijlstra and Ivan Levkivskyi
- bpo-30605 [https://bugs.python.org/issue?@action=redirect&bpo=30605]: re.compile() no longer raises a BytesWarning when compiling a bytes instance with misplaced inline modifier. Patch by Roy Williams.
- bpo-29870 [https://bugs.python.org/issue?@action=redirect&bpo=29870]: Fix ssl sockets leaks when connection is aborted in asyncio/ssl implementation. Patch by Michaël Sghaïer.
- bpo-29743 [https://bugs.python.org/issue?@action=redirect&bpo=29743]: Closing transport during handshake process leaks open socket. Patch by Nikolay Kim

- bpo-27585 [https://bugs.python.org/issue?@action=redirect&bpo=27585]: Fix waiter cancellation in asyncio.Lock. Patch by Mathieu Sornay.
- bpo-30014 [https://bugs.python.org/issue?@action=redirect&bpo=30014]: modify() method of poll(), epoll() and devpoll() based classes of selectors module is around 10% faster. Patch by Giampaolo Rodola’.
- bpo-30418 [https://bugs.python.org/issue?@action=redirect&bpo=30418]: On Windows, subprocess.Popen.communicate() now also ignore EINVAL on stdin.write() if the child process is still running but closed the pipe.
- bpo-30463 [https://bugs.python.org/issue?@action=redirect&bpo=30463]: Added empty __slots__ to abc.ABC. This allows subclassers to deny __dict__ and __weakref__ creation. Patch by Aaron Hall.
- bpo-30520 [https://bugs.python.org/issue?@action=redirect&bpo=30520]: Loggers are now pickleable.
- bpo-30557 [https://bugs.python.org/issue?@action=redirect&bpo=30557]: faulthandler now correctly filters and displays exception codes on Windows
- bpo-30526 [https://bugs.python.org/issue?@action=redirect&bpo=30526]: Add TextIOWrapper.reconfigure() and a TextIOWrapper.write_through attribute.
- bpo-30245 [https://bugs.python.org/issue?@action=redirect&bpo=30245]: Fix possible overflow when organize struct.pack_into error message. Patch by Yuan Liu.
- bpo-30378 [https://bugs.python.org/issue?@action=redirect&bpo=30378]: Fix the problem that logging.handlers.SysLogHandler cannot handle IPv6 addresses.
- bpo-16500 [https://bugs.python.org/issue?@action=redirect&bpo=16500]: Allow registering at-fork handlers.

- bpo-30470 [https://bugs.python.org/issue?@action=redirect&bpo=30470]: Deprecate invalid ctypes call protection on Windows. Patch by Mariatta Wijaya.
- bpo-30414 [https://bugs.python.org/issue?@action=redirect&bpo=30414]: multiprocessing.Queue._feed background running thread do not break from main loop on exception.
- bpo-30003 [https://bugs.python.org/issue?@action=redirect&bpo=30003]: Fix handling escape characters in HZ codec. Based on patch by Ma Lin.
- bpo-30149 [https://bugs.python.org/issue?@action=redirect&bpo=30149]: inspect.signature() now supports callables with variable-argument parameters wrapped with partialmethod. Patch by Donghee Na.
- bpo-30436 [https://bugs.python.org/issue?@action=redirect&bpo=30436]: importlib.find_spec() raises ModuleNotFoundError instead of AttributeError if the specified parent module is not a package (i.e. lacks a __path__ attribute).
- bpo-30301 [https://bugs.python.org/issue?@action=redirect&bpo=30301]: Fix AttributeError when using SimpleQueue.empty() under *spawn* and *forkserver* start methods.
- bpo-30375 [https://bugs.python.org/issue?@action=redirect&bpo=30375]: Warnings emitted when compile a regular expression now always point to the line in the user code. Previously they could point into inners of the re module if emitted from inside of groups or conditionals.
- bpo-30329 [https://bugs.python.org/issue?@action=redirect&bpo=30329]: imaplib and poplib now catch the Windows socket WSAEINVAL error (code 10022) on shutdown(SHUT_RDWR): An invalid operation was attempted. This error occurs sometimes on SSL connections.

- bpo-29196 [https://bugs.python.org/issue?@action=redirect&bpo=29196]: Removed previously deprecated in Python 2.4 classes Plist, Dict and _InternalDict in the plistlib module. Dict values in the result of functions readPlist() and readPlistFromBytes() are now normal dicts. You no longer can use attribute access to access items of these dictionaries.
- bpo-9850 [https://bugs.python.org/issue?@action=redirect&bpo=9850]: The **`macpath`** is now deprecated and will be removed in Python 3.8.
- bpo-30299 [https://bugs.python.org/issue?@action=redirect&bpo=30299]: Compiling regular expression in debug mode on CPython now displays the compiled bytecode in human readable form.
- bpo-30048 [https://bugs.python.org/issue?@action=redirect&bpo=30048]: Fixed `Task.cancel()` can be ignored when the task is running coroutine and the coroutine returned without any more `await`.
- bpo-30266 [https://bugs.python.org/issue?@action=redirect&bpo=30266]: contextlib.AbstractContextManager now supports anti-registration by setting __enter__ = None or __exit__ = None, following the pattern introduced in bpo-25958 [https://bugs.python.org/issue?@action=redirect&bpo=25958]. Patch by Jelle Zijlstra.
- bpo-30340 [https://bugs.python.org/issue?@action=redirect&bpo=30340]: Enhanced regular expressions optimization. This increased the performance of matching some patterns up to 25 times.
- bpo-30298 [https://bugs.python.org/issue?@action=redirect&bpo=30298]: Weaken the condition of deprecation warnings for inline modifiers. Now allowed several subsequential inline modifiers at the start of the pattern (e.g. `'(?i)(?s)...'`). In verbose mode whitespaces and comments now are allowed before and between inline modifiers (e.g. `'(?x) (?i) (?s)...'`).

- bpo-30285 [https://bugs.python.org/issue?@action=redirect&bpo=30285]: Optimized case-insensitive matching and searching of regular expressions.
- bpo-29990 [https://bugs.python.org/issue?@action=redirect&bpo=29990]: Fix range checking in GB18030 decoder. Original patch by Ma Lin.
- bpo-29979 [https://bugs.python.org/issue?@action=redirect&bpo=29979]: rewrite cgi.parse_multipart, reusing the FieldStorage class and making its results consistent with those of FieldStorage for multipart/form-data requests. Patch by Pierre Quentel.
- bpo-30243 [https://bugs.python.org/issue?@action=redirect&bpo=30243]: Removed the __init__ methods of _json's scanner and encoder. Misusing them could cause memory leaks or crashes. Now scanner and encoder objects are completely initialized in the __new__ methods.
- bpo-30215 [https://bugs.python.org/issue?@action=redirect&bpo=30215]: Compiled regular expression objects with the re.LOCALE flag no longer depend on the locale at compile time. Only the locale at matching time affects the result of matching.
- bpo-30185 [https://bugs.python.org/issue?@action=redirect&bpo=30185]: Avoid KeyboardInterrupt tracebacks in forkserver helper process when Ctrl-C is received.
- bpo-30103 [https://bugs.python.org/issue?@action=redirect&bpo=30103]: binascii.b2a_uu() and uu.encode() now support using `'`'` as zero instead of space.
- bpo-28556 [https://bugs.python.org/issue?@action=redirect&bpo=28556]: Various updates to typing module: add typing.NoReturn type, use WrapperDescriptorType, minor bug-fixes. Original PRs by Jim Fasarakis-Hilliard and Ivan Levkivskyi.

- bpo-30205 [https://bugs.python.org/issue?@action=redirect&bpo=30205]: Fix getsockname() for unbound AF_UNIX sockets on Linux.
- bpo-30228 [https://bugs.python.org/issue?@action=redirect&bpo=30228]: The seek() and tell() methods of io.FileIO now set the internal seekable attribute to avoid one syscall on open() (in buffered or text mode).
- bpo-30190 [https://bugs.python.org/issue?@action=redirect&bpo=30190]: unittest's assertAlmostEqual and assertNotAlmostEqual provide a better message in case of failure which includes the difference between left and right arguments. (patch by Giampaolo Rodola')
- bpo-30101 [https://bugs.python.org/issue?@action=redirect&bpo=30101]: Add support for curses.A_ITALIC.
- bpo-29822 [https://bugs.python.org/issue?@action=redirect&bpo=29822]: inspect.isabstract() now works during __init_subclass__. Patch by Nate Soares.
- bpo-29960 [https://bugs.python.org/issue?@action=redirect&bpo=29960]: Preserve generator state when _random.Random.setstate() raises an exception. Patch by Bryan Olson.
- bpo-30070 [https://bugs.python.org/issue?@action=redirect&bpo=30070]: Fixed leaks and crashes in errors handling in the parser module.
- bpo-22352 [https://bugs.python.org/issue?@action=redirect&bpo=22352]: Column widths in the output of dis.dis() are now adjusted for large line numbers and instruction offsets.
- bpo-30061 [https://bugs.python.org/issue?@action=redirect&bpo=30061]: Fixed crashes in IOBase methods __next__() and readlines() when readline() or __next__() respectively return non-sizeable object. Fixed possible other errors caused by not checking results of PyObject_Size(), PySequence_Size(), or PyMapping_Size().

- bpo-30218 [https://bugs.python.org/issue?@action=redirect&bpo=30218]: Fix PathLike support for shutil.unpack_archive. Patch by Jelle Zijlstra.
- bpo-10076 [https://bugs.python.org/issue?@action=redirect&bpo=10076]: Compiled regular expression and match objects in the re module now support copy.copy() and copy.deepcopy() (they are considered atomic).
- bpo-30068 [https://bugs.python.org/issue?@action=redirect&bpo=30068]: _io._IOBase.readlines will check if it's closed first when hint is present.
- bpo-29694 [https://bugs.python.org/issue?@action=redirect&bpo=29694]: Fixed race condition in pathlib mkdir with flags parents=True. Patch by Armin Rigo.
- bpo-29692 [https://bugs.python.org/issue?@action=redirect&bpo=29692]: Fixed arbitrary unchaining of RuntimeError exceptions in contextlib.contextmanager. Patch by Siddharth Velankar.
- bpo-26187 [https://bugs.python.org/issue?@action=redirect&bpo=26187]: Test that sqlite3 trace callback is not called multiple times when schema is changing. Indirectly fixed by switching to use sqlite3_prepare_v2() in bpo-9303 [https://bugs.python.org/issue?@action=redirect&bpo=9303]. Patch by Aviv Palivoda.
- bpo-30017 [https://bugs.python.org/issue?@action=redirect&bpo=30017]: Allowed calling the close() method of the zip entry writer object multiple times. Writing to a closed writer now always produces a ValueError.
- bpo-29998 [https://bugs.python.org/issue?@action=redirect&bpo=29998]: Pickling and copying ImportError now preserves name and path attributes.
- bpo-29995 [https://bugs.python.org/issue?@action=redirect&bpo=29995]: re.escape() now escapes only regex special characters.

- bpo-29962 [https://bugs.python.org/issue?@action=redirect&bpo=29962]: Add math.remainder operation, implementing remainder as specified in IEEE 754.
- bpo-29649 [https://bugs.python.org/issue?@action=redirect&bpo=29649]: Improve struct.pack_into() exception messages for problems with the buffer size and offset. Patch by Andrew Nester.
- bpo-29654 [https://bugs.python.org/issue?@action=redirect&bpo=29654]: Support If-Modified-Since HTTP header (browser cache). Patch by Pierre Quentel.
- bpo-29931 [https://bugs.python.org/issue?@action=redirect&bpo=29931]: Fixed comparison check for ipaddress.ip_interface objects. Patch by Sanjay Sundaresan.
- bpo-29953 [https://bugs.python.org/issue?@action=redirect&bpo=29953]: Fixed memory leaks in the replace() method of datetime and time objects when pass out of bound fold argument.
- bpo-29942 [https://bugs.python.org/issue?@action=redirect&bpo=29942]: Fix a crash in itertools.chain.from_iterable when encountering long runs of empty iterables.
- bpo-10030 [https://bugs.python.org/issue?@action=redirect&bpo=10030]: Sped up reading encrypted ZIP files by 2 times.
- bpo-29204 [https://bugs.python.org/issue?@action=redirect&bpo=29204]: Element.getiterator() and the html parameter of XMLParser() were deprecated only in the documentation (since Python 3.2 and 3.4 correspondingly). Now using them emits a deprecation warning.
- bpo-27863 [https://bugs.python.org/issue?@action=redirect&bpo=27863]: Fixed multiple crashes in ElementTree caused by race conditions and wrong types.

- bpo-25996 [https://bugs.python.org/issue?@action=redirect&bpo=25996]: Added support of file descriptors in os.scandir() on Unix. os.fwalk() is sped up by 2 times by using os.scandir().
- bpo-28699 [https://bugs.python.org/issue?@action=redirect&bpo=28699]: Fixed a bug in pools in multiprocessing.pool that raising an exception at the very first of an iterable may swallow the exception or make the program hang. Patch by Davin Potts and Xiang Zhang.
- bpo-23890 [https://bugs.python.org/issue?@action=redirect&bpo=23890]: unittest.TestCase.assertRaises() now manually breaks a reference cycle to not keep objects alive longer than expected.
- bpo-29901 [https://bugs.python.org/issue?@action=redirect&bpo=29901]: The zipapp module now supports general path-like objects, not just pathlib.Path.
- bpo-25803 [https://bugs.python.org/issue?@action=redirect&bpo=25803]: Avoid incorrect errors raised by Path.mkdir(exist_ok=True) when the OS gives priority to errors such as EACCES over EEXIST.
- bpo-29861 [https://bugs.python.org/issue?@action=redirect&bpo=29861]: Release references to tasks, their arguments and their results as soon as they are finished in multiprocessing.Pool.
- bpo-19930 [https://bugs.python.org/issue?@action=redirect&bpo=19930]: The mode argument of os.makedirs() no longer affects the file permission bits of newly created intermediate-level directories.
- bpo-29884 [https://bugs.python.org/issue?@action=redirect&bpo=29884]: faulthandler: Restore the old sigaltstack during teardown. Patch by Christophe Zeitouny.
- bpo-25455 [https://bugs.python.org/issue?@action=redirect&bpo=25455]: Fixed crashes in repr of recursive buffered file-like objects.

- bpo-29800 [https://bugs.python.org/issue?@action=redirect&bpo=29800]: Fix crashes in partial.__repr__ if the keys of partial.keywords are not strings. Patch by Michael Seifert.
- bpo-8256 [https://bugs.python.org/issue?@action=redirect&bpo=8256]: Fixed possible failing or crashing input() if attributes “encoding” or “errors” of sys.stdin or sys.stdout are not set or are not strings.
- bpo-28692 [https://bugs.python.org/issue?@action=redirect&bpo=28692]: Using non-integer value for selecting a plural form in gettext is now deprecated.
- bpo-26121 [https://bugs.python.org/issue?@action=redirect&bpo=26121]: Use C library implementation for math functions erf() and erfc().
- bpo-29619 [https://bugs.python.org/issue?@action=redirect&bpo=29619]: os.stat() and os.DirEntry.inode() now convert inode (st_ino) using unsigned integers.
- bpo-28298 [https://bugs.python.org/issue?@action=redirect&bpo=28298]: Fix a bug that prevented array ‘Q’, ‘L’ and ‘I’ from accepting big intables (objects that have __int__) as elements.
- bpo-29645 [https://bugs.python.org/issue?@action=redirect&bpo=29645]: Speed up importing the webbrowser module. webbrowser.register() is now thread-safe.
- bpo-28231 [https://bugs.python.org/issue?@action=redirect&bpo=28231]: The zipfile module now accepts path-like objects for external paths.
- bpo-26915 [https://bugs.python.org/issue?@action=redirect&bpo=26915]: index() and count() methods of collections.abc.Sequence now check identity before checking equality when do comparisons.
- bpo-28682 [https://bugs.python.org/issue?@action=redirect&bpo=28682]: Added support for bytes paths in os.fwalk().

- bpo-29728 [https://bugs.python.org/issue?@action=redirect&bpo=29728]: Add new **`socket.TCP_NOTSENT_LOWAT`** (Linux 3.12) constant. Patch by Nathaniel J. Smith.
- bpo-29623 [https://bugs.python.org/issue?@action=redirect&bpo=29623]: Allow use of path-like object as a single argument in ConfigParser.read(). Patch by David Ellis.
- bpo-9303 [https://bugs.python.org/issue?@action=redirect&bpo=9303]: Migrate sqlite3 module to _v2 API. Patch by Aviv Palivoda.
- bpo-28963 [https://bugs.python.org/issue?@action=redirect&bpo=28963]: Fix out of bound iteration in asyncio.Future.remove_done_callback implemented in C.
- bpo-29704 [https://bugs.python.org/issue?@action=redirect&bpo=29704]: asyncio.subprocess.SubprocessStreamProtocol no longer closes before all pipes are closed.
- bpo-29271 [https://bugs.python.org/issue?@action=redirect&bpo=29271]: Fix Task.current_task and Task.all_tasks implemented in C to accept None argument as their pure Python implementation.
- bpo-29703 [https://bugs.python.org/issue?@action=redirect&bpo=29703]: Fix asyncio to support instantiation of new event loops in child processes.
- bpo-29615 [https://bugs.python.org/issue?@action=redirect&bpo=29615]: SimpleXMLRPCDispatcher no longer chains KeyError (or any other exception) to exception(s) raised in the dispatched methods. Patch by Petr Motejlek.
- bpo-7769 [https://bugs.python.org/issue?@action=redirect&bpo=7769]: Method register_function() of xmlrpc.server.SimpleXMLRPCDispatcher and its subclasses can now be used as a decorator.
- bpo-29376 [https://bugs.python.org/issue?@action=redirect&bpo=29376]: Fix assertion error in

threading._DummyThread.is_alive().

- bpo-28624 [https://bugs.python.org/issue?@action=redirect&bpo=28624]: Add a test that checks that cwd parameter of Popen() accepts PathLike objects. Patch by Sayan Chowdhury.
- bpo-28518 [https://bugs.python.org/issue?@action=redirect&bpo=28518]: Start a transaction implicitly before a DML statement. Patch by Aviv Palivoda.
- bpo-29742 [https://bugs.python.org/issue?@action=redirect&bpo=29742]: get_extra_info() raises exception if get called on closed ssl transport. Patch by Nikolay Kim.
- bpo-16285 [https://bugs.python.org/issue?@action=redirect&bpo=16285]: urllib.parse.quote is now based on RFC 3986 and hence includes '~' in the set of characters that is not quoted by default. Patch by Christian Theune and Ratnadeep Debnath.
- bpo-29532 [https://bugs.python.org/issue?@action=redirect&bpo=29532]: Altering a kwarg dictionary passed to functools.partial() no longer affects a partial object after creation.
- bpo-29110 [https://bugs.python.org/issue?@action=redirect&bpo=29110]: Fix file object leak in aifc.open() when file is given as a filesystem path and is not in valid AIFF format. Patch by Anthony Zhang.
- bpo-22807 [https://bugs.python.org/issue?@action=redirect&bpo=22807]: Add uuid.SafeUUID and uuid.UUID.is_safe to relay information from the platform about whether generated UUIDs are generated with a multiprocessing safe method.
- bpo-29576 [https://bugs.python.org/issue?@action=redirect&bpo=29576]: Improve some deprecations in importlib. Some deprecated methods now emit DeprecationWarnings and have better descriptive messages.

- bpo-29534 [https://bugs.python.org/issue?@action=redirect&bpo=29534]: Fixed different behaviour of Decimal.from_float() for _decimal and _pydecimal. Thanks Andrew Nester.
- bpo-10379 [https://bugs.python.org/issue?@action=redirect&bpo=10379]: locale.format_string now supports the ‘monetary’ keyword argument, and locale.format is deprecated.
- bpo-29851 [https://bugs.python.org/issue?@action=redirect&bpo=29851]: importlib.reload() now raises ModuleNotFoundError if the module lacks a spec.
- bpo-28556 [https://bugs.python.org/issue?@action=redirect&bpo=28556]: Various updates to typing module: typing.Counter, typing.ChainMap, improved ABC caching, etc. Original PRs by Jelle Zijlstra, Ivan Levkivskyi, Manuel Krebber, and Łukasz Langa.
- bpo-29100 [https://bugs.python.org/issue?@action=redirect&bpo=29100]: Fix datetime.fromtimestamp() regression introduced in Python 3.6.0: check minimum and maximum years.
- bpo-29416 [https://bugs.python.org/issue?@action=redirect&bpo=29416]: Prevent infinite loop in pathlib.Path.mkdir
- bpo-29444 [https://bugs.python.org/issue?@action=redirect&bpo=29444]: Fixed out-of-bounds buffer access in the group() method of the match object. Based on patch by WGH.
- bpo-29377 [https://bugs.python.org/issue?@action=redirect&bpo=29377]: Add WrapperDescriptorType, MethodWrapperType, and MethodDescriptorType built-in types to types module. Original patch by Manuel Krebber.
- bpo-29218 [https://bugs.python.org/issue?@action=redirect&bpo=29218]: Unused install_misc command is now removed. It has been documented as unused since 2000. Patch by Eric N. Vander Weele.

- bpo-29368 [https://bugs.python.org/issue?@action=redirect&bpo=29368]: The extend() method is now called instead of the append() method when unpickle collections.deque and other list-like objects. This can speed up unpickling to 2 times.
- bpo-29338 [https://bugs.python.org/issue?@action=redirect&bpo=29338]: The help of a builtin or extension class now includes the constructor signature if __text_signature__ is provided for the class.
- bpo-29335 [https://bugs.python.org/issue?@action=redirect&bpo=29335]: Fix subprocess.Popen.wait() when the child process has exited to a stopped instead of terminated state (ex: when under ptrace).
- bpo-29290 [https://bugs.python.org/issue?@action=redirect&bpo=29290]: Fix a regression in argparse that help messages would wrap at non-breaking spaces.
- bpo-28735 [https://bugs.python.org/issue?@action=redirect&bpo=28735]: Fixed the comparison of mock.MagickMock with mock.ANY.
- bpo-29197 [https://bugs.python.org/issue?@action=redirect&bpo=29197]: Removed deprecated function ntpath.splitunc().
- bpo-29210 [https://bugs.python.org/issue?@action=redirect&bpo=29210]: Removed support of deprecated argument “exclude” in tarfile.TarFile.add().
- bpo-29219 [https://bugs.python.org/issue?@action=redirect&bpo=29219]: Fixed infinite recursion in the repr of uninitialized ctypes.CDLL instances.
- bpo-29192 [https://bugs.python.org/issue?@action=redirect&bpo=29192]: Removed deprecated features in the http.cookies module.
- bpo-29193 [https://bugs.python.org/issue?@action=redirect&bpo=29193]: A format string argument for string.Formatter.format() is now positional-only.
- bpo-29195 [https://bugs.python.org/issue?@action=redirect&bpo=29195]: Removed support of

deprecated undocumented keyword arguments in methods of regular expression objects.

- bpo-28969 [https://bugs.python.org/issue?@action=redirect&bpo=28969]: Fixed race condition in C implementation of functools.lru_cache. KeyError could be raised when cached function with full cache was simultaneously called from different threads with the same uncached arguments.
- bpo-20804 [https://bugs.python.org/issue?@action=redirect&bpo=20804]: The unittest.mock.sentinel attributes now preserve their identity when they are copied or pickled.
- bpo-29142 [https://bugs.python.org/issue?@action=redirect&bpo=29142]: In urllib.request, suffixes in no_proxy environment variable with leading dots could match related hostnames again (e.g. .b.c matches a.b.c). Patch by Milan Oberkirch.
- bpo-28961 [https://bugs.python.org/issue?@action=redirect&bpo=28961]: Fix unittest.mock._Call helper: don’t ignore the name parameter anymore. Patch written by Jiajun Huang.
- bpo-15812 [https://bugs.python.org/issue?@action=redirect&bpo=15812]: inspect.getframeinfo() now correctly shows the first line of a context. Patch by Sam Breese.
- bpo-28985 [https://bugs.python.org/issue?@action=redirect&bpo=28985]: Update authorizer constants in sqlite3 module. Patch by Dingyuan Wang.
- bpo-29079 [https://bugs.python.org/issue?@action=redirect&bpo=29079]: Prevent infinite loop in pathlib.resolve() on Windows
- bpo-13051 [https://bugs.python.org/issue?@action=redirect&bpo=13051]: Fixed recursion errors in large or resized curses.textpad.Textbox. Based on patch by Tycho Andersen.

- bpo-9770 [https://bugs.python.org/issue?@action=redirect&bpo=9770]: curses.ascii predicates now work correctly with negative integers.
- bpo-28427 [https://bugs.python.org/issue?@action=redirect&bpo=28427]: old keys should not remove new values from WeakValueDictionary when collecting from another thread.
- bpo-28923 [https://bugs.python.org/issue?@action=redirect&bpo=28923]: Remove editor artifacts from Tix.py.
- bpo-28871 [https://bugs.python.org/issue?@action=redirect&bpo=28871]: Fixed a crash when deallocate deep ElementTree.
- bpo-19542 [https://bugs.python.org/issue?@action=redirect&bpo=19542]: Fix bugs in WeakValueDictionary.setdefault() and WeakValueDictionary.pop() when a GC collection happens in another thread.
- bpo-20191 [https://bugs.python.org/issue?@action=redirect&bpo=20191]: Fixed a crash in resource.prlimit() when passing a sequence that doesn’t own its elements as limits.
- bpo-16255 [https://bugs.python.org/issue?@action=redirect&bpo=16255]: subprocess.Popen uses /system/bin/sh on Android as the shell, instead of /bin/sh.
- bpo-28779 [https://bugs.python.org/issue?@action=redirect&bpo=28779]: multiprocessing.set_forkserver_preload() would crash the forkserver process if a preloaded module instantiated some multiprocessing objects such as locks.
- bpo-26937 [https://bugs.python.org/issue?@action=redirect&bpo=26937]: The chown() method of the tarfile.TarFile class does not fail now when the grp module cannot be imported, as for example on Android platforms.

- bpo-28847 [https://bugs.python.org/issue?@action=redirect&bpo=28847]: dbm.dumb now supports reading read-only files and no longer writes the index file when it is not changed. A deprecation warning is now emitted if the index file is missed and recreated in the ‘r’ and ‘w’ modes (will be an error in future Python releases).
- bpo-27030 [https://bugs.python.org/issue?@action=redirect&bpo=27030]: Unknown escapes consisting of `'\'` and an ASCII letter in re.sub() replacement templates regular expressions now are errors.
- bpo-28835 [https://bugs.python.org/issue?@action=redirect&bpo=28835]: Fix a regression introduced in warnings.catch_warnings(): call warnings.showwarning() if it was overridden inside the context manager.
- bpo-27172 [https://bugs.python.org/issue?@action=redirect&bpo=27172]: To assist with upgrades from 2.7, the previously documented deprecation of `inspect.getfullargspec()` has been reversed. This decision may be revisited again after the Python 2.7 branch is no longer officially supported.
- bpo-28740 [https://bugs.python.org/issue?@action=redirect&bpo=28740]: Add sys.getandroidapilevel(): return the build time API version of Android as an integer. Function only available on Android.
- bpo-26273 [https://bugs.python.org/issue?@action=redirect&bpo=26273]: Add new **`socket.TCP_CONGESTION`** (Linux 2.6.13) and **`socket.TCP_USER_TIMEOUT`** (Linux 2.6.37) constants. Patch written by Omar Sandoval.
- bpo-28752 [https://bugs.python.org/issue?@action=redirect&bpo=28752]: Restored the __reduce__() methods of datetime objects.
- bpo-28727 [https://bugs.python.org/issue?@action=redirect&bpo=28727]: Regular expression patterns, _sre.SRE_Pattern objects created by re.compile(), become comparable (only x==y and x!=y operators).

This change should fix the bpo-18383 [https://bugs.python.org/issue?@action=redirect&bpo=18383]: don’t duplicate warning filters when the warnings module is reloaded (thing usually only done in unit tests).

- bpo-20572 [https://bugs.python.org/issue?@action=redirect&bpo=20572]: Remove the subprocess.Popen.wait endtime parameter. It was deprecated in 3.4 and undocumented prior to that.
- bpo-25659 [https://bugs.python.org/issue?@action=redirect&bpo=25659]: In ctypes, prevent a crash calling the from_buffer() and from_buffer_copy() methods on abstract classes like Array.
- bpo-28548 [https://bugs.python.org/issue?@action=redirect&bpo=28548]: In the “http.server” module, parse the protocol version if possible, to avoid using HTTP 0.9 in some error responses.
- bpo-19717 [https://bugs.python.org/issue?@action=redirect&bpo=19717]: Makes Path.resolve() succeed on paths that do not exist. Patch by Vajrasky Kok
- bpo-28563 [https://bugs.python.org/issue?@action=redirect&bpo=28563]: Fixed possible DoS and arbitrary code execution when handle plural form selections in the gettext module. The expression parser now supports exact syntax supported by GNU gettext.
- bpo-28387 [https://bugs.python.org/issue?@action=redirect&bpo=28387]: Fixed possible crash in _io.TextIOWrapper deallocator when the garbage collector is invoked in other thread. Based on patch by Sebastian Cufre.
- bpo-27517 [https://bugs.python.org/issue?@action=redirect&bpo=27517]: LZMA compressor and decompressor no longer raise exceptions if given empty data twice. Patch by Benjamin Fogle.
- bpo-28549 [https://bugs.python.org/issue?@action=redirect&bpo=28549]: Fixed segfault in curses’s addch() with ncurses6.

- bpo-28449 [https://bugs.python.org/issue?@action=redirect&bpo=28449]: tarfile.open() with mode “r” or “r:” now tries to open a tar file with compression before trying to open it without compression. Otherwise it had 50% chance failed with ignore_zeros=True.
- bpo-23262 [https://bugs.python.org/issue?@action=redirect&bpo=23262]: The webbrowser module now supports Firefox 36+ and derived browsers. Based on patch by Oleg Broytman.
- bpo-24241 [https://bugs.python.org/issue?@action=redirect&bpo=24241]: The webbrowser in an X environment now prefers using the default browser directly. Also, the webbrowser register() function now has a documented ‘preferred’ argument, to specify browsers to be returned by get() with no arguments. Patch by David Steele
- bpo-27939 [https://bugs.python.org/issue?@action=redirect&bpo=27939]: Fixed bugs in tkinter.ttk.LabeledScale and tkinter.Scale caused by representing the scale as float value internally in Tk. tkinter.IntVar now works if float value is set to underlying Tk variable.
- bpo-28255 [https://bugs.python.org/issue?@action=redirect&bpo=28255]: calendar.TextCalendar.prweek() no longer prints a space after a weeks’s calendar. calendar.TextCalendar.pryear() no longer prints redundant newline after a year’s calendar. Based on patch by Xiang Zhang.
- bpo-28255 [https://bugs.python.org/issue?@action=redirect&bpo=28255]: calendar.TextCalendar.prmonth() no longer prints a space at the start of new line after printing a month’s calendar. Patch by Xiang Zhang.
- bpo-20491 [https://bugs.python.org/issue?@action=redirect&bpo=20491]: The textwrap.TextWrapper

class now honors non-breaking spaces. Based on patch by Kaarle Ritvanen.

- bpo-28353 [https://bugs.python.org/issue?@action=redirect&bpo=28353]: os.fwalk() no longer fails on broken links.
- bpo-28430 [https://bugs.python.org/issue?@action=redirect&bpo=28430]: Fix iterator of C implemented asyncio.Future doesn’t accept non-None value is passed to it.send(val).
- bpo-27025 [https://bugs.python.org/issue?@action=redirect&bpo=27025]: Generated names for Tkinter widgets now start by the “!” prefix for readability.
- bpo-25464 [https://bugs.python.org/issue?@action=redirect&bpo=25464]: Fixed HList.header_exists() in tkinter.tix module by addin a workaround to Tix library bug.
- bpo-28488 [https://bugs.python.org/issue?@action=redirect&bpo=28488]: shutil.make_archive() no longer adds entry “./” to ZIP archive.
- bpo-25953 [https://bugs.python.org/issue?@action=redirect&bpo=25953]: re.sub() now raises an error for invalid numerical group reference in replacement template even if the pattern is not found in the string. Error message for invalid group reference now includes the group index and the position of the reference. Based on patch by SilentGhost.
- bpo-28469 [https://bugs.python.org/issue?@action=redirect&bpo=28469]: timeit now uses the sequence 1, 2, 5, 10, 20, 50,… instead of 1, 10, 100,… for autoranging.
- bpo-28115 [https://bugs.python.org/issue?@action=redirect&bpo=28115]: Command-line interface of the zipfile module now uses argparse. Added support of long options.
- bpo-18219 [https://bugs.python.org/issue?@action=redirect&bpo=18219]: Optimize csv.DictWriter for

large number of columns. Patch by Mariatta Wijaya.

- bpo-28448 [https://bugs.python.org/issue?@action=redirect&bpo=28448]: Fix C implemented asyncio.Future didn’t work on Windows.
- bpo-23214 [https://bugs.python.org/issue?@action=redirect&bpo=23214]: In the “io” module, the argument to BufferedReader and BytesIO’s read1() methods is now optional and can be -1, matching the BufferedIOBase specification.
- bpo-28480 [https://bugs.python.org/issue?@action=redirect&bpo=28480]: Fix error building socket module when multithreading is disabled.
- bpo-28240 [https://bugs.python.org/issue?@action=redirect&bpo=28240]: timeit: remove `-c/--clock` and `-t/--time` command line options which were deprecated since Python 3.3.
- bpo-28240 [https://bugs.python.org/issue?@action=redirect&bpo=28240]: timeit now repeats the benchmarks 5 times instead of only 3 to make benchmarks more reliable.
- bpo-28240 [https://bugs.python.org/issue?@action=redirect&bpo=28240]: timeit autorange now uses a single loop iteration if the benchmark takes less than 10 seconds, instead of 10 iterations. “python3 -m timeit -s ‘import time’ ‘time.sleep(1)’” now takes 4 seconds instead of 40 seconds.
- Distutils.sdist now looks for README and setup.py files with case sensitivity. This behavior matches that found in Setuptools 6.0 and later. See setuptools 100 [https://github.com/pypa/setuptools/issues/100] for rationale.
- bpo-24452 [https://bugs.python.org/issue?@action=redirect&bpo=24452]: Make webbrowser support Chrome on Mac OS X. Patch by Ned Batchelder.
- bpo-20766 [https://bugs.python.org/issue?@action=redirect&bpo=20766]: Fix references leaked by pdb in the handling of SIGINT handlers.

- bpo-27998 [https://bugs.python.org/issue?@action=redirect&bpo=27998]: Fixed bytes path support in os.scandir() on Windows. Patch by Eryk Sun.
- bpo-28317 [https://bugs.python.org/issue?@action=redirect&bpo=28317]: The disassembler now decodes FORMAT_VALUE argument.
- bpo-28380 [https://bugs.python.org/issue?@action=redirect&bpo=28380]: unittest.mock Mock autospec functions now properly support assert_called, assert_not_called, and assert_called_once.
- bpo-28229 [https://bugs.python.org/issue?@action=redirect&bpo=28229]: lzma module now supports pathlib.
- bpo-28321 [https://bugs.python.org/issue?@action=redirect&bpo=28321]: Fixed writing non-BMP characters with binary format in plistlib.
- bpo-28225 [https://bugs.python.org/issue?@action=redirect&bpo=28225]: bz2 module now supports pathlib. Initial patch by Ethan Furman.
- bpo-28227 [https://bugs.python.org/issue?@action=redirect&bpo=28227]: gzip now supports pathlib. Patch by Ethan Furman.
- bpo-28332 [https://bugs.python.org/issue?@action=redirect&bpo=28332]: Deprecated silent truncations in socket.htons and socket.ntohs. Original patch by Oren Milman.
- bpo-27358 [https://bugs.python.org/issue?@action=redirect&bpo=27358]: Optimized merging var-keyword arguments and improved error message when passing a non-mapping as a var-keyword argument.
- bpo-28257 [https://bugs.python.org/issue?@action=redirect&bpo=28257]: Improved error message when passing a non-iterable as a var-positional argument. Added opcode BUILD_TUPLE_UNPACK_WITH_CALL.
- bpo-28322 [https://bugs.python.org/issue?@action=redirect&bpo=28322]: Fixed possible crashes when

unpickle itertools objects from incorrect pickle data. Based on patch by John Leitch.

- bpo-28228 [https://bugs.python.org/issue?@action=redirect&bpo=28228]: imghdr now supports pathlib.
- bpo-28226 [https://bugs.python.org/issue?@action=redirect&bpo=28226]: compileall now supports pathlib.
- bpo-28314 [https://bugs.python.org/issue?@action=redirect&bpo=28314]: Fix function declaration (C flags) for the getiterator() method of xml.etree.ElementTree.Element.
- bpo-28148 [https://bugs.python.org/issue?@action=redirect&bpo=28148]: Stop using localtime() and gmtime() in the time module. Introduced platform independent _PyTime_localtime API that is similar to POSIX localtime_r, but available on all platforms. Patch by Ed Schouten.
- bpo-28253 [https://bugs.python.org/issue?@action=redirect&bpo=28253]: Fixed calendar functions for extreme months: 0001-01 and 9999-12. Methods itermonthdays() and itermonthdays2() are reimplemented so that they don’t call itermonthdates() which can cause datetime.date under/overflow.
- bpo-28275 [https://bugs.python.org/issue?@action=redirect&bpo=28275]: Fixed possible use after free in the decompress() methods of the LZMADecompressor and BZ2Decompressor classes. Original patch by John Leitch.
- bpo-27897 [https://bugs.python.org/issue?@action=redirect&bpo=27897]: Fixed possible crash in sqlite3.Connection.create_collation() if pass invalid string-like object as a name. Patch by Xiang Zhang.
- bpo-18844 [https://bugs.python.org/issue?@action=redirect&bpo=18844]: random.choices() now has k as a keyword-only argument to improve the readability of

common cases and come into line with the signature used in other languages.

- bpo-18893 [https://bugs.python.org/issue?@action=redirect&bpo=18893]: Fix invalid exception handling in Lib/ctypes/macholib/dyld.py. Patch by Madison May.
- bpo-27611 [https://bugs.python.org/issue?@action=redirect&bpo=27611]: Fixed support of default root window in the tkinter.tix module. Added the master parameter in the DisplayStyle constructor.
- bpo-27348 [https://bugs.python.org/issue?@action=redirect&bpo=27348]: In the traceback module, restore the formatting of exception messages like “Exception: None”. This fixes a regression introduced in 3.5a2.
- bpo-25651 [https://bugs.python.org/issue?@action=redirect&bpo=25651]: Allow false values to be used for msg parameter of subTest().
- bpo-27778 [https://bugs.python.org/issue?@action=redirect&bpo=27778]: Fix a memory leak in os.getrandom() when the getrandom() is interrupted by a signal and a signal handler raises a Python exception.
- bpo-28200 [https://bugs.python.org/issue?@action=redirect&bpo=28200]: Fix memory leak on Windows in the os module (fix path_converter() function).
- bpo-25400 [https://bugs.python.org/issue?@action=redirect&bpo=25400]: RobotFileParser now correctly returns default values for crawl_delay and request_rate. Initial patch by Peter Wirtz.
- bpo-27932 [https://bugs.python.org/issue?@action=redirect&bpo=27932]: Prevent memory leak in win32_ver().
- Fix UnboundLocalError in socket._sendfile_use_sendfile.
- bpo-28075 [https://bugs.python.org/issue?@action=redirect&bpo=28075]: Check for ERROR_ACCESS_DENIED in Windows implementation of os.stat(). Patch by Eryk Sun.

- bpo-22493 [https://bugs.python.org/issue?@action=redirect&bpo=22493]: Warning message emitted by using inline flags in the middle of regular expression now contains a (truncated) regex pattern. Patch by Tim Graham.
- bpo-25270 [https://bugs.python.org/issue?@action=redirect&bpo=25270]: Prevent codecs.escape_encode() from raising SystemError when an empty bytestring is passed.
- bpo-28181 [https://bugs.python.org/issue?@action=redirect&bpo=28181]: Get antigravity over HTTPS. Patch by Kaartic Sivaraam.
- bpo-25895 [https://bugs.python.org/issue?@action=redirect&bpo=25895]: Enable WebSocket URL schemes in urllib.parse.urljoin. Patch by Gergely Imreh and Markus Holtermann.
- bpo-28114 [https://bugs.python.org/issue?@action=redirect&bpo=28114]: Fix a crash in parse_envlist() when env contains byte strings. Patch by Eryk Sun.
- bpo-27599 [https://bugs.python.org/issue?@action=redirect&bpo=27599]: Fixed buffer overrun in binascii.b2a_qp() and binascii.a2b_qp().
- bpo-27906 [https://bugs.python.org/issue?@action=redirect&bpo=27906]: Fix socket accept exhaustion during high TCP traffic. Patch by Kevin Conway.
- bpo-28174 [https://bugs.python.org/issue?@action=redirect&bpo=28174]: Handle when SO_REUSEPORT isn’t properly supported. Patch by Seth Michael Larson.
- bpo-26654 [https://bugs.python.org/issue?@action=redirect&bpo=26654]: Inspect functools.partial in asyncio.Handle.__repr__. Patch by iceboy.
- bpo-26909 [https://bugs.python.org/issue?@action=redirect&bpo=26909]: Fix slow pipes IO in asyncio. Patch by INADA Naoki.
- bpo-28176 [https://bugs.python.org/issue?@action=redirect&bpo=28176]: Fix callbacks race in

asyncio.SelectorLoop.sock_connect.
- bpo-27759 [https://bugs.python.org/issue?@action=redirect&bpo=27759]: Fix selectors incorrectly retain invalid file descriptors. Patch by Mark Williams.
- bpo-28325 [https://bugs.python.org/issue?@action=redirect&bpo=28325]: Remove vestigial MacOS 9 macurl2path module and its tests.
- bpo-28368 [https://bugs.python.org/issue?@action=redirect&bpo=28368]: Refuse monitoring processes if the child watcher has no loop attached. Patch by Vincent Michel.
- bpo-28369 [https://bugs.python.org/issue?@action=redirect&bpo=28369]: Raise RuntimeError when transport's FD is used with add_reader, add_writer, etc.
- bpo-28370 [https://bugs.python.org/issue?@action=redirect&bpo=28370]: Speedup asyncio.StreamReader.readexactly. Patch by Коренберг Марк.
- bpo-28371 [https://bugs.python.org/issue?@action=redirect&bpo=28371]: Deprecate passing asyncio.Handles to run_in_executor.
- bpo-28372 [https://bugs.python.org/issue?@action=redirect&bpo=28372]: Fix asyncio to support formatting of non-python coroutines.
- bpo-28399 [https://bugs.python.org/issue?@action=redirect&bpo=28399]: Remove UNIX socket from FS before binding. Patch by Коренберг Марк.
- bpo-27972 [https://bugs.python.org/issue?@action=redirect&bpo=27972]: Prohibit Tasks to await on themselves.
- bpo-24142 [https://bugs.python.org/issue?@action=redirect&bpo=24142]: Reading a corrupt config file left configparser in an invalid state. Original patch by Florian Höch.
- bpo-29581 [https://bugs.python.org/issue?@action=redirect&bpo=29581]: ABCMeta.__new__ now accepts

`**kwargs`, allowing abstract base classes to use keyword parameters in __init_subclass__. Patch by Nate Soares.
- bpo-25532 [https://bugs.python.org/issue?@action=redirect&bpo=25532]: inspect.unwrap() will now only try to unwrap an object sys.getrecursionlimit() times, to protect against objects which create a new object on every attribute access.
- bpo-30177 [https://bugs.python.org/issue?@action=redirect&bpo=30177]: path.resolve(strict=False) no longer cuts the path after the first element not present in the filesystem. Patch by Antoine Pietri.

Documentation

- bpo-31294 [https://bugs.python.org/issue?@action=redirect&bpo=31294]: Fix incomplete code snippet in the ZeroMQSocketListener and ZeroMQSocketHandler examples and adapt them to Python 3.
- bpo-21649 [https://bugs.python.org/issue?@action=redirect&bpo=21649]: Add RFC 7525 and Mozilla server side TLS links to SSL documentation.
- bpo-31128 [https://bugs.python.org/issue?@action=redirect&bpo=31128]: Allow the pydoc server to bind to arbitrary hostnames.
- bpo-30803 [https://bugs.python.org/issue?@action=redirect&bpo=30803]: Clarify doc on truth value testing. Original patch by Peter Thomassen.
- bpo-30176 [https://bugs.python.org/issue?@action=redirect&bpo=30176]: Add missing attribute related constants in curses documentation.
- bpo-30052 [https://bugs.python.org/issue?@action=redirect&bpo=30052]: the link targets for **`bytes()`** and **`bytearray()`** are now their respective type definitions, rather than the corresponding builtin function entries. Use bytes and bytearray to reference the latter. In order to ensure this and future cross-reference updates are

applied automatically, the daily documentation builds now disable the default output caching features in Sphinx.

- bpo-26985 [https://bugs.python.org/issue?@action=redirect&bpo=26985]: Add missing info of code object in inspect documentation.
- bpo-19824 [https://bugs.python.org/issue?@action=redirect&bpo=19824]: Improve the documentation for, and links to, template strings by emphasizing their utility for internationalization, and by clarifying some usage constraints. (See also: bpo-20314 [https://bugs.python.org/issue?@action=redirect&bpo=20314], bpo-12518 [https://bugs.python.org/issue?@action=redirect&bpo=12518])
- bpo-28929 [https://bugs.python.org/issue?@action=redirect&bpo=28929]: Link the documentation to its source file on GitHub.
- bpo-25008 [https://bugs.python.org/issue?@action=redirect&bpo=25008]: Document smtpd.py as effectively deprecated and add a pointer to aiosmtpd, a third-party asyncio-based replacement.
- bpo-26355 [https://bugs.python.org/issue?@action=redirect&bpo=26355]: Add canonical header link on each page to corresponding major version of the documentation. Patch by Matthias Bussonnier.
- bpo-29349 [https://bugs.python.org/issue?@action=redirect&bpo=29349]: Fix Python 2 syntax in code for building the documentation.
- bpo-23722 [https://bugs.python.org/issue?@action=redirect&bpo=23722]: The data model reference and the porting section in the 3.6 What’s New guide now cover the additional `__classcell__` handling needed for custom metaclasses to fully support **PEP 487** [https://peps.python.org/pep-0487/] and zero-argument `super()`.
- bpo-28513 [https://bugs.python.org/issue?@action=redirect&bpo=28513]: Documented command-line interface of zipfile.

Tests

- bpo-29639 [https://bugs.python.org/issue?@action=redirect&bpo=29639]: test.support.HOST is now “localhost”, a new HOSTv4 constant has been added for your `127.0.0.1` needs, similar to the existing HOSTv6 constant.
- bpo-31320 [https://bugs.python.org/issue?@action=redirect&bpo=31320]: Silence traceback in test_ssl
- bpo-31346 [https://bugs.python.org/issue?@action=redirect&bpo=31346]: Prefer PROTOCOL_TLS_CLIENT and PROTOCOL_TLS_SERVER protocols for SSLContext.
- bpo-25674 [https://bugs.python.org/issue?@action=redirect&bpo=25674]: Remove sha256.tbs-internet.com ssl test
- bpo-30715 [https://bugs.python.org/issue?@action=redirect&bpo=30715]: Address ALPN callback changes for OpenSSL 1.1.0f. The latest version behaves like OpenSSL 1.0.2 and no longer aborts handshake.
- bpo-30822 [https://bugs.python.org/issue?@action=redirect&bpo=30822]: regrtest: Exclude tzdata from regrtest –all. When running the test suite using –use=all / -u all, exclude tzdata since it makes test_datetime too slow (15-20 min on some buildbots) which then times out on some buildbots. Fix also regrtest command line parser to allow passing -u extralargefile to run test_zipfile64.
- bpo-30695 [https://bugs.python.org/issue?@action=redirect&bpo=30695]: Add the `set_nomemory(start, stop)` and `remove_mem_hooks()` functions to the `_testcapi` module.
- bpo-30357 [https://bugs.python.org/issue?@action=redirect&bpo=30357]: test_thread: setUp() now uses support.threading_setup() and support.threading_cleanup() to wait until threads

complete to avoid random side effects on following tests. Initial patch written by Grzegorz Grzywacz.

- bpo-30197 [https://bugs.python.org/issue?@action=redirect&bpo=30197]: Enhanced functions swap_attr() and swap_item() in the test.support module. They now work when delete replaced attribute or item inside the with statement. The old value of the attribute or item (or None if it doesn’t exist) now will be assigned to the target of the “as” clause, if there is one.
- bpo-24932 [https://bugs.python.org/issue?@action=redirect&bpo=24932]: Use proper command line parsing in _testembed
- bpo-28950 [https://bugs.python.org/issue?@action=redirect&bpo=28950]: Disallow -j0 to be combined with -T/-l in regrtest command line arguments.
- bpo-28683 [https://bugs.python.org/issue?@action=redirect&bpo=28683]: Fix the tests that bind() a unix socket and raise PermissionError on Android for a non-root user.
- bpo-26936 [https://bugs.python.org/issue?@action=redirect&bpo=26936]: Fix the test_socket failures on Android - getservbyname(), getservbyport() and getaddrinfo() are broken on some Android API levels.
- bpo-28666 [https://bugs.python.org/issue?@action=redirect&bpo=28666]: Now test.support.rmtree is able to remove unwritable or unreadable directories.
- bpo-23839 [https://bugs.python.org/issue?@action=redirect&bpo=23839]: Various caches now are cleared before running every test file.
- bpo-26944 [https://bugs.python.org/issue?@action=redirect&bpo=26944]: Fix test_posix for Android where ‘id -G’ is entirely wrong or missing the effective gid.
- bpo-28409 [https://bugs.python.org/issue?@action=redirect&bpo=28409]: regrtest: fix the parser of command line arguments.

- bpo-28217 [https://bugs.python.org/issue?@action=redirect&bpo=28217]: Adds _testconsole module to test console input.
- bpo-26939 [https://bugs.python.org/issue?@action=redirect&bpo=26939]: Add the support.setswitchinterval() function to fix test_functools hanging on the Android armv7 qemu emulator.

Build

- bpo-31354 [https://bugs.python.org/issue?@action=redirect&bpo=31354]: Allow `--with-lto` to be used on all builds, not just `make profile-opt`.
- bpo-31370 [https://bugs.python.org/issue?@action=redirect&bpo=31370]: Remove support for building –without-threads. This option is not really useful anymore in the 21st century. Removing lots of conditional paths allows us to simplify the code base, including in difficult to maintain low-level internal code.
- bpo-31341 [https://bugs.python.org/issue?@action=redirect&bpo=31341]: Per **PEP 11** [https://peps.python.org/pep-0011/], support for the IRIX operating system was removed.
- bpo-30854 [https://bugs.python.org/issue?@action=redirect&bpo=30854]: Fix compile error when compiling –without-threads. Patch by Masayuki Yamamoto.
- bpo-30687 [https://bugs.python.org/issue?@action=redirect&bpo=30687]: Locate msbuild.exe on Windows when building rather than vcvarsall.bat
- bpo-20210 [https://bugs.python.org/issue?@action=redirect&bpo=20210]: Support the *disabled* marker in Setup files. Extension modules listed after this marker are not built at all, neither by the Makefile nor by setup.py.

- bpo-29941 [https://bugs.python.org/issue?@action=redirect&bpo=29941]: Add `--with-assertions` configure flag to explicitly enable C `assert()` checks. Defaults to off. `--with-pydebug` implies `--with-assertions`.
- bpo-28787 [https://bugs.python.org/issue?@action=redirect&bpo=28787]: Fix out-of-tree builds of Python when configured with `--with--dtrace`.
- bpo-29243 [https://bugs.python.org/issue?@action=redirect&bpo=29243]: Prevent unnecessary rebuilding of Python during `make test`, `make install` and some other make targets when configured with `--enable-optimizations`.
- bpo-23404 [https://bugs.python.org/issue?@action=redirect&bpo=23404]: Don't regenerate generated files based on file modification time anymore: the action is now explicit. Replace `make touch` with `make regen-all`.
- bpo-29643 [https://bugs.python.org/issue?@action=redirect&bpo=29643]: Fix `--enable-optimization` didn't work.
- bpo-27593 [https://bugs.python.org/issue?@action=redirect&bpo=27593]: sys.version and the platform module python_build(), python_branch(), and python_revision() functions now use git information rather than hg when building from a repo.
- bpo-29572 [https://bugs.python.org/issue?@action=redirect&bpo=29572]: Update Windows build and OS X installers to use OpenSSL 1.0.2k.
- bpo-27659 [https://bugs.python.org/issue?@action=redirect&bpo=27659]: Prohibit implicit C function declarations: use `-Werror=implicit-function-declaration` when possible (GCC and Clang, but it depends on the compiler version). Patch written by Chi Hsuan Yen.
- bpo-29384 [https://bugs.python.org/issue?@action=redirect&bpo=29384]: Remove old Be OS helper scripts.

- bpo-26851 [https://bugs.python.org/issue?@action=redirect&bpo=26851]: Set Android compilation and link flags.
- bpo-28768 [https://bugs.python.org/issue?@action=redirect&bpo=28768]: Fix implicit declaration of function _setmode. Patch by Masayuki Yamamoto
- bpo-29080 [https://bugs.python.org/issue?@action=redirect&bpo=29080]: Removes hard dependency on hg.exe from PCBuild/build.bat
- bpo-23903 [https://bugs.python.org/issue?@action=redirect&bpo=23903]: Added missed names to PC/python3.def.
- bpo-28762 [https://bugs.python.org/issue?@action=redirect&bpo=28762]: lockf() is available on Android API level 24, but the F_LOCK macro is not defined in android-ndk-r13.
- bpo-28538 [https://bugs.python.org/issue?@action=redirect&bpo=28538]: Fix the compilation error that occurs because if_nameindex() is available on Android API level 24, but the if_nameindex structure is not defined.
- bpo-20211 [https://bugs.python.org/issue?@action=redirect&bpo=20211]: Do not add the directory for installing C header files and the directory for installing object code libraries to the cross compilation search paths. Original patch by Thomas Petazzoni.
- bpo-28849 [https://bugs.python.org/issue?@action=redirect&bpo=28849]: Do not define sys.implementation._multiarch on Android.
- bpo-10656 [https://bugs.python.org/issue?@action=redirect&bpo=10656]: Fix out-of-tree building on AIX. Patch by Tristan Carel and Michael Haubenwallner.
- bpo-26359 [https://bugs.python.org/issue?@action=redirect&bpo=26359]: Rename –with-optimiations to –enable-optimizations.

- bpo-28444 [https://bugs.python.org/issue?@action=redirect&bpo=28444]: Fix missing extensions modules when cross compiling.
- bpo-28208 [https://bugs.python.org/issue?@action=redirect&bpo=28208]: Update Windows build and OS X installers to use SQLite 3.14.2.
- bpo-28248 [https://bugs.python.org/issue?@action=redirect&bpo=28248]: Update Windows build and OS X installers to use OpenSSL 1.0.2j.
- bpo-21124 [https://bugs.python.org/issue?@action=redirect&bpo=21124]: Fix building the _struct module on Cygwin by passing `NULL` instead of `&PyType_Type` to PyVarObject_HEAD_INIT. Patch by Masayuki Yamamoto.
- bpo-13756 [https://bugs.python.org/issue?@action=redirect&bpo=13756]: Fix building extensions modules on Cygwin. Patch by Roumen Petrov, based on original patch by Jason Tishler.
- bpo-21085 [https://bugs.python.org/issue?@action=redirect&bpo=21085]: Add configure check for siginfo_t.si_band, which Cygwin does not provide. Patch by Masayuki Yamamoto with review and rebase by Erik Bray.
- bpo-28258 [https://bugs.python.org/issue?@action=redirect&bpo=28258]: Fixed build with Estonian locale (python-config and distclean targets in Makefile). Patch by Arfrever Frehtes Taifersar Arahesis.
- bpo-26661 [https://bugs.python.org/issue?@action=redirect&bpo=26661]: setup.py now detects system libffi with multiarch wrapper.
- bpo-27979 [https://bugs.python.org/issue?@action=redirect&bpo=27979]: A full copy of libffi is no longer bundled for use when building _ctypes on non-OSX UNIX platforms. An installed copy of libffi is now required when building _ctypes on such platforms.
- bpo-15819 [https://bugs.python.org/issue?@action=redirect&bpo=15819]: Remove redundant include

search directory option for building outside the source tree.
- bpo-28676 [https://bugs.python.org/issue?@action=redirect&bpo=28676]: Prevent missing ‘getentropy’ declaration warning on macOS. Patch by Gareth Rees.

Windows

- bpo-31392 [https://bugs.python.org/issue?@action=redirect&bpo=31392]: Update Windows build to use OpenSSL 1.1.0f
- bpo-30389 [https://bugs.python.org/issue?@action=redirect&bpo=30389]: Adds detection of Visual Studio 2017 to distutils on Windows.
- bpo-31358 [https://bugs.python.org/issue?@action=redirect&bpo=31358]: zlib is no longer bundled in the CPython source, instead it is downloaded on demand just like bz2, lzma, OpenSSL, Tcl/Tk, and SQLite.
- bpo-31340 [https://bugs.python.org/issue?@action=redirect&bpo=31340]: Change to building with MSVC v141 (included with Visual Studio 2017)
- bpo-30581 [https://bugs.python.org/issue?@action=redirect&bpo=30581]: os.cpu_count() now returns the correct number of processors on Windows when the number of logical processors is greater than 64.
- bpo-30916 [https://bugs.python.org/issue?@action=redirect&bpo=30916]: Pre-build OpenSSL, Tcl and Tk and include the binaries in the build.
- bpo-30731 [https://bugs.python.org/issue?@action=redirect&bpo=30731]: Add a missing xmlns to python.manifest so that it matches the schema.
- bpo-30291 [https://bugs.python.org/issue?@action=redirect&bpo=30291]: Allow requiring 64-bit interpreters from py.exe using -64 suffix. Contributed by Steve (Gadget) Barnes.

- bpo-30362 [https://bugs.python.org/issue?@action=redirect&bpo=30362]: Adds list options (-0, -0p) to py.exe launcher. Contributed by Steve Barnes.
- bpo-23451 [https://bugs.python.org/issue?@action=redirect&bpo=23451]: Fix socket deprecation warnings in socketmodule.c. Patch by Segev Finer.
- bpo-30450 [https://bugs.python.org/issue?@action=redirect&bpo=30450]: The build process on Windows no longer depends on Subversion, instead pulling external code from GitHub via a Python script. If Python 3.6 is not found on the system (via `py -3.6`), NuGet is used to download a copy of 32-bit Python.
- bpo-29579 [https://bugs.python.org/issue?@action=redirect&bpo=29579]: Removes readme.txt from the installer.
- bpo-25778 [https://bugs.python.org/issue?@action=redirect&bpo=25778]: winreg does not truncate string correctly (Patch by Eryk Sun)
- bpo-28896 [https://bugs.python.org/issue?@action=redirect&bpo=28896]: Deprecate WindowsRegistryFinder and disable it by default
- bpo-28522 [https://bugs.python.org/issue?@action=redirect&bpo=28522]: Fixes mishandled buffer reallocation in getpathp.c
- bpo-28402 [https://bugs.python.org/issue?@action=redirect&bpo=28402]: Adds signed catalog files for stdlib on Windows.
- bpo-28333 [https://bugs.python.org/issue?@action=redirect&bpo=28333]: Enables Unicode for ps1/ps2 and input() prompts. (Patch by Eryk Sun)
- bpo-28251 [https://bugs.python.org/issue?@action=redirect&bpo=28251]: Improvements to help manuals on Windows.
- bpo-28110 [https://bugs.python.org/issue?@action=redirect&bpo=28110]: launcher.msi has different product codes between 32-bit and 64-bit

- bpo-28161 [https://bugs.python.org/issue?@action=redirect&bpo=28161]: Opening CON for write access fails
- bpo-28162 [https://bugs.python.org/issue?@action=redirect&bpo=28162]: WindowsConsoleIO readall() fails if first line starts with Ctrl+Z
- bpo-28163 [https://bugs.python.org/issue?@action=redirect&bpo=28163]: WindowsConsoleIO fileno() passes wrong flags to _open_osfhandle
- bpo-28164 [https://bugs.python.org/issue?@action=redirect&bpo=28164]: _PyIO_get_console_type fails for various paths
- bpo-28137 [https://bugs.python.org/issue?@action=redirect&bpo=28137]: Renames Windows path file to ._pth
- bpo-28138 [https://bugs.python.org/issue?@action=redirect&bpo=28138]: Windows ._pth file should allow import site

IDLE

- bpo-31493 [https://bugs.python.org/issue?@action=redirect&bpo=31493]: IDLE code context – fix code update and font update timers. Canceling timers prevents a warning message when test_idle completes.
- bpo-31488 [https://bugs.python.org/issue?@action=redirect&bpo=31488]: IDLE - Update non-key options in former extension classes. When applying configdialog changes, call .reload for each feature class. Change ParenMatch so updated options affect existing instances attached to existing editor windows.
- bpo-31477 [https://bugs.python.org/issue?@action=redirect&bpo=31477]: IDLE - Improve rstrip entry in doc. Strip trailing whitespace strips more than blank spaces. Multiline string literals are not skipped.

- bpo-31480 [https://bugs.python.org/issue?@action=redirect&bpo=31480]: IDLE - make tests pass with zzdummy extension disabled by default.
- bpo-31421 [https://bugs.python.org/issue?@action=redirect&bpo=31421]: Document how IDLE runs tkinter programs. IDLE calls tcl/tk update in the background in order to make live interaction and experimentation with tkinter applications much easier.
- bpo-31414 [https://bugs.python.org/issue?@action=redirect&bpo=31414]: IDLE – fix tk entry box tests by deleting first. Adding to an int entry is not the same as deleting and inserting because int(‘’) will fail.
- bpo-31051 [https://bugs.python.org/issue?@action=redirect&bpo=31051]: Rearrange IDLE configdialog GenPage into Window, Editor, and Help sections.
- bpo-30617 [https://bugs.python.org/issue?@action=redirect&bpo=30617]: IDLE - Add docstrings and tests for outwin subclass of editor. Move some data and functions from the class to module level. Patch by Cheryl Sabella.
- bpo-31287 [https://bugs.python.org/issue?@action=redirect&bpo=31287]: IDLE - Do not modify tkinter.message in test_configdialog.
- bpo-27099 [https://bugs.python.org/issue?@action=redirect&bpo=27099]: Convert IDLE’s built-in ‘extensions’ to regular features. About 10 IDLE features were implemented as supposedly optional extensions. Their different behavior could be confusing or worse for users and not good for maintenance. Hence the conversion. The main difference for users is that user configurable key bindings for builtin features are now handled uniformly. Now, editing a binding in a keyset only affects its value in the keyset. All bindings are defined together in the system-specific default keysets in config-extensions.def. All custom keysets are saved as a whole in config-extension.cfg. All take effect as

soon as one clicks Apply or Ok. The affected events are ‘<<force-open-completions>>’, ‘<<expand-word>>’, ‘<<force-open-calltip>>’, ‘<<flash-paren>>’, ‘<<format-paragraph>>’, ‘<<run-module>>’, ‘<<check-module>>’, and ‘<<zoom-height>>’. Any (global) customizations made before 3.6.3 will not affect their keyset-specific customization after 3.6.3. and vice versa. Initial patch by Charles Wohlganger.

- bpo-31206 [https://bugs.python.org/issue?@action=redirect&bpo=31206]: IDLE: Factor HighPage(Frame) class from ConfigDialog. Patch by Cheryl Sabella.
- bpo-31001 [https://bugs.python.org/issue?@action=redirect&bpo=31001]: Add tests for configdialog highlight tab. Patch by Cheryl Sabella.
- bpo-31205 [https://bugs.python.org/issue?@action=redirect&bpo=31205]: IDLE: Factor KeysPage(Frame) class from ConfigDialog. The slightly modified tests continue to pass. Patch by Cheryl Sabella.
- bpo-31130 [https://bugs.python.org/issue?@action=redirect&bpo=31130]: IDLE – stop leaks in test_configdialog. Initial patch by Victor Stinner.
- bpo-31002 [https://bugs.python.org/issue?@action=redirect&bpo=31002]: Add tests for configdialog keys tab. Patch by Cheryl Sabella.
- bpo-19903 [https://bugs.python.org/issue?@action=redirect&bpo=19903]: IDLE: Calltips use `inspect.signature` instead of `inspect.getfullargspec`. This improves calltips for builtins converted to use Argument Clinic. Patch by Louie Lu.
- bpo-31083 [https://bugs.python.org/issue?@action=redirect&bpo=31083]: IDLE - Add an outline of a TabPage class in configdialog. Update existing classes to match outline. Initial patch by Cheryl Sabella.
- bpo-31050 [https://bugs.python.org/issue?@action=redirect&bpo=31050]: Factor GenPage(Frame) class

from ConfigDialog. The slightly modified tests continue to pass. Patch by Cheryl Sabella.

- bpo-31004 [https://bugs.python.org/issue?@action=redirect&bpo=31004]: IDLE - Factor FontPage(Frame) class from ConfigDialog. Slightly modified tests continue to pass. Fix General tests. Patch mostly by Cheryl Sabella.
- bpo-30781 [https://bugs.python.org/issue?@action=redirect&bpo=30781]: IDLE - Use ttk widgets in ConfigDialog. Patches by Terry Jan Reedy and Cheryl Sabella.
- bpo-31060 [https://bugs.python.org/issue?@action=redirect&bpo=31060]: IDLE - Finish rearranging methods of ConfigDialog Grouping methods pertaining to each tab and the buttons will aid writing tests and improving the tabs and will enable splitting the groups into classes.
- bpo-30853 [https://bugs.python.org/issue?@action=redirect&bpo=30853]: IDLE – Factor a VarTrace class out of ConfigDialog. Instance tracers manages pairs consisting of a tk variable and a callback function. When tracing is turned on, setting the variable calls the function. Test coverage for the new class is 100%.
- bpo-31003 [https://bugs.python.org/issue?@action=redirect&bpo=31003]: IDLE: Add more tests for General tab.
- bpo-30993 [https://bugs.python.org/issue?@action=redirect&bpo=30993]: IDLE - Improve configdialog font page and tests. In configdialog: Document causal pathways in create_font_tab docstring. Simplify some attribute names. Move set_samples calls to var_changed_font (idea from Cheryl Sabella). Move related functions to positions after the create widgets function. In test_configdialog: Fix test_font_set so not order dependent. Fix renamed test_indent_scale so it tests the widget. Adjust tests for movement of

set_samples call. Add tests for load functions. Put all font tests in one class and tab indent tests in another. Except for two lines, these tests completely cover the related functions.

- bpo-30981 [https://bugs.python.org/issue?@action=redirect&bpo=30981]: IDLE – Add more configdialog font page tests.
- bpo-28523 [https://bugs.python.org/issue?@action=redirect&bpo=28523]: IDLE: replace ‘colour’ with ‘color’ in configdialog.
- bpo-30917 [https://bugs.python.org/issue?@action=redirect&bpo=30917]: Add tests for idlelib.config.IdleConf. Increase coverage from 46% to 96%. Patch by Louie Lu.
- bpo-30934 [https://bugs.python.org/issue?@action=redirect&bpo=30934]: Document coverage details for idlelib tests. Add section to idlelib/idle-test/README.txt. Include check that branches are taken both ways. Exclude IDLE-specific code that does not run during unit tests.
- bpo-30913 [https://bugs.python.org/issue?@action=redirect&bpo=30913]: IDLE: Document ConfigDialog tk Vars, methods, and widgets in docstrings This will facilitate improving the dialog and splitting up the class. Original patch by Cheryl Sabella.
- bpo-30899 [https://bugs.python.org/issue?@action=redirect&bpo=30899]: IDLE: Add tests for ConfigParser subclasses in config. Patch by Louie Lu.
- bpo-30881 [https://bugs.python.org/issue?@action=redirect&bpo=30881]: IDLE: Add docstrings to browser.py. Patch by Cheryl Sabella.
- bpo-30851 [https://bugs.python.org/issue?@action=redirect&bpo=30851]: IDLE: Remove unused variables in configdialog. One is a duplicate, one is set but cannot be altered by users. Patch by Cheryl Sabella.

- bpo-30870 [https://bugs.python.org/issue?@action=redirect&bpo=30870]: IDLE: In Settings dialog, select font with Up, Down keys as well as mouse. Initial patch by Louie Lu.
- bpo-8231 [https://bugs.python.org/issue?@action=redirect&bpo=8231]: IDLE: call config.IdleConf.GetUserCfgDir only once.
- bpo-30779 [https://bugs.python.org/issue?@action=redirect&bpo=30779]: IDLE: Factor ConfigChanges class from configdialog, put in config; test. * In config, put dump test code in a function; run it and unittest in ‘if __name__ == ‘__main__’. * Add class config.ConfigChanges based on changes_class_v4.py on bpo issue. * Add class test_config.ChangesTest, partly using configdialog_tests_v1.py. * Revise configdialog to use ConfigChanges; see tracker msg297804. * Revise test_configdialog to match configdialog changes. * Remove configdialog functions unused or moved to ConfigChanges. Cheryl Sabella contributed parts of the patch.
- bpo-30777 [https://bugs.python.org/issue?@action=redirect&bpo=30777]: IDLE: configdialog - Add docstrings and fix comments. Patch by Cheryl Sabella.
- bpo-30495 [https://bugs.python.org/issue?@action=redirect&bpo=30495]: IDLE: Improve textview with docstrings, PEP8 names, and more tests. Patch by Cheryl Sabella.
- bpo-30723 [https://bugs.python.org/issue?@action=redirect&bpo=30723]: IDLE: Make several improvements to parenmatch. Add ‘parens’ style to highlight both opener and closer. Make ‘default’ style, which is not default, a synonym for ‘opener’. Make time-delay work the same with all styles. Add help for config dialog extensions tab, including help for parenmatch. Add new tests. Original patch by Charles Wohlganger.
- bpo-30674 [https://bugs.python.org/issue?@action=redirect&bpo=30674]: IDLE: add docstrings to grep

module. Patch by Cheryl Sabella

- bpo-21519 [https://bugs.python.org/issue?@action=redirect&bpo=21519]: IDLE's basic custom key entry dialog now detects duplicates properly. Original patch by Saimadhav Heblikar.
- bpo-29910 [https://bugs.python.org/issue?@action=redirect&bpo=29910]: IDLE no longer deletes a character after commenting out a region by a key shortcut. Add `return 'break'` for this and other potential conflicts between IDLE and default key bindings.
- bpo-30728 [https://bugs.python.org/issue?@action=redirect&bpo=30728]: Review and change idlelib.configdialog names. Lowercase method and attribute names. Replace 'colour' with 'color', expand overly cryptic names, delete unneeded underscores. Replace `import *` with specific imports. Patches by Cheryl Sabella.
- bpo-6739 [https://bugs.python.org/issue?@action=redirect&bpo=6739]: IDLE: Verify user-entered key sequences by trying to bind them with tk. Add tests for all 3 validation functions. Original patch by G Polo. Tests added by Cheryl Sabella.
- bpo-15786 [https://bugs.python.org/issue?@action=redirect&bpo=15786]: Fix several problems with IDLE's autocompletion box. The following should now work: clicking on selection box items; using the scrollbar; selecting an item by hitting Return. Hangs on MacOSX should no longer happen. Patch by Louie Lu.
- bpo-25514 [https://bugs.python.org/issue?@action=redirect&bpo=25514]: Add doc subsubsection about IDLE failure to start. Popup no-connection message directs users to this section.
- bpo-30642 [https://bugs.python.org/issue?@action=redirect&bpo=30642]: Fix reference leaks in IDLE tests. Patches by Louie Lu and Terry Jan Reedy.

- bpo-30495 [https://bugs.python.org/issue?@action=redirect&bpo=30495]: Add docstrings for textview.py and use PEP8 names. Patches by Cheryl Sabella and Terry Jan Reedy.
- bpo-30290 [https://bugs.python.org/issue?@action=redirect&bpo=30290]: Help-about: use pep8 names and add tests. Increase coverage to 100%. Patches by Louie Lu, Cheryl Sabella, and Terry Jan Reedy.
- bpo-30303 [https://bugs.python.org/issue?@action=redirect&bpo=30303]: Add _utest option to textview; add new tests. Increase coverage to 100%. Patches by Louie Lu and Terry Jan Reedy.
- bpo-29071 [https://bugs.python.org/issue?@action=redirect&bpo=29071]: IDLE colors f-string prefixes (but not invalid ur prefixes).
- bpo-28572 [https://bugs.python.org/issue?@action=redirect&bpo=28572]: Add 10% to coverage of IDLE’s test_configdialog. Update and augment description of the configuration system.

Tools/Demos

- bpo-30983 [https://bugs.python.org/issue?@action=redirect&bpo=30983]: gdb integration commands (py-bt, etc.) work on optimized shared builds now, too. **PEP 523** [https://peps.python.org/pep-0523/] introduced _PyEval_EvalFrameDefault which inlines PyEval_EvalFrameEx on non-debug shared builds. This broke the ability to use py-bt, py-up, and a few other Python-specific gdb integrations. The problem is fixed by only looking for _PyEval_EvalFrameDefault frames in python-gdb.py. Original patch by Bruno “Polaco” Penteado.
- bpo-29748 [https://bugs.python.org/issue?@action=redirect&bpo=29748]: Added the slice index converter in Argument Clinic.

- bpo-24037 [https://bugs.python.org/issue?@action=redirect&bpo=24037]: Argument Clinic now uses the converter `bool(accept={int})` rather than `int` for semantical booleans. This avoids repeating the default value for Python and C and will help in converting to `bool` in future.
- bpo-29367 [https://bugs.python.org/issue?@action=redirect&bpo=29367]: python-gdb.py now supports also `method-wrapper` (`wrapperobject`) objects.
- bpo-28023 [https://bugs.python.org/issue?@action=redirect&bpo=28023]: Fix python-gdb.py didn't support new dict implementation.
- bpo-15369 [https://bugs.python.org/issue?@action=redirect&bpo=15369]: The pybench and pystone microbenchmark have been removed from Tools. Please use the new Python benchmark suite https://github.com/python/pyperformance which is more reliable and includes a portable version of pybench working on Python 2 and Python 3.
- bpo-28102 [https://bugs.python.org/issue?@action=redirect&bpo=28102]: The zipfile module CLI now prints usage to stderr. Patch by Stephen J. Turnbull.

C API

- bpo-31338 [https://bugs.python.org/issue?@action=redirect&bpo=31338]: Added the `Py_UNREACHABLE()` macro for code paths which are never expected to be reached. This and a few other useful macros are now documented in the C API manual.
- bpo-30832 [https://bugs.python.org/issue?@action=redirect&bpo=30832]: Remove own implementation for thread-local storage. CPython has provided the own implementation for thread-local storage (TLS) on Python/thread.c, it's used in the case which a platform has not supplied native TLS. However, currently all

supported platforms (Windows and pthreads) have provided native TLS and defined the Py_HAVE_NATIVE_TLS macro with unconditional in any case.

- bpo-30708 [https://bugs.python.org/issue?@action=redirect&bpo=30708]: PyUnicode_AsWideCharString() now raises a ValueError if the second argument is NULL and the wchar_t* string contains null characters.
- bpo-16500 [https://bugs.python.org/issue?@action=redirect&bpo=16500]: Deprecate PyOS_AfterFork() and add PyOS_BeforeFork(), PyOS_AfterFork_Parent() and PyOS_AfterFork_Child().
- bpo-6532 [https://bugs.python.org/issue?@action=redirect&bpo=6532]: The type of results of PyThread_start_new_thread() and PyThread_get_thread_ident(), and the id parameter of PyThreadState_SetAsyncExc() changed from “long” to “unsigned long”.
- bpo-27867 [https://bugs.python.org/issue?@action=redirect&bpo=27867]: Function PySlice_GetIndicesEx() is deprecated and replaced with a macro if Py_LIMITED_API is not set or set to the value between 0x03050400 and 0x03060000 (not including) or 0x03060100 or higher. Added functions PySlice_Unpack() and PySlice_AdjustIndices().
- bpo-29083 [https://bugs.python.org/issue?@action=redirect&bpo=29083]: Fixed the declaration of some public API functions. PyArg_VaParse() and PyArg_VaParseTupleAndKeywords() were not available in limited API. PyArg_ValidateKeywordArguments(), PyArg_UnpackTuple() and Py_BuildValue() were not available in limited API of version < 3.3 when PY_SSIZE_T_CLEAN is defined.
- bpo-28769 [https://bugs.python.org/issue?@action=redirect&bpo=28769]: The result of PyUnicode_AsUTF8AndSize() and PyUnicode_AsUTF8() is now of type `const char *` rather of `char *`.

- bpo-29058 [https://bugs.python.org/issue?@action=redirect&bpo=29058]: All stable API extensions added after Python 3.2 are now available only when Py_LIMITED_API is set to the PY_VERSION_HEX value of the minimum Python version supporting this API.
- bpo-28822 [https://bugs.python.org/issue?@action=redirect&bpo=28822]: The index parameters *start* and *end* of PyUnicode_FindChar() are now adjusted to behave like `str[start:end]`.
- bpo-28808 [https://bugs.python.org/issue?@action=redirect&bpo=28808]: PyUnicode_CompareWithASCIIString() now never raises exceptions.
- bpo-28761 [https://bugs.python.org/issue?@action=redirect&bpo=28761]: The fields name and doc of structures PyMemberDef, PyGetSetDef, PyStructSequence_Field, PyStructSequence_Desc, and wrapperbase are now of type `const char *` rather of `char *`.
- bpo-28748 [https://bugs.python.org/issue?@action=redirect&bpo=28748]: Private variable _Py_PackageContext is now of type `const char *` rather of `char *`.
- bpo-19569 [https://bugs.python.org/issue?@action=redirect&bpo=19569]: Compiler warnings are now emitted if use most of deprecated functions.
- bpo-28426 [https://bugs.python.org/issue?@action=redirect&bpo=28426]: Deprecated undocumented functions PyUnicode_AsEncodedObject(), PyUnicode_AsDecodedObject(), PyUnicode_AsDecodedUnicode() and PyUnicode_AsEncodedUnicode().

Python 3.6.6 final

Release date: 2018-06-27

There were no new changes in version 3.6.6.

Python 3.6.6 release candidate 1

Release date: 2018-06-11

Core and Builtins

- bpo-33786 [https://bugs.python.org/issue?@action=redirect&bpo=33786]: Fix asynchronous generators to handle GeneratorExit in athrow() correctly
- bpo-30654 [https://bugs.python.org/issue?@action=redirect&bpo=30654]: Fixed reset of the SIGINT handler to SIG_DFL on interpreter shutdown even when there was a custom handler set previously. Patch by Philipp Kerling.
- bpo-33622 [https://bugs.python.org/issue?@action=redirect&bpo=33622]: Fixed a leak when the garbage collector fails to add an object with the `__del__` method or referenced by it into the **`gc.garbage`** list. **`PyGC_Collect()`** can now be called when an exception is set and preserves it.
- bpo-31849 [https://bugs.python.org/issue?@action=redirect&bpo=31849]: Fix signed/unsigned comparison warning in pyhash.c.
- bpo-33391 [https://bugs.python.org/issue?@action=redirect&bpo=33391]: Fix a leak in set_symmetric_difference().
- bpo-28055 [https://bugs.python.org/issue?@action=redirect&bpo=28055]: Fix unaligned accesses in siphash24(). Patch by Rolf Eike Beer.
- bpo-33231 [https://bugs.python.org/issue?@action=redirect&bpo=33231]: Fix potential memory leak in

`normalizestring()`.

- bpo-29922 [https://bugs.python.org/issue?@action=redirect&bpo=29922]: Improved error messages in ‘async with’ when `__aenter__()` or `__aexit__()` return non-awaitable object.
- bpo-33199 [https://bugs.python.org/issue?@action=redirect&bpo=33199]: Fix `ma_version_tag` in dict implementation is uninitialized when copying from key-sharing dict.
- bpo-33041 [https://bugs.python.org/issue?@action=redirect&bpo=33041]: Fixed jumping when the function contains an `async for` loop.
- bpo-32282 [https://bugs.python.org/issue?@action=redirect&bpo=32282]: Fix an unnecessary ifdef in the include of VersionHelpers.h in socketmodule on Windows.
- bpo-21983 [https://bugs.python.org/issue?@action=redirect&bpo=21983]: Fix a crash in `ctypes.cast()` in case the type argument is a ctypes structured data type. Patch by Eryk Sun and Oren Milman.

Library

- bpo-30167 [https://bugs.python.org/issue?@action=redirect&bpo=30167]: Prevent site.main() exception if PYTHONSTARTUP is set. Patch by Steve Weber.
- bpo-33812 [https://bugs.python.org/issue?@action=redirect&bpo=33812]: Datetime instance d with non-None tzinfo, but with d.tzinfo.utcoffset(d) returning None is now treated as naive by the astimezone() method.
- bpo-30805 [https://bugs.python.org/issue?@action=redirect&bpo=30805]: Avoid race condition with debug logging
- bpo-33767 [https://bugs.python.org/issue?@action=redirect&bpo=33767]: The concatenation (+) and repetition (*) sequence operations now raise **`TypeError`**

instead of `SystemError` when performed on `mmap.mmap` objects. Patch by Zackery Spytz.

- bpo-32684 [https://bugs.python.org/issue?@action=redirect&bpo=32684]: Fix gather to propagate cancellation of itself even with return_exceptions.
- bpo-33674 [https://bugs.python.org/issue?@action=redirect&bpo=33674]: Fix a race condition in SSLProtocol.connection_made() of asyncio.sslproto: start immediately the handshake instead of using call_soon(). Previously, data_received() could be called before the handshake started, causing the handshake to hang or fail.
- bpo-31647 [https://bugs.python.org/issue?@action=redirect&bpo=31647]: Fixed bug where calling write_eof() on a _SelectorSocketTransport after it's already closed raises AttributeError.
- bpo-33672 [https://bugs.python.org/issue?@action=redirect&bpo=33672]: Fix Task.__repr__ crash with Cython's bogus coroutines
- bpo-33469 [https://bugs.python.org/issue?@action=redirect&bpo=33469]: Fix RuntimeError after closing loop that used run_in_executor
- bpo-11874 [https://bugs.python.org/issue?@action=redirect&bpo=11874]: Use a better regex when breaking usage into wrappable parts. Avoids bogus assertion errors from custom metavar strings.
- bpo-30877 [https://bugs.python.org/issue?@action=redirect&bpo=30877]: Fixed a bug in the Python implementation of the JSON decoder that prevented the cache of parsed strings from clearing after finishing the decoding. Based on patch by c-fos.
- bpo-33548 [https://bugs.python.org/issue?@action=redirect&bpo=33548]: tempfile._candidate_tempdir_list should consider common TEMP locations

- bpo-33542 [https://bugs.python.org/issue?@action=redirect&bpo=33542]: Prevent `uuid.get_node` from using a DUID instead of a MAC on Windows. Patch by Zvi Effron
- bpo-26819 [https://bugs.python.org/issue?@action=redirect&bpo=26819]: Fix race condition with `ReadTransport.resume_reading` in Windows proactor event loop.
- bpo-28556 [https://bugs.python.org/issue?@action=redirect&bpo=28556]: Minor fixes in typing module: add annotations to `NamedTuple.__new__`, pass `*args` and `**kwds` in `Generic.__new__`. Original PRs by Paulius Šarka and Chad Dombrova.
- bpo-20087 [https://bugs.python.org/issue?@action=redirect&bpo=20087]: Updated alias mapping with glibc 2.27 supported locales.
- bpo-33422 [https://bugs.python.org/issue?@action=redirect&bpo=33422]: Fix trailing quotation marks getting deleted when looking up byte/string literals on pydoc. Patch by Andrés Delfino.
- bpo-33197 [https://bugs.python.org/issue?@action=redirect&bpo=33197]: Update error message when constructing invalid inspect.Parameters Patch by Donghee Na.
- bpo-33383 [https://bugs.python.org/issue?@action=redirect&bpo=33383]: Fixed crash in the get() method of the **`dbm.ndbm`** database object when it is called with a single argument.
- bpo-33329 [https://bugs.python.org/issue?@action=redirect&bpo=33329]: Fix multiprocessing regression on newer glibcs
- bpo-991266 [https://bugs.python.org/issue?@action=redirect&bpo=991266]: Fix quoting of the `Comment` attribute of **`http.cookies.SimpleCookie`**.
- bpo-33131 [https://bugs.python.org/issue?@action=redirect&bpo=33131]: Upgrade bundled version of pip

to 10.0.1.

- bpo-33308 [https://bugs.python.org/issue?@action=redirect&bpo=33308]: Fixed a crash in the **`parser`** module when converting an ST object to a tree of tuples or lists with `line_info=False` and `col_info=True`.
- bpo-33263 [https://bugs.python.org/issue?@action=redirect&bpo=33263]: Fix FD leak in `_SelectorSocketTransport` Patch by Vlad Starostin.
- bpo-33256 [https://bugs.python.org/issue?@action=redirect&bpo=33256]: Fix display of `<module>` call in the html produced by `cgitb.html()`. Patch by Stéphane Blondon.
- bpo-33203 [https://bugs.python.org/issue?@action=redirect&bpo=33203]: `random.Random.choice()` now raises `IndexError` for empty sequences consistently even when called from subclasses without a `getrandbits()` implementation.
- bpo-33224 [https://bugs.python.org/issue?@action=redirect&bpo=33224]: Update difflib.mdiff() for **PEP 479** [https://peps.python.org/pep-0479/]. Convert an uncaught StopIteration in a generator into a return-statement.
- bpo-33209 [https://bugs.python.org/issue?@action=redirect&bpo=33209]: End framing at the end of C implementation of **`pickle.Pickler.dump()`**.
- bpo-32861 [https://bugs.python.org/issue?@action=redirect&bpo=32861]: The urllib.robotparser’s `__str__` representation now includes wildcard entries and the “Crawl-delay” and “Request-rate” fields. Patch by Michael Lazar.
- bpo-33096 [https://bugs.python.org/issue?@action=redirect&bpo=33096]: Allow ttk.Treeview.insert to insert iid that has a false boolean value. Note iid=0 and iid=False would be same. Patch by Garvit Khatri.
- bpo-33127 [https://bugs.python.org/issue?@action=redirect&bpo=33127]: The ssl module now compiles with LibreSSL 2.7.1.

- bpo-33021 [https://bugs.python.org/issue?@action=redirect&bpo=33021]: Release the GIL during fstat() calls, avoiding hang of all threads when calling mmap.mmap(), os.urandom(), and random.seed(). Patch by Nir Soffer.
- bpo-27683 [https://bugs.python.org/issue?@action=redirect&bpo=27683]: Fix a regression in **`ipaddress`** that result of **`hosts()`** is empty when the network is constructed by a tuple containing an integer mask and only 1 bit left for addresses.
- bpo-32844 [https://bugs.python.org/issue?@action=redirect&bpo=32844]: Fix wrong redirection of a low descriptor (0 or 1) to stderr in subprocess if another low descriptor is closed.
- bpo-31908 [https://bugs.python.org/issue?@action=redirect&bpo=31908]: Fix output of cover files for `trace` module command-line tool. Previously emitted cover files only when `--missing` option was used. Patch by Michael Selik.
- bpo-31457 [https://bugs.python.org/issue?@action=redirect&bpo=31457]: If nested log adapters are used, the inner `process()` methods are no longer omitted.
- bpo-16865 [https://bugs.python.org/issue?@action=redirect&bpo=16865]: Support arrays >=2GiB in **`ctypes`**. Patch by Segev Finer.
- bpo-31238 [https://bugs.python.org/issue?@action=redirect&bpo=31238]: pydoc: the stop() method of the private ServerThread class now waits until DocServer.serve_until_quit() completes and then explicitly sets its docserver attribute to None to break a reference cycle.

Documentation

- bpo-33503 [https://bugs.python.org/issue?@action=redirect&bpo=33503]: Fix broken pypi link

- bpo-33421 [https://bugs.python.org/issue?@action=redirect&bpo=33421]: Add missing documentation for `typing.AsyncContextManager`.
- bpo-33378 [https://bugs.python.org/issue?@action=redirect&bpo=33378]: Add Korean language switcher for https://docs.python.org/3/
- bpo-33276 [https://bugs.python.org/issue?@action=redirect&bpo=33276]: Clarify that the `__path__` attribute on modules cannot be just any value.
- bpo-33201 [https://bugs.python.org/issue?@action=redirect&bpo=33201]: Modernize documentation for writing C extension types.
- bpo-33195 [https://bugs.python.org/issue?@action=redirect&bpo=33195]: Deprecate `Py_UNICODE` usage in `c-api/arg` document. `Py_UNICODE` related APIs are deprecated since Python 3.3, but it is missed in the document.
- bpo-33126 [https://bugs.python.org/issue?@action=redirect&bpo=33126]: Document PyBuffer_ToContiguous().
- bpo-27212 [https://bugs.python.org/issue?@action=redirect&bpo=27212]: Modify documentation for the **`islice()`** recipe to consume initial values up to the start index.
- bpo-28247 [https://bugs.python.org/issue?@action=redirect&bpo=28247]: Update **`zipapp`** documentation to describe how to make standalone applications.
- bpo-18802 [https://bugs.python.org/issue?@action=redirect&bpo=18802]: Documentation changes for ipaddress. Patch by Jon Foster and Berker Peksag.
- bpo-27428 [https://bugs.python.org/issue?@action=redirect&bpo=27428]: Update documentation to clarify that `WindowsRegistryFinder` implements `MetaPathFinder`. (Patch by Himanshu Lakhara)
- bpo-8243 [https://bugs.python.org/issue?@action=redirect&bpo=8243]: Add a note about curses.addch and curses.addstr

exception behavior when writing outside a window, or pad.

- bpo-31432 [https://bugs.python.org/issue?@action=redirect&bpo=31432]: Clarify meaning of CERT_NONE, CERT_OPTIONAL, and CERT_REQUIRED flags for ssl.SSLContext.verify_mode.

Tests

- bpo-33655 [https://bugs.python.org/issue?@action=redirect&bpo=33655]: Ignore test_posix_fallocate failures on BSD platforms that might be due to running on ZFS.
- bpo-19417 [https://bugs.python.org/issue?@action=redirect&bpo=19417]: Add test_bdb.py.

Build

- bpo-5755 [https://bugs.python.org/issue?@action=redirect&bpo=5755]: Move `-Wstrict-prototypes` option to `CFLAGS_NODIST` from `OPT`. This option emitted annoying warnings when building extension modules written in C++.
- bpo-33614 [https://bugs.python.org/issue?@action=redirect&bpo=33614]: Ensures module definition files for the stable ABI on Windows are correctly regenerated.
- bpo-33522 [https://bugs.python.org/issue?@action=redirect&bpo=33522]: Enable CI builds on Visual Studio Team Services at https://python.visualstudio.com/cpython
- bpo-33012 [https://bugs.python.org/issue?@action=redirect&bpo=33012]: Add `-Wno-cast-function-type` for gcc 8 for silencing warnings about function casts like casting to PyCFunction in method definition lists.
- bpo-33394 [https://bugs.python.org/issue?@action=redirect&bpo=33394]: Enable the verbose build for

extension modules, when GNU make is passed macros on the command line.

Windows

- bpo-33184 [https://bugs.python.org/issue?@action=redirect&bpo=33184]: Update Windows installer to OpenSSL 1.0.2o.

macOS

- bpo-33184 [https://bugs.python.org/issue?@action=redirect&bpo=33184]: Update macOS installer build to use OpenSSL 1.0.2o.

IDLE

- bpo-33656 [https://bugs.python.org/issue?@action=redirect&bpo=33656]: On Windows, add API call saying that tk scales for DPI. On Windows 8.1+ or 10, with DPI compatibility properties of the Python binary unchanged, and a monitor resolution greater than 96 DPI, this should make text and lines sharper. It should otherwise have no effect.
- bpo-33768 [https://bugs.python.org/issue?@action=redirect&bpo=33768]: Clicking on a context line moves that line to the top of the editor window.
- bpo-33763 [https://bugs.python.org/issue?@action=redirect&bpo=33763]: IDLE: Use read-only text widget for code context instead of label widget.
- bpo-33664 [https://bugs.python.org/issue?@action=redirect&bpo=33664]: Scroll IDLE editor text by lines. Previously, the mouse wheel and scrollbar slider moved text by a fixed number of pixels, resulting in partial lines at the top of the editor box. The change also applies to

the shell and grep output windows, but not to read-only text views.

- bpo-33679 [https://bugs.python.org/issue?@action=redirect&bpo=33679]: Enable theme-specific color configuration for Code Context. Use the Highlights tab to see the setting for built-in themes or add settings to custom themes.
- bpo-33642 [https://bugs.python.org/issue?@action=redirect&bpo=33642]: Display up to maxlines non-blank lines for Code Context. If there is no current context, show a single blank line.
- bpo-33628 [https://bugs.python.org/issue?@action=redirect&bpo=33628]: IDLE: Cleanup codecontext.py and its test.
- bpo-33564 [https://bugs.python.org/issue?@action=redirect&bpo=33564]: IDLE's code context now recognizes async as a block opener.
- bpo-29706 [https://bugs.python.org/issue?@action=redirect&bpo=29706]: IDLE now colors async and await as keywords in 3.6. They become full keywords in 3.7.
- bpo-21474 [https://bugs.python.org/issue?@action=redirect&bpo=21474]: Update word/identifier definition from ascii to unicode. In text and entry boxes, this affects selection by double-click, movement left/right by control-left/right, and deletion left/right by control-BACKSPACE/DEL.
- bpo-33204 [https://bugs.python.org/issue?@action=redirect&bpo=33204]: IDLE: consistently color invalid string prefixes. A ‘u’ string prefix cannot be paired with either ‘r’ or ‘f’. Consistently color as much of the prefix, starting at the right, as is valid. Revise and extend colorizer test.
- bpo-32831 [https://bugs.python.org/issue?@action=redirect&bpo=32831]: Add docstrings and tests for codecontext.

Tools/Demos

- bpo-33189 [https://bugs.python.org/issue?@action=redirect&bpo=33189]: **pygettext.py** now recognizes only literal strings as docstrings and translatable strings, and rejects bytes literals and f-string expressions.
- bpo-31920 [https://bugs.python.org/issue?@action=redirect&bpo=31920]: Fixed handling directories as arguments in the `pygettext` script. Based on patch by Oleg Krasnikov.
- bpo-29673 [https://bugs.python.org/issue?@action=redirect&bpo=29673]: Fix pystackv and pystack gdbinit macros.
- bpo-32885 [https://bugs.python.org/issue?@action=redirect&bpo=32885]: Add an `-n` flag for `Tools/scripts/pathfix.py` to disable automatic backup creation (files with ~ suffix).
- bpo-31583 [https://bugs.python.org/issue?@action=redirect&bpo=31583]: Fix 2to3 for using with –add-suffix option but without –output-dir option for relative path to files in current directory.

C API

- bpo-32374 [https://bugs.python.org/issue?@action=redirect&bpo=32374]: Document that m_traverse for multi-phase initialized modules can be called with m_state=NULL, and add a sanity check

Python 3.6.5 final

Release date: 2018-03-28

Tests

- bpo-32872 [https://bugs.python.org/issue?@action=redirect&bpo=32872]: Avoid regrtest compatibility issue with namespace packages.

Build

- bpo-33163 [https://bugs.python.org/issue?@action=redirect&bpo=33163]: Upgrade pip to 9.0.3 and setuptools to v39.0.1.

Python 3.6.5 release candidate 1

Release date: 2018-03-13

Security

- bpo-33001 [https://bugs.python.org/issue?@action=redirect&bpo=33001]: Minimal fix to prevent buffer overrun in os.symlink on Windows
- bpo-32981 [https://bugs.python.org/issue?@action=redirect&bpo=32981]: Regexes in difflib and poplib were vulnerable to catastrophic backtracking. These regexes formed potential DOS vectors (REDOS). They have been refactored. This resolves **CVE 2018-1060** [https://www.cve.org/CVERecord?id=CVE-2018-1060] and **CVE 2018-1061** [https://www.cve.org/CVERecord?id=CVE-2018-1061]. Patch by Jamie Davis.

Core and Builtins

- bpo-33026 [https://bugs.python.org/issue?@action=redirect&bpo=33026]: Fixed jumping out of “with” block by setting f_lineno.
- bpo-17288 [https://bugs.python.org/issue?@action=redirect&bpo=17288]: Prevent jumps from ‘return’

and ‘exception’ trace events.

- bpo-32889 [https://bugs.python.org/issue?@action=redirect&bpo=32889]: Update Valgrind suppression list to account for the rename of `Py_ADDRESS_IN_RANG` to `address_in_range`.
- bpo-32650 [https://bugs.python.org/issue?@action=redirect&bpo=32650]: Pdb and other debuggers dependent on bdb.py will correctly step over (next command) native coroutines. Patch by Pablo Galindo.
- bpo-32685 [https://bugs.python.org/issue?@action=redirect&bpo=32685]: Improve suggestion when the Python 2 form of print statement is either present on the same line as the header of a compound statement or else terminated by a semi-colon instead of a newline. Patch by Nitish Chandra.
- bpo-32583 [https://bugs.python.org/issue?@action=redirect&bpo=32583]: Fix possible crashing in builtin Unicode decoders caused by write out-of-bound errors when using customized decode error handlers.
- bpo-26163 [https://bugs.python.org/issue?@action=redirect&bpo=26163]: Improved frozenset() hash to create more distinct hash values when faced with datasets containing many similar values.
- bpo-27169 [https://bugs.python.org/issue?@action=redirect&bpo=27169]: The `__debug__` constant is now optimized out at compile time. This fixes also bpo-22091 [https://bugs.python.org/issue?@action=redirect&bpo=22091].
- bpo-32329 [https://bugs.python.org/issue?@action=redirect&bpo=32329]: `sys.flags.hash_randomization` is now properly set to 0 when hash randomization is turned off by `PYTHONHASHSEED=0`.
- bpo-30416 [https://bugs.python.org/issue?@action=redirect&bpo=30416]: The optimizer is now protected from spending much time doing complex calculations and consuming much memory for creating large constants in constant folding.

- bpo-18533 [https://bugs.python.org/issue?@action=redirect&bpo=18533]: `repr()` on a dict containing its own `values()` or `items()` no longer raises `RecursionError`; OrderedDict similarly. Instead, use `...`, as for other recursive structures. Patch by Ben North.
- bpo-32028 [https://bugs.python.org/issue?@action=redirect&bpo=32028]: Leading whitespace is now correctly ignored when generating suggestions for converting Py2 print statements to Py3 builtin print function calls. Patch by Sanyam Khurana.
- bpo-32137 [https://bugs.python.org/issue?@action=redirect&bpo=32137]: The repr of deeply nested dict now raises a RecursionError instead of crashing due to a stack overflow.

Library

- bpo-33064 [https://bugs.python.org/issue?@action=redirect&bpo=33064]: lib2to3 now properly supports trailing commas after `*args` and `**kwargs` in function signatures.
- bpo-31804 [https://bugs.python.org/issue?@action=redirect&bpo=31804]: Avoid failing in multiprocessing.Process if the standard streams are closed or None at exit.
- bpo-33037 [https://bugs.python.org/issue?@action=redirect&bpo=33037]: Skip sending/receiving data after SSL transport closing.
- bpo-30353 [https://bugs.python.org/issue?@action=redirect&bpo=30353]: Fix ctypes pass-by-value for structs on 64-bit Cygwin/MinGW.
- bpo-33009 [https://bugs.python.org/issue?@action=redirect&bpo=33009]: Fix inspect.signature() for single-parameter partialmethods.
- bpo-32969 [https://bugs.python.org/issue?@action=redirect&bpo=32969]: Expose several missing

constants in zlib and fix corresponding documentation.

- bpo-32713 [https://bugs.python.org/issue?@action=redirect&bpo=32713]: Fixed tarfile.itn handling of out-of-bounds float values. Patch by Joffrey Fuhrer.
- bpo-30622 [https://bugs.python.org/issue?@action=redirect&bpo=30622]: The ssl module now detects missing NPN support in LibreSSL.
- bpo-32922 [https://bugs.python.org/issue?@action=redirect&bpo=32922]: dbm.open() now encodes filename with the filesystem encoding rather than default encoding.
- bpo-32859 [https://bugs.python.org/issue?@action=redirect&bpo=32859]: In `os.dup2`, don’t check every call whether the `dup3` syscall exists or not.
- bpo-21060 [https://bugs.python.org/issue?@action=redirect&bpo=21060]: Rewrite confusing message from setup.py upload from “No dist file created in earlier command” to the more helpful “Must create and upload files in one command”.
- bpo-32857 [https://bugs.python.org/issue?@action=redirect&bpo=32857]: In **`tkinter`**, `after_cancel(None)` now raises a **`ValueError`** instead of canceling the first scheduled function. Patch by Cheryl Sabella.
- bpo-32852 [https://bugs.python.org/issue?@action=redirect&bpo=32852]: Make sure sys.argv remains as a list when running trace.
- bpo-32841 [https://bugs.python.org/issue?@action=redirect&bpo=32841]: Fixed `asyncio.Condition` issue which silently ignored cancellation after notifying and cancelling a conditional lock. Patch by Bar Harel.
- bpo-31787 [https://bugs.python.org/issue?@action=redirect&bpo=31787]: Fixed refleaks of `__init__()` methods in various modules. (Contributed by Oren Milman)
- bpo-30157 [https://bugs.python.org/issue?@action=redirect&bpo=30157]: Fixed guessing quote and

delimiter in csv.Sniffer.sniff() when only the last field is quoted. Patch by Jake Davis.

- bpo-32394 [https://bugs.python.org/issue?@action=redirect&bpo=32394]: socket: Remove TCP_FASTOPEN, TCP_KEEPCNT flags on older version Windows during run-time.
- bpo-32777 [https://bugs.python.org/issue?@action=redirect&bpo=32777]: Fix a rare but potential pre-exec child process deadlock in subprocess on POSIX systems when marking file descriptors inheritable on exec in the child process. This bug appears to have been introduced in 3.4.
- bpo-32647 [https://bugs.python.org/issue?@action=redirect&bpo=32647]: The ctypes module used to depend on indirect linking for dlopen. The shared extension is now explicitly linked against libdl on platforms with dl.
- bpo-32734 [https://bugs.python.org/issue?@action=redirect&bpo=32734]: Fixed `asyncio.Lock()` safety issue which allowed acquiring and locking the same lock multiple times, without it being free. Patch by Bar Harel.
- bpo-32727 [https://bugs.python.org/issue?@action=redirect&bpo=32727]: Do not include name field in SMTP envelope from address. Patch by Stéphane Wirtel
- bpo-27931 [https://bugs.python.org/issue?@action=redirect&bpo=27931]: Fix email address header parsing error when the username is an empty quoted string. Patch by Xiang Zhang.
- bpo-32304 [https://bugs.python.org/issue?@action=redirect&bpo=32304]: distutils’ upload command no longer corrupts tar files ending with a CR byte, and no longer tries to convert CR to CRLF in any of the upload text fields.
- bpo-32502 [https://bugs.python.org/issue?@action=redirect&bpo=32502]: uuid.uuid1 no longer raises an exception if a 64-bit hardware address is encountered.

- bpo-31848 [https://bugs.python.org/issue?@action=redirect&bpo=31848]: Fix the error handling in Aifc_read.initfp() when the SSND chunk is not found. Patch by Zackery Spytz.
- bpo-32555 [https://bugs.python.org/issue?@action=redirect&bpo=32555]: On FreeBSD and Solaris, os.strerror() now always decode the byte string from the current locale encoding, rather than using ASCII/surrogateescape in some cases.
- bpo-32521 [https://bugs.python.org/issue?@action=redirect&bpo=32521]: The nis module is now compatible with new libnsl and headers location.
- bpo-32473 [https://bugs.python.org/issue?@action=redirect&bpo=32473]: Improve ABCMeta._dump_registry() output readability
- bpo-32521 [https://bugs.python.org/issue?@action=redirect&bpo=32521]: glibc has removed Sun RPC. Use replacement libtirpc headers and library in nis module.
- bpo-32228 [https://bugs.python.org/issue?@action=redirect&bpo=32228]: Ensure that `truncate()` preserves the file position (as reported by `tell()`) after writes longer than the buffer size.
- bpo-26133 [https://bugs.python.org/issue?@action=redirect&bpo=26133]: Don’t unsubscribe signals in asyncio UNIX event loop on interpreter shutdown.
- bpo-32185 [https://bugs.python.org/issue?@action=redirect&bpo=32185]: The SSL module no longer sends IP addresses in SNI TLS extension on platforms with OpenSSL 1.0.2+ or inet_pton.
- bpo-32323 [https://bugs.python.org/issue?@action=redirect&bpo=32323]: **`urllib.parse.urlsplit()`** does not convert zone-id (scope) to lower case for scoped IPv6 addresses in hostnames now.
- bpo-32302 [https://bugs.python.org/issue?@action=redirect&bpo=32302]: Fix bdist_wininst of distutils for

CRT v142: it binary compatible with CRT v140.

- bpo-32255 [https://bugs.python.org/issue?@action=redirect&bpo=32255]: A single empty field is now always quoted when written into a CSV file. This allows to distinguish an empty row from a row consisting of a single empty field. Patch by Licht Takeuchi.
- bpo-32277 [https://bugs.python.org/issue?@action=redirect&bpo=32277]: Raise `NotImplementedError` instead of `SystemError` on platforms where `chmod(..., follow_symlinks=False)` is not supported. Patch by Anthony Sottile.
- bpo-32199 [https://bugs.python.org/issue?@action=redirect&bpo=32199]: The getnode() ip getter now uses ‘ip link’ instead of ‘ip link list’.
- bpo-27456 [https://bugs.python.org/issue?@action=redirect&bpo=27456]: Ensure TCP_NODELAY is set on Linux. Tests by Victor Stinner.
- bpo-31900 [https://bugs.python.org/issue?@action=redirect&bpo=31900]: The **`locale.localeconv()`** function now sets temporarily the `LC_CTYPE` locale to the `LC_NUMERIC` locale to decode `decimal_point` and `thousands_sep` byte strings if they are non-ASCII or longer than 1 byte, and the `LC_NUMERIC` locale is different than the `LC_CTYPE` locale. This temporary change affects other threads. Same change for the **`str.format()`** method when formatting a number (**`int`**, **`float`**, **`float`** and subclasses) with the `n` type (ex: `'{:n}'.format(1234)`).
- bpo-31802 [https://bugs.python.org/issue?@action=redirect&bpo=31802]: Importing native path module (`posixpath`, `ntpath`) now works even if the `os` module still is not imported.

Documentation

- bpo-17232 [https://bugs.python.org/issue?@action=redirect&bpo=17232]: Clarify docs for -O and -OO.

Patch by Terry Reedy.

- bpo-32800 [https://bugs.python.org/issue?@action=redirect&bpo=32800]: Update link to w3c doc for xml default namespaces.
- bpo-8722 [https://bugs.python.org/issue?@action=redirect&bpo=8722]: Document **`__getattr__()`** behavior when property **`get()`** method raises **`AttributeError`**.
- bpo-32614 [https://bugs.python.org/issue?@action=redirect&bpo=32614]: Modify RE examples in documentation to use raw strings to prevent **`DeprecationWarning`** and add text to REGEX HOWTO to highlight the deprecation.
- bpo-31972 [https://bugs.python.org/issue?@action=redirect&bpo=31972]: Improve docstrings for `pathlib.PurePath` subclasses.
- bpo-17799 [https://bugs.python.org/issue?@action=redirect&bpo=17799]: Explain real behaviour of sys.settrace and sys.setprofile and their C-API counterparts regarding which type of events are received in each function. Patch by Pablo Galindo Salgado.

Tests

- bpo-32517 [https://bugs.python.org/issue?@action=redirect&bpo=32517]: Fix failing `test_asyncio` on macOS 10.12.2+ due to transport of `KqueueSelector` loop was not being closed.
- bpo-32721 [https://bugs.python.org/issue?@action=redirect&bpo=32721]: Fix test_hashlib to not fail if the _md5 module is not built.
- bpo-32252 [https://bugs.python.org/issue?@action=redirect&bpo=32252]: Fix faulthandler_suppress_crash_report() used to prevent core dump files when testing crashes. getrlimit() returns zero on success.

- bpo-31518 [https://bugs.python.org/issue?@action=redirect&bpo=31518]: Debian Unstable has disabled TLS 1.0 and 1.1 for SSLv23_METHOD(). Change TLS/SSL protocol of some tests to PROTOCOL_TLS or PROTOCOL_TLSv1_2 to make them pass on Debian.

Build

- bpo-32635 [https://bugs.python.org/issue?@action=redirect&bpo=32635]: Fix segfault of the crypt module when libxcrypt is provided instead of libcrypt at the system.

Windows

- bpo-33016 [https://bugs.python.org/issue?@action=redirect&bpo=33016]: Fix potential use of uninitialized memory in nt._getfinalpathname
- bpo-32903 [https://bugs.python.org/issue?@action=redirect&bpo=32903]: Fix a memory leak in os.chdir() on Windows if the current directory is set to a UNC path.
- bpo-31966 [https://bugs.python.org/issue?@action=redirect&bpo=31966]: Fixed WindowsConsoleIO.write() for writing empty data.
- bpo-32409 [https://bugs.python.org/issue?@action=redirect&bpo=32409]: Ensures activate.bat can handle Unicode contents.
- bpo-32457 [https://bugs.python.org/issue?@action=redirect&bpo=32457]: Improves handling of denormalized executable path when launching Python.
- bpo-32370 [https://bugs.python.org/issue?@action=redirect&bpo=32370]: Use the correct encoding for ipconfig output in the uuid module. Patch by Segev Finer.
- bpo-29248 [https://bugs.python.org/issue?@action=redirect&bpo=29248]: Fix `os.readlink()` on Windows,

which was mistakenly treating the `PrintNameOffset` field of the reparse data buffer as a number of characters instead of bytes. Patch by Craig Holmquist and SSE4.

- bpo-32588 [https://bugs.python.org/issue?@action=redirect&bpo=32588]: Create standalone _distutils_findvs module.

macOS

- bpo-32726 [https://bugs.python.org/issue?@action=redirect&bpo=32726]: Provide an additional, more modern macOS installer variant that supports macOS 10.9+ systems in 64-bit mode only. Upgrade the supplied third-party libraries to OpenSSL 1.0.2n, XZ 5.2.3, and SQLite 3.22.0. The 10.9+ installer now links with and supplies its own copy of Tcl/Tk 8.6.8.

IDLE

- bpo-32984 [https://bugs.python.org/issue?@action=redirect&bpo=32984]: Set `__file__` while running a startup file. Like Python, IDLE optionally runs one startup file in the Shell window before presenting the first interactive input prompt. For IDLE, `-s` runs a file named in environmental variable **`IDLESTARTUP`** or **`PYTHONSTARTUP`**; `-r file` runs `file`. Python sets `__file__` to the startup file name before running the file and unsets it before the first prompt. IDLE now does the same when run normally, without the `-n` option.
- bpo-32940 [https://bugs.python.org/issue?@action=redirect&bpo=32940]: Simplify and rename StringTranslatePseudoMapping in pyparse.
- bpo-32916 [https://bugs.python.org/issue?@action=redirect&bpo=32916]: Change `str` to `code` in pyparse.
- bpo-32905 [https://bugs.python.org/issue?@action=redirect&bpo=32905]: Remove unused code in

pyparse module.

- bpo-32874 [https://bugs.python.org/issue?@action=redirect&bpo=32874]: Add tests for pyparse.
- bpo-32837 [https://bugs.python.org/issue?@action=redirect&bpo=32837]: Using the system and place-dependent default encoding for open() is a bad idea for IDLE’s system and location-independent files.
- bpo-32826 [https://bugs.python.org/issue?@action=redirect&bpo=32826]: Add “encoding=utf-8” to open() in IDLE’s test_help_about. GUI test test_file_buttons() only looks at initial ascii-only lines, but failed on systems where open() defaults to ‘ascii’ because readline() internally reads and decodes far enough ahead to encounter a non-ascii character in CREDITS.txt.
- bpo-32765 [https://bugs.python.org/issue?@action=redirect&bpo=32765]: Update configdialog General tab docstring to add new widgets to the widget list.

Tools/Demos

- bpo-24960 [https://bugs.python.org/issue?@action=redirect&bpo=24960]: 2to3 and lib2to3 can now read pickled grammar files using pkgutil.get_data() rather than probing the filesystem. This lets 2to3 and lib2to3 work when run from a zipfile.
- bpo-32222 [https://bugs.python.org/issue?@action=redirect&bpo=32222]: Fix pygettext not extracting docstrings for functions with type annotated arguments. Patch by Toby Harradine.

C API

- bpo-29084 [https://bugs.python.org/issue?@action=redirect&bpo=29084]: Undocumented C API for OrderedDict has been excluded from the limited C API. It was added by mistake and actually never worked in the limited C API.

Python 3.6.4 final

Release date: 2017-12-18

There were no new code changes in version 3.6.4 since v3.6.4rc1.

Python 3.6.4 release candidate 1

Release date: 2017-12-05

Core and Builtins

- bpo-32176 [https://bugs.python.org/issue?@action=redirect&bpo=32176]: co_flags.CO_NOFREE is now always set correctly by the code object constructor based on freevars and cellvars, rather than needing to be set correctly by the caller. This ensures it will be cleared automatically when additional cell references are injected into a modified code object and function.
- bpo-31949 [https://bugs.python.org/issue?@action=redirect&bpo=31949]: Fixed several issues in printing tracebacks (PyTraceBack_Print()). Setting sys.tracebacklimit to 0 or less now suppresses printing tracebacks. Setting sys.tracebacklimit to None now causes using the default limit. Setting sys.tracebacklimit to an integer larger than LONG_MAX now means using the limit LONG_MAX rather than the default limit. Fixed integer overflows in the case of more than `2**31` traceback items on Windows. Fixed output errors handling.

- bpo-30696 [https://bugs.python.org/issue?@action=redirect&bpo=30696]: Fix the interactive interpreter looping endlessly when no memory.
- bpo-20047 [https://bugs.python.org/issue?@action=redirect&bpo=20047]: Bytearray methods partition() and rpartition() now accept only bytes-like objects as separator, as documented. In particular they now raise TypeError rather of returning a bogus result when an integer is passed as a separator.
- bpo-31852 [https://bugs.python.org/issue?@action=redirect&bpo=31852]: Fix a segmentation fault caused by a combination of the async soft keyword and continuation lines.
- bpo-21720 [https://bugs.python.org/issue?@action=redirect&bpo=21720]: BytesWarning no longer emitted when the *fromlist* argument of `__import__()` or the `__all__` attribute of the module contain bytes instances.
- bpo-31825 [https://bugs.python.org/issue?@action=redirect&bpo=31825]: Fixed OverflowError in the ‘unicode-escape’ codec and in codecs.escape_decode() when decode an escaped non-ascii byte.
- bpo-28603 [https://bugs.python.org/issue?@action=redirect&bpo=28603]: Print the full context/cause chain of exceptions on interpreter exit, even if an exception in the chain is unhashable or compares equal to later ones. Patch by Zane Bitter.
- bpo-31786 [https://bugs.python.org/issue?@action=redirect&bpo=31786]: Fix timeout rounding in the select module to round correctly negative timeouts between -1.0 and 0.0. The functions now block waiting for events as expected. Previously, the call was incorrectly non-blocking. Patch by Pablo Galindo.
- bpo-31642 [https://bugs.python.org/issue?@action=redirect&bpo=31642]: Restored blocking “from

package import module” by setting sys.modules[“package.module”] to None.

- bpo-31626 [https://bugs.python.org/issue?@action=redirect&bpo=31626]: Fixed a bug in debug memory allocator. There was a write to freed memory after shrinking a memory block.
- bpo-31619 [https://bugs.python.org/issue?@action=redirect&bpo=31619]: Fixed a ValueError when convert a string with large number of underscores to integer with binary base.
- bpo-31592 [https://bugs.python.org/issue?@action=redirect&bpo=31592]: Fixed an assertion failure in Python parser in case of a bad `unicodedata.normalize()`. Patch by Oren Milman.
- bpo-31588 [https://bugs.python.org/issue?@action=redirect&bpo=31588]: Raise a `TypeError` with a helpful error message when class creation fails due to a metaclass with a bad `__prepare__()` method. Patch by Oren Milman.
- bpo-31566 [https://bugs.python.org/issue?@action=redirect&bpo=31566]: Fix an assertion failure in `_warnings.warn()` in case of a bad `__name__` global. Patch by Oren Milman.
- bpo-31505 [https://bugs.python.org/issue?@action=redirect&bpo=31505]: Fix an assertion failure in `json`, in case `_json.make_encoder()` received a bad `encoder()` argument. Patch by Oren Milman.
- bpo-31492 [https://bugs.python.org/issue?@action=redirect&bpo=31492]: Fix assertion failures in case of failing to import from a module with a bad `__name__` attribute, and in case of failing to access an attribute of such a module. Patch by Oren Milman.
- bpo-31490 [https://bugs.python.org/issue?@action=redirect&bpo=31490]: Fix an assertion failure in `ctypes` class definition, in case the class has an attribute whose

name is specified in `_anonymous_` but not in `_fields_`. Patch by Oren Milman.

- bpo-31478 [https://bugs.python.org/issue?@action=redirect&bpo=31478]: Fix an assertion failure in `_random.Random.seed()` in case the argument has a bad `__abs__()` method. Patch by Oren Milman.
- bpo-31315 [https://bugs.python.org/issue?@action=redirect&bpo=31315]: Fix an assertion failure in imp.create_dynamic(), when spec.name is not a string. Patch by Oren Milman.
- bpo-31311 [https://bugs.python.org/issue?@action=redirect&bpo=31311]: Fix a crash in the `__setstate__()` method of `ctypes._CData`, in case of a bad `__dict__`. Patch by Oren Milman.
- bpo-31293 [https://bugs.python.org/issue?@action=redirect&bpo=31293]: Fix crashes in true division and multiplication of a timedelta object by a float with a bad as_integer_ratio() method. Patch by Oren Milman.
- bpo-31285 [https://bugs.python.org/issue?@action=redirect&bpo=31285]: Fix an assertion failure in `warnings.warn_explicit`, when the return value of the received loader's `get_source()` has a bad `splitlines()` method. Patch by Oren Milman.
- bpo-30817 [https://bugs.python.org/issue?@action=redirect&bpo=30817]: `PyErr_PrintEx()` clears now the ignored exception that may be raised by `_PySys_SetObjectId()`, for example when no memory.

Library

- bpo-28556 [https://bugs.python.org/issue?@action=redirect&bpo=28556]: Two minor fixes for `typing` module: allow shallow copying instances of generic classes, improve interaction of `__init_subclass__` with generics. Original PRs by Ivan Levkivskyi.

- bpo-27240 [https://bugs.python.org/issue?@action=redirect&bpo=27240]: The header folding algorithm for the new email policies has been rewritten, which also fixes bpo-30788 [https://bugs.python.org/issue?@action=redirect&bpo=30788], bpo-31831 [https://bugs.python.org/issue?@action=redirect&bpo=31831], and bpo-32182 [https://bugs.python.org/issue?@action=redirect&bpo=32182]. In particular, RFC2231 folding is now done correctly.
- bpo-32186 [https://bugs.python.org/issue?@action=redirect&bpo=32186]: io.FileIO.readall() and io.FileIO.read() now release the GIL when getting the file size. Fixed hang of all threads with inaccessible NFS server. Patch by Nir Soffer.
- bpo-12239 [https://bugs.python.org/issue?@action=redirect&bpo=12239]: Make **`msilib.SummaryInformation.GetProperty()`** return `None` when the value of property is `VT_EMPTY`. Initial patch by Mark Mc Mahon.
- bpo-31325 [https://bugs.python.org/issue?@action=redirect&bpo=31325]: Fix wrong usage of **`collections.namedtuple()`** in the **`RobotFileParser.parse()`** method. Initial patch by Robin Wellner.
- bpo-12382 [https://bugs.python.org/issue?@action=redirect&bpo=12382]: **`msilib.OpenDatabase()`** now raises a better exception message when it couldn't open or create an MSI file. Initial patch by William Tisäter.
- bpo-32110 [https://bugs.python.org/issue?@action=redirect&bpo=32110]: `codecs.StreamReader.read(n)` now returns not more than *n* characters/bytes for non-negative *n*. This makes it compatible with `read()` methods of other file-like objects.
- bpo-32072 [https://bugs.python.org/issue?@action=redirect&bpo=32072]: Fixed issues with binary plists: Fixed saving bytearrays. Identical objects will be saved only once. Equal references will be load as identical

objects. Added support for saving and loading recursive data structures.

- bpo-32034 [https://bugs.python.org/issue?@action=redirect&bpo=32034]: Make asyncio.IncompleteReadError and LimitOverrunError pickleable.
- bpo-32015 [https://bugs.python.org/issue?@action=redirect&bpo=32015]: Fixed the looping of asyncio in the case of reconnection the socket during waiting async read/write from/to the socket.
- bpo-32011 [https://bugs.python.org/issue?@action=redirect&bpo=32011]: Restored support of loading marshal files with the TYPE_INT64 code. These files can be produced in Python 2.7.
- bpo-31970 [https://bugs.python.org/issue?@action=redirect&bpo=31970]: Reduce performance overhead of asyncio debug mode.
- bpo-9678 [https://bugs.python.org/issue?@action=redirect&bpo=9678]: Fixed determining the MAC address in the uuid module: Using ifconfig on NetBSD and OpenBSD. Using arp on Linux, FreeBSD, NetBSD and OpenBSD. Based on patch by Takayuki Shimizukawa.
- bpo-30057 [https://bugs.python.org/issue?@action=redirect&bpo=30057]: Fix potential missed signal in signal.signal().
- bpo-31933 [https://bugs.python.org/issue?@action=redirect&bpo=31933]: Fix Blake2 params leaf_size and node_offset on big endian platforms. Patch by Jack O’Connor.
- bpo-31927 [https://bugs.python.org/issue?@action=redirect&bpo=31927]: Fixed compilation of the socket module on NetBSD 8. Fixed assertion failure or reading arbitrary data when parse a AF_BLUETOOTH address on NetBSD and DragonFly BSD.
- bpo-27666 [https://bugs.python.org/issue?@action=redirect&bpo=27666]: Fixed stack corruption in

curses.box() and curses.ungetmouse() when the size of types chtype or mmask_t is less than the size of C long. curses.box() now accepts characters as arguments. Based on patch by Steve Fink.

- bpo-31897 [https://bugs.python.org/issue?@action=redirect&bpo=31897]: plistlib now catches more errors when read binary plists and raises InvalidFileException instead of unexpected exceptions.
- bpo-25720 [https://bugs.python.org/issue?@action=redirect&bpo=25720]: Fix the method for checking pad state of curses WINDOW. Patch by Masayuki Yamamoto.
- bpo-31893 [https://bugs.python.org/issue?@action=redirect&bpo=31893]: Fixed the layout of the kqueue_event structure on OpenBSD and NetBSD. Fixed the comparison of the kqueue_event objects.
- bpo-31891 [https://bugs.python.org/issue?@action=redirect&bpo=31891]: Fixed building the curses module on NetBSD.
- bpo-28416 [https://bugs.python.org/issue?@action=redirect&bpo=28416]: Instances of pickle.Pickler subclass with the persistent_id() method and pickle.Unpickler subclass with the persistent_load() method no longer create reference cycles.
- bpo-28326 [https://bugs.python.org/issue?@action=redirect&bpo=28326]: Fix multiprocessing.Process when stdout and/or stderr is closed or None.
- bpo-31457 [https://bugs.python.org/issue?@action=redirect&bpo=31457]: If nested log adapters are used, the inner `process()` methods are no longer omitted.
- bpo-31457 [https://bugs.python.org/issue?@action=redirect&bpo=31457]: The `manager` property on LoggerAdapter objects is now properly settable.
- bpo-31806 [https://bugs.python.org/issue?@action=redirect&bpo=31806]: Fix timeout rounding in time.sleep(), threading.Lock.acquire() and

socket.socket.settimeout() to round correctly negative timeouts between -1.0 and 0.0. The functions now block waiting for events as expected. Previously, the call was incorrectly non-blocking. Patch by Pablo Galindo.

- bpo-28603 [https://bugs.python.org/issue?@action=redirect&bpo=28603]: traceback: Fix a TypeError that occurred during printing of exception tracebacks when either the current exception or an exception in its context/cause chain is unhashable. Patch by Zane Bitter.
- bpo-30058 [https://bugs.python.org/issue?@action=redirect&bpo=30058]: Fixed buffer overflow in select.kqueue.control().
- bpo-31770 [https://bugs.python.org/issue?@action=redirect&bpo=31770]: Prevent a crash when calling the `__init__()` method of a `sqlite3.Cursor` object more than once. Patch by Oren Milman.
- bpo-31672 [https://bugs.python.org/issue?@action=redirect&bpo=31672]: `idpattern` in `string.Template` matched some non-ASCII characters. Now it uses `-i` regular expression local flag to avoid non-ASCII characters.
- bpo-31764 [https://bugs.python.org/issue?@action=redirect&bpo=31764]: Prevent a crash in `sqlite3.Cursor.close()` in case the `Cursor` object is uninitialized. Patch by Oren Milman.
- bpo-31752 [https://bugs.python.org/issue?@action=redirect&bpo=31752]: Fix possible crash in timedelta constructor called with custom integers.
- bpo-31701 [https://bugs.python.org/issue?@action=redirect&bpo=31701]: On Windows, faulthandler.enable() now ignores MSC and COM exceptions.
- bpo-31728 [https://bugs.python.org/issue?@action=redirect&bpo=31728]: Prevent crashes in `_elementtree` due to unsafe cleanup of `Element.text` and `Element.tail`. Patch by Oren Milman.

- bpo-31620 [https://bugs.python.org/issue?@action=redirect&bpo=31620]: an empty asyncio.Queue now doesn’t leak memory when queue.get pollers timeout
- bpo-31632 [https://bugs.python.org/issue?@action=redirect&bpo=31632]: Fix method set_protocol() of class _SSLProtocolTransport in asyncio module. This method was previously modifying a wrong reference to the protocol.
- bpo-31675 [https://bugs.python.org/issue?@action=redirect&bpo=31675]: Fixed memory leaks in Tkinter’s methods splitlist() and split() when pass a string larger than 2 GiB.
- bpo-31673 [https://bugs.python.org/issue?@action=redirect&bpo=31673]: Fixed typo in the name of Tkinter’s method adderrorinfo().
- bpo-30806 [https://bugs.python.org/issue?@action=redirect&bpo=30806]: Fix the string representation of a netrc object.
- bpo-15037 [https://bugs.python.org/issue?@action=redirect&bpo=15037]: Added a workaround for getkey() in curses for ncurses 5.7 and earlier.
- bpo-25351 [https://bugs.python.org/issue?@action=redirect&bpo=25351]: Avoid venv activate failures with undefined variables
- bpo-25532 [https://bugs.python.org/issue?@action=redirect&bpo=25532]: inspect.unwrap() will now only try to unwrap an object sys.getrecursionlimit() times, to protect against objects which create a new object on every attribute access.
- bpo-30347 [https://bugs.python.org/issue?@action=redirect&bpo=30347]: Stop crashes when concurrently iterate over itertools.groupby() iterators.
- bpo-31516 [https://bugs.python.org/issue?@action=redirect&bpo=31516]: `threading.current_thread()` should not return a dummy thread at shutdown.

- bpo-31351 [https://bugs.python.org/issue?@action=redirect&bpo=31351]: python -m ensurepip now exits with non-zero exit code if pip bootstrapping has failed.
- bpo-31482 [https://bugs.python.org/issue?@action=redirect&bpo=31482]: `random.seed()` now works with bytes in version=1
- bpo-31334 [https://bugs.python.org/issue?@action=redirect&bpo=31334]: Fix `poll.poll([timeout])` in the `select` module for arbitrary negative timeouts on all OSes where it can only be a non-negative integer or -1. Patch by Riccardo Coccioli.
- bpo-31310 [https://bugs.python.org/issue?@action=redirect&bpo=31310]: multiprocessing's semaphore tracker should be launched again if crashed.
- bpo-31308 [https://bugs.python.org/issue?@action=redirect&bpo=31308]: Make multiprocessing's forkserver process immune to Ctrl-C and other user interruptions. If it crashes, restart it when necessary.

Documentation

- bpo-32105 [https://bugs.python.org/issue?@action=redirect&bpo=32105]: Added asyncio.BaseEventLoop.connect_accepted_socket versionadded marker.
- bpo-31537 [https://bugs.python.org/issue?@action=redirect&bpo=31537]: Fix incorrect usage of `get_history_length` in readline documentation example code. Patch by Brad Smith.
- bpo-30085 [https://bugs.python.org/issue?@action=redirect&bpo=30085]: The operator functions without double underscores are preferred for clarity. The one with underscores are only kept for back-compatibility.

Tests

- bpo-31380 [https://bugs.python.org/issue?@action=redirect&bpo=31380]: Skip test_httpservers test_undecodable_file on macOS: fails on APFS.
- bpo-31705 [https://bugs.python.org/issue?@action=redirect&bpo=31705]: Skip test_socket.test_sha256() on Linux kernel older than 4.5. The test fails with ENOKEY on kernel 3.10 (on ppc64le). A fix was merged into the kernel 4.5.
- bpo-31174 [https://bugs.python.org/issue?@action=redirect&bpo=31174]: Fix test_tools.test_unparse: DirectoryTestCase now stores the names sample to always test the same files. It prevents false alarms when hunting reference leaks.
- bpo-30695 [https://bugs.python.org/issue?@action=redirect&bpo=30695]: Add the `set_nomemory(start, stop)` and `remove_mem_hooks()` functions to the `_testcapi` module.

Build

- bpo-32059 [https://bugs.python.org/issue?@action=redirect&bpo=32059]: `detect_modules()` in `setup.py` now also searches the sysroot paths when cross-compiling.
- bpo-31957 [https://bugs.python.org/issue?@action=redirect&bpo=31957]: Fixes Windows SDK version detection when building for Windows.
- bpo-31609 [https://bugs.python.org/issue?@action=redirect&bpo=31609]: Fixes quotes in PCbuild/clean.bat
- bpo-31934 [https://bugs.python.org/issue?@action=redirect&bpo=31934]: Abort the build when building out of a not clean source tree.
- bpo-31926 [https://bugs.python.org/issue?@action=redirect&bpo=31926]: Fixed Argument Clinic sometimes causing compilation errors when there was

more than one function and/or method in a .c file with the same name.
- bpo-28791 [https://bugs.python.org/issue?@action=redirect&bpo=28791]: Update Windows builds to use SQLite 3.21.0.
- bpo-28791 [https://bugs.python.org/issue?@action=redirect&bpo=28791]: Update OS X installer to use SQLite 3.21.0.
- bpo-22140 [https://bugs.python.org/issue?@action=redirect&bpo=22140]: Prevent double substitution of prefix in python-config.sh.
- bpo-31536 [https://bugs.python.org/issue?@action=redirect&bpo=31536]: Avoid wholesale rebuild after `make regen-all` if nothing changed.

Windows

- bpo-1102 [https://bugs.python.org/issue?@action=redirect&bpo=1102]: Return `None` when `View.Fetch()` returns `ERROR_NO_MORE_ITEMS` instead of raising `MSIError`. Initial patch by Anthony Tuininga.
- bpo-31944 [https://bugs.python.org/issue?@action=redirect&bpo=31944]: Fixes Modify button in Apps and Features dialog.

macOS

- bpo-31392 [https://bugs.python.org/issue?@action=redirect&bpo=31392]: Update macOS installer to use OpenSSL 1.0.2m

IDLE

- bpo-32207 [https://bugs.python.org/issue?@action=redirect&bpo=32207]: Improve tk event exception tracebacks in IDLE. When tk event handling is driven by

IDLE’s run loop, a confusing and distracting queue.EMPTY traceback context is no longer added to tk event exception tracebacks. The traceback is now the same as when event handling is driven by user code. Patch based on a suggestion by Serhiy Storchaka.

- bpo-32164 [https://bugs.python.org/issue?@action=redirect&bpo=32164]: Delete unused file idlelib/tabbedpages.py. Use of TabbedPageSet in configdialog was replaced by ttk.Notebook.
- bpo-32100 [https://bugs.python.org/issue?@action=redirect&bpo=32100]: IDLE: Fix old and new bugs in pathbrowser; improve tests. Patch mostly by Cheryl Sabella.
- bpo-31858 [https://bugs.python.org/issue?@action=redirect&bpo=31858]: IDLE – Restrict shell prompt manipulation to the shell. Editor and output windows only see an empty last prompt line. This simplifies the code and fixes a minor bug when newline is inserted. Sys.ps1, if present, is read on Shell start-up, but is not set or changed.
- bpo-31860 [https://bugs.python.org/issue?@action=redirect&bpo=31860]: The font sample in the IDLE configuration dialog is now editable. Changes persist while IDLE remains open
- bpo-31836 [https://bugs.python.org/issue?@action=redirect&bpo=31836]: Test_code_module now passes if run after test_idle, which sets ps1. The code module uses sys.ps1 if present or sets it to ‘>>> ‘ if not. Test_code_module now properly tests both behaviors. Ditto for ps2.
- bpo-28603 [https://bugs.python.org/issue?@action=redirect&bpo=28603]: Fix a TypeError that caused a shell restart when printing a traceback that includes an exception that is unhashable. Patch by Zane Bitter.
- bpo-13802 [https://bugs.python.org/issue?@action=redirect&bpo=13802]: Use non-Latin characters in the

IDLE's Font settings sample. Even if one selects a font that defines a limited subset of the unicode Basic Multilingual Plane, tcl/tk will use other fonts that define a character. The expanded example give users of non-Latin characters a better idea of what they might see in IDLE's shell and editors. To make room for the expanded sample, frames on the Font tab are re-arranged. The Font/Tabs help explains a bit about the additions.

- bpo-31460 [https://bugs.python.org/issue?@action=redirect&bpo=31460]: Simplify the API of IDLE's Module Browser. Passing a widget instead of an flist with a root widget opens the option of creating a browser frame that is only part of a window. Passing a full file name instead of pieces assumed to come from a .py file opens the possibility of browsing python files that do not end in .py.
- bpo-31649 [https://bugs.python.org/issue?@action=redirect&bpo=31649]: IDLE - Make _htest, _utest parameters keyword only.
- bpo-31559 [https://bugs.python.org/issue?@action=redirect&bpo=31559]: Remove test order dependence in idle_test.test_browser.
- bpo-31459 [https://bugs.python.org/issue?@action=redirect&bpo=31459]: Rename IDLE's module browser from Class Browser to Module Browser. The original module-level class and method browser became a module browser, with the addition of module-level functions, years ago. Nested classes and functions were added yesterday. For back-compatibility, the virtual event <<open-class-browser>>, which appears on the Keys tab of the Settings dialog, is not changed. Patch by Cheryl Sabella.
- bpo-31500 [https://bugs.python.org/issue?@action=redirect&bpo=31500]: Default fonts now are scaled on HiDPI displays.

- bpo-1612262 [https://bugs.python.org/issue?@action=redirect&bpo=1612262]: IDLE module browser now shows nested classes and functions. Original patches for code and tests by Guilherme Polo and Cheryl Sabella, respectively.

Tools/Demos

- bpo-30722 [https://bugs.python.org/issue?@action=redirect&bpo=30722]: Make redemo work with Python 3.6 and newer versions. Also, remove the `LOCALE` option since it doesn’t work with string patterns in Python 3. Patch by Christoph Sarnowski.

C API

- bpo-20891 [https://bugs.python.org/issue?@action=redirect&bpo=20891]: Fix PyGILState_Ensure(). When PyGILState_Ensure() is called in a non-Python thread before PyEval_InitThreads(), only call PyEval_InitThreads() after calling PyThreadState_New() to fix a crash.
- bpo-31532 [https://bugs.python.org/issue?@action=redirect&bpo=31532]: Fix memory corruption due to allocator mix in getpath.c between Py_GetPath() and Py_SetPath()
- bpo-30697 [https://bugs.python.org/issue?@action=redirect&bpo=30697]: The `PyExc_RecursionErrorInst` singleton is removed and `PyErr_NormalizeException()` does not use it anymore. This singleton is persistent and its members being never cleared may cause a segfault during finalization of the interpreter. See also bpo-22898 [https://bugs.python.org/issue?@action=redirect&bpo=22898].

Python 3.6.3 final

Release date: 2017-10-03

Library

- bpo-31641 [https://bugs.python.org/issue?@action=redirect&bpo=31641]: Re-allow arbitrary iterables in `concurrent.futures.as_completed()`. Fixes regression in 3.6.3rc1.

Build

- bpo-31662 [https://bugs.python.org/issue?@action=redirect&bpo=31662]: Fix typos in Windows `uploadrelease.bat` script. Fix Windows Doc build issues in `Doc/make.bat`.
- bpo-31423 [https://bugs.python.org/issue?@action=redirect&bpo=31423]: Fix building the PDF documentation with newer versions of Sphinx.

Python 3.6.3 release candidate 1

Release date: 2017-09-18

Security

- bpo-29781 [https://bugs.python.org/issue?@action=redirect&bpo=29781]: SSLObject.version() now correctly returns None when handshake over BIO has not been performed yet.
- bpo-30947 [https://bugs.python.org/issue?@action=redirect&bpo=30947]: Upgrade libexpat embedded copy from version 2.2.1 to 2.2.3 to get security fixes.

Core and Builtins

- bpo-31471 [https://bugs.python.org/issue?@action=redirect&bpo=31471]: Fix an assertion failure in `subprocess.Popen()` on Windows, in case the env argument has a bad `keys()` method. Patch by Oren Milman.
- bpo-31418 [https://bugs.python.org/issue?@action=redirect&bpo=31418]: Fix an assertion failure in `PyErr_WriteUnraisable()` in case of an exception with a bad `__module__` attribute. Patch by Oren Milman.
- bpo-31416 [https://bugs.python.org/issue?@action=redirect&bpo=31416]: Fix assertion failures in case of a bad warnings.filters or warnings.defaultaction. Patch by Oren Milman.
- bpo-31411 [https://bugs.python.org/issue?@action=redirect&bpo=31411]: Raise a TypeError instead of SystemError in case warnings.onceregistry is not a dictionary. Patch by Oren Milman.
- bpo-31373 [https://bugs.python.org/issue?@action=redirect&bpo=31373]: Fix several possible instances of undefined behavior due to floating-point demotions.
- bpo-30465 [https://bugs.python.org/issue?@action=redirect&bpo=30465]: Location information (`lineno` and `col_offset`) in f-strings is now (mostly) correct. This fixes tools like flake8 from showing warnings on the wrong line (typically the first line of the file).
- bpo-31343 [https://bugs.python.org/issue?@action=redirect&bpo=31343]: Include sys/sysmacros.h for major(), minor(), and makedev(). GNU C library plans to remove the functions from sys/types.h.
- bpo-31291 [https://bugs.python.org/issue?@action=redirect&bpo=31291]: Fix an assertion failure in `zipimport.zipimporter.get_data` on Windows, when the return value of `pathname.replace('/','\\')` isn’t a string. Patch by Oren Milman.
- bpo-31271 [https://bugs.python.org/issue?@action=redirect&bpo=31271]: Fix an assertion failure in the

`write()` method of `io.TextIOWrapper`, when the encoder doesn’t return a bytes object. Patch by Oren Milman.

- bpo-31243 [https://bugs.python.org/issue?@action=redirect&bpo=31243]: Fix a crash in some methods of `io.TextIOWrapper`, when the decoder’s state is invalid. Patch by Oren Milman.
- bpo-30721 [https://bugs.python.org/issue?@action=redirect&bpo=30721]: `print` now shows correct usage hint for using Python 2 redirection syntax. Patch by Sanyam Khurana.
- bpo-31070 [https://bugs.python.org/issue?@action=redirect&bpo=31070]: Fix a race condition in importlib _get_module_lock().
- bpo-31095 [https://bugs.python.org/issue?@action=redirect&bpo=31095]: Fix potential crash during GC caused by `tp_dealloc` which doesn’t call `PyObject_GC_UnTrack()`.
- bpo-31071 [https://bugs.python.org/issue?@action=redirect&bpo=31071]: Avoid masking original TypeError in call with * unpacking when other arguments are passed.
- bpo-30978 [https://bugs.python.org/issue?@action=redirect&bpo=30978]: str.format_map() now passes key lookup exceptions through. Previously any exception was replaced with a KeyError exception.
- bpo-30808 [https://bugs.python.org/issue?@action=redirect&bpo=30808]: Use _Py_atomic API for concurrency-sensitive signal state.
- bpo-30876 [https://bugs.python.org/issue?@action=redirect&bpo=30876]: Relative import from unloaded package now reimports the package instead of failing with SystemError. Relative import from non-package now fails with ImportError rather than SystemError.
- bpo-30703 [https://bugs.python.org/issue?@action=redirect&bpo=30703]: Improve signal delivery. Avoid using Py_AddPendingCall from signal handler, to avoid

calling signal-unsafe functions. The tests I'm adding here fail without the rest of the patch, on Linux and OS X. This means our signal delivery logic had defects (some signals could be lost).

- bpo-30765 [https://bugs.python.org/issue?@action=redirect&bpo=30765]: Avoid blocking in pthread_mutex_lock() when PyThread_acquire_lock() is asked not to block.
- bpo-31161 [https://bugs.python.org/issue?@action=redirect&bpo=31161]: Make sure the 'Missing parentheses' syntax error message is only applied to SyntaxError, not to subclasses. Patch by Martijn Pieters.
- bpo-30814 [https://bugs.python.org/issue?@action=redirect&bpo=30814]: Fixed a race condition when import a submodule from a package.
- bpo-30597 [https://bugs.python.org/issue?@action=redirect&bpo=30597]: `print` now shows expected input in custom error message when used as a Python 2 statement. Patch by Sanyam Khurana.

Library

- bpo-31499 [https://bugs.python.org/issue?@action=redirect&bpo=31499]: xml.etree: Fix a crash when a parser is part of a reference cycle.
- bpo-28556 [https://bugs.python.org/issue?@action=redirect&bpo=28556]: typing.get_type_hints now finds the right globalns for classes and modules by default (when no `globalns` was specified by the caller).
- bpo-28556 [https://bugs.python.org/issue?@action=redirect&bpo=28556]: Speed improvements to the `typing` module. Original PRs by Ivan Levkivskyi and Mitar.
- bpo-31544 [https://bugs.python.org/issue?@action=redirect&bpo=31544]: The C accelerator module of ElementTree ignored exceptions raised when looking up TreeBuilder target methods in XMLParser().

- bpo-31234 [https://bugs.python.org/issue?@action=redirect&bpo=31234]: socket.create_connection() now fixes manually a reference cycle: clear the variable storing the last exception on success.
- bpo-31457 [https://bugs.python.org/issue?@action=redirect&bpo=31457]: LoggerAdapter objects can now be nested.
- bpo-31400 [https://bugs.python.org/issue?@action=redirect&bpo=31400]: Improves SSL error handling to avoid losing error numbers.
- bpo-28958 [https://bugs.python.org/issue?@action=redirect&bpo=28958]: ssl.SSLContext() now uses OpenSSL error information when a context cannot be instantiated.
- bpo-27340 [https://bugs.python.org/issue?@action=redirect&bpo=27340]: SSLSocket.sendall() now uses memoryview to create slices of data. This fixes support for all bytes-like object. It is also more efficient and avoids costly copies.
- bpo-31178 [https://bugs.python.org/issue?@action=redirect&bpo=31178]: Fix string concatenation bug in rare error path in the subprocess module
- bpo-31350 [https://bugs.python.org/issue?@action=redirect&bpo=31350]: Micro-optimize **`asyncio._get_running_loop()`** to become up to 10% faster.
- bpo-31170 [https://bugs.python.org/issue?@action=redirect&bpo=31170]: expat: Update libexpat from 2.2.3 to 2.2.4. Fix copying of partial characters for UTF-8 input (libexpat bug 115): https://github.com/libexpat/libexpat/issues/115
- bpo-29136 [https://bugs.python.org/issue?@action=redirect&bpo=29136]: Add TLS 1.3 cipher suites and OP_NO_TLSv1_3.
- bpo-29212 [https://bugs.python.org/issue?@action=redirect&bpo=29212]: Fix concurrent.futures.thread.ThreadPoolExecutor threads

to have a non repr() based thread name by default when no thread_name_prefix is supplied. They will now identify themselves as “ThreadPoolExecutor-y_n”.

- bpo-9146 [https://bugs.python.org/issue?@action=redirect&bpo=9146]: Fix a segmentation fault in _hashopenssl when standard hash functions such as md5 are not available in the linked OpenSSL library. As in some special FIPS-140 build environments.
- bpo-27144 [https://bugs.python.org/issue?@action=redirect&bpo=27144]: The `map()` and `as_completed()` iterators in `concurrent.futures` now avoid keeping a reference to yielded objects.
- bpo-10746 [https://bugs.python.org/issue?@action=redirect&bpo=10746]: Fix ctypes producing wrong **PEP 3118** [https://peps.python.org/pep-3118/] type codes for integer types.
- bpo-22536 [https://bugs.python.org/issue?@action=redirect&bpo=22536]: The subprocess module now sets the filename when FileNotFoundError is raised on POSIX systems due to the executable or cwd not being found.
- bpo-31249 [https://bugs.python.org/issue?@action=redirect&bpo=31249]: concurrent.futures: WorkItem.run() used by ThreadPoolExecutor now breaks a reference cycle between an exception object and the WorkItem object.
- bpo-31247 [https://bugs.python.org/issue?@action=redirect&bpo=31247]: xmlrpc.server now explicitly breaks reference cycles when using sys.exc_info() in code handling exceptions.
- bpo-30102 [https://bugs.python.org/issue?@action=redirect&bpo=30102]: The ssl and hashlib modules now call OPENSSL_add_all_algorithms_noconf() on OpenSSL < 1.1.0. The function detects CPU features and enables optimizations on some CPU architectures such

as POWER8. Patch is based on research from Gustavo Serra Scalet.

- bpo-31185 [https://bugs.python.org/issue?@action=redirect&bpo=31185]: Fixed miscellaneous errors in asyncio speedup module.
- bpo-31135 [https://bugs.python.org/issue?@action=redirect&bpo=31135]: ttk: fix the destroy() method of LabeledScale and OptionMenu classes. Call the parent destroy() method even if the used attribute doesn’t exist. The LabeledScale.destroy() method now also explicitly clears label and scale attributes to help the garbage collector to destroy all widgets.
- bpo-31107 [https://bugs.python.org/issue?@action=redirect&bpo=31107]: Fix `copyreg._slotnames()` mangled attribute calculation for classes whose name begins with an underscore. Patch by Shane Harvey.
- bpo-31061 [https://bugs.python.org/issue?@action=redirect&bpo=31061]: Fixed a crash when using asyncio and threads.
- bpo-30502 [https://bugs.python.org/issue?@action=redirect&bpo=30502]: Fix handling of long oids in ssl. Based on patch by Christian Heimes.
- bpo-30119 [https://bugs.python.org/issue?@action=redirect&bpo=30119]: ftplib.FTP.putline() now throws ValueError on commands that contains CR or LF. Patch by Donghee Na.
- bpo-30595 [https://bugs.python.org/issue?@action=redirect&bpo=30595]: multiprocessing.Queue.get() with a timeout now polls its reader in non-blocking mode if it succeeded to acquire the lock but the acquire took longer than the timeout.
- bpo-29403 [https://bugs.python.org/issue?@action=redirect&bpo=29403]: Fix `unittest.mock`’s autospec to not fail on method-bound builtin functions. Patch by Aaron Gallagher.

- bpo-30961 [https://bugs.python.org/issue?@action=redirect&bpo=30961]: Fix decrementing a borrowed reference in tracemalloc.
- bpo-25684 [https://bugs.python.org/issue?@action=redirect&bpo=25684]: Change `ttk.OptionMenu` radiobuttons to be unique across instances of `OptionMenu`.
- bpo-30886 [https://bugs.python.org/issue?@action=redirect&bpo=30886]: Fix multiprocessing.Queue.join_thread(): it now waits until the thread completes, even if the thread was started by the same process which created the queue.
- bpo-29854 [https://bugs.python.org/issue?@action=redirect&bpo=29854]: Fix segfault in readline when using readline's history-size option. Patch by Nir Soffer.
- bpo-30319 [https://bugs.python.org/issue?@action=redirect&bpo=30319]: socket.close() now ignores ECONNRESET error.
- bpo-30828 [https://bugs.python.org/issue?@action=redirect&bpo=30828]: Fix out of bounds write in `asyncio.CFuture.remove_done_callback()`.
- bpo-30807 [https://bugs.python.org/issue?@action=redirect&bpo=30807]: signal.setitimer() may disable the timer when passed a tiny value. Tiny values (such as 1e-6) are valid non-zero values for setitimer(), which is specified as taking microsecond-resolution intervals. However, on some platform, our conversion routine could convert 1e-6 into a zero interval, therefore disabling the timer instead of (re-)scheduling it.
- bpo-30441 [https://bugs.python.org/issue?@action=redirect&bpo=30441]: Fix bug when modifying os.environ while iterating over it
- bpo-30532 [https://bugs.python.org/issue?@action=redirect&bpo=30532]: Fix email header value parser dropping folding white space in certain cases.
- bpo-30879 [https://bugs.python.org/issue?@action=redirect&bpo=30879]: os.listdir() and os.scandir() now

emit bytes names when called with bytes-like argument.

- bpo-30746 [https://bugs.python.org/issue?@action=redirect&bpo=30746]: Prohibited the ‘=’ character in environment variable names in `os.putenv()` and `os.spawn*()`.
- bpo-29755 [https://bugs.python.org/issue?@action=redirect&bpo=29755]: Fixed the lgettext() family of functions in the gettext module. They now always return bytes.

Documentation

- bpo-31294 [https://bugs.python.org/issue?@action=redirect&bpo=31294]: Fix incomplete code snippet in the ZeroMQSocketListener and ZeroMQSocketHandler examples and adapt them to Python 3.
- bpo-21649 [https://bugs.python.org/issue?@action=redirect&bpo=21649]: Add RFC 7525 and Mozilla server side TLS links to SSL documentation.
- bpo-30803 [https://bugs.python.org/issue?@action=redirect&bpo=30803]: Clarify doc on truth value testing. Original patch by Peter Thomassen.

Tests

- bpo-31320 [https://bugs.python.org/issue?@action=redirect&bpo=31320]: Silence traceback in test_ssl
- bpo-25674 [https://bugs.python.org/issue?@action=redirect&bpo=25674]: Remove sha256.tbs-internet.com ssl test
- bpo-30715 [https://bugs.python.org/issue?@action=redirect&bpo=30715]: Address ALPN callback changes for OpenSSL 1.1.0f. The latest version behaves like OpenSSL 1.0.2 and no longer aborts handshake.
- bpo-30822 [https://bugs.python.org/issue?@action=redirect&bpo=30822]: regrtest: Exclude tzdata from

regrtest –all. When running the test suite using –use=all / -u all, exclude tzdata since it makes test_datetime too slow (15-20 min on some buildbots) which then times out on some buildbots. Fix also regrtest command line parser to allow passing -u extralargefile to run test_zipfile64.

Build

- bpo-30854 [https://bugs.python.org/issue?@action=redirect&bpo=30854]: Fix compile error when compiling –without-threads. Patch by Masayuki Yamamoto.

Windows

- bpo-30389 [https://bugs.python.org/issue?@action=redirect&bpo=30389]: Adds detection of Visual Studio 2017 to distutils on Windows.
- bpo-31340 [https://bugs.python.org/issue?@action=redirect&bpo=31340]: Change to building with MSVC v141 (included with Visual Studio 2017)
- bpo-30581 [https://bugs.python.org/issue?@action=redirect&bpo=30581]: os.cpu_count() now returns the correct number of processors on Windows when the number of logical processors is greater than 64.
- bpo-30731 [https://bugs.python.org/issue?@action=redirect&bpo=30731]: Add a missing xmlns to python.manifest so that it matches the schema.

IDLE

- bpo-31493 [https://bugs.python.org/issue?@action=redirect&bpo=31493]: IDLE code context – fix code update and font update timers. Canceling timers prevents a warning message when test_idle completes.

- bpo-31488 [https://bugs.python.org/issue?@action=redirect&bpo=31488]: IDLE - Update non-key options in former extension classes. When applying configdialog changes, call .reload for each feature class. Change ParenMatch so updated options affect existing instances attached to existing editor windows.
- bpo-31477 [https://bugs.python.org/issue?@action=redirect&bpo=31477]: IDLE - Improve rstrip entry in doc. Strip trailing whitespace strips more than blank spaces. Multiline string literals are not skipped.
- bpo-31480 [https://bugs.python.org/issue?@action=redirect&bpo=31480]: IDLE - make tests pass with zzdummy extension disabled by default.
- bpo-31421 [https://bugs.python.org/issue?@action=redirect&bpo=31421]: Document how IDLE runs tkinter programs. IDLE calls tcl/tk update in the background in order to make live interaction and experimentation with tkinter applications much easier.
- bpo-31414 [https://bugs.python.org/issue?@action=redirect&bpo=31414]: IDLE – fix tk entry box tests by deleting first. Adding to an int entry is not the same as deleting and inserting because int(‘’) will fail.
- bpo-31051 [https://bugs.python.org/issue?@action=redirect&bpo=31051]: Rearrange IDLE configdialog GenPage into Window, Editor, and Help sections.
- bpo-30617 [https://bugs.python.org/issue?@action=redirect&bpo=30617]: IDLE - Add docstrings and tests for outwin subclass of editor. Move some data and functions from the class to module level. Patch by Cheryl Sabella.
- bpo-31287 [https://bugs.python.org/issue?@action=redirect&bpo=31287]: IDLE - Do not modify tkinter.message in test_configdialog.
- bpo-27099 [https://bugs.python.org/issue?@action=redirect&bpo=27099]: Convert IDLE’s built-in ‘extensions’ to regular features. About 10 IDLE features

were implemented as supposedly optional extensions. Their different behavior could be confusing or worse for users and not good for maintenance. Hence the conversion. The main difference for users is that user configurable key bindings for builtin features are now handled uniformly. Now, editing a binding in a keyset only affects its value in the keyset. All bindings are defined together in the system-specific default keysets in config-extensions.def. All custom keysets are saved as a whole in config-extension.cfg. All take effect as soon as one clicks Apply or Ok. The affected events are ‘<<force-open-completions>>’, ‘<<expand-word>>’, ‘<<force-open-calltip>>’, ‘<<flash-paren>>’, ‘<<format-paragraph>>’, ‘<<run-module>>’, ‘<<check-module>>’, and ‘<<zoom-height>>’. Any (global) customizations made before 3.6.3 will not affect their keyset-specific customization after 3.6.3. and vice versa. Initial patch by Charles Wohlganger.

- bpo-31206 [https://bugs.python.org/issue?@action=redirect&bpo=31206]: IDLE: Factor HighPage(Frame) class from ConfigDialog. Patch by Cheryl Sabella.
- bpo-31001 [https://bugs.python.org/issue?@action=redirect&bpo=31001]: Add tests for configdialog highlight tab. Patch by Cheryl Sabella.
- bpo-31205 [https://bugs.python.org/issue?@action=redirect&bpo=31205]: IDLE: Factor KeysPage(Frame) class from ConfigDialog. The slightly modified tests continue to pass. Patch by Cheryl Sabella.
- bpo-31130 [https://bugs.python.org/issue?@action=redirect&bpo=31130]: IDLE – stop leaks in test_configdialog. Initial patch by Victor Stinner.
- bpo-31002 [https://bugs.python.org/issue?@action=redirect&bpo=31002]: Add tests for configdialog keys tab. Patch by Cheryl Sabella.
- bpo-19903 [https://bugs.python.org/issue?@action=redirect&bpo=19903]: IDLE: Calltips use

`inspect.signature` instead of `inspect.getfullargspec`. This improves calltips for builtins converted to use Argument Clinic. Patch by Louie Lu.

- bpo-31083 [https://bugs.python.org/issue?@action=redirect&bpo=31083]: IDLE - Add an outline of a TabPage class in configdialog. Update existing classes to match outline. Initial patch by Cheryl Sabella.
- bpo-31050 [https://bugs.python.org/issue?@action=redirect&bpo=31050]: Factor GenPage(Frame) class from ConfigDialog. The slightly modified tests continue to pass. Patch by Cheryl Sabella.
- bpo-31004 [https://bugs.python.org/issue?@action=redirect&bpo=31004]: IDLE - Factor FontPage(Frame) class from ConfigDialog. Slightly modified tests continue to pass. Fix General tests. Patch mostly by Cheryl Sabella.
- bpo-30781 [https://bugs.python.org/issue?@action=redirect&bpo=30781]: IDLE - Use ttk widgets in ConfigDialog. Patches by Terry Jan Reedy and Cheryl Sabella.
- bpo-31060 [https://bugs.python.org/issue?@action=redirect&bpo=31060]: IDLE - Finish rearranging methods of ConfigDialog Grouping methods pertaining to each tab and the buttons will aid writing tests and improving the tabs and will enable splitting the groups into classes.
- bpo-30853 [https://bugs.python.org/issue?@action=redirect&bpo=30853]: IDLE – Factor a VarTrace class out of ConfigDialog. Instance tracers manages pairs consisting of a tk variable and a callback function. When tracing is turned on, setting the variable calls the function. Test coverage for the new class is 100%.
- bpo-31003 [https://bugs.python.org/issue?@action=redirect&bpo=31003]: IDLE: Add more tests for General tab.

- bpo-30993 [https://bugs.python.org/issue?@action=redirect&bpo=30993]: IDLE - Improve configdialog font page and tests. In configdialog: Document causal pathways in create_font_tab docstring. Simplify some attribute names. Move set_samples calls to var_changed_font (idea from Cheryl Sabella). Move related functions to positions after the create widgets function. In test_configdialog: Fix test_font_set so not order dependent. Fix renamed test_indent_scale so it tests the widget. Adjust tests for movement of set_samples call. Add tests for load functions. Put all font tests in one class and tab indent tests in another. Except for two lines, these tests completely cover the related functions.
- bpo-30981 [https://bugs.python.org/issue?@action=redirect&bpo=30981]: IDLE – Add more configdialog font page tests.
- bpo-28523 [https://bugs.python.org/issue?@action=redirect&bpo=28523]: IDLE: replace ‘colour’ with ‘color’ in configdialog.
- bpo-30917 [https://bugs.python.org/issue?@action=redirect&bpo=30917]: Add tests for idlelib.config.IdleConf. Increase coverage from 46% to 96%. Patch by Louie Lu.
- bpo-30934 [https://bugs.python.org/issue?@action=redirect&bpo=30934]: Document coverage details for idlelib tests. Add section to idlelib/idle-test/README.txt. Include check that branches are taken both ways. Exclude IDLE-specific code that does not run during unit tests.
- bpo-30913 [https://bugs.python.org/issue?@action=redirect&bpo=30913]: IDLE: Document ConfigDialog tk Vars, methods, and widgets in docstrings This will facilitate improving the dialog and splitting up the class. Original patch by Cheryl Sabella.

- bpo-30899 [https://bugs.python.org/issue?@action=redirect&bpo=30899]: IDLE: Add tests for ConfigParser subclasses in config. Patch by Louie Lu.
- bpo-30881 [https://bugs.python.org/issue?@action=redirect&bpo=30881]: IDLE: Add docstrings to browser.py. Patch by Cheryl Sabella.
- bpo-30851 [https://bugs.python.org/issue?@action=redirect&bpo=30851]: IDLE: Remove unused variables in configdialog. One is a duplicate, one is set but cannot be altered by users. Patch by Cheryl Sabella.
- bpo-30870 [https://bugs.python.org/issue?@action=redirect&bpo=30870]: IDLE: In Settings dialog, select font with Up, Down keys as well as mouse. Initial patch by Louie Lu.
- bpo-8231 [https://bugs.python.org/issue?@action=redirect&bpo=8231]: IDLE: call config.IdleConf.GetUserCfgDir only once.
- bpo-30779 [https://bugs.python.org/issue?@action=redirect&bpo=30779]: IDLE: Factor ConfigChanges class from configdialog, put in config; test. * In config, put dump test code in a function; run it and unittest in ‘if __name__ == ‘__main__’. * Add class config.ConfigChanges based on changes_class_v4.py on bpo issue. * Add class test_config.ChangesTest, partly using configdialog_tests_v1.py. * Revise configdialog to use ConfigChanges; see tracker msg297804. * Revise test_configdialog to match configdialog changes. * Remove configdialog functions unused or moved to ConfigChanges. Cheryl Sabella contributed parts of the patch.
- bpo-30777 [https://bugs.python.org/issue?@action=redirect&bpo=30777]: IDLE: configdialog - Add docstrings and fix comments. Patch by Cheryl Sabella.
- bpo-30495 [https://bugs.python.org/issue?@action=redirect&bpo=30495]: IDLE: Improve textview with docstrings, PEP8 names, and more tests. Patch by Cheryl Sabella.

- bpo-30723 [https://bugs.python.org/issue?@action=redirect&bpo=30723]: IDLE: Make several improvements to parenmatch. Add ‘parens’ style to highlight both opener and closer. Make ‘default’ style, which is not default, a synonym for ‘opener’. Make time-delay work the same with all styles. Add help for config dialog extensions tab, including help for parenmatch. Add new tests. Original patch by Charles Wohlganger.
- bpo-30674 [https://bugs.python.org/issue?@action=redirect&bpo=30674]: IDLE: add docstrings to grep module. Patch by Cheryl Sabella
- bpo-21519 [https://bugs.python.org/issue?@action=redirect&bpo=21519]: IDLE’s basic custom key entry dialog now detects duplicates properly. Original patch by Saimadhav Heblikar.
- bpo-29910 [https://bugs.python.org/issue?@action=redirect&bpo=29910]: IDLE no longer deletes a character after commenting out a region by a key shortcut. Add `return 'break'` for this and other potential conflicts between IDLE and default key bindings.
- bpo-30728 [https://bugs.python.org/issue?@action=redirect&bpo=30728]: Review and change idlelib.configdialog names. Lowercase method and attribute names. Replace ‘colour’ with ‘color’, expand overly cryptic names, delete unneeded underscores. Replace `import *` with specific imports. Patches by Cheryl Sabella.
- bpo-6739 [https://bugs.python.org/issue?@action=redirect&bpo=6739]: IDLE: Verify user-entered key sequences by trying to bind them with tk. Add tests for all 3 validation functions. Original patch by G Polo. Tests added by Cheryl Sabella.

Tools/Demos

- bpo-30983 [https://bugs.python.org/issue?@action=redirect&bpo=30983]: gdb integration commands (py-bt, etc.) work on optimized shared builds now, too. **PEP 523** [https://peps.python.org/pep-0523/] introduced _PyEval_EvalFrameDefault which inlines PyEval_EvalFrameEx on non-debug shared builds. This broke the ability to use py-bt, py-up, and a few other Python-specific gdb integrations. The problem is fixed by only looking for _PyEval_EvalFrameDefault frames in python-gdb.py. Original patch by Bruno “Polaco” Penteado.

Python 3.6.2 final

Release date: 2017-07-17

No changes since release candidate 2

Python 3.6.2 release candidate 2

Release date: 2017-07-07

Security

- bpo-30730 [https://bugs.python.org/issue?@action=redirect&bpo=30730]: Prevent environment variables injection in subprocess on Windows. Prevent passing other environment variables and command arguments.
- bpo-30694 [https://bugs.python.org/issue?@action=redirect&bpo=30694]: Upgrade expat copy from 2.2.0 to 2.2.1 to get fixes of multiple security vulnerabilities including: **CVE 2017-9233** [https://www.cve.org/CVERecord?id=CVE-2017-9233] (External entity infinite loop DoS), **CVE 2016-9063** [https://www.cve.org/CVERecord?id=CVE-2016-9063] (Integer overflow, re-fix), **CVE 2016-0718**

[https://www.cve.org/CVERecord?id=CVE-2016-0718] (Fix regression bugs from 2.2.0’s fix to **CVE 2016-0718** [https://www.cve.org/CVERecord?id=CVE-2016-0718]) and **CVE 2012-0876** [https://www.cve.org/CVERecord?id=CVE-2012-0876] (Counter hash flooding with SipHash). Note: the **CVE 2016-5300** [https://www.cve.org/CVERecord?id=CVE-2016-5300] (Use os-specific entropy sources like getrandom) doesn’t impact Python, since Python already gets entropy from the OS to set the expat secret using `XML_SetHashSalt()`.

- bpo-30500 [https://bugs.python.org/issue?@action=redirect&bpo=30500]: Fix urllib.parse.splithost() to correctly parse fragments. For example, `splithost('//127.0.0.1#@evil.com/')` now correctly returns the `127.0.0.1` host, instead of treating `@evil.com` as the host in an authentication (`login@host`).

Python 3.6.2 release candidate 1

Release date: 2017-06-17

Security

- bpo-29591 [https://bugs.python.org/issue?@action=redirect&bpo=29591]: Update expat copy from 2.1.1 to 2.2.0 to get fixes of **CVE 2016-0718** [https://www.cve.org/CVERecord?id=CVE-2016-0718] and **CVE 2016-4472** [https://www.cve.org/CVERecord?id=CVE-2016-4472]. See https://sourceforge.net/p/expat/bugs/537/ for more information.

Core and Builtins

- bpo-30682 [https://bugs.python.org/issue?@action=redirect&bpo=30682]: Removed a too-strict assertion that failed for certain f-strings, such as eval(“f’\n’”) and eval(“f’\r’”).

- bpo-30604 [https://bugs.python.org/issue?@action=redirect&bpo=30604]: Move co_extra_freefuncs to not be per-thread to avoid crashes
- bpo-29104 [https://bugs.python.org/issue?@action=redirect&bpo=29104]: Fixed parsing backslashes in f-strings.
- bpo-27945 [https://bugs.python.org/issue?@action=redirect&bpo=27945]: Fixed various segfaults with dict when input collections are mutated during searching, inserting or comparing. Based on patches by Duane Griffin and Tim Mitchell.
- bpo-25794 [https://bugs.python.org/issue?@action=redirect&bpo=25794]: Fixed type.__setattr__() and type.__delattr__() for non-interned attribute names. Based on patch by Eryk Sun.
- bpo-30039 [https://bugs.python.org/issue?@action=redirect&bpo=30039]: If a KeyboardInterrupt happens when the interpreter is in the middle of resuming a chain of nested ‘yield from’ or ‘await’ calls, it’s now correctly delivered to the innermost frame.
- bpo-12414 [https://bugs.python.org/issue?@action=redirect&bpo=12414]: sys.getsizeof() on a code object now returns the sizes which includes the code struct and sizes of objects which it references. Patch by Donghee Na.
- bpo-29949 [https://bugs.python.org/issue?@action=redirect&bpo=29949]: Fix memory usage regression of set and frozenset object.
- bpo-29935 [https://bugs.python.org/issue?@action=redirect&bpo=29935]: Fixed error messages in the index() method of tuple, list and deque when pass indices of wrong type.
- bpo-29859 [https://bugs.python.org/issue?@action=redirect&bpo=29859]: Show correct error messages

when any of the pthread_* calls in thread_pthread.h fails.

- bpo-28876 [https://bugs.python.org/issue?@action=redirect&bpo=28876]: `bool(range)` works even if `len(range)` raises **`OverflowError`**.
- bpo-29600 [https://bugs.python.org/issue?@action=redirect&bpo=29600]: Fix wrapping coroutine return values in StopIteration.
- bpo-28856 [https://bugs.python.org/issue?@action=redirect&bpo=28856]: Fix an oversight that %b format for bytes should support objects follow the buffer protocol.
- bpo-29714 [https://bugs.python.org/issue?@action=redirect&bpo=29714]: Fix a regression that bytes format may fail when containing zero bytes inside.
- bpo-29478 [https://bugs.python.org/issue?@action=redirect&bpo=29478]: If max_line_length=None is specified while using the Compat32 policy, it is no longer ignored. Patch by Mircea Cosbuc.

Library

- bpo-30616 [https://bugs.python.org/issue?@action=redirect&bpo=30616]: Functional API of enum allows to create empty enums. Patched by Donghee Na
- bpo-30038 [https://bugs.python.org/issue?@action=redirect&bpo=30038]: Fix race condition between signal delivery and wakeup file descriptor. Patch by Nathaniel Smith.
- bpo-23894 [https://bugs.python.org/issue?@action=redirect&bpo=23894]: lib2to3 now recognizes `rb'...'` and `f'...'` strings.
- bpo-23890 [https://bugs.python.org/issue?@action=redirect&bpo=23890]: unittest.TestCase.assertRaises() now manually breaks a reference cycle to not keep objects alive longer than expected.

- bpo-30149 [https://bugs.python.org/issue?@action=redirect&bpo=30149]: inspect.signature() now supports callables with variable-argument parameters wrapped with partialmethod. Patch by Donghee Na.
- bpo-30645 [https://bugs.python.org/issue?@action=redirect&bpo=30645]: Fix path calculation in imp.load_package(), fixing it for cases when a package is only shipped with bytecodes. Patch by Alexandru Ardelean.
- bpo-29931 [https://bugs.python.org/issue?@action=redirect&bpo=29931]: Fixed comparison check for ipaddress.ip_interface objects. Patch by Sanjay Sundaresan.
- bpo-30605 [https://bugs.python.org/issue?@action=redirect&bpo=30605]: re.compile() no longer raises a BytesWarning when compiling a bytes instance with misplaced inline modifier. Patch by Roy Williams.
- bpo-24484 [https://bugs.python.org/issue?@action=redirect&bpo=24484]: Avoid race condition in multiprocessing cleanup (#2159)
- bpo-28994 [https://bugs.python.org/issue?@action=redirect&bpo=28994]: The traceback no longer displayed for SystemExit raised in a callback registered by atexit.
- bpo-30508 [https://bugs.python.org/issue?@action=redirect&bpo=30508]: Don’t log exceptions if Task/Future “cancel()” method was called.
- bpo-28556 [https://bugs.python.org/issue?@action=redirect&bpo=28556]: Updates to typing module: Add generic AsyncContextManager, add support for ContextManager on all versions. Original PRs by Jelle Zijlstra and Ivan Levkivskyi
- bpo-29870 [https://bugs.python.org/issue?@action=redirect&bpo=29870]: Fix ssl sockets leaks when connection is aborted in asyncio/ssl implementation. Patch by Michaël Sghaïer.

- bpo-29743 [https://bugs.python.org/issue?@action=redirect&bpo=29743]: Closing transport during handshake process leaks open socket. Patch by Nikolay Kim
- bpo-27585 [https://bugs.python.org/issue?@action=redirect&bpo=27585]: Fix waiter cancellation in asyncio.Lock. Patch by Mathieu Sornay.
- bpo-30418 [https://bugs.python.org/issue?@action=redirect&bpo=30418]: On Windows, subprocess.Popen.communicate() now also ignore EINVAL on stdin.write() if the child process is still running but closed the pipe.
- bpo-29822 [https://bugs.python.org/issue?@action=redirect&bpo=29822]: inspect.isabstract() now works during __init_subclass__. Patch by Nate Soares.
- bpo-29581 [https://bugs.python.org/issue?@action=redirect&bpo=29581]: ABCMeta.__new__ now accepts `**kwargs`, allowing abstract base classes to use keyword parameters in __init_subclass__. Patch by Nate Soares.
- bpo-30557 [https://bugs.python.org/issue?@action=redirect&bpo=30557]: faulthandler now correctly filters and displays exception codes on Windows
- bpo-30378 [https://bugs.python.org/issue?@action=redirect&bpo=30378]: Fix the problem that logging.handlers.SysLogHandler cannot handle IPv6 addresses.
- bpo-29960 [https://bugs.python.org/issue?@action=redirect&bpo=29960]: Preserve generator state when _random.Random.setstate() raises an exception. Patch by Bryan Olson.
- bpo-30414 [https://bugs.python.org/issue?@action=redirect&bpo=30414]: multiprocessing.Queue._feed background running thread do not break from main loop on exception.
- bpo-30003 [https://bugs.python.org/issue?@action=redirect&bpo=30003]: Fix handling escape characters

in HZ codec. Based on patch by Ma Lin.

- bpo-30301 [https://bugs.python.org/issue?@action=redirect&bpo=30301]: Fix AttributeError when using SimpleQueue.empty() under *spawn* and *forkserver* start methods.
- bpo-30329 [https://bugs.python.org/issue?@action=redirect&bpo=30329]: imaplib and poplib now catch the Windows socket WSAEINVAL error (code 10022) on shutdown(SHUT_RDWR): An invalid operation was attempted. This error occurs sometimes on SSL connections.
- bpo-30375 [https://bugs.python.org/issue?@action=redirect&bpo=30375]: Warnings emitted when compile a regular expression now always point to the line in the user code. Previously they could point into inners of the re module if emitted from inside of groups or conditionals.
- bpo-30048 [https://bugs.python.org/issue?@action=redirect&bpo=30048]: Fixed `Task.cancel()` can be ignored when the task is running coroutine and the coroutine returned without any more `await`.
- bpo-30266 [https://bugs.python.org/issue?@action=redirect&bpo=30266]: contextlib.AbstractContextManager now supports anti-registration by setting __enter__ = None or __exit__ = None, following the pattern introduced in bpo-25958 [https://bugs.python.org/issue?@action=redirect&bpo=25958]. Patch by Jelle Zijlstra.
- bpo-30298 [https://bugs.python.org/issue?@action=redirect&bpo=30298]: Weaken the condition of deprecation warnings for inline modifiers. Now allowed several subsequential inline modifiers at the start of the pattern (e.g. `'(?i)(?s)...'`). In verbose mode whitespaces and comments now are allowed before and between inline modifiers (e.g. `'(?x) (?i) (?s)...'`).

- bpo-29990 [https://bugs.python.org/issue?@action=redirect&bpo=29990]: Fix range checking in GB18030 decoder. Original patch by Ma Lin.
- bpo-26293 [https://bugs.python.org/issue?@action=redirect&bpo=26293]: Change resulted because of zipfile breakage. (See also: bpo-29094 [https://bugs.python.org/issue?@action=redirect&bpo=29094])
- bpo-30243 [https://bugs.python.org/issue?@action=redirect&bpo=30243]: Removed the __init__ methods of _json’s scanner and encoder. Misusing them could cause memory leaks or crashes. Now scanner and encoder objects are completely initialized in the __new__ methods.
- bpo-30185 [https://bugs.python.org/issue?@action=redirect&bpo=30185]: Avoid KeyboardInterrupt tracebacks in forkserver helper process when Ctrl-C is received.
- bpo-28556 [https://bugs.python.org/issue?@action=redirect&bpo=28556]: Various updates to typing module: add typing.NoReturn type, use WrapperDescriptorType, minor bug-fixes. Original PRs by Jim Fasarakis-Hilliard and Ivan Levkivskyi.
- bpo-30205 [https://bugs.python.org/issue?@action=redirect&bpo=30205]: Fix getsockname() for unbound AF_UNIX sockets on Linux.
- bpo-30070 [https://bugs.python.org/issue?@action=redirect&bpo=30070]: Fixed leaks and crashes in errors handling in the parser module.
- bpo-30061 [https://bugs.python.org/issue?@action=redirect&bpo=30061]: Fixed crashes in IOBase methods __next__() and readlines() when readline() or __next__() respectively return non-sizeable object. Fixed possible other errors caused by not checking results of PyObject_Size(), PySequence_Size(), or PyMapping_Size().

- bpo-30017 [https://bugs.python.org/issue?@action=redirect&bpo=30017]: Allowed calling the close() method of the zip entry writer object multiple times. Writing to a closed writer now always produces a ValueError.
- bpo-30068 [https://bugs.python.org/issue?@action=redirect&bpo=30068]: _io._IOBase.readlines will check if it’s closed first when hint is present.
- bpo-29694 [https://bugs.python.org/issue?@action=redirect&bpo=29694]: Fixed race condition in pathlib mkdir with flags parents=True. Patch by Armin Rigo.
- bpo-29692 [https://bugs.python.org/issue?@action=redirect&bpo=29692]: Fixed arbitrary unchaining of RuntimeError exceptions in contextlib.contextmanager. Patch by Siddharth Velankar.
- bpo-29998 [https://bugs.python.org/issue?@action=redirect&bpo=29998]: Pickling and copying ImportError now preserves name and path attributes.
- bpo-29953 [https://bugs.python.org/issue?@action=redirect&bpo=29953]: Fixed memory leaks in the replace() method of datetime and time objects when pass out of bound fold argument.
- bpo-29942 [https://bugs.python.org/issue?@action=redirect&bpo=29942]: Fix a crash in itertools.chain.from_iterable when encountering long runs of empty iterables.
- bpo-27863 [https://bugs.python.org/issue?@action=redirect&bpo=27863]: Fixed multiple crashes in ElementTree caused by race conditions and wrong types.
- bpo-28699 [https://bugs.python.org/issue?@action=redirect&bpo=28699]: Fixed a bug in pools in multiprocessing.pool that raising an exception at the very first of an iterable may swallow the exception or make the program hang. Patch by Davin Potts and Xiang Zhang.

- bpo-25803 [https://bugs.python.org/issue?@action=redirect&bpo=25803]: Avoid incorrect errors raised by Path.mkdir(exist_ok=True) when the OS gives priority to errors such as EACCES over EEXIST.
- bpo-29861 [https://bugs.python.org/issue?@action=redirect&bpo=29861]: Release references to tasks, their arguments and their results as soon as they are finished in multiprocessing.Pool.
- bpo-29884 [https://bugs.python.org/issue?@action=redirect&bpo=29884]: faulthandler: Restore the old sigaltstack during teardown. Patch by Christophe Zeitouny.
- bpo-25455 [https://bugs.python.org/issue?@action=redirect&bpo=25455]: Fixed crashes in repr of recursive buffered file-like objects.
- bpo-29800 [https://bugs.python.org/issue?@action=redirect&bpo=29800]: Fix crashes in partial.__repr__ if the keys of partial.keywords are not strings. Patch by Michael Seifert.
- bpo-29742 [https://bugs.python.org/issue?@action=redirect&bpo=29742]: get_extra_info() raises exception if get called on closed ssl transport. Patch by Nikolay Kim.
- bpo-8256 [https://bugs.python.org/issue?@action=redirect&bpo=8256]: Fixed possible failing or crashing input() if attributes “encoding” or “errors” of sys.stdin or sys.stdout are not set or are not strings.
- bpo-28298 [https://bugs.python.org/issue?@action=redirect&bpo=28298]: Fix a bug that prevented array ‘Q’, ‘L’ and ‘I’ from accepting big intables (objects that have __int__) as elements. Patch by Oren Milman.
- bpo-28231 [https://bugs.python.org/issue?@action=redirect&bpo=28231]: The zipfile module now accepts path-like objects for external paths.
- bpo-26915 [https://bugs.python.org/issue?@action=redirect&bpo=26915]: index() and count() methods of

collections.abc.Sequence now check identity before checking equality when do comparisons.

- bpo-29615 [https://bugs.python.org/issue?@action=redirect&bpo=29615]: SimpleXMLRPCDispatcher no longer chains KeyError (or any other exception) to exception(s) raised in the dispatched methods. Patch by Petr Motejlek.
- bpo-30177 [https://bugs.python.org/issue?@action=redirect&bpo=30177]: path.resolve(strict=False) no longer cuts the path after the first element not present in the filesystem. Patch by Antoine Pietri.

IDLE

- bpo-15786 [https://bugs.python.org/issue?@action=redirect&bpo=15786]: Fix several problems with IDLE’s autocompletion box. The following should now work: clicking on selection box items; using the scrollbar; selecting an item by hitting Return. Hangs on MacOSX should no longer happen. Patch by Louie Lu.
- bpo-25514 [https://bugs.python.org/issue?@action=redirect&bpo=25514]: Add doc subsubsection about IDLE failure to start. Popup no-connection message directs users to this section.
- bpo-30642 [https://bugs.python.org/issue?@action=redirect&bpo=30642]: Fix reference leaks in IDLE tests. Patches by Louie Lu and Terry Jan Reedy.
- bpo-30495 [https://bugs.python.org/issue?@action=redirect&bpo=30495]: Add docstrings for textview.py and use PEP8 names. Patches by Cheryl Sabella and Terry Jan Reedy.
- bpo-30290 [https://bugs.python.org/issue?@action=redirect&bpo=30290]: Help-about: use pep8 names and add tests. Increase coverage to 100%. Patches by Louie Lu, Cheryl Sabella, and Terry Jan Reedy.

- bpo-30303 [https://bugs.python.org/issue?@action=redirect&bpo=30303]: Add _utest option to textview; add new tests. Increase coverage to 100%. Patches by Louie Lu and Terry Jan Reedy.

C API

- bpo-27867 [https://bugs.python.org/issue?@action=redirect&bpo=27867]: Function PySlice_GetIndicesEx() no longer replaced with a macro if Py_LIMITED_API is not set.

Build

- bpo-29941 [https://bugs.python.org/issue?@action=redirect&bpo=29941]: Add `--with-assertions` configure flag to explicitly enable C `assert()` checks. Defaults to off. `--with-pydebug` implies `--with-assertions`.
- bpo-28787 [https://bugs.python.org/issue?@action=redirect&bpo=28787]: Fix out-of-tree builds of Python when configured with `--with--dtrace`.
- bpo-29243 [https://bugs.python.org/issue?@action=redirect&bpo=29243]: Prevent unnecessary rebuilding of Python during `make test`, `make install` and some other make targets when configured with `--enable-optimizations`.
- bpo-23404 [https://bugs.python.org/issue?@action=redirect&bpo=23404]: Don’t regenerate generated files based on file modification time anymore: the action is now explicit. Replace `make touch` with `make regen-all`.
- bpo-29643 [https://bugs.python.org/issue?@action=redirect&bpo=29643]: Fix `--enable-optimization` didn’t work.

Documentation

- bpo-30176 [https://bugs.python.org/issue?@action=redirect&bpo=30176]: Add missing attribute related constants in curses documentation.
- bpo-30052 [https://bugs.python.org/issue?@action=redirect&bpo=30052]: the link targets for **`bytes()`** and **`bytearray()`** are now their respective type definitions, rather than the corresponding builtin function entries. Use bytes and bytearray to reference the latter. In order to ensure this and future cross-reference updates are applied automatically, the daily documentation builds now disable the default output caching features in Sphinx.
- bpo-26985 [https://bugs.python.org/issue?@action=redirect&bpo=26985]: Add missing info of code object in inspect documentation.

Tools/Demos

- bpo-29367 [https://bugs.python.org/issue?@action=redirect&bpo=29367]: python-gdb.py now supports also `method-wrapper` (`wrapperobject`) objects.

Tests

- bpo-30357 [https://bugs.python.org/issue?@action=redirect&bpo=30357]: test_thread: setUp() now uses support.threading_setup() and support.threading_cleanup() to wait until threads complete to avoid random side effects on following tests. Initial patch written by Grzegorz Grzywacz.
- bpo-30197 [https://bugs.python.org/issue?@action=redirect&bpo=30197]: Enhanced functions swap_attr() and swap_item() in the test.support module. They now work when delete replaced attribute or item inside the with statement. The old value of the attribute or item

(or None if it doesn’t exist) now will be assigned to the target of the “as” clause, if there is one.

Windows

- bpo-30687 [https://bugs.python.org/issue?@action=redirect&bpo=30687]: Locate msbuild.exe on Windows when building rather than vcvarsall.bat
- bpo-30450 [https://bugs.python.org/issue?@action=redirect&bpo=30450]: The build process on Windows no longer depends on Subversion, instead pulling external code from GitHub via a Python script. If Python 3.6 is not found on the system (via `py -3.6`), NuGet is used to download a copy of 32-bit Python.

Python 3.6.1 final

Release date: 2017-03-21

Core and Builtins

- bpo-29723 [https://bugs.python.org/issue?@action=redirect&bpo=29723]: The `sys.path[0]` initialization change for bpo-29139 [https://bugs.python.org/issue?@action=redirect&bpo=29139] caused a regression by revealing an inconsistency in how sys.path is initialized when executing `__main__` from a zipfile, directory, or other import location. The interpreter now consistently avoids ever adding the import location’s parent directory to `sys.path`, and ensures no other `sys.path` entries are inadvertently modified when inserting the import location named on the command line.

Build

- bpo-27593 [https://bugs.python.org/issue?@action=redirect&bpo=27593]: fix format of git information used in sys.version
- Fix incompatible comment in python.h

Python 3.6.1 release candidate 1

Release date: 2017-03-04

Core and Builtins

- bpo-28893 [https://bugs.python.org/issue?@action=redirect&bpo=28893]: Set correct __cause__ for errors about invalid awaitables returned from __aiter__ and __anext__.
- bpo-29683 [https://bugs.python.org/issue?@action=redirect&bpo=29683]: Fixes to memory allocation in _PyCode_SetExtra. Patch by Brian Coleman.
- bpo-29684 [https://bugs.python.org/issue?@action=redirect&bpo=29684]: Fix minor regression of PyEval_CallObjectWithKeywords. It should raise TypeError when kwargs is not a dict. But it might cause segv when args=NULL and kwargs is not a dict.
- bpo-28598 [https://bugs.python.org/issue?@action=redirect&bpo=28598]: Support __rmod__ for subclasses of str being called before str.__mod__. Patch by Martijn Pieters.
- bpo-29607 [https://bugs.python.org/issue?@action=redirect&bpo=29607]: Fix stack_effect computation for CALL_FUNCTION_EX. Patch by Matthieu Dartiailh.
- bpo-29602 [https://bugs.python.org/issue?@action=redirect&bpo=29602]: Fix incorrect handling of signed zeros in complex constructor for complex subclasses and for inputs having a __complex__ method. Patch by Serhiy Storchaka.

- bpo-29347 [https://bugs.python.org/issue?@action=redirect&bpo=29347]: Fixed possibly dereferencing undefined pointers when creating weakref objects.
- bpo-29438 [https://bugs.python.org/issue?@action=redirect&bpo=29438]: Fixed use-after-free problem in key sharing dict.
- bpo-29319 [https://bugs.python.org/issue?@action=redirect&bpo=29319]: Prevent RunMainFromImporter overwriting sys.path[0].
- bpo-29337 [https://bugs.python.org/issue?@action=redirect&bpo=29337]: Fixed possible BytesWarning when compare the code objects. Warnings could be emitted at compile time.
- bpo-29327 [https://bugs.python.org/issue?@action=redirect&bpo=29327]: Fixed a crash when pass the iterable keyword argument to sorted().
- bpo-29034 [https://bugs.python.org/issue?@action=redirect&bpo=29034]: Fix memory leak and use-after-free in os module (path_converter).
- bpo-29159 [https://bugs.python.org/issue?@action=redirect&bpo=29159]: Fix regression in bytes(x) when x.__index__() raises Exception.
- bpo-28932 [https://bugs.python.org/issue?@action=redirect&bpo=28932]: Do not include <sys/random.h> if it does not exist.
- bpo-25677 [https://bugs.python.org/issue?@action=redirect&bpo=25677]: Correct the positioning of the syntax error caret for indented blocks. Based on patch by Michael Layzell.
- bpo-29000 [https://bugs.python.org/issue?@action=redirect&bpo=29000]: Fixed bytes formatting of octals with zero padding in alternate form.
- bpo-26919 [https://bugs.python.org/issue?@action=redirect&bpo=26919]: On Android, operating system data is now always encoded/decoded to/from UTF-8, instead of the locale encoding to avoid inconsistencies

with os.fsencode() and os.fsdecode() which are already using UTF-8.

- bpo-28991 [https://bugs.python.org/issue?@action=redirect&bpo=28991]: functools.lru_cache() was susceptible to an obscure reentrancy bug triggerable by a monkey-patched len() function.
- bpo-28739 [https://bugs.python.org/issue?@action=redirect&bpo=28739]: f-string expressions are no longer accepted as docstrings and by ast.literal_eval() even if they do not include expressions.
- bpo-28512 [https://bugs.python.org/issue?@action=redirect&bpo=28512]: Fixed setting the offset attribute of SyntaxError by PyErr_SyntaxLocationEx() and PyErr_SyntaxLocationObject().
- bpo-28918 [https://bugs.python.org/issue?@action=redirect&bpo=28918]: Fix the cross compilation of xxlimited when Python has been built with Py_DEBUG defined.
- bpo-28731 [https://bugs.python.org/issue?@action=redirect&bpo=28731]: Optimize _PyDict_NewPresized() to create correct size dict. Improve speed of dict literal with constant keys up to 30%.

Library

- bpo-29169 [https://bugs.python.org/issue?@action=redirect&bpo=29169]: Update zlib to 1.2.11.
- bpo-29623 [https://bugs.python.org/issue?@action=redirect&bpo=29623]: Allow use of path-like object as a single argument in ConfigParser.read(). Patch by David Ellis.
- bpo-28963 [https://bugs.python.org/issue?@action=redirect&bpo=28963]: Fix out of bound iteration in asyncio.Future.remove_done_callback implemented in C.

- bpo-29704 [https://bugs.python.org/issue?@action=redirect&bpo=29704]: asyncio.subprocess.SubprocessStreamProtocol no longer closes before all pipes are closed.
- bpo-29271 [https://bugs.python.org/issue?@action=redirect&bpo=29271]: Fix Task.current_task and Task.all_tasks implemented in C to accept None argument as their pure Python implementation.
- bpo-29703 [https://bugs.python.org/issue?@action=redirect&bpo=29703]: Fix asyncio to support instantiation of new event loops in child processes.
- bpo-29376 [https://bugs.python.org/issue?@action=redirect&bpo=29376]: Fix assertion error in threading._DummyThread.is_alive().
- bpo-28624 [https://bugs.python.org/issue?@action=redirect&bpo=28624]: Add a test that checks that cwd parameter of Popen() accepts PathLike objects. Patch by Sayan Chowdhury.
- bpo-28518 [https://bugs.python.org/issue?@action=redirect&bpo=28518]: Start a transaction implicitly before a DML statement. Patch by Aviv Palivoda.
- bpo-29532 [https://bugs.python.org/issue?@action=redirect&bpo=29532]: Altering a kwarg dictionary passed to functools.partial() no longer affects a partial object after creation.
- bpo-29110 [https://bugs.python.org/issue?@action=redirect&bpo=29110]: Fix file object leak in aifc.open() when file is given as a filesystem path and is not in valid AIFF format. Patch by Anthony Zhang.
- bpo-28556 [https://bugs.python.org/issue?@action=redirect&bpo=28556]: Various updates to typing module: typing.Counter, typing.ChainMap, improved ABC caching, etc. Original PRs by Jelle Zijlstra, Ivan Levkivskyi, Manuel Krebber, and Łukasz Langa.
- bpo-29100 [https://bugs.python.org/issue?@action=redirect&bpo=29100]: Fix datetime.fromtimestamp()

regression introduced in Python 3.6.0: check minimum and maximum years.

- bpo-29519 [https://bugs.python.org/issue?@action=redirect&bpo=29519]: Fix weakref spewing exceptions during interpreter shutdown when used with a rare combination of multiprocessing and custom codecs.
- bpo-29416 [https://bugs.python.org/issue?@action=redirect&bpo=29416]: Prevent infinite loop in pathlib.Path.mkdir
- bpo-29444 [https://bugs.python.org/issue?@action=redirect&bpo=29444]: Fixed out-of-bounds buffer access in the group() method of the match object. Based on patch by WGH.
- bpo-29335 [https://bugs.python.org/issue?@action=redirect&bpo=29335]: Fix subprocess.Popen.wait() when the child process has exited to a stopped instead of terminated state (ex: when under ptrace).
- bpo-29290 [https://bugs.python.org/issue?@action=redirect&bpo=29290]: Fix a regression in argparse that help messages would wrap at non-breaking spaces.
- bpo-28735 [https://bugs.python.org/issue?@action=redirect&bpo=28735]: Fixed the comparison of mock.MagickMock with mock.ANY.
- bpo-29316 [https://bugs.python.org/issue?@action=redirect&bpo=29316]: Restore the provisional status of typing module, add corresponding note to documentation. Patch by Ivan L.
- bpo-29219 [https://bugs.python.org/issue?@action=redirect&bpo=29219]: Fixed infinite recursion in the repr of uninitialized ctypes.CDLL instances.
- bpo-29011 [https://bugs.python.org/issue?@action=redirect&bpo=29011]: Fix an important omission by adding Deque to the typing module.
- bpo-28969 [https://bugs.python.org/issue?@action=redirect&bpo=28969]: Fixed race condition in C implementation of functools.lru_cache. KeyError could

be raised when cached function with full cache was simultaneously called from different threads with the same uncached arguments.

- bpo-29142 [https://bugs.python.org/issue?@action=redirect&bpo=29142]: In urllib.request, suffixes in no_proxy environment variable with leading dots could match related hostnames again (e.g. .b.c matches a.b.c). Patch by Milan Oberkirch.
- bpo-28961 [https://bugs.python.org/issue?@action=redirect&bpo=28961]: Fix unittest.mock._Call helper: don’t ignore the name parameter anymore. Patch written by Jiajun Huang.
- bpo-29203 [https://bugs.python.org/issue?@action=redirect&bpo=29203]: functools.lru_cache() now respects **PEP 468** [https://peps.python.org/pep-0468/] and preserves the order of keyword arguments. f(a=1, b=2) is now cached separately from f(b=2, a=1) since both calls could potentially give different results.
- bpo-15812 [https://bugs.python.org/issue?@action=redirect&bpo=15812]: inspect.getframeinfo() now correctly shows the first line of a context. Patch by Sam Breese.
- bpo-29094 [https://bugs.python.org/issue?@action=redirect&bpo=29094]: Offsets in a ZIP file created with extern file object and modes “w” and “x” now are relative to the start of the file.
- bpo-29085 [https://bugs.python.org/issue?@action=redirect&bpo=29085]: Allow random.Random.seed() to use high quality OS randomness rather than the pid and time.
- bpo-29061 [https://bugs.python.org/issue?@action=redirect&bpo=29061]: Fixed bug in secrets.randbelow() which would hang when given a negative input. Patch by Brendan Donegan.
- bpo-29079 [https://bugs.python.org/issue?@action=redirect&bpo=29079]: Prevent infinite loop in

pathlib.resolve() on Windows

- bpo-13051 [https://bugs.python.org/issue?@action=redirect&bpo=13051]: Fixed recursion errors in large or resized curses.textpad.Textbox. Based on patch by Tycho Andersen.
- bpo-29119 [https://bugs.python.org/issue?@action=redirect&bpo=29119]: Fix weakrefs in the pure python version of collections.OrderedDict move_to_end() method. Contributed by Andra Bogildea.
- bpo-9770 [https://bugs.python.org/issue?@action=redirect&bpo=9770]: curses.ascii predicates now work correctly with negative integers.
- bpo-28427 [https://bugs.python.org/issue?@action=redirect&bpo=28427]: old keys should not remove new values from WeakValueDictionary when collecting from another thread.
- bpo-28923 [https://bugs.python.org/issue?@action=redirect&bpo=28923]: Remove editor artifacts from Tix.py.
- bpo-29055 [https://bugs.python.org/issue?@action=redirect&bpo=29055]: Neaten-up empty population error on random.choice() by suppressing the upstream exception.
- bpo-28871 [https://bugs.python.org/issue?@action=redirect&bpo=28871]: Fixed a crash when deallocate deep ElementTree.
- bpo-19542 [https://bugs.python.org/issue?@action=redirect&bpo=19542]: Fix bugs in WeakValueDictionary.setdefault() and WeakValueDictionary.pop() when a GC collection happens in another thread.
- bpo-20191 [https://bugs.python.org/issue?@action=redirect&bpo=20191]: Fixed a crash in resource.prlimit() when passing a sequence that doesn’t own its elements as limits.

- bpo-28779 [https://bugs.python.org/issue?@action=redirect&bpo=28779]: multiprocessing.set_forkserver_preload() would crash the forkserver process if a preloaded module instantiated some multiprocessing objects such as locks.
- bpo-28847 [https://bugs.python.org/issue?@action=redirect&bpo=28847]: dbm.dumb now supports reading read-only files and no longer writes the index file when it is not changed.
- bpo-26937 [https://bugs.python.org/issue?@action=redirect&bpo=26937]: The chown() method of the tarfile.TarFile class does not fail now when the grp module cannot be imported, as for example on Android platforms.

IDLE

- bpo-29071 [https://bugs.python.org/issue?@action=redirect&bpo=29071]: IDLE colors f-string prefixes (but not invalid ur prefixes).
- bpo-28572 [https://bugs.python.org/issue?@action=redirect&bpo=28572]: Add 10% to coverage of IDLE’s test_configdialog. Update and augment description of the configuration system.

Windows

- bpo-29579 [https://bugs.python.org/issue?@action=redirect&bpo=29579]: Removes readme.txt from the installer
- bpo-29326 [https://bugs.python.org/issue?@action=redirect&bpo=29326]: Ignores blank lines in ._pth files (Patch by Alexey Izbyshev)
- bpo-28164 [https://bugs.python.org/issue?@action=redirect&bpo=28164]: Correctly handle special console filenames (patch by Eryk Sun)

- bpo-29409 [https://bugs.python.org/issue?@action=redirect&bpo=29409]: Implement **PEP 529** [https://peps.python.org/pep-0529/] for io.FileIO (Patch by Eryk Sun)
- bpo-29392 [https://bugs.python.org/issue?@action=redirect&bpo=29392]: Prevent crash when passing invalid arguments into msvcrt module.
- bpo-25778 [https://bugs.python.org/issue?@action=redirect&bpo=25778]: winreg does not truncate string correctly (Patch by Eryk Sun)
- bpo-28896 [https://bugs.python.org/issue?@action=redirect&bpo=28896]: Deprecate WindowsRegistryFinder and disable it by default.

C API

- bpo-27867 [https://bugs.python.org/issue?@action=redirect&bpo=27867]: Function PySlice_GetIndicesEx() is replaced with a macro if Py_LIMITED_API is not set or set to the value between 0x03050400 and 0x03060000 (not including) or 0x03060100 or higher.
- bpo-29083 [https://bugs.python.org/issue?@action=redirect&bpo=29083]: Fixed the declaration of some public API functions. PyArg_VaParse() and PyArg_VaParseTupleAndKeywords() were not available in limited API. PyArg_ValidateKeywordArguments(), PyArg_UnpackTuple() and Py_BuildValue() were not available in limited API of version < 3.3 when PY_SSIZE_T_CLEAN is defined.
- bpo-29058 [https://bugs.python.org/issue?@action=redirect&bpo=29058]: All stable API extensions added after Python 3.2 are now available only when Py_LIMITED_API is set to the PY_VERSION_HEX value of the minimum Python version supporting this API.

Documentation

- bpo-28929 [https://bugs.python.org/issue?@action=redirect&bpo=28929]: Link the documentation to its source file on GitHub.
- bpo-25008 [https://bugs.python.org/issue?@action=redirect&bpo=25008]: Document smtpd.py as effectively deprecated and add a pointer to aiosmtpd, a third-party asyncio-based replacement.
- bpo-26355 [https://bugs.python.org/issue?@action=redirect&bpo=26355]: Add canonical header link on each page to corresponding major version of the documentation. Patch by Matthias Bussonnier.
- bpo-29349 [https://bugs.python.org/issue?@action=redirect&bpo=29349]: Fix Python 2 syntax in code for building the documentation.

Tests

- bpo-28087 [https://bugs.python.org/issue?@action=redirect&bpo=28087]: Skip test_asyncore and test_eintr poll failures on macOS. Skip some tests of select.poll when running on macOS due to unresolved issues with the underlying system poll function on some macOS versions.
- bpo-29571 [https://bugs.python.org/issue?@action=redirect&bpo=29571]: to match the behaviour of the `re.LOCALE` flag, test_re.test_locale_flag now uses `locale.getpreferredencoding(False)` to determine the candidate encoding for the test regex (allowing it to correctly skip the test when the default locale encoding is a multi-byte encoding)
- bpo-28950 [https://bugs.python.org/issue?@action=redirect&bpo=28950]: Disallow -j0 to be combined with -T/-l in regrtest command line arguments.

- bpo-28683 [https://bugs.python.org/issue?@action=redirect&bpo=28683]: Fix the tests that bind() a unix socket and raise PermissionError on Android for a non-root user.
- bpo-26939 [https://bugs.python.org/issue?@action=redirect&bpo=26939]: Add the support.setswitchinterval() function to fix test_functools hanging on the Android armv7 qemu emulator.

Build

- bpo-27593 [https://bugs.python.org/issue?@action=redirect&bpo=27593]: sys.version and the platform module python_build(), python_branch(), and python_revision() functions now use git information rather than hg when building from a repo.
- bpo-29572 [https://bugs.python.org/issue?@action=redirect&bpo=29572]: Update Windows build and OS X installers to use OpenSSL 1.0.2k.
- bpo-26851 [https://bugs.python.org/issue?@action=redirect&bpo=26851]: Set Android compilation and link flags.
- bpo-28768 [https://bugs.python.org/issue?@action=redirect&bpo=28768]: Fix implicit declaration of function _setmode. Patch by Masayuki Yamamoto
- bpo-29080 [https://bugs.python.org/issue?@action=redirect&bpo=29080]: Removes hard dependency on hg.exe from PCBuild/build.bat
- bpo-23903 [https://bugs.python.org/issue?@action=redirect&bpo=23903]: Added missed names to PC/python3.def.
- bpo-28762 [https://bugs.python.org/issue?@action=redirect&bpo=28762]: lockf() is available on Android API level 24, but the F_LOCK macro is not defined in android-ndk-r13.

- bpo-28538 [https://bugs.python.org/issue?@action=redirect&bpo=28538]: Fix the compilation error that occurs because if_nameindex() is available on Android API level 24, but the if_nameindex structure is not defined.
- bpo-20211 [https://bugs.python.org/issue?@action=redirect&bpo=20211]: Do not add the directory for installing C header files and the directory for installing object code libraries to the cross compilation search paths. Original patch by Thomas Petazzoni.
- bpo-28849 [https://bugs.python.org/issue?@action=redirect&bpo=28849]: Do not define sys.implementation._multiarch on Android.

Python 3.6.0 final

Release date: 2016-12-23

No changes since release candidate 2

Python 3.6.0 release candidate 2

Release date: 2016-12-16

Core and Builtins

- bpo-28147 [https://bugs.python.org/issue?@action=redirect&bpo=28147]: Fix a memory leak in split-table dictionaries: setattr() must not convert combined table into split table. Patch written by INADA Naoki.
- bpo-28990 [https://bugs.python.org/issue?@action=redirect&bpo=28990]: Fix asyncio SSL hanging if connection is closed before handshake is completed. (Patch by HoHo-Ho)

Tools/Demos

- bpo-28770 [https://bugs.python.org/issue?@action=redirect&bpo=28770]: Fix python-gdb.py for fastcalls.

Windows

- bpo-28896 [https://bugs.python.org/issue?@action=redirect&bpo=28896]: Deprecate WindowsRegistryFinder.

Build

- bpo-28898 [https://bugs.python.org/issue?@action=redirect&bpo=28898]: Prevent gdb build errors due to HAVE_LONG_LONG redefinition.

Python 3.6.0 release candidate 1

Release date: 2016-12-06

Core and Builtins

- bpo-23722 [https://bugs.python.org/issue?@action=redirect&bpo=23722]: Rather than silently producing a class that doesn’t support zero-argument `super()` in methods, failing to pass the new `__classcell__` namespace entry up to `type.__new__` now results in a `DeprecationWarning` and a class that supports zero-argument `super()`.
- bpo-28797 [https://bugs.python.org/issue?@action=redirect&bpo=28797]: Modifying the class __dict__ inside the __set_name__ method of a descriptor that is used inside that class no longer prevents calling the __set_name__ method of other descriptors.

- bpo-28782 [https://bugs.python.org/issue?@action=redirect&bpo=28782]: Fix a bug in the implementation `yield from` when checking if the next instruction is YIELD_FROM. Regression introduced by WORDCODE (bpo-26647 [https://bugs.python.org/issue?@action=redirect&bpo=26647]).

Library

- bpo-27030 [https://bugs.python.org/issue?@action=redirect&bpo=27030]: Unknown escapes in re.sub() replacement template are allowed again. But they still are deprecated and will be disabled in 3.7.
- bpo-28835 [https://bugs.python.org/issue?@action=redirect&bpo=28835]: Fix a regression introduced in warnings.catch_warnings(): call warnings.showwarning() if it was overridden inside the context manager.
- bpo-27172 [https://bugs.python.org/issue?@action=redirect&bpo=27172]: To assist with upgrades from 2.7, the previously documented deprecation of `inspect.getfullargspec()` has been reversed. This decision may be revisited again after the Python 2.7 branch is no longer officially supported.
- bpo-26273 [https://bugs.python.org/issue?@action=redirect&bpo=26273]: Add new **`socket.TCP_CONGESTION`** (Linux 2.6.13) and **`socket.TCP_USER_TIMEOUT`** (Linux 2.6.37) constants. Patch written by Omar Sandoval.
- bpo-24142 [https://bugs.python.org/issue?@action=redirect&bpo=24142]: Reading a corrupt config file left configparser in an invalid state. Original patch by Florian Höch.
- bpo-28843 [https://bugs.python.org/issue?@action=redirect&bpo=28843]: Fix asyncio C Task to handle exceptions __traceback__.

C API

- bpo-28808 [https://bugs.python.org/issue?@action=redirect&bpo=28808]: PyUnicode_CompareWithASCIIString() now never raises exceptions.

Documentation

- bpo-23722 [https://bugs.python.org/issue?@action=redirect&bpo=23722]: The data model reference and the porting section in the What's New guide now cover the additional `__classcell__` handling needed for custom metaclasses to fully support **PEP 487** [https://peps.python.org/pep-0487/] and zero-argument `super()`.

Tools/Demos

- bpo-28023 [https://bugs.python.org/issue?@action=redirect&bpo=28023]: Fix python-gdb.py didn't support new dict implementation.

Python 3.6.0 beta 4

Release date: 2016-11-21

Core and Builtins

- bpo-28532 [https://bugs.python.org/issue?@action=redirect&bpo=28532]: Show sys.version when -V option is supplied twice.
- bpo-27100 [https://bugs.python.org/issue?@action=redirect&bpo=27100]: The with-statement now checks for __enter__ before it checks for __exit__. This gives less

confusing error messages when both methods are missing. Patch by Jonathan Ellington.

- bpo-28746 [https://bugs.python.org/issue?@action=redirect&bpo=28746]: Fix the set_inheritable() file descriptor method on platforms that do not have the ioctl FIOCLEX and FIONCLEX commands.
- bpo-26920 [https://bugs.python.org/issue?@action=redirect&bpo=26920]: Fix not getting the locale’s charset upon initializing the interpreter, on platforms that do not have langinfo.
- bpo-28648 [https://bugs.python.org/issue?@action=redirect&bpo=28648]: Fixed crash in Py_DecodeLocale() in debug build on Mac OS X when decode astral characters. Patch by Xiang Zhang.
- bpo-19398 [https://bugs.python.org/issue?@action=redirect&bpo=19398]: Extra slash no longer added to sys.path components in case of empty compile-time PYTHONPATH components.
- bpo-28665 [https://bugs.python.org/issue?@action=redirect&bpo=28665]: Improve speed of the STORE_DEREF opcode by 40%.
- bpo-28583 [https://bugs.python.org/issue?@action=redirect&bpo=28583]: PyDict_SetDefault didn’t combine split table when needed. Patch by Xiang Zhang.
- bpo-27243 [https://bugs.python.org/issue?@action=redirect&bpo=27243]: Change PendingDeprecationWarning -> DeprecationWarning. As it was agreed in the issue, __aiter__ returning an awaitable should result in PendingDeprecationWarning in 3.5 and in DeprecationWarning in 3.6.
- bpo-26182 [https://bugs.python.org/issue?@action=redirect&bpo=26182]: Fix a refleak in code that raises DeprecationWarning.
- bpo-28721 [https://bugs.python.org/issue?@action=redirect&bpo=28721]: Fix asynchronous generators

aclose() and athrow() to handle StopAsyncIteration propagation properly.

Library

- bpo-28752 [https://bugs.python.org/issue?@action=redirect&bpo=28752]: Restored the __reduce__() methods of datetime objects.
- bpo-28727 [https://bugs.python.org/issue?@action=redirect&bpo=28727]: Regular expression patterns, _sre.SRE_Pattern objects created by re.compile(), become comparable (only x==y and x!=y operators). This change should fix the bpo-18383 [https://bugs.python.org/issue?@action=redirect&bpo=18383]: don’t duplicate warning filters when the warnings module is reloaded (thing usually only done in unit tests).
- bpo-20572 [https://bugs.python.org/issue?@action=redirect&bpo=20572]: The subprocess.Popen.wait method’s undocumented endtime parameter now raises a DeprecationWarning.
- bpo-25659 [https://bugs.python.org/issue?@action=redirect&bpo=25659]: In ctypes, prevent a crash calling the from_buffer() and from_buffer_copy() methods on abstract classes like Array.
- bpo-19717 [https://bugs.python.org/issue?@action=redirect&bpo=19717]: Makes Path.resolve() succeed on paths that do not exist. Patch by Vajrasky Kok
- bpo-28563 [https://bugs.python.org/issue?@action=redirect&bpo=28563]: Fixed possible DoS and arbitrary code execution when handle plural form selections in the gettext module. The expression parser now supports exact syntax supported by GNU gettext.
- bpo-28387 [https://bugs.python.org/issue?@action=redirect&bpo=28387]: Fixed possible crash in _io.TextIOWrapper deallocator when the garbage

collector is invoked in other thread. Based on patch by Sebastian Cufre.
- bpo-28600 [https://bugs.python.org/issue?@action=redirect&bpo=28600]: Optimize loop.call_soon.
- bpo-28613 [https://bugs.python.org/issue?@action=redirect&bpo=28613]: Fix get_event_loop() return the current loop if called from coroutines/callbacks.
- bpo-28634 [https://bugs.python.org/issue?@action=redirect&bpo=28634]: Fix asyncio.isfuture() to support unittest.Mock.
- bpo-26081 [https://bugs.python.org/issue?@action=redirect&bpo=26081]: Fix refleak in _asyncio.Future.__iter__().throw.
- bpo-28639 [https://bugs.python.org/issue?@action=redirect&bpo=28639]: Fix inspect.isawaitable to always return bool Patch by Justin Mayfield.
- bpo-28652 [https://bugs.python.org/issue?@action=redirect&bpo=28652]: Make loop methods reject socket kinds they do not support.
- bpo-28653 [https://bugs.python.org/issue?@action=redirect&bpo=28653]: Fix a refleak in functools.lru_cache.
- bpo-28703 [https://bugs.python.org/issue?@action=redirect&bpo=28703]: Fix asyncio.iscoroutinefunction to handle Mock objects.
- bpo-28704 [https://bugs.python.org/issue?@action=redirect&bpo=28704]: Fix create_unix_server to support Path-like objects (PEP 519).
- bpo-28720 [https://bugs.python.org/issue?@action=redirect&bpo=28720]: Add collections.abc.AsyncGenerator.

Documentation

- bpo-28513 [https://bugs.python.org/issue?@action=redirect&bpo=28513]: Documented command-line interface of zipfile.

Tests

- bpo-28666 [https://bugs.python.org/issue?@action=redirect&bpo=28666]: Now test.support.rmtree is able to remove unwritable or unreadable directories.
- bpo-23839 [https://bugs.python.org/issue?@action=redirect&bpo=23839]: Various caches now are cleared before running every test file.

Build

- bpo-10656 [https://bugs.python.org/issue?@action=redirect&bpo=10656]: Fix out-of-tree building on AIX. Patch by Tristan Carel and Michael Haubenwallner.
- bpo-26359 [https://bugs.python.org/issue?@action=redirect&bpo=26359]: Rename –with-optimiations to –enable-optimizations.
- bpo-28676 [https://bugs.python.org/issue?@action=redirect&bpo=28676]: Prevent missing ‘getentropy’ declaration warning on macOS. Patch by Gareth Rees.

Python 3.6.0 beta 3

Release date: 2016-10-31

Core and Builtins

- bpo-28128 [https://bugs.python.org/issue?@action=redirect&bpo=28128]: Deprecation warning for invalid str and byte escape sequences now prints better information about where the error occurs. Patch by Serhiy Storchaka and Eric Smith.

- bpo-28509 [https://bugs.python.org/issue?@action=redirect&bpo=28509]: dict.update() no longer allocate unnecessary large memory.
- bpo-28426 [https://bugs.python.org/issue?@action=redirect&bpo=28426]: Fixed potential crash in PyUnicode_AsDecodedObject() in debug build.
- bpo-28517 [https://bugs.python.org/issue?@action=redirect&bpo=28517]: Fixed of-by-one error in the peephole optimizer that caused keeping unreachable code.
- bpo-28214 [https://bugs.python.org/issue?@action=redirect&bpo=28214]: Improved exception reporting for problematic __set_name__ attributes.
- bpo-23782 [https://bugs.python.org/issue?@action=redirect&bpo=23782]: Fixed possible memory leak in _PyTraceback_Add() and exception loss in PyTraceBack_Here().
- bpo-28471 [https://bugs.python.org/issue?@action=redirect&bpo=28471]: Fix “Python memory allocator called without holding the GIL” crash in socket.setblocking.

Library

- bpo-27517 [https://bugs.python.org/issue?@action=redirect&bpo=27517]: LZMA compressor and decompressor no longer raise exceptions if given empty data twice. Patch by Benjamin Fogle.
- bpo-28549 [https://bugs.python.org/issue?@action=redirect&bpo=28549]: Fixed segfault in curses’s addch() with ncurses6.
- bpo-28449 [https://bugs.python.org/issue?@action=redirect&bpo=28449]: tarfile.open() with mode “r” or “r:” now tries to open a tar file with compression before trying to open it without compression. Otherwise it had 50% chance failed with ignore_zeros=True.

- bpo-23262 [https://bugs.python.org/issue?@action=redirect&bpo=23262]: The webbrowser module now supports Firefox 36+ and derived browsers. Based on patch by Oleg Broytman.
- bpo-27939 [https://bugs.python.org/issue?@action=redirect&bpo=27939]: Fixed bugs in tkinter.ttk.LabeledScale and tkinter.Scale caused by representing the scale as float value internally in Tk. tkinter.IntVar now works if float value is set to underlying Tk variable.
- bpo-18844 [https://bugs.python.org/issue?@action=redirect&bpo=18844]: The various ways of specifying weights for random.choices() now produce the same result sequences.
- bpo-28255 [https://bugs.python.org/issue?@action=redirect&bpo=28255]: calendar.TextCalendar().prmonth() no longer prints a space at the start of new line after printing a month’s calendar. Patch by Xiang Zhang.
- bpo-20491 [https://bugs.python.org/issue?@action=redirect&bpo=20491]: The textwrap.TextWrapper class now honors non-breaking spaces. Based on patch by Kaarle Ritvanen.
- bpo-28353 [https://bugs.python.org/issue?@action=redirect&bpo=28353]: os.fwalk() no longer fails on broken links.
- bpo-28430 [https://bugs.python.org/issue?@action=redirect&bpo=28430]: Fix iterator of C implemented asyncio.Future doesn’t accept non-None value is passed to it.send(val).
- bpo-27025 [https://bugs.python.org/issue?@action=redirect&bpo=27025]: Generated names for Tkinter widgets now start by the “!” prefix for readability.
- bpo-25464 [https://bugs.python.org/issue?@action=redirect&bpo=25464]: Fixed HList.header_exists() in

tkinter.tix module by addin a workaround to Tix library bug.
- bpo-28488 [https://bugs.python.org/issue?@action=redirect&bpo=28488]: shutil.make_archive() no longer adds entry “./” to ZIP archive.
- bpo-25953 [https://bugs.python.org/issue?@action=redirect&bpo=25953]: re.sub() now raises an error for invalid numerical group reference in replacement template even if the pattern is not found in the string. Error message for invalid group reference now includes the group index and the position of the reference. Based on patch by SilentGhost.
- bpo-18219 [https://bugs.python.org/issue?@action=redirect&bpo=18219]: Optimize csv.DictWriter for large number of columns. Patch by Mariatta Wijaya.
- bpo-28448 [https://bugs.python.org/issue?@action=redirect&bpo=28448]: Fix C implemented asyncio.Future didn’t work on Windows.
- bpo-28480 [https://bugs.python.org/issue?@action=redirect&bpo=28480]: Fix error building socket module when multithreading is disabled.
- bpo-24452 [https://bugs.python.org/issue?@action=redirect&bpo=24452]: Make webbrowser support Chrome on Mac OS X.
- bpo-20766 [https://bugs.python.org/issue?@action=redirect&bpo=20766]: Fix references leaked by pdb in the handling of SIGINT handlers.
- bpo-28492 [https://bugs.python.org/issue?@action=redirect&bpo=28492]: Fix how StopIteration exception is raised in _asyncio.Future.
- bpo-28500 [https://bugs.python.org/issue?@action=redirect&bpo=28500]: Fix asyncio to handle async gens GC from another thread.
- bpo-26923 [https://bugs.python.org/issue?@action=redirect&bpo=26923]: Fix asyncio.Gather to refuse

being cancelled once all children are done. Patch by Johannes Ebke.
- bpo-26796 [https://bugs.python.org/issue?@action=redirect&bpo=26796]: Don’t configure the number of workers for default threadpool executor. Initial patch by Hans Lawrenz.
- bpo-28544 [https://bugs.python.org/issue?@action=redirect&bpo=28544]: Implement asyncio.Task in C.

Windows

- bpo-28522 [https://bugs.python.org/issue?@action=redirect&bpo=28522]: Fixes mishandled buffer reallocation in getpathp.c

Build

- bpo-28444 [https://bugs.python.org/issue?@action=redirect&bpo=28444]: Fix missing extensions modules when cross compiling.
- bpo-28208 [https://bugs.python.org/issue?@action=redirect&bpo=28208]: Update Windows build and OS X installers to use SQLite 3.14.2.
- bpo-28248 [https://bugs.python.org/issue?@action=redirect&bpo=28248]: Update Windows build and OS X installers to use OpenSSL 1.0.2j.

Tests

- bpo-26944 [https://bugs.python.org/issue?@action=redirect&bpo=26944]: Fix test_posix for Android where ‘id -G’ is entirely wrong or missing the effective gid.
- bpo-28409 [https://bugs.python.org/issue?@action=redirect&bpo=28409]: regrtest: fix the parser of command line arguments.

Python 3.6.0 beta 2

Release date: 2016-10-10

Core and Builtins

- bpo-28183 [https://bugs.python.org/issue?@action=redirect&bpo=28183]: Optimize and cleanup dict iteration.
- bpo-26081 [https://bugs.python.org/issue?@action=redirect&bpo=26081]: Added C implementation of asyncio.Future. Original patch by Yury Selivanov.
- bpo-28379 [https://bugs.python.org/issue?@action=redirect&bpo=28379]: Added sanity checks and tests for PyUnicode_CopyCharacters(). Patch by Xiang Zhang.
- bpo-28376 [https://bugs.python.org/issue?@action=redirect&bpo=28376]: The type of long range iterator is now registered as Iterator. Patch by Oren Milman.
- bpo-28376 [https://bugs.python.org/issue?@action=redirect&bpo=28376]: Creating instances of range_iterator by calling range_iterator type now is deprecated. Patch by Oren Milman.
- bpo-28376 [https://bugs.python.org/issue?@action=redirect&bpo=28376]: The constructor of range_iterator now checks that step is not 0. Patch by Oren Milman.
- bpo-26906 [https://bugs.python.org/issue?@action=redirect&bpo=26906]: Resolving special methods of uninitialized type now causes implicit initialization of the type instead of a fail.
- bpo-18287 [https://bugs.python.org/issue?@action=redirect&bpo=18287]: PyType_Ready() now checks that tp_name is not NULL. Original patch by Niklas Koep.
- bpo-24098 [https://bugs.python.org/issue?@action=redirect&bpo=24098]: Fixed possible crash when AST

is changed in process of compiling it.

- bpo-28201 [https://bugs.python.org/issue?@action=redirect&bpo=28201]: Dict reduces possibility of 2nd conflict in hash table when hashes have same lower bits.
- bpo-28350 [https://bugs.python.org/issue?@action=redirect&bpo=28350]: String constants with null character no longer interned.
- bpo-26617 [https://bugs.python.org/issue?@action=redirect&bpo=26617]: Fix crash when GC runs during weakref callbacks.
- bpo-27942 [https://bugs.python.org/issue?@action=redirect&bpo=27942]: String constants now interned recursively in tuples and frozensets.
- bpo-21578 [https://bugs.python.org/issue?@action=redirect&bpo=21578]: Fixed misleading error message when ImportError called with invalid keyword args.
- bpo-28203 [https://bugs.python.org/issue?@action=redirect&bpo=28203]: Fix incorrect type in complex(1.0, {2:3}) error message. Patch by Soumya Sharma.
- bpo-28086 [https://bugs.python.org/issue?@action=redirect&bpo=28086]: Single var-positional argument of tuple subtype was passed unscathed to the C-defined function. Now it is converted to exact tuple.
- bpo-28214 [https://bugs.python.org/issue?@action=redirect&bpo=28214]: Now __set_name__ is looked up on the class instead of the instance.
- bpo-27955 [https://bugs.python.org/issue?@action=redirect&bpo=27955]: Fallback on reading /dev/urandom device when the getrandom() syscall fails with EPERM, for example when blocked by SECCOMP.
- bpo-28192 [https://bugs.python.org/issue?@action=redirect&bpo=28192]: Don’t import readline in isolated mode.

- Upgrade internal unicode databases to Unicode version 9.0.0.
- bpo-28131 [https://bugs.python.org/issue?@action=redirect&bpo=28131]: Fix a regression in zipimport's compile_source(). zipimport should use the same optimization level as the interpreter.
- bpo-28126 [https://bugs.python.org/issue?@action=redirect&bpo=28126]: Replace Py_MEMCPY with memcpy(). Visual Studio can properly optimize memcpy().
- bpo-28120 [https://bugs.python.org/issue?@action=redirect&bpo=28120]: Fix dict.pop() for split dictionary when trying to remove a "pending key" (Not yet inserted in split-table). Patch by Xiang Zhang.
- bpo-26182 [https://bugs.python.org/issue?@action=redirect&bpo=26182]: Raise DeprecationWarning when async and await keywords are used as variable/attribute/class/function name.

Library

- bpo-27998 [https://bugs.python.org/issue?@action=redirect&bpo=27998]: Fixed bytes path support in os.scandir() on Windows. Patch by Eryk Sun.
- bpo-28317 [https://bugs.python.org/issue?@action=redirect&bpo=28317]: The disassembler now decodes FORMAT_VALUE argument.
- bpo-26293 [https://bugs.python.org/issue?@action=redirect&bpo=26293]: Fixed writing ZIP files that starts not from the start of the file. Offsets in ZIP file now are relative to the start of the archive in conforming to the specification.
- bpo-28380 [https://bugs.python.org/issue?@action=redirect&bpo=28380]: unittest.mock Mock autospec functions now properly support assert_called, assert_not_called, and assert_called_once.

- bpo-27181 [https://bugs.python.org/issue?@action=redirect&bpo=27181]: remove statistics.geometric_mean and defer until 3.7.
- bpo-28229 [https://bugs.python.org/issue?@action=redirect&bpo=28229]: lzma module now supports pathlib.
- bpo-28321 [https://bugs.python.org/issue?@action=redirect&bpo=28321]: Fixed writing non-BMP characters with binary format in plistlib.
- bpo-28225 [https://bugs.python.org/issue?@action=redirect&bpo=28225]: bz2 module now supports pathlib. Initial patch by Ethan Furman.
- bpo-28227 [https://bugs.python.org/issue?@action=redirect&bpo=28227]: gzip now supports pathlib. Patch by Ethan Furman.
- bpo-27358 [https://bugs.python.org/issue?@action=redirect&bpo=27358]: Optimized merging var-keyword arguments and improved error message when passing a non-mapping as a var-keyword argument.
- bpo-28257 [https://bugs.python.org/issue?@action=redirect&bpo=28257]: Improved error message when passing a non-iterable as a var-positional argument. Added opcode BUILD_TUPLE_UNPACK_WITH_CALL.
- bpo-28322 [https://bugs.python.org/issue?@action=redirect&bpo=28322]: Fixed possible crashes when unpickle itertools objects from incorrect pickle data. Based on patch by John Leitch.
- bpo-28228 [https://bugs.python.org/issue?@action=redirect&bpo=28228]: imghdr now supports pathlib.
- bpo-28226 [https://bugs.python.org/issue?@action=redirect&bpo=28226]: compileall now supports pathlib.
- bpo-28314 [https://bugs.python.org/issue?@action=redirect&bpo=28314]: Fix function declaration (C flags) for the getiterator() method of xml.etree.ElementTree.Element.

- bpo-28148 [https://bugs.python.org/issue?@action=redirect&bpo=28148]: Stop using localtime() and gmtime() in the time module. Introduced platform independent _PyTime_localtime API that is similar to POSIX localtime_r, but available on all platforms. Patch by Ed Schouten.
- bpo-28253 [https://bugs.python.org/issue?@action=redirect&bpo=28253]: Fixed calendar functions for extreme months: 0001-01 and 9999-12. Methods itermonthdays() and itermonthdays2() are reimplemented so that they don’t call itermonthdates() which can cause datetime.date under/overflow.
- bpo-28275 [https://bugs.python.org/issue?@action=redirect&bpo=28275]: Fixed possible use after free in the decompress() methods of the LZMADecompressor and BZ2Decompressor classes. Original patch by John Leitch.
- bpo-27897 [https://bugs.python.org/issue?@action=redirect&bpo=27897]: Fixed possible crash in sqlite3.Connection.create_collation() if pass invalid string-like object as a name. Patch by Xiang Zhang.
- bpo-18844 [https://bugs.python.org/issue?@action=redirect&bpo=18844]: random.choices() now has k as a keyword-only argument to improve the readability of common cases and come into line with the signature used in other languages.
- bpo-18893 [https://bugs.python.org/issue?@action=redirect&bpo=18893]: Fix invalid exception handling in Lib/ctypes/macholib/dyld.py. Patch by Madison May.
- bpo-27611 [https://bugs.python.org/issue?@action=redirect&bpo=27611]: Fixed support of default root window in the tkinter.tix module. Added the master parameter in the DisplayStyle constructor.
- bpo-27348 [https://bugs.python.org/issue?@action=redirect&bpo=27348]: In the traceback module, restore the formatting of exception messages like

“Exception: None”. This fixes a regression introduced in 3.5a2.

- bpo-25651 [https://bugs.python.org/issue?@action=redirect&bpo=25651]: Allow false values to be used for msg parameter of subTest().
- bpo-27778 [https://bugs.python.org/issue?@action=redirect&bpo=27778]: Fix a memory leak in os.getrandom() when the getrandom() is interrupted by a signal and a signal handler raises a Python exception.
- bpo-28200 [https://bugs.python.org/issue?@action=redirect&bpo=28200]: Fix memory leak on Windows in the os module (fix path_converter() function).
- bpo-25400 [https://bugs.python.org/issue?@action=redirect&bpo=25400]: RobotFileParser now correctly returns default values for crawl_delay and request_rate. Initial patch by Peter Wirtz.
- bpo-27932 [https://bugs.python.org/issue?@action=redirect&bpo=27932]: Prevent memory leak in win32_ver().
- Fix UnboundLocalError in socket._sendfile_use_sendfile.
- bpo-28075 [https://bugs.python.org/issue?@action=redirect&bpo=28075]: Check for ERROR_ACCESS_DENIED in Windows implementation of os.stat(). Patch by Eryk Sun.
- bpo-22493 [https://bugs.python.org/issue?@action=redirect&bpo=22493]: Warning message emitted by using inline flags in the middle of regular expression now contains a (truncated) regex pattern. Patch by Tim Graham.
- bpo-25270 [https://bugs.python.org/issue?@action=redirect&bpo=25270]: Prevent codecs.escape_encode() from raising SystemError when an empty bytestring is passed.
- bpo-28181 [https://bugs.python.org/issue?@action=redirect&bpo=28181]: Get antigravity over HTTPS. Patch by Kaartic Sivaraam.

- bpo-25895 [https://bugs.python.org/issue?@action=redirect&bpo=25895]: Enable WebSocket URL schemes in urllib.parse.urljoin. Patch by Gergely Imreh and Markus Holtermann.
- bpo-28114 [https://bugs.python.org/issue?@action=redirect&bpo=28114]: Fix a crash in parse_envlist() when env contains byte strings. Patch by Eryk Sun.
- bpo-27599 [https://bugs.python.org/issue?@action=redirect&bpo=27599]: Fixed buffer overrun in binascii.b2a_qp() and binascii.a2b_qp().
- bpo-27906 [https://bugs.python.org/issue?@action=redirect&bpo=27906]: Fix socket accept exhaustion during high TCP traffic. Patch by Kevin Conway.
- bpo-28174 [https://bugs.python.org/issue?@action=redirect&bpo=28174]: Handle when SO_REUSEPORT isn’t properly supported. Patch by Seth Michael Larson.
- bpo-26654 [https://bugs.python.org/issue?@action=redirect&bpo=26654]: Inspect functools.partial in asyncio.Handle.__repr__. Patch by iceboy.
- bpo-26909 [https://bugs.python.org/issue?@action=redirect&bpo=26909]: Fix slow pipes IO in asyncio. Patch by INADA Naoki.
- bpo-28176 [https://bugs.python.org/issue?@action=redirect&bpo=28176]: Fix callbacks race in asyncio.SelectorLoop.sock_connect.
- bpo-27759 [https://bugs.python.org/issue?@action=redirect&bpo=27759]: Fix selectors incorrectly retain invalid file descriptors. Patch by Mark Williams.
- bpo-28368 [https://bugs.python.org/issue?@action=redirect&bpo=28368]: Refuse monitoring processes if the child watcher has no loop attached. Patch by Vincent Michel.
- bpo-28369 [https://bugs.python.org/issue?@action=redirect&bpo=28369]: Raise RuntimeError when transport’s FD is used with add_reader, add_writer, etc.

- bpo-28370 [https://bugs.python.org/issue?@action=redirect&bpo=28370]: Speedup asyncio.StreamReader.readexactly. Patch by Коренберг Марк.
- bpo-28371 [https://bugs.python.org/issue?@action=redirect&bpo=28371]: Deprecate passing asyncio.Handles to run_in_executor.
- bpo-28372 [https://bugs.python.org/issue?@action=redirect&bpo=28372]: Fix asyncio to support formatting of non-python coroutines.
- bpo-28399 [https://bugs.python.org/issue?@action=redirect&bpo=28399]: Remove UNIX socket from FS before binding. Patch by Коренберг Марк.
- bpo-27972 [https://bugs.python.org/issue?@action=redirect&bpo=27972]: Prohibit Tasks to await on themselves.

Windows

- bpo-28402 [https://bugs.python.org/issue?@action=redirect&bpo=28402]: Adds signed catalog files for stdlib on Windows.
- bpo-28333 [https://bugs.python.org/issue?@action=redirect&bpo=28333]: Enables Unicode for ps1/ps2 and input() prompts. (Patch by Eryk Sun)
- bpo-28251 [https://bugs.python.org/issue?@action=redirect&bpo=28251]: Improvements to help manuals on Windows.
- bpo-28110 [https://bugs.python.org/issue?@action=redirect&bpo=28110]: launcher.msi has different product codes between 32-bit and 64-bit
- bpo-28161 [https://bugs.python.org/issue?@action=redirect&bpo=28161]: Opening CON for write access fails
- bpo-28162 [https://bugs.python.org/issue?@action=redirect&bpo=28162]: WindowsConsoleIO readall()

fails if first line starts with Ctrl+Z

- bpo-28163 [https://bugs.python.org/issue?@action=redirect&bpo=28163]: WindowsConsoleIO fileno() passes wrong flags to _open_osfhandle
- bpo-28164 [https://bugs.python.org/issue?@action=redirect&bpo=28164]: _PyIO_get_console_type fails for various paths
- bpo-28137 [https://bugs.python.org/issue?@action=redirect&bpo=28137]: Renames Windows path file to ._pth
- bpo-28138 [https://bugs.python.org/issue?@action=redirect&bpo=28138]: Windows ._pth file should allow import site

C API

- bpo-28426 [https://bugs.python.org/issue?@action=redirect&bpo=28426]: Deprecated undocumented functions PyUnicode_AsEncodedObject(), PyUnicode_AsDecodedObject(), PyUnicode_AsDecodedUnicode() and PyUnicode_AsEncodedUnicode().

Build

- bpo-28258 [https://bugs.python.org/issue?@action=redirect&bpo=28258]: Fixed build with Estonian locale (python-config and distclean targets in Makefile). Patch by Arfrever Frehtes Taifersar Arahesis.
- bpo-26661 [https://bugs.python.org/issue?@action=redirect&bpo=26661]: setup.py now detects system libffi with multiarch wrapper.
- bpo-15819 [https://bugs.python.org/issue?@action=redirect&bpo=15819]: Remove redundant include search directory option for building outside the source tree.

Tests

- bpo-28217 [https://bugs.python.org/issue?@action=redirect&bpo=28217]: Adds _testconsole module to test console input.

Python 3.6.0 beta 1

Release date: 2016-09-12

Core and Builtins

- bpo-23722 [https://bugs.python.org/issue?@action=redirect&bpo=23722]: The __class__ cell used by zero-argument super() is now initialized from type.__new__ rather than __build_class__, so class methods relying on that will now work correctly when called from metaclass methods during class creation. Patch by Martin Teichmann.
- bpo-25221 [https://bugs.python.org/issue?@action=redirect&bpo=25221]: Fix corrupted result from PyLong_FromLong(0) when Python is compiled with NSMALLPOSINTS = 0.
- bpo-27080 [https://bugs.python.org/issue?@action=redirect&bpo=27080]: Implement formatting support for **PEP 515** [https://peps.python.org/pep-0515/]. Initial patch by Chris Angelico.
- bpo-27199 [https://bugs.python.org/issue?@action=redirect&bpo=27199]: In tarfile, expose copyfileobj bufsize to improve throughput. Patch by Jason Fried.
- bpo-27948 [https://bugs.python.org/issue?@action=redirect&bpo=27948]: In f-strings, only allow backslashes inside the braces (where the expressions are). This is a breaking change from the 3.6 alpha releases, where backslashes are allowed anywhere in an

f-string. Also, require that expressions inside f-strings be enclosed within literal braces, and not escapes like `f'\x7b"hi"\x7d'`.

- bpo-28046 [https://bugs.python.org/issue?@action=redirect&bpo=28046]: Remove platform-specific directories from sys.path.
- bpo-28071 [https://bugs.python.org/issue?@action=redirect&bpo=28071]: Add early-out for differencing from an empty set.
- bpo-25758 [https://bugs.python.org/issue?@action=redirect&bpo=25758]: Prevents zipimport from unnecessarily encoding a filename (patch by Eryk Sun)
- bpo-25856 [https://bugs.python.org/issue?@action=redirect&bpo=25856]: The __module__ attribute of extension classes and functions now is interned. This leads to more compact pickle data with protocol 4.
- bpo-27213 [https://bugs.python.org/issue?@action=redirect&bpo=27213]: Rework CALL_FUNCTION* opcodes to produce shorter and more efficient bytecode. Patch by Demur Rumed, design by Serhiy Storchaka, reviewed by Serhiy Storchaka and Victor Stinner.
- bpo-26331 [https://bugs.python.org/issue?@action=redirect&bpo=26331]: Implement tokenizing support for **PEP 515** [https://peps.python.org/pep-0515/]. Patch by Georg Brandl.
- bpo-27999 [https://bugs.python.org/issue?@action=redirect&bpo=27999]: Make “global after use” a SyntaxError, and ditto for nonlocal. Patch by Ivan Levkivskyi.
- bpo-28003 [https://bugs.python.org/issue?@action=redirect&bpo=28003]: Implement **PEP 525** [https://peps.python.org/pep-0525/] – Asynchronous Generators.
- bpo-27985 [https://bugs.python.org/issue?@action=redirect&bpo=27985]: Implement **PEP 526**

[https://peps.python.org/pep-0526/] – Syntax for Variable Annotations. Patch by Ivan Levkivskyi.

- bpo-26058 [https://bugs.python.org/issue?@action=redirect&bpo=26058]: Add a new private version to the builtin dict type, incremented at each dictionary creation and at each dictionary change. Implementation of the PEP 509.
- bpo-27364 [https://bugs.python.org/issue?@action=redirect&bpo=27364]: A backslash-character pair that is not a valid escape sequence now generates a DeprecationWarning. Patch by Emanuel Barry.
- bpo-27350 [https://bugs.python.org/issue?@action=redirect&bpo=27350]: `dict` implementation is changed like PyPy. It is more compact and preserves insertion order. (Concept developed by Raymond Hettinger and patch by Inada Naoki.)
- bpo-27911 [https://bugs.python.org/issue?@action=redirect&bpo=27911]: Remove unnecessary error checks in `exec_builtin_or_dynamic()`.
- bpo-27078 [https://bugs.python.org/issue?@action=redirect&bpo=27078]: Added BUILD_STRING opcode. Optimized f-strings evaluation.
- bpo-17884 [https://bugs.python.org/issue?@action=redirect&bpo=17884]: Python now requires systems with inttypes.h and stdint.h
- bpo-27961 [https://bugs.python.org/issue?@action=redirect&bpo=27961]: Require platforms to support `long long`. Python hasn’t compiled without `long long` for years, so this is basically a formality.
- bpo-27355 [https://bugs.python.org/issue?@action=redirect&bpo=27355]: Removed support for Windows CE. It was never finished, and Windows CE is no longer a relevant platform for Python.
- Implement **PEP 523** [https://peps.python.org/pep-0523/].
- bpo-27870 [https://bugs.python.org/issue?@action=redirect&bpo=27870]: A left shift of zero by a large

integer no longer attempts to allocate large amounts of memory.

- bpo-25402 [https://bugs.python.org/issue?@action=redirect&bpo=25402]: In int-to-decimal-string conversion, improve the estimate of the intermediate memory required, and remove an unnecessarily strict overflow check. Patch by Serhiy Storchaka.
- bpo-27214 [https://bugs.python.org/issue?@action=redirect&bpo=27214]: In long_invert, be more careful about modifying object returned by long_add, and remove an unnecessary check for small longs. Thanks Oren Milman for analysis and patch.
- bpo-27506 [https://bugs.python.org/issue?@action=redirect&bpo=27506]: Support passing the bytes/bytearray.translate() "delete" argument by keyword.
- bpo-27812 [https://bugs.python.org/issue?@action=redirect&bpo=27812]: Properly clear out a generator's frame's backreference to the generator to prevent crashes in frame.clear().
- bpo-27811 [https://bugs.python.org/issue?@action=redirect&bpo=27811]: Fix a crash when a coroutine that has not been awaited is finalized with warnings-as-errors enabled.
- bpo-27587 [https://bugs.python.org/issue?@action=redirect&bpo=27587]: Fix another issue found by PVS-Studio: Null pointer check after use of 'def' in _PyState_AddModule(). Initial patch by Christian Heimes.
- bpo-27792 [https://bugs.python.org/issue?@action=redirect&bpo=27792]: The modulo operation applied to `bool` and other `int` subclasses now always returns an `int`. Previously the return type depended on the input values. Patch by Xiang Zhang.
- bpo-26984 [https://bugs.python.org/issue?@action=redirect&bpo=26984]: int() now always returns an instance of exact int.

- bpo-25604 [https://bugs.python.org/issue?@action=redirect&bpo=25604]: Fix a minor bug in integer true division; this bug could potentially have caused off-by-one-ulp results on platforms with unreliable ldexp implementations.
- bpo-24254 [https://bugs.python.org/issue?@action=redirect&bpo=24254]: Make class definition namespace ordered by default.
- bpo-27662 [https://bugs.python.org/issue?@action=redirect&bpo=27662]: Fix an overflow check in `List_New`: the original code was checking against `Py_SIZE_MAX` instead of the correct upper bound of `Py_SSIZE_T_MAX`. Patch by Xiang Zhang.
- bpo-27782 [https://bugs.python.org/issue?@action=redirect&bpo=27782]: Multi-phase extension module import now correctly allows the `m_methods` field to be used to add module level functions to instances of non-module types returned from `Py_create_mod`. Patch by Xiang Zhang.
- bpo-27936 [https://bugs.python.org/issue?@action=redirect&bpo=27936]: The round() function accepted a second None argument for some types but not for others. Fixed the inconsistency by accepting None for all numeric types.
- bpo-27487 [https://bugs.python.org/issue?@action=redirect&bpo=27487]: Warn if a submodule argument to “python -m” or runpy.run_module() is found in sys.modules after parent packages are imported, but before the submodule is executed.
- bpo-27157 [https://bugs.python.org/issue?@action=redirect&bpo=27157]: Make only type() itself accept the one-argument form. Patch by Eryk Sun and Emanuel Barry.
- bpo-27558 [https://bugs.python.org/issue?@action=redirect&bpo=27558]: Fix a SystemError in the implementation of “raise” statement. In a brand new

thread, raise a RuntimeError since there is no active exception to reraise. Patch written by Xiang Zhang.

- bpo-28008 [https://bugs.python.org/issue?@action=redirect&bpo=28008]: Implement **PEP 530** [https://peps.python.org/pep-0530/] – asynchronous comprehensions.
- bpo-27942 [https://bugs.python.org/issue?@action=redirect&bpo=27942]: Fix memory leak in codeobject.c

Library

- bpo-28732 [https://bugs.python.org/issue?@action=redirect&bpo=28732]: Fix crash in os.spawnv() with no elements in args
- bpo-28485 [https://bugs.python.org/issue?@action=redirect&bpo=28485]: Always raise ValueError for negative compileall.compile_dir(workers=…) parameter, even when multithreading is unavailable.
- bpo-28037 [https://bugs.python.org/issue?@action=redirect&bpo=28037]: Use sqlite3_get_autocommit() instead of setting Connection->inTransaction manually.
- bpo-25283 [https://bugs.python.org/issue?@action=redirect&bpo=25283]: Attributes tm_gmtoff and tm_zone are now available on all platforms in the return values of time.localtime() and time.gmtime().
- bpo-24454 [https://bugs.python.org/issue?@action=redirect&bpo=24454]: Regular expression match object groups are now accessible using __getitem__. “mo[x]” is equivalent to “mo.group(x)”.
- bpo-10740 [https://bugs.python.org/issue?@action=redirect&bpo=10740]: sqlite3 no longer implicitly commit an open transaction before DDL statements.
- bpo-17941 [https://bugs.python.org/issue?@action=redirect&bpo=17941]: Add a *module* parameter to collections.namedtuple().

- bpo-22493 [https://bugs.python.org/issue?@action=redirect&bpo=22493]: Inline flags now should be used only at the start of the regular expression. Deprecation warning is emitted if uses them in the middle of the regular expression.
- bpo-26885 [https://bugs.python.org/issue?@action=redirect&bpo=26885]: xmlrpc now supports unmarshalling additional data types used by Apache XML-RPC implementation for numerics and None.
- bpo-28070 [https://bugs.python.org/issue?@action=redirect&bpo=28070]: Fixed parsing inline verbose flag in regular expressions.
- bpo-19500 [https://bugs.python.org/issue?@action=redirect&bpo=19500]: Add client-side SSL session resumption to the ssl module.
- bpo-28022 [https://bugs.python.org/issue?@action=redirect&bpo=28022]: Deprecate ssl-related arguments in favor of SSLContext. The deprecation include manual creation of SSLSocket and certfile/keyfile (or similar) in ftplib, httplib, imaplib, smtplib, poplib and urllib.
- bpo-28043 [https://bugs.python.org/issue?@action=redirect&bpo=28043]: SSLContext has improved default settings: OP_NO_SSLv2, OP_NO_SSLv3, OP_NO_COMPRESSION, OP_CIPHER_SERVER_PREFERENCE, OP_SINGLE_DH_USE, OP_SINGLE_ECDH_USE and HIGH ciphers without MD5.
- bpo-24693 [https://bugs.python.org/issue?@action=redirect&bpo=24693]: Changed some RuntimeError’s in the zipfile module to more appropriate types. Improved some error messages and debugging output.
- bpo-17909 [https://bugs.python.org/issue?@action=redirect&bpo=17909]: `json.load` and `json.loads` now support binary input encoded as UTF-8, UTF-16 or UTF-32. Patch by Serhiy Storchaka.

- bpo-27137 [https://bugs.python.org/issue?@action=redirect&bpo=27137]: the pure Python fallback implementation of `functools.partial` now matches the behaviour of its accelerated C counterpart for subclassing, pickling and text representation purposes. Patch by Emanuel Barry and Serhiy Storchaka.
- Fix possible integer overflows and crashes in the mmap module with unusual usage patterns.
- bpo-1703178 [https://bugs.python.org/issue?@action=redirect&bpo=1703178]: Fix the ability to pass the –link-objects option to the distutils build_ext command.
- bpo-28019 [https://bugs.python.org/issue?@action=redirect&bpo=28019]: itertools.count() no longer rounds non-integer step in range between 1.0 and 2.0 to 1.
- bpo-18401 [https://bugs.python.org/issue?@action=redirect&bpo=18401]: Pdb now supports the 'readrc' keyword argument to control whether .pdbrc files should be read. Patch by Martin Matusiak and Sam Kimbrel.
- bpo-25969 [https://bugs.python.org/issue?@action=redirect&bpo=25969]: Update the lib2to3 grammar to handle the unpacking generalizations added in 3.5.
- bpo-14977 [https://bugs.python.org/issue?@action=redirect&bpo=14977]: mailcap now respects the order of the lines in the mailcap files ("first match"), as required by RFC 1542. Patch by Michael Lazar.
- bpo-28082 [https://bugs.python.org/issue?@action=redirect&bpo=28082]: Convert re flag constants to IntFlag.
- bpo-28025 [https://bugs.python.org/issue?@action=redirect&bpo=28025]: Convert all ssl module constants to IntEnum and IntFlags. SSLContext properties now return flags and enums.
- bpo-23591 [https://bugs.python.org/issue?@action=redirect&bpo=23591]: Add Flag, IntFlag, and auto() to enum module.

- bpo-433028 [https://bugs.python.org/issue?@action=redirect&bpo=433028]: Added support of modifier spans in regular expressions.
- bpo-24594 [https://bugs.python.org/issue?@action=redirect&bpo=24594]: Validates persist parameter when opening MSI database
- bpo-17582 [https://bugs.python.org/issue?@action=redirect&bpo=17582]: xml.etree.ElementTree nows preserves whitespaces in attributes (Patch by Duane Griffin. Reviewed and approved by Stefan Behnel.)
- bpo-28047 [https://bugs.python.org/issue?@action=redirect&bpo=28047]: Fixed calculation of line length used for the base64 CTE in the new email policies.
- bpo-27576 [https://bugs.python.org/issue?@action=redirect&bpo=27576]: Fix call order in OrderedDict.__init__().
- email.generator.DecodedGenerator now supports the policy keyword.
- bpo-28027 [https://bugs.python.org/issue?@action=redirect&bpo=28027]: Remove undocumented modules from `Lib/plat-*`: IN, CDROM, DLFCN, TYPES, CDIO, and STROPTS.
- bpo-27445 [https://bugs.python.org/issue?@action=redirect&bpo=27445]: Don’t pass str(_charset) to MIMEText.set_payload(). Patch by Claude Paroz.
- bpo-24277 [https://bugs.python.org/issue?@action=redirect&bpo=24277]: The new email API is no longer provisional, and the docs have been reorganized and rewritten to emphasize the new API.
- bpo-22450 [https://bugs.python.org/issue?@action=redirect&bpo=22450]: urllib now includes an `Accept: */*` header among the default headers. This makes the results of REST API requests more consistent and predictable especially when proxy servers are involved.
- lib2to3.pgen3.driver.load_grammar() now creates a stable cache file between runs given the same

Grammar.txt input regardless of the hash randomization setting.
- bpo-28005 [https://bugs.python.org/issue?@action=redirect&bpo=28005]: Allow ImportErrors in encoding implementation to propagate.
- bpo-26667 [https://bugs.python.org/issue?@action=redirect&bpo=26667]: Support path-like objects in importlib.util.
- bpo-27570 [https://bugs.python.org/issue?@action=redirect&bpo=27570]: Avoid zero-length memcpy() etc calls with null source pointers in the “ctypes” and “array” modules.
- bpo-22233 [https://bugs.python.org/issue?@action=redirect&bpo=22233]: Break email header lines *only* on the RFC specified CR and LF characters, not on arbitrary unicode line breaks. This also fixes a bug in HTTP header parsing.
- bpo-27331 [https://bugs.python.org/issue?@action=redirect&bpo=27331]: The email.mime classes now all accept an optional policy keyword.
- bpo-27988 [https://bugs.python.org/issue?@action=redirect&bpo=27988]: Fix email iter_attachments incorrect mutation of payload list.
- bpo-16113 [https://bugs.python.org/issue?@action=redirect&bpo=16113]: Add SHA-3 and SHAKE support to hashlib module.
- Eliminate a tautological-pointer-compare warning in _scproxy.c.
- bpo-27776 [https://bugs.python.org/issue?@action=redirect&bpo=27776]: The `os.urandom()` function does now block on Linux 3.17 and newer until the system urandom entropy pool is initialized to increase the security. This change is part of the **PEP 524** [https://peps.python.org/pep-0524/].
- bpo-27778 [https://bugs.python.org/issue?@action=redirect&bpo=27778]: Expose the Linux `getrandom()`

syscall as a new `os.getrandom()` function. This change is part of the **PEP 524** [https://peps.python.org/pep-0524/].
- bpo-27691 [https://bugs.python.org/issue?@action=redirect&bpo=27691]: Fix ssl module's parsing of GEN_RID subject alternative name fields in X.509 certs.
- bpo-18844 [https://bugs.python.org/issue?@action=redirect&bpo=18844]: Add random.choices().
- bpo-25761 [https://bugs.python.org/issue?@action=redirect&bpo=25761]: Improved error reporting about truncated pickle data in C implementation of unpickler. UnpicklingError is now raised instead of AttributeError and ValueError in some cases.
- bpo-26798 [https://bugs.python.org/issue?@action=redirect&bpo=26798]: Add BLAKE2 (blake2b and blake2s) to hashlib.
- bpo-26032 [https://bugs.python.org/issue?@action=redirect&bpo=26032]: Optimized globbing in pathlib by using os.scandir(); it is now about 1.5–4 times faster.
- bpo-25596 [https://bugs.python.org/issue?@action=redirect&bpo=25596]: Optimized glob() and iglob() functions in the glob module; they are now about 3–6 times faster.
- bpo-27928 [https://bugs.python.org/issue?@action=redirect&bpo=27928]: Add scrypt (password-based key derivation function) to hashlib module (requires OpenSSL 1.1.0).
- bpo-27850 [https://bugs.python.org/issue?@action=redirect&bpo=27850]: Remove 3DES from ssl module's default cipher list to counter measure sweet32 attack (**CVE 2016-2183** [https://www.cve.org/CVERecord?id=CVE-2016-2183]).
- bpo-27766 [https://bugs.python.org/issue?@action=redirect&bpo=27766]: Add ChaCha20 Poly1305 to ssl module's default cipher list. (Required OpenSSL 1.1.0 or LibreSSL).

- bpo-25387 [https://bugs.python.org/issue?@action=redirect&bpo=25387]: Check return value of winsound.MessageBeep.
- bpo-27866 [https://bugs.python.org/issue?@action=redirect&bpo=27866]: Add SSLContext.get_ciphers() method to get a list of all enabled ciphers.
- bpo-27744 [https://bugs.python.org/issue?@action=redirect&bpo=27744]: Add AF_ALG (Linux Kernel crypto) to socket module.
- bpo-26470 [https://bugs.python.org/issue?@action=redirect&bpo=26470]: Port ssl and hashlib module to OpenSSL 1.1.0.
- bpo-11620 [https://bugs.python.org/issue?@action=redirect&bpo=11620]: Fix support for SND_MEMORY in winsound.PlaySound. Based on a patch by Tim Lesher.
- bpo-11734 [https://bugs.python.org/issue?@action=redirect&bpo=11734]: Add support for IEEE 754 half-precision floats to the struct module. Based on a patch by Eli Stevens.
- bpo-27919 [https://bugs.python.org/issue?@action=redirect&bpo=27919]: Deprecated `extra_path` distribution option in distutils packaging.
- bpo-23229 [https://bugs.python.org/issue?@action=redirect&bpo=23229]: Add new `cmath` constants: `cmath.inf` and `cmath.nan` to match `math.inf` and `math.nan`, and also `cmath.infj` and `cmath.nanj` to match the format used by complex repr.
- bpo-27842 [https://bugs.python.org/issue?@action=redirect&bpo=27842]: The csv.DictReader now returns rows of type OrderedDict. (Contributed by Steve Holden.)
- Remove support for passing a file descriptor to os.access. It never worked but previously didn’t raise.
- bpo-12885 [https://bugs.python.org/issue?@action=redirect&bpo=12885]: Fix error when distutils encounters symlink.

- bpo-27881 [https://bugs.python.org/issue?@action=redirect&bpo=27881]: Fixed possible bugs when setting sqlite3.Connection.isolation_level. Based on patch by Xiang Zhang.
- bpo-27861 [https://bugs.python.org/issue?@action=redirect&bpo=27861]: Fixed a crash in sqlite3.Connection.cursor() when a factory creates not a cursor. Patch by Xiang Zhang.
- bpo-19884 [https://bugs.python.org/issue?@action=redirect&bpo=19884]: Avoid spurious output on OS X with Gnu Readline.
- bpo-27706 [https://bugs.python.org/issue?@action=redirect&bpo=27706]: Restore deterministic behavior of random.Random().seed() for string seeds using seeding version 1. Allows sequences of calls to random() to exactly match those obtained in Python 2. Patch by Nofar Schnider.
- bpo-10513 [https://bugs.python.org/issue?@action=redirect&bpo=10513]: Fix a regression in Connection.commit(). Statements should not be reset after a commit.
- bpo-12319 [https://bugs.python.org/issue?@action=redirect&bpo=12319]: Chunked transfer encoding support added to http.client.HTTPConnection requests. The urllib.request.AbstractHTTPHandler class does not enforce a Content-Length header any more. If a HTTP request has a file or iterable body, but no Content-Length header, the library now falls back to use chunked transfer-encoding.
- A new version of typing.py from https://github.com/python/typing: - Collection (only for 3.6) (bpo-27598 [https://bugs.python.org/issue?@action=redirect&bpo=27598]) - Add FrozenSet to __all__ (upstream #261) - fix crash in _get_type_vars() (upstream #259) - Remove the dict constraint in ForwardRef._eval_type (upstream #252)

- bpo-27832 [https://bugs.python.org/issue?@action=redirect&bpo=27832]: Make `_normalize` parameter to `Fraction` constructor keyword-only, so that `Fraction(2, 3, 4)` now raises `TypeError`.
- bpo-27539 [https://bugs.python.org/issue?@action=redirect&bpo=27539]: Fix unnormalised `Fraction.__pow__` result in the case of negative exponent and negative base.
- bpo-21718 [https://bugs.python.org/issue?@action=redirect&bpo=21718]: cursor.description is now available for queries using CTEs.
- bpo-27819 [https://bugs.python.org/issue?@action=redirect&bpo=27819]: In distutils sdists, simply produce the “gztar” (gzipped tar format) distributions on all platforms unless “formats” is supplied.
- bpo-2466 [https://bugs.python.org/issue?@action=redirect&bpo=2466]: posixpath.ismount now correctly recognizes mount points which the user does not have permission to access.
- bpo-9998 [https://bugs.python.org/issue?@action=redirect&bpo=9998]: On Linux, ctypes.util.find_library now looks in LD_LIBRARY_PATH for shared libraries.
- bpo-27573 [https://bugs.python.org/issue?@action=redirect&bpo=27573]: exit message for code.interact is now configurable.
- bpo-27930 [https://bugs.python.org/issue?@action=redirect&bpo=27930]: Improved behaviour of logging.handlers.QueueListener. Thanks to Paulo Andrade and Petr Viktorin for the analysis and patch.
- bpo-6766 [https://bugs.python.org/issue?@action=redirect&bpo=6766]: Distributed reference counting added to multiprocessing to support nesting of shared values / proxy objects.
- bpo-21201 [https://bugs.python.org/issue?@action=redirect&bpo=21201]: Improves readability of multiprocessing error message. Thanks to Wojciech Walczak for patch.

- asyncio: Add set_protocol / get_protocol to Transports.
- bpo-27456 [https://bugs.python.org/issue?@action=redirect&bpo=27456]: asyncio: Set TCP_NODELAY by default.

IDLE

- bpo-15308 [https://bugs.python.org/issue?@action=redirect&bpo=15308]: Add ‘interrupt execution’ (^C) to Shell menu. Patch by Roger Serwy, updated by Bayard Randel.
- bpo-27922 [https://bugs.python.org/issue?@action=redirect&bpo=27922]: Stop IDLE tests from ‘flashing’ gui widgets on the screen.
- bpo-27891 [https://bugs.python.org/issue?@action=redirect&bpo=27891]: Consistently group and sort imports within idlelib modules.
- bpo-17642 [https://bugs.python.org/issue?@action=redirect&bpo=17642]: add larger font sizes for classroom projection.
- Add version to title of IDLE help window.
- bpo-25564 [https://bugs.python.org/issue?@action=redirect&bpo=25564]: In section on IDLE – console differences, mention that using exec means that __builtins__ is defined for each statement.
- bpo-27821 [https://bugs.python.org/issue?@action=redirect&bpo=27821]: Fix 3.6.0a3 regression that prevented custom key sets from being selected when no custom theme was defined.

C API

- bpo-26900 [https://bugs.python.org/issue?@action=redirect&bpo=26900]: Excluded underscored names and other private API from limited API.

- bpo-26027 [https://bugs.python.org/issue?@action=redirect&bpo=26027]: Add support for path-like objects in PyUnicode_FSConverter() & PyUnicode_FSDecoder().

Tests

- bpo-27427 [https://bugs.python.org/issue?@action=redirect&bpo=27427]: Additional tests for the math module. Patch by Francisco Couzo.
- bpo-27953 [https://bugs.python.org/issue?@action=redirect&bpo=27953]: Skip math and cmath tests that fail on OS X 10.4 due to a poor libm implementation of tan.
- bpo-26040 [https://bugs.python.org/issue?@action=redirect&bpo=26040]: Improve test_math and test_cmath coverage and rigour. Patch by Jeff Allen.
- bpo-27787 [https://bugs.python.org/issue?@action=redirect&bpo=27787]: Call gc.collect() before checking each test for “dangling threads”, since the dangling threads are weak references.

Build

- bpo-27566 [https://bugs.python.org/issue?@action=redirect&bpo=27566]: Fix clean target in freeze makefile (patch by Lisa Roach)
- bpo-27705 [https://bugs.python.org/issue?@action=redirect&bpo=27705]: Update message in validate_ucrtbase.py
- bpo-27976 [https://bugs.python.org/issue?@action=redirect&bpo=27976]: Deprecate building _ctypes with the bundled copy of libffi on non-OSX UNIX platforms.
- bpo-27983 [https://bugs.python.org/issue?@action=redirect&bpo=27983]: Cause lack of llvm-profdata tool

when using clang as required for PGO linking to be a configure time error rather than make time when `--with-optimizations` is enabled. Also improve our ability to find the llvm-profdata tool on MacOS and some Linuxes.

- bpo-21590 [https://bugs.python.org/issue?@action=redirect&bpo=21590]: Support for DTrace and SystemTap probes.
- bpo-26307 [https://bugs.python.org/issue?@action=redirect&bpo=26307]: The profile-opt build now applies PGO to the built-in modules.
- bpo-26359 [https://bugs.python.org/issue?@action=redirect&bpo=26359]: Add the –with-optimizations flag to turn on LTO and PGO build support when available.
- bpo-27917 [https://bugs.python.org/issue?@action=redirect&bpo=27917]: Set platform triplets for Android builds.
- bpo-25825 [https://bugs.python.org/issue?@action=redirect&bpo=25825]: Update references to the $(LIBPL) installation path on AIX. This path was changed in 3.2a4.
- Update OS X installer to use SQLite 3.14.1 and XZ 5.2.2.
- bpo-21122 [https://bugs.python.org/issue?@action=redirect&bpo=21122]: Fix LTO builds on OS X.
- bpo-17128 [https://bugs.python.org/issue?@action=redirect&bpo=17128]: Build OS X installer with a private copy of OpenSSL. Also provide a sample Install Certificates command script to install a set of root certificates from the third-party certifi module.

Tools/Demos

- bpo-27952 [https://bugs.python.org/issue?@action=redirect&bpo=27952]: Get Tools/scripts/fixcid.py working with Python 3 and the current “re” module, avoid invalid Python backslash escapes, and fix a bug parsing escaped C quote signs.

Windows

- bpo-28065 [https://bugs.python.org/issue?@action=redirect&bpo=28065]: Update xz dependency to 5.2.2 and build it from source.
- bpo-25144 [https://bugs.python.org/issue?@action=redirect&bpo=25144]: Ensures TargetDir is set before continuing with custom install.
- bpo-1602 [https://bugs.python.org/issue?@action=redirect&bpo=1602]: Windows console doesn’t input or print Unicode (PEP 528)
- bpo-27781 [https://bugs.python.org/issue?@action=redirect&bpo=27781]: Change file system encoding on Windows to UTF-8 (PEP 529)
- bpo-27731 [https://bugs.python.org/issue?@action=redirect&bpo=27731]: Opt-out of MAX_PATH on Windows 10
- bpo-6135 [https://bugs.python.org/issue?@action=redirect&bpo=6135]: Adds encoding and errors parameters to subprocess.
- bpo-27959 [https://bugs.python.org/issue?@action=redirect&bpo=27959]: Adds oem encoding, alias ansi to mbcs, move aliasmbcs to codec lookup.
- bpo-27982 [https://bugs.python.org/issue?@action=redirect&bpo=27982]: The functions of the winsound module now accept keyword arguments.
- bpo-20366 [https://bugs.python.org/issue?@action=redirect&bpo=20366]: Build full text search support into SQLite on Windows.
- bpo-27756 [https://bugs.python.org/issue?@action=redirect&bpo=27756]: Adds new icons for Python files and processes on Windows. Designs by Cherry Wang.
- bpo-27883 [https://bugs.python.org/issue?@action=redirect&bpo=27883]: Update sqlite to 3.14.1.0 on

Windows.

Python 3.6.0 alpha 4

Release date: 2016-08-15

Core and Builtins

- bpo-27704 [https://bugs.python.org/issue?@action=redirect&bpo=27704]: Optimized creating bytes and bytearray from byte-like objects and iterables. Speed up to 3 times for short objects. Original patch by Naoki Inada.
- bpo-26823 [https://bugs.python.org/issue?@action=redirect&bpo=26823]: Large sections of repeated lines in tracebacks are now abbreviated as “[Previous line repeated {count} more times]” by the builtin traceback rendering. Patch by Emanuel Barry.
- bpo-27574 [https://bugs.python.org/issue?@action=redirect&bpo=27574]: Decreased an overhead of parsing keyword arguments in functions implemented with using Argument Clinic.
- bpo-22557 [https://bugs.python.org/issue?@action=redirect&bpo=22557]: Now importing already imported modules is up to 2.5 times faster.
- bpo-17596 [https://bugs.python.org/issue?@action=redirect&bpo=17596]: Include <wincrypt.h> to help with Min GW building.
- bpo-17599 [https://bugs.python.org/issue?@action=redirect&bpo=17599]: On Windows, rename the privately defined REPARSE_DATA_BUFFER structure to avoid conflicting with the definition from Min GW.
- bpo-27507 [https://bugs.python.org/issue?@action=redirect&bpo=27507]: Add integer overflow check in bytearray.extend(). Patch by Xiang Zhang.

- bpo-27581 [https://bugs.python.org/issue?@action=redirect&bpo=27581]: Don’t rely on wrapping for overflow check in PySequence_Tuple(). Patch by Xiang Zhang.
- bpo-1621 [https://bugs.python.org/issue?@action=redirect&bpo=1621]: Avoid signed integer overflow in list and tuple operations. Patch by Xiang Zhang.
- bpo-27419 [https://bugs.python.org/issue?@action=redirect&bpo=27419]: Standard __import__() no longer look up “__import__” in globals or builtins for importing submodules or “from import”. Fixed a crash if raise a warning about unabling to resolve package from __spec__ or __package__.
- bpo-27083 [https://bugs.python.org/issue?@action=redirect&bpo=27083]: Respect the PYTHONCASEOK environment variable under Windows.
- bpo-27514 [https://bugs.python.org/issue?@action=redirect&bpo=27514]: Make having too many statically nested blocks a SyntaxError instead of SystemError.
- bpo-27366 [https://bugs.python.org/issue?@action=redirect&bpo=27366]: Implemented **PEP 487** [https://peps.python.org/pep-0487/] (Simpler customization of class creation). Upon subclassing, the __init_subclass__ classmethod is called on the base class. Descriptors are initialized with __set_name__ after class creation.

Library

- bpo-26027 [https://bugs.python.org/issue?@action=redirect&bpo=26027]: Add **PEP 519** [https://peps.python.org/pep-0519/]/__fspath__() support to the os and os.path modules. Includes code from Jelle Zijlstra. (See also: bpo-27524 [https://bugs.python.org/issue?@action=redirect&bpo=27524])

- bpo-27598 [https://bugs.python.org/issue?@action=redirect&bpo=27598]: Add Collections to collections.abc. Patch by Ivan Levkivskyi, docs by Neil Girdhar.
- bpo-25958 [https://bugs.python.org/issue?@action=redirect&bpo=25958]: Support “anti-registration” of special methods from various ABCs, like __hash__, __iter__ or __len__. All these (and several more) can be set to None in an implementation class and the behavior will be as if the method is not defined at all. (Previously, this mechanism existed only for __hash__, to make mutable classes unhashable.) Code contributed by Andrew Barnert and Ivan Levkivskyi.
- bpo-16764 [https://bugs.python.org/issue?@action=redirect&bpo=16764]: Support keyword arguments to zlib.decompress(). Patch by Xiang Zhang.
- bpo-27736 [https://bugs.python.org/issue?@action=redirect&bpo=27736]: Prevent segfault after interpreter re-initialization due to ref count problem introduced in code for bpo-27038 [https://bugs.python.org/issue?@action=redirect&bpo=27038] in 3.6.0a3. Patch by Xiang Zhang.
- bpo-25628 [https://bugs.python.org/issue?@action=redirect&bpo=25628]: The *verbose* and *rename* parameters for collections.namedtuple are now keyword-only.
- bpo-12345 [https://bugs.python.org/issue?@action=redirect&bpo=12345]: Add mathematical constant tau to math and cmath. See also **PEP 628** [https://peps.python.org/pep-0628/].
- bpo-26823 [https://bugs.python.org/issue?@action=redirect&bpo=26823]: traceback.StackSummary.format now abbreviates large sections of repeated lines as “[Previous line repeated {count} more times]” (this change then further affects

other traceback display operations in the module). Patch by Emanuel Barry.

- bpo-27664 [https://bugs.python.org/issue?@action=redirect&bpo=27664]: Add to concurrent.futures.thread.ThreadPoolExecutor() the ability to specify a thread name prefix.
- bpo-27181 [https://bugs.python.org/issue?@action=redirect&bpo=27181]: Add geometric_mean and harmonic_mean to statistics module.
- bpo-27573 [https://bugs.python.org/issue?@action=redirect&bpo=27573]: code.interact now prints an message when exiting.
- bpo-6422 [https://bugs.python.org/issue?@action=redirect&bpo=6422]: Add autorange method to timeit.Timer objects.
- bpo-27773 [https://bugs.python.org/issue?@action=redirect&bpo=27773]: Correct some memory management errors server_hostname in _ssl.wrap_socket().
- bpo-26750 [https://bugs.python.org/issue?@action=redirect&bpo=26750]: unittest.mock.create_autospec() now works properly for subclasses of property() and other data descriptors. Removes the never publicly used, never documented unittest.mock.DescriptorTypes tuple.
- bpo-26754 [https://bugs.python.org/issue?@action=redirect&bpo=26754]: Undocumented support of general bytes-like objects as path in compile() and similar functions is now deprecated.
- bpo-26800 [https://bugs.python.org/issue?@action=redirect&bpo=26800]: Undocumented support of general bytes-like objects as paths in os functions is now deprecated.
- bpo-26981 [https://bugs.python.org/issue?@action=redirect&bpo=26981]: Add _order_ compatibility shim to enum.Enum for Python 2/3 code bases.

- bpo-27661 [https://bugs.python.org/issue?@action=redirect&bpo=27661]: Added tzinfo keyword argument to datetime.combine.
- In the curses module, raise an error if window.getstr() or window.instr() is passed a negative value.
- bpo-27783 [https://bugs.python.org/issue?@action=redirect&bpo=27783]: Fix possible usage of uninitialized memory in operator.methodcaller.
- bpo-27774 [https://bugs.python.org/issue?@action=redirect&bpo=27774]: Fix possible Py_DECREF on unowned object in _sre.
- bpo-27760 [https://bugs.python.org/issue?@action=redirect&bpo=27760]: Fix possible integer overflow in binascii.b2a_qp.
- bpo-27758 [https://bugs.python.org/issue?@action=redirect&bpo=27758]: Fix possible integer overflow in the _csv module for large record lengths.
- bpo-27568 [https://bugs.python.org/issue?@action=redirect&bpo=27568]: Prevent HTTPoxy attack (**CVE 2016-1000110** [https://www.cve.org/CVERecord?id=CVE-2016-1000110]). Ignore the HTTP_PROXY variable when REQUEST_METHOD environment is set, which indicates that the script is in CGI mode.
- bpo-7063 [https://bugs.python.org/issue?@action=redirect&bpo=7063]: Remove dead code from the “array” module’s slice handling. Patch by Chuck.
- bpo-27656 [https://bugs.python.org/issue?@action=redirect&bpo=27656]: Do not assume sched.h defines any SCHED_* constants.
- bpo-27130 [https://bugs.python.org/issue?@action=redirect&bpo=27130]: In the “zlib” module, fix handling of large buffers (typically 4 GiB) when compressing and decompressing. Previously, inputs were limited to 4 GiB, and compression and decompression operations did not properly handle results of 4 GiB.

- bpo-24773 [https://bugs.python.org/issue?@action=redirect&bpo=24773]: Implemented **PEP 495** [https://peps.python.org/pep-0495/] (Local Time Disambiguation).
- Expose the EPOLLEXCLUSIVE constant (when it is defined) in the select module.
- bpo-27567 [https://bugs.python.org/issue?@action=redirect&bpo=27567]: Expose the EPOLLRDHUP and POLLRDHUP constants in the select module.
- bpo-1621 [https://bugs.python.org/issue?@action=redirect&bpo=1621]: Avoid signed int negation overflow in the “audioop” module.
- bpo-27533 [https://bugs.python.org/issue?@action=redirect&bpo=27533]: Release GIL in nt._isdir
- bpo-17711 [https://bugs.python.org/issue?@action=redirect&bpo=17711]: Fixed unpickling by the persistent ID with protocol 0. Original patch by Alexandre Vassalotti.
- bpo-27522 [https://bugs.python.org/issue?@action=redirect&bpo=27522]: Avoid an unintentional reference cycle in email.feedparser.
- bpo-27512 [https://bugs.python.org/issue?@action=redirect&bpo=27512]: Fix a segfault when os.fspath() called an __fspath__() method that raised an exception. Patch by Xiang Zhang.

IDLE

- bpo-27714 [https://bugs.python.org/issue?@action=redirect&bpo=27714]: text_textview and test_autocomplete now pass when re-run in the same process. This occurs when test_idle fails when run with the -w option but without -jn. Fix warning from test_config.
- bpo-27621 [https://bugs.python.org/issue?@action=redirect&bpo=27621]: Put query response validation error messages in the query box itself instead of in a

separate messagebox. Redo tests to match. Add Mac OSX refinements. Original patch by Mark Roseman.

- bpo-27620 [https://bugs.python.org/issue?@action=redirect&bpo=27620]: Escape key now closes Query box as cancelled.
- bpo-27609 [https://bugs.python.org/issue?@action=redirect&bpo=27609]: IDLE: tab after initial whitespace should tab, not autocomplete. This fixes problem with writing docstrings at least twice indented.
- bpo-27609 [https://bugs.python.org/issue?@action=redirect&bpo=27609]: Explicitly return None when there are also non-None returns. In a few cases, reverse a condition and eliminate a return.
- bpo-25507 [https://bugs.python.org/issue?@action=redirect&bpo=25507]: IDLE no longer runs buggy code because of its tkinter imports. Users must include the same imports required to run directly in Python.
- bpo-27173 [https://bugs.python.org/issue?@action=redirect&bpo=27173]: Add ‘IDLE Modern Unix’ to the built-in key sets. Make the default key set depend on the platform. Add tests for the changes to the config module.
- bpo-27452 [https://bugs.python.org/issue?@action=redirect&bpo=27452]: add line counter and crc to IDLE configHandler test dump.

Tests

- bpo-25805 [https://bugs.python.org/issue?@action=redirect&bpo=25805]: Skip a test in test_pkgutil as needed that doesn’t work when `__name__ == __main__`. Patch by SilentGhost.
- bpo-27472 [https://bugs.python.org/issue?@action=redirect&bpo=27472]: Add test.support.unix_shell as the path to the default shell.

- bpo-27369 [https://bugs.python.org/issue?@action=redirect&bpo=27369]: In test_pyexpat, avoid testing an error message detail that changed in Expat 2.2.0.
- bpo-27594 [https://bugs.python.org/issue?@action=redirect&bpo=27594]: Prevent assertion error when running test_ast with coverage enabled: ensure code object has a valid first line number. Patch suggested by Ivan Levkivskyi.

Windows

- bpo-27647 [https://bugs.python.org/issue?@action=redirect&bpo=27647]: Update bundled Tcl/Tk to 8.6.6.
- bpo-27610 [https://bugs.python.org/issue?@action=redirect&bpo=27610]: Adds **PEP 514** [https://peps.python.org/pep-0514/] metadata to Windows installer
- bpo-27469 [https://bugs.python.org/issue?@action=redirect&bpo=27469]: Adds a shell extension to the launcher so that drag and drop works correctly.
- bpo-27309 [https://bugs.python.org/issue?@action=redirect&bpo=27309]: Enables proper Windows styles in python[w].exe manifest.

Build

- bpo-27713 [https://bugs.python.org/issue?@action=redirect&bpo=27713]: Suppress spurious build warnings when updating importlib’s bootstrap files. Patch by Xiang Zhang
- bpo-25825 [https://bugs.python.org/issue?@action=redirect&bpo=25825]: Correct the references to Modules/python.exp, which is required on AIX. The references were accidentally changed in 3.5.0a1.
- bpo-27453 [https://bugs.python.org/issue?@action=redirect&bpo=27453]: CPP invocation in configure

must use CPPFLAGS. Patch by Chi Hsuan Yen.
- bpo-27641 [https://bugs.python.org/issue?@action=redirect&bpo=27641]: The configure script now inserts comments into the makefile to prevent the pgen and _freeze_importlib executables from being cross-compiled.
- bpo-26662 [https://bugs.python.org/issue?@action=redirect&bpo=26662]: Set PYTHON_FOR_GEN in configure as the Python program to be used for file generation during the build.
- bpo-10910 [https://bugs.python.org/issue?@action=redirect&bpo=10910]: Avoid C++ compilation errors on FreeBSD and OS X. Also update FreedBSD version checks for the original ctype UTF-8 workaround.

Python 3.6.0 alpha 3

Release date: 2016-07-11

Security

- bpo-27278 [https://bugs.python.org/issue?@action=redirect&bpo=27278]: Fix os.urandom() implementation using getrandom() on Linux. Truncate size to INT_MAX and loop until we collected enough random bytes, instead of casting a directly Py_ssize_t to int.
- bpo-22636 [https://bugs.python.org/issue?@action=redirect&bpo=22636]: Avoid shell injection problems with ctypes.util.find_library().

Core and Builtins

- bpo-27473 [https://bugs.python.org/issue?@action=redirect&bpo=27473]: Fixed possible integer overflow in bytes and bytearray concatenations. Patch by Xiang Zhang.

- bpo-23034 [https://bugs.python.org/issue?@action=redirect&bpo=23034]: The output of a special Python build with defined COUNT_ALLOCS, SHOW_ALLOC_COUNT or SHOW_TRACK_COUNT macros is now off by default. It can be re-enabled using the “-X showalloccount” option. It now outputs to stderr instead of stdout.
- bpo-27443 [https://bugs.python.org/issue?@action=redirect&bpo=27443]: __length_hint__() of bytearray iterators no longer return a negative integer for a resized bytearray.
- bpo-27007 [https://bugs.python.org/issue?@action=redirect&bpo=27007]: The fromhex() class methods of bytes and bytearray subclasses now return an instance of corresponding subclass.

Library

- bpo-26844 [https://bugs.python.org/issue?@action=redirect&bpo=26844]: Fix error message for imp.find_module() to refer to ‘path’ instead of ‘name’. Patch by Lev Maximov.
- bpo-23804 [https://bugs.python.org/issue?@action=redirect&bpo=23804]: Fix SSL zero-length recv() calls to not block and not raise an error about unclean EOF.
- bpo-27466 [https://bugs.python.org/issue?@action=redirect&bpo=27466]: Change time format returned by http.cookie.time2netscape, confirming the netscape cookie format and making it consistent with documentation.
- bpo-21708 [https://bugs.python.org/issue?@action=redirect&bpo=21708]: Deprecated dbm.dumb behavior that differs from common dbm behavior:

creating a database in ‘r’ and ‘w’ modes and modifying a database in ‘r’ mode.

- bpo-26721 [https://bugs.python.org/issue?@action=redirect&bpo=26721]: Change the socketserver.StreamRequestHandler.wfile attribute to implement BufferedIOBase. In particular, the write() method no longer does partial writes.
- bpo-22115 [https://bugs.python.org/issue?@action=redirect&bpo=22115]: Added methods trace_add, trace_remove and trace_info in the tkinter.Variable class. They replace old methods trace_variable, trace, trace_vdelete and trace_vinfo that use obsolete Tcl commands and might not work in future versions of Tcl. Fixed old tracing methods: trace_vdelete() with wrong mode no longer break tracing, trace_vinfo() now always returns a list of pairs of strings, tracing in the “u” mode now works.
- bpo-26243 [https://bugs.python.org/issue?@action=redirect&bpo=26243]: Only the level argument to zlib.compress() is keyword argument now. The first argument is positional-only.
- bpo-27038 [https://bugs.python.org/issue?@action=redirect&bpo=27038]: Expose the DirEntry type as os.DirEntry. Code patch by Jelle Zijlstra.
- bpo-27186 [https://bugs.python.org/issue?@action=redirect&bpo=27186]: Update os.fspath()/PyOS_FSPath() to check the return value of __fspath__() to be either str or bytes.
- bpo-18726 [https://bugs.python.org/issue?@action=redirect&bpo=18726]: All optional parameters of the dump(), dumps(), load() and loads() functions and JSONEncoder and JSONDecoder class constructors in the json module are now keyword-only.
- bpo-27319 [https://bugs.python.org/issue?@action=redirect&bpo=27319]: Methods selection_set(), selection_add(), selection_remove() and

selection_toggle() of ttk.TreeView now allow passing multiple items as multiple arguments instead of passing them as a tuple. Deprecated undocumented ability of calling the selection() method with arguments.

- bpo-27079 [https://bugs.python.org/issue?@action=redirect&bpo=27079]: Fixed curses.ascii functions isblank(), iscntrl() and ispunct().
- bpo-27294 [https://bugs.python.org/issue?@action=redirect&bpo=27294]: Numerical state in the repr for Tkinter event objects is now represented as a combination of known flags.
- bpo-27177 [https://bugs.python.org/issue?@action=redirect&bpo=27177]: Match objects in the re module now support index-like objects as group indices. Based on patches by Jeroen Demeyer and Xiang Zhang.
- bpo-26754 [https://bugs.python.org/issue?@action=redirect&bpo=26754]: Some functions (compile() etc) accepted a filename argument encoded as an iterable of integers. Now only strings and byte-like objects are accepted.
- bpo-26536 [https://bugs.python.org/issue?@action=redirect&bpo=26536]: socket.ioctl now supports SIO_LOOPBACK_FAST_PATH. Patch by Daniel Stokes.
- bpo-27048 [https://bugs.python.org/issue?@action=redirect&bpo=27048]: Prevents distutils failing on Windows when environment variables contain non-ASCII characters
- bpo-27330 [https://bugs.python.org/issue?@action=redirect&bpo=27330]: Fixed possible leaks in the ctypes module.
- bpo-27238 [https://bugs.python.org/issue?@action=redirect&bpo=27238]: Got rid of bare excepts in the turtle module. Original patch by Jelle Zijlstra.
- bpo-27122 [https://bugs.python.org/issue?@action=redirect&bpo=27122]: When an exception is raised within the context being managed by a

contextlib.ExitStack() and one of the exit stack generators catches and raises it in a chain, do not re-raise the original exception when exiting, let the new chained one through. This avoids the **PEP 479** [https://peps.python.org/pep-0479/] bug described in issue25782.
- bpo-16864 [https://bugs.python.org/issue?@action=redirect&bpo=16864]: sqlite3.Cursor.lastrowid now supports REPLACE statement. Initial patch by Alex LordThorsen.
- bpo-26386 [https://bugs.python.org/issue?@action=redirect&bpo=26386]: Fixed ttk.TreeView selection operations with item id’s containing spaces.
- bpo-8637 [https://bugs.python.org/issue?@action=redirect&bpo=8637]: Honor a pager set by the env var MANPAGER (in preference to one set by the env var PAGER).
- bpo-16182 [https://bugs.python.org/issue?@action=redirect&bpo=16182]: Fix various functions in the “readline” module to use the locale encoding, and fix get_begidx() and get_endidx() to return code point indexes.
- bpo-27392 [https://bugs.python.org/issue?@action=redirect&bpo=27392]: Add loop.connect_accepted_socket(). Patch by Jim Fulton.

IDLE

- bpo-27477 [https://bugs.python.org/issue?@action=redirect&bpo=27477]: IDLE search dialogs now use ttk widgets.
- bpo-27173 [https://bugs.python.org/issue?@action=redirect&bpo=27173]: Add ‘IDLE Modern Unix’ to the built-in key sets. Make the default key set depend on the platform. Add tests for the changes to the config module.

- bpo-27452 [https://bugs.python.org/issue?@action=redirect&bpo=27452]: make command line “idle-test> python test_help.py” work. __file__ is relative when python is started in the file’s directory.
- bpo-27452 [https://bugs.python.org/issue?@action=redirect&bpo=27452]: add line counter and crc to IDLE configHandler test dump.
- bpo-27380 [https://bugs.python.org/issue?@action=redirect&bpo=27380]: IDLE: add query.py with base Query dialog and ttk widgets. Module had subclasses SectionName, ModuleName, and HelpSource, which are used to get information from users by configdialog and file =>Load Module. Each subclass has itw own validity checks. Using ModuleName allows users to edit bad module names instead of starting over. Add tests and delete the two files combined into the new one.
- bpo-27372 [https://bugs.python.org/issue?@action=redirect&bpo=27372]: Test_idle no longer changes the locale.
- bpo-27365 [https://bugs.python.org/issue?@action=redirect&bpo=27365]: Allow non-ascii chars in IDLE NEWS.txt, for contributor names.
- bpo-27245 [https://bugs.python.org/issue?@action=redirect&bpo=27245]: IDLE: Cleanly delete custom themes and key bindings. Previously, when IDLE was started from a console or by import, a cascade of warnings was emitted. Patch by Serhiy Storchaka.
- bpo-24137 [https://bugs.python.org/issue?@action=redirect&bpo=24137]: Run IDLE, test_idle, and htest with tkinter default root disabled. Fix code and tests that fail with this restriction. Fix htests to not create a second and redundant root and mainloop.
- bpo-27310 [https://bugs.python.org/issue?@action=redirect&bpo=27310]: Fix IDLE.app failure to launch on OS X due to vestigial import.

C API

- bpo-26754 [https://bugs.python.org/issue?@action=redirect&bpo=26754]: PyUnicode_FSDecoder() accepted a filename argument encoded as an iterable of integers. Now only strings and byte-like objects are accepted.

Build

- bpo-28066 [https://bugs.python.org/issue?@action=redirect&bpo=28066]: Fix the logic that searches build directories for generated include files when building outside the source tree.
- bpo-27442 [https://bugs.python.org/issue?@action=redirect&bpo=27442]: Expose the Android API level that python was built against, in sysconfig.get_config_vars() as 'ANDROID_API_LEVEL'.
- bpo-27434 [https://bugs.python.org/issue?@action=redirect&bpo=27434]: The interpreter that runs the cross-build, found in PATH, must now be of the same feature version (e.g. 3.6) as the source being built.
- bpo-26930 [https://bugs.python.org/issue?@action=redirect&bpo=26930]: Update Windows builds to use OpenSSL 1.0.2h.
- bpo-23968 [https://bugs.python.org/issue?@action=redirect&bpo=23968]: Rename the platform directory from plat-$(MACHDEP) to plat-$(PLATFORM_TRIPLET). Rename the config directory (LIBPL) from config-$(LDVERSION) to config-$(LDVERSION)-$(PLATFORM_TRIPLET). Install the platform specific _sysconfigdata module into the platform directory and rename it to include the ABIFLAGS.
- Don't use largefile support for GNU/Hurd.

Tools/Demos

- bpo-27332 [https://bugs.python.org/issue?@action=redirect&bpo=27332]: Fixed the type of the first argument of module-level functions generated by Argument Clinic. Patch by Petr Viktorin.
- bpo-27418 [https://bugs.python.org/issue?@action=redirect&bpo=27418]: Fixed Tools/importbench/importbench.py.

Documentation

- bpo-19489 [https://bugs.python.org/issue?@action=redirect&bpo=19489]: Moved the search box from the sidebar to the header and footer of each page. Patch by Ammar Askar.
- bpo-27285 [https://bugs.python.org/issue?@action=redirect&bpo=27285]: Update documentation to reflect the deprecation of `pyvenv` and normalize on the term “virtual environment”. Patch by Steve Piercy.

Tests

- bpo-27027 [https://bugs.python.org/issue?@action=redirect&bpo=27027]: Added test.support.is_android that is True when this is an Android build.

Python 3.6.0 alpha 2

Release date: 2016-06-13

Security

- bpo-26556 [https://bugs.python.org/issue?@action=redirect&bpo=26556]: Update expat to 2.1.1, fixes

CVE 2015-1283 [https://www.cve.org/CVERecord?id=CVE-2015-1283].
- Fix TLS stripping vulnerability in smtplib, **CVE 2016-0772** [https://www.cve.org/CVERecord?id=CVE-2016-0772]. Reported by Team Oststrom.
- bpo-26839 [https://bugs.python.org/issue?@action=redirect&bpo=26839]: On Linux, **`os.urandom()`** now calls `getrandom()` with `GRND_NONBLOCK` to fall back on reading `/dev/urandom` if the urandom entropy pool is not initialized yet. Patch written by Colm Buckley.

Core and Builtins

- bpo-27095 [https://bugs.python.org/issue?@action=redirect&bpo=27095]: Simplified MAKE_FUNCTION and removed MAKE_CLOSURE opcodes. Patch by Demur Rumed.
- bpo-27190 [https://bugs.python.org/issue?@action=redirect&bpo=27190]: Raise NotSupportedError if sqlite3 is older than 3.3.1. Patch by Dave Sawyer.
- bpo-27286 [https://bugs.python.org/issue?@action=redirect&bpo=27286]: Fixed compiling BUILD_MAP_UNPACK_WITH_CALL opcode. Calling function with generalized unpacking (PEP 448) and conflicting keyword names could cause undefined behavior.
- bpo-27140 [https://bugs.python.org/issue?@action=redirect&bpo=27140]: Added BUILD_CONST_KEY_MAP opcode.
- bpo-27186 [https://bugs.python.org/issue?@action=redirect&bpo=27186]: Add support for os.PathLike objects to open() (part of **PEP 519** [https://peps.python.org/pep-0519/]).
- bpo-27066 [https://bugs.python.org/issue?@action=redirect&bpo=27066]: Fixed SystemError if a custom

opener (for open()) returns a negative number without setting an exception.

- bpo-26983 [https://bugs.python.org/issue?@action=redirect&bpo=26983]: float() now always return an instance of exact float. The deprecation warning is emitted if __float__ returns an instance of a strict subclass of float. In a future versions of Python this can be an error.
- bpo-27097 [https://bugs.python.org/issue?@action=redirect&bpo=27097]: Python interpreter is now about 7% faster due to optimized instruction decoding. Based on patch by Demur Rumed.
- bpo-26647 [https://bugs.python.org/issue?@action=redirect&bpo=26647]: Python interpreter now uses 16-bit wordcode instead of bytecode. Patch by Demur Rumed.
- bpo-23275 [https://bugs.python.org/issue?@action=redirect&bpo=23275]: Allow assigning to an empty target list in round brackets: () = iterable.
- bpo-27243 [https://bugs.python.org/issue?@action=redirect&bpo=27243]: Update the __aiter__ protocol: instead of returning an awaitable that resolves to an asynchronous iterator, the asynchronous iterator should be returned directly. Doing the former will trigger a PendingDeprecationWarning.

Library

- Comment out socket (SO_REUSEPORT) and posix (O_SHLOCK, O_EXLOCK) constants exposed on the API which are not implemented on GNU/Hurd. They would not work at runtime anyway.
- bpo-27025 [https://bugs.python.org/issue?@action=redirect&bpo=27025]: Generated names for Tkinter widgets are now more meaningful and recognizable.

- bpo-25455 [https://bugs.python.org/issue?@action=redirect&bpo=25455]: Fixed crashes in repr of recursive ElementTree.Element and functools.partial objects.
- bpo-27294 [https://bugs.python.org/issue?@action=redirect&bpo=27294]: Improved repr for Tkinter event objects.
- bpo-20508 [https://bugs.python.org/issue?@action=redirect&bpo=20508]: Improve exception message of IPv{4,6}Network.__getitem__. Patch by Gareth Rees.
- bpo-21386 [https://bugs.python.org/issue?@action=redirect&bpo=21386]: Implement missing IPv4Address.is_global property. It was documented since 07a5610bae9d. Initial patch by Roger Luethi.
- bpo-27029 [https://bugs.python.org/issue?@action=redirect&bpo=27029]: Removed deprecated support of universal newlines mode from ZipFile.open().
- bpo-27030 [https://bugs.python.org/issue?@action=redirect&bpo=27030]: Unknown escapes consisting of `'\'` and an ASCII letter in regular expressions now are errors. The re.LOCALE flag now can be used only with bytes patterns.
- bpo-27186 [https://bugs.python.org/issue?@action=redirect&bpo=27186]: Add os.PathLike support to DirEntry (part of **PEP 519** [https://peps.python.org/pep-0519/]). Initial patch by Jelle Zijlstra.
- bpo-20900 [https://bugs.python.org/issue?@action=redirect&bpo=20900]: distutils register command now decodes HTTP responses correctly. Initial patch by ingrid.
- bpo-27186 [https://bugs.python.org/issue?@action=redirect&bpo=27186]: Add os.PathLike support to pathlib, removing its provisional status (part of PEP 519). Initial patch by Dusty Phillips.
- bpo-27186 [https://bugs.python.org/issue?@action=redirect&bpo=27186]: Add support for os.PathLike

objects to os.fsencode() and os.fsdecode() (part of **PEP 519** [https://peps.python.org/pep-0519/]).

- bpo-27186 [https://bugs.python.org/issue?@action=redirect&bpo=27186]: Introduce os.PathLike and os.fspath() (part of **PEP 519** [https://peps.python.org/pep-0519/]).
- A new version of typing.py provides several new classes and features: @overload outside stubs, Reversible, DefaultDict, Text, ContextManager, Type[], NewType(), TYPE_CHECKING, and numerous bug fixes (note that some of the new features are not yet implemented in mypy or other static analyzers). Also classes for **PEP 492** [https://peps.python.org/pep-0492/] (Awaitable, AsyncIterable, AsyncIterator) have been added (in fact they made it into 3.5.1 but were never mentioned).
- bpo-25738 [https://bugs.python.org/issue?@action=redirect&bpo=25738]: Stop http.server.BaseHTTPRequestHandler.send_error() from sending a message body for 205 Reset Content. Also, don’t send Content header fields in responses that don’t have a body. Patch by Susumu Koshiba.
- bpo-21313 [https://bugs.python.org/issue?@action=redirect&bpo=21313]: Fix the “platform” module to tolerate when sys.version contains truncated build information.
- bpo-23883 [https://bugs.python.org/issue?@action=redirect&bpo=23883]: Added missing APIs to __all__ to match the documented APIs for the following modules: cgi, mailbox, mimetypes, plistlib and smtpd. Patches by Jacek Kołodziej.
- bpo-27164 [https://bugs.python.org/issue?@action=redirect&bpo=27164]: In the zlib module, allow decompressing raw Deflate streams with a predefined zdict. Based on patch by Xiang Zhang.
- bpo-24291 [https://bugs.python.org/issue?@action=redirect&bpo=24291]: Fix

wsgiref.simple_server.WSGIRequestHandler to completely write data to the client. Previously it could do partial writes and truncate data. Also, wsgiref.handler.ServerHandler can now handle stdout doing partial writes, but this is deprecated.

- bpo-21272 [https://bugs.python.org/issue?@action=redirect&bpo=21272]: Use _sysconfigdata.py to initialize distutils.sysconfig.
- bpo-19611 [https://bugs.python.org/issue?@action=redirect&bpo=19611]: **`inspect`** now reports the implicit `.0` parameters generated by the compiler for comprehension and generator expression scopes as if they were positional-only parameters called `implicit0`. Patch by Jelle Zijlstra.
- bpo-26809 [https://bugs.python.org/issue?@action=redirect&bpo=26809]: Add `__all__` to **`string`**. Patch by Emanuel Barry.
- bpo-26373 [https://bugs.python.org/issue?@action=redirect&bpo=26373]: subprocess.Popen.communicate now correctly ignores BrokenPipeError when the child process dies before .communicate() is called in more/all circumstances.
- signal, socket, and ssl module IntEnum constant name lookups now return a consistent name for values having multiple names. Ex: signal.Signals(6) now refers to itself as signal.SIGALRM rather than flipping between that and signal.SIGIOT based on the interpreter’s hash randomization seed.
- bpo-27167 [https://bugs.python.org/issue?@action=redirect&bpo=27167]: Clarify the subprocess.CalledProcessError error message text when the child process died due to a signal.
- bpo-25931 [https://bugs.python.org/issue?@action=redirect&bpo=25931]: Don’t define socketserver.Forking* names on platforms such as Windows that do not support os.fork().

- bpo-21776 [https://bugs.python.org/issue?@action=redirect&bpo=21776]: distutils.upload now correctly handles HTTPError. Initial patch by Claudiu Popa.
- bpo-26526 [https://bugs.python.org/issue?@action=redirect&bpo=26526]: Replace custom parse tree validation in the parser module with a simple DFA validator.
- bpo-27114 [https://bugs.python.org/issue?@action=redirect&bpo=27114]: Fix SSLContext._load_windows_store_certs fails with PermissionError
- bpo-18383 [https://bugs.python.org/issue?@action=redirect&bpo=18383]: Avoid creating duplicate filters when using filterwarnings and simplefilter. Based on patch by Alex Shkop.
- bpo-23026 [https://bugs.python.org/issue?@action=redirect&bpo=23026]: winreg.QueryValueEx() now return an integer for REG_QWORD type.
- bpo-26741 [https://bugs.python.org/issue?@action=redirect&bpo=26741]: subprocess.Popen destructor now emits a ResourceWarning warning if the child process is still running.
- bpo-27056 [https://bugs.python.org/issue?@action=redirect&bpo=27056]: Optimize pickle.load() and pickle.loads(), up to 10% faster to deserialize a lot of small objects.
- bpo-21271 [https://bugs.python.org/issue?@action=redirect&bpo=21271]: New keyword only parameters in reset_mock call.

IDLE

- bpo-5124 [https://bugs.python.org/issue?@action=redirect&bpo=5124]: Paste with text selected now replaces the selection on X11. This matches how paste works on Windows, Mac, most modern Linux apps, and ttk widgets. Original patch by Serhiy Storchaka.

- bpo-24750 [https://bugs.python.org/issue?@action=redirect&bpo=24750]: Switch all scrollbars in IDLE to ttk versions. Where needed, minimal tests are added to cover changes.
- bpo-24759 [https://bugs.python.org/issue?@action=redirect&bpo=24759]: IDLE requires tk 8.5 and availability ttk widgets. Delete now unneeded tk version tests and code for older versions. Add test for IDLE syntax colorizer.
- bpo-27239 [https://bugs.python.org/issue?@action=redirect&bpo=27239]: idlelib.macosx.isXyzTk functions initialize as needed.
- bpo-27262 [https://bugs.python.org/issue?@action=redirect&bpo=27262]: move Aqua unbinding code, which enable context menus, to macosx.
- bpo-24759 [https://bugs.python.org/issue?@action=redirect&bpo=24759]: Make clear in idlelib.idle_test.__init__ that the directory is a private implementation of test.test_idle and tool for maintainers.
- bpo-27196 [https://bugs.python.org/issue?@action=redirect&bpo=27196]: Stop ‘ThemeChanged’ warnings when running IDLE tests. These persisted after other warnings were suppressed in #20567. Apply Serhiy Storchaka’s update_idletasks solution to four test files. Record this additional advice in idle_test/README.txt
- bpo-20567 [https://bugs.python.org/issue?@action=redirect&bpo=20567]: Revise idle_test/README.txt with advice about avoiding tk warning messages from tests. Apply advice to several IDLE tests.
- bpo-24225 [https://bugs.python.org/issue?@action=redirect&bpo=24225]: Update idlelib/README.txt with new file names and event handlers.

- bpo-27156 [https://bugs.python.org/issue?@action=redirect&bpo=27156]: Remove obsolete code not used by IDLE.
- bpo-27117 [https://bugs.python.org/issue?@action=redirect&bpo=27117]: Make colorizer htest and turtledemo work with dark themes. Move code for configuring text widget colors to a new function.
- bpo-24225 [https://bugs.python.org/issue?@action=redirect&bpo=24225]: Rename many `idlelib/*.py` and `idle_test/test_*.py` files. Edit files to replace old names with new names when the old name referred to the module rather than the class it contained. See the issue and IDLE section in What's New in 3.6 for more.
- bpo-26673 [https://bugs.python.org/issue?@action=redirect&bpo=26673]: When tk reports font size as 0, change to size 10. Such fonts on Linux prevented the configuration dialog from opening.
- bpo-21939 [https://bugs.python.org/issue?@action=redirect&bpo=21939]: Add test for IDLE's percolator. Original patch by Saimadhav Heblikar.
- bpo-21676 [https://bugs.python.org/issue?@action=redirect&bpo=21676]: Add test for IDLE's replace dialog. Original patch by Saimadhav Heblikar.
- bpo-18410 [https://bugs.python.org/issue?@action=redirect&bpo=18410]: Add test for IDLE's search dialog. Original patch by Westley Martínez.
- bpo-21703 [https://bugs.python.org/issue?@action=redirect&bpo=21703]: Add test for undo delegator. Patch mostly by Saimadhav Heblikar .
- bpo-27044 [https://bugs.python.org/issue?@action=redirect&bpo=27044]: Add ConfigDialog.remove_var_callbacks to stop memory leaks.
- bpo-23977 [https://bugs.python.org/issue?@action=redirect&bpo=23977]: Add more asserts to test_delegator.

Documentation

- bpo-16484 [https://bugs.python.org/issue?@action=redirect&bpo=16484]: Change the default PYTHONDOCS URL to “https:”, and fix the resulting links to use lowercase. Patch by Sean Rodman, test by Kaushik Nadikuditi.
- bpo-24136 [https://bugs.python.org/issue?@action=redirect&bpo=24136]: Document the new **PEP 448** [https://peps.python.org/pep-0448/] unpacking syntax of 3.5.
- bpo-22558 [https://bugs.python.org/issue?@action=redirect&bpo=22558]: Add remaining doc links to source code for Python-coded modules. Patch by Yoni Lavi.

Tests

- bpo-25285 [https://bugs.python.org/issue?@action=redirect&bpo=25285]: regrtest now uses subprocesses when the -j1 command line option is used: each test file runs in a fresh child process. Before, the -j1 option was ignored.
- bpo-25285 [https://bugs.python.org/issue?@action=redirect&bpo=25285]: Tools/buildbot/test.bat script now uses -j1 by default to run each test file in fresh child process.

Windows

- bpo-27064 [https://bugs.python.org/issue?@action=redirect&bpo=27064]: The py.exe launcher now defaults to Python 3. The Windows launcher `py.exe` no longer prefers an installed Python 2 version over Python 3 by default when used interactively.

- bpo-17500 [https://bugs.python.org/issue?@action=redirect&bpo=17500]: Remove unused and outdated icons. (See also: https://github.com/python/pythondotorg/issues/945)

Build

- bpo-27229 [https://bugs.python.org/issue?@action=redirect&bpo=27229]: Fix the cross-compiling pgen rule for in-tree builds. Patch by Xavier de Gaye.
- bpo-26930 [https://bugs.python.org/issue?@action=redirect&bpo=26930]: Update OS X 10.5+ 32-bit-only installer to build and link with OpenSSL 1.0.2h.

C API

- bpo-27186 [https://bugs.python.org/issue?@action=redirect&bpo=27186]: Add the PyOS_FSPath() function (part of **PEP 519** [https://peps.python.org/pep-0519/]).
- bpo-26282 [https://bugs.python.org/issue?@action=redirect&bpo=26282]: PyArg_ParseTupleAndKeywords() now supports positional-only parameters.

Tools/Demos

- bpo-26282 [https://bugs.python.org/issue?@action=redirect&bpo=26282]: Argument Clinic now supports positional-only and keyword parameters in the same function.

Python 3.6.0 alpha 1

Release date: 2016-05-16

Security

- bpo-26657 [https://bugs.python.org/issue?@action=redirect&bpo=26657]: Fix directory traversal vulnerability with http.server on Windows. This fixes a regression that was introduced in 3.3.4rc1 and 3.4.0rc1. Based on patch by Philipp Hagemeister.
- bpo-26313 [https://bugs.python.org/issue?@action=redirect&bpo=26313]: ssl.py _load_windows_store_certs fails if windows cert store is empty. Patch by Baji.
- bpo-25939 [https://bugs.python.org/issue?@action=redirect&bpo=25939]: On Windows open the cert store readonly in ssl.enum_certificates.

Core and Builtins

- bpo-20041 [https://bugs.python.org/issue?@action=redirect&bpo=20041]: Fixed TypeError when frame.f_trace is set to None. Patch by Xavier de Gaye.
- bpo-26168 [https://bugs.python.org/issue?@action=redirect&bpo=26168]: Fixed possible refleaks in failing Py_BuildValue() with the “N” format unit.
- bpo-26991 [https://bugs.python.org/issue?@action=redirect&bpo=26991]: Fix possible refleak when creating a function with annotations.
- bpo-27039 [https://bugs.python.org/issue?@action=redirect&bpo=27039]: Fixed bytearray.remove() for values greater than 127. Based on patch by Joe Jevnik.
- bpo-23640 [https://bugs.python.org/issue?@action=redirect&bpo=23640]: int.from_bytes() no longer bypasses constructors for subclasses.
- bpo-27005 [https://bugs.python.org/issue?@action=redirect&bpo=27005]: Optimized the float.fromhex() class method for exact float. It is now 2 times faster.

- bpo-18531 [https://bugs.python.org/issue?@action=redirect&bpo=18531]: Single var-keyword argument of dict subtype was passed unscathed to the C-defined function. Now it is converted to exact dict.
- bpo-26811 [https://bugs.python.org/issue?@action=redirect&bpo=26811]: gc.get_objects() no longer contains a broken tuple with NULL pointer.
- bpo-20120 [https://bugs.python.org/issue?@action=redirect&bpo=20120]: Use RawConfigParser for .pypirc parsing, removing support for interpolation unintentionally added with move to Python 3. Behavior no longer does any interpolation in .pypirc files, matching behavior in Python 2.7 and Setuptools 19.0.
- bpo-26249 [https://bugs.python.org/issue?@action=redirect&bpo=26249]: Memory functions of the **`PyMem_Malloc()`** domain (**`PYMEM_DOMAIN_MEM`**) now use the pymalloc allocator rather than system **`malloc()`**. Applications calling **`PyMem_Malloc()`** without holding the GIL can now crash: use `PYTHONMALLOC=debug` environment variable to validate the usage of memory allocators in your application.
- bpo-26802 [https://bugs.python.org/issue?@action=redirect&bpo=26802]: Optimize function calls only using unpacking like `func(*tuple)` (no other positional argument, no keyword): avoid copying the tuple. Patch written by Joe Jevnik.
- bpo-26659 [https://bugs.python.org/issue?@action=redirect&bpo=26659]: Make the builtin slice type support cycle collection.
- bpo-26718 [https://bugs.python.org/issue?@action=redirect&bpo=26718]: super.__init__ no longer leaks memory if called multiple times. NOTE: A direct call of super.__init__ is not endorsed!
- bpo-27138 [https://bugs.python.org/issue?@action=redirect&bpo=27138]: Fix the doc comment for FileFinder.find_spec().

- bpo-27147 [https://bugs.python.org/issue?@action=redirect&bpo=27147]: Mention **PEP 420** [https://peps.python.org/pep-0420/] in the importlib docs.
- bpo-25339 [https://bugs.python.org/issue?@action=redirect&bpo=25339]: PYTHONIOENCODING now has priority over locale in setting the error handler for stdin and stdout.
- bpo-26494 [https://bugs.python.org/issue?@action=redirect&bpo=26494]: Fixed crash on iterating exhausting iterators. Affected classes are generic sequence iterators, iterators of str, bytes, bytearray, list, tuple, set, frozenset, dict, OrderedDict, corresponding views and os.scandir() iterator.
- bpo-26574 [https://bugs.python.org/issue?@action=redirect&bpo=26574]: Optimize `bytes.replace(b'', b'.')` and `bytearray.replace(b'', b'.')`. Patch written by Josh Snider.
- bpo-26581 [https://bugs.python.org/issue?@action=redirect&bpo=26581]: If coding cookie is specified multiple times on a line in Python source code file, only the first one is taken to account.
- bpo-19711 [https://bugs.python.org/issue?@action=redirect&bpo=19711]: Add tests for reloading namespace packages.
- bpo-21099 [https://bugs.python.org/issue?@action=redirect&bpo=21099]: Switch applicable importlib tests to use **PEP 451** [https://peps.python.org/pep-0451/] API.
- bpo-26563 [https://bugs.python.org/issue?@action=redirect&bpo=26563]: Debug hooks on Python memory allocators now raise a fatal error if functions of the `PyMem_Malloc()` family are called without holding the GIL.
- bpo-26564 [https://bugs.python.org/issue?@action=redirect&bpo=26564]: On error, the debug hooks on Python memory allocators now use the `tracemalloc`

module to get the traceback where a memory block was allocated.

- bpo-26558 [https://bugs.python.org/issue?@action=redirect&bpo=26558]: The debug hooks on Python memory allocator `PyObject_Malloc()` now detect when functions are called without holding the GIL.
- bpo-26516 [https://bugs.python.org/issue?@action=redirect&bpo=26516]: Add `PYTHONMALLOC` environment variable to set the Python memory allocators and/or install debug hooks.
- bpo-26516 [https://bugs.python.org/issue?@action=redirect&bpo=26516]: The `PyMem_SetupDebugHooks()` function can now also be used on Python compiled in release mode.
- bpo-26516 [https://bugs.python.org/issue?@action=redirect&bpo=26516]: The `PYTHONMALLOCSTATS` environment variable can now also be used on Python compiled in release mode. It now has no effect if set to an empty string.
- bpo-26516 [https://bugs.python.org/issue?@action=redirect&bpo=26516]: In debug mode, debug hooks are now also installed on Python memory allocators when Python is configured without pymalloc.
- bpo-26464 [https://bugs.python.org/issue?@action=redirect&bpo=26464]: Fix str.translate() when string is ASCII and first replacements removes character, but next replacement uses a non-ASCII character or a string longer than 1 character. Regression introduced in Python 3.5.0.
- bpo-22836 [https://bugs.python.org/issue?@action=redirect&bpo=22836]: Ensure exception reports from PyErr_Display() and PyErr_WriteUnraisable() are sensible even when formatting them produces secondary errors. This affects the reports produced by sys.__excepthook__() and when __del__() raises an exception.

- bpo-26302 [https://bugs.python.org/issue?@action=redirect&bpo=26302]: Correct behavior to reject comma as a legal character for cookie names.
- bpo-26136 [https://bugs.python.org/issue?@action=redirect&bpo=26136]: Upgrade the warning when a generator raises StopIteration from PendingDeprecationWarning to DeprecationWarning. Patch by Anish Shah.
- bpo-26204 [https://bugs.python.org/issue?@action=redirect&bpo=26204]: The compiler now ignores all constant statements: bytes, str, int, float, complex, name constants (None, False, True), Ellipsis and ast.Constant; not only str and int. For example, `1.0` is now ignored in `def f(): 1.0`.
- bpo-4806 [https://bugs.python.org/issue?@action=redirect&bpo=4806]: Avoid masking the original TypeError exception when using star (`*`) unpacking in function calls. Based on patch by Hagen Fürstenau and Daniel Urban.
- bpo-26146 [https://bugs.python.org/issue?@action=redirect&bpo=26146]: Add a new kind of AST node: `ast.Constant`. It can be used by external AST optimizers, but the compiler does not emit directly such node.
- bpo-23601 [https://bugs.python.org/issue?@action=redirect&bpo=23601]: Sped-up allocation of dict key objects by using Python’s small object allocator. (Contributed by Julian Taylor.)
- bpo-18018 [https://bugs.python.org/issue?@action=redirect&bpo=18018]: Import raises ImportError instead of SystemError if a relative import is attempted without a known parent package.
- bpo-25843 [https://bugs.python.org/issue?@action=redirect&bpo=25843]: When compiling code, don’t merge constants if they are equal but have a different types. For example, `f1, f2 = lambda: 1, lambda: 1.0` is now correctly compiled to two different functions: `f1()`

returns `1` (`int`) and `f2()` returns `1.0` (`float`), even if `1` and `1.0` are equal.

- bpo-26107 [https://bugs.python.org/issue?@action=redirect&bpo=26107]: The format of the `co_lnotab` attribute of code objects changes to support negative line number delta.
- bpo-26154 [https://bugs.python.org/issue?@action=redirect&bpo=26154]: Add a new private _PyThreadState_UncheckedGet() function to get the current Python thread state, but don't issue a fatal error if it is NULL. This new function must be used instead of accessing directly the _PyThreadState_Current variable. The variable is no more exposed since Python 3.5.1 to hide the exact implementation of atomic C types, to avoid compiler issues.
- bpo-25791 [https://bugs.python.org/issue?@action=redirect&bpo=25791]: If __package__ != __spec__.parent or if neither __package__ or __spec__ are defined then ImportWarning is raised.
- bpo-22995 [https://bugs.python.org/issue?@action=redirect&bpo=22995]: [UPDATE] Comment out the one of the pickleability tests in _PyObject_GetState() due to regressions observed in Cython-based projects.
- bpo-25961 [https://bugs.python.org/issue?@action=redirect&bpo=25961]: Disallowed null characters in the type name.
- bpo-25973 [https://bugs.python.org/issue?@action=redirect&bpo=25973]: Fix segfault when an invalid nonlocal statement binds a name starting with two underscores.
- bpo-22995 [https://bugs.python.org/issue?@action=redirect&bpo=22995]: Instances of extension types with a state that aren't subclasses of list or dict and haven't implemented any pickle-related methods (__reduce__, __reduce_ex__, __getnewargs__,

__getnewargs_ex__, or __getstate__), can no longer be pickled. Including memoryview.

- bpo-20440 [https://bugs.python.org/issue?@action=redirect&bpo=20440]: Massive replacing unsafe attribute setting code with special macro Py_SETREF.
- bpo-25766 [https://bugs.python.org/issue?@action=redirect&bpo=25766]: Special method __bytes__() now works in str subclasses.
- bpo-25421 [https://bugs.python.org/issue?@action=redirect&bpo=25421]: __sizeof__ methods of builtin types now use dynamic basic size. This allows sys.getsize() to work correctly with their subclasses with __slots__ defined.
- bpo-25709 [https://bugs.python.org/issue?@action=redirect&bpo=25709]: Fixed problem with in-place string concatenation and utf-8 cache.
- bpo-5319 [https://bugs.python.org/issue?@action=redirect&bpo=5319]: New Py_FinalizeEx() API allowing Python to set an exit status of 120 on failure to flush buffered streams.
- bpo-25485 [https://bugs.python.org/issue?@action=redirect&bpo=25485]: telnetlib.Telnet is now a context manager.
- bpo-24097 [https://bugs.python.org/issue?@action=redirect&bpo=24097]: Fixed crash in object.__reduce__() if slot name is freed inside __getattr__.
- bpo-24731 [https://bugs.python.org/issue?@action=redirect&bpo=24731]: Fixed crash on converting objects with special methods __bytes__, __trunc__, and __float__ returning instances of subclasses of bytes, int, and float to subclasses of bytes, int, and float correspondingly.
- bpo-25630 [https://bugs.python.org/issue?@action=redirect&bpo=25630]: Fix a possible segfault during argument parsing in functions that accept filesystem paths.

- bpo-23564 [https://bugs.python.org/issue?@action=redirect&bpo=23564]: Fixed a partially broken sanity check in the _posixsubprocess internals regarding how fds_to_pass were passed to the child. The bug had no actual impact as subprocess.py already avoided it.
- bpo-25388 [https://bugs.python.org/issue?@action=redirect&bpo=25388]: Fixed tokenizer crash when processing undecodable source code with a null byte.
- bpo-25462 [https://bugs.python.org/issue?@action=redirect&bpo=25462]: The hash of the key now is calculated only once in most operations in C implementation of OrderedDict.
- bpo-22995 [https://bugs.python.org/issue?@action=redirect&bpo=22995]: Default implementation of __reduce__ and __reduce_ex__ now rejects builtin types with not defined __new__.
- bpo-24802 [https://bugs.python.org/issue?@action=redirect&bpo=24802]: Avoid buffer overreads when int(), float(), compile(), exec() and eval() are passed bytes-like objects. These objects are not necessarily terminated by a null byte, but the functions assumed they were.
- bpo-25555 [https://bugs.python.org/issue?@action=redirect&bpo=25555]: Fix parser and AST: fill lineno and col_offset of “arg” node when compiling AST from Python objects.
- bpo-24726 [https://bugs.python.org/issue?@action=redirect&bpo=24726]: Fixed a crash and leaking NULL in repr() of OrderedDict that was mutated by direct calls of dict methods.
- bpo-25449 [https://bugs.python.org/issue?@action=redirect&bpo=25449]: Iterating OrderedDict with keys with unstable hash now raises KeyError in C implementations as well as in Python implementation.
- bpo-25395 [https://bugs.python.org/issue?@action=redirect&bpo=25395]: Fixed crash when highly nested

OrderedDict structures were garbage collected.

- bpo-25401 [https://bugs.python.org/issue?@action=redirect&bpo=25401]: Optimize bytes.fromhex() and bytearray.fromhex(): they are now between 2x and 3.5x faster.
- bpo-25399 [https://bugs.python.org/issue?@action=redirect&bpo=25399]: Optimize bytearray % args using the new private _PyBytesWriter API. Formatting is now between 2.5 and 5 times faster.
- bpo-25274 [https://bugs.python.org/issue?@action=redirect&bpo=25274]: sys.setrecursionlimit() now raises a RecursionError if the new recursion limit is too low depending at the current recursion depth. Modify also the “lower-water mark” formula to make it monotonic. This mark is used to decide when the overflowed flag of the thread state is reset.
- bpo-24402 [https://bugs.python.org/issue?@action=redirect&bpo=24402]: Fix input() to prompt to the redirected stdout when sys.stdout.fileno() fails.
- bpo-25349 [https://bugs.python.org/issue?@action=redirect&bpo=25349]: Optimize bytes % args using the new private _PyBytesWriter API. Formatting is now up to 2 times faster.
- bpo-24806 [https://bugs.python.org/issue?@action=redirect&bpo=24806]: Prevent builtin types that are not allowed to be subclassed from being subclassed through multiple inheritance.
- bpo-25301 [https://bugs.python.org/issue?@action=redirect&bpo=25301]: The UTF-8 decoder is now up to 15 times as fast for error handlers: `ignore`, `replace` and `surrogateescape`.
- bpo-24848 [https://bugs.python.org/issue?@action=redirect&bpo=24848]: Fixed a number of bugs in UTF-7 decoding of misformed data.
- bpo-25267 [https://bugs.python.org/issue?@action=redirect&bpo=25267]: The UTF-8 encoder is now up to

75 times as fast for error handlers: `ignore`, `replace`, `surrogateescape`, `surrogatepass`. Patch co-written with Serhiy Storchaka.

- bpo-25280 [https://bugs.python.org/issue?@action=redirect&bpo=25280]: Import trace messages emitted in verbose (-v) mode are no longer formatted twice.
- bpo-25227 [https://bugs.python.org/issue?@action=redirect&bpo=25227]: Optimize ASCII and latin1 encoders with the `surrogateescape` error handler: the encoders are now up to 3 times as fast. Initial patch written by Serhiy Storchaka.
- bpo-25003 [https://bugs.python.org/issue?@action=redirect&bpo=25003]: On Solaris 11.3 or newer, os.urandom() now uses the getrandom() function instead of the getentropy() function. The getentropy() function is blocking to generate very good quality entropy, os.urandom() doesn't need such high-quality entropy.
- bpo-9232 [https://bugs.python.org/issue?@action=redirect&bpo=9232]: Modify Python's grammar to allow trailing commas in the argument list of a function declaration. For example, "def f(*, a = 3,): pass" is now legal. Patch from Mark Dickinson.
- bpo-24965 [https://bugs.python.org/issue?@action=redirect&bpo=24965]: Implement **PEP 498** [https://peps.python.org/pep-0498/] "Literal String Interpolation". This allows you to embed expressions inside f-strings, which are converted to normal strings at run time. Given x=3, then f'value={x}' == 'value=3'. Patch by Eric V. Smith.
- bpo-26478 [https://bugs.python.org/issue?@action=redirect&bpo=26478]: Fix semantic bugs when using binary operators with dictionary views and tuples.
- bpo-26171 [https://bugs.python.org/issue?@action=redirect&bpo=26171]: Fix possible integer overflow and heap corruption in zipimporter.get_data().

- bpo-25660 [https://bugs.python.org/issue?@action=redirect&bpo=25660]: Fix TAB key behaviour in REPL with readline.
- bpo-26288 [https://bugs.python.org/issue?@action=redirect&bpo=26288]: Optimize PyLong_AsDouble.
- bpo-26289 [https://bugs.python.org/issue?@action=redirect&bpo=26289]: Optimize floor and modulo division for single-digit longs. Microbenchmarks show 2-2.5x improvement. Built-in ‘divmod’ function is now also ~10% faster. (See also: bpo-26315 [https://bugs.python.org/issue?@action=redirect&bpo=26315])
- bpo-25887 [https://bugs.python.org/issue?@action=redirect&bpo=25887]: Raise a RuntimeError when a coroutine object is awaited more than once.

Library

- bpo-27057 [https://bugs.python.org/issue?@action=redirect&bpo=27057]: Fix os.set_inheritable() on Android, ioctl() is blocked by SELinux and fails with EACCESS. The function now falls back to fcntl(). Patch written by Michał Bednarski.
- bpo-27014 [https://bugs.python.org/issue?@action=redirect&bpo=27014]: Fix infinite recursion using typing.py. Thanks to Kalle Tuure!
- bpo-27031 [https://bugs.python.org/issue?@action=redirect&bpo=27031]: Removed dummy methods in Tkinter widget classes: tk_menuBar() and tk_bindForTraversal().
- bpo-14132 [https://bugs.python.org/issue?@action=redirect&bpo=14132]: Fix urllib.request redirect handling when the target only has a query string. Original fix by Ján Janech.
- bpo-17214 [https://bugs.python.org/issue?@action=redirect&bpo=17214]: The “urllib.request” module now percent-encodes non-ASCII bytes found in redirect

target URLs. Some servers send Location header fields with non-ASCII bytes, but “http.client” requires the request target to be ASCII-encodable, otherwise a UnicodeEncodeError is raised. Based on patch by Christian Heimes.

- bpo-27033 [https://bugs.python.org/issue?@action=redirect&bpo=27033]: The default value of the decode_data parameter for smtpd.SMTPChannel and smtpd.SMTPServer constructors is changed to False.
- bpo-27034 [https://bugs.python.org/issue?@action=redirect&bpo=27034]: Removed deprecated class asynchat.fifo.
- bpo-26870 [https://bugs.python.org/issue?@action=redirect&bpo=26870]: Added readline.set_auto_history(), which can stop entries being automatically added to the history list. Based on patch by Tyler Crompton.
- bpo-26039 [https://bugs.python.org/issue?@action=redirect&bpo=26039]: zipfile.ZipFile.open() can now be used to write data into a ZIP file, as well as for extracting data. Patch by Thomas Kluyver.
- bpo-26892 [https://bugs.python.org/issue?@action=redirect&bpo=26892]: Honor debuglevel flag in urllib.request.HTTPHandler. Patch contributed by Chi Hsuan Yen.
- bpo-22274 [https://bugs.python.org/issue?@action=redirect&bpo=22274]: In the subprocess module, allow stderr to be redirected to stdout even when stdout is not redirected. Patch by Akira Li.
- bpo-26807 [https://bugs.python.org/issue?@action=redirect&bpo=26807]: mock_open ‘files’ no longer error on readline at end of file. Patch from Yolanda Robla.
- bpo-25745 [https://bugs.python.org/issue?@action=redirect&bpo=25745]: Fixed leaking a userptr in curses panel destructor.

- bpo-26977 [https://bugs.python.org/issue?@action=redirect&bpo=26977]: Removed unnecessary, and ignored, call to sum of squares helper in statistics.pvariance.
- bpo-26002 [https://bugs.python.org/issue?@action=redirect&bpo=26002]: Use bisect in statistics.median instead of a linear search. Patch by Upendra Kuma.
- bpo-25974 [https://bugs.python.org/issue?@action=redirect&bpo=25974]: Make use of new Decimal.as_integer_ratio() method in statistics module. Patch by Stefan Krah.
- bpo-26996 [https://bugs.python.org/issue?@action=redirect&bpo=26996]: Add secrets module as described in **PEP 506** [https://peps.python.org/pep-0506/].
- bpo-26881 [https://bugs.python.org/issue?@action=redirect&bpo=26881]: The modulefinder module now supports extended opcode arguments.
- bpo-23815 [https://bugs.python.org/issue?@action=redirect&bpo=23815]: Fixed crashes related to directly created instances of types in _tkinter and curses.panel modules.
- bpo-17765 [https://bugs.python.org/issue?@action=redirect&bpo=17765]: weakref.ref() no longer silently ignores keyword arguments. Patch by Georg Brandl.
- bpo-26873 [https://bugs.python.org/issue?@action=redirect&bpo=26873]: xmlrpc now raises ResponseError on unsupported type tags instead of silently return incorrect result.
- bpo-26915 [https://bugs.python.org/issue?@action=redirect&bpo=26915]: The __contains__ methods in the collections ABCs now check for identity before checking equality. This better matches the behavior of the concrete classes, allows sensible handling of NaNs, and makes it easier to reason about container invariants.

- bpo-26711 [https://bugs.python.org/issue?@action=redirect&bpo=26711]: Fixed the comparison of plistlib.Data with other types.
- bpo-24114 [https://bugs.python.org/issue?@action=redirect&bpo=24114]: Fix an uninitialized variable in `ctypes.util`. The bug only occurs on SunOS when the ctypes implementation searches for the `crle` program. Patch by Xiang Zhang. Tested on SunOS by Kees Bos.
- bpo-26864 [https://bugs.python.org/issue?@action=redirect&bpo=26864]: In urllib.request, change the proxy bypass host checking against no_proxy to be case-insensitive, and to not match unrelated host names that happen to have a bypassed hostname as a suffix. Patch by Xiang Zhang.
- bpo-24902 [https://bugs.python.org/issue?@action=redirect&bpo=24902]: Print server URL on http.server startup. Initial patch by Felix Kaiser.
- bpo-25788 [https://bugs.python.org/issue?@action=redirect&bpo=25788]: fileinput.hook_encoded() now supports an “errors” argument for passing to open. Original patch by Joseph Hackman.
- bpo-26634 [https://bugs.python.org/issue?@action=redirect&bpo=26634]: recursive_repr() now sets __qualname__ of wrapper. Patch by Xiang Zhang.
- bpo-26804 [https://bugs.python.org/issue?@action=redirect&bpo=26804]: urllib.request will prefer lower_case proxy environment variables over UPPER_CASE or Mixed_Case ones. Patch contributed by Hans-Peter Jansen.
- bpo-26837 [https://bugs.python.org/issue?@action=redirect&bpo=26837]: assertSequenceEqual() now correctly outputs non-stringified differing items (like bytes in the -b mode). This affects assertListEqual() and assertTupleEqual().
- bpo-26041 [https://bugs.python.org/issue?@action=redirect&bpo=26041]: Remove “will be removed in

Python 3.7” from deprecation messages of platform.dist() and platform.linux_distribution(). Patch by Kumaripaba Miyurusara Athukorala.

- bpo-26822 [https://bugs.python.org/issue?@action=redirect&bpo=26822]: itemgetter, attrgetter and methodcaller objects no longer silently ignore keyword arguments.
- bpo-26733 [https://bugs.python.org/issue?@action=redirect&bpo=26733]: Disassembling a class now disassembles class and static methods. Patch by Xiang Zhang.
- bpo-26801 [https://bugs.python.org/issue?@action=redirect&bpo=26801]: Fix error handling in **`shutil.get_terminal_size()`**, catch **`AttributeError`** instead of **`NameError`**. Patch written by Emanuel Barry.
- bpo-24838 [https://bugs.python.org/issue?@action=redirect&bpo=24838]: tarfile’s ustar and gnu formats now correctly calculate name and link field limits for multibyte character encodings like utf-8.
- bpo-26717 [https://bugs.python.org/issue?@action=redirect&bpo=26717]: Stop encoding Latin-1-ized WSGI paths with UTF-8. Patch by Anthony Sottile.
- bpo-26782 [https://bugs.python.org/issue?@action=redirect&bpo=26782]: Add STARTUPINFO to subprocess.__all__ on Windows.
- bpo-26404 [https://bugs.python.org/issue?@action=redirect&bpo=26404]: Add context manager to socketserver. Patch by Aviv Palivoda.
- bpo-26735 [https://bugs.python.org/issue?@action=redirect&bpo=26735]: Fix **`os.urandom()`** on Solaris 11.3 and newer when reading more than 1,024 bytes: call `getrandom()` multiple times with a limit of 1024 bytes per call.
- bpo-26585 [https://bugs.python.org/issue?@action=redirect&bpo=26585]: Eliminate

http.server._quote_html() and use html.escape(quote=False). Patch by Xiang Zhang.

- bpo-26685 [https://bugs.python.org/issue?@action=redirect&bpo=26685]: Raise OSError if closing a socket fails.
- bpo-16329 [https://bugs.python.org/issue?@action=redirect&bpo=16329]: Add .webm to mimetypes.types_map. Patch by Giampaolo Rodola’.
- bpo-13952 [https://bugs.python.org/issue?@action=redirect&bpo=13952]: Add .csv to mimetypes.types_map. Patch by Geoff Wilson.
- bpo-26587 [https://bugs.python.org/issue?@action=redirect&bpo=26587]: the site module now allows .pth files to specify files to be added to sys.path (e.g. zip files).
- bpo-25609 [https://bugs.python.org/issue?@action=redirect&bpo=25609]: Introduce contextlib.AbstractContextManager and typing.ContextManager.
- bpo-26709 [https://bugs.python.org/issue?@action=redirect&bpo=26709]: Fixed Y2038 problem in loading binary PLists.
- bpo-23735 [https://bugs.python.org/issue?@action=redirect&bpo=23735]: Handle terminal resizing with Readline 6.3+ by installing our own SIGWINCH handler. Patch by Eric Price.
- bpo-25951 [https://bugs.python.org/issue?@action=redirect&bpo=25951]: Change SSLSocket.sendall() to return None, as explicitly documented for plain socket objects. Patch by Aviv Palivoda.
- bpo-26586 [https://bugs.python.org/issue?@action=redirect&bpo=26586]: In http.server, respond with “413 Request header fields too large” if there are too many header fields to parse, rather than killing the connection and raising an unhandled exception. Patch by Xiang Zhang.

- bpo-26676 [https://bugs.python.org/issue?@action=redirect&bpo=26676]: Added missing XMLPullParser to ElementTree.__all__.
- bpo-22854 [https://bugs.python.org/issue?@action=redirect&bpo=22854]: Change BufferedReader.writable() and BufferedWriter.readable() to always return False.
- bpo-26492 [https://bugs.python.org/issue?@action=redirect&bpo=26492]: Exhausted iterator of array.array now conforms with the behavior of iterators of other mutable sequences: it lefts exhausted even if iterated array is extended.
- bpo-26641 [https://bugs.python.org/issue?@action=redirect&bpo=26641]: doctest.DocFileTest and doctest.testfile() now support packages (module split into multiple directories) for the package parameter.
- bpo-25195 [https://bugs.python.org/issue?@action=redirect&bpo=25195]: Fix a regression in mock.MagicMock. _Call is a subclass of tuple (changeset 3603bae63c13 only works for classes) so we need to implement __ne__ ourselves. Patch by Andrew Plummer.
- bpo-26644 [https://bugs.python.org/issue?@action=redirect&bpo=26644]: Raise ValueError rather than SystemError when a negative length is passed to SSLSocket.recv() or read().
- bpo-23804 [https://bugs.python.org/issue?@action=redirect&bpo=23804]: Fix SSL recv(0) and read(0) methods to return zero bytes instead of up to 1024.
- bpo-26616 [https://bugs.python.org/issue?@action=redirect&bpo=26616]: Fixed a bug in datetime.astimezone() method.
- bpo-26637 [https://bugs.python.org/issue?@action=redirect&bpo=26637]: The `importlib` module now emits an `ImportError` rather than a `TypeError` if `__import__()` is tried during the Python shutdown process but `sys.path` is already cleared (set to `None`).

- bpo-21925 [https://bugs.python.org/issue?@action=redirect&bpo=21925]: `warnings.formatwarning()` now catches exceptions when calling `linecache.getline()` and `tracemalloc.get_object_traceback()` to be able to log `ResourceWarning` emitted late during the Python shutdown process.
- bpo-23848 [https://bugs.python.org/issue?@action=redirect&bpo=23848]: On Windows, faulthandler.enable() now also installs an exception handler to dump the traceback of all Python threads on any Windows exception, not only on UNIX signals (SIGSEGV, SIGFPE, SIGABRT).
- bpo-26530 [https://bugs.python.org/issue?@action=redirect&bpo=26530]: Add C functions `_PyTraceMalloc_Track()` and `_PyTraceMalloc_Untrack()` to track memory blocks using the `tracemalloc` module. Add `_PyTraceMalloc_GetTraceback()` to get the traceback of an object.
- bpo-26588 [https://bugs.python.org/issue?@action=redirect&bpo=26588]: The _tracemalloc now supports tracing memory allocations of multiple address spaces (domains).
- bpo-24266 [https://bugs.python.org/issue?@action=redirect&bpo=24266]: Ctrl+C during Readline history search now cancels the search mode when compiled with Readline 7.
- bpo-26590 [https://bugs.python.org/issue?@action=redirect&bpo=26590]: Implement a safe finalizer for the _socket.socket type. It now releases the GIL to close the socket.
- bpo-18787 [https://bugs.python.org/issue?@action=redirect&bpo=18787]: spwd.getspnam() now raises a PermissionError if the user doesn’t have privileges.
- bpo-26560 [https://bugs.python.org/issue?@action=redirect&bpo=26560]: Avoid potential ValueError in

BaseHandler.start_response. Initial patch by Peter Inglesby.

- bpo-26567 [https://bugs.python.org/issue?@action=redirect&bpo=26567]: Add a new function `PyErr_ResourceWarning()` function to pass the destroyed object. Add a *source* attribute to **`warnings.WarningMessage`**. Add warnings._showwarnmsg() which uses tracemalloc to get the traceback where source object was allocated.
- bpo-26569 [https://bugs.python.org/issue?@action=redirect&bpo=26569]: Fix `pyclbr.readmodule()` and `pyclbr.readmodule_ex()` to support importing packages.
- bpo-26499 [https://bugs.python.org/issue?@action=redirect&bpo=26499]: Account for remaining Content-Length in HTTPResponse.readline() and read1(). Based on patch by Silent Ghost. Also document that HTTPResponse now supports these methods.
- bpo-25320 [https://bugs.python.org/issue?@action=redirect&bpo=25320]: Handle sockets in directories unittest discovery is scanning. Patch from Victor van den Elzen.
- bpo-16181 [https://bugs.python.org/issue?@action=redirect&bpo=16181]: cookiejar.http2time() now returns None if year is higher than datetime.MAXYEAR.
- bpo-26513 [https://bugs.python.org/issue?@action=redirect&bpo=26513]: Fixes platform module detection of Windows Server
- bpo-23718 [https://bugs.python.org/issue?@action=redirect&bpo=23718]: Fixed parsing time in week 0 before Jan 1. Original patch by Tamás Bence Gedai.
- bpo-26323 [https://bugs.python.org/issue?@action=redirect&bpo=26323]: Add Mock.assert_called() and Mock.assert_called_once() methods to unittest.mock. Patch written by Amit Saha.
- bpo-20589 [https://bugs.python.org/issue?@action=redirect&bpo=20589]: Invoking Path.owner() and

Path.group() on Windows now raise NotImplementedError instead of ImportError.

- bpo-26177 [https://bugs.python.org/issue?@action=redirect&bpo=26177]: Fixed the keys() method for Canvas and Scrollbar widgets.
- bpo-15068 [https://bugs.python.org/issue?@action=redirect&bpo=15068]: Got rid of excessive buffering in fileinput. The bufsize parameter is now deprecated and ignored.
- bpo-19475 [https://bugs.python.org/issue?@action=redirect&bpo=19475]: Added an optional argument timespec to the datetime isoformat() method to choose the precision of the time component.
- bpo-2202 [https://bugs.python.org/issue?@action=redirect&bpo=2202]: Fix UnboundLocalError in AbstractDigestAuthHandler.get_algorithm_impls. Initial patch by Mathieu Dupuy.
- bpo-26167 [https://bugs.python.org/issue?@action=redirect&bpo=26167]: Minimized overhead in copy.copy() and copy.deepcopy(). Optimized copying and deepcopying bytearrays, NotImplemented, slices, short lists, tuples, dicts, sets.
- bpo-25718 [https://bugs.python.org/issue?@action=redirect&bpo=25718]: Fixed pickling and copying the accumulate() iterator with total is None.
- bpo-26475 [https://bugs.python.org/issue?@action=redirect&bpo=26475]: Fixed debugging output for regular expressions with the (?x) flag.
- bpo-26482 [https://bugs.python.org/issue?@action=redirect&bpo=26482]: Allowed pickling recursive dequeues.
- bpo-26335 [https://bugs.python.org/issue?@action=redirect&bpo=26335]: Make mmap.write() return the number of bytes written like other write methods. Patch by Jakub Stasiak.

- bpo-26457 [https://bugs.python.org/issue?@action=redirect&bpo=26457]: Fixed the subnets() methods in IP network classes for the case when resulting prefix length is equal to maximal prefix length. Based on patch by Xiang Zhang.
- bpo-26385 [https://bugs.python.org/issue?@action=redirect&bpo=26385]: Remove the file if the internal open() call in NamedTemporaryFile() fails. Patch by Silent Ghost.
- bpo-26402 [https://bugs.python.org/issue?@action=redirect&bpo=26402]: Fix XML-RPC client to retry when the server shuts down a persistent connection. This was a regression related to the new http.client.RemoteDisconnected exception in 3.5.0a4.
- bpo-25913 [https://bugs.python.org/issue?@action=redirect&bpo=25913]: Leading `<~` is optional now in base64.a85decode() with adobe=True. Patch by Swati Jaiswal.
- bpo-26186 [https://bugs.python.org/issue?@action=redirect&bpo=26186]: Remove an invalid type check in importlib.util.LazyLoader.
- bpo-26367 [https://bugs.python.org/issue?@action=redirect&bpo=26367]: importlib.__import__() raises ImportError like builtins.__import__() when `level` is specified but without an accompanying package specified.
- bpo-26309 [https://bugs.python.org/issue?@action=redirect&bpo=26309]: In the “socketserver” module, shut down the request (closing the connected socket) when verify_request() returns false. Patch by Aviv Palivoda.
- bpo-23430 [https://bugs.python.org/issue?@action=redirect&bpo=23430]: Change the socketserver module to only catch exceptions raised from a request handler that are derived from Exception (instead of BaseException). Therefore SystemExit and

KeyboardInterrupt no longer trigger the handle_error() method, and will now to stop a single-threaded server.

- bpo-25995 [https://bugs.python.org/issue?@action=redirect&bpo=25995]: os.walk() no longer uses FDs proportional to the tree depth.
- bpo-25994 [https://bugs.python.org/issue?@action=redirect&bpo=25994]: Added the close() method and the support of the context manager protocol for the os.scandir() iterator.
- bpo-23992 [https://bugs.python.org/issue?@action=redirect&bpo=23992]: multiprocessing: make MapResult not fail-fast upon exception.
- bpo-26243 [https://bugs.python.org/issue?@action=redirect&bpo=26243]: Support keyword arguments to zlib.compress(). Patch by Aviv Palivoda.
- bpo-26117 [https://bugs.python.org/issue?@action=redirect&bpo=26117]: The os.scandir() iterator now closes file descriptor not only when the iteration is finished, but when it was failed with error.
- bpo-25949 [https://bugs.python.org/issue?@action=redirect&bpo=25949]: __dict__ for an OrderedDict instance is now created only when needed.
- bpo-25911 [https://bugs.python.org/issue?@action=redirect&bpo=25911]: Restored support of bytes paths in os.walk() on Windows.
- bpo-26045 [https://bugs.python.org/issue?@action=redirect&bpo=26045]: Add UTF-8 suggestion to error message when posting a non-Latin-1 string with http.client.
- bpo-26039 [https://bugs.python.org/issue?@action=redirect&bpo=26039]: Added zipfile.ZipInfo.from_file() and zipinfo.ZipInfo.is_dir(). Patch by Thomas Kluyver.
- bpo-12923 [https://bugs.python.org/issue?@action=redirect&bpo=12923]: Reset FancyURLopener’s redirect counter even if there is an exception. Based on patches by Brian Brazil and Daniel Rocco.

- bpo-25945 [https://bugs.python.org/issue?@action=redirect&bpo=25945]: Fixed a crash when unpickle the functools.partial object with wrong state. Fixed a leak in failed functools.partial constructor. “args” and “keywords” attributes of functools.partial have now always types tuple and dict correspondingly.
- bpo-26202 [https://bugs.python.org/issue?@action=redirect&bpo=26202]: copy.deepcopy() now correctly copies range() objects with non-atomic attributes.
- bpo-23076 [https://bugs.python.org/issue?@action=redirect&bpo=23076]: Path.glob() now raises a ValueError if it’s called with an invalid pattern. Patch by Thomas Nyberg.
- bpo-19883 [https://bugs.python.org/issue?@action=redirect&bpo=19883]: Fixed possible integer overflows in zipimport.
- bpo-26227 [https://bugs.python.org/issue?@action=redirect&bpo=26227]: On Windows, getnameinfo(), gethostbyaddr() and gethostbyname_ex() functions of the socket module now decode the hostname from the ANSI code page rather than UTF-8.
- bpo-26099 [https://bugs.python.org/issue?@action=redirect&bpo=26099]: The site module now writes an error into stderr if sitecustomize module can be imported but executing the module raise an ImportError. Same change for usercustomize.
- bpo-26147 [https://bugs.python.org/issue?@action=redirect&bpo=26147]: xmlrpc now works with strings not encodable with used non-UTF-8 encoding.
- bpo-25935 [https://bugs.python.org/issue?@action=redirect&bpo=25935]: Garbage collector now breaks reference loops with OrderedDict.
- bpo-16620 [https://bugs.python.org/issue?@action=redirect&bpo=16620]: Fixed AttributeError in msilib.Directory.glob().

- bpo-26013 [https://bugs.python.org/issue?@action=redirect&bpo=26013]: Added compatibility with broken protocol 2 pickles created in old Python 3 versions (3.4.3 and lower).
- bpo-26129 [https://bugs.python.org/issue?@action=redirect&bpo=26129]: Deprecated accepting non-integers in grp.getgrgid().
- bpo-25850 [https://bugs.python.org/issue?@action=redirect&bpo=25850]: Use cross-compilation by default for 64-bit Windows.
- bpo-25822 [https://bugs.python.org/issue?@action=redirect&bpo=25822]: Add docstrings to the fields of urllib.parse results. Patch contributed by Swati Jaiswal.
- bpo-22642 [https://bugs.python.org/issue?@action=redirect&bpo=22642]: Convert trace module option parsing mechanism to argparse. Patch contributed by SilentGhost.
- bpo-24705 [https://bugs.python.org/issue?@action=redirect&bpo=24705]: Fix sysconfig._parse_makefile not expanding ${} vars appearing before $() vars.
- bpo-26069 [https://bugs.python.org/issue?@action=redirect&bpo=26069]: Remove the deprecated apis in the trace module.
- bpo-22138 [https://bugs.python.org/issue?@action=redirect&bpo=22138]: Fix mock.patch behavior when patching descriptors. Restore original values after patching. Patch contributed by Sean McCully.
- bpo-25672 [https://bugs.python.org/issue?@action=redirect&bpo=25672]: In the ssl module, enable the SSL_MODE_RELEASE_BUFFERS mode option if it is safe to do so.
- bpo-26012 [https://bugs.python.org/issue?@action=redirect&bpo=26012]: Don’t traverse into symlinks for ** pattern in pathlib.Path.[r]glob().
- bpo-24120 [https://bugs.python.org/issue?@action=redirect&bpo=24120]: Ignore PermissionError when

traversing a tree with pathlib.Path.[r]glob(). Patch by Ulrich Petri.
- bpo-21815 [https://bugs.python.org/issue?@action=redirect&bpo=21815]: Accept] characters in the data portion of imap responses, in order to handle the flags with square brackets accepted and produced by servers such as gmail.
- bpo-25447 [https://bugs.python.org/issue?@action=redirect&bpo=25447]: fileinput now uses sys.stdin as-is if it does not have a buffer attribute (restores backward compatibility).
- bpo-25971 [https://bugs.python.org/issue?@action=redirect&bpo=25971]: Optimized creating Fractions from floats by 2 times and from Decimals by 3 times.
- bpo-25802 [https://bugs.python.org/issue?@action=redirect&bpo=25802]: Document as deprecated the remaining implementations of importlib.abc.Loader.load_module().
- bpo-25928 [https://bugs.python.org/issue?@action=redirect&bpo=25928]: Add Decimal.as_integer_ratio().
- bpo-25447 [https://bugs.python.org/issue?@action=redirect&bpo=25447]: Copying the lru_cache() wrapper object now always works, independently from the type of the wrapped object (by returning the original object unchanged).
- bpo-25768 [https://bugs.python.org/issue?@action=redirect&bpo=25768]: Have the functions in compileall return booleans instead of ints and add proper documentation and tests for the return values.
- bpo-24103 [https://bugs.python.org/issue?@action=redirect&bpo=24103]: Fixed possible use after free in ElementTree.XMLPullParser.
- bpo-25860 [https://bugs.python.org/issue?@action=redirect&bpo=25860]: os.fwalk() no longer skips remaining directories when error occurs. Original patch by Samson Lee.

- bpo-25914 [https://bugs.python.org/issue?@action=redirect&bpo=25914]: Fixed and simplified OrderedDict.__sizeof__.
- bpo-25869 [https://bugs.python.org/issue?@action=redirect&bpo=25869]: Optimized deepcopying ElementTree; it is now 20 times faster.
- bpo-25873 [https://bugs.python.org/issue?@action=redirect&bpo=25873]: Optimized iterating ElementTree. Iterating elements Element.iter() is now 40% faster, iterating text Element.itertext() is now up to 2.5 times faster.
- bpo-25902 [https://bugs.python.org/issue?@action=redirect&bpo=25902]: Fixed various refcount issues in ElementTree iteration.
- bpo-22227 [https://bugs.python.org/issue?@action=redirect&bpo=22227]: The TarFile iterator is reimplemented using generator. This implementation is simpler that using class.
- bpo-25638 [https://bugs.python.org/issue?@action=redirect&bpo=25638]: Optimized ElementTree.iterparse(); it is now 2x faster. Optimized ElementTree parsing; it is now 10% faster.
- bpo-25761 [https://bugs.python.org/issue?@action=redirect&bpo=25761]: Improved detecting errors in broken pickle data.
- bpo-25717 [https://bugs.python.org/issue?@action=redirect&bpo=25717]: Restore the previous behaviour of tolerating most fstat() errors when opening files. This was a regression in 3.5a1, and stopped anonymous temporary files from working in special cases.
- bpo-24903 [https://bugs.python.org/issue?@action=redirect&bpo=24903]: Fix regression in number of arguments compileall accepts when ‘-d’ is specified. The check on the number of arguments has been dropped completely as it never worked correctly anyway.

- bpo-25764 [https://bugs.python.org/issue?@action=redirect&bpo=25764]: In the subprocess module, preserve any exception caused by fork() failure when preexec_fn is used.
- bpo-25771 [https://bugs.python.org/issue?@action=redirect&bpo=25771]: Tweak the exception message for importlib.util.resolve_name() when ‘package’ isn’t specified but necessary.
- bpo-6478 [https://bugs.python.org/issue?@action=redirect&bpo=6478]: _strptime’s regexp cache now is reset after changing timezone with time.tzset().
- bpo-14285 [https://bugs.python.org/issue?@action=redirect&bpo=14285]: When executing a package with the “python -m package” option, and package initialization fails, a proper traceback is now reported. The “runpy” module now lets exceptions from package initialization pass back to the caller, rather than raising ImportError.
- bpo-19771 [https://bugs.python.org/issue?@action=redirect&bpo=19771]: Also in runpy and the “-m” option, omit the irrelevant message “… is a package and cannot be directly executed” if the package could not even be initialized (e.g. due to a bad `*.pyc` file).
- bpo-25177 [https://bugs.python.org/issue?@action=redirect&bpo=25177]: Fixed problem with the mean of very small and very large numbers. As a side effect, statistics.mean and statistics.variance should be significantly faster.
- bpo-25718 [https://bugs.python.org/issue?@action=redirect&bpo=25718]: Fixed copying object with state with boolean value is false.
- bpo-10131 [https://bugs.python.org/issue?@action=redirect&bpo=10131]: Fixed deep copying of minidom documents. Based on patch by Marian Ganisin.
- bpo-7990 [https://bugs.python.org/issue?@action=redirect&bpo=7990]: dir() on ElementTree.Element now lists properties: “tag”,

“text”, “tail” and “attrib”. Original patch by Santoso Wijaya.

- bpo-25725 [https://bugs.python.org/issue?@action=redirect&bpo=25725]: Fixed a reference leak in pickle.loads() when unpickling invalid data including tuple instructions.
- bpo-25663 [https://bugs.python.org/issue?@action=redirect&bpo=25663]: In the Readline completer, avoid listing duplicate global names, and search the global namespace before searching builtins.
- bpo-25688 [https://bugs.python.org/issue?@action=redirect&bpo=25688]: Fixed file leak in ElementTree.iterparse() raising an error.
- bpo-23914 [https://bugs.python.org/issue?@action=redirect&bpo=23914]: Fixed SystemError raised by unpickler on broken pickle data.
- bpo-25691 [https://bugs.python.org/issue?@action=redirect&bpo=25691]: Fixed crash on deleting ElementTree.Element attributes.
- bpo-25624 [https://bugs.python.org/issue?@action=redirect&bpo=25624]: ZipFile now always writes a ZIP_STORED header for directory entries. Patch by Dingyuan Wang.
- bpo-25626 [https://bugs.python.org/issue?@action=redirect&bpo=25626]: Change three zlib functions to accept sizes that fit in Py_ssize_t, but internally cap those sizes to UINT_MAX. This resolves a regression in 3.5 where GzipFile.read() failed to read chunks larger than 2 or 4 GiB. The change affects the zlib.Decompress.decompress() max_length parameter, the zlib.decompress() bufsize parameter, and the zlib.Decompress.flush() length parameter.
- bpo-25583 [https://bugs.python.org/issue?@action=redirect&bpo=25583]: Avoid incorrect errors raised by os.makedirs(exist_ok=True) when the OS gives priority to errors such as EACCES over EEXIST.

- bpo-25593 [https://bugs.python.org/issue?@action=redirect&bpo=25593]: Change semantics of EventLoop.stop() in asyncio.
- bpo-6973 [https://bugs.python.org/issue?@action=redirect&bpo=6973]: When we know a subprocess.Popen process has died, do not allow the send_signal(), terminate(), or kill() methods to do anything as they could potentially signal a different process.
- bpo-23883 [https://bugs.python.org/issue?@action=redirect&bpo=23883]: Added missing APIs to __all__ to match the documented APIs for the following modules: calendar, csv, enum, fileinput, ftplib, logging, optparse, tarfile, threading and wave. Also added a test.support.check__all__() helper. Patches by Jacek Kołodziej, Mauro S. M. Rodrigues and Joel Taddei.
- bpo-25590 [https://bugs.python.org/issue?@action=redirect&bpo=25590]: In the Readline completer, only call getattr() once per attribute. Also complete names of attributes such as properties and slots which are listed by dir() but not yet created on an instance.
- bpo-25498 [https://bugs.python.org/issue?@action=redirect&bpo=25498]: Fix a crash when garbage-collecting ctypes objects created by wrapping a memoryview. This was a regression made in 3.5a1. Based on patch by Eryksun.
- bpo-25584 [https://bugs.python.org/issue?@action=redirect&bpo=25584]: Added “escape” to the __all__ list in the glob module.
- bpo-25584 [https://bugs.python.org/issue?@action=redirect&bpo=25584]: Fixed recursive glob() with patterns starting with `**`.
- bpo-25446 [https://bugs.python.org/issue?@action=redirect&bpo=25446]: Fix regression in smtplib’s AUTH LOGIN support.
- bpo-18010 [https://bugs.python.org/issue?@action=redirect&bpo=18010]: Fix the pydoc web server’s

module search function to handle exceptions from importing packages.

- bpo-25554 [https://bugs.python.org/issue?@action=redirect&bpo=25554]: Got rid of circular references in regular expression parsing.
- bpo-18973 [https://bugs.python.org/issue?@action=redirect&bpo=18973]: Command-line interface of the calendar module now uses argparse instead of optparse.
- bpo-25510 [https://bugs.python.org/issue?@action=redirect&bpo=25510]: fileinput.FileInput.readline() now returns b'' instead of '' at the end if the FileInput was opened with binary mode. Patch by Ryosuke Ito.
- bpo-25503 [https://bugs.python.org/issue?@action=redirect&bpo=25503]: Fixed inspect.getdoc() for inherited docstrings of properties. Original patch by John Mark Vandenberg.
- bpo-25515 [https://bugs.python.org/issue?@action=redirect&bpo=25515]: Always use os.urandom as a source of randomness in uuid.uuid4.
- bpo-21827 [https://bugs.python.org/issue?@action=redirect&bpo=21827]: Fixed textwrap.dedent() for the case when largest common whitespace is a substring of smallest leading whitespace. Based on patch by Robert Li.
- bpo-25447 [https://bugs.python.org/issue?@action=redirect&bpo=25447]: The lru_cache() wrapper objects now can be copied and pickled (by returning the original object unchanged).
- bpo-25390 [https://bugs.python.org/issue?@action=redirect&bpo=25390]: typing: Don't crash on Union[str, Pattern].
- bpo-25441 [https://bugs.python.org/issue?@action=redirect&bpo=25441]: asyncio: Raise error from drain() when socket is closed.
- bpo-25410 [https://bugs.python.org/issue?@action=redirect&bpo=25410]: Cleaned up and fixed minor

bugs in C implementation of OrderedDict.

- bpo-25411 [https://bugs.python.org/issue?@action=redirect&bpo=25411]: Improved Unicode support in SMTPHandler through better use of the email package. Thanks to user simon04 for the patch.
- Move the imp module from a PendingDeprecationWarning to DeprecationWarning.
- bpo-25407 [https://bugs.python.org/issue?@action=redirect&bpo=25407]: Remove mentions of the formatter module being removed in Python 3.6.
- bpo-25406 [https://bugs.python.org/issue?@action=redirect&bpo=25406]: Fixed a bug in C implementation of OrderedDict.move_to_end() that caused segmentation fault or hang in iterating after moving several items to the start of ordered dict.
- bpo-25382 [https://bugs.python.org/issue?@action=redirect&bpo=25382]: pickletools.dis() now outputs implicit memo index for the MEMOIZE opcode.
- bpo-25357 [https://bugs.python.org/issue?@action=redirect&bpo=25357]: Add an optional newline parameter to binascii.b2a_base64(). base64.b64encode() uses it to avoid a memory copy.
- bpo-24164 [https://bugs.python.org/issue?@action=redirect&bpo=24164]: Objects that need calling `__new__` with keyword arguments, can now be pickled using pickle protocols older than protocol version 4.
- bpo-25364 [https://bugs.python.org/issue?@action=redirect&bpo=25364]: zipfile now works in threads disabled builds.
- bpo-25328 [https://bugs.python.org/issue?@action=redirect&bpo=25328]: smtpd’s SMTPChannel now correctly raises a ValueError if both decode_data and enable_SMTPUTF8 are set to true.
- bpo-16099 [https://bugs.python.org/issue?@action=redirect&bpo=16099]: RobotFileParser now supports

Crawl-delay and Request-rate extensions. Patch by Nikolay Bogoychev.

- bpo-25316 [https://bugs.python.org/issue?@action=redirect&bpo=25316]: distutils raises OSError instead of DistutilsPlatformError when MSVC is not installed.
- bpo-25380 [https://bugs.python.org/issue?@action=redirect&bpo=25380]: Fixed protocol for the STACK_GLOBAL opcode in pickletools.opcodes.
- bpo-23972 [https://bugs.python.org/issue?@action=redirect&bpo=23972]: Updates asyncio datagram create method allowing reuseport and reuseaddr socket options to be set prior to binding the socket. Mirroring the existing asyncio create_server method the reuseaddr option for datagram sockets defaults to True if the O/S is ‘posix’ (except if the platform is Cygwin). Patch by Chris Laws.
- bpo-25304 [https://bugs.python.org/issue?@action=redirect&bpo=25304]: Add asyncio.run_coroutine_threadsafe(). This lets you submit a coroutine to a loop from another thread, returning a concurrent.futures.Future. By Vincent Michel.
- bpo-25232 [https://bugs.python.org/issue?@action=redirect&bpo=25232]: Fix CGIRequestHandler to split the query from the URL at the first question mark (?) rather than the last. Patch from Xiang Zhang.
- bpo-24657 [https://bugs.python.org/issue?@action=redirect&bpo=24657]: Prevent CGIRequestHandler from collapsing slashes in the query part of the URL as if it were a path. Patch from Xiang Zhang.
- bpo-25287 [https://bugs.python.org/issue?@action=redirect&bpo=25287]: Don’t add crypt.METHOD_CRYPT to crypt.methods if it’s not supported. Check if it is supported, it may not be supported on OpenBSD for example.
- bpo-23600 [https://bugs.python.org/issue?@action=redirect&bpo=23600]: Default implementation of

tzinfo.fromutc() was returning wrong results in some cases.

- bpo-25203 [https://bugs.python.org/issue?@action=redirect&bpo=25203]: Failed readline.set_completer_delims() no longer left the module in inconsistent state.
- bpo-25011 [https://bugs.python.org/issue?@action=redirect&bpo=25011]: rlcompleter now omits private and special attribute names unless the prefix starts with underscores.
- bpo-25209 [https://bugs.python.org/issue?@action=redirect&bpo=25209]: rlcompleter now can add a space or a colon after completed keyword.
- bpo-22241 [https://bugs.python.org/issue?@action=redirect&bpo=22241]: timezone.utc name is now plain ‘UTC’, not ‘UTC-00:00’.
- bpo-23517 [https://bugs.python.org/issue?@action=redirect&bpo=23517]: fromtimestamp() and utcfromtimestamp() methods of datetime.datetime now round microseconds to nearest with ties going to nearest even integer (ROUND_HALF_EVEN), as round(float), instead of rounding towards -Infinity (ROUND_FLOOR).
- bpo-23552 [https://bugs.python.org/issue?@action=redirect&bpo=23552]: Timeit now warns when there is substantial (4x) variance between best and worst times. Patch from Serhiy Storchaka.
- bpo-24633 [https://bugs.python.org/issue?@action=redirect&bpo=24633]: site-packages/README -> README.txt.
- bpo-24879 [https://bugs.python.org/issue?@action=redirect&bpo=24879]: help() and pydoc can now list named tuple fields in the order they were defined rather than alphabetically. The ordering is determined by the _fields attribute if present.

- bpo-24874 [https://bugs.python.org/issue?@action=redirect&bpo=24874]: Improve speed of itertools.cycle() and make its pickle more compact.
- Fix crash in itertools.cycle.__setstate__() when the first argument wasn't a list.
- bpo-20059 [https://bugs.python.org/issue?@action=redirect&bpo=20059]: urllib.parse raises ValueError on all invalid ports. Patch by Martin Panter.
- bpo-24360 [https://bugs.python.org/issue?@action=redirect&bpo=24360]: Improve __repr__ of argparse.Namespace() for invalid identifiers. Patch by Matthias Bussonnier.
- bpo-23426 [https://bugs.python.org/issue?@action=redirect&bpo=23426]: run_setup was broken in distutils. Patch from Alexander Belopolsky.
- bpo-13938 [https://bugs.python.org/issue?@action=redirect&bpo=13938]: 2to3 converts StringTypes to a tuple. Patch from Mark Hammond.
- bpo-2091 [https://bugs.python.org/issue?@action=redirect&bpo=2091]: open() accepted a 'U' mode string containing '+', but 'U' can only be used with 'r'. Patch from Jeff Balogh and John O'Connor.
- bpo-8585 [https://bugs.python.org/issue?@action=redirect&bpo=8585]: improved tests for zipimporter2. Patch from Mark Lawrence.
- bpo-18622 [https://bugs.python.org/issue?@action=redirect&bpo=18622]: unittest.mock.mock_open().reset_mock would recurse infinitely. Patch from Nicola Palumbo and Laurent De Buyst.
- bpo-24426 [https://bugs.python.org/issue?@action=redirect&bpo=24426]: Fast searching optimization in regular expressions now works for patterns that starts with capturing groups. Fast searching optimization now can't be disabled at compile time.

- bpo-23661 [https://bugs.python.org/issue?@action=redirect&bpo=23661]: unittest.mock side_effects can now be exceptions again. This was a regression vs Python 3.4. Patch from Ignacio Rossi
- bpo-13248 [https://bugs.python.org/issue?@action=redirect&bpo=13248]: Remove deprecated inspect.getmoduleinfo function.
- bpo-25578 [https://bugs.python.org/issue?@action=redirect&bpo=25578]: Fix (another) memory leak in SSLSocket.getpeercer().
- bpo-25530 [https://bugs.python.org/issue?@action=redirect&bpo=25530]: Disable the vulnerable SSLv3 protocol by default when creating ssl.SSLContext.
- bpo-25569 [https://bugs.python.org/issue?@action=redirect&bpo=25569]: Fix memory leak in SSLSocket.getpeercert().
- bpo-25471 [https://bugs.python.org/issue?@action=redirect&bpo=25471]: Sockets returned from accept() shouldn’t appear to be nonblocking.
- bpo-25319 [https://bugs.python.org/issue?@action=redirect&bpo=25319]: When threading.Event is reinitialized, the underlying condition should use a regular lock rather than a recursive lock.
- Skip getaddrinfo if host is already resolved. Patch by A. Jesse Jiryu Davis.
- bpo-26050 [https://bugs.python.org/issue?@action=redirect&bpo=26050]: Add asyncio.StreamReader.readuntil() method. Patch by Марк Коренберг.
- bpo-25924 [https://bugs.python.org/issue?@action=redirect&bpo=25924]: Avoid unnecessary serialization of getaddrinfo(3) calls on OS X versions 10.5 or higher. Original patch by A. Jesse Jiryu Davis.
- bpo-26406 [https://bugs.python.org/issue?@action=redirect&bpo=26406]: Avoid unnecessary serialization

of getaddrinfo(3) calls on current versions of OpenBSD and NetBSD. Patch by A. Jesse Jiryu Davis.

- bpo-26848 [https://bugs.python.org/issue?@action=redirect&bpo=26848]: Fix asyncio/subprocess.communicate() to handle empty input. Patch by Jack O’Connor.
- bpo-27040 [https://bugs.python.org/issue?@action=redirect&bpo=27040]: Add loop.get_exception_handler method
- bpo-27041 [https://bugs.python.org/issue?@action=redirect&bpo=27041]: asyncio: Add loop.create_future method

IDLE

- bpo-20640 [https://bugs.python.org/issue?@action=redirect&bpo=20640]: Add tests for idlelib.configHelpSourceEdit. Patch by Saimadhav Heblikar.
- In the ‘IDLE-console differences’ section of the IDLE doc, clarify how running with IDLE affects sys.modules and the standard streams.
- bpo-25507 [https://bugs.python.org/issue?@action=redirect&bpo=25507]: fix incorrect change in IOBinding that prevented printing. Augment IOBinding htest to include all major IOBinding functions.
- bpo-25905 [https://bugs.python.org/issue?@action=redirect&bpo=25905]: Revert unwanted conversion of ‘ to ’ RIGHT SINGLE QUOTATION MARK in README.txt and open this and NEWS.txt with ‘ascii’. Re-encode CREDITS.txt to utf-8 and open it with ‘utf-8’.
- bpo-15348 [https://bugs.python.org/issue?@action=redirect&bpo=15348]: Stop the debugger engine (normally in a user process) before closing the debugger window (running in the IDLE process). This prevents the RuntimeErrors that were being caught and ignored.

- bpo-24455 [https://bugs.python.org/issue?@action=redirect&bpo=24455]: Prevent IDLE from hanging when a) closing the shell while the debugger is active (15347); b) closing the debugger with the [X] button (15348); and c) activating the debugger when already active (24455). The patch by Mark Roseman does this by making two changes. 1. Suspend and resume the gui.interaction method with the tcl vwait mechanism intended for this purpose (instead of root.mainloop & .quit). 2. In gui.run, allow any existing interaction to terminate first.
- Change 'The program' to 'Your program' in an IDLE 'kill program?' message to make it clearer that the program referred to is the currently running user program, not IDLE itself.
- bpo-24750 [https://bugs.python.org/issue?@action=redirect&bpo=24750]: Improve the appearance of the IDLE editor window status bar. Patch by Mark Roseman.
- bpo-25313 [https://bugs.python.org/issue?@action=redirect&bpo=25313]: Change the handling of new built-in text color themes to better address the compatibility problem introduced by the addition of IDLE Dark. Consistently use the revised idleConf.CurrentTheme everywhere in idlelib.
- bpo-24782 [https://bugs.python.org/issue?@action=redirect&bpo=24782]: Extension configuration is now a tab in the IDLE Preferences dialog rather than a separate dialog. The former tabs are now a sorted list. Patch by Mark Roseman.
- bpo-22726 [https://bugs.python.org/issue?@action=redirect&bpo=22726]: Re-activate the config dialog help button with some content about the other buttons and the new IDLE Dark theme.
- bpo-24820 [https://bugs.python.org/issue?@action=redirect&bpo=24820]: IDLE now has an 'IDLE Dark' built-in text color theme. It is more or less IDLE Classic

inverted, with a cobalt blue background. Strings, comments, keywords, … are still green, red, orange, … . To use it with IDLEs released before November 2015, hit the ‘Save as New Custom Theme’ button and enter a new name, such as ‘Custom Dark’. The custom theme will work with any IDLE release, and can be modified.

- bpo-25224 [https://bugs.python.org/issue?@action=redirect&bpo=25224]: README.txt is now an idlelib index for IDLE developers and curious users. The previous user content is now in the IDLE doc chapter. ‘IDLE’ now means ‘Integrated Development and Learning Environment’.
- bpo-24820 [https://bugs.python.org/issue?@action=redirect&bpo=24820]: Users can now set breakpoint colors in Settings -> Custom Highlighting. Original patch by Mark Roseman.
- bpo-24972 [https://bugs.python.org/issue?@action=redirect&bpo=24972]: Inactive selection background now matches active selection background, as configured by users, on all systems. Found items are now always highlighted on Windows. Initial patch by Mark Roseman.
- bpo-24570 [https://bugs.python.org/issue?@action=redirect&bpo=24570]: Idle: make calltip and completion boxes appear on Macs affected by a tk regression. Initial patch by Mark Roseman.
- bpo-24988 [https://bugs.python.org/issue?@action=redirect&bpo=24988]: Idle ScrolledList context menus (used in debugger) now work on Mac Aqua. Patch by Mark Roseman.
- bpo-24801 [https://bugs.python.org/issue?@action=redirect&bpo=24801]: Make right-click for context menu work on Mac Aqua. Patch by Mark Roseman.
- bpo-25173 [https://bugs.python.org/issue?@action=redirect&bpo=25173]: Associate tkinter messageboxes with a specific widget. For Mac OSX, make them a ‘sheet’. Patch by Mark Roseman.

- bpo-25198 [https://bugs.python.org/issue?@action=redirect&bpo=25198]: Enhance the initial html viewer now used for Idle Help. Properly indent fixed-pitch text (patch by Mark Roseman). Give code snippet a very Sphinx-like light blueish-gray background. Re-use initial width and height set by users for shell and editor. When the Table of Contents (TOC) menu is used, put the section header at the top of the screen.
- bpo-25225 [https://bugs.python.org/issue?@action=redirect&bpo=25225]: Condense and rewrite Idle doc section on text colors.
- bpo-21995 [https://bugs.python.org/issue?@action=redirect&bpo=21995]: Explain some differences between IDLE and console Python.
- bpo-22820 [https://bugs.python.org/issue?@action=redirect&bpo=22820]: Explain need for *print* when running file from Idle editor.
- bpo-25224 [https://bugs.python.org/issue?@action=redirect&bpo=25224]: Doc: augment Idle feature list and no-subprocess section.
- bpo-25219 [https://bugs.python.org/issue?@action=redirect&bpo=25219]: Update doc for Idle command line options. Some were missing and notes were not correct.
- bpo-24861 [https://bugs.python.org/issue?@action=redirect&bpo=24861]: Most of idlelib is private and subject to change. Use idleib.idle.* to start Idle. See idlelib.__init__.__doc__.
- bpo-25199 [https://bugs.python.org/issue?@action=redirect&bpo=25199]: Idle: add synchronization comments for future maintainers.
- bpo-16893 [https://bugs.python.org/issue?@action=redirect&bpo=16893]: Replace help.txt with help.html for Idle doc display. The new idlelib/help.html is rstripped Doc/build/html/library/idle.html. It looks better than help.txt and will better document Idle as released.

The tkinter html viewer that works for this file was written by Rose Roseman. The now unused EditorWindow.HelpDialog class and helt.txt file are deprecated.

- bpo-24199 [https://bugs.python.org/issue?@action=redirect&bpo=24199]: Deprecate unused idlelib.idlever with possible removal in 3.6.
- bpo-24790 [https://bugs.python.org/issue?@action=redirect&bpo=24790]: Remove extraneous code (which also create 2 & 3 conflicts).

Documentation

- bpo-26736 [https://bugs.python.org/issue?@action=redirect&bpo=26736]: Used HTTPS for external links in the documentation if possible.
- bpo-6953 [https://bugs.python.org/issue?@action=redirect&bpo=6953]: Rework the Readline module documentation to group related functions together, and add more details such as what underlying Readline functions and variables are accessed.
- bpo-23606 [https://bugs.python.org/issue?@action=redirect&bpo=23606]: Adds note to ctypes documentation regarding cdll.msvcrt.
- bpo-24952 [https://bugs.python.org/issue?@action=redirect&bpo=24952]: Clarify the default size argument of stack_size() in the “threading” and “_thread” modules. Patch from Mattip.
- bpo-26014 [https://bugs.python.org/issue?@action=redirect&bpo=26014]: Update 3.x packaging documentation: * “See also” links to the new docs are now provided in the legacy pages * links to setuptools documentation have been updated

Tests

- bpo-21916 [https://bugs.python.org/issue?@action=redirect&bpo=21916]: Added tests for the turtle module. Patch by ingrid, Gregory Loyse and Jelle Zijlstra.
- bpo-26295 [https://bugs.python.org/issue?@action=redirect&bpo=26295]: When using “python3 -m test –testdir=TESTDIR”, regrtest doesn’t add “test.” prefix to test module names.
- bpo-26523 [https://bugs.python.org/issue?@action=redirect&bpo=26523]: The multiprocessing thread pool (multiprocessing.dummy.Pool) was untested.
- bpo-26015 [https://bugs.python.org/issue?@action=redirect&bpo=26015]: Added new tests for pickling iterators of mutable sequences.
- bpo-26325 [https://bugs.python.org/issue?@action=redirect&bpo=26325]: Added test.support.check_no_resource_warning() to check that no ResourceWarning is emitted.
- bpo-25940 [https://bugs.python.org/issue?@action=redirect&bpo=25940]: Changed test_ssl to use its internal local server more. This avoids relying on svn.python.org, which recently changed root certificate.
- bpo-25616 [https://bugs.python.org/issue?@action=redirect&bpo=25616]: Tests for OrderedDict are extracted from test_collections into separate file test_ordered_dict.
- bpo-25449 [https://bugs.python.org/issue?@action=redirect&bpo=25449]: Added tests for OrderedDict subclasses.
- bpo-25188 [https://bugs.python.org/issue?@action=redirect&bpo=25188]: Add -P/–pgo to test.regrtest to suppress error output when running the test suite for the purposes of a PGO build. Initial patch by Alecsandru Patrascu.
- bpo-22806 [https://bugs.python.org/issue?@action=redirect&bpo=22806]: Add `python -m test --list-tests` command to list tests.

- bpo-18174 [https://bugs.python.org/issue?@action=redirect&bpo=18174]: `python -m test --huntrleaks ...` now also checks for leak of file descriptors. Patch written by Richard Oudkerk.
- bpo-25260 [https://bugs.python.org/issue?@action=redirect&bpo=25260]: Fix `python -m test --coverage` on Windows. Remove the list of ignored directories.
- `PCbuild\rt.bat` now accepts an unlimited number of arguments to pass along to regrtest.py. Previously there was a limit of 9.
- bpo-26583 [https://bugs.python.org/issue?@action=redirect&bpo=26583]: Skip test_timestamp_overflow in test_import if bytecode files cannot be written.

Build

- bpo-21277 [https://bugs.python.org/issue?@action=redirect&bpo=21277]: Don’t try to link _ctypes with a ffi_convenience library.
- bpo-26884 [https://bugs.python.org/issue?@action=redirect&bpo=26884]: Fix linking extension modules for cross builds. Patch by Xavier de Gaye.
- bpo-26932 [https://bugs.python.org/issue?@action=redirect&bpo=26932]: Fixed support of RTLD_* constants defined as enum values, not via macros (in particular on Android). Patch by Chi Hsuan Yen.
- bpo-22359 [https://bugs.python.org/issue?@action=redirect&bpo=22359]: Disable the rules for running _freeze_importlib and pgen when cross-compiling. The output of these programs is normally saved with the source code anyway, and is still regenerated when doing a native build. Patch by Xavier de Gaye.
- bpo-21668 [https://bugs.python.org/issue?@action=redirect&bpo=21668]: Link audioop, _datetime, _ctypes_test modules to libm, except on Mac OS X. Patch written by Chi Hsuan Yen.

- bpo-25702 [https://bugs.python.org/issue?@action=redirect&bpo=25702]: A –with-lto configure option has been added that will enable link time optimizations at build time during a make profile-opt. Some compilers and toolchains are known to not produce stable code when using LTO, be sure to test things thoroughly before relying on it. It can provide a few % speed up over profile-opt alone.
- bpo-26624 [https://bugs.python.org/issue?@action=redirect&bpo=26624]: Adds validation of ucrtbase[d].dll version with warning for old versions.
- bpo-17603 [https://bugs.python.org/issue?@action=redirect&bpo=17603]: Avoid error about nonexistent fileblocks.o file by using a lower-level check for st_blocks in struct stat.
- bpo-26079 [https://bugs.python.org/issue?@action=redirect&bpo=26079]: Fixing the build output folder for tix-8.4.3.6. Patch by Bjoern Thiel.
- bpo-26465 [https://bugs.python.org/issue?@action=redirect&bpo=26465]: Update Windows builds to use OpenSSL 1.0.2g.
- bpo-25348 [https://bugs.python.org/issue?@action=redirect&bpo=25348]: Added `--pgo` and `--pgo-job` arguments to `PCbuild\build.bat` for building with Profile-Guided Optimization. The old `PCbuild\build_pgo.bat` script is removed.
- bpo-25827 [https://bugs.python.org/issue?@action=redirect&bpo=25827]: Add support for building with ICC to `configure`, including a new `--with-icc` flag.
- bpo-25696 [https://bugs.python.org/issue?@action=redirect&bpo=25696]: Fix installation of Python on UNIX with make -j9.
- bpo-24986 [https://bugs.python.org/issue?@action=redirect&bpo=24986]: It is now possible to build Python on Windows without errors when external libraries are not available.

- bpo-24421 [https://bugs.python.org/issue?@action=redirect&bpo=24421]: Compile Modules/_math.c once, before building extensions. Previously it could fail to compile properly if the math and cmath builds were concurrent.
- bpo-26465 [https://bugs.python.org/issue?@action=redirect&bpo=26465]: Update OS X 10.5+ 32-bit-only installer to build and link with OpenSSL 1.0.2g.
- bpo-26268 [https://bugs.python.org/issue?@action=redirect&bpo=26268]: Update Windows builds to use OpenSSL 1.0.2f.
- bpo-25136 [https://bugs.python.org/issue?@action=redirect&bpo=25136]: Support Apple Xcode 7's new textual SDK stub libraries.
- bpo-24324 [https://bugs.python.org/issue?@action=redirect&bpo=24324]: Do not enable unreachable code warnings when using gcc as the option does not work correctly in older versions of gcc and has been silently removed as of gcc-4.5.

Windows

- bpo-27053 [https://bugs.python.org/issue?@action=redirect&bpo=27053]: Updates make_zip.py to correctly generate library ZIP file.
- bpo-26268 [https://bugs.python.org/issue?@action=redirect&bpo=26268]: Update the prepare_ssl.py script to handle OpenSSL releases that don't include the contents of the include directory (that is, 1.0.2e and later).
- bpo-26071 [https://bugs.python.org/issue?@action=redirect&bpo=26071]: bdist_wininst created binaries fail to start and find 32bit Python
- bpo-26073 [https://bugs.python.org/issue?@action=redirect&bpo=26073]: Update the list of magic numbers in launcher

- bpo-26065 [https://bugs.python.org/issue?@action=redirect&bpo=26065]: Excludes venv from library when generating embeddable distro.
- bpo-25022 [https://bugs.python.org/issue?@action=redirect&bpo=25022]: Removed very outdated PC/example_nt/ directory.

Tools/Demos

- bpo-26799 [https://bugs.python.org/issue?@action=redirect&bpo=26799]: Fix python-gdb.py: don't get C types once when the Python code is loaded, but get C types on demand. The C types can change if python-gdb.py is loaded before the Python executable. Patch written by Thomas Ilsche.
- bpo-26271 [https://bugs.python.org/issue?@action=redirect&bpo=26271]: Fix the Freeze tool to properly use flags passed through configure. Patch by Daniel Shaulov.
- bpo-26489 [https://bugs.python.org/issue?@action=redirect&bpo=26489]: Add dictionary unpacking support to Tools/parser/unparse.py. Patch by Guo Ci Teo.
- bpo-26316 [https://bugs.python.org/issue?@action=redirect&bpo=26316]: Fix variable name typo in Argument Clinic.
- bpo-25440 [https://bugs.python.org/issue?@action=redirect&bpo=25440]: Fix output of python-config –extension-suffix.
- bpo-25154 [https://bugs.python.org/issue?@action=redirect&bpo=25154]: The pyvenv script has been deprecated in favour of `python3 -m venv`.

C API

- bpo-26312 [https://bugs.python.org/issue?@action=redirect&bpo=26312]: SystemError is now raised in all programming bugs with using PyArg_ParseTupleAndKeywords(). RuntimeError did raised before in some programming bugs.

- bpo-26198 [https://bugs.python.org/issue?@action=redirect&bpo=26198]: ValueError is now raised instead of TypeError on buffer overflow in parsing “es#” and “et#” format units. SystemError is now raised instead of TypeError on programmatical error in parsing format string.

Python 3.5.5 final

Release date: 2018-02-04

There were no new changes in version 3.5.5.

Python 3.5.5 release candidate 1

Release date: 2018-01-23

Security

- bpo-32551 [https://bugs.python.org/issue?@action=redirect&bpo=32551]: The `sys.path[0]` initialization change for bpo-29139 [https://bugs.python.org/issue?@action=redirect&bpo=29139] caused a regression by revealing an inconsistency in how sys.path is initialized when executing `__main__` from a zipfile, directory, or other import location. This is considered a potential security issue, as it may lead to privileged processes unexpectedly loading code from user controlled directories in situations where that was not previously the case. The interpreter now consistently avoids ever adding the import location’s parent directory to `sys.path`, and ensures no other `sys.path` entries are inadvertently modified when inserting the import location named on the command line. (Originally reported as bpo-29723 [https://bugs.python.org/issue?@action=redirect&bpo=29723] against Python 3.6rc1, but it was missed at the time that the then upcoming Python 3.5.4 release would also be affected)

- bpo-30657 [https://bugs.python.org/issue?@action=redirect&bpo=30657]: Fixed possible integer overflow in PyBytes_DecodeEscape, **CVE 2017-1000158** [https://www.cve.org/CVERecord?id=CVE-2017-1000158]. Original patch by Jay Bosamiya; rebased to Python 3 by Miro Hrončok.
- bpo-30947 [https://bugs.python.org/issue?@action=redirect&bpo=30947]: Upgrade libexpat embedded copy from version 2.2.1 to 2.2.3 to get security fixes.

Core and Builtins

- bpo-31095 [https://bugs.python.org/issue?@action=redirect&bpo=31095]: Fix potential crash during GC caused by `tp_dealloc` which doesn’t call `PyObject_GC_UnTrack()`.

Library

- bpo-32072 [https://bugs.python.org/issue?@action=redirect&bpo=32072]: Fixed issues with binary plists: Fixed saving bytearrays. Identical objects will be saved only once. Equal references will be load as identical objects. Added support for saving and loading recursive data structures.
- bpo-31170 [https://bugs.python.org/issue?@action=redirect&bpo=31170]: expat: Update libexpat from 2.2.3 to 2.2.4. Fix copying of partial characters for UTF-8 input (libexpat bug 115): https://github.com/libexpat/libexpat/issues/115

Python 3.5.4 final

Release date: 2017-08-07

Library

- bpo-30119 [https://bugs.python.org/issue?@action=redirect&bpo=30119]: ftplib.FTP.putline() now throws ValueError on commands that contains CR or LF. Patch by Donghee Na.

Python 3.5.4 release candidate 1

Release date: 2017-07-23

Security

- bpo-30730 [https://bugs.python.org/issue?@action=redirect&bpo=30730]: Prevent environment variables injection in subprocess on Windows. Prevent passing other environment variables and command arguments.
- bpo-30694 [https://bugs.python.org/issue?@action=redirect&bpo=30694]: Upgrade expat copy from 2.2.0 to 2.2.1 to get fixes of multiple security vulnerabilities including: **CVE 2017-9233** [https://www.cve.org/CVERecord?id=CVE-2017-9233] (External entity infinite loop DoS), **CVE 2016-9063** [https://www.cve.org/CVERecord?id=CVE-2016-9063] (Integer overflow, re-fix), **CVE 2016-0718** [https://www.cve.org/CVERecord?id=CVE-2016-0718] (Fix regression bugs from 2.2.0’s fix to **CVE 2016-0718** [https://www.cve.org/CVERecord?id=CVE-2016-0718]) and **CVE 2012-0876** [https://www.cve.org/CVERecord?id=CVE-2012-0876] (Counter hash flooding with SipHash). Note: the **CVE 2016-5300** [https://www.cve.org/CVERecord?id=CVE-2016-5300] (Use os-specific entropy sources like getrandom) doesn’t

impact Python, since Python already gets entropy from the OS to set the expat secret using `XML_SetHashSalt()`.

- bpo-30500 [https://bugs.python.org/issue?@action=redirect&bpo=30500]: Fix urllib.parse.splithost() to correctly parse fragments. For example, `splithost('//127.0.0.1#@evil.com/')` now correctly returns the `127.0.0.1` host, instead of treating `@evil.com` as the host in an authentication (`login@host`).
- bpo-29591 [https://bugs.python.org/issue?@action=redirect&bpo=29591]: Update expat copy from 2.1.1 to 2.2.0 to get fixes of **CVE 2016-0718** [https://www.cve.org/CVERecord?id=CVE-2016-0718] and **CVE 2016-4472** [https://www.cve.org/CVERecord?id=CVE-2016-4472]. See https://sourceforge.net/p/expat/bugs/537/ for more information.

Core and Builtins

- bpo-30876 [https://bugs.python.org/issue?@action=redirect&bpo=30876]: Relative import from unloaded package now reimports the package instead of failing with SystemError. Relative import from non-package now fails with ImportError rather than SystemError.
- bpo-30765 [https://bugs.python.org/issue?@action=redirect&bpo=30765]: Avoid blocking in pthread_mutex_lock() when PyThread_acquire_lock() is asked not to block.
- bpo-27945 [https://bugs.python.org/issue?@action=redirect&bpo=27945]: Fixed various segfaults with dict when input collections are mutated during searching, inserting or comparing. Based on patches by Duane Griffin and Tim Mitchell.
- bpo-25794 [https://bugs.python.org/issue?@action=redirect&bpo=25794]: Fixed type.__setattr__() and type.__delattr__() for non-interned attribute names. Based on patch by Eryk Sun.

- bpo-29935 [https://bugs.python.org/issue?@action=redirect&bpo=29935]: Fixed error messages in the index() method of tuple, list and deque when pass indices of wrong type.
- bpo-28876 [https://bugs.python.org/issue?@action=redirect&bpo=28876]: `bool(range)` works even if `len(range)` raises **`OverflowError`**.
- bpo-29600 [https://bugs.python.org/issue?@action=redirect&bpo=29600]: Fix wrapping coroutine return values in StopIteration.
- bpo-29537 [https://bugs.python.org/issue?@action=redirect&bpo=29537]: Restore runtime compatibility with bytecode files generated by CPython 3.5.0 to 3.5.2, and adjust the eval loop to avoid the problems that could be caused by the malformed variant of the BUILD_MAP_UNPACK_WITH_CALL opcode that they may contain. Patch by Petr Viktorin, Serhiy Storchaka, and Nick Coghlan.
- bpo-28598 [https://bugs.python.org/issue?@action=redirect&bpo=28598]: Support __rmod__ for subclasses of str being called before str.__mod__. Patch by Martijn Pieters.
- bpo-29602 [https://bugs.python.org/issue?@action=redirect&bpo=29602]: Fix incorrect handling of signed zeros in complex constructor for complex subclasses and for inputs having a __complex__ method. Patch by Serhiy Storchaka.
- bpo-29347 [https://bugs.python.org/issue?@action=redirect&bpo=29347]: Fixed possibly dereferencing undefined pointers when creating weakref objects.
- bpo-29438 [https://bugs.python.org/issue?@action=redirect&bpo=29438]: Fixed use-after-free problem in key sharing dict.
- bpo-29319 [https://bugs.python.org/issue?@action=redirect&bpo=29319]: Prevent RunMainFromImporter overwriting sys.path[0].

- bpo-29337 [https://bugs.python.org/issue?@action=redirect&bpo=29337]: Fixed possible BytesWarning when compare the code objects. Warnings could be emitted at compile time.
- bpo-29478 [https://bugs.python.org/issue?@action=redirect&bpo=29478]: If max_line_length=None is specified while using the Compat32 policy, it is no longer ignored. Patch by Mircea Cosbuc.

Library

- bpo-29403 [https://bugs.python.org/issue?@action=redirect&bpo=29403]: Fix `unittest.mock`'s autospec to not fail on method-bound builtin functions. Patch by Aaron Gallagher.
- bpo-30961 [https://bugs.python.org/issue?@action=redirect&bpo=30961]: Fix decrementing a borrowed reference in tracemalloc.
- bpo-30886 [https://bugs.python.org/issue?@action=redirect&bpo=30886]: Fix multiprocessing.Queue.join_thread(): it now waits until the thread completes, even if the thread was started by the same process which created the queue.
- bpo-29854 [https://bugs.python.org/issue?@action=redirect&bpo=29854]: Fix segfault in readline when using readline's history-size option. Patch by Nir Soffer.
- bpo-30807 [https://bugs.python.org/issue?@action=redirect&bpo=30807]: signal.setitimer() may disable the timer when passed a tiny value. Tiny values (such as 1e-6) are valid non-zero values for setitimer(), which is specified as taking microsecond-resolution intervals. However, on some platform, our conversion routine could convert 1e-6 into a zero interval, therefore disabling the timer instead of (re-)scheduling it.
- bpo-30441 [https://bugs.python.org/issue?@action=redirect&bpo=30441]: Fix bug when modifying

os.environ while iterating over it

- bpo-30532 [https://bugs.python.org/issue?@action=redirect&bpo=30532]: Fix email header value parser dropping folding white space in certain cases.
- bpo-29169 [https://bugs.python.org/issue?@action=redirect&bpo=29169]: Update zlib to 1.2.11.
- bpo-30879 [https://bugs.python.org/issue?@action=redirect&bpo=30879]: os.listdir() and os.scandir() now emit bytes names when called with bytes-like argument.
- bpo-30746 [https://bugs.python.org/issue?@action=redirect&bpo=30746]: Prohibited the ‘=’ character in environment variable names in `os.putenv()` and `os.spawn*()`.
- bpo-29755 [https://bugs.python.org/issue?@action=redirect&bpo=29755]: Fixed the lgettext() family of functions in the gettext module. They now always return bytes.
- bpo-30645 [https://bugs.python.org/issue?@action=redirect&bpo=30645]: Fix path calculation in imp.load_package(), fixing it for cases when a package is only shipped with bytecodes. Patch by Alexandru Ardelean.
- bpo-23890 [https://bugs.python.org/issue?@action=redirect&bpo=23890]: unittest.TestCase.assertRaises() now manually breaks a reference cycle to not keep objects alive longer than expected.
- bpo-30149 [https://bugs.python.org/issue?@action=redirect&bpo=30149]: inspect.signature() now supports callables with variable-argument parameters wrapped with partialmethod. Patch by Donghee Na.
- bpo-29931 [https://bugs.python.org/issue?@action=redirect&bpo=29931]: Fixed comparison check for ipaddress.ip_interface objects. Patch by Sanjay Sundaresan.
- bpo-24484 [https://bugs.python.org/issue?@action=redirect&bpo=24484]: Avoid race condition in

multiprocessing cleanup.

- bpo-28994 [https://bugs.python.org/issue?@action=redirect&bpo=28994]: The traceback no longer displayed for SystemExit raised in a callback registered by atexit.
- bpo-30508 [https://bugs.python.org/issue?@action=redirect&bpo=30508]: Don’t log exceptions if Task/Future “cancel()” method was called.
- bpo-28556 [https://bugs.python.org/issue?@action=redirect&bpo=28556]: Updates to typing module: Add generic AsyncContextManager, add support for ContextManager on all versions. Original PRs by Jelle Zijlstra and Ivan Levkivskyi
- bpo-29870 [https://bugs.python.org/issue?@action=redirect&bpo=29870]: Fix ssl sockets leaks when connection is aborted in asyncio/ssl implementation. Patch by Michaël Sghaïer.
- bpo-29743 [https://bugs.python.org/issue?@action=redirect&bpo=29743]: Closing transport during handshake process leaks open socket. Patch by Nikolay Kim
- bpo-27585 [https://bugs.python.org/issue?@action=redirect&bpo=27585]: Fix waiter cancellation in asyncio.Lock. Patch by Mathieu Sornay.
- bpo-30418 [https://bugs.python.org/issue?@action=redirect&bpo=30418]: On Windows, subprocess.Popen.communicate() now also ignore EINVAL on stdin.write() if the child process is still running but closed the pipe.
- bpo-30378 [https://bugs.python.org/issue?@action=redirect&bpo=30378]: Fix the problem that logging.handlers.SysLogHandler cannot handle IPv6 addresses.
- bpo-29960 [https://bugs.python.org/issue?@action=redirect&bpo=29960]: Preserve generator state when

_random.Random.setstate() raises an exception. Patch by Bryan Olson.

- bpo-30414 [https://bugs.python.org/issue?@action=redirect&bpo=30414]: multiprocessing.Queue._feed background running thread do not break from main loop on exception.
- bpo-30003 [https://bugs.python.org/issue?@action=redirect&bpo=30003]: Fix handling escape characters in HZ codec. Based on patch by Ma Lin.
- bpo-30301 [https://bugs.python.org/issue?@action=redirect&bpo=30301]: Fix AttributeError when using SimpleQueue.empty() under *spawn* and *forkserver* start methods.
- bpo-30329 [https://bugs.python.org/issue?@action=redirect&bpo=30329]: imaplib and poplib now catch the Windows socket WSAEINVAL error (code 10022) on shutdown(SHUT_RDWR): An invalid operation was attempted. This error occurs sometimes on SSL connections.
- bpo-30375 [https://bugs.python.org/issue?@action=redirect&bpo=30375]: Warnings emitted when compile a regular expression now always point to the line in the user code. Previously they could point into inners of the re module if emitted from inside of groups or conditionals.
- bpo-30048 [https://bugs.python.org/issue?@action=redirect&bpo=30048]: Fixed `Task.cancel()` can be ignored when the task is running coroutine and the coroutine returned without any more `await`.
- bpo-29990 [https://bugs.python.org/issue?@action=redirect&bpo=29990]: Fix range checking in GB18030 decoder. Original patch by Ma Lin.
- bpo-26293 [https://bugs.python.org/issue?@action=redirect&bpo=26293]: Change resulted because of zipfile breakage. (See also: bpo-29094 [https://bugs.python.org/issue?@action=redirect&bpo=29094])

- bpo-30243 [https://bugs.python.org/issue?@action=redirect&bpo=30243]: Removed the __init__ methods of _json's scanner and encoder. Misusing them could cause memory leaks or crashes. Now scanner and encoder objects are completely initialized in the __new__ methods.
- bpo-30185 [https://bugs.python.org/issue?@action=redirect&bpo=30185]: Avoid KeyboardInterrupt tracebacks in forkserver helper process when Ctrl-C is received.
- bpo-28556 [https://bugs.python.org/issue?@action=redirect&bpo=28556]: Various updates to typing module: add typing.NoReturn type, use WrapperDescriptorType, minor bug-fixes. Original PRs by Jim Fasarakis-Hilliard and Ivan Levkivskyi.
- bpo-30205 [https://bugs.python.org/issue?@action=redirect&bpo=30205]: Fix getsockname() for unbound AF_UNIX sockets on Linux.
- bpo-30070 [https://bugs.python.org/issue?@action=redirect&bpo=30070]: Fixed leaks and crashes in errors handling in the parser module.
- bpo-30061 [https://bugs.python.org/issue?@action=redirect&bpo=30061]: Fixed crashes in IOBase methods __next__() and readlines() when readline() or __next__() respectively return non-sizeable object. Fixed possible other errors caused by not checking results of PyObject_Size(), PySequence_Size(), or PyMapping_Size().
- bpo-30068 [https://bugs.python.org/issue?@action=redirect&bpo=30068]: _io._IOBase.readlines will check if it's closed first when hint is present.
- bpo-29694 [https://bugs.python.org/issue?@action=redirect&bpo=29694]: Fixed race condition in pathlib mkdir with flags parents=True. Patch by Armin Rigo.
- bpo-29692 [https://bugs.python.org/issue?@action=redirect&bpo=29692]: Fixed arbitrary unchaining of

RuntimeError exceptions in contextlib.contextmanager. Patch by Siddharth Velankar.

- bpo-29998 [https://bugs.python.org/issue?@action=redirect&bpo=29998]: Pickling and copying ImportError now preserves name and path attributes.
- bpo-29942 [https://bugs.python.org/issue?@action=redirect&bpo=29942]: Fix a crash in itertools.chain.from_iterable when encountering long runs of empty iterables.
- bpo-27863 [https://bugs.python.org/issue?@action=redirect&bpo=27863]: Fixed multiple crashes in ElementTree caused by race conditions and wrong types.
- bpo-28699 [https://bugs.python.org/issue?@action=redirect&bpo=28699]: Fixed a bug in pools in multiprocessing.pool that raising an exception at the very first of an iterable may swallow the exception or make the program hang. Patch by Davin Potts and Xiang Zhang.
- bpo-25803 [https://bugs.python.org/issue?@action=redirect&bpo=25803]: Avoid incorrect errors raised by Path.mkdir(exist_ok=True) when the OS gives priority to errors such as EACCES over EEXIST.
- bpo-29861 [https://bugs.python.org/issue?@action=redirect&bpo=29861]: Release references to tasks, their arguments and their results as soon as they are finished in multiprocessing.Pool.
- bpo-29884 [https://bugs.python.org/issue?@action=redirect&bpo=29884]: faulthandler: Restore the old sigaltstack during teardown. Patch by Christophe Zeitouny.
- bpo-25455 [https://bugs.python.org/issue?@action=redirect&bpo=25455]: Fixed crashes in repr of recursive buffered file-like objects.
- bpo-29800 [https://bugs.python.org/issue?@action=redirect&bpo=29800]: Fix crashes in partial.__repr__ if

the keys of partial.keywords are not strings. Patch by Michael Seifert.

- bpo-29742 [https://bugs.python.org/issue?@action=redirect&bpo=29742]: get_extra_info() raises exception if get called on closed ssl transport. Patch by Nikolay Kim.
- bpo-8256 [https://bugs.python.org/issue?@action=redirect&bpo=8256]: Fixed possible failing or crashing input() if attributes “encoding” or “errors” of sys.stdin or sys.stdout are not set or are not strings.
- bpo-28298 [https://bugs.python.org/issue?@action=redirect&bpo=28298]: Fix a bug that prevented array ‘Q’, ‘L’ and ‘I’ from accepting big intables (objects that have __int__) as elements. Patch by Oren Milman.
- bpo-29615 [https://bugs.python.org/issue?@action=redirect&bpo=29615]: SimpleXMLRPCDispatcher no longer chains KeyError (or any other exception) to exception(s) raised in the dispatched methods. Patch by Petr Motejlek.
- bpo-29704 [https://bugs.python.org/issue?@action=redirect&bpo=29704]: asyncio.subprocess.SubprocessStreamProtocol no longer closes before all pipes are closed.
- bpo-29703 [https://bugs.python.org/issue?@action=redirect&bpo=29703]: Fix asyncio to support instantiation of new event loops in child processes.
- bpo-29376 [https://bugs.python.org/issue?@action=redirect&bpo=29376]: Fix assertion error in threading._DummyThread.is_alive().
- bpo-29110 [https://bugs.python.org/issue?@action=redirect&bpo=29110]: Fix file object leak in aifc.open() when file is given as a filesystem path and is not in valid AIFF format. Patch by Anthony Zhang.
- bpo-28961 [https://bugs.python.org/issue?@action=redirect&bpo=28961]: Fix unittest.mock._Call helper:

don’t ignore the name parameter anymore. Patch written by Jiajun Huang.

- bpo-29532 [https://bugs.python.org/issue?@action=redirect&bpo=29532]: Altering a kwarg dictionary passed to functools.partial() no longer affects a partial object after creation.
- bpo-28556 [https://bugs.python.org/issue?@action=redirect&bpo=28556]: Various updates to typing module: typing.Counter, typing.ChainMap, improved ABC caching, etc. Original PRs by Jelle Zijlstra, Ivan Levkivskyi, Manuel Krebber, and Łukasz Langa.
- bpo-29100 [https://bugs.python.org/issue?@action=redirect&bpo=29100]: Fix datetime.fromtimestamp() regression introduced in Python 3.6.0: check minimum and maximum years.
- bpo-29519 [https://bugs.python.org/issue?@action=redirect&bpo=29519]: Fix weakref spewing exceptions during interpreter shutdown when used with a rare combination of multiprocessing and custom codecs.
- bpo-29416 [https://bugs.python.org/issue?@action=redirect&bpo=29416]: Prevent infinite loop in pathlib.Path.mkdir
- bpo-29444 [https://bugs.python.org/issue?@action=redirect&bpo=29444]: Fixed out-of-bounds buffer access in the group() method of the match object. Based on patch by WGH.
- bpo-29335 [https://bugs.python.org/issue?@action=redirect&bpo=29335]: Fix subprocess.Popen.wait() when the child process has exited to a stopped instead of terminated state (ex: when under ptrace).
- bpo-29290 [https://bugs.python.org/issue?@action=redirect&bpo=29290]: Fix a regression in argparse that help messages would wrap at non-breaking spaces.
- bpo-28735 [https://bugs.python.org/issue?@action=redirect&bpo=28735]: Fixed the comparison of mock.MagickMock with mock.ANY.

- bpo-29011 [https://bugs.python.org/issue?@action=redirect&bpo=29011]: Fix an important omission by adding Deque to the typing module.
- bpo-29219 [https://bugs.python.org/issue?@action=redirect&bpo=29219]: Fixed infinite recursion in the repr of uninitialized ctypes.CDLL instances.
- bpo-28969 [https://bugs.python.org/issue?@action=redirect&bpo=28969]: Fixed race condition in C implementation of functools.lru_cache. KeyError could be raised when cached function with full cache was simultaneously called from different threads with the same uncached arguments.
- bpo-29142 [https://bugs.python.org/issue?@action=redirect&bpo=29142]: In urllib.request, suffixes in no_proxy environment variable with leading dots could match related hostnames again (e.g. .b.c matches a.b.c). Patch by Milan Oberkirch.

Documentation

- bpo-30176 [https://bugs.python.org/issue?@action=redirect&bpo=30176]: Add missing attribute related constants in curses documentation.
- bpo-26985 [https://bugs.python.org/issue?@action=redirect&bpo=26985]: Add missing info of code object in inspect documentation.
- bpo-28929 [https://bugs.python.org/issue?@action=redirect&bpo=28929]: Link the documentation to its source file on GitHub.
- bpo-25008 [https://bugs.python.org/issue?@action=redirect&bpo=25008]: Document smtpd.py as effectively deprecated and add a pointer to aiosmtpd, a third-party asyncio-based replacement.
- bpo-26355 [https://bugs.python.org/issue?@action=redirect&bpo=26355]: Add canonical header link on

each page to corresponding major version of the documentation. Patch by Matthias Bussonnier.
- bpo-29349 [https://bugs.python.org/issue?@action=redirect&bpo=29349]: Fix Python 2 syntax in code for building the documentation.

Tests

- bpo-30822 [https://bugs.python.org/issue?@action=redirect&bpo=30822]: Fix regrtest command line parser to allow passing -u extralargefile to run test_zipfile64.
- bpo-30383 [https://bugs.python.org/issue?@action=redirect&bpo=30383]: regrtest: Enhance regrtest and backport features from the master branch. Add options: –coverage, –testdir, –list-tests (list test files, don’t run them), –list-cases (list test identifiers, don’t run them, bpo-30523 [https://bugs.python.org/issue?@action=redirect&bpo=30523]), –matchfile (load a list of test filters from a text file, bpo-30540 [https://bugs.python.org/issue?@action=redirect&bpo=30540]), –slowest (alias to –slow). Enhance output: add timestamp, test result, currently running tests, “Tests result: xxx” summary with total duration, etc. Fix reference leak hunting in regrtest, –huntrleaks: regrtest now warms up caches, create explicitly all internal singletons which are created on demand to prevent false positives when checking for reference leaks. (bpo-30675 [https://bugs.python.org/issue?@action=redirect&bpo=30675]).
- bpo-30357 [https://bugs.python.org/issue?@action=redirect&bpo=30357]: test_thread: setUp() now uses support.threading_setup() and support.threading_cleanup() to wait until threads complete to avoid random side effects on following tests. Initial patch written by Grzegorz Grzywacz.

- bpo-28087 [https://bugs.python.org/issue?@action=redirect&bpo=28087]: Skip test_asyncore and test_eintr poll failures on macOS. Skip some tests of select.poll when running on macOS due to unresolved issues with the underlying system poll function on some macOS versions.
- bpo-30197 [https://bugs.python.org/issue?@action=redirect&bpo=30197]: Enhanced functions swap_attr() and swap_item() in the test.support module. They now work when delete replaced attribute or item inside the with statement. The old value of the attribute or item (or None if it doesn't exist) now will be assigned to the target of the "as" clause, if there is one.
- bpo-29571 [https://bugs.python.org/issue?@action=redirect&bpo=29571]: to match the behaviour of the `re.LOCALE` flag, test_re.test_locale_flag now uses `locale.getpreferredencoding(False)` to determine the candidate encoding for the test regex (allowing it to correctly skip the test when the default locale encoding is a multi-byte encoding)

Build

- bpo-29243 [https://bugs.python.org/issue?@action=redirect&bpo=29243]: Prevent unnecessary rebuilding of Python during `make test`, `make install` and some other make targets when configured with `--enable-optimizations`.
- bpo-23404 [https://bugs.python.org/issue?@action=redirect&bpo=23404]: Don't regenerate generated files based on file modification time anymore: the action is now explicit. Replace `make touch` with `make regen-all`.
- bpo-29643 [https://bugs.python.org/issue?@action=redirect&bpo=29643]: Fix `--enable-optimization` didn't work.

Windows

- bpo-30687 [https://bugs.python.org/issue?@action=redirect&bpo=30687]: Locate msbuild.exe on Windows when building rather than vcvarsall.bat
- bpo-29392 [https://bugs.python.org/issue?@action=redirect&bpo=29392]: Prevent crash when passing invalid arguments into msvcrt module.

C API

- bpo-27867 [https://bugs.python.org/issue?@action=redirect&bpo=27867]: Function PySlice_GetIndicesEx() is replaced with a macro if Py_LIMITED_API is set to the value between 0x03050400 and 0x03060000 (not including) or 0x03060100 or higher.
- bpo-29083 [https://bugs.python.org/issue?@action=redirect&bpo=29083]: Fixed the declaration of some public API functions. PyArg_VaParse() and PyArg_VaParseTupleAndKeywords() were not available in limited API. PyArg_ValidateKeywordArguments(), PyArg_UnpackTuple() and Py_BuildValue() were not available in limited API of version < 3.3 when PY_SSIZE_T_CLEAN is defined.

Python 3.5.3 final

Release date: 2017-01-17

There were no code changes between 3.5.3rc1 and 3.5.3 final.

Python 3.5.3 release candidate 1

Release date: 2017-01-02

Security

- bpo-27278 [https://bugs.python.org/issue?@action=redirect&bpo=27278]: Fix os.urandom() implementation using getrandom() on Linux. Truncate size to INT_MAX and loop until we collected enough random bytes, instead of casting a directly Py_ssize_t to int.
- bpo-22636 [https://bugs.python.org/issue?@action=redirect&bpo=22636]: Avoid shell injection problems with ctypes.util.find_library().

Core and Builtins

- bpo-29073 [https://bugs.python.org/issue?@action=redirect&bpo=29073]: bytearray formatting no longer truncates on first null byte.
- bpo-28932 [https://bugs.python.org/issue?@action=redirect&bpo=28932]: Do not include <sys/random.h> if it does not exist.
- bpo-28147 [https://bugs.python.org/issue?@action=redirect&bpo=28147]: Fix a memory leak in split-table dictionaries: setattr() must not convert combined table into split table.
- bpo-25677 [https://bugs.python.org/issue?@action=redirect&bpo=25677]: Correct the positioning of the syntax error caret for indented blocks. Based on patch by Michael Layzell.
- bpo-29000 [https://bugs.python.org/issue?@action=redirect&bpo=29000]: Fixed bytes formatting of octals with zero padding in alternate form.
- bpo-28512 [https://bugs.python.org/issue?@action=redirect&bpo=28512]: Fixed setting the offset attribute of SyntaxError by PyErr_SyntaxLocationEx() and PyErr_SyntaxLocationObject().

- bpo-28991 [https://bugs.python.org/issue?@action=redirect&bpo=28991]: functools.lru_cache() was susceptible to an obscure reentrancy bug caused by a monkey-patched len() function.
- bpo-28648 [https://bugs.python.org/issue?@action=redirect&bpo=28648]: Fixed crash in Py_DecodeLocale() in debug build on Mac OS X when decode astral characters. Patch by Xiang Zhang.
- bpo-19398 [https://bugs.python.org/issue?@action=redirect&bpo=19398]: Extra slash no longer added to sys.path components in case of empty compile-time PYTHONPATH components.
- bpo-28426 [https://bugs.python.org/issue?@action=redirect&bpo=28426]: Fixed potential crash in PyUnicode_AsDecodedObject() in debug build.
- bpo-23782 [https://bugs.python.org/issue?@action=redirect&bpo=23782]: Fixed possible memory leak in _PyTraceback_Add() and exception loss in PyTraceBack_Here().
- bpo-28379 [https://bugs.python.org/issue?@action=redirect&bpo=28379]: Added sanity checks and tests for PyUnicode_CopyCharacters(). Patch by Xiang Zhang.
- bpo-28376 [https://bugs.python.org/issue?@action=redirect&bpo=28376]: The type of long range iterator is now registered as Iterator. Patch by Oren Milman.
- bpo-28376 [https://bugs.python.org/issue?@action=redirect&bpo=28376]: The constructor of range_iterator now checks that step is not 0. Patch by Oren Milman.
- bpo-26906 [https://bugs.python.org/issue?@action=redirect&bpo=26906]: Resolving special methods of uninitialized type now causes implicit initialization of the type instead of a fail.
- bpo-18287 [https://bugs.python.org/issue?@action=redirect&bpo=18287]: PyType_Ready() now checks that tp_name is not NULL. Original patch by Niklas Koep.

- bpo-24098 [https://bugs.python.org/issue?@action=redirect&bpo=24098]: Fixed possible crash when AST is changed in process of compiling it.
- bpo-28350 [https://bugs.python.org/issue?@action=redirect&bpo=28350]: String constants with null character no longer interned.
- bpo-26617 [https://bugs.python.org/issue?@action=redirect&bpo=26617]: Fix crash when GC runs during weakref callbacks.
- bpo-27942 [https://bugs.python.org/issue?@action=redirect&bpo=27942]: String constants now interned recursively in tuples and frozensets.
- bpo-21578 [https://bugs.python.org/issue?@action=redirect&bpo=21578]: Fixed misleading error message when ImportError called with invalid keyword args.
- bpo-28203 [https://bugs.python.org/issue?@action=redirect&bpo=28203]: Fix incorrect type in error message from `complex(1.0, {2:3})`. Patch by Soumya Sharma.
- bpo-27955 [https://bugs.python.org/issue?@action=redirect&bpo=27955]: Fallback on reading /dev/urandom device when the getrandom() syscall fails with EPERM, for example when blocked by SECCOMP.
- bpo-28131 [https://bugs.python.org/issue?@action=redirect&bpo=28131]: Fix a regression in zipimport’s compile_source(). zipimport should use the same optimization level as the interpreter.
- bpo-25221 [https://bugs.python.org/issue?@action=redirect&bpo=25221]: Fix corrupted result from PyLong_FromLong(0) when Python is compiled with NSMALLPOSINTS = 0.
- bpo-25758 [https://bugs.python.org/issue?@action=redirect&bpo=25758]: Prevents zipimport from unnecessarily encoding a filename (patch by Eryk Sun)
- bpo-28189 [https://bugs.python.org/issue?@action=redirect&bpo=28189]: dictitems_contains no longer

swallows compare errors. (Patch by Xiang Zhang)

- bpo-27812 [https://bugs.python.org/issue?@action=redirect&bpo=27812]: Properly clear out a generator’s frame’s backreference to the generator to prevent crashes in frame.clear().
- bpo-27811 [https://bugs.python.org/issue?@action=redirect&bpo=27811]: Fix a crash when a coroutine that has not been awaited is finalized with warnings-as-errors enabled.
- bpo-27587 [https://bugs.python.org/issue?@action=redirect&bpo=27587]: Fix another issue found by PVS-Studio: Null pointer check after use of ‘def’ in _PyState_AddModule(). Initial patch by Christian Heimes.
- bpo-26020 [https://bugs.python.org/issue?@action=redirect&bpo=26020]: set literal evaluation order did not match documented behaviour.
- bpo-27782 [https://bugs.python.org/issue?@action=redirect&bpo=27782]: Multi-phase extension module import now correctly allows the `m_methods` field to be used to add module level functions to instances of non-module types returned from `Py_create_mod`. Patch by Xiang Zhang.
- bpo-27936 [https://bugs.python.org/issue?@action=redirect&bpo=27936]: The round() function accepted a second None argument for some types but not for others. Fixed the inconsistency by accepting None for all numeric types.
- bpo-27487 [https://bugs.python.org/issue?@action=redirect&bpo=27487]: Warn if a submodule argument to “python -m” or runpy.run_module() is found in sys.modules after parent packages are imported, but before the submodule is executed.
- bpo-27558 [https://bugs.python.org/issue?@action=redirect&bpo=27558]: Fix a SystemError in the implementation of “raise” statement. In a brand new

thread, raise a RuntimeError since there is no active exception to reraise. Patch written by Xiang Zhang.

- bpo-27419 [https://bugs.python.org/issue?@action=redirect&bpo=27419]: Standard __import__() no longer look up "__import__" in globals or builtins for importing submodules or "from import". Fixed handling an error of non-string package name.
- bpo-27083 [https://bugs.python.org/issue?@action=redirect&bpo=27083]: Respect the PYTHONCASEOK environment variable under Windows.
- bpo-27514 [https://bugs.python.org/issue?@action=redirect&bpo=27514]: Make having too many statically nested blocks a SyntaxError instead of SystemError.
- bpo-27473 [https://bugs.python.org/issue?@action=redirect&bpo=27473]: Fixed possible integer overflow in bytes and bytearray concatenations. Patch by Xiang Zhang.
- bpo-27507 [https://bugs.python.org/issue?@action=redirect&bpo=27507]: Add integer overflow check in bytearray.extend(). Patch by Xiang Zhang.
- bpo-27581 [https://bugs.python.org/issue?@action=redirect&bpo=27581]: Don't rely on wrapping for overflow check in PySequence_Tuple(). Patch by Xiang Zhang.
- bpo-27443 [https://bugs.python.org/issue?@action=redirect&bpo=27443]: __length_hint__() of bytearray iterators no longer return a negative integer for a resized bytearray.
- bpo-27942 [https://bugs.python.org/issue?@action=redirect&bpo=27942]: Fix memory leak in codeobject.c

Library

- bpo-15812 [https://bugs.python.org/issue?@action=redirect&bpo=15812]: inspect.getframeinfo() now correctly shows the first line of a context. Patch by Sam Breese.
- bpo-29094 [https://bugs.python.org/issue?@action=redirect&bpo=29094]: Offsets in a ZIP file created with extern file object and modes “w” and “x” now are relative to the start of the file.
- bpo-13051 [https://bugs.python.org/issue?@action=redirect&bpo=13051]: Fixed recursion errors in large or resized curses.textpad.Textbox. Based on patch by Tycho Andersen.
- bpo-29119 [https://bugs.python.org/issue?@action=redirect&bpo=29119]: Fix weakrefs in the pure python version of collections.OrderedDict move_to_end() method. Contributed by Andra Bogildea.
- bpo-9770 [https://bugs.python.org/issue?@action=redirect&bpo=9770]: curses.ascii predicates now work correctly with negative integers.
- bpo-28427 [https://bugs.python.org/issue?@action=redirect&bpo=28427]: old keys should not remove new values from WeakValueDictionary when collecting from another thread.
- bpo-28923 [https://bugs.python.org/issue?@action=redirect&bpo=28923]: Remove editor artifacts from Tix.py.
- bpo-28871 [https://bugs.python.org/issue?@action=redirect&bpo=28871]: Fixed a crash when deallocate deep ElementTree.
- bpo-19542 [https://bugs.python.org/issue?@action=redirect&bpo=19542]: Fix bugs in WeakValueDictionary.setdefault() and WeakValueDictionary.pop() when a GC collection happens in another thread.
- bpo-20191 [https://bugs.python.org/issue?@action=redirect&bpo=20191]: Fixed a crash in

resource.prlimit() when pass a sequence that doesn’t own its elements as limits.

- bpo-28779 [https://bugs.python.org/issue?@action=redirect&bpo=28779]: multiprocessing.set_forkserver_preload() would crash the forkserver process if a preloaded module instantiated some multiprocessing objects such as locks.
- bpo-28847 [https://bugs.python.org/issue?@action=redirect&bpo=28847]: dbm.dumb now supports reading read-only files and no longer writes the index file when it is not changed.
- bpo-25659 [https://bugs.python.org/issue?@action=redirect&bpo=25659]: In ctypes, prevent a crash calling the from_buffer() and from_buffer_copy() methods on abstract classes like Array.
- bpo-28732 [https://bugs.python.org/issue?@action=redirect&bpo=28732]: Fix crash in os.spawnv() with no elements in args
- bpo-28485 [https://bugs.python.org/issue?@action=redirect&bpo=28485]: Always raise ValueError for negative compileall.compile_dir(workers=…) parameter, even when multithreading is unavailable.
- bpo-28387 [https://bugs.python.org/issue?@action=redirect&bpo=28387]: Fixed possible crash in _io.TextIOWrapper deallocator when the garbage collector is invoked in other thread. Based on patch by Sebastian Cufre.
- bpo-27517 [https://bugs.python.org/issue?@action=redirect&bpo=27517]: LZMA compressor and decompressor no longer raise exceptions if given empty data twice. Patch by Benjamin Fogle.
- bpo-28549 [https://bugs.python.org/issue?@action=redirect&bpo=28549]: Fixed segfault in curses’s addch() with ncurses6.
- bpo-28449 [https://bugs.python.org/issue?@action=redirect&bpo=28449]: tarfile.open() with mode “r” or

"r:" now tries to open a tar file with compression before trying to open it without compression. Otherwise it had 50% chance failed with ignore_zeros=True.

- bpo-23262 [https://bugs.python.org/issue?@action=redirect&bpo=23262]: The webbrowser module now supports Firefox 36+ and derived browsers. Based on patch by Oleg Broytman.
- bpo-27939 [https://bugs.python.org/issue?@action=redirect&bpo=27939]: Fixed bugs in tkinter.ttk.LabeledScale and tkinter.Scale caused by representing the scale as float value internally in Tk. tkinter.IntVar now works if float value is set to underlying Tk variable.
- bpo-28255 [https://bugs.python.org/issue?@action=redirect&bpo=28255]: calendar.TextCalendar().prmonth() no longer prints a space at the start of new line after printing a month's calendar. Patch by Xiang Zhang.
- bpo-20491 [https://bugs.python.org/issue?@action=redirect&bpo=20491]: The textwrap.TextWrapper class now honors non-breaking spaces. Based on patch by Kaarle Ritvanen.
- bpo-28353 [https://bugs.python.org/issue?@action=redirect&bpo=28353]: os.fwalk() no longer fails on broken links.
- bpo-25464 [https://bugs.python.org/issue?@action=redirect&bpo=25464]: Fixed HList.header_exists() in tkinter.tix module by addin a workaround to Tix library bug.
- bpo-28488 [https://bugs.python.org/issue?@action=redirect&bpo=28488]: shutil.make_archive() no longer add entry "./" to ZIP archive.
- bpo-24452 [https://bugs.python.org/issue?@action=redirect&bpo=24452]: Make webbrowser support Chrome on Mac OS X.

- bpo-20766 [https://bugs.python.org/issue?@action=redirect&bpo=20766]: Fix references leaked by pdb in the handling of SIGINT handlers.
- bpo-26293 [https://bugs.python.org/issue?@action=redirect&bpo=26293]: Fixed writing ZIP files that starts not from the start of the file. Offsets in ZIP file now are relative to the start of the archive in conforming to the specification.
- bpo-28321 [https://bugs.python.org/issue?@action=redirect&bpo=28321]: Fixed writing non-BMP characters with binary format in plistlib.
- bpo-28322 [https://bugs.python.org/issue?@action=redirect&bpo=28322]: Fixed possible crashes when unpickle itertools objects from incorrect pickle data. Based on patch by John Leitch.
- Fix possible integer overflows and crashes in the mmap module with unusual usage patterns.
- bpo-1703178 [https://bugs.python.org/issue?@action=redirect&bpo=1703178]: Fix the ability to pass the –link-objects option to the distutils build_ext command.
- bpo-28253 [https://bugs.python.org/issue?@action=redirect&bpo=28253]: Fixed calendar functions for extreme months: 0001-01 and 9999-12. Methods itermonthdays() and itermonthdays2() are reimplemented so that they don’t call itermonthdates() which can cause datetime.date under/overflow.
- bpo-28275 [https://bugs.python.org/issue?@action=redirect&bpo=28275]: Fixed possible use after free in the decompress() methods of the LZMADecompressor and BZ2Decompressor classes. Original patch by John Leitch.
- bpo-27897 [https://bugs.python.org/issue?@action=redirect&bpo=27897]: Fixed possible crash in sqlite3.Connection.create_collation() if pass invalid string-like object as a name. Patch by Xiang Zhang.

- bpo-18893 [https://bugs.python.org/issue?@action=redirect&bpo=18893]: Fix invalid exception handling in Lib/ctypes/macholib/dyld.py. Patch by Madison May.
- bpo-27611 [https://bugs.python.org/issue?@action=redirect&bpo=27611]: Fixed support of default root window in the tkinter.tix module.
- bpo-27348 [https://bugs.python.org/issue?@action=redirect&bpo=27348]: In the traceback module, restore the formatting of exception messages like "Exception: None". This fixes a regression introduced in 3.5a2.
- bpo-25651 [https://bugs.python.org/issue?@action=redirect&bpo=25651]: Allow false values to be used for msg parameter of subTest().
- bpo-27932 [https://bugs.python.org/issue?@action=redirect&bpo=27932]: Prevent memory leak in win32_ver().
- Fix UnboundLocalError in socket._sendfile_use_sendfile.
- bpo-28075 [https://bugs.python.org/issue?@action=redirect&bpo=28075]: Check for ERROR_ACCESS_DENIED in Windows implementation of os.stat(). Patch by Eryk Sun.
- bpo-25270 [https://bugs.python.org/issue?@action=redirect&bpo=25270]: Prevent codecs.escape_encode() from raising SystemError when an empty bytestring is passed.
- bpo-28181 [https://bugs.python.org/issue?@action=redirect&bpo=28181]: Get antigravity over HTTPS. Patch by Kaartic Sivaraam.
- bpo-25895 [https://bugs.python.org/issue?@action=redirect&bpo=25895]: Enable WebSocket URL schemes in urllib.parse.urljoin. Patch by Gergely Imreh and Markus Holtermann.
- bpo-27599 [https://bugs.python.org/issue?@action=redirect&bpo=27599]: Fixed buffer overrun in binascii.b2a_qp() and binascii.a2b_qp().

- bpo-19003 [https://bugs.python.org/issue?@action=redirect&bpo=19003]: m email.generator now replaces only `\r` and/or `\n` line endings, per the RFC, instead of all unicode line endings.
- bpo-28019 [https://bugs.python.org/issue?@action=redirect&bpo=28019]: itertools.count() no longer rounds non-integer step in range between 1.0 and 2.0 to 1.
- bpo-25969 [https://bugs.python.org/issue?@action=redirect&bpo=25969]: Update the lib2to3 grammar to handle the unpacking generalizations added in 3.5.
- bpo-14977 [https://bugs.python.org/issue?@action=redirect&bpo=14977]: mailcap now respects the order of the lines in the mailcap files (“first match”), as required by RFC 1542. Patch by Michael Lazar.
- bpo-24594 [https://bugs.python.org/issue?@action=redirect&bpo=24594]: Validates persist parameter when opening MSI database
- bpo-17582 [https://bugs.python.org/issue?@action=redirect&bpo=17582]: xml.etree.ElementTree nows preserves whitespaces in attributes (Patch by Duane Griffin. Reviewed and approved by Stefan Behnel.)
- bpo-28047 [https://bugs.python.org/issue?@action=redirect&bpo=28047]: Fixed calculation of line length used for the base64 CTE in the new email policies.
- bpo-27445 [https://bugs.python.org/issue?@action=redirect&bpo=27445]: Don’t pass str(_charset) to MIMEText.set_payload(). Patch by Claude Paroz.
- bpo-22450 [https://bugs.python.org/issue?@action=redirect&bpo=22450]: urllib now includes an `Accept: */*` header among the default headers. This makes the results of REST API requests more consistent and predictable especially when proxy servers are involved.
- lib2to3.pgen3.driver.load_grammar() now creates a stable cache file between runs given the same

Grammar.txt input regardless of the hash randomization setting.

- bpo-27570 [https://bugs.python.org/issue?@action=redirect&bpo=27570]: Avoid zero-length memcpy() etc calls with null source pointers in the “ctypes” and “array” modules.
- bpo-22233 [https://bugs.python.org/issue?@action=redirect&bpo=22233]: Break email header lines *only* on the RFC specified CR and LF characters, not on arbitrary unicode line breaks. This also fixes a bug in HTTP header parsing.
- bpo-27988 [https://bugs.python.org/issue?@action=redirect&bpo=27988]: Fix email iter_attachments incorrect mutation of payload list.
- bpo-27691 [https://bugs.python.org/issue?@action=redirect&bpo=27691]: Fix ssl module’s parsing of GEN_RID subject alternative name fields in X.509 certs.
- bpo-27850 [https://bugs.python.org/issue?@action=redirect&bpo=27850]: Remove 3DES from ssl module’s default cipher list to counter measure sweet32 attack (**CVE 2016-2183** [https://www.cve.org/CVERecord?id=CVE-2016-2183]).
- bpo-27766 [https://bugs.python.org/issue?@action=redirect&bpo=27766]: Add ChaCha20 Poly1305 to ssl module’s default cipher list. (Required OpenSSL 1.1.0 or LibreSSL).
- bpo-26470 [https://bugs.python.org/issue?@action=redirect&bpo=26470]: Port ssl and hashlib module to OpenSSL 1.1.0.
- Remove support for passing a file descriptor to os.access. It never worked but previously didn’t raise.
- bpo-12885 [https://bugs.python.org/issue?@action=redirect&bpo=12885]: Fix error when distutils encounters symlink.
- bpo-27881 [https://bugs.python.org/issue?@action=redirect&bpo=27881]: Fixed possible bugs when

setting sqlite3.Connection.isolation_level. Based on patch by Xiang Zhang.

- bpo-27861 [https://bugs.python.org/issue?@action=redirect&bpo=27861]: Fixed a crash in sqlite3.Connection.cursor() when a factory creates not a cursor. Patch by Xiang Zhang.
- bpo-19884 [https://bugs.python.org/issue?@action=redirect&bpo=19884]: Avoid spurious output on OS X with Gnu Readline.
- bpo-27706 [https://bugs.python.org/issue?@action=redirect&bpo=27706]: Restore deterministic behavior of random.Random().seed() for string seeds using seeding version 1. Allows sequences of calls to random() to exactly match those obtained in Python 2. Patch by Nofar Schnider.
- bpo-10513 [https://bugs.python.org/issue?@action=redirect&bpo=10513]: Fix a regression in Connection.commit(). Statements should not be reset after a commit.
- A new version of typing.py from https://github.com/python/typing: Collection (only for 3.6) (bpo-27598 [https://bugs.python.org/issue?@action=redirect&bpo=27598]). Add FrozenSet to __all__ (upstream #261). Fix crash in _get_type_vars() (upstream #259). Remove the dict constraint in ForwardRef._eval_type (upstream #252).
- bpo-27539 [https://bugs.python.org/issue?@action=redirect&bpo=27539]: Fix unnormalised `Fraction.__pow__` result in the case of negative exponent and negative base.
- bpo-21718 [https://bugs.python.org/issue?@action=redirect&bpo=21718]: cursor.description is now available for queries using CTEs.
- bpo-2466 [https://bugs.python.org/issue?@action=redirect&bpo=2466]: posixpath.ismount now correctly recognizes mount

points which the user does not have permission to access.

- bpo-27773 [https://bugs.python.org/issue?@action=redirect&bpo=27773]: Correct some memory management errors server_hostname in _ssl.wrap_socket().
- bpo-26750 [https://bugs.python.org/issue?@action=redirect&bpo=26750]: unittest.mock.create_autospec() now works properly for subclasses of property() and other data descriptors.
- In the curses module, raise an error if window.getstr() or window.instr() is passed a negative value.
- bpo-27783 [https://bugs.python.org/issue?@action=redirect&bpo=27783]: Fix possible usage of uninitialized memory in operator.methodcaller.
- bpo-27774 [https://bugs.python.org/issue?@action=redirect&bpo=27774]: Fix possible Py_DECREF on unowned object in _sre.
- bpo-27760 [https://bugs.python.org/issue?@action=redirect&bpo=27760]: Fix possible integer overflow in binascii.b2a_qp.
- bpo-27758 [https://bugs.python.org/issue?@action=redirect&bpo=27758]: Fix possible integer overflow in the _csv module for large record lengths.
- bpo-27568 [https://bugs.python.org/issue?@action=redirect&bpo=27568]: Prevent HTTPoxy attack (**CVE 2016-1000110** [https://www.cve.org/CVERecord?id=CVE-2016-1000110]). Ignore the HTTP_PROXY variable when REQUEST_METHOD environment is set, which indicates that the script is in CGI mode.
- bpo-27656 [https://bugs.python.org/issue?@action=redirect&bpo=27656]: Do not assume sched.h defines any SCHED_* constants.
- bpo-27130 [https://bugs.python.org/issue?@action=redirect&bpo=27130]: In the “zlib” module, fix handling of large buffers (typically 4 GiB) when

compressing and decompressing. Previously, inputs were limited to 4 GiB, and compression and decompression operations did not properly handle results of 4 GiB.

- bpo-27533 [https://bugs.python.org/issue?@action=redirect&bpo=27533]: Release GIL in nt._isdir
- bpo-17711 [https://bugs.python.org/issue?@action=redirect&bpo=17711]: Fixed unpickling by the persistent ID with protocol 0. Original patch by Alexandre Vassalotti.
- bpo-27522 [https://bugs.python.org/issue?@action=redirect&bpo=27522]: Avoid an unintentional reference cycle in email.feedparser.
- bpo-26844 [https://bugs.python.org/issue?@action=redirect&bpo=26844]: Fix error message for imp.find_module() to refer to ‘path’ instead of ‘name’. Patch by Lev Maximov.
- bpo-23804 [https://bugs.python.org/issue?@action=redirect&bpo=23804]: Fix SSL zero-length recv() calls to not block and not raise an error about unclean EOF.
- bpo-27466 [https://bugs.python.org/issue?@action=redirect&bpo=27466]: Change time format returned by http.cookie.time2netscape, confirming the netscape cookie format and making it consistent with documentation.
- bpo-26664 [https://bugs.python.org/issue?@action=redirect&bpo=26664]: Fix activate.fish by removing mis-use of `$`.
- bpo-22115 [https://bugs.python.org/issue?@action=redirect&bpo=22115]: Fixed tracing Tkinter variables: trace_vdelete() with wrong mode no longer break tracing, trace_vinfo() now always returns a list of pairs of strings, tracing in the “u” mode now works.
- Fix a scoping issue in importlib.util.LazyLoader which triggered an UnboundLocalError when lazy-loading a module that was already put into sys.modules.

- bpo-27079 [https://bugs.python.org/issue?@action=redirect&bpo=27079]: Fixed curses.ascii functions isblank(), iscntrl() and ispunct().
- bpo-26754 [https://bugs.python.org/issue?@action=redirect&bpo=26754]: Some functions (compile() etc) accepted a filename argument encoded as an iterable of integers. Now only strings and byte-like objects are accepted.
- bpo-27048 [https://bugs.python.org/issue?@action=redirect&bpo=27048]: Prevents distutils failing on Windows when environment variables contain non-ASCII characters
- bpo-27330 [https://bugs.python.org/issue?@action=redirect&bpo=27330]: Fixed possible leaks in the ctypes module.
- bpo-27238 [https://bugs.python.org/issue?@action=redirect&bpo=27238]: Got rid of bare excepts in the turtle module. Original patch by Jelle Zijlstra.
- bpo-27122 [https://bugs.python.org/issue?@action=redirect&bpo=27122]: When an exception is raised within the context being managed by a contextlib.ExitStack() and one of the exit stack generators catches and raises it in a chain, do not re-raise the original exception when exiting, let the new chained one through. This avoids the **PEP 479** [https://peps.python.org/pep-0479/] bug described in issue25782.
- bpo-26386 [https://bugs.python.org/issue?@action=redirect&bpo=26386]: Fixed ttk.TreeView selection operations with item id’s containing spaces.
- bpo-16182 [https://bugs.python.org/issue?@action=redirect&bpo=16182]: Fix various functions in the “readline” module to use the locale encoding, and fix get_begidx() and get_endidx() to return code point indexes.

- bpo-27392 [https://bugs.python.org/issue?@action=redirect&bpo=27392]: Add loop.connect_accepted_socket(). Patch by Jim Fulton.
- bpo-27930 [https://bugs.python.org/issue?@action=redirect&bpo=27930]: Improved behaviour of logging.handlers.QueueListener. Thanks to Paulo Andrade and Petr Viktorin for the analysis and patch.
- bpo-21201 [https://bugs.python.org/issue?@action=redirect&bpo=21201]: Improves readability of multiprocessing error message. Thanks to Wojciech Walczak for patch.
- bpo-27456 [https://bugs.python.org/issue?@action=redirect&bpo=27456]: asyncio: Set TCP_NODELAY by default.
- bpo-27906 [https://bugs.python.org/issue?@action=redirect&bpo=27906]: Fix socket accept exhaustion during high TCP traffic. Patch by Kevin Conway.
- bpo-28174 [https://bugs.python.org/issue?@action=redirect&bpo=28174]: Handle when SO_REUSEPORT isn’t properly supported. Patch by Seth Michael Larson.
- bpo-26654 [https://bugs.python.org/issue?@action=redirect&bpo=26654]: Inspect functools.partial in asyncio.Handle.__repr__. Patch by iceboy.
- bpo-26909 [https://bugs.python.org/issue?@action=redirect&bpo=26909]: Fix slow pipes IO in asyncio. Patch by INADA Naoki.
- bpo-28176 [https://bugs.python.org/issue?@action=redirect&bpo=28176]: Fix callbacks race in asyncio.SelectorLoop.sock_connect.
- bpo-27759 [https://bugs.python.org/issue?@action=redirect&bpo=27759]: Fix selectors incorrectly retain invalid file descriptors. Patch by Mark Williams.
- bpo-28368 [https://bugs.python.org/issue?@action=redirect&bpo=28368]: Refuse monitoring processes if the child watcher has no loop attached. Patch by Vincent Michel.

- bpo-28369 [https://bugs.python.org/issue?@action=redirect&bpo=28369]: Raise RuntimeError when transport's FD is used with add_reader, add_writer, etc.
- bpo-28370 [https://bugs.python.org/issue?@action=redirect&bpo=28370]: Speedup asyncio.StreamReader.readexactly. Patch by Коренберг Марк.
- bpo-28371 [https://bugs.python.org/issue?@action=redirect&bpo=28371]: Deprecate passing asyncio.Handles to run_in_executor.
- bpo-28372 [https://bugs.python.org/issue?@action=redirect&bpo=28372]: Fix asyncio to support formatting of non-python coroutines.
- bpo-28399 [https://bugs.python.org/issue?@action=redirect&bpo=28399]: Remove UNIX socket from FS before binding. Patch by Коренберг Марк.
- bpo-27972 [https://bugs.python.org/issue?@action=redirect&bpo=27972]: Prohibit Tasks to await on themselves.
- bpo-26923 [https://bugs.python.org/issue?@action=redirect&bpo=26923]: Fix asyncio.Gather to refuse being cancelled once all children are done. Patch by Johannes Ebke.
- bpo-26796 [https://bugs.python.org/issue?@action=redirect&bpo=26796]: Don't configure the number of workers for default threadpool executor. Initial patch by Hans Lawrenz.
- bpo-28600 [https://bugs.python.org/issue?@action=redirect&bpo=28600]: Optimize loop.call_soon().
- bpo-28613 [https://bugs.python.org/issue?@action=redirect&bpo=28613]: Fix get_event_loop() return the current loop if called from coroutines/callbacks.
- bpo-28639 [https://bugs.python.org/issue?@action=redirect&bpo=28639]: Fix inspect.isawaitable to always return bool Patch by Justin Mayfield.

- bpo-28652 [https://bugs.python.org/issue?@action=redirect&bpo=28652]: Make loop methods reject socket kinds they do not support.
- bpo-28653 [https://bugs.python.org/issue?@action=redirect&bpo=28653]: Fix a refleak in functools.lru_cache.
- bpo-28703 [https://bugs.python.org/issue?@action=redirect&bpo=28703]: Fix asyncio.iscoroutinefunction to handle Mock objects.
- bpo-24142 [https://bugs.python.org/issue?@action=redirect&bpo=24142]: Reading a corrupt config file left the parser in an invalid state. Original patch by Florian Höch.
- bpo-28990 [https://bugs.python.org/issue?@action=redirect&bpo=28990]: Fix SSL hanging if connection is closed before handshake completed. (Patch by HoHo-Ho)

IDLE

- bpo-15308 [https://bugs.python.org/issue?@action=redirect&bpo=15308]: Add ‘interrupt execution’ (^C) to Shell menu. Patch by Roger Serwy, updated by Bayard Randel.
- bpo-27922 [https://bugs.python.org/issue?@action=redirect&bpo=27922]: Stop IDLE tests from ‘flashing’ gui widgets on the screen.
- Add version to title of IDLE help window.
- bpo-25564 [https://bugs.python.org/issue?@action=redirect&bpo=25564]: In section on IDLE – console differences, mention that using exec means that __builtins__ is defined for each statement.
- bpo-27714 [https://bugs.python.org/issue?@action=redirect&bpo=27714]: text_textview and test_autocomplete now pass when re-run in the same process. This occurs when test_idle fails when run with

the -w option but without -jn. Fix warning from test_config.

- bpo-25507 [https://bugs.python.org/issue?@action=redirect&bpo=25507]: IDLE no longer runs buggy code because of its tkinter imports. Users must include the same imports required to run directly in Python.
- bpo-27452 [https://bugs.python.org/issue?@action=redirect&bpo=27452]: add line counter and crc to IDLE configHandler test dump.
- bpo-27365 [https://bugs.python.org/issue?@action=redirect&bpo=27365]: Allow non-ascii chars in IDLE NEWS.txt, for contributor names.
- bpo-27245 [https://bugs.python.org/issue?@action=redirect&bpo=27245]: IDLE: Cleanly delete custom themes and key bindings. Previously, when IDLE was started from a console or by import, a cascade of warnings was emitted. Patch by Serhiy Storchaka.

C API

- bpo-28808 [https://bugs.python.org/issue?@action=redirect&bpo=28808]: PyUnicode_CompareWithASCIIString() now never raises exceptions.
- bpo-26754 [https://bugs.python.org/issue?@action=redirect&bpo=26754]: PyUnicode_FSDecoder() accepted a filename argument encoded as an iterable of integers. Now only strings and bytes-like objects are accepted.

Documentation

- bpo-28513 [https://bugs.python.org/issue?@action=redirect&bpo=28513]: Documented command-line interface of zipfile.

Tests

- bpo-28950 [https://bugs.python.org/issue?@action=redirect&bpo=28950]: Disallow -j0 to be combined with -T/-l/-M in regrtest command line arguments.
- bpo-28666 [https://bugs.python.org/issue?@action=redirect&bpo=28666]: Now test.support.rmtree is able to remove unwritable or unreadable directories.
- bpo-23839 [https://bugs.python.org/issue?@action=redirect&bpo=23839]: Various caches now are cleared before running every test file.
- bpo-28409 [https://bugs.python.org/issue?@action=redirect&bpo=28409]: regrtest: fix the parser of command line arguments.
- bpo-27787 [https://bugs.python.org/issue?@action=redirect&bpo=27787]: Call gc.collect() before checking each test for “dangling threads”, since the dangling threads are weak references.
- bpo-27369 [https://bugs.python.org/issue?@action=redirect&bpo=27369]: In test_pyexpat, avoid testing an error message detail that changed in Expat 2.2.0.

Tools/Demos

- bpo-27952 [https://bugs.python.org/issue?@action=redirect&bpo=27952]: Get Tools/scripts/fixcid.py working with Python 3 and the current “re” module, avoid invalid Python backslash escapes, and fix a bug parsing escaped C quote signs.
- bpo-27332 [https://bugs.python.org/issue?@action=redirect&bpo=27332]: Fixed the type of the first argument of module-level functions generated by Argument Clinic. Patch by Petr Viktorin.
- bpo-27418 [https://bugs.python.org/issue?@action=redirect&bpo=27418]: Fixed

Tools/importbench/importbench.py.

Windows

- bpo-28251 [https://bugs.python.org/issue?@action=redirect&bpo=28251]: Improvements to help manuals on Windows.
- bpo-28110 [https://bugs.python.org/issue?@action=redirect&bpo=28110]: launcher.msi has different product codes between 32-bit and 64-bit
- bpo-25144 [https://bugs.python.org/issue?@action=redirect&bpo=25144]: Ensures TargetDir is set before continuing with custom install.
- bpo-27469 [https://bugs.python.org/issue?@action=redirect&bpo=27469]: Adds a shell extension to the launcher so that drag and drop works correctly.
- bpo-27309 [https://bugs.python.org/issue?@action=redirect&bpo=27309]: Enabled proper Windows styles in python[w].exe manifest.

Build

- bpo-29080 [https://bugs.python.org/issue?@action=redirect&bpo=29080]: Removes hard dependency on hg.exe from PCBuild/build.bat
- bpo-23903 [https://bugs.python.org/issue?@action=redirect&bpo=23903]: Added missed names to PC/python3.def.
- bpo-10656 [https://bugs.python.org/issue?@action=redirect&bpo=10656]: Fix out-of-tree building on AIX. Patch by Tristan Carel and Michael Haubenwallner.
- bpo-26359 [https://bugs.python.org/issue?@action=redirect&bpo=26359]: Rename –with-optimiations to –enable-optimizations.
- bpo-28444 [https://bugs.python.org/issue?@action=redirect&bpo=28444]: Fix missing extensions modules

when cross compiling.

- bpo-28248 [https://bugs.python.org/issue?@action=redirect&bpo=28248]: Update Windows build and OS X installers to use OpenSSL 1.0.2j.
- bpo-28258 [https://bugs.python.org/issue?@action=redirect&bpo=28258]: Fixed build with Estonian locale (python-config and distclean targets in Makefile). Patch by Arfrever Frehtes Taifersar Arahesis.
- bpo-26661 [https://bugs.python.org/issue?@action=redirect&bpo=26661]: setup.py now detects system libffi with multiarch wrapper.
- bpo-28066 [https://bugs.python.org/issue?@action=redirect&bpo=28066]: Fix the logic that searches build directories for generated include files when building outside the source tree.
- bpo-15819 [https://bugs.python.org/issue?@action=redirect&bpo=15819]: Remove redundant include search directory option for building outside the source tree.
- bpo-27566 [https://bugs.python.org/issue?@action=redirect&bpo=27566]: Fix clean target in freeze makefile (patch by Lisa Roach)
- bpo-27705 [https://bugs.python.org/issue?@action=redirect&bpo=27705]: Update message in validate_ucrtbase.py
- bpo-27983 [https://bugs.python.org/issue?@action=redirect&bpo=27983]: Cause lack of llvm-profdata tool when using clang as required for PGO linking to be a configure time error rather than make time when –with-optimizations is enabled. Also improve our ability to find the llvm-profdata tool on MacOS and some Linuxes.
- bpo-26307 [https://bugs.python.org/issue?@action=redirect&bpo=26307]: The profile-opt build now applies PGO to the built-in modules.
- bpo-26359 [https://bugs.python.org/issue?@action=redirect&bpo=26359]: Add the –with-optimizations

configure flag.

- bpo-27713 [https://bugs.python.org/issue?@action=redirect&bpo=27713]: Suppress spurious build warnings when updating importlib’s bootstrap files. Patch by Xiang Zhang
- bpo-25825 [https://bugs.python.org/issue?@action=redirect&bpo=25825]: Correct the references to Modules/python.exp and ld_so_aix, which are required on AIX. This updates references to an installation path that was changed in 3.2a4, and undoes changed references to the build tree that were made in 3.5.0a1.
- bpo-27453 [https://bugs.python.org/issue?@action=redirect&bpo=27453]: CPP invocation in configure must use CPPFLAGS. Patch by Chi Hsuan Yen.
- bpo-27641 [https://bugs.python.org/issue?@action=redirect&bpo=27641]: The configure script now inserts comments into the makefile to prevent the pgen and _freeze_importlib executables from being cross-compiled.
- bpo-26662 [https://bugs.python.org/issue?@action=redirect&bpo=26662]: Set PYTHON_FOR_GEN in configure as the Python program to be used for file generation during the build.
- bpo-10910 [https://bugs.python.org/issue?@action=redirect&bpo=10910]: Avoid C++ compilation errors on FreeBSD and OS X. Also update FreedBSD version checks for the original ctype UTF-8 workaround.
- bpo-28676 [https://bugs.python.org/issue?@action=redirect&bpo=28676]: Prevent missing ‘getentropy’ declaration warning on macOS. Patch by Gareth Rees.

Python 3.5.2 final

Release date: 2016-06-26

Core and Builtins

- bpo-26930 [https://bugs.python.org/issue?@action=redirect&bpo=26930]: Update Windows builds to use OpenSSL 1.0.2h.

Tests

- bpo-26867 [https://bugs.python.org/issue?@action=redirect&bpo=26867]: Ubuntu’s openssl OP_NO_SSLv3 is forced on by default; fix test.

IDLE

- bpo-27365 [https://bugs.python.org/issue?@action=redirect&bpo=27365]: Allow non-ascii in idlelib/NEWS.txt - minimal part for 3.5.2.

Python 3.5.2 release candidate 1

Release date: 2016-06-12

Security

- bpo-26556 [https://bugs.python.org/issue?@action=redirect&bpo=26556]: Update expat to 2.1.1, fixes **CVE 2015-1283** [https://www.cve.org/CVERecord?id=CVE-2015-1283].
- Fix TLS stripping vulnerability in smtplib, **CVE 2016-0772** [https://www.cve.org/CVERecord?id=CVE-2016-0772]. Reported by Team Oststrom.
- bpo-26839 [https://bugs.python.org/issue?@action=redirect&bpo=26839]: On Linux, `os.urandom()` now calls `getrandom()` with `GRND_NONBLOCK` to fall back on reading

`/dev/urandom` if the urandom entropy pool is not initialized yet. Patch written by Colm Buckley.

- bpo-26657 [https://bugs.python.org/issue?@action=redirect&bpo=26657]: Fix directory traversal vulnerability with http.server on Windows. This fixes a regression that was introduced in 3.3.4rc1 and 3.4.0rc1. Based on patch by Philipp Hagemeister.
- bpo-26313 [https://bugs.python.org/issue?@action=redirect&bpo=26313]: ssl.py _load_windows_store_certs fails if windows cert store is empty. Patch by Baji.
- bpo-25939 [https://bugs.python.org/issue?@action=redirect&bpo=25939]: On Windows open the cert store readonly in ssl.enum_certificates.

Core and Builtins

- bpo-27066 [https://bugs.python.org/issue?@action=redirect&bpo=27066]: Fixed SystemError if a custom opener (for open()) returns a negative number without setting an exception.
- bpo-20041 [https://bugs.python.org/issue?@action=redirect&bpo=20041]: Fixed TypeError when frame.f_trace is set to None. Patch by Xavier de Gaye.
- bpo-26168 [https://bugs.python.org/issue?@action=redirect&bpo=26168]: Fixed possible refleaks in failing Py_BuildValue() with the “N” format unit.
- bpo-26991 [https://bugs.python.org/issue?@action=redirect&bpo=26991]: Fix possible refleak when creating a function with annotations.
- bpo-27039 [https://bugs.python.org/issue?@action=redirect&bpo=27039]: Fixed bytearray.remove() for values greater than 127. Patch by Joe Jevnik.
- bpo-23640 [https://bugs.python.org/issue?@action=redirect&bpo=23640]: int.from_bytes() no longer bypasses constructors for subclasses.

- bpo-26811 [https://bugs.python.org/issue?@action=redirect&bpo=26811]: gc.get_objects() no longer contains a broken tuple with NULL pointer.
- bpo-20120 [https://bugs.python.org/issue?@action=redirect&bpo=20120]: Use RawConfigParser for .pypirc parsing, removing support for interpolation unintentionally added with move to Python 3. Behavior no longer does any interpolation in .pypirc files, matching behavior in Python 2.7 and Setuptools 19.0.
- bpo-26659 [https://bugs.python.org/issue?@action=redirect&bpo=26659]: Make the builtin slice type support cycle collection.
- bpo-26718 [https://bugs.python.org/issue?@action=redirect&bpo=26718]: super.__init__ no longer leaks memory if called multiple times. NOTE: A direct call of super.__init__ is not endorsed!
- bpo-25339 [https://bugs.python.org/issue?@action=redirect&bpo=25339]: PYTHONIOENCODING now has priority over locale in setting the error handler for stdin and stdout.
- bpo-26494 [https://bugs.python.org/issue?@action=redirect&bpo=26494]: Fixed crash on iterating exhausting iterators. Affected classes are generic sequence iterators, iterators of str, bytes, bytearray, list, tuple, set, frozenset, dict, OrderedDict, corresponding views and os.scandir() iterator.
- bpo-26581 [https://bugs.python.org/issue?@action=redirect&bpo=26581]: If coding cookie is specified multiple times on a line in Python source code file, only the first one is taken to account.
- bpo-26464 [https://bugs.python.org/issue?@action=redirect&bpo=26464]: Fix str.translate() when string is ASCII and first replacements removes character, but next replacement uses a non-ASCII character or a string longer than 1 character. Regression introduced in Python 3.5.0.

- bpo-22836 [https://bugs.python.org/issue?@action=redirect&bpo=22836]: Ensure exception reports from PyErr_Display() and PyErr_WriteUnraisable() are sensible even when formatting them produces secondary errors. This affects the reports produced by sys.__excepthook__() and when __del__() raises an exception.
- bpo-26302 [https://bugs.python.org/issue?@action=redirect&bpo=26302]: Correct behavior to reject comma as a legal character for cookie names.
- bpo-4806 [https://bugs.python.org/issue?@action=redirect&bpo=4806]: Avoid masking the original TypeError exception when using star (*) unpacking in function calls. Based on patch by Hagen Fürstenau and Daniel Urban.
- bpo-27138 [https://bugs.python.org/issue?@action=redirect&bpo=27138]: Fix the doc comment for FileFinder.find_spec().
- bpo-26154 [https://bugs.python.org/issue?@action=redirect&bpo=26154]: Add a new private _PyThreadState_UncheckedGet() function to get the current Python thread state, but don't issue a fatal error if it is NULL. This new function must be used instead of accessing directly the _PyThreadState_Current variable. The variable is no more exposed since Python 3.5.1 to hide the exact implementation of atomic C types, to avoid compiler issues.
- bpo-26194 [https://bugs.python.org/issue?@action=redirect&bpo=26194]: Deque.insert() gave odd results for bounded deques that had reached their maximum size. Now an IndexError will be raised when attempting to insert into a full deque.
- bpo-25843 [https://bugs.python.org/issue?@action=redirect&bpo=25843]: When compiling code, don't merge constants if they are equal but have a different types. For example, `f1, f2 = lambda: 1, lambda: 1.0` is now correctly compiled to two different functions: `f1()`

returns `1` (`int`) and `f2()` returns `1.0` (`int`), even if `1` and `1.0` are equal.

- bpo-22995 [https://bugs.python.org/issue?@action=redirect&bpo=22995]: [UPDATE] Comment out the one of the pickleability tests in _PyObject_GetState() due to regressions observed in Cython-based projects.
- bpo-25961 [https://bugs.python.org/issue?@action=redirect&bpo=25961]: Disallowed null characters in the type name.
- bpo-25973 [https://bugs.python.org/issue?@action=redirect&bpo=25973]: Fix segfault when an invalid nonlocal statement binds a name starting with two underscores.
- bpo-22995 [https://bugs.python.org/issue?@action=redirect&bpo=22995]: Instances of extension types with a state that aren’t subclasses of list or dict and haven’t implemented any pickle-related methods (__reduce__, __reduce_ex__, __getnewargs__, __getnewargs_ex__, or __getstate__), can no longer be pickled. Including memoryview.
- bpo-20440 [https://bugs.python.org/issue?@action=redirect&bpo=20440]: Massive replacing unsafe attribute setting code with special macro Py_SETREF.
- bpo-25766 [https://bugs.python.org/issue?@action=redirect&bpo=25766]: Special method __bytes__() now works in str subclasses.
- bpo-25421 [https://bugs.python.org/issue?@action=redirect&bpo=25421]: __sizeof__ methods of builtin types now use dynamic basic size. This allows sys.getsize() to work correctly with their subclasses with __slots__ defined.
- bpo-25709 [https://bugs.python.org/issue?@action=redirect&bpo=25709]: Fixed problem with in-place string concatenation and utf-8 cache.
- bpo-27147 [https://bugs.python.org/issue?@action=redirect&bpo=27147]: Mention **PEP 420**

[https://peps.python.org/pep-0420/] in the importlib docs.

- bpo-24097 [https://bugs.python.org/issue?@action=redirect&bpo=24097]: Fixed crash in object.__reduce__() if slot name is freed inside __getattr__.
- bpo-24731 [https://bugs.python.org/issue?@action=redirect&bpo=24731]: Fixed crash on converting objects with special methods __bytes__, __trunc__, and __float__ returning instances of subclasses of bytes, int, and float to subclasses of bytes, int, and float correspondingly.
- bpo-26478 [https://bugs.python.org/issue?@action=redirect&bpo=26478]: Fix semantic bugs when using binary operators with dictionary views and tuples.
- bpo-26171 [https://bugs.python.org/issue?@action=redirect&bpo=26171]: Fix possible integer overflow and heap corruption in zipimporter.get_data().
- bpo-25660 [https://bugs.python.org/issue?@action=redirect&bpo=25660]: Fix TAB key behaviour in REPL with readline.
- bpo-25887 [https://bugs.python.org/issue?@action=redirect&bpo=25887]: Raise a RuntimeError when a coroutine object is awaited more than once.
- bpo-27243 [https://bugs.python.org/issue?@action=redirect&bpo=27243]: Update the __aiter__ protocol: instead of returning an awaitable that resolves to an asynchronous iterator, the asynchronous iterator should be returned directly. Doing the former will trigger a PendingDeprecationWarning.

Library

- bpo-21386 [https://bugs.python.org/issue?@action=redirect&bpo=21386]: Implement missing IPv4Address.is_global property. It was documented since 07a5610bae9d. Initial patch by Roger Luethi.

- bpo-20900 [https://bugs.python.org/issue?@action=redirect&bpo=20900]: distutils register command now decodes HTTP responses correctly. Initial patch by ingrid.
- A new version of typing.py provides several new classes and features: @overload outside stubs, Reversible, DefaultDict, Text, ContextManager, Type[], NewType(), TYPE_CHECKING, and numerous bug fixes (note that some of the new features are not yet implemented in mypy or other static analyzers). Also classes for **PEP 492** [https://peps.python.org/pep-0492/] (Awaitable, AsyncIterable, AsyncIterator) have been added (in fact they made it into 3.5.1 but were never mentioned).
- bpo-25738 [https://bugs.python.org/issue?@action=redirect&bpo=25738]: Stop http.server.BaseHTTPRequestHandler.send_error() from sending a message body for 205 Reset Content. Also, don’t send Content header fields in responses that don’t have a body. Patch by Susumu Koshiba.
- bpo-21313 [https://bugs.python.org/issue?@action=redirect&bpo=21313]: Fix the “platform” module to tolerate when sys.version contains truncated build information.
- bpo-27164 [https://bugs.python.org/issue?@action=redirect&bpo=27164]: In the zlib module, allow decompressing raw Deflate streams with a predefined zdict. Based on patch by Xiang Zhang.
- bpo-24291 [https://bugs.python.org/issue?@action=redirect&bpo=24291]: Fix wsgiref.simple_server.WSGIRequestHandler to completely write data to the client. Previously it could do partial writes and truncate data. Also, wsgiref.handler.ServerHandler can now handle stdout doing partial writes, but this is deprecated.
- bpo-26809 [https://bugs.python.org/issue?@action=redirect&bpo=26809]: Add `__all__` to **`string`**. Patch by

Emanuel Barry.
- bpo-26373 [https://bugs.python.org/issue?@action=redirect&bpo=26373]: subprocess.Popen.communicate now correctly ignores BrokenPipeError when the child process dies before .communicate() is called in more/all circumstances.
- bpo-21776 [https://bugs.python.org/issue?@action=redirect&bpo=21776]: distutils.upload now correctly handles HTTPError. Initial patch by Claudiu Popa.
- bpo-27114 [https://bugs.python.org/issue?@action=redirect&bpo=27114]: Fix SSLContext._load_windows_store_certs fails with PermissionError
- bpo-18383 [https://bugs.python.org/issue?@action=redirect&bpo=18383]: Avoid creating duplicate filters when using filterwarnings and simplefilter. Based on patch by Alex Shkop.
- bpo-27057 [https://bugs.python.org/issue?@action=redirect&bpo=27057]: Fix os.set_inheritable() on Android, ioctl() is blocked by SELinux and fails with EACCESS. The function now falls back to fcntl(). Patch written by Michał Bednarski.
- bpo-27014 [https://bugs.python.org/issue?@action=redirect&bpo=27014]: Fix infinite recursion using typing.py. Thanks to Kalle Tuure!
- bpo-14132 [https://bugs.python.org/issue?@action=redirect&bpo=14132]: Fix urllib.request redirect handling when the target only has a query string. Original fix by Ján Janech.
- bpo-17214 [https://bugs.python.org/issue?@action=redirect&bpo=17214]: The “urllib.request” module now percent-encodes non-ASCII bytes found in redirect target URLs. Some servers send Location header fields with non-ASCII bytes, but “http.client” requires the request target to be ASCII-encodable, otherwise a

UnicodeEncodeError is raised. Based on patch by Christian Heimes.

- bpo-26892 [https://bugs.python.org/issue?@action=redirect&bpo=26892]: Honor debuglevel flag in urllib.request.HTTPHandler. Patch contributed by Chi Hsuan Yen.
- bpo-22274 [https://bugs.python.org/issue?@action=redirect&bpo=22274]: In the subprocess module, allow stderr to be redirected to stdout even when stdout is not redirected. Patch by Akira Li.
- bpo-26807 [https://bugs.python.org/issue?@action=redirect&bpo=26807]: mock_open ‘files’ no longer error on readline at end of file. Patch from Yolanda Robla.
- bpo-25745 [https://bugs.python.org/issue?@action=redirect&bpo=25745]: Fixed leaking a userptr in curses panel destructor.
- bpo-26977 [https://bugs.python.org/issue?@action=redirect&bpo=26977]: Removed unnecessary, and ignored, call to sum of squares helper in statistics.pvariance.
- bpo-26881 [https://bugs.python.org/issue?@action=redirect&bpo=26881]: The modulefinder module now supports extended opcode arguments.
- bpo-23815 [https://bugs.python.org/issue?@action=redirect&bpo=23815]: Fixed crashes related to directly created instances of types in _tkinter and curses.panel modules.
- bpo-17765 [https://bugs.python.org/issue?@action=redirect&bpo=17765]: weakref.ref() no longer silently ignores keyword arguments. Patch by Georg Brandl.
- bpo-26873 [https://bugs.python.org/issue?@action=redirect&bpo=26873]: xmlrpc now raises ResponseError on unsupported type tags instead of silently return incorrect result.

- bpo-26711 [https://bugs.python.org/issue?@action=redirect&bpo=26711]: Fixed the comparison of plistlib.Data with other types.
- bpo-24114 [https://bugs.python.org/issue?@action=redirect&bpo=24114]: Fix an uninitialized variable in `ctypes.util`. The bug only occurs on SunOS when the ctypes implementation searches for the `crle` program. Patch by Xiang Zhang. Tested on SunOS by Kees Bos.
- bpo-26864 [https://bugs.python.org/issue?@action=redirect&bpo=26864]: In urllib.request, change the proxy bypass host checking against no_proxy to be case-insensitive, and to not match unrelated host names that happen to have a bypassed hostname as a suffix. Patch by Xiang Zhang.
- bpo-26634 [https://bugs.python.org/issue?@action=redirect&bpo=26634]: recursive_repr() now sets __qualname__ of wrapper. Patch by Xiang Zhang.
- bpo-26804 [https://bugs.python.org/issue?@action=redirect&bpo=26804]: urllib.request will prefer lower_case proxy environment variables over UPPER_CASE or Mixed_Case ones. Patch contributed by Hans-Peter Jansen.
- bpo-26837 [https://bugs.python.org/issue?@action=redirect&bpo=26837]: assertSequenceEqual() now correctly outputs non-stringified differing items (like bytes in the -b mode). This affects assertListEqual() and assertTupleEqual().
- bpo-26041 [https://bugs.python.org/issue?@action=redirect&bpo=26041]: Remove “will be removed in Python 3.7” from deprecation messages of platform.dist() and platform.linux_distribution(). Patch by Kumaripaba Miyurusara Athukorala.
- bpo-26822 [https://bugs.python.org/issue?@action=redirect&bpo=26822]: itemgetter, attrgetter and methodcaller objects no longer silently ignore keyword arguments.

- bpo-26733 [https://bugs.python.org/issue?@action=redirect&bpo=26733]: Disassembling a class now disassembles class and static methods. Patch by Xiang Zhang.
- bpo-26801 [https://bugs.python.org/issue?@action=redirect&bpo=26801]: Fix error handling in `shutil.get_terminal_size()`, catch `AttributeError` instead of `NameError`. Patch written by Emanuel Barry.
- bpo-24838 [https://bugs.python.org/issue?@action=redirect&bpo=24838]: tarfile's ustar and gnu formats now correctly calculate name and link field limits for multibyte character encodings like utf-8.
- bpo-26717 [https://bugs.python.org/issue?@action=redirect&bpo=26717]: Stop encoding Latin-1-ized WSGI paths with UTF-8. Patch by Anthony Sottile.
- bpo-26735 [https://bugs.python.org/issue?@action=redirect&bpo=26735]: Fix `os.urandom()` on Solaris 11.3 and newer when reading more than 1,024 bytes: call `getrandom()` multiple times with a limit of 1024 bytes per call.
- bpo-16329 [https://bugs.python.org/issue?@action=redirect&bpo=16329]: Add .webm to mimetypes.types_map. Patch by Giampaolo Rodola'.
- bpo-13952 [https://bugs.python.org/issue?@action=redirect&bpo=13952]: Add .csv to mimetypes.types_map. Patch by Geoff Wilson.
- bpo-26709 [https://bugs.python.org/issue?@action=redirect&bpo=26709]: Fixed Y2038 problem in loading binary PLists.
- bpo-23735 [https://bugs.python.org/issue?@action=redirect&bpo=23735]: Handle terminal resizing with Readline 6.3+ by installing our own SIGWINCH handler. Patch by Eric Price.
- bpo-26586 [https://bugs.python.org/issue?@action=redirect&bpo=26586]: In http.server, respond with "413 Request header fields too large" if there are too

many header fields to parse, rather than killing the connection and raising an unhandled exception. Patch by Xiang Zhang.

- bpo-22854 [https://bugs.python.org/issue?@action=redirect&bpo=22854]: Change BufferedReader.writable() and BufferedWriter.readable() to always return False.
- bpo-25195 [https://bugs.python.org/issue?@action=redirect&bpo=25195]: Fix a regression in mock.MagicMock. _Call is a subclass of tuple (changeset 3603bae63c13 only works for classes) so we need to implement __ne__ ourselves. Patch by Andrew Plummer.
- bpo-26644 [https://bugs.python.org/issue?@action=redirect&bpo=26644]: Raise ValueError rather than SystemError when a negative length is passed to SSLSocket.recv() or read().
- bpo-23804 [https://bugs.python.org/issue?@action=redirect&bpo=23804]: Fix SSL recv(0) and read(0) methods to return zero bytes instead of up to 1024.
- bpo-26616 [https://bugs.python.org/issue?@action=redirect&bpo=26616]: Fixed a bug in datetime.astimezone() method.
- bpo-21925 [https://bugs.python.org/issue?@action=redirect&bpo=21925]: **`warnings.formatwarning()`** now catches exceptions on `linecache.getline(...)` to be able to log **`ResourceWarning`** emitted late during the Python shutdown process.
- bpo-24266 [https://bugs.python.org/issue?@action=redirect&bpo=24266]: Ctrl+C during Readline history search now cancels the search mode when compiled with Readline 7.
- bpo-26560 [https://bugs.python.org/issue?@action=redirect&bpo=26560]: Avoid potential ValueError in BaseHandler.start_response. Initial patch by Peter Inglesby.

- bpo-26569 [https://bugs.python.org/issue?@action=redirect&bpo=26569]: Fix `pyclbr.readmodule()` and `pyclbr.readmodule_ex()` to support importing packages.
- bpo-26499 [https://bugs.python.org/issue?@action=redirect&bpo=26499]: Account for remaining Content-Length in HTTPResponse.readline() and read1(). Based on patch by Silent Ghost. Also document that HTTPResponse now supports these methods.
- bpo-25320 [https://bugs.python.org/issue?@action=redirect&bpo=25320]: Handle sockets in directories unittest discovery is scanning. Patch from Victor van den Elzen.
- bpo-16181 [https://bugs.python.org/issue?@action=redirect&bpo=16181]: cookiejar.http2time() now returns None if year is higher than datetime.MAXYEAR.
- bpo-26513 [https://bugs.python.org/issue?@action=redirect&bpo=26513]: Fixes platform module detection of Windows Server
- bpo-23718 [https://bugs.python.org/issue?@action=redirect&bpo=23718]: Fixed parsing time in week 0 before Jan 1. Original patch by Tamás Bence Gedai.
- bpo-20589 [https://bugs.python.org/issue?@action=redirect&bpo=20589]: Invoking Path.owner() and Path.group() on Windows now raise NotImplementedError instead of ImportError.
- bpo-26177 [https://bugs.python.org/issue?@action=redirect&bpo=26177]: Fixed the keys() method for Canvas and Scrollbar widgets.
- bpo-15068 [https://bugs.python.org/issue?@action=redirect&bpo=15068]: Got rid of excessive buffering in the fileinput module. The bufsize parameter is no longer used.
- bpo-2202 [https://bugs.python.org/issue?@action=redirect&bpo=2202]: Fix UnboundLocalError in AbstractDigestAuthHandler.get_algorithm_impls. Initial patch by Mathieu Dupuy.

- bpo-25718 [https://bugs.python.org/issue?@action=redirect&bpo=25718]: Fixed pickling and copying the accumulate() iterator with total is None.
- bpo-26475 [https://bugs.python.org/issue?@action=redirect&bpo=26475]: Fixed debugging output for regular expressions with the (?x) flag.
- bpo-26457 [https://bugs.python.org/issue?@action=redirect&bpo=26457]: Fixed the subnets() methods in IP network classes for the case when resulting prefix length is equal to maximal prefix length. Based on patch by Xiang Zhang.
- bpo-26385 [https://bugs.python.org/issue?@action=redirect&bpo=26385]: Remove the file if the internal open() call in NamedTemporaryFile() fails. Patch by Silent Ghost.
- bpo-26402 [https://bugs.python.org/issue?@action=redirect&bpo=26402]: Fix XML-RPC client to retry when the server shuts down a persistent connection. This was a regression related to the new http.client.RemoteDisconnected exception in 3.5.0a4.
- bpo-25913 [https://bugs.python.org/issue?@action=redirect&bpo=25913]: Leading `<~` is optional now in base64.a85decode() with adobe=True. Patch by Swati Jaiswal.
- bpo-26186 [https://bugs.python.org/issue?@action=redirect&bpo=26186]: Remove an invalid type check in importlib.util.LazyLoader.
- bpo-26367 [https://bugs.python.org/issue?@action=redirect&bpo=26367]: importlib.__import__() raises SystemError like builtins.__import__() when `level` is specified but without an accompanying package specified.
- bpo-26309 [https://bugs.python.org/issue?@action=redirect&bpo=26309]: In the “socketserver” module, shut down the request (closing the connected socket)

when verify_request() returns false. Patch by Aviv Palivoda.

- bpo-25995 [https://bugs.python.org/issue?@action=redirect&bpo=25995]: os.walk() no longer uses FDs proportional to the tree depth.
- bpo-26117 [https://bugs.python.org/issue?@action=redirect&bpo=26117]: The os.scandir() iterator now closes file descriptor not only when the iteration is finished, but when it was failed with error.
- bpo-25911 [https://bugs.python.org/issue?@action=redirect&bpo=25911]: Restored support of bytes paths in os.walk() on Windows.
- bpo-26045 [https://bugs.python.org/issue?@action=redirect&bpo=26045]: Add UTF-8 suggestion to error message when posting a non-Latin-1 string with http.client.
- bpo-12923 [https://bugs.python.org/issue?@action=redirect&bpo=12923]: Reset FancyURLopener’s redirect counter even if there is an exception. Based on patches by Brian Brazil and Daniel Rocco.
- bpo-25945 [https://bugs.python.org/issue?@action=redirect&bpo=25945]: Fixed a crash when unpickle the functools.partial object with wrong state. Fixed a leak in failed functools.partial constructor. “args” and “keywords” attributes of functools.partial have now always types tuple and dict correspondingly.
- bpo-26202 [https://bugs.python.org/issue?@action=redirect&bpo=26202]: copy.deepcopy() now correctly copies range() objects with non-atomic attributes.
- bpo-23076 [https://bugs.python.org/issue?@action=redirect&bpo=23076]: Path.glob() now raises a ValueError if it’s called with an invalid pattern. Patch by Thomas Nyberg.
- bpo-19883 [https://bugs.python.org/issue?@action=redirect&bpo=19883]: Fixed possible integer overflows in zipimport.

- bpo-26227 [https://bugs.python.org/issue?@action=redirect&bpo=26227]: On Windows, getnameinfo(), gethostbyaddr() and gethostbyname_ex() functions of the socket module now decode the hostname from the ANSI code page rather than UTF-8.
- bpo-26147 [https://bugs.python.org/issue?@action=redirect&bpo=26147]: xmlrpc now works with strings not encodable with used non-UTF-8 encoding.
- bpo-25935 [https://bugs.python.org/issue?@action=redirect&bpo=25935]: Garbage collector now breaks reference loops with OrderedDict.
- bpo-16620 [https://bugs.python.org/issue?@action=redirect&bpo=16620]: Fixed AttributeError in msilib.Directory.glob().
- bpo-26013 [https://bugs.python.org/issue?@action=redirect&bpo=26013]: Added compatibility with broken protocol 2 pickles created in old Python 3 versions (3.4.3 and lower).
- bpo-25850 [https://bugs.python.org/issue?@action=redirect&bpo=25850]: Use cross-compilation by default for 64-bit Windows.
- bpo-17633 [https://bugs.python.org/issue?@action=redirect&bpo=17633]: Improve zipimport’s support for namespace packages.
- bpo-24705 [https://bugs.python.org/issue?@action=redirect&bpo=24705]: Fix sysconfig._parse_makefile not expanding ${} vars appearing before $() vars.
- bpo-22138 [https://bugs.python.org/issue?@action=redirect&bpo=22138]: Fix mock.patch behavior when patching descriptors. Restore original values after patching. Patch contributed by Sean McCully.
- bpo-25672 [https://bugs.python.org/issue?@action=redirect&bpo=25672]: In the ssl module, enable the SSL_MODE_RELEASE_BUFFERS mode option if it is safe to do so.

- bpo-26012 [https://bugs.python.org/issue?@action=redirect&bpo=26012]: Don’t traverse into symlinks for ** pattern in pathlib.Path.[r]glob().
- bpo-24120 [https://bugs.python.org/issue?@action=redirect&bpo=24120]: Ignore PermissionError when traversing a tree with pathlib.Path.[r]glob(). Patch by Ulrich Petri.
- bpo-25447 [https://bugs.python.org/issue?@action=redirect&bpo=25447]: fileinput now uses sys.stdin as-is if it does not have a buffer attribute (restores backward compatibility).
- bpo-25447 [https://bugs.python.org/issue?@action=redirect&bpo=25447]: Copying the lru_cache() wrapper object now always works, independently from the type of the wrapped object (by returning the original object unchanged).
- bpo-24103 [https://bugs.python.org/issue?@action=redirect&bpo=24103]: Fixed possible use after free in ElementTree.XMLPullParser.
- bpo-25860 [https://bugs.python.org/issue?@action=redirect&bpo=25860]: os.fwalk() no longer skips remaining directories when error occurs. Original patch by Samson Lee.
- bpo-25914 [https://bugs.python.org/issue?@action=redirect&bpo=25914]: Fixed and simplified OrderedDict.__sizeof__.
- bpo-25902 [https://bugs.python.org/issue?@action=redirect&bpo=25902]: Fixed various refcount issues in ElementTree iteration.
- bpo-25717 [https://bugs.python.org/issue?@action=redirect&bpo=25717]: Restore the previous behaviour of tolerating most fstat() errors when opening files. This was a regression in 3.5a1, and stopped anonymous temporary files from working in special cases.
- bpo-24903 [https://bugs.python.org/issue?@action=redirect&bpo=24903]: Fix regression in number of

arguments compileall accepts when ‘-d’ is specified. The check on the number of arguments has been dropped completely as it never worked correctly anyway.

- bpo-25764 [https://bugs.python.org/issue?@action=redirect&bpo=25764]: In the subprocess module, preserve any exception caused by fork() failure when preexec_fn is used.
- bpo-6478 [https://bugs.python.org/issue?@action=redirect&bpo=6478]: _strptime’s regexp cache now is reset after changing timezone with time.tzset().
- bpo-14285 [https://bugs.python.org/issue?@action=redirect&bpo=14285]: When executing a package with the “python -m package” option, and package initialization fails, a proper traceback is now reported. The “runpy” module now lets exceptions from package initialization pass back to the caller, rather than raising ImportError.
- bpo-19771 [https://bugs.python.org/issue?@action=redirect&bpo=19771]: Also in runpy and the “-m” option, omit the irrelevant message “… is a package and cannot be directly executed” if the package could not even be initialized (e.g. due to a bad `*.pyc` file).
- bpo-25177 [https://bugs.python.org/issue?@action=redirect&bpo=25177]: Fixed problem with the mean of very small and very large numbers. As a side effect, statistics.mean and statistics.variance should be significantly faster.
- bpo-25718 [https://bugs.python.org/issue?@action=redirect&bpo=25718]: Fixed copying object with state with boolean value is false.
- bpo-10131 [https://bugs.python.org/issue?@action=redirect&bpo=10131]: Fixed deep copying of minidom documents. Based on patch by Marian Ganisin.
- bpo-25725 [https://bugs.python.org/issue?@action=redirect&bpo=25725]: Fixed a reference leak in

pickle.loads() when unpickling invalid data including tuple instructions.

- bpo-25663 [https://bugs.python.org/issue?@action=redirect&bpo=25663]: In the Readline completer, avoid listing duplicate global names, and search the global namespace before searching builtins.
- bpo-25688 [https://bugs.python.org/issue?@action=redirect&bpo=25688]: Fixed file leak in ElementTree.iterparse() raising an error.
- bpo-23914 [https://bugs.python.org/issue?@action=redirect&bpo=23914]: Fixed SystemError raised by unpickler on broken pickle data.
- bpo-25691 [https://bugs.python.org/issue?@action=redirect&bpo=25691]: Fixed crash on deleting ElementTree.Element attributes.
- bpo-25624 [https://bugs.python.org/issue?@action=redirect&bpo=25624]: ZipFile now always writes a ZIP_STORED header for directory entries. Patch by Dingyuan Wang.
- Skip getaddrinfo if host is already resolved. Patch by A. Jesse Jiryu Davis.
- bpo-26050 [https://bugs.python.org/issue?@action=redirect&bpo=26050]: Add asyncio.StreamReader.readuntil() method. Patch by Марк Коренберг.
- bpo-25924 [https://bugs.python.org/issue?@action=redirect&bpo=25924]: Avoid unnecessary serialization of getaddrinfo(3) calls on OS X versions 10.5 or higher. Original patch by A. Jesse Jiryu Davis.
- bpo-26406 [https://bugs.python.org/issue?@action=redirect&bpo=26406]: Avoid unnecessary serialization of getaddrinfo(3) calls on current versions of OpenBSD and NetBSD. Patch by A. Jesse Jiryu Davis.
- bpo-26848 [https://bugs.python.org/issue?@action=redirect&bpo=26848]: Fix

asyncio/subprocess.communicate() to handle empty input. Patch by Jack O’Connor.

- bpo-27040 [https://bugs.python.org/issue?@action=redirect&bpo=27040]: Add loop.get_exception_handler method
- bpo-27041 [https://bugs.python.org/issue?@action=redirect&bpo=27041]: asyncio: Add loop.create_future method
- bpo-27223 [https://bugs.python.org/issue?@action=redirect&bpo=27223]: asyncio: Fix _read_ready and _write_ready to respect _conn_lost. Patch by Łukasz Langa.
- bpo-22970 [https://bugs.python.org/issue?@action=redirect&bpo=22970]: asyncio: Fix inconsistency cancelling Condition.wait. Patch by David Coles.

IDLE

- bpo-5124 [https://bugs.python.org/issue?@action=redirect&bpo=5124]: Paste with text selected now replaces the selection on X11. This matches how paste works on Windows, Mac, most modern Linux apps, and ttk widgets. Original patch by Serhiy Storchaka.
- bpo-24759 [https://bugs.python.org/issue?@action=redirect&bpo=24759]: Make clear in idlelib.idle_test.__init__ that the directory is a private implementation of test.test_idle and tool for maintainers.
- bpo-27196 [https://bugs.python.org/issue?@action=redirect&bpo=27196]: Stop ‘ThemeChanged’ warnings when running IDLE tests. These persisted after other warnings were suppressed in #20567. Apply Serhiy Storchaka’s update_idletasks solution to four test files. Record this additional advice in idle_test/README.txt
- bpo-20567 [https://bugs.python.org/issue?@action=redirect&bpo=20567]: Revise idle_test/README.txt

with advice about avoiding tk warning messages from tests. Apply advice to several IDLE tests.

- bpo-27117 [https://bugs.python.org/issue?@action=redirect&bpo=27117]: Make colorizer htest and turtledemo work with dark themes. Move code for configuring text widget colors to a new function.
- bpo-26673 [https://bugs.python.org/issue?@action=redirect&bpo=26673]: When tk reports font size as 0, change to size 10. Such fonts on Linux prevented the configuration dialog from opening.
- bpo-21939 [https://bugs.python.org/issue?@action=redirect&bpo=21939]: Add test for IDLE’s percolator. Original patch by Saimadhav Heblikar.
- bpo-21676 [https://bugs.python.org/issue?@action=redirect&bpo=21676]: Add test for IDLE’s replace dialog. Original patch by Saimadhav Heblikar.
- bpo-18410 [https://bugs.python.org/issue?@action=redirect&bpo=18410]: Add test for IDLE’s search dialog. Original patch by Westley Martínez.
- bpo-21703 [https://bugs.python.org/issue?@action=redirect&bpo=21703]: Add test for IDLE’s undo delegator. Original patch by Saimadhav Heblikar .
- bpo-27044 [https://bugs.python.org/issue?@action=redirect&bpo=27044]: Add ConfigDialog.remove_var_callbacks to stop memory leaks.
- bpo-23977 [https://bugs.python.org/issue?@action=redirect&bpo=23977]: Add more asserts to test_delegator.
- bpo-20640 [https://bugs.python.org/issue?@action=redirect&bpo=20640]: Add tests for idlelib.configHelpSourceEdit. Patch by Saimadhav Heblikar.
- In the ‘IDLE-console differences’ section of the IDLE doc, clarify how running with IDLE affects sys.modules and the standard streams.

- bpo-25507 [https://bugs.python.org/issue?@action=redirect&bpo=25507]: fix incorrect change in IOBinding that prevented printing. Augment IOBinding htest to include all major IOBinding functions.
- bpo-25905 [https://bugs.python.org/issue?@action=redirect&bpo=25905]: Revert unwanted conversion of ‘ to ’ RIGHT SINGLE QUOTATION MARK in README.txt and open this and NEWS.txt with ‘ascii’. Re-encode CREDITS.txt to utf-8 and open it with ‘utf-8’.

Documentation

- bpo-19489 [https://bugs.python.org/issue?@action=redirect&bpo=19489]: Moved the search box from the sidebar to the header and footer of each page. Patch by Ammar Askar.
- bpo-24136 [https://bugs.python.org/issue?@action=redirect&bpo=24136]: Document the new **PEP 448** [https://peps.python.org/pep-0448/] unpacking syntax of 3.5.
- bpo-26736 [https://bugs.python.org/issue?@action=redirect&bpo=26736]: Used HTTPS for external links in the documentation if possible.
- bpo-6953 [https://bugs.python.org/issue?@action=redirect&bpo=6953]: Rework the Readline module documentation to group related functions together, and add more details such as what underlying Readline functions and variables are accessed.
- bpo-23606 [https://bugs.python.org/issue?@action=redirect&bpo=23606]: Adds note to ctypes documentation regarding cdll.msvcrt.
- bpo-25500 [https://bugs.python.org/issue?@action=redirect&bpo=25500]: Fix documentation to not claim that __import__ is searched for in the global scope.
- bpo-26014 [https://bugs.python.org/issue?@action=redirect&bpo=26014]: Update 3.x packaging documentation: * “See also” links to the new docs are

now provided in the legacy pages * links to setuptools documentation have been updated

Tests

- bpo-21916 [https://bugs.python.org/issue?@action=redirect&bpo=21916]: Added tests for the turtle module. Patch by ingrid, Gregory Loyse and Jelle Zijlstra.
- bpo-26523 [https://bugs.python.org/issue?@action=redirect&bpo=26523]: The multiprocessing thread pool (multiprocessing.dummy.Pool) was untested.
- bpo-26015 [https://bugs.python.org/issue?@action=redirect&bpo=26015]: Added new tests for pickling iterators of mutable sequences.
- bpo-26325 [https://bugs.python.org/issue?@action=redirect&bpo=26325]: Added test.support.check_no_resource_warning() to check that no ResourceWarning is emitted.
- bpo-25940 [https://bugs.python.org/issue?@action=redirect&bpo=25940]: Changed test_ssl to use self-signed.pythontest.net. This avoids relying on svn.python.org, which recently changed root certificate.
- bpo-25616 [https://bugs.python.org/issue?@action=redirect&bpo=25616]: Tests for OrderedDict are extracted from test_collections into separate file test_ordered_dict.
- bpo-26583 [https://bugs.python.org/issue?@action=redirect&bpo=26583]: Skip test_timestamp_overflow in test_import if bytecode files cannot be written.

Build

- bpo-26884 [https://bugs.python.org/issue?@action=redirect&bpo=26884]: Fix linking extension modules for cross builds. Patch by Xavier de Gaye.

- bpo-22359 [https://bugs.python.org/issue?@action=redirect&bpo=22359]: Disable the rules for running _freeze_importlib and pgen when cross-compiling. The output of these programs is normally saved with the source code anyway, and is still regenerated when doing a native build. Patch by Xavier de Gaye.
- bpo-27229 [https://bugs.python.org/issue?@action=redirect&bpo=27229]: Fix the cross-compiling pgen rule for in-tree builds. Patch by Xavier de Gaye.
- bpo-21668 [https://bugs.python.org/issue?@action=redirect&bpo=21668]: Link audioop, _datetime, _ctypes_test modules to libm, except on Mac OS X. Patch written by Xavier de Gaye.
- bpo-25702 [https://bugs.python.org/issue?@action=redirect&bpo=25702]: A –with-lto configure option has been added that will enable link time optimizations at build time during a make profile-opt. Some compilers and toolchains are known to not produce stable code when using LTO, be sure to test things thoroughly before relying on it. It can provide a few % speed up over profile-opt alone.
- bpo-26624 [https://bugs.python.org/issue?@action=redirect&bpo=26624]: Adds validation of ucrtbase[d].dll version with warning for old versions.
- bpo-17603 [https://bugs.python.org/issue?@action=redirect&bpo=17603]: Avoid error about nonexistent fileblocks.o file by using a lower-level check for st_blocks in struct stat.
- bpo-26079 [https://bugs.python.org/issue?@action=redirect&bpo=26079]: Fixing the build output folder for tix-8.4.3.6. Patch by Bjoern Thiel.
- bpo-26465 [https://bugs.python.org/issue?@action=redirect&bpo=26465]: Update Windows builds to use OpenSSL 1.0.2g.
- bpo-24421 [https://bugs.python.org/issue?@action=redirect&bpo=24421]: Compile Modules/_math.c once,

before building extensions. Previously it could fail to compile properly if the math and cmath builds were concurrent.

- bpo-25348 [https://bugs.python.org/issue?@action=redirect&bpo=25348]: Added `--pgo` and `--pgo-job` arguments to `PCbuild\build.bat` for building with Profile-Guided Optimization. The old `PCbuild\build_pgo.bat` script is now deprecated, and simply calls `PCbuild\build.bat --pgo %*`.
- bpo-25827 [https://bugs.python.org/issue?@action=redirect&bpo=25827]: Add support for building with ICC to `configure`, including a new `--with-icc` flag.
- bpo-25696 [https://bugs.python.org/issue?@action=redirect&bpo=25696]: Fix installation of Python on UNIX with make -j9.
- bpo-26930 [https://bugs.python.org/issue?@action=redirect&bpo=26930]: Update OS X 10.5+ 32-bit-only installer to build and link with OpenSSL 1.0.2h.
- bpo-26268 [https://bugs.python.org/issue?@action=redirect&bpo=26268]: Update Windows builds to use OpenSSL 1.0.2f.
- bpo-25136 [https://bugs.python.org/issue?@action=redirect&bpo=25136]: Support Apple Xcode 7’s new textual SDK stub libraries.
- bpo-24324 [https://bugs.python.org/issue?@action=redirect&bpo=24324]: Do not enable unreachable code warnings when using gcc as the option does not work correctly in older versions of gcc and has been silently removed as of gcc-4.5.

Windows

- bpo-27053 [https://bugs.python.org/issue?@action=redirect&bpo=27053]: Updates make_zip.py to correctly generate library ZIP file.

- bpo-26268 [https://bugs.python.org/issue?@action=redirect&bpo=26268]: Update the prepare_ssl.py script to handle OpenSSL releases that don't include the contents of the include directory (that is, 1.0.2e and later).
- bpo-26071 [https://bugs.python.org/issue?@action=redirect&bpo=26071]: bdist_wininst created binaries fail to start and find 32bit Python
- bpo-26073 [https://bugs.python.org/issue?@action=redirect&bpo=26073]: Update the list of magic numbers in launcher
- bpo-26065 [https://bugs.python.org/issue?@action=redirect&bpo=26065]: Excludes venv from library when generating embeddable distro.
- bpo-17500 [https://bugs.python.org/issue?@action=redirect&bpo=17500]: Remove unused and outdated icons. (See also: https://github.com/python/pythondotorg/issues/945)

Tools/Demos

- bpo-26799 [https://bugs.python.org/issue?@action=redirect&bpo=26799]: Fix python-gdb.py: don't get C types once when the Python code is loaded, but get C types on demand. The C types can change if python-gdb.py is loaded before the Python executable. Patch written by Thomas Ilsche.
- bpo-26271 [https://bugs.python.org/issue?@action=redirect&bpo=26271]: Fix the Freeze tool to properly use flags passed through configure. Patch by Daniel Shaulov.
- bpo-26489 [https://bugs.python.org/issue?@action=redirect&bpo=26489]: Add dictionary unpacking support to Tools/parser/unparse.py. Patch by Guo Ci Teo.
- bpo-26316 [https://bugs.python.org/issue?@action=redirect&bpo=26316]: Fix variable name typo in

Argument Clinic.

Python 3.5.1 final

Release date: 2015-12-06

Core and Builtins

- bpo-25709 [https://bugs.python.org/issue?@action=redirect&bpo=25709]: Fixed problem with in-place string concatenation and utf-8 cache.

Windows

- bpo-25715 [https://bugs.python.org/issue?@action=redirect&bpo=25715]: Python 3.5.1 installer shows wrong upgrade path and incorrect logic for launcher detection.

Python 3.5.1 release candidate 1

Release date: 2015-11-22

Core and Builtins

- bpo-25630 [https://bugs.python.org/issue?@action=redirect&bpo=25630]: Fix a possible segfault during argument parsing in functions that accept filesystem paths.
- bpo-23564 [https://bugs.python.org/issue?@action=redirect&bpo=23564]: Fixed a partially broken sanity check in the _posixsubprocess internals regarding how fds_to_pass were passed to the child. The bug had no actual impact as subprocess.py already avoided it.

- bpo-25388 [https://bugs.python.org/issue?@action=redirect&bpo=25388]: Fixed tokenizer crash when processing undecodable source code with a null byte.
- bpo-25462 [https://bugs.python.org/issue?@action=redirect&bpo=25462]: The hash of the key now is calculated only once in most operations in C implementation of OrderedDict.
- bpo-22995 [https://bugs.python.org/issue?@action=redirect&bpo=22995]: Default implementation of __reduce__ and __reduce_ex__ now rejects builtin types with not defined __new__.
- bpo-25555 [https://bugs.python.org/issue?@action=redirect&bpo=25555]: Fix parser and AST: fill lineno and col_offset of “arg” node when compiling AST from Python objects.
- bpo-24802 [https://bugs.python.org/issue?@action=redirect&bpo=24802]: Avoid buffer overreads when int(), float(), compile(), exec() and eval() are passed bytes-like objects. These objects are not necessarily terminated by a null byte, but the functions assumed they were.
- bpo-24726 [https://bugs.python.org/issue?@action=redirect&bpo=24726]: Fixed a crash and leaking NULL in repr() of OrderedDict that was mutated by direct calls of dict methods.
- bpo-25449 [https://bugs.python.org/issue?@action=redirect&bpo=25449]: Iterating OrderedDict with keys with unstable hash now raises KeyError in C implementations as well as in Python implementation.
- bpo-25395 [https://bugs.python.org/issue?@action=redirect&bpo=25395]: Fixed crash when highly nested OrderedDict structures were garbage collected.
- bpo-25274 [https://bugs.python.org/issue?@action=redirect&bpo=25274]: sys.setrecursionlimit() now raises a RecursionError if the new recursion limit is too low depending at the current recursion depth. Modify

also the “lower-water mark” formula to make it monotonic. This mark is used to decide when the overflowed flag of the thread state is reset.
- bpo-24402 [https://bugs.python.org/issue?@action=redirect&bpo=24402]: Fix input() to prompt to the redirected stdout when sys.stdout.fileno() fails.
- bpo-24806 [https://bugs.python.org/issue?@action=redirect&bpo=24806]: Prevent builtin types that are not allowed to be subclassed from being subclassed through multiple inheritance.
- bpo-24848 [https://bugs.python.org/issue?@action=redirect&bpo=24848]: Fixed a number of bugs in UTF-7 decoding of misformed data.
- bpo-25280 [https://bugs.python.org/issue?@action=redirect&bpo=25280]: Import trace messages emitted in verbose (-v) mode are no longer formatted twice.
- bpo-25003 [https://bugs.python.org/issue?@action=redirect&bpo=25003]: On Solaris 11.3 or newer, os.urandom() now uses the getrandom() function instead of the getentropy() function. The getentropy() function is blocking to generate very good quality entropy, os.urandom() doesn’t need such high-quality entropy.
- bpo-25182 [https://bugs.python.org/issue?@action=redirect&bpo=25182]: The stdprinter (used as sys.stderr before the io module is imported at startup) now uses the backslashreplace error handler.
- bpo-25131 [https://bugs.python.org/issue?@action=redirect&bpo=25131]: Make the line number and column offset of set/dict literals and comprehensions correspond to the opening brace.
- bpo-25150 [https://bugs.python.org/issue?@action=redirect&bpo=25150]: Hide the private `_Py_atomic_xxx` symbols from the public Python.h header to fix a compilation error with OpenMP. PyThreadState_GET()

becomes an alias to PyThreadState_Get() to avoid ABI incompatibilities.

Library

- bpo-25626 [https://bugs.python.org/issue?@action=redirect&bpo=25626]: Change three zlib functions to accept sizes that fit in Py_ssize_t, but internally cap those sizes to UINT_MAX. This resolves a regression in 3.5 where GzipFile.read() failed to read chunks larger than 2 or 4 GiB. The change affects the zlib.Decompress.decompress() max_length parameter, the zlib.decompress() bufsize parameter, and the zlib.Decompress.flush() length parameter.
- bpo-25583 [https://bugs.python.org/issue?@action=redirect&bpo=25583]: Avoid incorrect errors raised by os.makedirs(exist_ok=True) when the OS gives priority to errors such as EACCES over EEXIST.
- bpo-25593 [https://bugs.python.org/issue?@action=redirect&bpo=25593]: Change semantics of EventLoop.stop() in asyncio.
- bpo-6973 [https://bugs.python.org/issue?@action=redirect&bpo=6973]: When we know a subprocess.Popen process has died, do not allow the send_signal(), terminate(), or kill() methods to do anything as they could potentially signal a different process.
- bpo-25590 [https://bugs.python.org/issue?@action=redirect&bpo=25590]: In the Readline completer, only call getattr() once per attribute.
- bpo-25498 [https://bugs.python.org/issue?@action=redirect&bpo=25498]: Fix a crash when garbage-collecting ctypes objects created by wrapping a memoryview. This was a regression made in 3.5a1. Based on patch by Eryksun.
- bpo-25584 [https://bugs.python.org/issue?@action=redirect&bpo=25584]: Added “escape” to the __all__

list in the glob module.

- bpo-25584 [https://bugs.python.org/issue?@action=redirect&bpo=25584]: Fixed recursive glob() with patterns starting with **.
- bpo-25446 [https://bugs.python.org/issue?@action=redirect&bpo=25446]: Fix regression in smtplib's AUTH LOGIN support.
- bpo-18010 [https://bugs.python.org/issue?@action=redirect&bpo=18010]: Fix the pydoc web server's module search function to handle exceptions from importing packages.
- bpo-25554 [https://bugs.python.org/issue?@action=redirect&bpo=25554]: Got rid of circular references in regular expression parsing.
- bpo-25510 [https://bugs.python.org/issue?@action=redirect&bpo=25510]: fileinput.FileInput.readline() now returns b'' instead of '' at the end if the FileInput was opened with binary mode. Patch by Ryosuke Ito.
- bpo-25503 [https://bugs.python.org/issue?@action=redirect&bpo=25503]: Fixed inspect.getdoc() for inherited docstrings of properties. Original patch by John Mark Vandenberg.
- bpo-25515 [https://bugs.python.org/issue?@action=redirect&bpo=25515]: Always use os.urandom as a source of randomness in uuid.uuid4.
- bpo-21827 [https://bugs.python.org/issue?@action=redirect&bpo=21827]: Fixed textwrap.dedent() for the case when largest common whitespace is a substring of smallest leading whitespace. Based on patch by Robert Li.
- bpo-25447 [https://bugs.python.org/issue?@action=redirect&bpo=25447]: The lru_cache() wrapper objects now can be copied and pickled (by returning the original object unchanged).
- bpo-25390 [https://bugs.python.org/issue?@action=redirect&bpo=25390]: typing: Don't crash on

Union[str, Pattern].

- bpo-25441 [https://bugs.python.org/issue?@action=redirect&bpo=25441]: asyncio: Raise error from drain() when socket is closed.
- bpo-25410 [https://bugs.python.org/issue?@action=redirect&bpo=25410]: Cleaned up and fixed minor bugs in C implementation of OrderedDict.
- bpo-25411 [https://bugs.python.org/issue?@action=redirect&bpo=25411]: Improved Unicode support in SMTPHandler through better use of the email package. Thanks to user simon04 for the patch.
- bpo-25407 [https://bugs.python.org/issue?@action=redirect&bpo=25407]: Remove mentions of the formatter module being removed in Python 3.6.
- bpo-25406 [https://bugs.python.org/issue?@action=redirect&bpo=25406]: Fixed a bug in C implementation of OrderedDict.move_to_end() that caused segmentation fault or hang in iterating after moving several items to the start of ordered dict.
- bpo-25364 [https://bugs.python.org/issue?@action=redirect&bpo=25364]: zipfile now works in threads disabled builds.
- bpo-25328 [https://bugs.python.org/issue?@action=redirect&bpo=25328]: smtpd’s SMTPChannel now correctly raises a ValueError if both decode_data and enable_SMTPUTF8 are set to true.
- bpo-25316 [https://bugs.python.org/issue?@action=redirect&bpo=25316]: distutils raises OSError instead of DistutilsPlatformError when MSVC is not installed.
- bpo-25380 [https://bugs.python.org/issue?@action=redirect&bpo=25380]: Fixed protocol for the STACK_GLOBAL opcode in pickletools.opcodes.
- bpo-23972 [https://bugs.python.org/issue?@action=redirect&bpo=23972]: Updates asyncio datagram create method allowing reuseport and reuseaddr socket options to be set prior to binding the socket. Mirroring

the existing asyncio create_server method the reuseaddr option for datagram sockets defaults to True if the O/S is ‘posix’ (except if the platform is Cygwin). Patch by Chris Laws.

- bpo-25304 [https://bugs.python.org/issue?@action=redirect&bpo=25304]: Add asyncio.run_coroutine_threadsafe(). This lets you submit a coroutine to a loop from another thread, returning a concurrent.futures.Future. By Vincent Michel.
- bpo-25232 [https://bugs.python.org/issue?@action=redirect&bpo=25232]: Fix CGIRequestHandler to split the query from the URL at the first question mark (?) rather than the last. Patch from Xiang Zhang.
- bpo-24657 [https://bugs.python.org/issue?@action=redirect&bpo=24657]: Prevent CGIRequestHandler from collapsing slashes in the query part of the URL as if it were a path. Patch from Xiang Zhang.
- bpo-24483 [https://bugs.python.org/issue?@action=redirect&bpo=24483]: C implementation of functools.lru_cache() now calculates key’s hash only once.
- bpo-22958 [https://bugs.python.org/issue?@action=redirect&bpo=22958]: Constructor and update method of weakref.WeakValueDictionary now accept the self and the dict keyword arguments.
- bpo-22609 [https://bugs.python.org/issue?@action=redirect&bpo=22609]: Constructor of collections.UserDict now accepts the self keyword argument.
- bpo-25111 [https://bugs.python.org/issue?@action=redirect&bpo=25111]: Fixed comparison of traceback.FrameSummary.
- bpo-25262 [https://bugs.python.org/issue?@action=redirect&bpo=25262]: Added support for BINBYTES8 opcode in Python implementation of unpickler. Highest 32 bits of 64-bit size for BINUNICODE8 and BINBYTES8

opcodes no longer silently ignored on 32-bit platforms in C implementation.

- bpo-25034 [https://bugs.python.org/issue?@action=redirect&bpo=25034]: Fix string.Formatter problem with auto-numbering and nested format_specs. Patch by Anthon van der Neut.
- bpo-25233 [https://bugs.python.org/issue?@action=redirect&bpo=25233]: Rewrite the guts of asyncio.Queue and asyncio.Semaphore to be more understandable and correct.
- bpo-25203 [https://bugs.python.org/issue?@action=redirect&bpo=25203]: Failed readline.set_completer_delims() no longer left the module in inconsistent state.
- bpo-23600 [https://bugs.python.org/issue?@action=redirect&bpo=23600]: Default implementation of tzinfo.fromutc() was returning wrong results in some cases.
- bpo-23329 [https://bugs.python.org/issue?@action=redirect&bpo=23329]: Allow the ssl module to be built with older versions of LibreSSL.
- Prevent overflow in _Unpickler_Read.
- bpo-25047 [https://bugs.python.org/issue?@action=redirect&bpo=25047]: The XML encoding declaration written by Element Tree now respects the letter case given by the user. This restores the ability to write encoding names in uppercase like “UTF-8”, which worked in Python 2.
- bpo-25135 [https://bugs.python.org/issue?@action=redirect&bpo=25135]: Make deque_clear() safer by emptying the deque before clearing. This helps avoid possible reentrancy issues.
- bpo-19143 [https://bugs.python.org/issue?@action=redirect&bpo=19143]: platform module now reads Windows version from kernel32.dll to avoid compatibility shims.

- bpo-25092 [https://bugs.python.org/issue?@action=redirect&bpo=25092]: Fix datetime.strftime() failure when errno was already set to EINVAL.
- bpo-23517 [https://bugs.python.org/issue?@action=redirect&bpo=23517]: Fix rounding in fromtimestamp() and utcfromtimestamp() methods of datetime.datetime: microseconds are now rounded to nearest with ties going to nearest even integer (ROUND_HALF_EVEN), instead of being rounding towards minus infinity (ROUND_FLOOR). It's important that these methods use the same rounding mode than datetime.timedelta to keep the property: (datetime(1970,1,1) + timedelta(seconds=t)) == datetime.utcfromtimestamp(t). It also the rounding mode used by round(float) for example.
- bpo-25155 [https://bugs.python.org/issue?@action=redirect&bpo=25155]: Fix datetime.datetime.now() and datetime.datetime.utcnow() on Windows to support date after year 2038. It was a regression introduced in Python 3.5.0.
- bpo-25108 [https://bugs.python.org/issue?@action=redirect&bpo=25108]: Omitted internal frames in traceback functions print_stack(), format_stack(), and extract_stack() called without arguments.
- bpo-25118 [https://bugs.python.org/issue?@action=redirect&bpo=25118]: Fix a regression of Python 3.5.0 in os.waitpid() on Windows.
- bpo-24684 [https://bugs.python.org/issue?@action=redirect&bpo=24684]: socket.socket.getaddrinfo() now calls PyUnicode_AsEncodedString() instead of calling the encode() method of the host, to handle correctly custom string with an encode() method which doesn't return a byte string. The encoder of the IDNA codec is now called directly instead of calling the encode() method of the string.

- bpo-25060 [https://bugs.python.org/issue?@action=redirect&bpo=25060]: Correctly compute stack usage of the BUILD_MAP opcode.
- bpo-24857 [https://bugs.python.org/issue?@action=redirect&bpo=24857]: Comparing call_args to a long sequence now correctly returns a boolean result instead of raising an exception. Patch by A Kaptur.
- bpo-23144 [https://bugs.python.org/issue?@action=redirect&bpo=23144]: Make sure that HTMLParser.feed() returns all the data, even when convert_charrefs is True.
- bpo-24982 [https://bugs.python.org/issue?@action=redirect&bpo=24982]: shutil.make_archive() with the "zip" format now adds entries for directories (including empty directories) in ZIP file.
- bpo-25019 [https://bugs.python.org/issue?@action=redirect&bpo=25019]: Fixed a crash caused by setting non-string key of expat parser. Based on patch by John Leitch.
- bpo-16180 [https://bugs.python.org/issue?@action=redirect&bpo=16180]: Exit pdb if file has syntax error, instead of trapping user in an infinite loop. Patch by Xavier de Gaye.
- bpo-24891 [https://bugs.python.org/issue?@action=redirect&bpo=24891]: Fix a race condition at Python startup if the file descriptor of stdin (0), stdout (1) or stderr (2) is closed while Python is creating sys.stdin, sys.stdout and sys.stderr objects. These attributes are now set to None if the creation of the object failed, instead of raising an OSError exception. Initial patch written by Marco Paolini.
- bpo-24992 [https://bugs.python.org/issue?@action=redirect&bpo=24992]: Fix error handling and a race condition (related to garbage collection) in collections.OrderedDict constructor.

- bpo-24881 [https://bugs.python.org/issue?@action=redirect&bpo=24881]: Fixed setting binary mode in Python implementation of FileIO on Windows and Cygwin. Patch from Akira Li.
- bpo-25578 [https://bugs.python.org/issue?@action=redirect&bpo=25578]: Fix (another) memory leak in SSLSocket.getpeercer().
- bpo-25530 [https://bugs.python.org/issue?@action=redirect&bpo=25530]: Disable the vulnerable SSLv3 protocol by default when creating ssl.SSLContext.
- bpo-25569 [https://bugs.python.org/issue?@action=redirect&bpo=25569]: Fix memory leak in SSLSocket.getpeercert().
- bpo-25471 [https://bugs.python.org/issue?@action=redirect&bpo=25471]: Sockets returned from accept() shouldn’t appear to be nonblocking.
- bpo-25319 [https://bugs.python.org/issue?@action=redirect&bpo=25319]: When threading.Event is reinitialized, the underlying condition should use a regular lock rather than a recursive lock.
- bpo-21112 [https://bugs.python.org/issue?@action=redirect&bpo=21112]: Fix regression in unittest.expectedFailure on subclasses. Patch from Berker Peksag.
- bpo-24764 [https://bugs.python.org/issue?@action=redirect&bpo=24764]: cgi.FieldStorage.read_multi() now ignores the Content-Length header in part headers. Patch written by Peter Landry and reviewed by Pierre Quentel.
- bpo-24913 [https://bugs.python.org/issue?@action=redirect&bpo=24913]: Fix overrun error in deque.index(). Found by John Leitch and Bryce Darling.
- bpo-24774 [https://bugs.python.org/issue?@action=redirect&bpo=24774]: Fix docstring in http.server.test. Patch from Chiu-Hsiang Hsu.

- bpo-21159 [https://bugs.python.org/issue?@action=redirect&bpo=21159]: Improve message in configparser.InterpolationMissingOptionError. Patch from Łukasz Langa.
- bpo-20362 [https://bugs.python.org/issue?@action=redirect&bpo=20362]: Honour TestCase.longMessage correctly in assertRegex. Patch from Ilia Kurenkov.
- bpo-23572 [https://bugs.python.org/issue?@action=redirect&bpo=23572]: Fixed functools.singledispatch on classes with false metaclasses. Patch by Ethan Furman.
- asyncio: ensure_future() now accepts awaitable objects.

IDLE

- bpo-15348 [https://bugs.python.org/issue?@action=redirect&bpo=15348]: Stop the debugger engine (normally in a user process) before closing the debugger window (running in the IDLE process). This prevents the RuntimeErrors that were being caught and ignored.
- bpo-24455 [https://bugs.python.org/issue?@action=redirect&bpo=24455]: Prevent IDLE from hanging when a) closing the shell while the debugger is active (15347); b) closing the debugger with the [X] button (15348); and c) activating the debugger when already active (24455). The patch by Mark Roseman does this by making two changes. 1. Suspend and resume the gui.interaction method with the tcl vwait mechanism intended for this purpose (instead of root.mainloop & .quit). 2. In gui.run, allow any existing interaction to terminate first.
- Change ‘The program’ to ‘Your program’ in an IDLE ‘kill program?’ message to make it clearer that the program referred to is the currently running user program, not IDLE itself.

- bpo-24750 [https://bugs.python.org/issue?@action=redirect&bpo=24750]: Improve the appearance of the IDLE editor window status bar. Patch by Mark Roseman.
- bpo-25313 [https://bugs.python.org/issue?@action=redirect&bpo=25313]: Change the handling of new built-in text color themes to better address the compatibility problem introduced by the addition of IDLE Dark. Consistently use the revised idleConf.CurrentTheme everywhere in idlelib.
- bpo-24782 [https://bugs.python.org/issue?@action=redirect&bpo=24782]: Extension configuration is now a tab in the IDLE Preferences dialog rather than a separate dialog. The former tabs are now a sorted list. Patch by Mark Roseman.
- bpo-22726 [https://bugs.python.org/issue?@action=redirect&bpo=22726]: Re-activate the config dialog help button with some content about the other buttons and the new IDLE Dark theme.
- bpo-24820 [https://bugs.python.org/issue?@action=redirect&bpo=24820]: IDLE now has an ‘IDLE Dark’ built-in text color theme. It is more or less IDLE Classic inverted, with a cobalt blue background. Strings, comments, keywords, … are still green, red, orange, … . To use it with IDLEs released before November 2015, hit the ‘Save as New Custom Theme’ button and enter a new name, such as ‘Custom Dark’. The custom theme will work with any IDLE release, and can be modified.
- bpo-25224 [https://bugs.python.org/issue?@action=redirect&bpo=25224]: README.txt is now an idlelib index for IDLE developers and curious users. The previous user content is now in the IDLE doc chapter. ‘IDLE’ now means ‘Integrated Development and Learning Environment’.
- bpo-24820 [https://bugs.python.org/issue?@action=redirect&bpo=24820]: Users can now set breakpoint

colors in Settings -> Custom Highlighting. Original patch by Mark Roseman.

- bpo-24972 [https://bugs.python.org/issue?@action=redirect&bpo=24972]: Inactive selection background now matches active selection background, as configured by users, on all systems. Found items are now always highlighted on Windows. Initial patch by Mark Roseman.
- bpo-24570 [https://bugs.python.org/issue?@action=redirect&bpo=24570]: Idle: make calltip and completion boxes appear on Macs affected by a tk regression. Initial patch by Mark Roseman.
- bpo-24988 [https://bugs.python.org/issue?@action=redirect&bpo=24988]: Idle ScrolledList context menus (used in debugger) now work on Mac Aqua. Patch by Mark Roseman.
- bpo-24801 [https://bugs.python.org/issue?@action=redirect&bpo=24801]: Make right-click for context menu work on Mac Aqua. Patch by Mark Roseman.
- bpo-25173 [https://bugs.python.org/issue?@action=redirect&bpo=25173]: Associate tkinter messageboxes with a specific widget. For Mac OSX, make them a ‘sheet’. Patch by Mark Roseman.
- bpo-25198 [https://bugs.python.org/issue?@action=redirect&bpo=25198]: Enhance the initial html viewer now used for Idle Help. Properly indent fixed-pitch text (patch by Mark Roseman). Give code snippet a very Sphinx-like light blueish-gray background. Re-use initial width and height set by users for shell and editor. When the Table of Contents (TOC) menu is used, put the section header at the top of the screen.
- bpo-25225 [https://bugs.python.org/issue?@action=redirect&bpo=25225]: Condense and rewrite Idle doc section on text colors.
- bpo-21995 [https://bugs.python.org/issue?@action=redirect&bpo=21995]: Explain some differences between IDLE and console Python.

- bpo-22820 [https://bugs.python.org/issue?@action=redirect&bpo=22820]: Explain need for *print* when running file from Idle editor.
- bpo-25224 [https://bugs.python.org/issue?@action=redirect&bpo=25224]: Doc: augment Idle feature list and no-subprocess section.
- bpo-25219 [https://bugs.python.org/issue?@action=redirect&bpo=25219]: Update doc for Idle command line options. Some were missing and notes were not correct.
- bpo-24861 [https://bugs.python.org/issue?@action=redirect&bpo=24861]: Most of idlelib is private and subject to change. Use idleib.idle.* to start Idle. See idlelib.__init__.__doc__.
- bpo-25199 [https://bugs.python.org/issue?@action=redirect&bpo=25199]: Idle: add synchronization comments for future maintainers.
- bpo-16893 [https://bugs.python.org/issue?@action=redirect&bpo=16893]: Replace help.txt with help.html for Idle doc display. The new idlelib/help.html is rstripped Doc/build/html/library/idle.html. It looks better than help.txt and will better document Idle as released. The tkinter html viewer that works for this file was written by Mark Roseman. The now unused EditorWindow.HelpDialog class and helt.txt file are deprecated.
- bpo-24199 [https://bugs.python.org/issue?@action=redirect&bpo=24199]: Deprecate unused idlelib.idlever with possible removal in 3.6.
- bpo-24790 [https://bugs.python.org/issue?@action=redirect&bpo=24790]: Remove extraneous code (which also create 2 & 3 conflicts).

Documentation

- bpo-22558 [https://bugs.python.org/issue?@action=redirect&bpo=22558]: Add remaining doc links to source code for Python-coded modules. Patch by Yoni Lavi.
- bpo-12067 [https://bugs.python.org/issue?@action=redirect&bpo=12067]: Rewrite Comparisons section in the Expressions chapter of the language reference. Some of the details of comparing mixed types were incorrect or ambiguous. NotImplemented is only relevant at a lower level than the Expressions chapter. Added details of comparing range() objects, and default behaviour and consistency suggestions for user-defined classes. Patch from Andy Maier.
- bpo-24952 [https://bugs.python.org/issue?@action=redirect&bpo=24952]: Clarify the default size argument of stack_size() in the “threading” and “_thread” modules. Patch from Mattip.
- bpo-23725 [https://bugs.python.org/issue?@action=redirect&bpo=23725]: Overhaul tempfile docs. Note deprecated status of mktemp. Patch from Zbigniew Jędrzejewski-Szmek.
- bpo-24808 [https://bugs.python.org/issue?@action=redirect&bpo=24808]: Update the types of some PyTypeObject fields. Patch by Joseph Weston.
- bpo-22812 [https://bugs.python.org/issue?@action=redirect&bpo=22812]: Fix unittest discovery examples. Patch from Pam McA’Nulty.

Tests

- bpo-25449 [https://bugs.python.org/issue?@action=redirect&bpo=25449]: Added tests for OrderedDict subclasses.
- bpo-25099 [https://bugs.python.org/issue?@action=redirect&bpo=25099]: Make test_compileall not fail

when an entry on sys.path cannot be written to (commonly seen in administrative installs on Windows).

- bpo-23919 [https://bugs.python.org/issue?@action=redirect&bpo=23919]: Prevents assert dialogs appearing in the test suite.
- `PCbuild\rt.bat` now accepts an unlimited number of arguments to pass along to regrtest.py. Previously there was a limit of 9.

Build

- bpo-24915 [https://bugs.python.org/issue?@action=redirect&bpo=24915]: Add LLVM support for PGO builds and use the test suite to generate the profile data. Initial patch by Alecsandru Patrascu of Intel.
- bpo-24910 [https://bugs.python.org/issue?@action=redirect&bpo=24910]: Windows MSIs now have unique display names.
- bpo-24986 [https://bugs.python.org/issue?@action=redirect&bpo=24986]: It is now possible to build Python on Windows without errors when external libraries are not available.

Windows

- bpo-25450 [https://bugs.python.org/issue?@action=redirect&bpo=25450]: Updates shortcuts to start Python in installation directory.
- bpo-25164 [https://bugs.python.org/issue?@action=redirect&bpo=25164]: Changes default all-users install directory to match per-user directory.
- bpo-25143 [https://bugs.python.org/issue?@action=redirect&bpo=25143]: Improves installer error messages for unsupported platforms.
- bpo-25163 [https://bugs.python.org/issue?@action=redirect&bpo=25163]: Display correct directory in

installer when using non-default settings.

- bpo-25361 [https://bugs.python.org/issue?@action=redirect&bpo=25361]: Disables use of SSE2 instructions in Windows 32-bit build
- bpo-25089 [https://bugs.python.org/issue?@action=redirect&bpo=25089]: Adds logging to installer for case where launcher is not selected on upgrade.
- bpo-25165 [https://bugs.python.org/issue?@action=redirect&bpo=25165]: Windows uninstallation should not remove launcher if other versions remain
- bpo-25112 [https://bugs.python.org/issue?@action=redirect&bpo=25112]: py.exe launcher is missing icons
- bpo-25102 [https://bugs.python.org/issue?@action=redirect&bpo=25102]: Windows installer does not precompile for -O or -OO.
- bpo-25081 [https://bugs.python.org/issue?@action=redirect&bpo=25081]: Makes Back button in installer go back to upgrade page when upgrading.
- bpo-25091 [https://bugs.python.org/issue?@action=redirect&bpo=25091]: Increases font size of the installer.
- bpo-25126 [https://bugs.python.org/issue?@action=redirect&bpo=25126]: Clarifies that the non-web installer will download some components.
- bpo-25213 [https://bugs.python.org/issue?@action=redirect&bpo=25213]: Restores requestedExecutionLevel to manifest to disable UAC virtualization.
- bpo-25022 [https://bugs.python.org/issue?@action=redirect&bpo=25022]: Removed very outdated PC/example_nt/ directory.

Tools/Demos

- bpo-25440 [https://bugs.python.org/issue?@action=redirect&bpo=25440]: Fix output of python-config –extension-suffix.

Python 3.5.0 final

Release date: 2015-09-13

Build

- bpo-25071 [https://bugs.python.org/issue?@action=redirect&bpo=25071]: Windows installer should not require TargetDir parameter when installing quietly.

Python 3.5.0 release candidate 4

Release date: 2015-09-09

Library

- bpo-25029 [https://bugs.python.org/issue?@action=redirect&bpo=25029]: Fixes MemoryError in test_strptime.

Build

- bpo-25027 [https://bugs.python.org/issue?@action=redirect&bpo=25027]: Reverts partial-static build options and adds vcruntime140.dll to Windows installation.

Python 3.5.0 release candidate 3

Release date: 2015-09-07

Core and Builtins

- bpo-24305 [https://bugs.python.org/issue?@action=redirect&bpo=24305]: Prevent import subsystem stack frames from being counted by the warnings.warn(stacklevel=) parameter.
- bpo-24912 [https://bugs.python.org/issue?@action=redirect&bpo=24912]: Prevent __class__ assignment to immutable built-in objects.
- bpo-24975 [https://bugs.python.org/issue?@action=redirect&bpo=24975]: Fix AST compilation for **PEP 448** [https://peps.python.org/pep-0448/] syntax.

Library

- bpo-24917 [https://bugs.python.org/issue?@action=redirect&bpo=24917]: time_strftime() buffer over-read.
- bpo-24748 [https://bugs.python.org/issue?@action=redirect&bpo=24748]: To resolve a compatibility problem found with py2exe and pywin32, imp.load_dynamic() once again ignores previously loaded modules to support Python modules replacing themselves with extension modules. Patch by Petr Viktorin.
- bpo-24635 [https://bugs.python.org/issue?@action=redirect&bpo=24635]: Fixed a bug in typing.py where isinstance([], typing.Iterable) would return True once, then False on subsequent calls.
- bpo-24989 [https://bugs.python.org/issue?@action=redirect&bpo=24989]: Fixed buffer overread in BytesIO.readline() if a position is set beyond size. Based on patch by John Leitch.
- bpo-24913 [https://bugs.python.org/issue?@action=redirect&bpo=24913]: Fix overrun error in

deque.index(). Found by John Leitch and Bryce Darling.

Python 3.5.0 release candidate 2

Release date: 2015-08-25

Core and Builtins

- bpo-24769 [https://bugs.python.org/issue?@action=redirect&bpo=24769]: Interpreter now starts properly when dynamic loading is disabled. Patch by Petr Viktorin.
- bpo-21167 [https://bugs.python.org/issue?@action=redirect&bpo=21167]: NAN operations are now handled correctly when python is compiled with ICC even if -fp-model strict is not specified.
- bpo-24492 [https://bugs.python.org/issue?@action=redirect&bpo=24492]: A “package” lacking a __name__ attribute when trying to perform a `from .. import ...` statement will trigger an ImportError instead of an AttributeError.

Library

- bpo-24847 [https://bugs.python.org/issue?@action=redirect&bpo=24847]: Removes vcruntime140.dll dependency from Tcl/Tk.
- bpo-24839 [https://bugs.python.org/issue?@action=redirect&bpo=24839]: platform._syscmd_ver raises DeprecationWarning
- bpo-24867 [https://bugs.python.org/issue?@action=redirect&bpo=24867]: Fix Task.get_stack() for ‘async def’ coroutines

Python 3.5.0 release candidate 1

Release date: 2015-08-09

Core and Builtins

- bpo-24667 [https://bugs.python.org/issue?@action=redirect&bpo=24667]: Resize odict in all cases that the underlying dict resizes.

Library

- bpo-24824 [https://bugs.python.org/issue?@action=redirect&bpo=24824]: Signatures of codecs.encode() and codecs.decode() now are compatible with pydoc.
- bpo-24634 [https://bugs.python.org/issue?@action=redirect&bpo=24634]: Importing uuid should not try to load libc on Windows
- bpo-24798 [https://bugs.python.org/issue?@action=redirect&bpo=24798]: _msvccompiler.py doesn’t properly support manifests
- bpo-4395 [https://bugs.python.org/issue?@action=redirect&bpo=4395]: Better testing and documentation of binary operators. Patch by Martin Panter.
- bpo-23973 [https://bugs.python.org/issue?@action=redirect&bpo=23973]: Update typing.py from GitHub repo.
- bpo-23004 [https://bugs.python.org/issue?@action=redirect&bpo=23004]: mock_open() now reads binary data correctly when the type of read_data is bytes. Initial patch by Aaron Hill.
- bpo-23888 [https://bugs.python.org/issue?@action=redirect&bpo=23888]: Handle fractional time in cookie expiry. Patch by ssh.
- bpo-23652 [https://bugs.python.org/issue?@action=redirect&bpo=23652]: Make it possible to compile the select module against the libc headers from the Linux

Standard Base, which do not include some EPOLL macros. Patch by Matt Frank.

- bpo-22932 [https://bugs.python.org/issue?@action=redirect&bpo=22932]: Fix timezones in email.utils.formatdate. Patch from Dmitry Shachnev.
- bpo-23779 [https://bugs.python.org/issue?@action=redirect&bpo=23779]: imaplib raises TypeError if authenticator tries to abort. Patch from Craig Holmquist.
- bpo-23319 [https://bugs.python.org/issue?@action=redirect&bpo=23319]: Fix ctypes.BigEndianStructure, swap correctly bytes. Patch written by Matthieu Gautier.
- bpo-23254 [https://bugs.python.org/issue?@action=redirect&bpo=23254]: Document how to close the TCPServer listening socket. Patch from Martin Panter.
- bpo-19450 [https://bugs.python.org/issue?@action=redirect&bpo=19450]: Update Windows and OS X installer builds to use SQLite 3.8.11.
- bpo-17527 [https://bugs.python.org/issue?@action=redirect&bpo=17527]: Add PATCH to wsgiref.validator. Patch from Luca Sbardella.
- bpo-24791 [https://bugs.python.org/issue?@action=redirect&bpo=24791]: Fix grammar regression for call syntax: ‘g(*a or b)’.

IDLE

- bpo-23672 [https://bugs.python.org/issue?@action=redirect&bpo=23672]: Allow Idle to edit and run files with astral chars in name. Patch by Mohd Sanad Zaki Rizvi.
- bpo-24745 [https://bugs.python.org/issue?@action=redirect&bpo=24745]: Idle editor default font. Switch from Courier to platform-sensitive TkFixedFont. This should not affect current customized font selections. If there is a problem, edit $HOME/.idlerc/config-main.cfg

and remove '`fontxxx`' entries from [Editor Window]. Patch by Mark Roseman.

- bpo-21192 [https://bugs.python.org/issue?@action=redirect&bpo=21192]: Idle editor. When a file is run, put its name in the restart bar. Do not print false prompts. Original patch by Adnan Umer.
- bpo-13884 [https://bugs.python.org/issue?@action=redirect&bpo=13884]: Idle menus. Remove tearoff lines. Patch by Roger Serwy.

Documentation

- bpo-24129 [https://bugs.python.org/issue?@action=redirect&bpo=24129]: Clarify the reference documentation for name resolution. This includes removing the assumption that readers will be familiar with the name resolution scheme Python used prior to the introduction of lexical scoping for function namespaces. Patch by Ivan Levkivskyi.
- bpo-20769 [https://bugs.python.org/issue?@action=redirect&bpo=20769]: Improve reload() docs. Patch by Dorian Pula.
- bpo-23589 [https://bugs.python.org/issue?@action=redirect&bpo=23589]: Remove duplicate sentence from the FAQ. Patch by Yongzhi Pan.
- bpo-24729 [https://bugs.python.org/issue?@action=redirect&bpo=24729]: Correct IO tutorial to match implementation regarding encoding parameter to open function.

Tests

- bpo-24751 [https://bugs.python.org/issue?@action=redirect&bpo=24751]: When running regrtest with the `-w` command line option, a test run is no longer marked as a failure if all tests succeed when re-run.

Python 3.5.0 beta 4

Release date: 2015-07-26

Core and Builtins

- bpo-23573 [https://bugs.python.org/issue?@action=redirect&bpo=23573]: Restored optimization of bytes.rfind() and bytearray.rfind() for single-byte argument on Linux.
- bpo-24569 [https://bugs.python.org/issue?@action=redirect&bpo=24569]: Make **PEP 448** [https://peps.python.org/pep-0448/] dictionary evaluation more consistent.
- bpo-24583 [https://bugs.python.org/issue?@action=redirect&bpo=24583]: Fix crash when set is mutated while being updated.
- bpo-24407 [https://bugs.python.org/issue?@action=redirect&bpo=24407]: Fix crash when dict is mutated while being updated.
- bpo-24619 [https://bugs.python.org/issue?@action=redirect&bpo=24619]: New approach for tokenizing async/await. As a consequence, it is now possible to have one-line ‘async def foo(): await ..’ functions.
- bpo-24687 [https://bugs.python.org/issue?@action=redirect&bpo=24687]: Plug refleak on SyntaxError in function parameters annotations.
- bpo-15944 [https://bugs.python.org/issue?@action=redirect&bpo=15944]: memoryview: Allow arbitrary formats when casting to bytes. Patch by Martin Panter.

Library

- bpo-23441 [https://bugs.python.org/issue?@action=redirect&bpo=23441]: rcompleter now prints a tab character instead of displaying possible completions for an empty word. Initial patch by Martin Sekera.

- bpo-24683 [https://bugs.python.org/issue?@action=redirect&bpo=24683]: Fixed crashes in _json functions called with arguments of inappropriate type.
- bpo-21697 [https://bugs.python.org/issue?@action=redirect&bpo=21697]: shutil.copytree() now correctly handles symbolic links that point to directories. Patch by Eduardo Seabra and Thomas Kluyver.
- bpo-14373 [https://bugs.python.org/issue?@action=redirect&bpo=14373]: Fixed segmentation fault when gc.collect() is called during constructing lru_cache (C implementation).
- bpo-24695 [https://bugs.python.org/issue?@action=redirect&bpo=24695]: Fix a regression in traceback.print_exception(). If exc_traceback is None we shouldn’t print a traceback header like described in the documentation.
- bpo-24620 [https://bugs.python.org/issue?@action=redirect&bpo=24620]: Random.setstate() now validates the value of state last element.
- bpo-22485 [https://bugs.python.org/issue?@action=redirect&bpo=22485]: Fixed an issue that caused `inspect.getsource` to return incorrect results on nested functions.
- bpo-22153 [https://bugs.python.org/issue?@action=redirect&bpo=22153]: Improve unittest docs. Patch from Martin Panter and evilzero.
- bpo-24580 [https://bugs.python.org/issue?@action=redirect&bpo=24580]: Symbolic group references to open group in re patterns now are explicitly forbidden as well as numeric group references.
- bpo-24206 [https://bugs.python.org/issue?@action=redirect&bpo=24206]: Fixed __eq__ and __ne__ methods of inspect classes.

- bpo-24631 [https://bugs.python.org/issue?@action=redirect&bpo=24631]: Fixed regression in the timeit module with multiline setup.
- bpo-18622 [https://bugs.python.org/issue?@action=redirect&bpo=18622]: unittest.mock.mock_open().reset_mock would recurse infinitely. Patch from Nicola Palumbo and Laurent De Buyst.
- bpo-23661 [https://bugs.python.org/issue?@action=redirect&bpo=23661]: unittest.mock side_effects can now be exceptions again. This was a regression vs Python 3.4. Patch from Ignacio Rossi
- bpo-24608 [https://bugs.python.org/issue?@action=redirect&bpo=24608]: chunk.Chunk.read() now always returns bytes, not str.
- bpo-18684 [https://bugs.python.org/issue?@action=redirect&bpo=18684]: Fixed reading out of the buffer in the re module.
- bpo-24259 [https://bugs.python.org/issue?@action=redirect&bpo=24259]: tarfile now raises a ReadError if an archive is truncated inside a data segment.
- bpo-15014 [https://bugs.python.org/issue?@action=redirect&bpo=15014]: SMTP.auth() and SMTP.login() now support RFC 4954’s optional initial-response argument to the SMTP AUTH command.
- bpo-24669 [https://bugs.python.org/issue?@action=redirect&bpo=24669]: Fix inspect.getsource() for ‘async def’ functions. Patch by Kai Groner.
- bpo-24688 [https://bugs.python.org/issue?@action=redirect&bpo=24688]: ast.get_docstring() for ‘async def’ functions.

Build

- bpo-24603 [https://bugs.python.org/issue?@action=redirect&bpo=24603]: Update Windows builds and OS

X 10.5 installer to use OpenSSL 1.0.2d.

Python 3.5.0 beta 3

Release date: 2015-07-05

Core and Builtins

- bpo-24467 [https://bugs.python.org/issue?@action=redirect&bpo=24467]: Fixed possible buffer over-read in bytearray. The bytearray object now always allocates place for trailing null byte and it’s buffer now is always null-terminated.
- Upgrade to Unicode 8.0.0.
- bpo-24345 [https://bugs.python.org/issue?@action=redirect&bpo=24345]: Add Py_tp_finalize slot for the stable ABI.
- bpo-24400 [https://bugs.python.org/issue?@action=redirect&bpo=24400]: Introduce a distinct type for **PEP 492** [https://peps.python.org/pep-0492/] coroutines; add types.CoroutineType, inspect.getcoroutinestate, inspect.getcoroutinelocals; coroutines no longer use CO_GENERATOR flag; sys.set_coroutine_wrapper works only for ‘async def’ coroutines; inspect.iscoroutine no longer uses collections.abc.Coroutine, it’s intended to test for pure ‘async def’ coroutines only; add new opcode: GET_YIELD_FROM_ITER; fix generators wrapper used in types.coroutine to be instance of collections.abc.Generator; collections.abc.Awaitable and collections.abc.Coroutine can no longer be used to detect generator-based coroutines–use inspect.isawaitable instead.
- bpo-24450 [https://bugs.python.org/issue?@action=redirect&bpo=24450]: Add gi_yieldfrom to generators

and cr_await to coroutines. Contributed by Benno Leslie and Yury Selivanov.

- bpo-19235 [https://bugs.python.org/issue?@action=redirect&bpo=19235]: Add new RecursionError exception. Patch by Georg Brandl.

Library

- bpo-21750 [https://bugs.python.org/issue?@action=redirect&bpo=21750]: mock_open.read_data can now be read from each instance, as it could in Python 3.3.
- bpo-24552 [https://bugs.python.org/issue?@action=redirect&bpo=24552]: Fix use after free in an error case of the _pickle module.
- bpo-24514 [https://bugs.python.org/issue?@action=redirect&bpo=24514]: tarfile now tolerates number fields consisting of only whitespace.
- bpo-19176 [https://bugs.python.org/issue?@action=redirect&bpo=19176]: Fixed doctype() related bugs in C implementation of ElementTree. A deprecation warning no longer issued by XMLParser subclass with default doctype() method. Direct call of doctype() now issues a warning. Parser’s doctype() now is not called if target’s doctype() is called. Based on patch by Martin Panter.
- bpo-20387 [https://bugs.python.org/issue?@action=redirect&bpo=20387]: Restore semantic round-trip correctness in tokenize/untokenize for tab-indented blocks.
- bpo-24456 [https://bugs.python.org/issue?@action=redirect&bpo=24456]: Fixed possible buffer over-read in adpcm2lin() and lin2adpcm() functions of the audioop module.
- bpo-24336 [https://bugs.python.org/issue?@action=redirect&bpo=24336]: The contextmanager decorator

now works with functions with keyword arguments called “func” and “self”. Patch by Martin Panter.

- bpo-24522 [https://bugs.python.org/issue?@action=redirect&bpo=24522]: Fix possible integer overflow in json accelerator module.
- bpo-24489 [https://bugs.python.org/issue?@action=redirect&bpo=24489]: ensure a previously set C errno doesn’t disturb cmath.polar().
- bpo-24408 [https://bugs.python.org/issue?@action=redirect&bpo=24408]: Fixed AttributeError in measure() and metrics() methods of tkinter.Font.
- bpo-14373 [https://bugs.python.org/issue?@action=redirect&bpo=14373]: C implementation of functools.lru_cache() now can be used with methods.
- bpo-24347 [https://bugs.python.org/issue?@action=redirect&bpo=24347]: Set KeyError if PyDict_GetItemWithError returns NULL.
- bpo-24348 [https://bugs.python.org/issue?@action=redirect&bpo=24348]: Drop superfluous incref/decref.
- bpo-24359 [https://bugs.python.org/issue?@action=redirect&bpo=24359]: Check for changed OrderedDict size during iteration.
- bpo-24368 [https://bugs.python.org/issue?@action=redirect&bpo=24368]: Support keyword arguments in OrderedDict methods.
- bpo-24362 [https://bugs.python.org/issue?@action=redirect&bpo=24362]: Simplify the C OrderedDict fast nodes resize logic.
- bpo-24377 [https://bugs.python.org/issue?@action=redirect&bpo=24377]: Fix a ref leak in OrderedDict.__repr__.
- bpo-24369 [https://bugs.python.org/issue?@action=redirect&bpo=24369]: Defend against key-changes during iteration.

Tests

- bpo-24373 [https://bugs.python.org/issue?@action=redirect&bpo=24373]: _testmultiphase and xxlimited now use tp_traverse and tp_finalize to avoid reference leaks encountered when combining tp_dealloc with PyType_FromSpec (see bpo-16690 [https://bugs.python.org/issue?@action=redirect&bpo=16690] for details)

Documentation

- bpo-24458 [https://bugs.python.org/issue?@action=redirect&bpo=24458]: Update documentation to cover multi-phase initialization for extension modules (PEP 489). Patch by Petr Viktorin.
- bpo-24351 [https://bugs.python.org/issue?@action=redirect&bpo=24351]: Clarify what is meant by “identifier” in the context of string.Template instances.

Build

- bpo-24432 [https://bugs.python.org/issue?@action=redirect&bpo=24432]: Update Windows builds and OS X 10.5 installer to use OpenSSL 1.0.2c.

Python 3.5.0 beta 2

Release date: 2015-05-31

Core and Builtins

- bpo-24284 [https://bugs.python.org/issue?@action=redirect&bpo=24284]: The startswith and endswith methods of the str class no longer return True when finding the empty string and the indexes are completely out of range.

- bpo-24115 [https://bugs.python.org/issue?@action=redirect&bpo=24115]: Update uses of PyObject_IsTrue(), PyObject_Not(), PyObject_IsInstance(), PyObject_RichCompareBool() and _PyDict_Contains() to check for and handle errors correctly.
- bpo-24328 [https://bugs.python.org/issue?@action=redirect&bpo=24328]: Fix importing one character extension modules.
- bpo-11205 [https://bugs.python.org/issue?@action=redirect&bpo=11205]: In dictionary displays, evaluate the key before the value.
- bpo-24285 [https://bugs.python.org/issue?@action=redirect&bpo=24285]: Fixed regression that prevented importing extension modules from inside packages. Patch by Petr Viktorin.

Library

- bpo-23247 [https://bugs.python.org/issue?@action=redirect&bpo=23247]: Fix a crash in the StreamWriter.reset() of CJK codecs.
- bpo-24270 [https://bugs.python.org/issue?@action=redirect&bpo=24270]: Add math.isclose() and cmath.isclose() functions as per **PEP 485** [https://peps.python.org/pep-0485/]. Contributed by Chris Barker and Tal Einat.
- bpo-5633 [https://bugs.python.org/issue?@action=redirect&bpo=5633]: Fixed timeit when the statement is a string and the setup is not.
- bpo-24326 [https://bugs.python.org/issue?@action=redirect&bpo=24326]: Fixed audioop.ratecv() with non-default weightB argument. Original patch by David Moore.

- bpo-16991 [https://bugs.python.org/issue?@action=redirect&bpo=16991]: Add a C implementation of OrderedDict.
- bpo-23934 [https://bugs.python.org/issue?@action=redirect&bpo=23934]: Fix inspect.signature to fail correctly for builtin types lacking signature information. Initial patch by James Powell.

Python 3.5.0 beta 1

Release date: 2015-05-24

Core and Builtins

- bpo-24276 [https://bugs.python.org/issue?@action=redirect&bpo=24276]: Fixed optimization of property descriptor getter.
- bpo-24268 [https://bugs.python.org/issue?@action=redirect&bpo=24268]: PEP 489: Multi-phase extension module initialization. Patch by Petr Viktorin.
- bpo-23955 [https://bugs.python.org/issue?@action=redirect&bpo=23955]: Add pyvenv.cfg option to suppress registry/environment lookup for generating sys.path on Windows.
- bpo-24257 [https://bugs.python.org/issue?@action=redirect&bpo=24257]: Fixed system error in the comparison of faked types.SimpleNamespace.
- bpo-22939 [https://bugs.python.org/issue?@action=redirect&bpo=22939]: Fixed integer overflow in iterator object. Patch by Clement Rouault.
- bpo-23985 [https://bugs.python.org/issue?@action=redirect&bpo=23985]: Fix a possible buffer overrun when deleting a slice from the front of a bytearray and then appending some other bytes data.

- bpo-24102 [https://bugs.python.org/issue?@action=redirect&bpo=24102]: Fixed exception type checking in standard error handlers.
- bpo-15027 [https://bugs.python.org/issue?@action=redirect&bpo=15027]: The UTF-32 encoder is now 3x to 7x faster.
- bpo-23290 [https://bugs.python.org/issue?@action=redirect&bpo=23290]: Optimize set_merge() for cases where the target is empty. (Contributed by Serhiy Storchaka.)
- bpo-2292 [https://bugs.python.org/issue?@action=redirect&bpo=2292]: PEP 448: Additional Unpacking Generalizations.
- bpo-24096 [https://bugs.python.org/issue?@action=redirect&bpo=24096]: Make warnings.warn_explicit more robust against mutation of the warnings.filters list.
- bpo-23996 [https://bugs.python.org/issue?@action=redirect&bpo=23996]: Avoid a crash when a delegated generator raises an unnormalized StopIteration exception. Patch by Stefan Behnel.
- bpo-23910 [https://bugs.python.org/issue?@action=redirect&bpo=23910]: Optimize property() getter calls. Patch by Joe Jevnik.
- bpo-23911 [https://bugs.python.org/issue?@action=redirect&bpo=23911]: Move path-based importlib bootstrap code to a separate frozen module.
- bpo-24192 [https://bugs.python.org/issue?@action=redirect&bpo=24192]: Fix namespace package imports.
- bpo-24022 [https://bugs.python.org/issue?@action=redirect&bpo=24022]: Fix tokenizer crash when processing undecodable source code.
- bpo-9951 [https://bugs.python.org/issue?@action=redirect&bpo=9951]: Added a hex() method to bytes, bytearray, and memoryview.
- bpo-22906 [https://bugs.python.org/issue?@action=redirect&bpo=22906]: PEP 479: Change StopIteration

handling inside generators.

- bpo-24017 [https://bugs.python.org/issue?@action=redirect&bpo=24017]: PEP 492: Coroutines with async and await syntax.

Library

- bpo-14373 [https://bugs.python.org/issue?@action=redirect&bpo=14373]: Added C implementation of functools.lru_cache(). Based on patches by Matt Joiner and Alexey Kachayev.
- bpo-24230 [https://bugs.python.org/issue?@action=redirect&bpo=24230]: The tempfile module now accepts bytes for prefix, suffix and dir parameters and returns bytes in such situations (matching the os module APIs).
- bpo-22189 [https://bugs.python.org/issue?@action=redirect&bpo=22189]: collections.UserString now supports __getnewargs__(), __rmod__(), casefold(), format_map(), isprintable(), and maketrans(). Patch by Joe Jevnik.
- bpo-24244 [https://bugs.python.org/issue?@action=redirect&bpo=24244]: Prevents termination when an invalid format string is encountered on Windows in strftime.
- bpo-23973 [https://bugs.python.org/issue?@action=redirect&bpo=23973]: PEP 484: Add the typing module.
- bpo-23086 [https://bugs.python.org/issue?@action=redirect&bpo=23086]: The collections.abc.Sequence() abstract base class added *start* and *stop* parameters to the index() mixin. Patch by Devin Jeanpierre.
- bpo-20035 [https://bugs.python.org/issue?@action=redirect&bpo=20035]: Replaced the `tkinter._fix` module used for setting up the Tcl/Tk environment on

Windows with a private function in the `_tkinter` module that makes no permanent changes to the environment.

- bpo-24257 [https://bugs.python.org/issue?@action=redirect&bpo=24257]: Fixed segmentation fault in sqlite3.Row constructor with faked cursor type.
- bpo-15836 [https://bugs.python.org/issue?@action=redirect&bpo=15836]: assertRaises(), assertRaisesRegex(), assertWarns() and assertWarnsRegex() assertments now check the type of the first argument to prevent possible user error. Based on patch by Daniel Wagner-Hall.
- bpo-9858 [https://bugs.python.org/issue?@action=redirect&bpo=9858]: Add missing method stubs to _io.RawIOBase. Patch by Laura Rupprecht.
- bpo-22955 [https://bugs.python.org/issue?@action=redirect&bpo=22955]: attrgetter, itemgetter and methodcaller objects in the operator module now support pickling. Added readable and evaluable repr for these objects. Based on patch by Josh Rosenberg.
- bpo-22107 [https://bugs.python.org/issue?@action=redirect&bpo=22107]: tempfile.gettempdir() and tempfile.mkdtemp() now try again when a directory with the chosen name already exists on Windows as well as on Unix. tempfile.mkstemp() now fails early if parent directory is not valid (not exists or is a file) on Windows.
- bpo-23780 [https://bugs.python.org/issue?@action=redirect&bpo=23780]: Improved error message in os.path.join() with single argument.
- bpo-6598 [https://bugs.python.org/issue?@action=redirect&bpo=6598]: Increased time precision and random number range in email.utils.make_msgid() to strengthen the uniqueness of the message ID.
- bpo-24091 [https://bugs.python.org/issue?@action=redirect&bpo=24091]: Fixed various crashes in corner cases in C implementation of ElementTree.

- bpo-21931 [https://bugs.python.org/issue?@action=redirect&bpo=21931]: msilib.FCICreate() now raises TypeError in the case of a bad argument instead of a ValueError with a bogus FCI error number. Patch by Jeffrey Armstrong.
- bpo-13866 [https://bugs.python.org/issue?@action=redirect&bpo=13866]: *quote_via* argument added to urllib.parse.urlencode.
- bpo-20098 [https://bugs.python.org/issue?@action=redirect&bpo=20098]: New mangle_from policy option for email, default True for compat32, but False for all other policies.
- bpo-24211 [https://bugs.python.org/issue?@action=redirect&bpo=24211]: The email library now supports RFC 6532: it can generate headers using utf-8 instead of encoded words.
- bpo-16314 [https://bugs.python.org/issue?@action=redirect&bpo=16314]: Added support for the LZMA compression in distutils.
- bpo-21804 [https://bugs.python.org/issue?@action=redirect&bpo=21804]: poplib now supports RFC 6856 (UTF8).
- bpo-18682 [https://bugs.python.org/issue?@action=redirect&bpo=18682]: Optimized pprint functions for builtin scalar types.
- bpo-22027 [https://bugs.python.org/issue?@action=redirect&bpo=22027]: smtplib now supports RFC 6531 (SMTPUTF8).
- bpo-23488 [https://bugs.python.org/issue?@action=redirect&bpo=23488]: Random generator objects now consume 2x less memory on 64-bit.
- bpo-1322 [https://bugs.python.org/issue?@action=redirect&bpo=1322]: platform.dist() and platform.linux_distribution() functions are now deprecated. Initial patch by Vajrasky Kok.

- bpo-22486 [https://bugs.python.org/issue?@action=redirect&bpo=22486]: Added the math.gcd() function. The fractions.gcd() function now is deprecated. Based on patch by Mark Dickinson.
- bpo-24064 [https://bugs.python.org/issue?@action=redirect&bpo=24064]: Property() docstrings are now writeable. (Patch by Berker Peksag.)
- bpo-22681 [https://bugs.python.org/issue?@action=redirect&bpo=22681]: Added support for the koi8_t encoding.
- bpo-22682 [https://bugs.python.org/issue?@action=redirect&bpo=22682]: Added support for the kz1048 encoding.
- bpo-23796 [https://bugs.python.org/issue?@action=redirect&bpo=23796]: peek and read1 methods of BufferedReader now raise ValueError if they called on a closed object. Patch by John Hergenroeder.
- bpo-21795 [https://bugs.python.org/issue?@action=redirect&bpo=21795]: smtpd now supports the 8BITMIME extension whenever the new *decode_data* constructor argument is set to False.
- bpo-24155 [https://bugs.python.org/issue?@action=redirect&bpo=24155]: optimize heapq.heapify() for better cache performance when heapifying large lists.
- bpo-21800 [https://bugs.python.org/issue?@action=redirect&bpo=21800]: imaplib now supports RFC 5161 (enable), RFC 6855 (utf8/internationalized email) and automatically encodes non-ASCII usernames and passwords to UTF8.
- bpo-20274 [https://bugs.python.org/issue?@action=redirect&bpo=20274]: When calling a _sqlite.Connection, it now complains if passed any keyword arguments. Previously it silently ignored them.
- bpo-20274 [https://bugs.python.org/issue?@action=redirect&bpo=20274]: Remove ignored and erroneous

“kwargs” parameters from three METH_VARARGS methods on _sqlite.Connection.

- bpo-24134 [https://bugs.python.org/issue?@action=redirect&bpo=24134]: assertRaises(), assertRaisesRegex(), assertWarns() and assertWarnsRegex() checks now emits a deprecation warning when callable is None or keyword arguments except msg is passed in the context manager mode.
- bpo-24018 [https://bugs.python.org/issue?@action=redirect&bpo=24018]: Add a collections.abc.Generator abstract base class. Contributed by Stefan Behnel.
- bpo-23880 [https://bugs.python.org/issue?@action=redirect&bpo=23880]: Tkinter’s getint() and getdouble() now support Tcl_Obj. Tkinter’s getdouble() now supports any numbers (in particular int).
- bpo-22619 [https://bugs.python.org/issue?@action=redirect&bpo=22619]: Added negative limit support in the traceback module. Based on patch by Dmitry Kazakov.
- bpo-24094 [https://bugs.python.org/issue?@action=redirect&bpo=24094]: Fix possible crash in json.encode with poorly behaved dict subclasses.
- bpo-9246 [https://bugs.python.org/issue?@action=redirect&bpo=9246]: On POSIX, os.getcwd() now supports paths longer than 1025 bytes. Patch written by William Orr.
- bpo-17445 [https://bugs.python.org/issue?@action=redirect&bpo=17445]: add difflib.diff_bytes() to support comparison of byte strings (fixes a regression from Python 2).
- bpo-23917 [https://bugs.python.org/issue?@action=redirect&bpo=23917]: Fall back to sequential compilation when ProcessPoolExecutor doesn’t exist. Patch by Claudiu Popa.
- bpo-23008 [https://bugs.python.org/issue?@action=redirect&bpo=23008]: Fixed resolving attributes with

boolean value is False in pydoc.

- Fix asyncio issue 235: LifoQueue and PriorityQueue’s put didn’t increment unfinished tasks (this bug was introduced when JoinableQueue was merged with Queue).
- bpo-23908 [https://bugs.python.org/issue?@action=redirect&bpo=23908]: os functions now reject paths with embedded null character on Windows instead of silently truncating them.
- bpo-23728 [https://bugs.python.org/issue?@action=redirect&bpo=23728]: binascii.crc_hqx() could return an integer outside of the range 0-0xffff for empty data.
- bpo-23887 [https://bugs.python.org/issue?@action=redirect&bpo=23887]: urllib.error.HTTPError now has a proper repr() representation. Patch by Berker Peksag.
- asyncio: New event loop APIs: set_task_factory() and get_task_factory().
- asyncio: async() function is deprecated in favour of ensure_future().
- bpo-24178 [https://bugs.python.org/issue?@action=redirect&bpo=24178]: asyncio.Lock, Condition, Semaphore, and BoundedSemaphore support new ‘async with’ syntax. Contributed by Yury Selivanov.
- bpo-24179 [https://bugs.python.org/issue?@action=redirect&bpo=24179]: Support ‘async for’ for asyncio.StreamReader. Contributed by Yury Selivanov.
- bpo-24184 [https://bugs.python.org/issue?@action=redirect&bpo=24184]: Add AsyncIterator and AsyncIterable ABCs to collections.abc. Contributed by Yury Selivanov.
- bpo-22547 [https://bugs.python.org/issue?@action=redirect&bpo=22547]: Implement informative __repr__ for inspect.BoundArguments. Contributed by Yury Selivanov.
- bpo-24190 [https://bugs.python.org/issue?@action=redirect&bpo=24190]: Implement

inspect.BoundArgument.apply_defaults() method. Contributed by Yury Selivanov.
- bpo-20691 [https://bugs.python.org/issue?@action=redirect&bpo=20691]: Add ‘follow_wrapped’ argument to inspect.Signature.from_callable() and inspect.signature(). Contributed by Yury Selivanov.
- bpo-24248 [https://bugs.python.org/issue?@action=redirect&bpo=24248]: Deprecate inspect.Signature.from_function() and inspect.Signature.from_builtin().
- bpo-23898 [https://bugs.python.org/issue?@action=redirect&bpo=23898]: Fix inspect.classify_class_attrs() to support attributes with overloaded __eq__ and __bool__. Patch by Mike Bayer.
- bpo-24298 [https://bugs.python.org/issue?@action=redirect&bpo=24298]: Fix inspect.signature() to correctly unwrap wrappers around bound methods.

IDLE

- bpo-23184 [https://bugs.python.org/issue?@action=redirect&bpo=23184]: remove unused names and imports in idlelib. Initial patch by Al Sweigart.

Tests

- bpo-21520 [https://bugs.python.org/issue?@action=redirect&bpo=21520]: test_zipfile no longer fails if the word ‘bad’ appears anywhere in the name of the current directory.
- bpo-9517 [https://bugs.python.org/issue?@action=redirect&bpo=9517]: Move script_helper into the support package. Patch by Christie Wilson.

Documentation

- bpo-22155 [https://bugs.python.org/issue?@action=redirect&bpo=22155]: Add File Handlers subsection with createfilehandler to tkinter doc. Remove obsolete example from FAQ. Patch by Martin Panter.
- bpo-24029 [https://bugs.python.org/issue?@action=redirect&bpo=24029]: Document the name binding behavior for submodule imports.
- bpo-24077 [https://bugs.python.org/issue?@action=redirect&bpo=24077]: Fix typo in man page for -I command option: -s, not -S

Tools/Demos

- bpo-24000 [https://bugs.python.org/issue?@action=redirect&bpo=24000]: Improved Argument Clinic's mapping of converters to legacy "format units". Updated the documentation to match.
- bpo-24001 [https://bugs.python.org/issue?@action=redirect&bpo=24001]: Argument Clinic converters now use accept={type} instead of types={'type'} to specify the types the converter accepts.
- bpo-23330 [https://bugs.python.org/issue?@action=redirect&bpo=23330]: h2py now supports arbitrary filenames in #include.
- bpo-24031 [https://bugs.python.org/issue?@action=redirect&bpo=24031]: make patchcheck now supports git checkouts, too.

Python 3.5.0 alpha 4

Release date: 2015-04-19

Core and Builtins

- bpo-22980 [https://bugs.python.org/issue?@action=redirect&bpo=22980]: Under Linux, GNU/KFreeBSD and the Hurd, C extensions now include the architecture triplet in the extension name, to make it easy to test builds for different ABIs in the same working tree. Under OS X, the extension name now includes **PEP 3149** [https://peps.python.org/pep-3149/]-style information.
- bpo-22631 [https://bugs.python.org/issue?@action=redirect&bpo=22631]: Added Linux-specific socket constant CAN_RAW_FD_FRAMES. Patch courtesy of Joe Jevnik.
- bpo-23731 [https://bugs.python.org/issue?@action=redirect&bpo=23731]: Implement **PEP 488** [https://peps.python.org/pep-0488/]: removal of .pyo files.
- bpo-23726 [https://bugs.python.org/issue?@action=redirect&bpo=23726]: Don’t enable GC for user subclasses of non-GC types that don’t add any new fields. Patch by Eugene Toder.
- bpo-23309 [https://bugs.python.org/issue?@action=redirect&bpo=23309]: Avoid a deadlock at shutdown if a daemon thread is aborted while it is holding a lock to a buffered I/O object, and the main thread tries to use the same I/O object (typically stdout or stderr). A fatal error is emitted instead.
- bpo-22977 [https://bugs.python.org/issue?@action=redirect&bpo=22977]: Fixed formatting Windows error messages on Wine. Patch by Martin Panter.
- bpo-23466 [https://bugs.python.org/issue?@action=redirect&bpo=23466]: %c, %o, %x, and %X in bytes formatting now raise TypeError on non-integer input.
- bpo-24044 [https://bugs.python.org/issue?@action=redirect&bpo=24044]: Fix possible null pointer dereference in list.sort in out of memory conditions.
- bpo-21354 [https://bugs.python.org/issue?@action=redirect&bpo=21354]: PyCFunction_New function is exposed by python DLL again.

Library

- bpo-23840 [https://bugs.python.org/issue?@action=redirect&bpo=23840]: tokenize.open() now closes the temporary binary file on error to fix a resource warning.
- bpo-16914 [https://bugs.python.org/issue?@action=redirect&bpo=16914]: new debuglevel 2 in smtplib adds timestamps to debug output.
- bpo-7159 [https://bugs.python.org/issue?@action=redirect&bpo=7159]: urllib.request now supports sending auth credentials automatically after the first 401. This enhancement is a superset of the enhancement from bpo-19494 [https://bugs.python.org/issue?@action=redirect&bpo=19494] and supersedes that change.
- bpo-23703 [https://bugs.python.org/issue?@action=redirect&bpo=23703]: Fix a regression in urljoin() introduced in 901e4e52b20a. Patch by Demian Brecht.
- bpo-4254 [https://bugs.python.org/issue?@action=redirect&bpo=4254]: Adds _curses.update_lines_cols(). Patch by Arnon Yaari
- bpo-19933 [https://bugs.python.org/issue?@action=redirect&bpo=19933]: Provide default argument for ndigits in round. Patch by Vajrasky Kok.
- bpo-23193 [https://bugs.python.org/issue?@action=redirect&bpo=23193]: Add a numeric_owner parameter to tarfile.TarFile.extract and tarfile.TarFile.extractall. Patch by Michael Vogt and Eric Smith.
- bpo-23342 [https://bugs.python.org/issue?@action=redirect&bpo=23342]: Add a subprocess.run() function than returns a CalledProcess instance for a more consistent API than the existing call* functions.
- bpo-21217 [https://bugs.python.org/issue?@action=redirect&bpo=21217]: inspect.getsourcelines() now tries to compute the start and end lines from the code object, fixing an issue when a lambda function is used

as decorator argument. Patch by Thomas Ballinger and Allison Kaptur.

- bpo-24521 [https://bugs.python.org/issue?@action=redirect&bpo=24521]: Fix possible integer overflows in the pickle module.
- bpo-22931 [https://bugs.python.org/issue?@action=redirect&bpo=22931]: Allow ‘[’ and ‘]’ in cookie values.
- The keywords attribute of functools.partial is now always a dictionary.
- bpo-23811 [https://bugs.python.org/issue?@action=redirect&bpo=23811]: Add missing newline to the PyCompileError error message. Patch by Alex Shkop.
- bpo-21116 [https://bugs.python.org/issue?@action=redirect&bpo=21116]: Avoid blowing memory when allocating a multiprocessing shared array that’s larger than 50% of the available RAM. Patch by Médéric Boquien.
- bpo-22982 [https://bugs.python.org/issue?@action=redirect&bpo=22982]: Improve BOM handling when seeking to multiple positions of a writable text file.
- bpo-23464 [https://bugs.python.org/issue?@action=redirect&bpo=23464]: Removed deprecated asyncio JoinableQueue.
- bpo-23529 [https://bugs.python.org/issue?@action=redirect&bpo=23529]: Limit the size of decompressed data when reading from GzipFile, BZ2File or LZMAFile. This defeats denial of service attacks using compressed bombs (i.e. compressed payloads which decompress to a huge size). Patch by Martin Panter and Nikolaus Rath.
- bpo-21859 [https://bugs.python.org/issue?@action=redirect&bpo=21859]: Added Python implementation of io.FileIO.
- bpo-23865 [https://bugs.python.org/issue?@action=redirect&bpo=23865]: close() methods in multiple modules now are idempotent and more robust at

shutdown. If they need to release multiple resources, all are released even if errors occur.

- bpo-23400 [https://bugs.python.org/issue?@action=redirect&bpo=23400]: Raise same exception on both Python 2 and 3 if sem_open is not available. Patch by Davin Potts.
- bpo-10838 [https://bugs.python.org/issue?@action=redirect&bpo=10838]: The subprocess now module includes SubprocessError and TimeoutError in its list of exported names for the users wild enough to use `from subprocess import *`.
- bpo-23411 [https://bugs.python.org/issue?@action=redirect&bpo=23411]: Added DefragResult, ParseResult, SplitResult, DefragResultBytes, ParseResultBytes, and SplitResultBytes to urllib.parse.__all__. Patch by Martin Panter.
- bpo-23881 [https://bugs.python.org/issue?@action=redirect&bpo=23881]: urllib.request.ftpwrapper constructor now closes the socket if the FTP connection failed to fix a ResourceWarning.
- bpo-23853 [https://bugs.python.org/issue?@action=redirect&bpo=23853]: **`socket.socket.sendall()`** does no more reset the socket timeout each time data is sent successfully. The socket timeout is now the maximum total duration to send all data.
- bpo-22721 [https://bugs.python.org/issue?@action=redirect&bpo=22721]: An order of multiline pprint output of set or dict containing orderable and non-orderable elements no longer depends on iteration order of set or dict.
- bpo-15133 [https://bugs.python.org/issue?@action=redirect&bpo=15133]: _tkinter.tkapp.getboolean() now supports Tcl_Obj and always returns bool. tkinter.BooleanVar now validates input values (accepted bool, int, str, and Tcl_Obj). tkinter.BooleanVar.get() now always returns bool.

- bpo-10590 [https://bugs.python.org/issue?@action=redirect&bpo=10590]: xml.sax.parseString() now supports string argument.
- bpo-23338 [https://bugs.python.org/issue?@action=redirect&bpo=23338]: Fixed formatting ctypes error messages on Cygwin. Patch by Makoto Kato.
- bpo-15582 [https://bugs.python.org/issue?@action=redirect&bpo=15582]: inspect.getdoc() now follows inheritance chains.
- bpo-2175 [https://bugs.python.org/issue?@action=redirect&bpo=2175]: SAX parsers now support a character stream of InputSource object.
- bpo-16840 [https://bugs.python.org/issue?@action=redirect&bpo=16840]: Tkinter now supports 64-bit integers added in Tcl 8.4 and arbitrary precision integers added in Tcl 8.5.
- bpo-23834 [https://bugs.python.org/issue?@action=redirect&bpo=23834]: Fix socket.sendto(), use the C Py_ssize_t type to store the result of sendto() instead of the C int type.
- bpo-23618 [https://bugs.python.org/issue?@action=redirect&bpo=23618]: **`socket.socket.connect()`** now waits until the connection completes instead of raising **`InterruptedError`** if the connection is interrupted by signals, signal handlers don’t raise an exception and the socket is blocking or has a timeout. **`socket.socket.connect()`** still raise **`InterruptedError`** for non-blocking sockets.
- bpo-21526 [https://bugs.python.org/issue?@action=redirect&bpo=21526]: Tkinter now supports new boolean type in Tcl 8.5.
- bpo-23836 [https://bugs.python.org/issue?@action=redirect&bpo=23836]: Fix the faulthandler module to handle reentrant calls to its signal handlers.
- bpo-23838 [https://bugs.python.org/issue?@action=redirect&bpo=23838]: linecache now clears the cache

and returns an empty result on MemoryError.

- bpo-10395 [https://bugs.python.org/issue?@action=redirect&bpo=10395]: Added os.path.commonpath(). Implemented in posixpath and ntpath. Based on patch by Rafik Draoui.
- bpo-23611 [https://bugs.python.org/issue?@action=redirect&bpo=23611]: Serializing more “lookupable” objects (such as unbound methods or nested classes) now are supported with pickle protocols < 4.
- bpo-13583 [https://bugs.python.org/issue?@action=redirect&bpo=13583]: sqlite3.Row now supports slice indexing.
- bpo-18473 [https://bugs.python.org/issue?@action=redirect&bpo=18473]: Fixed 2to3 and 3to2 compatible pickle mappings. Fixed ambiguous reverse mappings. Added many new mappings. Import mapping is no longer applied to modules already mapped with full name mapping.
- bpo-23485 [https://bugs.python.org/issue?@action=redirect&bpo=23485]: select.select() is now retried automatically with the recomputed timeout when interrupted by a signal, except if the signal handler raises an exception. This change is part of the **PEP 475** [https://peps.python.org/pep-0475/].
- bpo-23752 [https://bugs.python.org/issue?@action=redirect&bpo=23752]: When built from an existing file descriptor, io.FileIO() now only calls fstat() once. Before fstat() was called twice, which was not necessary.
- bpo-23704 [https://bugs.python.org/issue?@action=redirect&bpo=23704]: collections.deque() objects now support __add__, __mul__, and __imul__().
- bpo-23171 [https://bugs.python.org/issue?@action=redirect&bpo=23171]: csv.Writer.writerow() now supports arbitrary iterables.
- bpo-23745 [https://bugs.python.org/issue?@action=redirect&bpo=23745]: The new email header parser

now handles duplicate MIME parameter names without error, similar to how get_param behaves.

- bpo-22117 [https://bugs.python.org/issue?@action=redirect&bpo=22117]: Fix os.utime(), it now rounds the timestamp towards minus infinity (-inf) instead of rounding towards zero.
- bpo-23310 [https://bugs.python.org/issue?@action=redirect&bpo=23310]: Fix MagicMock’s initializer to work with __methods__, just like configure_mock(). Patch by Kasia Jachim.

Build

- bpo-23817 [https://bugs.python.org/issue?@action=redirect&bpo=23817]: FreeBSD now uses “1.0” in the SOVERSION as other operating systems, instead of just “1”.
- bpo-23501 [https://bugs.python.org/issue?@action=redirect&bpo=23501]: Argument Clinic now generates code into separate files by default.

Tests

- bpo-23799 [https://bugs.python.org/issue?@action=redirect&bpo=23799]: Added test.support.start_threads() for running and cleaning up multiple threads.
- bpo-22390 [https://bugs.python.org/issue?@action=redirect&bpo=22390]: test.regrtest now emits a warning if temporary files or directories are left after running a test.

Tools/Demos

- bpo-18128 [https://bugs.python.org/issue?@action=redirect&bpo=18128]: pygettext now uses standard

+NNNN format in the POT-Creation-Date header.
- bpo-23935 [https://bugs.python.org/issue?@action=redirect&bpo=23935]: Argument Clinic's understanding of format units accepting bytes, bytearrays, and buffers is now consistent with both the documentation and the implementation.
- bpo-23944 [https://bugs.python.org/issue?@action=redirect&bpo=23944]: Argument Clinic now wraps long impl prototypes at column 78.
- bpo-20586 [https://bugs.python.org/issue?@action=redirect&bpo=20586]: Argument Clinic now ensures that functions without docstrings have signatures.
- bpo-23492 [https://bugs.python.org/issue?@action=redirect&bpo=23492]: Argument Clinic now generates argument parsing code with PyArg_Parse instead of PyArg_ParseTuple if possible.
- bpo-23500 [https://bugs.python.org/issue?@action=redirect&bpo=23500]: Argument Clinic is now smarter about generating the "#ifndef" (empty) definition of the methoddef macro: it's only generated once, even if Argument Clinic processes the same symbol multiple times, and it's emitted at the end of all processing rather than immediately after the first use.

C API

- bpo-23998 [https://bugs.python.org/issue?@action=redirect&bpo=23998]: PyImport_ReInitLock() now checks for lock allocation error

Python 3.5.0 alpha 3

Release date: 2015-03-28

Core and Builtins

- bpo-23573 [https://bugs.python.org/issue?@action=redirect&bpo=23573]: Increased performance of string search operations (str.find, str.index, str.count, the in operator, str.split, str.partition) with arguments of different kinds (UCS1, UCS2, UCS4).
- bpo-23753 [https://bugs.python.org/issue?@action=redirect&bpo=23753]: Python doesn’t support anymore platforms without stat() or fstat(), these functions are always required.
- bpo-23681 [https://bugs.python.org/issue?@action=redirect&bpo=23681]: The -b option now affects comparisons of bytes with int.
- bpo-23632 [https://bugs.python.org/issue?@action=redirect&bpo=23632]: Memoryviews now allow tuple indexing (including for multi-dimensional memoryviews).
- bpo-23192 [https://bugs.python.org/issue?@action=redirect&bpo=23192]: Fixed generator lambdas. Patch by Bruno Cauet.
- bpo-23629 [https://bugs.python.org/issue?@action=redirect&bpo=23629]: Fix the default __sizeof__ implementation for variable-sized objects.

Library

- bpo-14260 [https://bugs.python.org/issue?@action=redirect&bpo=14260]: The groupindex attribute of regular expression pattern object now is non-modifiable mapping.
- bpo-23792 [https://bugs.python.org/issue?@action=redirect&bpo=23792]: Ignore KeyboardInterrupt when the pydoc pager is active. This mimics the behavior of the standard unix pagers, and prevents pipepager from shutting down while the pager itself is still running.
- bpo-23775 [https://bugs.python.org/issue?@action=redirect&bpo=23775]: pprint() of OrderedDict now

outputs the same representation as repr().

- bpo-23765 [https://bugs.python.org/issue?@action=redirect&bpo=23765]: Removed IsBadStringPtr calls in ctypes
- bpo-22364 [https://bugs.python.org/issue?@action=redirect&bpo=22364]: Improved some re error messages using regex for hints.
- bpo-23742 [https://bugs.python.org/issue?@action=redirect&bpo=23742]: ntpath.expandvars() no longer loses unbalanced single quotes.
- bpo-21717 [https://bugs.python.org/issue?@action=redirect&bpo=21717]: The zipfile.ZipFile.open function now supports ‘x’ (exclusive creation) mode.
- bpo-21802 [https://bugs.python.org/issue?@action=redirect&bpo=21802]: The reader in BufferedRWPair now is closed even when closing writer failed in BufferedRWPair.close().
- bpo-23622 [https://bugs.python.org/issue?@action=redirect&bpo=23622]: Unknown escapes in regular expressions that consist of `'\'` and ASCII letter now raise a deprecation warning and will be forbidden in Python 3.6.
- bpo-23671 [https://bugs.python.org/issue?@action=redirect&bpo=23671]: string.Template now allows specifying the “self” parameter as a keyword argument. string.Formatter now allows specifying the “self” and the “format_string” parameters as keyword arguments.
- bpo-23502 [https://bugs.python.org/issue?@action=redirect&bpo=23502]: The pprint module now supports mapping proxies.
- bpo-17530 [https://bugs.python.org/issue?@action=redirect&bpo=17530]: pprint now wraps long bytes objects and bytearrays.
- bpo-22687 [https://bugs.python.org/issue?@action=redirect&bpo=22687]: Fixed some corner cases in

breaking words in tetxtwrap. Got rid of quadratic complexity in breaking long words.

- bpo-4727 [https://bugs.python.org/issue?@action=redirect&bpo=4727]: The copy module now uses pickle protocol 4 (PEP 3154) and supports copying of instances of classes whose __new__ method takes keyword-only arguments.
- bpo-23491 [https://bugs.python.org/issue?@action=redirect&bpo=23491]: Added a zipapp module to support creating executable zip file archives of Python code. Registered ".pyz" and ".pyzw" extensions on Windows for these archives (PEP 441).
- bpo-23657 [https://bugs.python.org/issue?@action=redirect&bpo=23657]: Avoid explicit checks for str in zipapp, adding support for pathlib.Path objects as arguments.
- bpo-23688 [https://bugs.python.org/issue?@action=redirect&bpo=23688]: Added support of arbitrary bytes-like objects and avoided unnecessary copying of memoryview in gzip.GzipFile.write(). Original patch by Wolfgang Maier.
- bpo-23252 [https://bugs.python.org/issue?@action=redirect&bpo=23252]: Added support for writing ZIP files to unseekable streams.
- bpo-23647 [https://bugs.python.org/issue?@action=redirect&bpo=23647]: Increase imaplib's MAXLINE to accommodate modern mailbox sizes.
- bpo-23539 [https://bugs.python.org/issue?@action=redirect&bpo=23539]: If body is None, http.client.HTTPConnection.request now sets Content-Length to 0 for PUT, POST, and PATCH headers to avoid 411 errors from some web servers.
- bpo-22351 [https://bugs.python.org/issue?@action=redirect&bpo=22351]: The nntplib.NNTP constructor no longer leaves the connection and socket open until the garbage collector cleans them up. Patch by Martin Panter.

- bpo-23704 [https://bugs.python.org/issue?@action=redirect&bpo=23704]: collections.deque() objects now support methods for index(), insert(), and copy(). This allows deques to be registered as a MutableSequence and it improves their substitutability for lists.
- bpo-23715 [https://bugs.python.org/issue?@action=redirect&bpo=23715]: **`signal.sigwaitinfo()`** and **`signal.sigtimedwait()`** are now retried when interrupted by a signal not in the *sigset* parameter, if the signal handler does not raise an exception. signal.sigtimedwait() recomputes the timeout with a monotonic clock when it is retried.
- bpo-23001 [https://bugs.python.org/issue?@action=redirect&bpo=23001]: Few functions in modules mmap, ossaudiodev, socket, ssl, and codecs, that accepted only read-only bytes-like object now accept writable bytes-like object too.
- bpo-23646 [https://bugs.python.org/issue?@action=redirect&bpo=23646]: If time.sleep() is interrupted by a signal, the sleep is now retried with the recomputed delay, except if the signal handler raises an exception (PEP 475).
- bpo-23136 [https://bugs.python.org/issue?@action=redirect&bpo=23136]: _strptime now uniformly handles all days in week 0, including Dec 30 of previous year. Based on patch by Jim Carroll.
- bpo-23700 [https://bugs.python.org/issue?@action=redirect&bpo=23700]: Iterator of NamedTemporaryFile now keeps a reference to NamedTemporaryFile instance. Patch by Bohuslav Kabrda.
- bpo-22903 [https://bugs.python.org/issue?@action=redirect&bpo=22903]: The fake test case created by unittest.loader when it fails importing a test module is now picklable.
- bpo-22181 [https://bugs.python.org/issue?@action=redirect&bpo=22181]: On Linux, os.urandom() now

uses the new getrandom() syscall if available, syscall introduced in the Linux kernel 3.17. It is more reliable and more secure, because it avoids the need of a file descriptor and waits until the kernel has enough entropy.

- bpo-2211 [https://bugs.python.org/issue?@action=redirect&bpo=2211]: Updated the implementation of the http.cookies.Morsel class. Setting attributes key, value and coded_value directly now is deprecated. update() and setdefault() now transform and check keys. Comparing for equality now takes into account attributes key, value and coded_value. copy() now returns a Morsel, not a dict. repr() now contains all attributes. Optimized checking keys and quoting values. Added new tests. Original patch by Demian Brecht.
- bpo-18983 [https://bugs.python.org/issue?@action=redirect&bpo=18983]: Allow selection of output units in timeit. Patch by Julian Gindi.
- bpo-23631 [https://bugs.python.org/issue?@action=redirect&bpo=23631]: Fix traceback.format_list when a traceback has been mutated.
- bpo-23568 [https://bugs.python.org/issue?@action=redirect&bpo=23568]: Add rdivmod support to MagicMock() objects. Patch by Håkan Lövdahl.
- bpo-2052 [https://bugs.python.org/issue?@action=redirect&bpo=2052]: Add charset parameter to HtmlDiff.make_file().
- bpo-23668 [https://bugs.python.org/issue?@action=redirect&bpo=23668]: Support os.truncate and os.ftruncate on Windows.
- bpo-23138 [https://bugs.python.org/issue?@action=redirect&bpo=23138]: Fixed parsing cookies with absent keys or values in cookiejar. Patch by Demian Brecht.
- bpo-23051 [https://bugs.python.org/issue?@action=redirect&bpo=23051]: multiprocessing.Pool methods imap() and imap_unordered() now handle exceptions

raised by an iterator. Patch by Alon Diamant and Davin Potts.

- bpo-23581 [https://bugs.python.org/issue?@action=redirect&bpo=23581]: Add matmul support to MagicMock. Patch by Håkan Lövdahl.
- bpo-23566 [https://bugs.python.org/issue?@action=redirect&bpo=23566]: enable(), register(), dump_traceback() and dump_traceback_later() functions of faulthandler now accept file descriptors. Patch by Wei Wu.
- bpo-22928 [https://bugs.python.org/issue?@action=redirect&bpo=22928]: Disabled HTTP header injections in http.client. Original patch by Demian Brecht.
- bpo-23615 [https://bugs.python.org/issue?@action=redirect&bpo=23615]: Modules bz2, tarfile and tokenize now can be reloaded with imp.reload(). Patch by Thomas Kluyver.
- bpo-23605 [https://bugs.python.org/issue?@action=redirect&bpo=23605]: os.walk() now calls os.scandir() instead of os.listdir(). The usage of os.scandir() reduces the number of calls to os.stat(). Initial patch written by Ben Hoyt.

Build

- bpo-23585 [https://bugs.python.org/issue?@action=redirect&bpo=23585]: make patchcheck will ensure the interpreter is built.

Tests

- bpo-23583 [https://bugs.python.org/issue?@action=redirect&bpo=23583]: Added tests for standard IO streams in IDLE.

- bpo-22289 [https://bugs.python.org/issue?@action=redirect&bpo=22289]: Prevent test_urllib2net failures due to ftp connection timeout.

Tools/Demos

- bpo-22826 [https://bugs.python.org/issue?@action=redirect&bpo=22826]: The result of open() in Tools/freeze/bkfile.py is now better compatible with regular files (in particular it now supports the context management protocol).

Python 3.5.0 alpha 2

Release date: 2015-03-09

Core and Builtins

- bpo-23571 [https://bugs.python.org/issue?@action=redirect&bpo=23571]: PyObject_Call() and PyCFunction_Call() now raise a SystemError if a function returns a result and raises an exception. The SystemError is chained to the previous exception.

Library

- bpo-22524 [https://bugs.python.org/issue?@action=redirect&bpo=22524]: New os.scandir() function, part of the **PEP 471** [https://peps.python.org/pep-0471/]: “os.scandir() function – a better and faster directory iterator”. Patch written by Ben Hoyt.
- bpo-23103 [https://bugs.python.org/issue?@action=redirect&bpo=23103]: Reduced the memory consumption of IPv4Address and IPv6Address.

- bpo-21793 [https://bugs.python.org/issue?@action=redirect&bpo=21793]: BaseHTTPRequestHandler again logs response code as numeric, not as stringified enum. Patch by Demian Brecht.
- bpo-23476 [https://bugs.python.org/issue?@action=redirect&bpo=23476]: In the ssl module, enable OpenSSL’s X509_V_FLAG_TRUSTED_FIRST flag on certificate stores when it is available.
- bpo-23576 [https://bugs.python.org/issue?@action=redirect&bpo=23576]: Avoid stalling in SSL reads when EOF has been reached in the SSL layer but the underlying connection hasn’t been closed.
- bpo-23504 [https://bugs.python.org/issue?@action=redirect&bpo=23504]: Added an __all__ to the types module.
- bpo-23563 [https://bugs.python.org/issue?@action=redirect&bpo=23563]: Optimized utility functions in urllib.parse.
- bpo-7830 [https://bugs.python.org/issue?@action=redirect&bpo=7830]: Flatten nested functools.partial.
- bpo-20204 [https://bugs.python.org/issue?@action=redirect&bpo=20204]: Added the __module__ attribute to _tkinter classes.
- bpo-19980 [https://bugs.python.org/issue?@action=redirect&bpo=19980]: Improved help() for non-recognized strings. help(‘’) now shows the help on str. help(‘help’) now shows the help on help(). Original patch by Mark Lawrence.
- bpo-23521 [https://bugs.python.org/issue?@action=redirect&bpo=23521]: Corrected pure python implementation of timedelta division. Eliminated OverflowError from `timedelta * float` for some floats; Corrected rounding in timedelta true division.
- bpo-21619 [https://bugs.python.org/issue?@action=redirect&bpo=21619]: Popen objects no longer leave

a zombie after exit in the with statement if the pipe was broken. Patch by Martin Panter.

- bpo-22936 [https://bugs.python.org/issue?@action=redirect&bpo=22936]: Make it possible to show local variables in tracebacks for both the traceback module and unittest.
- bpo-15955 [https://bugs.python.org/issue?@action=redirect&bpo=15955]: Add an option to limit the output size in bz2.decompress(). Patch by Nikolaus Rath.
- bpo-6639 [https://bugs.python.org/issue?@action=redirect&bpo=6639]: Module-level turtle functions no longer raise TclError after closing the window.
- bpo-814253 [https://bugs.python.org/issue?@action=redirect&bpo=814253]: Group references and conditional group references now work in lookbehind assertions in regular expressions. (See also: bpo-9179 [https://bugs.python.org/issue?@action=redirect&bpo=9179])
- bpo-23215 [https://bugs.python.org/issue?@action=redirect&bpo=23215]: Multibyte codecs with custom error handlers that ignores errors consumed too much memory and raised SystemError or MemoryError. Original patch by Aleksi Torhamo.
- bpo-5700 [https://bugs.python.org/issue?@action=redirect&bpo=5700]: io.FileIO() called flush() after closing the file. flush() was not called in close() if closefd=False.
- bpo-23374 [https://bugs.python.org/issue?@action=redirect&bpo=23374]: Fixed pydoc failure with non-ASCII files when stdout encoding differs from file system encoding (e.g. on Mac OS).
- bpo-23481 [https://bugs.python.org/issue?@action=redirect&bpo=23481]: Remove RC4 from the SSL module’s default cipher list.
- bpo-21548 [https://bugs.python.org/issue?@action=redirect&bpo=21548]: Fix pydoc.synopsis() and pydoc.apropos() on modules with empty docstrings.

- bpo-22885 [https://bugs.python.org/issue?@action=redirect&bpo=22885]: Fixed arbitrary code execution vulnerability in the dbm.dumb module. Original patch by Claudiu Popa.
- bpo-23239 [https://bugs.python.org/issue?@action=redirect&bpo=23239]: ssl.match_hostname() now supports matching of IP addresses.
- bpo-23146 [https://bugs.python.org/issue?@action=redirect&bpo=23146]: Fix mishandling of absolute Windows paths with forward slashes in pathlib.
- bpo-23096 [https://bugs.python.org/issue?@action=redirect&bpo=23096]: Pickle representation of floats with protocol 0 now is the same for both Python and C implementations.
- bpo-19105 [https://bugs.python.org/issue?@action=redirect&bpo=19105]: pprint now more efficiently uses free space at the right.
- bpo-14910 [https://bugs.python.org/issue?@action=redirect&bpo=14910]: Add allow_abbrev parameter to argparse.ArgumentParser. Patch by Jonathan Paugh, Steven Bethard, paul j3 and Daniel Eriksson.
- bpo-21717 [https://bugs.python.org/issue?@action=redirect&bpo=21717]: tarfile.open() now supports ‘x’ (exclusive creation) mode.
- bpo-23344 [https://bugs.python.org/issue?@action=redirect&bpo=23344]: marshal.dumps() is now 20-25% faster on average.
- bpo-20416 [https://bugs.python.org/issue?@action=redirect&bpo=20416]: marshal.dumps() with protocols 3 and 4 is now 40-50% faster on average.
- bpo-23421 [https://bugs.python.org/issue?@action=redirect&bpo=23421]: Fixed compression in tarfile CLI. Patch by wdv4758h.
- bpo-23367 [https://bugs.python.org/issue?@action=redirect&bpo=23367]: Fix possible overflows in the unicodedata module.

- bpo-23361 [https://bugs.python.org/issue?@action=redirect&bpo=23361]: Fix possible overflow in Windows subprocess creation code.
- logging.handlers.QueueListener now takes a respect_handler_level keyword argument which, if set to True, will pass messages to handlers taking handler levels into account.
- bpo-19705 [https://bugs.python.org/issue?@action=redirect&bpo=19705]: turtledemo now has a visual sorting algorithm demo. Original patch from Jason Yeo.
- bpo-23801 [https://bugs.python.org/issue?@action=redirect&bpo=23801]: Fix issue where cgi.FieldStorage did not always ignore the entire preamble to a multipart body.

Build

- bpo-23445 [https://bugs.python.org/issue?@action=redirect&bpo=23445]: pydebug builds now use “gcc -Og” where possible, to make the resulting executable faster.
- bpo-23686 [https://bugs.python.org/issue?@action=redirect&bpo=23686]: Update OS X 10.5 installer build to use OpenSSL 1.0.2a.

C API

- bpo-20204 [https://bugs.python.org/issue?@action=redirect&bpo=20204]: Deprecation warning is now raised for builtin types without the __module__ attribute.

Windows

- bpo-23465 [https://bugs.python.org/issue?@action=redirect&bpo=23465]: Implement **PEP 486**

[https://peps.python.org/pep-0486/] - Make the Python Launcher aware of virtual environments. Patch by Paul Moore.
- bpo-23437 [https://bugs.python.org/issue?@action=redirect&bpo=23437]: Make user scripts directory versioned on Windows. Patch by Paul Moore.

Python 3.5.0 alpha 1

Release date: 2015-02-08

Core and Builtins

- bpo-23285 [https://bugs.python.org/issue?@action=redirect&bpo=23285]: PEP 475 - EINTR handling.
- bpo-22735 [https://bugs.python.org/issue?@action=redirect&bpo=22735]: Fix many edge cases (including crashes) involving custom mro() implementations.
- bpo-22896 [https://bugs.python.org/issue?@action=redirect&bpo=22896]: Avoid using PyObject_AsCharBuffer(), PyObject_AsReadBuffer() and PyObject_AsWriteBuffer().
- bpo-21295 [https://bugs.python.org/issue?@action=redirect&bpo=21295]: Revert some changes (bpo-16795 [https://bugs.python.org/issue?@action=redirect&bpo=16795]) to AST line numbers and column offsets that constituted a regression.
- bpo-22986 [https://bugs.python.org/issue?@action=redirect&bpo=22986]: Allow changing an object’s __class__ between a dynamic type and static type in some cases.
- bpo-15859 [https://bugs.python.org/issue?@action=redirect&bpo=15859]: PyUnicode_EncodeFSDefault(), PyUnicode_EncodeMBCS() and PyUnicode_EncodeCodePage() now raise an exception if the object is not a Unicode object. For

PyUnicode_EncodeFSDefault(), it was already the case on platforms other than Windows. Patch written by Campbell Barton.

- bpo-21408 [https://bugs.python.org/issue?@action=redirect&bpo=21408]: The default __ne__() now returns NotImplemented if __eq__() returned NotImplemented. Original patch by Martin Panter.
- bpo-23321 [https://bugs.python.org/issue?@action=redirect&bpo=23321]: Fixed a crash in str.decode() when error handler returned replacement string longer than malformed input data.
- bpo-22286 [https://bugs.python.org/issue?@action=redirect&bpo=22286]: The “backslashreplace” error handlers now works with decoding and translating.
- bpo-23253 [https://bugs.python.org/issue?@action=redirect&bpo=23253]: Delay-load ShellExecute[AW] in os.startfile for reduced startup overhead on Windows.
- bpo-22038 [https://bugs.python.org/issue?@action=redirect&bpo=22038]: pyatomic.h now uses stdatomic.h or GCC built-in functions for atomic memory access if available. Patch written by Vitor de Lima and Gustavo Temple.
- bpo-20284 [https://bugs.python.org/issue?@action=redirect&bpo=20284]: %-interpolation (aka printf) formatting added for bytes and bytearray.
- bpo-23048 [https://bugs.python.org/issue?@action=redirect&bpo=23048]: Fix jumping out of an infinite while loop in the pdb.
- bpo-20335 [https://bugs.python.org/issue?@action=redirect&bpo=20335]: bytes constructor now raises TypeError when encoding or errors is specified with non-string argument. Based on patch by Renaud Blanch.
- bpo-22834 [https://bugs.python.org/issue?@action=redirect&bpo=22834]: If the current working directory ends up being set to a non-existent directory then import will no longer raise FileNotFoundError.

- bpo-22869 [https://bugs.python.org/issue?@action=redirect&bpo=22869]: Move the interpreter startup & shutdown code to a new dedicated pylifecycle.c module
- bpo-22847 [https://bugs.python.org/issue?@action=redirect&bpo=22847]: Improve method cache efficiency.
- bpo-22335 [https://bugs.python.org/issue?@action=redirect&bpo=22335]: Fix crash when trying to enlarge a bytearray to 0x7fffffff bytes on a 32-bit platform.
- bpo-22653 [https://bugs.python.org/issue?@action=redirect&bpo=22653]: Fix an assertion failure in debug mode when doing a reentrant dict insertion in debug mode.
- bpo-22643 [https://bugs.python.org/issue?@action=redirect&bpo=22643]: Fix integer overflow in Unicode case operations (upper, lower, title, swapcase, casefold).
- bpo-17636 [https://bugs.python.org/issue?@action=redirect&bpo=17636]: Circular imports involving relative imports are now supported.
- bpo-22604 [https://bugs.python.org/issue?@action=redirect&bpo=22604]: Fix assertion error in debug mode when dividing a complex number by (nan+0j).
- bpo-21052 [https://bugs.python.org/issue?@action=redirect&bpo=21052]: Do not raise ImportWarning when sys.path_hooks or sys.meta_path are set to None.
- bpo-16518 [https://bugs.python.org/issue?@action=redirect&bpo=16518]: Use ‘bytes-like object required’ in error messages that previously used the far more cryptic “‘x’ does not support the buffer protocol.
- bpo-22470 [https://bugs.python.org/issue?@action=redirect&bpo=22470]: Fixed integer overflow issues in “backslashreplace”, “xmlcharrefreplace”, and “surrogatepass” error handlers.

- bpo-22540 [https://bugs.python.org/issue?@action=redirect&bpo=22540]: speed up `PyObject_IsInstance` and `PyObject_IsSubclass` in the common case that the second argument has metaclass `type`.
- bpo-18711 [https://bugs.python.org/issue?@action=redirect&bpo=18711]: Add a new `PyErr_FormatV` function, similar to `PyErr_Format` but accepting a `va_list` argument.
- bpo-22520 [https://bugs.python.org/issue?@action=redirect&bpo=22520]: Fix overflow checking when generating the repr of a unicode object.
- bpo-22519 [https://bugs.python.org/issue?@action=redirect&bpo=22519]: Fix overflow checking in PyBytes_Repr.
- bpo-22518 [https://bugs.python.org/issue?@action=redirect&bpo=22518]: Fix integer overflow issues in latin-1 encoding.
- bpo-16324 [https://bugs.python.org/issue?@action=redirect&bpo=16324]: _charset parameter of MIMEText now also accepts email.charset.Charset instances. Initial patch by Claude Paroz.
- bpo-1764286 [https://bugs.python.org/issue?@action=redirect&bpo=1764286]: Fix inspect.getsource() to support decorated functions. Patch by Claudiu Popa.
- bpo-18554 [https://bugs.python.org/issue?@action=redirect&bpo=18554]: os.__all__ includes posix functions.
- bpo-21391 [https://bugs.python.org/issue?@action=redirect&bpo=21391]: Use os.path.abspath in the shutil module.
- bpo-11471 [https://bugs.python.org/issue?@action=redirect&bpo=11471]: avoid generating a JUMP_FORWARD instruction at the end of an if-block if there is no else-clause. Original patch by Eugene Toder.
- bpo-22215 [https://bugs.python.org/issue?@action=redirect&bpo=22215]: Now ValueError is raised

instead of TypeError when str or bytes argument contains not permitted null character or byte.

- bpo-22258 [https://bugs.python.org/issue?@action=redirect&bpo=22258]: Fix the internal function set_inheritable() on Illumos. This platform exposes the function `ioctl(FIOCLEX)`, but calling it fails with errno is ENOTTY: “Inappropriate ioctl for device”. set_inheritable() now falls back to the slower `fcntl()` (`F_GETFD` and then `F_SETFD`).
- bpo-21389 [https://bugs.python.org/issue?@action=redirect&bpo=21389]: Displaying the __qualname__ of the underlying function in the repr of a bound method.
- bpo-22206 [https://bugs.python.org/issue?@action=redirect&bpo=22206]: Using pthread, PyThread_create_key() now sets errno to ENOMEM and returns -1 (error) on integer overflow.
- bpo-20184 [https://bugs.python.org/issue?@action=redirect&bpo=20184]: Argument Clinic based signature introspection added for 30 of the builtin functions.
- bpo-22116 [https://bugs.python.org/issue?@action=redirect&bpo=22116]: C functions and methods (of the ‘builtin_function_or_method’ type) can now be weakref’ed. Patch by Wei Wu.
- bpo-22077 [https://bugs.python.org/issue?@action=redirect&bpo=22077]: Improve index error messages for bytearrays, bytes, lists, and tuples by adding ‘or slices’. Added ‘, not <typename>’ for bytearrays. Original patch by Claudiu Popa.
- bpo-20179 [https://bugs.python.org/issue?@action=redirect&bpo=20179]: Apply Argument Clinic to bytes and bytearray. Patch by Tal Einat.
- bpo-22082 [https://bugs.python.org/issue?@action=redirect&bpo=22082]: Clear interned strings in slotdefs.
- Upgrade Unicode database to Unicode 7.0.0.

- bpo-21897 [https://bugs.python.org/issue?@action=redirect&bpo=21897]: Fix a crash with the f_locals attribute with closure variables when frame.clear() has been called.
- bpo-21205 [https://bugs.python.org/issue?@action=redirect&bpo=21205]: Add a new `__qualname__` attribute to generator, the qualified name, and use it in the representation of a generator (`repr(gen)`). The default name of the generator (`__name__` attribute) is now get from the function instead of the code. Use `gen.gi_code.co_name` to get the name of the code.
- bpo-21669 [https://bugs.python.org/issue?@action=redirect&bpo=21669]: With the aid of heuristics in SyntaxError.__init__, the parser now attempts to generate more meaningful (or at least more search engine friendly) error messages when “exec” and “print” are used as statements.
- bpo-21642 [https://bugs.python.org/issue?@action=redirect&bpo=21642]: In the conditional if-else expression, allow an integer written with no space between itself and the `else` keyword (e.g. `True if 42else False`) to be valid syntax.
- bpo-21523 [https://bugs.python.org/issue?@action=redirect&bpo=21523]: Fix over-pessimistic computation of the stack effect of some opcodes in the compiler. This also fixes a quadratic compilation time issue noticeable when compiling code with a large number of “and” and “or” operators.
- bpo-21418 [https://bugs.python.org/issue?@action=redirect&bpo=21418]: Fix a crash in the builtin function super() when called without argument and without current frame (ex: embedded Python).
- bpo-21425 [https://bugs.python.org/issue?@action=redirect&bpo=21425]: Fix flushing of standard streams in the interactive interpreter.

- bpo-21435 [https://bugs.python.org/issue?@action=redirect&bpo=21435]: In rare cases, when running finalizers on objects in cyclic trash a bad pointer dereference could occur due to a subtle flaw in internal iteration logic.
- bpo-21377 [https://bugs.python.org/issue?@action=redirect&bpo=21377]: PyBytes_Concat() now tries to concatenate in-place when the first argument has a reference count of 1. Patch by Nikolaus Rath.
- bpo-20355 [https://bugs.python.org/issue?@action=redirect&bpo=20355]: -W command line options now have higher priority than the PYTHONWARNINGS environment variable. Patch by Arfrever.
- bpo-21274 [https://bugs.python.org/issue?@action=redirect&bpo=21274]: Define PATH_MAX for GNU/Hurd in Python/pythonrun.c.
- bpo-20904 [https://bugs.python.org/issue?@action=redirect&bpo=20904]: Support setting FPU precision on m68k.
- bpo-21209 [https://bugs.python.org/issue?@action=redirect&bpo=21209]: Fix sending tuples to custom generator objects with the yield from syntax.
- bpo-21193 [https://bugs.python.org/issue?@action=redirect&bpo=21193]: pow(a, b, c) now raises ValueError rather than TypeError when b is negative. Patch by Josh Rosenberg.
- bpo-21176 [https://bugs.python.org/issue?@action=redirect&bpo=21176]: PEP 465: Add the ‘@’ operator for matrix multiplication.
- bpo-21134 [https://bugs.python.org/issue?@action=redirect&bpo=21134]: Fix segfault when str is called on an uninitialized UnicodeEncodeError, UnicodeDecodeError, or UnicodeTranslateError object.
- bpo-19537 [https://bugs.python.org/issue?@action=redirect&bpo=19537]: Fix PyUnicode_DATA() alignment under m68k. Patch by Andreas Schwab.

- bpo-20929 [https://bugs.python.org/issue?@action=redirect&bpo=20929]: Add a type cast to avoid shifting a negative number.
- bpo-20731 [https://bugs.python.org/issue?@action=redirect&bpo=20731]: Properly position in source code files even if they are opened in text mode. Patch by Serhiy Storchaka.
- bpo-20637 [https://bugs.python.org/issue?@action=redirect&bpo=20637]: Key-sharing now also works for instance dictionaries of subclasses. Patch by Peter Ingebretson.
- bpo-8297 [https://bugs.python.org/issue?@action=redirect&bpo=8297]: Attributes missing from modules now include the module name in the error text. Original patch by ysj.ray.
- bpo-19995 [https://bugs.python.org/issue?@action=redirect&bpo=19995]: %c, %o, %x, and %X now raise TypeError on non-integer input.
- bpo-19655 [https://bugs.python.org/issue?@action=redirect&bpo=19655]: The ASDL parser - used by the build process to generate code for managing the Python AST in C - was rewritten. The new parser is self contained and does not require to carry long the spark.py parser-generator library; spark.py was removed from the source base.
- bpo-12546 [https://bugs.python.org/issue?@action=redirect&bpo=12546]: Allow `\x00` to be used as a fill character when using str, int, float, and complex __format__ methods.
- bpo-20480 [https://bugs.python.org/issue?@action=redirect&bpo=20480]: Add ipaddress.reverse_pointer. Patch by Leon Weber.
- bpo-13598 [https://bugs.python.org/issue?@action=redirect&bpo=13598]: Modify string.Formatter to support auto-numbering of replacement fields. It now matches the behavior of str.format() in this regard. Patches by Phil Elson and Ramchandra Apte.

- bpo-8931 [https://bugs.python.org/issue?@action=redirect&bpo=8931]: Make alternate formatting (‘#’) for type ‘c’ raise an exception. In versions prior to 3.5, ‘#’ with ‘c’ had no effect. Now specifying it is an error. Patch by Torsten Landschoff.
- bpo-23165 [https://bugs.python.org/issue?@action=redirect&bpo=23165]: Perform overflow checks before allocating memory in the _Py_char2wchar function.

Library

- bpo-23399 [https://bugs.python.org/issue?@action=redirect&bpo=23399]: pyvenv creates relative symlinks where possible.
- bpo-20289 [https://bugs.python.org/issue?@action=redirect&bpo=20289]: cgi.FieldStorage() now supports the context management protocol.
- bpo-13128 [https://bugs.python.org/issue?@action=redirect&bpo=13128]: Print response headers for CONNECT requests when debuglevel > 0. Patch by Demian Brecht.
- bpo-15381 [https://bugs.python.org/issue?@action=redirect&bpo=15381]: Optimized io.BytesIO to make less allocations and copyings.
- bpo-22818 [https://bugs.python.org/issue?@action=redirect&bpo=22818]: Splitting on a pattern that could match an empty string now raises a warning. Patterns that can only match empty strings are now rejected.
- bpo-23099 [https://bugs.python.org/issue?@action=redirect&bpo=23099]: Closing io.BytesIO with exported buffer is rejected now to prevent corrupting exported buffer.
- bpo-23326 [https://bugs.python.org/issue?@action=redirect&bpo=23326]: Removed __ne__ implementations. Since fixing default __ne__

implementation in bpo-21408 [https://bugs.python.org/issue?@action=redirect&bpo=21408] they are redundant.
- bpo-23363 [https://bugs.python.org/issue?@action=redirect&bpo=23363]: Fix possible overflow in itertools.permutations.
- bpo-23364 [https://bugs.python.org/issue?@action=redirect&bpo=23364]: Fix possible overflow in itertools.product.
- bpo-23366 [https://bugs.python.org/issue?@action=redirect&bpo=23366]: Fixed possible integer overflow in itertools.combinations.
- bpo-23369 [https://bugs.python.org/issue?@action=redirect&bpo=23369]: Fixed possible integer overflow in _json.encode_basestring_ascii.
- bpo-23353 [https://bugs.python.org/issue?@action=redirect&bpo=23353]: Fix the exception handling of generators in PyEval_EvalFrameEx(). At entry, save or swap the exception state even if PyEval_EvalFrameEx() is called with throwflag=0. At exit, the exception state is now always restored or swapped, not only if why is WHY_YIELD or WHY_RETURN. Patch co-written with Antoine Pitrou.
- bpo-14099 [https://bugs.python.org/issue?@action=redirect&bpo=14099]: Restored support of writing ZIP files to tellable but non-seekable streams.
- bpo-14099 [https://bugs.python.org/issue?@action=redirect&bpo=14099]: Writing to ZipFile and reading multiple ZipExtFiles is threadsafe now.
- bpo-19361 [https://bugs.python.org/issue?@action=redirect&bpo=19361]: JSON decoder now raises JSONDecodeError instead of ValueError.
- bpo-18518 [https://bugs.python.org/issue?@action=redirect&bpo=18518]: timeit now rejects statements which can’t be compiled outside a function or a loop (e.g. “return” or “break”).

- bpo-23094 [https://bugs.python.org/issue?@action=redirect&bpo=23094]: Fixed readline with frames in Python implementation of pickle.
- bpo-23268 [https://bugs.python.org/issue?@action=redirect&bpo=23268]: Fixed bugs in the comparison of ipaddress classes.
- bpo-21408 [https://bugs.python.org/issue?@action=redirect&bpo=21408]: Removed incorrect implementations of __ne__() which didn’t returned NotImplemented if __eq__() returned NotImplemented. The default __ne__() now works correctly.
- bpo-19996 [https://bugs.python.org/issue?@action=redirect&bpo=19996]: **`email.feedparser.FeedParser`** now handles (malformed) headers with no key rather than assuming the body has started.
- bpo-20188 [https://bugs.python.org/issue?@action=redirect&bpo=20188]: Support Application-Layer Protocol Negotiation (ALPN) in the ssl module.
- bpo-23133 [https://bugs.python.org/issue?@action=redirect&bpo=23133]: Pickling of ipaddress objects now produces more compact and portable representation.
- bpo-23248 [https://bugs.python.org/issue?@action=redirect&bpo=23248]: Update ssl error codes from latest OpenSSL git master.
- bpo-23266 [https://bugs.python.org/issue?@action=redirect&bpo=23266]: Much faster implementation of ipaddress.collapse_addresses() when there are many non-consecutive addresses.
- bpo-23098 [https://bugs.python.org/issue?@action=redirect&bpo=23098]: 64-bit dev_t is now supported in the os module.
- bpo-21817 [https://bugs.python.org/issue?@action=redirect&bpo=21817]: When an exception is raised in a task submitted to a ProcessPoolExecutor, the remote

traceback is now displayed in the parent process. Patch by Claudiu Popa.

- bpo-15955 [https://bugs.python.org/issue?@action=redirect&bpo=15955]: Add an option to limit output size when decompressing LZMA data. Patch by Nikolaus Rath and Martin Panter.
- bpo-23250 [https://bugs.python.org/issue?@action=redirect&bpo=23250]: In the http.cookies module, capitalize "HttpOnly" and "Secure" as they are written in the standard.
- bpo-23063 [https://bugs.python.org/issue?@action=redirect&bpo=23063]: In the distutils' check command, fix parsing of reST with code or code-block directives.
- bpo-23209 [https://bugs.python.org/issue?@action=redirect&bpo=23209]: selectors.BaseSelector.get_key() now raises a RuntimeError if the selector is closed. And selectors.BaseSelector.close() now clears its internal reference to the selector mapping to break a reference cycle. Initial patch written by Martin Richard. (See also: bpo-23225 [https://bugs.python.org/issue?@action=redirect&bpo=23225])
- bpo-17911 [https://bugs.python.org/issue?@action=redirect&bpo=17911]: Provide a way to seed the linecache for a PEP-302 module without actually loading the code.
- bpo-17911 [https://bugs.python.org/issue?@action=redirect&bpo=17911]: Provide a new object API for traceback, including the ability to not lookup lines at all until the traceback is actually rendered, without any trace of the original objects being kept alive.
- bpo-19777 [https://bugs.python.org/issue?@action=redirect&bpo=19777]: Provide a home() classmethod on Path objects. Contributed by Victor Salgado and Mayank Tripathi.

- bpo-23206 [https://bugs.python.org/issue?@action=redirect&bpo=23206]: Make `json.dumps(..., ensure_ascii=False)` as fast as the default case of `ensure_ascii=True`. Patch by Naoki Inada.
- bpo-23185 [https://bugs.python.org/issue?@action=redirect&bpo=23185]: Add math.inf and math.nan constants.
- bpo-23186 [https://bugs.python.org/issue?@action=redirect&bpo=23186]: Add ssl.SSLObject.shared_ciphers() and ssl.SSLSocket.shared_ciphers() to fetch the client's list ciphers sent at handshake.
- bpo-23143 [https://bugs.python.org/issue?@action=redirect&bpo=23143]: Remove compatibility with OpenSSLs older than 0.9.8.
- bpo-23132 [https://bugs.python.org/issue?@action=redirect&bpo=23132]: Improve performance and introspection support of comparison methods created by functool.total_ordering.
- bpo-19776 [https://bugs.python.org/issue?@action=redirect&bpo=19776]: Add an expanduser() method on Path objects.
- bpo-23112 [https://bugs.python.org/issue?@action=redirect&bpo=23112]: Fix SimpleHTTPServer to correctly carry the query string and fragment when it redirects to add a trailing slash.
- bpo-21793 [https://bugs.python.org/issue?@action=redirect&bpo=21793]: Added http.HTTPStatus enums (i.e. HTTPStatus.OK, HTTPStatus.NOT_FOUND). Patch by Demian Brecht.
- bpo-23093 [https://bugs.python.org/issue?@action=redirect&bpo=23093]: In the io, module allow more operations to work on detached streams.
- bpo-23111 [https://bugs.python.org/issue?@action=redirect&bpo=23111]: In the ftplib, make ssl.PROTOCOL_SSLv23 the default protocol version.

- bpo-22585 [https://bugs.python.org/issue?@action=redirect&bpo=22585]: On OpenBSD 5.6 and newer, os.urandom() now calls getentropy(), instead of reading /dev/urandom, to get pseudo-random bytes.
- bpo-19104 [https://bugs.python.org/issue?@action=redirect&bpo=19104]: pprint now produces evaluable output for wrapped strings.
- bpo-23071 [https://bugs.python.org/issue?@action=redirect&bpo=23071]: Added missing names to codecs.__all__. Patch by Martin Panter.
- bpo-22783 [https://bugs.python.org/issue?@action=redirect&bpo=22783]: Pickling now uses the NEWOBJ opcode instead of the NEWOBJ_EX opcode if possible.
- bpo-15513 [https://bugs.python.org/issue?@action=redirect&bpo=15513]: Added a __sizeof__ implementation for pickle classes.
- bpo-19858 [https://bugs.python.org/issue?@action=redirect&bpo=19858]: pickletools.optimize() now aware of the MEMOIZE opcode, can produce more compact result and no longer produces invalid output if input data contains MEMOIZE opcodes together with PUT or BINPUT opcodes.
- bpo-22095 [https://bugs.python.org/issue?@action=redirect&bpo=22095]: Fixed HTTPConnection.set_tunnel with default port. The port value in the host header was set to “None”. Patch by Demian Brecht.
- bpo-23016 [https://bugs.python.org/issue?@action=redirect&bpo=23016]: A warning no longer produces an AttributeError when the program is run with pythonw.exe.
- bpo-21775 [https://bugs.python.org/issue?@action=redirect&bpo=21775]: shutil.copytree(): fix crash when copying to VFAT. An exception handler assumed that OSError objects always have a ‘winerror’ attribute. That is not the case, so the exception handler itself

raised AttributeError when run on Linux (and, presumably, any other non-Windows OS). Patch by Greg Ward.

- bpo-1218234 [https://bugs.python.org/issue?@action=redirect&bpo=1218234]: Fix inspect.getsource() to load updated source of reloaded module. Initial patch by Berker Peksag.
- bpo-21740 [https://bugs.python.org/issue?@action=redirect&bpo=21740]: Support wrapped callables in doctest. Patch by Claudiu Popa.
- bpo-23009 [https://bugs.python.org/issue?@action=redirect&bpo=23009]: Make sure selectors.EpollSelector.select() works when no FD is registered.
- bpo-22959 [https://bugs.python.org/issue?@action=redirect&bpo=22959]: In the constructor of http.client.HTTPSConnection, prefer the context’s check_hostname attribute over the *check_hostname* parameter.
- bpo-22696 [https://bugs.python.org/issue?@action=redirect&bpo=22696]: Add function `sys.is_finalizing()` to know about interpreter shutdown.
- bpo-16043 [https://bugs.python.org/issue?@action=redirect&bpo=16043]: Add a default limit for the amount of data xmlrpclib.gzip_decode will return. This resolves **CVE 2013-1753** [https://www.cve.org/CVERecord?id=CVE-2013-1753].
- bpo-14099 [https://bugs.python.org/issue?@action=redirect&bpo=14099]: ZipFile.open() no longer reopen the underlying file. Objects returned by ZipFile.open() can now operate independently of the ZipFile even if the ZipFile was created by passing in a file-like object as the first argument to the constructor.
- bpo-22966 [https://bugs.python.org/issue?@action=redirect&bpo=22966]: Fix __pycache__ pyc file name

clobber when pyc_compile is asked to compile a source file containing multiple dots in the source file name.

- bpo-21971 [https://bugs.python.org/issue?@action=redirect&bpo=21971]: Update turtledemo doc and add module to the index.
- bpo-21032 [https://bugs.python.org/issue?@action=redirect&bpo=21032]: Fixed socket leak if HTTPConnection.getresponse() fails. Original patch by Martin Panter.
- bpo-22407 [https://bugs.python.org/issue?@action=redirect&bpo=22407]: Deprecated the use of re.LOCALE flag with str patterns or re.ASCII. It was newer worked.
- bpo-22902 [https://bugs.python.org/issue?@action=redirect&bpo=22902]: The “ip” command is now used on Linux to determine MAC address in uuid.getnode(). Patch by Bruno Cauet.
- bpo-22960 [https://bugs.python.org/issue?@action=redirect&bpo=22960]: Add a context argument to xmlrpclib.ServerProxy constructor.
- bpo-22389 [https://bugs.python.org/issue?@action=redirect&bpo=22389]: Add contextlib.redirect_stderr().
- bpo-21356 [https://bugs.python.org/issue?@action=redirect&bpo=21356]: Make ssl.RAND_egd() optional to support LibreSSL. The availability of the function is checked during the compilation. Patch written by Bernard Spil.
- bpo-22915 [https://bugs.python.org/issue?@action=redirect&bpo=22915]: SAX parser now supports files opened with file descriptor or bytes path.
- bpo-22609 [https://bugs.python.org/issue?@action=redirect&bpo=22609]: Constructors and update methods of mapping classes in the collections module now accept the self keyword argument.
- bpo-22940 [https://bugs.python.org/issue?@action=redirect&bpo=22940]: Add

readline.append_history_file.

- bpo-19676 [https://bugs.python.org/issue?@action=redirect&bpo=19676]: Added the “namereplace” error handler.
- bpo-22788 [https://bugs.python.org/issue?@action=redirect&bpo=22788]: Add *context* parameter to logging.handlers.HTTPHandler.
- bpo-22921 [https://bugs.python.org/issue?@action=redirect&bpo=22921]: Allow SSLContext to take the *hostname* parameter even if OpenSSL doesn’t support SNI.
- bpo-22894 [https://bugs.python.org/issue?@action=redirect&bpo=22894]: TestCase.subTest() would cause the test suite to be stopped when in failfast mode, even in the absence of failures.
- bpo-22796 [https://bugs.python.org/issue?@action=redirect&bpo=22796]: HTTP cookie parsing is now stricter, in order to protect against potential injection attacks.
- bpo-22370 [https://bugs.python.org/issue?@action=redirect&bpo=22370]: Windows detection in pathlib is now more robust.
- bpo-22841 [https://bugs.python.org/issue?@action=redirect&bpo=22841]: Reject coroutines in asyncio add_signal_handler(). Patch by Ludovic.Gasc.
- bpo-19494 [https://bugs.python.org/issue?@action=redirect&bpo=19494]: Added urllib.request.HTTPBasicPriorAuthHandler. Patch by Matej Cepl.
- bpo-22578 [https://bugs.python.org/issue?@action=redirect&bpo=22578]: Added attributes to the re.error class.
- bpo-22849 [https://bugs.python.org/issue?@action=redirect&bpo=22849]: Fix possible double free in the io.TextIOWrapper constructor.

- bpo-12728 [https://bugs.python.org/issue?@action=redirect&bpo=12728]: Different Unicode characters having the same uppercase but different lowercase are now matched in case-insensitive regular expressions.
- bpo-22821 [https://bugs.python.org/issue?@action=redirect&bpo=22821]: Fixed fcntl() with integer argument on 64-bit big-endian platforms.
- bpo-21650 [https://bugs.python.org/issue?@action=redirect&bpo=21650]: Add an `--sort-keys` option to `json.tool` CLI.
- bpo-22824 [https://bugs.python.org/issue?@action=redirect&bpo=22824]: Updated reprlib output format for sets to use set literals. Patch contributed by Berker Peksag.
- bpo-22824 [https://bugs.python.org/issue?@action=redirect&bpo=22824]: Updated reprlib output format for arrays to display empty arrays without an unnecessary empty list. Suggested by Serhiy Storchaka.
- bpo-22406 [https://bugs.python.org/issue?@action=redirect&bpo=22406]: Fixed the uu_codec codec incorrectly ported to 3.x. Based on patch by Martin Panter.
- bpo-17293 [https://bugs.python.org/issue?@action=redirect&bpo=17293]: uuid.getnode() now determines MAC address on AIX using netstat. Based on patch by Aivars Kalvāns.
- bpo-22769 [https://bugs.python.org/issue?@action=redirect&bpo=22769]: Fixed ttk.Treeview.tag_has() when called without arguments.
- bpo-22417 [https://bugs.python.org/issue?@action=redirect&bpo=22417]: Verify certificates by default in httplib (PEP 476).
- bpo-22775 [https://bugs.python.org/issue?@action=redirect&bpo=22775]: Fixed unpickling of http.cookies.SimpleCookie with protocol 2 and above. Patch by Tim Graham.

- bpo-22776 [https://bugs.python.org/issue?@action=redirect&bpo=22776]: Brought excluded code into the scope of a try block in SysLogHandler.emit().
- bpo-22665 [https://bugs.python.org/issue?@action=redirect&bpo=22665]: Add missing get_terminal_size and SameFileError to shutil.__all__.
- bpo-6623 [https://bugs.python.org/issue?@action=redirect&bpo=6623]: Remove deprecated Netrc class in the ftplib module. Patch by Matt Chaput.
- bpo-17381 [https://bugs.python.org/issue?@action=redirect&bpo=17381]: Fixed handling of case-insensitive ranges in regular expressions.
- bpo-22410 [https://bugs.python.org/issue?@action=redirect&bpo=22410]: Module level functions in the re module now cache compiled locale-dependent regular expressions taking into account the locale.
- bpo-22759 [https://bugs.python.org/issue?@action=redirect&bpo=22759]: Query methods on pathlib.Path() (exists(), is_dir(), etc.) now return False when the underlying stat call raises NotADirectoryError.
- bpo-8876 [https://bugs.python.org/issue?@action=redirect&bpo=8876]: distutils now falls back to copying files when hard linking doesn’t work. This allows use with special filesystems such as VirtualBox shared folders.
- bpo-22217 [https://bugs.python.org/issue?@action=redirect&bpo=22217]: Implemented reprs of classes in the zipfile module.
- bpo-22457 [https://bugs.python.org/issue?@action=redirect&bpo=22457]: Honour load_tests in the start_dir of discovery.
- bpo-18216 [https://bugs.python.org/issue?@action=redirect&bpo=18216]: gettext now raises an error when a .mo file has an unsupported major version number. Patch by Aaron Hill.
- bpo-13918 [https://bugs.python.org/issue?@action=redirect&bpo=13918]: Provide a locale.delocalize()

function which can remove locale-specific number formatting from a string representing a number, without then converting it to a specific type. Patch by Cédric Krier.

- bpo-22676 [https://bugs.python.org/issue?@action=redirect&bpo=22676]: Make the pickling of global objects which don't have a __module__ attribute less slow.
- bpo-18853 [https://bugs.python.org/issue?@action=redirect&bpo=18853]: Fixed ResourceWarning in shlex.__nain__.
- bpo-9351 [https://bugs.python.org/issue?@action=redirect&bpo=9351]: Defaults set with set_defaults on an argparse subparser are no longer ignored when also set on the parent parser.
- bpo-7559 [https://bugs.python.org/issue?@action=redirect&bpo=7559]: unittest test loading ImportErrors are reported as import errors with their import exception rather than as attribute errors after the import has already failed.
- bpo-19746 [https://bugs.python.org/issue?@action=redirect&bpo=19746]: Make it possible to examine the errors from unittest discovery without executing the test suite. The new `errors` attribute on `TestLoader` exposes these non-fatal errors encountered during discovery.
- bpo-21991 [https://bugs.python.org/issue?@action=redirect&bpo=21991]: Make email.headerregistry's header 'params' attributes be read-only (MappingProxyType). Previously the dictionary was modifiable but a new one was created on each access of the attribute.
- bpo-22638 [https://bugs.python.org/issue?@action=redirect&bpo=22638]: SSLv3 is now disabled throughout the standard library. It can still be enabled by instantiating a SSLContext manually.
- bpo-22641 [https://bugs.python.org/issue?@action=redirect&bpo=22641]: In asyncio, the default SSL

context for client connections is now created using ssl.create_default_context(), for stronger security.

- bpo-17401 [https://bugs.python.org/issue?@action=redirect&bpo=17401]: Include closefd in io.FileIO repr.
- bpo-21338 [https://bugs.python.org/issue?@action=redirect&bpo=21338]: Add silent mode for compileall. quiet parameters of compile_{dir, file, path} functions now have a multilevel value. Also, -q option of the CLI now have a multilevel value. Patch by Thomas Kluyver.
- bpo-20152 [https://bugs.python.org/issue?@action=redirect&bpo=20152]: Convert the array and cmath modules to Argument Clinic.
- bpo-18643 [https://bugs.python.org/issue?@action=redirect&bpo=18643]: Add socket.socketpair() on Windows.
- bpo-22435 [https://bugs.python.org/issue?@action=redirect&bpo=22435]: Fix a file descriptor leak when socketserver bind fails.
- bpo-13096 [https://bugs.python.org/issue?@action=redirect&bpo=13096]: Fixed segfault in CTypes POINTER handling of large values.
- bpo-11694 [https://bugs.python.org/issue?@action=redirect&bpo=11694]: Raise ConversionError in xdrlib as documented. Patch by Filip Gruszczyński and Claudiu Popa.
- bpo-19380 [https://bugs.python.org/issue?@action=redirect&bpo=19380]: Optimized parsing of regular expressions.
- bpo-1519638 [https://bugs.python.org/issue?@action=redirect&bpo=1519638]: Now unmatched groups are replaced with empty strings in re.sub() and re.subn().
- bpo-18615 [https://bugs.python.org/issue?@action=redirect&bpo=18615]: sndhdr.what/whathdr now return a namedtuple.
- bpo-22462 [https://bugs.python.org/issue?@action=redirect&bpo=22462]: Fix pyexpat’s creation of a

dummy frame to make it appear in exception tracebacks.

- bpo-21965 [https://bugs.python.org/issue?@action=redirect&bpo=21965]: Add support for in-memory SSL to the ssl module. Patch by Geert Jansen.
- bpo-21173 [https://bugs.python.org/issue?@action=redirect&bpo=21173]: Fix len() on a WeakKeyDictionary when .clear() was called with an iterator alive.
- bpo-11866 [https://bugs.python.org/issue?@action=redirect&bpo=11866]: Eliminated race condition in the computation of names for new threads.
- bpo-21905 [https://bugs.python.org/issue?@action=redirect&bpo=21905]: Avoid RuntimeError in pickle.whichmodule() when sys.modules is mutated while iterating. Patch by Olivier Grisel.
- bpo-11271 [https://bugs.python.org/issue?@action=redirect&bpo=11271]: concurrent.futures.Executor.map() now takes a *chunksize* argument to allow batching of tasks in child processes and improve performance of ProcessPoolExecutor. Patch by Dan O’Reilly.
- bpo-21883 [https://bugs.python.org/issue?@action=redirect&bpo=21883]: os.path.join() and os.path.relpath() now raise a TypeError with more helpful error message for unsupported or mismatched types of arguments.
- bpo-22219 [https://bugs.python.org/issue?@action=redirect&bpo=22219]: The zipfile module CLI now adds entries for directories (including empty directories) in ZIP file.
- bpo-22449 [https://bugs.python.org/issue?@action=redirect&bpo=22449]: In the ssl.SSLContext.load_default_certs, consult the environmental variables SSL_CERT_DIR and SSL_CERT_FILE on Windows.

- bpo-22508 [https://bugs.python.org/issue?@action=redirect&bpo=22508]: The email.__version__ variable has been removed; the email code is no longer shipped separately from the stdlib, and __version__ hasn't been updated in several releases.
- bpo-20076 [https://bugs.python.org/issue?@action=redirect&bpo=20076]: Added non derived UTF-8 aliases to locale aliases table.
- bpo-20079 [https://bugs.python.org/issue?@action=redirect&bpo=20079]: Added locales supported in glibc 2.18 to locale alias table.
- bpo-20218 [https://bugs.python.org/issue?@action=redirect&bpo=20218]: Added convenience methods read_text/write_text and read_bytes/ write_bytes to pathlib.Path objects.
- bpo-22396 [https://bugs.python.org/issue?@action=redirect&bpo=22396]: On 32-bit AIX platform, don't expose os.posix_fadvise() nor os.posix_fallocate() because their prototypes in system headers are wrong.
- bpo-22517 [https://bugs.python.org/issue?@action=redirect&bpo=22517]: When an io.BufferedRWPair object is deallocated, clear its weakrefs.
- bpo-22437 [https://bugs.python.org/issue?@action=redirect&bpo=22437]: Number of capturing groups in regular expression is no longer limited by 100.
- bpo-17442 [https://bugs.python.org/issue?@action=redirect&bpo=17442]: InteractiveInterpreter now displays the full chained traceback in its showtraceback method, to match the built in interactive interpreter.
- bpo-23392 [https://bugs.python.org/issue?@action=redirect&bpo=23392]: Added tests for marshal C API that works with FILE*.
- bpo-10510 [https://bugs.python.org/issue?@action=redirect&bpo=10510]: distutils register and upload methods now use HTML standards compliant CRLF line endings.

- bpo-9850 [https://bugs.python.org/issue?@action=redirect&bpo=9850]: Fixed macpath.join() for empty first component. Patch by Oleg Oshmyan.
- bpo-5309 [https://bugs.python.org/issue?@action=redirect&bpo=5309]: distutils' build and build_ext commands now accept a `-j` option to enable parallel building of extension modules.
- bpo-22448 [https://bugs.python.org/issue?@action=redirect&bpo=22448]: Improve canceled timer handles cleanup to prevent unbound memory usage. Patch by Joshua Moore-Oliva.
- bpo-22427 [https://bugs.python.org/issue?@action=redirect&bpo=22427]: TemporaryDirectory no longer attempts to clean up twice when used in the with statement in generator.
- bpo-22362 [https://bugs.python.org/issue?@action=redirect&bpo=22362]: Forbidden ambiguous octal escapes out of range 0-0o377 in regular expressions.
- bpo-20912 [https://bugs.python.org/issue?@action=redirect&bpo=20912]: Now directories added to ZIP file have correct Unix and MS-DOS directory attributes.
- bpo-21866 [https://bugs.python.org/issue?@action=redirect&bpo=21866]: ZipFile.close() no longer writes ZIP64 central directory records if allowZip64 is false.
- bpo-22278 [https://bugs.python.org/issue?@action=redirect&bpo=22278]: Fix urljoin problem with relative urls, a regression observed after changes to issue22118 were submitted.
- bpo-22415 [https://bugs.python.org/issue?@action=redirect&bpo=22415]: Fixed debugging output of the GROUPREF_EXISTS opcode in the re module. Removed trailing spaces in debugging output.
- bpo-22423 [https://bugs.python.org/issue?@action=redirect&bpo=22423]: Unhandled exception in thread no longer causes unhandled AttributeError when sys.stderr is None.

- bpo-21332 [https://bugs.python.org/issue?@action=redirect&bpo=21332]: Ensure that `bufsize=1` in subprocess.Popen() selects line buffering, rather than block buffering. Patch by Akira Li.
- bpo-21091 [https://bugs.python.org/issue?@action=redirect&bpo=21091]: Fix API bug: email.message.EmailMessage.is_attachment is now a method.
- bpo-21079 [https://bugs.python.org/issue?@action=redirect&bpo=21079]: Fix email.message.EmailMessage.is_attachment to return the correct result when the header has parameters as well as a value.
- bpo-22247 [https://bugs.python.org/issue?@action=redirect&bpo=22247]: Add NNTPError to nntplib.__all__.
- bpo-22366 [https://bugs.python.org/issue?@action=redirect&bpo=22366]: urllib.request.urlopen will accept a context object (SSLContext) as an argument which will then be used for HTTPS connection. Patch by Alex Gaynor.
- bpo-4180 [https://bugs.python.org/issue?@action=redirect&bpo=4180]: The warnings registries are now reset when the filters are modified.
- bpo-22419 [https://bugs.python.org/issue?@action=redirect&bpo=22419]: Limit the length of incoming HTTP request in wsgiref server to 65536 bytes and send a 414 error code for higher lengths. Patch contributed by Devin Cook.
- Lax cookie parsing in http.cookies could be a security issue when combined with non-standard cookie handling in some web browsers. Reported by Sergey Bobrov.
- bpo-20537 [https://bugs.python.org/issue?@action=redirect&bpo=20537]: logging methods now accept an exception instance as well as a Boolean value or exception tuple. Thanks to Yury Selivanov for the patch.

- bpo-22384 [https://bugs.python.org/issue?@action=redirect&bpo=22384]: An exception in Tkinter callback no longer crashes the program when it is run with pythonw.exe.
- bpo-22168 [https://bugs.python.org/issue?@action=redirect&bpo=22168]: Prevent turtle AttributeError with non-default Canvas on OS X.
- bpo-21147 [https://bugs.python.org/issue?@action=redirect&bpo=21147]: sqlite3 now raises an exception if the request contains a null character instead of truncating it. Based on patch by Victor Stinner.
- bpo-13968 [https://bugs.python.org/issue?@action=redirect&bpo=13968]: The glob module now supports recursive search in subdirectories using the `**` pattern.
- bpo-21951 [https://bugs.python.org/issue?@action=redirect&bpo=21951]: Fixed a crash in Tkinter on AIX when called Tcl command with empty string or tuple argument.
- bpo-21951 [https://bugs.python.org/issue?@action=redirect&bpo=21951]: Tkinter now most likely raises MemoryError instead of crash if the memory allocation fails.
- bpo-22338 [https://bugs.python.org/issue?@action=redirect&bpo=22338]: Fix a crash in the json module on memory allocation failure.
- bpo-12410 [https://bugs.python.org/issue?@action=redirect&bpo=12410]: imaplib.IMAP4 now supports the context management protocol. Original patch by Tarek Ziadé.
- bpo-21270 [https://bugs.python.org/issue?@action=redirect&bpo=21270]: We now override tuple methods in mock.call objects so that they can be used as normal call attributes.
- bpo-16662 [https://bugs.python.org/issue?@action=redirect&bpo=16662]: `load_tests()` is now unconditionally run when it is present in a package’s

`__init__.py`. `TestLoader.loadTestsFromModule()` still accepts use_load_tests, but it is deprecated and ignored. A new keyword-only attribute `pattern` is added and documented. Patch given by Robert Collins, tweaked by Barry Warsaw.

- bpo-22226 [https://bugs.python.org/issue?@action=redirect&bpo=22226]: First letter no longer is stripped from the “status” key in the result of Treeview.heading().
- bpo-19524 [https://bugs.python.org/issue?@action=redirect&bpo=19524]: Fixed resource leak in the HTTP connection when an invalid response is received. Patch by Martin Panter.
- bpo-20421 [https://bugs.python.org/issue?@action=redirect&bpo=20421]: Add a .version() method to SSL sockets exposing the actual protocol version in use.
- bpo-19546 [https://bugs.python.org/issue?@action=redirect&bpo=19546]: configparser exceptions no longer expose implementation details. Chained KeyErrors are removed, which leads to cleaner tracebacks. Patch by Claudiu Popa.
- bpo-22051 [https://bugs.python.org/issue?@action=redirect&bpo=22051]: turtledemo no longer reloads examples to re-run them. Initialization of variables and gui setup should be done in main(), which is called each time a demo is run, but not on import.
- bpo-21933 [https://bugs.python.org/issue?@action=redirect&bpo=21933]: Turtledemo users can change the code font size with a menu selection or control(command) ‘-’ or ‘+’ or control-mousewheel. Original patch by Lita Cho.
- bpo-21597 [https://bugs.python.org/issue?@action=redirect&bpo=21597]: The separator between the turtledemo text pane and the drawing canvas can now be grabbed and dragged with a mouse. The code text pane can be widened to easily view or copy the full

width of the text. The canvas can be widened on small screens. Original patches by Jan Kanis and Lita Cho.

- bpo-18132 [https://bugs.python.org/issue?@action=redirect&bpo=18132]: Turtledemo buttons no longer disappear when the window is shrunk. Original patches by Jan Kanis and Lita Cho.
- bpo-22043 [https://bugs.python.org/issue?@action=redirect&bpo=22043]: time.monotonic() is now always available. `threading.Lock.acquire()`, `threading.RLock.acquire()` and socket operations now use a monotonic clock, instead of the system clock, when a timeout is used.
- bpo-21527 [https://bugs.python.org/issue?@action=redirect&bpo=21527]: Add a default number of workers to ThreadPoolExecutor equal to 5 times the number of CPUs. Patch by Claudiu Popa.
- bpo-22216 [https://bugs.python.org/issue?@action=redirect&bpo=22216]: smtplib now resets its state more completely after a quit. The most obvious consequence of the previous behavior was a STARTTLS failure during a connect/starttls/quit/connect/starttls sequence.
- bpo-22098 [https://bugs.python.org/issue?@action=redirect&bpo=22098]: ctypes’ BigEndianStructure and LittleEndianStructure now define an empty __slots__ so that subclasses don’t always get an instance dict. Patch by Claudiu Popa.
- bpo-22185 [https://bugs.python.org/issue?@action=redirect&bpo=22185]: Fix an occasional RuntimeError in threading.Condition.wait() caused by mutation of the waiters queue without holding the lock. Patch by Doug Zongker.
- bpo-22287 [https://bugs.python.org/issue?@action=redirect&bpo=22287]: On UNIX, _PyTime_gettimeofday() now uses clock_gettime(CLOCK_REALTIME) if available. As a side

effect, Python now depends on the librt library on Solaris and on Linux (only with glibc older than 2.17).

- bpo-22182 [https://bugs.python.org/issue?@action=redirect&bpo=22182]: Use e.args to unpack exceptions correctly in distutils.file_util.move_file. Patch by Claudiu Popa.
- The webbrowser module now uses subprocess's start_new_session=True rather than a potentially risky preexec_fn=os.setsid call.
- bpo-22042 [https://bugs.python.org/issue?@action=redirect&bpo=22042]: signal.set_wakeup_fd(fd) now raises an exception if the file descriptor is in blocking mode.
- bpo-16808 [https://bugs.python.org/issue?@action=redirect&bpo=16808]: inspect.stack() now returns a named tuple instead of a tuple. Patch by Daniel Shahaf.
- bpo-22236 [https://bugs.python.org/issue?@action=redirect&bpo=22236]: Fixed Tkinter images copying operations in NoDefaultRoot mode.
- bpo-2527 [https://bugs.python.org/issue?@action=redirect&bpo=2527]: Add a *globals* argument to timeit functions, in order to override the globals namespace in which the timed code is executed. Patch by Ben Roberts.
- bpo-22118 [https://bugs.python.org/issue?@action=redirect&bpo=22118]: Switch urllib.parse to use RFC 3986 semantics for the resolution of relative URLs, rather than RFCs 1808 and 2396. Patch by Demian Brecht.
- bpo-21549 [https://bugs.python.org/issue?@action=redirect&bpo=21549]: Added the "members" parameter to TarFile.list().
- bpo-19628 [https://bugs.python.org/issue?@action=redirect&bpo=19628]: Allow compileall recursion depth to be specified with a -r option.
- bpo-15696 [https://bugs.python.org/issue?@action=redirect&bpo=15696]: Add a __sizeof__

implementation for mmap objects on Windows.

- bpo-22068 [https://bugs.python.org/issue?@action=redirect&bpo=22068]: Avoided reference loops with Variables and Fonts in Tkinter.
- bpo-22165 [https://bugs.python.org/issue?@action=redirect&bpo=22165]: SimpleHTTPRequestHandler now supports undecodable file names.
- bpo-15381 [https://bugs.python.org/issue?@action=redirect&bpo=15381]: Optimized line reading in io.BytesIO.
- bpo-8797 [https://bugs.python.org/issue?@action=redirect&bpo=8797]: Raise HTTPError on failed Basic Authentication immediately. Initial patch by Sam Bull.
- bpo-20729 [https://bugs.python.org/issue?@action=redirect&bpo=20729]: Restored the use of lazy iterkeys()/itervalues()/iteritems() in the mailbox module.
- bpo-21448 [https://bugs.python.org/issue?@action=redirect&bpo=21448]: Changed FeedParser feed() to avoid $O(n^2)$ behavior when parsing long line. Original patch by Raymond Hettinger.
- bpo-22184 [https://bugs.python.org/issue?@action=redirect&bpo=22184]: The functools LRU Cache decorator factory now gives an earlier and clearer error message when the user forgets the required parameters.
- bpo-17923 [https://bugs.python.org/issue?@action=redirect&bpo=17923]: glob() patterns ending with a slash no longer match non-dirs on AIX. Based on patch by Delhallt.
- bpo-21725 [https://bugs.python.org/issue?@action=redirect&bpo=21725]: Added support for RFC 6531 (SMTPUTF8) in smtpd.
- bpo-22176 [https://bugs.python.org/issue?@action=redirect&bpo=22176]: Update the ctypes module’s libffi to v3.1. This release adds support for the Linux

AArch64 and POWERPC ELF ABIv2 little endian architectures.

- bpo-5411 [https://bugs.python.org/issue?@action=redirect&bpo=5411]: Added support for the “xztar” format in the shutil module.
- bpo-21121 [https://bugs.python.org/issue?@action=redirect&bpo=21121]: Don’t force 3rd party C extensions to be built with -Werror=declaration-after-statement.
- bpo-21975 [https://bugs.python.org/issue?@action=redirect&bpo=21975]: Fixed crash when using uninitialized sqlite3.Row (in particular when unpickling pickled sqlite3.Row). sqlite3.Row is now initialized in the __new__() method.
- bpo-20170 [https://bugs.python.org/issue?@action=redirect&bpo=20170]: Convert posixmodule to use Argument Clinic.
- bpo-21539 [https://bugs.python.org/issue?@action=redirect&bpo=21539]: Add an *exists_ok* argument to `Pathlib.mkdir()` to mimic `mkdir -p` and `os.makedirs()` functionality. When true, ignore `FileExistsErrors`. Patch by Berker Peksag.
- bpo-22127 [https://bugs.python.org/issue?@action=redirect&bpo=22127]: Bypass IDNA for pure-ASCII host names in the socket module (in particular for numeric IPs).
- bpo-21047 [https://bugs.python.org/issue?@action=redirect&bpo=21047]: set the default value for the *convert_charrefs* argument of HTMLParser to True. Patch by Berker Peksag.
- Add an __all__ to html.entities.
- bpo-15114 [https://bugs.python.org/issue?@action=redirect&bpo=15114]: the strict mode and argument of HTMLParser, HTMLParser.error, and the HTMLParserError exception have been removed.

- bpo-22085 [https://bugs.python.org/issue?@action=redirect&bpo=22085]: Dropped support of Tk 8.3 in Tkinter.
- bpo-21580 [https://bugs.python.org/issue?@action=redirect&bpo=21580]: Now Tkinter correctly handles bytes arguments passed to Tk. In particular this allows initializing images from binary data.
- bpo-22003 [https://bugs.python.org/issue?@action=redirect&bpo=22003]: When initialized from a bytes object, io.BytesIO() now defers making a copy until it is mutated, improving performance and memory use on some use cases. Patch by David Wilson.
- bpo-22018 [https://bugs.python.org/issue?@action=redirect&bpo=22018]: On Windows, signal.set_wakeup_fd() now also supports sockets. A side effect is that Python depends to the WinSock library.
- bpo-22054 [https://bugs.python.org/issue?@action=redirect&bpo=22054]: Add os.get_blocking() and os.set_blocking() functions to get and set the blocking mode of a file descriptor (False if the O_NONBLOCK flag is set, True otherwise). These functions are not available on Windows.
- bpo-17172 [https://bugs.python.org/issue?@action=redirect&bpo=17172]: Make turtledemo start as active on OS X even when run with subprocess. Patch by Lita Cho.
- bpo-21704 [https://bugs.python.org/issue?@action=redirect&bpo=21704]: Fix build error for _multiprocessing when semaphores are not available. Patch by Arfrever Frehtes Taifersar Arahesis.
- bpo-20173 [https://bugs.python.org/issue?@action=redirect&bpo=20173]: Convert sha1, sha256, sha512 and md5 to ArgumentClinic. Patch by Vajrasky Kok.
- Fix repr(_socket.socket) on Windows 64-bit: don’t fail with OverflowError on closed socket. repr(socket.socket)

already works fine.

- bpo-22033 [https://bugs.python.org/issue?@action=redirect&bpo=22033]: Reprs of most Python implemented classes now contain actual class name instead of hardcoded one.
- bpo-21947 [https://bugs.python.org/issue?@action=redirect&bpo=21947]: The dis module can now disassemble generator-iterator objects based on their gi_code attribute. Patch by Clement Rouault.
- bpo-16133 [https://bugs.python.org/issue?@action=redirect&bpo=16133]: The asynchat.async_chat.handle_read() method now ignores BlockingIOError exceptions.
- bpo-22044 [https://bugs.python.org/issue?@action=redirect&bpo=22044]: Fixed premature DECREF in call_tzinfo_method. Patch by Tom Flanagan.
- bpo-19884 [https://bugs.python.org/issue?@action=redirect&bpo=19884]: readline: Disable the meta modifier key if stdout is not a terminal to not write the ANSI sequence `"\033[1034h"` into stdout. This sequence is used on some terminal (ex: TERM=xterm-256color”) to enable support of 8 bit characters.
- bpo-4350 [https://bugs.python.org/issue?@action=redirect&bpo=4350]: Removed a number of out-of-dated and non-working for a long time Tkinter methods.
- bpo-6167 [https://bugs.python.org/issue?@action=redirect&bpo=6167]: Scrollbar.activate() now returns the name of active element if the argument is not specified. Scrollbar.set() now always accepts only 2 arguments.
- bpo-15275 [https://bugs.python.org/issue?@action=redirect&bpo=15275]: Clean up and speed up the ntpath module.
- bpo-21888 [https://bugs.python.org/issue?@action=redirect&bpo=21888]: plistlib’s load() and loads() now work if the fmt parameter is specified.

- bpo-22032 [https://bugs.python.org/issue?@action=redirect&bpo=22032]: __qualname__ instead of __name__ is now always used to format fully qualified class names of Python implemented classes.
- bpo-22031 [https://bugs.python.org/issue?@action=redirect&bpo=22031]: Reprs now always use hexadecimal format with the “0x” prefix when contain an id in form “ at 0x…”.
- bpo-22018 [https://bugs.python.org/issue?@action=redirect&bpo=22018]: signal.set_wakeup_fd() now raises an OSError instead of a ValueError on `fstat()` failure.
- bpo-21044 [https://bugs.python.org/issue?@action=redirect&bpo=21044]: tarfile.open() now handles fileobj with an integer ‘name’ attribute. Based on patch by Antoine Pietri.
- bpo-21966 [https://bugs.python.org/issue?@action=redirect&bpo=21966]: Respect -q command-line option when code module is ran.
- bpo-19076 [https://bugs.python.org/issue?@action=redirect&bpo=19076]: Don’t pass the redundant ‘file’ argument to self.error().
- bpo-16382 [https://bugs.python.org/issue?@action=redirect&bpo=16382]: Improve exception message of warnings.warn() for bad category. Initial patch by Phil Elson.
- bpo-21932 [https://bugs.python.org/issue?@action=redirect&bpo=21932]: os.read() now uses a **`Py_ssize_t()`** type instead of int for the size to support reading more than 2 GB at once. On Windows, the size is truncated to INT_MAX. As any call to os.read(), the OS may read less bytes than the number of requested bytes.
- bpo-21942 [https://bugs.python.org/issue?@action=redirect&bpo=21942]: Fixed source file viewing in pydoc’s server mode on Windows.

- bpo-11259 [https://bugs.python.org/issue?@action=redirect&bpo=11259]: asynchat.async_chat().set_terminator() now raises a ValueError if the number of received bytes is negative.
- bpo-12523 [https://bugs.python.org/issue?@action=redirect&bpo=12523]: asynchat.async_chat.push() now raises a TypeError if it doesn’t get a bytes string
- bpo-21707 [https://bugs.python.org/issue?@action=redirect&bpo=21707]: Add missing kwonlyargcount argument to ModuleFinder.replace_paths_in_code().
- bpo-20639 [https://bugs.python.org/issue?@action=redirect&bpo=20639]: calling Path.with_suffix(‘’) allows removing the suffix again. Patch by July Tikhonov.
- bpo-21714 [https://bugs.python.org/issue?@action=redirect&bpo=21714]: Disallow the construction of invalid paths using Path.with_name(). Original patch by Antony Lee.
- bpo-15014 [https://bugs.python.org/issue?@action=redirect&bpo=15014]: Added ‘auth’ method to smtplib to make implementing auth mechanisms simpler, and used it internally in the login method.
- bpo-21151 [https://bugs.python.org/issue?@action=redirect&bpo=21151]: Fixed a segfault in the winreg module when `None` is passed as a `REG_BINARY` value to SetValueEx. Patch by John Ehresman.
- bpo-21090 [https://bugs.python.org/issue?@action=redirect&bpo=21090]: io.FileIO.readall() does not ignore I/O errors anymore. Before, it ignored I/O errors if at least the first C call read() succeed.
- bpo-5800 [https://bugs.python.org/issue?@action=redirect&bpo=5800]: headers parameter of wsgiref.headers.Headers is now optional. Initial patch by Pablo Torres Navarrete and SilentGhost.
- bpo-21781 [https://bugs.python.org/issue?@action=redirect&bpo=21781]: ssl.RAND_add() now supports strings longer than 2 GB.

- bpo-21679 [https://bugs.python.org/issue?@action=redirect&bpo=21679]: Prevent extraneous fstat() calls during open(). Patch by Bohuslav Kabrda.
- bpo-21863 [https://bugs.python.org/issue?@action=redirect&bpo=21863]: cProfile now displays the module name of C extension functions, in addition to their own name.
- bpo-11453 [https://bugs.python.org/issue?@action=redirect&bpo=11453]: asyncore: emit a ResourceWarning when an unclosed file_wrapper object is destroyed. The destructor now closes the file if needed. The close() method can now be called twice: the second call does nothing.
- bpo-21858 [https://bugs.python.org/issue?@action=redirect&bpo=21858]: Better handling of Python exceptions in the sqlite3 module.
- bpo-21476 [https://bugs.python.org/issue?@action=redirect&bpo=21476]: Make sure the email.parser.BytesParser TextIOWrapper is discarded after parsing, so the input file isn’t unexpectedly closed.
- bpo-20295 [https://bugs.python.org/issue?@action=redirect&bpo=20295]: imghdr now recognizes OpenEXR format images.
- bpo-21729 [https://bugs.python.org/issue?@action=redirect&bpo=21729]: Used the “with” statement in the dbm.dumb module to ensure files closing. Patch by Claudiu Popa.
- bpo-21491 [https://bugs.python.org/issue?@action=redirect&bpo=21491]: socketserver: Fix a race condition in child processes reaping.
- bpo-21719 [https://bugs.python.org/issue?@action=redirect&bpo=21719]: Added the `st_file_attributes` field to os.stat_result on Windows.
- bpo-21832 [https://bugs.python.org/issue?@action=redirect&bpo=21832]: Require named tuple inputs to be exact strings.

- bpo-21722 [https://bugs.python.org/issue?@action=redirect&bpo=21722]: The distutils “upload” command now exits with a non-zero return code when uploading fails. Patch by Martin Dengler.
- bpo-21723 [https://bugs.python.org/issue?@action=redirect&bpo=21723]: asyncio.Queue: support any type of number (ex: float) for the maximum size. Patch written by Vajrasky Kok.
- bpo-21711 [https://bugs.python.org/issue?@action=redirect&bpo=21711]: support for “site-python” directories has now been removed from the site module (it was deprecated in 3.4).
- bpo-17552 [https://bugs.python.org/issue?@action=redirect&bpo=17552]: new socket.sendfile() method allowing a file to be sent over a socket by using high-performance os.sendfile() on UNIX. Patch by Giampaolo Rodola’.
- bpo-18039 [https://bugs.python.org/issue?@action=redirect&bpo=18039]: dbm.dump.open() now always creates a new database when the flag has the value ‘n’. Patch by Claudiu Popa.
- bpo-21326 [https://bugs.python.org/issue?@action=redirect&bpo=21326]: Add a new is_closed() method to asyncio.BaseEventLoop. run_forever() and run_until_complete() methods of asyncio.BaseEventLoop now raise an exception if the event loop was closed.
- bpo-21766 [https://bugs.python.org/issue?@action=redirect&bpo=21766]: Prevent a security hole in CGIHTTPServer by URL unquoting paths before checking for a CGI script at that path.
- bpo-21310 [https://bugs.python.org/issue?@action=redirect&bpo=21310]: Fixed possible resource leak in failed open().
- bpo-21256 [https://bugs.python.org/issue?@action=redirect&bpo=21256]: Printout of keyword args should

be in deterministic order in a mock function call. This will help to write better doctests.

- bpo-21677 [https://bugs.python.org/issue?@action=redirect&bpo=21677]: Fixed chaining nonnormalized exceptions in io close() methods.
- bpo-11709 [https://bugs.python.org/issue?@action=redirect&bpo=11709]: Fix the pydoc.help function to not fail when sys.stdin is not a valid file.
- bpo-21515 [https://bugs.python.org/issue?@action=redirect&bpo=21515]: tempfile.TemporaryFile now uses os.O_TMPFILE flag is available.
- bpo-13223 [https://bugs.python.org/issue?@action=redirect&bpo=13223]: Fix pydoc.writedoc so that the HTML documentation for methods that use 'self' in the example code is generated correctly.
- bpo-21463 [https://bugs.python.org/issue?@action=redirect&bpo=21463]: In urllib.request, fix pruning of the FTP cache.
- bpo-21618 [https://bugs.python.org/issue?@action=redirect&bpo=21618]: The subprocess module could fail to close open fds that were inherited by the calling process and already higher than POSIX resource limits would otherwise allow. On systems with a functioning /proc/self/fd or /dev/fd interface the max is now ignored and all fds are closed.
- bpo-20383 [https://bugs.python.org/issue?@action=redirect&bpo=20383]: Introduce importlib.util.module_from_spec() as the preferred way to create a new module.
- bpo-21552 [https://bugs.python.org/issue?@action=redirect&bpo=21552]: Fixed possible integer overflow of too long string lengths in the tkinter module on 64-bit platforms.
- bpo-14315 [https://bugs.python.org/issue?@action=redirect&bpo=14315]: The zipfile module now ignores extra fields in the central directory that are too short to

be parsed instead of letting a struct.unpack error bubble up as this “bad data” appears in many real world zip files in the wild and is ignored by other zip tools.

- bpo-13742 [https://bugs.python.org/issue?@action=redirect&bpo=13742]: Added “key” and “reverse” parameters to heapq.merge(). (First draft of patch contributed by Simon Sapin.)
- bpo-21402 [https://bugs.python.org/issue?@action=redirect&bpo=21402]: tkinter.ttk now works when default root window is not set.
- bpo-3015 [https://bugs.python.org/issue?@action=redirect&bpo=3015]: `_tkinter.create()` now creates `tkapp` object with `wantobjects=1` by default.
- bpo-10203 [https://bugs.python.org/issue?@action=redirect&bpo=10203]: sqlite3.Row now truly supports sequence protocol. In particular it supports reverse() and negative indices. Original patch by Claudiu Popa.
- bpo-18807 [https://bugs.python.org/issue?@action=redirect&bpo=18807]: If copying (no symlinks) specified for a venv, then the python interpreter aliases (python, python3) are now created by copying rather than symlinking.
- bpo-20197 [https://bugs.python.org/issue?@action=redirect&bpo=20197]: Added support for the WebP image type in the imghdr module. Patch by Fabrice Aneche and Claudiu Popa.
- bpo-21513 [https://bugs.python.org/issue?@action=redirect&bpo=21513]: Speedup some properties of IP addresses (IPv4Address, IPv6Address) such as .is_private or .is_multicast.
- bpo-21137 [https://bugs.python.org/issue?@action=redirect&bpo=21137]: Improve the repr for threading.Lock() and its variants by showing the “locked” or “unlocked” status. Patch by Berker Peksag.
- bpo-21538 [https://bugs.python.org/issue?@action=redirect&bpo=21538]: The plistlib module now

supports loading of binary plist files when reference or offset size is not a power of two.

- bpo-21455 [https://bugs.python.org/issue?@action=redirect&bpo=21455]: Add a default backlog to socket.listen().
- bpo-21525 [https://bugs.python.org/issue?@action=redirect&bpo=21525]: Most Tkinter methods which accepted tuples now accept lists too.
- bpo-22166 [https://bugs.python.org/issue?@action=redirect&bpo=22166]: With the assistance of a new internal _codecs._forget_codec helping function, test_codecs now clears the encoding caches to avoid the appearance of a reference leak
- bpo-22236 [https://bugs.python.org/issue?@action=redirect&bpo=22236]: Tkinter tests now don’t reuse default root window. New root window is created for every test class.
- bpo-10744 [https://bugs.python.org/issue?@action=redirect&bpo=10744]: Fix **PEP 3118** [https://peps.python.org/pep-3118/] format strings on ctypes objects with a nontrivial shape.
- bpo-20826 [https://bugs.python.org/issue?@action=redirect&bpo=20826]: Optimize ipaddress.collapse_addresses().
- bpo-21487 [https://bugs.python.org/issue?@action=redirect&bpo=21487]: Optimize ipaddress.summarize_address_range() and ipaddress.{IPv4Network,IPv6Network}.subnets().
- bpo-21486 [https://bugs.python.org/issue?@action=redirect&bpo=21486]: Optimize parsing of netmasks in ipaddress.IPv4Network and ipaddress.IPv6Network.
- bpo-13916 [https://bugs.python.org/issue?@action=redirect&bpo=13916]: Disallowed the surrogatepass error handler for non UTF-* encodings.
- bpo-20998 [https://bugs.python.org/issue?@action=redirect&bpo=20998]: Fixed re.fullmatch() of repeated

single character pattern with ignore case. Original patch by Matthew Barnett.

- bpo-21075 [https://bugs.python.org/issue?@action=redirect&bpo=21075]: fileinput.FileInput now reads bytes from standard stream if binary mode is specified. Patch by Sam Kimbrel.
- bpo-19775 [https://bugs.python.org/issue?@action=redirect&bpo=19775]: Add a samefile() method to pathlib Path objects. Initial patch by Vajrasky Kok.
- bpo-21226 [https://bugs.python.org/issue?@action=redirect&bpo=21226]: Set up modules properly in PyImport_ExecCodeModuleObject (and friends).
- bpo-21398 [https://bugs.python.org/issue?@action=redirect&bpo=21398]: Fix a unicode error in the pydoc pager when the documentation contains characters not encodable to the stdout encoding.
- bpo-16531 [https://bugs.python.org/issue?@action=redirect&bpo=16531]: ipaddress.IPv4Network and ipaddress.IPv6Network now accept an (address, netmask) tuple argument, so as to easily construct network objects from existing addresses.
- bpo-21156 [https://bugs.python.org/issue?@action=redirect&bpo=21156]: importlib.abc.InspectLoader.source_to_code() is now a staticmethod.
- bpo-21424 [https://bugs.python.org/issue?@action=redirect&bpo=21424]: Simplified and optimized heaqp.nlargest() and nmsmallest() to make fewer tuple comparisons.
- bpo-21396 [https://bugs.python.org/issue?@action=redirect&bpo=21396]: Fix TextIOWrapper(…, write_through=True) to not force a flush() on the underlying binary stream. Patch by akira.
- bpo-18314 [https://bugs.python.org/issue?@action=redirect&bpo=18314]: Unlink now removes junctions on Windows. Patch by Kim Gräsman

- bpo-21088 [https://bugs.python.org/issue?@action=redirect&bpo=21088]: Bugfix for curses.window.addch() regression in 3.4.0. In porting to Argument Clinic, the first two arguments were reversed.
- bpo-21407 [https://bugs.python.org/issue?@action=redirect&bpo=21407]: _decimal: The module now supports function signatures.
- bpo-10650 [https://bugs.python.org/issue?@action=redirect&bpo=10650]: Remove the non-standard 'watchexp' parameter from the Decimal.quantize() method in the Python version. It had never been present in the C version.
- bpo-21469 [https://bugs.python.org/issue?@action=redirect&bpo=21469]: Reduced the risk of false positives in robotparser by checking to make sure that robots.txt has been read or does not exist prior to returning True in can_fetch().
- bpo-19414 [https://bugs.python.org/issue?@action=redirect&bpo=19414]: Have the OrderedDict mark deleted links as unusable. This gives an early failure if the link is deleted during iteration.
- bpo-21421 [https://bugs.python.org/issue?@action=redirect&bpo=21421]: Add __slots__ to the MappingViews ABC. Patch by Josh Rosenberg.
- bpo-21101 [https://bugs.python.org/issue?@action=redirect&bpo=21101]: Eliminate double hashing in the C speed-up code for collections.Counter().
- bpo-21321 [https://bugs.python.org/issue?@action=redirect&bpo=21321]: itertools.islice() now releases the reference to the source iterator when the slice is exhausted. Patch by Anton Afanasyev.
- bpo-21057 [https://bugs.python.org/issue?@action=redirect&bpo=21057]: TextIOWrapper now allows the underlying binary stream's read() or read1() method to return an arbitrary bytes-like object (such as a memoryview). Patch by Nikolaus Rath.

- bpo-20951 [https://bugs.python.org/issue?@action=redirect&bpo=20951]: SSLSocket.send() now raises either SSLWantReadError or SSLWantWriteError on a non-blocking socket if the operation would block. Previously, it would return 0. Patch by Nikolaus Rath.
- bpo-13248 [https://bugs.python.org/issue?@action=redirect&bpo=13248]: removed previously deprecated asyncore.dispatcher __getattr__ cheap inheritance hack.
- bpo-9815 [https://bugs.python.org/issue?@action=redirect&bpo=9815]: assertRaises now tries to clear references to local variables in the exception’s traceback.
- bpo-19940 [https://bugs.python.org/issue?@action=redirect&bpo=19940]: ssl.cert_time_to_seconds() now interprets the given time string in the UTC timezone (as specified in RFC 5280), not the local timezone.
- bpo-13204 [https://bugs.python.org/issue?@action=redirect&bpo=13204]: Calling sys.flags.__new__ would crash the interpreter, now it raises a TypeError.
- bpo-19385 [https://bugs.python.org/issue?@action=redirect&bpo=19385]: Make operations on a closed dbm.dumb database always raise the same exception.
- bpo-21207 [https://bugs.python.org/issue?@action=redirect&bpo=21207]: Detect when the os.urandom cached fd has been closed or replaced, and open it anew.
- bpo-21291 [https://bugs.python.org/issue?@action=redirect&bpo=21291]: subprocess’s Popen.wait() is now thread safe so that multiple threads may be calling wait() or poll() on a Popen instance at the same time without losing the Popen.returncode value.
- bpo-21127 [https://bugs.python.org/issue?@action=redirect&bpo=21127]: Path objects can now be instantiated from str subclass instances (such as `numpy.str_`).
- bpo-15002 [https://bugs.python.org/issue?@action=redirect&bpo=15002]: urllib.response object to use

_TemporaryFileWrapper (and _TemporaryFileCloser) facility. Provides a better way to handle file descriptor close. Patch contributed by Christian Theune.

- bpo-12220 [https://bugs.python.org/issue?@action=redirect&bpo=12220]: mindom now raises a custom ValueError indicating it doesn’t support spaces in URIs instead of letting a ‘split’ ValueError bubble up.
- bpo-21068 [https://bugs.python.org/issue?@action=redirect&bpo=21068]: The ssl.PROTOCOL* constants are now enum members.
- bpo-21276 [https://bugs.python.org/issue?@action=redirect&bpo=21276]: posixmodule: Don’t define USE_XATTRS on KFreeBSD and the Hurd.
- bpo-21262 [https://bugs.python.org/issue?@action=redirect&bpo=21262]: New method assert_not_called for Mock. It raises AssertionError if the mock has been called.
- bpo-21238 [https://bugs.python.org/issue?@action=redirect&bpo=21238]: New keyword argument `unsafe` to Mock. It raises `AttributeError` in case of an attribute startswith assert or assret.
- bpo-20896 [https://bugs.python.org/issue?@action=redirect&bpo=20896]: ssl.get_server_certificate() now uses PROTOCOL_SSLv23, not PROTOCOL_SSLv3, for maximum compatibility.
- bpo-21239 [https://bugs.python.org/issue?@action=redirect&bpo=21239]: patch.stopall() didn’t work deterministically when the same name was patched more than once.
- bpo-21203 [https://bugs.python.org/issue?@action=redirect&bpo=21203]: Updated fileConfig and dictConfig to remove inconsistencies. Thanks to Jure Koren for the patch.
- bpo-21222 [https://bugs.python.org/issue?@action=redirect&bpo=21222]: Passing name keyword argument to mock.create_autospec now works.

- bpo-21197 [https://bugs.python.org/issue?@action=redirect&bpo=21197]: Add lib64 -> lib symlink in venvs on 64-bit non-OS X POSIX.
- bpo-17498 [https://bugs.python.org/issue?@action=redirect&bpo=17498]: Some SMTP servers disconnect after certain errors, violating strict RFC conformance. Instead of losing the error code when we issue the subsequent RSET, smtplib now returns the error code and defers raising the SMTPServerDisconnected error until the next command is issued.
- bpo-17826 [https://bugs.python.org/issue?@action=redirect&bpo=17826]: setting an iterable side_effect on a mock function created by create_autospec now works. Patch by Kushal Das.
- bpo-7776 [https://bugs.python.org/issue?@action=redirect&bpo=7776]: Fix `Host:` header and reconnection when using http.client.HTTPConnection.set_tunnel(). Patch by Nikolaus Rath.
- bpo-20968 [https://bugs.python.org/issue?@action=redirect&bpo=20968]: unittest.mock.MagicMock now supports division. Patch by Johannes Baiter.
- bpo-21529 [https://bugs.python.org/issue?@action=redirect&bpo=21529]: Fix arbitrary memory access in JSONDecoder.raw_decode with a negative second parameter. Bug reported by Guido Vranken. (See also: **CVE 2014-4616** [https://www.cve.org/CVERecord?id=CVE-2014-4616])
- bpo-21169 [https://bugs.python.org/issue?@action=redirect&bpo=21169]: getpass now handles non-ascii characters that the input stream encoding cannot encode by re-encoding using the replace error handler.
- bpo-21171 [https://bugs.python.org/issue?@action=redirect&bpo=21171]: Fixed undocumented filter API of the rot13 codec. Patch by Berker Peksag.
- bpo-20539 [https://bugs.python.org/issue?@action=redirect&bpo=20539]: Improved math.factorial error

message for large positive inputs and changed exception type (OverflowError -> ValueError) for large negative inputs.

- bpo-21172 [https://bugs.python.org/issue?@action=redirect&bpo=21172]: isinstance check relaxed from dict to collections.Mapping.
- bpo-21155 [https://bugs.python.org/issue?@action=redirect&bpo=21155]: asyncio.EventLoop.create_unix_server() now raises a ValueError if path and sock are specified at the same time.
- bpo-21136 [https://bugs.python.org/issue?@action=redirect&bpo=21136]: Avoid unnecessary normalization of Fractions resulting from power and other operations. Patch by Raymond Hettinger.
- bpo-17621 [https://bugs.python.org/issue?@action=redirect&bpo=17621]: Introduce importlib.util.LazyLoader.
- bpo-21076 [https://bugs.python.org/issue?@action=redirect&bpo=21076]: signal module constants were turned into enums. Patch by Giampaolo Rodola’.
- bpo-20636 [https://bugs.python.org/issue?@action=redirect&bpo=20636]: Improved the repr of Tkinter widgets.
- bpo-19505 [https://bugs.python.org/issue?@action=redirect&bpo=19505]: The items, keys, and values views of OrderedDict now support reverse iteration using reversed().
- bpo-21149 [https://bugs.python.org/issue?@action=redirect&bpo=21149]: Improved thread-safety in logging cleanup during interpreter shutdown. Thanks to Devin Jeanpierre for the patch.
- bpo-21058 [https://bugs.python.org/issue?@action=redirect&bpo=21058]: Fix a leak of file descriptor in `tempfile.NamedTemporaryFile()`, close the file descriptor if `io.open()` fails

- bpo-21200 [https://bugs.python.org/issue?@action=redirect&bpo=21200]: Return None from pkgutil.get_loader() when __spec__ is missing.
- bpo-21013 [https://bugs.python.org/issue?@action=redirect&bpo=21013]: Enhance ssl.create_default_context() when used for server side sockets to provide better security by default.
- bpo-20145 [https://bugs.python.org/issue?@action=redirect&bpo=20145]: `assertRaisesRegex` and `assertWarnsRegex` now raise a `TypeError` if the second argument is not a string or compiled regex.
- bpo-20633 [https://bugs.python.org/issue?@action=redirect&bpo=20633]: Replace relative import by absolute import.
- bpo-20980 [https://bugs.python.org/issue?@action=redirect&bpo=20980]: Stop wrapping exception when using ThreadPool.
- bpo-21082 [https://bugs.python.org/issue?@action=redirect&bpo=21082]: In os.makedirs, do not set the process-wide umask. Note this changes behavior of makedirs when exist_ok=True.
- bpo-20990 [https://bugs.python.org/issue?@action=redirect&bpo=20990]: Fix issues found by pyflakes for multiprocessing.
- bpo-21015 [https://bugs.python.org/issue?@action=redirect&bpo=21015]: SSL contexts will now automatically select an elliptic curve for ECDH key exchange on OpenSSL 1.0.2 and later, and otherwise default to “prime256v1”.
- bpo-21000 [https://bugs.python.org/issue?@action=redirect&bpo=21000]: Improve the command-line interface of json.tool.
- bpo-20995 [https://bugs.python.org/issue?@action=redirect&bpo=20995]: Enhance default ciphers used by the ssl module to enable better security and prioritize perfect forward secrecy.

- bpo-20884 [https://bugs.python.org/issue?@action=redirect&bpo=20884]: Don’t assume that __file__ is defined on importlib.__init__.
- bpo-21499 [https://bugs.python.org/issue?@action=redirect&bpo=21499]: Ignore __builtins__ in several test_importlib.test_api tests.
- bpo-20627 [https://bugs.python.org/issue?@action=redirect&bpo=20627]: xmlrpc.client.ServerProxy is now a context manager.
- bpo-19165 [https://bugs.python.org/issue?@action=redirect&bpo=19165]: The formatter module now raises DeprecationWarning instead of PendingDeprecationWarning.
- bpo-13936 [https://bugs.python.org/issue?@action=redirect&bpo=13936]: Remove the ability of datetime.time instances to be considered false in boolean contexts.
- bpo-18931 [https://bugs.python.org/issue?@action=redirect&bpo=18931]: selectors module now supports /dev/poll on Solaris. Patch by Giampaolo Rodola’.
- bpo-19977 [https://bugs.python.org/issue?@action=redirect&bpo=19977]: When the `LC_TYPE` locale is the POSIX locale (`C` locale), **`sys.stdin`** and **`sys.stdout`** are now using the `surrogateescape` error handler, instead of the `strict` error handler.
- bpo-20574 [https://bugs.python.org/issue?@action=redirect&bpo=20574]: Implement incremental decoder for cp65001 code (Windows code page 65001, Microsoft UTF-8).
- bpo-20879 [https://bugs.python.org/issue?@action=redirect&bpo=20879]: Delay the initialization of encoding and decoding tables for base32, ascii85 and base85 codecs in the base64 module, and delay the initialization of the unquote_to_bytes() table of the urllib.parse module, to not waste memory if these modules are not used.

- bpo-19157 [https://bugs.python.org/issue?@action=redirect&bpo=19157]: Include the broadcast address in the usable hosts for IPv6 in ipaddress.
- bpo-11599 [https://bugs.python.org/issue?@action=redirect&bpo=11599]: When an external command (e.g. compiler) fails, distutils now prints out the whole command line (instead of just the command name) if the environment variable DISTUTILS_DEBUG is set.
- bpo-4931 [https://bugs.python.org/issue?@action=redirect&bpo=4931]: distutils should not produce unhelpful "error: None" messages anymore. distutils.util.grok_environment_error is kept but doc-deprecated.
- bpo-20875 [https://bugs.python.org/issue?@action=redirect&bpo=20875]: Prevent possible gzip "'read' is not defined" NameError. Patch by Claudiu Popa.
- bpo-11558 [https://bugs.python.org/issue?@action=redirect&bpo=11558]: `email.message.Message.attach` now returns a more useful error message if `attach` is called on a message for which `is_multipart` is False.
- bpo-20283 [https://bugs.python.org/issue?@action=redirect&bpo=20283]: RE pattern methods now accept the string keyword parameters as documented. The pattern and source keyword parameters are left as deprecated aliases.
- bpo-20778 [https://bugs.python.org/issue?@action=redirect&bpo=20778]: Fix modulefinder to work with bytecode-only modules.
- bpo-20791 [https://bugs.python.org/issue?@action=redirect&bpo=20791]: copy.copy() now doesn't make a copy when the input is a bytes object. Initial patch by Peter Otten.
- bpo-19748 [https://bugs.python.org/issue?@action=redirect&bpo=19748]: On AIX, time.mktime() now raises an OverflowError for year outsize range [1902; 2037].

- bpo-19573 [https://bugs.python.org/issue?@action=redirect&bpo=19573]: inspect.signature: Use enum for parameter kind constants.
- bpo-20726 [https://bugs.python.org/issue?@action=redirect&bpo=20726]: inspect.signature: Make Signature and Parameter picklable.
- bpo-17373 [https://bugs.python.org/issue?@action=redirect&bpo=17373]: Add inspect.Signature.from_callable method.
- bpo-20378 [https://bugs.python.org/issue?@action=redirect&bpo=20378]: Improve repr of inspect.Signature and inspect.Parameter.
- bpo-20816 [https://bugs.python.org/issue?@action=redirect&bpo=20816]: Fix inspect.getcallargs() to raise correct TypeError for missing keyword-only arguments. Patch by Jeremiah Lowin.
- bpo-20817 [https://bugs.python.org/issue?@action=redirect&bpo=20817]: Fix inspect.getcallargs() to fail correctly if more than 3 arguments are missing. Patch by Jeremiah Lowin.
- bpo-6676 [https://bugs.python.org/issue?@action=redirect&bpo=6676]: Ensure a meaningful exception is raised when attempting to parse more than one XML document per pyexpat xmlparser instance. (Original patches by Hirokazu Yamamoto and Amaury Forgeot d’Arc, with suggested wording by David Gutteridge)
- bpo-21117 [https://bugs.python.org/issue?@action=redirect&bpo=21117]: Fix inspect.signature to better support functools.partial. Due to the specifics of functools.partial implementation, positional-or-keyword arguments passed as keyword arguments become keyword-only.
- bpo-20334 [https://bugs.python.org/issue?@action=redirect&bpo=20334]: inspect.Signature and inspect.Parameter are now hashable. Thanks to Antony Lee for bug reports and suggestions.

- bpo-15916 [https://bugs.python.org/issue?@action=redirect&bpo=15916]: doctest.DocTestSuite returns an empty unittest.TestSuite instead of raising ValueError if it finds no tests
- bpo-21209 [https://bugs.python.org/issue?@action=redirect&bpo=21209]: Fix asyncio.tasks.CoroWrapper to workaround a bug in yield-from implementation in CPythons prior to 3.4.1.
- asyncio: Add gi_{frame,running,code} properties to CoroWrapper (upstream bpo-163 [https://bugs.python.org/issue?@action=redirect&bpo=163]).
- bpo-21311 [https://bugs.python.org/issue?@action=redirect&bpo=21311]: Avoid exception in _osx_support with non-standard compiler configurations. Patch by John Szakmeister.
- bpo-11571 [https://bugs.python.org/issue?@action=redirect&bpo=11571]: Ensure that the turtle window becomes the topmost window when launched on OS X.
- bpo-21801 [https://bugs.python.org/issue?@action=redirect&bpo=21801]: Validate that __signature__ is None or an instance of Signature.
- bpo-21923 [https://bugs.python.org/issue?@action=redirect&bpo=21923]: Prevent AttributeError in distutils.sysconfig.customize_compiler due to possible uninitialized _config_vars.
- bpo-21323 [https://bugs.python.org/issue?@action=redirect&bpo=21323]: Fix http.server to again handle scripts in CGI subdirectories, broken by the fix for security bpo-19435 [https://bugs.python.org/issue?@action=redirect&bpo=19435]. Patch by Zach Byrne.
- bpo-22733 [https://bugs.python.org/issue?@action=redirect&bpo=22733]: Fix ffi_prep_args not zero-extending argument values correctly on 64-bit Windows.
- bpo-23302 [https://bugs.python.org/issue?@action=redirect&bpo=23302]: Default to TCP_NODELAY=1 upon establishing an HTTPConnection. Removed use of

hard-coded MSS as it’s an optimization that’s no longer needed with Nagle disabled.

IDLE

- bpo-20577 [https://bugs.python.org/issue?@action=redirect&bpo=20577]: Configuration of the max line length for the FormatParagraph extension has been moved from the General tab of the Idle preferences dialog to the FormatParagraph tab of the Config Extensions dialog. Patch by Tal Einat.
- bpo-16893 [https://bugs.python.org/issue?@action=redirect&bpo=16893]: Update Idle doc chapter to match current Idle and add new information.
- bpo-3068 [https://bugs.python.org/issue?@action=redirect&bpo=3068]: Add Idle extension configuration dialog to Options menu. Changes are written to HOME/.idlerc/config-extensions.cfg. Original patch by Tal Einat.
- bpo-16233 [https://bugs.python.org/issue?@action=redirect&bpo=16233]: A module browser (File : Class Browser, Alt+C) requires an editor window with a filename. When Class Browser is requested otherwise, from a shell, output window, or ‘Untitled’ editor, Idle no longer displays an error box. It now pops up an Open Module box (Alt+M). If a valid name is entered and a module is opened, a corresponding browser is also opened.
- bpo-4832 [https://bugs.python.org/issue?@action=redirect&bpo=4832]: Save As to type Python files automatically adds .py to the name you enter (even if your system does not display it). Some systems automatically add .txt when type is Text files.
- bpo-21986 [https://bugs.python.org/issue?@action=redirect&bpo=21986]: Code objects are not normally pickled by the pickle module. To match this, they are no longer pickled when running under Idle.

- bpo-17390 [https://bugs.python.org/issue?@action=redirect&bpo=17390]: Adjust Editor window title; remove ‘Python’, move version to end.
- bpo-14105 [https://bugs.python.org/issue?@action=redirect&bpo=14105]: Idle debugger breakpoints no longer disappear when inserting or deleting lines.
- bpo-17172 [https://bugs.python.org/issue?@action=redirect&bpo=17172]: Turtledemo can now be run from Idle. Currently, the entry is on the Help menu, but it may move to Run. Patch by Ramchandra Apt and Lita Cho.
- bpo-21765 [https://bugs.python.org/issue?@action=redirect&bpo=21765]: Add support for non-ascii identifiers to HyperParser.
- bpo-21940 [https://bugs.python.org/issue?@action=redirect&bpo=21940]: Add unittest for WidgetRedirector. Initial patch by Saimadhav Heblikar.
- bpo-18592 [https://bugs.python.org/issue?@action=redirect&bpo=18592]: Add unittest for SearchDialogBase. Patch by Phil Webster.
- bpo-21694 [https://bugs.python.org/issue?@action=redirect&bpo=21694]: Add unittest for ParenMatch. Patch by Saimadhav Heblikar.
- bpo-21686 [https://bugs.python.org/issue?@action=redirect&bpo=21686]: add unittest for HyperParser. Original patch by Saimadhav Heblikar.
- bpo-12387 [https://bugs.python.org/issue?@action=redirect&bpo=12387]: Add missing upper(lower)case versions of default Windows key bindings for Idle so Caps Lock does not disable them. Patch by Roger Serwy.
- bpo-21695 [https://bugs.python.org/issue?@action=redirect&bpo=21695]: Closing a Find-in-files output window while the search is still in progress no longer closes Idle.
- bpo-18910 [https://bugs.python.org/issue?@action=redirect&bpo=18910]: Add unittest for textView. Patch

by Phil Webster.

- bpo-18292 [https://bugs.python.org/issue?@action=redirect&bpo=18292]: Add unittest for AutoExpand. Patch by Saihadhav Heblikar.
- bpo-18409 [https://bugs.python.org/issue?@action=redirect&bpo=18409]: Add unittest for AutoComplete. Patch by Phil Webster.
- bpo-21477 [https://bugs.python.org/issue?@action=redirect&bpo=21477]: htest.py - Improve framework, complete set of tests. Patches by Saimadhav Heblikar
- bpo-18104 [https://bugs.python.org/issue?@action=redirect&bpo=18104]: Add idlelib/idle_test/htest.py with a few sample tests to begin consolidating and improving human-validated tests of Idle. Change other files as needed to work with htest. Running the module as __main__ runs all tests.
- bpo-21139 [https://bugs.python.org/issue?@action=redirect&bpo=21139]: Change default paragraph width to 72, the **PEP 8** [https://peps.python.org/pep-0008/] recommendation.
- bpo-21284 [https://bugs.python.org/issue?@action=redirect&bpo=21284]: Paragraph reformat test passes after user changes reformat width.
- bpo-17654 [https://bugs.python.org/issue?@action=redirect&bpo=17654]: Ensure IDLE menus are customized properly on OS X for non-framework builds and for all variants of Tk.
- bpo-23180 [https://bugs.python.org/issue?@action=redirect&bpo=23180]: Rename IDLE “Windows” menu item to “Window”. Patch by Al Sweigart.

Build

- bpo-15506 [https://bugs.python.org/issue?@action=redirect&bpo=15506]: Use standard

PKG_PROG_PKG_CONFIG autoconf macro in the configure script.

- bpo-22935 [https://bugs.python.org/issue?@action=redirect&bpo=22935]: Allow the ssl module to be compiled if openssl doesn't support SSL 3.
- bpo-22592 [https://bugs.python.org/issue?@action=redirect&bpo=22592]: Drop support of the Borland C compiler to build Python. The distutils module still supports it to build extensions.
- bpo-22591 [https://bugs.python.org/issue?@action=redirect&bpo=22591]: Drop support of MS-DOS, especially of the DJGPP compiler (MS-DOS port of GCC).
- bpo-16537 [https://bugs.python.org/issue?@action=redirect&bpo=16537]: Check whether self.extensions is empty in setup.py. Patch by Jonathan Hosmer.
- bpo-22359 [https://bugs.python.org/issue?@action=redirect&bpo=22359]: Remove incorrect uses of recursive make. Patch by Jonas Wagner.
- bpo-21958 [https://bugs.python.org/issue?@action=redirect&bpo=21958]: Define HAVE_ROUND when building with Visual Studio 2013 and above. Patch by Zachary Turner.
- bpo-18093 [https://bugs.python.org/issue?@action=redirect&bpo=18093]: the programs that embed the CPython runtime are now in a separate "Programs" directory, rather than being kept in the Modules directory.
- bpo-15759 [https://bugs.python.org/issue?@action=redirect&bpo=15759]: "make suspicious", "make linkcheck" and "make doctest" in Doc/ now display special message when and only when there are failures.
- bpo-21141 [https://bugs.python.org/issue?@action=redirect&bpo=21141]: The Windows build process no longer attempts to find Perl, instead relying on OpenSSL source being configured and ready to build. The `PCbuild\build_ssl.py` script has been re-written and re-

named to `PCbuild\prepare_ssl.py`, and takes care of configuring OpenSSL source for both 32 and 64 bit platforms. OpenSSL sources obtained from svn.python.org will always be pre-configured and ready to build.

- bpo-21037 [https://bugs.python.org/issue?@action=redirect&bpo=21037]: Add a build option to enable AddressSanitizer support.
- bpo-19962 [https://bugs.python.org/issue?@action=redirect&bpo=19962]: The Windows build process now creates “python.bat” in the root of the source tree, which passes all arguments through to the most recently built interpreter.
- bpo-21285 [https://bugs.python.org/issue?@action=redirect&bpo=21285]: Refactor and fix curses configure check to always search in a ncursesw directory.
- bpo-15234 [https://bugs.python.org/issue?@action=redirect&bpo=15234]: For BerkeleyDB and Sqlite, only add the found library and include directories if they aren’t already being searched. This avoids an explicit runtime library dependency.
- bpo-17861 [https://bugs.python.org/issue?@action=redirect&bpo=17861]: Tools/scripts/generate_opcode_h.py automatically regenerates Include/opcode.h from Lib/opcode.py if the latter gets any change.
- bpo-20644 [https://bugs.python.org/issue?@action=redirect&bpo=20644]: OS X installer build support for documentation build changes in 3.4.1: assume externally supplied sphinx-build is available in /usr/bin.
- bpo-20022 [https://bugs.python.org/issue?@action=redirect&bpo=20022]: Eliminate use of deprecated bundlebuilder in OS X builds.
- bpo-15968 [https://bugs.python.org/issue?@action=redirect&bpo=15968]: Incorporated Tcl, Tk, and Tix

builds into the Windows build solution.
- bpo-17095 [https://bugs.python.org/issue?@action=redirect&bpo=17095]: Fix Modules/Setup *shared* support.
- bpo-21811 [https://bugs.python.org/issue?@action=redirect&bpo=21811]: Anticipated fixes to support OS X versions > 10.9.
- bpo-21166 [https://bugs.python.org/issue?@action=redirect&bpo=21166]: Prevent possible segfaults and other random failures of python –generate-posix-vars in pybuilddir.txt build target.
- bpo-18096 [https://bugs.python.org/issue?@action=redirect&bpo=18096]: Fix library order returned by python-config.
- bpo-17219 [https://bugs.python.org/issue?@action=redirect&bpo=17219]: Add library build dir for Python extension cross-builds.
- bpo-22919 [https://bugs.python.org/issue?@action=redirect&bpo=22919]: Windows build updated to support VC 14.0 (Visual Studio 2015), which will be used for the official release.
- bpo-21236 [https://bugs.python.org/issue?@action=redirect&bpo=21236]: Build _msi.pyd with cabinet.lib instead of fci.lib
- bpo-17128 [https://bugs.python.org/issue?@action=redirect&bpo=17128]: Use private version of OpenSSL for OS X 10.5+ installer.

C API

- bpo-14203 [https://bugs.python.org/issue?@action=redirect&bpo=14203]: Remove obsolete support for view==NULL in PyBuffer_FillInfo(), bytearray_getbuffer(), bytesiobuf_getbuffer() and array_buffer_getbuf(). All functions now raise BufferError in that case.

- bpo-22445 [https://bugs.python.org/issue?@action=redirect&bpo=22445]: PyBuffer_IsContiguous() now implements precise contiguity tests, compatible with NumPy’s NPY_RELAXED_STRIDES_CHECKING compilation flag. Previously the function reported false negatives for corner cases.
- bpo-22079 [https://bugs.python.org/issue?@action=redirect&bpo=22079]: PyType_Ready() now checks that statically allocated type has no dynamically allocated bases.
- bpo-22453 [https://bugs.python.org/issue?@action=redirect&bpo=22453]: Removed non-documented macro PyObject_REPR().
- bpo-18395 [https://bugs.python.org/issue?@action=redirect&bpo=18395]: Rename `_Py_char2wchar()` to **`Py_DecodeLocale()`**, rename `_Py_wchar2char()` to **`Py_EncodeLocale()`**, and document these functions.
- bpo-21233 [https://bugs.python.org/issue?@action=redirect&bpo=21233]: Add new C functions: PyMem_RawCalloc(), PyMem_Calloc(), PyObject_Calloc(), _PyObject_GC_Calloc(). bytes(int) is now using `calloc()` instead of `malloc()` for large objects which is faster and use less memory.
- bpo-20942 [https://bugs.python.org/issue?@action=redirect&bpo=20942]: PyImport_ImportFrozenModuleObject() no longer sets __file__ to match what importlib does; this affects _frozen_importlib as well as any module loaded using imp.init_frozen().

Documentation

- bpo-19548 [https://bugs.python.org/issue?@action=redirect&bpo=19548]: Update the codecs module documentation to better cover the distinction between text encodings and other codecs, together with other clarifications. Patch by Martin Panter.

- bpo-22394 [https://bugs.python.org/issue?@action=redirect&bpo=22394]: Doc/Makefile now supports `make venv PYTHON=../python` to create a venv for generating the documentation, e.g., `make html PYTHON=venv/bin/python3`.
- bpo-21514 [https://bugs.python.org/issue?@action=redirect&bpo=21514]: The documentation of the json module now refers to new JSON RFC 7159 instead of obsoleted RFC 4627.
- bpo-21777 [https://bugs.python.org/issue?@action=redirect&bpo=21777]: The binary sequence methods on bytes and bytearray are now documented explicitly, rather than assuming users will be able to derive the expected behaviour from the behaviour of the corresponding str methods.
- bpo-6916 [https://bugs.python.org/issue?@action=redirect&bpo=6916]: undocument deprecated asynchat.fifo class.
- bpo-17386 [https://bugs.python.org/issue?@action=redirect&bpo=17386]: Expanded functionality of the `Doc/make.bat` script to make it much more comparable to `Doc/Makefile`.
- bpo-21312 [https://bugs.python.org/issue?@action=redirect&bpo=21312]: Update the thread_foobar.h template file to include newer threading APIs. Patch by Jack McCracken.
- bpo-21043 [https://bugs.python.org/issue?@action=redirect&bpo=21043]: Remove the recommendation for specific CA organizations and to mention the ability to load the OS certificates.
- bpo-20765 [https://bugs.python.org/issue?@action=redirect&bpo=20765]: Add missing documentation for PurePath.with_name() and PurePath.with_suffix().
- bpo-19407 [https://bugs.python.org/issue?@action=redirect&bpo=19407]: New package installation and distribution guides based on the Python Packaging

Authority tools. Existing guides have been retained as legacy links from the distutils docs, as they still contain some required reference material for tool developers that isn't recorded anywhere else.

- bpo-19697 [https://bugs.python.org/issue?@action=redirect&bpo=19697]: Document cases where __main__.__spec__ is None.

Tests

- bpo-18982 [https://bugs.python.org/issue?@action=redirect&bpo=18982]: Add tests for CLI of the calendar module.
- bpo-19548 [https://bugs.python.org/issue?@action=redirect&bpo=19548]: Added some additional checks to test_codecs to ensure that statements in the updated documentation remain accurate. Patch by Martin Panter.
- bpo-22838 [https://bugs.python.org/issue?@action=redirect&bpo=22838]: All test_re tests now work with unittest test discovery.
- bpo-22173 [https://bugs.python.org/issue?@action=redirect&bpo=22173]: Update lib2to3 tests to use unittest test discovery.
- bpo-16000 [https://bugs.python.org/issue?@action=redirect&bpo=16000]: Convert test_curses to use unittest.
- bpo-21456 [https://bugs.python.org/issue?@action=redirect&bpo=21456]: Skip two tests in test_urllib2net.py if _ssl module not present. Patch by Remi Pointel.
- bpo-20746 [https://bugs.python.org/issue?@action=redirect&bpo=20746]: Fix test_pdb to run in refleak mode (-R). Patch by Xavier de Gaye.
- bpo-22060 [https://bugs.python.org/issue?@action=redirect&bpo=22060]: test_ctypes has been

somewhat cleaned up and simplified; it now uses unittest test discovery to find its tests.

- bpo-22104 [https://bugs.python.org/issue?@action=redirect&bpo=22104]: regrtest.py no longer holds a reference to the suite of tests loaded from test modules that don't define test_main().
- bpo-22111 [https://bugs.python.org/issue?@action=redirect&bpo=22111]: Assorted cleanups in test_imaplib. Patch by Milan Oberkirch.
- bpo-22002 [https://bugs.python.org/issue?@action=redirect&bpo=22002]: Added `load_package_tests` function to test.support and used it to implement/augment test discovery in test_asyncio, test_email, test_importlib, test_json, and test_tools.
- bpo-21976 [https://bugs.python.org/issue?@action=redirect&bpo=21976]: Fix test_ssl to accept LibreSSL version strings. Thanks to William Orr.
- bpo-21918 [https://bugs.python.org/issue?@action=redirect&bpo=21918]: Converted test_tools from a module to a package containing separate test files for each tested script.
- bpo-9554 [https://bugs.python.org/issue?@action=redirect&bpo=9554]: Use modern unittest features in test_argparse. Initial patch by Denver Coneybeare and Radu Voicilas.
- bpo-20155 [https://bugs.python.org/issue?@action=redirect&bpo=20155]: Changed HTTP method names in failing tests in test_httpservers so that packet filtering software (specifically Windows Base Filtering Engine) does not interfere with the transaction semantics expected by the tests.
- bpo-19493 [https://bugs.python.org/issue?@action=redirect&bpo=19493]: Refactored the ctypes test package to skip tests explicitly rather than silently.
- bpo-18492 [https://bugs.python.org/issue?@action=redirect&bpo=18492]: All resources are now allowed when tests are not run by regrtest.py.

- bpo-21634 [https://bugs.python.org/issue?@action=redirect&bpo=21634]: Fix pystone micro-benchmark: use floor division instead of true division to benchmark integers instead of floating-point numbers. Set pystone version to 1.2. Patch written by Lennart Regebro.
- bpo-21605 [https://bugs.python.org/issue?@action=redirect&bpo=21605]: Added tests for Tkinter images.
- bpo-21493 [https://bugs.python.org/issue?@action=redirect&bpo=21493]: Added test for ntpath.expanduser(). Original patch by Claudiu Popa.
- bpo-19925 [https://bugs.python.org/issue?@action=redirect&bpo=19925]: Added tests for the spwd module. Original patch by Vajrasky Kok.
- bpo-21522 [https://bugs.python.org/issue?@action=redirect&bpo=21522]: Added Tkinter tests for Listbox.itemconfigure(), PanedWindow.paneconfigure(), and Menu.entryconfigure().
- bpo-17756 [https://bugs.python.org/issue?@action=redirect&bpo=17756]: Fix test_code test when run from the installed location.
- bpo-17752 [https://bugs.python.org/issue?@action=redirect&bpo=17752]: Fix distutils tests when run from the installed location.
- bpo-18604 [https://bugs.python.org/issue?@action=redirect&bpo=18604]: Consolidated checks for GUI availability. All platforms now at least check whether Tk can be instantiated when the GUI resource is requested.
- bpo-21275 [https://bugs.python.org/issue?@action=redirect&bpo=21275]: Fix a socket test on KFreeBSD.
- bpo-21223 [https://bugs.python.org/issue?@action=redirect&bpo=21223]: Pass test_site/test_startup_imports when some of the extensions are built as builtins.
- bpo-20635 [https://bugs.python.org/issue?@action=redirect&bpo=20635]: Added tests for Tk geometry managers.

- Add test case for freeze.
- bpo-20743 [https://bugs.python.org/issue?@action=redirect&bpo=20743]: Fix a reference leak in test_tcl.
- bpo-21097 [https://bugs.python.org/issue?@action=redirect&bpo=21097]: Move test_namespace_pkgs into test_importlib.
- bpo-21503 [https://bugs.python.org/issue?@action=redirect&bpo=21503]: Use test_both() consistently in test_importlib.
- bpo-20939 [https://bugs.python.org/issue?@action=redirect&bpo=20939]: Avoid various network test failures due to new redirect of http://www.python.org/ to https://www.python.org: use http://www.example.com instead.
- bpo-20668 [https://bugs.python.org/issue?@action=redirect&bpo=20668]: asyncio tests no longer rely on tests.txt file. (Patch by Vajrasky Kok)
- bpo-21093 [https://bugs.python.org/issue?@action=redirect&bpo=21093]: Prevent failures of ctypes test_macholib on OS X if a copy of libz exists in $HOME/lib or /usr/local/lib.
- bpo-22770 [https://bugs.python.org/issue?@action=redirect&bpo=22770]: Prevent some Tk segfaults on OS X when running gui tests.
- bpo-23211 [https://bugs.python.org/issue?@action=redirect&bpo=23211]: Workaround test_logging failure on some OS X 10.6 systems.
- bpo-23345 [https://bugs.python.org/issue?@action=redirect&bpo=23345]: Prevent test_ssl failures with large OpenSSL patch level values (like 0.9.8zc).

Tools/Demos

- bpo-22314 [https://bugs.python.org/issue?@action=redirect&bpo=22314]: pydoc now works when the LINES environment variable is set.

- bpo-22615 [https://bugs.python.org/issue?@action=redirect&bpo=22615]: Argument Clinic now supports the “type” argument for the int converter. This permits using the int converter with enums and typedefs.
- bpo-20076 [https://bugs.python.org/issue?@action=redirect&bpo=20076]: The makelocalealias.py script no longer ignores UTF-8 mapping.
- bpo-20079 [https://bugs.python.org/issue?@action=redirect&bpo=20079]: The makelocalealias.py script now can parse the SUPPORTED file from glibc sources and supports command line options for source paths.
- bpo-22201 [https://bugs.python.org/issue?@action=redirect&bpo=22201]: Command-line interface of the zipfile module now correctly extracts ZIP files with directory entries. Patch by Ryan Wilson.
- bpo-22120 [https://bugs.python.org/issue?@action=redirect&bpo=22120]: For functions using an unsigned integer return converter, Argument Clinic now generates a cast to that type for the comparison to -1 in the generated code. (This suppresses a compilation warning.)
- bpo-18974 [https://bugs.python.org/issue?@action=redirect&bpo=18974]: Tools/scripts/diff.py now uses argparse instead of optparse.
- bpo-21906 [https://bugs.python.org/issue?@action=redirect&bpo=21906]: Make Tools/scripts/md5sum.py work in Python 3. Patch by Zachary Ware.
- bpo-21629 [https://bugs.python.org/issue?@action=redirect&bpo=21629]: Fix Argument Clinic’s “–converters” feature.
- Add support for `yield from` to 2to3.
- Add support for the **PEP 465** [https://peps.python.org/pep-0465/] matrix multiplication operator to 2to3.
- bpo-16047 [https://bugs.python.org/issue?@action=redirect&bpo=16047]: Fix module exception list and __file__ handling in freeze. Patch by Meador Inge.

- bpo-11824 [https://bugs.python.org/issue?@action=redirect&bpo=11824]: Consider ABI tags in freeze. Patch by Meador Inge.
- bpo-20535 [https://bugs.python.org/issue?@action=redirect&bpo=20535]: PYTHONWARNING no longer affects the run_tests.py script. Patch by Arfrever Frehtes Taifersar Arahesis.

Windows

- bpo-23260 [https://bugs.python.org/issue?@action=redirect&bpo=23260]: Update Windows installer
- The bundled version of Tcl/Tk has been updated to 8.6.3. The most visible result of this change is the addition of new native file dialogs when running on Windows Vista or newer. See Tcl/Tk’s TIP 432 for more information. Also, this version of Tcl/Tk includes support for Windows 10.
- bpo-17896 [https://bugs.python.org/issue?@action=redirect&bpo=17896]: The Windows build scripts now expect external library sources to be in `PCbuild\..\externals` rather than `PCbuild\..\..`.
- bpo-17717 [https://bugs.python.org/issue?@action=redirect&bpo=17717]: The Windows build scripts now use a copy of NASM pulled from svn.python.org to build OpenSSL.
- bpo-21907 [https://bugs.python.org/issue?@action=redirect&bpo=21907]: Improved the batch scripts provided for building Python.
- bpo-22644 [https://bugs.python.org/issue?@action=redirect&bpo=22644]: The bundled version of OpenSSL has been updated to 1.0.1j.
- bpo-10747 [https://bugs.python.org/issue?@action=redirect&bpo=10747]: Use versioned labels in the Windows start menu. Patch by Olive Kilburn.

- bpo-22980 [https://bugs.python.org/issue?@action=redirect&bpo=22980]: .pyd files with a version and platform tag (for example, “.cp35-win32.pyd”) will now be loaded in preference to those without tags.

(For information about older versions, consult the HISTORY file.)

The Python Tutorial

Tip

This tutorial is designed for *programmers* that are new to the Python language, **not** *beginners* who are new to programming.

Python is an easy to learn, powerful programming language. It has efficient high-level data structures and a simple but effective approach to object-oriented programming. Python’s elegant syntax and dynamic typing, together with its interpreted nature, make it an ideal language for scripting and rapid application development in many areas on most platforms.

The Python interpreter and the extensive standard library are freely available in source or binary form for all major platforms from the Python website, https://www.python.org/, and may be freely distributed. The same site also contains distributions of and pointers to many free third party Python modules, programs and tools, and additional documentation.

The Python interpreter is easily extended with new functions and data types implemented in C or C++ (or other languages callable from C). Python is also suitable as an extension language for customizable applications.

This tutorial introduces the reader informally to the basic concepts and features of the Python language and system. Be aware that it expects you to have a basic understanding of programming in general. It helps to have a Python interpreter handy for hands-on experience, but all examples are self-contained, so the tutorial can be read off-line as well.

For a description of standard objects and modules, see The Python Standard Library. The Python Language Reference gives a more formal definition of the language. To write extensions in C or C++, read Extending and Embedding the Python Interpreter and Python/C API Reference Manual. There are also several books covering Python in depth.

This tutorial does not attempt to be comprehensive and cover every single feature, or even every commonly used feature. Instead, it introduces many of Python’s most noteworthy features, and will give you a good idea of the language’s flavor and style. After reading it, you will be able to read and write Python modules and programs, and you will be ready to learn more about the various Python library modules described in The Python Standard Library.

The Glossary is also worth going through.

- 1. Whetting Your Appetite
- 2. Using the Python Interpreter
  - 2.1. Invoking the Interpreter
    - 2.1.1. Argument Passing
    - 2.1.2. Interactive Mode
  - 2.2. The Interpreter and Its Environment
    - 2.2.1. Source Code Encoding
- 3. An Informal Introduction to Python
  - 3.1. Using Python as a Calculator
    - 3.1.1. Numbers
    - 3.1.2. Text
    - 3.1.3. Lists
  - 3.2. First Steps Towards Programming
- 4. More Control Flow Tools

    - 4.1. `if` Statements
    - 4.2. `for` Statements
    - 4.3. The `range()` Function
    - 4.4. `break` and `continue` Statements
    - 4.5. `else` Clauses on Loops
    - 4.6. `pass` Statements
    - 4.7. `match` Statements
    - 4.8. Defining Functions
    - 4.9. More on Defining Functions
        - 4.9.1. Default Argument Values
        - 4.9.2. Keyword Arguments
        - 4.9.3. Special parameters
            - 4.9.3.1. Positional-or-Keyword Arguments
            - 4.9.3.2. Positional-Only Parameters
            - 4.9.3.3. Keyword-Only Arguments
            - 4.9.3.4. Function Examples
            - 4.9.3.5. Recap
        - 4.9.4. Arbitrary Argument Lists
        - 4.9.5. Unpacking Argument Lists
        - 4.9.6. Lambda Expressions
        - 4.9.7. Documentation Strings
        - 4.9.8. Function Annotations
    - 4.10. Intermezzo: Coding Style
- 5. Data Structures
    - 5.1. More on Lists
        - 5.1.1. Using Lists as Stacks
        - 5.1.2. Using Lists as Queues
        - 5.1.3. List Comprehensions
        - 5.1.4. Nested List Comprehensions
    - 5.2. The `del` statement
    - 5.3. Tuples and Sequences
    - 5.4. Sets
    - 5.5. Dictionaries
    - 5.6. Looping Techniques
    - 5.7. More on Conditions
    - 5.8. Comparing Sequences and Other Types

- 6. Modules
    - 6.1. More on Modules
        - 6.1.1. Executing modules as scripts
        - 6.1.2. The Module Search Path
        - 6.1.3. “Compiled” Python files
    - 6.2. Standard Modules
    - 6.3. The `dir()` Function
    - 6.4. Packages
        - 6.4.1. Importing * From a Package
        - 6.4.2. Intra-package References
        - 6.4.3. Packages in Multiple Directories
- 7. Input and Output
    - 7.1. Fancier Output Formatting
        - 7.1.1. Formatted String Literals
        - 7.1.2. The String format() Method
        - 7.1.3. Manual String Formatting
        - 7.1.4. Old string formatting
    - 7.2. Reading and Writing Files
        - 7.2.1. Methods of File Objects
        - 7.2.2. Saving structured data with `json`
- 8. Errors and Exceptions
    - 8.1. Syntax Errors
    - 8.2. Exceptions
    - 8.3. Handling Exceptions
    - 8.4. Raising Exceptions
    - 8.5. Exception Chaining
    - 8.6. User-defined Exceptions
    - 8.7. Defining Clean-up Actions
    - 8.8. Predefined Clean-up Actions
    - 8.9. Raising and Handling Multiple Unrelated Exceptions
    - 8.10. Enriching Exceptions with Notes
- 9. Classes
    - 9.1. A Word About Names and Objects
    - 9.2. Python Scopes and Namespaces
        - 9.2.1. Scopes and Namespaces Example

  - 9.3. A First Look at Classes
    - 9.3.1. Class Definition Syntax
    - 9.3.2. Class Objects
    - 9.3.3. Instance Objects
    - 9.3.4. Method Objects
    - 9.3.5. Class and Instance Variables
  - 9.4. Random Remarks
  - 9.5. Inheritance
    - 9.5.1. Multiple Inheritance
  - 9.6. Private Variables
  - 9.7. Odds and Ends
  - 9.8. Iterators
  - 9.9. Generators
  - 9.10. Generator Expressions
- 10. Brief Tour of the Standard Library
  - 10.1. Operating System Interface
  - 10.2. File Wildcards
  - 10.3. Command Line Arguments
  - 10.4. Error Output Redirection and Program Termination
  - 10.5. String Pattern Matching
  - 10.6. Mathematics
  - 10.7. Internet Access
  - 10.8. Dates and Times
  - 10.9. Data Compression
  - 10.10. Performance Measurement
  - 10.11. Quality Control
  - 10.12. Batteries Included
- 11. Brief Tour of the Standard Library — Part II
  - 11.1. Output Formatting
  - 11.2. Templating
  - 11.3. Working with Binary Data Record Layouts
  - 11.4. Multi-threading
  - 11.5. Logging
  - 11.6. Weak References
  - 11.7. Tools for Working with Lists

  - 11.8. Decimal Floating-Point Arithmetic
- 12. Virtual Environments and Packages
  - 12.1. Introduction
  - 12.2. Creating Virtual Environments
  - 12.3. Managing Packages with pip
- 13. What Now?
- 14. Interactive Input Editing and History Substitution
  - 14.1. Tab Completion and History Editing
  - 14.2. Alternatives to the Interactive Interpreter
- 15. Floating-Point Arithmetic: Issues and Limitations
  - 15.1. Representation Error
- 16. Appendix
  - 16.1. Interactive Mode
    - 16.1.1. Error Handling
    - 16.1.2. Executable Python Scripts
    - 16.1.3. The Interactive Startup File
    - 16.1.4. The Customization Modules

1. Whetting Your Appetite

If you do much work on computers, eventually you find that there’s some task you’d like to automate. For example, you may wish to perform a search-and-replace over a large number of text files, or rename and rearrange a bunch of photo files in a complicated way. Perhaps you’d like to write a small custom database, or a specialized GUI application, or a simple game.

If you’re a professional software developer, you may have to work with several C/C++/Java libraries but find the usual write/compile/test/re-compile cycle is too slow. Perhaps you’re writing a test suite for such a library and find writing the testing code a tedious task. Or maybe you’ve written a program that could use an extension language, and you don’t want to design and implement a whole new language for your application.

Python is just the language for you.

You could write a Unix shell script or Windows batch files for some of these tasks, but shell scripts are best at moving around files and changing text data, not well-suited for GUI applications or games. You could write a C/C++/Java program, but it can take a lot of development time to get even a first-draft program. Python is simpler to use, available on Windows, macOS, and Unix operating systems, and will help you get the job done more quickly.

Python is simple to use, but it is a real programming language, offering much more structure and support for large programs than shell scripts or batch files can offer. On the other hand, Python also offers much more error checking than C, and, being a *very-high-level language*, it has high-level data types built in, such as flexible arrays and dictionaries. Because of its more general data types Python is applicable to a much larger problem domain than Awk or even Perl, yet many things are at least as easy in Python as in those languages.

Python allows you to split your program into modules that can be reused in other Python programs. It comes with a large collection of standard modules that you can use as the basis of your programs — or as examples to start learning to program in Python. Some of these modules provide things like file I/O, system calls, sockets, and even interfaces to graphical user interface toolkits like Tk.

Python is an interpreted language, which can save you considerable time during program development because no compilation and linking is necessary. The interpreter can be used interactively, which makes it easy to experiment with features of the language, to write throw-away programs, or to test functions during bottom-up program development. It is also a handy desk calculator.

Python enables programs to be written compactly and readably. Programs written in Python are typically much shorter than equivalent C, C++, or Java programs, for several reasons:

- the high-level data types allow you to express complex operations in a single statement;
- statement grouping is done by indentation instead of beginning and ending brackets;
- no variable or argument declarations are necessary.

Python is *extensible*: if you know how to program in C it is easy to add a new built-in function or module to the interpreter, either to perform critical operations at maximum speed, or to link Python programs to libraries that may only be available in binary form (such as a vendor-specific graphics library). Once you are really hooked, you can link the Python interpreter into an application written in C and use it as an extension or command language for that application.

By the way, the language is named after the BBC show “Monty Python’s Flying Circus” and has nothing to do with reptiles. Making references to Monty Python skits in documentation is not only allowed, it is encouraged!

Now that you are all excited about Python, you’ll want to examine it in some more detail. Since the best way to learn a language is to use it, the tutorial invites you to play with the Python interpreter as you read.

In the next chapter, the mechanics of using the interpreter are explained. This is rather mundane information, but essential for trying out the examples shown later.

The rest of the tutorial introduces various features of the Python language and system through examples, beginning with simple expressions, statements and data types, through functions and modules, and finally touching upon advanced concepts like exceptions and user-defined classes.

2. Using the Python Interpreter

2.1. Invoking the Interpreter

The Python interpreter is usually installed as `/usr/local/bin/python3.14` on those machines where it is available; putting `/usr/local/bin` in your Unix shell’s search path makes it possible to start it by typing the command:

```
python3.14
```

to the shell. [[1]] Since the choice of the directory where the interpreter lives is an installation option, other places are possible; check with your local Python guru or system administrator. (E.g., `/usr/local/python` is a popular alternative location.)

On Windows machines where you have installed Python from the Microsoft Store, the `python3.14` command will be available. If you have the py.exe launcher installed, you can use the `py` command. See Python install manager for other ways to launch Python.

Typing an end-of-file character (`Control-D` on Unix, `Control-Z` on Windows) at the primary prompt causes the interpreter to exit with a zero exit status. If that doesn’t work, you can exit the interpreter by typing the following command: `quit()`.

The interpreter’s line-editing features include interactive editing, history substitution and code completion on most systems. Perhaps the quickest check to see whether command line editing is supported is typing a word in on the

Python prompt, then pressing Left arrow (or `Control-b`). If the cursor moves, you have command line editing; see Appendix Interactive Input Editing and History Substitution for an introduction to the keys. If nothing appears to happen, or if a sequence like `^[[D` or `^B` appears, command line editing isn't available; you'll only be able to use backspace to remove characters from the current line.

The interpreter operates somewhat like the Unix shell: when called with standard input connected to a tty device, it reads and executes commands interactively; when called with a file name argument or with a file as standard input, it reads and executes a *script* from that file.

A second way of starting the interpreter is `python -c command [arg] ...`, which executes the statement(s) in *command*, analogous to the shell's **`-c`** option. Since Python statements often contain spaces or other characters that are special to the shell, it is usually advised to quote *command* in its entirety.

Some Python modules are also useful as scripts. These can be invoked using `python -m module [arg] ...`, which executes the source file for *module* as if you had spelled out its full name on the command line.

When a script file is used, it is sometimes useful to be able to run the script and enter interactive mode afterwards. This can be done by passing **`-i`** before the script.

All command line options are described in Command line and environment.

2.1.1. Argument Passing

When known to the interpreter, the script name and additional arguments thereafter are turned into a list of strings and assigned to the `argv` variable in the `sys` module. You can access this list by executing `import sys`. The length of the list is at least one; when no script and no arguments are given, `sys.argv[0]` is an empty string. When the script name is given as `'-'` (meaning standard input), `sys.argv[0]` is set to `'-'`. When **`-c`** *command* is used, `sys.argv[0]` is set to `'-c'`. When **`-m`** *module* is used, `sys.argv[0]` is set to the full name of the located module. Options found after **`-c`** *command* or **`-m`** *module* are not consumed by the Python interpreter's option processing but left in `sys.argv` for the command or module to handle.

2.1.2. Interactive Mode

When commands are read from a tty, the interpreter is said to be in *interactive mode*. In this mode it prompts for the next command with the *primary prompt*, usually three greater-than signs (`>>>`); for continuation lines it prompts with the *secondary prompt*, by default three dots (`...`). The interpreter prints a welcome message stating its version number and a copyright notice before printing the first prompt:

```
$ python3.14
Python 3.14 (default, April 4 2024, 09:25:04)
[GCC 10.2.0] on linux
Type "help", "copyright", "credits" or "license" for more
information.
>>>
```

Continuation lines are needed when entering a multi-line construct. As an example, take a look at this **`if`** statement:

```
>>> the_world_is_flat = True
>>> if the_world_is_flat:
...     print("Be careful not to fall off!")
```

```
...
Be careful not to fall off!
```

For more on interactive mode, see Interactive Mode.

2.2. The Interpreter and Its Environment

2.2.1. Source Code Encoding

By default, Python source files are treated as encoded in UTF-8. In that encoding, characters of most languages in the world can be used simultaneously in string literals, identifiers and comments — although the standard library only uses ASCII characters for identifiers, a convention that any portable code should follow. To display all these characters properly, your editor must recognize that the file is UTF-8, and it must use a font that supports all the characters in the file.

To declare an encoding other than the default one, a special comment line should be added as the *first* line of the file. The syntax is as follows:

```
# -*- coding: encoding -*-
```

where *encoding* is one of the valid **`codecs`** supported by Python.

For example, to declare that Windows-1252 encoding is to be used, the first line of your source code file should be:

```
# -*- coding: cp1252 -*-
```

One exception to the *first line* rule is when the source code starts with a UNIX “shebang” line. In this case, the encoding declaration should be added as the second line of the file. For example:

```
#!/usr/bin/env python3
# -*- coding: cp1252 -*-
```

Footnotes

[1]

On Unix, the Python 3.x interpreter is by default not installed with the executable named `python`, so that it does not conflict with a simultaneously installed Python 2.x executable.

3. An Informal Introduction to Python

In the following examples, input and output are distinguished by the presence or absence of prompts (>>> and ...): to repeat the example, you must type everything after the prompt, when the prompt appears; lines that do not begin with a prompt are output from the interpreter. Note that a secondary prompt on a line by itself in an example means you must type a blank line; this is used to end a multi-line command.

You can use the “Copy” button (it appears in the upper-right corner when hovering over or tapping a code example), which strips prompts and omits output, to copy and paste the input lines into your interpreter.

Many of the examples in this manual, even those entered at the interactive prompt, include comments. Comments in Python start with the hash character, `#`, and extend to the end of the physical line. A comment may appear at the start of a line or following whitespace or code, but not within a string literal. A hash character within a string literal is just a hash character. Since comments are to clarify code and are not interpreted by Python, they may be omitted when typing in examples.

Some examples:

```
# this is the first comment
spam = 1  # and this is the second comment
          # ... and now a third!
text = "# This is not a comment because it's inside quotes."
```

3.1. Using Python as a Calculator

Let’s try some simple Python commands. Start the interpreter and wait for the primary prompt, `>>>`. (It shouldn’t take long.)

3.1.1. Numbers

The interpreter acts as a simple calculator: you can type an expression into it and it will write the value. Expression syntax is straightforward: the operators `+`, `-`, `*` and `/` can be used to perform arithmetic; parentheses (`()`) can be used for grouping. For example:

```
>>> 2 + 2
4
>>> 50 - 5*6
20
>>> (50 - 5*6) / 4
5.0
>>> 8 / 5  # division always returns a floating-point number
1.6
```

The integer numbers (e.g. `2`, `4`, `20`) have type **`int`**, the ones with a fractional part (e.g. `5.0`, `1.6`) have type **`float`**. We will see more about numeric types later in the tutorial.

Division (`/`) always returns a float. To do floor division and get an integer result you can use the `//` operator; to calculate the remainder you can use `%`:

```
>>> 17 / 3  # classic division returns a float
5.666666666666667
>>>
>>> 17 // 3  # floor division discards the fractional part
5
>>> 17 % 3  # the % operator returns the remainder of the
division
```

```
2
>>> 5 * 3 + 2  # floored quotient * divisor + remainder
17
```

With Python, it is possible to use the `**` operator to calculate powers [[1]]:

```
>>> 5 ** 2  # 5 squared
25
>>> 2 ** 7  # 2 to the power of 7
128
```

The equal sign (`=`) is used to assign a value to a variable. Afterwards, no result is displayed before the next interactive prompt:

```
>>> width = 20
>>> height = 5 * 9
>>> width * height
900
```

If a variable is not “defined” (assigned a value), trying to use it will give you an error:

```
>>> n  # try to access an undefined variable
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'n' is not defined
```

There is full support for floating point; operators with mixed type operands convert the integer operand to floating point:

```
>>> 4 * 3.75 - 1
14.0
```

In interactive mode, the last printed expression is assigned to the variable `_`. This means that when you are using Python as a desk calculator, it is somewhat easier to continue calculations, for example:

```
>>> tax = 12.5 / 100
>>> price = 100.50
>>> price * tax
12.5625
>>> price + _
113.0625
>>> round(_, 2)
113.06
```

This variable should be treated as read-only by the user. Don’t explicitly assign a value to it — you would create an independent local variable with the same name masking the built-in variable with its magic behavior.

In addition to **`int`** and **`float`**, Python supports other types of numbers, such as **`Decimal`** and **`Fraction`**. Python also has built-in support for complex numbers, and uses the `j` or `J` suffix to indicate the imaginary part (e.g. `3+5j`).

3.1.2. Text

Python can manipulate text (represented by type **`str`**, so-called “strings”) as well as numbers. This includes characters “`!`”, words “`rabbit`”, names “`Paris`”, sentences “`Got your back.`”, etc. “`Yay! :)`”. They can be enclosed in single quotes (`'...'`) or double quotes (`"..."`) with the same result [[2]].

```
>>> 'spam eggs'  # single quotes
'spam eggs'
>>> "Paris rabbit got your back :)! Yay!"  # double quotes
'Paris rabbit got your back :)! Yay!'
>>> '1975'  # digits and numerals enclosed in quotes are also
strings
'1975'
```

To quote a quote, we need to “escape” it, by preceding it with `\`. Alternatively, we can use the other type of quotation marks:

```
>>> 'doesn\'t'  # use \' to escape the single quote...
"doesn't"
>>> "doesn't"  # ...or use double quotes instead
"doesn't"
>>> '"Yes," they said.'
'"Yes," they said.'
>>> "\"Yes,\" they said."
'"Yes," they said.'
>>> '"Isn\'t," they said.'
'"Isn\'t," they said.'
```

In the Python shell, the string definition and output string can look different. The **`print()`** function produces a more readable output, by omitting the enclosing quotes and by printing escaped and special characters:

```
>>> s = 'First line.\nSecond line.'  # \n means newline
>>> s  # without print(), special characters are included in
the string
'First line.\nSecond line.'
>>> print(s)  # with print(), special characters are
interpreted, so \n produces new line
First line.
Second line.
```

If you don’t want characters prefaced by `\` to be interpreted as special characters, you can use *raw strings* by adding an `r` before the first quote:

```
>>> print('C:\some\name')  # here \n means newline!
C:\some
ame
>>> print(r'C:\some\name')  # note the r before the quote
C:\some\name
```

There is one subtle aspect to raw strings: a raw string may not end in an odd number of `\` characters; see the FAQ entry for more information and workarounds.

String literals can span multiple lines. One way is using triple-quotes: `"""..."""` or `'''...'''`. End-of-line characters are automatically included in the string, but it’s possible to prevent this by adding a `\` at the end of the line. In the following example, the initial newline is not included:

```
>>> print("""\
... Usage: thingy [OPTIONS]
...      -h                        Display this usage message
...      -H hostname               Hostname to connect to
... """)
Usage: thingy [OPTIONS]
     -h                        Display this usage message
     -H hostname               Hostname to connect to

>>>
```

Strings can be concatenated (glued together) with the `+` operator, and repeated with `*`:

```
>>> # 3 times 'un', followed by 'ium'
>>> 3 * 'un' + 'ium'
'unununium'
```

Two or more *string literals* (i.e. the ones enclosed between quotes) next to each other are automatically concatenated.

```
>>> 'Py' 'thon'
'Python'
```

This feature is particularly useful when you want to break long strings:

```
>>> text = ('Put several strings within parentheses '
...         'to have them joined together.')
>>> text
'Put several strings within parentheses to have them joined
together.'
```

This only works with two literals though, not with variables or expressions:

```
>>> prefix = 'Py'
>>> prefix 'thon'  # can't concatenate a variable and a string
literal
  File "<stdin>", line 1
    prefix 'thon'
           ^^^^^^
SyntaxError: invalid syntax
>>> ('un' * 3) 'ium'
  File "<stdin>", line 1
    ('un' * 3) 'ium'
               ^^^^^
SyntaxError: invalid syntax
```

If you want to concatenate variables or a variable and a literal, use `+`:

```
>>> prefix + 'thon'
'Python'
```

Strings can be *indexed* (subscripted), with the first character having index 0. There is no separate character type; a character is simply a string of size one:

```
>>> word = 'Python'
>>> word[0]  # character in position 0
'P'
>>> word[5]  # character in position 5
'n'
```

Indices may also be negative numbers, to start counting from the right:

```
>>> word[-1]  # last character
'n'
>>> word[-2]  # second-last character
'o'
>>> word[-6]
'P'
```

Note that since -0 is the same as 0, negative indices start from -1.

In addition to indexing, *slicing* is also supported. While indexing is used to obtain individual characters, *slicing* allows you to obtain a substring:

```
>>> word[0:2]  # characters from position 0 (included) to 2
(excluded)
'Py'
>>> word[2:5]  # characters from position 2 (included) to 5
(excluded)
'tho'
```

Slice indices have useful defaults; an omitted first index defaults to zero, an omitted second index defaults to the size of the string being sliced.

```
>>> word[:2]   # character from the beginning to position 2
(excluded)
'Py'
>>> word[4:]   # characters from position 4 (included) to the
end
'on'
>>> word[-2:]  # characters from the second-last (included) to
the end
'on'
```

Note how the start is always included, and the end always excluded. This makes sure that `s[:i] + s[i:]` is always equal to `s`:

```
>>> word[:2] + word[2:]
'Python'
>>> word[:4] + word[4:]
'Python'
```

One way to remember how slices work is to think of the indices as pointing *between* characters, with the left edge of the first character numbered 0. Then the right edge of the last character of a string of *n* characters has index *n*, for example:

```
 +---+---+---+---+---+---+
 | P | y | t | h | o | n |
 +---+---+---+---+---+---+
 0   1   2   3   4   5   6
-6  -5  -4  -3  -2  -1
```

The first row of numbers gives the position of the indices 0…6 in the string; the second row gives the corresponding negative indices. The slice from *i* to *j* consists of all characters between the edges labeled *i* and *j*, respectively.

For non-negative indices, the length of a slice is the difference of the indices, if both are within bounds. For example, the length of `word[1:3]` is 2.

Attempting to use an index that is too large will result in an error:

```
>>> word[42]  # the word only has 6 characters
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: string index out of range
```

However, out of range slice indexes are handled gracefully when used for slicing:

```
>>> word[4:42]
'on'
>>> word[42:]
''
```

Python strings cannot be changed — they are immutable. Therefore, assigning to an indexed position in the string results in an error:

```
>>> word[0] = 'J'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'str' object does not support item assignment
>>> word[2:] = 'py'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'str' object does not support item assignment
```

If you need a different string, you should create a new one:

```
>>> 'J' + word[1:]
'Jython'
>>> word[:2] + 'py'
'Pypy'
```

The built-in function `len()` returns the length of a string:

```
>>> s = 'supercalifragilisticexpialidocious'
>>> len(s)
34
```

See also

Text Sequence Type — str
Strings are examples of *sequence types*, and support the common operations supported by such types.

String Methods
Strings support a large number of methods for basic transformations and searching.

f-strings
String literals that have embedded expressions.

Format String Syntax
Information about string formatting with `str.format()`.

printf-style String Formatting
The old formatting operations invoked when strings are the left operand of the `%` operator are described in more

detail here.

3.1.3. Lists

Python knows a number of *compound* data types, used to group together other values. The most versatile is the *list*, which can be written as a list of comma-separated values (items) between square brackets. Lists might contain items of different types, but usually the items all have the same type.

```
>>> squares = [1, 4, 9, 16, 25]
>>> squares
[1, 4, 9, 16, 25]
```

Like strings (and all other built-in sequence types), lists can be indexed and sliced:

```
>>> squares[0]  # indexing returns the item
1
>>> squares[-1]
25
>>> squares[-3:]  # slicing returns a new list
[9, 16, 25]
```

Lists also support operations like concatenation:

```
>>> squares + [36, 49, 64, 81, 100]
[1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
```

Unlike strings, which are immutable, lists are a mutable type, i.e. it is possible to change their content:

```
>>> cubes = [1, 8, 27, 65, 125]  # something's wrong here
>>> 4 ** 3  # the cube of 4 is 64, not 65!
64
>>> cubes[3] = 64  # replace the wrong value
>>> cubes
[1, 8, 27, 64, 125]
```

You can also add new items at the end of the list, by using the **`list.append()`** *method* (we will see more about methods later):

```
>>> cubes.append(216)  # add the cube of 6
>>> cubes.append(7 ** 3)  # and the cube of 7
>>> cubes
[1, 8, 27, 64, 125, 216, 343]
```

Simple assignment in Python never copies data. When you assign a list to a variable, the variable refers to the *existing list*. Any changes you make to the list through one variable will be seen through all other variables that refer to it.:

```
>>> rgb = ["Red", "Green", "Blue"]
>>> rgba = rgb
>>> id(rgb) == id(rgba)  # they reference the same object
True
>>> rgba.append("Alph")
>>> rgb
["Red", "Green", "Blue", "Alph"]
```

All slice operations return a new list containing the requested elements. This means that the following slice returns a shallow copy of the list:

```
>>> correct_rgba = rgba[:]
>>> correct_rgba[-1] = "Alpha"
>>> correct_rgba
["Red", "Green", "Blue", "Alpha"]
>>> rgba
["Red", "Green", "Blue", "Alph"]
```

Assignment to slices is also possible, and this can even change the size of the list or clear it entirely:

```
>>> letters = ['a', 'b', 'c', 'd', 'e', 'f', 'g']
>>> letters
['a', 'b', 'c', 'd', 'e', 'f', 'g']
>>> # replace some values
>>> letters[2:5] = ['C', 'D', 'E']
>>> letters
['a', 'b', 'C', 'D', 'E', 'f', 'g']
>>> # now remove them
>>> letters[2:5] = []
>>> letters
['a', 'b', 'f', 'g']
>>> # clear the list by replacing all the elements with an
empty list
>>> letters[:] = []
>>> letters
[]
```

The built-in function `len()` also applies to lists:

```
>>> letters = ['a', 'b', 'c', 'd']
>>> len(letters)
4
```

It is possible to nest lists (create lists containing other lists), for example:

```
>>> a = ['a', 'b', 'c']
>>> n = [1, 2, 3]
>>> x = [a, n]
>>> x
[['a', 'b', 'c'], [1, 2, 3]]
>>> x[0]
['a', 'b', 'c']
>>> x[0][1]
'b'
```

3.2. First Steps Towards Programming

Of course, we can use Python for more complicated tasks than adding two and two together. For instance, we can write an initial sub-sequence of the Fibonacci series [https://en.wikipedia.org/wiki/Fibonacci_sequence] as follows:

```
>>> # Fibonacci series:
>>> # the sum of two elements defines the next
>>> a, b = 0, 1
>>> while a < 10:
...     print(a)
...     a, b = b, a+b
...
0
1
1
2
3
5
8
```

This example introduces several new features.

- The first line contains a *multiple assignment*: the variables `a` and `b` simultaneously get the new values 0 and 1. On the last line this is used again, demonstrating that the expressions on the right-hand side are all evaluated first before any of the assignments take place. The right-hand side expressions are evaluated from the left to the right.

- The **`while`** loop executes as long as the condition (here: `a < 10`) remains true. In Python, like in C, any non-zero integer value is true; zero is false. The condition may also be a string or list value, in fact any sequence; anything with a non-zero length is true, empty sequences are false. The test used in the example is a simple comparison. The standard comparison operators are written the same as in C: `<` (less than), `>` (greater than), `==` (equal to), `<=` (less than or equal to), `>=` (greater than or equal to) and `!=` (not equal to).

- The *body* of the loop is *indented*: indentation is Python’s way of grouping statements. At the interactive prompt, you have to type a tab or space(s) for each indented line. In practice you will prepare more complicated input for Python with a text editor; all decent text editors have an auto-indent facility. When a compound statement is entered interactively, it must be followed by a blank line to indicate completion (since the parser cannot guess when you have typed the last line). Note that each line within a basic block must be indented by the same amount.

- The `print()` function writes the value of the argument(s) it is given. It differs from just writing the expression you want to write (as we did earlier in the calculator examples) in the way it handles multiple arguments, floating-point quantities, and strings. Strings are printed without quotes, and a space is inserted between items, so you can format things nicely, like this:

  ```
  >>> i = 256*256
  >>> print('The value of i is', i)
  The value of i is 65536
  ```

  The keyword argument *end* can be used to avoid the newline after the output, or end the output with a different string:

  ```
  >>> a, b = 0, 1
  >>> while a < 1000:
  ...     print(a, end=',')
  ...     a, b = b, a+b
  ...
  0,1,1,2,3,5,8,13,21,34,55,89,144,233,377,610,987,
  ```

Footnotes

[1]

Since `**` has higher precedence than `-`, `-3**2` will be interpreted as `-(3**2)` and thus result in `-9`. To avoid this and

get `9`, you can use `(-3)**2`.
[2]

Unlike other languages, special characters such as `\n` have the same meaning with both single (`'...'`) and double (`"..."`) quotes. The only difference between the two is that within single quotes you don’t need to escape `"` (but you have to escape `\'`) and vice versa.

4. More Control Flow Tools

As well as the `while` statement just introduced, Python uses a few more that we will encounter in this chapter.

4.1. `if` Statements

Perhaps the most well-known statement type is the `if` statement. For example:

```
>>> x = int(input("Please enter an integer: "))
Please enter an integer: 42
>>> if x < 0:
...     x = 0
...     print('Negative changed to zero')
... elif x == 0:
...     print('Zero')
... elif x == 1:
...     print('Single')
... else:
...     print('More')
...
More
```

There can be zero or more `elif` parts, and the `else` part is optional. The keyword ‘**`elif`**’ is short for ‘else if’, and is useful to avoid excessive indentation. An **`if`** … **`elif`** … **`elif`** … sequence is a substitute for the `switch` or `case` statements found in other languages.

If you’re comparing the same value to several constants, or checking for specific types or attributes, you may also find the **`match`** statement useful. For more details see match Statements.

4.2. `for` Statements

The `for` statement in Python differs a bit from what you may be used to in C or Pascal. Rather than always iterating over an arithmetic progression of numbers (like in Pascal), or giving the user the ability to define both the iteration step and halting condition (as C), Python’s **`for`** statement iterates over the items of any sequence (a list or a string), in the order that they appear in the sequence. For example (no pun intended):

```
>>> # Measure some strings:
>>> words = ['cat', 'window', 'defenestrate']
>>> for w in words:
...     print(w, len(w))
...
cat 3
window 6
defenestrate 12
```

Code that modifies a collection while iterating over that same collection can be tricky to get right. Instead, it is usually more straight-forward to loop over a copy of the collection or to create a new collection:

```
# Create a sample collection
users = {'Hans': 'active', 'Éléonore': 'inactive', '景太郎':
'active'}

# Strategy:  Iterate over a copy
for user, status in users.copy().items():
    if status == 'inactive':
        del users[user]

# Strategy:  Create a new collection
active_users = {}
for user, status in users.items():
    if status == 'active':
        active_users[user] = status
```

4.3. The `range()` Function

If you do need to iterate over a sequence of numbers, the built-in function `range()` comes in handy. It generates arithmetic progressions:

```
>>> for i in range(5):
...     print(i)
...
0
1
2
3
4
```

The given end point is never part of the generated sequence; `range(10)` generates 10 values, the legal indices for items of a sequence of length 10. It is possible to let the range start at another number, or to specify a different increment (even negative; sometimes this is called the ‘step’):

```
>>> list(range(5, 10))
[5, 6, 7, 8, 9]

>>> list(range(0, 10, 3))
[0, 3, 6, 9]

>>> list(range(-10, -100, -30))
[-10, -40, -70]
```

To iterate over the indices of a sequence, you can combine `range()` and `len()` as follows:

```
>>> a = ['Mary', 'had', 'a', 'little', 'lamb']
>>> for i in range(len(a)):
...     print(i, a[i])
...
0 Mary
1 had
```

```
2 a
3 little
4 lamb
```

In most such cases, however, it is convenient to use the **`enumerate()`** function, see Looping Techniques.

A strange thing happens if you just print a range:

```
>>> range(10)
range(0, 10)
```

In many ways the object returned by **`range()`** behaves as if it is a list, but in fact it isn't. It is an object which returns the successive items of the desired sequence when you iterate over it, but it doesn't really make the list, thus saving space.

We say such an object is iterable, that is, suitable as a target for functions and constructs that expect something from which they can obtain successive items until the supply is exhausted. We have seen that the **`for`** statement is such a construct, while an example of a function that takes an iterable is **`sum()`**:

```
>>> sum(range(4))  # 0 + 1 + 2 + 3
6
```

Later we will see more functions that return iterables and take iterables as arguments. In chapter Data Structures, we will discuss in more detail about **`list()`**.

4.4. `break` and `continue` Statements

The **`break`** statement breaks out of the innermost enclosing **`for`** or **`while`** loop:

```
>>> for n in range(2, 10):
...     for x in range(2, n):
```

```
...         if n % x == 0:
...             print(f"{n} equals {x} * {n//x}")
...             break
...
4 equals 2 * 2
6 equals 2 * 3
8 equals 2 * 4
9 equals 3 * 3
```

The `continue` statement continues with the next iteration of the loop:

```
>>> for num in range(2, 10):
...     if num % 2 == 0:
...         print(f"Found an even number {num}")
...         continue
...     print(f"Found an odd number {num}")
...
Found an even number 2
Found an odd number 3
Found an even number 4
Found an odd number 5
Found an even number 6
Found an odd number 7
Found an even number 8
Found an odd number 9
```

4.5. `else` Clauses on Loops

In a **`for`** or **`while`** loop the **`break`** statement may be paired with an **`else`** clause. If the loop finishes without executing the **`break`**, the **`else`** clause executes.

In a `for` loop, the **`else`** clause is executed after the loop finishes its final iteration, that is, if no break occurred.

In a `while` loop, it’s executed after the loop’s condition becomes false.

In either kind of loop, the **`else`** clause is **not** executed if the loop was terminated by a `break`. Of course, other ways of ending the loop early, such as a `return` or a raised exception, will also skip execution of the `else` clause.

This is exemplified in the following **`for`** loop, which searches for prime numbers:

```
>>> for n in range(2, 10):
...     for x in range(2, n):
...         if n % x == 0:
...             print(n, 'equals', x, '*', n//x)
...             break
...     else:
...         # loop fell through without finding a factor
...         print(n, 'is a prime number')
...
2 is a prime number
3 is a prime number
4 equals 2 * 2
5 is a prime number
6 equals 2 * 3
7 is a prime number
8 equals 2 * 4
9 equals 3 * 3
```

(Yes, this is the correct code. Look closely: the `else` clause belongs to the `for` loop, **not** the `if` statement.)

One way to think of the else clause is to imagine it paired with the `if` inside the loop. As the loop executes, it will run a sequence like if/if/if/else. The `if` is inside the loop, encountered a number of times. If the condition is ever true, a `break` will happen. If the condition is never true, the `else` clause outside the loop will execute.

When used with a loop, the `else` clause has more in common with the `else` clause of a `try` statement than it does with that of `if` statements: a `try` statement’s `else` clause runs when no exception occurs, and a loop's `else` clause runs when no `break` occurs. For more on the `try` statement and exceptions, see Handling Exceptions.

4.6. `pass` Statements

The **`pass`** statement does nothing. It can be used when a statement is required syntactically but the program requires no action. For example:

```
>>> while True:
...     pass  # Busy-wait for keyboard interrupt (Ctrl+C)
...
```

This is commonly used for creating minimal classes:

```
>>> class MyEmptyClass:
...     pass
...
```

Another place **`pass`** can be used is as a place-holder for a function or conditional body when you are working on new code, allowing you to keep thinking at a more abstract level. The **`pass`** is silently ignored:

```
>>> def initlog(*args):
...     pass   # Remember to implement this!
...
```

For this last case, many people use the ellipsis literal `...` instead of `pass`. This use has no special meaning to Python, and is not part of the language definition (you could use any constant expression here), but `...` is used conventionally as a placeholder body as well. See The Ellipsis Object.

4.7. `match` Statements

A `match` statement takes an expression and compares its value to successive patterns given as one or more case blocks. This is superficially similar to a switch statement in C, Java or JavaScript (and many other languages), but it’s more similar to pattern matching in languages like Rust or Haskell. Only the first pattern that matches gets executed and it can also extract components (sequence elements or object attributes) from the value into variables. If no case matches, none of the branches is executed.

The simplest form compares a subject value against one or more literals:

```
def http_error(status):
    match status:
        case 400:
            return "Bad request"
        case 404:
            return "Not found"
        case 418:
            return "I'm a teapot"
        case _:
            return "Something's wrong with the internet"
```

Note the last block: the “variable name” `_` acts as a *wildcard* and never fails to match.

You can combine several literals in a single pattern using `|` (“or”):

```
case 401 | 403 | 404:
    return "Not allowed"
```

Patterns can look like unpacking assignments, and can be used to bind variables:

```
# point is an (x, y) tuple
match point:
    case (0, 0):
        print("Origin")
    case (0, y):
        print(f"Y={y}")
    case (x, 0):
        print(f"X={x}")
    case (x, y):
        print(f"X={x}, Y={y}")
    case _:
        raise ValueError("Not a point")
```

Study that one carefully! The first pattern has two literals, and can be thought of as an extension of the literal pattern shown above. But the next two patterns combine a literal and a variable, and the variable *binds* a value from the subject (`point`). The fourth pattern captures two values, which makes it conceptually similar to the unpacking assignment `(x, y) = point`.

If you are using classes to structure your data you can use the class name followed by an argument list resembling a constructor, but with the ability to capture attributes into variables:

```
class Point:
    def __init__(self, x, y):
        self.x = x
        self.y = y

def where_is(point):
    match point:
        case Point(x=0, y=0):
            print("Origin")
        case Point(x=0, y=y):
            print(f"Y={y}")
        case Point(x=x, y=0):
            print(f"X={x}")
        case Point():
            print("Somewhere else")
        case _:
            print("Not a point")
```

You can use positional parameters with some builtin classes that provide an ordering for their attributes (e.g. dataclasses). You can also define a specific position for attributes in patterns by setting the `__match_args__` special attribute in your classes. If it’s set to (“x”, “y”), the following patterns are all equivalent (and all bind the `y` attribute to the `var` variable):

```
Point(1, var)
Point(1, y=var)
Point(x=1, y=var)
Point(y=var, x=1)
```

A recommended way to read patterns is to look at them as an extended form of what you would put on the left of an assignment, to understand which variables would be set to what. Only the standalone names (like `var` above) are assigned to by a match statement. Dotted names (like `foo.bar`), attribute names (the `x=` and `y=` above) or class names (recognized by the “(…)” next to them like `Point` above) are never assigned to.

Patterns can be arbitrarily nested. For example, if we have a short list of Points, with `__match_args__` added, we could match it like this:

```
class Point:
    __match_args__ = ('x', 'y')
    def __init__(self, x, y):
        self.x = x
        self.y = y

match points:
    case []:
        print("No points")
    case [Point(0, 0)]:
        print("The origin")
    case [Point(x, y)]:
        print(f"Single point {x}, {y}")
    case [Point(0, y1), Point(0, y2)]:
        print(f"Two on the Y axis at {y1}, {y2}")
    case _:
        print("Something else")
```

We can add an `if` clause to a pattern, known as a “guard”. If the guard is false, `match` goes on to try the next case block. Note that value capture happens before the guard is evaluated:

```
match point:
    case Point(x, y) if x == y:
        print(f"Y=X at {x}")
    case Point(x, y):
        print(f"Not on the diagonal")
```

Several other key features of this statement:

- Like unpacking assignments, tuple and list patterns have exactly the same meaning and actually match arbitrary sequences. An important exception is that they don’t match iterators or strings.

- Sequence patterns support extended unpacking: `[x, y, *rest]` and `(x, y, *rest)` work similar to unpacking assignments. The name after `*` may also be `_`, so `(x, y, *_)` matches a sequence of at least two items without binding the remaining items.

- Mapping patterns: `{"bandwidth": b, "latency": l}` captures the `"bandwidth"` and `"latency"` values from a dictionary. Unlike sequence patterns, extra keys are ignored. An unpacking like `**rest` is also supported. (But `**_` would be redundant, so it is not allowed.)

- Subpatterns may be captured using the `as` keyword:

  ```
  case (Point(x1, y1), Point(x2, y2) as p2): ...
  ```

will capture the second element of the input as `p2` (as long as the input is a sequence of two points)

- Most literals are compared by equality, however the singletons `True`, `False` and `None` are compared by identity.
- Patterns may use named constants. These must be dotted names to prevent them from being interpreted as capture variable:

```
from enum import Enum
class Color(Enum):
    RED = 'red'
    GREEN = 'green'
    BLUE = 'blue'

color = Color(input("Enter your choice of 'red', 'blue' or
'green': "))

match color:
    case Color.RED:
        print("I see red!")
    case Color.GREEN:
        print("Grass is green")
    case Color.BLUE:
        print("I'm feeling the blues :(")
```

For a more detailed explanation and additional examples, you can look into **PEP 636** [https://peps.python.org/pep-0636/] which is written in a tutorial format.

4.8. Defining Functions

We can create a function that writes the Fibonacci series to an arbitrary boundary:

```
>>> def fib(n):    # write Fibonacci series less than n
...     """Print a Fibonacci series less than n."""
...     a, b = 0, 1
...     while a < n:
...         print(a, end=' ')
...         a, b = b, a+b
...     print()
...
>>> # Now call the function we just defined:
>>> fib(2000)
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987 1597
```

The keyword **`def`** introduces a function *definition*. It must be followed by the function name and the parenthesized list of formal parameters. The statements that form the body of the function start at the next line, and must be indented.

The first statement of the function body can optionally be a string literal; this string literal is the function’s documentation string, or *docstring*. (More about docstrings can be found in the section Documentation Strings.) There are tools which use docstrings to automatically produce online or printed documentation, or to let the user interactively browse through code; it’s good practice to include docstrings in code that you write, so make a habit of it.

The *execution* of a function introduces a new symbol table used for the local variables of the function. More precisely, all variable assignments in a function store the value in the local symbol table; whereas variable references first look in the local symbol table, then in the local symbol tables of enclosing functions, then in the global symbol table, and finally in the table of built-in names. Thus, global variables and variables of enclosing functions cannot be directly assigned a value within a function (unless, for global variables, named in a **`global`** statement, or, for variables of enclosing functions, named in a **`nonlocal`** statement), although they may be referenced.

The actual parameters (arguments) to a function call are introduced in the local symbol table of the called function when it is called; thus, arguments are passed using *call by value* (where the *value* is always an object *reference*, not the value of the object). [[1]] When a function calls another function, or calls itself recursively, a new local symbol table is created for that call.

A function definition associates the function name with the function object in the current symbol table. The interpreter recognizes the object pointed to by that name as a user-defined function. Other names can also point to that same function object and can also be used to access the function:

```
>>> fib
<function fib at 10042ed0>
>>> f = fib
>>> f(100)
0 1 1 2 3 5 8 13 21 34 55 89
```

Coming from other languages, you might object that `fib` is not a function but a procedure since it doesn’t return a value. In fact, even functions without a **`return`** statement do return a value, albeit a rather boring one. This value is called `None` (it’s a built-in name). Writing the value `None` is normally suppressed by the interpreter if it would be the only value written. You can see it if you really want to using **`print()`**:

```
>>> fib(0)
>>> print(fib(0))
None
```

It is simple to write a function that returns a list of the numbers of the Fibonacci series, instead of printing it:

```
>>> def fib2(n):  # return Fibonacci series up to n
...     """Return a list containing the Fibonacci series up to
```

```
n."""
...     result = []
...     a, b = 0, 1
...     while a < n:
...         result.append(a)    # see below
...         a, b = b, a+b
...     return result
...
>>> f100 = fib2(100)    # call it
>>> f100                # write the result
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89]
```

This example, as usual, demonstrates some new Python features:

- The **`return`** statement returns with a value from a function. **`return`** without an expression argument returns `None`. Falling off the end of a function also returns `None`.
- The statement `result.append(a)` calls a *method* of the list object `result`. A method is a function that ‘belongs’ to an object and is named `obj.methodname`, where `obj` is some object (this may be an expression), and `methodname` is the name of a method that is defined by the object’s type. Different types define different methods. Methods of different types may have the same name without causing ambiguity. (It is possible to define your own object types and methods, using *classes*, see Classes) The method **`append()`** shown in the example is defined for list objects; it adds a new element at the end of the list. In this example it is equivalent to `result = result + [a]`, but more efficient.

4.9. More on Defining Functions

It is also possible to define functions with a variable number of arguments. There are three forms, which can be combined.

4.9.1. Default Argument Values

The most useful form is to specify a default value for one or more arguments. This creates a function that can be called with fewer arguments than it is defined to allow. For example:

```
def ask_ok(prompt, retries=4, reminder='Please try again!'):
    while True:
        reply = input(prompt)
        if reply in {'y', 'ye', 'yes'}:
            return True
        if reply in {'n', 'no', 'nop', 'nope'}:
            return False
        retries = retries - 1
        if retries < 0:
            raise ValueError('invalid user response')
        print(reminder)
```

This function can be called in several ways:

- giving only the mandatory argument: `ask_ok('Do you really want to quit?')`
- giving one of the optional arguments: `ask_ok('OK to overwrite the file?', 2)`
- or even giving all arguments: `ask_ok('OK to overwrite the file?', 2, 'Come on, only yes or no!')`

This example also introduces the **`in`** keyword. This tests whether or not a sequence contains a certain value.

The default values are evaluated at the point of function definition in the *defining* scope, so that

```
i = 5

def f(arg=i):
    print(arg)
```

```
i = 6
f()
```

will print `5`.

Important warning: The default value is evaluated only once. This makes a difference when the default is a mutable object such as a list, dictionary, or instances of most classes. For example, the following function accumulates the arguments passed to it on subsequent calls:

```
def f(a, L=[]):
    L.append(a)
    return L

print(f(1))
print(f(2))
print(f(3))
```

This will print

```
[1]
[1, 2]
[1, 2, 3]
```

If you don’t want the default to be shared between subsequent calls, you can write the function like this instead:

```
def f(a, L=None):
    if L is None:
        L = []
    L.append(a)
    return L
```

4.9.2. Keyword Arguments

Functions can also be called using keyword arguments of the form `kwarg=value`. For instance, the following function:

```
def parrot(voltage, state='a stiff', action='voom',
type='Norwegian Blue'):
    print("-- This parrot wouldn't", action, end=' ')
    print("if you put", voltage, "volts through it.")
    print("-- Lovely plumage, the", type)
    print("-- It's", state, "!")
```

accepts one required argument (`voltage`) and three optional arguments (`state`, `action`, and `type`). This function can be called in any of the following ways:

```
parrot(1000)                                          # 1
positional argument
parrot(voltage=1000)                                  # 1
keyword argument
parrot(voltage=1000000, action='VOOOOOM')             # 2
keyword arguments
parrot(action='VOOOOOM', voltage=1000000)             # 2
keyword arguments
parrot('a million', 'bereft of life', 'jump')         # 3
positional arguments
parrot('a thousand', state='pushing up the daisies')  # 1
positional, 1 keyword
```

but all the following calls would be invalid:

```
parrot()                     # required argument missing
parrot(voltage=5.0, 'dead')  # non-keyword argument after a
keyword argument
parrot(110, voltage=220)     # duplicate value for the same
argument
parrot(actor='John Cleese')  # unknown keyword argument
```

In a function call, keyword arguments must follow positional arguments. All the keyword arguments passed must match one of the arguments accepted by the function (e.g. `actor` is not a valid argument for the `parrot` function), and their order is not important. This also includes non-optional arguments (e.g. `parrot(voltage=1000)` is valid too). No argument may receive a value more than once. Here’s an example that fails due to this restriction:

```
>>> def function(a):
...     pass
...
>>> function(0, a=0)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: function() got multiple values for argument 'a'
```

When a final formal parameter of the form `**name` is present, it receives a dictionary (see Mapping Types — dict) containing all keyword arguments except for those corresponding to a formal parameter. This may be combined with a formal parameter of the form `*name` (described in the next subsection) which receives a tuple containing the positional arguments beyond the formal parameter list. (`*name` must occur before `**name`.) For example, if we define a function like this:

```
def cheeseshop(kind, *arguments, **keywords):
    print("-- Do you have any", kind, "?")
    print("-- I'm sorry, we're all out of", kind)
    for arg in arguments:
        print(arg)
    print("-" * 40)
    for kw in keywords:
        print(kw, ":", keywords[kw])
```

It could be called like this:

```
cheeseshop("Limburger", "It's very runny, sir.",
           "It's really very, VERY runny, sir.",
           shopkeeper="Michael Palin",
           client="John Cleese",
           sketch="Cheese Shop Sketch")
```

and of course it would print:

```
-- Do you have any Limburger ?
-- I'm sorry, we're all out of Limburger
It's very runny, sir.
It's really very, VERY runny, sir.
----------------------------------------
shopkeeper : Michael Palin
client : John Cleese
sketch : Cheese Shop Sketch
```

Note that the order in which the keyword arguments are printed is guaranteed to match the order in which they were provided in the function call.

4.9.3. Special parameters

By default, arguments may be passed to a Python function either by position or explicitly by keyword. For readability and performance, it makes sense to restrict the way arguments can be passed so that a developer need only look at the function definition to determine if items are passed by position, by position or keyword, or by keyword.

A function definition may look like:

```
def f(pos1, pos2, /, pos_or_kwd, *, kwd1, kwd2):
      -----------    ----------     ----------
        |             |                  |
        |        Positional or keyword   |
        |                                - Keyword only
         -- Positional only
```

where `/` and `*` are optional. If used, these symbols indicate the kind of parameter by how the arguments may be passed to the function: positional-only, positional-or-keyword, and keyword-only. Keyword parameters are also referred to as named parameters.

4.9.3.1. Positional-or-Keyword Arguments

If `/` and `*` are not present in the function definition, arguments may be passed to a function by position or by keyword.

4.9.3.2. Positional-Only Parameters

Looking at this in a bit more detail, it is possible to mark certain parameters as *positional-only*. If *positional-only*, the parameters' order matters, and the parameters cannot be passed by keyword. Positional-only parameters are placed before a `/` (forward-slash). The `/` is used to logically separate the positional-only parameters from the rest of the parameters. If there is no `/` in the function definition, there are no positional-only parameters.

Parameters following the `/` may be *positional-or-keyword* or *keyword-only*.

4.9.3.3. Keyword-Only Arguments

To mark parameters as *keyword-only*, indicating the parameters must be passed by keyword argument, place an `*` in the arguments list just before the first *keyword-only* parameter.

4.9.3.4. Function Examples

Consider the following example function definitions paying close attention to the markers `/` and `*`:

```
>>> def standard_arg(arg):
...     print(arg)
...
>>> def pos_only_arg(arg, /):
...     print(arg)
...
```

```
>>> def kwd_only_arg(*, arg):
...     print(arg)
...
>>> def combined_example(pos_only, /, standard, *, kwd_only):
...     print(pos_only, standard, kwd_only)
```

The first function definition, `standard_arg`, the most familiar form, places no restrictions on the calling convention and arguments may be passed by position or keyword:

```
>>> standard_arg(2)
2

>>> standard_arg(arg=2)
2
```

The second function `pos_only_arg` is restricted to only use positional parameters as there is a `/` in the function definition:

```
>>> pos_only_arg(1)
1

>>> pos_only_arg(arg=1)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: pos_only_arg() got some positional-only arguments
passed as keyword arguments: 'arg'
```

The third function `kwd_only_arg` only allows keyword arguments as indicated by a `*` in the function definition:

```
>>> kwd_only_arg(3)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: kwd_only_arg() takes 0 positional arguments but 1
was given

>>> kwd_only_arg(arg=3)
3
```

And the last uses all three calling conventions in the same function definition:

```
>>> combined_example(1, 2, 3)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: combined_example() takes 2 positional arguments but
3 were given

>>> combined_example(1, 2, kwd_only=3)
1 2 3

>>> combined_example(1, standard=2, kwd_only=3)
1 2 3

>>> combined_example(pos_only=1, standard=2, kwd_only=3)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: combined_example() got some positional-only
arguments passed as keyword arguments: 'pos_only'
```

Finally, consider this function definition which has a potential collision between the positional argument `name` and `**kwds` which has `name` as a key:

```
def foo(name, **kwds):
    return 'name' in kwds
```

There is no possible call that will make it return `True` as the keyword `'name'` will always bind to the first parameter. For example:

```
>>> foo(1, **{'name': 2})
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: foo() got multiple values for argument 'name'
>>>
```

But using `/` (positional only arguments), it is possible since it allows `name` as a positional argument and `'name'` as a key in the keyword arguments:

```
>>> def foo(name, /, **kwds):
...     return 'name' in kwds
...
>>> foo(1, **{'name': 2})
True
```

In other words, the names of positional-only parameters can be used in `**kwds` without ambiguity.

4.9.3.5. Recap

The use case will determine which parameters to use in the function definition:

```
def f(pos1, pos2, /, pos_or_kwd, *, kwd1, kwd2):
```

As guidance:

- Use positional-only if you want the name of the parameters to not be available to the user. This is useful when parameter names have no real meaning, if you want to enforce the order of the arguments when the function is called or if you need to take some positional parameters and arbitrary keywords.
- Use keyword-only when names have meaning and the function definition is more understandable by being explicit with names or you want to prevent users relying on the position of the argument being passed.
- For an API, use positional-only to prevent breaking API changes if the parameter's name is modified in the future.

4.9.4. Arbitrary Argument Lists

Finally, the least frequently used option is to specify that a function can be called with an arbitrary number of arguments. These arguments will be wrapped up in a tuple (see Tuples and Sequences). Before the variable number of arguments, zero or more normal arguments may occur.

```
def write_multiple_items(file, separator, *args):
    file.write(separator.join(args))
```

Normally, these *variadic* arguments will be last in the list of formal parameters, because they scoop up all remaining input arguments that are passed to the function. Any formal parameters which occur after the `*args` parameter are ‘keyword-only’ arguments, meaning that they can only be used as keywords rather than positional arguments.

```
>>> def concat(*args, sep="/"):
...     return sep.join(args)
...
>>> concat("earth", "mars", "venus")
'earth/mars/venus'
>>> concat("earth", "mars", "venus", sep=".")
'earth.mars.venus'
```

4.9.5. Unpacking Argument Lists

The reverse situation occurs when the arguments are already in a list or tuple but need to be unpacked for a function call requiring separate positional arguments. For instance, the built-in **`range()`** function expects separate *start* and *stop* arguments. If they are not available separately, write the function call with the `*`-operator to unpack the arguments out of a list or tuple:

```
>>> list(range(3, 6))            # normal call with separate
arguments
[3, 4, 5]
>>> args = [3, 6]
>>> list(range(*args))            # call with arguments
unpacked from a list
[3, 4, 5]
```

In the same fashion, dictionaries can deliver keyword arguments with the `**`-operator:

```
>>> def parrot(voltage, state='a stiff', action='voom'):
...     print("-- This parrot wouldn't", action, end=' ')
...     print("if you put", voltage, "volts through it.", end='
')
...     print("E's", state, "!")
...
>>> d = {"voltage": "four million", "state": "bleedin'
demised", "action": "VOOM"}
>>> parrot(**d)
-- This parrot wouldn't VOOM if you put four million volts
through it. E's bleedin' demised !
```

4.9.6. Lambda Expressions

Small anonymous functions can be created with the **`lambda`** keyword. This function returns the sum of its two arguments: `lambda a, b: a+b`. Lambda functions can be used wherever function objects are required. They are syntactically restricted to a single expression. Semantically, they are just syntactic sugar for a normal function definition. Like nested function definitions, lambda functions can reference variables from the containing scope:

```
>>> def make_incrementor(n):
...     return lambda x: x + n
...
>>> f = make_incrementor(42)
>>> f(0)
42
>>> f(1)
43
```

The above example uses a lambda expression to return a function. Another use is to pass a small function as an argument. For instance, **`list.sort()`** takes a sorting key function *key* which can be a lambda function:

```
>>> pairs = [(1, 'one'), (2, 'two'), (3, 'three'), (4, 'four')]
>>> pairs.sort(key=lambda pair: pair[1])
>>> pairs
[(4, 'four'), (1, 'one'), (3, 'three'), (2, 'two')]
```

4.9.7. Documentation Strings

Here are some conventions about the content and formatting of documentation strings.

The first line should always be a short, concise summary of the object’s purpose. For brevity, it should not explicitly state the object’s name or type, since these are available by other means (except if the name happens to be a verb describing a function’s operation). This line should begin with a capital letter and end with a period.

If there are more lines in the documentation string, the second line should be blank, visually separating the summary from the rest of the description. The following lines should be one or more paragraphs describing the object’s calling conventions, its side effects, etc.

The Python parser strips indentation from multi-line string literals when they serve as module, class, or function docstrings.

Here is an example of a multi-line docstring:

```
>>> def my_function():
...     """Do nothing, but document it.
...
...     No, really, it doesn't do anything:
...
...         >>> my_function()
...         >>>
...     """
...     pass
...
```

```
>>> print(my_function.__doc__)
Do nothing, but document it.

No, really, it doesn't do anything:

    >>> my_function()
    >>>
```

4.9.8. Function Annotations

Function annotations are completely optional metadata information about the types used by user-defined functions (see **PEP 3107** [https://peps.python.org/pep-3107/] and **PEP 484** [https://peps.python.org/pep-0484/] for more information).

Annotations are stored in the **`__annotations__`** attribute of the function as a dictionary and have no effect on any other part of the function. Parameter annotations are defined by a colon after the parameter name, followed by an expression evaluating to the value of the annotation. Return annotations are defined by a literal `->`, followed by an expression, between the parameter list and the colon denoting the end of the **`def`** statement. The following example has a required argument, an optional argument, and the return value annotated:

```
>>> def f(ham: str, eggs: str = 'eggs') -> str:
...     print("Annotations:", f.__annotations__)
...     print("Arguments:", ham, eggs)
...     return ham + ' and ' + eggs
...
>>> f('spam')
Annotations: {'ham': <class 'str'>, 'return': <class 'str'>,
'eggs': <class 'str'>}
Arguments: spam eggs
'spam and eggs'
```

4.10. Intermezzo: Coding Style

Now that you are about to write longer, more complex pieces of Python, it is a good time to talk about *coding style*. Most languages can be written (or more concise, *formatted*) in different styles; some are more readable than others. Making it easy for others to read your code is always a good idea, and adopting a nice coding style helps tremendously for that.

For Python, **PEP 8** [https://peps.python.org/pep-0008/] has emerged as the style guide that most projects adhere to; it promotes a very readable and eye-pleasing coding style. Every Python developer should read it at some point; here are the most important points extracted for you:

- Use 4-space indentation, and no tabs.

  4 spaces are a good compromise between small indentation (allows greater nesting depth) and large indentation (easier to read). Tabs introduce confusion, and are best left out.

- Wrap lines so that they don't exceed 79 characters.

  This helps users with small displays and makes it possible to have several code files side-by-side on larger displays.

- Use blank lines to separate functions and classes, and larger blocks of code inside functions.

- When possible, put comments on a line of their own.

- Use docstrings.

- Use spaces around operators and after commas, but not directly inside bracketing constructs: `a = f(1, 2) + g(3, 4)`.

- Name your classes and functions consistently; the convention is to use `UpperCamelCase` for classes and `lowercase_with_underscores` for functions and methods. Always use `self` as the name for the first method argument (see A First Look at Classes for more on classes and methods).

- Don’t use fancy encodings if your code is meant to be used in international environments. Python’s default, UTF-8, or even plain ASCII work best in any case.

- Likewise, don’t use non-ASCII characters in identifiers if there is only the slightest chance people speaking a different language will read or maintain the code.

Footnotes

[1]

Actually, *call by object reference* would be a better description, since if a mutable object is passed, the caller will see any changes the callee makes to it (items inserted into a list).

5. Data Structures

This chapter describes some things you’ve learned about already in more detail, and adds some new things as well.

5.1. More on Lists

The list data type has some more methods. Here are all of the methods of list objects:

list.append(*x*)

Add an item to the end of the list. Similar to `a[len(a):] = [x]`.

list.extend(*iterable*)

Extend the list by appending all the items from the iterable. Similar to `a[len(a):] = iterable`.

list.insert(*i*, *x*)

Insert an item at a given position. The first argument is the index of the element before which to insert, so `a.insert(0, x)` inserts at the front of the list, and `a.insert(len(a), x)` is equivalent to `a.append(x)`.

list.remove(*x*)

Remove the first item from the list whose value is equal to *x*. It raises a **`ValueError`** if there is no such item.

list.pop([*i*])

Remove the item at the given position in the list, and return it. If no index is specified, `a.pop()` removes and returns the last item in the list. It raises an `IndexError` if the list is empty or the index is outside the list range.

list.clear()

Remove all items from the list. Similar to `del a[:]`.

list.index(*x*[, *start*[, *end*]])

Return zero-based index of the first occurrence of *x* in the list. Raises a `ValueError` if there is no such item.

The optional arguments *start* and *end* are interpreted as in the slice notation and are used to limit the search to a particular subsequence of the list. The returned index is computed relative to the beginning of the full sequence rather than the *start* argument.

list.count(*x*)

Return the number of times *x* appears in the list.

list.sort(*, *key=None*, *reverse=False*)

Sort the items of the list in place (the arguments can be used for sort customization, see `sorted()` for their explanation).

list.reverse()

Reverse the elements of the list in place.

list.copy()

Return a shallow copy of the list. Similar to `a[:]`.

An example that uses most of the list methods:

```
>>> fruits = ['orange', 'apple', 'pear', 'banana', 'kiwi',
'apple', 'banana']
>>> fruits.count('apple')
2
```

```
>>> fruits.count('tangerine')
0
>>> fruits.index('banana')
3
>>> fruits.index('banana', 4)  # Find next banana starting at
position 4
6
>>> fruits.reverse()
>>> fruits
['banana', 'apple', 'kiwi', 'banana', 'pear', 'apple',
'orange']
>>> fruits.append('grape')
>>> fruits
['banana', 'apple', 'kiwi', 'banana', 'pear', 'apple',
'orange', 'grape']
>>> fruits.sort()
>>> fruits
['apple', 'apple', 'banana', 'banana', 'grape', 'kiwi',
'orange', 'pear']
>>> fruits.pop()
'pear'
```

You might have noticed that methods like `insert`, `remove` or `sort` that only modify the list have no return value printed – they return the default `None`. [[1]] This is a design principle for all mutable data structures in Python.

Another thing you might notice is that not all data can be sorted or compared. For instance, `[None, 'hello', 10]` doesn’t sort because integers can’t be compared to strings and `None` can’t be compared to other types. Also, there are some types that don’t have a defined ordering relation. For example, `3+4j < 5+7j` isn’t a valid comparison.

5.1.1. Using Lists as Stacks

The list methods make it very easy to use a list as a stack, where the last element added is the first element retrieved (“last-in, first-out”). To add an item to the top of the stack, use `append()`. To retrieve an item from the top of the stack, use `pop()` without an explicit index. For example:

```
>>> stack = [3, 4, 5]
>>> stack.append(6)
>>> stack.append(7)
>>> stack
[3, 4, 5, 6, 7]
>>> stack.pop()
7
>>> stack
[3, 4, 5, 6]
>>> stack.pop()
6
>>> stack.pop()
5
>>> stack
[3, 4]
```

5.1.2. Using Lists as Queues

It is also possible to use a list as a queue, where the first element added is the first element retrieved ("first-in, first-out"); however, lists are not efficient for this purpose. While appends and pops from the end of list are fast, doing inserts or pops from the beginning of a list is slow (because all of the other elements have to be shifted by one).

To implement a queue, use `collections.deque` which was designed to have fast appends and pops from both ends. For example:

```
>>> from collections import deque
>>> queue = deque(["Eric", "John", "Michael"])
>>> queue.append("Terry")           # Terry arrives
>>> queue.append("Graham")          # Graham arrives
>>> queue.popleft()                 # The first to arrive now
leaves
'Eric'
>>> queue.popleft()                 # The second to arrive now
```

```
leaves
'John'
>>> queue                           # Remaining queue in order
of arrival
deque(['Michael', 'Terry', 'Graham'])
```

5.1.3. List Comprehensions

List comprehensions provide a concise way to create lists. Common applications are to make new lists where each element is the result of some operations applied to each member of another sequence or iterable, or to create a subsequence of those elements that satisfy a certain condition.

For example, assume we want to create a list of squares, like:

```
>>> squares = []
>>> for x in range(10):
...     squares.append(x**2)
...
>>> squares
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

Note that this creates (or overwrites) a variable named `x` that still exists after the loop completes. We can calculate the list of squares without any side effects using:

```
squares = list(map(lambda x: x**2, range(10)))
```

or, equivalently:

```
squares = [x**2 for x in range(10)]
```

which is more concise and readable.

A list comprehension consists of brackets containing an expression followed by a **`for`** clause, then zero or more **`for`** or

`if` clauses. The result will be a new list resulting from evaluating the expression in the context of the **`for`** and **`if`** clauses which follow it. For example, this listcomp combines the elements of two lists if they are not equal:

```
>>> [(x, y) for x in [1,2,3] for y in [3,1,4] if x != y]
[(1, 3), (1, 4), (2, 3), (2, 1), (2, 4), (3, 1), (3, 4)]
```

and it’s equivalent to:

```
>>> combs = []
>>> for x in [1,2,3]:
...     for y in [3,1,4]:
...         if x != y:
...             combs.append((x, y))
...
>>> combs
[(1, 3), (1, 4), (2, 3), (2, 1), (2, 4), (3, 1), (3, 4)]
```

Note how the order of the **`for`** and **`if`** statements is the same in both these snippets.

If the expression is a tuple (e.g. the `(x, y)` in the previous example), it must be parenthesized.

```
>>> vec = [-4, -2, 0, 2, 4]
>>> # create a new list with the values doubled
>>> [x*2 for x in vec]
[-8, -4, 0, 4, 8]
>>> # filter the list to exclude negative numbers
>>> [x for x in vec if x >= 0]
[0, 2, 4]
>>> # apply a function to all the elements
>>> [abs(x) for x in vec]
[4, 2, 0, 2, 4]
>>> # call a method on each element
>>> freshfruit = ['  banana', '  loganberry ', 'passion fruit
']
>>> [weapon.strip() for weapon in freshfruit]
['banana', 'loganberry', 'passion fruit']
>>> # create a list of 2-tuples like (number, square)
>>> [(x, x**2) for x in range(6)]
[(0, 0), (1, 1), (2, 4), (3, 9), (4, 16), (5, 25)]
>>> # the tuple must be parenthesized, otherwise an error is
raised
>>> [x, x**2 for x in range(6)]
  File "<stdin>", line 1
    [x, x**2 for x in range(6)]
     ^^^^^^^
SyntaxError: did you forget parentheses around the
comprehension target?
>>> # flatten a list using a listcomp with two 'for'
>>> vec = [[1,2,3], [4,5,6], [7,8,9]]
>>> [num for elem in vec for num in elem]
[1, 2, 3, 4, 5, 6, 7, 8, 9]
```

List comprehensions can contain complex expressions and nested functions:

```
>>> from math import pi
>>> [str(round(pi, i)) for i in range(1, 6)]
['3.1', '3.14', '3.142', '3.1416', '3.14159']
```

5.1.4. Nested List Comprehensions

The initial expression in a list comprehension can be any arbitrary expression, including another list comprehension.

Consider the following example of a 3x4 matrix implemented as a list of 3 lists of length 4:

```
>>> matrix = [
...     [1, 2, 3, 4],
...     [5, 6, 7, 8],
...     [9, 10, 11, 12],
... ]
```

The following list comprehension will transpose rows and columns:

```
>>> [[row[i] for row in matrix] for i in range(4)]
[[1, 5, 9], [2, 6, 10], [3, 7, 11], [4, 8, 12]]
```

As we saw in the previous section, the inner list comprehension is evaluated in the context of the `for` that follows it, so this example is equivalent to:

```
>>> transposed = []
>>> for i in range(4):
...     transposed.append([row[i] for row in matrix])
...
>>> transposed
[[1, 5, 9], [2, 6, 10], [3, 7, 11], [4, 8, 12]]
```

which, in turn, is the same as:

```
>>> transposed = []
>>> for i in range(4):
...     # the following 3 lines implement the nested listcomp
...     transposed_row = []
...     for row in matrix:
...         transposed_row.append(row[i])
...     transposed.append(transposed_row)
...
>>> transposed
[[1, 5, 9], [2, 6, 10], [3, 7, 11], [4, 8, 12]]
```

In the real world, you should prefer built-in functions to complex flow statements. The `zip()` function would do a great job for this use case:

```
>>> list(zip(*matrix))
[(1, 5, 9), (2, 6, 10), (3, 7, 11), (4, 8, 12)]
```

See Unpacking Argument Lists for details on the asterisk in this line.

5.2. The `del` statement

There is a way to remove an item from a list given its index instead of its value: the `del` statement. This differs from the `pop()` method which returns a value. The `del` statement can also be used to remove slices from a list or clear the entire list (which we did earlier by assignment of an empty list to the slice). For example:

```
>>> a = [-1, 1, 66.25, 333, 333, 1234.5]
>>> del a[0]
>>> a
[1, 66.25, 333, 333, 1234.5]
>>> del a[2:4]
>>> a
[1, 66.25, 1234.5]
>>> del a[:]
>>> a
[]
```

`del` can also be used to delete entire variables:

```
>>> del a
```

Referencing the name `a` hereafter is an error (at least until another value is assigned to it). We’ll find other uses for **`del`** later.

5.3. Tuples and Sequences

We saw that lists and strings have many common properties, such as indexing and slicing operations. They are two examples of *sequence* data types (see Sequence Types — list, tuple, range). Since Python is an evolving language, other sequence data types may be added. There is also another standard sequence data type: the *tuple*.

A tuple consists of a number of values separated by commas, for instance:

```
>>> t = 12345, 54321, 'hello!'
>>> t[0]
12345
>>> t
```

```
(12345, 54321, 'hello!')
>>> # Tuples may be nested:
>>> u = t, (1, 2, 3, 4, 5)
>>> u
((12345, 54321, 'hello!'), (1, 2, 3, 4, 5))
>>> # Tuples are immutable:
>>> t[0] = 88888
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
>>> # but they can contain mutable objects:
>>> v = ([1, 2, 3], [3, 2, 1])
>>> v
([1, 2, 3], [3, 2, 1])
```

As you see, on output tuples are always enclosed in parentheses, so that nested tuples are interpreted correctly; they may be input with or without surrounding parentheses, although often parentheses are necessary anyway (if the tuple is part of a larger expression). It is not possible to assign to the individual items of a tuple, however it is possible to create tuples which contain mutable objects, such as lists.

Though tuples may seem similar to lists, they are often used in different situations and for different purposes. Tuples are immutable, and usually contain a heterogeneous sequence of elements that are accessed via unpacking (see later in this section) or indexing (or even by attribute in the case of **`namedtuples`**). Lists are mutable, and their elements are usually homogeneous and are accessed by iterating over the list.

A special problem is the construction of tuples containing 0 or 1 items: the syntax has some extra quirks to accommodate these. Empty tuples are constructed by an empty pair of parentheses; a tuple with one item is constructed by following a value with a comma (it is not sufficient to enclose a single value in parentheses). Ugly, but effective. For example:

```
>>> empty = ()
>>> singleton = 'hello',    # <-- note trailing comma
>>> len(empty)
0
>>> len(singleton)
1
>>> singleton
('hello',)
```

The statement `t = 12345, 54321, 'hello!'` is an example of *tuple packing*: the values `12345`, `54321` and `'hello!'` are packed together in a tuple. The reverse operation is also possible:

```
>>> x, y, z = t
```

This is called, appropriately enough, *sequence unpacking* and works for any sequence on the right-hand side. Sequence unpacking requires that there are as many variables on the left side of the equals sign as there are elements in the sequence. Note that multiple assignment is really just a combination of tuple packing and sequence unpacking.

5.4. Sets

Python also includes a data type for sets. A set is an unordered collection with no duplicate elements. Basic uses include membership testing and eliminating duplicate entries. Set objects also support mathematical operations like union, intersection, difference, and symmetric difference.

Curly braces or the **`set()`** function can be used to create sets. Note: to create an empty set you have to use `set()`, not `{}`; the latter creates an empty dictionary, a data structure that we discuss in the next section.

Here is a brief demonstration:

```
>>> basket = {'apple', 'orange', 'apple', 'pear', 'orange', 
'banana'}
>>> print(basket)                      # show that duplicates 
have been removed
{'orange', 'banana', 'pear', 'apple'}
>>> 'orange' in basket                 # fast membership 
testing
True
>>> 'crabgrass' in basket
False

>>> # Demonstrate set operations on unique letters from two 
words
>>>
>>> a = set('abracadabra')
>>> b = set('alacazam')
>>> a                                  # unique letters in a
{'a', 'r', 'b', 'c', 'd'}
>>> a - b                              # letters in a but not 
in b
{'r', 'd', 'b'}
>>> a | b                              # letters in a or b or 
both
{'a', 'c', 'r', 'd', 'b', 'm', 'z', 'l'}
>>> a & b                              # letters in both a and 
b
{'a', 'c'}
>>> a ^ b                              # letters in a or b but 
not both
{'r', 'd', 'b', 'm', 'z', 'l'}
```

Similarly to list comprehensions, set comprehensions are also supported:

```
>>> a = {x for x in 'abracadabra' if x not in 'abc'}
>>> a
{'r', 'd'}
```

5.5. Dictionaries

Another useful data type built into Python is the *dictionary* (see Mapping Types — dict). Dictionaries are sometimes found in other languages as “associative memories” or “associative arrays”. Unlike sequences, which are indexed by a range of numbers, dictionaries are indexed by *keys*, which can be any immutable type; strings and numbers can always be keys. Tuples can be used as keys if they contain only strings, numbers, or tuples; if a tuple contains any mutable object either directly or indirectly, it cannot be used as a key. You can’t use lists as keys, since lists can be modified in place using index assignments, slice assignments, or methods like **`append()`** and **`extend()`**.

It is best to think of a dictionary as a set of *key: value* pairs, with the requirement that the keys are unique (within one dictionary). A pair of braces creates an empty dictionary: `{}`. Placing a comma-separated list of key:value pairs within the braces adds initial key:value pairs to the dictionary; this is also the way dictionaries are written on output.

The main operations on a dictionary are storing a value with some key and extracting the value given the key. It is also possible to delete a key:value pair with `del`. If you store using a key that is already in use, the old value associated with that key is forgotten.

Extracting a value for a non-existent key by subscripting (`d[key]`) raises a **`KeyError`**. To avoid getting this error when trying to access a possibly non-existent key, use the **`get()`**

method instead, which returns `None` (or a specified default value) if the key is not in the dictionary.

Performing `list(d)` on a dictionary returns a list of all the keys used in the dictionary, in insertion order (if you want it sorted, just use `sorted(d)` instead). To check whether a single key is in the dictionary, use the **`in`** keyword.

Here is a small example using a dictionary:

```
>>> tel = {'jack': 4098, 'sape': 4139}
>>> tel['guido'] = 4127
>>> tel
{'jack': 4098, 'sape': 4139, 'guido': 4127}
>>> tel['jack']
4098
>>> tel['irv']
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 'irv'
>>> print(tel.get('irv'))
None
>>> del tel['sape']
>>> tel['irv'] = 4127
>>> tel
{'jack': 4098, 'guido': 4127, 'irv': 4127}
>>> list(tel)
['jack', 'guido', 'irv']
>>> sorted(tel)
['guido', 'irv', 'jack']
>>> 'guido' in tel
True
>>> 'jack' not in tel
False
```

The **`dict()`** constructor builds dictionaries directly from sequences of key-value pairs:

```
>>> dict([('sape', 4139), ('guido', 4127), ('jack', 4098)])
{'sape': 4139, 'guido': 4127, 'jack': 4098}
```

In addition, dict comprehensions can be used to create dictionaries from arbitrary key and value expressions:

```
>>> {x: x**2 for x in (2, 4, 6)}
{2: 4, 4: 16, 6: 36}
```

When the keys are simple strings, it is sometimes easier to specify pairs using keyword arguments:

```
>>> dict(sape=4139, guido=4127, jack=4098)
{'sape': 4139, 'guido': 4127, 'jack': 4098}
```

5.6. Looping Techniques

When looping through dictionaries, the key and corresponding value can be retrieved at the same time using the `items()` method.

```
>>> knights = {'gallahad': 'the pure', 'robin': 'the brave'}
>>> for k, v in knights.items():
...     print(k, v)
...
gallahad the pure
robin the brave
```

When looping through a sequence, the position index and corresponding value can be retrieved at the same time using the `enumerate()` function.

```
>>> for i, v in enumerate(['tic', 'tac', 'toe']):
...     print(i, v)
...
0 tic
1 tac
2 toe
```

To loop over two or more sequences at the same time, the entries can be paired with the `zip()` function.

```
>>> questions = ['name', 'quest', 'favorite color']
>>> answers = ['lancelot', 'the holy grail', 'blue']
>>> for q, a in zip(questions, answers):
...     print('What is your {0}?  It is {1}.'.format(q, a))
...
What is your name?  It is lancelot.
What is your quest?  It is the holy grail.
What is your favorite color?  It is blue.
```

To loop over a sequence in reverse, first specify the sequence in a forward direction and then call the **`reversed()`** function.

```
>>> for i in reversed(range(1, 10, 2)):
...     print(i)
...
9
7
5
3
1
```

To loop over a sequence in sorted order, use the **`sorted()`** function which returns a new sorted list while leaving the source unaltered.

```
>>> basket = ['apple', 'orange', 'apple', 'pear', 'orange',
'banana']
>>> for i in sorted(basket):
...     print(i)
...
apple
apple
banana
orange
orange
pear
```

Using **`set()`** on a sequence eliminates duplicate elements. The use of **`sorted()`** in combination with **`set()`** over a sequence is an idiomatic way to loop over unique elements of the sequence in sorted order.

```
>>> basket = ['apple', 'orange', 'apple', 'pear', 'orange',
'banana']
>>> for f in sorted(set(basket)):
...     print(f)
...
apple
banana
orange
pear
```

It is sometimes tempting to change a list while you are looping over it; however, it is often simpler and safer to create a new list instead.

```
>>> import math
>>> raw_data = [56.2, float('NaN'), 51.7, 55.3, 52.5,
float('NaN'), 47.8]
>>> filtered_data = []
>>> for value in raw_data:
...     if not math.isnan(value):
...         filtered_data.append(value)
...
>>> filtered_data
[56.2, 51.7, 55.3, 52.5, 47.8]
```

5.7. More on Conditions

The conditions used in `while` and `if` statements can contain any operators, not just comparisons.

The comparison operators `in` and `not in` are membership tests that determine whether a value is in (or not in) a container. The operators `is` and `is not` compare whether two objects are really the same object. All comparison operators have the same priority, which is lower than that of all numerical operators.

Comparisons can be chained. For example, `a < b == c` tests whether `a` is less than `b` and moreover `b` equals `c`.

Comparisons may be combined using the Boolean operators `and` and `or`, and the outcome of a comparison (or of any other Boolean expression) may be negated with `not`. These have lower priorities than comparison operators; between them, `not` has the highest priority and `or` the lowest, so that `A and not B or C` is equivalent to `(A and (not B)) or C`. As always, parentheses can be used to express the desired composition.

The Boolean operators `and` and `or` are so-called *short-circuit* operators: their arguments are evaluated from left to right, and evaluation stops as soon as the outcome is determined. For example, if `A` and `C` are true but `B` is false, `A and B and C` does not evaluate the expression `C`. When used as a general value and not as a Boolean, the return value of a short-circuit operator is the last evaluated argument.

It is possible to assign the result of a comparison or other Boolean expression to a variable. For example,

```
>>> string1, string2, string3 = '', 'Trondheim', 'Hammer Dance'
>>> non_null = string1 or string2 or string3
>>> non_null
'Trondheim'
```

Note that in Python, unlike C, assignment inside expressions must be done explicitly with the walrus operator `:=`. This avoids a common class of problems encountered in C programs: typing `=` in an expression when `==` was intended.

5.8. Comparing Sequences and Other Types

Sequence objects typically may be compared to other objects with the same sequence type. The comparison uses *lexicographical* ordering: first the first two items are compared, and if they differ this determines the outcome of the comparison; if they are equal, the next two items are compared, and so on, until either sequence is exhausted. If two items to be compared are themselves sequences of the same type, the lexicographical comparison is carried out recursively. If all items of two sequences compare equal, the sequences are considered equal. If one sequence is an initial sub-sequence of the other, the shorter sequence is the smaller (lesser) one. Lexicographical ordering for strings uses the Unicode code point number to order individual characters. Some examples of comparisons between sequences of the same type:

```
(1, 2, 3)              < (1, 2, 4)
[1, 2, 3]              < [1, 2, 4]
'ABC' < 'C' < 'Pascal' < 'Python'
(1, 2, 3, 4)           < (1, 2, 4)
(1, 2)                 < (1, 2, -1)
(1, 2, 3)             == (1.0, 2.0, 3.0)
(1, 2, ('aa', 'ab'))   < (1, 2, ('abc', 'a'), 4)
```

Note that comparing objects of different types with `<` or `>` is legal provided that the objects have appropriate comparison methods. For example, mixed numeric types are compared according to their numeric value, so 0 equals 0.0, etc. Otherwise, rather than providing an arbitrary ordering, the interpreter will raise a **`TypeError`** exception.

Footnotes

[1]

Other languages may return the mutated object, which allows method chaining, such as `d->insert("a")->remove("b")-`

```
>sort();.
```

6. Modules

If you quit from the Python interpreter and enter it again, the definitions you have made (functions and variables) are lost. Therefore, if you want to write a somewhat longer program, you are better off using a text editor to prepare the input for the interpreter and running it with that file as input instead. This is known as creating a *script*. As your program gets longer, you may want to split it into several files for easier maintenance. You may also want to use a handy function that you've written in several programs without copying its definition into each program.

To support this, Python has a way to put definitions in a file and use them in a script or in an interactive instance of the interpreter. Such a file is called a *module*; definitions from a module can be *imported* into other modules or into the *main* module (the collection of variables that you have access to in a script executed at the top level and in calculator mode).

A module is a file containing Python definitions and statements. The file name is the module name with the suffix `.py` appended. Within a module, the module's name (as a string) is available as the value of the global variable `__name__`. For instance, use your favorite text editor to create a file called `fibo.py` in the current directory with the following contents:

```
# Fibonacci numbers module

def fib(n):
    """Write Fibonacci series up to n."""
    a, b = 0, 1
    while a < n:
        print(a, end=' ')
        a, b = b, a+b
    print()

def fib2(n):
    """Return Fibonacci series up to n."""
    result = []
    a, b = 0, 1
    while a < n:
        result.append(a)
        a, b = b, a+b
    return result
```

Now enter the Python interpreter and import this module with the following command:

```
>>> import fibo
```

This does not add the names of the functions defined in `fibo` directly to the current namespace (see Python Scopes and Namespaces for more details); it only adds the module name `fibo` there. Using the module name you can access the functions:

```
>>> fibo.fib(1000)
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987
>>> fibo.fib2(100)
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89]
>>> fibo.__name__
'fibo'
```

If you intend to use a function often you can assign it to a local name:

```
>>> fib = fibo.fib
>>> fib(500)
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377
```

6.1. More on Modules

A module can contain executable statements as well as function definitions. These statements are intended to initialize the module. They are executed only the *first* time the module name is encountered in an import statement. [[1]] (They are also run if the file is executed as a script.)

Each module has its own private namespace, which is used as the global namespace by all functions defined in the module. Thus, the author of a module can use global variables in the module without worrying about accidental clashes with a user's global variables. On the other hand, if you know what you are doing you can touch a module's global variables with the same notation used to refer to its functions, `modname.itemname`.

Modules can import other modules. It is customary but not required to place all **`import`** statements at the beginning of a module (or script, for that matter). The imported module names, if placed at the top level of a module (outside any functions or classes), are added to the module's global namespace.

There is a variant of the **`import`** statement that imports names from a module directly into the importing module's namespace. For example:

```
>>> from fibo import fib, fib2
>>> fib(500)
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377
```

This does not introduce the module name from which the imports are taken in the local namespace (so in the example, `fibo` is not defined).

There is even a variant to import all names that a module defines:

```
>>> from fibo import *
>>> fib(500)
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377
```

This imports all names except those beginning with an underscore (_). In most cases Python programmers do not use this facility since it introduces an unknown set of names into the interpreter, possibly hiding some things you have already defined.

Note that in general the practice of importing `*` from a module or package is frowned upon, since it often causes poorly readable code. However, it is okay to use it to save typing in interactive sessions.

If the module name is followed by **`as`**, then the name following **`as`** is bound directly to the imported module.

```
>>> import fibo as fib
>>> fib.fib(500)
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377
```

This is effectively importing the module in the same way that `import fibo` will do, with the only difference of it being available as `fib`.

It can also be used when utilising **`from`** with similar effects:

```
>>> from fibo import fib as fibonacci
>>> fibonacci(500)
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377
```

Note

For efficiency reasons, each module is only imported once per interpreter session. Therefore, if you change your modules, you must restart the interpreter – or, if it’s just one module you want to test interactively, use

`importlib.reload()`, e.g. `import importlib; importlib.reload(modulename)`.

6.1.1. Executing modules as scripts

When you run a Python module with

```
python fibo.py <arguments>
```

the code in the module will be executed, just as if you imported it, but with the `__name__` set to `"__main__"`. That means that by adding this code at the end of your module:

```
if __name__ == "__main__":
    import sys
    fib(int(sys.argv[1]))
```

you can make the file usable as a script as well as an importable module, because the code that parses the command line only runs if the module is executed as the “main” file:

```
$ python fibo.py 50
0 1 1 2 3 5 8 13 21 34
```

If the module is imported, the code is not run:

```
>>> import fibo
>>>
```

This is often used either to provide a convenient user interface to a module, or for testing purposes (running the module as a script executes a test suite).

6.1.2. The Module Search Path

When a module named `spam` is imported, the interpreter first searches for a built-in module with that name. These module names are listed in `sys.builtin_module_names`. If not found, it then searches for a file named `spam.py` in a list of directories given by the variable `sys.path`. `sys.path` is initialized from these locations:

- The directory containing the input script (or the current directory when no file is specified).
- `PYTHONPATH` (a list of directory names, with the same syntax as the shell variable `PATH`).
- The installation-dependent default (by convention including a `site-packages` directory, handled by the `site` module).

More details are at The initialization of the sys.path module search path.

Note

On file systems which support symlinks, the directory containing the input script is calculated after the symlink is followed. In other words the directory containing the symlink is **not** added to the module search path.

After initialization, Python programs can modify `sys.path`. The directory containing the script being run is placed at the beginning of the search path, ahead of the standard library path. This means that scripts in that directory will be loaded instead of modules of the same name in the library directory. This is an error unless the replacement is intended. See section Standard Modules for more information.

6.1.3. “Compiled” Python files

To speed up loading modules, Python caches the compiled version of each module in the `__pycache__` directory under the name `module.`*`version`*`.pyc`, where the version encodes the format of the compiled file; it generally contains the Python version number. For example, in CPython release 3.3 the compiled version of spam.py would be cached as `__pycache__/spam.cpython-33.pyc`. This naming convention allows compiled modules from different releases and different versions of Python to coexist.

Python checks the modification date of the source against the compiled version to see if it's out of date and needs to be recompiled. This is a completely automatic process. Also, the compiled modules are platform-independent, so the same library can be shared among systems with different architectures.

Python does not check the cache in two circumstances. First, it always recompiles and does not store the result for the module that's loaded directly from the command line. Second, it does not check the cache if there is no source module. To support a non-source (compiled only) distribution, the compiled module must be in the source directory, and there must not be a source module.

Some tips for experts:

- You can use the **`-O`** or **`-OO`** switches on the Python command to reduce the size of a compiled module. The `-O` switch removes assert statements, the `-OO` switch removes both assert statements and __doc__ strings. Since some programs may rely on having these available, you should only use this option if you know what you're doing. “Optimized” modules have an `opt-` tag and are usually smaller. Future releases may change the effects of optimization.

- A program doesn’t run any faster when it is read from a `.pyc` file than when it is read from a `.py` file; the only thing that’s faster about `.pyc` files is the speed with which they are loaded.
- The module **`compileall`** can create .pyc files for all modules in a directory.
- There is more detail on this process, including a flow chart of the decisions, in **PEP 3147** [https://peps.python.org/pep-3147/].

6.2. Standard Modules

Python comes with a library of standard modules, described in a separate document, the Python Library Reference (“Library Reference” hereafter). Some modules are built into the interpreter; these provide access to operations that are not part of the core of the language but are nevertheless built in, either for efficiency or to provide access to operating system primitives such as system calls. The set of such modules is a configuration option which also depends on the underlying platform. For example, the **`winreg`** module is only provided on Windows systems. One particular module deserves some attention: **`sys`**, which is built into every Python interpreter. The variables `sys.ps1` and `sys.ps2` define the strings used as primary and secondary prompts:

```
>>> import sys
>>> sys.ps1
'>>> '
>>> sys.ps2
'... '
>>> sys.ps1 = 'C> '
C> print('Yuck!')
Yuck!
C>
```

These two variables are only defined if the interpreter is in interactive mode.

The variable `sys.path` is a list of strings that determines the interpreter’s search path for modules. It is initialized to a default path taken from the environment variable **`PYTHONPATH`**, or from a built-in default if **`PYTHONPATH`** is not set. You can modify it using standard list operations:

```
>>> import sys
>>> sys.path.append('/ufs/guido/lib/python')
```

6.3. The `dir()` Function

The built-in function `dir()` is used to find out which names a module defines. It returns a sorted list of strings:

```
>>> import fibo, sys
>>> dir(fibo)
['__name__', 'fib', 'fib2']
>>> dir(sys)
['__breakpointhook__', '__displayhook__', '__doc__',
'__excepthook__',
 '__interactivehook__', '__loader__', '__name__',
'__package__', '__spec__',
 '__stderr__', '__stdin__', '__stdout__', '__unraisablehook__',
 '_clear_type_cache', '_current_frames', '_debugmallocstats',
'_framework',
 '_getframe', '_git', '_home', '_xoptions', 'abiflags',
'addaudithook',
 'api_version', 'argv', 'audit', 'base_exec_prefix',
'base_prefix',
 'breakpointhook', 'builtin_module_names', 'byteorder',
'call_tracing',
 'callstats', 'copyright', 'displayhook',
'dont_write_bytecode', 'exc_info',
 'excepthook', 'exec_prefix', 'executable', 'exit', 'flags',
'float_info',
 'float_repr_style', 'get_asyncgen_hooks',
'get_coroutine_origin_tracking_depth',
 'getallocatedblocks', 'getdefaultencoding', 'getdlopenflags',
 'getfilesystemencodeerrors', 'getfilesystemencoding',
'getprofile',
 'getrecursionlimit', 'getrefcount', 'getsizeof',
'getswitchinterval',
 'gettrace', 'hash_info', 'hexversion', 'implementation',
'int_info',
 'intern', 'is_finalizing', 'last_traceback', 'last_type',
'last_value',
 'maxsize', 'maxunicode', 'meta_path', 'modules', 'path',
'path_hooks',
 'path_importer_cache', 'platform', 'prefix', 'ps1', 'ps2',
'pycache_prefix',
 'set_asyncgen_hooks', 'set_coroutine_origin_tracking_depth',
'setdlopenflags',
 'setprofile', 'setrecursionlimit', 'setswitchinterval',
'settrace', 'stderr',
 'stdin', 'stdout', 'thread_info', 'unraisablehook', 'version',
'version_info',
 'warnoptions']
```

Without arguments, `dir()` lists the names you have defined currently:

```
>>> a = [1, 2, 3, 4, 5]
>>> import fibo
>>> fib = fibo.fib
>>> dir()
['__builtins__', '__name__', 'a', 'fib', 'fibo', 'sys']
```

Note that it lists all types of names: variables, modules, functions, etc.

`dir()` does not list the names of built-in functions and variables. If you want a list of those, they are defined in the standard module `builtins`:

```
>>> import builtins
>>> dir(builtins)
['ArithmeticError', 'AssertionError', 'AttributeError',
'BaseException',
 'BlockingIOError', 'BrokenPipeError', 'BufferError',
'BytesWarning',
 'ChildProcessError', 'ConnectionAbortedError',
'ConnectionError',
 'ConnectionRefusedError', 'ConnectionResetError',
'DeprecationWarning',
 'EOFError', 'Ellipsis', 'EnvironmentError', 'Exception',
'False',
 'FileExistsError', 'FileNotFoundError', 'FloatingPointError',
 'FutureWarning', 'GeneratorExit', 'IOError', 'ImportError',
 'ImportWarning', 'IndentationError', 'IndexError',
'InterruptedError',
 'IsADirectoryError', 'KeyError', 'KeyboardInterrupt',
'LookupError',
 'MemoryError', 'NameError', 'None', 'NotADirectoryError',
'NotImplemented',
 'NotImplementedError', 'OSError', 'OverflowError',
 'PendingDeprecationWarning', 'PermissionError',
'ProcessLookupError',
 'ReferenceError', 'ResourceWarning', 'RuntimeError',
'RuntimeWarning',
 'StopIteration', 'SyntaxError', 'SyntaxWarning',
'SystemError',
 'SystemExit', 'TabError', 'TimeoutError', 'True', 'TypeError',
 'UnboundLocalError', 'UnicodeDecodeError',
'UnicodeEncodeError',
 'UnicodeError', 'UnicodeTranslateError', 'UnicodeWarning',
'UserWarning',
 'ValueError', 'Warning', 'ZeroDivisionError', '_',
'__build_class__',
 '__debug__', '__doc__', '__import__', '__name__',
'__package__', 'abs',
 'all', 'any', 'ascii', 'bin', 'bool', 'bytearray', 'bytes',
'callable',
 'chr', 'classmethod', 'compile', 'complex', 'copyright',
'credits',
 'delattr', 'dict', 'dir', 'divmod', 'enumerate', 'eval',
'exec', 'exit',
 'filter', 'float', 'format', 'frozenset', 'getattr',
'globals', 'hasattr',
 'hash', 'help', 'hex', 'id', 'input', 'int', 'isinstance',
'issubclass',
 'iter', 'len', 'license', 'list', 'locals', 'map', 'max',
'memoryview',
 'min', 'next', 'object', 'oct', 'open', 'ord', 'pow', 'print',
'property',
 'quit', 'range', 'repr', 'reversed', 'round', 'set',
'setattr', 'slice',
 'sorted', 'staticmethod', 'str', 'sum', 'super', 'tuple',
'type', 'vars',
 'zip']
```

6.4. Packages

Packages are a way of structuring Python’s module namespace by using “dotted module names”. For example, the module name **`A.B`** designates a submodule named `B` in a package named `A`. Just like the use of modules saves the authors of different modules from having to worry about each other’s global variable names, the use of dotted module names saves the authors of multi-module packages like NumPy or Pillow from having to worry about each other’s module names.

Suppose you want to design a collection of modules (a “package”) for the uniform handling of sound files and sound data. There are many different sound file formats (usually recognized by their extension, for example: `.wav`, `.aiff`, `.au`), so you may need to create and maintain a growing collection of modules for the conversion between the various file formats. There are also many different operations you might want to perform on sound data (such as mixing, adding echo, applying an equalizer function, creating an artificial stereo effect), so in addition you will be writing a never-ending stream of modules to perform these operations. Here’s a possible structure for your package (expressed in terms of a hierarchical filesystem):

```
sound/                          Top-level package
      __init__.py               Initialize the sound package
      formats/                  Subpackage for file format
conversions
              __init__.py
              wavread.py
              wavwrite.py
              aiffread.py
              aiffwrite.py
              auread.py
              auwrite.py
              ...
      effects/                  Subpackage for sound effects
              __init__.py
              echo.py
              surround.py
              reverse.py
              ...
      filters/                  Subpackage for filters
              __init__.py
              equalizer.py
              vocoder.py
              karaoke.py
              ...
```

When importing the package, Python searches through the directories on `sys.path` looking for the package subdirectory.

The `__init__.py` files are required to make Python treat directories containing the file as packages (unless using a namespace package, a relatively advanced feature). This prevents directories with a common name, such as `string`, from unintentionally hiding valid modules that occur later on the module search path. In the simplest case, `__init__.py` can just be an empty file, but it can also execute initialization code for the package or set the `__all__` variable, described later.

Users of the package can import individual modules from the package, for example:

```
import sound.effects.echo
```

This loads the submodule **`sound.effects.echo`**. It must be referenced with its full name.

```
sound.effects.echo.echofilter(input, output, delay=0.7,
atten=4)
```

An alternative way of importing the submodule is:

```
from sound.effects import echo
```

This also loads the submodule **`echo`**, and makes it available without its package prefix, so it can be used as follows:

```
echo.echofilter(input, output, delay=0.7, atten=4)
```

Yet another variation is to import the desired function or variable directly:

```
from sound.effects.echo import echofilter
```

Again, this loads the submodule **`echo`**, but this makes its function **`echofilter()`** directly available:

```
echofilter(input, output, delay=0.7, atten=4)
```

Note that when using `from package import item`, the item can be either a submodule (or subpackage) of the package, or some other name defined in the package, like a function, class or variable. The `import` statement first tests whether the item is defined in the package; if not, it assumes it is a module and attempts to load it. If it fails to find it, an **`ImportError`** exception is raised.

Contrarily, when using syntax like `import item.subitem.subsubitem`, each item except for the last must be a package; the last item can be a module or a package but can’t be a class or function or variable defined in the previous item.

6.4.1. Importing * From a Package

Now what happens when the user writes `from sound.effects import *`? Ideally, one would hope that this somehow goes out to the filesystem, finds which submodules are present in the package, and imports them all. This could take a long time and importing sub-modules might have unwanted side-effects that should only happen when the sub-module is explicitly imported.

The only solution is for the package author to provide an explicit index of the package. The **`import`** statement uses the following convention: if a package's `__init__.py` code defines a list named `__all__`, it is taken to be the list of module names that should be imported when `from package import *` is encountered. It is up to the package author to keep this list up-to-date when a new version of the package is released. Package authors may also decide not to support it, if they don't see a use for importing * from their package. For example, the file `sound/effects/__init__.py` could contain the following code:

```
__all__ = ["echo", "surround", "reverse"]
```

This would mean that `from sound.effects import *` would import the three named submodules of the **`sound.effects`** package.

Be aware that submodules might become shadowed by locally defined names. For example, if you added a `reverse` function to the `sound/effects/__init__.py` file, the `from sound.effects import *` would only import the two submodules `echo` and `surround`, but *not* the `reverse` submodule, because it is shadowed by the locally defined `reverse` function:

```
__all__ = [
    "echo",      # refers to the 'echo.py' file
    "surround",  # refers to the 'surround.py' file
    "reverse",   # !!! refers to the 'reverse' function now !!!
]

def reverse(msg: str):  # <-- this name shadows the
'reverse.py' submodule
    return msg[::-1]    #     in the case of a 'from
sound.effects import *'
```

If `__all__` is not defined, the statement `from sound.effects import *` does *not* import all submodules from the package **sound.effects** into the current namespace; it only ensures that the package **sound.effects** has been imported (possibly running any initialization code in `__init__.py`) and then imports whatever names are defined in the package. This includes any names defined (and submodules explicitly loaded) by `__init__.py`. It also includes any submodules of the package that were explicitly loaded by previous **import** statements. Consider this code:

```
import sound.effects.echo
import sound.effects.surround
from sound.effects import *
```

In this example, the **echo** and **surround** modules are imported in the current namespace because they are defined in the **sound.effects** package when the `from...import` statement is executed. (This also works when `__all__` is defined.)

Although certain modules are designed to export only names that follow certain patterns when you use `import *`, it is still considered bad practice in production code.

Remember, there is nothing wrong with using `from package import specific_submodule`! In fact, this is the recommended notation unless the importing module needs to use submodules with the same name from different packages.

6.4.2. Intra-package References

When packages are structured into subpackages (as with the **`sound`** package in the example), you can use absolute imports to refer to submodules of siblings packages. For example, if the module **`sound.filters.vocoder`** needs to use the **`echo`** module in the **`sound.effects`** package, it can use `from sound.effects import echo`.

You can also write relative imports, with the `from module import name` form of import statement. These imports use leading dots to indicate the current and parent packages involved in the relative import. From the **`surround`** module for example, you might use:

```
from . import echo
from .. import formats
from ..filters import equalizer
```

Note that relative imports are based on the name of the current module's package. Since the main module does not have a package, modules intended for use as the main module of a Python application must always use absolute imports.

6.4.3. Packages in Multiple Directories

Packages support one more special attribute, **`__path__`**. This is initialized to be a sequence of strings containing the name of the directory holding the package's `__init__.py` before the code in that file is executed. This variable can be modified; doing so affects future searches for modules and subpackages contained in the package.

While this feature is not often needed, it can be used to extend the set of modules found in a package.

Footnotes

[1]

In fact function definitions are also ‘statements’ that are ‘executed’; the execution of a module-level function definition adds the function name to the module’s global namespace.

7. Input and Output

There are several ways to present the output of a program; data can be printed in a human-readable form, or written to a file for future use. This chapter will discuss some of the possibilities.

7.1. Fancier Output Formatting

So far we’ve encountered two ways of writing values: *expression statements* and the **`print()`** function. (A third way is using the **`write()`** method of file objects; the standard output file can be referenced as `sys.stdout`. See the Library Reference for more information on this.)

Often you’ll want more control over the formatting of your output than simply printing space-separated values. There are several ways to format output.

- To use formatted string literals, begin a string with `f` or `F` before the opening quotation mark or triple quotation mark. Inside this string, you can write a Python expression between `{` and `}` characters that can refer to variables or literal values.

  ```
  >>> year = 2016
  >>> event = 'Referendum'
  >>> f'Results of the {year} {event}'
  'Results of the 2016 Referendum'
  ```

- The **`str.format()`** method of strings requires more manual effort. You’ll still use `{` and `}` to mark where a variable will be substituted and can provide detailed formatting directives, but you’ll also need to provide the information to be formatted. In the following code block there are two examples of how to format variables:

```
>>> yes_votes = 42_572_654
>>> total_votes = 85_705_149
>>> percentage = yes_votes / total_votes
>>> '{:-9} YES votes  {:2.2%}'.format(yes_votes,
percentage)
' 42572654 YES votes  49.67%'
```

Notice how the `yes_votes` are padded with spaces and a negative sign only for negative numbers. The example also prints `percentage` multiplied by 100, with 2 decimal places and followed by a percent sign (see Format Specification Mini-Language for details).

- Finally, you can do all the string handling yourself by using string slicing and concatenation operations to create any layout you can imagine. The string type has some methods that perform useful operations for padding strings to a given column width.

When you don’t need fancy output but just want a quick display of some variables for debugging purposes, you can convert any value to a string with the **`repr()`** or **`str()`** functions.

The **`str()`** function is meant to return representations of values which are fairly human-readable, while **`repr()`** is meant to generate representations which can be read by the interpreter (or will force a **`SyntaxError`** if there is no equivalent syntax). For objects which don’t have a particular representation for human consumption, **`str()`** will return the same value as **`repr()`**. Many values, such as numbers or structures like lists and dictionaries, have the same representation using either function. Strings, in particular, have two distinct representations.

Some examples:

```
>>> s = 'Hello, world.'
>>> str(s)
'Hello, world.'
>>> repr(s)
"'Hello, world.'"
>>> str(1/7)
'0.14285714285714285'
>>> x = 10 * 3.25
>>> y = 200 * 200
>>> s = 'The value of x is ' + repr(x) + ', and y is ' +
repr(y) + '...'
>>> print(s)
The value of x is 32.5, and y is 40000...
>>> # The repr() of a string adds string quotes and
backslashes:
>>> hello = 'hello, world\n'
>>> hellos = repr(hello)
>>> print(hellos)
'hello, world\n'
>>> # The argument to repr() may be any Python object:
>>> repr((x, y, ('spam', 'eggs')))
"(32.5, 40000, ('spam', 'eggs'))"
```

The **`string`** module contains support for a simple templating approach based upon regular expressions, via **`string.Template`**. This offers yet another way to substitute values into strings, using placeholders like `$x` and replacing them with values from a dictionary. This syntax is easy to use, although it offers much less control for formatting.

7.1.1. Formatted String Literals

Formatted string literals (also called f-strings for short) let you include the value of Python expressions inside a string by prefixing the string with `f` or `F` and writing expressions as `{expression}`.

An optional format specifier can follow the expression. This allows greater control over how the value is formatted. The following example rounds pi to three places after the decimal:

```
>>> import math
>>> print(f'The value of pi is approximately {math.pi:.3f}.')
The value of pi is approximately 3.142.
```

Passing an integer after the `':'` will cause that field to be a minimum number of characters wide. This is useful for making columns line up.

```
>>> table = {'Sjoerd': 4127, 'Jack': 4098, 'Dcab': 7678}
>>> for name, phone in table.items():
...     print(f'{name:10} ==> {phone:10d}')
...
Sjoerd     ==>       4127
Jack       ==>       4098
Dcab       ==>       7678
```

Other modifiers can be used to convert the value before it is formatted. `'!a'` applies `ascii()`, `'!s'` applies `str()`, and `'!r'` applies `repr()`:

```
>>> animals = 'eels'
>>> print(f'My hovercraft is full of {animals}.')
My hovercraft is full of eels.
>>> print(f'My hovercraft is full of {animals!r}.')
My hovercraft is full of 'eels'.
```

The `=` specifier can be used to expand an expression to the text of the expression, an equal sign, then the representation of the evaluated expression:

```
>>> bugs = 'roaches'
>>> count = 13
>>> area = 'living room'
>>> print(f'Debugging {bugs=} {count=} {area=}')
Debugging bugs='roaches' count=13 area='living room'
```

See self-documenting expressions for more information on the `=` specifier. For a reference on these format specifications, see the reference guide for the Format Specification Mini-Language.

7.1.2. The String format() Method

Basic usage of the **`str.format()`** method looks like this:

```
>>> print('We are the {} who say "{}!"'.format('knights',
'Ni'))
We are the knights who say "Ni!"
```

The brackets and characters within them (called format fields) are replaced with the objects passed into the **`str.format()`** method. A number in the brackets can be used to refer to the position of the object passed into the **`str.format()`** method.

```
>>> print('{0} and {1}'.format('spam', 'eggs'))
spam and eggs
>>> print('{1} and {0}'.format('spam', 'eggs'))
eggs and spam
```

If keyword arguments are used in the **`str.format()`** method, their values are referred to by using the name of the argument.

```
>>> print('This {food} is {adjective}.'.format(
...       food='spam', adjective='absolutely horrible'))
This spam is absolutely horrible.
```

Positional and keyword arguments can be arbitrarily combined:

```
>>> print('The story of {0}, {1}, and {other}.'.format('Bill',
'Manfred',
...
other='Georg'))
The story of Bill, Manfred, and Georg.
```

If you have a really long format string that you don’t want to split up, it would be nice if you could reference the variables to be formatted by name instead of by position. This can be done by simply passing the dict and using square brackets `'[]'` to access the keys.

```
>>> table = {'Sjoerd': 4127, 'Jack': 4098, 'Dcab': 8637678}
>>> print('Jack: {0[Jack]:d}; Sjoerd: {0[Sjoerd]:d}; '
...       'Dcab: {0[Dcab]:d}'.format(table))
Jack: 4098; Sjoerd: 4127; Dcab: 8637678
```

This could also be done by passing the `table` dictionary as keyword arguments with the `**` notation.

```
>>> table = {'Sjoerd': 4127, 'Jack': 4098, 'Dcab': 8637678}
>>> print('Jack: {Jack:d}; Sjoerd: {Sjoerd:d}; Dcab:
{Dcab:d}'.format(**table))
Jack: 4098; Sjoerd: 4127; Dcab: 8637678
```

This is particularly useful in combination with the built-in function `vars()`, which returns a dictionary containing all local variables:

```
>>> table = {k: str(v) for k, v in vars().items()}
>>> message = " ".join([f'{k}: ' + '{' + k +'};' for k in
table.keys()])
>>> print(message.format(**table))
__name__: __main__; __doc__: None; __package__: None;
__loader__: ...
```

As an example, the following lines produce a tidily aligned set of columns giving integers and their squares and cubes:

```
>>> for x in range(1, 11):
...     print('{0:2d} {1:3d} {2:4d}'.format(x, x*x, x*x*x))
...
 1   1    1
 2   4    8
 3   9   27
 4  16   64
 5  25  125
 6  36  216
 7  49  343
 8  64  512
 9  81  729
10 100 1000
```

For a complete overview of string formatting with `str.format()`, see Format String Syntax.

7.1.3. Manual String Formatting

Here’s the same table of squares and cubes, formatted manually:

```
>>> for x in range(1, 11):
...     print(repr(x).rjust(2), repr(x*x).rjust(3), end=' ')
...     # Note use of 'end' on previous line
...     print(repr(x*x*x).rjust(4))
...
 1   1    1
 2   4    8
 3   9   27
 4  16   64
 5  25  125
 6  36  216
 7  49  343
 8  64  512
 9  81  729
10 100 1000
```

(Note that the one space between each column was added by the way `print()` works: it always adds spaces between its arguments.)

The `str.rjust()` method of string objects right-justifies a string in a field of a given width by padding it with spaces on the left. There are similar methods **`str.ljust()`** and **`str.center()`**. These methods do not write anything, they just return a new string. If the input string is too long, they don’t truncate it, but return it unchanged; this will mess up your column lay-out but that’s usually better than the alternative, which would be lying about a value. (If you really want truncation you can always add a slice operation, as in `x.ljust(n)[:n]`.)

There is another method, **`str.zfill()`**, which pads a numeric string on the left with zeros. It understands about plus and minus signs:

```
>>> '12'.zfill(5)
'00012'
>>> '-3.14'.zfill(7)
'-003.14'
>>> '3.14159265359'.zfill(5)
'3.14159265359'
```

7.1.4. Old string formatting

The % operator (modulo) can also be used for string formatting. Given `format % values` (where *format* is a string), `%` conversion specifications in *format* are replaced with zero or more elements of *values*. This operation is commonly known as string interpolation. For example:

```
>>> import math
>>> print('The value of pi is approximately %5.3f.' % math.pi)
The value of pi is approximately 3.142.
```

More information can be found in the printf-style String Formatting section.

7.2. Reading and Writing Files

`open()` returns a file object, and is most commonly used with two positional arguments and one keyword argument: `open(filename, mode, encoding=None)`

```
>>> f = open('workfile', 'w', encoding="utf-8")
```

The first argument is a string containing the filename. The second argument is another string containing a few characters describing the way in which the file will be used. *mode* can be `'r'` when the file will only be read, `'w'` for only writing (an existing file with the same name will be erased), and `'a'` opens the file for appending; any data written to the file is automatically added to the end. `'r+'` opens the file for both reading and writing. The *mode* argument is optional; `'r'` will be assumed if it's omitted.

Normally, files are opened in *text mode*, that means, you read and write strings from and to the file, which are encoded in a specific *encoding*. If *encoding* is not specified, the default is platform dependent (see `open()`). Because UTF-8 is the modern de-facto standard, `encoding="utf-8"` is recommended unless you know that you need to use a different encoding. Appending a `'b'` to the mode opens the file in *binary mode*. Binary mode data is read and written as `bytes` objects. You can not specify *encoding* when opening file in binary mode.

In text mode, the default when reading is to convert platform-specific line endings (`\n` on Unix, `\r\n` on Windows) to just `\n`. When writing in text mode, the default is to convert occurrences of `\n` back to platform-specific line endings. This behind-the-scenes modification to file data is fine for text files, but will corrupt binary data like that in `JPEG` or `EXE` files. Be very careful to use binary mode when reading and writing such files.

It is good practice to use the `with` keyword when dealing with file objects. The advantage is that the file is properly closed after its suite finishes, even if an exception is raised at some point. Using **`with`** is also much shorter than writing equivalent `try`-`finally` blocks:

```
>>> with open('workfile', encoding="utf-8") as f:
...     read_data = f.read()

>>> # We can check that the file has been automatically closed.
>>> f.closed
True
```

If you’re not using the `with` keyword, then you should call `f.close()` to close the file and immediately free up any system resources used by it.

Warning

Calling `f.write()` without using the **`with`** keyword or calling `f.close()` **might** result in the arguments of `f.write()` not being completely written to the disk, even if the program exits successfully.

After a file object is closed, either by a `with` statement or by calling `f.close()`, attempts to use the file object will automatically fail.

```
>>> f.close()
>>> f.read()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: I/O operation on closed file.
```

7.2.1. Methods of File Objects

The rest of the examples in this section will assume that a file object called `f` has already been created.

To read a file’s contents, call `f.read(size)`, which reads some quantity of data and returns it as a string (in text mode) or bytes object (in binary mode). *size* is an optional numeric argument. When *size* is omitted or negative, the entire contents of the file will be read and returned; it’s your problem if the file is twice as large as your machine’s memory. Otherwise, at most *size* characters (in text mode) or *size* bytes (in binary mode) are read and returned. If the end of the file has been reached, `f.read()` will return an empty string (`''`).

```
>>> f.read()
'This is the entire file.\n'
>>> f.read()
''
```

`f.readline()` reads a single line from the file; a newline character (`\n`) is left at the end of the string, and is only omitted on the last line of the file if the file doesn’t end in a newline. This makes the return value unambiguous; if `f.readline()` returns an empty string, the end of the file has been reached, while a blank line is represented by `'\n'`, a string containing only a single newline.

```
>>> f.readline()
'This is the first line of the file.\n'
>>> f.readline()
'Second line of the file\n'
>>> f.readline()
''
```

For reading lines from a file, you can loop over the file object. This is memory efficient, fast, and leads to simple code:

```
>>> for line in f:
...     print(line, end='')
...
This is the first line of the file.
Second line of the file
```

If you want to read all the lines of a file in a list you can also use `list(f)` or `f.readlines()`.

`f.write(string)` writes the contents of *string* to the file, returning the number of characters written.

```
>>> f.write('This is a test\n')
15
```

Other types of objects need to be converted – either to a string (in text mode) or a bytes object (in binary mode) – before writing them:

```
>>> value = ('the answer', 42)
>>> s = str(value)  # convert the tuple to string
>>> f.write(s)
18
```

`f.tell()` returns an integer giving the file object’s current position in the file represented as number of bytes from the beginning of the file when in binary mode and an opaque number when in text mode.

To change the file object’s position, use `f.seek(offset, whence)`. The position is computed from adding *offset* to a reference point; the reference point is selected by the *whence* argument. A *whence* value of 0 measures from the beginning of the file, 1 uses the current file position, and 2 uses the end of the file as the reference point. *whence* can be omitted and defaults to 0, using the beginning of the file as the reference point.

```
>>> f = open('workfile', 'rb+')
>>> f.write(b'0123456789abcdef')
16
>>> f.seek(5)      # Go to the 6th byte in the file
5
>>> f.read(1)
b'5'
>>> f.seek(-3, 2)  # Go to the 3rd byte before the end
13
>>> f.read(1)
b'd'
```

In text files (those opened without a `b` in the mode string), only seeks relative to the beginning of the file are allowed (the exception being seeking to the very file end with `seek(0, 2)`) and the only valid *offset* values are those returned from the `f.tell()`, or zero. Any other *offset* value produces undefined behaviour.

File objects have some additional methods, such as **`isatty()`** and **`truncate()`** which are less frequently used; consult the Library Reference for a complete guide to file objects.

7.2.2. Saving structured data with `json`

Strings can easily be written to and read from a file. Numbers take a bit more effort, since the **`read()`** method only returns strings, which will have to be passed to a function like **`int()`**, which takes a string like `'123'` and returns its numeric value 123. When you want to save more complex data types like nested lists and dictionaries, parsing and serializing by hand becomes complicated.

Rather than having users constantly writing and debugging code to save complicated data types to files, Python allows you to use the popular data interchange format called JSON (JavaScript Object Notation) [https://json.org]. The standard module called **`json`** can take Python data hierarchies, and convert them to string representations; this process is called *serializing*. Reconstructing the data from the string representation is called *deserializing*. Between serializing and deserializing, the string representing the object may have been stored in a file or data, or sent over a network connection to some distant machine.

Note

The JSON format is commonly used by modern applications to allow for data exchange. Many programmers are already familiar with it, which makes it a good choice for interoperability.

If you have an object `x`, you can view its JSON string representation with a simple line of code:

```
>>> import json
>>> x = [1, 'simple', 'list']
>>> json.dumps(x)
'[1, "simple", "list"]'
```

Another variant of the **`dumps()`** function, called **`dump()`**, simply serializes the object to a text file. So if `f` is a text file object opened for writing, we can do this:

```
json.dump(x, f)
```

To decode the object again, if `f` is a binary file or text file object which has been opened for reading:

```
x = json.load(f)
```

Note

JSON files must be encoded in UTF-8. Use `encoding="utf-8"` when opening JSON file as a text file for both of reading and writing.

This simple serialization technique can handle lists and dictionaries, but serializing arbitrary class instances in JSON requires a bit of extra effort. The reference for the `json` module contains an explanation of this.

See also

`pickle` - the pickle module

Contrary to JSON, *pickle* is a protocol which allows the serialization of arbitrarily complex Python objects. As such, it is specific to Python and cannot be used to communicate with applications written in other languages. It is also insecure by default: deserializing pickle data coming from an untrusted source can execute arbitrary code, if the data was crafted by a skilled attacker.

8. Errors and Exceptions

Until now error messages haven’t been more than mentioned, but if you have tried out the examples you have probably seen some. There are (at least) two distinguishable kinds of errors: *syntax errors* and *exceptions*.

8.1. Syntax Errors

Syntax errors, also known as parsing errors, are perhaps the most common kind of complaint you get while you are still learning Python:

```
>>> while True print('Hello world')
  File "<stdin>", line 1
    while True print('Hello world')
               ^^^^^
SyntaxError: invalid syntax
```

The parser repeats the offending line and displays little arrows pointing at the place where the error was detected. Note that this is not always the place that needs to be fixed. In the example, the error is detected at the function `print()`, since a colon (`':'`) is missing just before it.

The file name (`<stdin>` in our example) and line number are printed so you know where to look in case the input came from a file.

8.2. Exceptions

Even if a statement or expression is syntactically correct, it may cause an error when an attempt is made to execute it. Errors detected during execution are called *exceptions* and are not unconditionally fatal: you will soon learn how to handle them in Python programs. Most exceptions are not handled by programs, however, and result in error messages as shown here:

```
>>> 10 * (1/0)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
    10 * (1/0)
          ~^~
ZeroDivisionError: division by zero
>>> 4 + spam*3
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
    4 + spam*3
        ^^^^
NameError: name 'spam' is not defined
>>> '2' + 2
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
    '2' + 2
    ~~~~^~~
TypeError: can only concatenate str (not "int") to str
```

The last line of the error message indicates what happened. Exceptions come in different types, and the type is printed as part of the message: the types in the example are `ZeroDivisionError`, `NameError` and `TypeError`. The string printed as the exception type is the name of the built-in exception that occurred. This is true for all built-in exceptions, but need not be true for user-defined exceptions (although it is a useful convention). Standard exception names are built-in identifiers (not reserved keywords).

The rest of the line provides detail based on the type of exception and what caused it.

The preceding part of the error message shows the context where the exception occurred, in the form of a stack traceback. In general it contains a stack traceback listing source lines; however, it will not display lines read from standard input.

Built-in Exceptions lists the built-in exceptions and their meanings.

8.3. Handling Exceptions

It is possible to write programs that handle selected exceptions. Look at the following example, which asks the user for input until a valid integer has been entered, but allows the user to interrupt the program (using `Control-C` or whatever the operating system supports); note that a user-generated interruption is signalled by raising the **`KeyboardInterrupt`** exception.

```
>>> while True:
...     try:
...         x = int(input("Please enter a number: "))
...         break
...     except ValueError:
...         print("Oops!  That was no valid number.  Try
again...")
...
```

The **`try`** statement works as follows.

- First, the *try clause* (the statement(s) between the **`try`** and **`except`** keywords) is executed.
- If no exception occurs, the *except clause* is skipped and execution of the **`try`** statement is finished.
- If an exception occurs during execution of the **`try`** clause, the rest of the clause is skipped. Then, if its type matches the exception named after the **`except`** keyword, the *except clause* is executed, and then execution continues after the try/except block.

- If an exception occurs which does not match the exception named in the *except clause*, it is passed on to outer **`try`** statements; if no handler is found, it is an *unhandled exception* and execution stops with an error message.

A **`try`** statement may have more than one *except clause*, to specify handlers for different exceptions. At most one handler will be executed. Handlers only handle exceptions that occur in the corresponding *try clause*, not in other handlers of the same **`try`** statement. An *except clause* may name multiple exceptions as a parenthesized tuple, for example:

```
... except (RuntimeError, TypeError, NameError):
...     pass
```

A class in an **`except`** clause matches exceptions which are instances of the class itself or one of its derived classes (but not the other way around — an *except clause* listing a derived class does not match instances of its base classes). For example, the following code will print B, C, D in that order:

```
class B(Exception):
    pass

class C(B):
    pass

class D(C):
    pass

for cls in [B, C, D]:
    try:
        raise cls()
    except D:
        print("D")
    except C:
        print("C")
    except B:
        print("B")
```

Note that if the *except clauses* were reversed (with `except B` first), it would have printed B, B, B — the first matching *except clause* is triggered.

When an exception occurs, it may have associated values, also known as the exception’s *arguments*. The presence and types of the arguments depend on the exception type.

The *except clause* may specify a variable after the exception name. The variable is bound to the exception instance which typically has an `args` attribute that stores the arguments. For convenience, builtin exception types define `__str__()` to print all the arguments without explicitly accessing `.args`.

```
>>> try:
...     raise Exception('spam', 'eggs')
... except Exception as inst:
...     print(type(inst))    # the exception type
...     print(inst.args)     # arguments stored in .args
...     print(inst)          # __str__ allows args to be
printed directly,
...                          # but may be overridden in
exception subclasses
...     x, y = inst.args     # unpack args
...     print('x =', x)
...     print('y =', y)
...
<class 'Exception'>
('spam', 'eggs')
('spam', 'eggs')
x = spam
y = eggs
```

The exception’s `__str__()` output is printed as the last part (‘detail’) of the message for unhandled exceptions.

`BaseException` is the common base class of all exceptions. One of its subclasses, `Exception`, is the base class of all the non-fatal exceptions. Exceptions which are not subclasses of `Exception` are not typically handled, because they are used to indicate that the program should terminate. They include `SystemExit` which is raised by `sys.exit()` and `KeyboardInterrupt` which is raised when a user wishes to interrupt the program.

`Exception` can be used as a wildcard that catches (almost) everything. However, it is good practice to be as specific as possible with the types of exceptions that we intend to handle, and to allow any unexpected exceptions to propagate on.

The most common pattern for handling `Exception` is to print or log the exception and then re-raise it (allowing a caller to handle the exception as well):

```
import sys

try:
    f = open('myfile.txt')
    s = f.readline()
    i = int(s.strip())
except OSError as err:
    print("OS error:", err)
except ValueError:
    print("Could not convert data to an integer.")
except Exception as err:
    print(f"Unexpected {err=}, {type(err)=}")
    raise
```

The `try` … `except` statement has an optional *else clause*, which, when present, must follow all *except clauses*. It is useful for code that must be executed if the *try clause* does not raise an exception. For example:

```
for arg in sys.argv[1:]:
    try:
        f = open(arg, 'r')
    except OSError:
        print('cannot open', arg)
    else:
        print(arg, 'has', len(f.readlines()), 'lines')
        f.close()
```

The use of the **`else`** clause is better than adding additional code to the [`try`] clause because it avoids accidentally catching an exception that wasn't raised by the code being protected by the **`try`** … **`except`** statement.

Exception handlers do not handle only exceptions that occur immediately in the *try clause*, but also those that occur inside functions that are called (even indirectly) in the *try clause*. For example:

```
>>> def this_fails():
...     x = 1/0
...
>>> try:
...     this_fails()
... except ZeroDivisionError as err:
...     print('Handling run-time error:', err)
...
Handling run-time error: division by zero
```

8.4. Raising Exceptions

The **`raise`** statement allows the programmer to force a specified exception to occur. For example:

```
>>> raise NameError('HiThere')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
    raise NameError('HiThere')
NameError: HiThere
```

The sole argument to `raise` indicates the exception to be raised. This must be either an exception instance or an exception class (a class that derives from `BaseException`, such as `Exception` or one of its subclasses). If an exception class is passed, it will be implicitly instantiated by calling its constructor with no arguments:

```
raise ValueError  # shorthand for 'raise ValueError()'
```

If you need to determine whether an exception was raised but don’t intend to handle it, a simpler form of the `raise` statement allows you to re-raise the exception:

```
>>> try:
...     raise NameError('HiThere')
... except NameError:
...     print('An exception flew by!')
...     raise
...
An exception flew by!
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
    raise NameError('HiThere')
NameError: HiThere
```

8.5. Exception Chaining

If an unhandled exception occurs inside an `except` section, it will have the exception being handled attached to it and included in the error message:

```
>>> try:
...     open("database.sqlite")
... except OSError:
...     raise RuntimeError("unable to handle error")
...
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
    open("database.sqlite")
    ~~~~^^^^^^^^^^^^^^^^^^^
FileNotFoundError: [Errno 2] No such file or directory:
'database.sqlite'

During handling of the above exception, another exception
occurred:

Traceback (most recent call last):
  File "<stdin>", line 4, in <module>
    raise RuntimeError("unable to handle error")
RuntimeError: unable to handle error
```

To indicate that an exception is a direct consequence of another, the **`raise`** statement allows an optional **`from`** clause:

```
# exc must be exception instance or None.
raise RuntimeError from exc
```

This can be useful when you are transforming exceptions. For example:

```
>>> def func():
...     raise ConnectionError
...
>>> try:
...     func()
... except ConnectionError as exc:
...     raise RuntimeError('Failed to open database') from exc
...
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
    func()
    ~~~~^^
  File "<stdin>", line 2, in func
ConnectionError

The above exception was the direct cause of the following
exception:

Traceback (most recent call last):
  File "<stdin>", line 4, in <module>
```

```
    raise RuntimeError('Failed to open database') from exc
RuntimeError: Failed to open database
```

It also allows disabling automatic exception chaining using the `from None` idiom:

```
>>> try:
...     open('database.sqlite')
... except OSError:
...     raise RuntimeError from None
...
Traceback (most recent call last):
  File "<stdin>", line 4, in <module>
    raise RuntimeError from None
RuntimeError
```

For more information about chaining mechanics, see Built-in Exceptions.

8.6. User-defined Exceptions

Programs may name their own exceptions by creating a new exception class (see Classes for more about Python classes). Exceptions should typically be derived from the **`Exception`** class, either directly or indirectly.

Exception classes can be defined which do anything any other class can do, but are usually kept simple, often only offering a number of attributes that allow information about the error to be extracted by handlers for the exception.

Most exceptions are defined with names that end in “Error”, similar to the naming of the standard exceptions.

Many standard modules define their own exceptions to report errors that may occur in functions they define.

8.7. Defining Clean-up Actions

The `try` statement has another optional clause which is intended to define clean-up actions that must be executed under all circumstances. For example:

```
>>> try:
...     raise KeyboardInterrupt
... finally:
...     print('Goodbye, world!')
...
Goodbye, world!
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
    raise KeyboardInterrupt
KeyboardInterrupt
```

If a `finally` clause is present, the **`finally`** clause will execute as the last task before the `try` statement completes. The **`finally`** clause runs whether or not the **`try`** statement produces an exception. The following points discuss more complex cases when an exception occurs:

- If an exception occurs during execution of the **`try`** clause, the exception may be handled by an `except` clause. If the exception is not handled by an **`except`** clause, the exception is re-raised after the **`finally`** clause has been executed.
- An exception could occur during execution of an **`except`** or **`else`** clause. Again, the exception is re-raised after the **`finally`** clause has been executed.
- If the **`finally`** clause executes a `break`, `continue` or `return` statement, exceptions are not re-raised. This can be confusing and is therefore discouraged. From version 3.14 the compiler emits a `SyntaxWarning` for it (see **PEP 765** [https://peps.python.org/pep-0765/]).
- If the **`try`** statement reaches a `break`, `continue` or `return` statement, the **`finally`** clause will execute just prior to the **`break`**, **`continue`** or **`return`** statement's execution.

- If a **`finally`** clause includes a **`return`** statement, the returned value will be the one from the **`finally`** clause's **`return`** statement, not the value from the **`try`** clause's **`return`** statement. This can be confusing and is therefore discouraged. From version 3.14 the compiler emits a `SyntaxWarning` for it (see **PEP 765** [https://peps.python.org/pep-0765/]).

For example:

```
>>> def bool_return():
...     try:
...         return True
...     finally:
...         return False
...
>>> bool_return()
False
```

A more complicated example:

```
>>> def divide(x, y):
...     try:
...         result = x / y
...     except ZeroDivisionError:
...         print("division by zero!")
...     else:
...         print("result is", result)
...     finally:
...         print("executing finally clause")
...
>>> divide(2, 1)
result is 2.0
executing finally clause
>>> divide(2, 0)
division by zero!
executing finally clause
>>> divide("2", "1")
executing finally clause
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
```

```
    divide("2", "1")
    ~~~~~~^^^^^^^^^^
  File "<stdin>", line 3, in divide
    result = x / y
             ~~^~~
TypeError: unsupported operand type(s) for /: 'str' and 'str'
```

As you can see, the `finally` clause is executed in any event. The `TypeError` raised by dividing two strings is not handled by the `except` clause and therefore re-raised after the **`finally`** clause has been executed.

In real world applications, the `finally` clause is useful for releasing external resources (such as files or network connections), regardless of whether the use of the resource was successful.

8.8. Predefined Clean-up Actions

Some objects define standard clean-up actions to be undertaken when the object is no longer needed, regardless of whether or not the operation using the object succeeded or failed. Look at the following example, which tries to open a file and print its contents to the screen.

```
for line in open("myfile.txt"):
    print(line, end="")
```

The problem with this code is that it leaves the file open for an indeterminate amount of time after this part of the code has finished executing. This is not an issue in simple scripts, but can be a problem for larger applications. The `with` statement allows objects like files to be used in a way that ensures they are always cleaned up promptly and correctly.

```
with open("myfile.txt") as f:
    for line in f:
        print(line, end="")
```

After the statement is executed, the file *f* is always closed, even if a problem was encountered while processing the lines. Objects which, like files, provide predefined clean-up actions will indicate this in their documentation.

8.9. Raising and Handling Multiple Unrelated Exceptions

There are situations where it is necessary to report several exceptions that have occurred. This is often the case in concurrency frameworks, when several tasks may have failed in parallel, but there are also other use cases where it is desirable to continue execution and collect multiple errors rather than raise the first exception.

The builtin `ExceptionGroup` wraps a list of exception instances so that they can be raised together. It is an exception itself, so it can be caught like any other exception.

```
>>> def f():
...     excs = [OSError('error 1'), SystemError('error 2')]
...     raise ExceptionGroup('there were problems', excs)
...
>>> f()
  + Exception Group Traceback (most recent call last):
  |   File "<stdin>", line 1, in <module>
  |     f()
  |     ~^^
  |   File "<stdin>", line 3, in f
  |     raise ExceptionGroup('there were problems', excs)
  | ExceptionGroup: there were problems (2 sub-exceptions)
  +-+---------------- 1 ----------------
    | OSError: error 1
    +---------------- 2 ----------------
    | SystemError: error 2
    +------------------------------------
>>> try:
...     f()
... except Exception as e:
...     print(f'caught {type(e)}: e')
...
caught <class 'ExceptionGroup'>: e
>>>
```

By using `except*` instead of `except`, we can selectively handle only the exceptions in the group that match a certain type. In the following example, which shows a nested exception group, each `except*` clause extracts from the group exceptions of a certain type while letting all other exceptions propagate to other clauses and eventually to be reraised.

```
>>> def f():
...     raise ExceptionGroup(
...         "group1",
...         [
...             OSError(1),
...             SystemError(2),
...             ExceptionGroup(
...                 "group2",
...                 [
...                     OSError(3),
...                     RecursionError(4)
...                 ]
...             )
...         ]
...     )
...
>>> try:
...     f()
... except* OSError as e:
...     print("There were OSErrors")
... except* SystemError as e:
...     print("There were SystemErrors")
...
There were OSErrors
There were SystemErrors
  + Exception Group Traceback (most recent call last):
  |   File "<stdin>", line 2, in <module>
  |     f()
  |     ~^^
  |   File "<stdin>", line 2, in f
  |     raise ExceptionGroup(
  |     ...<12 lines>...
  |     )
  | ExceptionGroup: group1 (1 sub-exception)
  +-+---------------- 1 ----------------
    | ExceptionGroup: group2 (1 sub-exception)
    +-+---------------- 1 ----------------
      | RecursionError: 4
      +------------------------------------
>>>
```

Note that the exceptions nested in an exception group must be instances, not types. This is because in practice the exceptions would typically be ones that have already been raised and caught by the program, along the following pattern:

```
>>> excs = []
... for test in tests:
...     try:
...         test.run()
...     except Exception as e:
...         excs.append(e)
...
>>> if excs:
...    raise ExceptionGroup("Test Failures", excs)
...
```

8.10. Enriching Exceptions with Notes

When an exception is created in order to be raised, it is usually initialized with information that describes the error that has occurred. There are cases where it is useful to add information after the exception was caught. For this purpose, exceptions have a method `add_note(note)` that accepts a string and adds it to the exception’s notes list.

The standard traceback rendering includes all notes, in the order they were added, after the exception.

```
>>> try:
...     raise TypeError('bad type')
... except Exception as e:
...     e.add_note('Add some information')
...     e.add_note('Add some more information')
...     raise
...
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
    raise TypeError('bad type')
TypeError: bad type
Add some information
Add some more information
>>>
```

For example, when collecting exceptions into an exception group, we may want to add context information for the individual errors. In the following each exception in the group has a note indicating when this error has occurred.

```
>>> def f():
...     raise OSError('operation failed')
...
>>> excs = []
>>> for i in range(3):
...     try:
...         f()
...     except Exception as e:
...         e.add_note(f'Happened in Iteration {i+1}')
...         excs.append(e)
...
>>> raise ExceptionGroup('We have some problems', excs)
  + Exception Group Traceback (most recent call last):
  |   File "<stdin>", line 1, in <module>
  |     raise ExceptionGroup('We have some problems', excs)
  | ExceptionGroup: We have some problems (3 sub-exceptions)
  +-+---------------- 1 ----------------
    | Traceback (most recent call last):
    |   File "<stdin>", line 3, in <module>
    |     f()
    |     ~^^
    |   File "<stdin>", line 2, in f
    |     raise OSError('operation failed')
    | OSError: operation failed
    | Happened in Iteration 1
    +---------------- 2 ----------------
    | Traceback (most recent call last):
    |   File "<stdin>", line 3, in <module>
    |     f()
    |     ~^^
    |   File "<stdin>", line 2, in f
    |     raise OSError('operation failed')
    | OSError: operation failed
    | Happened in Iteration 2
    +---------------- 3 ----------------
    | Traceback (most recent call last):
    |   File "<stdin>", line 3, in <module>
    |     f()
    |     ~^^
    |   File "<stdin>", line 2, in f
    |     raise OSError('operation failed')
    | OSError: operation failed
    | Happened in Iteration 3
    +------------------------------------
>>>
```

9. Classes

Classes provide a means of bundling data and functionality together. Creating a new class creates a new *type* of object, allowing new *instances* of that type to be made. Each class instance can have attributes attached to it for maintaining its state. Class instances can also have methods (defined by its class) for modifying its state.

Compared with other programming languages, Python's class mechanism adds classes with a minimum of new syntax and semantics. It is a mixture of the class mechanisms found in C++ and Modula-3. Python classes provide all the standard features of Object Oriented Programming: the class inheritance mechanism allows multiple base classes, a derived class can override any methods of its base class or classes, and a method can call the method of a base class with the same name. Objects can contain arbitrary amounts and kinds of data. As is true for modules, classes partake of the dynamic nature of Python: they are created at runtime, and can be modified further after creation.

In C++ terminology, normally class members (including the data members) are *public* (except see below Private Variables), and all member functions are *virtual*. As in Modula-3, there are no shorthands for referencing the object's members from its methods: the method function is declared with an explicit first argument representing the object, which is provided implicitly by the call. As in Smalltalk, classes themselves are objects. This provides semantics for importing and renaming. Unlike C++ and Modula-3, built-in types can be used as base classes for extension by the user. Also, like in C++, most built-in operators with special syntax (arithmetic operators, subscripting etc.) can be redefined for class instances.

(Lacking universally accepted terminology to talk about classes, I will make occasional use of Smalltalk and C++ terms. I would use Modula-3 terms, since its object-oriented semantics are closer to those of Python than C++, but I expect that few readers have heard of it.)

9.1. A Word About Names and Objects

Objects have individuality, and multiple names (in multiple scopes) can be bound to the same object. This is known as aliasing in other languages. This is usually not appreciated on a first glance at Python, and can be safely ignored when dealing with immutable basic types (numbers, strings, tuples). However, aliasing has a possibly surprising effect on the semantics of Python code involving mutable objects such as lists, dictionaries, and most other types. This is usually used to the benefit of the program, since aliases behave like pointers in some respects. For example, passing an object is cheap since only a pointer is passed by the implementation; and if a function modifies an object passed as an argument, the caller will see the change — this eliminates the need for two different argument passing mechanisms as in Pascal.

9.2. Python Scopes and Namespaces

Before introducing classes, I first have to tell you something about Python’s scope rules. Class definitions play some neat tricks with namespaces, and you need to know how scopes and namespaces work to fully understand what’s going on. Incidentally, knowledge about this subject is useful for any advanced Python programmer.

Let’s begin with some definitions.

A *namespace* is a mapping from names to objects. Most namespaces are currently implemented as Python dictionaries, but that’s normally not noticeable in any way (except for performance), and it may change in the future. Examples of namespaces are: the set of built-in names (containing functions such as **`abs()`**, and built-in exception names); the global names in a module; and the local names in a function invocation. In a sense the set of attributes of an object also form a namespace. The important thing to know about namespaces is that there is absolutely no relation between names in different namespaces; for instance, two different modules may both define a function `maximize` without confusion — users of the modules must prefix it with the module name.

By the way, I use the word *attribute* for any name following a dot — for example, in the expression `z.real`, `real` is an attribute of the object `z`. Strictly speaking, references to names in modules are attribute references: in the expression `modname.funcname`, `modname` is a module object and `funcname` is an attribute of it. In this case there happens to be a straightforward mapping between the module’s attributes and the global names defined in the module: they share the same namespace! [[1]]

Attributes may be read-only or writable. In the latter case, assignment to attributes is possible. Module attributes are writable: you can write `modname.the_answer = 42`. Writable attributes may also be deleted with the **`del`** statement. For example, `del modname.the_answer` will remove the attribute **`the_answer`** from the object named by `modname`.

Namespaces are created at different moments and have different lifetimes. The namespace containing the built-in names is created when the Python interpreter starts up, and is never deleted. The global namespace for a module is created when the module definition is read in; normally, module namespaces also last until the interpreter quits. The statements executed by the top-level invocation of the interpreter, either read from a script file or interactively, are considered part of a module called `__main__`, so they have their own global namespace. (The built-in names actually also live in a module; this is called `builtins`.)

The local namespace for a function is created when the function is called, and deleted when the function returns or raises an exception that is not handled within the function. (Actually, forgetting would be a better way to describe what actually happens.) Of course, recursive invocations each have their own local namespace.

A *scope* is a textual region of a Python program where a namespace is directly accessible. “Directly accessible” here means that an unqualified reference to a name attempts to find the name in the namespace.

Although scopes are determined statically, they are used dynamically. At any time during execution, there are 3 or 4 nested scopes whose namespaces are directly accessible:

- the innermost scope, which is searched first, contains the local names
- the scopes of any enclosing functions, which are searched starting with the nearest enclosing scope, contain non-local, but also non-global names
- the next-to-last scope contains the current module’s global names
- the outermost scope (searched last) is the namespace containing built-in names

If a name is declared global, then all references and assignments go directly to the next-to-last scope containing the module's global names. To rebind variables found outside of the innermost scope, the `nonlocal` statement can be used; if not declared nonlocal, those variables are read-only (an attempt to write to such a variable will simply create a *new* local variable in the innermost scope, leaving the identically named outer variable unchanged).

Usually, the local scope references the local names of the (textually) current function. Outside functions, the local scope references the same namespace as the global scope: the module's namespace. Class definitions place yet another namespace in the local scope.

It is important to realize that scopes are determined textually: the global scope of a function defined in a module is that module's namespace, no matter from where or by what alias the function is called. On the other hand, the actual search for names is done dynamically, at run time — however, the language definition is evolving towards static name resolution, at "compile" time, so don't rely on dynamic name resolution! (In fact, local variables are already determined statically.)

A special quirk of Python is that – if no `global` or `nonlocal` statement is in effect – assignments to names always go into the innermost scope. Assignments do not copy data — they just bind names to objects. The same is true for deletions: the statement `del x` removes the binding of `x` from the namespace referenced by the local scope. In fact, all operations that introduce new names use the local scope: in particular, `import` statements and function definitions bind the module or function name in the local scope.

The `global` statement can be used to indicate that particular variables live in the global scope and should be rebound there; the `nonlocal` statement indicates that particular variables live in an enclosing scope and should be rebound there.

9.2.1. Scopes and Namespaces Example

This is an example demonstrating how to reference the different scopes and namespaces, and how `global` and `nonlocal` affect variable binding:

```
def scope_test():
    def do_local():
        spam = "local spam"

    def do_nonlocal():
        nonlocal spam
        spam = "nonlocal spam"

    def do_global():
        global spam
        spam = "global spam"

    spam = "test spam"
    do_local()
    print("After local assignment:", spam)
    do_nonlocal()
    print("After nonlocal assignment:", spam)
    do_global()
    print("After global assignment:", spam)

scope_test()
print("In global scope:", spam)
```

The output of the example code is:

```
After local assignment: test spam
After nonlocal assignment: nonlocal spam
After global assignment: nonlocal spam
In global scope: global spam
```

Note how the *local* assignment (which is default) didn't change *scope_test*'s binding of *spam*. The **`nonlocal`** assignment changed *scope_test*'s binding of *spam*, and the **`global`** assignment changed the module-level binding.

You can also see that there was no previous binding for *spam* before the **`global`** assignment.

9.3. A First Look at Classes

Classes introduce a little bit of new syntax, three new object types, and some new semantics.

9.3.1. Class Definition Syntax

The simplest form of class definition looks like this:

```
class ClassName:
    <statement-1>
    .
    .
    .
    <statement-N>
```

Class definitions, like function definitions (**`def`** statements) must be executed before they have any effect. (You could conceivably place a class definition in a branch of an **`if`** statement, or inside a function.)

In practice, the statements inside a class definition will usually be function definitions, but other statements are allowed, and sometimes useful — we'll come back to this later. The function definitions inside a class normally have a peculiar form of argument list, dictated by the calling conventions for methods — again, this is explained later.

When a class definition is entered, a new namespace is created, and used as the local scope — thus, all assignments to local variables go into this new namespace. In particular, function definitions bind the name of the new function here.

When a class definition is left normally (via the end), a *class object* is created. This is basically a wrapper around the contents of the namespace created by the class definition; we’ll learn more about class objects in the next section. The original local scope (the one in effect just before the class definition was entered) is reinstated, and the class object is bound here to the class name given in the class definition header (**`ClassName`** in the example).

9.3.2. Class Objects

Class objects support two kinds of operations: attribute references and instantiation.

Attribute references use the standard syntax used for all attribute references in Python: `obj.name`. Valid attribute names are all the names that were in the class’s namespace when the class object was created. So, if the class definition looked like this:

```
class MyClass:
    """A simple example class"""
    i = 12345

    def f(self):
        return 'hello world'
```

then `MyClass.i` and `MyClass.f` are valid attribute references, returning an integer and a function object, respectively. Class attributes can also be assigned to, so you can change the value of `MyClass.i` by assignment. **`__doc__`** is also a valid attribute, returning the docstring belonging to the class: `"A simple example class"`.

Class *instantiation* uses function notation. Just pretend that the class object is a parameterless function that returns a new instance of the class. For example (assuming the above class):

```
x = MyClass()
```

creates a new *instance* of the class and assigns this object to the local variable `x`.

The instantiation operation (“calling” a class object) creates an empty object. Many classes like to create objects with instances customized to a specific initial state. Therefore a class may define a special method named **`__init__()`**, like this:

```
def __init__(self):
    self.data = []
```

When a class defines an **`__init__()`** method, class instantiation automatically invokes **`__init__()`** for the newly created class instance. So in this example, a new, initialized instance can be obtained by:

```
x = MyClass()
```

Of course, the **`__init__()`** method may have arguments for greater flexibility. In that case, arguments given to the class instantiation operator are passed on to **`__init__()`**. For example,

```
>>> class Complex:
...     def __init__(self, realpart, imagpart):
...         self.r = realpart
...         self.i = imagpart
...
>>> x = Complex(3.0, -4.5)
>>> x.r, x.i
(3.0, -4.5)
```

9.3.3. Instance Objects

Now what can we do with instance objects? The only operations understood by instance objects are attribute references. There are two kinds of valid attribute names: data attributes and methods.

Data attributes correspond to “instance variables” in Smalltalk, and to “data members” in C++. Data attributes need not be declared; like local variables, they spring into existence when they are first assigned to. For example, if `x` is the instance of **`MyClass`** created above, the following piece of code will print the value `16`, without leaving a trace:

```
x.counter = 1
while x.counter < 10:
    x.counter = x.counter * 2
print(x.counter)
del x.counter
```

The other kind of instance attribute reference is a *method*. A method is a function that “belongs to” an object.

Valid method names of an instance object depend on its class. By definition, all attributes of a class that are function objects define corresponding methods of its instances. So in our example, `x.f` is a valid method reference, since `MyClass.f` is a function, but `x.i` is not, since `MyClass.i` is not. But `x.f` is not the same thing as `MyClass.f` — it is a *method object*, not a function object.

9.3.4. Method Objects

Usually, a method is called right after it is bound:

```
x.f()
```

If `x = MyClass()`, as above, this will return the string `'hello world'`. However, it is not necessary to call a method right away: `x.f` is a method object, and can be stored away and called at a later time. For example:

```
xf = x.f
while True:
    print(xf())
```

will continue to print `hello world` until the end of time.

What exactly happens when a method is called? You may have noticed that `x.f()` was called without an argument above, even though the function definition for **`f()`** specified an argument. What happened to the argument? Surely Python raises an exception when a function that requires an argument is called without any — even if the argument isn't actually used…

Actually, you may have guessed the answer: the special thing about methods is that the instance object is passed as the first argument of the function. In our example, the call `x.f()` is exactly equivalent to `MyClass.f(x)`. In general, calling a method with a list of *n* arguments is equivalent to calling the corresponding function with an argument list that is created by inserting the method's instance object before the first argument.

In general, methods work as follows. When a non-data attribute of an instance is referenced, the instance’s class is searched. If the name denotes a valid class attribute that is a function object, references to both the instance object and the function object are packed into a method object. When the method object is called with an argument list, a new argument list is constructed from the instance object and the argument list, and the function object is called with this new argument list.

9.3.5. Class and Instance Variables

Generally speaking, instance variables are for data unique to each instance and class variables are for attributes and methods shared by all instances of the class:

```
class Dog:

    kind = 'canine'         # class variable shared by all
instances

    def __init__(self, name):
        self.name = name    # instance variable unique to each
instance

>>> d = Dog('Fido')
>>> e = Dog('Buddy')
>>> d.kind                  # shared by all dogs
'canine'
>>> e.kind                  # shared by all dogs
'canine'
>>> d.name                  # unique to d
'Fido'
>>> e.name                  # unique to e
'Buddy'
```

As discussed in A Word About Names and Objects, shared data can have possibly surprising effects with involving mutable objects such as lists and dictionaries. For example, the *tricks* list in the following code should not be used as a class variable because just a single list would be shared by all *Dog* instances:

```
class Dog:

    tricks = []             # mistaken use of a class variable

    def __init__(self, name):
        self.name = name

    def add_trick(self, trick):
        self.tricks.append(trick)

>>> d = Dog('Fido')
>>> e = Dog('Buddy')
>>> d.add_trick('roll over')
>>> e.add_trick('play dead')
>>> d.tricks                # unexpectedly shared by all dogs
['roll over', 'play dead']
```

Correct design of the class should use an instance variable instead:

```
class Dog:

    def __init__(self, name):
        self.name = name
        self.tricks = []    # creates a new empty list for each
dog

    def add_trick(self, trick):
        self.tricks.append(trick)

>>> d = Dog('Fido')
>>> e = Dog('Buddy')
>>> d.add_trick('roll over')
>>> e.add_trick('play dead')
>>> d.tricks
['roll over']
>>> e.tricks
['play dead']
```

9.4. Random Remarks

If the same attribute name occurs in both an instance and in a class, then attribute lookup prioritizes the instance:

```
>>> class Warehouse:
...    purpose = 'storage'
...    region = 'west'
...
>>> w1 = Warehouse()
>>> print(w1.purpose, w1.region)
storage west
>>> w2 = Warehouse()
>>> w2.region = 'east'
>>> print(w2.purpose, w2.region)
storage east
```

Data attributes may be referenced by methods as well as by ordinary users (“clients”) of an object. In other words, classes are not usable to implement pure abstract data types. In fact, nothing in Python makes it possible to enforce data hiding — it is all based upon convention. (On the other hand, the Python implementation, written in C, can completely hide implementation details and control access to an object if necessary; this can be used by extensions to Python written in C.)

Clients should use data attributes with care — clients may mess up invariants maintained by the methods by stamping on their data attributes. Note that clients may add data attributes of their own to an instance object without affecting the validity of the methods, as long as name conflicts are avoided — again, a naming convention can save a lot of headaches here.

There is no shorthand for referencing data attributes (or other methods!) from within methods. I find that this actually increases the readability of methods: there is no chance of confusing local variables and instance variables when glancing through a method.

Often, the first argument of a method is called `self`. This is nothing more than a convention: the name `self` has absolutely no special meaning to Python. Note, however, that by not following the convention your code may be less readable to other Python programmers, and it is also conceivable that a *class browser* program might be written that relies upon such a convention.

Any function object that is a class attribute defines a method for instances of that class. It is not necessary that the function definition is textually enclosed in the class definition: assigning a function object to a local variable in the class is also ok. For example:

```
# Function defined outside the class
def f1(self, x, y):
    return min(x, x+y)

class C:
    f = f1

    def g(self):
        return 'hello world'

    h = g
```

Now `f`, `g` and `h` are all attributes of class **`C`** that refer to function objects, and consequently they are all methods of instances of **`C`** — `h` being exactly equivalent to `g`. Note that this practice usually only serves to confuse the reader of a program.

Methods may call other methods by using method attributes of the `self` argument:

```
class Bag:
    def __init__(self):
        self.data = []

    def add(self, x):
        self.data.append(x)

    def addtwice(self, x):
        self.add(x)
        self.add(x)
```

Methods may reference global names in the same way as ordinary functions. The global scope associated with a method is the module containing its definition. (A class is never used as a global scope.) While one rarely encounters a good reason for using global data in a method, there are many legitimate uses of the global scope: for one thing, functions and modules imported into the global scope can be used by methods, as well as functions and classes defined in it. Usually, the class containing the method is itself defined in this global scope, and in the next section we'll find some good reasons why a method would want to reference its own class.

Each value is an object, and therefore has a *class* (also called its *type*). It is stored as `object.__class__`.

9.5. Inheritance

Of course, a language feature would not be worthy of the name "class" without supporting inheritance. The syntax for a derived class definition looks like this:

```
class DerivedClassName(BaseClassName):
    <statement-1>
    .
    .
    .
    <statement-N>
```

The name **`BaseClassName`** must be defined in a namespace accessible from the scope containing the derived class definition. In place of a base class name, other arbitrary expressions are also allowed. This can be useful, for example, when the base class is defined in another module:

```
class DerivedClassName(modname.BaseClassName):
```

Execution of a derived class definition proceeds the same as for a base class. When the class object is constructed, the base class is remembered. This is used for resolving attribute references: if a requested attribute is not found in the class, the search proceeds to look in the base class. This rule is applied recursively if the base class itself is derived from some other class.

There's nothing special about instantiation of derived classes: `DerivedClassName()` creates a new instance of the class. Method references are resolved as follows: the corresponding class attribute is searched, descending down the chain of base classes if necessary, and the method reference is valid if this yields a function object.

Derived classes may override methods of their base classes. Because methods have no special privileges when calling other methods of the same object, a method of a base class that calls another method defined in the same base class may end up calling a method of a derived class that overrides it. (For C++ programmers: all methods in Python are effectively `virtual`.)

An overriding method in a derived class may in fact want to extend rather than simply replace the base class method of the same name. There is a simple way to call the base class method directly: just call `BaseClassName.methodname(self, arguments)`. This is occasionally useful to clients as well. (Note that this only works if the base class is accessible as `BaseClassName` in the global scope.)

Python has two built-in functions that work with inheritance:

- Use **`isinstance()`** to check an instance's type: `isinstance(obj, int)` will be `True` only if `obj.__class__` is **`int`** or some class derived from **`int`**.
- Use **`issubclass()`** to check class inheritance: `issubclass(bool, int)` is `True` since **`bool`** is a subclass of **`int`**. However, `issubclass(float, int)` is `False` since **`float`** is not a subclass of **`int`**.

9.5.1. Multiple Inheritance

Python supports a form of multiple inheritance as well. A class definition with multiple base classes looks like this:

```
class DerivedClassName(Base1, Base2, Base3):
    <statement-1>
    .
    .
    .
    <statement-N>
```

For most purposes, in the simplest cases, you can think of the search for attributes inherited from a parent class as depth-first, left-to-right, not searching twice in the same class where there is an overlap in the hierarchy. Thus, if an attribute is not found in **`DerivedClassName`**, it is searched for in **`Base1`**, then (recursively) in the base classes of **`Base1`**, and if it was not found there, it was searched for in **`Base2`**, and so on.

In fact, it is slightly more complex than that; the method resolution order changes dynamically to support cooperative calls to `super()`. This approach is known in some other multiple-inheritance languages as call-next-method and is more powerful than the super call found in single-inheritance languages.

Dynamic ordering is necessary because all cases of multiple inheritance exhibit one or more diamond relationships (where at least one of the parent classes can be accessed through multiple paths from the bottommost class). For example, all classes inherit from `object`, so any case of multiple inheritance provides more than one path to reach `object`. To keep the base classes from being accessed more than once, the dynamic algorithm linearizes the search order in a way that preserves the left-to-right ordering specified in each class, that calls each parent only once, and that is monotonic (meaning that a class can be subclassed without affecting the precedence order of its parents). Taken together, these properties make it possible to design reliable and extensible classes with multiple inheritance. For more detail, see The Python 2.3 Method Resolution Order.

9.6. Private Variables

“Private” instance variables that cannot be accessed except from inside an object don’t exist in Python. However, there is a convention that is followed by most Python code: a name prefixed with an underscore (e.g. `_spam`) should be treated as a non-public part of the API (whether it is a function, a method or a data member). It should be considered an implementation detail and subject to change without notice.

Since there is a valid use-case for class-private members (namely to avoid name clashes of names with names defined by subclasses), there is limited support for such a mechanism, called *name mangling*. Any identifier of the form `__spam` (at least two leading underscores, at most one trailing underscore) is textually replaced with `_classname__spam`, where `classname` is the current class name with leading underscore(s) stripped. This mangling is done without regard to the syntactic position of the identifier, as long as it occurs within the definition of a class.

See also

The private name mangling specifications for details and special cases.

Name mangling is helpful for letting subclasses override methods without breaking intraclass method calls. For example:

```
class Mapping:
    def __init__(self, iterable):
        self.items_list = []
        self.__update(iterable)

    def update(self, iterable):
        for item in iterable:
            self.items_list.append(item)

    __update = update   # private copy of original update()
method

class MappingSubclass(Mapping):

    def update(self, keys, values):
        # provides new signature for update()
        # but does not break __init__()
        for item in zip(keys, values):
            self.items_list.append(item)
```

The above example would work even if `MappingSubclass` were to introduce a `__update` identifier since it is replaced with `_Mapping__update` in the `Mapping` class and `_MappingSubclass__update` in the `MappingSubclass` class respectively.

Note that the mangling rules are designed mostly to avoid accidents; it still is possible to access or modify a variable that is considered private. This can even be useful in special circumstances, such as in the debugger.

Notice that code passed to `exec()` or `eval()` does not consider the classname of the invoking class to be the current class; this is similar to the effect of the `global` statement, the effect of which is likewise restricted to code that is byte-compiled together. The same restriction applies to `getattr()`, `setattr()` and `delattr()`, as well as when referencing `__dict__` directly.

9.7. Odds and Ends

Sometimes it is useful to have a data type similar to the Pascal “record” or C “struct”, bundling together a few named data items. The idiomatic approach is to use **`dataclasses`** for this purpose:

```
from dataclasses import dataclass

@dataclass
class Employee:
    name: str
    dept: str
    salary: int
```

```
>>> john = Employee('john', 'computer lab', 1000)
>>> john.dept
'computer lab'
>>> john.salary
1000
```

A piece of Python code that expects a particular abstract data type can often be passed a class that emulates the methods of that data type instead. For instance, if you have a function that formats some data from a file object, you can define a class with methods `read()` and `readline()` that get the data from a string buffer instead, and pass it as an argument.

Instance method objects have attributes, too: `m.__self__` is the instance object with the method `m()`, and `m.__func__` is the function object corresponding to the method.

9.8. Iterators

By now you have probably noticed that most container objects can be looped over using a `for` statement:

```
for element in [1, 2, 3]:
    print(element)
for element in (1, 2, 3):
    print(element)
for key in {'one':1, 'two':2}:
    print(key)
for char in "123":
    print(char)
for line in open("myfile.txt"):
    print(line, end='')
```

This style of access is clear, concise, and convenient. The use of iterators pervades and unifies Python. Behind the scenes, the `for` statement calls `iter()` on the container object. The function returns an iterator object that defines the method `__next__()` which accesses elements in the container one at a time. When there are no more elements, `__next__()` raises a `StopIteration` exception which tells the `for` loop to terminate. You can call the `__next__()` method using the `next()` built-in function; this example shows how it all works:

```
>>> s = 'abc'
>>> it = iter(s)
>>> it
<str_iterator object at 0x10c90e650>
>>> next(it)
'a'
>>> next(it)
'b'
>>> next(it)
'c'
>>> next(it)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
    next(it)
StopIteration
```

Having seen the mechanics behind the iterator protocol, it is easy to add iterator behavior to your classes. Define an `__iter__()` method which returns an object with a `__next__()` method. If the class defines **`__next__()`**, then **`__iter__()`** can just return `self`:

```
class Reverse:
    """Iterator for looping over a sequence backwards."""
    def __init__(self, data):
        self.data = data
        self.index = len(data)

    def __iter__(self):
        return self

    def __next__(self):
        if self.index == 0:
            raise StopIteration
        self.index = self.index - 1
        return self.data[self.index]
```

```
>>> rev = Reverse('spam')
>>> iter(rev)
```

```
<__main__.Reverse object at 0x00A1DB50>
>>> for char in rev:
...     print(char)
...
m
a
p
s
```

9.9. Generators

Generators are a simple and powerful tool for creating iterators. They are written like regular functions but use the **`yield`** statement whenever they want to return data. Each time **`next()`** is called on it, the generator resumes where it left off (it remembers all the data values and which statement was last executed). An example shows that generators can be trivially easy to create:

```
def reverse(data):
    for index in range(len(data)-1, -1, -1):
        yield data[index]
```

```
>>> for char in reverse('golf'):
...     print(char)
...
f
l
o
g
```

Anything that can be done with generators can also be done with class-based iterators as described in the previous section. What makes generators so compact is that the **`__iter__()`** and **`__next__()`** methods are created automatically.

Another key feature is that the local variables and execution state are automatically saved between calls. This made the function easier to write and much more clear than an approach using instance variables like `self.index` and `self.data`.

In addition to automatic method creation and saving program state, when generators terminate, they automatically raise **`StopIteration`**. In combination, these features make it easy to create iterators with no more effort than writing a regular function.

9.10. Generator Expressions

Some simple generators can be coded succinctly as expressions using a syntax similar to list comprehensions but with parentheses instead of square brackets. These expressions are designed for situations where the generator is used right away by an enclosing function. Generator expressions are more compact but less versatile than full generator definitions and tend to be more memory friendly than equivalent list comprehensions.

Examples:

```
>>> sum(i*i for i in range(10))                 # sum of
squares
285

>>> xvec = [10, 20, 30]
>>> yvec = [7, 5, 3]
>>> sum(x*y for x,y in zip(xvec, yvec))         # dot product
260

>>> unique_words = set(word for line in page  for word in
line.split())

>>> valedictorian = max((student.gpa, student.name) for student
in graduates)

>>> data = 'golf'
>>> list(data[i] for i in range(len(data)-1, -1, -1))
['f', 'l', 'o', 'g']
```

Footnotes

[1]

Except for one thing. Module objects have a secret read-only attribute called `__dict__` which returns the dictionary used to implement the module’s namespace; the name `__dict__` is an attribute but not a global name. Obviously, using this violates the abstraction of namespace implementation, and should be restricted to things like post-mortem debuggers.

10. Brief Tour of the Standard Library

10.1. Operating System Interface

The **`os`** module provides dozens of functions for interacting with the operating system:

```
>>> import os
>>> os.getcwd()      # Return the current working directory
'C:\\Python314'
>>> os.chdir('/server/accesslogs')   # Change current working
directory
>>> os.system('mkdir today')   # Run the command mkdir in the
system shell
0
```

Be sure to use the `import os` style instead of `from os import *`. This will keep **`os.open()`** from shadowing the built-in **`open()`** function which operates much differently.

The built-in **`dir()`** and **`help()`** functions are useful as interactive aids for working with large modules like **`os`**:

```
>>> import os
>>> dir(os)
<returns a list of all module functions>
>>> help(os)
<returns an extensive manual page created from the module's
docstrings>
```

For daily file and directory management tasks, the **`shutil`** module provides a higher level interface that is easier to use:

```
>>> import shutil
>>> shutil.copyfile('data.db', 'archive.db')
'archive.db'
>>> shutil.move('/build/executables', 'installdir')
'installdir'
```

10.2. File Wildcards

The **`glob`** module provides a function for making file lists from directory wildcard searches:

```
>>> import glob
>>> glob.glob('*.py')
['primes.py', 'random.py', 'quote.py']
```

10.3. Command Line Arguments

Common utility scripts often need to process command line arguments. These arguments are stored in the **`sys`** module's *argv* attribute as a list. For instance, let's take the following `demo.py` file:

```
# File demo.py
import sys
print(sys.argv)
```

Here is the output from running `python demo.py one two three` at the command line:

```
['demo.py', 'one', 'two', 'three']
```

The **`argparse`** module provides a more sophisticated mechanism to process command line arguments. The following script extracts one or more filenames and an optional number of lines to be displayed:

```
import argparse
```

```
parser = argparse.ArgumentParser(
    prog='top',
    description='Show top lines from each file')
parser.add_argument('filenames', nargs='+')
parser.add_argument('-l', '--lines', type=int, default=10)
args = parser.parse_args()
print(args)
```

When run at the command line with `python top.py --lines=5 alpha.txt beta.txt`, the script sets `args.lines` to `5` and `args.filenames` to `['alpha.txt', 'beta.txt']`.

10.4. Error Output Redirection and Program Termination

The **sys** module also has attributes for *stdin*, *stdout*, and *stderr*. The latter is useful for emitting warnings and error messages to make them visible even when *stdout* has been redirected:

```
>>> sys.stderr.write('Warning, log file not found starting a
new one\n')
Warning, log file not found starting a new one
```

The most direct way to terminate a script is to use `sys.exit()`.

10.5. String Pattern Matching

The **re** module provides regular expression tools for advanced string processing. For complex matching and manipulation, regular expressions offer succinct, optimized solutions:

```
>>> import re
>>> re.findall(r'\bf[a-z]*', 'which foot or hand fell fastest')
['foot', 'fell', 'fastest']
```

```
>>> re.sub(r'(\b[a-z]+) \1', r'\1', 'cat in the the hat')
'cat in the hat'
```

When only simple capabilities are needed, string methods are preferred because they are easier to read and debug:

```
>>> 'tea for too'.replace('too', 'two')
'tea for two'
```

10.6. Mathematics

The `math` module gives access to the underlying C library functions for floating-point math:

```
>>> import math
>>> math.cos(math.pi / 4)
0.70710678118654757
>>> math.log(1024, 2)
10.0
```

The `random` module provides tools for making random selections:

```
>>> import random
>>> random.choice(['apple', 'pear', 'banana'])
'apple'
>>> random.sample(range(100), 10)   # sampling without
replacement
[30, 83, 16, 4, 8, 81, 41, 50, 18, 33]
>>> random.random()    # random float from the interval [0.0,
1.0)
0.17970987693706186
>>> random.randrange(6)    # random integer chosen from
range(6)
4
```

The `statistics` module calculates basic statistical properties (the mean, median, variance, etc.) of numeric data:

```
>>> import statistics
>>> data = [2.75, 1.75, 1.25, 0.25, 0.5, 1.25, 3.5]
>>> statistics.mean(data)
1.6071428571428572
>>> statistics.median(data)
1.25
>>> statistics.variance(data)
1.3720238095238095
```

The SciPy project <https://scipy.org> has many other modules for numerical computations.

10.7. Internet Access

There are a number of modules for accessing the internet and processing internet protocols. Two of the simplest are **`urllib.request`** for retrieving data from URLs and **`smtplib`** for sending mail:

```
>>> from urllib.request import urlopen
>>> with urlopen('https://docs.python.org/3/') as response:
...     for line in response:
...         line = line.decode()             # Convert bytes to
a str
...         if 'updated' in line:
...             print(line.rstrip())         # Remove trailing
newline
...
      Last updated on Nov 11, 2025 (20:11 UTC).
```

```
>>> import smtplib
>>> server = smtplib.SMTP('localhost')
>>> server.sendmail('soothsayer@example.org',
'jcaesar@example.org',
... """To: jcaesar@example.org
... From: soothsayer@example.org
...
... Beware the Ides of March.
... """)
>>> server.quit()
```

(Note that the second example needs a mailserver running on localhost.)

10.8. Dates and Times

The **`datetime`** module supplies classes for manipulating dates and times in both simple and complex ways. While date and time arithmetic is supported, the focus of the implementation is on efficient member extraction for output formatting and manipulation. The module also supports objects that are timezone aware.

```
>>> # dates are easily constructed and formatted
>>> from datetime import date
>>> now = date.today()
>>> now
datetime.date(2003, 12, 2)
>>> now.strftime("%m-%d-%y. %d %b %Y is a %A on the %d day of
%B.")
'12-02-03. 02 Dec 2003 is a Tuesday on the 02 day of December.'

>>> # dates support calendar arithmetic
>>> birthday = date(1964, 7, 31)
>>> age = now - birthday
>>> age.days
14368
```

10.9. Data Compression

Common data archiving and compression formats are directly supported by modules including: **`zlib`**, **`gzip`**, **`bz2`**, **`lzma`**, **`zipfile`** and **`tarfile`**.

```
>>> import zlib
>>> s = b'witch which has which witches wrist watch'
>>> len(s)
41
>>> t = zlib.compress(s)
>>> len(t)
37
>>> zlib.decompress(t)
b'witch which has which witches wrist watch'
>>> zlib.crc32(s)
226805979
```

10.10. Performance Measurement

Some Python users develop a deep interest in knowing the relative performance of different approaches to the same problem. Python provides a measurement tool that answers those questions immediately.

For example, it may be tempting to use the tuple packing and unpacking feature instead of the traditional approach to swapping arguments. The **`timeit`** module quickly demonstrates a modest performance advantage:

```
>>> from timeit import Timer
>>> Timer('t=a; a=b; b=t', 'a=1; b=2').timeit()
0.57535828626024577
>>> Timer('a,b = b,a', 'a=1; b=2').timeit()
0.54962537085770791
```

In contrast to **`timeit`**’s fine level of granularity, the **`profile`** and **`pstats`** modules provide tools for identifying time critical sections in larger blocks of code.

10.11. Quality Control

One approach for developing high quality software is to write tests for each function as it is developed and to run those tests frequently during the development process.

The **`doctest`** module provides a tool for scanning a module and validating tests embedded in a program’s docstrings.

Test construction is as simple as cutting-and-pasting a typical call along with its results into the docstring. This improves the documentation by providing the user with an example and it allows the doctest module to make sure the code remains true to the documentation:

```
def average(values):
    """Computes the arithmetic mean of a list of numbers.

    >>> print(average([20, 30, 70]))
    40.0
    """
    return sum(values) / len(values)

import doctest
doctest.testmod()   # automatically validate the embedded tests
```

The **`unittest`** module is not as effortless as the **`doctest`** module, but it allows a more comprehensive set of tests to be maintained in a separate file:

```
import unittest

class TestStatisticalFunctions(unittest.TestCase):

    def test_average(self):
        self.assertEqual(average([20, 30, 70]), 40.0)
        self.assertEqual(round(average([1, 5, 7]), 1), 4.3)
        with self.assertRaises(ZeroDivisionError):
            average([])
        with self.assertRaises(TypeError):
            average(20, 30, 70)

unittest.main()  # Calling from the command line invokes all
tests
```

10.12. Batteries Included

Python has a “batteries included” philosophy. This is best seen through the sophisticated and robust capabilities of its larger packages. For example:

- The `xmlrpc.client` and `xmlrpc.server` modules make implementing remote procedure calls into an almost trivial task. Despite the modules' names, no direct knowledge or handling of XML is needed.
- The `email` package is a library for managing email messages, including MIME and other **RFC 5322** [https://datatracker.ietf.org/doc/html/rfc5322.html]-based message documents. Unlike `smtplib` and `poplib` which actually send and receive messages, the email package has a complete toolset for building or decoding complex message structures (including attachments) and for implementing internet encoding and header protocols.
- The `json` package provides robust support for parsing this popular data interchange format. The `csv` module supports direct reading and writing of files in Comma-Separated Value format, commonly supported by databases and spreadsheets. XML processing is supported by the `xml.etree.ElementTree`, `xml.dom` and `xml.sax` packages. Together, these modules and packages greatly simplify data interchange between Python applications and other tools.
- The `sqlite3` module is a wrapper for the SQLite database library, providing a persistent database that can be updated and accessed using slightly nonstandard SQL syntax.
- Internationalization is supported by a number of modules including `gettext`, `locale`, and the `codecs` package.

11. Brief Tour of the Standard Library — Part II

This second tour covers more advanced modules that support professional programming needs. These modules rarely occur in small scripts.

11.1. Output Formatting

The **`reprlib`** module provides a version of **`repr()`** customized for abbreviated displays of large or deeply nested containers:

```
>>> import reprlib
>>> reprlib.repr(set('supercalifragilisticexpialidocious'))
"{'a', 'c', 'd', 'e', 'f', 'g', ...}"
```

The **`pprint`** module offers more sophisticated control over printing both built-in and user defined objects in a way that is readable by the interpreter. When the result is longer than one line, the “pretty printer” adds line breaks and indentation to more clearly reveal data structure:

```
>>> import pprint
>>> t = [[[['black', 'cyan'], 'white', ['green', 'red']], 
[['magenta',
...     'yellow'], 'blue']]]
...
>>> pprint.pprint(t, width=30)
[[[['black', 'cyan'],
   'white',
   ['green', 'red']],
  [['magenta', 'yellow'],
   'blue']]]
```

The `textwrap` module formats paragraphs of text to fit a given screen width:

```
>>> import textwrap
>>> doc = """The wrap() method is just like fill() except that
it returns
... a list of strings instead of one big string with newlines
to separate
... the wrapped lines."""
...
>>> print(textwrap.fill(doc, width=40))
The wrap() method is just like fill()
except that it returns a list of strings
instead of one big string with newlines
to separate the wrapped lines.
```

The `locale` module accesses a database of culture specific data formats. The grouping attribute of locale’s format function provides a direct way of formatting numbers with group separators:

```
>>> import locale
>>> locale.setlocale(locale.LC_ALL, 'English_United
States.1252')
'English_United States.1252'
>>> conv = locale.localeconv()          # get a mapping of
conventions
>>> x = 1234567.8
>>> locale.format_string("%d", x, grouping=True)
'1,234,567'
>>> locale.format_string("%s%.*f", (conv['currency_symbol'],
...                      conv['frac_digits'], x),
grouping=True)
'$1,234,567.80'
```

11.2. Templating

The `string` module includes a versatile `Template` class with a simplified syntax suitable for editing by end-users. This allows users to customize their applications without having to alter the application.

The format uses placeholder names formed by `$` with valid Python identifiers (alphanumeric characters and underscores). Surrounding the placeholder with braces allows it to be followed by more alphanumeric letters with no intervening spaces. Writing `$$` creates a single escaped `$`:

```
>>> from string import Template
>>> t = Template('${village}folk send $$10 to $cause.')
>>> t.substitute(village='Nottingham', cause='the ditch fund')
'Nottinghamfolk send $10 to the ditch fund.'
```

The **`substitute()`** method raises a **`KeyError`** when a placeholder is not supplied in a dictionary or a keyword argument. For mail-merge style applications, user supplied data may be incomplete and the **`safe_substitute()`** method may be more appropriate — it will leave placeholders unchanged if data is missing:

```
>>> t = Template('Return the $item to $owner.')
>>> d = dict(item='unladen swallow')
>>> t.substitute(d)
Traceback (most recent call last):
  ...
KeyError: 'owner'
>>> t.safe_substitute(d)
'Return the unladen swallow to $owner.'
```

Template subclasses can specify a custom delimiter. For example, a batch renaming utility for a photo browser may elect to use percent signs for placeholders such as the current date, image sequence number, or file format:

```
>>> import time, os.path
>>> photofiles = ['img_1074.jpg', 'img_1076.jpg',
'img_1077.jpg']
>>> class BatchRename(Template):
```

```
...     delimiter = '%'
...
>>> fmt = input('Enter rename style (%d-date %n-seqnum %f-
format):  ')
Enter rename style (%d-date %n-seqnum %f-format):  Ashley_%n%f

>>> t = BatchRename(fmt)
>>> date = time.strftime('%d%b%y')
>>> for i, filename in enumerate(photofiles):
...     base, ext = os.path.splitext(filename)
...     newname = t.substitute(d=date, n=i, f=ext)
...     print('{0} --> {1}'.format(filename, newname))

img_1074.jpg --> Ashley_0.jpg
img_1076.jpg --> Ashley_1.jpg
img_1077.jpg --> Ashley_2.jpg
```

Another application for templating is separating program logic from the details of multiple output formats. This makes it possible to substitute custom templates for XML files, plain text reports, and HTML web reports.

11.3. Working with Binary Data Record Layouts

The **`struct`** module provides **`pack()`** and **`unpack()`** functions for working with variable length binary record formats. The following example shows how to loop through header information in a ZIP file without using the **`zipfile`** module. Pack codes `"H"` and `"I"` represent two and four byte unsigned numbers respectively. The `"<"` indicates that they are standard size and in little-endian byte order:

```
import struct

with open('myfile.zip', 'rb') as f:
    data = f.read()

start = 0
```

```
for i in range(3):                      # show the first 3 file
headers
    start += 14
    fields = struct.unpack('<IIIHH', data[start:start+16])
    crc32, comp_size, uncomp_size, filenamesize, extra_size =
fields

    start += 16
    filename = data[start:start+filenamesize]
    start += filenamesize
    extra = data[start:start+extra_size]
    print(filename, hex(crc32), comp_size, uncomp_size)

    start += extra_size + comp_size     # skip to the next
header
```

11.4. Multi-threading

Threading is a technique for decoupling tasks which are not sequentially dependent. Threads can be used to improve the responsiveness of applications that accept user input while other tasks run in the background. A related use case is running I/O in parallel with computations in another thread.

The following code shows how the high level `threading` module can run tasks in background while the main program continues to run:

```
import threading, zipfile

class AsyncZip(threading.Thread):
    def __init__(self, infile, outfile):
        threading.Thread.__init__(self)
        self.infile = infile
        self.outfile = outfile

    def run(self):
        f = zipfile.ZipFile(self.outfile, 'w',
zipfile.ZIP_DEFLATED)
```

```
        f.write(self.infile)
        f.close()
        print('Finished background zip of:', self.infile)

background = AsyncZip('mydata.txt', 'myarchive.zip')
background.start()
print('The main program continues to run in foreground.')

background.join()    # Wait for the background task to finish
print('Main program waited until background was done.')
```

The principal challenge of multi-threaded applications is coordinating threads that share data or other resources. To that end, the threading module provides a number of synchronization primitives including locks, events, condition variables, and semaphores.

While those tools are powerful, minor design errors can result in problems that are difficult to reproduce. So, the preferred approach to task coordination is to concentrate all access to a resource in a single thread and then use the **`queue`** module to feed that thread with requests from other threads. Applications using **`Queue`** objects for inter-thread communication and coordination are easier to design, more readable, and more reliable.

11.5. Logging

The **`logging`** module offers a full featured and flexible logging system. At its simplest, log messages are sent to a file or to `sys.stderr`:

```
import logging
logging.debug('Debugging information')
logging.info('Informational message')
logging.warning('Warning:config file %s not found',
'server.conf')
```

```
logging.error('Error occurred')
logging.critical('Critical error -- shutting down')
```

This produces the following output:

```
WARNING:root:Warning:config file server.conf not found
ERROR:root:Error occurred
CRITICAL:root:Critical error -- shutting down
```

By default, informational and debugging messages are suppressed and the output is sent to standard error. Other output options include routing messages through email, datagrams, sockets, or to an HTTP Server. New filters can select different routing based on message priority: **`DEBUG`**, **`INFO`**, **`WARNING`**, **`ERROR`**, and **`CRITICAL`**.

The logging system can be configured directly from Python or can be loaded from a user editable configuration file for customized logging without altering the application.

11.6. Weak References

Python does automatic memory management (reference counting for most objects and garbage collection to eliminate cycles). The memory is freed shortly after the last reference to it has been eliminated.

This approach works fine for most applications but occasionally there is a need to track objects only as long as they are being used by something else. Unfortunately, just tracking them creates a reference that makes them permanent. The **`weakref`** module provides tools for tracking objects without creating a reference. When the object is no longer needed, it is automatically removed from a weakref table and a callback is triggered for weakref objects. Typical applications include caching objects that are expensive to create:

```
>>> import weakref, gc
>>> class A:
...     def __init__(self, value):
...         self.value = value
...     def __repr__(self):
...         return str(self.value)
...
>>> a = A(10)                   # create a reference
>>> d = weakref.WeakValueDictionary()
>>> d['primary'] = a            # does not create a reference
>>> d['primary']                # fetch the object if it is
still alive
10
>>> del a                       # remove the one reference
>>> gc.collect()                # run garbage collection right
away
0
>>> d['primary']                # entry was automatically
removed
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
    d['primary']                # entry was automatically
removed
  File "C:/python314/lib/weakref.py", line 46, in __getitem__
    o = self.data[key]()
KeyError: 'primary'
```

11.7. Tools for Working with Lists

Many data structure needs can be met with the built-in list type. However, sometimes there is a need for alternative implementations with different performance trade-offs.

The `array` module provides an `array` object that is like a list that stores only homogeneous data and stores it more compactly. The following example shows an array of numbers stored as two byte unsigned binary numbers (typecode `"H"`) rather than the usual 16 bytes per entry for regular lists of Python int objects:

```
>>> from array import array
>>> a = array('H', [4000, 10, 700, 22222])
>>> sum(a)
26932
>>> a[1:3]
array('H', [10, 700])
```

The **`collections`** module provides a **`deque`** object that is like a list with faster appends and pops from the left side but slower lookups in the middle. These objects are well suited for implementing queues and breadth first tree searches:

```
>>> from collections import deque
>>> d = deque(["task1", "task2", "task3"])
>>> d.append("task4")
>>> print("Handling", d.popleft())
Handling task1
```

```
unsearched = deque([starting_node])
def breadth_first_search(unsearched):
    node = unsearched.popleft()
    for m in gen_moves(node):
        if is_goal(m):
            return m
        unsearched.append(m)
```

In addition to alternative list implementations, the library also offers other tools such as the **`bisect`** module with functions for manipulating sorted lists:

```
>>> import bisect
>>> scores = [(100, 'perl'), (200, 'tcl'), (400, 'lua'), (500,
'python')]
>>> bisect.insort(scores, (300, 'ruby'))
>>> scores
[(100, 'perl'), (200, 'tcl'), (300, 'ruby'), (400, 'lua'),
(500, 'python')]
```

The **`heapq`** module provides functions for implementing heaps based on regular lists. The lowest valued entry is always kept at position zero. This is useful for applications which repeatedly access the smallest element but do not want to run a full list sort:

```
>>> from heapq import heapify, heappop, heappush
>>> data = [1, 3, 5, 7, 9, 2, 4, 6, 8, 0]
>>> heapify(data)                      # rearrange the list
into heap order
>>> heappush(data, -5)                 # add a new entry
>>> [heappop(data) for i in range(3)]  # fetch the three
smallest entries
[-5, 0, 1]
```

11.8. Decimal Floating-Point Arithmetic

The `decimal` module offers a `Decimal` datatype for decimal floating-point arithmetic. Compared to the built-in `float` implementation of binary floating point, the class is especially helpful for

- financial applications and other uses which require exact decimal representation,
- control over precision,
- control over rounding to meet legal or regulatory requirements,
- tracking of significant decimal places, or
- applications where the user expects the results to match calculations done by hand.

For example, calculating a 5% tax on a 70 cent phone charge gives different results in decimal floating point and binary floating point. The difference becomes significant if the results are rounded to the nearest cent:

```
>>> from decimal import *
>>> round(Decimal('0.70') * Decimal('1.05'), 2)
Decimal('0.74')
```

```
>>> round(.70 * 1.05, 2)
0.73
```

The **`Decimal`** result keeps a trailing zero, automatically inferring four place significance from multiplicands with two place significance. Decimal reproduces mathematics as done by hand and avoids issues that can arise when binary floating point cannot exactly represent decimal quantities.

Exact representation enables the **`Decimal`** class to perform modulo calculations and equality tests that are unsuitable for binary floating point:

```
>>> Decimal('1.00') % Decimal('.10')
Decimal('0.00')
>>> 1.00 % 0.10
0.09999999999999995

>>> sum([Decimal('0.1')]*10) == Decimal('1.0')
True
>>> 0.1 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1
== 1.0
False
```

The **`decimal`** module provides arithmetic with as much precision as needed:

```
>>> getcontext().prec = 36
>>> Decimal(1) / Decimal(7)
Decimal('0.142857142857142857142857142857142857')
```

12. Virtual Environments and Packages

12.1. Introduction

Python applications will often use packages and modules that don’t come as part of the standard library. Applications will sometimes need a specific version of a library, because the application may require that a particular bug has been fixed or the application may be written using an obsolete version of the library’s interface.

This means it may not be possible for one Python installation to meet the requirements of every application. If application A needs version 1.0 of a particular module but application B needs version 2.0, then the requirements are in conflict and installing either version 1.0 or 2.0 will leave one application unable to run.

The solution for this problem is to create a virtual environment, a self-contained directory tree that contains a Python installation for a particular version of Python, plus a number of additional packages.

Different applications can then use different virtual environments. To resolve the earlier example of conflicting requirements, application A can have its own virtual environment with version 1.0 installed while application B has another virtual environment with version 2.0. If application B requires a library be upgraded to version 3.0, this will not affect application A’s environment.

12.2. Creating Virtual Environments

The module used to create and manage virtual environments is called `venv`. `venv` will install the Python version from which the command was run (as reported by the `--version` option). For instance, executing the command with `python3.12` will install version 3.12.

To create a virtual environment, decide upon a directory where you want to place it, and run the `venv` module as a script with the directory path:

```
python -m venv tutorial-env
```

This will create the `tutorial-env` directory if it doesn't exist, and also create directories inside it containing a copy of the Python interpreter and various supporting files.

A common directory location for a virtual environment is `.venv`. This name keeps the directory typically hidden in your shell and thus out of the way while giving it a name that explains why the directory exists. It also prevents clashing with `.env` environment variable definition files that some tooling supports.

Once you've created a virtual environment, you may activate it.

On Windows, run:

```
tutorial-env\Scripts\activate
```

On Unix or MacOS, run:

```
source tutorial-env/bin/activate
```

(This script is written for the bash shell. If you use the **csh** or **fish** shells, there are alternate `activate.csh` and `activate.fish` scripts you should use instead.)

Activating the virtual environment will change your shell's prompt to show what virtual environment you're using, and modify the environment so that running `python` will get you that particular version and installation of Python. For example:

```
$ source ~/envs/tutorial-env/bin/activate
(tutorial-env) $ python
Python 3.5.1 (default, May  6 2016, 10:59:36)
  ...
>>> import sys
>>> sys.path
['', '/usr/local/lib/python35.zip', ...,
'~/envs/tutorial-env/lib/python3.5/site-packages']
>>>
```

To deactivate a virtual environment, type:

```
deactivate
```

into the terminal.

12.3. Managing Packages with pip

You can install, upgrade, and remove packages using a program called **pip**. By default `pip` will install packages from the Python Package Index [https://pypi.org]. You can browse the Python Package Index by going to it in your web browser.

`pip` has a number of subcommands: "install", "uninstall", "freeze", etc. (Consult the Installing Python Modules guide for complete documentation for `pip`.)

You can install the latest version of a package by specifying a package's name:

```
(tutorial-env) $ python -m pip install novas
Collecting novas
  Downloading novas-3.1.1.3.tar.gz (136kB)
Installing collected packages: novas
  Running setup.py install for novas
Successfully installed novas-3.1.1.3
```

You can also install a specific version of a package by giving the package name followed by `==` and the version number:

```
(tutorial-env) $ python -m pip install requests==2.6.0
Collecting requests==2.6.0
  Using cached requests-2.6.0-py2.py3-none-any.whl
Installing collected packages: requests
Successfully installed requests-2.6.0
```

If you re-run this command, `pip` will notice that the requested version is already installed and do nothing. You can supply a different version number to get that version, or you can run `python -m pip install --upgrade` to upgrade the package to the latest version:

```
(tutorial-env) $ python -m pip install --upgrade requests
Collecting requests
Installing collected packages: requests
  Found existing installation: requests 2.6.0
    Uninstalling requests-2.6.0:
      Successfully uninstalled requests-2.6.0
Successfully installed requests-2.7.0
```

`python -m pip uninstall` followed by one or more package names will remove the packages from the virtual environment.

`python -m pip show` will display information about a particular package:

```
(tutorial-env) $ python -m pip show requests
---
Metadata-Version: 2.0
Name: requests
Version: 2.7.0
Summary: Python HTTP for Humans.
Home-page: http://python-requests.org
Author: Kenneth Reitz
Author-email: me@kennethreitz.com
License: Apache 2.0
Location: /Users/akuchling/envs/tutorial-
env/lib/python3.4/site-packages
Requires:
```

`python -m pip list` will display all of the packages installed in the virtual environment:

```
(tutorial-env) $ python -m pip list
novas (3.1.1.3)
numpy (1.9.2)
pip (7.0.3)
requests (2.7.0)
setuptools (16.0)
```

`python -m pip freeze` will produce a similar list of the installed packages, but the output uses the format that `python -m pip install` expects. A common convention is to put this list in a `requirements.txt` file:

```
(tutorial-env) $ python -m pip freeze > requirements.txt
(tutorial-env) $ cat requirements.txt
novas==3.1.1.3
numpy==1.9.2
requests==2.7.0
```

The `requirements.txt` can then be committed to version control and shipped as part of an application. Users can then install all the necessary packages with `install -r`:

```
(tutorial-env) $ python -m pip install -r requirements.txt
Collecting novas==3.1.1.3 (from -r requirements.txt (line 1))
```

```
  ...
Collecting numpy==1.9.2 (from -r requirements.txt (line 2))
  ...
Collecting requests==2.7.0 (from -r requirements.txt (line 3))
  ...
Installing collected packages: novas, numpy, requests
  Running setup.py install for novas
Successfully installed novas-3.1.1.3 numpy-1.9.2 requests-2.7.0
```

`pip` has many more options. Consult the Installing Python Modules guide for complete documentation for `pip`. When you’ve written a package and want to make it available on the Python Package Index, consult the Python packaging user guide [https://packaging.python.org/en/latest/tutorials/packaging-projects/].

13. What Now?

Reading this tutorial has probably reinforced your interest in using Python — you should be eager to apply Python to solving your real-world problems. Where should you go to learn more?

This tutorial is part of Python’s documentation set. Some other documents in the set are:

- The Python Standard Library:

  You should browse through this manual, which gives complete (though terse) reference material about types, functions, and the modules in the standard library. The standard Python distribution includes a *lot* of additional code. There are modules to read Unix mailboxes, retrieve documents via HTTP, generate random numbers, parse command-line options, compress data, and many other tasks. Skimming through the Library Reference will give you an idea of what’s available.

- Installing Python Modules explains how to install additional modules written by other Python users.

- The Python Language Reference: A detailed explanation of Python’s syntax and semantics. It’s heavy reading, but is useful as a complete guide to the language itself.

More Python resources:

- https://www.python.org: The major Python website. It contains code, documentation, and pointers to Python-related pages around the web.

- https://docs.python.org: Fast access to Python's documentation.
- https://pypi.org: The Python Package Index, previously also nicknamed the Cheese Shop [[1]], is an index of user-created Python modules that are available for download. Once you begin releasing code, you can register it here so that others can find it.
- https://code.activestate.com/recipes/langs/python/: The Python Cookbook is a sizable collection of code examples, larger modules, and useful scripts. Particularly notable contributions are collected in a book also titled Python Cookbook (O'Reilly & Associates, ISBN 0-596-00797-3.)
- https://pyvideo.org collects links to Python-related videos from conferences and user-group meetings.
- https://scipy.org: The Scientific Python project includes modules for fast array computations and manipulations plus a host of packages for such things as linear algebra, Fourier transforms, non-linear solvers, random number distributions, statistical analysis and the like.

For Python-related questions and problem reports, you can post to the newsgroup *comp.lang.python*, or send them to the mailing list at python-list@python.org. The newsgroup and mailing list are gatewayed, so messages posted to one will automatically be forwarded to the other. There are hundreds of postings a day, asking (and answering) questions, suggesting new features, and announcing new modules. Mailing list archives are available at https://mail.python.org/pipermail/.

Before posting, be sure to check the list of Frequently Asked Questions (also called the FAQ). The FAQ answers many of the questions that come up again and again, and may already contain the solution for your problem.

Footnotes

[1]

“Cheese Shop” is a Monty Python’s sketch: a customer enters a cheese shop, but whatever cheese he asks for, the clerk says it’s missing.

14. Interactive Input Editing and History Substitution

Some versions of the Python interpreter support editing of the current input line and history substitution, similar to facilities found in the Korn shell and the GNU Bash shell. This is implemented using the GNU Readline [https://tiswww.case.edu/php/chet/readline/rltop.html] library, which supports various styles of editing. This library has its own documentation which we won’t duplicate here.

14.1. Tab Completion and History Editing

Completion of variable and module names is automatically enabled at interpreter startup so that the `Tab` key invokes the completion function; it looks at Python statement names, the current local variables, and the available module names. For dotted expressions such as `string.a`, it will evaluate the expression up to the final `'.'` and then suggest completions from the attributes of the resulting object. Note that this may execute application-defined code if an object with a **`__getattr__()`** method is part of the expression. The default configuration also saves your history into a file named `.python_history` in your user directory. The history will be available again during the next interactive interpreter session.

14.2. Alternatives to the Interactive Interpreter

This facility is an enormous step forward compared to earlier versions of the interpreter; however, some wishes are left: It would be nice if the proper indentation were suggested on continuation lines (the parser knows if an **`INDENT`** token is required next). The completion mechanism might use the interpreter’s symbol table. A command to check (or even suggest) matching parentheses, quotes, etc., would also be useful.

One alternative enhanced interactive interpreter that has been around for quite some time is IPython [https://ipython.org/], which features tab completion, object exploration and advanced history management. It can also be thoroughly customized and embedded into other applications. Another similar enhanced interactive environment is bpython [https://bpython-interpreter.org/].

15. Floating-Point Arithmetic: Issues and Limitations

Floating-point numbers are represented in computer hardware as base 2 (binary) fractions. For example, the **decimal** fraction `0.625` has value 6/10 + 2/100 + 5/1000, and in the same way the **binary** fraction `0.101` has value 1/2 + 0/4 + 1/8. These two fractions have identical values, the only real difference being that the first is written in base 10 fractional notation, and the second in base 2.

Unfortunately, most decimal fractions cannot be represented exactly as binary fractions. A consequence is that, in general, the decimal floating-point numbers you enter are only approximated by the binary floating-point numbers actually stored in the machine.

The problem is easier to understand at first in base 10. Consider the fraction 1/3. You can approximate that as a base 10 fraction:

```
0.3
```

or, better,

```
0.33
```

or, better,

```
0.333
```

and so on. No matter how many digits you’re willing to write down, the result will never be exactly 1/3, but will be an increasingly better approximation of 1/3.

In the same way, no matter how many base 2 digits you’re willing to use, the decimal value 0.1 cannot be represented exactly as a base 2 fraction. In base 2, 1/10 is the infinitely repeating fraction

```
0.0001100110011001100110011001100110011001100110011...
```

Stop at any finite number of bits, and you get an approximation. On most machines today, floats are approximated using a binary fraction with the numerator using the first 53 bits starting with the most significant bit and with the denominator as a power of two. In the case of 1/10, the binary fraction is `3602879701896397 / 2 ** 55` which is close to but not exactly equal to the true value of 1/10.

Many users are not aware of the approximation because of the way values are displayed. Python only prints a decimal approximation to the true decimal value of the binary approximation stored by the machine. On most machines, if Python were to print the true decimal value of the binary approximation stored for 0.1, it would have to display:

```
>>> 0.1
0.1000000000000000055511151231257827021181583404541015625
```

That is more digits than most people find useful, so Python keeps the number of digits manageable by displaying a rounded value instead:

```
>>> 1 / 10
0.1
```

Just remember, even though the printed result looks like the exact value of 1/10, the actual stored value is the nearest representable binary fraction.

Interestingly, there are many different decimal numbers that share the same nearest approximate binary fraction. For example, the numbers `0.1` and `0.10000000000000001` and `0.1000000000000000055511151231257827021181583404541015625` are all approximated by `3602879701896397 / 2 ** 55`. Since all of these decimal values share the same approximation, any one of them could be displayed while still preserving the invariant `eval(repr(x)) == x`.

Historically, the Python prompt and built-in `repr()` function would choose the one with 17 significant digits, `0.10000000000000001`. Starting with Python 3.1, Python (on most systems) is now able to choose the shortest of these and simply display `0.1`.

Note that this is in the very nature of binary floating point: this is not a bug in Python, and it is not a bug in your code either. You'll see the same kind of thing in all languages that support your hardware's floating-point arithmetic (although some languages may not *display* the difference by default, or in all output modes).

For more pleasant output, you may wish to use string formatting to produce a limited number of significant digits:

```
>>> format(math.pi, '.12g')  # give 12 significant digits
'3.14159265359'

>>> format(math.pi, '.2f')   # give 2 digits after the point
'3.14'

>>> repr(math.pi)
'3.141592653589793'
```

It's important to realize that this is, in a real sense, an illusion: you're simply rounding the *display* of the true machine value.

One illusion may beget another. For example, since 0.1 is not exactly 1/10, summing three values of 0.1 may not yield exactly 0.3, either:

```
>>> 0.1 + 0.1 + 0.1 == 0.3
False
```

Also, since the 0.1 cannot get any closer to the exact value of 1/10 and 0.3 cannot get any closer to the exact value of 3/10, then pre-rounding with **`round()`** function cannot help:

```
>>> round(0.1, 1) + round(0.1, 1) + round(0.1, 1) == round(0.3, 
1)
False
```

Though the numbers cannot be made closer to their intended exact values, the **`math.isclose()`** function can be useful for comparing inexact values:

```
>>> math.isclose(0.1 + 0.1 + 0.1, 0.3)
True
```

Alternatively, the **`round()`** function can be used to compare rough approximations:

```
>>> round(math.pi, ndigits=2) == round(22 / 7, ndigits=2)
True
```

Binary floating-point arithmetic holds many surprises like this. The problem with "0.1" is explained in precise detail below, in the "Representation Error" section. See Examples of Floating Point Problems [https://jvns.ca/blog/2023/01/13/examples-of-floating-point-problems/] for a pleasant summary of how binary floating point works and the kinds of problems commonly encountered in practice. Also see The Perils of Floating Point [http://www.indowsway.com/floatingpoint.htm] for a more complete account of other common surprises.

As that says near the end, “there are no easy answers.” Still, don’t be unduly wary of floating point! The errors in Python float operations are inherited from the floating-point hardware, and on most machines are on the order of no more than 1 part in 2**53 per operation. That’s more than adequate for most tasks, but you do need to keep in mind that it’s not decimal arithmetic and that every float operation can suffer a new rounding error.

While pathological cases do exist, for most casual use of floating-point arithmetic you’ll see the result you expect in the end if you simply round the display of your final results to the number of decimal digits you expect. `str()` usually suffices, and for finer control see the `str.format()` method’s format specifiers in Format String Syntax.

For use cases which require exact decimal representation, try using the `decimal` module which implements decimal arithmetic suitable for accounting applications and high-precision applications.

Another form of exact arithmetic is supported by the `fractions` module which implements arithmetic based on rational numbers (so the numbers like 1/3 can be represented exactly).

If you are a heavy user of floating-point operations you should take a look at the NumPy package and many other packages for mathematical and statistical operations supplied by the SciPy project. See <https://scipy.org>.

Python provides tools that may help on those rare occasions when you really *do* want to know the exact value of a float. The `float.as_integer_ratio()` method expresses the value of a float as a fraction:

```
>>> x = 3.14159
>>> x.as_integer_ratio()
(3537115888337719, 1125899906842624)
```

Since the ratio is exact, it can be used to losslessly recreate the original value:

```
>>> x == 3537115888337719 / 1125899906842624
True
```

The **`float.hex()`** method expresses a float in hexadecimal (base 16), again giving the exact value stored by your computer:

```
>>> x.hex()
'0x1.921f9f01b866ep+1'
```

This precise hexadecimal representation can be used to reconstruct the float value exactly:

```
>>> x == float.fromhex('0x1.921f9f01b866ep+1')
True
```

Since the representation is exact, it is useful for reliably porting values across different versions of Python (platform independence) and exchanging data with other languages that support the same format (such as Java and C99).

Another helpful tool is the **`sum()`** function which helps mitigate loss-of-precision during summation. It uses extended precision for intermediate rounding steps as values are added onto a running total. That can make a difference in overall accuracy so that the errors do not accumulate to the point where they affect the final total:

```
>>> 0.1 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1
== 1.0
False
```

```
>>> sum([0.1] * 10) == 1.0
True
```

The `math.fsum()` goes further and tracks all of the “lost digits” as values are added onto a running total so that the result has only a single rounding. This is slower than `sum()` but will be more accurate in uncommon cases where large magnitude inputs mostly cancel each other out leaving a final sum near zero:

```
>>> arr = [-0.10430216751806065, -266310978.67179024,
143401161448607.16,
...        -143401161400469.7, 266262841.31058735,
-0.003244936839808227]
>>> float(sum(map(Fraction, arr)))   # Exact summation with
single rounding
8.042173697819788e-13
>>> math.fsum(arr)                   # Single rounding
8.042173697819788e-13
>>> sum(arr)                         # Multiple roundings in
extended precision
8.042178034628478e-13
>>> total = 0.0
>>> for x in arr:
...     total += x                   # Multiple roundings in
standard precision
...
>>> total                            # Straight addition has no
correct digits!
-0.0051575902860057365
```

15.1. Representation Error

This section explains the “0.1” example in detail, and shows how you can perform an exact analysis of cases like this yourself. Basic familiarity with binary floating-point representation is assumed.

Representation error refers to the fact that some (most, actually) decimal fractions cannot be represented exactly as binary (base 2) fractions. This is the chief reason why Python (or Perl, C, C++, Java, Fortran, and many others) often won't display the exact decimal number you expect.

Why is that? 1/10 is not exactly representable as a binary fraction. Since at least 2000, almost all machines use IEEE 754 binary floating-point arithmetic, and almost all platforms map Python floats to IEEE 754 binary64 "double precision" values. IEEE 754 binary64 values contain 53 bits of precision, so on input the computer strives to convert 0.1 to the closest fraction it can of the form *J*/2***N* where *J* is an integer containing exactly 53 bits. Rewriting

```
1 / 10 ~= J / (2**N)
```

as

```
J ~= 2**N / 10
```

and recalling that *J* has exactly 53 bits (is `>= 2**52` but `< 2**53`), the best value for *N* is 56:

```
>>> 2**52 <=  2**56 // 10  < 2**53
True
```

That is, 56 is the only value for *N* that leaves *J* with exactly 53 bits. The best possible value for *J* is then that quotient rounded:

```
>>> q, r = divmod(2**56, 10)
>>> r
6
```

Since the remainder is more than half of 10, the best approximation is obtained by rounding up:

```
>>> q+1
7205759403792794
```

Therefore the best possible approximation to 1/10 in IEEE 754 double precision is:

```
7205759403792794 / 2 ** 56
```

Dividing both the numerator and denominator by two reduces the fraction to:

```
3602879701896397 / 2 ** 55
```

Note that since we rounded up, this is actually a little bit larger than 1/10; if we had not rounded up, the quotient would have been a little bit smaller than 1/10. But in no case can it be *exactly* 1/10!

So the computer never “sees” 1/10: what it sees is the exact fraction given above, the best IEEE 754 double approximation it can get:

```
>>> 0.1 * 2 ** 55
3602879701896397.0
```

If we multiply that fraction by 10**55, we can see the value out to 55 decimal digits:

```
>>> 3602879701896397 * 10 ** 55 // 2 ** 55
1000000000000000055511151231257827021181583404541015625
```

meaning that the exact number stored in the computer is equal to the decimal value 0.1000000000000000055511151231257827021181583404541015625. Instead of displaying the full decimal value, many languages (including older versions of Python), round the result to 17 significant digits:

```
>>> format(0.1, '.17f')
'0.10000000000000001'
```

The **`fractions`** and **`decimal`** modules make these calculations easy:

```
>>> from decimal import Decimal
>>> from fractions import Fraction

>>> Fraction.from_float(0.1)
Fraction(3602879701896397, 36028797018963968)

>>> (0.1).as_integer_ratio()
(3602879701896397, 36028797018963968)

>>> Decimal.from_float(0.1)
Decimal('0.1000000000000000055511151231257827021181583404541015
625')

>>> format(Decimal.from_float(0.1), '.17')
'0.10000000000000001'
```

16. Appendix

16.1. Interactive Mode

There are two variants of the interactive REPL. The classic basic interpreter is supported on all platforms with minimal line control capabilities.

Since Python 3.13, a new interactive shell is used by default. This one supports color, multiline editing, history browsing, and paste mode. To disable color, see Controlling color for details. Function keys provide some additional functionality. `F1` enters the interactive help browser **`pydoc`**. `F2` allows for browsing command-line history with neither output nor the >>> and ... prompts. `F3` enters “paste mode”, which makes pasting larger blocks of code easier. Press `F3` to return to the regular prompt.

When using the new interactive shell, exit the shell by typing `exit` or `quit`. Adding call parentheses after those commands is not required.

If the new interactive shell is not desired, it can be disabled via the **`PYTHON_BASIC_REPL`** environment variable.

16.1.1. Error Handling

When an error occurs, the interpreter prints an error message and a stack trace. In interactive mode, it then returns to the primary prompt; when input came from a file, it exits with a nonzero exit status after printing the stack trace. (Exceptions handled by an **`except`** clause in a **`try`** statement are not errors in this context.) Some errors are unconditionally fatal and cause an exit with a nonzero exit status; this applies to internal inconsistencies and some cases of running out of memory. All error messages are written to the standard error stream; normal output from executed commands is written to standard output.

Typing the interrupt character (usually `Control-C` or `Delete`) to the primary or secondary prompt cancels the input and returns to the primary prompt. [[1]] Typing an interrupt while a command is executing raises the **`KeyboardInterrupt`** exception, which may be handled by a **`try`** statement.

16.1.2. Executable Python Scripts

On BSD’ish Unix systems, Python scripts can be made directly executable, like shell scripts, by putting the line

```
#!/usr/bin/env python3
```

(assuming that the interpreter is on the user’s **`PATH`**) at the beginning of the script and giving the file an executable mode. The `#!` must be the first two characters of the file. On some platforms, this first line must end with a Unix-style line ending (`'\n'`), not a Windows (`'\r\n'`) line ending. Note that the hash, or pound, character, `'#'`, is used to start a comment in Python.

The script can be given an executable mode, or permission, using the **chmod** command.

```
$ chmod +x myscript.py
```

On Windows systems, there is no notion of an “executable mode”. The Python installer automatically associates `.py` files with `python.exe` so that a double-click on a Python file will run it as a script. The extension can also be `.pyw`, in that case, the console window that normally appears is suppressed.

16.1.3. The Interactive Startup File

When you use Python interactively, it is frequently handy to have some standard commands executed every time the interpreter is started. You can do this by setting an environment variable named **`PYTHONSTARTUP`** to the name of a file containing your start-up commands. This is similar to the `.profile` feature of the Unix shells.

This file is only read in interactive sessions, not when Python reads commands from a script, and not when `/dev/tty` is given as the explicit source of commands (which otherwise behaves like an interactive session). It is executed in the same namespace where interactive commands are executed, so that objects that it defines or imports can be used without qualification in the interactive session. You can also change the prompts `sys.ps1` and `sys.ps2` in this file.

If you want to read an additional start-up file from the current directory, you can program this in the global start-up file using code like `if os.path.isfile('.pythonrc.py'): exec(open('.pythonrc.py').read())`. If you want to use the startup file in a script, you must do this explicitly in the script:

```
import os
filename = os.environ.get('PYTHONSTARTUP')
if filename and os.path.isfile(filename):
    with open(filename) as fobj:
        startup_file = fobj.read()
    exec(startup_file)
```

16.1.4. The Customization Modules

Python provides two hooks to let you customize it: sitecustomize and usercustomize. To see how it works, you need first to find the location of your user site-packages directory. Start Python and run this code:

```
>>> import site
>>> site.getusersitepackages()
'/home/user/.local/lib/python3.x/site-packages'
```

Now you can create a file named `usercustomize.py` in that directory and put anything you want in it. It will affect every invocation of Python, unless it is started with the **`-s`** option to disable the automatic import.

sitecustomize works in the same way, but is typically created by an administrator of the computer in the global site-packages directory, and is imported before usercustomize. See the documentation of the **`site`** module for more details.

Footnotes

---

[1] A problem with the GNU Readline package may prevent this.

Python Setup and Usage

This part of the documentation is devoted to general information on the setup of the Python environment on different platforms, the invocation of the interpreter and things that make working with Python easier.

- 1. Command line and environment
  - 1.1. Command line
  - 1.2. Environment variables
- 2. Using Python on Unix platforms
  - 2.1. Getting and installing the latest version of Python
  - 2.2. Building Python
  - 2.3. Python-related paths and files
  - 2.4. Miscellaneous
  - 2.5. Custom OpenSSL
- 3. Configure Python
  - 3.1. Build Requirements
  - 3.2. Generated files
  - 3.3. Configure Options
  - 3.4. Python Build System
  - 3.5. Compiler and linker flags
- 4. Using Python on Windows
  - 4.1. Python install manager
  - 4.2. The embeddable package
  - 4.3. The nuget.org packages
  - 4.4. Alternative bundles
  - 4.5. Supported Windows versions
  - 4.6. Removing the MAX_PATH limitation
  - 4.7. UTF-8 mode
  - 4.8. Finding modules
  - 4.9. Additional modules

  - 4.10. Compiling Python on Windows
  - 4.11. The full installer (deprecated)
  - 4.12. Python launcher for Windows (deprecated)
- 5. Using Python on macOS
  - 5.1. Using Python for macOS from **`python.org`**
  - 5.2. Alternative Distributions
  - 5.3. Installing Additional Python Packages
  - 5.4. GUI Programming
  - 5.5. Advanced Topics
  - 5.6. Other Resources
- 6. Using Python on Android
  - 6.1. Adding Python to an Android app
  - 6.2. Building a Python package for Android
- 7. Using Python on iOS
  - 7.1. Python at runtime on iOS
  - 7.2. Installing Python on iOS
  - 7.3. App Store Compliance
- 8. Editors and IDEs
  - 8.1. IDLE — Python editor and shell
  - 8.2. Other Editors and IDEs

1. Command line and environment

The CPython interpreter scans the command line and the environment for various settings.

CPython implementation detail: Other implementations' command line schemes may differ. See Alternate Implementations for further resources.

1.1. Command line

When invoking Python, you may specify any of these options:

```
python [-bBdEhiIOPqRsSuvVWx?] [-c command | -m module-name |
script | - ] [args]
```

The most common use case is, of course, a simple invocation of a script:

```
python myscript.py
```

1.1.1. Interface options

The interpreter interface resembles that of the UNIX shell, but provides some additional methods of invocation:

- When called with standard input connected to a tty device, it prompts for commands and executes them until an EOF (an end-of-file character, you can produce that with `Ctrl-D` on UNIX or `Ctrl-Z`, `Enter` on Windows) is read. For more on interactive mode, see Interactive Mode.

- When called with a file name argument or with a file as standard input, it reads and executes a script from that file.
- When called with a directory name argument, it reads and executes an appropriately named script from that directory.
- When called with `-c command`, it executes the Python statement(s) given as *command*. Here *command* may contain multiple statements separated by newlines. Leading whitespace is significant in Python statements!
- When called with `-m module-name`, the given module is located on the Python module path and executed as a script.

In non-interactive mode, the entire input is parsed before it is executed.

An interface option terminates the list of options consumed by the interpreter, all consecutive arguments will end up in **`sys.argv`** – note that the first element, subscript zero (`sys.argv[0]`), is a string reflecting the program’s source.

-c <command>

Execute the Python code in *command*. *command* can be one or more statements separated by newlines, with significant leading whitespace as in normal module code.

If this option is given, the first element of **`sys.argv`** will be `"-c"` and the current directory will be added to the start of **`sys.path`** (allowing modules in that directory to be imported as top level modules).

Raises an auditing event `cpython.run_command` with argument `command`.

Changed in version 3.14: command is automatically dedented before execution.

-m <module-name>

Search `sys.path` for the named module and execute its contents as the `__main__` module.

Since the argument is a *module* name, you must not give a file extension (`.py`). The module name should be a valid absolute Python module name, but the implementation may not always enforce this (e.g. it may allow you to use a name that includes a hyphen).

Package names (including namespace packages) are also permitted. When a package name is supplied instead of a normal module, the interpreter will execute `<pkg>.__main__` as the main module. This behaviour is deliberately similar to the handling of directories and zipfiles that are passed to the interpreter as the script argument.

Note

This option cannot be used with built-in modules and extension modules written in C, since they do not have Python module files. However, it can still be used for precompiled modules, even if the original source file is not available.

If this option is given, the first element of `sys.argv` will be the full path to the module file (while the module file is being located, the first element will be set to `"-m"`). As with the `-c` option, the current directory will be added to the start of `sys.path`.

`-I` option can be used to run the script in isolated mode where `sys.path` contains neither the current directory nor the user's site-packages directory. All `PYTHON*` environment variables are ignored, too.

Many standard library modules contain code that is invoked on their execution as a script. An example is the **`timeit`** module:

```
python -m timeit -s "setup here" "benchmarked code here"
python -m timeit -h # for details
```

Raises an auditing event `cpython.run_module` with argument `module-name`.

See also

`runpy.run_module()`
Equivalent functionality directly available to Python code

PEP 338 [https://peps.python.org/pep-0338/] – Executing modules as scripts

Changed in version 3.1: Supply the package name to run a `__main__` submodule.

Changed in version 3.4: namespace packages are also supported

-

Read commands from standard input (**`sys.stdin`**). If standard input is a terminal, **`-i`** is implied.

If this option is given, the first element of **`sys.argv`** will be `"-"` and the current directory will be added to the start of **`sys.path`**.

Raises an auditing event `cpython.run_stdin` with no arguments.

<script>

Execute the Python code contained in *script*, which must be a filesystem path (absolute or relative) referring to either a Python file, a directory containing a `__main__.py` file, or a zipfile containing a `__main__.py` file.

If this option is given, the first element of **`sys.argv`** will be the script name as given on the command line.

If the script name refers directly to a Python file, the directory containing that file is added to the start of **`sys.path`**, and the file is executed as the **`__main__`** module.

If the script name refers to a directory or zipfile, the script name is added to the start of **`sys.path`** and the `__main__.py` file in that location is executed as the **`__main__`** module.

`-I` option can be used to run the script in isolated mode where **`sys.path`** contains neither the script's directory nor the user's site-packages directory. All `PYTHON*` environment variables are ignored, too.

Raises an auditing event `cpython.run_file` with argument `filename`.

See also

`runpy.run_path()`

Equivalent functionality directly available to Python code

If no interface option is given, `-i` is implied, `sys.argv[0]` is an empty string (`""`) and the current directory will be added to the start of `sys.path`. Also, tab-completion and history editing is automatically enabled, if available on your platform (see Readline configuration).

See also

Invoking the Interpreter

Changed in version 3.4: Automatic enabling of tab-completion and history editing.

1.1.2. Generic options

-?
-h
--help

Print a short description of all command line options and corresponding environment variables and exit.

--help-env

Print a short description of Python-specific environment variables and exit.

Added in version 3.11.

--help-xoptions

Print a description of implementation-specific `-X` options and exit.

Added in version 3.11.

--help-all

Print complete usage information and exit.

Added in version 3.11.

-V
--version

Print the Python version number and exit. Example output could be:

```
Python 3.8.0b2+
```

When given twice, print more information about the build, like:

```
Python 3.8.0b2+ (3.8:0c076caaa8, Apr 20 2019, 21:55:00)
[GCC 6.2.0 20161005]
```

Added in version 3.6: The `-VV` option.

1.1.3. Miscellaneous options

-b

Issue a warning when converting `bytes` or `bytearray` to `str` without specifying encoding or comparing **`bytes`** or **`bytearray`** with **`str`** or **`bytes`** with `int`. Issue an error when the option is given twice (**`-bb`**).

Changed in version 3.5: Affects also comparisons of `bytes` with `int`.

-B

If given, Python won't try to write `.pyc` files on the import of source modules. See also `PYTHONDONTWRITEBYTECODE`.

--check-hash-based-pycs default|always|never

Control the validation behavior of hash-based `.pyc` files. See Cached bytecode invalidation. When set to `default`, checked and unchecked hash-based bytecode cache files are validated according to their default semantics. When set to `always`, all hash-based `.pyc` files, whether checked or unchecked, are validated against their corresponding source file. When set to `never`, hash-based `.pyc` files are not validated against their corresponding source files.

The semantics of timestamp-based `.pyc` files are unaffected by this option.

-d

Turn on parser debugging output (for expert only). See also the **`PYTHONDEBUG`** environment variable.

This option requires a debug build of Python, otherwise it's ignored.

-E

Ignore all `PYTHON*` environment variables, e.g. **`PYTHONPATH`** and **`PYTHONHOME`**, that might be set.

See also the **`-P`** and **`-I`** (isolated) options.

-i

Enter interactive mode after execution.

Using the **`-i`** option will enter interactive mode in any of the following circumstances:

- When a script is passed as first argument
- When the **`-c`** option is used
- When the **`-m`** option is used

Interactive mode will start even when **`sys.stdin`** does not appear to be a terminal. The **`PYTHONSTARTUP`** file is not read.

This can be useful to inspect global variables or a stack trace when a script raises an exception. See also **`PYTHONINSPECT`**.

-I

Run Python in isolated mode. This also implies **`-E`**, **`-P`** and **`-s`** options.

In isolated mode **`sys.path`** contains neither the script's directory nor the user's site-packages directory. All `PYTHON*` environment variables are ignored, too. Further restrictions may be imposed to prevent the user from injecting malicious code.

Added in version 3.4.

-O

Remove assert statements and any code conditional on the value of **`__debug__`**. Augment the filename for compiled (bytecode) files by adding `.opt-1` before the `.pyc` extension (see **PEP 488** [https://peps.python.org/pep-0488/]). See also **`PYTHONOPTIMIZE`**.

Changed in version 3.5: Modify `.pyc` filenames according to **PEP 488** [https://peps.python.org/pep-0488/].

-OO

Do **`-O`** and also discard docstrings. Augment the filename for compiled (bytecode) files by adding `.opt-2` before the `.pyc` extension (see **PEP 488** [https://peps.python.org/pep-0488/]).

Changed in version 3.5: Modify `.pyc` filenames according to **PEP 488** [https://peps.python.org/pep-0488/].

-P

Don't prepend a potentially unsafe path to **`sys.path`**:

- `python -m module` command line: Don't prepend the current working directory.

- `python script.py` command line: Don’t prepend the script’s directory. If it’s a symbolic link, resolve symbolic links.
- `python -c code` and `python` (REPL) command lines: Don’t prepend an empty string, which means the current working directory.

See also the **`PYTHONSAFEPATH`** environment variable, and **`-E`** and **`-I`** (isolated) options.

Added in version 3.11.

-q

Don’t display the copyright and version messages even in interactive mode.

Added in version 3.2.

-R

Turn on hash randomization. This option only has an effect if the **`PYTHONHASHSEED`** environment variable is set to anything other than `random`, since hash randomization is enabled by default.

On previous versions of Python, this option turns on hash randomization, so that the `__hash__()` values of str and bytes objects are “salted” with an unpredictable random value. Although they remain constant within an individual Python process, they are not predictable between repeated invocations of Python.

Hash randomization is intended to provide protection against a denial-of-service caused by carefully chosen inputs that exploit the worst case performance of a dict construction, $O(n^2)$ complexity. See http://ocert.org/advisories/ocert-2011-003.html for details.

`PYTHONHASHSEED` allows you to set a fixed value for the hash seed secret.

Added in version 3.2.3.

Changed in version 3.7: The option is no longer ignored.

-s

Don’t add the `user site-packages directory` to `sys.path`.

See also `PYTHONNOUSERSITE`.

See also

PEP 370 [https://peps.python.org/pep-0370/] – Per user site-packages directory

-S

Disable the import of the module `site` and the site-dependent manipulations of `sys.path` that it entails. Also disable these manipulations if `site` is explicitly imported later (call `site.main()` if you want them to be triggered).

-u

Force the stdout and stderr streams to be unbuffered. This option has no effect on the stdin stream.

See also `PYTHONUNBUFFERED`.

Changed in version 3.7: The text layer of the stdout and stderr streams now is unbuffered.

-v

Print a message each time a module is initialized, showing the place (filename or built-in module) from which it is loaded. When given twice (**`-vv`**), print a message for each file that is checked for when searching for a module. Also provides information on module cleanup at exit.

Changed in version 3.10: The **`site`** module reports the site-specific paths and `.pth` files being processed.

See also **`PYTHONVERBOSE`**.

-W arg

Warning control. Python’s warning machinery by default prints warning messages to **`sys.stderr`**.

The simplest settings apply a particular action unconditionally to all warnings emitted by a process (even those that are otherwise ignored by default):

```
-Wdefault  # Warn once per call location
-Werror    # Convert to exceptions
-Walways   # Warn every time
-Wall      # Same as -Walways
-Wmodule   # Warn once per calling module
-Wonce     # Warn once per Python process
-Wignore   # Never warn
```

The action names can be abbreviated as desired and the interpreter will resolve them to the appropriate action name. For example, `-Wi` is the same as `-Wignore`.

The full form of argument is:

```
action:message:category:module:lineno
```

Empty fields match all values; trailing empty fields may be omitted. For example `-W ignore::DeprecationWarning` ignores all DeprecationWarning warnings.

The *action* field is as explained above but only applies to warnings that match the remaining fields.

The *message* field must match the whole warning message; this match is case-insensitive.

The *category* field matches the warning category (ex: `DeprecationWarning`). This must be a class name; the match test whether the actual warning category of the message is a subclass of the specified warning category.

The *module* field matches the (fully qualified) module name; this match is case-sensitive.

The *lineno* field matches the line number, where zero matches all line numbers and is thus equivalent to an omitted line number.

Multiple **`-W`** options can be given; when a warning matches more than one option, the action for the last matching option is performed. Invalid **`-W`** options are ignored (though, a warning message is printed about invalid options when the first warning is issued).

Warnings can also be controlled using the **`PYTHONWARNINGS`** environment variable and from within a Python program using the **`warnings`** module. For example, the **`warnings.filterwarnings()`** function can be used to use a regular expression on the warning message.

See The Warnings Filter and Describing Warning Filters for more details.

-x

Skip the first line of the source, allowing use of non-Unix forms of `#!cmd`. This is intended for a DOS specific hack only.

-X

Reserved for various implementation-specific options. CPython currently defines the following possible values:

- `-X faulthandler` to enable **`faulthandler`**. See also **`PYTHONFAULTHANDLER`**.

  Added in version 3.3.

- `-X showrefcount` to output the total reference count and number of used memory blocks when the program finishes or after each statement in the interactive interpreter. This only works on debug builds.

  Added in version 3.4.

- `-X tracemalloc` to start tracing Python memory allocations using the **`tracemalloc`** module. By default, only the most recent frame is stored in a traceback of a trace. Use `-X tracemalloc=NFRAME` to start tracing with a traceback limit of *NFRAME* frames. See **`tracemalloc.start()`** and **`PYTHONTRACEMALLOC`** for more information.

  Added in version 3.4.

- `-X int_max_str_digits` configures the integer string conversion length limitation. See also **`PYTHONINTMAXSTRDIGITS`**.

  Added in version 3.11.

- `-X importtime` to show how long each import takes. It shows module name, cumulative time (including nested imports) and self time (excluding nested imports). Note that its output may be broken in multi-threaded application. Typical usage is `python -X importtime -c 'import asyncio'`.

`-X importtime=2` enables additional output that indicates when an imported module has already been loaded. In such cases, the string `cached` will be printed in both time columns.

See also **`PYTHONPROFILEIMPORTTIME`**.

Added in version 3.7.

Changed in version 3.14: Added `-X importtime=2` to also trace imports of loaded modules, and reserved values other than `1` and `2` for future use.

- `-X dev`: enable Python Development Mode, introducing additional runtime checks that are too expensive to be enabled by default. See also **`PYTHONDEVMODE`**.

  Added in version 3.7.

- `-X utf8` enables the Python UTF-8 Mode. `-X utf8=0` explicitly disables Python UTF-8 Mode (even when it would otherwise activate automatically). See also **`PYTHONUTF8`**.

  Added in version 3.7.

- `-X pycache_prefix=PATH` enables writing `.pyc` files to a parallel tree rooted at the given directory instead of to the code tree. See also **`PYTHONPYCACHEPREFIX`**.

  Added in version 3.8.

- `-X warn_default_encoding` issues a **`EncodingWarning`** when the locale-specific default encoding is used for opening files. See also **`PYTHONWARNDEFAULTENCODING`**.

Added in version 3.10.

- `-X no_debug_ranges` disables the inclusion of the tables mapping extra location information (end line, start column offset and end column offset) to every instruction in code objects. This is useful when smaller code objects and pyc files are desired as well as suppressing the extra visual location indicators when the interpreter displays tracebacks. See also **`PYTHONNODEBUGRANGES`**.

  Added in version 3.11.

- `-X frozen_modules` determines whether or not frozen modules are ignored by the import machinery. A value of `on` means they get imported and `off` means they are ignored. The default is `on` if this is an installed Python (the normal case). If it's under development (running from the source tree) then the default is `off`. Note that the **`importlib_bootstrap`** and **`importlib_bootstrap_external`** frozen modules are always used, even if this flag is set to `off`. See also **`PYTHON_FROZEN_MODULES`**.

  Added in version 3.11.

- `-X perf` enables support for the Linux `perf` profiler. When this option is provided, the `perf` profiler will be able to report Python calls. This option is only available on some platforms and will do nothing if is not supported on the current system. The default value is "off". See also **`PYTHONPERFSUPPORT`** and Python support for the Linux perf profiler.

  Added in version 3.12.

- `-X perf_jit` enables support for the Linux `perf` profiler with DWARF support. When this option is provided, the `perf` profiler will be able to report Python calls using DWARF information. This option is only available on some platforms and will do nothing if is not supported on the current system. The default value is “off”. See also **`PYTHON_PERF_JIT_SUPPORT`** and Python support for the Linux perf profiler.

  Added in version 3.13.

- `-X disable_remote_debug` disables the remote debugging support as described in **PEP 768** [https://peps.python.org/pep-0768/]. This includes both the functionality to schedule code for execution in another process and the functionality to receive code for execution in the current process.

  This option is only available on some platforms and will do nothing if is not supported on the current system. See also **`PYTHON_DISABLE_REMOTE_DEBUG`** and **PEP 768** [https://peps.python.org/pep-0768/].

  Added in version 3.14.

- `-X cpu_count=n` overrides **`os.cpu_count()`**, **`os.process_cpu_count()`**, and **`multiprocessing.cpu_count()`**. *n* must be greater than or equal to 1. This option may be useful for users who need to limit CPU resources of a container system. See also **`PYTHON_CPU_COUNT`**. If *n* is `default`, nothing is overridden.

  Added in version 3.13.

- `-X presite=package.module` specifies a module that should be imported before the **`site`** module is executed and before the **`__main__`** module exists.

Therefore, the imported module isn't **`__main__`**. This can be used to execute code early during Python initialization. Python needs to be built in debug mode for this option to exist. See also **`PYTHON_PRESITE`**.

Added in version 3.13.

- `-X gil=0,1` forces the GIL to be disabled or enabled, respectively. Setting to `0` is only available in builds configured with **`--disable-gil`**. See also **`PYTHON_GIL`** and Free-threaded CPython.

  Added in version 3.13.

- `-X thread_inherit_context=0,1` causes **`Thread`** to, by default, use a copy of context of the caller of `Thread.start()` when starting. Otherwise, threads will start with an empty context. If unset, the value of this option defaults to `1` on free-threaded builds and to `0` otherwise. See also **`PYTHON_THREAD_INHERIT_CONTEXT`**.

  Added in version 3.14.

- `-X context_aware_warnings=0,1` causes the **`warnings.catch_warnings`** context manager to use a **`ContextVar`** to store warnings filter state. If unset, the value of this option defaults to `1` on free-threaded builds and to `0` otherwise. See also **`PYTHON_CONTEXT_AWARE_WARNINGS`**.

  Added in version 3.14.

- `-X tlbc=0,1` enables (1, the default) or disables (0) thread-local bytecode in builds configured with **`--disable-gil`**. When disabled, this also disables the specializing interpreter. See also **`PYTHON_TLBC`**.

  Added in version 3.14.

It also allows passing arbitrary values and retrieving them through the **`sys._xoptions`** dictionary.

Added in version 3.2.

Changed in version 3.9: Removed the `-X showalloccount` option.

Changed in version 3.10: Removed the `-X oldparser` option.

Removed in version 3.14: **`-J`** is no longer reserved for use by Jython [https://www.jython.org/], and now has no special meaning.

1.1.4. Controlling color

The Python interpreter is configured by default to use colors to highlight output in certain situations such as when displaying tracebacks. This behavior can be controlled by setting different environment variables.

Setting the environment variable `TERM` to `dumb` will disable color.

If the **`FORCE_COLOR`** [https://force-color.org/] environment variable is set, then color will be enabled regardless of the value of TERM. This is useful on CI systems which aren’t terminals but can still display ANSI escape sequences.

If the **`NO_COLOR`** [https://no-color.org/] environment variable is set, Python will disable all color in the output. This takes precedence over `FORCE_COLOR`.

All these environment variables are used also by other tools to control color output. To control the color output only in the Python interpreter, the **`PYTHON_COLORS`** environment variable can be used. This variable takes precedence over `NO_COLOR`, which in turn takes precedence over `FORCE_COLOR`.

1.2. Environment variables

These environment variables influence Python’s behavior, they are processed before the command-line switches other than -E or -I. It is customary that command-line switches override environmental variables where there is a conflict.

PYTHONHOME

Change the location of the standard Python libraries. By default, the libraries are searched in `prefix/lib/pythonversion` and `exec_prefix/lib/pythonversion`, where `prefix` and `exec_prefix` are installation-dependent directories, both defaulting to `/usr/local`.

When **`PYTHONHOME`** is set to a single directory, its value replaces both `prefix` and `exec_prefix`. To specify different values for these, set **`PYTHONHOME`** to `prefix:exec_prefix`.

PYTHONPATH

Augment the default search path for module files. The format is the same as the shell’s **`PATH`**: one or more directory pathnames separated by **`os.pathsep`** (e.g. colons on Unix or semicolons on Windows). Non-existent directories are silently ignored.

In addition to normal directories, individual **`PYTHONPATH`** entries may refer to zipfiles containing pure Python modules (in either source or compiled form). Extension modules cannot be imported from zipfiles.

The default search path is installation dependent, but generally begins with `prefix/lib/pythonversion` (see **`PYTHONHOME`** above). It is *always* appended to **`PYTHONPATH`**.

An additional directory will be inserted in the search path in front of `PYTHONPATH` as described above under Interface options. The search path can be manipulated from within a Python program as the variable `sys.path`.

PYTHONSAFEPATH

If this is set to a non-empty string, don't prepend a potentially unsafe path to `sys.path`: see the `-P` option for details.

Added in version 3.11.

PYTHONPLATLIBDIR

If this is set to a non-empty string, it overrides the `sys.platlibdir` value.

Added in version 3.9.

PYTHONSTARTUP

If this is the name of a readable file, the Python commands in that file are executed before the first prompt is displayed in interactive mode. The file is executed in the same namespace where interactive commands are executed so that objects defined or imported in it can be used without qualification in the interactive session. You can also change the prompts `sys.ps1` and `sys.ps2` and the hook `sys.__interactivehook__` in this file.

Raises an auditing event `cpython.run_startup` with the filename as the argument when called on startup.

PYTHONOPTIMIZE

If this is set to a non-empty string it is equivalent to specifying the `-O` option. If set to an integer, it is equivalent to specifying `-O` multiple times.

PYTHONBREAKPOINT

If this is set, it names a callable using dotted-path notation. The module containing the callable will be imported and then the callable will be run by the default implementation of `sys.breakpointhook()` which itself is called by built-in `breakpoint()`. If not set, or set to the empty string, it is equivalent to the value "pdb.set_trace". Setting this to the string "0" causes the default implementation of `sys.breakpointhook()` to do nothing but return immediately.

Added in version 3.7.

PYTHONDEBUG

If this is set to a non-empty string it is equivalent to specifying the `-d` option. If set to an integer, it is equivalent to specifying `-d` multiple times.

This environment variable requires a debug build of Python, otherwise it's ignored.

PYTHONINSPECT

If this is set to a non-empty string it is equivalent to specifying the `-i` option.

This variable can also be modified by Python code using `os.environ` to force inspect mode on program termination.

Raises an auditing event `cpython.run_stdin` with no arguments.

Changed in version 3.12.5: (also 3.11.10, 3.10.15, 3.9.20, and 3.8.20) Emits audit events.

Changed in version 3.13: Uses PyREPL if possible, in which case `PYTHONSTARTUP` is also executed. Emits audit events.

PYTHONUNBUFFERED

If this is set to a non-empty string it is equivalent to specifying the `-u` option.

PYTHONVERBOSE

If this is set to a non-empty string it is equivalent to specifying the `-v` option. If set to an integer, it is equivalent to specifying `-v` multiple times.

PYTHONCASEOK

If this is set, Python ignores case in `import` statements. This only works on Windows and macOS.

PYTHONDONTWRITEBYTECODE

If this is set to a non-empty string, Python won't try to write `.pyc` files on the import of source modules. This is equivalent to specifying the `-B` option.

PYTHONPYCACHEPREFIX

If this is set, Python will write `.pyc` files in a mirror directory tree at this path, instead of in `__pycache__` directories within the source tree. This is equivalent to specifying the `-X` `pycache_prefix=PATH` option.

Added in version 3.8.

PYTHONHASHSEED

If this variable is not set or set to `random`, a random value is used to seed the hashes of str and bytes objects.

If `PYTHONHASHSEED` is set to an integer value, it is used as a fixed seed for generating the hash() of the types covered by the hash randomization.

Its purpose is to allow repeatable hashing, such as for selftests for the interpreter itself, or to allow a cluster of python processes to share hash values.

The integer must be a decimal number in the range [0,4294967295]. Specifying the value 0 will disable hash randomization.

Added in version 3.2.3.

PYTHONINTMAXSTRDIGITS

If this variable is set to an integer, it is used to configure the interpreter's global integer string conversion length limitation.

Added in version 3.11.

PYTHONIOENCODING

If this is set before running the interpreter, it overrides the encoding used for stdin/stdout/stderr, in the syntax `encodingname:errorhandler`. Both the `encodingname` and the `:errorhandler` parts are optional and have the same meaning as in **`str.encode()`**.

For stderr, the `:errorhandler` part is ignored; the handler will always be `'backslashreplace'`.

Changed in version 3.4: The `encodingname` part is now optional.

Changed in version 3.6: On Windows, the encoding specified by this variable is ignored for interactive console buffers unless **`PYTHONLEGACYWINDOWSSTDIO`** is also specified. Files and pipes redirected through the standard streams are not affected.

PYTHONNOUSERSITE

If this is set, Python won't add the `user site-packages directory` to `sys.path`.

See also

PEP 370 [https://peps.python.org/pep-0370/] – Per user site-packages directory

PYTHONUSERBASE

Defines the `user base directory`, which is used to compute the path of the `user site-packages directory` and installation paths for `python -m pip install --user`.

See also

PEP 370 [https://peps.python.org/pep-0370/] – Per user site-packages directory

PYTHONEXECUTABLE

If this environment variable is set, `sys.argv[0]` will be set to its value instead of the value got through the C runtime. Only works on macOS.

PYTHONWARNINGS

This is equivalent to the `-W` option. If set to a comma separated string, it is equivalent to specifying `-W` multiple times, with filters later in the list taking precedence over those earlier in the list.

The simplest settings apply a particular action unconditionally to all warnings emitted by a process (even those that are otherwise ignored by default):

```
PYTHONWARNINGS=default  # Warn once per call location
PYTHONWARNINGS=error    # Convert to exceptions
PYTHONWARNINGS=always   # Warn every time
```

```
PYTHONWARNINGS=all      # Same as PYTHONWARNINGS=always
PYTHONWARNINGS=module   # Warn once per calling module
PYTHONWARNINGS=once     # Warn once per Python process
PYTHONWARNINGS=ignore   # Never warn
```

See The Warnings Filter and Describing Warning Filters for more details.

PYTHONFAULTHANDLER

If this environment variable is set to a non-empty string, **`faulthandler.enable()`** is called at startup: install a handler for **`SIGSEGV`**, **`SIGFPE`**, **`SIGABRT`**, **`SIGBUS`** and **`SIGILL`** signals to dump the Python traceback. This is equivalent to **`-X`** `faulthandler` option.

Added in version 3.3.

PYTHONTRACEMALLOC

If this environment variable is set to a non-empty string, start tracing Python memory allocations using the **`tracemalloc`** module. The value of the variable is the maximum number of frames stored in a traceback of a trace. For example, `PYTHONTRACEMALLOC=1` stores only the most recent frame. See the **`tracemalloc.start()`** function for more information. This is equivalent to setting the **`-X`** `tracemalloc` option.

Added in version 3.4.

PYTHONPROFILEIMPORTTIME

If this environment variable is set to `1`, Python will show how long each import takes. If set to `2`, Python will include output for imported modules that have already been loaded. This is equivalent to setting the **`-X`** `importtime` option.

Added in version 3.7.

Changed in version 3.14: Added `PYTHONPROFILEIMPORTTIME=2` to also trace imports of loaded modules.

PYTHONASYNCIODEBUG

If this environment variable is set to a non-empty string, enable the debug mode of the **`asyncio`** module.

Added in version 3.4.

PYTHONMALLOC

Set the Python memory allocators and/or install debug hooks.

Set the family of memory allocators used by Python:

- `default`: use the default memory allocators.
- `malloc`: use the **`malloc()`** function of the C library for all domains (**`PYMEM_DOMAIN_RAW`**, **`PYMEM_DOMAIN_MEM`**, **`PYMEM_DOMAIN_OBJ`**).
- `pymalloc`: use the pymalloc allocator for **`PYMEM_DOMAIN_MEM`** and **`PYMEM_DOMAIN_OBJ`** domains and use the **`malloc()`** function for the **`PYMEM_DOMAIN_RAW`** domain.
- `mimalloc`: use the mimalloc allocator for **`PYMEM_DOMAIN_MEM`** and **`PYMEM_DOMAIN_OBJ`** domains and use the **`malloc()`** function for the **`PYMEM_DOMAIN_RAW`** domain.

Install debug hooks:

- `debug`: install debug hooks on top of the default memory allocators.
- `malloc_debug`: same as `malloc` but also install debug hooks.
- `pymalloc_debug`: same as `pymalloc` but also install debug hooks.
- `mimalloc_debug`: same as `mimalloc` but also install debug hooks.

Added in version 3.6.

Changed in version 3.7: Added the `"default"` allocator.

PYTHONMALLOCSTATS

If set to a non-empty string, Python will print statistics of the pymalloc memory allocator every time a new pymalloc object arena is created, and on shutdown.

This variable is ignored if the **`PYTHONMALLOC`** environment variable is used to force the **`malloc()`** allocator of the C library, or if Python is configured without `pymalloc` support.

Changed in version 3.6: This variable can now also be used on Python compiled in release mode. It now has no effect if set to an empty string.

PYTHONLEGACYWINDOWSFSENCODING

If set to a non-empty string, the default filesystem encoding and error handler mode will revert to their pre-3.6 values of ‘mbcs’ and ‘replace’, respectively. Otherwise, the new defaults ‘utf-8’ and ‘surrogatepass’ are used.

This may also be enabled at runtime with **`sys._enablelegacywindowsfsencoding()`**.

Availability: Windows.

Added in version 3.6: See **PEP 529** [https://peps.python.org/pep-0529/] for more details.

PYTHONLEGACYWINDOWSSTDIO

If set to a non-empty string, does not use the new console reader and writer. This means that Unicode characters will be encoded according to the active console code page, rather than using utf-8.

This variable is ignored if the standard streams are redirected (to files or pipes) rather than referring to console buffers.

Availability: Windows.

Added in version 3.6.

PYTHONCOERCECLOCALE

If set to the value `0`, causes the main Python command line application to skip coercing the legacy ASCII-based C and POSIX locales to a more capable UTF-8 based alternative.

If this variable is *not* set (or is set to a value other than `0`), the `LC_ALL` locale override environment variable is also not set, and the current locale reported for the `LC_CTYPE` category is either the default `C` locale, or else the explicitly ASCII-based `POSIX` locale, then the Python CLI will attempt to configure the following locales for the `LC_CTYPE` category in the order listed before loading the interpreter runtime:

- `C.UTF-8`
- `C.utf8`
- `UTF-8`

If setting one of these locale categories succeeds, then the `LC_CTYPE` environment variable will also be set accordingly in the current process environment before the Python runtime is initialized. This ensures that in addition to being seen by both the interpreter itself and other locale-aware components running in the same process (such as the GNU `readline` library), the updated setting is also seen in subprocesses (regardless of whether or not those processes are running a Python interpreter), as well as in operations that query the environment rather than the current C locale (such as Python’s own **`locale.getdefaultlocale()`**).

Configuring one of these locales (either explicitly or via the above implicit locale coercion) automatically enables the `surrogateescape` error handler for **`sys.stdin`** and **`sys.stdout`** (**`sys.stderr`** continues to use `backslashreplace` as it does in any other locale). This stream handling behavior can be overridden using **`PYTHONIOENCODING`** as usual.

For debugging purposes, setting `PYTHONCOERCECLOCALE=warn` will cause Python to emit warning messages on `stderr` if either the locale coercion activates, or else if a locale that *would* have triggered coercion is still active when the Python runtime is initialized.

Also note that even when locale coercion is disabled, or when it fails to find a suitable target locale, **`PYTHONUTF8`** will still activate by default in legacy ASCII-based locales. Both features must be disabled in order to force the interpreter to use `ASCII` instead of `UTF-8` for system interfaces.

Availability: Unix.

Added in version 3.7: See **PEP 538** [https://peps.python.org/pep-0538/] for more details.

PYTHONDEVMODE

If this environment variable is set to a non-empty string, enable Python Development Mode, introducing additional runtime checks that are too expensive to be enabled by default. This is equivalent to setting the **`-X`** `dev` option.

Added in version 3.7.

PYTHONUTF8

If set to `1`, enable the Python UTF-8 Mode.

If set to `0`, disable the Python UTF-8 Mode.

Setting any other non-empty string causes an error during interpreter initialisation.

Added in version 3.7.

PYTHONWARNDEFAULTENCODING

If this environment variable is set to a non-empty string, issue a **`EncodingWarning`** when the locale-specific default encoding is used.

See Opt-in EncodingWarning for details.

Added in version 3.10.

PYTHONNODEBUGRANGES

If this variable is set, it disables the inclusion of the tables mapping extra location information (end line, start column offset and end column offset) to every instruction in code objects. This is useful when smaller code objects and pyc files are desired as well as suppressing the extra visual location indicators when the interpreter displays tracebacks.

Added in version 3.11.

PYTHONPERFSUPPORT

If this variable is set to a nonzero value, it enables support for the Linux `perf` profiler so Python calls can be detected by it.

If set to `0`, disable Linux `perf` profiler support.

See also the **`-X perf`** command-line option and Python support for the Linux perf profiler.

Added in version 3.12.

PYTHON_PERF_JIT_SUPPORT

If this variable is set to a nonzero value, it enables support for the Linux `perf` profiler so Python calls can be detected by it using DWARF information.

If set to `0`, disable Linux `perf` profiler support.

See also the **`-X perf_jit`** command-line option and Python support for the Linux perf profiler.

Added in version 3.13.

PYTHON_DISABLE_REMOTE_DEBUG

If this variable is set to a non-empty string, it disables the remote debugging feature described in **PEP 768** [https://peps.python.org/pep-0768/]. This includes both the functionality to schedule code for execution in another process and the functionality to receive code for execution in the current process.

See also the **`-X disable_remote_debug`** command-line option.

Added in version 3.14.

PYTHON_CPU_COUNT

If this variable is set to a positive integer, it overrides the return values of `os.cpu_count()` and `os.process_cpu_count()`.

See also the `-X cpu_count` command-line option.

Added in version 3.13.

PYTHON_FROZEN_MODULES

If this variable is set to `on` or `off`, it determines whether or not frozen modules are ignored by the import machinery. A value of `on` means they get imported and `off` means they are ignored. The default is `on` for non-debug builds (the normal case) and `off` for debug builds. Note that the **`importlib_bootstrap`** and **`importlib_bootstrap_external`** frozen modules are always used, even if this flag is set to `off`.

See also the `-X frozen_modules` command-line option.

Added in version 3.13.

PYTHON_COLORS

If this variable is set to `1`, the interpreter will colorize various kinds of output. Setting it to `0` deactivates this behavior. See also Controlling color.

Added in version 3.13.

PYTHON_BASIC_REPL

If this variable is set to any value, the interpreter will not attempt to load the Python-based REPL that requires `readline`, and will instead use the traditional parser-based REPL.

Added in version 3.13.

PYTHON_HISTORY

This environment variable can be used to set the location of a `.python_history` file (by default, it is `.python_history` in the user's home directory).

Added in version 3.13.

PYTHON_GIL

If this variable is set to `1`, the global interpreter lock (GIL) will be forced on. Setting it to `0` forces the GIL off (needs Python configured with the **`--disable-gil`** build option).

See also the **`-X gil`** command-line option, which takes precedence over this variable, and Free-threaded CPython.

Added in version 3.13.

PYTHON_THREAD_INHERIT_CONTEXT

If this variable is set to `1` then **`Thread`** will, by default, use a copy of context of the caller of `Thread.start()` when starting. Otherwise, new threads will start with an empty context. If unset, this variable defaults to `1` on free-threaded builds and to `0` otherwise. See also **`-X thread_inherit_context`**.

Added in version 3.14.

PYTHON_CONTEXT_AWARE_WARNINGS

If set to `1` then the **`warnings.catch_warnings`** context manager will use a **`ContextVar`** to store warnings filter state. If unset, this variable defaults to `1` on free-threaded builds and to `0` otherwise. See **`-X context_aware_warnings`**.

Added in version 3.14.

PYTHON_JIT

On builds where experimental just-in-time compilation is available, this variable can force the JIT to be disabled (`0`) or enabled (`1`) at interpreter startup.

Added in version 3.13.

PYTHON_TLBC

If set to `1` enables thread-local bytecode. If set to `0` thread-local bytecode and the specializing interpreter are disabled. Only applies to builds configured with **`--disable-gil`**.

See also the **`-X tlbc`** command-line option.

Added in version 3.14.

1.2.1. Debug-mode variables

PYTHONDUMPREFS

If set, Python will dump objects and reference counts still alive after shutting down the interpreter.

Needs Python configured with the **`--with-trace-refs`** build option.

PYTHONDUMPREFSFILE

If set, Python will dump objects and reference counts still alive after shutting down the interpreter into a file under the path given as the value to this environment variable.

Needs Python configured with the **`--with-trace-refs`** build option.

Added in version 3.11.

PYTHON_PRESITE

If this variable is set to a module, that module will be imported early in the interpreter lifecycle, before the `site` module is executed, and before the `__main__` module is created. Therefore, the imported module is not treated as `__main__`.

This can be used to execute code early during Python initialization.

To import a submodule, use `package.module` as the value, like in an import statement.

See also the `-X presite` command-line option, which takes precedence over this variable.

Needs Python configured with the `--with-pydebug` build option.

Added in version 3.13.

2. Using Python on Unix platforms

2.1. Getting and installing the latest version of Python

2.1.1. On Linux

Python comes preinstalled on most Linux distributions, and is available as a package on all others. However there are certain features you might want to use that are not available on your distro’s package. You can compile the latest version of Python from source.

In the event that the latest version of Python doesn’t come preinstalled and isn’t in the repositories as well, you can make packages for your own distro. Have a look at the following links:

See also

https://www.debian.org/doc/manuals/maint-guide/first.en.html
for Debian users

https://en.opensuse.org/Portal:Packaging
for OpenSuse users

https://docs.fedoraproject.org/en-US/package-maintainers/Packaging_Tutorial_GNU_Hello/
for Fedora users

https://slackbook.org/html/package-management-making-packages.html
for Slackware users

2.1.1.1. Installing IDLE

In some cases, IDLE might not be included in your Python installation.

- For Debian and Ubuntu users:

  ```
  sudo apt update
  sudo apt install idle
  ```

- For Fedora, RHEL, and CentOS users:

  ```
  sudo dnf install python3-idle
  ```

- For SUSE and OpenSUSE users:

  ```
  sudo zypper install python3-idle
  ```

- For Alpine Linux users:

  ```
  sudo apk add python3-idle
  ```

2.1.2. On FreeBSD and OpenBSD

- FreeBSD users, to add the package use:

  ```
  pkg install python3
  ```

- OpenBSD users, to add the package use:

  ```
  pkg_add -r python

  pkg_add
  ftp://ftp.openbsd.org/pub/OpenBSD/4.2/packages/<insert your
  architecture here>/python-<version>.tgz
  ```

For example i386 users get the 2.5.1 version of Python using:

```
pkg_add
ftp://ftp.openbsd.org/pub/OpenBSD/4.2/packages/i386/python-
2.5.1p2.tgz
```

2.2. Building Python

See also

If you want to contribute to CPython, refer to the devguide [https://devguide.python.org/getting-started/setup-building/], which includes build instructions and other tips on setting up environment.

If you want to compile CPython yourself, first thing you should do is get the source [https://www.python.org/downloads/source/]. You can download either the latest release's source or grab a fresh clone [https://devguide.python.org/setup/#get-the-source-code]. You will also need to install the build requirements.

The build process consists of the usual commands:

```
./configure
make
make install
```

Configuration options and caveats for specific Unix platforms are extensively documented in the README.rst [https://github.com/python/cpython/tree/3.14/README.rst] file in the root of the Python source tree.

Warning

`make install` can overwrite or masquerade the `python3` binary. `make altinstall` is therefore recommended instead of `make install` since it only installs `exec_prefix/bin/pythonversion`.

2.3. Python-related paths and files

These are subject to difference depending on local installation conventions; `prefix` and `exec_prefix` are installation-dependent and should be interpreted as for GNU software; they may be the same.

For example, on most Linux systems, the default for both is `/usr`.

File/directory	Meaning
`exec_prefix/bin/python3`	Recommended location of the interpreter.
`prefix/lib/pythonversion`, `exec_prefix/lib/pythonversion`	Recommended locations of the directories containing the standard modules.
`prefix/include/pythonversion`, `exec_prefix/include/pythonversion`	Recommended locations of the directories containing the include files needed for developing Python extensions and

File/directory	Meaning
	embedding the interpreter.

2.4. Miscellaneous

To easily use Python scripts on Unix, you need to make them executable, e.g. with

```
$ chmod +x script
```

and put an appropriate Shebang line at the top of the script. A good choice is usually

```
#!/usr/bin/env python3
```

which searches for the Python interpreter in the whole **`PATH`**. However, some Unices may not have the **env** command, so you may need to hardcode `/usr/bin/python3` as the interpreter path.

To use shell commands in your Python scripts, look at the `subprocess` module.

2.5. Custom OpenSSL

1. To use your vendor's OpenSSL configuration and system trust store, locate the directory with `openssl.cnf` file or symlink in `/etc`. On most distribution the file is either in `/etc/ssl` or `/etc/pki/tls`. The directory should also contain a `cert.pem` file and/or a `certs` directory.

   ```
   $ find /etc/ -name openssl.cnf -printf "%h\n"
   /etc/ssl
   ```

2. Download, build, and install OpenSSL. Make sure you use `install_sw` and not `install`. The `install_sw` target does not override `openssl.cnf`.

   ```
   $ curl -O https://www.openssl.org/source/openssl-
   VERSION.tar.gz
   $ tar xzf openssl-VERSION
   $ pushd openssl-VERSION
   $ ./config \
       --prefix=/usr/local/custom-openssl \
       --libdir=lib \
       --openssldir=/etc/ssl
   $ make -j1 depend
   $ make -j8
   $ make install_sw
   $ popd
   ```

3. Build Python with custom OpenSSL (see the configure `--with-openssl` and `--with-openssl-rpath` options)

   ```
   $ pushd python-3.x.x
   $ ./configure -C \
       --with-openssl=/usr/local/custom-openssl \
       --with-openssl-rpath=auto \
       --prefix=/usr/local/python-3.x.x
   $ make -j8
   $ make altinstall
   ```

Note

Patch releases of OpenSSL have a backwards compatible ABI. You don’t need to recompile Python to update OpenSSL. It’s sufficient to replace the custom OpenSSL installation with a newer version.

3. Configure Python

3.1. Build Requirements

To build CPython, you will need:

- A C11 [https://en.cppreference.com/w/c/11] compiler. Optional C11 features [https://en.wikipedia.org/wiki/C11_(C_standard_revision)#Optional_features] are not required.
- On Windows, Microsoft Visual Studio 2017 or later is required.
- Support for IEEE 754 [https://en.wikipedia.org/wiki/IEEE_754] floating-point numbers and floating-point Not-a-Number (NaN) [https://en.wikipedia.org/wiki/NaN#Floating_point].
- Support for threads.

Changed in version 3.5: On Windows, Visual Studio 2015 or later is now required.

Changed in version 3.6: Selected C99 features, like `<stdint.h>` and `static inline` functions, are now required.

Changed in version 3.7: Thread support is now required.

Changed in version 3.11: C11 compiler, IEEE 754 and NaN support are now required. On Windows, Visual Studio 2017 or later is required.

See also **PEP 7** [https://peps.python.org/pep-0007/] “Style Guide for C Code” and **PEP 11** [https://peps.python.org/pep-0011/] “CPython platform support”.

3.1.1. Requirements for optional modules

Some optional modules of the standard library require third-party libraries installed for development (for example, header files must be available).

Missing requirements are reported in the `configure` output. Modules that are missing due to missing dependencies are listed near the end of the `make` output, sometimes using an internal name, for example, `_ctypes` for **`ctypes`** module.

If you distribute a CPython interpreter without optional modules, it’s best practice to advise users, who generally expect that standard library modules are available.

Dependencies to build optional modules are:

Dependency	**Minimum version**	**Python module**
libbz2		**`bz2`**
libffi	3.3.0 recommended	**`ctypes`**
liblzma		**`lzma`**
libmpdec	2.5.0	**`decimal`** [[1]]
libreadline or libedit [[2]]		**`readline`**
libuuid		`_uuid` [[3]]

Dependency	**Minimum version**	**Python module**
ncurses [[4]]		`curses`
OpenSSL	3.0.18 recommended (1.1.1 minimum)	`ssl`, `hashlib` [[5]]
SQLite	3.15.2	`sqlite3`
Tcl/Tk	8.5.12	`tkinter`, IDLE, `turtle`
zlib	1.2.2.1	`zlib`, `gzip`, `ensurepip`
zstd	1.4.5	`compression.zstd`

[1]

If *libmpdec* is not available, the `decimal` module will use a pure-Python implementation. See `--with-system-libmpdec` for details.

[2]

See `--with-readline` for choosing the backend for the `readline` module.

[3]

The **`uuid`** module uses `_uuid` to generate “safe” UUIDs. See the module documentation for details.

[4]

The **`curses`** module requires the `libncurses` or `libncursesw` library. The **`curses.panel`** module additionally requires the `libpanel` or `libpanelw` library.

[5]

If OpenSSL is not available, the **`hashlib`** module will use bundled implementations of several hash functions. See **`--with-builtin-hashlib-hashes`** for *forcing* usage of OpenSSL.

Note that the table does not include all optional modules; in particular, platform-specific modules like **`winreg`** are not listed here.

See also

- The devguide [https://devguide.python.org/getting-started/setup-building/#install-dependencies] includes a full list of dependencies required to build all modules and instructions on how to install them on common platforms.
- **`--with-system-expat`** allows building with an external libexpat [https://libexpat.github.io/] library.
- Options for third-party dependencies

Changed in version 3.1: Tcl/Tk version 8.3.1 is now required for **`tkinter`**.

Changed in version 3.5: Tcl/Tk version 8.4 is now required for **`tkinter`**.

Changed in version 3.7: OpenSSL 1.0.2 is now required for **`hashlib`** and **`ssl`**.

Changed in version 3.10: OpenSSL 1.1.1 is now required for **`hashlib`** and **`ssl`**. SQLite 3.7.15 is now required for **`sqlite3`**.

Changed in version 3.11: Tcl/Tk version 8.5.12 is now required for **`tkinter`**.

Changed in version 3.13: SQLite 3.15.2 is now required for **`sqlite3`**.

3.2. Generated files

To reduce build dependencies, Python source code contains multiple generated files. Commands to regenerate all generated files:

```
make regen-all
make regen-stdlib-module-names
make regen-limited-abi
make regen-configure
```

The `Makefile.pre.in` file documents generated files, their inputs, and tools used to regenerate them. Search for `regen-*` make targets.

3.2.1. configure script

The `make regen-configure` command regenerates the `aclocal.m4` file and the `configure` script using the `Tools/build/regen-configure.sh` shell script which uses an Ubuntu container to get the same tools versions and have a reproducible output.

The container is optional, the following command can be run locally:

```
autoreconf -ivf -Werror
```

The generated files can change depending on the exact versions of the tools used. The container that CPython uses has Autoconf [https://gnu.org/software/autoconf] 2.72, `aclocal` from Automake [https://www.gnu.org/software/automake] 1.16.5, and pkg-config [https://www.freedesktop.org/wiki/Software/pkg-config/] 1.8.1.

Changed in version 3.13: Autoconf 2.71 and aclocal 1.16.5 and are now used to regenerate `configure`.

Changed in version 3.14: Autoconf 2.72 is now used to regenerate `configure`.

3.3. Configure Options

List all `configure` script options using:

```
./configure --help
```

See also the `Misc/SpecialBuilds.txt` in the Python source distribution.

3.3.1. General Options

--enable-loadable-sqlite-extensions

Support loadable extensions in the **`_sqlite`** extension module (default is no) of the **`sqlite3`** module.

See the **`sqlite3.Connection.enable_load_extension()`** method of the **`sqlite3`** module.

Added in version 3.6.

--disable-ipv6

Disable IPv6 support (enabled by default if supported), see the **`socket`** module.

--enable-big-digits=[15|30]

Define the size in bits of Python **`int`** digits: 15 or 30 bits.

By default, the digit size is 30.

Define the `PYLONG_BITS_IN_DIGIT` to `15` or `30`.

See **`sys.int_info.bits_per_digit`**.

--with-suffix=SUFFIX

Set the Python executable suffix to *SUFFIX*.

The default suffix is `.exe` on Windows and macOS (`python.exe` executable), `.js` on Emscripten node, `.html` on Emscripten browser, `.wasm` on WASI, and an empty string on other platforms (`python` executable).

Changed in version 3.11: The default suffix on WASM platform is one of `.js`, `.html` or `.wasm`.

--with-tzpath=<list of absolute paths separated by pathsep>

Select the default time zone search path for **`zoneinfo.TZPATH`**. See the Compile-time configuration of the **`zoneinfo`** module.

Default: `/usr/share/zoneinfo:/usr/lib/zoneinfo:/usr/share/lib/zoneinfo:/etc/zoneinfo`.

See **`os.pathsep`** path separator.

Added in version 3.9.

--without-decimal-contextvar

Build the `_decimal` extension module using a thread-local context rather than a coroutine-local context (default), see the **`decimal`** module.

See **`decimal.HAVE_CONTEXTVAR`** and the **`contextvars`** module.

Added in version 3.9.

--with-dbmliborder=<list of backend names>

Override order to check db backends for the **`dbm`** module

A valid value is a colon (`:`) separated string with the backend names:

- `ndbm`;
- `gdbm`;
- `bdb`.

--without-c-locale-coercion

Disable C locale coercion to a UTF-8 based locale (enabled by default).

Don’t define the `PY_COERCE_C_LOCALE` macro.

See **`PYTHONCOERCECLOCALE`** and the **PEP 538** [https://peps.python.org/pep-0538/].

--with-platlibdir=DIRNAME

Python library directory name (default is `lib`).

Fedora and SuSE use `lib64` on 64-bit platforms.

See **`sys.platlibdir`**.

Added in version 3.9.

--with-wheel-pkg-dir=PATH

Directory of wheel packages used by the `ensurepip` module (none by default).

Some Linux distribution packaging policies recommend against bundling dependencies. For example, Fedora installs wheel packages in the `/usr/share/python-wheels/` directory and don’t install the **`ensurepip._bundled`** package.

Added in version 3.10.

--with-pkg-config=[check|yes|no]

Whether configure should use **pkg-config** to detect build dependencies.

- `check` (default): **pkg-config** is optional
- `yes`: **pkg-config** is mandatory
- `no`: configure does not use **pkg-config** even when present

Added in version 3.11.

--enable-pystats

Turn on internal Python performance statistics gathering.

By default, statistics gathering is off. Use `python3 -X pystats` command or set `PYTHONSTATS=1` environment variable to turn on statistics gathering at Python startup.

At Python exit, dump statistics if statistics gathering was on and not cleared.

Effects:

- Add `-X pystats` command line option.
- Add **`PYTHONSTATS`** environment variable.
- Define the `Py_STATS` macro.

- Add functions to the `sys` module:
    - **`sys._stats_on()`**: Turns on statistics gathering.
    - **`sys._stats_off()`**: Turns off statistics gathering.
    - **`sys._stats_clear()`**: Clears the statistics.
    - **`sys._stats_dump()`**: Dump statistics to file, and clears the statistics.

The statistics will be dumped to a arbitrary (probably unique) file in `/tmp/py_stats/` (Unix) or `C:\temp\py_stats\` (Windows). If that directory does not exist, results will be printed on stderr.

Use `Tools/scripts/summarize_stats.py` to read the stats.

Statistics:

- Opcode:
    - Specialization: success, failure, hit, deferred, miss, deopt, failures;
    - Execution count;
    - Pair count.
- Call:
    - Inlined Python calls;
    - PyEval calls;
    - Frames pushed;
    - Frame object created;
    - Eval calls: vector, generator, legacy, function VECTORCALL, build class, slot, function “ex”, API, method.
- Object:
    - incref and decref;
    - interpreter incref and decref;
    - allocations: all, 512 bytes, 4 kiB, big;
    - free;
    - to/from free lists;
    - dictionary materialized/dematerialized;
    - type cache;

    - optimization attempts;
    - optimization traces created/executed;
    - uops executed.
- Garbage collector:
    - Garbage collections;
    - Objects visited;
    - Objects collected.

Added in version 3.11.

--disable-gil

Enables support for running Python without the global interpreter lock (GIL): free threading build.

Defines the `Py_GIL_DISABLED` macro and adds `"t"` to **`sys.abiflags`**.

See Free-threaded CPython for more detail.

Added in version 3.13.

--enable-experimental-jit=[no|yes|yes-off|interpreter]

Indicate how to integrate the experimental just-in-time compiler.

- `no`: Don’t build the JIT.
- `yes`: Enable the JIT. To disable it at runtime, set the environment variable **`PYTHON_JIT=0`**.
- `yes-off`: Build the JIT, but disable it by default. To enable it at runtime, set the environment variable **`PYTHON_JIT=1`**.
- `interpreter`: Enable the “JIT interpreter” (only useful for those debugging the JIT itself). To disable it at runtime, set the environment variable **`PYTHON_JIT=0`**.

`--enable-experimental-jit=no` is the default behavior if the option is not provided, and `--enable-experimental-jit` is shorthand for `--enable-experimental-jit=yes`. See `Tools/jit/README.md` for more information, including how to install the necessary build-time dependencies.

Note

When building CPython with JIT enabled, ensure that your system has Python 3.11 or later installed.

Added in version 3.13.

PKG_CONFIG

Path to `pkg-config` utility.

PKG_CONFIG_LIBDIR

PKG_CONFIG_PATH

`pkg-config` options.

3.3.2. C compiler options

CC

C compiler command.

CFLAGS

C compiler flags.

CPP

C preprocessor command.

CPPFLAGS

C preprocessor flags, e.g. `-Iinclude_dir`.

3.3.3. Linker options

LDFLAGS

Linker flags, e.g. `-L`*`library_directory`*.

LIBS

Libraries to pass to the linker, e.g. `-l`*`library`*.

MACHDEP

Name for machine-dependent library files.

3.3.4. Options for third-party dependencies

Added in version 3.11.

BZIP2_CFLAGS

BZIP2_LIBS

C compiler and linker flags to link Python to `libbz2`, used by **`bz2`** module, overriding `pkg-config`.

CURSES_CFLAGS

CURSES_LIBS

C compiler and linker flags for `libncurses` or `libncursesw`, used by **`curses`** module, overriding `pkg-config`.

GDBM_CFLAGS

GDBM_LIBS

C compiler and linker flags for `gdbm`.

LIBEDIT_CFLAGS

LIBEDIT_LIBS

C compiler and linker flags for `libedit`, used by **`readline`** module, overriding `pkg-config`.

LIBFFI_CFLAGS

LIBFFI_LIBS

C compiler and linker flags for `libffi`, used by **`ctypes`** module, overriding `pkg-config`.

LIBMPDEC_CFLAGS

LIBMPDEC_LIBS

C compiler and linker flags for `libmpdec`, used by **`decimal`** module, overriding `pkg-config`.

Note

These environment variables have no effect unless **`--with-system-libmpdec`** is specified.

LIBLZMA_CFLAGS

LIBLZMA_LIBS

C compiler and linker flags for `liblzma`, used by **`lzma`** module, overriding `pkg-config`.

LIBREADLINE_CFLAGS

LIBREADLINE_LIBS

C compiler and linker flags for `libreadline`, used by **`readline`** module, overriding `pkg-config`.

LIBSQLITE3_CFLAGS

LIBSQLITE3_LIBS

C compiler and linker flags for `libsqlite3`, used by **`sqlite3`** module, overriding `pkg-config`.

LIBUUID_CFLAGS

LIBUUID_LIBS

C compiler and linker flags for `libuuid`, used by **`uuid`** module, overriding `pkg-config`.

LIBZSTD_CFLAGS

LIBZSTD_LIBS

C compiler and linker flags for `libzstd`, used by **`compression.zstd`** module, overriding `pkg-config`.

Added in version 3.14.

PANEL_CFLAGS

PANEL_LIBS

C compiler and linker flags for PANEL, overriding `pkg-config`.

C compiler and linker flags for `libpanel` or `libpanelw`, used by **`curses.panel`** module, overriding `pkg-config`.

TCLTK_CFLAGS

TCLTK_LIBS

C compiler and linker flags for TCLTK, overriding `pkg-config`.

ZLIB_CFLAGS

ZLIB_LIBS

C compiler and linker flags for `libzlib`, used by **`gzip`** module, overriding `pkg-config`.

3.3.5. WebAssembly Options

--enable-wasm-dynamic-linking

Turn on dynamic linking support for WASM.

Dynamic linking enables `dlopen`. File size of the executable increases due to limited dead code elimination and additional features.

Added in version 3.11.

--enable-wasm-pthreads

Turn on pthreads support for WASM.

Added in version 3.11.

3.3.6. Install Options

--prefix=PREFIX

Install architecture-independent files in PREFIX. On Unix, it defaults to `/usr/local`.

This value can be retrieved at runtime using **`sys.prefix`**.

As an example, one can use `--prefix="$HOME/.local/"` to install a Python in its home directory.

--exec-prefix=EPREFIX

Install architecture-dependent files in EPREFIX, defaults to **`--prefix`**.

This value can be retrieved at runtime using **`sys.exec_prefix`**.

--disable-test-modules

Don’t build nor install test modules, like the **`test`** package or the **`_testcapi`** extension module (built and installed by default).

Added in version 3.10.

--with-ensurepip=[upgrade|install|no]

Select the `ensurepip` command run on Python installation:

- `upgrade` (default): run `python -m ensurepip --altinstall --upgrade` command.
- `install`: run `python -m ensurepip --altinstall` command;
- `no`: don't run ensurepip;

Added in version 3.6.

3.3.7. Performance options

Configuring Python using `--enable-optimizations --with-lto` (PGO + LTO) is recommended for best performance. The experimental `--enable-bolt` flag can also be used to improve performance.

--enable-optimizations

Enable Profile Guided Optimization (PGO) using `PROFILE_TASK` (disabled by default).

The C compiler Clang requires `llvm-profdata` program for PGO. On macOS, GCC also requires it: GCC is just an alias to Clang on macOS.

Disable also semantic interposition in libpython if `--enable-shared` and GCC is used: add `-fno-semantic-interposition` to the compiler and linker flags.

Note

During the build, you may encounter compiler warnings about profile data not being available for some source files. These warnings are harmless, as only a subset of the code is exercised during profile data acquisition. To disable these warnings on Clang, manually suppress them by adding `-Wno-profile-instr-unprofiled` to **`CFLAGS`**.

Added in version 3.6.

Changed in version 3.10: Use `-fno-semantic-interposition` on GCC.

PROFILE_TASK

Environment variable used in the Makefile: Python command line arguments for the PGO generation task.

Default: `-m test --pgo --timeout=$(TESTTIMEOUT)`.

Added in version 3.8.

Changed in version 3.13: Task failure is no longer ignored silently.

--with-lto=[full|thin|no|yes]

Enable Link Time Optimization (LTO) in any build (disabled by default).

The C compiler Clang requires `llvm-ar` for LTO (`ar` on macOS), as well as an LTO-aware linker (`ld.gold` or `lld`).

Added in version 3.6.

Added in version 3.11: To use ThinLTO feature, use `--with-lto=thin` on Clang.

Changed in version 3.12: Use ThinLTO as the default optimization policy on Clang if the compiler accepts the flag.

--enable-bolt

Enable usage of the BOLT post-link binary optimizer [https://github.com/llvm/llvm-project/tree/main/bolt] (disabled by default).

BOLT is part of the LLVM project but is not always included in their binary distributions. This flag requires that `llvm-bolt` and `merge-fdata` are available.

BOLT is still a fairly new project so this flag should be considered experimental for now. Because this tool operates on machine code its success is dependent on a combination of the build environment + the other optimization configure args + the CPU architecture, and not all combinations are supported. BOLT versions before LLVM 16 are known to crash BOLT under some scenarios. Use of LLVM 16 or newer for BOLT optimization is strongly encouraged.

The **`BOLT_INSTRUMENT_FLAGS`** and **`BOLT_APPLY_FLAGS`** **configure** variables can be defined to override the default set of arguments for **llvm-bolt** to instrument and apply BOLT data to binaries, respectively.

Added in version 3.12.

BOLT_APPLY_FLAGS

Arguments to `llvm-bolt` when creating a BOLT optimized binary [https://github.com/facebookarchive/BOLT].

Added in version 3.12.

BOLT_INSTRUMENT_FLAGS

Arguments to `llvm-bolt` when instrumenting binaries.

Added in version 3.12.

--with-computed-gotos

Enable computed gotos in evaluation loop (enabled by default on supported compilers).

--with-tail-call-interp

Enable interpreters using tail calls in CPython. If enabled, enabling PGO (**`--enable-optimizations`**) is highly recommended. This option specifically requires a C compiler with proper tail call support, and the preserve_none [https://clang.llvm.org/docs/AttributeReference.html#preserve-none] calling convention. For example, Clang 19 and newer supports this feature.

Added in version 3.14.

--without-mimalloc

Disable the fast mimalloc allocator (enabled by default).

See also **`PYTHONMALLOC`** environment variable.

--without-pymalloc

Disable the specialized Python memory allocator pymalloc (enabled by default).

See also **`PYTHONMALLOC`** environment variable.

--without-doc-strings

Disable static documentation strings to reduce the memory footprint (enabled by default). Documentation strings defined in Python are not affected.

Don’t define the `WITH_DOC_STRINGS` macro.

See the `PyDoc_STRVAR()` macro.

--enable-profiling

Enable C-level code profiling with `gprof` (disabled by default).

--with-strict-overflow

Add `-fstrict-overflow` to the C compiler flags (by default we add `-fno-strict-overflow` instead).

--without-remote-debug

Deactivate remote debugging support described in **PEP 768** [https://peps.python.org/pep-0768/] (enabled by default). When this flag is provided the code that allows the interpreter to schedule the execution of a Python file in a separate process as described in **PEP 768** [https://peps.python.org/pep-0768/] is not compiled. This includes both the functionality to schedule code to be executed and the functionality to receive code to be executed.

Py_REMOTE_DEBUG

This macro is defined by default, unless Python is configured with `--without-remote-debug`.

Note that even if the macro is defined, remote debugging may not be available (for example, on an incompatible platform).

Added in version 3.14.

3.3.8. Python Debug Build

A debug build is Python built with the `--with-pydebug` configure option.

Effects of a debug build:

- Display all warnings by default: the list of default warning filters is empty in the `warnings` module.

- Add `d` to `sys.abiflags`.
- Add **`sys.gettotalrefcount()`** function.
- Add `-X showrefcount` command line option.
- Add `-d` command line option and `PYTHONDEBUG` environment variable to debug the parser.
- Add support for the `__lltrace__` variable: enable low-level tracing in the bytecode evaluation loop if the variable is defined.
- Install debug hooks on memory allocators to detect buffer overflow and other memory errors.
- Define `Py_DEBUG` and `Py_REF_DEBUG` macros.
- Add runtime checks: code surrounded by `#ifdef Py_DEBUG` and `#endif`. Enable `assert(...)` and `_PyObject_ASSERT(...)` assertions: don't set the `NDEBUG` macro (see also the `--with-assertions` configure option). Main runtime checks:
  - Add sanity checks on the function arguments.
  - Unicode and int objects are created with their memory filled with a pattern to detect usage of uninitialized objects.
  - Ensure that functions which can clear or replace the current exception are not called with an exception raised.
  - Check that deallocator functions don't change the current exception.
  - The garbage collector (`gc.collect()` function) runs some basic checks on objects consistency.
  - The **`Py_SAFE_DOWNCAST()`** macro checks for integer underflow and overflow when downcasting from wide types to narrow types.

See also the Python Development Mode and the `--with-trace-refs` configure option.

Changed in version 3.8: Release builds and debug builds are now ABI compatible: defining the `Py_DEBUG` macro no longer implies the `Py_TRACE_REFS` macro (see the `--with-trace-refs` option).

3.3.9. Debug options

--with-pydebug

Build Python in debug mode: define the `Py_DEBUG` macro (disabled by default).

--with-trace-refs

Enable tracing references for debugging purpose (disabled by default).

Effects:

- Define the `Py_TRACE_REFS` macro.
- Add `sys.getobjects()` function.
- Add `PYTHONDUMPREFS` environment variable.

The `PYTHONDUMPREFS` environment variable can be used to dump objects and reference counts still alive at Python exit.

Statically allocated objects are not traced.

Added in version 3.8.

Changed in version 3.13: This build is now ABI compatible with release build and debug build.

--with-assertions

Build with C assertions enabled (default is no): `assert(...);` and `_PyObject_ASSERT(...);`.

If set, the `NDEBUG` macro is not defined in the `OPT` compiler variable.

See also the **`--with-pydebug`** option (debug build) which also enables assertions.

Added in version 3.6.

--with-valgrind

Enable Valgrind support (default is no).

--with-dtrace

Enable DTrace support (default is no).

See Instrumenting CPython with DTrace and SystemTap.

Added in version 3.6.

--with-address-sanitizer

Enable AddressSanitizer memory error detector, `asan` (default is no). To improve ASan detection capabilities you may also want to combine this with **`--without-pymalloc`** to disable the specialized small-object allocator whose allocations are not tracked by ASan.

Added in version 3.6.

--with-memory-sanitizer

Enable MemorySanitizer allocation error detector, `msan` (default is no).

Added in version 3.6.

--with-undefined-behavior-sanitizer

Enable UndefinedBehaviorSanitizer undefined behaviour detector, `ubsan` (default is no).

Added in version 3.6.

--with-thread-sanitizer

Enable ThreadSanitizer data race detector, `tsan` (default is no).

Added in version 3.13.

3.3.10. Linker options

--enable-shared

Enable building a shared Python library: `libpython` (default is no).

--without-static-libpython

Do not build `libpythonMAJOR.MINOR.a` and do not install `python.o` (built and enabled by default).

Added in version 3.10.

3.3.11. Libraries options

--with-libs='lib1 ...'

Link against additional libraries (default is no).

--with-system-expat

Build the **`pyexpat`** module using an installed `expat` library (default is no).

--with-system-libmpdec

Build the `_decimal` extension module using an installed `mpdecimal` library, see the **`decimal`** module (default is yes).

Added in version 3.3.

Changed in version 3.13: Default to using the installed `mpdecimal` library.

Changed in version 3.15: A bundled copy of the library will no longer be selected implicitly if an installed `mpdecimal` library is not found. In Python 3.15 only, it can still be selected explicitly using `--with-system-libmpdec=no` or `--without-system-libmpdec`.

Deprecated since version 3.13, will be removed in version 3.16: A copy of the `mpdecimal` library sources will no longer be distributed with Python 3.16.

See also

`LIBMPDEC_CFLAGS` and `LIBMPDEC_LIBS`.

--with-readline=readline|editline

Designate a backend library for the `readline` module.

- readline: Use readline as the backend.
- editline: Use editline as the backend.

Added in version 3.10.

--without-readline

Don’t build the `readline` module (built by default).

Don’t define the `HAVE_LIBREADLINE` macro.

Added in version 3.10.

--with-libm=STRING

Override `libm` math library to *STRING* (default is system-dependent).

--with-libc=STRING

Override `libc` C library to *STRING* (default is system-dependent).

--with-openssl=DIR

Root of the OpenSSL directory.

Added in version 3.7.

--with-openssl-rpath=[no|auto|DIR]

Set runtime library directory (rpath) for OpenSSL libraries:

- `no` (default): don’t set rpath;
- `auto`: auto-detect rpath from **`--with-openssl`** and `pkg-config`;
- *DIR*: set an explicit rpath.

Added in version 3.10.

3.3.12. Security Options

--with-hash-algorithm=[fnv|siphash13|siphash24]

Select hash algorithm for use in `Python/pyhash.c`:

- `siphash13` (default);
- `siphash24`;
- `fnv`.

Added in version 3.4.

Added in version 3.11: `siphash13` is added and it is the new default.

--with-builtin-hashlib-hashes=md5,sha1,sha256,sha512,sha3,blake2

Built-in hash modules:

- `md5`;
- `sha1`;
- `sha256`;

- `sha512`;
- `sha3` (with shake);
- `blake2`.

Added in version 3.9.

--with-ssl-default-suites=[python|openssl|STRING]

Override the OpenSSL default cipher suites string:

- `python` (default): use Python’s preferred selection;
- `openssl`: leave OpenSSL’s defaults untouched;
- *STRING*: use a custom string

See the `ssl` module.

Added in version 3.7.

Changed in version 3.10: The settings `python` and *STRING* also set TLS 1.2 as minimum protocol version.

--disable-safety

Disable compiler options that are recommended by OpenSSF [https://github.com/ossf/wg-best-practices-os-developers/blob/main/docs/Compiler-Hardening-Guides/Compiler-Options-Hardening-Guide-for-C-and-C++.md] for security reasons with no performance overhead. If this option is not enabled, CPython will be built based on safety compiler options with no slow down. When this option is enabled, CPython will not be built with the compiler options listed below.

The following compiler options are disabled with **`--disable-safety`**:

- -fstack-protector-strong [https://github.com/ossf/wg-best-practices-os-developers/blob/main/docs/Compiler-Hardening-Guides/Compiler-Options-Hardening-Guide-for-C-and-C++.md#enable-run-time-checks-for-stack-based-buffer-overflows]: Enable run-time checks for stack-based buffer overflows.

- -Wtrampolines [https://github.com/ossf/wg-best-practices-os-developers/blob/main/docs/Compiler-Hardening-Guides/Compiler-Options-Hardening-Guide-for-C-and-C++.md#enable-warning-about-trampolines-that-require-executable-stacks]: Enable warnings about trampolines that require executable stacks.

Added in version 3.14.

--enable-slower-safety

Enable compiler options that are recommended by OpenSSF [https://github.com/ossf/wg-best-practices-os-developers/blob/main/docs/Compiler-Hardening-Guides/Compiler-Options-Hardening-Guide-for-C-and-C++.md] for security reasons which require overhead. If this option is not enabled, CPython will not be built based on safety compiler options which performance impact. When this option is enabled, CPython will be built with the compiler options listed below.

The following compiler options are enabled with **`--enable-slower-safety`**:

- -D_FORTIFY_SOURCE=3 [https://github.com/ossf/wg-best-practices-os-developers/blob/main/docs/Compiler-Hardening-Guides/Compiler-Options-Hardening-Guide-for-C-and-C++.md#fortify-sources-for-unsafe-libc-usage-and-buffer-overflows]: Fortify sources with compile- and run-time checks for unsafe libc usage and buffer overflows.

Added in version 3.14.

3.3.13. macOS Options

See Mac/README.rst [https://github.com/python/cpython/tree/3.14/Mac/README.rst].

--enable-universalsdk

--enable-universalsdk=SDKDIR

Create a universal binary build. *SDKDIR* specifies which macOS SDK should be used to perform the build (default is no).

--enable-framework

--enable-framework=INSTALLDIR

Create a Python.framework rather than a traditional Unix install. Optional *INSTALLDIR* specifies the installation path (default is no).

--with-universal-archs=ARCH

Specify the kind of universal binary that should be created. This option is only valid when **`--enable-universalsdk`** is set.

Options:

- `universal2` (x86-64 and arm64);
- `32-bit` (PPC and i386);
- `64-bit` (PPC64 and x86-64);
- `3-way` (i386, PPC and x86-64);
- `intel` (i386 and x86-64);
- `intel-32` (i386);
- `intel-64` (x86-64);
- `all` (PPC, i386, PPC64 and x86-64).

Note that values for this configuration item are *not* the same as the identifiers used for universal binary wheels on macOS. See the Python Packaging User Guide for details on the packaging platform compatibility tags used on macOS [https://packaging.python.org/en/latest/specifications/platform-compatibility-tags/#macos]

--with-framework-name=FRAMEWORK

Specify the name for the python framework on macOS only valid when `--enable-framework` is set (default: `Python`).

--with-app-store-compliance

--with-app-store-compliance=PATCH-FILE

The Python standard library contains strings that are known to trigger automated inspection tool errors when submitted for distribution by the macOS and iOS App Stores. If enabled, this option will apply the list of patches that are known to correct app store compliance. A custom patch file can also be specified. This option is disabled by default.

Added in version 3.13.

3.3.14. iOS Options

See iOS/README.rst [https://github.com/python/cpython/tree/3.14/iOS/README.rst].

--enable-framework=INSTALLDIR

Create a Python.framework. Unlike macOS, the *INSTALLDIR* argument specifying the installation path is mandatory.

--with-framework-name=FRAMEWORK

Specify the name for the framework (default: `Python`).

3.3.15. Cross Compiling Options

Cross compiling, also known as cross building, can be used to build Python for another CPU architecture or platform. Cross compiling requires a Python interpreter for the build platform. The version of the build Python must match the version of the cross compiled host Python.

--build=BUILD

configure for building on BUILD, usually guessed by **config.guess**.

--host=HOST

cross-compile to build programs to run on HOST (target platform)

--with-build-python=path/to/python

path to build `python` binary for cross compiling

Added in version 3.11.

CONFIG_SITE=file

An environment variable that points to a file with configure overrides.

Example *config.site* file:

```
# config.site-aarch64
ac_cv_buggy_getaddrinfo=no
ac_cv_file__dev_ptmx=yes
ac_cv_file__dev_ptc=no
```

HOSTRUNNER

Program to run CPython for the host platform for cross-compilation.

Added in version 3.11.

Cross compiling example:

```
CONFIG_SITE=config.site-aarch64 ../configure \
    --build=x86_64-pc-linux-gnu \
    --host=aarch64-unknown-linux-gnu \
    --with-build-python=../x86_64/python
```

3.4. Python Build System

3.4.1. Main files of the build system

- `configure.ac` => `configure`;
- `Makefile.pre.in` => `Makefile` (created by `configure`);
- `pyconfig.h` (created by `configure`);
- `Modules/Setup`: C extensions built by the Makefile using `Module/makesetup` shell script;

3.4.2. Main build steps

- C files (`.c`) are built as object files (`.o`).
- A static `libpython` library (`.a`) is created from objects files.
- `python.o` and the static `libpython` library are linked into the final `python` program.
- C extensions are built by the Makefile (see `Modules/Setup`).

3.4.3. Main Makefile targets

3.4.3.1. make

For the most part, when rebuilding after editing some code or refreshing your checkout from upstream, all you need to do is execute `make`, which (per Make’s semantics) builds the default target, the first one defined in the Makefile. By tradition (including in the CPython project) this is usually the `all` target. The `configure` script expands an `autoconf` variable, `@DEF_MAKE_ALL_RULE@` to describe precisely which targets `make all` will build. The three choices are:

- `profile-opt` (configured with `--enable-optimizations`)
- `build_wasm` (chosen if the host platform matches `wasm32-wasi*` or `wasm32-emscripten`)

- `build_all` (configured without explicitly using either of the others)

Depending on the most recent source file changes, Make will rebuild any targets (object files and executables) deemed out-of-date, including running `configure` again if necessary. Source/target dependencies are many and maintained manually however, so Make sometimes doesn’t have all the information necessary to correctly detect all targets which need to be rebuilt. Depending on which targets aren’t rebuilt, you might experience a number of problems. If you have build or test problems which you can’t otherwise explain, `make clean && make` should work around most dependency problems, at the expense of longer build times.

3.4.3.2. make platform

Build the `python` program, but don’t build the standard library extension modules. This generates a file named `platform` which contains a single line describing the details of the build platform, e.g., `macosx-14.3-arm64-3.12` or `linux-x86_64-3.13`.

3.4.3.3. make profile-opt

Build Python using profile-guided optimization (PGO). You can use the configure `--enable-optimizations` option to make this the default target of the `make` command (`make all` or just `make`).

3.4.3.4. make clean

Remove built files.

3.4.3.5. make distclean

In addition to the work done by `make clean`, remove files created by the configure script. `configure` will have to be run before building again. [[6]]

3.4.3.6. make install

Build the `all` target and install Python.

3.4.3.7. make test

Build the `all` target and run the Python test suite with the `--fast-ci` option without GUI tests. Variables:

- `TESTOPTS`: additional regrtest command-line options.
- `TESTPYTHONOPTS`: additional Python command-line options.
- `TESTTIMEOUT`: timeout in seconds (default: 10 minutes).

3.4.3.8. make ci

This is similar to `make test`, but uses the `-ugui` to also run GUI tests.

Added in version 3.14.

3.4.3.9. make buildbottest

This is similar to `make test`, but uses the `--slow-ci` option and default timeout of 20 minutes, instead of `--fast-ci` option.

3.4.3.10. make regen-all

Regenerate (almost) all generated files. These include (but are not limited to) bytecode cases, and parser generator file. `make regen-stdlib-module-names` and `autoconf` must be run separately for the remaining generated files.

3.4.4. C extensions

Some C extensions are built as built-in modules, like the `sys` module. They are built with the `Py_BUILD_CORE_BUILTIN` macro defined. Built-in modules have no `__file__` attribute:

```
>>> import sys
>>> sys
<module 'sys' (built-in)>
>>> sys.__file__
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: module 'sys' has no attribute '__file__'
```

Other C extensions are built as dynamic libraries, like the `_asyncio` module. They are built with the `Py_BUILD_CORE_MODULE` macro defined. Example on Linux x86-64:

```
>>> import _asyncio
>>> _asyncio
<module '_asyncio' from '/usr/lib64/python3.9/lib-
dynload/_asyncio.cpython-39-x86_64-linux-gnu.so'>
>>> _asyncio.__file__
'/usr/lib64/python3.9/lib-dynload/_asyncio.cpython-39-x86_64-
linux-gnu.so'
```

`Modules/Setup` is used to generate Makefile targets to build C extensions. At the beginning of the files, C extensions are built as built-in modules. Extensions defined after the `*shared*` marker are built as dynamic libraries.

The **`PyAPI_FUNC()`**, **`PyAPI_DATA()`** and **`PyMODINIT_FUNC`** macros of `Include/exports.h` are defined differently depending if the `Py_BUILD_CORE_MODULE` macro is defined:

- Use `Py_EXPORTED_SYMBOL` if the `Py_BUILD_CORE_MODULE` is defined
- Use `Py_IMPORTED_SYMBOL` otherwise.

If the `Py_BUILD_CORE_BUILTIN` macro is used by mistake on a C extension built as a shared library, its `PyInit_xxx()` function is not exported, causing an **`ImportError`** on import.

3.5. Compiler and linker flags

Options set by the `./configure` script and environment variables and used by `Makefile`.

3.5.1. Preprocessor flags

CONFIGURE_CPPFLAGS

Value of **`CPPFLAGS`** variable passed to the `./configure` script.

Added in version 3.6.

CPPFLAGS

(Objective) C/C++ preprocessor flags, e.g. `-Iinclude_dir` if you have headers in a nonstandard directory *include_dir*.

Both **`CPPFLAGS`** and **`LDFLAGS`** need to contain the shell's value to be able to build extension modules using the directories specified in the environment variables.

BASECPPFLAGS

Added in version 3.4.

PY_CPPFLAGS

Extra preprocessor flags added for building the interpreter object files.

Default: `$(BASECPPFLAGS) -I. -I$(srcdir)/Include $(CONFIGURE_CPPFLAGS) $(CPPFLAGS)`.

Added in version 3.2.

3.5.2. Compiler flags

CC

C compiler command.

Example: `gcc -pthread`.

CXX

C++ compiler command.

Example: `g++ -pthread`.

CFLAGS

C compiler flags.

CFLAGS_NODIST

`CFLAGS_NODIST` is used for building the interpreter and stdlib C extensions. Use it when a compiler flag should *not* be part of **`CFLAGS`** once Python is installed (gh-65320 [https://github.com/python/cpython/issues/65320]).

In particular, **`CFLAGS`** should not contain:

- the compiler flag `-I` (for setting the search path for include files). The `-I` flags are processed from left to right, and any flags in **`CFLAGS`** would take precedence over user- and package-supplied `-I` flags.
- hardening flags such as `-Werror` because distributions cannot control whether packages installed by users conform to such heightened standards.

Added in version 3.5.

COMPILEALL_OPTS

Options passed to the **`compileall`** command line when building PYC files in `make install`. Default: `-j0`.

Added in version 3.12.

EXTRA_CFLAGS

Extra C compiler flags.

CONFIGURE_CFLAGS

Value of **`CFLAGS`** variable passed to the `./configure` script.

Added in version 3.2.

CONFIGURE_CFLAGS_NODIST

Value of **`CFLAGS_NODIST`** variable passed to the `./configure` script.

Added in version 3.5.

BASECFLAGS

Base compiler flags.

OPT

Optimization flags.

CFLAGS_ALIASING

Strict or non-strict aliasing flags used to compile `Python/dtoa.c`.

Added in version 3.7.

CCSHARED

Compiler flags used to build a shared library.

For example, `-fPIC` is used on Linux and on BSD.

CFLAGSFORSHARED

Extra C flags added for building the interpreter object files.

Default: `$(CCSHARED)` when **`--enable-shared`** is used, or an empty string otherwise.

PY_CFLAGS

Default: `$(BASECFLAGS) $(OPT) $(CONFIGURE_CFLAGS) $(CFLAGS) $(EXTRA_CFLAGS)`.

PY_CFLAGS_NODIST

Default: `$(CONFIGURE_CFLAGS_NODIST) $(CFLAGS_NODIST) -I$(srcdir)/Include/internal`.

Added in version 3.5.

PY_STDMODULE_CFLAGS

C flags used for building the interpreter object files.

Default: `$(PY_CFLAGS) $(PY_CFLAGS_NODIST) $(PY_CPPFLAGS) $(CFLAGSFORSHARED)`.

Added in version 3.7.

PY_CORE_CFLAGS

Default: `$(PY_STDMODULE_CFLAGS) -DPy_BUILD_CORE`.

Added in version 3.2.

PY_BUILTIN_MODULE_CFLAGS

Compiler flags to build a standard library extension module as a built-in module, like the **`posix`** module.

Default: `$(PY_STDMODULE_CFLAGS) -DPy_BUILD_CORE_BUILTIN`.

Added in version 3.8.

PURIFY

Purify command. Purify is a memory debugger program.

Default: empty string (not used).

3.5.3. Linker flags

LINKCC

Linker command used to build programs like `python` and `_testembed`.

Default: `$(PURIFY) $(CC)`.

CONFIGURE_LDFLAGS

Value of **`LDFLAGS`** variable passed to the `./configure` script.

Avoid assigning **`CFLAGS`**, **`LDFLAGS`**, etc. so users can use them on the command line to append to these values without stomping the pre-set values.

Added in version 3.2.

LDFLAGS_NODIST

`LDFLAGS_NODIST` is used in the same manner as **`CFLAGS_NODIST`**. Use it when a linker flag should *not* be part of **`LDFLAGS`** once Python is installed (gh-65320 [https://github.com/python/cpython/issues/65320]).

In particular, **`LDFLAGS`** should not contain:

- the compiler flag `-L` (for setting the search path for libraries). The `-L` flags are processed from left to right, and any flags in **`LDFLAGS`** would take precedence over user- and package-supplied `-L` flags.

CONFIGURE_LDFLAGS_NODIST

Value of `LDFLAGS_NODIST` variable passed to the `./configure` script.

Added in version 3.8.

LDFLAGS

Linker flags, e.g. `-Llib_dir` if you have libraries in a nonstandard directory *lib_dir*.

Both `CPPFLAGS` and `LDFLAGS` need to contain the shell’s value to be able to build extension modules using the directories specified in the environment variables.

LIBS

Linker flags to pass libraries to the linker when linking the Python executable.

Example: `-lrt`.

LDSHARED

Command to build a shared library.

Default: `@LDSHARED@ $(PY_LDFLAGS)`.

BLDSHARED

Command to build `libpython` shared library.

Default: `@BLDSHARED@ $(PY_CORE_LDFLAGS)`.

PY_LDFLAGS

Default: `$(CONFIGURE_LDFLAGS) $(LDFLAGS)`.

PY_LDFLAGS_NODIST

Default: `$(CONFIGURE_LDFLAGS_NODIST) $(LDFLAGS_NODIST)`.

Added in version 3.8.

PY_CORE_LDFLAGS

Linker flags used for building the interpreter object files.

Added in version 3.8.

Footnotes

[6]

`git clean -fdx` is an even more extreme way to “clean” your checkout. It removes all files not known to Git. When bug hunting using `git bisect`, this is recommended between probes [https://github.com/python/cpython/issues/114505#issuecomment-1907021718] to guarantee a completely clean build. **Use with care**, as it will delete all files not checked into Git, including your new, uncommitted work.

4. Using Python on Windows

This document aims to give an overview of Windows-specific behaviour you should know about when using Python on Microsoft Windows.

Unlike most Unix systems and services, Windows does not include a system supported installation of Python. Instead, Python can be obtained from a number of distributors, including directly from the CPython team. Each Python distribution will have its own benefits and drawbacks, however, consistency with other tools you are using is generally a worthwhile benefit. Before committing to the process described here, we recommend investigating your existing tools to see if they can provide Python directly.

To obtain Python from the CPython team, use the Python Install Manager. This is a standalone tool that makes Python available as global commands on your Windows machine, integrates with the system, and supports updates over time. You can download the Python Install Manager from python.org/downloads [https://www.python.org/downloads/] or through the Microsoft Store app [https://apps.microsoft.com/detail/9NQ7512CXL7T].

Once you have installed the Python Install Manager, the global `python` command can be used from any terminal to launch your current latest version of Python. This version may change over time as you add or remove different versions, and the `py list` command will show which is current.

In general, we recommend that you create a virtual environment for each project and run `<env>\Scripts\Activate` in your terminal to use it. This provides isolation between projects, consistency over time, and ensures that additional commands added by packages are also available in your session. Create a virtual environment using `python -m venv <env path>`.

If the `python` or `py` commands do not seem to be working, please see the Troubleshooting section below. There are sometimes additional manual steps required to configure your PC.

Apart from using the Python install manager, Python can also be obtained as NuGet packages. See The nuget.org packages below for more information on these packages.

The embeddable distros are minimal packages of Python suitable for embedding into larger applications. They can be installed using the Python install manager. See The embeddable package below for more information on these packages.

4.1. Python install manager

4.1.1. Installation

The Python install manager can be installed from the Microsoft Store app [https://apps.microsoft.com/detail/9NQ7512CXL7T] or downloaded and installed from python.org/downloads [https://www.python.org/downloads/]. The two versions are identical.

To install through the Store, simply click “Install”. After it has completed, open a terminal and type `python` to get started.

To install the file downloaded from python.org, either double-click and select “Install”, or run `Add-AppxPackage <path to MSIX>` in Windows Powershell.

After installation, the `python`, `py`, and `pymanager` commands should be available. If you have existing installations of Python, or you have modified your **`PATH`** variable, you may need to remove them or undo the modifications. See Troubleshooting for more help with fixing non-working commands.

When you first install a runtime, you will likely be prompted to add a directory to your **`PATH`**. This is optional, if you prefer to use the `py` command, but is offered for those who prefer the full range of aliases (such as `python3.14.exe`) to be available. The directory will be `%LocalAppData%\Python\bin` by default, but may be customized by an administrator. Click Start and search for “Edit environment variables for your account” for the system settings page to add the path.

Each Python runtime you install will have its own directory for scripts. These also need to be added to **`PATH`** if you want to use them.

The Python install manager will be automatically updated to new releases. This does not affect any installs of Python runtimes. Uninstalling the Python install manager does not uninstall any Python runtimes.

If you are not able to install an MSIX in your context, for example, you are using automated deployment software that does not support it, or are targeting Windows Server 2019, please see Advanced installation below for more information.

4.1.2. Basic use

The recommended command for launching Python is `python`, which will either launch the version requested by the script being launched, an active virtual environment, or the default installed version, which will be the latest stable release unless configured otherwise. If no version is specifically requested and no runtimes are installed at all, the current latest release will be installed automatically.

For all scenarios involving multiple runtime versions, the recommended command is `py`. This may be used anywhere in place of `python` or the older `py.exe` launcher. By default, `py` matches the behaviour of `python`, but also allows command line options to select a specific version as well as subcommands to manage installations. These are detailed below.

Because the `py` command may already be taken by the previous version, there is also an unambiguous `pymanager` command. Scripted installs that are intending to use Python install manager should consider using `pymanager`, due to the lower chance of encountering a conflict with existing installs. The only difference between the two commands is when running without any arguments: `py` will install and launch your default interpreter, while `pymanager` will display help (`pymanager exec ...` provides equivalent behaviour to `py ...`).

Each of these commands also has a windowed version that avoids creating a console window. These are `pyw`, `pythonw` and `pymanagerw`. A `python3` command is also included that mimics the `python` command. It is intended to catch accidental uses of the typical POSIX command on Windows, but is not meant to be widely used or recommended.

To launch your default runtime, run `python` or `py` with the arguments you want to be passed to the runtime (such as script files or the module to launch):

```
$> py
...
$> python my-script.py
...
$> py -m this
...
```

The default runtime can be overridden with the **`PYTHON_MANAGER_DEFAULT`** environment variable, or a configuration file. See Configuration for information about configuration settings.

To launch a specific runtime, the `py` command accepts a `-V:<TAG>` option. This option must be specified before any others. The tag is part or all of the identifier for the runtime; for those from the CPython team, it looks like the version, potentially with the platform. For compatibility, the `V:` may be omitted in cases where the tag refers to an official release and starts with `3`.

```
$> py -V:3.14 ...
$> py -V:3-arm64 ...
```

Runtimes from other distributors may require the *company* to be included as well. This should be separated from the tag by a slash, and may be a prefix. Specifying the company is optional when it is `PythonCore`, and specifying the tag is optional (but not the slash) when you want the latest release from a specific company.

```
$> py -V:Distributor\1.0 ...
$> py -V:distrib/ ...
```

If no version is specified, but a script file is passed, the script will be inspected for a *shebang line*. This is a special format for the first line in a file that allows overriding the command. See Shebang lines for more information. When there is no shebang line, or it cannot be resolved, the script will be launched with the default runtime.

If you are running in an active virtual environment, have not requested a particular version, and there is no shebang line, the default runtime will be that virtual environment. In this scenario, the `python` command was likely already overridden and none of these checks occurred. However, this behaviour ensures that the `py` command can be used interchangeably.

When you launch either `python` or `py` but do not have any runtimes installed, and the requested version is the default, it will be installed automatically and then launched. Otherwise, the requested version will be installed if automatic installation is configured (most likely by setting `PYTHON_MANAGER_AUTOMATIC_INSTALL` to `true`), or if the `py exec` or `pymanager exec` forms of the command were used.

4.1.3. Command help

The `py help` command will display the full list of supported commands, along with their options. Any command may be passed the `-?` option to display its help, or its name passed to `py help`.

```
$> py help
$> py help install
$> py install /?
```

All commands support some common options, which will be shown by `py help`. These options must be specified after any subcommand. Specifying `-v` or `--verbose` will increase the amount of output shown, and `-vv` will increase it further for debugging purposes. Passing `-q` or `--quiet` will reduce output, and `-qq` will reduce it further.

The `--config=<PATH>` option allows specifying a configuration file to override multiple settings at once. See Configuration below for more information about these files.

4.1.4. Listing runtimes

```
$> py list [-f=|--format=<FMT>] [-1|--one] [--online|-s=|--source=<URL>] [<TAG>...]
```

The list of installed runtimes can be seen using `py list`. A filter may be added in the form of one or more tags (with or without company specifier), and each may include a `<`, `<=`, `>=` or `>` prefix to restrict to a range.

A range of formats are supported, and can be passed as the `--format=<FMT>` or `-f <FMT>` option. Formats include `table` (a user friendly table view), `csv` (comma-separated table), `json` (a single JSON blob), `jsonl` (one JSON blob per result), `exe` (just the executable path), `prefix` (just the prefix path).

The `--one` or `-1` option only displays a single result. If the default runtime is included, it will be the one. Otherwise, the “best” result is shown (“best” is deliberately vaguely defined, but will usually be the most recent version). The result shown by `py list --one <TAG>` will match the runtime that would be launched by `py -V:<TAG>`.

The `--only-managed` option excludes results that were not installed by the Python install manager. This is useful when determining which runtimes may be updated or uninstalled through the `py` command.

The `--online` option is short for passing `--source=<URL>` with the default source. Passing either of these options will search the online index for runtimes that can be installed. The result shown by `py list --online --one <TAG>` will match the runtime that would be installed by `py install <TAG>`.

```
$> py list --online 3.14
```

For compatibility with the old launcher, the `--list`, `--list-paths`, `-0` and `-0p` commands (e.g. `py -0p`) are retained. They do not allow additional options, and will produce legacy formatted output.

4.1.5. Installing runtimes

```
$> py install [-s=|--source=<URL>] [-f|--force] [-u|--update] [--dry-run] [<TAG>...]
```

New runtime versions may be added using `py install`. One or more tags may be specified, and the special tag `default` may be used to select the default. Ranges are not supported for installation.

The `--source=<URL>` option allows overriding the online index that is used to obtain runtimes. This may be used with an offline index, as shown in Offline installs.

Passing `--force` will ignore any cached files and remove any existing install to replace it with the specified one.

Passing `--update` will replace existing installs if the new version is newer. Otherwise, they will be left. If no tags are provided with `--update`, all installs managed by the Python install manager will be updated if newer versions are available. Updates will remove any modifications made to the install, including globally installed packages, but virtual environments will continue to work.

Passing `--dry-run` will generate output and logs, but will not modify any installs.

In addition to the above options, the `--target` option will extract the runtime to the specified directory instead of doing a normal install. This is useful for embedding runtimes into larger applications. Unlike a normal install, `py` will not be aware of the extracted runtime, and no Start menu or other shortcuts will be created. To launch the runtime, directly execute the main executable (typically `python.exe`) in the target directory.

```
$> py install ... [-t=|--target=<PATH>] <TAG>
```

4.1.6. Offline installs

To perform offline installs of Python, you will need to first create an offline index on a machine that has network access.

```
$> py install --download=<PATH> ... <TAG>...
```

The `--download=<PATH>` option will download the packages for the listed tags and create a directory containing them and an `index.json` file suitable for later installation. This entire directory can be moved to the offline machine and used to install one or more of the bundled runtimes:

```
$> py install --source="<PATH>\index.json" <TAG>...
```

The Python install manager can be installed by downloading its installer and moving it to another machine before installing.

Alternatively, the ZIP files in an offline index directory can simply be transferred to another machine and extracted. This will not register the install in any way, and so it must be launched by directly referencing the executables in the extracted directory, but it is sometimes a preferable approach in cases where installing the Python install manager is not possible or convenient.

In this way, Python runtimes can be installed and managed on a machine without access to the internet.

4.1.7. Uninstalling runtimes

```
$> py uninstall [-y|--yes] <TAG>...
```

Runtimes may be removed using the `py uninstall` command. One or more tags must be specified. Ranges are not supported here.

The `--yes` option bypasses the confirmation prompt before uninstalling.

Instead of passing tags individually, the `--purge` option may be specified. This will remove all runtimes managed by the Python install manager, including cleaning up the Start menu, registry, and any download caches. Runtimes that were not installed by the Python install manager will not be impacted, and neither will manually created configuration files.

```
$> py uninstall [-y|--yes] --purge
```

The Python install manager can be uninstalled through the Windows “Installed apps” settings page. This does not remove any runtimes, and they will still be usable, though the global `python` and `py` commands will be removed. Reinstalling the Python install manager will allow you to manage these runtimes again. To completely clean up all Python runtimes, run with `--purge` before uninstalling the Python install manager.

4.1.8. Configuration

Python install manager is configured with a hierarchy of configuration files, environment variables, command-line options, and registry settings. In general, configuration files have the ability to configure everything, including the location of other configuration files, while registry settings are administrator-only and will override configuration files. Command-line options override all other settings, but not every option is available.

This section will describe the defaults, but be aware that modified or overridden installs may resolve settings differently.

A global configuration file may be configured by an administrator, and would be read first. The user configuration file is stored at `%AppData%\Python\pymanager.json` (note that this location is under `Roaming`, not `Local`) and is read next, overwriting any settings from earlier files. An additional configuration file may be specified as the `PYTHON_MANAGER_CONFIG` environment variable or the `--config` command line option (but not both). These locations may be modified by administrative customization options listed later.

The following settings are those that are considered likely to be modified in normal use. Later sections list those that are intended for administrative customization.

Standard configuration options

Config Key	Environment Variable	Description
`default_tag`	PYTHON_MANAGER_DEFAULT	The preferred default version to launch or install. By default, this is interpreted as the most recent non-prerelease version from the CPython team.
`default_platform`	`PYTHON_MANAGER_DEFAULT_PLATFORM`	The preferred default platform to launch or install. This is treated as a suffix to the specified tag, such that `py -V:3.14` would prefer an install for `3.14-64` if it exists (and `default_platform` is `-64`), but will use `3.14` if no tagged install exists.

Config Key	Environment Variable	Description
`logs_dir`	`PYTHON_MANAGER_LOGS`	The location where log files are written. By default, `%TEMP%`.
`automatic_install`	`PYTHON_MANAGER_AUTOMATIC_INSTALL`	True to allow automatic installs when using `py exec` to launch. Other commands will not automatically install. By default, true.
`include_unmanaged`	`PYTHON_MANAGER_INCLUDE_UNMANAGED`	True to allow listing and launching runtimes that were not installed by the Python install manager, or false to exclude them. By default, true.
`shebang_can_run_anything`	`PYTHON_MANAGER_SHEBANG_CAN_RUN_ANYTHING`	True to allow shebangs in `.py` files to launch applications other than Python runtimes, or false to prevent it. By default, true.
`log_level`	`PYMANAGER_VERBOSE`, `PYMANAGER_DEBUG`	Set the default level of output (0-50). By default, 20. Lower values produce more output. The

Config Key	Environment Variable	Description
		environment variables are boolean, and may produce additional output during startup that is later suppressed by other configuration.
`confirm`	`PYTHON_MANAGER_CONFIRM`	True to confirm certain actions before taking them (such as uninstall), or false to skip the confirmation. By default, true.
`install.source`	`PYTHON_MANAGER_SOURCE_URL`	Override the index feed to obtain new installs from.
`list.format`	`PYTHON_MANAGER_LIST_FORMAT`	Specify the default format used by the `py list` command. By default, `table`.
`install_dir`	(none)	Specify the root directory that runtimes will be installed into. If you change this setting, previously installed runtimes will not be usable unless you move them to the new location.

Config Key	Environment Variable	Description
`global_dir`	(none)	Specify the directory where global commands (such as `python3.14.exe`) are stored. This directory should be added to your **`PATH`** to make the commands available from your terminal.
`download_dir`	(none)	Specify the directory where downloaded files are stored. This directory is a temporary cache, and can be cleaned up from time to time.

Dotted names should be nested inside JSON objects, for example, `list.format` would be specified as `{"list": {"format": "table"}}`.

4.1.9. Shebang lines

If the first line of a script file starts with `#!`, it is known as a "shebang" line. Linux and other Unix like operating systems have native support for such lines and they are commonly used on such systems to indicate how a script should be executed. The `python` and `py` commands allow the same facilities to be used with Python scripts on Windows.

To allow shebang lines in Python scripts to be portable between Unix and Windows, a number of 'virtual' commands are supported to specify which interpreter to use. The supported virtual commands are:

- `/usr/bin/env <ALIAS>`
- `/usr/bin/env -S <ALIAS>`
- `/usr/bin/<ALIAS>`
- `/usr/local/bin/<ALIAS>`
- `<ALIAS>`

For example, if the first line of your script starts with

```
#! /usr/bin/python
```

The default Python or an active virtual environment will be located and used. As many Python scripts written to work on Unix will already have this line, you should find these scripts can be used by the launcher without modification. If you are writing a new script on Windows which you hope will be useful on Unix, you should use one of the shebang lines starting with `/usr`.

Any of the above virtual commands can have `<ALIAS>` replaced by an alias from an installed runtime. That is, any command generated in the global aliases directory (which you may have added to your **`PATH`** environment variable) can be used in a shebang, even if it is not on your **`PATH`**. This allows the use of shebangs like `/usr/bin/python3.12` to select a particular runtime.

If no runtimes are installed, or if automatic installation is enabled, the requested runtime will be installed if necessary. See Configuration for information about configuration settings.

The `/usr/bin/env` form of shebang line will also search the **`PATH`** environment variable for unrecognized commands. This corresponds to the behaviour of the Unix `env` program, which performs the same search, but prefers launching known Python commands. A warning may be displayed when searching for arbitrary executables, and this search may be disabled by the `shebang_can_run_anything` configuration option.

Shebang lines that do not match any of patterns are treated as *Windows* executable paths that are absolute or relative to the directory containing the script file. This is a convenience for Windows-only scripts, such as those generated by an installer, since the behavior is not compatible with Unix-style shells. These paths may be quoted, and may include multiple arguments, after which the path to the script and any additional arguments will be appended. This functionality may be disabled by the `shebang_can_run_anything` configuration option.

Note

The behaviour of shebangs in the Python install manager is subtly different from the previous `py.exe` launcher, and the old configuration options no longer apply. If you are specifically reliant on the old behaviour or configuration, we recommend installing the legacy launcher [https://www.python.org/ftp/python/3.14.0/win32/launcher.msi]. The legacy launcher's `py` command will override PyManager's one by default, and you will need to use `pymanager` commands for installing and uninstalling.

4.1.10. Advanced installation

For situations where an MSIX cannot be installed, such as some older administrative distribution platforms, there is an MSI available from the python.org downloads page. This MSI has no user interface, and can only perform per-machine installs to its default location in Program Files. It will attempt to modify the system **PATH** environment variable to include this install location, but be sure to validate this on your configuration.

Note

Windows Server 2019 is the only version of Windows that CPython supports that does not support MSIX. For Windows Server 2019, you should use the MSI.

Be aware that the MSI package does not bundle any runtimes, and so is not suitable for installs into offline environments without also creating an offline install index. See Offline installs and Administrative configuration for information on handling these scenarios.

Runtimes installed by the MSI are shared with those installed by the MSIX, and are all per-user only. The Python install manager does not support installing runtimes per-machine. To emulate a per-machine install, you can use `py install --target=<shared location>` as administrator and add your own system-wide modifications to **PATH**, the registry, or the Start menu.

When the MSIX is installed, but commands are not available in the **PATH** environment variable, they can be found under `%LocalAppData%\Microsoft\WindowsApps\PythonSoftwareFoundation.PythonManager_3847v3x7pw1km` or `%LocalAppData%\Microsoft\WindowsApps\PythonSoftwareFoundation.PythonManager_qbz5n2kfra8p0`, depending on whether it was installed from python.org or through the Windows Store. Attempting to run the executable directly from Program Files is not recommended.

To programmatically install the Python install manager, it is easiest to use WinGet, which is included with all supported versions of Windows:

```
$> winget install 9NQ7512CXL7T -e --accept-package-agreements --disable-interactivity

# Optionally run the configuration checker and accept all changes
$> py install --configure -y
```

To download the Python install manager and install on another machine, the following WinGet command will download the required files from the Store to your Downloads directory (add `-d <location>` to customize the output location). This also generates a YAML file that appears to be unnecessary, as the downloaded MSIX can be installed by launching or using the commands below.

```
$> winget download 9NQ7512CXL7T -e --skip-license --accept-package-agreements --
accept-source-agreements
```

To programmatically install or uninstall an MSIX using only PowerShell, the Add-AppxPackage [https://learn.microsoft.com/powershell/module/appx/add-appxpackage] and Remove-

[AppxPackage](https://learn.microsoft.com/powershell/module/appx/remove-appxpackage) [https://learn.microsoft.com/powershell/module/appx/remove-appxpackage] PowerShell cmdlets are recommended:

```
$> Add-AppxPackage C:\Downloads\python-manager-25.0.msix
...
$> Get-AppxPackage PythonSoftwareFoundation.PythonManager | Remove-AppxPackage
```

The latest release can be downloaded and installed by Windows by passing the AppInstaller file to the Add-AppxPackage command. This installs using the MSIX on python.org, and is only recommended for cases where installing via the Store (interactively or using WinGet) is not possible.

```
$> Add-AppxPackage -AppInstallerFile
https://www.python.org/ftp/python/pymanager/pymanager.appinstaller
```

Other tools and APIs may also be used to provision an MSIX package for all users on a machine, but Python does not consider this a supported scenario. We suggest looking into the PowerShell [Add-AppxProvisionedPackage](https://learn.microsoft.com/powershell/module/dism/add-appxprovisionedpackage) [https://learn.microsoft.com/powershell/module/dism/add-appxprovisionedpackage] cmdlet, the native Windows [PackageManager](https://learn.microsoft.com/uwp/api/windows.management.deployment.packagemanager) [https://learn.microsoft.com/uwp/api/windows.management.deployment.packagemanager] class, or the documentation and support for your deployment tool.

Regardless of the install method, users will still need to install their own copies of Python itself, as there is no way to trigger those installs without being a logged in user. When using the MSIX, the latest version of Python will be available for all users to install without network access.

Note that the MSIX downloadable from the Store and from the Python website are subtly different and cannot be installed at the same time. Wherever possible, we suggest using the above WinGet commands to download the package from the Store to reduce the risk of setting up conflicting installs. There are no licensing restrictions on the Python install manager that would prevent using the Store package in this way.

4.1.11. Administrative configuration

There are a number of options that may be useful for administrators to override configuration of the Python install manager. These can be used to provide local caching, disable certain shortcut types, override bundled content. All of the above configuration options may be set, as well as those below.

Configuration options may be overridden in the registry by setting values under `HKEY_LOCAL_MACHINE\Software\Policies\Python\PyManager`, where the value name matches the configuration key and the value type is `REG_SZ`. Note that this key can itself be customized, but only by modifying the core config file distributed with the Python install manager. We recommend, however, that registry values are used only to set `base_config` to a JSON file containing the full set of overrides. Registry key overrides will replace any other configured setting, while `base_config` allows users to further modify settings they may need.

Note that most settings with environment variables support those variables because their default setting specifies the variable. If you override them, the environment variable will no longer work, unless you override it with another one. For example, the default value of `confirm` is literally `%PYTHON_MANAGER_CONFIRM%`, which will resolve the variable at load time. If you override the value to `yes`, then the environment variable will no longer be used. If you override the value to `%CONFIRM%`, then that environment variable will be used instead.

Configuration settings that are paths are interpreted as relative to the directory containing the configuration file that specified them.

Administrative configuration options

Config Key	Description
`base_config`	The highest priority configuration file to read. Note that only the built-in configuration file and the registry can modify this setting.
`user_config`	The second configuration file to read.
`additional_config`	The third configuration file to read.
`registry_override_key`	Registry location to check for overrides. Note that only the built-in configuration file can modify this setting.
`bundled_dir`	Read-only directory containing locally cached files.
`install.fallback_source`	Path or URL to an index to consult when the main index cannot be accessed.
`install.enable_shortcut_kinds`	Comma-separated list of shortcut kinds to allow (e.g. `"pep514,start"`). Enabled shortcuts may still be disabled by `disable_shortcut_kinds`.
`install.disable_shortcut_kinds`	Comma-separated list of shortcut kinds to exclude (e.g. `"pep514,start"`). Disabled shortcuts are not reactivated by `enable_shortcut_kinds`.

Config Key	Description
`pep514_root`	Registry location to read and write PEP 514 entries into. By default, `HKEY_CURRENT_USER\Software\Python`.
`start_folder`	Start menu folder to write shortcuts into. By default, `Python`. This path is relative to the user's Programs folder.
`virtual_env`	Path to the active virtual environment. By default, this is `%VIRTUAL_ENV%`, but may be set empty to disable venv detection.
`shebang_can_run_anything_silently`	True to suppress visible warnings when a shebang launches an application other than a Python runtime.

4.1.12. Installing free-threaded binaries

Added in version 3.13.

Pre-built distributions of the free-threaded build are available by installing tags with the `t` suffix.

```
$> py install 3.14t
$> py install 3.14t-arm64
$> py install 3.14t-32
```

This will install and register as normal. If you have no other runtimes installed, then `python` will launch this one. Otherwise, you will need to use `py -V:3.14t ...` or, if you have added the global aliases directory to your **PATH** environment variable, the `python3.14t.exe` commands.

4.1.13. Troubleshooting

If your Python install manager does not seem to be working correctly, please work through these tests and fixes to see if it helps. If not, please report an issue at our bug tracker [https://github.com/python/cpython/issues], including any relevant log files (written to your `%TEMP%` directory by default).

Troubleshooting

Symptom	**Things to try**
`python` gives me a “command not found” error or opens the Store app when I type it in my terminal.	Did you install the Python install manager?
	Click Start, open “Manage app execution aliases”, and check that the aliases for “Python (default)” are enabled. If they already are, try disabling and re-enabling to refresh the command. The “Python (default windowed)” and “Python install manager” commands may also need refreshing.
	Check that the `py` and `pymanager` commands work.
	Ensure your **PATH** variable contains the entry for `%UserProfile%\AppData\Local\Microsoft\WindowsApps`. The operating system includes this entry once by default, after other user paths. If removed, shortcuts will not be found.
`py` gives me a “command not found” error when I type it in my terminal.	Did you install the Python install manager?
	Click Start, open “Manage app execution aliases”, and check that the aliases for “Python (default)” are enabled. If they already are, try disabling and re-enabling to refresh the command. The “Python (default windowed)” and “Python install manager” commands may also need refreshing.
	Ensure your **PATH** variable contains the entry for `%UserProfile%\AppData\Local\Microsoft\WindowsApps`. The operating system includes this entry once by default, after other user paths. If removed, shortcuts will not be found.

Symptom	Things to try
`py` gives me a “can’t open file” error when I type commands in my terminal.	This usually means you have the legacy launcher installed and it has priority over the Python install manager. To remove, click Start, open “Installed apps”, search for “Python launcher” and uninstall it.
`python` doesn’t launch the same runtime as `py`	Click Start, open “Installed apps”, look for any existing Python runtimes, and either remove them or Modify and disable the **`PATH`** options.
	Click Start, open “Manage app execution aliases”, and check that your `python.exe` alias is set to “Python (default)”
`python` and `py` don’t launch the runtime I expect	Check your **`PYTHON_MANAGER_DEFAULT`** environment variable or `default_tag` configuration. The `py list` command will show your default based on these settings.
	Installs that are managed by the Python install manager will be chosen ahead of unmanaged installs. Use `py install` to install the runtime you expect, or configure your default tag.
	Prerelease and experimental installs that are not managed by the Python install manager may be chosen ahead of stable releases. Configure your default tag or uninstall the prerelease runtime and reinstall it using `py install`.
`pythonw` or `pyw` don’t launch the same runtime as `python` or `py`	Click Start, open “Manage app execution aliases”, and check that your `pythonw.exe` and `pyw.exe` aliases are consistent with your others.
`pip` gives me a “command not found” error	Have you activated a virtual environment? Run the `.venv\Scripts\activate` script in your terminal to activate.

Symptom	Things to try
when I type it in my terminal.	
	The package may be available but missing the generated executable. We recommend using the `python -m pip` command instead, or alternatively the `python -m pip install --force pip` command will recreate the executables and show you the path to add to **`PATH`**. These scripts are separated for each runtime, and so you may need to add multiple paths.
Typing `script-name.py` in the terminal opens in a new window.	This is a known limitation of the operating system. Either specify `py` before the script name, create a batch file containing `@py "%~dpn0.py" %*` with the same name as the script, or install the legacy launcher and select it as the association for scripts.
Drag-dropping files onto a script doesn't work	This is a known limitation of the operating system. It is supported with the legacy launcher, or with the Python install manager when installed from the MSI.
I have installed the Python install manager multiple times.	It is possible to install from the Store or WinGet, from the MSIX on the Python website, and from the MSI, all at once. They are all compatible and will share configuration and runtimes.
	See the earlier Advanced installation section for ways to uninstall the install manager other than the typical Installed Apps (Add and Remove Programs) settings page.
My old `py.ini` settings no longer work.	The new Python install manager no longer supports this configuration file or its settings, and so it will be ignored. See Configuration for information about configuration settings.

4.2. The embeddable package

Added in version 3.5.

The embedded distribution is a ZIP file containing a minimal Python environment. It is intended for acting as part of another application, rather than being directly accessed by end-users.

To install an embedded distribution, we recommend using `py install` with the `--target` option:

```
$> py install 3.14-embed --target=<directory>
```

When extracted, the embedded distribution is (almost) fully isolated from the user's system, including environment variables, system registry settings, and installed packages. The standard library is included as pre-compiled and optimized `.pyc` files in a ZIP, and `python3.dll`, `python313.dll`, `python.exe` and `pythonw.exe` are all provided. Tcl/tk (including all dependents, such as Idle), pip and the Python documentation are not included.

A default `._pth` file is included, which further restricts the default search paths (as described below in Finding modules). This file is intended for embedders to modify as necessary.

Third-party packages should be installed by the application installer alongside the embedded distribution. Using pip to manage dependencies as for a regular Python installation is not supported with this distribution, though with some care it may be possible to include and use pip for automatic updates. In general, third-party packages should be treated as part of the application ("vendoring") so that the developer can ensure compatibility with newer versions before providing updates to users.

The two recommended use cases for this distribution are described below.

4.2.1. Python application

An application written in Python does not necessarily require users to be aware of that fact. The embedded distribution may be used in this case to include a private version of Python in an install package. Depending on how transparent it should be (or conversely, how professional it should appear), there are two options.

Using a specialized executable as a launcher requires some coding, but provides the most transparent experience for users. With a customized launcher, there are no obvious indications that the program is running on Python: icons can be customized, company and version information can be specified, and file associations behave properly. In most cases, a custom launcher should simply be able to call `Py_Main` with a hard-coded command line.

The simpler approach is to provide a batch file or generated shortcut that directly calls the `python.exe` or `pythonw.exe` with the required command-line arguments. In this case, the application will appear to be Python and not its actual name, and users may have trouble distinguishing it from other running Python processes or file associations.

With the latter approach, packages should be installed as directories alongside the Python executable to ensure they are available on the path. With the specialized launcher, packages can be located in other locations as there is an opportunity to specify the search path before launching the application.

4.2.2. Embedding Python

Applications written in native code often require some form of scripting language, and the embedded Python distribution can be used for this purpose. In general, the majority of the application is in native code, and some part will either invoke `python.exe` or directly use `python3.dll`. For either case, extracting the embedded distribution to a subdirectory of the application installation is sufficient to provide a loadable Python interpreter.

As with the application use, packages can be installed to any location as there is an opportunity to specify search paths before initializing the interpreter. Otherwise, there is no fundamental differences between using the embedded distribution and a regular installation.

4.3. The nuget.org packages

Added in version 3.5.2.

The nuget.org package is a reduced size Python environment intended for use on continuous integration and build systems that do not have a system-wide install of Python. While nuget is “the package manager for .NET”, it also works perfectly fine for packages containing build-time tools.

Visit nuget.org [https://www.nuget.org/] for the most up-to-date information on using nuget. What follows is a summary that is sufficient for Python developers.

The `nuget.exe` command line tool may be downloaded directly from `https://aka.ms/nugetclidl`, for example, using curl or PowerShell. With the tool, the latest version of Python for 64-bit or 32-bit machines is installed using:

```
nuget.exe install python -ExcludeVersion -OutputDirectory .
nuget.exe install pythonx86 -ExcludeVersion -OutputDirectory .
```

To select a particular version, add a `-Version 3.x.y`. The output directory may be changed from `.`, and the package will be installed into a subdirectory. By default, the subdirectory is named the same as the package, and without the `-ExcludeVersion` option this name will include the specific version installed. Inside the subdirectory is a `tools` directory that contains the Python installation:

```
# Without -ExcludeVersion
> .\python.3.5.2\tools\python.exe -V
Python 3.5.2

# With -ExcludeVersion
> .\python\tools\python.exe -V
Python 3.5.2
```

In general, nuget packages are not upgradeable, and newer versions should be installed side-by-side and referenced using the full path. Alternatively, delete the package directory manually and install it again. Many CI systems will do this automatically if they do not preserve files between builds.

Alongside the `tools` directory is a `build\native` directory. This contains a MSBuild properties file `python.props` that can be used in a C++ project to reference the Python install. Including the settings will automatically use the headers and import libraries in your build.

The package information pages on nuget.org are www.nuget.org/packages/python [https://www.nuget.org/packages/python] for the 64-bit version, www.nuget.org/packages/pythonx86 [https://www.nuget.org/packages/pythonx86] for the 32-bit version, and www.nuget.org/packages/pythonarm64 [https://www.nuget.org/packages/pythonarm64] for the ARM64 version

4.3.1. Free-threaded packages

Added in version 3.13.

Packages containing free-threaded binaries are named python-freethreaded [https://www.nuget.org/packages/python-freethreaded] for the 64-bit version, pythonx86-freethreaded [https://www.nuget.org/packages/pythonx86-freethreaded] for the 32-bit version, and pythonarm64-freethreaded [https://www.nuget.org/packages/pythonarm64-freethreaded] for the ARM64 version. These packages contain both the `python3.13t.exe` and `python.exe` entry points, both of which run free threaded.

4.4. Alternative bundles

Besides the standard CPython distribution, there are modified packages including additional functionality. The following is a list of popular versions and their key features:

ActivePython [https://www.activestate.com/products/python/]

Installer with multi-platform compatibility, documentation, PyWin32

Anaconda [https://www.anaconda.com/download/]

Popular scientific modules (such as numpy, scipy and pandas) and the `conda` package manager.

Enthought Deployment Manager [https://assets.enthought.com/downloads/edm/]

"The Next Generation Python Environment and Package Manager".

Previously Enthought provided Canopy, but it reached end of life in 2016 [https://support.enthought.com/hc/en-us/articles/360038600051-Canopy-GUI-end-of-life-transition-to-the-Enthought-Deployment-Manager-EDM-and-Visual-Studio-Code].

WinPython [https://winpython.github.io/]

> Windows-specific distribution with prebuilt scientific packages and tools for building packages.

Note that these packages may not include the latest versions of Python or other libraries, and are not maintained or supported by the core Python team.

4.5. Supported Windows versions

As specified in **PEP 11** [https://peps.python.org/pep-0011/], a Python release only supports a Windows platform while Microsoft considers the platform under extended support. This means that Python 3.14 supports Windows 10 and newer. If you require Windows 7 support, please install Python 3.8. If you require Windows 8.1 support, please install Python 3.12.

4.6. Removing the MAX_PATH limitation

Windows historically has limited path lengths to 260 characters. This meant that paths longer than this would not resolve and errors would result.

In the latest versions of Windows, this limitation can be expanded to over 32,000 characters. Your administrator will need to activate the “Enable Win32 long paths” group policy, or set `LongPathsEnabled` to `1` in the registry key `HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control\FileSystem`.

This allows the `open()` function, the `os` module and most other path functionality to accept and return paths longer than 260 characters.

After changing the above option and rebooting, no further configuration is required.

4.7. UTF-8 mode

Added in version 3.7.

Windows still uses legacy encodings for the system encoding (the ANSI Code Page). Python uses it for the default encoding of text files (e.g. `locale.getencoding()`).

This may cause issues because UTF-8 is widely used on the internet and most Unix systems, including WSL (Windows Subsystem for Linux).

You can use the Python UTF-8 Mode to change the default text encoding to UTF-8. You can enable the Python UTF-8 Mode via the `-X utf8` command line option, or the `PYTHONUTF8=1` environment variable. See `PYTHONUTF8` for enabling UTF-8 mode, and Python install manager for how to modify environment variables.

When the Python UTF-8 Mode is enabled, you can still use the system encoding (the ANSI Code Page) via the “mbcs” codec.

Note that adding `PYTHONUTF8=1` to the default environment variables will affect all Python 3.7+ applications on your system. If you have any Python 3.7+ applications which rely on the legacy system encoding, it is recommended to set the environment variable temporarily or use the `-X utf8` command line option.

Note

Even when UTF-8 mode is disabled, Python uses UTF-8 by default on Windows for:

- Console I/O including standard I/O (see **PEP 528** [https://peps.python.org/pep-0528/] for details).
- The filesystem encoding (see **PEP 529** [https://peps.python.org/pep-0529/] for details).

4.8. Finding modules

These notes supplement the description at The initialization of the sys.path module search path with detailed Windows notes.

When no `._pth` file is found, this is how **`sys.path`** is populated on Windows:

- An empty entry is added at the start, which corresponds to the current directory.
- If the environment variable **`PYTHONPATH`** exists, as described in Environment variables, its entries are added next. Note that on Windows, paths in this variable must be separated by semicolons, to distinguish them from the colon used in drive identifiers (`C:\` etc.).
- Additional “application paths” can be added in the registry as subkeys of `\SOFTWARE\Python\PythonCore{version}\PythonPath` under both the `HKEY_CURRENT_USER` and `HKEY_LOCAL_MACHINE` hives. Subkeys which have semicolon-delimited path strings as their default value will cause each path to be added to **`sys.path`**. (Note that all known installers only use HKLM, so HKCU is typically empty.)
- If the environment variable **`PYTHONHOME`** is set, it is assumed as “Python Home”. Otherwise, the path of the main Python executable is used to locate a “landmark file” (either `Lib\os.py` or `pythonXY.zip`) to deduce the “Python Home”. If a Python home is found, the relevant sub-directories added to **`sys.path`** (`Lib`, `plat-win`, etc) are based on that folder. Otherwise, the core Python path is constructed from the PythonPath stored in the registry.
- If the Python Home cannot be located, no **`PYTHONPATH`** is specified in the environment, and no registry entries can be found, a default path with relative entries is used (e.g. `.\Lib;.\plat-win`, etc).

If a `pyvenv.cfg` file is found alongside the main executable or in the directory one level above the executable, the following variations apply:

- If `home` is an absolute path and **`PYTHONHOME`** is not set, this path is used instead of the path to the main executable when deducing the home location.

The end result of all this is:

- When running `python.exe`, or any other .exe in the main Python directory (either an installed version, or directly from the PCbuild directory), the core path is deduced, and the core paths in the registry are ignored. Other “application paths” in the registry are always read.
- When Python is hosted in another .exe (different directory, embedded via COM, etc), the “Python Home” will not be deduced, so the core path from the registry is used. Other “application paths” in the registry are always read.
- If Python can’t find its home and there are no registry value (frozen .exe, some very strange installation setup) you get a path with some default, but relative, paths.

For those who want to bundle Python into their application or distribution, the following advice will prevent conflicts with other installations:

- Include a `._pth` file alongside your executable containing the directories to include. This will ignore paths listed in the registry and environment variables, and also ignore **`site`** unless `import site` is listed.
- If you are loading `python3.dll` or `python37.dll` in your own executable, explicitly set **`PyConfig.module_search_paths`** before **`Py_InitializeFromConfig()`**.
- Clear and/or overwrite **`PYTHONPATH`** and set **`PYTHONHOME`** before launching `python.exe` from your application.
- If you cannot use the previous suggestions (for example, you are a distribution that allows people to run `python.exe` directly), ensure that the landmark file (`Lib\os.py`) exists in your install directory. (Note that it will not be detected inside a ZIP file, but a correctly named ZIP file will be detected instead.)

These will ensure that the files in a system-wide installation will not take precedence over the copy of the standard library bundled with your application. Otherwise, your users may experience problems using your application. Note that the first suggestion is the best, as the others may still be susceptible to non-standard paths in the registry and user site-packages.

Changed in version 3.6: Add `._pth` file support and removes `applocal` option from `pyvenv.cfg`.

Changed in version 3.6: Add `python`*`XX`*`.zip` as a potential landmark when directly adjacent to the executable.

Deprecated since version 3.6: Modules specified in the registry under `Modules` (not `PythonPath`) may be imported by **`importlib.machinery.WindowsRegistryFinder`**. This finder is enabled on Windows in 3.6.0 and earlier, but may need to be explicitly added to **`sys.meta_path`** in the future.

4.9. Additional modules

Even though Python aims to be portable among all platforms, there are features that are unique to Windows. A couple of modules, both in the standard library and external, and snippets exist to use these features.

The Windows-specific standard modules are documented in MS Windows Specific Services.

4.9.1. PyWin32

The PyWin32 [https://pypi.org/project/PyWin32/] module by Mark Hammond is a collection of modules for advanced Windows-specific support. This includes utilities for:

- Component Object Model [https://learn.microsoft.com/windows/win32/com/component-object-model--com--portal] (COM)
- Win32 API calls
- Registry
- Event log
- Microsoft Foundation Classes [https://learn.microsoft.com/cpp/mfc/mfc-desktop-applications] (MFC) user interfaces

PythonWin [https://web.archive.org/web/20060524042422/https://www.python.org/windows/pythonwin/] is a sample MFC application shipped with PyWin32. It is an embeddable IDE with a built-in debugger.

See also

Win32 How Do I...? [https://timgolden.me.uk/python/win32_how_do_i.html]
by Tim Golden

Python and COM [https://www.boddie.org.uk/python/COM.html]
by David and Paul Boddie

4.9.2. cx_Freeze

cx_Freeze [https://cx-freeze.readthedocs.io/en/latest/] wraps Python scripts into executable Windows programs (`*.exe` files). When you have done this, you can distribute your application without requiring your users to install Python.

4.10. Compiling Python on Windows

If you want to compile CPython yourself, first thing you should do is get the source [https://www.python.org/downloads/source/]. You can download either the latest release's source or just grab a fresh checkout [https://devguide.python.org/setup/#get-the-source-code].

The source tree contains a build solution and project files for Microsoft Visual Studio, which is the compiler used to build the official Python releases. These files are in the `PCbuild` directory.

Check `PCbuild/readme.txt` for general information on the build process.

For extension modules, consult [Building C and C++ Extensions on Windows]().

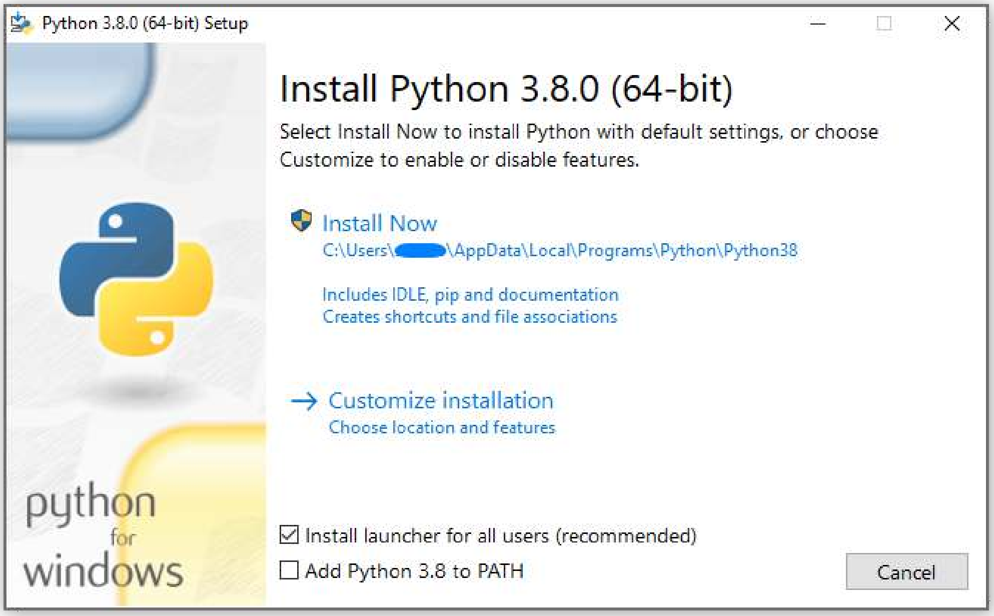
4.11. The full installer (deprecated)

Deprecated since version 3.14: This installer is deprecated since 3.14 and will not be produced for Python 3.16 or later. See [Python install manager]() for the modern installer.

4.11.1. Installation steps

Four Python 3.14 installers are available for download - two each for the 32-bit and 64-bit versions of the interpreter. The *web installer* is a small initial download, and it will automatically download the required components as necessary. The *offline installer* includes the components necessary for a default installation and only requires an internet connection for optional features. See [Installing without downloading]() for other ways to avoid downloading during installation.

After starting the installer, one of two options may be selected:



Install Python 3.8.0 (64-bit)

Select Install Now to install Python with default settings, or choose Customize to enable or disable features.

Install Now
C:\Users\[redacted]\AppData\Local\Programs\Python\Python38

Includes IDLE, pip and documentation
Creates shortcuts and file associations

Customize installation
Choose location and features

☑ Install launcher for all users (recommended)
☐ Add Python 3.8 to PATH

If you select “Install Now”:

- You will *not* need to be an administrator (unless a system update for the C Runtime Library is required or you install the [Python install manager]() for all users)
- Python will be installed into your user directory
- The [Python install manager]() will be installed according to the option at the bottom of the first page
- The standard library, test suite, launcher and pip will be installed
- If selected, the install directory will be added to your `PATH`

- Shortcuts will only be visible for the current user

Selecting “Customize installation” will allow you to select the features to install, the installation location and other options or post-install actions. To install debugging symbols or binaries, you will need to use this option.

To perform an all-users installation, you should select “Customize installation”. In this case:

- You may be required to provide administrative credentials or approval
- Python will be installed into the Program Files directory
- The Python install manager will be installed into the Windows directory
- Optional features may be selected during installation
- The standard library can be pre-compiled to bytecode
- If selected, the install directory will be added to the system **`PATH`**
- Shortcuts are available for all users

4.11.2. Removing the MAX_PATH limitation

Windows historically has limited path lengths to 260 characters. This meant that paths longer than this would not resolve and errors would result.

In the latest versions of Windows, this limitation can be expanded to approximately 32,000 characters. Your administrator will need to activate the “Enable Win32 long paths” group policy, or set `LongPathsEnabled` to `1` in the registry key `HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control\FileSystem`.

This allows the **`open()`** function, the **`os`** module and most other path functionality to accept and return paths longer than 260 characters.

After changing the above option, no further configuration is required.

Changed in version 3.6: Support for long paths was enabled in Python.

4.11.3. Installing without UI

All of the options available in the installer UI can also be specified from the command line, allowing scripted installers to replicate an installation on many machines without user interaction. These options may also be set without suppressing the UI in order to change some of the defaults.

The following options (found by executing the installer with `/?`) can be passed into the installer:

Name	Description
/passive	to display progress without requiring user interaction

Name	Description
/quiet	to install/uninstall without displaying any UI
/simple	to prevent user customization
/uninstall	to remove Python (without confirmation)
/layout [directory]	to pre-download all components
/log [filename]	to specify log files location

All other options are passed as `name=value`, where the value is usually `0` to disable a feature, `1` to enable a feature, or a path. The full list of available options is shown below.

Name	Description	Default
InstallAllUsers	Perform a system-wide installation.	0
TargetDir	The installation directory	Selected based on InstallAllUsers
DefaultAllUsersTargetDir	The default installation directory for all-user installs	`%ProgramFiles%\Python X.Y` or `%ProgramFiles(x86)%\Python X.Y`
DefaultJustForMeTargetDir	The default install directory for just-for-me installs	`%LocalAppData%\Programs\Python\PythonXY` or `%LocalAppData%\Programs\Python\PythonXY-32` or `%LocalAppData%\Programs\Python\PythonXY-64`

Name	Description	Default
DefaultCustomTargetDir	The default custom install directory displayed in the UI	(empty)
AssociateFiles	Create file associations if the launcher is also installed.	1
CompileAll	Compile all `.py` files to `.pyc`.	0
PrependPath	Prepend install and Scripts directories to **`PATH`** and add `.PY` to **`PATHEXT`**	0
AppendPath	Append install and Scripts directories to **`PATH`** and add `.PY` to **`PATHEXT`**	0
Shortcuts	Create shortcuts for the interpreter, documentation and IDLE if installed.	1
Include_doc	Install Python manual	1
Include_debug	Install debug binaries	0

Name	Description	Default
Include_dev	Install developer headers and libraries. Omitting this may lead to an unusable installation.	1
Include_exe	Install `python.exe` and related files. Omitting this may lead to an unusable installation.	1
Include_launcher	Install Python install manager.	1
InstallLauncherAllUsers	Installs the launcher for all users. Also requires `Include_launcher` to be set to 1	1
Include_lib	Install standard library and extension modules. Omitting this may lead to an unusable installation.	1
Include_pip	Install bundled pip and setuptools	1

Name	Description	Default
Include_symbols	Install debugging symbols (`*.pdb`)	0
Include_tcltk	Install Tcl/Tk support and IDLE	1
Include_test	Install standard library test suite	1
Include_tools	Install utility scripts	1
LauncherOnly	Only installs the launcher. This will override most other options.	0
SimpleInstall	Disable most install UI	0
SimpleInstallDescription	A custom message to display when the simplified install UI is used.	(empty)

For example, to silently install a default, system-wide Python installation, you could use the following command (from an elevated command prompt):

```
python-3.9.0.exe /quiet InstallAllUsers=1 PrependPath=1 Include_test=0
```

To allow users to easily install a personal copy of Python without the test suite, you could provide a shortcut with the following command. This will display a simplified initial page and disallow customization:

```
python-3.9.0.exe InstallAllUsers=0 Include_launcher=0 Include_test=0
    SimpleInstall=1 SimpleInstallDescription="Just for me, no test suite."
```

(Note that omitting the launcher also omits file associations, and is only recommended for per-user installs when there is also a system-wide installation that included the launcher.)

The options listed above can also be provided in a file named `unattend.xml` alongside the executable. This file specifies a list of options and values. When a value is provided as an attribute, it will be converted to a number if possible. Values provided as element text are always left as strings. This example file sets the same options as the previous example:

```
<Options>
    <Option Name="InstallAllUsers" Value="no" />
    <Option Name="Include_launcher" Value="0" />
    <Option Name="Include_test" Value="no" />
    <Option Name="SimpleInstall" Value="yes" />
    <Option Name="SimpleInstallDescription">Just for me, no test suite</Option>
</Options>
```

4.11.4. Installing without downloading

As some features of Python are not included in the initial installer download, selecting those features may require an internet connection. To avoid this need, all possible components may be downloaded on-demand to create a complete *layout* that will no longer require an internet connection regardless of the selected features. Note that this download may be bigger than required, but where a large number of installations are going to be performed it is very useful to have a locally cached copy.

Execute the following command from Command Prompt to download all possible required files. Remember to substitute `python-3.9.0.exe` for the actual name of your installer, and to create layouts in their own directories to avoid collisions between files with the same name.

```
python-3.9.0.exe /layout [optional target directory]
```

You may also specify the `/quiet` option to hide the progress display.

4.11.5. Modifying an install

Once Python has been installed, you can add or remove features through the Programs and Features tool that is part of Windows. Select the Python entry and choose “Uninstall/Change” to open the installer in maintenance mode.

“Modify” allows you to add or remove features by modifying the checkboxes - unchanged checkboxes will not install or remove anything. Some options cannot be changed in this mode, such as the install directory; to modify these, you will need to remove and then reinstall Python completely.

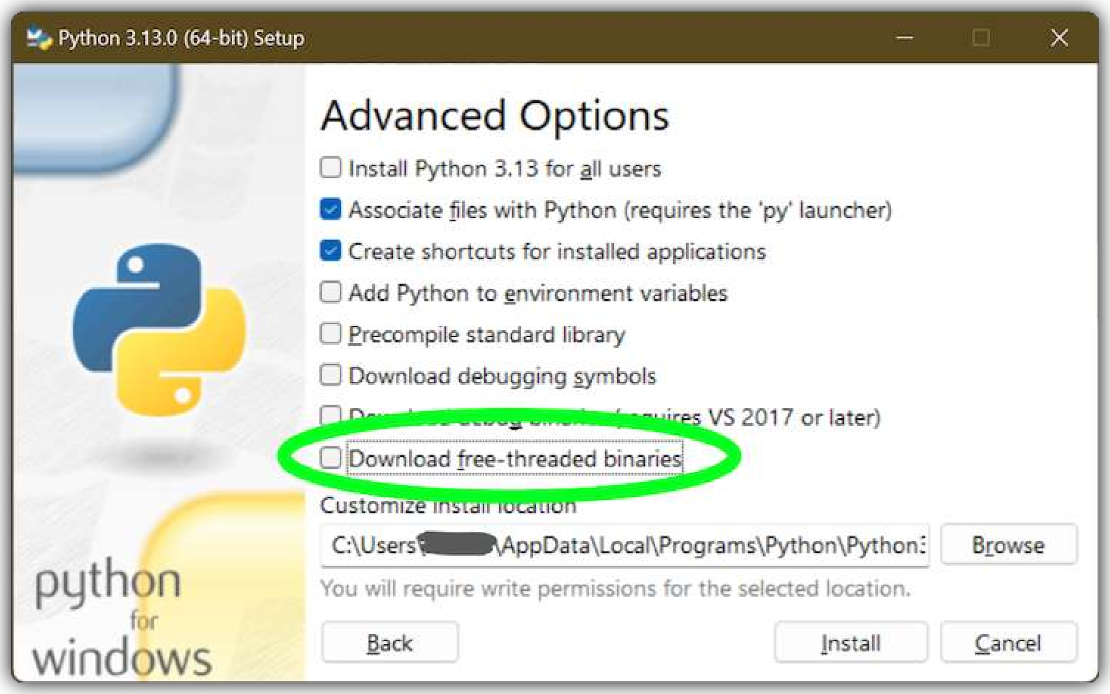
“Repair” will verify all the files that should be installed using the current settings and replace any that have been removed or modified.

“Uninstall” will remove Python entirely, with the exception of the Python install manager, which has its own entry in Programs and Features.

4.11.6. Installing free-threaded binaries

Added in version 3.13.

To install pre-built binaries with free-threading enabled (see **PEP 703** [https://peps.python.org/pep-0703/]), you should select “Customize installation”. The second page of options includes the “Download free-threaded binaries” checkbox.



Advanced Options

☐ Install Python 3.13 for all users
☑ Associate files with Python (requires the 'py' launcher)
☑ Create shortcuts for installed applications
☐ Add Python to environment variables
☐ Precompile standard library
☐ Download debugging symbols
☐ Download debug binaries (requires VS 2017 or later)
☐ Download free-threaded binaries

Customize install location
C:\Users\[redacted]\AppData\Local\Programs\Python\Python
You will require write permissions for the selected location.

Selecting this option will download and install additional binaries to the same location as the main Python install. The main executable is called `python3.13t.exe`, and other binaries either receive a `t` suffix or a full ABI suffix. Python source files and bundled third-party dependencies are shared with the main install.

The free-threaded version is registered as a regular Python install with the tag `3.13t` (with a `-32` or `-arm64` suffix as normal for those platforms). This allows tools to discover it, and for the Python install manager to support `py.exe -3.13t`. Note that the launcher will interpret `py.exe -3` (or a `python3` shebang) as “the latest 3.x install”, which will prefer the free-threaded binaries over the regular ones, while `py.exe -3.13` will not. If you use the short style of option, you may prefer to not install the free-threaded binaries at this time.

To specify the install option at the command line, use `Include_freethreaded=1`. See Installing without downloading for instructions on pre-emptively downloading the additional binaries for offline install. The options to include debug symbols and binaries also apply to the free-threaded builds.

Free-threaded binaries are also available on nuget.org.

4.12. Python launcher for Windows (deprecated)

Deprecated since version 3.14: The launcher and this documentation have been superseded by the Python Install Manager described above. This is preserved temporarily for historical interest.

Added in version 3.3.

The Python launcher for Windows is a utility which aids in locating and executing of different Python versions. It allows scripts (or the command-line) to indicate a preference for a specific Python version, and will locate and execute that version.

Unlike the **PATH** variable, the launcher will correctly select the most appropriate version of Python. It will prefer per-user installations over system-wide ones, and orders by language version rather than using the most recently installed version.

The launcher was originally specified in **PEP 397** [https://peps.python.org/pep-0397/].

4.12.1. Getting started

4.12.1.1. From the command-line

Changed in version 3.6.

System-wide installations of Python 3.3 and later will put the launcher on your **PATH**. The launcher is compatible with all available versions of Python, so it does not matter which version is installed. To check that the launcher is available, execute the following command in Command Prompt:

```
py
```

You should find that the latest version of Python you have installed is started - it can be exited as normal, and any additional command-line arguments specified will be sent directly to Python.

If you have multiple versions of Python installed (e.g., 3.7 and 3.14) you will have noticed that Python 3.14 was started - to launch Python 3.7, try the command:

```
py -3.7
```

If you want the latest version of Python 2 you have installed, try the command:

```
py -2
```

If you see the following error, you do not have the launcher installed:

```
'py' is not recognized as an internal or external command,
operable program or batch file.
```

The command:

```
py --list
```

displays the currently installed version(s) of Python.

The `-x.y` argument is the short form of the `-V:Company/Tag` argument, which allows selecting a specific Python runtime, including those that may have come from somewhere other than python.org. Any runtime registered by following **PEP 514** [https://peps.python.org/pep-0514/] will be discoverable. The `--list` command lists all available runtimes using the `-V:` format.

When using the `-V:` argument, specifying the Company will limit selection to runtimes from that provider, while specifying only the Tag will select from all providers. Note that omitting the slash implies a tag:

```
# Select any '3.*' tagged runtime
py -V:3

# Select any 'PythonCore' released runtime
py -V:PythonCore/

# Select PythonCore's latest Python 3 runtime
py -V:PythonCore/3
```

The short form of the argument (`-3`) only ever selects from core Python releases, and not other distributions. However, the longer form (`-V:3`) will select from any.

The Company is matched on the full string, case-insensitive. The Tag is matched on either the full string, or a prefix, provided the next character is a dot or a hyphen. This allows `-V:3.1` to match `3.1-32`, but not `3.10`. Tags are sorted using numerical ordering (`3.10` is newer than `3.1`), but are compared using text (`-V:3.01` does not match `3.1`).

4.12.1.2. Virtual environments

Added in version 3.5.

If the launcher is run with no explicit Python version specification, and a virtual environment (created with the standard library `venv` module or the external `virtualenv` tool) active, the launcher will run the virtual environment’s interpreter rather than the global one. To run the global interpreter, either deactivate the virtual environment, or explicitly specify the global Python version.

4.12.1.3. From a script

Let’s create a test Python script - create a file called `hello.py` with the following contents

```
#! python
import sys
sys.stdout.write("hello from Python %s\n" % (sys.version,))
```

From the directory in which hello.py lives, execute the command:

```
py hello.py
```

You should notice the version number of your latest Python 2.x installation is printed. Now try changing the first line to be:

```
#! python3
```

Re-executing the command should now print the latest Python 3.x information. As with the above command-line examples, you can specify a more explicit version qualifier. Assuming you have Python 3.7 installed, try changing the first line to `#! python3.7` and you should find the 3.7 version information printed.

Note that unlike interactive use, a bare “python” will use the latest version of Python 2.x that you have installed. This is for backward compatibility and for compatibility with Unix, where the command `python` typically refers to Python 2.

4.12.1.4. From file associations

The launcher should have been associated with Python files (i.e. `.py`, `.pyw`, `.pyc` files) when it was installed. This means that when you double-click on one of these files from Windows explorer the launcher will be used, and therefore you can use the same facilities described above to have the script specify the version which should be used.

The key benefit of this is that a single launcher can support multiple Python versions at the same time depending on the contents of the first line.

4.12.2. Shebang lines

If the first line of a script file starts with `#!`, it is known as a “shebang” line. Linux and other Unix like operating systems have native support for such lines and they are commonly used on such systems to indicate how a script should be executed. This launcher allows the same facilities to be used with Python scripts on Windows and the examples above demonstrate their use.

To allow shebang lines in Python scripts to be portable between Unix and Windows, this launcher supports a number of ‘virtual’ commands to specify which interpreter to use. The supported virtual commands are:

- `/usr/bin/env`

- `/usr/bin/python`
- `/usr/local/bin/python`
- `python`

For example, if the first line of your script starts with

```
#! /usr/bin/python
```

The default Python or an active virtual environment will be located and used. As many Python scripts written to work on Unix will already have this line, you should find these scripts can be used by the launcher without modification. If you are writing a new script on Windows which you hope will be useful on Unix, you should use one of the shebang lines starting with `/usr`.

Any of the above virtual commands can be suffixed with an explicit version (either just the major version, or the major and minor version). Furthermore the 32-bit version can be requested by adding “-32” after the minor version. I.e. `/usr/bin/python3.7-32` will request usage of the 32-bit Python 3.7. If a virtual environment is active, the version will be ignored and the environment will be used.

Added in version 3.7: Beginning with python launcher 3.7 it is possible to request 64-bit version by the “-64” suffix. Furthermore it is possible to specify a major and architecture without minor (i.e. `/usr/bin/python3-64`).

Changed in version 3.11: The “-64” suffix is deprecated, and now implies “any architecture that is not provably i386/32-bit”. To request a specific environment, use the new `-V:TAG` argument with the complete tag.

Changed in version 3.13: Virtual commands referencing `python` now prefer an active virtual environment rather than searching **`PATH`**. This handles cases where the shebang specifies `/usr/bin/env python3` but `python3.exe` is not present in the active environment.

The `/usr/bin/env` form of shebang line has one further special property. Before looking for installed Python interpreters, this form will search the executable **`PATH`** for a Python executable matching the name provided as the first argument. This corresponds to the behaviour of the Unix `env` program, which performs a **`PATH`** search. If an executable matching the first argument after the `env` command cannot be found, but the argument starts with `python`, it will be handled as described for the other virtual commands. The environment variable **`PYLAUNCHER_NO_SEARCH_PATH`** may be set (to any value) to skip this search of **`PATH`**.

Shebang lines that do not match any of these patterns are looked up in the `[commands]` section of the launcher’s .INI file. This may be used to handle certain commands in a way that makes sense for your system. The name of the command must be a single argument (no spaces in the shebang executable), and the value substituted is the full path to the executable (additional arguments specified in the .INI will be quoted as part of the filename).

```
[commands]
/bin/xpython=C:\Program Files\XPython\python.exe
```

Any commands not found in the .INI file are treated as **Windows** executable paths that are absolute or relative to the directory containing the script file. This is a convenience for Windows-only scripts, such as those generated by an installer, since the behavior is not compatible with Unix-style shells. These paths may be quoted, and may include multiple arguments, after which the path to the script and any additional arguments will be appended.

4.12.3. Arguments in shebang lines

The shebang lines can also specify additional options to be passed to the Python interpreter. For example, if you have a shebang line:

```
#! /usr/bin/python -v
```

Then Python will be started with the `-v` option

4.12.4. Customization

4.12.4.1. Customization via INI files

Two .ini files will be searched by the launcher - `py.ini` in the current user's application data directory (`%LOCALAPPDATA%` or `$env:LocalAppData`) and `py.ini` in the same directory as the launcher. The same .ini files are used for both the 'console' version of the launcher (i.e. py.exe) and for the 'windows' version (i.e. pyw.exe).

Customization specified in the "application directory" will have precedence over the one next to the executable, so a user, who may not have write access to the .ini file next to the launcher, can override commands in that global .ini file.

4.12.4.2. Customizing default Python versions

In some cases, a version qualifier can be included in a command to dictate which version of Python will be used by the command. A version qualifier starts with a major version number and can optionally be followed by a period ('.') and a minor version specifier. Furthermore it is possible to specify if a 32 or 64 bit implementation shall be requested by adding "-32" or "-64".

For example, a shebang line of `#!python` has no version qualifier, while `#!python3` has a version qualifier which specifies only a major version.

If no version qualifiers are found in a command, the environment variable **`PY_PYTHON`** can be set to specify the default version qualifier. If it is not set, the default is "3". The variable can specify any value that may be passed on the command line, such as "3", "3.7", "3.7-32" or "3.7-64". (Note that the "-64" option is only available with the launcher included with Python 3.7 or newer.)

If no minor version qualifiers are found, the environment variable `PY_PYTHON{major}` (where `{major}` is the current major version qualifier as determined above) can be set to specify the full version. If no such option is found, the launcher will enumerate the installed Python versions and use the latest minor release found for the major version, which is likely, although not guaranteed, to be the most recently installed version in that family.

On 64-bit Windows with both 32-bit and 64-bit implementations of the same (major.minor) Python version installed, the 64-bit version will always be preferred. This will be true for both 32-bit and 64-bit implementations of the launcher - a 32-bit launcher will prefer to execute a 64-bit Python installation of the specified version if available. This is so the behavior of the launcher can be predicted knowing only what versions are installed on the PC and without regard to the order in which they were installed (i.e., without knowing whether a 32 or 64-bit version of Python and corresponding launcher was installed last). As noted above, an optional “-32” or “-64” suffix can be used on a version specifier to change this behaviour.

Examples:

- If no relevant options are set, the commands `python` and `python2` will use the latest Python 2.x version installed and the command `python3` will use the latest Python 3.x installed.
- The command `python3.7` will not consult any options at all as the versions are fully specified.
- If `PY_PYTHON=3`, the commands `python` and `python3` will both use the latest installed Python 3 version.
- If `PY_PYTHON=3.7-32`, the command `python` will use the 32-bit implementation of 3.7 whereas the command `python3` will use the latest installed Python (PY_PYTHON was not considered at all as a major version was specified.)
- If `PY_PYTHON=3` and `PY_PYTHON3=3.7`, the commands `python` and `python3` will both use specifically 3.7

In addition to environment variables, the same settings can be configured in the .INI file used by the launcher. The section in the INI file is called `[defaults]` and the key name will be the same as the environment variables without the leading `PY_` prefix (and note that the key names in the INI file are case insensitive.) The contents of an environment variable will override things specified in the INI file.

For example:

- Setting `PY_PYTHON=3.7` is equivalent to the INI file containing:

```
[defaults]
python=3.7
```

- Setting `PY_PYTHON=3` and `PY_PYTHON3=3.7` is equivalent to the INI file containing:

```
[defaults]
python=3
python3=3.7
```

4.12.5. Diagnostics

If an environment variable **`PYLAUNCHER_DEBUG`** is set (to any value), the launcher will print diagnostic information to stderr (i.e. to the console). While this information manages to be simultaneously verbose *and* terse, it should allow you to see what versions of Python were located, why a particular version was chosen and the exact command-line used to execute the target Python. It is primarily intended for testing and debugging.

4.12.6. Dry run

If an environment variable **`PYLAUNCHER_DRYRUN`** is set (to any value), the launcher will output the command it would have run, but will not actually launch Python. This may be useful for tools that want to use the launcher to detect and then launch Python directly. Note that the command written to standard output is always encoded using UTF-8, and may not render correctly in the console.

4.12.7. Install on demand

If an environment variable **`PYLAUNCHER_ALLOW_INSTALL`** is set (to any value), and the requested Python version is not installed but is available on the Microsoft Store, the launcher will attempt to install it. This may require user interaction to complete, and you may need to run the command again.

An additional **`PYLAUNCHER_ALWAYS_INSTALL`** variable causes the launcher to always try to install Python, even if it is detected. This is mainly intended for testing (and should be used with **`PYLAUNCHER_DRYRUN`**).

4.12.8. Return codes

The following exit codes may be returned by the Python launcher. Unfortunately, there is no way to distinguish these from the exit code of Python itself.

The names of codes are as used in the sources, and are only for reference. There is no way to access or resolve them apart from reading this page. Entries are listed in alphabetical order of names.

Name	Value	Description
RC_BAD_VENV_CFG	107	A `pyvenv.cfg` was found but is corrupt.
RC_CREATE_PROCESS	101	Failed to launch Python.
RC_INSTALLING	111	An install was started, but the command will need to be re-run after it completes.

Name	Value	Description
RC_INTERNAL_ERROR	109	Unexpected error. Please report a bug.
RC_NO_COMMANDLINE	108	Unable to obtain command line from the operating system.
RC_NO_PYTHON	103	Unable to locate the requested version.
RC_NO_VENV_CFG	106	A `pyvenv.cfg` was required but not found.

5. Using Python on macOS

This document aims to give an overview of macOS-specific behavior you should know about to get started with Python on Mac computers. Python on a Mac running macOS is very similar to Python on other Unix-derived platforms, but there are some differences in installation and some features.

There are various ways to obtain and install Python for macOS. Pre-built versions of the most recent versions of Python are available from a number of distributors. Much of this document describes use of the Pythons provided by the CPython release team for download from the python.org website [https://www.python.org/downloads/]. See Alternative Distributions for some other options.

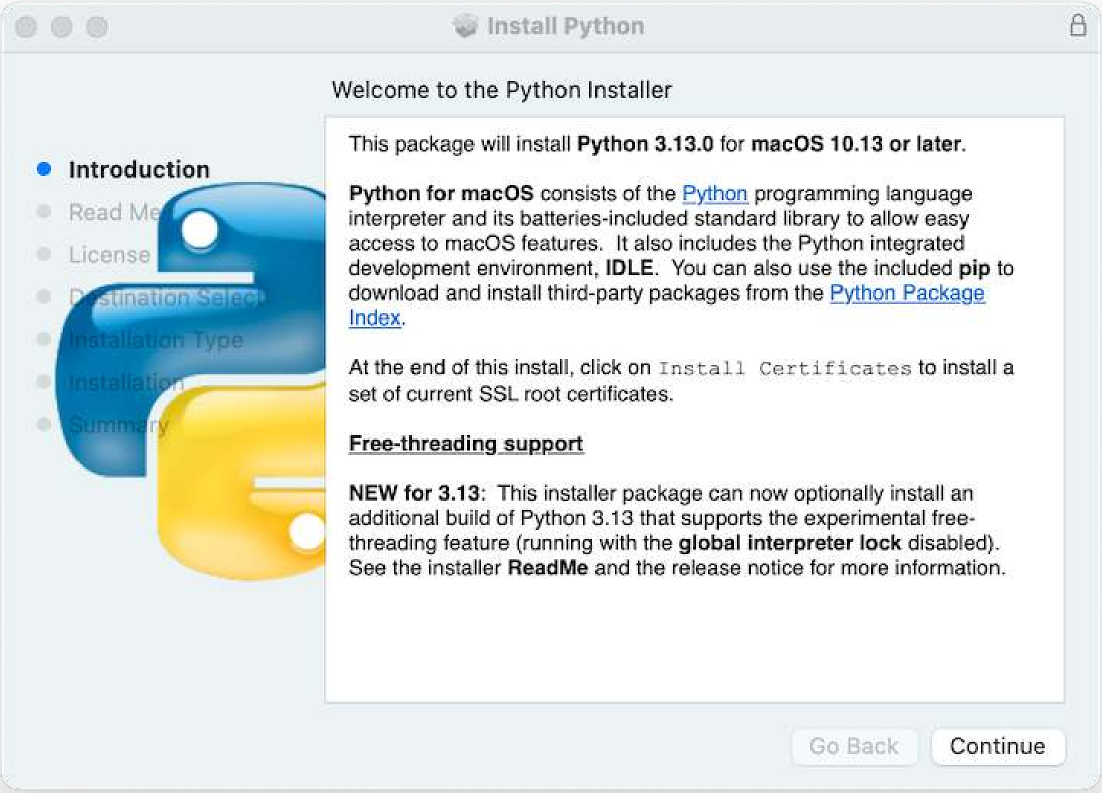
5.1. Using Python for macOS from `python.org`

5.1.1. Installation steps

For current Python versions [https://www.python.org/downloads/] (other than those in `security` status), the release team produces a **Python for macOS** installer package for each new release. A list of available installers is available here [https://www.python.org/downloads/macos/]. We recommend using the most recent supported Python version where possible. Current installers provide a universal2 binary [https://en.wikipedia.org/wiki/Universal_binary] build of Python which runs natively on all Macs (Apple Silicon and Intel) that are supported by a wide range of macOS versions, currently typically from at least **macOS 10.15 Catalina** on.

The downloaded file is a standard macOS installer package file (`.pkg`). File integrity information (checksum, size, sigstore signature, etc) for each file is included on the release download page. Installer packages and their contents are signed and notarized with `Python Software Foundation` Apple Developer ID certificates to meet macOS Gatekeeper requirements [https://support.apple.com/en-us/102445].

For a default installation, double-click on the downloaded installer package file. This should launch the standard macOS Installer app and display the first of several installer windows steps.



Welcome to the Python Installer

This package will install **Python 3.13.0** for **macOS 10.13 or later.**

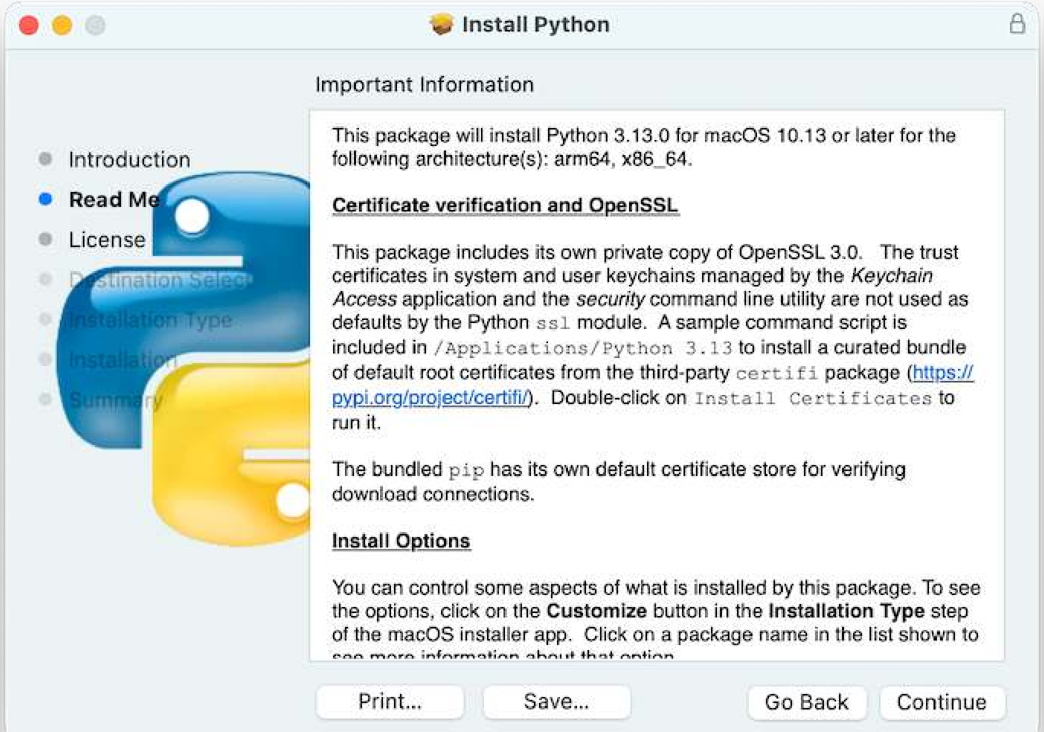
Python for macOS consists of the Python programming language interpreter and its batteries-included standard library to allow easy access to macOS features. It also includes the Python integrated development environment, **IDLE**. You can also use the included **pip** to download and install third-party packages from the Python Package Index.

At the end of this install, click on `Install Certificates` to install a set of current SSL root certificates.

Free-threading support

NEW for 3.13: This installer package can now optionally install an additional build of Python 3.13 that supports the experimental free-threading feature (running with the **global interpreter lock** disabled). See the installer **ReadMe** and the release notice for more information.

Clicking on the **Continue** button brings up the **Read Me** for this installer. Besides other important information, the **Read Me** documents which Python version is going to be installed and on what versions of macOS it is supported. You may need to scroll through to read the whole file. By default, this **Read Me** will also be installed in `/Applications/Python 3.14/` and available to read anytime.



Important Information

This package will install Python 3.13.0 for macOS 10.13 or later for the following architecture(s): arm64, x86_64.

Certificate verification and OpenSSL

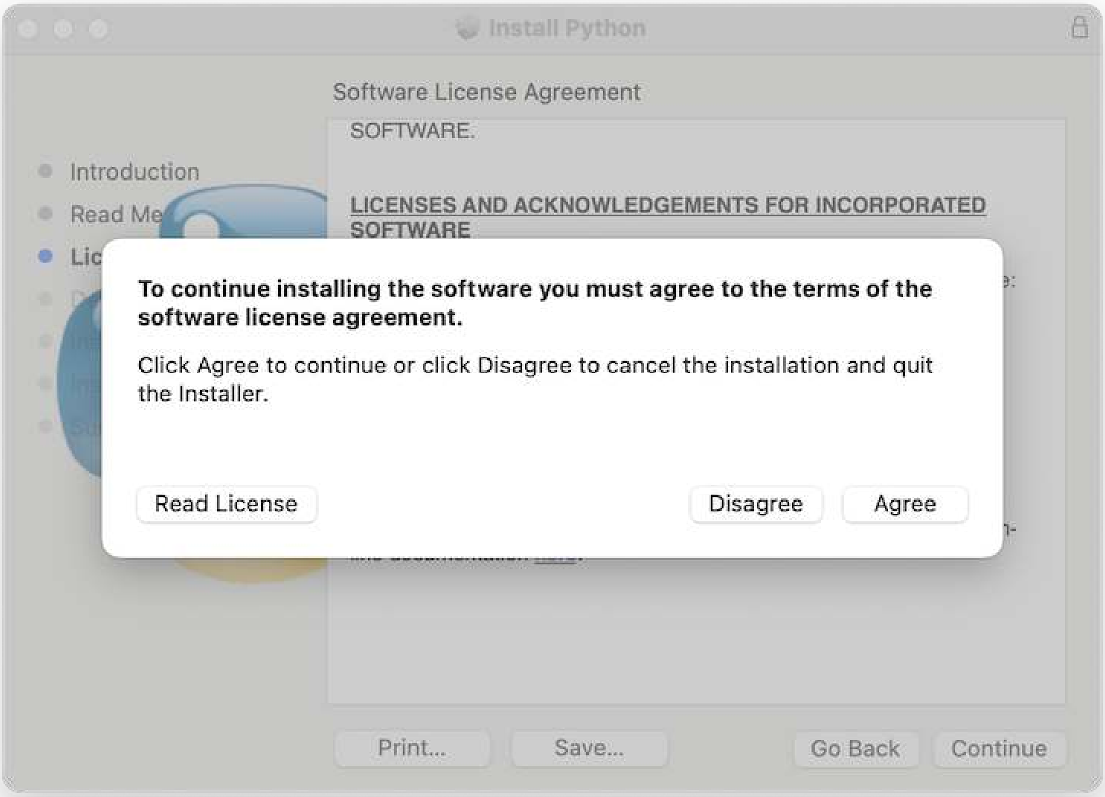
This package includes its own private copy of OpenSSL 3.0. The trust certificates in system and user keychains managed by the *Keychain Access* application and the *security* command line utility are not used as defaults by the Python `ssl` module. A sample command script is included in `/Applications/Python 3.13` to install a curated bundle of default root certificates from the third-party `certifi` package (https://pypi.org/project/certifi/). Double-click on `Install Certificates` to run it.

The bundled `pip` has its own default certificate store for verifying download connections.

Install Options

You can control some aspects of what is installed by this package. To see the options, click on the **Customize** button in the **Installation Type** step of the macOS installer app. Click on a package name in the list shown to see more information about that option.

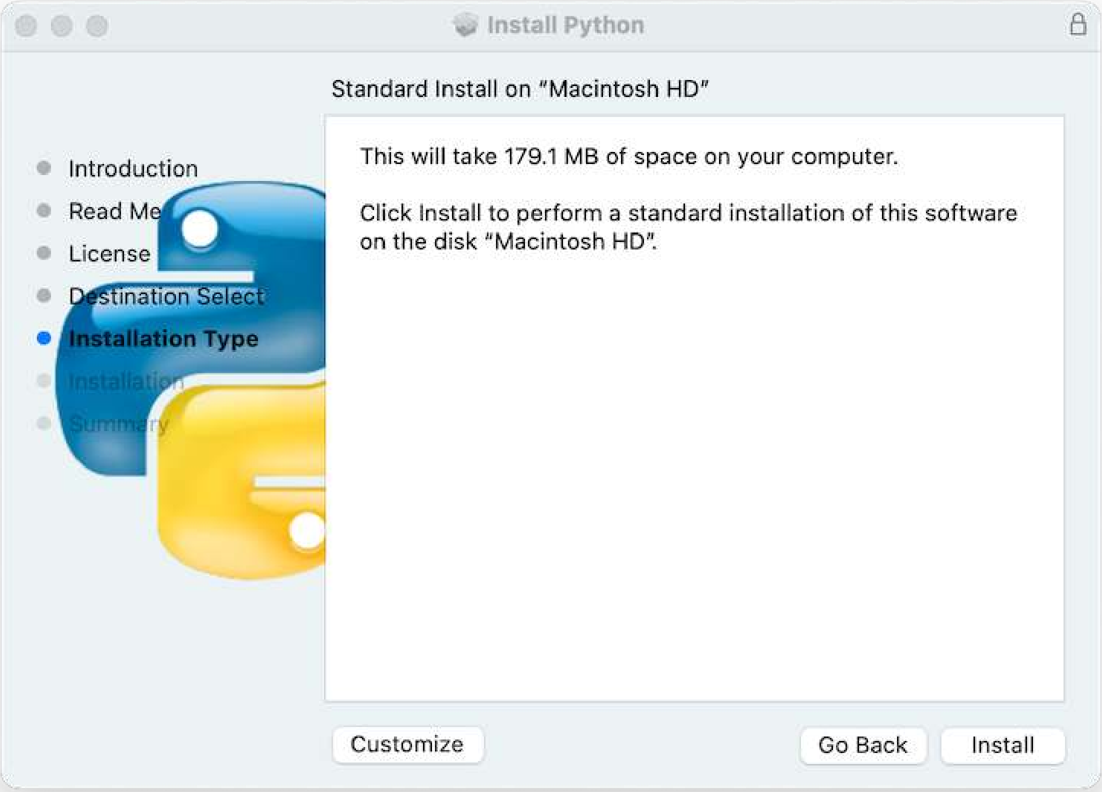
Clicking on **Continue** proceeds to display the license for Python and for other included software. You will then need to **Agree** to the license terms before proceeding to the next step. This license file will also be installed and available to be read later.


Install Python
Introduction
Read Me
Software License Agreement
SOFTWARE.
LICENSES AND ACKNOWLEDGEMENTS FOR INCORPORATED SOFTWARE
Read License
Disagree
Agree
Print...
Save...
Go Back
Continue

To continue installing the software you must agree to the terms of the software license agreement.

Click Agree to continue or click Disagree to cancel the installation and quit the Installer.

After the license terms are accepted, the next step is the **Installation Type** display. For most uses, the standard set of installation operations is appropriate.

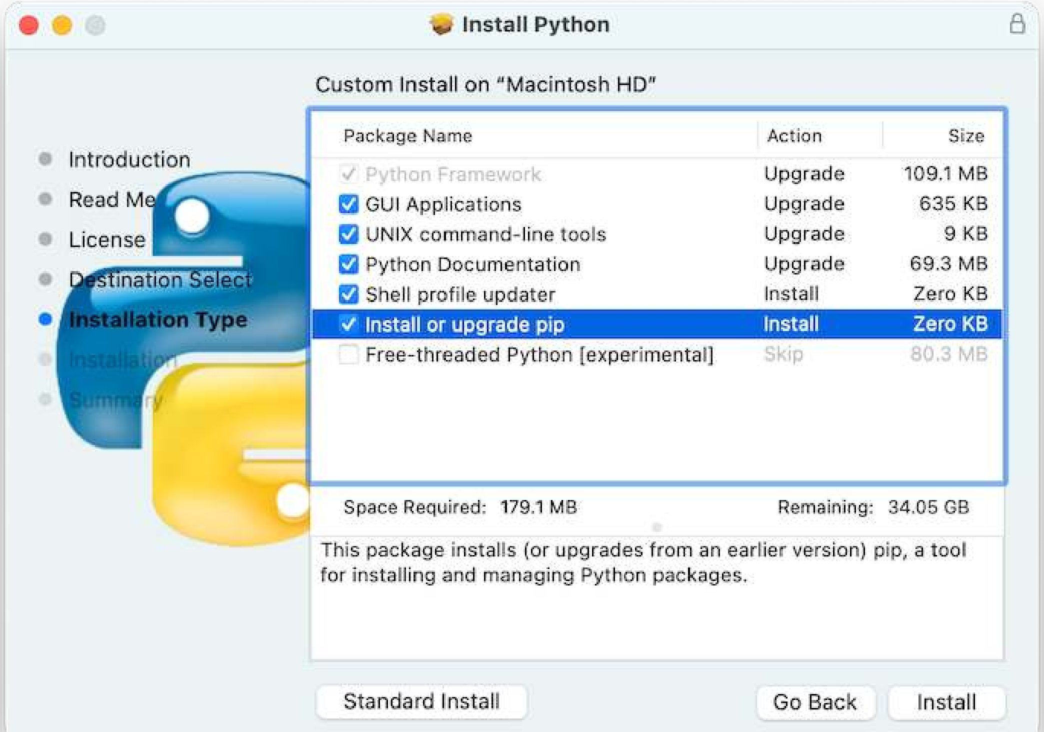

Install Python
Introduction
Read Me
License
Destination Select
Installation Type
Installation
Summary
Customize
Go Back
Install

Standard Install on "Macintosh HD"

This will take 179.1 MB of space on your computer.

Click Install to perform a standard installation of this software on the disk "Macintosh HD".

By pressing the **Customize** button, you can choose to omit or select certain package components of the installer. Click on each package name to see a description of what it installs. To also install support for the optional free-threaded feature, see Installing Free-threaded Binaries.


Install Python
Introduction
Read Me
License
Destination Select
Installation Type
Installation
Summary
Standard Install
Go Back
Install

Custom Install on “Macintosh HD”

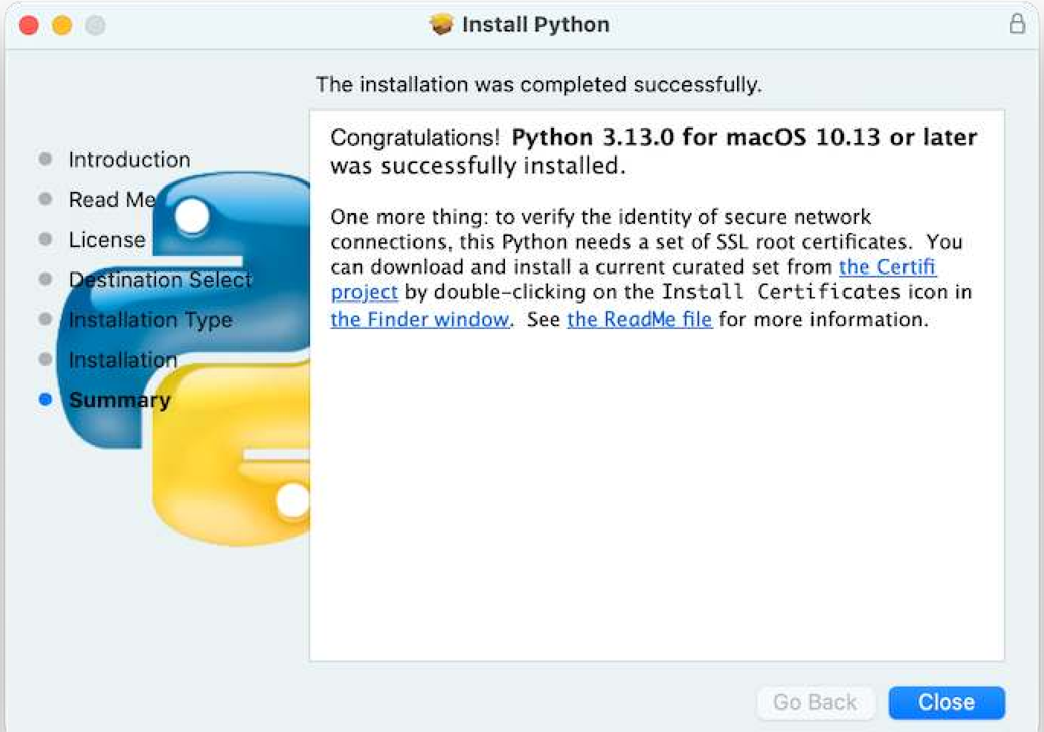
Package Name	Action	Size
☑ Python Framework	Upgrade	109.1 MB
☑ GUI Applications	Upgrade	635 KB
☑ UNIX command-line tools	Upgrade	9 KB
☑ Python Documentation	Upgrade	69.3 MB
☑ Shell profile updater	Install	Zero KB
☑ Install or upgrade pip	Install	Zero KB
☐ Free-threaded Python [experimental]	Skip	80.3 MB

Space Required: 179.1 MB Remaining: 34.05 GB

This package installs (or upgrades from an earlier version) pip, a tool for installing and managing Python packages.

In either case, clicking **Install** will begin the install process by asking permission to install new software. A macOS user name with `Administrator` privilege is needed as the installed Python will be available to all users of the Mac.

When the installation is complete, the **Summary** window will appear.

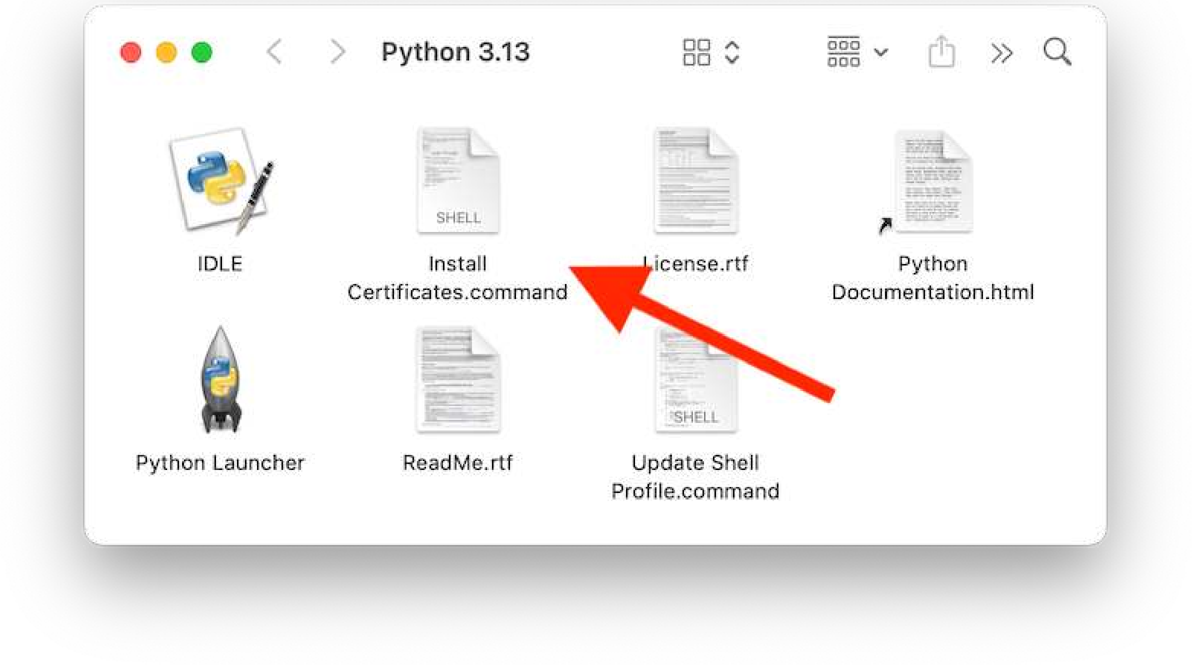

Install Python
Introduction
Read Me
License
Destination Select
Installation Type
Installation
Summary
Go Back
Close

The installation was completed successfully.

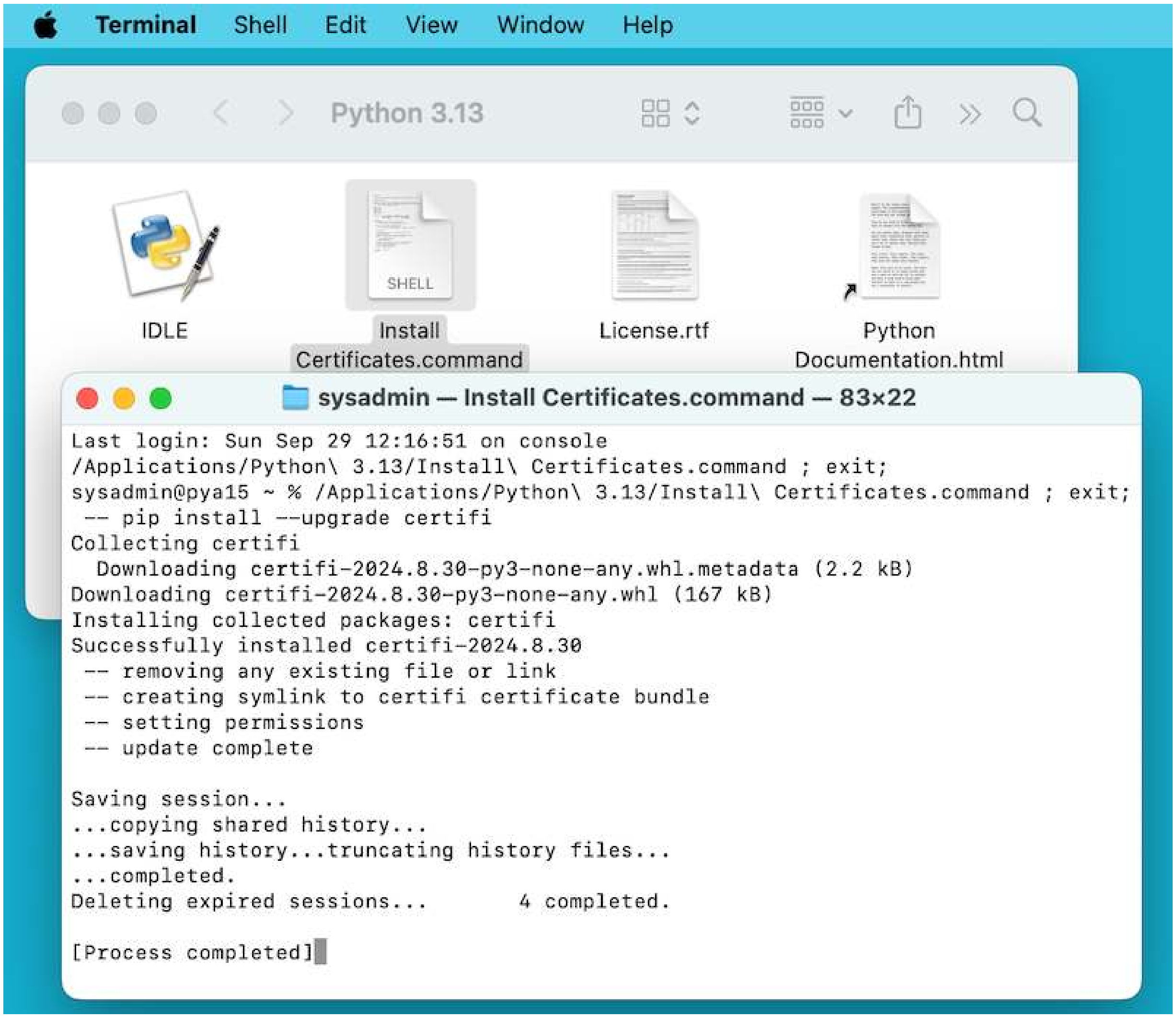
Congratulations! **Python 3.13.0 for macOS 10.13 or later** was successfully installed.

One more thing: to verify the identity of secure network connections, this Python needs a set of SSL root certificates. You can download and install a current curated set from the Certifi project by double-clicking on the `Install Certificates` icon in the Finder window. See the ReadMe file for more information.

Double-click on the **Install Certificates.command** icon or file in the `/Applications/Python 3.14/` window to complete the installation.


Python 3.13
IDLE
SHELL
Install Certificates.command
License.rtf
Python Documentation.html
Python Launcher
ReadMe.rtf
SHELL
Update Shell Profile.command

This will open a temporary **Terminal** shell window that will use the new Python to download and install SSL root certificates for its use.


Terminal
Shell
Edit
View
Window
Help
Python 3.13
IDLE
SHELL
Install Certificates.command
License.rtf
Python Documentation.html
sysadmin — Install Certificates.command — 83×22

```
Last login: Sun Sep 29 12:16:51 on console
/Applications/Python\ 3.13/Install\ Certificates.command ; exit;
sysadmin@pya15 ~ % /Applications/Python\ 3.13/Install\ Certificates.command ; exit;
 -- pip install --upgrade certifi
Collecting certifi
  Downloading certifi-2024.8.30-py3-none-any.whl.metadata (2.2 kB)
Downloading certifi-2024.8.30-py3-none-any.whl (167 kB)
Installing collected packages: certifi
Successfully installed certifi-2024.8.30
 -- removing any existing file or link
 -- creating symlink to certifi certificate bundle
 -- setting permissions
 -- update complete

Saving session...
...copying shared history...
...saving history...truncating history files...
...completed.
Deleting expired sessions...       4 completed.

[Process completed]
```

If `Successfully installed certifi` and `update complete` appears in the terminal window, the installation is complete. Close this terminal window and the installer window.

A default install will include:

- A `Python 3.14` folder in your `Applications` folder. In here you find **IDLE**, the development environment that is a standard part of official Python distributions; and **Python Launcher**, which handles double-clicking Python scripts from the macOS Finder [https://support.apple.com/en-us/HT201732].

- A framework `/Library/Frameworks/Python.framework`, which includes the Python executable and libraries. The installer adds this location to your shell path. To uninstall Python, you can remove these three things. Symlinks to the Python executable are placed in `/usr/local/bin/`.

Note

Recent versions of macOS include a **python3** command in `/usr/bin/python3` that links to a usually older and incomplete version of Python provided by and for use by the Apple development tools, **Xcode** or the **Command Line Tools for Xcode**. You should never modify or attempt to delete this installation, as it is Apple-controlled and is used by Apple-provided or third-party software. If you choose to install a newer Python version from `python.org`, you will have two different but functional Python installations on your computer that can co-exist. The default installer options should ensure that its **python3** will be used instead of the system **python3**.

5.1.2. How to run a Python script

There are two ways to invoke the Python interpreter. If you are familiar with using a Unix shell in a terminal window, you can invoke `python3.14` or `python3` optionally followed by one or more command line options (described in Command line and environment). The Python tutorial also has a useful section on using Python interactively from a shell.

You can also invoke the interpreter through an integrated development environment. IDLE — Python editor and shell is a basic editor and interpreter environment which is included with the standard distribution of Python. **IDLE** includes a

Help menu that allows you to access Python documentation. If you are completely new to Python, you can read the tutorial introduction in that document.

There are many other editors and IDEs available, see Editors and IDEs for more information.

To run a Python script file from the terminal window, you can invoke the interpreter with the name of the script file:

```
python3.14 myscript.py
```

To run your script from the Finder, you can either:

- Drag it to **Python Launcher**.
- Select **Python Launcher** as the default application to open your script (or any `.py` script) through the Finder Info window and double-click it. **Python Launcher** has various preferences to control how your script is launched. Option-dragging allows you to change these for one invocation, or use its `Preferences` menu to change things globally.

Be aware that running the script directly from the macOS Finder might produce different results than when running from a terminal window as the script will not be run in the usual shell environment including any setting of environment variables in shell profiles. And, as with any other script or program, be certain of what you are about to run.

5.2. Alternative Distributions

Besides the standard `python.org` for macOS installer, there are third-party distributions for macOS that may include additional functionality. Some popular distributions and their key features:

ActivePython [https://www.activestate.com/products/python/]
Installer with multi-platform compatibility, documentation

Anaconda [https://www.anaconda.com/download/]
Popular scientific modules (such as numpy, scipy, and pandas) and the `conda` package manager.

Homebrew [https://brew.sh]
Package manager for macOS including multiple versions of Python and many third-party Python-based packages (including numpy, scipy, and pandas).

MacPorts [https://www.macports.org]
Another package manager for macOS including multiple versions of Python and many third-party Python-based packages. May include pre-built versions of Python and many packages for older versions of macOS.

Note that distributions might not include the latest versions of Python or other libraries, and are not maintained or supported by the core Python team.

5.3. Installing Additional Python Packages

Refer to the Python Packaging User Guide [https://packaging.python.org/en/latest/tutorials/installing-packages/] for more information.

5.4. GUI Programming

There are several options for building GUI applications on the Mac with Python.

The standard Python GUI toolkit is **`tkinter`**, based on the cross-platform Tk toolkit (https://www.tcl.tk). A macOS-native version of Tk is included with the installer.

PyObjC is a Python binding to Apple's Objective-C/Cocoa framework. Information on PyObjC is available from pyobjc [https://pypi.org/project/pyobjc/].

A number of alternative macOS GUI toolkits are available including:

- PySide [https://www.qt.io/qt-for-python]: Official Python bindings to the Qt GUI toolkit [https://wiki.qt.io/Qt_for_Python].
- PyQt [https://riverbankcomputing.com/software/pyqt/]: Alternative Python bindings to Qt.
- Kivy [https://kivy.org]: A cross-platform GUI toolkit that supports desktop and mobile platforms.
- Toga [https://toga.readthedocs.io]: Part of the BeeWare Project [https://beeware.org]; supports desktop, mobile, web and console apps.
- wxPython [https://wxpython.org]: A cross-platform toolkit that supports desktop operating systems.

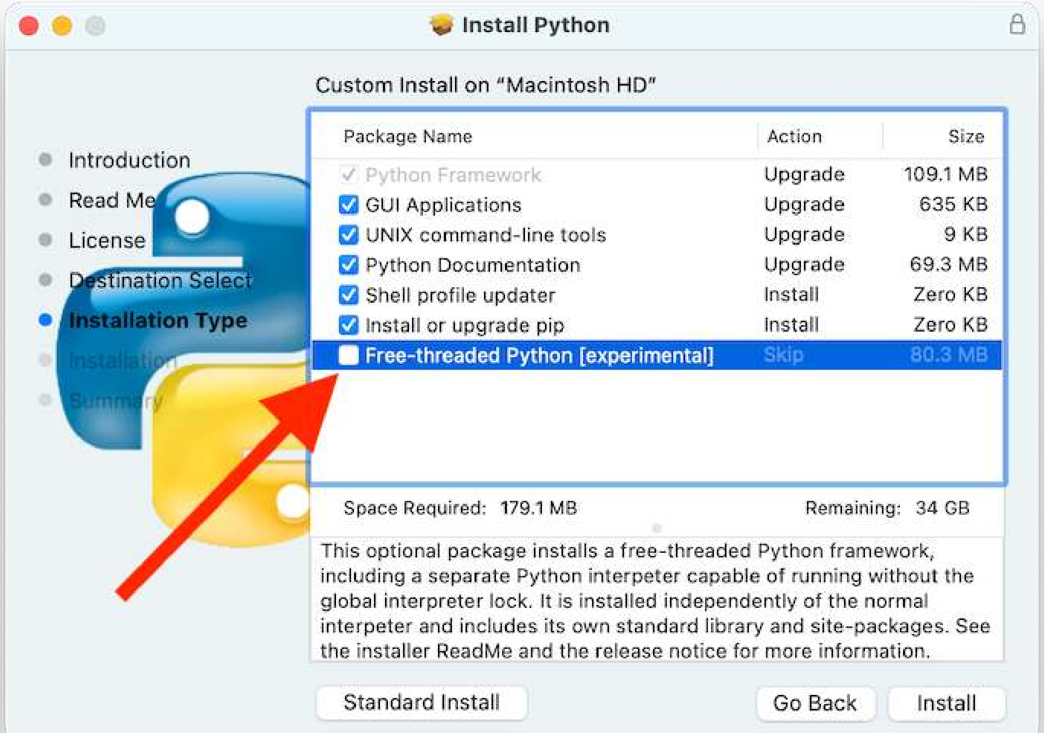
5.5. Advanced Topics

5.5.1. Installing Free-threaded Binaries

Added in version 3.13.

The `python.org` Python for macOS installer package can optionally install an additional build of Python 3.14 that supports **PEP 703** [https://peps.python.org/pep-0703/], the free-threading feature (running with the global interpreter lock disabled). Check the release page on `python.org` for possible updated information.

The free-threaded mode is working and continues to be improved, but there is some additional overhead in single-threaded workloads compared to the regular build. Additionally, third-party packages, in particular ones with an extension module, may not be ready for use in a free-threaded build, and will re-enable the GIL. Therefore, the support for free-threading is not installed by default. It is packaged as a separate install option, available by clicking the **Customize** button on the **Installation Type** step of the installer as described above.



Custom Install on "Macintosh HD"

Package Name	Action	Size
☑ Python Framework	Upgrade	109.1 MB
☑ GUI Applications	Upgrade	635 KB
☑ UNIX command-line tools	Upgrade	9 KB
☑ Python Documentation	Upgrade	69.3 MB
☑ Shell profile updater	Install	Zero KB
☑ Install or upgrade pip	Install	Zero KB
☐ Free-threaded Python [experimental]	Skip	80.3 MB

Space Required: 179.1 MB Remaining: 34 GB

This optional package installs a free-threaded Python framework, including a separate Python interpeter capable of running without the global interpreter lock. It is installed independently of the normal interpeter and includes its own standard library and site-packages. See the installer ReadMe and the release notice for more information.

If the box next to the **Free-threaded Python** package name is checked, a separate `PythonT.framework` will also be installed alongside the normal `Python.framework` in `/Library/Frameworks`. This configuration allows a free-threaded Python 3.14 build to co-exist on your system with a traditional (GIL only) Python 3.14 build with minimal risk while installing or testing. This installation layout may change in future releases.

Known cautions and limitations:

- The **UNIX command-line tools** package, which is selected by default, will install links in `/usr/local/bin` for `python3.14t`, the free-threaded interpreter, and `python3.14t-config`, a configuration utility which may be useful for package builders. Since `/usr/local/bin` is typically included in your shell `PATH`, in most cases no changes to your `PATH` environment variables should be needed to use `python3.14t`.

- For this release, the **Shell profile updater** package and the `Update Shell Profile.command` in `/Applications/Python 3.14/` do not support the free-threaded package.

- The free-threaded build and the traditional build have separate search paths and separate `site-packages` directories so, by default, if you need a package available in both builds, it may need to be installed in both. The free-threaded package will install a separate instance of **pip** for use with `python3.14t`.

  - To install a package using **pip** without a **venv**:

    ```
    python3.14t -m pip install <package_name>
    ```

- When working with multiple Python environments, it is usually safest and easiest to create and use virtual environments. This can avoid possible command name conflicts and confusion about which Python is in use:

  ```
  python3.14t -m venv <venv_name>
  ```

  then **activate**.

- To run a free-threaded version of IDLE:

  ```
  python3.14t -m idlelib
  ```

- The interpreters in both builds respond to the same PYTHON environment variables which may have unexpected results, for example, if you have `PYTHONPATH` set in a shell profile. If necessary, there are command line options like `-E` to ignore these environment variables.

- The free-threaded build links to the third-party shared libraries, such as `OpenSSL` and `Tk`, installed in the traditional framework. This means that both builds also share one set of trust certificates as installed by the **Install Certificates.command** script, thus it only needs to be run once.

- If you cannot depend on the link in `/usr/local/bin` pointing to the `python.org` free-threaded `python3.14t` (for example, if you want to install your own version there or some other distribution does), you can explicitly set your shell `PATH` environment variable to include the `PythonT` framework `bin` directory:

  ```
  export
  PATH="/Library/Frameworks/PythonT.framework/Versions/3.14/b
  in":"$PATH"
  ```

The traditional framework installation by default does something similar, except for `Python.framework`. Be aware that having both framework `bin` directories in `PATH` can lead to confusion if there are duplicate names like `python3.14` in both; which one is actually used depends on the order they appear in `PATH`. The `which python3.x` or `which python3.xt` commands can show which path is being used. Using virtual environments can help avoid such ambiguities. Another option might be to create a shell **alias** to the desired interpreter, like:

```
alias
py3.14="/Library/Frameworks/Python.framework/Versions/3.14/
bin/python3.14"
alias
py3.14t="/Library/Frameworks/PythonT.framework/Versions/3.1
4/bin/python3.14t"
```

5.5.2. Installing using the command line

If you want to use automation to install the `python.org` installer package (rather than by using the familiar macOS **Installer** GUI app), the macOS command line **installer** utility lets you select non-default options, too. If you are not familiar with **installer**, it can be somewhat cryptic (see **man installer** for more information). As an example, the following shell snippet shows one way to do it, using the `3.14.0b2` release and selecting the free-threaded interpreter option:

```
RELEASE="python-3.140b2-macos11.pkg"

# download installer pkg
curl -O https://www.python.org/ftp/python/3.14.0/${RELEASE}

# create installer choicechanges to customize the install:
#    enable the PythonTFramework-3.14 package
#    while accepting the other defaults (install all other
packages)
cat > ./choicechanges.plist <<EOF
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE plist PUBLIC "-//Apple//DTD PLIST 1.0//EN"
"http://www.apple.com/DTDs/PropertyList-1.0.dtd">
<plist version="1.0">
<array>
        <dict>
                <key>attributeSetting</key>
                <integer>1</integer>
                <key>choiceAttribute</key>
                <string>selected</string>
                <key>choiceIdentifier</key>
                <string>org.python.Python.PythonTFramework-
3.14</string>
        </dict>
</array>
</plist>
EOF

sudo installer -pkg ./${RELEASE} -applyChoiceChangesXML
./choicechanges.plist -target /
```

You can then test that both installer builds are now available with something like:

```
$ # test that the free-threaded interpreter was installed if
the Unix Command Tools package was enabled
$ /usr/local/bin/python3.14t -VV
Python 3.14.0b2 free-threading build (v3.14.0b2:3a83b172af, Jun
5 2024, 12:57:31) [Clang 15.0.0 (clang-1500.3.9.4)]
$ #    and the traditional interpreter
$ /usr/local/bin/python3.14 -VV
Python 3.14.0b2 (v3.14.0b2:3a83b172af, Jun  5 2024, 12:50:24)
[Clang 15.0.0 (clang-1500.3.9.4)]
$ # test that they are also available without the prefix if
/usr/local/bin is on $PATH
$ python3.14t -VV
Python 3.14.0b2 free-threading build (v3.14.0b2:3a83b172af, Jun
5 2024, 12:57:31) [Clang 15.0.0 (clang-1500.3.9.4)]
$ python3.14 -VV
```

```
Python 3.14.0b2 (v3.14.0b2:3a83b172af, Jun  5 2024, 12:50:24)
[Clang 15.0.0 (clang-1500.3.9.4)]
```

Note

Current `python.org` installers only install to fixed locations like `/Library/Frameworks/`, `/Applications`, and `/usr/local/bin`. You cannot use the **installer** `-domain` option to install to other locations.

5.5.3. Distributing Python Applications

A range of tools exist for converting your Python code into a standalone distributable application:

- py2app [https://pypi.org/project/py2app/]: Supports creating macOS `.app` bundles from a Python project.
- Briefcase [https://briefcase.readthedocs.io]: Part of the BeeWare Project [https://beeware.org]; a cross-platform packaging tool that supports creation of `.app` bundles on macOS, as well as managing signing and notarization.
- PyInstaller [https://pyinstaller.org/]: A cross-platform packaging tool that creates a single file or folder as a distributable artifact.

5.5.4. App Store Compliance

Apps submitted for distribution through the macOS App Store must pass Apple’s app review process. This process includes a set of automated validation rules that inspect the submitted application bundle for problematic code.

The Python standard library contains some code that is known to violate these automated rules. While these violations appear to be false positives, Apple’s review rules cannot be challenged. Therefore, it is necessary to modify the Python standard library for an app to pass App Store review.

The Python source tree contains a patch file [https://github.com/python/cpython/tree/3.14/Mac/Resources/app-store-compliance.patch] that will remove all code that is known to cause issues with the App Store review process. This patch is applied automatically when CPython is configured with the `--with-app-store-compliance` option.

This patch is not normally required to use CPython on a Mac; nor is it required if you are distributing an app *outside* the macOS App Store. It is *only* required if you are using the macOS App Store as a distribution channel.

5.6. Other Resources

The python.org Help page [https://www.python.org/about/help/] has links to many useful resources. The Pythonmac-SIG mailing list [https://www.python.org/community/sigs/current/pythonmac-sig/] is another support resource specifically for Python users and developers on the Mac.

6. Using Python on Android

Python on Android is unlike Python on desktop platforms. On a desktop platform, Python is generally installed as a system resource that can be used by any user of that computer. Users then interact with Python by running a **python** executable and entering commands at an interactive prompt, or by running a Python script.

On Android, there is no concept of installing as a system resource. The only unit of software distribution is an “app”. There is also no console where you could run a **python** executable, or interact with a Python REPL.

As a result, the only way you can use Python on Android is in embedded mode – that is, by writing a native Android application, embedding a Python interpreter using `libpython`, and invoking Python code using the Python embedding API. The full Python interpreter, the standard library, and all your Python code is then packaged into your app for its own private use.

The Python standard library has some notable omissions and restrictions on Android. See the API availability guide for details.

6.1. Adding Python to an Android app

Most app developers should use one of the following tools, which will provide a much easier experience:

- Briefcase [https://briefcase.readthedocs.io], from the BeeWare project
- Buildozer [https://buildozer.readthedocs.io], from the Kivy project
- Chaquopy [https://chaquo.com/chaquopy]
- pyqtdeploy [https://www.riverbankcomputing.com/static/Docs/pyqtdeploy/]
- Termux [https://termux.dev/en/]

If you’re sure you want to do all of this manually, read on. You can use the testbed app [https://github.com/python/cpython/tree/3.14/Android/testbed] as a guide; each step below contains a link to the relevant file.

- First, acquire a build of Python for Android:
  - The easiest way is to download an Android release from python.org [https://www.python.org/downloads/android/]. The `prefix` directory mentioned below is at the top level of the package.
  - Or if you want to build it yourself, follow the instructions in Android/README.md [https://github.com/python/cpython/tree/3.14/Android/README.md]. The `prefix` directory will be created under `cross-build/`*`HOST`*.
- Add code to your build.gradle [https://github.com/python/cpython/tree/3.14/Android/testbed/app/build.gradle.kts] file to copy the following items into your project. All except your own Python code can be copied from `prefix/lib`:
  - In your JNI libraries:
    - `libpython*.*.so`
    - `lib*_python.so` (external libraries such as OpenSSL)
  - In your assets:
    - `python*.*` (the Python standard library)
    - `python*.*/site-packages` (your own Python code)

- Add code to your app to extract the assets to the filesystem [https://github.com/python/cpython/tree/3.14/Android/testbed/app/src/main/java/org/python/testbed/MainActivity.kt].
- Add code to your app to start Python in embedded mode [https://github.com/python/cpython/tree/3.14/Android/testbed/app/src/main/c/main_activity.c]. This will need to be C code called via JNI.

6.2. Building a Python package for Android

Python packages can be built for Android as wheels and released on PyPI. The recommended tool for doing this is cibuildwheel [https://cibuildwheel.pypa.io/en/stable/platforms/#android], which automates all the details of setting up a cross-compilation environment, building the wheel, and testing it on an emulator.

7. Using Python on iOS

Authors:: Russell Keith-Magee (2024-03)

Python on iOS is unlike Python on desktop platforms. On a desktop platform, Python is generally installed as a system resource that can be used by any user of that computer. Users then interact with Python by running a **python** executable and entering commands at an interactive prompt, or by running a Python script.

On iOS, there is no concept of installing as a system resource. The only unit of software distribution is an “app”. There is also no console where you could run a **python** executable, or interact with a Python REPL.

As a result, the only way you can use Python on iOS is in embedded mode - that is, by writing a native iOS application, and embedding a Python interpreter using `libPython`, and invoking Python code using the Python embedding API. The full Python interpreter, the standard library, and all your Python code is then packaged as a standalone bundle that can be distributed via the iOS App Store.

If you’re looking to experiment for the first time with writing an iOS app in Python, projects such as BeeWare [https://beeware.org] and Kivy [https://kivy.org] will provide a much more approachable user experience. These projects manage the complexities associated with getting an iOS project running, so you only need to deal with the Python code itself.

7.1. Python at runtime on iOS

7.1.1. iOS version compatibility

The minimum supported iOS version is specified at compile time, using the `--host` option to `configure`. By default, when compiled for iOS, Python will be compiled with a minimum supported iOS version of 13.0. To use a different minimum iOS version, provide the version number as part of the **`--host`** argument - for example, `--host=arm64-apple-ios15.4-simulator` would compile an ARM64 simulator build with a deployment target of 15.4.

7.1.2. Platform identification

When executing on iOS, `sys.platform` will report as `ios`. This value will be returned on an iPhone or iPad, regardless of whether the app is running on the simulator or a physical device.

Information about the specific runtime environment, including the iOS version, device model, and whether the device is a simulator, can be obtained using `platform.ios_ver()`. `platform.system()` will report `iOS` or `iPadOS`, depending on the device.

`os.uname()` reports kernel-level details; it will report a name of `Darwin`.

7.1.3. Standard library availability

The Python standard library has some notable omissions and restrictions on iOS. See the API availability guide for iOS for details.

7.1.4. Binary extension modules

One notable difference about iOS as a platform is that App Store distribution imposes hard requirements on the packaging of an application. One of these requirements governs how binary extension modules are distributed.

The iOS App Store requires that *all* binary modules in an iOS app must be dynamic libraries, contained in a framework with appropriate metadata, stored in the `Frameworks` folder of the packaged app. There can be only a single binary per framework, and there can be no executable binary material outside the `Frameworks` folder.

This conflicts with the usual Python approach for distributing binaries, which allows a binary extension module to be loaded from any location on `sys.path`. To ensure compliance with App Store policies, an iOS project must post-process any Python packages, converting `.so` binary modules into individual standalone frameworks with appropriate metadata and signing. For details on how to perform this post-processing, see the guide for adding Python to your project.

To help Python discover binaries in their new location, the original `.so` file on `sys.path` is replaced with a `.fwork` file. This file is a text file containing the location of the framework binary, relative to the app bundle. To allow the framework to resolve back to the original location, the framework must contain a `.origin` file that contains the location of the `.fwork` file, relative to the app bundle.

For example, consider the case of an import `from foo.bar import _whiz`, where `_whiz` is implemented with the binary module `sources/foo/bar/_whiz.abi3.so`, with `sources` being the location registered on `sys.path`, relative to the application bundle. This module *must* be distributed as `Frameworks/foo.bar._whiz.framework/foo.bar._whiz` (creating the framework name from the full import path of the module), with an `Info.plist` file in the `.framework` directory identifying the binary as a framework. The `foo.bar._whiz` module would be represented in the original location with a `sources/foo/bar/_whiz.abi3.fwork` marker file, containing the path `Frameworks/foo.bar._whiz/foo.bar._whiz`. The framework would also contain `Frameworks/foo.bar._whiz.framework/foo.bar._whiz.origin`, containing the path to the `.fwork` file.

When running on iOS, the Python interpreter will install an `AppleFrameworkLoader` that is able to read and import `.fwork` files. Once imported, the `__file__` attribute of the binary module will report as the location of the `.fwork` file. However, the `ModuleSpec` for the loaded module will report the `origin` as the location of the binary in the framework folder.

7.1.5. Compiler stub binaries

Xcode doesn’t expose explicit compilers for iOS; instead, it uses an `xcrun` script that resolves to a full compiler path (e.g., `xcrun --sdk iphoneos clang` to get the `clang` for an iPhone device). However, using this script poses two problems:

- The output of `xcrun` includes paths that are machine specific, resulting in a sysconfig module that cannot be shared between users; and
- It results in `CC`/`CPP`/`LD`/`AR` definitions that include spaces. There is a lot of C ecosystem tooling that assumes that you can split a command line at the first space to get the path to the compiler executable; this isn’t the case when using `xcrun`.

To avoid these problems, Python provided stubs for these tools. These stubs are shell script wrappers around the underingly `xcrun` tools, distributed in a `bin` folder distributed alongside the compiled iOS framework. These scripts are relocatable, and will always resolve to the appropriate local system paths. By including these scripts in the bin folder that accompanies a framework, the contents of the `sysconfig` module becomes useful for end-users to compile their own modules. When compiling third-party Python modules for iOS, you should ensure these stub binaries are on your path.

7.2. Installing Python on iOS

7.2.1. Tools for building iOS apps

Building for iOS requires the use of Apple’s Xcode tooling. It is strongly recommended that you use the most recent stable release of Xcode. This will require the use of the most (or second-most) recently released macOS version, as Apple does not maintain Xcode for older macOS versions. The Xcode Command Line Tools are not sufficient for iOS development; you need a *full* Xcode install.

If you want to run your code on the iOS simulator, you’ll also need to install an iOS Simulator Platform. You should be prompted to select an iOS Simulator Platform when you first run Xcode. Alternatively, you can add an iOS Simulator Platform by selecting from the Platforms tab of the Xcode Settings panel.

7.2.2. Adding Python to an iOS project

Python can be added to any iOS project, using either Swift or Objective C. The following examples will use Objective C; if you are using Swift, you may find a library like PythonKit [https://github.com/pvieito/PythonKit] to be helpful.

To add Python to an iOS Xcode project:

1. Build or obtain a Python `XCFramework`. See the instructions in Apple/iOS/README.md [https://github.com/python/cpython/tree/3.14/Apple/iOS/README.md] (in the CPython source distribution) for details on how to build a Python `XCFramework`. At a minimum, you will need a build that supports `arm64-apple-ios`, plus one of either `arm64-apple-ios-simulator` or `x86_64-apple-ios-simulator`.

2. Drag the `XCframework` into your iOS project. In the following instructions, we’ll assume you’ve dropped the `XCframework` into the root of your project; however, you can use any other location that you want by adjusting paths as needed.

3. Add your application code as a folder in your Xcode project. In the following instructions, we’ll assume that your user code is in a folder named `app` in the root of your project; you can use any other location by adjusting paths as needed. Ensure that this folder is associated with your app target.

4. Select the app target by selecting the root node of your Xcode project, then the target name in the sidebar that appears.

5. In the “General” settings, under “Frameworks, Libraries and Embedded Content”, add `Python.xcframework`, with “Embed & Sign” selected.

6. In the “Build Settings” tab, modify the following:

    - Build Options
        - User Script Sandboxing: No
        - Enable Testability: Yes
    - Search Paths
        - Framework Search Paths: `$(PROJECT_DIR)`

    - Header Search Paths: `"$(BUILT_PRODUCTS_DIR)/Python.framework/Headers"`
  - Apple Clang - Warnings - All languages
    - Quoted Include In Framework Header: No

7. Add a build step that processes the Python standard library, and your own Python binary dependencies. In the “Build Phases” tab, add a new “Run Script” build step *before* the “Embed Frameworks” step, but *after* the “Copy Bundle Resources” step. Name the step “Process Python libraries”, disable the “Based on dependency analysis” checkbox, and set the script content to:

   ```
   set -e
   source $PROJECT_DIR/Python.xcframework/build/build_utils.sh
   install_python Python.xcframework app
   ```

   If you have placed your XCframework somewhere other than the root of your project, modify the path to the first argument.

8. Add Objective C code to initialize and use a Python interpreter in embedded mode. You should ensure that:

   - UTF-8 mode (`PyPreConfig.utf8_mode`) is *enabled*;
   - Buffered stdio (`PyConfig.buffered_stdio`) is *disabled*;
   - Writing bytecode (`PyConfig.write_bytecode`) is *disabled*;
   - Signal handlers (`PyConfig.install_signal_handlers`) are *enabled*;
   - System logging (`PyConfig.use_system_logger`) is *enabled* (optional, but strongly recommended; this is enabled by default);
   - `PYTHONHOME` for the interpreter is configured to point at the `python` subfolder of your app’s bundle; and
   - The `PYTHONPATH` for the interpreter includes:

    - the `python/lib/python3.X` subfolder of your app’s bundle,
    - the `python/lib/python3.X/lib-dynload` subfolder of your app’s bundle, and
    - the `app` subfolder of your app’s bundle

  Your app’s bundle location can be determined using `[[NSBundle mainBundle] resourcePath]`.

Steps 7 and 8 of these instructions assume that you have a single folder of pure Python application code, named `app`. If you have third-party binary modules in your app, some additional steps will be required:

- You need to ensure that any folders containing third-party binaries are either associated with the app target, or are explicitly copied as part of step 7. Step 7 should also purge any binaries that are not appropriate for the platform a specific build is targeting (i.e., delete any device binaries if you’re building an app targeting the simulator).
- If you’re using a separate folder for third-party packages, ensure that folder is added to the end of the call to `install_python` in step 7, and as part of the **`PYTHONPATH`** configuration in step 8.
- If any of the folders that contain third-party packages will contain `.pth` files, you should add that folder as a *site directory* (using **`site.addsitedir()`**), rather than adding to **`PYTHONPATH`** or **`sys.path`** directly.

7.2.3. Testing a Python package

The CPython source tree contains a testbed project [https://github.com/python/cpython/tree/3.14/Apple/iOS/testbed] that is used to run the CPython test suite on the iOS simulator. This testbed can also be used as a testbed project for running your Python library’s test suite on iOS.

After building or obtaining an iOS XCFramework (see Apple/iOS/README.md [https://github.com/python/cpython/tree/3.14/Apple/iOS/README.md] for details), create a clone of the Python iOS testbed project. If you used the `Apple` build script to build the XCframework, you can run:

```
$ python cross-build/iOS/testbed clone --app <path/to/module1>
--app <path/to/module2> app-testbed
```

Or, if you’ve sourced your own XCframework, by running:

```
$ python Apple/testbed clone --platform iOS --framework
<path/to/Python.xcframework> --app <path/to/module1> --app
<path/to/module2> app-testbed
```

Any folders specified with the `--app` flag will be copied into the cloned testbed project. The resulting testbed will be created in the `app-testbed` folder. In this example, the `module1` and `module2` would be importable modules at runtime. If your project has additional dependencies, they can be installed into the `app-testbed/Testbed/app_packages` folder (using `pip install --target app-testbed/Testbed/app_packages` or similar).

You can then use the `app-testbed` folder to run the test suite for your app, For example, if `module1.tests` was the entry point to your test suite, you could run:

```
$ python app-testbed run -- module1.tests
```

This is the equivalent of running `python -m module1.tests` on a desktop Python build. Any arguments after the `--` will be passed to the testbed as if they were arguments to `python -m` on a desktop machine.

You can also open the testbed project in Xcode by running:

```
$ open app-testbed/iOSTestbed.xcodeproj
```

This will allow you to use the full Xcode suite of tools for debugging.

The arguments used to run the test suite are defined as part of the test plan. To modify the test plan, select the test plan node of the project tree (it should be the first child of the root node), and select the “Configurations” tab. Modify the “Arguments Passed On Launch” value to change the testing arguments.

The test plan also disables parallel testing, and specifies the use of the `Testbed.lldbinit` file for providing configuration of the debugger. The default debugger configuration disables automatic breakpoints on the `SIGINT`, `SIGUSR1`, `SIGUSR2`, and `SIGXFSZ` signals.

7.3. App Store Compliance

The only mechanism for distributing apps to third-party iOS devices is to submit the app to the iOS App Store; apps submitted for distribution must pass Apple’s app review process. This process includes a set of automated validation rules that inspect the submitted application bundle for problematic code. There are some steps that must be taken to ensure that your app will be able to pass these validation steps.

7.3.1. Incompatible code in the standard library

The Python standard library contains some code that is known to violate these automated rules. While these violations appear to be false positives, Apple’s review rules cannot be challenged; so, it is necessary to modify the Python standard library for an app to pass App Store review.

The Python source tree contains a patch file [https://github.com/python/cpython/tree/3.14/Mac/Resources/app-store-compliance.patch] that will remove all code that is known to cause issues with the App Store review process. This patch is applied automatically when building for iOS.

7.3.2. Privacy manifests

In April 2025, Apple introduced a requirement for certain third-party libraries to provide a Privacy Manifest [https://developer.apple.com/support/third-party-SDK-requirements]. As a result, if you have a binary module that uses one of the affected libraries, you must provide an `.xcprivacy` file for that library. OpenSSL is one library affected by this requirement, but there are others.

If you produce a binary module named `mymodule.so`, and use you the Xcode build script described in step 7 above, you can place a `mymodule.xcprivacy` file next to `mymodule.so`, and the privacy manifest will be installed into the required location when the binary module is converted into a framework.

8. Editors and IDEs

There are a number of IDEs that support Python programming language. Many editors and IDEs provide syntax highlighting, debugging tools, and **PEP 8** [https://peps.python.org/pep-0008/] checks.

8.1. IDLE — Python editor and shell

IDLE is Python’s Integrated Development and Learning Environment and is generally bundled with Python installs. If you are on Linux and do not have IDLE installed see Installing IDLE on Linux. For more information see the IDLE docs.

8.2. Other Editors and IDEs

Python’s community wiki has information submitted by the community on Editors and IDEs. Please go to Python Editors [https://wiki.python.org/moin/PythonEditors] and Integrated Development Environments [https://wiki.python.org/moin/IntegratedDevelopmentEnvironments] for a comprehensive list.

The Python Language Reference

This reference manual describes the syntax and “core semantics” of the language. It is terse, but attempts to be exact and complete. The semantics of non-essential built-in object types and of the built-in functions and modules are described in The Python Standard Library. For an informal introduction to the language, see The Python Tutorial. For C or C++ programmers, two additional manuals exist: Extending and Embedding the Python Interpreter describes the high-level picture of how to write a Python extension module, and the Python/C API Reference Manual describes the interfaces available to C/C++ programmers in detail.

- 1. Introduction
    - 1.1. Alternate Implementations
    - 1.2. Notation
- 2. Lexical analysis
    - 2.1. Line structure
    - 2.2. Other tokens
    - 2.3. Names (identifiers and keywords)
    - 2.4. Literals
    - 2.5. String and Bytes literals
    - 2.6. Numeric literals
    - 2.7. Operators and delimiters
- 3. Data model
    - 3.1. Objects, values and types
    - 3.2. The standard type hierarchy
    - 3.3. Special method names
    - 3.4. Coroutines
- 4. Execution model

  - 4.1. Structure of a program
  - 4.2. Naming and binding
  - 4.3. Exceptions
  - 4.4. Runtime Components
- 5. The import system
  - 5.1. `importlib`
  - 5.2. Packages
  - 5.3. Searching
  - 5.4. Loading
  - 5.5. The Path Based Finder
  - 5.6. Replacing the standard import system
  - 5.7. Package Relative Imports
  - 5.8. Special considerations for __main__
  - 5.9. References
- 6. Expressions
  - 6.1. Arithmetic conversions
  - 6.2. Atoms
  - 6.3. Primaries
  - 6.4. Await expression
  - 6.5. The power operator
  - 6.6. Unary arithmetic and bitwise operations
  - 6.7. Binary arithmetic operations
  - 6.8. Shifting operations
  - 6.9. Binary bitwise operations
  - 6.10. Comparisons
  - 6.11. Boolean operations
  - 6.12. Assignment expressions
  - 6.13. Conditional expressions
  - 6.14. Lambdas
  - 6.15. Expression lists
  - 6.16. Evaluation order
  - 6.17. Operator precedence
- 7. Simple statements
  - 7.1. Expression statements
  - 7.2. Assignment statements
  - 7.3. The `assert` statement

  - 7.4. The `pass` statement
  - 7.5. The `del` statement
  - 7.6. The `return` statement
  - 7.7. The `yield` statement
  - 7.8. The `raise` statement
  - 7.9. The `break` statement
  - 7.10. The `continue` statement
  - 7.11. The `import` statement
  - 7.12. The `global` statement
  - 7.13. The `nonlocal` statement
  - 7.14. The `type` statement
- 8. Compound statements
  - 8.1. The `if` statement
  - 8.2. The `while` statement
  - 8.3. The `for` statement
  - 8.4. The `try` statement
  - 8.5. The `with` statement
  - 8.6. The `match` statement
  - 8.7. Function definitions
  - 8.8. Class definitions
  - 8.9. Coroutines
  - 8.10. Type parameter lists
  - 8.11. Annotations
- 9. Top-level components
  - 9.1. Complete Python programs
  - 9.2. File input
  - 9.3. Interactive input
  - 9.4. Expression input
- 10. Full Grammar specification

1. Introduction

This reference manual describes the Python programming language. It is not intended as a tutorial.

While I am trying to be as precise as possible, I chose to use English rather than formal specifications for everything except syntax and lexical analysis. This should make the document more understandable to the average reader, but will leave room for ambiguities. Consequently, if you were coming from Mars and tried to re-implement Python from this document alone, you might have to guess things and in fact you would probably end up implementing quite a different language. On the other hand, if you are using Python and wonder what the precise rules about a particular area of the language are, you should definitely be able to find them here. If you would like to see a more formal definition of the language, maybe you could volunteer your time — or invent a cloning machine :-).

It is dangerous to add too many implementation details to a language reference document — the implementation may change, and other implementations of the same language may work differently. On the other hand, CPython is the one Python implementation in widespread use (although alternate implementations continue to gain support), and its particular quirks are sometimes worth being mentioned, especially where the implementation imposes additional limitations. Therefore, you'll find short "implementation notes" sprinkled throughout the text.

Every Python implementation comes with a number of built-in and standard modules. These are documented in The Python Standard Library. A few built-in modules are mentioned when they interact in a significant way with the language definition.

1.1. Alternate Implementations

Though there is one Python implementation which is by far the most popular, there are some alternate implementations which are of particular interest to different audiences.

Known implementations include:

CPython
: This is the original and most-maintained implementation of Python, written in C. New language features generally appear here first.

Jython
: Python implemented in Java. This implementation can be used as a scripting language for Java applications, or can be used to create applications using the Java class libraries. It is also often used to create tests for Java libraries. More information can be found at the Jython website [https://www.jython.org/].

Python for .NET
: This implementation actually uses the CPython implementation, but is a managed .NET application and makes .NET libraries available. It was created by Brian Lloyd. For more information, see the Python for .NET home page [https://pythonnet.github.io/].

IronPython
: An alternate Python for .NET. Unlike Python.NET, this is a complete Python implementation that generates IL, and compiles Python code directly to .NET assemblies. It was created by Jim Hugunin, the original creator of Jython. For

more information, see the IronPython website [https://ironpython.net/].

PyPy

An implementation of Python written completely in Python. It supports several advanced features not found in other implementations like stackless support and a Just in Time compiler. One of the goals of the project is to encourage experimentation with the language itself by making it easier to modify the interpreter (since it is written in Python). Additional information is available on the PyPy project's home page [https://pypy.org/].

Each of these implementations varies in some way from the language as documented in this manual, or introduces specific information beyond what's covered in the standard Python documentation. Please refer to the implementation-specific documentation to determine what else you need to know about the specific implementation you're using.

1.2. Notation

The descriptions of lexical analysis and syntax use a grammar notation that is a mixture of EBNF [https://en.wikipedia.org/wiki/Extended_Backus%E2%80%93Naur_form] and PEG [https://en.wikipedia.org/wiki/Parsing_expression_grammar]. For example:

```
name:   letter (letter | digit | "_")*
letter: "a"..."z" | "A"..."Z"
digit:  "0"..."9"
```

In this example, the first line says that a `name` is a `letter` followed by a sequence of zero or more `letter`s, `digit`s, and underscores. A `letter` in turn is any of the single characters `'a'` through `'z'` and `A` through `Z`; a `digit` is a single character from `0` to `9`.

Each rule begins with a name (which identifies the rule that's being defined) followed by a colon, `:`. The definition to the right of the colon uses the following syntax elements:

- `name`: A name refers to another rule. Where possible, it is a link to the rule's definition.
    - `TOKEN`: An uppercase name refers to a token. For the purposes of grammar definitions, tokens are the same as rules.
- `"text"`, `'text'`: Text in single or double quotes must match literally (without the quotes). The type of quote is chosen according to the meaning of `text`:
    - `'if'`: A name in single quotes denotes a keyword.
    - `"case"`: A name in double quotes denotes a soft-keyword.
    - `'@'`: A non-letter symbol in single quotes denotes an **`OP`** token, that is, a delimiter or operator.
- `e1 e2`: Items separated only by whitespace denote a sequence. Here, `e1` must be followed by `e2`.
- `e1 | e2`: A vertical bar is used to separate alternatives. It denotes PEG's "ordered choice": if `e1` matches, `e2` is not considered. In traditional PEG grammars, this is written as a slash, `/`, rather than a vertical bar. See **PEP 617** [https://peps.python.org/pep-0617/] for more background and details.
- `e*`: A star means zero or more repetitions of the preceding item.
- `e+`: Likewise, a plus means one or more repetitions.
- `[e]`: A phrase enclosed in square brackets means zero or one occurrences. In other words, the enclosed phrase is optional.
- `e?`: A question mark has exactly the same meaning as square brackets: the preceding item is optional.

- `(e)`: Parentheses are used for grouping.

The following notation is only used in lexical definitions.

- `"a"..."z"`: Two literal characters separated by three dots mean a choice of any single character in the given (inclusive) range of ASCII characters.
- `<...>`: A phrase between angular brackets gives an informal description of the matched symbol (for example, `<any ASCII character except "\">`), or an abbreviation that is defined in nearby text (for example, `<Lu>`).

Some definitions also use *lookaheads*, which indicate that an element must (or must not) match at a given position, but without consuming any input:

- `&e`: a positive lookahead (that is, `e` is required to match)
- `!e`: a negative lookahead (that is, `e` is required *not* to match)

The unary operators (`*`, `+`, `?`) bind as tightly as possible; the vertical bar (`|`) binds most loosely.

White space is only meaningful to separate tokens.

Rules are normally contained on a single line, but rules that are too long may be wrapped:

```
literal: stringliteral | bytesliteral
         | integer | floatnumber | imagnumber
```

Alternatively, rules may be formatted with the first line ending at the colon, and each alternative beginning with a vertical bar on a new line. For example:

```
literal:
   | stringliteral
   | bytesliteral
   | integer
   | floatnumber
   | imagnumber
```

This does *not* mean that there is an empty first alternative.

1.2.1. Lexical and Syntactic definitions

There is some difference between *lexical* and *syntactic* analysis: the lexical analyzer operates on the individual characters of the input source, while the *parser* (syntactic analyzer) operates on the stream of tokens generated by the lexical analysis. However, in some cases the exact boundary between the two phases is a CPython implementation detail.

The practical difference between the two is that in *lexical* definitions, all whitespace is significant. The lexical analyzer discards all whitespace that is not converted to tokens like **`token.INDENT`** or **`NEWLINE`**. *Syntactic* definitions then use these tokens, rather than source characters.

This documentation uses the same BNF grammar for both styles of definitions. All uses of BNF in the next chapter (Lexical analysis) are lexical definitions; uses in subsequent chapters are syntactic definitions.

2. Lexical analysis

A Python program is read by a *parser*. Input to the parser is a stream of tokens, generated by the *lexical analyzer* (also known as the *tokenizer*). This chapter describes how the lexical analyzer produces these tokens.

The lexical analyzer determines the program text’s encoding (UTF-8 by default), and decodes the text into source characters. If the text cannot be decoded, a `SyntaxError` is raised.

Next, the lexical analyzer uses the source characters to generate a stream of tokens. The type of a generated token generally depends on the next source character to be processed. Similarly, other special behavior of the analyzer depends on the first source character that hasn’t yet been processed. The following table gives a quick summary of these source characters, with links to sections that contain more information.

Character	Next token (or other relevant documentation)
• space • tab • formfeed	• Whitespace
• CR, LF	• New line • Indentation

Character	**Next token (or other relevant documentation)**
• backslash (`\`)	• Explicit line joining • (Also significant in string escape sequences)
• hash (`#`)	• Comment
• quote (`'`, `"`)	• String literal
• ASCII letter (`a`-`z`, `A`-`Z`) • non-ASCII character	• Name • Prefixed string or bytes literal
• underscore (`_`)	• Name • (Can also be part of numeric literals)
• number (`0`-`9`)	• Numeric literal
• dot (`.`)	• Numeric literal • Operator
• question mark (`?`) • dollar (`$`) • backquote (`` ` ``)	• Error (outside string literals and comments)

Character	Next token (or other relevant documentation)
- control character	
- other printing character	- Operator or delimiter
- end of file	- End marker

2.1. Line structure

A Python program is divided into a number of *logical lines*.

2.1.1. Logical lines

The end of a logical line is represented by the token `NEWLINE`. Statements cannot cross logical line boundaries except where `NEWLINE` is allowed by the syntax (e.g., between statements in compound statements). A logical line is constructed from one or more *physical lines* by following the explicit or implicit *line joining* rules.

2.1.2. Physical lines

A physical line is a sequence of characters terminated by one the following end-of-line sequences:

- the Unix form using ASCII LF (linefeed),
- the Windows form using the ASCII sequence CR LF (return followed by linefeed),

- the ‘Classic Mac OS [https://en.wikipedia.org/wiki/Classic_Mac_OS]’ form using the ASCII CR (return) character.

Regardless of platform, each of these sequences is replaced by a single ASCII LF (linefeed) character. (This is done even inside string literals.) Each line can use any of the sequences; they do not need to be consistent within a file.

The end of input also serves as an implicit terminator for the final physical line.

Formally:

```
newline: <ASCII LF> | <ASCII CR> <ASCII LF> | <ASCII CR>
```

2.1.3. Comments

A comment starts with a hash character (`#`) that is not part of a string literal, and ends at the end of the physical line. A comment signifies the end of the logical line unless the implicit line joining rules are invoked. Comments are ignored by the syntax.

2.1.4. Encoding declarations

If a comment in the first or second line of the Python script matches the regular expression `coding[=:]\s*([-\w.]+)`, this comment is processed as an encoding declaration; the first group of this expression names the encoding of the source code file. The encoding declaration must appear on a line of its own. If it is the second line, the first line must also be a comment-only line. The recommended forms of an encoding expression are

```
# -*- coding: <encoding-name> -*-
```

which is recognized also by GNU Emacs, and

```
# vim:fileencoding=<encoding-name>
```

which is recognized by Bram Moolenaar’s VIM.

If no encoding declaration is found, the default encoding is UTF-8. If the implicit or explicit encoding of a file is UTF-8, an initial UTF-8 byte-order mark (`b'\xef\xbb\xbf'`) is ignored rather than being a syntax error.

If an encoding is declared, the encoding name must be recognized by Python (see Standard Encodings). The encoding is used for all lexical analysis, including string literals, comments and identifiers.

All lexical analysis, including string literals, comments and identifiers, works on Unicode text decoded using the source encoding. Any Unicode code point, except the NUL control character, can appear in Python source.

```
source_character:  <any Unicode code point, except NUL>
```

2.1.5. Explicit line joining

Two or more physical lines may be joined into logical lines using backslash characters (\), as follows: when a physical line ends in a backslash that is not part of a string literal or comment, it is joined with the following forming a single logical line, deleting the backslash and the following end-of-line character. For example:

```
if 1900 < year < 2100 and 1 <= month <= 12 \
   and 1 <= day <= 31 and 0 <= hour < 24 \
   and 0 <= minute < 60 and 0 <= second < 60:   # Looks like a 
valid date
        return 1
```

A line ending in a backslash cannot carry a comment. A backslash does not continue a comment. A backslash does not continue a token except for string literals (i.e., tokens other than string literals cannot be split across physical lines using a backslash). A backslash is illegal elsewhere on a line outside a string literal.

2.1.6. Implicit line joining

Expressions in parentheses, square brackets or curly braces can be split over more than one physical line without using backslashes. For example:

```
month_names = ['Januari', 'Februari', 'Maart',      # These are 
the
               'April',   'Mei',      'Juni',       # Dutch 
names
               'Juli',    'Augustus', 'September',  # for the 
months
               'Oktober', 'November', 'December']   # of the 
year
```

Implicitly continued lines can carry comments. The indentation of the continuation lines is not important. Blank continuation lines are allowed. There is no NEWLINE token between implicit continuation lines. Implicitly continued lines can also occur within triple-quoted strings (see below); in that case they cannot carry comments.

2.1.7. Blank lines

A logical line that contains only spaces, tabs, formfeeds and possibly a comment, is ignored (i.e., no `NEWLINE` token is generated). During interactive input of statements, handling of a blank line may differ depending on the implementation of the read-eval-print loop. In the standard interactive interpreter, an entirely blank logical line (that is, one containing not even whitespace or a comment) terminates a multi-line statement.

2.1.8. Indentation

Leading whitespace (spaces and tabs) at the beginning of a logical line is used to compute the indentation level of the line, which in turn is used to determine the grouping of statements.

Tabs are replaced (from left to right) by one to eight spaces such that the total number of characters up to and including the replacement is a multiple of eight (this is intended to be the same rule as used by Unix). The total number of spaces preceding the first non-blank character then determines the line’s indentation. Indentation cannot be split over multiple physical lines using backslashes; the whitespace up to the first backslash determines the indentation.

Indentation is rejected as inconsistent if a source file mixes tabs and spaces in a way that makes the meaning dependent on the worth of a tab in spaces; a `TabError` is raised in that case.

Cross-platform compatibility note: because of the nature of text editors on non-UNIX platforms, it is unwise to use a mixture of spaces and tabs for the indentation in a single source file. It should also be noted that different platforms may explicitly limit the maximum indentation level.

A formfeed character may be present at the start of the line; it will be ignored for the indentation calculations above. Formfeed characters occurring elsewhere in the leading whitespace have an undefined effect (for instance, they may reset the space count to zero).

The indentation levels of consecutive lines are used to generate `INDENT` and `DEDENT` tokens, using a stack, as follows.

Before the first line of the file is read, a single zero is pushed on the stack; this will never be popped off again. The numbers pushed on the stack will always be strictly increasing from bottom to top. At the beginning of each logical line, the line's indentation level is compared to the top of the stack. If it is equal, nothing happens. If it is larger, it is pushed on the stack, and one **`INDENT`** token is generated. If it is smaller, it *must* be one of the numbers occurring on the stack; all numbers on the stack that are larger are popped off, and for each number popped off a **`DEDENT`** token is generated. At the end of the file, a **`DEDENT`** token is generated for each number remaining on the stack that is larger than zero.

Here is an example of a correctly (though confusingly) indented piece of Python code:

```
def perm(l):
        # Compute the list of all permutations of l
    if len(l) <= 1:
                  return [l]
    r = []
    for i in range(len(l)):
             s = l[:i] + l[i+1:]
             p = perm(s)
             for x in p:
              r.append(l[i:i+1] + x)
    return r
```

The following example shows various indentation errors:

```
 def perm(l):                       # error: first line
indented
for i in range(len(l)):             # error: not indented
    s = l[:i] + l[i+1:]
        p = perm(l[:i] + l[i+1:])   # error: unexpected indent
        for x in p:
                r.append(l[i:i+1] + x)
            return r                # error: inconsistent
dedent
```

(Actually, the first three errors are detected by the parser; only the last error is found by the lexical analyzer — the indentation of `return r` does not match a level popped off the stack.)

2.1.9. Whitespace between tokens

Except at the beginning of a logical line or in string literals, the whitespace characters space, tab and formfeed can be used interchangeably to separate tokens:

```
whitespace:  ' ' | tab | formfeed
```

Whitespace is needed between two tokens only if their concatenation could otherwise be interpreted as a different token. For example, `ab` is one token, but `a b` is two tokens. However, `+a` and `+ a` both produce two tokens, `+` and `a`, as `+a` is not a valid token.

2.1.10. End marker

At the end of non-interactive input, the lexical analyzer generates an **`ENDMARKER`** token.

2.2. Other tokens

Besides **`NEWLINE`**, **`INDENT`** and **`DEDENT`**, the following categories of tokens exist: *identifiers* and *keywords* (**`NAME`**), *literals* (such as **`NUMBER`** and **`STRING`**), and other symbols (*operators* and *delimiters*, **`OP`**). Whitespace characters (other than logical line terminators, discussed earlier) are not tokens, but serve to delimit tokens. Where ambiguity exists, a token comprises the longest possible string that forms a legal token, when read from left to right.

2.3. Names (identifiers and keywords)

`NAME` tokens represent *identifiers*, *keywords*, and *soft keywords*.

Names are composed of the following characters:

- uppercase and lowercase letters (`A-Z` and `a-z`),
- the underscore (`_`),
- digits (`0` through `9`), which cannot appear as the first character, and
- non-ASCII characters. Valid names may only contain “letter-like” and “digit-like” characters; see Non-ASCII characters in names for details.

Names must contain at least one character, but have no upper length limit. Case is significant.

Formally, names are described by the following lexical definitions:

```
NAME:          name_start name_continue*
name_start:    "a"..."z" | "A"..."Z" | "_" | <non-ASCII
character>
name_continue: name_start | "0"..."9"
identifier:    <NAME, except keywords>
```

Note that not all names matched by this grammar are valid; see Non-ASCII characters in names for details.

2.3.1. Keywords

The following names are used as reserved words, or *keywords* of the language, and cannot be used as ordinary identifiers. They must be spelled exactly as written here:

```
False      await      else       import     pass
None       break      except     in         raise
True       class      finally    is         return
and        continue   for        lambda     try
as         def        from       nonlocal   while
assert     del        global     not        with
async      elif       if         or         yield
```

2.3.2. Soft Keywords

Added in version 3.10.

Some names are only reserved under specific contexts. These are known as *soft keywords*:

- `match`, `case`, and `_`, when used in the **`match`** statement.
- `type`, when used in the **`type`** statement.

These syntactically act as keywords in their specific contexts, but this distinction is done at the parser level, not when tokenizing.

As soft keywords, their use in the grammar is possible while still preserving compatibility with existing code that uses these names as identifier names.

Changed in version 3.12: `type` is now a soft keyword.

2.3.3. Reserved classes of identifiers

Certain classes of identifiers (besides keywords) have special meanings. These classes are identified by the patterns of leading and trailing underscore characters:

`_*`

Not imported by `from module import *`.

`_`

In a `case` pattern within a **`match`** statement, `_` is a soft keyword that denotes a wildcard.

Separately, the interactive interpreter makes the result of the last evaluation available in the variable `_`. (It is stored in the **`builtins`** module, alongside built-in functions like `print`.)

Elsewhere, `_` is a regular identifier. It is often used to name “special” items, but it is not special to Python itself.

Note

The name `_` is often used in conjunction with internationalization; refer to the documentation for the **`gettext`** module for more information on this convention.

It is also commonly used for unused variables.

`__*__`

System-defined names, informally known as “dunder” names. These names are defined by the interpreter and its implementation (including the standard library). Current system names are discussed in the Special method names section and elsewhere. More will likely be defined in future versions of Python. *Any* use of `__*__` names, in any context, that does not follow explicitly documented use, is subject to breakage without warning.

`__*`

Class-private names. Names in this category, when used within the context of a class definition, are re-written to use a mangled form to help avoid name clashes between “private” attributes of base and derived classes. See section Identifiers (Names).

2.3.4. Non-ASCII characters in names

Names that contain non-ASCII characters need additional normalization and validation beyond the rules and grammar explained above. For example, `ř_1`, `蛇`, or `साँप` are valid names, but `r〰2`, `€`, or `🐍` are not.

This section explains the exact rules.

All names are converted into the normalization form [https://www.unicode.org/reports/tr15/#Norm_Forms] NFKC while parsing. This means that, for example, some typographic variants of characters are converted to their “basic” form. For example, `ﬁⁿₐˡᵢᶻₐᵗᵢᵒₙ` normalizes to `finalization`, so Python treats them as the same name:

```
>>> ﬁⁿₐˡᵢᶻₐᵗᵢᵒₙ = 3
>>> finalization
3
```

Note

Normalization is done at the lexical level only. Run-time functions that take names as *strings* generally do not normalize their arguments. For example, the variable defined above is accessible at run time in the **`globals()`** dictionary as `globals()["finalization"]` but not `globals()["ﬁⁿₐˡᵢᶻₐᵗᵢᵒₙ"]`.

Similarly to how ASCII-only names must contain only letters, digits and the underscore, and cannot start with a digit, a valid name must start with a character in the "letter-like" set `xid_start`, and the remaining characters must be in the "letter- and digit-like" set `xid_continue`.

These sets based on the *XID_Start* and *XID_Continue* sets as defined by the Unicode standard annex UAX-31 [https://www.unicode.org/reports/tr31/]. Python's `xid_start` additionally includes the underscore (_). Note that Python does not necessarily conform to UAX-31 [https://www.unicode.org/reports/tr31/].

A non-normative listing of characters in the *XID_Start* and *XID_Continue* sets as defined by Unicode is available in the DerivedCoreProperties.txt [https://www.unicode.org/Public/16.0.0/ucd/DerivedCoreProperties.txt] file in the Unicode Character Database. For reference, the construction rules for the `xid_*` sets are given below.

The set `id_start` is defined as the union of:

- Unicode category `<Lu>` - uppercase letters (includes `A` to `Z`)
- Unicode category `<Ll>` - lowercase letters (includes `a` to `z`)
- Unicode category `<Lt>` - titlecase letters
- Unicode category `<Lm>` - modifier letters
- Unicode category `<Lo>` - other letters
- Unicode category `<Nl>` - letter numbers
- {`"_"`} - the underscore
- `<Other_ID_Start>` - an explicit set of characters in PropList.txt [https://www.unicode.org/Public/16.0.0/ucd/PropList.txt] to support backwards compatibility

The set `xid_start` then closes this set under NFKC normalization, by removing all characters whose normalization is not of the form `id_start id_continue*`.

The set `id_continue` is defined as the union of:

- `id_start` (see above)
- Unicode category `<Nd>` - decimal numbers (includes `0` to `9`)
- Unicode category `<Pc>` - connector punctuations
- Unicode category `<Mn>` - nonspacing marks
- Unicode category `<Mc>` - spacing combining marks
- `<Other_ID_Continue>` - another explicit set of characters in PropList.txt [https://www.unicode.org/Public/16.0.0/ucd/PropList.txt] to support backwards compatibility

Again, `xid_continue` closes this set under NFKC normalization.

Unicode categories use the version of the Unicode Character Database as included in the **`unicodedata`** module.

See also

- **PEP 3131** [https://peps.python.org/pep-3131/] – Supporting Non-ASCII Identifiers
- **PEP 672** [https://peps.python.org/pep-0672/] – Unicode-related Security Considerations for Python

2.4. Literals

Literals are notations for constant values of some built-in types.

In terms of lexical analysis, Python has string, bytes and numeric literals.

Other “literals” are lexically denoted using keywords (`None`, `True`, `False`) and the special ellipsis token (`...`).

2.5. String and Bytes literals

String literals are text enclosed in single quotes (`'`) or double quotes (`"`). For example:

```
"spam"
'eggs'
```

The quote used to start the literal also terminates it, so a string literal can only contain the other quote (except with escape sequences, see below). For example:

```
'Say "Hello", please.'
"Don't do that!"
```

Except for this limitation, the choice of quote character (`'` or `"`) does not affect how the literal is parsed.

Inside a string literal, the backslash (`\`) character introduces an *escape sequence*, which has special meaning depending on the character after the backslash. For example, `\"` denotes the double quote character, and does *not* end the string:

```
>>> print("Say \"Hello\" to everyone!")
Say "Hello" to everyone!
```

See escape sequences below for a full list of such sequences, and more details.

2.5.1. Triple-quoted strings

Strings can also be enclosed in matching groups of three single or double quotes. These are generally referred to as *triple-quoted strings*:

```
"""This is a triple-quoted string."""
```

In triple-quoted literals, unescaped quotes are allowed (and are retained), except that three unescaped quotes in a row terminate the literal, if they are of the same kind (`'` or `"`) used at the start:

```
"""This string has "quotes" inside."""
```

Unescaped newlines are also allowed and retained:

```
'''This triple-quoted string
continues on the next line.'''
```

2.5.2. String prefixes

String literals can have an optional *prefix* that influences how the content of the literal is parsed, for example:

```
b"data"
f'{result=}'
```

The allowed prefixes are:

- `b`: Bytes literal
- `r`: Raw string
- `f`: Formatted string literal (“f-string”)
- `t`: Template string literal (“t-string”)
- `u`: No effect (allowed for backwards compatibility)

See the linked sections for details on each type.

Prefixes are case-insensitive (for example, ‘`B`’ works the same as ‘`b`’). The ‘`r`’ prefix can be combined with ‘`f`’, ‘`t`’ or ‘`b`’, so ‘`fr`’, ‘`rf`’, ‘`tr`’, ‘`rt`’, ‘`br`’, and ‘`rb`’ are also valid prefixes.

Added in version 3.3: The `'rb'` prefix of raw bytes literals has been added as a synonym of `'br'`.

Support for the unicode legacy literal (`u'value'`) was reintroduced to simplify the maintenance of dual Python 2.x and 3.x codebases. See **PEP 414** [https://peps.python.org/pep-0414/] for more information.

2.5.3. Formal grammar

String literals, except “f-strings” and “t-strings”, are described by the following lexical definitions.

These definitions use negative lookaheads (`!`) to indicate that an ending quote ends the literal.

```
STRING:          [stringprefix] (stringcontent)
stringprefix:    <("r" | "u" | "b" | "br" | "rb"), case-
insensitive>
stringcontent:
   | "'''" ( !"'''" longstringitem)* "'''"
   | '"""' ( !'"""' longstringitem)* '"""'
   | "'" ( !"'" stringitem)* "'"
   | '"' ( !'"' stringitem)* '"'
stringitem:      stringchar | stringescapeseq
stringchar:      <any source_character, except backslash and
newline>
longstringitem:  stringitem | newline
stringescapeseq: "\" <any source_character>
```

Note that as in all lexical definitions, whitespace is significant. In particular, the prefix (if any) must be immediately followed by the starting quote.

2.5.4. Escape sequences

Unless an ‘`r`’ or ‘`R`’ prefix is present, escape sequences in string and bytes literals are interpreted according to rules similar to those used by Standard C. The recognized escape sequences are:

Escape Sequence	Meaning
`\<newline>`	Ignored end of line
`\\`	Backslash
`\'`	Single quote
`\"`	Double quote
`\a`	ASCII Bell (BEL)
`\b`	ASCII Backspace (BS)
`\f`	ASCII Formfeed (FF)
`\n`	ASCII Linefeed (LF)
`\r`	ASCII Carriage Return (CR)
`\t`	ASCII Horizontal Tab (TAB)
`\v`	ASCII Vertical Tab (VT)

Escape Sequence	Meaning
`\`*`ooo`*	Octal character
`\x`*`hh`*	Hexadecimal character
`\N{`*`name`*`}`	Named Unicode character
`\u`*`xxxx`*	Hexadecimal Unicode character
`\U`*`xxxxxxxx`*	Hexadecimal Unicode character

2.5.4.1. Ignored end of line

A backslash can be added at the end of a line to ignore the newline:

```
>>> 'This string will not include \
... backslashes or newline characters.'
'This string will not include backslashes or newline
characters.'
```

The same result can be achieved using triple-quoted strings, or parentheses and string literal concatenation.

2.5.4.2. Escaped characters

To include a backslash in a non-raw Python string literal, it must be doubled. The `\\` escape sequence denotes a single backslash character:

```
>>> print('C:\\Program Files')
C:\Program Files
```

Similarly, the `\'` and `\"` sequences denote the single and double quote character, respectively:

```
>>> print('\' and \"')
' and "
```

2.5.4.3. Octal character

The sequence `\`*`ooo`* denotes a *character* with the octal (base 8) value *ooo*:

```
>>> '\120'
'P'
```

Up to three octal digits (0 through 7) are accepted.

In a bytes literal, *character* means a *byte* with the given value. In a string literal, it means a Unicode character with the given value.

Changed in version 3.11: Octal escapes with value larger than `0o377` (255) produce a `DeprecationWarning`.

Changed in version 3.12: Octal escapes with value larger than `0o377` (255) produce a `SyntaxWarning`. In a future Python version they will raise a `SyntaxError`.

2.5.4.4. Hexadecimal character

The sequence `\x`*`hh`* denotes a *character* with the hex (base 16) value *hh*:

```
>>> '\x50'
'P'
```

Unlike in Standard C, exactly two hex digits are required.

In a bytes literal, *character* means a *byte* with the given value. In a string literal, it means a Unicode character with the given value.

2.5.4.5. Named Unicode character

The sequence `\N{name}` denotes a Unicode character with the given *name*:

```
>>> '\N{LATIN CAPITAL LETTER P}'
'P'
>>> '\N{SNAKE}'
'🐍'
```

This sequence cannot appear in bytes literals.

Changed in version 3.3: Support for name aliases [https://www.unicode.org/Public/16.0.0/ucd/NameAliases.txt] has been added.

2.5.4.6. Hexadecimal Unicode characters

These sequences `\uxxxx` and `\Uxxxxxxxx` denote the Unicode character with the given hex (base 16) value. Exactly four digits are required for `\u`; exactly eight digits are required for `\U`. The latter can encode any Unicode character.

```
>>> '\u1234'
'ሴ'
>>> '\U0001f40d'
'🐍'
```

These sequences cannot appear in bytes literals.

2.5.4.7. Unrecognized escape sequences

Unlike in Standard C, all unrecognized escape sequences are left in the string unchanged, that is, *the backslash is left in the result*:

```
>>> print('\q')
\q
>>> list('\q')
['\\', 'q']
```

Note that for bytes literals, the escape sequences only recognized in string literals (`\N...`, `\u...`, `\U...`) fall into the category of unrecognized escapes.

Changed in version 3.6: Unrecognized escape sequences produce a **`DeprecationWarning`**.

Changed in version 3.12: Unrecognized escape sequences produce a **`SyntaxWarning`**. In a future Python version they will raise a **`SyntaxError`**.

2.5.5. Bytes literals

Bytes literals are always prefixed with '`b`' or '`B`'; they produce an instance of the **`bytes`** type instead of the **`str`** type. They may only contain ASCII characters; bytes with a numeric value of 128 or greater must be expressed with escape sequences (typically Hexadecimal character or Octal character):

```
>>> b'\x89PNG\r\n\x1a\n'
b'\x89PNG\r\n\x1a\n'
>>> list(b'\x89PNG\r\n\x1a\n')
[137, 80, 78, 71, 13, 10, 26, 10]
```

Similarly, a zero byte must be expressed using an escape sequence (typically `\0` or `\x00`).

2.5.6. Raw string literals

Both string and bytes literals may optionally be prefixed with a letter ‘`r`’ or ‘`R`’; such constructs are called *raw string literals* and *raw bytes literals* respectively and treat backslashes as literal characters. As a result, in raw string literals, escape sequences are not treated specially:

```
>>> r'\d{4}-\d{2}-\d{2}'
'\\d{4}-\\d{2}-\\d{2}'
```

Even in a raw literal, quotes can be escaped with a backslash, but the backslash remains in the result; for example, `r"\""` is a valid string literal consisting of two characters: a backslash and a double quote; `r"\"` is not a valid string literal (even a raw string cannot end in an odd number of backslashes). Specifically, *a raw literal cannot end in a single backslash* (since the backslash would escape the following quote character). Note also that a single backslash followed by a newline is interpreted as those two characters as part of the literal, *not* as a line continuation.

2.5.7. f-strings

Added in version 3.6.

Changed in version 3.7: The **`await`** and **`async for`** can be used in expressions within f-strings.

Changed in version 3.8: Added the debug specifier (`=`)

Changed in version 3.12: Many restrictions on expressions within f-strings have been removed. Notably, nested strings, comments, and backslashes are now permitted.

A *formatted string literal* or *f-string* is a string literal that is prefixed with ‘`f`’ or ‘`F`’. Unlike other string literals, f-strings do not have a constant value. They may contain *replacement fields* delimited by curly braces `{}`.

Replacement fields contain expressions which are evaluated at run time. For example:

```
>>> who = 'nobody'
>>> nationality = 'Spanish'
>>> f'{who.title()} expects the {nationality} Inquisition!'
'Nobody expects the Spanish Inquisition!'
```

Any doubled curly braces (`{{` or `}}`) outside replacement fields are replaced with the corresponding single curly brace:

```
>>> print(f'{{...}}')
{...}
```

Other characters outside replacement fields are treated like in ordinary string literals. This means that escape sequences are decoded (except when a literal is also marked as a raw string), and newlines are possible in triple-quoted f-strings:

```
>>> name = 'Galahad'
>>> favorite_color = 'blue'
>>> print(f'{name}:\t{favorite_color}')
Galahad:       blue
>>> print(rf"C:\Users\{name}")
C:\Users\Galahad
>>> print(f'''Three shall be the number of the counting
... and the number of the counting shall be three.''')
Three shall be the number of the counting
and the number of the counting shall be three.
```

Expressions in formatted string literals are treated like regular Python expressions. Each expression is evaluated in the context where the formatted string literal appears, in order from left to right. An empty expression is not allowed, and both **`lambda`** and assignment expressions `:=` must be surrounded by explicit parentheses:

```
>>> f'{(half := 1/2)}, {half * 42}'
'0.5, 21.0'
```

Reusing the outer f-string quoting type inside a replacement field is permitted:

```
>>> a = dict(x=2)
>>> f"abc {a["x"]} def"
'abc 2 def'
```

Backslashes are also allowed in replacement fields and are evaluated the same way as in any other context:

```
>>> a = ["a", "b", "c"]
>>> print(f"List a contains:\n{"\n".join(a)}")
List a contains:
a
b
c
```

It is possible to nest f-strings:

```
>>> name = 'world'
>>> f'Repeated:{f' hello {name}' * 3}'
'Repeated: hello world hello world hello world'
```

Portable Python programs should not use more than 5 levels of nesting.

CPython implementation detail: CPython does not limit nesting of f-strings.

Replacement expressions can contain newlines in both single-quoted and triple-quoted f-strings and they can contain comments. Everything that comes after a `#` inside a replacement field is a comment (even closing braces and quotes). This means that replacement fields with comments must be closed in a different line:

```
>>> a = 2
>>> f"abc{a  # This comment  }"  continues until the end of the
line
```

```
...       + 3}"
'abc5'
```

After the expression, replacement fields may optionally contain:

- a *debug specifier* – an equal sign (`=`), optionally surrounded by whitespace on one or both sides;
- a *conversion specifier* – `!s`, `!r` or `!a`; and/or
- a *format specifier* prefixed with a colon (`:`).

See the Standard Library section on f-strings for details on how these fields are evaluated.

As that section explains, *format specifiers* are passed as the second argument to the **`format()`** function to format a replacement field value. For example, they can be used to specify a field width and padding characters using the Format Specification Mini-Language:

```
>>> number = 14.3
>>> f'{number:20.7f}'
'          14.3000000'
```

Top-level format specifiers may include nested replacement fields:

```
>>> field_size = 20
>>> precision = 7
>>> f'{number:{field_size}.{precision}f}'
'          14.3000000'
```

These nested fields may include their own conversion fields and format specifiers:

```
>>> number = 3
>>> f'{number:{field_size}}'
'                   3'
```

```
>>> f'{number:{field_size:05}}'
'00000000000000000003'
```

However, these nested fields may not include more deeply nested replacement fields.

Formatted string literals cannot be used as docstrings, even if they do not include expressions:

```
>>> def foo():
...     f"Not a docstring"
...
>>> print(foo.__doc__)
None
```

See also

- **PEP 498** [https://peps.python.org/pep-0498/] – Literal String Interpolation
- **PEP 701** [https://peps.python.org/pep-0701/] – Syntactic formalization of f-strings
- **`str.format()`**, which uses a related format string mechanism.

2.5.8. t-strings

Added in version 3.14.

A *template string literal* or *t-string* is a string literal that is prefixed with ‘`t`’ or ‘`T`’. These strings follow the same syntax rules as formatted string literals. For differences in evaluation rules, see the Standard Library section on t-strings

2.5.9. Formal grammar for f-strings

F-strings are handled partly by the lexical analyzer, which produces the tokens `FSTRING_START`, `FSTRING_MIDDLE` and `FSTRING_END`, and partly by the parser, which handles expressions in the replacement field. The exact way the work is split is a CPython implementation detail.

Correspondingly, the f-string grammar is a mix of lexical and syntactic definitions.

Whitespace is significant in these situations:

- There may be no whitespace in `FSTRING_START` (between the prefix and quote).
- Whitespace in `FSTRING_MIDDLE` is part of the literal string contents.
- In `fstring_replacement_field`, if `f_debug_specifier` is present, all whitespace after the opening brace until the `f_debug_specifier`, as well as whitespace immediately following `f_debug_specifier`, is retained as part of the expression.

  CPython implementation detail: The expression is not handled in the tokenization phase; it is retrieved from the source code using locations of the `{` token and the token after `=`.

The `FSTRING_MIDDLE` definition uses negative lookaheads (`!`) to indicate special characters (backslash, newline, `{`, `}`) and sequences (`f_quote`).

```
fstring:    FSTRING_START fstring_middle* FSTRING_END

FSTRING_START:      fstringprefix ("'" | '"' | "'''" | '"""')
FSTRING_END:        f_quote
fstringprefix:      <("f" | "fr" | "rf"), case-insensitive>
```

```
f_debug_specifier:  '='
f_quote:            <the quote character(s) used in
FSTRING_START>

fstring_middle:
   | fstring_replacement_field
   | FSTRING_MIDDLE
FSTRING_MIDDLE:
   | (!"\" !newline !'{' !'}' !f_quote) source_character
   | stringescapeseq
   | "{{"
   | "}}"
   | <newline, in triple-quoted f-strings only>
fstring_replacement_field:
   | '{' f_expression [f_debug_specifier] [fstring_conversion]
         [fstring_full_format_spec] '}'
fstring_conversion:
   | "!" ("s" | "r" | "a")
fstring_full_format_spec:
   | ':' fstring_format_spec*
fstring_format_spec:
   | FSTRING_MIDDLE
   | fstring_replacement_field
f_expression:
   | ','.(conditional_expression | "*" or_expr)+ [","]
   | yield_expression
```

Note

In the above grammar snippet, the `f_quote` and `FSTRING_MIDDLE` rules are context-sensitive – they depend on the contents of `FSTRING_START` of the nearest enclosing `fstring`.

Constructing a more traditional formal grammar from this template is left as an exercise for the reader.

The grammar for t-strings is identical to the one for f-strings, with *t* instead of *f* at the beginning of rule and token names and in the prefix.

```
tstring:    TSTRING_START tstring_middle* TSTRING_END

<rest of the t-string grammar is omitted; see above>
```

2.6. Numeric literals

`NUMBER` tokens represent numeric literals, of which there are three types: integers, floating-point numbers, and imaginary numbers.

```
NUMBER: integer | floatnumber | imagnumber
```

The numeric value of a numeric literal is the same as if it were passed as a string to the `int`, `float` or `complex` class constructor, respectively. Note that not all valid inputs for those constructors are also valid literals.

Numeric literals do not include a sign; a phrase like `-1` is actually an expression composed of the unary operator '`-`' and the literal `1`.

2.6.1. Integer literals

Integer literals denote whole numbers. For example:

```
7
3
2147483647
```

There is no limit for the length of integer literals apart from what can be stored in available memory:

```
7922816251426433759354395033679228162514264337593543950336
```

Underscores can be used to group digits for enhanced readability, and are ignored for determining the numeric value of the literal. For example, the following literals are equivalent:

```
100_000_000_000
100000000000
1_00_00_00_00_000
```

Underscores can only occur between digits. For example, `_123`, `321_`, and `123__321` are *not* valid literals.

Integers can be specified in binary (base 2), octal (base 8), or hexadecimal (base 16) using the prefixes `0b`, `0o` and `0x`, respectively. Hexadecimal digits 10 through 15 are represented by letters `A`-`F`, case-insensitive. For example:

```
0b100110111
0b_1110_0101
0o177
0o377
0xdeadbeef
0xDead_Beef
```

An underscore can follow the base specifier. For example, `0x_1f` is a valid literal, but `0_x1f` and `0x__1f` are not.

Leading zeros in a non-zero decimal number are not allowed. For example, `0123` is not a valid literal. This is for disambiguation with C-style octal literals, which Python used before version 3.0.

Formally, integer literals are described by the following lexical definitions:

```
integer:      decinteger | bininteger | octinteger | hexinteger
| zerointeger
decinteger:   nonzerodigit (["_"] digit)*
```

```
bininteger:   "0" ("b" | "B") (["_"] bindigit)+
octinteger:   "0" ("o" | "O") (["_"] octdigit)+
hexinteger:   "0" ("x" | "X") (["_"] hexdigit)+
zerointeger:  "0"+ (["_"] "0")*
nonzerodigit: "1"..."9"
digit:        "0"..."9"
bindigit:     "0" | "1"
octdigit:     "0"..."7"
hexdigit:     digit | "a"..."f" | "A"..."F"
```

Changed in version 3.6: Underscores are now allowed for grouping purposes in literals.

2.6.2. Floating-point literals

Floating-point (float) literals, such as `3.14` or `1.5`, denote approximations of real numbers.

They consist of *integer* and *fraction* parts, each composed of decimal digits. The parts are separated by a decimal point, `.`:

```
2.71828
4.0
```

Unlike in integer literals, leading zeros are allowed. For example, `077.010` is legal, and denotes the same number as `77.01`.

As in integer literals, single underscores may occur between digits to help readability:

```
96_485.332_123
3.14_15_93
```

Either of these parts, but not both, can be empty. For example:

```
10.  # (equivalent to 10.0)
.001  # (equivalent to 0.001)
```

Optionally, the integer and fraction may be followed by an *exponent*: the letter `e` or `E`, followed by an optional sign, `+` or `-`, and a number in the same format as the integer and fraction parts. The `e` or `E` represents “times ten raised to the power of”:

```
1.0e3  # (represents 1.0×10³, or 1000.0)
1.166e-5  # (represents 1.166×10⁻⁵, or 0.00001166)
6.02214076e+23  # (represents 6.02214076×10²³, or
602214076000000000000000.)
```

In floats with only integer and exponent parts, the decimal point may be omitted:

```
1e3  # (equivalent to 1.e3 and 1.0e3)
0e0  # (equivalent to 0.)
```

Formally, floating-point literals are described by the following lexical definitions:

```
floatnumber:
   | digitpart "." [digitpart] [exponent]
   | "." digitpart [exponent]
   | digitpart exponent
digitpart: digit (["_"] digit)*
exponent:  ("e" | "E") ["+" | "-"] digitpart
```

Changed in version 3.6: Underscores are now allowed for grouping purposes in literals.

2.6.3. Imaginary literals

Python has complex number objects, but no complex literals. Instead, *imaginary literals* denote complex numbers with a zero real part.

For example, in math, the complex number 3+4.2*i* is written as the real number 3 added to the imaginary number 4.2*i*. Python uses a similar syntax, except the imaginary unit is written as `j` rather than *i*:

```
3+4.2j
```

This is an expression composed of the integer literal `3`, the operator ‘`+`’, and the imaginary literal `4.2j`. Since these are three separate tokens, whitespace is allowed between them:

```
3 + 4.2j
```

No whitespace is allowed *within* each token. In particular, the `j` suffix, may not be separated from the number before it.

The number before the `j` has the same syntax as a floating-point literal. Thus, the following are valid imaginary literals:

```
4.2j
3.14j
10.j
.001j
1e100j
3.14e-10j
3.14_15_93j
```

Unlike in a floating-point literal the decimal point can be omitted if the imaginary number only has an integer part. The number is still evaluated as a floating-point number, not an integer:

```
10j
0j
1000000000000000000000000j   # equivalent to 1e+24j
```

The `j` suffix is case-insensitive. That means you can use `J` instead:

```
3.14J   # equivalent to 3.14j
```

Formally, imaginary literals are described by the following lexical definition:

```
imagnumber: (floatnumber | digitpart) ("j" | "J")
```

2.7. Operators and delimiters

The following grammar defines *operator* and *delimiter* tokens, that is, the generic `OP` token type. A list of these tokens and their names is also available in the **`token`** module documentation.

```
OP:
   | assignment_operator
   | bitwise_operator
   | comparison_operator
   | enclosing_delimiter
   | other_delimiter
   | arithmetic_operator
   | "..."
   | other_op

assignment_operator:    "+=" | "-=" | "*=" | "**=" | "/="  |
"//=" | "%=" |
                        "&=" | "|=" | "^=" | "<<=" | ">>=" |
"@="  | ":="
bitwise_operator:       "&"  | "|"  | "^"  | "~"   | "<<"  |
">>"
comparison_operator:    "<=" | ">=" | "<"  | ">"   | "=="  |
"!="
enclosing_delimiter:    "("  | ")"  | "["  | "]"   | "{"   | "}"
other_delimiter:        ","  | ":"  | "!"  | ";"   | "="   | "-
>"
arithmetic_operator:    "+"  | "-"  | "**" | "*"   | "//"  | "/"
| "%"
other_op:               "."  | "@"
```

Note

Generally, *operators* are used to combine expressions, while *delimiters* serve other purposes. However, there is no clear, formal distinction between the two categories.

Some tokens can serve as either operators or delimiters, depending on usage. For example, `*` is both the multiplication operator and a delimiter used for sequence unpacking, and `@` is both the matrix multiplication and a delimiter that introduces decorators.

For some tokens, the distinction is unclear. For example, some people consider `.`, `(`, and `)` to be delimiters, while others see the **`getattr()`** operator and the function call operator(s).

Some of Python’s operators, like `and`, `or`, and `not in`, use keyword tokens rather than “symbols” (operator tokens).

A sequence of three consecutive periods (`...`) has a special meaning as an **`Ellipsis`** literal.

3. Data model

3.1. Objects, values and types

Objects are Python’s abstraction for data. All data in a Python program is represented by objects or by relations between objects. Even code is represented by objects.

Every object has an identity, a type and a value. An object’s *identity* never changes once it has been created; you may think of it as the object’s address in memory. The **`is`** operator compares the identity of two objects; the **`id()`** function returns an integer representing its identity.

CPython implementation detail: For CPython, `id(x)` is the memory address where `x` is stored.

An object’s type determines the operations that the object supports (e.g., “does it have a length?”) and also defines the possible values for objects of that type. The **`type()`** function returns an object’s type (which is an object itself). Like its identity, an object’s *type* is also unchangeable. [[1]]

The *value* of some objects can change. Objects whose value can change are said to be *mutable*; objects whose value is unchangeable once they are created are called *immutable*. (The value of an immutable container object that contains a reference to a mutable object can change when the latter’s value is changed; however the container is still considered immutable, because the collection of objects it contains cannot be changed. So, immutability is not strictly the same as having an unchangeable value, it is more subtle.) An object’s mutability is determined by its type; for instance, numbers, strings and tuples are immutable, while dictionaries and lists are mutable.

Objects are never explicitly destroyed; however, when they become unreachable they may be garbage-collected. An implementation is allowed to postpone garbage collection or omit it altogether — it is a matter of implementation quality how garbage collection is implemented, as long as no objects are collected that are still reachable.

CPython implementation detail: CPython currently uses a reference-counting scheme with (optional) delayed detection of cyclically linked garbage, which collects most objects as soon as they become unreachable, but is not guaranteed to collect garbage containing circular references. See the documentation of the `gc` module for information on controlling the collection of cyclic garbage. Other implementations act differently and CPython may change. Do not depend on immediate finalization of objects when they become unreachable (so you should always close files explicitly).

Note that the use of the implementation’s tracing or debugging facilities may keep objects alive that would normally be collectable. Also note that catching an exception with a `try`…`except` statement may keep objects alive.

Some objects contain references to “external” resources such as open files or windows. It is understood that these resources are freed when the object is garbage-collected, but since garbage collection is not guaranteed to happen, such objects also provide an explicit way to release the external resource, usually a `close()` method. Programs are strongly recommended to explicitly close such objects. The `try`...`finally` statement and the `with` statement provide convenient ways to do this.

Some objects contain references to other objects; these are called *containers*. Examples of containers are tuples, lists and dictionaries. The references are part of a container's value. In most cases, when we talk about the value of a container, we imply the values, not the identities of the contained objects; however, when we talk about the mutability of a container, only the identities of the immediately contained objects are implied. So, if an immutable container (like a tuple) contains a reference to a mutable object, its value changes if that mutable object is changed.

Types affect almost all aspects of object behavior. Even the importance of object identity is affected in some sense: for immutable types, operations that compute new values may actually return a reference to any existing object with the same type and value, while for mutable objects this is not allowed. For example, after `a = 1; b = 1`, *a* and *b* may or may not refer to the same object with the value one, depending on the implementation. This is because `int` is an immutable type, so the reference to `1` can be reused. This behaviour depends on the implementation used, so should not be relied upon, but is something to be aware of when making use of object identity tests. However, after `c = []; d = []`, *c* and *d* are guaranteed to refer to two different, unique, newly created empty lists. (Note that `e = f = []` assigns the *same* object to both *e* and *f*.)

3.2. The standard type hierarchy

Below is a list of the types that are built into Python. Extension modules (written in C, Java, or other languages, depending on the implementation) can define additional types. Future versions of Python may add types to the type hierarchy (e.g., rational numbers, efficiently stored arrays of integers, etc.), although such additions will often be provided via the standard library instead.

Some of the type descriptions below contain a paragraph listing ‘special attributes.’ These are attributes that provide access to the implementation and are not intended for general use. Their definition may change in the future.

3.2.1. None

This type has a single value. There is a single object with this value. This object is accessed through the built-in name `None`. It is used to signify the absence of a value in many situations, e.g., it is returned from functions that don’t explicitly return anything. Its truth value is false.

3.2.2. NotImplemented

This type has a single value. There is a single object with this value. This object is accessed through the built-in name **`NotImplemented`**. Numeric methods and rich comparison methods should return this value if they do not implement the operation for the operands provided. (The interpreter will then try the reflected operation, or some other fallback, depending on the operator.) It should not be evaluated in a boolean context.

See Implementing the arithmetic operations for more details.

Changed in version 3.9: Evaluating **`NotImplemented`** in a boolean context was deprecated.

Changed in version 3.14: Evaluating `NotImplemented` in a boolean context now raises a `TypeError`. It previously evaluated to `True` and emitted a `DeprecationWarning` since Python 3.9.

3.2.3. Ellipsis

This type has a single value. There is a single object with this value. This object is accessed through the literal `...` or the built-in name `Ellipsis`. Its truth value is true.

3.2.4. `numbers.Number`

These are created by numeric literals and returned as results by arithmetic operators and arithmetic built-in functions. Numeric objects are immutable; once created their value never changes. Python numbers are of course strongly related to mathematical numbers, but subject to the limitations of numerical representation in computers.

The string representations of the numeric classes, computed by `__repr__()` and `__str__()`, have the following properties:

- They are valid numeric literals which, when passed to their class constructor, produce an object having the value of the original numeric.
- The representation is in base 10, when possible.
- Leading zeros, possibly excepting a single zero before a decimal point, are not shown.
- Trailing zeros, possibly excepting a single zero after a decimal point, are not shown.
- A sign is shown only when the number is negative.

Python distinguishes between integers, floating-point numbers, and complex numbers:

3.2.4.1. `numbers.Integral`

These represent elements from the mathematical set of integers (positive and negative).

Note

The rules for integer representation are intended to give the most meaningful interpretation of shift and mask operations involving negative integers.

There are two types of integers:

Integers (`int`)

These represent numbers in an unlimited range, subject to available (virtual) memory only. For the purpose of shift and mask operations, a binary representation is assumed, and negative numbers are represented in a variant of 2's complement which gives the illusion of an infinite string of sign bits extending to the left.

Booleans (`bool`)

These represent the truth values False and True. The two objects representing the values `False` and `True` are the only Boolean objects. The Boolean type is a subtype of the integer type, and Boolean values behave like the values 0 and 1, respectively, in almost all contexts, the exception being that when converted to a string, the strings `"False"` or `"True"` are returned, respectively.

3.2.4.2. `numbers.Real` (`float`)

These represent machine-level double precision floating-point numbers. You are at the mercy of the underlying machine architecture (and C or Java implementation) for the accepted range and handling of overflow. Python does not support single-precision floating-point numbers; the savings in processor and memory usage that are usually the reason for using these are dwarfed by the overhead of using objects in Python, so there is no reason to complicate the language with two kinds of floating-point numbers.

3.2.4.3. `numbers.Complex` (`complex`)

These represent complex numbers as a pair of machine-level double precision floating-point numbers. The same caveats apply as for floating-point numbers. The real and imaginary parts of a complex number `z` can be retrieved through the read-only attributes `z.real` and `z.imag`.

3.2.5. Sequences

These represent finite ordered sets indexed by non-negative numbers. The built-in function `len()` returns the number of items of a sequence. When the length of a sequence is *n*, the index set contains the numbers 0, 1, …, *n*-1. Item *i* of sequence *a* is selected by `a[i]`. Some sequences, including built-in sequences, interpret negative subscripts by adding the sequence length. For example, `a[-2]` equals `a[n-2]`, the second to last item of sequence a with length `n`.

Sequences also support slicing: `a[i:j]` selects all items with index *k* such that *i* <= *k* < *j*. When used as an expression, a slice is a sequence of the same type. The comment above about negative indexes also applies to negative slice positions.

Some sequences also support “extended slicing” with a third “step” parameter: `a[i:j:k]` selects all items of *a* with index *x* where `x = i + n*k`, *n* >= `0` and *i* <= *x* < *j*.

Sequences are distinguished according to their mutability:

3.2.5.1. Immutable sequences

An object of an immutable sequence type cannot change once it is created. (If the object contains references to other objects, these other objects may be mutable and may be changed; however, the collection of objects directly referenced by an immutable object cannot change.)

The following types are immutable sequences:

Strings

A string is a sequence of values that represent Unicode code points. All the code points in the range `U+0000 - U+10FFFF` can be represented in a string. Python doesn’t have a char type; instead, every code point in the string is represented as a string object with length `1`. The built-in function **`ord()`** converts a code point from its string form to an integer in the range `0 - 10FFFF`; **`chr()`** converts an integer in the range `0 - 10FFFF` to the corresponding length `1` string object. **`str.encode()`** can be used to convert a **`str`** to **`bytes`** using the given text encoding, and **`bytes.decode()`** can be used to achieve the opposite.

Tuples

The items of a tuple are arbitrary Python objects. Tuples of two or more items are formed by comma-separated lists of expressions. A tuple of one item (a ‘singleton’) can be formed by affixing a comma to an expression (an expression by itself does not create a tuple, since parentheses must be usable for grouping of expressions). An empty tuple can be formed by an empty pair of parentheses.

Bytes

A bytes object is an immutable array. The items are 8-bit bytes, represented by integers in the range 0 <= x < 256. Bytes literals (like `b'abc'`) and the built-in `bytes()` constructor can be used to create bytes objects. Also, bytes objects can be decoded to strings via the `decode()` method.

3.2.5.2. Mutable sequences

Mutable sequences can be changed after they are created. The subscription and slicing notations can be used as the target of assignment and `del` (delete) statements.

Note

The `collections` and `array` module provide additional examples of mutable sequence types.

There are currently two intrinsic mutable sequence types:

Lists

The items of a list are arbitrary Python objects. Lists are formed by placing a comma-separated list of expressions in square brackets. (Note that there are no special cases needed to form lists of length 0 or 1.)

Byte Arrays

A bytearray object is a mutable array. They are created by the built-in `bytearray()` constructor. Aside from being mutable (and hence unhashable), byte arrays otherwise provide the same interface and functionality as immutable `bytes` objects.

3.2.6. Set types

These represent unordered, finite sets of unique, immutable objects. As such, they cannot be indexed by any subscript. However, they can be iterated over, and the built-in function `len()` returns the number of items in a set. Common uses for sets are fast membership testing, removing duplicates from a sequence, and computing mathematical operations such as intersection, union, difference, and symmetric difference.

For set elements, the same immutability rules apply as for dictionary keys. Note that numeric types obey the normal rules for numeric comparison: if two numbers compare equal (e.g., `1` and `1.0`), only one of them can be contained in a set.

There are currently two intrinsic set types:

Sets

These represent a mutable set. They are created by the built-in `set()` constructor and can be modified afterwards by several methods, such as `add()`.

Frozen sets

These represent an immutable set. They are created by the built-in `frozenset()` constructor. As a frozenset is immutable and hashable, it can be used again as an element of another set, or as a dictionary key.

3.2.7. Mappings

These represent finite sets of objects indexed by arbitrary index sets. The subscript notation `a[k]` selects the item indexed by `k` from the mapping `a`; this can be used in expressions and as the target of assignments or `del` statements. The built-in function `len()` returns the number of items in a mapping.

There is currently a single intrinsic mapping type:

3.2.7.1. Dictionaries

These represent finite sets of objects indexed by nearly arbitrary values. The only types of values not acceptable as keys are values containing lists or dictionaries or other mutable types that are compared by value rather than by object identity, the reason being that the efficient implementation of dictionaries requires a key’s hash value to remain constant. Numeric types used for keys obey the normal rules for numeric comparison: if two numbers compare equal (e.g., `1` and `1.0`) then they can be used interchangeably to index the same dictionary entry.

Dictionaries preserve insertion order, meaning that keys will be produced in the same order they were added sequentially over the dictionary. Replacing an existing key does not change the order, however removing a key and re-inserting it will add it to the end instead of keeping its old place.

Dictionaries are mutable; they can be created by the `{}` notation (see section Dictionary displays).

The extension modules **`dbm.ndbm`** and **`dbm.gnu`** provide additional examples of mapping types, as does the **`collections`** module.

Changed in version 3.7: Dictionaries did not preserve insertion order in versions of Python before 3.6. In CPython 3.6, insertion order was preserved, but it was considered an implementation detail at that time rather than a language guarantee.

3.2.8. Callable types

These are the types to which the function call operation (see section Calls) can be applied:

3.2.8.1. User-defined functions

A user-defined function object is created by a function definition (see section Function definitions). It should be called with an argument list containing the same number of items as the function's formal parameter list.

3.2.8.1.1. Special read-only attributes

Attribute	Meaning
function.__globals__	A reference to the **`dictionary`** that holds the function's global variables – the global namespace of the module in which the function was defined.
function.__closure__	`None` or a **`tuple`** of cells that contain bindings for the names specified in the **`co_freevars`** attribute of the function's **`code object`**. A cell object has the attribute `cell_contents`. This can be used to get the value of the cell, as well as set the value.

3.2.8.1.2. Special writable attributes

Most of these attributes check the type of the assigned value:

Attribute	**Meaning**
function.__doc__	The function’s documentation string, or `None` if unavailable.
function.__name__	The function’s name. See also: `__name__ attributes`.
function.__qualname__	The function’s qualified name. See also: `__qualname__ attributes`. *Added in version 3.3.*
function.__module__	The name of the module the function was defined in, or `None` if unavailable.
function.__defaults__	A `tuple` containing default parameter values for those parameters that have defaults, or `None` if no parameters have a default value.
function.__code__	The code object representing the compiled function body.

Attribute	Meaning
function.__dict__	The namespace supporting arbitrary function attributes. See also: `__dict__ attributes`.
function.__annotations__	A `dictionary` containing annotations of parameters. The keys of the dictionary are the parameter names, and `'return'` for the return annotation, if provided. See also: `object.__annotations__`. *Changed in version 3.14:* Annotations are now lazily evaluated. See **PEP 649**.
function.__annotate__	The annotate function for this function, or `None` if the function has no annotations. See `object.__annotate__`. *Added in version 3.14.*
function.__kwdefaults__	A `dictionary` containing defaults for keyword-only parameters.
function.__type_params__	A `tuple` containing the type parameters of a generic function.

Attribute	Meaning
	Added in version 3.12.

Function objects also support getting and setting arbitrary attributes, which can be used, for example, to attach metadata to functions. Regular attribute dot-notation is used to get and set such attributes.

CPython implementation detail: CPython’s current implementation only supports function attributes on user-defined functions. Function attributes on built-in functions may be supported in the future.

Additional information about a function’s definition can be retrieved from its code object (accessible via the **`__code__`** attribute).

3.2.8.2. Instance methods

An instance method object combines a class, a class instance and any callable object (normally a user-defined function).

Special read-only attributes:

method.__self__	Refers to the class instance object to which the method is bound
method.__func__	Refers to the original function object

method.__doc__	The method’s documentation (same as `method.__func__.__doc__`). A `string` if the original function had a docstring, else `None`.
method.__name__	The name of the method (same as `method.__func__.__name__`)
method.__module__	The name of the module the method was defined in, or `None` if unavailable.

Methods also support accessing (but not setting) the arbitrary function attributes on the underlying function object.

User-defined method objects may be created when getting an attribute of a class (perhaps via an instance of that class), if that attribute is a user-defined function object or a `classmethod` object.

When an instance method object is created by retrieving a user-defined function object from a class via one of its instances, its `__self__` attribute is the instance, and the method object is said to be *bound*. The new method’s `__func__` attribute is the original function object.

When an instance method object is created by retrieving a `classmethod` object from a class or instance, its `__self__` attribute is the class itself, and its `__func__` attribute is the function object underlying the class method.

When an instance method object is called, the underlying function (`__func__`) is called, inserting the class instance (`__self__`) in front of the argument list. For instance, when **`C`** is a class which contains a definition for a function **`f()`**, and `x` is an instance of **`C`**, calling `x.f(1)` is equivalent to calling `C.f(x, 1)`.

When an instance method object is derived from a **`classmethod`** object, the “class instance” stored in **`__self__`** will actually be the class itself, so that calling either `x.f(1)` or `C.f(1)` is equivalent to calling `f(C,1)` where `f` is the underlying function.

It is important to note that user-defined functions which are attributes of a class instance are not converted to bound methods; this *only* happens when the function is an attribute of the class.

3.2.8.3. Generator functions

A function or method which uses the **`yield`** statement (see section The yield statement) is called a *generator function*. Such a function, when called, always returns an iterator object which can be used to execute the body of the function: calling the iterator’s **`iterator.__next__()`** method will cause the function to execute until it provides a value using the **`yield`** statement. When the function executes a **`return`** statement or falls off the end, a **`StopIteration`** exception is raised and the iterator will have reached the end of the set of values to be returned.

3.2.8.4. Coroutine functions

A function or method which is defined using **`async def`** is called a *coroutine function*. Such a function, when called, returns a coroutine object. It may contain **`await`** expressions, as well as `async with` and `async for` statements. See also the Coroutine Objects section.

3.2.8.5. Asynchronous generator functions

A function or method which is defined using `async def` and which uses the `yield` statement is called a *asynchronous generator function*. Such a function, when called, returns an asynchronous iterator object which can be used in an `async for` statement to execute the body of the function.

Calling the asynchronous iterator's `aiterator.__anext__` method will return an awaitable which when awaited will execute until it provides a value using the `yield` expression. When the function executes an empty `return` statement or falls off the end, a `StopAsyncIteration` exception is raised and the asynchronous iterator will have reached the end of the set of values to be yielded.

3.2.8.6. Built-in functions

A built-in function object is a wrapper around a C function. Examples of built-in functions are `len()` and `math.sin()` (`math` is a standard built-in module). The number and type of the arguments are determined by the C function. Special read-only attributes:

- **`__doc__`** is the function's documentation string, or `None` if unavailable. See `function.__doc__`.
- **`__name__`** is the function's name. See `function.__name__`.
- **`__self__`** is set to `None` (but see the next item).
- **`__module__`** is the name of the module the function was defined in or `None` if unavailable. See `function.__module__`.

3.2.8.7. Built-in methods

This is really a different disguise of a built-in function, this time containing an object passed to the C function as an implicit extra argument. An example of a built-in method is `alist.append()`, assuming *alist* is a list object. In this case, the special read-only attribute **`__self__`** is set to the object denoted by *alist*. (The attribute has the same semantics as it does with **`other instance methods`**.)

3.2.8.8. Classes

Classes are callable. These objects normally act as factories for new instances of themselves, but variations are possible for class types that override **`__new__()`**. The arguments of the call are passed to **`__new__()`** and, in the typical case, to **`__init__()`** to initialize the new instance.

3.2.8.9. Class Instances

Instances of arbitrary classes can be made callable by defining a **`__call__()`** method in their class.

3.2.9. Modules

Modules are a basic organizational unit of Python code, and are created by the import system as invoked either by the **`import`** statement, or by calling functions such as **`importlib.import_module()`** and built-in **`__import__()`**. A module object has a namespace implemented by a **`dictionary`** object (this is the dictionary referenced by the **`__globals__`** attribute of functions defined in the module). Attribute references are translated to lookups in this dictionary, e.g., `m.x` is equivalent to `m.__dict__["x"]`. A module object does not contain the code object used to initialize the module (since it isn’t needed once the initialization is done).

Attribute assignment updates the module’s namespace dictionary, e.g., `m.x = 1` is equivalent to `m.__dict__["x"] = 1`.

3.2.9.1. Import-related attributes on module objects

Module objects have the following attributes that relate to the import system. When a module is created using the machinery associated with the import system, these attributes are filled in based on the module’s spec, before the loader executes and loads the module.

To create a module dynamically rather than using the import system, it’s recommended to use **`importlib.util.module_from_spec()`**, which will set the various import-controlled attributes to appropriate values. It’s also possible to use the **`types.ModuleType`** constructor to create modules directly, but this technique is more error-prone, as most attributes must be manually set on the module object after it has been created when using this approach.

Caution

With the exception of **`__name__`**, it is **strongly** recommended that you rely on **`__spec__`** and its attributes instead of any of the other individual attributes listed in this subsection. Note that updating an attribute on **`__spec__`** will not update the corresponding attribute on the module itself:

```
>>> import typing
>>> typing.__name__, typing.__spec__.name
('typing', 'typing')
>>> typing.__spec__.name = 'spelling'
>>> typing.__name__, typing.__spec__.name
('typing', 'spelling')
>>> typing.__name__ = 'keyboard_smashing'
>>> typing.__name__, typing.__spec__.name
('keyboard_smashing', 'spelling')
```

module.__name__

The name used to uniquely identify the module in the import system. For a directly executed module, this will be set to `"__main__"`.

This attribute must be set to the fully qualified name of the module. It is expected to match the value of **`module.__spec__.name`**.

module.__spec__

A record of the module’s import-system-related state.

Set to the **`module spec`** that was used when importing the module. See Module specs for more details.

Added in version 3.4.

module.__package__

The package a module belongs to.

If the module is top-level (that is, not a part of any specific package) then the attribute should be set to `''` (the empty string). Otherwise, it should be set to the name of the module’s package (which can be equal to **`module.__name__`** if the module itself is a package). See **PEP 366** [https://peps.python.org/pep-0366/] for further details.

This attribute is used instead of **`__name__`** to calculate explicit relative imports for main modules. It defaults to `None` for modules created dynamically using the **`types.ModuleType`** constructor; use **`importlib.util.module_from_spec()`** instead to ensure the attribute is set to a **`str`**.

It is **strongly** recommended that you use `module.__spec__.parent` instead of `module.__package__`. `__package__` is now only used as a fallback if `__spec__.parent` is not set, and this fallback path is deprecated.

Changed in version 3.4: This attribute now defaults to `None` for modules created dynamically using the `types.ModuleType` constructor. Previously the attribute was optional.

Changed in version 3.6: The value of `__package__` is expected to be the same as `__spec__.parent`. `__package__` is now only used as a fallback during import resolution if `__spec__.parent` is not defined.

Changed in version 3.10: `ImportWarning` is raised if an import resolution falls back to `__package__` instead of `__spec__.parent`.

Changed in version 3.12: Raise `DeprecationWarning` instead of `ImportWarning` when falling back to `__package__` during import resolution.

Deprecated since version 3.13, will be removed in version 3.15: `__package__` will cease to be set or taken into consideration by the import system or standard library.

module.__loader__

The loader object that the import machinery used to load the module.

This attribute is mostly useful for introspection, but can be used for additional loader-specific functionality, for example getting data associated with a loader.

`__loader__` defaults to `None` for modules created dynamically using the `types.ModuleType` constructor; use `importlib.util.module_from_spec()` instead to ensure the attribute is set to a loader object.

It is **strongly** recommended that you use `module.__spec__.loader` instead of `module.__loader__`.

Changed in version 3.4: This attribute now defaults to `None` for modules created dynamically using the `types.ModuleType` constructor. Previously the attribute was optional.

Deprecated since version 3.12, will be removed in version 3.16: Setting `__loader__` on a module while failing to set `__spec__.loader` is deprecated. In Python 3.16, `__loader__` will cease to be set or taken into consideration by the import system or the standard library.

module.__path__

A (possibly empty) sequence of strings enumerating the locations where the package’s submodules will be found. Non-package modules should not have a `__path__` attribute. See __path__ attributes on modules for more details.

It is **strongly** recommended that you use `module.__spec__.submodule_search_locations` instead of `module.__path__`.

module.__file__

module.__cached__

`__file__` and `__cached__` are both optional attributes that may or may not be set. Both attributes should be a `str` when they are available.

`__file__` indicates the pathname of the file from which the module was loaded (if loaded from a file), or the pathname of the shared library file for extension modules loaded dynamically from a shared library. It might be missing for certain types of modules, such as C modules that are statically linked into the interpreter, and the import system may opt to leave it unset if it has no semantic meaning (for example, a module loaded from a database).

If `__file__` is set then the `__cached__` attribute might also be set, which is the path to any compiled version of the code (for example, a byte-compiled file). The file does not need to exist to set this attribute; the path can simply point to where the compiled file *would* exist (see **PEP 3147** [https://peps.python.org/pep-3147/]).

Note that `__cached__` may be set even if `__file__` is not set. However, that scenario is quite atypical. Ultimately, the loader is what makes use of the module spec provided by the finder (from which `__file__` and `__cached__` are derived). So if a loader can load from a cached module but otherwise does not load from a file, that atypical scenario may be appropriate.

It is **strongly** recommended that you use `module.__spec__.cached` instead of `module.__cached__`.

Deprecated since version 3.13, will be removed in version 3.15: Setting `__cached__` on a module while failing to set `__spec__.cached` is deprecated. In Python 3.15, `__cached__` will cease to be set or taken into consideration by the import system or standard library.

3.2.9.2. Other writable attributes on module objects

As well as the import-related attributes listed above, module objects also have the following writable attributes:

module.__doc__

The module’s documentation string, or `None` if unavailable. See also: **`__doc__ attributes`**.

module.__annotations__

A dictionary containing variable annotations collected during module body execution. For best practices on working with **`__annotations__`**, see **`annotationlib`**.

Changed in version 3.14: Annotations are now lazily evaluated. See **PEP 649** [https://peps.python.org/pep-0649/].

module.__annotate__

The annotate function for this module, or `None` if the module has no annotations. See also: **`__annotate__`** attributes.

Added in version 3.14.

3.2.9.3. Module dictionaries

Module objects also have the following special read-only attribute:

module.__dict__

The module’s namespace as a dictionary object. Uniquely among the attributes listed here, **`__dict__`** cannot be accessed as a global variable from within a module; it can only be accessed as an attribute on module objects.

CPython implementation detail: Because of the way CPython clears module dictionaries, the module dictionary will be cleared when the module falls out of scope even if the dictionary still has live references. To avoid this, copy the dictionary or keep the module around while using its dictionary directly.

3.2.10. Custom classes

Custom class types are typically created by class definitions (see section Class definitions). A class has a namespace implemented by a dictionary object. Class attribute references are translated to lookups in this dictionary, e.g., `C.x` is translated to `C.__dict__["x"]` (although there are a number of hooks which allow for other means of locating attributes). When the attribute name is not found there, the attribute search continues in the base classes. This search of the base classes uses the C3 method resolution order which behaves correctly even in the presence of ‘diamond’ inheritance structures where there are multiple inheritance paths leading back to a common ancestor. Additional details on the C3 MRO used by Python can be found at The Python 2.3 Method Resolution Order.

When a class attribute reference (for class **`C`**, say) would yield a class method object, it is transformed into an instance method object whose **`__self__`** attribute is **`C`**. When it would yield a **`staticmethod`** object, it is transformed into the object wrapped by the static method object. See section Implementing Descriptors for another way in which attributes retrieved from a class may differ from those actually contained in its **`__dict__`**.

Class attribute assignments update the class’s dictionary, never the dictionary of a base class.

A class object can be called (see above) to yield a class instance (see below).

3.2.10.1. Special attributes

Attribute	Meaning
type.__name__	The class's name. See also: `__name__ attributes`.
type.__qualname__	The class's qualified name. See also: `__qualname__ attributes`.
type.__module__	The name of the module in which the class was defined.
type.__dict__	A `mapping proxy` providing a read-only view of the class's namespace. See also: `__dict__ attributes`.
type.__bases__	A `tuple` containing the class's bases. In most cases, for a class defined as `class X(A, B, C)`, `X.__bases__` will be exactly equal to `(A, B, C)`.
type.__base__	**CPython implementation detail:** The single base class in the inheritance chain that is responsible for the memory layout of instances. This

Attribute	Meaning
	attribute corresponds to `tp_base` at the C level.
type.__doc__	The class's documentation string, or `None` if undefined. Not inherited by subclasses.
type.__annotations__	A dictionary containing variable annotations collected during class body execution. See also: `__annotations__ attributes`. For best practices on working with `__annotations__`, please see `annotationlib`. Use `annotationlib.get_annotations()` instead of accessing this attribute directly. **Warning** Accessing the `__annotations__` attribute directly on a class object may return annotations for the wrong class, specifically in certain cases where the class, its base class, or a metaclass is defined under `from __future__ import annotations`. See **749** for details.

Attribute	**Meaning**
	This attribute does not exist on certain builtin classes. On user-defined classes without `__annotations__`, it is an empty dictionary. *Changed in version 3.14:* Annotations are now lazily evaluated. See **PEP 649**.
type.__annotate__()	The annotate function for this class, or `None` if the class has no annotations. See also: **`__annotate__ attributes`**. *Added in version 3.14.*
type.__type_params__	A **`tuple`** containing the type parameters of a generic class. *Added in version 3.12.*
type.__static_attributes__	A **`tuple`** containing names of attributes of this class which are assigned through `self.X` from any function in its body. *Added in version 3.13.*

Attribute	Meaning
type.__firstlineno__	The line number of the first line of the class definition, including decorators. Setting the `__module__` attribute removes the `__firstlineno__` item from the type's dictionary. *Added in version 3.13.*
type.__mro__	The `tuple` of classes that are considered when looking for base classes during method resolution.

3.2.10.2. Special methods

In addition to the special attributes described above, all Python classes also have the following two methods available:

type.mro()

This method can be overridden by a metaclass to customize the method resolution order for its instances. It is called at class instantiation, and its result is stored in `__mro__`.

type.__subclasses__()

Each class keeps a list of weak references to its immediate subclasses. This method returns a list of all those references still alive. The list is in definition order. Example:

```
>>> class A: pass
>>> class B(A): pass
>>> A.__subclasses__()
[<class 'B'>]
```

3.2.11. Class instances

A class instance is created by calling a class object (see above). A class instance has a namespace implemented as a dictionary which is the first place in which attribute references are searched. When an attribute is not found there, and the instance's class has an attribute by that name, the search continues with the class attributes. If a class attribute is found that is a user-defined function object, it is transformed into an instance method object whose **`__self__`** attribute is the instance. Static method and class method objects are also transformed; see above under "Classes". See section Implementing Descriptors for another way in which attributes of a class retrieved via its instances may differ from the objects actually stored in the class's **`__dict__`**. If no class attribute is found, and the object's class has a **`__getattr__()`** method, that is called to satisfy the lookup.

Attribute assignments and deletions update the instance's dictionary, never a class's dictionary. If the class has a **`__setattr__()`** or **`__delattr__()`** method, this is called instead of updating the instance dictionary directly.

Class instances can pretend to be numbers, sequences, or mappings if they have methods with certain special names. See section Special method names.

3.2.11.1. Special attributes

object.__class__

The class to which a class instance belongs.

object.__dict__

A dictionary or other mapping object used to store an object’s (writable) attributes. Not all instances have a **`__dict__`** attribute; see the section on __slots__ for more details.

3.2.12. I/O objects (also known as file objects)

A file object represents an open file. Various shortcuts are available to create file objects: the **`open()`** built-in function, and also **`os.popen()`**, **`os.fdopen()`**, and the **`makefile()`** method of socket objects (and perhaps by other functions or methods provided by extension modules).

The objects `sys.stdin`, `sys.stdout` and `sys.stderr` are initialized to file objects corresponding to the interpreter’s standard input, output and error streams; they are all open in text mode and therefore follow the interface defined by the **`io.TextIOBase`** abstract class.

3.2.13. Internal types

A few types used internally by the interpreter are exposed to the user. Their definitions may change with future versions of the interpreter, but they are mentioned here for completeness.

3.2.13.1. Code objects

Code objects represent *byte-compiled* executable Python code, or bytecode. The difference between a code object and a function object is that the function object contains an explicit reference to the function’s globals (the module in which it was defined), while a code object contains no context; also the default argument values are stored in the function object, not in the code object (because they represent values calculated at run-time). Unlike function objects, code objects are immutable and contain no references (directly or indirectly) to mutable objects.

3.2.13.1.1. Special read-only attributes

codeobject.co_name	The function name
codeobject.co_qualname	The fully qualified function name *Added in version 3.11.*
codeobject.co_argcount	The total number of positional parameters (including positional-only parameters and parameters with default values) that the function has
codeobject.co_posonlyargcount	The number of positional-only parameters (including arguments with default values) that the function has

codeobject.co_kwonlyargcount	The number of keyword-only parameters (including arguments with default values) that the function has
codeobject.co_nlocals	The number of local variables used by the function (including parameters)
codeobject.co_varnames	A `tuple` containing the names of the local variables in the function (starting with the parameter names)
codeobject.co_cellvars	A `tuple` containing the names of local variables that are referenced from at least one nested scope inside the function
codeobject.co_freevars	A `tuple` containing the names of free (closure) variables that a nested scope references in an outer scope. See also `function.__closure__`. Note: references to global and builtin names are *not*

	included.
codeobject.co_code	A string representing the sequence of bytecode instructions in the function
codeobject.co_consts	A `tuple` containing the literals used by the bytecode in the function
codeobject.co_names	A `tuple` containing the names used by the bytecode in the function
codeobject.co_filename	The name of the file from which the code was compiled
codeobject.co_firstlineno	The line number of the first line of the function
codeobject.co_lnotab	A string encoding the mapping from bytecode offsets to line numbers. For details, see the source code of the interpreter. *Deprecated since version 3.12:* This attribute of code objects is

	deprecated, and may be removed in Python 3.15.
codeobject.co_stacksize	The required stack size of the code object
codeobject.co_flags	An **`integer`** encoding a number of flags for the interpreter.

The following flag bits are defined for **`co_flags`**: bit `0x04` is set if the function uses the `*arguments` syntax to accept an arbitrary number of positional arguments; bit `0x08` is set if the function uses the `**keywords` syntax to accept arbitrary keyword arguments; bit `0x20` is set if the function is a generator. See Code Objects Bit Flags for details on the semantics of each flags that might be present.

Future feature declarations (for example, `from __future__ import division`) also use bits in **`co_flags`** to indicate whether a code object was compiled with a particular feature enabled. See **`compiler_flag`**.

Other bits in **`co_flags`** are reserved for internal use.

If a code object represents a function and has a docstring, the **`CO_HAS_DOCSTRING`** bit is set in **`co_flags`** and the first item in **`co_consts`** is the docstring of the function.

3.2.13.1.2. Methods on code objects

codeobject.co_positions()

Returns an iterable over the source code positions of each bytecode instruction in the code object.

The iterator returns **`tuple`**s containing the `(start_line, end_line, start_column, end_column)`. The *i-th* tuple corresponds to the position of the source code that compiled to the *i-th* code unit. Column information is 0-indexed utf-8 byte offsets on the given source line.

This positional information can be missing. A non-exhaustive lists of cases where this may happen:

- Running the interpreter with **`-X`** `no_debug_ranges`.
- Loading a pyc file compiled while using **`-X`** `no_debug_ranges`.
- Position tuples corresponding to artificial instructions.
- Line and column numbers that can't be represented due to implementation specific limitations.

When this occurs, some or all of the tuple elements can be **`None`**.

Added in version 3.11.

Note

This feature requires storing column positions in code objects which may result in a small increase of disk usage of compiled Python files or interpreter memory usage. To avoid storing the extra information and/or deactivate printing the extra traceback information, the **`-X`** `no_debug_ranges` command line flag or the **`PYTHONNODEBUGRANGES`** environment variable can be used.

codeobject.co_lines()

Returns an iterator that yields information about successive ranges of bytecodes. Each item yielded is a `(start, end, lineno)` **`tuple`**:

- `start` (an **`int`**) represents the offset (inclusive) of the start of the bytecode range
- `end` (an **`int`**) represents the offset (exclusive) of the end of the bytecode range
- `lineno` is an **`int`** representing the line number of the bytecode range, or `None` if the bytecodes in the given range have no line number

The items yielded will have the following properties:

- The first range yielded will have a `start` of 0.
- The `(start, end)` ranges will be non-decreasing and consecutive. That is, for any pair of **`tuple`**s, the `start` of the second will be equal to the `end` of the first.
- No range will be backwards: `end >= start` for all triples.
- The last **`tuple`** yielded will have `end` equal to the size of the bytecode.

Zero-width ranges, where `start == end`, are allowed. Zero-width ranges are used for lines that are present in the source code, but have been eliminated by the bytecode compiler.

Added in version 3.10.

See also

PEP 626 [https://peps.python.org/pep-0626/] - Precise line numbers for debugging and other tools.

The PEP that introduced the **`co_lines()`** method.

codeobject.replace(***kwargs*)

Return a copy of the code object with new values for the specified fields.

Code objects are also supported by the generic function **`copy.replace()`**.

Added in version 3.8.

3.2.13.2. Frame objects

Frame objects represent execution frames. They may occur in traceback objects, and are also passed to registered trace functions.

3.2.13.2.1. Special read-only attributes

frame.f_back	Points to the previous stack frame (towards the caller), or `None` if this is the bottom stack frame
frame.f_code	The code object being executed in this frame. Accessing this attribute raises an auditing event `object.__getattr__` with arguments `obj` and `"f_code"`.
frame.f_locals	The mapping used by the frame to look up local variables. If the frame refers to an optimized scope, this may return a write-through proxy object. *Changed in version 3.13:* Return a proxy for optimized scopes.
frame.f_globals	The dictionary used by the frame to look up global variables

frame.f_builtins	The dictionary used by the frame to look up built-in (intrinsic) names
frame.f_lasti	The “precise instruction” of the frame object (this is an index into the bytecode string of the code object)
frame.f_generator	The generator or coroutine object that owns this frame, or `None` if the frame is a normal function. *Added in version 3.14.*

3.2.13.2.2. Special writable attributes

frame.f_trace	If not `None`, this is a function called for various events during code execution (this is used by debuggers). Normally an event is triggered for each new source line (see **`f_trace_lines`**).
frame.f_trace_lines	Set this attribute to **`False`** to disable triggering a tracing event for each source line.
frame.f_trace_opcodes	Set this attribute to **`True`** to allow per-opcode events to be requested. Note that this may lead to undefined interpreter behaviour

	if exceptions raised by the trace function escape to the function being traced.
frame.f_lineno	The current line number of the frame – writing to this from within a trace function jumps to the given line (only for the bottom-most frame). A debugger can implement a Jump command (aka Set Next Statement) by writing to this attribute.

3.2.13.2.3. Frame object methods

Frame objects support one method:

frame.clear()

This method clears all references to local variables held by the frame. Also, if the frame belonged to a generator, the generator is finalized. This helps break reference cycles involving frame objects (for example when catching an exception and storing its traceback for later use).

`RuntimeError` is raised if the frame is currently executing or suspended.

Added in version 3.4.

Changed in version 3.13: Attempting to clear a suspended frame raises **`RuntimeError`** (as has always been the case for executing frames).

3.2.13.3. Traceback objects

Traceback objects represent the stack trace of an exception. A traceback object is implicitly created when an exception occurs, and may also be explicitly created by calling `types.TracebackType`.

Changed in version 3.7: Traceback objects can now be explicitly instantiated from Python code.

For implicitly created tracebacks, when the search for an exception handler unwinds the execution stack, at each unwound level a traceback object is inserted in front of the current traceback. When an exception handler is entered, the stack trace is made available to the program. (See section The try statement.) It is accessible as the third item of the tuple returned by `sys.exc_info()`, and as the `__traceback__` attribute of the caught exception.

When the program contains no suitable handler, the stack trace is written (nicely formatted) to the standard error stream; if the interpreter is interactive, it is also made available to the user as `sys.last_traceback`.

For explicitly created tracebacks, it is up to the creator of the traceback to determine how the `tb_next` attributes should be linked to form a full stack trace.

Special read-only attributes:

traceback.tb_frame	Points to the execution frame of the current level. Accessing this attribute raises an auditing event `object.__getattr__` with arguments `obj` and `"tb_frame"`.

traceback.tb_lineno	Gives the line number where the exception occurred
traceback.tb_lasti	Indicates the “precise instruction”.

The line number and last instruction in the traceback may differ from the line number of its frame object if the exception occurred in a **`try`** statement with no matching except clause or with a **`finally`** clause.

traceback.tb_next

The special writable attribute **`tb_next`** is the next level in the stack trace (towards the frame where the exception occurred), or `None` if there is no next level.

Changed in version 3.7: This attribute is now writable

3.2.13.4. Slice objects

Slice objects are used to represent slices for **`__getitem__()`** methods. They are also created by the built-in **`slice()`** function.

Special read-only attributes: **`start`** is the lower bound; **`stop`** is the upper bound; **`step`** is the step value; each is `None` if omitted. These attributes can have any type.

Slice objects support one method:

slice.indices(*self*, *length*)

This method takes a single integer argument *length* and computes information about the slice that the slice object would describe if applied to a sequence of *length* items.

> It returns a tuple of three integers; respectively these are the *start* and *stop* indices and the *step* or stride length of the slice. Missing or out-of-bounds indices are handled in a manner consistent with regular slices.

3.2.13.5. Static method objects

Static method objects provide a way of defeating the transformation of function objects to method objects described above. A static method object is a wrapper around any other object, usually a user-defined method object. When a static method object is retrieved from a class or a class instance, the object actually returned is the wrapped object, which is not subject to any further transformation. Static method objects are also callable. Static method objects are created by the built-in `staticmethod()` constructor.

3.2.13.6. Class method objects

A class method object, like a static method object, is a wrapper around another object that alters the way in which that object is retrieved from classes and class instances. The behaviour of class method objects upon such retrieval is described above, under “instance methods”. Class method objects are created by the built-in `classmethod()` constructor.

3.3. Special method names

A class can implement certain operations that are invoked by special syntax (such as arithmetic operations or subscripting and slicing) by defining methods with special names. This is Python’s approach to *operator overloading*, allowing classes to define their own behavior with respect to language operators. For instance, if a class defines a method named `__getitem__()`, and `x` is an instance of this class, then `x[i]` is roughly equivalent to `type(x).__getitem__(x, i)`. Except where mentioned, attempts to execute an operation raise an exception when no appropriate method is defined (typically `AttributeError` or `TypeError`).

Setting a special method to `None` indicates that the corresponding operation is not available. For example, if a class sets `__iter__()` to `None`, the class is not iterable, so calling `iter()` on its instances will raise a `TypeError` (without falling back to `__getitem__()`). [[2]]

When implementing a class that emulates any built-in type, it is important that the emulation only be implemented to the degree that it makes sense for the object being modelled. For example, some sequences may work well with retrieval of individual elements, but extracting a slice may not make sense. (One example of this is the NodeList interface in the W3C's Document Object Model.)

3.3.1. Basic customization

object.__new__(*cls*[, ...])

Called to create a new instance of class *cls*. `__new__()` is a static method (special-cased so you need not declare it as such) that takes the class of which an instance was requested as its first argument. The remaining arguments are those passed to the object constructor expression (the call to the class). The return value of `__new__()` should be the new object instance (usually an instance of *cls*).

Typical implementations create a new instance of the class by invoking the superclass's `__new__()` method using `super().__new__(cls[, ...])` with appropriate arguments and then modifying the newly created instance as necessary before returning it.

If `__new__()` is invoked during object construction and it returns an instance of *cls*, then the new instance's `__init__()` method will be invoked like `__init__(self[, ...])`, where *self* is the new instance and the remaining arguments are the same as were passed to the object constructor.

If `__new__()` does not return an instance of *cls*, then the new instance's `__init__()` method will not be invoked.

`__new__()` is intended mainly to allow subclasses of immutable types (like int, str, or tuple) to customize instance creation. It is also commonly overridden in custom metaclasses in order to customize class creation.

object.__init__(*self*[, *...*])

Called after the instance has been created (by `__new__()`), but before it is returned to the caller. The arguments are those passed to the class constructor expression. If a base class has an `__init__()` method, the derived class's `__init__()` method, if any, must explicitly call it to ensure proper initialization of the base class part of the instance; for example: `super().__init__([args...])`.

Because `__new__()` and `__init__()` work together in constructing objects (`__new__()` to create it, and `__init__()` to customize it), no non-`None` value may be returned by `__init__()`; doing so will cause a `TypeError` to be raised at runtime.

object.__del__(*self*)

Called when the instance is about to be destroyed. This is also called a finalizer or (improperly) a destructor. If a base class has a **`__del__()`** method, the derived class's **`__del__()`** method, if any, must explicitly call it to ensure proper deletion of the base class part of the instance.

It is possible (though not recommended!) for the **`__del__()`** method to postpone destruction of the instance by creating a new reference to it. This is called object *resurrection*. It is implementation-dependent whether **`__del__()`** is called a second time when a resurrected object is about to be destroyed; the current CPython implementation only calls it once.

It is not guaranteed that **`__del__()`** methods are called for objects that still exist when the interpreter exits. **`weakref.finalize`** provides a straightforward way to register a cleanup function to be called when an object is garbage collected.

Note

`del x` doesn't directly call `x.__del__()` — the former decrements the reference count for `x` by one, and the latter is only called when `x`'s reference count reaches zero.

CPython implementation detail: It is possible for a reference cycle to prevent the reference count of an object from going to zero. In this case, the cycle will be later detected and deleted by the cyclic garbage collector. A common cause of reference cycles is when an exception has been caught in a local variable. The frame's locals then reference the exception, which references its own traceback, which references the locals of all frames caught in the traceback.

See also

Documentation for the `gc` module.

Warning

Due to the precarious circumstances under which `__del__()` methods are invoked, exceptions that occur during their execution are ignored, and a warning is printed to `sys.stderr` instead. In particular:

- `__del__()` can be invoked when arbitrary code is being executed, including from any arbitrary thread. If `__del__()` needs to take a lock or invoke any other blocking resource, it may deadlock as the resource may already be taken by the code that gets interrupted to execute `__del__()`.
- `__del__()` can be executed during interpreter shutdown. As a consequence, the global variables it needs to access (including other modules) may already have been deleted or set to `None`. Python guarantees that globals whose name begins with a single underscore are deleted from their module before other globals are deleted; if no other references to such globals exist, this may help in assuring that imported modules are still available at the time when the `__del__()` method is called.

object.__repr__(*self*)

Called by the `repr()` built-in function to compute the “official” string representation of an object. If at all possible, this should look like a valid Python expression that could be used to recreate an object with the same value (given an appropriate environment). If this is not possible, a string of the form `<...some useful` `description...>` should be returned. The return value must be a string object. If a class defines `__repr__()` but not `__str__()`, then `__repr__()` is also used when an “informal” string representation of instances of that class is required.

This is typically used for debugging, so it is important that the representation is information-rich and unambiguous. A default implementation is provided by the `object` class itself.

object.__str__(*self*)

Called by `str(object)`, the default `__format__()` implementation, and the built-in function `print()`, to compute the “informal” or nicely printable string representation of an object. The return value must be a str object.

This method differs from `object.__repr__()` in that there is no expectation that `__str__()` return a valid Python expression: a more convenient or concise representation can be used.

The default implementation defined by the built-in type `object` calls `object.__repr__()`.

object.__bytes__(*self*)

Called by bytes to compute a byte-string representation of an object. This should return a `bytes` object. The `object` class itself does not provide this method.

object.__format__(*self*, *format_spec*)

Called by the `format()` built-in function, and by extension, evaluation of formatted string literals and the `str.format()` method, to produce a “formatted” string representation of an object. The *format_spec* argument is a string that contains a description of the formatting options desired. The interpretation of the *format_spec* argument is up to the type implementing `__format__()`, however most classes will either delegate formatting to one of the built-in types, or use a similar formatting option syntax.

See Format Specification Mini-Language for a description of the standard formatting syntax.

The return value must be a string object.

The default implementation by the `object` class should be given an empty *format_spec* string. It delegates to `__str__()`.

Changed in version 3.4: The __format__ method of `object` itself raises a `TypeError` if passed any non-empty string.

Changed in version 3.7: `object.__format__(x, '')` is now equivalent to `str(x)` rather than `format(str(x), '')`.

object.__lt__(*self*, *other*)
object.__le__(*self*, *other*)
object.__eq__(*self*, *other*)
object.__ne__(*self*, *other*)
object.__gt__(*self*, *other*)
object.__ge__(*self*, *other*)

These are the so-called “rich comparison” methods. The correspondence between operator symbols and method names is as follows: `x<y` calls `x.__lt__(y)`, `x<=y` calls `x.__le__(y)`, `x==y` calls `x.__eq__(y)`, `x!=y` calls `x.__ne__(y)`, `x>y` calls `x.__gt__(y)`, and `x>=y` calls `x.__ge__(y)`.

A rich comparison method may return the singleton `NotImplemented` if it does not implement the operation for a given pair of arguments. By convention, `False` and `True` are returned for a successful comparison. However, these methods can return any value, so if the comparison operator is used in a Boolean context (e.g., in the condition of an `if` statement), Python will call **`bool()`** on the value to determine if the result is true or false.

By default, `object` implements **`__eq__()`** by using `is`, returning **`NotImplemented`** in the case of a false comparison: `True if x is y else NotImplemented`. For **`__ne__()`**, by default it delegates to **`__eq__()`** and inverts the result unless it is **`NotImplemented`**. There are no other implied relationships among the comparison operators or default implementations; for example, the truth of `(x<y or x==y)` does not imply `x<=y`. To automatically generate ordering operations from a single root operation, see **`functools.total_ordering()`**.

By default, the **`object`** class provides implementations consistent with Value comparisons: equality compares according to object identity, and order comparisons raise **`TypeError`**. Each default method may generate these results directly, but may also return **`NotImplemented`**.

See the paragraph on **`__hash__()`** for some important notes on creating hashable objects which support custom comparison operations and are usable as dictionary keys.

There are no swapped-argument versions of these methods (to be used when the left argument does not support the operation but the right argument does); rather, **`__lt__()`** and **`__gt__()`** are each other’s reflection, **`__le__()`** and **`__ge__()`** are each other’s reflection, and **`__eq__()`** and **`__ne__()`** are their own reflection. If the operands are of different types, and the right operand’s type is a direct or indirect subclass of the left operand’s type, the reflected method of the right operand has priority, otherwise the left operand’s method has priority. Virtual subclassing is not considered.

When no appropriate method returns any value other than **`NotImplemented`**, the `==` and `!=` operators will fall back to `is` and `is not`, respectively.

object.__hash__(*self*)

Called by built-in function **`hash()`** and for operations on members of hashed collections including **`set`**, **`frozenset`**, and **`dict`**. The `__hash__()` method should return an integer. The only required property is that objects which compare equal have the same hash value; it is advised to mix together the hash values of the components of the object that also play a part in comparison of objects by packing them into a tuple and hashing the tuple. Example:

```
def __hash__(self):
    return hash((self.name, self.nick, self.color))
```

Note

`hash()` truncates the value returned from an object’s custom **`__hash__()`** method to the size of a **`Py_ssize_t`**. This is typically 8 bytes on 64-bit builds and 4 bytes on 32-bit builds. If an object’s **`__hash__()`** must interoperate on builds of different bit sizes, be sure to check the width on all supported builds. An easy way to do this is with `python -c "import sys; print(sys.hash_info.width)"`.

If a class does not define an **`__eq__()`** method it should not define a **`__hash__()`** operation either; if it defines **`__eq__()`** but not **`__hash__()`**, its instances will not be usable as items in hashable collections. If a class defines mutable objects and implements an **`__eq__()`** method, it should not implement **`__hash__()`**, since the implementation of hashable collections requires that a key’s hash value is immutable (if the object’s hash value changes, it will be in the wrong hash bucket).

User-defined classes have `__eq__()` and `__hash__()` methods by default (inherited from the `object` class); with them, all objects compare unequal (except with themselves) and `x.__hash__()` returns an appropriate value such that `x == y` implies both that `x is y` and `hash(x) == hash(y)`.

A class that overrides `__eq__()` and does not define `__hash__()` will have its `__hash__()` implicitly set to `None`. When the `__hash__()` method of a class is `None`, instances of the class will raise an appropriate `TypeError` when a program attempts to retrieve their hash value, and will also be correctly identified as unhashable when checking `isinstance(obj, collections.abc.Hashable)`.

If a class that overrides `__eq__()` needs to retain the implementation of `__hash__()` from a parent class, the interpreter must be told this explicitly by setting `__hash__ = <ParentClass>.__hash__`.

If a class that does not override `__eq__()` wishes to suppress hash support, it should include `__hash__ = None` in the class definition. A class which defines its own `__hash__()` that explicitly raises a `TypeError` would be incorrectly identified as hashable by an `isinstance(obj, collections.abc.Hashable)` call.

Note

By default, the `__hash__()` values of str and bytes objects are “salted” with an unpredictable random value. Although they remain constant within an individual

Python process, they are not predictable between repeated invocations of Python.

This is intended to provide protection against a denial-of-service caused by carefully chosen inputs that exploit the worst case performance of a dict insertion, $O(n^2)$ complexity. See http://ocert.org/advisories/ocert-2011-003.html for details.

Changing hash values affects the iteration order of sets. Python has never made guarantees about this ordering (and it typically varies between 32-bit and 64-bit builds).

See also **`PYTHONHASHSEED`**.

Changed in version 3.3: Hash randomization is enabled by default.

object.__bool__(*self*)

Called to implement truth value testing and the built-in operation `bool()`; should return `False` or `True`. When this method is not defined, **`__len__()`** is called, if it is defined, and the object is considered true if its result is nonzero. If a class defines neither **`__len__()`** nor **`__bool__()`** (which is true of the **`object`** class itself), all its instances are considered true.

3.3.2. Customizing attribute access

The following methods can be defined to customize the meaning of attribute access (use of, assignment to, or deletion of `x.name`) for class instances.

object.__getattr__(*self*, *name*)

Called when the default attribute access fails with an `AttributeError` (either `__getattribute__()` raises an `AttributeError` because *name* is not an instance attribute or an attribute in the class tree for `self`; or `__get__()` of a *name* property raises `AttributeError`). This method should either return the (computed) attribute value or raise an `AttributeError` exception. The `object` class itself does not provide this method.

Note that if the attribute is found through the normal mechanism, `__getattr__()` is not called. (This is an intentional asymmetry between `__getattr__()` and `__setattr__()`.) This is done both for efficiency reasons and because otherwise `__getattr__()` would have no way to access other attributes of the instance. Note that at least for instance variables, you can take total control by not inserting any values in the instance attribute dictionary (but instead inserting them in another object). See the `__getattribute__()` method below for a way to actually get total control over attribute access.

object.__getattribute__(*self*, *name*)

Called unconditionally to implement attribute accesses for instances of the class. If the class also defines `__getattr__()`, the latter will not be called unless `__getattribute__()` either calls it explicitly or raises an `AttributeError`. This method should return the (computed) attribute value or raise an `AttributeError` exception. In order to avoid infinite recursion in this method, its implementation should always call the base class method with the same name to access any attributes it needs, for example, `object.__getattribute__(self, name)`.

Note

> This method may still be bypassed when looking up special methods as the result of implicit invocation via language syntax or built-in functions. See Special method lookup.

For certain sensitive attribute accesses, raises an auditing event `object.__getattr__` with arguments `obj` and `name`.

object.__setattr__(*self*, *name*, *value*)

Called when an attribute assignment is attempted. This is called instead of the normal mechanism (i.e. store the value in the instance dictionary). *name* is the attribute name, *value* is the value to be assigned to it.

If **`__setattr__()`** wants to assign to an instance attribute, it should call the base class method with the same name, for example, `object.__setattr__(self, name, value)`.

For certain sensitive attribute assignments, raises an auditing event `object.__setattr__` with arguments `obj`, `name`, `value`.

object.__delattr__(*self*, *name*)

Like **`__setattr__()`** but for attribute deletion instead of assignment. This should only be implemented if `del obj.name` is meaningful for the object.

For certain sensitive attribute deletions, raises an auditing event `object.__delattr__` with arguments `obj` and `name`.

object.__dir__(*self*)

Called when **`dir()`** is called on the object. An iterable must be returned. **`dir()`** converts the returned iterable to a list and sorts it.

3.3.2.1. Customizing module attribute access

module.__getattr__()
module.__dir__()

Special names `__getattr__` and `__dir__` can be also used to customize access to module attributes. The `__getattr__` function at the module level should accept one argument which is the name of an attribute and return the computed value or raise an **`AttributeError`**. If an attribute is not found on a module object through the normal lookup, i.e. **`object.__getattribute__()`**, then `__getattr__` is searched in the module `__dict__` before raising an **`AttributeError`**. If found, it is called with the attribute name and the result is returned.

The `__dir__` function should accept no arguments, and return an iterable of strings that represents the names accessible on module. If present, this function overrides the standard **`dir()`** search on a module.

module.__class__

For a more fine grained customization of the module behavior (setting attributes, properties, etc.), one can set the `__class__` attribute of a module object to a subclass of **`types.ModuleType`**. For example:

```
import sys
from types import ModuleType

class VerboseModule(ModuleType):
    def __repr__(self):
        return f'Verbose {self.__name__}'

    def __setattr__(self, attr, value):
        print(f'Setting {attr}...')
        super().__setattr__(attr, value)

sys.modules[__name__].__class__ = VerboseModule
```

Note

Defining module `__getattr__` and setting module `__class__` only affect lookups made using the attribute access syntax – directly accessing the module globals (whether by code within the module, or via a reference to the module's globals dictionary) is unaffected.

Changed in version 3.5: `__class__` module attribute is now writable.

Added in version 3.7: `__getattr__` and `__dir__` module attributes.

See also

PEP 562 [https://peps.python.org/pep-0562/] - Module __getattr__ and __dir__
Describes the `__getattr__` and `__dir__` functions on modules.

3.3.2.2. Implementing Descriptors

The following methods only apply when an instance of the class containing the method (a so-called *descriptor* class) appears in an *owner* class (the descriptor must be in either the owner's class dictionary or in the class dictionary for one of its parents). In the examples below, "the attribute" refers to the attribute whose name is the key of the property in the owner class’ `__dict__`. The `object` class itself does not implement any of these protocols.

object.__get__(*self*, *instance*, *owner=None*)

Called to get the attribute of the owner class (class attribute access) or of an instance of that class (instance attribute access). The optional *owner* argument is the owner class, while *instance* is the instance that the attribute was accessed through, or `None` when the attribute is accessed through the *owner*.

This method should return the computed attribute value or raise an `AttributeError` exception.

PEP 252 [https://peps.python.org/pep-0252/] specifies that `__get__()` is callable with one or two arguments. Python’s own built-in descriptors support this specification; however, it is likely that some third-party tools have descriptors that require both arguments. Python’s own `__getattribute__()` implementation always passes in both arguments whether they are required or not.

object.__set__(*self*, *instance*, *value*)

Called to set the attribute on an instance *instance* of the owner class to a new value, *value*.

Note, adding `__set__()` or `__delete__()` changes the kind of descriptor to a “data descriptor”. See Invoking Descriptors for more details.

object.__delete__(*self*, *instance*)

Called to delete the attribute on an instance *instance* of the owner class.

Instances of descriptors may also have the **`__objclass__`** attribute present:

object.__objclass__

The attribute **`__objclass__`** is interpreted by the `inspect` module as specifying the class where this object was defined (setting this appropriately can assist in runtime introspection of dynamic class attributes). For callables, it may indicate that an instance of the given type (or a subclass) is expected or required as the first positional argument (for example, CPython sets this attribute for unbound methods that are implemented in C).

3.3.2.3. Invoking Descriptors

In general, a descriptor is an object attribute with “binding behavior”, one whose attribute access has been overridden by methods in the descriptor protocol: `__get__()`, `__set__()`, and `__delete__()`. If any of those methods are defined for an object, it is said to be a descriptor.

The default behavior for attribute access is to get, set, or delete the attribute from an object’s dictionary. For instance, `a.x` has a lookup chain starting with `a.__dict__['x']`, then `type(a).__dict__['x']`, and continuing through the base classes of `type(a)` excluding metaclasses.

However, if the looked-up value is an object defining one of the descriptor methods, then Python may override the default behavior and invoke the descriptor method instead. Where this occurs in the precedence chain depends on which descriptor methods were defined and how they were called.

The starting point for descriptor invocation is a binding, `a.x`. How the arguments are assembled depends on `a`:

Direct Call

The simplest and least common call is when user code directly invokes a descriptor method: `x.__get__(a)`.

Instance Binding

If binding to an object instance, `a.x` is transformed into the call: `type(a).__dict__['x'].__get__(a, type(a))`.

Class Binding

If binding to a class, `A.x` is transformed into the call: `A.__dict__['x'].__get__(None, A)`.

Super Binding

A dotted lookup such as `super(A, a).x` searches `a.__class__.__mro__` for a base class `B` following `A` and then returns `B.__dict__['x'].__get__(a, A)`. If not a descriptor, `x` is returned unchanged.

For instance bindings, the precedence of descriptor invocation depends on which descriptor methods are defined. A descriptor can define any combination of `__get__()`, `__set__()` and `__delete__()`. If it does not define **`__get__()`**, then accessing the attribute will return the descriptor object itself unless there is a value in the object's instance dictionary. If the descriptor defines **`__set__()`** and/or **`__delete__()`**, it is a data descriptor; if it defines neither, it is a non-data descriptor. Normally, data descriptors define both **`__get__()`** and **`__set__()`**, while non-data descriptors have just the **`__get__()`** method. Data descriptors with **`__get__()`** and **`__set__()`** (and/or **`__delete__()`**) defined always override a redefinition in an instance dictionary. In contrast, non-data descriptors can be overridden by instances.

Python methods (including those decorated with `@staticmethod` and `@classmethod`) are implemented as non-data descriptors. Accordingly, instances can redefine and override methods. This allows individual instances to acquire behaviors that differ from other instances of the same class.

The `property()` function is implemented as a data descriptor. Accordingly, instances cannot override the behavior of a property.

3.3.2.4. __slots__

__*slots*__ allow us to explicitly declare data members (like properties) and deny the creation of `__dict__` and __*weakref*__ (unless explicitly declared in __*slots*__ or available in a parent.)

The space saved over using `__dict__` can be significant. Attribute lookup speed can be significantly improved as well.

object.__slots__

> This class variable can be assigned a string, iterable, or sequence of strings with variable names used by instances. __*slots*__ reserves space for the declared variables and prevents the automatic creation of `__dict__` and __*weakref*__ for each instance.

Notes on using __*slots*__:

- When inheriting from a class without __*slots*__, the `__dict__` and __*weakref*__ attribute of the instances will always be accessible.
- Without a `__dict__` variable, instances cannot be assigned new variables not listed in the __*slots*__ definition. Attempts to assign to an unlisted variable name raises `AttributeError`. If dynamic assignment of new variables is desired, then add `'__dict__'` to the sequence of strings in the __*slots*__ declaration.

- Without a __*weakref*__ variable for each instance, classes defining __*slots*__ do not support **`weak references`** to its instances. If weak reference support is needed, then add `'__weakref__'` to the sequence of strings in the __*slots*__ declaration.
- __*slots*__ are implemented at the class level by creating descriptors for each variable name. As a result, class attributes cannot be used to set default values for instance variables defined by __*slots*__; otherwise, the class attribute would overwrite the descriptor assignment.
- The action of a __*slots*__ declaration is not limited to the class where it is defined. __*slots*__ declared in parents are available in child classes. However, instances of a child subclass will get a **`__dict__`** and __*weakref*__ unless the subclass also defines __*slots*__ (which should only contain names of any *additional* slots).
- If a class defines a slot also defined in a base class, the instance variable defined by the base class slot is inaccessible (except by retrieving its descriptor directly from the base class). This renders the meaning of the program undefined. In the future, a check may be added to prevent this.
- **`TypeError`** will be raised if nonempty __*slots*__ are defined for a class derived from a **`"variable-length" built-in type`** such as **`int`**, **`bytes`**, and **`tuple`**.
- Any non-string iterable may be assigned to __*slots*__.
- If a **`dictionary`** is used to assign __*slots*__, the dictionary keys will be used as the slot names. The values of the dictionary can be used to provide per-attribute docstrings that will be recognised by **`inspect.getdoc()`** and displayed in the output of **`help()`**.
- **`__class__`** assignment works only if both classes have the same __*slots*__.
- Multiple inheritance with multiple slotted parent classes can be used, but only one parent is allowed to have

attributes created by slots (the other bases must have empty slot layouts) - violations raise `TypeError`.
- If an iterator is used for *__slots__* then a descriptor is created for each of the iterator's values. However, the *__slots__* attribute will be an empty iterator.

3.3.3. Customizing class creation

Whenever a class inherits from another class, `__init_subclass__()` is called on the parent class. This way, it is possible to write classes which change the behavior of subclasses. This is closely related to class decorators, but where class decorators only affect the specific class they're applied to, `__init_subclass__` solely applies to future subclasses of the class defining the method.

classmethod object.__init_subclass__(*cls*)

This method is called whenever the containing class is subclassed. *cls* is then the new subclass. If defined as a normal instance method, this method is implicitly converted to a class method.

Keyword arguments which are given to a new class are passed to the parent class's `__init_subclass__`. For compatibility with other classes using `__init_subclass__`, one should take out the needed keyword arguments and pass the others over to the base class, as in:

```
class Philosopher:
    def __init_subclass__(cls, /, default_name, **kwargs):
        super().__init_subclass__(**kwargs)
        cls.default_name = default_name

class AustralianPhilosopher(Philosopher,
default_name="Bruce"):
    pass
```

The default implementation `object.__init_subclass__` does nothing, but raises an error if it is called with any arguments.

Note

The metaclass hint `metaclass` is consumed by the rest of the type machinery, and is never passed to `__init_subclass__` implementations. The actual metaclass (rather than the explicit hint) can be accessed as `type(cls)`.

Added in version 3.6.

When a class is created, **`type.__new__()`** scans the class variables and makes callbacks to those with a [**`__set_name__()`**]() hook.

object.__set_name__(*self*, *owner*, *name*)

Automatically called at the time the owning class *owner* is created. The object has been assigned to *name* in that class:

```
class A:
    x = C()  # Automatically calls: x.__set_name__(A, 'x')
```

If the class variable is assigned after the class is created, **`__set_name__()`** will not be called automatically. If needed, **`__set_name__()`** can be called directly:

```
class A:
   pass

c = C()
A.x = c                  # The hook is not called
c.__set_name__(A, 'x')   # Manually invoke the hook
```

See Creating the class object for more details.

Added in version 3.6.

3.3.3.1. Metaclasses

By default, classes are constructed using `type()`. The class body is executed in a new namespace and the class name is bound locally to the result of `type(name, bases, namespace)`.

The class creation process can be customized by passing the `metaclass` keyword argument in the class definition line, or by inheriting from an existing class that included such an argument. In the following example, both `MyClass` and `MySubclass` are instances of `Meta`:

```
class Meta(type):
    pass

class MyClass(metaclass=Meta):
    pass

class MySubclass(MyClass):
    pass
```

Any other keyword arguments that are specified in the class definition are passed through to all metaclass operations described below.

When a class definition is executed, the following steps occur:

- MRO entries are resolved;
- the appropriate metaclass is determined;
- the class namespace is prepared;
- the class body is executed;
- the class object is created.

3.3.3.2. Resolving MRO entries

object.__mro_entries__(*self*, *bases*)

If a base that appears in a class definition is not an instance of `type`, then an `__mro_entries__()` method is searched on the base. If an `__mro_entries__()` method is found, the base is substituted with the result of a call to `__mro_entries__()` when creating the class. The method is called with the original bases tuple passed to the *bases* parameter, and must return a tuple of classes that will be used instead of the base. The returned tuple may be empty: in these cases, the original base is ignored.

See also

`types.resolve_bases()`

Dynamically resolve bases that are not instances of `type`.

`types.get_original_bases()`

Retrieve a class's "original bases" prior to modifications by `__mro_entries__()`.

PEP 560 **[https://peps.python.org/pep-0560/]**

Core support for typing module and generic types.

3.3.3.3. Determining the appropriate metaclass

The appropriate metaclass for a class definition is determined as follows:

- if no bases and no explicit metaclass are given, then `type()` is used;
- if an explicit metaclass is given and it is *not* an instance of `type()`, then it is used directly as the metaclass;

- if an instance of `type()` is given as the explicit metaclass, or bases are defined, then the most derived metaclass is used.

The most derived metaclass is selected from the explicitly specified metaclass (if any) and the metaclasses (i.e. `type(cls)`) of all specified base classes. The most derived metaclass is one which is a subtype of *all* of these candidate metaclasses. If none of the candidate metaclasses meets that criterion, then the class definition will fail with `TypeError`.

3.3.3.4. Preparing the class namespace

Once the appropriate metaclass has been identified, then the class namespace is prepared. If the metaclass has a `__prepare__` attribute, it is called as `namespace = metaclass.__prepare__(name, bases, **kwds)` (where the additional keyword arguments, if any, come from the class definition). The `__prepare__` method should be implemented as a `classmethod`. The namespace returned by `__prepare__` is passed in to `__new__`, but when the final class object is created the namespace is copied into a new `dict`.

If the metaclass has no `__prepare__` attribute, then the class namespace is initialised as an empty ordered mapping.

See also

PEP 3115 [https://peps.python.org/pep-3115/] - Metaclasses in Python 3000
Introduced the `__prepare__` namespace hook

3.3.3.5. Executing the class body

The class body is executed (approximately) as `exec(body, globals(), namespace)`. The key difference from a normal call to **`exec()`** is that lexical scoping allows the class body (including any methods) to reference names from the current and outer scopes when the class definition occurs inside a function.

However, even when the class definition occurs inside the function, methods defined inside the class still cannot see names defined at the class scope. Class variables must be accessed through the first parameter of instance or class methods, or through the implicit lexically scoped `__class__` reference described in the next section.

3.3.3.6. Creating the class object

Once the class namespace has been populated by executing the class body, the class object is created by calling `metaclass(name, bases, namespace, **kwds)` (the additional keywords passed here are the same as those passed to `__prepare__`).

This class object is the one that will be referenced by the zero-argument form of **`super()`**. `__class__` is an implicit closure reference created by the compiler if any methods in a class body refer to either `__class__` or `super`. This allows the zero argument form of **`super()`** to correctly identify the class being defined based on lexical scoping, while the class or instance that was used to make the current call is identified based on the first argument passed to the method.

CPython implementation detail: In CPython 3.6 and later, the `__class__` cell is passed to the metaclass as a `__classcell__` entry in the class namespace. If present, this must be propagated up to the `type.__new__` call in order for the class to be initialised correctly. Failing to do so will result in a `RuntimeError` in Python 3.8.

When using the default metaclass `type`, or any metaclass that ultimately calls `type.__new__`, the following additional customization steps are invoked after creating the class object:

1. The `type.__new__` method collects all of the attributes in the class namespace that define a `__set_name__()` method;
2. Those `__set_name__` methods are called with the class being defined and the assigned name of that particular attribute;
3. The `__init_subclass__()` hook is called on the immediate parent of the new class in its method resolution order.

After the class object is created, it is passed to the class decorators included in the class definition (if any) and the resulting object is bound in the local namespace as the defined class.

When a new class is created by `type.__new__`, the object provided as the namespace parameter is copied to a new ordered mapping and the original object is discarded. The new copy is wrapped in a read-only proxy, which becomes the `__dict__` attribute of the class object.

See also

PEP 3135 [https://peps.python.org/pep-3135/] - New super
Describes the implicit `__class__` closure reference

3.3.3.7. Uses for metaclasses

The potential uses for metaclasses are boundless. Some ideas that have been explored include enum, logging, interface checking, automatic delegation, automatic property creation, proxies, frameworks, and automatic resource locking/synchronization.

3.3.4. Customizing instance and subclass checks

The following methods are used to override the default behavior of the `isinstance()` and `issubclass()` built-in functions.

In particular, the metaclass `abc.ABCMeta` implements these methods in order to allow the addition of Abstract Base Classes (ABCs) as “virtual base classes” to any class or type (including built-in types), including other ABCs.

type.__instancecheck__(*self*, *instance*)

> Return true if *instance* should be considered a (direct or indirect) instance of *class*. If defined, called to implement `isinstance(instance, class)`.

type.__subclasscheck__(*self*, *subclass*)

> Return true if *subclass* should be considered a (direct or indirect) subclass of *class*. If defined, called to implement `issubclass(subclass, class)`.

Note that these methods are looked up on the type (metaclass) of a class. They cannot be defined as class methods in the actual class. This is consistent with the lookup of special methods that are called on instances, only in this case the instance is itself a class.

See also

PEP 3119 [https://peps.python.org/pep-3119/] - Introducing Abstract Base Classes

Includes the specification for customizing `isinstance()` and `issubclass()` behavior through `__instancecheck__()` and `__subclasscheck__()`, with motivation for this functionality in the context of adding Abstract Base Classes (see the `abc` module) to the language.

3.3.5. Emulating generic types

When using type annotations, it is often useful to *parameterize* a generic type using Python's square-brackets notation. For example, the annotation `list[int]` might be used to signify a `list` in which all the elements are of type `int`.

See also

PEP 484 [https://peps.python.org/pep-0484/] - Type Hints

Introducing Python's framework for type annotations

Generic Alias Types

Documentation for objects representing parameterized generic classes

Generics, user-defined generics and `typing.Generic`

Documentation on how to implement generic classes that can be parameterized at runtime and understood by static type-checkers.

A class can *generally* only be parameterized if it defines the special class method `__class_getitem__()`.

classmethod object.__class_getitem__(*cls*, *key*)

Return an object representing the specialization of a generic class by type arguments found in *key*.

When defined on a class, `__class_getitem__()` is automatically a class method. As such, there is no need for it to be decorated with **`@classmethod`** when it is defined.

3.3.5.1. The purpose of ***__class_getitem__***

The purpose of **`__class_getitem__()`** is to allow runtime parameterization of standard-library generic classes in order to more easily apply type hints to these classes.

To implement custom generic classes that can be parameterized at runtime and understood by static type-checkers, users should either inherit from a standard library class that already implements **`__class_getitem__()`**, or inherit from **`typing.Generic`**, which has its own implementation of `__class_getitem__()`.

Custom implementations of **`__class_getitem__()`** on classes defined outside of the standard library may not be understood by third-party type-checkers such as mypy. Using `__class_getitem__()` on any class for purposes other than type hinting is discouraged.

3.3.5.2. ***__class_getitem__*** versus ***__getitem__***

Usually, the subscription of an object using square brackets will call the **`__getitem__()`** instance method defined on the object’s class. However, if the object being subscribed is itself a class, the class method **`__class_getitem__()`** may be called instead. `__class_getitem__()` should return a GenericAlias object if it is properly defined.

Presented with the expression `obj[x]`, the Python interpreter follows something like the following process to decide whether **`__getitem__()`** or **`__class_getitem__()`** should be called:

```
from inspect import isclass

def subscribe(obj, x):
    """Return the result of the expression 'obj[x]'"""

    class_of_obj = type(obj)

    # If the class of obj defines __getitem__,
    # call class_of_obj.__getitem__(obj, x)
    if hasattr(class_of_obj, '__getitem__'):
        return class_of_obj.__getitem__(obj, x)

    # Else, if obj is a class and defines __class_getitem__,
    # call obj.__class_getitem__(x)
    elif isclass(obj) and hasattr(obj, '__class_getitem__'):
        return obj.__class_getitem__(x)

    # Else, raise an exception
    else:
        raise TypeError(
            f"'{class_of_obj.__name__}' object is not
subscriptable"
        )
```

In Python, all classes are themselves instances of other classes. The class of a class is known as that class's metaclass, and most classes have the **`type`** class as their metaclass. **`type`** does not define **`__getitem__()`**, meaning that expressions such as `list[int]`, `dict[str, float]` and `tuple[str, bytes]` all result in **`__class_getitem__()`** being called:

```
>>> # list has class "type" as its metaclass, like most
classes:
>>> type(list)
<class 'type'>
>>> type(dict) == type(list) == type(tuple) == type(str) ==
type(bytes)
```

```
True
>>> # "list[int]" calls "list.__class_getitem__(int)"
>>> list[int]
list[int]
>>> # list.__class_getitem__ returns a GenericAlias object:
>>> type(list[int])
<class 'types.GenericAlias'>
```

However, if a class has a custom metaclass that defines `__getitem__()`, subscribing the class may result in different behaviour. An example of this can be found in the `enum` module:

```
>>> from enum import Enum
>>> class Menu(Enum):
...     """A breakfast menu"""
...     SPAM = 'spam'
...     BACON = 'bacon'
...
>>> # Enum classes have a custom metaclass:
>>> type(Menu)
<class 'enum.EnumMeta'>
>>> # EnumMeta defines __getitem__,
>>> # so __class_getitem__ is not called,
>>> # and the result is not a GenericAlias object:
>>> Menu['SPAM']
<Menu.SPAM: 'spam'>
>>> type(Menu['SPAM'])
<enum 'Menu'>
```

See also

PEP 560 [https://peps.python.org/pep-0560/] - Core Support for typing module and generic types

Introducing `__class_getitem__()`, and outlining when a subscription results in `__class_getitem__()` being called instead of `__getitem__()`

3.3.6. Emulating callable objects

object.__call__(*self*[, *args...*])

Called when the instance is “called” as a function; if this method is defined, `x(arg1, arg2, ...)` roughly translates to `type(x).__call__(x, arg1, ...)`. The **`object`** class itself does not provide this method.

3.3.7. Emulating container types

The following methods can be defined to implement container objects. None of them are provided by the **`object`** class itself. Containers usually are sequences (such as **`lists`** or **`tuples`**) or mappings (like dictionaries), but can represent other containers as well. The first set of methods is used either to emulate a sequence or to emulate a mapping; the difference is that for a sequence, the allowable keys should be the integers *k* for which `0 <= k < N` where *N* is the length of the sequence, or **`slice`** objects, which define a range of items. It is also recommended that mappings provide the methods **`keys()`**, **`values()`**, **`items()`**, **`get()`**, **`clear()`**, **`setdefault()`**, **`pop()`**, **`popitem()`**, **`copy()`**, and **`update()`** behaving similar to those for Python’s standard **`dictionary`** objects. The **`collections.abc`** module provides a **`MutableMapping`** abstract base class to help create those methods from a base set of **`__getitem__()`**, **`__setitem__()`**, **`__delitem__()`**, and **`keys()`**.

Mutable sequences should provide methods **`append()`**, **`clear()`**, **`count()`**, **`extend()`**, **`index()`**, **`insert()`**, **`pop()`**, **`remove()`**, and **`reverse()`**, like Python standard **`list`** objects. Finally, sequence types should implement addition (meaning concatenation) and multiplication (meaning repetition) by defining the methods **`__add__()`**, **`__radd__()`**, **`__iadd__()`**, **`__mul__()`**, **`__rmul__()`** and **`__imul__()`** described below; they should not define other numerical operators.

It is recommended that both mappings and sequences implement the `__contains__()` method to allow efficient use of the `in` operator; for mappings, `in` should search the mapping's keys; for sequences, it should search through the values. It is further recommended that both mappings and sequences implement the `__iter__()` method to allow efficient iteration through the container; for mappings, **`__iter__()`** should iterate through the object's keys; for sequences, it should iterate through the values.

object.__len__(*self*)

Called to implement the built-in function `len()`. Should return the length of the object, an integer >= 0. Also, an object that doesn't define a `__bool__()` method and whose **`__len__()`** method returns zero is considered to be false in a Boolean context.

CPython implementation detail: In CPython, the length is required to be at most `sys.maxsize`. If the length is larger than **`sys.maxsize`** some features (such as `len()`) may raise `OverflowError`. To prevent raising **`OverflowError`** by truth value testing, an object must define a `__bool__()` method.

object.__length_hint__(*self*)

Called to implement `operator.length_hint()`. Should return an estimated length for the object (which may be greater or less than the actual length). The length must be an integer >= 0. The return value may also be `NotImplemented`, which is treated the same as if the `__length_hint__` method didn't exist at all. This method is purely an optimization and is never required for correctness.

Added in version 3.4.

Note

Slicing is done exclusively with the following three methods. A call like

```
a[1:2] = b
```

is translated to

```
a[slice(1, 2, None)] = b
```

and so forth. Missing slice items are always filled in with `None`.

object.__getitem__(*self*, *key*)

Called to implement evaluation of `self[key]`. For sequence types, the accepted keys should be integers. Optionally, they may support **`slice`** objects as well. Negative index support is also optional. If *key* is of an inappropriate type, **`TypeError`** may be raised; if *key* is a value outside the set of indexes for the sequence (after any special interpretation of negative values), **`IndexError`** should be raised. For mapping types, if *key* is missing (not in the container), **`KeyError`** should be raised.

Note

`for` loops expect that an **`IndexError`** will be raised for illegal indexes to allow proper detection of the end of the sequence.

Note

When subscripting a *class*, the special class method **`__class_getitem__()`** may be called instead of `__getitem__()`. See __class_getitem__ versus __getitem__ for more details.

object.__setitem__(*self*, *key*, *value*)

Called to implement assignment to `self[key]`. Same note as for [`__getitem__()`](). This should only be implemented for mappings if the objects support changes to the values for keys, or if new keys can be added, or for sequences if elements can be replaced. The same exceptions should be raised for improper *key* values as for the [`__getitem__()`]() method.

object.__delitem__(*self*, *key*)

Called to implement deletion of `self[key]`. Same note as for [`__getitem__()`](). This should only be implemented for mappings if the objects support removal of keys, or for sequences if elements can be removed from the sequence. The same exceptions should be raised for improper *key* values as for the [`__getitem__()`]() method.

object.__missing__(*self*, *key*)

Called by `dict`.`__getitem__()` to implement `self[key]` for dict subclasses when key is not in the dictionary.

object.__iter__(*self*)

This method is called when an iterator is required for a container. This method should return a new iterator object that can iterate over all the objects in the container. For mappings, it should iterate over the keys of the container.

object.__reversed__(*self*)

Called (if present) by the `reversed()` built-in to implement reverse iteration. It should return a new iterator object that iterates over all the objects in the container in reverse order.

If the `__reversed__()` method is not provided, the `reversed()` built-in will fall back to using the sequence protocol (`__len__()` and `__getitem__()`). Objects that support the sequence protocol should only provide `__reversed__()` if they can provide an implementation that is more efficient than the one provided by `reversed()`.

The membership test operators (`in` and `not in`) are normally implemented as an iteration through a container. However, container objects can supply the following special method with a more efficient implementation, which also does not require the object be iterable.

object.__contains__(*self*, *item*)

Called to implement membership test operators. Should return true if *item* is in *self*, false otherwise. For mapping objects, this should consider the keys of the mapping rather than the values or the key-item pairs.

For objects that don't define `__contains__()`, the membership test first tries iteration via `__iter__()`, then the old sequence iteration protocol via `__getitem__()`, see this section in the language reference.

3.3.8. Emulating numeric types

The following methods can be defined to emulate numeric objects. Methods corresponding to operations that are not supported by the particular kind of number implemented (e.g., bitwise operations for non-integral numbers) should be left undefined.

object.__add__(*self*, *other*)
object.__sub__(*self*, *other*)
object.__mul__(*self*, *other*)

object.__matmul__(*self*, *other*)
object.__truediv__(*self*, *other*)
object.__floordiv__(*self*, *other*)
object.__mod__(*self*, *other*)
object.__divmod__(*self*, *other*)
object.__pow__(*self*, *other*[, *modulo*])
object.__lshift__(*self*, *other*)
object.__rshift__(*self*, *other*)
object.__and__(*self*, *other*)
object.__xor__(*self*, *other*)
object.__or__(*self*, *other*)

These methods are called to implement the binary arithmetic operations (`+`, `-`, `*`, `@`, `/`, `//`, `%`, `divmod()`, `pow()`, `**`, `<<`, `>>`, `&`, `^`, `|`). For instance, to evaluate the expression `x + y`, where *x* is an instance of a class that has an `__add__()` method, `type(x).__add__(x, y)` is called. The `__divmod__()` method should be the equivalent to using `__floordiv__()` and `__mod__()`; it should not be related to `__truediv__()`. Note that `__pow__()` should be defined to accept an optional third argument if the three-argument version of the built-in `pow()` function is to be supported.

If one of those methods does not support the operation with the supplied arguments, it should return `NotImplemented`.

object.__radd__(*self*, *other*)
object.__rsub__(*self*, *other*)
object.__rmul__(*self*, *other*)
object.__rmatmul__(*self*, *other*)
object.__rtruediv__(*self*, *other*)
object.__rfloordiv__(*self*, *other*)
object.__rmod__(*self*, *other*)

object.__rdivmod__(*self*, *other*)

object.__rpow__(*self*, *other*[, *modulo*])

object.__rlshift__(*self*, *other*)

object.__rrshift__(*self*, *other*)

object.__rand__(*self*, *other*)

object.__rxor__(*self*, *other*)

object.__ror__(*self*, *other*)

These methods are called to implement the binary arithmetic operations (`+`, `-`, `*`, `@`, `/`, `//`, `%`, **`divmod()`**, **`pow()`**, `**`, `<<`, `>>`, `&`, `^`, `|`) with reflected (swapped) operands. These functions are only called if the operands are of different types, when the left operand does not support the corresponding operation [[3]], or the right operand's class is derived from the left operand's class. [[4]] For instance, to evaluate the expression `x - y`, where *y* is an instance of a class that has an **`__rsub__()`** method, `type(y).__rsub__(y, x)` is called if `type(x).__sub__(x, y)` returns **`NotImplemented`** or `type(y)` is a subclass of `type(x)`. [[5]]

Note that **`__rpow__()`** should be defined to accept an optional third argument if the three-argument version of the built-in **`pow()`** function is to be supported.

Changed in version 3.14: Three-argument **`pow()`** now try calling **`__rpow__()`** if necessary. Previously it was only called in two-argument **`pow()`** and the binary power operator.

Note

If the right operand's type is a subclass of the left operand's type and that subclass provides a different implementation of the reflected method for the operation, this method will be called before the left operand’s non-reflected method. This behavior allows subclasses to override their ancestors’ operations.

object.__iadd__(*self*, *other*)
object.__isub__(*self*, *other*)
object.__imul__(*self*, *other*)
object.__imatmul__(*self*, *other*)
object.__itruediv__(*self*, *other*)
object.__ifloordiv__(*self*, *other*)
object.__imod__(*self*, *other*)
object.__ipow__(*self*, *other*[, *modulo*])
object.__ilshift__(*self*, *other*)
object.__irshift__(*self*, *other*)
object.__iand__(*self*, *other*)
object.__ixor__(*self*, *other*)
object.__ior__(*self*, *other*)

These methods are called to implement the augmented arithmetic assignments (`+=`, `-=`, `*=`, `@=`, `/=`, `//=`, `%=`, `**=`, `<<=`, `>>=`, `&=`, `^=`, `|=`). These methods should attempt to do the operation in-place (modifying *self*) and return the result (which could be, but does not have to be, *self*). If a specific method is not defined, or if that method returns **`NotImplemented`**, the augmented assignment falls back to the normal methods. For instance, if *x* is an instance of a class with an **`__iadd__()`** method, `x += y` is equivalent to `x = x.__iadd__(y)` . If **`__iadd__()`** does not exist, or if `x.__iadd__(y)` returns **`NotImplemented`**, `x.__add__(y)` and `y.__radd__(x)` are considered, as with the evaluation of `x + y`. In certain situations, augmented assignment can result in unexpected errors (see Why does a_tuple[i] += [‘item’] raise an exception when the addition works?), but this behavior is in fact part of the data model.

object.__neg__(*self*)
object.__pos__(*self*)
object.__abs__(*self*)
object.__invert__(*self*)

Called to implement the unary arithmetic operations (`-`, `+`, `abs()` and `~`).

object.__complex__(*self*)
object.__int__(*self*)
object.__float__(*self*)

Called to implement the built-in functions `complex()`, `int()` and `float()`. Should return a value of the appropriate type.

object.__index__(*self*)

Called to implement `operator.index()`, and whenever Python needs to losslessly convert the numeric object to an integer object (such as in slicing, or in the built-in `bin()`, `hex()` and `oct()` functions). Presence of this method indicates that the numeric object is an integer type. Must return an integer.

If `__int__()`, `__float__()` and `__complex__()` are not defined then corresponding built-in functions `int()`, `float()` and `complex()` fall back to `__index__()`.

object.__round__(*self*[, *ndigits*])
object.__trunc__(*self*)
object.__floor__(*self*)
object.__ceil__(*self*)

Called to implement the built-in function `round()` and `math` functions `trunc()`, `floor()` and `ceil()`. Unless *ndigits* is passed to `__round__()` all these methods should return the value of the object truncated to an `Integral` (typically an `int`).

Changed in version 3.14: `int()` no longer delegates to the `__trunc__()` method.

3.3.9. With Statement Context Managers

A *context manager* is an object that defines the runtime context to be established when executing a `with` statement. The context manager handles the entry into, and the exit from, the desired runtime context for the execution of the block of code. Context managers are normally invoked using the **`with`** statement (described in section The with statement), but can also be used by directly invoking their methods.

Typical uses of context managers include saving and restoring various kinds of global state, locking and unlocking resources, closing opened files, etc.

For more information on context managers, see Context Manager Types. The `object` class itself does not provide the context manager methods.

object.__enter__(*self*)

Enter the runtime context related to this object. The `with` statement will bind this method’s return value to the target(s) specified in the **`as`** clause of the statement, if any.

object.__exit__(*self*, *exc_type*, *exc_value*, *traceback*)

Exit the runtime context related to this object. The parameters describe the exception that caused the context to be exited. If the context was exited without an exception, all three arguments will be `None`.

If an exception is supplied, and the method wishes to suppress the exception (i.e., prevent it from being propagated), it should return a true value. Otherwise, the exception will be processed normally upon exit from this method.

Note that `__exit__()` methods should not reraise the passed-in exception; this is the caller's responsibility.

See also

PEP 343 [https://peps.python.org/pep-0343/] - The "with" statement

The specification, background, and examples for the Python `with` statement.

3.3.10. Customizing positional arguments in class pattern matching

When using a class name in a pattern, positional arguments in the pattern are not allowed by default, i.e. `case MyClass(x, y)` is typically invalid without special support in `MyClass`. To be able to use that kind of pattern, the class needs to define a *__match_args__* attribute.

object.__match_args__

This class variable can be assigned a tuple of strings. When this class is used in a class pattern with positional arguments, each positional argument will be converted into a keyword argument, using the corresponding value in *__match_args__* as the keyword. The absence of this attribute is equivalent to setting it to `()`.

For example, if `MyClass.__match_args__` is `("left", "center", "right")` that means that `case MyClass(x, y)` is equivalent to `case MyClass(left=x, center=y)`. Note that the number of arguments in the pattern must be smaller than or equal to the number of elements in *__match_args__*; if it is larger, the pattern match attempt will raise a `TypeError`.

Added in version 3.10.

See also

PEP 634 [https://peps.python.org/pep-0634/] - Structural Pattern Matching
The specification for the Python `match` statement.

3.3.11. Emulating buffer types

The buffer protocol provides a way for Python objects to expose efficient access to a low-level memory array. This protocol is implemented by builtin types such as `bytes` and `memoryview`, and third-party libraries may define additional buffer types.

While buffer types are usually implemented in C, it is also possible to implement the protocol in Python.

object.__buffer__(*self*, *flags*)

Called when a buffer is requested from *self* (for example, by the `memoryview` constructor). The *flags* argument is an integer representing the kind of buffer requested, affecting for example whether the returned buffer is read-only or writable. `inspect.BufferFlags` provides a convenient way to interpret the flags. The method must return a `memoryview` object.

object.__release_buffer__(*self*, *buffer*)

Called when a buffer is no longer needed. The *buffer* argument is a `memoryview` object that was previously returned by `__buffer__()`. The method must release any resources associated with the buffer. This method should return `None`. Buffer objects that do not need to perform any cleanup are not required to implement this method.

Added in version 3.12.

See also

PEP 688 [https://peps.python.org/pep-0688/] - Making the buffer protocol accessible in Python
Introduces the Python `__buffer__` and `__release_buffer__` methods.

`collections.abc.Buffer`
ABC for buffer types.

3.3.12. Annotations

Functions, classes, and modules may contain annotations, which are a way to associate information (usually type hints) with a symbol.

object.__annotations__
This attribute contains the annotations for an object. It is lazily evaluated, so accessing the attribute may execute arbitrary code and raise exceptions. If evaluation is successful, the attribute is set to a dictionary mapping from variable names to annotations.

Changed in version 3.14: Annotations are now lazily evaluated.

object.__annotate__(*format*)

An annotate function. Returns a new dictionary object mapping attribute/parameter names to their annotation values.

Takes a format parameter specifying the format in which annotations values should be provided. It must be a member of the **`annotationlib.Format`** enum, or an integer with a value corresponding to a member of the enum.

If an annotate function doesn't support the requested format, it must raise **`NotImplementedError`**. Annotate functions must always support **`VALUE`** format; they must not raise **`NotImplementedError()`** when called with this format.

When called with **`VALUE`** format, an annotate function may raise **`NameError`**; it must not raise **`NameError`** when called requesting any other format.

If an object does not have any annotations, **`__annotate__`** should preferably be set to `None` (it can't be deleted), rather than set to a function that returns an empty dict.

Added in version 3.14.

See also

PEP 649 [https://peps.python.org/pep-0649/] — Deferred evaluation of annotation using descriptors

Introduces lazy evaluation of annotations and the `__annotate__` function.

3.3.13. Special method lookup

For custom classes, implicit invocations of special methods are only guaranteed to work correctly if defined on an object's type, not in the object's instance dictionary. That behaviour is the reason why the following code raises an exception:

```
>>> class C:
...     pass
...
>>> c = C()
>>> c.__len__ = lambda: 5
>>> len(c)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: object of type 'C' has no len()
```

The rationale behind this behaviour lies with a number of special methods such as `__hash__()` and `__repr__()` that are implemented by all objects, including type objects. If the implicit lookup of these methods used the conventional lookup process, they would fail when invoked on the type object itself:

```
>>> 1 .__hash__() == hash(1)
True
>>> int.__hash__() == hash(int)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: descriptor '__hash__' of 'int' object needs an
argument
```

Incorrectly attempting to invoke an unbound method of a class in this way is sometimes referred to as 'metaclass confusion', and is avoided by bypassing the instance when looking up special methods:

```
>>> type(1).__hash__(1) == hash(1)
True
>>> type(int).__hash__(int) == hash(int)
True
```

In addition to bypassing any instance attributes in the interest of correctness, implicit special method lookup generally also bypasses the `__getattribute__()` method even of the object’s metaclass:

```
>>> class Meta(type):
...     def __getattribute__(*args):
...         print("Metaclass getattribute invoked")
...         return type.__getattribute__(*args)
...
>>> class C(object, metaclass=Meta):
...     def __len__(self):
...         return 10
...     def __getattribute__(*args):
...         print("Class getattribute invoked")
...         return object.__getattribute__(*args)
...
>>> c = C()
>>> c.__len__()                 # Explicit lookup via instance
Class getattribute invoked
10
>>> type(c).__len__(c)          # Explicit lookup via type
Metaclass getattribute invoked
10
>>> len(c)                      # Implicit lookup
10
```

Bypassing the `__getattribute__()` machinery in this fashion provides significant scope for speed optimisations within the interpreter, at the cost of some flexibility in the handling of special methods (the special method *must* be set on the class object itself in order to be consistently invoked by the interpreter).

3.4. Coroutines

3.4.1. Awaitable Objects

An awaitable object generally implements an `__await__()` method. Coroutine objects returned from `async def` functions are awaitable.

Note

The generator iterator objects returned from generators decorated with `types.coroutine()` are also awaitable, but they do not implement `__await__()`.

object.__await__(*self*)

Must return an iterator. Should be used to implement awaitable objects. For instance, `asyncio.Future` implements this method to be compatible with the `await` expression. The `object` class itself is not awaitable and does not provide this method.

Note

The language doesn’t place any restriction on the type or value of the objects yielded by the iterator returned by `__await__`, as this is specific to the implementation of the asynchronous execution framework (e.g. `asyncio`) that will be managing the awaitable object.

Added in version 3.5.

See also

PEP 492 [https://peps.python.org/pep-0492/] for additional information about awaitable objects.

3.4.2. Coroutine Objects

Coroutine objects are awaitable objects. A coroutine's execution can be controlled by calling `__await__()` and iterating over the result. When the coroutine has finished executing and returns, the iterator raises `StopIteration`, and the exception's `value` attribute holds the return value. If the coroutine raises an exception, it is propagated by the iterator. Coroutines should not directly raise unhandled `StopIteration` exceptions.

Coroutines also have the methods listed below, which are analogous to those of generators (see Generator-iterator methods). However, unlike generators, coroutines do not directly support iteration.

Changed in version 3.5.2: It is a `RuntimeError` to await on a coroutine more than once.

coroutine.send(*value*)

Starts or resumes execution of the coroutine. If *value* is `None`, this is equivalent to advancing the iterator returned by `__await__()`. If *value* is not `None`, this method delegates to the `send()` method of the iterator that caused the coroutine to suspend. The result (return value, `StopIteration`, or other exception) is the same as when iterating over the **`__await__()`** return value, described above.

coroutine.throw(*value*)

coroutine.throw(*type*[, *value*[, *traceback*]])

Raises the specified exception in the coroutine. This method delegates to the `throw()` method of the iterator that caused the coroutine to suspend, if it has such a method. Otherwise, the exception is raised at the suspension point. The result (return value, `StopIteration`, or other exception) is the same as when iterating over the `__await__()` return value, described above. If the exception is not caught in the coroutine, it propagates back to the caller.

Changed in version 3.12: The second signature (type[, value[, traceback]]) is deprecated and may be removed in a future version of Python.

coroutine.close()

Causes the coroutine to clean itself up and exit. If the coroutine is suspended, this method first delegates to the `close()` method of the iterator that caused the coroutine to suspend, if it has such a method. Then it raises `GeneratorExit` at the suspension point, causing the coroutine to immediately clean itself up. Finally, the coroutine is marked as having finished executing, even if it was never started.

Coroutine objects are automatically closed using the above process when they are about to be destroyed.

3.4.3. Asynchronous Iterators

An *asynchronous iterator* can call asynchronous code in its `__anext__` method.

Asynchronous iterators can be used in an `async for` statement.

The `object` class itself does not provide these methods.

object.__aiter__(*self*)

Must return an *asynchronous iterator* object.

object.__anext__(*self*)

Must return an *awaitable* resulting in a next value of the iterator. Should raise a `StopAsyncIteration` error when the iteration is over.

An example of an asynchronous iterable object:

```
class Reader:
    async def readline(self):
        ...

    def __aiter__(self):
        return self

    async def __anext__(self):
        val = await self.readline()
        if val == b'':
            raise StopAsyncIteration
        return val
```

Added in version 3.5.

Changed in version 3.7: Prior to Python 3.7, `__aiter__()` could return an *awaitable* that would resolve to an asynchronous iterator.

Starting with Python 3.7, `__aiter__()` must return an asynchronous iterator object. Returning anything else will result in a `TypeError` error.

3.4.4. Asynchronous Context Managers

An *asynchronous context manager* is a *context manager* that is able to suspend execution in its `__aenter__` and `__aexit__` methods.

Asynchronous context managers can be used in an `async with` statement.

The `object` class itself does not provide these methods.

object.__aenter__(*self*)

Semantically similar to `__enter__()`, the only difference being that it must return an *awaitable*.

object.__aexit__(*self*, *exc_type*, *exc_value*, *traceback*)

Semantically similar to `__exit__()`, the only difference being that it must return an *awaitable*.

An example of an asynchronous context manager class:

```
class AsyncContextManager:
    async def __aenter__(self):
        await log('entering context')

    async def __aexit__(self, exc_type, exc, tb):
        await log('exiting context')
```

Added in version 3.5.

Footnotes

[1]

It *is* possible in some cases to change an object's type, under certain controlled conditions. It generally isn't a good idea though, since it can lead to some very strange behaviour if it is handled incorrectly.

[2]

The `__hash__()`, `__iter__()`, `__reversed__()`, `__contains__()`, `__class_getitem__()` and `__fspath__()` methods have special handling for this. Others will still raise a `TypeError`, but may do so by relying on the behavior that `None` is not callable.

[3]

“Does not support” here means that the class has no such method, or the method returns **`NotImplemented`**. Do not set the method to `None` if you want to force fallback to the right operand’s reflected method—that will instead have the opposite effect of explicitly *blocking* such fallback.
[4]

For operands of the same type, it is assumed that if the non-reflected method (such as **`__add__()`**) fails then the operation is not supported, which is why the reflected method is not called.

[5]

If the right operand’s type is a subclass of the left operand’s type, the reflected method having precedence allows subclasses to override their ancestors’ operations.

4. Execution model

4.1. Structure of a program

A Python program is constructed from code blocks. A *block* is a piece of Python program text that is executed as a unit. The following are blocks: a module, a function body, and a class definition. Each command typed interactively is a block. A script file (a file given as standard input to the interpreter or specified as a command line argument to the interpreter) is a code block. A script command (a command specified on the interpreter command line with the `-c` option) is a code block. A module run as a top level script (as module `__main__`) from the command line using a `-m` argument is also a code block. The string argument passed to the built-in functions `eval()` and `exec()` is a code block.

A code block is executed in an *execution frame*. A frame contains some administrative information (used for debugging) and determines where and how execution continues after the code block’s execution has completed.

4.2. Naming and binding

4.2.1. Binding of names

Names refer to objects. Names are introduced by name binding operations.

The following constructs bind names:

- formal parameters to functions,
- class definitions,

- function definitions,
- assignment expressions,
- targets that are identifiers if occurring in an assignment:
    - **`for`** loop header,
    - after **`as`** in a **`with`** statement, **`except`** clause, **`except*`** clause, or in the as-pattern in structural pattern matching,
    - in a capture pattern in structural pattern matching
- **`import`** statements.
- **`type`** statements.
- type parameter lists.

The **`import`** statement of the form `from ... import *` binds all names defined in the imported module, except those beginning with an underscore. This form may only be used at the module level.

A target occurring in a **`del`** statement is also considered bound for this purpose (though the actual semantics are to unbind the name).

Each assignment or import statement occurs within a block defined by a class or function definition or at the module level (the top-level code block).

If a name is bound in a block, it is a local variable of that block, unless declared as **`nonlocal`** or **`global`**. If a name is bound at the module level, it is a global variable. (The variables of the module code block are local and global.) If a variable is used in a code block but not defined there, it is a free variable.

Each occurrence of a name in the program text refers to the *binding* of that name established by the following name resolution rules.

4.2.2. Resolution of names

A *scope* defines the visibility of a name within a block. If a local variable is defined in a block, its scope includes that block. If the definition occurs in a function block, the scope extends to any blocks contained within the defining one, unless a contained block introduces a different binding for the name.

When a name is used in a code block, it is resolved using the nearest enclosing scope. The set of all such scopes visible to a code block is called the block's *environment*.

When a name is not found at all, a `NameError` exception is raised. If the current scope is a function scope, and the name refers to a local variable that has not yet been bound to a value at the point where the name is used, an `UnboundLocalError` exception is raised. `UnboundLocalError` is a subclass of `NameError`.

If a name binding operation occurs anywhere within a code block, all uses of the name within the block are treated as references to the current block. This can lead to errors when a name is used within a block before it is bound. This rule is subtle. Python lacks declarations and allows name binding operations to occur anywhere within a code block. The local variables of a code block can be determined by scanning the entire text of the block for name binding operations. See the FAQ entry on UnboundLocalError for examples.

If the `global` statement occurs within a block, all uses of the names specified in the statement refer to the bindings of those names in the top-level namespace. Names are resolved in the top-level namespace by searching the global namespace, i.e. the namespace of the module containing the code block, and the builtins namespace, the namespace of the module `builtins`. The global namespace is searched first. If the names are not found there, the builtins namespace is searched next. If the names are also not found in the builtins namespace, new variables are created in the global namespace. The global statement must precede all uses of the listed names.

The `global` statement has the same scope as a name binding operation in the same block. If the nearest enclosing scope for a free variable contains a global statement, the free variable is treated as a global.

The `nonlocal` statement causes corresponding names to refer to previously bound variables in the nearest enclosing function scope. `SyntaxError` is raised at compile time if the given name does not exist in any enclosing function scope. Type parameters cannot be rebound with the **`nonlocal`** statement.

The namespace for a module is automatically created the first time a module is imported. The main module for a script is always called `__main__`.

Class definition blocks and arguments to `exec()` and `eval()` are special in the context of name resolution. A class definition is an executable statement that may use and define names. These references follow the normal rules for name resolution with an exception that unbound local variables are looked up in the global namespace. The namespace of the class definition becomes the attribute dictionary of the class. The scope of names defined in a class block is limited to the class block; it does not extend to the code blocks of methods. This includes comprehensions and generator expressions, but it does not include annotation scopes, which have access to their enclosing class scopes. This means that the following will fail:

```
class A:
    a = 42
    b = list(a + i for i in range(10))
```

However, the following will succeed:

```
class A:
    type Alias = Nested
    class Nested: pass

print(A.Alias.__value__)  # <type 'A.Nested'>
```

4.2.3. Annotation scopes

Annotations, type parameter lists and **`type`** statements introduce *annotation scopes*, which behave mostly like function scopes, but with some exceptions discussed below.

Annotation scopes are used in the following contexts:

- Function annotations.
- Variable annotations.
- Type parameter lists for generic type aliases.
- Type parameter lists for generic functions. A generic function’s annotations are executed within the annotation scope, but its defaults and decorators are not.
- Type parameter lists for generic classes. A generic class’s base classes and keyword arguments are executed within the annotation scope, but its decorators are not.
- The bounds, constraints, and default values for type parameters (lazily evaluated).
- The value of type aliases (lazily evaluated).

Annotation scopes differ from function scopes in the following ways:

- Annotation scopes have access to their enclosing class namespace. If an annotation scope is immediately within a class scope, or within another annotation scope that is immediately within a class scope, the code in the annotation scope can use names defined in the class scope as if it were executed directly within the class body. This contrasts with regular functions defined within classes, which cannot access names defined in the class scope.
- Expressions in annotation scopes cannot contain **`yield`**, `yield from`, **`await`**, or **`:=`** expressions. (These expressions are allowed in other scopes contained within the annotation scope.)
- Names defined in annotation scopes cannot be rebound with **`nonlocal`** statements in inner scopes. This includes only type parameters, as no other syntactic elements that can appear within annotation scopes can introduce new names.
- While annotation scopes have an internal name, that name is not reflected in the qualified name of objects defined within the scope. Instead, the **`__qualname__`** of such objects is as if the object were defined in the enclosing scope.

Added in version 3.12: Annotation scopes were introduced in Python 3.12 as part of **PEP 695** [https://peps.python.org/pep-0695/].

Changed in version 3.13: Annotation scopes are also used for type parameter defaults, as introduced by **PEP 696** [https://peps.python.org/pep-0696/].

Changed in version 3.14: Annotation scopes are now also used for annotations, as specified in **PEP 649** [https://peps.python.org/pep-0649/] and **PEP 749** [https://peps.python.org/pep-0749/].

4.2.4. Lazy evaluation

Most annotation scopes are *lazily evaluated*. This includes annotations, the values of type aliases created through the `type` statement, and the bounds, constraints, and default values of type variables created through the type parameter syntax. This means that they are not evaluated when the type alias or type variable is created, or when the object carrying annotations is created. Instead, they are only evaluated when necessary, for example when the `__value__` attribute on a type alias is accessed.

Example:

```
>>> type Alias = 1/0
>>> Alias.__value__
Traceback (most recent call last):
  ...
ZeroDivisionError: division by zero
>>> def func[T: 1/0](): pass
>>> T = func.__type_params__[0]
>>> T.__bound__
Traceback (most recent call last):
  ...
ZeroDivisionError: division by zero
```

Here the exception is raised only when the `__value__` attribute of the type alias or the `__bound__` attribute of the type variable is accessed.

This behavior is primarily useful for references to types that have not yet been defined when the type alias or type variable is created. For example, lazy evaluation enables creation of mutually recursive type aliases:

```
from typing import Literal

type SimpleExpr = int | Parenthesized
type Parenthesized = tuple[Literal["("], Expr, Literal[")"]]
type Expr = SimpleExpr | tuple[SimpleExpr, Literal["+", "-"],
Expr]
```

Lazily evaluated values are evaluated in annotation scope, which means that names that appear inside the lazily evaluated value are looked up as if they were used in the immediately enclosing scope.

Added in version 3.12.

4.2.5. Builtins and restricted execution

CPython implementation detail: Users should not touch `__builtins__`; it is strictly an implementation detail. Users wanting to override values in the builtins namespace should **`import`** the **`builtins`** module and modify its attributes appropriately.

The builtins namespace associated with the execution of a code block is actually found by looking up the name `__builtins__` in its global namespace; this should be a dictionary or a module (in the latter case the module's dictionary is used). By default, when in the **`__main__`** module, `__builtins__` is the built-in module **`builtins`**; when in any other module, `__builtins__` is an alias for the dictionary of the **`builtins`** module itself.

4.2.6. Interaction with dynamic features

Name resolution of free variables occurs at runtime, not at compile time. This means that the following code will print 42:

```
i = 10
def f():
    print(i)
```

```
i = 42
f()
```

The `eval()` and `exec()` functions do not have access to the full environment for resolving names. Names may be resolved in the local and global namespaces of the caller. Free variables are not resolved in the nearest enclosing namespace, but in the global namespace. [[1]] The `exec()` and `eval()` functions have optional arguments to override the global and local namespace. If only one namespace is specified, it is used for both.

4.3. Exceptions

Exceptions are a means of breaking out of the normal flow of control of a code block in order to handle errors or other exceptional conditions. An exception is *raised* at the point where the error is detected; it may be *handled* by the surrounding code block or by any code block that directly or indirectly invoked the code block where the error occurred.

The Python interpreter raises an exception when it detects a run-time error (such as division by zero). A Python program can also explicitly raise an exception with the `raise` statement. Exception handlers are specified with the `try` … `except` statement. The `finally` clause of such a statement can be used to specify cleanup code which does not handle the exception, but is executed whether an exception occurred or not in the preceding code.

Python uses the “termination” model of error handling: an exception handler can find out what happened and continue execution at an outer level, but it cannot repair the cause of the error and retry the failing operation (except by re-entering the offending piece of code from the top).

When an exception is not handled at all, the interpreter terminates execution of the program, or returns to its interactive main loop. In either case, it prints a stack traceback, except when the exception is `SystemExit`.

Exceptions are identified by class instances. The `except` clause is selected depending on the class of the instance: it must reference the class of the instance or a non-virtual base class thereof. The instance can be received by the handler and can carry additional information about the exceptional condition.

Note

Exception messages are not part of the Python API. Their contents may change from one version of Python to the next without warning and should not be relied on by code which will run under multiple versions of the interpreter.

See also the description of the `try` statement in section The try statement and `raise` statement in section The raise statement.

4.4. Runtime Components

4.4.1. General Computing Model

Python’s execution model does not operate in a vacuum. It runs on a host machine and through that host’s runtime environment, including its operating system (OS), if there is one. When a program runs, the conceptual layers of how it runs on the host look something like this:

host machine

- **process** (global resources)
  - **thread** (runs machine code)

Each process represents a program running on the host. Think of each process itself as the data part of its program. Think of the process’ threads as the execution part of the program. This distinction will be important to understand the conceptual Python runtime.

The process, as the data part, is the execution context in which the program runs. It mostly consists of the set of resources assigned to the program by the host, including memory, signals, file handles, sockets, and environment variables.

Processes are isolated and independent from one another. (The same is true for hosts.) The host manages the process’ access to its assigned resources, in addition to coordinating between processes.

Each thread represents the actual execution of the program’s machine code, running relative to the resources assigned to the program’s process. It’s strictly up to the host how and when that execution takes place.

From the point of view of Python, a program always starts with exactly one thread. However, the program may grow to run in multiple simultaneous threads. Not all hosts support multiple threads per process, but most do. Unlike processes, threads in a process are not isolated and independent from one another. Specifically, all threads in a process share all of the process’ resources.

The fundamental point of threads is that each one does *run* independently, at the same time as the others. That may be only conceptually at the same time (“concurrently”) or physically (“in parallel”). Either way, the threads effectively run at a non-synchronized rate.

Note

That non-synchronized rate means none of the process’ memory is guaranteed to stay consistent for the code running in any given thread. Thus multi-threaded programs must take care to coordinate access to intentionally shared resources. Likewise, they must take care to be absolutely diligent about not accessing any *other* resources in multiple threads; otherwise two threads running at the same time might accidentally interfere with each other’s use of some shared data. All this is true for both Python programs and the Python runtime.

The cost of this broad, unstructured requirement is the tradeoff for the kind of raw concurrency that threads provide. The alternative to the required discipline generally means dealing with non-deterministic bugs and data corruption.

4.4.2. Python Runtime Model

The same conceptual layers apply to each Python program, with some extra data layers specific to Python:

- **host machine**
  - **process** (global resources)
    - Python global runtime (*state*)
      - Python interpreter (*state*)
        - **thread** (runs Python bytecode and “C-API”)
          - Python thread *state*

At the conceptual level: when a Python program starts, it looks exactly like that diagram, with one of each. The runtime may grow to include multiple interpreters, and each interpreter may grow to include multiple thread states.

Note

A Python implementation won't necessarily implement the runtime layers distinctly or even concretely. The only exception is places where distinct layers are directly specified or exposed to users, like through the `threading` module.

Note

The initial interpreter is typically called the "main" interpreter. Some Python implementations, like CPython, assign special roles to the main interpreter.

Likewise, the host thread where the runtime was initialized is known as the "main" thread. It may be different from the process' initial thread, though they are often the same. In some cases "main thread" may be even more specific and refer to the initial thread state. A Python runtime might assign specific responsibilities to the main thread, such as handling signals.

As a whole, the Python runtime consists of the global runtime state, interpreters, and thread states. The runtime ensures all that state stays consistent over its lifetime, particularly when used with multiple host threads.

The global runtime, at the conceptual level, is just a set of interpreters. While those interpreters are otherwise isolated and independent from one another, they may share some data or other resources. The runtime is responsible for managing these global resources safely. The actual nature and management of these resources is implementation-specific. Ultimately, the external utility of the global runtime is limited to managing interpreters.

In contrast, an “interpreter” is conceptually what we would normally think of as the (full-featured) “Python runtime”. When machine code executing in a host thread interacts with the Python runtime, it calls into Python in the context of a specific interpreter.

Note

The term “interpreter” here is not the same as the “bytecode interpreter”, which is what regularly runs in threads, executing compiled Python code.

In an ideal world, “Python runtime” would refer to what we currently call “interpreter”. However, it’s been called “interpreter” at least since introduced in 1997 (CPython:a027efa5b [https://github.com/python/cpython/commit/a027efa5b]).

Each interpreter completely encapsulates all of the non-process-global, non-thread-specific state needed for the Python runtime to work. Notably, the interpreter’s state persists between uses. It includes fundamental data like `sys.modules`. The runtime ensures multiple threads using the same interpreter will safely share it between them.

A Python implementation may support using multiple interpreters at the same time in the same process. They are independent and isolated from one another. For example, each interpreter has its own `sys.modules`.

For thread-specific runtime state, each interpreter has a set of thread states, which it manages, in the same way the global runtime contains a set of interpreters. It can have thread states for as many host threads as it needs. It may even have multiple thread states for the same host thread, though that isn’t as common.

Each thread state, conceptually, has all the thread-specific runtime data an interpreter needs to operate in one host thread. The thread state includes the current raised exception and the thread’s Python call stack. It may include other thread-specific resources.

Note

The term “Python thread” can sometimes refer to a thread state, but normally it means a thread created using the `threading` module.

Each thread state, over its lifetime, is always tied to exactly one interpreter and exactly one host thread. It will only ever be used in that thread and with that interpreter.

Multiple thread states may be tied to the same host thread, whether for different interpreters or even the same interpreter. However, for any given host thread, only one of the thread states tied to it can be used by the thread at a time.

Thread states are isolated and independent from one another and don’t share any data, except for possibly sharing an interpreter and objects or other resources belonging to that interpreter.

Once a program is running, new Python threads can be created using the `threading` module (on platforms and Python implementations that support threads). Additional processes can be created using the `os`, `subprocess`, and `multiprocessing` modules. Interpreters can be created and used with the `interpreters` module. Coroutines (async) can be run using `asyncio` in each interpreter, typically only in a single thread (often the main thread).

Footnotes

[1]

This limitation occurs because the code that is executed by these operations is not available at the time the module is compiled.

5. The import system

Python code in one module gains access to the code in another module by the process of importing it. The `import` statement is the most common way of invoking the import machinery, but it is not the only way. Functions such as `importlib.import_module()` and built-in `__import__()` can also be used to invoke the import machinery.

The `import` statement combines two operations; it searches for the named module, then it binds the results of that search to a name in the local scope. The search operation of the **`import`** statement is defined as a call to the `__import__()` function, with the appropriate arguments. The return value of `__import__()` is used to perform the name binding operation of the **`import`** statement. See the **`import`** statement for the exact details of that name binding operation.

A direct call to `__import__()` performs only the module search and, if found, the module creation operation. While certain side-effects may occur, such as the importing of parent packages, and the updating of various caches (including `sys.modules`), only the `import` statement performs a name binding operation.

When an `import` statement is executed, the standard builtin `__import__()` function is called. Other mechanisms for invoking the import system (such as `importlib.import_module()`) may choose to bypass `__import__()` and use their own solutions to implement import semantics.

When a module is first imported, Python searches for the module and if found, it creates a module object [[1]], initializing it. If the named module cannot be found, a

`ModuleNotFoundError` is raised. Python implements various strategies to search for the named module when the import machinery is invoked. These strategies can be modified and extended by using various hooks described in the sections below.

Changed in version 3.3: The import system has been updated to fully implement the second phase of **PEP 302** [https://peps.python.org/pep-0302/]. There is no longer any implicit import machinery - the full import system is exposed through `sys.meta_path`. In addition, native namespace package support has been implemented (see **PEP 420** [https://peps.python.org/pep-0420/]).

5.1. `importlib`

The `importlib` module provides a rich API for interacting with the import system. For example `importlib.import_module()` provides a recommended, simpler API than built-in `__import__()` for invoking the import machinery. Refer to the `importlib` library documentation for additional detail.

5.2. Packages

Python has only one type of module object, and all modules are of this type, regardless of whether the module is implemented in Python, C, or something else. To help organize modules and provide a naming hierarchy, Python has a concept of packages.

You can think of packages as the directories on a file system and modules as files within directories, but don’t take this analogy too literally since packages and modules need not originate from the file system. For the purposes of this documentation, we’ll use this convenient analogy of directories and files. Like file system directories, packages are organized hierarchically, and packages may themselves contain subpackages, as well as regular modules.

It’s important to keep in mind that all packages are modules, but not all modules are packages. Or put another way, packages are just a special kind of module. Specifically, any module that contains a `__path__` attribute is considered a package.

All modules have a name. Subpackage names are separated from their parent package name by a dot, akin to Python’s standard attribute access syntax. Thus you might have a package called **`email`**, which in turn has a subpackage called **`email.mime`** and a module within that subpackage called **`email.mime.text`**.

5.2.1. Regular packages

Python defines two types of packages, regular packages and namespace packages. Regular packages are traditional packages as they existed in Python 3.2 and earlier. A regular package is typically implemented as a directory containing an `__init__.py` file. When a regular package is imported, this `__init__.py` file is implicitly executed, and the objects it defines are bound to names in the package’s namespace. The `__init__.py` file can contain the same Python code that any other module can contain, and Python will add some additional attributes to the module when it is imported.

For example, the following file system layout defines a top level `parent` package with three subpackages:

```
parent/
    __init__.py
    one/
        __init__.py
    two/
        __init__.py
    three/
        __init__.py
```

Importing `parent.one` will implicitly execute `parent/__init__.py` and `parent/one/__init__.py`. Subsequent imports of `parent.two` or `parent.three` will execute `parent/two/__init__.py` and `parent/three/__init__.py` respectively.

5.2.2. Namespace packages

A namespace package is a composite of various portions, where each portion contributes a subpackage to the parent package. Portions may reside in different locations on the file system. Portions may also be found in zip files, on the network, or anywhere else that Python searches during import. Namespace packages may or may not correspond directly to objects on the file system; they may be virtual modules that have no concrete representation.

Namespace packages do not use an ordinary list for their `__path__` attribute. They instead use a custom iterable type which will automatically perform a new search for package portions on the next import attempt within that package if the path of their parent package (or **`sys.path`** for a top level package) changes.

With namespace packages, there is no `parent/__init__.py` file. In fact, there may be multiple `parent` directories found during import search, where each one is provided by a different portion. Thus `parent/one` may not be physically located next to `parent/two`. In this case, Python will create a namespace package for the top-level `parent` package whenever it or one of its subpackages is imported.

See also **PEP 420** [https://peps.python.org/pep-0420/] for the namespace package specification.

5.3. Searching

To begin the search, Python needs the fully qualified name of the module (or package, but for the purposes of this discussion, the difference is immaterial) being imported. This name may come from various arguments to the **`import`** statement, or from the parameters to the **`importlib.import_module()`** or **`__import__()`** functions.

This name will be used in various phases of the import search, and it may be the dotted path to a submodule, e.g. `foo.bar.baz`. In this case, Python first tries to import `foo`, then `foo.bar`, and finally `foo.bar.baz`. If any of the intermediate imports fail, a **`ModuleNotFoundError`** is raised.

5.3.1. The module cache

The first place checked during import search is **`sys.modules`**. This mapping serves as a cache of all modules that have been previously imported, including the intermediate paths. So if `foo.bar.baz` was previously imported, **`sys.modules`** will contain entries for `foo`, `foo.bar`, and `foo.bar.baz`. Each key will have as its value the corresponding module object.

During import, the module name is looked up in **`sys.modules`** and if present, the associated value is the module satisfying the import, and the process completes. However, if the value is `None`, then a **`ModuleNotFoundError`** is raised. If the module name is missing, Python will continue searching for the module.

`sys.modules` is writable. Deleting a key may not destroy the associated module (as other modules may hold references to it), but it will invalidate the cache entry for the named module, causing Python to search anew for the named module upon its next import. The key can also be assigned to `None`, forcing the next import of the module to result in a **`ModuleNotFoundError`**.

Beware though, as if you keep a reference to the module object, invalidate its cache entry in **`sys.modules`**, and then re-import the named module, the two module objects will *not* be the same. By contrast, **`importlib.reload()`** will reuse the *same* module object, and simply reinitialise the module contents by rerunning the module's code.

5.3.2. Finders and loaders

If the named module is not found in **`sys.modules`**, then Python's import protocol is invoked to find and load the module. This protocol consists of two conceptual objects, finders and loaders. A finder's job is to determine whether it can find the named module using whatever strategy it knows about. Objects that implement both of these interfaces are referred to as importers - they return themselves when they find that they can load the requested module.

Python includes a number of default finders and importers. The first one knows how to locate built-in modules, and the second knows how to locate frozen modules. A third default finder searches an import path for modules. The import path is a list of locations that may name file system paths or zip files. It can also be extended to search for any locatable resource, such as those identified by URLs.

The import machinery is extensible, so new finders can be added to extend the range and scope of module searching.

Finders do not actually load modules. If they can find the named module, they return a *module spec*, an encapsulation of the module's import-related information, which the import machinery then uses when loading the module.

The following sections describe the protocol for finders and loaders in more detail, including how you can create and register new ones to extend the import machinery.

Changed in version 3.4: In previous versions of Python, finders returned loaders directly, whereas now they return module specs which *contain* loaders. Loaders are still used during import but have fewer responsibilities.

5.3.3. Import hooks

The import machinery is designed to be extensible; the primary mechanism for this are the *import hooks*. There are two types of import hooks: *meta hooks* and *import path hooks*.

Meta hooks are called at the start of import processing, before any other import processing has occurred, other than **`sys.modules`** cache look up. This allows meta hooks to override **`sys.path`** processing, frozen modules, or even built-in modules. Meta hooks are registered by adding new finder objects to **`sys.meta_path`**, as described below.

Import path hooks are called as part of **`sys.path`** (or `package.__path__`) processing, at the point where their associated path item is encountered. Import path hooks are registered by adding new callables to **`sys.path_hooks`** as described below.

5.3.4. The meta path

When the named module is not found in `sys.modules`, Python next searches `sys.meta_path`, which contains a list of meta path finder objects. These finders are queried in order to see if they know how to handle the named module. Meta path finders must implement a method called `find_spec()` which takes three arguments: a name, an import path, and (optionally) a target module. The meta path finder can use any strategy it wants to determine whether it can handle the named module or not.

If the meta path finder knows how to handle the named module, it returns a spec object. If it cannot handle the named module, it returns `None`. If `sys.meta_path` processing reaches the end of its list without returning a spec, then a `ModuleNotFoundError` is raised. Any other exceptions raised are simply propagated up, aborting the import process.

The `find_spec()` method of meta path finders is called with two or three arguments. The first is the fully qualified name of the module being imported, for example `foo.bar.baz`. The second argument is the path entries to use for the module search. For top-level modules, the second argument is `None`, but for submodules or subpackages, the second argument is the value of the parent package's `__path__` attribute. If the appropriate `__path__` attribute cannot be accessed, a `ModuleNotFoundError` is raised. The third argument is an existing module object that will be the target of loading later. The import system passes in a target module only during reload.

The meta path may be traversed multiple times for a single import request. For example, assuming none of the modules involved has already been cached, importing `foo.bar.baz` will first perform a top level import, calling `mpf.find_spec("foo", None, None)` on each meta path finder (`mpf`). After `foo` has been imported, `foo.bar` will be imported by traversing the meta path a second time, calling `mpf.find_spec("foo.bar", foo.__path__, None)`. Once `foo.bar` has been imported, the final traversal will call `mpf.find_spec("foo.bar.baz", foo.bar.__path__, None)`.

Some meta path finders only support top level imports. These importers will always return `None` when anything other than `None` is passed as the second argument.

Python’s default **`sys.meta_path`** has three meta path finders, one that knows how to import built-in modules, one that knows how to import frozen modules, and one that knows how to import modules from an import path (i.e. the path based finder).

Changed in version 3.4: The **`find_spec()`** method of meta path finders replaced **`find_module()`**, which is now deprecated. While it will continue to work without change, the import machinery will try it only if the finder does not implement **`find_spec()`**.

Changed in version 3.10: Use of **`find_module()`** by the import system now raises **`ImportWarning`**.

Changed in version 3.12: **`find_module()`** has been removed. Use **`find_spec()`** instead.

5.4. Loading

If and when a module spec is found, the import machinery will use it (and the loader it contains) when loading the module. Here is an approximation of what happens during the loading portion of import:

```
module = None
if spec.loader is not None and hasattr(spec.loader,
'create_module'):
    # It is assumed 'exec_module' will also be defined on the
loader.
    module = spec.loader.create_module(spec)
if module is None:
    module = ModuleType(spec.name)
# The import-related module attributes get set here:
_init_module_attrs(spec, module)

if spec.loader is None:
    # unsupported
    raise ImportError
if spec.origin is None and spec.submodule_search_locations is
not None:
    # namespace package
    sys.modules[spec.name] = module
elif not hasattr(spec.loader, 'exec_module'):
    module = spec.loader.load_module(spec.name)
else:
    sys.modules[spec.name] = module
    try:
        spec.loader.exec_module(module)
    except BaseException:
        try:
            del sys.modules[spec.name]
        except KeyError:
            pass
        raise
return sys.modules[spec.name]
```

Note the following details:

- If there is an existing module object with the given name in `sys.modules`, import will have already returned it.
- The module will exist in `sys.modules` before the loader executes the module code. This is crucial because the module code may (directly or indirectly) import itself; adding it to `sys.modules` beforehand prevents unbounded recursion in the worst case and multiple loading in the best.
- If loading fails, the failing module – and only the failing module – gets removed from `sys.modules`. Any module

already in the `sys.modules` cache, and any module that was successfully loaded as a side-effect, must remain in the cache. This contrasts with reloading where even the failing module is left in `sys.modules`.
- After the module is created but before execution, the import machinery sets the import-related module attributes ("_init_module_attrs" in the pseudo-code example above), as summarized in a later section.
- Module execution is the key moment of loading in which the module's namespace gets populated. Execution is entirely delegated to the loader, which gets to decide what gets populated and how.
- The module created during loading and passed to exec_module() may not be the one returned at the end of import [[2]].

Changed in version 3.4: The import system has taken over the boilerplate responsibilities of loaders. These were previously performed by the `importlib.abc.Loader.load_module()` method.

5.4.1. Loaders

Module loaders provide the critical function of loading: module execution. The import machinery calls the `importlib.abc.Loader.exec_module()` method with a single argument, the module object to execute. Any value returned from `exec_module()` is ignored.

Loaders must satisfy the following requirements:

- If the module is a Python module (as opposed to a built-in module or a dynamically loaded extension), the loader should execute the module's code in the module's global name space (`module.__dict__`).

- If the loader cannot execute the module, it should raise an **`ImportError`**, although any other exception raised during **`exec_module()`** will be propagated.

In many cases, the finder and loader can be the same object; in such cases the **`find_spec()`** method would just return a spec with the loader set to `self`.

Module loaders may opt in to creating the module object during loading by implementing a **`create_module()`** method. It takes one argument, the module spec, and returns the new module object to use during loading. `create_module()` does not need to set any attributes on the module object. If the method returns `None`, the import machinery will create the new module itself.

Added in version 3.4: The **`create_module()`** method of loaders.

Changed in version 3.4: The **`load_module()`** method was replaced by **`exec_module()`** and the import machinery assumed all the boilerplate responsibilities of loading.

For compatibility with existing loaders, the import machinery will use the `load_module()` method of loaders if it exists and the loader does not also implement `exec_module()`. However, `load_module()` has been deprecated and loaders should implement `exec_module()` instead.

The `load_module()` method must implement all the boilerplate loading functionality described above in addition to executing the module. All the same constraints apply, with some additional clarification:

- If there is an existing module object with the given name in **`sys.modules`**, the loader must use that existing module. (Otherwise, **`importlib.reload()`** will not work correctly.) If the named module does not exist in

`sys.modules`, the loader must create a new module object and add it to `sys.modules`.
- The module *must* exist in `sys.modules` before the loader executes the module code, to prevent unbounded recursion or multiple loading.
- If loading fails, the loader must remove any modules it has inserted into `sys.modules`, but it must remove **only** the failing module(s), and only if the loader itself has loaded the module(s) explicitly.

Changed in version 3.5: A `DeprecationWarning` is raised when `exec_module()` is defined but `create_module()` is not.

Changed in version 3.6: An `ImportError` is raised when `exec_module()` is defined but `create_module()` is not.

Changed in version 3.10: Use of `load_module()` will raise `ImportWarning`.

5.4.2. Submodules

When a submodule is loaded using any mechanism (e.g. `importlib` APIs, the `import` or `import-from` statements, or built-in `__import__()`) a binding is placed in the parent module’s namespace to the submodule object. For example, if package `spam` has a submodule `foo`, after importing `spam.foo`, `spam` will have an attribute `foo` which is bound to the submodule. Let’s say you have the following directory structure:

```
spam/
    __init__.py
    foo.py
```

and `spam/__init__.py` has the following line in it:

```
from .foo import Foo
```

then executing the following puts name bindings for `foo` and `Foo` in the `spam` module:

```
>>> import spam
>>> spam.foo
<module 'spam.foo' from '/tmp/imports/spam/foo.py'>
>>> spam.Foo
<class 'spam.foo.Foo'>
```

Given Python’s familiar name binding rules this might seem surprising, but it’s actually a fundamental feature of the import system. The invariant holding is that if you have `sys.modules['spam']` and `sys.modules['spam.foo']` (as you would after the above import), the latter must appear as the `foo` attribute of the former.

5.4.3. Module specs

The import machinery uses a variety of information about each module during import, especially before loading. Most of the information is common to all modules. The purpose of a module’s spec is to encapsulate this import-related information on a per-module basis.

Using a spec during import allows state to be transferred between import system components, e.g. between the finder that creates the module spec and the loader that executes it. Most importantly, it allows the import machinery to perform the boilerplate operations of loading, whereas without a module spec the loader had that responsibility.

The module’s spec is exposed as **`module.__spec__`**. Setting **`__spec__`** appropriately applies equally to modules initialized during interpreter startup. The one exception is `__main__`, where **`__spec__`** is set to None in some cases.

See **`ModuleSpec`** for details on the contents of the module spec.

Added in version 3.4.

5.4.4. __path__ attributes on modules

The **`__path__`** attribute should be a (possibly empty) sequence of strings enumerating the locations where the package’s submodules will be found. By definition, if a module has a **`__path__`** attribute, it is a package.

A package’s **`__path__`** attribute is used during imports of its subpackages. Within the import machinery, it functions much the same as **`sys.path`**, i.e. providing a list of locations to search for modules during import. However, **`__path__`** is typically much more constrained than **`sys.path`**.

The same rules used for **`sys.path`** also apply to a package’s **`__path__`**. **`sys.path_hooks`** (described below) are consulted when traversing a package’s **`__path__`**.

A package’s `__init__.py` file may set or alter the package’s **`__path__`** attribute, and this was typically the way namespace packages were implemented prior to **PEP 420** [https://peps.python.org/pep-0420/]. With the adoption of **PEP 420** [https://peps.python.org/pep-0420/], namespace packages no longer need to supply `__init__.py` files containing only **`__path__`** manipulation code; the import machinery automatically sets **`__path__`** correctly for the namespace package.

5.4.5. Module reprs

By default, all modules have a usable repr, however depending on the attributes set above, and in the module’s spec, you can more explicitly control the repr of module objects.

If the module has a spec (`__spec__`), the import machinery will try to generate a repr from it. If that fails or there is no spec, the import system will craft a default repr using whatever information is available on the module. It will try to use the `module.__name__`, `module.__file__`, and `module.__loader__` as input into the repr, with defaults for whatever information is missing.

Here are the exact rules used:

- If the module has a `__spec__` attribute, the information in the spec is used to generate the repr. The “name”, “loader”, “origin”, and “has_location” attributes are consulted.
- If the module has a `__file__` attribute, this is used as part of the module’s repr.
- If the module has no `__file__` but does have a `__loader__` that is not `None`, then the loader’s repr is used as part of the module’s repr.
- Otherwise, just use the module’s `__name__` in the repr.

Changed in version 3.12: Use of **`module_repr()`**, having been deprecated since Python 3.4, was removed in Python 3.12 and is no longer called during the resolution of a module’s repr.

5.4.6. Cached bytecode invalidation

Before Python loads cached bytecode from a `.pyc` file, it checks whether the cache is up-to-date with the source `.py` file. By default, Python does this by storing the source’s last-modified timestamp and size in the cache file when writing it. At runtime, the import system then validates the cache file by checking the stored metadata in the cache file against the source's metadata.

Python also supports "hash-based" cache files, which store a hash of the source file's contents rather than its metadata. There are two variants of hash-based `.pyc` files: checked and unchecked. For checked hash-based `.pyc` files, Python validates the cache file by hashing the source file and comparing the resulting hash with the hash in the cache file. If a checked hash-based cache file is found to be invalid, Python regenerates it and writes a new checked hash-based cache file. For unchecked hash-based `.pyc` files, Python simply assumes the cache file is valid if it exists. Hash-based `.pyc` files validation behavior may be overridden with the **`--check-hash-based-pycs`** flag.

Changed in version 3.7: Added hash-based `.pyc` files. Previously, Python only supported timestamp-based invalidation of bytecode caches.

5.5. The Path Based Finder

As mentioned previously, Python comes with several default meta path finders. One of these, called the path based finder (**`PathFinder`**), searches an import path, which contains a list of path entries. Each path entry names a location to search for modules.

The path based finder itself doesn't know how to import anything. Instead, it traverses the individual path entries, associating each of them with a path entry finder that knows how to handle that particular kind of path.

The default set of path entry finders implement all the semantics for finding modules on the file system, handling special file types such as Python source code (`.py` files), Python byte code (`.pyc` files) and shared libraries (e.g. `.so` files). When supported by the **`zipimport`** module in the standard library, the default path entry finders also handle loading all of these file types (other than shared libraries) from zipfiles.

Path entries need not be limited to file system locations. They can refer to URLs, database queries, or any other location that can be specified as a string.

The path based finder provides additional hooks and protocols so that you can extend and customize the types of searchable path entries. For example, if you wanted to support path entries as network URLs, you could write a hook that implements HTTP semantics to find modules on the web. This hook (a callable) would return a path entry finder supporting the protocol described below, which was then used to get a loader for the module from the web.

A word of warning: this section and the previous both use the term *finder*, distinguishing between them by using the terms meta path finder and path entry finder. These two types of finders are very similar, support similar protocols, and function in similar ways during the import process, but it’s important to keep in mind that they are subtly different. In particular, meta path finders operate at the beginning of the import process, as keyed off the **`sys.meta_path`** traversal.

By contrast, path entry finders are in a sense an implementation detail of the path based finder, and in fact, if the path based finder were to be removed from **`sys.meta_path`**, none of the path entry finder semantics would be invoked.

5.5.1. Path entry finders

The path based finder is responsible for finding and loading Python modules and packages whose location is specified with a string path entry. Most path entries name locations in the file system, but they need not be limited to this.

As a meta path finder, the path based finder implements the **`find_spec()`** protocol previously described, however it exposes additional hooks that can be used to customize how modules are found and loaded from the import path.

Three variables are used by the path based finder, **`sys.path`**, **`sys.path_hooks`** and **`sys.path_importer_cache`**. The `__path__` attributes on package objects are also used. These provide additional ways that the import machinery can be customized.

`sys.path` contains a list of strings providing search locations for modules and packages. It is initialized from the **`PYTHONPATH`** environment variable and various other installation- and implementation-specific defaults. Entries in **`sys.path`** can name directories on the file system, zip files, and potentially other “locations” (see the **`site`** module) that should be searched for modules, such as URLs, or database queries. Only strings should be present on **`sys.path`**; all other data types are ignored.

The path based finder is a meta path finder, so the import machinery begins the import path search by calling the path based finder’s **`find_spec()`** method as described previously. When the `path` argument to **`find_spec()`** is given, it will be a list of string paths to traverse - typically a package’s `__path__` attribute for an import within that package. If the `path` argument is `None`, this indicates a top level import and **`sys.path`** is used.

The path based finder iterates over every entry in the search path, and for each of these, looks for an appropriate path entry finder (**`PathEntryFinder`**) for the path entry. Because this can be an expensive operation (e.g. there may be `stat()` call overheads for this search), the path based finder maintains a cache mapping path entries to path entry finders. This cache is maintained in **`sys.path_importer_cache`** (despite the name, this cache actually stores finder objects rather than being limited to importer objects). In this way, the expensive search for a particular path entry location's path entry finder need only be done once. User code is free to remove cache entries from **`sys.path_importer_cache`** forcing the path based finder to perform the path entry search again.

If the path entry is not present in the cache, the path based finder iterates over every callable in **`sys.path_hooks`**. Each of the path entry hooks in this list is called with a single argument, the path entry to be searched. This callable may either return a path entry finder that can handle the path entry, or it may raise **`ImportError`**. An **`ImportError`** is used by the path based finder to signal that the hook cannot find a path entry finder for that path entry. The exception is ignored and import path iteration continues. The hook should expect either a string or bytes object; the encoding of bytes objects is up to the hook (e.g. it may be a file system encoding, UTF-8, or something else), and if the hook cannot decode the argument, it should raise **`ImportError`**.

If **`sys.path_hooks`** iteration ends with no path entry finder being returned, then the path based finder's **`find_spec()`** method will store `None` in **`sys.path_importer_cache`** (to indicate that there is no finder for this path entry) and return `None`, indicating that this meta path finder could not find the module.

If a path entry finder *is* returned by one of the path entry hook callables on **`sys.path_hooks`**, then the following protocol is used to ask the finder for a module spec, which is then used when loading the module.

The current working directory – denoted by an empty string – is handled slightly differently from other entries on **`sys.path`**. First, if the current working directory cannot be determined or is found not to exist, no value is stored in **`sys.path_importer_cache`**. Second, the value for the current working directory is looked up fresh for each module lookup. Third, the path used for **`sys.path_importer_cache`** and returned by **`importlib.machinery.PathFinder.find_spec()`** will be the actual current working directory and not the empty string.

5.5.2. Path entry finder protocol

In order to support imports of modules and initialized packages and also to contribute portions to namespace packages, path entry finders must implement the **`find_spec()`** method.

`find_spec()` takes two arguments: the fully qualified name of the module being imported, and the (optional) target module. `find_spec()` returns a fully populated spec for the module. This spec will always have “loader” set (with one exception).

To indicate to the import machinery that the spec represents a namespace portion, the path entry finder sets `submodule_search_locations` to a list containing the portion.

Changed in version 3.4: **`find_spec()`** replaced **`find_loader()`** and **`find_module()`**, both of which are now deprecated, but will be used if `find_spec()` is not defined.

Older path entry finders may implement one of these two deprecated methods instead of `find_spec()`. The methods are still respected for the sake of backward compatibility. However, if `find_spec()` is implemented on the path entry finder, the legacy methods are ignored.

`find_loader()` takes one argument, the fully qualified name of the module being imported. `find_loader()` returns a 2-tuple where the first item is the loader and the second item is a namespace portion.

For backwards compatibility with other implementations of the import protocol, many path entry finders also support the same, traditional `find_module()` method that meta path finders support. However path entry finder `find_module()` methods are never called with a `path` argument (they are expected to record the appropriate path information from the initial call to the path hook).

The `find_module()` method on path entry finders is deprecated, as it does not allow the path entry finder to contribute portions to namespace packages. If both `find_loader()` and `find_module()` exist on a path entry finder, the import system will always call `find_loader()` in preference to `find_module()`.

Changed in version 3.10: Calls to **`find_module()`** and **`find_loader()`** by the import system will raise **`ImportWarning`**.

Changed in version 3.12: `find_module()` and `find_loader()` have been removed.

5.6. Replacing the standard import system

The most reliable mechanism for replacing the entire import system is to delete the default contents of `sys.meta_path`, replacing them entirely with a custom meta path hook.

If it is acceptable to only alter the behaviour of import statements without affecting other APIs that access the import system, then replacing the builtin `__import__()` function may be sufficient.

To selectively prevent the import of some modules from a hook early on the meta path (rather than disabling the standard import system entirely), it is sufficient to raise `ModuleNotFoundError` directly from `find_spec()` instead of returning `None`. The latter indicates that the meta path search should continue, while raising an exception terminates it immediately.

5.7. Package Relative Imports

Relative imports use leading dots. A single leading dot indicates a relative import, starting with the current package. Two or more leading dots indicate a relative import to the parent(s) of the current package, one level per dot after the first. For example, given the following package layout:

```
package/
    __init__.py
    subpackage1/
        __init__.py
        moduleX.py
        moduleY.py
    subpackage2/
        __init__.py
        moduleZ.py
    moduleA.py
```

In either `subpackage1/moduleX.py` or `subpackage1/__init__.py`, the following are valid relative imports:

```
from .moduleY import spam
from .moduleY import spam as ham
from . import moduleY
from ..subpackage1 import moduleY
from ..subpackage2.moduleZ import eggs
from ..moduleA import foo
```

Absolute imports may use either the `import <>` or `from <> import <>` syntax, but relative imports may only use the second form; the reason for this is that:

```
import XXX.YYY.ZZZ
```

should expose `XXX.YYY.ZZZ` as a usable expression, but .moduleY is not a valid expression.

5.8. Special considerations for __main__

The **`__main__`** module is a special case relative to Python's import system. As noted elsewhere, the `__main__` module is directly initialized at interpreter startup, much like **`sys`** and **`builtins`**. However, unlike those two, it doesn't strictly qualify as a built-in module. This is because the manner in which `__main__` is initialized depends on the flags and other options with which the interpreter is invoked.

5.8.1. __main__.__spec__

Depending on how **`__main__`** is initialized, `__main__.__spec__` gets set appropriately or to `None`.

When Python is started with the **`-m`** option, `__spec__` is set to the module spec of the corresponding module or package.

`__spec__` is also populated when the `__main__` module is loaded as part of executing a directory, zipfile or other `sys.path` entry.

In the remaining cases `__main__.__spec__` is set to `None`, as the code used to populate the `__main__` does not correspond directly with an importable module:

- interactive prompt
- `-c` option
- running from stdin
- running directly from a source or bytecode file

Note that `__main__.__spec__` is always `None` in the last case, *even if* the file could technically be imported directly as a module instead. Use the `-m` switch if valid module metadata is desired in `__main__`.

Note also that even when `__main__` corresponds with an importable module and `__main__.__spec__` is set accordingly, they're still considered *distinct* modules. This is due to the fact that blocks guarded by `if __name__ == "__main__":` checks only execute when the module is used to populate the `__main__` namespace, and not during normal import.

5.9. References

The import machinery has evolved considerably since Python's early days. The original specification for packages [https://www.python.org/doc/essays/packages/] is still available to read, although some details have changed since the writing of that document.

The original specification for `sys.meta_path` was **PEP 302** [https://peps.python.org/pep-0302/], with subsequent extension in **PEP 420** [https://peps.python.org/pep-0420/].

PEP 420 [https://peps.python.org/pep-0420/] introduced namespace packages for Python 3.3. **PEP 420** [https://peps.python.org/pep-0420/] also introduced the **`find_loader()`** protocol as an alternative to **`find_module()`**.

PEP 366 [https://peps.python.org/pep-0366/] describes the addition of the `__package__` attribute for explicit relative imports in main modules.

PEP 328 [https://peps.python.org/pep-0328/] introduced absolute and explicit relative imports and initially proposed `__name__` for semantics **PEP 366** [https://peps.python.org/pep-0366/] would eventually specify for `__package__`.

PEP 338 [https://peps.python.org/pep-0338/] defines executing modules as scripts.

PEP 451 [https://peps.python.org/pep-0451/] adds the encapsulation of per-module import state in spec objects. It also off-loads most of the boilerplate responsibilities of loaders back onto the import machinery. These changes allow the deprecation of several APIs in the import system and also addition of new methods to finders and loaders.

Footnotes

[1]

See `types.ModuleType`.

[2]

The importlib implementation avoids using the return value directly. Instead, it gets the module object by looking the module name up in `sys.modules`. The indirect effect of this is that an imported module may replace itself in `sys.modules`.

This is implementation-specific behavior that is not guaranteed to work in other Python implementations.

6. Expressions

This chapter explains the meaning of the elements of expressions in Python.

Syntax Notes: In this and the following chapters, extended BNF notation will be used to describe syntax, not lexical analysis. When (one alternative of) a syntax rule has the form

```
name: othername
```

and no semantics are given, the semantics of this form of `name` are the same as for `othername`.

6.1. Arithmetic conversions

When a description of an arithmetic operator below uses the phrase “the numeric arguments are converted to a common real type”, this means that the operator implementation for built-in types works as follows:

- If both arguments are complex numbers, no conversion is performed;
- if either argument is a complex or a floating-point number, the other is converted to a floating-point number;
- otherwise, both must be integers and no conversion is necessary.

Some additional rules apply for certain operators (e.g., a string as a left argument to the ‘%’ operator). Extensions must define their own conversion behavior.

6.2. Atoms

Atoms are the most basic elements of expressions. The simplest atoms are identifiers or literals. Forms enclosed in parentheses, brackets or braces are also categorized syntactically as atoms. The syntax for atoms is:

```
atom:      identifier | literal | enclosure
enclosure: parenth_form | list_display | dict_display |
set_display
           | generator_expression | yield_atom
```

6.2.1. Identifiers (Names)

An identifier occurring as an atom is a name. See section Names (identifiers and keywords) for lexical definition and section Naming and binding for documentation of naming and binding.

When the name is bound to an object, evaluation of the atom yields that object. When a name is not bound, an attempt to evaluate it raises a `NameError` exception.

6.2.1.1. Private name mangling

When an identifier that textually occurs in a class definition begins with two or more underscore characters and does not end in two or more underscores, it is considered a *private name* of that class.

See also

The class specifications.

More precisely, private names are transformed to a longer form before code is generated for them. If the transformed name is longer than 255 characters, implementation-defined truncation may happen.

The transformation is independent of the syntactical context in which the identifier is used but only the following private identifiers are mangled:

- Any name used as the name of a variable that is assigned or read or any name of an attribute being accessed.

  The `__name__` attribute of nested functions, classes, and type aliases is however not mangled.

- The name of imported modules, e.g., `__spam` in `import __spam`. If the module is part of a package (i.e., its name contains a dot), the name is *not* mangled, e.g., the `__foo` in `import __foo.bar` is not mangled.

- The name of an imported member, e.g., `__f` in `from spam import __f`.

The transformation rule is defined as follows:

- The class name, with leading underscores removed and a single leading underscore inserted, is inserted in front of the identifier, e.g., the identifier `__spam` occurring in a class named `Foo`, `_Foo` or `__Foo` is transformed to `_Foo__spam`.
- If the class name consists only of underscores, the transformation is the identity, e.g., the identifier `__spam` occurring in a class named `_` or `__` is left as is.

6.2.2. Literals

Python supports string and bytes literals and various numeric literals:

```
literal: strings | NUMBER
```

Evaluation of a literal yields an object of the given type (string, bytes, integer, floating-point number, complex number) with the given value. The value may be approximated in the case of floating-point and imaginary (complex) literals. See section Literals for details. See section String literal concatenation for details on `strings`.

All literals correspond to immutable data types, and hence the object’s identity is less important than its value. Multiple evaluations of literals with the same value (either the same occurrence in the program text or a different occurrence) may obtain the same object or a different object with the same value.

6.2.2.1. String literal concatenation

Multiple adjacent string or bytes literals (delimited by whitespace), possibly using different quoting conventions, are allowed, and their meaning is the same as their concatenation:

```
>>> "hello" 'world'
"helloworld"
```

Formally:

```
strings: ( STRING | fstring)+ | tstring+
```

This feature is defined at the syntactical level, so it only works with literals. To concatenate string expressions at run time, the ‘+’ operator may be used:

```
>>> greeting = "Hello"
>>> space = " "
>>> name = "Blaise"
>>> print(greeting + space + name)   # not: print(greeting
space name)
Hello Blaise
```

Literal concatenation can freely mix raw strings, triple-quoted strings, and formatted string literals. For example:

```
>>> "Hello" r', ' f"{name}!"
"Hello, Blaise!"
```

This feature can be used to reduce the number of backslashes needed, to split long strings conveniently across long lines, or even to add comments to parts of strings. For example:

```
re.compile("[A-Za-z_]"       # letter or underscore
           "[A-Za-z0-9_]*"   # letter, digit or underscore
          )
```

However, bytes literals may only be combined with other byte literals; not with string literals of any kind. Also, template string literals may only be combined with other template string literals:

```
>>> t"Hello" t"{name}!"
Template(strings=('Hello', '!'), interpolations=(...))
```

6.2.3. Parenthesized forms

A parenthesized form is an optional expression list enclosed in parentheses:

```
parenth_form: "(" [starred_expression] ")"
```

A parenthesized expression list yields whatever that expression list yields: if the list contains at least one comma, it yields a tuple; otherwise, it yields the single expression that makes up the expression list.

An empty pair of parentheses yields an empty tuple object. Since tuples are immutable, the same rules as for literals apply (i.e., two occurrences of the empty tuple may or may not yield the same object).

Note that tuples are not formed by the parentheses, but rather by use of the comma. The exception is the empty tuple, for which parentheses *are* required — allowing unparenthesized “nothing” in expressions would cause ambiguities and allow common typos to pass uncaught.

6.2.4. Displays for lists, sets and dictionaries

For constructing a list, a set or a dictionary Python provides special syntax called “displays”, each of them in two flavors:

- either the container contents are listed explicitly, or
- they are computed via a set of looping and filtering instructions, called a *comprehension*.

Common syntax elements for comprehensions are:

```
comprehension: assignment_expression comp_for
comp_for:      ["async"] "for" target_list "in" or_test [comp_iter]
comp_iter:     comp_for | comp_if
comp_if:       "if" or_test [comp_iter]
```

The comprehension consists of a single expression followed by at least one `for` clause and zero or more `for` or `if` clauses. In this case, the elements of the new container are those that would be produced by considering each of the `for` or `if` clauses a block, nesting from left to right, and evaluating the expression to produce an element each time the innermost block is reached.

However, aside from the iterable expression in the leftmost **`for`** clause, the comprehension is executed in a separate implicitly nested scope. This ensures that names assigned to in the target list don’t “leak” into the enclosing scope.

The iterable expression in the leftmost **`for`** clause is evaluated directly in the enclosing scope and then passed as an argument to the implicitly nested scope. Subsequent **`for`** clauses and any filter condition in the leftmost **`for`** clause cannot be evaluated in the enclosing scope as they may depend on the values obtained from the leftmost iterable. For example: `[x*y for x in range(10) for y in range(x, x+10)]`.

To ensure the comprehension always results in a container of the appropriate type, `yield` and `yield from` expressions are prohibited in the implicitly nested scope.

Since Python 3.6, in an **`async def`** function, an **`async for`** clause may be used to iterate over a asynchronous iterator. A comprehension in an **`async def`** function may consist of either a **`for`** or **`async for`** clause following the leading expression, may contain additional **`for`** or **`async for`** clauses, and may also use **`await`** expressions.

If a comprehension contains **`async for`** clauses, or if it contains **`await`** expressions or other asynchronous comprehensions anywhere except the iterable expression in the leftmost **`for`** clause, it is called an *asynchronous comprehension*. An asynchronous comprehension may suspend the execution of the coroutine function in which it appears. See also **PEP 530** [https://peps.python.org/pep-0530/].

Added in version 3.6: Asynchronous comprehensions were introduced.

Changed in version 3.8: `yield` and `yield from` prohibited in the implicitly nested scope.

Changed in version 3.11: Asynchronous comprehensions are now allowed inside comprehensions in asynchronous functions. Outer comprehensions implicitly become asynchronous.

6.2.5. List displays

A list display is a possibly empty series of expressions enclosed in square brackets:

```
list_display: "[" [flexible_expression_list | comprehension]
"]"
```

A list display yields a new list object, the contents being specified by either a list of expressions or a comprehension. When a comma-separated list of expressions is supplied, its elements are evaluated from left to right and placed into the list object in that order. When a comprehension is supplied, the list is constructed from the elements resulting from the comprehension.

6.2.6. Set displays

A set display is denoted by curly braces and distinguishable from dictionary displays by the lack of colons separating keys and values:

```
set_display: "{" (flexible_expression_list | comprehension) "}"
```

A set display yields a new mutable set object, the contents being specified by either a sequence of expressions or a comprehension. When a comma-separated list of expressions is supplied, its elements are evaluated from left to right and added to the set object. When a comprehension is supplied, the set is constructed from the elements resulting from the comprehension.

An empty set cannot be constructed with `{}`; this literal constructs an empty dictionary.

6.2.7. Dictionary displays

A dictionary display is a possibly empty series of dict items (key/value pairs) enclosed in curly braces:

```
dict_display:       "{" [dict_item_list | dict_comprehension]
"}"
dict_item_list:     dict_item ("," dict_item)* [","]
dict_item:          expression ":" expression | "**" or_expr
dict_comprehension: expression ":" expression comp_for
```

A dictionary display yields a new dictionary object.

If a comma-separated sequence of dict items is given, they are evaluated from left to right to define the entries of the dictionary: each key object is used as a key into the dictionary to store the corresponding value. This means that you can specify the same key multiple times in the dict item list, and the final dictionary’s value for that key will be the last one given.

A double asterisk `**` denotes *dictionary unpacking*. Its operand must be a mapping. Each mapping item is added to the new dictionary. Later values replace values already set by earlier dict items and earlier dictionary unpackings.

Added in version 3.5: Unpacking into dictionary displays, originally proposed by **PEP 448** [https://peps.python.org/pep-0448/].

A dict comprehension, in contrast to list and set comprehensions, needs two expressions separated with a colon followed by the usual “for” and “if” clauses. When the comprehension is run, the resulting key and value elements are inserted in the new dictionary in the order they are produced.

Restrictions on the types of the key values are listed earlier in section The standard type hierarchy. (To summarize, the key type should be hashable, which excludes all mutable objects.) Clashes between duplicate keys are not detected; the last value (textually rightmost in the display) stored for a given key value prevails.

Changed in version 3.8: Prior to Python 3.8, in dict comprehensions, the evaluation order of key and value was not well-defined. In CPython, the value was evaluated before the key. Starting with 3.8, the key is evaluated before the value, as proposed by **PEP 572** [https://peps.python.org/pep-0572/].

6.2.8. Generator expressions

A generator expression is a compact generator notation in parentheses:

```
generator_expression: "(" expression comp_for ")"
```

A generator expression yields a new generator object. Its syntax is the same as for comprehensions, except that it is enclosed in parentheses instead of brackets or curly braces.

Variables used in the generator expression are evaluated lazily when the `__next__()` method is called for the generator object (in the same fashion as normal generators). However, the iterable expression in the leftmost **`for`** clause is immediately evaluated, and the iterator is immediately created for that iterable, so that an error produced while creating the iterator will be emitted at the point where the generator expression is defined, rather than at the point where the first value is retrieved. Subsequent **`for`** clauses and any filter condition in the leftmost **`for`** clause cannot be evaluated in the enclosing scope as they may depend on the values obtained from the leftmost iterable. For example: `(x*y for x in range(10) for y in range(x, x+10))`.

The parentheses can be omitted on calls with only one argument. See section Calls for details.

To avoid interfering with the expected operation of the generator expression itself, `yield` and `yield from` expressions are prohibited in the implicitly defined generator.

If a generator expression contains either **`async for`** clauses or **`await`** expressions it is called an *asynchronous generator expression*. An asynchronous generator expression returns a new asynchronous generator object, which is an asynchronous iterator (see Asynchronous Iterators).

Added in version 3.6: Asynchronous generator expressions were introduced.

Changed in version 3.7: Prior to Python 3.7, asynchronous generator expressions could only appear in **`async def`** coroutines. Starting with 3.7, any function can use asynchronous generator expressions.

Changed in version 3.8: `yield` and `yield from` prohibited in the implicitly nested scope.

6.2.9. Yield expressions

```
yield_atom:       "(" yield_expression ")"
yield_from:       "yield" "from" expression
yield_expression: "yield" yield_list | yield_from
```

The yield expression is used when defining a generator function or an asynchronous generator function and thus can only be used in the body of a function definition. Using a yield expression in a function's body causes that function to be a generator function, and using it in an `async def` function's body causes that coroutine function to be an asynchronous generator function. For example:

```
def gen():  # defines a generator function
    yield 123

async def agen(): # defines an asynchronous generator function
    yield 123
```

Due to their side effects on the containing scope, `yield` expressions are not permitted as part of the implicitly defined scopes used to implement comprehensions and generator expressions.

Changed in version 3.8: Yield expressions prohibited in the implicitly nested scopes used to implement comprehensions and generator expressions.

Generator functions are described below, while asynchronous generator functions are described separately in section Asynchronous generator functions.

When a generator function is called, it returns an iterator known as a generator. That generator then controls the execution of the generator function. The execution starts when one of the generator's methods is called. At that time, the execution proceeds to the first yield expression, where it is suspended again, returning the value of `yield_list` to the generator’s caller, or `None` if `yield_list` is omitted. By suspended, we mean that all local state is retained, including the current bindings of local variables, the instruction pointer, the internal evaluation stack, and the state of any exception handling. When the execution is resumed by calling one of the generator’s methods, the function can proceed exactly as if the yield expression were just another external call. The value of the yield expression after resuming depends on the method which resumed the execution. If `__next__()` is used (typically via either a `for` or the `next()` builtin) then the result is `None`. Otherwise, if `send()` is used, then the result will be the value passed in to that method.

All of this makes generator functions quite similar to coroutines; they yield multiple times, they have more than one entry point and their execution can be suspended. The only difference is that a generator function cannot control where the execution should continue after it yields; the control is always transferred to the generator’s caller.

Yield expressions are allowed anywhere in a `try` construct. If the generator is not resumed before it is finalized (by reaching a zero reference count or by being garbage collected), the generator-iterator’s `close()` method will be called, allowing any pending `finally` clauses to execute.

When `yield from <expr>` is used, the supplied expression must be an iterable. The values produced by iterating that iterable are passed directly to the caller of the current generator’s methods. Any values passed in with `send()` and any exceptions passed in with `throw()` are passed to the underlying iterator if it has the appropriate methods. If this is not the case, then `send()` will raise `AttributeError` or

`TypeError`, while `throw()` will just raise the passed in exception immediately.

When the underlying iterator is complete, the `value` attribute of the raised `StopIteration` instance becomes the value of the yield expression. It can be either set explicitly when raising `StopIteration`, or automatically when the subiterator is a generator (by returning a value from the subgenerator).

Changed in version 3.3: Added `yield from <expr>` to delegate control flow to a subiterator.

The parentheses may be omitted when the yield expression is the sole expression on the right hand side of an assignment statement.

See also

PEP 255 [https://peps.python.org/pep-0255/] - Simple Generators
The proposal for adding generators and the `yield` statement to Python.

PEP 342 [https://peps.python.org/pep-0342/] - Coroutines via Enhanced Generators
The proposal to enhance the API and syntax of generators, making them usable as simple coroutines.

PEP 380 [https://peps.python.org/pep-0380/] - Syntax for Delegating to a Subgenerator
The proposal to introduce the `yield_from` syntax, making delegation to subgenerators easy.

PEP 525 [https://peps.python.org/pep-0525/] - Asynchronous Generators
The proposal that expanded on PEP 492 [https://peps.python.org/pep-0492/] by adding generator

capabilities to coroutine functions.

6.2.9.1. Generator-iterator methods

This subsection describes the methods of a generator iterator. They can be used to control the execution of a generator function.

Note that calling any of the generator methods below when the generator is already executing raises a `ValueError` exception.

generator.__next__()

Starts the execution of a generator function or resumes it at the last executed yield expression. When a generator function is resumed with a `__next__()` method, the current yield expression always evaluates to `None`. The execution then continues to the next yield expression, where the generator is suspended again, and the value of the `yield_list` is returned to `__next__()`'s caller. If the generator exits without yielding another value, a `StopIteration` exception is raised.

This method is normally called implicitly, e.g. by a `for` loop, or by the built-in `next()` function.

generator.send(*value*)

Resumes the execution and “sends” a value into the generator function. The *value* argument becomes the result of the current yield expression. The `send()` method returns the next value yielded by the generator, or raises `StopIteration` if the generator exits without yielding another value. When `send()` is called to start the generator, it must be called with `None` as the argument, because there is no yield expression that could receive the value.

generator.throw(*value*)

generator.throw(*type*[, *value*[, *traceback*]])

Raises an exception at the point where the generator was paused, and returns the next value yielded by the generator function. If the generator exits without yielding another value, a `StopIteration` exception is raised. If the generator function does not catch the passed-in exception, or raises a different exception, then that exception propagates to the caller.

In typical use, this is called with a single exception instance similar to the way the `raise` keyword is used.

For backwards compatibility, however, the second signature is supported, following a convention from older versions of Python. The *type* argument should be an exception class, and *value* should be an exception instance. If the *value* is not provided, the *type* constructor is called to get an instance. If *traceback* is provided, it is set on the exception, otherwise any existing `__traceback__` attribute stored in *value* may be cleared.

Changed in version 3.12: The second signature (type[, value[, traceback]]) is deprecated and may be removed in a future version of Python.

generator.close()

Raises a `GeneratorExit` exception at the point where the generator function was paused (equivalent to calling `throw(GeneratorExit)`). The exception is raised by the yield expression where the generator was paused. If the generator function catches the exception and returns a value, this value is returned from `close()`. If the generator function is already closed, or raises `GeneratorExit` (by not catching the exception), `close()` returns `None`. If the generator yields a value, a `RuntimeError` is raised. If the generator raises any other exception, it is propagated to the caller. If the generator has already exited due to an exception or normal exit, `close()` returns `None` and has no other effect.

Changed in version 3.13: If a generator returns a value upon being closed, the value is returned by `close()`.

6.2.9.2. Examples

Here is a simple example that demonstrates the behavior of generators and generator functions:

```
>>> def echo(value=None):
...     print("Execution starts when 'next()' is called for the
first time.")
...     try:
...         while True:
...             try:
...                 value = (yield value)
...             except Exception as e:
...                 value = e
...     finally:
...         print("Don't forget to clean up when 'close()' is
called.")
...
>>> generator = echo(1)
>>> print(next(generator))
Execution starts when 'next()' is called for the first time.
1
>>> print(next(generator))
None
>>> print(generator.send(2))
2
```

```
>>> generator.throw(TypeError, "spam")
TypeError('spam',)
>>> generator.close()
Don't forget to clean up when 'close()' is called.
```

For examples using `yield from`, see PEP 380: Syntax for Delegating to a Subgenerator in “What’s New in Python.”

6.2.9.3. Asynchronous generator functions

The presence of a yield expression in a function or method defined using **`async def`** further defines the function as an asynchronous generator function.

When an asynchronous generator function is called, it returns an asynchronous iterator known as an asynchronous generator object. That object then controls the execution of the generator function. An asynchronous generator object is typically used in an **`async for`** statement in a coroutine function analogously to how a generator object would be used in a **`for`** statement.

Calling one of the asynchronous generator’s methods returns an awaitable object, and the execution starts when this object is awaited on. At that time, the execution proceeds to the first yield expression, where it is suspended again, returning the value of **`yield_list`** to the awaiting coroutine. As with a generator, suspension means that all local state is retained, including the current bindings of local variables, the instruction pointer, the internal evaluation stack, and the state of any exception handling. When the execution is resumed by awaiting on the next object returned by the asynchronous generator’s methods, the function can proceed exactly as if the yield expression were just another external call. The value of the yield expression after resuming depends on the method which resumed the execution. If `__anext__()` is used then the result is `None`. Otherwise, if `asend()` is used, then the result will be the value passed in to that method.

If an asynchronous generator happens to exit early by `break`, the caller task being cancelled, or other exceptions, the generator's async cleanup code will run and possibly raise exceptions or access context variables in an unexpected context–perhaps after the lifetime of tasks it depends, or during the event loop shutdown when the async-generator garbage collection hook is called. To prevent this, the caller must explicitly close the async generator by calling `aclose()` method to finalize the generator and ultimately detach it from the event loop.

In an asynchronous generator function, yield expressions are allowed anywhere in a `try` construct. However, if an asynchronous generator is not resumed before it is finalized (by reaching a zero reference count or by being garbage collected), then a yield expression within a **`try`** construct could result in a failure to execute pending `finally` clauses. In this case, it is the responsibility of the event loop or scheduler running the asynchronous generator to call the asynchronous generator-iterator's `aclose()` method and run the resulting coroutine object, thus allowing any pending **`finally`** clauses to execute.

To take care of finalization upon event loop termination, an event loop should define a *finalizer* function which takes an asynchronous generator-iterator and presumably calls `aclose()` and executes the coroutine. This *finalizer* may be registered by calling `sys.set_asyncgen_hooks()`. When first iterated over, an asynchronous generator-iterator will store the registered *finalizer* to be called upon finalization. For a reference example of a *finalizer* method see the implementation of `asyncio.Loop.shutdown_asyncgens` in

Lib/asyncio/base_events.py
[https://github.com/python/cpython/tree/3.14/Lib/asyncio/base_events.py].

The expression `yield from <expr>` is a syntax error when used in an asynchronous generator function.

6.2.9.4. Asynchronous generator-iterator methods

This subsection describes the methods of an asynchronous generator iterator, which are used to control the execution of a generator function.

async agen.__anext__()

Returns an awaitable which when run starts to execute the asynchronous generator or resumes it at the last executed yield expression. When an asynchronous generator function is resumed with an **`__anext__()`** method, the current yield expression always evaluates to **`None`** in the returned awaitable, which when run will continue to the next yield expression. The value of the **`yield_list`** of the yield expression is the value of the **`StopIteration`** exception raised by the completing coroutine. If the asynchronous generator exits without yielding another value, the awaitable instead raises a **`StopAsyncIteration`** exception, signalling that the asynchronous iteration has completed.

This method is normally called implicitly by a **`async for`** loop.

async agen.asend(*value*)

Returns an awaitable which when run resumes the execution of the asynchronous generator. As with the **`send()`** method for a generator, this “sends” a value into the asynchronous generator function, and the *value* argument becomes the result of the current yield expression. The awaitable returned by the `asend()` method will return the next value yielded by the generator as the value of the raised `StopIteration`, or raises `StopAsyncIteration` if the asynchronous generator exits without yielding another value. When `asend()` is called to start the asynchronous generator, it must be called with `None` as the argument, because there is no yield expression that could receive the value.

async agen.athrow(*value*)

async agen.athrow(*type*[, *value*[, *traceback*]])

Returns an awaitable that raises an exception of type `type` at the point where the asynchronous generator was paused, and returns the next value yielded by the generator function as the value of the raised `StopIteration` exception. If the asynchronous generator exits without yielding another value, a `StopAsyncIteration` exception is raised by the awaitable. If the generator function does not catch the passed-in exception, or raises a different exception, then when the awaitable is run that exception propagates to the caller of the awaitable.

Changed in version 3.12: The second signature (type[, value[, traceback]]) is deprecated and may be removed in a future version of Python.

async agen.aclose()

Returns an awaitable that when run will throw a `GeneratorExit` into the asynchronous generator function at the point where it was paused. If the asynchronous generator function then exits gracefully, is already closed, or raises `GeneratorExit` (by not catching the exception), then the returned awaitable will raise a `StopIteration` exception. Any further awaitables returned by subsequent calls to the asynchronous generator will raise a `StopAsyncIteration` exception. If the asynchronous generator yields a value, a `RuntimeError` is raised by the awaitable. If the asynchronous generator raises any other exception, it is propagated to the caller of the awaitable. If the asynchronous generator has already exited due to an exception or normal exit, then further calls to `aclose()` will return an awaitable that does nothing.

6.3. Primaries

Primaries represent the most tightly bound operations of the language. Their syntax is:

```
primary: atom | attributeref | subscription | slicing | call
```

6.3.1. Attribute references

An attribute reference is a primary followed by a period and a name:

```
attributeref: primary "." identifier
```

The primary must evaluate to an object of a type that supports attribute references, which most objects do. This object is then asked to produce the attribute whose name is the identifier. The type and value produced is determined by the object. Multiple evaluations of the same attribute reference may yield different objects.

This production can be customized by overriding the `__getattribute__()` method or the `__getattr__()` method. The `__getattribute__()` method is called first and either returns a value or raises `AttributeError` if the attribute is not available.

If an `AttributeError` is raised and the object has a **`__getattr__()`** method, that method is called as a fallback.

6.3.2. Subscriptions

The subscription of an instance of a container class will generally select an element from the container. The subscription of a generic class will generally return a GenericAlias object.

```
subscription: primary "[" flexible_expression_list "]"
```

When an object is subscripted, the interpreter will evaluate the primary and the expression list.

The primary must evaluate to an object that supports subscription. An object may support subscription through defining one or both of `__getitem__()` and `__class_getitem__()`. When the primary is subscripted, the evaluated result of the expression list will be passed to one of these methods. For more details on when `__class_getitem__` is called instead of `__getitem__`, see __class_getitem__ versus __getitem__.

If the expression list contains at least one comma, or if any of the expressions are starred, the expression list will evaluate to a `tuple` containing the items of the expression list. Otherwise, the expression list will evaluate to the value of the list’s sole member.

Changed in version 3.11: Expressions in an expression list may be starred. See **PEP 646** [https://peps.python.org/pep-0646/].

For built-in objects, there are two types of objects that support subscription via `__getitem__()`:

1. Mappings. If the primary is a mapping, the expression list must evaluate to an object whose value is one of the keys of the mapping, and the subscription selects the value in the mapping that corresponds to that key. An example of a builtin mapping class is the `dict` class.
2. Sequences. If the primary is a sequence, the expression list must evaluate to an `int` or a `slice` (as discussed in the following section). Examples of builtin sequence classes include the `str`, `list` and `tuple` classes.

The formal syntax makes no special provision for negative indices in sequences. However, built-in sequences all provide a `__getitem__()` method that interprets negative indices by adding the length of the sequence to the index so that, for example, `x[-1]` selects the last item of `x`. The resulting value must be a nonnegative integer less than the number of items in the sequence, and the subscription selects the item whose index is that value (counting from zero). Since the support for negative indices and slicing occurs in the object's `__getitem__()` method, subclasses overriding this method will need to explicitly add that support.

A `string` is a special kind of sequence whose items are *characters*. A character is not a separate data type but a string of exactly one character.

6.3.3. Slicings

A slicing selects a range of items in a sequence object (e.g., a string, tuple or list). Slicings may be used as expressions or as targets in assignment or `del` statements. The syntax for a slicing:

```
slicing:      primary "[" slice_list "]"
slice_list:   slice_item ("," slice_item)* [","]
```

```
slice_item:   expression | proper_slice
proper_slice: [lower_bound] ":" [upper_bound] [ ":" [stride] ]
lower_bound:  expression
upper_bound:  expression
stride:       expression
```

There is ambiguity in the formal syntax here: anything that looks like an expression list also looks like a slice list, so any subscription can be interpreted as a slicing. Rather than further complicating the syntax, this is disambiguated by defining that in this case the interpretation as a subscription takes priority over the interpretation as a slicing (this is the case if the slice list contains no proper slice).

The semantics for a slicing are as follows. The primary is indexed (using the same `__getitem__()` method as normal subscription) with a key that is constructed from the slice list, as follows. If the slice list contains at least one comma, the key is a tuple containing the conversion of the slice items; otherwise, the conversion of the lone slice item is the key. The conversion of a slice item that is an expression is that expression. The conversion of a proper slice is a slice object (see section The standard type hierarchy) whose `start`, `stop` and `step` attributes are the values of the expressions given as lower bound, upper bound and stride, respectively, substituting `None` for missing expressions.

6.3.4. Calls

A call calls a callable object (e.g., a function) with a possibly empty series of arguments:

```
call:                 primary "(" [argument_list [","] |
comprehension] ")"
argument_list:        positional_arguments [","
starred_and_keywords]
                        ["," keywords_arguments]
```

```
                          | starred_and_keywords [","
keywords_arguments]
                          | keywords_arguments
positional_arguments: positional_item ("," positional_item)*
positional_item:      assignment_expression | "*" expression
starred_and_keywords: ("*" expression | keyword_item)
                      ("," "*" expression | "," keyword_item)*
keywords_arguments:   (keyword_item | "**" expression)
                      ("," keyword_item | "," "**" expression)*
keyword_item:         identifier "=" expression
```

An optional trailing comma may be present after the positional and keyword arguments but does not affect the semantics.

The primary must evaluate to a callable object (user-defined functions, built-in functions, methods of built-in objects, class objects, methods of class instances, and all objects having a `__call__()` method are callable). All argument expressions are evaluated before the call is attempted. Please refer to section Function definitions for the syntax of formal parameter lists.

If keyword arguments are present, they are first converted to positional arguments, as follows. First, a list of unfilled slots is created for the formal parameters. If there are N positional arguments, they are placed in the first N slots. Next, for each keyword argument, the identifier is used to determine the corresponding slot (if the identifier is the same as the first formal parameter name, the first slot is used, and so on). If the slot is already filled, a `TypeError` exception is raised. Otherwise, the argument is placed in the slot, filling it (even if the expression is `None`, it fills the slot). When all arguments have been processed, the slots that are still unfilled are filled with the corresponding default value from the function definition. (Default values are calculated, once, when the function is defined; thus, a mutable object such as a list or dictionary used as default value will be shared by all calls that don’t specify an argument value for the corresponding slot; this should usually be avoided.) If there are any unfilled slots for which no default value is specified, a **`TypeError`** exception is raised. Otherwise, the list of filled slots is used as the argument list for the call.

CPython implementation detail: An implementation may provide built-in functions whose positional parameters do not have names, even if they are ‘named’ for the purpose of documentation, and which therefore cannot be supplied by keyword. In CPython, this is the case for functions implemented in C that use **`PyArg_ParseTuple()`** to parse their arguments.

If there are more positional arguments than there are formal parameter slots, a **`TypeError`** exception is raised, unless a formal parameter using the syntax `*identifier` is present; in this case, that formal parameter receives a tuple containing the excess positional arguments (or an empty tuple if there were no excess positional arguments).

If any keyword argument does not correspond to a formal parameter name, a **`TypeError`** exception is raised, unless a formal parameter using the syntax `**identifier` is present; in this case, that formal parameter receives a dictionary containing the excess keyword arguments (using the keywords as keys and the argument values as corresponding values), or a (new) empty dictionary if there were no excess keyword arguments.

If the syntax `*expression` appears in the function call, `expression` must evaluate to an iterable. Elements from these iterables are treated as if they were additional positional arguments. For the call `f(x1, x2, *y, x3, x4)`, if *y* evaluates to a sequence *y1*, …, *yM*, this is equivalent to a call with M+4 positional arguments *x1*, *x2*, *y1*, …, *yM*, *x3*, *x4*.

A consequence of this is that although the `*expression` syntax may appear *after* explicit keyword arguments, it is processed *before* the keyword arguments (and any `**expression` arguments – see below). So:

```
>>> def f(a, b):
...     print(a, b)
...
>>> f(b=1, *(2,))
2 1
>>> f(a=1, *(2,))
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: f() got multiple values for keyword argument 'a'
>>> f(1, *(2,))
1 2
```

It is unusual for both keyword arguments and the `*expression` syntax to be used in the same call, so in practice this confusion does not often arise.

If the syntax `**expression` appears in the function call, `expression` must evaluate to a mapping, the contents of which are treated as additional keyword arguments. If a parameter matching a key has already been given a value (by an explicit keyword argument, or from another unpacking), a **`TypeError`** exception is raised.

When `**expression` is used, each key in this mapping must be a string. Each value from the mapping is assigned to the first formal parameter eligible for keyword assignment whose name is equal to the key. A key need not be a Python identifier (e.g. `"max-temp °F"` is acceptable, although it will not match any formal parameter that could be declared). If there is no match to a formal parameter the key-value pair is collected by the `**` parameter, if there is one, or if there is not, a **`TypeError`** exception is raised.

Formal parameters using the syntax `*identifier` or `**identifier` cannot be used as positional argument slots or as keyword argument names.

Changed in version 3.5: Function calls accept any number of `*` and `**` unpackings, positional arguments may follow iterable unpackings (`*`), and keyword arguments may follow dictionary unpackings (`**`). Originally proposed by **PEP 448** [https://peps.python.org/pep-0448/].

A call always returns some value, possibly `None`, unless it raises an exception. How this value is computed depends on the type of the callable object.

If it is—

a user-defined function:

The code block for the function is executed, passing it the argument list. The first thing the code block will do is bind the formal parameters to the arguments; this is described in section Function definitions. When the code block executes a **`return`** statement, this specifies the return value of the function call. If execution reaches the end of the code block without executing a **`return`** statement, the return value is `None`.

a built-in function or method:

The result is up to the interpreter; see Built-in Functions for the descriptions of built-in functions and methods.

a class object:

A new instance of that class is returned.

a class instance method:

The corresponding user-defined function is called, with an argument list that is one longer than the argument list of the call: the instance becomes the first argument.

a class instance:
: The class must define a `__call__()` method; the effect is then the same as if that method was called.

6.4. Await expression

Suspend the execution of coroutine on an awaitable object. Can only be used inside a coroutine function.

```
await_expr: "await" primary
```

Added in version 3.5.

6.5. The power operator

The power operator binds more tightly than unary operators on its left; it binds less tightly than unary operators on its right. The syntax is:

```
power: (await_expr | primary) ["**" u_expr]
```

Thus, in an unparenthesized sequence of power and unary operators, the operators are evaluated from right to left (this does not constrain the evaluation order for the operands): `-1**2` results in `-1`.

The power operator has the same semantics as the built-in `pow()` function, when called with two arguments: it yields its left argument raised to the power of its right argument. The numeric arguments are first converted to a common type, and the result is of that type.

For int operands, the result has the same type as the operands unless the second argument is negative; in that case, all arguments are converted to float and a float result is delivered. For example, `10**2` returns `100`, but `10**-2` returns `0.01`.

Raising `0.0` to a negative power results in a `ZeroDivisionError`. Raising a negative number to a fractional power results in a `complex` number. (In earlier versions it raised a `ValueError`.)

This operation can be customized using the special `__pow__()` and `__rpow__()` methods.

6.6. Unary arithmetic and bitwise operations

All unary arithmetic and bitwise operations have the same priority:

```
u_expr: power | "-" u_expr | "+" u_expr | "~" u_expr
```

The unary `-` (minus) operator yields the negation of its numeric argument; the operation can be overridden with the `__neg__()` special method.

The unary `+` (plus) operator yields its numeric argument unchanged; the operation can be overridden with the `__pos__()` special method.

The unary `~` (invert) operator yields the bitwise inversion of its integer argument. The bitwise inversion of `x` is defined as `-(x+1)`. It only applies to integral numbers or to custom objects that override the `__invert__()` special method.

In all three cases, if the argument does not have the proper type, a `TypeError` exception is raised.

6.7. Binary arithmetic operations

The binary arithmetic operations have the conventional priority levels. Note that some of these operations also apply to certain non-numeric types. Apart from the power operator, there are only two levels, one for multiplicative operators and one for additive operators:

```
m_expr: u_expr | m_expr "*" u_expr | m_expr "@" m_expr |
        m_expr "//" u_expr | m_expr "/" u_expr |
        m_expr "%" u_expr
a_expr: m_expr | a_expr "+" m_expr | a_expr "-" m_expr
```

The `*` (multiplication) operator yields the product of its arguments. The arguments must either both be numbers, or one argument must be an integer and the other must be a sequence. In the former case, the numbers are converted to a common real type and then multiplied together. In the latter case, sequence repetition is performed; a negative repetition factor yields an empty sequence.

This operation can be customized using the special `__mul__()` and `__rmul__()` methods.

Changed in version 3.14: If only one operand is a complex number, the other operand is converted to a floating-point number.

The `@` (at) operator is intended to be used for matrix multiplication. No builtin Python types implement this operator.

This operation can be customized using the special `__matmul__()` and `__rmatmul__()` methods.

Added in version 3.5.

The `/` (division) and `//` (floor division) operators yield the quotient of their arguments. The numeric arguments are first converted to a common type. Division of integers yields a float, while floor division of integers results in an integer; the result is that of mathematical division with the ‘floor’ function applied to the result. Division by zero raises the `ZeroDivisionError` exception.

The division operation can be customized using the special `__truediv__()` and `__rtruediv__()` methods. The floor division operation can be customized using the special `__floordiv__()` and `__rfloordiv__()` methods.

The `%` (modulo) operator yields the remainder from the division of the first argument by the second. The numeric arguments are first converted to a common type. A zero right argument raises the `ZeroDivisionError` exception. The arguments may be floating-point numbers, e.g., `3.14%0.7` equals `0.34` (since `3.14` equals `4*0.7 + 0.34`.) The modulo operator always yields a result with the same sign as its second operand (or zero); the absolute value of the result is strictly smaller than the absolute value of the second operand [[1]].

The floor division and modulo operators are connected by the following identity: `x == (x//y)*y + (x%y)`. Floor division and modulo are also connected with the built-in function `divmod()`: `divmod(x, y) == (x//y, x%y)`. [[2]].

In addition to performing the modulo operation on numbers, the `%` operator is also overloaded by string objects to perform old-style string formatting (also known as interpolation). The syntax for string formatting is described in the Python Library Reference, section printf-style String Formatting.

The *modulo* operation can be customized using the special `__mod__()` and `__rmod__()` methods.

The floor division operator, the modulo operator, and the `divmod()` function are not defined for complex numbers. Instead, convert to a floating-point number using the `abs()` function if appropriate.

The `+` (addition) operator yields the sum of its arguments. The arguments must either both be numbers or both be sequences of the same type. In the former case, the numbers are converted to a common real type and then added together. In the latter case, the sequences are concatenated.

This operation can be customized using the special `__add__()` and `__radd__()` methods.

Changed in version 3.14: If only one operand is a complex number, the other operand is converted to a floating-point number.

The `-` (subtraction) operator yields the difference of its arguments. The numeric arguments are first converted to a common real type.

This operation can be customized using the special `__sub__()` and `__rsub__()` methods.

Changed in version 3.14: If only one operand is a complex number, the other operand is converted to a floating-point number.

6.8. Shifting operations

The shifting operations have lower priority than the arithmetic operations:

```
shift_expr: a_expr | shift_expr ("<<" | ">>") a_expr
```

These operators accept integers as arguments. They shift the first argument to the left or right by the number of bits given by the second argument.

The left shift operation can be customized using the special `__lshift__()` and `__rlshift__()` methods. The right shift operation can be customized using the special `__rshift__()` and `__rrshift__()` methods.

A right shift by *n* bits is defined as floor division by `pow(2,n)`. A left shift by *n* bits is defined as multiplication with `pow(2,n)`.

6.9. Binary bitwise operations

Each of the three bitwise operations has a different priority level:

```
and_expr: shift_expr | and_expr "&" shift_expr
xor_expr: and_expr | xor_expr "^" and_expr
or_expr:  xor_expr | or_expr "|" xor_expr
```

The `&` operator yields the bitwise AND of its arguments, which must be integers or one of them must be a custom object overriding `__and__()` or `__rand__()` special methods.

The `^` operator yields the bitwise XOR (exclusive OR) of its arguments, which must be integers or one of them must be a custom object overriding `__xor__()` or `__rxor__()` special methods.

The `|` operator yields the bitwise (inclusive) OR of its arguments, which must be integers or one of them must be a custom object overriding `__or__()` or `__ror__()` special methods.

6.10. Comparisons

Unlike C, all comparison operations in Python have the same priority, which is lower than that of any arithmetic, shifting or bitwise operation. Also unlike C, expressions like `a < b < c` have the interpretation that is conventional in mathematics:

```
comparison:    or_expr (comp_operator or_expr)*
comp_operator: "<" | ">" | "==" | ">=" | "<=" | "!="
               | "is" ["not"] | ["not"] "in"
```

Comparisons yield boolean values: `True` or `False`. Custom *rich comparison methods* may return non-boolean values. In this case Python will call `bool()` on such value in boolean contexts.

Comparisons can be chained arbitrarily, e.g., `x < y <= z` is equivalent to `x < y and y <= z`, except that `y` is evaluated only once (but in both cases `z` is not evaluated at all when `x < y` is found to be false).

Formally, if *a*, *b*, *c*, …, *y*, *z* are expressions and *op1*, *op2*, …, *opN* are comparison operators, then `a op1 b op2 c ... y opN z` is equivalent to `a op1 b and b op2 c and ... y opN z`, except that each expression is evaluated at most once.

Note that `a op1 b op2 c` doesn't imply any kind of comparison between *a* and *c*, so that, e.g., `x < y > z` is perfectly legal (though perhaps not pretty).

6.10.1. Value comparisons

The operators `<`, `>`, `==`, `>=`, `<=`, and `!=` compare the values of two objects. The objects do not need to have the same type.

Chapter Objects, values and types states that objects have a value (in addition to type and identity). The value of an object is a rather abstract notion in Python: For example, there is no canonical access method for an object's value. Also, there is no requirement that the value of an object should be constructed in a particular way, e.g. comprised of all its data attributes. Comparison operators implement a particular notion of what the value of an object is. One can think of them as defining the value of an object indirectly, by means of their comparison implementation.

Because all types are (direct or indirect) subtypes of **`object`**, they inherit the default comparison behavior from **`object`**. Types can customize their comparison behavior by implementing *rich comparison methods* like **`__lt__()`**, described in Basic customization.

The default behavior for equality comparison (`==` and `!=`) is based on the identity of the objects. Hence, equality comparison of instances with the same identity results in equality, and equality comparison of instances with different identities results in inequality. A motivation for this default behavior is the desire that all objects should be reflexive (i.e. `x is y` implies `x == y`).

A default order comparison (`<`, `>`, `<=`, and `>=`) is not provided; an attempt raises **`TypeError`**. A motivation for this default behavior is the lack of a similar invariant as for equality.

The behavior of the default equality comparison, that instances with different identities are always unequal, may be in contrast to what types will need that have a sensible definition of object value and value-based equality. Such types will need to customize their comparison behavior, and in fact, a number of built-in types have done that.

The following list describes the comparison behavior of the most important built-in types.

- Numbers of built-in numeric types (Numeric Types — int, float, complex) and of the standard library types **`fractions.Fraction`** and **`decimal.Decimal`** can be compared within and across their types, with the restriction that complex numbers do not support order comparison. Within the limits of the types involved, they compare mathematically (algorithmically) correct without loss of precision.

  The not-a-number values `float('NaN')` and `decimal.Decimal('NaN')` are special. Any ordered comparison of a number to a not-a-number value is false. A counter-intuitive implication is that not-a-number values are not equal to themselves. For example, if `x = float('NaN')`, `3 < x`, `x < 3` and `x == x` are all false, while `x != x` is true. This behavior is compliant with IEEE 754.

- `None` and **`NotImplemented`** are singletons. **PEP 8** [https://peps.python.org/pep-0008/] advises that comparisons for singletons should always be done with `is` or `is not`, never the equality operators.

- Binary sequences (instances of **`bytes`** or **`bytearray`**) can be compared within and across their types. They compare lexicographically using the numeric values of their elements.

- Strings (instances of `str`) compare lexicographically using the numerical Unicode code points (the result of the built-in function `ord()`) of their characters. [[3]]

  Strings and binary sequences cannot be directly compared.

- Sequences (instances of `tuple`, `list`, or `range`) can be compared only within each of their types, with the restriction that ranges do not support order comparison. Equality comparison across these types results in inequality, and ordering comparison across these types raises `TypeError`.

  Sequences compare lexicographically using comparison of corresponding elements. The built-in containers typically assume identical objects are equal to themselves. That lets them bypass equality tests for identical objects to improve performance and to maintain their internal invariants.

  Lexicographical comparison between built-in collections works as follows:

  - For two collections to compare equal, they must be of the same type, have the same length, and each pair of corresponding elements must compare equal (for example, `[1,2] == (1,2)` is false because the type is not the same).
  - Collections that support order comparison are ordered the same as their first unequal elements (for example, `[1,2,x] <= [1,2,y]` has the same value as `x <= y`). If a corresponding element does not exist,

the shorter collection is ordered first (for example, `[1,2] < [1,2,3]` is true).

- Mappings (instances of **`dict`**) compare equal if and only if they have equal `(key, value)` pairs. Equality comparison of the keys and values enforces reflexivity.

  Order comparisons (`<`, `>`, `<=`, and `>=`) raise **`TypeError`**.

- Sets (instances of **`set`** or **`frozenset`**) can be compared within and across their types.

  They define order comparison operators to mean subset and superset tests. Those relations do not define total orderings (for example, the two sets `{1,2}` and `{2,3}` are not equal, nor subsets of one another, nor supersets of one another). Accordingly, sets are not appropriate arguments for functions which depend on total ordering (for example, **`min()`**, **`max()`**, and **`sorted()`** produce undefined results given a list of sets as inputs).

  Comparison of sets enforces reflexivity of its elements.

- Most other built-in types have no comparison methods implemented, so they inherit the default comparison behavior.

User-defined classes that customize their comparison behavior should follow some consistency rules, if possible:

- Equality comparison should be reflexive. In other words, identical objects should compare equal:

  `x is y` implies `x == y`

- Comparison should be symmetric. In other words, the following expressions should have the same result:

`x == y` and `y == x`

`x != y` and `y != x`

`x < y` and `y > x`

`x <= y` and `y >= x`

- Comparison should be transitive. The following (non-exhaustive) examples illustrate that:

  `x > y and y > z` implies `x > z`

  `x < y and y <= z` implies `x < z`

- Inverse comparison should result in the boolean negation. In other words, the following expressions should have the same result:

  `x == y` and `not x != y`

  `x < y` and `not x >= y` (for total ordering)

  `x > y` and `not x <= y` (for total ordering)

  The last two expressions apply to totally ordered collections (e.g. to sequences, but not to sets or mappings). See also the **`total_ordering()`** decorator.

- The **`hash()`** result should be consistent with equality. Objects that are equal should either have the same hash value, or be marked as unhashable.

Python does not enforce these consistency rules. In fact, the not-a-number values are an example for not following these rules.

6.10.2. Membership test operations

The operators **`in`** and **`not in`** test for membership. `x in s` evaluates to `True` if *x* is a member of *s*, and `False` otherwise. `x not in s` returns the negation of `x in s`. All built-in sequences and set types support this as well as dictionary, for which **`in`** tests whether the dictionary has a given key. For container types such as list, tuple, set, frozenset, dict, or collections.deque, the expression `x in y` is equivalent to `any(x is e or x == e for e in y)`.

For the string and bytes types, `x in y` is `True` if and only if *x* is a substring of *y*. An equivalent test is `y.find(x) != -1`. Empty strings are always considered to be a substring of any other string, so `"" in "abc"` will return `True`.

For user-defined classes which define the **`__contains__()`** method, `x in y` returns `True` if `y.__contains__(x)` returns a true value, and `False` otherwise.

For user-defined classes which do not define **`__contains__()`** but do define **`__iter__()`**, `x in y` is `True` if some value `z`, for which the expression `x is z or x == z` is true, is produced while iterating over `y`. If an exception is raised during the iteration, it is as if **`in`** raised that exception.

Lastly, the old-style iteration protocol is tried: if a class defines **`__getitem__()`**, `x in y` is `True` if and only if there is a non-negative integer index *i* such that `x is y[i] or x == y[i]`, and no lower integer index raises the **`IndexError`** exception. (If any other exception is raised, it is as if **`in`** raised that exception).

The operator **`not in`** is defined to have the inverse truth value of **`in`**.

6.10.3. Identity comparisons

The operators **`is`** and **`is not`** test for an object's identity: `x is y` is true if and only if *x* and *y* are the same object. An Object's identity is determined using the **`id()`** function. `x is not y` yields the inverse truth value. [[4]]

6.11. Boolean operations

```
or_test:  and_test | or_test "or" and_test
and_test: not_test | and_test "and" not_test
not_test: comparison | "not" not_test
```

In the context of Boolean operations, and also when expressions are used by control flow statements, the following values are interpreted as false: `False`, `None`, numeric zero of all types, and empty strings and containers (including strings, tuples, lists, dictionaries, sets and frozensets). All other values are interpreted as true. User-defined objects can customize their truth value by providing a **`__bool__()`** method.

The operator **`not`** yields `True` if its argument is false, `False` otherwise.

The expression `x and y` first evaluates *x*; if *x* is false, its value is returned; otherwise, *y* is evaluated and the resulting value is returned.

The expression `x or y` first evaluates *x*; if *x* is true, its value is returned; otherwise, *y* is evaluated and the resulting value is returned.

Note that neither **`and`** nor **`or`** restrict the value and type they return to `False` and `True`, but rather return the last evaluated argument. This is sometimes useful, e.g., if `s` is a string that should be replaced by a default value if it is empty, the expression `s or 'foo'` yields the desired value. Because **`not`** has to create a new value, it returns a boolean value regardless of the type of its argument (for example, `not 'foo'` produces `False` rather than `''`.)

6.12. Assignment expressions

assignment_expression: [identifier ":="] expression

An assignment expression (sometimes also called a “named expression” or “walrus”) assigns an **expression** to an **identifier**, while also returning the value of the **expression**.

One common use case is when handling matched regular expressions:

```
if matching := pattern.search(data):
    do_something(matching)
```

Or, when processing a file stream in chunks:

```
while chunk := file.read(9000):
    process(chunk)
```

Assignment expressions must be surrounded by parentheses when used as expression statements and when used as sub-expressions in slicing, conditional, lambda, keyword-argument, and comprehension-if expressions and in `assert`, `with`, and `assignment` statements. In all other places where they can be used, parentheses are not required, including in `if` and `while` statements.

Added in version 3.8: See **PEP 572** [https://peps.python.org/pep-0572/] for more details about assignment expressions.

6.13. Conditional expressions

```
conditional_expression: or_test ["if" or_test "else"
expression]
expression:             conditional_expression | lambda_expr
```

A conditional expression (sometimes called a “ternary operator”) is an alternative to the if-else statement. As it is an expression, it returns a value and can appear as a sub-expression.

The expression `x if C else y` first evaluates the condition, *C* rather than *x*. If *C* is true, *x* is evaluated and its value is returned; otherwise, *y* is evaluated and its value is returned.

See **PEP 308** [https://peps.python.org/pep-0308/] for more details about conditional expressions.

6.14. Lambdas

```
lambda_expr: "lambda" [parameter_list] ":" expression
```

Lambda expressions (sometimes called lambda forms) are used to create anonymous functions. The expression `lambda parameters: expression` yields a function object. The unnamed object behaves like a function object defined with:

```
def <lambda>(parameters):
    return expression
```

See section Function definitions for the syntax of parameter lists. Note that functions created with lambda expressions cannot contain statements or annotations.

6.15. Expression lists

```
starred_expression:       "*" or_expr | expression
flexible_expression:      assignment_expression |
starred_expression
flexible_expression_list: flexible_expression (","
flexible_expression)* [","]
starred_expression_list:  starred_expression (","
starred_expression)* [","]
expression_list:          expression ("," expression)* [","]
yield_list:               expression_list | starred_expression
"," [starred_expression_list]
```

Except when part of a list or set display, an expression list containing at least one comma yields a tuple. The length of the tuple is the number of expressions in the list. The expressions are evaluated from left to right.

An asterisk `*` denotes *iterable unpacking*. Its operand must be an iterable. The iterable is expanded into a sequence of items, which are included in the new tuple, list, or set, at the site of the unpacking.

Added in version 3.5: Iterable unpacking in expression lists, originally proposed by **PEP 448** [https://peps.python.org/pep-0448/].

Added in version 3.11: Any item in an expression list may be starred. See **PEP 646** [https://peps.python.org/pep-0646/].

A trailing comma is required only to create a one-item tuple, such as `1,`; it is optional in all other cases. A single expression without a trailing comma doesn't create a tuple, but rather yields the value of that expression. (To create an empty tuple, use an empty pair of parentheses: `()`.)

6.16. Evaluation order

Python evaluates expressions from left to right. Notice that while evaluating an assignment, the right-hand side is evaluated before the left-hand side.

In the following lines, expressions will be evaluated in the arithmetic order of their suffixes:

```
expr1, expr2, expr3, expr4
(expr1, expr2, expr3, expr4)
{expr1: expr2, expr3: expr4}
expr1 + expr2 * (expr3 - expr4)
expr1(expr2, expr3, *expr4, **expr5)
expr3, expr4 = expr1, expr2
```

6.17. Operator precedence

The following table summarizes the operator precedence in Python, from highest precedence (most binding) to lowest precedence (least binding). Operators in the same box have the same precedence. Unless the syntax is explicitly given, operators are binary. Operators in the same box group left to right (except for exponentiation and conditional expressions, which group from right to left).

Note that comparisons, membership tests, and identity tests, all have the same precedence and have a left-to-right chaining feature as described in the Comparisons section.

Operator	Description
`(expressions...)`, `[expressions...]`, `{key: value...}`, `{expressions...}`	Binding or parenthesized expression, list display, dictionary display, set display

Operator	Description
`x[index]`, `x[index:index]`, `x(arguments...)`, `x.attribute`	Subscription, slicing, call, attribute reference
`await x`	Await expression
`**`	Exponentiation [[5]]
`+x`, `-x`, `~x`	Positive, negative, bitwise NOT
`*`, `@`, `/`, `//`, `%`	Multiplication, matrix multiplication, division, floor division, remainder [[6]]
`+`, `-`	Addition and subtraction
`<<`, `>>`	Shifts
`&`	Bitwise AND
`^`	Bitwise XOR
`|`	Bitwise OR

Operator	Description
`in`, `not in`, `is`, `is not`, `<`, `<=`, `>`, `>=`, `!=`, `==`	Comparisons, including membership tests and identity tests
`not x`	Boolean NOT
`and`	Boolean AND
`or`	Boolean OR
`if` – `else`	Conditional expression
`lambda`	Lambda expression
`:=`	Assignment expression

Footnotes

[1]

While `abs(x%y) < abs(y)` is true mathematically, for floats it may not be true numerically due to roundoff. For example, and assuming a platform on which a Python float is an IEEE 754 double-precision number, in order that `-1e-100 % 1e100` have the same sign as `1e100`, the computed result is `-1e-100 + 1e100`, which is numerically exactly equal to `1e100`. The

function `math.fmod()` returns a result whose sign matches the sign of the first argument instead, and so returns `-1e-100` in this case. Which approach is more appropriate depends on the application.
[2]

If x is very close to an exact integer multiple of y, it’s possible for `x//y` to be one larger than `(x-x%y)//y` due to rounding. In such cases, Python returns the latter result, in order to preserve that `divmod(x,y)[0] * y + x % y` be very close to `x`.

[3]

The Unicode standard distinguishes between *code points* (e.g. U+0041) and *abstract characters* (e.g. “LATIN CAPITAL LETTER A”). While most abstract characters in Unicode are only represented using one code point, there is a number of abstract characters that can in addition be represented using a sequence of more than one code point. For example, the abstract character “LATIN CAPITAL LETTER C WITH CEDILLA” can be represented as a single *precomposed character* at code position U+00C7, or as a sequence of a *base character* at code position U+0043 (LATIN CAPITAL LETTER C), followed by a *combining character* at code position U+0327 (COMBINING CEDILLA).

The comparison operators on strings compare at the level of Unicode code points. This may be counter-intuitive to humans. For example, `"\u00C7" == "\u0043\u0327"` is `False`, even though both strings represent the same abstract character “LATIN CAPITAL LETTER C WITH CEDILLA”.

To compare strings at the level of abstract characters (that is, in a way intuitive to humans), use `unicodedata.normalize()`.

[4]

Due to automatic garbage-collection, free lists, and the dynamic nature of descriptors, you may notice seemingly unusual behaviour in certain uses of the `is` operator, like those involving comparisons between instance methods, or constants. Check their documentation for more info.
[5]

The power operator `**` binds less tightly than an arithmetic or bitwise unary operator on its right, that is, `2**-1` is `0.5`.

[6]

The `%` operator is also used for string formatting; the same precedence applies.

7. Simple statements

A simple statement is comprised within a single logical line. Several simple statements may occur on a single line separated by semicolons. The syntax for simple statements is:

```
simple_stmt: expression_stmt
             | assert_stmt
             | assignment_stmt
             | augmented_assignment_stmt
             | annotated_assignment_stmt
             | pass_stmt
             | del_stmt
             | return_stmt
             | yield_stmt
             | raise_stmt
             | break_stmt
             | continue_stmt
             | import_stmt
             | future_stmt
             | global_stmt
             | nonlocal_stmt
             | type_stmt
```

7.1. Expression statements

Expression statements are used (mostly interactively) to compute and write a value, or (usually) to call a procedure (a function that returns no meaningful result; in Python, procedures return the value `None`). Other uses of expression statements are allowed and occasionally useful. The syntax for an expression statement is:

```
expression_stmt: starred_expression
```

An expression statement evaluates the expression list (which may be a single expression).

In interactive mode, if the value is not `None`, it is converted to a string using the built-in `repr()` function and the resulting string is written to standard output on a line by itself (except if the result is `None`, so that procedure calls do not cause any output.)

7.2. Assignment statements

Assignment statements are used to (re)bind names to values and to modify attributes or items of mutable objects:

```
assignment_stmt: (target_list "=")+ (starred_expression |
yield_expression)
target_list:     target ("," target)* [","]
target:          identifier
                 | "(" [target_list] ")"
                 | "[" [target_list] "]"
                 | attributeref
                 | subscription
                 | slicing
                 | "*" target
```

(See section Primaries for the syntax definitions for *attributeref*, *subscription*, and *slicing*.)

An assignment statement evaluates the expression list (remember that this can be a single expression or a comma-separated list, the latter yielding a tuple) and assigns the single resulting object to each of the target lists, from left to right.

Assignment is defined recursively depending on the form of the target (list). When a target is part of a mutable object (an attribute reference, subscription or slicing), the mutable object must ultimately perform the assignment and decide about its validity, and may raise an exception if the assignment is unacceptable. The rules observed by various types and the exceptions raised are given with the definition of the object types (see section The standard type hierarchy).

Assignment of an object to a target list, optionally enclosed in parentheses or square brackets, is recursively defined as follows.

- If the target list is a single target with no trailing comma, optionally in parentheses, the object is assigned to that target.
- Else:
  - If the target list contains one target prefixed with an asterisk, called a “starred” target: The object must be an iterable with at least as many items as there are targets in the target list, minus one. The first items of the iterable are assigned, from left to right, to the targets before the starred target. The final items of the iterable are assigned to the targets after the starred target. A list of the remaining items in the iterable is then assigned to the starred target (the list can be empty).
  - Else: The object must be an iterable with the same number of items as there are targets in the target list, and the items are assigned, from left to right, to the corresponding targets.

Assignment of an object to a single target is recursively defined as follows.

- If the target is an identifier (name):

  - If the name does not occur in a **`global`** or **`nonlocal`** statement in the current code block: the name is bound to the object in the current local namespace.
  - Otherwise: the name is bound to the object in the global namespace or the outer namespace determined by **`nonlocal`**, respectively.

  The name is rebound if it was already bound. This may cause the reference count for the object previously bound to the name to reach zero, causing the object to be deallocated and its destructor (if it has one) to be called.

- If the target is an attribute reference: The primary expression in the reference is evaluated. It should yield an object with assignable attributes; if this is not the case, **`TypeError`** is raised. That object is then asked to assign the assigned object to the given attribute; if it cannot perform the assignment, it raises an exception (usually but not necessarily **`AttributeError`**).

  Note: If the object is a class instance and the attribute reference occurs on both sides of the assignment operator, the right-hand side expression, `a.x` can access either an instance attribute or (if no instance attribute exists) a class attribute. The left-hand side target `a.x` is always set as an instance attribute, creating it if necessary. Thus, the two occurrences of `a.x` do not necessarily refer to the same attribute: if the right-hand side expression refers to a class attribute, the left-hand side creates a new instance attribute as the target of the assignment:

  ```
  class Cls:
      x = 3             # class variable
  inst = Cls()
  ```

```
inst.x = inst.x + 1   # writes inst.x as 4 leaving Cls.x as
3
```

This description does not necessarily apply to descriptor attributes, such as properties created with `property()`.

- If the target is a subscription: The primary expression in the reference is evaluated. It should yield either a mutable sequence object (such as a list) or a mapping object (such as a dictionary). Next, the subscript expression is evaluated.

  If the primary is a mutable sequence object (such as a list), the subscript must yield an integer. If it is negative, the sequence's length is added to it. The resulting value must be a nonnegative integer less than the sequence's length, and the sequence is asked to assign the assigned object to its item with that index. If the index is out of range, `IndexError` is raised (assignment to a subscripted sequence cannot add new items to a list).

  If the primary is a mapping object (such as a dictionary), the subscript must have a type compatible with the mapping's key type, and the mapping is then asked to create a key/value pair which maps the subscript to the assigned object. This can either replace an existing key/value pair with the same key value, or insert a new key/value pair (if no key with the same value existed).

  For user-defined objects, the `__setitem__()` method is called with appropriate arguments.

- If the target is a slicing: The primary expression in the reference is evaluated. It should yield a mutable sequence object (such as a list). The assigned object should be a sequence object of the same type. Next, the lower and upper bound expressions are evaluated, insofar they are present; defaults are zero and the sequence’s length. The bounds should evaluate to integers. If either bound is negative, the sequence’s length is added to it. The resulting bounds are clipped to lie between zero and the sequence’s length, inclusive. Finally, the sequence object is asked to replace the slice with the items of the assigned sequence. The length of the slice may be different from the length of the assigned sequence, thus changing the length of the target sequence, if the target sequence allows it.

CPython implementation detail: In the current implementation, the syntax for targets is taken to be the same as for expressions, and invalid syntax is rejected during the code generation phase, causing less detailed error messages.

Although the definition of assignment implies that overlaps between the left-hand side and the right-hand side are ‘simultaneous’ (for example `a, b = b, a` swaps two variables), overlaps *within* the collection of assigned-to variables occur left-to-right, sometimes resulting in confusion. For instance, the following program prints `[0, 2]`:

```
x = [0, 1]
i = 0
i, x[i] = 1, 2         # i is updated, then x[i] is updated
print(x)
```

See also

PEP 3132 [https://peps.python.org/pep-3132/] - Extended Iterable Unpacking

The specification for the `*target` feature.

7.2.1. Augmented assignment statements

Augmented assignment is the combination, in a single statement, of a binary operation and an assignment statement:

```
augmented_assignment_stmt: augtarget augop (expression_list | yield_expression)
augtarget:                 identifier | attributeref | subscription | slicing
augop:                     "+=" | "-=" | "*=" | "@=" | "/=" | "//=" | "%=" | "**="
                           | ">>=" | "<<=" | "&=" | "^=" | "|="
```

(See section Primaries for the syntax definitions of the last three symbols.)

An augmented assignment evaluates the target (which, unlike normal assignment statements, cannot be an unpacking) and the expression list, performs the binary operation specific to the type of assignment on the two operands, and assigns the result to the original target. The target is only evaluated once.

An augmented assignment statement like `x += 1` can be rewritten as `x = x + 1` to achieve a similar, but not exactly equal effect. In the augmented version, `x` is only evaluated once. Also, when possible, the actual operation is performed *in-place*, meaning that rather than creating a new object and assigning that to the target, the old object is modified instead.

Unlike normal assignments, augmented assignments evaluate the left-hand side *before* evaluating the right-hand side. For example, `a[i] += f(x)` first looks-up `a[i]`, then it evaluates `f(x)` and performs the addition, and lastly, it writes the result back to `a[i]`.

With the exception of assigning to tuples and multiple targets in a single statement, the assignment done by augmented assignment statements is handled the same way as normal assignments. Similarly, with the exception of the possible *in-place* behavior, the binary operation performed by augmented assignment is the same as the normal binary operations.

For targets which are attribute references, the same caveat about class and instance attributes applies as for regular assignments.

7.2.2. Annotated assignment statements

Annotation assignment is the combination, in a single statement, of a variable or attribute annotation and an optional assignment statement:

```
annotated_assignment_stmt: augtarget ":" expression
                           ["=" (starred_expression |
yield_expression)]
```

The difference from normal Assignment statements is that only a single target is allowed.

The assignment target is considered “simple” if it consists of a single name that is not enclosed in parentheses. For simple assignment targets, if in class or module scope, the annotations are gathered in a lazily evaluated annotation scope. The annotations can be evaluated using the `__annotations__` attribute of a class or module, or using the facilities in the `annotationlib` module.

If the assignment target is not simple (an attribute, subscript node, or parenthesized name), the annotation is never evaluated.

If a name is annotated in a function scope, then this name is local for that scope. Annotations are never evaluated and stored in function scopes.

If the right hand side is present, an annotated assignment performs the actual assignment as if there was no annotation present. If the right hand side is not present for an expression target, then the interpreter evaluates the target except for the last `__setitem__()` or `__setattr__()` call.

See also

PEP 526 [https://peps.python.org/pep-0526/] - Syntax for Variable Annotations

The proposal that added syntax for annotating the types of variables (including class variables and instance variables), instead of expressing them through comments.

PEP 484 [https://peps.python.org/pep-0484/] - Type hints

The proposal that added the `typing` module to provide a standard syntax for type annotations that can be used in static analysis tools and IDEs.

Changed in version 3.8: Now annotated assignments allow the same expressions in the right hand side as regular assignments. Previously, some expressions (like un-parenthesized tuple expressions) caused a syntax error.

Changed in version 3.14: Annotations are now lazily evaluated in a separate annotation scope. If the assignment target is not simple, annotations are never evaluated.

7.3. The `assert` statement

Assert statements are a convenient way to insert debugging assertions into a program:

```
assert_stmt: "assert" expression ["," expression]
```

The simple form, `assert expression`, is equivalent to

```
if __debug__:
    if not expression: raise AssertionError
```

The extended form, `assert expression1, expression2`, is equivalent to

```
if __debug__:
    if not expression1: raise AssertionError(expression2)
```

These equivalences assume that `__debug__` and `AssertionError` refer to the built-in variables with those names. In the current implementation, the built-in variable `__debug__` is `True` under normal circumstances, `False` when optimization is requested (command line option `-O`). The current code generator emits no code for an `assert` statement when optimization is requested at compile time. Note that it is unnecessary to include the source code for the expression that failed in the error message; it will be displayed as part of the stack trace.

Assignments to `__debug__` are illegal. The value for the built-in variable is determined when the interpreter starts.

7.4. The `pass` statement

```
pass_stmt: "pass"
```

`pass` is a null operation — when it is executed, nothing happens. It is useful as a placeholder when a statement is required syntactically, but no code needs to be executed, for example:

```
def f(arg): pass    # a function that does nothing (yet)

class C: pass       # a class with no methods (yet)
```

7.5. The `del` statement

```
del_stmt: "del" target_list
```

Deletion is recursively defined very similar to the way assignment is defined. Rather than spelling it out in full details, here are some hints.

Deletion of a target list recursively deletes each target, from left to right.

Deletion of a name removes the binding of that name from the local or global namespace, depending on whether the name occurs in a `global` statement in the same code block. Trying to delete an unbound name raises a `NameError` exception.

Deletion of attribute references, subscriptions and slicings is passed to the primary object involved; deletion of a slicing is in general equivalent to assignment of an empty slice of the right type (but even this is determined by the sliced object).

Changed in version 3.2: Previously it was illegal to delete a name from the local namespace if it occurs as a free variable in a nested block.

7.6. The `return` statement

```
return_stmt: "return" [expression_list]
```

`return` may only occur syntactically nested in a function definition, not within a nested class definition.

If an expression list is present, it is evaluated, else `None` is substituted.

`return` leaves the current function call with the expression list (or `None`) as return value.

When `return` passes control out of a `try` statement with a `finally` clause, that **`finally`** clause is executed before really leaving the function.

In a generator function, the `return` statement indicates that the generator is done and will cause `StopIteration` to be raised. The returned value (if any) is used as an argument to construct `StopIteration` and becomes the `StopIteration.value` attribute.

In an asynchronous generator function, an empty `return` statement indicates that the asynchronous generator is done and will cause `StopAsyncIteration` to be raised. A non-empty **`return`** statement is a syntax error in an asynchronous generator function.

7.7. The `yield` statement

```
yield_stmt: yield_expression
```

A `yield` statement is semantically equivalent to a yield expression. The `yield` statement can be used to omit the parentheses that would otherwise be required in the equivalent yield expression statement. For example, the yield statements

```
yield <expr>
yield from <expr>
```

are equivalent to the yield expression statements

```
(yield <expr>)
(yield from <expr>)
```

Yield expressions and statements are only used when defining a generator function, and are only used in the body of the generator function. Using **`yield`** in a function definition is sufficient to cause that definition to create a generator function instead of a normal function.

For full details of **`yield`** semantics, refer to the Yield expressions section.

7.8. The `raise` statement

```
raise_stmt: "raise" [expression ["from" expression]]
```

If no expressions are present, **`raise`** re-raises the exception that is currently being handled, which is also known as the *active exception*. If there isn’t currently an active exception, a **`RuntimeError`** exception is raised indicating that this is an error.

Otherwise, **`raise`** evaluates the first expression as the exception object. It must be either a subclass or an instance of **`BaseException`**. If it is a class, the exception instance will be obtained when needed by instantiating the class with no arguments.

The *type* of the exception is the exception instance’s class, the *value* is the instance itself.

A traceback object is normally created automatically when an exception is raised and attached to it as the `__traceback__` attribute. You can create an exception and set your own traceback in one step using the `with_traceback()` exception method (which returns the same exception instance, with its traceback set to its argument), like so:

```
raise Exception("foo occurred").with_traceback(tracebackobj)
```

The `from` clause is used for exception chaining: if given, the second *expression* must be another exception class or instance. If the second expression is an exception instance, it will be attached to the raised exception as the `__cause__` attribute (which is writable). If the expression is an exception class, the class will be instantiated and the resulting exception instance will be attached to the raised exception as the **`__cause__`** attribute. If the raised exception is not handled, both exceptions will be printed:

```
>>> try:
...     print(1 / 0)
... except Exception as exc:
...     raise RuntimeError("Something bad happened") from exc
...
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
    print(1 / 0)
          ~~^~~
ZeroDivisionError: division by zero

The above exception was the direct cause of the following
exception:

Traceback (most recent call last):
  File "<stdin>", line 4, in <module>
```

```
    raise RuntimeError("Something bad happened") from exc
RuntimeError: Something bad happened
```

A similar mechanism works implicitly if a new exception is raised when an exception is already being handled. An exception may be handled when an **`except`** or **`finally`** clause, or a **`with`** statement, is used. The previous exception is then attached as the new exception’s **`__context__`** attribute:

```
>>> try:
...     print(1 / 0)
... except:
...     raise RuntimeError("Something bad happened")
...
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
    print(1 / 0)
          ~~^~~
ZeroDivisionError: division by zero

During handling of the above exception, another exception
occurred:

Traceback (most recent call last):
  File "<stdin>", line 4, in <module>
    raise RuntimeError("Something bad happened")
RuntimeError: Something bad happened
```

Exception chaining can be explicitly suppressed by specifying **`None`** in the `from` clause:

```
>>> try:
...     print(1 / 0)
... except:
...     raise RuntimeError("Something bad happened") from None
...
Traceback (most recent call last):
  File "<stdin>", line 4, in <module>
RuntimeError: Something bad happened
```

Additional information on exceptions can be found in section Exceptions, and information about handling exceptions is in section The try statement.

Changed in version 3.3: **`None`** is now permitted as `Y` in `raise X from Y`.

Added the **`__suppress_context__`** attribute to suppress automatic display of the exception context.

Changed in version 3.11: If the traceback of the active exception is modified in an **`except`** clause, a subsequent `raise` statement re-raises the exception with the modified traceback. Previously, the exception was re-raised with the traceback it had when it was caught.

7.9. The `break` statement

```
break_stmt: "break"
```

`break` may only occur syntactically nested in a **`for`** or **`while`** loop, but not nested in a function or class definition within that loop.

It terminates the nearest enclosing loop, skipping the optional **`else`** clause if the loop has one.

If a **`for`** loop is terminated by **`break`**, the loop control target keeps its current value.

When **`break`** passes control out of a **`try`** statement with a **`finally`** clause, that **`finally`** clause is executed before really leaving the loop.

7.10. The `continue` statement

```
continue_stmt: "continue"
```

`continue` may only occur syntactically nested in a `for` or `while` loop, but not nested in a function or class definition within that loop. It continues with the next cycle of the nearest enclosing loop.

When `continue` passes control out of a `try` statement with a `finally` clause, that `finally` clause is executed before really starting the next loop cycle.

7.11. The `import` statement

```
import_stmt:     "import" module ["as" identifier] ("," module
["as" identifier])*
                 | "from" relative_module "import" identifier
["as" identifier]
                 ("," identifier ["as" identifier])*
                 | "from" relative_module "import" "("
identifier ["as" identifier]
                 ("," identifier ["as" identifier])* [","] ")"
                 | "from" relative_module "import" "*"
module:          (identifier ".")* identifier
relative_module: "."* module | "."+
```

The basic import statement (no `from` clause) is executed in two steps:

1. find a module, loading and initializing it if necessary
2. define a name or names in the local namespace for the scope where the `import` statement occurs.

When the statement contains multiple clauses (separated by commas) the two steps are carried out separately for each clause, just as though the clauses had been separated out into individual import statements.

The details of the first step, finding and loading modules, are described in greater detail in the section on the import system, which also describes the various types of packages and modules that can be imported, as well as all the hooks that can be used to customize the import system. Note that failures in this step may indicate either that the module could not be located, *or* that an error occurred while initializing the module, which includes execution of the module's code.

If the requested module is retrieved successfully, it will be made available in the local namespace in one of three ways:

- If the module name is followed by **`as`**, then the name following **`as`** is bound directly to the imported module.
- If no other name is specified, and the module being imported is a top level module, the module's name is bound in the local namespace as a reference to the imported module
- If the module being imported is *not* a top level module, then the name of the top level package that contains the module is bound in the local namespace as a reference to the top level package. The imported module must be accessed using its full qualified name rather than directly

The **`from`** form uses a slightly more complex process:

1. find the module specified in the **`from`** clause, loading and initializing it if necessary;
2. for each of the identifiers specified in the **`import`** clauses:
    1. check if the imported module has an attribute by that name
    2. if not, attempt to import a submodule with that name and then check the imported module again for that attribute

3. if the attribute is not found, `ImportError` is raised.
4. otherwise, a reference to that value is stored in the local namespace, using the name in the **as** clause if it is present, otherwise using the attribute name

Examples:

```
import foo                 # foo imported and bound locally
import foo.bar.baz         # foo, foo.bar, and foo.bar.baz
imported, foo bound locally
import foo.bar.baz as fbb  # foo, foo.bar, and foo.bar.baz
imported, foo.bar.baz bound as fbb
from foo.bar import baz    # foo, foo.bar, and foo.bar.baz
imported, foo.bar.baz bound as baz
from foo import attr       # foo imported and foo.attr bound as
attr
```

If the list of identifiers is replaced by a star (`'*'`), all public names defined in the module are bound in the local namespace for the scope where the `import` statement occurs.

The *public names* defined by a module are determined by checking the module's namespace for a variable named `__all__`; if defined, it must be a sequence of strings which are names defined or imported by that module. The names given in `__all__` are all considered public and are required to exist. If `__all__` is not defined, the set of public names includes all names found in the module's namespace which do not begin with an underscore character (`'_'`). `__all__` should contain the entire public API. It is intended to avoid accidentally exporting items that are not part of the API (such as library modules which were imported and used within the module).

The wild card form of import — `from module import *` — is only allowed at the module level. Attempting to use it in class or function definitions will raise a `SyntaxError`.

When specifying what module to import you do not have to specify the absolute name of the module. When a module or package is contained within another package it is possible to make a relative import within the same top package without having to mention the package name. By using leading dots in the specified module or package after **`from`** you can specify how high to traverse up the current package hierarchy without specifying exact names. One leading dot means the current package where the module making the import exists. Two dots means up one package level. Three dots is up two levels, etc. So if you execute `from . import mod` from a module in the `pkg` package then you will end up importing `pkg.mod`. If you execute `from ..subpkg2 import mod` from within `pkg.subpkg1` you will import `pkg.subpkg2.mod`. The specification for relative imports is contained in the Package Relative Imports section.

`importlib.import_module()` is provided to support applications that determine dynamically the modules to be loaded.

Raises an auditing event `import` with arguments `module`, `filename`, `sys.path`, `sys.meta_path`, `sys.path_hooks`.

7.11.1. Future statements

A *future statement* is a directive to the compiler that a particular module should be compiled using syntax or semantics that will be available in a specified future release of Python where the feature becomes standard.

The future statement is intended to ease migration to future versions of Python that introduce incompatible changes to the language. It allows use of the new features on a per-module basis before the release in which the feature becomes standard.

```
future_stmt: "from" "__future__" "import" feature ["as"
identifier]
             ("," feature ["as" identifier])*
             | "from" "__future__" "import" "(" feature ["as"
identifier]
             ("," feature ["as" identifier])* [","] ")"
feature:     identifier
```

A future statement must appear near the top of the module. The only lines that can appear before a future statement are:

- the module docstring (if any),
- comments,
- blank lines, and
- other future statements.

The only feature that requires using the future statement is `annotations` (see **PEP 563** [https://peps.python.org/pep-0563/]).

All historical features enabled by the future statement are still recognized by Python 3. The list includes `absolute_import`, `division`, `generators`, `generator_stop`, `unicode_literals`, `print_function`, `nested_scopes` and `with_statement`. They are all redundant because they are always enabled, and only kept for backwards compatibility.

A future statement is recognized and treated specially at compile time: Changes to the semantics of core constructs are often implemented by generating different code. It may even be the case that a new feature introduces new incompatible syntax (such as a new reserved word), in which case the compiler may need to parse the module differently. Such decisions cannot be pushed off until runtime.

For any given release, the compiler knows which feature names have been defined, and raises a compile-time error if a future statement contains a feature not known to it.

The direct runtime semantics are the same as for any import statement: there is a standard module `__future__`, described later, and it will be imported in the usual way at the time the future statement is executed.

The interesting runtime semantics depend on the specific feature enabled by the future statement.

Note that there is nothing special about the statement:

```
import __future__ [as name]
```

That is not a future statement; it’s an ordinary import statement with no special semantics or syntax restrictions.

Code compiled by calls to the built-in functions `exec()` and `compile()` that occur in a module `M` containing a future statement will, by default, use the new syntax or semantics associated with the future statement. This can be controlled by optional arguments to `compile()` — see the documentation of that function for details.

A future statement typed at an interactive interpreter prompt will take effect for the rest of the interpreter session. If an interpreter is started with the `-i` option, is passed a script name to execute, and the script includes a future statement, it will be in effect in the interactive session started after the script is executed.

See also

PEP 236 [https://peps.python.org/pep-0236/] - Back to the __future__

The original proposal for the __future__ mechanism.

7.12. The `global` statement

```
global_stmt: "global" identifier ("," identifier)*
```

The `global` statement causes the listed identifiers to be interpreted as globals. It would be impossible to assign to a global variable without `global`, although free variables may refer to globals without being declared global.

The `global` statement applies to the entire current scope (module, function body or class definition). A `SyntaxError` is raised if a variable is used or assigned to prior to its global declaration in the scope.

At the module level, all variables are global, so a `global` statement has no effect. However, variables must still not be used or assigned to prior to their `global` declaration. This requirement is relaxed in the interactive prompt (REPL).

Programmer's note: `global` is a directive to the parser. It applies only to code parsed at the same time as the `global` statement. In particular, a `global` statement contained in a string or code object supplied to the built-in `exec()` function does not affect the code block *containing* the function call, and code contained in such a string is unaffected by `global` statements in the code containing the function call. The same applies to the `eval()` and `compile()` functions.

7.13. The `nonlocal` statement

```
nonlocal_stmt: "nonlocal" identifier ("," identifier)*
```

When the definition of a function or class is nested (enclosed) within the definitions of other functions, its nonlocal scopes are the local scopes of the enclosing functions. The `nonlocal` statement causes the listed identifiers to refer to names previously bound in nonlocal scopes. It allows encapsulated code to rebind such nonlocal identifiers. If a name is bound in more than one nonlocal scope, the nearest binding is used. If a name is not bound in any nonlocal scope, or if there is no nonlocal scope, a `SyntaxError` is raised.

The `nonlocal` statement applies to the entire scope of a function or class body. A `SyntaxError` is raised if a variable is used or assigned to prior to its nonlocal declaration in the scope.

See also

PEP 3104 [https://peps.python.org/pep-3104/] - Access to Names in Outer Scopes
The specification for the `nonlocal` statement.

Programmer’s note: `nonlocal` is a directive to the parser and applies only to code parsed along with it. See the note for the `global` statement.

7.14. The `type` statement

```
type_stmt: 'type' identifier [type_params] "=" expression
```

The **`type`** statement declares a type alias, which is an instance of `typing.TypeAliasType`.

For example, the following statement creates a type alias:

```
type Point = tuple[float, float]
```

This code is roughly equivalent to:

```
annotation-def VALUE_OF_Point():
    return tuple[float, float]
Point = typing.TypeAliasType("Point", VALUE_OF_Point())
```

`annotation-def` indicates an annotation scope, which behaves mostly like a function, but with several small differences.

The value of the type alias is evaluated in the annotation scope. It is not evaluated when the type alias is created, but only when the value is accessed through the type alias's **`__value__`** attribute (see Lazy evaluation). This allows the type alias to refer to names that are not yet defined.

Type aliases may be made generic by adding a type parameter list after the name. See Generic type aliases for more.

`type` is a soft keyword.

Added in version 3.12.

See also

PEP 695 [https://peps.python.org/pep-0695/] - Type Parameter Syntax

Introduced the **`type`** statement and syntax for generic classes and functions.

8. Compound statements

Compound statements contain (groups of) other statements; they affect or control the execution of those other statements in some way. In general, compound statements span multiple lines, although in simple incarnations a whole compound statement may be contained in one line.

The `if`, `while` and `for` statements implement traditional control flow constructs. `try` specifies exception handlers and/or cleanup code for a group of statements, while the `with` statement allows the execution of initialization and finalization code around a block of code. Function and class definitions are also syntactically compound statements.

A compound statement consists of one or more 'clauses.' A clause consists of a header and a 'suite.' The clause headers of a particular compound statement are all at the same indentation level. Each clause header begins with a uniquely identifying keyword and ends with a colon. A suite is a group of statements controlled by a clause. A suite can be one or more semicolon-separated simple statements on the same line as the header, following the header's colon, or it can be one or more indented statements on subsequent lines. Only the latter form of a suite can contain nested compound statements; the following is illegal, mostly because it wouldn't be clear to which `if` clause a following `else` clause would belong:

```
if test1: if test2: print(x)
```

Also note that the semicolon binds tighter than the colon in this context, so that in the following example, either all or none of the `print()` calls are executed:

```
if x < y < z: print(x); print(y); print(z)
```

Summarizing:

```
compound_stmt: if_stmt
               | while_stmt
               | for_stmt
               | try_stmt
               | with_stmt
               | match_stmt
               | funcdef
               | classdef
               | async_with_stmt
               | async_for_stmt
               | async_funcdef
suite:         stmt_list NEWLINE | NEWLINE INDENT statement+
DEDENT
statement:     stmt_list NEWLINE | compound_stmt
stmt_list:     simple_stmt (";" simple_stmt)* [";"]
```

Note that statements always end in a `NEWLINE` possibly followed by a `DEDENT`. Also note that optional continuation clauses always begin with a keyword that cannot start a statement, thus there are no ambiguities (the ‘dangling `else`’ problem is solved in Python by requiring nested `if` statements to be indented).

The formatting of the grammar rules in the following sections places each clause on a separate line for clarity.

8.1. The `if` statement

The `if` statement is used for conditional execution:

```
if_stmt: "if" assignment_expression ":" suite
```

```
         ("elif" assignment_expression ":" suite)*
         ["else" ":" suite]
```

It selects exactly one of the suites by evaluating the expressions one by one until one is found to be true (see section Boolean operations for the definition of true and false); then that suite is executed (and no other part of the `if` statement is executed or evaluated). If all expressions are false, the suite of the `else` clause, if present, is executed.

8.2. The `while` statement

The `while` statement is used for repeated execution as long as an expression is true:

```
while_stmt: "while" assignment_expression ":" suite
            ["else" ":" suite]
```

This repeatedly tests the expression and, if it is true, executes the first suite; if the expression is false (which may be the first time it is tested) the suite of the `else` clause, if present, is executed and the loop terminates.

A `break` statement executed in the first suite terminates the loop without executing the `else` clause’s suite. A `continue` statement executed in the first suite skips the rest of the suite and goes back to testing the expression.

8.3. The `for` statement

The `for` statement is used to iterate over the elements of a sequence (such as a string, tuple or list) or other iterable object:

```
for_stmt: "for" target_list "in" starred_expression_list ":"
```

```
suite
          ["else" ":" suite]
```

The `starred_expression_list` expression is evaluated once; it should yield an iterable object. An iterator is created for that iterable. The first item provided by the iterator is then assigned to the target list using the standard rules for assignments (see Assignment statements), and the suite is executed. This repeats for each item provided by the iterator. When the iterator is exhausted, the suite in the **`else`** clause, if present, is executed, and the loop terminates.

A `break` statement executed in the first suite terminates the loop without executing the **`else`** clause’s suite. A `continue` statement executed in the first suite skips the rest of the suite and continues with the next item, or with the **`else`** clause if there is no next item.

The for-loop makes assignments to the variables in the target list. This overwrites all previous assignments to those variables including those made in the suite of the for-loop:

```
for i in range(10):
    print(i)
    i = 5             # this will not affect the for-loop
                      # because i will be overwritten with the
next
                      # index in the range
```

Names in the target list are not deleted when the loop is finished, but if the sequence is empty, they will not have been assigned to at all by the loop. Hint: the built-in type `range()` represents immutable arithmetic sequences of integers. For instance, iterating `range(3)` successively yields 0, 1, and then 2.

Changed in version 3.11: Starred elements are now allowed in the expression list.

8.4. The `try` statement

The `try` statement specifies exception handlers and/or cleanup code for a group of statements:

```
try_stmt:  try1_stmt | try2_stmt | try3_stmt
try1_stmt: "try" ":" suite
           ("except" [expression ["as" identifier]] ":" suite)+
           ["else" ":" suite]
           ["finally" ":" suite]
try2_stmt: "try" ":" suite
           ("except" "*" expression ["as" identifier] ":"
suite)+
           ["else" ":" suite]
           ["finally" ":" suite]
try3_stmt: "try" ":" suite
           "finally" ":" suite
```

Additional information on exceptions can be found in section Exceptions, and information on using the `raise` statement to generate exceptions may be found in section The raise statement.

Changed in version 3.14: Support for optionally dropping grouping parentheses when using multiple exception types. See **PEP 758** [https://peps.python.org/pep-0758/].

8.4.1. `except` clause

The `except` clause(s) specify one or more exception handlers. When no exception occurs in the `try` clause, no exception handler is executed. When an exception occurs in the `try` suite, a search for an exception handler is started. This search inspects the `except` clauses in turn until one is found that matches the exception. An expression-less `except` clause, if present, must be last; it matches any exception.

For an **except** clause with an expression, the expression must evaluate to an exception type or a tuple of exception types. Parentheses can be dropped if multiple exception types are provided and the `as` clause is not used. The raised exception matches an **except** clause whose expression evaluates to the class or a non-virtual base class of the exception object, or to a tuple that contains such a class.

If no **except** clause matches the exception, the search for an exception handler continues in the surrounding code and on the invocation stack. [[1]]

If the evaluation of an expression in the header of an **except** clause raises an exception, the original search for a handler is canceled and a search starts for the new exception in the surrounding code and on the call stack (it is treated as if the entire **try** statement raised the exception).

When a matching **except** clause is found, the exception is assigned to the target specified after the **as** keyword in that **except** clause, if present, and the **except** clause's suite is executed. All **except** clauses must have an executable block. When the end of this block is reached, execution continues normally after the entire **try** statement. (This means that if two nested handlers exist for the same exception, and the exception occurs in the **try** clause of the inner handler, the outer handler will not handle the exception.)

When an exception has been assigned using `as target`, it is cleared at the end of the **except** clause. This is as if

```
except E as N:
    foo
```

was translated to

```
except E as N:
    try:
        foo
    finally:
        del N
```

This means the exception must be assigned to a different name to be able to refer to it after the **`except`** clause. Exceptions are cleared because with the traceback attached to them, they form a reference cycle with the stack frame, keeping all locals in that frame alive until the next garbage collection occurs.

Before an **`except`** clause's suite is executed, the exception is stored in the `sys` module, where it can be accessed from within the body of the **`except`** clause by calling `sys.exception()`. When leaving an exception handler, the exception stored in the `sys` module is reset to its previous value:

```
>>> print(sys.exception())
None
>>> try:
...     raise TypeError
... except:
...     print(repr(sys.exception()))
...     try:
...          raise ValueError
...     except:
...         print(repr(sys.exception()))
...     print(repr(sys.exception()))
...
TypeError()
ValueError()
TypeError()
>>> print(sys.exception())
None
```

8.4.2. `except*` clause

The **`except*`** clause(s) specify one or more handlers for groups of exceptions ([`BaseExceptionGroup`](#) instances). A [`try`](#) statement can have either [`except`](#) or **`except*`** clauses, but not both. The exception type for matching is mandatory in the case of **`except*`**, so `except*:` is a syntax error. The type is interpreted as in the case of **`except`**, but matching is performed on the exceptions contained in the group that is being handled. An [`TypeError`](#) is raised if a matching type is a subclass of **`BaseExceptionGroup`**, because that would have ambiguous semantics.

When an exception group is raised in the try block, each **`except*`** clause splits (see [`split()`](#)) it into the subgroups of matching and non-matching exceptions. If the matching subgroup is not empty, it becomes the handled exception (the value returned from [`sys.exception()`](#)) and assigned to the target of the **`except*`** clause (if there is one). Then, the body of the **`except*`** clause executes. If the non-matching subgroup is not empty, it is processed by the next **`except*`** in the same manner. This continues until all exceptions in the group have been matched, or the last **`except*`** clause has run.

After all **`except*`** clauses execute, the group of unhandled exceptions is merged with any exceptions that were raised or re-raised from within **`except*`** clauses. This merged exception group propagates on.:

```
>>> try:
...     raise ExceptionGroup("eg",
...         [ValueError(1), TypeError(2), OSError(3),
OSError(4)])
... except* TypeError as e:
...     print(f'caught {type(e)} with nested {e.exceptions}')
... except* OSError as e:
...     print(f'caught {type(e)} with nested {e.exceptions}')
...
caught <class 'ExceptionGroup'> with nested (TypeError(2),)
caught <class 'ExceptionGroup'> with nested (OSError(3),
OSError(4))
  + Exception Group Traceback (most recent call last):
  |   File "<doctest default[0]>", line 2, in <module>
  |     raise ExceptionGroup("eg",
  |         [ValueError(1), TypeError(2), OSError(3),
OSError(4)])
  | ExceptionGroup: eg (1 sub-exception)
  +-+---------------- 1 ----------------
    | ValueError: 1
    +------------------------------------
```

If the exception raised from the `try` block is not an exception group and its type matches one of the **`except*`** clauses, it is caught and wrapped by an exception group with an empty message string. This ensures that the type of the target `e` is consistently `BaseExceptionGroup`:

```
>>> try:
...     raise BlockingIOError
... except* BlockingIOError as e:
...     print(repr(e))
...
ExceptionGroup('', (BlockingIOError(),))
```

`break`, `continue` and `return` cannot appear in an **`except*`** clause.

8.4.3. `else` clause

The optional **`else`** clause is executed if the control flow leaves the `try` suite, no exception was raised, and no `return`, `continue`, or `break` statement was executed. Exceptions in the **`else`** clause are not handled by the preceding `except` clauses.

8.4.4. `finally` clause

If **`finally`** is present, it specifies a ‘cleanup’ handler. The `try` clause is executed, including any `except` and `else` clauses. If an exception occurs in any of the clauses and is not handled, the exception is temporarily saved. The **`finally`** clause is executed. If there is a saved exception it is re-raised at the end of the **`finally`** clause. If the **`finally`** clause raises another exception, the saved exception is set as the context of the new exception. If the **`finally`** clause executes a `return`, `break` or `continue` statement, the saved exception is discarded. For example, this function returns 42.

```
def f():
    try:
        1/0
    finally:
        return 42
```

The exception information is not available to the program during execution of the **`finally`** clause.

When a `return`, `break` or `continue` statement is executed in the `try` suite of a **`try`**…**`finally`** statement, the **`finally`** clause is also executed 'on the way out.'

The return value of a function is determined by the last `return` statement executed. Since the **`finally`** clause always executes, a **`return`** statement executed in the **`finally`** clause will always be the last one executed. The following function returns 'finally'.

```
def foo():
    try:
        return 'try'
    finally:
        return 'finally'
```

Changed in version 3.8: Prior to Python 3.8, a `continue` statement was illegal in the **`finally`** clause due to a problem with the implementation.

Changed in version 3.14: The compiler emits a `SyntaxWarning` when a `return`, `break` or `continue` appears in a **`finally`** block (see **PEP 765** [https://peps.python.org/pep-0765/]).

8.5. The `with` statement

The `with` statement is used to wrap the execution of a block with methods defined by a context manager (see section With Statement Context Managers). This allows common `try`…`except`…`finally` usage patterns to be encapsulated for convenient reuse.

```
with_stmt:          "with" ( "(" with_stmt_contents ","? ")" |
with_stmt_contents ) ":" suite
with_stmt_contents: with_item ("," with_item)*
with_item:          expression ["as" target]
```

The execution of the `with` statement with one “item” proceeds as follows:

1. The context expression (the expression given in the `with_item`) is evaluated to obtain a context manager.
2. The context manager’s `__enter__()` is loaded for later use.
3. The context manager’s `__exit__()` is loaded for later use.
4. The context manager’s `__enter__()` method is invoked.
5. If a target was included in the `with` statement, the return value from `__enter__()` is assigned to it.

   Note

   The `with` statement guarantees that if the `__enter__()` method returns without an error, then `__exit__()` will always be called. Thus, if an error occurs during the assignment to the target list, it will be treated the same as an error occurring within the suite would be. See step 7 below.

6. The suite is executed.
7. The context manager's `__exit__()` method is invoked. If an exception caused the suite to be exited, its type, value, and traceback are passed as arguments to `__exit__()`. Otherwise, three `None` arguments are supplied.

   If the suite was exited due to an exception, and the return value from the `__exit__()` method was false, the exception is reraised. If the return value was true, the exception is suppressed, and execution continues with the statement following the `with` statement.

   If the suite was exited for any reason other than an exception, the return value from `__exit__()` is ignored, and execution proceeds at the normal location for the kind of exit that was taken.

The following code:

```
with EXPRESSION as TARGET:
    SUITE
```

is semantically equivalent to:

```
manager = (EXPRESSION)
enter = type(manager).__enter__
exit = type(manager).__exit__
value = enter(manager)
hit_except = False

try:
    TARGET = value
    SUITE
except:
    hit_except = True
    if not exit(manager, *sys.exc_info()):
        raise
finally:
    if not hit_except:
        exit(manager, None, None, None)
```

With more than one item, the context managers are processed as if multiple `with` statements were nested:

```
with A() as a, B() as b:
    SUITE
```

is semantically equivalent to:

```
with A() as a:
    with B() as b:
        SUITE
```

You can also write multi-item context managers in multiple lines if the items are surrounded by parentheses. For example:

```
with (
    A() as a,
    B() as b,
):
    SUITE
```

Changed in version 3.1: Support for multiple context expressions.

Changed in version 3.10: Support for using grouping parentheses to break the statement in multiple lines.

See also

PEP 343 [https://peps.python.org/pep-0343/] - The “with” statement

The specification, background, and examples for the Python `with` statement.

8.6. The `match` statement

Added in version 3.10.

The match statement is used for pattern matching. Syntax:

```
match_stmt:   'match' subject_expr ":" NEWLINE INDENT
case_block+ DEDENT
subject_expr: `!star_named_expression` ","
`!star_named_expressions`?
              | `!named_expression`
case_block:   'case' patterns [guard] ":" `!block`
```

Note

This section uses single quotes to denote soft keywords.

Pattern matching takes a pattern as input (following `case`) and a subject value (following `match`). The pattern (which may contain subpatterns) is matched against the subject value. The outcomes are:

- A match success or failure (also termed a pattern success or failure).
- Possible binding of matched values to a name. The prerequisites for this are further discussed below.

The `match` and `case` keywords are soft keywords.

See also

- **PEP 634** [https://peps.python.org/pep-0634/] – Structural Pattern Matching: Specification
- **PEP 636** [https://peps.python.org/pep-0636/] – Structural Pattern Matching: Tutorial

8.6.1. Overview

Here’s an overview of the logical flow of a match statement:

1. The subject expression `subject_expr` is evaluated and a resulting subject value obtained. If the subject expression contains a comma, a tuple is constructed using the standard rules.

2. Each pattern in a `case_block` is attempted to match with the subject value. The specific rules for success or failure are described below. The match attempt can also bind some or all of the standalone names within the pattern. The precise pattern binding rules vary per pattern type and are specified below. **Name bindings made during a successful pattern match outlive the executed block and can be used after the match statement**.

   Note

   During failed pattern matches, some subpatterns may succeed. Do not rely on bindings being made for a failed match. Conversely, do not rely on variables remaining unchanged after a failed match. The exact behavior is dependent on implementation and may vary. This is an intentional decision made to allow different implementations to add optimizations.

3. If the pattern succeeds, the corresponding guard (if present) is evaluated. In this case all name bindings are guaranteed to have happened.
    - If the guard evaluates as true or is missing, the `block` inside `case_block` is executed.
    - Otherwise, the next `case_block` is attempted as described above.
    - If there are no further case blocks, the match statement is completed.

Note

Users should generally never rely on a pattern being evaluated. Depending on implementation, the interpreter may cache values or use other optimizations which skip repeated evaluations.

A sample match statement:

```
>>> flag = False
>>> match (100, 200):
...    case (100, 300):  # Mismatch: 200 != 300
...        print('Case 1')
...    case (100, 200) if flag:  # Successful match, but guard
fails
...        print('Case 2')
...    case (100, y):  # Matches and binds y to 200
...        print(f'Case 3, y: {y}')
...    case _:  # Pattern not attempted
...        print('Case 4, I match anything!')
...
Case 3, y: 200
```

In this case, `if flag` is a guard. Read more about that in the next section.

8.6.2. Guards

```
guard: "if" `!named_expression`
```

A `guard` (which is part of the `case`) must succeed for code inside the `case` block to execute. It takes the form: **`if`** followed by an expression.

The logical flow of a `case` block with a `guard` follows:

1. Check that the pattern in the `case` block succeeded. If the pattern failed, the `guard` is not evaluated and the next `case` block is checked.
2. If the pattern succeeded, evaluate the `guard`.
   - If the `guard` condition evaluates as true, the case block is selected.
   - If the `guard` condition evaluates as false, the case block is not selected.
   - If the `guard` raises an exception during evaluation, the exception bubbles up.

Guards are allowed to have side effects as they are expressions. Guard evaluation must proceed from the first to the last case block, one at a time, skipping case blocks whose pattern(s) don’t all succeed. (I.e., guard evaluation must happen in order.) Guard evaluation must stop once a case block is selected.

8.6.3. Irrefutable Case Blocks

An irrefutable case block is a match-all case block. A match statement may have at most one irrefutable case block, and it must be last.

A case block is considered irrefutable if it has no guard and its pattern is irrefutable. A pattern is considered irrefutable if we can prove from its syntax alone that it will always succeed. Only the following patterns are irrefutable:

- AS Patterns whose left-hand side is irrefutable
- OR Patterns containing at least one irrefutable pattern
- Capture Patterns
- Wildcard Patterns
- parenthesized irrefutable patterns

8.6.4. Patterns

Note

This section uses grammar notations beyond standard EBNF:

- the notation `SEP.RULE+` is shorthand for `RULE (SEP RULE)*`
- the notation `!RULE` is shorthand for a negative lookahead assertion

The top-level syntax for `patterns` is:

```
patterns:       open_sequence_pattern | pattern
pattern:        as_pattern | or_pattern
closed_pattern: | literal_pattern
                | capture_pattern
                | wildcard_pattern
                | value_pattern
                | group_pattern
                | sequence_pattern
                | mapping_pattern
                | class_pattern
```

The descriptions below will include a description “in simple terms” of what a pattern does for illustration purposes (credits to Raymond Hettinger for a document that inspired most of the descriptions). Note that these descriptions are purely for illustration purposes and **may not** reflect the underlying implementation. Furthermore, they do not cover all valid forms.

8.6.4.1. OR Patterns

An OR pattern is two or more patterns separated by vertical bars `|`. Syntax:

```
or_pattern: "|".closed_pattern+
```

Only the final subpattern may be irrefutable, and each subpattern must bind the same set of names to avoid ambiguity.

An OR pattern matches each of its subpatterns in turn to the subject value, until one succeeds. The OR pattern is then considered successful. Otherwise, if none of the subpatterns succeed, the OR pattern fails.

In simple terms, `P1 | P2 | ...` will try to match `P1`, if it fails it will try to match `P2`, succeeding immediately if any succeeds, failing otherwise.

8.6.4.2. AS Patterns

An AS pattern matches an OR pattern on the left of the `as` keyword against a subject. Syntax:

```
as_pattern: or_pattern "as" capture_pattern
```

If the OR pattern fails, the AS pattern fails. Otherwise, the AS pattern binds the subject to the name on the right of the as keyword and succeeds. `capture_pattern` cannot be a `_`.

In simple terms `P as NAME` will match with `P`, and on success it will set `NAME = <subject>`.

8.6.4.3. Literal Patterns

A literal pattern corresponds to most literals in Python. Syntax:

```
literal_pattern: signed_number
                 | signed_number "+" NUMBER
                 | signed_number "-" NUMBER
                 | strings
                 | "None"
                 | "True"
                 | "False"
signed_number:   ["-"] NUMBER
```

The rule `strings` and the token `NUMBER` are defined in the standard Python grammar. Triple-quoted strings are supported. Raw strings and byte strings are supported. f-strings and t-strings are not supported.

The forms `signed_number '+' NUMBER` and `signed_number '-' NUMBER` are for expressing complex numbers; they require a real number on the left and an imaginary number on the right. E.g. `3 + 4j`.

In simple terms, `LITERAL` will succeed only if `<subject> == LITERAL`. For the singletons `None`, `True` and `False`, the `is` operator is used.

8.6.4.4. Capture Patterns

A capture pattern binds the subject value to a name. Syntax:

```
capture_pattern: !'_' NAME
```

A single underscore _ is not a capture pattern (this is what `!'_'` expresses). It is instead treated as a **`wildcard_pattern`**.

In a given pattern, a given name can only be bound once. E.g. `case x, x: ...` is invalid while `case [x] | x: ...` is allowed.

Capture patterns always succeed. The binding follows scoping rules established by the assignment expression operator in **PEP 572** [https://peps.python.org/pep-0572/]; the name becomes a local variable in the closest containing function scope unless there's an applicable **`global`** or **`nonlocal`** statement.

In simple terms `NAME` will always succeed and it will set `NAME = <subject>`.

8.6.4.5. Wildcard Patterns

A wildcard pattern always succeeds (matches anything) and binds no name. Syntax:

```
wildcard_pattern: '_'
```

_ is a soft keyword within any pattern, but only within patterns. It is an identifier, as usual, even within `match` subject expressions, `guard`s, and `case` blocks.

In simple terms, _ will always succeed.

8.6.4.6. Value Patterns

A value pattern represents a named value in Python. Syntax:

```
value_pattern: attr
attr:          name_or_attr "." NAME
name_or_attr:  attr | NAME
```

The dotted name in the pattern is looked up using standard Python name resolution rules. The pattern succeeds if the value found compares equal to the subject value (using the `==` equality operator).

In simple terms `NAME1.NAME2` will succeed only if `<subject> == NAME1.NAME2`

Note

If the same value occurs multiple times in the same match statement, the interpreter may cache the first value found and reuse it rather than repeat the same lookup. This cache is strictly tied to a given execution of a given match statement.

8.6.4.7. Group Patterns

A group pattern allows users to add parentheses around patterns to emphasize the intended grouping. Otherwise, it has no additional syntax. Syntax:

```
group_pattern: "(" pattern ")"
```

In simple terms `(P)` has the same effect as `P`.

8.6.4.8. Sequence Patterns

A sequence pattern contains several subpatterns to be matched against sequence elements. The syntax is similar to the unpacking of a list or tuple.

```
sequence_pattern:       "[" [maybe_sequence_pattern] "]"
                        | "(" [open_sequence_pattern] ")"
open_sequence_pattern:  maybe_star_pattern ","
[maybe_sequence_pattern]
maybe_sequence_pattern: ",".maybe_star_pattern+ ","?
maybe_star_pattern:     star_pattern | pattern
star_pattern:           "*" (capture_pattern |
wildcard_pattern)
```

There is no difference if parentheses or square brackets are used for sequence patterns (i.e. `(...)` vs `[...]`).

Note

A single pattern enclosed in parentheses without a trailing comma (e.g. `(3 | 4)`) is a group pattern. While a single pattern enclosed in square brackets (e.g. `[3 | 4]`) is still a sequence pattern.

At most one star subpattern may be in a sequence pattern. The star subpattern may occur in any position. If no star subpattern is present, the sequence pattern is a fixed-length sequence pattern; otherwise it is a variable-length sequence pattern.

The following is the logical flow for matching a sequence pattern against a subject value:

1. If the subject value is not a sequence [[2]], the sequence pattern fails.

2. If the subject value is an instance of `str`, `bytes` or `bytearray` the sequence pattern fails.

3. The subsequent steps depend on whether the sequence pattern is fixed or variable-length.

   If the sequence pattern is fixed-length:

   1. If the length of the subject sequence is not equal to the number of subpatterns, the sequence pattern fails
   2. Subpatterns in the sequence pattern are matched to their corresponding items in the subject sequence from left to right. Matching stops as soon as a subpattern fails. If all subpatterns succeed in matching their corresponding item, the sequence pattern succeeds.

   Otherwise, if the sequence pattern is variable-length:

   1. If the length of the subject sequence is less than the number of non-star subpatterns, the sequence pattern fails.
   2. The leading non-star subpatterns are matched to their corresponding items as for fixed-length sequences.
   3. If the previous step succeeds, the star subpattern matches a list formed of the remaining subject items, excluding the remaining items corresponding to non-star subpatterns following the star subpattern.
   4. Remaining non-star subpatterns are matched to their corresponding subject items, as for a fixed-length sequence.

   Note

> The length of the subject sequence is obtained via **`len()`** (i.e. via the **`__len__()`** protocol). This length may be cached by the interpreter in a similar manner as value patterns.

In simple terms `[P1, P2, P3, … , P<N>]` matches only if all the following happens:

- check `<subject>` is a sequence
- `len(subject) == <N>`
- `P1` matches `<subject>[0]` (note that this match can also bind names)
- `P2` matches `<subject>[1]` (note that this match can also bind names)
- … and so on for the corresponding pattern/element.

8.6.4.9. Mapping Patterns

A mapping pattern contains one or more key-value patterns. The syntax is similar to the construction of a dictionary. Syntax:

```
mapping_pattern:     "{" [items_pattern] "}"
items_pattern:       ",".key_value_pattern+ ","?
key_value_pattern:   (literal_pattern | value_pattern) ":"
pattern
                     | double_star_pattern
double_star_pattern: "**" capture_pattern
```

At most one double star pattern may be in a mapping pattern. The double star pattern must be the last subpattern in the mapping pattern.

Duplicate keys in mapping patterns are disallowed. Duplicate literal keys will raise a **`SyntaxError`**. Two keys that otherwise have the same value will raise a `ValueError` at runtime.

The following is the logical flow for matching a mapping pattern against a subject value:

1. If the subject value is not a mapping [[3]],the mapping pattern fails.
2. If every key given in the mapping pattern is present in the subject mapping, and the pattern for each key matches the corresponding item of the subject mapping, the mapping pattern succeeds.
3. If duplicate keys are detected in the mapping pattern, the pattern is considered invalid. A `SyntaxError` is raised for duplicate literal values; or a `ValueError` for named keys of the same value.

Note

Key-value pairs are matched using the two-argument form of the mapping subject's `get()` method. Matched key-value pairs must already be present in the mapping, and not created on-the-fly via `__missing__()` or `__getitem__()`.

In simple terms `{KEY1: P1, KEY2: P2, ... }` matches only if all the following happens:

- check `<subject>` is a mapping
- `KEY1 in <subject>`
- `P1` matches `<subject>[KEY1]`
- … and so on for the corresponding KEY/pattern pair.

8.6.4.10. Class Patterns

A class pattern represents a class and its positional and keyword arguments (if any). Syntax:

```
class_pattern:       name_or_attr "(" [pattern_arguments ","?]
")"
pattern_arguments:   positional_patterns ["," keyword_patterns]
                     | keyword_patterns
positional_patterns: ",".pattern+
keyword_patterns:    ",".keyword_pattern+
keyword_pattern:     NAME "=" pattern
```

The same keyword should not be repeated in class patterns.

The following is the logical flow for matching a class pattern against a subject value:

1. If `name_or_attr` is not an instance of the builtin `type` , raise `TypeError`.

2. If the subject value is not an instance of `name_or_attr` (tested via `isinstance()`), the class pattern fails.

3. If no pattern arguments are present, the pattern succeeds. Otherwise, the subsequent steps depend on whether keyword or positional argument patterns are present.

   For a number of built-in types (specified below), a single positional subpattern is accepted which will match the entire subject; for these types keyword patterns also work as for other types.

   If only keyword patterns are present, they are processed as follows, one by one:

   I. The keyword is looked up as an attribute on the subject.

    - If this raises an exception other than `AttributeError`, the exception bubbles up.
    - If this raises `AttributeError`, the class pattern has failed.
    - Else, the subpattern associated with the keyword pattern is matched against the subject’s attribute value. If this fails, the class pattern fails; if this succeeds, the match proceeds to the next keyword.

II. If all keyword patterns succeed, the class pattern succeeds.

If any positional patterns are present, they are converted to keyword patterns using the `__match_args__` attribute on the class `name_or_attr` before matching:

I. The equivalent of `getattr(cls, "__match_args__", ())` is called.

    - If this raises an exception, the exception bubbles up.
    - If the returned value is not a tuple, the conversion fails and `TypeError` is raised.
    - If there are more positional patterns than `len(cls.__match_args__)`, `TypeError` is raised.
    - Otherwise, positional pattern `i` is converted to a keyword pattern using `__match_args__[i]` as the keyword. `__match_args__[i]` must be a string; if not `TypeError` is raised.
    - If there are duplicate keywords, `TypeError` is raised.

See also

[Customizing positional arguments in class pattern matching]

II. Once all positional patterns have been converted to keyword patterns, the match proceeds as if there were only keyword patterns.

For the following built-in types the handling of positional subpatterns is different:

- **`bool`**
- **`bytearray`**
- **`bytes`**
- **`dict`**
- **`float`**
- **`frozenset`**
- **`int`**
- **`list`**
- **`set`**
- **`str`**
- **`tuple`**

These classes accept a single positional argument, and the pattern there is matched against the whole object rather than an attribute. For example `int(0|1)` matches the value `0`, but not the value `0.0`.

In simple terms `CLS(P1, attr=P2)` matches only if the following happens:

- `isinstance(<subject>, CLS)`
- convert `P1` to a keyword pattern using `CLS.__match_args__`
- For each keyword argument `attr=P2`:
  - `hasattr(<subject>, "attr")`
  - `P2` matches `<subject>.attr`

- … and so on for the corresponding keyword argument/pattern pair.

See also

- **PEP 634** [https://peps.python.org/pep-0634/] – Structural Pattern Matching: Specification
- **PEP 636** [https://peps.python.org/pep-0636/] – Structural Pattern Matching: Tutorial

8.7. Function definitions

A function definition defines a user-defined function object (see section The standard type hierarchy):

```
funcdef:                   [decorators] "def" funcname
[type_params] "(" [parameter_list] ")"
                           ["->" expression] ":" suite
decorators:                decorator+
decorator:                 "@" assignment_expression NEWLINE
parameter_list:            defparameter ("," defparameter)* ","
"/" ["," [parameter_list_no_posonly]]
                             | parameter_list_no_posonly
parameter_list_no_posonly: defparameter ("," defparameter)*
["," [parameter_list_starargs]]
                           | parameter_list_starargs
parameter_list_starargs:   "*" [star_parameter] (","
defparameter)* ["," [parameter_star_kwargs]]
                           | "*" ("," defparameter)+ [","
[parameter_star_kwargs]]
                           | parameter_star_kwargs
parameter_star_kwargs:     "**" parameter [","]
parameter:                 identifier [":" expression]
star_parameter:            identifier [":" ["*"] expression]
defparameter:              parameter ["=" expression]
funcname:                  identifier
```

A function definition is an executable statement. Its execution binds the function name in the current local namespace to a function object (a wrapper around the executable code for the function). This function object contains a reference to the current global namespace as the global namespace to be used when the function is called.

The function definition does not execute the function body; this gets executed only when the function is called. [[4]]

A function definition may be wrapped by one or more decorator expressions. Decorator expressions are evaluated when the function is defined, in the scope that contains the function definition. The result must be a callable, which is invoked with the function object as the only argument. The returned value is bound to the function name instead of the function object. Multiple decorators are applied in nested fashion. For example, the following code

```
@f1(arg)
@f2
def func(): pass
```

is roughly equivalent to

```
def func(): pass
func = f1(arg)(f2(func))
```

except that the original function is not temporarily bound to the name `func`.

Changed in version 3.9: Functions may be decorated with any valid **`assignment_expression`**. Previously, the grammar was much more restrictive; see **PEP 614** [https://peps.python.org/pep-0614/] for details.

A list of type parameters may be given in square brackets between the function’s name and the opening parenthesis for its parameter list. This indicates to static type checkers that the function is generic. At runtime, the type parameters can be retrieved from the function’s **`__type_params__`** attribute. See Generic functions for more.

Changed in version 3.12: Type parameter lists are new in Python 3.12.

When one or more parameters have the form *parameter* `=` *expression*, the function is said to have “default parameter values.” For a parameter with a default value, the corresponding argument may be omitted from a call, in which case the parameter’s default value is substituted. If a parameter has a default value, all following parameters up until the “`*`” must also have a default value — this is a syntactic restriction that is not expressed by the grammar.

Default parameter values are evaluated from left to right when the function definition is executed. This means that the expression is evaluated once, when the function is defined, and that the same “pre-computed” value is used for each call. This is especially important to understand when a default parameter value is a mutable object, such as a list or a dictionary: if the function modifies the object (e.g. by appending an item to a list), the default parameter value is in effect modified. This is generally not what was intended. A way around this is to use `None` as the default, and explicitly test for it in the body of the function, e.g.:

```
def whats_on_the_telly(penguin=None):
    if penguin is None:
        penguin = []
    penguin.append("property of the zoo")
    return penguin
```

Function call semantics are described in more detail in section Calls. A function call always assigns values to all parameters mentioned in the parameter list, either from positional arguments, from keyword arguments, or from default values. If the form “`*identifier`” is present, it is initialized to a tuple receiving any excess positional parameters, defaulting to the empty tuple. If the form “`**identifier`” is present, it is initialized to a new ordered mapping receiving any excess keyword arguments, defaulting to a new empty mapping of the same type. Parameters after “`*`” or “`*identifier`” are keyword-only parameters and may only be passed by keyword arguments. Parameters before “`/`” are positional-only parameters and may only be passed by positional arguments.

Changed in version 3.8: The `/` function parameter syntax may be used to indicate positional-only parameters. See **PEP 570** [https://peps.python.org/pep-0570/] for details.

Parameters may have an annotation of the form “`: expression`” following the parameter name. Any parameter may have an annotation, even those of the form `*identifier` or `**identifier`. (As a special case, parameters of the form `*identifier` may have an annotation “`: *expression`”.) Functions may have “return” annotation of the form “`-> expression`” after the parameter list. These annotations can be any valid Python expression. The presence of annotations does not change the semantics of a function. See Annotations for more information on annotations.

Changed in version 3.11: Parameters of the form “`*identifier`” may have an annotation “`: *expression`”. See **PEP 646** [https://peps.python.org/pep-0646/].

It is also possible to create anonymous functions (functions not bound to a name), for immediate use in expressions.

This uses lambda expressions, described in section Lambdas. Note that the lambda expression is merely a shorthand for a simplified function definition; a function defined in a “`def`” statement can be passed around or assigned to another name just like a function defined by a lambda expression. The “`def`” form is actually more powerful since it allows the execution of multiple statements and annotations.

Programmer’s note: Functions are first-class objects. A “`def`” statement executed inside a function definition defines a local function that can be returned or passed around. Free variables used in the nested function can access the local variables of the function containing the def. See section Naming and binding for details.

See also

PEP 3107 [https://peps.python.org/pep-3107/] - Function Annotations
The original specification for function annotations.

PEP 484 [https://peps.python.org/pep-0484/] - Type Hints
Definition of a standard meaning for annotations: type hints.

PEP 526 [https://peps.python.org/pep-0526/] - Syntax for Variable Annotations
Ability to type hint variable declarations, including class variables and instance variables.

PEP 563 [https://peps.python.org/pep-0563/] - Postponed Evaluation of Annotations
Support for forward references within annotations by preserving annotations in a string form at runtime instead of eager evaluation.

PEP 318 [https://peps.python.org/pep-0318/] - Decorators for Functions and Methods

Function and method decorators were introduced. Class decorators were introduced in **PEP 3129** [https://peps.python.org/pep-3129/].

8.8. Class definitions

A class definition defines a class object (see section The standard type hierarchy):

```
classdef:    [decorators] "class" classname [type_params]
[inheritance] ":" suite
inheritance: "(" [argument_list] ")"
classname:   identifier
```

A class definition is an executable statement. The inheritance list usually gives a list of base classes (see Metaclasses for more advanced uses), so each item in the list should evaluate to a class object which allows subclassing. Classes without an inheritance list inherit, by default, from the base class `object`; hence,

```
class Foo:
    pass
```

is equivalent to

```
class Foo(object):
    pass
```

The class’s suite is then executed in a new execution frame (see Naming and binding), using a newly created local namespace and the original global namespace. (Usually, the suite contains mostly function definitions.) When the class’s suite finishes execution, its execution frame is discarded but its local namespace is saved. [[5]] A class object is then created using the inheritance list for the base classes and the saved local namespace for the attribute dictionary. The class name is bound to this class object in the original local namespace.

The order in which attributes are defined in the class body is preserved in the new class's **`__dict__`**. Note that this is reliable only right after the class is created and only for classes that were defined using the definition syntax.

Class creation can be customized heavily using metaclasses.

Classes can also be decorated: just like when decorating functions,

```
@f1(arg)
@f2
class Foo: pass
```

is roughly equivalent to

```
class Foo: pass
Foo = f1(arg)(f2(Foo))
```

The evaluation rules for the decorator expressions are the same as for function decorators. The result is then bound to the class name.

Changed in version 3.9: Classes may be decorated with any valid **`assignment_expression`**. Previously, the grammar was much more restrictive; see **PEP 614** [https://peps.python.org/pep-0614/] for details.

A list of type parameters may be given in square brackets immediately after the class's name. This indicates to static type checkers that the class is generic. At runtime, the type parameters can be retrieved from the class’s `__type_params__` attribute. See Generic classes for more.

Changed in version 3.12: Type parameter lists are new in Python 3.12.

Programmer’s note: Variables defined in the class definition are class attributes; they are shared by instances. Instance attributes can be set in a method with `self.name = value`. Both class and instance attributes are accessible through the notation “`self.name`”, and an instance attribute hides a class attribute with the same name when accessed in this way. Class attributes can be used as defaults for instance attributes, but using mutable values there can lead to unexpected results. Descriptors can be used to create instance variables with different implementation details.

See also

PEP 3115 [https://peps.python.org/pep-3115/] - Metaclasses in Python 3000

The proposal that changed the declaration of metaclasses to the current syntax, and the semantics for how classes with metaclasses are constructed.

PEP 3129 [https://peps.python.org/pep-3129/] - Class Decorators

The proposal that added class decorators. Function and method decorators were introduced in **PEP 318** [https://peps.python.org/pep-0318/].

8.9. Coroutines

Added in version 3.5.

8.9.1. Coroutine function definition

```
async_funcdef: [decorators] "async" "def" funcname "("
[parameter_list] ")"
               ["->" expression] ":" suite
```

Execution of Python coroutines can be suspended and resumed at many points (see coroutine). `await` expressions, `async for` and `async with` can only be used in the body of a coroutine function.

Functions defined with `async def` syntax are always coroutine functions, even if they do not contain `await` or `async` keywords.

It is a `SyntaxError` to use a `yield from` expression inside the body of a coroutine function.

An example of a coroutine function:

```
async def func(param1, param2):
    do_stuff()
    await some_coroutine()
```

Changed in version 3.7: `await` and `async` are now keywords; previously they were only treated as such inside the body of a coroutine function.

8.9.2. The `async for` statement

```
async_for_stmt: "async" for_stmt
```

An asynchronous iterable provides an `__aiter__` method that directly returns an asynchronous iterator, which can call asynchronous code in its `__anext__` method.

The `async for` statement allows convenient iteration over asynchronous iterables.

The following code:

```
async for TARGET in ITER:
    SUITE
else:
    SUITE2
```

Is semantically equivalent to:

```
iter = (ITER)
iter = type(iter).__aiter__(iter)
running = True

while running:
    try:
        TARGET = await type(iter).__anext__(iter)
    except StopAsyncIteration:
        running = False
    else:
        SUITE
else:
    SUITE2
```

See also **`__aiter__()`** and **`__anext__()`** for details.

It is a **`SyntaxError`** to use an `async for` statement outside the body of a coroutine function.

8.9.3. The `async with` statement

```
async_with_stmt: "async" with_stmt
```

An asynchronous context manager is a context manager that is able to suspend execution in its *enter* and *exit* methods.

The following code:

```
async with EXPRESSION as TARGET:
    SUITE
```

is semantically equivalent to:

```
manager = (EXPRESSION)
aenter = type(manager).__aenter__
aexit = type(manager).__aexit__
value = await aenter(manager)
hit_except = False

try:
    TARGET = value
    SUITE
except:
    hit_except = True
    if not await aexit(manager, *sys.exc_info()):
        raise
finally:
    if not hit_except:
        await aexit(manager, None, None, None)
```

See also `__aenter__()` and `__aexit__()` for details.

It is a `SyntaxError` to use an `async with` statement outside the body of a coroutine function.

See also

PEP 492 [https://peps.python.org/pep-0492/] - Coroutines with async and await syntax
The proposal that made coroutines a proper standalone concept in Python, and added supporting syntax.

8.10. Type parameter lists

Added in version 3.12.

Changed in version 3.13: Support for default values was added (see **PEP 696** [https://peps.python.org/pep-0696/]).

```
type_params:  "[" type_param ("," type_param)* "]"
type_param:   typevar | typevartuple | paramspec
typevar:      identifier (":" expression)? ("=" expression)?
typevartuple: "*" identifier ("=" expression)?
paramspec:    "**" identifier ("=" expression)?
```

Functions (including coroutines), classes and type aliases may contain a type parameter list:

```
def max[T](args: list[T]) -> T:
    ...

async def amax[T](args: list[T]) -> T:
    ...

class Bag[T]:
    def __iter__(self) -> Iterator[T]:
        ...

    def add(self, arg: T) -> None:
        ...

type ListOrSet[T] = list[T] | set[T]
```

Semantically, this indicates that the function, class, or type alias is generic over a type variable. This information is primarily used by static type checkers, and at runtime, generic objects behave much like their non-generic counterparts.

Type parameters are declared in square brackets (`[]`) immediately after the name of the function, class, or type alias. The type parameters are accessible within the scope of the generic object, but not elsewhere. Thus, after a declaration `def func[T](): pass`, the name `T` is not available in the module scope. Below, the semantics of generic objects are described with more precision. The scope of type parameters is modeled with a special function (technically, an annotation scope) that wraps the creation of the generic object.

Generic functions, classes, and type aliases have a **`__type_params__`** attribute listing their type parameters.

Type parameters come in three kinds:

- **`typing.TypeVar`**, introduced by a plain name (e.g., `T`). Semantically, this represents a single type to a type checker.
- **`typing.TypeVarTuple`**, introduced by a name prefixed with a single asterisk (e.g., `*Ts`). Semantically, this stands for a tuple of any number of types.
- **`typing.ParamSpec`**, introduced by a name prefixed with two asterisks (e.g., `**P`). Semantically, this stands for the parameters of a callable.

`typing.TypeVar` declarations can define *bounds* and *constraints* with a colon (`:`) followed by an expression. A single expression after the colon indicates a bound (e.g. `T: int`). Semantically, this means that the **`typing.TypeVar`** can only represent types that are a subtype of this bound. A parenthesized tuple of expressions after the colon indicates a set of constraints (e.g. `T: (str, bytes)`). Each member of the tuple should be a type (again, this is not enforced at runtime). Constrained type variables can only take on one of the types in the list of constraints.

For **`typing.TypeVar`**s declared using the type parameter list syntax, the bound and constraints are not evaluated when the generic object is created, but only when the value is explicitly accessed through the attributes `__bound__` and `__constraints__`. To accomplish this, the bounds or constraints are evaluated in a separate annotation scope.

`typing.TypeVarTuple`s and **`typing.ParamSpec`**s cannot have bounds or constraints.

All three flavors of type parameters can also have a *default value*, which is used when the type parameter is not explicitly provided. This is added by appending a single equals sign (`=`) followed by an expression. Like the bounds and constraints of type variables, the default value is not evaluated when the object is created, but only when the type parameter's `__default__` attribute is accessed. To this end, the default value is evaluated in a separate annotation scope. If no default value is specified for a type parameter, the `__default__` attribute is set to the special sentinel object **`typing.NoDefault`**.

The following example indicates the full set of allowed type parameter declarations:

```
def overly_generic[
   SimpleTypeVar,
   TypeVarWithDefault = int,
   TypeVarWithBound: int,
   TypeVarWithConstraints: (str, bytes),
   *SimpleTypeVarTuple = (int, float),
   **SimpleParamSpec = (str, bytearray),
](
   a: SimpleTypeVar,
   b: TypeVarWithDefault,
   c: TypeVarWithBound,
   d: Callable[SimpleParamSpec, TypeVarWithConstraints],
   *e: SimpleTypeVarTuple,
): ...
```

8.10.1. Generic functions

Generic functions are declared as follows:

```
def func[T](arg: T): ...
```

This syntax is equivalent to:

```
annotation-def TYPE_PARAMS_OF_func():
    T = typing.TypeVar("T")
    def func(arg: T): ...
    func.__type_params__ = (T,)
    return func
func = TYPE_PARAMS_OF_func()
```

Here `annotation-def` indicates an annotation scope, which is not actually bound to any name at runtime. (One other liberty is taken in the translation: the syntax does not go through attribute access on the **`typing`** module, but creates an instance of **`typing.TypeVar`** directly.)

The annotations of generic functions are evaluated within the annotation scope used for declaring the type parameters, but the function’s defaults and decorators are not.

The following example illustrates the scoping rules for these cases, as well as for additional flavors of type parameters:

```
@decorator
def func[T: int, *Ts, **P](*args: *Ts, arg: Callable[P, T] = 
some_default):
    ...
```

Except for the lazy evaluation of the **`TypeVar`** bound, this is equivalent to:

```
DEFAULT_OF_arg = some_default

annotation-def TYPE_PARAMS_OF_func():

    annotation-def BOUND_OF_T():
        return int
    # In reality, BOUND_OF_T() is evaluated only on demand.
    T = typing.TypeVar("T", bound=BOUND_OF_T())

    Ts = typing.TypeVarTuple("Ts")
    P = typing.ParamSpec("P")

    def func(*args: *Ts, arg: Callable[P, T] = DEFAULT_OF_arg):
        ...

    func.__type_params__ = (T, Ts, P)
    return func
func = decorator(TYPE_PARAMS_OF_func())
```

The capitalized names like `DEFAULT_OF_arg` are not actually bound at runtime.

8.10.2. Generic classes

Generic classes are declared as follows:

```
class Bag[T]: ...
```

This syntax is equivalent to:

```
annotation-def TYPE_PARAMS_OF_Bag():
    T = typing.TypeVar("T")
    class Bag(typing.Generic[T]):
        __type_params__ = (T,)
        ...
    return Bag
Bag = TYPE_PARAMS_OF_Bag()
```

Here again `annotation-def` (not a real keyword) indicates an annotation scope, and the name `TYPE_PARAMS_OF_Bag` is not actually bound at runtime.

Generic classes implicitly inherit from **`typing.Generic`**. The base classes and keyword arguments of generic classes are evaluated within the type scope for the type parameters, and decorators are evaluated outside that scope. This is illustrated by this example:

```
@decorator
class Bag(Base[T], arg=T): ...
```

This is equivalent to:

```
annotation-def TYPE_PARAMS_OF_Bag():
    T = typing.TypeVar("T")
    class Bag(Base[T], typing.Generic[T], arg=T):
        __type_params__ = (T,)
        ...
    return Bag
Bag = decorator(TYPE_PARAMS_OF_Bag())
```

8.10.3. Generic type aliases

The **`type`** statement can also be used to create a generic type alias:

```
type ListOrSet[T] = list[T] | set[T]
```

Except for the lazy evaluation of the value, this is equivalent to:

```
annotation-def TYPE_PARAMS_OF_ListOrSet():
    T = typing.TypeVar("T")

    annotation-def VALUE_OF_ListOrSet():
        return list[T] | set[T]
    # In reality, the value is lazily evaluated
    return typing.TypeAliasType("ListOrSet",
VALUE_OF_ListOrSet(), type_params=(T,))
ListOrSet = TYPE_PARAMS_OF_ListOrSet()
```

Here, `annotation-def` (not a real keyword) indicates an annotation scope. The capitalized names like `TYPE_PARAMS_OF_ListOrSet` are not actually bound at runtime.

8.11. Annotations

Changed in version 3.14: Annotations are now lazily evaluated by default.

Variables and function parameters may carry annotations, created by adding a colon after the name, followed by an expression:

```
x: annotation = 1
def f(param: annotation): ...
```

Functions may also carry a return annotation following an arrow:

```
def f() -> annotation: ...
```

Annotations are conventionally used for type hints, but this is not enforced by the language, and in general annotations may contain arbitrary expressions. The presence of annotations does not change the runtime semantics of the code, except if some mechanism is used that introspects and uses the annotations (such as **`dataclasses`** or **`functools.singledispatch()`**).

By default, annotations are lazily evaluated in an annotation scope. This means that they are not evaluated when the code containing the annotation is evaluated. Instead, the interpreter saves information that can be used to evaluate the annotation later if requested. The **`annotationlib`** module provides tools for evaluating annotations.

If the future statement `from __future__ import annotations` is present, all annotations are instead stored as strings:

```
>>> from __future__ import annotations
>>> def f(param: annotation): ...
>>> f.__annotations__
{'param': 'annotation'}
```

This future statement will be deprecated and removed in a future version of Python, but not before Python 3.13 reaches its end of life (see **PEP 749** [https://peps.python.org/pep-0749/]). When it is used, introspection tools like `annotationlib.get_annotations()` and `typing.get_type_hints()` are less likely to be able to resolve annotations at runtime.

Footnotes

[1]

The exception is propagated to the invocation stack unless there is a `finally` clause which happens to raise another exception. That new exception causes the old one to be lost.

[2]

In pattern matching, a sequence is defined as one of the following:

- a class that inherits from `collections.abc.Sequence`
- a Python class that has been registered as `collections.abc.Sequence`
- a builtin class that has its (CPython) `Py_TPFLAGS_SEQUENCE` bit set
- a class that inherits from any of the above

The following standard library classes are sequences:

- `array.array`
- `collections.deque`
- `list`
- `memoryview`
- `range`

- `tuple`

Note

Subject values of type `str`, `bytes`, and `bytearray` do not match sequence patterns.

[3]

In pattern matching, a mapping is defined as one of the following:

- a class that inherits from `collections.abc.Mapping`
- a Python class that has been registered as `collections.abc.Mapping`
- a builtin class that has its (CPython) `Py_TPFLAGS_MAPPING` bit set
- a class that inherits from any of the above

The standard library classes `dict` and `types.MappingProxyType` are mappings.

[4]

A string literal appearing as the first statement in the function body is transformed into the function's `__doc__` attribute and therefore the function's docstring.

[5]

A string literal appearing as the first statement in the class body is transformed into the namespace's `__doc__` item and therefore the class's docstring.

9. Top-level components

The Python interpreter can get its input from a number of sources: from a script passed to it as standard input or as program argument, typed in interactively, from a module source file, etc. This chapter gives the syntax used in these cases.

9.1. Complete Python programs

While a language specification need not prescribe how the language interpreter is invoked, it is useful to have a notion of a complete Python program. A complete Python program is executed in a minimally initialized environment: all built-in and standard modules are available, but none have been initialized, except for `sys` (various system services), `builtins` (built-in functions, exceptions and `None`) and `__main__`. The latter is used to provide the local and global namespace for execution of the complete program.

The syntax for a complete Python program is that for file input, described in the next section.

The interpreter may also be invoked in interactive mode; in this case, it does not read and execute a complete program but reads and executes one statement (possibly compound) at a time. The initial environment is identical to that of a complete program; each statement is executed in the namespace of `__main__`.

A complete program can be passed to the interpreter in three forms: with the `-c` *string* command line option, as a file passed as the first command line argument, or as standard input. If the file or standard input is a tty device, the interpreter enters interactive mode; otherwise, it executes the file as a complete program.

9.2. File input

All input read from non-interactive files has the same form:

```
file_input: (NEWLINE | statement)* ENDMARKER
```

This syntax is used in the following situations:

- when parsing a complete Python program (from a file or from a string);
- when parsing a module;
- when parsing a string passed to the `exec()` function;

9.3. Interactive input

Input in interactive mode is parsed using the following grammar:

```
interactive_input: [stmt_list] NEWLINE | compound_stmt NEWLINE
| ENDMARKER
```

Note that a (top-level) compound statement must be followed by a blank line in interactive mode; this is needed to help the parser detect the end of the input.

9.4. Expression input

`eval()` is used for expression input. It ignores leading whitespace. The string argument to `eval()` must have the following form:

```
eval_input: expression_list NEWLINE* ENDMARKER
```

10. Full Grammar specification

This is the full Python grammar, derived directly from the grammar used to generate the CPython parser (see Grammar/python.gram [https://github.com/python/cpython/tree/3.14/Grammar/python.gram]). The version here omits details related to code generation and error recovery.

The notation used here is the same as in the preceding docs, and is described in the notation section, except for an extra complication:

- `~` ("cut"): commit to the current alternative; fail the rule if the alternative fails to parse

  Python mainly uses cuts for optimizations or improved error messages. They often appear to be useless in the listing below.

  Cuts currently don't appear inside parentheses, brackets, lookaheads and similar. Their behavior in these contexts is deliberately left unspecified.

```
# PEG grammar for Python



# ========================= START OF THE GRAMMAR
=========================

# General grammatical elements and rules:
#
# * Strings with double quotes (") denote SOFT KEYWORDS
# * Strings with single quotes (') denote KEYWORDS
# * Upper case names (NAME) denote tokens in the Grammar/Tokens
file
# * Rule names starting with "invalid_" are used for
specialized syntax errors
#     - These rules are NOT used in the first pass of the
parser.
#     - Only if the first pass fails to parse, a second pass
including the invalid
#       rules will be executed.
#     - If the parser fails in the second phase with a generic
syntax error, the
#       location of the generic failure of the first pass will
be used (this avoids
#       reporting incorrect locations due to the invalid
rules).
#     - The order of the alternatives involving invalid rules
matter
#       (like any rule in PEG).
#
# Grammar Syntax (see PEP 617 for more information):
#
# rule_name: expression
#   Optionally, a type can be included right after the rule
name, which
#   specifies the return type of the C or Python function
corresponding to the
#   rule:
# rule_name[return_type]: expression
#   If the return type is omitted, then a void * is returned in
C and an Any in
#   Python.
# e1 e2
#   Match e1, then match e2.
# e1 | e2
#   Match e1 or e2.
#   The first alternative can also appear on the line after the
rule name for
#   formatting purposes. In that case, a | must be used before
the first
#   alternative, like so:
#       rule_name[return_type]:
#            | first_alt
#            | second_alt
# ( e )
#   Match e (allows also to use other operators in the group
like '(e)*')
# [ e ] or e?
#   Optionally match e.
# e*
#   Match zero or more occurrences of e.
# e+
#   Match one or more occurrences of e.
# s.e+
#   Match one or more occurrences of e, separated by s. The
generated parse tree
#   does not include the separator. This is otherwise identical
to (e (s e)*).
# &e
#   Succeed if e can be parsed, without consuming any input.
# !e
#   Fail if e can be parsed, without consuming any input.
# ~
#   Commit to the current alternative, even if it fails to
parse.
# &&e
#   Eager parse e. The parser will not backtrack and will
immediately
#   fail with SyntaxError if e cannot be parsed.
#

# STARTING RULES
# ==============

file: [statements] ENDMARKER
interactive: statement_newline
eval: expressions NEWLINE* ENDMARKER
func_type: '(' [type_expressions] ')' '->' expression NEWLINE*
ENDMARKER

# GENERAL STATEMENTS
# ==================

statements: statement+

statement:
    | compound_stmt
    | simple_stmts

single_compound_stmt:
    | compound_stmt

statement_newline:
    | single_compound_stmt NEWLINE
    | simple_stmts
    | NEWLINE
    | ENDMARKER

simple_stmts:
    | simple_stmt !';' NEWLINE  # Not needed, there for speedup
    | ';'.simple_stmt+ [';'] NEWLINE

# NOTE: assignment MUST precede expression, else parsing a
simple assignment
# will throw a SyntaxError.
simple_stmt:
    | assignment
    | type_alias
    | star_expressions
    | return_stmt
    | import_stmt
    | raise_stmt
    | pass_stmt
    | del_stmt
    | yield_stmt
    | assert_stmt
    | break_stmt
    | continue_stmt
    | global_stmt
    | nonlocal_stmt

compound_stmt:
    | function_def
    | if_stmt
    | class_def
    | with_stmt
    | for_stmt
    | try_stmt
    | while_stmt
    | match_stmt

# SIMPLE STATEMENTS
# =================

# NOTE: annotated_rhs may start with 'yield'; yield_expr must
start with 'yield'
assignment:
    | NAME ':' expression ['=' annotated_rhs ]
    | ('(' single_target ')'
         | single_subscript_attribute_target) ':' expression
['=' annotated_rhs ]
    | (star_targets '=' )+ annotated_rhs !'=' [TYPE_COMMENT]
    | single_target augassign ~ annotated_rhs

annotated_rhs: yield_expr | star_expressions

augassign:
    | '+='
    | '-='
    | '*='
    | '@='
    | '/='
    | '%='
    | '&='
    | '|='
    | '^='
    | '<<='
    | '>>='
    | '**='
    | '//='

return_stmt:
    | 'return' [star_expressions]

raise_stmt:
    | 'raise' expression ['from' expression ]
    | 'raise'

pass_stmt:
    | 'pass'

break_stmt:
    | 'break'

continue_stmt:
    | 'continue'

global_stmt: 'global' ','.NAME+

nonlocal_stmt: 'nonlocal' ','.NAME+

del_stmt:
    | 'del' del_targets &(';' | NEWLINE)

yield_stmt: yield_expr

assert_stmt: 'assert' expression [',' expression ]

import_stmt:
    | import_name
    | import_from

# Import statements
# -----------------

import_name: 'import' dotted_as_names
# note below: the ('.' | '...') is necessary because '...' is
tokenized as ELLIPSIS
import_from:
    | 'from' ('.' | '...')* dotted_name 'import'
import_from_targets
    | 'from' ('.' | '...')+ 'import' import_from_targets
import_from_targets:
    | '(' import_from_as_names [','] ')'
    | import_from_as_names !','
    | '*'
import_from_as_names:
    | ','.import_from_as_name+
import_from_as_name:
    | NAME ['as' NAME ]

dotted_as_names:
    | ','.dotted_as_name+
dotted_as_name:
    | dotted_name ['as' NAME ]
```

```
dotted_name:
    | dotted_name '.' NAME
    | NAME

# COMPOUND STATEMENTS
# ===================

# Common elements
# ---------------

block:
    | NEWLINE INDENT statements DEDENT
    | simple_stmts

decorators: ('@' named_expression NEWLINE )+

# Class definitions
# -----------------

class_def:
    | decorators class_def_raw
    | class_def_raw

class_def_raw:
    | 'class' NAME [type_params] ['(' [arguments] ')' ] ':'
block

# Function definitions
# --------------------

function_def:
    | decorators function_def_raw
    | function_def_raw

function_def_raw:
    | 'def' NAME [type_params] '(' [params] ')' ['->'
expression ] ':' [func_type_comment] block
    | 'async' 'def' NAME [type_params] '(' [params] ')' ['->'
expression ] ':' [func_type_comment] block

# Function parameters
# -------------------
```

```
params:
    | parameters

parameters:
    | slash_no_default param_no_default* param_with_default*
[star_etc]
    | slash_with_default param_with_default* [star_etc]
    | param_no_default+ param_with_default* [star_etc]
    | param_with_default+ [star_etc]
    | star_etc

# Some duplication here because we can't write (',' | &')'),
# which is because we don't support empty alternatives (yet).

slash_no_default:
    | param_no_default+ '/' ','
    | param_no_default+ '/' &')'
slash_with_default:
    | param_no_default* param_with_default+ '/' ','
    | param_no_default* param_with_default+ '/' &')'

star_etc:
    | '*' param_no_default param_maybe_default* [kwds]
    | '*' param_no_default_star_annotation param_maybe_default*
[kwds]
    | '*' ',' param_maybe_default+ [kwds]
    | kwds

kwds:
    | '**' param_no_default

# One parameter.  This *includes* a following comma and type
comment.
#
# There are three styles:
# - No default
# - With default
# - Maybe with default
#
# There are two alternative forms of each, to deal with type
comments:
# - Ends in a comma followed by an optional type comment
# - No comma, optional type comment, must be followed by close
paren
# The latter form is for a final parameter without trailing
comma.
#

param_no_default:
    | param ',' TYPE_COMMENT?
    | param TYPE_COMMENT? &')'
param_no_default_star_annotation:
    | param_star_annotation ',' TYPE_COMMENT?
    | param_star_annotation TYPE_COMMENT? &')'
param_with_default:
    | param default ',' TYPE_COMMENT?
    | param default TYPE_COMMENT? &')'
param_maybe_default:
    | param default? ',' TYPE_COMMENT?
    | param default? TYPE_COMMENT? &')'
param: NAME annotation?
param_star_annotation: NAME star_annotation
annotation: ':' expression
star_annotation: ':' star_expression
default: '=' expression  | invalid_default

# If statement
# ------------

if_stmt:
    | 'if' named_expression ':' block elif_stmt
    | 'if' named_expression ':' block [else_block]
elif_stmt:
    | 'elif' named_expression ':' block elif_stmt
    | 'elif' named_expression ':' block [else_block]
else_block:
    | 'else' ':' block

# While statement
# ---------------

while_stmt:
    | 'while' named_expression ':' block [else_block]

# For statement
# ------------

for_stmt:
    | 'for' star_targets 'in' ~ star_expressions ':'
[TYPE_COMMENT] block [else_block]
    | 'async' 'for' star_targets 'in' ~ star_expressions ':'
[TYPE_COMMENT] block [else_block]

# With statement
# --------------

with_stmt:
    | 'with' '(' ','.with_item+ ','? ')' ':' [TYPE_COMMENT]
block
    | 'with' ','.with_item+ ':' [TYPE_COMMENT] block
    | 'async' 'with' '(' ','.with_item+ ','? ')' ':' block
    | 'async' 'with' ','.with_item+ ':' [TYPE_COMMENT] block

with_item:
    | expression 'as' star_target &(',' | ')' | ':')
    | expression

# Try statement
# -------------

try_stmt:
    | 'try' ':' block finally_block
    | 'try' ':' block except_block+ [else_block]
[finally_block]
    | 'try' ':' block except_star_block+ [else_block]
[finally_block]


# Except statement
# ----------------

except_block:
    | 'except' expression ':' block
    | 'except' expression 'as' NAME ':' block
    | 'except' expressions ':' block
    | 'except' ':' block
except_star_block:
    | 'except' '*' expression ':' block
    | 'except' '*' expression 'as' NAME ':' block
    | 'except' '*' expressions ':' block
finally_block:
    | 'finally' ':' block

# Match statement
# ---------------

match_stmt:
    | "match" subject_expr ':' NEWLINE INDENT case_block+
DEDENT

subject_expr:
    | star_named_expression ',' star_named_expressions?
    | named_expression

case_block:
    | "case" patterns guard? ':' block

guard: 'if' named_expression

patterns:
    | open_sequence_pattern
    | pattern

pattern:
    | as_pattern
    | or_pattern

as_pattern:
    | or_pattern 'as' pattern_capture_target

or_pattern:
    | '|'.closed_pattern+

closed_pattern:
    | literal_pattern
    | capture_pattern
    | wildcard_pattern
    | value_pattern
    | group_pattern
    | sequence_pattern
    | mapping_pattern
    | class_pattern

# Literal patterns are used for equality and identity
constraints
literal_pattern:
    | signed_number !('+' | '-')
    | complex_number
    | strings
    | 'None'
    | 'True'
    | 'False'

# Literal expressions are used to restrict permitted mapping
pattern keys
literal_expr:
    | signed_number !('+' | '-')
    | complex_number
    | strings
    | 'None'
    | 'True'
    | 'False'

complex_number:
    | signed_real_number '+' imaginary_number
    | signed_real_number '-' imaginary_number

signed_number:
    | NUMBER
    | '-' NUMBER

signed_real_number:
    | real_number
    | '-' real_number

real_number:
    | NUMBER

imaginary_number:
    | NUMBER

capture_pattern:
    | pattern_capture_target
```

```
pattern_capture_target:
    | !"_" NAME !('.' | '(' | '=')

wildcard_pattern:
    | "_"

value_pattern:
    | attr !('.' | '(' | '=')

attr:
    | name_or_attr '.' NAME

name_or_attr:
    | attr
    | NAME

group_pattern:
    | '(' pattern ')'

sequence_pattern:
    | '[' maybe_sequence_pattern? ']'
    | '(' open_sequence_pattern? ')'

open_sequence_pattern:
    | maybe_star_pattern ',' maybe_sequence_pattern?

maybe_sequence_pattern:
    | ','.maybe_star_pattern+ ','?

maybe_star_pattern:
    | star_pattern
    | pattern

star_pattern:
    | '*' pattern_capture_target
    | '*' wildcard_pattern

mapping_pattern:
    | '{' '}'
    | '{' double_star_pattern ','? '}'
    | '{' items_pattern ',' double_star_pattern ','? '}'
    | '{' items_pattern ','? '}'
```

```
items_pattern:
    | ','.key_value_pattern+

key_value_pattern:
    | (literal_expr | attr) ':' pattern

double_star_pattern:
    | '**' pattern_capture_target

class_pattern:
    | name_or_attr '(' ')'
    | name_or_attr '(' positional_patterns ','? ')'
    | name_or_attr '(' keyword_patterns ','? ')'
    | name_or_attr '(' positional_patterns ',' keyword_patterns
',' ? ')'

positional_patterns:
    | ','.pattern+

keyword_patterns:
    | ','.keyword_pattern+

keyword_pattern:
    | NAME '=' pattern

# Type statement
# ---------------

type_alias:
    | "type" NAME [type_params] '=' expression

# Type parameter declaration
# --------------------------

type_params:
    | '[' type_param_seq ']'

type_param_seq: ','.type_param+ [',']

type_param:
    | NAME [type_param_bound] [type_param_default]
    | '*' NAME [type_param_starred_default]
    | '**' NAME [type_param_default]
```

```
type_param_bound: ':' expression
type_param_default: '=' expression
type_param_starred_default: '=' star_expression

# EXPRESSIONS
# -----------

expressions:
    | expression (',' expression )+ [',']
    | expression ','
    | expression

expression:
    | disjunction 'if' disjunction 'else' expression
    | disjunction
    | lambdef

yield_expr:
    | 'yield' 'from' expression
    | 'yield' [star_expressions]

star_expressions:
    | star_expression (',' star_expression )+ [',']
    | star_expression ','
    | star_expression

star_expression:
    | '*' bitwise_or
    | expression

star_named_expressions: ','.star_named_expression+ [',']

star_named_expression:
    | '*' bitwise_or
    | named_expression

assignment_expression:
    | NAME ':=' ~ expression

named_expression:
    | assignment_expression
    | expression !':='
```

```
disjunction:
    | conjunction ('or' conjunction )+
    | conjunction

conjunction:
    | inversion ('and' inversion )+
    | inversion

inversion:
    | 'not' inversion
    | comparison

# Comparison operators
# --------------------

comparison:
    | bitwise_or compare_op_bitwise_or_pair+
    | bitwise_or

compare_op_bitwise_or_pair:
    | eq_bitwise_or
    | noteq_bitwise_or
    | lte_bitwise_or
    | lt_bitwise_or
    | gte_bitwise_or
    | gt_bitwise_or
    | notin_bitwise_or
    | in_bitwise_or
    | isnot_bitwise_or
    | is_bitwise_or

eq_bitwise_or: '==' bitwise_or
noteq_bitwise_or:
    | ('!=' ) bitwise_or
lte_bitwise_or: '<=' bitwise_or
lt_bitwise_or: '<' bitwise_or
gte_bitwise_or: '>=' bitwise_or
gt_bitwise_or: '>' bitwise_or
notin_bitwise_or: 'not' 'in' bitwise_or
in_bitwise_or: 'in' bitwise_or
isnot_bitwise_or: 'is' 'not' bitwise_or
is_bitwise_or: 'is' bitwise_or
```

```
# Bitwise operators
# -----------------

bitwise_or:
    | bitwise_or '|' bitwise_xor
    | bitwise_xor

bitwise_xor:
    | bitwise_xor '^' bitwise_and
    | bitwise_and

bitwise_and:
    | bitwise_and '&' shift_expr
    | shift_expr

shift_expr:
    | shift_expr '<<' sum
    | shift_expr '>>' sum
    | sum

# Arithmetic operators
# --------------------

sum:
    | sum '+' term
    | sum '-' term
    | term

term:
    | term '*' factor
    | term '/' factor
    | term '//' factor
    | term '%' factor
    | term '@' factor
    | factor

factor:
    | '+' factor
    | '-' factor
    | '~' factor
    | power
```

```
power:
    | await_primary '**' factor
    | await_primary

# Primary elements
# ----------------

# Primary elements are things like "obj.something.something",
"obj[something]", "obj(something)", "obj" ...

await_primary:
    | 'await' primary
    | primary

primary:
    | primary '.' NAME
    | primary genexp
    | primary '(' [arguments] ')'
    | primary '[' slices ']'
    | atom

slices:
    | slice !','
    | ','.(slice | starred_expression)+ [',']

slice:
    | [expression] ':' [expression] [':' [expression] ]
    | named_expression

atom:
    | NAME
    | 'True'
    | 'False'
    | 'None'
    | strings
    | NUMBER
    | (tuple | group | genexp)
    | (list | listcomp)
    | (dict | set | dictcomp | setcomp)
    | '...'

group:
    | '(' (yield_expr | named_expression) ')'
```

```
# Lambda functions
# ----------------

lambdef:
    | 'lambda' [lambda_params] ':' expression

lambda_params:
    | lambda_parameters

# lambda_parameters etc. duplicates parameters but without
annotations
# or type comments, and if there's no comma after a parameter,
we expect
# a colon, not a close parenthesis.  (For more, see parameters
above.)
#
lambda_parameters:
    | lambda_slash_no_default lambda_param_no_default*
lambda_param_with_default* [lambda_star_etc]
    | lambda_slash_with_default lambda_param_with_default*
[lambda_star_etc]
    | lambda_param_no_default+ lambda_param_with_default*
[lambda_star_etc]
    | lambda_param_with_default+ [lambda_star_etc]
    | lambda_star_etc

lambda_slash_no_default:
    | lambda_param_no_default+ '/' ','
    | lambda_param_no_default+ '/' &':'

lambda_slash_with_default:
    | lambda_param_no_default* lambda_param_with_default+ '/'
','
    | lambda_param_no_default* lambda_param_with_default+ '/'
&':'

lambda_star_etc:
    | '*' lambda_param_no_default lambda_param_maybe_default*
[lambda_kwds]
    | '*' ',' lambda_param_maybe_default+ [lambda_kwds]
    | lambda_kwds
```

```
lambda_kwds:
    | '**' lambda_param_no_default

lambda_param_no_default:
    | lambda_param ','
    | lambda_param &':'
lambda_param_with_default:
    | lambda_param default ','
    | lambda_param default &':'
lambda_param_maybe_default:
    | lambda_param default? ','
    | lambda_param default? &':'
lambda_param: NAME

# LITERALS
# ========

fstring_middle:
    | fstring_replacement_field
    | FSTRING_MIDDLE
fstring_replacement_field:
    | '{' annotated_rhs '='? [fstring_conversion]
[fstring_full_format_spec] '}'
fstring_conversion:
    | "!" NAME
fstring_full_format_spec:
    | ':' fstring_format_spec*
fstring_format_spec:
    | FSTRING_MIDDLE
    | fstring_replacement_field
fstring:
    | FSTRING_START fstring_middle* FSTRING_END

tstring_format_spec_replacement_field:
    | '{' annotated_rhs '='? [fstring_conversion]
[tstring_full_format_spec] '}'
tstring_format_spec:
    | TSTRING_MIDDLE
    | tstring_format_spec_replacement_field
tstring_full_format_spec:
    | ':' tstring_format_spec*
tstring_replacement_field:
    | '{' annotated_rhs '='? [fstring_conversion]
```

```
[tstring_full_format_spec] '}'
tstring_middle:
    | tstring_replacement_field
    | TSTRING_MIDDLE
tstring:
    | TSTRING_START tstring_middle* TSTRING_END

string: STRING
strings:
    | (fstring|string)+
    | tstring+

list:
    | '[' [star_named_expressions] ']'

tuple:
    | '(' [star_named_expression ',' [star_named_expressions]
] ')'

set: '{' star_named_expressions '}'

# Dicts
# -----

dict:
    | '{' [double_starred_kvpairs] '}'

double_starred_kvpairs: ','.double_starred_kvpair+ [',']

double_starred_kvpair:
    | '**' bitwise_or
    | kvpair

kvpair: expression ':' expression

# Comprehensions & Generators
# ---------------------------

for_if_clauses:
    | for_if_clause+

for_if_clause:
    | 'async' 'for' star_targets 'in' ~ disjunction ('if'
disjunction )*
    | 'for' star_targets 'in' ~ disjunction ('if' disjunction
)*

listcomp:
    | '[' named_expression for_if_clauses ']'

setcomp:
    | '{' named_expression for_if_clauses '}'

genexp:
    | '(' ( assignment_expression | expression !':=')
for_if_clauses ')'

dictcomp:
    | '{' kvpair for_if_clauses '}'

# FUNCTION CALL ARGUMENTS
# =======================

arguments:
    | args [','] &')'

args:
    | ','.(starred_expression | ( assignment_expression |
expression !':=') !'=')+ [',' kwargs ]
    | kwargs

kwargs:
    | ','.kwarg_or_starred+ ',' ','.kwarg_or_double_starred+
    | ','.kwarg_or_starred+
    | ','.kwarg_or_double_starred+

starred_expression:
    | '*' expression

kwarg_or_starred:
    | NAME '=' expression
    | starred_expression

kwarg_or_double_starred:
    | NAME '=' expression
    | '**' expression
```

```
# ASSIGNMENT TARGETS
# ==================

# Generic targets
# ---------------

# NOTE: star_targets may contain *bitwise_or, targets may not.
star_targets:
    | star_target !','
    | star_target (',' star_target )* [',']

star_targets_list_seq: ','.star_target+ [',']

star_targets_tuple_seq:
    | star_target (',' star_target )+ [',']
    | star_target ','

star_target:
    | '*' (!'*' star_target)
    | target_with_star_atom

target_with_star_atom:
    | t_primary '.' NAME !t_lookahead
    | t_primary '[' slices ']' !t_lookahead
    | star_atom

star_atom:
    | NAME
    | '(' target_with_star_atom ')'
    | '(' [star_targets_tuple_seq] ')'
    | '[' [star_targets_list_seq] ']'

single_target:
    | single_subscript_attribute_target
    | NAME
    | '(' single_target ')'

single_subscript_attribute_target:
    | t_primary '.' NAME !t_lookahead
    | t_primary '[' slices ']' !t_lookahead

t_primary:
    | t_primary '.' NAME &t_lookahead
    | t_primary '[' slices ']' &t_lookahead
    | t_primary genexp &t_lookahead
    | t_primary '(' [arguments] ')' &t_lookahead
    | atom &t_lookahead

t_lookahead: '(' | '[' | '.'

# Targets for del statements
# --------------------------

del_targets: ','.del_target+ [',']

del_target:
    | t_primary '.' NAME !t_lookahead
    | t_primary '[' slices ']' !t_lookahead
    | del_t_atom

del_t_atom:
    | NAME
    | '(' del_target ')'
    | '(' [del_targets] ')'
    | '[' [del_targets] ']'

# TYPING ELEMENTS
# ---------------

# type_expressions allow */** but ignore them
type_expressions:
    | ','.expression+ ',' '*' expression ',' '**' expression
    | ','.expression+ ',' '*' expression
    | ','.expression+ ',' '**' expression
    | '*' expression ',' '**' expression
    | '*' expression
    | '**' expression
    | ','.expression+

func_type_comment:
    | NEWLINE TYPE_COMMENT &(NEWLINE INDENT)   # Must be
followed by indented block
    | TYPE_COMMENT

# ========================= END OF THE GRAMMAR
```

```
===========================


# ========================= START OF INVALID RULES
=======================
```

The Python Standard Library

While The Python Language Reference describes the exact syntax and semantics of the Python language, this library reference manual describes the standard library that is distributed with Python. It also describes some of the optional components that are commonly included in Python distributions.

Python’s standard library is very extensive, offering a wide range of facilities as indicated by the long table of contents listed below. The library contains built-in modules (written in C) that provide access to system functionality such as file I/O that would otherwise be inaccessible to Python programmers, as well as modules written in Python that provide standardized solutions for many problems that occur in everyday programming. Some of these modules are explicitly designed to encourage and enhance the portability of Python programs by abstracting away platform-specifics into platform-neutral APIs.

The Python installers for the Windows platform usually include the entire standard library and often also include many additional components. For Unix-like operating systems Python is normally provided as a collection of packages, so it may be necessary to use the packaging tools provided with the operating system to obtain some or all of the optional components.

In addition to the standard library, there is an active collection of hundreds of thousands of components (from individual programs and modules to packages and entire application development frameworks), available from the Python Package Index [https://pypi.org].

- Introduction
    - Notes on availability
- Built-in Functions
- Built-in Constants
    - Constants added by the `site` module
- Built-in Types
    - Truth Value Testing
    - Boolean Operations — `and`, `or`, `not`
    - Comparisons
    - Numeric Types — `int`, `float`, `complex`
    - Boolean Type - `bool`
    - Iterator Types
    - Sequence Types — `list`, `tuple`, `range`
    - Text and Binary Sequence Type Methods Summary
    - Text Sequence Type — `str`
    - Binary Sequence Types — `bytes`, `bytearray`, `memoryview`
    - Set Types — `set`, `frozenset`
    - Mapping Types — `dict`
    - Context Manager Types
    - Type Annotation Types — Generic Alias, Union
    - Other Built-in Types
    - Special Attributes
    - Integer string conversion length limitation
- Built-in Exceptions
    - Exception context
    - Inheriting from built-in exceptions
    - Base classes
    - Concrete exceptions
    - Warnings
    - Exception groups
    - Exception hierarchy
- Text Processing Services
    - `string` — Common string operations

    - `string.templatelib` — Support for template string literals
    - `re` — Regular expression operations
    - `difflib` — Helpers for computing deltas
    - `textwrap` — Text wrapping and filling
    - `unicodedata` — Unicode Database
    - `stringprep` — Internet String Preparation
    - `readline` — GNU readline interface
    - `rlcompleter` — Completion function for GNU readline
- Binary Data Services
    - `struct` — Interpret bytes as packed binary data
    - `codecs` — Codec registry and base classes
- Data Types
    - `datetime` — Basic date and time types
    - `zoneinfo` — IANA time zone support
    - `calendar` — General calendar-related functions
    - `collections` — Container datatypes
    - `collections.abc` — Abstract Base Classes for Containers
    - `heapq` — Heap queue algorithm
    - `bisect` — Array bisection algorithm
    - `array` — Efficient arrays of numeric values
    - `weakref` — Weak references
    - `types` — Dynamic type creation and names for built-in types
    - `copy` — Shallow and deep copy operations
    - `pprint` — Data pretty printer
    - `reprlib` — Alternate `repr()` implementation
    - `enum` — Support for enumerations
    - `graphlib` — Functionality to operate with graph-like structures
- Numeric and Mathematical Modules
    - `numbers` — Numeric abstract base classes
    - `math` — Mathematical functions
    - `cmath` — Mathematical functions for complex numbers

    - `decimal` — Decimal fixed-point and floating-point arithmetic
    - `fractions` — Rational numbers
    - `random` — Generate pseudo-random numbers
    - `statistics` — Mathematical statistics functions
- Functional Programming Modules
    - `itertools` — Functions creating iterators for efficient looping
    - `functools` — Higher-order functions and operations on callable objects
    - `operator` — Standard operators as functions
- File and Directory Access
    - `pathlib` — Object-oriented filesystem paths
    - `os.path` — Common pathname manipulations
    - `stat` — Interpreting `stat()` results
    - `filecmp` — File and Directory Comparisons
    - `tempfile` — Generate temporary files and directories
    - `glob` — Unix style pathname pattern expansion
    - `fnmatch` — Unix filename pattern matching
    - `linecache` — Random access to text lines
    - `shutil` — High-level file operations
- Data Persistence
    - `pickle` — Python object serialization
    - `copyreg` — Register `pickle` support functions
    - `shelve` — Python object persistence
    - `marshal` — Internal Python object serialization
    - `dbm` — Interfaces to Unix “databases”
    - `sqlite3` — DB-API 2.0 interface for SQLite databases
- Data Compression and Archiving
    - The `compression` package
    - `compression.zstd` — Compression compatible with the Zstandard format
    - `zlib` — Compression compatible with **gzip**
    - `gzip` — Support for **gzip** files
    - `bz2` — Support for **bzip2** compression
    - `lzma` — Compression using the LZMA algorithm

  - `zipfile` — Work with ZIP archives
  - `tarfile` — Read and write tar archive files
- File Formats
  - `csv` — CSV File Reading and Writing
  - `configparser` — Configuration file parser
  - `tomllib` — Parse TOML files
  - `netrc` — netrc file processing
  - `plistlib` — Generate and parse Apple `.plist` files
- Cryptographic Services
  - `hashlib` — Secure hashes and message digests
  - `hmac` — Keyed-Hashing for Message Authentication
  - `secrets` — Generate secure random numbers for managing secrets
- Generic Operating System Services
  - `os` — Miscellaneous operating system interfaces
  - `io` — Core tools for working with streams
  - `time` — Time access and conversions
  - `logging` — Logging facility for Python
  - `logging.config` — Logging configuration
  - `logging.handlers` — Logging handlers
  - `platform` — Access to underlying platform’s identifying data
  - `errno` — Standard errno system symbols
  - `ctypes` — A foreign function library for Python
- Command-line interface libraries
  - `argparse` — Parser for command-line options, arguments and subcommands
  - `optparse` — Parser for command line options
  - `getpass` — Portable password input
  - `fileinput` — Iterate over lines from multiple input streams
  - `curses` — Terminal handling for character-cell displays
  - `curses.textpad` — Text input widget for curses programs
  - `curses.ascii` — Utilities for ASCII characters

  - `curses.panel` — A panel stack extension for curses
  - `cmd` — Support for line-oriented command interpreters
- Concurrent Execution
  - `threading` — Thread-based parallelism
  - `multiprocessing` — Process-based parallelism
  - `multiprocessing.shared_memory` — Shared memory for direct access across processes
  - The `concurrent` package
  - `concurrent.futures` — Launching parallel tasks
  - `concurrent.interpreters` — Multiple interpreters in the same process
  - `subprocess` — Subprocess management
  - `sched` — Event scheduler
  - `queue` — A synchronized queue class
  - `contextvars` — Context Variables
  - `_thread` — Low-level threading API
- Networking and Interprocess Communication
  - `asyncio` — Asynchronous I/O
  - `socket` — Low-level networking interface
  - `ssl` — TLS/SSL wrapper for socket objects
  - `select` — Waiting for I/O completion
  - `selectors` — High-level I/O multiplexing
  - `signal` — Set handlers for asynchronous events
  - `mmap` — Memory-mapped file support
- Internet Data Handling
  - `email` — An email and MIME handling package
  - `json` — JSON encoder and decoder
  - `mailbox` — Manipulate mailboxes in various formats
  - `mimetypes` — Map filenames to MIME types
  - `base64` — Base16, Base32, Base64, Base85 Data Encodings
  - `binascii` — Convert between binary and ASCII
  - `quopri` — Encode and decode MIME quoted-printable data
- Structured Markup Processing Tools

  - `html` — HyperText Markup Language support
  - `html.parser` — Simple HTML and XHTML parser
  - `html.entities` — Definitions of HTML general entities
  - XML Processing Modules
  - `xml.etree.ElementTree` — The ElementTree XML API
  - `xml.dom` — The Document Object Model API
  - `xml.dom.minidom` — Minimal DOM implementation
  - `xml.dom.pulldom` — Support for building partial DOM trees
  - `xml.sax` — Support for SAX2 parsers
  - `xml.sax.handler` — Base classes for SAX handlers
  - `xml.sax.saxutils` — SAX Utilities
  - `xml.sax.xmlreader` — Interface for XML parsers
  - `xml.parsers.expat` — Fast XML parsing using Expat
- Internet Protocols and Support
  - `webbrowser` — Convenient web-browser controller
  - `wsgiref` — WSGI Utilities and Reference Implementation
  - `urllib` — URL handling modules
  - `urllib.request` — Extensible library for opening URLs
  - `urllib.response` — Response classes used by urllib
  - `urllib.parse` — Parse URLs into components
  - `urllib.error` — Exception classes raised by urllib.request
  - `urllib.robotparser` — Parser for robots.txt
  - `http` — HTTP modules
  - `http.client` — HTTP protocol client
  - `ftplib` — FTP protocol client
  - `poplib` — POP3 protocol client
  - `imaplib` — IMAP4 protocol client
  - `smtplib` — SMTP protocol client
  - `uuid` — UUID objects according to **RFC 9562**
  - `socketserver` — A framework for network servers
  - `http.server` — HTTP servers
  - `http.cookies` — HTTP state management
  - `http.cookiejar` — Cookie handling for HTTP clients

  - `xmlrpc` — XMLRPC server and client modules
  - `xmlrpc.client` — XML-RPC client access
  - `xmlrpc.server` — Basic XML-RPC servers
  - `ipaddress` — IPv4/IPv6 manipulation library
- Multimedia Services
  - `wave` — Read and write WAV files
  - `colorsys` — Conversions between color systems
- Internationalization
  - `gettext` — Multilingual internationalization services
  - `locale` — Internationalization services
- Graphical user interfaces with Tk
  - `tkinter` — Python interface to Tcl/Tk
  - `tkinter.colorchooser` — Color choosing dialog
  - `tkinter.font` — Tkinter font wrapper
  - Tkinter Dialogs
  - `tkinter.messagebox` — Tkinter message prompts
  - `tkinter.scrolledtext` — Scrolled Text Widget
  - `tkinter.dnd` — Drag and drop support
  - `tkinter.ttk` — Tk themed widgets
  - IDLE — Python editor and shell
  - `turtle` — Turtle graphics
- Development Tools
  - `typing` — Support for type hints
  - `pydoc` — Documentation generator and online help system
  - Python Development Mode
  - `doctest` — Test interactive Python examples
  - `unittest` — Unit testing framework
  - `unittest.mock` — mock object library
  - `unittest.mock` — getting started
  - `test` — Regression tests package for Python
  - `test.support` — Utilities for the Python test suite
  - `test.support.socket_helper` — Utilities for socket tests
  - `test.support.script_helper` — Utilities for the Python execution tests

  - `test.support.bytecode_helper` — Support tools for testing correct bytecode generation
  - `test.support.threading_helper` — Utilities for threading tests
  - `test.support.os_helper` — Utilities for os tests
  - `test.support.import_helper` — Utilities for import tests
  - `test.support.warnings_helper` — Utilities for warnings tests
- Debugging and Profiling
  - Audit events table
  - `bdb` — Debugger framework
  - `faulthandler` — Dump the Python traceback
  - `pdb` — The Python Debugger
  - The Python Profilers
  - `timeit` — Measure execution time of small code snippets
  - `trace` — Trace or track Python statement execution
  - `tracemalloc` — Trace memory allocations
- Software Packaging and Distribution
  - `ensurepip` — Bootstrapping the `pip` installer
  - `venv` — Creation of virtual environments
  - `zipapp` — Manage executable Python zip archives
- Python Runtime Services
  - `sys` — System-specific parameters and functions
  - `sys.monitoring` — Execution event monitoring
  - `sysconfig` — Provide access to Python’s configuration information
  - `builtins` — Built-in objects
  - `__main__` — Top-level code environment
  - `warnings` — Warning control
  - `dataclasses` — Data Classes
  - `contextlib` — Utilities for `with`-statement contexts
  - `abc` — Abstract Base Classes
  - `atexit` — Exit handlers
  - `traceback` — Print or retrieve a stack traceback
  - `__future__` — Future statement definitions

  - `gc` — Garbage Collector interface
  - `inspect` — Inspect live objects
  - `annotationlib` — Functionality for introspecting annotations
  - `site` — Site-specific configuration hook
- Custom Python Interpreters
  - `code` — Interpreter base classes
  - `codeop` — Compile Python code
- Importing Modules
  - `zipimport` — Import modules from Zip archives
  - `pkgutil` — Package extension utility
  - `modulefinder` — Find modules used by a script
  - `runpy` — Locating and executing Python modules
  - `importlib` — The implementation of `import`
  - `importlib.resources` – Package resource reading, opening and access
  - `importlib.resources.abc` – Abstract base classes for resources
  - `importlib.metadata` – Accessing package metadata
  - The initialization of the `sys.path` module search path
- Python Language Services
  - `ast` — Abstract syntax trees
  - `symtable` — Access to the compiler’s symbol tables
  - `token` — Constants used with Python parse trees
  - `keyword` — Testing for Python keywords
  - `tokenize` — Tokenizer for Python source
  - `tabnanny` — Detection of ambiguous indentation
  - `pyclbr` — Python module browser support
  - `py_compile` — Compile Python source files
  - `compileall` — Byte-compile Python libraries
  - `dis` — Disassembler for Python bytecode
  - `pickletools` — Tools for pickle developers
- MS Windows Specific Services
  - `msvcrt` — Useful routines from the MS VC++ runtime
  - `winreg` — Windows registry access
  - `winsound` — Sound-playing interface for Windows

- Unix-specific services
  - `shlex` — Simple lexical analysis
  - `posix` — The most common POSIX system calls
  - `pwd` — The password database
  - `grp` — The group database
  - `termios` — POSIX style tty control
  - `tty` — Terminal control functions
  - `pty` — Pseudo-terminal utilities
  - `fcntl` — The `fcntl` and `ioctl` system calls
  - `resource` — Resource usage information
  - `syslog` — Unix syslog library routines
- Modules command-line interface (CLI)
- Superseded Modules
  - `getopt` — C-style parser for command line options
- Removed Modules
- Security Considerations

Introduction

The “Python library” contains several different kinds of components.

It contains data types that would normally be considered part of the “core” of a language, such as numbers and lists. For these types, the Python language core defines the form of literals and places some constraints on their semantics, but does not fully define the semantics. (On the other hand, the language core does define syntactic properties like the spelling and priorities of operators.)

The library also contains built-in functions and exceptions — objects that can be used by all Python code without the need of an `import` statement. Some of these are defined by the core language, but many are not essential for the core semantics and are only described here.

The bulk of the library, however, consists of a collection of modules. There are many ways to dissect this collection. Some modules are written in C and built in to the Python interpreter; others are written in Python and imported in source form. Some modules provide interfaces that are highly specific to Python, like printing a stack trace; some provide interfaces that are specific to particular operating systems, such as access to specific hardware; others provide interfaces that are specific to a particular application domain, like the World Wide Web. Some modules are available in all versions and ports of Python; others are only available when the underlying system supports or requires them; yet others are available only when a particular configuration option was chosen at the time when Python was compiled and installed.

This manual is organized “from the inside out:” it first describes the built-in functions, data types and exceptions, and finally the modules, grouped in chapters of related modules.

This means that if you start reading this manual from the start, and skip to the next chapter when you get bored, you will get a reasonable overview of the available modules and application areas that are supported by the Python library. Of course, you don’t *have* to read it like a novel — you can also browse the table of contents (in front of the manual), or look for a specific function, module or term in the index (in the back). And finally, if you enjoy learning about random subjects, you choose a random page number (see module `random`) and read a section or two. Regardless of the order in which you read the sections of this manual, it helps to start with chapter Built-in Functions, as the remainder of the manual assumes familiarity with this material.

Let the show begin!

Notes on availability

- An “Availability: Unix” note means that this function is commonly found on Unix systems. It does not make any claims about its existence on a specific operating system.
- If not separately noted, all functions that claim “Availability: Unix” are supported on macOS, iOS and Android, all of which build on a Unix core.
- If an availability note contains both a minimum Kernel version and a minimum libc version, then both conditions must hold. For example a feature with note *Availability: Linux >= 3.17 with glibc >= 2.27* requires both Linux 3.17 or newer and glibc 2.27 or newer.

WebAssembly platforms

The WebAssembly [https://webassembly.org/] platforms `wasm32-emscripten` (Emscripten [https://emscripten.org/]) and `wasm32-wasi` (WASI [https://wasi.dev/]) provide a subset of POSIX APIs. WebAssembly runtimes and browsers are sandboxed and have limited access to the host and external resources. Any Python standard library module that uses processes, threading, networking, signals, or other forms of inter-process communication (IPC), is either not available or may not work as on other Unix-like systems. File I/O, file system, and Unix permission-related functions are restricted, too. Emscripten does not permit blocking I/O. Other blocking operations like **`sleep()`** block the browser event loop.

The properties and behavior of Python on WebAssembly platforms depend on the Emscripten [https://emscripten.org/]-SDK or WASI [https://wasi.dev/]-SDK version, WASM runtimes (browser, NodeJS, wasmtime [https://wasmtime.dev/]), and Python build time flags. WebAssembly, Emscripten, and WASI are evolving standards; some features like networking may be supported in the future.

For Python in the browser, users should consider Pyodide [https://pyodide.org/] or PyScript [https://pyscript.net/]. PyScript is built on top of Pyodide, which itself is built on top of CPython and Emscripten. Pyodide provides access to browsers' JavaScript and DOM APIs as well as limited networking capabilities with JavaScript's `XMLHttpRequest` and `Fetch` APIs.

- Process-related APIs are not available or always fail with an error. That includes APIs that spawn new processes (**`fork()`**, **`execve()`**), wait for processes (**`waitpid()`**), send signals (**`kill()`**), or otherwise interact with processes. The **`subprocess`** is importable but does not work.

- The `socket` module is available, but is limited and behaves differently from other platforms. On Emscripten, sockets are always non-blocking and require additional JavaScript code and helpers on the server to proxy TCP through WebSockets; see Emscripten Networking [https://emscripten.org/docs/porting/networking.html] for more information. WASI snapshot preview 1 only permits sockets from an existing file descriptor.
- Some functions are stubs that either don't do anything and always return hardcoded values.
- Functions related to file descriptors, file permissions, file ownership, and links are limited and don't support some operations. For example, WASI does not permit symlinks with absolute file names.

Mobile platforms

Android and iOS are, in most respects, POSIX operating systems. File I/O, socket handling, and threading all behave as they would on any POSIX operating system. However, there are several major differences:

- Mobile platforms can only use Python in "embedded" mode. There is no Python REPL, and no ability to use separate executables such as **python** or **pip**. To add Python code to your mobile app, you must use the Python embedding API. For more details, see Using Python on Android and Using Python on iOS.

- Subprocesses:

    - On Android, creating subprocesses is possible but officially unsupported [https://issuetracker.google.com/issues/128554619#comment4]. In

  particular, Android does not support any part of the System V IPC API, so `multiprocessing` is not available.
  - An iOS app cannot use any form of subprocessing, multiprocessing, or inter-process communication. If an iOS app attempts to create a subprocess, the process creating the subprocess will either lock up, or crash. An iOS app has no visibility of other applications that are running, nor any ability to communicate with other running applications, outside of the iOS-specific APIs that exist for this purpose.

- Mobile apps have limited access to modify system resources (such as the system clock). These resources will often be *readable*, but attempts to modify those resources will usually fail.

- Console input and output:

  - On Android, the native `stdout` and `stderr` are not connected to anything, so Python installs its own streams which redirect messages to the system log. These can be seen under the tags `python.stdout` and `python.stderr` respectively.

  - iOS apps have a limited concept of console output. `stdout` and `stderr` *exist*, and content written to `stdout` and `stderr` will be visible in logs when running in Xcode, but this content *won't* be recorded in the system log. If a user who has installed your app provides their app logs as a diagnostic aid, they will not include any detail written to `stdout` or `stderr`.

  - Mobile apps have no usable `stdin` at all. While apps can display an on-screen keyboard, this is a software feature, not something that is attached to `stdin`.

As a result, Python modules that involve console manipulation (such as **`curses`** and **`readline`**) are not available on mobile platforms.

Built-in Functions

The Python interpreter has a number of functions and types built into it that are always available. They are listed here in alphabetical order.

Built-in Functions

A
`abs()`
`aiter()`
`all()`
`anext()`
`any()`
`ascii()`

B
`bin()`
`bool()`
`breakpoint()`
`bytearray()`
`bytes()`

C
`callable()`
`chr()`
`classmethod()`
`compile()`
`complex()`

D
`delattr()`
`dict()`

E
`enumerate()`
`eval()`
`exec()`

F
`filter()`
`float()`
`format()`
`frozenset()`

G
`getattr()`
`globals()`

H
`hasattr()`
`hash()`
`help()`
`hex()`

I
`id()`
`input()`
`int()`

L
`len()`
`list()`
`locals()`

M
`map()`
`max()`
`memoryview()`
`min()`

N
`next()`

O
`object()`
`oct()`
`open()`
`ord()`

P
`pow()`
`print()`
`property()`

R
`range()`
`repr()`
`reversed()`
`round()`

S
`set()`
`setattr()`
`slice()`
`sorted()`
`staticmethod()`
`str()`
`sum()`
`super()`

T
`tuple()`
`type()`

V
`vars()`

Z
`zip()`

Built-in Functions

`dir()`	`isinstance()`	
`divmod()`	`issubclass()`	_
	`iter()`	`__import__()`

abs(*number*, /)

Return the absolute value of a number. The argument may be an integer, a floating-point number, or an object implementing `__abs__()`. If the argument is a complex number, its magnitude is returned.

aiter(*async_iterable*, /)

Return an asynchronous iterator for an asynchronous iterable. Equivalent to calling `x.__aiter__()`.

Note: Unlike `iter()`, `aiter()` has no 2-argument variant.

Added in version 3.10.

all(*iterable*, /)

Return `True` if all elements of the *iterable* are true (or if the iterable is empty). Equivalent to:

```
def all(iterable):
    for element in iterable:
        if not element:
            return False
    return True
```

awaitable anext(*async_iterator*, /)
awaitable anext(*async_iterator*, *default*, /)

When awaited, return the next item from the given asynchronous iterator, or *default* if given and the iterator is exhausted.

This is the async variant of the **`next()`** builtin, and behaves similarly.

This calls the **`__anext__()`** method of *async_iterator*, returning an awaitable. Awaiting this returns the next value of the iterator. If *default* is given, it is returned if the iterator is exhausted, otherwise **`StopAsyncIteration`** is raised.

Added in version 3.10.

any(*iterable*, /)

Return `True` if any element of the *iterable* is true. If the iterable is empty, return `False`. Equivalent to:

```
def any(iterable):
    for element in iterable:
        if element:
            return True
    return False
```

ascii(*object*, /)

As **`repr()`**, return a string containing a printable representation of an object, but escape the non-ASCII characters in the string returned by **`repr()`** using `\x`, `\u`, or `\U` escapes. This generates a string similar to that returned by **`repr()`** in Python 2.

bin(*integer*, /)

Convert an integer number to a binary string prefixed with “0b”. The result is a valid Python expression. If *integer* is not a Python **`int`** object, it has to define an **`__index__()`** method that returns an integer. Some examples:

```
>>> bin(3)
'0b11'
>>> bin(-10)
'-0b1010'
```

If the prefix “0b” is desired or not, you can use either of the following ways.

```
>>> format(14, '#b'), format(14, 'b')
('0b1110', '1110')
>>> f'{14:#b}', f'{14:b}'
('0b1110', '1110')
```

See also **`enum.bin()`** to represent negative values as twos-complement.

See also **`format()`** for more information.

class bool(*object=False*, /)

Return a Boolean value, i.e. one of `True` or `False`. The argument is converted using the standard truth testing procedure. If the argument is false or omitted, this returns `False`; otherwise, it returns `True`. The **`bool`** class is a subclass of **`int`** (see Numeric Types — int, float, complex). It cannot be subclassed further. Its only instances are `False` and `True` (see Boolean Type - bool).

Changed in version 3.7: The parameter is now positional-only.

breakpoint(**args*, ***kws*)

This function drops you into the debugger at the call site. Specifically, it calls **`sys.breakpointhook()`**, passing `args` and `kws` straight through. By default, `sys.breakpointhook()` calls **`pdb.set_trace()`** expecting no arguments. In this case, it is purely a convenience function so you don’t have to explicitly import **`pdb`** or type as much code to enter the debugger. However, **`sys.breakpointhook()`** can be set to some other function and **`breakpoint()`** will automatically call that, allowing you to drop into the debugger of choice. If **`sys.breakpointhook()`** is not accessible, this function will raise **`RuntimeError`**.

By default, the behavior of **`breakpoint()`** can be changed with the **`PYTHONBREAKPOINT`** environment variable. See **`sys.breakpointhook()`** for usage details.

Note that this is not guaranteed if **`sys.breakpointhook()`** has been replaced.

Raises an auditing event `builtins.breakpoint` with argument `breakpointhook`.

Added in version 3.7.

class bytearray(*source=b''*)
class bytearray(*source*, *encoding*, *errors='strict'*)

Return a new array of bytes. The **`bytearray`** class is a mutable sequence of integers in the range 0 <= x < 256. It has most of the usual methods of mutable sequences, described in Mutable Sequence Types, as well as most methods that the **`bytes`** type has, see Bytes and Bytearray Operations.

The optional *source* parameter can be used to initialize the array in a few different ways:

- If it is a *string*, you must also give the *encoding* (and optionally, *errors*) parameters; **`bytearray()`** then converts the string to bytes using **`str.encode()`**.
- If it is an *integer*, the array will have that size and will be initialized with null bytes.
- If it is an object conforming to the buffer interface, a read-only buffer of the object will be used to initialize the bytes array.
- If it is an *iterable*, it must be an iterable of integers in the range `0 <= x < 256`, which are used as the initial contents of the array.

Without an argument, an array of size 0 is created.

See also Binary Sequence Types — bytes, bytearray, memoryview and Bytearray Objects.

class bytes(*source=b''*)
class bytes(*source*, *encoding*, *errors='strict'*)

Return a new “bytes” object which is an immutable sequence of integers in the range `0 <= x < 256`. **`bytes`** is an immutable version of **`bytearray`** – it has the same non-mutating methods and the same indexing and slicing behavior.

Accordingly, constructor arguments are interpreted as for **`bytearray()`**.

Bytes objects can also be created with literals, see String and Bytes literals.

See also Binary Sequence Types — bytes, bytearray, memoryview, Bytes Objects, and Bytes and Bytearray Operations.

callable(*object*, */*)

Return **`True`** if the *object* argument appears callable, **`False`** if not. If this returns `True`, it is still possible that a call fails, but if it is `False`, calling *object* will never succeed. Note that classes are callable (calling a class returns a new instance); instances are callable if their class has a **`__call__()`** method.

Added in version 3.2: This function was first removed in Python 3.0 and then brought back in Python 3.2.

chr(*codepoint*, */*)

Return the string representing a character with the specified Unicode code point. For example, `chr(97)` returns the string `'a'`, while `chr(8364)` returns the string `'€'`. This is the inverse of **`ord()`**.

The valid range for the argument is from 0 through 1,114,111 (0x10FFFF in base 16). **`ValueError`** will be raised if it is outside that range.

@classmethod

Transform a method into a class method.

A class method receives the class as an implicit first argument, just like an instance method receives the instance. To declare a class method, use this idiom:

```
class C:
    @classmethod
    def f(cls, arg1, arg2): ...
```

The `@classmethod` form is a function decorator – see Function definitions for details.

A class method can be called either on the class (such as `C.f()`) or on an instance (such as `C().f()`). The instance is ignored except for its class. If a class method is called for a derived class, the derived class object is passed as the implied first argument.

Class methods are different than C++ or Java static methods. If you want those, see **`staticmethod()`** in this section. For more information on class methods, see The standard type hierarchy.

Changed in version 3.9: Class methods can now wrap other descriptors such as **`property()`**.

Changed in version 3.10: Class methods now inherit the method attributes (**`__module__`**, **`__name__`**, **`__qualname__`**, **`__doc__`** and **`__annotations__`**) and have a new `__wrapped__` attribute.

Deprecated since version 3.11, removed in version 3.13: Class methods can no longer wrap other descriptors such as `property()`.

compile(*source*, *filename*, *mode*, *flags=0*, *dont_inherit=False*, *optimize=-1*)

Compile the *source* into a code or AST object. Code objects can be executed by `exec()` or `eval()`. *source* can either be a normal string, a byte string, or an AST object. Refer to the `ast` module documentation for information on how to work with AST objects.

The *filename* argument should give the file from which the code was read; pass some recognizable value if it wasn’t read from a file (`'<string>'` is commonly used).

The *mode* argument specifies what kind of code must be compiled; it can be `'exec'` if *source* consists of a sequence of statements, `'eval'` if it consists of a single expression, or `'single'` if it consists of a single interactive statement (in the latter case, expression statements that evaluate to something other than `None` will be printed).

The optional arguments *flags* and *dont_inherit* control which compiler options should be activated and which future features should be allowed. If neither is present (or both are zero) the code is compiled with the same flags that affect the code that is calling `compile()`. If the *flags* argument is given and *dont_inherit* is not (or is zero) then the compiler options and the future statements specified by the *flags* argument are used in addition to those that would be used anyway. If *dont_inherit* is a non-zero integer then the *flags* argument is it – the flags (future features and compiler options) in the surrounding code are ignored.

Compiler options and future statements are specified by bits which can be bitwise ORed together to specify multiple options. The bitfield required to specify a given future feature can be found as the `compiler_flag` attribute on the `_Feature` instance in the `__future__` module. Compiler flags can be found in `ast` module, with `PyCF_` prefix.

The argument *optimize* specifies the optimization level of the compiler; the default value of `-1` selects the optimization level of the interpreter as given by `-O` options. Explicit levels are `0` (no optimization; `__debug__` is true), `1` (asserts are removed, `__debug__` is false) or `2` (docstrings are removed too).

This function raises `SyntaxError` or `ValueError` if the compiled source is invalid.

If you want to parse Python code into its AST representation, see `ast.parse()`.

Raises an auditing event `compile` with arguments `source` and `filename`. This event may also be raised by implicit compilation.

Note

When compiling a string with multi-line code in `'single'` or `'eval'` mode, input must be terminated by at least one newline character. This is to facilitate detection of incomplete and complete statements in the `code` module.

Warning

It is possible to crash the Python interpreter with a sufficiently large/complex string when compiling to an AST object due to stack depth limitations in Python’s AST compiler.

Changed in version 3.2: Allowed use of Windows and Mac newlines. Also, input in `'exec'` mode does not have to end in a newline anymore. Added the *optimize* parameter.

Changed in version 3.5: Previously, **`TypeError`** was raised when null bytes were encountered in *source*.

Added in version 3.8: `ast.PyCF_ALLOW_TOP_LEVEL_AWAIT` can now be passed in flags to enable support for top-level `await`, `async for`, and `async with`.

class complex(*number=0*, */*)
class complex(*string*, */*)
class complex(*real=0*, *imag=0*)

Convert a single string or number to a complex number, or create a complex number from real and imaginary parts.

Examples:

```
>>> complex('+1.23')
(1.23+0j)
>>> complex('-4.5j')
-4.5j
>>> complex('-1.23+4.5j')
(-1.23+4.5j)
>>> complex('\t( -1.23+4.5J )\n')
(-1.23+4.5j)
>>> complex('-Infinity+NaNj')
(-inf+nanj)
>>> complex(1.23)
(1.23+0j)
>>> complex(imag=-4.5)
-4.5j
>>> complex(-1.23, 4.5)
(-1.23+4.5j)
```

If the argument is a string, it must contain either a real part (in the same format as for **`float()`**) or an imaginary part (in the same format but with a `'j'` or `'J'` suffix), or both real and imaginary parts (the sign of the imaginary part is mandatory in this case). The string can optionally be surrounded by whitespaces and the round parentheses `'('` and `')'`, which are ignored. The string must not contain whitespace between `'+'`, `'-'`, the `'j'` or `'J'` suffix, and the decimal number. For example, `complex('1+2j')` is fine, but `complex('1 + 2j')` raises **`ValueError`**. More precisely, the input must conform to the **`complexvalue`** production rule in the following grammar, after parentheses and leading and trailing whitespace characters are removed:

```
complexvalue: floatvalue |
              floatvalue ("j" | "J") |
              floatvalue sign absfloatvalue ("j" | "J")
```

If the argument is a number, the constructor serves as a numeric conversion like **`int`** and **`float`**. For a general Python object `x`, `complex(x)` delegates to `x.__complex__()`. If **`__complex__()`** is not defined then it falls back to **`__float__()`**. If **`__float__()`** is not defined then it falls back to **`__index__()`**.

If two arguments are provided or keyword arguments are used, each argument may be any numeric type (including complex). If both arguments are real numbers, return a complex number with the real component *real* and the imaginary component *imag*. If both arguments are complex numbers, return a complex number with the real component `real.real-imag.imag` and the imaginary component `real.imag+imag.real`. If one of arguments is a real number, only its real component is used in the above expressions.

See also **`complex.from_number()`** which only accepts a single numeric argument.

If all arguments are omitted, returns `0j`.

The complex type is described in Numeric Types — int, float, complex.

Changed in version 3.6: Grouping digits with underscores as in code literals is allowed.

Changed in version 3.8: Falls back to `__index__()` if `__complex__()` and `__float__()` are not defined.

Deprecated since version 3.14: Passing a complex number as the *real* or *imag* argument is now deprecated; it should only be passed as a single positional argument.

delattr(*object*, *name*, */*)

This is a relative of `setattr()`. The arguments are an object and a string. The string must be the name of one of the object's attributes. The function deletes the named attribute, provided the object allows it. For example, `delattr(x, 'foobar')` is equivalent to `del x.foobar`. *name* need not be a Python identifier (see `setattr()`).

class dict(***kwargs*)
class dict(*mapping*, */*, ***kwargs*)
class dict(*iterable*, */*, ***kwargs*)

Create a new dictionary. The `dict` object is the dictionary class. See `dict` and Mapping Types — dict for documentation about this class.

For other containers see the built-in `list`, `set`, and `tuple` classes, as well as the `collections` module.

dir()
dir(*object*, */*)

Without arguments, return the list of names in the current local scope. With an argument, attempt to return a list of valid attributes for that object.

If the object has a method named `__dir__()`, this method will be called and must return the list of attributes. This allows objects that implement a custom `__getattr__()` or `__getattribute__()` function to customize the way `dir()` reports their attributes.

If the object does not provide `__dir__()`, the function tries its best to gather information from the object's `__dict__` attribute, if defined, and from its type object. The resulting list is not necessarily complete and may be inaccurate when the object has a custom `__getattr__()`.

The default `dir()` mechanism behaves differently with different types of objects, as it attempts to produce the most relevant, rather than complete, information:

- If the object is a module object, the list contains the names of the module's attributes.
- If the object is a type or class object, the list contains the names of its attributes, and recursively of the attributes of its bases.
- Otherwise, the list contains the object's attributes' names, the names of its class's attributes, and recursively of the attributes of its class's base classes.

The resulting list is sorted alphabetically. For example:

```
>>> import struct
>>> dir()   # show the names in the module namespace
['__builtins__', '__name__', 'struct']
>>> dir(struct)   # show the names in the struct module
['Struct', '__all__', '__builtins__', '__cached__',
'__doc__', '__file__',
 '__initializing__', '__loader__', '__name__', '__package__',
 '_clearcache', 'calcsize', 'error', 'pack', 'pack_into',
 'unpack', 'unpack_from']
>>> class Shape:
...     def __dir__(self):
...         return ['area', 'perimeter', 'location']
...
>>> s = Shape()
>>> dir(s)
['area', 'location', 'perimeter']
```

Note

Because `dir()` is supplied primarily as a convenience for use at an interactive prompt, it tries to supply an interesting set of names more than it tries to supply a rigorously or consistently defined set of names, and its detailed behavior may change across releases. For example, metaclass attributes are not in the result list when the argument is a class.

divmod(*a*, *b*, */*)

Take two (non-complex) numbers as arguments and return a pair of numbers consisting of their quotient and remainder when using integer division. With mixed operand types, the rules for binary arithmetic operators apply. For integers, the result is the same as `(a // b, a % b)`. For floating-point numbers the result is `(q, a % b)`, where *q* is usually `math.floor(a / b)` but may be 1 less than that. In any case `q * b + a % b` is very close to *a*, if `a % b` is non-zero it has the same sign as *b*, and `0 <= abs(a % b) < abs(b)`.

enumerate(*iterable*, *start=0*)

Return an enumerate object. *iterable* must be a sequence, an iterator, or some other object which supports iteration. The `__next__()` method of the iterator returned by `enumerate()` returns a tuple containing a count (from *start* which defaults to 0) and the values obtained from iterating over *iterable*.

```
>>> seasons = ['Spring', 'Summer', 'Fall', 'Winter']
>>> list(enumerate(seasons))
[(0, 'Spring'), (1, 'Summer'), (2, 'Fall'), (3, 'Winter')]
```

```
>>> list(enumerate(seasons, start=1))
[(1, 'Spring'), (2, 'Summer'), (3, 'Fall'), (4, 'Winter')]
```

Equivalent to:

```
def enumerate(iterable, start=0):
    n = start
    for elem in iterable:
        yield n, elem
        n += 1
```

eval(*source*, */*, *globals=None*, *locals=None*)

Parameters::

- **source** (`str` | code object) – A Python expression.
- **globals** (`dict` | `None`) – The global namespace (default: `None`).
- **locals** (mapping | `None`) – The local namespace (default: `None`).

Returns:: The result of the evaluated expression.

Raises:: Syntax errors are reported as exceptions.

Warning

This function executes arbitrary code. Calling it with user-supplied input may lead to security vulnerabilities.

The *source* argument is parsed and evaluated as a Python expression (technically speaking, a condition list) using the *globals* and *locals* mappings as global and local namespace. If the *globals* dictionary is present and does not contain a value for the key `__builtins__`, a reference to the dictionary of the built-in module `builtins` is inserted under that key before *source* is parsed. That way you can control what builtins are available to the executed code by inserting your own `__builtins__` dictionary into *globals* before passing it to `eval()`. If the *locals* mapping is omitted it defaults to the *globals* dictionary. If both mappings are omitted, the source is executed with the *globals* and *locals* in the environment where `eval()` is called. Note, *eval()* will only have access to the nested scopes (non-locals) in the enclosing environment if they are already referenced in the scope that is calling `eval()` (e.g. via a `nonlocal` statement).

Example:

```
>>> x = 1
>>> eval('x+1')
2
```

This function can also be used to execute arbitrary code objects (such as those created by `compile()`). In this case, pass a code object instead of a string. If the code object has been compiled with `'exec'` as the *mode* argument, `eval()`'s return value will be `None`.

Hints: dynamic execution of statements is supported by the `exec()` function. The `globals()` and `locals()` functions return the current global and local dictionary, respectively, which may be useful to pass around for use by `eval()` or `exec()`.

If the given source is a string, then leading and trailing spaces and tabs are stripped.

See `ast.literal_eval()` for a function that can safely evaluate strings with expressions containing only literals.

Raises an auditing event `exec` with the code object as the argument. Code compilation events may also be raised.

Changed in version 3.13: The *globals* and *locals* arguments can now be passed as keywords.

Changed in version 3.13: The semantics of the default *locals* namespace have been adjusted as described for the `locals()` builtin.

exec(*source*, /, *globals=None*, *locals=None*, *, *closure=None*)

Warning

This function executes arbitrary code. Calling it with user-supplied input may lead to security vulnerabilities.

This function supports dynamic execution of Python code. *source* must be either a string or a code object. If it is a string, the string is parsed as a suite of Python statements which is then executed (unless a syntax error occurs). [[1]] If it is a code object, it is simply executed. In all cases, the code that's executed is expected to be valid as file input (see the section File input in the Reference Manual). Be aware that the `nonlocal`, `yield`, and `return` statements may not be used outside of function definitions even within the context of code passed to the `exec()` function. The return value is `None`.

In all cases, if the optional parts are omitted, the code is executed in the current scope. If only *globals* is provided, it must be a dictionary (and not a subclass of dictionary), which will be used for both the global and the local variables. If *globals* and *locals* are given, they are used for the global and local variables, respectively. If provided, *locals* can be any mapping object. Remember that at the module level, globals and locals are the same dictionary.

Note

When `exec` gets two separate objects as *globals* and *locals*, the code will be executed as if it were embedded in a class definition. This means functions and classes defined in the executed code will not be able to access variables assigned at the top level (as the “top level” variables are treated as class variables in a class definition).

If the *globals* dictionary does not contain a value for the key `__builtins__`, a reference to the dictionary of the built-in module **`builtins`** is inserted under that key. That way you can control what builtins are available to the executed code by inserting your own `__builtins__` dictionary into *globals* before passing it to **`exec()`**.

The *closure* argument specifies a closure–a tuple of cellvars. It’s only valid when the *object* is a code object containing free (closure) variables. The length of the tuple must exactly match the length of the code object’s **`co_freevars`** attribute.

Raises an auditing event `exec` with the code object as the argument. Code compilation events may also be raised.

Note

The built-in functions **`globals()`** and **`locals()`** return the current global and local namespace, respectively, which may be useful to pass around for use as the second and third argument to **`exec()`**.

Note

The default *locals* act as described for function **`locals()`** below. Pass an explicit *locals* dictionary if you need to see effects of the code on *locals* after function **`exec()`** returns.

Changed in version 3.11: Added the *closure* parameter.

Changed in version 3.13: The *globals* and *locals* arguments can now be passed as keywords.

Changed in version 3.13: The semantics of the default *locals* namespace have been adjusted as described for the **`locals()`** builtin.

filter(*function*, *iterable*, */*)

Construct an iterator from those elements of *iterable* for which *function* is true. *iterable* may be either a sequence, a container which supports iteration, or an iterator. If *function* is `None`, the identity function is assumed, that is, all elements of *iterable* that are false are removed.

Note that `filter(function, iterable)` is equivalent to the generator expression `(item for item in iterable if function(item))` if function is not `None` and `(item for item in iterable if item)` if function is `None`.

See **`itertools.filterfalse()`** for the complementary function that returns elements of *iterable* for which *function* is false.

class float(*number=0.0*, */*)
class float(*string*, */*)

Return a floating-point number constructed from a number or a string.

Examples:

```
>>> float('+1.23')
1.23
>>> float('   -12345\n')
-12345.0
>>> float('1e-003')
0.001
>>> float('+1E6')
1000000.0
```

```
>>> float('-Infinity')
-inf
```

If the argument is a string, it should contain a decimal number, optionally preceded by a sign, and optionally embedded in whitespace. The optional sign may be `'+'` or `'-'`; a `'+'` sign has no effect on the value produced. The argument may also be a string representing a NaN (not-a-number), or positive or negative infinity. More precisely, the input must conform to the `floatvalue` production rule in the following grammar, after leading and trailing whitespace characters are removed:

```
sign:          "+" | "-"
infinity:      "Infinity" | "inf"
nan:           "nan"
digit:         <a Unicode decimal digit, i.e. characters in
Unicode general category Nd>
digitpart:     digit (["_"] digit)*
number:        [digitpart] "." digitpart | digitpart ["."]
exponent:      ("e" | "E") [sign] digitpart
floatnumber:   number [exponent]
absfloatvalue: floatnumber | infinity | nan
floatvalue:    [sign] absfloatvalue
```

Case is not significant, so, for example, “inf”, “Inf”, “INFINITY”, and “iNfINity” are all acceptable spellings for positive infinity.

Otherwise, if the argument is an integer or a floating-point number, a floating-point number with the same value (within Python’s floating-point precision) is returned. If the argument is outside the range of a Python float, an `OverflowError` will be raised.

For a general Python object `x`, `float(x)` delegates to `x.__float__()`. If `__float__()` is not defined then it falls back to `__index__()`.

See also **`float.from_number()`** which only accepts a numeric argument.

If no argument is given, `0.0` is returned.

The float type is described in Numeric Types — int, float, complex.

Changed in version 3.6: Grouping digits with underscores as in code literals is allowed.

Changed in version 3.7: The parameter is now positional-only.

Changed in version 3.8: Falls back to **`__index__()`** if **`__float__()`** is not defined.

format(*value*, *format_spec*='', */*)

Convert a *value* to a “formatted” representation, as controlled by *format_spec*. The interpretation of *format_spec* will depend on the type of the *value* argument; however, there is a standard formatting syntax that is used by most built-in types: Format Specification Mini-Language.

The default *format_spec* is an empty string which usually gives the same effect as calling **`str(value)`**.

A call to `format(value, format_spec)` is translated to `type(value).__format__(value, format_spec)` which bypasses the instance dictionary when searching for the value’s **`__format__()`** method. A **`TypeError`** exception is raised if the method search reaches **`object`** and the *format_spec* is non-empty, or if either the *format_spec* or the return value are not strings.

Changed in version 3.4: `object().__format__(format_spec)` raises **`TypeError`** if *format_spec* is not an empty string.

class frozenset(*iterable=()*, */*)

Return a new **`frozenset`** object, optionally with elements taken from *iterable*. `frozenset` is a built-in class. See **`frozenset`** and Set Types — set, frozenset for documentation about this class.

For other containers see the built-in **`set`**, **`list`**, **`tuple`**, and **`dict`** classes, as well as the **`collections`** module.

getattr(*object*, *name*, */*)
getattr(*object*, *name*, *default*, */*)

Return the value of the named attribute of *object*. *name* must be a string. If the string is the name of one of the object's attributes, the result is the value of that attribute. For example, `getattr(x, 'foobar')` is equivalent to `x.foobar`. If the named attribute does not exist, *default* is returned if provided, otherwise **`AttributeError`** is raised. *name* need not be a Python identifier (see **`setattr()`**).

Note

Since private name mangling happens at compilation time, one must manually mangle a private attribute's (attributes with two leading underscores) name in order to retrieve it with **`getattr()`**.

globals()

Return the dictionary implementing the current module namespace. For code within functions, this is set when the function is defined and remains the same regardless of where the function is called.

hasattr(*object*, *name*, */*)

The arguments are an object and a string. The result is `True` if the string is the name of one of the object's attributes, `False` if not. (This is implemented by calling `getattr(object, name)` and seeing whether it raises an **`AttributeError`** or not.)

hash(*object*, */*)

Return the hash value of the object (if it has one). Hash values are integers. They are used to quickly compare dictionary keys during a dictionary lookup. Numeric values that compare equal have the same hash value (even if they are of different types, as is the case for 1 and 1.0).

Note

For objects with custom **`__hash__()`** methods, note that **`hash()`** truncates the return value based on the bit width of the host machine.

help()
help(*request*)

Invoke the built-in help system. (This function is intended for interactive use.) If no argument is given, the interactive help system starts on the interpreter console. If the argument is a string, then the string is looked up as the name of a module, function, class, method, keyword, or documentation topic, and a help page is printed on the console. If the argument is any other kind of object, a help page on the object is generated.

Note that if a slash(/) appears in the parameter list of a function when invoking **`help()`**, it means that the parameters prior to the slash are positional-only. For more info, see the FAQ entry on positional-only parameters.

This function is added to the built-in namespace by the **`site`** module.

Changed in version 3.4: Changes to **`pydoc`** and **`inspect`** mean that the reported signatures for callables are now more comprehensive and consistent.

hex(*integer*, /)

Convert an integer number to a lowercase hexadecimal string prefixed with “0x”. If *integer* is not a Python **`int`** object, it has to define an **`__index__()`** method that returns an integer. Some examples:

```
>>> hex(255)
'0xff'
>>> hex(-42)
'-0x2a'
```

If you want to convert an integer number to an uppercase or lower hexadecimal string with prefix or not, you can use either of the following ways:

```
>>> '%#x' % 255, '%x' % 255, '%X' % 255
('0xff', 'ff', 'FF')
>>> format(255, '#x'), format(255, 'x'), format(255, 'X')
('0xff', 'ff', 'FF')
>>> f'{255:#x}', f'{255:x}', f'{255:X}'
('0xff', 'ff', 'FF')
```

See also **`format()`** for more information.

See also **`int()`** for converting a hexadecimal string to an integer using a base of 16.

Note

To obtain a hexadecimal string representation for a float, use the **`float.hex()`** method.

id(*object*, /)

Return the “identity” of an object. This is an integer which is guaranteed to be unique and constant for this object during its lifetime. Two objects with non-overlapping lifetimes may have the same `id()` value.

CPython implementation detail: This is the address of the object in memory.

Raises an auditing event `builtins.id` with argument `id`.

input()
input(*prompt*, /)

If the *prompt* argument is present, it is written to standard output without a trailing newline. The function then reads a line from input, converts it to a string (stripping a trailing newline), and returns that. When EOF is read, `EOFError` is raised. Example:

```
>>> s = input('--> ')
--> Monty Python's Flying Circus
>>> s
"Monty Python's Flying Circus"
```

If the `readline` module was loaded, then `input()` will use it to provide elaborate line editing and history features.

Raises an auditing event `builtins.input` with argument `prompt` before reading input

Raises an auditing event `builtins.input/result` with the result after successfully reading input.

class int(*number=0*, /)
class int(*string*, /, *base=10*)

Return an integer object constructed from a number or a string, or return `0` if no arguments are given.

Examples:

```
>>> int(123.45)
123
>>> int('123')
123
>>> int('   -12_345\n')
-12345
>>> int('FACE', 16)
64206
>>> int('0xface', 0)
64206
>>> int('01110011', base=2)
115
```

If the argument defines **`__int__()`**, `int(x)` returns `x.__int__()`. If the argument defines **`__index__()`**, it returns `x.__index__()`. For floating-point numbers, this truncates towards zero.

If the argument is not a number or if *base* is given, then it must be a string, **`bytes`**, or **`bytearray`** instance representing an integer in radix *base*. Optionally, the string can be preceded by `+` or `-` (with no space in between), have leading zeros, be surrounded by whitespace, and have single underscores interspersed between digits.

A base-n integer string contains digits, each representing a value from 0 to n-1. The values 0–9 can be represented by any Unicode decimal digit. The values 10–35 can be represented by `a` to `z` (or `A` to `Z`). The default *base* is 10. The allowed bases are 0 and 2–36. Base-2, -8, and -16 strings can be optionally prefixed with `0b`/`0B`, `0o`/`0O`, or `0x`/`0X`, as with integer literals in code. For base 0, the string is interpreted in a similar way to an integer literal in code, in that the actual base is 2, 8, 10, or 16 as determined by the prefix. Base 0 also disallows leading zeros: `int('010', 0)` is not legal, while `int('010')` and `int('010', 8)` are.

The integer type is described in Numeric Types — int, float, complex.

Changed in version 3.4: If *base* is not an instance of **`int`** and the *base* object has a **`base.__index__`** method, that method is called to obtain an integer for the base. Previous versions used **`base.__int__`** instead of **`base.__index__`**.

Changed in version 3.6: Grouping digits with underscores as in code literals is allowed.

Changed in version 3.7: The first parameter is now positional-only.

Changed in version 3.8: Falls back to **`__index__()`** if **`__int__()`** is not defined.

Changed in version 3.11: **`int`** string inputs and string representations can be limited to help avoid denial of service attacks. A **`ValueError`** is raised when the limit is exceeded while converting a string to an **`int`** or when converting an **`int`** into a string would exceed the limit. See the integer string conversion length limitation documentation.

Changed in version 3.14: **`int()`** no longer delegates to the **`__trunc__()`** method.

isinstance(*object*, *classinfo*, /)

Return `True` if the *object* argument is an instance of the *classinfo* argument, or of a (direct, indirect, or virtual) subclass thereof. If *object* is not an object of the given type, the function always returns `False`. If *classinfo* is a tuple of type objects (or recursively, other such tuples) or a Union Type of multiple types, return `True` if *object* is an instance of any of the types. If *classinfo* is not a type or tuple of types and such tuples, a **`TypeError`** exception is raised. **`TypeError`** may not be raised for an invalid type if an earlier check succeeds.

Changed in version 3.10: classinfo can be a Union Type.

issubclass(*class*, *classinfo*, */*)

Return `True` if *class* is a subclass (direct, indirect, or virtual) of *classinfo*. A class is considered a subclass of itself. *classinfo* may be a tuple of class objects (or recursively, other such tuples) or a Union Type, in which case return `True` if *class* is a subclass of any entry in *classinfo*. In any other case, a **`TypeError`** exception is raised.

Changed in version 3.10: classinfo can be a Union Type.

iter(*iterable*, */*)
iter(*callable*, *sentinel*, */*)

Return an iterator object. The first argument is interpreted very differently depending on the presence of the second argument. Without a second argument, the single argument must be a collection object which supports the iterable protocol (the **`__iter__()`** method), or it must support the sequence protocol (the **`__getitem__()`** method with integer arguments starting at `0`). If it does not support either of those protocols, **`TypeError`** is raised. If the second argument, *sentinel*, is given, then the first argument must be a callable object. The iterator created in this case will call *callable* with no arguments for each call to its **`__next__()`** method; if the value returned is equal to *sentinel*, **`StopIteration`** will be raised, otherwise the value will be returned.

See also Iterator Types.

One useful application of the second form of **`iter()`** is to build a block-reader. For example, reading fixed-width blocks from a binary database file until the end of file is reached:

```
from functools import partial
with open('mydata.db', 'rb') as f:
    for block in iter(partial(f.read, 64), b''):
        process_block(block)
```

len(*object*, /)

Return the length (the number of items) of an object. The argument may be a sequence (such as a string, bytes, tuple, list, or range) or a collection (such as a dictionary, set, or frozen set).

CPython implementation detail: `len` raises `OverflowError` on lengths larger than `sys.maxsize`, such as `range(2 ** 100)`.

class list(*iterable=()*, /)

Rather than being a function, `list` is actually a mutable sequence type, as documented in Lists and Sequence Types — list, tuple, range.

locals()

Return a mapping object representing the current local symbol table, with variable names as the keys, and their currently bound references as the values.

At module scope, as well as when using `exec()` or `eval()` with a single namespace, this function returns the same namespace as `globals()`.

At class scope, it returns the namespace that will be passed to the metaclass constructor.

When using `exec()` or `eval()` with separate local and global arguments, it returns the local namespace passed in to the function call.

In all of the above cases, each call to `locals()` in a given frame of execution will return the *same* mapping object. Changes made through the mapping object returned from `locals()` will be visible as assigned, reassigned, or deleted local variables, and assigning, reassigning, or deleting local variables will immediately affect the contents of the returned mapping object.

In an optimized scope (including functions, generators, and coroutines), each call to `locals()` instead returns a fresh dictionary containing the current bindings of the function’s local variables and any nonlocal cell references. In this case, name binding changes made via the returned dict are *not* written back to the corresponding local variables or nonlocal cell references, and assigning, reassigning, or deleting local variables and nonlocal cell references does *not* affect the contents of previously returned dictionaries.

Calling `locals()` as part of a comprehension in a function, generator, or coroutine is equivalent to calling it in the containing scope, except that the comprehension’s initialised iteration variables will be included. In other scopes, it behaves as if the comprehension were running as a nested function.

Calling `locals()` as part of a generator expression is equivalent to calling it in a nested generator function.

Changed in version 3.12: The behaviour of `locals()` in a comprehension has been updated as described in **PEP 709** [https://peps.python.org/pep-0709/].

Changed in version 3.13: As part of **PEP 667** [https://peps.python.org/pep-0667/], the semantics of mutating the mapping objects returned from this function are now defined. The behavior in optimized scopes is now as described above. Aside from being defined, the behaviour in other scopes remains unchanged from previous versions.

map(*function*, *iterable*, */*, **iterables*, *strict=False*)

Return an iterator that applies *function* to every item of *iterable*, yielding the results. If additional *iterables* arguments are passed, *function* must take that many arguments and is applied to the items from all iterables in parallel. With multiple iterables, the iterator stops when the shortest iterable is exhausted. If *strict* is `True` and one of the iterables is exhausted before the others, a **`ValueError`** is raised. For cases where the function inputs are already arranged into argument tuples, see **`itertools.starmap()`**.

Changed in version 3.14: Added the *strict* parameter.

max(*iterable*, */*, ***, *key=None*)
max(*iterable*, */*, ***, *default*, *key=None*)
max(*arg1*, *arg2*, */*, **args*, *key=None*)

Return the largest item in an iterable or the largest of two or more arguments.

If one positional argument is provided, it should be an iterable. The largest item in the iterable is returned. If two or more positional arguments are provided, the largest of the positional arguments is returned.

There are two optional keyword-only arguments. The *key* argument specifies a one-argument ordering function like that used for **`list.sort()`**. The *default* argument specifies an object to return if the provided iterable is empty. If the iterable is empty and *default* is not provided, a **`ValueError`** is raised.

If multiple items are maximal, the function returns the first one encountered. This is consistent with other sort-stability preserving tools such as `sorted(iterable, key=keyfunc, reverse=True)[0]` and `heapq.nlargest(1, iterable, key=keyfunc)`.

Changed in version 3.4: Added the *default* keyword-only parameter.

Changed in version 3.8: The *key* can be `None`.

class memoryview(*object*)

Return a “memory view” object created from the given argument. See Memory Views for more information.

min(*iterable*, */*, ***, *key=None*)
min(*iterable*, */*, ***, *default*, *key=None*)
min(*arg1*, *arg2*, */*, **args*, *key=None*)

Return the smallest item in an iterable or the smallest of two or more arguments.

If one positional argument is provided, it should be an iterable. The smallest item in the iterable is returned. If two or more positional arguments are provided, the smallest of the positional arguments is returned.

There are two optional keyword-only arguments. The *key* argument specifies a one-argument ordering function like that used for **`list.sort()`**. The *default* argument specifies an object to return if the provided iterable is empty. If the iterable is empty and *default* is not provided, a **`ValueError`** is raised.

If multiple items are minimal, the function returns the first one encountered. This is consistent with other sort-stability preserving tools such as `sorted(iterable, key=keyfunc)[0]` and `heapq.nsmallest(1, iterable, key=keyfunc)`.

Changed in version 3.4: Added the *default* keyword-only parameter.

Changed in version 3.8: The *key* can be `None`.

next(*iterator*, /)
next(*iterator*, *default*, /)

Retrieve the next item from the iterator by calling its `__next__()` method. If *default* is given, it is returned if the iterator is exhausted, otherwise `StopIteration` is raised.

class object

This is the ultimate base class of all other classes. It has methods that are common to all instances of Python classes. When the constructor is called, it returns a new featureless object. The constructor does not accept any arguments.

Note

`object` instances do *not* have `__dict__` attributes, so you can’t assign arbitrary attributes to an instance of `object`.

oct(*integer*, /)

Convert an integer number to an octal string prefixed with “0o”. The result is a valid Python expression. If *integer* is not a Python `int` object, it has to define an `__index__()` method that returns an integer. For example:

```
>>> oct(8)
'0o10'
>>> oct(-56)
'-0o70'
```

If you want to convert an integer number to an octal string either with the prefix “0o” or not, you can use either of the following ways.

```
>>> '%#o' % 10, '%o' % 10
('0o12', '12')
>>> format(10, '#o'), format(10, 'o')
('0o12', '12')
```

```
>>> f'{10:#o}', f'{10:o}'
('0o12', '12')
```

See also `format()` for more information.

open(*file*, *mode='r'*, *buffering=-1*, *encoding=None*, *errors=None*, *newline=None*, *closefd=True*, *opener=None*)

Open *file* and return a corresponding file object. If the file cannot be opened, an `OSError` is raised. See Reading and Writing Files for more examples of how to use this function.

file is a path-like object giving the pathname (absolute or relative to the current working directory) of the file to be opened or an integer file descriptor of the file to be wrapped. (If a file descriptor is given, it is closed when the returned I/O object is closed unless *closefd* is set to `False`.)

mode is an optional string that specifies the mode in which the file is opened. It defaults to `'r'` which means open for reading in text mode. Other common values are `'w'` for writing (truncating the file if it already exists), `'x'` for exclusive creation, and `'a'` for appending (which on *some* Unix systems, means that *all* writes append to the end of the file regardless of the current seek position). In text mode, if *encoding* is not specified the encoding used is platform-dependent: `locale.getencoding()` is called to get the current locale encoding. (For reading and writing raw bytes use binary mode and leave *encoding* unspecified.) The available modes are:

Character	Meaning
`'r'`	open for reading (default)
`'w'`	open for writing, truncating the file first

Character	Meaning
`'x'`	open for exclusive creation, failing if the file already exists
`'a'`	open for writing, appending to the end of file if it exists
`'b'`	binary mode
`'t'`	text mode (default)
`'+'`	open for updating (reading and writing)

The default mode is `'r'` (open for reading text, a synonym of `'rt'`). Modes `'w+'` and `'w+b'` open and truncate the file. Modes `'r+'` and `'r+b'` open the file with no truncation.

As mentioned in the Overview, Python distinguishes between binary and text I/O. Files opened in binary mode (including `'b'` in the *mode* argument) return contents as **`bytes`** objects without any decoding. In text mode (the default, or when `'t'` is included in the *mode* argument), the contents of the file are returned as **`str`**, the bytes having been first decoded using a platform-dependent encoding or using the specified *encoding* if given.

Note

Python doesn’t depend on the underlying operating system’s notion of text files; all the processing is done by Python itself, and is therefore platform-independent.

buffering is an optional integer used to set the buffering policy. Pass 0 to switch buffering off (only allowed in binary mode), 1 to select line buffering (only usable when writing in text mode), and an integer > 1 to indicate the size in bytes of a fixed-size chunk buffer. Note that specifying a buffer size this way applies for binary buffered I/O, but `TextIOWrapper` (i.e., files opened with `mode='r+'`) would have another buffering. To disable buffering in `TextIOWrapper`, consider using the `write_through` flag for **`io.TextIOWrapper.reconfigure()`**. When no *buffering* argument is given, the default buffering policy works as follows:

- Binary files are buffered in fixed-size chunks; the size of the buffer is `max(min(blocksize, 8 MiB), DEFAULT_BUFFER_SIZE)` when the device block size is available. On most systems, the buffer will typically be 128 kilobytes long.
- “Interactive” text files (files for which **`isatty()`** returns `True`) use line buffering. Other text files use the policy described above for binary files.

encoding is the name of the encoding used to decode or encode the file. This should only be used in text mode. The default encoding is platform dependent (whatever **`locale.getencoding()`** returns), but any text encoding supported by Python can be used. See the **`codecs`** module for the list of supported encodings.

errors is an optional string that specifies how encoding and decoding errors are to be handled—this cannot be used in binary mode. A variety of standard error handlers are available (listed under Error Handlers), though any error handling name that has been registered with **`codecs.register_error()`** is also valid. The standard names include:

- `'strict'` to raise a **`ValueError`** exception if there is an encoding error. The default value of `None` has the same effect.

- `'ignore'` ignores errors. Note that ignoring encoding errors can lead to data loss.
- `'replace'` causes a replacement marker (such as `'?'`) to be inserted where there is malformed data.
- `'surrogateescape'` will represent any incorrect bytes as low surrogate code units ranging from U+DC80 to U+DCFF. These surrogate code units will then be turned back into the same bytes when the `surrogateescape` error handler is used when writing data. This is useful for processing files in an unknown encoding.
- `'xmlcharrefreplace'` is only supported when writing to a file. Characters not supported by the encoding are replaced with the appropriate XML character reference `&#nnn;`.
- `'backslashreplace'` replaces malformed data by Python's backslashed escape sequences.
- `'namereplace'` (also only supported when writing) replaces unsupported characters with `\N{...}` escape sequences.

newline determines how to parse newline characters from the stream. It can be `None`, `''`, `'\n'`, `'\r'`, and `'\r\n'`. It works as follows:

- When reading input from the stream, if *newline* is `None`, universal newlines mode is enabled. Lines in the input can end in `'\n'`, `'\r'`, or `'\r\n'`, and these are translated into `'\n'` before being returned to the caller. If it is `''`, universal newlines mode is enabled, but line endings are returned to the caller untranslated. If it has any of the other legal values, input lines are only terminated by the given string, and the line ending is returned to the caller untranslated.
- When writing output to the stream, if *newline* is `None`, any `'\n'` characters written are translated to the system default line separator, `os.linesep`. If *newline* is

`''` or `'\n'`, no translation takes place. If *newline* is any of the other legal values, any `'\n'` characters written are translated to the given string.

If *closefd* is `False` and a file descriptor rather than a filename was given, the underlying file descriptor will be kept open when the file is closed. If a filename is given *closefd* must be `True` (the default); otherwise, an error will be raised.

A custom opener can be used by passing a callable as *opener*. The underlying file descriptor for the file object is then obtained by calling *opener* with (*file*, *flags*). *opener* must return an open file descriptor (passing **`os.open`** as *opener* results in functionality similar to passing `None`).

The newly created file is non-inheritable.

The following example uses the dir_fd parameter of the **`os.open()`** function to open a file relative to a given directory:

```
>>> import os
>>> dir_fd = os.open('somedir', os.O_RDONLY)
>>> def opener(path, flags):
...     return os.open(path, flags, dir_fd=dir_fd)
...
>>> with open('spamspam.txt', 'w', opener=opener) as f:
...     print('This will be written to somedir/spamspam.txt',
file=f)
...
>>> os.close(dir_fd)  # don't leak a file descriptor
```

The type of file object returned by the **`open()`** function depends on the mode. When **`open()`** is used to open a file in a text mode (`'w'`, `'r'`, `'wt'`, `'rt'`, etc.), it returns a subclass of **`io.TextIOBase`** (specifically **`io.TextIOWrapper`**). When used to open a file in a binary mode with buffering, the returned class is a subclass of **`io.BufferedIOBase`**. The exact class varies: in read binary mode, it returns an **`io.BufferedReader`**;

in write binary and append binary modes, it returns an **`io.BufferedWriter`**, and in read/write mode, it returns an **`io.BufferedRandom`**. When buffering is disabled, the raw stream, a subclass of **`io.RawIOBase`**, **`io.FileIO`**, is returned.

See also the file handling modules, such as **`fileinput`**, **`io`** (where **`open()`** is declared), **`os`**, **`os.path`**, **`tempfile`**, and **`shutil`**.

Raises an auditing event `open` with arguments `path`, `mode`, `flags`.

The `mode` and `flags` arguments may have been modified or inferred from the original call.

Changed in version 3.3:

- The *opener* parameter was added.
- The `'x'` mode was added.
- **`IOError`** used to be raised, it is now an alias of **`OSError`**.
- **`FileExistsError`** is now raised if the file opened in exclusive creation mode (`'x'`) already exists.

Changed in version 3.4:

- The file is now non-inheritable.

Changed in version 3.5:

- If the system call is interrupted and the signal handler does not raise an exception, the function now retries the system call instead of raising an **`InterruptedError`** exception (see **PEP 475** [https://peps.python.org/pep-0475/] for the rationale).
- The `'namereplace'` error handler was added.

Changed in version 3.6:

- Support added to accept objects implementing **`os.PathLike`**.

- On Windows, opening a console buffer may return a subclass of **`io.RawIOBase`** other than **`io.FileIO`**.

Changed in version 3.11: The `'U'` mode has been removed.

ord(*character*, */*)

Return the ordinal value of a character.

If the argument is a one-character string, return the Unicode code point of that character. For example, `ord('a')` returns the integer `97` and `ord('€')` (Euro sign) returns `8364`. This is the inverse of **`chr()`**.

If the argument is a **`bytes`** or **`bytearray`** object of length 1, return its single byte value. For example, `ord(b'a')` returns the integer `97`.

pow(*base*, *exp*, *mod=None*)

Return *base* to the power *exp*; if *mod* is present, return *base* to the power *exp*, modulo *mod* (computed more efficiently than `pow(base, exp) % mod`). The two-argument form `pow(base, exp)` is equivalent to using the power operator: `base**exp`.

When arguments are builtin numeric types with mixed operand types, the coercion rules for binary arithmetic operators apply. For **`int`** operands, the result has the same type as the operands (after coercion) unless the second argument is negative; in that case, all arguments are converted to float and a float result is delivered. For example, `pow(10, 2)` returns `100`, but `pow(10, -2)` returns `0.01`. For a negative base of type **`int`** or **`float`** and a non-integral exponent, a complex result is delivered. For example, `pow(-9, 0.5)` returns a value close to `3j`. Whereas, for a negative base of type **`int`** or **`float`** with an integral exponent, a float result is delivered. For example, `pow(-9, 2.0)` returns `81.0`.

For **`int`** operands *base* and *exp*, if *mod* is present, *mod* must also be of integer type and *mod* must be nonzero. If *mod* is present and *exp* is negative, *base* must be relatively prime to *mod*. In that case, `pow(inv_base, -exp, mod)` is returned, where *inv_base* is an inverse to *base* modulo *mod*.

Here’s an example of computing an inverse for `38` modulo `97`:

```
>>> pow(38, -1, mod=97)
23
>>> 23 * 38 % 97 == 1
True
```

Changed in version 3.8: For **`int`** operands, the three-argument form of `pow` now allows the second argument to be negative, permitting computation of modular inverses.

Changed in version 3.8: Allow keyword arguments. Formerly, only positional arguments were supported.

print(**objects*, *sep=' '*, *end='\n'*, *file=None*, *flush=False*)

Print *objects* to the text stream *file*, separated by *sep* and followed by *end*. *sep*, *end*, *file*, and *flush*, if present, must be given as keyword arguments.

All non-keyword arguments are converted to strings like **`str()`** does and written to the stream, separated by *sep* and followed by *end*. Both *sep* and *end* must be strings; they can also be `None`, which means to use the default values. If no *objects* are given, **`print()`** will just write *end*.

The *file* argument must be an object with a `write(string)` method; if it is not present or `None`, **`sys.stdout`** will be used. Since printed arguments are converted to text strings, **`print()`** cannot be used with binary mode file objects. For these, use `file.write(...)` instead.

Output buffering is usually determined by *file*. However, if *flush* is true, the stream is forcibly flushed.

Changed in version 3.3: Added the *flush* keyword argument.

class property(*fget=None*, *fset=None*, *fdel=None*, *doc=None*)

Return a property attribute.

fget is a function for getting an attribute value. *fset* is a function for setting an attribute value. *fdel* is a function for deleting an attribute value. And *doc* creates a docstring for the attribute.

A typical use is to define a managed attribute `x`:

```
class C:
    def __init__(self):
        self._x = None

    def getx(self):
        return self._x

    def setx(self, value):
        self._x = value

    def delx(self):
        del self._x

    x = property(getx, setx, delx, "I'm the 'x' property.")
```

If *c* is an instance of *C*, `c.x` will invoke the getter, `c.x = value` will invoke the setter, and `del c.x` the deleter.

If given, *doc* will be the docstring of the property attribute. Otherwise, the property will copy *fget*’s docstring (if it exists). This makes it possible to create read-only properties easily using **`property()`** as a decorator:

```
class Parrot:
    def __init__(self):
        self._voltage = 100000

    @property
    def voltage(self):
        """Get the current voltage."""
        return self._voltage
```

The `@property` decorator turns the **`voltage()`** method into a “getter” for a read-only attribute with the same name, and it sets the docstring for *voltage* to “Get the current voltage.”

@getter

@setter

@deleter

A property object has `getter`, `setter`, and `deleter` methods usable as decorators that create a copy of the property with the corresponding accessor function set to the decorated function. This is best explained with an example:

```
class C:
    def __init__(self):
        self._x = None

    @property
    def x(self):
        """I'm the 'x' property."""
        return self._x

    @x.setter
    def x(self, value):
        self._x = value

    @x.deleter
    def x(self):
        del self._x
```

This code is exactly equivalent to the first example. Be sure to give the additional functions the same name as the original property (`x` in this case.)

The returned property object also has the attributes `fget`, `fset`, and `fdel` corresponding to the constructor arguments.

Changed in version 3.5: The docstrings of property objects are now writeable.

__name__

Attribute holding the name of the property. The name of the property can be changed at runtime.

Added in version 3.13.

class range(*stop*, */*)
class range(*start*, *stop*, *step=1*, */*)

Rather than being a function, **`range`** is actually an immutable sequence type, as documented in Ranges and Sequence Types — list, tuple, range.

repr(*object*, */*)

Return a string containing a printable representation of an object. For many types, this function makes an attempt to return a string that would yield an object with the same value when passed to **`eval()`**; otherwise, the representation is a string enclosed in angle brackets that contains the name of the type of the object together with additional information often including the name and address of the object. A class can control what this function returns for its instances by defining a **`__repr__()`** method. If **`sys.displayhook()`** is not accessible, this function will raise **`RuntimeError`**.

This class has a custom representation that can be evaluated:

```
class Person:
   def __init__(self, name, age):
      self.name = name
      self.age = age

   def __repr__(self):
      return f"Person('{self.name}', {self.age})"
```

reversed(*object*, /)

Return a reverse iterator. The argument must be an object which has a **`__reversed__()`** method or supports the sequence protocol (the **`__len__()`** method and the **`__getitem__()`** method with integer arguments starting at `0`).

round(*number*, *ndigits=None*)

Return *number* rounded to *ndigits* precision after the decimal point. If *ndigits* is omitted or is `None`, it returns the nearest integer to its input.

For the built-in types supporting **`round()`**, values are rounded to the closest multiple of 10 to the power minus *ndigits*; if two multiples are equally close, rounding is done toward the even choice (so, for example, both `round(0.5)` and `round(-0.5)` are `0`, and `round(1.5)` is `2`). Any integer value is valid for *ndigits* (positive, zero, or negative). The return value is an integer if *ndigits* is omitted or `None`. Otherwise, the return value has the same type as *number*.

For a general Python object `number`, `round` delegates to `number.__round__`.

Note

The behavior of **`round()`** for floats can be surprising: for example, `round(2.675, 2)` gives `2.67` instead of the expected `2.68`. This is not a bug: it’s a result of the fact that most decimal fractions can’t be represented exactly as a float. See Floating-Point Arithmetic: Issues and Limitations for more information.

class set(*iterable=()*, */*)

Return a new **`set`** object, optionally with elements taken from *iterable*. `set` is a built-in class. See **`set`** and Set Types — set, frozenset for documentation about this class.

For other containers see the built-in **`frozenset`**, **`list`**, **`tuple`**, and **`dict`** classes, as well as the **`collections`** module.

setattr(*object*, *name*, *value*, */*)

This is the counterpart of **`getattr()`**. The arguments are an object, a string, and an arbitrary value. The string may name an existing attribute or a new attribute. The function assigns the value to the attribute, provided the object allows it. For example, `setattr(x, 'foobar', 123)` is equivalent to `x.foobar = 123`.

name need not be a Python identifier as defined in Names (identifiers and keywords) unless the object chooses to enforce that, for example in a custom **`__getattribute__()`** or via **`__slots__`**. An attribute whose name is not an identifier will not be accessible using the dot notation, but is accessible through **`getattr()`** etc..

Note

Since private name mangling happens at compilation time, one must manually mangle a private attribute’s (attributes with two leading underscores) name in order to set it with **`setattr()`**.

class slice(*stop*, */*)
class slice(*start*, *stop*, *step=None*, */*)

Return a slice object representing the set of indices specified by `range(start, stop, step)`. The *start* and *step* arguments default to `None`.

Slice objects have read-only data attributes **`start`**, **`stop`**, and **`step`** which merely return the argument values (or their default). They have no other explicit functionality; however, they are used by NumPy and other third-party packages.

start

stop

step

Slice objects are also generated when extended indexing syntax is used. For example: `a[start:stop:step]` or `a[start:stop, i]`. See **`itertools.islice()`** for an alternate version that returns an iterator.

Changed in version 3.12: Slice objects are now hashable (provided **`start`**, **`stop`**, and **`step`** are hashable).

sorted(*iterable*, */*, ***, *key=None*, *reverse=False*)

Return a new sorted list from the items in *iterable*.

Has two optional arguments which must be specified as keyword arguments.

key specifies a function of one argument that is used to extract a comparison key from each element in *iterable* (for example, `key=str.lower`). The default value is `None` (compare the elements directly).

reverse is a boolean value. If set to `True`, then the list elements are sorted as if each comparison were reversed.

Use **`functools.cmp_to_key()`** to convert an old-style *cmp* function to a *key* function.

The built-in **`sorted()`** function is guaranteed to be stable. A sort is stable if it guarantees not to change the relative order of elements that compare equal — this is helpful for sorting in multiple passes (for example, sort by department, then by salary grade).

The sort algorithm uses only `<` comparisons between items. While defining an **`__lt__()`** method will suffice for sorting, **PEP 8** [https://peps.python.org/pep-0008/] recommends that all six rich comparisons be implemented. This will help avoid bugs when using the same data with other ordering tools such as **`max()`** that rely on a different underlying method. Implementing all six comparisons also helps avoid confusion for mixed type comparisons which can call the reflected **`__gt__()`** method.

For sorting examples and a brief sorting tutorial, see Sorting Techniques.

@staticmethod

Transform a method into a static method.

A static method does not receive an implicit first argument. To declare a static method, use this idiom:

```
class C:
    @staticmethod
    def f(arg1, arg2, argN): ...
```

The `@staticmethod` form is a function decorator – see Function definitions for details.

A static method can be called either on the class (such as `C.f()`) or on an instance (such as `C().f()`). Moreover, the static method descriptor is also callable, so it can be used in the class definition (such as `f()`).

Static methods in Python are similar to those found in Java or C++. Also, see **`classmethod()`** for a variant that is useful for creating alternate class constructors.

Like all decorators, it is also possible to call `staticmethod` as a regular function and do something with its result. This is needed in some cases where you need a reference to a function from a class body and you want to avoid the automatic transformation to instance method. For these cases, use this idiom:

```
def regular_function():
    ...

class C:
    method = staticmethod(regular_function)
```

For more information on static methods, see The standard type hierarchy.

Changed in version 3.10: Static methods now inherit the method attributes (**`__module__`**, **`__name__`**, **`__qualname__`**, **`__doc__`** and **`__annotations__`**), have a new `__wrapped__` attribute, and are now callable as regular functions.

class str(*, *encoding*='*utf-8*', *errors*='*strict*')
class str(*object*)
class str(*object*, *encoding*, *errors*='*strict*')
class str(*object*, *, *errors*)

Return a **`str`** version of *object*. See **`str()`** for details.

`str` is the built-in string class. For general information about strings, see Text Sequence Type — str.

sum(*iterable*, /, *start=0*)

Sums *start* and the items of an *iterable* from left to right and returns the total. The *iterable*’s items are normally numbers, and the start value is not allowed to be a string.

For some use cases, there are good alternatives to **`sum()`**. The preferred, fast way to concatenate a sequence of strings is by calling `''.join(sequence)`. To add floating-point values with extended precision, see **`math.fsum()`**. To concatenate a series of iterables, consider using **`itertools.chain()`**.

Changed in version 3.8: The *start* parameter can be specified as a keyword argument.

Changed in version 3.12: Summation of floats switched to an algorithm that gives higher accuracy and better commutativity on most builds.

Changed in version 3.14: Added specialization for summation of complexes, using same algorithm as for summation of floats.

class super
class super(*type*, *object_or_type=None*, /)

Return a proxy object that delegates method calls to a parent or sibling class of *type*. This is useful for accessing inherited methods that have been overridden in a class.

The *object_or_type* determines the method resolution order to be searched. The search starts from the class right after the *type*.

For example, if **`__mro__`** of *object_or_type* is `D -> B -> C -> A -> object` and the value of *type* is `B`, then **`super()`** searches `C -> A -> object`.

The `__mro__` attribute of the class corresponding to *object_or_type* lists the method resolution search order used by both `getattr()` and `super()`. The attribute is dynamic and can change whenever the inheritance hierarchy is updated.

If the second argument is omitted, the super object returned is unbound. If the second argument is an object, `isinstance(obj, type)` must be true. If the second argument is a type, `issubclass(type2, type)` must be true (this is useful for classmethods).

When called directly within an ordinary method of a class, both arguments may be omitted (“zero-argument **`super()`**”). In this case, *type* will be the enclosing class, and *obj* will be the first argument of the immediately enclosing function (typically `self`). (This means that zero-argument **`super()`** will not work as expected within nested functions, including generator expressions, which implicitly create nested functions.)

There are two typical use cases for *super*. In a class hierarchy with single inheritance, *super* can be used to refer to parent classes without naming them explicitly, thus making the code more maintainable. This use closely parallels the use of *super* in other programming languages.

The second use case is to support cooperative multiple inheritance in a dynamic execution environment. This use case is unique to Python and is not found in statically compiled languages or languages that only support single inheritance. This makes it possible to implement “diamond diagrams” where multiple base classes implement the same method. Good design dictates that such implementations have the same calling signature in every case (because the order of calls is determined at runtime, because that order adapts to changes in the class hierarchy, and because that order can include sibling classes that are unknown prior to runtime).

For both use cases, a typical superclass call looks like this:

```
class C(B):
    def method(self, arg):
        super().method(arg)    # This does the same thing as:
                               # super(C, self).method(arg)
```

In addition to method lookups, **`super()`** also works for attribute lookups. One possible use case for this is calling descriptors in a parent or sibling class.

Note that **`super()`** is implemented as part of the binding process for explicit dotted attribute lookups such as `super().__getitem__(name)`. It does so by implementing its own **`__getattribute__()`** method for searching classes in a predictable order that supports cooperative multiple inheritance. Accordingly, **`super()`** is undefined for implicit lookups using statements or operators such as `super()[name]`.

Also note that, aside from the zero argument form, **`super()`** is not limited to use inside methods. The two argument form specifies the arguments exactly and makes the appropriate references. The zero argument form only works inside a class definition, as the compiler fills in the necessary details to correctly retrieve the class being defined, as well as accessing the current instance for ordinary methods.

For practical suggestions on how to design cooperative classes using **`super()`**, see guide to using super() [https://rhettinger.wordpress.com/2011/05/26/super-considered-super/].

Changed in version 3.14: `super` objects are now `pickleable` and `copyable`.

class tuple(*iterable=()*, */*)

Rather than being a function, `tuple` is actually an immutable sequence type, as documented in Tuples and Sequence Types — list, tuple, range.

class type(*object*, */*)
class type(*name*, *bases*, *dict*, */*, ***kwargs*)

With one argument, return the type of an *object*. The return value is a type object and generally the same object as returned by `object.__class__`.

The `isinstance()` built-in function is recommended for testing the type of an object, because it takes subclasses into account.

With three arguments, return a new type object. This is essentially a dynamic form of the `class` statement. The *name* string is the class name and becomes the `__name__` attribute. The *bases* tuple contains the base classes and becomes the `__bases__` attribute; if empty, `object`, the ultimate base of all classes, is added. The *dict* dictionary contains attribute and method definitions for the class body; it may be copied or wrapped before becoming the `__dict__` attribute. The following two statements create identical **`type`** objects:

```
>>> class X:
...     a = 1
...
>>> X = type('X', (), dict(a=1))
```

See also:

- Documentation on attributes and methods on classes.
- Type Objects

Keyword arguments provided to the three argument form are passed to the appropriate metaclass machinery (usually `__init_subclass__()`) in the same way that keywords in a class definition (besides *metaclass*) would.

See also Customizing class creation.

Changed in version 3.6: Subclasses of **`type`** which don't override `type.__new__` may no longer use the one-argument form to get the type of an object.

vars()
vars(*object*, /)

Return the `__dict__` attribute for a module, class, instance, or any other object with a **`__dict__`** attribute.

Objects such as modules and instances have an updateable `__dict__` attribute; however, other objects may have write restrictions on their **`__dict__`** attributes (for example, classes use a `types.MappingProxyType` to prevent direct dictionary updates).

Without an argument, `vars()` acts like `locals()`.

A `TypeError` exception is raised if an object is specified but it doesn't have a `__dict__` attribute (for example, if its class defines the `__slots__` attribute).

Changed in version 3.13: The result of calling this function without an argument has been updated as described for the `locals()` builtin.

zip(**iterables*, *strict=False*)

Iterate over several iterables in parallel, producing tuples with an item from each one.

Example:

```
>>> for item in zip([1, 2, 3], ['sugar', 'spice', 'everything
nice']):
...     print(item)
...
(1, 'sugar')
(2, 'spice')
(3, 'everything nice')
```

More formally: **`zip()`** returns an iterator of tuples, where the *i*-th tuple contains the *i*-th element from each of the argument iterables.

Another way to think of **`zip()`** is that it turns rows into columns, and columns into rows. This is similar to transposing a matrix [https://en.wikipedia.org/wiki/Transpose].

`zip()` is lazy: The elements won't be processed until the iterable is iterated on, e.g. by a **`for`** loop or by wrapping in a **`list`**.

One thing to consider is that the iterables passed to **`zip()`** could have different lengths; sometimes by design, and sometimes because of a bug in the code that prepared these iterables. Python offers three different approaches to dealing with this issue:

- By default, **`zip()`** stops when the shortest iterable is exhausted. It will ignore the remaining items in the longer iterables, cutting off the result to the length of the shortest iterable:

  ```
  >>> list(zip(range(3), ['fee', 'fi', 'fo', 'fum']))
  [(0, 'fee'), (1, 'fi'), (2, 'fo')]
  ```

- **`zip()`** is often used in cases where the iterables are assumed to be of equal length. In such cases, it's recommended to use the `strict=True` option. Its output is the same as regular **`zip()`**:

```
>>> list(zip(('a', 'b', 'c'), (1, 2, 3), strict=True))
[('a', 1), ('b', 2), ('c', 3)]
```

Unlike the default behavior, it raises a **`ValueError`** if one iterable is exhausted before the others:

```
>>> for item in zip(range(3), ['fee', 'fi', 'fo', 'fum'],
strict=True):
...     print(item)
...
(0, 'fee')
(1, 'fi')
(2, 'fo')
Traceback (most recent call last):
  ...
ValueError: zip() argument 2 is longer than argument 1
```

Without the `strict=True` argument, any bug that results in iterables of different lengths will be silenced, possibly manifesting as a hard-to-find bug in another part of the program.

- Shorter iterables can be padded with a constant value to make all the iterables have the same length. This is done by **`itertools.zip_longest()`**.

Edge cases: With a single iterable argument, **`zip()`** returns an iterator of 1-tuples. With no arguments, it returns an empty iterator.

Tips and tricks:

- The left-to-right evaluation order of the iterables is guaranteed. This makes possible an idiom for clustering a data series into n-length groups using `zip(*[iter(s)]*n, strict=True)`. This repeats the *same* iterator `n` times so that each output tuple has the result of `n` calls to the iterator. This has the effect of dividing the input into n-length chunks.

- `zip()` in conjunction with the `*` operator can be used to unzip a list:

  ```
  >>> x = [1, 2, 3]
  >>> y = [4, 5, 6]
  >>> list(zip(x, y))
  [(1, 4), (2, 5), (3, 6)]
  >>> x2, y2 = zip(*zip(x, y))
  >>> x == list(x2) and y == list(y2)
  True
  ```

Changed in version 3.10: Added the `strict` argument.

__import__(*name*, *globals=None*, *locals=None*, *fromlist=()*, *level=0*)

Note

This is an advanced function that is not needed in everyday Python programming, unlike `importlib.import_module()`.

This function is invoked by the `import` statement. It can be replaced (by importing the `builtins` module and assigning to `builtins.__import__`) in order to change semantics of the **`import`** statement, but doing so is **strongly** discouraged as it is usually simpler to use import hooks (see **PEP 302** [https://peps.python.org/pep-0302/]) to attain the same goals and does not cause issues with code which assumes the default import implementation is in use. Direct use of `__import__()` is also discouraged in favor of `importlib.import_module()`.

The function imports the module *name*, potentially using the given *globals* and *locals* to determine how to interpret the name in a package context. The *fromlist* gives the names of objects or submodules that should be imported from the module given by *name*. The standard implementation does not use its *locals* argument at all and uses its *globals* only to determine the package context of the **`import`** statement.

level specifies whether to use absolute or relative imports. `0` (the default) means only perform absolute imports. Positive values for *level* indicate the number of parent directories to search relative to the directory of the module calling `__import__()` (see **PEP 328** [https://peps.python.org/pep-0328/] for the details).

When the *name* variable is of the form `package.module`, normally, the top-level package (the name up till the first dot) is returned, *not* the module named by *name*. However, when a non-empty *fromlist* argument is given, the module named by *name* is returned.

For example, the statement `import spam` results in bytecode resembling the following code:

```
spam = __import__('spam', globals(), locals(), [], 0)
```

The statement `import spam.ham` results in this call:

```
spam = __import__('spam.ham', globals(), locals(), [], 0)
```

Note how `__import__()` returns the toplevel module here because this is the object that is bound to a name by the **`import`** statement.

On the other hand, the statement `from spam.ham import eggs, sausage as saus` results in

```
_temp = __import__('spam.ham', globals(), locals(), ['eggs',
'sausage'], 0)
eggs = _temp.eggs
saus = _temp.sausage
```

Here, the `spam.ham` module is returned from `__import__()`. From this object, the names to import are retrieved and assigned to their respective names.

If you simply want to import a module (potentially within a package) by name, use `importlib.import_module()`.

Changed in version 3.3: Negative values for *level* are no longer supported (which also changes the default value to 0).

Changed in version 3.9: When the command line options `-E` or `-I` are being used, the environment variable `PYTHONCASEOK` is now ignored.

Footnotes

[1]

Note that the parser only accepts the Unix-style end of line convention. If you are reading the code from a file, make sure to use newline conversion mode to convert Windows or Mac-style newlines.

Built-in Constants

A small number of constants live in the built-in namespace. They are:

False

The false value of the `bool` type. Assignments to `False` are illegal and raise a `SyntaxError`.

True

The true value of the `bool` type. Assignments to `True` are illegal and raise a `SyntaxError`.

None

An object frequently used to represent the absence of a value, as when default arguments are not passed to a function. Assignments to `None` are illegal and raise a `SyntaxError`. `None` is the sole instance of the `NoneType` type.

NotImplemented

A special value which should be returned by the binary special methods (e.g. `__eq__()`, `__lt__()`, `__add__()`, `__rsub__()`, etc.) to indicate that the operation is not implemented with respect to the other type; may be returned by the in-place binary special methods (e.g. `__imul__()`, `__iand__()`, etc.) for the same purpose. It should not be evaluated in a boolean context. **`NotImplemented`** is the sole instance of the `types.NotImplementedType` type.

Note

When a binary (or in-place) method returns **`NotImplemented`** the interpreter will try the reflected operation on the other type (or some other fallback, depending on the operator). If all attempts return **`NotImplemented`**, the interpreter will raise an appropriate exception. Incorrectly returning **`NotImplemented`** will result in a misleading error message or the **`NotImplemented`** value being returned to Python code.

See Implementing the arithmetic operations for examples.

Caution

`NotImplemented` and **`NotImplementedError`** are not interchangeable. This constant should only be used as described above; see **`NotImplementedError`** for details on correct usage of the exception.

Changed in version 3.9: Evaluating **`NotImplemented`** in a boolean context was deprecated.

Changed in version 3.14: Evaluating **`NotImplemented`** in a boolean context now raises a **`TypeError`**. It previously evaluated to **`True`** and emitted a **`DeprecationWarning`** since Python 3.9.

Ellipsis

The same as the ellipsis literal “`...`”, an object frequently used to indicate that something is omitted. Assignment to `Ellipsis` is possible, but assignment to `...` raises a **`SyntaxError`**. `Ellipsis` is the sole instance of the **`types.EllipsisType`** type.

__debug__

This constant is true if Python was not started with an **`-O`** option. See also the **`assert`** statement.

Note

The names `None`, `False`, `True` and `__debug__` cannot be reassigned (assignments to them, even as an attribute name, raise `SyntaxError`), so they can be considered “true” constants.

Constants added by the `site` module

The `site` module (which is imported automatically during startup, except if the `-S` command-line option is given) adds several constants to the built-in namespace. They are useful for the interactive interpreter shell and should not be used in programs.

quit(*code=None*)
exit(*code=None*)

Objects that when printed, print a message like “Use quit() or Ctrl-D (i.e. EOF) to exit”, and when accessed directly in the interactive interpreter or called as functions, raise `SystemExit` with the specified exit code.

help

Object that when printed, prints the message “Type help() for interactive help, or help(object) for help about object.”, and when accessed directly in the interactive interpreter, invokes the built-in help system (see `help()`).

copyright
credits

Objects that when printed or called, print the text of copyright or credits, respectively.

license

Object that when printed, prints the message “Type license() to see the full license text”, and when called, displays the full license text in a pager-like fashion (one screen at a time).

Built-in Types

The following sections describe the standard types that are built into the interpreter.

The principal built-in types are numerics, sequences, mappings, classes, instances and exceptions.

Some collection classes are mutable. The methods that add, subtract, or rearrange their members in place, and don't return a specific item, never return the collection instance itself but `None`.

Some operations are supported by several object types; in particular, practically all objects can be compared for equality, tested for truth value, and converted to a string (with the **`repr()`** function or the slightly different **`str()`** function). The latter function is implicitly used when an object is written by the **`print()`** function.

Truth Value Testing

Any object can be tested for truth value, for use in an **`if`** or **`while`** condition or as operand of the Boolean operations below.

By default, an object is considered true unless its class defines either a **`__bool__()`** method that returns `False` or a **`__len__()`** method that returns zero, when called with the object. [[1]] If one of the methods raises an exception when called, the exception is propagated and the object does not have a truth value (for example, **`NotImplemented`**). Here are most of the built-in objects considered false:

- constants defined to be false: `None` and `False`
- zero of any numeric type: `0`, `0.0`, `0j`, `Decimal(0)`, `Fraction(0, 1)`
- empty sequences and collections: `''`, `()`, `[]`, `{}`, `set()`, `range(0)`

Operations and built-in functions that have a Boolean result always return `0` or `False` for false and `1` or `True` for true, unless otherwise stated. (Important exception: the Boolean operations `or` and `and` always return one of their operands.)

Boolean Operations — `and`, `or`, `not`

These are the Boolean operations, ordered by ascending priority:

Operation	Result	Notes
`x or y`	if *x* is true, then *x*, else *y*	(1)
`x and y`	if *x* is false, then *x*, else *y*	(2)
`not x`	if *x* is false, then `True`, else `False`	(3)

Notes:

1. This is a short-circuit operator, so it only evaluates the second argument if the first one is false.
2. This is a short-circuit operator, so it only evaluates the second argument if the first one is true.
3. `not` has a lower priority than non-Boolean operators, so `not a == b` is interpreted as `not (a == b)`, and `a == not b` is a syntax error.

Comparisons

There are eight comparison operations in Python. They all have the same priority (which is higher than that of the Boolean operations). Comparisons can be chained arbitrarily; for example, `x < y <= z` is equivalent to `x < y and y <= z`, except that *y* is evaluated only once (but in both cases *z* is not evaluated at all when `x < y` is found to be false).

This table summarizes the comparison operations:

Operation	Meaning
`<`	strictly less than
`<=`	less than or equal
`>`	strictly greater than
`>=`	greater than or equal
`==`	equal
`!=`	not equal
`is`	object identity
`is not`	negated object identity

Unless stated otherwise, objects of different types never compare equal. The `==` operator is always defined but for some object types (for example, class objects) is equivalent to `is`. The `<`, `<=`, `>` and `>=` operators are only defined where they make sense; for example, they raise a `TypeError` exception when one of the arguments is a complex number.

Non-identical instances of a class normally compare as non-equal unless the class defines the `__eq__()` method.

Instances of a class cannot be ordered with respect to other instances of the same class, or other types of object, unless the class defines enough of the methods `__lt__()`, `__le__()`, `__gt__()`, and `__ge__()` (in general, `__lt__()` and `__eq__()` are sufficient, if you want the conventional meanings of the comparison operators).

The behavior of the `is` and `is not` operators cannot be customized; also they can be applied to any two objects and never raise an exception.

Two more operations with the same syntactic priority, `in` and `not in`, are supported by types that are iterable or implement the `__contains__()` method.

Numeric Types — `int`, `float`, `complex`

There are three distinct numeric types: *integers*, *floating-point numbers*, and *complex numbers*. In addition, Booleans are a subtype of integers. Integers have unlimited precision. Floating-point numbers are usually implemented using double in C; information about the precision and internal representation of floating-point numbers for the machine on which your program is running is available in `sys.float_info`. Complex numbers have a real and imaginary part, which are each a floating-point number. To extract these parts from a complex number *z*, use `z.real` and `z.imag`. (The standard library includes the additional numeric types `fractions.Fraction`, for rationals, and `decimal.Decimal`, for floating-point numbers with user-definable precision.)

Numbers are created by numeric literals or as the result of built-in functions and operators. Unadorned integer literals (including hex, octal and binary numbers) yield integers. Numeric literals containing a decimal point or an exponent sign yield floating-point numbers. Appending `'j'` or `'J'` to a numeric literal yields an imaginary number (a complex number with a zero real part) which you can add to an integer or float to get a complex number with real and imaginary parts.

The constructors `int()`, `float()`, and `complex()` can be used to produce numbers of a specific type.

Python fully supports mixed arithmetic: when a binary arithmetic operator has operands of different numeric types, the operand with the “narrower” type is widened to that of the other, where integer is narrower than floating point. Arithmetic with complex and real operands is defined by the usual mathematical formula, for example:

```
x + complex(u, v) = complex(x + u, v)
x * complex(u, v) = complex(x * u, x * v)
```

A comparison between numbers of different types behaves as though the exact values of those numbers were being compared. [[2]]

All numeric types (except complex) support the following operations (for priorities of the operations, see Operator precedence):

Operation	**Result**	**Notes**	**Full documentation**
`x + y`	sum of *x* and *y*		
`x - y`	difference of *x* and *y*		
`x * y`	product of *x* and *y*		
`x / y`	quotient of *x* and *y*		
`x // y`	floored quotient of *x* and *y*	(1)(2)	
`x % y`	remainder of `x / y`	(2)	
`-x`	*x* negated		
`+x`	*x* unchanged		

Operation	Result	Notes	Full documentation
`abs(x)`	absolute value or magnitude of *x*		`abs()`
`int(x)`	*x* converted to integer	(3)(6)	`int()`
`float(x)`	*x* converted to floating point	(4)(6)	`float()`
`complex(re, im)`	a complex number with real part *re*, imaginary part *im*. *im* defaults to zero.	(6)	`complex()`
`c.conjugate()`	conjugate of the complex number *c*		
`divmod(x, y)`	the pair `(x // y, x % y)`	(2)	`divmod()`
`pow(x, y)`	*x* to the power *y*	(5)	`pow()`
`x ** y`	*x* to the power *y*	(5)	

Notes:

1. Also referred to as integer division. For operands of type `int`, the result has type `int`. For operands of type `float`, the result has type `float`. In general, the result is a whole integer, though the result's type is not necessarily `int`. The result is always rounded

towards minus infinity: `1//2` is `0`, `(-1)//2` is `-1`, `1//(-2)` is `-1`, and `(-1)//(-2)` is `0`.

2. Not for complex numbers. Instead convert to floats using **`abs()`** if appropriate.

3. Conversion from **`float`** to **`int`** truncates, discarding the fractional part. See functions **`math.floor()`** and **`math.ceil()`** for alternative conversions.

4. float also accepts the strings “nan” and “inf” with an optional prefix “+” or “-” for Not a Number (NaN) and positive or negative infinity.

5. Python defines `pow(0, 0)` and `0 ** 0` to be `1`, as is common for programming languages.

6. The numeric literals accepted include the digits `0` to `9` or any Unicode equivalent (code points with the `Nd` property).

   See the Unicode Standard [https://unicode.org/Public/UNIDATA/extracted/DerivedNumericType.txt] for a complete list of code points with the `Nd` property.

All **`numbers.Real`** types (**`int`** and **`float`**) also include the following operations:

Operation	Result
`math.trunc(x)`	*x* truncated to **`Integral`**
`round(x[, n])`	*x* rounded to *n* digits, rounding half to even. If *n* is omitted, it defaults to 0.
`math.floor(x)`	the greatest **`Integral`** <= *x*
`math.ceil(x)`	the least **`Integral`** >= *x*

For additional numeric operations see the `math` and `cmath` modules.

Bitwise Operations on Integer Types

Bitwise operations only make sense for integers. The result of bitwise operations is calculated as though carried out in two's complement with an infinite number of sign bits.

The priorities of the binary bitwise operations are all lower than the numeric operations and higher than the comparisons; the unary operation `~` has the same priority as the other unary numeric operations (`+` and `-`).

This table lists the bitwise operations sorted in ascending priority:

Operation	Result	Notes
`x | y`	bitwise *or* of *x* and *y*	(4)
`x ^ y`	bitwise *exclusive or* of *x* and *y*	(4)
`x & y`	bitwise *and* of *x* and *y*	(4)
`x << n`	*x* shifted left by *n* bits	(1)(2)
`x >> n`	*x* shifted right by *n* bits	(1)(3)
`~x`	the bits of *x* inverted	

Notes:

1. Negative shift counts are illegal and cause a `ValueError` to be raised.
2. A left shift by *n* bits is equivalent to multiplication by `pow(2, n)`.

3. A right shift by *n* bits is equivalent to floor division by `pow(2, n)`.
4. Performing these calculations with at least one extra sign extension bit in a finite two’s complement representation (a working bit-width of `1 + max(x.bit_length(), y.bit_length())` or more) is sufficient to get the same result as if there were an infinite number of sign bits.

Additional Methods on Integer Types

The int type implements the **`numbers.Integral`** abstract base class. In addition, it provides a few more methods:

int.bit_length()

Return the number of bits necessary to represent an integer in binary, excluding the sign and leading zeros:

```
>>> n = -37
>>> bin(n)
'-0b100101'
>>> n.bit_length()
6
```

More precisely, if `x` is nonzero, then `x.bit_length()` is the unique positive integer `k` such that `2**(k-1) <= abs(x) < 2**k`. Equivalently, when `abs(x)` is small enough to have a correctly rounded logarithm, then `k = 1 + int(log(abs(x), 2))`. If `x` is zero, then `x.bit_length()` returns `0`.

Equivalent to:

```
def bit_length(self):
    s = bin(self)       # binary representation:  bin(-37) -->
'-0b100101'
    s = s.lstrip('-0b') # remove leading zeros and minus sign
    return len(s)       # len('100101') --> 6
```

Added in version 3.1.

int.bit_count()

Return the number of ones in the binary representation of the absolute value of the integer. This is also known as the population count. Example:

```
>>> n = 19
>>> bin(n)
'0b10011'
>>> n.bit_count()
3
>>> (-n).bit_count()
3
```

Equivalent to:

```
def bit_count(self):
    return bin(self).count("1")
```

Added in version 3.10.

int.to_bytes(*length=1*, *byteorder='big'*, ***, *signed=False*)

Return an array of bytes representing an integer.

```
>>> (1024).to_bytes(2, byteorder='big')
b'\x04\x00'
>>> (1024).to_bytes(10, byteorder='big')
b'\x00\x00\x00\x00\x00\x00\x00\x00\x04\x00'
>>> (-1024).to_bytes(10, byteorder='big', signed=True)
b'\xff\xff\xff\xff\xff\xff\xff\xff\xfc\x00'
>>> x = 1000
>>> x.to_bytes((x.bit_length() + 7) // 8, byteorder='little')
b'\xe8\x03'
```

The integer is represented using *length* bytes, and defaults to 1. An **`OverflowError`** is raised if the integer is not representable with the given number of bytes.

The *byteorder* argument determines the byte order used to represent the integer, and defaults to `"big"`. If *byteorder* is `"big"`, the most significant byte is at the beginning of the byte array. If *byteorder* is `"little"`, the most significant byte is at the end of the byte array.

The *signed* argument determines whether two’s complement is used to represent the integer. If *signed* is `False` and a negative integer is given, an **`OverflowError`** is raised. The default value for *signed* is `False`.

The default values can be used to conveniently turn an integer into a single byte object:

```
>>> (65).to_bytes()
b'A'
```

However, when using the default arguments, don’t try to convert a value greater than 255 or you’ll get an **`OverflowError`**.

Equivalent to:

```
def to_bytes(n, length=1, byteorder='big', signed=False):
    if byteorder == 'little':
        order = range(length)
    elif byteorder == 'big':
        order = reversed(range(length))
    else:
        raise ValueError("byteorder must be either 'little' or
'big'")

    return bytes((n >> i*8) & 0xff for i in order)
```

Added in version 3.2.

Changed in version 3.11: Added default argument values for `length` and `byteorder`.

classmethod int.from_bytes(*bytes*, *byteorder='big'*, *, *signed=False*)

Return the integer represented by the given array of bytes.

```
>>> int.from_bytes(b'\x00\x10', byteorder='big')
16
>>> int.from_bytes(b'\x00\x10', byteorder='little')
4096
>>> int.from_bytes(b'\xfc\x00', byteorder='big', signed=True)
-1024
>>> int.from_bytes(b'\xfc\x00', byteorder='big', signed=False)
64512
>>> int.from_bytes([255, 0, 0], byteorder='big')
16711680
```

The argument *bytes* must either be a bytes-like object or an iterable producing bytes.

The *byteorder* argument determines the byte order used to represent the integer, and defaults to `"big"`. If *byteorder* is `"big"`, the most significant byte is at the beginning of the byte array. If *byteorder* is `"little"`, the most significant byte is at the end of the byte array. To request the native byte order of the host system, use `sys.byteorder` as the byte order value.

The *signed* argument indicates whether two’s complement is used to represent the integer.

Equivalent to:

```
def from_bytes(bytes, byteorder='big', signed=False):
    if byteorder == 'little':
        little_ordered = list(bytes)
    elif byteorder == 'big':
        little_ordered = list(reversed(bytes))
    else:
        raise ValueError("byteorder must be either 'little' or
'big'")

    n = sum(b << i*8 for i, b in enumerate(little_ordered))
    if signed and little_ordered and (little_ordered[-1] & 0x80):
        n -= 1 << 8*len(little_ordered)

    return n
```

Added in version 3.2.

Changed in version 3.11: Added default argument value for `byteorder`.

int.as_integer_ratio()

Return a pair of integers whose ratio is equal to the original integer and has a positive denominator. The integer ratio of integers (whole numbers) is always the integer as the numerator and `1` as the denominator.

Added in version 3.8.

int.is_integer()

Returns `True`. Exists for duck type compatibility with `float.is_integer()`.

Added in version 3.12.

Additional Methods on Float

The float type implements the **`numbers.Real`** abstract base class. float also has the following additional methods.

classmethod float.from_number(*x*)

Class method to return a floating-point number constructed from a number *x*.

If the argument is an integer or a floating-point number, a floating-point number with the same value (within Python's floating-point precision) is returned. If the argument is outside the range of a Python float, an **`OverflowError`** will be raised.

For a general Python object `x`, `float.from_number(x)` delegates to `x.__float__()`. If **`__float__()`** is not defined then it falls back to **`__index__()`**.

Added in version 3.14.

float.as_integer_ratio()

Return a pair of integers whose ratio is exactly equal to the original float. The ratio is in lowest terms and has a positive denominator. Raises **`OverflowError`** on infinities and a **`ValueError`** on NaNs.

float.is_integer()

Return `True` if the float instance is finite with integral value, and `False` otherwise:

```
>>> (-2.0).is_integer()
True
>>> (3.2).is_integer()
False
```

Two methods support conversion to and from hexadecimal strings. Since Python's floats are stored internally as binary numbers, converting a float to or from a *decimal* string usually involves a small rounding error. In contrast, hexadecimal strings allow exact representation and specification of floating-point numbers. This can be useful when debugging, and in numerical work.

float.hex()

Return a representation of a floating-point number as a hexadecimal string. For finite floating-point numbers, this representation will always include a leading `0x` and a trailing `p` and exponent.

classmethod float.fromhex(*s*)

Class method to return the float represented by a hexadecimal string *s*. The string *s* may have leading and trailing whitespace.

Note that **`float.hex()`** is an instance method, while **`float.fromhex()`** is a class method.

A hexadecimal string takes the form:

```
[sign] ['0x'] integer ['.' fraction] ['p' exponent]
```

where the optional `sign` may by either `+` or `-`, `integer` and `fraction` are strings of hexadecimal digits, and `exponent` is a decimal integer with an optional leading sign. Case is not significant, and there must be at least one hexadecimal digit in either the integer or the fraction. This syntax is similar to the syntax specified in section 6.4.4.2 of the C99 standard, and also to the syntax used in Java 1.5 onwards. In particular, the output of **`float.hex()`** is usable as a hexadecimal floating-point literal in C or Java code, and hexadecimal strings produced by C's `%a` format character or Java's `Double.toHexString` are accepted by **`float.fromhex()`**.

Note that the exponent is written in decimal rather than hexadecimal, and that it gives the power of 2 by which to multiply the coefficient. For example, the hexadecimal string `0x3.a7p10` represents the floating-point number `(3 + 10./16 + 7./16**2) * 2.0**10`, or `3740.0`:

```
>>> float.fromhex('0x3.a7p10')
3740.0
```

Applying the reverse conversion to `3740.0` gives a different hexadecimal string representing the same number:

```
>>> float.hex(3740.0)
'0x1.d380000000000p+11'
```

Additional Methods on Complex

The `complex` type implements the `numbers.Complex` abstract base class. `complex` also has the following additional methods.

classmethod complex.from_number(*x*)

Class method to convert a number to a complex number.

For a general Python object `x`, `complex.from_number(x)` delegates to `x.__complex__()`. If `__complex__()` is not defined then it falls back to `__float__()`. If `__float__()` is not defined then it falls back to `__index__()`.

Added in version 3.14.

Hashing of numeric types

For numbers `x` and `y`, possibly of different types, it's a requirement that `hash(x) == hash(y)` whenever `x == y` (see the `__hash__()` method documentation for more details). For ease of implementation and efficiency across a variety of numeric types (including `int`, `float`, `decimal.Decimal` and `fractions.Fraction`) Python's hash for numeric types is based on a single mathematical function that's defined for any rational number, and hence applies to all instances of `int` and `fractions.Fraction`, and all finite instances of `float` and `decimal.Decimal`. Essentially, this function is given by reduction modulo `P` for a fixed prime `P`. The value of `P` is made available to Python as the `modulus` attribute of `sys.hash_info`.

CPython implementation detail: Currently, the prime used is `P = 2**31 - 1` on machines with 32-bit C longs and `P = 2**61 - 1` on machines with 64-bit C longs.

Here are the rules in detail:

- If `x = m / n` is a nonnegative rational number and `n` is not divisible by `P`, define `hash(x)` as `m * invmod(n, P) % P`, where `invmod(n, P)` gives the inverse of `n` modulo `P`.
- If `x = m / n` is a nonnegative rational number and `n` is divisible by `P` (but `m` is not) then `n` has no inverse modulo `P` and the rule

above doesn’t apply; in this case define `hash(x)` to be the constant value `sys.hash_info.inf`.
- If `x = m / n` is a negative rational number define `hash(x)` as `-hash(-x)`. If the resulting hash is `-1`, replace it with `-2`.
- The particular values `sys.hash_info.inf` and `-sys.hash_info.inf` are used as hash values for positive infinity or negative infinity (respectively).
- For a **`complex`** number `z`, the hash values of the real and imaginary parts are combined by computing `hash(z.real) + sys.hash_info.imag * hash(z.imag)`, reduced modulo `2**sys.hash_info.width` so that it lies in `range(-2**(sys.hash_info.width - 1), 2**(sys.hash_info.width - 1))`. Again, if the result is `-1`, it’s replaced with `-2`.

To clarify the above rules, here’s some example Python code, equivalent to the built-in hash, for computing the hash of a rational number, **`float`**, or **`complex`**:

```
import sys, math

def hash_fraction(m, n):
    """Compute the hash of a rational number m / n.

    Assumes m and n are integers, with n positive.
    Equivalent to hash(fractions.Fraction(m, n)).

    """
    P = sys.hash_info.modulus
    # Remove common factors of P.  (Unnecessary if m and n already
coprime.)
    while m % P == n % P == 0:
        m, n = m // P, n // P

    if n % P == 0:
        hash_value = sys.hash_info.inf
    else:
        # Fermat's Little Theorem: pow(n, P-1, P) is 1, so
        # pow(n, P-2, P) gives the inverse of n modulo P.
        hash_value = (abs(m) % P) * pow(n, P - 2, P) % P
    if m < 0:
        hash_value = -hash_value
    if hash_value == -1:
        hash_value = -2
    return hash_value
```

```
def hash_float(x):
    """Compute the hash of a float x."""

    if math.isnan(x):
        return object.__hash__(x)
    elif math.isinf(x):
        return sys.hash_info.inf if x > 0 else -sys.hash_info.inf
    else:
        return hash_fraction(*x.as_integer_ratio())

def hash_complex(z):
    """Compute the hash of a complex number z."""

    hash_value = hash_float(z.real) + sys.hash_info.imag *
hash_float(z.imag)
    # do a signed reduction modulo 2**sys.hash_info.width
    M = 2**(sys.hash_info.width - 1)
    hash_value = (hash_value & (M - 1)) - (hash_value & M)
    if hash_value == -1:
        hash_value = -2
    return hash_value
```

Boolean Type - `bool`

Booleans represent truth values. The **`bool`** type has exactly two constant instances: `True` and `False`.

The built-in function **`bool()`** converts any value to a boolean, if the value can be interpreted as a truth value (see section Truth Value Testing above).

For logical operations, use the boolean operators `and`, `or` and `not`. When applying the bitwise operators `&`, `|`, `^` to two booleans, they return a bool equivalent to the logical operations “and”, “or”, “xor”. However, the logical operators `and`, `or` and `!=` should be preferred over `&`, `|` and `^`.

Deprecated since version 3.12: The use of the bitwise inversion operator `~` is deprecated and will raise an error in Python 3.16.

`bool` is a subclass of **`int`** (see Numeric Types — int, float, complex). In many numeric contexts, `False` and `True` behave like the integers 0 and 1, respectively. However, relying on this is discouraged; explicitly convert using **`int()`** instead.

Iterator Types

Python supports a concept of iteration over containers. This is implemented using two distinct methods; these are used to allow user-defined classes to support iteration. Sequences, described below in more detail, always support the iteration methods.

One method needs to be defined for container objects to provide iterable support:

container.__iter__()

Return an iterator object. The object is required to support the iterator protocol described below. If a container supports different types of iteration, additional methods can be provided to specifically request iterators for those iteration types. (An example of an object supporting multiple forms of iteration would be a tree structure which supports both breadth-first and depth-first traversal.) This method corresponds to the **`tp_iter`** slot of the type structure for Python objects in the Python/C API.

The iterator objects themselves are required to support the following two methods, which together form the *iterator protocol*:

iterator.__iter__()

Return the iterator object itself. This is required to allow both containers and iterators to be used with the **`for`** and **`in`** statements. This method corresponds to the **`tp_iter`** slot of the type structure for Python objects in the Python/C API.

iterator.__next__()

Return the next item from the iterator. If there are no further items, raise the **`StopIteration`** exception. This method corresponds to the **`tp_iternext`** slot of the type structure for Python objects in the Python/C API.

Python defines several iterator objects to support iteration over general and specific sequence types, dictionaries, and other more specialized forms. The specific types are not important beyond their implementation of the iterator protocol.

Once an iterator’s `__next__()` method raises `StopIteration`, it must continue to do so on subsequent calls. Implementations that do not obey this property are deemed broken.

Generator Types

Python’s generators provide a convenient way to implement the iterator protocol. If a container object’s `__iter__()` method is implemented as a generator, it will automatically return an iterator object (technically, a generator object) supplying the `__iter__()` and `__next__()` methods. More information about generators can be found in the documentation for the yield expression.

Sequence Types — `list`, `tuple`, `range`

There are three basic sequence types: lists, tuples, and range objects. Additional sequence types tailored for processing of binary data and text strings are described in dedicated sections.

Common Sequence Operations

The operations in the following table are supported by most sequence types, both mutable and immutable. The `collections.abc.Sequence` ABC is provided to make it easier to correctly implement these operations on custom sequence types.

This table lists the sequence operations sorted in ascending priority. In the table, *s* and *t* are sequences of the same type, *n*, *i*, *j* and *k* are integers and *x* is an arbitrary object that meets any type and value restrictions imposed by *s*.

The `in` and `not in` operations have the same priorities as the comparison operations. The `+` (concatenation) and `*` (repetition) operations have the same priority as the corresponding numeric operations. [[3]]

Operation	**Result**	**Notes**
`x in s`	`True` if an item of *s* is equal to *x*, else `False`	(1)

Operation	**Result**	**Notes**
`x not in s`	`False` if an item of *s* is equal to *x*, else `True`	(1)
`s + t`	the concatenation of *s* and *t*	(6)(7)
`s * n` or `n * s`	equivalent to adding *s* to itself *n* times	(2)(7)
`s[i]`	*i*th item of *s*, origin 0	(3)(8)
`s[i:j]`	slice of *s* from *i* to *j*	(3)(4)
`s[i:j:k]`	slice of *s* from *i* to *j* with step *k*	(3)(5)
`len(s)`	length of *s*	
`min(s)`	smallest item of *s*	
`max(s)`	largest item of *s*	

Sequences of the same type also support comparisons. In particular, tuples and lists are compared lexicographically by comparing corresponding elements. This means that to compare equal, every element must compare equal and the two sequences must be of the same type and have the same length. (For full details see Comparisons in the language reference.)

Forward and reversed iterators over mutable sequences access values using an index. That index will continue to march forward (or backward) even if the underlying sequence is mutated. The iterator terminates only when an **`IndexError`** or a **`StopIteration`** is encountered (or when the index drops below zero).

Notes:

1. While the `in` and `not in` operations are used only for simple containment testing in the general case, some specialised sequences (such as **`str`**, **`bytes`** and **`bytearray`**) also use them for subsequence testing:

   ```
   >>> "gg" in "eggs"
   True
   ```

2. Values of *n* less than `0` are treated as `0` (which yields an empty sequence of the same type as *s*). Note that items in the sequence *s* are not copied; they are referenced multiple times. This often haunts new Python programmers; consider:

   ```
   >>> lists = [[]] * 3
   >>> lists
   [[], [], []]
   >>> lists[0].append(3)
   >>> lists
   [[3], [3], [3]]
   ```

   What has happened is that `[[]]` is a one-element list containing an empty list, so all three elements of `[[]] * 3` are references to this single empty list. Modifying any of the elements of `lists` modifies this single list. You can create a list of different lists this way:

   ```
   >>> lists = [[] for i in range(3)]
   >>> lists[0].append(3)
   >>> lists[1].append(5)
   >>> lists[2].append(7)
   >>> lists
   [[3], [5], [7]]
   ```

   Further explanation is available in the FAQ entry How do I create a multidimensional list?.

3. If *i* or *j* is negative, the index is relative to the end of sequence *s*: `len(s) + i` or `len(s) + j` is substituted. But note that `-0` is still `0`.

4. The slice of *s* from *i* to *j* is defined as the sequence of items with index *k* such that `i <= k < j`.

    - If *i* is omitted or `None`, use `0`.
    - If *j* is omitted or `None`, use `len(s)`.
    - If *i* or *j* is less than `-len(s)`, use `0`.
    - If *i* or *j* is greater than `len(s)`, use `len(s)`.
    - If *i* is greater than or equal to *j*, the slice is empty.

5. The slice of *s* from *i* to *j* with step *k* is defined as the sequence of items with index `x = i + n*k` such that `0 <= n < (j-i)/k`. In other words, the indices are `i`, `i+k`, `i+2*k`, `i+3*k` and so on, stopping when *j* is reached (but never including *j*). When *k* is positive, *i* and *j* are reduced to `len(s)` if they are greater. When *k* is negative, *i* and *j* are reduced to `len(s) - 1` if they are greater. If *i* or *j* are omitted or `None`, they become “end” values (which end depends on the sign of *k*). Note, *k* cannot be zero. If *k* is `None`, it is treated like `1`.

6. Concatenating immutable sequences always results in a new object. This means that building up a sequence by repeated concatenation will have a quadratic runtime cost in the total sequence length. To get a linear runtime cost, you must switch to one of the alternatives below:

    - if concatenating **`str`** objects, you can build a list and use **`str.join()`** at the end or else write to an **`io.StringIO`** instance and retrieve its value when complete
    - if concatenating **`bytes`** objects, you can similarly use **`bytes.join()`** or **`io.BytesIO`**, or you can do in-place concatenation with a **`bytearray`** object. **`bytearray`** objects are mutable and have an efficient overallocation mechanism
    - if concatenating **`tuple`** objects, extend a **`list`** instead
    - for other types, investigate the relevant class documentation

7. Some sequence types (such as **`range`**) only support item sequences that follow specific patterns, and hence don’t support sequence concatenation or repetition.

8. An **`IndexError`** is raised if *i* is outside the sequence range.

Sequence Methods

Sequence types also support the following methods:

sequence.count(*value*, /)

Return the total number of occurrences of *value* in *sequence*.

sequence.index(*value[, start[, stop]]*)

Return the index of the first occurrence of *value* in *sequence*.

Raises `ValueError` if *value* is not found in *sequence*.

The *start* or *stop* arguments allow for efficient searching of subsections of the sequence, beginning at *start* and ending at *stop*. This is roughly equivalent to `start + sequence[start:stop].index(value)`, only without copying any data.

Caution

Not all sequence types support passing the *start* and *stop* arguments.

Immutable Sequence Types

The only operation that immutable sequence types generally implement that is not also implemented by mutable sequence types is support for the `hash()` built-in.

This support allows immutable sequences, such as `tuple` instances, to be used as `dict` keys and stored in `set` and `frozenset` instances.

Attempting to hash an immutable sequence that contains unhashable values will result in `TypeError`.

Mutable Sequence Types

The operations in the following table are defined on mutable sequence types. The `collections.abc.MutableSequence` ABC is provided to make it easier to correctly implement these operations on custom sequence types.

In the table *s* is an instance of a mutable sequence type, *t* is any iterable object and *x* is an arbitrary object that meets any type and value restrictions imposed by *s* (for example, `bytearray` only accepts integers that meet the value restriction `0 <= x <= 255`).

Operation	Result	Notes
`s[i] = x`	item *i* of *s* is replaced by *x*	
`del s[i]`	removes item *i* of *s*	
`s[i:j] = t`	slice of *s* from *i* to *j* is replaced by the contents of the iterable *t*	
`del s[i:j]`	removes the elements of `s[i:j]` from the list (same as `s[i:j] = []`)	
`s[i:j:k] = t`	the elements of `s[i:j:k]` are replaced by those of *t*	(1)
`del s[i:j:k]`	removes the elements of `s[i:j:k]` from the list	
`s += t`	extends *s* with the contents of *t* (for the most part the same as `s[len(s):len(s)] = t`)	
`s *= n`	updates *s* with its contents repeated *n* times	(2)

Notes:

1. If *k* is not equal to `1`, *t* must have the same length as the slice it is replacing.

2. The value *n* is an integer, or an object implementing `__index__()`. Zero and negative values of *n* clear the sequence. Items in the sequence are not copied; they are referenced multiple times, as explained for `s * n` under Common Sequence Operations.

Mutable Sequence Methods

Mutable sequence types also support the following methods:

sequence.append(*value*, /)

Append *value* to the end of the sequence This is equivalent to writing `seq[len(seq):len(seq)] = [value]`.

sequence.clear()

Added in version 3.3.

Remove all items from *sequence*. This is equivalent to writing `del sequence[:]`.

sequence.copy()

Added in version 3.3.

Create a shallow copy of *sequence*. This is equivalent to writing `sequence[:]`.

Hint

The **`copy()`** method is not part of the `MutableSequence` ABC, but most concrete mutable sequence types provide it.

sequence.extend(*iterable*, /)

Extend *sequence* with the contents of *iterable*. For the most part, this is the same as writing `seq[len(seq):len(seq)] = iterable`.

sequence.insert(*index*, *value*, /)

Insert *value* into *sequence* at the given *index*. This is equivalent to writing `sequence[index:index] = [value]`.

sequence.pop(*index=-1*, /)

Retrieve the item at *index* and also removes it from *sequence*. By default, the last item in *sequence* is removed and returned.

sequence.remove(*value*, */*)

Remove the first item from *sequence* where `sequence[i] == value`.

Raises **`ValueError`** if *value* is not found in *sequence*.

sequence.reverse()

Reverse the items of *sequence* in place. This method maintains economy of space when reversing a large sequence. To remind users that it operates by side-effect, it returns `None`.

Lists

Lists are mutable sequences, typically used to store collections of homogeneous items (where the precise degree of similarity will vary by application).

class list(*iterable=()*, */*)

Lists may be constructed in several ways:

- Using a pair of square brackets to denote the empty list: `[]`
- Using square brackets, separating items with commas: `[a]`, `[a, b, c]`
- Using a list comprehension: `[x for x in iterable]`
- Using the type constructor: `list()` or `list(iterable)`

The constructor builds a list whose items are the same and in the same order as *iterable*'s items. *iterable* may be either a sequence, a container that supports iteration, or an iterator object. If *iterable* is already a list, a copy is made and returned, similar to `iterable[:]`. For example, `list('abc')` returns `['a', 'b', 'c']` and `list( (1, 2, 3) )` returns `[1, 2, 3]`. If no argument is given, the constructor creates a new empty list, `[]`.

Many other operations also produce lists, including the **`sorted()`** built-in.

Lists implement all of the common and mutable sequence operations. Lists also provide the following additional method:

sort(*, *key=None*, *reverse=False*)

This method sorts the list in place, using only `<` comparisons between items. Exceptions are not suppressed - if any comparison operations fail, the entire sort operation will fail (and the list will likely be left in a partially modified state).

`sort()` accepts two arguments that can only be passed by keyword (keyword-only arguments):

key specifies a function of one argument that is used to extract a comparison key from each list element (for example, `key=str.lower`). The key corresponding to each item in the list is calculated once and then used for the entire sorting process. The default value of `None` means that list items are sorted directly without calculating a separate key value.

The `functools.cmp_to_key()` utility is available to convert a 2.x style *cmp* function to a *key* function.

reverse is a boolean value. If set to `True`, then the list elements are sorted as if each comparison were reversed.

This method modifies the sequence in place for economy of space when sorting a large sequence. To remind users that it operates by side effect, it does not return the sorted sequence (use `sorted()` to explicitly request a new sorted list instance).

The `sort()` method is guaranteed to be stable. A sort is stable if it guarantees not to change the relative order of elements that compare equal — this is helpful for sorting in multiple passes (for example, sort by department, then by salary grade).

For sorting examples and a brief sorting tutorial, see Sorting Techniques.

CPython implementation detail: While a list is being sorted, the effect of attempting to mutate, or even inspect, the list is undefined. The C implementation of Python makes the list appear empty for the duration, and raises `ValueError` if it can detect that the list has been mutated during a sort.

Thread safety

Reading a single element from a **`list`** is atomic:

```
lst[i]   # list.__getitem__
```

The following methods traverse the list and use atomic reads of each item to perform their function. That means that they may return results affected by concurrent modifications:

```
item in lst
lst.index(item)
lst.count(item)
```

All of the above methods/operations are also lock-free. They do not block concurrent modifications. Other operations that hold a lock will not block these from observing intermediate states.

All other operations from here on block using the per-object lock.

Writing a single item via `lst[i] = x` is safe to call from multiple threads and will not corrupt the list.

The following operations return new objects and appear atomic to other threads:

```
lst1 + lst2    # concatenates two lists into a new list
x * lst        # repeats lst x times into a new list
lst.copy()     # returns a shallow copy of the list
```

Methods that only operate on a single elements with no shifting required are atomic:

```
lst.append(x)  # append to the end of the list, no shifting required
lst.pop()      # pop element from the end of the list, no shifting
required
```

The **`clear()`** method is also atomic. Other threads cannot observe elements being removed.

The **`sort()`** method is not atomic. Other threads cannot observe intermediate states during sorting, but the list appears empty for the duration of the sort.

The following operations may allow lock-free operations to observe intermediate states since they modify multiple elements in place:

```
lst.insert(idx, item)  # shifts elements
lst.pop(idx)           # idx not at the end of the list, shifts
elements
lst *= x               # copies elements in place
```

The `remove()` method may allow concurrent modifications since element comparison may execute arbitrary Python code (via `__eq__()`).

`extend()` is safe to call from multiple threads. However, its guarantees depend on the iterable passed to it. If it is a `list`, a `tuple`, a `set`, a `frozenset`, a `dict` or a dictionary view object (but not their subclasses), the `extend` operation is safe from concurrent modifications to the iterable. Otherwise, an iterator is created which can be concurrently modified by another thread. The same applies to inplace concatenation of a list with other iterables when using `lst += iterable`.

Similarly, assigning to a list slice with `lst[i:j] = iterable` is safe to call from multiple threads, but `iterable` is only locked when it is also a `list` (but not its subclasses).

Operations that involve multiple accesses, as well as iteration, are never atomic. For example:

```
# NOT atomic: read-modify-write
lst[i] = lst[i] + 1

# NOT atomic: check-then-act
if lst:
    item = lst.pop()

# NOT thread-safe: iteration while modifying
for item in lst:
    process(item)  # another thread may modify lst
```

Consider external synchronization when sharing `list` instances across threads. See Python support for free threading for more information.

Tuples

Tuples are immutable sequences, typically used to store collections of heterogeneous data (such as the 2-tuples produced by the **`enumerate()`** built-in). Tuples are also used for cases where an immutable sequence of homogeneous data is needed (such as allowing storage in a **`set`** or **`dict`** instance).

class tuple(*iterable=()*, */*)

Tuples may be constructed in a number of ways:

- Using a pair of parentheses to denote the empty tuple: `()`
- Using a trailing comma for a singleton tuple: `a,` or `(a,)`
- Separating items with commas: `a, b, c` or `(a, b, c)`
- Using the **`tuple()`** built-in: `tuple()` or `tuple(iterable)`

The constructor builds a tuple whose items are the same and in the same order as *iterable*'s items. *iterable* may be either a sequence, a container that supports iteration, or an iterator object. If *iterable* is already a tuple, it is returned unchanged. For example, `tuple('abc')` returns `('a', 'b', 'c')` and `tuple( [1, 2, 3] )` returns `(1, 2, 3)`. If no argument is given, the constructor creates a new empty tuple, `()`.

Note that it is actually the comma which makes a tuple, not the parentheses. The parentheses are optional, except in the empty tuple case, or when they are needed to avoid syntactic ambiguity. For example, `f(a, b, c)` is a function call with three arguments, while `f((a, b, c))` is a function call with a 3-tuple as the sole argument.

Tuples implement all of the common sequence operations.

For heterogeneous collections of data where access by name is clearer than access by index, **`collections.namedtuple()`** may be a more appropriate choice than a simple tuple object.

Ranges

The **`range`** type represents an immutable sequence of numbers and is commonly used for looping a specific number of times in **`for`** loops.

class range(*stop*, */*)
class range(*start*, *stop*, *step=1*, */*)

The arguments to the range constructor must be integers (either built-in **`int`** or any object that implements the **`__index__()`** special method). If the *step* argument is omitted, it defaults to `1`. If the *start* argument is omitted, it defaults to `0`. If *step* is zero, **`ValueError`** is raised.

For a positive *step*, the contents of a range `r` are determined by the formula `r[i] = start + step*i` where `i >= 0` and `r[i] < stop`.

For a negative *step*, the contents of the range are still determined by the formula `r[i] = start + step*i`, but the constraints are `i >= 0` and `r[i] > stop`.

A range object will be empty if `r[0]` does not meet the value constraint. Ranges do support negative indices, but these are interpreted as indexing from the end of the sequence determined by the positive indices.

Ranges containing absolute values larger than **`sys.maxsize`** are permitted but some features (such as **`len()`**) may raise **`OverflowError`**.

Range examples:

```
>>> list(range(10))
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> list(range(1, 11))
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
>>> list(range(0, 30, 5))
[0, 5, 10, 15, 20, 25]
>>> list(range(0, 10, 3))
[0, 3, 6, 9]
>>> list(range(0, -10, -1))
[0, -1, -2, -3, -4, -5, -6, -7, -8, -9]
>>> list(range(0))
[]
>>> list(range(1, 0))
[]
```

Ranges implement all of the common sequence operations except concatenation and repetition (due to the fact that range objects can only represent sequences that follow a strict pattern and repetition and concatenation will usually violate that pattern).

start

The value of the *start* parameter (or `0` if the parameter was not supplied)

stop

The value of the *stop* parameter

step

The value of the *step* parameter (or `1` if the parameter was not supplied)

The advantage of the **`range`** type over a regular **`list`** or **`tuple`** is that a **`range`** object will always take the same (small) amount of memory, no matter the size of the range it represents (as it only stores the `start`, `stop` and `step` values, calculating individual items and subranges as needed).

Range objects implement the **`collections.abc.Sequence`** ABC, and provide features such as containment tests, element index lookup, slicing and support for negative indices (see Sequence Types — list, tuple, range):

```
>>> r = range(0, 20, 2)
>>> r
range(0, 20, 2)
>>> 11 in r
False
>>> 10 in r
True
>>> r.index(10)
5
>>> r[5]
10
>>> r[:5]
range(0, 10, 2)
>>> r[-1]
18
```

Testing range objects for equality with `==` and `!=` compares them as sequences. That is, two range objects are considered equal if they represent the same sequence of values. (Note that two range objects that compare equal might have different **`start`**, **`stop`** and **`step`** attributes, for example `range(0) == range(2, 1, 3)` or `range(0, 3, 2) == range(0, 4, 2)`.)

Changed in version 3.2: Implement the Sequence ABC. Support slicing and negative indices. Test **`int`** objects for membership in constant time instead of iterating through all items.

Changed in version 3.3: Define '==' and '!=' to compare range objects based on the sequence of values they define (instead of comparing based on object identity).

Added the **`start`**, **`stop`** and **`step`** attributes.

See also

- The linspace recipe [https://code.activestate.com/recipes/579000-equally-spaced-numbers-linspace/] shows how to implement a lazy version of range suitable for floating-point applications.

Text and Binary Sequence Type Methods Summary

The following table summarizes the text and binary sequence types methods by category.

Category	`str` **methods**		`bytes` **and** `bytearray` **methods**	
Formatting	`str.format()`			
	`str.format_map()`			
	f-strings			
	printf-style String Formatting		printf-style Bytes Formatting	
Searching and Replacing	`str.find()`	`str.rfind()`	`bytes.find()`	`bytes.rfind()`
	`str.index()`	`str.rindex()`	`bytes.index()`	`bytes.rindex()`
	`str.startswith()`		`bytes.startswith()`	
	`str.endswith()`		`bytes.endswith()`	
	`str.count()`		`bytes.count()`	
	`str.replace()`		`bytes.replace()`	

Category	`str` **methods**	`bytes` **and** `bytearray` **methods**
Splitting and Joining	`str.split()` `str.rsplit()`	`bytes.split()` `bytes.rsplit()`
	`str.splitlines()`	`bytes.splitlines()`
	`str.partition()`	`bytes.partition()`
	`str.rpartition()`	`bytes.rpartition()`
	`str.join()`	`bytes.join()`

Category	str **methods**	bytes **and** bytearray **methods**
String Classification	`str.isalpha()`	`bytes.isalpha()`
	`str.isdecimal()`	
	`str.isdigit()`	`bytes.isdigit()`
	`str.isnumeric()`	
	`str.isalnum()`	`bytes.isalnum()`
	`str.isidentifier()`	
	`str.islower()`	`bytes.islower()`
	`str.isupper()`	`bytes.isupper()`
	`str.istitle()`	`bytes.istitle()`
	`str.isspace()`	`bytes.isspace()`
	`str.isprintable()`	

Category	`str` **methods**	`bytes` **and** `bytearray` **methods**
Case Manipulation	`str.lower()`	`bytes.lower()`
	`str.upper()`	`bytes.upper()`
	`str.casefold()`	
	`str.capitalize()`	`bytes.capitalize()`
	`str.title()`	`bytes.title()`
	`str.swapcase()`	`bytes.swapcase()`
Padding and Stripping	`str.ljust()` `str.rjust()`	`bytes.ljust()` `bytes.rjust()`
	`str.center()`	`bytes.center()`
	`str.expandtabs()`	`bytes.expandtabs()`
	`str.strip()`	`bytes.strip()`
	`str.lstrip()` `str.rstrip()`	`bytes.lstrip()` `bytes.rstrip()`

Category	`str` **methods**	`bytes` **and** `bytearray` **methods**
Translation and Encoding	`str.translate()`	`bytes.translate()`
	`str.maketrans()`	`bytes.maketrans()`
	`str.encode()`	
		`bytes.decode()`

Text Sequence Type — `str`

Textual data in Python is handled with `str` objects, or *strings*. Strings are immutable sequences of Unicode code points. String literals are written in a variety of ways:

- Single quotes: `'allows embedded "double" quotes'`
- Double quotes: `"allows embedded 'single' quotes"`
- Triple quoted: `'''Three single quotes'''`, `"""Three double quotes"""`

Triple quoted strings may span multiple lines - all associated whitespace will be included in the string literal.

String literals that are part of a single expression and have only whitespace between them will be implicitly converted to a single string literal. That is, `("spam " "eggs") == "spam eggs"`.

See String and Bytes literals for more about the various forms of string literal, including supported escape sequences, and the `r` (“raw”) prefix that disables most escape sequence processing.

Strings may also be created from other objects using the `str` constructor.

Since there is no separate “character” type, indexing a string produces strings of length 1. That is, for a non-empty string *s*, `s[0] == s[0:1]`.

There is also no mutable string type, but **`str.join()`** or **`io.StringIO`** can be used to efficiently construct strings from multiple fragments.

Changed in version 3.3: For backwards compatibility with the Python 2 series, the `u` prefix is once again permitted on string literals. It has no effect on the meaning of string literals and cannot be combined with the `r` prefix.

class str(*, *encoding='utf-8'*, *errors='strict'*)
class str(*object*)
class str(*object*, *encoding*, *errors='strict'*)
class str(*object*, *, *errors*)

Return a string version of *object*. If *object* is not provided, returns the empty string. Otherwise, the behavior of `str()` depends on whether *encoding* or *errors* is given, as follows.

If neither *encoding* nor *errors* is given, `str(object)` returns **`type(object).__str__(object)`**, which is the “informal” or nicely printable string representation of *object*. For string objects, this is the string itself. If *object* does not have a **`__str__()`** method, then **`str()`** falls back to returning **`repr(object)`**.

If at least one of *encoding* or *errors* is given, *object* should be a bytes-like object (e.g. **`bytes`** or **`bytearray`**). In this case, if *object* is a **`bytes`** (or **`bytearray`**) object, then `str(bytes, encoding, errors)` is equivalent to **`bytes.decode(encoding, errors)`**. Otherwise, the bytes object underlying the buffer object is obtained before calling **`bytes.decode()`**. See Binary Sequence Types — bytes, bytearray, memoryview and Buffer Protocol for information on buffer objects.

Passing a **`bytes`** object to **`str()`** without the *encoding* or *errors* arguments falls under the first case of returning the informal string representation (see also the **`-b`** command-line option to Python). For example:

```
>>> str(b'Zoot!')
"b'Zoot!'"
```

For more information on the `str` class and its methods, see Text Sequence Type — str and the String Methods section below. To output formatted strings, see the f-strings and Format String Syntax sections. In addition, see the Text Processing Services section.

String Methods

Strings implement all of the common sequence operations, along with the additional methods described below.

Strings also support two styles of string formatting, one providing a large degree of flexibility and customization (see **`str.format()`**, Format String Syntax and Custom String Formatting) and the other based on C `printf` style formatting that handles a narrower range of types and is slightly harder to use correctly, but is often faster for the cases it can handle (printf-style String Formatting).

The Text Processing Services section of the standard library covers a number of other modules that provide various text related utilities (including regular expression support in the **`re`** module).

str.capitalize()

Return a copy of the string with its first character capitalized and the rest lowercased.

Changed in version 3.8: The first character is now put into titlecase rather than uppercase. This means that characters like digraphs will only have their first letter capitalized, instead of the full character.

str.casefold()

Return a casefolded copy of the string. Casefolded strings may be used for caseless matching.

Casefolding is similar to lowercasing but more aggressive because it is intended to remove all case distinctions in a string. For example, the German lowercase letter `'ß'` is equivalent to `"ss"`. Since it is already lowercase, **`lower()`** would do nothing to `'ß'`; **`casefold()`** converts it to `"ss"`. For example:

```
>>> 'straße'.lower()
'straße'
>>> 'straße'.casefold()
'strasse'
```

The casefolding algorithm is described in section 3.13 ‘Default Case Folding’ of the Unicode Standard [https://www.unicode.org/versions/Unicode16.0.0/core-spec/chapter-3/#G33992].

Added in version 3.3.

str.center(*width*, *fillchar*=' ', */*)

Return centered in a string of length *width*. Padding is done using the specified *fillchar* (default is an ASCII space). The original string is returned if *width* is less than or equal to `len(s)`. For example:

```
>>> 'Python'.center(10)
'  Python  '
>>> 'Python'.center(10, '-')
'--Python--'
>>> 'Python'.center(4)
'Python'
```

str.count(*sub*[, *start*[, *end*]])

Return the number of non-overlapping occurrences of substring *sub* in the range [*start*, *end*]. Optional arguments *start* and *end* are interpreted as in slice notation.

If *sub* is empty, returns the number of empty strings between characters which is the length of the string plus one. For example:

```
>>> 'spam, spam, spam'.count('spam')
3
>>> 'spam, spam, spam'.count('spam', 5)
2
>>> 'spam, spam, spam'.count('spam', 5, 10)
1
>>> 'spam, spam, spam'.count('eggs')
0
```

```
>>> 'spam, spam, spam'.count('')
17
```

str.encode(*encoding='utf-8'*, *errors='strict'*)

Return the string encoded to **`bytes`**.

encoding defaults to `'utf-8'`; see Standard Encodings for possible values.

errors controls how encoding errors are handled. If `'strict'` (the default), a **`UnicodeError`** exception is raised. Other possible values are `'ignore'`, `'replace'`, `'xmlcharrefreplace'`, `'backslashreplace'` and any other name registered via **`codecs.register_error()`**. See Error Handlers for details.

For performance reasons, the value of *errors* is not checked for validity unless an encoding error actually occurs, Python Development Mode is enabled or a debug build is used. For example:

```
>>> encoded_str_to_bytes = 'Python'.encode()
>>> type(encoded_str_to_bytes)
<class 'bytes'>
>>> encoded_str_to_bytes
b'Python'
```

Changed in version 3.1: Added support for keyword arguments.

Changed in version 3.9: The value of the *errors* argument is now checked in Python Development Mode and in debug mode.

str.endswith(*suffix*[, *start*[, *end*]])

Return `True` if the string ends with the specified *suffix*, otherwise return `False`. *suffix* can also be a tuple of suffixes to look for. With optional *start*, test beginning at that position. With optional *end*, stop comparing at that position. Using *start* and *end* is equivalent to `str[start:end].endswith(suffix)`. For example:

```
>>> 'Python'.endswith('on')
True
>>> 'a tuple of suffixes'.endswith(('at', 'in'))
False
>>> 'a tuple of suffixes'.endswith(('at', 'es'))
```

```
True
>>> 'Python is amazing'.endswith('is', 0, 9)
True
```

See also `startswith()` and `removesuffix()`.

str.expandtabs(*tabsize=8*)

Return a copy of the string where all tab characters are replaced by one or more spaces, depending on the current column and the given tab size. Tab positions occur every *tabsize* characters (default is 8, giving tab positions at columns 0, 8, 16 and so on). To expand the string, the current column is set to zero and the string is examined character by character. If the character is a tab (`\t`), one or more space characters are inserted in the result until the current column is equal to the next tab position. (The tab character itself is not copied.) If the character is a newline (`\n`) or return (`\r`), it is copied and the current column is reset to zero. Any other character is copied unchanged and the current column is incremented by one regardless of how the character is represented when printed. For example:

```
>>> '01\t012\t0123\t01234'.expandtabs()
'01      012     0123    01234'
>>> '01\t012\t0123\t01234'.expandtabs(4)
'01  012 0123    01234'
>>> print('01\t012\n0123\t01234'.expandtabs(4))
01  012
0123    01234
```

str.find(*sub*[, *start*[, *end*]])

Return the lowest index in the string where substring *sub* is found within the slice `s[start:end]`. Optional arguments *start* and *end* are interpreted as in slice notation. Return `-1` if *sub* is not found. For example:

```
>>> 'spam, spam, spam'.find('sp')
0
>>> 'spam, spam, spam'.find('sp', 5)
6
```

See also `rfind()` and `index()`.

Note

The **`find()`** method should be used only if you need to know the position of *sub*. To check if *sub* is a substring or not, use the **`in`** operator:

```
>>> 'Py' in 'Python'
True
```

str.format(**args*, ***kwargs*)

Perform a string formatting operation. The string on which this method is called can contain literal text or replacement fields delimited by braces `{}`. Each replacement field contains either the numeric index of a positional argument, or the name of a keyword argument. Returns a copy of the string where each replacement field is replaced with the string value of the corresponding argument. For example:

```
>>> "The sum of 1 + 2 is {0}".format(1+2)
'The sum of 1 + 2 is 3'
>>> "The sum of {a} + {b} is {answer}".format(answer=1+2, a=1, b=2)
'The sum of 1 + 2 is 3'
>>> "{1} expects the {0} Inquisition!".format("Spanish", "Nobody")
'Nobody expects the Spanish Inquisition!'
```

See Format String Syntax for a description of the various formatting options that can be specified in format strings.

Note

When formatting a number (**`int`**, **`float`**, **`complex`**, **`decimal.Decimal`** and subclasses) with the `n` type (ex: `'{:n}'.format(1234)`), the function temporarily sets the `LC_CTYPE` locale to the `LC_NUMERIC` locale to decode `decimal_point` and `thousands_sep` fields of **`localeconv()`** if they are non-ASCII or longer than 1 byte, and the `LC_NUMERIC` locale is different than the `LC_CTYPE` locale. This temporary change affects other threads.

Changed in version 3.7: When formatting a number with the `n` type, the function sets temporarily the `LC_CTYPE` locale to the `LC_NUMERIC` locale in some cases.

str.format_map(*mapping*, */*)

Similar to `str.format(**mapping)`, except that `mapping` is used directly and not copied to a [**`dict`**](). This is useful if for example `mapping` is a dict subclass:

```
>>> class Default(dict):
...     def __missing__(self, key):
...         return key
...
>>> '{name} was born in {country}'.format_map(Default(name='Guido'))
'Guido was born in country'
```

Added in version 3.2.

str.index(*sub*[, *start*[, *end*]])

Like **`find()`**, but raise **`ValueError`** when the substring is not found. For example:

```
>>> 'spam, spam, spam'.index('eggs')
Traceback (most recent call last):
  File "<python-input-0>", line 1, in <module>
    'spam, spam, spam'.index('eggs')
    ~~~~~~~~~~~~~~~~~~~~~~~~^^^^^^^^
ValueError: substring not found
```

See also **`rindex()`**.

str.isalnum()

Return `True` if all characters in the string are alphanumeric and there is at least one character, `False` otherwise. A character `c` is alphanumeric if one of the following returns `True`: `c.isalpha()`, `c.isdecimal()`, `c.isdigit()`, or `c.isnumeric()`.

str.isalpha()

Return `True` if all characters in the string are alphabetic and there is at least one character, `False` otherwise. Alphabetic characters are those characters defined in the Unicode character database as “Letter”, i.e., those with general category property being one of “Lm”, “Lt”, “Lu”, “Ll”, or “Lo”. Note that this is different from the Alphabetic property defined in the section 4.10 ‘Letters, Alphabetic, and Ideographic’ of the Unicode Standard [https://www.unicode.org/versions/Unicode16.0.0/core-spec/chapter-4/#G91002]. For example:

```
>>> 'Letters and spaces'.isalpha()
False
>>> 'LettersOnly'.isalpha()
True
>>> 'µ'.isalpha()  # non-ASCII characters can be considered
alphabetical too
True
```

See Unicode Properties.

str.isascii()

Return `True` if the string is empty or all characters in the string are ASCII, `False` otherwise. ASCII characters have code points in the range U+0000-U+007F. For example:

```
>>> 'ASCII characters'.isascii()
True
>>> 'µ'.isascii()
False
```

Added in version 3.7.

str.isdecimal()

Return `True` if all characters in the string are decimal characters and there is at least one character, `False` otherwise. Decimal characters are those that can be used to form numbers in base 10, such as U+0660, ARABIC-INDIC DIGIT ZERO. Formally a decimal character is a character in the Unicode General Category “Nd”. For example:

```
>>> '0123456789'.isdecimal()
True
>>> '٠١٢٣٤٥٦٧٨٩'.isdecimal()  # Arabic-Indic digits zero to nine
True
>>> 'alphabetic'.isdecimal()
False
```

str.isdigit()

Return `True` if all characters in the string are digits and there is at least one character, `False` otherwise. Digits include decimal characters and digits that need special handling, such as the compatibility superscript digits. This covers digits which cannot be used to form numbers in base 10, like the Kharosthi numbers.

Formally, a digit is a character that has the property value Numeric_Type=Digit or Numeric_Type=Decimal.

str.isidentifier()

Return `True` if the string is a valid identifier according to the language definition, section Names (identifiers and keywords).

`keyword.iskeyword()` can be used to test whether string `s` is a reserved identifier, such as **`def`** and **`class`**.

Example:

```
>>> from keyword import iskeyword

>>> 'hello'.isidentifier(), iskeyword('hello')
(True, False)
>>> 'def'.isidentifier(), iskeyword('def')
(True, True)
```

str.islower()

Return `True` if all cased characters [[4]] in the string are lowercase and there is at least one cased character, `False` otherwise.

str.isnumeric()

Return `True` if all characters in the string are numeric characters, and there is at least one character, `False` otherwise. Numeric characters include digit characters, and all characters that have the Unicode numeric value property, e.g. U+2155, VULGAR FRACTION ONE FIFTH. Formally, numeric characters are those with the property value Numeric_Type=Digit, Numeric_Type=Decimal or Numeric_Type=Numeric. For example:

```
>>> '0123456789'.isnumeric()
True
>>> '٠١٢٣٤٥٦٧٨٩'.isnumeric()  # Arabic-indic digit zero to nine
True
>>> '⅕'.isnumeric()  # Vulgar fraction one fifth
True
>>> '²'.isdecimal(), '²'.isdigit(),  '²'.isnumeric()
(False, True, True)
```

See also **`isdecimal()`** and **`isdigit()`**. Numeric characters are a superset of decimal numbers.

str.isprintable()

Return `True` if all characters in the string are printable, `False` if it contains at least one non-printable character.

Here “printable” means the character is suitable for **`repr()`** to use in its output; “non-printable” means that **`repr()`** on built-in types will hex-escape the character. It has no bearing on the handling of strings written to **`sys.stdout`** or **`sys.stderr`**.

The printable characters are those which in the Unicode character database (see **`unicodedata`**) have a general category in group Letter, Mark, Number, Punctuation, or Symbol (L, M, N, P, or S); plus the ASCII space 0x20. Nonprintable characters are those in group Separator or Other (Z or C), except the ASCII space.

For example:

```
>>> ''.isprintable(), ' '.isprintable()
(True, True)
>>> '\t'.isprintable(), '\n'.isprintable()
(False, False)
```

str.isspace()

Return `True` if there are only whitespace characters in the string and there is at least one character, `False` otherwise.

A character is *whitespace* if in the Unicode character database (see **`unicodedata`**), either its general category is `Zs` (“Separator, space”), or its bidirectional class is one of `WS`, `B`, or `S`.

str.istitle()

Return `True` if the string is a titlecased string and there is at least one character, for example uppercase characters may only follow uncased characters and lowercase characters only cased ones. Return `False` otherwise.

For example:

```
>>> 'Spam, Spam, Spam'.istitle()
True
>>> 'spam, spam, spam'.istitle()
```

```
False
>>> 'SPAM, SPAM, SPAM'.istitle()
False
```

See also `title()`.

str.isupper()

Return `True` if all cased characters [[4]] in the string are uppercase and there is at least one cased character, `False` otherwise.

```
>>> 'BANANA'.isupper()
True
>>> 'banana'.isupper()
False
>>> 'baNana'.isupper()
False
>>> ' '.isupper()
False
```

str.join(*iterable*, */*)

Return a string which is the concatenation of the strings in *iterable*. A `TypeError` will be raised if there are any non-string values in *iterable*, including `bytes` objects. The separator between elements is the string providing this method. For example:

```
>>> ', '.join(['spam', 'spam', 'spam'])
'spam, spam, spam'
>>> '-'.join('Python')
'P-y-t-h-o-n'
```

See also `split()`.

str.ljust(*width*, *fillchar*=' ', */*)

Return the string left justified in a string of length *width*. Padding is done using the specified *fillchar* (default is an ASCII space). The original string is returned if *width* is less than or equal to `len(s)`.

For example:

```
>>> 'Python'.ljust(10)
'Python    '
>>> 'Python'.ljust(10, '.')
'Python....'
```

```
>>> 'Monty Python'.ljust(10, '.')
'Monty Python'
```

See also `rjust()`.

str.lower()

Return a copy of the string with all the cased characters [[4]] converted to lowercase. For example:

```
>>> 'Lower Method Example'.lower()
'lower method example'
```

The lowercasing algorithm used is described in section 3.13 ‘Default Case Folding’ of the Unicode Standard [https://www.unicode.org/versions/Unicode16.0.0/core-spec/chapter-3/#G33992].

str.lstrip(*chars=None*, */*)

Return a copy of the string with leading characters removed. The *chars* argument is a string specifying the set of characters to be removed. If omitted or `None`, the *chars* argument defaults to removing whitespace. The *chars* argument is not a prefix; rather, all combinations of its values are stripped:

```
>>> '   spacious   '.lstrip()
'spacious   '
>>> 'www.example.com'.lstrip('cmowz.')
'example.com'
```

See `str.removeprefix()` for a method that will remove a single prefix string rather than all of a set of characters. For example:

```
>>> 'Arthur: three!'.lstrip('Arthur: ')
'ee!'
>>> 'Arthur: three!'.removeprefix('Arthur: ')
'three!'
```

static str.maketrans(*dict*, */*)
static str.maketrans(*from*, *to*, *remove*='', */*)

This static method returns a translation table usable for `str.translate()`.

If there is only one argument, it must be a dictionary mapping Unicode ordinals (integers) or characters (strings of length 1) to

Unicode ordinals, strings (of arbitrary lengths) or `None`. Character keys will then be converted to ordinals.

If there are two arguments, they must be strings of equal length, and in the resulting dictionary, each character in *from* will be mapped to the character at the same position in *to*. If there is a third argument, it must be a string, whose characters will be mapped to `None` in the result.

str.partition(*sep*, */*)

Split the string at the first occurrence of *sep*, and return a 3-tuple containing the part before the separator, the separator itself, and the part after the separator. If the separator is not found, return a 3-tuple containing the string itself, followed by two empty strings.

str.removeprefix(*prefix*, */*)

If the string starts with the *prefix* string, return `string[len(prefix):]`. Otherwise, return a copy of the original string:

```
>>> 'TestHook'.removeprefix('Test')
'Hook'
>>> 'BaseTestCase'.removeprefix('Test')
'BaseTestCase'
```

Added in version 3.9.

See also **`removesuffix()`** and **`startswith()`**.

str.removesuffix(*suffix*, */*)

If the string ends with the *suffix* string and that *suffix* is not empty, return `string[:-len(suffix)]`. Otherwise, return a copy of the original string:

```
>>> 'MiscTests'.removesuffix('Tests')
'Misc'
>>> 'TmpDirMixin'.removesuffix('Tests')
'TmpDirMixin'
```

Added in version 3.9.

See also **`removeprefix()`** and **`endswith()`**.

str.replace(*old*, *new*, /, *count=-1*)

Return a copy of the string with all occurrences of substring *old* replaced by *new*. If *count* is given, only the first *count* occurrences are replaced. If *count* is not specified or `-1`, then all occurrences are replaced. For example:

```
>>> 'spam, spam, spam'.replace('spam', 'eggs')
'eggs, eggs, eggs'
>>> 'spam, spam, spam'.replace('spam', 'eggs', 1)
'eggs, spam, spam'
```

Changed in version 3.13: count is now supported as a keyword argument.

str.rfind(*sub*[, *start*[, *end*]])

Return the highest index in the string where substring *sub* is found, such that *sub* is contained within `s[start:end]`. Optional arguments *start* and *end* are interpreted as in slice notation. Return `-1` on failure. For example:

```
>>> 'spam, spam, spam'.rfind('sp')
12
>>> 'spam, spam, spam'.rfind('sp', 0, 10)
6
```

See also **`find()`** and **`rindex()`**.

str.rindex(*sub*[, *start*[, *end*]])

Like **`rfind()`** but raises **`ValueError`** when the substring *sub* is not found.

str.rjust(*width*, *fillchar*=' ', /)

Return the string right justified in a string of length *width*. Padding is done using the specified *fillchar* (default is an ASCII space). The original string is returned if *width* is less than or equal to `len(s)`.

str.rpartition(*sep*, /)

Split the string at the last occurrence of *sep*, and return a 3-tuple containing the part before the separator, the separator itself, and the part after the separator. If the separator is not found, return a 3-tuple containing two empty strings, followed by the string itself.

str.rsplit(*sep=None*, *maxsplit=-1*)

Return a list of the words in the string, using *sep* as the delimiter string. If *maxsplit* is given, at most *maxsplit* splits are done, the *rightmost* ones. If *sep* is not specified or `None`, any whitespace string is a separator. Except for splitting from the right, **`rsplit()`** behaves like **`split()`** which is described in detail below.

str.rstrip(*chars=None*, */*)

Return a copy of the string with trailing characters removed. The *chars* argument is a string specifying the set of characters to be removed. If omitted or `None`, the *chars* argument defaults to removing whitespace. The *chars* argument is not a suffix; rather, all combinations of its values are stripped:

```
>>> '   spacious   '.rstrip()
'   spacious'
>>> 'mississippi'.rstrip('ipz')
'mississ'
```

See **`str.removesuffix()`** for a method that will remove a single suffix string rather than all of a set of characters. For example:

```
>>> 'Monty Python'.rstrip(' Python')
'M'
>>> 'Monty Python'.removesuffix(' Python')
'Monty'
```

str.split(*sep=None*, *maxsplit=-1*)

Return a list of the words in the string, using *sep* as the delimiter string. If *maxsplit* is given, at most *maxsplit* splits are done (thus, the list will have at most `maxsplit+1` elements). If *maxsplit* is not specified or `-1`, then there is no limit on the number of splits (all possible splits are made).

If *sep* is given, consecutive delimiters are not grouped together and are deemed to delimit empty strings (for example, `'1,,2'.split(',')` returns `['1', '', '2']`). The *sep* argument may consist of multiple characters as a single delimiter (to split with multiple delimiters, use `re.split()`). Splitting an empty string with a specified separator returns `['']`.

For example:

```
>>> '1,2,3'.split(',')
['1', '2', '3']
>>> '1,2,3'.split(',', maxsplit=1)
['1', '2,3']
>>> '1,2,,3,'.split(',')
['1', '2', '', '3', '']
>>> '1<>2<>3<4'.split('<>')
['1', '2', '3<4']
```

If *sep* is not specified or is `None`, a different splitting algorithm is applied: runs of consecutive whitespace are regarded as a single separator, and the result will contain no empty strings at the start or end if the string has leading or trailing whitespace. Consequently, splitting an empty string or a string consisting of just whitespace with a `None` separator returns `[]`.

For example:

```
>>> '1 2 3'.split()
['1', '2', '3']
>>> '1 2 3'.split(maxsplit=1)
['1', '2 3']
>>> '   1   2   3   '.split()
['1', '2', '3']
```

If *sep* is not specified or is `None` and *maxsplit* is `0`, only leading runs of consecutive whitespace are considered.

For example:

```
>>> "".split(None, 0)
[]
>>> "   ".split(None, 0)
[]
>>> "   foo   ".split(maxsplit=0)
['foo   ']
```

See also `join()`.

str.splitlines(*keepends=False*)

Return a list of the lines in the string, breaking at line boundaries. Line breaks are not included in the resulting list unless *keepends* is given and true.

This method splits on the following line boundaries. In particular, the boundaries are a superset of universal newlines.

Representation	**Description**
`\n`	Line Feed
`\r`	Carriage Return
`\r\n`	Carriage Return + Line Feed
`\v` or `\x0b`	Line Tabulation
`\f` or `\x0c`	Form Feed
`\x1c`	File Separator
`\x1d`	Group Separator
`\x1e`	Record Separator
`\x85`	Next Line (C1 Control Code)
`\u2028`	Line Separator
`\u2029`	Paragraph Separator

Changed in version 3.2: `\v` and `\f` added to list of line boundaries.

For example:

```
>>> 'ab c\n\nde fg\rkl\r\n'.splitlines()
['ab c', '', 'de fg', 'kl']
>>> 'ab c\n\nde fg\rkl\r\n'.splitlines(keepends=True)
['ab c\n', '\n', 'de fg\r', 'kl\r\n']
```

Unlike `split()` when a delimiter string *sep* is given, this method returns an empty list for the empty string, and a terminal line break does not result in an extra line:

```
>>> "".splitlines()
[]
>>> "One line\n".splitlines()
['One line']
```

For comparison, `split('\n')` gives:

```
>>> ''.split('\n')
['']
>>> 'Two lines\n'.split('\n')
['Two lines', '']
```

str.startswith(*prefix*[, *start*[, *end*]])

Return `True` if string starts with the *prefix*, otherwise return `False`. *prefix* can also be a tuple of prefixes to look for. With optional *start*, test string beginning at that position. With optional *end*, stop comparing string at that position.

str.strip(*chars=None*, */*)

Return a copy of the string with the leading and trailing characters removed. The *chars* argument is a string specifying the set of characters to be removed. If omitted or `None`, the *chars* argument defaults to removing whitespace. The *chars* argument is not a prefix or suffix; rather, all combinations of its values are stripped:

```
>>> '   spacious   '.strip()
'spacious'
>>> 'www.example.com'.strip('cmowz.')
'example'
```

The outermost leading and trailing *chars* argument values are stripped from the string. Characters are removed from the leading end until reaching a string character that is not contained in the set of characters in *chars*. A similar action takes place on the trailing end. For example:

```
>>> comment_string = '#....... Section 3.2.1 Issue #32 .......'
>>> comment_string.strip('.#! ')
'Section 3.2.1 Issue #32'
```

str.swapcase()

Return a copy of the string with uppercase characters converted to lowercase and vice versa. Note that it is not necessarily true that `s.swapcase().swapcase() == s`.

str.title()

Return a titlecased version of the string where words start with an uppercase character and the remaining characters are lowercase.

For example:

```
>>> 'Hello world'.title()
'Hello World'
```

The algorithm uses a simple language-independent definition of a word as groups of consecutive letters. The definition works in many contexts but it means that apostrophes in contractions and possessives form word boundaries, which may not be the desired result:

```
>>> "they're bill's friends from the UK".title()
"They'Re Bill'S Friends From The Uk"
```

The **`string.capwords()`** function does not have this problem, as it splits words on spaces only.

Alternatively, a workaround for apostrophes can be constructed using regular expressions:

```
>>> import re
>>> def titlecase(s):
...     return re.sub(r"[A-Za-z]+('[A-Za-z]+)?",
...                   lambda mo: mo.group(0).capitalize(),
...                   s)
...
>>> titlecase("they're bill's friends.")
"They're Bill's Friends."
```

See also **`istitle()`**.

str.translate(*table*, /)

Return a copy of the string in which each character has been mapped through the given translation table. The table must be an object that implements indexing via **`__getitem__()`**, typically a mapping or sequence. When indexed by a Unicode ordinal (an integer), the table object can do any of the following: return a Unicode ordinal or a string, to map the character to one or more other characters; return `None`, to delete the character from the return string; or raise a **`LookupError`** exception, to map the character to itself.

You can use **`str.maketrans()`** to create a translation map from character-to-character mappings in different formats.

See also the **`codecs`** module for a more flexible approach to custom character mappings.

str.upper()

Return a copy of the string with all the cased characters [[4]] converted to uppercase. Note that `s.upper().isupper()` might be `False` if `s` contains uncased characters or if the Unicode category of the resulting character(s) is not “Lu” (Letter, uppercase), but e.g. “Lt” (Letter, titlecase).

The uppercasing algorithm used is described in section 3.13 ‘Default Case Folding’ of the Unicode Standard [https://www.unicode.org/versions/Unicode16.0.0/core-spec/chapter-3/#G33992].

str.zfill(*width*, /)

Return a copy of the string left filled with ASCII `'0'` digits to make a string of length *width*. A leading sign prefix (`'+'`/`'-'`) is handled by inserting the padding *after* the sign character rather than before. The original string is returned if *width* is less than or equal to `len(s)`.

For example:

```
>>> "42".zfill(5)
'00042'
>>> "-42".zfill(5)
'-0042'
```

Formatted String Literals (f-strings)

Added in version 3.6.

Changed in version 3.7: The `await` and `async for` can be used in expressions within f-strings.

Changed in version 3.8: Added the debug specifier (`=`)

Changed in version 3.12: Many restrictions on expressions within f-strings have been removed. Notably, nested strings, comments, and backslashes are now permitted.

An *f-string* (formally a *formatted string literal*) is a string literal that is prefixed with `f` or `F`. This type of string literal allows embedding the results of arbitrary Python expressions within *replacement fields*, which are delimited by curly brackets (`{}`). Each replacement field must contain an expression, optionally followed by:

- a *debug specifier* – an equal sign (`=`);
- a *conversion specifier* – `!s`, `!r` or `!a`; and/or
- a *format specifier* prefixed with a colon (`:`).

See the Lexical Analysis section on f-strings for details on the syntax of these fields.

Debug specifier

Added in version 3.8.

If a debug specifier – an equal sign (`=`) – appears after the replacement field expression, the resulting f-string will contain the expression’s source, the equal sign, and the value of the expression. This is often useful for debugging:

```
>>> number = 14.3
>>> f'{number=}'
'number=14.3'
```

Whitespace before, inside and after the expression, as well as whitespace after the equal sign, is significant — it is retained in the result:

```
>>> f'{ number  -  4  = }'
' number  -  4  = 10.3'
```

Conversion specifier

By default, the value of a replacement field expression is converted to a string using `str()`:

```
>>> from fractions import Fraction
>>> one_third = Fraction(1, 3)
>>> f'{one_third}'
'1/3'
```

When a debug specifier but no format specifier is used, the default conversion instead uses `repr()`:

```
>>> f'{one_third = }'
'one_third = Fraction(1, 3)'
```

The conversion can be specified explicitly using one of these specifiers:

- `!s` for `str()`
- `!r` for `repr()`
- `!a` for `ascii()`

For example:

```
>>> str(one_third)
'1/3'
>>> repr(one_third)
'Fraction(1, 3)'

>>> f'{one_third!s} is {one_third!r}'
'1/3 is Fraction(1, 3)'

>>> string = "¡kočka 😸!"
>>> ascii(string)
"'\\xa1ko\\u010dka \\U0001f638!'"

>>> f'{string = !a}'
"string = '\\xa1ko\\u010dka \\U0001f638!'"
```

Format specifier

After the expression has been evaluated, and possibly converted using an explicit conversion specifier, it is formatted using the

`format()` function. If the replacement field includes a *format specifier* introduced by a colon (`:`), the specifier is passed to **`format()`** as the second argument. The result of **`format()`** is then used as the final value for the replacement field. For example:

```
>>> from fractions import Fraction
>>> one_third = Fraction(1, 3)
>>> f'{one_third:.6f}'
'0.333333'
>>> f'{one_third:_^+10}'
'___+1/3___'
>>> >>> f'{one_third!r:_^20}'
'___Fraction(1, 3)___'
>>> f'{one_third = :~>10}~'
'one_third = ~~~~~~~1/3~'
```

Template String Literals (t-strings)

An *t-string* (formally a *template string literal*) is a string literal that is prefixed with `t` or `T`.

These strings follow the same syntax and evaluation rules as formatted string literals, with for the following differences:

- Rather than evaluating to a `str` object, template string literals evaluate to a **`string.templatelib.Template`** object.
- The **`format()`** protocol is not used. Instead, the format specifier and conversions (if any) are passed to a new **`Interpolation`** object that is created for each evaluated expression. It is up to code that processes the resulting **`Template`** object to decide how to handle format specifiers and conversions.
- Format specifiers containing nested replacement fields are evaluated eagerly, prior to being passed to the **`Interpolation`** object. For instance, an interpolation of the form `{amount:.{precision}f}` will evaluate the inner expression `{precision}` to determine the value of the `format_spec` attribute. If `precision` were to be `2`, the resulting format specifier would be `'.2f'`.
- When the equals sign `'='` is provided in an interpolation expression, the text of the expression is appended to the literal string that precedes the relevant interpolation. This includes the equals sign and any surrounding whitespace. The **`Interpolation`** instance for the expression will be created as normal, except

that **`conversion`** will be set to ‘`r`’ (**`repr()`**) by default. If an explicit conversion or format specifier are provided, this will override the default behaviour.

`printf`-style String Formatting

Note

The formatting operations described here exhibit a variety of quirks that lead to a number of common errors (such as failing to display tuples and dictionaries correctly).

Using formatted string literals, the **`str.format()`** interface, or **`string.Template`** may help avoid these errors. Each of these alternatives provides their own trade-offs and benefits of simplicity, flexibility, and/or extensibility.

String objects have one unique built-in operation: the `%` operator (modulo). This is also known as the string *formatting* or *interpolation* operator. Given `format % values` (where *format* is a string), `%` conversion specifications in *format* are replaced with zero or more elements of *values*. The effect is similar to using the **`sprintf()`** function in the C language. For example:

```
>>> print('%s has %d quote types.' % ('Python', 2))
Python has 2 quote types.
```

If *format* requires a single argument, *values* may be a single non-tuple object. [[5]] Otherwise, *values* must be a tuple with exactly the number of items specified by the format string, or a single mapping object (for example, a dictionary).

A conversion specifier contains two or more characters and has the following components, which must occur in this order:

1. The `'%'` character, which marks the start of the specifier.
2. Mapping key (optional), consisting of a parenthesised sequence of characters (for example, `(somename)`).
3. Conversion flags (optional), which affect the result of some conversion types.

4. Minimum field width (optional). If specified as an `'*'` (asterisk), the actual width is read from the next element of the tuple in *values*, and the object to convert comes after the minimum field width and optional precision.
5. Precision (optional), given as a `'.'` (dot) followed by the precision. If specified as `'*'` (an asterisk), the actual precision is read from the next element of the tuple in *values*, and the value to convert comes after the precision.
6. Length modifier (optional).
7. Conversion type.

When the right argument is a dictionary (or other mapping type), then the formats in the string *must* include a parenthesised mapping key into that dictionary inserted immediately after the `'%'` character. The mapping key selects the value to be formatted from the mapping. For example:

```
>>> print('%(language)s has %(number)03d quote types.' %
...       {'language': "Python", "number": 2})
Python has 002 quote types.
```

In this case no `*` specifiers may occur in a format (since they require a sequential parameter list).

The conversion flag characters are:

Flag	Meaning
`'#'`	The value conversion will use the “alternate form” (where defined below).
`'0'`	The conversion will be zero padded for numeric values.
`'-'`	The converted value is left adjusted (overrides the `'0'` conversion if both are given).

Flag	Meaning
`' '`	(a space) A blank should be left before a positive number (or empty string) produced by a signed conversion.
`'+'`	A sign character (`'+'` or `'-'`) will precede the conversion (overrides a “space” flag).

A length modifier (`h`, `l`, or `L`) may be present, but is ignored as it is not necessary for Python – so e.g. `%ld` is identical to `%d`.

The conversion types are:

Conversion	Meaning	Notes
`'d'`	Signed integer decimal.	
`'i'`	Signed integer decimal.	
`'o'`	Signed octal value.	(1)
`'u'`	Obsolete type – it is identical to `'d'`.	(6)
`'x'`	Signed hexadecimal (lowercase).	(2)
`'X'`	Signed hexadecimal (uppercase).	(2)
`'e'`	Floating-point exponential format (lowercase).	(3)

Conversion	Meaning	Notes
`'E'`	Floating-point exponential format (uppercase).	(3)
`'f'`	Floating-point decimal format.	(3)
`'F'`	Floating-point decimal format.	(3)
`'g'`	Floating-point format. Uses lowercase exponential format if exponent is less than -4 or not less than precision, decimal format otherwise.	(4)
`'G'`	Floating-point format. Uses uppercase exponential format if exponent is less than -4 or not less than precision, decimal format otherwise.	(4)
`'c'`	Single character (accepts integer or single character string).	
`'r'`	String (converts any Python object using `repr()`).	(5)
`'s'`	String (converts any Python object using `str()`).	(5)
`'a'`	String (converts any Python object using `ascii()`).	(5)

Conversion	Meaning	Notes
`'%'`	No argument is converted, results in a `'%'` character in the result.	

Notes:

1. The alternate form causes a leading octal specifier (`'0o'`) to be inserted before the first digit.
2. The alternate form causes a leading `'0x'` or `'0X'` (depending on whether the `'x'` or `'X'` format was used) to be inserted before the first digit.
3. The alternate form causes the result to always contain a decimal point, even if no digits follow it.

   The precision determines the number of digits after the decimal point and defaults to 6.
4. The alternate form causes the result to always contain a decimal point, and trailing zeroes are not removed as they would otherwise be.

   The precision determines the number of significant digits before and after the decimal point and defaults to 6.
5. If precision is `N`, the output is truncated to `N` characters.
6. See **PEP 237** [https://peps.python.org/pep-0237/].

Since Python strings have an explicit length, `%s` conversions do not assume that `'\0'` is the end of the string.

Changed in version 3.1: `%f` conversions for numbers whose absolute value is over 1e50 are no longer replaced by `%g` conversions.

Binary Sequence Types — `bytes`, `bytearray`, `memoryview`

The core built-in types for manipulating binary data are `bytes` and `bytearray`. They are supported by `memoryview` which uses the buffer protocol to access the memory of other binary objects without needing to make a copy.

The `array` module supports efficient storage of basic data types like 32-bit integers and IEEE754 double-precision floating values.

Bytes Objects

Bytes objects are immutable sequences of single bytes. Since many major binary protocols are based on the ASCII text encoding, bytes objects offer several methods that are only valid when working with ASCII compatible data and are closely related to string objects in a variety of other ways.

class bytes(*source=b''*)
class bytes(*source*, *encoding*, *errors='strict'*)

Firstly, the syntax for bytes literals is largely the same as that for string literals, except that a `b` prefix is added:

- Single quotes: `b'still allows embedded "double" quotes'`
- Double quotes: `b"still allows embedded 'single' quotes"`
- Triple quoted: `b'''3 single quotes'''`, `b"""3 double quotes"""`

Only ASCII characters are permitted in bytes literals (regardless of the declared source code encoding). Any binary values over 127 must be entered into bytes literals using the appropriate escape sequence.

As with string literals, bytes literals may also use a `r` prefix to disable processing of escape sequences. See String and Bytes literals for more about the various forms of bytes literal, including supported escape sequences.

While bytes literals and representations are based on ASCII text, bytes objects actually behave like immutable sequences of integers, with each value in the sequence restricted such that `0 <= x < 256` (attempts to violate this restriction will trigger `ValueError`). This is done deliberately to emphasise that while many binary formats include ASCII based elements and can be usefully manipulated with some text-oriented algorithms, this is not generally the case for arbitrary binary data (blindly applying text processing algorithms to binary data formats that are not ASCII compatible will usually lead to data corruption).

In addition to the literal forms, bytes objects can be created in a number of other ways:

- A zero-filled bytes object of a specified length: `bytes(10)`
- From an iterable of integers: `bytes(range(20))`
- Copying existing binary data via the buffer protocol: `bytes(obj)`

Also see the bytes built-in.

Since 2 hexadecimal digits correspond precisely to a single byte, hexadecimal numbers are a commonly used format for describing binary data. Accordingly, the bytes type has an additional class method to read data in that format:

classmethod fromhex(*string*, */*)

This **`bytes`** class method returns a bytes object, decoding the given string object. The string must contain two hexadecimal digits per byte, with ASCII whitespace being ignored.

```
>>> bytes.fromhex('2Ef0 F1f2  ')
b'.\xf0\xf1\xf2'
```

Changed in version 3.7: **`bytes.fromhex()`** now skips all ASCII whitespace in the string, not just spaces.

Changed in version 3.14: **`bytes.fromhex()`** now accepts ASCII **`bytes`** and bytes-like objects as input.

A reverse conversion function exists to transform a bytes object into its hexadecimal representation.

hex(*, *bytes_per_sep=1*)
hex(*sep*, *bytes_per_sep=1*)

Return a string object containing two hexadecimal digits for each byte in the instance.

```
>>> b'\xf0\xf1\xf2'.hex()
'f0f1f2'
```

If you want to make the hex string easier to read, you can specify a single character separator *sep* parameter to include in the output. By default, this separator will be included between each byte. A second optional *bytes_per_sep* parameter controls the spacing. Positive values calculate the separator position from the right, negative values from the left.

```
>>> value = b'\xf0\xf1\xf2'
>>> value.hex('-')
'f0-f1-f2'
>>> value.hex('_', 2)
'f0_f1f2'
>>> b'UUDDLRLRAB'.hex(' ', -4)
'55554444 4c524c52 4142'
```

Added in version 3.5.

Changed in version 3.8: `bytes.hex()` now supports optional *sep* and *bytes_per_sep* parameters to insert separators between bytes in the hex output.

Since bytes objects are sequences of integers (akin to a tuple), for a bytes object *b*, `b[0]` will be an integer, while `b[0:1]` will be a bytes object of length 1. (This contrasts with text strings, where both indexing and slicing will produce a string of length 1)

The representation of bytes objects uses the literal format (`b'...'`) since it is often more useful than e.g. `bytes([46, 46, 46])`. You can always convert a bytes object into a list of integers using `list(b)`.

Bytearray Objects

`bytearray` objects are a mutable counterpart to `bytes` objects.

class bytearray(*source=b''*)
class bytearray(*source*, *encoding*, *errors='strict'*)

There is no dedicated literal syntax for bytearray objects, instead they are always created by calling the constructor:

- Creating an empty instance: `bytearray()`
- Creating a zero-filled instance with a given length: `bytearray(10)`
- From an iterable of integers: `bytearray(range(20))`
- Copying existing binary data via the buffer protocol: `bytearray(b'Hi!')`

As bytearray objects are mutable, they support the mutable sequence operations in addition to the common bytes and bytearray operations described in Bytes and Bytearray Operations.

Also see the bytearray built-in.

Since 2 hexadecimal digits correspond precisely to a single byte, hexadecimal numbers are a commonly used format for describing binary data. Accordingly, the bytearray type has an additional class method to read data in that format:

classmethod fromhex(*string*, */*)

This **`bytearray`** class method returns bytearray object, decoding the given string object. The string must contain two hexadecimal digits per byte, with ASCII whitespace being ignored.

```
>>> bytearray.fromhex('2Ef0 F1f2  ')
bytearray(b'.\xf0\xf1\xf2')
```

Changed in version 3.7: **`bytearray.fromhex()`** now skips all ASCII whitespace in the string, not just spaces.

Changed in version 3.14: **`bytearray.fromhex()`** now accepts ASCII **`bytes`** and bytes-like objects as input.

A reverse conversion function exists to transform a bytearray object into its hexadecimal representation.

hex(*, *bytes_per_sep=1*)
hex(*sep*, *bytes_per_sep=1*)

Return a string object containing two hexadecimal digits for each byte in the instance.

```
>>> bytearray(b'\xf0\xf1\xf2').hex()
'f0f1f2'
```

Added in version 3.5.

Changed in version 3.8: Similar to `bytes.hex()`, `bytearray.hex()` now supports optional *sep* and *bytes_per_sep* parameters to insert separators between bytes in the hex output.

resize(*size*, */*)

Resize the `bytearray` to contain *size* bytes. *size* must be greater than or equal to 0.

If the `bytearray` needs to shrink, bytes beyond *size* are truncated.

If the `bytearray` needs to grow, all new bytes, those beyond *size*, will be set to null bytes.

This is equivalent to:

```
>>> def resize(ba, size):
...     if len(ba) > size:
...         del ba[size:]
...     else:
...         ba += b'\0' * (size - len(ba))
```

Examples:

```
>>> shrink = bytearray(b'abc')
>>> shrink.resize(1)
>>> (shrink, len(shrink))
(bytearray(b'a'), 1)
>>> grow = bytearray(b'abc')
>>> grow.resize(5)
>>> (grow, len(grow))
(bytearray(b'abc\x00\x00'), 5)
```

Added in version 3.14.

Since bytearray objects are sequences of integers (akin to a list), for a bytearray object *b*, `b[0]` will be an integer, while `b[0:1]` will be a bytearray object of length 1. (This contrasts with text strings, where both indexing and slicing will produce a string of length 1)

The representation of bytearray objects uses the bytes literal format (`bytearray(b'...')`) since it is often more useful than e.g. `bytearray([46, 46, 46])`. You can always convert a bytearray object into a list of integers using `list(b)`.

Bytes and Bytearray Operations

Both bytes and bytearray objects support the common sequence operations. They interoperate not just with operands of the same type, but with any bytes-like object. Due to this flexibility, they can be freely mixed in operations without causing errors. However, the return type of the result may depend on the order of operands.

Note

The methods on bytes and bytearray objects don’t accept strings as their arguments, just as the methods on strings don’t accept bytes as their arguments. For example, you have to write:

```
a = "abc"
b = a.replace("a", "f")
```

and:

```
a = b"abc"
b = a.replace(b"a", b"f")
```

Some bytes and bytearray operations assume the use of ASCII compatible binary formats, and hence should be avoided when working with arbitrary binary data. These restrictions are covered below.

Note

Using these ASCII based operations to manipulate binary data that is not stored in an ASCII based format may lead to data corruption.

The following methods on bytes and bytearray objects can be used with arbitrary binary data.

bytes.count(*sub*[, *start*[, *end*]])

bytearray.count(*sub*[, *start*[, *end*]])

Return the number of non-overlapping occurrences of subsequence *sub* in the range [*start*, *end*]. Optional arguments *start* and *end* are interpreted as in slice notation.

The subsequence to search for may be any bytes-like object or an integer in the range 0 to 255.

If *sub* is empty, returns the number of empty slices between characters which is the length of the bytes object plus one.

Changed in version 3.3: Also accept an integer in the range 0 to 255 as the subsequence.

bytes.removeprefix(*prefix*, /)

bytearray.removeprefix(*prefix*, /)

If the binary data starts with the *prefix* string, return `bytes[len(prefix):]`. Otherwise, return a copy of the original binary data:

```
>>> b'TestHook'.removeprefix(b'Test')
b'Hook'
>>> b'BaseTestCase'.removeprefix(b'Test')
b'BaseTestCase'
```

The *prefix* may be any bytes-like object.

Note

The bytearray version of this method does *not* operate in place - it always produces a new object, even if no changes were made.

Added in version 3.9.

bytes.removesuffix(*suffix*, /)

bytearray.removesuffix(*suffix*, /)

If the binary data ends with the *suffix* string and that *suffix* is not empty, return `bytes[:-len(suffix)]`. Otherwise, return a copy of the original binary data:

```
>>> b'MiscTests'.removesuffix(b'Tests')
b'Misc'
>>> b'TmpDirMixin'.removesuffix(b'Tests')
b'TmpDirMixin'
```

The *suffix* may be any bytes-like object.

Note

The bytearray version of this method does *not* operate in place - it always produces a new object, even if no changes were made.

Added in version 3.9.

bytes.decode(*encoding='utf-8'*, *errors='strict'*)
bytearray.decode(*encoding='utf-8'*, *errors='strict'*)

Return the bytes decoded to a **`str`**.

encoding defaults to `'utf-8'`; see Standard Encodings for possible values.

errors controls how decoding errors are handled. If `'strict'` (the default), a **`UnicodeError`** exception is raised. Other possible values are `'ignore'`, `'replace'`, and any other name registered via **`codecs.register_error()`**. See Error Handlers for details.

For performance reasons, the value of *errors* is not checked for validity unless a decoding error actually occurs, Python Development Mode is enabled or a debug build is used.

Note

Passing the *encoding* argument to **`str`** allows decoding any bytes-like object directly, without needing to make a temporary **`bytes`** or **`bytearray`** object.

Changed in version 3.1: Added support for keyword arguments.

Changed in version 3.9: The value of the *errors* argument is now checked in Python Development Mode and in debug mode.

bytes.endswith(*suffix*[, *start*[, *end*]])

bytearray.endswith(*suffix*[, *start*[, *end*]])

Return `True` if the binary data ends with the specified *suffix*, otherwise return `False`. *suffix* can also be a tuple of suffixes to look for. With optional *start*, test beginning at that position. With optional *end*, stop comparing at that position.

The suffix(es) to search for may be any bytes-like object.

bytes.find(*sub*[, *start*[, *end*]])

bytearray.find(*sub*[, *start*[, *end*]])

Return the lowest index in the data where the subsequence *sub* is found, such that *sub* is contained in the slice `s[start:end]`. Optional arguments *start* and *end* are interpreted as in slice notation. Return `-1` if *sub* is not found.

The subsequence to search for may be any bytes-like object or an integer in the range 0 to 255.

Note

The **`find()`** method should be used only if you need to know the position of *sub*. To check if *sub* is a substring or not, use the **`in`** operator:

```
>>> b'Py' in b'Python'
True
```

Changed in version 3.3: Also accept an integer in the range 0 to 255 as the subsequence.

bytes.index(*sub*[, *start*[, *end*]])

bytearray.index(*sub*[, *start*[, *end*]])

Like **`find()`**, but raise **`ValueError`** when the subsequence is not found.

The subsequence to search for may be any bytes-like object or an integer in the range 0 to 255.

Changed in version 3.3: Also accept an integer in the range 0 to 255 as the subsequence.

bytes.join(*iterable*, */*)
bytearray.join(*iterable*, */*)

Return a bytes or bytearray object which is the concatenation of the binary data sequences in *iterable*. A **`TypeError`** will be raised if there are any values in *iterable* that are not bytes-like objects, including **`str`** objects. The separator between elements is the contents of the bytes or bytearray object providing this method.

static bytes.maketrans(*from*, *to*, */*)
static bytearray.maketrans(*from*, *to*, */*)

This static method returns a translation table usable for **`bytes.translate()`** that will map each character in *from* into the character at the same position in *to*; *from* and *to* must both be bytes-like objects and have the same length.

Added in version 3.1.

bytes.partition(*sep*, */*)
bytearray.partition(*sep*, */*)

Split the sequence at the first occurrence of *sep*, and return a 3-tuple containing the part before the separator, the separator itself or its bytearray copy, and the part after the separator. If the separator is not found, return a 3-tuple containing a copy of the original sequence, followed by two empty bytes or bytearray objects.

The separator to search for may be any bytes-like object.

bytes.replace(*old*, *new*, *count=-1*, */*)
bytearray.replace(*old*, *new*, *count=-1*, */*)

Return a copy of the sequence with all occurrences of subsequence *old* replaced by *new*. If the optional argument *count* is given, only the first *count* occurrences are replaced.

The subsequence to search for and its replacement may be any bytes-like object.

Note

The bytearray version of this method does *not* operate in place - it always produces a new object, even if no changes were made.

bytes.rfind(*sub*[, *start*[, *end*]])

bytearray.rfind(*sub*[, *start*[, *end*]])

Return the highest index in the sequence where the subsequence *sub* is found, such that *sub* is contained within `s[start:end]`. Optional arguments *start* and *end* are interpreted as in slice notation. Return `-1` on failure.

The subsequence to search for may be any bytes-like object or an integer in the range 0 to 255.

Changed in version 3.3: Also accept an integer in the range 0 to 255 as the subsequence.

bytes.rindex(*sub*[, *start*[, *end*]])

bytearray.rindex(*sub*[, *start*[, *end*]])

Like **`rfind()`** but raises **`ValueError`** when the subsequence *sub* is not found.

The subsequence to search for may be any bytes-like object or an integer in the range 0 to 255.

Changed in version 3.3: Also accept an integer in the range 0 to 255 as the subsequence.

bytes.rpartition(*sep*, /)

bytearray.rpartition(*sep*, /)

Split the sequence at the last occurrence of *sep*, and return a 3-tuple containing the part before the separator, the separator itself or its bytearray copy, and the part after the separator. If the separator is not found, return a 3-tuple containing two empty bytes or bytearray objects, followed by a copy of the original sequence.

The separator to search for may be any bytes-like object.

bytes.startswith(*prefix*[, *start*[, *end*]])

bytearray.startswith(*prefix*[, *start*[, *end*]])

Return `True` if the binary data starts with the specified *prefix*, otherwise return `False`. *prefix* can also be a tuple of prefixes to look for. With optional *start*, test beginning at that position. With optional *end*, stop comparing at that position.

The prefix(es) to search for may be any bytes-like object.

bytes.translate(*table*, */*, *delete*=*b*'')

bytearray.translate(*table*, */*, *delete*=*b*'')

Return a copy of the bytes or bytearray object where all bytes occurring in the optional argument *delete* are removed, and the remaining bytes have been mapped through the given translation table, which must be a bytes object of length 256.

You can use the **`bytes.maketrans()`** method to create a translation table.

Set the *table* argument to `None` for translations that only delete characters:

```
>>> b'read this short text'.translate(None, b'aeiou')
b'rd ths shrt txt'
```

Changed in version 3.6: delete is now supported as a keyword argument.

The following methods on bytes and bytearray objects have default behaviours that assume the use of ASCII compatible binary formats, but can still be used with arbitrary binary data by passing appropriate arguments. Note that all of the bytearray methods in this section do *not* operate in place, and instead produce new objects.

bytes.center(*width*, *fillbyte*=*b*' ', */*)

bytearray.center(*width*, *fillbyte*=*b*' ', */*)

Return a copy of the object centered in a sequence of length *width*. Padding is done using the specified *fillbyte* (default is an ASCII space). For **`bytes`** objects, the original sequence is returned if *width* is less than or equal to `len(s)`.

Note

The bytearray version of this method does *not* operate in place - it always produces a new object, even if no changes were made.

bytes.ljust(*width*, *fillbyte=b' '*, */*)
bytearray.ljust(*width*, *fillbyte=b' '*, */*)

Return a copy of the object left justified in a sequence of length *width*. Padding is done using the specified *fillbyte* (default is an ASCII space). For **`bytes`** objects, the original sequence is returned if *width* is less than or equal to `len(s)`.

Note

The bytearray version of this method does *not* operate in place - it always produces a new object, even if no changes were made.

bytes.lstrip(*bytes=None*, */*)
bytearray.lstrip(*bytes=None*, */*)

Return a copy of the sequence with specified leading bytes removed. The *bytes* argument is a binary sequence specifying the set of byte values to be removed. If omitted or `None`, the *bytes* argument defaults to removing ASCII whitespace. The *bytes* argument is not a prefix; rather, all combinations of its values are stripped:

```
>>> b'   spacious   '.lstrip()
b'spacious   '
>>> b'www.example.com'.lstrip(b'cmowz.')
b'example.com'
```

The binary sequence of byte values to remove may be any bytes-like object. See **`removeprefix()`** for a method that will remove a single prefix string rather than all of a set of characters. For example:

```
>>> b'Arthur: three!'.lstrip(b'Arthur: ')
b'ee!'
>>> b'Arthur: three!'.removeprefix(b'Arthur: ')
b'three!'
```

Note

The bytearray version of this method does *not* operate in place - it always produces a new object, even if no changes were made.

bytes.rjust(*width*, *fillbyte=b' '*, */*)
bytearray.rjust(*width*, *fillbyte=b' '*, */*)

Return a copy of the object right justified in a sequence of length *width*. Padding is done using the specified *fillbyte* (default is an ASCII space). For `bytes` objects, the original sequence is returned if *width* is less than or equal to `len(s)`.

Note

The bytearray version of this method does *not* operate in place - it always produces a new object, even if no changes were made.

bytes.rsplit(*sep=None*, *maxsplit=-1*)
bytearray.rsplit(*sep=None*, *maxsplit=-1*)

Split the binary sequence into subsequences of the same type, using *sep* as the delimiter string. If *maxsplit* is given, at most *maxsplit* splits are done, the *rightmost* ones. If *sep* is not specified or `None`, any subsequence consisting solely of ASCII whitespace is a separator. Except for splitting from the right, `rsplit()` behaves like `split()` which is described in detail below.

bytes.rstrip(*bytes=None*, */*)
bytearray.rstrip(*bytes=None*, */*)

Return a copy of the sequence with specified trailing bytes removed. The *bytes* argument is a binary sequence specifying the set of byte values to be removed. If omitted or `None`, the *bytes* argument defaults to removing ASCII whitespace. The *bytes* argument is not a suffix; rather, all combinations of its values are stripped:

```
>>> b'   spacious   '.rstrip()
b'   spacious'
>>> b'mississippi'.rstrip(b'ipz')
b'mississ'
```

The binary sequence of byte values to remove may be any bytes-like object. See **`removesuffix()`** for a method that will remove a single suffix string rather than all of a set of characters. For example:

```
>>> b'Monty Python'.rstrip(b' Python')
b'M'
>>> b'Monty Python'.removesuffix(b' Python')
b'Monty'
```

Note

The bytearray version of this method does *not* operate in place - it always produces a new object, even if no changes were made.

bytes.split(*sep=None*, *maxsplit=-1*)
bytearray.split(*sep=None*, *maxsplit=-1*)

Split the binary sequence into subsequences of the same type, using *sep* as the delimiter string. If *maxsplit* is given and non-negative, at most *maxsplit* splits are done (thus, the list will have at most `maxsplit+1` elements). If *maxsplit* is not specified or is `-1`, then there is no limit on the number of splits (all possible splits are made).

If *sep* is given, consecutive delimiters are not grouped together and are deemed to delimit empty subsequences (for example, `b'1,,2'.split(b',')` returns `[b'1', b'', b'2']`). The *sep* argument may consist of a multibyte sequence as a single delimiter. Splitting an empty sequence with a specified separator returns `[b'']` or `[bytearray(b'')]` depending on the type of object being split. The *sep* argument may be any bytes-like object.

For example:

```
>>> b'1,2,3'.split(b',')
[b'1', b'2', b'3']
>>> b'1,2,3'.split(b',', maxsplit=1)
[b'1', b'2,3']
>>> b'1,2,,3,'.split(b',')
[b'1', b'2', b'', b'3', b'']
>>> b'1<>2<>3<4'.split(b'<>')
[b'1', b'2', b'3<4']
```

If *sep* is not specified or is `None`, a different splitting algorithm is applied: runs of consecutive ASCII whitespace are regarded as a single separator, and the result will contain no empty strings at the start or end if the sequence has leading or trailing whitespace. Consequently, splitting an empty sequence or a sequence consisting solely of ASCII whitespace without a specified separator returns `[]`.

For example:

```
>>> b'1 2 3'.split()
[b'1', b'2', b'3']
>>> b'1 2 3'.split(maxsplit=1)
[b'1', b'2 3']
>>> b'   1   2   3   '.split()
[b'1', b'2', b'3']
```

bytes.strip(*bytes=None*, /)
bytearray.strip(*bytes=None*, /)

Return a copy of the sequence with specified leading and trailing bytes removed. The *bytes* argument is a binary sequence specifying the set of byte values to be removed. If omitted or `None`, the *bytes* argument defaults to removing ASCII whitespace. The *bytes* argument is not a prefix or suffix; rather, all combinations of its values are stripped:

```
>>> b'   spacious   '.strip()
b'spacious'
>>> b'www.example.com'.strip(b'cmowz.')
b'example'
```

The binary sequence of byte values to remove may be any bytes-like object.

Note

The bytearray version of this method does *not* operate in place - it always produces a new object, even if no changes were made.

The following methods on bytes and bytearray objects assume the use of ASCII compatible binary formats and should not be applied to arbitrary binary data. Note that all of the bytearray methods in this section do *not* operate in place, and instead produce new objects.

bytes.capitalize()

bytearray.capitalize()

Return a copy of the sequence with each byte interpreted as an ASCII character, and the first byte capitalized and the rest lowercased. Non-ASCII byte values are passed through unchanged.

Note

The bytearray version of this method does *not* operate in place - it always produces a new object, even if no changes were made.

bytes.expandtabs(*tabsize=8*)

bytearray.expandtabs(*tabsize=8*)

Return a copy of the sequence where all ASCII tab characters are replaced by one or more ASCII spaces, depending on the current column and the given tab size. Tab positions occur every *tabsize* bytes (default is 8, giving tab positions at columns 0, 8, 16 and so on). To expand the sequence, the current column is set to zero and the sequence is examined byte by byte. If the byte is an ASCII tab character (`b'\t'`), one or more space characters are inserted in the result until the current column is equal to the next tab position. (The tab character itself is not copied.) If the current byte is an ASCII newline (`b'\n'`) or carriage return (`b'\r'`), it is copied and the current column is reset to zero. Any other byte value is copied unchanged and the current column is incremented by one regardless of how the byte value is represented when printed:

```
>>> b'01\t012\t0123\t01234'.expandtabs()
b'01      012     0123    01234'
>>> b'01\t012\t0123\t01234'.expandtabs(4)
b'01  012 0123    01234'
```

Note

The bytearray version of this method does *not* operate in place - it always produces a new object, even if no changes were made.

bytes.isalnum()
bytearray.isalnum()

Return `True` if all bytes in the sequence are alphabetical ASCII characters or ASCII decimal digits and the sequence is not empty, `False` otherwise. Alphabetic ASCII characters are those byte values in the sequence `b'abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ'`. ASCII decimal digits are those byte values in the sequence `b'0123456789'`.

For example:

```
>>> b'ABCabc1'.isalnum()
True
>>> b'ABC abc1'.isalnum()
False
```

bytes.isalpha()
bytearray.isalpha()

Return `True` if all bytes in the sequence are alphabetic ASCII characters and the sequence is not empty, `False` otherwise. Alphabetic ASCII characters are those byte values in the sequence `b'abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ'`.

For example:

```
>>> b'ABCabc'.isalpha()
True
```

```
>>> b'ABCabc1'.isalpha()
False
```

bytes.isascii()
bytearray.isascii()

Return `True` if the sequence is empty or all bytes in the sequence are ASCII, `False` otherwise. ASCII bytes are in the range 0-0x7F.

Added in version 3.7.

bytes.isdigit()
bytearray.isdigit()

Return `True` if all bytes in the sequence are ASCII decimal digits and the sequence is not empty, `False` otherwise. ASCII decimal digits are those byte values in the sequence `b'0123456789'`.

For example:

```
>>> b'1234'.isdigit()
True
>>> b'1.23'.isdigit()
False
```

bytes.islower()
bytearray.islower()

Return `True` if there is at least one lowercase ASCII character in the sequence and no uppercase ASCII characters, `False` otherwise.

For example:

```
>>> b'hello world'.islower()
True
>>> b'Hello world'.islower()
False
```

Lowercase ASCII characters are those byte values in the sequence `b'abcdefghijklmnopqrstuvwxyz'`. Uppercase ASCII characters are those byte values in the sequence `b'ABCDEFGHIJKLMNOPQRSTUVWXYZ'`.

bytes.isspace()

bytearray.isspace()

Return `True` if all bytes in the sequence are ASCII whitespace and the sequence is not empty, `False` otherwise. ASCII whitespace characters are those byte values in the sequence `b' \t\n\r\x0b\f'` (space, tab, newline, carriage return, vertical tab, form feed).

bytes.istitle()
bytearray.istitle()

Return `True` if the sequence is ASCII titlecase and the sequence is not empty, `False` otherwise. See **`bytes.title()`** for more details on the definition of “titlecase”.

For example:

```
>>> b'Hello World'.istitle()
True
>>> b'Hello world'.istitle()
False
```

bytes.isupper()
bytearray.isupper()

Return `True` if there is at least one uppercase alphabetic ASCII character in the sequence and no lowercase ASCII characters, `False` otherwise.

For example:

```
>>> b'HELLO WORLD'.isupper()
True
>>> b'Hello world'.isupper()
False
```

Lowercase ASCII characters are those byte values in the sequence `b'abcdefghijklmnopqrstuvwxyz'`. Uppercase ASCII characters are those byte values in the sequence `b'ABCDEFGHIJKLMNOPQRSTUVWXYZ'`.

bytes.lower()
bytearray.lower()

Return a copy of the sequence with all the uppercase ASCII characters converted to their corresponding lowercase counterpart.

For example:

```
>>> b'Hello World'.lower()
b'hello world'
```

Lowercase ASCII characters are those byte values in the sequence `b'abcdefghijklmnopqrstuvwxyz'`. Uppercase ASCII characters are those byte values in the sequence `b'ABCDEFGHIJKLMNOPQRSTUVWXYZ'`.

Note

The bytearray version of this method does *not* operate in place - it always produces a new object, even if no changes were made.

bytes.splitlines(*keepends=False*)
bytearray.splitlines(*keepends=False*)

Return a list of the lines in the binary sequence, breaking at ASCII line boundaries. This method uses the universal newlines approach to splitting lines. Line breaks are not included in the resulting list unless *keepends* is given and true.

For example:

```
>>> b'ab c\n\nde fg\rkl\r\n'.splitlines()
[b'ab c', b'', b'de fg', b'kl']
>>> b'ab c\n\nde fg\rkl\r\n'.splitlines(keepends=True)
[b'ab c\n', b'\n', b'de fg\r', b'kl\r\n']
```

Unlike **`split()`** when a delimiter string *sep* is given, this method returns an empty list for the empty string, and a terminal line break does not result in an extra line:

```
>>> b"".split(b'\n'), b"Two lines\n".split(b'\n')
([b''], [b'Two lines', b''])
>>> b"".splitlines(), b"One line\n".splitlines()
([], [b'One line'])
```

bytes.swapcase()
bytearray.swapcase()

Return a copy of the sequence with all the lowercase ASCII characters converted to their corresponding uppercase counterpart and vice-versa.

For example:

```
>>> b'Hello World'.swapcase()
b'hELLO wORLD'
```

Lowercase ASCII characters are those byte values in the sequence `b'abcdefghijklmnopqrstuvwxyz'`. Uppercase ASCII characters are those byte values in the sequence `b'ABCDEFGHIJKLMNOPQRSTUVWXYZ'`.

Unlike `str.swapcase()`, it is always the case that `bin.swapcase().swapcase() == bin` for the binary versions. Case conversions are symmetrical in ASCII, even though that is not generally true for arbitrary Unicode code points.

Note

The bytearray version of this method does *not* operate in place - it always produces a new object, even if no changes were made.

bytes.title()
bytearray.title()

Return a titlecased version of the binary sequence where words start with an uppercase ASCII character and the remaining characters are lowercase. Uncased byte values are left unmodified.

For example:

```
>>> b'Hello world'.title()
b'Hello World'
```

Lowercase ASCII characters are those byte values in the sequence `b'abcdefghijklmnopqrstuvwxyz'`. Uppercase ASCII characters are those byte values in the sequence `b'ABCDEFGHIJKLMNOPQRSTUVWXYZ'`. All other byte values are uncased.

The algorithm uses a simple language-independent definition of a word as groups of consecutive letters. The definition works in many contexts but it means that apostrophes in contractions and possessives form word boundaries, which may not be the desired result:

```
>>> b"they're bill's friends from the UK".title()
b"They'Re Bill'S Friends From The Uk"
```

A workaround for apostrophes can be constructed using regular expressions:

```
>>> import re
>>> def titlecase(s):
...     return re.sub(rb"[A-Za-z]+('[A-Za-z]+)?",
...                   lambda mo: mo.group(0)[0:1].upper() +
...                              mo.group(0)[1:].lower(),
...                   s)
...
>>> titlecase(b"they're bill's friends.")
b"They're Bill's Friends."
```

Note

The bytearray version of this method does *not* operate in place - it always produces a new object, even if no changes were made.

bytes.upper()
bytearray.upper()

Return a copy of the sequence with all the lowercase ASCII characters converted to their corresponding uppercase counterpart.

For example:

```
>>> b'Hello World'.upper()
b'HELLO WORLD'
```

Lowercase ASCII characters are those byte values in the sequence `b'abcdefghijklmnopqrstuvwxyz'`. Uppercase ASCII characters are those byte values in the sequence `b'ABCDEFGHIJKLMNOPQRSTUVWXYZ'`.

Note

The bytearray version of this method does *not* operate in place - it always produces a new object, even if no changes were made.

bytes.zfill(*width*, /)
bytearray.zfill(*width*, /)

Return a copy of the sequence left filled with ASCII `b'0'` digits to make a sequence of length *width*. A leading sign prefix (`b'+'`/ `b'-'`) is handled by inserting the padding *after* the sign character rather than before. For **`bytes`** objects, the original sequence is returned if *width* is less than or equal to `len(seq)`.

For example:

```
>>> b"42".zfill(5)
b'00042'
>>> b"-42".zfill(5)
b'-0042'
```

Note

The bytearray version of this method does *not* operate in place - it always produces a new object, even if no changes were made.

`printf`-style Bytes Formatting

Note

The formatting operations described here exhibit a variety of quirks that lead to a number of common errors (such as failing to display tuples and dictionaries correctly). If the value being printed may be a tuple or dictionary, wrap it in a tuple.

Bytes objects (`bytes`/`bytearray`) have one unique built-in operation: the `%` operator (modulo). This is also known as the bytes *formatting* or *interpolation* operator. Given `format % values` (where *format* is a bytes object), `%` conversion specifications in *format* are replaced with zero or more elements of *values*. The effect is similar to using the **`sprintf()`** in the C language.

If *format* requires a single argument, *values* may be a single non-tuple object. [[5]] Otherwise, *values* must be a tuple with exactly the number of items specified by the format bytes object, or a single mapping object (for example, a dictionary).

A conversion specifier contains two or more characters and has the following components, which must occur in this order:

1. The `'%'` character, which marks the start of the specifier.
2. Mapping key (optional), consisting of a parenthesised sequence of characters (for example, `(somename)`).
3. Conversion flags (optional), which affect the result of some conversion types.
4. Minimum field width (optional). If specified as an `'*'` (asterisk), the actual width is read from the next element of the tuple in *values*, and the object to convert comes after the minimum field width and optional precision.
5. Precision (optional), given as a `'.'` (dot) followed by the precision. If specified as `'*'` (an asterisk), the actual precision is read from the next element of the tuple in *values*, and the value to convert comes after the precision.
6. Length modifier (optional).
7. Conversion type.

When the right argument is a dictionary (or other mapping type), then the formats in the bytes object *must* include a parenthesised mapping key into that dictionary inserted immediately after the `'%'` character. The mapping key selects the value to be formatted from the mapping. For example:

```
>>> print(b'%(language)s has %(number)03d quote types.' %
...       {b'language': b"Python", b"number": 2})
b'Python has 002 quote types.'
```

In this case no `*` specifiers may occur in a format (since they require a sequential parameter list).

The conversion flag characters are:

Flag	Meaning
`'#'`	The value conversion will use the “alternate form” (where defined below).
`'0'`	The conversion will be zero padded for numeric values.
`'-'`	The converted value is left adjusted (overrides the `'0'` conversion if both are given).
`' '`	(a space) A blank should be left before a positive number (or empty string) produced by a signed conversion.
`'+'`	A sign character (`'+'` or `'-'`) will precede the conversion (overrides a “space” flag).

A length modifier (`h`, `l`, or `L`) may be present, but is ignored as it is not necessary for Python – so e.g. `%ld` is identical to `%d`.

The conversion types are:

Conversion	Meaning	Notes
`'d'`	Signed integer decimal.	
`'i'`	Signed integer decimal.	
`'o'`	Signed octal value.	(1)
`'u'`	Obsolete type – it is identical to `'d'`.	(8)

Conversion	Meaning	Notes
`'x'`	Signed hexadecimal (lowercase).	(2)
`'X'`	Signed hexadecimal (uppercase).	(2)
`'e'`	Floating-point exponential format (lowercase).	(3)
`'E'`	Floating-point exponential format (uppercase).	(3)
`'f'`	Floating-point decimal format.	(3)
`'F'`	Floating-point decimal format.	(3)
`'g'`	Floating-point format. Uses lowercase exponential format if exponent is less than -4 or not less than precision, decimal format otherwise.	(4)
`'G'`	Floating-point format. Uses uppercase exponential format if exponent is less than -4 or not less than precision, decimal format otherwise.	(4)
`'c'`	Single byte (accepts integer or single byte objects).	
`'b'`	Bytes (any object that follows the buffer	(5)

Conversion	Meaning	Notes
	protocol or has `__bytes__()`).	
`'s'`	`'s'` is an alias for `'b'` and should only be used for Python2/3 code bases.	(6)
`'a'`	Bytes (converts any Python object using `repr(obj).encode('ascii', 'backslashreplace')`).	(5)
`'r'`	`'r'` is an alias for `'a'` and should only be used for Python2/3 code bases.	(7)
`'%'`	No argument is converted, results in a `'%'` character in the result.	

Notes:

1. The alternate form causes a leading octal specifier (`'0o'`) to be inserted before the first digit.

2. The alternate form causes a leading `'0x'` or `'0X'` (depending on whether the `'x'` or `'X'` format was used) to be inserted before the first digit.

3. The alternate form causes the result to always contain a decimal point, even if no digits follow it.

   The precision determines the number of digits after the decimal point and defaults to 6.

4. The alternate form causes the result to always contain a decimal point, and trailing zeroes are not removed as they would otherwise be.

The precision determines the number of significant digits before and after the decimal point and defaults to 6.

5. If precision is `N`, the output is truncated to `N` characters.

6. `b'%s'` is deprecated, but will not be removed during the 3.x series.

7. `b'%r'` is deprecated, but will not be removed during the 3.x series.

8. See **PEP 237** [https://peps.python.org/pep-0237/].

Note

The bytearray version of this method does *not* operate in place - it always produces a new object, even if no changes were made.

See also

PEP 461 [https://peps.python.org/pep-0461/] - Adding % formatting to bytes and bytearray

Added in version 3.5.

Memory Views

`memoryview` objects allow Python code to access the internal data of an object that supports the buffer protocol without copying.

class memoryview(*object*)

Create a **`memoryview`** that references *object*. *object* must support the buffer protocol. Built-in objects that support the buffer protocol include **`bytes`** and **`bytearray`**.

A **`memoryview`** has the notion of an *element*, which is the atomic memory unit handled by the originating *object*. For many simple types such as **`bytes`** and **`bytearray`**, an element is a single byte, but other types such as **`array.array`** may have bigger elements.

`len(view)` is equal to the length of **`tolist`**, which is the nested list representation of the view. If `view.ndim = 1`, this is equal to the number of elements in the view.

Changed in version 3.12: If `view.ndim == 0`, `len(view)` now raises **`TypeError`** instead of returning 1.

The **`itemsize`** attribute will give you the number of bytes in a single element.

A **`memoryview`** supports slicing and indexing to expose its data. One-dimensional slicing will result in a subview:

```
>>> v = memoryview(b'abcefg')
>>> v[1]
98
>>> v[-1]
103
>>> v[1:4]
<memory at 0x7f3ddc9f4350>
>>> bytes(v[1:4])
b'bce'
```

If **`format`** is one of the native format specifiers from the **`struct`** module, indexing with an integer or a tuple of integers is also supported and returns a single *element* with the correct type. One-dimensional memoryviews can be indexed with an integer or a one-integer tuple. Multi-dimensional memoryviews can be indexed with tuples of exactly *ndim* integers where *ndim* is the number of dimensions. Zero-dimensional memoryviews can be indexed with the empty tuple.

Here is an example with a non-byte format:

```
>>> import array
>>> a = array.array('l', [-11111111, 22222222, -33333333,
44444444])
>>> m = memoryview(a)
>>> m[0]
-11111111
```

```
>>> m[-1]
44444444
>>> m[::2].tolist()
[-11111111, -33333333]
```

If the underlying object is writable, the memoryview supports one-dimensional slice assignment. Resizing is not allowed:

```
>>> data = bytearray(b'abcefg')
>>> v = memoryview(data)
>>> v.readonly
False
>>> v[0] = ord(b'z')
>>> data
bytearray(b'zbcefg')
>>> v[1:4] = b'123'
>>> data
bytearray(b'z123fg')
>>> v[2:3] = b'spam'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: memoryview assignment: lvalue and rvalue have
different structures
>>> v[2:6] = b'spam'
>>> data
bytearray(b'z1spam')
```

One-dimensional memoryviews of hashable (read-only) types with formats ‘B’, ‘b’ or ‘c’ are also hashable. The hash is defined as `hash(m) == hash(m.tobytes())`:

```
>>> v = memoryview(b'abcefg')
>>> hash(v) == hash(b'abcefg')
True
>>> hash(v[2:4]) == hash(b'ce')
True
>>> hash(v[::-2]) == hash(b'abcefg'[::-2])
True
```

Changed in version 3.3: One-dimensional memoryviews can now be sliced. One-dimensional memoryviews with formats ‘B’, ‘b’ or ‘c’ are now hashable.

Changed in version 3.4: memoryview is now registered automatically with **`collections.abc.Sequence`**

Changed in version 3.5: memoryviews can now be indexed with tuple of integers.

Changed in version 3.14: memoryview is now a generic type.

`memoryview` has several methods:

__eq__(*exporter*)

A memoryview and a **PEP 3118** [https://peps.python.org/pep-3118/] exporter are equal if their shapes are equivalent and if all corresponding values are equal when the operands' respective format codes are interpreted using **`struct`** syntax.

For the subset of **`struct`** format strings currently supported by **`tolist()`**, `v` and `w` are equal if `v.tolist() == w.tolist()`:

```
>>> import array
>>> a = array.array('I', [1, 2, 3, 4, 5])
>>> b = array.array('d', [1.0, 2.0, 3.0, 4.0, 5.0])
>>> c = array.array('b', [5, 3, 1])
>>> x = memoryview(a)
>>> y = memoryview(b)
>>> x == a == y == b
True
>>> x.tolist() == a.tolist() == y.tolist() == b.tolist()
True
>>> z = y[::-2]
>>> z == c
True
>>> z.tolist() == c.tolist()
True
```

If either format string is not supported by the **`struct`** module, then the objects will always compare as unequal (even if the format strings and buffer contents are identical):

```
>>> from ctypes import BigEndianStructure, c_long
>>> class BEPoint(BigEndianStructure):
...     _fields_ = [("x", c_long), ("y", c_long)]
...
>>> point = BEPoint(100, 200)
>>> a = memoryview(point)
>>> b = memoryview(point)
>>> a == point
False
>>> a == b
False
```

Note that, as with floating-point numbers, `v is w` does *not* imply `v == w` for memoryview objects.

Changed in version 3.3: Previous versions compared the raw memory disregarding the item format and the logical array structure.

tobytes(*order='C'*)

Return the data in the buffer as a bytestring. This is equivalent to calling the **`bytes`** constructor on the memoryview.

```
>>> m = memoryview(b"abc")
>>> m.tobytes()
b'abc'
>>> bytes(m)
b'abc'
```

For non-contiguous arrays the result is equal to the flattened list representation with all elements converted to bytes. **`tobytes()`** supports all format strings, including those that are not in **`struct`** module syntax.

Added in version 3.8: order can be {‘C’, ‘F’, ‘A’}. When *order* is ‘C’ or ‘F’, the data of the original array is converted to C or Fortran order. For contiguous views, ‘A’ returns an exact copy of the physical memory. In particular, in-memory Fortran order is preserved. For non-contiguous views, the data is converted to C first. *order=None* is the same as *order=’C’*.

hex(*, *bytes_per_sep=1*)
hex(*sep*, *bytes_per_sep=1*)

Return a string object containing two hexadecimal digits for each byte in the buffer.

```
>>> m = memoryview(b"abc")
>>> m.hex()
'616263'
```

Added in version 3.5.

Changed in version 3.8: Similar to `bytes.hex()`, `memoryview.hex()` now supports optional *sep* and *bytes_per_sep* parameters to insert separators between bytes in the hex output.

tolist()

Return the data in the buffer as a list of elements.

```
>>> memoryview(b'abc').tolist()
[97, 98, 99]
>>> import array
>>> a = array.array('d', [1.1, 2.2, 3.3])
>>> m = memoryview(a)
>>> m.tolist()
[1.1, 2.2, 3.3]
```

Changed in version 3.3: `tolist()` now supports all single character native formats in `struct` module syntax as well as multi-dimensional representations.

toreadonly()

Return a readonly version of the memoryview object. The original memoryview object is unchanged.

```
>>> m = memoryview(bytearray(b'abc'))
>>> mm = m.toreadonly()
>>> mm.tolist()
[97, 98, 99]
>>> mm[0] = 42
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: cannot modify read-only memory
>>> m[0] = 43
>>> mm.tolist()
[43, 98, 99]
```

Added in version 3.8.

release()

Release the underlying buffer exposed by the memoryview object. Many objects take special actions when a view is held on them (for example, a `bytearray` would temporarily forbid resizing); therefore, calling release() is handy to remove these restrictions (and free any dangling resources) as soon as possible.

After this method has been called, any further operation on the view raises a **`ValueError`** (except **`release()`** itself which can be called multiple times):

```
>>> m = memoryview(b'abc')
>>> m.release()
>>> m[0]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: operation forbidden on released memoryview object
```

The context management protocol can be used for a similar effect, using the `with` statement:

```
>>> with memoryview(b'abc') as m:
...     m[0]
...
97
>>> m[0]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: operation forbidden on released memoryview object
```

Added in version 3.2.

cast(*format*, /)
cast(*format*, *shape*, /)

Cast a memoryview to a new format or shape. *shape* defaults to `[byte_length//new_itemsize]`, which means that the result view will be one-dimensional. The return value is a new memoryview, but the buffer itself is not copied. Supported casts are 1D -> C-contiguous and C-contiguous -> 1D.

The destination format is restricted to a single element native format in **`struct`** syntax. One of the formats must be a byte format (‘B’, ‘b’ or ‘c’). The byte length of the result must be the same as the original length. Note that all byte lengths may depend on the operating system.

Cast 1D/long to 1D/unsigned bytes:

```
>>> import array
>>> a = array.array('l', [1,2,3])
>>> x = memoryview(a)
>>> x.format
'l'
>>> x.itemsize
8
>>> len(x)
3
>>> x.nbytes
24
>>> y = x.cast('B')
>>> y.format
'B'
>>> y.itemsize
1
>>> len(y)
24
>>> y.nbytes
24
```

Cast 1D/unsigned bytes to 1D/char:

```
>>> b = bytearray(b'zyz')
>>> x = memoryview(b)
>>> x[0] = b'a'
Traceback (most recent call last):
  ...
TypeError: memoryview: invalid type for format 'B'
>>> y = x.cast('c')
>>> y[0] = b'a'
>>> b
bytearray(b'ayz')
```

Cast 1D/bytes to 3D/ints to 1D/signed char:

```
>>> import struct
>>> buf = struct.pack("i"*12, *list(range(12)))
>>> x = memoryview(buf)
>>> y = x.cast('i', shape=[2,2,3])
>>> y.tolist()
[[[0, 1, 2], [3, 4, 5]], [[6, 7, 8], [9, 10, 11]]]
>>> y.format
'i'
>>> y.itemsize
4
```

```
>>> len(y)
2
>>> y.nbytes
48
>>> z = y.cast('b')
>>> z.format
'b'
>>> z.itemsize
1
>>> len(z)
48
>>> z.nbytes
48
```

Cast 1D/unsigned long to 2D/unsigned long:

```
>>> buf = struct.pack("L"*6, *list(range(6)))
>>> x = memoryview(buf)
>>> y = x.cast('L', shape=[2,3])
>>> len(y)
2
>>> y.nbytes
48
>>> y.tolist()
[[0, 1, 2], [3, 4, 5]]
```

Added in version 3.3.

Changed in version 3.5: The source format is no longer restricted when casting to a byte view.

count(*value*, /)

Count the number of occurrences of *value*.

Added in version 3.14.

index(*value*, *start=0*, *stop=sys.maxsize*, /)

Return the index of the first occurrence of *value* (at or after index *start* and before index *stop*).

Raises a `ValueError` if *value* cannot be found.

Added in version 3.14.

There are also several readonly attributes available:

obj

The underlying object of the memoryview:

```
>>> b  = bytearray(b'xyz')
>>> m = memoryview(b)
>>> m.obj is b
True
```

Added in version 3.3.

nbytes

`nbytes == product(shape) * itemsize == len(m.tobytes())`. This is the amount of space in bytes that the array would use in a contiguous representation. It is not necessarily equal to `len(m)`:

```
>>> import array
>>> a = array.array('i', [1,2,3,4,5])
>>> m = memoryview(a)
>>> len(m)
5
>>> m.nbytes
20
>>> y = m[::2]
>>> len(y)
3
>>> y.nbytes
12
>>> len(y.tobytes())
12
```

Multi-dimensional arrays:

```
>>> import struct
>>> buf = struct.pack("d"*12, *[1.5*x for x in range(12)])
>>> x = memoryview(buf)
>>> y = x.cast('d', shape=[3,4])
>>> y.tolist()
[[0.0, 1.5, 3.0, 4.5], [6.0, 7.5, 9.0, 10.5], [12.0, 13.5,
15.0, 16.5]]
>>> len(y)
3
>>> y.nbytes
96
```

Added in version 3.3.

readonly

A bool indicating whether the memory is read only.

format

A string containing the format (in **`struct`** module style) for each element in the view. A memoryview can be created from exporters with arbitrary format strings, but some methods (e.g. **`tolist()`**) are restricted to native single element formats.

Changed in version 3.3: format `'B'` is now handled according to the struct module syntax. This means that `memoryview(b'abc')[0] == b'abc'[0] == 97`.

itemsize

The size in bytes of each element of the memoryview:

```
>>> import array, struct
>>> m = memoryview(array.array('H', [32000, 32001, 32002]))
>>> m.itemsize
2
>>> m[0]
32000
>>> struct.calcsize('H') == m.itemsize
True
```

ndim

An integer indicating how many dimensions of a multi-dimensional array the memory represents.

shape

A tuple of integers the length of **`ndim`** giving the shape of the memory as an N-dimensional array.

Changed in version 3.3: An empty tuple instead of `None` when ndim = 0.

strides

A tuple of integers the length of **`ndim`** giving the size in bytes to access each element for each dimension of the array.

Changed in version 3.3: An empty tuple instead of `None` when ndim = 0.

suboffsets

Used internally for PIL-style arrays. The value is informational only.

c_contiguous

A bool indicating whether the memory is C-contiguous.

Added in version 3.3.

f_contiguous

A bool indicating whether the memory is Fortran contiguous.

Added in version 3.3.

contiguous

A bool indicating whether the memory is contiguous.

Added in version 3.3.

Set Types — `set`, `frozenset`

A *set* object is an unordered collection of distinct hashable objects. Common uses include membership testing, removing duplicates from a sequence, and computing mathematical operations such as intersection, union, difference, and symmetric difference. (For other containers see the built-in **`dict`**, **`list`**, and **`tuple`** classes, and the **`collections`** module.)

Like other collections, sets support `x in set`, `len(set)`, and `for x in set`. Being an unordered collection, sets do not record element position or order of insertion. Accordingly, sets do not support indexing, slicing, or other sequence-like behavior.

There are currently two built-in set types, **`set`** and **`frozenset`**. The **`set`** type is mutable — the contents can be changed using methods like **`add()`** and **`remove()`**. Since it is mutable, it has no hash value and cannot be used as either a dictionary key or as an element of another set. The **`frozenset`** type is immutable and hashable — its contents cannot be altered after it is created; it can therefore be used as a dictionary key or as an element of another set.

Non-empty sets (not frozensets) can be created by placing a comma-separated list of elements within braces, for example: `{'jack', 'sjoerd'}`, in addition to the **`set`** constructor.

The constructors for both classes work the same:

class set(*iterable=()*, /)
class frozenset(*iterable=()*, /)

Return a new set or frozenset object whose elements are taken from *iterable*. The elements of a set must be hashable. To represent sets of sets, the inner sets must be **`frozenset`** objects. If *iterable* is not specified, a new empty set is returned.

Sets can be created by several means:

- Use a comma-separated list of elements within braces: `{'jack', 'sjoerd'}`
- Use a set comprehension: `{c for c in 'abracadabra' if c not in 'abc'}`
- Use the type constructor: `set()`, `set('foobar')`, `set(['a', 'b', 'foo'])`

Instances of **`set`** and **`frozenset`** provide the following operations:

len(s)

Return the number of elements in set *s* (cardinality of *s*).

x in s

Test *x* for membership in *s*.

x not in s

Test *x* for non-membership in *s*.

frozenset.isdisjoint(*other*, /)
set.isdisjoint(*other*, /)

Return `True` if the set has no elements in common with *other*. Sets are disjoint if and only if their intersection is the empty set.

frozenset.issubset(*other*, /)
set.issubset(*other*, /)

set <= other
Test whether every element in the set is in *other*.

set < other
Test whether the set is a proper subset of *other*, that is, `set <= other and set != other`.

frozenset.issuperset(*other*, /)
set.issuperset(*other*, /)

set >= other
Test whether every element in *other* is in the set.

set > other
Test whether the set is a proper superset of *other*, that is, `set >= other and set != other`.

frozenset.union(**others*)
set.union(**others*)

set | other | ...
Return a new set with elements from the set and all others.

frozenset.intersection(**others*)
set.intersection(**others*)

set & other & ...
Return a new set with elements common to the set and all others.

frozenset.difference(**others*)
set.difference(**others*)

set - other - ...
Return a new set with elements in the set that are not in the others.

frozenset.symmetric_difference(*other*, /)
set.symmetric_difference(*other*, /)

set ^ other

Return a new set with elements in either the set or *other* but not both.

frozenset.copy()
set.copy()

Return a shallow copy of the set.

Note, the non-operator versions of **`union()`**, **`intersection()`**, **`difference()`**, **`symmetric_difference()`**, **`issubset()`**, and **`issuperset()`** methods will accept any iterable as an argument. In contrast, their operator based counterparts require their arguments to be sets. This precludes error-prone constructions like `set('abc') & 'cbs'` in favor of the more readable `set('abc').intersection('cbs')`.

Both **`set`** and **`frozenset`** support set to set comparisons. Two sets are equal if and only if every element of each set is contained in the other (each is a subset of the other). A set is less than another set if and only if the first set is a proper subset of the second set (is a subset, but is not equal). A set is greater than another set if and only if the first set is a proper superset of the second set (is a superset, but is not equal).

Instances of **`set`** are compared to instances of **`frozenset`** based on their members. For example, `set('abc') == frozenset('abc')` returns `True` and so does `set('abc') in set([frozenset('abc')])`.

The subset and equality comparisons do not generalize to a total ordering function. For example, any two nonempty disjoint sets are not equal and are not subsets of each other, so *all* of the following return `False`: `a<b`, `a==b`, or `a>b`.

Since sets only define partial ordering (subset relationships), the output of the **`list.sort()`** method is undefined for lists of sets.

Set elements, like dictionary keys, must be hashable.

Binary operations that mix `set` instances with `frozenset` return the type of the first operand. For example: `frozenset('ab') | set('bc')` returns an instance of `frozenset`.

The following table lists operations available for `set` that do not apply to immutable instances of `frozenset`:

set.update(**others*)

set |= other | ...

Update the set, adding elements from all others.

set.intersection_update(**others*)

set &= other & ...

Update the set, keeping only elements found in it and all others.

set.difference_update(**others*)

set -= other | ...

Update the set, removing elements found in others.

set.symmetric_difference_update(*other*, /)

set ^= other

Update the set, keeping only elements found in either set, but not in both.

set.add(*elem*, /)

Add element *elem* to the set.

set.remove(*elem*, /)

Remove element *elem* from the set. Raises `KeyError` if *elem* is not contained in the set.

set.discard(*elem*, /)

Remove element *elem* from the set if it is present.

set.pop()

Remove and return an arbitrary element from the set. Raises `KeyError` if the set is empty.

set.clear()

Remove all elements from the set.

Note, the non-operator versions of the `update()`, `intersection_update()`, `difference_update()`, and `symmetric_difference_update()` methods will accept any iterable as an argument.

Note, the *elem* argument to the `__contains__()`, `remove()`, and `discard()` methods may be a set. To support searching for an equivalent frozenset, a temporary one is created from *elem*.

Mapping Types — `dict`

A mapping object maps hashable values to arbitrary objects. Mappings are mutable objects. There is currently only one standard mapping type, the *dictionary*. (For other containers see the built-in `list`, `set`, and `tuple` classes, and the `collections` module.)

A dictionary's keys are *almost* arbitrary values. Values that are not hashable, that is, values containing lists, dictionaries or other mutable types (that are compared by value rather than by object identity) may not be used as keys. Values that compare equal (such as `1`, `1.0`, and `True`) can be used interchangeably to index the same dictionary entry.

class dict(***kwargs*)
class dict(*mapping*, /, ***kwargs*)
class dict(*iterable*, /, ***kwargs*)

Return a new dictionary initialized from an optional positional argument and a possibly empty set of keyword arguments.

Dictionaries can be created by several means:

- Use a comma-separated list of `key: value` pairs within braces: `{'jack': 4098, 'sjoerd': 4127}` or `{4098: 'jack', 4127: 'sjoerd'}`
- Use a dict comprehension: `{}`, `{x: x ** 2 for x in range(10)}`

- Use the type constructor: `dict()`, `dict([('foo', 100), ('bar', 200)])`, `dict(foo=100, bar=200)`

If no positional argument is given, an empty dictionary is created. If a positional argument is given and it defines a `keys()` method, a dictionary is created by calling **`__getitem__()`** on the argument with each returned key from the method. Otherwise, the positional argument must be an iterable object. Each item in the iterable must itself be an iterable with exactly two elements. The first element of each item becomes a key in the new dictionary, and the second element the corresponding value. If a key occurs more than once, the last value for that key becomes the corresponding value in the new dictionary.

If keyword arguments are given, the keyword arguments and their values are added to the dictionary created from the positional argument. If a key being added is already present, the value from the keyword argument replaces the value from the positional argument.

Dictionaries compare equal if and only if they have the same `(key, value)` pairs (regardless of ordering). Order comparisons (‘<’, ‘<=’, ‘>=’, ‘>’) raise **`TypeError`**. To illustrate dictionary creation and equality, the following examples all return a dictionary equal to `{"one": 1, "two": 2, "three": 3}`:

```
>>> a = dict(one=1, two=2, three=3)
>>> b = {'one': 1, 'two': 2, 'three': 3}
>>> c = dict(zip(['one', 'two', 'three'], [1, 2, 3]))
>>> d = dict([('two', 2), ('one', 1), ('three', 3)])
>>> e = dict({'three': 3, 'one': 1, 'two': 2})
>>> f = dict({'one': 1, 'three': 3}, two=2)
>>> a == b == c == d == e == f
True
```

Providing keyword arguments as in the first example only works for keys that are valid Python identifiers. Otherwise, any valid keys can be used.

Dictionaries preserve insertion order. Note that updating a key does not affect the order. Keys added after deletion are inserted at the end.

```
>>> d = {"one": 1, "two": 2, "three": 3, "four": 4}
>>> d
{'one': 1, 'two': 2, 'three': 3, 'four': 4}
>>> list(d)
['one', 'two', 'three', 'four']
>>> list(d.values())
[1, 2, 3, 4]
>>> d["one"] = 42
>>> d
{'one': 42, 'two': 2, 'three': 3, 'four': 4}
>>> del d["two"]
>>> d["two"] = None
>>> d
{'one': 42, 'three': 3, 'four': 4, 'two': None}
```

Changed in version 3.7: Dictionary order is guaranteed to be insertion order. This behavior was an implementation detail of CPython from 3.6.

These are the operations that dictionaries support (and therefore, custom mapping types should support too):

list(d)

Return a list of all the keys used in the dictionary *d*.

len(d)

Return the number of items in the dictionary *d*.

d[key]

Return the item of *d* with key *key*. Raises a **`KeyError`** if *key* is not in the map.

If a subclass of dict defines a method **`__missing__()`** and *key* is not present, the `d[key]` operation calls that method with the key *key* as argument. The `d[key]` operation then returns or raises whatever is returned or raised by the `__missing__(key)` call. No other operations or methods invoke **`__missing__()`**. If **`__missing__()`** is not defined, **`KeyError`** is raised. **`__missing__()`** must be a method; it cannot be an instance variable:

```
>>> class Counter(dict):
...     def __missing__(self, key):
...         return 0
...
```

```
>>> c = Counter()
>>> c['red']
0
>>> c['red'] += 1
>>> c['red']
1
```

The example above shows part of the implementation of **`collections.Counter`**. A different **`__missing__()`** method is used by **`collections.defaultdict`**.

d[key] = value

Set `d[key]` to *value*.

del d[key]

Remove `d[key]` from *d*. Raises a **`KeyError`** if *key* is not in the map.

key in d

Return `True` if *d* has a key *key*, else `False`.

key not in d

Equivalent to `not key in d`.

iter(d)

Return an iterator over the keys of the dictionary. This is a shortcut for `iter(d.keys())`.

clear()

Remove all items from the dictionary.

copy()

Return a shallow copy of the dictionary.

classmethod fromkeys(*iterable*, *value=None*, */*)

Create a new dictionary with keys from *iterable* and values set to *value*.

`fromkeys()` is a class method that returns a new dictionary. *value* defaults to `None`. All of the values refer to just a single instance, so it generally doesn’t make sense for *value* to be a mutable object such as an empty list. To get distinct values, use a dict comprehension instead.

get(*key*, *default=None*, */*)

Return the value for *key* if *key* is in the dictionary, else *default*. If *default* is not given, it defaults to `None`, so that this method never raises a **`KeyError`**.

items()

Return a new view of the dictionary’s items (`(key, value)` pairs). See the documentation of view objects.

keys()

Return a new view of the dictionary’s keys. See the documentation of view objects.

pop(*key*, */*)
pop(*key*, *default*, */*)

If *key* is in the dictionary, remove it and return its value, else return *default*. If *default* is not given and *key* is not in the dictionary, a **`KeyError`** is raised.

popitem()

Remove and return a `(key, value)` pair from the dictionary. Pairs are returned in LIFO order.

`popitem()` is useful to destructively iterate over a dictionary, as often used in set algorithms. If the dictionary is empty, calling **`popitem()`** raises a **`KeyError`**.

Changed in version 3.7: LIFO order is now guaranteed. In prior versions, **`popitem()`** would return an arbitrary key/value pair.

reversed(d)

Return a reverse iterator over the keys of the dictionary. This is a shortcut for `reversed(d.keys())`.

Added in version 3.8.

setdefault(*key*, *default=None*, */*)

If *key* is in the dictionary, return its value. If not, insert *key* with a value of *default* and return *default*. *default* defaults to `None`.

update(***kwargs*)
update(*mapping*, /, ***kwargs*)
update(*iterable*, /, ***kwargs*)

Update the dictionary with the key/value pairs from *mapping* or *iterable* and *kwargs*, overwriting existing keys. Return `None`.

`update()` accepts either another object with a `keys()` method (in which case **`__getitem__()`** is called with every key returned from the method) or an iterable of key/value pairs (as tuples or other iterables of length two). If keyword arguments are specified, the dictionary is then updated with those key/value pairs: `d.update(red=1, blue=2)`.

values()

Return a new view of the dictionary's values. See the documentation of view objects.

An equality comparison between one `dict.values()` view and another will always return `False`. This also applies when comparing `dict.values()` to itself:

```
>>> d = {'a': 1}
>>> d.values() == d.values()
False
```

d | other

Create a new dictionary with the merged keys and values of *d* and *other*, which must both be dictionaries. The values of *other* take priority when *d* and *other* share keys.

Added in version 3.9.

d |= other

Update the dictionary *d* with keys and values from *other*, which may be either a mapping or an iterable of key/value pairs. The values of *other* take priority when *d* and *other* share keys.

Added in version 3.9.

Dictionaries and dictionary views are reversible.

```
>>> d = {"one": 1, "two": 2, "three": 3, "four": 4}
>>> d
{'one': 1, 'two': 2, 'three': 3, 'four': 4}
>>> list(reversed(d))
['four', 'three', 'two', 'one']
>>> list(reversed(d.values()))
[4, 3, 2, 1]
>>> list(reversed(d.items()))
[('four', 4), ('three', 3), ('two', 2), ('one', 1)]
```

Changed in version 3.8: Dictionaries are now reversible.

See also

`types.MappingProxyType` can be used to create a read-only view of a `dict`.

Dictionary view objects

The objects returned by `dict.keys()`, `dict.values()` and `dict.items()` are *view objects*. They provide a dynamic view on the dictionary's entries, which means that when the dictionary changes, the view reflects these changes.

Dictionary views can be iterated over to yield their respective data, and support membership tests:

len(dictview)

Return the number of entries in the dictionary.

iter(dictview)

Return an iterator over the keys, values or items (represented as tuples of `(key, value)`) in the dictionary.

Keys and values are iterated over in insertion order. This allows the creation of `(value, key)` pairs using `zip()`: `pairs = zip(d.values(), d.keys())`. Another way to create the same list is `pairs = [(v, k) for (k, v) in d.items()]`.

Iterating views while adding or deleting entries in the dictionary may raise a `RuntimeError` or fail to iterate over all entries.

Changed in version 3.7: Dictionary order is guaranteed to be insertion order.

x in dictview

Return `True` if *x* is in the underlying dictionary’s keys, values or items (in the latter case, *x* should be a `(key, value)` tuple).

reversed(dictview)

Return a reverse iterator over the keys, values or items of the dictionary. The view will be iterated in reverse order of the insertion.

Changed in version 3.8: Dictionary views are now reversible.

dictview.mapping

Return a **`types.MappingProxyType`** that wraps the original dictionary to which the view refers.

Added in version 3.10.

Keys views are set-like since their entries are unique and hashable. Items views also have set-like operations since the (key, value) pairs are unique and the keys are hashable. If all values in an items view are hashable as well, then the items view can interoperate with other sets. (Values views are not treated as set-like since the entries are generally not unique.) For set-like views, all of the operations defined for the abstract base class **`collections.abc.Set`** are available (for example, `==`, `<`, or `^`). While using set operators, set-like views accept any iterable as the other operand, unlike sets which only accept sets as the input.

An example of dictionary view usage:

```
>>> dishes = {'eggs': 2, 'sausage': 1, 'bacon': 1, 'spam': 500}
>>> keys = dishes.keys()
>>> values = dishes.values()

>>> # iteration
>>> n = 0
>>> for val in values:
...     n += val
...
>>> print(n)
>>> # keys and values are iterated over in the same order (insertion
order)
>>> list(keys)
['eggs', 'sausage', 'bacon', 'spam']
>>> list(values)
[2, 1, 1, 500]

>>> # view objects are dynamic and reflect dict changes
>>> del dishes['eggs']
>>> del dishes['sausage']
>>> list(keys)
['bacon', 'spam']

>>> # set operations
>>> keys & {'eggs', 'bacon', 'salad'}
{'bacon'}
>>> keys ^ {'sausage', 'juice'} == {'juice', 'sausage', 'bacon',
'spam'}
True
>>> keys | ['juice', 'juice', 'juice'] == {'bacon', 'spam', 'juice'}
True

>>> # get back a read-only proxy for the original dictionary
>>> values.mapping
mappingproxy({'bacon': 1, 'spam': 500})
>>> values.mapping['spam']
500
```

Context Manager Types

Python’s `with` statement supports the concept of a runtime context defined by a context manager. This is implemented using a pair of methods that allow user-defined classes to define a runtime context that is entered before the statement body is executed and exited when the statement ends:

contextmanager.__enter__()

Enter the runtime context and return either this object or another object related to the runtime context. The value returned by this method is bound to the identifier in the **as** clause of `with` statements using this context manager.

An example of a context manager that returns itself is a file object. File objects return themselves from __enter__() to allow `open()` to be used as the context expression in a `with` statement.

An example of a context manager that returns a related object is the one returned by `decimal.localcontext()`. These managers set the active decimal context to a copy of the original decimal context and then return the copy. This allows changes to be made to the current decimal context in the body of the `with` statement without affecting code outside the **`with`** statement.

contextmanager.__exit__(*exc_type*, *exc_val*, *exc_tb*)

Exit the runtime context and return a Boolean flag indicating if any exception that occurred should be suppressed. If an exception occurred while executing the body of the `with` statement, the arguments contain the exception type, value and traceback information. Otherwise, all three arguments are `None`.

Returning a true value from this method will cause the `with` statement to suppress the exception and continue execution with the statement immediately following the **`with`** statement. Otherwise the exception continues propagating after this method has finished executing.

If this method raises an exception while handling an earlier exception from the `with` block, the new exception is raised, and the original exception is stored in its `__context__` attribute.

The exception passed in should never be reraised explicitly - instead, this method should return a false value to indicate that the method completed successfully and does not want to suppress the raised exception. This allows context management code to easily detect whether or not an `__exit__()` method has actually failed.

Python defines several context managers to support easy thread synchronisation, prompt closure of files or other objects, and simpler manipulation of the active decimal arithmetic context. The specific types are not treated specially beyond their implementation of the context management protocol. See the `contextlib` module for some examples.

Python’s generators and the **`contextlib.contextmanager`** decorator provide a convenient way to implement these protocols. If a generator function is decorated with the **`contextlib.contextmanager`** decorator, it will return a context manager implementing the necessary **`__enter__()`** and **`__exit__()`** methods, rather than the iterator produced by an undecorated generator function.

Note that there is no specific slot for any of these methods in the type structure for Python objects in the Python/C API. Extension types wanting to define these methods must provide them as a normal Python accessible method. Compared to the overhead of setting up the runtime context, the overhead of a single class dictionary lookup is negligible.

Type Annotation Types — Generic Alias, Union

The core built-in types for type annotations are Generic Alias and Union.

Generic Alias Type

`GenericAlias` objects are generally created by subscripting a class. They are most often used with container classes, such as **`list`** or **`dict`**. For example, `list[int]` is a `GenericAlias` object created by subscripting the `list` class with the argument **`int`**. `GenericAlias` objects are intended primarily for use with type annotations.

Note

It is generally only possible to subscript a class if the class implements the special method **`__class_getitem__()`**.

A `GenericAlias` object acts as a proxy for a generic type, implementing *parameterized generics*.

For a container class, the argument(s) supplied to a subscription of the class may indicate the type(s) of the elements an object contains. For example, `set[bytes]` can be used in type annotations to signify a **`set`** in which all the elements are of type **`bytes`**.

For a class which defines **`__class_getitem__()`** but is not a container, the argument(s) supplied to a subscription of the class will often indicate the return type(s) of one or more methods defined on an object. For example, **`regular expressions`** can be used on both the **`str`** data type and the **`bytes`** data type:

- If `x = re.search('foo', 'foo')`, `x` will be a re.Match object where the return values of `x.group(0)` and `x[0]` will both be of type **`str`**. We can represent this kind of object in type annotations with the `GenericAlias` `re.Match[str]`.
- If `y = re.search(b'bar', b'bar')`, (note the `b` for **`bytes`**), `y` will also be an instance of `re.Match`, but the return values of `y.group(0)` and `y[0]` will both be of type **`bytes`**. In type annotations, we would represent this variety of re.Match objects with `re.Match[bytes]`.

`GenericAlias` objects are instances of the class **`types.GenericAlias`**, which can also be used to create `GenericAlias` objects directly.

T[X, Y, ...]

Creates a `GenericAlias` representing a type `T` parameterized by types *X*, *Y*, and more depending on the `T` used. For example, a function expecting a **`list`** containing **`float`** elements:

```
def average(values: list[float]) -> float:
    return sum(values) / len(values)
```

Another example for mapping objects, using a **`dict`**, which is a generic type expecting two type parameters representing the key type and the value type. In this example, the function expects a `dict` with keys of type **`str`** and values of type **`int`**:

```
def send_post_request(url: str, body: dict[str, int]) -> None:
    ...
```

The builtin functions **`isinstance()`** and **`issubclass()`** do not accept `GenericAlias` types for their second argument:

```
>>> isinstance([1, 2], list[str])
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: isinstance() argument 2 cannot be a parameterized generic
```

The Python runtime does not enforce type annotations. This extends to generic types and their type parameters. When creating a container object from a `GenericAlias`, the elements in the container are not checked against their type. For example, the following code is discouraged, but will run without errors:

```
>>> t = list[str]
>>> t([1, 2, 3])
[1, 2, 3]
```

Furthermore, parameterized generics erase type parameters during object creation:

```
>>> t = list[str]
>>> type(t)
<class 'types.GenericAlias'>

>>> l = t()
>>> type(l)
<class 'list'>
```

Calling **`repr()`** or **`str()`** on a generic shows the parameterized type:

```
>>> repr(list[int])
'list[int]'

>>> str(list[int])
'list[int]'
```

The **`__getitem__()`** method of generic containers will raise an exception to disallow mistakes like `dict[str][str]`:

```
>>> dict[str][str]
Traceback (most recent call last):
  ...
TypeError: dict[str] is not a generic class
```

However, such expressions are valid when type variables are used. The index must have as many elements as there are type variable items in the `GenericAlias` object's **`__args__`**.

```
>>> from typing import TypeVar
>>> Y = TypeVar('Y')
>>> dict[str, Y][int]
dict[str, int]
```

Standard Generic Classes

The following standard library classes support parameterized generics. This list is non-exhaustive.

- `tuple`
- `list`
- `dict`
- `set`
- `frozenset`
- `type`
- `asyncio.Future`
- `asyncio.Task`
- `collections.deque`
- `collections.defaultdict`
- `collections.OrderedDict`
- `collections.Counter`
- `collections.ChainMap`
- `collections.abc.Awaitable`
- `collections.abc.Coroutine`
- `collections.abc.AsyncIterable`
- `collections.abc.AsyncIterator`
- `collections.abc.AsyncGenerator`
- `collections.abc.Iterable`
- `collections.abc.Iterator`
- `collections.abc.Generator`
- `collections.abc.Reversible`
- `collections.abc.Container`
- `collections.abc.Collection`
- `collections.abc.Callable`
- `collections.abc.Set`
- `collections.abc.MutableSet`
- `collections.abc.Mapping`
- `collections.abc.MutableMapping`
- `collections.abc.Sequence`
- `collections.abc.MutableSequence`
- `collections.abc.ByteString`

- `collections.abc.MappingView`
- `collections.abc.KeysView`
- `collections.abc.ItemsView`
- `collections.abc.ValuesView`
- `contextlib.AbstractContextManager`
- `contextlib.AbstractAsyncContextManager`
- `dataclasses.Field`
- `functools.cached_property`
- `functools.partialmethod`
- `os.PathLike`
- `queue.LifoQueue`
- `queue.Queue`
- `queue.PriorityQueue`
- `queue.SimpleQueue`
- re.Pattern
- re.Match
- `shelve.BsdDbShelf`
- `shelve.DbfilenameShelf`
- `shelve.Shelf`
- `types.MappingProxyType`
- `weakref.WeakKeyDictionary`
- `weakref.WeakMethod`
- `weakref.WeakSet`
- `weakref.WeakValueDictionary`

Special Attributes of `GenericAlias` objects

All parameterized generics implement special read-only attributes.

genericalias.__origin__

This attribute points at the non-parameterized generic class:

```
>>> list[int].__origin__
<class 'list'>
```

genericalias.__args__

This attribute is a `tuple` (possibly of length 1) of generic types passed to the original `__class_getitem__()` of the generic class:

```
>>> dict[str, list[int]].__args__
(<class 'str'>, list[int])
```

genericalias.__parameters__

This attribute is a lazily computed tuple (possibly empty) of unique type variables found in `__args__`:

```
>>> from typing import TypeVar

>>> T = TypeVar('T')
>>> list[T].__parameters__
(~T,)
```

Note

A `GenericAlias` object with [`typing.ParamSpec`](#) parameters may not have correct `__parameters__` after substitution because [`typing.ParamSpec`](#) is intended primarily for static type checking.

genericalias.__unpacked__

A boolean that is true if the alias has been unpacked using the `*` operator (see [`TypeVarTuple`](#)).

Added in version 3.11.

See also

PEP 484 [https://peps.python.org/pep-0484/] - Type Hints

Introducing Python’s framework for type annotations.

PEP 585 [https://peps.python.org/pep-0585/] - Type Hinting Generics In Standard Collections

Introducing the ability to natively parameterize standard-library classes, provided they implement the special class method `__class_getitem__()`.

Generics, user-defined generics and `typing.Generic`

Documentation on how to implement generic classes that can be parameterized at runtime and understood by static type-checkers.

Added in version 3.9.

Union Type

A union object holds the value of the `|` (bitwise or) operation on multiple type objects. These types are intended primarily for type annotations. The union type expression enables cleaner type hinting syntax compared to subscripting **`typing.Union`**.

X | Y | ...

Defines a union object which holds types *X*, *Y*, and so forth. `X | Y` means either X or Y. It is equivalent to `typing.Union[X, Y]`. For example, the following function expects an argument of type **`int`** or **`float`**:

```
def square(number: int | float) -> int | float:
    return number ** 2
```

Note

The `|` operand cannot be used at runtime to define unions where one or more members is a forward reference. For example, `int | "Foo"`, where `"Foo"` is a reference to a class not yet defined, will fail at runtime. For unions which include forward references, present the whole expression as a string, e.g. `"int | Foo"`.

union_object == other

Union objects can be tested for equality with other union objects. Details:

- Unions of unions are flattened:

  ```
  (int | str) | float == int | str | float
  ```

- Redundant types are removed:

  ```
  int | str | int == int | str
  ```

- When comparing unions, the order is ignored:

  ```
  int | str == str | int
  ```

- It creates instances of **`typing.Union`**:

```
int | str == typing.Union[int, str]
type(int | str) is typing.Union
```

- Optional types can be spelled as a union with `None`:

  ```
  str | None == typing.Optional[str]
  ```

isinstance(obj, union_object)

issubclass(obj, union_object)

Calls to **`isinstance()`** and **`issubclass()`** are also supported with a union object:

```
>>> isinstance("", int | str)
True
```

However, parameterized generics in union objects cannot be checked:

```
>>> isinstance(1, int | list[int])  # short-circuit evaluation
True
>>> isinstance([1], int | list[int])
Traceback (most recent call last):
  ...
TypeError: isinstance() argument 2 cannot be a parameterized generic
```

The user-exposed type for the union object can be accessed from **`typing.Union`** and used for **`isinstance()`** checks:

```
>>> import typing
>>> isinstance(int | str, typing.Union)
True
>>> typing.Union()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: cannot create 'typing.Union' instances
```

Note

The **`__or__()`** method for type objects was added to support the syntax `X | Y`. If a metaclass implements **`__or__()`**, the Union may override it:

```
>>> class M(type):
...     def __or__(self, other):
...         return "Hello"
...
```

```
>>> class C(metaclass=M):
...     pass
...
>>> C | int
'Hello'
>>> int | C
int | C
```

See also

PEP 604 [https://peps.python.org/pep-0604/] – PEP proposing the `X | Y` syntax and the Union type.

Added in version 3.10.

Changed in version 3.14: Union objects are now instances of `typing.Union`. Previously, they were instances of `types.UnionType`, which remains an alias for `typing.Union`.

Other Built-in Types

The interpreter supports several other kinds of objects. Most of these support only one or two operations.

Modules

The only special operation on a module is attribute access: `m.name`, where *m* is a module and *name* accesses a name defined in *m*'s symbol table. Module attributes can be assigned to. (Note that the `import` statement is not, strictly speaking, an operation on a module object; `import foo` does not require a module object named *foo* to exist, rather it requires an (external) *definition* for a module named *foo* somewhere.)

A special attribute of every module is `__dict__`. This is the dictionary containing the module's symbol table. Modifying this dictionary will actually change the module's symbol table, but direct assignment to the `__dict__` attribute is not possible (you can write `m.__dict__['a'] = 1`, which defines `m.a` to be `1`, but you can't write `m.__dict__ = {}`). Modifying `__dict__` directly is not recommended.

Modules built into the interpreter are written like this: `<module 'sys' (built-in)>`. If loaded from a file, they are written as `<module 'os' from '/usr/local/lib/pythonX.Y/os.pyc'>`.

Classes and Class Instances

See Objects, values and types and Class definitions for these.

Functions

Function objects are created by function definitions. The only operation on a function object is to call it: `func(argument-list)`.

There are really two flavors of function objects: built-in functions and user-defined functions. Both support the same operation (to call the function), but the implementation is different, hence the different object types.

See Function definitions for more information.

Methods

Methods are functions that are called using the attribute notation. There are two flavors: built-in methods (such as **`append()`** on lists) and class instance method. Built-in methods are described with the types that support them.

If you access a method (a function defined in a class namespace) through an instance, you get a special object: a *bound method* (also called instance method) object. When called, it will add the `self` argument to the argument list. Bound methods have two special read-only attributes: **`m.__self__`** is the object on which the method operates, and **`m.__func__`** is the function implementing the method. Calling `m(arg-1, arg-2, ..., arg-n)` is completely equivalent to calling `m.__func__(m.__self__, arg-1, arg-2, ..., arg-n)`.

Like function objects, bound method objects support getting arbitrary attributes. However, since method attributes are actually stored on the underlying function object (**`method.__func__`**), setting method attributes on bound methods is disallowed. Attempting to set an attribute on a method results in an **`AttributeError`** being raised. In order to set a method attribute, you need to explicitly set it on the underlying function object:

```
>>> class C:
...     def method(self):
...         pass
...
>>> c = C()
>>> c.method.whoami = 'my name is method'  # can't set on the method
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'method' object has no attribute 'whoami'
>>> c.method.__func__.whoami = 'my name is method'
>>> c.method.whoami
'my name is method'
```

See Instance methods for more information.

Code Objects

Code objects are used by the implementation to represent “pseudo-compiled” executable Python code such as a function body. They differ from function objects because they don’t contain a reference to their global execution environment. Code objects are returned by the built-in `compile()` function and can be extracted from function objects through their `__code__` attribute. See also the `code` module.

Accessing `__code__` raises an auditing event `object.__getattr__` with arguments `obj` and `"__code__"`.

A code object can be executed or evaluated by passing it (instead of a source string) to the `exec()` or `eval()` built-in functions.

See The standard type hierarchy for more information.

Type Objects

Type objects represent the various object types. An object’s type is accessed by the built-in function `type()`. There are no special operations on types. The standard module `types` defines names for all standard built-in types.

Types are written like this: `<class 'int'>`.

The Null Object

This object is returned by functions that don’t explicitly return a value. It supports no special operations. There is exactly one null object, named `None` (a built-in name). `type(None)()` produces the same singleton.

It is written as `None`.

The Ellipsis Object

This object is commonly used to indicate that something is omitted. It supports no special operations. There is exactly one ellipsis object, named **`Ellipsis`** (a built-in name). `type(Ellipsis)()` produces the **`Ellipsis`** singleton.

It is written as `Ellipsis` or `...`.

In typical use, `...` as the `Ellipsis` object appears in a few different places, for instance:

- In type annotations, such as callable arguments or tuple elements.
- As the body of a function instead of a pass statement.
- In third-party libraries, such as Numpy’s slicing and striding [https://numpy.org/doc/stable/user/basics.indexing.html#slicing-and-striding].

Python also uses three dots in ways that are not `Ellipsis` objects, for instance:

- Doctest’s **`ELLIPSIS`**, as a pattern for missing content.
- The default Python prompt of the interactive shell when partial input is incomplete.

Lastly, the Python documentation often uses three dots in conventional English usage to mean omitted content, even in code examples that also use them as the `Ellipsis`.

The NotImplemented Object

This object is returned from comparisons and binary operations when they are asked to operate on types they don't support. See Comparisons for more information. There is exactly one **`NotImplemented`** object. `type(NotImplemented)()` produces the singleton instance.

It is written as `NotImplemented`.

Internal Objects

See The standard type hierarchy for this information. It describes stack frame objects, traceback objects, and slice objects.

Special Attributes

The implementation adds a few special read-only attributes to several object types, where they are relevant. Some of these are not reported by the **`dir()`** built-in function.

definition.__name__

The name of the class, function, method, descriptor, or generator instance.

definition.__qualname__

The qualified name of the class, function, method, descriptor, or generator instance.

Added in version 3.3.

definition.__module__

The name of the module in which a class or function was defined.

definition.__doc__

The documentation string of a class or function, or `None` if undefined.

definition.__type_params__

The type parameters of generic classes, functions, and type aliases. For classes and functions that are not generic, this will be an empty tuple.

Added in version 3.12.

Integer string conversion length limitation

CPython has a global limit for converting between **`int`** and **`str`** to mitigate denial of service attacks. This limit *only* applies to decimal or other non-power-of-two number bases. Hexadecimal, octal, and binary conversions are unlimited. The limit can be configured.

The **`int`** type in CPython is an arbitrary length number stored in binary form (commonly known as a “bignum”). There exists no algorithm that can convert a string to a binary integer or a binary integer to a string in linear time, *unless* the base is a power of 2. Even the best known algorithms for base 10 have sub-quadratic complexity. Converting a large value such as `int('1' * 500_000)` can take over a second on a fast CPU.

Limiting conversion size offers a practical way to avoid **CVE 2020-10735** [https://www.cve.org/CVERecord?id=CVE-2020-10735].

The limit is applied to the number of digit characters in the input or output string when a non-linear conversion algorithm would be involved. Underscores and the sign are not counted towards the limit.

When an operation would exceed the limit, a **`ValueError`** is raised:

```
>>> import sys
>>> sys.set_int_max_str_digits(4300)  # Illustrative, this is the
default.
>>> _ = int('2' * 5432)
Traceback (most recent call last):
...
ValueError: Exceeds the limit (4300 digits) for integer string
conversion: value has 5432 digits; use sys.set_int_max_str_digits() to
increase the limit
>>> i = int('2' * 4300)
>>> len(str(i))
4300
>>> i_squared = i*i
>>> len(str(i_squared))
Traceback (most recent call last):
...
```

```
ValueError: Exceeds the limit (4300 digits) for integer string
conversion; use sys.set_int_max_str_digits() to increase the limit
>>> len(hex(i_squared))
7144
>>> assert int(hex(i_squared), base=16) == i*i  # Hexadecimal is
unlimited.
```

The default limit is 4300 digits as provided in `sys.int_info.default_max_str_digits`. The lowest limit that can be configured is 640 digits as provided in `sys.int_info.str_digits_check_threshold`.

Verification:

```
>>> import sys
>>> assert sys.int_info.default_max_str_digits == 4300, sys.int_info
>>> assert sys.int_info.str_digits_check_threshold == 640, sys.int_info
>>> msg =
int('578966293710682886880994035146873798396722250538762761564'
...
'9252925514383915483333812743580549779436104706260696366600'
...           '571186405732').to_bytes(53, 'big')
...
```

Added in version 3.11.

Affected APIs

The limitation only applies to potentially slow conversions between `int` and `str` or `bytes`:

- `int(string)` with default base 10.
- `int(string, base)` for all bases that are not a power of 2.
- `str(integer)`.
- `repr(integer)`.
- any other string conversion to base 10, for example `f"{integer}"`, `"{}".format(integer)`, or `b"%d" % integer`.

The limitations do not apply to functions with a linear algorithm:

- `int(string, base)` with base 2, 4, 8, 16, or 32.
- `int.from_bytes()` and `int.to_bytes()`.
- `hex()`, `oct()`, `bin()`.

- Format Specification Mini-Language for hex, octal, and binary numbers.
- `str` to `float`.
- `str` to `decimal.Decimal`.

Configuring the limit

Before Python starts up you can use an environment variable or an interpreter command line flag to configure the limit:

- `PYTHONINTMAXSTRDIGITS`, e.g. `PYTHONINTMAXSTRDIGITS=640 python3` to set the limit to 640 or `PYTHONINTMAXSTRDIGITS=0 python3` to disable the limitation.
- `-X int_max_str_digits`, e.g. `python3 -X int_max_str_digits=640`
- `sys.flags.int_max_str_digits` contains the value of `PYTHONINTMAXSTRDIGITS` or `-X int_max_str_digits`. If both the env var and the `-X` option are set, the `-X` option takes precedence. A value of *-1* indicates that both were unset, thus a value of `sys.int_info.default_max_str_digits` was used during initialization.

From code, you can inspect the current limit and set a new one using these `sys` APIs:

- `sys.get_int_max_str_digits()` and `sys.set_int_max_str_digits()` are a getter and setter for the interpreter-wide limit. Subinterpreters have their own limit.

Information about the default and minimum can be found in `sys.int_info`:

- `sys.int_info.default_max_str_digits` is the compiled-in default limit.
- `sys.int_info.str_digits_check_threshold` is the lowest accepted value for the limit (other than 0 which disables it).

Added in version 3.11.

Caution

Setting a low limit *can* lead to problems. While rare, code exists that contains integer constants in decimal in their source that exceed the minimum threshold. A consequence of setting the limit is that Python source code containing decimal integer literals longer than the limit will encounter an error during parsing, usually at startup time or import time or even at installation time - anytime an up to date `.pyc` does not already exist for the code. A workaround for source that contains such large constants is to convert them to `0x` hexadecimal form as it has no limit.

Test your application thoroughly if you use a low limit. Ensure your tests run with the limit set early via the environment or flag so that it applies during startup and even during any installation step that may invoke Python to precompile `.py` sources to `.pyc` files.

Recommended configuration

The default **`sys.int_info.default_max_str_digits`** is expected to be reasonable for most applications. If your application requires a different limit, set it from your main entry point using Python version agnostic code as these APIs were added in security patch releases in versions before 3.12.

Example:

```
>>> import sys
>>> if hasattr(sys, "set_int_max_str_digits"):
...     upper_bound = 68000
...     lower_bound = 4004
...     current_limit = sys.get_int_max_str_digits()
...     if current_limit == 0 or current_limit > upper_bound:
...         sys.set_int_max_str_digits(upper_bound)
...     elif current_limit < lower_bound:
...         sys.set_int_max_str_digits(lower_bound)
```

If you need to disable it entirely, set it to `0`.

Footnotes

[1]

Additional information on these special methods may be found in the Python Reference Manual (Basic customization).

[2]

As a consequence, the list `[1, 2]` is considered equal to `[1.0, 2.0]`, and similarly for tuples.

[3]

They must have since the parser can’t tell the type of the operands.

[4] (1,2,3,4)

Cased characters are those with general category property being one of “Lu” (Letter, uppercase), “Ll” (Letter, lowercase), or “Lt” (Letter, titlecase).

[5] (1,2)

To format only a tuple you should therefore provide a singleton tuple whose only element is the tuple to be formatted.

Built-in Exceptions

In Python, all exceptions must be instances of a class that derives from `BaseException`. In a `try` statement with an `except` clause that mentions a particular class, that clause also handles any exception classes derived from that class (but not exception classes from which *it* is derived). Two exception classes that are not related via subclassing are never equivalent, even if they have the same name.

The built-in exceptions listed in this chapter can be generated by the interpreter or built-in functions. Except where mentioned, they have an “associated value” indicating the detailed cause of the error. This may be a string or a tuple of several items of information (e.g., an error code and a string explaining the code). The associated value is usually passed as arguments to the exception class’s constructor.

User code can raise built-in exceptions. This can be used to test an exception handler or to report an error condition “just like” the situation in which the interpreter raises the same exception; but beware that there is nothing to prevent user code from raising an inappropriate error.

The built-in exception classes can be subclassed to define new exceptions; programmers are encouraged to derive new exceptions from the `Exception` class or one of its subclasses, and not from `BaseException`. More information on defining exceptions is available in the Python Tutorial under User-defined Exceptions.

Exception context

Three attributes on exception objects provide information about the context in which the exception was raised:

BaseException.__context__
BaseException.__cause__
BaseException.__suppress_context__

When raising a new exception while another exception is already being handled, the new exception’s **`__context__`** attribute is automatically set to the handled exception. An exception may be handled when an **`except`** or **`finally`** clause, or a **`with`** statement, is used.

This implicit exception context can be supplemented with an explicit cause by using **`from`** with **`raise`**:

```
raise new_exc from original_exc
```

The expression following **`from`** must be an exception or `None`. It will be set as **`__cause__`** on the raised exception. Setting **`__cause__`** also implicitly sets the **`__suppress_context__`** attribute to `True`, so that using `raise new_exc from None` effectively replaces the old exception with the new one for display purposes (e.g. converting **`KeyError`** to **`AttributeError`**), while leaving the old exception available in **`__context__`** for introspection when debugging.

The default traceback display code shows these chained exceptions in addition to the traceback for the exception itself. An explicitly chained exception in **`__cause__`** is always shown when present. An implicitly chained exception in **`__context__`** is shown only if **`__cause__`** is **`None`** and **`__suppress_context__`** is false.

In either case, the exception itself is always shown after any chained exceptions so that the final line of the traceback always shows the last exception that was raised.

Inheriting from built-in exceptions

User code can create subclasses that inherit from an exception type. It’s recommended to only subclass one exception type at a time to avoid any possible conflicts between how the bases handle the `args` attribute, as well as due to possible memory layout incompatibilities.

CPython implementation detail: Most built-in exceptions are implemented in C for efficiency, see: Objects/exceptions.c [https://github.com/python/cpython/tree/3.14/Objects/exceptions.c]. Some have custom memory layouts which makes it impossible to create a subclass that inherits from multiple exception types. The memory layout of a type is an implementation detail and might change between Python versions, leading to new conflicts in the future. Therefore, it’s recommended to avoid subclassing multiple exception types altogether.

Base classes

The following exceptions are used mostly as base classes for other exceptions.

exception BaseException

The base class for all built-in exceptions. It is not meant to be directly inherited by user-defined classes (for that, use **`Exception`**). If **`str()`** is called on an instance of this class, the representation of the argument(s) to the instance are returned, or the empty string when there were no arguments.

args

The tuple of arguments given to the exception constructor. Some built-in exceptions (like **`OSError`**) expect a certain number of arguments and assign a special meaning to the elements of this tuple, while others are usually called only with a single string giving an error message.

with_traceback(*tb*)

This method sets *tb* as the new traceback for the exception and returns the exception object. It was more commonly used before the exception chaining features of **PEP 3134** [https://peps.python.org/pep-3134/] became available. The following example shows how we can convert an instance of `SomeException` into an instance of `OtherException` while preserving the traceback. Once raised, the current frame is pushed onto the traceback of the `OtherException`, as would have happened to the traceback of the original `SomeException` had we allowed it to propagate to the caller.

```
try:
    ...
except SomeException:
    tb = sys.exception().__traceback__
    raise OtherException(...).with_traceback(tb)
```

__traceback__

A writable field that holds the traceback object associated with this exception. See also: The raise statement.

add_note(*note*)

Add the string `note` to the exception’s notes which appear in the standard traceback after the exception string. A **`TypeError`** is raised if `note` is not a string.

Added in version 3.11.

__notes__

A list of the notes of this exception, which were added with `add_note()`. This attribute is created when `add_note()` is called.

Added in version 3.11.

exception Exception

All built-in, non-system-exiting exceptions are derived from this class. All user-defined exceptions should also be derived from this class.

exception ArithmeticError

The base class for those built-in exceptions that are raised for various arithmetic errors: `OverflowError`, `ZeroDivisionError`, `FloatingPointError`.

exception BufferError

Raised when a buffer related operation cannot be performed.

exception LookupError

The base class for the exceptions that are raised when a key or index used on a mapping or sequence is invalid: `IndexError`, `KeyError`. This can be raised directly by `codecs.lookup()`.

Concrete exceptions

The following exceptions are the exceptions that are usually raised.

exception AssertionError

Raised when an `assert` statement fails.

exception AttributeError

Raised when an attribute reference (see Attribute references) or assignment fails. (When an object does not support attribute references or attribute assignments at all, **`TypeError`** is raised.)

The optional *name* and *obj* keyword-only arguments set the corresponding attributes:

name
: The name of the attribute that was attempted to be accessed.

obj
: The object that was accessed for the named attribute.

Changed in version 3.10: Added the **`name`** and **`obj`** attributes.

exception EOFError

Raised when the **`input()`** function hits an end-of-file condition (EOF) without reading any data. (Note: the **`io.IOBase.read()`** and **`io.IOBase.readline()`** methods return an empty string when they hit EOF.)

exception FloatingPointError

Not currently used.

exception GeneratorExit

Raised when a generator or coroutine is closed; see **`generator.close()`** and **`coroutine.close()`**. It directly inherits from **`BaseException`** instead of **`Exception`** since it is technically not an error.

exception ImportError

Raised when the **`import`** statement has troubles trying to load a module. Also raised when the “from list” in `from`

`...` `import` has a name that cannot be found.

The optional *name* and *path* keyword-only arguments set the corresponding attributes:

name

The name of the module that was attempted to be imported.

path

The path to any file which triggered the exception.

Changed in version 3.3: Added the **`name`** and **`path`** attributes.

exception ModuleNotFoundError

A subclass of **`ImportError`** which is raised by **`import`** when a module could not be located. It is also raised when `None` is found in **`sys.modules`**.

Added in version 3.6.

exception IndexError

Raised when a sequence subscript is out of range. (Slice indices are silently truncated to fall in the allowed range; if an index is not an integer, **`TypeError`** is raised.)

exception KeyError

Raised when a mapping (dictionary) key is not found in the set of existing keys.

exception KeyboardInterrupt

Raised when the user hits the interrupt key (normally `Control-C` or `Delete`). During execution, a check for interrupts is made regularly. The exception inherits from **`BaseException`** so as to not be accidentally caught by code that catches `Exception` and thus prevent the interpreter from exiting.

Note

Catching a `KeyboardInterrupt` requires special consideration. Because it can be raised at unpredictable points, it may, in some circumstances, leave the running program in an inconsistent state. It is generally best to allow `KeyboardInterrupt` to end the program as quickly as possible or avoid raising it entirely. (See Note on Signal Handlers and Exceptions.)

exception MemoryError

Raised when an operation runs out of memory but the situation may still be rescued (by deleting some objects). The associated value is a string indicating what kind of (internal) operation ran out of memory. Note that because of the underlying memory management architecture (C’s `malloc()` function), the interpreter may not always be able to completely recover from this situation; it nevertheless raises an exception so that a stack traceback can be printed, in case a run-away program was the cause.

exception NameError

Raised when a local or global name is not found. This applies only to unqualified names. The associated value is an error message that includes the name that could not be found.

The optional *name* keyword-only argument sets the attribute:

name

The name of the variable that was attempted to be accessed.

Changed in version 3.10: Added the `name` attribute.

exception NotImplementedError

This exception is derived from `RuntimeError`. In user defined base classes, abstract methods should raise this exception when they require derived classes to override the method, or while the class is being developed to indicate that the real implementation still needs to be added.

Note

It should not be used to indicate that an operator or method is not meant to be supported at all – in that case either leave the operator / method undefined or, if a subclass, set it to `None`.

Caution

`NotImplementedError` and **`NotImplemented`** are not interchangeable. This exception should only be used as described above; see `NotImplemented` for details on correct usage of the built-in constant.

exception OSError([*arg*])

exception OSError(*errno*, *strerror*[, *filename*[, *winerror*[, *filename2*]]])

This exception is raised when a system function returns a system-related error, including I/O failures such as “file not found” or “disk full” (not for illegal argument types or other incidental errors).

The second form of the constructor sets the corresponding attributes, described below. The attributes default to **`None`** if not specified. For backwards compatibility, if three arguments are passed, the **`args`** attribute contains only a 2-tuple of the first two constructor arguments.

The constructor often actually returns a subclass of **`OSError`**, as described in OS exceptions below. The particular subclass depends on the final **`errno`** value. This behaviour only occurs when constructing **`OSError`** directly or via an alias, and is not inherited when subclassing.

errno

A numeric error code from the C variable **`errno`**.

winerror

Under Windows, this gives you the native Windows error code. The **`errno`** attribute is then an approximate translation, in POSIX terms, of that native error code.

Under Windows, if the *winerror* constructor argument is an integer, the **`errno`** attribute is determined from the Windows error code, and the *errno* argument is ignored. On other platforms, the *winerror* argument is ignored, and the **`winerror`** attribute does not exist.

strerror

The corresponding error message, as provided by the operating system. It is formatted by the C functions **`perror()`** under POSIX, and **`FormatMessage()`** under Windows.

filename
filename2

For exceptions that involve a file system path (such as `open()` or `os.unlink()`), `filename` is the file name passed to the function. For functions that involve two file system paths (such as `os.rename()`), `filename2` corresponds to the second file name passed to the function.

Changed in version 3.3: `EnvironmentError`, `IOError`, `WindowsError`, `socket.error`, `select.error` and `mmap.error` have been merged into `OSError`, and the constructor may return a subclass.

Changed in version 3.4: The `filename` attribute is now the original file name passed to the function, instead of the name encoded to or decoded from the filesystem encoding and error handler. Also, the *filename2* constructor argument and attribute was added.

exception OverflowError

Raised when the result of an arithmetic operation is too large to be represented. This cannot occur for integers (which would rather raise `MemoryError` than give up). However, for historical reasons, OverflowError is sometimes raised for integers that are outside a required range. Because of the lack of standardization of floating-point exception handling in C, most floating-point operations are not checked.

exception PythonFinalizationError

This exception is derived from `RuntimeError`. It is raised when an operation is blocked during interpreter shutdown also known as Python finalization.

Examples of operations which can be blocked with a `PythonFinalizationError` during the Python finalization:

- Creating a new Python thread.
- `Joining` a running daemon thread.
- `os.fork()`.

See also the `sys.is_finalizing()` function.

Added in version 3.13: Previously, a plain `RuntimeError` was raised.

Changed in version 3.14: `threading.Thread.join()` can now raise this exception.

exception RecursionError

This exception is derived from `RuntimeError`. It is raised when the interpreter detects that the maximum recursion depth (see `sys.getrecursionlimit()`) is exceeded.

Added in version 3.5: Previously, a plain `RuntimeError` was raised.

exception ReferenceError

This exception is raised when a weak reference proxy, created by the `weakref.proxy()` function, is used to access an attribute of the referent after it has been garbage collected. For more information on weak references, see the `weakref` module.

exception RuntimeError

Raised when an error is detected that doesn’t fall in any of the other categories. The associated value is a string indicating what precisely went wrong.

exception StopIteration

Raised by built-in function `next()` and an iterator's `__next__()` method to signal that there are no further items produced by the iterator.

value

The exception object has a single attribute **`value`**, which is given as an argument when constructing the exception, and defaults to **`None`**.

When a generator or coroutine function returns, a new **`StopIteration`** instance is raised, and the value returned by the function is used as the **`value`** parameter to the constructor of the exception.

If a generator code directly or indirectly raises **`StopIteration`**, it is converted into a **`RuntimeError`** (retaining the **`StopIteration`** as the new exception's cause).

Changed in version 3.3: Added `value` attribute and the ability for generator functions to use it to return a value.

Changed in version 3.5: Introduced the RuntimeError transformation via `from __future__ import generator_stop`, see **PEP 479** [https://peps.python.org/pep-0479/].

Changed in version 3.7: Enable **PEP 479** [https://peps.python.org/pep-0479/] for all code by default: a **`StopIteration`** error raised in a generator is transformed into a **`RuntimeError`**.

exception StopAsyncIteration

Must be raised by **`__anext__()`** method of an asynchronous iterator object to stop the iteration.

Added in version 3.5.

exception SyntaxError(*message*, *details*)

Raised when the parser encounters a syntax error. This may occur in an **`import`** statement, in a call to the built-in functions `compile()`, `exec()`, or `eval()`, or when reading the initial script or standard input (also interactively).

The `str()` of the exception instance returns only the error message. Details is a tuple whose members are also available as separate attributes.

filename

The name of the file the syntax error occurred in.

lineno

Which line number in the file the error occurred in. This is 1-indexed: the first line in the file has a `lineno` of 1.

offset

The column in the line where the error occurred. This is 1-indexed: the first character in the line has an `offset` of 1.

text

The source code text involved in the error.

end_lineno

Which line number in the file the error occurred ends in. This is 1-indexed: the first line in the file has a `lineno` of 1.

end_offset

The column in the end line where the error occurred finishes. This is 1-indexed: the first character in the line has an `offset` of 1.

For errors in f-string fields, the message is prefixed by “f-string: ” and the offsets are offsets in a text constructed from the replacement expression. For example, compiling f’Bad {a b} field’ results in this args attribute: (‘f-string: …’, (‘’, 1, 2, ‘(a b)n’, 1, 5)).

Changed in version 3.10: Added the **`end_lineno`** and **`end_offset`** attributes.

exception IndentationError

Base class for syntax errors related to incorrect indentation. This is a subclass of **`SyntaxError`**.

exception TabError

Raised when indentation contains an inconsistent use of tabs and spaces. This is a subclass of **`IndentationError`**.

exception SystemError

Raised when the interpreter finds an internal error, but the situation does not look so serious to cause it to abandon all hope. The associated value is a string indicating what went wrong (in low-level terms). In CPython, this could be raised by incorrectly using Python’s C API, such as returning a `NULL` value without an exception set.

If you’re confident that this exception wasn’t your fault, or the fault of a package you’re using, you should report this to the author or maintainer of your Python interpreter. Be sure to report the version of the Python interpreter (`sys.version`; it is also printed at the start of an interactive Python session), the exact error message (the exception’s associated value) and if possible the source of the program that triggered the error.

exception SystemExit

This exception is raised by the **`sys.exit()`** function. It inherits from **`BaseException`** instead of **`Exception`** so that it is not accidentally caught by code that catches **`Exception`**.

This allows the exception to properly propagate up and cause the interpreter to exit. When it is not handled, the Python interpreter exits; no stack traceback is printed. The constructor accepts the same optional argument passed to `sys.exit()`. If the value is an integer, it specifies the system exit status (passed to C's **`exit()`** function); if it is `None`, the exit status is zero; if it has another type (such as a string), the object's value is printed and the exit status is one.

A call to `sys.exit()` is translated into an exception so that clean-up handlers (`finally` clauses of `try` statements) can be executed, and so that a debugger can execute a script without running the risk of losing control. The `os._exit()` function can be used if it is absolutely positively necessary to exit immediately (for example, in the child process after a call to `os.fork()`).

code

The exit status or error message that is passed to the constructor. (Defaults to `None`.)

exception TypeError

Raised when an operation or function is applied to an object of inappropriate type. The associated value is a string giving details about the type mismatch.

This exception may be raised by user code to indicate that an attempted operation on an object is not supported, and is not meant to be. If an object is meant to support a given operation but has not yet provided an implementation, `NotImplementedError` is the proper exception to raise.

Passing arguments of the wrong type (e.g. passing a `list` when an `int` is expected) should result in a `TypeError`, but passing arguments with the wrong value (e.g. a number outside expected boundaries) should result in a **`ValueError`**.

exception UnboundLocalError

Raised when a reference is made to a local variable in a function or method, but no value has been bound to that variable. This is a subclass of **`NameError`**.

exception UnicodeError

Raised when a Unicode-related encoding or decoding error occurs. It is a subclass of **`ValueError`**.

`UnicodeError` has attributes that describe the encoding or decoding error. For example, `err.object[err.start:err.end]` gives the particular invalid input that the codec failed on.

encoding

The name of the encoding that raised the error.

reason

A string describing the specific codec error.

object

The object the codec was attempting to encode or decode.

start

The first index of invalid data in **`object`**.

This value should not be negative as it is interpreted as an absolute offset but this constraint is not enforced at runtime.

end

The index after the last invalid data in **`object`**.

This value should not be negative as it is interpreted as an absolute offset but this constraint is not enforced at runtime.

exception UnicodeEncodeError

Raised when a Unicode-related error occurs during encoding. It is a subclass of `UnicodeError`.

exception UnicodeDecodeError

Raised when a Unicode-related error occurs during decoding. It is a subclass of `UnicodeError`.

exception UnicodeTranslateError

Raised when a Unicode-related error occurs during translating. It is a subclass of `UnicodeError`.

exception ValueError

Raised when an operation or function receives an argument that has the right type but an inappropriate value, and the situation is not described by a more precise exception such as `IndexError`.

exception ZeroDivisionError

Raised when the second argument of a division or modulo operation is zero. The associated value is a string indicating the type of the operands and the operation.

The following exceptions are kept for compatibility with previous versions; starting from Python 3.3, they are aliases of `OSError`.

exception EnvironmentError

exception IOError

exception WindowsError

Only available on Windows.

OS exceptions

The following exceptions are subclasses of `OSError`, they get raised depending on the system error code.

exception BlockingIOError

Raised when an operation would block on an object (e.g. socket) set for non-blocking operation. Corresponds to **`errno`** `EAGAIN`, `EALREADY`, `EWOULDBLOCK` and `EINPROGRESS`.

In addition to those of `OSError`, `BlockingIOError` can have one more attribute:

characters_written

An integer containing the number of **bytes** written to the stream before it blocked. This attribute is available when using the buffered I/O classes from the `io` module.

exception ChildProcessError

Raised when an operation on a child process failed. Corresponds to **`errno`** `ECHILD`.

exception ConnectionError

A base class for connection-related issues.

Subclasses are `BrokenPipeError`, `ConnectionAbortedError`, `ConnectionRefusedError` and `ConnectionResetError`.

exception BrokenPipeError

A subclass of `ConnectionError`, raised when trying to write on a pipe while the other end has been closed, or trying to write on a socket which has been shutdown for writing. Corresponds to **`errno`** `EPIPE` and `ESHUTDOWN`.

exception ConnectionAbortedError

A subclass of `ConnectionError`, raised when a connection attempt is aborted by the peer. Corresponds to **`errno`** `ECONNABORTED`.

exception ConnectionRefusedError

A subclass of `ConnectionError`, raised when a connection attempt is refused by the peer. Corresponds to **`errno`** `ECONNREFUSED`.

exception ConnectionResetError

A subclass of `ConnectionError`, raised when a connection is reset by the peer. Corresponds to **`errno`** `ECONNRESET`.

exception FileExistsError

Raised when trying to create a file or directory which already exists. Corresponds to **`errno`** `EEXIST`.

exception FileNotFoundError

Raised when a file or directory is requested but doesn’t exist. Corresponds to **`errno`** `ENOENT`.

exception InterruptedError

Raised when a system call is interrupted by an incoming signal. Corresponds to **`errno`** `EINTR`.

Changed in version 3.5: Python now retries system calls when a syscall is interrupted by a signal, except if the signal handler raises an exception (see **PEP 475** [https://peps.python.org/pep-0475/] for the rationale), instead of raising `InterruptedError`.

exception IsADirectoryError

Raised when a file operation (such as `os.remove()`) is requested on a directory. Corresponds to **`errno`** `EISDIR`.

exception NotADirectoryError

Raised when a directory operation (such as `os.listdir()`) is requested on something which is not a directory. On most POSIX platforms, it may also be raised if an operation attempts to open or traverse a non-directory file as if it were a directory. Corresponds to **`errno`** `ENOTDIR`.

exception PermissionError

Raised when trying to run an operation without the adequate access rights - for example filesystem permissions. Corresponds to **`errno`** `EACCES`, `EPERM`, and `ENOTCAPABLE`.

Changed in version 3.11.1: WASI's `ENOTCAPABLE` is now mapped to `PermissionError`.

exception ProcessLookupError

Raised when a given process doesn't exist. Corresponds to **`errno`** `ESRCH`.

exception TimeoutError

Raised when a system function timed out at the system level. Corresponds to **`errno`** `ETIMEDOUT`.

Added in version 3.3: All the above `OSError` subclasses were added.

See also

PEP 3151 [https://peps.python.org/pep-3151/] - Reworking the OS and IO exception hierarchy

Warnings

The following exceptions are used as warning categories; see the Warning Categories documentation for more details.

exception Warning

Base class for warning categories.

exception UserWarning

Base class for warnings generated by user code.

exception DeprecationWarning

Base class for warnings about deprecated features when those warnings are intended for other Python developers.

Ignored by the default warning filters, except in the `__main__` module (**PEP 565** [https://peps.python.org/pep-0565/]). Enabling the Python Development Mode shows this warning.

The deprecation policy is described in **PEP 387** [https://peps.python.org/pep-0387/].

exception PendingDeprecationWarning

Base class for warnings about features which are obsolete and expected to be deprecated in the future, but are not deprecated at the moment.

This class is rarely used as emitting a warning about a possible upcoming deprecation is unusual, and **`DeprecationWarning`** is preferred for already active deprecations.

Ignored by the default warning filters. Enabling the Python Development Mode shows this warning.

The deprecation policy is described in **PEP 387** [https://peps.python.org/pep-0387/].

exception SyntaxWarning

Base class for warnings about dubious syntax.

This warning is typically emitted when compiling Python source code, and usually won't be reported when running already compiled code.

exception RuntimeWarning

Base class for warnings about dubious runtime behavior.

exception FutureWarning

Base class for warnings about deprecated features when those warnings are intended for end users of applications that are written in Python.

exception ImportWarning

Base class for warnings about probable mistakes in module imports.

Ignored by the default warning filters. Enabling the Python Development Mode shows this warning.

exception UnicodeWarning

Base class for warnings related to Unicode.

exception EncodingWarning

Base class for warnings related to encodings.

See Opt-in EncodingWarning for details.

Added in version 3.10.

exception BytesWarning

Base class for warnings related to `bytes` and `bytearray`.

exception ResourceWarning

Base class for warnings related to resource usage.

Ignored by the default warning filters. Enabling the Python Development Mode shows this warning.

Added in version 3.2.

Exception groups

The following are used when it is necessary to raise multiple unrelated exceptions. They are part of the exception hierarchy so they can be handled with **`except`** like all other exceptions. In addition, they are recognised by **`except*`**, which matches their subgroups based on the types of the contained exceptions.

exception ExceptionGroup(*msg*, *excs*)

exception BaseExceptionGroup(*msg*, *excs*)

Both of these exception types wrap the exceptions in the sequence `excs`. The `msg` parameter must be a string. The difference between the two classes is that **`BaseExceptionGroup`** extends **`BaseException`** and it can wrap any exception, while **`ExceptionGroup`** extends **`Exception`** and it can only wrap subclasses of **`Exception`**. This design is so that `except Exception` catches an **`ExceptionGroup`** but not **`BaseExceptionGroup`**.

The **`BaseExceptionGroup`** constructor returns an **`ExceptionGroup`** rather than a **`BaseExceptionGroup`** if all contained exceptions are **`Exception`** instances, so it can be used to make the selection automatic. The **`ExceptionGroup`** constructor, on the other hand, raises a **`TypeError`** if any contained exception is not an **`Exception`** subclass.

message

The `msg` argument to the constructor. This is a read-only attribute.

exceptions

A tuple of the exceptions in the `excs` sequence given to the constructor. This is a read-only attribute.

subgroup(*condition*)

Returns an exception group that contains only the exceptions from the current group that match *condition*, or `None` if the result is empty.

The condition can be an exception type or tuple of exception types, in which case each exception is checked for a match using the same check that is used in an `except` clause. The condition can also be a callable (other than a type object) that accepts an exception as its single argument and returns true for the exceptions that should be in the subgroup.

The nesting structure of the current exception is preserved in the result, as are the values of its **`message`**, **`__traceback__`**, **`__cause__`**, **`__context__`** and **`__notes__`** fields. Empty nested groups are omitted from the result.

The condition is checked for all exceptions in the nested exception group, including the top-level and any nested exception groups. If the condition is true for such an exception group, it is included in the result in full.

Added in version 3.13: `condition` can be any callable which is not a type object.

split(*condition*)

Like `subgroup()`, but returns the pair `(match, rest)` where `match` is `subgroup(condition)` and `rest` is the remaining non-matching part.

derive(*excs*)

Returns an exception group with the same `message`, but which wraps the exceptions in `excs`.

This method is used by `subgroup()` and `split()`, which are used in various contexts to break up an exception group. A subclass needs to override it in order to make `subgroup()` and `split()` return instances of the subclass rather than `ExceptionGroup`.

`subgroup()` and `split()` copy the `__traceback__`, `__cause__`, `__context__` and `__notes__` fields from the original exception group to the one returned by `derive()`, so these fields do not need to be updated by `derive()`.

```
>>> class MyGroup(ExceptionGroup):
...     def derive(self, excs):
...         return MyGroup(self.message, excs)
...
>>> e = MyGroup("eg", [ValueError(1), TypeError(2)])
>>> e.add_note("a note")
>>> e.__context__ = Exception("context")
>>> e.__cause__ = Exception("cause")
>>> try:
...    raise e
... except Exception as e:
...    exc = e
...
>>> match, rest = exc.split(ValueError)
>>> exc, exc.__context__, exc.__cause__, exc.__notes__
(MyGroup('eg', [ValueError(1), TypeError(2)]),
Exception('context'), Exception('cause'), ['a note'])
>>> match, match.__context__, match.__cause__,
match.__notes__
(MyGroup('eg', [ValueError(1)]), Exception('context'),
Exception('cause'), ['a note'])
```

```
>>> rest, rest.__context__, rest.__cause__,
rest.__notes__
(MyGroup('eg', [TypeError(2)]), Exception('context'),
Exception('cause'), ['a note'])
>>> exc.__traceback__ is match.__traceback__ is
rest.__traceback__
True
```

Note that **`BaseExceptionGroup`** defines `__new__()`, so subclasses that need a different constructor signature need to override that rather than `__init__()`. For example, the following defines an exception group subclass which accepts an exit_code and constructs the group's message from it.

```
class Errors(ExceptionGroup):
   def __new__(cls, errors, exit_code):
      self = super().__new__(Errors, f"exit code:
{exit_code}", errors)
      self.exit_code = exit_code
      return self

   def derive(self, excs):
      return Errors(excs, self.exit_code)
```

Like **`ExceptionGroup`**, any subclass of **`BaseExceptionGroup`** which is also a subclass of **`Exception`** can only wrap instances of **`Exception`**.

Added in version 3.11.

Exception hierarchy

The class hierarchy for built-in exceptions is:

```
BaseException
 ├── BaseExceptionGroup
 ├── GeneratorExit
 ├── KeyboardInterrupt
 ├── SystemExit
```

```
└── Exception
     ├── ArithmeticError
     │    ├── FloatingPointError
     │    ├── OverflowError
     │    └── ZeroDivisionError
     ├── AssertionError
     ├── AttributeError
     ├── BufferError
     ├── EOFError
     ├── ExceptionGroup [BaseExceptionGroup]
     ├── ImportError
     │    └── ModuleNotFoundError
     ├── LookupError
     │    ├── IndexError
     │    └── KeyError
     ├── MemoryError
     ├── NameError
     │    └── UnboundLocalError
     ├── OSError
     │    ├── BlockingIOError
     │    ├── ChildProcessError
     │    ├── ConnectionError
     │    │    ├── BrokenPipeError
     │    │    ├── ConnectionAbortedError
     │    │    ├── ConnectionRefusedError
     │    │    └── ConnectionResetError
     │    ├── FileExistsError
     │    ├── FileNotFoundError
     │    ├── InterruptedError
     │    ├── IsADirectoryError
     │    ├── NotADirectoryError
     │    ├── PermissionError
     │    ├── ProcessLookupError
     │    └── TimeoutError
     ├── ReferenceError
     ├── RuntimeError
     │    ├── NotImplementedError
     │    ├── PythonFinalizationError
     │    └── RecursionError
     ├── StopAsyncIteration
     ├── StopIteration
     ├── SyntaxError
     │    └── IndentationError
     │
```

```
│         └── TabError
├── SystemError
├── TypeError
├── ValueError
│    └── UnicodeError
│         ├── UnicodeDecodeError
│         ├── UnicodeEncodeError
│         └── UnicodeTranslateError
└── Warning
     ├── BytesWarning
     ├── DeprecationWarning
     ├── EncodingWarning
     ├── FutureWarning
     ├── ImportWarning
     ├── PendingDeprecationWarning
     ├── ResourceWarning
     ├── RuntimeWarning
     ├── SyntaxWarning
     ├── UnicodeWarning
     └── UserWarning
```

Text Processing Services

The modules described in this chapter provide a wide range of string manipulation operations and other text processing services.

The `codecs` module described under Binary Data Services is also highly relevant to text processing. In addition, see the documentation for Python’s built-in string type in Text Sequence Type — str.

- `string` — Common string operations
    - String constants
    - Custom String Formatting
    - Format String Syntax
        - Format Specification Mini-Language
        - Format examples
    - Template strings ($-strings)
    - Helper functions
- `string.templatelib` — Support for template string literals
    - Template strings
    - Types
    - Helper functions
- `re` — Regular expression operations
    - Regular Expression Syntax
    - Module Contents
        - Flags
        - Functions
        - Exceptions
    - Regular Expression Objects
    - Match Objects
    - Regular Expression Examples
        - Checking for a Pair
        - Simulating scanf()

        - search() vs. match()
        - Making a Phonebook
        - Text Munging
        - Finding all Adverbs
        - Finding all Adverbs and their Positions
        - Raw String Notation
        - Writing a Tokenizer
- **`difflib`** — Helpers for computing deltas
    - SequenceMatcher Objects
    - SequenceMatcher Examples
    - Differ Objects
    - Differ Example
    - A command-line interface to difflib
    - ndiff example
- **`textwrap`** — Text wrapping and filling
- **`unicodedata`** — Unicode Database
- **`stringprep`** — Internet String Preparation
- **`readline`** — GNU readline interface
    - Init file
    - Line buffer
    - History file
    - History list
    - Startup hooks
    - Completion
    - Example
- **`rlcompleter`** — Completion function for GNU readline

`string` — Common string operations

Source code: Lib/string/__init__.py
[https://github.com/python/cpython/tree/3.14/Lib/string/__init__.py]

See also

Text Sequence Type — str

String Methods

String constants

The constants defined in this module are:

string.ascii_letters
The concatenation of the **`ascii_lowercase`** and **`ascii_uppercase`** constants described below. This value is not locale-dependent.

string.ascii_lowercase
The lowercase letters `'abcdefghijklmnopqrstuvwxyz'`. This value is not locale-dependent and will not change.

string.ascii_uppercase
The uppercase letters `'ABCDEFGHIJKLMNOPQRSTUVWXYZ'`. This value is not locale-dependent and will not change.

string.digits
The string `'0123456789'`.

string.hexdigits

The string `'0123456789abcdefABCDEF'`.

string.octdigits

The string `'01234567'`.

string.punctuation

String of ASCII characters which are considered punctuation characters in the `C` locale: `!"#$%&'()*+,-./:;<=>?@[\]^_`{|}~`.

string.printable

String of ASCII characters which are considered printable by Python. This is a combination of **`digits`**, **`ascii_letters`**, **`punctuation`**, and **`whitespace`**.

Note

By design, **`string.printable.isprintable()`** returns **`False`**. In particular, `string.printable` is not printable in the POSIX sense (see *LC_CTYPE* *[https://manpages.debian.org/locale(5)]*).

string.whitespace

A string containing all ASCII characters that are considered whitespace. This includes the characters space, tab, linefeed, return, formfeed, and vertical tab.

Custom String Formatting

The built-in string class provides the ability to do complex variable substitutions and value formatting via the **`format()`** method described in **PEP 3101** [https://peps.python.org/pep-3101/]. The **`Formatter`** class in the **`string`** module allows you to create and customize your own string formatting behaviors using the same implementation as the built-in **`format()`** method.

class string.Formatter

The **`Formatter`** class has the following public methods:

format(*format_string*, */*, **args*, ***kwargs*)

The primary API method. It takes a format string and an arbitrary set of positional and keyword arguments. It is just a wrapper that calls **`vformat()`**.

Changed in version 3.7: A format string argument is now positional-only.

vformat(*format_string*, *args*, *kwargs*)

This function does the actual work of formatting. It is exposed as a separate function for cases where you want to pass in a predefined dictionary of arguments, rather than unpacking and repacking the dictionary as individual arguments using the `*args` and `**kwargs` syntax. **`vformat()`** does the work of breaking up the format string into character data and replacement fields. It calls the various methods described below.

In addition, the **`Formatter`** defines a number of methods that are intended to be replaced by subclasses:

parse(*format_string*)

Loop over the format_string and return an iterable of tuples (*literal_text*, *field_name*, *format_spec*, *conversion*). This is used by **`vformat()`** to break the string into either literal text, or replacement fields.

The values in the tuple conceptually represent a span of literal text followed by a single replacement field. If there is no literal text (which can happen if two replacement fields occur consecutively), then *literal_text* will be a zero-length string. If there is no replacement field, then the values of *field_name*, *format_spec* and *conversion* will be `None`. The value of *field_name* is unmodified and auto-numbering of non-numbered positional fields is done by `vformat()`.

get_field(*field_name*, *args*, *kwargs*)

Given *field_name*, convert it to an object to be formatted. Auto-numbering of *field_name* returned from `parse()` is done by `vformat()` before calling this method. Returns a tuple (obj, used_key). The default version takes strings of the form defined in **PEP 3101** [https://peps.python.org/pep-3101/], such as “0[name]” or “label.title”. *args* and *kwargs* are as passed in to `vformat()`. The return value *used_key* has the same meaning as the *key* parameter to `get_value()`.

get_value(*key*, *args*, *kwargs*)

Retrieve a given field value. The *key* argument will be either an integer or a string. If it is an integer, it represents the index of the positional argument in *args*; if it is a string, then it represents a named argument in *kwargs*.

The *args* parameter is set to the list of positional arguments to `vformat()`, and the *kwargs* parameter is set to the dictionary of keyword arguments.

For compound field names, these functions are only called for the first component of the field name; subsequent components are handled through normal attribute and indexing operations.

So for example, the field expression ‘0.name’ would cause `get_value()` to be called with a *key* argument of 0. The `name` attribute will be looked up after `get_value()` returns by calling the built-in `getattr()` function.

If the index or keyword refers to an item that does not exist, then an `IndexError` or `KeyError` should be raised.

check_unused_args(*used_args*, *args*, *kwargs*)

Implement checking for unused arguments if desired. The arguments to this function is the set of all argument keys that were actually referred to in the format string (integers for positional arguments, and strings for named arguments), and a reference to the *args* and *kwargs* that was passed to vformat. The set of unused args can be calculated from these parameters. `check_unused_args()` is assumed to raise an exception if the check fails.

format_field(*value*, *format_spec*)

`format_field()` simply calls the global `format()` built-in. The method is provided so that subclasses can override it.

convert_field(*value*, *conversion*)

Converts the value (returned by `get_field()`) given a conversion type (as in the tuple returned by the `parse()` method). The default version understands ‘s’ (str), ‘r’ (repr) and ‘a’ (ascii) conversion types.

Format String Syntax

The `str.format()` method and the `Formatter` class share the same syntax for format strings (although in the case of `Formatter`, subclasses can define their own format string syntax). The syntax is related to that of formatted string literals and template string literals, but it is less sophisticated and, in particular, does not support arbitrary expressions in interpolations.

Format strings contain “replacement fields” surrounded by curly braces `{}`. Anything that is not contained in braces is considered literal text, which is copied unchanged to the output. If you need to include a brace character in the literal text, it can be escaped by doubling: `{{` and `}}`.

The grammar for a replacement field is as follows:

```
replacement_field: "{" [field_name] ["!" conversion] [":"
format_spec] "}"
field_name:        arg_name ("." attribute_name | "["
element_index "]")*
arg_name:          [identifier | digit+]
attribute_name:    identifier
element_index:     digit+ | index_string
index_string:      <any source character except "]"> +
conversion:        "r" | "s" | "a"
format_spec:       format-spec:format_spec
```

In less formal terms, the replacement field can start with a *field_name* that specifies the object whose value is to be formatted and inserted into the output instead of the replacement field. The *field_name* is optionally followed by a *conversion* field, which is preceded by an exclamation point `'!'`, and a *format_spec*, which is preceded by a colon `':'`. These specify a non-default format for the replacement value.

See also the Format Specification Mini-Language section.

The *field_name* itself begins with an *arg_name* that is either a number or a keyword. If it’s a number, it refers to a positional argument, and if it’s a keyword, it refers to a named keyword argument. An *arg_name* is treated as a number if a call to `str.isdecimal()` on the string would return true. If the numerical arg_names in a format string are 0, 1, 2, … in sequence, they can all be omitted (not just some) and the numbers 0, 1, 2, … will be automatically inserted in that order. Because *arg_name* is not quote-delimited, it is not possible to specify arbitrary dictionary keys (e.g., the strings `'10'` or `':-]'`) within a format string. The *arg_name* can be followed by any number of index or attribute expressions. An expression of the form `'.name'` selects the named attribute using [`getattr()`], while an expression of the form `'[index]'` does an index lookup using [`__getitem__()`].

Changed in version 3.1: The positional argument specifiers can be omitted for [`str.format()`], so `'{} {}'.format(a, b)` is equivalent to `'{0} {1}'.format(a, b)`.

Changed in version 3.4: The positional argument specifiers can be omitted for [`Formatter`].

Some simple format string examples:

```
"First, thou shalt count to {0}"  # References first positional
argument
"Bring me a {}"                   # Implicitly references the
first positional argument
"From {} to {}"                   # Same as "From {0} to {1}"
"My quest is {name}"              # References keyword argument
'name'
"Weight in tons {0.weight}"       # 'weight' attribute of first
positional arg
"Units destroyed: {players[0]}"   # First element of keyword
argument 'players'.
```

The *conversion* field causes a type coercion before formatting. Normally, the job of formatting a value is done by the [`__format__()`] method of the value itself. However, in some cases it is desirable to force a type to be formatted as a string, overriding its own definition of formatting. By converting the value to a string before calling [`__format__()`], the normal formatting logic is bypassed.

Three conversion flags are currently supported: `'!s'` which calls **`str()`** on the value, `'!r'` which calls **`repr()`** and `'!a'` which calls **`ascii()`**.

Some examples:

```
"Harold's a clever {0!s}"        # Calls str() on the argument
first
"Bring out the holy {name!r}"    # Calls repr() on the argument
first
"More {!a}"                      # Calls ascii() on the
argument first
```

The *format_spec* field contains a specification of how the value should be presented, including such details as field width, alignment, padding, decimal precision and so on. Each value type can define its own “formatting mini-language” or interpretation of the *format_spec*.

Most built-in types support a common formatting mini-language, which is described in the next section.

A *format_spec* field can also include nested replacement fields within it. These nested replacement fields may contain a field name, conversion flag and format specification, but deeper nesting is not allowed. The replacement fields within the format_spec are substituted before the *format_spec* string is interpreted. This allows the formatting of a value to be dynamically specified.

See the Format examples section for some examples.

Format Specification Mini-Language

“Format specifications” are used within replacement fields contained within a format string to define how individual values are presented (see Format String Syntax, f-strings, and t-strings). They can also be passed directly to the built-in `format()` function. Each formattable type may define how the format specification is to be interpreted.

Most built-in types implement the following options for format specifications, although some of the formatting options are only supported by the numeric types.

A general convention is that an empty format specification produces the same result as if you had called `str()` on the value. A non-empty format specification typically modifies the result.

The general form of a *standard format specifier* is:

```
format_spec:             [options][width_and_precision][type]
options:                 [[fill]align][sign]["z"]["#"]["0"]
fill:                    <any character>
align:                   "<" | ">" | "=" | "^"
sign:                    "+" | "-" | " "
width_and_precision:     [width_with_grouping]
[precision_with_grouping]
width_with_grouping:     [width][grouping]
precision_with_grouping: "." [precision][grouping] | "."
grouping
width:                   digit+
precision:               digit+
grouping:                "," | "_"
type:                    "b" | "c" | "d" | "e" | "E" | "f" |
"F" | "g"
                         | "G" | "n" | "o" | "s" | "x" | "X" |
"%"
```

If a valid *align* value is specified, it can be preceded by a *fill* character that can be any character and defaults to a space if omitted. It is not possible to use a literal curly brace (”`{`” or “`}`”) as the *fill* character in a formatted string literal or when using the `str.format()` method. However, it is possible to insert a curly brace with a nested replacement field. This limitation doesn't affect the `format()` function.

The meaning of the various alignment options is as follows:

Option	Meaning
`'<'`	Forces the field to be left-aligned within the available space (this is the default for most objects).
`'>'`	Forces the field to be right-aligned within the available space (this is the default for numbers).
`'='`	Forces the padding to be placed after the sign (if any) but before the digits. This is used for printing fields in the form '+000000120'. This alignment option is only valid for numeric types, excluding `complex`. It becomes the default for numbers when '0' immediately precedes the field width.
`'^'`	Forces the field to be centered within the available space.

Note that unless a minimum field width is defined, the field width will always be the same size as the data to fill it, so that the alignment option has no meaning in this case.

The *sign* option is only valid for number types, and can be one of the following:

Option	Meaning
`'+'`	Indicates that a sign should be used for both positive as well as negative numbers.
`'-'`	Indicates that a sign should be used only for negative numbers (this is the default behavior).
space	Indicates that a leading space should be used on positive numbers, and a minus sign on negative numbers.

The `'z'` option coerces negative zero floating-point values to positive zero after rounding to the format precision. This option is only valid for floating-point presentation types.

Changed in version 3.11: Added the `'z'` option (see also **PEP 682** [https://peps.python.org/pep-0682/]).

The `'#'` option causes the “alternate form” to be used for the conversion. The alternate form is defined differently for different types. This option is only valid for integer, float and complex types. For integers, when binary, octal, or hexadecimal output is used, this option adds the respective prefix `'0b'`, `'0o'`, `'0x'`, or `'0X'` to the output value. For float and complex the alternate form causes the result of the conversion to always contain a decimal-point character, even if no digits follow it. Normally, a decimal-point character appears in the result of these conversions only if a digit follows it. In addition, for `'g'` and `'G'` conversions, trailing zeros are not removed from the result.

The *width* is a decimal integer defining the minimum total field width, including any prefixes, separators, and other formatting characters. If not specified, then the field width will be determined by the content.

When no explicit alignment is given, preceding the *width* field by a zero (`'0'`) character enables sign-aware zero-padding for numeric types, excluding `complex`. This is equivalent to a *fill* character of `'0'` with an *alignment* type of `'='`.

Changed in version 3.10: Preceding the *width* field by `'0'` no longer affects the default alignment for strings.

The *precision* is a decimal integer indicating how many digits should be displayed after the decimal point for presentation types `'f'` and `'F'`, or before and after the decimal point for presentation types `'g'` or `'G'`. For string presentation types the field indicates the maximum field size - in other words, how many characters will be used from the field content. The *precision* is not allowed for integer presentation types.

The *grouping* option after *width* and *precision* fields specifies a digit group separator for the integral and fractional parts of a number respectively. It can be one of the following:

Option	**Meaning**
`','`	Inserts a comma every 3 digits for integer presentation type `'d'` and floating-point presentation types, excluding `'n'`. For other presentation types, this option is not supported.

Option	Meaning
`'_'`	Inserts an underscore every 3 digits for integer presentation type `'d'` and floating-point presentation types, excluding `'n'`. For integer presentation types `'b'`, `'o'`, `'x'`, and `'X'`, underscores are inserted every 4 digits. For other presentation types, this option is not supported.

For a locale aware separator, use the `'n'` presentation type instead.

Changed in version 3.1: Added the `','` option (see also **PEP 378** [https://peps.python.org/pep-0378/]).

Changed in version 3.6: Added the `'_'` option (see also **PEP 515** [https://peps.python.org/pep-0515/]).

Changed in version 3.14: Support the *grouping* option for the fractional part.

Finally, the *type* determines how the data should be presented.

The available string presentation types are:

Type	Meaning
`'s'`	String format. This is the default type for strings and may be omitted.
None	The same as `'s'`.

Type	Meaning

The available integer presentation types are:

Type	Meaning
`'b'`	Binary format. Outputs the number in base 2.
`'c'`	Character. Converts the integer to the corresponding unicode character before printing.
`'d'`	Decimal Integer. Outputs the number in base 10.
`'o'`	Octal format. Outputs the number in base 8.
`'x'`	Hex format. Outputs the number in base 16, using lower-case letters for the digits above 9.
`'X'`	Hex format. Outputs the number in base 16, using upper-case letters for the digits above 9. In case `'#'` is specified, the prefix `'0x'` will be upper-cased to `'0X'` as well.
`'n'`	Number. This is the same as `'d'`, except that it uses the current locale setting to insert the appropriate digit group separators.

Type	Meaning
None	The same as `'d'`.

In addition to the above presentation types, integers can be formatted with the floating-point presentation types listed below (except `'n'` and `None`). When doing so, `float()` is used to convert the integer to a floating-point number before formatting.

The available presentation types for `float` and `Decimal` values are:

Type	Meaning
`'e'`	Scientific notation. For a given precision `p`, formats the number in scientific notation with the letter 'e' separating the coefficient from the exponent. The coefficient has one digit before and `p` digits after the decimal point, for a total of `p + 1` significant digits. With no precision given, uses a precision of `6` digits after the decimal point for `float`, and shows all coefficient digits for `Decimal`. If `p=0`, the decimal point is omitted unless the `#` option is used. For `float`, the exponent always contains at least two digits, and is zero if the value is zero.
`'E'`	Scientific notation. Same as `'e'` except it uses an upper case 'E' as the separator character.

Type	Meaning
`'f'`	Fixed-point notation. For a given precision `p`, formats the number as a decimal number with exactly `p` digits following the decimal point. With no precision given, uses a precision of `6` digits after the decimal point for **`float`**, and uses a precision large enough to show all coefficient digits for **`Decimal`**. If `p=0`, the decimal point is omitted unless the `#` option is used.
`'F'`	Fixed-point notation. Same as `'f'`, but converts `nan` to `NAN` and `inf` to `INF`.
`'g'`	General format. For a given precision `p >= 1`, this rounds the number to `p` significant digits and then formats the result in either fixed-point format or in scientific notation, depending on its magnitude. A precision of `0` is treated as equivalent to a precision of `1`. The precise rules are as follows: suppose that the result formatted with presentation type `'e'` and precision `p-1` would have exponent `exp`. Then, if `m <= exp < p`, where `m` is -4 for floats and -6 for **`Decimals`**, the number is formatted with presentation type `'f'` and precision `p-1-exp`. Otherwise, the number is formatted with presentation type `'e'` and precision `p-1`. In both cases insignificant trailing zeros are removed from the significand, and the decimal point is

Type	Meaning
	also removed if there are no remaining digits following it, unless the `'#'` option is used. With no precision given, uses a precision of `6` significant digits for `float`. For `Decimal`, the coefficient of the result is formed from the coefficient digits of the value; scientific notation is used for values smaller than `1e-6` in absolute value and values where the place value of the least significant digit is larger than 1, and fixed-point notation is used otherwise. Positive and negative infinity, positive and negative zero, and nans, are formatted as `inf`, `-inf`, `0`, `-0` and `nan` respectively, regardless of the precision.
`'G'`	General format. Same as `'g'` except switches to `'E'` if the number gets too large. The representations of infinity and NaN are uppercased, too.
`'n'`	Number. This is the same as `'g'`, except that it uses the current locale setting to insert the appropriate digit group separators for the integral part of a number.
`'%'`	Percentage. Multiplies the number by 100 and displays in fixed (`'f'`) format, followed by a percent sign.

Type	Meaning
None	For **`float`** this is like the `'g'` type, except that when fixed-point notation is used to format the result, it always includes at least one digit past the decimal point, and switches to the scientific notation when `exp >= p - 1`. When the precision is not specified, the latter will be as large as needed to represent the given value faithfully. For **`Decimal`**, this is the same as either `'g'` or `'G'` depending on the value of `context.capitals` for the current decimal context. The overall effect is to match the output of **`str()`** as altered by the other format modifiers.

The result should be correctly rounded to a given precision `p` of digits after the decimal point. The rounding mode for **`float`** matches that of the **`round()`** builtin. For **`Decimal`**, the rounding mode of the current context will be used.

The available presentation types for **`complex`** are the same as those for **`float`** (`'%'` is not allowed). Both the real and imaginary components of a complex number are formatted as floating-point numbers, according to the specified presentation type. They are separated by the mandatory sign of the imaginary part, the latter being terminated by a `j` suffix. If the presentation type is missing, the result will match the output of **`str()`** (complex numbers with a non-zero real part are also surrounded by parentheses), possibly altered by other format modifiers.

Format examples

This section contains examples of the `str.format()` syntax and comparison with the old `%`-formatting.

In most of the cases the syntax is similar to the old `%`-formatting, with the addition of the `{}` and with `:` used instead of `%`. For example, `'%03.2f'` can be translated to `'{:03.2f}'`.

The new format syntax also supports new and different options, shown in the following examples.

Accessing arguments by position:

```
>>> '{0}, {1}, {2}'.format('a', 'b', 'c')
'a, b, c'
>>> '{}, {}, {}'.format('a', 'b', 'c')  # 3.1+ only
'a, b, c'
>>> '{2}, {1}, {0}'.format('a', 'b', 'c')
'c, b, a'
>>> '{2}, {1}, {0}'.format(*'abc')      # unpacking argument
sequence
'c, b, a'
>>> '{0}{1}{0}'.format('abra', 'cad')   # arguments' indices
can be repeated
'abracadabra'
```

Accessing arguments by name:

```
>>> 'Coordinates: {latitude},
{longitude}'.format(latitude='37.24N', longitude='-115.81W')
'Coordinates: 37.24N, -115.81W'
>>> coord = {'latitude': '37.24N', 'longitude': '-115.81W'}
>>> 'Coordinates: {latitude}, {longitude}'.format(**coord)
'Coordinates: 37.24N, -115.81W'
```

Accessing arguments’ attributes:

```
>>> c = 3-5j
>>> ('The complex number {0} is formed from the real part
{0.real} '
...  'and the imaginary part {0.imag}.').format(c)
```

```
'The complex number (3-5j) is formed from the real part 3.0 and
the imaginary part -5.0.'
>>> class Point:
...     def __init__(self, x, y):
...         self.x, self.y = x, y
...     def __str__(self):
...         return 'Point({self.x},
{self.y})'.format(self=self)
...
>>> str(Point(4, 2))
'Point(4, 2)'
```

Accessing arguments’ items:

```
>>> coord = (3, 5)
>>> 'X: {0[0]};  Y: {0[1]}'.format(coord)
'X: 3;  Y: 5'
```

Replacing `%s` and `%r`:

```
>>> "repr() shows quotes: {!r}; str() doesn't:
{!s}".format('test1', 'test2')
"repr() shows quotes: 'test1'; str() doesn't: test2"
```

Aligning the text and specifying a width:

```
>>> '{:<30}'.format('left aligned')
'left aligned                  '
>>> '{:>30}'.format('right aligned')
'                 right aligned'
>>> '{:^30}'.format('centered')
'           centered           '
>>> '{:*^30}'.format('centered')  # use '*' as a fill char
'***********centered***********'
```

Replacing `%+f`, `%-f`, and `% f` and specifying a sign:

```
>>> '{:+f}; {:+f}'.format(3.14, -3.14)  # show it always
'+3.140000; -3.140000'
>>> '{: f}; {: f}'.format(3.14, -3.14)  # show a space for
positive numbers
' 3.140000; -3.140000'
```

```
>>> '{:-f}; {:-f}'.format(3.14, -3.14)  # show only the minus -
- same as '{:f}; {:f}'
'3.140000; -3.140000'
```

Replacing `%x` and `%o` and converting the value to different bases:

```
>>> # format also supports binary numbers
>>> "int: {0:d};  hex: {0:x};  oct: {0:o};  bin:
{0:b}".format(42)
'int: 42;  hex: 2a;  oct: 52;  bin: 101010'
>>> # with 0x, 0o, or 0b as prefix:
>>> "int: {0:d};  hex: {0:#x};  oct: {0:#o};  bin:
{0:#b}".format(42)
'int: 42;  hex: 0x2a;  oct: 0o52;  bin: 0b101010'
```

Using the comma or the underscore as a digit group separator:

```
>>> '{:,}'.format(1234567890)
'1,234,567,890'
>>> '{:_}'.format(1234567890)
'1_234_567_890'
>>> '{:_b}'.format(1234567890)
'100_1001_1001_0110_0000_0010_1101_0010'
>>> '{:_x}'.format(1234567890)
'4996_02d2'
>>> '{:_}'.format(123456789.123456789)
'123_456_789.12345679'
>>> '{:.,}'.format(123456789.123456789)
'123456789.123,456,79'
>>> '{:,._}'.format(123456789.123456789)
'123,456,789.123_456_79'
```

Expressing a percentage:

```
>>> points = 19
>>> total = 22
>>> 'Correct answers: {:.2%}'.format(points/total)
'Correct answers: 86.36%'
```

Using type-specific formatting:

```
>>> import datetime
>>> d = datetime.datetime(2010, 7, 4, 12, 15, 58)
>>> '{:%Y-%m-%d %H:%M:%S}'.format(d)
'2010-07-04 12:15:58'
```

Nesting arguments and more complex examples:

```
>>> for align, text in zip('<^>', ['left', 'center', 'right']):
...     '{0:{fill}{align}16}'.format(text, fill=align,
align=align)
...
'left<<<<<<<<<<<<'
'^^^^^center^^^^^'
'>>>>>>>>>>>right'
>>>
>>> octets = [192, 168, 0, 1]
>>> '{:02X}{:02X}{:02X}{:02X}'.format(*octets)
'C0A80001'
>>> int(_, 16)
3232235521
>>>
>>> width = 5
>>> for num in range(5,12):
...     for base in 'dXob':
...         print('{0:{width}{base}}'.format(num, base=base,
width=width), end=' ')
...     print()
...
    5     5     5   101
    6     6     6   110
    7     7     7   111
    8     8    10  1000
    9     9    11  1001
   10     A    12  1010
   11     B    13  1011
```

Template strings ($-strings)

Note

The feature described here was introduced in Python 2.4; a simple templating method based upon regular expressions. It predates **`str.format()`**, formatted string literals, and template string literals.

It is unrelated to template string literals (t-strings), which were introduced in Python 3.14. These evaluate to **`string.templatelib.Template`** objects, found in the **`string.templatelib`** module.

Template strings provide simpler string substitutions as described in **PEP 292** [https://peps.python.org/pep-0292/]. A primary use case for template strings is for internationalization (i18n) since in that context, the simpler syntax and functionality makes it easier to translate than other built-in string formatting facilities in Python. As an example of a library built on template strings for i18n, see the flufl.i18n [https://flufli18n.readthedocs.io/en/latest/] package.

Template strings support `$`-based substitutions, using the following rules:

- `$$` is an escape; it is replaced with a single `$`.
- `$identifier` names a substitution placeholder matching a mapping key of `"identifier"`. By default, `"identifier"` is restricted to any case-insensitive ASCII alphanumeric string (including underscores) that starts with an underscore or ASCII letter. The first non-identifier character after the `$` character terminates this placeholder specification.
- `${identifier}` is equivalent to `$identifier`. It is required when valid identifier characters follow the placeholder but are not part of the placeholder, such as `"${noun}ification"`.

Any other appearance of `$` in the string will result in a `ValueError` being raised.

The `string` module provides a `Template` class that implements these rules. The methods of `Template` are:

class string.Template(*template*)

The constructor takes a single argument which is the template string.

substitute(*mapping={}*, */*, ***kwds*)

Performs the template substitution, returning a new string. *mapping* is any dictionary-like object with keys that match the placeholders in the template. Alternatively, you can provide keyword arguments, where the keywords are the placeholders. When both *mapping* and *kwds* are given and there are duplicates, the placeholders from *kwds* take precedence.

safe_substitute(*mapping={}*, */*, ***kwds*)

Like `substitute()`, except that if placeholders are missing from *mapping* and *kwds*, instead of raising a `KeyError` exception, the original placeholder will appear in the resulting string intact. Also, unlike with `substitute()`, any other appearances of the `$` will simply return `$` instead of raising `ValueError`.

While other exceptions may still occur, this method is called “safe” because it always tries to return a usable string instead of raising an exception. In another sense, `safe_substitute()` may be anything other than safe, since it will silently ignore malformed templates containing dangling delimiters, unmatched braces, or placeholders that are not valid Python identifiers.

is_valid()

Returns `False` if the template has invalid placeholders that will cause `substitute()` to raise `ValueError`.

Added in version 3.11.

get_identifiers()

Returns a list of the valid identifiers in the template, in the order they first appear, ignoring any invalid identifiers.

Added in version 3.11.

`Template` instances also provide one public data attribute:

template

This is the object passed to the constructor's *template* argument. In general, you shouldn't change it, but read-only access is not enforced.

Here is an example of how to use a Template:

```
>>> from string import Template
>>> s = Template('$who likes $what')
>>> s.substitute(who='tim', what='kung pao')
'tim likes kung pao'
>>> d = dict(who='tim')
>>> Template('Give $who $100').substitute(d)
Traceback (most recent call last):
...
ValueError: Invalid placeholder in string: line 1, col 11
>>> Template('$who likes $what').substitute(d)
Traceback (most recent call last):
...
KeyError: 'what'
>>> Template('$who likes $what').safe_substitute(d)
'tim likes $what'
```

Advanced usage: you can derive subclasses of `Template` to customize the placeholder syntax, delimiter character, or the entire regular expression used to parse template strings. To do this, you can override these class attributes:

- *delimiter* – This is the literal string describing a placeholder introducing delimiter. The default value is `$`. Note that this should *not* be a regular expression, as the implementation will call `re.escape()` on this string as needed. Note further that you cannot change the delimiter after class creation (i.e. a different delimiter must be set in the subclass's class namespace).

- *idpattern* – This is the regular expression describing the pattern for non-braced placeholders. The default value is the regular expression `(?a:[_a-z][_a-z0-9]*)`. If this is given and *braceidpattern* is `None` this pattern will also apply to braced placeholders.

  Note

  Since default *flags* is `re.IGNORECASE`, pattern `[a-z]` can match with some non-ASCII characters. That's why we use the local `a` flag here.

  Changed in version 3.7: braceidpattern can be used to define separate patterns used inside and outside the braces.

- *braceidpattern* – This is like *idpattern* but describes the pattern for braced placeholders. Defaults to `None` which means to fall back to *idpattern* (i.e. the same pattern is used both inside and outside braces). If given, this allows you to define different patterns for braced and unbraced placeholders.

  Added in version 3.7.

- *flags* – The regular expression flags that will be applied when compiling the regular expression used for recognizing substitutions. The default value is `re.IGNORECASE`. Note that `re.VERBOSE` will always be added to the flags, so custom *idpattern*s must follow conventions for verbose regular expressions.

  Added in version 3.2.

Alternatively, you can provide the entire regular expression pattern by overriding the class attribute *pattern*. If you do this, the value must be a regular expression object with four named capturing groups. The capturing groups correspond to the rules given above, along with the invalid placeholder rule:

- *escaped* – This group matches the escape sequence, e.g. `$$`, in the default pattern.
- *named* – This group matches the unbraced placeholder name; it should not include the delimiter in capturing group.
- *braced* – This group matches the brace enclosed placeholder name; it should not include either the delimiter or braces in the capturing group.
- *invalid* – This group matches any other delimiter pattern (usually a single delimiter), and it should appear last in the regular expression.

The methods on this class will raise `ValueError` if the pattern matches the template without one of these named groups matching.

Helper functions

string.capwords(*s*, *sep=None*)

Split the argument into words using **`str.split()`**, capitalize each word using **`str.capitalize()`**, and join the capitalized words using **`str.join()`**. If the optional second argument *sep* is absent or `None`, runs of whitespace characters are replaced by a single space and leading and trailing whitespace are removed, otherwise *sep* is used to split and join the words.

`string.templatelib` — Support for template string literals

Source code: Lib/string/templatelib.py [https://github.com/python/cpython/tree/3.14/Lib/string/templatelib.py]

See also

- Format strings
- Template string literal (t-string) syntax
- **PEP 750** [https://peps.python.org/pep-0750/]

Template strings

Added in version 3.14.

Template strings are a mechanism for custom string processing. They have the full flexibility of Python’s f-strings, but return a **`Template`** instance that gives access to the static and interpolated (in curly brackets) parts of a string *before* they are combined.

To write a t-string, use a `'t'` prefix instead of an `'f'`, like so:

```
>>> pi = 3.14
>>> t't-strings are new in Python {pi!s}!'
Template(
   strings=('t-strings are new in Python ', '!'),
   interpolations=(Interpolation(3.14, 'pi', 's', ''),)
)
```

Types

class string.templatelib.Template

The **`Template`** class describes the contents of a template string. It is immutable, meaning that attributes of a template cannot be reassigned.

The most common way to create a **`Template`** instance is to use the template string literal syntax. This syntax is identical to that of f-strings, except that it uses a `t` prefix in place of an `f`:

```
>>> cheese = 'Red Leicester'
>>> template = t"We're fresh out of {cheese}, sir."
>>> type(template)
<class 'string.templatelib.Template'>
```

Templates are stored as sequences of literal **`strings`** and dynamic **`interpolations`**. A **`values`** attribute holds the values of the interpolations:

```
>>> cheese = 'Camembert'
>>> template = t'Ah! We do have {cheese}.'
>>> template.strings
('Ah! We do have ', '.')
>>> template.interpolations
(Interpolation('Camembert', ...),)
>>> template.values
('Camembert',)
```

The **`strings`** tuple has one more element than **`interpolations`** and **`values`**; the interpolations “belong” between the strings. This may be easier to understand when tuples are aligned

```
template.strings:  ('Ah! We do have ',              '.')
template.values:   (                   'Camembert',    )
```

Attributes

strings*: tuple[str, ...]*

A **`tuple`** of the static strings in the template.

```
>>> cheese = 'Camembert'
>>> template = t'Ah! We do have {cheese}.'
>>> template.strings
('Ah! We do have ', '.')
```

Empty strings *are* included in the tuple:

```
>>> response = 'We do have '
>>> cheese = 'Camembert'
>>> template = t'Ah! {response}{cheese}.'
>>> template.strings
('Ah! ', '', '.')
```

The `strings` tuple is never empty, and always contains one more string than the `interpolations` and `values` tuples:

```
>>> t''.strings
('',)
>>> t''.values
()
>>> t'{'cheese'}'.strings
('', '')
>>> t'{'cheese'}'.values
('cheese',)
```

interpolations*: tuple[Interpolation, ...]*

A **`tuple`** of the interpolations in the template.

```
>>> cheese = 'Camembert'
>>> template = t'Ah! We do have {cheese}.'
>>> template.interpolations
(Interpolation('Camembert', 'cheese', None, ''),)
```

The `interpolations` tuple may be empty and always contains one fewer values than the `strings` tuple:

```
>>> t'Red Leicester'.interpolations
()
```

values*: tuple[object, ...]*

A tuple of all interpolated values in the template.

```
>>> cheese = 'Camembert'
>>> template = t'Ah! We do have {cheese}.'
>>> template.values
('Camembert',)
```

The `values` tuple always has the same length as the `interpolations` tuple. It is always equivalent to `tuple(i.value for i in template.interpolations)`.

Methods

__new__(**args: str | Interpolation*)

While literal syntax is the most common way to create a **`Template`**, it is also possible to create them directly using the constructor:

```
>>> from string.templatelib import Interpolation, Template
>>> cheese = 'Camembert'
>>> template = Template(
...     'Ah! We do have ', Interpolation(cheese,
'cheese'), '.'
... )
>>> list(template)
['Ah! We do have ', Interpolation('Camembert', 'cheese',
None, ''), '.']
```

If multiple strings are passed consecutively, they will be concatenated into a single value in the **`strings`** attribute. For example, the following code creates a **`Template`** with a single final string:

```
>>> from string.templatelib import Template
>>> template = Template('Ah! We do have ', 'Camembert',
'.')
>>> template.strings
('Ah! We do have Camembert.',)
```

If multiple interpolations are passed consecutively, they will be treated as separate interpolations and an empty string will be inserted between them. For example, the following code creates a template with empty placeholders in the [`strings`] attribute:

```
>>> from string.templatelib import Interpolation, Template
>>> template = Template(
...     Interpolation('Camembert', 'cheese'),
...     Interpolation('.', 'punctuation'),
... )
>>> template.strings
('', '', '')
```

iter(template)

Iterate over the template, yielding each non-empty string and `Interpolation` in the correct order:

```
>>> cheese = 'Camembert'
>>> list(t'Ah! We do have {cheese}.')
['Ah! We do have ', Interpolation('Camembert', 'cheese',
None, ''), '.']
```

Caution

Empty strings are **not** included in the iteration:

```
>>> response = 'We do have '
>>> cheese = 'Camembert'
>>> list(t'Ah! {response}{cheese}.')
['Ah! ',
 Interpolation('We do have ', 'response', None, ''),
 Interpolation('Camembert', 'cheese', None, ''),
 '.']
```

template + other
template += other

Concatenate this template with another, returning a new **`Template`** instance:

```
>>> cheese = 'Camembert'
>>> list(t'Ah! ' + t'We do have {cheese}.')
['Ah! We do have ', Interpolation('Camembert', 'cheese',
None, ''), '.']
```

Concatenating a **`Template`** and a `str` is **not** supported. This is because it is unclear whether the string should be treated as a static string or an interpolation. If you want to concatenate a **`Template`** with a string, you should either wrap the string directly in a **`Template`** (to treat it as a static string) or use an **`Interpolation`** (to treat it as dynamic):

```
>>> from string.templatelib import Interpolation, Template
>>> template = t'Ah! '
>>> # Treat 'We do have ' as a static string
>>> template += Template('We do have ')
>>> # Treat cheese as an interpolation
>>> cheese = 'Camembert'
>>> template += Template(Interpolation(cheese, 'cheese'))
>>> list(template)
['Ah! We do have ', Interpolation('Camembert', 'cheese',
None, '')]
```

class string.templatelib.Interpolation

The **`Interpolation`** type represents an expression inside a template string. It is immutable, meaning that attributes of an interpolation cannot be reassigned.

Interpolations support pattern matching, allowing you to match against their attributes with the match statement:

```
>>> from string.templatelib import Interpolation
>>> interpolation = t'{1. + 2.:.2f}'.interpolations[0]
>>> interpolation
Interpolation(3.0, '1. + 2.', None, '.2f')
>>> match interpolation:
...     case Interpolation(value, expression, conversion,
format_spec):
...         print(value, expression, conversion, format_spec,
sep=' | ')
...
3.0 | 1. + 2. | None | .2f
```

Attributes

value*: object*

The evaluated value of the interpolation.

```
>>> t'{1 + 2}'.interpolations[0].value
3
```

expression*: str*

For interpolations created by t-string literals, **`expression`** is the expression text found inside the curly brackets (`{` & `}`), including any whitespace, excluding the curly brackets themselves, and ending before the first `!`, `:`, or `=` if any is present. For manually created interpolations, **`expression`** is the arbitrary string provided when constructing the interpolation instance.

We recommend using valid Python expressions or the empty string for the `expression` field of manually created **`Interpolation`** instances, although this is not enforced at runtime.

```
>>> t'{1 + 2}'.interpolations[0].expression
'1 + 2'
```

conversion*: Literal['a', 'r', 's'] | None*

The conversion to apply to the value, or `None`.

The **`conversion`** is the optional conversion to apply to the value:

```
>>> t'{1 + 2!a}'.interpolations[0].conversion
'a'
```

Note

Unlike f-strings, where conversions are applied automatically, the expected behavior with t-strings is that code that *processes* the **`Template`** will decide how to interpret and whether to apply the **`conversion`**. For convenience, the `convert()` function can be used to mimic f-string conversion semantics.

format_spec*: str*

The format specification to apply to the value.

The `format_spec` is an optional, arbitrary string used as the format specification to present the value:

```
>>> t'{1 + 2:.2f}'.interpolations[0].format_spec
'.2f'
```

Note

Unlike f-strings, where format specifications are applied automatically via the `format()` protocol, the expected behavior with t-strings is that code that *processes* the interpolation will decide how to interpret and whether to apply the format specification. As a result, `format_spec` values in interpolations can be arbitrary strings, including those that do not conform to the `format()` protocol.

Methods

__new__(*value: object, expression: str, conversion: Literal['a', 'r', 's'] | None = None, format_spec: str = ''*)

Create a new `Interpolation` object from component parts.

Parameters::

- **value** – The evaluated, in-scope result of the interpolation.
- **expression** – The text of a valid Python expression, or an empty string.
- **conversion** – The conversion to be used, one of `None`, `'a'`, `'r'`, or `'s'`.
- **format_spec** – An optional, arbitrary string used as the format specification to present the value.

Helper functions

string.templatelib.convert(*obj*, */*, *conversion*)

Applies formatted string literal conversion semantics to the given object *obj*. This is frequently useful for custom template string processing logic.

Three conversion flags are currently supported:

- `'s'` which calls **`str()`** on the value (like `!s`),
- `'r'` which calls **`repr()`** (like `!r`), and
- `'a'` which calls **`ascii()`** (like `!a`).

If the conversion flag is `None`, *obj* is returned unchanged.

`re` — Regular expression operations

Source code: Lib/re/ [https://github.com/python/cpython/tree/3.14/Lib/re/]

This module provides regular expression matching operations similar to those found in Perl.

Both patterns and strings to be searched can be Unicode strings (**`str`**) as well as 8-bit strings (**`bytes`**). However, Unicode strings and 8-bit strings cannot be mixed: that is, you cannot match a Unicode string with a bytes pattern or vice-versa; similarly, when asking for a substitution, the replacement string must be of the same type as both the pattern and the search string.

Regular expressions use the backslash character (`'\'`) to indicate special forms or to allow special characters to be used without invoking their special meaning. This collides with Python’s usage of the same character for the same purpose in string literals; for example, to match a literal backslash, one might have to write `'\\\\'` as the pattern string, because the regular expression must be `\\`, and each backslash must be expressed as `\\` inside a regular Python string literal. Also, please note that any invalid escape sequences in Python’s usage of the backslash in string literals now generate a **`SyntaxWarning`** and in the future this will become a **`SyntaxError`**. This behaviour will happen even if it is a valid escape sequence for a regular expression.

The solution is to use Python’s raw string notation for regular expression patterns; backslashes are not handled in any special way in a string literal prefixed with `'r'`. So `r"\n"` is a two-character string containing `'\'` and `'n'`, while `"\n"` is a one-character string containing a newline. Usually patterns will be expressed in Python code using this raw string notation.

It is important to note that most regular expression operations are available as module-level functions and methods on compiled regular expressions. The functions are shortcuts that don’t require you to compile a regex object first, but miss some fine-tuning parameters.

See also

The third-party regex [https://pypi.org/project/regex/] module, which has an API compatible with the standard library **re** module, but offers additional functionality and a more thorough Unicode support.

Regular Expression Syntax

A regular expression (or RE) specifies a set of strings that matches it; the functions in this module let you check if a particular string matches a given regular expression (or if a given regular expression matches a particular string, which comes down to the same thing).

Regular expressions can be concatenated to form new regular expressions; if *A* and *B* are both regular expressions, then *AB* is also a regular expression. In general, if a string *p* matches *A* and another string *q* matches *B*, the string *pq* will match AB. This holds unless *A* or *B* contain low precedence operations; boundary conditions between *A* and *B*; or have numbered group references. Thus, complex expressions can easily be constructed from simpler primitive expressions like the ones described here. For details of the theory and implementation of regular expressions, consult the Friedl book [Frie09], or almost any textbook about compiler construction.

A brief explanation of the format of regular expressions follows. For further information and a gentler presentation, consult the Regular Expression HOWTO.

Regular expressions can contain both special and ordinary characters. Most ordinary characters, like `'A'`, `'a'`, or `'0'`, are the simplest regular expressions; they simply match themselves. You can concatenate ordinary characters, so `last` matches the string `'last'`. (In the rest of this section, we'll write RE's in `this special style`, usually without quotes, and strings to be matched `'in single quotes'`.)

Some characters, like `'|'` or `'('`, are special. Special characters either stand for classes of ordinary characters, or affect how the regular expressions around them are interpreted.

Repetition operators or quantifiers (`*`, `+`, `?`, `{m,n}`, etc) cannot be directly nested. This avoids ambiguity with the non-greedy modifier suffix `?`, and with other modifiers in other implementations. To apply a second repetition to an inner repetition, parentheses may be used. For example, the expression `(?:a{6})*` matches any multiple of six `'a'` characters.

The special characters are:

`.`

(Dot.) In the default mode, this matches any character except a newline. If the **`DOTALL`** flag has been specified, this matches any character including a newline. `(?s:.)` matches any character regardless of flags.

`^`

(Caret.) Matches the start of the string, and in **`MULTILINE`** mode also matches immediately after each newline.

`$`

Matches the end of the string or just before the newline at the end of the string, and in **`MULTILINE`** mode also matches before a newline. `foo` matches both ‘foo’ and ‘foobar’, while the regular expression `foo$` matches only ‘foo’. More interestingly, searching for `foo.$` in `'foo1\nfoo2\n'` matches ‘foo2’ normally, but ‘foo1’ in **`MULTILINE`** mode; searching for a single `$` in `'foo\n'` will find two (empty) matches: one just before the newline, and one at the end of the string.

`*`

Causes the resulting RE to match 0 or more repetitions of the preceding RE, as many repetitions as are possible. `ab*` will match ‘a’, ‘ab’, or ‘a’ followed by any number of ‘b’s.

`+`

Causes the resulting RE to match 1 or more repetitions of the preceding RE. `ab+` will match ‘a’ followed by any non-zero number of ‘b’s; it will not match just ‘a’.

`?`

Causes the resulting RE to match 0 or 1 repetitions of the preceding RE. `ab?` will match either ‘a’ or ‘ab’.

`*?`, `+?`, `??`

The `'*'`, `'+'`, and `'?'` quantifiers are all *greedy*; they match as much text as possible. Sometimes this behaviour isn't desired; if the RE `<.*>` is matched against `'<a> b <c>'`, it will match the entire string, and not just `'<a>'`. Adding `?` after the quantifier makes it perform the match in *non-greedy* or *minimal* fashion; as *few* characters as possible will be matched. Using the RE `<.*?>` will match only `'<a>'`.

`*+`, `++`, `?+`

Like the `'*'`, `'+'`, and `'?'` quantifiers, those where `'+'` is appended also match as many times as possible. However, unlike the true greedy quantifiers, these do not allow back-tracking when the expression following it fails to match. These are known as *possessive* quantifiers. For example, `a*a` will match `'aaaa'` because the `a*` will match all 4 `'a'`s, but, when the final `'a'` is encountered, the expression is backtracked so that in the end the `a*` ends up matching 3 `'a'`s total, and the fourth `'a'` is matched by the final `'a'`. However, when `a*+a` is used to match `'aaaa'`, the `a*+` will match all 4 `'a'`, but when the final `'a'` fails to find any more characters to match, the expression cannot be backtracked and will thus fail to match. `x*+`, `x++` and `x?+` are equivalent to `(?>x*)`, `(?>x+)` and `(?>x?)` correspondingly.

Added in version 3.11.

`{m}`

Specifies that exactly *m* copies of the previous RE should be matched; fewer matches cause the entire RE not to match. For example, `a{6}` will match exactly six `'a'` characters, but not five.

`{m,n}`

Causes the resulting RE to match from *m* to *n* repetitions of the preceding RE, attempting to match as many repetitions as possible. For example, `a{3,5}` will match from 3 to 5 `'a'` characters. Omitting *m* specifies a lower bound of zero, and omitting *n* specifies an infinite upper bound. As an example, `a{4,}b` will match `'aaaab'` or a thousand `'a'` characters followed by a `'b'`, but not `'aaab'`. The comma may not be omitted or the modifier would be confused with the previously described form.

`{m,n}?`

Causes the resulting RE to match from *m* to *n* repetitions of the preceding RE, attempting to match as *few* repetitions as possible. This is the non-greedy version of the previous quantifier. For example, on the 6-character string `'aaaaaa'`, `a{3,5}` will match 5 `'a'` characters, while `a{3,5}?` will only match 3 characters.

`{m,n}+`

Causes the resulting RE to match from *m* to *n* repetitions of the preceding RE, attempting to match as many repetitions as possible *without* establishing any backtracking points. This is the possessive version of the quantifier above. For example, on the 6-character string `'aaaaaa'`, `a{3,5}+aa` attempt to match 5 `'a'` characters, then, requiring 2 more `'a'`s, will need more characters than available and thus fail, while `a{3,5}aa` will match with `a{3,5}` capturing 5, then 4 `'a'`s by backtracking and then the final 2 `'a'`s are matched by the final `aa` in the pattern. `x{m,n}+` is equivalent to `(?>x{m,n})`.

Added in version 3.11.

`\`

Either escapes special characters (permitting you to match characters like `'*'`, `'?'`, and so forth), or signals a special sequence; special sequences are discussed below.

If you’re not using a raw string to express the pattern, remember that Python also uses the backslash as an escape sequence in string literals; if the escape sequence isn’t recognized by Python’s parser, the backslash and subsequent character are included in the resulting string. However, if Python would recognize the resulting sequence, the backslash should be repeated twice. This is complicated and hard to understand, so it’s highly recommended that you use raw strings for all but the simplest expressions.

`[]`

Used to indicate a set of characters. In a set:

- Characters can be listed individually, e.g. `[amk]` will match `'a'`, `'m'`, or `'k'`.
- Ranges of characters can be indicated by giving two characters and separating them by a `'-'`, for example `[a-z]` will match any lowercase ASCII letter, `[0-5][0-9]` will match all the two-digits numbers from `00` to `59`, and `[0-9A-Fa-f]` will match any hexadecimal digit. If `-` is escaped (e.g. `[a\-z]`) or if it’s placed as the first or last character (e.g. `[-a]` or `[a-]`), it will match a literal `'-'`.
- Special characters except backslash lose their special meaning inside sets. For example, `[(+*)]` will match any of the literal characters `'('`, `'+'`, `'*'`, or `')'`.
- Backslash either escapes characters which have special meaning in a set such as `'-'`, `']'`, `'^'` and `'\\'` itself or signals a special sequence which represents a single character such as `\xa0` or `\n` or a character class such as `\w` or `\S` (defined below). Note that `\b`

represents a single “backspace” character, not a word boundary as outside a set, and numeric escapes such as `\1` are always octal escapes, not group references. Special sequences which do not match a single character such as `\A` and `\z` are not allowed.

- Characters that are not within a range can be matched by *complementing* the set. If the first character of the set is `'^'`, all the characters that are *not* in the set will be matched. For example, `[^5]` will match any character except `'5'`, and `[^^]` will match any character except `'^'`. `^` has no special meaning if it’s not the first character in the set.
- To match a literal `']'` inside a set, precede it with a backslash, or place it at the beginning of the set. For example, both `[()[\]{}]` and `[]()[{}]` will match a right bracket, as well as left bracket, braces, and parentheses.
- Support of nested sets and set operations as in Unicode Technical Standard #18 [https://unicode.org/reports/tr18/] might be added in the future. This would change the syntax, so to facilitate this change a **`FutureWarning`** will be raised in ambiguous cases for the time being. That includes sets starting with a literal `'['` or containing literal character sequences `'--'`, `'&&'`, `'~~'`, and `'||'`. To avoid a warning escape them with a backslash.

Changed in version 3.7: **`FutureWarning`** is raised if a character set contains constructs that will change semantically in the future.

|

`A|B`, where *A* and *B* can be arbitrary REs, creates a regular expression that will match either *A* or *B*. An arbitrary number of REs can be separated by the `'|'` in this way. This can be used inside groups (see below) as well. As the target string is scanned, REs separated by `'|'` are tried from left to right. When one pattern completely matches, that branch is accepted. This means that once *A* matches, *B* will not be tested further, even if it would produce a longer overall match. In other words, the `'|'` operator is never greedy. To match a literal `'|'`, use `\|`, or enclose it inside a character class, as in `[|]`.

`(...)`

Matches whatever regular expression is inside the parentheses, and indicates the start and end of a group; the contents of a group can be retrieved after a match has been performed, and can be matched later in the string with the `\number` special sequence, described below. To match the literals `'('` or `')'`, use `\(` or `\)`, or enclose them inside a character class: `[(]`, `[)]`.

`(?...)`

This is an extension notation (a `'?'` following a `'('` is not meaningful otherwise). The first character after the `'?'` determines what the meaning and further syntax of the construct is. Extensions usually do not create a new group; `(?P<name>...)` is the only exception to this rule. Following are the currently supported extensions.

`(?aiLmsux)`

(One or more letters from the set `'a'`, `'i'`, `'L'`, `'m'`, `'s'`, `'u'`, `'x'`.) The group matches the empty string; the letters set the corresponding flags for the entire regular expression:

- `re.A` (ASCII-only matching)
- `re.I` (ignore case)
- `re.L` (locale dependent)

- `re.M` (multi-line)
- `re.S` (dot matches all)
- `re.U` (Unicode matching)
- `re.X` (verbose)

(The flags are described in Module Contents.) This is useful if you wish to include the flags as part of the regular expression, instead of passing a *flag* argument to the `re.compile()` function. Flags should be used first in the expression string.

Changed in version 3.11: This construction can only be used at the start of the expression.

`(?:...)`

A non-capturing version of regular parentheses. Matches whatever regular expression is inside the parentheses, but the substring matched by the group *cannot* be retrieved after performing a match or referenced later in the pattern.

`(?aiLmsux-imsx:...)`

(Zero or more letters from the set `'a'`, `'i'`, `'L'`, `'m'`, `'s'`, `'u'`, `'x'`, optionally followed by `'-'` followed by one or more letters from the `'i'`, `'m'`, `'s'`, `'x'`.) The letters set or remove the corresponding flags for the part of the expression:

- `re.A` (ASCII-only matching)
- `re.I` (ignore case)
- `re.L` (locale dependent)
- `re.M` (multi-line)
- `re.S` (dot matches all)
- `re.U` (Unicode matching)
- `re.X` (verbose)

(The flags are described in Module Contents.)

The letters `'a'`, `'L'` and `'u'` are mutually exclusive when used as inline flags, so they can't be combined or follow `'-'`. Instead, when one of them appears in an inline group, it overrides the matching mode in the enclosing group. In Unicode patterns `(?a:...)` switches to ASCII-only matching, and `(?u:...)` switches to Unicode matching (default). In bytes patterns `(?L:...)` switches to locale dependent matching, and `(?a:...)` switches to ASCII-only matching (default). This override is only in effect for the narrow inline group, and the original matching mode is restored outside of the group.

Added in version 3.6.

Changed in version 3.7: The letters `'a'`, `'L'` and `'u'` also can be used in a group.

`(?>...)`

Attempts to match `...` as if it was a separate regular expression, and if successful, continues to match the rest of the pattern following it. If the subsequent pattern fails to match, the stack can only be unwound to a point *before* the `(?>...)` because once exited, the expression, known as an *atomic group*, has thrown away all stack points within itself. Thus, `(?>.*).` would never match anything because first the `.*` would match all characters possible, then, having nothing left to match, the final `.` would fail to match. Since there are no stack points saved in the Atomic Group, and there is no stack point before it, the entire expression would thus fail to match.

Added in version 3.11.

`(?P<name>...)`

Similar to regular parentheses, but the substring matched by the group is accessible via the symbolic group name *name*. Group names must be valid Python identifiers, and in `bytes` patterns they can only contain bytes in the ASCII range. Each group name must be defined only once within a regular expression. A symbolic group is also a numbered group, just as if the group were not named.

Named groups can be referenced in three contexts. If the pattern is `(?P<quote>['"]).*?(?P=quote)` (i.e. matching a string quoted with either single or double quotes):

Context of reference to group “quote”	**Ways to reference it**
in the same pattern itself	• `(?P=quote)` (as shown) • `\1`
when processing match object *m*	• `m.group('quote')` • `m.end('quote')` (etc.)
in a string passed to the *repl* argument of `re.sub()`	• `\g<quote>` • `\g<1>` • `\1`

Changed in version 3.12: In `bytes` patterns, group *name* can only contain bytes in the ASCII range (`b'\x00'`-`b'\x7f'`).

`(?P=name)`

A backreference to a named group; it matches whatever text was matched by the earlier group named *name*.

`(?#...)`

A comment; the contents of the parentheses are simply ignored.

`(?=...)`

Matches if `...` matches next, but doesn't consume any of the string. This is called a *lookahead assertion*. For example, `Isaac (?=Asimov)` will match `'Isaac '` only if it's followed by `'Asimov'`.

`(?!...)`

Matches if `...` doesn't match next. This is a *negative lookahead assertion*. For example, `Isaac (?!Asimov)` will match `'Isaac '` only if it's *not* followed by `'Asimov'`.

`(?<=...)`

Matches if the current position in the string is preceded by a match for `...` that ends at the current position. This is called a *positive lookbehind assertion*. `(?<=abc)def` will find a match in `'abcdef'`, since the lookbehind will back up 3 characters and check if the contained pattern matches. The contained pattern must only match strings of some fixed length, meaning that `abc` or `a|b` are allowed, but `a*` and `a{3,4}` are not. Note that patterns which start with positive lookbehind assertions will not match at the beginning of the string being searched; you will most likely want to use the **`search()`** function rather than the **`match()`** function:

```
>>> import re
>>> m = re.search('(?<=abc)def', 'abcdef')
>>> m.group(0)
'def'
```

This example looks for a word following a hyphen:

```
>>> m = re.search(r'(?<=-)\w+', 'spam-egg')
>>> m.group(0)
'egg'
```

Changed in version 3.5: Added support for group references of fixed length.

`(?<!...)`

Matches if the current position in the string is not preceded by a match for `...`. This is called a *negative lookbehind assertion*. Similar to positive lookbehind assertions, the contained pattern must only match strings of some fixed length. Patterns which start with negative lookbehind assertions may match at the beginning of the string being searched.

`(?(id/name)yes-pattern|no-pattern)`

Will try to match with `yes-pattern` if the group with given *id* or *name* exists, and with `no-pattern` if it doesn't. `no-pattern` is optional and can be omitted. For example, `(<)?(\w+@\w+(?:\.\w+)+)(?(1)>|$)` is a poor email matching pattern, which will match with `'<user@host.com>'` as well as `'user@host.com'`, but not with `'<user@host.com'` nor `'user@host.com>'`.

Changed in version 3.12: Group *id* can only contain ASCII digits. In **`bytes`** patterns, group *name* can only contain bytes in the ASCII range (`b'\x00'`-`b'\x7f'`).

The special sequences consist of `'\'` and a character from the list below. If the ordinary character is not an ASCII digit or an ASCII letter, then the resulting RE will match the second character. For example, `\$` matches the character `'$'`.

`\number`

Matches the contents of the group of the same number. Groups are numbered starting from 1. For example, `(.+) \1` matches `'the the'` or `'55 55'`, but not `'thethe'` (note the space after the group). This special sequence can only be used to match one of the first 99 groups. If the first digit of *number* is 0, or *number* is 3 octal digits long, it will not be interpreted as a group match, but as the character with octal value *number*. Inside the `'['` and `']'` of a character class, all numeric escapes are treated as characters.

`\A`

Matches only at the start of the string.

`\b`

Matches the empty string, but only at the beginning or end of a word. A word is defined as a sequence of word characters. Note that formally, `\b` is defined as the boundary between a `\w` and a `\W` character (or vice versa), or between `\w` and the beginning or end of the string. This means that `r'\bat\b'` matches `'at'`, `'at.'`, `'(at)'`, and `'as at ay'` but not `'attempt'` or `'atlas'`.

The default word characters in Unicode (str) patterns are Unicode alphanumerics and the underscore, but this can be changed by using the **`ASCII`** flag. Word boundaries are determined by the current locale if the **`LOCALE`** flag is used.

Note

Inside a character range, `\b` represents the backspace character, for compatibility with Python’s string literals.

`\B`

Matches the empty string, but only when it is *not* at the beginning or end of a word. This means that `r'at\B'` matches `'athens'`, `'atom'`, `'attorney'`, but not `'at'`, `'at.'`, or `'at!'`. `\B` is the opposite of `\b`, so word characters in Unicode (str) patterns are Unicode alphanumerics or the underscore, although this can be changed by using the **`ASCII`** flag. Word boundaries are determined by the current locale if the **`LOCALE`** flag is used.

Changed in version 3.14: `\B` now matches empty input string.

`\d`

For Unicode (str) patterns:

Matches any Unicode decimal digit (that is, any character in Unicode character category [Nd] [https://www.unicode.org/versions/Unicode15.0.0/ch04.pdf#G134153]). This includes `[0-9]`, and also many other digit characters.

Matches `[0-9]` if the **`ASCII`** flag is used.

For 8-bit (bytes) patterns:

Matches any decimal digit in the ASCII character set; this is equivalent to `[0-9]`.

`\D`

Matches any character which is not a decimal digit. This is the opposite of `\d`.

Matches `[^0-9]` if the **`ASCII`** flag is used.

`\s`

For Unicode (str) patterns:

Matches Unicode whitespace characters (as defined by **`str.isspace()`**). This includes `[ \t\n\r\f\v]`, and also many other characters, for example the non-breaking spaces mandated by typography rules in many languages.

Matches `[ \t\n\r\f\v]` if the **`ASCII`** flag is used.

For 8-bit (bytes) patterns:

Matches characters considered whitespace in the ASCII character set; this is equivalent to `[ \t\n\r\f\v]`.

`\S`

Matches any character which is not a whitespace character. This is the opposite of `\s`.

Matches `[^ \t\n\r\f\v]` if the **`ASCII`** flag is used.

`\w`

For Unicode (str) patterns:

Matches Unicode word characters; this includes all Unicode alphanumeric characters (as defined by **`str.isalnum()`**), as well as the underscore (_).

Matches `[a-zA-Z0-9_]` if the **`ASCII`** flag is used.

For 8-bit (bytes) patterns:

Matches characters considered alphanumeric in the ASCII character set; this is equivalent to `[a-zA-Z0-9_]`. If the **`LOCALE`** flag is used, matches characters considered alphanumeric in the current locale and the underscore.

`\W`

Matches any character which is not a word character. This is the opposite of `\w`. By default, matches non-underscore (_) characters for which **`str.isalnum()`** returns `False`.

Matches `[^a-zA-Z0-9_]` if the **`ASCII`** flag is used.

If the **`LOCALE`** flag is used, matches characters which are neither alphanumeric in the current locale nor the underscore.

`\z`

Matches only at the end of the string.

Added in version 3.14.

`\Z`

The same as `\z`. For compatibility with old Python versions.

Most of the escape sequences supported by Python string literals are also accepted by the regular expression parser:

```
\a      \b      \f      \n
\N      \r      \t      \u
\U      \v      \x      \\
```

(Note that `\b` is used to represent word boundaries, and means “backspace” only inside character classes.)

`'\u'`, `'\U'`, and `'\N'` escape sequences are only recognized in Unicode (str) patterns. In bytes patterns they are errors. Unknown escapes of ASCII letters are reserved for future use and treated as errors.

Octal escapes are included in a limited form. If the first digit is a 0, or if there are three octal digits, it is considered an octal escape. Otherwise, it is a group reference. As for string literals, octal escapes are always at most three digits in length.

Changed in version 3.3: The `'\u'` and `'\U'` escape sequences have been added.

Changed in version 3.6: Unknown escapes consisting of `'\'` and an ASCII letter now are errors.

Changed in version 3.8: The `'\N{name}'` escape sequence has been added. As in string literals, it expands to the named Unicode character (e.g. `'\N{EM DASH}'`).

Module Contents

The module defines several functions, constants, and an exception. Some of the functions are simplified versions of the full featured methods for compiled regular expressions. Most non-trivial applications always use the compiled form.

Flags

Changed in version 3.6: Flag constants are now instances of **`RegexFlag`**, which is a subclass of **`enum.IntFlag`**.

class re.RegexFlag

An **`enum.IntFlag`** class containing the regex options listed below.

Added in version 3.11: - added to `__all__`

re.A
re.ASCII

Make `\w`, `\W`, `\b`, `\B`, `\d`, `\D`, `\s` and `\S` perform ASCII-only matching instead of full Unicode matching. This is only meaningful for Unicode (str) patterns, and is ignored for bytes patterns.

Corresponds to the inline flag `(?a)`.

Note

The **`U`** flag still exists for backward compatibility, but is redundant in Python 3 since matches are Unicode by default for `str` patterns, and Unicode matching isn’t allowed for bytes patterns. **`UNICODE`** and the inline flag `(?u)` are similarly redundant.

re.DEBUG

Display debug information about compiled expression.

No corresponding inline flag.

re.I
re.IGNORECASE

Perform case-insensitive matching; expressions like `[A-Z]` will also match lowercase letters. Full Unicode matching (such as `Ü` matching `ü`) also works unless the **`ASCII`** flag is used to disable non-ASCII matches. The current locale does not change the effect of this flag unless the **`LOCALE`** flag is also used.

Corresponds to the inline flag `(?i)`.

Note that when the Unicode patterns `[a-z]` or `[A-Z]` are used in combination with the **`IGNORECASE`** flag, they will match the 52 ASCII letters and 4 additional non-ASCII letters: ‘İ’ (U+0130, Latin capital letter I with dot above), ‘ı’ (U+0131, Latin small letter dotless i), ‘ſ’ (U+017F, Latin small letter long s) and ‘K’ (U+212A, Kelvin sign). If the **`ASCII`** flag is used, only letters ‘a’ to ‘z’ and ‘A’ to ‘Z’ are matched.

re.L
re.LOCALE

Make `\w`, `\W`, `\b`, `\B` and case-insensitive matching dependent on the current locale. This flag can be used only with bytes patterns.

Corresponds to the inline flag `(?L)`.

Warning

This flag is discouraged; consider Unicode matching instead. The locale mechanism is very unreliable as it only handles one “culture” at a time and only works with 8-bit locales. Unicode matching is enabled by default for Unicode (str) patterns and it is able to handle different locales and languages.

Changed in version 3.6: **`LOCALE`** can be used only with bytes patterns and is not compatible with **`ASCII`**.

Changed in version 3.7: Compiled regular expression objects with the **`LOCALE`** flag no longer depend on the locale at compile time. Only the locale at matching time affects the result of matching.

re.M
re.MULTILINE

When specified, the pattern character `'^'` matches at the beginning of the string and at the beginning of each line (immediately following each newline); and the pattern character `'$'` matches at the end of the string and at the end of each line (immediately preceding each newline). By default, `'^'` matches only at the beginning of the string, and `'$'` only at the end of the string and immediately before the newline (if any) at the end of the string.

Corresponds to the inline flag `(?m)`.

re.NOFLAG

Indicates no flag being applied, the value is `0`. This flag may be used as a default value for a function keyword argument or as a base value that will be conditionally ORed with other flags. Example of use as a default value:

```
def myfunc(text, flag=re.NOFLAG):
    return re.match(text, flag)
```

Added in version 3.11.

re.S
re.DOTALL

Make the `'.'` special character match any character at all, including a newline; without this flag, `'.'` will match anything *except* a newline.

Corresponds to the inline flag `(?s)`.

re.U
re.UNICODE

In Python 3, Unicode characters are matched by default for `str` patterns. This flag is therefore redundant with **no effect** and is only kept for backward compatibility.

See `ASCII` to restrict matching to ASCII characters instead.

re.X
re.VERBOSE

This flag allows you to write regular expressions that look nicer and are more readable by allowing you to visually separate logical sections of the pattern and add comments. Whitespace within the pattern is ignored, except when in a character class, or when preceded by an unescaped backslash, or within tokens like `*?`, `(?:` or `(?P<...>`. For example, `(? :` and `* ?` are not allowed. When a line contains a `#` that is not in a character class and is not preceded by an unescaped backslash, all characters from the leftmost such `#` through the end of the line are ignored.

This means that the two following regular expression objects that match a decimal number are functionally equal:

```
a = re.compile(r"""\d +  # the integral part
                   \.    # the decimal point
                   \d *  # some fractional digits""", re.X)
b = re.compile(r"\d+\.\d*")
```

Corresponds to the inline flag `(?x)`.

Functions

re.compile(*pattern*, *flags=0*)

Compile a regular expression pattern into a regular expression object, which can be used for matching using its **`match()`**, **`search()`** and other methods, described below.

The expression's behaviour can be modified by specifying a *flags* value. Values can be any of the flags variables, combined using bitwise OR (the `|` operator).

The sequence

```
prog = re.compile(pattern)
result = prog.match(string)
```

is equivalent to

```
result = re.match(pattern, string)
```

but using **`re.compile()`** and saving the resulting regular expression object for reuse is more efficient when the expression will be used several times in a single program.

Note

The compiled versions of the most recent patterns passed to **`re.compile()`** and the module-level matching functions are cached, so programs that use only a few regular expressions at a time needn't worry about compiling regular expressions.

re.search(*pattern*, *string*, *flags=0*)

Scan through *string* looking for the first location where the regular expression *pattern* produces a match, and return a corresponding **`Match`**. Return `None` if no position in the string matches the pattern; note that this is different from finding a zero-length match at some point in the string.

The expression's behaviour can be modified by specifying a *flags* value. Values can be any of the flags variables, combined using bitwise OR (the `|` operator).

re.match(*pattern*, *string*, *flags=0*)

If zero or more characters at the beginning of *string* match the regular expression *pattern*, return a corresponding **`Match`**. Return `None` if the string does not match the pattern; note that this is different from a zero-length match.

Note that even in **`MULTILINE`** mode, **`re.match()`** will only match at the beginning of the string and not at the beginning of each line.

If you want to locate a match anywhere in *string*, use **`search()`** instead (see also search() vs. match()).

The expression's behaviour can be modified by specifying a *flags* value. Values can be any of the flags variables, combined using bitwise OR (the `|` operator).

re.fullmatch(*pattern*, *string*, *flags=0*)

If the whole *string* matches the regular expression *pattern*, return a corresponding `Match`. Return `None` if the string does not match the pattern; note that this is different from a zero-length match.

The expression’s behaviour can be modified by specifying a *flags* value. Values can be any of the flags variables, combined using bitwise OR (the `|` operator).

Added in version 3.4.

re.split(*pattern*, *string*, *maxsplit=0*, *flags=0*)

Split *string* by the occurrences of *pattern*. If capturing parentheses are used in *pattern*, then the text of all groups in the pattern are also returned as part of the resulting list. If *maxsplit* is nonzero, at most *maxsplit* splits occur, and the remainder of the string is returned as the final element of the list.

```
>>> re.split(r'\W+', 'Words, words, words.')
['Words', 'words', 'words', '']
>>> re.split(r'(\W+)', 'Words, words, words.')
['Words', ', ', 'words', ', ', 'words', '.', '']
>>> re.split(r'\W+', 'Words, words, words.', maxsplit=1)
['Words', 'words, words.']
>>> re.split('[a-f]+', '0a3B9', flags=re.IGNORECASE)
['0', '3', '9']
```

If there are capturing groups in the separator and it matches at the start of the string, the result will start with an empty string. The same holds for the end of the string:

```
>>> re.split(r'(\W+)', '...words, words...')
['', '...', 'words', ', ', 'words', '...', '']
```

That way, separator components are always found at the same relative indices within the result list.

Adjacent empty matches are not possible, but an empty match can occur immediately after a non-empty match.

```
>>> re.split(r'\b', 'Words, words, words.')
['', 'Words', ', ', 'words', ', ', 'words', '.']
>>> re.split(r'\W*', '...words...')
['', '', 'w', 'o', 'r', 'd', 's', '', '']
>>> re.split(r'(\W*)', '...words...')
['', '...', '', '', 'w', '', 'o', '', 'r', '', 'd', '', 's',
'...', '', '', '']
```

The expression’s behaviour can be modified by specifying a *flags* value. Values can be any of the flags variables, combined using bitwise OR (the `|` operator).

Changed in version 3.1: Added the optional flags argument.

Changed in version 3.7: Added support of splitting on a pattern that could match an empty string.

Deprecated since version 3.13: Passing *maxsplit* and *flags* as positional arguments is deprecated. In future Python versions they will be keyword-only parameters.

re.findall(*pattern*, *string*, *flags=0*)

Return all non-overlapping matches of *pattern* in *string*, as a list of strings or tuples. The *string* is scanned left-to-right, and matches are returned in the order found. Empty matches are included in the result.

The result depends on the number of capturing groups in the pattern. If there are no groups, return a list of strings matching the whole pattern. If there is exactly one group, return a list of strings matching that group. If multiple groups are present, return a list of tuples of strings matching the groups. Non-capturing groups do not affect the form of the result.

```
>>> re.findall(r'\bf[a-z]*', 'which foot or hand fell
fastest')
['foot', 'fell', 'fastest']
>>> re.findall(r'(\w+)=(\d+)', 'set width=20 and height=10')
[('width', '20'), ('height', '10')]
```

The expression’s behaviour can be modified by specifying a *flags* value. Values can be any of the flags variables, combined using bitwise OR (the `|` operator).

Changed in version 3.7: Non-empty matches can now start just after a previous empty match.

re.finditer(*pattern*, *string*, *flags=0*)

Return an iterator yielding **`Match`** objects over all non-overlapping matches for the RE *pattern* in *string*. The *string* is scanned left-to-right, and matches are returned in the order found. Empty matches are included in the result.

The expression’s behaviour can be modified by specifying a *flags* value. Values can be any of the flags variables, combined using bitwise OR (the `|` operator).

Changed in version 3.7: Non-empty matches can now start just after a previous empty match.

re.sub(*pattern*, *repl*, *string*, *count=0*, *flags=0*)

Return the string obtained by replacing the leftmost non-overlapping occurrences of *pattern* in *string* by the replacement *repl*. If the pattern isn’t found, *string* is returned unchanged. *repl* can be a string or a function; if it is a string, any backslash escapes in it are processed. That is, `\n` is converted to a single newline character, `\r` is converted to a carriage return, and so forth. Unknown escapes of ASCII letters are reserved for future use and treated as errors. Other unknown escapes such as `\&` are left alone. Backreferences, such as `\6`, are replaced with the substring matched by group 6 in the pattern. For example:

```
>>> re.sub(r'def\s+([a-zA-Z_][a-zA-Z_0-9]*)\s*\(\s*\):',
...        r'static PyObject*\npy_\1(void)\n{',
...        'def myfunc():')
'static PyObject*\npy_myfunc(void)\n{'
```

If *repl* is a function, it is called for every non-overlapping occurrence of *pattern*. The function takes a single **`Match`** argument, and returns the replacement string. For example:

```
>>> def dashrepl(matchobj):
...     if matchobj.group(0) == '-': return ' '
...     else: return '-'
...
>>> re.sub('-{1,2}', dashrepl, 'pro----gram-files')
'pro--gram files'
>>> re.sub(r'\sAND\s', ' & ', 'Baked Beans And Spam',
flags=re.IGNORECASE)
'Baked Beans & Spam'
```

The pattern may be a string or a **`Pattern`**.

The optional argument *count* is the maximum number of pattern occurrences to be replaced; *count* must be a non-negative integer. If omitted or zero, all occurrences will be replaced.

Adjacent empty matches are not possible, but an empty match can occur immediately after a non-empty match. As a result, `sub('x*', '-', 'abxd')` returns `'-a-b--d-'` instead of `'-a-b-d-'`.

In string-type *repl* arguments, in addition to the character escapes and backreferences described above, `\g<name>` will use the substring matched by the group named `name`, as defined by the `(?P<name>...)` syntax. `\g<number>` uses the corresponding group number; `\g<2>` is therefore equivalent to `\2`, but isn't ambiguous in a replacement such as `\g<2>0`. `\20` would be interpreted as a reference to group 20, not a reference to group 2 followed by the literal character `'0'`. The backreference `\g<0>` substitutes in the entire substring matched by the RE.

The expression's behaviour can be modified by specifying a *flags* value. Values can be any of the flags variables, combined using bitwise OR (the `|` operator).

Changed in version 3.1: Added the optional flags argument.

Changed in version 3.5: Unmatched groups are replaced with an empty string.

Changed in version 3.6: Unknown escapes in *pattern* consisting of `'\'` and an ASCII letter now are errors.

Changed in version 3.7: Unknown escapes in *repl* consisting of `'\'` and an ASCII letter now are errors. An empty match can occur immediately after a non-empty match.

Changed in version 3.12: Group *id* can only contain ASCII digits. In **`bytes`** replacement strings, group *name* can only contain bytes in the ASCII range (`b'\x00'`-`b'\x7f'`).

Deprecated since version 3.13: Passing *count* and *flags* as positional arguments is deprecated. In future Python versions they will be keyword-only parameters.

re.subn(*pattern*, *repl*, *string*, *count=0*, *flags=0*)

Perform the same operation as **`sub()`**, but return a tuple `(new_string, number_of_subs_made)`.

The expression’s behaviour can be modified by specifying a *flags* value. Values can be any of the flags variables, combined using bitwise OR (the `|` operator).

re.escape(*pattern*)

Escape special characters in *pattern*. This is useful if you want to match an arbitrary literal string that may have regular expression metacharacters in it. For example:

```
>>> print(re.escape('https://www.python.org'))
https://www\.python\.org

>>> legal_chars = string.ascii_lowercase + string.digits +
"!#$%&'*+-.^_`|~:"
>>> print('[%s]+' % re.escape(legal_chars))
[abcdefghijklmnopqrstuvwxyz0123456789!\#\$%\&'\*\+\-
\.\^_`\|\~:]+

>>> operators = ['+', '-', '*', '/', '**']
>>> print('|'.join(map(re.escape, sorted(operators,
reverse=True))))
/|\-|\+|\*\*|\*
```

This function must not be used for the replacement string in **`sub()`** and **`subn()`**, only backslashes should be escaped. For example:

```
>>> digits_re = r'\d+'
>>> sample = '/usr/sbin/sendmail - 0 errors, 12 warnings'
>>> print(re.sub(digits_re, digits_re.replace('\\', r'\\'),
sample))
/usr/sbin/sendmail - \d+ errors, \d+ warnings
```

Changed in version 3.3: The `'_'` character is no longer escaped.

Changed in version 3.7: Only characters that can have special meaning in a regular expression are escaped. As a result, `'!'`, `'"'`, `'%'`, `"'"`, `','`, `'/'`, `':'`, `';'`, `'<'`, `'='`, `'>'`, `'@'`, and ``"`"`` are no longer escaped.

re.purge()

Clear the regular expression cache.

Exceptions

exception re.PatternError(*msg*, *pattern=None*, *pos=None*)

Exception raised when a string passed to one of the functions here is not a valid regular expression (for example, it might contain unmatched parentheses) or when some other error occurs during compilation or matching. It is never an error if a string contains no match for a pattern. The `PatternError` instance has the following additional attributes:

msg

The unformatted error message.

pattern

The regular expression pattern.

pos

The index in *pattern* where compilation failed (may be `None`).

lineno

The line corresponding to *pos* (may be `None`).

colno

The column corresponding to *pos* (may be `None`).

Changed in version 3.5: Added additional attributes.

Changed in version 3.13: `PatternError` was originally named `error`; the latter is kept as an alias for backward compatibility.

Regular Expression Objects

class re.Pattern

Compiled regular expression object returned by **`re.compile()`**.

Changed in version 3.9: **`re.Pattern`** supports `[]` to indicate a Unicode (str) or bytes pattern. See Generic Alias Type.

Pattern.search(*string*[, *pos*[, *endpos*]])

Scan through *string* looking for the first location where this regular expression produces a match, and return a corresponding **`Match`**. Return `None` if no position in the string matches the pattern; note that this is different from finding a zero-length match at some point in the string.

The optional second parameter *pos* gives an index in the string where the search is to start; it defaults to `0`. This is not completely equivalent to slicing the string; the `'^'` pattern character matches at the real beginning of the string and at positions just after a newline, but not necessarily at the index where the search is to start.

The optional parameter *endpos* limits how far the string will be searched; it will be as if the string is *endpos* characters long, so only the characters from *pos* to `endpos - 1` will be searched for a match. If *endpos* is less than *pos*, no match will be found; otherwise, if *rx* is a compiled regular expression object, `rx.search(string, 0, 50)` is equivalent to `rx.search(string[:50], 0)`.

```
>>> pattern = re.compile("d")
>>> pattern.search("dog")     # Match at index 0
<re.Match object; span=(0, 1), match='d'>
```

```
>>> pattern.search("dog", 1)  # No match; search doesn't
include the "d"
```

Pattern.match(*string*[, *pos*[, *endpos*]])

If zero or more characters at the *beginning* of *string* match this regular expression, return a corresponding **`Match`**. Return `None` if the string does not match the pattern; note that this is different from a zero-length match.

The optional *pos* and *endpos* parameters have the same meaning as for the **`search()`** method.

```
>>> pattern = re.compile("o")
>>> pattern.match("dog")      # No match as "o" is not at
the start of "dog".
>>> pattern.match("dog", 1)   # Match as "o" is the 2nd
character of "dog".
<re.Match object; span=(1, 2), match='o'>
```

If you want to locate a match anywhere in *string*, use **`search()`** instead (see also search() vs. match()).

Pattern.fullmatch(*string*[, *pos*[, *endpos*]])

If the whole *string* matches this regular expression, return a corresponding **`Match`**. Return `None` if the string does not match the pattern; note that this is different from a zero-length match.

The optional *pos* and *endpos* parameters have the same meaning as for the **`search()`** method.

```
>>> pattern = re.compile("o[gh]")
>>> pattern.fullmatch("dog")      # No match as "o" is not
at the start of "dog".
>>> pattern.fullmatch("ogre")     # No match as not the full
string matches.
>>> pattern.fullmatch("doggie", 1, 3)   # Matches within
```

```
given limits.
<re.Match object; span=(1, 3), match='og'>
```

Added in version 3.4.

Pattern.split(*string*, *maxsplit=0*)

Identical to the `split()` function, using the compiled pattern.

Pattern.findall(*string*[, *pos*[, *endpos*]])

Similar to the `findall()` function, using the compiled pattern, but also accepts optional *pos* and *endpos* parameters that limit the search region like for `search()`.

Pattern.finditer(*string*[, *pos*[, *endpos*]])

Similar to the `finditer()` function, using the compiled pattern, but also accepts optional *pos* and *endpos* parameters that limit the search region like for `search()`.

Pattern.sub(*repl*, *string*, *count=0*)

Identical to the `sub()` function, using the compiled pattern.

Pattern.subn(*repl*, *string*, *count=0*)

Identical to the `subn()` function, using the compiled pattern.

Pattern.flags

The regex matching flags. This is a combination of the flags given to `compile()`, any `(?...)` inline flags in the pattern, and implicit flags such as `UNICODE` if the pattern is a Unicode string.

Pattern.groups

The number of capturing groups in the pattern.

Pattern.groupindex

A dictionary mapping any symbolic group names defined by `(?P<id>)` to group numbers. The dictionary is empty if no symbolic groups were used in the pattern.

Pattern.pattern

The pattern string from which the pattern object was compiled.

Changed in version 3.7: Added support of **`copy.copy()`** and **`copy.deepcopy()`**. Compiled regular expression objects are considered atomic.

Match Objects

Match objects always have a boolean value of `True`. Since **`match()`** and **`search()`** return `None` when there is no match, you can test whether there was a match with a simple `if` statement:

```
match = re.search(pattern, string)
if match:
    process(match)
```

class re.Match

Match object returned by successful `match`es and `search`es.

Changed in version 3.9: **`re.Match`** supports `[]` to indicate a Unicode (str) or bytes match. See Generic Alias Type.

Match.expand(*template*)

Return the string obtained by doing backslash substitution on the template string *template*, as done by the **`sub()`** method. Escapes such as `\n` are converted to the appropriate characters, and numeric backreferences (`\1`, `\2`) and named backreferences (`\g<1>`, `\g<name>`) are replaced by the contents of the corresponding group. The backreference `\g<0>` will be replaced by the entire match.

Changed in version 3.5: Unmatched groups are replaced with an empty string.

Match.group([*group1*, *...*])

Returns one or more subgroups of the match. If there is a single argument, the result is a single string; if there are multiple arguments, the result is a tuple with one item per argument. Without arguments, *group1* defaults to zero (the whole match is returned). If a *groupN* argument is zero, the corresponding return value is the entire matching string; if it is in the inclusive range [1..99], it is the string matching the corresponding parenthesized group. If a group number is negative or larger than the number of groups defined in the pattern, an **`IndexError`** exception is raised. If a group is contained in a part of the pattern that did not match, the corresponding result is `None`. If a group is contained in a part of the pattern that matched multiple times, the last match is returned.

```
>>> m = re.match(r"(\w+) (\w+)", "Isaac Newton, physicist")
>>> m.group(0)       # The entire match
'Isaac Newton'
>>> m.group(1)       # The first parenthesized subgroup.
'Isaac'
>>> m.group(2)       # The second parenthesized subgroup.
'Newton'
>>> m.group(1, 2)    # Multiple arguments give us a tuple.
('Isaac', 'Newton')
```

If the regular expression uses the `(?P<name>...)` syntax, the *groupN* arguments may also be strings identifying groups by their group name. If a string argument is not used as a group name in the pattern, an **`IndexError`** exception is raised.

A moderately complicated example:

```
>>> m = re.match(r"(?P<first_name>\w+) (?P<last_name>\w+)",
"Malcolm Reynolds")
>>> m.group('first_name')
'Malcolm'
>>> m.group('last_name')
'Reynolds'
```

Named groups can also be referred to by their index:

```
>>> m.group(1)
'Malcolm'
>>> m.group(2)
'Reynolds'
```

If a group matches multiple times, only the last match is accessible:

```
>>> m = re.match(r"(..)+", "a1b2c3")  # Matches 3 times.
>>> m.group(1)                        # Returns only the
last match.
'c3'
```

Match.__getitem__(*g*)

This is identical to `m.group(g)`. This allows easier access to an individual group from a match:

```
>>> m = re.match(r"(\w+) (\w+)", "Isaac Newton, physicist")
>>> m[0]       # The entire match
'Isaac Newton'
>>> m[1]       # The first parenthesized subgroup.
'Isaac'
>>> m[2]       # The second parenthesized subgroup.
'Newton'
```

Named groups are supported as well:

```
>>> m = re.match(r"(?P<first_name>\w+) (?P<last_name>\w+)",
"Isaac Newton")
>>> m['first_name']
```

```
'Isaac'
>>> m['last_name']
'Newton'
```

Added in version 3.6.

Match.groups(*default=None*)

Return a tuple containing all the subgroups of the match, from 1 up to however many groups are in the pattern. The *default* argument is used for groups that did not participate in the match; it defaults to `None`.

For example:

```
>>> m = re.match(r"(\d+)\.(\d+)", "24.1632")
>>> m.groups()
('24', '1632')
```

If we make the decimal place and everything after it optional, not all groups might participate in the match. These groups will default to `None` unless the *default* argument is given:

```
>>> m = re.match(r"(\d+)\.?(\d+)?", "24")
>>> m.groups()      # Second group defaults to None.
('24', None)
>>> m.groups('0')   # Now, the second group defaults to '0'.
('24', '0')
```

Match.groupdict(*default=None*)

Return a dictionary containing all the *named* subgroups of the match, keyed by the subgroup name. The *default* argument is used for groups that did not participate in the match; it defaults to `None`. For example:

```
>>> m = re.match(r"(?P<first_name>\w+) (?P<last_name>\w+)",
"Malcolm Reynolds")
>>> m.groupdict()
{'first_name': 'Malcolm', 'last_name': 'Reynolds'}
```

Match.start([*group*])

Match.end([*group*])

Return the indices of the start and end of the substring matched by *group*; *group* defaults to zero (meaning the whole matched substring). Return `-1` if *group* exists but did not contribute to the match. For a match object *m*, and a group *g* that did contribute to the match, the substring matched by group *g* (equivalent to `m.group(g)`) is

```
m.string[m.start(g):m.end(g)]
```

Note that `m.start(group)` will equal `m.end(group)` if *group* matched a null string. For example, after `m = re.search('b(c?)', 'cba')`, `m.start(0)` is 1, `m.end(0)` is 2, `m.start(1)` and `m.end(1)` are both 2, and `m.start(2)` raises an **`IndexError`** exception.

An example that will remove *remove_this* from email addresses:

```
>>> email = "tony@tiremove_thisger.net"
>>> m = re.search("remove_this", email)
>>> email[:m.start()] + email[m.end():]
'tony@tiger.net'
```

Match.span([*group*])

For a match *m*, return the 2-tuple `(m.start(group), m.end(group))`. Note that if *group* did not contribute to the match, this is `(-1, -1)`. *group* defaults to zero, the entire match.

Match.pos

The value of *pos* which was passed to the **`search()`** or **`match()`** method of a regex object. This is the index into the string at which the RE engine started looking for a match.

Match.endpos

The value of *endpos* which was passed to the `search()` or `match()` method of a regex object. This is the index into the string beyond which the RE engine will not go.

Match.lastindex

The integer index of the last matched capturing group, or `None` if no group was matched at all. For example, the expressions `(a)b`, `((a)(b))`, and `((ab))` will have `lastindex == 1` if applied to the string `'ab'`, while the expression `(a)(b)` will have `lastindex == 2`, if applied to the same string.

Match.lastgroup

The name of the last matched capturing group, or `None` if the group didn’t have a name, or if no group was matched at all.

Match.re

The regular expression object whose `match()` or `search()` method produced this match instance.

Match.string

The string passed to `match()` or `search()`.

Changed in version 3.7: Added support of `copy.copy()` and `copy.deepcopy()`. Match objects are considered atomic.

Regular Expression Examples

Checking for a Pair

In this example, we’ll use the following helper function to display match objects a little more gracefully:

```
def displaymatch(match):
    if match is None:
        return None
    return '<Match: %r, groups=%r>' % (match.group(),
match.groups())
```

Suppose you are writing a poker program where a player’s hand is represented as a 5-character string with each character representing a card, “a” for ace, “k” for king, “q” for queen, “j” for jack, “t” for 10, and “2” through “9” representing the card with that value.

To see if a given string is a valid hand, one could do the following:

```
>>> valid = re.compile(r"^[a2-9tjqk]{5}$")
>>> displaymatch(valid.match("akt5q"))  # Valid.
"<Match: 'akt5q', groups=()>"
>>> displaymatch(valid.match("akt5e"))  # Invalid.
>>> displaymatch(valid.match("akt"))    # Invalid.
>>> displaymatch(valid.match("727ak"))  # Valid.
"<Match: '727ak', groups=()>"
```

That last hand, `"727ak"`, contained a pair, or two of the same valued cards. To match this with a regular expression, one could use backreferences as such:

```
>>> pair = re.compile(r".*(.).*\1")
>>> displaymatch(pair.match("717ak"))     # Pair of 7s.
"<Match: '717', groups=('7',)>"
>>> displaymatch(pair.match("718ak"))     # No pairs.
>>> displaymatch(pair.match("354aa"))     # Pair of aces.
"<Match: '354aa', groups=('a',)>"
```

To find out what card the pair consists of, one could use the `group()` method of the match object in the following manner:

```
>>> pair = re.compile(r".*(.).*\1")
>>> pair.match("717ak").group(1)
'7'

# Error because re.match() returns None, which doesn't have a
group() method:
>>> pair.match("718ak").group(1)
Traceback (most recent call last):
  File "<pyshell#23>", line 1, in <module>
    re.match(r".*(.).*\1", "718ak").group(1)
AttributeError: 'NoneType' object has no attribute 'group'

>>> pair.match("354aa").group(1)
'a'
```

Simulating scanf()

Python does not currently have an equivalent to **`scanf()`**. Regular expressions are generally more powerful, though also more verbose, than **`scanf()`** format strings. The table below offers some more-or-less equivalent mappings between **`scanf()`** format tokens and regular expressions.

`scanf()` Token	**Regular Expression**
`%c`	`.`
`%5c`	`.{5}`
`%d`	`[-+]?\d+`
`%e`, `%E`, `%f`, `%g`	`[-+]?(\d+(\.\d*)?|\.\d+)([eE][-+]?\d+)?`

`scanf()` Token	**Regular Expression**
`%i`	`[-+]?(0[xX][\dA-Fa-f]+|0[0-7]*|\d+)`
`%o`	`[-+]?[0-7]+`
`%s`	`\S+`
`%u`	`\d+`
`%x`, `%X`	`[-+]?(0[xX])?[\dA-Fa-f]+`

To extract the filename and numbers from a string like

```
/usr/sbin/sendmail - 0 errors, 4 warnings
```

you would use a **`scanf()`** format like

```
%s - %d errors, %d warnings
```

The equivalent regular expression would be

```
(\S+) - (\d+) errors, (\d+) warnings
```

search() vs. match()

Python offers different primitive operations based on regular expressions:

- `re.match()` checks for a match only at the beginning of the string

- `re.search()` checks for a match anywhere in the string (this is what Perl does by default)
- `re.fullmatch()` checks for entire string to be a match

For example:

```
>>> re.match("c", "abcdef")    # No match
>>> re.search("c", "abcdef")   # Match
<re.Match object; span=(2, 3), match='c'>
>>> re.fullmatch("p.*n", "python") # Match
<re.Match object; span=(0, 6), match='python'>
>>> re.fullmatch("r.*n", "python") # No match
```

Regular expressions beginning with `'^'` can be used with `search()` to restrict the match at the beginning of the string:

```
>>> re.match("c", "abcdef")    # No match
>>> re.search("^c", "abcdef")  # No match
>>> re.search("^a", "abcdef")  # Match
<re.Match object; span=(0, 1), match='a'>
```

Note however that in `MULTILINE` mode `match()` only matches at the beginning of the string, whereas using `search()` with a regular expression beginning with `'^'` will match at the beginning of each line.

```
>>> re.match("X", "A\nB\nX", re.MULTILINE)  # No match
>>> re.search("^X", "A\nB\nX", re.MULTILINE)  # Match
<re.Match object; span=(4, 5), match='X'>
```

Making a Phonebook

`split()` splits a string into a list delimited by the passed pattern. The method is invaluable for converting textual data into data structures that can be easily read and modified by Python as demonstrated in the following example that creates a phonebook.

First, here is the input. Normally it may come from a file, here we are using triple-quoted string syntax

```
>>> text = """Ross McFluff: 834.345.1254 155 Elm Street
...
... Ronald Heathmore: 892.345.3428 436 Finley Avenue
... Frank Burger: 925.541.7625 662 South Dogwood Way
...
...
... Heather Albrecht: 548.326.4584 919 Park Place"""
```

The entries are separated by one or more newlines. Now we convert the string into a list with each nonempty line having its own entry:

```
>>> entries = re.split("\n+", text)
>>> entries
['Ross McFluff: 834.345.1254 155 Elm Street',
'Ronald Heathmore: 892.345.3428 436 Finley Avenue',
'Frank Burger: 925.541.7625 662 South Dogwood Way',
'Heather Albrecht: 548.326.4584 919 Park Place']
```

Finally, split each entry into a list with first name, last name, telephone number, and address. We use the `maxsplit` parameter of `split()` because the address has spaces, our splitting pattern, in it:

```
>>> [re.split(":? ", entry, maxsplit=3) for entry in entries]
[['Ross', 'McFluff', '834.345.1254', '155 Elm Street'],
['Ronald', 'Heathmore', '892.345.3428', '436 Finley Avenue'],
['Frank', 'Burger', '925.541.7625', '662 South Dogwood Way'],
['Heather', 'Albrecht', '548.326.4584', '919 Park Place']]
```

The `:?` pattern matches the colon after the last name, so that it does not occur in the result list. With a `maxsplit` of `4`, we could separate the house number from the street name:

```
>>> [re.split(":? ", entry, maxsplit=4) for entry in entries]
[['Ross', 'McFluff', '834.345.1254', '155', 'Elm Street'],
['Ronald', 'Heathmore', '892.345.3428', '436', 'Finley
Avenue'],
['Frank', 'Burger', '925.541.7625', '662', 'South Dogwood
Way'],
['Heather', 'Albrecht', '548.326.4584', '919', 'Park Place']]
```

Text Munging

`sub()` replaces every occurrence of a pattern with a string or the result of a function. This example demonstrates using **`sub()`** with a function to “munge” text, or randomize the order of all the characters in each word of a sentence except for the first and last characters:

```
>>> def repl(m):
...     inner_word = list(m.group(2))
...     random.shuffle(inner_word)
...     return m.group(1) + "".join(inner_word) + m.group(3)
...
>>> text = "Professor Abdolmalek, please report your absences
promptly."
>>> re.sub(r"(\w)(\w+)(\w)", repl, text)
'Poefsrosr Aealmlobdk, pslaee reorpt your abnseces plmrptoy.'
>>> re.sub(r"(\w)(\w+)(\w)", repl, text)
'Pofsroser Aodlambelk, plasee reoprt yuor asnebces potlmrpy.'
```

Finding all Adverbs

`findall()` matches *all* occurrences of a pattern, not just the first one as **`search()`** does. For example, if a writer wanted to find all of the adverbs in some text, they might use **`findall()`** in the following manner:

```
>>> text = "He was carefully disguised but captured quickly by
police."
>>> re.findall(r"\w+ly\b", text)
['carefully', 'quickly']
```

Finding all Adverbs and their Positions

If one wants more information about all matches of a pattern than the matched text, `finditer()` is useful as it provides `Match` objects instead of strings. Continuing with the previous example, if a writer wanted to find all of the adverbs *and their positions* in some text, they would use `finditer()` in the following manner:

```
>>> text = "He was carefully disguised but captured quickly by
police."
>>> for m in re.finditer(r"\w+ly\b", text):
...     print('%02d-%02d: %s' % (m.start(), m.end(),
m.group(0)))
07-16: carefully
40-47: quickly
```

Raw String Notation

Raw string notation (`r"text"`) keeps regular expressions sane. Without it, every backslash (`'\'`) in a regular expression would have to be prefixed with another one to escape it. For example, the two following lines of code are functionally identical:

```
>>> re.match(r"\W(.)\1\W", " ff ")
<re.Match object; span=(0, 4), match=' ff '>
>>> re.match("\\W(.)\\1\\W", " ff ")
<re.Match object; span=(0, 4), match=' ff '>
```

When one wants to match a literal backslash, it must be escaped in the regular expression. With raw string notation, this means `r"\\"`. Without raw string notation, one must use `"\\\\"`, making the following lines of code functionally identical:

```
>>> re.match(r"\\", r"\\")
<re.Match object; span=(0, 1), match='\\'>
>>> re.match("\\\\", r"\\")
<re.Match object; span=(0, 1), match='\\'>
```

Writing a Tokenizer

A tokenizer or scanner [https://en.wikipedia.org/wiki/Lexical_analysis] analyzes a string to categorize groups of characters. This is a useful first step in writing a compiler or interpreter.

The text categories are specified with regular expressions. The technique is to combine those into a single master regular expression and to loop over successive matches:

```
from typing import NamedTuple
import re

class Token(NamedTuple):
    type: str
    value: str
    line: int
    column: int

def tokenize(code):
    keywords = {'IF', 'THEN', 'ENDIF', 'FOR', 'NEXT', 'GOSUB',
'RETURN'}
    token_specification = [
        ('NUMBER',   r'\d+(\.\d*)?'),  # Integer or decimal
number
        ('ASSIGN',   r':='),           # Assignment operator
        ('END',      r';'),            # Statement terminator
        ('ID',       r'[A-Za-z]+'),    # Identifiers
        ('OP',       r'[+\-*/]'),      # Arithmetic operators
        ('NEWLINE',  r'\n'),           # Line endings
        ('SKIP',     r'[ \t]+'),       # Skip over spaces and
tabs
        ('MISMATCH', r'.'),            # Any other character
    ]
    tok_regex = '|'.join('(?P<%s>%s)' % pair for pair in
token_specification)
    line_num = 1
    line_start = 0
    for mo in re.finditer(tok_regex, code):
        kind = mo.lastgroup
        value = mo.group()
        column = mo.start() - line_start
        if kind == 'NUMBER':
            value = float(value) if '.' in value else
int(value)
        elif kind == 'ID' and value in keywords:
            kind = value
        elif kind == 'NEWLINE':
            line_start = mo.end()
            line_num += 1
            continue
        elif kind == 'SKIP':
            continue
        elif kind == 'MISMATCH':
            raise RuntimeError(f'{value!r} unexpected on line
{line_num}')
        yield Token(kind, value, line_num, column)

statements = '''
    IF quantity THEN
        total := total + price * quantity;
        tax := price * 0.05;
    ENDIF;
'''

for token in tokenize(statements):
    print(token)
```

The tokenizer produces the following output:

```
Token(type='IF', value='IF', line=2, column=4)
Token(type='ID', value='quantity', line=2, column=7)
Token(type='THEN', value='THEN', line=2, column=16)
Token(type='ID', value='total', line=3, column=8)
Token(type='ASSIGN', value=':=', line=3, column=14)
Token(type='ID', value='total', line=3, column=17)
Token(type='OP', value='+', line=3, column=23)
Token(type='ID', value='price', line=3, column=25)
Token(type='OP', value='*', line=3, column=31)
Token(type='ID', value='quantity', line=3, column=33)
Token(type='END', value=';', line=3, column=41)
Token(type='ID', value='tax', line=4, column=8)
Token(type='ASSIGN', value=':=', line=4, column=12)
Token(type='ID', value='price', line=4, column=15)
Token(type='OP', value='*', line=4, column=21)
Token(type='NUMBER', value=0.05, line=4, column=23)
Token(type='END', value=';', line=4, column=27)
Token(type='ENDIF', value='ENDIF', line=5, column=4)
Token(type='END', value=';', line=5, column=9)
```

[Frie09]

Friedl, Jeffrey. Mastering Regular Expressions. 3rd ed., O’Reilly Media, 2009. The third edition of the book no longer covers Python at all, but the first edition covered writing good regular expression patterns in great detail.

`difflib` — Helpers for computing deltas

Source code: Lib/difflib.py
[https://github.com/python/cpython/tree/3.14/Lib/difflib.py]

This module provides classes and functions for comparing sequences. It can be used for example, for comparing files, and can produce information about file differences in various formats, including HTML and context and unified diffs. For comparing directories and files, see also, the **`filecmp`** module.

class difflib.SequenceMatcher

This is a flexible class for comparing pairs of sequences of any type, so long as the sequence elements are hashable. The basic algorithm predates, and is a little fancier than, an algorithm published in the late 1980’s by Ratcliff and Obershelp under the hyperbolic name “gestalt pattern matching.” The idea is to find the longest contiguous matching subsequence that contains no “junk” elements; these “junk” elements are ones that are uninteresting in some sense, such as blank lines or whitespace. (Handling junk is an extension to the Ratcliff and Obershelp algorithm.) The same idea is then applied recursively to the pieces of the sequences to the left and to the right of the matching subsequence. This does not yield minimal edit sequences, but does tend to yield matches that “look right” to people.

Timing: The basic Ratcliff-Obershelp algorithm is cubic time in the worst case and quadratic time in the expected case. `SequenceMatcher` is quadratic time for the worst case and has expected-case behavior dependent in a complicated way on how many elements the sequences have in common; best case time is linear.

Automatic junk heuristic: `SequenceMatcher` supports a heuristic that automatically treats certain sequence items as junk. The heuristic counts how many times each individual item appears in the sequence. If an item’s duplicates (after the first one) account for more than 1% of the sequence and the sequence is at least 200 items long, this item is marked as “popular” and is treated as junk for the purpose of sequence matching. This heuristic can be turned off by setting the `autojunk` argument to `False` when creating the `SequenceMatcher`.

Changed in version 3.2: Added the *autojunk* parameter.

class difflib.Differ

This is a class for comparing sequences of lines of text, and producing human-readable differences or deltas. Differ uses `SequenceMatcher` both to compare sequences of lines, and to compare sequences of characters within similar (near-matching) lines.

Each line of a `Differ` delta begins with a two-letter code:

Code	Meaning
`'- '`	line unique to sequence 1
`'+ '`	line unique to sequence 2
`'  '`	line common to both sequences

Code	**Meaning**
`'? '`	line not present in either input sequence

Lines beginning with ‘`?`’ attempt to guide the eye to intraline differences, and were not present in either input sequence. These lines can be confusing if the sequences contain whitespace characters, such as spaces, tabs or line breaks.

class difflib.HtmlDiff

This class can be used to create an HTML table (or a complete HTML file containing the table) showing a side by side, line by line comparison of text with inter-line and intra-line change highlights. The table can be generated in either full or contextual difference mode.

The constructor for this class is:

__init__(*tabsize=8*, *wrapcolumn=None*, *linejunk=None*, *charjunk=IS_CHARACTER_JUNK*)

Initializes instance of `HtmlDiff`.

tabsize is an optional keyword argument to specify tab stop spacing and defaults to `8`.

wrapcolumn is an optional keyword to specify column number where lines are broken and wrapped, defaults to `None` where lines are not wrapped.

linejunk and *charjunk* are optional keyword arguments passed into `ndiff()` (used by `HtmlDiff` to generate the side by side HTML differences). See `ndiff()` documentation for argument default values and descriptions.

The following methods are public:

make_file(*fromlines*, *tolines*, *fromdesc*='', *todesc*='', *context=False*, *numlines=5*, *, *charset='utf-8'*)

Compares *fromlines* and *tolines* (lists of strings) and returns a string which is a complete HTML file containing a table showing line by line differences with inter-line and intra-line changes highlighted.

fromdesc and *todesc* are optional keyword arguments to specify from/to file column header strings (both default to an empty string).

context and *numlines* are both optional keyword arguments. Set *context* to `True` when contextual differences are to be shown, else the default is `False` to show the full files. *numlines* defaults to `5`. When *context* is `True` *numlines* controls the number of context lines which surround the difference highlights. When *context* is `False` *numlines* controls the number of lines which are shown before a difference highlight when using the “next” hyperlinks (setting to zero would cause the “next” hyperlinks to place the next difference highlight at the top of the browser without any leading context).

Note

fromdesc and *todesc* are interpreted as unescaped HTML and should be properly escaped while receiving input from untrusted sources.

Changed in version 3.5: charset keyword-only argument was added. The default charset of HTML document changed from `'ISO-8859-1'` to `'utf-8'`.

make_table(*fromlines*, *tolines*, *fromdesc=''*, *todesc=''*, *context=False*, *numlines=5*)

Compares *fromlines* and *tolines* (lists of strings) and returns a string which is a complete HTML table showing line by line differences with inter-line and intra-line changes highlighted.

The arguments for this method are the same as those for the `make_file()` method.

difflib.context_diff(*a*, *b*, *fromfile=''*, *tofile=''*, *fromfiledate=''*, *tofiledate=''*, *n=3*, *lineterm='\n'*)

Compare *a* and *b* (lists of strings); return a delta (a generator generating the delta lines) in context diff format.

Context diffs are a compact way of showing just the lines that have changed plus a few lines of context. The changes are shown in a before/after style. The number of context lines is set by *n* which defaults to three.

By default, the diff control lines (those with `***` or `---`) are created with a trailing newline. This is helpful so that inputs created from `io.IOBase.readlines()` result in diffs that are suitable for use with `io.IOBase.writelines()` since both the inputs and outputs have trailing newlines.

For inputs that do not have trailing newlines, set the *lineterm* argument to `""` so that the output will be uniformly newline free.

The context diff format normally has a header for filenames and modification times. Any or all of these may be specified using strings for *fromfile*, *tofile*, *fromfiledate*, and *tofiledate*. The modification times are normally expressed in the ISO 8601 format. If not specified, the strings default to blanks.

```
>>> import sys
>>> from difflib import *
>>> s1 = ['bacon\n', 'eggs\n', 'ham\n', 'guido\n']
>>> s2 = ['python\n', 'eggy\n', 'hamster\n', 'guido\n']
>>> sys.stdout.writelines(context_diff(s1, s2,
fromfile='before.py',
...                        tofile='after.py'))
*** before.py
--- after.py
***************
*** 1,4 ****
! bacon
! eggs
! ham
  guido
--- 1,4 ----
! python
! eggy
! hamster
  guido
```

See A command-line interface to difflib for a more detailed example.

difflib.get_close_matches(*word*, *possibilities*, *n=3*, *cutoff=0.6*)

Return a list of the best “good enough” matches. *word* is a sequence for which close matches are desired (typically a string), and *possibilities* is a list of sequences against which to match *word* (typically a list of strings).

Optional argument *n* (default `3`) is the maximum number of close matches to return; *n* must be greater than `0`.

Optional argument *cutoff* (default `0.6`) is a float in the range [0, 1]. Possibilities that don’t score at least that similar to *word* are ignored.

The best (no more than *n*) matches among the possibilities are returned in a list, sorted by similarity score, most similar first.

```
>>> get_close_matches('appel', ['ape', 'apple', 'peach',
'puppy'])
['apple', 'ape']
>>> import keyword
>>> get_close_matches('wheel', keyword.kwlist)
['while']
>>> get_close_matches('pineapple', keyword.kwlist)
[]
>>> get_close_matches('accept', keyword.kwlist)
['except']
```

difflib.ndiff(*a*, *b*, *linejunk=None*, *charjunk=IS_CHARACTER_JUNK*)

Compare *a* and *b* (lists of strings); return a **`Differ`**-style delta (a generator generating the delta lines).

Optional keyword parameters *linejunk* and *charjunk* are filtering functions (or `None`):

linejunk: A function that accepts a single string argument, and returns true if the string is junk, or false if not. The default is `None`. There is also a module-level function **`IS_LINE_JUNK()`**, which filters out lines without visible characters, except for at most one hash character (`'#'`) – however the underlying **`SequenceMatcher`** class does a dynamic analysis of which lines are so frequent as to constitute noise, and this usually works better than using this function.

charjunk: A function that accepts a character (a string of length 1), and returns if the character is junk, or false if not. The default is module-level function

`IS_CHARACTER_JUNK()`, which filters out whitespace characters (a blank or tab; it’s a bad idea to include newline in this!).

```
>>> diff =
ndiff('one\ntwo\nthree\n'.splitlines(keepends=True),
...
'ore\ntree\nemu\n'.splitlines(keepends=True))
>>> print(''.join(diff), end="")
- one
?  ^
+ ore
?  ^
- two
- three
?  -
+ tree
+ emu
```

difflib.restore(*sequence*, *which*)

Return one of the two sequences that generated a delta.

Given a *sequence* produced by `Differ.compare()` or `ndiff()`, extract lines originating from file 1 or 2 (parameter *which*), stripping off line prefixes.

Example:

```
>>> diff =
ndiff('one\ntwo\nthree\n'.splitlines(keepends=True),
...
'ore\ntree\nemu\n'.splitlines(keepends=True))
>>> diff = list(diff) # materialize the generated delta into
a list
>>> print(''.join(restore(diff, 1)), end="")
one
two
three
>>> print(''.join(restore(diff, 2)), end="")
ore
```

```
tree
emu
```

difflib.unified_diff(*a*, *b*, *fromfile=''*, *tofile=''*, *fromfiledate=''*, *tofiledate=''*, *n=3*, *lineterm='\n'*)

Compare *a* and *b* (lists of strings); return a delta (a generator generating the delta lines) in unified diff format.

Unified diffs are a compact way of showing just the lines that have changed plus a few lines of context. The changes are shown in an inline style (instead of separate before/after blocks). The number of context lines is set by *n* which defaults to three.

By default, the diff control lines (those with `---`, `+++`, or `@@`) are created with a trailing newline. This is helpful so that inputs created from **`io.IOBase.readlines()`** result in diffs that are suitable for use with **`io.IOBase.writelines()`** since both the inputs and outputs have trailing newlines.

For inputs that do not have trailing newlines, set the *lineterm* argument to `""` so that the output will be uniformly newline free.

The unified diff format normally has a header for filenames and modification times. Any or all of these may be specified using strings for *fromfile*, *tofile*, *fromfiledate*, and *tofiledate*. The modification times are normally expressed in the ISO 8601 format. If not specified, the strings default to blanks.

```
>>> s1 = ['bacon\n', 'eggs\n', 'ham\n', 'guido\n']
>>> s2 = ['python\n', 'eggy\n', 'hamster\n', 'guido\n']
>>> sys.stdout.writelines(unified_diff(s1, s2,
fromfile='before.py', tofile='after.py'))
--- before.py
```

```
+++ after.py
@@ -1,4 +1,4 @@
-bacon
-eggs
-ham
+python
+eggy
+hamster
 guido
```

See A command-line interface to difflib for a more detailed example.

difflib.diff_bytes(*dfunc*, *a*, *b*, *fromfile=b''*, *tofile=b''*, *fromfiledate=b''*, *tofiledate=b''*, *n=3*, *lineterm=b'\n'*)

Compare *a* and *b* (lists of bytes objects) using *dfunc*; yield a sequence of delta lines (also bytes) in the format returned by *dfunc*. *dfunc* must be a callable, typically either **`unified_diff()`** or **`context_diff()`**.

Allows you to compare data with unknown or inconsistent encoding. All inputs except *n* must be bytes objects, not str. Works by losslessly converting all inputs (except *n*) to str, and calling `dfunc(a, b, fromfile, tofile, fromfiledate, tofiledate, n, lineterm)`. The output of *dfunc* is then converted back to bytes, so the delta lines that you receive have the same unknown/inconsistent encodings as *a* and *b*.

Added in version 3.5.

difflib.IS_LINE_JUNK(*line*)

Return `True` for ignorable lines. The line *line* is ignorable if *line* is blank or contains a single `'#'`, otherwise it is not ignorable. Used as a default for parameter *linejunk* in **`ndiff()`** in older versions.

difflib.IS_CHARACTER_JUNK(*ch*)

Return `True` for ignorable characters. The character *ch* is ignorable if *ch* is a space or tab, otherwise it is not ignorable. Used as a default for parameter *charjunk* in `ndiff()`.

See also

Pattern Matching: The Gestalt Approach [https://jacobfilipp.com/DrDobbs/articles/DDJ/1988/8807/8807c/8807c.htm]

Discussion of a similar algorithm by John W. Ratcliff and D. E. Metzener. This was published in Dr. Dobb’s Journal in July, 1988.

SequenceMatcher Objects

The `SequenceMatcher` class has this constructor:

class difflib.SequenceMatcher(*isjunk=None*, *a=''*, *b=''*, *autojunk=True*)

Optional argument *isjunk* must be `None` (the default) or a one-argument function that takes a sequence element and returns true if and only if the element is “junk” and should be ignored. Passing `None` for *isjunk* is equivalent to passing `lambda x: False`; in other words, no elements are ignored. For example, pass:

```
lambda x: x in " \t"
```

if you’re comparing lines as sequences of characters, and don’t want to synch up on blanks or hard tabs.

The optional arguments *a* and *b* are sequences to be compared; both default to empty strings. The elements of both sequences must be hashable.

The optional argument *autojunk* can be used to disable the automatic junk heuristic.

Changed in version 3.2: Added the *autojunk* parameter.

SequenceMatcher objects get three data attributes: *bjunk* is the set of elements of *b* for which *isjunk* is `True`; *bpopular* is the set of non-junk elements considered popular by the heuristic (if it is not disabled); *b2j* is a dict mapping the remaining elements of *b* to a list of positions where they occur. All three are reset whenever *b* is reset with `set_seqs()` or `set_seq2()`.

Added in version 3.2: The *bjunk* and *bpopular* attributes.

`SequenceMatcher` objects have the following methods:

set_seqs(*a*, *b*)

Set the two sequences to be compared.

`SequenceMatcher` computes and caches detailed information about the second sequence, so if you want to compare one sequence against many sequences, use `set_seq2()` to set the commonly used sequence once and call `set_seq1()` repeatedly, once for each of the other sequences.

set_seq1(*a*)

Set the first sequence to be compared. The second sequence to be compared is not changed.

set_seq2(*b*)

Set the second sequence to be compared. The first sequence to be compared is not changed.

find_longest_match(*alo=0*, *ahi=None*, *blo=0*, *bhi=None*)

Find longest matching block in `a[alo:ahi]` and `b[blo:bhi]`.

If *isjunk* was omitted or `None`, **`find_longest_match()`** returns `(i, j, k)` such that `a[i:i+k]` is equal to `b[j:j+k]`, where `alo <= i <= i+k <= ahi` and `blo <= j <= j+k <= bhi`. For all `(i', j', k')` meeting those conditions, the additional conditions `k >= k'`, `i <= i'`, and if `i == i'`, `j <= j'` are also met. In other words, of all maximal matching blocks, return one that starts earliest in *a*, and of all those maximal matching blocks that start earliest in *a*, return the one that starts earliest in *b*.

```
>>> s = SequenceMatcher(None, " abcd", "abcd abcd")
>>> s.find_longest_match(0, 5, 0, 9)
Match(a=0, b=4, size=5)
```

If *isjunk* was provided, first the longest matching block is determined as above, but with the additional restriction that no junk element appears in the block. Then that block is extended as far as possible by matching (only) junk elements on both sides. So the resulting block never matches on junk except as identical junk happens to be adjacent to an interesting match.

Here’s the same example as before, but considering blanks to be junk. That prevents `' abcd'` from matching the `' abcd'` at the tail end of the second sequence directly. Instead only the `'abcd'` can match, and matches the leftmost `'abcd'` in the second sequence:

```
>>> s = SequenceMatcher(lambda x: x==" ", " abcd", "abcd
abcd")
>>> s.find_longest_match(0, 5, 0, 9)
Match(a=1, b=0, size=4)
```

If no blocks match, this returns `(alo, blo, 0)`.

This method returns a named tuple `Match(a, b, size)`.

Changed in version 3.9: Added default arguments.

get_matching_blocks()

Return list of triples describing non-overlapping matching subsequences. Each triple is of the form `(i, j, n)`, and means that `a[i:i+n] == b[j:j+n]`. The triples are monotonically increasing in *i* and *j*.

The last triple is a dummy, and has the value `(len(a), len(b), 0)`. It is the only triple with `n == 0`. If `(i, j, n)` and `(i', j', n')` are adjacent triples in the list, and the second is not the last triple in the list, then `i+n < i'` or `j+n < j'`; in other words, adjacent triples always describe non-adjacent equal blocks.

```
>>> s = SequenceMatcher(None, "abxcd", "abcd")
>>> s.get_matching_blocks()
[Match(a=0, b=0, size=2), Match(a=3, b=2, size=2),
Match(a=5, b=4, size=0)]
```

get_opcodes()

Return list of 5-tuples describing how to turn *a* into *b*. Each tuple is of the form `(tag, i1, i2, j1, j2)`. The first tuple has `i1 == j1 == 0`, and remaining tuples have *i1* equal to the *i2* from the preceding tuple, and, likewise, *j1* equal to the previous *j2*.

The *tag* values are strings, with these meanings:

Value	Meaning
`'replace'`	`a[i1:i2]` should be replaced by `b[j1:j2]`.

Value	**Meaning**
`'delete'`	`a[i1:i2]` should be deleted. Note that `j1 == j2` in this case.
`'insert'`	`b[j1:j2]` should be inserted at `a[i1:i1]`. Note that `i1 == i2` in this case.
`'equal'`	`a[i1:i2] == b[j1:j2]` (the sub-sequences are equal).

For example:

```
>>> a = "qabxcd"
>>> b = "abycdf"
>>> s = SequenceMatcher(None, a, b)
>>> for tag, i1, i2, j1, j2 in s.get_opcodes():
...     print('{:7}   a[{}:{}] --> b[{}:{}] {!r:>8} --> 
{!r}'.format(
...         tag, i1, i2, j1, j2, a[i1:i2], b[j1:j2]))
delete    a[0:1] --> b[0:0]      'q' --> ''
equal     a[1:3] --> b[0:2]     'ab' --> 'ab'
replace   a[3:4] --> b[2:3]      'x' --> 'y'
equal     a[4:6] --> b[3:5]     'cd' --> 'cd'
insert    a[6:6] --> b[5:6]       '' --> 'f'
```

get_grouped_opcodes(*n=3*)

Return a generator of groups with up to *n* lines of context.

Starting with the groups returned by **`get_opcodes()`**, this method splits out smaller change clusters and eliminates intervening ranges which have no changes.

The groups are returned in the same format as **`get_opcodes()`**.

ratio()

Return a measure of the sequences' similarity as a float in the range [0, 1].

Where T is the total number of elements in both sequences, and M is the number of matches, this is 2.0*M / T. Note that this is `1.0` if the sequences are identical, and `0.0` if they have nothing in common.

This is expensive to compute if `get_matching_blocks()` or `get_opcodes()` hasn't already been called, in which case you may want to try `quick_ratio()` or `real_quick_ratio()` first to get an upper bound.

Note

Caution: The result of a `ratio()` call may depend on the order of the arguments. For instance:

```
>>> SequenceMatcher(None, 'tide', 'diet').ratio()
0.25
>>> SequenceMatcher(None, 'diet', 'tide').ratio()
0.5
```

quick_ratio()

Return an upper bound on `ratio()` relatively quickly.

real_quick_ratio()

Return an upper bound on `ratio()` very quickly.

The three methods that return the ratio of matching to total characters can give different results due to differing levels of approximation, although `quick_ratio()` and `real_quick_ratio()` are always at least as large as `ratio()`:

```
>>> s = SequenceMatcher(None, "abcd", "bcde")
>>> s.ratio()
0.75
```

```
>>> s.quick_ratio()
0.75
>>> s.real_quick_ratio()
1.0
```

SequenceMatcher Examples

This example compares two strings, considering blanks to be “junk”:

```
>>> s = SequenceMatcher(lambda x: x == " ",
...                      "private Thread currentThread;",
...                      "private volatile Thread
currentThread;")
```

`ratio()` returns a float in [0, 1], measuring the similarity of the sequences. As a rule of thumb, a `ratio()` value over 0.6 means the sequences are close matches:

```
>>> print(round(s.ratio(), 3))
0.866
```

If you’re only interested in where the sequences match, `get_matching_blocks()` is handy:

```
>>> for block in s.get_matching_blocks():
...     print("a[%d] and b[%d] match for %d elements" % block)
a[0] and b[0] match for 8 elements
a[8] and b[17] match for 21 elements
a[29] and b[38] match for 0 elements
```

Note that the last tuple returned by `get_matching_blocks()` is always a dummy, `(len(a), len(b), 0)`, and this is the only case in which the last tuple element (number of elements matched) is `0`.

If you want to know how to change the first sequence into the second, use `get_opcodes()`:

```
>>> for opcode in s.get_opcodes():
...     print("%6s a[%d:%d] b[%d:%d]" % opcode)
 equal a[0:8] b[0:8]
insert a[8:8] b[8:17]
 equal a[8:29] b[17:38]
```

See also

- The `get_close_matches()` function in this module which shows how simple code building on `SequenceMatcher` can be used to do useful work.
- Simple version control recipe [https://code.activestate.com/recipes/576729-simple-version-control/] for a small application built with `SequenceMatcher`.

Differ Objects

Note that `Differ`-generated deltas make no claim to be **minimal** diffs. To the contrary, minimal diffs are often counter-intuitive, because they synch up anywhere possible, sometimes accidental matches 100 pages apart. Restricting synch points to contiguous matches preserves some notion of locality, at the occasional cost of producing a longer diff.

The `Differ` class has this constructor:

class difflib.Differ(*linejunk=None*, *charjunk=None*)

Optional keyword parameters *linejunk* and *charjunk* are for filter functions (or `None`):

linejunk: A function that accepts a single string argument, and returns true if the string is junk. The default is `None`, meaning that no line is considered junk.

charjunk: A function that accepts a single character argument (a string of length 1), and returns true if the character is junk. The default is `None`, meaning that no character is considered junk.

These junk-filtering functions speed up matching to find differences and do not cause any differing lines or characters to be ignored. Read the description of the `find_longest_match()` method's *isjunk* parameter for an explanation.

`Differ` objects are used (deltas generated) via a single method:

compare(*a*, *b*)

Compare two sequences of lines, and generate the delta (a sequence of lines).

Each sequence must contain individual single-line strings ending with newlines. Such sequences can be obtained from the `readlines()` method of file-like objects. The delta generated also consists of newline-terminated strings, ready to be printed as-is via the `writelines()` method of a file-like object.

Differ Example

This example compares two texts. First we set up the texts, sequences of individual single-line strings ending with newlines (such sequences can also be obtained from the `readlines()` method of file-like objects):

```
>>> text1 = '''  1. Beautiful is better than ugly.
...   2. Explicit is better than implicit.
...   3. Simple is better than complex.
...   4. Complex is better than complicated.
... '''.splitlines(keepends=True)
```

```
>>> len(text1)
4
>>> text1[0][-1]
'\n'
>>> text2 = '''  1. Beautiful is better than ugly.
...   3.   Simple is better than complex.
...   4. Complicated is better than complex.
...   5. Flat is better than nested.
... '''.splitlines(keepends=True)
```

Next we instantiate a Differ object:

```
>>> d = Differ()
```

Note that when instantiating a **`Differ`** object we may pass functions to filter out line and character “junk.” See the **`Differ()`** constructor for details.

Finally, we compare the two:

```
>>> result = list(d.compare(text1, text2))
```

`result` is a list of strings, so let’s pretty-print it:

```
>>> from pprint import pprint
>>> pprint(result)
['    1. Beautiful is better than ugly.\n',
 '-   2. Explicit is better than implicit.\n',
 '-   3. Simple is better than complex.\n',
 '+   3.   Simple is better than complex.\n',
 '?     ++\n',
 '-   4. Complex is better than complicated.\n',
 '?            ^                     ---- ^\n',
 '+   4. Complicated is better than complex.\n',
 '?           ++++ ^                      ^\n',
 '+   5. Flat is better than nested.\n']
```

As a single multi-line string it looks like this:

```
>>> import sys
>>> sys.stdout.writelines(result)
```

```
    1. Beautiful is better than ugly.
-   2. Explicit is better than implicit.
-   3. Simple is better than complex.
+   3.   Simple is better than complex.
?     ++
-   4. Complex is better than complicated.
?            ^                     ---- ^
+   4. Complicated is better than complex.
?           ++++ ^                      ^
+   5. Flat is better than nested.
```

A command-line interface to difflib

This example shows how to use difflib to create a `diff`-like utility.

```
""" Command line interface to difflib.py providing diffs in
four formats:

* ndiff:    lists every line and highlights interline changes.
* context:  highlights clusters of changes in a before/after
format.
* unified:  highlights clusters of changes in an inline format.
* html:     generates side by side comparison with change
highlights.

"""

import sys, os, difflib, argparse
from datetime import datetime, timezone

def file_mtime(path):
    t = datetime.fromtimestamp(os.stat(path).st_mtime,
                               timezone.utc)
    return t.astimezone().isoformat()

def main():

    parser = argparse.ArgumentParser()
    parser.add_argument('-c', action='store_true',
default=False,
                        help='Produce a context format diff
(default)')
    parser.add_argument('-u', action='store_true',
default=False,
                        help='Produce a unified format diff')
    parser.add_argument('-m', action='store_true',
default=False,
                        help='Produce HTML side by side diff '
                             '(can use -c and -l in
conjunction)')
    parser.add_argument('-n', action='store_true',
default=False,
                        help='Produce a ndiff format diff')
    parser.add_argument('-l', '--lines', type=int, default=3,
                        help='Set number of context lines
(default 3)')
    parser.add_argument('fromfile')
    parser.add_argument('tofile')
    options = parser.parse_args()

    n = options.lines
    fromfile = options.fromfile
    tofile = options.tofile

    fromdate = file_mtime(fromfile)
    todate = file_mtime(tofile)
    with open(fromfile) as ff:
        fromlines = ff.readlines()
    with open(tofile) as tf:
        tolines = tf.readlines()

    if options.u:
        diff = difflib.unified_diff(fromlines, tolines,
fromfile, tofile, fromdate, todate, n=n)
    elif options.n:
        diff = difflib.ndiff(fromlines, tolines)
    elif options.m:
        diff =
difflib.HtmlDiff().make_file(fromlines,tolines,fromfile,tofile,
context=options.c,numlines=n)
    else:
        diff = difflib.context_diff(fromlines, tolines,
fromfile, tofile, fromdate, todate, n=n)
```

```
    sys.stdout.writelines(diff)

if __name__ == '__main__':
    main()
```

ndiff example

This example shows how to use `difflib.ndiff()`.

```
"""ndiff [-q] file1 file2
    or
ndiff (-r1 | -r2) < ndiff_output > file1_or_file2

Print a human-friendly file difference report to stdout.  Both inter-
and intra-line differences are noted.  In the second form, recreate file1
(-r1) or file2 (-r2) on stdout, from an ndiff report on stdin.

In the first form, if -q ("quiet") is not specified, the first two lines
of output are

-: file1
+: file2

Each remaining line begins with a two-letter code:

    "- "    line unique to file1
    "+ "    line unique to file2
    "  "    line common to both files
    "? "    line not present in either input file

Lines beginning with "? " attempt to guide the eye to intraline
differences, and were not present in either input file.  These lines can be
confusing if the source files contain tab characters.

The first file can be recovered by retaining only lines that begin with
```

```
"  " or "- ", and deleting those 2-character prefixes; use
ndiff with -r1.

The second file can be recovered similarly, but by retaining
only "  " and
"+ " lines; use ndiff with -r2; or, on Unix, the second file
can be
recovered by piping the output through

    sed -n '/^[+ ] /s/^..//p'
"""

__version__ = 1, 7, 0

import difflib, sys

def fail(msg):
    out = sys.stderr.write
    out(msg + "\n\n")
    out(__doc__)
    return 0

# open a file & return the file object; gripe and return 0 if
it
# couldn't be opened
def fopen(fname):
    try:
        return open(fname)
    except IOError as detail:
        return fail("couldn't open " + fname + ": " +
str(detail))

# open two files & spray the diff to stdout; return false iff a
problem
def fcompare(f1name, f2name):
    f1 = fopen(f1name)
    f2 = fopen(f2name)
    if not f1 or not f2:
        return 0

    a = f1.readlines(); f1.close()
    b = f2.readlines(); f2.close()
    for line in difflib.ndiff(a, b):
            print(line, end=' ')

        return 1

# crack args (sys.argv[1:] is normal) & compare;
# return false iff a problem

def main(args):
    import getopt
    try:
        opts, args = getopt.getopt(args, "qr:")
    except getopt.error as detail:
        return fail(str(detail))
    noisy = 1
    qseen = rseen = 0
    for opt, val in opts:
        if opt == "-q":
            qseen = 1
            noisy = 0
        elif opt == "-r":
            rseen = 1
            whichfile = val
    if qseen and rseen:
        return fail("can't specify both -q and -r")
    if rseen:
        if args:
            return fail("no args allowed with -r option")
        if whichfile in ("1", "2"):
            restore(whichfile)
            return 1
        return fail("-r value must be 1 or 2")
    if len(args) != 2:
        return fail("need 2 filename args")
    f1name, f2name = args
    if noisy:
        print('-:', f1name)
        print('+:', f2name)
    return fcompare(f1name, f2name)

# read ndiff output from stdin, and print file1 (which=='1') or
# file2 (which=='2') to stdout

def restore(which):
    restored = difflib.restore(sys.stdin.readlines(), which)
    sys.stdout.writelines(restored)

if __name__ == '__main__':
    main(sys.argv[1:])
```

`textwrap` — Text wrapping and filling

Source code: Lib/textwrap.py
[https://github.com/python/cpython/tree/3.14/Lib/textwrap.py]

The **`textwrap`** module provides some convenience functions, as well as **`TextWrapper`**, the class that does all the work. If you're just wrapping or filling one or two text strings, the convenience functions should be good enough; otherwise, you should use an instance of **`TextWrapper`** for efficiency.

textwrap.wrap(*text*, *width=70*, ***, *initial_indent=''*, *subsequent_indent=''*, *expand_tabs=True*, *replace_whitespace=True*, *fix_sentence_endings=False*, *break_long_words=True*, *drop_whitespace=True*, *break_on_hyphens=True*, *tabsize=8*, *max_lines=None*, *placeholder=' [...]'*)

> Wraps the single paragraph in *text* (a string) so every line is at most *width* characters long. Returns a list of output lines, without final newlines.
>
> Optional keyword arguments correspond to the instance attributes of **`TextWrapper`**, documented below.
>
> See the **`TextWrapper.wrap()`** method for additional details on how **`wrap()`** behaves.

textwrap.fill(*text*, *width=70*, ***, *initial_indent=''*, *subsequent_indent=''*, *expand_tabs=True*, *replace_whitespace=True*, *fix_sentence_endings=False*, *break_long_words=True*, *drop_whitespace=True*, *break_on_hyphens=True*, *tabsize=8*, *max_lines=None*, *placeholder=' [...]'*)

Wraps the single paragraph in *text*, and returns a single string containing the wrapped paragraph. `fill()` is shorthand for

```
"\n".join(wrap(text, ...))
```

In particular, `fill()` accepts exactly the same keyword arguments as `wrap()`.

textwrap.shorten(*text*, *width*, ***, *fix_sentence_endings=False*, *break_long_words=True*, *break_on_hyphens=True*, *placeholder=' [...]'*)

Collapse and truncate the given *text* to fit in the given *width*.

First the whitespace in *text* is collapsed (all whitespace is replaced by single spaces). If the result fits in the *width*, it is returned. Otherwise, enough words are dropped from the end so that the remaining words plus the *placeholder* fit within *width*:

```
>>> textwrap.shorten("Hello  world!", width=12)
'Hello world!'
>>> textwrap.shorten("Hello  world!", width=11)
'Hello [...]'
>>> textwrap.shorten("Hello world", width=10,
placeholder="...")
'Hello...'
```

Optional keyword arguments correspond to the instance attributes of `TextWrapper`, documented below. Note that the whitespace is collapsed before the text is passed to the `TextWrapper` `fill()` function, so changing the value of `tabsize`, `expand_tabs`, `drop_whitespace`, and `replace_whitespace` will have no effect.

Added in version 3.4.

textwrap.dedent(*text*)

Remove any common leading whitespace from every line in *text*.

This can be used to make triple-quoted strings line up with the left edge of the display, while still presenting them in the source code in indented form.

Note that tabs and spaces are both treated as whitespace, but they are not equal: the lines `"  hello"` and `"\thello"` are considered to have no common leading whitespace.

Lines containing only whitespace are ignored in the input and normalized to a single newline character in the output.

For example:

```
def test():
    # end first line with \ to avoid the empty line!
    s = '''\
    hello
      world
    '''
    print(repr(s))          # prints '    hello\n      
world\n    '
    print(repr(dedent(s)))  # prints 'hello\n  world\n'
```

Changed in version 3.14: The **`dedent()`** function now correctly normalizes blank lines containing only whitespace characters. Previously, the implementation only normalized blank lines containing tabs and spaces.

textwrap.indent(*text*, *prefix*, *predicate=None*)

Add *prefix* to the beginning of selected lines in *text*.

Lines are separated by calling `text.splitlines(True)`.

By default, *prefix* is added to all lines that do not consist solely of whitespace (including any line endings).

For example:

```
>>> s = 'hello\n\n \nworld'
>>> indent(s, '  ')
'  hello\n\n \n  world'
```

The optional *predicate* argument can be used to control which lines are indented. For example, it is easy to add *prefix* to even empty and whitespace-only lines:

```
>>> print(indent(s, '+ ', lambda line: True))
+ hello
+
+
+ world
```

Added in version 3.3.

`wrap()`, `fill()` and `shorten()` work by creating a `TextWrapper` instance and calling a single method on it. That instance is not reused, so for applications that process many text strings using `wrap()` and/or `fill()`, it may be more efficient to create your own `TextWrapper` object.

Text is preferably wrapped on whitespaces and right after the hyphens in hyphenated words; only then will long words be broken if necessary, unless `TextWrapper.break_long_words` is set to false.

class textwrap.TextWrapper(***kwargs*)

The `TextWrapper` constructor accepts a number of optional keyword arguments. Each keyword argument corresponds to an instance attribute, so for example

```
wrapper = TextWrapper(initial_indent="* ")
```

is the same as

```
wrapper = TextWrapper()
wrapper.initial_indent = "* "
```

You can reuse the same `TextWrapper` object many times, and you can change any of its options through direct assignment to instance attributes between uses.

The `TextWrapper` instance attributes (and keyword arguments to the constructor) are as follows:

width

(default: `70`) The maximum length of wrapped lines. As long as there are no individual words in the input text longer than `width`, `TextWrapper` guarantees that no output line will be longer than `width` characters.

expand_tabs

(default: `True`) If true, then all tab characters in *text* will be expanded to spaces using the `expandtabs()` method of *text*.

tabsize

(default: `8`) If `expand_tabs` is true, then all tab characters in *text* will be expanded to zero or more spaces, depending on the current column and the given tab size.

Added in version 3.3.

replace_whitespace

(default: `True`) If true, after tab expansion but before wrapping, the `wrap()` method will replace each whitespace character with a single space. The whitespace characters replaced are as follows: tab, newline, vertical tab, formfeed, and carriage return (`'\t\n\v\f\r'`).

Note

If `expand_tabs` is false and `replace_whitespace` is true, each tab character will be replaced by a single space, which is *not* the same as tab expansion.

Note

If `replace_whitespace` is false, newlines may appear in the middle of a line and cause strange output. For this reason, text should be split into paragraphs (using `str.splitlines()` or similar) which are wrapped separately.

drop_whitespace

(default: `True`) If true, whitespace at the beginning and ending of every line (after wrapping but before indenting) is dropped. Whitespace at the beginning of the paragraph, however, is not dropped if non-whitespace follows it. If whitespace being dropped takes up an entire line, the whole line is dropped.

initial_indent

(default: `''`) String that will be prepended to the first line of wrapped output. Counts towards the length of the first line. The empty string is not indented.

subsequent_indent

(default: `''`) String that will be prepended to all lines of wrapped output except the first. Counts towards the length of each line except the first.

fix_sentence_endings

(default: `False`) If true, **`TextWrapper`** attempts to detect sentence endings and ensure that sentences are always separated by exactly two spaces. This is generally desired for text in a monospaced font. However, the sentence detection algorithm is imperfect: it assumes that a sentence ending consists of a lowercase letter followed by one of `'.'`, `'!'`, or `'?'`, possibly followed by one of `'"'` or `"'"`, followed by a space. One problem with this algorithm is that it is unable to detect the difference between “Dr.” in

```
[...] Dr. Frankenstein's monster [...]
```

and “Spot.” in

```
[...] See Spot. See Spot run [...]
```

`fix_sentence_endings` is false by default.

Since the sentence detection algorithm relies on `string.lowercase` for the definition of “lowercase letter”, and a convention of using two spaces after a period to separate sentences on the same line, it is specific to English-language texts.

break_long_words

(default: `True`) If true, then words longer than **`width`** will be broken in order to ensure that no lines are longer than **`width`**. If it is false, long words will not be broken, and some lines may be longer than **`width`**. (Long words will be put on a line by themselves, in order to minimize the amount by which **`width`** is exceeded.)

break_on_hyphens

(default: `True`) If true, wrapping will occur preferably on whitespaces and right after hyphens in compound words, as it is customary in English. If false, only whitespaces will be considered as potentially good places for line breaks, but you need to set `break_long_words` to false if you want truly insecable words. Default behaviour in previous versions was to always allow breaking hyphenated words.

max_lines

(default: `None`) If not `None`, then the output will contain at most *max_lines* lines, with *placeholder* appearing at the end of the output.

Added in version 3.4.

placeholder

(default: `' [...]'`) String that will appear at the end of the output text if it has been truncated.

Added in version 3.4.

`TextWrapper` also provides some public methods, analogous to the module-level convenience functions:

wrap(*text*)

Wraps the single paragraph in *text* (a string) so every line is at most `width` characters long. All wrapping options are taken from instance attributes of the `TextWrapper` instance. Returns a list of output lines, without final newlines. If the wrapped output has no content, the returned list is empty.

fill(*text*)

Wraps the single paragraph in *text*, and returns a single string containing the wrapped paragraph.

`unicodedata` — Unicode Database

This module provides access to the Unicode Character Database (UCD) which defines character properties for all Unicode characters. The data contained in this database is compiled from the UCD version 16.0.0 [https://www.unicode.org/Public/16.0.0/ucd].

The module uses the same names and symbols as defined by Unicode Standard Annex #44, “Unicode Character Database” [https://www.unicode.org/reports/tr44/]. It defines the following functions:

See also

The Unicode HOWTO for more information about Unicode and how to use this module.

unicodedata.lookup(*name*)

Look up character by name. If a character with the given name is found, return the corresponding character. If not found, **`KeyError`** is raised. For example:

```
>>> unicodedata.lookup('LEFT CURLY BRACKET')
'{'
```

The characters returned by this function are the same as those produced by `\N` escape sequence in string literals. For example:

```
>>> unicodedata.lookup('MIDDLE DOT') == '\N{MIDDLE DOT}'
True
```

Changed in version 3.3: Support for name aliases [[1]] and named sequences [[2]] has been added.

unicodedata.name(*chr*, *default=None*, */*)

Returns the name assigned to the character *chr* as a string. If no name is defined, *default* is returned, or, if not given, `ValueError` is raised. For example:

```
>>> unicodedata.name('½')
'VULGAR FRACTION ONE HALF'
>>> unicodedata.name('\uFFFF', 'fallback')
'fallback'
```

unicodedata.decimal(*chr*, *default=None*, */*)

Returns the decimal value assigned to the character *chr* as integer. If no such value is defined, *default* is returned, or, if not given, `ValueError` is raised. For example:

```
>>> unicodedata.decimal('\N{ARABIC-INDIC DIGIT NINE}')
9
>>> unicodedata.decimal('\N{SUPERSCRIPT NINE}', -1)
-1
```

unicodedata.digit(*chr*, *default=None*, */*)

Returns the digit value assigned to the character *chr* as integer. If no such value is defined, *default* is returned, or, if not given, `ValueError` is raised:

```
>>> unicodedata.digit('\N{SUPERSCRIPT NINE}')
9
```

unicodedata.numeric(*chr*, *default=None*, */*)

Returns the numeric value assigned to the character *chr* as float. If no such value is defined, *default* is returned, or, if not given, `ValueError` is raised:

```
>>> unicodedata.numeric('½')
0.5
```

unicodedata.category(*chr*)

Returns the general category assigned to the character *chr* as string. General category names consist of two letters. See the General Category Values section of the Unicode Character Database documentation [https://www.unicode.org/reports/tr44/tr44-34.html#General_Category_Values] for a list of category codes. For example:

```
>>> unicodedata.category('A')  # 'L'etter, 'u'ppercase
'Lu'
```

unicodedata.bidirectional(*chr*)

Returns the bidirectional class assigned to the character *chr* as string. If no such value is defined, an empty string is returned. See the Bidirectional Class Values section of the Unicode Character Database [https://www.unicode.org/reports/tr44/tr44-34.html#Bidi_Class_Values] documentation for a list of bidirectional codes. For example:

```
>>> unicodedata.bidirectional('\N{ARABIC-INDIC DIGIT
SEVEN}') # 'A'rabic, 'N'umber
'AN'
```

unicodedata.combining(*chr*)

Returns the canonical combining class assigned to the character *chr* as integer. Returns `0` if no combining class is defined. See the Canonical Combining Class Values section of the Unicode Character Database [https://www.unicode.org/reports/tr44/tr44-34.html#Canonical_Combining_Class_Values] for more information.

unicodedata.east_asian_width(*chr*)

Returns the east asian width assigned to the character *chr* as string. For a list of widths and or more information, see the Unicode Standard Annex #11 [https://www.unicode.org/reports/tr11/tr11-43.html].

unicodedata.mirrored(*chr*)

Returns the mirrored property assigned to the character *chr* as integer. Returns `1` if the character has been identified as a “mirrored” character in bidirectional text, `0` otherwise. For example:

```
>>> unicodedata.mirrored('>')
1
```

unicodedata.decomposition(*chr*)

Returns the character decomposition mapping assigned to the character *chr* as string. An empty string is returned in case no such mapping is defined. For example:

```
>>> unicodedata.decomposition('Ã')
'0041 0303'
```

unicodedata.normalize(*form*, *unistr*)

Return the normal form *form* for the Unicode string *unistr*. Valid values for *form* are ‘NFC’, ‘NFKC’, ‘NFD’, and ‘NFKD’.

The Unicode standard defines various normalization forms of a Unicode string, based on the definition of canonical equivalence and compatibility equivalence. In Unicode, several characters can be expressed in various way. For example, the character U+00C7 (LATIN CAPITAL LETTER C WITH CEDILLA) can also be expressed as the sequence U+0043 (LATIN CAPITAL LETTER C) U+0327 (COMBINING CEDILLA).

For each character, there are two normal forms: normal form C and normal form D. Normal form D (NFD) is also known as canonical decomposition, and translates each character into its decomposed form. Normal form C (NFC) first applies a canonical decomposition, then composes pre-combined characters again.

In addition to these two forms, there are two additional normal forms based on compatibility equivalence. In Unicode, certain characters are supported which normally would be unified with other characters. For example, U+2160 (ROMAN NUMERAL ONE) is really the same thing as U+0049 (LATIN CAPITAL LETTER I). However, it is supported in Unicode for compatibility with existing character sets (for example, gb2312).

The normal form KD (NFKD) will apply the compatibility decomposition, that is, replace all compatibility characters with their equivalents. The normal form KC (NFKC) first applies the compatibility decomposition, followed by the canonical composition.

Even if two unicode strings are normalized and look the same to a human reader, if one has combining characters and the other doesn’t, they may not compare equal.

unicodedata.is_normalized(*form*, *unistr*)

Return whether the Unicode string *unistr* is in the normal form *form*. Valid values for *form* are ‘NFC’, ‘NFKC’, ‘NFD’, and ‘NFKD’.

Added in version 3.8.

In addition, the module exposes the following constant:

unicodedata.unidata_version

The version of the Unicode database used in this module.

unicodedata.ucd_3_2_0

This is an object that has the same methods as the entire module, but uses the Unicode database version 3.2 instead, for applications that require this specific version of the Unicode database (such as IDNA).

Footnotes

[1]

https://www.unicode.org/Public/16.0.0/ucd/NameAliases.txt

[2]

https://www.unicode.org/Public/16.0.0/ucd/NamedSequences.txt

`stringprep` — Internet String Preparation

Source code: Lib/stringprep.py [https://github.com/python/cpython/tree/3.14/Lib/stringprep.py]

When identifying things (such as host names) in the internet, it is often necessary to compare such identifications for “equality”. Exactly how this comparison is executed may depend on the application domain, e.g. whether it should be case-insensitive or not. It may be also necessary to restrict the possible identifications, to allow only identifications consisting of “printable” characters.

RFC 3454 [https://datatracker.ietf.org/doc/html/rfc3454.html] defines a procedure for “preparing” Unicode strings in internet protocols. Before passing strings onto the wire, they are processed with the preparation procedure, after which they have a certain normalized form. The RFC defines a set of tables, which can be combined into profiles. Each profile must define which tables it uses, and what other optional parts of the `stringprep` procedure are part of the profile. One example of a `stringprep` profile is `nameprep`, which is used for internationalized domain names.

The module **`stringprep`** only exposes the tables from **RFC 3454** [https://datatracker.ietf.org/doc/html/rfc3454.html]. As these tables would be very large to represent as dictionaries or lists, the module uses the Unicode character database internally. The module source code itself was generated using the `mkstringprep.py` utility.

As a result, these tables are exposed as functions, not as data structures. There are two kinds of tables in the RFC: sets and mappings. For a set, `stringprep` provides the “characteristic function”, i.e. a function that returns `True` if the parameter is part of the set. For mappings, it provides the mapping function: given the key, it returns the associated value. Below is a list of all functions available in the module.

stringprep.in_table_a1(*code*)

Determine whether *code* is in tableA.1 (Unassigned code points in Unicode 3.2).

stringprep.in_table_b1(*code*)

Determine whether *code* is in tableB.1 (Commonly mapped to nothing).

stringprep.map_table_b2(*code*)

Return the mapped value for *code* according to tableB.2 (Mapping for case-folding used with NFKC).

stringprep.map_table_b3(*code*)

Return the mapped value for *code* according to tableB.3 (Mapping for case-folding used with no normalization).

stringprep.in_table_c11(*code*)

Determine whether *code* is in tableC.1.1 (ASCII space characters).

stringprep.in_table_c12(*code*)

Determine whether *code* is in tableC.1.2 (Non-ASCII space characters).

stringprep.in_table_c11_c12(*code*)

Determine whether *code* is in tableC.1 (Space characters, union of C.1.1 and C.1.2).

stringprep.in_table_c21(*code*)

Determine whether *code* is in tableC.2.1 (ASCII control characters).

stringprep.in_table_c22(*code*)

Determine whether *code* is in tableC.2.2 (Non-ASCII control characters).

stringprep.in_table_c21_c22(*code*)

Determine whether *code* is in tableC.2 (Control characters, union of C.2.1 and C.2.2).

stringprep.in_table_c3(*code*)

Determine whether *code* is in tableC.3 (Private use).

stringprep.in_table_c4(*code*)

Determine whether *code* is in tableC.4 (Non-character code points).

stringprep.in_table_c5(*code*)

Determine whether *code* is in tableC.5 (Surrogate codes).

stringprep.in_table_c6(*code*)

Determine whether *code* is in tableC.6 (Inappropriate for plain text).

stringprep.in_table_c7(*code*)

Determine whether *code* is in tableC.7 (Inappropriate for canonical representation).

stringprep.in_table_c8(*code*)

Determine whether *code* is in tableC.8 (Change display properties or are deprecated).

stringprep.in_table_c9(*code*)

Determine whether *code* is in tableC.9 (Tagging characters).

stringprep.in_table_d1(*code*)

Determine whether *code* is in tableD.1 (Characters with bidirectional property “R” or “AL”).

stringprep.in_table_d2(*code*)

Determine whether *code* is in tableD.2 (Characters with bidirectional property “L”).

`readline` — GNU readline interface

The `readline` module defines a number of functions to facilitate completion and reading/writing of history files from the Python interpreter. This module can be used directly, or via the `rlcompleter` module, which supports completion of Python identifiers at the interactive prompt. Settings made using this module affect the behaviour of both the interpreter's interactive prompt and the prompts offered by the built-in `input()` function.

Readline keybindings may be configured via an initialization file, typically `.inputrc` in your home directory. See Readline Init File [https://tiswww.cwru.edu/php/chet/readline/rluserman.html#Readline-Init-File] in the GNU Readline manual for information about the format and allowable constructs of that file, and the capabilities of the Readline library in general.

Availability: not Android, not iOS, not WASI.

This module is not supported on mobile platforms or WebAssembly platforms.

This is an optional module. If it is missing from your copy of CPython, look for documentation from your distributor (that is, whoever provided Python to you). If you are the distributor, see Requirements for optional modules.

Note

The underlying Readline library API may be implemented by the `editline` (`libedit`) library instead of GNU readline. On macOS the `readline` module detects which library is being used at run time.

The configuration file for `editline` is different from that of GNU readline. If you programmatically load configuration strings you can use `backend` to determine which library is being used.

If you use `editline`/`libedit` readline emulation on macOS, the initialization file located in your home directory is named `.editrc`. For example, the following content in `~/.editrc` will turn ON *vi* keybindings and TAB completion:

```
python:bind -v
python:bind ^I rl_complete
```

Also note that different libraries may use different history file formats. When switching the underlying library, existing history files may become unusable.

readline.backend

The name of the underlying Readline library being used, either `"readline"` or `"editline"`.

Added in version 3.13.

Init file

The following functions relate to the init file and user configuration:

readline.parse_and_bind(*string*)

Execute the init line provided in the *string* argument. This calls **`rl_parse_and_bind()`** in the underlying library.

readline.read_init_file([*filename*])

Execute a readline initialization file. The default filename is the last filename used. This calls **`rl_read_init_file()`** in the underlying library. It raises an auditing event `open` with the file name if given, and `"<readline_init_file>"` otherwise, regardless of which file the library resolves.

Changed in version 3.14: The auditing event was added.

Line buffer

The following functions operate on the line buffer:

readline.get_line_buffer()

Return the current contents of the line buffer (**`rl_line_buffer`** in the underlying library).

readline.insert_text(*string*)

Insert text into the line buffer at the cursor position. This calls **`rl_insert_text()`** in the underlying library, but ignores the return value.

readline.redisplay()

Change what's displayed on the screen to reflect the current contents of the line buffer. This calls **`rl_redisplay()`** in the underlying library.

History file

The following functions operate on a history file:

readline.read_history_file([*filename*])

Load a readline history file, and append it to the history list. The default filename is `~/.history`. This calls **`read_history()`** in the underlying library and raises an auditing event `open` with the file name if given and `"~/.history"` otherwise.

Changed in version 3.14: The auditing event was added.

readline.write_history_file([*filename*])

Save the history list to a readline history file, overwriting any existing file. The default filename is `~/.history`. This calls **`write_history()`** in the underlying library and raises an auditing event `open` with the file name if given and `"~/.history"` otherwise.

Changed in version 3.14: The auditing event was added.

readline.append_history_file(*nelements*[, *filename*])

Append the last *nelements* items of history to a file. The default filename is `~/.history`. The file must already exist. This calls **`append_history()`** in the underlying library. This function only exists if Python was compiled for a version of the library that supports it. It raises an auditing event `open` with the file name if given and `"~/.history"` otherwise.

Added in version 3.5.

Changed in version 3.14: The auditing event was added.

readline.get_history_length()
readline.set_history_length(*length*)

Set or return the desired number of lines to save in the history file. The **`write_history_file()`** function uses this value to truncate the history file, by calling **`history_truncate_file()`** in the underlying library. Negative values imply unlimited history file size.

History list

The following functions operate on a global history list:

readline.clear_history()

Clear the current history. This calls **`clear_history()`** in the underlying library. The Python function only exists if Python was compiled for a version of the library that supports it.

readline.get_current_history_length()

Return the number of items currently in the history. (This is different from `get_history_length()`, which returns the maximum number of lines that will be written to a history file.)

readline.get_history_item(*index*)

Return the current contents of history item at *index*. The item index is one-based. This calls **`history_get()`** in the underlying library.

readline.remove_history_item(*pos*)

Remove history item specified by its position from the history. The position is zero-based. This calls **`remove_history()`** in the underlying library.

readline.replace_history_item(*pos*, *line*)

Replace history item specified by its position with *line*. The position is zero-based. This calls **`replace_history_entry()`** in the underlying library.

readline.add_history(*line*)

Append *line* to the history buffer, as if it was the last line typed. This calls **`add_history()`** in the underlying library.

readline.set_auto_history(*enabled*)

Enable or disable automatic calls to **`add_history()`** when reading input via readline. The *enabled* argument should be a Boolean value that when true, enables auto history, and that when false, disables auto history.

Added in version 3.6.

CPython implementation detail: Auto history is enabled by default, and changes to this do not persist across multiple sessions.

Startup hooks

readline.set_startup_hook([*function*])

Set or remove the function invoked by the **`rl_startup_hook`** callback of the underlying library. If *function* is specified, it will be used as the new hook function; if omitted or `None`, any function already installed is removed. The hook is called with no arguments just before readline prints the first prompt.

readline.set_pre_input_hook([*function*])

Set or remove the function invoked by the **`rl_pre_input_hook`** callback of the underlying library. If *function* is specified, it will be used as the new hook function; if omitted or `None`, any function already installed is removed. The hook is called with no arguments after the first prompt has been printed and just before readline starts reading input characters. This function only exists if Python was compiled for a version of the library that supports it.

Completion

The following functions relate to implementing a custom word completion function. This is typically operated by the Tab key, and can suggest and automatically complete a word being typed. By default, Readline is set up to be used by `rlcompleter` to complete Python identifiers for the interactive interpreter. If the `readline` module is to be used with a custom completer, a different set of word delimiters should be set.

readline.set_completer([*function*])

Set or remove the completer function. If *function* is specified, it will be used as the new completer function; if omitted or `None`, any completer function already installed is removed. The completer function is called as `function(text, state)`, for *state* in `0`, `1`, `2`, ..., until it returns a non-string value. It should return the next possible completion starting with *text*.

The installed completer function is invoked by the *entry_func* callback passed to **`rl_completion_matches()`** in the underlying library. The *text* string comes from the first parameter to the **`rl_attempted_completion_function`** callback of the underlying library.

readline.get_completer()

Get the completer function, or `None` if no completer function has been set.

readline.get_completion_type()

Get the type of completion being attempted. This returns the **`rl_completion_type`** variable in the underlying library as an integer.

readline.get_begidx()

readline.get_endidx()

Get the beginning or ending index of the completion scope. These indexes are the *start* and *end* arguments passed to the **`rl_attempted_completion_function`** callback of the underlying library. The values may be different in the same input editing scenario based on the underlying C readline implementation. Ex: libedit is known to behave differently than libreadline.

readline.set_completer_delims(*string*)

readline.get_completer_delims()

Set or get the word delimiters for completion. These determine the start of the word to be considered for completion (the completion scope). These functions access the **`rl_completer_word_break_characters`** variable in the underlying library.

readline.set_completion_display_matches_hook([*function*])

Set or remove the completion display function. If *function* is specified, it will be used as the new completion display function; if omitted or `None`, any completion display function already installed is removed. This sets or clears the **`rl_completion_display_matches_hook`** callback in the underlying library. The completion display function is called as `function(substitution, [matches], longest_match_length)` once each time matches need to be displayed.

Example

The following example demonstrates how to use the **`readline`** module's history reading and writing functions to automatically load and save a history file named `.python_history` from the user's home directory. The code below would normally be executed automatically during interactive sessions from the user's **`PYTHONSTARTUP`** file.

```
import atexit
import os
import readline

histfile = os.path.join(os.path.expanduser("~"),
".python_history")
try:
    readline.read_history_file(histfile)
    # default history len is -1 (infinite), which may grow
unruly
    readline.set_history_length(1000)
except FileNotFoundError:
    pass

atexit.register(readline.write_history_file, histfile)
```

This code is actually automatically run when Python is run in interactive mode (see Readline configuration).

The following example achieves the same goal but supports concurrent interactive sessions, by only appending the new history.

```
import atexit
import os
import readline
histfile = os.path.join(os.path.expanduser("~"),
".python_history")

try:
    readline.read_history_file(histfile)
    h_len = readline.get_current_history_length()
except FileNotFoundError:
    open(histfile, 'wb').close()
    h_len = 0

def save(prev_h_len, histfile):
    new_h_len = readline.get_current_history_length()
    readline.set_history_length(1000)
    readline.append_history_file(new_h_len - prev_h_len,
histfile)
atexit.register(save, h_len, histfile)
```

The following example extends the `code.InteractiveConsole` class to support history save/restore.

```
import atexit
import code
import os
import readline

class HistoryConsole(code.InteractiveConsole):
    def __init__(self, locals=None, filename="<console>",
                 histfile=os.path.expanduser("~/.console-
history")):
        code.InteractiveConsole.__init__(self, locals,
filename)
        self.init_history(histfile)

    def init_history(self, histfile):
        readline.parse_and_bind("tab: complete")
        if hasattr(readline, "read_history_file"):
            try:
                readline.read_history_file(histfile)
            except FileNotFoundError:
                pass
            atexit.register(self.save_history, histfile)

    def save_history(self, histfile):
        readline.set_history_length(1000)
        readline.write_history_file(histfile)
```

Note

The new REPL introduced in version 3.13 doesn’t support readline. However, readline can still be used by setting the

`PYTHON_BASIC_REPL` environment variable.

`rlcompleter` — Completion function for GNU readline

Source code: Lib/rlcompleter.py
[https://github.com/python/cpython/tree/3.14/Lib/rlcompleter.py]

The **`rlcompleter`** module defines a completion function suitable to be passed to **`set_completer()`** in the **`readline`** module.

When this module is imported on a Unix platform with the **`readline`** module available, an instance of the **`Completer`** class is automatically created and its **`complete()`** method is set as the readline completer. The method provides completion of valid Python identifiers and keywords.

Example:

```
>>> import rlcompleter
>>> import readline
>>> readline.parse_and_bind("tab: complete")
>>> readline. <TAB PRESSED>
readline.__doc__          readline.get_line_buffer(
readline.read_init_file(
readline.__file__         readline.insert_text(
readline.set_completer(
readline.__name__         readline.parse_and_bind(
>>> readline.
```

The **`rlcompleter`** module is designed for use with Python's interactive mode. Unless Python is run with the **`-S`** option, the module is automatically imported and configured (see Readline configuration).

On platforms without **`readline`**, the **`Completer`** class defined by this module can still be used for custom purposes.

class rlcompleter.Completer

Completer objects have the following method:

complete(*text*, *state*)

Return the next possible completion for *text*.

When called by the **`readline`** module, this method is called successively with `state == 0, 1, 2, ...` until the method returns `None`.

If called for *text* that doesn’t include a period character (`'.'`), it will complete from names currently defined in **`__main__`**, **`builtins`** and keywords (as defined by the **`keyword`** module).

If called for a dotted name, it will try to evaluate anything without obvious side-effects (functions will not be evaluated, but it can generate calls to **`__getattr__()`**) up to the last part, and find matches for the rest via the **`dir()`** function. Any exception raised during the evaluation of the expression is caught, silenced and **`None`** is returned.

Binary Data Services

The modules described in this chapter provide some basic services operations for manipulation of binary data. Other operations on binary data, specifically in relation to file formats and network protocols, are described in the relevant sections.

Some libraries described under Text Processing Services also work with either ASCII-compatible binary formats (for example, **`re`**) or all binary data (for example, **`difflib`**).

In addition, see the documentation for Python's built-in binary data types in Binary Sequence Types — bytes, bytearray, memoryview.

- **`struct`** — Interpret bytes as packed binary data
    - Functions and Exceptions
    - Format Strings
        - Byte Order, Size, and Alignment
        - Format Characters
        - Examples
    - Applications
        - Native Formats
        - Standard Formats
    - Classes
- **`codecs`** — Codec registry and base classes
    - Codec Base Classes
        - Error Handlers
        - Stateless Encoding and Decoding
        - Incremental Encoding and Decoding
            - IncrementalEncoder Objects
            - IncrementalDecoder Objects
        - Stream Encoding and Decoding

        - StreamWriter Objects
        - StreamReader Objects
        - StreamReaderWriter Objects
        - StreamRecoder Objects
    - Encodings and Unicode
    - Standard Encodings
    - Python Specific Encodings
        - Text Encodings
        - Binary Transforms
        - Standalone Codec Functions
        - Text Transforms
    - `encodings` — Encodings package
    - `encodings.idna` — Internationalized Domain Names in Applications
    - `encodings.mbcs` — Windows ANSI codepage
    - `encodings.utf_8_sig` — UTF-8 codec with BOM signature

`struct` — Interpret bytes as packed binary data

Source code: Lib/struct.py
[https://github.com/python/cpython/tree/3.14/Lib/struct.py]

This module converts between Python values and C structs represented as Python **`bytes`** objects. Compact format strings describe the intended conversions to/from Python values. The module's functions and objects can be used for two largely distinct applications, data exchange with external sources (files or network connections), or data transfer between the Python application and the C layer.

Note

When no prefix character is given, native mode is the default. It packs or unpacks data based on the platform and compiler on which the Python interpreter was built. The result of packing a given C struct includes pad bytes which maintain proper alignment for the C types involved; similarly, alignment is taken into account when unpacking. In contrast, when communicating data between external sources, the programmer is responsible for defining byte ordering and padding between elements. See Byte Order, Size, and Alignment for details.

Several **`struct`** functions (and methods of **`Struct`**) take a *buffer* argument. This refers to objects that implement the Buffer Protocol and provide either a readable or read-writable buffer. The most common types used for that purpose are `bytes` and `bytearray`, but many other types that can be viewed as an array of bytes implement the buffer protocol, so that they can be read/filled without additional copying from a `bytes` object.

Functions and Exceptions

The module defines the following exception and functions:

exception struct.error

Exception raised on various occasions; argument is a string describing what is wrong.

struct.pack(*format*, *v1*, *v2*, *...*)

Return a bytes object containing the values *v1*, *v2*, … packed according to the format string *format*. The arguments must match the values required by the format exactly.

struct.pack_into(*format*, *buffer*, *offset*, *v1*, *v2*, *...*)

Pack the values *v1*, *v2*, … according to the format string *format* and write the packed bytes into the writable buffer *buffer* starting at position *offset*. Note that *offset* is a required argument.

struct.unpack(*format*, *buffer*)

Unpack from the buffer *buffer* (presumably packed by `pack(format, ...)`) according to the format string *format*. The result is a tuple even if it contains exactly one item. The buffer's size in bytes must match the size required by the format, as reflected by `calcsize()`.

struct.unpack_from(*format*, */*, *buffer*, *offset=0*)

Unpack from *buffer* starting at position *offset*, according to the format string *format*. The result is a tuple even if it contains exactly one item. The buffer's size in bytes, starting at position *offset*, must be at least the size required by the format, as reflected by `calcsize()`.

struct.iter_unpack(*format*, *buffer*)

Iteratively unpack from the buffer *buffer* according to the format string *format*. This function returns an iterator which will read equally sized chunks from the buffer until all its contents have been consumed. The buffer's size in bytes must be a multiple of the size required by the format, as reflected by `calcsize()`.

Each iteration yields a tuple as specified by the format string.

Added in version 3.4.

struct.calcsize(*format*)

Return the size of the struct (and hence of the bytes object produced by `pack(format, ...)`) corresponding to the format string *format*.

Format Strings

Format strings describe the data layout when packing and unpacking data. They are built up from format characters, which specify the type of data being packed/unpacked. In addition, special characters control the byte order, size and alignment. Each format string consists of an optional prefix character which describes the overall properties of the data and one or more format characters which describe the actual data values and padding.

Byte Order, Size, and Alignment

By default, C types are represented in the machine’s native format and byte order, and properly aligned by skipping pad bytes if necessary (according to the rules used by the C compiler). This behavior is chosen so that the bytes of a packed struct correspond exactly to the memory layout of the corresponding C struct. Whether to use native byte ordering and padding or standard formats depends on the application.

Alternatively, the first character of the format string can be used to indicate the byte order, size and alignment of the packed data, according to the following table:

Character	Byte order	Size	Alignment
`@`	native	native	native
`=`	native	standard	none
`<`	little-endian	standard	none
`>`	big-endian	standard	none
`!`	network (= big-endian)	standard	none

If the first character is not one of these, `'@'` is assumed.

Note

The number 1023 (`0x3ff` in hexadecimal) has the following byte representations:

- `03 ff` in big-endian (`>`)
- `ff 03` in little-endian (`<`)

Python example:

```
>>> import struct
>>> struct.pack('>h', 1023)
b'\x03\xff'
>>> struct.pack('<h', 1023)
b'\xff\x03'
```

Native byte order is big-endian or little-endian, depending on the host system. For example, Intel x86, AMD64 (x86-64), and Apple M1 are little-endian; IBM z and many legacy architectures are big-endian. Use **`sys.byteorder`** to check the endianness of your system.

Native size and alignment are determined using the C compiler’s `sizeof` expression. This is always combined with native byte order.

Standard size depends only on the format character; see the table in the Format Characters section.

Note the difference between `'@'` and `'='`: both use native byte order, but the size and alignment of the latter is standardized.

The form `'!'` represents the network byte order which is always big-endian as defined in IETF RFC 1700 [https://datatracker.ietf.org/doc/html/rfc1700].

There is no way to indicate non-native byte order (force byte-swapping); use the appropriate choice of `'<'` or `'>'`.

Notes:

1. Padding is only automatically added between successive structure members. No padding is added at the beginning or the end of the encoded struct.
2. No padding is added when using non-native size and alignment, e.g. with ‘<’, ‘>’, ‘=’, and ‘!’.
3. To align the end of a structure to the alignment requirement of a particular type, end the format with the code for that type with a repeat count of zero. See Examples.

Format Characters

Format characters have the following meaning; the conversion between C and Python values should be obvious given their types. The ‘Standard size’ column refers to the size of the packed value in bytes when using standard size; that is, when the format string starts with one of `'<'`, `'>'`, `'!'` or `'='`. When using native size, the size of the packed value is platform-dependent.

Format	C Type	Python type	Standard size	Notes
`x`	pad byte	no value		(7)
`c`	char	bytes of length 1	1	
`b`	signed char	integer	1	(1), (2)

Format	C Type	Python type	Standard size	Notes
`B`	unsigned char	integer	1	(2)
`?`	_Bool	bool	1	(1)
`h`	short	integer	2	(2)
`H`	unsigned short	integer	2	(2)
`i`	int	integer	4	(2)
`I`	unsigned int	integer	4	(2)
`l`	long	integer	4	(2)
`L`	unsigned long	integer	4	(2)
`q`	long long	integer	8	(2)
`Q`	unsigned long long	integer	8	(2)

Format	**C Type**	**Python type**	**Standard size**	**Notes**
`n`	**`ssize_t`**	integer		(3)
`N`	**`size_t`**	integer		(3)
`e`	(6)	float	2	(4)
`f`	float	float	4	(4)
`d`	double	float	8	(4)
`F`	float complex	complex	8	(10)
`D`	double complex	complex	16	(10)
`s`	char[]	bytes		(9)
`p`	char[]	bytes		(8)
`P`	void*	integer		(5)

Changed in version 3.3: Added support for the `'n'` and `'N'` formats.

Changed in version 3.6: Added support for the `'e'` format.

Changed in version 3.14: Added support for the `'F'` and `'D'` formats.

Notes:

1. The `'?'` conversion code corresponds to the _Bool type defined by C standards since C99. In standard mode, it is represented by one byte.

2. When attempting to pack a non-integer using any of the integer conversion codes, if the non-integer has a `__index__()` method then that method is called to convert the argument to an integer before packing.

   Changed in version 3.2: Added use of the `__index__()` method for non-integers.

3. The `'n'` and `'N'` conversion codes are only available for the native size (selected as the default or with the `'@'` byte order character). For the standard size, you can use whichever of the other integer formats fits your application.

4. For the `'f'`, `'d'` and `'e'` conversion codes, the packed representation uses the IEEE 754 binary32, binary64 or binary16 format (for `'f'`, `'d'` or `'e'` respectively), regardless of the floating-point format used by the platform.

5. The `'P'` format character is only available for the native byte ordering (selected as the default or with the `'@'` byte order character). The byte order character `'='` chooses to use little- or big-endian ordering based on the host system. The struct module does not interpret

this as native ordering, so the `'P'` format is not available.

6. The IEEE 754 binary16 “half precision” type was introduced in the 2008 revision of the IEEE 754 standard [https://en.wikipedia.org/wiki/IEEE_754-2008_revision]. It has a sign bit, a 5-bit exponent and 11-bit precision (with 10 bits explicitly stored), and can represent numbers between approximately `6.1e-05` and `6.5e+04` at full precision. This type is not widely supported by C compilers: on a typical machine, an unsigned short can be used for storage, but not for math operations. See the Wikipedia page on the half-precision floating-point format [https://en.wikipedia.org/wiki/Half-precision_floating-point_format] for more information.

7. When packing, `'x'` inserts one NUL byte.

8. The `'p'` format character encodes a “Pascal string”, meaning a short variable-length string stored in a *fixed number of bytes*, given by the count. The first byte stored is the length of the string, or 255, whichever is smaller. The bytes of the string follow. If the string passed in to **`pack()`** is too long (longer than the count minus 1), only the leading `count-1` bytes of the string are stored. If the string is shorter than `count-1`, it is padded with null bytes so that exactly count bytes in all are used. Note that for **`unpack()`**, the `'p'` format character consumes `count` bytes, but that the string returned can never contain more than 255 bytes.

9. For the `'s'` format character, the count is interpreted as the length of the bytes, not a repeat count like for the other format characters; for example, `'10s'` means a single 10-byte string mapping to or from a single Python byte string, while `'10c'` means 10 separate one byte

character elements (e.g., `cccccccccc`) mapping to or from ten different Python byte objects. (See Examples for a concrete demonstration of the difference.) If a count is not given, it defaults to 1. For packing, the string is truncated or padded with null bytes as appropriate to make it fit. For unpacking, the resulting bytes object always has exactly the specified number of bytes. As a special case, `'0s'` means a single, empty string (while `'0c'` means 0 characters).

10. For the `'F'` and `'D'` format characters, the packed representation uses the IEEE 754 binary32 and binary64 format for components of the complex number, regardless of the floating-point format used by the platform. Note that complex types (`F` and `D`) are available unconditionally, despite complex types being an optional feature in C. As specified in the C11 standard, each complex type is represented by a two-element C array containing, respectively, the real and imaginary parts.

A format character may be preceded by an integral repeat count. For example, the format string `'4h'` means exactly the same as `'hhhh'`.

Whitespace characters between formats are ignored; a count and its format must not contain whitespace though.

When packing a value `x` using one of the integer formats (`'b'`, `'B'`, `'h'`, `'H'`, `'i'`, `'I'`, `'l'`, `'L'`, `'q'`, `'Q'`), if `x` is outside the valid range for that format then **`struct.error`** is raised.

Changed in version 3.1: Previously, some of the integer formats wrapped out-of-range values and raised **`DeprecationWarning`** instead of **`struct.error`**.

For the `'?'` format character, the return value is either `True` or `False`. When packing, the truth value of the argument object is used. Either 0 or 1 in the native or standard bool representation will be packed, and any non-zero value will be `True` when unpacking.

Examples

Note

Native byte order examples (designated by the `'@'` format prefix or lack of any prefix character) may not match what the reader's machine produces as that depends on the platform and compiler.

Pack and unpack integers of three different sizes, using big endian ordering:

```
>>> from struct import *
>>> pack(">bhl", 1, 2, 3)
b'\x01\x00\x02\x00\x00\x00\x03'
>>> unpack('>bhl', b'\x01\x00\x02\x00\x00\x00\x03')
(1, 2, 3)
>>> calcsize('>bhl')
7
```

Attempt to pack an integer which is too large for the defined field:

```
>>> pack(">h", 99999)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
struct.error: 'h' format requires -32768 <= number <= 32767
```

Demonstrate the difference between `'s'` and `'c'` format characters:

```
>>> pack("@ccc", b'1', b'2', b'3')
b'123'
>>> pack("@3s", b'123')
b'123'
```

Unpacked fields can be named by assigning them to variables or by wrapping the result in a named tuple:

```
>>> record = b'raymond   \x32\x12\x08\x01\x08'
>>> name, serialnum, school, gradelevel = unpack('<10sHHb',
record)

>>> from collections import namedtuple
>>> Student = namedtuple('Student', 'name serialnum school
gradelevel')
>>> Student._make(unpack('<10sHHb', record))
Student(name=b'raymond   ', serialnum=4658, school=264,
gradelevel=8)
```

The ordering of format characters may have an impact on size in native mode since padding is implicit. In standard mode, the user is responsible for inserting any desired padding. Note in the first `pack` call below that three NUL bytes were added after the packed `'#'` to align the following integer on a four-byte boundary. In this example, the output was produced on a little endian machine:

```
>>> pack('@ci', b'#', 0x12131415)
b'#\x00\x00\x00\x15\x14\x13\x12'
>>> pack('@ic', 0x12131415, b'#')
b'\x15\x14\x13\x12#'
>>> calcsize('@ci')
8
>>> calcsize('@ic')
5
```

The following format `'llh0l'` results in two pad bytes being added at the end, assuming the platform’s longs are aligned on 4-byte boundaries:

```
>>> pack('@llh0l', 1, 2, 3)
b'\x00\x00\x00\x01\x00\x00\x00\x02\x00\x03\x00\x00'
```

See also

Module `array`
Packed binary storage of homogeneous data.

Module `json`
JSON encoder and decoder.

Module `pickle`
Python object serialization.

Applications

Two main applications for the `struct` module exist, data interchange between Python and C code within an application or another application compiled using the same compiler (native formats), and data interchange between applications using agreed upon data layout (standard formats). Generally speaking, the format strings constructed for these two domains are distinct.

Native Formats

When constructing format strings which mimic native layouts, the compiler and machine architecture determine byte ordering and padding. In such cases, the `@` format character should be used to specify native byte ordering and data sizes. Internal pad bytes are normally inserted automatically. It is possible that a zero-repeat format code will be needed at the end of a format string to round up to the correct byte boundary for proper alignment of consecutive chunks of data.

Consider these two simple examples (on a 64-bit, little-endian machine):

```
>>> calcsize('@lhl')
24
>>> calcsize('@llh')
18
```

Data is not padded to an 8-byte boundary at the end of the second format string without the use of extra padding. A zero-repeat format code solves that problem:

```
>>> calcsize('@llh0l')
24
```

The `'x'` format code can be used to specify the repeat, but for native formats it is better to use a zero-repeat format like `'0l'`.

By default, native byte ordering and alignment is used, but it is better to be explicit and use the `'@'` prefix character.

Standard Formats

When exchanging data beyond your process such as networking or storage, be precise. Specify the exact byte order, size, and alignment. Do not assume they match the native order of a particular machine. For example, network byte order is big-endian, while many popular CPUs are little-endian. By defining this explicitly, the user need not care about the specifics of the platform their code is running on. The first character should typically be `<` or `>` (or `!`). Padding is the responsibility of the programmer. The zero-repeat format character won’t work. Instead, the user must explicitly add `'x'` pad bytes where needed. Revisiting the examples from the previous section, we have:

```
>>> calcsize('<qh6xq')
24
>>> pack('<qh6xq', 1, 2, 3) == pack('@lhl', 1, 2, 3)
True
>>> calcsize('@llh')
18
>>> pack('@llh', 1, 2, 3) == pack('<qqh', 1, 2, 3)
True
>>> calcsize('<qqh6x')
24
>>> calcsize('@llh0l')
24
>>> pack('@llh0l', 1, 2, 3) == pack('<qqh6x', 1, 2, 3)
True
```

The above results (executed on a 64-bit machine) aren't guaranteed to match when executed on different machines. For example, the examples below were executed on a 32-bit machine:

```
>>> calcsize('<qqh6x')
24
>>> calcsize('@llh0l')
12
>>> pack('@llh0l', 1, 2, 3) == pack('<qqh6x', 1, 2, 3)
False
```

Classes

The `struct` module also defines the following type:

class struct.Struct(*format*)

Return a new Struct object which writes and reads binary data according to the format string *format*. Creating a `Struct` object once and calling its methods is more efficient than calling module-level functions with the same format since the format string is only compiled once.

Note

The compiled versions of the most recent format strings passed to the module-level functions are cached, so programs that use only a few format strings needn't worry about reusing a single `Struct` instance.

Compiled Struct objects support the following methods and attributes:

pack(*v1*, *v2*, *...*)

Identical to the `pack()` function, using the compiled format. (`len(result)` will equal `size`.)

pack_into(*buffer*, *offset*, *v1*, *v2*, *...*)

Identical to the `pack_into()` function, using the compiled format.

unpack(*buffer*)

Identical to the `unpack()` function, using the compiled format. The buffer's size in bytes must equal `size`.

unpack_from(*buffer*, *offset=0*)

Identical to the `unpack_from()` function, using the compiled format. The buffer's size in bytes, starting at position *offset*, must be at least `size`.

iter_unpack(*buffer*)

Identical to the `iter_unpack()` function, using the compiled format. The buffer's size in bytes must be a multiple of `size`.

Added in version 3.4.

format

The format string used to construct this Struct object.

Changed in version 3.7: The format string type is now `str` instead of `bytes`.

size

The calculated size of the struct (and hence of the bytes object produced by the `pack()` method) corresponding to `format`.

Changed in version 3.13: The *repr()* of structs has changed. It is now:

```
>>> Struct('i')
Struct('i')
```

`codecs` — Codec registry and base classes

Source code: Lib/codecs.py
[https://github.com/python/cpython/tree/3.14/Lib/codecs.py]

This module defines base classes for standard Python codecs (encoders and decoders) and provides access to the internal Python codec registry, which manages the codec and error handling lookup process. Most standard codecs are text encodings, which encode text to bytes (and decode bytes to text), but there are also codecs provided that encode text to text, and bytes to bytes. Custom codecs may encode and decode between arbitrary types, but some module features are restricted to be used specifically with text encodings or with codecs that encode to **`bytes`**.

The module defines the following functions for encoding and decoding with any codec:

codecs.encode(*obj*, *encoding='utf-8'*, *errors='strict'*)

Encodes *obj* using the codec registered for *encoding*.

Errors may be given to set the desired error handling scheme. The default error handler is `'strict'` meaning that encoding errors raise **`ValueError`** (or a more codec specific subclass, such as **`UnicodeEncodeError`**). Refer to Codec Base Classes for more information on codec error handling.

codecs.decode(*obj*, *encoding='utf-8'*, *errors='strict'*)

Decodes *obj* using the codec registered for *encoding*.

Errors may be given to set the desired error handling scheme. The default error handler is `'strict'` meaning that decoding errors raise **`ValueError`** (or a more codec specific subclass, such as **`UnicodeDecodeError`**). Refer to Codec Base Classes for more information on codec error handling.

codecs.charmap_build(*string*)

Return a mapping suitable for encoding with a custom single-byte encoding. Given a **`str`** *string* of up to 256 characters representing a decoding table, returns either a compact internal mapping object `EncodingMap` or a **`dictionary`** mapping character ordinals to byte values. Raises a **`TypeError`** on invalid input.

The full details for each codec can also be looked up directly:

codecs.lookup(*encoding*, */*)

Looks up the codec info in the Python codec registry and returns a **`CodecInfo`** object as defined below.

Encodings are first looked up in the registry's cache. If not found, the list of registered search functions is scanned. If no **`CodecInfo`** object is found, a **`LookupError`** is raised. Otherwise, the **`CodecInfo`** object is stored in the cache and returned to the caller.

class codecs.CodecInfo(*encode*, *decode*, *streamreader=None*, *streamwriter=None*, *incrementalencoder=None*, *incrementaldecoder=None*, *name=None*)

Codec details when looking up the codec registry. The constructor arguments are stored in attributes of the same name:

name

The name of the encoding.

encode
decode

The stateless encoding and decoding functions. These must be functions or methods which have the same interface as the **`encode()`** and **`decode()`** methods of Codec instances (see Codec Interface). The functions or methods are expected to work in a stateless mode.

incrementalencoder
incrementaldecoder

Incremental encoder and decoder classes or factory functions. These have to provide the interface defined by the base classes **`IncrementalEncoder`** and **`IncrementalDecoder`**, respectively. Incremental codecs can maintain state.

streamwriter
streamreader

Stream writer and reader classes or factory functions. These have to provide the interface defined by the base classes `StreamWriter` and `StreamReader`, respectively. Stream codecs can maintain state.

To simplify access to the various codec components, the module provides these additional functions which use `lookup()` for the codec lookup:

codecs.getencoder(*encoding*)

Look up the codec for the given encoding and return its encoder function.

Raises a `LookupError` in case the encoding cannot be found.

codecs.getdecoder(*encoding*)

Look up the codec for the given encoding and return its decoder function.

Raises a `LookupError` in case the encoding cannot be found.

codecs.getincrementalencoder(*encoding*)

Look up the codec for the given encoding and return its incremental encoder class or factory function.

Raises a `LookupError` in case the encoding cannot be found or the codec doesn’t support an incremental encoder.

codecs.getincrementaldecoder(*encoding*)

Look up the codec for the given encoding and return its incremental decoder class or factory function.

Raises a `LookupError` in case the encoding cannot be found or the codec doesn’t support an incremental decoder.

codecs.getreader(*encoding*)

Look up the codec for the given encoding and return its `StreamReader` class or factory function.

Raises a `LookupError` in case the encoding cannot be found.

codecs.getwriter(*encoding*)

Look up the codec for the given encoding and return its `StreamWriter` class or factory function.

Raises a `LookupError` in case the encoding cannot be found.

Custom codecs are made available by registering a suitable codec search function:

codecs.register(*search_function*, */*)

Register a codec search function. Search functions are expected to take one argument, being the encoding name in all lower case letters with hyphens and spaces converted to underscores, and return a `CodecInfo` object. In case a search function cannot find a given encoding, it should return `None`.

Changed in version 3.9: Hyphens and spaces are converted to underscore.

codecs.unregister(*search_function*, */*)

Unregister a codec search function and clear the registry’s cache. If the search function is not registered, do nothing.

Added in version 3.10.

While the builtin `open()` and the associated `io` module are the recommended approach for working with encoded text files, this module provides additional utility functions and classes that allow the use of a wider range of codecs when working with binary files:

codecs.open(*filename*, *mode='r'*, *encoding=None*, *errors='strict'*, *buffering=-1*)

Open an encoded file using the given *mode* and return an instance of `StreamReaderWriter`, providing transparent encoding/decoding. The default file mode is `'r'`, meaning to open the file in read mode.

Note

If *encoding* is not `None`, then the underlying encoded files are always opened in binary mode. No automatic conversion of `'\n'` is done on reading and writing. The *mode* argument may be any binary mode acceptable to the built-in `open()` function; the `'b'` is automatically added.

encoding specifies the encoding which is to be used for the file. Any encoding that encodes to and decodes from bytes is allowed, and the data types supported by the file methods depend on the codec used.

errors may be given to define the error handling. It defaults to `'strict'` which causes a `ValueError` to be raised in case an encoding error occurs.

buffering has the same meaning as for the built-in `open()` function. It defaults to -1 which means that the default buffer size will be used.

Changed in version 3.11: The `'U'` mode has been removed.

Deprecated since version 3.14: `codecs.open()` has been superseded by `open()`.

codecs.EncodedFile(*file*, *data_encoding*, *file_encoding=None*, *errors='strict'*)

Return a `StreamRecoder` instance, a wrapped version of *file* which provides transparent transcoding. The original file is closed when the wrapped version is closed.

Data written to the wrapped file is decoded according to the given *data_encoding* and then written to the original file as bytes using *file_encoding*. Bytes read from the original file are decoded according to *file_encoding*, and the result is encoded using *data_encoding*.

If *file_encoding* is not given, it defaults to *data_encoding*.

errors may be given to define the error handling. It defaults to `'strict'`, which causes `ValueError` to be raised in case an encoding error occurs.

codecs.iterencode(*iterator*, *encoding*, *errors*='*strict*', ***kwargs*)

Uses an incremental encoder to iteratively encode the input provided by *iterator*. *iterator* must yield **`str`** objects. This function is a generator. The *errors* argument (as well as any other keyword argument) is passed through to the incremental encoder.

This function requires that the codec accept text **`str`** objects to encode. Therefore it does not support bytes-to-bytes encoders such as `base64_codec`.

codecs.iterdecode(*iterator*, *encoding*, *errors*='*strict*', ***kwargs*)

Uses an incremental decoder to iteratively decode the input provided by *iterator*. *iterator* must yield **`bytes`** objects. This function is a generator. The *errors* argument (as well as any other keyword argument) is passed through to the incremental decoder.

This function requires that the codec accept **`bytes`** objects to decode. Therefore it does not support text-to-text encoders such as `rot_13`, although `rot_13` may be used equivalently with **`iterencode()`**.

codecs.readbuffer_encode(*buffer*, *errors*=*None*, */*)

Return a **`tuple`** containing the raw bytes of *buffer*, a buffer-compatible object or **`str`** (encoded to UTF-8 before processing), and their length in bytes.

The *errors* argument is ignored.

```
>>> codecs.readbuffer_encode(b"Zito")
(b'Zito', 4)
```

The module also provides the following constants which are useful for reading and writing to platform dependent files:

codecs.BOM
codecs.BOM_BE
codecs.BOM_LE
codecs.BOM_UTF8
codecs.BOM_UTF16
codecs.BOM_UTF16_BE
codecs.BOM_UTF16_LE
codecs.BOM_UTF32

codecs.BOM_UTF32_BE
codecs.BOM_UTF32_LE

These constants define various byte sequences, being Unicode byte order marks (BOMs) for several encodings. They are used in UTF-16 and UTF-32 data streams to indicate the byte order used, and in UTF-8 as a Unicode signature. `BOM_UTF16` is either `BOM_UTF16_BE` or `BOM_UTF16_LE` depending on the platform's native byte order, `BOM` is an alias for `BOM_UTF16`, `BOM_LE` for `BOM_UTF16_LE` and `BOM_BE` for `BOM_UTF16_BE`. The others represent the BOM in UTF-8 and UTF-32 encodings.

Codec Base Classes

The `codecs` module defines a set of base classes which define the interfaces for working with codec objects, and can also be used as the basis for custom codec implementations.

Each codec has to define four interfaces to make it usable as codec in Python: stateless encoder, stateless decoder, stream reader and stream writer. The stream reader and writers typically reuse the stateless encoder/decoder to implement the file protocols. Codec authors also need to define how the codec will handle encoding and decoding errors.

Error Handlers

To simplify and standardize error handling, codecs may implement different error handling schemes by accepting the *errors* string argument:

```
>>> 'German ß, ♬'.encode(encoding='ascii', errors='backslashreplace')
b'German \\xdf, \\u266c'
>>> 'German ß, ♬'.encode(encoding='ascii', errors='xmlcharrefreplace')
b'German &#223;, &#9836;'
```

The following error handlers can be used with all Python Standard Encodings codecs:

Value	Meaning
`'strict'`	Raise `UnicodeError` (or a subclass), this is the default. Implemented in `strict_errors()`.
`'ignore'`	Ignore the malformed data and continue without further notice. Implemented in `ignore_errors()`.
`'replace'`	Replace with a replacement marker. On encoding, use `?` (ASCII character). On decoding, use `�` (U+FFFD, the official REPLACEMENT CHARACTER). Implemented in `replace_errors()`.
`'backslashreplace'`	Replace with backslashed escape sequences. On encoding, use hexadecimal form of Unicode code point with formats `\xhh` `\uxxxx` `\Uxxxxxxxx`. On decoding, use hexadecimal form of byte value with format `\xhh`. Implemented in `backslashreplace_errors()`.
`'surrogateescape'`	On decoding, replace byte with individual surrogate code ranging from `U+DC80` to `U+DCFF`. This code will then be turned back into the same byte when the `'surrogateescape'` error handler is used when encoding the data. (See **PEP 383** for more.)

The following error handlers are only applicable to encoding (within text encodings):

Value	Meaning
`'xmlcharrefreplace'`	Replace with XML/HTML numeric character reference, which is a decimal form of Unicode

Value	**Meaning**
	code point with format `&#num;`. Implemented in `xmlcharrefreplace_errors()`.
`'namereplace'`	Replace with `\N{...}` escape sequences, what appears in the braces is the Name property from Unicode Character Database. Implemented in `namereplace_errors()`.

In addition, the following error handler is specific to the given codecs:

Value	**Codecs**	**Meaning**
`'surrogatepass'`	utf-8, utf-16, utf-32, utf-16-be, utf-16-le, utf-32-be, utf-32-le	Allow encoding and decoding surrogate code point (`U+D800` - `U+DFFF`) as normal code point. Otherwise these codecs treat the presence of surrogate code point in `str` as an error.

Added in version 3.1: The `'surrogateescape'` and `'surrogatepass'` error handlers.

Changed in version 3.4: The `'surrogatepass'` error handler now works with utf-16* and utf-32* codecs.

Added in version 3.5: The `'namereplace'` error handler.

Changed in version 3.5: The `'backslashreplace'` error handler now works with decoding and translating.

The set of allowed values can be extended by registering a new named error handler:

codecs.register_error(*name*, *error_handler*, */*)

Register the error handling function *error_handler* under the name *name*. The *error_handler* argument will be called during encoding and decoding in case of an error, when *name* is specified as the errors parameter.

For encoding, *error_handler* will be called with a **`UnicodeEncodeError`** instance, which contains information about the location of the error. The error handler must either raise this or a different exception, or return a tuple with a replacement for the unencodable part of the input and a position where encoding should continue. The replacement may be either **`str`** or **`bytes`**. If the replacement is bytes, the encoder will simply copy them into the output buffer. If the replacement is a string, the encoder will encode the replacement. Encoding continues on original input at the specified position. Negative position values will be treated as being relative to the end of the input string. If the resulting position is out of bound an **`IndexError`** will be raised.

Decoding and translating works similarly, except **`UnicodeDecodeError`** or **`UnicodeTranslateError`** will be passed to the handler and that the replacement from the error handler will be put into the output directly.

Previously registered error handlers (including the standard error handlers) can be looked up by name:

codecs.lookup_error(*name*, /)

Return the error handler previously registered under the name *name*.

Raises a **`LookupError`** in case the handler cannot be found.

The following standard error handlers are also made available as module level functions:

codecs.strict_errors(*exception*)

Implements the `'strict'` error handling.

Each encoding or decoding error raises a **`UnicodeError`**.

codecs.ignore_errors(*exception*)

Implements the `'ignore'` error handling.

Malformed data is ignored; encoding or decoding is continued without further notice.

codecs.replace_errors(*exception*)

Implements the `'replace'` error handling.

Substitutes `?` (ASCII character) for encoding errors or � (U+FFFD, the official REPLACEMENT CHARACTER) for decoding errors.

codecs.backslashreplace_errors(*exception*)

Implements the `'backslashreplace'` error handling.

Malformed data is replaced by a backslashed escape sequence. On encoding, use the hexadecimal form of Unicode code point with formats `\xhh` `\uxxxx` `\Uxxxxxxxx`. On decoding, use the hexadecimal form of byte value with format `\xhh`.

Changed in version 3.5: Works with decoding and translating.

codecs.xmlcharrefreplace_errors(*exception*)

Implements the `'xmlcharrefreplace'` error handling (for encoding within text encoding only).

The unencodable character is replaced by an appropriate XML/HTML numeric character reference, which is a decimal form of Unicode code point with format `&#num;` .

codecs.namereplace_errors(*exception*)

Implements the `'namereplace'` error handling (for encoding within text encoding only).

The unencodable character is replaced by a `\N{...}` escape sequence. The set of characters that appear in the braces is the Name property from Unicode Character Database. For example, the German lowercase letter `'ß'` will be converted to byte sequence `\N{LATIN SMALL LETTER SHARP S}` .

Added in version 3.5.

Stateless Encoding and Decoding

The base **`Codec`** class defines these methods which also define the function interfaces of the stateless encoder and decoder:

class codecs.Codec

encode(*input*, *errors='strict'*)

Encodes the object *input* and returns a tuple (output object, length consumed). For instance, text encoding converts a string object to a bytes object using a particular character set encoding (e.g., `cp1252` or `iso-8859-1`).

The *errors* argument defines the error handling to apply. It defaults to `'strict'` handling.

The method may not store state in the **`Codec`** instance. Use **`StreamWriter`** for codecs which have to keep state in order to make encoding efficient.

The encoder must be able to handle zero length input and return an empty object of the output object type in this situation.

decode(*input*, *errors='strict'*)

Decodes the object *input* and returns a tuple (output object, length consumed). For instance, for a text encoding, decoding converts a bytes object encoded using a particular character set encoding to a string object.

For text encodings and bytes-to-bytes codecs, *input* must be a bytes object or one which provides the read-only buffer interface – for example, buffer objects and memory mapped files.

The *errors* argument defines the error handling to apply. It defaults to `'strict'` handling.

The method may not store state in the **`Codec`** instance. Use **`StreamReader`** for codecs which have to keep state in order to make decoding efficient.

The decoder must be able to handle zero length input and return an empty object of the output object type in this situation.

Incremental Encoding and Decoding

The `IncrementalEncoder` and `IncrementalDecoder` classes provide the basic interface for incremental encoding and decoding. Encoding/decoding the input isn’t done with one call to the stateless encoder/decoder function, but with multiple calls to the `encode()`/`decode()` method of the incremental encoder/decoder. The incremental encoder/decoder keeps track of the encoding/decoding process during method calls.

The joined output of calls to the `encode()`/`decode()` method is the same as if all the single inputs were joined into one, and this input was encoded/decoded with the stateless encoder/decoder.

IncrementalEncoder Objects

The `IncrementalEncoder` class is used for encoding an input in multiple steps. It defines the following methods which every incremental encoder must define in order to be compatible with the Python codec registry.

class codecs.IncrementalEncoder(*errors*='*strict*')

Constructor for an `IncrementalEncoder` instance.

All incremental encoders must provide this constructor interface. They are free to add additional keyword arguments, but only the ones defined here are used by the Python codec registry.

The `IncrementalEncoder` may implement different error handling schemes by providing the *errors* keyword argument. See Error Handlers for possible values.

The *errors* argument will be assigned to an attribute of the same name. Assigning to this attribute makes it possible to switch between different error handling strategies during the lifetime of the `IncrementalEncoder` object.

encode(*object*, *final*=*False*)

Encodes *object* (taking the current state of the encoder into account) and returns the resulting encoded object. If this is the last call to `encode()` *final* must be true (the default is false).

reset()

Reset the encoder to the initial state. The output is discarded: call `.encode(object, final=True)`, passing an empty byte or text string if necessary, to reset the encoder and to get the output.

getstate()

Return the current state of the encoder which must be an integer. The implementation should make sure that `0` is the most common state. (States that are more complicated than integers can be converted into an integer by marshaling/pickling the state and encoding the bytes of the resulting string into an integer.)

setstate(*state*)

Set the state of the encoder to *state*. *state* must be an encoder state returned by **`getstate()`**.

IncrementalDecoder Objects

The **`IncrementalDecoder`** class is used for decoding an input in multiple steps. It defines the following methods which every incremental decoder must define in order to be compatible with the Python codec registry.

class codecs.IncrementalDecoder(*errors*='*strict*')

Constructor for an **`IncrementalDecoder`** instance.

All incremental decoders must provide this constructor interface. They are free to add additional keyword arguments, but only the ones defined here are used by the Python codec registry.

The **`IncrementalDecoder`** may implement different error handling schemes by providing the *errors* keyword argument. See Error Handlers for possible values.

The *errors* argument will be assigned to an attribute of the same name. Assigning to this attribute makes it possible to switch between different error handling strategies during the lifetime of the **`IncrementalDecoder`** object.

decode(*object*, *final=False*)

Decodes *object* (taking the current state of the decoder into account) and returns the resulting decoded object. If this is the last call to `decode()` *final* must be true (the default is false). If *final* is true the decoder must decode the input completely and must flush all buffers. If this isn't possible (e.g. because of incomplete byte sequences at the end of the input) it must initiate error handling just like in the stateless case (which might raise an exception).

reset()

Reset the decoder to the initial state.

getstate()

Return the current state of the decoder. This must be a tuple with two items, the first must be the buffer containing the still undecoded input. The second must be an integer and can be additional state info. (The implementation should make sure that `0` is the most common additional state info.) If this additional state info is `0` it must be possible to set the decoder to the state which has no input buffered and `0` as the additional state info, so that feeding the previously buffered input to the decoder returns it to the previous state without producing any output. (Additional state info that is more complicated than integers can be converted into an integer by marshaling/pickling the info and encoding the bytes of the resulting string into an integer.)

setstate(*state*)

Set the state of the decoder to *state*. *state* must be a decoder state returned by `getstate()`.

Stream Encoding and Decoding

The `StreamWriter` and `StreamReader` classes provide generic working interfaces which can be used to implement new encoding submodules very easily. See **`encodings.utf_8`** for an example of how this is done.

StreamWriter Objects

The `StreamWriter` class is a subclass of `Codec` and defines the following methods which every stream writer must define in order to be compatible with the Python codec registry.

class codecs.StreamWriter(*stream*, *errors='strict'*)

Constructor for a `StreamWriter` instance.

All stream writers must provide this constructor interface. They are free to add additional keyword arguments, but only the ones defined here are used by the Python codec registry.

The *stream* argument must be a file-like object open for writing text or binary data, as appropriate for the specific codec.

The `StreamWriter` may implement different error handling schemes by providing the *errors* keyword argument. See Error Handlers for the standard error handlers the underlying stream codec may support.

The *errors* argument will be assigned to an attribute of the same name. Assigning to this attribute makes it possible to switch between different error handling strategies during the lifetime of the `StreamWriter` object.

write(*object*)

Writes the object’s contents encoded to the stream.

writelines(*list*)

Writes the concatenated iterable of strings to the stream (possibly by reusing the `write()` method). Infinite or very large iterables are not supported. The standard bytes-to-bytes codecs do not support this method.

reset()

Resets the codec buffers used for keeping internal state.

Calling this method should ensure that the data on the output is put into a clean state that allows appending of new fresh data without having to rescan the whole stream to recover state.

In addition to the above methods, the `StreamWriter` must also inherit all other methods and attributes from the underlying stream.

StreamReader Objects

The `StreamReader` class is a subclass of `Codec` and defines the following methods which every stream reader must define in order to be compatible with the Python codec registry.

class codecs.StreamReader(*stream*, *errors*='*strict*')

Constructor for a `StreamReader` instance.

All stream readers must provide this constructor interface. They are free to add additional keyword arguments, but only the ones defined here are used by the Python codec registry.

The *stream* argument must be a file-like object open for reading text or binary data, as appropriate for the specific codec.

The `StreamReader` may implement different error handling schemes by providing the *errors* keyword argument. See Error Handlers for the standard error handlers the underlying stream codec may support.

The *errors* argument will be assigned to an attribute of the same name. Assigning to this attribute makes it possible to switch between different error handling strategies during the lifetime of the `StreamReader` object.

The set of allowed values for the *errors* argument can be extended with `register_error()`.

read(*size=-1*, *chars=-1*, *firstline=False*)

Decodes data from the stream and returns the resulting object.

The *chars* argument indicates the number of decoded code points or bytes to return. The `read()` method will never return more data than requested, but it might return less, if there is not enough available.

The *size* argument indicates the approximate maximum number of encoded bytes or code points to read for decoding.

The decoder can modify this setting as appropriate. The default value -1 indicates to read and decode as much as possible. This parameter is intended to prevent having to decode huge files in one step.

The *firstline* flag indicates that it would be sufficient to only return the first line, if there are decoding errors on later lines.

The method should use a greedy read strategy meaning that it should read as much data as is allowed within the definition of the encoding and the given size, e.g. if optional encoding endings or state markers are available on the stream, these should be read too.

readline(*size=None*, *keepends=True*)

Read one line from the input stream and return the decoded data.

size, if given, is passed as size argument to the stream’s `read()` method.

If *keepends* is false line-endings will be stripped from the lines returned.

readlines(*sizehint=None*, *keepends=True*)

Read all lines available on the input stream and return them as a list of lines.

Line-endings are implemented using the codec’s `decode()` method and are included in the list entries if *keepends* is true.

sizehint, if given, is passed as the *size* argument to the stream’s `read()` method.

reset()

Resets the codec buffers used for keeping internal state.

Note that no stream repositioning should take place. This method is primarily intended to be able to recover from decoding errors.

In addition to the above methods, the `StreamReader` must also inherit all other methods and attributes from the underlying stream.

StreamReaderWriter Objects

The `StreamReaderWriter` is a convenience class that allows wrapping streams which work in both read and write modes.

The design is such that one can use the factory functions returned by the `lookup()` function to construct the instance.

class codecs.StreamReaderWriter(*stream*, *Reader*, *Writer*, *errors='strict'*)

> Creates a `StreamReaderWriter` instance. *stream* must be a file-like object. *Reader* and *Writer* must be factory functions or classes providing the `StreamReader` and `StreamWriter` interface resp. Error handling is done in the same way as defined for the stream readers and writers.

`StreamReaderWriter` instances define the combined interfaces of `StreamReader` and `StreamWriter` classes. They inherit all other methods and attributes from the underlying stream.

StreamRecoder Objects

The `StreamRecoder` translates data from one encoding to another, which is sometimes useful when dealing with different encoding environments.

The design is such that one can use the factory functions returned by the `lookup()` function to construct the instance.

class codecs.StreamRecoder(*stream*, *encode*, *decode*, *Reader*, *Writer*, *errors='strict'*)

> Creates a `StreamRecoder` instance which implements a two-way conversion: *encode* and *decode* work on the frontend — the data visible to code calling `read()` and `write()`, while *Reader* and *Writer* work on the backend — the data in *stream*.
>
> You can use these objects to do transparent transcodings, e.g., from Latin-1 to UTF-8 and back.

The *stream* argument must be a file-like object.

The *encode* and *decode* arguments must adhere to the **`Codec`** interface. *Reader* and *Writer* must be factory functions or classes providing objects of the **`StreamReader`** and **`StreamWriter`** interface respectively.

Error handling is done in the same way as defined for the stream readers and writers.

`StreamRecoder` instances define the combined interfaces of **`StreamReader`** and **`StreamWriter`** classes. They inherit all other methods and attributes from the underlying stream.

Encodings and Unicode

Strings are stored internally as sequences of code points in range `U+0000`–`U+10FFFF`. (See **PEP 393** [https://peps.python.org/pep-0393/] for more details about the implementation.) Once a string object is used outside of CPU and memory, endianness and how these arrays are stored as bytes become an issue. As with other codecs, serialising a string into a sequence of bytes is known as *encoding*, and recreating the string from the sequence of bytes is known as *decoding*. There are a variety of different text serialisation codecs, which are collectivity referred to as text encodings.

The simplest text encoding (called `'latin-1'` or `'iso-8859-1'`) maps the code points 0–255 to the bytes `0x0`–`0xff`, which means that a string object that contains code points above `U+00FF` can't be encoded with this codec. Doing so will raise a **`UnicodeEncodeError`** that looks like the following (although the details of the error message may differ): `UnicodeEncodeError: 'latin-1' codec can't encode character '\u1234' in position 3: ordinal not in range(256)`.

There's another group of encodings (the so called charmap encodings) that choose a different subset of all Unicode code points and how these code points are mapped to the bytes `0x0`–`0xff`. To see how this is done simply open e.g. `encodings/cp1252.py` (which is an encoding that is used primarily on Windows). There's a string constant with 256 characters that shows you which character is mapped to which byte value.

All of these encodings can only encode 256 of the 1114112 code points defined in Unicode. A simple and straightforward way that can store each Unicode code point, is to store each code point as four consecutive bytes. There are two possibilities: store the bytes in big endian or in little endian order. These two encodings are called `UTF-32-BE` and `UTF-32-LE` respectively. Their disadvantage is that if, for example, you use `UTF-32-BE` on a little endian machine you will always have to swap bytes on encoding and decoding. Python's `UTF-16` and `UTF-32` codecs avoid this problem by using the platform's native byte order when no BOM is present. Python follows prevailing platform practice, so native-endian data round-trips without redundant byte swapping, even though the Unicode Standard defaults to big-endian when the byte order is unspecified. When these bytes are read by a CPU with a different endianness, the bytes have to be swapped. To be able to detect the endianness of a `UTF-16` or `UTF-32` byte sequence, a BOM ("Byte Order Mark") is used. This is the Unicode character `U+FEFF`. This character can be prepended to every `UTF-16` or `UTF-32` byte sequence. The byte swapped version of this character (`0xFFFE`) is an illegal character that may not appear in a Unicode text. When the first character of a `UTF-16` or `UTF-32` byte sequence is `U+FFFE`, the bytes have to be swapped on decoding.

Unfortunately the character `U+FEFF` had a second purpose as a `ZERO WIDTH NO-BREAK SPACE`: a character that has no width and doesn't allow a word to be split. It can e.g. be used to give hints to a ligature algorithm. With Unicode 4.0 using `U+FEFF` as a `ZERO WIDTH NO-BREAK SPACE` has been deprecated (with `U+2060` (`WORD JOINER`) assuming this role). Nevertheless Unicode software still must be able to handle `U+FEFF` in both roles: as a BOM it's a device to determine the storage layout of the encoded bytes, and vanishes once the byte sequence has been decoded into a string; as a `ZERO WIDTH NO-BREAK SPACE` it's a normal character that will be decoded like any other.

There's another encoding that is able to encode the full range of Unicode characters: UTF-8. UTF-8 is an 8-bit encoding, which means there are no issues with byte order in UTF-8. Each byte in a UTF-8 byte sequence consists of two parts: marker bits (the most significant bits) and payload bits. The marker bits are a sequence of zero to four `1` bits followed by a `0` bit. Unicode characters are encoded like this (with x being payload bits, which when concatenated give the Unicode character):

Range	Encoding
`U-00000000` … `U-0000007F`	0xxxxxxx
`U-00000080` … `U-000007FF`	110xxxxx 10xxxxxx
`U-00000800` … `U-0000FFFF`	1110xxxx 10xxxxxx 10xxxxxx
`U-00010000` … `U-0010FFFF`	11110xxx 10xxxxxx 10xxxxxx 10xxxxxx

The least significant bit of the Unicode character is the rightmost x bit.

As UTF-8 is an 8-bit encoding no BOM is required and any `U+FEFF` character in the decoded string (even if it's the first character) is treated as a `ZERO WIDTH NO-BREAK SPACE`.

Without external information it's impossible to reliably determine which encoding was used for encoding a string. Each charmap encoding can decode any random byte sequence. However that's not possible with UTF-8, as UTF-8 byte sequences have a structure that doesn't allow arbitrary byte sequences. To increase the reliability with which a UTF-8 encoding can be detected, Microsoft invented a variant of UTF-8 (that Python calls `"utf-8-sig"`) for its Notepad program: Before any of the Unicode characters is written to the file, a UTF-8 encoded BOM (which looks like this as a byte sequence: `0xef`, `0xbb`, `0xbf`) is written. As it's rather improbable that any charmap encoded file starts with these byte values (which would e.g. map to

> LATIN SMALL LETTER I WITH DIAERESIS
> RIGHT-POINTING DOUBLE ANGLE QUOTATION MARK
> INVERTED QUESTION MARK

in iso-8859-1), this increases the probability that a `utf-8-sig` encoding can be correctly guessed from the byte sequence. So here the BOM is not used to be able to determine the byte order used for generating the byte sequence, but as a signature that helps in guessing the encoding. On encoding the utf-8-sig codec will write `0xef`, `0xbb`, `0xbf` as the first three bytes to the file. On decoding `utf-8-sig` will skip those three bytes if they appear as the first three bytes in the file. In UTF-8, the use of the BOM is discouraged and should generally be avoided.

Standard Encodings

Python comes with a number of codecs built-in, either implemented as C functions or with dictionaries as mapping tables. The following table lists the codecs by name, together with a few common aliases, and the languages for which the encoding is likely used. Neither the list of aliases nor the list of languages is meant to be exhaustive. Notice that spelling alternatives that only differ in case or use a hyphen instead of an underscore are also valid aliases because they are equivalent when normalized by `normalize_encoding()`. For example, `'utf-8'` is a valid alias for the `'utf_8'` codec.

Note

The below table lists the most common aliases, for a complete list refer to the source aliases.py [https://github.com/python/cpython/tree/3.14/Lib/encodings/aliases.py] file.

On Windows, `cpXXX` codecs are available for all code pages. But only codecs listed in the following table are guarantead to exist on other platforms.

CPython implementation detail: Some common encodings can bypass the codecs lookup machinery to improve performance. These optimization opportunities are only recognized by CPython for a limited set of (case insensitive) aliases: utf-8, utf8, latin-1, latin1, iso-8859-1, iso8859-1, mbcs (Windows only), ascii, us-ascii, utf-16, utf16, utf-32, utf32, and the same using underscores instead of dashes. Using alternative aliases for these encodings may result in slower execution.

Changed in version 3.6: Optimization opportunity recognized for us-ascii.

Many of the character sets support the same languages. They vary in individual characters (e.g. whether the EURO SIGN is supported or not), and in the assignment of characters to code positions. For the European languages in particular, the following variants typically exist:

- an ISO 8859 codeset
- a Microsoft Windows code page, which is typically derived from an 8859 codeset, but replaces control characters with additional graphic characters
- an IBM EBCDIC code page
- an IBM PC code page, which is ASCII compatible

Codec	Aliases	Languages
ascii	646, us-ascii	English
big5	big5-tw, csbig5	Traditional Chinese
big5hkscs	big5-hkscs, hkscs	Traditional Chinese
cp037	IBM037, IBM039	English
cp273	273, IBM273, csIBM273	German *Added in version 3.4.*
cp424	EBCDIC-CP-HE, IBM424	Hebrew
cp437	437, IBM437	English
cp500	EBCDIC-CP-BE, EBCDIC-CP-CH, IBM500	Western Europe

Codec	Aliases	Languages
cp720		Arabic
cp737		Greek
cp775	IBM775	Baltic languages
cp850	850, IBM850	Western Europe
cp852	852, IBM852	Central and Eastern Europe
cp855	855, IBM855	Belarusian, Bulgarian, Macedonian, Russian, Serbian
cp856		Hebrew
cp857	857, IBM857	Turkish
cp858	858, IBM858	Western Europe
cp860	860, IBM860	Portuguese
cp861	861, CP-IS, IBM861	Icelandic
cp862	862, IBM862	Hebrew

Codec	Aliases	Languages
cp863	863, IBM863	Canadian
cp864	IBM864	Arabic
cp865	865, IBM865	Danish, Norwegian
cp866	866, IBM866	Russian
cp869	869, CP-GR, IBM869	Greek
cp874		Thai
cp875		Greek
cp932	932, ms932, mskanji, ms-kanji, windows-31j	Japanese
cp949	949, ms949, uhc	Korean
cp950	950, ms950	Traditional Chinese
cp1006		Urdu
cp1026	ibm1026	Turkish
cp1125	1125, ibm1125, cp866u, ruscii	Ukrainian

Codec	Aliases	Languages
		Added in version 3.4.
cp1140	ibm1140	Western Europe
cp1250	windows-1250	Central and Eastern Europe
cp1251	windows-1251	Belarusian, Bulgarian, Macedonian, Russian, Serbian
cp1252	windows-1252	Western Europe
cp1253	windows-1253	Greek
cp1254	windows-1254	Turkish
cp1255	windows-1255	Hebrew
cp1256	windows-1256	Arabic
cp1257	windows-1257	Baltic languages
cp1258	windows-1258	Vietnamese
euc_jp	eucjp, ujis, u-jis	Japanese

Codec	**Aliases**	**Languages**
euc_jis_2004	jisx0213, eucjis2004	Japanese
euc_jisx0213	eucjisx0213	Japanese
euc_kr	euckr, korean, ksc5601, ks_c-5601, ks_c-5601-1987, ksx1001, ks_x-1001	Korean
gb2312	chinese, csiso58gb231280, euc-cn, euccn, eucgb2312-cn, gb2312-1980, gb2312-80, iso-ir-58	Simplified Chinese
gbk	936, cp936, ms936	Unified Chinese
gb18030	gb18030-2000	Unified Chinese
hz	hzgb, hz-gb, hz-gb-2312	Simplified Chinese
iso2022_jp	csiso2022jp, iso2022jp, iso-2022-jp	Japanese
iso2022_jp_1	iso2022jp-1, iso-2022-jp-1	Japanese
iso2022_jp_2	iso2022jp-2, iso-2022-jp-2	Japanese, Korean, Simplified Chinese, Western Europe, Greek

Codec	Aliases	Languages
iso2022_jp_2004	iso2022jp-2004, iso-2022-jp-2004	Japanese
iso2022_jp_3	iso2022jp-3, iso-2022-jp-3	Japanese
iso2022_jp_ext	iso2022jp-ext, iso-2022-jp-ext	Japanese
iso2022_kr	csiso2022kr, iso2022kr, iso-2022-kr	Korean
latin_1	iso-8859-1, iso8859-1, 8859, cp819, latin, latin1, L1	Western Europe
iso8859_2	iso-8859-2, latin2, L2	Central and Eastern Europe
iso8859_3	iso-8859-3, latin3, L3	Esperanto, Maltese
iso8859_4	iso-8859-4, latin4, L4	Northern Europe
iso8859_5	iso-8859-5, cyrillic	Belarusian, Bulgarian, Macedonian, Russian, Serbian
iso8859_6	iso-8859-6, arabic	Arabic
iso8859_7	iso-8859-7, greek, greek8	Greek

Codec	Aliases	Languages
iso8859_8	iso-8859-8, hebrew	Hebrew
iso8859_9	iso-8859-9, latin5, L5	Turkish
iso8859_10	iso-8859-10, latin6, L6	Nordic languages
iso8859_11	iso-8859-11, thai	Thai languages
iso8859_13	iso-8859-13, latin7, L7	Baltic languages
iso8859_14	iso-8859-14, latin8, L8	Celtic languages
iso8859_15	iso-8859-15, latin9, L9	Western Europe
iso8859_16	iso-8859-16, latin10, L10	South-Eastern Europe
johab	cp1361, ms1361	Korean
koi8_r		Russian
koi8_t		Tajik *Added in version 3.5.*
koi8_u		Ukrainian

Codec	**Aliases**	**Languages**
kz1048	kz_1048, strk1048_2002, rk1048	Kazakh *Added in version 3.5.*
mac_cyrillic	maccyrillic	Belarusian, Bulgarian, Macedonian, Russian, Serbian
mac_greek	macgreek	Greek
mac_iceland	maciceland	Icelandic
mac_latin2	maclatin2, maccentraleurope, mac_centeuro	Central and Eastern Europe
mac_roman	macroman, macintosh	Western Europe
mac_turkish	macturkish	Turkish
ptcp154	csptcp154, pt154, cp154, cyrillic-asian	Kazakh
shift_jis	csshiftjis, shiftjis, sjis, s_jis	Japanese
shift_jis_2004	shiftjis2004, sjis_2004, sjis2004	Japanese

Codec	Aliases	Languages
shift_jisx0213	shiftjisx0213, sjisx0213, s_jisx0213	Japanese
utf_32	U32, utf32	all languages
utf_32_be	UTF-32BE	all languages
utf_32_le	UTF-32LE	all languages
utf_16	U16, utf16	all languages
utf_16_be	UTF-16BE	all languages
utf_16_le	UTF-16LE	all languages
utf_7	U7, unicode-1-1-utf-7	all languages
utf_8	U8, UTF, utf8, cp65001	all languages
utf_8_sig		all languages

Changed in version 3.4: The utf-16* and utf-32* encoders no longer allow surrogate code points (`U+D800`–`U+DFFF`) to be encoded. The utf-32* decoders no longer decode byte sequences that correspond to surrogate code points.

Changed in version 3.8: `cp65001` is now an alias to `utf_8`.

Changed in version 3.14: On Windows, `cpXXX` codecs are now available for all code pages.

Python Specific Encodings

A number of predefined codecs are specific to Python, so their codec names have no meaning outside Python. These are listed in the tables below based on the expected input and output types (note that while text encodings are the most common use case for codecs, the underlying codec infrastructure supports arbitrary data transforms rather than just text encodings). For asymmetric codecs, the stated meaning describes the encoding direction.

Text Encodings

The following codecs provide `str` to `bytes` encoding and bytes-like object to `str` decoding, similar to the Unicode text encodings.

Codec	Aliases	Meaning
idna		Implement **RFC 3490**, see also `encodings.idna`. Only `errors='strict'` is supported.
mbcs	ansi, dbcs	Windows only: Encode the operand according to the ANSI codepage (CP_ACP).
oem		Windows only: Encode the operand according to the OEM codepage (CP_OEMCP). *Added in version 3.6.*
palmos		Encoding of PalmOS 3.5.
punycode		Implement **RFC 3492**. Stateful codecs are not supported.

Codec	**Aliases**	**Meaning**
raw_unicode_escape		Latin-1 encoding with `\u`*xxxx* and `\U`*xxxxxxxx* for other code points. Existing backslashes are not escaped in any way. It is used in the Python pickle protocol.
undefined		This Codec should only be used for testing purposes. Raise an exception for all conversions, even empty strings. The error handler is ignored.
unicode_escape		Encoding suitable as the contents of a Unicode literal in ASCII-encoded Python source code, except that quotes are not escaped. Decode from Latin-1 source code. Beware that Python source code actually uses UTF-8 by default.

Changed in version 3.8: “unicode_internal” codec is removed.

Binary Transforms

The following codecs provide binary transforms: bytes-like object to `bytes` mappings. They are not supported by `bytes.decode()` (which only produces `str` output).

Codec	**Aliases**	**Meaning**	**Encoder / decoder**
base64_codec [[1]]	base64, base_64	Convert the operand to	`base64.encodebytes()` /

Codec	Aliases	Meaning	Encoder / decoder
		multiline MIME base64 (the result always includes a trailing `'\n'`). *Changed in version 3.4:* accepts any bytes-like object as input for encoding and decoding	`base64.decodebytes()`
bz2_codec	bz2	Compress the operand using bz2.	`bz2.compress()` / `bz2.decompress()`
hex_codec	hex	Convert the operand to hexadecimal representation, with two digits per byte.	`binascii.b2a_hex()` / `binascii.a2b_hex()`
quopri_codec	quopri, quotedprintable, quoted_printable	Convert the operand to MIME quoted printable.	`quopri.encode()` with `quotetabs=True` / `quopri.decode()`
uu_codec	uu	Convert the operand using uuencode.	

Codec	**Aliases**	**Meaning**	**Encoder / decoder**
zlib_codec	zip, zlib	Compress the operand using gzip.	`zlib.compress()` / `zlib.decompress()`

[1]

In addition to bytes-like objects, `'base64_codec'` also accepts ASCII-only instances of `str` for decoding

Added in version 3.2: Restoration of the binary transforms.

Changed in version 3.4: Restoration of the aliases for the binary transforms.

Standalone Codec Functions

The following functions provide encoding and decoding functionality similar to codecs, but are not available as named codecs through `codecs.encode()` or `codecs.decode()`. They are used internally (for example, by `pickle`) and behave similarly to the `string_escape` codec that was removed in Python 3.

codecs.escape_encode(*input*, *errors=None*)

Encode *input* using escape sequences. Similar to how `repr()` on bytes produces escaped byte values.

input must be a `bytes` object.

Returns a tuple `(output, length)` where *output* is a `bytes` object and *length* is the number of bytes consumed.

codecs.escape_decode(*input*, *errors=None*)

Decode *input* from escape sequences back to the original bytes.

input must be a bytes-like object.

Returns a tuple `(output, length)` where *output* is a `bytes` object and *length* is the number of bytes consumed.

Text Transforms

The following codec provides a text transform: a `str` to `str` mapping. It is not supported by `str.encode()` (which only produces `bytes` output).

Codec	Aliases	Meaning
rot_13	rot13	Return the Caesar-cypher encryption of the operand.

Added in version 3.2: Restoration of the `rot_13` text transform.

Changed in version 3.4: Restoration of the `rot13` alias.

`encodings` — Encodings package

This module implements the following functions:

encodings.normalize_encoding(*encoding*)

Normalize encoding name *encoding*.

Normalization works as follows: all non-alphanumeric characters except the dot used for Python package names are collapsed and replaced with a single underscore, leading and trailing underscores are removed. For example, `'  -;#'` becomes `'_'`.

Note that *encoding* should be ASCII only.

Note

The following functions should not be used directly, except for testing purposes; `codecs.lookup()` should be used instead.

encodings.search_function(*encoding*)

Search for the codec module corresponding to the given encoding name *encoding*.

This function first normalizes the *encoding* using **`normalize_encoding()`**, then looks for a corresponding alias. It attempts to import a codec module from the encodings package using either the alias or the normalized name. If the module is found and defines a valid `getregentry()` function that returns a **`codecs.CodecInfo`** object, the codec is cached and returned.

If the codec module defines a `getaliases()` function any returned aliases are registered for future use.

encodings.win32_code_page_search_function(*encoding*)

Search for a Windows code page encoding *encoding* of the form `cpXXXX`.

If the code page is valid and supported, return a **`codecs.CodecInfo`** object for it.

Availability: Windows.

Added in version 3.14.

This module implements the following exception:

exception encodings.CodecRegistryError

Raised when a codec is invalid or incompatible.

`encodings.idna` — Internationalized Domain Names in Applications

This module implements **RFC 3490** [https://datatracker.ietf.org/doc/html/rfc3490.html] (Internationalized Domain Names in Applications) and **RFC 3492** [https://datatracker.ietf.org/doc/html/rfc3492.html] (Nameprep: A Stringprep Profile for Internationalized Domain Names (IDN)). It builds upon the `punycode` encoding and **`stringprep`**.

If you need the IDNA 2008 standard from **RFC 5891** [https://datatracker.ietf.org/doc/html/rfc5891.html] and **RFC 5895** [https://datatracker.ietf.org/doc/html/rfc5895.html], use the third-party idna [https://pypi.org/project/idna/] module.

These RFCs together define a protocol to support non-ASCII characters in domain names. A domain name containing non-ASCII characters (such as `www.Alliancefrançaise.nu`) is converted into an ASCII-compatible encoding (ACE, such as `www.xn--alliancefranaise-npb.nu`). The ACE form of the domain name is then used in all places where arbitrary characters are not allowed by the protocol, such as DNS queries, HTTP *Host* fields, and so on. This conversion is carried out in the application; if possible invisible to the user: The application should transparently convert Unicode domain labels to IDNA on the wire, and convert back ACE labels to Unicode before presenting them to the user.

Python supports this conversion in several ways: the `idna` codec performs conversion between Unicode and ACE, separating an input string into labels based on the separator characters defined in **section 3.1 of RFC 3490** [https://datatracker.ietf.org/doc/html/rfc3490.html#section-3.1] and converting each label to ACE as required, and conversely separating an input byte string into labels based on the `.` separator and converting any ACE labels found into unicode. Furthermore, the **`socket`** module transparently converts Unicode host names to ACE, so that applications need not be concerned about converting host names themselves when they pass them to the socket module. On top of that, modules that have host names as function parameters, such as **`http.client`** and **`ftplib`**, accept Unicode host names (**`http.client`** then also transparently sends an IDNA hostname in the *Host* field if it sends that field at all).

When receiving host names from the wire (such as in reverse name lookup), no automatic conversion to Unicode is performed: applications wishing to present such host names to the user should decode them to Unicode.

The module **`encodings.idna`** also implements the nameprep procedure, which performs certain normalizations on host names, to achieve case-insensitivity of international domain names, and to unify similar characters. The nameprep functions can be used directly if desired.

encodings.idna.nameprep(*label*)

Return the nameprepped version of *label*. The implementation currently assumes query strings, so `AllowUnassigned` is true.

encodings.idna.ToASCII(*label*)

Convert a label to ASCII, as specified in **RFC 3490** [https://datatracker.ietf.org/doc/html/rfc3490.html]. `UseSTD3ASCIIRules` is assumed to be false.

encodings.idna.ToUnicode(*label*)

Convert a label to Unicode, as specified in **RFC 3490** [https://datatracker.ietf.org/doc/html/rfc3490.html].

`encodings.mbcs` — Windows ANSI codepage

This module implements the ANSI codepage (CP_ACP).

Availability: Windows.

Changed in version 3.2: Before 3.2, the *errors* argument was ignored; `'replace'` was always used to encode, and `'ignore'` to decode.

Changed in version 3.3: Support any error handler.

`encodings.utf_8_sig` — UTF-8 codec with BOM signature

This module implements a variant of the UTF-8 codec. On encoding, a UTF-8 encoded BOM will be prepended to the UTF-8 encoded bytes. For the stateful encoder this is only done once (on the first write to the byte stream). On decoding, an optional UTF-8 encoded BOM at the start of the data will be skipped.

Data Types

The modules described in this chapter provide a variety of specialized data types such as dates and times, fixed-type arrays, heap queues, double-ended queues, and enumerations.

Python also provides some built-in data types, in particular, **`dict`**, **`list`**, **`set`** and **`frozenset`**, and **`tuple`**. The **`str`** class is used to hold Unicode strings, and the **`bytes`** and **`bytearray`** classes are used to hold binary data.

The following modules are documented in this chapter:

- **`datetime`** — Basic date and time types
  - Aware and Naive Objects
  - Constants
  - Available Types
    - Common Properties
    - Determining if an Object is Aware or Naive
  - **`timedelta`** Objects
    - Examples of usage: **`timedelta`**
  - **`date`** Objects
    - Examples of Usage: **`date`**
  - **`datetime`** Objects
    - Examples of Usage: **`datetime`**
  - **`time`** Objects
    - Examples of Usage: **`time`**
  - **`tzinfo`** Objects
  - **`timezone`** Objects
  - **`strftime()`** and **`strptime()`** Behavior
    - **`strftime()`** and **`strptime()`** Format Codes
    - Technical Detail
- **`zoneinfo`** — IANA time zone support

    - Using `ZoneInfo`
    - Data sources
        - Configuring the data sources
            - Compile-time configuration
            - Environment configuration
            - Runtime configuration
    - The `ZoneInfo` class
        - String representations
        - Pickle serialization
    - Functions
    - Globals
    - Exceptions and warnings
- `calendar` — General calendar-related functions
    - Command-line usage
- `collections` — Container datatypes
    - `ChainMap` objects
        - `ChainMap` Examples and Recipes
    - `Counter` objects
    - `deque` objects
        - `deque` Recipes
    - `defaultdict` objects
        - `defaultdict` Examples
    - `namedtuple()` Factory Function for Tuples with Named Fields
    - `OrderedDict` objects
        - `OrderedDict` Examples and Recipes
    - `UserDict` objects
    - `UserList` objects
    - `UserString` objects
- `collections.abc` — Abstract Base Classes for Containers
    - Collections Abstract Base Classes
    - Collections Abstract Base Classes – Detailed Descriptions
    - Examples and Recipes
- `heapq` — Heap queue algorithm
    - Basic Examples

  - Other Applications
  - Priority Queue Implementation Notes
  - Theory
- `bisect` — Array bisection algorithm
  - Performance Notes
  - Searching Sorted Lists
  - Examples
- `array` — Efficient arrays of numeric values
- `weakref` — Weak references
  - Weak Reference Objects
  - Example
  - Finalizer Objects
  - Comparing finalizers with `__del__()` methods
- `types` — Dynamic type creation and names for built-in types
  - Dynamic Type Creation
  - Standard Interpreter Types
  - Additional Utility Classes and Functions
  - Coroutine Utility Functions
- `copy` — Shallow and deep copy operations
- `pprint` — Data pretty printer
  - Functions
  - PrettyPrinter Objects
  - Example
- `reprlib` — Alternate `repr()` implementation
  - Repr Objects
  - Subclassing Repr Objects
- `enum` — Support for enumerations
  - Module Contents
  - Data Types
    - Supported `__dunder__` names
    - Supported `_sunder_` names
  - Utilities and Decorators
  - Notes
- `graphlib` — Functionality to operate with graph-like structures

    - Exceptions

`datetime` — Basic date and time types

Source code: Lib/datetime.py [https://github.com/python/cpython/tree/3.14/Lib/datetime.py]

The **`datetime`** module supplies classes for manipulating dates and times.

While date and time arithmetic is supported, the focus of the implementation is on efficient attribute extraction for output formatting and manipulation.

Tip

Skip to the format codes.

See also

Module `calendar`
General calendar related functions.

Module `time`
Time access and conversions.

Module `zoneinfo`
Concrete time zones representing the IANA time zone database.

Package dateutil [https://dateutil.readthedocs.io/en/stable/]

Third-party library with expanded time zone and parsing support.

Package DateType [https://pypi.org/project/DateType/]
Third-party library that introduces distinct static types to e.g. allow static type checkers to differentiate between naive and aware datetimes.

Aware and Naive Objects

Date and time objects may be categorized as “aware” or “naive” depending on whether or not they include time zone information.

With sufficient knowledge of applicable algorithmic and political time adjustments, such as time zone and daylight saving time information, an **aware** object can locate itself relative to other aware objects. An aware object represents a specific moment in time that is not open to interpretation. [[1]]

A **naive** object does not contain enough information to unambiguously locate itself relative to other date/time objects. Whether a naive object represents Coordinated Universal Time (UTC), local time, or time in some other time zone is purely up to the program, just like it is up to the program whether a particular number represents metres, miles, or mass. Naive objects are easy to understand and to work with, at the cost of ignoring some aspects of reality.

For applications requiring aware objects, `datetime` and `time` objects have an optional time zone information attribute, `tzinfo`, that can be set to an instance of a subclass of the abstract `tzinfo` class. These `tzinfo` objects capture information about the offset from UTC time, the time zone name, and whether daylight saving time is in effect.

Only one concrete `tzinfo` class, the `timezone` class, is supplied by the **`datetime`** module. The **`timezone`** class can represent simple time zones with fixed offsets from UTC, such as UTC itself or North American EST and EDT time zones. Supporting time zones at deeper levels of detail is up to the application. The rules for time adjustment across the world are more political than rational, change frequently, and there is no standard suitable for every application aside from UTC.

Constants

The **`datetime`** module exports the following constants:

datetime.MINYEAR

The smallest year number allowed in a `date` or `datetime` object. `MINYEAR` is 1.

datetime.MAXYEAR

The largest year number allowed in a `date` or `datetime` object. `MAXYEAR` is 9999.

datetime.UTC

Alias for the UTC time zone singleton `datetime.timezone.utc`.

Added in version 3.11.

Available Types

class datetime.date

An idealized naive date, assuming the current Gregorian calendar always was, and always will be, in effect. Attributes: `year`, `month`, and `day`.

class datetime.time

An idealized time, independent of any particular day, assuming that every day has exactly 24*60*60 seconds. (There is no notion of “leap seconds” here.) Attributes: `hour`, `minute`, `second`, `microsecond`, and `tzinfo`.

class datetime.datetime

A combination of a date and a time. Attributes: `year`, `month`, `day`, `hour`, `minute`, `second`, `microsecond`, and `tzinfo`.

class datetime.timedelta

A duration expressing the difference between two `datetime` or `date` instances to microsecond resolution.

class datetime.tzinfo

An abstract base class for time zone information objects. These are used by the `datetime` and `time` classes to provide a customizable notion of time adjustment (for example, to account for time zone and/or daylight saving time).

class datetime.timezone

A class that implements the `tzinfo` abstract base class as a fixed offset from the UTC.

Added in version 3.2.

Objects of these types are immutable.

Subclass relationships:

```
object
    timedelta
```

```
tzinfo
    timezone
time
date
    datetime
```

Common Properties

The **`date`**, **`datetime`**, **`time`**, and **`timezone`** types share these common features:

- Objects of these types are immutable.
- Objects of these types are hashable, meaning that they can be used as dictionary keys.
- Objects of these types support efficient pickling via the **`pickle`** module.

Determining if an Object is Aware or Naive

Objects of the **`date`** type are always naive.

An object of type **`time`** or **`datetime`** may be aware or naive.

A **`datetime`** object `d` is aware if both of the following hold:

1. `d.tzinfo` is not `None`
2. `d.tzinfo.utcoffset(d)` does not return `None`

Otherwise, `d` is naive.

A **`time`** object `t` is aware if both of the following hold:

1. `t.tzinfo` is not `None`
2. `t.tzinfo.utcoffset(None)` does not return `None`.

Otherwise, `t` is naive.

The distinction between aware and naive doesn’t apply to **`timedelta`** objects.

`timedelta` Objects

A **`timedelta`** object represents a duration, the difference between two **`datetime`** or **`date`** instances.

class datetime.timedelta(*days=0*, *seconds=0*, *microseconds=0*, *milliseconds=0*, *minutes=0*, *hours=0*, *weeks=0*)

> All arguments are optional and default to 0. Arguments may be integers or floats, and may be positive or negative.
>
> Only *days*, *seconds* and *microseconds* are stored internally. Arguments are converted to those units:
>
> - A millisecond is converted to 1000 microseconds.
> - A minute is converted to 60 seconds.
> - An hour is converted to 3600 seconds.
> - A week is converted to 7 days.
>
> and days, seconds and microseconds are then normalized so that the representation is unique, with
>
> - `0 <= microseconds < 1000000`
> - `0 <= seconds < 3600*24` (the number of seconds in one day)
> - `-999999999 <= days <= 999999999`
>
> The following example illustrates how any arguments besides *days*, *seconds* and *microseconds* are “merged” and normalized into those three resulting attributes:

```
>>> from datetime import timedelta
>>> delta = timedelta(
...     days=50,
...     seconds=27,
...     microseconds=10,
...     milliseconds=29000,
...     minutes=5,
...     hours=8,
...     weeks=2
... )
>>> # Only days, seconds, and microseconds remain
>>> delta
datetime.timedelta(days=64, seconds=29156, microseconds=10)
```

If any argument is a float and there are fractional microseconds, the fractional microseconds left over from all arguments are combined and their sum is rounded to the nearest microsecond using round-half-to-even tiebreaker. If no argument is a float, the conversion and normalization processes are exact (no information is lost).

If the normalized value of days lies outside the indicated range, `OverflowError` is raised.

Note that normalization of negative values may be surprising at first. For example:

```
>>> from datetime import timedelta
>>> d = timedelta(microseconds=-1)
>>> (d.days, d.seconds, d.microseconds)
(-1, 86399, 999999)
```

Since the string representation of **`timedelta`** objects can be confusing, use the following recipe to produce a more readable format:

```
>>> def pretty_timedelta(td):
...     if td.days >= 0:
...         return str(td)
...     return f'-({-td!s})'
...
>>> d = timedelta(hours=-1)
>>> str(d)  # not human-friendly
'-1 day, 23:00:00'
>>> pretty_timedelta(d)
'-(1:00:00)'
```

Class attributes:

timedelta.min

The most negative **`timedelta`** object, `timedelta(-999999999)`.

timedelta.max

The most positive **`timedelta`** object, `timedelta(days=999999999, hours=23, minutes=59, seconds=59, microseconds=999999)`.

timedelta.resolution

The smallest possible difference between non-equal **`timedelta`** objects, `timedelta(microseconds=1)`.

Note that, because of normalization, `timedelta.max` is greater than `-timedelta.min`. `-timedelta.max` is not representable as a **`timedelta`** object.

Instance attributes (read-only):

timedelta.days

Between -999,999,999 and 999,999,999 inclusive.

timedelta.seconds

Between 0 and 86,399 inclusive.

Caution

It is a somewhat common bug for code to unintentionally use this attribute when it is actually intended to get a **`total_seconds()`** value instead:

```
>>> from datetime import timedelta
>>> duration = timedelta(seconds=11235813)
>>> duration.days, duration.seconds
(130, 3813)
>>> duration.total_seconds()
11235813.0
```

timedelta.microseconds

Between 0 and 999,999 inclusive.

Supported operations:

Operation	Result
`t1 = t2 + t3`	Sum of `t2` and `t3`. Afterwards `t1 - t2 == t3` and `t1 - t3 == t2` are true. (1)
`t1 = t2 - t3`	Difference of `t2` and `t3`. Afterwards `t1 == t2 - t3` and `t2 == t1 + t3` are true. (1)(6)
`t1 = t2 * i or t1 = i * t2`	Delta multiplied by an integer. Afterwards `t1 // i == t2` is true, provided `i != 0`.
	In general, `t1  * i == t1 * (i-1) + t1` is true. (1)
`t1 = t2 * f or t1 = f * t2`	Delta multiplied by a float. The result is rounded to the nearest multiple of timedelta.resolution using round-half-to-even.

Operation	**Result**
`f = t2 / t3`	Division (3) of overall duration `t2` by interval unit `t3`. Returns a **`float`** object.
`t1 = t2 / f` or `t1 = t2 / i`	Delta divided by a float or an int. The result is rounded to the nearest multiple of timedelta.resolution using round-half-to-even.
`t1 = t2 // i` or `t1 = t2 // t3`	The floor is computed and the remainder (if any) is thrown away. In the second case, an integer is returned. (3)
`t1 = t2 % t3`	The remainder is computed as a **`timedelta`** object. (3)
`q, r = divmod(t1, t2)`	Computes the quotient and the remainder: `q = t1 // t2` (3) and `r = t1 % t2`. `q` is an integer and `r` is a **`timedelta`** object.
`+t1`	Returns a **`timedelta`** object with the same value. (2)
`-t1`	Equivalent to `timedelta(-t1.days, -t1.seconds, -t1.microseconds)`, and to `t1 * -1`. (1)(4)

Operation	Result
`abs(t)`	Equivalent to `+t` when `t.days >= 0`, and to `-t` when `t.days < 0`. (2)
`str(t)`	Returns a string in the form `[D day[s], ][H]H:MM:SS[.UUUUUU]`, where D is negative for negative `t`. (5)
`repr(t)`	Returns a string representation of the **`timedelta`** object as a constructor call with canonical attribute values.

Notes:

1. This is exact but may overflow.
2. This is exact and cannot overflow.
3. Division by zero raises **`ZeroDivisionError`**.
4. `-timedelta.max` is not representable as a **`timedelta`** object.
5. String representations of **`timedelta`** objects are normalized similarly to their internal representation. This leads to somewhat unusual results for negative timedeltas. For example:

```
>>> timedelta(hours=-5)
datetime.timedelta(days=-1, seconds=68400)
>>> print(_)
-1 day, 19:00:00
```

6. The expression `t2 - t3` will always be equal to the expression `t2 + (-t3)` except when t3 is equal to `timedelta.max`; in that case the former will produce a result while the latter will overflow.

In addition to the operations listed above, **`timedelta`** objects support certain additions and subtractions with **`date`** and **`datetime`** objects (see below).

Changed in version 3.2: Floor division and true division of a **`timedelta`** object by another **`timedelta`** object are now supported, as are remainder operations and the **`divmod()`** function. True division and multiplication of a **`timedelta`** object by a **`float`** object are now supported.

`timedelta` objects support equality and order comparisons.

In Boolean contexts, a **`timedelta`** object is considered to be true if and only if it isn't equal to `timedelta(0)`.

Instance methods:

timedelta.total_seconds()

Return the total number of seconds contained in the duration. Equivalent to `td / timedelta(seconds=1)`. For interval units other than seconds, use the division form directly (e.g. `td / timedelta(microseconds=1)`).

Note that for very large time intervals (greater than 270 years on most platforms) this method will lose microsecond accuracy.

Added in version 3.2.

Examples of usage: `timedelta`

An additional example of normalization:

```
>>> # Components of another_year add up to exactly 365 days
>>> from datetime import timedelta
>>> year = timedelta(days=365)
>>> another_year = timedelta(weeks=40, days=84, hours=23,
...                          minutes=50, seconds=600)
>>> year == another_year
True
>>> year.total_seconds()
31536000.0
```

Examples of **`timedelta`** arithmetic:

```
>>> from datetime import timedelta
>>> year = timedelta(days=365)
>>> ten_years = 10 * year
>>> ten_years
datetime.timedelta(days=3650)
>>> ten_years.days // 365
10
>>> nine_years = ten_years - year
>>> nine_years
datetime.timedelta(days=3285)
>>> three_years = nine_years // 3
>>> three_years, three_years.days // 365
(datetime.timedelta(days=1095), 3)
```

`date` Objects

A **`date`** object represents a date (year, month and day) in an idealized calendar, the current Gregorian calendar indefinitely extended in both directions.

January 1 of year 1 is called day number 1, January 2 of year 1 is called day number 2, and so on. [[2]]

class datetime.date(*year*, *month*, *day*)

All arguments are required. Arguments must be integers, in the following ranges:

- `MINYEAR <= year <= MAXYEAR`
- `1 <= month <= 12`
- `1 <= day <= number of days in the given month and year`

If an argument outside those ranges is given, **`ValueError`** is raised.

Other constructors, all class methods:

classmethod date.today()

Return the current local date.

This is equivalent to `date.fromtimestamp(time.time())`.

classmethod date.fromtimestamp(*timestamp*)

Return the local date corresponding to the POSIX timestamp, such as is returned by **`time.time()`**.

This may raise **`OverflowError`**, if the timestamp is out of the range of values supported by the platform C **`localtime()`** function, and **`OSError`** on **`localtime()`** failure. It’s common for this to be restricted to years from 1970 through 2038. Note that on non-POSIX systems that include leap seconds in their notion of a timestamp, leap seconds are ignored by **`fromtimestamp()`**.

Changed in version 3.3: Raise **`OverflowError`** instead of **`ValueError`** if the timestamp is out of the range of values supported by the platform C **`localtime()`** function. Raise **`OSError`** instead of **`ValueError`** on **`localtime()`** failure.

classmethod date.fromordinal(*ordinal*)

Return the date corresponding to the proleptic Gregorian ordinal, where January 1 of year 1 has ordinal 1.

`ValueError` is raised unless `1 <= ordinal <= date.max.toordinal()`. For any date `d`, `date.fromordinal(d.toordinal()) == d`.

classmethod date.fromisoformat(*date_string*)

Return a **`date`** corresponding to a *date_string* given in any valid ISO 8601 format, with the following exceptions:

1. Reduced precision dates are not currently supported (`YYYY-MM`, `YYYY`).
2. Extended date representations are not currently supported (`±YYYYYY-MM-DD`).
3. Ordinal dates are not currently supported (`YYYY-OOO`).

Examples:

```
>>> from datetime import date
>>> date.fromisoformat('2019-12-04')
datetime.date(2019, 12, 4)
>>> date.fromisoformat('20191204')
datetime.date(2019, 12, 4)
>>> date.fromisoformat('2021-W01-1')
datetime.date(2021, 1, 4)
```

Added in version 3.7.

Changed in version 3.11: Previously, this method only supported the format `YYYY-MM-DD`.

classmethod date.fromisocalendar(*year*, *week*, *day*)

Return a **`date`** corresponding to the ISO calendar date specified by year, week and day. This is the inverse of the function **`date.isocalendar()`**.

Added in version 3.8.

classmethod date.strptime(*date_string*, *format*)

Return a **`date`** corresponding to *date_string*, parsed according to *format*. This is equivalent to:

```
date(*(time.strptime(date_string, format)[0:3]))
```

`ValueError` is raised if the date_string and format can't be parsed by **`time.strptime()`** or if it returns a value which isn't a time tuple. See also strftime() and strptime() Behavior and **`date.fromisoformat()`**.

Note

If *format* specifies a day of month without a year a **`DeprecationWarning`** is emitted. This is to avoid a quadrennial leap year bug in code seeking to parse only a month and day as the default year used in absence of one in the format is not a leap year. Such *format* values may raise an error as of Python 3.15. The workaround is to always include a year in your *format*. If parsing *date_string* values that do not have a year, explicitly add a year that is a leap year before parsing:

```
>>> from datetime import date
>>> date_string = "02/29"
>>> when = date.strptime(f"{date_string};1984",
"%m/%d;%Y")  # Avoids leap year bug.
>>> when.strftime("%B %d")
'February 29'
```

Added in version 3.14.

Class attributes:

date.min

The earliest representable date, `date(MINYEAR, 1, 1)`.

date.max

The latest representable date, `date(MAXYEAR, 12, 31)`.

date.resolution

The smallest possible difference between non-equal date objects, `timedelta(days=1)`.

Instance attributes (read-only):

date.year
  Between `MINYEAR` and `MAXYEAR` inclusive.

date.month
  Between 1 and 12 inclusive.

date.day
  Between 1 and the number of days in the given month of the given year.

Supported operations:

Operation	**Result**
`date2 = date1 + timedelta`	`date2` will be `timedelta.days` days after `date1`. (1)
`date2 = date1 - timedelta`	Computes `date2` such that `date2 + timedelta == date1`. (2)
`timedelta = date1 - date2`	(3)
`date1 == date2` `date1 != date2`	Equality comparison. (4)

`date1 < date2`
`date1 > date2`
`date1 <= date2`
	Order comparison. (5)

Operation	Result
`date1 >= date2`	

Notes:

1. *date2* is moved forward in time if `timedelta.days > 0`, or backward if `timedelta.days < 0`. Afterward `date2 - date1 == timedelta.days`. `timedelta.seconds` and `timedelta.microseconds` are ignored. **`OverflowError`** is raised if `date2.year` would be smaller than **`MINYEAR`** or larger than **`MAXYEAR`**.

2. `timedelta.seconds` and `timedelta.microseconds` are ignored.

3. This is exact, and cannot overflow. `timedelta.seconds` and `timedelta.microseconds` are 0, and `date2 + timedelta == date1` after.

4. **`date`** objects are equal if they represent the same date.

   `date` objects that are not also **`datetime`** instances are never equal to **`datetime`** objects, even if they represent the same date.

5. *date1* is considered less than *date2* when *date1* precedes *date2* in time. In other words, `date1 < date2` if and only if `date1.toordinal() < date2.toordinal()`.

   Order comparison between a **`date`** object that is not also a **`datetime`** instance and a **`datetime`** object raises **`TypeError`**.

Changed in version 3.13: Comparison between **`datetime`** object and an instance of the **`date`** subclass that is not a **`datetime`** subclass no longer converts the latter to **`date`**, ignoring the time part and the time zone. The default behavior can be changed by overriding the special comparison methods in subclasses.

In Boolean contexts, all `date` objects are considered to be true.

Instance methods:

date.replace(*year=self.year*, *month=self.month*, *day=self.day*)

Return a new `date` object with the same values, but with specified parameters updated.

Example:

```
>>> from datetime import date
>>> d = date(2002, 12, 31)
>>> d.replace(day=26)
datetime.date(2002, 12, 26)
```

The generic function `copy.replace()` also supports `date` objects.

date.timetuple()

Return a `time.struct_time` such as returned by `time.localtime()`.

The hours, minutes and seconds are 0, and the DST flag is -1.

`d.timetuple()` is equivalent to:

```
time.struct_time((d.year, d.month, d.day, 0, 0, 0,
d.weekday(), yday, -1))
```

where `yday = d.toordinal() - date(d.year, 1, 1).toordinal() + 1` is the day number within the current year starting with 1 for January 1st.

date.toordinal()

Return the proleptic Gregorian ordinal of the date, where January 1 of year 1 has ordinal 1. For any **`date`** object `d`, `date.fromordinal(d.toordinal()) == d`.

date.weekday()

Return the day of the week as an integer, where Monday is 0 and Sunday is 6. For example, `date(2002, 12, 4).weekday() == 2`, a Wednesday. See also **`isoweekday()`**.

date.isoweekday()

Return the day of the week as an integer, where Monday is 1 and Sunday is 7. For example, `date(2002, 12, 4).isoweekday() == 3`, a Wednesday. See also **`weekday()`**, **`isocalendar()`**.

date.isocalendar()

Return a named tuple object with three components: `year`, `week` and `weekday`.

The ISO calendar is a widely used variant of the Gregorian calendar. [[3]]

The ISO year consists of 52 or 53 full weeks, and where a week starts on a Monday and ends on a Sunday. The first week of an ISO year is the first (Gregorian) calendar week of a year containing a Thursday. This is called week number 1, and the ISO year of that Thursday is the same as its Gregorian year.

For example, 2004 begins on a Thursday, so the first week of ISO year 2004 begins on Monday, 29 Dec 2003 and ends on Sunday, 4 Jan 2004:

```
>>> from datetime import date
>>> date(2003, 12, 29).isocalendar()
datetime.IsoCalendarDate(year=2004, week=1, weekday=1)
>>> date(2004, 1, 4).isocalendar()
datetime.IsoCalendarDate(year=2004, week=1, weekday=7)
```

Changed in version 3.9: Result changed from a tuple to a named tuple.

date.isoformat()

Return a string representing the date in ISO 8601 format, `YYYY-MM-DD`:

```
>>> from datetime import date
>>> date(2002, 12, 4).isoformat()
'2002-12-04'
```

date.__str__()

For a date `d`, `str(d)` is equivalent to `d.isoformat()`.

date.ctime()

Return a string representing the date:

```
>>> from datetime import date
>>> date(2002, 12, 4).ctime()
'Wed Dec  4 00:00:00 2002'
```

`d.ctime()` is equivalent to:

```
time.ctime(time.mktime(d.timetuple()))
```

on platforms where the native C **`ctime()`** function (which **`time.ctime()`** invokes, but which **`date.ctime()`** does not invoke) conforms to the C standard.

date.strftime(*format*)

Return a string representing the date, controlled by an explicit format string. Format codes referring to hours, minutes or seconds will see 0 values. See also strftime() and strptime() Behavior and **`date.isoformat()`**.

date.__format__(*format*)

Same as **`date.strftime()`**. This makes it possible to specify a format string for a **`date`** object in formatted string literals and when using **`str.format()`**. See also strftime() and strptime() Behavior and **`date.isoformat()`**.

Examples of Usage: `date`

Example of counting days to an event:

```
>>> import time
>>> from datetime import date
>>> today = date.today()
>>> today
datetime.date(2007, 12, 5)
>>> today == date.fromtimestamp(time.time())
True
>>> my_birthday = date(today.year, 6, 24)
>>> if my_birthday < today:
...     my_birthday = my_birthday.replace(year=today.year + 1)
...
>>> my_birthday
datetime.date(2008, 6, 24)
>>> time_to_birthday = abs(my_birthday - today)
>>> time_to_birthday.days
202
```

More examples of working with **`date`**:

```
>>> from datetime import date
>>> d = date.fromordinal(730920) # 730920th day after 1. 1.
0001
>>> d
datetime.date(2002, 3, 11)

>>> # Methods related to formatting string output
>>> d.isoformat()
```

```
'2002-03-11'
>>> d.strftime("%d/%m/%y")
'11/03/02'
>>> d.strftime("%A %d. %B %Y")
'Monday 11. March 2002'
>>> d.ctime()
'Mon Mar 11 00:00:00 2002'
>>> 'The {1} is {0:%d}, the {2} is {0:%B}.'.format(d, "day",
"month")
'The day is 11, the month is March.'

>>> # Methods for to extracting 'components' under different
calendars
>>> t = d.timetuple()
>>> for i in t:
...     print(i)
2002                # year
3                   # month
11                  # day
0
0
0
0                   # weekday (0 = Monday)
70                  # 70th day in the year
-1
>>> ic = d.isocalendar()
>>> for i in ic:
...     print(i)
2002                # ISO year
11                  # ISO week number
1                   # ISO day number ( 1 = Monday )

>>> # A date object is immutable; all operations produce a new
object
>>> d.replace(year=2005)
datetime.date(2005, 3, 11)
```

`datetime` Objects

A `datetime` object is a single object containing all the information from a `date` object and a `time` object.

Like a `date` object, `datetime` assumes the current Gregorian calendar extended in both directions; like a `time` object, `datetime` assumes there are exactly 3600*24 seconds in every day.

Constructor:

class datetime.datetime(*year*, *month*, *day*, *hour=0*, *minute=0*, *second=0*, *microsecond=0*, *tzinfo=None*, *, *fold=0*)

The *year*, *month* and *day* arguments are required. *tzinfo* may be `None`, or an instance of a `tzinfo` subclass. The remaining arguments must be integers in the following ranges:

- `MINYEAR <= year <= MAXYEAR,`
- `1 <= month <= 12,`
- `1 <= day <= number of days in the given month and year,`
- `0 <= hour < 24,`
- `0 <= minute < 60,`
- `0 <= second < 60,`
- `0 <= microsecond < 1000000,`
- `fold in [0, 1].`

If an argument outside those ranges is given, `ValueError` is raised.

Changed in version 3.6: Added the *fold* parameter.

Other constructors, all class methods:

classmethod datetime.today()

Return the current local date and time, with `tzinfo` `None`.

Equivalent to:

```
datetime.fromtimestamp(time.time())
```

See also `now()`, `fromtimestamp()`.

This method is functionally equivalent to `now()`, but without a `tz` parameter.

classmethod datetime.now(*tz=None*)

Return the current local date and time.

If optional argument *tz* is `None` or not specified, this is like `today()`, but, if possible, supplies more precision than can be gotten from going through a `time.time()` timestamp (for example, this may be possible on platforms supplying the C **`gettimeofday()`** function).

If *tz* is not `None`, it must be an instance of a `tzinfo` subclass, and the current date and time are converted to *tz*’s time zone.

This function is preferred over `today()` and `utcnow()`.

Note

Subsequent calls to **`datetime.now()`** may return the same instant depending on the precision of the underlying clock.

classmethod datetime.utcnow()

Return the current UTC date and time, with `tzinfo` `None`.

This is like `now()`, but returns the current UTC date and time, as a naive `datetime` object. An aware current UTC datetime can be obtained by calling `datetime.now(timezone.utc)`. See also `now()`.

Warning

> Because naive `datetime` objects are treated by many `datetime` methods as local times, it is preferred to use aware datetimes to represent times in UTC. As such, the recommended way to create an object representing the current time in UTC is by calling `datetime.now(timezone.utc)`.

Deprecated since version 3.12: Use **`datetime.now()`** with **`UTC`** instead.

classmethod datetime.fromtimestamp(*timestamp*, *tz=None*)

Return the local date and time corresponding to the POSIX timestamp, such as is returned by **`time.time()`**. If optional argument *tz* is `None` or not specified, the timestamp is converted to the platform's local date and time, and the returned **`datetime`** object is naive.

If *tz* is not `None`, it must be an instance of a **`tzinfo`** subclass, and the timestamp is converted to *tz*'s time zone.

`fromtimestamp()` may raise **`OverflowError`**, if the timestamp is out of the range of values supported by the platform C **`localtime()`** or **`gmtime()`** functions, and **`OSError`** on **`localtime()`** or **`gmtime()`** failure. It's common for this to be restricted to years in 1970 through 2038. Note that on non-POSIX systems that include leap seconds in their notion of a timestamp, leap seconds are ignored by **`fromtimestamp()`**, and then it's possible to have two timestamps differing by a second that yield identical **`datetime`** objects. This method is preferred over **`utcfromtimestamp()`**.

Changed in version 3.3: Raise **`OverflowError`** instead of **`ValueError`** if the timestamp is out of the range of values

supported by the platform C **`localtime()`** or **`gmtime()`** functions. Raise `OSError` instead of `ValueError` on **`localtime()`** or **`gmtime()`** failure.
Changed in version 3.6: `fromtimestamp()` may return instances with `fold` set to 1.

classmethod datetime.utcfromtimestamp(*timestamp*)

Return the UTC `datetime` corresponding to the POSIX timestamp, with `tzinfo` `None`. (The resulting object is naive.)

This may raise `OverflowError`, if the timestamp is out of the range of values supported by the platform C **`gmtime()`** function, and `OSError` on **`gmtime()`** failure. It’s common for this to be restricted to years in 1970 through 2038.

To get an aware `datetime` object, call `fromtimestamp()`:

```
datetime.fromtimestamp(timestamp, timezone.utc)
```

On the POSIX compliant platforms, it is equivalent to the following expression:

```
datetime(1970, 1, 1, tzinfo=timezone.utc) +
timedelta(seconds=timestamp)
```

except the latter formula always supports the full years range: between `MINYEAR` and `MAXYEAR` inclusive.

Warning

Because naive `datetime` objects are treated by many `datetime` methods as local times, it is preferred to use aware datetimes to represent times in UTC. As such, the recommended way to create an object representing a specific timestamp in UTC is by calling `datetime.fromtimestamp(timestamp, tz=timezone.utc)`.

Changed in version 3.3: Raise `OverflowError` instead of `ValueError` if the timestamp is out of the range of values supported by the platform C **`gmtime()`** function. Raise `OSError` instead of `ValueError` on **`gmtime()`** failure.

Deprecated since version 3.12: Use `datetime.fromtimestamp()` with `UTC` instead.

classmethod datetime.fromordinal(*ordinal*)

Return the `datetime` corresponding to the proleptic Gregorian ordinal, where January 1 of year 1 has ordinal 1. `ValueError` is raised unless `1 <= ordinal <= datetime.max.toordinal()`. The hour, minute, second and microsecond of the result are all 0, and `tzinfo` is `None`.

classmethod datetime.combine(*date*, *time*, *tzinfo=time.tzinfo*)

Return a new `datetime` object whose date components are equal to the given `date` object's, and whose time components are equal to the given `time` object's. If the *tzinfo* argument is provided, its value is used to set the `tzinfo` attribute of the result, otherwise the `tzinfo` attribute of the *time* argument is used. If the *date* argument is a `datetime` object, its time components and `tzinfo` attributes are ignored.

For any `datetime` object `d`, `d == datetime.combine(d.date(), d.time(), d.tzinfo)`.

Changed in version 3.6: Added the *tzinfo* argument.

classmethod datetime.fromisoformat(*date_string*)

Return a `datetime` corresponding to a *date_string* in any valid ISO 8601 format, with the following exceptions:

1. Time zone offsets may have fractional seconds.

2. The `T` separator may be replaced by any single unicode character.
3. Fractional hours and minutes are not supported.
4. Reduced precision dates are not currently supported (`YYYY-MM`, `YYYY`).
5. Extended date representations are not currently supported (`±YYYYYY-MM-DD`).
6. Ordinal dates are not currently supported (`YYYY-OOO`).

Examples:

```
>>> from datetime import datetime
>>> datetime.fromisoformat('2011-11-04')
datetime.datetime(2011, 11, 4, 0, 0)
>>> datetime.fromisoformat('20111104')
datetime.datetime(2011, 11, 4, 0, 0)
>>> datetime.fromisoformat('2011-11-04T00:05:23')
datetime.datetime(2011, 11, 4, 0, 5, 23)
>>> datetime.fromisoformat('2011-11-04T00:05:23Z')
datetime.datetime(2011, 11, 4, 0, 5, 23,
tzinfo=datetime.timezone.utc)
>>> datetime.fromisoformat('20111104T000523')
datetime.datetime(2011, 11, 4, 0, 5, 23)
>>> datetime.fromisoformat('2011-W01-2T00:05:23.283')
datetime.datetime(2011, 1, 4, 0, 5, 23, 283000)
>>> datetime.fromisoformat('2011-11-04 00:05:23.283')
datetime.datetime(2011, 11, 4, 0, 5, 23, 283000)
>>> datetime.fromisoformat('2011-11-04 00:05:23.283+00:00')
datetime.datetime(2011, 11, 4, 0, 5, 23, 283000,
tzinfo=datetime.timezone.utc)
>>> datetime.fromisoformat('2011-11-04T00:05:23+04:00')
datetime.datetime(2011, 11, 4, 0, 5, 23,

tzinfo=datetime.timezone(datetime.timedelta(seconds=14400)))
```

Added in version 3.7.

Changed in version 3.11: Previously, this method only supported formats that could be emitted by `date.isoformat()` or `datetime.isoformat()`.

classmethod datetime.fromisocalendar(*year*, *week*, *day*)

Return a `datetime` corresponding to the ISO calendar date specified by year, week and day. The non-date components of the datetime are populated with their normal default values. This is the inverse of the function `datetime.isocalendar()`.

Added in version 3.8.

classmethod datetime.strptime(*date_string*, *format*)

Return a `datetime` corresponding to *date_string*, parsed according to *format*.

If *format* does not contain microseconds or time zone information, this is equivalent to:

```
datetime(*(time.strptime(date_string, format)[0:6]))
```

`ValueError` is raised if the date_string and format can't be parsed by `time.strptime()` or if it returns a value which isn't a time tuple. See also strftime() and strptime() Behavior and `datetime.fromisoformat()`.

Changed in version 3.13: If *format* specifies a day of month without a year a `DeprecationWarning` is now emitted. This is to avoid a quadrennial leap year bug in code seeking to parse only a month and day as the default year used in absence of one in the format is not a leap year. Such *format* values may raise an error as of Python 3.15. The workaround is to always include a year in your *format*. If parsing *date_string* values that do not have a year, explicitly add a year that is a leap year before parsing:

```
>>> from datetime import datetime
>>> date_string = "02/29"
>>> when = datetime.strptime(f"{date_string};1984",
"%m/%d;%Y")  # Avoids leap year bug.
>>> when.strftime("%B %d")
'February 29'
```

Class attributes:

datetime.min

The earliest representable **`datetime`**, `datetime(MINYEAR, 1, 1, tzinfo=None)`.

datetime.max

The latest representable **`datetime`**, `datetime(MAXYEAR, 12, 31, 23, 59, 59, 999999, tzinfo=None)`.

datetime.resolution

The smallest possible difference between non-equal **`datetime`** objects, `timedelta(microseconds=1)`.

Instance attributes (read-only):

datetime.year

Between **`MINYEAR`** and **`MAXYEAR`** inclusive.

datetime.month

Between 1 and 12 inclusive.

datetime.day

Between 1 and the number of days in the given month of the given year.

datetime.hour

In `range(24)`.

datetime.minute

In `range(60)`.

datetime.second

In `range(60)`.

datetime.microsecond

In `range(1000000)`.

datetime.tzinfo

The object passed as the *tzinfo* argument to the **`datetime`** constructor, or `None` if none was passed.

datetime.fold

In `[0, 1]`. Used to disambiguate wall times during a repeated interval. (A repeated interval occurs when clocks are rolled back at the end of daylight saving time or when the UTC offset for the current zone is decreased for political reasons.) The values 0 and 1 represent, respectively, the earlier and later of the two moments with the same wall time representation.

Added in version 3.6.

Supported operations:

Operation	**Result**
`datetime2 = datetime1 + timedelta`	(1)
`datetime2 = datetime1 - timedelta`	(2)
`timedelta = datetime1 - datetime2`	(3)
`datetime1 == datetime2` `datetime1 != datetime2`	Equality comparison. (4)

Operation	Result

`datetime1 < datetime2`
`datetime1 > datetime2`
`datetime1 <= datetime2`
`datetime1 >= datetime2`
	Order comparison. (5)

1. `datetime2` is a duration of `timedelta` removed from `datetime1`, moving forward in time if `timedelta.days > 0`, or backward if `timedelta.days < 0`. The result has the same **`tzinfo`** attribute as the input datetime, and `datetime2 - datetime1 == timedelta` after. **`OverflowError`** is raised if `datetime2.year` would be smaller than **`MINYEAR`** or larger than **`MAXYEAR`**. Note that no time zone adjustments are done even if the input is an aware object.

2. Computes the `datetime2` such that `datetime2 + timedelta == datetime1`. As for addition, the result has the same **`tzinfo`** attribute as the input datetime, and no time zone adjustments are done even if the input is aware.

3. Subtraction of a **`datetime`** from a **`datetime`** is defined only if both operands are naive, or if both are aware. If one is aware and the other is naive, **`TypeError`** is raised.

   If both are naive, or both are aware and have the same **`tzinfo`** attribute, the **`tzinfo`** attributes are ignored, and the result is a **`timedelta`** object `t` such that `datetime2 + t == datetime1`. No time zone adjustments are done in this case.

   If both are aware and have different **`tzinfo`** attributes, `a-b` acts as if `a` and `b` were first converted to naive UTC datetimes. The result is `(a.replace(tzinfo=None) - a.utcoffset()) - (b.replace(tzinfo=None) - b.utcoffset())` except that the implementation never overflows.

4. **`datetime`** objects are equal if they represent the same date and time, taking into account the time zone.

   Naive and aware **`datetime`** objects are never equal.

   If both comparands are aware, and have the same **`tzinfo`** attribute, the **`tzinfo`** and **`fold`** attributes are ignored and the base datetimes are compared. If both comparands are aware and have different **`tzinfo`** attributes, the comparison acts as comparands were first converted to UTC datetimes except that the implementation never overflows. **`datetime`** instances in a repeated interval are never equal to **`datetime`** instances in other time zone.

5. *datetime1* is considered less than *datetime2* when *datetime1* precedes *datetime2* in time, taking into account the time zone.

   Order comparison between naive and aware **`datetime`** objects raises **`TypeError`**.

   If both comparands are aware, and have the same **`tzinfo`** attribute, the **`tzinfo`** and **`fold`** attributes are ignored and the base datetimes are compared. If both comparands are aware and have different **`tzinfo`** attributes, the comparison acts as comparands were first converted to UTC datetimes except that the implementation never overflows.

Changed in version 3.3: Equality comparisons between aware and naive **`datetime`** instances don’t raise **`TypeError`**.

Changed in version 3.13: Comparison between `datetime` object and an instance of the `date` subclass that is not a **`datetime`** subclass no longer converts the latter to **`date`**, ignoring the time part and the time zone. The default behavior can be changed by overriding the special comparison methods in subclasses.

Instance methods:

datetime.date()

Return `date` object with same year, month and day.

datetime.time()

Return `time` object with same hour, minute, second, microsecond and fold. `tzinfo` is `None`. See also method `timetz()`.

Changed in version 3.6: The fold value is copied to the returned `time` object.

datetime.timetz()

Return `time` object with same hour, minute, second, microsecond, fold, and tzinfo attributes. See also method `time()`.

Changed in version 3.6: The fold value is copied to the returned `time` object.

datetime.replace(*year=self.year*, *month=self.month*, *day=self.day*, *hour=self.hour*, *minute=self.minute*, *second=self.second*, *microsecond=self.microsecond*, *tzinfo=self.tzinfo*, *, *fold=0*)

Return a new `datetime` object with the same attributes, but with specified parameters updated. Note that `tzinfo=None` can be specified to create a naive datetime from an aware datetime with no conversion of date and time data.

`datetime` objects are also supported by generic function **`copy.replace()`**.

Changed in version 3.6: Added the *fold* parameter.

datetime.astimezone(*tz=None*)

Return a **`datetime`** object with new **`tzinfo`** attribute *tz*, adjusting the date and time data so the result is the same UTC time as *self*, but in *tz*’s local time.

If provided, *tz* must be an instance of a **`tzinfo`** subclass, and its **`utcoffset()`** and **`dst()`** methods must not return `None`. If *self* is naive, it is presumed to represent time in the system time zone.

If called without arguments (or with `tz=None`) the system local time zone is assumed for the target time zone. The `.tzinfo` attribute of the converted datetime instance will be set to an instance of **`timezone`** with the zone name and offset obtained from the OS.

If `self.tzinfo` is *tz*, `self.astimezone(tz)` is equal to *self*: no adjustment of date or time data is performed. Else the result is local time in the time zone *tz*, representing the same UTC time as *self*: after `astz = dt.astimezone(tz)`, `astz - astz.utcoffset()` will have the same date and time data as `dt - dt.utcoffset()`.

If you merely want to attach a **`timezone`** object *tz* to a datetime *dt* without adjustment of date and time data, use `dt.replace(tzinfo=tz)`. If you merely want to remove the **`timezone`** object from an aware datetime *dt* without conversion of date and time data, use `dt.replace(tzinfo=None)`.

Note that the default **`tzinfo.fromutc()`** method can be overridden in a **`tzinfo`** subclass to affect the result returned by **`astimezone()`**. Ignoring error cases, **`astimezone()`** acts like:

```
def astimezone(self, tz):
    if self.tzinfo is tz:
        return self
    # Convert self to UTC, and attach the new timezone
object.
    utc = (self - self.utcoffset()).replace(tzinfo=tz)
    # Convert from UTC to tz's local time.
    return tz.fromutc(utc)
```

Changed in version 3.3: tz now can be omitted.

Changed in version 3.6: The **`astimezone()`** method can now be called on naive instances that are presumed to represent system local time.

datetime.utcoffset()

If **`tzinfo`** is `None`, returns `None`, else returns `self.tzinfo.utcoffset(self)`, and raises an exception if the latter doesn’t return `None` or a **`timedelta`** object with magnitude less than one day.

Changed in version 3.7: The UTC offset is not restricted to a whole number of minutes.

datetime.dst()

If **`tzinfo`** is `None`, returns `None`, else returns `self.tzinfo.dst(self)`, and raises an exception if the latter doesn’t return `None` or a **`timedelta`** object with magnitude less than one day.

Changed in version 3.7: The DST offset is not restricted to a whole number of minutes.

datetime.tzname()

If **`tzinfo`** is `None`, returns `None`, else returns `self.tzinfo.tzname(self)`, raises an exception if the latter doesn't return `None` or a string object,

datetime.timetuple()

Return a **`time.struct_time`** such as returned by **`time.localtime()`**.

`d.timetuple()` is equivalent to:

```
time.struct_time((d.year, d.month, d.day,
                  d.hour, d.minute, d.second,
                  d.weekday(), yday, dst))
```

where `yday = d.toordinal() - date(d.year, 1, 1).toordinal() + 1` is the day number within the current year starting with 1 for January 1st. The **`tm_isdst`** flag of the result is set according to the **`dst()`** method: **`tzinfo`** is `None` or **`dst()`** returns `None`, **`tm_isdst`** is set to `-1`; else if **`dst()`** returns a non-zero value, **`tm_isdst`** is set to 1; else **`tm_isdst`** is set to 0.

datetime.utctimetuple()

If **`datetime`** instance `d` is naive, this is the same as `d.timetuple()` except that **`tm_isdst`** is forced to 0 regardless of what `d.dst()` returns. DST is never in effect for a UTC time.

If `d` is aware, `d` is normalized to UTC time, by subtracting `d.utcoffset()`, and a **`time.struct_time`** for the normalized time is returned. **`tm_isdst`** is forced to 0. Note that an **`OverflowError`** may be raised if `d.year` was `MINYEAR` or `MAXYEAR` and UTC adjustment spills over a year boundary.

Warning

Because naive `datetime` objects are treated by many `datetime` methods as local times, it is preferred to use aware datetimes to represent times in UTC; as a result, using **`datetime.utctimetuple()`** may give misleading results. If you have a naive `datetime` representing UTC, use `datetime.replace(tzinfo=timezone.utc)` to make it aware, at which point you can use **`datetime.timetuple()`**.

datetime.toordinal()

Return the proleptic Gregorian ordinal of the date. The same as `self.date().toordinal()`.

datetime.timestamp()

Return POSIX timestamp corresponding to the **`datetime`** instance. The return value is a **`float`** similar to that returned by **`time.time()`**.

Naive **`datetime`** instances are assumed to represent local time and this method relies on the platform C **`mktime()`** function to perform the conversion. Since **`datetime`** supports wider range of values than **`mktime()`** on many platforms, this method may raise **`OverflowError`** or **`OSError`** for times far in the past or far in the future.

For aware **`datetime`** instances, the return value is computed as:

```
(dt - datetime(1970, 1, 1,
tzinfo=timezone.utc)).total_seconds()
```

Added in version 3.3.

Changed in version 3.6: The **`timestamp()`** method uses the **`fold`** attribute to disambiguate the times during a repeated interval.

Note

There is no method to obtain the POSIX timestamp directly from a naive `datetime` instance representing UTC time. If your application uses this convention and your system time zone is not set to UTC, you can obtain the POSIX timestamp by supplying `tzinfo=timezone.utc`:

```
timestamp = dt.replace(tzinfo=timezone.utc).timestamp()
```

or by calculating the timestamp directly:

```
timestamp = (dt - datetime(1970, 1, 1)) /
timedelta(seconds=1)
```

datetime.weekday()

Return the day of the week as an integer, where Monday is 0 and Sunday is 6. The same as `self.date().weekday()`. See also `isoweekday()`.

datetime.isoweekday()

Return the day of the week as an integer, where Monday is 1 and Sunday is 7. The same as `self.date().isoweekday()`. See also `weekday()`, `isocalendar()`.

datetime.isocalendar()

Return a named tuple with three components: `year`, `week` and `weekday`. The same as `self.date().isocalendar()`.

datetime.isoformat(*sep*='*T*', *timespec*='*auto*')

Return a string representing the date and time in ISO 8601 format:

- `YYYY-MM-DDTHH:MM:SS.ffffff`, if `microsecond` is not 0
- `YYYY-MM-DDTHH:MM:SS`, if `microsecond` is 0

If `utcoffset()` does not return `None`, a string is appended, giving the UTC offset:

- `YYYY-MM-DDTHH:MM:SS.ffffff+HH:MM[:SS[.ffffff]]`, if `microsecond` is not 0
- `YYYY-MM-DDTHH:MM:SS+HH:MM[:SS[.ffffff]]`, if `microsecond` is 0

Examples:

```
>>> from datetime import datetime, timezone
>>> datetime(2019, 5, 18, 15, 17, 8, 132263).isoformat()
'2019-05-18T15:17:08.132263'
>>> datetime(2019, 5, 18, 15, 17,
tzinfo=timezone.utc).isoformat()
'2019-05-18T15:17:00+00:00'
```

The optional argument *sep* (default `'T'`) is a one-character separator, placed between the date and time portions of the result. For example:

```
>>> from datetime import tzinfo, timedelta, datetime
>>> class TZ(tzinfo):
...     """A time zone with an arbitrary, constant -06:39
offset."""
...     def utcoffset(self, dt):
...         return timedelta(hours=-6, minutes=-39)
...
>>> datetime(2002, 12, 25, tzinfo=TZ()).isoformat(' ')
'2002-12-25 00:00:00-06:39'
>>> datetime(2009, 11, 27, microsecond=100,
tzinfo=TZ()).isoformat()
'2009-11-27T00:00:00.000100-06:39'
```

The optional argument *timespec* specifies the number of additional components of the time to include (the default is `'auto'`). It can be one of the following:

- `'auto'`: Same as `'seconds'` if `microsecond` is 0, same as `'microseconds'` otherwise.

- `'hours'`: Include the **`hour`** in the two-digit `HH` format.
- `'minutes'`: Include **`hour`** and **`minute`** in `HH:MM` format.
- `'seconds'`: Include **`hour`**, **`minute`**, and **`second`** in `HH:MM:SS` format.
- `'milliseconds'`: Include full time, but truncate fractional second part to milliseconds. `HH:MM:SS.sss` format.
- `'microseconds'`: Include full time in `HH:MM:SS.ffffff` format.

Note

Excluded time components are truncated, not rounded.

`ValueError` will be raised on an invalid *timespec* argument:

```
>>> from datetime import datetime
>>> datetime.now().isoformat(timespec='minutes')
'2002-12-25T00:00'
>>> dt = datetime(2015, 1, 1, 12, 30, 59, 0)
>>> dt.isoformat(timespec='microseconds')
'2015-01-01T12:30:59.000000'
```

Changed in version 3.6: Added the *timespec* parameter.

datetime.__str__()

For a **`datetime`** instance `d`, `str(d)` is equivalent to `d.isoformat(' ')`.

datetime.ctime()

Return a string representing the date and time:

```
>>> from datetime import datetime
>>> datetime(2002, 12, 4, 20, 30, 40).ctime()
'Wed Dec  4 20:30:40 2002'
```

The output string will *not* include time zone information, regardless of whether the input is aware or naive.

`d.ctime()` is equivalent to:

```
time.ctime(time.mktime(d.timetuple()))
```

on platforms where the native C **`ctime()`** function (which **`time.ctime()`** invokes, but which **`datetime.ctime()`** does not invoke) conforms to the C standard.

datetime.strftime(*format*)

Return a string representing the date and time, controlled by an explicit format string. See also strftime() and strptime() Behavior and **`datetime.isoformat()`**.

datetime.__format__(*format*)

Same as **`datetime.strftime()`**. This makes it possible to specify a format string for a **`datetime`** object in formatted string literals and when using **`str.format()`**. See also strftime() and strptime() Behavior and **`datetime.isoformat()`**.

Examples of Usage: `datetime`

Examples of working with **`datetime`** objects:

```
>>> from datetime import datetime, date, time, timezone

>>> # Using datetime.combine()
>>> d = date(2005, 7, 14)
>>> t = time(12, 30)
>>> datetime.combine(d, t)
datetime.datetime(2005, 7, 14, 12, 30)

>>> # Using datetime.now()
>>> datetime.now()
datetime.datetime(2007, 12, 6, 16, 29, 43, 79043)   # GMT +1
>>> datetime.now(timezone.utc)
datetime.datetime(2007, 12, 6, 15, 29, 43, 79060,
tzinfo=datetime.timezone.utc)
```

```
>>> # Using datetime.strptime()
>>> dt = datetime.strptime("21/11/06 16:30", "%d/%m/%y %H:%M")
>>> dt
datetime.datetime(2006, 11, 21, 16, 30)

>>> # Using datetime.timetuple() to get tuple of all attributes
>>> tt = dt.timetuple()
>>> for it in tt:
...     print(it)
...
2006    # year
11      # month
21      # day
16      # hour
30      # minute
0       # second
1       # weekday (0 = Monday)
325     # number of days since 1st January
-1      # dst - method tzinfo.dst() returned None

>>> # Date in ISO format
>>> ic = dt.isocalendar()
>>> for it in ic:
...     print(it)
...
2006    # ISO year
47      # ISO week
2       # ISO weekday

>>> # Formatting a datetime
>>> dt.strftime("%A, %d. %B %Y %I:%M%p")
'Tuesday, 21. November 2006 04:30PM'
>>> 'The {1} is {0:%d}, the {2} is {0:%B}, the {3} is
{0:%I:%M%p}.'.format(dt, "day", "month", "time")
'The day is 21, the month is November, the time is 04:30PM.'
```

The example below defines a `tzinfo` subclass capturing time zone information for Kabul, Afghanistan, which used +4 UTC until 1945 and then +4:30 UTC thereafter:

```
from datetime import timedelta, datetime, tzinfo, timezone
```

```
class KabulTz(tzinfo):
    # Kabul used +4 until 1945, when they moved to +4:30
    UTC_MOVE_DATE = datetime(1944, 12, 31, 20,
tzinfo=timezone.utc)

    def utcoffset(self, dt):
        if dt.year < 1945:
            return timedelta(hours=4)
        elif (1945, 1, 1, 0, 0) <= dt.timetuple()[:5] < (1945,
1, 1, 0, 30):
            # An ambiguous ("imaginary") half-hour range
representing
            # a 'fold' in time due to the shift from +4 to
+4:30.
            # If dt falls in the imaginary range, use fold to
decide how
            # to resolve. See PEP495.
            return timedelta(hours=4, minutes=(30 if dt.fold
else 0))
        else:
            return timedelta(hours=4, minutes=30)

    def fromutc(self, dt):
        # Follow same validations as in datetime.tzinfo
        if not isinstance(dt, datetime):
            raise TypeError("fromutc() requires a datetime
argument")
        if dt.tzinfo is not self:
            raise ValueError("dt.tzinfo is not self")

        # A custom implementation is required for fromutc as
        # the input to this function is a datetime with utc
values
        # but with a tzinfo set to self.
        # See datetime.astimezone or fromtimestamp.
        if dt.replace(tzinfo=timezone.utc) >=
self.UTC_MOVE_DATE:
            return dt + timedelta(hours=4, minutes=30)
        else:
            return dt + timedelta(hours=4)

    def dst(self, dt):
        # Kabul does not observe daylight saving time.
        return timedelta(0)

    def tzname(self, dt):
        if dt >= self.UTC_MOVE_DATE:
            return "+04:30"
        return "+04"
```

Usage of `KabulTz` from above:

```
>>> tz1 = KabulTz()

>>> # Datetime before the change
>>> dt1 = datetime(1900, 11, 21, 16, 30, tzinfo=tz1)
>>> print(dt1.utcoffset())
4:00:00

>>> # Datetime after the change
>>> dt2 = datetime(2006, 6, 14, 13, 0, tzinfo=tz1)
>>> print(dt2.utcoffset())
4:30:00

>>> # Convert datetime to another time zone
>>> dt3 = dt2.astimezone(timezone.utc)
>>> dt3
datetime.datetime(2006, 6, 14, 8, 30,
tzinfo=datetime.timezone.utc)
>>> dt2
datetime.datetime(2006, 6, 14, 13, 0, tzinfo=KabulTz())
>>> dt2 == dt3
True
```

`time` Objects

A `time` object represents a (local) time of day, independent of any particular day, and subject to adjustment via a `tzinfo` object.

class datetime.time(*hour=0*, *minute=0*, *second=0*, *microsecond=0*, *tzinfo=None*, ***, *fold=0*)

All arguments are optional. *tzinfo* may be `None`, or an instance of a **`tzinfo`** subclass. The remaining arguments must be integers in the following ranges:

- `0 <= hour < 24`,
- `0 <= minute < 60`,
- `0 <= second < 60`,
- `0 <= microsecond < 1000000`,
- `fold in [0, 1]`.

If an argument outside those ranges is given, **`ValueError`** is raised. All default to 0 except *tzinfo*, which defaults to `None`.

Class attributes:

time.min

The earliest representable **`time`**, `time(0, 0, 0, 0)`.

time.max

The latest representable **`time`**, `time(23, 59, 59, 999999)`.

time.resolution

The smallest possible difference between non-equal **`time`** objects, `timedelta(microseconds=1)`, although note that arithmetic on **`time`** objects is not supported.

Instance attributes (read-only):

time.hour

In `range(24)`.

time.minute

In `range(60)`.

time.second

In `range(60)`.

time.microsecond

In `range(1000000)`.

time.tzinfo

The object passed as the tzinfo argument to the **`time`** constructor, or `None` if none was passed.

time.fold

In `[0, 1]`. Used to disambiguate wall times during a repeated interval. (A repeated interval occurs when clocks are rolled back at the end of daylight saving time or when the UTC offset for the current zone is decreased for political reasons.) The values 0 and 1 represent, respectively, the earlier and later of the two moments with the same wall time representation.

Added in version 3.6.

`time` objects support equality and order comparisons, where `a` is considered less than `b` when `a` precedes `b` in time.

Naive and aware **`time`** objects are never equal. Order comparison between naive and aware **`time`** objects raises **`TypeError`**.

If both comparands are aware, and have the same **`tzinfo`** attribute, the **`tzinfo`** and **`fold`** attributes are ignored and the base times are compared. If both comparands are aware and have different **`tzinfo`** attributes, the comparands are first adjusted by subtracting their UTC offsets (obtained from `self.utcoffset()`).

Changed in version 3.3: Equality comparisons between aware and naive **`time`** instances don’t raise **`TypeError`**.

In Boolean contexts, a **`time`** object is always considered to be true.

Changed in version 3.5: Before Python 3.5, a **`time`** object was considered to be false if it represented midnight in UTC. This behavior was considered obscure and error-prone and has been removed in Python 3.5. See bpo-13936 [https://bugs.python.org/issue?@action=redirect&bpo=13936] for full details.

Other constructors:

classmethod time.fromisoformat(*time_string*)

Return a **`time`** corresponding to a *time_string* in any valid ISO 8601 format, with the following exceptions:

1. Time zone offsets may have fractional seconds.
2. The leading `T`, normally required in cases where there may be ambiguity between a date and a time, is not required.
3. Fractional seconds may have any number of digits (anything beyond 6 will be truncated).
4. Fractional hours and minutes are not supported.

Examples:

```
>>> from datetime import time
>>> time.fromisoformat('04:23:01')
datetime.time(4, 23, 1)
>>> time.fromisoformat('T04:23:01')
datetime.time(4, 23, 1)
>>> time.fromisoformat('T042301')
datetime.time(4, 23, 1)
>>> time.fromisoformat('04:23:01.000384')
datetime.time(4, 23, 1, 384)
>>> time.fromisoformat('04:23:01,000384')
datetime.time(4, 23, 1, 384)
>>> time.fromisoformat('04:23:01+04:00')
datetime.time(4, 23, 1,
tzinfo=datetime.timezone(datetime.timedelta(seconds=14400)))
>>> time.fromisoformat('04:23:01Z')
datetime.time(4, 23, 1, tzinfo=datetime.timezone.utc)
>>> time.fromisoformat('04:23:01+00:00')
datetime.time(4, 23, 1, tzinfo=datetime.timezone.utc)
```

Added in version 3.7.

Changed in version 3.11: Previously, this method only supported formats that could be emitted by **`time.isoformat()`**.

classmethod time.strptime(*date_string*, *format*)

Return a **`time`** corresponding to *date_string*, parsed according to *format*.

If *format* does not contain microseconds or timezone information, this is equivalent to:

```
time(*(time.strptime(date_string, format)[3:6]))
```

`ValueError` is raised if the *date_string* and *format* cannot be parsed by **`time.strptime()`** or if it returns a value which is not a time tuple. See also strftime() and strptime() Behavior and **`time.fromisoformat()`**.

Added in version 3.14.

Instance methods:

time.replace(*hour=self.hour*, *minute=self.minute*, *second=self.second*, *microsecond=self.microsecond*, *tzinfo=self.tzinfo*, *, *fold=0*)

Return a new **`time`** with the same values, but with specified parameters updated. Note that `tzinfo=None` can be specified to create a naive **`time`** from an aware **`time`**, without conversion of the time data.

[`time`] objects are also supported by generic function [`copy.replace()`].

Changed in version 3.6: Added the *fold* parameter.

time.isoformat(*timespec*='*auto*')

Return a string representing the time in ISO 8601 format, one of:

- `HH:MM:SS.ffffff`, if **`microsecond`** is not 0
- `HH:MM:SS`, if **`microsecond`** is 0
- `HH:MM:SS.ffffff+HH:MM[:SS[.ffffff]]`, if **`utcoffset()`** does not return `None`
- `HH:MM:SS+HH:MM[:SS[.ffffff]]`, if **`microsecond`** is 0 and **`utcoffset()`** does not return `None`

The optional argument *timespec* specifies the number of additional components of the time to include (the default is `'auto'`). It can be one of the following:

- `'auto'`: Same as `'seconds'` if **`microsecond`** is 0, same as `'microseconds'` otherwise.
- `'hours'`: Include the **`hour`** in the two-digit `HH` format.
- `'minutes'`: Include **`hour`** and **`minute`** in `HH:MM` format.
- `'seconds'`: Include **`hour`**, **`minute`**, and **`second`** in `HH:MM:SS` format.
- `'milliseconds'`: Include full time, but truncate fractional second part to milliseconds. `HH:MM:SS.sss` format.
- `'microseconds'`: Include full time in `HH:MM:SS.ffffff` format.

Note

Excluded time components are truncated, not rounded.

`ValueError` will be raised on an invalid *timespec* argument.

Example:

```
>>> from datetime import time
>>> time(hour=12, minute=34, second=56,
microsecond=123456).isoformat(timespec='minutes')
'12:34'
>>> dt = time(hour=12, minute=34, second=56, microsecond=0)
>>> dt.isoformat(timespec='microseconds')
'12:34:56.000000'
>>> dt.isoformat(timespec='auto')
'12:34:56'
```

Changed in version 3.6: Added the *timespec* parameter.

time.__str__()

For a time `t`, `str(t)` is equivalent to `t.isoformat()`.

time.strftime(*format*)

Return a string representing the time, controlled by an explicit format string. See also strftime() and strptime() Behavior and **`time.isoformat()`**.

time.__format__(*format*)

Same as **`time.strftime()`**. This makes it possible to specify a format string for a **`time`** object in formatted string literals and when using **`str.format()`**. See also strftime() and strptime() Behavior and **`time.isoformat()`**.

time.utcoffset()

If **`tzinfo`** is `None`, returns `None`, else returns `self.tzinfo.utcoffset(None)`, and raises an exception if the latter doesn't return `None` or a **`timedelta`** object with magnitude less than one day.

Changed in version 3.7: The UTC offset is not restricted to a whole number of minutes.

time.dst()

If **`tzinfo`** is `None`, returns `None`, else returns `self.tzinfo.dst(None)`, and raises an exception if the latter doesn't return `None`, or a **`timedelta`** object with magnitude less than one day.

Changed in version 3.7: The DST offset is not restricted to a whole number of minutes.

time.tzname()

If **`tzinfo`** is `None`, returns `None`, else returns `self.tzinfo.tzname(None)`, or raises an exception if the latter doesn't return `None` or a string object.

Examples of Usage: `time`

Examples of working with a **`time`** object:

```
>>> from datetime import time, tzinfo, timedelta
>>> class TZ1(tzinfo):
...     def utcoffset(self, dt):
...         return timedelta(hours=1)
...     def dst(self, dt):
...         return timedelta(0)
...     def tzname(self,dt):
...         return "+01:00"
...     def  __repr__(self):
...         return f"{self.__class__.__name__}()"
...
>>> t = time(12, 10, 30, tzinfo=TZ1())
>>> t
datetime.time(12, 10, 30, tzinfo=TZ1())
>>> t.isoformat()
'12:10:30+01:00'
>>> t.dst()
datetime.timedelta(0)
>>> t.tzname()
'+01:00'
>>> t.strftime("%H:%M:%S %Z")
```

```
'12:10:30 +01:00'
>>> 'The {} is {:%H:%M}.'.format("time", t)
'The time is 12:10.'
```

`tzinfo` Objects

class datetime.tzinfo

This is an abstract base class, meaning that this class should not be instantiated directly. Define a subclass of `tzinfo` to capture information about a particular time zone.

An instance of (a concrete subclass of) `tzinfo` can be passed to the constructors for `datetime` and `time` objects. The latter objects view their attributes as being in local time, and the `tzinfo` object supports methods revealing offset of local time from UTC, the name of the time zone, and DST offset, all relative to a date or time object passed to them.

You need to derive a concrete subclass, and (at least) supply implementations of the standard `tzinfo` methods needed by the `datetime` methods you use. The **`datetime`** module provides `timezone`, a simple concrete subclass of `tzinfo` which can represent time zones with fixed offset from UTC such as UTC itself or North American EST and EDT.

Special requirement for pickling: A `tzinfo` subclass must have an `__init__()` method that can be called with no arguments, otherwise it can be pickled but possibly not unpickled again. This is a technical requirement that may be relaxed in the future.

A concrete subclass of `tzinfo` may need to implement the following methods. Exactly which methods are needed depends on the uses made of aware **`datetime`** objects. If in doubt, simply implement all of them.

tzinfo.utcoffset(*dt*)

Return offset of local time from UTC, as a `timedelta` object that is positive east of UTC. If local time is west of UTC, this should be negative.

This represents the *total* offset from UTC; for example, if a `tzinfo` object represents both time zone and DST adjustments, `utcoffset()` should return their sum. If the UTC offset isn't known, return `None`. Else the value returned must be a `timedelta` object strictly between `-timedelta(hours=24)` and `timedelta(hours=24)` (the magnitude of the offset must be less than one day). Most implementations of `utcoffset()` will probably look like one of these two:

```
return CONSTANT                 # fixed-offset class
return CONSTANT + self.dst(dt)  # daylight-aware class
```

If `utcoffset()` does not return `None`, `dst()` should not return `None` either.

The default implementation of `utcoffset()` raises `NotImplementedError`.

Changed in version 3.7: The UTC offset is not restricted to a whole number of minutes.

tzinfo.dst(*dt*)

Return the daylight saving time (DST) adjustment, as a `timedelta` object or `None` if DST information isn't known.

Return `timedelta(0)` if DST is not in effect. If DST is in effect, return the offset as a `timedelta` object (see `utcoffset()` for details). Note that DST offset, if applicable, has already been added to the UTC offset returned by `utcoffset()`, so there’s no need to consult `dst()` unless you’re interested in obtaining DST info separately. For example, `datetime.timetuple()` calls its `tzinfo` attribute’s `dst()` method to determine how the `tm_isdst` flag should be set, and `tzinfo.fromutc()` calls `dst()` to account for DST changes when crossing time zones.

An instance *tz* of a `tzinfo` subclass that models both standard and daylight times must be consistent in this sense:

```
tz.utcoffset(dt) - tz.dst(dt)
```

must return the same result for every `datetime` *dt* with `dt.tzinfo == tz`. For sane `tzinfo` subclasses, this expression yields the time zone’s “standard offset”, which should not depend on the date or the time, but only on geographic location. The implementation of `datetime.astimezone()` relies on this, but cannot detect violations; it’s the programmer’s responsibility to ensure it. If a `tzinfo` subclass cannot guarantee this, it may be able to override the default implementation of `tzinfo.fromutc()` to work correctly with `astimezone()` regardless.

Most implementations of `dst()` will probably look like one of these two:

```
def dst(self, dt):
    # a fixed-offset class:  doesn't account for DST
    return timedelta(0)
```

or:

```
def dst(self, dt):
    # Code to set dston and dstoff to the time zone's DST
    # transition times based on the input dt.year, and
```

```
expressed
    # in standard local time.

    if dston <= dt.replace(tzinfo=None) < dstoff:
        return timedelta(hours=1)
    else:
        return timedelta(0)
```

The default implementation of `dst()` raises `NotImplementedError`.

Changed in version 3.7: The DST offset is not restricted to a whole number of minutes.

tzinfo.tzname(*dt*)

Return the time zone name corresponding to the `datetime` object *dt*, as a string. Nothing about string names is defined by the **`datetime`** module, and there's no requirement that it mean anything in particular. For example, `"GMT"`, `"UTC"`, `"-500"`, `"-5:00"`, `"EDT"`, `"US/Eastern"`, `"America/New York"` are all valid replies. Return `None` if a string name isn't known. Note that this is a method rather than a fixed string primarily because some `tzinfo` subclasses will wish to return different names depending on the specific value of *dt* passed, especially if the `tzinfo` class is accounting for daylight time.

The default implementation of `tzname()` raises `NotImplementedError`.

These methods are called by a `datetime` or `time` object, in response to their methods of the same names. A `datetime` object passes itself as the argument, and a `time` object passes `None` as the argument. A `tzinfo` subclass's methods should therefore be prepared to accept a *dt* argument of `None`, or of class `datetime`.

When `None` is passed, it’s up to the class designer to decide the best response. For example, returning `None` is appropriate if the class wishes to say that time objects don’t participate in the **`tzinfo`** protocols. It may be more useful for `utcoffset(None)` to return the standard UTC offset, as there is no other convention for discovering the standard offset.

When a **`datetime`** object is passed in response to a **`datetime`** method, `dt.tzinfo` is the same object as *self*. **`tzinfo`** methods can rely on this, unless user code calls **`tzinfo`** methods directly. The intent is that the **`tzinfo`** methods interpret *dt* as being in local time, and not need worry about objects in other time zones.

There is one more **`tzinfo`** method that a subclass may wish to override:

tzinfo.fromutc(*dt*)

> This is called from the default **`datetime.astimezone()`** implementation. When called from that, `dt.tzinfo` is *self*, and *dt*’s date and time data are to be viewed as expressing a UTC time. The purpose of **`fromutc()`** is to adjust the date and time data, returning an equivalent datetime in *self*’s local time.
>
> Most **`tzinfo`** subclasses should be able to inherit the default **`fromutc()`** implementation without problems. It’s strong enough to handle fixed-offset time zones, and time zones accounting for both standard and daylight time, and the latter even if the DST transition times differ in different years. An example of a time zone the default **`fromutc()`** implementation may not handle correctly in all cases is one where the standard offset (from UTC) depends on the specific date and time passed, which can happen for political reasons. The default implementations of **`astimezone()`** and **`fromutc()`** may not produce the result

you want if the result is one of the hours straddling the moment the standard offset changes.

Skipping code for error cases, the default `fromutc()` implementation acts like:

```
def fromutc(self, dt):
    # raise ValueError error if dt.tzinfo is not self
    dtoff = dt.utcoffset()
    dtdst = dt.dst()
    # raise ValueError if dtoff is None or dtdst is None
    delta = dtoff - dtdst  # this is self's standard offset
    if delta:
        dt += delta   # convert to standard local time
        dtdst = dt.dst()
        # raise ValueError if dtdst is None
    if dtdst:
        return dt + dtdst
    else:
        return dt
```

In the following **`tzinfo_examples.py`** file there are some examples of `tzinfo` classes:

```
from datetime import tzinfo, timedelta, datetime

ZERO = timedelta(0)
HOUR = timedelta(hours=1)
SECOND = timedelta(seconds=1)

# A class capturing the platform's idea of local time.
# (May result in wrong values on historical times in
#  timezones where UTC offset and/or the DST rules had
#  changed in the past.)
import time as _time

STDOFFSET = timedelta(seconds = -_time.timezone)
if _time.daylight:
    DSTOFFSET = timedelta(seconds = -_time.altzone)
else:
    DSTOFFSET = STDOFFSET
```

```
DSTDIFF = DSTOFFSET - STDOFFSET

class LocalTimezone(tzinfo):

    def fromutc(self, dt):
        assert dt.tzinfo is self
        stamp = (dt - datetime(1970, 1, 1, tzinfo=self)) //
SECOND
        args = _time.localtime(stamp)[:6]
        dst_diff = DSTDIFF // SECOND
        # Detect fold
        fold = (args == _time.localtime(stamp - dst_diff))
        return datetime(*args, microsecond=dt.microsecond,
                        tzinfo=self, fold=fold)

    def utcoffset(self, dt):
        if self._isdst(dt):
            return DSTOFFSET
        else:
            return STDOFFSET

    def dst(self, dt):
        if self._isdst(dt):
            return DSTDIFF
        else:
            return ZERO

    def tzname(self, dt):
        return _time.tzname[self._isdst(dt)]

    def _isdst(self, dt):
        tt = (dt.year, dt.month, dt.day,
              dt.hour, dt.minute, dt.second,
              dt.weekday(), 0, 0)
        stamp = _time.mktime(tt)
        tt = _time.localtime(stamp)
        return tt.tm_isdst > 0

Local = LocalTimezone()


# A complete implementation of current DST rules for major US
time zones.
```

```
def first_sunday_on_or_after(dt):
    days_to_go = 6 - dt.weekday()
    if days_to_go:
        dt += timedelta(days_to_go)
    return dt


# US DST Rules
#
# This is a simplified (i.e., wrong for a few cases) set of
rules for US
# DST start and end times. For a complete and up-to-date set of
DST rules
# and timezone definitions, visit the Olson Database (or try
pytz):
# http://www.twinsun.com/tz/tz-link.htm
# https://sourceforge.net/projects/pytz/ (might not be up-to-
date)
#
# In the US, since 2007, DST starts at 2am (standard time) on
the second
# Sunday in March, which is the first Sunday on or after Mar 8.
DSTSTART_2007 = datetime(1, 3, 8, 2)
# and ends at 2am (DST time) on the first Sunday of Nov.
DSTEND_2007 = datetime(1, 11, 1, 2)
# From 1987 to 2006, DST used to start at 2am (standard time)
on the first
# Sunday in April and to end at 2am (DST time) on the last
# Sunday of October, which is the first Sunday on or after Oct
25.
DSTSTART_1987_2006 = datetime(1, 4, 1, 2)
DSTEND_1987_2006 = datetime(1, 10, 25, 2)
# From 1967 to 1986, DST used to start at 2am (standard time)
on the last
# Sunday in April (the one on or after April 24) and to end at
2am (DST time)
# on the last Sunday of October, which is the first Sunday
# on or after Oct 25.
DSTSTART_1967_1986 = datetime(1, 4, 24, 2)
DSTEND_1967_1986 = DSTEND_1987_2006

def us_dst_range(year):
    # Find start and end times for US DST. For years before
1967, return
    # start = end for no DST.
    if 2006 < year:
        dststart, dstend = DSTSTART_2007, DSTEND_2007
    elif 1986 < year < 2007:
        dststart, dstend = DSTSTART_1987_2006, DSTEND_1987_2006
    elif 1966 < year < 1987:
        dststart, dstend = DSTSTART_1967_1986, DSTEND_1967_1986
    else:
        return (datetime(year, 1, 1), ) * 2

    start =
first_sunday_on_or_after(dststart.replace(year=year))
    end = first_sunday_on_or_after(dstend.replace(year=year))
    return start, end


class USTimeZone(tzinfo):

    def __init__(self, hours, reprname, stdname, dstname):
        self.stdoffset = timedelta(hours=hours)
        self.reprname = reprname
        self.stdname = stdname
        self.dstname = dstname

    def __repr__(self):
        return self.reprname

    def tzname(self, dt):
        if self.dst(dt):
            return self.dstname
        else:
            return self.stdname

    def utcoffset(self, dt):
        return self.stdoffset + self.dst(dt)

    def dst(self, dt):
        if dt is None or dt.tzinfo is None:
            # An exception may be sensible here, in one or both
cases.
            # It depends on how you want to treat them.  The
```

```
default
            # fromutc() implementation (called by the default
astimezone()
            # implementation) passes a datetime with dt.tzinfo
is self.
            return ZERO
        assert dt.tzinfo is self
        start, end = us_dst_range(dt.year)
        # Can't compare naive to aware objects, so strip the
timezone from
        # dt first.
        dt = dt.replace(tzinfo=None)
        if start + HOUR <= dt < end - HOUR:
            # DST is in effect.
            return HOUR
        if end - HOUR <= dt < end:
            # Fold (an ambiguous hour): use dt.fold to
disambiguate.
            return ZERO if dt.fold else HOUR
        if start <= dt < start + HOUR:
            # Gap (a non-existent hour): reverse the fold rule.
            return HOUR if dt.fold else ZERO
        # DST is off.
        return ZERO

    def fromutc(self, dt):
        assert dt.tzinfo is self
        start, end = us_dst_range(dt.year)
        start = start.replace(tzinfo=self)
        end = end.replace(tzinfo=self)
        std_time = dt + self.stdoffset
        dst_time = std_time + HOUR
        if end <= dst_time < end + HOUR:
            # Repeated hour
            return std_time.replace(fold=1)
        if std_time < start or dst_time >= end:
            # Standard time
            return std_time
        if start <= std_time < end - HOUR:
            # Daylight saving time
            return dst_time
```

```
Eastern  = USTimeZone(-5, "Eastern",  "EST", "EDT")
Central  = USTimeZone(-6, "Central",  "CST", "CDT")
Mountain = USTimeZone(-7, "Mountain", "MST", "MDT")
Pacific  = USTimeZone(-8, "Pacific",  "PST", "PDT")
```

Note that there are unavoidable subtleties twice per year in a **`tzinfo`** subclass accounting for both standard and daylight time, at the DST transition points. For concreteness, consider US Eastern (UTC -0500), where EDT begins the minute after 1:59 (EST) on the second Sunday in March, and ends the minute after 1:59 (EDT) on the first Sunday in November:

```
  UTC   3:MM  4:MM  5:MM  6:MM  7:MM  8:MM
  EST  22:MM 23:MM  0:MM  1:MM  2:MM  3:MM
  EDT  23:MM  0:MM  1:MM  2:MM  3:MM  4:MM

start  22:MM 23:MM  0:MM  1:MM  3:MM  4:MM

  end  23:MM  0:MM  1:MM  1:MM  2:MM  3:MM
```

When DST starts (the “start” line), the local wall clock leaps from 1:59 to 3:00. A wall time of the form 2:MM doesn’t really make sense on that day, so `astimezone(Eastern)` won’t deliver a result with `hour == 2` on the day DST begins. For example, at the Spring forward transition of 2016, we get:

```
>>> from datetime import datetime, timezone
>>> from tzinfo_examples import HOUR, Eastern
>>> u0 = datetime(2016, 3, 13, 5, tzinfo=timezone.utc)
>>> for i in range(4):
...     u = u0 + i*HOUR
...     t = u.astimezone(Eastern)
...     print(u.time(), 'UTC =', t.time(), t.tzname())
...
05:00:00 UTC = 00:00:00 EST
06:00:00 UTC = 01:00:00 EST
07:00:00 UTC = 03:00:00 EDT
08:00:00 UTC = 04:00:00 EDT
```

When DST ends (the “end” line), there’s a potentially worse problem: there’s an hour that can’t be spelled unambiguously in local wall time: the last hour of daylight time. In Eastern, that’s times of the form 5:MM UTC on the day daylight time ends. The local wall clock leaps from 1:59 (daylight time) back to 1:00 (standard time) again. Local times of the form 1:MM are ambiguous. `astimezone()` mimics the local clock’s behavior by mapping two adjacent UTC hours into the same local hour then. In the Eastern example, UTC times of the form 5:MM and 6:MM both map to 1:MM when converted to Eastern, but earlier times have the `fold` attribute set to 0 and the later times have it set to 1. For example, at the Fall back transition of 2016, we get:

```
>>> u0 = datetime(2016, 11, 6, 4, tzinfo=timezone.utc)
>>> for i in range(4):
...     u = u0 + i*HOUR
...     t = u.astimezone(Eastern)
...     print(u.time(), 'UTC =', t.time(), t.tzname(), t.fold)
...
04:00:00 UTC = 00:00:00 EDT 0
05:00:00 UTC = 01:00:00 EDT 0
06:00:00 UTC = 01:00:00 EST 1
07:00:00 UTC = 02:00:00 EST 0
```

Note that the `datetime` instances that differ only by the value of the `fold` attribute are considered equal in comparisons.

Applications that can’t bear wall-time ambiguities should explicitly check the value of the `fold` attribute or avoid using hybrid `tzinfo` subclasses; there are no ambiguities when using `timezone`, or any other fixed-offset `tzinfo` subclass (such as a class representing only EST (fixed offset -5 hours), or only EDT (fixed offset -4 hours)).

See also

`zoneinfo`

> The **`datetime`** module has a basic [`timezone`] class (for handling arbitrary fixed offsets from UTC) and its [`timezone.utc`] attribute (a UTC **`timezone`** instance).
>
> `zoneinfo` brings the *IANA time zone database* (also known as the Olson database) to Python, and its usage is recommended.

IANA time zone database [https://www.iana.org/time-zones]
  The Time Zone Database (often called tz, tzdata or zoneinfo) contains code and data that represent the history of local time for many representative locations around the globe. It is updated periodically to reflect changes made by political bodies to time zone boundaries, UTC offsets, and daylight-saving rules.

`timezone` Objects

The `timezone` class is a subclass of `tzinfo`, each instance of which represents a time zone defined by a fixed offset from UTC.

Objects of this class cannot be used to represent time zone information in the locations where different offsets are used in different days of the year or where historical changes have been made to civil time.

class datetime.timezone(*offset*, *name=None*)

  The *offset* argument must be specified as a `timedelta` object representing the difference between the local time and UTC. It must be strictly between `-timedelta(hours=24)` and `timedelta(hours=24)`, otherwise `ValueError` is raised.

  The *name* argument is optional. If specified it must be a string that will be used as the value returned by the `datetime.tzname()` method.

Added in version 3.2.

Changed in version 3.7: The UTC offset is not restricted to a whole number of minutes.

timezone.utcoffset(*dt*)

Return the fixed value specified when the `timezone` instance is constructed.

The *dt* argument is ignored. The return value is a `timedelta` instance equal to the difference between the local time and UTC.

Changed in version 3.7: The UTC offset is not restricted to a whole number of minutes.

timezone.tzname(*dt*)

Return the fixed value specified when the `timezone` instance is constructed.

If *name* is not provided in the constructor, the name returned by `tzname(dt)` is generated from the value of the `offset` as follows. If *offset* is `timedelta(0)`, the name is “UTC”, otherwise it is a string in the format `UTC±HH:MM`, where ± is the sign of `offset`, HH and MM are two digits of `offset.hours` and `offset.minutes` respectively.

Changed in version 3.6: Name generated from `offset=timedelta(0)` is now plain `'UTC'`, not `'UTC+00:00'`.

timezone.dst(*dt*)

Always returns `None`.

timezone.fromutc(*dt*)

Return `dt + offset`. The *dt* argument must be an aware **`datetime`** instance, with `tzinfo` set to `self`.

Class attributes:

timezone.utc

The UTC time zone, `timezone(timedelta(0))`.

`strftime()` and `strptime()` Behavior

`date`, **`datetime`**, and **`time`** objects all support a `strftime(format)` method, to create a string representing the time under the control of an explicit format string.

Conversely, the **`date.strptime()`**, **`datetime.strptime()`** and **`time.strptime()`** class methods create an object from a string representing the time and a corresponding format string.

The table below provides a high-level comparison of **`strftime()`** versus **`strptime()`**:

	`strftime`	**`strptime`**
Usage	Convert object to a string according to a given format	Parse a string into an object given a corresponding format
Type of method	Instance method	Class method
Signature	`strftime(format)`	`strptime(date_string, format)`

`strftime()` and `strptime()` Format Codes

These methods accept format codes that can be used to parse and format dates:

```
>>> datetime.strptime('31/01/22 23:59:59.999999',
...                   '%d/%m/%y %H:%M:%S.%f')
datetime.datetime(2022, 1, 31, 23, 59, 59, 999999)
>>> _.strftime('%a %d %b %Y, %I:%M%p')
'Mon 31 Jan 2022, 11:59PM'
```

The following is a list of all the format codes that the 1989 C standard requires, and these work on all platforms with a standard C implementation.

Directive	Meaning	Example	Notes
`%a`	Weekday as locale’s abbreviated name.	Sun, Mon, …, Sat (en_US); So, Mo, …, Sa (de_DE)	(1)
`%A`	Weekday as locale’s full name.	Sunday, Monday, …, Saturday (en_US); Sonntag, Montag, …, Samstag (de_DE)	(1)
`%w`	Weekday as a decimal number, where 0 is Sunday and 6 is Saturday.	0, 1, …, 6	

Directive	Meaning	Example	Notes
`%d`	Day of the month as a zero-padded decimal number.	01, 02, …, 31	(9)
`%b`	Month as locale’s abbreviated name.	Jan, Feb, …, Dec (en_US); Jan, Feb, …, Dez (de_DE)	(1)
`%B`	Month as locale’s full name.	January, February, …, December (en_US); Januar, Februar, …, Dezember (de_DE)	(1)
`%m`	Month as a zero-padded decimal number.	01, 02, …, 12	(9)
`%y`	Year without century as a zero-padded decimal number.	00, 01, …, 99	(9)
`%Y`	Year with century as a decimal number.	0001, 0002, …, 2013, 2014, …, 9998, 9999	(2)

Directive	Meaning	Example	Notes
`%H`	Hour (24-hour clock) as a zero-padded decimal number.	00, 01, …, 23	(9)
`%I`	Hour (12-hour clock) as a zero-padded decimal number.	01, 02, …, 12	(9)
`%p`	Locale’s equivalent of either AM or PM.	AM, PM (en_US); am, pm (de_DE)	(1), (3)
`%M`	Minute as a zero-padded decimal number.	00, 01, …, 59	(9)
`%S`	Second as a zero-padded decimal number.	00, 01, …, 59	(4), (9)
`%f`	Microsecond as a decimal number, zero-padded to 6 digits.	000000, 000001, …, 999999	(5)
`%z`	UTC offset in the form `±HHMM[SS[.ffffff]]`	(empty), +0000, -0400, +1030,	(6)

Directive	Meaning	Example	Notes
	(empty string if the object is naive).	+063415, -030712.345216	
`%Z`	Time zone name (empty string if the object is naive).	(empty), UTC, GMT	(6)
`%j`	Day of the year as a zero-padded decimal number.	001, 002, …, 366	(9)
`%U`	Week number of the year (Sunday as the first day of the week) as a zero-padded decimal number. All days in a new year preceding the first Sunday are considered to be in week 0.	00, 01, …, 53	(7), (9)
`%W`	Week number of the year (Monday as the first day of the week) as a zero-padded decimal number. All days in a new year preceding the first Monday are	00, 01, …, 53	(7), (9)

Directive	Meaning	Example	Notes
	considered to be in week 0.		
`%c`	Locale’s appropriate date and time representation.	Tue Aug 16 21:30:00 1988 (en_US); Di 16 Aug 21:30:00 1988 (de_DE)	(1)
`%x`	Locale’s appropriate date representation.	08/16/88 (None); 08/16/1988 (en_US); 16.08.1988 (de_DE)	(1)
`%X`	Locale’s appropriate time representation.	21:30:00 (en_US); 21:30:00 (de_DE)	(1)
`%%`	A literal `'%'` character.	%	

Several additional directives not required by the C89 standard are included for convenience. These parameters all correspond to ISO 8601 date values.

Directive	Meaning	Example	Notes
`%G`	ISO 8601 year with century representing the year that contains the greater part of the ISO week (`%V`).	0001, 0002, …, 2013, 2014, …, 9998, 9999	(8)
`%u`	ISO 8601 weekday as a decimal number where 1 is Monday.	1, 2, …, 7	
`%V`	ISO 8601 week as a decimal number with Monday as the first day of the week. Week 01 is the week containing Jan 4.	01, 02, …, 53	(8), (9)
`%:z`	UTC offset in the form `±HH:MM[:SS[.ffffff]]` (empty string if the object is naive).	(empty), +00:00, -04:00, +10:30, +06:34:15, -03:07:12.345216	(6)

These may not be available on all platforms when used with the `strftime()` method. The ISO 8601 year and ISO 8601 week directives are not interchangeable with the year and week number directives above. Calling `strptime()` with incomplete or ambiguous ISO 8601 directives will raise a `ValueError`.

The full set of format codes supported varies across platforms, because Python calls the platform C library's **`strftime()`** function, and platform variations are common. To see the full set of format codes supported on your platform, consult the *strftime(3)* *[https://manpages.debian.org/strftime(3)]* documentation. There are also differences between platforms in handling of unsupported format specifiers.

Added in version 3.6: `%G`, `%u` and `%V` were added.

Added in version 3.12: `%:z` was added.

Technical Detail

Broadly speaking, `d.strftime(fmt)` acts like the `time` module's `time.strftime(fmt, d.timetuple())` although not all objects support a `timetuple()` method.

For the `datetime.strptime()` and `date.strptime()` class methods, the default value is `1900-01-01T00:00:00.000`: any components not specified in the format string will be pulled from the default value.

Note

When used to parse partial dates lacking a year, `datetime.strptime()` and `date.strptime()` will raise when encountering February 29 because the default year of 1900 is *not* a leap year. Always add a default leap year to partial date strings before parsing.

```
>>> from datetime import datetime
>>> value = "2/29"
>>> datetime.strptime(value, "%m/%d")
Traceback (most recent call last):
...
ValueError: day 29 must be in range 1..28 for month 2 in year
1900
>>> datetime.strptime(f"1904 {value}", "%Y %m/%d")
datetime.datetime(1904, 2, 29, 0, 0)
```

Using `datetime.strptime(date_string, format)` is equivalent to:

```
datetime(*(time.strptime(date_string, format)[0:6]))
```

except when the format includes sub-second components or time zone offset information, which are supported in `datetime.strptime` but are discarded by `time.strptime`.

For `time` objects, the format codes for year, month, and day should not be used, as **`time`** objects have no such values. If they're used anyway, 1900 is substituted for the year, and 1 for the month and day.

For `date` objects, the format codes for hours, minutes, seconds, and microseconds should not be used, as `date` objects have no such values. If they're used anyway, 0 is substituted for them.

For the same reason, handling of format strings containing Unicode code points that can't be represented in the charset of the current locale is also platform-dependent. On some platforms such code points are preserved intact in the output, while on others `strftime` may raise `UnicodeError` or return an empty string instead.

Notes:

1. Because the format depends on the current locale, care should be taken when making assumptions about the

output value. Field orderings will vary (for example, “month/day/year” versus “day/month/year”), and the output may contain non-ASCII characters.

2. The `strptime()` method can parse years in the full [1, 9999] range, but years < 1000 must be zero-filled to 4-digit width.

   Changed in version 3.2: In previous versions, `strftime()` method was restricted to years >= 1900.

   Changed in version 3.3: In version 3.2, `strftime()` method was restricted to years >= 1000.

3. When used with the `strptime()` method, the `%p` directive only affects the output hour field if the `%I` directive is used to parse the hour.

4. Unlike the `time` module, the **`datetime`** module does not support leap seconds.

5. When used with the `strptime()` method, the `%f` directive accepts from one to six digits and zero pads on the right. `%f` is an extension to the set of format characters in the C standard (but implemented separately in datetime objects, and therefore always available).

6. For a naive object, the `%z`, `%:z` and `%Z` format codes are replaced by empty strings.

   For an aware object:

   `%z`

      `utcoffset()` is transformed into a string of the form `±HHMM[SS[.ffffff]]`, where `HH` is a 2-digit string giving the number of UTC offset hours, `MM` is a 2-digit string giving the number of UTC offset minutes, `SS` is a 2-

digit string giving the number of UTC offset seconds and `ffffff` is a 6-digit string giving the number of UTC offset microseconds. The `ffffff` part is omitted when the offset is a whole number of seconds and both the `ffffff` and the `SS` part is omitted when the offset is a whole number of minutes. For example, if **`utcoffset()`** returns `timedelta(hours=-3, minutes=-30)`, `%z` is replaced with the string `'-0330'`.

Changed in version 3.7: The UTC offset is not restricted to a whole number of minutes.

Changed in version 3.7: When the `%z` directive is provided to the **`strptime()`** method, the UTC offsets can have a colon as a separator between hours, minutes and seconds. For example, `'+01:00:00'` will be parsed as an offset of one hour. In addition, providing `'Z'` is identical to `'+00:00'`.

`%:z`

Behaves exactly as `%z`, but has a colon separator added between hours, minutes and seconds.

`%Z`

In **`strftime()`**, `%Z` is replaced by an empty string if **`tzname()`** returns `None`; otherwise `%Z` is replaced by the returned value, which must be a string.

`strptime()` only accepts certain values for `%Z`:

1. any value in `time.tzname` for your machine's locale
2. the hard-coded values `UTC` and `GMT`

So someone living in Japan may have `JST`, `UTC`, and `GMT` as valid values, but probably not `EST`. It will raise `ValueError` for invalid values.

Changed in version 3.2: When the `%z` directive is provided to the **`strptime()`** method, an aware **`datetime`** object will be produced. The `tzinfo` of the result will be set to a **`timezone`** instance.

7. When used with the **`strptime()`** method, `%U` and `%W` are only used in calculations when the day of the week and the calendar year (`%Y`) are specified.

8. Similar to `%U` and `%W`, `%V` is only used in calculations when the day of the week and the ISO year (`%G`) are specified in a **`strptime()`** format string. Also note that `%G` and `%Y` are not interchangeable.

9. When used with the **`strptime()`** method, the leading zero is optional for formats `%d`, `%m`, `%H`, `%I`, `%M`, `%S`, `%j`, `%U`, `%W`, and `%V`. Format `%y` does require a leading zero.

10. When parsing a month and day using **`strptime()`**, always include a year in the format. If the value you need to parse lacks a year, append an explicit dummy leap year. Otherwise your code will raise an exception when it encounters leap day because the default year used by the parser (1900) is not a leap year. Users run into that bug every leap year.

    ```
    >>> month_day = "02/29"
    >>> datetime.strptime(f"{month_day};1984", "%m/%d;%Y")  #
    No leap year bug.
    datetime.datetime(1984, 2, 29, 0, 0)
    ```

    Deprecated since version 3.13, will be removed in version 3.15: **`strptime()`** calls using a format string containing a day of month without a year now emit a **`DeprecationWarning`**. In 3.15 or later we may change this into an error or change the default year to a leap year. See gh-70647 [https://github.com/python/cpython/issues/70647].

Footnotes

[1]

If, that is, we ignore the effects of Relativity

[2]

This matches the definition of the “proleptic Gregorian” calendar in Dershowitz and Reingold’s book *Calendrical Calculations*, where it’s the base calendar for all computations. See the book for algorithms for converting between proleptic Gregorian ordinals and many other calendar systems.

[3]

See R. H. van Gent’s guide to the mathematics of the ISO 8601 calendar [https://web.archive.org/web/20220531051136/https://webspace.science.uu.nl/~gent0113/calendar/isocalendar.htm] for a good explanation.

`zoneinfo` — IANA time zone support

Added in version 3.9.

Source code: Lib/zoneinfo [https://github.com/python/cpython/tree/3.14/Lib/zoneinfo]

The **`zoneinfo`** module provides a concrete time zone implementation to support the IANA time zone database as originally specified in **PEP 615** [https://peps.python.org/pep-0615/]. By default, **`zoneinfo`** uses the system's time zone data if available; if no system time zone data is available, the library will fall back to using the first-party tzdata [https://pypi.org/project/tzdata/] package available on PyPI.

See also

Module: `datetime`
Provides the **`time`** and **`datetime`** types with which the **`ZoneInfo`** class is designed to be used.

Package tzdata [https://pypi.org/project/tzdata/]
First-party package maintained by the CPython core developers to supply time zone data via PyPI.

Availability: not WASI.

This module does not work or is not available on WebAssembly. See WebAssembly platforms for more information.

Using `ZoneInfo`

`ZoneInfo` is a concrete implementation of the `datetime.tzinfo` abstract base class, and is intended to be attached to `tzinfo`, either via the constructor, the `datetime.replace` method or `datetime.astimezone`:

```
>>> from zoneinfo import ZoneInfo
>>> from datetime import datetime, timedelta

>>> dt = datetime(2020, 10, 31, 12,
tzinfo=ZoneInfo("America/Los_Angeles"))
>>> print(dt)
2020-10-31 12:00:00-07:00

>>> dt.tzname()
'PDT'
```

Datetimes constructed in this way are compatible with datetime arithmetic and handle daylight saving time transitions with no further intervention:

```
>>> dt_add = dt + timedelta(days=1)

>>> print(dt_add)
2020-11-01 12:00:00-08:00

>>> dt_add.tzname()
'PST'
```

These time zones also support the `fold` attribute introduced in **PEP 495** [https://peps.python.org/pep-0495/]. During offset transitions which induce ambiguous times (such as a daylight saving time to standard time transition), the offset from *before* the transition is used when `fold=0`, and the offset *after* the transition is used when `fold=1`, for example:

```
>>> dt = datetime(2020, 11, 1, 1,
tzinfo=ZoneInfo("America/Los_Angeles"))
```

```
>>> print(dt)
2020-11-01 01:00:00-07:00

>>> print(dt.replace(fold=1))
2020-11-01 01:00:00-08:00
```

When converting from another time zone, the fold will be set to the correct value:

```
>>> from datetime import timezone
>>> LOS_ANGELES = ZoneInfo("America/Los_Angeles")
>>> dt_utc = datetime(2020, 11, 1, 8, tzinfo=timezone.utc)

>>> # Before the PDT -> PST transition
>>> print(dt_utc.astimezone(LOS_ANGELES))
2020-11-01 01:00:00-07:00

>>> # After the PDT -> PST transition
>>> print((dt_utc +
timedelta(hours=1)).astimezone(LOS_ANGELES))
2020-11-01 01:00:00-08:00
```

Data sources

The `zoneinfo` module does not directly provide time zone data, and instead pulls time zone information from the system time zone database or the first-party PyPI package tzdata [https://pypi.org/project/tzdata/], if available. Some systems, including notably Windows systems, do not have an IANA database available, and so for projects targeting cross-platform compatibility that require time zone data, it is recommended to declare a dependency on tzdata. If neither system data nor tzdata are available, all calls to **`ZoneInfo`** will raise **`ZoneInfoNotFoundError`**.

Configuring the data sources

When `ZoneInfo(key)` is called, the constructor first searches the directories specified in **`TZPATH`** for a file matching `key`, and on failure looks for a match in the tzdata package. This behavior can be configured in three ways:

1. The default **`TZPATH`** when not otherwise specified can be configured at compile time.
2. **`TZPATH`** can be configured using an environment variable.
3. At runtime, the search path can be manipulated using the **`reset_tzpath()`** function.

Compile-time configuration

The default **`TZPATH`** includes several common deployment locations for the time zone database (except on Windows, where there are no “well-known” locations for time zone data). On POSIX systems, downstream distributors and those building Python from source who know where their system time zone data is deployed may change the default time zone path by specifying the compile-time option `TZPATH` (or, more likely, the **`configure flag --with-tzpath`**), which should be a string delimited by **`os.pathsep`**.

On all platforms, the configured value is available as the `TZPATH` key in **`sysconfig.get_config_var()`**.

Environment configuration

When initializing **`TZPATH`** (either at import time or whenever **`reset_tzpath()`** is called with no arguments), the `zoneinfo` module will use the environment variable `PYTHONTZPATH`, if it exists, to set the search path.

PYTHONTZPATH

This is an `os.pathsep`-separated string containing the time zone search path to use. It must consist of only absolute rather than relative paths. Relative components specified in `PYTHONTZPATH` will not be used, but otherwise the behavior when a relative path is specified is implementation-defined; CPython will raise `InvalidTZPathWarning`, but other implementations are free to silently ignore the erroneous component or raise an exception.

To set the system to ignore the system data and use the tzdata package instead, set `PYTHONTZPATH=""`.

Runtime configuration

The TZ search path can also be configured at runtime using the `reset_tzpath()` function. This is generally not an advisable operation, though it is reasonable to use it in test functions that require the use of a specific time zone path (or require disabling access to the system time zones).

The `ZoneInfo` class

class zoneinfo.ZoneInfo(*key*)

A concrete `datetime.tzinfo` subclass that represents an IANA time zone specified by the string `key`. Calls to the primary constructor will always return objects that compare identically; put another way, barring cache invalidation via `ZoneInfo.clear_cache()`, for all values of `key`, the following assertion will always be true:

```
a = ZoneInfo(key)
b = ZoneInfo(key)
assert a is b
```

`key` must be in the form of a relative, normalized POSIX path, with no up-level references. The constructor will raise `ValueError` if a non-conforming key is passed.

If no file matching `key` is found, the constructor will raise `ZoneInfoNotFoundError`.

The `ZoneInfo` class has two alternate constructors:

classmethod ZoneInfo.from_file(*file_obj*, */*, *key=None*)

Constructs a `ZoneInfo` object from a file-like object returning bytes (e.g. a file opened in binary mode or an `io.BytesIO` object). Unlike the primary constructor, this always constructs a new object.

The `key` parameter sets the name of the zone for the purposes of `__str__()` and `__repr__()`.

Objects created via this constructor cannot be pickled (see pickling).

`ValueError` is raised if the data read from *file_obj* is not a valid TZif file.

classmethod ZoneInfo.no_cache(*key*)

An alternate constructor that bypasses the constructor’s cache. It is identical to the primary constructor, but returns a new object on each call. This is most likely to be useful for testing or demonstration purposes, but it can also be used to create a system with a different cache invalidation strategy.

Objects created via this constructor will also bypass the cache of a deserializing process when unpickled.

Caution

Using this constructor may change the semantics of your datetimes in surprising ways, only use it if you know that you need to.

The following class methods are also available:

classmethod ZoneInfo.clear_cache(*, *only_keys=None*)

A method for invalidating the cache on the `ZoneInfo` class. If no arguments are passed, all caches are invalidated and the next call to the primary constructor for each key will return a new instance.

If an iterable of key names is passed to the `only_keys` parameter, only the specified keys will be removed from the cache. Keys passed to `only_keys` but not found in the cache are ignored.

Warning

Invoking this function may change the semantics of datetimes using `ZoneInfo` in surprising ways; this modifies module state and thus may have wide-ranging effects. Only use it if you know that you need to.

The class has one attribute:

ZoneInfo.key

This is a read-only attribute that returns the value of `key` passed to the constructor, which should be a lookup key in the IANA time zone database (e.g. `America/New_York`, `Europe/Paris` or `Asia/Tokyo`).

For zones constructed from file without specifying a `key` parameter, this will be set to `None`.

Note

Although it is a somewhat common practice to expose these to end users, these values are designed to be primary keys for representing the relevant zones and not necessarily user-facing elements. Projects like CLDR (the Unicode Common Locale Data Repository) can be used to get more user-friendly strings from these keys.

String representations

The string representation returned when calling `str` on a `ZoneInfo` object defaults to using the `ZoneInfo.key` attribute (see the note on usage in the attribute documentation):

```
>>> zone = ZoneInfo("Pacific/Kwajalein")
>>> str(zone)
'Pacific/Kwajalein'

>>> dt = datetime(2020, 4, 1, 3, 15, tzinfo=zone)
>>> f"{dt.isoformat()} [{dt.tzinfo}]"
'2020-04-01T03:15:00+12:00 [Pacific/Kwajalein]'
```

For objects constructed from a file without specifying a `key` parameter, `str` falls back to calling `repr()`. `ZoneInfo`’s `repr` is implementation-defined and not necessarily stable between versions, but it is guaranteed not to be a valid `ZoneInfo` key.

Pickle serialization

Rather than serializing all transition data, `ZoneInfo` objects are serialized by key, and `ZoneInfo` objects constructed from files (even those with a value for `key` specified) cannot be pickled.

The behavior of a `ZoneInfo` file depends on how it was constructed:

1. `ZoneInfo(key)`: When constructed with the primary constructor, a `ZoneInfo` object is serialized by key, and when deserialized, the deserializing process uses the primary and thus it is expected that these are the same object as other references to the same time zone. For example, if `europe_berlin_pkl` is a string containing a pickle constructed from `ZoneInfo("Europe/Berlin")`, one would expect the following behavior:

   ```
   >>> a = ZoneInfo("Europe/Berlin")
   >>> b = pickle.loads(europe_berlin_pkl)
   >>> a is b
   True
   ```

2. `ZoneInfo.no_cache(key)`: When constructed from the cache-bypassing constructor, the `ZoneInfo` object is also serialized by key, but when deserialized, the deserializing process uses the cache bypassing constructor. If `europe_berlin_pkl_nc` is a string containing a pickle constructed from `ZoneInfo.no_cache("Europe/Berlin")`, one would expect the following behavior:

   ```
   >>> a = ZoneInfo("Europe/Berlin")
   >>> b = pickle.loads(europe_berlin_pkl_nc)
   >>> a is b
   False
   ```

3. `ZoneInfo.from_file(file_obj, /, key=None)`: When constructed from a file, the `ZoneInfo` object raises an exception on pickling. If an end user wants to pickle a `ZoneInfo` constructed from a file, it is recommended that they use a wrapper type or a custom serialization function: either serializing by key or storing the contents of the file object and serializing that.

This method of serialization requires that the time zone data for the required key be available on both the serializing and deserializing side, similar to the way that references to classes and functions are expected to exist in both the serializing and deserializing environments. It also means that no guarantees are made about the consistency of results when unpickling a `ZoneInfo` pickled in an environment with a different version of the time zone data.

Functions

zoneinfo.available_timezones()

Get a set containing all the valid keys for IANA time zones available anywhere on the time zone path. This is recalculated on every call to the function.

This function only includes canonical zone names and does not include “special” zones such as those under the `posix/` and `right/` directories, or the `posixrules` zone.

Caution

This function may open a large number of files, as the best way to determine if a file on the time zone path is a valid time zone is to read the “magic string” at the beginning.

Note

These values are not designed to be exposed to end-users; for user facing elements, applications should use something like CLDR (the Unicode Common Locale Data Repository) to get more user-friendly strings. See also the cautionary note on `ZoneInfo.key`.

zoneinfo.reset_tzpath(*to=None*)

Sets or resets the time zone search path (**`TZPATH`**) for the module. When called with no arguments, **`TZPATH`** is set to the default value.

Calling `reset_tzpath` will not invalidate the **`ZoneInfo`** cache, and so calls to the primary `ZoneInfo` constructor will only use the new `TZPATH` in the case of a cache miss.

The `to` parameter must be a sequence of strings or **`os.PathLike`** and not a string, all of which must be absolute paths. **`ValueError`** will be raised if something other than an absolute path is passed.

Globals

zoneinfo.TZPATH

A read-only sequence representing the time zone search path – when constructing a `ZoneInfo` from a key, the key is joined to each entry in the `TZPATH`, and the first file found is used.

`TZPATH` may contain only absolute paths, never relative paths, regardless of how it is configured.

The object that `zoneinfo.TZPATH` points to may change in response to a call to **`reset_tzpath()`**, so it is recommended to use `zoneinfo.TZPATH` rather than importing `TZPATH` from `zoneinfo` or assigning a long-lived variable to `zoneinfo.TZPATH`.

For more information on configuring the time zone search path, see Configuring the data sources.

Exceptions and warnings

exception zoneinfo.ZoneInfoNotFoundError

Raised when construction of a `ZoneInfo` object fails because the specified key could not be found on the system. This is a subclass of `KeyError`.

exception zoneinfo.InvalidTZPathWarning

Raised when `PYTHONTZPATH` contains an invalid component that will be filtered out, such as a relative path.

`calendar` — General calendar-related functions

Source code: Lib/calendar.py
[https://github.com/python/cpython/tree/3.14/Lib/calendar.py]

This module allows you to output calendars like the Unix **cal** program, and provides additional useful functions related to the calendar. By default, these calendars have Monday as the first day of the week, and Sunday as the last (the European convention). Use `setfirstweekday()` to set the first day of the week to Sunday (6) or to any other weekday. Parameters that specify dates are given as integers. For related functionality, see also the `datetime` and `time` modules.

The functions and classes defined in this module use an idealized calendar, the current Gregorian calendar extended indefinitely in both directions. This matches the definition of the “proleptic Gregorian” calendar in Dershowitz and Reingold’s book “Calendrical Calculations”, where it’s the base calendar for all computations. Zero and negative years are interpreted as prescribed by the ISO 8601 standard. Year 0 is 1 BC, year -1 is 2 BC, and so on.

class calendar.Calendar(*firstweekday=0*)

Creates a `Calendar` object. *firstweekday* is an integer specifying the first day of the week. `MONDAY` is `0` (the default), `SUNDAY` is `6`.

A `Calendar` object provides several methods that can be used for preparing the calendar data for formatting. This class doesn’t do any formatting itself. This is the job of subclasses.

`Calendar` instances have the following methods and attributes:

firstweekday

> The first weekday as an integer (0–6).
>
> This property can also be set and read using `setfirstweekday()` and `getfirstweekday()` respectively.

getfirstweekday()

> Return an `int` for the current first weekday (0–6).
>
> Identical to reading the `firstweekday` property.

setfirstweekday(*firstweekday*)

> Set the first weekday to *firstweekday*, passed as an `int` (0–6)
>
> Identical to setting the `firstweekday` property.

iterweekdays()

> Return an iterator for the week day numbers that will be used for one week. The first value from the iterator will be the same as the value of the `firstweekday` property.

itermonthdates(*year*, *month*)

> Return an iterator for the month *month* (1–12) in the year *year*. This iterator will return all days (as `datetime.date` objects) for the month and all days before the start of the month or after the end of the month that are required to get a complete week.

itermonthdays(*year*, *month*)

Return an iterator for the month *month* in the year *year* similar to `itermonthdates()`, but not restricted by the `datetime.date` range. Days returned will simply be day of the month numbers. For the days outside of the specified month, the day number is `0`.

itermonthdays2(*year*, *month*)

Return an iterator for the month *month* in the year *year* similar to `itermonthdates()`, but not restricted by the `datetime.date` range. Days returned will be tuples consisting of a day of the month number and a week day number.

itermonthdays3(*year*, *month*)

Return an iterator for the month *month* in the year *year* similar to `itermonthdates()`, but not restricted by the `datetime.date` range. Days returned will be tuples consisting of a year, a month and a day of the month numbers.

Added in version 3.7.

itermonthdays4(*year*, *month*)

Return an iterator for the month *month* in the year *year* similar to `itermonthdates()`, but not restricted by the `datetime.date` range. Days returned will be tuples consisting of a year, a month, a day of the month, and a day of the week numbers.

Added in version 3.7.

monthdatescalendar(*year*, *month*)

Return a list of the weeks in the month *month* of the *year* as full weeks. Weeks are lists of seven `datetime.date` objects.

monthdays2calendar(*year*, *month*)

Return a list of the weeks in the month *month* of the *year* as full weeks. Weeks are lists of seven tuples of day numbers and weekday numbers.

monthdayscalendar(*year*, *month*)

Return a list of the weeks in the month *month* of the *year* as full weeks. Weeks are lists of seven day numbers.

yeardatescalendar(*year*, *width=3*)

Return the data for the specified year ready for formatting. The return value is a list of month rows. Each month row contains up to *width* months (defaulting to 3). Each month contains between 4 and 6 weeks and each week contains 1–7 days. Days are `datetime.date` objects.

yeardays2calendar(*year*, *width=3*)

Return the data for the specified year ready for formatting (similar to `yeardatescalendar()`). Entries in the week lists are tuples of day numbers and weekday numbers. Day numbers outside this month are zero.

yeardayscalendar(*year*, *width=3*)

Return the data for the specified year ready for formatting (similar to `yeardatescalendar()`). Entries in the week lists are day numbers. Day numbers outside this month are zero.

class calendar.TextCalendar(*firstweekday=0*)

This class can be used to generate plain text calendars.

`TextCalendar` instances have the following methods:

formatday(*theday*, *weekday*, *width*)

Return a string representing a single day formatted with the given *width*. If *theday* is `0`, return a string of spaces of the specified width, representing an empty day. The *weekday* parameter is unused.

formatweek(*theweek*, *w=0*)

Return a single week in a string with no newline. If *w* is provided, it specifies the width of the date columns, which are centered. Depends on the first weekday as specified in the constructor or set by the `setfirstweekday()` method.

formatweekday(*weekday*, *width*)

Return a string representing the name of a single weekday formatted to the specified *width*. The *weekday* parameter is an integer representing the day of the week, where `0` is Monday and `6` is Sunday.

formatweekheader(*width*)

Return a string containing the header row of weekday names, formatted with the given *width* for each column. The names depend on the locale settings and are padded to the specified width.

formatmonth(*theyear*, *themonth*, *w=0*, *l=0*)

Return a month’s calendar in a multi-line string. If *w* is provided, it specifies the width of the date columns, which are centered. If *l* is given, it specifies the number of lines that each week will use. Depends on the first weekday as specified in the constructor or set by the `setfirstweekday()` method.

formatmonthname(*theyear*, *themonth*, *width=0*, *withyear=True*)

Return a string representing the month's name centered within the specified *width*. If *withyear* is `True`, include the year in the output. The *theyear* and *themonth* parameters specify the year and month for the name to be formatted respectively.

prmonth(*theyear*, *themonth*, *w=0*, *l=0*)

Print a month's calendar as returned by `formatmonth()`.

formatyear(*theyear*, *w=2*, *l=1*, *c=6*, *m=3*)

Return a *m*-column calendar for an entire year as a multi-line string. Optional parameters *w*, *l*, and *c* are for date column width, lines per week, and number of spaces between month columns, respectively. Depends on the first weekday as specified in the constructor or set by the `setfirstweekday()` method. The earliest year for which a calendar can be generated is platform-dependent.

pryear(*theyear*, *w=2*, *l=1*, *c=6*, *m=3*)

Print the calendar for an entire year as returned by `formatyear()`.

class calendar.HTMLCalendar(*firstweekday=0*)

This class can be used to generate HTML calendars.

`HTMLCalendar` instances have the following methods:

formatmonth(*theyear*, *themonth*, *withyear=True*)

Return a month's calendar as an HTML table. If *withyear* is true the year will be included in the header, otherwise just the month name will be used.

formatyear(*theyear*, *width=3*)

Return a year’s calendar as an HTML table. *width* (defaulting to 3) specifies the number of months per row.

formatyearpage(*theyear*, *width=3*, *css='calendar.css'*, *encoding=None*)

Return a year’s calendar as a complete HTML page. *width* (defaulting to 3) specifies the number of months per row. *css* is the name for the cascading style sheet to be used. `None` can be passed if no style sheet should be used. *encoding* specifies the encoding to be used for the output (defaulting to the system default encoding).

formatmonthname(*theyear*, *themonth*, *withyear=True*)

Return a month name as an HTML table row. If *withyear* is true the year will be included in the row, otherwise just the month name will be used.

`HTMLCalendar` has the following attributes you can override to customize the CSS classes used by the calendar:

cssclasses

A list of CSS classes used for each weekday. The default class list is:

```
cssclasses = ["mon", "tue", "wed", "thu", "fri", "sat",
"sun"]
```

more styles can be added for each day:

```
cssclasses = ["mon text-bold", "tue", "wed", "thu",
"fri", "sat", "sun red"]
```

Note that the length of this list must be seven items.

cssclass_noday

The CSS class for a weekday occurring in the previous or coming month.

Added in version 3.7.

cssclasses_weekday_head

A list of CSS classes used for weekday names in the header row. The default is the same as `cssclasses`.

Added in version 3.7.

cssclass_month_head

The month's head CSS class (used by `formatmonthname()`). The default value is `"month"`.

Added in version 3.7.

cssclass_month

The CSS class for the whole month's table (used by `formatmonth()`). The default value is `"month"`.

Added in version 3.7.

cssclass_year

The CSS class for the whole year's table of tables (used by `formatyear()`). The default value is `"year"`.

Added in version 3.7.

cssclass_year_head

The CSS class for the table head for the whole year (used by `formatyear()`). The default value is `"year"`.

Added in version 3.7.

Note that although the naming for the above described class attributes is singular (e.g. `cssclass_month` `cssclass_noday`), one can replace the single CSS class with a space separated list of CSS classes, for example:

```
"text-bold text-red"
```

Here is an example how **`HTMLCalendar`** can be customized:

```
class CustomHTMLCal(calendar.HTMLCalendar):
    cssclasses = [style + " text-nowrap" for style in
                  calendar.HTMLCalendar.cssclasses]
    cssclass_month_head = "text-center month-head"
    cssclass_month = "text-center month"
    cssclass_year = "text-italic lead"
```

class calendar.LocaleTextCalendar(*firstweekday=0*, *locale=None*)

This subclass of **`TextCalendar`** can be passed a locale name in the constructor and will return month and weekday names in the specified locale.

class calendar.LocaleHTMLCalendar(*firstweekday=0*, *locale=None*)

This subclass of **`HTMLCalendar`** can be passed a locale name in the constructor and will return month and weekday names in the specified locale.

Note

The constructor, **`formatweekday()`** and **`formatmonthname()`** methods of these two classes temporarily change the `LC_TIME` locale to the given *locale*. Because the current locale is a process-wide setting, they are not thread-safe.

For simple text calendars this module provides the following functions.

calendar.setfirstweekday(*weekday*)

Sets the weekday (`0` is Monday, `6` is Sunday) to start each week. The values `MONDAY`, `TUESDAY`, `WEDNESDAY`, `THURSDAY`, `FRIDAY`, `SATURDAY`, and `SUNDAY` are provided for convenience. For example, to set the first weekday to Sunday:

```
import calendar
calendar.setfirstweekday(calendar.SUNDAY)
```

calendar.firstweekday()

Returns the current setting for the weekday to start each week.

calendar.isleap(*year*)

Returns `True` if *year* is a leap year, otherwise `False`.

calendar.leapdays(*y1*, *y2*)

Returns the number of leap years in the range from *y1* to *y2* (exclusive), where *y1* and *y2* are years.

This function works for ranges spanning a century change.

calendar.weekday(*year*, *month*, *day*)

Returns the day of the week (`0` is Monday) for *year* (`1970`–...), *month* (`1`–`12`), *day* (`1`–`31`).

calendar.weekheader(*n*)

Return a header containing abbreviated weekday names. *n* specifies the width in characters for one weekday.

calendar.monthrange(*year*, *month*)

Returns weekday of first day of the month and number of days in month, for the specified *year* and *month*.

calendar.monthcalendar(*year*, *month*)

Returns a matrix representing a month's calendar. Each row represents a week; days outside of the month are represented by zeros. Each week begins with Monday unless set by `setfirstweekday()`.

calendar.prmonth(*theyear*, *themonth*, *w=0*, *l=0*)

Prints a month's calendar as returned by `month()`.

calendar.month(*theyear*, *themonth*, *w=0*, *l=0*)

Returns a month's calendar in a multi-line string using the `formatmonth()` of the `TextCalendar` class.

calendar.prcal(*year*, *w=0*, *l=0*, *c=6*, *m=3*)

Prints the calendar for an entire year as returned by `calendar()`.

calendar.calendar(*year*, *w=2*, *l=1*, *c=6*, *m=3*)

Returns a 3-column calendar for an entire year as a multi-line string using the `formatyear()` of the `TextCalendar` class.

calendar.timegm(*tuple*)

An unrelated but handy function that takes a time tuple such as returned by the `gmtime()` function in the `time` module, and returns the corresponding Unix timestamp value, assuming an epoch of 1970, and the POSIX encoding. In fact, `time.gmtime()` and `timegm()` are each others' inverse.

The `calendar` module exports the following data attributes:

calendar.day_name

A sequence that represents the days of the week in the current locale, where Monday is day number 0.

```
>>> import calendar
>>> list(calendar.day_name)
['Monday', 'Tuesday', 'Wednesday', 'Thursday', 'Friday',
'Saturday', 'Sunday']
```

calendar.day_abbr

A sequence that represents the abbreviated days of the week in the current locale, where Mon is day number 0.

```
>>> import calendar
>>> list(calendar.day_abbr)
['Mon', 'Tue', 'Wed', 'Thu', 'Fri', 'Sat', 'Sun']
```

calendar.MONDAY
calendar.TUESDAY
calendar.WEDNESDAY
calendar.THURSDAY
calendar.FRIDAY
calendar.SATURDAY
calendar.SUNDAY

Aliases for the days of the week, where `MONDAY` is `0` and `SUNDAY` is `6`.

Added in version 3.12.

class calendar.Day

Enumeration defining days of the week as integer constants. The members of this enumeration are exported to the module scope as **`MONDAY`** through **`SUNDAY`**.

Added in version 3.12.

calendar.month_name

A sequence that represents the months of the year in the current locale. This follows normal convention of January being month number 1, so it has a length of 13 and `month_name[0]` is the empty string.

```
>>> import calendar
>>> list(calendar.month_name)
['', 'January', 'February', 'March', 'April', 'May', 'June',
'July', 'August', 'September', 'October', 'November',
'December']
```

calendar.month_abbr

A sequence that represents the abbreviated months of the year in the current locale. This follows normal convention of January being month number 1, so it has a length of 13 and `month_abbr[0]` is the empty string.

```
>>> import calendar
>>> list(calendar.month_abbr)
['', 'Jan', 'Feb', 'Mar', 'Apr', 'May', 'Jun', 'Jul', 'Aug',
'Sep', 'Oct', 'Nov', 'Dec']
```

calendar.JANUARY
calendar.FEBRUARY
calendar.MARCH
calendar.APRIL
calendar.MAY
calendar.JUNE
calendar.JULY
calendar.AUGUST
calendar.SEPTEMBER
calendar.OCTOBER
calendar.NOVEMBER
calendar.DECEMBER

Aliases for the months of the year, where `JANUARY` is `1` and `DECEMBER` is `12`.

Added in version 3.12.

class calendar.Month

Enumeration defining months of the year as integer constants. The members of this enumeration are exported to the module scope as `JANUARY` through `DECEMBER`.

Added in version 3.12.

The `calendar` module defines the following exceptions:

exception calendar.IllegalMonthError(*month*)

A subclass of `ValueError`, raised when the given month number is outside of the range 1-12 (inclusive).

month

The invalid month number.

exception calendar.IllegalWeekdayError(*weekday*)

A subclass of `ValueError`, raised when the given weekday number is outside of the range 0-6 (inclusive).

weekday

The invalid weekday number.

See also

Module `datetime`

Object-oriented interface to dates and times with similar functionality to the `time` module.

Module `time`

Low-level time related functions.

Command-line usage

Added in version 2.5.

The `calendar` module can be executed as a script from the command line to interactively print a calendar.

```
python -m calendar [-h] [-L LOCALE] [-e ENCODING] [-t
{text,html}]
                   [-w WIDTH] [-l LINES] [-s SPACING] [-m
MONTHS] [-c CSS]
                   [-f FIRST_WEEKDAY] [year] [month]
```

For example, to print a calendar for the year 2000:

```
$ python -m calendar 2000
                                  2000

      January                   February
March
Mo Tu We Th Fr Sa Su      Mo Tu We Th Fr Sa Su      Mo Tu We Th
Fr Sa Su
                1  2          1  2  3  4  5  6             1  2
3  4  5
 3  4  5  6  7  8  9       7  8  9 10 11 12 13       6  7  8  9
10 11 12
10 11 12 13 14 15 16      14 15 16 17 18 19 20      13 14 15 16
17 18 19
17 18 19 20 21 22 23      21 22 23 24 25 26 27      20 21 22 23
24 25 26
24 25 26 27 28 29 30      28 29                     27 28 29 30
31
31

       April                      May
June
Mo Tu We Th Fr Sa Su      Mo Tu We Th Fr Sa Su      Mo Tu We Th
Fr Sa Su
                1  2       1  2  3  4  5  6  7                1
2  3  4
 3  4  5  6  7  8  9       8  9 10 11 12 13 14       5  6  7  8
9 10 11
10 11 12 13 14 15 16      15 16 17 18 19 20 21      12 13 14 15
16 17 18
```

```
17 18 19 20 21 22 23      22 23 24 25 26 27 28      19 20 21 22
23 24 25
24 25 26 27 28 29 30      29 30 31                  26 27 28 29
30

        July                     August
September
Mo Tu We Th Fr Sa Su      Mo Tu We Th Fr Sa Su      Mo Tu We Th
Fr Sa Su
                1  2          1  2  3  4  5  6
1  2  3
 3  4  5  6  7  8  9       7  8  9 10 11 12 13       4  5  6  7
8  9 10
10 11 12 13 14 15 16      14 15 16 17 18 19 20      11 12 13 14
15 16 17
17 18 19 20 21 22 23      21 22 23 24 25 26 27      18 19 20 21
22 23 24
24 25 26 27 28 29 30      28 29 30 31               25 26 27 28
29 30
31

      October                  November
December
Mo Tu We Th Fr Sa Su      Mo Tu We Th Fr Sa Su      Mo Tu We Th
Fr Sa Su
                   1             1  2  3  4  5
1  2  3
 2  3  4  5  6  7  8       6  7  8  9 10 11 12       4  5  6  7
8  9 10
 9 10 11 12 13 14 15      13 14 15 16 17 18 19      11 12 13 14
15 16 17
16 17 18 19 20 21 22      20 21 22 23 24 25 26      18 19 20 21
22 23 24
23 24 25 26 27 28 29      27 28 29 30               25 26 27 28
29 30 31
30 31
```

The following options are accepted:

--help, -h
    Show the help message and exit.

--locale LOCALE, -L LOCALE

The locale to use for month and weekday names. Defaults to English.

--encoding ENCODING, -e ENCODING

The encoding to use for output. `--encoding` is required if `--locale` is set.

--type {text,html}, -t {text,html}

Print the calendar to the terminal as text, or as an HTML document.

--first-weekday FIRST_WEEKDAY, -f FIRST_WEEKDAY

The weekday to start each week. Must be a number between 0 (Monday) and 6 (Sunday). Defaults to 0.

Added in version 3.13.

year

The year to print the calendar for. Defaults to the current year.

month

The month of the specified `year` to print the calendar for. Must be a number between 1 and 12, and may only be used in text mode. Defaults to printing a calendar for the full year.

Text-mode options:

--width WIDTH, -w WIDTH

The width of the date column in terminal columns. The date is printed centred in the column. Any value lower than 2 is ignored. Defaults to 2.

--lines LINES, -l LINES

The number of lines for each week in terminal rows. The date is printed top-aligned. Any value lower than 1 is ignored. Defaults to 1.

--spacing SPACING, -s SPACING

The space between months in columns. Any value lower than 2 is ignored. Defaults to 6.

--months MONTHS, -m MONTHS

The number of months printed per row. Defaults to 3.

Changed in version 3.14: By default, today's date is highlighted in color and can be controlled using environment variables.

HTML-mode options:

--css CSS, -c CSS

The path of a CSS stylesheet to use for the calendar. This must either be relative to the generated HTML, or an absolute HTTP or `file:///` URL.

`collections` — Container datatypes

Source code: Lib/collections/__init__.py
[https://github.com/python/cpython/tree/3.14/Lib/collections/__init__.py]

This module implements specialized container datatypes providing alternatives to Python’s general purpose built-in containers, `dict`, `list`, `set`, and `tuple`.

`namedtuple()`	factory function for creating tuple subclasses with named fields
`deque`	list-like container with fast appends and pops on either end
`ChainMap`	dict-like class for creating a single view of multiple mappings
`Counter`	dict subclass for counting hashable objects
`OrderedDict`	dict subclass that remembers the order entries were added
`defaultdict`	dict subclass that calls a factory function to supply missing values

`UserDict`	wrapper around dictionary objects for easier dict subclassing
`UserList`	wrapper around list objects for easier list subclassing
`UserString`	wrapper around string objects for easier string subclassing

`ChainMap` objects

Added in version 3.3.

A `ChainMap` class is provided for quickly linking a number of mappings so they can be treated as a single unit. It is often much faster than creating a new dictionary and running multiple `update()` calls.

The class can be used to simulate nested scopes and is useful in templating.

class collections.ChainMap(**maps*)

> A `ChainMap` groups multiple dicts or other mappings together to create a single, updateable view. If no *maps* are specified, a single empty dictionary is provided so that a new chain always has at least one mapping.
>
> The underlying mappings are stored in a list. That list is public and can be accessed or updated using the *maps* attribute. There is no other state.

Lookups search the underlying mappings successively until a key is found. In contrast, writes, updates, and deletions only operate on the first mapping.

A **`ChainMap`** incorporates the underlying mappings by reference. So, if one of the underlying mappings gets updated, those changes will be reflected in **`ChainMap`**.

All of the usual dictionary methods are supported. In addition, there is a *maps* attribute, a method for creating new subcontexts, and a property for accessing all but the first mapping:

maps

> A user updateable list of mappings. The list is ordered from first-searched to last-searched. It is the only stored state and can be modified to change which mappings are searched. The list should always contain at least one mapping.

new_child(*m=None*, ***kwargs*)

> Returns a new **`ChainMap`** containing a new map followed by all of the maps in the current instance. If `m` is specified, it becomes the new map at the front of the list of mappings; if not specified, an empty dict is used, so that a call to `d.new_child()` is equivalent to: `ChainMap({}, *d.maps)`. If any keyword arguments are specified, they update passed map or new empty dict. This method is used for creating subcontexts that can be updated without altering values in any of the parent mappings.
>
> *Changed in version 3.4:* The optional `m` parameter was added.

Changed in version 3.10: Keyword arguments support was added.

parents

Property returning a new **`ChainMap`** containing all of the maps in the current instance except the first one. This is useful for skipping the first map in the search. Use cases are similar to those for the **`nonlocal`** keyword used in nested scopes. The use cases also parallel those for the built-in **`super()`** function. A reference to `d.parents` is equivalent to: `ChainMap(*d.maps[1:])`.

Note, the iteration order of a **`ChainMap`** is determined by scanning the mappings last to first:

```
>>> baseline = {'music': 'bach', 'art': 'rembrandt'}
>>> adjustments = {'art': 'van gogh', 'opera': 'carmen'}
>>> list(ChainMap(adjustments, baseline))
['music', 'art', 'opera']
```

This gives the same ordering as a series of **`dict.update()`** calls starting with the last mapping:

```
>>> combined = baseline.copy()
>>> combined.update(adjustments)
>>> list(combined)
['music', 'art', 'opera']
```

Changed in version 3.9: Added support for `|` and `|=` operators, specified in **PEP 584** [https://peps.python.org/pep-0584/].

See also

- The MultiContext class [https://github.com/enthought/codetools/blob/4.0.0/codetools/contexts/multi_context.py] in the Enthought CodeTools package [https://github.com/enthought/codetools] has options to support writing to any mapping in the chain.

- Django's Context class [https://github.com/django/django/blob/main/django/template/context.py] for templating is a read-only chain of mappings. It also features pushing and popping of contexts similar to the `new_child()` method and the `parents` property.
- The Nested Contexts recipe [https://code.activestate.com/recipes/577434-nested-contexts-a-chain-of-mapping-objects/] has options to control whether writes and other mutations apply only to the first mapping or to any mapping in the chain.
- A greatly simplified read-only version of Chainmap [https://code.activestate.com/recipes/305268/].

`ChainMap` Examples and Recipes

This section shows various approaches to working with chained maps.

Example of simulating Python's internal lookup chain:

```
import builtins
pylookup = ChainMap(locals(), globals(), vars(builtins))
```

Example of letting user specified command-line arguments take precedence over environment variables which in turn take precedence over default values:

```
import os, argparse

defaults = {'color': 'red', 'user': 'guest'}

parser = argparse.ArgumentParser()
parser.add_argument('-u', '--user')
parser.add_argument('-c', '--color')
namespace = parser.parse_args()
command_line_args = {k: v for k, v in vars(namespace).items()
if v is not None}
```

```
combined = ChainMap(command_line_args, os.environ, defaults)
print(combined['color'])
print(combined['user'])
```

Example patterns for using the [`ChainMap`] class to simulate nested contexts:

```
c = ChainMap()        # Create root context
d = c.new_child()     # Create nested child context
e = c.new_child()     # Child of c, independent from d
e.maps[0]             # Current context dictionary -- like
Python's locals()
e.maps[-1]            # Root context -- like Python's globals()
e.parents             # Enclosing context chain -- like
Python's nonlocals

d['x'] = 1            # Set value in current context
d['x']                # Get first key in the chain of contexts
del d['x']            # Delete from current context
list(d)               # All nested values
k in d                # Check all nested values
len(d)                # Number of nested values
d.items()             # All nested items
dict(d)               # Flatten into a regular dictionary
```

The `ChainMap` class only makes updates (writes and deletions) to the first mapping in the chain while lookups will search the full chain. However, if deep writes and deletions are desired, it is easy to make a subclass that updates keys found deeper in the chain:

```
class DeepChainMap(ChainMap):
    'Variant of ChainMap that allows direct updates to inner
scopes'

    def __setitem__(self, key, value):
        for mapping in self.maps:
            if key in mapping:
                mapping[key] = value
                return
        self.maps[0][key] = value

    def __delitem__(self, key):
        for mapping in self.maps:
            if key in mapping:
                del mapping[key]
                return
        raise KeyError(key)

>>> d = DeepChainMap({'zebra': 'black'}, {'elephant': 'blue'},
{'lion': 'yellow'})
>>> d['lion'] = 'orange'         # update an existing key two
levels down
>>> d['snake'] = 'red'           # new keys get added to the
topmost dict
>>> del d['elephant']            # remove an existing key one
level down
>>> d                            # display result
DeepChainMap({'zebra': 'black', 'snake': 'red'}, {}, {'lion':
'orange'})
```

`Counter` objects

A counter tool is provided to support convenient and rapid tallies. For example:

```
>>> # Tally occurrences of words in a list
>>> cnt = Counter()
>>> for word in ['red', 'blue', 'red', 'green', 'blue',
'blue']:
...     cnt[word] += 1
...
>>> cnt
Counter({'blue': 3, 'red': 2, 'green': 1})

>>> # Find the ten most common words in Hamlet
>>> import re
>>> words = re.findall(r'\w+',
open('hamlet.txt').read().lower())
>>> Counter(words).most_common(10)
[('the', 1143), ('and', 966), ('to', 762), ('of', 669), ('i',
631),
 ('you', 554),  ('a', 546), ('my', 514), ('hamlet', 471),
('in', 451)]
```

class collections.Counter([*iterable-or-mapping*])

A **`Counter`** is a **`dict`** subclass for counting hashable objects. It is a collection where elements are stored as dictionary keys and their counts are stored as dictionary values. Counts are allowed to be any integer value including zero or negative counts. The **`Counter`** class is similar to bags or multisets in other languages.

Elements are counted from an *iterable* or initialized from another *mapping* (or counter):

```
>>> c = Counter()                           # a new, empty
counter
>>> c = Counter('gallahad')                 # a new counter
from an iterable
>>> c = Counter({'red': 4, 'blue': 2})      # a new counter
from a mapping
>>> c = Counter(cats=4, dogs=8)             # a new counter
from keyword args
```

Counter objects have a dictionary interface except that they return a zero count for missing items instead of raising a **`KeyError`**:

```
>>> c = Counter(['eggs', 'ham'])
>>> c['bacon']                              # count of a
missing element is zero
0
```

Setting a count to zero does not remove an element from a counter. Use `del` to remove it entirely:

```
>>> c['sausage'] = 0                        # counter entry
with a zero count
>>> del c['sausage']                        # del actually
removes the entry
```

Added in version 3.1.

Changed in version 3.7: As a `dict` subclass, `Counter` inherited the capability to remember insertion order. Math operations on *Counter* objects also preserve order. Results are ordered according to when an element is first encountered in the left operand and then by the order encountered in the right operand.

Counter objects support additional methods beyond those available for all dictionaries:

elements()

Return an iterator over elements repeating each as many times as its count. Elements are returned in the order first encountered. If an element’s count is less than one, `elements()` will ignore it.

```
>>> c = Counter(a=4, b=2, c=0, d=-2)
>>> sorted(c.elements())
['a', 'a', 'a', 'a', 'b', 'b']
```

most_common([*n*])

Return a list of the *n* most common elements and their counts from the most common to the least. If *n* is omitted or `None`, `most_common()` returns *all* elements in the counter. Elements with equal counts are ordered in the order first encountered:

```
>>> Counter('abracadabra').most_common(3)
[('a', 5), ('b', 2), ('r', 2)]
```

subtract([*iterable-or-mapping*])

Elements are subtracted from an *iterable* or from another *mapping* (or counter). Like `dict.update()` but subtracts counts instead of replacing them. Both inputs and outputs may be zero or negative.

```
>>> c = Counter(a=4, b=2, c=0, d=-2)
>>> d = Counter(a=1, b=2, c=3, d=4)
>>> c.subtract(d)
>>> c
Counter({'a': 3, 'b': 0, 'c': -3, 'd': -6})
```

Added in version 3.2.

total()

Compute the sum of the counts.

```
>>> c = Counter(a=10, b=5, c=0)
>>> c.total()
15
```

Added in version 3.10.

The usual dictionary methods are available for **`Counter`** objects except for two which work differently for counters.

fromkeys(*iterable*)

This class method is not implemented for **`Counter`** objects.

update([*iterable-or-mapping*])

Elements are counted from an *iterable* or added-in from another *mapping* (or counter). Like **`dict.update()`** but adds counts instead of replacing them. Also, the *iterable* is expected to be a sequence of elements, not a sequence of `(key, value)` pairs.

Counters support rich comparison operators for equality, subset, and superset relationships: `==`, `!=`, `<`, `<=`, `>`, `>=`. All of those tests treat missing elements as having zero counts so that `Counter(a=1) == Counter(a=1, b=0)` returns true.

Changed in version 3.10: Rich comparison operations were added.

Changed in version 3.10: In equality tests, missing elements are treated as having zero counts. Formerly, `Counter(a=3)` and `Counter(a=3, b=0)` were considered distinct.

Common patterns for working with **`Counter`** objects:

```
c.total()                       # total of all counts
c.clear()                       # reset all counts
list(c)                         # list unique elements
set(c)                          # convert to a set
dict(c)                         # convert to a regular
dictionary
c.items()                       # access the (elem, cnt) pairs
Counter(dict(list_of_pairs))    # convert from a list of (elem,
cnt) pairs
c.most_common()[:-n-1:-1]       # n least common elements
+c                              # remove zero and negative
counts
```

Several mathematical operations are provided for combining **`Counter`** objects to produce multisets (counters that have counts greater than zero). Addition and subtraction combine counters by adding or subtracting the counts of corresponding elements. Intersection and union return the minimum and maximum of corresponding counts. Equality and inclusion compare corresponding counts. Each operation can accept inputs with signed counts, but the output will exclude results with counts of zero or less.

```
>>> c = Counter(a=3, b=1)
>>> d = Counter(a=1, b=2)
>>> c + d                       # add two counters together:
c[x] + d[x]
Counter({'a': 4, 'b': 3})
>>> c - d                       # subtract (keeping only
positive counts)
Counter({'a': 2})
```

```
>>> c & d                       # intersection:  min(c[x],
d[x])
Counter({'a': 1, 'b': 1})
>>> c | d                       # union:  max(c[x], d[x])
Counter({'a': 3, 'b': 2})
>>> c == d                      # equality:  c[x] == d[x]
False
>>> c <= d                      # inclusion:  c[x] <= d[x]
False
```

Unary addition and subtraction are shortcuts for adding an empty counter or subtracting from an empty counter.

```
>>> c = Counter(a=2, b=-4)
>>> +c
Counter({'a': 2})
>>> -c
Counter({'b': 4})
```

Added in version 3.3: Added support for unary plus, unary minus, and in-place multiset operations.

Note

Counters were primarily designed to work with positive integers to represent running counts; however, care was taken to not unnecessarily preclude use cases needing other types or negative values. To help with those use cases, this section documents the minimum range and type restrictions.

- The `Counter` class itself is a dictionary subclass with no restrictions on its keys and values. The values are intended to be numbers representing counts, but you *could* store anything in the value field.
- The `most_common()` method requires only that the values be orderable.
- For in-place operations such as `c[key] += 1`, the value type need only support addition and subtraction. So

fractions, floats, and decimals would work and negative values are supported. The same is also true for `update()` and `subtract()` which allow negative and zero values for both inputs and outputs.

- The multiset methods are designed only for use cases with positive values. The inputs may be negative or zero, but only outputs with positive values are created. There are no type restrictions, but the value type needs to support addition, subtraction, and comparison.
- The `elements()` method requires integer counts. It ignores zero and negative counts.

See also

- Bag class [https://www.gnu.org/software/smalltalk/manual-base/html_node/Bag.html] in Smalltalk.

- Wikipedia entry for Multisets [https://en.wikipedia.org/wiki/Multiset].

- C++ multisets [http://www.java2s.com/Tutorial/Cpp/0380__set-multiset/Catalog0380__set-multiset.htm] tutorial with examples.

- For mathematical operations on multisets and their use cases, see *Knuth, Donald. The Art of Computer Programming Volume II, Section 4.6.3, Exercise 19*.

- To enumerate all distinct multisets of a given size over a given set of elements, see `itertools.combinations_with_replacement()`:

  ```
  map(Counter, combinations_with_replacement('ABC', 2)) # -
  -> AA AB AC BB BC CC
  ```

`deque` objects

class collections.deque([*iterable*[, *maxlen*]])

Returns a new deque object initialized left-to-right (using `append()`) with data from *iterable*. If *iterable* is not specified, the new deque is empty.

Deques are a generalization of stacks and queues (the name is pronounced “deck” and is short for “double-ended queue”). Deques support thread-safe, memory efficient appends and pops from either side of the deque with approximately the same $O(1)$ performance in either direction.

Though `list` objects support similar operations, they are optimized for fast fixed-length operations and incur $O(n)$ memory movement costs for `pop(0)` and `insert(0, v)` operations which change both the size and position of the underlying data representation.

If *maxlen* is not specified or is `None`, deques may grow to an arbitrary length. Otherwise, the deque is bounded to the specified maximum length. Once a bounded length deque is full, when new items are added, a corresponding number of items are discarded from the opposite end. Bounded length deques provide functionality similar to the `tail` filter in Unix. They are also useful for tracking transactions and other pools of data where only the most recent activity is of interest.

Deque objects support the following methods:

append(*x*)

Add *x* to the right side of the deque.

appendleft(*x*)

Add *x* to the left side of the deque.

clear()

Remove all elements from the deque leaving it with length 0.

copy()

Create a shallow copy of the deque.

Added in version 3.5.

count(*x*)

Count the number of deque elements equal to *x*.

Added in version 3.2.

extend(*iterable*)

Extend the right side of the deque by appending elements from the iterable argument.

extendleft(*iterable*)

Extend the left side of the deque by appending elements from *iterable*. Note, the series of left appends results in reversing the order of elements in the iterable argument.

index(*x*[, *start*[, *stop*]])

Return the position of *x* in the deque (at or after index *start* and before index *stop*). Returns the first match or raises `ValueError` if not found.

Added in version 3.5.

insert(*i*, *x*)

Insert *x* into the deque at position *i*.

If the insertion would cause a bounded deque to grow beyond *maxlen*, an `IndexError` is raised.

Added in version 3.5.

pop()

Remove and return an element from the right side of the deque. If no elements are present, raises an `IndexError`.

popleft()

Remove and return an element from the left side of the deque. If no elements are present, raises an `IndexError`.

remove(*value*)

Remove the first occurrence of *value*. If not found, raises a `ValueError`.

reverse()

Reverse the elements of the deque in-place and then return `None`.

Added in version 3.2.

rotate(*n=1*)

Rotate the deque *n* steps to the right. If *n* is negative, rotate to the left.

When the deque is not empty, rotating one step to the right is equivalent to `d.appendleft(d.pop())`, and rotating one step to the left is equivalent to `d.append(d.popleft())`.

Deque objects also provide one read-only attribute:

maxlen

Maximum size of a deque or `None` if unbounded.

Added in version 3.1.

In addition to the above, deques support iteration, pickling, `len(d)`, `reversed(d)`, `copy.copy(d)`, `copy.deepcopy(d)`, membership testing with the **`in`** operator, and subscript references such as `d[0]` to access the first element. Indexed access is $O(1)$ at both ends but slows to $O(n)$ in the middle. For fast random access, use lists instead.

Starting in version 3.5, deques support `__add__()`, `__mul__()`, and `__imul__()`.

Example:

```
>>> from collections import deque
>>> d = deque('ghi')                 # make a new deque with
three items
>>> for elem in d:                   # iterate over the deque's
elements
...     print(elem.upper())
G
H
I

>>> d.append('j')                    # add a new entry to the
right side
>>> d.appendleft('f')                # add a new entry to the
left side
>>> d                                # show the representation
of the deque
deque(['f', 'g', 'h', 'i', 'j'])

>>> d.pop()                          # return and remove the
rightmost item
'j'
```

```
>>> d.popleft()                      # return and remove the
leftmost item
'f'
>>> list(d)                          # list the contents of the
deque
['g', 'h', 'i']
>>> d[0]                             # peek at leftmost item
'g'
>>> d[-1]                            # peek at rightmost item
'i'

>>> list(reversed(d))                # list the contents of a
deque in reverse
['i', 'h', 'g']
>>> 'h' in d                         # search the deque
True
>>> d.extend('jkl')                  # add multiple elements at
once
>>> d
deque(['g', 'h', 'i', 'j', 'k', 'l'])
>>> d.rotate(1)                      # right rotation
>>> d
deque(['l', 'g', 'h', 'i', 'j', 'k'])
>>> d.rotate(-1)                     # left rotation
>>> d
deque(['g', 'h', 'i', 'j', 'k', 'l'])

>>> deque(reversed(d))               # make a new deque in
reverse order
deque(['l', 'k', 'j', 'i', 'h', 'g'])
>>> d.clear()                        # empty the deque
>>> d.pop()                          # cannot pop from an empty
deque
Traceback (most recent call last):
    File "<pyshell#6>", line 1, in -toplevel-
        d.pop()
IndexError: pop from an empty deque

>>> d.extendleft('abc')              # extendleft() reverses
the input order
>>> d
deque(['c', 'b', 'a'])
```

`deque` Recipes

This section shows various approaches to working with deques.

Bounded length deques provide functionality similar to the `tail` filter in Unix:

```
def tail(filename, n=10):
    'Return the last n lines of a file'
    with open(filename) as f:
        return deque(f, n)
```

Another approach to using deques is to maintain a sequence of recently added elements by appending to the right and popping to the left:

```
def moving_average(iterable, n=3):
    # moving_average([40, 30, 50, 46, 39, 44]) --> 40.0 42.0
45.0 43.0
    # https://en.wikipedia.org/wiki/Moving_average
    it = iter(iterable)
    d = deque(itertools.islice(it, n-1))
    d.appendleft(0)
    s = sum(d)
    for elem in it:
        s += elem - d.popleft()
        d.append(elem)
        yield s / n
```

A round-robin scheduler [https://en.wikipedia.org/wiki/Round-robin_scheduling] can be implemented with input iterators stored in a `deque`. Values are yielded from the active iterator in position zero. If that iterator is exhausted, it can be removed with `popleft()`; otherwise, it can be cycled back to the end with the `rotate()` method:

```
def roundrobin(*iterables):
    "roundrobin('ABC', 'D', 'EF') --> A D E B F C"
    iterators = deque(map(iter, iterables))
    while iterators:
        try:
            while True:
                yield next(iterators[0])
                iterators.rotate(-1)
        except StopIteration:
            # Remove an exhausted iterator.
            iterators.popleft()
```

The **`rotate()`** method provides a way to implement **`deque`** slicing and deletion. For example, a pure Python implementation of `del d[n]` relies on the `rotate()` method to position elements to be popped:

```
def delete_nth(d, n):
    d.rotate(-n)
    d.popleft()
    d.rotate(n)
```

To implement **`deque`** slicing, use a similar approach applying **`rotate()`** to bring a target element to the left side of the deque. Remove old entries with **`popleft()`**, add new entries with **`extend()`**, and then reverse the rotation. With minor variations on that approach, it is easy to implement Forth style stack manipulations such as `dup`, `drop`, `swap`, `over`, `pick`, `rot`, and `roll`.

`defaultdict` objects

class collections.defaultdict(*default_factory=None*, */*[, *...*])

Return a new dictionary-like object. **`defaultdict`** is a subclass of the built-in **`dict`** class. It overrides one method and adds one writable instance variable. The remaining functionality is the same as for the **`dict`** class and is not documented here.

The first argument provides the initial value for the `default_factory` attribute; it defaults to `None`. All remaining arguments are treated the same as if they were passed to the `dict` constructor, including keyword arguments.

`defaultdict` objects support the following method in addition to the standard `dict` operations:

__missing__(*key*)

If the `default_factory` attribute is `None`, this raises a `KeyError` exception with the *key* as argument.

If `default_factory` is not `None`, it is called without arguments to provide a default value for the given *key*, this value is inserted in the dictionary for the *key*, and returned.

If calling `default_factory` raises an exception this exception is propagated unchanged.

This method is called by the `__getitem__()` method of the `dict` class when the requested key is not found; whatever it returns or raises is then returned or raised by `__getitem__()`.

Note that `__missing__()` is *not* called for any operations besides `__getitem__()`. This means that `get()` will, like normal dictionaries, return `None` as a default rather than using `default_factory`.

`defaultdict` objects support the following instance variable:

default_factory

This attribute is used by the `__missing__()` method; it is initialized from the first argument to the constructor, if present, or to `None`, if absent.

Changed in version 3.9: Added merge (`|`) and update (`|=`) operators, specified in **PEP 584** [https://peps.python.org/pep-0584/].

`defaultdict` Examples

Using `list` as the `default_factory`, it is easy to group a sequence of key-value pairs into a dictionary of lists:

```
>>> s = [('yellow', 1), ('blue', 2), ('yellow', 3), ('blue',
4), ('red', 1)]
>>> d = defaultdict(list)
>>> for k, v in s:
...     d[k].append(v)
...
>>> sorted(d.items())
[('blue', [2, 4]), ('red', [1]), ('yellow', [1, 3])]
```

When each key is encountered for the first time, it is not already in the mapping; so an entry is automatically created using the `default_factory` function which returns an empty `list`. The `list.append()` operation then attaches the value to the new list. When keys are encountered again, the look-up proceeds normally (returning the list for that key) and the `list.append()` operation adds another value to the list. This technique is simpler and faster than an equivalent technique using `dict.setdefault()`:

```
>>> d = {}
>>> for k, v in s:
...     d.setdefault(k, []).append(v)
...
>>> sorted(d.items())
[('blue', [2, 4]), ('red', [1]), ('yellow', [1, 3])]
```

Setting the `default_factory` to `int` makes the `defaultdict` useful for counting (like a bag or multiset in other languages):

```
>>> s = 'mississippi'
>>> d = defaultdict(int)
>>> for k in s:
...     d[k] += 1
...
>>> sorted(d.items())
[('i', 4), ('m', 1), ('p', 2), ('s', 4)]
```

When a letter is first encountered, it is missing from the mapping, so the **`default_factory`** function calls **`int()`** to supply a default count of zero. The increment operation then builds up the count for each letter.

The function **`int()`** which always returns zero is just a special case of constant functions. A faster and more flexible way to create constant functions is to use a lambda function which can supply any constant value (not just zero):

```
>>> def constant_factory(value):
...     return lambda: value
...
>>> d = defaultdict(constant_factory('<missing>'))
>>> d.update(name='John', action='ran')
>>> '%(name)s %(action)s to %(object)s' % d
'John ran to <missing>'
```

Setting the **`default_factory`** to **`set`** makes the **`defaultdict`** useful for building a dictionary of sets:

```
>>> s = [('red', 1), ('blue', 2), ('red', 3), ('blue', 4),
('red', 1), ('blue', 4)]
>>> d = defaultdict(set)
>>> for k, v in s:
...     d[k].add(v)
...
>>> sorted(d.items())
[('blue', {2, 4}), ('red', {1, 3})]
```

`namedtuple()` Factory Function for Tuples with Named Fields

Named tuples assign meaning to each position in a tuple and allow for more readable, self-documenting code. They can be used wherever regular tuples are used, and they add the ability to access fields by name instead of position index.

collections.namedtuple(*typename*, *field_names*, *, *rename=False*, *defaults=None*, *module=None*)

Returns a new tuple subclass named *typename*. The new subclass is used to create tuple-like objects that have fields accessible by attribute lookup as well as being indexable and iterable. Instances of the subclass also have a helpful docstring (with *typename* and *field_names*) and a helpful `__repr__()` method which lists the tuple contents in a `name=value` format.

The *field_names* are a sequence of strings such as `['x', 'y']`. Alternatively, *field_names* can be a single string with each fieldname separated by whitespace and/or commas, for example `'x y'` or `'x, y'`.

Any valid Python identifier may be used for a fieldname except for names starting with an underscore. Valid identifiers consist of letters, digits, and underscores but do not start with a digit or underscore and cannot be a `keyword` such as *class*, *for*, *return*, *global*, *pass*, or *raise*.

If *rename* is true, invalid fieldnames are automatically replaced with positional names. For example, `['abc', 'def', 'ghi', 'abc']` is converted to `['abc', '_1', 'ghi', '_3']`, eliminating the keyword `def` and the duplicate fieldname `abc`.

defaults can be `None` or an iterable of default values. Since fields with a default value must come after any fields without a default, the *defaults* are applied to the rightmost parameters. For example, if the fieldnames are `['x', 'y', 'z']` and the defaults are `(1, 2)`, then `x` will be a required argument, `y` will default to `1`, and `z` will default to `2`.

If *module* is defined, the **`__module__`** attribute of the named tuple is set to that value.

Named tuple instances do not have per-instance dictionaries, so they are lightweight and require no more memory than regular tuples.

To support pickling, the named tuple class should be assigned to a variable that matches *typename*.

Changed in version 3.1: Added support for *rename*.

Changed in version 3.6: The *verbose* and *rename* parameters became keyword-only arguments.

Changed in version 3.6: Added the *module* parameter.

Changed in version 3.7: Removed the *verbose* parameter and the **`_source`** attribute.

Changed in version 3.7: Added the *defaults* parameter and the **`_field_defaults`** attribute.

```
>>> # Basic example
>>> Point = namedtuple('Point', ['x', 'y'])
>>> p = Point(11, y=22)     # instantiate with positional or
keyword arguments
>>> p[0] + p[1]             # indexable like the plain tuple
(11, 22)
33
>>> x, y = p                # unpack like a regular tuple
>>> x, y
(11, 22)
>>> p.x + p.y                # fields also accessible by name
33
>>> p                        # readable __repr__ with a
name=value style
Point(x=11, y=22)
```

Named tuples are especially useful for assigning field names to result tuples returned by the **`csv`** or **`sqlite3`** modules:

```
EmployeeRecord = namedtuple('EmployeeRecord', 'name, age,
title, department, paygrade')

import csv
for emp in map(EmployeeRecord._make,
csv.reader(open("employees.csv", "rb"))):
    print(emp.name, emp.title)

import sqlite3
conn = sqlite3.connect('/companydata')
cursor = conn.cursor()
cursor.execute('SELECT name, age, title, department, paygrade
FROM employees')
for emp in map(EmployeeRecord._make, cursor.fetchall()):
    print(emp.name, emp.title)
```

In addition to the methods inherited from tuples, named tuples support three additional methods and two attributes. To prevent conflicts with field names, the method and attribute names start with an underscore.

classmethod somenamedtuple._make(*iterable*)

Class method that makes a new instance from an existing sequence or iterable.

```
>>> t = [11, 22]
>>> Point._make(t)
Point(x=11, y=22)
```

somenamedtuple._asdict()

Return a new `dict` which maps field names to their corresponding values:

```
>>> p = Point(x=11, y=22)
>>> p._asdict()
{'x': 11, 'y': 22}
```

Changed in version 3.1: Returns an `OrderedDict` instead of a regular `dict`.

Changed in version 3.8: Returns a regular `dict` instead of an `OrderedDict`. As of Python 3.7, regular dicts are guaranteed to be ordered. If the extra features of `OrderedDict` are required, the suggested remediation is to cast the result to the desired type: `OrderedDict(nt._asdict())`.

somenamedtuple._replace(***kwargs*)

Return a new instance of the named tuple replacing specified fields with new values:

```
>>> p = Point(x=11, y=22)
>>> p._replace(x=33)
Point(x=33, y=22)

>>> for partnum, record in inventory.items():
...     inventory[partnum] =
record._replace(price=newprices[partnum],
timestamp=time.now())
```

Named tuples are also supported by generic function `copy.replace()`.

Changed in version 3.13: Raise `TypeError` instead of `ValueError` for invalid keyword arguments.

somenamedtuple._fields

Tuple of strings listing the field names. Useful for introspection and for creating new named tuple types from existing named tuples.

```
>>> p._fields            # view the field names
('x', 'y')

>>> Color = namedtuple('Color', 'red green blue')
>>> Pixel = namedtuple('Pixel', Point._fields +
Color._fields)
>>> Pixel(11, 22, 128, 255, 0)
Pixel(x=11, y=22, red=128, green=255, blue=0)
```

somenamedtuple._field_defaults

Dictionary mapping field names to default values.

```
>>> Account = namedtuple('Account', ['type', 'balance'],
defaults=[0])
>>> Account._field_defaults
{'balance': 0}
>>> Account('premium')
Account(type='premium', balance=0)
```

To retrieve a field whose name is stored in a string, use the **`getattr()`** function:

```
>>> getattr(p, 'x')
11
```

To convert a dictionary to a named tuple, use the double-star-operator (as described in Unpacking Argument Lists):

```
>>> d = {'x': 11, 'y': 22}
>>> Point(**d)
Point(x=11, y=22)
```

Since a named tuple is a regular Python class, it is easy to add or change functionality with a subclass. Here is how to add a calculated field and a fixed-width print format:

```
>>> class Point(namedtuple('Point', ['x', 'y'])):
...     __slots__ = ()
...     @property
...     def hypot(self):
...         return (self.x ** 2 + self.y ** 2) ** 0.5
...     def __str__(self):
...         return 'Point: x=%6.3f  y=%6.3f  hypot=%6.3f' %
(self.x, self.y, self.hypot)

>>> for p in Point(3, 4), Point(14, 5/7):
...     print(p)
Point: x= 3.000  y= 4.000  hypot= 5.000
Point: x=14.000  y= 0.714  hypot=14.018
```

The subclass shown above sets `__slots__` to an empty tuple. This helps keep memory requirements low by preventing the creation of instance dictionaries.

Subclassing is not useful for adding new, stored fields. Instead, simply create a new named tuple type from the **`_fields`** attribute:

```
>>> Point3D = namedtuple('Point3D', Point._fields + ('z',))
```

Docstrings can be customized by making direct assignments to the `__doc__` fields:

```
>>> Book = namedtuple('Book', ['id', 'title', 'authors'])
>>> Book.__doc__ += ': Hardcover book in active collection'
>>> Book.id.__doc__ = '13-digit ISBN'
>>> Book.title.__doc__ = 'Title of first printing'
>>> Book.authors.__doc__ = 'List of authors sorted by last
name'
```

Changed in version 3.5: Property docstrings became writeable.

See also

- See `typing.NamedTuple` for a way to add type hints for named tuples. It also provides an elegant notation using the `class` keyword:

  ```
  class Component(NamedTuple):
      part_number: int
      weight: float
      description: Optional[str] = None
  ```

- See `types.SimpleNamespace()` for a mutable namespace based on an underlying dictionary instead of a tuple.

- The `dataclasses` module provides a decorator and functions for automatically adding generated special methods to user-defined classes.

`OrderedDict` objects

Ordered dictionaries are just like regular dictionaries but have some extra capabilities relating to ordering operations. They have become less important now that the built-in `dict` class gained the ability to remember insertion order (this new behavior became guaranteed in Python 3.7).

Some differences from `dict` still remain:

- The regular `dict` was designed to be very good at mapping operations. Tracking insertion order was secondary.

- The `OrderedDict` was designed to be good at reordering operations. Space efficiency, iteration speed, and the performance of update operations were secondary.

- The `OrderedDict` algorithm can handle frequent reordering operations better than `dict`. As shown in the recipes below, this makes it suitable for implementing various kinds of LRU caches.

- The equality operation for `OrderedDict` checks for matching order.

  A regular `dict` can emulate the order sensitive equality test with `p == q and all(k1 == k2 for k1, k2 in zip(p, q))`.

- The `popitem()` method of `OrderedDict` has a different signature. It accepts an optional argument to specify which item is popped.

  A regular `dict` can emulate OrderedDict's `od.popitem(last=True)` with `d.popitem()` which is guaranteed to pop the rightmost (last) item.

  A regular `dict` can emulate OrderedDict's `od.popitem(last=False)` with `(k := next(iter(d)), d.pop(k))` which will return and remove the leftmost (first) item if it exists.

- `OrderedDict` has a `move_to_end()` method to efficiently reposition an element to an endpoint.

  A regular `dict` can emulate OrderedDict's `od.move_to_end(k, last=True)` with `d[k] = d.pop(k)` which will move the key and its associated value to the rightmost (last) position.

  A regular `dict` does not have an efficient equivalent for OrderedDict's `od.move_to_end(k, last=False)` which moves the key and its associated value to the leftmost (first) position.

- Until Python 3.8, `dict` lacked a `__reversed__()` method.

class collections.OrderedDict([*items*])

Return an instance of a **`dict`** subclass that has methods specialized for rearranging dictionary order.

Added in version 3.1.

popitem(*last=True*)

The **`popitem()`** method for ordered dictionaries returns and removes a (key, value) pair. The pairs are returned in LIFO order if *last* is true or FIFO order if false.

move_to_end(*key*, *last=True*)

Move an existing *key* to either end of an ordered dictionary. The item is moved to the right end if *last* is true (the default) or to the beginning if *last* is false. Raises **`KeyError`** if the *key* does not exist:

```
>>> d = OrderedDict.fromkeys('abcde')
>>> d.move_to_end('b')
>>> ''.join(d)
'acdeb'
>>> d.move_to_end('b', last=False)
>>> ''.join(d)
'bacde'
```

Added in version 3.2.

In addition to the usual mapping methods, ordered dictionaries also support reverse iteration using **`reversed()`**.

Equality tests between **`OrderedDict`** objects are order-sensitive and are roughly equivalent to `list(od1.items())==list(od2.items())`.

Equality tests between **`OrderedDict`** objects and other **`Mapping`** objects are order-insensitive like regular dictionaries. This allows **`OrderedDict`** objects to be substituted anywhere a regular dictionary is used.

Changed in version 3.5: The items, keys, and values views of **`OrderedDict`** now support reverse iteration using **`reversed()`**.

Changed in version 3.6: With the acceptance of **PEP 468** [https://peps.python.org/pep-0468/], order is retained for keyword arguments passed to the **`OrderedDict`** constructor and its **`update()`** method.

Changed in version 3.9: Added merge (`|`) and update (`|=`) operators, specified in **PEP 584** [https://peps.python.org/pep-0584/].

`OrderedDict` Examples and Recipes

It is straightforward to create an ordered dictionary variant that remembers the order the keys were *last* inserted. If a new entry overwrites an existing entry, the original insertion position is changed and moved to the end:

```
class LastUpdatedOrderedDict(OrderedDict):
    'Store items in the order the keys were last added'

    def __setitem__(self, key, value):
        super().__setitem__(key, value)
        self.move_to_end(key)
```

An **`OrderedDict`** would also be useful for implementing variants of **`functools.lru_cache()`**:

```
from collections import OrderedDict
from time import time

class TimeBoundedLRU:
    "LRU Cache that invalidates and refreshes old entries."

    def __init__(self, func, maxsize=128, maxage=30):
        self.cache = OrderedDict()      # { args : (timestamp,
result)}
        self.func = func
        self.maxsize = maxsize
        self.maxage = maxage

    def __call__(self, *args):
        if args in self.cache:
            self.cache.move_to_end(args)
            timestamp, result = self.cache[args]
            if time() - timestamp <= self.maxage:
                return result
        result = self.func(*args)
        self.cache[args] = time(), result
        if len(self.cache) > self.maxsize:
            self.cache.popitem(last=False)
        return result

class MultiHitLRUCache:
    """ LRU cache that defers caching a result until
        it has been requested multiple times.

        To avoid flushing the LRU cache with one-time requests,
        we don't cache until a request has been made more than
once.

    """

    def __init__(self, func, maxsize=128, maxrequests=4096,
cache_after=1):
        self.requests = OrderedDict()   # { uncached_key :
request_count }
        self.cache = OrderedDict()      # { cached_key :
function_result }
        self.func = func
        self.maxrequests = maxrequests  # max number of
uncached requests
        self.maxsize = maxsize          # max number of stored
return values
        self.cache_after = cache_after

    def __call__(self, *args):
        if args in self.cache:
            self.cache.move_to_end(args)
            return self.cache[args]
        result = self.func(*args)
        self.requests[args] = self.requests.get(args, 0) + 1
        if self.requests[args] <= self.cache_after:
            self.requests.move_to_end(args)
            if len(self.requests) > self.maxrequests:
                self.requests.popitem(last=False)
        else:
            self.requests.pop(args, None)
            self.cache[args] = result
            if len(self.cache) > self.maxsize:
                self.cache.popitem(last=False)
        return result
```

`UserDict` objects

The class, `UserDict` acts as a wrapper around dictionary objects. The need for this class has been partially supplanted by the ability to subclass directly from `dict`; however, this class can be easier to work with because the underlying dictionary is accessible as an attribute.

class collections.UserDict([*initialdata*])

Class that simulates a dictionary. The instance’s contents are kept in a regular dictionary, which is accessible via the `data` attribute of `UserDict` instances. If *initialdata* is provided, `data` is initialized with its contents; note that a reference to *initialdata* will not be kept, allowing it to be used for other purposes.

In addition to supporting the methods and operations of mappings, `UserDict` instances provide the following attribute:

data

A real dictionary used to store the contents of the `UserDict` class.

`UserList` objects

This class acts as a wrapper around list objects. It is a useful base class for your own list-like classes which can inherit from them and override existing methods or add new ones. In this way, one can add new behaviors to lists.

The need for this class has been partially supplanted by the ability to subclass directly from `list`; however, this class can be easier to work with because the underlying list is accessible as an attribute.

class collections.UserList([*list*])

Class that simulates a list. The instance’s contents are kept in a regular list, which is accessible via the `data` attribute of `UserList` instances. The instance’s contents are initially set to a copy of *list*, defaulting to the empty list `[]`. *list* can be any iterable, for example a real Python list or a `UserList` object.

In addition to supporting the methods and operations of mutable sequences, `UserList` instances provide the following attribute:

data

A real `list` object used to store the contents of the `UserList` class.

Subclassing requirements: Subclasses of `UserList` are expected to offer a constructor which can be called with either no arguments or one argument. List operations which return a new sequence attempt to create an instance of the actual implementation class. To do so, it assumes that the constructor can be called with a single parameter, which is a sequence object used as a data source.

If a derived class does not wish to comply with this requirement, all of the special methods supported by this class will need to be overridden; please consult the sources for information about the methods which need to be provided in that case.

`UserString` objects

The class, `UserString` acts as a wrapper around string objects. The need for this class has been partially supplanted by the ability to subclass directly from `str`; however, this class can be easier to work with because the underlying string is accessible as an attribute.

class collections.UserString(*seq*)

Class that simulates a string object. The instance's content is kept in a regular string object, which is accessible via the `data` attribute of `UserString` instances. The instance's contents are initially set to a copy of *seq*. The *seq* argument can be any object which can be converted into a string using the built-in `str()` function.

In addition to supporting the methods and operations of strings, `UserString` instances provide the following attribute:

data

A real `str` object used to store the contents of the `UserString` class.

Changed in version 3.5: New methods `__getnewargs__`, `__rmod__`, `casefold`, `format_map`, `isprintable`, and `maketrans`.

`collections.abc` — Abstract Base Classes for Containers

Added in version 3.3: Formerly, this module was part of the **`collections`** module.

Source code: Lib/_collections_abc.py
[https://github.com/python/cpython/tree/3.14/Lib/_collections_abc.py]

This module provides abstract base classes that can be used to test whether a class provides a particular interface; for example, whether it is hashable or whether it is a mapping.

An **`issubclass()`** or **`isinstance()`** test for an interface works in one of three ways.

1. A newly written class can inherit directly from one of the abstract base classes. The class must supply the required abstract methods. The remaining mixin methods come from inheritance and can be overridden if desired. Other methods may be added as needed:

```
class C(Sequence):                      # Direct
inheritance
    def __init__(self): ...             # Extra method not
required by the ABC
    def __getitem__(self, index):  ...  # Required abstract
method
    def __len__(self):  ...             # Required abstract
method
```

```
    def count(self, value): ...         # Optionally
override a mixin method

>>> issubclass(C, Sequence)
True
>>> isinstance(C(), Sequence)
True
```

2. Existing classes and built-in classes can be registered as “virtual subclasses” of the ABCs. Those classes should define the full API including all of the abstract methods and all of the mixin methods. This lets users rely on **`issubclass()`** or **`isinstance()`** tests to determine whether the full interface is supported. The exception to this rule is for methods that are automatically inferred from the rest of the API:

```
class D:                                 # No inheritance
    def __init__(self): ...              # Extra method not
required by the ABC
    def __getitem__(self, index):  ...   # Abstract method
    def __len__(self):  ...              # Abstract method
    def count(self, value): ...          # Mixin method
    def index(self, value): ...          # Mixin method

Sequence.register(D)                     # Register instead
of inherit

>>> issubclass(D, Sequence)
True
>>> isinstance(D(), Sequence)
True
```

In this example, class **`D`** does not need to define `__contains__`, `__iter__`, and `__reversed__` because the in-operator, the iteration logic, and the **`reversed()`** function automatically fall back to using `__getitem__` and `__len__`.

3. Some simple interfaces are directly recognizable by the presence of the required methods (unless those

methods have been set to **`None`**):

```
class E:
    def __iter__(self): ...
    def __next__(self): ...

>>> issubclass(E, Iterable)
True
>>> isinstance(E(), Iterable)
True
```

Complex interfaces do not support this last technique because an interface is more than just the presence of method names. Interfaces specify semantics and relationships between methods that cannot be inferred solely from the presence of specific method names. For example, knowing that a class supplies `__getitem__`, `__len__`, and `__iter__` is insufficient for distinguishing a **`Sequence`** from a **`Mapping`**.

Added in version 3.9: These abstract classes now support `[]`. See Generic Alias Type and **PEP 585** [https://peps.python.org/pep-0585/].

Collections Abstract Base Classes

The collections module offers the following ABCs:

ABC	Inherits from	Abstract Methods	Mixin Methods
`Container` [[1]]		`__contains__`	
`Hashable` [[1]]		`__hash__`	

ABC	Inherits from	Abstract Methods	Mixin Methods
`Iterable` [[1]] [[2]]		`__iter__`	
`Iterator` [[1]]	`Iterable`	`__next__`	`__iter__`
`Reversible` [[1]]	`Iterable`	`__reversed__`	
`Generator` [[1]]	`Iterator`	`send`, `throw`	`close`, `__iter__`, `__next__`
`Sized` [[1]]		`__len__`	
`Callable` [[1]]		`__call__`	
`Collection` [[1]]	`Sized`, `Iterable`, `Container`	`__contains__`, `__iter__`, `__len__`	
`Sequence`	`Reversible`, `Collection`	`__getitem__`, `__len__`	`__contains__`, `__iter__`, `__reversed__`, `index`, and `count`

ABC	Inherits from	Abstract Methods	Mixin Methods
`MutableSequence`	`Sequence`	`__getitem__`, `__setitem__`, `__delitem__`, `__len__`, `insert`	Inherited `Sequence` methods and `append`, `clear`, `reverse`, `extend`, `pop`, `remove`, and `__iadd__`
`ByteString`	`Sequence`	`__getitem__`, `__len__`	Inherited `Sequence` methods
`Set`	`Collection`	`__contains__`, `__iter__`, `__len__`	`__le__`, `__lt__`, `__eq__`, `__ne__`, `__gt__`, `__ge__`, `__and__`, `__or__`, `__sub__`, `__rsub__`, `__xor__`, `__rxor__` and `isdisjoint`
`MutableSet`	`Set`	`__contains__`, `__iter__`, `__len__`, `add`, `discard`	Inherited `Set` methods and `clear`, `pop`, `remove`, `__ior__`, `__iand__`,

ABC	Inherits from	Abstract Methods	Mixin Methods
			`__ixor__`, and `__isub__`
`Mapping`	`Collection`	`__getitem__`, `__iter__`, `__len__`	`__contains__`, `keys`, `items`, `values`, `get`, `__eq__`, and `__ne__`
`MutableMapping`	`Mapping`	`__getitem__`, `__setitem__`, `__delitem__`, `__iter__`, `__len__`	Inherited `Mapping` methods and `pop`, `popitem`, `clear`, `update`, and `setdefault`
`MappingView`	`Sized`		`__init__`, `__len__` and `__repr__`
`ItemsView`	`MappingView`, `Set`		`__contains__`, `__iter__`
`KeysView`	`MappingView`, `Set`		`__contains__`, `__iter__`

ABC	Inherits from	Abstract Methods	Mixin Methods
`ValuesView`	`MappingView`, `Collection`		`__contains__`, `__iter__`
`Awaitable` [[1]]		`__await__`	
`Coroutine` [[1]]	`Awaitable`	`send`, `throw`	`close`
`AsyncIterable` [[1]]		`__aiter__`	
`AsyncIterator` [[1]]	`AsyncIterable`	`__anext__`	`__aiter__`
`AsyncGenerator` [[1]]	`AsyncIterator`	`asend`, `athrow`	`aclose`, `__aiter__`, `__anext__`
`Buffer` [[1]]		`__buffer__`	

Footnotes

[1] (1,2,3,4,5,6,7,8,9,10,11,12,13,14,15)

These ABCs override `__subclasshook__()` to support testing an interface by verifying the required methods are present and

have not been set to **`None`**. This only works for simple interfaces. More complex interfaces require registration or direct subclassing.
[2]

Checking `isinstance(obj, Iterable)` detects classes that are registered as **`Iterable`** or that have an **`__iter__()`** method, but it does not detect classes that iterate with the **`__getitem__()`** method. The only reliable way to determine whether an object is iterable is to call `iter(obj)`.

Collections Abstract Base Classes – Detailed Descriptions

class collections.abc.Container

ABC for classes that provide the **`__contains__()`** method.

class collections.abc.Hashable

ABC for classes that provide the **`__hash__()`** method.

class collections.abc.Sized

ABC for classes that provide the **`__len__()`** method.

class collections.abc.Callable

ABC for classes that provide the **`__call__()`** method.

See Annotating callable objects for details on how to use **`Callable`** in type annotations.

class collections.abc.Iterable

ABC for classes that provide the **`__iter__()`** method.

Checking `isinstance(obj, Iterable)` detects classes that are registered as **`Iterable`** or that have an **`__iter__()`** method, but it does not detect classes that iterate with the `__getitem__()` method. The only reliable way to determine whether an object is iterable is to call `iter(obj)`.

class collections.abc.Collection

ABC for sized iterable container classes.

Added in version 3.6.

class collections.abc.Iterator

ABC for classes that provide the `__iter__()` and `__next__()` methods. See also the definition of iterator.

class collections.abc.Reversible

ABC for iterable classes that also provide the `__reversed__()` method.

Added in version 3.6.

class collections.abc.Generator

ABC for generator classes that implement the protocol defined in **PEP 342** [https://peps.python.org/pep-0342/] that extends iterators with the `send()`, `throw()` and `close()` methods.

See Annotating generators and coroutines for details on using `Generator` in type annotations.

Added in version 3.5.

class collections.abc.Sequence
class collections.abc.MutableSequence
class collections.abc.ByteString

ABCs for read-only and mutable sequences.

Implementation note: Some of the mixin methods, such as `__iter__()`, `__reversed__()`, and `index()` make repeated calls to the underlying **`__getitem__()`** method. Consequently, if **`__getitem__()`** is implemented with constant access speed, the mixin methods will have linear performance; however, if the underlying method is linear (as it would be with a linked list), the mixins will have quadratic performance and will likely need to be overridden.

index(*value*, *start=0*, *stop=None*)

Return first index of *value*.

Raises **`ValueError`** if the value is not present.

Supporting the *start* and *stop* arguments is optional, but recommended.

Changed in version 3.5: The **`index()`** method gained support for the *stop* and *start* arguments.

Deprecated since version 3.12, will be removed in version 3.17: The **`ByteString`** ABC has been deprecated.

Use `isinstance(obj, collections.abc.Buffer)` to test if `obj` implements the buffer protocol at runtime. For use in type annotations, either use **`Buffer`** or a union that explicitly specifies the types your code supports (e.g., `bytes | bytearray | memoryview`).

`ByteString` was originally intended to be an abstract class that would serve as a supertype of both **`bytes`** and **`bytearray`**. However, since the ABC never had any methods, knowing that an object was an instance of **`ByteString`** never actually told you anything useful about the object. Other common buffer types such as **`memoryview`** were also never understood as subtypes of **`ByteString`** (either at runtime or by static type checkers).

See **PEP 688** [https://peps.python.org/pep-0688/#current-options] for more details.

class collections.abc.Set
class collections.abc.MutableSet

ABCs for read-only and mutable sets.

class collections.abc.Mapping
class collections.abc.MutableMapping

ABCs for read-only and mutable mappings.

class collections.abc.MappingView
class collections.abc.ItemsView
class collections.abc.KeysView
class collections.abc.ValuesView

ABCs for mapping, items, keys, and values views.

class collections.abc.Awaitable

ABC for awaitable objects, which can be used in **`await`** expressions. Custom implementations must provide the **`__await__()`** method.

Coroutine objects and instances of the **`Coroutine`** ABC are all instances of this ABC.

Note

In CPython, generator-based coroutines (generators decorated with **`@types.coroutine`**) are *awaitables*, even though they do not have an **`__await__()`** method. Using `isinstance(gencoro, Awaitable)` for them will return `False`. Use **`inspect.isawaitable()`** to detect them.

Added in version 3.5.

class collections.abc.Coroutine

ABC for coroutine compatible classes. These implement the following methods, defined in Coroutine Objects: `send()`, `throw()`, and `close()`. Custom implementations must also implement `__await__()`. All `Coroutine` instances are also instances of `Awaitable`.

Note

In CPython, generator-based coroutines (generators decorated with `@types.coroutine`) are *awaitables*, even though they do not have an `__await__()` method. Using `isinstance(gencoro, Coroutine)` for them will return `False`. Use `inspect.isawaitable()` to detect them.

See Annotating generators and coroutines for details on using **`Coroutine`** in type annotations. The variance and order of type parameters correspond to those of `Generator`.

Added in version 3.5.

class collections.abc.AsyncIterable

ABC for classes that provide an `__aiter__` method. See also the definition of asynchronous iterable.

Added in version 3.5.

class collections.abc.AsyncIterator

ABC for classes that provide `__aiter__` and `__anext__` methods. See also the definition of asynchronous iterator.

Added in version 3.5.

class collections.abc.AsyncGenerator

ABC for asynchronous generator classes that implement the protocol defined in **PEP 525** [https://peps.python.org/pep-

0525/] and **PEP 492** [https://peps.python.org/pep-0492/].

See Annotating generators and coroutines for details on using **`AsyncGenerator`** in type annotations.

Added in version 3.6.

class collections.abc.Buffer

ABC for classes that provide the **`__buffer__()`** method, implementing the buffer protocol. See **PEP 688** [https://peps.python.org/pep-0688/].

Added in version 3.12.

Examples and Recipes

ABCs allow us to ask classes or instances if they provide particular functionality, for example:

```
size = None
if isinstance(myvar, collections.abc.Sized):
    size = len(myvar)
```

Several of the ABCs are also useful as mixins that make it easier to develop classes supporting container APIs. For example, to write a class supporting the full **`Set`** API, it is only necessary to supply the three underlying abstract methods: **`__contains__()`**, **`__iter__()`**, and **`__len__()`**. The ABC supplies the remaining methods such as **`__and__()`** and **`isdisjoint()`**:

```
class ListBasedSet(collections.abc.Set):
    ''' Alternate set implementation favoring space over speed
        and not requiring the set elements to be hashable. '''
    def __init__(self, iterable):
        self.elements = lst = []
        for value in iterable:
            if value not in lst:
                lst.append(value)

    def __iter__(self):
        return iter(self.elements)

    def __contains__(self, value):
        return value in self.elements

    def __len__(self):
        return len(self.elements)

s1 = ListBasedSet('abcdef')
s2 = ListBasedSet('defghi')
overlap = s1 & s2            # The __and__() method is
supported automatically
```

Notes on using **`Set`** and **`MutableSet`** as a mixin:

1. Since some set operations create new sets, the default mixin methods need a way to create new instances from an iterable. The class constructor is assumed to have a signature in the form `ClassName(iterable)`. That assumption is factored-out to an internal **`classmethod`** called **`_from_iterable()`** which calls `cls(iterable)` to produce a new set. If the **`Set`** mixin is being used in a class with a different constructor signature, you will need to override **`_from_iterable()`** with a classmethod or regular method that can construct new instances from an iterable argument.
2. To override the comparisons (presumably for speed, as the semantics are fixed), redefine **`__le__()`** and **`__ge__()`**, then the other operations will automatically follow suit.
3. The **`Set`** mixin provides a **`_hash()`** method to compute a hash value for the set; however, **`__hash__()`** is not defined because not all sets are hashable or immutable. To add set hashability using mixins, inherit from both **`Set()`** and **`Hashable()`**, then define `__hash__ = Set._hash`.

See also

- OrderedSet recipe [https://code.activestate.com/recipes/576694/] for an example built on `MutableSet`.
- For more about ABCs, see the `abc` module and **PEP 3119** [https://peps.python.org/pep-3119/].

`heapq` — Heap queue algorithm

Source code: Lib/heapq.py
[https://github.com/python/cpython/tree/3.14/Lib/heapq.py]

This module provides an implementation of the heap queue algorithm, also known as the priority queue algorithm.

Min-heaps are binary trees for which every parent node has a value less than or equal to any of its children. We refer to this condition as the heap invariant.

For min-heaps, this implementation uses lists for which `heap[k] <= heap[2*k+1]` and `heap[k] <= heap[2*k+2]` for all *k* for which the compared elements exist. Elements are counted from zero. The interesting property of a min-heap is that its smallest element is always the root, `heap[0]`.

Max-heaps satisfy the reverse invariant: every parent node has a value *greater* than any of its children. These are implemented as lists for which `maxheap[2*k+1] <= maxheap[k]` and `maxheap[2*k+2] <= maxheap[k]` for all *k* for which the compared elements exist. The root, `maxheap[0]`, contains the *largest* element; `heap.sort(reverse=True)` maintains the max-heap invariant.

The **`heapq`** API differs from textbook heap algorithms in two aspects: (a) We use zero-based indexing. This makes the relationship between the index for a node and the indexes for its children slightly less obvious, but is more suitable since Python uses zero-based indexing. (b) Textbooks often focus on max-heaps, due to their suitability for in-place sorting. Our implementation favors min-heaps as they better correspond to Python **`lists`**.

These two aspects make it possible to view the heap as a regular Python list without surprises: `heap[0]` is the smallest item, and `heap.sort()` maintains the heap invariant!

Like **`list.sort()`**, this implementation uses only the `<` operator for comparisons, for both min-heaps and max-heaps.

In the API below, and in this documentation, the unqualified term *heap* generally refers to a min-heap. The API for max-heaps is named using a `_max` suffix.

To create a heap, use a list initialized as `[]`, or transform an existing list into a min-heap or max-heap using the **`heapify()`** or **`heapify_max()`** functions, respectively.

The following functions are provided for min-heaps:

heapq.heapify(*x*)

Transform list *x* into a min-heap, in-place, in linear time.

heapq.heappush(*heap*, *item*)

Push the value *item* onto the *heap*, maintaining the min-heap invariant.

heapq.heappop(*heap*)

Pop and return the smallest item from the *heap*, maintaining the min-heap invariant. If the heap is empty, **`IndexError`** is raised. To access the smallest item without popping it, use `heap[0]`.

heapq.heappushpop(*heap*, *item*)

Push *item* on the heap, then pop and return the smallest item from the *heap*. The combined action runs more efficiently than `heappush()` followed by a separate call to `heappop()`.

heapq.heapreplace(*heap*, *item*)

Pop and return the smallest item from the *heap*, and also push the new *item*. The heap size doesn't change. If the heap is empty, `IndexError` is raised.

This one step operation is more efficient than a `heappop()` followed by `heappush()` and can be more appropriate when using a fixed-size heap. The pop/push combination always returns an element from the heap and replaces it with *item*.

The value returned may be larger than the *item* added. If that isn't desired, consider using `heappushpop()` instead. Its push/pop combination returns the smaller of the two values, leaving the larger value on the heap.

For max-heaps, the following functions are provided:

heapq.heapify_max(*x*)

Transform list *x* into a max-heap, in-place, in linear time.

Added in version 3.14.

heapq.heappush_max(*heap*, *item*)

Push the value *item* onto the max-heap *heap*, maintaining the max-heap invariant.

Added in version 3.14.

heapq.heappop_max(*heap*)

Pop and return the largest item from the max-heap *heap*, maintaining the max-heap invariant. If the max-heap is empty, **`IndexError`** is raised. To access the largest item without popping it, use `maxheap[0]`.

Added in version 3.14.

heapq.heappushpop_max(*heap*, *item*)

Push *item* on the max-heap *heap*, then pop and return the largest item from *heap*. The combined action runs more efficiently than **`heappush_max()`** followed by a separate call to **`heappop_max()`**.

Added in version 3.14.

heapq.heapreplace_max(*heap*, *item*)

Pop and return the largest item from the max-heap *heap* and also push the new *item*. The max-heap size doesn't change. If the max-heap is empty, **`IndexError`** is raised.

The value returned may be smaller than the *item* added. Refer to the analogous function **`heapreplace()`** for detailed usage notes.

Added in version 3.14.

The module also offers three general purpose functions based on heaps.

heapq.merge(**iterables*, *key=None*, *reverse=False*)

Merge multiple sorted inputs into a single sorted output (for example, merge timestamped entries from multiple log files). Returns an iterator over the sorted values.

Similar to `sorted(itertools.chain(*iterables))` but returns an iterable, does not pull the data into memory all at once, and assumes that each of the input streams is already sorted (smallest to largest).

Has two optional arguments which must be specified as keyword arguments.

key specifies a key function of one argument that is used to extract a comparison key from each input element. The default value is `None` (compare the elements directly).

reverse is a boolean value. If set to `True`, then the input elements are merged as if each comparison were reversed. To achieve behavior similar to `sorted(itertools.chain(*iterables), reverse=True)`, all iterables must be sorted from largest to smallest.

Changed in version 3.5: Added the optional *key* and *reverse* parameters.

heapq.nlargest(*n*, *iterable*, *key=None*)

Return a list with the *n* largest elements from the dataset defined by *iterable*. *key*, if provided, specifies a function of one argument that is used to extract a comparison key from each element in *iterable* (for example, `key=str.lower`). Equivalent to: `sorted(iterable, key=key, reverse=True)[:n]`.

heapq.nsmallest(*n*, *iterable*, *key=None*)

Return a list with the *n* smallest elements from the dataset defined by *iterable*. *key*, if provided, specifies a function of one argument that is used to extract a comparison key from each element in *iterable* (for example, `key=str.lower`). Equivalent to: `sorted(iterable, key=key)[:n]`.

The latter two functions perform best for smaller values of *n*. For larger values, it is more efficient to use the **`sorted()`** function. Also, when `n==1`, it is more efficient to use the built-in **`min()`** and **`max()`** functions. If repeated usage of these functions is required, consider turning the iterable into an actual heap.

Basic Examples

A heapsort [https://en.wikipedia.org/wiki/Heapsort] can be implemented by pushing all values onto a heap and then popping off the smallest values one at a time:

```
>>> def heapsort(iterable):
...     h = []
...     for value in iterable:
...         heappush(h, value)
...     return [heappop(h) for i in range(len(h))]
...
>>> heapsort([1, 3, 5, 7, 9, 2, 4, 6, 8, 0])
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

This is similar to `sorted(iterable)`, but unlike **`sorted()`**, this implementation is not stable.

Heap elements can be tuples. This is useful for assigning comparison values (such as task priorities) alongside the main record being tracked:

```
>>> h = []
>>> heappush(h, (5, 'write code'))
>>> heappush(h, (7, 'release product'))
>>> heappush(h, (1, 'write spec'))
>>> heappush(h, (3, 'create tests'))
>>> heappop(h)
(1, 'write spec')
```

Other Applications

Medians [https://en.wikipedia.org/wiki/Median] are a measure of central tendency for a set of numbers. In distributions skewed by outliers, the median provides a more stable estimate than an average (arithmetic mean). A running median is an online algorithm [https://en.wikipedia.org/wiki/Online_algorithm] that updates continuously as new data arrives.

A running median can be efficiently implemented by balancing two heaps, a max-heap for values at or below the midpoint and a min-heap for values above the midpoint. When the two heaps have the same size, the new median is the average of the tops of the two heaps; otherwise, the median is at the top of the larger heap:

```
def running_median(iterable):
    "Yields the cumulative median of values seen so far."

    lo = []  # max-heap
    hi = []  # min-heap (same size as or one smaller than lo)

    for x in iterable:
        if len(lo) == len(hi):
            heappush_max(lo, heappushpop(hi, x))
            yield lo[0]
        else:
            heappush(hi, heappushpop_max(lo, x))
            yield (lo[0] + hi[0]) / 2
```

For example:

```
>>> list(running_median([5.0, 9.0, 4.0, 12.0, 8.0, 9.0]))
[5.0, 7.0, 5.0, 7.0, 8.0, 8.5]
```

Priority Queue Implementation Notes

A priority queue [https://en.wikipedia.org/wiki/Priority_queue] is common use for a heap, and it presents several implementation challenges:

- Sort stability: how do you get two tasks with equal priorities to be returned in the order they were originally added?
- Tuple comparison breaks for (priority, task) pairs if the priorities are equal and the tasks do not have a default comparison order.
- If the priority of a task changes, how do you move it to a new position in the heap?
- Or if a pending task needs to be deleted, how do you find it and remove it from the queue?

A solution to the first two challenges is to store entries as 3-element list including the priority, an entry count, and the task. The entry count serves as a tie-breaker so that two tasks with the same priority are returned in the order they were added. And since no two entry counts are the same, the tuple comparison will never attempt to directly compare two tasks.

Another solution to the problem of non-comparable tasks is to create a wrapper class that ignores the task item and only compares the priority field:

```
from dataclasses import dataclass, field
from typing import Any

@dataclass(order=True)
class PrioritizedItem:
    priority: int
    item: Any=field(compare=False)
```

The remaining challenges revolve around finding a pending task and making changes to its priority or removing it entirely. Finding a task can be done with a dictionary pointing to an entry in the queue.

Removing the entry or changing its priority is more difficult because it would break the heap structure invariants. So, a possible solution is to mark the entry as removed and add a new entry with the revised priority:

```
pq = []                         # list of entries arranged in a
heap
entry_finder = {}               # mapping of tasks to entries
REMOVED = '<removed-task>'      # placeholder for a removed
task
counter = itertools.count()     # unique sequence count

def add_task(task, priority=0):
    'Add a new task or update the priority of an existing task'
    if task in entry_finder:
        remove_task(task)
    count = next(counter)
    entry = [priority, count, task]
    entry_finder[task] = entry
    heappush(pq, entry)

def remove_task(task):
    'Mark an existing task as REMOVED.  Raise KeyError if not
found.'
    entry = entry_finder.pop(task)
    entry[-1] = REMOVED

def pop_task():
    'Remove and return the lowest priority task. Raise KeyError
if empty.'
    while pq:
        priority, count, task = heappop(pq)
        if task is not REMOVED:
            del entry_finder[task]
            return task
    raise KeyError('pop from an empty priority queue')
```

Theory

Heaps are arrays for which `a[k] <= a[2*k+1]` and `a[k] <= a[2*k+2]` for all *k*, counting elements from 0. For the sake of comparison, non-existing elements are considered to be infinite. The interesting property of a heap is that `a[0]` is always its smallest element.

The strange invariant above is meant to be an efficient memory representation for a tournament. The numbers below are *k*, not `a[k]`:

```
                               0

              1                                 2

      3               4                5               6

  7       8       9       10      11      12      13      14

15 16   17 18   19 20   21 22   23 24   25 26   27 28   29 30
```

In the tree above, each cell *k* is topping `2*k+1` and `2*k+2`. In a usual binary tournament we see in sports, each cell is the winner over the two cells it tops, and we can trace the winner down the tree to see all opponents s/he had. However, in many computer applications of such tournaments, we do not need to trace the history of a winner. To be more memory efficient, when a winner is promoted, we try to replace it by something else at a lower level, and the rule becomes that a cell and the two cells it tops contain three different items, but the top cell “wins” over the two topped cells.

If this heap invariant is protected at all time, index 0 is clearly the overall winner. The simplest algorithmic way to remove it and find the “next” winner is to move some loser (let’s say cell 30 in the diagram above) into the 0 position, and then percolate this new 0 down the tree, exchanging values, until the invariant is re-established. This is clearly logarithmic on the total number of items in the tree. By iterating over all items, you get an $O(n \log n)$ sort.

A nice feature of this sort is that you can efficiently insert new items while the sort is going on, provided that the inserted items are not “better” than the last 0’th element you extracted. This is especially useful in simulation contexts, where the tree holds all incoming events, and the “win” condition means the smallest scheduled time. When an event schedules other events for execution, they are scheduled into the future, so they can easily go into the heap. So, a heap is a good structure for implementing schedulers (this is what I used for my MIDI sequencer :-).

Various structures for implementing schedulers have been extensively studied, and heaps are good for this, as they are reasonably speedy, the speed is almost constant, and the worst case is not much different than the average case. However, there are other representations which are more efficient overall, yet the worst cases might be terrible.

Heaps are also very useful in big disk sorts. You most probably all know that a big sort implies producing “runs” (which are pre-sorted sequences, whose size is usually related to the amount of CPU memory), followed by a merging passes for these runs, which merging is often very cleverly organised [[1]]. It is very important that the initial sort produces the longest runs possible. Tournaments are a good way to achieve that. If, using all the memory available to hold a tournament, you replace and percolate items that happen to fit the current run, you’ll produce runs which are twice the size of the memory for random input, and much better for input fuzzily ordered.

Moreover, if you output the 0’th item on disk and get an input which may not fit in the current tournament (because the value “wins” over the last output value), it cannot fit in the heap, so the size of the heap decreases. The freed memory could be cleverly reused immediately for progressively building a second heap, which grows at exactly the same rate the first heap is melting. When the first heap completely vanishes, you switch heaps and start a new run. Clever and quite effective!

In a word, heaps are useful memory structures to know. I use them in a few applications, and I think it is good to keep a ‘heap’ module around. :-)

Footnotes

[1]

The disk balancing algorithms which are current, nowadays, are more annoying than clever, and this is a consequence of the seeking capabilities of the disks. On devices which cannot seek, like big tape drives, the story was quite different, and one had to be very clever to ensure (far in advance) that each tape movement will be the most effective possible (that is, will best participate at “progressing” the merge). Some tapes were even able to read backwards, and this was also used to avoid the rewinding time. Believe me, real good tape sorts were quite spectacular to watch! From all times, sorting has always been a Great Art! :-)

`bisect` — Array bisection algorithm

Source code: Lib/bisect.py
[https://github.com/python/cpython/tree/3.14/Lib/bisect.py]

This module provides support for maintaining a list in sorted order without having to sort the list after each insertion. For long lists of items with expensive comparison operations, this can be an improvement over linear searches or frequent resorting.

The module is called `bisect` because it uses a basic bisection algorithm to do its work. Unlike other bisection tools that search for a specific value, the functions in this module are designed to locate an insertion point. Accordingly, the functions never call an `__eq__()` method to determine whether a value has been found. Instead, the functions only call the `__lt__()` method and will return an insertion point between values in an array.

Note

The functions in this module are not thread-safe. If multiple threads concurrently use `bisect` functions on the same sequence, this may result in undefined behaviour. Likewise, if the provided sequence is mutated by a different thread while a `bisect` function is operating on it, the result is undefined. For example, using `insort_left()` on the same list from multiple threads may result in the list becoming unsorted.

The following functions are provided:

bisect.bisect_left(*a*, *x*, *lo=0*, *hi=len(a)*, ***, *key=None*)

Locate the insertion point for *x* in *a* to maintain sorted order. The parameters *lo* and *hi* may be used to specify a subset of the list which should be considered; by default the entire list is used. If *x* is already present in *a*, the insertion point will be before (to the left of) any existing entries. The return value is suitable for use as the first parameter to `list.insert()` assuming that *a* is already sorted.

The returned insertion point *ip* partitions the array *a* into two slices such that `all(elem < x for elem in a[lo : ip])` is true for the left slice and `all(elem >= x for elem in a[ip : hi])` is true for the right slice.

key specifies a key function of one argument that is used to extract a comparison key from each element in the array. To support searching complex records, the key function is not applied to the *x* value.

If *key* is `None`, the elements are compared directly and no key function is called.

Changed in version 3.10: Added the *key* parameter.

bisect.bisect_right(*a*, *x*, *lo=0*, *hi=len(a)*, ***, *key=None*)
bisect.bisect(*a*, *x*, *lo=0*, *hi=len(a)*, ***, *key=None*)

Similar to **`bisect_left()`**, but returns an insertion point which comes after (to the right of) any existing entries of *x* in *a*.

The returned insertion point *ip* partitions the array *a* into two slices such that `all(elem <= x for elem in a[lo : ip])` is true for the left slice and `all(elem > x for elem in a[ip : hi])` is true for the right slice.

Changed in version 3.10: Added the *key* parameter.

bisect.insort_left(*a*, *x*, *lo=0*, *hi=len(a)*, ***, *key=None*)

Insert *x* in *a* in sorted order.

This function first runs **`bisect_left()`** to locate an insertion point. Next, it runs the **`insert()`** method on *a* to insert *x* at the appropriate position to maintain sort order.

To support inserting records in a table, the *key* function (if any) is applied to *x* for the search step but not for the insertion step.

Keep in mind that the *O*(log *n*) search is dominated by the slow *O*(*n*) insertion step.

Changed in version 3.10: Added the *key* parameter.

bisect.insort_right(*a*, *x*, *lo=0*, *hi=len(a)*, ***, *key=None*)
bisect.insort(*a*, *x*, *lo=0*, *hi=len(a)*, ***, *key=None*)

Similar to **`insort_left()`**, but inserting *x* in *a* after any existing entries of *x*.

This function first runs **`bisect_right()`** to locate an insertion point. Next, it runs the **`insert()`** method on *a* to insert *x* at the appropriate position to maintain sort order.

To support inserting records in a table, the *key* function (if any) is applied to *x* for the search step but not for the insertion step.

Keep in mind that the *O*(log *n*) search is dominated by the slow *O*(*n*) insertion step.

Changed in version 3.10: Added the *key* parameter.

Performance Notes

When writing time sensitive code using *bisect()* and *insort()*, keep these thoughts in mind:

- Bisection is effective for searching ranges of values. For locating specific values, dictionaries are more performant.
- The *insort()* functions are $O(n)$ because the logarithmic search step is dominated by the linear time insertion step.
- The search functions are stateless and discard key function results after they are used. Consequently, if the search functions are used in a loop, the key function may be called again and again on the same array elements. If the key function isn't fast, consider wrapping it with `functools.cache()` to avoid duplicate computations. Alternatively, consider searching an array of precomputed keys to locate the insertion point (as shown in the examples section below).

See also

- Sorted Collections [https://grantjenks.com/docs/sortedcollections/] is a high performance module that uses *bisect* to managed sorted collections of data.
- The SortedCollection recipe [https://code.activestate.com/recipes/577197-sortedcollection/] uses bisect to build a full-featured collection class with straight-forward search methods and support for a key-function. The keys are precomputed to save unnecessary calls to the key function during searches.

Searching Sorted Lists

The above bisect functions are useful for finding insertion points but can be tricky or awkward to use for common searching tasks. The following five functions show how to transform them into the standard lookups for sorted lists:

```
def index(a, x):
    'Locate the leftmost value exactly equal to x'
    i = bisect_left(a, x)
    if i != len(a) and a[i] == x:
        return i
    raise ValueError

def find_lt(a, x):
    'Find rightmost value less than x'
    i = bisect_left(a, x)
    if i:
        return a[i-1]
    raise ValueError

def find_le(a, x):
    'Find rightmost value less than or equal to x'
    i = bisect_right(a, x)
    if i:
        return a[i-1]
    raise ValueError

def find_gt(a, x):
    'Find leftmost value greater than x'
    i = bisect_right(a, x)
    if i != len(a):
        return a[i]
    raise ValueError

def find_ge(a, x):
    'Find leftmost item greater than or equal to x'
    i = bisect_left(a, x)
    if i != len(a):
        return a[i]
    raise ValueError
```

Examples

The `bisect()` function can be useful for numeric table lookups. This example uses `bisect()` to look up a letter grade for an exam score (say) based on a set of ordered numeric breakpoints: 90 and up is an ‘A’, 80 to 89 is a ‘B’, and so on:

```
>>> def grade(score, breakpoints=[60, 70, 80, 90],
grades='FDCBA'):
...     i = bisect(breakpoints, score)
...     return grades[i]
...
>>> [grade(score) for score in [33, 99, 77, 70, 89, 90, 100]]
['F', 'A', 'C', 'C', 'B', 'A', 'A']
```

The `bisect()` and `insort()` functions also work with lists of tuples. The *key* argument can serve to extract the field used for ordering records in a table:

```
>>> from collections import namedtuple
>>> from operator import attrgetter
>>> from bisect import bisect, insort
>>> from pprint import pprint

>>> Movie = namedtuple('Movie', ('name', 'released',
'director'))

>>> movies = [
...     Movie('Jaws', 1975, 'Spielberg'),
...     Movie('Titanic', 1997, 'Cameron'),
...     Movie('The Birds', 1963, 'Hitchcock'),
...     Movie('Aliens', 1986, 'Cameron')
... ]

>>> # Find the first movie released after 1960
>>> by_year = attrgetter('released')
>>> movies.sort(key=by_year)
>>> movies[bisect(movies, 1960, key=by_year)]
Movie(name='The Birds', released=1963, director='Hitchcock')
```

```
>>> # Insert a movie while maintaining sort order
>>> romance = Movie('Love Story', 1970, 'Hiller')
>>> insort(movies, romance, key=by_year)
>>> pprint(movies)
[Movie(name='The Birds', released=1963, director='Hitchcock'),
 Movie(name='Love Story', released=1970, director='Hiller'),
 Movie(name='Jaws', released=1975, director='Spielberg'),
 Movie(name='Aliens', released=1986, director='Cameron'),
 Movie(name='Titanic', released=1997, director='Cameron')]
```

If the key function is expensive, it is possible to avoid repeated function calls by searching a list of precomputed keys to find the index of a record:

```
>>> data = [('red', 5), ('blue', 1), ('yellow', 8), ('black', 
0)]
>>> data.sort(key=lambda r: r[1])       # Or use 
operator.itemgetter(1).
>>> keys = [r[1] for r in data]         # Precompute a list of 
keys.
>>> data[bisect_left(keys, 0)]
('black', 0)
>>> data[bisect_left(keys, 1)]
('blue', 1)
>>> data[bisect_left(keys, 5)]
('red', 5)
>>> data[bisect_left(keys, 8)]
('yellow', 8)
```

`array` — Efficient arrays of numeric values

This module defines an object type which can compactly represent an array of basic values: characters, integers, floating-point numbers. Arrays are mutable sequence types and behave very much like lists, except that the type of objects stored in them is constrained. The type is specified at object creation time by using a *type code*, which is a single character. The following type codes are defined:

Type code	C Type	Python Type	Minimum size in bytes	Notes
`'b'`	signed char	int	1	
`'B'`	unsigned char	int	1	
`'u'`	wchar_t	Unicode character	2	(1)
`'w'`	Py_UCS4	Unicode character	4	(2)
`'h'`	signed short	int	2	

Type code	C Type	Python Type	Minimum size in bytes	Notes
`'H'`	unsigned short	int	2	
`'i'`	signed int	int	2	
`'I'`	unsigned int	int	2	
`'l'`	signed long	int	4	
`'L'`	unsigned long	int	4	
`'q'`	signed long long	int	8	
`'Q'`	unsigned long long	int	8	
`'f'`	float	float	4	
`'d'`	double	float	8	

Notes:

1. It can be 16 bits or 32 bits depending on the platform.

   Changed in version 3.9: `array('u')` now uses **`wchar_t`** as C type instead of deprecated `Py_UNICODE`. This change doesn't affect its behavior because `Py_UNICODE` is alias of **`wchar_t`** since Python 3.3.

   Deprecated since version 3.3, will be removed in version 3.16: Please migrate to `'w'` typecode.

2. *Added in version 3.13.*

The actual representation of values is determined by the machine architecture (strictly speaking, by the C implementation). The actual size can be accessed through the **`array.itemsize`** attribute.

The module defines the following item:

array.typecodes

A string with all available type codes.

The module defines the following type:

class array.array(*typecode*[, *initializer*])

A new array whose items are restricted by *typecode*, and initialized from the optional *initializer* value, which must be a **`bytes`** or **`bytearray`** object, a Unicode string, or iterable over elements of the appropriate type.

If given a **`bytes`** or **`bytearray`** object, the initializer is passed to the new array's **`frombytes()`** method; if given a Unicode string, the initializer is passed to the **`fromunicode()`** method; otherwise, the initializer's iterator is passed to the **`extend()`** method to add initial items to the array.

Array objects support the ordinary mutable sequence operations of indexing, slicing, concatenation, and multiplication. When using slice assignment, the assigned value must be an array object with the same type code; in all other cases, **`TypeError`** is raised. Array objects also implement the buffer interface, and may be used wherever bytes-like objects are supported.

Raises an auditing event `array.__new__` with arguments `typecode`, `initializer`.

typecode

The typecode character used to create the array.

itemsize

The length in bytes of one array item in the internal representation.

append(*x*)

Append a new item with value *x* to the end of the array.

buffer_info()

Return a tuple `(address, length)` giving the current memory address and the length in elements of the buffer used to hold array’s contents. The size of the memory buffer in bytes can be computed as `array.buffer_info()[1] * array.itemsize`. This is occasionally useful when working with low-level (and inherently unsafe) I/O interfaces that require memory addresses, such as certain **`ioctl()`** operations. The returned numbers are valid as long as the array exists and no length-changing operations are applied to it.

Note

When using array objects from code written in C or C++ (the only way to effectively make use of this information), it makes more sense to use the buffer interface supported by array objects. This method is maintained for backward compatibility and should be avoided in new code. The buffer interface is documented in Buffer Protocol.

byteswap()

“Byteswap” all items of the array. This is only supported for values which are 1, 2, 4, or 8 bytes in size; for other types of values, **`RuntimeError`** is raised. It is useful when reading data from a file written on a machine with a different byte order.

count(*x*)

Return the number of occurrences of *x* in the array.

extend(*iterable*)

Append items from *iterable* to the end of the array. If *iterable* is another array, it must have *exactly* the same type code; if not, **`TypeError`** will be raised. If *iterable* is not an array, it must be iterable and its elements must be the right type to be appended to the array.

frombytes(*buffer*)

Appends items from the bytes-like object, interpreting its content as an array of machine values (as if it had been read from a file using the **`fromfile()`** method).

Added in version 3.2: **`fromstring()`** is renamed to **`frombytes()`** for clarity.

fromfile(*f*, *n*)

Read *n* items (as machine values) from the file object *f* and append them to the end of the array. If less than *n* items are available, **`EOFError`** is raised, but the items that were available are still inserted into the array.

fromlist(*list*)

Append items from the list. This is equivalent to `for x in list: a.append(x)` except that if there is a type error, the array is unchanged.

fromunicode(*s*)

Extends this array with data from the given Unicode string. The array must have type code `'u'` or `'w'`; otherwise a **`ValueError`** is raised. Use `array.frombytes(unicodestring.encode(enc))` to append Unicode data to an array of some other type.

index(*x*[, *start*[, *stop*]])

Return the smallest *i* such that *i* is the index of the first occurrence of *x* in the array. The optional arguments *start* and *stop* can be specified to search for *x* within a subsection of the array. Raise **`ValueError`** if *x* is not found.

Changed in version 3.10: Added optional *start* and *stop* parameters.

insert(*i*, *x*)

Insert a new item with value *x* in the array before position *i*. Negative values are treated as being relative to the end of the array.

pop([*i*])

Removes the item with the index *i* from the array and returns it. The optional argument defaults to `-1`, so that by default the last item is removed and returned.

remove(*x*)

Remove the first occurrence of *x* from the array.

clear()

Remove all elements from the array.

Added in version 3.13.

reverse()

Reverse the order of the items in the array.

tobytes()

Convert the array to an array of machine values and return the bytes representation (the same sequence of bytes that would be written to a file by the **`tofile()`** method.)

Added in version 3.2: **`tostring()`** is renamed to **`tobytes()`** for clarity.

tofile(*f*)

Write all items (as machine values) to the file object *f*.

tolist()

Convert the array to an ordinary list with the same items.

tounicode()

Convert the array to a Unicode string. The array must have a type `'u'` or `'w'`; otherwise a **`ValueError`** is raised.

> Use `array.tobytes().decode(enc)` to obtain a Unicode string from an array of some other type.

The string representation of array objects has the form `array(typecode, initializer)`. The *initializer* is omitted if the array is empty, otherwise it is a Unicode string if the *typecode* is `'u'` or `'w'`, otherwise it is a list of numbers. The string representation is guaranteed to be able to be converted back to an array with the same type and value using **`eval()`**, so long as the **`array`** class has been imported using `from array import array`. Variables `inf` and `nan` must also be defined if it contains corresponding floating-point values. Examples:

```
array('l')
array('w', 'hello \u2641')
array('l', [1, 2, 3, 4, 5])
array('d', [1.0, 2.0, 3.14, -inf, nan])
```

See also

Module `struct`
Packing and unpacking of heterogeneous binary data.

NumPy [https://numpy.org/]
The NumPy package defines another array type.

`weakref` — Weak references

Source code: Lib/weakref.py
[https://github.com/python/cpython/tree/3.14/Lib/weakref.py]

The `weakref` module allows the Python programmer to create *weak references* to objects.

In the following, the term *referent* means the object which is referred to by a weak reference.

A weak reference to an object is not enough to keep the object alive: when the only remaining references to a referent are weak references, garbage collection is free to destroy the referent and reuse its memory for something else. However, until the object is actually destroyed the weak reference may return the object even if there are no strong references to it.

A primary use for weak references is to implement caches or mappings holding large objects, where it’s desired that a large object not be kept alive solely because it appears in a cache or mapping.

For example, if you have a number of large binary image objects, you may wish to associate a name with each. If you used a Python dictionary to map names to images, or images to names, the image objects would remain alive just because they appeared as values or keys in the dictionaries. The `WeakKeyDictionary` and `WeakValueDictionary` classes supplied by the `weakref` module are an alternative, using weak references to construct mappings that don’t keep objects alive solely because they appear in the mapping objects. If, for example, an image object is a value in a `WeakValueDictionary`, then when the last remaining references to that image object are the weak references held by weak mappings, garbage collection can reclaim the object, and its corresponding entries in weak mappings are simply deleted.

`WeakKeyDictionary` and `WeakValueDictionary` use weak references in their implementation, setting up callback functions on the weak references that notify the weak dictionaries when a key or value has been reclaimed by garbage collection. `WeakSet` implements the `set` interface, but keeps weak references to its elements, just like a `WeakKeyDictionary` does.

`finalize` provides a straight forward way to register a cleanup function to be called when an object is garbage collected. This is simpler to use than setting up a callback function on a raw weak reference, since the module automatically ensures that the finalizer remains alive until the object is collected.

Most programs should find that using one of these weak container types or `finalize` is all they need – it's not usually necessary to create your own weak references directly. The low-level machinery is exposed by the `weakref` module for the benefit of advanced uses.

Not all objects can be weakly referenced. Objects which support weak references include class instances, functions written in Python (but not in C), instance methods, sets, frozensets, some file objects, generators, type objects, sockets, arrays, deques, regular expression pattern objects, and code objects.

Changed in version 3.2: Added support for thread.lock, threading.Lock, and code objects.

Several built-in types such as `list` and `dict` do not directly support weak references but can add support through subclassing:

```
class Dict(dict):
    pass

obj = Dict(red=1, green=2, blue=3)   # this object is weak
referenceable
```

CPython implementation detail: Other built-in types such as `tuple` and `int` do not support weak references even when subclassed.

Extension types can easily be made to support weak references; see Weak Reference Support.

When `__slots__` are defined for a given type, weak reference support is disabled unless a `'__weakref__'` string is also present in the sequence of strings in the `__slots__` declaration. See __slots__ documentation for details.

class weakref.ref(*object*[, *callback*])

Return a weak reference to *object*. The original object can be retrieved by calling the reference object if the referent is still alive; if the referent is no longer alive, calling the reference object will cause `None` to be returned. If *callback* is provided and not `None`, and the returned weakref object is still alive, the callback will be called when the object is about to be finalized; the weak reference object will be passed as the only parameter to the callback; the referent will no longer be available.

It is allowable for many weak references to be constructed for the same object. Callbacks registered for each weak reference will be called from the most recently registered callback to the oldest registered callback.

Exceptions raised by the callback will be noted on the standard error output, but cannot be propagated; they are handled in exactly the same way as exceptions raised from an object’s `__del__()` method.

Weak references are hashable if the *object* is hashable. They will maintain their hash value even after the *object* was deleted. If `hash()` is called the first time only after the *object* was deleted, the call will raise `TypeError`.

Weak references support tests for equality, but not ordering. If the referents are still alive, two references have the same equality relationship as their referents (regardless of the *callback*). If either referent has been deleted, the references are equal only if the reference objects are the same object.

This is a subclassable type rather than a factory function.

__callback__

This read-only attribute returns the callback currently associated to the weakref. If there is no callback or if the referent of the weakref is no longer alive then this attribute will have value `None`.

Changed in version 3.4: Added the `__callback__` attribute.

weakref.proxy(*object*[, *callback*])

Return a proxy to *object* which uses a weak reference. This supports use of the proxy in most contexts instead of requiring the explicit dereferencing used with weak reference objects. The returned object will have a type of either `ProxyType` or `CallableProxyType`, depending on whether *object* is callable. Proxy objects are not hashable regardless of the referent; this avoids a number of problems related to their fundamentally mutable nature, and prevents their use as dictionary keys. *callback* is the same as the parameter of the same name to the `ref()` function.

Accessing an attribute of the proxy object after the referent is garbage collected raises `ReferenceError`.

Changed in version 3.8: Extended the operator support on proxy objects to include the matrix multiplication operators `@` and `@=`.

weakref.getweakrefcount(*object*)

Return the number of weak references and proxies which refer to *object*.

weakref.getweakrefs(*object*)

Return a list of all weak reference and proxy objects which refer to *object*.

class weakref.WeakKeyDictionary([*dict*])

Mapping class that references keys weakly. Entries in the dictionary will be discarded when there is no longer a strong reference to the key. This can be used to associate additional data with an object owned by other parts of an application without adding attributes to those objects. This can be especially useful with objects that override attribute accesses.

Note that when a key with equal value to an existing key (but not equal identity) is inserted into the dictionary, it replaces the value but does not replace the existing key.

Due to this, when the reference to the original key is deleted, it also deletes the entry in the dictionary:

```
>>> class T(str): pass
...
>>> k1, k2 = T(), T()
>>> d = weakref.WeakKeyDictionary()
>>> d[k1] = 1   # d = {k1: 1}
>>> d[k2] = 2   # d = {k1: 2}
>>> del k1      # d = {}
```

A workaround would be to remove the key prior to reassignment:

```
>>> class T(str): pass
...
>>> k1, k2 = T(), T()
>>> d = weakref.WeakKeyDictionary()
>>> d[k1] = 1   # d = {k1: 1}
>>> del d[k1]
>>> d[k2] = 2   # d = {k2: 2}
>>> del k1      # d = {k2: 2}
```

Changed in version 3.9: Added support for `|` and `|=` operators, as specified in **PEP 584** [https://peps.python.org/pep-0584/].

`WeakKeyDictionary` objects have an additional method that exposes the internal references directly. The references are not guaranteed to be “live” at the time they are used, so the result of calling the references needs to be checked before being used. This can be used to avoid creating references that will cause the garbage collector to keep the keys around longer than needed.

WeakKeyDictionary.keyrefs()

Return an iterable of the weak references to the keys.

class weakref.WeakValueDictionary([*dict*])

Mapping class that references values weakly. Entries in the dictionary will be discarded when no strong reference to the value exists any more.

Changed in version 3.9: Added support for `|` and `|=` operators, as specified in **PEP 584** [https://peps.python.org/pep-0584/].

`WeakValueDictionary` objects have an additional method that has the same issues as the `WeakKeyDictionary.keyrefs()` method.

WeakValueDictionary.valuerefs()

Return an iterable of the weak references to the values.

class weakref.WeakSet([*elements*])

Set class that keeps weak references to its elements. An element will be discarded when no strong reference to it exists any more.

class weakref.WeakMethod(*method*[, *callback*])

A custom `ref` subclass which simulates a weak reference to a bound method (i.e., a method defined on a class and looked up on an instance). Since a bound method is ephemeral, a standard weak reference cannot keep hold of it. `WeakMethod` has special code to recreate the bound method until either the object or the original function dies:

```
>>> class C:
...     def method(self):
...         print("method called!")
...
>>> c = C()
>>> r = weakref.ref(c.method)
>>> r()
>>> r = weakref.WeakMethod(c.method)
>>> r()
<bound method C.method of <__main__.C object at
0x7fc859830220>>
>>> r()()
method called!
>>> del c
>>> gc.collect()
0
>>> r()
>>>
```

callback is the same as the parameter of the same name to the `ref()` function.

Added in version 3.4.

class weakref.finalize(*obj*, *func*, */*, **args*, ***kwargs*)

Return a callable finalizer object which will be called when *obj* is garbage collected. Unlike an ordinary weak reference, a finalizer will always survive until the reference object is collected, greatly simplifying lifecycle management.

A finalizer is considered *alive* until it is called (either explicitly or at garbage collection), and after that it is *dead*. Calling a live finalizer returns the result of evaluating `func(*arg, **kwargs)`, whereas calling a dead finalizer returns `None`.

Exceptions raised by finalizer callbacks during garbage collection will be shown on the standard error output, but cannot be propagated. They are handled in the same way as exceptions raised from an object's `__del__()` method or a weak reference's callback.

When the program exits, each remaining live finalizer is called unless its `atexit` attribute has been set to false. They are called in reverse order of creation.

A finalizer will never invoke its callback during the later part of the interpreter shutdown when module globals are liable to have been replaced by `None`.

__call__()

If *self* is alive then mark it as dead and return the result of calling `func(*args, **kwargs)`. If *self* is dead then return `None`.

detach()

If *self* is alive then mark it as dead and return the tuple `(obj, func, args, kwargs)`. If *self* is dead then return `None`.

peek()

If *self* is alive then return the tuple `(obj, func, args, kwargs)`. If *self* is dead then return `None`.

alive

Property which is true if the finalizer is alive, false otherwise.

atexit

A writable boolean property which by default is true. When the program exits, it calls all remaining live finalizers for which `atexit` is true. They are called in reverse order of creation.

Note

It is important to ensure that *func*, *args* and *kwargs* do not own any references to *obj*, either directly or indirectly, since otherwise *obj* will never be garbage collected. In particular, *func* should not be a bound method of *obj*.

Added in version 3.4.

weakref.ReferenceType

The type object for weak references objects.

weakref.ProxyType

The type object for proxies of objects which are not callable.

weakref.CallableProxyType

The type object for proxies of callable objects.

weakref.ProxyTypes

Sequence containing all the type objects for proxies. This can make it simpler to test if an object is a proxy without being dependent on naming both proxy types.

See also

PEP 205 [https://peps.python.org/pep-0205/] - Weak References

The proposal and rationale for this feature, including links to earlier implementations and information about similar features in other languages.

Weak Reference Objects

Weak reference objects have no methods and no attributes besides `ref.__callback__`. A weak reference object allows the referent to be obtained, if it still exists, by calling it:

```
>>> import weakref
>>> class Object:
...     pass
...
```

```
>>> o = Object()
>>> r = weakref.ref(o)
>>> o2 = r()
>>> o is o2
True
```

If the referent no longer exists, calling the reference object returns **None**:

```
>>> del o, o2
>>> print(r())
None
```

Testing that a weak reference object is still live should be done using the expression `ref() is not None`. Normally, application code that needs to use a reference object should follow this pattern:

```
# r is a weak reference object
o = r()
if o is None:
    # referent has been garbage collected
    print("Object has been deallocated; can't frobnicate.")
else:
    print("Object is still live!")
    o.do_something_useful()
```

Using a separate test for “liveness” creates race conditions in threaded applications; another thread can cause a weak reference to become invalidated before the weak reference is called; the idiom shown above is safe in threaded applications as well as single-threaded applications.

Specialized versions of **ref** objects can be created through subclassing. This is used in the implementation of the **WeakValueDictionary** to reduce the memory overhead for each entry in the mapping. This may be most useful to associate additional information with a reference, but could also be used to insert additional processing on calls to retrieve the referent.

This example shows how a subclass of `ref` can be used to store additional information about an object and affect the value that’s returned when the referent is accessed:

```
import weakref

class ExtendedRef(weakref.ref):
    def __init__(self, ob, callback=None, /, **annotations):
        super().__init__(ob, callback)
        self.__counter = 0
        for k, v in annotations.items():
            setattr(self, k, v)

    def __call__(self):
        """Return a pair containing the referent and the number
of
        times the reference has been called.
        """
        ob = super().__call__()
        if ob is not None:
            self.__counter += 1
            ob = (ob, self.__counter)
        return ob
```

Example

This simple example shows how an application can use object IDs to retrieve objects that it has seen before. The IDs of the objects can then be used in other data structures without forcing the objects to remain alive, but the objects can still be retrieved by ID if they do.

```
import weakref

_id2obj_dict = weakref.WeakValueDictionary()

def remember(obj):
    oid = id(obj)
    _id2obj_dict[oid] = obj
    return oid

def id2obj(oid):
    return _id2obj_dict[oid]
```

Finalizer Objects

The main benefit of using `finalize` is that it makes it simple to register a callback without needing to preserve the returned finalizer object. For instance

```
>>> import weakref
>>> class Object:
...     pass
...
>>> kenny = Object()
>>> weakref.finalize(kenny, print, "You killed Kenny!")
<finalize object at ...; for 'Object' at ...>
>>> del kenny
You killed Kenny!
```

The finalizer can be called directly as well. However the finalizer will invoke the callback at most once.

```
>>> def callback(x, y, z):
...     print("CALLBACK")
...     return x + y + z
...
>>> obj = Object()
>>> f = weakref.finalize(obj, callback, 1, 2, z=3)
>>> assert f.alive
>>> assert f() == 6
CALLBACK
>>> assert not f.alive
>>> f()                     # callback not called because
finalizer dead
>>> del obj                 # callback not called because
finalizer dead
```

You can unregister a finalizer using its `detach()` method. This kills the finalizer and returns the arguments passed to the constructor when it was created.

```
>>> obj = Object()
>>> f = weakref.finalize(obj, callback, 1, 2, z=3)
>>> f.detach()
(<...Object object ...>, <function callback ...>, (1, 2), {'z':
3})
>>> newobj, func, args, kwargs = _
>>> assert not f.alive
>>> assert newobj is obj
>>> assert func(*args, **kwargs) == 6
CALLBACK
```

Unless you set the `atexit` attribute to `False`, a finalizer will be called when the program exits if it is still alive. For instance

```
>>> obj = Object()
>>> weakref.finalize(obj, print, "obj dead or exiting")
<finalize object at ...; for 'Object' at ...>
>>> exit()
obj dead or exiting
```

Comparing finalizers with `__del__()` methods

Suppose we want to create a class whose instances represent temporary directories. The directories should be deleted with their contents when the first of the following events occurs:

- the object is garbage collected,
- the object’s **`remove()`** method is called, or
- the program exits.

We might try to implement the class using a `__del__()` method as follows:

```
class TempDir:
    def __init__(self):
        self.name = tempfile.mkdtemp()

    def remove(self):
        if self.name is not None:
            shutil.rmtree(self.name)
            self.name = None

    @property
    def removed(self):
        return self.name is None

    def __del__(self):
        self.remove()
```

Starting with Python 3.4, `__del__()` methods no longer prevent reference cycles from being garbage collected, and module globals are no longer forced to `None` during interpreter shutdown. So this code should work without any issues on CPython.

However, handling of `__del__()` methods is notoriously implementation specific, since it depends on internal details of the interpreter's garbage collector implementation.

A more robust alternative can be to define a finalizer which only references the specific functions and objects that it needs, rather than having access to the full state of the object:

```
class TempDir:
    def __init__(self):
        self.name = tempfile.mkdtemp()
        self._finalizer = weakref.finalize(self, shutil.rmtree, 
self.name)

    def remove(self):
        self._finalizer()
    @property
    def removed(self):
        return not self._finalizer.alive
```

Defined like this, our finalizer only receives a reference to the details it needs to clean up the directory appropriately. If the object never gets garbage collected the finalizer will still be called at exit.

The other advantage of weakref based finalizers is that they can be used to register finalizers for classes where the definition is controlled by a third party, such as running code when a module is unloaded:

```
import weakref, sys
def unloading_module():
    # implicit reference to the module globals from the
function body
weakref.finalize(sys.modules[__name__], unloading_module)
```

Note

If you create a finalizer object in a daemonic thread just as the program exits then there is the possibility that the finalizer does not get called at exit. However, in a daemonic thread `atexit.register()`, `try: ... finally: ...` and `with: ...` do not guarantee that cleanup occurs either.

`types` — Dynamic type creation and names for built-in types

Source code: Lib/types.py
[https://github.com/python/cpython/tree/3.14/Lib/types.py]

This module defines utility functions to assist in dynamic creation of new types.

It also defines names for some object types that are used by the standard Python interpreter, but not exposed as builtins like **`int`** or **`str`** are.

Finally, it provides some additional type-related utility classes and functions that are not fundamental enough to be builtins.

Dynamic Type Creation

types.new_class(*name*, *bases=()*, *kwds=None*, *exec_body=None*)

Creates a class object dynamically using the appropriate metaclass.

The first three arguments are the components that make up a class definition header: the class name, the base classes (in order), the keyword arguments (such as `metaclass`).

The *exec_body* argument is a callback that is used to populate the freshly created class namespace. It should accept the class namespace as its sole argument and update the namespace directly with the class contents. If no callback is provided, it has the same effect as passing in `lambda ns: None`.

Added in version 3.3.

types.prepare_class(*name*, *bases=()*, *kwds=None*)

Calculates the appropriate metaclass and creates the class namespace.

The arguments are the components that make up a class definition header: the class name, the base classes (in order) and the keyword arguments (such as `metaclass`).

The return value is a 3-tuple: `metaclass, namespace, kwds`

metaclass is the appropriate metaclass, *namespace* is the prepared class namespace and *kwds* is an updated copy of the passed in *kwds* argument with any `'metaclass'` entry removed. If no *kwds* argument is passed in, this will be an empty dict.

Added in version 3.3.

Changed in version 3.6: The default value for the `namespace` element of the returned tuple has changed. Now an insertion-order-preserving mapping is used when the metaclass does not have a `__prepare__` method.

See also

Metaclasses

Full details of the class creation process supported by these functions

PEP 3115 [https://peps.python.org/pep-3115/] - Metaclasses in Python 3000

Introduced the `__prepare__` namespace hook

types.resolve_bases(*bases*)

Resolve MRO entries dynamically as specified by **PEP 560** [https://peps.python.org/pep-0560/].

This function looks for items in *bases* that are not instances of `type`, and returns a tuple where each such object that has an `__mro_entries__()` method is replaced with an unpacked result of calling this method. If a *bases* item is an instance of `type`, or it doesn't have an `__mro_entries__()` method, then it is included in the return tuple unchanged.

Added in version 3.7.

types.get_original_bases(*cls*, */*)

Return the tuple of objects originally given as the bases of *cls* before the `__mro_entries__()` method has been called on any bases (following the mechanisms laid out in **PEP 560** [https://peps.python.org/pep-0560/]). This is useful for introspecting Generics.

For classes that have an `__orig_bases__` attribute, this function returns the value of `cls.__orig_bases__`. For classes without the `__orig_bases__` attribute, `cls.__bases__` is returned.

Examples:

```
from typing import TypeVar, Generic, NamedTuple, TypedDict

T = TypeVar("T")
class Foo(Generic[T]): ...
class Bar(Foo[int], float): ...
class Baz(list[str]): ...
Eggs = NamedTuple("Eggs", [("a", int), ("b", str)])
Spam = TypedDict("Spam", {"a": int, "b": str})

assert Bar.__bases__ == (Foo, float)
assert get_original_bases(Bar) == (Foo[int], float)

assert Baz.__bases__ == (list,)
assert get_original_bases(Baz) == (list[str],)

assert Eggs.__bases__ == (tuple,)
assert get_original_bases(Eggs) == (NamedTuple,)

assert Spam.__bases__ == (dict,)
assert get_original_bases(Spam) == (TypedDict,)

assert int.__bases__ == (object,)
assert get_original_bases(int) == (object,)
```

Added in version 3.12.

See also

PEP 560 [https://peps.python.org/pep-0560/] - Core support for typing module and generic types

Standard Interpreter Types

This module provides names for many of the types that are required to implement a Python interpreter. It deliberately avoids including some of the types that arise only incidentally during processing such as the `listiterator` type.

Typical use of these names is for **`isinstance()`** or **`issubclass()`** checks.

If you instantiate any of these types, note that signatures may vary between Python versions.

Standard names are defined for the following types:

types.NoneType

The type of **`None`**.

Added in version 3.10.

types.FunctionType
types.LambdaType

The type of user-defined functions and functions created by **`lambda`** expressions.

Raises an auditing event `function.__new__` with argument `code`.

The audit event only occurs for direct instantiation of function objects, and is not raised for normal compilation.

types.GeneratorType

The type of generator-iterator objects, created by generator functions.

types.CoroutineType

The type of coroutine objects, created by **`async def`** functions.

Added in version 3.5.

types.AsyncGeneratorType

The type of asynchronous generator-iterator objects, created by asynchronous generator functions.

Added in version 3.6.

class types.CodeType(***kwargs*)

The type of code objects such as returned by **`compile()`**.

Raises an auditing event `code.__new__` with arguments `code`, `filename`, `name`, `argcount`, `posonlyargcount`, `kwonlyargcount`, `nlocals`, `stacksize`, `flags`.

Note that the audited arguments may not match the names or positions required by the initializer. The audit event only occurs for direct instantiation of code objects, and is not raised for normal compilation.

types.CellType

The type for cell objects: such objects are used as containers for a function’s closure variables.

Added in version 3.8.

types.MethodType

The type of methods of user-defined class instances.

types.BuiltinFunctionType
types.BuiltinMethodType

The type of built-in functions like **`len()`** or **`sys.exit()`**, and methods of built-in classes. (Here, the term “built-in” means “written in C”.)

types.WrapperDescriptorType

The type of methods of some built-in data types and base classes such as **`object.__init__()`** or **`object.__lt__()`**.

Added in version 3.7.

types.MethodWrapperType

The type of *bound* methods of some built-in data types and base classes. For example it is the type of `object().__str__`.

Added in version 3.7.

types.NotImplementedType

The type of **`NotImplemented`**.

Added in version 3.10.

types.MethodDescriptorType

The type of methods of some built-in data types such as **`str.join()`**.

Added in version 3.7.

types.ClassMethodDescriptorType

The type of *unbound* class methods of some built-in data types such as `dict.__dict__['fromkeys']`.

Added in version 3.7.

class types.ModuleType(*name*, *doc=None*)

The type of modules. The constructor takes the name of the module to be created and optionally its docstring.

See also

Documentation on module objects

Provides details on the special attributes that can be found on instances of **`ModuleType`**.

`importlib.util.module_from_spec()`

Modules created using the **`ModuleType`** constructor are created with many of their special attributes unset or set to default values. **`module_from_spec()`** provides a more robust way of creating **`ModuleType`** instances which ensures the various attributes are set appropriately.

types.EllipsisType

The type of **`Ellipsis`**.

Added in version 3.10.

class types.GenericAlias(*t_origin*, *t_args*)

The type of parameterized generics such as `list[int]`.

`t_origin` should be a non-parameterized generic class, such as `list`, `tuple` or `dict`. `t_args` should be a **`tuple`** (possibly of length 1) of types which parameterize `t_origin`:

```
>>> from types import GenericAlias

>>> list[int] == GenericAlias(list, (int,))
True
>>> dict[str, int] == GenericAlias(dict, (str, int))
True
```

Added in version 3.9.

Changed in version 3.9.2: This type can now be subclassed.

See also

Generic Alias Types

In-depth documentation on instances of **`types.GenericAlias`**

PEP 585 [https://peps.python.org/pep-0585/] - Type Hinting Generics In Standard Collections

Introducing the **`types.GenericAlias`** class

class types.UnionType

The type of union type expressions.

Added in version 3.10.

Changed in version 3.14: This is now an alias for **`typing.Union`**.

class types.TracebackType(*tb_next*, *tb_frame*, *tb_lasti*, *tb_lineno*)

The type of traceback objects such as found in `sys.exception().__traceback__`.

See the language reference for details of the available attributes and operations, and guidance on creating tracebacks dynamically.

types.FrameType

The type of frame objects such as found in **`tb.tb_frame`** if `tb` is a traceback object.

types.GetSetDescriptorType

The type of objects defined in extension modules with `PyGetSetDef`, such as **`FrameType.f_locals`** or `array.array.typecode`. This type is used as descriptor for object attributes; it has the same purpose as the **`property`** type, but for classes defined in extension modules.

types.MemberDescriptorType

The type of objects defined in extension modules with `PyMemberDef`, such as `datetime.timedelta.days`. This type is used as descriptor for simple C data members which use standard conversion functions; it has the same purpose as the `property` type, but for classes defined in extension modules.

In addition, when a class is defined with a `__slots__` attribute, then for each slot, an instance of **`MemberDescriptorType`** will be added as an attribute on the class. This allows the slot to appear in the class's `__dict__`.

CPython implementation detail: In other implementations of Python, this type may be identical to `GetSetDescriptorType`.

class types.MappingProxyType(*mapping*)

Read-only proxy of a mapping. It provides a dynamic view on the mapping's entries, which means that when the mapping changes, the view reflects these changes.

Added in version 3.3.

Changed in version 3.9: Updated to support the new union (`|`) operator from **PEP 584** [https://peps.python.org/pep-0584/], which simply delegates to the underlying mapping.

key in proxy

Return `True` if the underlying mapping has a key *key*, else `False`.

proxy[key]

Return the item of the underlying mapping with key *key*. Raises a `KeyError` if *key* is not in the underlying mapping.

iter(proxy)

Return an iterator over the keys of the underlying mapping. This is a shortcut for `iter(proxy.keys())`.

len(proxy)

Return the number of items in the underlying mapping.

copy()

Return a shallow copy of the underlying mapping.

get(*key*[, *default*])

Return the value for *key* if *key* is in the underlying mapping, else *default*. If *default* is not given, it defaults to `None`, so that this method never raises a [`KeyError`](#).

items()

Return a new view of the underlying mapping's items (`(key, value)` pairs).

keys()

Return a new view of the underlying mapping's keys.

values()

Return a new view of the underlying mapping's values.

reversed(proxy)

Return a reverse iterator over the keys of the underlying mapping.

Added in version 3.9.

hash(proxy)

Return a hash of the underlying mapping.

Added in version 3.12.

class types.CapsuleType

The type of capsule objects.

Added in version 3.13.

Additional Utility Classes and Functions

class types.SimpleNamespace

A simple **`object`** subclass that provides attribute access to its namespace, as well as a meaningful repr.

Unlike **`object`**, with **`SimpleNamespace`** you can add and remove attributes.

`SimpleNamespace` objects may be initialized in the same way as **`dict`**: either with keyword arguments, with a single positional argument, or with both. When initialized with keyword arguments, those are directly added to the underlying namespace. Alternatively, when initialized with a positional argument, the underlying namespace will be updated with key-value pairs from that argument (either a mapping object or an iterable object producing key-value pairs). All such keys must be strings.

The type is roughly equivalent to the following code:

```
class SimpleNamespace:
    def __init__(self, mapping_or_iterable=(), /, **kwargs):
        self.__dict__.update(mapping_or_iterable)
        self.__dict__.update(kwargs)

    def __repr__(self):
        items = (f"{k}={v!r}" for k, v in
self.__dict__.items())
        return "{}({})".format(type(self).__name__, ",
".join(items))
```

```
    def __eq__(self, other):
        if isinstance(self, SimpleNamespace) and isinstance(other, SimpleNamespace):
           return self.__dict__ == other.__dict__
        return NotImplemented
```

`SimpleNamespace` may be useful as a replacement for `class NS: pass`. However, for a structured record type use [`namedtuple()`] instead.

`SimpleNamespace` objects are supported by [`copy.replace()`].

Added in version 3.3.

Changed in version 3.9: Attribute order in the repr changed from alphabetical to insertion (like `dict`).

Changed in version 3.13: Added support for an optional positional argument.

types.DynamicClassAttribute(*fget=None*, *fset=None*, *fdel=None*, *doc=None*)

Route attribute access on a class to __getattr__.

This is a descriptor, used to define attributes that act differently when accessed through an instance and through a class. Instance access remains normal, but access to an attribute through a class will be routed to the class’s __getattr__ method; this is done by raising AttributeError.

This allows one to have properties active on an instance, and have virtual attributes on the class with the same name (see [`enum.Enum`] for an example).

Added in version 3.4.

Coroutine Utility Functions

types.coroutine(*gen_func*)

This function transforms a generator function into a coroutine function which returns a generator-based coroutine. The generator-based coroutine is still a generator iterator, but is also considered to be a coroutine object and is awaitable. However, it may not necessarily implement the `__await__()` method.

If *gen_func* is a generator function, it will be modified in-place.

If *gen_func* is not a generator function, it will be wrapped. If it returns an instance of `collections.abc.Generator`, the instance will be wrapped in an *awaitable* proxy object. All other types of objects will be returned as is.

Added in version 3.5.

`copy` — Shallow and deep copy operations

Source code: Lib/copy.py
[https://github.com/python/cpython/tree/3.14/Lib/copy.py]

Assignment statements in Python do not copy objects, they create bindings between a target and an object. For collections that are mutable or contain mutable items, a copy is sometimes needed so one can change one copy without changing the other. This module provides generic shallow and deep copy operations (explained below).

Interface summary:

copy.copy(*obj*)

Return a shallow copy of *obj*.

copy.deepcopy(*obj*[, *memo*])

Return a deep copy of *obj*.

copy.replace(*obj*, /, ***changes*)

Creates a new object of the same type as *obj*, replacing fields with values from *changes*.

Added in version 3.13.

exception copy.Error

Raised for module specific errors.

The difference between shallow and deep copying is only relevant for compound objects (objects that contain other objects, like lists or class instances):

- A *shallow copy* constructs a new compound object and then (to the extent possible) inserts *references* into it to the objects found in the original.
- A *deep copy* constructs a new compound object and then, recursively, inserts *copies* into it of the objects found in the original.

Two problems often exist with deep copy operations that don't exist with shallow copy operations:

- Recursive objects (compound objects that, directly or indirectly, contain a reference to themselves) may cause a recursive loop.
- Because deep copy copies everything it may copy too much, such as data which is intended to be shared between copies.

The **`deepcopy()`** function avoids these problems by:

- keeping a `memo` dictionary of objects already copied during the current copying pass; and
- letting user-defined classes override the copying operation or the set of components copied.

This module does not copy types like module, method, stack trace, stack frame, file, socket, window, or any similar types. It does "copy" functions and classes (shallow and deeply), by returning the original object unchanged; this is compatible with the way these are treated by the **`pickle`** module.

Shallow copies of dictionaries can be made using **`dict.copy()`**, and of lists by assigning a slice of the entire list, for example, `copied_list = original_list[:]`.

Classes can use the same interfaces to control copying that they use to control pickling. See the description of module **`pickle`** for information on these methods. In fact, the **`copy`** module uses the registered pickle functions from the **`copyreg`** module.

In order for a class to define its own copy implementation, it can define special methods **`__copy__()`** and **`__deepcopy__()`**.

object.__copy__(*self*)

Called to implement the shallow copy operation; no additional arguments are passed.

object.__deepcopy__(*self*, *memo*)

Called to implement the deep copy operation; it is passed one argument, the *memo* dictionary. If the `__deepcopy__` implementation needs to make a deep copy of a component, it should call the **`deepcopy()`** function with the component as first argument and the *memo* dictionary as second argument. The *memo* dictionary should be treated as an opaque object.

Function **`copy.replace()`** is more limited than **`copy()`** and **`deepcopy()`**, and only supports named tuples created by **`namedtuple()`**, **`dataclasses`**, and other classes which define method **`__replace__()`**.

object.__replace__(*self*, */*, ***changes*)

This method should create a new object of the same type, replacing fields with values from *changes*.

Added in version 3.13.

See also

Module `pickle`

Discussion of the special methods used to support object state retrieval and restoration.

`pprint` — Data pretty printer

Source code: Lib/pprint.py
[https://github.com/python/cpython/tree/3.14/Lib/pprint.py]

The **`pprint`** module provides a capability to “pretty-print” arbitrary Python data structures in a form which can be used as input to the interpreter. If the formatted structures include objects which are not fundamental Python types, the representation may not be loadable. This may be the case if objects such as files, sockets or classes are included, as well as many other objects which are not representable as Python literals.

The formatted representation keeps objects on a single line if it can, and breaks them onto multiple lines if they don’t fit within the allowed width, adjustable by the *width* parameter defaulting to 80 characters.

Changed in version 3.9: Added support for pretty-printing **`types.SimpleNamespace`**.

Changed in version 3.10: Added support for pretty-printing **`dataclasses.dataclass`**.

Functions

pprint.pp(*object*, *stream=None*, *indent=1*, *width=80*, *depth=None*, *, *compact=False*, *sort_dicts=False*, *underscore_numbers=False*)

Prints the formatted representation of *object*, followed by a newline. This function may be used in the interactive interpreter instead of the **`print()`** function for inspecting values. Tip: you can reassign `print = pprint.pp` for use within a scope.

Parameters::

- **object** – The object to be printed.
- **stream** (file-like object | None) – A file-like object to which the output will be written by calling its **`write()`** method. If `None` (the default), **`sys.stdout`** is used.
- **indent** (*int*) – The amount of indentatio added for each nesting level.
- **width** (*int*) – The desired maximum number of characters per line in the output. If a structure cannot be formatted within the width constraint, a best effort will be made.
- **depth** (*int* | *None*) – The number of nesting levels which may be printed. If the data structure being printed is too deep, the next contained level is replaced by `...`. If `None` (the default), there is no constraint on the depth of th objects being formatted.
- **compact** (*bool*) – Control the way long sequences are formatted. If `False` (the default), each item of a sequence will b formatted on a separate line, otherwise as many items as will fit within the *widt* will be formatted on each output line.
- **sort_dicts** (*bool*) – If `True`, dictionaries will be formatted with their keys sorted, otherwise they will be displayed in insertion order (the default).
- **underscore_numbers** (*bool*) – If `True`, integers will be formatted with the _ character for a thousands separator, otherwise underscores are not displaye (the default).

```
>>> import pprint
>>> stuff = ['spam', 'eggs', 'lumberjack', 'knights', 'ni']
>>> stuff.insert(0, stuff)
>>> pprint.pp(stuff)
[<Recursion on list with id=...>,
 'spam',
 'eggs',
 'lumberjack',
 'knights',
 'ni']
```

Added in version 3.8.

pprint.pprint(*object*, *stream=None*, *indent=1*, *width=80*, *depth=None*, *, *compact=False*, *sort_dicts=True*, *underscore_numbers=False*)

Alias for `pp()` with *sort_dicts* set to `True` by default, which would automatically sort the dictionaries' keys, you might want to use `pp()` instead where it is `False` by default.

pprint.pformat(*object*, *indent=1*, *width=80*, *depth=None*, *, *compact=False*, *sort_dicts=True*, *underscore_numbers=False*)

Return the formatted representation of *object* as a string. *indent*, *width*, *depth*, *compact*, *sort_dicts* and *underscore_numbers* are passed to the `PrettyPrinter` constructor as formatting parameters and their meanings are as described in the documentation above.

pprint.isreadable(*object*)

Determine if the formatted representation of *object* is "readable", or can be used to reconstruct the value using `eval()`. This always returns `False` for recursive objects.

```
>>> pprint.isreadable(stuff)
False
```

pprint.isrecursive(*object*)

Determine if *object* requires a recursive representation. This function is subject to the same limitations as noted in `saferepr()` below and may raise an `RecursionError` if it fails to detect a recursive object.

pprint.saferepr(*object*)

Return a string representation of *object*, protected against recursion in some common data structures, namely instances of `dict`, `list` and `tuple` or subclasses whose `__repr__` has not been overridden. If the representation of object exposes a recursive entry, the recursive reference will be represented as `<Recursion on typename with id=number>`. The representation is not otherwise formatted.

```
>>> pprint.saferepr(stuff)
"[<Recursion on list with id=...>, 'spam', 'eggs',
'lumberjack', 'knights', 'ni']"
```

PrettyPrinter Objects

class pprint.PrettyPrinter(*indent=1*, *width=80*, *depth=None*, *stream=None*, *, *compact=False*, *sort_dicts=True*, *underscore_numbers=False*)

Construct a `PrettyPrinter` instance.

Arguments have the same meaning as for `pp()`. Note that they are in a different order, and that *sort_dicts* defaults to `True`.

```
>>> import pprint
>>> stuff = ['spam', 'eggs', 'lumberjack', 'knights', 'ni']
>>> stuff.insert(0, stuff[:])
>>> pp = pprint.PrettyPrinter(indent=4)
>>> pp.pprint(stuff)
[   ['spam', 'eggs', 'lumberjack', 'knights', 'ni'],
    'spam',
    'eggs',
    'lumberjack',
    'knights',
    'ni']
```

```
>>> pp = pprint.PrettyPrinter(width=41, compact=True)
>>> pp.pprint(stuff)
[['spam', 'eggs', 'lumberjack',
  'knights', 'ni'],
 'spam', 'eggs', 'lumberjack', 'knights',
 'ni']
>>> tup = ('spam', ('eggs', ('lumberjack', ('knights', ('ni',
('dead',
... ('parrot', ('fresh fruit',))))))))
>>> pp = pprint.PrettyPrinter(depth=6)
>>> pp.pprint(tup)
('spam', ('eggs', ('lumberjack', ('knights', ('ni', ('dead',
(...)))))))
```

Changed in version 3.4: Added the *compact* parameter.

Changed in version 3.8: Added the *sort_dicts* parameter.

Changed in version 3.10: Added the *underscore_numbers* parameter.

Changed in version 3.11: No longer attempts to write to **`sys.stdout`** if it is `None`.

`PrettyPrinter` instances have the following methods:

PrettyPrinter.pformat(*object*)

Return the formatted representation of *object*. This takes into account the options passed to the **`PrettyPrinter`** constructor.

PrettyPrinter.pprint(*object*)

Print the formatted representation of *object* on the configured stream, followed by a newline.

The following methods provide the implementations for the corresponding functions of the same names. Using these methods on an instance is slightly more efficient since new **`PrettyPrinter`** objects don’t need to be created.

PrettyPrinter.isreadable(*object*)

Determine if the formatted representation of the object is "readable," or can be used to reconstruct the value using `eval()`. Note that this returns `False` for recursive objects. If the *depth* parameter of the `PrettyPrinter` is set and the object is deeper than allowed, this returns `False`.

PrettyPrinter.isrecursive(*object*)

Determine if the object requires a recursive representation.

This method is provided as a hook to allow subclasses to modify the way objects are converted to strings. The default implementation uses the internals of the `saferepr()` implementation.

PrettyPrinter.format(*object*, *context*, *maxlevels*, *level*)

Returns three values: the formatted version of *object* as a string, a flag indicating whether the result is readable, and a flag indicating whether recursion was detected. The first argument is the object to be presented. The second is a dictionary which contains the `id()` of objects that are part of the current presentation context (direct and indirect containers for *object* that are affecting the presentation) as the keys; if an object needs to be presented which is already represented in *context*, the third return value should be `True`. Recursive calls to the `format()` method should add additional entries for containers to this dictionary. The third argument, *maxlevels*, gives the requested limit to recursion; this will be `0` if there is no requested limit. This argument should be passed unmodified to recursive calls. The fourth argument, *level*, gives the current level; recursive calls should be passed a value less than that of the current call.

Example

To demonstrate several uses of the `pp()` function and its parameters, let’s fetch information about a project from PyPI [https://pypi.org]:

```
>>> import json
>>> import pprint
>>> from urllib.request import urlopen
>>> with
urlopen('https://pypi.org/pypi/sampleproject/1.2.0/json') as
resp:
...     project_info = json.load(resp)['info']
```

In its basic form, `pp()` shows the whole object:

```
>>> pprint.pp(project_info)
{'author': 'The Python Packaging Authority',
 'author_email': 'pypa-dev@googlegroups.com',
 'bugtrack_url': None,
 'classifiers': ['Development Status :: 3 - Alpha',
                 'Intended Audience :: Developers',
                 'License :: OSI Approved :: MIT License',
                 'Programming Language :: Python :: 2',
                 'Programming Language :: Python :: 2.6',
                 'Programming Language :: Python :: 2.7',
                 'Programming Language :: Python :: 3',
                 'Programming Language :: Python :: 3.2',
                 'Programming Language :: Python :: 3.3',
                 'Programming Language :: Python :: 3.4',
                 'Topic :: Software Development :: Build
Tools'],
 'description': 'A sample Python project\n'
                '=======================\n'
                '\n'
                'This is the description file for the
project.\n'
                '\n'
                'The file should use UTF-8 encoding and be
written using '
                'ReStructured Text. It\n'
                'will be used to generate the project webpage on
PyPI, and '
                'should be written for\n'
                'that purpose.\n'
                '\n'
                'Typical contents for this file would include an 
overview of '
                'the project, basic\n'
                'usage examples, etc. Generally, including the 
project '
                'changelog in here is not\n'
                'a good idea, although a simple "What\'s New" 
section for the '
                'most recent version\n'
                'may be appropriate.',
 'description_content_type': None,
 'docs_url': None,
 'download_url': 'UNKNOWN',
 'downloads': {'last_day': -1, 'last_month': -1, 'last_week': 
-1},
 'home_page': 'https://github.com/pypa/sampleproject',
 'keywords': 'sample setuptools development',
 'license': 'MIT',
 'maintainer': None,
 'maintainer_email': None,
 'name': 'sampleproject',
 'package_url': 'https://pypi.org/project/sampleproject/',
 'platform': 'UNKNOWN',
 'project_url': 'https://pypi.org/project/sampleproject/',
 'project_urls': {'Download': 'UNKNOWN',
                  'Homepage': 
'https://github.com/pypa/sampleproject'},
 'release_url': 'https://pypi.org/project/sampleproject/1.2.0/',
 'requires_dist': None,
 'requires_python': None,
 'summary': 'A sample Python project',
 'version': '1.2.0'}
```

The result can be limited to a certain *depth* (ellipsis is used for deeper contents):

```
>>> pprint.pp(project_info, depth=1)
{'author': 'The Python Packaging Authority',
 'author_email': 'pypa-dev@googlegroups.com',
 'bugtrack_url': None,
 'classifiers': [...],
 'description': 'A sample Python project\n'
```

```
                '=======================\n'
                '\n'
                'This is the description file for the
project.\n'
                '\n'
                'The file should use UTF-8 encoding and be
written using '
                'ReStructured Text. It\n'
                'will be used to generate the project webpage on
PyPI, and '
                'should be written for\n'
                'that purpose.\n'
                '\n'
                'Typical contents for this file would include an
overview of '
                'the project, basic\n'
                'usage examples, etc. Generally, including the
project '
                'changelog in here is not\n'
                'a good idea, although a simple "What\'s New"
section for the '
                'most recent version\n'
                'may be appropriate.',
 'description_content_type': None,
 'docs_url': None,
 'download_url': 'UNKNOWN',
 'downloads': {...},
 'home_page': 'https://github.com/pypa/sampleproject',
 'keywords': 'sample setuptools development',
 'license': 'MIT',
 'maintainer': None,
 'maintainer_email': None,
 'name': 'sampleproject',
 'package_url': 'https://pypi.org/project/sampleproject/',
 'platform': 'UNKNOWN',
 'project_url': 'https://pypi.org/project/sampleproject/',
 'project_urls': {...},
 'release_url': 'https://pypi.org/project/sampleproject/1.2.0/',
 'requires_dist': None,
 'requires_python': None,
 'summary': 'A sample Python project',
 'version': '1.2.0'}
```

Additionally, maximum character *width* can be suggested. If a long object cannot be split, the specified width will be exceeded:

```
>>> pprint.pp(project_info, depth=1, width=60)
{'author': 'The Python Packaging Authority',
 'author_email': 'pypa-dev@googlegroups.com',
 'bugtrack_url': None,
 'classifiers': [...],
 'description': 'A sample Python project\n'
                '=======================\n'
                '\n'
                'This is the description file for the '
                'project.\n'
                '\n'
                'The file should use UTF-8 encoding and be '
                'written using ReStructured Text. It\n'
                'will be used to generate the project '
                'webpage on PyPI, and should be written '
                'for\n'
                'that purpose.\n'
                '\n'
                'Typical contents for this file would '
                'include an overview of the project, '
                'basic\n'
                'usage examples, etc. Generally, including '
                'the project changelog in here is not\n'
                'a good idea, although a simple "What\'s '
                'New" section for the most recent version\n'
                'may be appropriate.',
 'description_content_type': None,
 'docs_url': None,
 'download_url': 'UNKNOWN',
 'downloads': {...},
 'home_page': 'https://github.com/pypa/sampleproject',
 'keywords': 'sample setuptools development',
 'license': 'MIT',
 'maintainer': None,
 'maintainer_email': None,
 'name': 'sampleproject',
 'package_url': 'https://pypi.org/project/sampleproject/',
 'platform': 'UNKNOWN',
 'project_url': 'https://pypi.org/project/sampleproject/',
'project_urls': {...},
'release_url': 'https://pypi.org/project/sampleproject/1.2.0/',
'requires_dist': None,
'requires_python': None,
'summary': 'A sample Python project',
'version': '1.2.0'}
```

`reprlib` — Alternate `repr()` implementation

Source code: Lib/reprlib.py [https://github.com/python/cpython/tree/3.14/Lib/reprlib.py]

The **`reprlib`** module provides a means for producing object representations with limits on the size of the resulting strings. This is used in the Python debugger and may be useful in other contexts as well.

This module provides a class, an instance, and a function:

class reprlib.Repr(*, *maxlevel=6*, *maxtuple=6*, *maxlist=6*, *maxarray=5*, *maxdict=4*, *maxset=6*, *maxfrozenset=6*, *maxdeque=6*, *maxstring=30*, *maxlong=40*, *maxother=30*, *fillvalue='...'*, *indent=None*)

Class which provides formatting services useful in implementing functions similar to the built-in **`repr()`**; size limits for different object types are added to avoid the generation of representations which are excessively long.

The keyword arguments of the constructor can be used as a shortcut to set the attributes of the **`Repr`** instance. Which means that the following initialization:

```
aRepr = reprlib.Repr(maxlevel=3)
```

Is equivalent to:

```
aRepr = reprlib.Repr()
aRepr.maxlevel = 3
```

See section Repr Objects for more information about `Repr` attributes.

Changed in version 3.12: Allow attributes to be set via keyword arguments.

reprlib.aRepr

This is an instance of `Repr` which is used to provide the `repr()` function described below. Changing the attributes of this object will affect the size limits used by `repr()` and the Python debugger.

reprlib.repr(*obj*)

This is the `repr()` method of `aRepr`. It returns a string similar to that returned by the built-in function of the same name, but with limits on most sizes.

In addition to size-limiting tools, the module also provides a decorator for detecting recursive calls to `__repr__()` and substituting a placeholder string instead.

@reprlib.recursive_repr(*fillvalue='...'*)

Decorator for `__repr__()` methods to detect recursive calls within the same thread. If a recursive call is made, the *fillvalue* is returned, otherwise, the usual `__repr__()` call is made. For example:

```
>>> from reprlib import recursive_repr
>>> class MyList(list):
...     @recursive_repr()
...     def __repr__(self):
...         return '<' + '|'.join(map(repr, self)) + '>'
...
>>> m = MyList('abc')
>>> m.append(m)
>>> m.append('x')
>>> print(m)
<'a'|'b'|'c'|...|'x'>
```

Added in version 3.2.

Repr Objects

`Repr` instances provide several attributes which can be used to provide size limits for the representations of different object types, and methods which format specific object types.

Repr.fillvalue

This string is displayed for recursive references. It defaults to `...`.

Added in version 3.11.

Repr.maxlevel

Depth limit on the creation of recursive representations. The default is `6`.

Repr.maxdict
Repr.maxlist
Repr.maxtuple
Repr.maxset
Repr.maxfrozenset
Repr.maxdeque
Repr.maxarray

Limits on the number of entries represented for the named object type. The default is `4` for `maxdict`, `5` for `maxarray`, and `6` for the others.

Repr.maxlong

Maximum number of characters in the representation for an integer. Digits are dropped from the middle. The default is `40`.

Repr.maxstring

Limit on the number of characters in the representation of the string. Note that the “normal” representation of the string is used as the character source: if escape sequences are needed in the representation, these may be mangled when the representation is shortened. The default is `30`.

Repr.maxother

This limit is used to control the size of object types for which no specific formatting method is available on the **`Repr`** object. It is applied in a similar manner as **`maxstring`**. The default is `20`.

Repr.indent

If this attribute is set to `None` (the default), the output is formatted with no line breaks or indentation, like the standard **`repr()`**. For example:

```
>>> example = [
...     1, 'spam', {'a': 2, 'b': 'spam eggs', 'c': {3: 4.5,
6: []}}, 'ham']
>>> import reprlib
>>> aRepr = reprlib.Repr()
>>> print(aRepr.repr(example))
[1, 'spam', {'a': 2, 'b': 'spam eggs', 'c': {3: 4.5, 6:
[]}}, 'ham']
```

If **`indent`** is set to a string, each recursion level is placed on its own line, indented by that string:

```
>>> aRepr.indent = '-->'
>>> print(aRepr.repr(example))
[
-->1,
-->'spam',
-->{
-->-->'a': 2,
-->-->'b': 'spam eggs',
-->-->'c': {
```

```
-->-->-->3: 4.5,
-->-->-->6: [],
-->-->},
-->},
-->'ham',
]
```

Setting `indent` to a positive integer value behaves as if it was set to a string with that number of spaces:

```
>>> aRepr.indent = 4
>>> print(aRepr.repr(example))
[
    1,
    'spam',
    {
        'a': 2,
        'b': 'spam eggs',
        'c': {
            3: 4.5,
            6: [],
        },
    },
    'ham',
]
```

Added in version 3.12.

Repr.repr(*obj*)

The equivalent to the built-in `repr()` that uses the formatting imposed by the instance.

Repr.repr1(*obj*, *level*)

Recursive implementation used by `repr()`. This uses the type of *obj* to determine which formatting method to call, passing it *obj* and *level*. The type-specific methods should call `repr1()` to perform recursive formatting, with `level - 1` for the value of *level* in the recursive call.

Repr.repr_TYPE(*obj*, *level*)

Formatting methods for specific types are implemented as methods with a name based on the type name. In the method name, **TYPE** is replaced by `'_'.join(type(obj).__name__.split())`. Dispatch to these methods is handled by `repr1()`. Type-specific methods which need to recursively format a value should call `self.repr1(subobj, level - 1)`.

Subclassing Repr Objects

The use of dynamic dispatching by `Repr.repr1()` allows subclasses of `Repr` to add support for additional built-in object types or to modify the handling of types already supported. This example shows how special support for file objects could be added:

```
import reprlib
import sys

class MyRepr(reprlib.Repr):

    def repr_TextIOWrapper(self, obj, level):
        if obj.name in {'<stdin>', '<stdout>', '<stderr>'}:
            return obj.name
        return repr(obj)

aRepr = MyRepr()
print(aRepr.repr(sys.stdin))         # prints '<stdin>'
```

```
<stdin>
```

`enum` — Support for enumerations

Added in version 3.4.

Source code: Lib/enum.py [https://github.com/python/cpython/tree/3.14/Lib/enum.py]

Important

This page contains the API reference information. For tutorial information and discussion of more advanced topics, see

- Basic Tutorial
- Advanced Tutorial
- Enum Cookbook

An enumeration:

- is a set of symbolic names (members) bound to unique values
- can be iterated over to return its canonical (i.e. non-alias) members in definition order
- uses *call* syntax to return members by value
- uses *index* syntax to return members by name

Enumerations are created either by using **`class`** syntax, or by using function-call syntax:

```
>>> from enum import Enum

>>> # class syntax
>>> class Color(Enum):
...     RED = 1
...     GREEN = 2
...     BLUE = 3

>>> # functional syntax
>>> Color = Enum('Color', [('RED', 1), ('GREEN', 2), ('BLUE',
3)])
```

Even though we can use **`class`** syntax to create Enums, Enums are not normal Python classes. See How are Enums different? for more details.

Note

Nomenclature

- The class **`Color`** is an *enumeration* (or *enum*)
- The attributes **`Color.RED`**, **`Color.GREEN`**, etc., are *enumeration members* (or *members*) and are functionally constants.
- The enum members have *names* and *values* (the name of **`Color.RED`** is `RED`, the value of **`Color.BLUE`** is `3`, etc.)

Module Contents

`EnumType`

The `type` for Enum and its subclasses.

`Enum`

Base class for creating enumerated constants.

`IntEnum`

Base class for creating enumerated constants that are also subclasses of **`int`**. (Notes)

`StrEnum`

Base class for creating enumerated constants that are also subclasses of `str`. (Notes)

`Flag`

Base class for creating enumerated constants that can be combined using the bitwise operations without losing their `Flag` membership.

`IntFlag`

Base class for creating enumerated constants that can be combined using the bitwise operators without losing their `IntFlag` membership. `IntFlag` members are also subclasses of `int`. (Notes)

`ReprEnum`

Used by `IntEnum`, `StrEnum`, and `IntFlag` to keep the `str()` of the mixed-in type.

`EnumCheck`

An enumeration with the values `CONTINUOUS`, `NAMED_FLAGS`, and `UNIQUE`, for use with `verify()` to ensure various constraints are met by a given enumeration.

`FlagBoundary`

An enumeration with the values `STRICT`, `CONFORM`, `EJECT`, and `KEEP` which allows for more fine-grained control over how invalid values are dealt with in an enumeration.

`EnumDict`

A subclass of `dict` for use when subclassing `EnumType`.

`auto`

Instances are replaced with an appropriate value for Enum members. `StrEnum` defaults to the lower-cased version of the member name, while other Enums default to 1 and increase from there.

`property()`

Allows `Enum` members to have attributes without conflicting with member names. The `value` and `name` attributes are implemented this way.

`unique()`

Enum class decorator that ensures only one name is bound to any one value.

`verify()`

Enum class decorator that checks user-selectable constraints on an enumeration.

`member()`

Make `obj` a member. Can be used as a decorator.

`nonmember()`

Do not make `obj` a member. Can be used as a decorator.

`global_enum()`

Modify the `str()` and `repr()` of an enum to show its members as belonging to the module instead of its class, and export the enum members to the global namespace.

`show_flag_values()`

Return a list of all power-of-two integers contained in a flag.

`enum.bin()`

Like built-in **`bin()`**, except negative values are represented in two’s complement, and the leading bit always indicates sign (`0` implies positive, `1` implies negative).

Added in version 3.6: `Flag`, `IntFlag`, `auto`

Added in version 3.11: `StrEnum`, `EnumCheck`, `ReprEnum`, `FlagBoundary`, `property`, `member`, `nonmember`, `global_enum`, `show_flag_values`

Added in version 3.13: `EnumDict`

Data Types

class enum.EnumType

EnumType is the metaclass for *enum* enumerations. It is possible to subclass *EnumType* – see Subclassing EnumType for details.

`EnumType` is responsible for setting the correct **`__repr__()`**, **`__str__()`**, **`__format__()`**, and **`__reduce__()`** methods on the final *enum*, as well as creating the enum members, properly handling duplicates, providing iteration over the enum class, etc.

Added in version 3.11: Before 3.11 `EnumType` was called `EnumMeta`, which is still available as an alias.

__call__(*cls*, *value*, *names=None*, ***, *module=None*, *qualname=None*, *type=None*, *start=1*, *boundary=None*)

This method is called in two different ways:

- to look up an existing member:

cls::	The enum class being called.
value::	The value to lookup.

- to use the `cls` enum to create a new enum (only if the existing enum does not have any members):

cls::	The enum class being called.
value::	The name of the new Enum to create.
names::	The names/values of the members for the new Enum.
module::	The name of the module the new Enum is created in.
qualname::	The actual location in the module where this Enum can be found.
type::	A mix-in type for the new Enum.
start::	The first integer value for the Enum (used by `auto`).
boundary::	How to handle out-of-range values from bit operations (`Flag` only).

__contains__(*cls*, *member*)

Returns `True` if member belongs to the `cls`:

```
>>> some_var = Color.RED
>>> some_var in Color
True
>>> Color.RED.value in Color
True
```

Changed in version 3.12: Before Python 3.12, a `TypeError` is raised if a non-Enum-member is used in a containment check.

__dir__(*cls*)

Returns `['__class__', '__doc__', '__members__', '__module__']` and the names of the members in *cls*:

```
>>> dir(Color)
['BLUE', 'GREEN', 'RED', '__class__', '__contains__',
'__doc__', '__getitem__', '__init_subclass__',
'__iter__', '__len__', '__members__', '__module__',
'__name__', '__qualname__']
```

__getitem__(*cls*, *name*)

Returns the Enum member in *cls* matching *name*, or raises a `KeyError`:

```
>>> Color['BLUE']
<Color.BLUE: 3>
```

__iter__(*cls*)

Returns each member in *cls* in definition order:

```
>>> list(Color)
[<Color.RED: 1>, <Color.GREEN: 2>, <Color.BLUE: 3>]
```

__len__(*cls*)

Returns the number of member in *cls*:

```
>>> len(Color)
3
```

__members__

Returns a mapping of every enum name to its member, including aliases

__reversed__(*cls*)

Returns each member in *cls* in reverse definition order:

```
>>> list(reversed(Color))
[<Color.BLUE: 3>, <Color.GREEN: 2>, <Color.RED: 1>]
```

class enum.Enum

Enum is the base class for all *enum* enumerations.

name

The name used to define the `Enum` member:

```
>>> Color.BLUE.name
'BLUE'
```

value

The value given to the `Enum` member:

```
>>> Color.RED.value
1
```

Value of the member, can be set in `__new__()`.

Note

Enum member values

Member values can be anything: `int`, `str`, etc. If the exact value is unimportant you may use `auto` instances and an appropriate value will be chosen for you. See `auto` for the details.

While mutable/unhashable values, such as `dict`, `list` or a mutable `dataclass`, can be used, they will have a quadratic performance impact during creation relative to the total number of mutable/unhashable values in the enum.

name

Name of the member.

value

Value of the member, can be set in `__new__()`.

order

No longer used, kept for backward compatibility. (class attribute, removed during class creation).

ignore

`_ignore_` is only used during creation and is removed from the enumeration once creation is complete.

`_ignore_` is a list of names that will not become members, and whose names will also be removed from the completed enumeration. See TimePeriod for an example.

__dir__(*self*)

Returns `['__class__', '__doc__', '__module__', 'name', 'value']` and any public methods defined on *self.__class__*:

```
>>> from enum import Enum
>>> from datetime import date
>>> class Weekday(Enum):
...     MONDAY = 1
...     TUESDAY = 2
...     WEDNESDAY = 3
...     THURSDAY = 4
...     FRIDAY = 5
...     SATURDAY = 6
...     SUNDAY = 7
...     @classmethod
...     def today(cls):
...         print('today is %s' %
cls(date.today().isoweekday()).name)
...
>>> dir(Weekday.SATURDAY)
['__class__', '__doc__', '__eq__', '__hash__',
'__module__', 'name', 'today', 'value']
```

_generate_next_value_(*name*, *start*, *count*, *last_values*)

name::	The name of the member being defined (e.g. 'RED').
start::	The start value for the Enum; the default is 1.
count::	The number of members currently defined, not including this one.
last_values::	A list of the previous values.

A *staticmethod* that is used to determine the next value returned by `auto`:

```
>>> from enum import auto, Enum
>>> class PowersOfThree(Enum):
...     @staticmethod
...     def _generate_next_value_(name, start, count,
last_values):
...         return 3 ** (count + 1)
...     FIRST = auto()
...     SECOND = auto()
...
>>> PowersOfThree.SECOND.value
9
```

__init__(*self*, **args*, ***kwds*)

By default, does nothing. If multiple values are given in the member assignment, those values become separate arguments to `__init__`; e.g.

```
>>> from enum import Enum
>>> class Weekday(Enum):
...     MONDAY = 1, 'Mon'
```

`Weekday.__init__()` would be called as `Weekday.__init__(self, 1, 'Mon')`

__init_subclass__(*cls*, ***kwds*)

A *classmethod* that is used to further configure subsequent subclasses. By default, does nothing.

missing(*cls*, *value*)

A *classmethod* for looking up values not found in *cls*. By default it does nothing, but can be overridden to implement custom search behavior:

```
>>> from enum import auto, StrEnum
>>> class Build(StrEnum):
...     DEBUG = auto()
...     OPTIMIZED = auto()
...     @classmethod
...     def _missing_(cls, value):
...         value = value.lower()
...         for member in cls:
...             if member.value == value:
...                 return member
...         return None
...
>>> Build.DEBUG.value
'debug'
>>> Build('deBUG')
<Build.DEBUG: 'debug'>
```

__new__(*cls*, **args*, ***kwds*)

By default, doesn’t exist. If specified, either in the enum class definition or in a mixin class (such as `int`), all values given in the member assignment will be passed; e.g.

```
>>> from enum import Enum
>>> class MyIntEnum(int, Enum):
...     TWENTYSIX = '1a', 16
```

results in the call `int('1a', 16)` and a value of `26` for the member.

Note

When writing a custom `__new__`, do not use `super().__new__` – call the appropriate `__new__` instead.

__repr__(*self*)

Returns the string used for *repr()* calls. By default, returns the *Enum* name, member name, and value, but can be overridden:

```
>>> from enum import auto, Enum
>>> class OtherStyle(Enum):
...     ALTERNATE = auto()
...     OTHER = auto()
...     SOMETHING_ELSE = auto()
...     def __repr__(self):
...         cls_name = self.__class__.__name__
...         return f'{cls_name}.{self.name}'
...
>>> OtherStyle.ALTERNATE, str(OtherStyle.ALTERNATE), f"
{OtherStyle.ALTERNATE}"
(OtherStyle.ALTERNATE, 'OtherStyle.ALTERNATE',
'OtherStyle.ALTERNATE')
```

__str__(*self*)

Returns the string used for *str()* calls. By default, returns the *Enum* name and member name, but can be overridden:

```
>>> from enum import auto, Enum
>>> class OtherStyle(Enum):
...     ALTERNATE = auto()
...     OTHER = auto()
...     SOMETHING_ELSE = auto()
...     def __str__(self):
...         return f'{self.name}'
...
>>> OtherStyle.ALTERNATE, str(OtherStyle.ALTERNATE), f"
{OtherStyle.ALTERNATE}"
(<OtherStyle.ALTERNATE: 1>, 'ALTERNATE', 'ALTERNATE')
```

__format__(*self*)

Returns the string used for *format()* and *f-string* calls. By default, returns `__str__()` return value, but can be overridden:

```
>>> from enum import auto, Enum
>>> class OtherStyle(Enum):
...     ALTERNATE = auto()
...     OTHER = auto()
...     SOMETHING_ELSE = auto()
...     def __format__(self, spec):
...         return f'{self.name}'
...
>>> OtherStyle.ALTERNATE, str(OtherStyle.ALTERNATE), f"
{OtherStyle.ALTERNATE}"
(<OtherStyle.ALTERNATE: 1>, 'OtherStyle.ALTERNATE',
'ALTERNATE')
```

Note

Using `auto` with `Enum` results in integers of increasing value, starting with `1`.

Changed in version 3.12: Added Dataclass support

_add_alias_()

Adds a new name as an alias to an existing member:

```
>>> Color.RED._add_alias_("ERROR")
>>> Color.ERROR
<Color.RED: 1>
```

Raises a `NameError` if the name is already assigned to a different member.

Added in version 3.13.

_add_value_alias_()

Adds a new value as an alias to an existing member:

```
>>> Color.RED._add_value_alias_(42)
>>> Color(42)
<Color.RED: 1>
```

Raises a `ValueError` if the value is already linked with a different member.

Added in version 3.13.

class enum.IntEnum

IntEnum is the same as `Enum`, but its members are also integers and can be used anywhere that an integer can be used. If any integer operation is performed with an *IntEnum* member, the resulting value loses its enumeration status.

```
>>> from enum import IntEnum
>>> class Number(IntEnum):
...     ONE = 1
...     TWO = 2
...     THREE = 3
...
>>> Number.THREE
<Number.THREE: 3>
>>> Number.ONE + Number.TWO
3
>>> Number.THREE + 5
8
>>> Number.THREE == 3
True
```

Note

Using `auto` with `IntEnum` results in integers of increasing value, starting with `1`.

Changed in version 3.11: `__str__()` is now **`int.__str__()`** to better support the *replacement of existing constants* use-case. `__format__()` was already **`int.__format__()`** for that same reason.

class enum.StrEnum

StrEnum is the same as `Enum`, but its members are also strings and can be used in most of the same places that a string can be used. The result of any string operation performed on or with a *StrEnum* member is not part of the enumeration.

```
>>> from enum import StrEnum, auto
>>> class Color(StrEnum):
...     RED = 'r'
...     GREEN = 'g'
...     BLUE = 'b'
...     UNKNOWN = auto()
...
>>> Color.RED
<Color.RED: 'r'>
>>> Color.UNKNOWN
<Color.UNKNOWN: 'unknown'>
>>> str(Color.UNKNOWN)
'unknown'
```

Note

There are places in the stdlib that check for an exact `str` instead of a `str` subclass (i.e. `type(unknown) == str` instead of `isinstance(unknown, str)`), and in those locations you will need to use `str(MyStrEnum.MY_MEMBER)`.

Note

Using `auto` with `StrEnum` results in the lower-cased member name as the value.

Note

`__str__()` is **`str.__str__()`** to better support the *replacement of existing constants* use-case. `__format__()` is likewise **`str.__format__()`** for that same reason.

Added in version 3.11.

class enum.Flag

`Flag` is the same as `Enum`, but its members support the bitwise operators `&` (*AND*), `|` (*OR*), `^` (*XOR*), and `~` (*INVERT*); the results of those operations are (aliases of) members of the enumeration.

__contains__(*self*, *value*)

Returns *True* if value is in self:

```
>>> from enum import Flag, auto
>>> class Color(Flag):
...     RED = auto()
...     GREEN = auto()
...     BLUE = auto()
...
>>> purple = Color.RED | Color.BLUE
>>> white = Color.RED | Color.GREEN | Color.BLUE
>>> Color.GREEN in purple
False
>>> Color.GREEN in white
True
>>> purple in white
True
>>> white in purple
False
```

__iter__(self):

Returns all contained non-alias members:

```
>>> list(Color.RED)
[<Color.RED: 1>]
>>> list(purple)
[<Color.RED: 1>, <Color.BLUE: 4>]
```

Added in version 3.11.

__len__(self):

Returns number of members in flag:

```
>>> len(Color.GREEN)
1
>>> len(white)
3
```

Added in version 3.11.

__bool__(self):

Returns *True* if any members in flag, *False* otherwise:

```
>>> bool(Color.GREEN)
True
>>> bool(white)
True
>>> black = Color(0)
>>> bool(black)
False
```

__or__(*self*, *other*)

Returns current flag binary or'ed with other:

```
>>> Color.RED | Color.GREEN
<Color.RED|GREEN: 3>
```

__and__(*self*, *other*)

Returns current flag binary and'ed with other:

```
>>> purple & white
<Color.RED|BLUE: 5>
```

```
>>> purple & Color.GREEN
<Color: 0>
```

__xor__(*self*, *other*)

Returns current flag binary xor’ed with other:

```
>>> purple ^ white
<Color.GREEN: 2>
>>> purple ^ Color.GREEN
<Color.RED|GREEN|BLUE: 7>
```

__invert__(self):

Returns all the flags in *type(self)* that are not in *self*:

```
>>> ~white
<Color: 0>
>>> ~purple
<Color.GREEN: 2>
>>> ~Color.RED
<Color.GREEN|BLUE: 6>
```

_numeric_repr_()

Function used to format any remaining unnamed numeric values. Default is the value’s repr; common choices are `hex()` and `oct()`.

Note

Using `auto` with `Flag` results in integers that are powers of two, starting with `1`.

Changed in version 3.11: The *repr()* of zero-valued flags has changed. It is now:

```
>>> Color(0)
<Color: 0>
```

class enum.IntFlag

`IntFlag` is the same as **`Flag`**, but its members are also integers and can be used anywhere that an integer can be used.

```
>>> from enum import IntFlag, auto
>>> class Color(IntFlag):
...     RED = auto()
...     GREEN = auto()
...     BLUE = auto()
...
>>> Color.RED & 2
<Color: 0>
>>> Color.RED | 2
<Color.RED|GREEN: 3>
```

If any integer operation is performed with an *IntFlag* member, the result is not an *IntFlag*:

```
>>> Color.RED + 2
3
```

If a **`Flag`** operation is performed with an *IntFlag* member and:

- the result is a valid *IntFlag*: an *IntFlag* is returned
- the result is not a valid *IntFlag*: the result depends on the **`FlagBoundary`** setting

The **`repr()`** of unnamed zero-valued flags has changed. It is now:

```
>>> Color(0)
<Color: 0>
```

Note

Using **`auto`** with **`IntFlag`** results in integers that are powers of two, starting with `1`.

Changed in version 3.11: `__str__()` is now **`int.__str__()`** to better support the *replacement of existing constants* use-case. `__format__()` was already **`int.__format__()`** for that same reason.

Inversion of an **`IntFlag`** now returns a positive value that is the union of all flags not in the given flag, rather than a negative value. This matches the existing `Flag` behavior.

class enum.ReprEnum

`ReprEnum` uses the `repr()` of `Enum`, but the `str()` of the mixed-in data type:

- **`int.__str__()`** for `IntEnum` and `IntFlag`
- **`str.__str__()`** for `StrEnum`

Inherit from **`ReprEnum`** to keep the `str()` / `format()` of the mixed-in data type instead of using the `Enum`-default `str()`.

Added in version 3.11.

class enum.EnumCheck

EnumCheck contains the options used by the `verify()` decorator to ensure various constraints; failed constraints result in a `ValueError`.

UNIQUE

Ensure that each value has only one name:

```
>>> from enum import Enum, verify, UNIQUE
>>> @verify(UNIQUE)
... class Color(Enum):
...     RED = 1
...     GREEN = 2
...     BLUE = 3
...     CRIMSON = 1
Traceback (most recent call last):
...
```

```
ValueError: aliases found in <enum 'Color'>: CRIMSON ->
RED
```

CONTINUOUS

Ensure that there are no missing values between the lowest-valued member and the highest-valued member:

```
>>> from enum import Enum, verify, CONTINUOUS
>>> @verify(CONTINUOUS)
... class Color(Enum):
...     RED = 1
...     GREEN = 2
...     BLUE = 5
Traceback (most recent call last):
...
ValueError: invalid enum 'Color': missing values 3, 4
```

NAMED_FLAGS

Ensure that any flag groups/masks contain only named flags – useful when values are specified instead of being generated by `auto()`:

```
>>> from enum import Flag, verify, NAMED_FLAGS
>>> @verify(NAMED_FLAGS)
... class Color(Flag):
...     RED = 1
...     GREEN = 2
...     BLUE = 4
...     WHITE = 15
...     NEON = 31
Traceback (most recent call last):
...
ValueError: invalid Flag 'Color': aliases WHITE and NEON
are missing combined values of 0x18 [use
enum.show_flag_values(value) for details]
```

Note

CONTINUOUS and NAMED_FLAGS are designed to work with integer-valued members.

Added in version 3.11.

class enum.FlagBoundary

`FlagBoundary` controls how out-of-range values are handled in **`Flag`** and its subclasses.

STRICT

Out-of-range values cause a **`ValueError`** to be raised. This is the default for **`Flag`**:

```
>>> from enum import Flag, STRICT, auto
>>> class StrictFlag(Flag, boundary=STRICT):
...     RED = auto()
...     GREEN = auto()
...     BLUE = auto()
...
>>> StrictFlag(2**2 + 2**4)
Traceback (most recent call last):
...
ValueError: <flag 'StrictFlag'> invalid value 20
    given 0b0 10100
  allowed 0b0 00111
```

CONFORM

Out-of-range values have invalid values removed, leaving a valid **`Flag`** value:

```
>>> from enum import Flag, CONFORM, auto
>>> class ConformFlag(Flag, boundary=CONFORM):
...     RED = auto()
...     GREEN = auto()
...     BLUE = auto()
...
>>> ConformFlag(2**2 + 2**4)
<ConformFlag.BLUE: 4>
```

EJECT

Out-of-range values lose their **`Flag`** membership and revert to **`int`**.

```
>>> from enum import Flag, EJECT, auto
>>> class EjectFlag(Flag, boundary=EJECT):
...     RED = auto()
...     GREEN = auto()
...     BLUE = auto()
...
>>> EjectFlag(2**2 + 2**4)
20
```

KEEP

Out-of-range values are kept, and the **`Flag`** membership is kept. This is the default for **`IntFlag`**:

```
>>> from enum import Flag, KEEP, auto
>>> class KeepFlag(Flag, boundary=KEEP):
...     RED = auto()
...     GREEN = auto()
...     BLUE = auto()
...
>>> KeepFlag(2**2 + 2**4)
<KeepFlag.BLUE|16: 20>
```

Added in version 3.11.

class enum.EnumDict

EnumDict is a subclass of **`dict`** that is used as the namespace for defining enum classes (see Preparing the class namespace). It is exposed to allow subclasses of **`EnumType`** with advanced behavior like having multiple values per member. It should be called with the name of the enum class being created, otherwise private names and internal classes will not be handled correctly.

Note that only the **`MutableMapping`** interface (**`__setitem__()`** and **`update()`**) is overridden. It may be possible to bypass the checks using other **`dict`** operations like `|=`.

member_names
A list of member names.

Added in version 3.13.

Supported `__dunder__` names

`__members__` is a read-only ordered mapping of `member_name`:`member` items. It is only available on the class.

`__new__()`, if specified, must create and return the enum members; it is also a very good idea to set the member's **`_value_`** appropriately. Once all the members are created it is no longer used.

Supported `_sunder_` names

- **`_name_`** – name of the member
- **`_value_`** – value of the member; can be set in `__new__`
- **`_missing_()`** – a lookup function used when a value is not found; may be overridden
- **`_ignore_`** – a list of names, either as a **`list`** or a **`str`**, that will not be transformed into members, and will be removed from the final class
- **`_order_`** – no longer used, kept for backward compatibility (class attribute, removed during class creation)
- **`_generate_next_value_()`** – used to get an appropriate value for an enum member; may be overridden

Note

For standard **`Enum`** classes the next value chosen is the highest value seen incremented by one.

For **`Flag`** classes the next value chosen will be the next highest power-of-two.

- **`_add_alias_()`** – adds a new name as an alias to an existing member.
- **`_add_value_alias_()`** – adds a new value as an alias to an existing member.
- While `_sunder_` names are generally reserved for the further development of the **`Enum`** class and can not be used, some are explicitly allowed:
    - `_repr_*` (e.g. `_repr_html_`), as used in IPython’s rich display [https://ipython.readthedocs.io/en/stable/config/integrating.html#rich-display]

Added in version 3.6: `_missing_`, `_order_`, `_generate_next_value_`

Added in version 3.7: `_ignore_`

Added in version 3.13: `_add_alias_`, `_add_value_alias_`, `_repr_*`

Utilities and Decorators

class enum.auto

auto can be used in place of a value. If used, the *Enum* machinery will call an **`Enum`**’s **`_generate_next_value_()`** to get an appropriate value. For `Enum` and `IntEnum` that appropriate value will be the last value plus one; for `Flag` and `IntFlag` it will be the first power-of-two greater than the highest value; for `StrEnum` it will be the lower-cased version of the member's name. Care must be taken if mixing *auto()* with manually specified values.

auto instances are only resolved when at the top level of an assignment, either by itself or as part of a tuple:

- `FIRST = auto()` will work (auto() is replaced with `1`);
- `SECOND = auto(), -2` will work (auto is replaced with `2`, so `2, -2` is used to create the `SECOND` enum member;
- `THREE = [auto(), -3]` will *not* work (`[<auto instance>, -3]` is used to create the `THREE` enum member)

Changed in version 3.11.1: In prior versions, `auto()` had to be the only thing on the assignment line to work properly.

`_generate_next_value_` can be overridden to customize the values used by *auto*.

Note

in 3.13 the default `_generate_next_value_` will always return the highest member value incremented by 1, and will fail if any member is an incompatible type.

@enum.property

A decorator similar to the built-in *property*, but specifically for enumerations. It allows member attributes to have the same names as members themselves.

Note

the *property* and the member must be defined in separate classes; for example, the *value* and *name* attributes are defined in the *Enum* class, and *Enum* subclasses can define members with the names `value` and `name`.

Added in version 3.11.

@enum.unique

A **`class`** decorator specifically for enumerations. It searches an enumeration’s **`__members__`**, gathering any aliases it finds; if any are found **`ValueError`** is raised with the details:

```
>>> from enum import Enum, unique
>>> @unique
... class Mistake(Enum):
...     ONE = 1
...     TWO = 2
...     THREE = 3
...     FOUR = 3
...
Traceback (most recent call last):
...
ValueError: duplicate values found in <enum 'Mistake'>: FOUR
-> THREE
```

@enum.verify

A **`class`** decorator specifically for enumerations. Members from **`EnumCheck`** are used to specify which constraints should be checked on the decorated enumeration.

Added in version 3.11.

@enum.member

A decorator for use in enums: its target will become a member.

Added in version 3.11.

@enum.nonmember

A decorator for use in enums: its target will not become a member.

Added in version 3.11.

@enum.global_enum

A decorator to change the `str()` and `repr()` of an enum to show its members as belonging to the module instead of its class. Should only be used when the enum members are exported to the module global namespace (see `re.RegexFlag` for an example).

Added in version 3.11.

enum.show_flag_values(*value*)

Return a list of all power-of-two integers contained in a flag *value*.

Added in version 3.11.

enum.bin(*num*, *max_bits=None*)

Like built-in `bin()`, except negative values are represented in two's complement, and the leading bit always indicates sign (`0` implies positive, `1` implies negative).

```
>>> import enum
>>> enum.bin(10)
'0b0 1010'
>>> enum.bin(~10)   # ~10 is -11
'0b1 0101'
```

Added in version 3.10.

Notes

`IntEnum`, **`StrEnum`**, and **`IntFlag`**

These three enum types are designed to be drop-in replacements for existing integer- and string-based values; as such, they have extra limitations:

- `__str__` uses the value and not the name of the enum member
- `__format__`, because it uses `__str__`, will also use the value of the enum member instead of its name

If you do not need/want those limitations, you can either create your own base class by mixing in the `int` or `str` type yourself:

```
>>> from enum import Enum
>>> class MyIntEnum(int, Enum):
...     pass
```

or you can reassign the appropriate **`str()`**, etc., in your enum:

```
>>> from enum import Enum, IntEnum
>>> class MyIntEnum(IntEnum):
...     __str__ = Enum.__str__
```

`graphlib` — Functionality to operate with graph-like structures

Source code: Lib/graphlib.py
[https://github.com/python/cpython/tree/3.14/Lib/graphlib.py]

---

class graphlib.TopologicalSorter(*graph=None*)

Provides functionality to topologically sort a graph of hashable nodes.

A topological order is a linear ordering of the vertices in a graph such that for every directed edge u -> v from vertex u to vertex v, vertex u comes before vertex v in the ordering. For instance, the vertices of the graph may represent tasks to be performed, and the edges may represent constraints that one task must be performed before another; in this example, a topological ordering is just a valid sequence for the tasks. A complete topological ordering is possible if and only if the graph has no directed cycles, that is, if it is a directed acyclic graph.

If the optional *graph* argument is provided it must be a dictionary representing a directed acyclic graph where the keys are nodes and the values are iterables of all predecessors of that node in the graph (the nodes that have edges that point to the value in the key). Additional nodes can be added to the graph using the **`add()`** method.

In the general case, the steps required to perform the sorting of a given graph are as follows:

- Create an instance of the `TopologicalSorter` with an optional initial graph.
- Add additional nodes to the graph.
- Call `prepare()` on the graph.
- While `is_active()` is `True`, iterate over the nodes returned by `get_ready()` and process them. Call `done()` on each node as it finishes processing.

In case just an immediate sorting of the nodes in the graph is required and no parallelism is involved, the convenience method `TopologicalSorter.static_order()` can be used directly:

```
>>> graph = {"D": {"B", "C"}, "C": {"A"}, "B": {"A"}}
>>> ts = TopologicalSorter(graph)
>>> tuple(ts.static_order())
('A', 'C', 'B', 'D')
```

The class is designed to easily support parallel processing of the nodes as they become ready. For instance:

```
topological_sorter = TopologicalSorter()

# Add nodes to 'topological_sorter'...

topological_sorter.prepare()
while topological_sorter.is_active():
    for node in topological_sorter.get_ready():
        # Worker threads or processes take nodes to work on
off the
        # 'task_queue' queue.
        task_queue.put(node)

    # When the work for a node is done, workers put the node
in
    # 'finalized_tasks_queue' so we can get more nodes to
```

```
work on.
    # The definition of 'is_active()' guarantees that, at
this point, at
    # least one node has been placed on 'task_queue' that
hasn't yet
    # been passed to 'done()', so this blocking 'get()' must
(eventually)
    # succeed.  After calling 'done()', we loop back to call
'get_ready()'
    # again, so put newly freed nodes on 'task_queue' as
soon as
    # logically possible.
    node = finalized_tasks_queue.get()
    topological_sorter.done(node)
```

add(*node*, **predecessors*)

Add a new node and its predecessors to the graph. Both the *node* and all elements in *predecessors* must be hashable.

If called multiple times with the same node argument, the set of dependencies will be the union of all dependencies passed in.

It is possible to add a node with no dependencies (*predecessors* is not provided) or to provide a dependency twice. If a node that has not been provided before is included among *predecessors* it will be automatically added to the graph with no predecessors of its own.

Raises **`ValueError`** if called after **`prepare()`**.

prepare()

Mark the graph as finished and check for cycles in the graph. If any cycle is detected, **`CycleError`** will be raised, but **`get_ready()`** can still be used to obtain as many nodes as possible until cycles block more progress. After a call to this function, the graph cannot be modified, and therefore no more nodes can be added using `add()`.

A `ValueError` will be raised if the sort has been started by `static_order()` or `get_ready()`.

Changed in version 3.14: `prepare()` can now be called more than once as long as the sort has not started. Previously this raised `ValueError`.

is_active()

Returns `True` if more progress can be made and `False` otherwise. Progress can be made if cycles do not block the resolution and either there are still nodes ready that haven't yet been returned by `TopologicalSorter.get_ready()` or the number of nodes marked `TopologicalSorter.done()` is less than the number that have been returned by `TopologicalSorter.get_ready()`.

The `__bool__()` method of this class defers to this function, so instead of:

```
if ts.is_active():
    ...
```

it is possible to simply do:

```
if ts:
    ...
```

Raises `ValueError` if called without calling `prepare()` previously.

done(**nodes*)

Marks a set of nodes returned by `TopologicalSorter.get_ready()` as processed, unblocking any successor of each node in *nodes* for being returned in the future by a call to `TopologicalSorter.get_ready()`.

Raises `ValueError` if any node in *nodes* has already been marked as processed by a previous call to this method or if a node was not added to the graph by using `TopologicalSorter.add()`, if called without calling `prepare()` or if node has not yet been returned by `get_ready()`.

get_ready()

Returns a `tuple` with all the nodes that are ready. Initially it returns all nodes with no predecessors, and once those are marked as processed by calling `TopologicalSorter.done()`, further calls will return all new nodes that have all their predecessors already processed. Once no more progress can be made, empty tuples are returned.

Raises `ValueError` if called without calling `prepare()` previously.

static_order()

Returns an iterator object which will iterate over nodes in a topological order. When using this method, `prepare()` and `done()` should not be called. This method is equivalent to:

```
def static_order(self):
    self.prepare()
    while self.is_active():
        node_group = self.get_ready()
        yield from node_group
        self.done(*node_group)
```

The particular order that is returned may depend on the specific order in which the items were inserted in the graph. For example:

```
>>> ts = TopologicalSorter()
>>> ts.add(3, 2, 1)
>>> ts.add(1, 0)
>>> print([*ts.static_order()])
[2, 0, 1, 3]

>>> ts2 = TopologicalSorter()
>>> ts2.add(1, 0)
>>> ts2.add(3, 2, 1)
>>> print([*ts2.static_order()])
[0, 2, 1, 3]
```

This is due to the fact that “0” and “2” are in the same level in the graph (they would have been returned in the same call to `get_ready()`) and the order between them is determined by the order of insertion.

If any cycle is detected, `CycleError` will be raised.

Added in version 3.9.

Exceptions

The `graphlib` module defines the following exception classes:

exception graphlib.CycleError

Subclass of `ValueError` raised by `TopologicalSorter.prepare()` if cycles exist in the working graph. If multiple cycles exist, only one undefined choice among them will be reported and included in the exception.

The detected cycle can be accessed via the second element in the `args` attribute of the exception instance and consists in a list of nodes, such that each node is, in the graph, an immediate predecessor of the next node in the list. In the reported list, the first and the last node will be the same, to make it clear that it is cyclic.

Numeric and Mathematical Modules

The modules described in this chapter provide numeric and math-related functions and data types. The **`numbers`** module defines an abstract hierarchy of numeric types. The **`math`** and **`cmath`** modules contain various mathematical functions for floating-point and complex numbers. The **`decimal`** module supports exact representations of decimal numbers, using arbitrary precision arithmetic.

The following modules are documented in this chapter:

- **`numbers`** — Numeric abstract base classes
    - The numeric tower
    - Notes for type implementers
        - Adding More Numeric ABCs
        - Implementing the arithmetic operations
- **`math`** — Mathematical functions
    - Number-theoretic functions
    - Floating point arithmetic
    - Floating point manipulation functions
    - Power, exponential and logarithmic functions
    - Summation and product functions
    - Angular conversion
    - Trigonometric functions
    - Hyperbolic functions
    - Special functions
    - Constants
- **`cmath`** — Mathematical functions for complex numbers
    - Conversions to and from polar coordinates
    - Power and logarithmic functions
    - Trigonometric functions

    - Hyperbolic functions
    - Classification functions
    - Constants
- `decimal` — Decimal fixed-point and floating-point arithmetic
    - Quick-start tutorial
    - Decimal objects
        - Logical operands
    - Context objects
    - Constants
    - Rounding modes
    - Signals
    - Floating-point notes
        - Mitigating round-off error with increased precision
        - Special values
    - Working with threads
    - Recipes
    - Decimal FAQ
- `fractions` — Rational numbers
- `random` — Generate pseudo-random numbers
    - Bookkeeping functions
    - Functions for bytes
    - Functions for integers
    - Functions for sequences
    - Discrete distributions
    - Real-valued distributions
    - Alternative Generator
    - Notes on Reproducibility
    - Examples
    - Recipes
    - Command-line usage
    - Command-line example
- `statistics` — Mathematical statistics functions
    - Averages and measures of central location
    - Measures of spread

- Statistics for relations between two inputs
- Function details
- Exceptions
- `NormalDist` objects
- Examples and Recipes
    - Classic probability problems
    - Monte Carlo inputs for simulations
    - Approximating binomial distributions
    - Naive bayesian classifier

`numbers` — Numeric abstract base classes

Source code: Lib/numbers.py [https://github.com/python/cpython/tree/3.14/Lib/numbers.py]

The **`numbers`** module (**PEP 3141** [https://peps.python.org/pep-3141/]) defines a hierarchy of numeric abstract base classes which progressively define more operations. None of the types defined in this module are intended to be instantiated.

class numbers.Number

The root of the numeric hierarchy. If you just want to check if an argument *x* is a number, without caring what kind, use `isinstance(x, Number)`.

The numeric tower

class numbers.Complex

Subclasses of this type describe complex numbers and include the operations that work on the built-in **`complex`** type. These are: conversions to **`complex`** and **`bool`**, **`real`**, **`imag`**, `+`, `-`, `*`, `/`, `**`, **`abs()`**, **`conjugate()`**, `==`, and `!=`. All except `-` and `!=` are abstract.

real

Abstract. Retrieves the real component of this number.

imag

Abstract. Retrieves the imaginary component of this number.

abstractmethod conjugate()

Abstract. Returns the complex conjugate. For example, `(1+3j).conjugate() == (1-3j)`.

class numbers.Real

To `Complex`, **`Real`** adds the operations that work on real numbers.

In short, those are: a conversion to `float`, `math.trunc()`, `round()`, `math.floor()`, `math.ceil()`, `divmod()`, `//`, `%`, `<`, `<=`, `>`, and `>=`.

Real also provides defaults for `complex()`, `real`, `imag`, and `conjugate()`.

class numbers.Rational

Subtypes `Real` and adds `numerator` and `denominator` properties. It also provides a default for `float()`.

The `numerator` and `denominator` values should be instances of `Integral` and should be in lowest terms with `denominator` positive.

numerator

Abstract. The numerator of this rational number.

denominator

Abstract. The denominator of this rational number.

class numbers.Integral

Subtypes `Rational` and adds a conversion to `int`. Provides defaults for `float()`, `numerator`, and `denominator`. Adds abstract methods for `pow()` with modulus and bit-string operations: `<<`, `>>`, `&`, `^`, `|`, `~`.

Notes for type implementers

Implementers should be careful to make equal numbers equal and hash them to the same values. This may be subtle if there are two different extensions of the real numbers. For example, `fractions.Fraction` implements `hash()` as follows:

```
def __hash__(self):
    if self.denominator == 1:
        # Get integers right.
        return hash(self.numerator)
    # Expensive check, but definitely correct.
    if self == float(self):
        return hash(float(self))
    else:
        # Use tuple's hash to avoid a high collision rate on
        # simple fractions.
        return hash((self.numerator, self.denominator))
```

Adding More Numeric ABCs

There are, of course, more possible ABCs for numbers, and this would be a poor hierarchy if it precluded the possibility of adding those. You can add `MyFoo` between `Complex` and `Real` with:

```
class MyFoo(Complex): ...
MyFoo.register(Real)
```

Implementing the arithmetic operations

We want to implement the arithmetic operations so that mixed-mode operations either call an implementation whose author knew about the types of both arguments, or convert both to the nearest built in type and do the operation there. For subtypes of `Integral`, this means that `__add__()` and `__radd__()` should be defined as:

```
class MyIntegral(Integral):

    def __add__(self, other):
        if isinstance(other, MyIntegral):
            return do_my_adding_stuff(self, other)
        elif isinstance(other, OtherTypeIKnowAbout):
            return do_my_other_adding_stuff(self, other)
        else:
            return NotImplemented

    def __radd__(self, other):
        if isinstance(other, MyIntegral):
            return do_my_adding_stuff(other, self)
        elif isinstance(other, OtherTypeIKnowAbout):
            return do_my_other_adding_stuff(other, self)
        elif isinstance(other, Integral):
            return int(other) + int(self)
        elif isinstance(other, Real):
            return float(other) + float(self)
        elif isinstance(other, Complex):
            return complex(other) + complex(self)
        else:
            return NotImplemented
```

There are 5 different cases for a mixed-type operation on subclasses of `Complex`. I'll refer to all of the above code that doesn't refer to `MyIntegral` and `OtherTypeIKnowAbout` as "boilerplate". `a` will be an instance of `A`, which is a subtype of `Complex` (`a : A <: Complex`), and `b : B <: Complex`. I'll consider `a + b`:

1. If `A` defines an `__add__()` which accepts `b`, all is well.
2. If `A` falls back to the boilerplate code, and it were to return a value from `__add__()`, we'd miss the possibility that `B` defines a more intelligent `__radd__()`, so the

boilerplate should return `NotImplemented` from **`__add__()`**. (Or `A` may not implement **`__add__()`** at all.)
3. Then `B`'s `__radd__()` gets a chance. If it accepts `a`, all is well.
4. If it falls back to the boilerplate, there are no more possible methods to try, so this is where the default implementation should live.
5. If `B <: A`, Python tries `B.__radd__` before `A.__add__`. This is ok, because it was implemented with knowledge of `A`, so it can handle those instances before delegating to `Complex`.

If `A <: Complex` and `B <: Real` without sharing any other knowledge, then the appropriate shared operation is the one involving the built in `complex`, and both `__radd__()` s land there, so `a+b == b+a`.

Because most of the operations on any given type will be very similar, it can be useful to define a helper function which generates the forward and reverse instances of any given operator. For example, `fractions.Fraction` uses:

```
def _operator_fallbacks(monomorphic_operator,
fallback_operator):
    def forward(a, b):
        if isinstance(b, (int, Fraction)):
            return monomorphic_operator(a, b)
        elif isinstance(b, float):
            return fallback_operator(float(a), b)
        elif isinstance(b, complex):
            return fallback_operator(complex(a), b)
        else:
            return NotImplemented
    forward.__name__ = '__' + fallback_operator.__name__ + '__'
    forward.__doc__ = monomorphic_operator.__doc__

    def reverse(b, a):
        if isinstance(a, Rational):
            # Includes ints.
            return monomorphic_operator(a, b)
        elif isinstance(a, Real):
            return fallback_operator(float(a), float(b))
        elif isinstance(a, Complex):
            return fallback_operator(complex(a), complex(b))
        else:
            return NotImplemented
    reverse.__name__ = '__r' + fallback_operator.__name__ +
'__'
    reverse.__doc__ = monomorphic_operator.__doc__

    return forward, reverse

def _add(a, b):
    """a + b"""
    return Fraction(a.numerator * b.denominator +
                    b.numerator * a.denominator,
                    a.denominator * b.denominator)

__add__, __radd__ = _operator_fallbacks(_add, operator.add)

# ...
```

`math` — Mathematical functions

This module provides access to common mathematical functions and constants, including those defined by the C standard.

These functions cannot be used with complex numbers; use the functions of the same name from the `cmath` module if you require support for complex numbers. The distinction between functions which support complex numbers and those which don’t is made since most users do not want to learn quite as much mathematics as required to understand complex numbers. Receiving an exception instead of a complex result allows earlier detection of the unexpected complex number used as a parameter, so that the programmer can determine how and why it was generated in the first place.

The following functions are provided by this module. Except when explicitly noted otherwise, all return values are floats.

Number-theoretic functions

`comb(n, k)`	Number of ways to choose *k* items from *n* items without repetition and without order
`factorial(n)`	*n* factorial

`gcd(*integers)`	Greatest common divisor of the integer arguments
`isqrt(n)`	Integer square root of a nonnegative integer *n*
`lcm(*integers)`	Least common multiple of the integer arguments
`perm(n, k)`	Number of ways to choose *k* items from *n* items without repetition and with order
Floating point arithmetic	
`ceil(x)`	Ceiling of *x*, the smallest integer greater than or equal to *x*
`fabs(x)`	Absolute value of *x*
`floor(x)`	Floor of *x*, the largest integer less than or equal to *x*
`fma(x, y, z)`	Fused multiply-add operation: `(x * y) + z`

`fmod(x, y)`	Remainder of division `x / y`
`modf(x)`	Fractional and integer parts of *x*
`remainder(x, y)`	Remainder of *x* with respect to *y*
`trunc(x)`	Integer part of *x*

Floating point manipulation functions

`copysign(x, y)`	Magnitude (absolute value) of *x* with the sign of *y*
`frexp(x)`	Mantissa and exponent of *x*
`isclose(a, b, rel_tol, abs_tol)`	Check if the values *a* and *b* are close to each other
`isfinite(x)`	Check if *x* is neither an infinity nor a NaN
`isinf(x)`	Check if *x* is a positive or negative infinity
`isnan(x)`	Check if *x* is a NaN (not a number)

`ldexp(x, i)`	`x * (2**i)`, inverse of function `frexp()`
`nextafter(x, y, steps)`	Floating-point value *steps* steps after *x* towards *y*
`ulp(x)`	Value of the least significant bit of *x*
Power, exponential and logarithmic functions	
`cbrt(x)`	Cube root of *x*
`exp(x)`	*e* raised to the power *x*
`exp2(x)`	*2* raised to the power *x*
`expm1(x)`	*e* raised to the power *x*, minus 1
`log(x, base)`	Logarithm of *x* to the given base (*e* by default)
`log1p(x)`	Natural logarithm of *1+x* (base *e*)
`log2(x)`	Base-2 logarithm of *x*
`log10(x)`	Base-10 logarithm of *x*

`pow(x, y)`	*x* raised to the power *y*
`sqrt(x)`	Square root of *x*
Summation and product functions	
`dist(p, q)`	Euclidean distance between two points *p* and *q* given as an iterable of coordinates
`fsum(iterable)`	Sum of values in the input *iterable*
`hypot(*coordinates)`	Euclidean norm of an iterable of coordinates
`prod(iterable, start)`	Product of elements in the input *iterable* with a *start* value
`sumprod(p, q)`	Sum of products from two iterables *p* and *q*
Angular conversion	
`degrees(x)`	Convert angle *x* from radians to degrees

`radians(x)`	Convert angle *x* from degrees to radians
Trigonometric functions	
`acos(x)`	Arc cosine of *x*
`asin(x)`	Arc sine of *x*
`atan(x)`	Arc tangent of *x*
`atan2(y, x)`	`atan(y / x)`
`cos(x)`	Cosine of *x*
`sin(x)`	Sine of *x*
`tan(x)`	Tangent of *x*
Hyperbolic functions	
`acosh(x)`	Inverse hyperbolic cosine of *x*
`asinh(x)`	Inverse hyperbolic sine of *x*

`atanh(x)`	Inverse hyperbolic tangent of x
`cosh(x)`	Hyperbolic cosine of x
`sinh(x)`	Hyperbolic sine of x
`tanh(x)`	Hyperbolic tangent of x
Special functions	
`erf(x)`	Error function at x
`erfc(x)`	Complementary error function at x
`gamma(x)`	Gamma function at x
`lgamma(x)`	Natural logarithm of the absolute value of the Gamma function at x
Constants	
`pi`	$\pi = 3.141592\ldots$
`e`	$e = 2.718281\ldots$

`tau`	$\tau = 2\pi = 6.283185\ldots$
`inf`	Positive infinity
`nan`	“Not a number” (NaN)

Number-theoretic functions

math.comb(*n*, *k*)

Return the number of ways to choose *k* items from *n* items without repetition and without order.

Evaluates to `n! / (k! * (n - k)!)` when `k <= n` and evaluates to zero when `k > n`.

Also called the binomial coefficient because it is equivalent to the coefficient of k-th term in polynomial expansion of $(1 + x)^n$.

Raises `TypeError` if either of the arguments are not integers. Raises `ValueError` if either of the arguments are negative.

Added in version 3.8.

math.factorial(*n*)

Return factorial of the nonnegative integer *n*.

Changed in version 3.10: Floats with integral values (like `5.0`) are no longer accepted.

math.gcd(**integers*)

Return the greatest common divisor of the specified integer arguments. If any of the arguments is nonzero, then the returned value is the largest positive integer that is a divisor of all arguments. If all arguments are zero, then the returned value is `0`. `gcd()` without arguments returns `0`.

Added in version 3.5.

Changed in version 3.9: Added support for an arbitrary number of arguments. Formerly, only two arguments were supported.

math.isqrt(*n*)

Return the integer square root of the nonnegative integer n. This is the floor of the exact square root of n, or equivalently the greatest integer a such that $a^2 \leq n$.

For some applications, it may be more convenient to have the least integer a such that $n \leq a^2$, or in other words the ceiling of the exact square root of n. For positive n, this can be computed using `a = 1 + isqrt(n - 1)`.

Added in version 3.8.

math.lcm(**integers*)

Return the least common multiple of the specified integer arguments. If all arguments are nonzero, then the returned value is the smallest positive integer that is a multiple of all arguments. If any of the arguments is zero, then the returned value is `0`. `lcm()` without arguments returns `1`.

Added in version 3.9.

math.perm(*n*, *k=None*)

Return the number of ways to choose *k* items from *n* items without repetition and with order.

Evaluates to `n! / (n - k)!` when `k <= n` and evaluates to zero when `k > n`.

If *k* is not specified or is `None`, then *k* defaults to *n* and the function returns `n!`.

Raises `TypeError` if either of the arguments are not integers. Raises `ValueError` if either of the arguments are negative.

Added in version 3.8.

Floating point arithmetic

math.ceil(*x*)

Return the ceiling of *x*, the smallest integer greater than or equal to *x*. If *x* is not a float, delegates to `x.__ceil__`, which should return an `Integral` value.

math.fabs(*x*)

Return the absolute value of *x*.

math.floor(*x*)

Return the floor of *x*, the largest integer less than or equal to *x*. If *x* is not a float, delegates to `x.__floor__`, which should return an `Integral` value.

math.fma(*x*, *y*, *z*)

Fused multiply-add operation. Return `(x * y) + z`, computed as though with infinite precision and range followed by a single round to the `float` format. This operation often provides better accuracy than the direct expression `(x * y) + z`.

This function follows the specification of the fusedMultiplyAdd operation described in the IEEE 754 standard. The standard leaves one case implementation-defined, namely the result of `fma(0, inf, nan)` and `fma(inf, 0, nan)`. In these cases, `math.fma` returns a NaN, and does not raise any exception.

Added in version 3.13.

math.fmod(*x*, *y*)

Return the floating-point remainder of `x / y`, as defined by the platform C library function `fmod(x, y)`. Note that the Python expression `x % y` may not return the same result. The intent of the C standard is that `fmod(x, y)` be exactly (mathematically; to infinite precision) equal to `x - n*y` for some integer *n* such that the result has the same sign as *x* and magnitude less than `abs(y)`. Python's `x % y` returns a result with the sign of *y* instead, and may not be exactly computable for float arguments. For example, `fmod(-1e-100, 1e100)` is `-1e-100`, but the result of Python's `-1e-100 % 1e100` is `1e100-1e-100`, which cannot be represented exactly as a float, and rounds to the surprising `1e100`. For this reason, function **`fmod()`** is generally preferred when working with floats, while Python's `x % y` is preferred when working with integers.

math.modf(*x*)

Return the fractional and integer parts of *x*. Both results carry the sign of *x* and are floats.

Note that **`modf()`** has a different call/return pattern than its C equivalents: it takes a single argument and return a pair of values, rather than returning its second return value through an ‘output parameter’ (there is no such thing in Python).

math.remainder(*x*, *y*)

Return the IEEE 754-style remainder of *x* with respect to *y*. For finite *x* and finite nonzero *y*, this is the difference `x - n*y`, where `n` is the closest integer to the exact value of the quotient `x / y`. If `x / y` is exactly halfway between two consecutive integers, the nearest *even* integer is used for `n`. The remainder `r = remainder(x, y)` thus always satisfies `abs(r) <= 0.5 * abs(y)`.

Special cases follow IEEE 754: in particular, `remainder(x, math.inf)` is *x* for any finite *x*, and `remainder(x, 0)` and `remainder(math.inf, x)` raise **`ValueError`** for any non-NaN *x*. If the result of the remainder operation is zero, that zero will have the same sign as *x*.

On platforms using IEEE 754 binary floating point, the result of this operation is always exactly representable: no rounding error is introduced.

Added in version 3.7.

math.trunc(*x*)

Return *x* with the fractional part removed, leaving the integer part. This rounds toward 0: `trunc()` is equivalent to **`floor()`** for positive *x*, and equivalent to **`ceil()`** for negative *x*. If *x* is not a float, delegates to **`x.__trunc__`**, which should return an **`Integral`** value.

For the **`ceil()`**, **`floor()`**, and **`modf()`** functions, note that *all* floating-point numbers of sufficiently large magnitude are exact integers. Python floats typically carry no more than 53 bits of precision (the same as the platform C double type), in which case any float *x* with `abs(x) >= 2**52` necessarily has no fractional bits.

Floating point manipulation functions

math.copysign(*x*, *y*)

Return a float with the magnitude (absolute value) of *x* but the sign of *y*. On platforms that support signed zeros, `copysign(1.0, -0.0)` returns *-1.0*.

math.frexp(*x*)

Return the mantissa and exponent of *x* as the pair `(m, e)`. *m* is a float and *e* is an integer such that `x == m * 2**e` exactly. If *x* is zero, returns `(0.0, 0)`, otherwise `0.5 <= abs(m) < 1`. This is used to “pick apart” the internal representation of a float in a portable way.

Note that **`frexp()`** has a different call/return pattern than its C equivalents: it takes a single argument and return a pair of values, rather than returning its second return value through an ‘output parameter’ (there is no such thing in Python).

math.isclose(*a*, *b*, *, *rel_tol=1e-09*, *abs_tol=0.0*)

Return `True` if the values *a* and *b* are close to each other and `False` otherwise.

Whether or not two values are considered close is determined according to given absolute and relative tolerances. If no errors occur, the result will be: `abs(a-b) <= max(rel_tol * max(abs(a), abs(b)), abs_tol)`.

rel_tol is the relative tolerance – it is the maximum allowed difference between *a* and *b*, relative to the larger absolute value of *a* or *b*. For example, to set a tolerance of 5%, pass `rel_tol=0.05`. The default tolerance is `1e-09`, which assures that the two values are the same within about 9 decimal digits. *rel_tol* must be nonnegative and less than `1.0`.

abs_tol is the absolute tolerance; it defaults to `0.0` and it must be nonnegative. When comparing `x` to `0.0`, `isclose(x, 0)` is computed as `abs(x) <= rel_tol  * abs(x)`, which is `False` for any nonzero `x` and *rel_tol* less than `1.0`. So add an appropriate positive *abs_tol* argument to the call.

The IEEE 754 special values of `NaN`, `inf`, and `-inf` will be handled according to IEEE rules. Specifically, `NaN` is not considered close to any other value, including `NaN`. `inf` and `-inf` are only considered close to themselves.

Added in version 3.5.

See also

PEP 485 [https://peps.python.org/pep-0485/] – A function for testing approximate equality

math.isfinite(*x*)

Return `True` if *x* is neither an infinity nor a NaN, and `False` otherwise. (Note that `0.0` *is* considered finite.)

Added in version 3.2.

math.isinf(*x*)

Return `True` if *x* is a positive or negative infinity, and `False` otherwise.

math.isnan(*x*)

Return `True` if *x* is a NaN (not a number), and `False` otherwise.

math.ldexp(*x*, *i*)

Return `x * (2**i)`. This is essentially the inverse of function [**`frexp()`**].

math.nextafter(*x*, *y*, *steps=1*)

Return the floating-point value *steps* steps after *x* towards *y*.

If *x* is equal to *y*, return *y*, unless *steps* is zero.

Examples:

- `math.nextafter(x, math.inf)` goes up: towards positive infinity.
- `math.nextafter(x, -math.inf)` goes down: towards minus infinity.
- `math.nextafter(x, 0.0)` goes towards zero.
- `math.nextafter(x, math.copysign(math.inf, x))` goes away from zero.

See also [**`math.ulp()`**].

Added in version 3.9.

Changed in version 3.12: Added the *steps* argument.

math.ulp(*x*)

Return the value of the least significant bit of the float *x*:

- If *x* is a NaN (not a number), return *x*.
- If *x* is negative, return `ulp(-x)`.
- If *x* is a positive infinity, return *x*.
- If *x* is equal to zero, return the smallest positive *denormalized* representable float (smaller than the

minimum positive *normalized* float, `sys.float_info.min`).
- If *x* is equal to the largest positive representable float, return the value of the least significant bit of *x*, such that the first float smaller than *x* is `x - ulp(x)`.
- Otherwise (*x* is a positive finite number), return the value of the least significant bit of *x*, such that the first float bigger than *x* is `x + ulp(x)`.

ULP stands for “Unit in the Last Place”.

See also `math.nextafter()` and `sys.float_info.epsilon`.

Added in version 3.9.

Power, exponential and logarithmic functions

math.cbrt(*x*)

Return the cube root of *x*.

Added in version 3.11.

math.exp(*x*)

Return *e* raised to the power *x*, where *e* = 2.718281… is the base of natural logarithms. This is usually more accurate than `math.e ** x` or `pow(math.e, x)`.

math.exp2(*x*)

Return *2* raised to the power *x*.

Added in version 3.11.

math.expm1(*x*)

Return *e* raised to the power *x*, minus 1. Here *e* is the base of natural logarithms. For small floats *x*, the subtraction in `exp(x) - 1` can result in a significant loss of precision [https://en.wikipedia.org/wiki/Loss_of_significance]; the **`expm1()`** function provides a way to compute this quantity to full precision:

```
>>> from math import exp, expm1
>>> exp(1e-5) - 1  # gives result accurate to 11 places
1.0000050000069649e-05
>>> expm1(1e-5)    # result accurate to full precision
1.0000050000166668e-05
```

Added in version 3.2.

math.log(*x*[, *base*])

With one argument, return the natural logarithm of *x* (to base *e*).

With two arguments, return the logarithm of *x* to the given *base*, calculated as `log(x)/log(base)`.

math.log1p(*x*)

Return the natural logarithm of *1+x* (base *e*). The result is calculated in a way which is accurate for *x* near zero.

math.log2(*x*)

Return the base-2 logarithm of *x*. This is usually more accurate than `log(x, 2)`.

Added in version 3.3.

See also

`int.bit_length()` returns the number of bits necessary to represent an integer in binary, excluding the sign and leading zeros.

math.log10(*x*)

Return the base-10 logarithm of *x*. This is usually more accurate than `log(x, 10)`.

math.pow(*x*, *y*)

Return *x* raised to the power *y*. Exceptional cases follow the IEEE 754 standard as far as possible. In particular, `pow(1.0, x)` and `pow(x, 0.0)` always return `1.0`, even when *x* is a zero or a NaN. If both *x* and *y* are finite, *x* is negative, and *y* is not an integer then `pow(x, y)` is undefined, and raises **`ValueError`**.

Unlike the built-in `**` operator, **`math.pow()`** converts both its arguments to type **`float`**. Use `**` or the built-in **`pow()`** function for computing exact integer powers.

Changed in version 3.11: The special cases `pow(0.0, -inf)` and `pow(-0.0, -inf)` were changed to return `inf` instead of raising **`ValueError`**, for consistency with IEEE 754.

math.sqrt(*x*)

Return the square root of *x*.

Summation and product functions

math.dist(*p*, *q*)

Return the Euclidean distance between two points *p* and *q*, each given as a sequence (or iterable) of coordinates. The two points must have the same dimension.

Roughly equivalent to:

```
sqrt(sum((px - qx) ** 2.0 for px, qx in zip(p, q)))
```

Added in version 3.8.

math.fsum(*iterable*)

Return an accurate floating-point sum of values in the iterable. Avoids loss of precision by tracking multiple intermediate partial sums.

The algorithm's accuracy depends on IEEE-754 arithmetic guarantees and the typical case where the rounding mode is half-even. On some non-Windows builds, the underlying C library uses extended precision addition and may occasionally double-round an intermediate sum causing it to be off in its least significant bit.

For further discussion and two alternative approaches, see the ASPN cookbook recipes for accurate floating-point summation [https://code.activestate.com/recipes/393090-binary-floating-point-summation-accurate-to-full-p/].

math.hypot(**coordinates*)

Return the Euclidean norm, `sqrt(sum(x**2 for x in coordinates))`. This is the length of the vector from the origin to the point given by the coordinates.

For a two dimensional point `(x, y)`, this is equivalent to computing the hypotenuse of a right triangle using the Pythagorean theorem, `sqrt(x*x + y*y)`.

Changed in version 3.8: Added support for n-dimensional points. Formerly, only the two dimensional case was supported.

Changed in version 3.10: Improved the algorithm's accuracy so that the maximum error is under 1 ulp (unit in the last place). More typically, the result is almost always correctly rounded to within 1/2 ulp.

math.prod(*iterable*, *, *start=1*)

Calculate the product of all the elements in the input *iterable*. The default *start* value for the product is `1`.

When the iterable is empty, return the start value. This function is intended specifically for use with numeric values and may reject non-numeric types.

Added in version 3.8.

math.sumprod(*p*, *q*)

Return the sum of products of values from two iterables *p* and *q*.

Raises `ValueError` if the inputs do not have the same length.

Roughly equivalent to:

```
sum(map(operator.mul, p, q, strict=True))
```

For float and mixed int/float inputs, the intermediate products and sums are computed with extended precision.

Added in version 3.12.

Angular conversion

math.degrees(*x*)

Convert angle *x* from radians to degrees.

math.radians(*x*)

Convert angle *x* from degrees to radians.

Trigonometric functions

math.acos(*x*)

Return the arc cosine of *x*, in radians. The result is between `0` and `pi`.

math.asin(*x*)

Return the arc sine of *x*, in radians. The result is between `-pi/2` and `pi/2`.

math.atan(*x*)

Return the arc tangent of *x*, in radians. The result is between `-pi/2` and `pi/2`.

math.atan2(*y*, *x*)

Return `atan(y / x)`, in radians. The result is between `-pi` and `pi`. The vector in the plane from the origin to point `(x, y)` makes this angle with the positive X axis. The point of **`atan2()`** is that the signs of both inputs are known to it, so it can compute the correct quadrant for the angle. For example, `atan(1)` and `atan2(1, 1)` are both `pi/4`, but `atan2(-1, -1)` is `-3*pi/4`.

math.cos(*x*)

Return the cosine of *x* radians.

math.sin(*x*)

Return the sine of *x* radians.

math.tan(*x*)

Return the tangent of *x* radians.

Hyperbolic functions

Hyperbolic functions [https://en.wikipedia.org/wiki/Hyperbolic_functions] are analogs of trigonometric functions that are based on hyperbolas instead of circles.

math.acosh(*x*)

Return the inverse hyperbolic cosine of *x*.

math.asinh(*x*)

Return the inverse hyperbolic sine of *x*.

math.atanh(*x*)

Return the inverse hyperbolic tangent of *x*.

math.cosh(*x*)

Return the hyperbolic cosine of *x*.

math.sinh(*x*)

Return the hyperbolic sine of *x*.

math.tanh(*x*)

Return the hyperbolic tangent of *x*.

Special functions

math.erf(*x*)

Return the error function [https://en.wikipedia.org/wiki/Error_function] at *x*.

The `erf()` function can be used to compute traditional statistical functions such as the cumulative standard normal distribution [https://en.wikipedia.org/wiki/Cumulative_distribution_function]:

```
def phi(x):
    'Cumulative distribution function for the standard
normal distribution'
    return (1.0 + erf(x / sqrt(2.0))) / 2.0
```

Added in version 3.2.

math.erfc(*x*)

Return the complementary error function at *x*. The complementary error function [https://en.wikipedia.org/wiki/Error_function] is defined as `1.0 - erf(x)`. It is used for large values of *x* where a subtraction from one would cause a loss of significance [https://en.wikipedia.org/wiki/Loss_of_significance].

Added in version 3.2.

math.gamma(*x*)

Return the Gamma function [https://en.wikipedia.org/wiki/Gamma_function] at *x*.

Added in version 3.2.

math.lgamma(*x*)

Return the natural logarithm of the absolute value of the Gamma function at *x*.

Added in version 3.2.

Constants

math.pi

The mathematical constant π = 3.141592…, to available precision.

math.e

The mathematical constant *e* = 2.718281…, to available precision.

math.tau

The mathematical constant *τ* = 6.283185…, to available precision. Tau is a circle constant equal to 2*π*, the ratio of a circle’s circumference to its radius. To learn more about Tau, check out Vi Hart’s video Pi is (still) Wrong [https://vimeo.com/147792667], and start celebrating Tau day [https://tauday.com/] by eating twice as much pie!

Added in version 3.6.

math.inf

A floating-point positive infinity. (For negative infinity, use `-math.inf`.) Equivalent to the output of `float('inf')`.

Added in version 3.5.

math.nan

A floating-point “not a number” (NaN) value. Equivalent to the output of `float('nan')`. Due to the requirements of the IEEE-754 standard [https://en.wikipedia.org/wiki/IEEE_754], `math.nan` and `float('nan')` are not considered to equal to any other numeric value, including themselves. To check whether a number is a NaN, use the **`isnan()`** function to test for NaNs instead of `is` or `==`. Example:

```
>>> import math
>>> math.nan == math.nan
False
>>> float('nan') == float('nan')
False
>>> math.isnan(math.nan)
True
>>> math.isnan(float('nan'))
True
```

Added in version 3.5.

Changed in version 3.11: It is now always available.

CPython implementation detail: The `math` module consists mostly of thin wrappers around the platform C math library functions. Behavior in exceptional cases follows Annex F of the C99 standard where appropriate. The current implementation will raise `ValueError` for invalid operations like `sqrt(-1.0)` or `log(0.0)` (where C99 Annex F recommends signaling invalid operation or divide-by-zero), and `OverflowError` for results that overflow (for example, `exp(1000.0)`). A NaN will not be returned from any of the functions above unless one or more of the input arguments was a NaN; in that case, most functions will return a NaN, but (again following C99 Annex F) there are some exceptions to this rule, for example `pow(float('nan'), 0.0)` or `hypot(float('nan'), float('inf'))`.

Note that Python makes no effort to distinguish signaling NaNs from quiet NaNs, and behavior for signaling NaNs remains unspecified. Typical behavior is to treat all NaNs as though they were quiet.

See also

Module `cmath`
Complex number versions of many of these functions.

`cmath` — Mathematical functions for complex numbers

This module provides access to mathematical functions for complex numbers. The functions in this module accept integers, floating-point numbers or complex numbers as arguments. They will also accept any Python object that has either a `__complex__()` or a `__float__()` method: these methods are used to convert the object to a complex or floating-point number, respectively, and the function is then applied to the result of the conversion.

Note

For functions involving branch cuts, we have the problem of deciding how to define those functions on the cut itself. Following Kahan's "Branch cuts for complex elementary functions" paper, as well as Annex G of C99 and later C standards, we use the sign of zero to distinguish one side of the branch cut from the other: for a branch cut along (a portion of) the real axis we look at the sign of the imaginary part, while for a branch cut along the imaginary axis we look at the sign of the real part.

For example, the `cmath.sqrt()` function has a branch cut along the negative real axis. An argument of `-2-0j` is treated as though it lies *below* the branch cut, and so gives a result on the negative imaginary axis:

```
>>> cmath.sqrt(-2-0j)
-1.4142135623730951j
```

But an argument of `-2+0j` is treated as though it lies above the branch cut:

```
>>> cmath.sqrt(-2+0j)
1.4142135623730951j
```

Conversions to and from polar coordinates

`phase(z)`	Return the phase of *z*
`polar(z)`	Return the representation of *z* in polar coordinates
`rect(r, phi)`	Return the complex number *z* with polar coordinates *r* and *phi*

Power and logarithmic functions

`exp(z)`	Return *e* raised to the power *z*
`log(z[, base])`	Return the logarithm of *z* to the given *base* (*e* by default)
`log10(z)`	Return the base-10 logarithm of *z*
`sqrt(z)`	Return the square root of *z*

Trigonometric functions

`acos(z)`	Return the arc cosine of *z*
`asin(z)`	Return the arc sine of *z*
`atan(z)`	Return the arc tangent of *z*
`cos(z)`	Return the cosine of *z*
`sin(z)`	Return the sine of *z*
`tan(z)`	Return the tangent of *z*

Hyperbolic functions

`acosh(z)`	Return the inverse hyperbolic cosine of *z*
`asinh(z)`	Return the inverse hyperbolic sine of *z*
`atanh(z)`	Return the inverse hyperbolic tangent of *z*

`cosh(z)`	Return the hyperbolic cosine of *z*
`sinh(z)`	Return the hyperbolic sine of *z*
`tanh(z)`	Return the hyperbolic tangent of *z*
Classification functions	
`isfinite(z)`	Check if all components of *z* are finite
`isinf(z)`	Check if any component of *z* is infinite
`isnan(z)`	Check if any component of *z* is a NaN
`isclose(a, b, *, rel_tol, abs_tol)`	Check if the values *a* and *b* are close to each other
Constants	
`pi`	$\pi = 3.141592\ldots$
`e`	$e = 2.718281\ldots$

`tau`	$\tau = 2\pi = 6.283185\ldots$
`inf`	Positive infinity
`infj`	Pure imaginary infinity
`nan`	“Not a number” (NaN)
`nanj`	Pure imaginary NaN

Conversions to and from polar coordinates

A Python complex number `z` is stored internally using *rectangular* or *Cartesian* coordinates. It is completely determined by its *real part* `z.real` and its *imaginary part* `z.imag`.

Polar coordinates give an alternative way to represent a complex number. In polar coordinates, a complex number *z* is defined by the modulus *r* and the phase angle *phi*. The modulus *r* is the distance from *z* to the origin, while the phase *phi* is the counterclockwise angle, measured in radians, from the positive x-axis to the line segment that joins the origin to *z*.

The following functions can be used to convert from the native rectangular coordinates to polar coordinates and back.

cmath.phase(*z*)

Return the phase of *z* (also known as the *argument* of *z*), as a float. `phase(z)` is equivalent to `math.atan2(z.imag, z.real)`. The result lies in the range [-*π*, *π*], and the branch cut for this operation lies along the negative real axis. The sign of the result is the same as the sign of `z.imag`, even when `z.imag` is zero:

```
>>> phase(-1+0j)
3.141592653589793
>>> phase(-1-0j)
-3.141592653589793
```

Note

The modulus (absolute value) of a complex number *z* can be computed using the built-in `abs()` function. There is no separate `cmath` module function for this operation.

cmath.polar(*z*)

Return the representation of *z* in polar coordinates. Returns a pair `(r, phi)` where *r* is the modulus of *z* and *phi* is the phase of *z*. `polar(z)` is equivalent to `(abs(z), phase(z))`.

cmath.rect(*r*, *phi*)

Return the complex number *z* with polar coordinates *r* and *phi*. Equivalent to `complex(r * math.cos(phi), r * math.sin(phi))`.

Power and logarithmic functions

cmath.exp(*z*)

Return *e* raised to the power *z*, where *e* is the base of natural logarithms.

cmath.log(*z*[, *base*])

Return the logarithm of *z* to the given *base*. If the *base* is not specified, returns the natural logarithm of *z*. There is one branch cut, from 0 along the negative real axis to -∞.

cmath.log10(*z*)

Return the base-10 logarithm of *z*. This has the same branch cut as `log()`.

cmath.sqrt(*z*)

Return the square root of *z*. This has the same branch cut as `log()`.

Trigonometric functions

cmath.acos(*z*)

Return the arc cosine of *z*. There are two branch cuts: One extends right from 1 along the real axis to ∞. The other extends left from -1 along the real axis to -∞.

cmath.asin(*z*)

Return the arc sine of *z*. This has the same branch cuts as `acos()`.

cmath.atan(*z*)

Return the arc tangent of *z*. There are two branch cuts: One extends from `1j` along the imaginary axis to `∞j`. The other extends from `-1j` along the imaginary axis to `-∞j`.

cmath.cos(*z*)

Return the cosine of *z*.

cmath.sin(*z*)

Return the sine of *z*.

cmath.tan(*z*)

Return the tangent of *z*.

Hyperbolic functions

cmath.acosh(*z*)

Return the inverse hyperbolic cosine of *z*. There is one branch cut, extending left from 1 along the real axis to `-∞`.

cmath.asinh(*z*)

Return the inverse hyperbolic sine of *z*. There are two branch cuts: One extends from `1j` along the imaginary axis to `∞j`. The other extends from `-1j` along the imaginary axis to `-∞j`.

cmath.atanh(*z*)

Return the inverse hyperbolic tangent of *z*. There are two branch cuts: One extends from `1` along the real axis to `∞`. The other extends from `-1` along the real axis to `-∞`.

cmath.cosh(*z*)

Return the hyperbolic cosine of *z*.

cmath.sinh(*z*)

Return the hyperbolic sine of *z*.

cmath.tanh(*z*)

Return the hyperbolic tangent of *z*.

Classification functions

cmath.isfinite(*z*)

Return `True` if both the real and imaginary parts of *z* are finite, and `False` otherwise.

Added in version 3.2.

cmath.isinf(*z*)

Return `True` if either the real or the imaginary part of *z* is an infinity, and `False` otherwise.

cmath.isnan(*z*)

Return `True` if either the real or the imaginary part of *z* is a NaN, and `False` otherwise.

cmath.isclose(*a*, *b*, *, *rel_tol=1e-09*, *abs_tol=0.0*)

Return `True` if the values *a* and *b* are close to each other and `False` otherwise.

Whether or not two values are considered close is determined according to given absolute and relative tolerances. If no errors occur, the result will be: `abs(a-b) <= max(rel_tol * max(abs(a), abs(b)), abs_tol)`.

rel_tol is the relative tolerance – it is the maximum allowed difference between *a* and *b*, relative to the larger absolute value of *a* or *b*. For example, to set a tolerance of 5%, pass `rel_tol=0.05`. The default tolerance is `1e-09`, which assures that the two values are the same within about 9 decimal digits. *rel_tol* must be nonnegative and less than `1.0`.

abs_tol is the absolute tolerance; it defaults to `0.0` and it must be nonnegative. When comparing `x` to `0.0`, `isclose(x, 0)` is computed as `abs(x) <= rel_tol  * abs(x)`, which is `False` for any `x` and rel_tol less than `1.0`. So add an appropriate positive abs_tol argument to the call.

The IEEE 754 special values of `NaN`, `inf`, and `-inf` will be handled according to IEEE rules. Specifically, `NaN` is not considered close to any other value, including `NaN`. `inf` and `-inf` are only considered close to themselves.

Added in version 3.5.

See also

PEP 485 [https://peps.python.org/pep-0485/] – A function for testing approximate equality

Constants

cmath.pi

The mathematical constant π, as a float.

cmath.e

The mathematical constant e, as a float.

cmath.tau

The mathematical constant τ, as a float.

Added in version 3.6.

cmath.inf

Floating-point positive infinity. Equivalent to `float('inf')`.

Added in version 3.6.

cmath.infj

Complex number with zero real part and positive infinity imaginary part. Equivalent to `complex(0.0, float('inf'))`.

Added in version 3.6.

cmath.nan

A floating-point “not a number” (NaN) value. Equivalent to `float('nan')`. See also `math.nan`.

Added in version 3.6.

cmath.nanj

Complex number with zero real part and NaN imaginary part. Equivalent to `complex(0.0, float('nan'))`.

Added in version 3.6.

Note that the selection of functions is similar, but not identical, to that in module `math`. The reason for having two modules is that some users aren’t interested in complex numbers, and perhaps don’t even know what they are. They would rather have `math.sqrt(-1)` raise an exception than return a complex number. Also note that the functions defined in `cmath` always return a complex number, even if the answer can be expressed as a real number (in which case the complex number has an imaginary part of zero).

A note on branch cuts: They are curves along which the given function fails to be continuous. They are a necessary feature of many complex functions. It is assumed that if you need to compute with complex functions, you will understand about branch cuts. Consult almost any (not too elementary) book on complex variables for enlightenment. For information of the proper choice of branch cuts for numerical purposes, a good reference should be the following:

See also

Kahan, W: Branch cuts for complex elementary functions; or, Much ado about nothing’s sign bit. In Iserles, A., and

Powell, M. (eds.), The state of the art in numerical analysis. Clarendon Press (1987) pp165–211.

`decimal` — Decimal fixed-point and floating-point arithmetic

Source code: Lib/decimal.py
[https://github.com/python/cpython/tree/3.14/Lib/decimal.py]

The **`decimal`** module provides support for fast correctly rounded decimal floating-point arithmetic. It offers several advantages over the **`float`** datatype:

- Decimal "is based on a floating-point model [https://speleotrove.com/decimal/damodel.html#refnumber] which was designed with people in mind, and necessarily has a paramount guiding principle – computers must provide an arithmetic that works in the same way as the arithmetic that people learn at school." – excerpt from the decimal arithmetic specification.

- Decimal numbers can be represented exactly. In contrast, numbers like `1.1` and `2.2` do not have exact representations in binary floating point. End users typically would not expect `1.1 + 2.2` to display as `3.3000000000000003` as it does with binary floating point.

- The exactness carries over into arithmetic. In decimal floating point, `0.1 + 0.1 + 0.1 - 0.3` is exactly equal to zero. In binary floating point, the result is `5.5511151231257827e-017`. While near to zero, the differences prevent reliable equality testing and differences can accumulate. For this reason, decimal is preferred in accounting applications which have strict equality invariants.

- The decimal module incorporates a notion of significant places so that `1.30 + 1.20` is `2.50`. The trailing zero is kept to indicate significance. This is the customary presentation for monetary applications. For multiplication, the “schoolbook” approach uses all the figures in the multiplicands. For instance, `1.3 * 1.2` gives `1.56` while `1.30 * 1.20` gives `1.5600`.

- Unlike hardware based binary floating point, the decimal module has a user alterable precision (defaulting to 28 places) which can be as large as needed for a given problem:

  ```
  >>> from decimal import *
  >>> getcontext().prec = 6
  >>> Decimal(1) / Decimal(7)
  Decimal('0.142857')
  >>> getcontext().prec = 28
  >>> Decimal(1) / Decimal(7)
  Decimal('0.1428571428571428571428571429')
  ```

- Both binary and decimal floating point are implemented in terms of published standards. While the built-in float type exposes only a modest portion of its capabilities, the decimal module exposes all required parts of the standard. When needed, the programmer has full control over rounding and signal handling. This includes an option to enforce exact arithmetic by using exceptions to block any inexact operations.

- The decimal module was designed to support “without prejudice, both exact unrounded decimal arithmetic (sometimes called fixed-point arithmetic) and rounded floating-point arithmetic.” – excerpt from the decimal arithmetic specification.

The module design is centered around three concepts: the decimal number, the context for arithmetic, and signals.

A decimal number is immutable. It has a sign, coefficient digits, and an exponent. To preserve significance, the coefficient digits do not truncate trailing zeros. Decimals also include special values such as `Infinity`, `-Infinity`, and `NaN`. The standard also differentiates `-0` from `+0`.

The context for arithmetic is an environment specifying precision, rounding rules, limits on exponents, flags indicating the results of operations, and trap enablers which determine whether signals are treated as exceptions. Rounding options include `ROUND_CEILING`, `ROUND_DOWN`, `ROUND_FLOOR`, `ROUND_HALF_DOWN`, `ROUND_HALF_EVEN`, `ROUND_HALF_UP`, `ROUND_UP`, and `ROUND_05UP`.

Signals are groups of exceptional conditions arising during the course of computation. Depending on the needs of the application, signals may be ignored, considered as informational, or treated as exceptions. The signals in the decimal module are: `Clamped`, `InvalidOperation`, `DivisionByZero`, `Inexact`, `Rounded`, `Subnormal`, `Overflow`, `Underflow` and `FloatOperation`.

For each signal there is a flag and a trap enabler. When a signal is encountered, its flag is set to one, then, if the trap enabler is set to one, an exception is raised. Flags are sticky, so the user needs to reset them before monitoring a calculation.

See also

- IBM’s General Decimal Arithmetic Specification, The General Decimal Arithmetic Specification [https://speleotrove.com/decimal/decarith.html].

Quick-start tutorial

The usual start to using decimals is importing the module, viewing the current context with `getcontext()` and, if necessary, setting new values for precision, rounding, or enabled traps:

```
>>> from decimal import *
>>> getcontext()
Context(prec=28, rounding=ROUND_HALF_EVEN, Emin=-999999, Emax=999999,
        capitals=1, clamp=0, flags=[], traps=[Overflow,
DivisionByZero,
        InvalidOperation])

>>> getcontext().prec = 7       # Set a new precision
```

Decimal instances can be constructed from integers, strings, floats, or tuples. Construction from an integer or a float performs an exact conversion of the value of that integer or float. Decimal numbers include special values such as `NaN` which stands for “Not a number”, positive and negative `Infinity`, and `-0`:

```
>>> getcontext().prec = 28
>>> Decimal(10)
Decimal('10')
>>> Decimal('3.14')
Decimal('3.14')
>>> Decimal(3.14)
Decimal('3.140000000000000124344978758017532527446746826171875')
>>> Decimal((0, (3, 1, 4), -2))
Decimal('3.14')
>>> Decimal(str(2.0 ** 0.5))
Decimal('1.4142135623730951')
>>> Decimal(2) ** Decimal('0.5')
Decimal('1.414213562373095048801688724')
>>> Decimal('NaN')
Decimal('NaN')
>>> Decimal('-Infinity')
Decimal('-Infinity')
```

If the **`FloatOperation`** signal is trapped, accidental mixing of decimals and floats in constructors or ordering comparisons raises an exception:

```
>>> c = getcontext()
>>> c.traps[FloatOperation] = True
>>> Decimal(3.14)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
decimal.FloatOperation: [<class 'decimal.FloatOperation'>]
>>> Decimal('3.5') < 3.7
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
decimal.FloatOperation: [<class 'decimal.FloatOperation'>]
>>> Decimal('3.5') == 3.5
True
```

Added in version 3.3.

The significance of a new Decimal is determined solely by the number of digits input. Context precision and rounding only come into play during arithmetic operations.

```
>>> getcontext().prec = 6
>>> Decimal('3.0')
Decimal('3.0')
>>> Decimal('3.1415926535')
Decimal('3.1415926535')
>>> Decimal('3.1415926535') + Decimal('2.7182818285')
Decimal('5.85987')
>>> getcontext().rounding = ROUND_UP
>>> Decimal('3.1415926535') + Decimal('2.7182818285')
Decimal('5.85988')
```

If the internal limits of the C version are exceeded, constructing a decimal raises **`InvalidOperation`**:

```
>>> Decimal("1e9999999999999999999")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
decimal.InvalidOperation: [<class 'decimal.InvalidOperation'>]
```

Changed in version 3.3.

Decimals interact well with much of the rest of Python. Here is a small decimal floating-point flying circus:

```
>>> data = list(map(Decimal, '1.34 1.87 3.45 2.35 1.00 0.03
9.25'.split()))
>>> max(data)
Decimal('9.25')
>>> min(data)
Decimal('0.03')
>>> sorted(data)
[Decimal('0.03'), Decimal('1.00'), Decimal('1.34'), Decimal('1.87'),
 Decimal('2.35'), Decimal('3.45'), Decimal('9.25')]
>>> sum(data)
Decimal('19.29')
>>> a,b,c = data[:3]
>>> str(a)
'1.34'
>>> float(a)
1.34
>>> round(a, 1)
```

```
Decimal('1.3')
>>> int(a)
1
>>> a * 5
Decimal('6.70')
>>> a * b
Decimal('2.5058')
>>> c % a
Decimal('0.77')
```

Decimals can be formatted (with **`format()`** built-in or f-strings) in fixed-point or scientific notation, using the same formatting syntax (see Format Specification Mini-Language) as builtin **`float`** type:

```
>>> format(Decimal('2.675'), "f")
'2.675'
>>> format(Decimal('2.675'), ".2f")
'2.68'
>>> f"{Decimal('2.675'):.2f}"
'2.68'
>>> format(Decimal('2.675'), ".2e")
'2.68e+0'
>>> with localcontext() as ctx:
...     ctx.rounding = ROUND_DOWN
...     print(format(Decimal('2.675'), ".2f"))
...
2.67
```

And some mathematical functions are also available to Decimal:

```
>>> getcontext().prec = 28
>>> Decimal(2).sqrt()
Decimal('1.414213562373095048801688724')
>>> Decimal(1).exp()
Decimal('2.718281828459045235360287471')
>>> Decimal('10').ln()
Decimal('2.302585092994045684017991455')
>>> Decimal('10').log10()
Decimal('1')
```

The **`quantize()`** method rounds a number to a fixed exponent. This method is useful for monetary applications that often round results to a fixed number of places:

```
>>> Decimal('7.325').quantize(Decimal('.01'), rounding=ROUND_DOWN)
Decimal('7.32')
>>> Decimal('7.325').quantize(Decimal('1.'), rounding=ROUND_UP)
Decimal('8')
```

As shown above, the `getcontext()` function accesses the current context and allows the settings to be changed. This approach meets the needs of most applications.

For more advanced work, it may be useful to create alternate contexts using the `Context()` constructor. To make an alternate active, use the `setcontext()` function.

In accordance with the standard, the `decimal` module provides two ready to use standard contexts, `BasicContext` and `ExtendedContext`. The former is especially useful for debugging because many of the traps are enabled:

```
>>> myothercontext = Context(prec=60, rounding=ROUND_HALF_DOWN)
>>> setcontext(myothercontext)
>>> Decimal(1) / Decimal(7)
Decimal('0.142857142857142857142857142857142857142857142857142857142857')

>>> ExtendedContext
Context(prec=9, rounding=ROUND_HALF_EVEN, Emin=-999999, Emax=999999,
        capitals=1, clamp=0, flags=[], traps=[])
>>> setcontext(ExtendedContext)
>>> Decimal(1) / Decimal(7)
Decimal('0.142857143')
>>> Decimal(42) / Decimal(0)
Decimal('Infinity')

>>> setcontext(BasicContext)
>>> Decimal(42) / Decimal(0)
Traceback (most recent call last):
  File "<pyshell#143>", line 1, in -toplevel-
    Decimal(42) / Decimal(0)
DivisionByZero: x / 0
```

Contexts also have signal flags for monitoring exceptional conditions encountered during computations. The flags remain set until explicitly cleared, so it is best to clear the flags before each set of monitored computations by using the `clear_flags()` method.

```
>>> setcontext(ExtendedContext)
>>> getcontext().clear_flags()
>>> Decimal(355) / Decimal(113)
Decimal('3.14159292')
>>> getcontext()
Context(prec=9, rounding=ROUND_HALF_EVEN, Emin=-999999, Emax=999999,
        capitals=1, clamp=0, flags=[Inexact, Rounded], traps=[])
```

The *flags* entry shows that the rational approximation to pi was rounded (digits beyond the context precision were thrown away) and that the result is inexact (some of the discarded digits were non-zero).

Individual traps are set using the dictionary in the `traps` attribute of a context:

```
>>> setcontext(ExtendedContext)
>>> Decimal(1) / Decimal(0)
Decimal('Infinity')
>>> getcontext().traps[DivisionByZero] = 1
>>> Decimal(1) / Decimal(0)
Traceback (most recent call last):
  File "<pyshell#112>", line 1, in -toplevel-
    Decimal(1) / Decimal(0)
DivisionByZero: x / 0
```

Most programs adjust the current context only once, at the beginning of the program. And, in many applications, data is converted to `Decimal` with a single cast inside a loop. With context set and decimals created, the bulk of the program manipulates the data no differently than with other Python numeric types.

Decimal objects

class decimal.Decimal(*value='0'*, *context=None*)

Construct a new `Decimal` object based from *value*.

value can be an integer, string, tuple, `float`, or another `Decimal` object. If no *value* is given, returns `Decimal('0')`. If *value* is a string, it should conform to the decimal numeric string syntax after leading and trailing whitespace characters, as well as underscores throughout, are removed:

```
sign           ::=  '+' | '-'
digit          ::=  '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7'
| '8' | '9'
indicator      ::=  'e' | 'E'
digits         ::=  digit [digit]...
decimal-part   ::=  digits '.' [digits] | ['.'] digits
exponent-part  ::=  indicator [sign] digits
infinity       ::=  'Infinity' | 'Inf'
nan            ::=  'NaN' [digits] | 'sNaN' [digits]
numeric-value  ::=  decimal-part [exponent-part] | infinity
numeric-string ::=  [sign] numeric-value | [sign] nan
```

Other Unicode decimal digits are also permitted where `digit` appears above. These include decimal digits from various other alphabets (for example, Arabic-Indic and Devanāgarī digits) along with the fullwidth digits `'\uff10'` through `'\uff19'`. Case is not significant, so, for example, `inf`, `Inf`, `INFINITY`, and `iNfINity` are all acceptable spellings for positive infinity.

If *value* is a **`tuple`**, it should have three components, a sign (`0` for positive or `1` for negative), a **`tuple`** of digits, and an integer exponent. For example, `Decimal((0, (1, 4, 1, 4), -3))` returns `Decimal('1.414')`.

If *value* is a **`float`**, the binary floating-point value is losslessly converted to its exact decimal equivalent. This conversion can often require 53 or more digits of precision. For example, `Decimal(float('1.1'))` converts to `Decimal('1.100000000000000088817841970012523233890533447265625')`.

The *context* precision does not affect how many digits are stored. That is determined exclusively by the number of digits in *value*. For example, `Decimal('3.00000')` records all five zeros even if the context precision is only three.

The purpose of the *context* argument is determining what to do if *value* is a malformed string. If the context traps

`InvalidOperation`, an exception is raised; otherwise, the constructor returns a new Decimal with the value of `NaN`.

Once constructed, **`Decimal`** objects are immutable.

Changed in version 3.2: The argument to the constructor is now permitted to be a **`float`** instance.

Changed in version 3.3: **`float`** arguments raise an exception if the **`FloatOperation`** trap is set. By default the trap is off.

Changed in version 3.6: Underscores are allowed for grouping, as with integral and floating-point literals in code.

Decimal floating-point objects share many properties with the other built-in numeric types such as **`float`** and **`int`**. All of the usual math operations and special methods apply. Likewise, decimal objects can be copied, pickled, printed, used as dictionary keys, used as set elements, compared, sorted, and coerced to another type (such as **`float`** or **`int`**).

There are some small differences between arithmetic on Decimal objects and arithmetic on integers and floats. When the remainder operator `%` is applied to Decimal objects, the sign of the result is the sign of the *dividend* rather than the sign of the divisor:

```
>>> (-7) % 4
1
>>> Decimal(-7) % Decimal(4)
Decimal('-3')
```

The integer division operator `//` behaves analogously, returning the integer part of the true quotient (truncating towards zero) rather than its floor, so as to preserve the usual identity `x == (x // y) * y + x % y`:

```
>>> -7 // 4
-2
>>> Decimal(-7) // Decimal(4)
Decimal('-1')
```

The `%` and `//` operators implement the `remainder` and `divide-integer` operations (respectively) as described in the specification.

Decimal objects cannot generally be combined with floats or instances of **`fractions.Fraction`** in arithmetic operations: an attempt to add a **`Decimal`** to a **`float`**, for example, will raise a **`TypeError`**. However, it is possible to use Python's comparison operators to compare a **`Decimal`** instance `x` with another number `y`. This avoids confusing results when doing equality comparisons between numbers of different types.

Changed in version 3.2: Mixed-type comparisons between **`Decimal`** instances and other numeric types are now fully supported.

In addition to the standard numeric properties, decimal floating-point objects also have a number of specialized methods:

adjusted()

Return the adjusted exponent after shifting out the coefficient's rightmost digits until only the lead digit remains: `Decimal('321e+5').adjusted()` returns seven. Used for determining the position of the most significant digit with respect to the decimal point.

as_integer_ratio()

Return a pair `(n, d)` of integers that represent the given **`Decimal`** instance as a fraction, in lowest terms and with a positive denominator:

```
>>> Decimal('-3.14').as_integer_ratio()
(-157, 50)
```

The conversion is exact. Raise OverflowError on infinities and ValueError on NaNs.

Added in version 3.6.

as_tuple()

Return a named tuple representation of the number: `DecimalTuple(sign, digits, exponent)`.

canonical()

Return the canonical encoding of the argument. Currently, the encoding of a **`Decimal`** instance is always canonical, so this operation returns its argument unchanged.

compare(*other*, *context=None*)

Compare the values of two Decimal instances. **`compare()`** returns a Decimal instance, and if either operand is a NaN then the result is a NaN:

```
a or b is a NaN  ==> Decimal('NaN')
a < b            ==> Decimal('-1')
a == b           ==> Decimal('0')
a > b            ==> Decimal('1')
```

compare_signal(*other*, *context=None*)

This operation is identical to the **`compare()`** method, except that all NaNs signal. That is, if neither operand is a signaling NaN then any quiet NaN operand is treated as though it were a signaling NaN.

compare_total(*other*, *context=None*)

Compare two operands using their abstract representation rather than their numerical value. Similar to the **`compare()`** method, but the result gives a total ordering on **`Decimal`** instances. Two **`Decimal`** instances with the same numeric value but different representations compare unequal in this ordering:

```
>>> Decimal('12.0').compare_total(Decimal('12'))
Decimal('-1')
```

Quiet and signaling NaNs are also included in the total ordering. The result of this function is `Decimal('0')` if both operands have the same representation, `Decimal('-1')` if the first operand is lower in the total order than the second, and `Decimal('1')` if the first operand is higher in the total order than the second operand. See the specification for details of the total order.

This operation is unaffected by context and is quiet: no flags are changed and no rounding is performed. As an exception, the C version may raise InvalidOperation if the second operand cannot be converted exactly.

compare_total_mag(*other*, *context=None*)

Compare two operands using their abstract representation rather than their value as in **`compare_total()`**, but ignoring the sign of each operand. `x.compare_total_mag(y)` is equivalent to `x.copy_abs().compare_total(y.copy_abs())`.

This operation is unaffected by context and is quiet: no flags are changed and no rounding is performed. As an exception, the C version may raise InvalidOperation if the second operand cannot be converted exactly.

conjugate()

Just returns self, this method is only to comply with the Decimal Specification.

copy_abs()

Return the absolute value of the argument. This operation is unaffected by the context and is quiet: no flags are changed and no rounding is performed.

copy_negate()

Return the negation of the argument. This operation is unaffected by the context and is quiet: no flags are changed and no rounding is performed.

copy_sign(*other*, *context=None*)

Return a copy of the first operand with the sign set to be the same as the sign of the second operand. For example:

```
>>> Decimal('2.3').copy_sign(Decimal('-1.5'))
Decimal('-2.3')
```

This operation is unaffected by context and is quiet: no flags are changed and no rounding is performed. As an exception, the C version may raise InvalidOperation if the second operand cannot be converted exactly.

exp(*context=None*)

Return the value of the (natural) exponential function `e**x` at the given number. The result is correctly rounded using the **`ROUND_HALF_EVEN`** rounding mode.

```
>>> Decimal(1).exp()
Decimal('2.718281828459045235360287471')
>>> Decimal(321).exp()
Decimal('2.561702493119680037517373933E+139')
```

classmethod from_float(*f*, */*)

Alternative constructor that only accepts instances of **`float`** or **`int`**.

Note `Decimal.from_float(0.1)` is not the same as `Decimal('0.1')`. Since 0.1 is not exactly representable in binary floating point, the value is stored as the nearest representable value which is `0x1.999999999999ap-4`. That equivalent value in decimal is `0.1000000000000000055511151231257827021181583404541015625`.

Note

From Python 3.2 onwards, a **`Decimal`** instance can also be constructed directly from a **`float`**.

```
>>> Decimal.from_float(0.1)
Decimal('0.1000000000000000055511151231257827021181583404541015625')
>>> Decimal.from_float(float('nan'))
Decimal('NaN')
>>> Decimal.from_float(float('inf'))
Decimal('Infinity')
```

```
>>> Decimal.from_float(float('-inf'))
Decimal('-Infinity')
```

Added in version 3.1.

classmethod from_number(*number*, */*)

Alternative constructor that only accepts instances of **`float`**, **`int`** or **`Decimal`**, but not strings or tuples.

```
>>> Decimal.from_number(314)
Decimal('314')
>>> Decimal.from_number(0.1)
Decimal('0.1000000000000000055511151231257827021181583404541015625')
>>> Decimal.from_number(Decimal('3.14'))
Decimal('3.14')
```

Added in version 3.14.

fma(*other*, *third*, *context=None*)

Fused multiply-add. Return self*other+third with no rounding of the intermediate product self*other.

```
>>> Decimal(2).fma(3, 5)
Decimal('11')
```

is_canonical()

Return **`True`** if the argument is canonical and **`False`** otherwise. Currently, a **`Decimal`** instance is always canonical, so this operation always returns **`True`**.

is_finite()

Return **`True`** if the argument is a finite number, and **`False`** if the argument is an infinity or a NaN.

is_infinite()

Return **`True`** if the argument is either positive or negative infinity and **`False`** otherwise.

is_nan()

Return **`True`** if the argument is a (quiet or signaling) NaN and **`False`** otherwise.

is_normal(*context=None*)

Return `True` if the argument is a *normal* finite number. Return `False` if the argument is zero, subnormal, infinite or a NaN.

is_qnan()

Return `True` if the argument is a quiet NaN, and `False` otherwise.

is_signed()

Return `True` if the argument has a negative sign and `False` otherwise. Note that zeros and NaNs can both carry signs.

is_snan()

Return `True` if the argument is a signaling NaN and `False` otherwise.

is_subnormal(*context=None*)

Return `True` if the argument is subnormal, and `False` otherwise.

is_zero()

Return `True` if the argument is a (positive or negative) zero and `False` otherwise.

ln(*context=None*)

Return the natural (base e) logarithm of the operand. The result is correctly rounded using the `ROUND_HALF_EVEN` rounding mode.

log10(*context=None*)

Return the base ten logarithm of the operand. The result is correctly rounded using the `ROUND_HALF_EVEN` rounding mode.

logb(*context=None*)

For a nonzero number, return the adjusted exponent of its operand as a `Decimal` instance. If the operand is a zero then `Decimal('-Infinity')` is returned and the `DivisionByZero` flag is raised. If the operand is an infinity then `Decimal('Infinity')` is returned.

logical_and(*other*, *context=None*)

`logical_and()` is a logical operation which takes two *logical operands* (see Logical operands). The result is the digit-wise `and` of the two operands.

logical_invert(*context=None*)

`logical_invert()` is a logical operation. The result is the digit-wise inversion of the operand.

logical_or(*other*, *context=None*)

`logical_or()` is a logical operation which takes two *logical operands* (see Logical operands). The result is the digit-wise `or` of the two operands.

logical_xor(*other*, *context=None*)

`logical_xor()` is a logical operation which takes two *logical operands* (see Logical operands). The result is the digit-wise exclusive or of the two operands.

max(*other*, *context=None*)

Like `max(self, other)` except that the context rounding rule is applied before returning and that `NaN` values are either signaled or ignored (depending on the context and whether they are signaling or quiet).

max_mag(*other*, *context=None*)

Similar to the **`max()`** method, but the comparison is done using the absolute values of the operands.

min(*other*, *context=None*)

Like `min(self, other)` except that the context rounding rule is applied before returning and that `NaN` values are either signaled or ignored (depending on the context and whether they are signaling or quiet).

min_mag(*other*, *context=None*)

Similar to the `min()` method, but the comparison is done using the absolute values of the operands.

next_minus(*context=None*)

Return the largest number representable in the given context (or in the current thread’s context if no context is given) that is smaller than the given operand.

next_plus(*context=None*)

Return the smallest number representable in the given context (or in the current thread’s context if no context is given) that is larger than the given operand.

next_toward(*other*, *context=None*)

If the two operands are unequal, return the number closest to the first operand in the direction of the second operand. If both operands are numerically equal, return a copy of the first operand with the sign set to be the same as the sign of the second operand.

normalize(*context=None*)

Used for producing canonical values of an equivalence class within either the current context or the specified context.

This has the same semantics as the unary plus operation, except that if the final result is finite it is reduced to its simplest form, with all trailing zeros removed and its sign preserved. That is, while the coefficient is non-zero and a multiple of ten the coefficient is divided by ten and the exponent is incremented by 1. Otherwise (the coefficient is zero) the exponent is set to 0. In all cases the sign is unchanged.

For example, `Decimal('32.100')` and `Decimal('0.321000e+2')` both normalize to the equivalent value `Decimal('32.1')`.

Note that rounding is applied *before* reducing to simplest form.

In the latest versions of the specification, this operation is also known as `reduce`.

number_class(*context=None*)

Return a string describing the *class* of the operand. The returned value is one of the following ten strings.

- `"-Infinity"`, indicating that the operand is negative infinity.
- `"-Normal"`, indicating that the operand is a negative normal number.
- `"-Subnormal"`, indicating that the operand is negative and subnormal.
- `"-Zero"`, indicating that the operand is a negative zero.
- `"+Zero"`, indicating that the operand is a positive zero.
- `"+Subnormal"`, indicating that the operand is positive and subnormal.
- `"+Normal"`, indicating that the operand is a positive normal number.
- `"+Infinity"`, indicating that the operand is positive infinity.
- `"NaN"`, indicating that the operand is a quiet NaN (Not a Number).
- `"sNaN"`, indicating that the operand is a signaling NaN.

quantize(*exp*, *rounding=None*, *context=None*)

Return a value equal to the first operand after rounding and having the exponent of the second operand.

```
>>> Decimal('1.41421356').quantize(Decimal('1.000'))
Decimal('1.414')
```

Unlike other operations, if the length of the coefficient after the quantize operation would be greater than precision, then an **`InvalidOperation`** is signaled. This guarantees that, unless there is an error condition, the quantized exponent is always equal to that of the right-hand operand.

Also unlike other operations, quantize never signals Underflow, even if the result is subnormal and inexact.

If the exponent of the second operand is larger than that of the first then rounding may be necessary. In this case, the rounding mode is determined by the `rounding` argument if given, else by the given `context` argument; if neither argument is given the rounding mode of the current thread’s context is used.

An error is returned whenever the resulting exponent is greater than **`Emax`** or less than **`Etiny()`**.

radix()

Return `Decimal(10)`, the radix (base) in which the **`Decimal`** class does all its arithmetic. Included for compatibility with the specification.

remainder_near(*other*, *context=None*)

Return the remainder from dividing *self* by *other*. This differs from `self % other` in that the sign of the remainder is chosen so as to minimize its absolute value. More precisely, the return value is `self - n * other` where `n` is the integer nearest to the exact value of `self / other`, and if two integers are equally near then the even one is chosen.

If the result is zero then its sign will be the sign of *self*.

```
>>> Decimal(18).remainder_near(Decimal(10))
Decimal('-2')
>>> Decimal(25).remainder_near(Decimal(10))
Decimal('5')
>>> Decimal(35).remainder_near(Decimal(10))
Decimal('-5')
```

rotate(*other*, *context=None*)

Return the result of rotating the digits of the first operand by an amount specified by the second operand. The second operand must be an integer in the range -precision through precision. The absolute value of the second operand gives the number of places to rotate. If the second operand is positive then rotation is to the left; otherwise rotation is to the right. The coefficient of the first operand is padded on the left with zeros to length precision if necessary. The sign and exponent of the first operand are unchanged.

same_quantum(*other*, *context=None*)

Test whether self and other have the same exponent or whether both are `NaN`.

This operation is unaffected by context and is quiet: no flags are changed and no rounding is performed. As an exception, the C version may raise InvalidOperation if the second operand cannot be converted exactly.

scaleb(*other*, *context=None*)

Return the first operand with exponent adjusted by the second. Equivalently, return the first operand multiplied by `10**other`. The second operand must be an integer.

shift(*other*, *context=None*)

Return the result of shifting the digits of the first operand by an amount specified by the second operand. The second operand must be an integer in the range -precision through precision. The absolute value of the second operand gives the number of places to shift. If the second operand is positive then the shift is to the left; otherwise the shift is to the right. Digits shifted into the coefficient are zeros. The sign and exponent of the first operand are unchanged.

sqrt(*context=None*)

Return the square root of the argument to full precision.

to_eng_string(*context=None*)

Convert to a string, using engineering notation if an exponent is needed.

Engineering notation has an exponent which is a multiple of 3. This can leave up to 3 digits to the left of the decimal place and may require the addition of either one or two trailing zeros.

For example, this converts `Decimal('123E+1')` to `Decimal('1.23E+3')`.

to_integral(*rounding=None*, *context=None*)

Identical to the **`to_integral_value()`** method. The `to_integral` name has been kept for compatibility with older versions.

to_integral_exact(*rounding=None*, *context=None*)

Round to the nearest integer, signaling **`Inexact`** or **`Rounded`** as appropriate if rounding occurs. The rounding mode is determined by the `rounding` parameter if given, else by the given `context`. If neither parameter is given then the rounding mode of the current context is used.

to_integral_value(*rounding=None*, *context=None*)

Round to the nearest integer without signaling **`Inexact`** or **`Rounded`**. If given, applies *rounding*; otherwise, uses the rounding method in either the supplied *context* or the current context.

Decimal numbers can be rounded using the **`round()`** function:

round(number)

round(number, ndigits)

If *ndigits* is not given or `None`, returns the nearest **`int`** to *number*, rounding ties to even, and ignoring the rounding mode of the **`Decimal`** context. Raises **`OverflowError`** if *number* is an infinity or **`ValueError`** if it is a (quiet or signaling) NaN.

If *ndigits* is an **`int`**, the context’s rounding mode is respected and a **`Decimal`** representing *number* rounded to the nearest multiple of `Decimal('1E-ndigits')` is returned; in this case, `round(number, ndigits)` is equivalent to `self.quantize(Decimal('1E-ndigits'))`. Returns `Decimal('NaN')` if *number* is a quiet NaN. Raises **`InvalidOperation`** if *number* is an infinity, a signaling NaN, or if the length of the coefficient after the quantize operation would be greater than the current context’s precision. In other words, for the non-corner cases:

- if *ndigits* is positive, return *number* rounded to *ndigits* decimal places;
- if *ndigits* is zero, return *number* rounded to the nearest integer;
- if *ndigits* is negative, return *number* rounded to the nearest multiple of `10**abs(ndigits)`.

For example:

```
>>> from decimal import Decimal, getcontext, ROUND_DOWN
>>> getcontext().rounding = ROUND_DOWN
>>> round(Decimal('3.75'))     # context rounding ignored
4
>>> round(Decimal('3.5'))      # round-ties-to-even
4
>>> round(Decimal('3.75'), 0)  # uses the context rounding
Decimal('3')
>>> round(Decimal('3.75'), 1)
Decimal('3.7')
>>> round(Decimal('3.75'), -1)
Decimal('0E+1')
```

Logical operands

The **`logical_and()`**, **`logical_invert()`**, **`logical_or()`**, and **`logical_xor()`** methods expect their arguments to be *logical operands*. A *logical operand* is a **`Decimal`** instance whose exponent and sign are both zero, and whose digits are all either `0` or `1`.

Context objects

Contexts are environments for arithmetic operations. They govern precision, set rules for rounding, determine which signals are treated as exceptions, and limit the range for exponents.

Each thread has its own current context which is accessed or changed using the **`getcontext()`** and **`setcontext()`** functions:

decimal.getcontext()

Return the current context for the active thread.

decimal.setcontext(*c*, */*)

Set the current context for the active thread to *c*.

You can also use the `with` statement and the `localcontext()` function to temporarily change the active context.

decimal.localcontext(*ctx=None*, ***kwargs*)

Return a context manager that will set the current context for the active thread to a copy of *ctx* on entry to the with-statement and restore the previous context when exiting the with-statement. If no context is specified, a copy of the current context is used. The *kwargs* argument is used to set the attributes of the new context.

For example, the following code sets the current decimal precision to 42 places, performs a calculation, and then automatically restores the previous context:

```
from decimal import localcontext

with localcontext() as ctx:
    ctx.prec = 42   # Perform a high precision calculation
    s = calculate_something()
s = +s  # Round the final result back to the default precision
```

Using keyword arguments, the code would be the following:

```
from decimal import localcontext

with localcontext(prec=42) as ctx:
    s = calculate_something()
s = +s
```

Raises `TypeError` if *kwargs* supplies an attribute that `Context` doesn't support. Raises either `TypeError` or `ValueError` if *kwargs* supplies an invalid value for an attribute.

Changed in version 3.11: `localcontext()` now supports setting context attributes through the use of keyword arguments.

decimal.IEEEContext(*bits*)

Return a context object initialized to the proper values for one of the IEEE interchange formats. The argument must be a multiple of 32 and less than `IEEE_CONTEXT_MAX_BITS`.

Added in version 3.14.

New contexts can also be created using the `Context` constructor described below. In addition, the module provides three pre-made contexts:

decimal.BasicContext

This is a standard context defined by the General Decimal Arithmetic Specification. Precision is set to nine. Rounding is set to `ROUND_HALF_UP`. All flags are cleared. All traps are enabled (treated as exceptions) except `Inexact`, `Rounded`, and `Subnormal`.

Because many of the traps are enabled, this context is useful for debugging.

decimal.ExtendedContext

This is a standard context defined by the General Decimal Arithmetic Specification. Precision is set to nine. Rounding is set to `ROUND_HALF_EVEN`. All flags are cleared. No traps are enabled (so that exceptions are not raised during computations).

Because the traps are disabled, this context is useful for applications that prefer to have result value of `NaN` or `Infinity` instead of raising exceptions. This allows an application to complete a run in the presence of conditions that would otherwise halt the program.

decimal.DefaultContext

This context is used by the `Context` constructor as a prototype for new contexts. Changing a field (such a precision) has the effect of changing the default for new contexts created by the `Context` constructor.

This context is most useful in multi-threaded environments. Changing one of the fields before threads are started has the effect of setting system-wide defaults. Changing the fields after threads have started is not recommended as it would require thread synchronization to prevent race conditions.

In single threaded environments, it is preferable to not use this context at all. Instead, simply create contexts explicitly as described below.

The default values are `Context.prec`=`28`, `Context.rounding`=`ROUND_HALF_EVEN`, and enabled traps for `Overflow`, `InvalidOperation`, and `DivisionByZero`.

In addition to the three supplied contexts, new contexts can be created with the `Context` constructor.

class decimal.Context(*prec=None*, *rounding=None*, *Emin=None*, *Emax=None*, *capitals=None*, *clamp=None*, *flags=None*, *traps=None*)

Creates a new context. If a field is not specified or is `None`, the default values are copied from the `DefaultContext`. If the *flags* field is not specified or is `None`, all flags are cleared.

prec

An integer in the range [`1`, `MAX_PREC`] that sets the precision for arithmetic operations in the context.

rounding

One of the constants listed in the section Rounding Modes.

traps
flags

Lists of any signals to be set. Generally, new contexts should only set traps and leave the flags clear.

Emin
Emax

Integers specifying the outer limits allowable for exponents. *Emin* must be in the range [`MIN_EMIN`, `0`], *Emax* in the range [`0`, `MAX_EMAX`].

capitals

Either `0` or `1` (the default). If set to `1`, exponents are printed with a capital `E`; otherwise, a lowercase `e` is used: `Decimal('6.02e+23')`.

clamp

Either `0` (the default) or `1`. If set to `1`, the exponent `e` of a `Decimal` instance representable in this context is strictly limited to the range `Emin - prec + 1 <= e <= Emax - prec + 1`. If *clamp* is `0` then a weaker condition holds: the adjusted exponent of the `Decimal` instance is at most `Emax`. When *clamp* is `1`, a large normal number will, where possible, have its exponent reduced and a corresponding number of zeros added to its coefficient, in order to fit the exponent constraints; this preserves the value of the number but loses information about significant trailing zeros. For example:

```
>>> Context(prec=6, Emax=999,
clamp=1).create_decimal('1.23e999')
Decimal('1.23000E+999')
```

A *clamp* value of `1` allows compatibility with the fixed-width decimal interchange formats specified in IEEE 754.

The `Context` class defines several general purpose methods as well as a large number of methods for doing arithmetic directly in a given context. In addition, for each of the `Decimal` methods described above (with the exception of the `adjusted()` and `as_tuple()` methods) there is a corresponding `Context` method. For example, for a `Context` instance `C` and `Decimal` instance `x`, `C.exp(x)` is equivalent to `x.exp(context=C)`. Each `Context` method accepts a Python integer (an instance of `int`) anywhere that a Decimal instance is accepted.

clear_flags()

Resets all of the flags to `0`.

clear_traps()

Resets all of the traps to `0`.

Added in version 3.3.

copy()

Return a duplicate of the context.

copy_decimal(*num*, /)

Return a copy of the Decimal instance num.

create_decimal(*num='0'*, /)

Creates a new Decimal instance from *num* but using *self* as context. Unlike the `Decimal` constructor, the context precision, rounding method, flags, and traps are applied to the conversion.

This is useful because constants are often given to a greater precision than is needed by the application. Another benefit is that rounding immediately eliminates unintended effects from digits beyond the current precision. In the following example, using unrounded inputs means that adding zero to a sum can change the result:

```
>>> getcontext().prec = 3
>>> Decimal('3.4445') + Decimal('1.0023')
Decimal('4.45')
>>> Decimal('3.4445') + Decimal(0) + Decimal('1.0023')
Decimal('4.44')
```

This method implements the to-number operation of the IBM specification. If the argument is a string, no leading or trailing whitespace or underscores are permitted.

create_decimal_from_float(*f*, /)

Creates a new Decimal instance from a float *f* but rounding using *self* as the context. Unlike the `Decimal.from_float()`

class method, the context precision, rounding method, flags, and traps are applied to the conversion.

```
>>> context = Context(prec=5, rounding=ROUND_DOWN)
>>> context.create_decimal_from_float(math.pi)
Decimal('3.1415')
>>> context = Context(prec=5, traps=[Inexact])
>>> context.create_decimal_from_float(math.pi)
Traceback (most recent call last):
    ...
decimal.Inexact: None
```

Added in version 3.1.

Etiny()

Returns a value equal to `Emin - prec + 1` which is the minimum exponent value for subnormal results. When underflow occurs, the exponent is set to **`Etiny`**.

Etop()

Returns a value equal to `Emax - prec + 1`.

The usual approach to working with decimals is to create **`Decimal`** instances and then apply arithmetic operations which take place within the current context for the active thread. An alternative approach is to use context methods for calculating within a specific context. The methods are similar to those for the **`Decimal`** class and are only briefly recounted here.

abs(*x*, /)

Returns the absolute value of *x*.

add(*x*, *y*, /)

Return the sum of *x* and *y*.

canonical(*x*, /)

Returns the same Decimal object *x*.

compare(*x*, *y*, /)

Compares *x* and *y* numerically.

compare_signal(*x*, *y*, /)

Compares the values of the two operands numerically.

compare_total(*x*, *y*, /)

Compares two operands using their abstract representation.

compare_total_mag(*x*, *y*, /)

Compares two operands using their abstract representation, ignoring sign.

copy_abs(*x*, /)

Returns a copy of *x* with the sign set to 0.

copy_negate(*x*, /)

Returns a copy of *x* with the sign inverted.

copy_sign(*x*, *y*, /)

Copies the sign from *y* to *x*.

divide(*x*, *y*, /)

Return *x* divided by *y*.

divide_int(*x*, *y*, /)

Return *x* divided by *y*, truncated to an integer.

divmod(*x*, *y*, /)

Divides two numbers and returns the integer part of the result.

exp(*x*, /)

Returns `e ** x`.

fma(*x*, *y*, *z*, /)

Returns *x* multiplied by *y*, plus *z*.

is_canonical(*x*, /)

Returns `True` if *x* is canonical; otherwise returns `False`.

is_finite(*x*, /)

Returns `True` if *x* is finite; otherwise returns `False`.

is_infinite(*x*, /)

Returns `True` if *x* is infinite; otherwise returns `False`.

is_nan(*x*, /)

Returns `True` if *x* is a qNaN or sNaN; otherwise returns `False`.

is_normal(*x*, /)

Returns `True` if *x* is a normal number; otherwise returns `False`.

is_qnan(*x*, /)

Returns `True` if *x* is a quiet NaN; otherwise returns `False`.

is_signed(*x*, /)

Returns `True` if *x* is negative; otherwise returns `False`.

is_snan(*x*, /)

Returns `True` if *x* is a signaling NaN; otherwise returns `False`.

is_subnormal(*x*, /)

Returns `True` if *x* is subnormal; otherwise returns `False`.

is_zero(*x*, /)

Returns `True` if *x* is a zero; otherwise returns `False`.

ln(*x*, /)

Returns the natural (base e) logarithm of *x*.

log10(*x*, /)

Returns the base 10 logarithm of *x*.

logb(*x*, /)

Returns the exponent of the magnitude of the operand’s MSD.

logical_and(*x*, *y*, /)

Applies the logical operation *and* between each operand’s digits.

logical_invert(*x*, /)

Invert all the digits in *x*.

logical_or(*x*, *y*, /)

Applies the logical operation *or* between each operand’s digits.

logical_xor(*x*, *y*, /)

Applies the logical operation *xor* between each operand’s digits.

max(*x*, *y*, /)

Compares two values numerically and returns the maximum.

max_mag(*x*, *y*, /)

Compares the values numerically with their sign ignored.

min(*x*, *y*, /)

Compares two values numerically and returns the minimum.

min_mag(*x*, *y*, /)

Compares the values numerically with their sign ignored.

minus(*x*, /)

Minus corresponds to the unary prefix minus operator in Python.

multiply(*x*, *y*, /)

Return the product of *x* and *y*.

next_minus(*x*, /)

Returns the largest representable number smaller than *x*.

next_plus(*x*, /)

Returns the smallest representable number larger than *x*.

next_toward(*x*, *y*, /)

Returns the number closest to *x*, in direction towards *y*.

normalize(*x*, /)

Reduces *x* to its simplest form.

number_class(*x*, /)

Returns an indication of the class of *x*.

plus(*x*, /)

Plus corresponds to the unary prefix plus operator in Python. This operation applies the context precision and rounding, so it is *not* an identity operation.

power(*x*, *y*, *modulo=None*)

Return `x` to the power of `y`, reduced modulo `modulo` if given.

With two arguments, compute `x**y`. If `x` is negative then `y` must be integral. The result will be inexact unless `y` is integral and the result is finite and can be expressed exactly in ‘precision’ digits. The rounding mode of the context is used. Results are always correctly rounded in the Python version.

`Decimal(0) ** Decimal(0)` results in `InvalidOperation`, and if `InvalidOperation` is not trapped, then results in `Decimal('NaN')`.

Changed in version 3.3: The C module computes `power()` in terms of the correctly rounded `exp()` and `ln()` functions. The result is well-defined but only “almost always correctly rounded”.

With three arguments, compute `(x**y) % modulo`. For the three argument form, the following restrictions on the arguments hold:

- all three arguments must be integral
- `y` must be nonnegative
- at least one of `x` or `y` must be nonzero
- `modulo` must be nonzero and have at most 'precision' digits

The value resulting from `Context.power(x, y, modulo)` is equal to the value that would be obtained by computing `(x**y) % modulo` with unbounded precision, but is computed more efficiently. The exponent of the result is zero, regardless of the exponents of `x`, `y` and `modulo`. The result is always exact.

quantize(*x*, *y*, */*)

Returns a value equal to *x* (rounded), having the exponent of *y*.

radix()

Just returns 10, as this is Decimal, :)

remainder(*x*, *y*, */*)

Returns the remainder from integer division.

The sign of the result, if non-zero, is the same as that of the original dividend.

remainder_near(*x*, *y*, */*)

Returns `x - y * n`, where *n* is the integer nearest the exact value of `x / y` (if the result is 0 then its sign will be the sign of *x*).

rotate(*x*, *y*, */*)

Returns a rotated copy of *x*, *y* times.

same_quantum(*x*, *y*, */*)

Returns `True` if the two operands have the same exponent.

scaleb(*x*, *y*, /)

Returns the first operand after adding the second value its exp.

shift(*x*, *y*, /)

Returns a shifted copy of *x*, *y* times.

sqrt(*x*, /)

Square root of a non-negative number to context precision.

subtract(*x*, *y*, /)

Return the difference between *x* and *y*.

to_eng_string(*x*, /)

Convert to a string, using engineering notation if an exponent is needed.

Engineering notation has an exponent which is a multiple of 3. This can leave up to 3 digits to the left of the decimal place and may require the addition of either one or two trailing zeros.

to_integral_exact(*x*, /)

Rounds to an integer.

to_sci_string(*x*, /)

Converts a number to a string using scientific notation.

Constants

The constants in this section are only relevant for the C module. They are also included in the pure Python version for compatibility.

	32-bit	**64-bit**
decimal.MAX_PREC	`425000000`	`999999999999999999`
decimal.MAX_EMAX	`425000000`	`999999999999999999`
decimal.MIN_EMIN	`-425000000`	`-999999999999999999`
decimal.MIN_ETINY	`-849999999`	`-1999999999999999997`
decimal.IEEE_CONTEXT_MAX_BITS	`256`	`512`

decimal.HAVE_THREADS

The value is `True`. Deprecated, because Python now always has threads.

Deprecated since version 3.9.

decimal.HAVE_CONTEXTVAR

The default value is `True`. If Python is [`configured using the --without-decimal-contextvar option`], the C version uses a thread-local rather than a coroutine-local context and the value is `False`. This is slightly faster in some nested context scenarios.

Added in version 3.8.3.

Rounding modes

decimal.ROUND_CEILING

Round towards `Infinity`.

decimal.ROUND_DOWN

Round towards zero.

decimal.ROUND_FLOOR
Round towards `-Infinity`.

decimal.ROUND_HALF_DOWN
Round to nearest with ties going towards zero.

decimal.ROUND_HALF_EVEN
Round to nearest with ties going to nearest even integer.

decimal.ROUND_HALF_UP
Round to nearest with ties going away from zero.

decimal.ROUND_UP
Round away from zero.

decimal.ROUND_05UP
Round away from zero if last digit after rounding towards zero would have been 0 or 5; otherwise round towards zero.

Signals

Signals represent conditions that arise during computation. Each corresponds to one context flag and one context trap enabler.

The context flag is set whenever the condition is encountered. After the computation, flags may be checked for informational purposes (for instance, to determine whether a computation was exact). After checking the flags, be sure to clear all flags before starting the next computation.

If the context's trap enabler is set for the signal, then the condition causes a Python exception to be raised. For example, if the `DivisionByZero` trap is set, then a `DivisionByZero` exception is raised upon encountering the condition.

class decimal.Clamped
Altered an exponent to fit representation constraints.

Typically, clamping occurs when an exponent falls outside the context's `Emin` and `Emax` limits. If possible, the exponent is reduced to fit by adding zeros to the coefficient.

class decimal.DecimalException

Base class for other signals and a subclass of `ArithmeticError`.

class decimal.DivisionByZero

Signals the division of a non-infinite number by zero.

Can occur with division, modulo division, or when raising a number to a negative power. If this signal is not trapped, returns `Infinity` or `-Infinity` with the sign determined by the inputs to the calculation.

class decimal.Inexact

Indicates that rounding occurred and the result is not exact.

Signals when non-zero digits were discarded during rounding. The rounded result is returned. The signal flag or trap is used to detect when results are inexact.

class decimal.InvalidOperation

An invalid operation was performed.

Indicates that an operation was requested that does not make sense. If not trapped, returns `NaN`. Possible causes include:

```
Infinity - Infinity
0 * Infinity
Infinity / Infinity
x % 0
Infinity % x
sqrt(-x) and x > 0
0 ** 0
x ** (non-integer)
x ** Infinity
```

class decimal.Overflow

Numerical overflow.

Indicates the exponent is larger than `Context.Emax` after rounding has occurred. If not trapped, the result depends on the rounding mode, either pulling inward to the largest representable finite number or rounding outward to `Infinity`. In either case, `Inexact` and `Rounded` are also signaled.

class decimal.Rounded

Rounding occurred though possibly no information was lost.

Signaled whenever rounding discards digits; even if those digits are zero (such as rounding `5.00` to `5.0`). If not trapped, returns the result unchanged. This signal is used to detect loss of significant digits.

class decimal.Subnormal

Exponent was lower than `Emin` prior to rounding.

Occurs when an operation result is subnormal (the exponent is too small). If not trapped, returns the result unchanged.

class decimal.Underflow

Numerical underflow with result rounded to zero.

Occurs when a subnormal result is pushed to zero by rounding. `Inexact` and `Subnormal` are also signaled.

class decimal.FloatOperation

Enable stricter semantics for mixing floats and Decimals.

If the signal is not trapped (default), mixing floats and Decimals is permitted in the `Decimal` constructor, `create_decimal()` and all comparison operators. Both conversion and comparisons are exact. Any occurrence of a mixed operation is silently recorded by setting `FloatOperation` in the context flags. Explicit conversions with `from_float()` or `create_decimal_from_float()` do not set the flag.

Otherwise (the signal is trapped), only equality comparisons and explicit conversions are silent. All other mixed operations raise `FloatOperation`.

The following table summarizes the hierarchy of signals:

```
exceptions.ArithmeticError(exceptions.Exception)
    DecimalException
        Clamped
        DivisionByZero(DecimalException, 
exceptions.ZeroDivisionError)
        Inexact
            Overflow(Inexact, Rounded)
            Underflow(Inexact, Rounded, Subnormal)
        InvalidOperation
        Rounded
        Subnormal
        FloatOperation(DecimalException, exceptions.TypeError)
```

Floating-point notes

Mitigating round-off error with increased precision

The use of decimal floating point eliminates decimal representation error (making it possible to represent `0.1` exactly); however, some operations can still incur round-off error when non-zero digits exceed the fixed precision.

The effects of round-off error can be amplified by the addition or subtraction of nearly offsetting quantities resulting in loss of significance. Knuth provides two instructive examples where rounded floating-point arithmetic with insufficient precision causes the breakdown of the associative and distributive properties of addition:

```
# Examples from Seminumerical Algorithms, Section 4.2.2.
>>> from decimal import Decimal, getcontext
>>> getcontext().prec = 8

>>> u, v, w = Decimal(11111113), Decimal(-11111111), 
Decimal('7.51111111')
>>> (u + v) + w
Decimal('9.5111111')
>>> u + (v + w)
Decimal('10')

>>> u, v, w = Decimal(20000), Decimal(-6), Decimal('6.0000003')
>>> (u*v) + (u*w)
Decimal('0.01')
>>> u * (v+w)
Decimal('0.0060000')
```

The **decimal** module makes it possible to restore the identities by expanding the precision sufficiently to avoid loss of significance:

```
>>> getcontext().prec = 20
>>> u, v, w = Decimal(11111113), Decimal(-11111111),
Decimal('7.51111111')
>>> (u + v) + w
Decimal('9.51111111')
>>> u + (v + w)
Decimal('9.51111111')
>>>
>>> u, v, w = Decimal(20000), Decimal(-6), Decimal('6.0000003')
>>> (u*v) + (u*w)
Decimal('0.0060000')
>>> u * (v+w)
Decimal('0.0060000')
```

Special values

The number system for the **decimal** module provides special values including `NaN`, `sNaN`, `-Infinity`, `Infinity`, and two zeros, `+0` and `-0`.

Infinities can be constructed directly with: `Decimal('Infinity')`. Also, they can arise from dividing by zero when the **DivisionByZero** signal is not trapped. Likewise, when the **Overflow** signal is not trapped, infinity can result from rounding beyond the limits of the largest representable number.

The infinities are signed (affine) and can be used in arithmetic operations where they get treated as very large, indeterminate numbers. For instance, adding a constant to infinity gives another infinite result.

Some operations are indeterminate and return `NaN`, or if the **InvalidOperation** signal is trapped, raise an exception. For example, `0/0` returns `NaN` which means “not a number”. This variety of `NaN` is quiet and, once created, will flow through other computations always resulting in another `NaN`. This behavior can be useful for a series of computations that occasionally have missing inputs — it allows the calculation to proceed while flagging specific results as invalid.

A variant is `sNaN` which signals rather than remaining quiet after every operation. This is a useful return value when an invalid result needs to interrupt a calculation for special handling.

The behavior of Python’s comparison operators can be a little surprising where a `NaN` is involved. A test for equality where one of the operands is a quiet or signaling `NaN` always returns **`False`** (even when doing `Decimal('NaN')==Decimal('NaN')`), while a test for inequality always returns **`True`**. An attempt to compare two Decimals using any of the `<`, `<=`, `>` or `>=` operators will raise the **`InvalidOperation`** signal if either operand is a `NaN`, and return **`False`** if this signal is not trapped. Note that the General Decimal Arithmetic specification does not specify the behavior of direct comparisons; these rules for comparisons involving a `NaN` were taken from the IEEE 854 standard (see Table 3 in section 5.7). To ensure strict standards-compliance, use the **`compare()`** and **`compare_signal()`** methods instead.

The signed zeros can result from calculations that underflow. They keep the sign that would have resulted if the calculation had been carried out to greater precision. Since their magnitude is zero, both positive and negative zeros are treated as equal and their sign is informational.

In addition to the two signed zeros which are distinct yet equal, there are various representations of zero with differing precisions yet equivalent in value. This takes a bit of getting used to. For an eye accustomed to normalized floating-point representations, it is not immediately obvious that the following calculation returns a value equal to zero:

```
>>> 1 / Decimal('Infinity')
Decimal('0E-1000026')
```

Working with threads

The `getcontext()` function accesses a different `Context` object for each thread. Having separate thread contexts means that threads may make changes (such as `getcontext().prec=10`) without interfering with other threads.

Likewise, the `setcontext()` function automatically assigns its target to the current thread.

If `setcontext()` has not been called before `getcontext()`, then `getcontext()` will automatically create a new context for use in the current thread. New context objects have default values set from the `decimal.DefaultContext` object.

The `sys.flags.thread_inherit_context` flag affects the context for new threads. If the flag is false, new threads will start with an empty context. In this case, `getcontext()` will create a new context object when called and use the default values from *DefaultContext*. If the flag is true, new threads will start with a copy of context from the caller of `threading.Thread.start()`.

To control the defaults so that each thread will use the same values throughout the application, directly modify the *DefaultContext* object. This should be done *before* any threads are started so that there won't be a race condition between threads calling `getcontext()`. For example:

```
# Set applicationwide defaults for all threads about to be launched
DefaultContext.prec = 12
DefaultContext.rounding = ROUND_DOWN
DefaultContext.traps = ExtendedContext.traps.copy()
DefaultContext.traps[InvalidOperation] = 1
setcontext(DefaultContext)

# Afterwards, the threads can be started
t1.start()
t2.start()
t3.start()
 . . .
```

Recipes

Here are a few recipes that serve as utility functions and that demonstrate ways to work with the `Decimal` class:

```
def moneyfmt(value, places=2, curr='', sep=',', dp='.',
             pos='', neg='-', trailneg=''):
    """Convert Decimal to a money formatted string.

    places:  required number of places after the decimal point
    curr:    optional currency symbol before the sign (may be blank)
    sep:     optional grouping separator (comma, period, space, or
blank)
    dp:      decimal point indicator (comma or period)
             only specify as blank when places is zero
    pos:     optional sign for positive numbers: '+', space or blank
    neg:     optional sign for negative numbers: '-', '(', space or
blank
    trailneg:optional trailing minus indicator:  '-', ')', space or
blank

    >>> d = Decimal('-1234567.8901')
    >>> moneyfmt(d, curr='$')
    '-$1,234,567.89'
    >>> moneyfmt(d, places=0, sep='.', dp='', neg='', trailneg='-')
    '1.234.568-'
    >>> moneyfmt(d, curr='$', neg='(', trailneg=')')
    '($1,234,567.89)'
    >>> moneyfmt(Decimal(123456789), sep=' ')
    '123 456 789.00'
    >>> moneyfmt(Decimal('-0.02'), neg='<', trailneg='>')
    '<0.02>'

    """
    q = Decimal(10) ** -places      # 2 places --> '0.01'
    sign, digits, exp = value.quantize(q).as_tuple()
    result = []
    digits = list(map(str, digits))
    build, next = result.append, digits.pop
    if sign:
        build(trailneg)
    for i in range(places):
        build(next() if digits else '0')
    if places:
        build(dp)
    if not digits:
        build('0')
    i = 0
    while digits:
        build(next())
        i += 1
        if i == 3 and digits:
            i = 0
            build(sep)
    build(curr)
    build(neg if sign else pos)
    return ''.join(reversed(result))

def pi():
    """Compute Pi to the current precision.

    >>> print(pi())
    3.141592653589793238462643383

    """
    getcontext().prec += 2  # extra digits for intermediate steps
    three = Decimal(3)      # substitute "three=3.0" for regular
floats
    lasts, t, s, n, na, d, da = 0, three, 3, 1, 0, 0, 24
    while s != lasts:
        lasts = s
        n, na = n+na, na+8
        d, da = d+da, da+32
        t = (t * n) / d
        s += t
    getcontext().prec -= 2
    return +s               # unary plus applies the new precision

def exp(x):
    """Return e raised to the power of x.  Result type matches input
type.

    >>> print(exp(Decimal(1)))
    2.718281828459045235360287471
    >>> print(exp(Decimal(2)))
    7.389056098930650227230427461
    >>> print(exp(2.0))
    7.38905609893
    >>> print(exp(2+0j))
    (7.38905609893+0j)
    """
    getcontext().prec += 2
    i, lasts, s, fact, num = 0, 0, 1, 1, 1
    while s != lasts:
        lasts = s
        i += 1
        fact *= i
        num *= x
        s += num / fact
    getcontext().prec -= 2
    return +s

def cos(x):
    """Return the cosine of x as measured in radians.

    The Taylor series approximation works best for a small value of
x.
    For larger values, first compute x = x % (2 * pi).

    >>> print(cos(Decimal('0.5')))
    0.8775825618903727161162815826
    >>> print(cos(0.5))
    0.87758256189
    >>> print(cos(0.5+0j))
    (0.87758256189+0j)

    """
    getcontext().prec += 2
    i, lasts, s, fact, num, sign = 0, 0, 1, 1, 1, 1
    while s != lasts:
        lasts = s
        i += 2
        fact *= i * (i-1)
        num *= x * x
        sign *= -1
        s += num / fact * sign
    getcontext().prec -= 2
    return +s

def sin(x):
    """Return the sine of x as measured in radians.

    The Taylor series approximation works best for a small value of
x.
    For larger values, first compute x = x % (2 * pi).
    >>> print(sin(Decimal('0.5')))
    0.4794255386042030002732879352
    >>> print(sin(0.5))
    0.479425538604
    >>> print(sin(0.5+0j))
    (0.479425538604+0j)

    """
    getcontext().prec += 2
    i, lasts, s, fact, num, sign = 1, 0, x, 1, x, 1
    while s != lasts:
        lasts = s
        i += 2
        fact *= i * (i-1)
        num *= x * x
        sign *= -1
        s += num / fact * sign
    getcontext().prec -= 2
    return +s
```

Decimal FAQ

Q: It is cumbersome to type `decimal.Decimal('1234.5')`. Is there a way to minimize typing when using the interactive interpreter?

A: Some users abbreviate the constructor to just a single letter:

```
>>> D = decimal.Decimal
>>> D('1.23') + D('3.45')
Decimal('4.68')
```

Q: In a fixed-point application with two decimal places, some inputs have many places and need to be rounded. Others are not supposed to have excess digits and need to be validated. What methods should be used?

A: The **`quantize()`** method rounds to a fixed number of decimal places. If the **`Inexact`** trap is set, it is also useful for validation:

```
>>> TWOPLACES = Decimal(10) ** -2       # same as Decimal('0.01')

>>> # Round to two places
>>> Decimal('3.214').quantize(TWOPLACES)
Decimal('3.21')
```

```
>>> # Validate that a number does not exceed two places
>>> Decimal('3.21').quantize(TWOPLACES, context=Context(traps=
[Inexact]))
Decimal('3.21')

>>> Decimal('3.214').quantize(TWOPLACES, context=Context(traps=
[Inexact]))
Traceback (most recent call last):
   ...
Inexact: None
```

Q: Once I have valid two place inputs, how do I maintain that invariant throughout an application?

A: Some operations like addition, subtraction, and multiplication by an integer will automatically preserve fixed point. Others operations, like division and non-integer multiplication, will change the number of decimal places and need to be followed-up with a `quantize()` step:

```
>>> a = Decimal('102.72')           # Initial fixed-point values
>>> b = Decimal('3.17')
>>> a + b                           # Addition preserves fixed-point
Decimal('105.89')
>>> a - b
Decimal('99.55')
>>> a * 42                          # So does integer multiplication
Decimal('4314.24')
>>> (a * b).quantize(TWOPLACES)     # Must quantize non-integer
multiplication
Decimal('325.62')
>>> (b / a).quantize(TWOPLACES)     # And quantize division
Decimal('0.03')
```

In developing fixed-point applications, it is convenient to define functions to handle the `quantize()` step:

```
>>> def mul(x, y, fp=TWOPLACES):
...     return (x * y).quantize(fp)
...
>>> def div(x, y, fp=TWOPLACES):
...     return (x / y).quantize(fp)

>>> mul(a, b)                       # Automatically preserve fixed-
point
```

```
Decimal('325.62')
>>> div(b, a)
Decimal('0.03')
```

Q: There are many ways to express the same value. The numbers `200`, `200.000`, `2E2`, and `.02E+4` all have the same value at various precisions. Is there a way to transform them to a single recognizable canonical value?

A: The **`normalize()`** method maps all equivalent values to a single representative:

```
>>> values = map(Decimal, '200 200.000 2E2 .02E+4'.split())
>>> [v.normalize() for v in values]
[Decimal('2E+2'), Decimal('2E+2'), Decimal('2E+2'), Decimal('2E+2')]
```

Q: When does rounding occur in a computation?

A: It occurs *after* the computation. The philosophy of the decimal specification is that numbers are considered exact and are created independent of the current context. They can even have greater precision than current context. Computations process with those exact inputs and then rounding (or other context operations) is applied to the *result* of the computation:

```
>>> getcontext().prec = 5
>>> pi = Decimal('3.1415926535')   # More than 5 digits
>>> pi                             # All digits are retained
Decimal('3.1415926535')
>>> pi + 0                         # Rounded after an addition
Decimal('3.1416')
>>> pi - Decimal('0.00005')        # Subtract unrounded numbers,
then round
Decimal('3.1415')
>>> pi + 0 - Decimal('0.00005').   # Intermediate values are rounded
Decimal('3.1416')
```

Q: Some decimal values always print with exponential notation. Is there a way to get a non-exponential representation?

A: For some values, exponential notation is the only way to express the number of significant places in the coefficient. For example, expressing `5.0E+3` as `5000` keeps the value constant but cannot show the original’s two-place significance.

If an application does not care about tracking significance, it is easy to remove the exponent and trailing zeroes, losing significance, but keeping the value unchanged:

```
>>> def remove_exponent(d):
...     return d.quantize(Decimal(1)) if d == d.to_integral() else
d.normalize()

>>> remove_exponent(Decimal('5E+3'))
Decimal('5000')
```

Q: Is there a way to convert a regular float to a **`Decimal`**?

A: Yes, any binary floating-point number can be exactly expressed as a Decimal though an exact conversion may take more precision than intuition would suggest:

```
>>> Decimal(math.pi)
Decimal('3.141592653589793115997963468544185161590576171875')
```

Q: Within a complex calculation, how can I make sure that I haven’t gotten a spurious result because of insufficient precision or rounding anomalies.

A: The decimal module makes it easy to test results. A best practice is to re-run calculations using greater precision and with various rounding modes. Widely differing results indicate insufficient precision, rounding mode issues, ill-conditioned inputs, or a numerically unstable algorithm.

Q: I noticed that context precision is applied to the results of operations but not to the inputs. Is there anything to watch out for when mixing values of different precisions?

A: Yes. The principle is that all values are considered to be exact and so is the arithmetic on those values. Only the results are rounded. The advantage for inputs is that “what you type is what you get”. A disadvantage is that the results can look odd if you forget that the inputs haven’t been rounded:

```
>>> getcontext().prec = 3
>>> Decimal('3.104') + Decimal('2.104')
Decimal('5.21')
>>> Decimal('3.104') + Decimal('0.000') + Decimal('2.104')
Decimal('5.20')
```

The solution is either to increase precision or to force rounding of inputs using the unary plus operation:

```
>>> getcontext().prec = 3
>>> +Decimal('1.23456789')      # unary plus triggers rounding
Decimal('1.23')
```

Alternatively, inputs can be rounded upon creation using the **`Context.create_decimal()`** method:

```
>>> Context(prec=5, rounding=ROUND_DOWN).create_decimal('1.2345678')
Decimal('1.2345')
```

Q: Is the CPython implementation fast for large numbers?

A: Yes. In the CPython and PyPy3 implementations, the C/CFFI versions of the decimal module integrate the high speed libmpdec [https://www.bytereef.org/mpdecimal/doc/libmpdec/index.html] library for arbitrary precision correctly rounded decimal floating-point arithmetic [[1]]. `libmpdec` uses Karatsuba multiplication [https://en.wikipedia.org/wiki/Karatsuba_algorithm] for medium-sized numbers and the Number Theoretic Transform [https://en.wikipedia.org/wiki/Discrete_Fourier_transform_(general)#Number-theoretic_transform] for very large numbers.

The context must be adapted for exact arbitrary precision arithmetic. **`Emin`** and **`Emax`** should always be set to the maximum values, **`clamp`** should always be 0 (the default). Setting **`prec`** requires some care.

The easiest approach for trying out bignum arithmetic is to use the maximum value for **`prec`** as well [[2]]:

```
>>> setcontext(Context(prec=MAX_PREC, Emax=MAX_EMAX, Emin=MIN_EMIN))
>>> x = Decimal(2) ** 256
>>> x / 128
```

```
Decimal('904625697166532776746648320380374280103671755200316906558262
375061821325312')
```

For inexact results, **`MAX_PREC`** is far too large on 64-bit platforms and the available memory will be insufficient:

```
>>> Decimal(1) / 3
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
MemoryError
```

On systems with overallocation (e.g. Linux), a more sophisticated approach is to adjust **`prec`** to the amount of available RAM. Suppose that you have 8GB of RAM and expect 10 simultaneous operands using a maximum of 500MB each:

```
>>> import sys
>>>
>>> # Maximum number of digits for a single operand using 500MB in 8-
byte words
>>> # with 19 digits per word (4-byte and 9 digits for the 32-bit
build):
>>> maxdigits = 19 * ((500 * 1024**2) // 8)
>>>
>>> # Check that this works:
>>> c = Context(prec=maxdigits, Emax=MAX_EMAX, Emin=MIN_EMIN)
>>> c.traps[Inexact] = True
>>> setcontext(c)
>>>
>>> # Fill the available precision with nines:
>>> x = Decimal(0).logical_invert() * 9
>>> sys.getsizeof(x)
524288112
>>> x + 2
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  decimal.Inexact: [<class 'decimal.Inexact'>]
```

In general (and especially on systems without overallocation), it is recommended to estimate even tighter bounds and set the **`Inexact`** trap if all calculations are expected to be exact.

[1]

Added in version 3.3.
[2]

Changed in version 3.9: This approach now works for all exact results except for non-integer powers.

`fractions` — Rational numbers

Source code: Lib/fractions.py
[https://github.com/python/cpython/tree/3.14/Lib/fractions.py]

The **`fractions`** module provides support for rational number arithmetic.

A Fraction instance can be constructed from a pair of rational numbers, from a single number, or from a string.

class fractions.Fraction(*numerator=0*, *denominator=1*)
class fractions.Fraction(*number*)
class fractions.Fraction(*string*)

> The first version requires that *numerator* and *denominator* are instances of **`numbers.Rational`** and returns a new **`Fraction`** instance with a value equal to `numerator/denominator`. If *denominator* is zero, it raises a **`ZeroDivisionError`**.
>
> The second version requires that *number* is an instance of **`numbers.Rational`** or has the **`as_integer_ratio()`** method (this includes **`float`** and **`decimal.Decimal`**). It returns a **`Fraction`** instance with exactly the same value. Assumed, that the **`as_integer_ratio()`** method returns a pair of coprime integers and last one is positive. Note that due to the usual issues with binary point (see Floating-Point Arithmetic: Issues and Limitations), the argument to `Fraction(1.1)` is not exactly equal to 11/10, and so `Fraction(1.1)` does *not* return `Fraction(11, 10)` as one

might expect. (But see the documentation for the `limit_denominator()` method below.)

The last version of the constructor expects a string. The usual form for this instance is:

```
[sign] numerator ['/' denominator]
```

where the optional `sign` may be either ‘+’ or ‘-’ and `numerator` and `denominator` (if present) are strings of decimal digits (underscores may be used to delimit digits as with integral literals in code). In addition, any string that represents a finite value and is accepted by the **`float`** constructor is also accepted by the **`Fraction`** constructor. In either form the input string may also have leading and/or trailing whitespace. Here are some examples:

```
>>> from fractions import Fraction
>>> Fraction(16, -10)
Fraction(-8, 5)
>>> Fraction(123)
Fraction(123, 1)
>>> Fraction()
Fraction(0, 1)
>>> Fraction('3/7')
Fraction(3, 7)
>>> Fraction(' -3/7 ')
Fraction(-3, 7)
>>> Fraction('1.414213 \t\n')
Fraction(1414213, 1000000)
>>> Fraction('-.125')
Fraction(-1, 8)
>>> Fraction('7e-6')
Fraction(7, 1000000)
>>> Fraction(2.25)
Fraction(9, 4)
>>> Fraction(1.1)
Fraction(2476979795053773, 2251799813685248)
>>> from decimal import Decimal
>>> Fraction(Decimal('1.1'))
Fraction(11, 10)
```

The `Fraction` class inherits from the abstract base class `numbers.Rational`, and implements all of the methods and operations from that class. `Fraction` instances are hashable, and should be treated as immutable. In addition, `Fraction` has the following properties and methods:

Changed in version 3.2: The `Fraction` constructor now accepts `float` and `decimal.Decimal` instances.

Changed in version 3.9: The `math.gcd()` function is now used to normalize the *numerator* and *denominator*. `math.gcd()` always returns an `int` type. Previously, the GCD type depended on *numerator* and *denominator*.

Changed in version 3.11: Underscores are now permitted when creating a `Fraction` instance from a string, following **PEP 515** [https://peps.python.org/pep-0515/] rules.

Changed in version 3.11: `Fraction` implements `__int__` now to satisfy `typing.SupportsInt` instance checks.

Changed in version 3.12: Space is allowed around the slash for string inputs: `Fraction('2 / 3')`.

Changed in version 3.12: `Fraction` instances now support float-style formatting, with presentation types `"e"`, `"E"`, `"f"`, `"F"`, `"g"`, `"G"` and `"%""`.

Changed in version 3.13: Formatting of `Fraction` instances without a presentation type now supports fill, alignment, sign handling, minimum width and grouping.

Changed in version 3.14: The `Fraction` constructor now accepts any objects with the **`as_integer_ratio()`** method.

numerator

Numerator of the Fraction in lowest term.

denominator

Denominator of the Fraction in lowest terms. Guaranteed to be positive.

as_integer_ratio()

Return a tuple of two integers, whose ratio is equal to the original Fraction. The ratio is in lowest terms and has a positive denominator.

Added in version 3.8.

is_integer()

Return `True` if the Fraction is an integer.

Added in version 3.12.

classmethod from_float(*f*)

Alternative constructor which only accepts instances of `float` or `numbers.Integral`. Beware that `Fraction.from_float(0.3)` is not the same value as `Fraction(3, 10)`.

Note

From Python 3.2 onwards, you can also construct a `Fraction` instance directly from a `float`.

classmethod from_decimal(*dec*)

Alternative constructor which only accepts instances of `decimal.Decimal` or `numbers.Integral`.

Note

From Python 3.2 onwards, you can also construct a `Fraction` instance directly from a `decimal.Decimal` instance.

classmethod from_number(*number*)

Alternative constructor which only accepts instances of `numbers.Integral`, `numbers.Rational`, `float` or `decimal.Decimal`, and objects with the **`as_integer_ratio()`** method, but not strings.

Added in version 3.14.

limit_denominator(*max_denominator=1000000*)

Finds and returns the closest `Fraction` to `self` that has denominator at most max_denominator. This method is useful for finding rational approximations to a given floating-point number:

```
>>> from fractions import Fraction
>>>
Fraction('3.1415926535897932').limit_denominator(1000)
Fraction(355, 113)
```

or for recovering a rational number that’s represented as a float:

```
>>> from math import pi, cos
>>> Fraction(cos(pi/3))
Fraction(4503599627370497, 9007199254740992)
>>> Fraction(cos(pi/3)).limit_denominator()
Fraction(1, 2)
>>> Fraction(1.1).limit_denominator()
Fraction(11, 10)
```

__floor__()

Returns the greatest `int` `<= self`. This method can also be accessed through the `math.floor()` function:

```
>>> from math import floor
>>> floor(Fraction(355, 113))
3
```

__ceil__()

Returns the least **`int`** `>= self`. This method can also be accessed through the **`math.ceil()`** function.

__round__()
__round__(*ndigits*)

The first version returns the nearest **`int`** to `self`, rounding half to even. The second version rounds `self` to the nearest multiple of `Fraction(1, 10**ndigits)` (logically, if `ndigits` is negative), again rounding half toward even. This method can also be accessed through the **`round()`** function.

__format__(*format_spec*, */*)

Provides support for formatting of **`Fraction`** instances via the **`str.format()`** method, the **`format()`** built-in function, or Formatted string literals.

If the `format_spec` format specification string does not end with one of the presentation types `'e'`, `'E'`, `'f'`, `'F'`, `'g'`, `'G'` or `'%'` then formatting follows the general rules for fill, alignment, sign handling, minimum width, and grouping as described in the format specification mini-language. The “alternate form” flag `'#'` is supported: if present, it forces the output string to always include an explicit denominator, even when the value being formatted is an exact integer. The zero-fill flag `'0'` is not supported.

If the `format_spec` format specification string ends with one of the presentation types `'e'`, `'E'`, `'f'`, `'F'`, `'g'`, `'G'` or `'%'` then formatting follows the rules outlined for the **`float`** type in the Format Specification Mini-Language section.

Here are some examples:

```
>>> from fractions import Fraction
>>> format(Fraction(103993, 33102), '_')
'103_993/33_102'
>>> format(Fraction(1, 7), '.^+10')
'...+1/7...'
>>> format(Fraction(3, 1), '')
'3'
>>> format(Fraction(3, 1), '#')
'3/1'
>>> format(Fraction(1, 7), '.40g')
'0.1428571428571428571428571428571428571429'
>>> format(Fraction('1234567.855'), '_.2f')
'1_234_567.86'
>>> f"{Fraction(355, 113):*>20.6e}"
'********3.141593e+00'
>>> old_price, new_price = 499, 672
>>> "{:.2%} price increase".format(Fraction(new_price,
old_price) - 1)
'34.67% price increase'
```

See also

Module `numbers`

The abstract base classes making up the numeric tower.

`random` — Generate pseudo-random numbers

Source code: Lib/random.py
[https://github.com/python/cpython/tree/3.14/Lib/random.py]

This module implements pseudo-random number generators for various distributions.

For integers, there is uniform selection from a range. For sequences, there is uniform selection of a random element, a function to generate a random permutation of a list in-place, and a function for random sampling without replacement.

On the real line, there are functions to compute uniform, normal (Gaussian), lognormal, negative exponential, gamma, and beta distributions. For generating distributions of angles, the von Mises distribution is available.

Almost all module functions depend on the basic function `random()`, which generates a random float uniformly in the half-open range `0.0 <= X < 1.0`. Python uses the Mersenne Twister as the core generator. It produces 53-bit precision floats and has a period of 2**19937-1. The underlying implementation in C is both fast and threadsafe. The Mersenne Twister is one of the most extensively tested random number generators in existence. However, being completely deterministic, it is not suitable for all purposes, and is completely unsuitable for cryptographic purposes.

The functions supplied by this module are actually bound methods of a hidden instance of the `random.Random` class. You can instantiate your own instances of `Random` to get generators that don’t share state.

Class `Random` can also be subclassed if you want to use a different basic generator of your own devising: see the documentation on that class for more details.

The `random` module also provides the `SystemRandom` class which uses the system function `os.urandom()` to generate random numbers from sources provided by the operating system.

Warning

The pseudo-random generators of this module should not be used for security purposes. For security or cryptographic uses, see the `secrets` module.

See also

M. Matsumoto and T. Nishimura, “Mersenne Twister: A 623-dimensionally equidistributed uniform pseudorandom number generator”, ACM Transactions on Modeling and Computer Simulation Vol. 8, No. 1, January pp.3–30 1998.

Complementary-Multiply-with-Carry recipe [https://code.activestate.com/recipes/576707-long-period-random-number-generator/] for a compatible alternative random number generator with a long period and comparatively simple update operations.

Note

The global random number generator and instances of `Random` are thread-safe. However, in the free-threaded build, concurrent calls to the global generator or to the same instance of `Random` may encounter contention and poor performance. Consider using separate instances of `Random` per thread instead.

Bookkeeping functions

random.seed(*a=None*, *version=2*)

Initialize the random number generator.

If *a* is omitted or `None`, the current system time is used. If randomness sources are provided by the operating system, they are used instead of the system time (see the `os.urandom()` function for details on availability).

If *a* is an int, its absolute value is used directly.

With version 2 (the default), a `str`, `bytes`, or `bytearray` object gets converted to an `int` and all of its bits are used.

With version 1 (provided for reproducing random sequences from older versions of Python), the algorithm for `str` and `bytes` generates a narrower range of seeds.

Changed in version 3.2: Moved to the version 2 scheme which uses all of the bits in a string seed.

Changed in version 3.11: The *seed* must be one of the following types: `None`, `int`, `float`, `str`, `bytes`, or `bytearray`.

random.getstate()

Return an object capturing the current internal state of the generator. This object can be passed to `setstate()` to restore the state.

random.setstate(*state*)

state should have been obtained from a previous call to `getstate()`, and `setstate()` restores the internal state of the generator to what it was at the time `getstate()` was called.

Functions for bytes

random.randbytes(*n*)

Generate *n* random bytes.

This method should not be used for generating security tokens. Use `secrets.token_bytes()` instead.

Added in version 3.9.

Functions for integers

random.randrange(*stop*)

random.randrange(*start*, *stop*[, *step*])

Return a randomly selected element from `range(start, stop, step)`.

This is roughly equivalent to `choice(range(start, stop, step))` but supports arbitrarily large ranges and is optimized for common cases.

The positional argument pattern matches the `range()` function.

Keyword arguments should not be used because they can be interpreted in unexpected ways. For example `randrange(start=100)` is interpreted as `randrange(0, 100, 1)`.

Changed in version 3.2: **`randrange()`** is more sophisticated about producing equally distributed values. Formerly it used a style like `int(random()*n)` which could produce slightly uneven distributions.

Changed in version 3.12: Automatic conversion of non-integer types is no longer supported. Calls such as `randrange(10.0)` and `randrange(Fraction(10, 1))` now raise a **`TypeError`**.

random.randint(*a*, *b*)

Return a random integer *N* such that `a <= N <= b`. Alias for `randrange(a, b+1)`.

random.getrandbits(*k*)

Returns a non-negative Python integer with *k* random bits. This method is supplied with the Mersenne Twister generator and some other generators may also provide it as an optional part of the API. When available, **`getrandbits()`** enables **`randrange()`** to handle arbitrarily large ranges.

Changed in version 3.9: This method now accepts zero for *k*.

Functions for sequences

random.choice(*seq*)

Return a random element from the non-empty sequence *seq*. If *seq* is empty, raises **`IndexError`**.

random.choices(*population*, *weights=None*, ***, *cum_weights=None*, *k=1*)

Return a *k* sized list of elements chosen from the *population* with replacement. If the *population* is empty, raises `IndexError`.

If a *weights* sequence is specified, selections are made according to the relative weights. Alternatively, if a *cum_weights* sequence is given, the selections are made according to the cumulative weights (perhaps computed using `itertools.accumulate()`). For example, the relative weights `[10, 5, 30, 5]` are equivalent to the cumulative weights `[10, 15, 45, 50]`. Internally, the relative weights are converted to cumulative weights before making selections, so supplying the cumulative weights saves work.

If neither *weights* nor *cum_weights* are specified, selections are made with equal probability. If a weights sequence is supplied, it must be the same length as the *population* sequence. It is a `TypeError` to specify both *weights* and *cum_weights*.

The *weights* or *cum_weights* can use any numeric type that interoperates with the `float` values returned by `random()` (that includes integers, floats, and fractions but excludes decimals). Weights are assumed to be non-negative and finite. A `ValueError` is raised if all weights are zero.

For a given seed, the `choices()` function with equal weighting typically produces a different sequence than repeated calls to `choice()`. The algorithm used by `choices()` uses floating-point arithmetic for internal consistency and speed. The algorithm used by `choice()`

defaults to integer arithmetic with repeated selections to avoid small biases from round-off error.

Added in version 3.6.

Changed in version 3.9: Raises a `ValueError` if all weights are zero.

random.shuffle(*x*)

Shuffle the sequence *x* in place.

To shuffle an immutable sequence and return a new shuffled list, use `sample(x, k=len(x))` instead.

Note that even for small `len(x)`, the total number of permutations of *x* can quickly grow larger than the period of most random number generators. This implies that most permutations of a long sequence can never be generated. For example, a sequence of length 2080 is the largest that can fit within the period of the Mersenne Twister random number generator.

Changed in version 3.11: Removed the optional parameter *random*.

random.sample(*population*, *k*, *, *counts=None*)

Return a *k* length list of unique elements chosen from the population sequence. Used for random sampling without replacement.

Returns a new list containing elements from the population while leaving the original population unchanged. The resulting list is in selection order so that all sub-slices will also be valid random samples. This allows raffle winners (the sample) to be partitioned into grand prize and second place winners (the subslices).

Members of the population need not be hashable or unique. If the population contains repeats, then each occurrence is a possible selection in the sample.

Repeated elements can be specified one at a time or with the optional keyword-only *counts* parameter. For example, `sample(['red', 'blue'], counts=[4, 2], k=5)` is equivalent to `sample(['red', 'red', 'red', 'red', 'blue', 'blue'], k=5)`.

To choose a sample from a range of integers, use a `range()` object as an argument. This is especially fast and space efficient for sampling from a large population: `sample(range(10000000), k=60)`.

If the sample size is larger than the population size, a `ValueError` is raised.

Changed in version 3.9: Added the *counts* parameter.

Changed in version 3.11: The *population* must be a sequence. Automatic conversion of sets to lists is no longer supported.

Discrete distributions

The following function generates a discrete distribution.

random.binomialvariate(*n=1*, *p=0.5*)

Binomial distribution [https://mathworld.wolfram.com/BinomialDistribution.html]. Return the number of successes for *n* independent trials with the probability of success in each trial being *p*:

Mathematically equivalent to:

```
sum(random() < p for i in range(n))
```

The number of trials *n* should be a non-negative integer. The probability of success *p* should be between `0.0 <= p <= 1.0`. The result is an integer in the range `0 <= X <= n`.

Added in version 3.12.

Real-valued distributions

The following functions generate specific real-valued distributions. Function parameters are named after the corresponding variables in the distribution's equation, as used in common mathematical practice; most of these equations can be found in any statistics text.

random.random()

Return the next random floating-point number in the range `0.0 <= X < 1.0`

random.uniform(*a*, *b*)

Return a random floating-point number *N* such that `a <= N <= b` for `a <= b` and `b <= N <= a` for `b < a`.

The end-point value `b` may or may not be included in the range depending on floating-point rounding in the expression `a + (b-a) * random()`.

random.triangular(*low*, *high*, *mode*)

Return a random floating-point number *N* such that `low <= N <= high` and with the specified *mode* between those bounds. The *low* and *high* bounds default to zero and one. The *mode* argument defaults to the midpoint between the bounds, giving a symmetric distribution.

random.betavariate(*alpha*, *beta*)

Beta distribution. Conditions on the parameters are `alpha > 0` and `beta > 0`. Returned values range between 0 and 1.

random.expovariate(*lambd=1.0*)

Exponential distribution. *lambd* is 1.0 divided by the desired mean. It should be nonzero. (The parameter would be called “lambda”, but that is a reserved word in Python.) Returned values range from 0 to positive infinity if *lambd* is positive, and from negative infinity to 0 if *lambd* is negative.

Changed in version 3.12: Added the default value for `lambd`.

random.gammavariate(*alpha*, *beta*)

Gamma distribution. (*Not* the gamma function!) The shape and scale parameters, *alpha* and *beta*, must have positive values. (Calling conventions vary and some sources define ‘beta’ as the inverse of the scale).

The probability distribution function is:

```
          x ** (alpha - 1) * math.exp(-x / beta)
pdf(x) =  --------------------------------------
            math.gamma(alpha) * beta ** alpha
```

random.gauss(*mu=0.0*, *sigma=1.0*)

Normal distribution, also called the Gaussian distribution. *mu* is the mean, and *sigma* is the standard deviation. This is slightly faster than the `normalvariate()` function defined below.

Multithreading note: When two threads call this function simultaneously, it is possible that they will receive the same return value. This can be avoided in three ways. 1) Have each thread use a different instance of the random number generator. 2) Put locks around all calls. 3) Use the slower, but thread-safe `normalvariate()` function instead.

Changed in version 3.11: mu and *sigma* now have default arguments.

random.lognormvariate(*mu*, *sigma*)

Log normal distribution. If you take the natural logarithm of this distribution, you’ll get a normal distribution with mean *mu* and standard deviation *sigma*. *mu* can have any value, and *sigma* must be greater than zero.

random.normalvariate(*mu=0.0*, *sigma=1.0*)

Normal distribution. *mu* is the mean, and *sigma* is the standard deviation.

Changed in version 3.11: mu and *sigma* now have default arguments.

random.vonmisesvariate(*mu*, *kappa*)

mu is the mean angle, expressed in radians between 0 and 2**pi*, and *kappa* is the concentration parameter, which must be greater than or equal to zero. If *kappa* is equal to zero, this distribution reduces to a uniform random angle over the range 0 to 2**pi*.

random.paretovariate(*alpha*)

Pareto distribution. *alpha* is the shape parameter.

random.weibullvariate(*alpha*, *beta*)

Weibull distribution. *alpha* is the scale parameter and *beta* is the shape parameter.

Alternative Generator

class random.Random([*seed*])

Class that implements the default pseudo-random number generator used by the `random` module.

Changed in version 3.11: Formerly the *seed* could be any hashable object. Now it is limited to: `None`, `int`, `float`, `str`, `bytes`, or `bytearray`.

Subclasses of **`Random`** should override the following methods if they wish to make use of a different basic generator:

seed(*a=None*, *version=2*)

Override this method in subclasses to customise the `seed()` behaviour of **`Random`** instances.

getstate()

Override this method in subclasses to customise the `getstate()` behaviour of **`Random`** instances.

setstate(*state*)

Override this method in subclasses to customise the `setstate()` behaviour of **`Random`** instances.

random()

Override this method in subclasses to customise the `random()` behaviour of **`Random`** instances.

Optionally, a custom generator subclass can also supply the following method:

getrandbits(*k*)

Override this method in subclasses to customise the `getrandbits()` behaviour of **`Random`** instances.

randbytes(*n*)

Override this method in subclasses to customise the `randbytes()` behaviour of **`Random`** instances.

class random.SystemRandom([*seed*])

Class that uses the `os.urandom()` function for generating random numbers from sources provided by the operating system. Not available on all systems. Does not rely on software state, and sequences are not reproducible. Accordingly, the `seed()` method has no effect and is ignored. The `getstate()` and `setstate()` methods raise `NotImplementedError` if called.

Notes on Reproducibility

Sometimes it is useful to be able to reproduce the sequences given by a pseudo-random number generator. By reusing a seed value, the same sequence should be reproducible from run to run as long as multiple threads are not running.

Most of the random module’s algorithms and seeding functions are subject to change across Python versions, but two aspects are guaranteed not to change:

- If a new seeding method is added, then a backward compatible seeder will be offered.
- The generator’s `random()` method will continue to produce the same sequence when the compatible seeder is given the same seed.

Examples

Basic examples:

```
>>> random()                             # Random float:  0.0 <= x
< 1.0
0.37444887175646646

>>> uniform(2.5, 10.0)                   # Random float:  2.5 <= x
<= 10.0
3.1800146073117523

>>> expovariate(1 / 5)                   # Interval between
arrivals averaging 5 seconds
5.148957571865031

>>> randrange(10)                        # Integer from 0 to 9
inclusive
7

>>> randrange(0, 101, 2)                 # Even integer from 0 to
100 inclusive
26

>>> choice(['win', 'lose', 'draw'])   # Single random element
from a sequence
'draw'

>>> deck = 'ace two three four'.split()
>>> shuffle(deck)                        # Shuffle a list
>>> deck
['four', 'two', 'ace', 'three']

>>> sample([10, 20, 30, 40, 50], k=4) # Four samples without
replacement
[40, 10, 50, 30]
```

Simulations:

```
>>> # Six roulette wheel spins (weighted sampling with
replacement)
>>> choices(['red', 'black', 'green'], [18, 18, 2], k=6)
['red', 'green', 'black', 'black', 'red', 'black']

>>> # Deal 20 cards without replacement from a deck
>>> # of 52 playing cards, and determine the proportion of
cards
```

```
>>> # with a ten-value:  ten, jack, queen, or king.
>>> deal = sample(['tens', 'low cards'], counts=[16, 36], k=20)
>>> deal.count('tens') / 20
0.15

>>> # Estimate the probability of getting 5 or more heads from
7 spins
>>> # of a biased coin that settles on heads 60% of the time.
>>> sum(binomialvariate(n=7, p=0.6) >= 5 for i in
range(10_000)) / 10_000
0.4169

>>> # Probability of the median of 5 samples being in middle
two quartiles
>>> def trial():
...     return 2_500 <= sorted(choices(range(10_000), k=5))[2]
< 7_500
...
>>> sum(trial() for i in range(10_000)) / 10_000
0.7958
```

Example of statistical bootstrapping [https://en.wikipedia.org/wiki/Bootstrapping_(statistics)] using resampling with replacement to estimate a confidence interval for the mean of a sample:

```
# https://www.thoughtco.com/example-of-bootstrapping-3126155
from statistics import fmean as mean
from random import choices

data = [41, 50, 29, 37, 81, 30, 73, 63, 20, 35, 68, 22, 60, 31,
95]
means = sorted(mean(choices(data, k=len(data))) for i in
range(100))
print(f'The sample mean of {mean(data):.1f} has a 90%
confidence '
      f'interval from {means[5]:.1f} to {means[94]:.1f}')
```

Example of a resampling permutation test [https://en.wikipedia.org/wiki/Resampling_(statistics)#Permutation_tests] to determine the statistical significance or p-value [https://en.wikipedia.org/wiki/P-value] of an observed difference between the effects of a drug versus a placebo:

```
# Example from "Statistics is Easy" by Dennis Shasha and Manda
Wilson
from statistics import fmean as mean
from random import shuffle

drug = [54, 73, 53, 70, 73, 68, 52, 65, 65]
placebo = [54, 51, 58, 44, 55, 52, 42, 47, 58, 46]
observed_diff = mean(drug) - mean(placebo)

n = 10_000
count = 0
combined = drug + placebo
for i in range(n):
    shuffle(combined)
    new_diff = mean(combined[:len(drug)]) -
mean(combined[len(drug):])
    count += (new_diff >= observed_diff)

print(f'{n} label reshufflings produced only {count} instances
with a difference')
print(f'at least as extreme as the observed difference of
{observed_diff:.1f}.')
print(f'The one-sided p-value of {count / n:.4f} leads us to
reject the null')
print(f'hypothesis that there is no difference between the drug
and the placebo.')
```

Simulation of arrival times and service deliveries for a multiserver queue:

```
from heapq import heapify, heapreplace
from random import expovariate, gauss
from statistics import mean, quantiles

average_arrival_interval = 5.6
average_service_time = 15.0
stdev_service_time = 3.5
num_servers = 3
```

```
waits = []
arrival_time = 0.0
servers = [0.0] * num_servers  # time when each server becomes
available
heapify(servers)
for i in range(1_000_000):
    arrival_time += expovariate(1.0 / average_arrival_interval)
    next_server_available = servers[0]
    wait = max(0.0, next_server_available - arrival_time)
    waits.append(wait)
    service_duration = max(0.0, gauss(average_service_time,
stdev_service_time))
    service_completed = arrival_time + wait + service_duration
    heapreplace(servers, service_completed)

print(f'Mean wait: {mean(waits):.1f}   Max wait:
{max(waits):.1f}')
print('Quartiles:', [round(q, 1) for q in quantiles(waits)])
```

See also

Statistics for Hackers [https://www.youtube.com/watch?v=Iq9DzN6mvYA] a video tutorial by Jake Vanderplas [https://us.pycon.org/2016/speaker/profile/295/] on statistical analysis using just a few fundamental concepts including simulation, sampling, shuffling, and cross-validation.

Economics Simulation [https://nbviewer.org/url/norvig.com/ipython/Economics.ipynb] a simulation of a marketplace by Peter Norvig [https://norvig.com/bio.html] that shows effective use of many of the tools and distributions provided by this module (gauss, uniform, sample, betavariate, choice, triangular, and randrange).

A Concrete Introduction to Probability (using Python) [https://nbviewer.org/url/norvig.com/ipython/Probability.ipynb] a tutorial by Peter Norvig [https://norvig.com/bio.html] covering the basics of probability theory, how to write simulations, and how to perform data analysis using Python.

Recipes

These recipes show how to efficiently make random selections from the combinatoric iterators in the `itertools` module:

```
import random

def random_product(*iterables, repeat=1):
    "Random selection from itertools.product(*iterables,
repeat=repeat)"
    pools = tuple(map(tuple, iterables)) * repeat
    return tuple(map(random.choice, pools))

def random_permutation(iterable, r=None):
    "Random selection from itertools.permutations(iterable, r)"
    pool = tuple(iterable)
    r = len(pool) if r is None else r
    return tuple(random.sample(pool, r))

def random_combination(iterable, r):
    "Random selection from itertools.combinations(iterable, r)"
    pool = tuple(iterable)
    n = len(pool)
    indices = sorted(random.sample(range(n), r))
    return tuple(pool[i] for i in indices)

def random_combination_with_replacement(iterable, r):
    "Choose r elements with replacement.  Order the result to
match the iterable."
    # Result will be in
set(itertools.combinations_with_replacement(iterable, r)).
    pool = tuple(iterable)
    n = len(pool)
    indices = sorted(random.choices(range(n), k=r))
    return tuple(pool[i] for i in indices)
```

```
def random_derangement(iterable):
    "Choose a permutation where no element stays in its
original position."
    seq = tuple(iterable)
    if len(seq) < 2:
        if not seq:
            return ()
        raise IndexError('No derangments to choose from')
    perm = list(range(len(seq)))
    start = tuple(perm)
    while True:
        random.shuffle(perm)
        if all(p != q for p, q in zip(start, perm)):
            return tuple([seq[i] for i in perm])
```

The default `random()` returns multiples of 2^{-53} in the range $0.0 \leq x < 1.0$. All such numbers are evenly spaced and are exactly representable as Python floats. However, many other representable floats in that interval are not possible selections. For example, `0.05954861408025609` isn't an integer multiple of 2^{-53}.

The following recipe takes a different approach. All floats in the interval are possible selections. The mantissa comes from a uniform distribution of integers in the range $2^{52} \leq mantissa < 2^{53}$. The exponent comes from a geometric distribution where exponents smaller than *-53* occur half as often as the next larger exponent.

```
from random import Random
from math import ldexp

class FullRandom(Random):

    def random(self):
        mantissa = 0x10_0000_0000_0000 | self.getrandbits(52)
        exponent = -53
        x = 0
        while not x:
            x = self.getrandbits(32)
            exponent += x.bit_length() - 32
        return ldexp(mantissa, exponent)
```

All real valued distributions in the class will use the new method:

```
>>> fr = FullRandom()
>>> fr.random()
0.05954861408025609
>>> fr.expovariate(0.25)
8.87925541791544
```

The recipe is conceptually equivalent to an algorithm that chooses from all the multiples of 2^{-1074} in the range *0.0 ≤ x < 1.0*. All such numbers are evenly spaced, but most have to be rounded down to the nearest representable Python float. (The value 2^{-1074} is the smallest positive unnormalized float and is equal to `math.ulp(0.0)`.)

See also

Generating Pseudo-random Floating-Point Values [https://allendowney.com/research/rand/downey07randfloat.pdf] a paper by Allen B. Downey describing ways to generate more fine-grained floats than normally generated by **`random()`**.

Command-line usage

Added in version 3.13.

The **`random`** module can be executed from the command line.

```
python -m random [-h] [-c CHOICE [CHOICE ...] | -i N | -f N]
[input ...]
```

The following options are accepted:

-h, --help
Show the help message and exit.

-c CHOICE [CHOICE ...]
--choice CHOICE [CHOICE ...]
Print a random choice, using `choice()`.

-i <N>
--integer <N>
Print a random integer between 1 and N inclusive, using `randint()`.

-f <N>
--float <N>
Print a random floating-point number between 0 and N inclusive, using `uniform()`.

If no options are given, the output depends on the input:

- String or multiple: same as `--choice`.
- Integer: same as `--integer`.
- Float: same as `--float`.

Command-line example

Here are some examples of the **`random`** command-line interface:

```
$ # Choose one at random
$ python -m random egg bacon sausage spam "Lobster Thermidor
aux crevettes with a Mornay sauce"
Lobster Thermidor aux crevettes with a Mornay sauce

$ # Random integer
$ python -m random 6
6
```

```
$ # Random floating-point number
$ python -m random 1.8
1.7080016272295635

$ # With explicit arguments
$ python  -m random --choice egg bacon sausage spam "Lobster
Thermidor aux crevettes with a Mornay sauce"
egg

$ python -m random --integer 6
3

$ python -m random --float 1.8
1.5666339105010318

$ python -m random --integer 6
5

$ python -m random --float 6
3.1942323316565915
```

`statistics` — Mathematical statistics functions

Added in version 3.4.

Source code: Lib/statistics.py [https://github.com/python/cpython/tree/3.14/Lib/statistics.py]

This module provides functions for calculating mathematical statistics of numeric (**`Real`**-valued) data.

The module is not intended to be a competitor to third-party libraries such as NumPy [https://numpy.org], SciPy [https://scipy.org/], or proprietary full-featured statistics packages aimed at professional statisticians such as Minitab, SAS and Matlab. It is aimed at the level of graphing and scientific calculators.

Unless explicitly noted, these functions support **`int`**, **`float`**, **`Decimal`** and **`Fraction`**. Behaviour with other types (whether in the numeric tower or not) is currently unsupported. Collections with a mix of types are also undefined and implementation-dependent. If your input data consists of mixed types, you may be able to use **`map()`** to ensure a consistent result, for example: `map(float, input_data)`.

Some datasets use `NaN` (not a number) values to represent missing data. Since NaNs have unusual comparison semantics, they cause surprising or undefined behaviors in the statistics functions that sort data or that count occurrences. The functions affected are `median()`, `median_low()`, `median_high()`, `median_grouped()`, `mode()`, `multimode()`, and `quantiles()`. The `NaN` values should be stripped before calling these functions:

```
>>> from statistics import median
>>> from math import isnan
>>> from itertools import filterfalse

>>> data = [20.7, float('NaN'),19.2, 18.3, float('NaN'), 14.4]
>>> sorted(data)  # This has surprising behavior
[20.7, nan, 14.4, 18.3, 19.2, nan]
>>> median(data)  # This result is unexpected
16.35

>>> sum(map(isnan, data))    # Number of missing values
2
>>> clean = list(filterfalse(isnan, data))  # Strip NaN values
>>> clean
[20.7, 19.2, 18.3, 14.4]
>>> sorted(clean)  # Sorting now works as expected
[14.4, 18.3, 19.2, 20.7]
>>> median(clean)       # This result is now well defined
18.75
```

Averages and measures of central location

These functions calculate an average or typical value from a population or sample.

`mean()`	Arithmetic mean (“average”) of data.
`fmean()`	Fast, floating-point arithmetic mean, with optional weighting.
`geometric_mean()`	Geometric mean of data.

`harmonic_mean()`	Harmonic mean of data.
`kde()`	Estimate the probability density distribution of the data.
`kde_random()`	Random sampling from the PDF generated by kde().
`median()`	Median (middle value) of data.
`median_low()`	Low median of data.
`median_high()`	High median of data.
`median_grouped()`	Median (50th percentile) of grouped data.
`mode()`	Single mode (most common value) of discrete or nominal data.
`multimode()`	List of modes (most common values) of discrete or nominal data.
`quantiles()`	Divide data into intervals with equal probability.

Measures of spread

These functions calculate a measure of how much the population or sample tends to deviate from the typical or average values.

`pstdev()`	Population standard deviation of data.
`pvariance()`	Population variance of data.
`stdev()`	Sample standard deviation of data.
`variance()`	Sample variance of data.

Statistics for relations between two inputs

These functions calculate statistics regarding relations between two inputs.

`covariance()`	Sample covariance for two variables.
`correlation()`	Pearson and Spearman’s correlation coefficients.
`linear_regression()`	Slope and intercept for simple linear regression.

Function details

Note: The functions do not require the data given to them to be sorted. However, for reading convenience, most of the examples show sorted sequences.

statistics.mean(*data*)

Return the sample arithmetic mean of *data* which can be a sequence or iterable.

The arithmetic mean is the sum of the data divided by the number of data points. It is commonly called “the average”, although it is only one of many different mathematical averages. It is a measure of the central location of the data.

If *data* is empty, **`StatisticsError`** will be raised.

Some examples of use:

```
>>> mean([1, 2, 3, 4, 4])
2.8
>>> mean([-1.0, 2.5, 3.25, 5.75])
2.625

>>> from fractions import Fraction as F
>>> mean([F(3, 7), F(1, 21), F(5, 3), F(1, 3)])
Fraction(13, 21)

>>> from decimal import Decimal as D
>>> mean([D("0.5"), D("0.75"), D("0.625"), D("0.375")])
Decimal('0.5625')
```

Note

The mean is strongly affected by outliers [https://en.wikipedia.org/wiki/Outlier] and is not necessarily a typical example of the data points. For a more robust, although less efficient, measure of central tendency [https://en.wikipedia.org/wiki/Central_tendency], see **`median()`**.

The sample mean gives an unbiased estimate of the true population mean, so that when taken on average over all the possible samples, `mean(sample)` converges on the true mean of the entire population. If *data* represents the entire population rather than a sample, then `mean(data)` is equivalent to calculating the true population mean μ.

statistics.fmean(*data*, *weights=None*)

Convert *data* to floats and compute the arithmetic mean.

This runs faster than the **`mean()`** function and it always returns a **`float`**. The *data* may be a sequence or iterable. If the input dataset is empty, raises a **`StatisticsError`**.

```
>>> fmean([3.5, 4.0, 5.25])
4.25
```

Optional weighting is supported. For example, a professor assigns a grade for a course by weighting quizzes at 20%, homework at 20%, a midterm exam at 30%, and a final exam at 30%:

```
>>> grades = [85, 92, 83, 91]
>>> weights = [0.20, 0.20, 0.30, 0.30]
>>> fmean(grades, weights)
87.6
```

If *weights* is supplied, it must be the same length as the *data* or a **`ValueError`** will be raised.

Added in version 3.8.

Changed in version 3.11: Added support for *weights*.

statistics.geometric_mean(*data*)

Convert *data* to floats and compute the geometric mean.

The geometric mean indicates the central tendency or typical value of the *data* using the product of the values (as opposed to the arithmetic mean which uses their sum).

Raises a `StatisticsError` if the input dataset is empty, if it contains a zero, or if it contains a negative value. The *data* may be a sequence or iterable.

No special efforts are made to achieve exact results. (However, this may change in the future.)

```
>>> round(geometric_mean([54, 24, 36]), 1)
36.0
```

Added in version 3.8.

statistics.harmonic_mean(*data*, *weights=None*)

Return the harmonic mean of *data*, a sequence or iterable of real-valued numbers. If *weights* is omitted or `None`, then equal weighting is assumed.

The harmonic mean is the reciprocal of the arithmetic `mean()` of the reciprocals of the data. For example, the harmonic mean of three values *a*, *b* and *c* will be equivalent to `3/(1/a + 1/b + 1/c)`. If one of the values is zero, the result will be zero.

The harmonic mean is a type of average, a measure of the central location of the data. It is often appropriate when averaging ratios or rates, for example speeds.

Suppose a car travels 10 km at 40 km/hr, then another 10 km at 60 km/hr. What is the average speed?

```
>>> harmonic_mean([40, 60])
48.0
```

Suppose a car travels 40 km/hr for 5 km, and when traffic clears, speeds-up to 60 km/hr for the remaining 30 km of the journey. What is the average speed?

```
>>> harmonic_mean([40, 60], weights=[5, 30])
56.0
```

`StatisticsError` is raised if *data* is empty, any element is less than zero, or if the weighted sum isn't positive.

The current algorithm has an early-out when it encounters a zero in the input. This means that the subsequent inputs are not tested for validity. (This behavior may change in the future.)

Added in version 3.6.

Changed in version 3.10: Added support for *weights*.

statistics.kde(*data*, *h*, *kernel='normal'*, *, *cumulative=False*)

Kernel Density Estimation (KDE) [https://www.itm-conferences.org/articles/itmconf/pdf/2018/08/itmconf_sam2018_00037.pdf]: Create a continuous probability density function or cumulative distribution function from discrete samples.

The basic idea is to smooth the data using a kernel function [https://en.wikipedia.org/wiki/Kernel_(statistics)]. to help draw inferences about a population from a sample.

The degree of smoothing is controlled by the scaling parameter *h* which is called the bandwidth. Smaller values emphasize local features while larger values give smoother results.

The *kernel* determines the relative weights of the sample data points. Generally, the choice of kernel shape does not matter as much as the more influential bandwidth smoothing parameter.

Kernels that give some weight to every sample point include *normal* (*gauss*), *logistic*, and *sigmoid*.

Kernels that only give weight to sample points within the bandwidth include *rectangular* (*uniform*), *triangular*, *parabolic* (*epanechnikov*), *quartic* (*biweight*), *triweight*, and *cosine*.

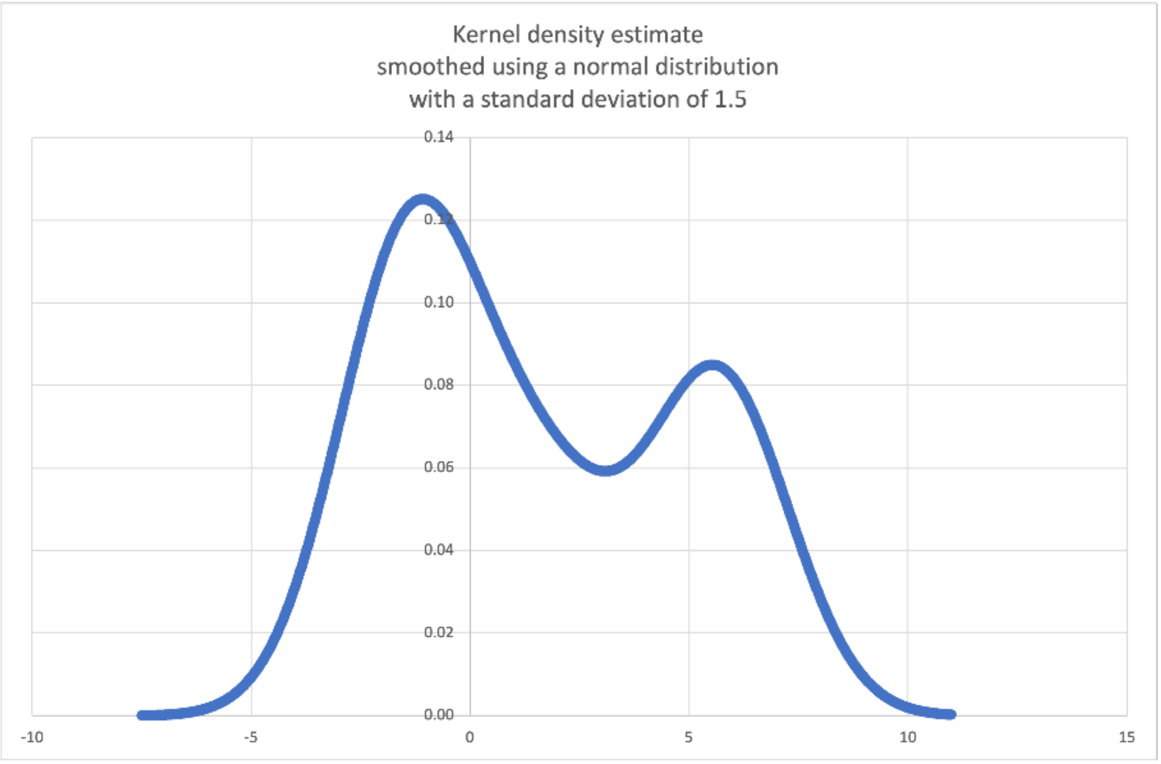
If *cumulative* is true, will return a cumulative distribution function.

A **`StatisticsError`** will be raised if the *data* sequence is empty.

Wikipedia has an example [https://en.wikipedia.org/wiki/Kernel_density_estimation#Example] where we can use **`kde()`** to generate and plot a probability density function estimated from a small sample:

```
>>> sample = [-2.1, -1.3, -0.4, 1.9, 5.1, 6.2]
>>> f_hat = kde(sample, h=1.5)
>>> xarr = [i/100 for i in range(-750, 1100)]
>>> yarr = [f_hat(x) for x in xarr]
```

The points in `xarr` and `yarr` can be used to make a PDF plot:



Added in version 3.13.

statistics.kde_random(*data*, *h*, *kernel='normal'*, ***, *seed=None*)

Return a function that makes a random selection from the estimated probability density function produced by `kde(data, h, kernel)`.

Providing a *seed* allows reproducible selections. In the future, the values may change slightly as more accurate kernel inverse CDF estimates are implemented. The seed may be an integer, float, str, or bytes.

A `StatisticsError` will be raised if the *data* sequence is empty.

Continuing the example for `kde()`, we can use `kde_random()` to generate new random selections from an estimated probability density function:

```
>>> data = [-2.1, -1.3, -0.4, 1.9, 5.1, 6.2]
>>> rand = kde_random(data, h=1.5, seed=8675309)
>>> new_selections = [rand() for i in range(10)]
>>> [round(x, 1) for x in new_selections]
[0.7, 6.2, 1.2, 6.9, 7.0, 1.8, 2.5, -0.5, -1.8, 5.6]
```

Added in version 3.13.

statistics.median(*data*)

Return the median (middle value) of numeric data, using the common “mean of middle two” method. If *data* is empty, **`StatisticsError`** is raised. *data* can be a sequence or iterable.

The median is a robust measure of central location and is less affected by the presence of outliers. When the number of data points is odd, the middle data point is returned:

```
>>> median([1, 3, 5])
3
```

When the number of data points is even, the median is interpolated by taking the average of the two middle values:

```
>>> median([1, 3, 5, 7])
4.0
```

This is suited for when your data is discrete, and you don’t mind that the median may not be an actual data point.

If the data is ordinal (supports order operations) but not numeric (doesn’t support addition), consider using **`median_low()`** or **`median_high()`** instead.

statistics.median_low(*data*)

Return the low median of numeric data. If *data* is empty, `StatisticsError` is raised. *data* can be a sequence or iterable.

The low median is always a member of the data set. When the number of data points is odd, the middle value is returned. When it is even, the smaller of the two middle values is returned.

```
>>> median_low([1, 3, 5])
3
>>> median_low([1, 3, 5, 7])
3
```

Use the low median when your data are discrete and you prefer the median to be an actual data point rather than interpolated.

statistics.median_high(*data*)

Return the high median of data. If *data* is empty, `StatisticsError` is raised. *data* can be a sequence or iterable.

The high median is always a member of the data set. When the number of data points is odd, the middle value is returned. When it is even, the larger of the two middle values is returned.

```
>>> median_high([1, 3, 5])
3
>>> median_high([1, 3, 5, 7])
5
```

Use the high median when your data are discrete and you prefer the median to be an actual data point rather than interpolated.

statistics.median_grouped(*data*, *interval=1.0*)

Estimates the median for numeric data that has been grouped or binned [https://en.wikipedia.org/wiki/Data_binning] around the midpoints of consecutive, fixed-width intervals.

The *data* can be any iterable of numeric data with each value being exactly the midpoint of a bin. At least one value must be present.

The *interval* is the width of each bin.

For example, demographic information may have been summarized into consecutive ten-year age groups with each group being represented by the 5-year midpoints of the intervals:

```
>>> from collections import Counter
>>> demographics = Counter({
...    25: 172,   # 20 to 30 years old
...    35: 484,   # 30 to 40 years old
...    45: 387,   # 40 to 50 years old
...    55:  22,   # 50 to 60 years old
...    65:   6,   # 60 to 70 years old
... })
...
```

The 50th percentile (median) is the 536th person out of the 1071 member cohort. That person is in the 30 to 40 year old age group.

The regular **`median()`** function would assume that everyone in the tricenarian age group was exactly 35 years old. A more tenable assumption is that the 484 members of that age group are evenly distributed between 30 and 40. For that, we use **`median_grouped()`**:

```
>>> data = list(demographics.elements())
>>> median(data)
35
```

```
>>> round(median_grouped(data, interval=10), 1)
37.5
```

The caller is responsible for making sure the data points are separated by exact multiples of *interval*. This is essential for getting a correct result. The function does not check this precondition.

Inputs may be any numeric type that can be coerced to a float during the interpolation step.

statistics.mode(*data*)

Return the single most common data point from discrete or nominal *data*. The mode (when it exists) is the most typical value and serves as a measure of central location.

If there are multiple modes with the same frequency, returns the first one encountered in the *data*. If the smallest or largest of those is desired instead, use `min(multimode(data))` or `max(multimode(data))`. If the input *data* is empty, **`StatisticsError`** is raised.

`mode` assumes discrete data and returns a single value. This is the standard treatment of the mode as commonly taught in schools:

```
>>> mode([1, 1, 2, 3, 3, 3, 3, 4])
3
```

The mode is unique in that it is the only statistic in this package that also applies to nominal (non-numeric) data:

```
>>> mode(["red", "blue", "blue", "red", "green", "red",
"red"])
'red'
```

Only hashable inputs are supported. To handle type **`set`**, consider casting to **`frozenset`**. To handle type **`list`**, consider casting to **`tuple`**. For mixed or nested inputs, consider using this slower quadratic algorithm that only depends on equality tests: `max(data, key=data.count)`.

Changed in version 3.8: Now handles multimodal datasets by returning the first mode encountered. Formerly, it raised **`StatisticsError`** when more than one mode was found.

statistics.multimode(*data*)

Return a list of the most frequently occurring values in the order they were first encountered in the *data*. Will return more than one result if there are multiple modes or an empty list if the *data* is empty:

```
>>> multimode('aabbbbccddddeeffffgg')
['b', 'd', 'f']
>>> multimode('')
[]
```

Added in version 3.8.

statistics.pstdev(*data*, *mu=None*)

Return the population standard deviation (the square root of the population variance). See **`pvariance()`** for arguments and other details.

```
>>> pstdev([1.5, 2.5, 2.5, 2.75, 3.25, 4.75])
0.986893273527251
```

statistics.pvariance(*data*, *mu=None*)

Return the population variance of *data*, a non-empty sequence or iterable of real-valued numbers. Variance, or second moment about the mean, is a measure of the variability (spread or dispersion) of data. A large variance indicates that the data is spread out; a small variance indicates it is clustered closely around the mean.

If the optional second argument *mu* is given, it should be the *population* mean of the *data*. It can also be used to compute the second moment around a point that is not the mean. If it is missing or `None` (the default), the arithmetic mean is automatically calculated.

Use this function to calculate the variance from the entire population. To estimate the variance from a sample, the `variance()` function is usually a better choice.

Raises `StatisticsError` if *data* is empty.

Examples:

```
>>> data = [0.0, 0.25, 0.25, 1.25, 1.5, 1.75, 2.75, 3.25]
>>> pvariance(data)
1.25
```

If you have already calculated the mean of your data, you can pass it as the optional second argument *mu* to avoid recalculation:

```
>>> mu = mean(data)
>>> pvariance(data, mu)
1.25
```

Decimals and Fractions are supported:

```
>>> from decimal import Decimal as D
>>> pvariance([D("27.5"), D("30.25"), D("30.25"), D("34.5"),
D("41.75")])
Decimal('24.815')

>>> from fractions import Fraction as F
>>> pvariance([F(1, 4), F(5, 4), F(1, 2)])
Fraction(13, 72)
```

Note

When called with the entire population, this gives the population variance σ². When called on a sample instead, this is the biased sample variance s², also known as variance with N degrees of freedom.

If you somehow know the true population mean μ, you may use this function to calculate the variance of a sample, giving the known population mean as the second argument. Provided the data points are a random sample of the population, the result will be an unbiased estimate of the population variance.

statistics.stdev(*data*, *xbar=None*)

Return the sample standard deviation (the square root of the sample variance). See `variance()` for arguments and other details.

```
>>> stdev([1.5, 2.5, 2.5, 2.75, 3.25, 4.75])
1.0810874155219827
```

statistics.variance(*data*, *xbar=None*)

Return the sample variance of *data*, an iterable of at least two real-valued numbers. Variance, or second moment about the mean, is a measure of the variability (spread or dispersion) of data. A large variance indicates that the data is spread out; a small variance indicates it is clustered closely around the mean.

If the optional second argument *xbar* is given, it should be the *sample* mean of *data*. If it is missing or `None` (the default), the mean is automatically calculated.

Use this function when your data is a sample from a population. To calculate the variance from the entire population, see `pvariance()`.

Raises `StatisticsError` if *data* has fewer than two values.

Examples:

```
>>> data = [2.75, 1.75, 1.25, 0.25, 0.5, 1.25, 3.5]
>>> variance(data)
1.3720238095238095
```

If you have already calculated the sample mean of your data, you can pass it as the optional second argument *xbar* to avoid recalculation:

```
>>> m = mean(data)
>>> variance(data, m)
1.3720238095238095
```

This function does not attempt to verify that you have passed the actual mean as *xbar*. Using arbitrary values for *xbar* can lead to invalid or impossible results.

Decimal and Fraction values are supported:

```
>>> from decimal import Decimal as D
>>> variance([D("27.5"), D("30.25"), D("30.25"), D("34.5"),
D("41.75")])
Decimal('31.01875')

>>> from fractions import Fraction as F
>>> variance([F(1, 6), F(1, 2), F(5, 3)])
Fraction(67, 108)
```

Note

This is the sample variance s^2 with Bessel’s correction, also known as variance with N-1 degrees of freedom. Provided that the data points are representative (e.g. independent and identically distributed), the result should be an unbiased estimate of the true population variance.

If you somehow know the actual population mean µ you should pass it to the `pvariance()` function as the *mu* parameter to get the variance of a sample.

statistics.quantiles(*data*, *, *n=4*, *method='exclusive'*)

Divide *data* into *n* continuous intervals with equal probability. Returns a list of `n - 1` cut points separating the intervals.

Set *n* to 4 for quartiles (the default). Set *n* to 10 for deciles. Set *n* to 100 for percentiles which gives the 99 cuts points that separate *data* into 100 equal sized groups. Raises `StatisticsError` if *n* is not least 1.

The *data* can be any iterable containing sample data. For meaningful results, the number of data points in *data* should be larger than *n*. Raises `StatisticsError` if there is not at least one data point.

The cut points are linearly interpolated from the two nearest data points. For example, if a cut point falls one-third of the distance between two sample values, `100` and `112`, the cut-point will evaluate to `104`.

The *method* for computing quantiles can be varied depending on whether the *data* includes or excludes the lowest and highest possible values from the population.

The default *method* is “exclusive” and is used for data sampled from a population that can have more extreme values than found in the samples. The portion of the population falling below the *i-th* of *m* sorted data points is computed as `i / (m + 1)`. Given nine sample values, the method sorts them and assigns the following percentiles: 10%, 20%, 30%, 40%, 50%, 60%, 70%, 80%, 90%.

Setting the *method* to “inclusive” is used for describing population data or for samples that are known to include the most extreme values from the population. The minimum value in *data* is treated as the 0th percentile and the maximum value is treated as the 100th percentile. The portion of the population falling below the *i-th* of *m* sorted data points is computed as `(i - 1) / (m - 1)`. Given 11 sample values, the method sorts them and assigns the following percentiles: 0%, 10%, 20%, 30%, 40%, 50%, 60%, 70%, 80%, 90%, 100%.

```
# Decile cut points for empirically sampled data
>>> data = [105, 129, 87, 86, 111, 111, 89, 81, 108, 92,
110,
...         100, 75, 105, 103, 109, 76, 119, 99, 91, 103,
129,
...         106, 101, 84, 111, 74, 87, 86, 103, 103, 106,
86,
...         111, 75, 87, 102, 121, 111, 88, 89, 101, 106,
95,
...         103, 107, 101, 81, 109, 104]
>>> [round(q, 1) for q in quantiles(data, n=10)]
[81.0, 86.2, 89.0, 99.4, 102.5, 103.6, 106.0, 109.8, 111.0]
```

Added in version 3.8.

Changed in version 3.13: No longer raises an exception for an input with only a single data point. This allows quantile estimates to be built up one sample point at a time becoming gradually more refined with each new data point.

statistics.covariance(*x*, *y*, */*)

Return the sample covariance of two inputs *x* and *y*. Covariance is a measure of the joint variability of two inputs.

Both inputs must be of the same length (no less than two), otherwise `StatisticsError` is raised.

Examples:

```
>>> x = [1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> y = [1, 2, 3, 1, 2, 3, 1, 2, 3]
>>> covariance(x, y)
0.75
>>> z = [9, 8, 7, 6, 5, 4, 3, 2, 1]
>>> covariance(x, z)
-7.5
>>> covariance(z, x)
-7.5
```

Added in version 3.10.

statistics.correlation(*x*, *y*, */*, ***, *method='linear'*)

Return the Pearson’s correlation coefficient [https://en.wikipedia.org/wiki/Pearson_correlation_coefficient] for two inputs. Pearson’s correlation coefficient *r* takes values between -1 and +1. It measures the strength and direction of a linear relationship.

If *method* is “ranked”, computes Spearman’s rank correlation coefficient [https://en.wikipedia.org/wiki/Spearman%27s_rank_correlation_coefficient] for two inputs. The data is replaced by ranks. Ties are averaged so that equal values receive the same rank. The resulting coefficient measures the strength of a monotonic relationship.

Spearman’s correlation coefficient is appropriate for ordinal data or for continuous data that doesn’t meet the linear proportion requirement for Pearson’s correlation coefficient.

Both inputs must be of the same length (no less than two), and need not to be constant, otherwise `StatisticsError` is raised.

Example with Kepler’s laws of planetary motion [https://en.wikipedia.org/wiki/Kepler's_laws_of_planetary_motion]:

```
>>> # Mercury, Venus, Earth, Mars, Jupiter, Saturn, Uranus,
and  Neptune
>>> orbital_period = [88, 225, 365, 687, 4331, 10_756,
30_687, 60_190]    # days
>>> dist_from_sun = [58, 108, 150, 228, 778, 1_400, 2_900,
4_500] # million km

>>> # Show that a perfect monotonic relationship exists
>>> correlation(orbital_period, dist_from_sun,
method='ranked')
1.0

>>> # Observe that a linear relationship is imperfect
>>> round(correlation(orbital_period, dist_from_sun), 4)
0.9882

>>> # Demonstrate Kepler's third law: There is a linear
correlation
>>> # between the square of the orbital period and the cube
of the
>>> # distance from the sun.
>>> period_squared = [p * p for p in orbital_period]
>>> dist_cubed = [d * d * d for d in dist_from_sun]
>>> round(correlation(period_squared, dist_cubed), 4)
1.0
```

Added in version 3.10.

Changed in version 3.12: Added support for Spearman’s rank correlation coefficient.

statistics.linear_regression(*x*, *y*, */*, ***, *proportional=False*)

Return the slope and intercept of simple linear regression [https://en.wikipedia.org/wiki/Simple_linear_regression] parameters estimated using ordinary least squares. Simple linear regression describes the relationship between an independent variable *x* and a dependent variable *y* in terms of this linear function:

$$y = slope * x + intercept + noise$$

where `slope` and `intercept` are the regression parameters that are estimated, and `noise` represents the variability of the data that was not explained by the linear regression (it is equal to the difference between predicted and actual values of the dependent variable).

Both inputs must be of the same length (no less than two), and the independent variable *x* cannot be constant; otherwise a **`StatisticsError`** is raised.

For example, we can use the release dates of the Monty Python films [https://en.wikipedia.org/wiki/Monty_Python#Films] to predict the cumulative number of Monty Python films that would have been produced by 2019 assuming that they had kept the pace.

```
>>> year = [1971, 1975, 1979, 1982, 1983]
>>> films_total = [1, 2, 3, 4, 5]
>>> slope, intercept = linear_regression(year, films_total)
>>> round(slope * 2019 + intercept)
16
```

If *proportional* is true, the independent variable *x* and the dependent variable *y* are assumed to be directly proportional. The data is fit to a line passing through the origin. Since the *intercept* will always be 0.0, the underlying linear function simplifies to:

$$y = slope * x + noise$$

Continuing the example from [`correlation()`](), we look to see how well a model based on major planets can predict the orbital distances for dwarf planets:

```
>>> model = linear_regression(period_squared, dist_cubed, 
proportional=True)
>>> slope = model.slope

>>> # Dwarf planets:   Pluto,  Eris,    Makemake, Haumea, 
Ceres
>>> orbital_periods = [90_560, 204_199, 111_845, 103_410, 
1_680]  # days
>>> predicted_dist = [math.cbrt(slope * (p * p)) for p in 
orbital_periods]
>>> list(map(round, predicted_dist))
[5912, 10166, 6806, 6459, 414]

>>> [5_906, 10_152, 6_796, 6_450, 414]  # actual distance in 
million km
[5906, 10152, 6796, 6450, 414]
```

Added in version 3.10.

Changed in version 3.11: Added support for *proportional*.

Exceptions

A single exception is defined:

exception statistics.StatisticsError

Subclass of **`ValueError`** for statistics-related exceptions.

`NormalDist` objects

`NormalDist` is a tool for creating and manipulating normal distributions of a random variable [http://www.stat.yale.edu/Courses/1997-98/101/ranvar.htm]. It is a class that treats the mean and standard deviation of data measurements as a single entity.

Normal distributions arise from the Central Limit Theorem [https://en.wikipedia.org/wiki/Central_limit_theorem] and have a wide range of applications in statistics.

class statistics.NormalDist(*mu=0.0*, *sigma=1.0*)

Returns a new *NormalDist* object where *mu* represents the arithmetic mean [https://en.wikipedia.org/wiki/Arithmetic_mean] and *sigma* represents the standard deviation [https://en.wikipedia.org/wiki/Standard_deviation].

If *sigma* is negative, raises **`StatisticsError`**.

mean

A read-only property for the arithmetic mean [https://en.wikipedia.org/wiki/Arithmetic_mean] of a normal distribution.

median

A read-only property for the median [https://en.wikipedia.org/wiki/Median] of a normal distribution.

mode

A read-only property for the mode [https://en.wikipedia.org/wiki/Mode_(statistics)] of a normal distribution.

stdev

A read-only property for the standard deviation [https://en.wikipedia.org/wiki/Standard_deviation] of a normal distribution.

variance

A read-only property for the variance [https://en.wikipedia.org/wiki/Variance] of a normal distribution. Equal to the square of the standard deviation.

classmethod from_samples(*data*)

Makes a normal distribution instance with *mu* and *sigma* parameters estimated from the *data* using `fmean()` and `stdev()`.

The *data* can be any iterable and should consist of values that can be converted to type `float`. If *data* does not contain at least two elements, raises `StatisticsError` because it takes at least one point to estimate a central value and at least two points to estimate dispersion.

samples(*n*, *, *seed=None*)

Generates *n* random samples for a given mean and standard deviation. Returns a `list` of `float` values.

If *seed* is given, creates a new instance of the underlying random number generator. This is useful for creating reproducible results, even in a multi-threading context.

Changed in version 3.13.

Switched to a faster algorithm. To reproduce samples from previous versions, use `random.seed()` and `random.gauss()`.

pdf(*x*)

Using a probability density function (pdf) [https://en.wikipedia.org/wiki/Probability_density_function], compute the relative likelihood that a random variable *X* will be near the given value *x*. Mathematically, it is the limit of the ratio `P(x <= X < x+dx) / dx` as *dx* approaches zero.

The relative likelihood is computed as the probability of a sample occurring in a narrow range divided by the width of the range (hence the word “density”). Since the likelihood is relative to other points, its value can be greater than `1.0`.

cdf(*x*)

Using a cumulative distribution function (cdf) [https://en.wikipedia.org/wiki/Cumulative_distribution_function], compute the probability that a random variable *X* will be less than or equal to *x*. Mathematically, it is written `P(X <= x)`.

inv_cdf(*p*)

Compute the inverse cumulative distribution function, also known as the quantile function [https://en.wikipedia.org/wiki/Quantile_function] or the percent-point [https://web.archive.org/web/20190203145224/https://www.statisticshowto.datasciencecentral.com/inverse-distribution-function/] function. Mathematically, it is written `x : P(X <= x) = p`.

Finds the value *x* of the random variable *X* such that the probability of the variable being less than or equal to that value equals the given probability *p*.

overlap(*other*)

Measures the agreement between two normal probability distributions. Returns a value between 0.0 and 1.0 giving the overlapping area for the two probability density functions [https://www.rasch.org/rmt/rmt101r.htm].

quantiles(*n=4*)

Divide the normal distribution into *n* continuous intervals with equal probability. Returns a list of (n - 1) cut points separating the intervals.

Set *n* to 4 for quartiles (the default). Set *n* to 10 for deciles. Set *n* to 100 for percentiles which gives the 99 cuts points that separate the normal distribution into 100 equal sized groups.

zscore(*x*)

Compute the Standard Score [https://www.statisticshowto.com/probability-and-statistics/z-score/] describing *x* in terms of the number of standard deviations above or below the mean of the normal distribution: `(x - mean) / stdev`.

Added in version 3.9.

Instances of **`NormalDist`** support addition, subtraction, multiplication and division by a constant. These operations are used for translation and scaling. For example:

```
>>> temperature_february = NormalDist(5, 2.5)             #
Celsius
>>> temperature_february * (9/5) + 32                     #
Fahrenheit
NormalDist(mu=41.0, sigma=4.5)
```

Dividing a constant by an instance of **`NormalDist`** is not supported because the result wouldn’t be normally distributed.

Since normal distributions arise from additive effects of independent variables, it is possible to add and subtract two independent normally distributed random variables [https://en.wikipedia.org/wiki/Sum_of_normally_distributed_random_variables] represented as instances of [`NormalDist`]. For example:

```
>>> birth_weights = NormalDist.from_samples([2.5, 3.1, 2.1,
2.4, 2.7, 3.5])
>>> drug_effects = NormalDist(0.4, 0.15)
>>> combined = birth_weights + drug_effects
>>> round(combined.mean, 1)
3.1
>>> round(combined.stdev, 1)
0.5
```

Added in version 3.8.

Examples and Recipes

Classic probability problems

`NormalDist` readily solves classic probability problems.

For example, given historical data for SAT exams [https://nces.ed.gov/programs/digest/d17/tables/dt17_226.40.asp] showing that scores are normally distributed with a mean of 1060 and a standard deviation of 195, determine the percentage of students with test scores between 1100 and 1200, after rounding to the nearest whole number:

```
>>> sat = NormalDist(1060, 195)
>>> fraction = sat.cdf(1200 + 0.5) - sat.cdf(1100 - 0.5)
>>> round(fraction * 100.0, 1)
18.4
```

Find the quartiles [https://en.wikipedia.org/wiki/Quartile] and deciles [https://en.wikipedia.org/wiki/Decile] for the SAT scores:

```
>>> list(map(round, sat.quantiles()))
[928, 1060, 1192]
>>> list(map(round, sat.quantiles(n=10)))
[810, 896, 958, 1011, 1060, 1109, 1162, 1224, 1310]
```

Monte Carlo inputs for simulations

To estimate the distribution for a model that isn’t easy to solve analytically, `NormalDist` can generate input samples for a Monte Carlo simulation [https://en.wikipedia.org/wiki/Monte_Carlo_method]:

```
>>> def model(x, y, z):
...     return (3*x + 7*x*y - 5*y) / (11 * z)
...
>>> n = 100_000
>>> X = NormalDist(10, 2.5).samples(n, seed=3652260728)
>>> Y = NormalDist(15, 1.75).samples(n, seed=4582495471)
>>> Z = NormalDist(50, 1.25).samples(n, seed=6582483453)
>>> quantiles(map(model, X, Y, Z))
[1.4591308524824727, 1.8035946855390597, 2.175091447274739]
```

Approximating binomial distributions

Normal distributions can be used to approximate Binomial distributions [https://mathworld.wolfram.com/BinomialDistribution.html] when the sample size is large and when the probability of a successful trial is near 50%.

For example, an open source conference has 750 attendees and two rooms with a 500 person capacity. There is a talk about Python and another about Ruby. In previous conferences, 65% of the attendees preferred to listen to Python talks. Assuming the population preferences haven’t changed, what is the probability that the Python room will stay within its capacity limits?

```
>>> n = 750             # Sample size
>>> p = 0.65            # Preference for Python
>>> q = 1.0 - p         # Preference for Ruby
>>> k = 500             # Room capacity

>>> # Approximation using the cumulative normal distribution
>>> from math import sqrt
>>> round(NormalDist(mu=n*p, sigma=sqrt(n*p*q)).cdf(k + 0.5),
4)
0.8402

>>> # Exact solution using the cumulative binomial distribution
>>> from math import comb, fsum
>>> round(fsum(comb(n, r) * p**r * q**(n-r) for r in
range(k+1)), 4)
0.8402

>>> # Approximation using a simulation
>>> from random import seed, binomialvariate
>>> seed(8675309)
>>> mean(binomialvariate(n, p) <= k for i in range(10_000))
0.8406
```

Naive bayesian classifier

Normal distributions commonly arise in machine learning problems.

Wikipedia has a nice example of a Naive Bayesian Classifier [https://en.wikipedia.org/wiki/Naive_Bayes_classifier#Person_classification]. The challenge is to predict a person's gender from measurements of normally distributed features including height, weight, and foot size.

We're given a training dataset with measurements for eight people. The measurements are assumed to be normally distributed, so we summarize the data with **`NormalDist`**:

```
>>> height_male = NormalDist.from_samples([6, 5.92, 5.58,
5.92])
>>> height_female = NormalDist.from_samples([5, 5.5, 5.42,
5.75])
>>> weight_male = NormalDist.from_samples([180, 190, 170, 165])
>>> weight_female = NormalDist.from_samples([100, 150, 130,
150])
>>> foot_size_male = NormalDist.from_samples([12, 11, 12, 10])
>>> foot_size_female = NormalDist.from_samples([6, 8, 7, 9])
```

Next, we encounter a new person whose feature measurements are known but whose gender is unknown:

```
>>> ht = 6.0        # height
>>> wt = 130        # weight
>>> fs = 8          # foot size
```

Starting with a 50% prior probability [https://en.wikipedia.org/wiki/Prior_probability] of being male or female, we compute the posterior as the prior times the product of likelihoods for the feature measurements given the gender:

```
>>> prior_male = 0.5
>>> prior_female = 0.5
>>> posterior_male = (prior_male * height_male.pdf(ht) *
...                   weight_male.pdf(wt) *
foot_size_male.pdf(fs))

>>> posterior_female = (prior_female * height_female.pdf(ht) *
...                     weight_female.pdf(wt) *
foot_size_female.pdf(fs))
```

The final prediction goes to the largest posterior. This is known as the maximum a posteriori [https://en.wikipedia.org/wiki/Maximum_a_posteriori_estimation] or MAP:

```
>>> 'male' if posterior_male > posterior_female else 'female'
'female'
```

Functional Programming Modules

The modules described in this chapter provide functions and classes that support a functional programming style, and general operations on callables.

The following modules are documented in this chapter:

- **`itertools`** — Functions creating iterators for efficient looping
    - Itertool Functions
    - Itertools Recipes
- **`functools`** — Higher-order functions and operations on callable objects
    - **`partial`** Objects
- **`operator`** — Standard operators as functions
    - Mapping Operators to Functions
    - In-place Operators

`itertools` — Functions creating iterators for efficient looping

This module implements a number of iterator building blocks inspired by constructs from APL, Haskell, and SML. Each has been recast in a form suitable for Python.

The module standardizes a core set of fast, memory efficient tools that are useful by themselves or in combination. Together, they form an “iterator algebra” making it possible to construct specialized tools succinctly and efficiently in pure Python.

For instance, SML provides a tabulation tool: `tabulate(f)` which produces a sequence `f(0), f(1), ...`. The same effect can be achieved in Python by combining **`map()`** and **`count()`** to form `map(f, count())`.

Infinite iterators:

Iterator	Arguments	Results	Example
`count()`	[start[, step]]	start, start+step, start+2*step, …	`count(10) → 10 11 12 13 14 ...`
`cycle()`	p	p0, p1, … plast, p0, p1, …	`cycle('ABCD') → A B C D A B C D ...`
`repeat()`	elem [,n]	elem, elem, elem, … endlessly or up to n times	`repeat(10, 3) → 10 10 10`

Iterators terminating on the shortest input sequence:

Iterator	Arguments	Results	Example
`accumulate()`	p [,func]	p0, p0+p1, p0+p1+p2, …	`accumulate([1,2,3,4,5]) → 1 3 6 10 15`
`batched()`	p, n	(p0, p1, …, p_n-1), …	`batched('ABCDEFG', n=3) → ABC DEF G`

Iterator	Arguments	Results	Example
`chain()`	p, q, …	p0, p1, … plast, q0, q1, …	`chain('ABC', 'DEF') → A B C D E F`
`chain.from_iterable()`	iterable	p0, p1, … plast, q0, q1, …	`chain.from_iterable(['ABC', 'DEF']) → A B C D E F`
`compress()`	data, selectors	(d[0] if s[0]), (d[1] if s[1]), …	`compress('ABCDEF', [1,0,1,0,1,1]) → A C E F`
`dropwhile()`	predicate, seq	seq[n], seq[n+1], starting when predicate fails	`dropwhile(lambda x: x<5, [1,4,6,3,8]) → 6 3 8`
`filterfalse()`	predicate, seq	elements of seq where predicate(elem) fails	`filterfalse(lambda x: x<5, [1,4,6,3,8]) → 6 8`
`groupby()`	iterable[, key]	sub-iterators grouped by value of key(v)	`groupby(['A','B','DEF'], len) → (1, A B) (3, DEF)`
`islice()`	seq, [start,] stop [, step]	elements from seq[start:stop:step]	`islice('ABCDEFG', 2, None) → C D E F G`
`pairwise()`	iterable	(p[0], p[1]), (p[1], p[2])	`pairwise('ABCDEFG') → AB BC CD DE EF FG`
`starmap()`	func, seq	func(*seq[0]), func(*seq[1]), …	`starmap(pow, [(2,5), (3,2), (10,3)]) → 32 9 1000`
`takewhile()`	predicate, seq	seq[0], seq[1], until predicate fails	`takewhile(lambda x: x<5, [1,4,6,3,8]) → 1 4`
`tee()`	it, n	it1, it2, … itn splits one iterator into n	`tee('ABC', 2) → A B C, A B C`

Iterator	Arguments	Results	Example
[`zip_longest()`]	p, q, …	(p[0], q[0]), (p[1], q[1]), …	`zip_longest('ABCD', 'xy', fillvalue='-') → Ax By C- D-`

Combinatoric iterators:

Iterator	Arguments	Results
`product()`	p, q, … [repeat=1]	cartesian product, equivalent to a nested for-loop
`permutations()`	p[, r]	r-length tuples, all possible orderings, no repeated elements
`combinations()`	p, r	r-length tuples, in sorted order, no repeated elements
`combinations_with_replacement()`	p, r	r-length tuples, in sorted order, with repeated elements

Examples	Results
`product('ABCD', repeat=2)`	`AA AB AC AD BA BB BC BD CA CB CC CD DA DB DC DD`
`permutations('ABCD', 2)`	`AB AC AD BA BC BD CA CB CD DA DB DC`
`combinations('ABCD', 2)`	`AB AC AD BC BD CD`
`combinations_with_replacement('ABCD', 2)`	`AA AB AC AD BB BC BD CC CD DD`

Itertool Functions

The following functions all construct and return iterators. Some provide streams of infinite length, so they should only be accessed by functions or loops that truncate the stream.

itertools.accumulate(*iterable*[, *function*, *, *initial=None*])

Make an iterator that returns accumulated sums or accumulated results from other binary functions.

The *function* defaults to addition. The *function* should accept two arguments, an accumulated total and a value from the *iterable*.

If an *initial* value is provided, the accumulation will start with that value and the output will have one more element than the input iterable.

Roughly equivalent to:

```
def accumulate(iterable, function=operator.add, *, initial=None):
    'Return running totals'
    # accumulate([1,2,3,4,5]) → 1 3 6 10 15
    # accumulate([1,2,3,4,5], initial=100) → 100 101 103 106 110 115
    # accumulate([1,2,3,4,5], operator.mul) → 1 2 6 24 120

    iterator = iter(iterable)
    total = initial
    if initial is None:
        try:
            total = next(iterator)
        except StopIteration:
            return

    yield total
    for element in iterator:
        total = function(total, element)
        yield total
```

To compute a running minimum, set *function* to `min()`. For a running maximum, set *function* to `max()`. Or for a running product, set *function* to `operator.mul()`. To build an amortization table [https://www.ramseysolutions.com/real-estate/amortization-schedule], accumulate the interest and apply payments:

```
>>> data = [3, 4, 6, 2, 1, 9, 0, 7, 5, 8]
>>> list(accumulate(data, max))              # running maximum
[3, 4, 6, 6, 6, 9, 9, 9, 9, 9]
>>> list(accumulate(data, operator.mul))     # running product
[3, 12, 72, 144, 144, 1296, 0, 0, 0, 0]

# Amortize a 5% loan of 1000 with 10 annual payments of 90
>>> update = lambda balance, payment: round(balance * 1.05) - payment
>>> list(accumulate(repeat(90, 10), update, initial=1_000))
[1000, 960, 918, 874, 828, 779, 728, 674, 618, 559, 497]
```

See `functools.reduce()` for a similar function that returns only the final accumulated value.

Added in version 3.2.

Changed in version 3.3: Added the optional *function* parameter.

Changed in version 3.8: Added the optional *initial* parameter.

itertools.batched(*iterable*, *n*, ***, *strict=False*)

Batch data from the *iterable* into tuples of length *n*. The last batch may be shorter than *n*.

If *strict* is true, will raise a `ValueError` if the final batch is shorter than *n*.

Loops over the input iterable and accumulates data into tuples up to size *n*. The input is consumed lazily, just enough to fill a batch. The result is yielded as soon as the batch is full or when the input iterable is exhausted:

```
>>> flattened_data = ['roses', 'red', 'violets', 'blue', 'sugar', 'sweet']
>>> unflattened = list(batched(flattened_data, 2))
>>> unflattened
[('roses', 'red'), ('violets', 'blue'), ('sugar', 'sweet')]
```

Roughly equivalent to:

```
def batched(iterable, n, *, strict=False):
    # batched('ABCDEFG', 3) → ABC DEF G
    if n < 1:
        raise ValueError('n must be at least one')
    iterator = iter(iterable)
    while batch := tuple(islice(iterator, n)):
        if strict and len(batch) != n:
            raise ValueError('batched(): incomplete batch')
        yield batch
```

Added in version 3.12.

Changed in version 3.13: Added the *strict* option.

itertools.chain(**iterables*)

Make an iterator that returns elements from the first iterable until it is exhausted, then proceeds to the next iterable, until all of the iterables are exhausted. This combines multiple data sources into a single iterator. Roughly equivalent to:

```
def chain(*iterables):
    # chain('ABC', 'DEF') → A B C D E F
    for iterable in iterables:
        yield from iterable
```

classmethod chain.from_iterable(*iterable*)

Alternate constructor for `chain()`. Gets chained inputs from a single iterable argument that is evaluated lazily. Roughly equivalent to:

```
def from_iterable(iterables):
    # chain.from_iterable(['ABC', 'DEF']) → A B C D E F
    for iterable in iterables:
        yield from iterable
```

itertools.combinations(*iterable*, *r*)

Return *r* length subsequences of elements from the input *iterable*.

The output is a subsequence of `product()` keeping only entries that are subsequences of the *iterable*. The length of the output is given by `math.comb()` which computes `n! / r! / (n - r)!` when `0 ≤ r ≤ n` or zero when `r > n`.

The combination tuples are emitted in lexicographic order according to the order of the input *iterable*. If the input *iterable* is sorted, the output tuples will be produced in sorted order.

Elements are treated as unique based on their position, not on their value. If the input elements are unique, there will be no repeated values within each combination.

Roughly equivalent to:

```
def combinations(iterable, r):
    # combinations('ABCD', 2) → AB AC AD BC BD CD
    # combinations(range(4), 3) → 012 013 023 123

    pool = tuple(iterable)
    n = len(pool)
    if r > n:
        return
    indices = list(range(r))

    yield tuple(pool[i] for i in indices)
    while True:
        for i in reversed(range(r)):
            if indices[i] != i + n - r:
                break
        else:
            return
        indices[i] += 1
        for j in range(i+1, r):
            indices[j] = indices[j-1] + 1
        yield tuple(pool[i] for i in indices)
```

itertools.combinations_with_replacement(*iterable*, *r*)

Return *r* length subsequences of elements from the input *iterable* allowing individual elements to be repeated more than once.

The output is a subsequence of `product()` that keeps only entries that are subsequences (with possible repeated elements) of the *iterable*. The number of subsequence returned is `(n + r - 1)! / r! / (n - 1)!` when `n > 0`.

The combination tuples are emitted in lexicographic order according to the order of the input *iterable*. if the input *iterable* is sorted, the output tuples will be produced in sorted order.

Elements are treated as unique based on their position, not on their value. If the input elements are unique, the generated combinations will also be unique.

Roughly equivalent to:

```
def combinations_with_replacement(iterable, r):
    # combinations_with_replacement('ABC', 2) → AA AB AC BB BC CC

    pool = tuple(iterable)
    n = len(pool)
    if not n and r:
        return
    indices = [0] * r

    yield tuple(pool[i] for i in indices)
    while True:
        for i in reversed(range(r)):
            if indices[i] != n - 1:
                break
        else:
            return
        indices[i:] = [indices[i] + 1] * (r - i)
        yield tuple(pool[i] for i in indices)
```

Added in version 3.1.

itertools.compress(*data*, *selectors*)

Make an iterator that returns elements from *data* where the corresponding element in *selectors* is true. Stops when either the *data* or *selectors* iterables have been exhausted. Roughly equivalent to:

```
def compress(data, selectors):
    # compress('ABCDEF', [1,0,1,0,1,1]) → A C E F
    return (datum for datum, selector in zip(data, selectors) if selector)
```

Added in version 3.1.

itertools.count(*start=0*, *step=1*)

Make an iterator that returns evenly spaced values beginning with *start*. Can be used with `map()` to generate consecutive data points or with `zip()` to add sequence numbers. Roughly equivalent to:

```
def count(start=0, step=1):
    # count(10) → 10 11 12 13 14 ...
    # count(2.5, 0.5) → 2.5 3.0 3.5 ...
    n = start
    while True:
        yield n
        n += step
```

When counting with floating-point numbers, better accuracy can sometimes be achieved by substituting multiplicative code such as: `(start + step * i for i in count())`.

Changed in version 3.1: Added *step* argument and allowed non-integer arguments.

itertools.cycle(*iterable*)

Make an iterator returning elements from the *iterable* and saving a copy of each. When the iterable is exhausted, return elements from the saved copy. Repeats indefinitely. Roughly equivalent to:

```
def cycle(iterable):
    # cycle('ABCD') → A B C D A B C D A B C D ...

    saved = []
    for element in iterable:
        yield element
        saved.append(element)

    while saved:
        for element in saved:
            yield element
```

This itertool may require significant auxiliary storage (depending on the length of the iterable).

itertools.dropwhile(*predicate*, *iterable*)

Make an iterator that drops elements from the *iterable* while the *predicate* is true and afterwards returns every element. Roughly equivalent to:

```
def dropwhile(predicate, iterable):
    # dropwhile(lambda x: x<5, [1,4,6,3,8]) → 6 3 8

    iterator = iter(iterable)
    for x in iterator:
        if not predicate(x):
            yield x
            break

    for x in iterator:
        yield x
```

Note this does not produce *any* output until the predicate first becomes false, so this itertool may have a lengthy start-up time.

itertools.filterfalse(*predicate*, *iterable*)

Make an iterator that filters elements from the *iterable* returning only those for which the *predicate* returns a false value. If *predicate* is `None`, returns the items that are false. Roughly equivalent to:

```
def filterfalse(predicate, iterable):
    # filterfalse(lambda x: x<5, [1,4,6,3,8]) → 6 8
    if predicate is None:
        predicate = bool

    for x in iterable:
        if not predicate(x):
            yield x
```

itertools.groupby(*iterable*, *key=None*)

Make an iterator that returns consecutive keys and groups from the *iterable*. The *key* is a function computing a key value for each element. If not specified or is `None`, *key* defaults to an identity function and returns the element unchanged. Generally, the iterable needs to already be sorted on the same key function.

The operation of **`groupby()`** is similar to the `uniq` filter in Unix. It generates a break or new group every time the value of the key function changes (which is why it is usually necessary to have sorted the data using the same key function). That behavior differs from SQL's GROUP BY which aggregates common elements regardless of their input order.

The returned group is itself an iterator that shares the underlying iterable with **`groupby()`**. Because the source is shared, when the **`groupby()`** object is advanced, the previous group is no longer visible. So, if that data is needed later, it should be stored as a list:

```
groups = []
uniquekeys = []
data = sorted(data, key=keyfunc)
for k, g in groupby(data, keyfunc):
    groups.append(list(g))      # Store group iterator as a list
    uniquekeys.append(k)
```

`groupby()` is roughly equivalent to:

```
def groupby(iterable, key=None):
    # [k for k, g in groupby('AAAABBBCCDAABBB')] → A B C D A B
    # [list(g) for k, g in groupby('AAAABBBCCD')] → AAAA BBB CC D

    keyfunc = (lambda x: x) if key is None else key
    iterator = iter(iterable)
    exhausted = False

    def _grouper(target_key):
        nonlocal curr_value, curr_key, exhausted
        yield curr_value
        for curr_value in iterator:
            curr_key = keyfunc(curr_value)
            if curr_key != target_key:
                return
            yield curr_value
        exhausted = True

    try:
        curr_value = next(iterator)
        except StopIteration:
            return
        curr_key = keyfunc(curr_value)

        while not exhausted:
            target_key = curr_key
            curr_group = _grouper(target_key)
            yield curr_key, curr_group
            if curr_key == target_key:
                for _ in curr_group:
                    pass
```

itertools.islice(*iterable*, *stop*)

itertools.islice(*iterable*, *start*, *stop*[, *step*])

Make an iterator that returns selected elements from the iterable. Works like sequence slicing but does not support negative values for *start*, *stop*, or *step*.

If *start* is zero or `None`, iteration starts at zero. Otherwise, elements from the iterable are skipped until *start* is reached.

If *stop* is `None`, iteration continues until the input is exhausted, if at all. Otherwise, it stops at the specified position.

If *step* is `None`, the step defaults to one. Elements are returned consecutively unless *step* is set higher than one which results in items being skipped.

Roughly equivalent to:

```
def islice(iterable, *args):
    # islice('ABCDEFG', 2) → A B
    # islice('ABCDEFG', 2, 4) → C D
    # islice('ABCDEFG', 2, None) → C D E F G
    # islice('ABCDEFG', 0, None, 2) → A C E G

    s = slice(*args)
    start = 0 if s.start is None else s.start
    stop = s.stop
    step = 1 if s.step is None else s.step
    if start < 0 or (stop is not None and stop < 0) or step <= 0:
        raise ValueError

    indices = count() if stop is None else range(max(start, stop))
    next_i = start
    for i, element in zip(indices, iterable):
        if i == next_i:
            yield element
            next_i += step
```

If the input is an iterator, then fully consuming the *islice* advances the input iterator by `max(start, stop)` steps regardless of the *step* value.

itertools.pairwise(*iterable*)

Return successive overlapping pairs taken from the input *iterable*.

The number of 2-tuples in the output iterator will be one fewer than the number of inputs. It will be empty if the input iterable has fewer than two values.

Roughly equivalent to:

```
def pairwise(iterable):
    # pairwise('ABCDEFG') → AB BC CD DE EF FG

    iterator = iter(iterable)
    a = next(iterator, None)

    for b in iterator:
        yield a, b
        a = b
```

Added in version 3.10.

itertools.permutations(*iterable*, *r=None*)

Return successive *r* length permutations of elements [https://www.britannica.com/science/permutation] from the *iterable*.

If *r* is not specified or is `None`, then *r* defaults to the length of the *iterable* and all possible full-length permutations are generated.

The output is a subsequence of **`product()`** where entries with repeated elements have been filtered out. The length of the output is given by **`math.perm()`** which computes `n! / (n - r)!` when `0 ≤ r ≤ n` or zero when `r > n`.

The permutation tuples are emitted in lexicographic order according to the order of the input *iterable*. If the input *iterable* is sorted, the output tuples will be produced in sorted order.

Elements are treated as unique based on their position, not on their value. If the input elements are unique, there will be no repeated values within a permutation.

Roughly equivalent to:

```
def permutations(iterable, r=None):
    # permutations('ABCD', 2) → AB AC AD BA BC BD CA CB CD DA DB DC
    # permutations(range(3)) → 012 021 102 120 201 210

    pool = tuple(iterable)
    n = len(pool)
    r = n if r is None else r
    if r > n:
        return

    indices = list(range(n))
    cycles = list(range(n, n-r, -1))
    yield tuple(pool[i] for i in indices[:r])

    while n:
        for i in reversed(range(r)):
            cycles[i] -= 1
            if cycles[i] == 0:
                indices[i:] = indices[i+1:] + indices[i:i+1]
                cycles[i] = n - i
            else:
                j = cycles[i]
                indices[i], indices[-j] = indices[-j], indices[i]
                yield tuple(pool[i] for i in indices[:r])
                break
        else:
            return
```

itertools.product(**iterables*, *repeat=1*)

Cartesian product [https://en.wikipedia.org/wiki/Cartesian_product] of the input iterables.

Roughly equivalent to nested for-loops in a generator expression. For example, `product(A, B)` returns the same as `((x,y) for x in A for y in B)`.

The nested loops cycle like an odometer with the rightmost element advancing on every iteration. This pattern creates a lexicographic ordering so that if the input's iterables are sorted, the product tuples are emitted in sorted order.

To compute the product of an iterable with itself, specify the number of repetitions with the optional *repeat* keyword argument. For example, `product(A, repeat=4)` means the same as `product(A, A, A, A)`.

This function is roughly equivalent to the following code, except that the actual implementation does not build up intermediate results in memory:

```
def product(*iterables, repeat=1):
    # product('ABCD', 'xy') → Ax Ay Bx By Cx Cy Dx Dy
    # product(range(2), repeat=3) → 000 001 010 011 100 101 110 111

    if repeat < 0:
        raise ValueError('repeat argument cannot be negative')
    pools = [tuple(pool) for pool in iterables] * repeat

    result = [[]]
    for pool in pools:
        result = [x+[y] for x in result for y in pool]

    for prod in result:
        yield tuple(prod)
```

Before **`product()`** runs, it completely consumes the input iterables, keeping pools of values in memory to generate the products. Accordingly, it is only useful with finite inputs.

itertools.repeat(*object*[, *times*])

Make an iterator that returns *object* over and over again. Runs indefinitely unless the *times* argument is specified.

Roughly equivalent to:

```
def repeat(object, times=None):
    # repeat(10, 3) → 10 10 10
    if times is None:
        while True:
            yield object
    else:
        for i in range(times):
            yield object
```

A common use for *repeat* is to supply a stream of constant values to *map* or *zip*:

```
>>> list(map(pow, range(10), repeat(2)))
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

itertools.starmap(*function*, *iterable*)

Make an iterator that computes the *function* using arguments obtained from the *iterable*. Used instead of `map()` when argument parameters have already been “pre-zipped” into tuples.

The difference between `map()` and `starmap()` parallels the distinction between `function(a,b)` and `function(*c)`. Roughly equivalent to:

```
def starmap(function, iterable):
    # starmap(pow, [(2,5), (3,2), (10,3)]) → 32 9 1000
    for args in iterable:
        yield function(*args)
```

itertools.takewhile(*predicate*, *iterable*)

Make an iterator that returns elements from the *iterable* as long as the *predicate* is true. Roughly equivalent to:

```
def takewhile(predicate, iterable):
    # takewhile(lambda x: x<5, [1,4,6,3,8]) → 1 4
    for x in iterable:
        if not predicate(x):
            break
        yield x
```

Note, the element that first fails the predicate condition is consumed from the input iterator and there is no way to access it. This could be an issue if an application wants to further consume the input iterator after *takewhile* has been run to exhaustion. To work around this problem, consider using more-itertools before_and_after() [https://more-itertools.readthedocs.io/en/stable/api.html#more_itertools.before_and_after] instead.

itertools.tee(*iterable*, *n=2*)

Return *n* independent iterators from a single iterable.

Roughly equivalent to:

```
def tee(iterable, n=2):
    if n < 0:
        raise ValueError
    if n == 0:
        return ()
    iterator = _tee(iterable)
    result = [iterator]
    for _ in range(n - 1):
        result.append(_tee(iterator))
    return tuple(result)

class _tee:

    def __init__(self, iterable):
        it = iter(iterable)
        if isinstance(it, _tee):
            self.iterator = it.iterator
            self.link = it.link
        else:
            self.iterator = it
            self.link = [None, None]

    def __iter__(self):
        return self

    def __next__(self):
        link = self.link
        if link[1] is None:
            link[0] = next(self.iterator)
            link[1] = [None, None]
        value, self.link = link
        return value
```

When the input *iterable* is already a tee iterator object, all members of the return tuple are constructed as if they had been produced by the upstream `tee()` call. This “flattening step” allows nested `tee()` calls to share the same underlying data chain and to have a single update step rather than a chain of calls.

The flattening property makes tee iterators efficiently peekable:

```
def lookahead(tee_iterator):
     "Return the next value without moving the input forward"
     [forked_iterator] = tee(tee_iterator, 1)
     return next(forked_iterator)

>>> iterator = iter('abcdef')
>>> [iterator] = tee(iterator, 1)   # Make the input peekable
>>> next(iterator)                  # Move the iterator forward
'a'
>>> lookahead(iterator)             # Check next value
'b'
>>> next(iterator)                  # Continue moving forward
'b'
```

`tee` iterators are not threadsafe. A `RuntimeError` may be raised when simultaneously using iterators returned by the same `tee()` call, even if the original *iterable* is threadsafe.

This itertool may require significant auxiliary storage (depending on how much temporary data needs to be stored). In general, if one iterator uses most or all of the data before another iterator starts, it is faster to use `list()` instead of `tee()`.

itertools.zip_longest(**iterables*, *fillvalue=None*)

Make an iterator that aggregates elements from each of the *iterables*.

If the iterables are of uneven length, missing values are filled-in with *fillvalue*. If not specified, *fillvalue* defaults to `None`.

Iteration continues until the longest iterable is exhausted.

Roughly equivalent to:

```
def zip_longest(*iterables, fillvalue=None):
    # zip_longest('ABCD', 'xy', fillvalue='-') → Ax By C- D-

    iterators = list(map(iter, iterables))
    num_active = len(iterators)
    if not num_active:
        return

    while True:
        values = []
        for i, iterator in enumerate(iterators):
            try:
                value = next(iterator)
            except StopIteration:
                num_active -= 1
                if not num_active:
                    return
                iterators[i] = repeat(fillvalue)
                value = fillvalue
            values.append(value)
        yield tuple(values)
```

If one of the iterables is potentially infinite, then the `zip_longest()` function should be wrapped with something that limits the number of calls (for example `islice()` or `takewhile()`).

Itertools Recipes

This section shows recipes for creating an extended toolset using the existing itertools as building blocks.

The primary purpose of the itertools recipes is educational. The recipes show various ways of thinking about individual tools — for example, that `chain.from_iterable` is related to the concept of flattening. The recipes also give ideas about ways that the tools can be combined — for example, how `starmap()`

and `repeat()` can work together. The recipes also show patterns for using itertools with the **`operator`** and **`collections`** modules as well as with the built-in itertools such as `map()`, `filter()`, `reversed()`, and `enumerate()`.

A secondary purpose of the recipes is to serve as an incubator. The `accumulate()`, `compress()`, and `pairwise()` itertools started out as recipes. Currently, the `sliding_window()`, `derangements()`, and `sieve()` recipes are being tested to see whether they prove their worth.

Substantially all of these recipes and many, many others can be installed from the more-itertools [https://pypi.org/project/more-itertools/] project found on the Python Package Index:

```
python -m pip install more-itertools
```

Many of the recipes offer the same high performance as the underlying toolset. Superior memory performance is kept by processing elements one at a time rather than bringing the whole iterable into memory all at once. Code volume is kept small by linking the tools together in a functional style [https://www.cs.kent.ac.uk/people/staff/dat/miranda/whyfp90.pdf]. High speed is retained by preferring “vectorized” building blocks over the use of for-loops and generators which incur interpreter overhead.

```
from itertools import (accumulate, batched, chain, combinations, compress,
     count, cycle, filterfalse, groupby, islice, permutations, product,
     repeat, starmap, tee, zip_longest)
from collections import Counter, deque
from contextlib import suppress
from functools import reduce
from math import comb, isqrt, prod, sumprod
from operator import getitem, is_not, itemgetter, mul, neg

# ==== Basic one liners ====

def take(n, iterable):
    "Return first n items of the iterable as a list."
    return list(islice(iterable, n))

def prepend(value, iterable):
    "Prepend a single value in front of an iterable."
    # prepend(1, [2, 3, 4]) → 1 2 3 4
    return chain([value], iterable)

def tabulate(function, start=0):
    "Return function(0), function(1), ..."
    return map(function, count(start))

def repeatfunc(function, times=None, *args):
    "Repeat calls to a function with specified arguments."
    if times is None:
        return starmap(function, repeat(args))
    return starmap(function, repeat(args, times))

def flatten(list_of_lists):
    "Flatten one level of nesting."
    return chain.from_iterable(list_of_lists)

def ncycles(iterable, n):
    "Returns the sequence elements n times."
    return chain.from_iterable(repeat(tuple(iterable), n))

def loops(n):
    "Loop n times. Like range(n) but without creating integers."
    # for _ in loops(100): ...
    return repeat(None, n)

def tail(n, iterable):
    "Return an iterator over the last n items."
    # tail(3, 'ABCDEFG') → E F G
    return iter(deque(iterable, maxlen=n))

def consume(iterator, n=None):
    "Advance the iterator n-steps ahead. If n is None, consume entirely."
    # Use functions that consume iterators at C speed.
    if n is None:
        deque(iterator, maxlen=0)
    else:
        next(islice(iterator, n, n), None)

def nth(iterable, n, default=None):
    "Returns the nth item or a default value."
    return next(islice(iterable, n, None), default)

def quantify(iterable, predicate=bool):
    "Given a predicate that returns True or False, count the True results."
    return sum(map(predicate, iterable))

def first_true(iterable, default=False, predicate=None):
    "Returns the first true value or the *default* if there is no true value."
    # first_true([a, b, c], x) → a or b or c or x
    # first_true([a, b], x, f) → a if f(a) else b if f(b) else x
    return next(filter(predicate, iterable), default)

def all_equal(iterable, key=None):
    "Returns True if all the elements are equal to each other."
    # all_equal('4٤௪౪໔', key=int) → True
    return len(take(2, groupby(iterable, key))) <= 1

# ==== Data pipelines ====

def unique_justseen(iterable, key=None):
    "Yield unique elements, preserving order. Remember only the element just seen."
    # unique_justseen('AAAABBBCCDAABBB') → A B C D A B
    # unique_justseen('ABBcCAD', str.casefold) → A B c A D
    if key is None:
        return map(itemgetter(0), groupby(iterable))
    return map(next, map(itemgetter(1), groupby(iterable, key)))

def unique_everseen(iterable, key=None):
    "Yield unique elements, preserving order. Remember all elements ever seen."
    # unique_everseen('AAAABBBCCDAABBB') → A B C D
    # unique_everseen('ABBcCAD', str.casefold) → A B c D
    seen = set()
    if key is None:
        for element in filterfalse(seen.__contains__, iterable):
            seen.add(element)
            yield element
    else:
        for element in iterable:
            k = key(element)
            if k not in seen:
                seen.add(k)
                yield element

def unique(iterable, key=None, reverse=False):
   "Yield unique elements in sorted order. Supports unhashable inputs."
   # unique([[1, 2], [3, 4], [1, 2]]) → [1, 2] [3, 4]
   sequenced = sorted(iterable, key=key, reverse=reverse)
   return unique_justseen(sequenced, key=key)

def sliding_window(iterable, n):
    "Collect data into overlapping fixed-length chunks or blocks."
    # sliding_window('ABCDEFG', 3) → ABC BCD CDE DEF EFG
    iterator = iter(iterable)
    window = deque(islice(iterator, n - 1), maxlen=n)
    for x in iterator:
        window.append(x)
        yield tuple(window)

def grouper(iterable, n, *, incomplete='fill', fillvalue=None):
    "Collect data into non-overlapping fixed-length chunks or blocks."
    # grouper('ABCDEFG', 3, fillvalue='x')       → ABC DEF Gxx
    # grouper('ABCDEFG', 3, incomplete='strict') → ABC DEF ValueError
    # grouper('ABCDEFG', 3, incomplete='ignore') → ABC DEF
    iterators = [iter(iterable)] * n
    match incomplete:
        case 'fill':
            return zip_longest(*iterators, fillvalue=fillvalue)
        case 'strict':
            return zip(*iterators, strict=True)
        case 'ignore':
            return zip(*iterators)
        case _:
            raise ValueError('Expected fill, strict, or ignore')

def roundrobin(*iterables):
    "Visit input iterables in a cycle until each is exhausted."
    # roundrobin('ABC', 'D', 'EF') → A D E B F C
    # Algorithm credited to George Sakkis
    iterators = map(iter, iterables)
    for num_active in range(len(iterables), 0, -1):
        iterators = cycle(islice(iterators, num_active))
        yield from map(next, iterators)

def subslices(seq):
    "Return all contiguous non-empty subslices of a sequence."
    # subslices('ABCD') → A AB ABC ABCD B BC BCD C CD D
    slices = starmap(slice, combinations(range(len(seq) + 1), 2))
    return map(getitem, repeat(seq), slices)
```

```
def derangements(iterable, r=None):
    "Produce r length permutations without fixed points."
    # derangements('ABCD') → BADC BCDA BDAC CADB CDAB CDBA DABC DCAB DCBA
    # Algorithm credited to Stefan Pochmann
    seq = tuple(iterable)
    pos = tuple(range(len(seq)))
    have_moved = map(map, repeat(is_not), repeat(pos), permutations(pos, r=r))
    valid_derangements = map(all, have_moved)
    return compress(permutations(seq, r=r), valid_derangements)

def iter_index(iterable, value, start=0, stop=None):
    "Return indices where a value occurs in a sequence or iterable."
    # iter_index('AABCADEAF', 'A') → 0 1 4 7
    seq_index = getattr(iterable, 'index', None)
    if seq_index is None:
        iterator = islice(iterable, start, stop)
        for i, element in enumerate(iterator, start):
            if element is value or element == value:
                yield i
    else:
        stop = len(iterable) if stop is None else stop
        i = start
        with suppress(ValueError):
            while True:
                yield (i := seq_index(value, i, stop))
                i += 1

def iter_except(function, exception, first=None):
    "Convert a call-until-exception interface to an iterator interface."
    # iter_except(d.popitem, KeyError) → non-blocking dictionary iterator
    with suppress(exception):
        if first is not None:
            yield first()
        while True:
            yield function()

# ==== Mathematical operations ====

def multinomial(*counts):
    "Number of distinct arrangements of a multiset."
    # Counter('abracadabra').values() → 5 2 2 1 1
    # multinomial(5, 2, 2, 1, 1) → 83160
    return prod(map(comb, accumulate(counts), counts))

def powerset(iterable):
    "Subsequences of the iterable from shortest to longest."
    # powerset([1,2,3]) → () (1,) (2,) (3,) (1,2) (1,3) (2,3) (1,2,3)
    s = list(iterable)
    return chain.from_iterable(combinations(s, r) for r in range(len(s)+1))

def sum_of_squares(iterable):
    "Add up the squares of the input values."
    # sum_of_squares([10, 20, 30]) → 1400
    return sumprod(*tee(iterable))

# ==== Matrix operations ====

def reshape(matrix, columns):
    "Reshape a 2-D matrix to have a given number of columns."
    # reshape([(0, 1), (2, 3), (4, 5)], 3) →  (0, 1, 2) (3, 4, 5)
    return batched(chain.from_iterable(matrix), columns, strict=True)

def transpose(matrix):
    "Swap the rows and columns of a 2-D matrix."
    # transpose([(1, 2, 3), (11, 22, 33)]) → (1, 11) (2, 22) (3, 33)
    return zip(*matrix, strict=True)

def matmul(m1, m2):
    "Multiply two matrices."
    # matmul([(7, 5), (3, 5)], [(2, 5), (7, 9)]) → (49, 80) (41, 60)
    n = len(m2[0])
    return batched(starmap(sumprod, product(m1, transpose(m2))), n)

# ==== Polynomial arithmetic ====

def convolve(signal, kernel):
    """Discrete linear convolution of two iterables.
    Equivalent to polynomial multiplication.

    Convolutions are mathematically commutative; however, the inputs are
    evaluated differently.  The signal is consumed lazily and can be
    infinite. The kernel is fully consumed before the calculations begin.

    Article:  https://betterexplained.com/articles/intuitive-convolution/
    Video:    https://www.youtube.com/watch?v=KuXjwB4LzSA
    """
    # convolve([1, -1, -20], [1, -3]) → 1 -4 -17 60
    # convolve(data, [0.25, 0.25, 0.25, 0.25]) → Moving average (blur)
    # convolve(data, [1/2, 0, -1/2]) → 1st derivative estimate
    # convolve(data, [1, -2, 1]) → 2nd derivative estimate
    kernel = tuple(kernel)[::-1]
    n = len(kernel)
    padded_signal = chain(repeat(0, n-1), signal, repeat(0, n-1))
    windowed_signal = sliding_window(padded_signal, n)
    return map(sumprod, repeat(kernel), windowed_signal)

def polynomial_from_roots(roots):
    """Compute a polynomial's coefficients from its roots.

       (x - 5) (x + 4) (x - 3)  expands to:   x³ -4x² -17x + 60
    """
    # polynomial_from_roots([5, -4, 3]) → [1, -4, -17, 60]
    factors = zip(repeat(1), map(neg, roots))
    return list(reduce(convolve, factors, [1]))

def polynomial_eval(coefficients, x):
    """Evaluate a polynomial at a specific value.

    Computes with better numeric stability than Horner's method.
    """
    # Evaluate x³ -4x² -17x + 60 at x = 5
    # polynomial_eval([1, -4, -17, 60], x=5) → 0
    n = len(coefficients)
    if not n:
        return type(x)(0)
    powers = map(pow, repeat(x), reversed(range(n)))
    return sumprod(coefficients, powers)

def polynomial_derivative(coefficients):
    """Compute the first derivative of a polynomial.

       f(x)  =  x³ -4x² -17x + 60
       f'(x) = 3x² -8x  -17
    """
    # polynomial_derivative([1, -4, -17, 60]) → [3, -8, -17]
    n = len(coefficients)
    powers = reversed(range(1, n))
    return list(map(mul, coefficients, powers))

# ==== Number theory ====

def sieve(n):
    "Primes less than n."
    # sieve(30) → 2 3 5 7 11 13 17 19 23 29
    if n > 2:
        yield 2
    data = bytearray((0, 1)) * (n // 2)
    for p in iter_index(data, 1, start=3, stop=isqrt(n) + 1):
        data[p*p : n : p+p] = bytes(len(range(p*p, n, p+p)))
    yield from iter_index(data, 1, start=3)

def factor(n):
    "Prime factors of n."
    # factor(99) → 3 3 11
    # factor(1_000_000_000_000_007) → 47 59 360620266859
    # factor(1_000_000_000_000_403) → 1000000000000403
    for prime in sieve(isqrt(n) + 1):
        while not n % prime:
            yield prime
            n //= prime
            if n == 1:
                return
    if n > 1:
        yield n

def is_prime(n):
    "Return True if n is prime."
    # is_prime(1_000_000_000_000_403) → True
    return n > 1 and next(factor(n)) == n

def totient(n):
    "Count of natural numbers up to n that are coprime to n."
    # https://mathworld.wolfram.com/TotientFunction.html
    # totient(12) → 4 because len([1, 5, 7, 11]) == 4
    for prime in set(factor(n)):
        n -= n // prime
    return n
```

`functools` — Higher-order functions and operations on callable objects

Source code: Lib/functools.py
[https://github.com/python/cpython/tree/3.14/Lib/functools.py]

The **`functools`** module is for higher-order functions: functions that act on or return other functions. In general, any callable object can be treated as a function for the purposes of this module.

The **`functools`** module defines the following functions:

@functools.cache(*user_function*)

Simple lightweight unbounded function cache. Sometimes called “memoize” [https://en.wikipedia.org/wiki/Memoization].

Returns the same as `lru_cache(maxsize=None)`, creating a thin wrapper around a dictionary lookup for the function arguments. Because it never needs to evict old values, this is smaller and faster than **`lru_cache()`** with a size limit.

For example:

```
@cache
def factorial(n):
    return n * factorial(n-1) if n else 1

>>> factorial(10)   # no previously cached result, makes 11
```

```
recursive calls
3628800
>>> factorial(5)    # no new calls, just returns the cached
result
120
>>> factorial(12)   # two new recursive calls, factorial(10)
is cached
479001600
```

The cache is threadsafe so that the wrapped function can be used in multiple threads. This means that the underlying data structure will remain coherent during concurrent updates.

It is possible for the wrapped function to be called more than once if another thread makes an additional call before the initial call has been completed and cached.

Added in version 3.9.

@functools.cached_property(*func*)

Transform a method of a class into a property whose value is computed once and then cached as a normal attribute for the life of the instance. Similar to `property()`, with the addition of caching. Useful for expensive computed properties of instances that are otherwise effectively immutable.

Example:

```
class DataSet:

    def __init__(self, sequence_of_numbers):
        self._data = tuple(sequence_of_numbers)

    @cached_property
    def stdev(self):
        return statistics.stdev(self._data)
```

The mechanics of `cached_property()` are somewhat different from `property()`. A regular property blocks attribute writes unless a setter is defined. In contrast, a *cached_property* allows writes.

The *cached_property* decorator only runs on lookups and only when an attribute of the same name doesn’t exist. When it does run, the *cached_property* writes to the attribute with the same name. Subsequent attribute reads and writes take precedence over the *cached_property* method and it works like a normal attribute.

The cached value can be cleared by deleting the attribute. This allows the *cached_property* method to run again.

The *cached_property* does not prevent a possible race condition in multi-threaded usage. The getter function could run more than once on the same instance, with the latest run setting the cached value. If the cached property is idempotent or otherwise not harmful to run more than once on an instance, this is fine. If synchronization is needed, implement the necessary locking inside the decorated getter function or around the cached property access.

Note, this decorator interferes with the operation of **PEP 412** [https://peps.python.org/pep-0412/] key-sharing dictionaries. This means that instance dictionaries can take more space than usual.

Also, this decorator requires that the `__dict__` attribute on each instance be a mutable mapping. This means it will not work with some types, such as metaclasses (since the `__dict__` attributes on type instances are read-only proxies for the class namespace), and those that specify `__slots__` without including `__dict__` as one of the defined slots (as such classes don't provide a `__dict__` attribute at all).

If a mutable mapping is not available or if space-efficient key sharing is desired, an effect similar to **`cached_property()`** can also be achieved by stacking **`property()`** on top of **`lru_cache()`**. See How do I cache method calls? for more details on how this differs from **`cached_property()`**.

Added in version 3.8.

Changed in version 3.12: Prior to Python 3.12, `cached_property` included an undocumented lock to ensure that in multi-threaded usage the getter function was guaranteed to run only once per instance. However, the lock was per-property, not per-instance, which could result in unacceptably high lock contention. In Python 3.12+ this locking is removed.

functools.cmp_to_key(*func*)

Transform an old-style comparison function to a key function. Used with tools that accept key functions (such as **`sorted()`**, **`min()`**, **`max()`**, **`heapq.nlargest()`**, **`heapq.nsmallest()`**, **`itertools.groupby()`**). This function is primarily used as a transition tool for programs being converted from Python 2 which supported the use of comparison functions.

A comparison function is any callable that accepts two arguments, compares them, and returns a negative number for less-than, zero for equality, or a positive number for greater-than. A key function is a callable that accepts one argument and returns another value to be used as the sort key.

Example:

```
sorted(iterable, key=cmp_to_key(locale.strcoll))  # locale-
aware sort order
```

For sorting examples and a brief sorting tutorial, see Sorting Techniques.

Added in version 3.2.

@functools.lru_cache(*user_function*)
@functools.lru_cache(*maxsize=128*, *typed=False*)

Decorator to wrap a function with a memoizing callable that saves up to the *maxsize* most recent calls. It can save time when an expensive or I/O bound function is periodically called with the same arguments.

The cache is threadsafe so that the wrapped function can be used in multiple threads. This means that the underlying data structure will remain coherent during concurrent updates.

It is possible for the wrapped function to be called more than once if another thread makes an additional call before the initial call has been completed and cached.

Since a dictionary is used to cache results, the positional and keyword arguments to the function must be hashable.

Distinct argument patterns may be considered to be distinct calls with separate cache entries. For example, `f(a=1, b=2)` and `f(b=2, a=1)` differ in their keyword argument order and may have two separate cache entries.

If *user_function* is specified, it must be a callable. This allows the *lru_cache* decorator to be applied directly to a user function, leaving the *maxsize* at its default value of 128:

```
@lru_cache
def count_vowels(sentence):
    return sum(sentence.count(vowel) for vowel in
'AEIOUaeiou')
```

If *maxsize* is set to `None`, the LRU feature is disabled and the cache can grow without bound.

If *typed* is set to true, function arguments of different types will be cached separately. If *typed* is false, the implementation will usually regard them as equivalent calls and only cache a single result. (Some types such as *str* and *int* may be cached separately even when *typed* is false.)

Note, type specificity applies only to the function's immediate arguments rather than their contents. The scalar arguments, `Decimal(42)` and `Fraction(42)` are be treated as distinct calls with distinct results. In contrast, the tuple arguments `('answer', Decimal(42))` and `('answer', Fraction(42))` are treated as equivalent.

The wrapped function is instrumented with a **`cache_parameters()`** function that returns a new **`dict`** showing the values for *maxsize* and *typed*. This is for information purposes only. Mutating the values has no effect.

To help measure the effectiveness of the cache and tune the *maxsize* parameter, the wrapped function is instrumented with a **`cache_info()`** function that returns a named tuple showing *hits*, *misses*, *maxsize* and *currsize*.

The decorator also provides a **`cache_clear()`** function for clearing or invalidating the cache.

The original underlying function is accessible through the **`__wrapped__`** attribute. This is useful for introspection, for bypassing the cache, or for rewrapping the function with a different cache.

The cache keeps references to the arguments and return values until they age out of the cache or until the cache is cleared.

If a method is cached, the `self` instance argument is included in the cache. See How do I cache method calls?

An LRU (least recently used) cache [https://en.wikipedia.org/wiki/Cache_replacement_policies#Least_Recently_Used_(LRU)] works best when the most recent calls are the best predictors of upcoming calls (for example, the most popular articles on a news server tend to change each day). The cache's size limit assures that the cache does not grow without bound on long-running processes such as web servers.

In general, the LRU cache should only be used when you want to reuse previously computed values. Accordingly, it doesn't make sense to cache functions with side-effects, functions that need to create distinct mutable objects on each call (such as generators and async functions), or impure functions such as time() or random().

Example of an LRU cache for static web content:

```
@lru_cache(maxsize=32)
def get_pep(num):
    'Retrieve text of a Python Enhancement Proposal'
    resource = f'https://peps.python.org/pep-{num:04d}'
    try:
        with urllib.request.urlopen(resource) as s:
            return s.read()
    except urllib.error.HTTPError:
        return 'Not Found'

>>> for n in 8, 290, 308, 320, 8, 218, 320, 279, 289, 320,
9991:
...     pep = get_pep(n)
...     print(n, len(pep))

>>> get_pep.cache_info()
CacheInfo(hits=3, misses=8, maxsize=32, currsize=8)
```

Example of efficiently computing Fibonacci numbers [https://en.wikipedia.org/wiki/Fibonacci_number] using a cache to implement a dynamic programming [https://en.wikipedia.org/wiki/Dynamic_programming] technique:

```
@lru_cache(maxsize=None)
def fib(n):
    if n < 2:
        return n
    return fib(n-1) + fib(n-2)

>>> [fib(n) for n in range(16)]
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377,
610]

>>> fib.cache_info()
CacheInfo(hits=28, misses=16, maxsize=None, currsize=16)
```

Added in version 3.2.

Changed in version 3.3: Added the *typed* option.

Changed in version 3.8: Added the *user_function* option.

Changed in version 3.9: Added the function **`cache_parameters()`**

@functools.total_ordering

Given a class defining one or more rich comparison ordering methods, this class decorator supplies the rest. This simplifies the effort involved in specifying all of the possible rich comparison operations:

The class must define one of `__lt__()`, `__le__()`, `__gt__()`, or `__ge__()`. In addition, the class should supply an `__eq__()` method.

For example:

```
@total_ordering
class Student:
    def _is_valid_operand(self, other):
        return (hasattr(other, "lastname") and
                hasattr(other, "firstname"))
    def __eq__(self, other):
        if not self._is_valid_operand(other):
            return NotImplemented
        return ((self.lastname.lower(),
self.firstname.lower()) ==
                (other.lastname.lower(),
other.firstname.lower()))
    def __lt__(self, other):
        if not self._is_valid_operand(other):
            return NotImplemented
        return ((self.lastname.lower(),
self.firstname.lower()) <
                (other.lastname.lower(),
other.firstname.lower()))
```

Note

While this decorator makes it easy to create well behaved totally ordered types, it *does* come at the cost of slower execution and more complex stack traces for the derived comparison methods. If performance benchmarking indicates this is a bottleneck for a given application, implementing all six rich comparison methods instead is likely to provide an easy speed boost.

Note

This decorator makes no attempt to override methods that have been declared in the class *or its superclasses*. Meaning that if a superclass defines a comparison operator, *total_ordering* will not implement it again, even if the original method is abstract.

Added in version 3.2.

Changed in version 3.4: Returning `NotImplemented` from the underlying comparison function for unrecognised types is now supported.

functools.Placeholder

A singleton object used as a sentinel to reserve a place for positional arguments when calling **`partial()`** and **`partialmethod()`**.

Added in version 3.14.

functools.partial(*func*, /, **args*, ***keywords*)

Return a new partial object which when called will behave like *func* called with the positional arguments *args* and keyword arguments *keywords*. If more arguments are supplied to the call, they are appended to *args*. If additional keyword arguments are supplied, they extend and override *keywords*. Roughly equivalent to:

```
def partial(func, /, *args, **keywords):
    def newfunc(*more_args, **more_keywords):
        return func(*args, *more_args, **(keywords |
more_keywords))
```

```
    newfunc.func = func
    newfunc.args = args
    newfunc.keywords = keywords
    return newfunc
```

The **`partial()`** function is used for partial function application which “freezes” some portion of a function’s arguments and/or keywords resulting in a new object with a simplified signature. For example, **`partial()`** can be used to create a callable that behaves like the **`int()`** function where the *base* argument defaults to `2`:

```
>>> basetwo = partial(int, base=2)
>>> basetwo.__doc__ = 'Convert base 2 string to an int.'
>>> basetwo('10010')
18
```

If **`Placeholder`** sentinels are present in *args*, they will be filled first when **`partial()`** is called. This makes it possible to pre-fill any positional argument with a call to **`partial()`**; without **`Placeholder`**, only the chosen number of leading positional arguments can be pre-filled.

If any **`Placeholder`** sentinels are present, all must be filled at call time:

```
>>> say_to_world = partial(print, Placeholder, Placeholder,
"world!")
>>> say_to_world('Hello', 'dear')
Hello dear world!
```

Calling `say_to_world('Hello')` raises a **`TypeError`**, because only one positional argument is provided, but there are two placeholders that must be filled in.

If **`partial()`** is applied to an existing **`partial()`** object, **`Placeholder`** sentinels of the input object are filled in with new positional arguments. A placeholder can be retained by inserting a new **`Placeholder`** sentinel to the place held by a previous **`Placeholder`**:

```
>>> from functools import partial, Placeholder as _
>>> remove = partial(str.replace, _, _, '')
>>> message = 'Hello, dear dear world!'
>>> remove(message, ' dear')
'Hello, world!'
>>> remove_dear = partial(remove, _, ' dear')
>>> remove_dear(message)
'Hello, world!'
>>> remove_first_dear = partial(remove_dear, _, 1)
>>> remove_first_dear(message)
'Hello, dear world!'
```

`Placeholder` cannot be passed to **`partial()`** as a keyword argument.

Changed in version 3.14: Added support for **`Placeholder`** in positional arguments.

class functools.partialmethod(*func*, */*, **args*, ***keywords*)

Return a new **`partialmethod`** descriptor which behaves like **`partial`** except that it is designed to be used as a method definition rather than being directly callable.

func must be a descriptor or a callable (objects which are both, like normal functions, are handled as descriptors).

When *func* is a descriptor (such as a normal Python function, **`classmethod()`**, **`staticmethod()`**, **`abstractmethod()`** or another instance of **`partialmethod`**), calls to `__get__` are delegated to the underlying descriptor, and an appropriate partial object returned as the result.

When *func* is a non-descriptor callable, an appropriate bound method is created dynamically. This behaves like a normal Python function when used as a method: the *self* argument will be inserted as the first positional argument, even before the *args* and *keywords* supplied to the **`partialmethod`** constructor.

Example:

```
>>> class Cell:
...     def __init__(self):
...         self._alive = False
...     @property
...     def alive(self):
...         return self._alive
...     def set_state(self, state):
...         self._alive = bool(state)
...     set_alive = partialmethod(set_state, True)
...     set_dead = partialmethod(set_state, False)
...
>>> c = Cell()
>>> c.alive
False
>>> c.set_alive()
>>> c.alive
True
```

Added in version 3.4.

functools.reduce(*function*, *iterable*, */*[, *initial*])

Apply *function* of two arguments cumulatively to the items of *iterable*, from left to right, so as to reduce the iterable to a single value. For example, `reduce(lambda x, y: x+y, [1, 2, 3, 4, 5])` calculates `((((1+2)+3)+4)+5)`. The left argument, *x*, is the accumulated value and the right argument, *y*, is the update value from the *iterable*. If the optional *initial* is present, it is placed before the items of the iterable in the calculation, and serves as a default when the iterable is empty. If *initial* is not given and *iterable* contains only one item, the first item is returned.

Roughly equivalent to:

```
initial_missing = object()

def reduce(function, iterable, /, initial=initial_missing):
    it = iter(iterable)
    if initial is initial_missing:
        value = next(it)
    else:
        value = initial
    for element in it:
        value = function(value, element)
    return value
```

See **`itertools.accumulate()`** for an iterator that yields all intermediate values.

Changed in version 3.14: initial is now supported as a keyword argument.

@functools.singledispatch

Transform a function into a single-dispatch generic function.

To define a generic function, decorate it with the `@singledispatch` decorator. When defining a function using `@singledispatch`, note that the dispatch happens on the type of the first argument:

```
>>> from functools import singledispatch
>>> @singledispatch
... def fun(arg, verbose=False):
...     if verbose:
...         print("Let me just say,", end=" ")
...     print(arg)
```

To add overloaded implementations to the function, use the **`register()`** attribute of the generic function, which can be used as a decorator. For functions annotated with types, the decorator will infer the type of the first argument automatically:

```
>>> @fun.register
... def _(arg: int, verbose=False):
...     if verbose:
...         print("Strength in numbers, eh?", end=" ")
...     print(arg)
...
>>> @fun.register
... def _(arg: list, verbose=False):
...     if verbose:
...         print("Enumerate this:")
...     for i, elem in enumerate(arg):
...         print(i, elem)
```

`typing.Union` can also be used:

```
>>> @fun.register
... def _(arg: int | float, verbose=False):
...     if verbose:
...         print("Strength in numbers, eh?", end=" ")
...     print(arg)
...
>>> from typing import Union
>>> @fun.register
... def _(arg: Union[list, set], verbose=False):
...     if verbose:
...         print("Enumerate this:")
...     for i, elem in enumerate(arg):
...         print(i, elem)
...
```

For code which doesn’t use type annotations, the appropriate type argument can be passed explicitly to the decorator itself:

```
>>> @fun.register(complex)
... def _(arg, verbose=False):
...     if verbose:
...         print("Better than complicated.", end=" ")
...     print(arg.real, arg.imag)
...
```

For code that dispatches on a collections type (e.g., `list`), but wants to typehint the items of the collection (e.g., `list[int]`), the dispatch type should be passed explicitly to the decorator itself with the typehint going into the function definition:

```
>>> @fun.register(list)
... def _(arg: list[int], verbose=False):
...     if verbose:
...         print("Enumerate this:")
...     for i, elem in enumerate(arg):
...         print(i, elem)
```

Note

At runtime the function will dispatch on an instance of a list regardless of the type contained within the list i.e. `[1,2,3]` will be dispatched the same as `["foo", "bar", "baz"]`. The annotation provided in this example is for static type checkers only and has no runtime impact.

To enable registering lambdas and pre-existing functions, the **`register()`** attribute can also be used in a functional form:

```
>>> def nothing(arg, verbose=False):
...     print("Nothing.")
...
>>> fun.register(type(None), nothing)
```

The **`register()`** attribute returns the undecorated function. This enables decorator stacking, **`pickling`**, and the creation of unit tests for each variant independently:

```
>>> @fun.register(float)
... @fun.register(Decimal)
... def fun_num(arg, verbose=False):
...     if verbose:
...         print("Half of your number:", end=" ")
...     print(arg / 2)
...
>>> fun_num is fun
False
```

When called, the generic function dispatches on the type of the first argument:

```
>>> fun("Hello, world.")
Hello, world.
>>> fun("test.", verbose=True)
Let me just say, test.
>>> fun(42, verbose=True)
Strength in numbers, eh? 42
>>> fun(['spam', 'spam', 'eggs', 'spam'], verbose=True)
Enumerate this:
0 spam
1 spam
2 eggs
3 spam
>>> fun(None)
Nothing.
>>> fun(1.23)
0.615
```

Where there is no registered implementation for a specific type, its method resolution order is used to find a more generic implementation. The original function decorated with `@singledispatch` is registered for the base **`object`** type, which means it is used if no better implementation is found.

If an implementation is registered to an abstract base class, virtual subclasses of the base class will be dispatched to that implementation:

```
>>> from collections.abc import Mapping
>>> @fun.register
... def _(arg: Mapping, verbose=False):
...     if verbose:
...         print("Keys & Values")
...     for key, value in arg.items():
...         print(key, "=>", value)
...
>>> fun({"a": "b"})
a => b
```

To check which implementation the generic function will choose for a given type, use the `dispatch()` attribute:

```
>>> fun.dispatch(float)
<function fun_num at 0x1035a2840>
>>> fun.dispatch(dict)    # note: default implementation
<function fun at 0x103fe0000>
```

To access all registered implementations, use the read-only `registry` attribute:

```
>>> fun.registry.keys()
dict_keys([<class 'NoneType'>, <class 'int'>, <class
'object'>,
          <class 'decimal.Decimal'>, <class 'list'>,
          <class 'float'>])
>>> fun.registry[float]
<function fun_num at 0x1035a2840>
>>> fun.registry[object]
<function fun at 0x103fe0000>
```

Added in version 3.4.

Changed in version 3.7: The **`register()`** attribute now supports using type annotations.

Changed in version 3.11: The **`register()`** attribute now supports **`typing.Union`** as a type annotation.

class functools.singledispatchmethod(*func*)

Transform a method into a single-dispatch generic function.

To define a generic method, decorate it with the `@singledispatchmethod` decorator. When defining a method using `@singledispatchmethod`, note that the dispatch happens on the type of the first non-*self* or non-*cls* argument:

```
class Negator:
    @singledispatchmethod
    def neg(self, arg):
        raise NotImplementedError("Cannot negate a")
    @neg.register
    def _(self, arg: int):
        return -arg

    @neg.register
    def _(self, arg: bool):
        return not arg
```

`@singledispatchmethod` supports nesting with other decorators such as [`@classmethod`](). Note that to allow for `dispatcher.register`, `singledispatchmethod` must be the *outer most* decorator. Here is the `Negator` class with the `neg` methods bound to the class, rather than an instance of the class:

```
class Negator:
    @singledispatchmethod
    @classmethod
    def neg(cls, arg):
        raise NotImplementedError("Cannot negate a")

    @neg.register
    @classmethod
    def _(cls, arg: int):
        return -arg

    @neg.register
    @classmethod
    def _(cls, arg: bool):
        return not arg
```

The same pattern can be used for other similar decorators: **`@staticmethod`**, **`@~abc.abstractmethod`**, and others.

Added in version 3.8.

functools.update_wrapper(*wrapper*, *wrapped*, *assigned=WRAPPER_ASSIGNMENTS*, *updated=WRAPPER_UPDATES*)

Update a *wrapper* function to look like the *wrapped* function. The optional arguments are tuples to specify which attributes of the original function are assigned directly to the matching attributes on the wrapper function and which attributes of the wrapper function are updated with the corresponding attributes from the original function. The default values for these arguments are the module level constants `WRAPPER_ASSIGNMENTS` (which assigns to the wrapper function's **`__module__`**, **`__name__`**, **`__qualname__`**, **`__annotations__`**, **`__type_params__`**, and **`__doc__`**, the documentation string) and `WRAPPER_UPDATES` (which updates the wrapper function's **`__dict__`**, i.e. the instance dictionary).

To allow access to the original function for introspection and other purposes (e.g. bypassing a caching decorator such as **`lru_cache()`**), this function automatically adds a `__wrapped__` attribute to the wrapper that refers to the function being wrapped.

The main intended use for this function is in decorator functions which wrap the decorated function and return the wrapper. If the wrapper function is not updated, the metadata of the returned function will reflect the wrapper definition rather than the original function definition, which is typically less than helpful.

`update_wrapper()` may be used with callables other than functions. Any attributes named in *assigned* or *updated* that are missing from the object being wrapped are ignored (i.e. this function will not attempt to set them on the wrapper function). **`AttributeError`** is still raised if the wrapper function itself is missing any attributes named in *updated*.

Changed in version 3.2: The `__wrapped__` attribute is now automatically added. The **`__annotations__`** attribute is now copied by default. Missing attributes no longer trigger an **`AttributeError`**.

Changed in version 3.4: The `__wrapped__` attribute now always refers to the wrapped function, even if that function defined a `__wrapped__` attribute. (see bpo-17482 [https://bugs.python.org/issue?@action=redirect&bpo=17482])

Changed in version 3.12: The **`__type_params__`** attribute is now copied by default.

@functools.wraps(*wrapped*, *assigned=WRAPPER_ASSIGNMENTS*, *updated=WRAPPER_UPDATES*)

This is a convenience function for invoking **`update_wrapper()`** as a function decorator when defining a wrapper function. It is equivalent to `partial(update_wrapper, wrapped=wrapped, assigned=assigned, updated=updated)`. For example:

```
>>> from functools import wraps
>>> def my_decorator(f):
...     @wraps(f)
...     def wrapper(*args, **kwds):
...         print('Calling decorated function')
...         return f(*args, **kwds)
...     return wrapper
...
>>> @my_decorator
... def example():
...     """Docstring"""
...     print('Called example function')
...
>>> example()
Calling decorated function
Called example function
>>> example.__name__
```

```
'example'
>>> example.__doc__
'Docstring'
```

Without the use of this decorator factory, the name of the example function would have been `'wrapper'`, and the docstring of the original **`example()`** would have been lost.

`partial` Objects

`partial` objects are callable objects created by **`partial()`**. They have three read-only attributes:

partial.func

A callable object or function. Calls to the **`partial`** object will be forwarded to **`func`** with new arguments and keywords.

partial.args

The leftmost positional arguments that will be prepended to the positional arguments provided to a **`partial`** object call.

partial.keywords

The keyword arguments that will be supplied when the **`partial`** object is called.

`partial` objects are like function objects in that they are callable, weak referenceable, and can have attributes. There are some important differences. For instance, the **`__name__`** and **`__doc__`** attributes are not created automatically.

`operator` — Standard operators as functions

Source code: Lib/operator.py [https://github.com/python/cpython/tree/3.14/Lib/operator.py]

The **`operator`** module exports a set of efficient functions corresponding to the intrinsic operators of Python. For example, `operator.add(x, y)` is equivalent to the expression `x+y`. Many function names are those used for special methods, without the double underscores. For backward compatibility, many of these have a variant with the double underscores kept. The variants without the double underscores are preferred for clarity.

The functions fall into categories that perform object comparisons, logical operations, mathematical operations and sequence operations.

The object comparison functions are useful for all objects, and are named after the rich comparison operators they support:

operator.lt(*a*, *b*)
operator.le(*a*, *b*)
operator.eq(*a*, *b*)
operator.ne(*a*, *b*)
operator.ge(*a*, *b*)
operator.gt(*a*, *b*)
operator.__lt__(*a*, *b*)
operator.__le__(*a*, *b*)

operator.__eq__(*a*, *b*)
operator.__ne__(*a*, *b*)
operator.__ge__(*a*, *b*)
operator.__gt__(*a*, *b*)

Perform “rich comparisons” between *a* and *b*. Specifically, `lt(a, b)` is equivalent to `a < b`, `le(a, b)` is equivalent to `a <= b`, `eq(a, b)` is equivalent to `a == b`, `ne(a, b)` is equivalent to `a != b`, `gt(a, b)` is equivalent to `a > b` and `ge(a, b)` is equivalent to `a >= b`. Note that these functions can return any value, which may or may not be interpretable as a Boolean value. See Comparisons for more information about rich comparisons.

The logical operations are also generally applicable to all objects, and support truth tests, identity tests, and boolean operations:

operator.not_(*obj*)
operator.__not__(*obj*)

Return the outcome of **`not`** *obj*. (Note that there is no **`__not__()`** method for object instances; only the interpreter core defines this operation. The result is affected by the **`__bool__()`** and **`__len__()`** methods.)

operator.truth(*obj*)

Return **`True`** if *obj* is true, and **`False`** otherwise. This is equivalent to using the **`bool`** constructor.

operator.is_(*a*, *b*)

Return `a is b`. Tests object identity.

operator.is_not(*a*, *b*)

Return `a is not b`. Tests object identity.

operator.is_none(*a*)

Return `a is None`. Tests object identity.

Added in version 3.14.

operator.is_not_none(*a*)

Return `a is not None`. Tests object identity.

Added in version 3.14.

The mathematical and bitwise operations are the most numerous:

operator.abs(*obj*)
operator.__abs__(*obj*)

Return the absolute value of *obj*.

operator.add(*a*, *b*)
operator.__add__(*a*, *b*)

Return `a + b`, for *a* and *b* numbers.

operator.and_(*a*, *b*)
operator.__and__(*a*, *b*)

Return the bitwise and of *a* and *b*.

operator.floordiv(*a*, *b*)
operator.__floordiv__(*a*, *b*)

Return `a // b`.

operator.index(*a*)
operator.__index__(*a*)

Return *a* converted to an integer. Equivalent to `a.__index__()`.

Changed in version 3.10: The result always has exact type **`int`**. Previously, the result could have been an instance of a subclass of `int`.

operator.inv(*obj*)
operator.invert(*obj*)
operator.__inv__(*obj*)
operator.__invert__(*obj*)

Return the bitwise inverse of the number *obj*. This is equivalent to `~obj`.

operator.lshift(*a*, *b*)
operator.__lshift__(*a*, *b*)

Return *a* shifted left by *b*.

operator.mod(*a*, *b*)
operator.__mod__(*a*, *b*)

Return `a % b`.

operator.mul(*a*, *b*)
operator.__mul__(*a*, *b*)

Return `a * b`, for *a* and *b* numbers.

operator.matmul(*a*, *b*)
operator.__matmul__(*a*, *b*)

Return `a @ b`.

Added in version 3.5.

operator.neg(*obj*)
operator.__neg__(*obj*)

Return *obj* negated (`-obj`).

operator.or_(*a*, *b*)
operator.__or__(*a*, *b*)

Return the bitwise or of *a* and *b*.

operator.pos(*obj*)
operator.__pos__(*obj*)

Return *obj* positive (`+obj`).

operator.pow(*a*, *b*)
operator.__pow__(*a*, *b*)

Return `a ** b`, for *a* and *b* numbers.

operator.rshift(*a*, *b*)
operator.__rshift__(*a*, *b*)

Return *a* shifted right by *b*.

operator.sub(*a*, *b*)
operator.__sub__(*a*, *b*)

Return `a - b`.

operator.truediv(*a*, *b*)
operator.__truediv__(*a*, *b*)

Return `a / b` where 2/3 is .66 rather than 0. This is also known as “true” division.

operator.xor(*a*, *b*)
operator.__xor__(*a*, *b*)

Return the bitwise exclusive or of *a* and *b*.

Operations which work with sequences (some of them with mappings too) include:

operator.concat(*a*, *b*)
operator.__concat__(*a*, *b*)

Return `a + b` for *a* and *b* sequences.

operator.contains(*a*, *b*)
operator.__contains__(*a*, *b*)

Return the outcome of the test `b in a`. Note the reversed operands.

operator.countOf(*a*, *b*)

Return the number of occurrences of *b* in *a*.

operator.delitem(*a*, *b*)
operator.__delitem__(*a*, *b*)

Remove the value of *a* at index *b*.

operator.getitem(*a*, *b*)
operator.__getitem__(*a*, *b*)

Return the value of *a* at index *b*.

operator.indexOf(*a*, *b*)

Return the index of the first of occurrence of *b* in *a*.

operator.setitem(*a*, *b*, *c*)
operator.__setitem__(*a*, *b*, *c*)

Set the value of *a* at index *b* to *c*.

operator.length_hint(*obj*, *default=0*)

Return an estimated length for the object *obj*. First try to return its actual length, then an estimate using `object.__length_hint__()`, and finally return the default value.

Added in version 3.4.

The following operation works with callables:

operator.call(*obj*, */*, **args*, ***kwargs*)
operator.__call__(*obj*, */*, **args*, ***kwargs*)

Return `obj(*args, **kwargs)`.

Added in version 3.11.

The `operator` module also defines tools for generalized attribute and item lookups. These are useful for making fast field extractors as arguments for `map()`, `sorted()`, `itertools.groupby()`, or other functions that expect a function argument.

operator.attrgetter(*attr*)
operator.attrgetter(**attrs*)

Return a callable object that fetches *attr* from its operand. If more than one attribute is requested, returns a tuple of attributes. The attribute names can also contain dots. For example:

- After `f = attrgetter('name')`, the call `f(b)` returns `b.name`.
- After `f = attrgetter('name', 'date')`, the call `f(b)` returns `(b.name, b.date)`.
- After `f = attrgetter('name.first', 'name.last')`, the call `f(b)` returns `(b.name.first, b.name.last)`.

Equivalent to:

```
def attrgetter(*items):
    if any(not isinstance(item, str) for item in items):
        raise TypeError('attribute name must be a string')
    if len(items) == 1:
        attr = items[0]
        def g(obj):
            return resolve_attr(obj, attr)
    else:
        def g(obj):
            return tuple(resolve_attr(obj, attr) for attr in
items)
    return g

def resolve_attr(obj, attr):
    for name in attr.split("."):
        obj = getattr(obj, name)
    return obj
```

operator.itemgetter(*item*)
operator.itemgetter(**items*)

Return a callable object that fetches *item* from its operand using the operand’s `__getitem__()` method. If multiple items are specified, returns a tuple of lookup values. For example:

- After `f = itemgetter(2)`, the call `f(r)` returns `r[2]`.
- After `g = itemgetter(2, 5, 3)`, the call `g(r)` returns `(r[2], r[5], r[3])`.

Equivalent to:

```
def itemgetter(*items):
    if len(items) == 1:
        item = items[0]
        def g(obj):
            return obj[item]
    else:
        def g(obj):
            return tuple(obj[item] for item in items)
    return g
```

The items can be any type accepted by the operand’s `__getitem__()` method. Dictionaries accept any hashable value. Lists, tuples, and strings accept an index or a slice:

```
>>> itemgetter(1)('ABCDEFG')
'B'
>>> itemgetter(1, 3, 5)('ABCDEFG')
('B', 'D', 'F')
>>> itemgetter(slice(2, None))('ABCDEFG')
'CDEFG'
>>> soldier = dict(rank='captain', name='dotterbart')
>>> itemgetter('rank')(soldier)
'captain'
```

Example of using `itemgetter()` to retrieve specific fields from a tuple record:

```
>>> inventory = [('apple', 3), ('banana', 2), ('pear', 5),
('orange', 1)]
>>> getcount = itemgetter(1)
>>> list(map(getcount, inventory))
[3, 2, 5, 1]
>>> sorted(inventory, key=getcount)
[('orange', 1), ('banana', 2), ('apple', 3), ('pear', 5)]
```

operator.methodcaller(*name*, /, **args*, ***kwargs*)

Return a callable object that calls the method *name* on its operand. If additional arguments and/or keyword arguments are given, they will be given to the method as well. For example:

- After `f = methodcaller('name')`, the call `f(b)` returns `b.name()`.
- After `f = methodcaller('name', 'foo', bar=1)`, the call `f(b)` returns `b.name('foo', bar=1)`.

Equivalent to:

```
def methodcaller(name, /, *args, **kwargs):
    def caller(obj):
        return getattr(obj, name)(*args, **kwargs)
    return caller
```

Mapping Operators to Functions

This table shows how abstract operations correspond to operator symbols in the Python syntax and the functions in the `operator` module.

Operation	**Syntax**	**Function**
Addition	`a + b`	`add(a, b)`
Concatenation	`seq1 + seq2`	`concat(seq1, seq2)`
Containment Test	`obj in seq`	`contains(seq, obj)`
Division	`a / b`	`truediv(a, b)`
Division	`a // b`	`floordiv(a, b)`
Bitwise And	`a & b`	`and_(a, b)`
Bitwise Exclusive Or	`a ^ b`	`xor(a, b)`
Bitwise Inversion	`~ a`	`invert(a)`
Bitwise Or	`a | b`	`or_(a, b)`
Exponentiation	`a ** b`	`pow(a, b)`
Identity	`a is b`	`is_(a, b)`

Operation	**Syntax**	**Function**
Identity	`a is not b`	`is_not(a, b)`
Identity	`a is None`	`is_none(a)`
Identity	`a is not None`	`is_not_none(a)`
Indexed Assignment	`obj[k] = v`	`setitem(obj, k, v)`
Indexed Deletion	`del obj[k]`	`delitem(obj, k)`
Indexing	`obj[k]`	`getitem(obj, k)`
Left Shift	`a << b`	`lshift(a, b)`
Modulo	`a % b`	`mod(a, b)`
Multiplication	`a * b`	`mul(a, b)`
Matrix Multiplication	`a @ b`	`matmul(a, b)`
Negation (Arithmetic)	`- a`	`neg(a)`

Operation	Syntax	Function
Negation (Logical)	`not a`	`not_(a)`
Positive	`+ a`	`pos(a)`
Right Shift	`a >> b`	`rshift(a, b)`
Slice Assignment	`seq[i:j] = values`	`setitem(seq, slice(i, j), values)`
Slice Deletion	`del seq[i:j]`	`delitem(seq, slice(i, j))`
Slicing	`seq[i:j]`	`getitem(seq, slice(i, j))`
String Formatting	`s % obj`	`mod(s, obj)`
Subtraction	`a - b`	`sub(a, b)`
Truth Test	`obj`	`truth(obj)`
Ordering	`a < b`	`lt(a, b)`
Ordering	`a <= b`	`le(a, b)`

Operation	Syntax	Function
Equality	`a == b`	`eq(a, b)`
Difference	`a != b`	`ne(a, b)`
Ordering	`a >= b`	`ge(a, b)`
Ordering	`a > b`	`gt(a, b)`

In-place Operators

Many operations have an “in-place” version. Listed below are functions providing a more primitive access to in-place operators than the usual syntax does; for example, the statement `x += y` is equivalent to `x = operator.iadd(x, y)`. Another way to put it is to say that `z = operator.iadd(x, y)` is equivalent to the compound statement `z = x; z += y`.

In those examples, note that when an in-place method is called, the computation and assignment are performed in two separate steps. The in-place functions listed below only do the first step, calling the in-place method. The second step, assignment, is not handled.

For immutable targets such as strings, numbers, and tuples, the updated value is computed, but not assigned back to the input variable:

```
>>> a = 'hello'
>>> iadd(a, ' world')
```

```
'hello world'
>>> a
'hello'
```

For mutable targets such as lists and dictionaries, the in-place method will perform the update, so no subsequent assignment is necessary:

```
>>> s = ['h', 'e', 'l', 'l', 'o']
>>> iadd(s, [' ', 'w', 'o', 'r', 'l', 'd'])
['h', 'e', 'l', 'l', 'o', ' ', 'w', 'o', 'r', 'l', 'd']
>>> s
['h', 'e', 'l', 'l', 'o', ' ', 'w', 'o', 'r', 'l', 'd']
```

operator.iadd(*a*, *b*)
operator.__iadd__(*a*, *b*)

`a = iadd(a, b)` is equivalent to `a += b`.

operator.iand(*a*, *b*)
operator.__iand__(*a*, *b*)

`a = iand(a, b)` is equivalent to `a &= b`.

operator.iconcat(*a*, *b*)
operator.__iconcat__(*a*, *b*)

`a = iconcat(a, b)` is equivalent to `a += b` for *a* and *b* sequences.

operator.ifloordiv(*a*, *b*)
operator.__ifloordiv__(*a*, *b*)

`a = ifloordiv(a, b)` is equivalent to `a //= b`.

operator.ilshift(*a*, *b*)
operator.__ilshift__(*a*, *b*)

`a = ilshift(a, b)` is equivalent to `a <<= b`.

operator.imod(*a*, *b*)

operator.__imod__(*a*, *b*)

`a = imod(a, b)` is equivalent to `a %= b`.

operator.imul(*a*, *b*)
operator.__imul__(*a*, *b*)

`a = imul(a, b)` is equivalent to `a *= b`.

operator.imatmul(*a*, *b*)
operator.__imatmul__(*a*, *b*)

`a = imatmul(a, b)` is equivalent to `a @= b`.

Added in version 3.5.

operator.ior(*a*, *b*)
operator.__ior__(*a*, *b*)

`a = ior(a, b)` is equivalent to `a |= b`.

operator.ipow(*a*, *b*)
operator.__ipow__(*a*, *b*)

`a = ipow(a, b)` is equivalent to `a **= b`.

operator.irshift(*a*, *b*)
operator.__irshift__(*a*, *b*)

`a = irshift(a, b)` is equivalent to `a >>= b`.

operator.isub(*a*, *b*)
operator.__isub__(*a*, *b*)

`a = isub(a, b)` is equivalent to `a -= b`.

operator.itruediv(*a*, *b*)
operator.__itruediv__(*a*, *b*)

`a = itruediv(a, b)` is equivalent to `a /= b`.

operator.ixor(*a*, *b*)

operator.__ixor__(*a*, *b*)

`a = ixor(a, b)` is equivalent to `a ^= b`.

File and Directory Access

The modules described in this chapter deal with disk files and directories. For example, there are modules for reading the properties of files, manipulating paths in a portable way, and creating temporary files. The full list of modules in this chapter is:

- `pathlib` — Object-oriented filesystem paths
  - Basic use
  - Exceptions
  - Pure paths
    - General properties
    - Operators
    - Accessing individual parts
    - Methods and properties
  - Concrete paths
    - Parsing and generating URIs
    - Expanding and resolving paths
    - Querying file type and status
    - Reading and writing files
    - Reading directories
    - Creating files and directories
    - Copying, moving and deleting
    - Permissions and ownership
  - Pattern language
  - Comparison to the `glob` module
  - Comparison to the `os` and `os.path` modules
    - Corresponding tools
  - Protocols
- `os.path` — Common pathname manipulations
- `stat` — Interpreting `stat()` results
- `filecmp` — File and Directory Comparisons

  - The **`dircmp`** class
- **`tempfile`** — Generate temporary files and directories
  - Examples
  - Deprecated functions and variables
- **`glob`** — Unix style pathname pattern expansion
  - Examples
- **`fnmatch`** — Unix filename pattern matching
- **`linecache`** — Random access to text lines
- **`shutil`** — High-level file operations
  - Directory and files operations
    - Platform-dependent efficient copy operations
    - copytree example
    - rmtree example
  - Archiving operations
    - Archiving example
    - Archiving example with *base_dir*
  - Querying the size of the output terminal

See also

Module `os`

Operating system interfaces, including functions to work with files at a lower level than Python file objects.

Module `io`

Python’s built-in I/O library, including both abstract classes and some concrete classes such as file I/O.

Built-in function `open()`

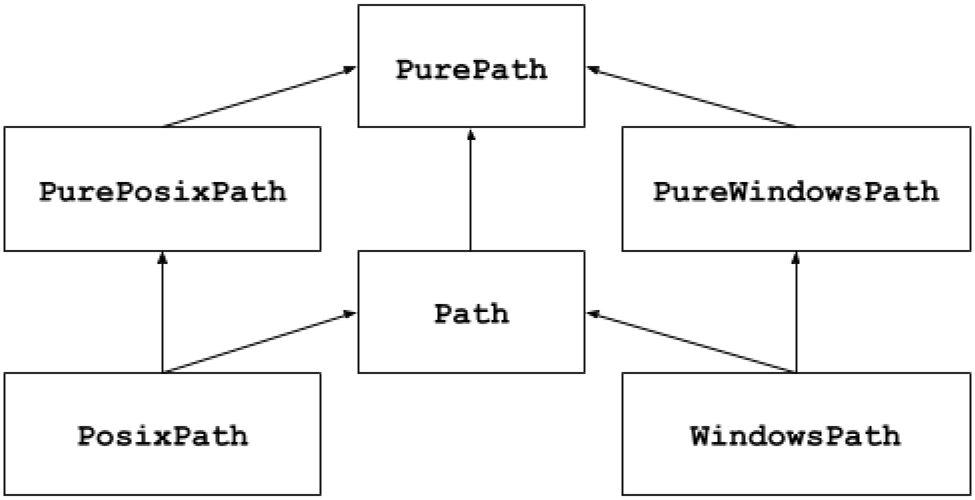
The standard way to open files for reading and writing with Python.

`pathlib` — Object-oriented filesystem paths

Added in version 3.4.

Source code: Lib/pathlib/
[https://github.com/python/cpython/tree/3.14/Lib/pathlib/]

This module offers classes representing filesystem paths with semantics appropriate for different operating systems. Path classes are divided between pure paths, which provide purely computational operations without I/O, and concrete paths, which inherit from pure paths but also provide I/O operations.


PurePath
PurePosixPath
PureWindowsPath
Path
PosixPath
WindowsPath

If you've never used this module before or just aren't sure which class is right for your task, **`Path`** is most likely what you need. It instantiates a concrete path for the platform the code is running on.

Pure paths are useful in some special cases; for example:

1. If you want to manipulate Windows paths on a Unix machine (or vice versa). You cannot instantiate a `WindowsPath` when running on Unix, but you can instantiate `PureWindowsPath`.
2. You want to make sure that your code only manipulates paths without actually accessing the OS. In this case, instantiating one of the pure classes may be useful since those simply don't have any OS-accessing operations.

See also

PEP 428 [https://peps.python.org/pep-0428/]: The pathlib module – object-oriented filesystem paths.

See also

For low-level path manipulation on strings, you can also use the `os.path` module.

Basic use

Importing the main class:

```
>>> from pathlib import Path
```

Listing subdirectories:

```
>>> p = Path('.')
>>> [x for x in p.iterdir() if x.is_dir()]
```

```
[PosixPath('.hg'), PosixPath('docs'), PosixPath('dist'),
 PosixPath('__pycache__'), PosixPath('build')]
```

Listing Python source files in this directory tree:

```
>>> list(p.glob('**/*.py'))
[PosixPath('test_pathlib.py'), PosixPath('setup.py'),
 PosixPath('pathlib.py'), PosixPath('docs/conf.py'),
 PosixPath('build/lib/pathlib.py')]
```

Navigating inside a directory tree:

```
>>> p = Path('/etc')
>>> q = p / 'init.d' / 'reboot'
>>> q
PosixPath('/etc/init.d/reboot')
>>> q.resolve()
PosixPath('/etc/rc.d/init.d/halt')
```

Querying path properties:

```
>>> q.exists()
True
>>> q.is_dir()
False
```

Opening a file:

```
>>> with q.open() as f: f.readline()
...
'#!/bin/bash\n'
```

Exceptions

exception pathlib.UnsupportedOperation

An exception inheriting `NotImplementedError` that is raised when an unsupported operation is called on a path object.

Added in version 3.13.

Pure paths

Pure path objects provide path-handling operations which don’t actually access a filesystem. There are three ways to access these classes, which we also call *flavours*:

class pathlib.PurePath(**pathsegments*)

A generic class that represents the system’s path flavour (instantiating it creates either a `PurePosixPath` or a `PureWindowsPath`):

```
>>> PurePath('setup.py')      # Running on a Unix machine
PurePosixPath('setup.py')
```

Each element of *pathsegments* can be either a string representing a path segment, or an object implementing the `os.PathLike` interface where the `__fspath__()` method returns a string, such as another path object:

```
>>> PurePath('foo', 'some/path', 'bar')
PurePosixPath('foo/some/path/bar')
>>> PurePath(Path('foo'), Path('bar'))
PurePosixPath('foo/bar')
```

When *pathsegments* is empty, the current directory is assumed:

```
>>> PurePath()
PurePosixPath('.')
```

If a segment is an absolute path, all previous segments are ignored (like `os.path.join()`):

```
>>> PurePath('/etc', '/usr', 'lib64')
PurePosixPath('/usr/lib64')
>>> PureWindowsPath('c:/Windows', 'd:bar')
PureWindowsPath('d:bar')
```

On Windows, the drive is not reset when a rooted relative path segment (e.g., `r'\foo'`) is encountered:

```
>>> PureWindowsPath('c:/Windows', '/Program Files')
PureWindowsPath('c:/Program Files')
```

Spurious slashes and single dots are collapsed, but double dots (`'..'`) and leading double slashes (`'//'`) are not, since this would change the meaning of a path for various reasons (e.g. symbolic links, UNC paths):

```
>>> PurePath('foo//bar')
PurePosixPath('foo/bar')
>>> PurePath('//foo/bar')
PurePosixPath('//foo/bar')
>>> PurePath('foo/./bar')
PurePosixPath('foo/bar')
>>> PurePath('foo/../bar')
PurePosixPath('foo/../bar')
```

(a naïve approach would make `PurePosixPath('foo/../bar')` equivalent to `PurePosixPath('bar')`, which is wrong if `foo` is a symbolic link to another directory)

Pure path objects implement the **`os.PathLike`** interface, allowing them to be used anywhere the interface is accepted.

Changed in version 3.6: Added support for the **`os.PathLike`** interface.

class pathlib.PurePosixPath(**pathsegments*)

A subclass of **`PurePath`**, this path flavour represents non-Windows filesystem paths:

```
>>> PurePosixPath('/etc/hosts')
PurePosixPath('/etc/hosts')
```

pathsegments is specified similarly to **`PurePath`**.

class pathlib.PureWindowsPath(**pathsegments*)

A subclass of **`PurePath`**, this path flavour represents Windows filesystem paths, including UNC paths [https://en.wikipedia.org/wiki/Path_(computing)#UNC]:

```
>>> PureWindowsPath('c:/', 'Users', 'Ximénez')
PureWindowsPath('c:/Users/Ximénez')
>>> PureWindowsPath('//server/share/file')
PureWindowsPath('//server/share/file')
```

pathsegments is specified similarly to **`PurePath`**.

Regardless of the system you’re running on, you can instantiate all of these classes, since they don’t provide any operation that does system calls.

General properties

Paths are immutable and hashable. Paths of a same flavour are comparable and orderable. These properties respect the flavour’s case-folding semantics:

```
>>> PurePosixPath('foo') == PurePosixPath('FOO')
False
>>> PureWindowsPath('foo') == PureWindowsPath('FOO')
True
>>> PureWindowsPath('FOO') in { PureWindowsPath('foo') }
True
>>> PureWindowsPath('C:') < PureWindowsPath('d:')
True
```

Paths of a different flavour compare unequal and cannot be ordered:

```
>>> PureWindowsPath('foo') == PurePosixPath('foo')
False
>>> PureWindowsPath('foo') < PurePosixPath('foo')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
```

```
TypeError: '<' not supported between instances of
'PureWindowsPath' and 'PurePosixPath'
```

Operators

The slash operator helps create child paths, like `os.path.join()`. If the argument is an absolute path, the previous path is ignored. On Windows, the drive is not reset when the argument is a rooted relative path (e.g., `r'\foo'`):

```
>>> p = PurePath('/etc')
>>> p
PurePosixPath('/etc')
>>> p / 'init.d' / 'apache2'
PurePosixPath('/etc/init.d/apache2')
>>> q = PurePath('bin')
>>> '/usr' / q
PurePosixPath('/usr/bin')
>>> p / '/an_absolute_path'
PurePosixPath('/an_absolute_path')
>>> PureWindowsPath('c:/Windows', '/Program Files')
PureWindowsPath('c:/Program Files')
```

A path object can be used anywhere an object implementing `os.PathLike` is accepted:

```
>>> import os
>>> p = PurePath('/etc')
>>> os.fspath(p)
'/etc'
```

The string representation of a path is the raw filesystem path itself (in native form, e.g. with backslashes under Windows), which you can pass to any function taking a file path as a string:

```
>>> p = PurePath('/etc')
>>> str(p)
'/etc'
>>> p = PureWindowsPath('c:/Program Files')
>>> str(p)
'c:\\Program Files'
```

Similarly, calling `bytes` on a path gives the raw filesystem path as a bytes object, as encoded by `os.fsencode()`:

```
>>> bytes(p)
b'/etc'
```

Note

Calling `bytes` is only recommended under Unix. Under Windows, the unicode form is the canonical representation of filesystem paths.

Accessing individual parts

To access the individual “parts” (components) of a path, use the following property:

PurePath.parts

A tuple giving access to the path’s various components:

```
>>> p = PurePath('/usr/bin/python3')
>>> p.parts
('/', 'usr', 'bin', 'python3')

>>> p = PureWindowsPath('c:/Program Files/PSF')
>>> p.parts
('c:\\', 'Program Files', 'PSF')
```

(note how the drive and local root are regrouped in a single part)

Methods and properties

Pure paths provide the following methods and properties:

PurePath.parser

The implementation of the `os.path` module used for low-level path parsing and joining: either **`posixpath`** or **`ntpath`**.

Added in version 3.13.

PurePath.drive

A string representing the drive letter or name, if any:

```
>>> PureWindowsPath('c:/Program Files/').drive
'c:'
>>> PureWindowsPath('/Program Files/').drive
''
>>> PurePosixPath('/etc').drive
''
```

UNC shares are also considered drives:

```
>>> PureWindowsPath('//host/share/foo.txt').drive
'\\\\host\\share'
```

PurePath.root

A string representing the (local or global) root, if any:

```
>>> PureWindowsPath('c:/Program Files/').root
'\\'
>>> PureWindowsPath('c:Program Files/').root
''
>>> PurePosixPath('/etc').root
'/'
```

UNC shares always have a root:

```
>>> PureWindowsPath('//host/share').root
'\\'
```

If the path starts with more than two successive slashes, `PurePosixPath` collapses them:

```
>>> PurePosixPath('//etc').root
'//'
>>> PurePosixPath('///etc').root
'/'
>>> PurePosixPath('////etc').root
'/'
```

Note

This behavior conforms to *The Open Group Base Specifications Issue 6*, paragraph 4.11 Pathname Resolution [https://pubs.opengroup.org/onlinepubs/009695399/basedefs/xbd_chap04.html#tag_04_11]:

“A pathname that begins with two successive slashes may be interpreted in an implementation-defined manner, although more than two leading slashes shall be treated as a single slash.”

PurePath.anchor

The concatenation of the drive and root:

```
>>> PureWindowsPath('c:/Program Files/').anchor
'c:\\'
>>> PureWindowsPath('c:Program Files/').anchor
'c:'
>>> PurePosixPath('/etc').anchor
'/'
>>> PureWindowsPath('//host/share').anchor
'\\\\host\\share\\'
```

PurePath.parents

An immutable sequence providing access to the logical ancestors of the path:

```
>>> p = PureWindowsPath('c:/foo/bar/setup.py')
>>> p.parents[0]
PureWindowsPath('c:/foo/bar')
```

```
>>> p.parents[1]
PureWindowsPath('c:/foo')
>>> p.parents[2]
PureWindowsPath('c:/')
```

Changed in version 3.10: The parents sequence now supports slices and negative index values.

PurePath.parent

The logical parent of the path:

```
>>> p = PurePosixPath('/a/b/c/d')
>>> p.parent
PurePosixPath('/a/b/c')
```

You cannot go past an anchor, or empty path:

```
>>> p = PurePosixPath('/')
>>> p.parent
PurePosixPath('/')
>>> p = PurePosixPath('.')
>>> p.parent
PurePosixPath('.')
```

Note

This is a purely lexical operation, hence the following behaviour:

```
>>> p = PurePosixPath('foo/..')
>>> p.parent
PurePosixPath('foo')
```

If you want to walk an arbitrary filesystem path upwards, it is recommended to first call **`Path.resolve()`** so as to resolve symlinks and eliminate `".."` components.

PurePath.name

A string representing the final path component, excluding the drive and root, if any:

```
>>> PurePosixPath('my/library/setup.py').name
'setup.py'
```

UNC drive names are not considered:

```
>>> PureWindowsPath('//some/share/setup.py').name
'setup.py'
>>> PureWindowsPath('//some/share').name
''
```

PurePath.suffix

The last dot-separated portion of the final component, if any:

```
>>> PurePosixPath('my/library/setup.py').suffix
'.py'
>>> PurePosixPath('my/library.tar.gz').suffix
'.gz'
>>> PurePosixPath('my/library').suffix
''
```

This is commonly called the file extension.

Changed in version 3.14: A single dot (”`.`”) is considered a valid suffix.

PurePath.suffixes

A list of the path’s suffixes, often called file extensions:

```
>>> PurePosixPath('my/library.tar.gar').suffixes
['.tar', '.gar']
>>> PurePosixPath('my/library.tar.gz').suffixes
['.tar', '.gz']
>>> PurePosixPath('my/library').suffixes
[]
```

Changed in version 3.14: A single dot (”.”) is considered a valid suffix.

PurePath.stem

The final path component, without its suffix:

```
>>> PurePosixPath('my/library.tar.gz').stem
'library.tar'
>>> PurePosixPath('my/library.tar').stem
'library'
>>> PurePosixPath('my/library').stem
'library'
```

PurePath.as_posix()

Return a string representation of the path with forward slashes (`/`):

```
>>> p = PureWindowsPath('c:\\windows')
>>> str(p)
'c:\\windows'
>>> p.as_posix()
'c:/windows'
```

PurePath.is_absolute()

Return whether the path is absolute or not. A path is considered absolute if it has both a root and (if the flavour allows) a drive:

```
>>> PurePosixPath('/a/b').is_absolute()
True
>>> PurePosixPath('a/b').is_absolute()
False

>>> PureWindowsPath('c:/a/b').is_absolute()
True
>>> PureWindowsPath('/a/b').is_absolute()
False
>>> PureWindowsPath('c:').is_absolute()
False
```

```
>>> PureWindowsPath('//some/share').is_absolute()
True
```

PurePath.is_relative_to(*other*)

Return whether or not this path is relative to the *other* path.

```
>>> p = PurePath('/etc/passwd')
>>> p.is_relative_to('/etc')
True
>>> p.is_relative_to('/usr')
False
```

This method is string-based; it neither accesses the filesystem nor treats “`..`” segments specially. The following code is equivalent:

```
>>> u = PurePath('/usr')
>>> u == p or u in p.parents
False
```

Added in version 3.9.

Deprecated since version 3.12, removed in version 3.14: Passing additional arguments is deprecated; if supplied, they are joined with *other*.

PurePath.is_reserved()

With **`PureWindowsPath`**, return `True` if the path is considered reserved under Windows, `False` otherwise. With **`PurePosixPath`**, `False` is always returned.

Changed in version 3.13: Windows path names that contain a colon, or end with a dot or a space, are considered reserved. UNC paths may be reserved.

Deprecated since version 3.13, will be removed in version 3.15: This method is deprecated; use **`os.path.isreserved()`** to detect reserved paths on Windows.

PurePath.joinpath(*pathsegments*)

Calling this method is equivalent to combining the path with each of the given *pathsegments* in turn:

```
>>> PurePosixPath('/etc').joinpath('passwd')
PurePosixPath('/etc/passwd')
>>> PurePosixPath('/etc').joinpath(PurePosixPath('passwd'))
PurePosixPath('/etc/passwd')
>>> PurePosixPath('/etc').joinpath('init.d', 'apache2')
PurePosixPath('/etc/init.d/apache2')
>>> PureWindowsPath('c:').joinpath('/Program Files')
PureWindowsPath('c:/Program Files')
```

PurePath.full_match(*pattern*, *, *case_sensitive=None*)

Match this path against the provided glob-style pattern. Return `True` if matching is successful, `False` otherwise. For example:

```
>>> PurePath('a/b.py').full_match('a/*.py')
True
>>> PurePath('a/b.py').full_match('*.py')
False
>>> PurePath('/a/b/c.py').full_match('/a/**')
True
>>> PurePath('/a/b/c.py').full_match('**/*.py')
True
```

See also

Pattern language documentation.

As with other methods, case-sensitivity follows platform defaults:

```
>>> PurePosixPath('b.py').full_match('*.PY')
False
>>> PureWindowsPath('b.py').full_match('*.PY')
True
```

Set *case_sensitive* to `True` or `False` to override this behaviour.

Added in version 3.13.

PurePath.match(*pattern*, *, *case_sensitive=None*)

Match this path against the provided non-recursive glob-style pattern. Return `True` if matching is successful, `False` otherwise.

This method is similar to **`full_match()`**, but empty patterns aren't allowed (**`ValueError`** is raised), the recursive wildcard "`**`" isn't supported (it acts like non-recursive "`*`"), and if a relative pattern is provided, then matching is done from the right:

```
>>> PurePath('a/b.py').match('*.py')
True
>>> PurePath('/a/b/c.py').match('b/*.py')
True
>>> PurePath('/a/b/c.py').match('a/*.py')
False
```

Changed in version 3.12: The *pattern* parameter accepts a path-like object.

Changed in version 3.12: The *case_sensitive* parameter was added.

PurePath.relative_to(*other*, *walk_up=False*)

Compute a version of this path relative to the path represented by *other*. If it's impossible, **`ValueError`** is raised:

```
>>> p = PurePosixPath('/etc/passwd')
>>> p.relative_to('/')
PurePosixPath('etc/passwd')
>>> p.relative_to('/etc')
```

```
PurePosixPath('passwd')
>>> p.relative_to('/usr')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "pathlib.py", line 941, in relative_to
    raise ValueError(error_message.format(str(self),
str(formatted)))
ValueError: '/etc/passwd' is not in the subpath of '/usr' OR
one path is relative and the other is absolute.
```

When *walk_up* is false (the default), the path must start with *other*. When the argument is true, `..` entries may be added to form the relative path. In all other cases, such as the paths referencing different drives, [`ValueError`] is raised.:

```
>>> p.relative_to('/usr', walk_up=True)
PurePosixPath('../etc/passwd')
>>> p.relative_to('foo', walk_up=True)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "pathlib.py", line 941, in relative_to
    raise ValueError(error_message.format(str(self),
str(formatted)))
ValueError: '/etc/passwd' is not on the same drive as 'foo'
OR one path is relative and the other is absolute.
```

Warning

This function is part of [`PurePath`] and works with strings. It does not check or access the underlying file structure. This can impact the *walk_up* option as it assumes that no symlinks are present in the path; call [`resolve()`] first if necessary to resolve symlinks.

Changed in version 3.12: The *walk_up* parameter was added (old behavior is the same as `walk_up=False`).

Deprecated since version 3.12, removed in version 3.14: Passing additional positional arguments is deprecated; if supplied, they are joined with *other*.

PurePath.with_name(*name*)

Return a new path with the `name` changed. If the original path doesn’t have a name, ValueError is raised:

```
>>> p = PureWindowsPath('c:/Downloads/pathlib.tar.gz')
>>> p.with_name('setup.py')
PureWindowsPath('c:/Downloads/setup.py')
>>> p = PureWindowsPath('c:/')
>>> p.with_name('setup.py')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "/home/antoine/cpython/default/Lib/pathlib.py", line
751, in with_name
    raise ValueError("%r has an empty name" % (self,))
ValueError: PureWindowsPath('c:/') has an empty name
```

PurePath.with_stem(*stem*)

Return a new path with the `stem` changed. If the original path doesn’t have a name, ValueError is raised:

```
>>> p = PureWindowsPath('c:/Downloads/draft.txt')
>>> p.with_stem('final')
PureWindowsPath('c:/Downloads/final.txt')
>>> p = PureWindowsPath('c:/Downloads/pathlib.tar.gz')
>>> p.with_stem('lib')
PureWindowsPath('c:/Downloads/lib.gz')
>>> p = PureWindowsPath('c:/')
>>> p.with_stem('')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "/home/antoine/cpython/default/Lib/pathlib.py", line
861, in with_stem
    return self.with_name(stem + self.suffix)
  File "/home/antoine/cpython/default/Lib/pathlib.py", line
851, in with_name
    raise ValueError("%r has an empty name" % (self,))
ValueError: PureWindowsPath('c:/') has an empty name
```

Added in version 3.9.

PurePath.with_suffix(*suffix*)

Return a new path with the **`suffix`** changed. If the original path doesn’t have a suffix, the new *suffix* is appended instead. If the *suffix* is an empty string, the original suffix is removed:

```
>>> p = PureWindowsPath('c:/Downloads/pathlib.tar.gz')
>>> p.with_suffix('.bz2')
PureWindowsPath('c:/Downloads/pathlib.tar.bz2')
>>> p = PureWindowsPath('README')
>>> p.with_suffix('.txt')
PureWindowsPath('README.txt')
>>> p = PureWindowsPath('README.txt')
>>> p.with_suffix('')
PureWindowsPath('README')
```

Changed in version 3.14: A single dot (”.”) is considered a valid suffix. In previous versions, **`ValueError`** is raised if a single dot is supplied.

PurePath.with_segments(**pathsegments*)

Create a new path object of the same type by combining the given *pathsegments*. This method is called whenever a derivative path is created, such as from **`parent`** and **`relative_to()`**. Subclasses may override this method to pass information to derivative paths, for example:

```
from pathlib import PurePosixPath

class MyPath(PurePosixPath):
    def __init__(self, *pathsegments, session_id):
        super().__init__(*pathsegments)
        self.session_id = session_id

    def with_segments(self, *pathsegments):
        return type(self)(*pathsegments,
session_id=self.session_id)

etc = MyPath('/etc', session_id=42)
```

```
hosts = etc / 'hosts'
print(hosts.session_id)  # 42
```

Added in version 3.12.

Concrete paths

Concrete paths are subclasses of the pure path classes. In addition to operations provided by the latter, they also provide methods to do system calls on path objects. There are three ways to instantiate concrete paths:

class pathlib.Path(**pathsegments*)

A subclass of `PurePath`, this class represents concrete paths of the system's path flavour (instantiating it creates either a `PosixPath` or a `WindowsPath`):

```
>>> Path('setup.py')
PosixPath('setup.py')
```

pathsegments is specified similarly to `PurePath`.

class pathlib.PosixPath(**pathsegments*)

A subclass of `Path` and `PurePosixPath`, this class represents concrete non-Windows filesystem paths:

```
>>> PosixPath('/etc/hosts')
PosixPath('/etc/hosts')
```

pathsegments is specified similarly to `PurePath`.

Changed in version 3.13: Raises `UnsupportedOperation` on Windows. In previous versions, `NotImplementedError` was raised instead.

class pathlib.WindowsPath(**pathsegments*)

A subclass of `Path` and `PureWindowsPath`, this class represents concrete Windows filesystem paths:

```
>>> WindowsPath('c:/', 'Users', 'Ximénez')
WindowsPath('c:/Users/Ximénez')
```

pathsegments is specified similarly to `PurePath`.

Changed in version 3.13: Raises `UnsupportedOperation` on non-Windows platforms. In previous versions, `NotImplementedError` was raised instead.

You can only instantiate the class flavour that corresponds to your system (allowing system calls on non-compatible path flavours could lead to bugs or failures in your application):

```
>>> import os
>>> os.name
'posix'
>>> Path('setup.py')
PosixPath('setup.py')
>>> PosixPath('setup.py')
PosixPath('setup.py')
>>> WindowsPath('setup.py')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "pathlib.py", line 798, in __new__
    % (cls.__name__,))
UnsupportedOperation: cannot instantiate 'WindowsPath' on your
system
```

Some concrete path methods can raise an `OSError` if a system call fails (for example because the path doesn't exist).

Parsing and generating URIs

Concrete path objects can be created from, and represented as, 'file' URIs conforming to **RFC 8089** [https://datatracker.ietf.org/doc/html/rfc8089.html].

Note

File URIs are not portable across machines with different filesystem encodings.

classmethod Path.from_uri(*uri*)

Return a new path object from parsing a ‘file’ URI. For example:

```
>>> p = Path.from_uri('file:///etc/hosts')
PosixPath('/etc/hosts')
```

On Windows, DOS device and UNC paths may be parsed from URIs:

```
>>> p = Path.from_uri('file:///c:/windows')
WindowsPath('c:/windows')
>>> p = Path.from_uri('file://server/share')
WindowsPath('//server/share')
```

Several variant forms are supported:

```
>>> p = Path.from_uri('file:////server/share')
WindowsPath('//server/share')
>>> p = Path.from_uri('file://///server/share')
WindowsPath('//server/share')
>>> p = Path.from_uri('file:c:/windows')
WindowsPath('c:/windows')
>>> p = Path.from_uri('file:/c|/windows')
WindowsPath('c:/windows')
```

`ValueError` is raised if the URI does not start with `file:`, or the parsed path isn’t absolute.

Added in version 3.13.

Changed in version 3.14: The URL authority is discarded if it matches the local hostname. Otherwise, if the authority isn't empty or `localhost`, then on Windows a UNC path is returned (as before), and on other platforms a **`ValueError`** is raised.

Path.as_uri()

Represent the path as a 'file' URI. **`ValueError`** is raised if the path isn't absolute.

```
>>> p = PosixPath('/etc/passwd')
>>> p.as_uri()
'file:///etc/passwd'
>>> p = WindowsPath('c:/Windows')
>>> p.as_uri()
'file:///c:/Windows'
```

Deprecated since version 3.14, will be removed in version 3.19: Calling this method from **`PurePath`** rather than **`Path`** is possible but deprecated. The method's use of **`os.fsencode()`** makes it strictly impure.

Expanding and resolving paths

classmethod Path.home()

Return a new path object representing the user's home directory (as returned by **`os.path.expanduser()`** with `~` construct). If the home directory can't be resolved, **`RuntimeError`** is raised.

```
>>> Path.home()
PosixPath('/home/antoine')
```

Added in version 3.5.

Path.expanduser()

Return a new path with expanded `~` and `~user` constructs, as returned by **`os.path.expanduser()`**. If a home directory can't be resolved, **`RuntimeError`** is raised.

```
>>> p = PosixPath('~/films/Monty Python')
>>> p.expanduser()
PosixPath('/home/eric/films/Monty Python')
```

Added in version 3.5.

classmethod Path.cwd()

Return a new path object representing the current directory (as returned by `os.getcwd()`):

```
>>> Path.cwd()
PosixPath('/home/antoine/pathlib')
```

Path.absolute()

Make the path absolute, without normalization or resolving symlinks. Returns a new path object:

```
>>> p = Path('tests')
>>> p
PosixPath('tests')
>>> p.absolute()
PosixPath('/home/antoine/pathlib/tests')
```

Path.resolve(*strict=False*)

Make the path absolute, resolving any symlinks. A new path object is returned:

```
>>> p = Path()
>>> p
PosixPath('.')
>>> p.resolve()
PosixPath('/home/antoine/pathlib')
```

“`..`” components are also eliminated (this is the only method to do so):

```
>>> p = Path('docs/../setup.py')
>>> p.resolve()
PosixPath('/home/antoine/pathlib/setup.py')
```

If a path doesn’t exist or a symlink loop is encountered, and *strict* is `True`, `OSError` is raised. If *strict* is `False`, the path is resolved as far as possible and any remainder is appended without checking whether it exists.

Changed in version 3.6: The *strict* parameter was added (pre-3.6 behavior is strict).

Changed in version 3.13: Symlink loops are treated like other errors: `OSError` is raised in strict mode, and no exception is raised in non-strict mode. In previous versions, `RuntimeError` is raised no matter the value of *strict*.

Path.readlink()

Return the path to which the symbolic link points (as returned by `os.readlink()`):

```
>>> p = Path('mylink')
>>> p.symlink_to('setup.py')
>>> p.readlink()
PosixPath('setup.py')
```

Added in version 3.9.

Changed in version 3.13: Raises `UnsupportedOperation` if `os.readlink()` is not available. In previous versions, `NotImplementedError` was raised.

Querying file type and status

Changed in version 3.8: `exists()`, `is_dir()`, `is_file()`, `is_mount()`, `is_symlink()`, `is_block_device()`, `is_char_device()`, `is_fifo()`, `is_socket()` now return `False` instead of raising an exception for paths that contain characters unrepresentable at the OS level.

Changed in version 3.14: The methods given above now return `False` instead of raising any **`OSError`** exception from the operating system. In previous versions, some kinds of **`OSError`** exception are raised, and others suppressed. The new behaviour is consistent with **`os.path.exists()`**, **`os.path.isdir()`**, etc. Use **`stat()`** to retrieve the file status without suppressing exceptions.

Path.stat(*, *follow_symlinks=True*)

Return an **`os.stat_result`** object containing information about this path, like **`os.stat()`**. The result is looked up at each call to this method.

This method normally follows symlinks; to stat a symlink add the argument `follow_symlinks=False`, or use **`lstat()`**.

```
>>> p = Path('setup.py')
>>> p.stat().st_size
956
>>> p.stat().st_mtime
1327883547.852554
```

Changed in version 3.10: The *follow_symlinks* parameter was added.

Path.lstat()

Like **`Path.stat()`** but, if the path points to a symbolic link, return the symbolic link's information rather than its target's.

Path.exists(*, *follow_symlinks=True*)

Return `True` if the path points to an existing file or directory. `False` will be returned if the path is invalid, inaccessible or missing. Use **`Path.stat()`** to distinguish between these cases.

This method normally follows symlinks; to check if a symlink exists, add the argument `follow_symlinks=False`.

```
>>> Path('.').exists()
True
>>> Path('setup.py').exists()
True
>>> Path('/etc').exists()
True
>>> Path('nonexistentfile').exists()
False
```

Changed in version 3.12: The *follow_symlinks* parameter was added.

Path.is_file(*, *follow_symlinks=True*)

Return `True` if the path points to a regular file. `False` will be returned if the path is invalid, inaccessible or missing, or if it points to something other than a regular file. Use **`Path.stat()`** to distinguish between these cases.

This method normally follows symlinks; to exclude symlinks, add the argument `follow_symlinks=False`.

Changed in version 3.13: The *follow_symlinks* parameter was added.

Path.is_dir(*, *follow_symlinks=True*)

Return `True` if the path points to a directory. `False` will be returned if the path is invalid, inaccessible or missing, or if it points to something other than a directory. Use **`Path.stat()`** to distinguish between these cases.

This method normally follows symlinks; to exclude symlinks to directories, add the argument `follow_symlinks=False`.

Changed in version 3.13: The *follow_symlinks* parameter was added.

Path.is_symlink()

Return `True` if the path points to a symbolic link, even if that symlink is broken. `False` will be returned if the path is invalid, inaccessible or missing, or if it points to something other than a symbolic link. Use **`Path.stat()`** to distinguish between these cases.

Path.is_junction()

Return `True` if the path points to a junction, and `False` for any other type of file. Currently only Windows supports junctions.

Added in version 3.12.

Path.is_mount()

Return `True` if the path is a *mount point*: a point in a file system where a different file system has been mounted. On POSIX, the function checks whether *path*’s parent, `path/..`, is on a different device than *path*, or whether `path/..` and *path* point to the same i-node on the same device — this should detect mount points for all Unix and POSIX variants. On Windows, a mount point is considered to be a drive letter root (e.g. `c:\`), a UNC share (e.g. `\\server\share`), or a mounted filesystem directory.

Added in version 3.7.

Changed in version 3.12: Windows support was added.

Path.is_socket()

Return `True` if the path points to a Unix socket. `False` will be returned if the path is invalid, inaccessible or missing, or if it points to something other than a Unix socket. Use **`Path.stat()`** to distinguish between these cases.

Path.is_fifo()

Return `True` if the path points to a FIFO. `False` will be returned if the path is invalid, inaccessible or missing, or if it points to something other than a FIFO. Use **`Path.stat()`** to distinguish between these cases.

Path.is_block_device()

Return `True` if the path points to a block device. `False` will be returned if the path is invalid, inaccessible or missing, or if it points to something other than a block device. Use **`Path.stat()`** to distinguish between these cases.

Path.is_char_device()

Return `True` if the path points to a character device. `False` will be returned if the path is invalid, inaccessible or missing, or if it points to something other than a character device. Use **`Path.stat()`** to distinguish between these cases.

Path.samefile(*other_path*)

Return whether this path points to the same file as *other_path*, which can be either a Path object, or a string. The semantics are similar to **`os.path.samefile()`** and **`os.path.samestat()`**.

An **`OSError`** can be raised if either file cannot be accessed for some reason.

```
>>> p = Path('spam')
>>> q = Path('eggs')
>>> p.samefile(q)
False
>>> p.samefile('spam')
True
```

Added in version 3.5.

Path.info

A **`PathInfo`** object that supports querying file type information. The object exposes methods that cache their results, which can help reduce the number of system calls needed when switching on file type. For example:

```
>>> p = Path('src')
>>> if p.info.is_symlink():
...     print('symlink')
... elif p.info.is_dir():
...     print('directory')
... elif p.info.exists():
...     print('something else')
... else:
...     print('not found')
...
directory
```

If the path was generated from **`Path.iterdir()`** then this attribute is initialized with some information about the file type gleaned from scanning the parent directory. Merely accessing **`Path.info`** does not perform any filesystem queries.

To fetch up-to-date information, it’s best to call **`Path.is_dir()`**, **`is_file()`** and **`is_symlink()`** rather than methods of this attribute. There is no way to reset the cache; instead you can create a new path object with an empty info cache via `p = Path(p)`.

Added in version 3.14.

Reading and writing files

Path.open(*mode='r'*, *buffering=-1*, *encoding=None*, *errors=None*, *newline=None*)

Open the file pointed to by the path, like the built-in `open()` function does:

```
>>> p = Path('setup.py')
>>> with p.open() as f:
...     f.readline()
...
'#!/usr/bin/env python3\n'
```

Path.read_text(*encoding=None*, *errors=None*, *newline=None*)

Return the decoded contents of the pointed-to file as a string:

```
>>> p = Path('my_text_file')
>>> p.write_text('Text file contents')
18
>>> p.read_text()
'Text file contents'
```

The file is opened and then closed. The optional parameters have the same meaning as in `open()`.

Added in version 3.5.

Changed in version 3.13: The *newline* parameter was added.

Path.read_bytes()

Return the binary contents of the pointed-to file as a bytes object:

```
>>> p = Path('my_binary_file')
>>> p.write_bytes(b'Binary file contents')
20
>>> p.read_bytes()
b'Binary file contents'
```

Added in version 3.5.

Path.write_text(*data*, *encoding=None*, *errors=None*, *newline=None*)

Open the file pointed to in text mode, write *data* to it, and close the file:

```
>>> p = Path('my_text_file')
>>> p.write_text('Text file contents')
18
>>> p.read_text()
'Text file contents'
```

An existing file of the same name is overwritten. The optional parameters have the same meaning as in `open()`.

Added in version 3.5.

Changed in version 3.10: The *newline* parameter was added.

Path.write_bytes(*data*)

Open the file pointed to in bytes mode, write *data* to it, and close the file:

```
>>> p = Path('my_binary_file')
>>> p.write_bytes(b'Binary file contents')
20
>>> p.read_bytes()
b'Binary file contents'
```

An existing file of the same name is overwritten.

Added in version 3.5.

Reading directories

Path.iterdir()

When the path points to a directory, yield path objects of the directory contents:

```
>>> p = Path('docs')
>>> for child in p.iterdir(): child
...
PosixPath('docs/conf.py')
PosixPath('docs/_templates')
PosixPath('docs/make.bat')
PosixPath('docs/index.rst')
PosixPath('docs/_build')
PosixPath('docs/_static')
PosixPath('docs/Makefile')
```

The children are yielded in arbitrary order, and the special entries `'.'` and `'..'` are not included. If a file is removed from or added to the directory after creating the iterator, it is unspecified whether a path object for that file is included.

If the path is not a directory or otherwise inaccessible, `OSError` is raised.

Path.glob(*pattern*, *, *case_sensitive=None*, *recurse_symlinks=False*)

Glob the given relative *pattern* in the directory represented by this path, yielding all matching files (of any kind):

```
>>> sorted(Path('.').glob('*.py'))
[PosixPath('pathlib.py'), PosixPath('setup.py'),
PosixPath('test_pathlib.py')]
>>> sorted(Path('.').glob('*/*.py'))
[PosixPath('docs/conf.py')]
>>> sorted(Path('.').glob('**/*.py'))
[PosixPath('build/lib/pathlib.py'),
 PosixPath('docs/conf.py'),
 PosixPath('pathlib.py'),
 PosixPath('setup.py'),
 PosixPath('test_pathlib.py')]
```

Note

The paths are returned in no particular order. If you need a specific order, sort the results.

See also

Pattern language documentation.

By default, or when the *case_sensitive* keyword-only argument is set to `None`, this method matches paths using platform-specific casing rules: typically, case-sensitive on POSIX, and case-insensitive on Windows. Set *case_sensitive* to `True` or `False` to override this behaviour.

By default, or when the *recurse_symlinks* keyword-only argument is set to `False`, this method follows symlinks except when expanding “`**`” wildcards. Set *recurse_symlinks* to `True` to always follow symlinks.

Raises an auditing event `pathlib.Path.glob` with arguments `self`, `pattern`.

Changed in version 3.12: The *case_sensitive* parameter was added.

Changed in version 3.13: The *recurse_symlinks* parameter was added.

Changed in version 3.13: The *pattern* parameter accepts a path-like object.

Changed in version 3.13: Any **`OSError`** exceptions raised from scanning the filesystem are suppressed. In previous versions, such exceptions are suppressed in many cases, but not all.

Path.rglob(*pattern*, *, *case_sensitive=None*, *recurse_symlinks=False*)

Glob the given relative *pattern* recursively. This is like calling **`Path.glob()`** with “`**/`” added in front of the *pattern*.

Note

The paths are returned in no particular order. If you need a specific order, sort the results.

See also

Pattern language and **`Path.glob()`** documentation.

Raises an auditing event `pathlib.Path.rglob` with arguments `self`, `pattern`.

Changed in version 3.12: The *case_sensitive* parameter was added.

Changed in version 3.13: The *recurse_symlinks* parameter was added.

Changed in version 3.13: The *pattern* parameter accepts a path-like object.

Path.walk(*top_down=True*, *on_error=None*, *follow_symlinks=False*)

Generate the file names in a directory tree by walking the tree either top-down or bottom-up.

For each directory in the directory tree rooted at *self* (including *self* but excluding ‘.’ and ‘..’), the method yields a 3-tuple of `(dirpath, dirnames, filenames)`.

dirpath is a `Path` to the directory currently being walked, *dirnames* is a list of strings for the names of subdirectories in *dirpath* (excluding `'.'` and `'..'`), and *filenames* is a list of strings for the names of the non-directory files in *dirpath*. To get a full path (which begins with *self*) to a file or directory in *dirpath*, do `dirpath / name`. Whether or not the lists are sorted is file system-dependent.

If the optional argument *top_down* is true (which is the default), the triple for a directory is generated before the triples for any of its subdirectories (directories are walked top-down). If *top_down* is false, the triple for a directory is generated after the triples for all of its subdirectories (directories are walked bottom-up). No matter the value of *top_down*, the list of subdirectories is retrieved before the triples for the directory and its subdirectories are walked.

When *top_down* is true, the caller can modify the *dirnames* list in-place (for example, using `del` or slice assignment), and `Path.walk()` will only recurse into the subdirectories whose names remain in *dirnames*. This can be used to prune the search, or to impose a specific order of visiting, or even to inform `Path.walk()` about directories the caller creates or renames before it resumes `Path.walk()` again. Modifying *dirnames* when *top_down* is false has no effect on the behavior of `Path.walk()` since the directories in *dirnames* have already been generated by the time *dirnames* is yielded to the caller.

By default, errors from `os.scandir()` are ignored. If the optional argument *on_error* is specified, it should be a callable; it will be called with one argument, an `OSError` instance. The callable can handle the error to continue the walk or re-raise it to stop the walk. Note that the filename is available as the `filename` attribute of the exception object.

By default, `Path.walk()` does not follow symbolic links, and instead adds them to the *filenames* list. Set *follow_symlinks* to true to resolve symlinks and place them in *dirnames* and *filenames* as appropriate for their targets, and consequently visit directories pointed to by symlinks (where supported).

Note

Be aware that setting *follow_symlinks* to true can lead to infinite recursion if a link points to a parent directory of itself. `Path.walk()` does not keep track of the directories it has already visited.

Note

`Path.walk()` assumes the directories it walks are not modified during execution. For example, if a directory from *dirnames* has been replaced with a symlink and *follow_symlinks* is false, `Path.walk()` will still try to descend into it. To prevent such behavior, remove directories from *dirnames* as appropriate.

Note

Unlike `os.walk()`, `Path.walk()` lists symlinks to directories in *filenames* if *follow_symlinks* is false.

This example displays the number of bytes used by all files in each directory, while ignoring `__pycache__` directories:

```
from pathlib import Path
for root, dirs, files in
Path("cpython/Lib/concurrent").walk(on_error=print):
  print(
      root,
      "consumes",
      sum((root / file).stat().st_size for file in files),
      "bytes in",
      len(files),
      "non-directory files"
  )
  if '__pycache__' in dirs:
        dirs.remove('__pycache__')
```

This next example is a simple implementation of `shutil.rmtree()`. Walking the tree bottom-up is essential as `rmdir()` doesn't allow deleting a directory before it is empty:

```
# Delete everything reachable from the directory "top".
# CAUTION:  This is dangerous! For example, if top ==
Path('/'),
# it could delete all of your files.
for root, dirs, files in top.walk(top_down=False):
    for name in files:
        (root / name).unlink()
    for name in dirs:
        (root / name).rmdir()
```

Added in version 3.12.

Creating files and directories

Path.touch(*mode=0o666*, *exist_ok=True*)

Create a file at this given path. If *mode* is given, it is combined with the process's `umask` value to determine the file mode and access flags. If the file already exists, the function succeeds when *exist_ok* is true (and its modification time is updated to the current time), otherwise `FileExistsError` is raised.

See also

The `open()`, `write_text()` and `write_bytes()` methods are often used to create files.

Path.mkdir(*mode=0o777*, *parents=False*, *exist_ok=False*)

Create a new directory at this given path. If *mode* is given, it is combined with the process's `umask` value to determine the file mode and access flags. If the path already exists, `FileExistsError` is raised.

If *parents* is true, any missing parents of this path are created as needed; they are created with the default permissions without taking *mode* into account (mimicking the POSIX `mkdir -p` command).

If *parents* is false (the default), a missing parent raises `FileNotFoundError`.

If *exist_ok* is false (the default), `FileExistsError` is raised if the target directory already exists.

If *exist_ok* is true, `FileExistsError` will not be raised unless the given path already exists in the file system and is not a directory (same behavior as the POSIX `mkdir -p` command).

Changed in version 3.5: The *exist_ok* parameter was added.

Path.symlink_to(*target*, *target_is_directory=False*)

Make this path a symbolic link pointing to *target*.

On Windows, a symlink represents either a file or a directory, and does not morph to the target dynamically. If the target is present, the type of the symlink will be created to match. Otherwise, the symlink will be created as a directory if *target_is_directory* is true or a file symlink (the default) otherwise. On non-Windows platforms, *target_is_directory* is ignored.

```
>>> p = Path('mylink')
>>> p.symlink_to('setup.py')
>>> p.resolve()
PosixPath('/home/antoine/pathlib/setup.py')
>>> p.stat().st_size
956
>>> p.lstat().st_size
8
```

Note

The order of arguments (link, target) is the reverse of `os.symlink()`’s.

Changed in version 3.13: Raises `UnsupportedOperation` if `os.symlink()` is not available. In previous versions, `NotImplementedError` was raised.

Path.hardlink_to(*target*)

Make this path a hard link to the same file as *target*.

Note

The order of arguments (link, target) is the reverse of `os.link()`’s.

Added in version 3.10.

Changed in version 3.13: Raises `UnsupportedOperation` if `os.link()` is not available. In previous versions, `NotImplementedError` was raised.

Copying, moving and deleting

Path.copy(*target*, *, *follow_symlinks=True*, *preserve_metadata=False*)

Copy this file or directory tree to the given *target*, and return a new `Path` instance pointing to *target*.

If the source is a file, the target will be replaced if it is an existing file. If the source is a symlink and *follow_symlinks* is true (the default), the symlink's target is copied. Otherwise, the symlink is recreated at the destination.

If *preserve_metadata* is false (the default), only directory structures and file data are guaranteed to be copied. Set *preserve_metadata* to true to ensure that file and directory permissions, flags, last access and modification times, and extended attributes are copied where supported. This argument has no effect when copying files on Windows (where metadata is always preserved).

Note

Where supported by the operating system and file system, this method performs a lightweight copy, where data blocks are only copied when modified. This is known as copy-on-write.

Added in version 3.14.

Path.copy_into(*target_dir*, *, *follow_symlinks=True*, *preserve_metadata=False*)

Copy this file or directory tree into the given *target_dir*, which should be an existing directory. Other arguments are handled identically to `Path.copy()`. Returns a new **`Path`** instance pointing to the copy.

Added in version 3.14.

Path.rename(*target*)

Rename this file or directory to the given *target*, and return a new **`Path`** instance pointing to *target*. On Unix, if *target* exists and is a file, it will be replaced silently if the user has permission. On Windows, if *target* exists, `FileExistsError` will be raised. *target* can be either a string or another path object:

```
>>> p = Path('foo')
>>> p.open('w').write('some text')
9
>>> target = Path('bar')
>>> p.rename(target)
PosixPath('bar')
>>> target.open().read()
'some text'
```

The target path may be absolute or relative. Relative paths are interpreted relative to the current working directory, *not* the directory of the **`Path`** object.

It is implemented in terms of `os.rename()` and gives the same guarantees.

Changed in version 3.8: Added return value, return the new **`Path`** instance.

Path.replace(*target*)

Rename this file or directory to the given *target*, and return a new **`Path`** instance pointing to *target*. If *target* points to an existing file or empty directory, it will be unconditionally replaced.

The target path may be absolute or relative. Relative paths are interpreted relative to the current working directory, *not* the directory of the **`Path`** object.

Changed in version 3.8: Added return value, return the new **`Path`** instance.

Path.move(*target*)

Move this file or directory tree to the given *target*, and return a new **`Path`** instance pointing to *target*.

If the *target* doesn't exist it will be created. If both this path and the *target* are existing files, then the target is overwritten. If both paths point to the same file or directory, or the *target* is a non-empty directory, then **`OSError`** is raised.

If both paths are on the same filesystem, the move is performed with **`os.replace()`**. Otherwise, this path is copied (preserving metadata and symlinks) and then deleted.

Added in version 3.14.

Path.move_into(*target_dir*)

Move this file or directory tree into the given *target_dir*, which should be an existing directory. Returns a new **`Path`** instance pointing to the moved path.

Added in version 3.14.

Path.unlink(*missing_ok=False*)

Remove this file or symbolic link. If the path points to a directory, use **`Path.rmdir()`** instead.

If *missing_ok* is false (the default), `FileNotFoundError` is raised if the path does not exist.

If *missing_ok* is true, `FileNotFoundError` exceptions will be ignored (same behavior as the POSIX `rm -f` command).

Changed in version 3.8: The *missing_ok* parameter was added.

Path.rmdir()

Remove this directory. The directory must be empty.

Permissions and ownership

Path.owner(*, *follow_symlinks=True*)

Return the name of the user owning the file. `KeyError` is raised if the file's user identifier (UID) isn't found in the system database.

This method normally follows symlinks; to get the owner of the symlink, add the argument `follow_symlinks=False`.

Changed in version 3.13: Raises `UnsupportedOperation` if the `pwd` module is not available. In earlier versions, `NotImplementedError` was raised.

Changed in version 3.13: The *follow_symlinks* parameter was added.

Path.group(*, *follow_symlinks=True*)

Return the name of the group owning the file. `KeyError` is raised if the file's group identifier (GID) isn't found in the system database.

This method normally follows symlinks; to get the group of the symlink, add the argument `follow_symlinks=False`.

Changed in version 3.13: Raises `UnsupportedOperation` if the `grp` module is not available. In earlier versions, `NotImplementedError` was raised.

Changed in version 3.13: The *follow_symlinks* parameter was added.

Path.chmod(*mode*, *, *follow_symlinks=True*)

Change the file mode and permissions, like `os.chmod()`.

This method normally follows symlinks. Some Unix flavours support changing permissions on the symlink itself; on these platforms you may add the argument `follow_symlinks=False`, or use `lchmod()`.

```
>>> p = Path('setup.py')
>>> p.stat().st_mode
33277
>>> p.chmod(0o444)
>>> p.stat().st_mode
33060
```

Changed in version 3.10: The *follow_symlinks* parameter was added.

Path.lchmod(*mode*)

Like `Path.chmod()` but, if the path points to a symbolic link, the symbolic link's mode is changed rather than its target's.

Pattern language

The following wildcards are supported in patterns for `full_match()`, `glob()` and `rglob()`:

** (entire segment)

Matches any number of file or directory segments, including zero.

`*` (entire segment)
Matches one file or directory segment.

`*` (part of a segment)
Matches any number of non-separator characters, including zero.

`?`
Matches one non-separator character.

`[seq]`
Matches one character in *seq*, where *seq* is a sequence of characters. Range expressions are supported; for example, `[a-z]` matches any lowercase ASCII letter. Multiple ranges can be combined: `[a-zA-Z0-9_]` matches any ASCII letter, digit, or underscore.

`[!seq]`
Matches one character not in *seq*, where *seq* follows the same rules as above.

For a literal match, wrap the meta-characters in brackets. For example, `"[?]"` matches the character `"?"`.

The “`**`” wildcard enables recursive globbing. A few examples:

Pattern	Meaning
“`**/*`”	Any path with at least one segment.

Pattern	Meaning
“`**/*.py`”	Any path with a final segment ending “`.py`”.
“`assets/**`”	Any path starting with “`assets/`”.
“`assets/**/*`”	Any path starting with “`assets/`”, excluding “`assets/`” itself.

Note

Globbing with the “`**`” wildcard visits every directory in the tree. Large directory trees may take a long time to search.

Changed in version 3.13: Globbing with a pattern that ends with “`**`” returns both files and directories. In previous versions, only directories were returned.

In `Path.glob()` and `rglob()`, a trailing slash may be added to the pattern to match only directories.

Changed in version 3.11: Globbing with a pattern that ends with a pathname components separator (`sep` or `altsep`) returns only directories.

Comparison to the `glob` module

The patterns accepted and results generated by `Path.glob()` and `Path.rglob()` differ slightly from those by the `glob` module:

1. Files beginning with a dot are not special in pathlib. This is like passing `include_hidden=True` to **`glob.glob()`**.
2. “**” pattern components are always recursive in pathlib. This is like passing `recursive=True` to **`glob.glob()`**.
3. “**” pattern components do not follow symlinks by default in pathlib. This behaviour has no equivalent in **`glob.glob()`**, but you can pass `recurse_symlinks=True` to **`Path.glob()`** for compatible behaviour.
4. Like all **`PurePath`** and **`Path`** objects, the values returned from **`Path.glob()`** and **`Path.rglob()`** don’t include trailing slashes.
5. The values returned from pathlib’s `path.glob()` and `path.rglob()` include the *path* as a prefix, unlike the results of `glob.glob(root_dir=path)`.
6. The values returned from pathlib’s `path.glob()` and `path.rglob()` may include *path* itself, for example when globbing “**”, whereas the results of `glob.glob(root_dir=path)` never include an empty string that would correspond to *path*.

Comparison to the `os` and `os.path` modules

pathlib implements path operations using **`PurePath`** and **`Path`** objects, and so it’s said to be *object-oriented*. On the other hand, the **`os`** and **`os.path`** modules supply functions that work with low-level `str` and `bytes` objects, which is a more *procedural* approach. Some users consider the object-oriented style to be more readable.

Many functions in **`os`** and **`os.path`** support `bytes` paths and paths relative to directory descriptors. These features aren’t available in pathlib.

Python's `str` and `bytes` types, and portions of the `os` and `os.path` modules, are written in C and are very speedy. pathlib is written in pure Python and is often slower, but rarely slow enough to matter.

pathlib's path normalization is slightly more opinionated and consistent than `os.path`. For example, whereas `os.path.abspath()` eliminates "`..`" segments from a path, which may change its meaning if symlinks are involved, `Path.absolute()` preserves these segments for greater safety.

pathlib's path normalization may render it unsuitable for some applications:

1. pathlib normalizes `Path("my_folder/")` to `Path("my_folder")`, which changes a path's meaning when supplied to various operating system APIs and command-line utilities. Specifically, the absence of a trailing separator may allow the path to be resolved as either a file or directory, rather than a directory only.
2. pathlib normalizes `Path("./my_program")` to `Path("my_program")`, which changes a path's meaning when used as an executable search path, such as in a shell or when spawning a child process. Specifically, the absence of a separator in the path may force it to be looked up in **`PATH`** rather than the current directory.

As a consequence of these differences, pathlib is not a drop-in replacement for `os.path`.

Corresponding tools

Below is a table mapping various `os` functions to their corresponding `PurePath`/`Path` equivalent.

`os` and `os.path`	`pathlib`
`os.path.dirname()`	`PurePath.parent`
`os.path.basename()`	`PurePath.name`
`os.path.splitext()`	`PurePath.stem`, `PurePath.suffix`
`os.path.join()`	`PurePath.joinpath()`
`os.path.isabs()`	`PurePath.is_absolute()`
`os.path.relpath()`	`PurePath.relative_to()` [[1]]
`os.path.expanduser()`	`Path.expanduser()` [[2]]
`os.path.realpath()`	`Path.resolve()`
`os.path.abspath()`	`Path.absolute()` [[3]]
`os.path.exists()`	`Path.exists()`
`os.path.isfile()`	`Path.is_file()`
`os.path.isdir()`	`Path.is_dir()`

os and os.path	pathlib
os.path.islink()	Path.is_symlink()
os.path.isjunction()	Path.is_junction()
os.path.ismount()	Path.is_mount()
os.path.samefile()	Path.samefile()
os.getcwd()	Path.cwd()
os.stat()	Path.stat()
os.lstat()	Path.lstat()
os.listdir()	Path.iterdir()
os.walk()	Path.walk() [[4]]
os.mkdir(), os.makedirs()	Path.mkdir()
os.link()	Path.hardlink_to()
os.symlink()	Path.symlink_to()

`os` and `os.path`	`pathlib`
`os.readlink()`	`Path.readlink()`
`os.rename()`	`Path.rename()`
`os.replace()`	`Path.replace()`
`os.remove()`, `os.unlink()`	`Path.unlink()`
`os.rmdir()`	`Path.rmdir()`
`os.chmod()`	`Path.chmod()`
`os.lchmod()`	`Path.lchmod()`

Footnotes

[1]

`os.path.relpath()` calls `abspath()` to make paths absolute and remove "`..`" parts, whereas `PurePath.relative_to()` is a lexical operation that raises `ValueError` when its inputs' anchors differ (e.g. if one path is absolute and the other relative.)

[2]

`os.path.expanduser()` returns the path unchanged if the home directory can't be resolved, whereas `Path.expanduser()` raises `RuntimeError`.

[3]

`os.path.abspath()` removes “..” components without resolving symlinks, which may change the meaning of the path, whereas `Path.absolute()` leaves any “..” components in the path.

[4]

`os.walk()` always follows symlinks when categorizing paths into *dirnames* and *filenames*, whereas `Path.walk()` categorizes all symlinks into *filenames* when *follow_symlinks* is false (the default.)

Protocols

The `pathlib.types` module provides types for static type checking.

Added in version 3.14.

class pathlib.types.PathInfo

A `typing.Protocol` describing the `Path.info` attribute. Implementations may return cached results from their methods.

exists(*, *follow_symlinks=True*)

Return `True` if the path is an existing file or directory, or any other kind of file; return `False` if the path doesn't exist.

If *follow_symlinks* is `False`, return `True` for symlinks without checking if their targets exist.

is_dir(*, *follow_symlinks=True*)

Return `True` if the path is a directory, or a symbolic link pointing to a directory; return `False` if the path is (or points to) any other kind of file, or if it doesn’t exist.

If *follow_symlinks* is `False`, return `True` only if the path is a directory (without following symlinks); return `False` if the path is any other kind of file, or if it doesn’t exist.

is_file(*, *follow_symlinks=True*)

Return `True` if the path is a file, or a symbolic link pointing to a file; return `False` if the path is (or points to) a directory or other non-file, or if it doesn’t exist.

If *follow_symlinks* is `False`, return `True` only if the path is a file (without following symlinks); return `False` if the path is a directory or other non-file, or if it doesn’t exist.

is_symlink()

Return `True` if the path is a symbolic link (even if broken); return `False` if the path is a directory or any kind of file, or if it doesn’t exist.

`os.path` — Common pathname manipulations

Source code: Lib/genericpath.py [https://github.com/python/cpython/tree/3.14/Lib/genericpath.py], Lib/posixpath.py [https://github.com/python/cpython/tree/3.14/Lib/posixpath.py] (for POSIX) and Lib/ntpath.py [https://github.com/python/cpython/tree/3.14/Lib/ntpath.py] (for Windows).

This module implements some useful functions on pathnames. To read or write files see **`open()`**, and for accessing the filesystem see the **`os`** module. The path parameters can be passed as strings, or bytes, or any object implementing the **`os.PathLike`** protocol.

Unlike a Unix shell, Python does not do any *automatic* path expansions. Functions such as **`expanduser()`** and **`expandvars()`** can be invoked explicitly when an application desires shell-like path expansion. (See also the **`glob`** module.)

See also

The **`pathlib`** module offers high-level path objects.

Note

All of these functions accept either only bytes or only string objects as their parameters. The result is an object of the same type, if a path or file name is returned.

Note

Since different operating systems have different path name conventions, there are several versions of this module in the standard library. The `os.path` module is always the path module suitable for the operating system Python is running on, and therefore usable for local paths. However, you can also import and use the individual modules if you want to manipulate a path that is *always* in one of the different formats. They all have the same interface:

- **`posixpath`** for UNIX-style paths
- **`ntpath`** for Windows paths

Changed in version 3.8: `exists()`, `lexists()`, `isdir()`, `isfile()`, `islink()`, and `ismount()` now return `False` instead of raising an exception for paths that contain characters or bytes unrepresentable at the OS level.

os.path.abspath(*path*)

Return a normalized absolutized version of the pathname *path*. On most platforms, this is equivalent to calling `normpath(join(os.getcwd(), path))`.

See also

`os.path.join()` and `os.path.normpath()`.

Changed in version 3.6: Accepts a path-like object.

os.path.basename(*path*, /)

Return the base name of pathname *path*. This is the second element of the pair returned by passing *path* to the function `split()`. Note that the result of this function is different from the Unix **basename** program; where **basename** for `'/foo/bar/'` returns `'bar'`, the `basename()` function returns an empty string (`''`).

Changed in version 3.6: Accepts a path-like object.

os.path.commonpath(*paths*)

Return the longest common sub-path of each pathname in the iterable *paths*. Raise `ValueError` if *paths* contain both absolute and relative pathnames, if *paths* are on different drives, or if *paths* is empty. Unlike `commonprefix()`, this returns a valid path.

Added in version 3.5.

Changed in version 3.6: Accepts a sequence of path-like objects.

Changed in version 3.13: Any iterable can now be passed, rather than just sequences.

os.path.commonprefix(*list*, */*)

Return the longest path prefix (taken character-by-character) that is a prefix of all paths in *list*. If *list* is empty, return the empty string (`''`).

Note

This function may return invalid paths because it works a character at a time. To obtain a valid path, see `commonpath()`.

```
>>> os.path.commonprefix(['/usr/lib', '/usr/local/lib'])
'/usr/l'

>>> os.path.commonpath(['/usr/lib', '/usr/local/lib'])
'/usr'
```

Changed in version 3.6: Accepts a path-like object.

os.path.dirname(*path*, /)

Return the directory name of pathname *path*. This is the first element of the pair returned by passing *path* to the function **`split()`**.

Changed in version 3.6: Accepts a path-like object.

os.path.exists(*path*)

Return `True` if *path* refers to an existing path or an open file descriptor. Returns `False` for broken symbolic links. On some platforms, this function may return `False` if permission is not granted to execute **`os.stat()`** on the requested file, even if the *path* physically exists.

Changed in version 3.3: *path* can now be an integer: `True` is returned if it is an open file descriptor, `False` otherwise.

Changed in version 3.6: Accepts a path-like object.

os.path.lexists(*path*)

Return `True` if *path* refers to an existing path, including broken symbolic links. Equivalent to **`exists()`** on platforms lacking **`os.lstat()`**.

Changed in version 3.6: Accepts a path-like object.

os.path.expanduser(*path*)

On Unix and Windows, return the argument with an initial component of `~` or `~user` replaced by that *user*'s home directory.

On Unix, an initial `~` is replaced by the environment variable **`HOME`** if it is set; otherwise the current user's home directory is looked up in the password directory through the built-in module **`pwd`**. An initial `~user` is looked up directly in the password directory.

On Windows, **`USERPROFILE`** will be used if set, otherwise a combination of **`HOMEPATH`** and **`HOMEDRIVE`** will be used. An initial `~user` is handled by checking that the last directory component of the current user's home directory matches **`USERNAME`**, and replacing it if so.

If the expansion fails or if the path does not begin with a tilde, the path is returned unchanged.

Changed in version 3.6: Accepts a path-like object.

Changed in version 3.8: No longer uses **`HOME`** on Windows.

os.path.expandvars(*path*)

Return the argument with environment variables expanded. Substrings of the form `$name` or `${name}` are replaced by the value of environment variable *name*. Malformed variable names and references to non-existing variables are left unchanged.

On Windows, `%name%` expansions are supported in addition to `$name` and `${name}`.

Changed in version 3.6: Accepts a path-like object.

os.path.getatime(*path*, */*)

Return the time of last access of *path*. The return value is a floating-point number giving the number of seconds since the epoch (see the **`time`** module). Raise **`OSError`** if the file does not exist or is inaccessible.

os.path.getmtime(*path*, */*)

Return the time of last modification of *path*. The return value is a floating-point number giving the number of seconds since the epoch (see the **`time`** module). Raise **`OSError`** if the file does not exist or is inaccessible.

Changed in version 3.6: Accepts a path-like object.

os.path.getctime(*path*, /)

Return the system's ctime which, on some systems (like Unix) is the time of the last metadata change, and, on others (like Windows), is the creation time for *path*. The return value is a number giving the number of seconds since the epoch (see the **`time`** module). Raise **`OSError`** if the file does not exist or is inaccessible.

Changed in version 3.6: Accepts a path-like object.

os.path.getsize(*path*, /)

Return the size, in bytes, of *path*. Raise **`OSError`** if the file does not exist or is inaccessible.

Changed in version 3.6: Accepts a path-like object.

os.path.isabs(*path*, /)

Return `True` if *path* is an absolute pathname. On Unix, that means it begins with a slash, on Windows that it begins with two (back)slashes, or a drive letter, colon, and (back)slash together.

See also

`abspath()`

Changed in version 3.6: Accepts a path-like object.

Changed in version 3.13: On Windows, returns `False` if the given path starts with exactly one (back)slash.

os.path.isfile(*path*)

Return `True` if *path* is an **`existing`** regular file. This follows symbolic links, so both **`islink()`** and **`isfile()`** can be true for the same path.

Changed in version 3.6: Accepts a path-like object.

os.path.isdir(*path*, /)

Return `True` if *path* is an **`existing`** directory. This follows symbolic links, so both **`islink()`** and **`isdir()`** can be true for the same path.

Changed in version 3.6: Accepts a path-like object.

os.path.isjunction(*path*)

Return `True` if *path* refers to an **`existing`** directory entry that is a junction. Always return `False` if junctions are not supported on the current platform.

Added in version 3.12.

os.path.islink(*path*)

Return `True` if *path* refers to an **`existing`** directory entry that is a symbolic link. Always `False` if symbolic links are not supported by the Python runtime.

Changed in version 3.6: Accepts a path-like object.

os.path.ismount(*path*)

Return `True` if pathname *path* is a *mount point*: a point in a file system where a different file system has been mounted. On POSIX, the function checks whether *path*'s parent, `path/..`, is on a different device than *path*, or whether *`path/..`* and *path* point to the same i-node on the same device — this should detect mount points for all Unix and POSIX variants. It is not able to reliably detect bind mounts on the same filesystem. On Linux systems, it will always return `True` for btrfs subvolumes, even if they aren't mount points. On Windows, a drive letter root and a share UNC are always mount points, and for any other path `GetVolumePathName` is called to see if it is different from the input path.

Changed in version 3.4: Added support for detecting non-root mount points on Windows.

Changed in version 3.6: Accepts a path-like object.

os.path.isdevdrive(*path*)

Return `True` if pathname *path* is located on a Windows Dev Drive. A Dev Drive is optimized for developer scenarios, and offers faster performance for reading and writing files. It is recommended for use for source code, temporary build directories, package caches, and other IO-intensive operations.

May raise an error for an invalid path, for example, one without a recognizable drive, but returns `False` on platforms that do not support Dev Drives. See the Windows documentation [https://learn.microsoft.com/windows/dev-drive/] for information on enabling and creating Dev Drives.

Added in version 3.12.

Changed in version 3.13: The function is now available on all platforms, and will always return `False` on those that have no support for Dev Drives os.path.isreserved(*path*)

Return `True` if *path* is a reserved pathname on the current system.

On Windows, reserved filenames include those that end with a space or dot; those that contain colons (i.e. file streams such as “name:stream”), wildcard characters (i.e. `'*?"<>'`), pipe, or ASCII control characters; as well as DOS device names such as “NUL”, “CON”, “CONIN$”, “CONOUT$”, “AUX”, “PRN”, “COM1”, and “LPT1”.

Note

This function approximates rules for reserved paths on most Windows systems. These rules change over time in various Windows releases. This function may be updated in future Python releases as changes to the rules become broadly available.

Availability: Windows.

Added in version 3.13.

os.path.join(*path*, */*, **paths*)

Join one or more path segments intelligently. The return value is the concatenation of *path* and all members of **paths*, with exactly one directory separator following each non-empty part, except the last. That is, the result will only end in a separator if the last part is either empty or ends in a separator.

If a segment is an absolute path (which on Windows requires both a drive and a root), then all previous segments are ignored and joining continues from the absolute path segment. On Linux, for example:

```
>>> os.path.join('/home/foo', 'bar')
'/home/foo/bar'
>>> os.path.join('/home/foo', '/home/bar')
'/home/bar'
```

On Windows, the drive is not reset when a rooted path segment (e.g., `r'\foo'`) is encountered. If a segment is on a different drive or is an absolute path, all previous segments are ignored and the drive is reset. For example:

```
>>> os.path.join('c:\\', 'foo')
'c:\\foo'
>>> os.path.join('c:\\foo', 'd:\\bar')
'd:\\bar'
```

Note that since there is a current directory for each drive, `os.path.join("c:", "foo")` represents a path relative to the current directory on drive `C:` (`c:foo`), not `c:\foo`.

Changed in version 3.6: Accepts a path-like object for *path* and *paths*.

os.path.normcase(*path*, */*)

Normalize the case of a pathname. On Windows, convert all characters in the pathname to lowercase, and also convert forward slashes to backward slashes. On other operating systems, return the path unchanged.

Changed in version 3.6: Accepts a path-like object.

os.path.normpath(*path*)

Normalize a pathname by collapsing redundant separators and up-level references so that `A//B`, `A/B/`, `A/./B` and `A/foo/../B` all become `A/B`. This string manipulation may change the meaning of a path that contains symbolic links. On Windows, it converts forward slashes to backward slashes. To normalize case, use `normcase()`.

Note

On POSIX systems, in accordance with IEEE Std 1003.1 2013 Edition; 4.13 Pathname Resolution [https://pubs.opengroup.org/onlinepubs/9699919799/basedefs/V1_chap04.html#tag_04_13], if a pathname begins with exactly two slashes, the first component following the leading characters may be interpreted in an implementation-defined manner, although more than two leading characters shall be treated as a single character.

Changed in version 3.6: Accepts a path-like object.

os.path.realpath(*path*, */*, ***, *strict=False*)

Return the canonical path of the specified filename, eliminating any symbolic links encountered in the path (if they are supported by the operating system). On Windows, this function will also resolve MS-DOS (also called 8.3) style names such as `C:\\PROGRA~1` to `C:\\Program Files`.

By default, the path is evaluated up to the first component that does not exist, is a symlink loop, or whose evaluation raises `OSError`. All such components are appended unchanged to the existing part of the path.

Some errors that are handled this way include “access denied”, “not a directory”, or “bad argument to internal function”. Thus, the resulting path may be missing or inaccessible, may still contain links or loops, and may traverse non-directories.

This behavior can be modified by keyword arguments:

If *strict* is `True`, the first error encountered when evaluating the path is re-raised. In particular, [`FileNotFoundError`]() is raised if *path* does not exist, or another [`OSError`]() if it is otherwise inaccessible.

If *strict* is [`os.path.ALLOW_MISSING`](), errors other than [`FileNotFoundError`]() are re-raised (as with `strict=True`). Thus, the returned path will not contain any symbolic links, but the named file and some of its parent directories may be missing.

Note

This function emulates the operating system's procedure for making a path canonical, which differs slightly between Windows and UNIX with respect to how links and subsequent path components interact.

Operating system APIs make paths canonical as needed, so it's not normally necessary to call this function.

Changed in version 3.6: Accepts a path-like object.

Changed in version 3.8: Symbolic links and junctions are now resolved on Windows.

Changed in version 3.10: The *strict* parameter was added.

Changed in version 3.14: The `ALLOW_MISSING` value for the *strict* parameter was added.

os.path.ALLOW_MISSING

Special value used for the *strict* argument in `realpath()`.

Added in version 3.14.

os.path.relpath(*path*, *start=os.curdir*)

Return a relative filepath to *path* either from the current directory or from an optional *start* directory. This is a path computation: the filesystem is not accessed to confirm the existence or nature of *path* or *start*. On Windows, `ValueError` is raised when *path* and *start* are on different drives.

start defaults to `os.curdir`.

Changed in version 3.6: Accepts a path-like object.

os.path.samefile(*path1*, *path2*, */*)

Return `True` if both pathname arguments refer to the same file or directory. This is determined by the device number and i-node number and raises an exception if an `os.stat()` call on either pathname fails.

Changed in version 3.2: Added Windows support.

Changed in version 3.4: Windows now uses the same implementation as all other platforms.

Changed in version 3.6: Accepts a path-like object.

os.path.sameopenfile(*fp1*, *fp2*)

Return `True` if the file descriptors *fp1* and *fp2* refer to the same file.

Changed in version 3.2: Added Windows support.

Changed in version 3.6: Accepts a path-like object.

os.path.samestat(*stat1*, *stat2*, */*)

Return `True` if the stat tuples *stat1* and *stat2* refer to the same file. These structures may have been returned by [**`os.fstat()`**](), [**`os.lstat()`**](), or [**`os.stat()`**](). This function implements the underlying comparison used by [**`samefile()`**]() and [**`sameopenfile()`**]().

Changed in version 3.4: Added Windows support.

os.path.split(*path*, */*)

Split the pathname *path* into a pair, `(head, tail)` where *tail* is the last pathname component and *head* is everything leading up to that. The *tail* part will never contain a slash; if *path* ends in a slash, *tail* will be empty. If there is no slash in *path*, *head* will be empty. If *path* is empty, both *head* and *tail* are empty. Trailing slashes are stripped from *head* unless it is the root (one or more slashes only). In all cases, `join(head, tail)` returns a path to the same location as *path* (but the strings may differ). Also see the functions [**`join()`**](), [**`dirname()`**]() and [**`basename()`**]().

Changed in version 3.6: Accepts a [path-like object]().

os.path.splitdrive(*path*, */*)

Split the pathname *path* into a pair `(drive, tail)` where *drive* is either a mount point or the empty string. On systems which do not use drive specifications, *drive* will always be the empty string. In all cases, `drive + tail` will be the same as *path*.

On Windows, splits a pathname into drive/UNC sharepoint and relative path.

If the path contains a drive letter, drive will contain everything up to and including the colon:

```
>>> splitdrive("c:/dir")
("c:", "/dir")
```

If the path contains a UNC path, drive will contain the host name and share:

```
>>> splitdrive("//host/computer/dir")
("//host/computer", "/dir")
```

Changed in version 3.6: Accepts a path-like object.

os.path.splitroot(*path*, */*)

Split the pathname *path* into a 3-item tuple `(drive, root, tail)` where *drive* is a device name or mount point, *root* is a string of separators after the drive, and *tail* is everything after the root. Any of these items may be the empty string. In all cases, `drive + root + tail` will be the same as *path*.

On POSIX systems, *drive* is always empty. The *root* may be empty (if *path* is relative), a single forward slash (if *path* is absolute), or two forward slashes (implementation-defined per IEEE Std 1003.1-2017; 4.13 Pathname Resolution [https://pubs.opengroup.org/onlinepubs/9699919799/basedefs/V1_chap04.html#tag_04_13].) For example:

```
>>> splitroot('/home/sam')
('', '/', 'home/sam')
>>> splitroot('//home/sam')
('', '//', 'home/sam')
>>> splitroot('///home/sam')
('', '/', '//home/sam')
```

On Windows, *drive* may be empty, a drive-letter name, a UNC share, or a device name. The *root* may be empty, a forward slash, or a backward slash. For example:

```
>>> splitroot('C:/Users/Sam')
('C:', '/', 'Users/Sam')
>>> splitroot('//Server/Share/Users/Sam')
('//Server/Share', '/', 'Users/Sam')
```

Added in version 3.12.

os.path.splitext(*path*, */*)

Split the pathname *path* into a pair `(root, ext)` such that `root + ext == path`, and the extension, *ext*, is empty or begins with a period and contains at most one period.

If the path contains no extension, *ext* will be `''`:

```
>>> splitext('bar')
('bar', '')
```

If the path contains an extension, then *ext* will be set to this extension, including the leading period. Note that previous periods will be ignored:

```
>>> splitext('foo.bar.exe')
('foo.bar', '.exe')
>>> splitext('/foo/bar.exe')
('/foo/bar', '.exe')
```

Leading periods of the last component of the path are considered to be part of the root:

```
>>> splitext('.cshrc')
('.cshrc', '')
>>> splitext('/foo/....jpg')
('/foo/....jpg', '')
```

Changed in version 3.6: Accepts a path-like object.

os.path.supports_unicode_filenames

`True` if arbitrary Unicode strings can be used as file names (within limitations imposed by the file system).

`stat` — Interpreting `stat()` results

Source code: Lib/stat.py
[https://github.com/python/cpython/tree/3.14/Lib/stat.py]

The `stat` module defines constants and functions for interpreting the results of `os.stat()`, `os.fstat()` and `os.lstat()` (if they exist). For complete details about the **`stat()`**, **`fstat()`** and **`lstat()`** calls, consult the documentation for your system.

Changed in version 3.4: The stat module is backed by a C implementation.

The `stat` module defines the following functions to test for specific file types:

stat.S_ISDIR(*mode*)

Return non-zero if the mode is from a directory.

stat.S_ISCHR(*mode*)

Return non-zero if the mode is from a character special device file.

stat.S_ISBLK(*mode*)

Return non-zero if the mode is from a block special device file.

stat.S_ISREG(*mode*)

Return non-zero if the mode is from a regular file.

stat.S_ISFIFO(*mode*)

Return non-zero if the mode is from a FIFO (named pipe).

stat.S_ISLNK(*mode*)

Return non-zero if the mode is from a symbolic link.

stat.S_ISSOCK(*mode*)

Return non-zero if the mode is from a socket.

stat.S_ISDOOR(*mode*)

Return non-zero if the mode is from a door.

Added in version 3.4.

stat.S_ISPORT(*mode*)

Return non-zero if the mode is from an event port.

Added in version 3.4.

stat.S_ISWHT(*mode*)

Return non-zero if the mode is from a whiteout.

Added in version 3.4.

Two additional functions are defined for more general manipulation of the file's mode:

stat.S_IMODE(*mode*)

Return the portion of the file's mode that can be set by `os.chmod()`—that is, the file's permission bits, plus the sticky bit, set-group-id, and set-user-id bits (on systems that support them).

stat.S_IFMT(*mode*)

Return the portion of the file’s mode that describes the file type (used by the **`S_IS*()`** functions above).

Normally, you would use the **`os.path.is*()`** functions for testing the type of a file; the functions here are useful when you are doing multiple tests of the same file and wish to avoid the overhead of the **`stat()`** system call for each test. These are also useful when checking for information about a file that isn’t handled by **`os.path`**, like the tests for block and character devices.

Example:

```
import os, sys
from stat import *

def walktree(top, callback):
    '''recursively descend the directory tree rooted at top,
       calling the callback function for each regular file'''

    for f in os.listdir(top):
        pathname = os.path.join(top, f)
        mode = os.lstat(pathname).st_mode
        if S_ISDIR(mode):
            # It's a directory, recurse into it
            walktree(pathname, callback)
        elif S_ISREG(mode):
            # It's a file, call the callback function
            callback(pathname)
        else:
            # Unknown file type, print a message
            print('Skipping %s' % pathname)

def visitfile(file):
    print('visiting', file)

if __name__ == '__main__':
    walktree(sys.argv[1], visitfile)
```

An additional utility function is provided to convert a file’s mode in a human readable string:

stat.filemode(*mode*)

Convert a file’s mode to a string of the form ‘-rwxrwxrwx’.

Added in version 3.3.

Changed in version 3.4: The function supports `S_IFDOOR`, `S_IFPORT` and `S_IFWHT`.

All the variables below are simply symbolic indexes into the 10-tuple returned by `os.stat()`, `os.fstat()` or `os.lstat()`.

stat.ST_MODE

Inode protection mode.

stat.ST_INO

Inode number.

stat.ST_DEV

Device inode resides on.

stat.ST_NLINK

Number of links to the inode.

stat.ST_UID

User id of the owner.

stat.ST_GID

Group id of the owner.

stat.ST_SIZE

Size in bytes of a plain file; amount of data waiting on some special files.

stat.ST_ATIME

Time of last access.

stat.ST_MTIME

Time of last modification.

stat.ST_CTIME

The “ctime” as reported by the operating system. On some systems (like Unix) is the time of the last metadata change, and, on others (like Windows), is the creation time (see platform documentation for details).

The interpretation of “file size” changes according to the file type. For plain files this is the size of the file in bytes. For FIFOs and sockets under most flavors of Unix (including Linux in particular), the “size” is the number of bytes waiting to be read at the time of the call to `os.stat()`, `os.fstat()`, or `os.lstat()`; this can sometimes be useful, especially for polling one of these special files after a non-blocking open. The meaning of the size field for other character and block devices varies more, depending on the implementation of the underlying system call.

The variables below define the flags used in the `ST_MODE` field.

Use of the functions above is more portable than use of the first set of flags:

stat.S_IFSOCK

Socket.

stat.S_IFLNK

Symbolic link.

stat.S_IFREG

Regular file.

stat.S_IFBLK
Block device.

stat.S_IFDIR
Directory.

stat.S_IFCHR
Character device.

stat.S_IFIFO
FIFO.

stat.S_IFDOOR
Door.

Added in version 3.4.

stat.S_IFPORT
Event port.

Added in version 3.4.

stat.S_IFWHT
Whiteout.

Added in version 3.4.

Note

`S_IFDOOR`, `S_IFPORT` or `S_IFWHT` are defined as 0 when the platform does not have support for the file types.

The following flags can also be used in the *mode* argument of `os.chmod()`:

stat.S_ISUID

Set UID bit.

stat.S_ISGID

Set-group-ID bit. This bit has several special uses. For a directory it indicates that BSD semantics is to be used for that directory: files created there inherit their group ID from the directory, not from the effective group ID of the creating process, and directories created there will also get the `S_ISGID` bit set. For a file that does not have the group execution bit (`S_IXGRP`) set, the set-group-ID bit indicates mandatory file/record locking (see also `S_ENFMT`).

stat.S_ISVTX

Sticky bit. When this bit is set on a directory it means that a file in that directory can be renamed or deleted only by the owner of the file, by the owner of the directory, or by a privileged process.

stat.S_IRWXU

Mask for file owner permissions.

stat.S_IRUSR

Owner has read permission.

stat.S_IWUSR

Owner has write permission.

stat.S_IXUSR

Owner has execute permission.

stat.S_IRWXG

Mask for group permissions.

stat.S_IRGRP

Group has read permission.

stat.S_IWGRP
  Group has write permission.

stat.S_IXGRP
  Group has execute permission.

stat.S_IRWXO
  Mask for permissions for others (not in group).

stat.S_IROTH
  Others have read permission.

stat.S_IWOTH
  Others have write permission.

stat.S_IXOTH
  Others have execute permission.

stat.S_ENFMT
  System V file locking enforcement. This flag is shared with `S_ISGID`: file/record locking is enforced on files that do not have the group execution bit (`S_IXGRP`) set.

stat.S_IREAD
  Unix V7 synonym for `S_IRUSR`.

stat.S_IWRITE
  Unix V7 synonym for `S_IWUSR`.

stat.S_IEXEC
  Unix V7 synonym for `S_IXUSR`.

The following flags can be used in the *flags* argument of `os.chflags()`:

stat.UF_SETTABLE

All user settable flags.

Added in version 3.13.

stat.UF_NODUMP
Do not dump the file.

stat.UF_IMMUTABLE
The file may not be changed.

stat.UF_APPEND
The file may only be appended to.

stat.UF_OPAQUE
The directory is opaque when viewed through a union stack.

stat.UF_NOUNLINK
The file may not be renamed or deleted.

stat.UF_COMPRESSED
The file is stored compressed (macOS 10.6+).

stat.UF_TRACKED
Used for handling document IDs (macOS)

Added in version 3.13.

stat.UF_DATAVAULT
The file needs an entitlement for reading or writing (macOS 10.13+)

Added in version 3.13.

stat.UF_HIDDEN
The file should not be displayed in a GUI (macOS 10.5+).

stat.SF_SETTABLE
All super-user changeable flags

Added in version 3.13.

stat.SF_SUPPORTED
All super-user supported flags

Availability: macOS

Added in version 3.13.

stat.SF_SYNTHETIC
All super-user read-only synthetic flags

Availability: macOS

Added in version 3.13.

stat.SF_ARCHIVED
The file may be archived.

stat.SF_IMMUTABLE
The file may not be changed.

stat.SF_APPEND
The file may only be appended to.

stat.SF_RESTRICTED
The file needs an entitlement to write to (macOS 10.13+)

Added in version 3.13.

stat.SF_NOUNLINK
The file may not be renamed or deleted.

stat.SF_SNAPSHOT

The file is a snapshot file.

stat.SF_FIRMLINK
The file is a firmlink (macOS 10.15+)

Added in version 3.13.

stat.SF_DATALESS
The file is a dataless object (macOS 10.15+)

Added in version 3.13.

See the *BSD or macOS systems man page *chflags(2)* *[https://manpages.debian.org/chflags(2)]* for more information.

On Windows, the following file attribute constants are available for use when testing bits in the `st_file_attributes` member returned by **`os.stat()`**. See the Windows API documentation [https://msdn.microsoft.com/en-us/library/windows/desktop/gg258117.aspx] for more detail on the meaning of these constants.

stat.FILE_ATTRIBUTE_ARCHIVE
stat.FILE_ATTRIBUTE_COMPRESSED
stat.FILE_ATTRIBUTE_DEVICE
stat.FILE_ATTRIBUTE_DIRECTORY
stat.FILE_ATTRIBUTE_ENCRYPTED
stat.FILE_ATTRIBUTE_HIDDEN
stat.FILE_ATTRIBUTE_INTEGRITY_STREAM
stat.FILE_ATTRIBUTE_NORMAL
stat.FILE_ATTRIBUTE_NOT_CONTENT_INDEXED
stat.FILE_ATTRIBUTE_NO_SCRUB_DATA
stat.FILE_ATTRIBUTE_OFFLINE
stat.FILE_ATTRIBUTE_READONLY
stat.FILE_ATTRIBUTE_REPARSE_POINT
stat.FILE_ATTRIBUTE_SPARSE_FILE

stat.FILE_ATTRIBUTE_SYSTEM
stat.FILE_ATTRIBUTE_TEMPORARY
stat.FILE_ATTRIBUTE_VIRTUAL

Added in version 3.5.

On Windows, the following constants are available for comparing against the `st_reparse_tag` member returned by **`os.lstat()`**. These are well-known constants, but are not an exhaustive list.

stat.IO_REPARSE_TAG_SYMLINK
stat.IO_REPARSE_TAG_MOUNT_POINT
stat.IO_REPARSE_TAG_APPEXECLINK

Added in version 3.8.

`filecmp` — File and Directory Comparisons

Source code: Lib/filecmp.py
[https://github.com/python/cpython/tree/3.14/Lib/filecmp.py]

The **`filecmp`** module defines functions to compare files and directories, with various optional time/correctness trade-offs. For comparing files, see also the **`difflib`** module.

The **`filecmp`** module defines the following functions:

filecmp.cmp(*f1*, *f2*, *shallow=True*)

Compare the files named *f1* and *f2*, returning `True` if they seem equal, `False` otherwise.

If *shallow* is true and the **`os.stat()`** signatures (file type, size, and modification time) of both files are identical, the files are taken to be equal.

Otherwise, the files are treated as different if their sizes or contents differ.

Note that no external programs are called from this function, giving it portability and efficiency.

This function uses a cache for past comparisons and the results, with cache entries invalidated if the **`os.stat()`** information for the file changes. The entire cache may be cleared using **`clear_cache()`**.

filecmp.cmpfiles(*dir1*, *dir2*, *common*, *shallow=True*)

Compare the files in the two directories *dir1* and *dir2* whose names are given by *common*.

Returns three lists of file names: *match*, *mismatch*, *errors*. *match* contains the list of files that match, *mismatch* contains the names of those that don’t, and *errors* lists the names of files which could not be compared. Files are listed in *errors* if they don’t exist in one of the directories, the user lacks permission to read them or if the comparison could not be done for some other reason.

The *shallow* parameter has the same meaning and default value as for `filecmp.cmp()`.

For example, `cmpfiles('a', 'b', ['c', 'd/e'])` will compare `a/c` with `b/c` and `a/d/e` with `b/d/e`. `'c'` and `'d/e'` will each be in one of the three returned lists.

filecmp.clear_cache()

Clear the filecmp cache. This may be useful if a file is compared so quickly after it is modified that it is within the mtime resolution of the underlying filesystem.

Added in version 3.4.

The `dircmp` class

class filecmp.dircmp(*a*, *b*, *ignore=None*, *hide=None*, ***, *shallow=True*)

Construct a new directory comparison object, to compare the directories *a* and *b*. *ignore* is a list of names to ignore, and defaults to `filecmp.DEFAULT_IGNORES`. *hide* is a list of names to hide, and defaults to `[os.curdir, os.pardir]`.

The `dircmp` class compares files by doing *shallow* comparisons as described for `filecmp.cmp()` by default using the *shallow* parameter.

Changed in version 3.13: Added the *shallow* parameter.

The `dircmp` class provides the following methods:

report()

Print (to `sys.stdout`) a comparison between *a* and *b*.

report_partial_closure()

Print a comparison between *a* and *b* and common immediate subdirectories.

report_full_closure()

Print a comparison between *a* and *b* and common subdirectories (recursively).

The `dircmp` class offers a number of interesting attributes that may be used to get various bits of information about the directory trees being compared.

Note that via `__getattr__()` hooks, all attributes are computed lazily, so there is no speed penalty if only those attributes which are lightweight to compute are used.

left

The directory *a*.

right

The directory *b*.

left_list

Files and subdirectories in *a*, filtered by *hide* and *ignore*.

right_list
Files and subdirectories in *b*, filtered by *hide* and *ignore*.

common
Files and subdirectories in both *a* and *b*.

left_only
Files and subdirectories only in *a*.

right_only
Files and subdirectories only in *b*.

common_dirs
Subdirectories in both *a* and *b*.

common_files
Files in both *a* and *b*.

common_funny
Names in both *a* and *b*, such that the type differs between the directories, or names for which `os.stat()` reports an error.

same_files
Files which are identical in both *a* and *b*, using the class's file comparison operator.

diff_files
Files which are in both *a* and *b*, whose contents differ according to the class's file comparison operator.

funny_files

Files which are in both *a* and *b*, but could not be compared.

subdirs

A dictionary mapping names in `common_dirs` to `dircmp` instances (or MyDirCmp instances if this instance is of type MyDirCmp, a subclass of `dircmp`).

Changed in version 3.10: Previously entries were always `dircmp` instances. Now entries are the same type as *self*, if *self* is a subclass of `dircmp`.

filecmp.DEFAULT_IGNORES

Added in version 3.4.

List of directories ignored by `dircmp` by default.

Here is a simplified example of using the `subdirs` attribute to search recursively through two directories to show common different files:

```
>>> from filecmp import dircmp
>>> def print_diff_files(dcmp):
...     for name in dcmp.diff_files:
...         print("diff_file %s found in %s and %s" % (name,
dcmp.left,
...               dcmp.right))
...     for sub_dcmp in dcmp.subdirs.values():
...         print_diff_files(sub_dcmp)
...
>>> dcmp = dircmp('dir1', 'dir2')
>>> print_diff_files(dcmp)
```

`tempfile` — Generate temporary files and directories

Source code: Lib/tempfile.py
[https://github.com/python/cpython/tree/3.14/Lib/tempfile.py]

This module creates temporary files and directories. It works on all supported platforms. **`TemporaryFile`**, **`NamedTemporaryFile`**, **`TemporaryDirectory`**, and **`SpooledTemporaryFile`** are high-level interfaces which provide automatic cleanup and can be used as context managers. **`mkstemp()`** and **`mkdtemp()`** are lower-level functions which require manual cleanup.

All the user-callable functions and constructors take additional arguments which allow direct control over the location and name of temporary files and directories. Files names used by this module include a string of random characters which allows those files to be securely created in shared temporary directories. To maintain backward compatibility, the argument order is somewhat odd; it is recommended to use keyword arguments for clarity.

The module defines the following user-callable items:

tempfile.TemporaryFile(*mode='w+b'*, *buffering=-1*, *encoding=None*, *newline=None*, *suffix=None*, *prefix=None*, *dir=None*, *, *errors=None*)

Return a file-like object that can be used as a temporary storage area. The file is created securely, using the same rules as **`mkstemp()`**. It will be destroyed as soon as it is closed (including an implicit close when the object is garbage collected). Under Unix, the directory entry for the file is either not created at all or is removed immediately after the file is created. Other platforms do not support this; your code should not rely on a temporary file created using this function having or not having a visible name in the file system.

The resulting object can be used as a context manager (see Examples). On completion of the context or destruction of the file object the temporary file will be removed from the filesystem.

The *mode* parameter defaults to `'w+b'` so that the file created can be read and written without being closed. Binary mode is used so that it behaves consistently on all platforms without regard for the data that is stored. *buffering*, *encoding*, *errors* and *newline* are interpreted as for **`open()`**.

The *dir*, *prefix* and *suffix* parameters have the same meaning and defaults as with **`mkstemp()`**.

The returned object is a true file object on POSIX platforms. On other platforms, it is a file-like object whose **`file`** attribute is the underlying true file object.

The **`os.O_TMPFILE`** flag is used if it is available and works (Linux-specific, requires Linux kernel 3.11 or later).

On platforms that are neither Posix nor Cygwin, TemporaryFile is an alias for NamedTemporaryFile.

Raises an auditing event `tempfile.mkstemp` with argument `fullpath`.

Changed in version 3.5: The `os.O_TMPFILE` flag is now used if available.

Changed in version 3.8: Added *errors* parameter.

tempfile.NamedTemporaryFile(*mode='w+b'*, *buffering=-1*, *encoding=None*, *newline=None*, *suffix=None*, *prefix=None*, *dir=None*, *delete=True*, ***, *errors=None*, *delete_on_close=True*)

This function operates exactly as `TemporaryFile()` does, except the following differences:

- This function returns a file that is guaranteed to have a visible name in the file system.
- To manage the named file, it extends the parameters of `TemporaryFile()` with *delete* and *delete_on_close* parameters that determine whether and how the named file should be automatically deleted.

The returned object is always a file-like object whose `file` attribute is the underlying true file object. This file-like object can be used in a `with` statement, just like a normal file. The name of the temporary file can be retrieved from the `name` attribute of the returned file-like object. On Unix, unlike with the `TemporaryFile()`, the directory entry does not get unlinked immediately after the file creation.

If *delete* is true (the default) and *delete_on_close* is true (the default), the file is deleted as soon as it is closed. If *delete* is true and *delete_on_close* is false, the file is deleted on context manager exit only, or else when the file-like object is finalized. Deletion is not always guaranteed in this case (see `object.__del__()`). If *delete* is false, the value of *delete_on_close* is ignored.

Therefore to use the name of the temporary file to reopen the file after closing it, either make sure not to delete the file upon closure (set the *delete* parameter to be false) or, in case the temporary file is created in a `with` statement, set the *delete_on_close* parameter to be false. The latter approach is recommended as it provides assistance in automatic cleaning of the temporary file upon the context manager exit.

Opening the temporary file again by its name while it is still open works as follows:

- On POSIX the file can always be opened again.
- On Windows, make sure that at least one of the following conditions are fulfilled:
  - *delete* is false
  - additional open shares delete access (e.g. by calling `os.open()` with the flag `O_TEMPORARY`)
  - *delete* is true but *delete_on_close* is false. Note, that in this case the additional opens that do not share delete access (e.g. created via builtin `open()`) must be closed before exiting the context manager, else the `os.unlink()` call on context manager exit will fail with a `PermissionError`.

On Windows, if *delete_on_close* is false, and the file is created in a directory for which the user lacks delete access, then the `os.unlink()` call on exit of the context manager will fail with a `PermissionError`. This cannot happen when *delete_on_close* is true because delete access is requested by the open, which fails immediately if the requested access is not granted.

On POSIX (only), a process that is terminated abruptly with SIGKILL cannot automatically delete any NamedTemporaryFiles it created.

Raises an auditing event `tempfile.mkstemp` with argument `fullpath`.

Changed in version 3.8: Added *errors* parameter.

Changed in version 3.12: Added *delete_on_close* parameter.

class tempfile.SpooledTemporaryFile(*max_size=0*, *mode='w+b'*, *buffering=-1*, *encoding=None*, *newline=None*, *suffix=None*, *prefix=None*, *dir=None*, ***, *errors=None*)

This class operates exactly as `TemporaryFile()` does, except that data is spooled in memory until the file size exceeds *max_size*, or until the file's `fileno()` method is called, at which point the contents are written to disk and operation proceeds as with `TemporaryFile()`.

rollover()

The resulting file has one additional method, **`rollover()`**, which causes the file to roll over to an on-disk file regardless of its size.

The returned object is a file-like object whose **`_file`** attribute is either an `io.BytesIO` or `io.TextIOWrapper` object (depending on whether binary or text *mode* was specified) or a true file object, depending on whether `rollover()` has been called. This file-like object can be used in a `with` statement, just like a normal file.

Changed in version 3.3: the truncate method now accepts a *size* argument.

Changed in version 3.8: Added *errors* parameter.

Changed in version 3.11: Fully implements the `io.BufferedIOBase` and `io.TextIOBase` abstract base classes (depending on whether binary or text *mode* was specified).

class tempfile.TemporaryDirectory(*suffix=None*, *prefix=None*, *dir=None*, *ignore_cleanup_errors=False*, ***, *delete=True*)

This class securely creates a temporary directory using the same rules as **`mkdtemp()`**. The resulting object can be used as a context manager (see Examples). On completion of the context or destruction of the temporary directory object, the newly created temporary directory and all its contents are removed from the filesystem.

name

The directory name can be retrieved from the **`name`** attribute of the returned object. When the returned object is used as a context manager, the **`name`** will be assigned to the target of the **`as`** clause in the **`with`** statement, if there is one.

cleanup()

The directory can be explicitly cleaned up by calling the **`cleanup()`** method. If *ignore_cleanup_errors* is true, any unhandled exceptions during explicit or implicit cleanup (such as a **`PermissionError`** removing open files on Windows) will be ignored, and the remaining removable items deleted on a “best-effort” basis. Otherwise, errors will be raised in whatever context cleanup occurs (the **`cleanup()`** call, exiting the context manager, when the object is garbage-collected or during interpreter shutdown).

The *delete* parameter can be used to disable cleanup of the directory tree upon exiting the context. While it may seem unusual for a context manager to disable the action taken when exiting the context, it can be useful during debugging or when you need your cleanup behavior to be conditional based on other logic.

Raises an auditing event `tempfile.mkdtemp` with argument `fullpath`.

Added in version 3.2.

Changed in version 3.10: Added *ignore_cleanup_errors* parameter.

Changed in version 3.12: Added the *delete* parameter.

tempfile.mkstemp(*suffix=None*, *prefix=None*, *dir=None*, *text=False*)

Creates a temporary file in the most secure manner possible. There are no race conditions in the file's creation, assuming that the platform properly implements the **`os.O_EXCL`** flag for **`os.open()`**. The file is readable and writable only by the creating user ID. If the platform uses permission bits to indicate whether a file is executable, the file is executable by no one.

The file descriptor is not inherited by child processes.

Unlike **`TemporaryFile()`**, the user of **`mkstemp()`** is responsible for deleting the temporary file when done with it.

If *suffix* is not `None`, the file name will end with that suffix, otherwise there will be no suffix. **`mkstemp()`** does not put a dot between the file name and the suffix; if you need one, put it at the beginning of *suffix*.

If *prefix* is not `None`, the file name will begin with that prefix; otherwise, a default prefix is used. The default is the return value of **`gettempprefix()`** or **`gettempprefixb()`**, as appropriate.

If *dir* is not `None`, the file will be created in that directory; otherwise, a default directory is used. The default directory is chosen from a platform-dependent list, but the user of the application can control the directory location by setting the *TMPDIR*, *TEMP* or *TMP* environment variables. There is thus no guarantee that the generated filename will have any nice properties, such as not requiring quoting when passed to external commands via `os.popen()`.

If any of *suffix*, *prefix*, and *dir* are not `None`, they must be the same type. If they are bytes, the returned name will be bytes instead of str. If you want to force a bytes return value with otherwise default behavior, pass `suffix=b''`.

If *text* is specified and true, the file is opened in text mode. Otherwise, (the default) the file is opened in binary mode.

`mkstemp()` returns a tuple containing an OS-level handle to an open file (as would be returned by **`os.open()`**) and the absolute pathname of that file, in that order.

Raises an auditing event `tempfile.mkstemp` with argument `fullpath`.

Changed in version 3.5: suffix, *prefix*, and *dir* may now be supplied in bytes in order to obtain a bytes return value. Prior to this, only str was allowed. *suffix* and *prefix* now accept and default to `None` to cause an appropriate default value to be used.

Changed in version 3.6: The *dir* parameter now accepts a path-like object.

tempfile.mkdtemp(*suffix=None*, *prefix=None*, *dir=None*)

Creates a temporary directory in the most secure manner possible. There are no race conditions in the directory’s creation. The directory is readable, writable, and searchable only by the creating user ID.

The user of **`mkdtemp()`** is responsible for deleting the temporary directory and its contents when done with it.

The *prefix*, *suffix*, and *dir* arguments are the same as for **`mkstemp()`**.

`mkdtemp()` returns the absolute pathname of the new directory.

Raises an auditing event `tempfile.mkdtemp` with argument `fullpath`.

Changed in version 3.5: suffix, *prefix*, and *dir* may now be supplied in bytes in order to obtain a bytes return value. Prior to this, only str was allowed. *suffix* and *prefix* now accept and default to `None` to cause an appropriate default value to be used.

Changed in version 3.6: The *dir* parameter now accepts a path-like object.

Changed in version 3.12: **`mkdtemp()`** now always returns an absolute path, even if *dir* is relative.

tempfile.gettempdir()

Return the name of the directory used for temporary files. This defines the default value for the *dir* argument to all functions in this module.

Python searches a standard list of directories to find one which the calling user can create files in. The list is:

1. The directory named by the **`TMPDIR`** environment variable.
2. The directory named by the **`TEMP`** environment variable.
3. The directory named by the **`TMP`** environment variable.
4. A platform-specific location:
    - On Windows, the directories `C:\TEMP`, `C:\TMP`, `\TEMP`, and `\TMP`, in that order.
    - On all other platforms, the directories `/tmp`, `/var/tmp`, and `/usr/tmp`, in that order.
5. As a last resort, the current working directory.

The result of this search is cached, see the description of **`tempdir`** below.

Changed in version 3.10: Always returns a str. Previously it would return any **`tempdir`** value regardless of type so long as it was not `None`.

tempfile.gettempdirb()

Same as **`gettempdir()`** but the return value is in bytes.

Added in version 3.5.

tempfile.gettempprefix()

Return the filename prefix used to create temporary files. This does not contain the directory component.

tempfile.gettempprefixb()

Same as **`gettempprefix()`** but the return value is in bytes.

Added in version 3.5.

The module uses a global variable to store the name of the directory used for temporary files returned by **`gettempdir()`**. It can be set directly to override the selection process, but this is discouraged. All functions in this module take a *dir* argument which can be used to specify the directory. This is the recommended approach that does not surprise other unsuspecting code by changing global API behavior.

tempfile.tempdir

When set to a value other than `None`, this variable defines the default value for the *dir* argument to the functions defined in this module, including its type, bytes or str. It cannot be a path-like object.

If `tempdir` is `None` (the default) at any call to any of the above functions except **`gettempprefix()`** it is initialized following the algorithm described in **`gettempdir()`**.

Note

Beware that if you set `tempdir` to a bytes value, there is a nasty side effect: The global default return type of **`mkstemp()`** and **`mkdtemp()`** changes to bytes when no explicit `prefix`, `suffix`, or `dir` arguments of type str are supplied. Please do not write code expecting or depending on this. This awkward behavior is maintained for compatibility with the historical implementation.

Examples

Here are some examples of typical usage of the **`tempfile`** module:

```
>>> import tempfile

# create a temporary file and write some data to it
>>> fp = tempfile.TemporaryFile()
>>> fp.write(b'Hello world!')
# read data from file
```

```
>>> fp.seek(0)
>>> fp.read()
b'Hello world!'
# close the file, it will be removed
>>> fp.close()

# create a temporary file using a context manager
>>> with tempfile.TemporaryFile() as fp:
...     fp.write(b'Hello world!')
...     fp.seek(0)
...     fp.read()
b'Hello world!'
>>>
# file is now closed and removed

# create a temporary file using a context manager
# close the file, use the name to open the file again
>>> with tempfile.NamedTemporaryFile(delete_on_close=False) as
fp:
...     fp.write(b'Hello world!')
...     fp.close()
... # the file is closed, but not removed
... # open the file again by using its name
...     with open(fp.name, mode='rb') as f:
...         f.read()
b'Hello world!'
>>>
# file is now removed

# create a temporary directory using the context manager
>>> with tempfile.TemporaryDirectory() as tmpdirname:
...     print('created temporary directory', tmpdirname)
>>>
# directory and contents have been removed
```

Deprecated functions and variables

A historical way to create temporary files was to first generate a file name with the `mktemp()` function and then create a file using this name. Unfortunately this is not secure, because a different process may create a file with this name in the time between the call to `mktemp()` and the subsequent attempt to create the file by the first process. The solution is to combine the two steps and create the file immediately. This approach is used by `mkstemp()` and the other functions described above.

tempfile.mktemp(*suffix*='', *prefix*='*tmp*', *dir*=*None*)

Deprecated since version 2.3: Use `mkstemp()` instead.

Return an absolute pathname of a file that did not exist at the time the call is made. The *prefix*, *suffix*, and *dir* arguments are similar to those of `mkstemp()`, except that bytes file names, `suffix=None` and `prefix=None` are not supported.

Warning

Use of this function may introduce a security hole in your program. By the time you get around to doing anything with the file name it returns, someone else may have beaten you to the punch. `mktemp()` usage can be replaced easily with `NamedTemporaryFile()`, passing it the `delete=False` parameter:

```
>>> f = NamedTemporaryFile(delete=False)
>>> f.name
'/tmp/tmptjujjt'
>>> f.write(b"Hello World!\n")
13
>>> f.close()
>>> os.unlink(f.name)
>>> os.path.exists(f.name)
False
```

`glob` — Unix style pathname pattern expansion

Source code: Lib/glob.py
[https://github.com/python/cpython/tree/3.14/Lib/glob.py]

The **`glob`** module finds pathnames using pattern matching rules similar to the Unix shell. No tilde expansion is done, but `*`, `?`, and character ranges expressed with `[]` will be correctly matched. This is done by using the **`os.scandir()`** and **`fnmatch.fnmatch()`** functions in concert, and not by actually invoking a subshell.

Note

The pathnames are returned in no particular order. If you need a specific order, sort the results.

Files beginning with a dot (`.`) can only be matched by patterns that also start with a dot, unlike **`fnmatch.fnmatch()`** or **`pathlib.Path.glob()`**. For tilde and shell variable expansion, use **`os.path.expanduser()`** and **`os.path.expandvars()`**.

For a literal match, wrap the meta-characters in brackets. For example, `'[?]'` matches the character `'?'`.

The **`glob`** module defines the following functions:

glob.glob(*pathname*, ***, *root_dir=None*, *dir_fd=None*, *recursive=False*, *include_hidden=False*)

Return a possibly empty list of path names that match *pathname*, which must be a string containing a path specification. *pathname* can be either absolute (like `/usr/src/Python-1.5/Makefile`) or relative (like `../../Tools/*/*.gif`), and can contain shell-style wildcards. Broken symlinks are included in the results (as in the shell). Whether or not the results are sorted depends on the file system. If a file that satisfies conditions is removed or added during the call of this function, whether a path name for that file will be included is unspecified.

If *root_dir* is not `None`, it should be a path-like object specifying the root directory for searching. It has the same effect on **`glob()`** as changing the current directory before calling it. If *pathname* is relative, the result will contain paths relative to *root_dir*.

This function can support paths relative to directory descriptors with the *dir_fd* parameter.

If *recursive* is true, the pattern "`**`" will match any files and zero or more directories, subdirectories and symbolic links to directories. If the pattern is followed by an **`os.sep`** or **`os.altsep`** then files will not match.

If *include_hidden* is true, "`**`" pattern will match hidden directories.

Raises an auditing event `glob.glob` with arguments `pathname`, `recursive`.

Raises an auditing event `glob.glob/2` with arguments `pathname`, `recursive`, `root_dir`, `dir_fd`.

Note

Using the “`**`” pattern in large directory trees may consume an inordinate amount of time.

Note

This function may return duplicate path names if *pathname* contains multiple “`**`” patterns and *recursive* is true.

Changed in version 3.5: Support for recursive globs using “`**`”.

Changed in version 3.10: Added the *root_dir* and *dir_fd* parameters.

Changed in version 3.11: Added the *include_hidden* parameter.

glob.iglob(*pathname*, *, *root_dir=None*, *dir_fd=None*, *recursive=False*, *include_hidden=False*)

Return an iterator which yields the same values as **`glob()`** without actually storing them all simultaneously.

Raises an auditing event `glob.glob` with arguments `pathname`, `recursive`.

Raises an auditing event `glob.glob/2` with arguments `pathname`, `recursive`, `root_dir`, `dir_fd`.

Note

This function may return duplicate path names if *pathname* contains multiple “`**`” patterns and *recursive* is true.

Changed in version 3.5: Support for recursive globs using “`**`”.

Changed in version 3.10: Added the *root_dir* and *dir_fd* parameters.

Changed in version 3.11: Added the *include_hidden* parameter.

glob.escape(*pathname*)

Escape all special characters (`'?'`, `'*'` and `'['`). This is useful if you want to match an arbitrary literal string that may have special characters in it. Special characters in drive/UNC sharepoints are not escaped, e.g. on Windows `escape('//?/c:/Quo vadis?.txt')` returns `'//?/c:/Quo vadis[?].txt'`.

Added in version 3.4.

glob.translate(*pathname*, *, *recursive=False*, *include_hidden=False*, *seps=None*)

Convert the given path specification to a regular expression for use with `re.match()`. The path specification can contain shell-style wildcards.

For example:

```
>>> import glob, re
>>>
>>> regex = glob.translate('**/*.txt', recursive=True,
include_hidden=True)
>>> regex
'(?s:(?:.+/)?[^/]*\\.txt)\\z'
>>> reobj = re.compile(regex)
>>> reobj.match('foo/bar/baz.txt')
<re.Match object; span=(0, 15), match='foo/bar/baz.txt'>
```

Path separators and segments are meaningful to this function, unlike **`fnmatch.translate()`**. By default wildcards do not match path separators, and `*` pattern segments match precisely one path segment.

If *recursive* is true, the pattern segment “`**`” will match any number of path segments.

If *include_hidden* is true, wildcards can match path segments that start with a dot (`.`).

A sequence of path separators may be supplied to the *seps* argument. If not given, **`os.sep`** and **`altsep`** (if available) are used.

See also

`pathlib.PurePath.full_match()` and **`pathlib.Path.glob()`** methods, which call this function to implement pattern matching and globbing.

Added in version 3.13.

Examples

Consider a directory containing the following files: `1.gif`, `2.txt`, `card.gif` and a subdirectory `sub` which contains only the file `3.txt`. **`glob()`** will produce the following results. Notice how any leading components of the path are preserved.

```
>>> import glob
>>> glob.glob('./[0-9].*')
['./1.gif', './2.txt']
>>> glob.glob('*.gif')
['1.gif', 'card.gif']
>>> glob.glob('?.gif')
['1.gif']
```

```
>>> glob.glob('**/*.txt', recursive=True)
['2.txt', 'sub/3.txt']
>>> glob.glob('./**/', recursive=True)
['./', './sub/']
```

If the directory contains files starting with `.` they won’t be matched by default. For example, consider a directory containing `card.gif` and `.card.gif`:

```
>>> import glob
>>> glob.glob('*.gif')
['card.gif']
>>> glob.glob('.c*')
['.card.gif']
```

See also

The **`fnmatch`** module offers shell-style filename (not path) expansion.

See also

The **`pathlib`** module offers high-level path objects.

`fnmatch` — Unix filename pattern matching

Source code: Lib/fnmatch.py [https://github.com/python/cpython/tree/3.14/Lib/fnmatch.py]

This module provides support for Unix shell-style wildcards, which are *not* the same as regular expressions (which are documented in the **`re`** module). The special characters used in shell-style wildcards are:

Pattern	**Meaning**
`*`	matches everything
`?`	matches any single character
`[seq]`	matches any character in *seq*
`[!seq]`	matches any character not in *seq*

For a literal match, wrap the meta-characters in brackets. For example, `'[?]'` matches the character `'?'`.

Note that the filename separator (`'/'` on Unix) is *not* special to this module. See module **`glob`** for pathname expansion (**`glob`** uses **`filter()`** to match pathname segments). Similarly, filenames starting with a period are not special for this module, and are matched by the `*` and `?` patterns.

Unless stated otherwise, “filename string” and “pattern string” either refer to `str` or `ISO-8859-1` encoded `bytes` objects. Note that the functions documented below do not allow to mix a **`bytes`** pattern with a **`str`** filename, and vice-versa.

Finally, note that `functools.lru_cache()` with a *maxsize* of 32768 is used to cache the (typed) compiled regex patterns in the following functions: `fnmatch()`, `fnmatchcase()`, `filter()`, `filterfalse()`.

fnmatch.fnmatch(*name*, *pat*)

Test whether the filename string *name* matches the pattern string *pat*, returning `True` or `False`. Both parameters are case-normalized using `os.path.normcase()`. `fnmatchcase()` can be used to perform a case-sensitive comparison, regardless of whether that’s standard for the operating system.

This example will print all file names in the current directory with the extension `.txt`:

```
import fnmatch
import os

for file in os.listdir('.'):
    if fnmatch.fnmatch(file, '*.txt'):
        print(file)
```

fnmatch.fnmatchcase(*name*, *pat*)

Test whether the filename string *name* matches the pattern string *pat*, returning `True` or `False`; the comparison is case-sensitive and does not apply `os.path.normcase()`.

fnmatch.filter(*names*, *pat*)

Construct a list from those elements of the iterable of filename strings *names* that match the pattern string *pat*. It is the same as `[n for n in names if fnmatch(n, pat)]`, but implemented more efficiently.

fnmatch.filterfalse(*names*, *pat*)

Construct a list from those elements of the iterable of filename strings *names* that do not match the pattern string *pat*. It is the same as `[n for n in names if not fnmatch(n, pat)]`, but implemented more efficiently.

Added in version 3.14.

fnmatch.translate(*pat*)

Return the shell-style pattern *pat* converted to a regular expression for using with **`re.match()`**. The pattern is expected to be a **`str`**.

Example:

```
>>> import fnmatch, re
>>>
>>> regex = fnmatch.translate('*.txt')
>>> regex
'(?s:.*\\.txt)\\z'
>>> reobj = re.compile(regex)
>>> reobj.match('foobar.txt')
<re.Match object; span=(0, 10), match='foobar.txt'>
```

See also

Module `glob`

Unix shell-style path expansion.

`linecache` — Random access to text lines

Source code: Lib/linecache.py
[https://github.com/python/cpython/tree/3.14/Lib/linecache.py]

The **`linecache`** module allows one to get any line from a Python source file, while attempting to optimize internally, using a cache, the common case where many lines are read from a single file. This is used by the **`traceback`** module to retrieve source lines for inclusion in the formatted traceback.

The **`tokenize.open()`** function is used to open files. This function uses **`tokenize.detect_encoding()`** to get the encoding of the file; in the absence of an encoding token, the file encoding defaults to UTF-8.

The **`linecache`** module defines the following functions:

linecache.getline(*filename*, *lineno*, *module_globals=None*)

Get line *lineno* from file named *filename*. This function will never raise an exception — it will return `''` on errors (the terminating newline character will be included for lines that are found).

If *filename* indicates a frozen module (starting with `'<frozen '`), the function will attempt to get the real file name from `module_globals['__file__']` if *module_globals* is not `None`.

If a file named *filename* is not found, the function first checks for a **PEP 302** [https://peps.python.org/pep-0302/] `__loader__` in *module_globals*. If there is such a loader and it defines a `get_source` method, then that determines the source lines (if `get_source()` returns `None`, then `''` is returned). Finally, if *filename* is a relative filename, it is looked up relative to the entries in the module search path, `sys.path`.

Changed in version 3.14: Support *filename* of frozen modules.

linecache.clearcache()

Clear the cache. Use this function if you no longer need lines from files previously read using `getline()`.

linecache.checkcache(*filename=None*)

Check the cache for validity. Use this function if files in the cache may have changed on disk, and you require the updated version. If *filename* is omitted, it will check all the entries in the cache.

linecache.lazycache(*filename*, *module_globals*)

Capture enough detail about a non-file-based module to permit getting its lines later via `getline()` even if *module_globals* is `None` in the later call. This avoids doing I/O until a line is actually needed, without having to carry the module globals around indefinitely.

Added in version 3.5.

Example:

```
>>> import linecache
>>> linecache.getline(linecache.__file__, 8)
'import sys\n'
```

`shutil` — High-level file operations

Source code: Lib/shutil.py [https://github.com/python/cpython/tree/3.14/Lib/shutil.py]

The `shutil` module offers a number of high-level operations on files and collections of files. In particular, functions are provided which support file copying and removal. For operations on individual files, see also the `os` module.

Warning

Even the higher-level file copying functions (`shutil.copy()`, `shutil.copy2()`) cannot copy all file metadata.

On POSIX platforms, this means that file owner and group are lost as well as ACLs. On Mac OS, the resource fork and other metadata are not used. This means that resources will be lost and file type and creator codes will not be correct. On Windows, file owners, ACLs and alternate data streams are not copied.

Directory and files operations

shutil.copyfileobj(*fsrc*, *fdst*[, *length*])

Copy the contents of the file-like object *fsrc* to the file-like object *fdst*. The integer *length*, if given, is the buffer size. In particular, a negative *length* value means to copy the data without looping over the source data in chunks;

by default the data is read in chunks to avoid uncontrolled memory consumption. Note that if the current file position of the *fsrc* object is not 0, only the contents from the current file position to the end of the file will be copied.

`copyfileobj()` will *not* guarantee that the destination stream has been flushed on completion of the copy. If you want to read from the destination at the completion of the copy operation (for example, reading the contents of a temporary file that has been copied from a HTTP stream), you must ensure that you have called **`flush()`** or **`close()`** on the file-like object before attempting to read the destination file.

shutil.copyfile(*src*, *dst*, *, *follow_symlinks=True*)

Copy the contents (no metadata) of the file named *src* to a file named *dst* and return *dst* in the most efficient way possible. *src* and *dst* are path-like objects or path names given as strings.

dst must be the complete target file name; look at **`copy()`** for a copy that accepts a target directory path. If *src* and *dst* specify the same file, **`SameFileError`** is raised.

The destination location must be writable; otherwise, an **`OSError`** exception will be raised. If *dst* already exists, it will be replaced. Special files such as character or block devices and pipes cannot be copied with this function.

If *follow_symlinks* is false and *src* is a symbolic link, a new symbolic link will be created instead of copying the file *src* points to.

Raises an auditing event `shutil.copyfile` with arguments `src`, `dst`.

Changed in version 3.3: `IOError` used to be raised instead of `OSError`. Added *follow_symlinks* argument. Now returns *dst*.

Changed in version 3.4: Raise `SameFileError` instead of `Error`. Since the former is a subclass of the latter, this change is backward compatible.

Changed in version 3.8: Platform-specific fast-copy syscalls may be used internally in order to copy the file more efficiently. See Platform-dependent efficient copy operations section.

exception shutil.SpecialFileError

This exception is raised when `copyfile()` or `copytree()` attempt to copy a named pipe.

Added in version 2.7.

exception shutil.SameFileError

This exception is raised if source and destination in `copyfile()` are the same file.

Added in version 3.4.

shutil.copymode(*src*, *dst*, *, *follow_symlinks=True*)

Copy the permission bits from *src* to *dst*. The file contents, owner, and group are unaffected. *src* and *dst* are path-like objects or path names given as strings. If *follow_symlinks* is false, and both *src* and *dst* are symbolic links, `copymode()` will attempt to modify the mode of *dst* itself (rather than the file it points to). This functionality is not available on every platform; please see `copystat()` for more information. If `copymode()` cannot modify symbolic links on the local platform, and it is asked to do so, it will do nothing and return.

Raises an auditing event `shutil.copymode` with arguments `src`, `dst`.

Changed in version 3.3: Added *follow_symlinks* argument.

shutil.copystat(*src*, *dst*, ***, *follow_symlinks=True*)

Copy the permission bits, last access time, last modification time, and flags from *src* to *dst*. On Linux, **`copystat()`** also copies the "extended attributes" where possible. The file contents, owner, and group are unaffected. *src* and *dst* are path-like objects or path names given as strings.

If *follow_symlinks* is false, and *src* and *dst* both refer to symbolic links, **`copystat()`** will operate on the symbolic links themselves rather than the files the symbolic links refer to—reading the information from the *src* symbolic link, and writing the information to the *dst* symbolic link.

Note

Not all platforms provide the ability to examine and modify symbolic links. Python itself can tell you what functionality is locally available.

- If `os.chmod in os.supports_follow_symlinks` is `True`, **`copystat()`** can modify the permission bits of a symbolic link.
- If `os.utime in os.supports_follow_symlinks` is `True`, **`copystat()`** can modify the last access and modification times of a symbolic link.
- If `os.chflags in os.supports_follow_symlinks` is `True`, **`copystat()`** can modify the flags of a symbolic link. (`os.chflags` is not available on all platforms.)

On platforms where some or all of this functionality is unavailable, when asked to modify a symbolic link, `copystat()` will copy everything it can. `copystat()` never returns failure.

Please see `os.supports_follow_symlinks` for more information.

Raises an auditing event `shutil.copystat` with arguments `src`, `dst`.

Changed in version 3.3: Added *follow_symlinks* argument and support for Linux extended attributes.

shutil.copy(*src*, *dst*, ***, *follow_symlinks=True*)

Copies the file *src* to the file or directory *dst*. *src* and *dst* should be path-like objects or strings. If *dst* specifies a directory, the file will be copied into *dst* using the base filename from *src*. If *dst* specifies a file that already exists, it will be replaced. Returns the path to the newly created file.

If *follow_symlinks* is false, and *src* is a symbolic link, *dst* will be created as a symbolic link. If *follow_symlinks* is true and *src* is a symbolic link, *dst* will be a copy of the file *src* refers to.

`copy()` copies the file data and the file’s permission mode (see `os.chmod()`). Other metadata, like the file’s creation and modification times, is not preserved. To preserve all file metadata from the original, use `copy2()` instead.

Raises an auditing event `shutil.copyfile` with arguments `src`, `dst`.

Raises an auditing event `shutil.copymode` with arguments `src`, `dst`.

Changed in version 3.3: Added *follow_symlinks* argument. Now returns path to the newly created file.

Changed in version 3.8: Platform-specific fast-copy syscalls may be used internally in order to copy the file more efficiently. See Platform-dependent efficient copy operations section.

shutil.copy2(*src*, *dst*, ***, *follow_symlinks=True*)

Identical to **`copy()`** except that **`copy2()`** also attempts to preserve file metadata.

When *follow_symlinks* is false, and *src* is a symbolic link, **`copy2()`** attempts to copy all metadata from the *src* symbolic link to the newly created *dst* symbolic link. However, this functionality is not available on all platforms. On platforms where some or all of this functionality is unavailable, **`copy2()`** will preserve all the metadata it can; **`copy2()`** never raises an exception because it cannot preserve file metadata.

`copy2()` uses **`copystat()`** to copy the file metadata. Please see **`copystat()`** for more information about platform support for modifying symbolic link metadata.

Raises an auditing event `shutil.copyfile` with arguments `src`, `dst`.

Raises an auditing event `shutil.copystat` with arguments `src`, `dst`.

Changed in version 3.3: Added *follow_symlinks* argument, try to copy extended file system attributes too (currently Linux only). Now returns path to the newly created file.

Changed in version 3.8: Platform-specific fast-copy syscalls may be used internally in order to copy the file more efficiently. See Platform-dependent efficient copy operations section.

shutil.ignore_patterns(**patterns*)

This factory function creates a function that can be used as a callable for `copytree()`'s *ignore* argument, ignoring files and directories that match one of the glob-style *patterns* provided. See the example below.

shutil.copytree(*src*, *dst*, *symlinks=False*, *ignore=None*, *copy_function=copy2*, *ignore_dangling_symlinks=False*, *dirs_exist_ok=False*)

Recursively copy an entire directory tree rooted at *src* to a directory named *dst* and return the destination directory. All intermediate directories needed to contain *dst* will also be created by default.

Permissions and times of directories are copied with `copystat()`, individual files are copied using `copy2()`.

If *symlinks* is true, symbolic links in the source tree are represented as symbolic links in the new tree and the metadata of the original links will be copied as far as the platform allows; if false or omitted, the contents and metadata of the linked files are copied to the new tree.

When *symlinks* is false, if the file pointed to by the symlink doesn’t exist, an exception will be added in the list of errors raised in an `Error` exception at the end of the copy process. You can set the optional *ignore_dangling_symlinks* flag to true if you want to silence this exception. Notice that this option has no effect on platforms that don’t support `os.symlink()`.

If *ignore* is given, it must be a callable that will receive as its arguments the directory being visited by `copytree()`, and a list of its contents, as returned by `os.listdir()`. Since `copytree()` is called recursively, the *ignore* callable will be called once for each directory that is copied. The callable must return a sequence of directory and file names relative to the current directory (i.e. a subset of the items in its second argument); these names will then be ignored in the copy process. `ignore_patterns()` can be used to create such a callable that ignores names based on glob-style patterns.

If exception(s) occur, an `Error` is raised with a list of reasons.

If *copy_function* is given, it must be a callable that will be used to copy each file. It will be called with the source path and the destination path as arguments. By default, `copy2()` is used, but any function that supports the same signature (like `copy()`) can be used.

If *dirs_exist_ok* is false (the default) and *dst* already exists, a `FileExistsError` is raised. If *dirs_exist_ok* is true, the copying operation will continue if it encounters existing directories, and files within the *dst* tree will be overwritten by corresponding files from the *src* tree.

Raises an auditing event `shutil.copytree` with arguments `src`, `dst`.

Changed in version 3.2: Added the *copy_function* argument to be able to provide a custom copy function. Added the *ignore_dangling_symlinks* argument to silence dangling symlinks errors when *symlinks* is false.

Changed in version 3.3: Copy metadata when *symlinks* is false. Now returns *dst*.

Changed in version 3.8: Platform-specific fast-copy syscalls may be used internally in order to copy the file more efficiently. See Platform-dependent efficient copy operations section.

Changed in version 3.8: Added the *dirs_exist_ok* parameter.

shutil.rmtree(*path*, *ignore_errors=False*, *onerror=None*, ***, *onexc=None*, *dir_fd=None*)

Delete an entire directory tree; *path* must point to a directory (but not a symbolic link to a directory). If *ignore_errors* is true, errors resulting from failed removals will be ignored; if false or omitted, such errors are handled by calling a handler specified by *onexc* or *onerror* or, if both are omitted, exceptions are propagated to the caller.

This function can support paths relative to directory descriptors.

Note

On platforms that support the necessary fd-based functions a symlink attack resistant version of `rmtree()` is used by default. On other platforms, the `rmtree()` implementation is susceptible to a symlink attack: given proper timing and circumstances, attackers can manipulate symlinks on the filesystem to delete files they wouldn’t be able to access otherwise. Applications can use the `rmtree.avoids_symlink_attacks` function attribute to determine which case applies.

If *onexc* is provided, it must be a callable that accepts three parameters: *function*, *path*, and *excinfo*.

The first parameter, *function*, is the function which raised the exception; it depends on the platform and implementation. The second parameter, *path*, will be the path name passed to *function*. The third parameter, *excinfo*, is the exception that was raised. Exceptions raised by *onexc* will not be caught.

The deprecated *onerror* is similar to *onexc*, except that the third parameter it receives is the tuple returned from **`sys.exc_info()`**.

See also

rmtree example for an example of handling the removal of a directory tree that contains read-only files.

Raises an auditing event `shutil.rmtree` with arguments `path`, `dir_fd`.

Changed in version 3.3: Added a symlink attack resistant version that is used automatically if platform supports fd-based functions.

Changed in version 3.8: On Windows, will no longer delete the contents of a directory junction before removing the junction.

Changed in version 3.11: Added the *dir_fd* parameter.

Changed in version 3.12: Added the *onexc* parameter, deprecated *onerror*.

Changed in version 3.13: **`rmtree()`** now ignores **`FileNotFoundError`** exceptions for all but the top-level path.

Exceptions other than `OSError` and subclasses of **`OSError`** are now always propagated to the caller.

rmtree.avoids_symlink_attacks

Indicates whether the current platform and implementation provides a symlink attack resistant version of `rmtree()`. Currently this is only true for platforms supporting fd-based directory access functions.

Added in version 3.3.

shutil.move(*src*, *dst*, *copy_function=copy2*)

Recursively move a file or directory (*src*) to another location and return the destination.

If *dst* is an existing directory or a symlink to a directory, then *src* is moved inside that directory. The destination path in that directory must not already exist.

If *dst* already exists but is not a directory, it may be overwritten depending on `os.rename()` semantics.

If the destination is on the current filesystem, then `os.rename()` is used. Otherwise, *src* is copied to the destination using *copy_function* and then removed. In case of symlinks, a new symlink pointing to the target of *src* will be created as the destination and *src* will be removed.

If *copy_function* is given, it must be a callable that takes two arguments, *src* and the destination, and will be used to copy *src* to the destination if `os.rename()` cannot be used. If the source is a directory, `copytree()` is called, passing it the *copy_function*. The default *copy_function* is `copy2()`. Using `copy()` as the *copy_function* allows the move to succeed when it is not possible to also copy the metadata, at the expense of not copying any of the metadata.

Raises an auditing event `shutil.move` with arguments `src`, `dst`.

Changed in version 3.3: Added explicit symlink handling for foreign filesystems, thus adapting it to the behavior of GNU's **mv**. Now returns *dst*.

Changed in version 3.5: Added the *copy_function* keyword argument.

Changed in version 3.8: Platform-specific fast-copy syscalls may be used internally in order to copy the file more efficiently. See Platform-dependent efficient copy operations section.

Changed in version 3.9: Accepts a path-like object for both *src* and *dst*.

shutil.disk_usage(*path*)

Return disk usage statistics about the given path as a named tuple with the attributes *total*, *used* and *free*, which are the amount of total, used and free space, in bytes. *path* may be a file or a directory.

Note

On Unix filesystems, *path* must point to a path within a **mounted** filesystem partition. On those platforms, CPython doesn't attempt to retrieve disk usage information from non-mounted filesystems.

Added in version 3.3.

Changed in version 3.8: On Windows, *path* can now be a file or directory.

Availability: Unix, Windows.

shutil.chown(*path*, *user=None*, *group=None*, ***, *dir_fd=None*, *follow_symlinks=True*)

Change owner *user* and/or *group* of the given *path*.

user can be a system user name or a uid; the same applies to *group*. At least one argument is required.

See also **`os.chown()`**, the underlying function.

Raises an auditing event `shutil.chown` with arguments `path`, `user`, `group`.

Availability: Unix.

Added in version 3.3.

Changed in version 3.13: Added *dir_fd* and *follow_symlinks* parameters.

shutil.which(*cmd*, *mode=os.F_OK | os.X_OK*, *path=None*)

Return the path to an executable which would be run if the given *cmd* was called. If no *cmd* would be called, return `None`.

mode is a permission mask passed to **`os.access()`**, by default determining if the file exists and is executable.

path is a “`PATH` string” specifying the directories to look in, delimited by **`os.pathsep`**. When no *path* is specified, the **`PATH`** environment variable is read from **`os.environ`**, falling back to **`os.defpath`** if it is not set.

If *cmd* contains a directory component, **`which()`** only checks the specified path directly and does not search the directories listed in *path* or in the system's **`PATH`** environment variable.

On Windows, the current directory is prepended to the *path* if *mode* does not include `os.X_OK`. When the *mode* does include `os.X_OK`, the Windows API `NeedCurrentDirectoryForExePathW` will be consulted to determine if the current directory should be prepended to *path*. To avoid consulting the current working directory for executables: set the environment variable `NoDefaultCurrentDirectoryInExePath`.

Also on Windows, the **`PATHEXT`** environment variable is used to resolve commands that may not already include an extension. For example, if you call `shutil.which("python")`, `which()` will search `PATHEXT` to know that it should look for `python.exe` within the *path* directories. For example, on Windows:

```
>>> shutil.which("python")
'C:\\Python33\\python.EXE'
```

This is also applied when *cmd* is a path that contains a directory component:

```
>>> shutil.which("C:\\Python33\\python")
'C:\\Python33\\python.EXE'
```

Added in version 3.3.

Changed in version 3.8: The `bytes` type is now accepted. If *cmd* type is `bytes`, the result type is also `bytes`.

Changed in version 3.12: On Windows, the current directory is no longer prepended to the search path if *mode* includes `os.X_OK` and WinAPI

`NeedCurrentDirectoryForExePathW(cmd)` is false, else the current directory is prepended even if it is already in the search path; `PATHEXT` is used now even when *cmd* includes a directory component or ends with an extension that is in `PATHEXT`; and filenames that have no extension can now be found.

exception shutil.Error

This exception collects exceptions that are raised during a multi-file operation. For **`copytree()`**, the exception argument is a list of 3-tuples (*srcname*, *dstname*, *exception*).

Platform-dependent efficient copy operations

Starting from Python 3.8, all functions involving a file copy (**`copyfile()`**, **`copy()`**, **`copy2()`**, **`copytree()`**, and **`move()`**) may use platform-specific “fast-copy” syscalls in order to copy the file more efficiently (see bpo-33671 [https://bugs.python.org/issue?@action=redirect&bpo=33671]). “fast-copy” means that the copying operation occurs within the kernel, avoiding the use of userspace buffers in Python as in “`outfd.write(infd.read())`”.

On macOS fcopyfile [http://www.manpagez.com/man/3/copyfile/] is used to copy the file content (not metadata).

On Linux **`os.copy_file_range()`** or **`os.sendfile()`** is used.

On Solaris **`os.sendfile()`** is used.

On Windows **`shutil.copyfile()`** uses a bigger default buffer size (1 MiB instead of 64 KiB) and a **`memoryview()`**-based variant of **`shutil.copyfileobj()`** is used.

If the fast-copy operation fails and no data was written in the destination file then shutil will silently fallback on using less efficient `copyfileobj()` function internally.

Changed in version 3.8.

Changed in version 3.14: Solaris now uses `os.sendfile()`.

Changed in version 3.14: Copy-on-write or server-side copy may be used internally via `os.copy_file_range()` on supported Linux filesystems.

copytree example

An example that uses the `ignore_patterns()` helper:

```
from shutil import copytree, ignore_patterns

copytree(source, destination, ignore=ignore_patterns('*.pyc',
'tmp*'))
```

This will copy everything except `.pyc` files and files or directories whose name starts with `tmp`.

Another example that uses the *ignore* argument to add a logging call:

```
from shutil import copytree
import logging

def _logpath(path, names):
    logging.info('Working in %s', path)
    return []   # nothing will be ignored

copytree(source, destination, ignore=_logpath)
```

rmtree example

This example shows how to remove a directory tree on Windows where some of the files have their read-only bit set. It uses the onexc callback to clear the readonly bit and reattempt the remove. Any subsequent failure will propagate.

```
import os, stat
import shutil

def remove_readonly(func, path, _):
    "Clear the readonly bit and reattempt the removal"
    os.chmod(path, stat.S_IWRITE)
    func(path)

shutil.rmtree(directory, onexc=remove_readonly)
```

Archiving operations

Added in version 3.2.

Changed in version 3.5: Added support for the *xztar* format.

High-level utilities to create and read compressed and archived files are also provided. They rely on the `zipfile` and `tarfile` modules.

shutil.make_archive(*base_name*, *format*[, *root_dir*[, *base_dir*[, *verbose*[, *dry_run*[, *owner*[, *group*[, *logger*]]]]]]])

Create an archive file (such as zip or tar) and return its name.

base_name is the name of the file to create, including the path, minus any format-specific extension.

format is the archive format: one of “zip” (if the **`zlib`** module is available), “tar”, “gztar” (if the **`zlib`** module is available), “bztar” (if the **`bz2`** module is available), “xztar” (if the **`lzma`** module is available), or “zstdtar” (if the **`compression.zstd`** module is available).

root_dir is a directory that will be the root directory of the archive, all paths in the archive will be relative to it; for example, we typically chdir into *root_dir* before creating the archive.

base_dir is the directory where we start archiving from; i.e. *base_dir* will be the common prefix of all files and directories in the archive. *base_dir* must be given relative to *root_dir*. See Archiving example with base_dir for how to use *base_dir* and *root_dir* together.

root_dir and *base_dir* both default to the current directory.

If *dry_run* is true, no archive is created, but the operations that would be executed are logged to *logger*.

owner and *group* are used when creating a tar archive. By default, uses the current owner and group.

logger must be an object compatible with **PEP 282** [https://peps.python.org/pep-0282/], usually an instance of **`logging.Logger`**.

The *verbose* argument is unused and deprecated.

Raises an auditing event `shutil.make_archive` with arguments `base_name`, `format`, `root_dir`, `base_dir`.

Note

This function is not thread-safe when custom archivers registered with `register_archive_format()` do not support the *root_dir* argument. In this case it temporarily changes the current working directory of the process to *root_dir* to perform archiving.

Changed in version 3.8: The modern pax (POSIX.1-2001) format is now used instead of the legacy GNU format for archives created with `format="tar"`.

Changed in version 3.10.6: This function is now made thread-safe during creation of standard `.zip` and tar archives.

shutil.get_archive_formats()

Return a list of supported formats for archiving. Each element of the returned sequence is a tuple `(name, description)`.

By default `shutil` provides these formats:

- *zip*: ZIP file (if the `zlib` module is available).
- *tar*: Uncompressed tar file. Uses POSIX.1-2001 pax format for new archives.
- *gztar*: gzip’ed tar-file (if the `zlib` module is available).
- *bztar*: bzip2’ed tar-file (if the `bz2` module is available).
- *xztar*: xz’ed tar-file (if the `lzma` module is available).
- *zstdtar*: Zstandard compressed tar-file (if the `compression.zstd` module is available).

You can register new formats or provide your own archiver for any existing formats, by using `register_archive_format()`.

shutil.register_archive_format(*name*, *function*[, *extra_args*[, *description*]])

Register an archiver for the format *name*.

function is the callable that will be used to unpack archives. The callable will receive the *base_name* of the file to create, followed by the *base_dir* (which defaults to `os.curdir`) to start archiving from. Further arguments are passed as keyword arguments: *owner*, *group*, *dry_run* and *logger* (as passed in `make_archive()`).

If *function* has the custom attribute `function.supports_root_dir` set to `True`, the *root_dir* argument is passed as a keyword argument. Otherwise the current working directory of the process is temporarily changed to *root_dir* before calling *function*. In this case `make_archive()` is not thread-safe.

If given, *extra_args* is a sequence of `(name, value)` pairs that will be used as extra keywords arguments when the archiver callable is used.

description is used by `get_archive_formats()` which returns the list of archivers. Defaults to an empty string.

Changed in version 3.12: Added support for functions supporting the *root_dir* argument.

shutil.unregister_archive_format(*name*)

Remove the archive format *name* from the list of supported formats.

shutil.unpack_archive(*filename*[, *extract_dir*[, *format*[, *filter*]]])

Unpack an archive. *filename* is the full path of the archive.

extract_dir is the name of the target directory where the archive is unpacked. If not provided, the current working directory is used.

format is the archive format: one of “zip”, “tar”, “gztar”, “bztar”, “xztar”, or “zstdtar”. Or any other format registered with **`register_unpack_format()`**. If not provided, **`unpack_archive()`** will use the archive file name extension and see if an unpacker was registered for that extension. In case none is found, a **`ValueError`** is raised.

The keyword-only *filter* argument is passed to the underlying unpacking function. For zip files, *filter* is not accepted. For tar files, it is recommended to use `'data'` (default since Python 3.14), unless using features specific to tar and UNIX-like filesystems. (See Extraction filters for details.)

Raises an auditing event `shutil.unpack_archive` with arguments `filename`, `extract_dir`, `format`.

Warning

Never extract archives from untrusted sources without prior inspection. It is possible that files are created outside of the path specified in the *extract_dir* argument, e.g. members that have absolute filenames starting with “/” or filenames with two dots “..”.

Since Python 3.14, the defaults for both built-in formats (zip and tar files) will prevent the most dangerous of such security issues, but will not prevent *all* unintended behavior. Read the Hints for further verification section for tar-specific details.

Changed in version 3.7: Accepts a path-like object for *filename* and *extract_dir*.

Changed in version 3.12: Added the *filter* argument.

shutil.register_unpack_format(*name*, *extensions*, *function*[, *extra_args*[, *description*]])

Registers an unpack format. *name* is the name of the format and *extensions* is a list of extensions corresponding to the format, like `.zip` for Zip files.

function is the callable that will be used to unpack archives. The callable will receive:

- the path of the archive, as a positional argument;
- the directory the archive must be extracted to, as a positional argument;
- possibly a *filter* keyword argument, if it was given to **`unpack_archive()`**;
- additional keyword arguments, specified by *extra_args* as a sequence of `(name, value)` tuples.

description can be provided to describe the format, and will be returned by the **`get_unpack_formats()`** function.

shutil.unregister_unpack_format(*name*)

Unregister an unpack format. *name* is the name of the format.

shutil.get_unpack_formats()

Return a list of all registered formats for unpacking. Each element of the returned sequence is a tuple `(name,` `extensions, description)`.

By default `shutil` provides these formats:

- *zip*: ZIP file (unpacking compressed files works only if the corresponding module is available).
- *tar*: uncompressed tar file.
- *gztar*: gzip’ed tar-file (if the `zlib` module is available).
- *bztar*: bzip2’ed tar-file (if the `bz2` module is available).
- *xztar*: xz’ed tar-file (if the `lzma` module is available).
- *zstdtar*: Zstandard compressed tar-file (if the `compression.zstd` module is available).

You can register new formats or provide your own unpacker for any existing formats, by using `register_unpack_format()`.

Archiving example

In this example, we create a gzip’ed tar-file archive containing all files found in the `.ssh` directory of the user:

```
>>> from shutil import make_archive
>>> import os
>>> archive_name = os.path.expanduser(os.path.join('~',
'myarchive'))
>>> root_dir = os.path.expanduser(os.path.join('~', '.ssh'))
>>> make_archive(archive_name, 'gztar', root_dir)
'/Users/tarek/myarchive.tar.gz'
```

The resulting archive contains:

```
$ tar -tzvf /Users/tarek/myarchive.tar.gz
drwx------ tarek/staff       0 2010-02-01 16:23:40 ./
-rw-r--r-- tarek/staff     609 2008-06-09 13:26:54
./authorized_keys
-rwxr-xr-x tarek/staff      65 2008-06-09 13:26:54 ./config
-rwx------ tarek/staff     668 2008-06-09 13:26:54 ./id_dsa
-rwxr-xr-x tarek/staff     609 2008-06-09 13:26:54 ./id_dsa.pub
```

```
-rw------- tarek/staff    1675 2008-06-09 13:26:54 ./id_rsa
-rw-r--r-- tarek/staff     397 2008-06-09 13:26:54 ./id_rsa.pub
-rw-r--r-- tarek/staff   37192 2010-02-06 18:23:10
./known_hosts
```

Archiving example with *base_dir*

In this example, similar to the one above, we show how to use `make_archive()`, but this time with the usage of *base_dir*. We now have the following directory structure:

```
$ tree tmp
tmp
└── root
    └── structure
        ├── content
            └── please_add.txt
        └── do_not_add.txt
```

In the final archive, `please_add.txt` should be included, but `do_not_add.txt` should not. Therefore we use the following:

```
>>> from shutil import make_archive
>>> import os
>>> archive_name = os.path.expanduser(os.path.join('~',
'myarchive'))
>>> make_archive(
...     archive_name,
...     'tar',
...     root_dir='tmp/root',
...     base_dir='structure/content',
... )
'/Users/tarek/myarchive.tar'
```

Listing the files in the resulting archive gives us:

```
$ python -m tarfile -l /Users/tarek/myarchive.tar
structure/content/
structure/content/please_add.txt
```

Querying the size of the output terminal

shutil.get_terminal_size(*fallback=(columns, lines)*)

Get the size of the terminal window.

For each of the two dimensions, the environment variable, `COLUMNS` and `LINES` respectively, is checked. If the variable is defined and the value is a positive integer, it is used.

When `COLUMNS` or `LINES` is not defined, which is the common case, the terminal connected to **`sys.__stdout__`** is queried by invoking **`os.get_terminal_size()`**.

If the terminal size cannot be successfully queried, either because the system doesn't support querying, or because we are not connected to a terminal, the value given in `fallback` parameter is used. `fallback` defaults to `(80, 24)` which is the default size used by many terminal emulators.

The value returned is a named tuple of type **`os.terminal_size`**.

See also: The Single UNIX Specification, Version 2, Other Environment Variables [https://pubs.opengroup.org/onlinepubs/7908799/xbd/envvar.html#tag_002_003].

Added in version 3.3.

Changed in version 3.11: The `fallback` values are also used if **`os.get_terminal_size()`** returns zeroes.

Data Persistence

The modules described in this chapter support storing Python data in a persistent form on disk. The `pickle` and `marshal` modules can turn many Python data types into a stream of bytes and then recreate the objects from the bytes. The various DBM-related modules support a family of hash-based file formats that store a mapping of strings to other strings.

The list of modules described in this chapter is:

- `pickle` — Python object serialization
    - Relationship to other Python modules
        - Comparison with `marshal`
        - Comparison with `json`
    - Data stream format
    - Module Interface
    - What can be pickled and unpickled?
    - Pickling Class Instances
        - Persistence of External Objects
        - Dispatch Tables
        - Handling Stateful Objects
    - Custom Reduction for Types, Functions, and Other Objects
    - Out-of-band Buffers
        - Provider API
        - Consumer API
        - Example
    - Restricting Globals
    - Performance
    - Examples
    - Command-line interface
- `copyreg` — Register `pickle` support functions

  - Example
- **`shelve`** — Python object persistence
  - Restrictions
  - Example
- **`marshal`** — Internal Python object serialization
- **`dbm`** — Interfaces to Unix “databases”
  - **`dbm.sqlite3`** — SQLite backend for dbm
  - **`dbm.gnu`** — GNU database manager
  - **`dbm.ndbm`** — New Database Manager
  - **`dbm.dumb`** — Portable DBM implementation
- **`sqlite3`** — DB-API 2.0 interface for SQLite databases
  - Tutorial
  - Reference
    - Module functions
    - Module constants
    - Connection objects
    - Cursor objects
    - Row objects
    - Blob objects
    - PrepareProtocol objects
    - Exceptions
    - SQLite and Python types
    - Default adapters and converters (deprecated)
    - Command-line interface
  - How-to guides
    - How to use placeholders to bind values in SQL queries
    - How to adapt custom Python types to SQLite values
      - How to write adaptable objects
      - How to register adapter callables
    - How to convert SQLite values to custom Python types
    - Adapter and converter recipes
    - How to use connection shortcut methods
    - How to use the connection context manager

    - How to work with SQLite URIs
    - How to create and use row factories
    - How to handle non-UTF-8 text encodings
- Explanation
    - Transaction control
        - Transaction control via the **`autocommit`** attribute
        - Transaction control via the **`isolation_level`** attribute

`pickle` — Python object serialization

Source code: Lib/pickle.py
[https://github.com/python/cpython/tree/3.14/Lib/pickle.py]

The **`pickle`** module implements binary protocols for serializing and de-serializing a Python object structure. *“Pickling”* is the process whereby a Python object hierarchy is converted into a byte stream, and *“unpickling”* is the inverse operation, whereby a byte stream (from a binary file or bytes-like object) is converted back into an object hierarchy. Pickling (and unpickling) is alternatively known as “serialization”, “marshalling,” [[1]] or “flattening”; however, to avoid confusion, the terms used here are “pickling” and “unpickling”.

Warning

The `pickle` module **is not secure**. Only unpickle data you trust.

It is possible to construct malicious pickle data which will **execute arbitrary code during unpickling**. Never unpickle data that could have come from an untrusted source, or that could have been tampered with.

Consider signing data with **`hmac`** if you need to ensure that it has not been tampered with.

Safer serialization formats such as **`json`** may be more appropriate if you are processing untrusted data. See

Comparison with json.

Relationship to other Python modules

Comparison with `marshal`

Python has a more primitive serialization module called **`marshal`**, but in general **`pickle`** should always be the preferred way to serialize Python objects. **`marshal`** exists primarily to support Python's `.pyc` files.

The **`pickle`** module differs from **`marshal`** in several significant ways:

- The **`pickle`** module keeps track of the objects it has already serialized, so that later references to the same object won't be serialized again. **`marshal`** doesn't do this.

  This has implications both for recursive objects and object sharing. Recursive objects are objects that contain references to themselves. These are not handled by marshal, and in fact, attempting to marshal recursive objects will crash your Python interpreter. Object sharing happens when there are multiple references to the same object in different places in the object hierarchy being serialized. **`pickle`** stores such objects only once, and ensures that all other references point to the master copy. Shared objects remain shared, which can be very important for mutable objects.

- **`marshal`** cannot be used to serialize user-defined classes and their instances. **`pickle`** can save and restore class instances transparently, however the class definition must be importable and live in the same module as when the object was stored.

- The **`marshal`** serialization format is not guaranteed to be portable across Python versions. Because its primary job in life is to support `.pyc` files, the Python implementers reserve the right to change the serialization format in non-backwards compatible ways should the need arise. The **`pickle`** serialization format is guaranteed to be backwards compatible across Python releases provided a compatible pickle protocol is chosen and pickling and unpickling code deals with Python 2 to Python 3 type differences if your data is crossing that unique breaking change language boundary.

Comparison with `json`

There are fundamental differences between the pickle protocols and JSON (JavaScript Object Notation) [https://json.org]:

- JSON is a text serialization format (it outputs unicode text, although most of the time it is then encoded to `utf-8`), while pickle is a binary serialization format;
- JSON is human-readable, while pickle is not;
- JSON is interoperable and widely used outside of the Python ecosystem, while pickle is Python-specific;
- JSON, by default, can only represent a subset of the Python built-in types, and no custom classes; pickle can represent an extremely large number of Python types (many of them automatically, by clever usage of Python’s introspection facilities; complex cases can be tackled by implementing specific object APIs);
- Unlike pickle, deserializing untrusted JSON does not in itself create an arbitrary code execution vulnerability.

See also

The `json` module: a standard library module allowing JSON serialization and deserialization.

Data stream format

The data format used by `pickle` is Python-specific. This has the advantage that there are no restrictions imposed by external standards such as JSON (which can’t represent pointer sharing); however it means that non-Python programs may not be able to reconstruct pickled Python objects.

By default, the `pickle` data format uses a relatively compact binary representation. If you need optimal size characteristics, you can efficiently compress pickled data.

The module `pickletools` contains tools for analyzing data streams generated by `pickle`. `pickletools` source code has extensive comments about opcodes used by pickle protocols.

There are currently 6 different protocols which can be used for pickling. The higher the protocol used, the more recent the version of Python needed to read the pickle produced.

- Protocol version 0 is the original “human-readable” protocol and is backwards compatible with earlier versions of Python.
- Protocol version 1 is an old binary format which is also compatible with earlier versions of Python.
- Protocol version 2 was introduced in Python 2.3. It provides much more efficient pickling of new-style classes. Refer to **PEP 307** [https://peps.python.org/pep-0307/] for information about improvements brought by protocol 2.

- Protocol version 3 was added in Python 3.0. It has explicit support for `bytes` objects and cannot be unpickled by Python 2.x. This was the default protocol in Python 3.0–3.7.
- Protocol version 4 was added in Python 3.4. It adds support for very large objects, pickling more kinds of objects, and some data format optimizations. This was the default protocol in Python 3.8–3.13. Refer to **PEP 3154** [https://peps.python.org/pep-3154/] for information about improvements brought by protocol 4.
- Protocol version 5 was added in Python 3.8. It adds support for out-of-band data and speedup for in-band data. It is the default protocol starting with Python 3.14. Refer to **PEP 574** [https://peps.python.org/pep-0574/] for information about improvements brought by protocol 5.

Note

Serialization is a more primitive notion than persistence; although `pickle` reads and writes file objects, it does not handle the issue of naming persistent objects, nor the (even more complicated) issue of concurrent access to persistent objects. The `pickle` module can transform a complex object into a byte stream and it can transform the byte stream into an object with the same internal structure. Perhaps the most obvious thing to do with these byte streams is to write them onto a file, but it is also conceivable to send them across a network or store them in a database. The `shelve` module provides a simple interface to pickle and unpickle objects on DBM-style database files.

Module Interface

To serialize an object hierarchy, you simply call the `dumps()` function. Similarly, to de-serialize a data stream, you call the `loads()` function. However, if you want more control over serialization and de-serialization, you can create a `Pickler` or an `Unpickler` object, respectively.

The `pickle` module provides the following constants:

pickle.HIGHEST_PROTOCOL

An integer, the highest protocol version available. This value can be passed as a *protocol* value to functions `dump()` and `dumps()` as well as the `Pickler` constructor.

pickle.DEFAULT_PROTOCOL

An integer, the default protocol version used for pickling. May be less than `HIGHEST_PROTOCOL`. Currently the default protocol is 5, introduced in Python 3.8 and incompatible with previous versions. This version introduces support for out-of-band buffers, where **PEP 3118** [https://peps.python.org/pep-3118/]-compatible data can be transmitted separately from the main pickle stream.

Changed in version 3.0: The default protocol is 3.

Changed in version 3.8: The default protocol is 4.

Changed in version 3.14: The default protocol is 5.

The `pickle` module provides the following functions to make the pickling process more convenient:

pickle.dump(*obj*, *file*, *protocol=None*, *, *fix_imports=True*, *buffer_callback=None*)

Write the pickled representation of the object *obj* to the open file object *file*. This is equivalent to `Pickler(file, protocol).dump(obj)`.

Arguments *file*, *protocol*, *fix_imports* and *buffer_callback* have the same meaning as in the **`Pickler`** constructor.

Changed in version 3.8: The *buffer_callback* argument was added.

pickle.dumps(*obj*, *protocol=None*, ***, *fix_imports=True*, *buffer_callback=None*)

Return the pickled representation of the object *obj* as a **`bytes`** object, instead of writing it to a file.

Arguments *protocol*, *fix_imports* and *buffer_callback* have the same meaning as in the **`Pickler`** constructor.

Changed in version 3.8: The *buffer_callback* argument was added.

pickle.load(*file*, ***, *fix_imports=True*, *encoding='ASCII'*, *errors='strict'*, *buffers=None*)

Read the pickled representation of an object from the open file object *file* and return the reconstituted object hierarchy specified therein. This is equivalent to `Unpickler(file).load()`.

The protocol version of the pickle is detected automatically, so no protocol argument is needed. Bytes past the pickled representation of the object are ignored.

Arguments *file*, *fix_imports*, *encoding*, *errors*, *strict* and *buffers* have the same meaning as in the **`Unpickler`** constructor.

Changed in version 3.8: The *buffers* argument was added.

pickle.loads(*data*, /, *, *fix_imports=True*, *encoding='ASCII'*, *errors='strict'*, *buffers=None*)

Return the reconstituted object hierarchy of the pickled representation *data* of an object. *data* must be a bytes-like object.

The protocol version of the pickle is detected automatically, so no protocol argument is needed. Bytes past the pickled representation of the object are ignored.

Arguments *fix_imports*, *encoding*, *errors*, *strict* and *buffers* have the same meaning as in the **`Unpickler`** constructor.

Changed in version 3.8: The *buffers* argument was added.

The **`pickle`** module defines three exceptions:

exception pickle.PickleError

Common base class for the other pickling exceptions. It inherits from **`Exception`**.

exception pickle.PicklingError

Error raised when an unpicklable object is encountered by **`Pickler`**. It inherits from **`PickleError`**.

Refer to What can be pickled and unpickled? to learn what kinds of objects can be pickled.

exception pickle.UnpicklingError

Error raised when there is a problem unpickling an object, such as a data corruption or a security violation. It inherits from **`PickleError`**.

Note that other exceptions may also be raised during unpickling, including (but not necessarily limited to) AttributeError, EOFError, ImportError, and IndexError.

The `pickle` module exports three classes, `Pickler`, `Unpickler` and `PickleBuffer`:

class pickle.Pickler(*file*, *protocol=None*, ***, *fix_imports=True*, *buffer_callback=None*)

This takes a binary file for writing a pickle data stream.

The optional *protocol* argument, an integer, tells the pickler to use the given protocol; supported protocols are 0 to `HIGHEST_PROTOCOL`. If not specified, the default is `DEFAULT_PROTOCOL`. If a negative number is specified, `HIGHEST_PROTOCOL` is selected.

The *file* argument must have a write() method that accepts a single bytes argument. It can thus be an on-disk file opened for binary writing, an `io.BytesIO` instance, or any other custom object that meets this interface.

If *fix_imports* is true and *protocol* is less than 3, pickle will try to map the new Python 3 names to the old module names used in Python 2, so that the pickle data stream is readable with Python 2.

If *buffer_callback* is `None` (the default), buffer views are serialized into *file* as part of the pickle stream.

If *buffer_callback* is not `None`, then it can be called any number of times with a buffer view. If the callback returns a false value (such as `None`), the given buffer is out-of-band; otherwise the buffer is serialized in-band, i.e. inside the pickle stream.

It is an error if *buffer_callback* is not `None` and *protocol* is `None` or smaller than 5.

Changed in version 3.8: The *buffer_callback* argument was added.

dump(*obj*)

Write the pickled representation of *obj* to the open file object given in the constructor.

persistent_id(*obj*)

Do nothing by default. This exists so a subclass can override it.

If **`persistent_id()`** returns `None`, *obj* is pickled as usual. Any other value causes **`Pickler`** to emit the returned value as a persistent ID for *obj*. The meaning of this persistent ID should be defined by **`Unpickler.persistent_load()`**. Note that the value returned by **`persistent_id()`** cannot itself have a persistent ID.

See Persistence of External Objects for details and examples of uses.

Changed in version 3.13: Add the default implementation of this method in the C implementation of **`Pickler`**.

dispatch_table

A pickler object's dispatch table is a registry of *reduction functions* of the kind which can be declared using **`copyreg.pickle()`**. It is a mapping whose keys are classes and whose values are reduction functions. A reduction function takes a single argument of the associated class and should conform to the same interface as a `__reduce__()` method.

By default, a pickler object will not have a `dispatch_table` attribute, and it will instead use the global dispatch table managed by the `copyreg` module. However, to customize the pickling for a specific pickler object one can set the `dispatch_table` attribute to a dict-like object. Alternatively, if a subclass of `Pickler` has a `dispatch_table` attribute then this will be used as the default dispatch table for instances of that class.

See Dispatch Tables for usage examples.

Added in version 3.3.

reducer_override(*obj*)

Special reducer that can be defined in `Pickler` subclasses. This method has priority over any reducer in the `dispatch_table`. It should conform to the same interface as a `__reduce__()` method, and can optionally return `NotImplemented` to fallback on `dispatch_table`-registered reducers to pickle `obj`.

For a detailed example, see Custom Reduction for Types, Functions, and Other Objects.

Added in version 3.8.

fast

Deprecated. Enable fast mode if set to a true value. The fast mode disables the usage of memo, therefore speeding the pickling process by not generating superfluous PUT opcodes. It should not be used with self-referential objects, doing otherwise will cause `Pickler` to recurse infinitely.

Use `pickletools.optimize()` if you need more compact pickles.

clear_memo()

Clears the pickler’s “memo”.

The memo is the data structure that remembers which objects the pickler has already seen, so that shared or recursive objects are pickled by reference and not by value. This method is useful when re-using picklers.

class pickle.Unpickler(*file*, *, *fix_imports=True*, *encoding='ASCII'*, *errors='strict'*, *buffers=None*)

This takes a binary file for reading a pickle data stream.

The protocol version of the pickle is detected automatically, so no protocol argument is needed.

The argument *file* must have three methods, a read() method that takes an integer argument, a readinto() method that takes a buffer argument and a readline() method that requires no arguments, as in the `io.BufferedIOBase` interface. Thus *file* can be an on-disk file opened for binary reading, an `io.BytesIO` object, or any other custom object that meets this interface.

The optional arguments *fix_imports*, *encoding* and *errors* are used to control compatibility support for pickle stream generated by Python 2. If *fix_imports* is true, pickle will try to map the old Python 2 names to the new names used in Python 3. The *encoding* and *errors* tell pickle how to decode 8-bit string instances pickled by Python 2; these default to ‘ASCII’ and ‘strict’, respectively. The *encoding* can be ‘bytes’ to read these 8-bit string instances as bytes objects. Using `encoding='latin1'` is required for unpickling NumPy arrays and instances of `datetime`, `date` and `time` pickled by Python 2.

If *buffers* is `None` (the default), then all data necessary for deserialization must be contained in the pickle stream. This means that the *buffer_callback* argument was `None` when a `Pickler` was instantiated (or when `dump()` or `dumps()` was called).

If *buffers* is not `None`, it should be an iterable of buffer-enabled objects that is consumed each time the pickle stream references an out-of-band buffer view. Such buffers have been given in order to the *buffer_callback* of a Pickler object.

Changed in version 3.8: The *buffers* argument was added.

load()

Read the pickled representation of an object from the open file object given in the constructor, and return the reconstituted object hierarchy specified therein. Bytes past the pickled representation of the object are ignored.

persistent_load(*pid*)

Raise an `UnpicklingError` by default.

If defined, `persistent_load()` should return the object specified by the persistent ID *pid*. If an invalid persistent ID is encountered, an `UnpicklingError` should be raised.

See Persistence of External Objects for details and examples of uses.

Changed in version 3.13: Add the default implementation of this method in the C implementation of **`Unpickler`**.

find_class(*module*, *name*)

Import *module* if necessary and return the object called *name* from it, where the *module* and *name* arguments are **`str`** objects. Note, unlike its name suggests, **`find_class()`** is also used for finding functions.

Subclasses may override this to gain control over what type of objects and how they can be loaded, potentially reducing security risks. Refer to Restricting Globals for details.

Raises an auditing event `pickle.find_class` with arguments `module`, `name`.

class pickle.PickleBuffer(*buffer*)

A wrapper for a buffer representing picklable data. *buffer* must be a buffer-providing object, such as a bytes-like object or a N-dimensional array.

`PickleBuffer` is itself a buffer provider, therefore it is possible to pass it to other APIs expecting a buffer-providing object, such as **`memoryview`**.

`PickleBuffer` objects can only be serialized using pickle protocol 5 or higher. They are eligible for out-of-band serialization.

Added in version 3.8.

raw()

Return a `memoryview` of the memory area underlying this buffer. The returned object is a one-dimensional, C-contiguous memoryview with format `B` (unsigned bytes). `BufferError` is raised if the buffer is neither C- nor Fortran-contiguous.

release()

Release the underlying buffer exposed by the PickleBuffer object.

What can be pickled and unpickled?

The following types can be pickled:

- built-in constants (`None`, `True`, `False`, `Ellipsis`, and `NotImplemented`);
- integers, floating-point numbers, complex numbers;
- strings, bytes, bytearrays;
- tuples, lists, sets, and dictionaries containing only picklable objects;
- functions (built-in and user-defined) accessible from the top level of a module (using `def`, not `lambda`);
- classes accessible from the top level of a module;
- instances of such classes whose the result of calling `__getstate__()` is picklable (see section Pickling Class Instances for details).

Attempts to pickle unpicklable objects will raise the `PicklingError` exception; when this happens, an unspecified number of bytes may have already been written to the underlying file. Trying to pickle a highly recursive data structure may exceed the maximum recursion depth, a `RecursionError` will be raised in this case. You can carefully raise this limit with `sys.setrecursionlimit()`.

Note that functions (built-in and user-defined) are pickled by fully qualified name, not by value. [[2]] This means that only the function name is pickled, along with the name of the containing module and classes. Neither the function's code, nor any of its function attributes are pickled. Thus the defining module must be importable in the unpickling environment, and the module must contain the named object, otherwise an exception will be raised. [[3]]

Similarly, classes are pickled by fully qualified name, so the same restrictions in the unpickling environment apply. Note that none of the class's code or data is pickled, so in the following example the class attribute `attr` is not restored in the unpickling environment:

```
class Foo:
    attr = 'A class attribute'

picklestring = pickle.dumps(Foo)
```

These restrictions are why picklable functions and classes must be defined at the top level of a module.

Similarly, when class instances are pickled, their class's code and data are not pickled along with them. Only the instance data are pickled. This is done on purpose, so you can fix bugs in a class or add methods to the class and still load objects that were created with an earlier version of the class. If you plan to have long-lived objects that will see many versions of a class, it may be worthwhile to put a version number in the objects so that suitable conversions can be made by the class's `__setstate__()` method.

Pickling Class Instances

In this section, we describe the general mechanisms available to you to define, customize, and control how class instances are pickled and unpickled.

In most cases, no additional code is needed to make instances picklable. By default, pickle will retrieve the class and the attributes of an instance via introspection. When a class instance is unpickled, its `__init__()` method is usually *not* invoked. The default behaviour first creates an uninitialized instance and then restores the saved attributes. The following code shows an implementation of this behaviour:

```
def save(obj):
    return (obj.__class__, obj.__dict__)

def restore(cls, attributes):
    obj = cls.__new__(cls)
    obj.__dict__.update(attributes)
    return obj
```

Classes can alter the default behaviour by providing one or several special methods:

object.__getnewargs_ex__()

In protocols 2 and newer, classes that implements the `__getnewargs_ex__()` method can dictate the values passed to the `__new__()` method upon unpickling. The method must return a pair `(args, kwargs)` where *args* is a tuple of positional arguments and *kwargs* a dictionary of named arguments for constructing the object. Those will be passed to the `__new__()` method upon unpickling.

You should implement this method if the `__new__()` method of your class requires keyword-only arguments. Otherwise, it is recommended for compatibility to implement `__getnewargs__()`.

Changed in version 3.6: `__getnewargs_ex__()` is now used in protocols 2 and 3.

object.__getnewargs__()

This method serves a similar purpose as `__getnewargs_ex__()`, but supports only positional arguments. It must return a tuple of arguments `args` which will be passed to the `__new__()` method upon unpickling.

`__getnewargs__()` will not be called if `__getnewargs_ex__()` is defined.

Changed in version 3.6: Before Python 3.6, `__getnewargs__()` was called instead of `__getnewargs_ex__()` in protocols 2 and 3.

object.__getstate__()

Classes can further influence how their instances are pickled by overriding the method `__getstate__()`. It is called and the returned object is pickled as the contents for the instance, instead of a default state. There are several cases:

- For a class that has no instance `__dict__` and no `__slots__`, the default state is `None`.
- For a class that has an instance `__dict__` and no `__slots__`, the default state is `self.__dict__`.
- For a class that has an instance `__dict__` and `__slots__`, the default state is a tuple consisting of two dictionaries: `self.__dict__`, and a dictionary mapping slot names to slot values. Only slots that have a value are included in the latter.
- For a class that has `__slots__` and no instance `__dict__`, the default state is a tuple whose first item is `None` and whose second item is a dictionary

mapping slot names to slot values described in the previous bullet.

Changed in version 3.11: Added the default implementation of the `__getstate__()` method in the `object` class.

object.__setstate__(*state*)

Upon unpickling, if the class defines `__setstate__()`, it is called with the unpickled state. In that case, there is no requirement for the state object to be a dictionary. Otherwise, the pickled state must be a dictionary and its items are assigned to the new instance’s dictionary.

Note

If `__reduce__()` returns a state with value `None` at pickling, the `__setstate__()` method will not be called upon unpickling.

Refer to the section Handling Stateful Objects for more information about how to use the methods `__getstate__()` and `__setstate__()`.

Note

At unpickling time, some methods like `__getattr__()`, `__getattribute__()`, or `__setattr__()` may be called upon the instance. In case those methods rely on some internal invariant being true, the type should implement `__new__()` to establish such an invariant, as `__init__()` is not called when unpickling an instance.

As we shall see, pickle does not use directly the methods described above. In fact, these methods are part of the copy protocol which implements the `__reduce__()` special method. The copy protocol provides a unified interface for retrieving the data necessary for pickling and copying objects. [[4]]

Although powerful, implementing `__reduce__()` directly in your classes is error prone. For this reason, class designers should use the high-level interface (i.e., `__getnewargs_ex__()`, `__getstate__()` and `__setstate__()`) whenever possible. We will show, however, cases where using **`__reduce__()`** is the only option or leads to more efficient pickling or both.

object.__reduce__()

The interface is currently defined as follows. The `__reduce__()` method takes no argument and shall return either a string or preferably a tuple (the returned object is often referred to as the “reduce value”).

If a string is returned, the string should be interpreted as the name of a global variable. It should be the object’s local name relative to its module; the pickle module searches the module namespace to determine the object’s module. This behaviour is typically useful for singletons.

When a tuple is returned, it must be between two and six items long. Optional items can either be omitted, or `None` can be provided as their value. The semantics of each item are in order:

- A callable object that will be called to create the initial version of the object.
- A tuple of arguments for the callable object. An empty tuple must be given if the callable does not accept any argument.

- Optionally, the object’s state, which will be passed to the object’s `__setstate__()` method as previously described. If the object has no such method then, the value must be a dictionary and it will be added to the object’s `__dict__` attribute.

- Optionally, an iterator (and not a sequence) yielding successive items. These items will be appended to the object either using `obj.append(item)` or, in batch, using `obj.extend(list_of_items)`. This is primarily used for list subclasses, but may be used by other classes as long as they have `append()` and `extend()` methods with the appropriate signature. (Whether **`append()`** or **`extend()`** is used depends on which pickle protocol version is used as well as the number of items to append, so both must be supported.)

- Optionally, an iterator (not a sequence) yielding successive key-value pairs. These items will be stored to the object using `obj[key] = value`. This is primarily used for dictionary subclasses, but may be used by other classes as long as they implement `__setitem__()`.

- Optionally, a callable with a `(obj, state)` signature. This callable allows the user to programmatically control the state-updating behavior of a specific object, instead of using `obj`’s static `__setstate__()` method. If not `None`, this callable will have priority over `obj`’s `__setstate__()`.

  Added in version 3.8: The optional sixth tuple item, `(obj, state)`, was added.

object.__reduce_ex__(*protocol*)

Alternatively, a `__reduce_ex__()` method may be defined. The only difference is this method should take a single integer argument, the protocol version. When defined, pickle will prefer it over the `__reduce__()` method. In addition, `__reduce__()` automatically becomes a synonym for the extended version. The main use for this method is to provide backwards-compatible reduce values for older Python releases.

Persistence of External Objects

For the benefit of object persistence, the `pickle` module supports the notion of a reference to an object outside the pickled data stream. Such objects are referenced by a persistent ID, which should be either a string of alphanumeric characters (for protocol 0) [[5]] or just an arbitrary object (for any newer protocol).

The resolution of such persistent IDs is not defined by the `pickle` module; it will delegate this resolution to the user-defined methods on the pickler and unpickler, `persistent_id()` and `persistent_load()` respectively.

To pickle objects that have an external persistent ID, the pickler must have a custom `persistent_id()` method that takes an object as an argument and returns either `None` or the persistent ID for that object. When `None` is returned, the pickler simply pickles the object as normal. When a persistent ID string is returned, the pickler will pickle that object, along with a marker so that the unpickler will recognize it as a persistent ID.

To unpickle external objects, the unpickler must have a custom `persistent_load()` method that takes a persistent ID object and returns the referenced object.

Here is a comprehensive example presenting how persistent ID can be used to pickle external objects by reference.

```
# Simple example presenting how persistent ID can be used to
pickle
# external objects by reference.

import pickle
import sqlite3
from collections import namedtuple

# Simple class representing a record in our database.
MemoRecord = namedtuple("MemoRecord", "key, task")

class DBPickler(pickle.Pickler):

    def persistent_id(self, obj):
        # Instead of pickling MemoRecord as a regular class
instance, we emit a
        # persistent ID.
        if isinstance(obj, MemoRecord):
            # Here, our persistent ID is simply a tuple,
containing a tag and a
            # key, which refers to a specific record in the
database.
            return ("MemoRecord", obj.key)
        else:
            # If obj does not have a persistent ID, return
None. This means obj
            # needs to be pickled as usual.
            return None


class DBUnpickler(pickle.Unpickler):

    def __init__(self, file, connection):
        super().__init__(file)
        self.connection = connection

    def persistent_load(self, pid):
        # This method is invoked whenever a persistent ID is
encountered.
```

```
        # Here, pid is the tuple returned by DBPickler.
        cursor = self.connection.cursor()
        type_tag, key_id = pid
        if type_tag == "MemoRecord":
            # Fetch the referenced record from the database and
return it.
            cursor.execute("SELECT * FROM memos WHERE key=?",
(str(key_id),))
            key, task = cursor.fetchone()
            return MemoRecord(key, task)
        else:
            # Always raises an error if you cannot return the
correct object.
            # Otherwise, the unpickler will think None is the
object referenced
            # by the persistent ID.
            raise pickle.UnpicklingError("unsupported
persistent object")


def main():
    import io
    import pprint

    # Initialize and populate our database.
    conn = sqlite3.connect(":memory:")
    cursor = conn.cursor()
    cursor.execute("CREATE TABLE memos(key INTEGER PRIMARY KEY,
task TEXT)")
    tasks = (
        'give food to fish',
        'prepare group meeting',
        'fight with a zebra',
        )
    for task in tasks:
        cursor.execute("INSERT INTO memos VALUES(NULL, ?)",
(task,))

    # Fetch the records to be pickled.
    cursor.execute("SELECT * FROM memos")
    memos = [MemoRecord(key, task) for key, task in cursor]
    # Save the records using our custom DBPickler.
    file = io.BytesIO()
    DBPickler(file).dump(memos)

    print("Pickled records:")
    pprint.pprint(memos)

    # Update a record, just for good measure.
    cursor.execute("UPDATE memos SET task='learn italian' WHERE
key=1")

    # Load the records from the pickle data stream.
    file.seek(0)
    memos = DBUnpickler(file, conn).load()

    print("Unpickled records:")
    pprint.pprint(memos)


if __name__ == '__main__':
    main()
```

Dispatch Tables

If one wants to customize pickling of some classes without disturbing any other code which depends on pickling, then one can create a pickler with a private dispatch table.

The global dispatch table managed by the `copyreg` module is available as **`copyreg.dispatch_table`**. Therefore, one may choose to use a modified copy of **`copyreg.dispatch_table`** as a private dispatch table.

For example

```
f = io.BytesIO()
p = pickle.Pickler(f)
p.dispatch_table = copyreg.dispatch_table.copy()
p.dispatch_table[SomeClass] = reduce_SomeClass
```

creates an instance of `pickle.Pickler` with a private dispatch table which handles the `SomeClass` class specially.

Alternatively, the code

```
class MyPickler(pickle.Pickler):
    dispatch_table = copyreg.dispatch_table.copy()
    dispatch_table[SomeClass] = reduce_SomeClass
f = io.BytesIO()
p = MyPickler(f)
```

does the same but all instances of `MyPickler` will by default share the private dispatch table. On the other hand, the code

```
copyreg.pickle(SomeClass, reduce_SomeClass)
f = io.BytesIO()
p = pickle.Pickler(f)
```

modifies the global dispatch table shared by all users of the **`copyreg`** module.

Handling Stateful Objects

Here’s an example that shows how to modify pickling behavior for a class. The **`TextReader`** class below opens a text file, and returns the line number and line contents each time its **`readline()`** method is called. If a **`TextReader`** instance is pickled, all attributes *except* the file object member are saved. When the instance is unpickled, the file is reopened, and reading resumes from the last location. The **`__setstate__()`** and **`__getstate__()`** methods are used to implement this behavior.

```
class TextReader:
    """Print and number lines in a text file."""

    def __init__(self, filename):
        self.filename = filename
        self.file = open(filename)
        self.lineno = 0
    def readline(self):
        self.lineno += 1
        line = self.file.readline()
        if not line:
            return None
        if line.endswith('\n'):
            line = line[:-1]
        return "%i: %s" % (self.lineno, line)

    def __getstate__(self):
        # Copy the object's state from self.__dict__ which
contains
        # all our instance attributes. Always use the
dict.copy()
        # method to avoid modifying the original state.
        state = self.__dict__.copy()
        # Remove the unpicklable entries.
        del state['file']
        return state

    def __setstate__(self, state):
        # Restore instance attributes (i.e., filename and
lineno).
        self.__dict__.update(state)
        # Restore the previously opened file's state. To do so,
we need to
        # reopen it and read from it until the line count is
restored.
        file = open(self.filename)
        for _ in range(self.lineno):
            file.readline()
        # Finally, save the file.
        self.file = file
```

A sample usage might be something like this:

```
>>> reader = TextReader("hello.txt")
>>> reader.readline()
'1: Hello world!'
>>> reader.readline()
'2: I am line number two.'
>>> new_reader = pickle.loads(pickle.dumps(reader))
>>> new_reader.readline()
'3: Goodbye!'
```

Custom Reduction for Types, Functions, and Other Objects

Added in version 3.8.

Sometimes, `dispatch_table` may not be flexible enough. In particular we may want to customize pickling based on another criterion than the object’s type, or we may want to customize the pickling of functions and classes.

For those cases, it is possible to subclass from the `Pickler` class and implement a `reducer_override()` method. This method can return an arbitrary reduction tuple (see `__reduce__()`). It can alternatively return `NotImplemented` to fallback to the traditional behavior.

If both the `dispatch_table` and `reducer_override()` are defined, then `reducer_override()` method takes priority.

Note

For performance reasons, `reducer_override()` may not be called for the following objects: `None`, `True`, `False`, and exact instances of `int`, `float`, `bytes`, `str`, `dict`, `set`, `frozenset`, `list` and `tuple`.

Here is a simple example where we allow pickling and reconstructing a given class:

```
import io
import pickle
```

```
class MyClass:
    my_attribute = 1

class MyPickler(pickle.Pickler):
    def reducer_override(self, obj):
        """Custom reducer for MyClass."""
        if getattr(obj, "__name__", None) == "MyClass":
            return type, (obj.__name__, obj.__bases__,
                          {'my_attribute': obj.my_attribute})
        else:
            # For any other object, fallback to usual reduction
            return NotImplemented

f = io.BytesIO()
p = MyPickler(f)
p.dump(MyClass)

del MyClass

unpickled_class = pickle.loads(f.getvalue())

assert isinstance(unpickled_class, type)
assert unpickled_class.__name__ == "MyClass"
assert unpickled_class.my_attribute == 1
```

Out-of-band Buffers

Added in version 3.8.

In some contexts, the `pickle` module is used to transfer massive amounts of data. Therefore, it can be important to minimize the number of memory copies, to preserve performance and resource consumption. However, normal operation of the `pickle` module, as it transforms a graph-like structure of objects into a sequential stream of bytes, intrinsically involves copying data to and from the pickle stream.

This constraint can be eschewed if both the *provider* (the implementation of the object types to be transferred) and the *consumer* (the implementation of the communications system) support the out-of-band transfer facilities provided by pickle protocol 5 and higher.

Provider API

The large data objects to be pickled must implement a `__reduce_ex__()` method specialized for protocol 5 and higher, which returns a `PickleBuffer` instance (instead of e.g. a `bytes` object) for any large data.

A `PickleBuffer` object *signals* that the underlying buffer is eligible for out-of-band data transfer. Those objects remain compatible with normal usage of the `pickle` module. However, consumers can also opt-in to tell `pickle` that they will handle those buffers by themselves.

Consumer API

A communications system can enable custom handling of the `PickleBuffer` objects generated when serializing an object graph.

On the sending side, it needs to pass a *buffer_callback* argument to `Pickler` (or to the `dump()` or `dumps()` function), which will be called with each `PickleBuffer` generated while pickling the object graph. Buffers accumulated by the *buffer_callback* will not see their data copied into the pickle stream, only a cheap marker will be inserted.

On the receiving side, it needs to pass a *buffers* argument to `Unpickler` (or to the `load()` or `loads()` function), which is an iterable of the buffers which were passed to *buffer_callback*. That iterable should produce buffers in the same order as they were passed to *buffer_callback*. Those buffers will provide the data expected by the reconstructors of the objects whose pickling produced the original `PickleBuffer` objects.

Between the sending side and the receiving side, the communications system is free to implement its own transfer mechanism for out-of-band buffers. Potential optimizations include the use of shared memory or datatype-dependent compression.

Example

Here is a trivial example where we implement a `bytearray` subclass able to participate in out-of-band buffer pickling:

```
class ZeroCopyByteArray(bytearray):

    def __reduce_ex__(self, protocol):
        if protocol >= 5:
            return type(self)._reconstruct,
(PickleBuffer(self),), None
        else:
            # PickleBuffer is forbidden with pickle protocols
<= 4.
            return type(self)._reconstruct, (bytearray(self),)

    @classmethod
    def _reconstruct(cls, obj):
        with memoryview(obj) as m:
            # Get a handle over the original buffer object
            obj = m.obj
            if type(obj) is cls:
                # Original buffer object is a
ZeroCopyByteArray, return it
                # as-is.
                return obj
            else:
                return cls(obj)
```

The reconstructor (the `_reconstruct` class method) returns the buffer’s providing object if it has the right type. This is an easy way to simulate zero-copy behaviour on this toy example.

On the consumer side, we can pickle those objects the usual way, which when unserialized will give us a copy of the original object:

```
b = ZeroCopyByteArray(b"abc")
data = pickle.dumps(b, protocol=5)
new_b = pickle.loads(data)
print(b == new_b)  # True
print(b is new_b)  # False: a copy was made
```

But if we pass a *buffer_callback* and then give back the accumulated buffers when unserializing, we are able to get back the original object:

```
b = ZeroCopyByteArray(b"abc")
buffers = []
data = pickle.dumps(b, protocol=5,
buffer_callback=buffers.append)
new_b = pickle.loads(data, buffers=buffers)
print(b == new_b)  # True
print(b is new_b)  # True: no copy was made
```

This example is limited by the fact that `bytearray` allocates its own memory: you cannot create a `bytearray` instance that is backed by another object’s memory. However, third-party datatypes such as NumPy arrays do not have this limitation, and allow use of zero-copy pickling (or making as few copies as possible) when transferring between distinct processes or systems.

See also

PEP 574 [https://peps.python.org/pep-0574/] – Pickle protocol 5 with out-of-band data

Restricting Globals

By default, unpickling will import any class or function that it finds in the pickle data. For many applications, this behaviour is unacceptable as it permits the unpickler to import and invoke arbitrary code. Just consider what this hand-crafted pickle data stream does when loaded:

```
>>> import pickle
>>> pickle.loads(b"cos\nsystem\n(S'echo hello world'\ntR.")
hello world
0
```

In this example, the unpickler imports the `os.system()` function and then apply the string argument “echo hello world”. Although this example is inoffensive, it is not difficult to imagine one that could damage your system.

For this reason, you may want to control what gets unpickled by customizing `Unpickler.find_class()`. Unlike its name suggests, `Unpickler.find_class()` is called whenever a global (i.e., a class or a function) is requested. Thus it is possible to either completely forbid globals or restrict them to a safe subset.

Here is an example of an unpickler allowing only few safe classes from the `builtins` module to be loaded:

```
import builtins
import io
import pickle

safe_builtins = {
    'range',
    'complex',
    'set',
    'frozenset',
    'slice',
}

class RestrictedUnpickler(pickle.Unpickler):

    def find_class(self, module, name):
        # Only allow safe classes from builtins.
        if module == "builtins" and name in safe_builtins:
            return getattr(builtins, name)
        # Forbid everything else.
        raise pickle.UnpicklingError("global '%s.%s' is 
forbidden" %
                                     (module, name))

def restricted_loads(s):
    """Helper function analogous to pickle.loads()."""
    return RestrictedUnpickler(io.BytesIO(s)).load()
```

A sample usage of our unpickler working as intended:

```
>>> restricted_loads(pickle.dumps([1, 2, range(15)]))
[1, 2, range(0, 15)]
>>> restricted_loads(b"cos\nsystem\n(S'echo hello world'\ntR.")
Traceback (most recent call last):
  ...
pickle.UnpicklingError: global 'os.system' is forbidden
>>> restricted_loads(b'cbuiltins\neval\n'
...                  b'(S\'getattr(__import__("os"), "system")'
...                  b'("echo hello world")\'\ntR.')
Traceback (most recent call last):
  ...
pickle.UnpicklingError: global 'builtins.eval' is forbidden
```

As our examples shows, you have to be careful with what you allow to be unpickled. Therefore if security is a concern, you may want to consider alternatives such as the marshalling API in `xmlrpc.client` or third-party solutions.

Performance

Recent versions of the pickle protocol (from protocol 2 and upwards) feature efficient binary encodings for several common features and built-in types. Also, the `pickle` module has a transparent optimizer written in C.

Examples

For the simplest code, use the `dump()` and `load()` functions.

```
import pickle

# An arbitrary collection of objects supported by pickle.
data = {
    'a': [1, 2.0, 3+4j],
    'b': ("character string", b"byte string"),
    'c': {None, True, False}
}

with open('data.pickle', 'wb') as f:
    # Pickle the 'data' dictionary using the highest protocol
available.
    pickle.dump(data, f, pickle.HIGHEST_PROTOCOL)
```

The following example reads the resulting pickled data.

```
import pickle

with open('data.pickle', 'rb') as f:
    # The protocol version used is detected automatically, so
we do not
    # have to specify it.
    data = pickle.load(f)
```

Command-line interface

The `pickle` module can be invoked as a script from the command line, it will display contents of the pickle files. However, when the pickle file that you want to examine comes from an untrusted source, `-m pickletools` is a safer option because it does not execute pickle bytecode, see pickletools CLI usage.

```
python -m pickle pickle_file [pickle_file ...]
```

The following option is accepted:

pickle_file

A pickle file to read, or `-` to indicate reading from standard input.

See also

Module `copyreg`

Pickle interface constructor registration for extension types.

Module `pickletools`

Tools for working with and analyzing pickled data.

Module `shelve`

Indexed databases of objects; uses `pickle`.

Module `copy`

Shallow and deep object copying.

Module `marshal`

High-performance serialization of built-in types.

Footnotes

[1]

Don’t confuse this with the `marshal` module
[2]

This is why `lambda` functions cannot be pickled: all **`lambda`** functions share the same name: `<lambda>`.

[3]

The exception raised will likely be an `ImportError` or an `AttributeError` but it could be something else.

[4]

The `copy` module uses this protocol for shallow and deep copying operations.

[5]

The limitation on alphanumeric characters is due to the fact that persistent IDs in protocol 0 are delimited by the newline character. Therefore if any kind of newline characters occurs in persistent IDs, the resulting pickled data will become unreadable.

`copyreg` — Register `pickle` support functions

Source code: Lib/copyreg.py
[https://github.com/python/cpython/tree/3.14/Lib/copyreg.py]

The **`copyreg`** module offers a way to define functions used while pickling specific objects. The **`pickle`** and **`copy`** modules use those functions when pickling/copying those objects. The module provides configuration information about object constructors which are not classes. Such constructors may be factory functions or class instances.

copyreg.constructor(*object*)

Declares *object* to be a valid constructor. If *object* is not callable (and hence not valid as a constructor), raises **`TypeError`**.

copyreg.pickle(*type*, *function*, *constructor_ob=None*)

Declares that *function* should be used as a “reduction” function for objects of type *type*. *function* must return either a string or a tuple containing between two and six elements. See the **`dispatch_table`** for more details on the interface of *function*.

The *constructor_ob* parameter is a legacy feature and is now ignored, but if passed it must be a callable.

Note that the **`dispatch_table`** attribute of a pickler object or subclass of **`pickle.Pickler`** can also be used for declaring reduction functions.

Example

The example below would like to show how to register a pickle function and how it will be used:

```
>>> import copyreg, copy, pickle
>>> class C:
...     def __init__(self, a):
...         self.a = a
...
>>> def pickle_c(c):
...     print("pickling a C instance...")
...     return C, (c.a,)
...
>>> copyreg.pickle(C, pickle_c)
>>> c = C(1)
>>> d = copy.copy(c)
pickling a C instance...
>>> p = pickle.dumps(c)
pickling a C instance...
```

`shelve` — Python object persistence

Source code: Lib/shelve.py [https://github.com/python/cpython/tree/3.14/Lib/shelve.py]

A “shelf” is a persistent, dictionary-like object. The difference with “dbm” databases is that the values (not the keys!) in a shelf can be essentially arbitrary Python objects — anything that the **`pickle`** module can handle. This includes most class instances, recursive data types, and objects containing lots of shared sub-objects. The keys are ordinary strings.

shelve.open(*filename*, *flag='c'*, *protocol=None*, *writeback=False*)

> Open a persistent dictionary. The filename specified is the base filename for the underlying database. As a side-effect, an extension may be added to the filename and more than one file may be created. By default, the underlying database file is opened for reading and writing. The optional *flag* parameter has the same interpretation as the *flag* parameter of **`dbm.open()`**.
>
> By default, pickles created with **`pickle.DEFAULT_PROTOCOL`** are used to serialize values. The version of the pickle protocol can be specified with the *protocol* parameter.
>
> Because of Python semantics, a shelf cannot know when a mutable persistent-dictionary entry is modified. By default modified objects are written *only* when assigned to the shelf (see Example). If the optional *writeback*

parameter is set to `True`, all entries accessed are also cached in memory, and written back on `sync()` and `close()`; this can make it handier to mutate mutable entries in the persistent dictionary, but, if many entries are accessed, it can consume vast amounts of memory for the cache, and it can make the close operation very slow since all accessed entries are written back (there is no way to determine which accessed entries are mutable, nor which ones were actually mutated).

Changed in version 3.10: `pickle.DEFAULT_PROTOCOL` is now used as the default pickle protocol.

Changed in version 3.11: Accepts path-like object for filename.

Note

Do not rely on the shelf being closed automatically; always call `close()` explicitly when you don't need it any more, or use `shelve.open()` as a context manager:

```
with shelve.open('spam') as db:
    db['eggs'] = 'eggs'
```

Warning

Because the `shelve` module is backed by `pickle`, it is insecure to load a shelf from an untrusted source. Like with pickle, loading a shelf can execute arbitrary code.

Shelf objects support most of methods and operations supported by dictionaries (except copying, constructors and operators `|` and `|=`). This eases the transition from dictionary based scripts to those requiring persistent storage.

Two additional methods are supported:

Shelf.sync()

Write back all entries in the cache if the shelf was opened with *writeback* set to `True`. Also empty the cache and synchronize the persistent dictionary on disk, if feasible. This is called automatically when the shelf is closed with `close()`.

Shelf.close()

Synchronize and close the persistent *dict* object. Operations on a closed shelf will fail with a `ValueError`.

See also

Persistent dictionary recipe [https://code.activestate.com/recipes/576642-persistent-dict-with-multiple-standard-file-format/] with widely supported storage formats and having the speed of native dictionaries.

Restrictions

- The choice of which database package will be used (such as `dbm.ndbm` or `dbm.gnu`) depends on which interface is available. Therefore it is not safe to open the database directly using `dbm`. The database is also (unfortunately) subject to the limitations of `dbm`, if it is used — this means that (the pickled representation of) the objects stored in the database should be fairly small, and in rare cases key collisions may cause the database to refuse updates.
- The `shelve` module does not support *concurrent* read/write access to shelved objects. (Multiple simultaneous read accesses are safe.) When a program

has a shelf open for writing, no other program should have it open for reading or writing. Unix file locking can be used to solve this, but this differs across Unix versions and requires knowledge about the database implementation used.
- On macOS `dbm.ndbm` can silently corrupt the database file on updates, which can cause hard crashes when trying to read from the database.

class shelve.Shelf(*dict*, *protocol=None*, *writeback=False*, *keyencoding='utf-8'*)

A subclass of `collections.abc.MutableMapping` which stores pickled values in the *dict* object.

By default, pickles created with `pickle.DEFAULT_PROTOCOL` are used to serialize values. The version of the pickle protocol can be specified with the *protocol* parameter. See the `pickle` documentation for a discussion of the pickle protocols.

If the *writeback* parameter is `True`, the object will hold a cache of all entries accessed and write them back to the *dict* at sync and close times. This allows natural operations on mutable entries, but can consume much more memory and make sync and close take a long time.

The *keyencoding* parameter is the encoding used to encode keys before they are used with the underlying dict.

A `Shelf` object can also be used as a context manager, in which case it will be automatically closed when the `with` block ends.

Changed in version 3.2: Added the *keyencoding* parameter; previously, keys were always encoded in UTF-

8.
Changed in version 3.4: Added context manager support.

Changed in version 3.10: `pickle.DEFAULT_PROTOCOL` is now used as the default pickle protocol.

class shelve.BsdDbShelf(*dict*, *protocol=None*, *writeback=False*, *keyencoding='utf-8'*)

A subclass of `Shelf` which exposes **`first()`**, **`next()`**, **`previous()`**, **`last()`** and **`set_location()`** methods. These are available in the third-party **`bsddb`** module from pybsddb [https://www.jcea.es/programacion/pybsddb.htm] but not in other database modules. The *dict* object passed to the constructor must support those methods. This is generally accomplished by calling one of **`bsddb.hashopen()`**, **`bsddb.btopen()`** or **`bsddb.rnopen()`**. The optional *protocol*, *writeback*, and *keyencoding* parameters have the same interpretation as for the `Shelf` class.

class shelve.DbfilenameShelf(*filename*, *flag='c'*, *protocol=None*, *writeback=False*)

A subclass of `Shelf` which accepts a *filename* instead of a dict-like object. The underlying file will be opened using `dbm.open()`. By default, the file will be created and opened for both read and write. The optional *flag* parameter has the same interpretation as for the `open()` function. The optional *protocol* and *writeback* parameters have the same interpretation as for the `Shelf` class.

Example

To summarize the interface (`key` is a string, `data` is an arbitrary object):

```
import shelve

d = shelve.open(filename)  # open -- file may get suffix added
by low-level
                           # library

d[key] = data              # store data at key (overwrites old
data if
                           # using an existing key)
data = d[key]              # retrieve a COPY of data at key
(raise KeyError
                           # if no such key)
del d[key]                 # delete data stored at key (raises
KeyError
                           # if no such key)

flag = key in d            # true if the key exists
klist = list(d.keys())     # a list of all existing keys
(slow!)

# as d was opened WITHOUT writeback=True, beware:
d['xx'] = [0, 1, 2]        # this works as expected, but...
d['xx'].append(3)          # *this doesn't!* -- d['xx'] is
STILL [0, 1, 2]!

# having opened d without writeback=True, you need to code
carefully:
temp = d['xx']             # extracts the copy
temp.append(5)             # mutates the copy
d['xx'] = temp             # stores the copy right back, to
persist it

# or, d=shelve.open(filename,writeback=True) would let you just
code
# d['xx'].append(5) and have it work as expected, BUT it would
also
# consume more memory and make the d.close() operation slower.

d.close()                  # close it
```

See also

Module `dbm`
Generic interface to `dbm`-style databases.

Module `pickle`
Object serialization used by `shelve`.

`marshal` — Internal Python object serialization

This module contains functions that can read and write Python values in a binary format. The format is specific to Python, but independent of machine architecture issues (e.g., you can write a Python value to a file on a PC, transport the file to a Mac, and read it back there). Details of the format are undocumented on purpose; it may change between Python versions (although it rarely does). [[1]]

This is not a general "persistence" module. For general persistence and transfer of Python objects through RPC calls, see the modules `pickle` and `shelve`. The `marshal` module exists mainly to support reading and writing the "pseudo-compiled" code for Python modules of `.pyc` files. Therefore, the Python maintainers reserve the right to modify the marshal format in backward incompatible ways should the need arise. The format of code objects is not compatible between Python versions, even if the version of the format is the same. De-serializing a code object in the incorrect Python version has undefined behavior. If you're serializing and de-serializing Python objects, use the `pickle` module instead – the performance is comparable, version independence is guaranteed, and pickle supports a substantially wider range of objects than marshal.

Warning

The `marshal` module is not intended to be secure against erroneous or maliciously constructed data. Never unmarshal data received from an untrusted or unauthenticated source.

There are functions that read/write files as well as functions operating on bytes-like objects.

Not all Python object types are supported; in general, only objects whose value is independent from a particular invocation of Python can be written and read by this module. The following types are supported:

- Numeric types: `int`, `bool`, `float`, `complex`.
- Strings (`str`) and `bytes`. Bytes-like objects like `bytearray` are marshalled as `bytes`.
- Containers: `tuple`, `list`, `set`, `frozenset`, and (since `version` 5), `slice`. It should be understood that these are supported only if the values contained therein are themselves supported. Recursive containers are supported since `version` 3.
- The singletons `None`, `Ellipsis` and `StopIteration`.
- `code` objects, if *allow_code* is true. See note above about version dependence.

Changed in version 3.4:

- Added format version 3, which supports marshalling recursive lists, sets and dictionaries.
- Added format version 4, which supports efficient representations of short strings.

Changed in version 3.14: Added format version 5, which allows marshalling slices.

The module defines these functions:

marshal.dump(*value*, *file*, *version=version*, /, *, *allow_code=True*)

Write the value on the open file. The value must be a supported type. The file must be a writeable binary file.

If the value has (or contains an object that has) an unsupported type, a **`ValueError`** exception is raised — but garbage data will also be written to the file. The object will not be properly read back by **`load()`**. Code objects are only supported if *allow_code* is true.

The *version* argument indicates the data format that `dump` should use (see below).

Raises an auditing event `marshal.dumps` with arguments `value`, `version`.

Changed in version 3.13: Added the *allow_code* parameter.

marshal.load(*file*, /, *, *allow_code=True*)

Read one value from the open file and return it. If no valid value is read (e.g. because the data has a different Python version's incompatible marshal format), raise **`EOFError`**, **`ValueError`** or **`TypeError`**. Code objects are only supported if *allow_code* is true. The file must be a readable binary file.

Raises an auditing event `marshal.load` with no arguments.

Note

If an object containing an unsupported type was marshalled with **`dump()`**, **`load()`** will substitute `None` for the unmarshallable type.

Changed in version 3.10: This call used to raise a `code.__new__` audit event for each code object. Now it raises a single `marshal.load` event for the entire load operation.

Changed in version 3.13: Added the *allow_code* parameter.

marshal.dumps(*value*, *version=version*, /, *, *allow_code=True*)

Return the bytes object that would be written to a file by `dump(value, file)`. The value must be a supported type. Raise a **`ValueError`** exception if value has (or contains an object that has) an unsupported type. Code objects are only supported if *allow_code* is true.

The *version* argument indicates the data format that `dumps` should use (see below).

Raises an auditing event `marshal.dumps` with arguments `value`, `version`.

Changed in version 3.13: Added the *allow_code* parameter.

marshal.loads(*bytes*, /, *, *allow_code=True*)

Convert the bytes-like object to a value. If no valid value is found, raise **`EOFError`**, **`ValueError`** or **`TypeError`**. Code objects are only supported if *allow_code* is true. Extra bytes in the input are ignored.

Raises an auditing event `marshal.loads` with argument `bytes`.

Changed in version 3.10: This call used to raise a `code.__new__` audit event for each code object. Now it raises a single `marshal.loads` event for the entire load operation.

Changed in version 3.13: Added the *allow_code* parameter.

In addition, the following constants are defined:

marshal.version

Indicates the format that the module uses. Version 0 is the historical first version; subsequent versions add new features. Generally, a new version becomes the default when it is introduced.

Version	**Available since**	**New features**
1	Python 2.4	Sharing interned strings
2	Python 2.5	Binary representation of floats
3	Python 3.4	Support for object instancing and recursion
4	Python 3.4	Efficient representation of short strings
5	Python 3.14	Support for `slice` objects

Footnotes

[1]

The name of this module stems from a bit of terminology used by the designers of Modula-3 (amongst others), who use the term “marshalling” for shipping of data around in a self-contained form. Strictly speaking, “to marshal” means to convert some data from internal to external form (in an RPC buffer for instance) and “unmarshalling” for the reverse process.

`dbm` — Interfaces to Unix “databases”

Source code: Lib/dbm/__init__.py [https://github.com/python/cpython/tree/3.14/Lib/dbm/__init__.py]

`dbm` is a generic interface to variants of the DBM database:

- **`dbm.sqlite3`**
- **`dbm.gnu`**
- **`dbm.ndbm`**

If none of these modules are installed, the slow-but-simple implementation in module **`dbm.dumb`** will be used. There is a third party interface [https://www.jcea.es/programacion/pybsddb.htm] to the Oracle Berkeley DB.

exception dbm.error

A tuple containing the exceptions that can be raised by each of the supported modules, with a unique exception also named **`dbm.error`** as the first item — the latter is used when **`dbm.error`** is raised.

dbm.whichdb(*filename*)

This function attempts to guess which of the several simple database modules available — **`dbm.sqlite3`**, **`dbm.gnu`**, **`dbm.ndbm`**, or **`dbm.dumb`** — should be used to open a given file.

Return one of the following values:

- `None` if the file can’t be opened because it’s unreadable or doesn’t exist

- the empty string (`''`) if the file’s format can’t be guessed
- a string containing the required module name, such as `'dbm.ndbm'` or `'dbm.gnu'`

Changed in version 3.11: filename accepts a path-like object.

dbm.open(*file*, *flag='r'*, *mode=0o666*)

Open a database and return the corresponding database object.

Parameters::

- **file** (path-like object) – The database file to open. If the database file already exists, the `whichdb()` function is used to determine its type and the appropriate module is used; if it does not exist, the first submodule listed above that can be imported is used.
- **flag** (*str*) –
  - `'r'` (default): Open existing database for reading only.
  - `'w'`: Open existing database for reading and writing.
  - `'c'`: Open database for reading and writing, creating it if it doesn’t exist.
  - `'n'`: Always create a new, empty database, open for reading and writing.
- **mode** (*int*) – The Unix file access mode of the file (default: octal `0o666`), used onl when the database has to be created.

Changed in version 3.11: file accepts a path-like object.

The object returned by `open()` supports the basic functionality of mutable mappings; keys and their corresponding values can be stored, retrieved, and deleted, and iteration, the `in` operator and methods **`keys()`**, **`get()`**, **`setdefault()`** and **`clear()`** are available. The **`keys()`** method returns a list instead of a view object. The **`setdefault()`** method requires two arguments.

Key and values are always stored as `bytes`. This means that when strings are used they are implicitly converted to the default encoding before being stored.

These objects also support being used in a `with` statement, which will automatically close them when done.

Changed in version 3.2: **`get()`** and **`setdefault()`** methods are now available for all `dbm` backends.

Changed in version 3.4: Added native support for the context management protocol to the objects returned by `open()`.

Changed in version 3.8: Deleting a key from a read-only database raises a database module specific exception instead of `KeyError`.

Changed in version 3.13: **`clear()`** methods are now available for all `dbm` backends.

The following example records some hostnames and a corresponding title, and then prints out the contents of the database:

```
import dbm

# Open database, creating it if necessary.
with dbm.open('cache', 'c') as db:

    # Record some values
    db[b'hello'] = b'there'
    db['www.python.org'] = 'Python Website'
    db['www.cnn.com'] = 'Cable News Network'

    # Note that the keys are considered bytes now.
    assert db[b'www.python.org'] == b'Python Website'
    # Notice how the value is now in bytes.
    assert db['www.cnn.com'] == b'Cable News Network'

    # Often-used methods of the dict interface work too.
    print(db.get('python.org', b'not present'))

    # Storing a non-string key or value will raise an exception
(most
    # likely a TypeError).
    db['www.yahoo.com'] = 4

# db is automatically closed when leaving the with statement.
```

See also

Module `shelve`
Persistence module which stores non-string data.

The individual submodules are described in the following sections.

`dbm.sqlite3` — SQLite backend for dbm

Added in version 3.13.

Source code: Lib/dbm/sqlite3.py
[https://github.com/python/cpython/tree/3.14/Lib/dbm/sqlite3.py]

This module uses the standard library `sqlite3` module to provide an SQLite backend for the `dbm` module. The files created by `dbm.sqlite3` can thus be opened by `sqlite3`, or any other SQLite browser, including the SQLite CLI.

Availability: not WASI.

This module does not work or is not available on WebAssembly. See WebAssembly platforms for more information.

dbm.sqlite3.open(*filename*, */*, *flag='r'*, *mode=0o666*)

Open an SQLite database.

Parameters::

- **filename** (path-like object) – The path t the database to be opened.
- **flag** (*str*) –
  - `'r'` (default): Open existing database for reading only.
  - `'w'`: Open existing database for reading and writing.
  - `'c'`: Open database for reading and writing, creating it if it doesn't exist.
  - `'n'`: Always create a new, empty database, open for reading and writing.
- **mode** – The Unix file access mode of th file (default: octal `0o666`), used only whe the database has to be created.

The returned database object behaves similar to a mutable mapping, but the **`keys()`** method returns a list, and the **`setdefault()`** method requires two arguments. It also supports a "closing" context manager via the **`with`** keyword.

The following method is also provided:

sqlite3.close()

Close the SQLite database.

`dbm.gnu` — GNU database manager

Source code: Lib/dbm/gnu.py
[https://github.com/python/cpython/tree/3.14/Lib/dbm/gnu.py]

The **`dbm.gnu`** module provides an interface to the GDBM library, similar to the **`dbm.ndbm`** module, but with additional functionality like crash tolerance.

Note

The file formats created by **`dbm.gnu`** and **`dbm.ndbm`** are incompatible and can not be used interchangeably.

Availability: not Android, not iOS, not WASI.

This module is not supported on mobile platforms or WebAssembly platforms.

exception dbm.gnu.error

Raised on **`dbm.gnu`**-specific errors, such as I/O errors. **`KeyError`** is raised for general mapping errors like specifying an incorrect key.

dbm.gnu.open_flags

A string of characters the *flag* parameter of **`open()`** supports.

dbm.gnu.open(*filename*, *flag='r'*, *mode=0o666*, */*)

Open a GDBM database and return a **`gdbm`** object.

Parameters::

- **filename** (path-like object) – The database file to open.
- **flag** (*str*) –
  - `'r'` (default): Open existing database for reading only.
  - `'w'`: Open existing database for reading and writing.
  - `'c'`: Open database for reading and writing, creating it if it doesn't exist.
  - `'n'`: Always create a new, empty database, open for reading and writing.

  The following additional characters may be appended to control how the database is opened:
  - `'f'`: Open the database in fast mode. Writes to the database will not be synchronized.
  - `'s'`: Synchronized mode. Changes to the database will be written immediately to the file.
  - `'u'`: Do not lock database.

  Not all flags are valid for all versions of GDBM. See the **`open_flags`** member for a list of supported flag characters.
- **mode** (*int*) – The Unix file access mode of the file (default: octal `0o666`), used onl when the database has to be created.

Raises:: **error** – If an invalid *flag* argument is passed.

Changed in version 3.11: filename accepts a path-like object.

`gdbm` objects behave similar to mutable mappings, but methods **`items()`**, **`values()`**, **`pop()`**, **`popitem()`**, and **`update()`** are not supported, the **`keys()`** method returns a list, and the

`setdefault()` method requires two arguments. It also supports a “closing” context manager via the **`with`** keyword.

Changed in version 3.2: Added the **`get()`** and **`setdefault()`** methods.

Changed in version 3.13: Added the **`clear()`** method.

The following methods are also provided:

gdbm.close()

Close the GDBM database.

gdbm.firstkey()

It’s possible to loop over every key in the database using this method and the **`nextkey()`** method. The traversal is ordered by GDBM’s internal hash values, and won’t be sorted by the key values. This method returns the starting key.

gdbm.nextkey(*key*)

Returns the key that follows *key* in the traversal. The following code prints every key in the database `db`, without having to create a list in memory that contains them all:

```
k = db.firstkey()
while k is not None:
    print(k)
    k = db.nextkey(k)
```

gdbm.reorganize()

If you have carried out a lot of deletions and would like to shrink the space used by the GDBM file, this routine will reorganize the database. **`gdbm`** objects will not shorten the length of a database file except by using this reorganization; otherwise, deleted file space will be kept and reused as new (key, value) pairs are added.

gdbm.sync()

When the database has been opened in fast mode, this method forces any unwritten data to be written to the disk.

`dbm.ndbm` — New Database Manager

Source code: Lib/dbm/ndbm.py [https://github.com/python/cpython/tree/3.14/Lib/dbm/ndbm.py]

The `dbm.ndbm` module provides an interface to the NDBM library. This module can be used with the “classic” NDBM interface or the GDBM compatibility interface.

Note

The file formats created by `dbm.gnu` and `dbm.ndbm` are incompatible and can not be used interchangeably.

Warning

The NDBM library shipped as part of macOS has an undocumented limitation on the size of values, which can result in corrupted database files when storing values larger than this limit. Reading such corrupted files can result in a hard crash (segmentation fault).

Availability: not Android, not iOS, not WASI.

This module is not supported on mobile platforms or WebAssembly platforms.

exception dbm.ndbm.error

Raised on **`dbm.ndbm`**-specific errors, such as I/O errors. **`KeyError`** is raised for general mapping errors like specifying an incorrect key.

dbm.ndbm.library

Name of the NDBM implementation library used.

dbm.ndbm.open(*filename*, *flag='r'*, *mode=0o666*, */*)

Open an NDBM database and return an **`ndbm`** object.

Parameters::

- **filename** (path-like object) – The basename of the database file (without the `.dir` or `.pag` extensions).
- **flag** (*str*) –
  - `'r'` (default): Open existing database for reading only.
  - `'w'`: Open existing database for reading and writing.
  - `'c'`: Open database for reading and writing, creating it if it doesn’t exist.
  - `'n'`: Always create a new, empty database, open for reading and writing.
- **mode** (*int*) – The Unix file access mode of the file (default: octal `0o666`), used onl when the database has to be created.

Changed in version 3.11: Accepts path-like object for filename.

`ndbm` objects behave similar to mutable mappings, but methods **`items()`**, **`values()`**, **`pop()`**, **`popitem()`**, and **`update()`** are not supported, the **`keys()`** method returns a list, and the **`setdefault()`** method requires two arguments. It also supports a “closing” context manager via the **`with`** keyword.

Changed in version 3.2: Added the **`get()`** and **`setdefault()`** methods.

Changed in version 3.13: Added the **`clear()`** method.

The following method is also provided:

ndbm.close()

Close the NDBM database.

`dbm.dumb` — Portable DBM implementation

Source code: Lib/dbm/dumb.py [https://github.com/python/cpython/tree/3.14/Lib/dbm/dumb.py]

Note

The `dbm.dumb` module is intended as a last resort fallback for the `dbm` module when a more robust module is not available. The `dbm.dumb` module is not written for speed and is not nearly as heavily used as the other database modules.

The `dbm.dumb` module provides a persistent `dict`-like interface which is written entirely in Python. Unlike other `dbm` backends, such as `dbm.gnu`, no external library is required.

The **`dbm.dumb`** module defines the following:

exception dbm.dumb.error

Raised on `dbm.dumb`-specific errors, such as I/O errors. `KeyError` is raised for general mapping errors like specifying an incorrect key.

dbm.dumb.open(*filename*, *flag='c'*, *mode=0o666*)

Open a **`dbm.dumb`** database.

Parameters::

- **filename** –
  The basename of the database file (without extensions). A new database creates the following files:
  - *`filename`*`.dat`
  - *`filename`*`.dir`
- **flag** (*str*) –
  - `'r'`: Open existing database for reading only.
  - `'w'`: Open existing database for reading and writing.
  - `'c'` (default): Open database for reading and writing, creating it if it doesn’t exist.
  - `'n'`: Always create a new, empty database, open for reading and writing.
- **mode** (*int*) – The Unix file access mode of the file (default: octal `0o666`), used onl when the database has to be created.

Warning

It is possible to crash the Python interpreter when loading a database with a sufficiently large/complex entry due to stack depth limitations in Python’s AST compiler.

Changed in version 3.5: **`open()`** always creates a new database when *flag* is `'n'`.

Changed in version 3.8: A database opened read-only if *flag* is `'r'`. A database is not created if it does not exist if *flag* is `'r'` or `'w'`.

Changed in version 3.11: filename accepts a path-like object.

The returned database object behaves similar to a mutable mapping, but the `keys()` and `items()` methods return lists, and the `setdefault()` method requires two arguments. It also supports a “closing” context manager via the `with` keyword.

The following methods are also provided:

dumbdbm.close()

Close the database.

dumbdbm.sync()

Synchronize the on-disk directory and data files. This method is called by the `shelve.Shelf.sync()` method.

`sqlite3` — DB-API 2.0 interface for SQLite databases

Source code: Lib/sqlite3/ [https://github.com/python/cpython/tree/3.14/Lib/sqlite3/]

SQLite is a C library that provides a lightweight disk-based database that doesn't require a separate server process and allows accessing the database using a nonstandard variant of the SQL query language. Some applications can use SQLite for internal data storage. It's also possible to prototype an application using SQLite and then port the code to a larger database such as PostgreSQL or Oracle.

The `sqlite3` module was written by Gerhard Häring. It provides an SQL interface compliant with the DB-API 2.0 specification described by **PEP 249** [https://peps.python.org/pep-0249/], and requires the third-party SQLite [https://sqlite.org/] library.

This is an optional module. If it is missing from your copy of CPython, look for documentation from your distributor (that is, whoever provided Python to you). If you are the distributor, see Requirements for optional modules.

This document includes four main sections:

- Tutorial teaches how to use the `sqlite3` module.
- Reference describes the classes and functions this module defines.
- How-to guides details how to handle specific tasks.
- Explanation provides in-depth background on transaction control.

See also

https://www.sqlite.org
The SQLite web page; the documentation describes the syntax and the available data types for the supported SQL dialect.

https://www.w3schools.com/sql/

Tutorial, reference and examples for learning SQL syntax.

PEP 249 [https://peps.python.org/pep-0249/] - Database API Specification 2.0

PEP written by Marc-André Lemburg.

Tutorial

In this tutorial, you will create a database of Monty Python movies using basic **`sqlite3`** functionality. It assumes a fundamental understanding of database concepts, including cursors [https://en.wikipedia.org/wiki/Cursor_(databases)] and transactions [https://en.wikipedia.org/wiki/Database_transaction].

First, we need to create a new database and open a database connection to allow **`sqlite3`** to work with it. Call **`sqlite3.connect()`** to create a connection to the database `tutorial.db` in the current working directory, implicitly creating it if it does not exist:

```
import sqlite3
con = sqlite3.connect("tutorial.db")
```

The returned **`Connection`** object `con` represents the connection to the on-disk database.

In order to execute SQL statements and fetch results from SQL queries, we will need to use a database cursor. Call **`con.cursor()`** to create the **`Cursor`**:

```
cur = con.cursor()
```

Now that we’ve got a database connection and a cursor, we can create a database table `movie` with columns for title, release year, and review score. For simplicity, we can just use column names in the table declaration – thanks to the flexible typing [https://www.sqlite.org/flextypegood.html] feature of SQLite, specifying the data types is optional. Execute the `CREATE TABLE` statement by calling **`cur.execute(...)`**:

```
cur.execute("CREATE TABLE movie(title, year, score)")
```

We can verify that the new table has been created by querying the `sqlite_master` table built-in to SQLite, which should now contain an entry for the `movie` table definition (see The Schema Table [https://www.sqlite.org/schematab.html] for details). Execute that query by calling **`cur.execute(...)`**, assign the result to `res`, and call **`res.fetchone()`** to fetch the resulting row:

```
>>> res = cur.execute("SELECT name FROM sqlite_master")
>>> res.fetchone()
('movie',)
```

We can see that the table has been created, as the query returns a **`tuple`** containing the table’s name. If we query `sqlite_master` for a non-existent table `spam`, **`res.fetchone()`** will return `None`:

```
>>> res = cur.execute("SELECT name FROM sqlite_master WHERE
name='spam'")
>>> res.fetchone() is None
True
```

Now, add two rows of data supplied as SQL literals by executing an `INSERT` statement, once again by calling **`cur.execute(...)`**:

```
cur.execute("""
    INSERT INTO movie VALUES
        ('Monty Python and the Holy Grail', 1975, 8.2),
        ('And Now for Something Completely Different', 1971, 7.5)
""")
```

The `INSERT` statement implicitly opens a transaction, which needs to be committed before changes are saved in the database (see Transaction control for details). Call **`con.commit()`** on the connection object to commit the transaction:

```
con.commit()
```

We can verify that the data was inserted correctly by executing a `SELECT` query. Use the now-familiar **`cur.execute(...)`** to assign the result to `res`, and call **`res.fetchall()`** to return all resulting rows:

```
>>> res = cur.execute("SELECT score FROM movie")
>>> res.fetchall()
[(8.2,), (7.5,)]
```

The result is a `list` of two **`tuple`**s, one per row, each containing that row’s `score` value.

Now, insert three more rows by calling `cur.executemany(...)`:

```
data = [
    ("Monty Python Live at the Hollywood Bowl", 1982, 7.9),
    ("Monty Python's The Meaning of Life", 1983, 7.5),
    ("Monty Python's Life of Brian", 1979, 8.0),
]
cur.executemany("INSERT INTO movie VALUES(?, ?, ?)", data)
con.commit()  # Remember to commit the transaction after executing
INSERT.
```

Notice that `?` placeholders are used to bind `data` to the query. Always use placeholders instead of string formatting to bind Python values to SQL statements, to avoid SQL injection attacks [https://en.wikipedia.org/wiki/SQL_injection] (see How to use placeholders to bind values in SQL queries for more details).

We can verify that the new rows were inserted by executing a `SELECT` query, this time iterating over the results of the query:

```
>>> for row in cur.execute("SELECT year, title FROM movie ORDER BY
year"):
...     print(row)
(1971, 'And Now for Something Completely Different')
(1975, 'Monty Python and the Holy Grail')
(1979, "Monty Python's Life of Brian")
(1982, 'Monty Python Live at the Hollywood Bowl')
(1983, "Monty Python's The Meaning of Life")
```

Each row is a two-item `tuple` of `(year, title)`, matching the columns selected in the query.

Finally, verify that the database has been written to disk by calling `con.close()` to close the existing connection, opening a new one, creating a new cursor, then querying the database:

```
>>> con.close()
>>> new_con = sqlite3.connect("tutorial.db")
>>> new_cur = new_con.cursor()
>>> res = new_cur.execute("SELECT title, year FROM movie ORDER BY score
DESC")
>>> title, year = res.fetchone()
>>> print(f'The highest scoring Monty Python movie is {title!r},
released in {year}')
The highest scoring Monty Python movie is 'Monty Python and the Holy
Grail', released in 1975
>>> new_con.close()
```

You’ve now created an SQLite database using the **`sqlite3`** module, inserted data and retrieved values from it in multiple ways.

See also

- How-to guides for further reading:
  - How to use placeholders to bind values in SQL queries
  - How to adapt custom Python types to SQLite values
  - How to convert SQLite values to custom Python types
  - How to use the connection context manager
  - How to create and use row factories
- Explanation for in-depth background on transaction control.

Reference

Module functions

sqlite3.connect(*database*, *timeout=5.0*, *detect_types=0*, *isolation_level='DEFERRED'*, *check_same_thread=True*, *factory=sqlite3.Connection*, *cached_statements=128*, *uri=False*, ***, *autocommit=sqlite3.LEGACY_TRANSACTION_CONTROL*)

Open a connection to an SQLite database.

Parameters::

- **database** (path-like object) – The path to the database file to be opened. You can pass `":memory:"` to create an SQLite database existin only in memory [https://sqlite.org/inmemorydb.html], and open a connection to it.
- **timeout** (*float*) – How many seconds the connection should wait before raising an **`OperationalError`** when a table is locked. If another connection opens a transaction to modify a table, that table will be locked until the transaction is committed. Default five seconds.
- **detect_types** (*int*) – Control whether and how data types not natively supported by SQLite are looked up to be converted to Python types using the converters registered with **`register_converter()`**. Set it to any combination (using `|`, bitwise or) of **`PARSE_DECLTYPES`** and **`PARSE_COLNAMES`** to enable this. Column names takes precedence over declared types if both flags are set. By default (`0`), type detection is disabled.
- **isolation_level** (*str* | *None*) – Control legacy transaction handling behaviour. See **`Connection.isolation_level`** and Transaction control via the isolation_level attribute for mor information. Can be `"DEFERRED"` (default), `"EXCLUSIVE"` or `"IMMEDIATE"`; or `None` to disable opening transactions implicitly. Has no effect unless **`Connection.autocommit`** is set to **`LEGACY_TRANSACTION_CONTROL`** (the default).
- **check_same_thread** (*bool*) – If `True` (default), **`ProgrammingError`** will be raised if the database connection is used by a thread other than the one that created it. If `False`, the connection ma be accessed in multiple threads; write operations may need to be serialized by the user to avoid data corruption. See **`threadsafety`** for more information.
- **factory** (*Connection*) – A custom subclass of **`Connection`** to create the connection with, if not

`Connection` to create the connection with, if not the default `Connection` class.

- **cached_statements** (*int*) – The number of statements that **`sqlite3`** should internally cache for this connection, to avoid parsing overhead. By default, 128 statements.
- **uri** (*bool*) – If set to `True`, *database* is interpreted as a URI with a file path and an optional query string. The scheme part *must* be `"file:"`, and the path can be relative or absolute. The query string allows passing parameters to SQLite, enabling various How to work with SQLite URIs.
- **autocommit** (*bool*) – Control **PEP 249** [https://peps.python.org/pep-0249/] transaction handling behaviour. See **`Connection.autocommit`** and Transaction control via the autocommit attribute for more information. *autocommit* currently defaults to **`LEGACY_TRANSACTION_CONTROL`**. The default will change to `False` in a future Python release.

Return type:: *Connection*

Raises an auditing event `sqlite3.connect` with argument `database`.

Raises an auditing event `sqlite3.connect/handle` with argument `connection_handle`.

Changed in version 3.4: Added the *uri* parameter.

Changed in version 3.7: database can now also be a path-like object, not only a string.

Changed in version 3.10: Added the `sqlite3.connect/handle` auditing event.

Changed in version 3.12: Added the *autocommit* parameter.

Changed in version 3.13: Positional use of the parameters *timeout*, *detect_types*, *isolation_level*, *check_same_thread*, *factory*, *cached_statements*, and *uri* is deprecated. They will become keyword-only parameters in Python 3.15.

sqlite3.complete_statement(*statement*)

Return `True` if the string *statement* appears to contain one or more complete SQL statements. No syntactic verification or parsing of any kind is performed, other than checking that there are no unclosed string literals and the statement is terminated by a semicolon.

For example:

```
>>> sqlite3.complete_statement("SELECT foo FROM bar;")
True
>>> sqlite3.complete_statement("SELECT foo")
False
```

This function may be useful during command-line input to determine if the entered text seems to form a complete SQL statement, or if additional input is needed before calling **`execute()`**.

See **`runsource()`** in Lib/sqlite3/__main__.py [https://github.com/python/cpython/tree/3.14/Lib/sqlite3/__main__.py] for real-world use.

sqlite3.enable_callback_tracebacks(*flag*, /)

Enable or disable callback tracebacks. By default you will not get any tracebacks in user-defined functions, aggregates, converters, authorizer callbacks etc. If you want to debug them, you can call this function with *flag* set to `True`. Afterwards, you will get tracebacks from callbacks on **`sys.stderr`**. Use `False` to disable the feature again.

Note

Errors in user-defined function callbacks are logged as unraisable exceptions. Use an **`unraisable hook handler`** for introspection of the failed callback.

sqlite3.register_adapter(*type*, *adapter*, /)

Register an *adapter* callable to adapt the Python type *type* into an SQLite type. The adapter is called with a Python object of type *type* as its sole argument, and must return a value of a type that SQLite natively understands.

sqlite3.register_converter(*typename*, *converter*, /)

Register the *converter* callable to convert SQLite objects of type *typename* into a Python object of a specific type. The converter is invoked for all SQLite values of type *typename*; it is passed a **`bytes`** object and should return an object of the desired Python type. Consult the parameter *detect_types* of **`connect()`** for information regarding how type detection works.

Note: *typename* and the name of the type in your query are matched case-insensitively.

Module constants

sqlite3.LEGACY_TRANSACTION_CONTROL

Set **`autocommit`** to this constant to select old style (pre-Python 3.12) transaction control behaviour. See Transaction control via the isolation_level attribute for more information.

sqlite3.PARSE_DECLTYPES

Pass this flag value to the *detect_types* parameter of **`connect()`** to look up a converter function using the declared types for each column. The types are declared when the database table is created. **`sqlite3`** will look up a converter function using the first word of the declared type as the converter dictionary key. For example:

```
CREATE TABLE test(
   i integer primary key,  ! will look up a converter named
"integer"
   p point,                ! will look up a converter named "point"
   n number(10)            ! will look up a converter named "number"
 )
```

This flag may be combined with **`PARSE_COLNAMES`** using the `|` (bitwise or) operator.

Note

Generated fields (for example `MAX(p)`) are returned as **`str`**. Use **`PARSE_COLNAMES`** to enforce types for such queries.

sqlite3.PARSE_COLNAMES

Pass this flag value to the *detect_types* parameter of **`connect()`** to look up a converter function by using the type name, parsed from the query column name, as the converter dictionary key. The query column name must be wrapped in double quotes (`"`) and the type name must be wrapped in square brackets (`[]`).

```
SELECT MAX(p) as "p [point]" FROM test;  ! will look up converter
"point"
```

This flag may be combined with **`PARSE_DECLTYPES`** using the `|` (bitwise or) operator.

sqlite3.SQLITE_OK
sqlite3.SQLITE_DENY
sqlite3.SQLITE_IGNORE

Flags that should be returned by the *authorizer_callback* callable passed to **`Connection.set_authorizer()`**, to indicate whether:

- Access is allowed (**`SQLITE_OK`**),
- The SQL statement should be aborted with an error (**`SQLITE_DENY`**)
- The column should be treated as a `NULL` value (**`SQLITE_IGNORE`**)

sqlite3.apilevel

String constant stating the supported DB-API level. Required by the DB-API. Hard-coded to `"2.0"`.

sqlite3.paramstyle

String constant stating the type of parameter marker formatting expected by the **`sqlite3`** module. Required by the DB-API. Hard-coded to `"qmark"`.

Note

The `named` DB-API parameter style is also supported.

sqlite3.sqlite_version

Version number of the runtime SQLite library as a **`string`**.

sqlite3.sqlite_version_info

Version number of the runtime SQLite library as a **`tuple`** of **`integers`**.

sqlite3.threadsafety

Integer constant required by the DB-API 2.0, stating the level of thread safety the **`sqlite3`** module supports. This attribute is set based on the default threading mode [https://sqlite.org/threadsafe.html] the underlying SQLite library is compiled with. The SQLite threading modes are:

1. **Single-thread**: In this mode, all mutexes are disabled and SQLite is unsafe to use in more than a single thread at once.
2. **Multi-thread**: In this mode, SQLite can be safely used by multiple threads provided that no single database connection is used simultaneously in two or more threads.
3. **Serialized**: In serialized mode, SQLite can be safely used by multiple threads with no restriction.

The mappings from SQLite threading modes to DB-API 2.0 threadsafety levels are as follows:

SQLite threading mode	threadsafety	SQLITE_THREADSAFE	DB-API 2.0 meaning
single-thread	0	0	Threads may not share the module
multi-thread	1	2	Threads may share the module, but not connections
serialized	3	1	Threads may share the module, connections and cursors

Changed in version 3.11: Set *threadsafety* dynamically instead of hard-coding it to `1`.

sqlite3.SQLITE_DBCONFIG_DEFENSIVE
sqlite3.SQLITE_DBCONFIG_DQS_DDL
sqlite3.SQLITE_DBCONFIG_DQS_DML
sqlite3.SQLITE_DBCONFIG_ENABLE_FKEY
sqlite3.SQLITE_DBCONFIG_ENABLE_FTS3_TOKENIZER
sqlite3.SQLITE_DBCONFIG_ENABLE_LOAD_EXTENSION
sqlite3.SQLITE_DBCONFIG_ENABLE_QPSG
sqlite3.SQLITE_DBCONFIG_ENABLE_TRIGGER
sqlite3.SQLITE_DBCONFIG_ENABLE_VIEW
sqlite3.SQLITE_DBCONFIG_LEGACY_ALTER_TABLE
sqlite3.SQLITE_DBCONFIG_LEGACY_FILE_FORMAT
sqlite3.SQLITE_DBCONFIG_NO_CKPT_ON_CLOSE
sqlite3.SQLITE_DBCONFIG_RESET_DATABASE

sqlite3.SQLITE_DBCONFIG_TRIGGER_EQP
sqlite3.SQLITE_DBCONFIG_TRUSTED_SCHEMA
sqlite3.SQLITE_DBCONFIG_WRITABLE_SCHEMA

These constants are used for the **`Connection.setconfig()`** and **`getconfig()`** methods.

The availability of these constants varies depending on the version of SQLite Python was compiled with.

Added in version 3.12.

See also

https://www.sqlite.org/c3ref/c_dbconfig_defensive.html
SQLite docs: Database Connection Configuration Options

Deprecated since version 3.12, removed in version 3.14: The **`version`** and **`version_info`** constants.

Connection objects

class sqlite3.Connection

Each open SQLite database is represented by a `Connection` object, which is created using **`sqlite3.connect()`**. Their main purpose is creating **`Cursor`** objects, and Transaction control.

See also

- How to use connection shortcut methods
- How to use the connection context manager

Changed in version 3.13: A **`ResourceWarning`** is emitted if **`close()`** is not called before a **`Connection`** object is deleted.

An SQLite database connection has the following attributes and methods:

cursor(*factory=Cursor*)

Create and return a **`Cursor`** object. The cursor method accepts a single optional parameter *factory*. If supplied, this must be a callable returning an instance of **`Cursor`** or its subclasses.

blobopen(*table*, *column*, *rowid*, */*, ***, *readonly=False*, *name='main'*)

Open a **`Blob`** handle to an existing BLOB.

Parameters::

- **table** (*str*) – The name of the table where the blob is located.
- **column** (*str*) – The name of the column where the blob is located.
- **rowid** (*int*) – The row id where the blob is located.
- **readonly** (*bool*) – Set to `True` if the blob should be opened without write permission Defaults to `False`.
- **name** (*str*) – The name of the database where the blob is located. Defaults to `"main"`

Raises:: **OperationalError** – When trying to open a blob in a `WITHOUT ROWID` table.

Return type:: Blob

Note

The blob size cannot be changed using the **`Blob`** class. Use the SQL function `zeroblob` to create a blob with a fixed size.

Added in version 3.11.

commit()

Commit any pending transaction to the database. If **`autocommit`** is `True`, or there is no open transaction, this method does nothing. If **`autocommit`** is `False`, a new transaction is implicitly opened if a pending transaction was committed by this method.

rollback()

Roll back to the start of any pending transaction. If **`autocommit`** is `True`, or there is no open transaction, this method does nothing. If **`autocommit`** is `False`, a new transaction is implicitly opened if a pending transaction was rolled back by this method.

close()

Close the database connection. If `autocommit` is `False`, any pending transaction is implicitly rolled back. If **`autocommit`** is `True` or `LEGACY_TRANSACTION_CONTROL`, no implicit transaction control is executed. Make sure to `commit()` before closing to avoid losing pending changes.

execute(*sql*, *parameters=()*, */*)

Create a new `Cursor` object and call `execute()` on it with the given *sql* and *parameters*. Return the new cursor object.

executemany(*sql*, *parameters*, */*)

Create a new `Cursor` object and call `executemany()` on it with the given *sql* and *parameters*. Return the new cursor object.

executescript(*sql_script*, */*)

Create a new `Cursor` object and call `executescript()` on it with the given *sql_script*. Return the new cursor object.

create_function(*name*, *narg*, *func*, ***, *deterministic=False*)

Create or remove a user-defined SQL function.

Parameters::

- **name** (*str*) – The name of the SQL function.
- **narg** (*int*) – The number of arguments the SQL function can accept. If `-1`, it may take any number of arguments.
- **func** (callback | None) – A callable that is called when the SQL function is invoked. Th callable must return a type natively supported by SQLite. Set to `None` to remove an existing SQL function.
- **deterministic** (*bool*) – If `True`, the created SQL function is marked as deterministic [https://sqlite.org/deterministic.html], which allows SQLite to perform additional optimizations.

Changed in version 3.8: Added the *deterministic* parameter.

Example:

```
>>> import hashlib
>>> def md5sum(t):
...     return hashlib.md5(t).hexdigest()
>>> con = sqlite3.connect(":memory:")
>>> con.create_function("md5", 1, md5sum)
>>> for row in con.execute("SELECT md5(?)", (b"foo",)):
...     print(row)
('acbd18db4cc2f85cedef654fccc4a4d8',)
>>> con.close()
```

Changed in version 3.13: Passing *name*, *narg*, and *func* as keyword arguments is deprecated. These parameters will become positional-only in Python 3.15.

create_aggregate(*name*, *n_arg*, *aggregate_class*)

Create or remove a user-defined SQL aggregate function.

Parameters::

- **name** (*str*) – The name of the SQL aggregate function.
- **n_arg** (*int*) – The number of arguments the SQL aggregate function can accept. If `-1`, it may take any number of arguments.
- **aggregate_class** (class | None) – A class must implement the following methods:
  - `step()`: Add a row to the aggregate.
  - `finalize()`: Return the final result of the aggregate as a type natively supported b SQLite.

  The number of arguments that the `step()` method must accept is controlled by *n_arg*.
  Set to `None` to remove an existing SQL aggregate function.

Example:

```
class MySum:
    def __init__(self):
        self.count = 0

    def step(self, value):
        self.count += value
    def finalize(self):
        return self.count

con = sqlite3.connect(":memory:")
con.create_aggregate("mysum", 1, MySum)
cur = con.execute("CREATE TABLE test(i)")
cur.execute("INSERT INTO test(i) VALUES(1)")
cur.execute("INSERT INTO test(i) VALUES(2)")
cur.execute("SELECT mysum(i) FROM test")
print(cur.fetchone()[0])

con.close()
```

Changed in version 3.13: Passing *name*, *n_arg*, and *aggregate_class* as keyword arguments is deprecated. These parameters will become positional-only in Python 3.15.

create_window_function(*name*, *num_params*, *aggregate_class*, */*)

Create or remove a user-defined aggregate window function.

Parameters::

- **name** (*str*) – The name of the SQL aggregate window function to create or remove.
- **num_params** (*int*) – The number of arguments the SQL aggregate window function can accept. If `-1`, it may take any number of arguments.
- **aggregate_class** (class | None) – A class that must implement the following methods:
    - `step()`: Add a row to the current window.
    - `value()`: Return the current value of the aggregate.
    - `inverse()`: Remove a row from the current window.
    - `finalize()`: Return the final result of the aggregate as a type natively supported b SQLite.

  The number of arguments that the `step()` and `value()` methods must accept is controlled by *num_params*.

  Set to `None` to remove an existing SQL aggregate window function.

Raises:: **NotSupportedError** – If used with a version of SQLite older than 3.25.0, which does not support aggregate window functions.

Added in version 3.11.

Example:

```
# Example taken from
https://www.sqlite.org/windowfunctions.html#udfwinfunc
class WindowSumInt:
    def __init__(self):
        self.count = 0

    def step(self, value):
        """Add a row to the current window."""
        self.count += value

    def value(self):
        """Return the current value of the aggregate."""
        return self.count

    def inverse(self, value):
        """Remove a row from the current window."""
        self.count -= value

    def finalize(self):
        """Return the final value of the aggregate.

        Any clean-up actions should be placed here.
        """
        return self.count


con = sqlite3.connect(":memory:")
cur = con.execute("CREATE TABLE test(x, y)")
values = [
    ("a", 4),
    ("b", 5),
    ("c", 3),
    ("d", 8),
    ("e", 1),
]
cur.executemany("INSERT INTO test VALUES(?, ?)", values)
con.create_window_function("sumint", 1, WindowSumInt)
cur.execute("""
    SELECT x, sumint(y) OVER (
        ORDER BY x ROWS BETWEEN 1 PRECEDING AND 1 FOLLOWING
    ) AS sum_y
    FROM test ORDER BY x
""")
print(cur.fetchall())
con.close()
```

create_collation(*name*, *callable*, */*)

Create a collation named *name* using the collating function *callable*. *callable* is passed two **`string`** arguments, and it should return an **`integer`**:

- `1` if the first is ordered higher than the second
- `-1` if the first is ordered lower than the second
- `0` if they are ordered equal

The following example shows a reverse sorting collation:

```
def collate_reverse(string1, string2):
    if string1 == string2:
        return 0
    elif string1 < string2:
        return 1
    else:
        return -1

con = sqlite3.connect(":memory:")
con.create_collation("reverse", collate_reverse)

cur = con.execute("CREATE TABLE test(x)")
cur.executemany("INSERT INTO test(x) VALUES(?)", [("a",),
("b",)])
cur.execute("SELECT x FROM test ORDER BY x COLLATE reverse")
for row in cur:
    print(row)
con.close()
```

Remove a collation function by setting *callable* to `None`.

Changed in version 3.11: The collation name can contain any Unicode character. Earlier, only ASCII characters were allowed.

interrupt()

Call this method from a different thread to abort any queries that might be executing on the connection. Aborted queries will raise an **`OperationalError`**.

set_authorizer(*authorizer_callback*)

Register callable *authorizer_callback* to be invoked for each attempt to access a column of a table in the database. The callback should return one of **`SQLITE_OK`**, **`SQLITE_DENY`**, or **`SQLITE_IGNORE`** to signal how access to the column should be handled by the underlying SQLite library.

The first argument to the callback signifies what kind of operation is to be authorized. The second and third argument will be arguments or `None` depending on the first argument. The 4th argument is the name of the database (“main”, “temp”, etc.) if applicable. The 5th argument is the name of the inner-most trigger or view that is responsible for the access attempt or `None` if this access attempt is directly from input SQL code.

Please consult the SQLite documentation about the possible values for the first argument and the meaning of the second and third argument depending on the first one. All necessary constants are available in the **`sqlite3`** module.

Passing `None` as *authorizer_callback* will disable the authorizer.

Changed in version 3.11: Added support for disabling the authorizer using `None`.

Changed in version 3.13: Passing *authorizer_callback* as a keyword argument is deprecated. The parameter will become positional-only in Python 3.15.

set_progress_handler(*progress_handler*, *n*)

Register callable *progress_handler* to be invoked for every *n* instructions of the SQLite virtual machine. This is useful if you want to get called from SQLite during long-running operations, for example to update a GUI.

If you want to clear any previously installed progress handler, call the method with `None` for *progress_handler*.

Returning a non-zero value from the handler function will terminate the currently executing query and cause it to raise a **`DatabaseError`** exception.

Changed in version 3.13: Passing *progress_handler* as a keyword argument is deprecated. The parameter will become positional-only in Python 3.15.

set_trace_callback(*trace_callback*)

Register callable *trace_callback* to be invoked for each SQL statement that is actually executed by the SQLite backend.

The only argument passed to the callback is the statement (as **`str`**) that is being executed. The return value of the callback is ignored. Note that the backend does not only run statements passed to the **`Cursor.execute()`** methods. Other sources include the transaction management of the **`sqlite3`** module and the execution of triggers defined in the current database.

Passing `None` as *trace_callback* will disable the trace callback.

Note

Exceptions raised in the trace callback are not propagated. As a development and debugging aid, use **`enable_callback_tracebacks()`** to enable printing tracebacks from exceptions raised in the trace callback.

Added in version 3.3.

Changed in version 3.13: Passing *trace_callback* as a keyword argument is deprecated. The parameter will become positional-only in Python 3.15.

enable_load_extension(*enabled*, /)

Enable the SQLite engine to load SQLite extensions from shared libraries if *enabled* is `True`; else, disallow loading SQLite extensions. SQLite extensions can define new functions, aggregates or whole new virtual table implementations. One well-known extension is the fulltext-search extension distributed with SQLite.

Note

The **`sqlite3`** module is not built with loadable extension support by default, because some platforms (notably macOS) have SQLite libraries which are compiled without this feature. To get loadable extension support, you must pass the **`--enable-loadable-sqlite-extensions`** option to **configure**.

Raises an auditing event `sqlite3.enable_load_extension` with arguments `connection`, `enabled`.

Added in version 3.2.

Changed in version 3.10: Added the `sqlite3.enable_load_extension` auditing event.

```
con.enable_load_extension(True)

# Load the fulltext search extension
con.execute("select load_extension('./fts3.so')")

# alternatively you can load the extension using an API call:
# con.load_extension("./fts3.so")

# disable extension loading again
con.enable_load_extension(False)

# example from SQLite wiki
con.execute("CREATE VIRTUAL TABLE recipe USING fts3(name,
ingredients)")
con.executescript("""
    INSERT INTO recipe (name, ingredients) VALUES('broccoli
stew', 'broccoli peppers cheese tomatoes');
    INSERT INTO recipe (name, ingredients) VALUES('pumpkin stew',
'pumpkin onions garlic celery');
    INSERT INTO recipe (name, ingredients) VALUES('broccoli pie',
'broccoli cheese onions flour');
    INSERT INTO recipe (name, ingredients) VALUES('pumpkin pie',
'pumpkin sugar flour butter');
    """)
```

```
for row in con.execute("SELECT rowid, name, ingredients FROM
recipe WHERE name MATCH 'pie'"):
    print(row)
```

load_extension(*path*, /, *, *entrypoint=None*)

Load an SQLite extension from a shared library. Enable extension loading with `enable_load_extension()` before calling this method.

Parameters::

- **path** (*str*) – The path to the SQLite extension.
- **entrypoint** (*str* | *None*) – Entry point name If `None` (the default), SQLite will come up wit an entry point name of its own; see the SQLite docs Loading an Extension [https://www.sqlite.org/loadext.html#loading_an_extensio] for details.

Raises an auditing event `sqlite3.load_extension` with arguments `connection`, `path`.

Added in version 3.2.

Changed in version 3.10: Added the `sqlite3.load_extension` auditing event.

Changed in version 3.12: Added the *entrypoint* parameter.

iterdump(*, *filter=None*)

Return an iterator to dump the database as SQL source code. Useful when saving an in-memory database for later restoration. Similar to the `.dump` command in the **sqlite3** shell.

Parameters:: **filter** (*str* | *None*) – An optional `LIKE` pattern for database objects to dump, e.g. `prefix_%`. If `None` (the default), all database objects will be included.

Example:

```
# Convert file example.db to SQL dump file dump.sql
con = sqlite3.connect('example.db')
with open('dump.sql', 'w') as f:
    for line in con.iterdump():
        f.write('%s\n' % line)
con.close()
```

See also

How to handle non-UTF-8 text encodings

Changed in version 3.13: Added the *filter* parameter.

backup(*target*, *, *pages=-1*, *progress=None*, *name='main'*, *sleep=0.250*)

Create a backup of an SQLite database.

Works even if the database is being accessed by other clients or concurrently by the same connection.

Parameters::

- **target** (*Connection*) – The database connection to save the backup to.
- **pages** (*int*) – The number of pages to copy at a time. If equal to or less than `0`, the entire database is copied in a single step. Defaults to `-1`.
- **progress** (callback | None) – If set to a callable, it is invoked with three integer arguments for every backup iteration: the *status* of the last iteration, the *remaining* number of pages still to be copied, and the *total* number of pages. Defaults to `None`.
- **name** (*str*) – The name of the database to back up. Either `"main"` (the default) for the main database, `"temp"` for the temporary database, or the name of a custom database as attached using the `ATTACH DATABASE` SQL statement.
- **sleep** (*float*) – The number of seconds to sleep between successive attempts to back up remaining pages.

Example 1, copy an existing database into another:

```
def progress(status, remaining, total):
    print(f'Copied {total-remaining} of {total} pages...')

src = sqlite3.connect('example.db')
dst = sqlite3.connect('backup.db')
with dst:
    src.backup(dst, pages=1, progress=progress)
dst.close()
src.close()
```

Example 2, copy an existing database into a transient copy:

```
src = sqlite3.connect('example.db')
dst = sqlite3.connect(':memory:')
src.backup(dst)
dst.close()
src.close()
```

Added in version 3.7.

See also

How to handle non-UTF-8 text encodings

getlimit(*category*, */*)

Get a connection runtime limit.

Parameters::	**category** (*int*) – The SQLite limit category [https://www.sqlite.org/c3ref/c_limit_attached.html] to be queried.
Return type::	int
Raises::	**ProgrammingError** – If *category* is not recognised by the underlying SQLite library.

Example, query the maximum length of an SQL statement for **`Connection`** `con` (the default is 1000000000):

```
>>> con.getlimit(sqlite3.SQLITE_LIMIT_SQL_LENGTH)
1000000000
```

Added in version 3.11.

setlimit(*category*, *limit*, */*)

Set a connection runtime limit. Attempts to increase a limit above its hard upper bound are silently truncated to the hard upper bound. Regardless of whether or not the limit was changed, the prior value of the limit is returned.

Parameters::
- **category** (*int*) – The SQLite limit category [https://www.sqlite.org/c3ref/c_limit_attached.html] to be set.
- **limit** (*int*) – The value of the new limit. If negative, the current limit is unchanged.

Return type:: int

Raises:: **ProgrammingError** – If *category* is not recognised by the underlying SQLite library.

Example, limit the number of attached databases to 1 for `Connection` `con` (the default limit is 10):

```
>>> con.setlimit(sqlite3.SQLITE_LIMIT_ATTACHED, 1)
10
>>> con.getlimit(sqlite3.SQLITE_LIMIT_ATTACHED)
1
```

Added in version 3.11.

getconfig(*op*, /)

Query a boolean connection configuration option.

Parameters:: **op** (*int*) – A SQLITE_DBCONFIG code.

Return type:: bool

Added in version 3.12.

setconfig(*op*, *enable=True*, /)

Set a boolean connection configuration option.

Parameters::
- **op** (*int*) – A SQLITE_DBCONFIG code.
- **enable** (*bool*) – `True` if the configuration option should be enabled (default); `False` if i should be disabled.

Added in version 3.12.

serialize(*, *name='main'*)

Serialize a database into a [`bytes`] object. For an ordinary on-disk database file, the serialization is just a copy of the disk file. For an in-memory database or a "temp" database, the serialization is the same sequence of bytes which would be written to disk if that database were backed up to disk.

Parameters::	**name** (*str*) – The database name to be serialized. Defaults to `"main"`.
Return type::	bytes

Note

This method is only available if the underlying SQLite library has the serialize API.

Added in version 3.11.

deserialize(*data*, */*, ***, *name='main'*)

Deserialize a `serialized` database into a `Connection`. This method causes the database connection to disconnect from database *name*, and reopen *name* as an in-memory database based on the serialization contained in *data*.

Parameters::	- **data** (*bytes*) – A serialized database. - **name** (*str*) – The database name to deserialize into. Defaults to `"main"`.
Raises::	- **OperationalError** – If the database connection is currently involved in a read transaction or a backup operation. - **DatabaseError** – If *data* does not contain a valid SQLite database. - **OverflowError** – If `len(data)` is larger than `2**63 - 1`.

Note

This method is only available if the underlying SQLite library has the deserialize API.

Added in version 3.11.

autocommit

This attribute controls **PEP 249** [https://peps.python.org/pep-0249/]-compliant transaction behaviour. **`autocommit`** has three allowed values:

- `False`: Select **PEP 249** [https://peps.python.org/pep-0249/]-compliant transaction behaviour, implying that **`sqlite3`** ensures a transaction is always open. Use **`commit()`** and **`rollback()`** to close transactions.

  This is the recommended value of **`autocommit`**.

- `True`: Use SQLite's autocommit mode [https://www.sqlite.org/lang_transaction.html#implicit_versus_explicit_transactions]. **`commit()`** and **`rollback()`** have no effect in this mode.

- **`LEGACY_TRANSACTION_CONTROL`**: Pre-Python 3.12 (non-**PEP 249** [https://peps.python.org/pep-0249/]-compliant) transaction control. See **`isolation_level`** for more details.

  This is currently the default value of **`autocommit`**.

Changing **`autocommit`** to `False` will open a new transaction, and changing it to `True` will commit any pending transaction.

See Transaction control via the autocommit attribute for more details.

Note

The **`isolation_level`** attribute has no effect unless **`autocommit`** is **`LEGACY_TRANSACTION_CONTROL`**.

Added in version 3.12.

in_transaction

This read-only attribute corresponds to the low-level SQLite autocommit mode [https://www.sqlite.org/lang_transaction.html#implicit_versus_explicit_transactions].

`True` if a transaction is active (there are uncommitted changes), `False` otherwise.

Added in version 3.2.

isolation_level

Controls the legacy transaction handling mode of **`sqlite3`**. If set to `None`, transactions are never implicitly opened. If set to one of `"DEFERRED"`, `"IMMEDIATE"`, or `"EXCLUSIVE"`, corresponding to the underlying SQLite transaction behaviour [https://www.sqlite.org/lang_transaction.html#deferred_immediate_and_exclusive_transactions], implicit transaction management is performed.

If not overridden by the *isolation_level* parameter of **`connect()`**, the default is `""`, which is an alias for `"DEFERRED"`.

Note

Using **`autocommit`** to control transaction handling is recommended over using **`isolation_level`**. **`isolation_level`** has no effect unless **`autocommit`** is set to **`LEGACY_TRANSACTION_CONTROL`** (the default).

row_factory

The initial **`row_factory`** for **`Cursor`** objects created from this connection. Assigning to this attribute does not affect the **`row_factory`** of existing cursors belonging to this connection, only new ones. Is `None` by default, meaning each row is returned as a **`tuple`**.

See How to create and use row factories for more details.

text_factory

A callable that accepts a **`bytes`** parameter and returns a text representation of it. The callable is invoked for SQLite values with the `TEXT` data type. By default, this attribute is set to **`str`**.

See How to handle non-UTF-8 text encodings for more details.

total_changes

Return the total number of database rows that have been modified, inserted, or deleted since the database connection was opened.

Cursor objects

A `Cursor` object represents a database cursor [https://en.wikipedia.org/wiki/Cursor_(databases)] which is used to execute SQL statements, and manage the context of a fetch operation. Cursors are created using **`Connection.cursor()`**, or by using any of the connection shortcut methods.

Cursor objects are iterators, meaning that if you **`execute()`** a `SELECT` query, you can simply iterate over the cursor to fetch the resulting rows:

```
for row in cur.execute("SELECT t FROM data"):
    print(row)
```

class sqlite3.Cursor

A **`Cursor`** instance has the following attributes and methods.

execute(*sql*, *parameters=()*, */*)

Execute a single SQL statement, optionally binding Python values using placeholders.

Parameters::
- **sql** (*str*) – A single SQL statement.
- **parameters** (**`dict`** | sequence) – Python values to bind to placeholders in *sql*. A **`dict`** if named placeholders are used. A sequence if unnamed placeholders are used. See How to use placeholders to bind values in SQL queries.

Raises:: **ProgrammingError** – When *sql* contains more than one SQL statement. When named placeholders are used and *parameters* is a sequence instead of a **`dict`**.

If **`autocommit`** is **`LEGACY_TRANSACTION_CONTROL`**, **`isolation_level`** is not `None`, *sql* is an `INSERT`, `UPDATE`, `DELETE`, or `REPLACE` statement, and there is no open transaction, a transaction is implicitly opened before executing *sql*.

Changed in version 3.14: **`ProgrammingError`** is emitted if named placeholders are used and *parameters* is a sequence instead of a **`dict`**.

Use **`executescript()`** to execute multiple SQL statements.

executemany(*sql*, *parameters*, */*)

For every item in *parameters*, repeatedly execute the parameterized DML SQL statement *sql*.

Uses the same implicit transaction handling as **`execute()`**.

Parameters::

- **sql** (*str*) – A single SQL DML statement.
- **parameters** (iterable) – An iterable of parameters to bind with the placeholders in *sql*. See How to use placeholders to bind values in SQL queries.

Raises:: **ProgrammingError** – When *sql* contains more than one SQL statement or is not a DML statement, When named placeholders are used and the items in *parameters* are sequences instead of **`dict`**s.

Example:

```
rows = [
    ("row1",),
    ("row2",),
]
# cur is an sqlite3.Cursor object
cur.executemany("INSERT INTO data VALUES(?)", rows)
```

Note

Any resulting rows are discarded, including DML statements with RETURNING clauses [https://www.sqlite.org/lang_returning.html].

Changed in version 3.14: **`ProgrammingError`** is emitted if named placeholders are used and the items in *parameters* are sequences instead of **`dict`**s.

executescript(*sql_script*, */*)

Execute the SQL statements in *sql_script*. If the `autocommit` is `LEGACY_TRANSACTION_CONTROL` and there is a pending transaction, an implicit `COMMIT` statement is executed first. No other implicit transaction control is performed; any transaction control must be added to *sql_script*.

sql_script must be a `string`.

Example:

```
# cur is an sqlite3.Cursor object
cur.executescript("""
    BEGIN;
    CREATE TABLE person(firstname, lastname, age);
    CREATE TABLE book(title, author, published);
    CREATE TABLE publisher(name, address);
    COMMIT;
""")
```

fetchone()

If `row_factory` is `None`, return the next row query result set as a `tuple`. Else, pass it to the row factory and return its result. Return `None` if no more data is available.

fetchmany(*size=cursor.arraysize*)

Return the next set of rows of a query result as a `list`. Return an empty list if no more rows are available.

The number of rows to fetch per call is specified by the *size* parameter. If *size* is not given, `arraysize` determines the number of rows to be fetched. If fewer than *size* rows are available, as many rows as are available are returned.

Note there are performance considerations involved with the *size* parameter. For optimal performance, it is usually best to use the arraysize attribute. If the *size* parameter is used, then it is best for it to retain the same value from one `fetchmany()` call to the next.

Changed in version 3.14.1: Negative *size* values are rejected by raising `ValueError`.

fetchall()

Return all (remaining) rows of a query result as a `list`. Return an empty list if no rows are available. Note that the `arraysize` attribute can affect the performance of this operation.

close()

Close the cursor now (rather than whenever `__del__` is called).

The cursor will be unusable from this point forward; a `ProgrammingError` exception will be raised if any operation is attempted with the cursor.

setinputsizes(*sizes*, */*)

Required by the DB-API. Does nothing in **`sqlite3`**.

setoutputsize(*size*, *column=None*, */*)

Required by the DB-API. Does nothing in **`sqlite3`**.

arraysize

Read/write attribute that controls the number of rows returned by `fetchmany()`. The default value is 1 which means a single row would be fetched per call.

Changed in version 3.14.1: Negative values are rejected by raising `ValueError`.

connection

Read-only attribute that provides the SQLite database `Connection` belonging to the cursor. A `Cursor` object created by calling `con.cursor()` will have a `connection` attribute that refers to *con*:

```
>>> con = sqlite3.connect(":memory:")
>>> cur = con.cursor()
>>> cur.connection == con
True
>>> con.close()
```

description

Read-only attribute that provides the column names of the last query. To remain compatible with the Python DB API, it returns a 7-tuple for each column where the last six items of each tuple are `None`.

It is set for `SELECT` statements without any matching rows as well.

lastrowid

Read-only attribute that provides the row id of the last inserted row. It is only updated after successful `INSERT` or `REPLACE` statements using the **`execute()`** method. For other statements, after **`executemany()`** or **`executescript()`**, or if the insertion failed, the value of `lastrowid` is left unchanged. The initial value of `lastrowid` is `None`.

Note

Inserts into `WITHOUT ROWID` tables are not recorded.

Changed in version 3.6: Added support for the `REPLACE` statement.

rowcount

Read-only attribute that provides the number of modified rows for `INSERT`, `UPDATE`, `DELETE`, and `REPLACE` statements; is `-1` for other statements, including CTE queries. It is only updated by the **`execute()`** and **`executemany()`** methods, after the statement has run to completion. This means that any resulting rows must be fetched in order for **`rowcount`** to be updated.

row_factory

Control how a row fetched from this **`Cursor`** is represented. If `None`, a row is represented as a **`tuple`**. Can be set to the included **`sqlite3.Row`**; or a callable that accepts two arguments, a **`Cursor`** object and the **`tuple`** of row values, and returns a custom object representing an SQLite row.

Defaults to what **`Connection.row_factory`** was set to when the **`Cursor`** was created. Assigning to this attribute does not affect **`Connection.row_factory`** of the parent connection.

See How to create and use row factories for more details.

Row objects

class sqlite3.Row

A **`Row`** instance serves as a highly optimized **`row_factory`** for **`Connection`** objects. It supports iteration, equality testing, **`len()`**, and mapping access by column name and index.

Two **`Row`** objects compare equal if they have identical column names and values.

See How to create and use row factories for more details.

keys()

Return a **`list`** of column names as **`strings`**. Immediately after a query, it is the first member of each tuple in **`Cursor.description`**.

Changed in version 3.5: Added support of slicing.

Blob objects

class sqlite3.Blob

Added in version 3.11.

A **`Blob`** instance is a file-like object that can read and write data in an SQLite BLOB. Call **`len(blob)`** to get the size (number of bytes) of the blob. Use indices and slices for direct access to the blob data.

Use the **`Blob`** as a context manager to ensure that the blob handle is closed after use.

```
con = sqlite3.connect(":memory:")
con.execute("CREATE TABLE test(blob_col blob)")
con.execute("INSERT INTO test(blob_col) VALUES(zeroblob(13))")

# Write to our blob, using two write operations:
with con.blobopen("test", "blob_col", 1) as blob:
    blob.write(b"hello, ")
    blob.write(b"world.")
    # Modify the first and last bytes of our blob
    blob[0] = ord("H")
    blob[-1] = ord("!")

# Read the contents of our blob
with con.blobopen("test", "blob_col", 1) as blob:
    greeting = blob.read()

print(greeting)  # outputs "b'Hello, world!'"
con.close()
```

close()

Close the blob.

The blob will be unusable from this point onward. An `Error` (or subclass) exception will be raised if any further operation is attempted with the blob.

read(*length=-1*, */*)

Read *length* bytes of data from the blob at the current offset position. If the end of the blob is reached, the data up to EOF will be returned. When *length* is not specified, or is negative, `read()` will read until the end of the blob.

write(*data*, */*)

Write *data* to the blob at the current offset. This function cannot change the blob length. Writing beyond the end of the blob will raise `ValueError`.

tell()

Return the current access position of the blob.

seek(*offset*, *origin=os.SEEK_SET*, */*)

Set the current access position of the blob to *offset*. The *origin* argument defaults to `os.SEEK_SET` (absolute blob positioning). Other values for *origin* are `os.SEEK_CUR` (seek relative to the current position) and `os.SEEK_END` (seek relative to the blob’s end).

PrepareProtocol objects

class sqlite3.PrepareProtocol

The PrepareProtocol type’s single purpose is to act as a **PEP 246** [https://peps.python.org/pep-0246/] style adaption protocol for objects that can adapt themselves to native SQLite types.

Exceptions

The exception hierarchy is defined by the DB-API 2.0 (**PEP 249** [https://peps.python.org/pep-0249/]).

exception sqlite3.Warning

This exception is not currently raised by the **`sqlite3`** module, but may be raised by applications using **`sqlite3`**, for example if a user-defined function truncates data while inserting. `Warning` is a subclass of **`Exception`**.

exception sqlite3.Error

The base class of the other exceptions in this module. Use this to catch all errors with one single **`except`** statement. `Error` is a subclass of **`Exception`**.

If the exception originated from within the SQLite library, the following two attributes are added to the exception:

sqlite_errorcode

The numeric error code from the SQLite API [https://sqlite.org/rescode.html]

Added in version 3.11.

sqlite_errorname

The symbolic name of the numeric error code from the SQLite API [https://sqlite.org/rescode.html]

Added in version 3.11.

exception sqlite3.InterfaceError

Exception raised for misuse of the low-level SQLite C API. In other words, if this exception is raised, it probably indicates a bug in the **`sqlite3`** module. `InterfaceError` is a subclass of **`Error`**.

exception sqlite3.DatabaseError

Exception raised for errors that are related to the database. This serves as the base exception for several types of database errors. It is only raised implicitly through the specialised subclasses. `DatabaseError` is a subclass of [`Error`].

exception sqlite3.DataError

Exception raised for errors caused by problems with the processed data, like numeric values out of range, and strings which are too long. `DataError` is a subclass of `DatabaseError`.

exception sqlite3.OperationalError

Exception raised for errors that are related to the database’s operation, and not necessarily under the control of the programmer. For example, the database path is not found, or a transaction could not be processed. `OperationalError` is a subclass of `DatabaseError`.

exception sqlite3.IntegrityError

Exception raised when the relational integrity of the database is affected, e.g. a foreign key check fails. It is a subclass of `DatabaseError`.

exception sqlite3.InternalError

Exception raised when SQLite encounters an internal error. If this is raised, it may indicate that there is a problem with the runtime SQLite library. `InternalError` is a subclass of `DatabaseError`.

exception sqlite3.ProgrammingError

Exception raised for **`sqlite3`** API programming errors, for example supplying the wrong number of bindings to a query, or trying to operate on a closed `Connection`. `ProgrammingError` is a subclass of `DatabaseError`.

exception sqlite3.NotSupportedError

Exception raised in case a method or database API is not supported by the underlying SQLite library. For example, setting *deterministic* to `True` in `create_function()`, if the underlying SQLite library does not support deterministic functions. `NotSupportedError` is a subclass of `DatabaseError`.

SQLite and Python types

SQLite natively supports the following types: `NULL`, `INTEGER`, `REAL`, `TEXT`, `BLOB`.

The following Python types can thus be sent to SQLite without any problem:

Python type	SQLite type
`None`	`NULL`
`int`	`INTEGER`
`float`	`REAL`
`str`	`TEXT`
`bytes`	`BLOB`

This is how SQLite types are converted to Python types by default:

SQLite type	Python type
`NULL`	`None`
`INTEGER`	`int`
`REAL`	`float`
`TEXT`	depends on `text_factory`, `str` by default

SQLite type	Python type
`BLOB`	`bytes`

The type system of the `sqlite3` module is extensible in two ways: you can store additional Python types in an SQLite database via object adapters, and you can let the `sqlite3` module convert SQLite types to Python types via converters.

Default adapters and converters (deprecated)

Note

The default adapters and converters are deprecated as of Python 3.12. Instead, use the Adapter and converter recipes and tailor them to your needs.

The deprecated default adapters and converters consist of:

- An adapter for `datetime.date` objects to `strings` in ISO 8601 [https://en.wikipedia.org/wiki/ISO_8601] format.
- An adapter for `datetime.datetime` objects to strings in ISO 8601 format.
- A converter for declared “date” types to `datetime.date` objects.
- A converter for declared “timestamp” types to `datetime.datetime` objects. Fractional parts will be truncated to 6 digits (microsecond precision).

Note

The default “timestamp” converter ignores UTC offsets in the database and always returns a naive `datetime.datetime` object. To preserve UTC offsets in timestamps, either leave converters disabled, or register an offset-aware converter with `register_converter()`.

Deprecated since version 3.12.

Command-line interface

The **`sqlite3`** module can be invoked as a script, using the interpreter's `-m` switch, in order to provide a simple SQLite shell. The argument signature is as follows:

```
python -m sqlite3 [-h] [-v] [filename] [sql]
```

Type `.quit` or CTRL-D to exit the shell.

-h, --help
    Print CLI help.

-v, --version
    Print underlying SQLite library version.

Added in version 3.12.

How-to guides

How to use placeholders to bind values in SQL queries

SQL operations usually need to use values from Python variables. However, beware of using Python's string operations to assemble queries, as they are vulnerable to SQL injection attacks [https://en.wikipedia.org/wiki/SQL_injection]. For example, an attacker can simply close the single quote and inject `OR TRUE` to select all rows:

```
>>> # Never do this -- insecure!
>>> symbol = input()
' OR TRUE; --
>>> sql = "SELECT * FROM stocks WHERE symbol = '%s'" % symbol
>>> print(sql)
SELECT * FROM stocks WHERE symbol = '' OR TRUE; --'
>>> cur.execute(sql)
```

Instead, use the DB-API's parameter substitution. To insert a variable into a query string, use a placeholder in the string, and substitute the actual values into the query by providing them as a

`tuple` of values to the second argument of the cursor's **`execute()`** method.

An SQL statement may use one of two kinds of placeholders: question marks (qmark style) or named placeholders (named style). For the qmark style, *parameters* must be a sequence whose length must match the number of placeholders, or a **`ProgrammingError`** is raised. For the named style, *parameters* must be an instance of a **`dict`** (or a subclass), which must contain keys for all named parameters; any extra items are ignored. Here's an example of both styles:

```
con = sqlite3.connect(":memory:")
cur = con.execute("CREATE TABLE lang(name, first_appeared)")

# This is the named style used with executemany():
data = (
    {"name": "C", "year": 1972},
    {"name": "Fortran", "year": 1957},
    {"name": "Python", "year": 1991},
    {"name": "Go", "year": 2009},
)
cur.executemany("INSERT INTO lang VALUES(:name, :year)", data)

# This is the qmark style used in a SELECT query:
params = (1972,)
cur.execute("SELECT * FROM lang WHERE first_appeared = ?", params)
print(cur.fetchall())
con.close()
```

Note

PEP 249 [https://peps.python.org/pep-0249/] numeric placeholders are *not* supported. If used, they will be interpreted as named placeholders.

How to adapt custom Python types to SQLite values

SQLite supports only a limited set of data types natively. To store custom Python types in SQLite databases, *adapt* them to one of the Python types SQLite natively understands.

There are two ways to adapt Python objects to SQLite types: letting your object adapt itself, or using an *adapter callable*. The latter will take precedence above the former. For a library that exports a custom type, it may make sense to enable that type to adapt itself. As an application developer, it may make more sense to take direct control by registering custom adapter functions.

How to write adaptable objects

Suppose we have a **`Point`** class that represents a pair of coordinates, `x` and `y`, in a Cartesian coordinate system. The coordinate pair will be stored as a text string in the database, using a semicolon to separate the coordinates. This can be implemented by adding a `__conform__(self, protocol)` method which returns the adapted value. The object passed to *protocol* will be of type **`PrepareProtocol`**.

```
class Point:
    def __init__(self, x, y):
        self.x, self.y = x, y

    def __conform__(self, protocol):
        if protocol is sqlite3.PrepareProtocol:
            return f"{self.x};{self.y}"

con = sqlite3.connect(":memory:")
cur = con.cursor()

cur.execute("SELECT ?", (Point(4.0, -3.2),))
print(cur.fetchone()[0])
con.close()
```

How to register adapter callables

The other possibility is to create a function that converts the Python object to an SQLite-compatible type. This function can then be registered using **`register_adapter()`**.

```
class Point:
    def __init__(self, x, y):
        self.x, self.y = x, y

def adapt_point(point):
    return f"{point.x};{point.y}"
```

```
sqlite3.register_adapter(Point, adapt_point)

con = sqlite3.connect(":memory:")
cur = con.cursor()

cur.execute("SELECT ?", (Point(1.0, 2.5),))
print(cur.fetchone()[0])
con.close()
```

How to convert SQLite values to custom Python types

Writing an adapter lets you convert *from* custom Python types *to* SQLite values. To be able to convert *from* SQLite values *to* custom Python types, we use *converters*.

Let’s go back to the **`Point`** class. We stored the x and y coordinates separated via semicolons as strings in SQLite.

First, we’ll define a converter function that accepts the string as a parameter and constructs a **`Point`** object from it.

Note

Converter functions are **always** passed a **`bytes`** object, no matter the underlying SQLite data type.

```
def convert_point(s):
    x, y = map(float, s.split(b";"))
    return Point(x, y)
```

We now need to tell **`sqlite3`** when it should convert a given SQLite value. This is done when connecting to a database, using the *detect_types* parameter of **`connect()`**. There are three options:

- Implicit: set *detect_types* to **`PARSE_DECLTYPES`**
- Explicit: set *detect_types* to **`PARSE_COLNAMES`**
- Both: set *detect_types* to `sqlite3.PARSE_DECLTYPES | sqlite3.PARSE_COLNAMES`. Column names take precedence over declared types.

The following example illustrates the implicit and explicit approaches:

```
class Point:
    def __init__(self, x, y):
        self.x, self.y = x, y

    def __repr__(self):
        return f"Point({self.x}, {self.y})"

def adapt_point(point):
    return f"{point.x};{point.y}"

def convert_point(s):
    x, y = list(map(float, s.split(b";")))
    return Point(x, y)

# Register the adapter and converter
sqlite3.register_adapter(Point, adapt_point)
sqlite3.register_converter("point", convert_point)

# 1) Parse using declared types
p = Point(4.0, -3.2)
con = sqlite3.connect(":memory:", detect_types=sqlite3.PARSE_DECLTYPES)
cur = con.execute("CREATE TABLE test(p point)")

cur.execute("INSERT INTO test(p) VALUES(?)", (p,))
cur.execute("SELECT p FROM test")
print("with declared types:", cur.fetchone()[0])
cur.close()
con.close()

# 2) Parse using column names
con = sqlite3.connect(":memory:", detect_types=sqlite3.PARSE_COLNAMES)
cur = con.execute("CREATE TABLE test(p)")

cur.execute("INSERT INTO test(p) VALUES(?)", (p,))
cur.execute('SELECT p AS "p [point]" FROM test')
print("with column names:", cur.fetchone()[0])
cur.close()
con.close()
```

Adapter and converter recipes

This section shows recipes for common adapters and converters.

```
import datetime
import sqlite3

def adapt_date_iso(val):
    """Adapt datetime.date to ISO 8601 date."""
    return val.isoformat()

def adapt_datetime_iso(val):
    """Adapt datetime.datetime to timezone-naive ISO 8601 date."""
    return val.replace(tzinfo=None).isoformat()

def adapt_datetime_epoch(val):
    """Adapt datetime.datetime to Unix timestamp."""
    return int(val.timestamp())

sqlite3.register_adapter(datetime.date, adapt_date_iso)
sqlite3.register_adapter(datetime.datetime, adapt_datetime_iso)
sqlite3.register_adapter(datetime.datetime, adapt_datetime_epoch)

def convert_date(val):
    """Convert ISO 8601 date to datetime.date object."""
    return datetime.date.fromisoformat(val.decode())

def convert_datetime(val):
    """Convert ISO 8601 datetime to datetime.datetime object."""
    return datetime.datetime.fromisoformat(val.decode())

def convert_timestamp(val):
    """Convert Unix epoch timestamp to datetime.datetime object."""
    return datetime.datetime.fromtimestamp(int(val))

sqlite3.register_converter("date", convert_date)
sqlite3.register_converter("datetime", convert_datetime)
sqlite3.register_converter("timestamp", convert_timestamp)
```

How to use connection shortcut methods

Using the `execute()`, `executemany()`, and `executescript()` methods of the `Connection` class, your code can be written more concisely because you don’t have to create the (often superfluous) `Cursor` objects explicitly. Instead, the `Cursor` objects are created implicitly and these shortcut methods return the cursor objects. This way, you can execute a `SELECT` statement and iterate over it directly using only a single call on the `Connection` object.

```
# Create and fill the table.
con = sqlite3.connect(":memory:")
con.execute("CREATE TABLE lang(name, first_appeared)")
data = [
    ("C++", 1985),
    ("Objective-C", 1984),
]
con.executemany("INSERT INTO lang(name, first_appeared) VALUES(?, ?)",
data)

# Print the table contents
for row in con.execute("SELECT name, first_appeared FROM lang"):
    print(row)

print("I just deleted", con.execute("DELETE FROM lang").rowcount,
"rows")

# close() is not a shortcut method and it's not called automatically;
# the connection object should be closed manually
con.close()
```

How to use the connection context manager

A `Connection` object can be used as a context manager that automatically commits or rolls back open transactions when leaving the body of the context manager. If the body of the `with` statement finishes without exceptions, the transaction is committed. If this commit fails, or if the body of the `with` statement raises an uncaught exception, the transaction is rolled back. If `autocommit` is `False`, a new transaction is implicitly opened after committing or rolling back.

If there is no open transaction upon leaving the body of the `with` statement, or if `autocommit` is `True`, the context manager does nothing.

Note

The context manager neither implicitly opens a new transaction nor closes the connection. If you need a closing context manager, consider using `contextlib.closing()`.

```
con = sqlite3.connect(":memory:")
con.execute("CREATE TABLE lang(id INTEGER PRIMARY KEY, name VARCHAR
UNIQUE)")

# Successful, con.commit() is called automatically afterwards
with con:
    con.execute("INSERT INTO lang(name) VALUES(?)", ("Python",))

# con.rollback() is called after the with block finishes with an
exception,
# the exception is still raised and must be caught
try:
    with con:
        con.execute("INSERT INTO lang(name) VALUES(?)", ("Python",))
except sqlite3.IntegrityError:
    print("couldn't add Python twice")

# Connection object used as context manager only commits or rollbacks
transactions,
# so the connection object should be closed manually
con.close()
```

How to work with SQLite URIs

Some useful URI tricks include:

- Open a database in read-only mode:

```
>>> con = sqlite3.connect("file:tutorial.db?mode=ro", uri=True)
>>> con.execute("CREATE TABLE readonly(data)")
Traceback (most recent call last):
OperationalError: attempt to write a readonly database
>>> con.close()
```

- Do not implicitly create a new database file if it does not already exist; will raise `OperationalError` if unable to create a new file:

```
>>> con = sqlite3.connect("file:nosuchdb.db?mode=rw", uri=True)
Traceback (most recent call last):
OperationalError: unable to open database file
```

- Create a shared named in-memory database:

```
db = "file:mem1?mode=memory&cache=shared"
con1 = sqlite3.connect(db, uri=True)
con2 = sqlite3.connect(db, uri=True)
with con1:
    con1.execute("CREATE TABLE shared(data)")
    con1.execute("INSERT INTO shared VALUES(28)")
```

```
res = con2.execute("SELECT data FROM shared")
assert res.fetchone() == (28,)

con1.close()
con2.close()
```

More information about this feature, including a list of parameters, can be found in the SQLite URI documentation [https://www.sqlite.org/uri.html].

How to create and use row factories

By default, `sqlite3` represents each row as a `tuple`. If a `tuple` does not suit your needs, you can use the `sqlite3.Row` class or a custom `row_factory`.

While `row_factory` exists as an attribute both on the `Cursor` and the `Connection`, it is recommended to set `Connection.row_factory`, so all cursors created from the connection will use the same row factory.

`Row` provides indexed and case-insensitive named access to columns, with minimal memory overhead and performance impact over a `tuple`. To use `Row` as a row factory, assign it to the `row_factory` attribute:

```
>>> con = sqlite3.connect(":memory:")
>>> con.row_factory = sqlite3.Row
```

Queries now return `Row` objects:

```
>>> res = con.execute("SELECT 'Earth' AS name, 6378 AS radius")
>>> row = res.fetchone()
>>> row.keys()
['name', 'radius']
>>> row[0]         # Access by index.
'Earth'
>>> row["name"]    # Access by name.
'Earth'
>>> row["RADIUS"]  # Column names are case-insensitive.
6378
>>> con.close()
```

Note

The `FROM` clause can be omitted in the `SELECT` statement, as in the above example. In such cases, SQLite returns a single row with columns defined by expressions, e.g. literals, with the given aliases `expr AS alias`.

You can create a custom **`row_factory`** that returns each row as a **`dict`**, with column names mapped to values:

```
def dict_factory(cursor, row):
    fields = [column[0] for column in cursor.description]
    return {key: value for key, value in zip(fields, row)}
```

Using it, queries now return a **`dict`** instead of a **`tuple`**:

```
>>> con = sqlite3.connect(":memory:")
>>> con.row_factory = dict_factory
>>> for row in con.execute("SELECT 1 AS a, 2 AS b"):
...     print(row)
{'a': 1, 'b': 2}
>>> con.close()
```

The following row factory returns a named tuple:

```
from collections import namedtuple

def namedtuple_factory(cursor, row):
    fields = [column[0] for column in cursor.description]
    cls = namedtuple("Row", fields)
    return cls._make(row)
```

`namedtuple_factory()` can be used as follows:

```
>>> con = sqlite3.connect(":memory:")
>>> con.row_factory = namedtuple_factory
>>> cur = con.execute("SELECT 1 AS a, 2 AS b")
>>> row = cur.fetchone()
>>> row
Row(a=1, b=2)
>>> row[0]  # Indexed access.
1
>>> row.b   # Attribute access.
2
>>> con.close()
```

With some adjustments, the above recipe can be adapted to use a `dataclass`, or any other custom class, instead of a `namedtuple`.

How to handle non-UTF-8 text encodings

By default, **`sqlite3`** uses `str` to adapt SQLite values with the TEXT data type. This works well for UTF-8 encoded text, but it might fail for other encodings and invalid UTF-8. You can use a custom `text_factory` to handle such cases.

Because of SQLite's flexible typing [https://www.sqlite.org/flextypegood.html], it is not uncommon to encounter table columns with the TEXT data type containing non-UTF-8 encodings, or even arbitrary data. To demonstrate, let's assume we have a database with ISO-8859-2 (Latin-2) encoded text, for example a table of Czech-English dictionary entries. Assuming we now have a `Connection` instance **`con`** connected to this database, we can decode the Latin-2 encoded text using this `text_factory`:

```
con.text_factory = lambda data: str(data, encoding="latin2")
```

For invalid UTF-8 or arbitrary data in stored in TEXT table columns, you can use the following technique, borrowed from the Unicode HOWTO:

```
con.text_factory = lambda data: str(data, errors="surrogateescape")
```

Note

The **`sqlite3`** module API does not support strings containing surrogates.

See also

Unicode HOWTO

Explanation

Transaction control

`sqlite3` offers multiple methods of controlling whether, when and how database transactions are opened and closed. Transaction control via the autocommit attribute is recommended, while Transaction control via the isolation_level attribute retains the pre-Python 3.12 behaviour.

Transaction control via the `autocommit` attribute

The recommended way of controlling transaction behaviour is through the **`Connection.autocommit`** attribute, which should preferably be set using the *autocommit* parameter of **`connect()`**.

It is suggested to set *autocommit* to `False`, which implies **PEP 249** [https://peps.python.org/pep-0249/]-compliant transaction control. This means:

- **`sqlite3`** ensures that a transaction is always open, so **`connect()`**, **`Connection.commit()`**, and **`Connection.rollback()`** will implicitly open a new transaction (immediately after closing the pending one, for the latter two). **`sqlite3`** uses `BEGIN DEFERRED` statements when opening transactions.
- Transactions should be committed explicitly using **`commit()`**.
- Transactions should be rolled back explicitly using **`rollback()`**.
- An implicit rollback is performed if the database is **`close()`**-ed with pending changes.

Set *autocommit* to `True` to enable SQLite's autocommit mode [https://www.sqlite.org/lang_transaction.html#implicit_versus_explicit_transactions]. In this mode, **`Connection.commit()`** and **`Connection.rollback()`** have no effect. Note that SQLite's autocommit mode is distinct from the **PEP 249** [https://peps.python.org/pep-0249/]-compliant **`Connection.autocommit`** attribute; use **`Connection.in_transaction`** to query the low-level SQLite autocommit mode.

Set *autocommit* to **`LEGACY_TRANSACTION_CONTROL`** to leave transaction control behaviour to the **`Connection.isolation_level`** attribute. See Transaction control via the isolation_level attribute for more information.

Transaction control via the `isolation_level` attribute

Note

The recommended way of controlling transactions is via the `autocommit` attribute. See Transaction control via the autocommit attribute.

If `Connection.autocommit` is set to `LEGACY_TRANSACTION_CONTROL` (the default), transaction behaviour is controlled using the `Connection.isolation_level` attribute. Otherwise, **`isolation_level`** has no effect.

If the connection attribute `isolation_level` is not `None`, new transactions are implicitly opened before `execute()` and `executemany()` executes `INSERT`, `UPDATE`, `DELETE`, or `REPLACE` statements; for other statements, no implicit transaction handling is performed. Use the `commit()` and `rollback()` methods to respectively commit and roll back pending transactions. You can choose the underlying SQLite transaction behaviour [https://www.sqlite.org/lang_transaction.html#deferred_immediate_and_exclusive_transactions] — that is, whether and what type of `BEGIN` statements **`sqlite3`** implicitly executes – via the `isolation_level` attribute.

If `isolation_level` is set to `None`, no transactions are implicitly opened at all. This leaves the underlying SQLite library in autocommit mode [https://www.sqlite.org/lang_transaction.html#implicit_versus_explicit_transactions], but also allows the user to perform their own transaction handling using explicit SQL statements. The underlying SQLite library autocommit mode can be queried using the `in_transaction` attribute.

The `executescript()` method implicitly commits any pending transaction before execution of the given SQL script, regardless of the value of `isolation_level`.

Changed in version 3.6: **`sqlite3`** used to implicitly commit an open transaction before DDL statements. This is no longer the case.

Changed in version 3.12: The recommended way of controlling transactions is now via the `autocommit` attribute.

Data Compression and Archiving

The modules described in this chapter support data compression with the zlib, gzip, bzip2, lzma, and zstd algorithms, and the creation of ZIP- and tar-format archives. See also Archiving operations provided by the `shutil` module.

- The `compression` package
- `compression.zstd` — Compression compatible with the Zstandard format
    - Exceptions
    - Reading and writing compressed files
    - Compressing and decompressing data in memory
    - Zstandard dictionaries
    - Advanced parameter control
    - Miscellaneous
    - Examples
- `zlib` — Compression compatible with **gzip**
- `gzip` — Support for **gzip** files
    - Examples of usage
    - Command-line interface
        - Command-line options
- `bz2` — Support for **bzip2** compression
    - (De)compression of files
    - Incremental (de)compression
    - One-shot (de)compression
    - Examples of usage
- `lzma` — Compression using the LZMA algorithm
    - Reading and writing compressed files
    - Compressing and decompressing data in memory

  - Miscellaneous
  - Specifying custom filter chains
  - Examples
- `zipfile` — Work with ZIP archives
  - ZipFile objects
  - Path objects
  - PyZipFile objects
  - ZipInfo objects
  - Command-line interface
    - Command-line options
  - Decompression pitfalls
    - From file itself
    - File system limitations
    - Resources limitations
    - Interruption
    - Default behaviors of extraction
- `tarfile` — Read and write tar archive files
  - TarFile Objects
  - TarInfo Objects
  - Extraction filters
    - Default named filters
    - Filter errors
    - Hints for further verification
    - Supporting older Python versions
    - Stateful extraction filter example
  - Command-Line Interface
    - Command-line options
  - Examples
    - Reading examples
    - Writing examples
  - Supported tar formats
  - Unicode issues

The `compression` package

Added in version 3.14.

The `compression` package contains the canonical compression modules containing interfaces to several different compression algorithms. Some of these modules have historically been available as separate modules; those will continue to be available under their original names for compatibility reasons, and will not be removed without a deprecation cycle. The use of modules in `compression` is encouraged where practical.

- `compression.bz2` – Re-exports `bz2`
- `compression.gzip` – Re-exports `gzip`
- `compression.lzma` – Re-exports `lzma`
- `compression.zlib` – Re-exports `zlib`
- `compression.zstd` – Wrapper for the Zstandard compression library

`compression.zstd` — Compression compatible with the Zstandard format

Added in version 3.14.

Source code: Lib/compression/zstd/__init__.py [https://github.com/python/cpython/tree/3.14/Lib/compression/zstd/__init__.py]

This module provides classes and functions for compressing and decompressing data using the Zstandard (or *zstd*) compression algorithm. The zstd manual [https://facebook.github.io/zstd/doc/api_manual_latest.html] describes Zstandard as “a fast lossless compression algorithm, targeting real-time compression scenarios at zlib-level and better compression ratios.” Also included is a file interface that supports reading and writing the contents of `.zst` files created by the **zstd** utility, as well as raw zstd compressed streams.

The **`compression.zstd`** module contains:

- The **`open()`** function and **`ZstdFile`** class for reading and writing compressed files.
- The **`ZstdCompressor`** and **`ZstdDecompressor`** classes for incremental (de)compression.
- The **`compress()`** and **`decompress()`** functions for one-shot (de)compression.
- The **`train_dict()`** and **`finalize_dict()`** functions and the **`ZstdDict`** class to train and manage Zstandard dictionaries.

- The `CompressionParameter`, `DecompressionParameter`, and `Strategy` classes for setting advanced (de)compression parameters.

This is an optional module. If it is missing from your copy of CPython, look for documentation from your distributor (that is, whoever provided Python to you). If you are the distributor, see Requirements for optional modules.

Exceptions

exception compression.zstd.ZstdError

This exception is raised when an error occurs during compression or decompression, or while initializing the (de)compressor state.

Reading and writing compressed files

compression.zstd.open(*file*, */*, *mode='rb'*, ***, *level=None*, *options=None*, *zstd_dict=None*, *encoding=None*, *errors=None*, *newline=None*)

Open a Zstandard-compressed file in binary or text mode, returning a file object.

The *file* argument can be either a file name (given as a `str`, `bytes` or path-like object), in which case the named file is opened, or it can be an existing file object to read from or write to.

The mode argument can be either `'rb'` for reading (default), `'wb'` for overwriting, `'ab'` for appending, or `'xb'` for exclusive creation. These can equivalently be given as `'r'`, `'w'`, `'a'`, and `'x'` respectively. You may also open in text mode with `'rt'`, `'wt'`, `'at'`, and `'xt'` respectively.

When reading, the *options* argument can be a dictionary providing advanced decompression parameters; see `DecompressionParameter` for detailed information about supported parameters. The *zstd_dict* argument is a `ZstdDict` instance to be used during decompression. When reading, if the *level* argument is not None, a **`TypeError`** will be raised.

When writing, the *options* argument can be a dictionary providing advanced compression parameters; see `CompressionParameter` for detailed information about supported parameters. The *level* argument is the compression level to use when writing compressed data. Only one of *level* or *options* may be non-None. The *zstd_dict* argument is a `ZstdDict` instance to be used during compression.

In binary mode, this function is equivalent to the `ZstdFile` constructor: `ZstdFile(file, mode, ...)`. In this case, the *encoding*, *errors*, and *newline* parameters must not be provided.

In text mode, a `ZstdFile` object is created, and wrapped in an `io.TextIOWrapper` instance with the specified encoding, error handling behavior, and line endings.

class compression.zstd.ZstdFile(*file*, */*, *mode='rb'*, ***, *level=None*, *options=None*, *zstd_dict=None*)

Open a Zstandard-compressed file in binary mode.

A `ZstdFile` can wrap an already-open file object, or operate directly on a named file. The *file* argument specifies either the file object to wrap, or the name of the file to open (as a `str`, `bytes` or path-like object). If wrapping an existing file object, the wrapped file will not be closed when the `ZstdFile` is closed.

The *mode* argument can be either `'rb'` for reading (default), `'wb'` for overwriting, `'xb'` for exclusive creation, or `'ab'` for appending. These can equivalently be given as `'r'`, `'w'`, `'x'` and `'a'` respectively.

If *file* is a file object (rather than an actual file name), a mode of `'w'` does not truncate the file, and is instead equivalent to `'a'`.

When reading, the *options* argument can be a dictionary providing advanced decompression parameters; see `DecompressionParameter` for detailed information about supported parameters. The *zstd_dict* argument is a `ZstdDict` instance to be used during decompression. When reading, if the *level* argument is not None, a **`TypeError`** will be raised.

When writing, the *options* argument can be a dictionary providing advanced compression parameters; see `CompressionParameter` for detailed information about supported parameters. The *level* argument is the compression level to use when writing compressed data. Only one of *level* or *options* may be passed. The *zstd_dict* argument is a `ZstdDict` instance to be used during compression.

`ZstdFile` supports all the members specified by `io.BufferedIOBase`, except for `detach()` and `truncate()`. Iteration and the `with` statement are supported.

The following method and attributes are also provided:

peek(*size=-1*)

Return buffered data without advancing the file position. At least one byte of data will be returned, unless EOF has been reached. The exact number of bytes returned is unspecified (the *size* argument is ignored).

Note

While calling `peek()` does not change the file position of the `ZstdFile`, it may change the position of the underlying file object (for example, if the `ZstdFile` was constructed by passing a file object for *file*).

mode

'rb' for reading and 'wb' for writing.

name

The name of the Zstandard file. Equivalent to the `name` attribute of the underlying file object.

Compressing and decompressing data in memory

compression.zstd.compress(*data*, *level=None*, *options=None*, *zstd_dict=None*)

Compress *data* (a bytes-like object), returning the compressed data as a `bytes` object.

The *level* argument is an integer controlling the level of compression. *level* is an alternative to setting `CompressionParameter.compression_level` in *options*. Use `bounds()` on `compression_level` to get the values that can be passed for *level*. If advanced compression options are needed, the *level* argument must be omitted and in the *options* dictionary the `CompressionParameter.compression_level` parameter should be set.

The *options* argument is a Python dictionary containing advanced compression parameters. The valid keys and values for compression parameters are documented as part of the `CompressionParameter` documentation.

The *zstd_dict* argument is an instance of `ZstdDict` containing trained data to improve compression efficiency. The function `train_dict()` can be used to generate a Zstandard dictionary.

compression.zstd.decompress(*data*, *zstd_dict=None*, *options=None*)

Decompress *data* (a bytes-like object), returning the uncompressed data as a `bytes` object.

The *options* argument is a Python dictionary containing advanced decompression parameters. The valid keys and values for compression parameters are documented as part of the `DecompressionParameter` documentation.

The *zstd_dict* argument is an instance of `ZstdDict` containing trained data used during compression. This must be the same Zstandard dictionary used during compression.

If *data* is the concatenation of multiple distinct compressed frames, decompress all of these frames, and return the concatenation of the results.

class compression.zstd.ZstdCompressor(*level=None*, *options=None*, *zstd_dict=None*)

Create a compressor object, which can be used to compress data incrementally.

For a more convenient way of compressing a single chunk of data, see the module-level function `compress()`.

The *level* argument is an integer controlling the level of compression. *level* is an alternative to setting `CompressionParameter.compression_level` in *options*. Use `bounds()` on `compression_level` to get the values that can be passed for *level*. If advanced compression options are needed, the *level* argument must be omitted and in the *options* dictionary the `CompressionParameter.compression_level` parameter should be set.

The *options* argument is a Python dictionary containing advanced compression parameters. The valid keys and values for compression parameters are documented as part of the `CompressionParameter` documentation.

The *zstd_dict* argument is an optional instance of `ZstdDict` containing trained data to improve compression efficiency. The function `train_dict()` can be used to generate a Zstandard dictionary.

compress(*data*, *mode=ZstdCompressor.CONTINUE*)

> Compress *data* (a bytes-like object), returning a `bytes` object with compressed data if possible, or otherwise an empty `bytes` object. Some of *data* may be buffered internally, for use in later calls to `compress()` and `flush()`. The returned data should be concatenated with the output of any previous calls to `compress()`.
>
> The *mode* argument is a `ZstdCompressor` attribute, either `CONTINUE`, `FLUSH_BLOCK`, or `FLUSH_FRAME`.
>
> When all data has been provided to the compressor, call the `flush()` method to finish the compression process. If `compress()` is called with *mode* set to `FLUSH_FRAME`, `flush()` should not be called, as it would write out a new empty frame.

flush(*mode=ZstdCompressor.FLUSH_FRAME*)

Finish the compression process, returning a `bytes` object containing any data stored in the compressor's internal buffers.

The *mode* argument is a `ZstdCompressor` attribute, either `FLUSH_BLOCK`, or `FLUSH_FRAME`.

set_pledged_input_size(*size*)

Specify the amount of uncompressed data *size* that will be provided for the next frame. *size* will be written into the frame header of the next frame unless `CompressionParameter.content_size_flag` is `False` or `0`. A size of `0` means that the frame is empty. If *size* is `None`, the frame header will omit the frame size. Frames that include the uncompressed data size require less memory to decompress, especially at higher compression levels.

If `last_mode` is not `FLUSH_FRAME`, a `ValueError` is raised as the compressor is not at the start of a frame. If the pledged size does not match the actual size of data provided to `compress()`, future calls to **`compress()`** or `flush()` may raise `ZstdError` and the last chunk of data may be lost.

After `flush()` or `compress()` are called with mode `FLUSH_FRAME`, the next frame will not include the frame size into the header unless **`set_pledged_input_size()`** is called again.

CONTINUE

Collect more data for compression, which may or may not generate output immediately. This mode optimizes the compression ratio by maximizing the amount of data per block and frame.

FLUSH_BLOCK

Complete and write a block to the data stream. The data returned so far can be immediately decompressed. Past data can still be referenced in future blocks generated by calls to `compress()`, improving compression.

FLUSH_FRAME

Complete and write out a frame. Future data provided to `compress()` will be written into a new frame and *cannot* reference past data.

last_mode

The last mode passed to either `compress()` or `flush()`. The value can be one of `CONTINUE`, `FLUSH_BLOCK`, or `FLUSH_FRAME`. The initial value is `FLUSH_FRAME`, signifying that the compressor is at the start of a new frame.

class compression.zstd.ZstdDecompressor(*zstd_dict=None*, *options=None*)

Create a decompressor object, which can be used to decompress data incrementally.

For a more convenient way of decompressing an entire compressed stream at once, see the module-level function `decompress()`.

The *options* argument is a Python dictionary containing advanced decompression parameters. The valid keys and values for compression parameters are documented as part of the `DecompressionParameter` documentation.

The *zstd_dict* argument is an instance of `ZstdDict` containing trained data used during compression. This must be the same Zstandard dictionary used during compression.

Note

This class does not transparently handle inputs containing multiple compressed frames, unlike the `decompress()` function and `ZstdFile` class. To decompress a multi-frame input, you should use `decompress()`, `ZstdFile` if working with a file object, or multiple **`ZstdDecompressor`** instances.

decompress(*data*, *max_length=-1*)

Decompress *data* (a bytes-like object), returning uncompressed data as bytes. Some of *data* may be buffered internally, for use in later calls to **`decompress()`**. The returned data should be concatenated with the output of any previous calls to **`decompress()`**.

If *max_length* is non-negative, the method returns at most *max_length* bytes of decompressed data. If this limit is reached and further output can be produced, the `needs_input` attribute will be set to `False`. In this case, the next call to `decompress()` may provide *data* as `b''` to obtain more of the output.

If all of the input data was decompressed and returned (either because this was less than *max_length* bytes, or because *max_length* was negative), the `needs_input` attribute will be set to `True`.

Attempting to decompress data after the end of a frame will raise a `ZstdError`. Any data found after the end of the frame is ignored and saved in the `unused_data` attribute.

eof

`True` if the end-of-stream marker has been reached.

unused_data

Data found after the end of the compressed stream.

Before the end of the stream is reached, this will be `b''`.

needs_input

`False` if the **`decompress()`** method can provide more decompressed data before requiring new compressed input.

Zstandard dictionaries

compression.zstd.train_dict(*samples*, *dict_size*)

Train a Zstandard dictionary, returning a **`ZstdDict`** instance. Zstandard dictionaries enable more efficient compression of smaller sizes of data, which is traditionally difficult to compress due to less repetition. If you are compressing multiple similar groups of data (such as similar files), Zstandard dictionaries can improve compression ratios and speed significantly.

The *samples* argument (an iterable of **`bytes`** objects), is the population of samples used to train the Zstandard dictionary.

The *dict_size* argument, an integer, is the maximum size (in bytes) the Zstandard dictionary should be. The Zstandard documentation suggests an absolute maximum of no more than 100 KB, but the maximum can often be smaller depending on the data. Larger dictionaries generally slow down compression, but improve compression ratios. Smaller dictionaries lead to faster compression, but reduce the compression ratio.

compression.zstd.finalize_dict(*zstd_dict*, */*, *samples*, *dict_size*, *level*)

An advanced function for converting a “raw content” Zstandard dictionary into a regular Zstandard dictionary. “Raw content” dictionaries are a sequence of bytes that do not need to follow the structure of a normal Zstandard dictionary.

The *zstd_dict* argument is a `ZstdDict` instance with the `dict_content` containing the raw dictionary contents.

The *samples* argument (an iterable of `bytes` objects), contains sample data for generating the Zstandard dictionary.

The *dict_size* argument, an integer, is the maximum size (in bytes) the Zstandard dictionary should be. See `train_dict()` for suggestions on the maximum dictionary size.

The *level* argument (an integer) is the compression level expected to be passed to the compressors using this dictionary. The dictionary information varies for each compression level, so tuning for the proper compression level can make compression more efficient.

class compression.zstd.ZstdDict(*dict_content*, */*, ***, *is_raw=False*)

A wrapper around Zstandard dictionaries. Dictionaries can be used to improve the compression of many small chunks of data. Use `train_dict()` if you need to train a new dictionary from sample data.

The *dict_content* argument (a bytes-like object), is the already trained dictionary information.

The *is_raw* argument, a boolean, is an advanced parameter controlling the meaning of *dict_content*. `True` means *dict_content* is a “raw content” dictionary, without any format restrictions. `False` means *dict_content* is an ordinary Zstandard dictionary, created from Zstandard functions, for example, **`train_dict()`** or the external **zstd** CLI.

When passing a **`ZstdDict`** to a function, the **`as_digested_dict`** and **`as_undigested_dict`** attributes can control how the dictionary is loaded by passing them as the `zstd_dict` argument, for example, `compress(data, zstd_dict=zd.as_digested_dict)`. Digesting a dictionary is a costly operation that occurs when loading a Zstandard dictionary. When making multiple calls to compression or decompression, passing a digested dictionary will reduce the overhead of loading the dictionary.

Difference for compression

	Digested dictionary	**Undigested dictionary**
Advanced parameters of the compressor which may be overridden by the dictionary’s parameters	`window_log`, `hash_log`, `chain_log`, `search_log`, `min_match`, `target_length`, `strategy`, `enable_long_distance_matching`, `ldm_hash_log`, `ldm_min_match`, `ldm_bucket_size_log`, `ldm_hash_rate_log`, and some non-public parameters.	None
`ZstdDict` internally	Yes. It’s faster when loading a digested dictionary again	No. If you wish to load an

	Digested dictionary	**Undigested dictionary**
caches the dictionary	with the same compression level.	undigested dictionary multiple times, consider reusing a compressor object.

If passing a **`ZstdDict`** without any attribute, an undigested dictionary is passed by default when compressing and a digested dictionary is generated if necessary and passed by default when decompressing.

dict_content

The content of the Zstandard dictionary, a `bytes` object. It’s the same as the *dict_content* argument in the `__init__` method. It can be used with other programs, such as the `zstd` CLI program.

dict_id

Identifier of the Zstandard dictionary, a non-negative int value.

Non-zero means the dictionary is ordinary, created by Zstandard functions and following the Zstandard format.

`0` means a “raw content” dictionary, free of any format restriction, used for advanced users.

Note

The meaning of `0` for **`ZstdDict.dict_id`** is different from the `dictionary_id` attribute to the **`get_frame_info()`** function.

as_digested_dict

Load as a digested dictionary.

as_undigested_dict

Load as an undigested dictionary.

Advanced parameter control

class compression.zstd.CompressionParameter

An **`IntEnum`** containing the advanced compression parameter keys that can be used when compressing data.

The **`bounds()`** method can be used on any attribute to get the valid values for that parameter.

Parameters are optional; any omitted parameter will have it’s value selected automatically.

Example getting the lower and upper bound of **`compression_level`**:

```
lower, upper =
CompressionParameter.compression_level.bounds()
```

Example setting the **`window_log`** to the maximum size:

```
_lower, upper = CompressionParameter.window_log.bounds()
options = {CompressionParameter.window_log: upper}
compress(b'venezuelan beaver cheese', options=options)
```

bounds()

Return the tuple of int bounds, `(lower, upper)`, of a compression parameter. This method should be called on the attribute you wish to retrieve the bounds of. For example, to get the valid values for `compression_level`, one may check the result of `CompressionParameter.compression_level.bounds()`.

Both the lower and upper bounds are inclusive.

compression_level

A high-level means of setting other compression parameters that affect the speed and ratio of compressing data.

Regular compression levels are greater than `0`. Values greater than `20` are considered “ultra” compression and require more memory than other levels. Negative values can be used to trade off faster compression for worse compression ratios.

Setting the level to zero uses **`COMPRESSION_LEVEL_DEFAULT`**.

window_log

Maximum allowed back-reference distance the compressor can use when compressing data, expressed as power of two, `1 << window_log` bytes. This parameter greatly influences the memory usage of compression. Higher values require more memory but gain better compression values.

A value of zero causes the value to be selected automatically.

hash_log

Size of the initial probe table, as a power of two. The resulting memory usage is `1 << (hash_log+2)` bytes. Larger tables improve compression ratio of strategies

<= `dfast`, and improve compression speed of strategies > `dfast`.

A value of zero causes the value to be selected automatically.

chain_log

Size of the multi-probe search table, as a power of two. The resulting memory usage is `1 << (chain_log+2)` bytes. Larger tables result in better and slower compression. This parameter has no effect for the `fast` strategy. It’s still useful when using `dfast` strategy, in which case it defines a secondary probe table.

A value of zero causes the value to be selected automatically.

search_log

Number of search attempts, as a power of two. More attempts result in better and slower compression. This parameter is useless for `fast` and `dfast` strategies.

A value of zero causes the value to be selected automatically.

min_match

Minimum size of searched matches. Larger values increase compression and decompression speed, but decrease ratio. Note that Zstandard can still find matches of smaller size, it just tweaks its search algorithm to look for this size and larger. For all strategies < `btopt`, the effective minimum is `4`; for all strategies > `fast`, the effective maximum is `6`.

A value of zero causes the value to be selected automatically.

target_length

The impact of this field depends on the selected `Strategy`.

For strategies `btopt`, `btultra` and `btultra2`, the value is the length of a match considered “good enough” to stop searching. Larger values make compression ratios better, but compresses slower.

For strategy `fast`, it is the distance between match sampling. Larger values make compression faster, but with a worse compression ratio.

A value of zero causes the value to be selected automatically.

strategy

The higher the value of selected strategy, the more complex the compression technique used by zstd, resulting in higher compression ratios but slower compression.

See also

`Strategy`

enable_long_distance_matching

Long distance matching can be used to improve compression for large inputs by finding large matches at greater distances. It increases memory usage and window size.

`True` or `1` enable long distance matching while `False` or `0` disable it.

Enabling this parameter increases default `window_log` to 128 MiB except when expressly set to a different value. This setting is enabled by default if **`window_log`** >= 128 MiB and the compression strategy >= `btopt` (compression level 16+).

ldm_hash_log

Size of the table for long distance matching, as a power of two. Larger values increase memory usage and compression ratio, but decrease compression speed.

A value of zero causes the value to be selected automatically.

ldm_min_match

Minimum match size for long distance matcher. Larger or too small values can often decrease the compression ratio.

A value of zero causes the value to be selected automatically.

ldm_bucket_size_log

Log size of each bucket in the long distance matcher hash table for collision resolution. Larger values improve collision resolution but decrease compression speed.

A value of zero causes the value to be selected automatically.

ldm_hash_rate_log

Frequency of inserting/looking up entries into the long distance matcher hash table. Larger values improve compression speed. Deviating far from the default value will likely result in a compression ratio decrease.

A value of zero causes the value to be selected automatically.

content_size_flag

Write the size of the data to be compressed into the Zstandard frame header when known prior to compressing.

This flag only takes effect under the following scenarios:

- Calling `compress()` for one-shot compression
- Providing all of the data to be compressed in the frame in a single `ZstdCompressor.compress()` call, with the `ZstdCompressor.FLUSH_FRAME` mode.
- Calling `ZstdCompressor.set_pledged_input_size()` with the exact amount of data that will be provided to the compressor prior to any calls to `ZstdCompressor.compress()` for the current frame. **`ZstdCompressor.set_pledged_input_size()`** must be called for each new frame.

All other compression calls may not write the size information into the frame header.

`True` or `1` enable the content size flag while `False` or `0` disable it.

checksum_flag

A four-byte checksum using XXHash64 of the uncompressed content is written at the end of each frame. Zstandard's decompression code verifies the checksum. If there is a mismatch a `ZstdError` exception is raised.

`True` or `1` enable checksum generation while `False` or `0` disable it.

dict_id_flag

When compressing with a `ZstdDict`, the dictionary's ID is written into the frame header.

`True` or `1` enable storing the dictionary ID while `False` or `0` disable it.

nb_workers

Select how many threads will be spawned to compress in parallel. When **`nb_workers`** > 0, enables multi-threaded compression, a value of `1` means "one-thread multi-threaded mode". More workers improve speed, but also increase memory usage and slightly reduce compression ratio.

A value of zero disables multi-threading.

job_size

Size of a compression job, in bytes. This value is enforced only when `nb_workers` >= 1. Each compression job is completed in parallel, so this value can indirectly impact the number of active threads.

A value of zero causes the value to be selected automatically.

overlap_log

Sets how much data is reloaded from previous jobs (threads) for new jobs to be used by the look behind window during compression. This value is only used when `nb_workers` >= 1. Acceptable values vary from 0 to 9.

- 0 means dynamically set the overlap amount
- 1 means no overlap

- 9 means use a full window size from the previous job

Each increment halves/doubles the overlap size. “8” means an overlap of `window_size/2`, “7” means an overlap of `window_size/4`, etc.

class compression.zstd.DecompressionParameter

An `IntEnum` containing the advanced decompression parameter keys that can be used when decompressing data. Parameters are optional; any omitted parameter will have it’s value selected automatically.

The `bounds()` method can be used on any attribute to get the valid values for that parameter.

Example setting the `window_log_max` to the maximum size:

```
data = compress(b'Some very long buffer of bytes...')

_lower, upper =
DecompressionParameter.window_log_max.bounds()

options = {DecompressionParameter.window_log_max: upper}
decompress(data, options=options)
```

bounds()

Return the tuple of int bounds, `(lower, upper)`, of a decompression parameter. This method should be called on the attribute you wish to retrieve the bounds of.

Both the lower and upper bounds are inclusive.

window_log_max

The base-two logarithm of the maximum size of the window used during decompression. This can be useful to limit the amount of memory used when decompressing data. A larger maximum window size leads to faster decompression.

A value of zero causes the value to be selected automatically.

class compression.zstd.Strategy

An **`IntEnum`** containing strategies for compression. Higher-numbered strategies correspond to more complex and slower compression.

Note

The values of attributes of **`Strategy`** are not necessarily stable across zstd versions. Only the ordering of the attributes may be relied upon. The attributes are listed below in order.

The following strategies are available:

fast

dfast

greedy

lazy

lazy2

btlazy2

btopt

btultra

btultra2

Miscellaneous

compression.zstd.get_frame_info(*frame_buffer*)

Retrieve a `FrameInfo` object containing metadata about a Zstandard frame. Frames contain metadata related to the compressed data they hold.

class compression.zstd.FrameInfo

Metadata related to a Zstandard frame.

decompressed_size

The size of the decompressed contents of the frame.

dictionary_id

An integer representing the Zstandard dictionary ID needed for decompressing the frame. `0` means the dictionary ID was not recorded in the frame header. This may mean that a Zstandard dictionary is not needed, or that the ID of a required dictionary was not recorded.

compression.zstd.COMPRESSION_LEVEL_DEFAULT

The default compression level for Zstandard: `3`.

compression.zstd.zstd_version_info

Version number of the runtime zstd library as a tuple of integers (major, minor, release).

Examples

Reading in a compressed file:

```
from compression import zstd

with zstd.open("file.zst") as f:
    file_content = f.read()
```

Creating a compressed file:

```
from compression import zstd

data = b"Insert Data Here"
with zstd.open("file.zst", "w") as f:
    f.write(data)
```

Compressing data in memory:

```
from compression import zstd

data_in = b"Insert Data Here"
data_out = zstd.compress(data_in)
```

Incremental compression:

```
from compression import zstd

comp = zstd.ZstdCompressor()
out1 = comp.compress(b"Some data\n")
out2 = comp.compress(b"Another piece of data\n")
out3 = comp.compress(b"Even more data\n")
out4 = comp.flush()
# Concatenate all the partial results:
result = b"".join([out1, out2, out3, out4])
```

Writing compressed data to an already-open file:

```
from compression import zstd

with open("myfile", "wb") as f:
    f.write(b"This data will not be compressed\n")
    with zstd.open(f, "w") as zstf:
        zstf.write(b"This *will* be compressed\n")
    f.write(b"Not compressed\n")
```

Creating a compressed file using compression parameters:

```
from compression import zstd

options = {
   zstd.CompressionParameter.checksum_flag: 1
}
```

```
with zstd.open("file.zst", "w", options=options) as f:
    f.write(b"Mind if I squeeze in?")
```

`zlib` — Compression compatible with gzip

For applications that require data compression, the functions in this module allow compression and decompression, using the zlib library [https://www.zlib.net].

This is an optional module. If it is missing from your copy of CPython, look for documentation from your distributor (that is, whoever provided Python to you). If you are the distributor, see Requirements for optional modules.

zlib’s functions have many options and often need to be used in a particular order. This documentation doesn’t attempt to cover all of the permutations; consult the zlib manual [https://www.zlib.net/manual.html] for authoritative information.

For reading and writing `.gz` files see the **`gzip`** module.

The available exception and functions in this module are:

exception zlib.error

Exception raised on compression and decompression errors.

zlib.adler32(*data*[, *value*])

Computes an Adler-32 checksum of *data*. (An Adler-32 checksum is almost as reliable as a CRC32 but can be computed much more quickly.) The result is an unsigned 32-bit integer. If *value* is present, it is used as the starting value of the checksum; otherwise, a default value of 1 is used. Passing in *value* allows computing a running checksum over the concatenation of several inputs. The algorithm is not cryptographically strong, and should not be used for authentication or digital signatures. Since the algorithm is designed for use as a checksum algorithm, it is not suitable for use as a general hash algorithm.

Changed in version 3.0: The result is always unsigned.

zlib.compress(*data*, */*, *level=Z_DEFAULT_COMPRESSION*, *wbits=MAX_WBITS*)

Compresses the bytes in *data*, returning a bytes object containing compressed data. *level* is an integer from `0` to `9` or `-1` controlling the level of compression; See **`Z_BEST_SPEED`** (`1`), **`Z_BEST_COMPRESSION`** (`9`), **`Z_NO_COMPRESSION`** (`0`), and the default, **`Z_DEFAULT_COMPRESSION`** (`-1`) for more information about these values.

The *wbits* argument controls the size of the history buffer (or the “window size”) used when compressing data, and whether a header and trailer is included in the output. It can take several ranges of values, defaulting to `15` (**`MAX_WBITS`**):

- +9 to +15: The base-two logarithm of the window size, which therefore ranges between 512 and 32768. Larger values produce better compression at the expense of greater memory usage. The resulting output will include a zlib-specific header and trailer.
- −9 to −15: Uses the absolute value of *wbits* as the window size logarithm, while producing a raw output stream with no header or trailing checksum.
- +25 to +31 = 16 + (9 to 15): Uses the low 4 bits of the value as the window size logarithm, while

including a basic **gzip** header and trailing checksum in the output.

Raises the `error` exception if any error occurs.

Changed in version 3.6: level can now be used as a keyword parameter.

Changed in version 3.11: The *wbits* parameter is now available to set window bits and compression type.

zlib.compressobj(*level=Z_DEFAULT_COMPRESSION*, *method=DEFLATED*, *wbits=MAX_WBITS*, *memLevel=DEF_MEM_LEVEL*, *strategy=Z_DEFAULT_STRATEGY*[, *zdict*])

Returns a compression object, to be used for compressing data streams that won't fit into memory at once.

level is the compression level – an integer from `0` to `9` or `-1`. See `Z_BEST_SPEED` (`1`), `Z_BEST_COMPRESSION` (`9`), `Z_NO_COMPRESSION` (`0`), and the default, `Z_DEFAULT_COMPRESSION` (`-1`) for more information about these values.

method is the compression algorithm. Currently, the only supported value is `DEFLATED`.

The *wbits* parameter controls the size of the history buffer (or the "window size"), and what header and trailer format will be used. It has the same meaning as described for compress().

The *memLevel* argument controls the amount of memory used for the internal compression state. Valid values range from `1` to `9`. Higher values use more memory, but are faster and produce smaller output.

strategy is used to tune the compression algorithm. Possible values are `Z_DEFAULT_STRATEGY`, `Z_FILTERED`, `Z_HUFFMAN_ONLY`, `Z_RLE` and `Z_FIXED`.

zdict is a predefined compression dictionary. This is a sequence of bytes (such as a `bytes` object) containing subsequences that are expected to occur frequently in the data that is to be compressed. Those subsequences that are expected to be most common should come at the end of the dictionary.

Changed in version 3.3: Added the *zdict* parameter and keyword argument support.

zlib.crc32(*data*[, *value*])

Computes a CRC (Cyclic Redundancy Check) checksum of *data*. The result is an unsigned 32-bit integer. If *value* is present, it is used as the starting value of the checksum; otherwise, a default value of 0 is used. Passing in *value* allows computing a running checksum over the concatenation of several inputs. The algorithm is not cryptographically strong, and should not be used for authentication or digital signatures. Since the algorithm is designed for use as a checksum algorithm, it is not suitable for use as a general hash algorithm.

Changed in version 3.0: The result is always unsigned.

zlib.decompress(*data*, */*, *wbits=MAX_WBITS*, *bufsize=DEF_BUF_SIZE*)

Decompresses the bytes in *data*, returning a bytes object containing the uncompressed data. The *wbits* parameter depends on the format of *data*, and is discussed further below. If *bufsize* is given, it is used as the initial size of the output buffer. Raises the `error` exception if any error occurs.

The *wbits* parameter controls the size of the history buffer (or “window size”), and what header and trailer format is expected. It is similar to the parameter for `compressobj()`, but accepts more ranges of values:

- +8 to +15: The base-two logarithm of the window size. The input must include a zlib header and trailer.
- 0: Automatically determine the window size from the zlib header. Only supported since zlib 1.2.3.5.
- −8 to −15: Uses the absolute value of *wbits* as the window size logarithm. The input must be a raw stream with no header or trailer.
- +24 to +31 = 16 + (8 to 15): Uses the low 4 bits of the value as the window size logarithm. The input must include a gzip header and trailer.
- +40 to +47 = 32 + (8 to 15): Uses the low 4 bits of the value as the window size logarithm, and automatically accepts either the zlib or gzip format.

When decompressing a stream, the window size must not be smaller than the size originally used to compress the stream; using a too-small value may result in an `error` exception. The default *wbits* value corresponds to the largest window size and requires a zlib header and trailer to be included.

bufsize is the initial size of the buffer used to hold decompressed data. If more space is required, the buffer size will be increased as needed, so you don’t have to get this value exactly right; tuning it will only save a few calls to `malloc()`.

Changed in version 3.6: wbits and *bufsize* can be used as keyword arguments.

zlib.decompressobj(*wbits=MAX_WBITS*[, *zdict*])

Returns a decompression object, to be used for decompressing data streams that won’t fit into memory at once.

The *wbits* parameter controls the size of the history buffer (or the “window size”), and what header and trailer format is expected. It has the same meaning as described for decompress().

The *zdict* parameter specifies a predefined compression dictionary. If provided, this must be the same dictionary as was used by the compressor that produced the data that is to be decompressed.

Note

If *zdict* is a mutable object (such as a **`bytearray`**), you must not modify its contents between the call to **`decompressobj()`** and the first call to the decompressor’s `decompress()` method.

Changed in version 3.3: Added the *zdict* parameter.

Compression objects support the following methods:

Compress.compress(*data*)

Compress *data*, returning a bytes object containing compressed data for at least part of the data in *data*. This data should be concatenated to the output produced by any preceding calls to the **`compress()`** method. Some input may be kept in internal buffers for later processing.

Compress.flush([*mode*])

All pending input is processed, and a bytes object containing the remaining compressed output is returned. *mode* can be selected from the constants `Z_NO_FLUSH`, `Z_PARTIAL_FLUSH`, `Z_SYNC_FLUSH`, `Z_FULL_FLUSH`, `Z_BLOCK`, or `Z_FINISH`, defaulting to `Z_FINISH`. Except `Z_FINISH`, all constants allow compressing further bytestrings of data, while `Z_FINISH` finishes the compressed stream and prevents compressing any more data. After calling `flush()` with *mode* set to `Z_FINISH`, the `compress()` method cannot be called again; the only realistic action is to delete the object.

Compress.copy()

Returns a copy of the compression object. This can be used to efficiently compress a set of data that share a common initial prefix.

Changed in version 3.8: Added `copy.copy()` and `copy.deepcopy()` support to compression objects.

Decompression objects support the following methods and attributes:

Decompress.unused_data

A bytes object which contains any bytes past the end of the compressed data. That is, this remains `b""` until the last byte that contains compression data is available. If the whole bytestring turned out to contain compressed data, this is `b""`, an empty bytes object.

Decompress.unconsumed_tail

A bytes object that contains any data that was not consumed by the last `decompress()` call because it exceeded the limit for the uncompressed data buffer. This data has not yet been seen by the zlib machinery, so you must feed it (possibly with further data concatenated to it) back to a subsequent `decompress()` method call in order to get correct output.

Decompress.eof

A boolean indicating whether the end of the compressed data stream has been reached.

This makes it possible to distinguish between a properly formed compressed stream, and an incomplete or truncated one.

Added in version 3.3.

Decompress.decompress(*data*, *max_length=0*)

Decompress *data*, returning a bytes object containing the uncompressed data corresponding to at least part of the data in *string*. This data should be concatenated to the output produced by any preceding calls to the `decompress()` method. Some of the input data may be preserved in internal buffers for later processing.

If the optional parameter *max_length* is non-zero then the return value will be no longer than *max_length*. This may mean that not all of the compressed input can be processed; and unconsumed data will be stored in the attribute `unconsumed_tail`. This bytestring must be passed to a subsequent call to `decompress()` if decompression is to continue. If *max_length* is zero then the whole input is decompressed, and `unconsumed_tail` is empty.

Changed in version 3.6: max_length can be used as a keyword argument.

Decompress.flush([*length*])

All pending input is processed, and a bytes object containing the remaining uncompressed output is returned. After calling `flush()`, the `decompress()` method cannot be called again; the only realistic action is to delete the object.

The optional parameter *length* sets the initial size of the output buffer.

Decompress.copy()

Returns a copy of the decompression object. This can be used to save the state of the decompressor midway through the data stream in order to speed up random seeks into the stream at a future point.

Changed in version 3.8: Added `copy.copy()` and `copy.deepcopy()` support to decompression objects.

The following constants are available to configure compression and decompression behavior:

zlib.DEFLATED

The deflate compression method.

zlib.MAX_WBITS

The maximum window size, expressed as a power of 2. For example, if **`MAX_WBITS`** is `15` it results in a window size of `32 KiB`.

zlib.DEF_MEM_LEVEL

The default memory level for compression objects.

zlib.DEF_BUF_SIZE

The default buffer size for decompression operations.

zlib.Z_NO_COMPRESSION

Compression level `0`; no compression.

Added in version 3.6.

zlib.Z_BEST_SPEED

Compression level `1`; fastest and produces the least compression.

zlib.Z_BEST_COMPRESSION

Compression level `9`; slowest and produces the most compression.

zlib.Z_DEFAULT_COMPRESSION

Default compression level (`-1`); a compromise between speed and compression. Currently equivalent to compression level `6`.

zlib.Z_DEFAULT_STRATEGY

Default compression strategy, for normal data.

zlib.Z_FILTERED

Compression strategy for data produced by a filter (or predictor).

zlib.Z_HUFFMAN_ONLY

Compression strategy that forces Huffman coding only.

zlib.Z_RLE

Compression strategy that limits match distances to one (run-length encoding).

This constant is only available if Python was compiled with zlib 1.2.0.1 or greater.

Added in version 3.6.

zlib.Z_FIXED

Compression strategy that prevents the use of dynamic Huffman codes.

This constant is only available if Python was compiled with zlib 1.2.2.2 or greater.

Added in version 3.6.

zlib.Z_NO_FLUSH

Flush mode `0`. No special flushing behavior.

Added in version 3.6.

zlib.Z_PARTIAL_FLUSH

Flush mode `1`. Flush as much output as possible.

zlib.Z_SYNC_FLUSH

Flush mode `2`. All output is flushed and the output is aligned to a byte boundary.

zlib.Z_FULL_FLUSH

Flush mode `3`. All output is flushed and the compression state is reset.

zlib.Z_FINISH

Flush mode `4`. All pending input is processed, no more input is expected.

zlib.Z_BLOCK

Flush mode `5`. A deflate block is completed and emitted.

This constant is only available if Python was compiled with zlib 1.2.2.2 or greater.

Added in version 3.6.

zlib.Z_TREES

Flush mode `6`, for inflate operations. Instructs inflate to return when it gets to the next deflate block boundary.

This constant is only available if Python was compiled with zlib 1.2.3.4 or greater.

Added in version 3.6.

Information about the version of the zlib library in use is available through the following constants:

zlib.ZLIB_VERSION

The version string of the zlib library that was used for building the module. This may be different from the zlib library actually used at runtime, which is available as `ZLIB_RUNTIME_VERSION`.

zlib.ZLIB_RUNTIME_VERSION

The version string of the zlib library actually loaded by the interpreter.

Added in version 3.3.

zlib.ZLIBNG_VERSION

The version string of the zlib-ng library that was used for building the module if zlib-ng was used. When present, the `ZLIB_VERSION` and `ZLIB_RUNTIME_VERSION` constants reflect the version of the zlib API provided by zlib-ng.

If zlib-ng was not used to build the module, this constant will be absent.

Added in version 3.14.

See also

Module `gzip`

Reading and writing **gzip**-format files.

https://www.zlib.net
The zlib library home page.

https://www.zlib.net/manual.html
The zlib manual explains the semantics and usage of the library’s many functions.

In case gzip (de)compression is a bottleneck, the python-isal [https://github.com/pycompression/python-isal] package speeds up (de)compression with a mostly compatible API.

`gzip` — Support for gzip files

Source code: Lib/gzip.py [https://github.com/python/cpython/tree/3.14/Lib/gzip.py]

This module provides a simple interface to compress and decompress files just like the GNU programs **gzip** and **gunzip** would.

This is an optional module. If it is missing from your copy of CPython, look for documentation from your distributor (that is, whoever provided Python to you). If you are the distributor, see Requirements for optional modules.

The data compression is provided by the `zlib` module.

The `gzip` module provides the `GzipFile` class, as well as the `open()`, `compress()` and `decompress()` convenience functions. The `GzipFile` class reads and writes **gzip**-format files, automatically compressing or decompressing the data so that it looks like an ordinary file object.

Note that additional file formats which can be decompressed by the **gzip** and **gunzip** programs, such as those produced by **compress** and **pack**, are not supported by this module.

The module defines the following items:

gzip.open(*filename*, *mode='rb'*, *compresslevel=9*, *encoding=None*, *errors=None*, *newline=None*)

Open a gzip-compressed file in binary or text mode, returning a file object.

The *filename* argument can be an actual filename (a **`str`** or **`bytes`** object), or an existing file object to read from or write to.

The *mode* argument can be any of `'r'`, `'rb'`, `'a'`, `'ab'`, `'w'`, `'wb'`, `'x'` or `'xb'` for binary mode, or `'rt'`, `'at'`, `'wt'`, or `'xt'` for text mode. The default is `'rb'`.

The *compresslevel* argument is an integer from 0 to 9, as for the **`GzipFile`** constructor.

For binary mode, this function is equivalent to the **`GzipFile`** constructor: `GzipFile(filename, mode, compresslevel)`. In this case, the *encoding*, *errors* and *newline* arguments must not be provided.

For text mode, a **`GzipFile`** object is created, and wrapped in an **`io.TextIOWrapper`** instance with the specified encoding, error handling behavior, and line ending(s).

Changed in version 3.3: Added support for *filename* being a file object, support for text mode, and the *encoding*, *errors* and *newline* arguments.

Changed in version 3.4: Added support for the `'x'`, `'xb'` and `'xt'` modes.

Changed in version 3.6: Accepts a path-like object.

exception gzip.BadGzipFile

An exception raised for invalid gzip files. It inherits from **`OSError`**. **`EOFError`** and **`zlib.error`** can also be raised for invalid gzip files.

Added in version 3.8.

class gzip.GzipFile(*filename=None*, *mode=None*, *compresslevel=9*, *fileobj=None*, *mtime=None*)

Constructor for the **`GzipFile`** class, which simulates most of the methods of a file object, with the exception of the **`truncate()`** method. At least one of *fileobj* and *filename* must be given a non-trivial value.

The new class instance is based on *fileobj*, which can be a regular file, an **`io.BytesIO`** object, or any other object which simulates a file. It defaults to `None`, in which case *filename* is opened to provide a file object.

When *fileobj* is not `None`, the *filename* argument is only used to be included in the **gzip** file header, which may include the original filename of the uncompressed file. It defaults to the filename of *fileobj*, if discernible; otherwise, it defaults to the empty string, and in this case the original filename is not included in the header.

The *mode* argument can be any of `'r'`, `'rb'`, `'a'`, `'ab'`, `'w'`, `'wb'`, `'x'`, or `'xb'`, depending on whether the file will be read or written. The default is the mode of *fileobj* if discernible; otherwise, the default is `'rb'`. In future Python releases the mode of *fileobj* will not be used. It is better to always specify *mode* for writing.

Note that the file is always opened in binary mode. To open a compressed file in text mode, use **`open()`** (or wrap your **`GzipFile`** with an **`io.TextIOWrapper`**).

The *compresslevel* argument is an integer from `0` to `9` controlling the level of compression; `1` is fastest and produces the least compression, and `9` is slowest and produces the most compression. `0` is no compression. The default is `9`.

The optional *mtime* argument is the timestamp requested by gzip. The time is in Unix format, i.e., seconds since 00:00:00 UTC, January 1, 1970. If *mtime* is omitted or `None`, the current time is used. Use *mtime* = 0 to generate a compressed stream that does not depend on creation time.

See below for the `mtime` attribute that is set when decompressing.

Calling a `GzipFile` object's **`close()`** method does not close *fileobj*, since you might wish to append more material after the compressed data. This also allows you to pass an `io.BytesIO` object opened for writing as *fileobj*, and retrieve the resulting memory buffer using the `io.BytesIO` object's `getvalue()` method.

`GzipFile` supports the `io.BufferedIOBase` interface, including iteration and the `with` statement. Only the `truncate()` method isn't implemented.

`GzipFile` also provides the following method and attribute:

peek(*n*)

Read *n* uncompressed bytes without advancing the file position. The number of bytes returned may be more or less than requested.

Note

While calling `peek()` does not change the file position of the `GzipFile`, it may change the position of the underlying file object (e.g. if the `GzipFile` was constructed with the *fileobj* parameter).

Added in version 3.2.

mode

'rb' for reading and 'wb' for writing.

Changed in version 3.13: In previous versions it was an integer `1` or `2`.

mtime

When decompressing, this attribute is set to the last timestamp in the most recently read header. It is an integer, holding the number of seconds since the Unix epoch (00:00:00 UTC, January 1, 1970). The initial value before reading any headers is `None`.

name

The path to the gzip file on disk, as a `str` or `bytes`. Equivalent to the output of `os.fspath()` on the original input path, with no other normalization, resolution or expansion.

Changed in version 3.1: Support for the `with` statement was added, along with the *mtime* constructor argument and `mtime` attribute.

Changed in version 3.2: Support for zero-padded and unseekable files was added.

Changed in version 3.3: The `io.BufferedIOBase.read1()` method is now implemented.

Changed in version 3.4: Added support for the `'x'` and `'xb'` modes.

Changed in version 3.5: Added support for writing arbitrary bytes-like objects. The **`read()`** method now accepts an argument of `None`.

Changed in version 3.6: Accepts a path-like object.

Deprecated since version 3.9: Opening **`GzipFile`** for writing without specifying the *mode* argument is deprecated.

Changed in version 3.12: Remove the `filename` attribute, use the **`name`** attribute instead.

gzip.compress(*data*, *compresslevel=9*, ***, *mtime=0*)

Compress the *data*, returning a **`bytes`** object containing the compressed data. *compresslevel* and *mtime* have the same meaning as in the **`GzipFile`** constructor above, but *mtime* defaults to 0 for reproducible output.

Added in version 3.2.

Changed in version 3.8: Added the *mtime* parameter for reproducible output.

Changed in version 3.11: Speed is improved by compressing all data at once instead of in a streamed fashion. Calls with *mtime* set to `0` are delegated to **`zlib.compress()`** for better speed. In this situation the output may contain a gzip header “OS” byte value other than 255 “unknown” as supplied by the underlying zlib implementation.

Changed in version 3.13: The gzip header OS byte is guaranteed to be set to 255 when this function is used as was the case in 3.10 and earlier.

Changed in version 3.14: The *mtime* parameter now defaults to 0 for reproducible output. For the previous behaviour of using the current time, pass `None` to *mtime*.

gzip.decompress(*data*)

Decompress the *data*, returning a **`bytes`** object containing the uncompressed data. This function is capable of decompressing multi-member gzip data (multiple gzip blocks concatenated together). When the data is certain to contain only one member the **`zlib.decompress()`** function with *wbits* set to 31 is faster.

Added in version 3.2.

Changed in version 3.11: Speed is improved by decompressing members at once in memory instead of in a streamed fashion.

Examples of usage

Example of how to read a compressed file:

```
import gzip
with gzip.open('/home/joe/file.txt.gz', 'rb') as f:
    file_content = f.read()
```

Example of how to create a compressed GZIP file:

```
import gzip
content = b"Lots of content here"
with gzip.open('/home/joe/file.txt.gz', 'wb') as f:
    f.write(content)
```

Example of how to GZIP compress an existing file:

```
import gzip
import shutil
with open('/home/joe/file.txt', 'rb') as f_in:
    with gzip.open('/home/joe/file.txt.gz', 'wb') as f_out:
        shutil.copyfileobj(f_in, f_out)
```

Example of how to GZIP compress a binary string:

```
import gzip
s_in = b"Lots of content here"
s_out = gzip.compress(s_in)
```

See also

Module `zlib`

The basic data compression module needed to support the **gzip** file format.

In case gzip (de)compression is a bottleneck, the python-isal [https://github.com/pycompression/python-isal] package speeds up (de)compression with a mostly compatible API.

Command-line interface

The `gzip` module provides a simple command line interface to compress or decompress files.

Once executed the `gzip` module keeps the input file(s).

Changed in version 3.8: Add a new command line interface with a usage. By default, when you will execute the CLI, the default compression level is 6.

Command-line options

file

If *file* is not specified, read from `sys.stdin`.

--fast

Indicates the fastest compression method (less compression).

--best

Indicates the slowest compression method (best compression).

-d, --decompress

Decompress the given file.

-h, --help

Show the help message.

`bz2` — Support for bzip2 compression

Source code: Lib/bz2.py
[https://github.com/python/cpython/tree/3.14/Lib/bz2.py]

This module provides a comprehensive interface for compressing and decompressing data using the bzip2 compression algorithm.

The **`bz2`** module contains:

- The **`open()`** function and **`BZ2File`** class for reading and writing compressed files.
- The **`BZ2Compressor`** and **`BZ2Decompressor`** classes for incremental (de)compression.
- The **`compress()`** and **`decompress()`** functions for one-shot (de)compression.

This is an optional module. If it is missing from your copy of CPython, look for documentation from your distributor (that is, whoever provided Python to you). If you are the distributor, see Requirements for optional modules.

(De)compression of files

bz2.open(*filename*, *mode='rb'*, *compresslevel=9*, *encoding=None*, *errors=None*, *newline=None*)

Open a bzip2-compressed file in binary or text mode, returning a file object.

As with the constructor for **`BZ2File`**, the *filename* argument can be an actual filename (a **`str`** or **`bytes`** object), or an existing file object to read from or write to.

The *mode* argument can be any of `'r'`, `'rb'`, `'w'`, `'wb'`, `'x'`, `'xb'`, `'a'` or `'ab'` for binary mode, or `'rt'`, `'wt'`, `'xt'`, or `'at'` for text mode. The default is `'rb'`.

The *compresslevel* argument is an integer from 1 to 9, as for the **`BZ2File`** constructor.

For binary mode, this function is equivalent to the **`BZ2File`** constructor: `BZ2File(filename, mode, compresslevel=compresslevel)`. In this case, the *encoding*, *errors* and *newline* arguments must not be provided.

For text mode, a **`BZ2File`** object is created, and wrapped in an **`io.TextIOWrapper`** instance with the specified encoding, error handling behavior, and line ending(s).

Added in version 3.3.

Changed in version 3.4: The `'x'` (exclusive creation) mode was added.

Changed in version 3.6: Accepts a path-like object.

class bz2.BZ2File(*filename*, *mode='r'*, *, *compresslevel=9*)

Open a bzip2-compressed file in binary mode.

If *filename* is a **`str`** or **`bytes`** object, open the named file directly. Otherwise, *filename* should be a file object, which will be used to read or write the compressed data.

The *mode* argument can be either `'r'` for reading (default), `'w'` for overwriting, `'x'` for exclusive creation, or `'a'` for appending. These can equivalently be given as `'rb'`, `'wb'`, `'xb'` and `'ab'` respectively.

If *filename* is a file object (rather than an actual file name), a mode of `'w'` does not truncate the file, and is instead equivalent to `'a'`.

If *mode* is `'w'` or `'a'`, *compresslevel* can be an integer between `1` and `9` specifying the level of compression: `1` produces the least compression, and `9` (default) produces the most compression.

If *mode* is `'r'`, the input file may be the concatenation of multiple compressed streams.

`BZ2File` provides all of the members specified by the **`io.BufferedIOBase`**, except for **`detach()`** and **`truncate()`**. Iteration and the **`with`** statement are supported.

`BZ2File` also provides the following methods and attributes:

peek([*n*])

Return buffered data without advancing the file position. At least one byte of data will be returned (unless at EOF). The exact number of bytes returned is unspecified.

Note

While calling **`peek()`** does not change the file position of the **`BZ2File`**, it may change the position of the underlying file object (e.g. if the **`BZ2File`** was constructed by passing a file object for *filename*).

Added in version 3.3.

fileno()

Return the file descriptor for the underlying file.

Added in version 3.3.

readable()

Return whether the file was opened for reading.

Added in version 3.3.

seekable()

Return whether the file supports seeking.

Added in version 3.3.

writable()

Return whether the file was opened for writing.

Added in version 3.3.

read1(*size=-1*)

Read up to *size* uncompressed bytes, while trying to avoid making multiple reads from the underlying stream. Reads up to a buffer’s worth of data if size is negative.

Returns `b''` if the file is at EOF.

Added in version 3.3.

readinto(*b*)

Read bytes into *b*.

Returns the number of bytes read (0 for EOF).

Added in version 3.3.

mode

`'rb'` for reading and `'wb'` for writing.

Added in version 3.13.

name

The bzip2 file name. Equivalent to the **`name`** attribute of the underlying file object.

Added in version 3.13.

Changed in version 3.1: Support for the **`with`** statement was added.

Changed in version 3.3: Support was added for *filename* being a file object instead of an actual filename.

The `'a'` (append) mode was added, along with support for reading multi-stream files.

Changed in version 3.4: The `'x'` (exclusive creation) mode was added.

Changed in version 3.5: The **`read()`** method now accepts an argument of `None`.

Changed in version 3.6: Accepts a path-like object.

Changed in version 3.9: The *buffering* parameter has been removed. It was ignored and deprecated since Python 3.0. Pass an open file object to control how the file is opened.

The *compresslevel* parameter became keyword-only.

Changed in version 3.10: This class is thread unsafe in the face of multiple simultaneous readers or writers, just like its equivalent classes in `gzip` and `lzma` have always been.

Incremental (de)compression

class bz2.BZ2Compressor(*compresslevel=9*)

Create a new compressor object. This object may be used to compress data incrementally. For one-shot compression, use the `compress()` function instead.

compresslevel, if given, must be an integer between `1` and `9`. The default is `9`.

compress(*data*)

Provide data to the compressor object. Returns a chunk of compressed data if possible, or an empty byte string otherwise.

When you have finished providing data to the compressor, call the `flush()` method to finish the compression process.

flush()

Finish the compression process. Returns the compressed data left in internal buffers.

The compressor object may not be used after this method has been called.

class bz2.BZ2Decompressor

Create a new decompressor object. This object may be used to decompress data incrementally. For one-shot compression, use the `decompress()` function instead.

Note

This class does not transparently handle inputs containing multiple compressed streams, unlike **`decompress()`** and **`BZ2File`**. If you need to decompress a multi-stream input with **`BZ2Decompressor`**, you must use a new decompressor for each stream.

decompress(*data*, *max_length=-1*)

Decompress *data* (a bytes-like object), returning uncompressed data as bytes. Some of *data* may be buffered internally, for use in later calls to **`decompress()`**. The returned data should be concatenated with the output of any previous calls to **`decompress()`**.

If *max_length* is nonnegative, returns at most *max_length* bytes of decompressed data. If this limit is reached and further output can be produced, the **`needs_input`** attribute will be set to `False`. In this case, the next call to **`decompress()`** may provide *data* as `b''` to obtain more of the output.

If all of the input data was decompressed and returned (either because this was less than *max_length* bytes, or because *max_length* was negative), the **`needs_input`** attribute will be set to `True`.

Attempting to decompress data after the end of stream is reached raises an **`EOFError`**. Any data found after the end of the stream is ignored and saved in the **`unused_data`** attribute.

Changed in version 3.5: Added the *max_length* parameter.

eof

`True` if the end-of-stream marker has been reached.

Added in version 3.3.

unused_data

Data found after the end of the compressed stream.

If this attribute is accessed before the end of the stream has been reached, its value will be `b''`.

needs_input

`False` if the **`decompress()`** method can provide more decompressed data before requiring new uncompressed input.

Added in version 3.5.

One-shot (de)compression

bz2.compress(*data*, *compresslevel=9*)

Compress *data*, a bytes-like object.

compresslevel, if given, must be an integer between `1` and `9`. The default is `9`.

For incremental compression, use a **`BZ2Compressor`** instead.

bz2.decompress(*data*)

Decompress *data*, a bytes-like object.

If *data* is the concatenation of multiple compressed streams, decompress all of the streams.

For incremental decompression, use a **`BZ2Decompressor`** instead.

Changed in version 3.3: Support for multi-stream inputs was added.

Examples of usage

Below are some examples of typical usage of the `bz2` module.

Using `compress()` and `decompress()` to demonstrate round-trip compression:

```
>>> import bz2
>>> data = b"""\
... Donec rhoncus quis sapien sit amet molestie. Fusce scelerisque vel augue
... nec ullamcorper. Nam rutrum pretium placerat. Aliquam vel tristique lorem,
... sit amet cursus ante. In interdum laoreet mi, sit amet ultrices purus
... pulvinar a. Nam gravida euismod magna, non varius justo tincidunt feugiat.
... Aliquam pharetra lacus non risus vehicula rutrum. Maecenas aliquam leo
... felis. Pellentesque semper nunc sit amet nibh ullamcorper, ac elementum
... dolor luctus. Curabitur lacinia mi ornare consectetur vestibulum."""
>>> c = bz2.compress(data)
>>> len(data) / len(c)  # Data compression ratio
1.513595166163142
>>> d = bz2.decompress(c)
>>> data == d  # Check equality to original object after round-trip
True
```

Using `BZ2Compressor` for incremental compression:

```
>>> import bz2
>>> def gen_data(chunks=10, chunksize=1000):
...     """Yield incremental blocks of chunksize bytes."""
...     for _ in range(chunks):
...         yield b"z" * chunksize
...
>>> comp = bz2.BZ2Compressor()
>>> out = b""
>>> for chunk in gen_data():
...     # Provide data to the compressor object
...     out = out + comp.compress(chunk)
...
>>> # Finish the compression process.  Call this once you have
>>> # finished providing data to the compressor.
>>> out = out + comp.flush()
```

The example above uses a very “nonrandom” stream of data (a stream of `b"z"` chunks). Random data tends to compress poorly, while ordered, repetitive data usually yields a high compression ratio.

Writing and reading a bzip2-compressed file in binary mode:

```
>>> import bz2
>>> data = b"""\
... Donec rhoncus quis sapien sit amet molestie. Fusce 
scelerisque vel augue
... nec ullamcorper. Nam rutrum pretium placerat. Aliquam vel 
tristique lorem,
... sit amet cursus ante. In interdum laoreet mi, sit amet 
ultrices purus
... pulvinar a. Nam gravida euismod magna, non varius justo 
tincidunt feugiat.
... Aliquam pharetra lacus non risus vehicula rutrum. Maecenas 
aliquam leo
... felis. Pellentesque semper nunc sit amet nibh ullamcorper, 
ac elementum
... dolor luctus. Curabitur lacinia mi ornare consectetur 
vestibulum."""
>>> with bz2.open("myfile.bz2", "wb") as f:
...     # Write compressed data to file
...     unused = f.write(data)
...
>>> with bz2.open("myfile.bz2", "rb") as f:
...     # Decompress data from file
...     content = f.read()
...
>>> content == data  # Check equality to original object after
```

```
round-trip
True
```

`lzma` — Compression using the LZMA algorithm

Added in version 3.3.

Source code: Lib/lzma.py
[https://github.com/python/cpython/tree/3.14/Lib/lzma.py]

This module provides classes and convenience functions for compressing and decompressing data using the LZMA compression algorithm. Also included is a file interface supporting the `.xz` and legacy `.lzma` file formats used by the **xz** utility, as well as raw compressed streams.

The interface provided by this module is very similar to that of the **`bz2`** module. Note that **`LZMAFile`** and **`bz2.BZ2File`** are *not* thread-safe, so if you need to use a single **`LZMAFile`** instance from multiple threads, it is necessary to protect it with a lock.

This is an optional module. If it is missing from your copy of CPython, look for documentation from your distributor (that is, whoever provided Python to you). If you are the distributor, see Requirements for optional modules.

exception lzma.LZMAError

This exception is raised when an error occurs during compression or decompression, or while initializing the compressor/decompressor state.

Reading and writing compressed files

lzma.open(*filename*, *mode='rb'*, *, *format=None*, *check=-1*, *preset=None*, *filters=None*, *encoding=None*, *errors=None*, *newline=None*)

Open an LZMA-compressed file in binary or text mode, returning a file object.

The *filename* argument can be either an actual file name (given as a **`str`**, **`bytes`** or path-like object), in which case the named file is opened, or it can be an existing file object to read from or write to.

The *mode* argument can be any of `"r"`, `"rb"`, `"w"`, `"wb"`, `"x"`, `"xb"`, `"a"` or `"ab"` for binary mode, or `"rt"`, `"wt"`, `"xt"`, or `"at"` for text mode. The default is `"rb"`.

When opening a file for reading, the *format* and *filters* arguments have the same meanings as for **`LZMADecompressor`**. In this case, the *check* and *preset* arguments should not be used.

When opening a file for writing, the *format*, *check*, *preset* and *filters* arguments have the same meanings as for **`LZMACompressor`**.

For binary mode, this function is equivalent to the **`LZMAFile`** constructor: `LZMAFile(filename, mode, ...)`. In this case, the *encoding*, *errors* and *newline* arguments must not be provided.

For text mode, a **`LZMAFile`** object is created, and wrapped in an **`io.TextIOWrapper`** instance with the specified encoding, error handling behavior, and line ending(s).

Changed in version 3.4: Added support for the `"x"`, `"xb"` and `"xt"` modes.

Changed in version 3.6: Accepts a path-like object.

class lzma.LZMAFile(*filename=None*, *mode='r'*, ***, *format=None*, *check=-1*, *preset=None*, *filters=None*)

Open an LZMA-compressed file in binary mode.

An **`LZMAFile`** can wrap an already-open file object, or operate directly on a named file. The *filename* argument specifies either the file object to wrap, or the name of the file to open (as a **`str`**, **`bytes`** or path-like object). When wrapping an existing file object, the wrapped file will not be closed when the **`LZMAFile`** is closed.

The *mode* argument can be either `"r"` for reading (default), `"w"` for overwriting, `"x"` for exclusive creation, or `"a"` for appending. These can equivalently be given as `"rb"`, `"wb"`, `"xb"` and `"ab"` respectively.

If *filename* is a file object (rather than an actual file name), a mode of `"w"` does not truncate the file, and is instead equivalent to `"a"`.

When opening a file for reading, the input file may be the concatenation of multiple separate compressed streams. These are transparently decoded as a single logical stream.

When opening a file for reading, the *format* and *filters* arguments have the same meanings as for **`LZMADecompressor`**. In this case, the *check* and *preset* arguments should not be used.

When opening a file for writing, the *format*, *check*, *preset* and *filters* arguments have the same meanings as for **`LZMACompressor`**.

`LZMAFile` supports all the members specified by `io.BufferedIOBase`, except for `detach()` and `truncate()`. Iteration and the `with` statement are supported.

The following method and attributes are also provided:

peek(*size=-1*)

Return buffered data without advancing the file position. At least one byte of data will be returned, unless EOF has been reached. The exact number of bytes returned is unspecified (the *size* argument is ignored).

> **Note**
>
> While calling `peek()` does not change the file position of the `LZMAFile`, it may change the position of the underlying file object (e.g. if the `LZMAFile` was constructed by passing a file object for *filename*).

mode

`'rb'` for reading and `'wb'` for writing.

Added in version 3.13.

name

The lzma file name. Equivalent to the `name` attribute of the underlying file object.

Added in version 3.13.

Changed in version 3.4: Added support for the `"x"` and `"xb"` modes.

Changed in version 3.5: The `read()` method now accepts an argument of `None`.

Changed in version 3.6: Accepts a path-like object.

Compressing and decompressing data in memory

class lzma.LZMACompressor(*format=FORMAT_XZ*, *check=-1*, *preset=None*, *filters=None*)

Create a compressor object, which can be used to compress data incrementally.

For a more convenient way of compressing a single chunk of data, see `compress()`.

The *format* argument specifies what container format should be used. Possible values are:

- **`FORMAT_XZ`**: The `.xz` container format.
  This is the default format.
- **`FORMAT_ALONE`**: The legacy `.lzma` container format.
  This format is more limited than `.xz` – it does not support integrity checks or multiple filters.
- **`FORMAT_RAW`**: A raw data stream, not using any container format.
  This format specifier does not support integrity checks, and requires that you always specify a custom filter chain (for both compression and decompression). Additionally, data compressed in this manner cannot be decompressed using **`FORMAT_AUTO`** (see `LZMADecompressor`).

The *check* argument specifies the type of integrity check to include in the compressed data. This check is used when decompressing, to ensure that the data has not been corrupted. Possible values are:

- **`CHECK_NONE`**: No integrity check. This is the default (and the only acceptable value) for **`FORMAT_ALONE`** and **`FORMAT_RAW`**.
- **`CHECK_CRC32`**: 32-bit Cyclic Redundancy Check.
- **`CHECK_CRC64`**: 64-bit Cyclic Redundancy Check. This is the default for **`FORMAT_XZ`**.
- **`CHECK_SHA256`**: 256-bit Secure Hash Algorithm.

If the specified check is not supported, an `LZMAError` is raised.

The compression settings can be specified either as a preset compression level (with the *preset* argument), or in detail as a custom filter chain (with the *filters* argument).

The *preset* argument (if provided) should be an integer between `0` and `9` (inclusive), optionally OR-ed with the constant **`PRESET_EXTREME`**. If neither *preset* nor *filters* are given, the default behavior is to use **`PRESET_DEFAULT`** (preset level `6`). Higher presets produce smaller output, but make the compression process slower.

Note

In addition to being more CPU-intensive, compression with higher presets also requires much more memory (and produces output that needs more memory to decompress). With preset `9` for example, the overhead for an `LZMACompressor` object can be as high as 800 MiB. For this reason, it is generally best to stick with the default preset.

The *filters* argument (if provided) should be a filter chain specifier. See Specifying custom filter chains for details.

compress(*data*)

Compress *data* (a `bytes` object), returning a `bytes` object containing compressed data for at least part of the input. Some of *data* may be buffered internally, for use in later calls to `compress()` and `flush()`. The returned data should be concatenated with the output of any previous calls to `compress()`.

flush()

Finish the compression process, returning a `bytes` object containing any data stored in the compressor's internal buffers.

The compressor cannot be used after this method has been called.

class lzma.LZMADecompressor(*format=FORMAT_AUTO*, *memlimit=None*, *filters=None*)

Create a decompressor object, which can be used to decompress data incrementally.

For a more convenient way of decompressing an entire compressed stream at once, see `decompress()`.

The *format* argument specifies the container format that should be used. The default is **`FORMAT_AUTO`**, which can decompress both `.xz` and `.lzma` files. Other possible values are **`FORMAT_XZ`**, **`FORMAT_ALONE`**, and **`FORMAT_RAW`**.

The *memlimit* argument specifies a limit (in bytes) on the amount of memory that the decompressor can use. When this argument is used, decompression will fail with an `LZMAError` if it is not possible to decompress the input within the given memory limit.

The *filters* argument specifies the filter chain that was used to create the stream being decompressed. This argument is required if *format* is **`FORMAT_RAW`**, but should not be used for other formats. See Specifying custom filter chains for more information about filter chains.

Note

This class does not transparently handle inputs containing multiple compressed streams, unlike **`decompress()`** and **`LZMAFile`**. To decompress a multi-stream input with **`LZMADecompressor`**, you must create a new decompressor for each stream.

decompress(*data*, *max_length=-1*)

Decompress *data* (a bytes-like object), returning uncompressed data as bytes. Some of *data* may be buffered internally, for use in later calls to **`decompress()`**. The returned data should be concatenated with the output of any previous calls to **`decompress()`**.

If *max_length* is nonnegative, returns at most *max_length* bytes of decompressed data. If this limit is reached and further output can be produced, the **`needs_input`** attribute will be set to `False`. In this case, the next call to **`decompress()`** may provide *data* as `b''` to obtain more of the output.

If all of the input data was decompressed and returned (either because this was less than *max_length* bytes, or because *max_length* was negative), the **`needs_input`** attribute will be set to `True`.

Attempting to decompress data after the end of stream is reached raises an **`EOFError`**. Any data found after the end of the stream is ignored and saved in the **`unused_data`** attribute.

Changed in version 3.5: Added the *max_length* parameter.

check

The ID of the integrity check used by the input stream. This may be **`CHECK_UNKNOWN`** until enough of the input has been decoded to determine what integrity check it uses.

eof

`True` if the end-of-stream marker has been reached.

unused_data

Data found after the end of the compressed stream.

Before the end of the stream is reached, this will be `b""`.

needs_input

`False` if the **`decompress()`** method can provide more decompressed data before requiring new uncompressed input.

Added in version 3.5.

lzma.compress(*data*, *format=FORMAT_XZ*, *check=-1*, *preset=None*, *filters=None*)

Compress *data* (a **`bytes`** object), returning the compressed data as a **`bytes`** object.

See **`LZMACompressor`** above for a description of the *format*, *check*, *preset* and *filters* arguments.

lzma.decompress(*data*, *format=FORMAT_AUTO*, *memlimit=None*, *filters=None*)

Decompress *data* (a `bytes` object), returning the uncompressed data as a `bytes` object.

If *data* is the concatenation of multiple distinct compressed streams, decompress all of these streams, and return the concatenation of the results.

See `LZMADecompressor` above for a description of the *format*, *memlimit* and *filters* arguments.

Miscellaneous

lzma.is_check_supported(*check*)

Return `True` if the given integrity check is supported on this system.

`CHECK_NONE` and **`CHECK_CRC32`** are always supported. **`CHECK_CRC64`** and **`CHECK_SHA256`** may be unavailable if you are using a version of **liblzma** that was compiled with a limited feature set.

Specifying custom filter chains

A filter chain specifier is a sequence of dictionaries, where each dictionary contains the ID and options for a single filter. Each dictionary must contain the key `"id"`, and may contain additional keys to specify filter-dependent options. Valid filter IDs are as follows:

- Compression filters:
  - **`FILTER_LZMA1`** (for use with **`FORMAT_ALONE`**)
  - **`FILTER_LZMA2`** (for use with **`FORMAT_XZ`** and **`FORMAT_RAW`**)
- Delta filter:
  - **`FILTER_DELTA`**
- Branch-Call-Jump (BCJ) filters:

  - **`FILTER_X86`**
  - **`FILTER_IA64`**
  - **`FILTER_ARM`**
  - **`FILTER_ARMTHUMB`**
  - **`FILTER_POWERPC`**
  - **`FILTER_SPARC`**

A filter chain can consist of up to 4 filters, and cannot be empty. The last filter in the chain must be a compression filter, and any other filters must be delta or BCJ filters.

Compression filters support the following options (specified as additional entries in the dictionary representing the filter):

- `preset`: A compression preset to use as a source of default values for options that are not specified explicitly.
- `dict_size`: Dictionary size in bytes. This should be between 4 KiB and 1.5 GiB (inclusive).
- `lc`: Number of literal context bits.
- `lp`: Number of literal position bits. The sum `lc + lp` must be at most 4.
- `pb`: Number of position bits; must be at most 4.
- `mode`: **`MODE_FAST`** or **`MODE_NORMAL`**.
- `nice_len`: What should be considered a “nice length” for a match. This should be 273 or less.
- `mf`: What match finder to use – **`MF_HC3`**, **`MF_HC4`**, **`MF_BT2`**, **`MF_BT3`**, or **`MF_BT4`**.
- `depth`: Maximum search depth used by match finder. 0 (default) means to select automatically based on other filter options.

The delta filter stores the differences between bytes, producing more repetitive input for the compressor in certain circumstances. It supports one option, `dist`. This indicates the distance between bytes to be subtracted. The default is 1, i.e. take the differences between adjacent bytes.

The BCJ filters are intended to be applied to machine code. They convert relative branches, calls and jumps in the code to use absolute addressing, with the aim of increasing the redundancy that can be exploited by the compressor. These filters support one option, `start_offset`. This specifies the address that should be mapped to the beginning of the input data. The default is 0.

Examples

Reading in a compressed file:

```
import lzma
with lzma.open("file.xz") as f:
    file_content = f.read()
```

Creating a compressed file:

```
import lzma
data = b"Insert Data Here"
with lzma.open("file.xz", "w") as f:
    f.write(data)
```

Compressing data in memory:

```
import lzma
data_in = b"Insert Data Here"
data_out = lzma.compress(data_in)
```

Incremental compression:

```
import lzma
lzc = lzma.LZMACompressor()
out1 = lzc.compress(b"Some data\n")
out2 = lzc.compress(b"Another piece of data\n")
```

```
out3 = lzc.compress(b"Even more data\n")
out4 = lzc.flush()
# Concatenate all the partial results:
result = b"".join([out1, out2, out3, out4])
```

Writing compressed data to an already-open file:

```
import lzma
with open("file.xz", "wb") as f:
    f.write(b"This data will not be compressed\n")
    with lzma.open(f, "w") as lzf:
        lzf.write(b"This *will* be compressed\n")
    f.write(b"Not compressed\n")
```

Creating a compressed file using a custom filter chain:

```
import lzma
my_filters = [
    {"id": lzma.FILTER_DELTA, "dist": 5},
    {"id": lzma.FILTER_LZMA2, "preset": 7 |
lzma.PRESET_EXTREME},
]
with lzma.open("file.xz", "w", filters=my_filters) as f:
    f.write(b"blah blah blah")
```

`zipfile` — Work with ZIP archives

Source code: Lib/zipfile/ [https://github.com/python/cpython/tree/3.14/Lib/zipfile/]

The ZIP file format is a common archive and compression standard. This module provides tools to create, read, write, append, and list a ZIP file. Any advanced use of this module will require an understanding of the format, as defined in PKZIP Application Note [https://pkware.cachefly.net/webdocs/casestudies/APPNOTE.TXT].

This module does not handle multipart ZIP files. It can handle ZIP files that use the ZIP64 extensions (that is ZIP files that are more than 4 GiB in size). It supports decryption of encrypted files in ZIP archives, but it cannot create an encrypted file. Decryption is extremely slow as it is implemented in native Python rather than C.

Handling compressed archives requires optional modules such as **`zlib`**, **`bz2`**, **`lzma`**, and **`compression.zstd`**. If any of them are missing from your copy of CPython, look for documentation from your distributor (that is, whoever provided Python to you). If you are the distributor, see Requirements for optional modules.

The module defines the following items:

exception zipfile.BadZipFile

The error raised for bad ZIP files.

Added in version 3.2.

exception zipfile.BadZipfile

Alias of `BadZipFile`, for compatibility with older Python versions.

Deprecated since version 3.2.

exception zipfile.LargeZipFile

The error raised when a ZIP file would require ZIP64 functionality but that has not been enabled.

class zipfile.ZipFile

The class for reading and writing ZIP files. See section ZipFile objects for constructor details.

class zipfile.Path

Class that implements a subset of the interface provided by `pathlib.Path`, including the full `importlib.resources.abc.Traversable` interface.

Added in version 3.8.

class zipfile.PyZipFile

Class for creating ZIP archives containing Python libraries.

class zipfile.ZipInfo(*filename='NoName'*, *date_time=(1980, 1, 1, 0, 0, 0)*)

Class used to represent information about a member of an archive. Instances of this class are returned by the `getinfo()` and `infolist()` methods of `ZipFile` objects. Most users of the `zipfile` module will not need to create these, but only use those created by this module. *filename* should be the full name of the archive member, and *date_time* should be a tuple containing six fields which describe the time of the last modification to the file; the fields are described in section ZipInfo objects.

Changed in version 3.13: A public **`compress_level`** attribute has been added to expose the formerly protected **`_compresslevel`**. The older protected name continues to work as a property for backwards compatibility.

_for_archive(*archive*)

Resolve the date_time, compression attributes, and external attributes to suitable defaults as used by **`ZipFile.writestr()`**.

Returns self for chaining.

Added in version 3.14.

zipfile.is_zipfile(*filename*)

Returns `True` if *filename* is a valid ZIP file based on its magic number, otherwise returns `False`. *filename* may be a file or file-like object too.

Changed in version 3.1: Support for file and file-like objects.

zipfile.ZIP_STORED

The numeric constant for an uncompressed archive member.

zipfile.ZIP_DEFLATED

The numeric constant for the usual ZIP compression method. This requires the **`zlib`** module.

zipfile.ZIP_BZIP2

The numeric constant for the BZIP2 compression method. This requires the **`bz2`** module.

Added in version 3.3.

zipfile.ZIP_LZMA

The numeric constant for the LZMA compression method. This requires the `lzma` module.

Added in version 3.3.

zipfile.ZIP_ZSTANDARD

The numeric constant for Zstandard compression. This requires the `compression.zstd` module.

Note

In APPNOTE 6.3.7, the method ID `20` was assigned to Zstandard compression. This was changed in APPNOTE 6.3.8 to method ID `93` to avoid conflicts, with method ID `20` being deprecated. For compatibility, the **`zipfile`** module reads both method IDs but will only write data with method ID `93`.

Added in version 3.14.

Note

The ZIP file format specification has included support for bzip2 compression since 2001, for LZMA compression since 2006, and Zstandard compression since 2020. However, some tools (including older Python releases) do not support these compression methods, and may either refuse to process the ZIP file altogether, or fail to extract individual files.

See also

PKZIP Application Note [https://pkware.cachefly.net/webdocs/casestudies/APPNOTE.TXT]
Documentation on the ZIP file format by Phil Katz, the creator of the format and algorithms used.

Info-ZIP Home Page [https://infozip.sourceforge.net/]
Information about the Info-ZIP project’s ZIP archive programs and development libraries.

ZipFile objects

class zipfile.ZipFile(*file*, *mode='r'*, *compression=ZIP_STORED*, *allowZip64=True*, *compresslevel=None*, ***, *strict_timestamps=True*, *metadata_encoding=None*)

Open a ZIP file, where *file* can be a path to a file (a string), a file-like object or a path-like object.

The *mode* parameter should be `'r'` to read an existing file, `'w'` to truncate and write a new file, `'a'` to append to an existing file, or `'x'` to exclusively create and write a new file. If *mode* is `'x'` and *file* refers to an existing file, a `FileExistsError` will be raised. If *mode* is `'a'` and *file* refers to an existing ZIP file, then additional files are added to it. If *file* does not refer to a ZIP file, then a new ZIP archive is appended to the file. This is meant for adding a ZIP archive to another file (such as `python.exe`). If *mode* is `'a'` and the file does not exist at all, it is created. If *mode* is `'r'` or `'a'`, the file should be seekable.

compression is the ZIP compression method to use when writing the archive, and should be `ZIP_STORED`, `ZIP_DEFLATED`, `ZIP_BZIP2`, `ZIP_LZMA`, or `ZIP_ZSTANDARD`; unrecognized values will cause `NotImplementedError` to be raised. If `ZIP_DEFLATED`, `ZIP_BZIP2`, `ZIP_LZMA`, or `ZIP_ZSTANDARD` is specified but the corresponding module (`zlib`, `bz2`, `lzma`, or `compression.zstd`) is not available, `RuntimeError` is raised. The default is `ZIP_STORED`.

If *allowZip64* is `True` (the default) zipfile will create ZIP files that use the ZIP64 extensions when the zipfile is larger than 4 GiB. If it is `false` `zipfile` will raise an exception when the ZIP file would require ZIP64 extensions.

The *compresslevel* parameter controls the compression level to use when writing files to the archive. When using `ZIP_STORED` or `ZIP_LZMA` it has no effect. When using `ZIP_DEFLATED` integers `0` through `9` are accepted (see `zlib` for more information). When using `ZIP_BZIP2` integers `1` through `9` are accepted (see `bz2` for more information). When using `ZIP_ZSTANDARD` integers `-131072` through `22` are commonly accepted (see `CompressionParameter.compression_level` for more on retrieving valid values and their meaning).

The *strict_timestamps* argument, when set to `False`, allows to zip files older than 1980-01-01 at the cost of setting the timestamp to 1980-01-01. Similar behavior occurs with files newer than 2107-12-31, the timestamp is also set to the limit.

When mode is `'r'`, *metadata_encoding* may be set to the name of a codec, which will be used to decode metadata such as the names of members and ZIP comments.

If the file is created with mode `'w'`, `'x'` or `'a'` and then `closed` without adding any files to the archive, the appropriate ZIP structures for an empty archive will be written to the file.

ZipFile is also a context manager and therefore supports the `with` statement. In the example, *myzip* is closed after the **`with`** statement's suite is finished—even if an exception occurs:

```
with ZipFile('spam.zip', 'w') as myzip:
    myzip.write('eggs.txt')
```

Note

metadata_encoding is an instance-wide setting for the ZipFile. It is not possible to set this on a per-member basis.

This attribute is a workaround for legacy implementations which produce archives with names in the current locale encoding or code page (mostly on Windows). According to the .ZIP standard, the encoding of metadata may be specified to be either IBM code page (default) or UTF-8 by a flag in the archive header. That flag takes precedence over *metadata_encoding*, which is a Python-specific extension.

Changed in version 3.2: Added the ability to use `ZipFile` as a context manager.

Changed in version 3.3: Added support for `bzip2` and `lzma` compression.

Changed in version 3.4: ZIP64 extensions are enabled by default.

Changed in version 3.5: Added support for writing to unseekable streams. Added support for the `'x'` mode.

Changed in version 3.6: Previously, a plain `RuntimeError` was raised for unrecognized compression values.

Changed in version 3.6.2: The *file* parameter accepts a path-like object.

Changed in version 3.7: Add the *compresslevel* parameter.

Changed in version 3.8: The *strict_timestamps* keyword-only parameter.

Changed in version 3.11: Added support for specifying member name encoding for reading metadata in the zipfile's directory and file headers.

ZipFile.close()

Close the archive file. You must call `close()` before exiting your program or essential records will not be written.

ZipFile.getinfo(*name*)

Return a `ZipInfo` object with information about the archive member *name*. Calling `getinfo()` for a name not currently contained in the archive will raise a `KeyError`.

ZipFile.infolist()

Return a list containing a `ZipInfo` object for each member of the archive. The objects are in the same order as their entries in the actual ZIP file on disk if an existing archive was opened.

ZipFile.namelist()

Return a list of archive members by name.

ZipFile.open(*name*, *mode='r'*, *pwd=None*, *, *force_zip64=False*)

Access a member of the archive as a binary file-like object. *name* can be either the name of a file within the archive or a **`ZipInfo`** object. The *mode* parameter, if included, must be `'r'` (the default) or `'w'`. *pwd* is the password used to decrypt encrypted ZIP files as a **`bytes`** object.

`open()` is also a context manager and therefore supports the **`with`** statement:

```
with ZipFile('spam.zip') as myzip:
    with myzip.open('eggs.txt') as myfile:
        print(myfile.read())
```

With *mode* `'r'` the file-like object (`ZipExtFile`) is read-only and provides the following methods: **`read()`**, **`readline()`**, **`readlines()`**, **`seek()`**, **`tell()`**, **`__iter__()`**, **`__next__()`**. These objects can operate independently of the ZipFile.

With `mode='w'`, a writable file handle is returned, which supports the **`write()`** method. While a writable file handle is open, attempting to read or write other files in the ZIP file will raise a **`ValueError`**.

In both cases the file-like object has also attributes **`name`**, which is equivalent to the name of a file within the archive, and **`mode`**, which is `'rb'` or `'wb'` depending on the input mode.

When writing a file, if the file size is not known in advance but may exceed 2 GiB, pass `force_zip64=True` to ensure that the header format is capable of supporting large files. If the file size is known in advance, construct a **`ZipInfo`** object with **`file_size`** set, and use that as the *name* parameter.

Note

The **`open()`**, **`read()`** and **`extract()`** methods can take a filename or a **`ZipInfo`** object. You will appreciate this when trying to read a ZIP file that contains members with duplicate names.

Changed in version 3.6: Removed support of `mode='U'`. Use **`io.TextIOWrapper`** for reading compressed text files in universal newlines mode.

Changed in version 3.6: **`ZipFile.open()`** can now be used to write files into the archive with the `mode='w'` option.

Changed in version 3.6: Calling **`open()`** on a closed ZipFile will raise a **`ValueError`**. Previously, a **`RuntimeError`** was raised.

Changed in version 3.13: Added attributes **`name`** and **`mode`** for the writeable file-like object. The value of the **`mode`** attribute for the readable file-like object was changed from `'r'` to `'rb'`.

ZipFile.extract(*member*, *path=None*, *pwd=None*)

Extract a member from the archive to the current working directory; *member* must be its full name or a **`ZipInfo`** object. Its file information is extracted as accurately as possible. *path* specifies a different directory to extract to. *member* can be a filename or a **`ZipInfo`** object. *pwd* is the password used for encrypted files as a **`bytes`** object.

Returns the normalized path created (a directory or new file).

Note

If a member filename is an absolute path, a drive/UNC sharepoint and leading (back)slashes will be stripped, e.g.: `///foo/bar` becomes `foo/bar` on Unix, and `C:\foo\bar` becomes `foo\bar` on Windows. And all `".."` components in a member filename will be removed, e.g.: `../../foo../../ba..r` becomes `foo../ba..r`. On Windows illegal characters (`:`, `<`, `>`, `|`, `"`, `?`, and `*`) replaced by underscore (`_`).

Changed in version 3.6: Calling **`extract()`** on a closed ZipFile will raise a **`ValueError`**. Previously, a **`RuntimeError`** was raised.

Changed in version 3.6.2: The *path* parameter accepts a path-like object.

ZipFile.extractall(*path=None*, *members=None*, *pwd=None*)

Extract all members from the archive to the current working directory. *path* specifies a different directory to extract to. *members* is optional and must be a subset of the list returned by **`namelist()`**. *pwd* is the password used for encrypted files as a **`bytes`** object.

Warning

Never extract archives from untrusted sources without prior inspection. It is possible that files are created outside of *path*, e.g. members that have absolute filenames starting with `"/"` or filenames with two dots `".."`. This module attempts to prevent that. See **`extract()`** note.

Changed in version 3.6: Calling **`extractall()`** on a closed ZipFile will raise a **`ValueError`**. Previously, a **`RuntimeError`** was raised.

Changed in version 3.6.2: The *path* parameter accepts a path-like object.

ZipFile.printdir()

Print a table of contents for the archive to `sys.stdout`.

ZipFile.setpassword(*pwd*)

Set *pwd* (a **`bytes`** object) as default password to extract encrypted files.

ZipFile.read(*name*, *pwd=None*)

Return the bytes of the file *name* in the archive. *name* is the name of the file in the archive, or a **`ZipInfo`** object. The archive must be open for read or append. *pwd* is the password used for encrypted files as a **`bytes`** object and, if specified, overrides the default password set with **`setpassword()`**. Calling **`read()`** on a ZipFile that uses a compression method other than **`ZIP_STORED`**, **`ZIP_DEFLATED`**, **`ZIP_BZIP2`**, **`ZIP_LZMA`**, or **`ZIP_ZSTANDARD`** will raise a **`NotImplementedError`**. An error will also be raised if the corresponding compression module is not available.

Changed in version 3.6: Calling **`read()`** on a closed ZipFile will raise a **`ValueError`**. Previously, a **`RuntimeError`** was raised.

ZipFile.testzip()

Read all the files in the archive and check their CRC’s and file headers. Return the name of the first bad file, or else return `None`.

Changed in version 3.6: Calling **`testzip()`** on a closed ZipFile will raise a **`ValueError`**. Previously, a **`RuntimeError`** was raised.

ZipFile.write(*filename*, *arcname=None*, *compress_type=None*, *compresslevel=None*)

Write the file named *filename* to the archive, giving it the archive name *arcname* (by default, this will be the same as *filename*, but without a drive letter and with leading path separators removed). If given, *compress_type* overrides the value given for the *compression* parameter to the constructor for the new entry. Similarly, *compresslevel* will override the constructor if given. The archive must be open with mode `'w'`, `'x'` or `'a'`.

Note

The ZIP file standard historically did not specify a metadata encoding, but strongly recommended CP437 (the original IBM PC encoding) for interoperability. Recent versions allow use of UTF-8 (only). In this module, UTF-8 will automatically be used to write the member names if they contain any non-ASCII characters. It is not possible to write member names in any encoding other than ASCII or UTF-8.

Note

Archive names should be relative to the archive root, that is, they should not start with a path separator.

Note

If `arcname` (or `filename`, if `arcname` is not given) contains a null byte, the name of the file in the archive will be truncated at the null byte.

Note

A leading slash in the filename may lead to the archive being impossible to open in some zip programs on Windows systems.

Changed in version 3.6: Calling `write()` on a ZipFile created with mode `'r'` or a closed ZipFile will raise a `ValueError`. Previously, a `RuntimeError` was raised.

ZipFile.writestr(*zinfo_or_arcname*, *data*, *compress_type=None*, *compresslevel=None*)

Write a file into the archive. The contents is *data*, which may be either a `str` or a `bytes` instance; if it is a `str`, it is encoded as UTF-8 first. *zinfo_or_arcname* is either the file name it will be given in the archive, or a `ZipInfo` instance. If it's an instance, at least the filename, date, and time must be given. If it's a name, the date and time is set to the current date and time. The archive must be opened with mode `'w'`, `'x'` or `'a'`.

If given, *compress_type* overrides the value given for the *compression* parameter to the constructor for the new entry, or in the *zinfo_or_arcname* (if that is a `ZipInfo` instance). Similarly, *compresslevel* will override the constructor if given.

Note

When passing a `ZipInfo` instance as the *zinfo_or_arcname* parameter, the compression method used will be that specified in the *compress_type* member of the given `ZipInfo` instance. By default, the `ZipInfo` constructor sets this member to `ZIP_STORED`.

Changed in version 3.2: The *compress_type* argument.

Changed in version 3.6: Calling `writestr()` on a ZipFile created with mode `'r'` or a closed ZipFile will raise a `ValueError`. Previously, a `RuntimeError` was raised.

ZipFile.mkdir(*zinfo_or_directory*, *mode=511*)

Create a directory inside the archive. If *zinfo_or_directory* is a string, a directory is created inside the archive with the mode that is specified in the *mode* argument. If, however, *zinfo_or_directory* is a `ZipInfo` instance then the *mode* argument is ignored.

The archive must be opened with mode `'w'`, `'x'` or `'a'`.

Added in version 3.11.

The following data attributes are also available:

ZipFile.filename

Name of the ZIP file.

ZipFile.debug

The level of debug output to use. This may be set from `0` (the default, no output) to `3` (the most output). Debugging information is written to `sys.stdout`.

ZipFile.comment

The comment associated with the ZIP file as a `bytes` object. If assigning a comment to a `ZipFile` instance created with mode `'w'`, `'x'` or `'a'`, it should be no longer than 65535 bytes. Comments longer than this will be truncated.

Path objects

class zipfile.Path(*root*, *at=''*)

Construct a Path object from a `root` zipfile (which may be a `ZipFile` instance or `file` suitable for passing to the `ZipFile` constructor).

`at` specifies the location of this Path within the zipfile, e.g. ‘dir/file.txt’, ‘dir/’, or ‘’. Defaults to the empty string, indicating the root.

Note

The `Path` class does not sanitize filenames within the ZIP archive. Unlike the `ZipFile.extract()` and `ZipFile.extractall()` methods, it is the caller’s responsibility to validate or sanitize filenames to prevent path traversal vulnerabilities (e.g., filenames containing “..” or absolute paths). When handling untrusted archives, consider resolving filenames using `os.path.abspath()` and checking against the target directory with `os.path.commonpath()`.

Path objects expose the following features of `pathlib.Path` objects:

Path objects are traversable using the `/` operator or `joinpath`.

Path.name

The final path component.

Path.open(*mode*='*r*', *, *pwd*, **)

Invoke `ZipFile.open()` on the current path. Allows opening for read or write, text or binary through supported modes: ‘r’, ‘w’, ‘rb’, ‘wb’. Positional and keyword arguments are passed through to `io.TextIOWrapper` when opened as text and ignored otherwise. `pwd` is the `pwd` parameter to `ZipFile.open()`.

Changed in version 3.9: Added support for text and binary modes for open. Default mode is now text.

Changed in version 3.11.2: The `encoding` parameter can be supplied as a positional argument without causing a `TypeError`. As it could in 3.9. Code needing to be compatible with unpatched 3.10 and 3.11 versions must pass all `io.TextIOWrapper` arguments, `encoding` included, as keywords.

Path.iterdir()

Enumerate the children of the current directory.

Path.is_dir()

Return `True` if the current context references a directory.

Path.is_file()

Return `True` if the current context references a file.

Path.is_symlink()

Return `True` if the current context references a symbolic link.

Added in version 3.12.

Changed in version 3.13: Previously, `is_symlink` would unconditionally return `False`.

Path.exists()

Return `True` if the current context references a file or directory in the zip file.

Path.suffix

The last dot-separated portion of the final component, if any. This is commonly called the file extension.

Added in version 3.11: Added `Path.suffix` property.

Path.stem

The final path component, without its suffix.

Added in version 3.11: Added `Path.stem` property.

Path.suffixes

A list of the path’s suffixes, commonly called file extensions.

Added in version 3.11: Added `Path.suffixes` property.

Path.read_text(*, **)

Read the current file as unicode text. Positional and keyword arguments are passed through to `io.TextIOWrapper` (except `buffer`, which is implied by the context).

Changed in version 3.11.2: The `encoding` parameter can be supplied as a positional argument without causing a `TypeError`. As it could in 3.9. Code needing to be compatible with unpatched 3.10 and 3.11 versions must pass all `io.TextIOWrapper` arguments, `encoding` included, as keywords.

Path.read_bytes()

Read the current file as bytes.

Path.joinpath(**other*)

Return a new Path object with each of the *other* arguments joined. The following are equivalent:

```
>>> Path(...).joinpath('child').joinpath('grandchild')
>>> Path(...).joinpath('child', 'grandchild')
>>> Path(...) / 'child' / 'grandchild'
```

Changed in version 3.10: Prior to 3.10, `joinpath` was undocumented and accepted exactly one parameter.

The zipp [https://pypi.org/project/zipp/] project provides backports of the latest path object functionality to older Pythons. Use `zipp.Path` in place of `zipfile.Path` for early access to changes.

PyZipFile objects

The **`PyZipFile`** constructor takes the same parameters as the **`ZipFile`** constructor, and one additional parameter, *optimize*.

class zipfile.PyZipFile(*file*, *mode='r'*, *compression=ZIP_STORED*, *allowZip64=True*, *optimize=-1*)

Changed in version 3.2: Added the *optimize* parameter.

Changed in version 3.4: ZIP64 extensions are enabled by default.

Instances have one method in addition to those of **`ZipFile`** objects:

writepy(*pathname*, *basename=''*, *filterfunc=None*)

Search for files `*.py` and add the corresponding file to the archive.

If the *optimize* parameter to **`PyZipFile`** was not given or `-1`, the corresponding file is a `*.pyc` file, compiling if necessary.

If the *optimize* parameter to **`PyZipFile`** was `0`, `1` or `2`, only files with that optimization level (see **`compile()`**) are added to the archive, compiling if necessary.

If *pathname* is a file, the filename must end with `.py`, and just the (corresponding `*.pyc`) file is added at the top level (no path information). If *pathname* is a file that does not end with `.py`, a **`RuntimeError`** will be raised. If it is a directory, and the directory is not a package directory, then all the files `*.pyc` are added at the top level. If the directory is a package directory, then all `*.pyc` are added under the package name as a file path, and if any subdirectories are package directories, all of these are added recursively in sorted order.

basename is intended for internal use only.

filterfunc, if given, must be a function taking a single string argument. It will be passed each path (including each individual full file path) before it is added to the archive. If *filterfunc* returns a false value, the path will not be added, and if it is a directory its contents will be ignored. For example, if our test files are all either in `test` directories or start with the string `test_`, we can use a *filterfunc* to exclude them:

```
>>> zf = PyZipFile('myprog.zip')
>>> def notests(s):
...     fn = os.path.basename(s)
...     return (not (fn == 'test' or
fn.startswith('test_')))
...
>>> zf.writepy('myprog', filterfunc=notests)
```

The **`writepy()`** method makes archives with file names like this:

```
string.pyc                   # Top level name
test/__init__.pyc            # Package directory
test/testall.pyc             # Module test.testall
test/bogus/__init__.pyc      # Subpackage directory
test/bogus/myfile.pyc        # Submodule
test.bogus.myfile
```

Changed in version 3.4: Added the *filterfunc* parameter.

Changed in version 3.6.2: The *pathname* parameter accepts a path-like object.

Changed in version 3.7: Recursion sorts directory entries.

ZipInfo objects

Instances of the **`ZipInfo`** class are returned by the **`getinfo()`** and **`infolist()`** methods of **`ZipFile`** objects. Each object stores information about a single member of the ZIP archive.

There is one classmethod to make a **`ZipInfo`** instance for a filesystem file:

classmethod ZipInfo.from_file(*filename*, *arcname=None*, *, *strict_timestamps=True*)

Construct a **`ZipInfo`** instance for a file on the filesystem, in preparation for adding it to a zip file.

filename should be the path to a file or directory on the filesystem.

If *arcname* is specified, it is used as the name within the archive. If *arcname* is not specified, the name will be the same as *filename*, but with any drive letter and leading path separators removed.

The *strict_timestamps* argument, when set to `False`, allows to zip files older than 1980-01-01 at the cost of setting the timestamp to 1980-01-01. Similar behavior occurs with files newer than 2107-12-31, the timestamp is also set to the limit.

Added in version 3.6.

Changed in version 3.6.2: The *filename* parameter accepts a path-like object.

Changed in version 3.8: Added the *strict_timestamps* keyword-only parameter.

Instances have the following methods and attributes:

ZipInfo.is_dir()

Return `True` if this archive member is a directory.

This uses the entry's name: directories should always end with `/`.

Added in version 3.6.

ZipInfo.filename

Name of the file in the archive.

ZipInfo.date_time

The time and date of the last modification to the archive member. This is a tuple of six values representing the "last [modified] file time" and "last [modified] file date" fields from the ZIP file's central directory.

The tuple contains:

Index	Value
`0`	Year (>= 1980)
`1`	Month (one-based)
`2`	Day of month (one-based)

Index	Value
3	Hours (zero-based)
4	Minutes (zero-based)
5	Seconds (zero-based)

Note

The ZIP format supports multiple timestamp fields in different locations (central directory, extra fields for NTFS/UNIX systems, etc.). This attribute specifically returns the timestamp from the central directory. The central directory timestamp format in ZIP files does not support timestamps before 1980. While some extra field formats (such as UNIX timestamps) can represent earlier dates, this attribute only returns the central directory timestamp.

The central directory timestamp is interpreted as representing local time, rather than UTC time, to match the behavior of other zip tools.

ZipInfo.compress_type

Type of compression for the archive member.

ZipInfo.comment

Comment for the individual archive member as a `bytes` object.

ZipInfo.extra

Expansion field data. The PKZIP Application Note [https://pkware.cachefly.net/webdocs/casestudies/APPNOTE.TXT] contains some comments on the internal structure of the data contained in this `bytes` object.

ZipInfo.create_system
System which created ZIP archive.

ZipInfo.create_version
PKZIP version which created ZIP archive.

ZipInfo.extract_version
PKZIP version needed to extract archive.

ZipInfo.reserved
Must be zero.

ZipInfo.flag_bits
ZIP flag bits.

ZipInfo.volume
Volume number of file header.

ZipInfo.internal_attr
Internal attributes.

ZipInfo.external_attr
External file attributes.

ZipInfo.header_offset
Byte offset to the file header.

ZipInfo.CRC
CRC-32 of the uncompressed file.

ZipInfo.compress_size
Size of the compressed data.

ZipInfo.file_size

Size of the uncompressed file.

Command-line interface

The `zipfile` module provides a simple command-line interface to interact with ZIP archives.

If you want to create a new ZIP archive, specify its name after the `-c` option and then list the filename(s) that should be included:

```
$ python -m zipfile -c monty.zip spam.txt eggs.txt
```

Passing a directory is also acceptable:

```
$ python -m zipfile -c monty.zip life-of-brian_1979/
```

If you want to extract a ZIP archive into the specified directory, use the `-e` option:

```
$ python -m zipfile -e monty.zip target-dir/
```

For a list of the files in a ZIP archive, use the `-l` option:

```
$ python -m zipfile -l monty.zip
```

Command-line options

-l <zipfile>
--list <zipfile>

List files in a zipfile.

-c <zipfile> <source1> ... <sourceN>
--create <zipfile> <source1> ... <sourceN>

Create zipfile from source files.

-e <zipfile> <output_dir>
--extract <zipfile> <output_dir>

Extract zipfile into target directory.

-t <zipfile>
--test <zipfile>

Test whether the zipfile is valid or not.

--metadata-encoding <encoding>

Specify encoding of member names for `-l`, `-e` and `-t`.

Added in version 3.11.

Decompression pitfalls

The extraction in zipfile module might fail due to some pitfalls listed below.

From file itself

Decompression may fail due to incorrect password / CRC checksum / ZIP format or unsupported compression method / decryption.

File system limitations

Exceeding limitations on different file systems can cause decompression failed. Such as allowable characters in the directory entries, length of the file name, length of the pathname, size of a single file, and number of files, etc.

Resources limitations

The lack of memory or disk volume would lead to decompression failed. For example, decompression bombs (aka ZIP bomb [https://en.wikipedia.org/wiki/Zip_bomb]) apply to zipfile library that can cause disk volume exhaustion.

Interruption

Interruption during the decompression, such as pressing control-C or killing the decompression process may result in incomplete decompression of the archive.

Default behaviors of extraction

Not knowing the default extraction behaviors can cause unexpected decompression results. For example, when extracting the same archive twice, it overwrites files without asking.

`tarfile` — Read and write tar archive files

Source code: Lib/tarfile.py
[https://github.com/python/cpython/tree/3.14/Lib/tarfile.py]

The **`tarfile`** module makes it possible to read and write tar archives, including those using gzip, bz2 and lzma compression. Use the **`zipfile`** module to read or write `.zip` files, or the higher-level functions in shutil.

Some facts and figures:

- reads and writes **`gzip`**, **`bz2`**, **`compression.zstd`**, and **`lzma`** compressed archives if the respective modules are available.

  If any of these optional modules are missing from your copy of CPython, look for documentation from your distributor (that is, whoever provided Python to you). If you are the distributor, see Requirements for optional modules.

- read/write support for the POSIX.1-1988 (ustar) format.
- read/write support for the GNU tar format including *longname* and *longlink* extensions, read-only support for all variants of the *sparse* extension including restoration of sparse files.
- read/write support for the POSIX.1-2001 (pax) format.

- handles directories, regular files, hardlinks, symbolic links, fifos, character devices and block devices and is able to acquire and restore file information like timestamp, access permissions and owner.

Changed in version 3.3: Added support for **`lzma`** compression.

Changed in version 3.12: Archives are extracted using a filter, which makes it possible to either limit surprising/dangerous features, or to acknowledge that they are expected and the archive is fully trusted.

Changed in version 3.14: Set the default extraction filter to **`data`**, which disallows some dangerous features such as links to absolute paths or paths outside of the destination. Previously, the filter strategy was equivalent to **`fully_trusted`**.

Changed in version 3.14: Added support for Zstandard compression using **`compression.zstd`**.

tarfile.open(*name=None*, *mode='r'*, *fileobj=None*, *bufsize=10240*, ***kwargs*)

Return a **`TarFile`** object for the pathname *name*. For detailed information on **`TarFile`** objects and the keyword arguments that are allowed, see TarFile Objects.

mode has to be a string of the form `'filemode[:compression]'`, it defaults to `'r'`. Here is a full list of mode combinations:

mode	action
`'r'` or `'r:*'`	Open for reading with transparent compression (recommended).

mode	action
`'r:'`	Open for reading exclusively without compression.
`'r:gz'`	Open for reading with gzip compression.
`'r:bz2'`	Open for reading with bzip2 compression.
`'r:xz'`	Open for reading with lzma compression.
`'r:zst'`	Open for reading with Zstandard compression.
`'x'` or `'x:'`	Create a tarfile exclusively without compression. Raise a `FileExistsError` exception if it already exists.
`'x:gz'`	Create a tarfile with gzip compression. Raise a `FileExistsError` exception if it already exists.
`'x:bz2'`	Create a tarfile with bzip2 compression. Raise a `FileExistsError` exception if it already exists.
`'x:xz'`	Create a tarfile with lzma compression. Raise a `FileExistsError` exception if it already exists.
`'x:zst'`	Create a tarfile with Zstandard compression. Raise a `FileExistsError` exception if it already exists.
`'a'` or `'a:'`	Open for appending with no compression. The file is created if it does not exist.
`'w'` or `'w:'`	Open for uncompressed writing.

mode	action
`'w:gz'`	Open for gzip compressed writing.
`'w:bz2'`	Open for bzip2 compressed writing.
`'w:xz'`	Open for lzma compressed writing.
`'w:zst'`	Open for Zstandard compressed writing.

Note that `'a:gz'`, `'a:bz2'` or `'a:xz'` is not possible. If *mode* is not suitable to open a certain (compressed) file for reading, **`ReadError`** is raised. Use *mode* `'r'` to avoid this. If a compression method is not supported, **`CompressionError`** is raised.

If *fileobj* is specified, it is used as an alternative to a file object opened in binary mode for *name*. It is supposed to be at position 0.

For modes `'w:gz'`, `'x:gz'`, `'w|gz'`, `'w:bz2'`, `'x:bz2'`, `'w|bz2'`, **`tarfile.open()`** accepts the keyword argument *compresslevel* (default `9`) to specify the compression level of the file.

For modes `'w:xz'`, `'x:xz'` and `'w|xz'`, **`tarfile.open()`** accepts the keyword argument *preset* to specify the compression level of the file.

For modes `'w:zst'`, `'x:zst'` and `'w|zst'`, **`tarfile.open()`** accepts the keyword argument *level* to specify the compression level of the file. The keyword argument *options* may also be passed, providing advanced Zstandard compression parameters described by **`CompressionParameter`**. The keyword argument *zstd_dict* can be passed to provide a **`ZstdDict`**, a Zstandard dictionary used to improve compression of smaller amounts of data.

For special purposes, there is a second format for *mode*: `'filemode|[compression]'`. **`tarfile.open()`** will return a **`TarFile`** object that processes its data as a stream of blocks. No random seeking will be done on the file. If given, *fileobj* may be any object that has a **`read()`** or **`write()`** method (depending on the *mode*) that works with bytes. *bufsize* specifies the blocksize and defaults to `20 * 512` bytes. Use this variant in combination with e.g. `sys.stdin.buffer`, a socket file object or a tape device. However, such a **`TarFile`** object is limited in that it does not allow random access, see Examples. The currently possible modes:

Mode	Action
`'r|*'`	Open a *stream* of tar blocks for reading with transparent compression.
`'r|'`	Open a *stream* of uncompressed tar blocks for reading.
`'r|gz'`	Open a gzip compressed *stream* for reading.
`'r|bz2'`	Open a bzip2 compressed *stream* for reading.
`'r|xz'`	Open an lzma compressed *stream* for reading.
`'r|zst'`	Open a Zstandard compressed *stream* for reading.
`'w|'`	Open an uncompressed *stream* for writing.
`'w|gz'`	Open a gzip compressed *stream* for writing.

Mode	Action
`'w|bz2'`	Open a bzip2 compressed *stream* for writing.
`'w|xz'`	Open an lzma compressed *stream* for writing.
`'w|zst'`	Open a Zstandard compressed *stream* for writing.

Changed in version 3.5: The `'x'` (exclusive creation) mode was added.

Changed in version 3.6: The *name* parameter accepts a path-like object.

Changed in version 3.12: The *compresslevel* keyword argument also works for streams.

Changed in version 3.14: The *preset* keyword argument also works for streams.

class tarfile.TarFile

Class for reading and writing tar archives. Do not use this class directly: use **`tarfile.open()`** instead. See TarFile Objects.

tarfile.is_tarfile(*name*)

Return **`True`** if *name* is a tar archive file, that the **`tarfile`** module can read. *name* may be a **`str`**, file, or file-like object.

Changed in version 3.9: Support for file and file-like objects.

The **`tarfile`** module defines the following exceptions:

exception tarfile.TarError

Base class for all `tarfile` exceptions.

exception tarfile.ReadError

Is raised when a tar archive is opened, that either cannot be handled by the `tarfile` module or is somehow invalid.

exception tarfile.CompressionError

Is raised when a compression method is not supported or when the data cannot be decoded properly.

exception tarfile.StreamError

Is raised for the limitations that are typical for stream-like `TarFile` objects.

exception tarfile.ExtractError

Is raised for *non-fatal* errors when using `TarFile.extract()`, but only if `TarFile.errorlevel`` == 2`.

exception tarfile.HeaderError

Is raised by `TarInfo.frombuf()` if the buffer it gets is invalid.

exception tarfile.FilterError

Base class for members refused by filters.

tarinfo

Information about the member that the filter refused to extract, as TarInfo.

exception tarfile.AbsolutePathError

Raised to refuse extracting a member with an absolute path.

exception tarfile.OutsideDestinationError

Raised to refuse extracting a member outside the destination directory.

exception tarfile.SpecialFileError

Raised to refuse extracting a special file (e.g. a device or pipe).

exception tarfile.AbsoluteLinkError

Raised to refuse extracting a symbolic link with an absolute path.

exception tarfile.LinkOutsideDestinationError

Raised to refuse extracting a symbolic link pointing outside the destination directory.

exception tarfile.LinkFallbackError

Raised to refuse emulating a link (hard or symbolic) by extracting another archive member, when that member would be rejected by the filter location. The exception that was raised to reject the replacement member is available as **`BaseException.__context__`**.

Added in version 3.14.

The following constants are available at the module level:

tarfile.ENCODING

The default character encoding: `'utf-8'` on Windows, the value returned by `sys.getfilesystemencoding()` otherwise.

tarfile.REGTYPE
tarfile.AREGTYPE

A regular file `type`.

tarfile.LNKTYPE

A link (inside tarfile) `type`.

tarfile.SYMTYPE

A symbolic link `type`.

tarfile.CHRTYPE
A character special device `type`.

tarfile.BLKTYPE
A block special device `type`.

tarfile.DIRTYPE
A directory `type`.

tarfile.FIFOTYPE
A FIFO special device `type`.

tarfile.CONTTYPE
A contiguous file `type`.

tarfile.GNUTYPE_LONGNAME
A GNU tar longname `type`.

tarfile.GNUTYPE_LONGLINK
A GNU tar longlink `type`.

tarfile.GNUTYPE_SPARSE
A GNU tar sparse file `type`.

Each of the following constants defines a tar archive format that the `tarfile` module is able to create. See section Supported tar formats for details.

tarfile.USTAR_FORMAT
POSIX.1-1988 (ustar) format.

tarfile.GNU_FORMAT
GNU tar format.

tarfile.PAX_FORMAT
POSIX.1-2001 (pax) format.

tarfile.DEFAULT_FORMAT

The default format for creating archives. This is currently `PAX_FORMAT`.

Changed in version 3.8: The default format for new archives was changed to `PAX_FORMAT` from `GNU_FORMAT`.

See also

Module `zipfile`

Documentation of the `zipfile` standard module.

Archiving operations

Documentation of the higher-level archiving facilities provided by the standard `shutil` module.

GNU tar manual, Basic Tar Format
[https://www.gnu.org/software/tar/manual/html_node/Standard.html]

Documentation for tar archive files, including GNU tar extensions.

TarFile Objects

The `TarFile` object provides an interface to a tar archive. A tar archive is a sequence of blocks. An archive member (a stored file) is made up of a header block followed by data blocks. It is possible to store a file in a tar archive several times. Each archive member is represented by a `TarInfo` object, see TarInfo Objects for details.

A `TarFile` object can be used as a context manager in a `with` statement. It will automatically be closed when the block is completed. Please note that in the event of an exception an archive opened for writing will not be finalized; only the internally used file object will be closed. See the Examples section for a use case.

Added in version 3.2: Added support for the context management protocol.

class tarfile.TarFile(*name=None*, *mode='r'*, *fileobj=None*, *format=DEFAULT_FORMAT*, *tarinfo=TarInfo*, *dereference=False*, *ignore_zeros=False*, *encoding=ENCODING*, *errors='surrogateescape'*, *pax_headers=None*, *debug=0*, *errorlevel=1*, *stream=False*)

All following arguments are optional and can be accessed as instance attributes as well.

name is the pathname of the archive. *name* may be a path-like object. It can be omitted if *fileobj* is given. In this case, the file object's **`name`** attribute is used if it exists.

mode is either `'r'` to read from an existing archive, `'a'` to append data to an existing file, `'w'` to create a new file overwriting an existing one, or `'x'` to create a new file only if it does not already exist.

If *fileobj* is given, it is used for reading or writing data. If it can be determined, *mode* is overridden by *fileobj*'s mode. *fileobj* will be used from position 0.

Note

fileobj is not closed, when **`TarFile`** is closed.

format controls the archive format for writing. It must be one of the constants **`USTAR_FORMAT`**, **`GNU_FORMAT`** or **`PAX_FORMAT`** that are defined at module level. When reading, format will be automatically detected, even if different formats are present in a single archive.

The *tarinfo* argument can be used to replace the default **`TarInfo`** class with a different one.

If *dereference* is **`False`**, add symbolic and hard links to the archive. If it is **`True`**, add the content of the target files to the archive. This has no effect on systems that do not support symbolic links.

If *ignore_zeros* is **`False`**, treat an empty block as the end of the archive. If it is **`True`**, skip empty (and invalid) blocks and try to get as many members as possible. This is only useful for reading concatenated or damaged archives.

debug can be set from `0` (no debug messages) up to `3` (all debug messages). The messages are written to `sys.stderr`.

errorlevel controls how extraction errors are handled, see **`the corresponding attribute`**.

The *encoding* and *errors* arguments define the character encoding to be used for reading or writing the archive and how conversion errors are going to be handled. The default settings will work for most users. See section Unicode issues for in-depth information.

The *pax_headers* argument is an optional dictionary of strings which will be added as a pax global header if *format* is **`PAX_FORMAT`**.

If *stream* is set to **`True`** then while reading the archive info about files in the archive are not cached, saving memory.

Changed in version 3.2: Use `'surrogateescape'` as the default for the *errors* argument.

Changed in version 3.5: The `'x'` (exclusive creation) mode was added.

Changed in version 3.6: The *name* parameter accepts a path-like object.

Changed in version 3.13: Add the *stream* parameter.

classmethod TarFile.open(*...*)

Alternative constructor. The **`tarfile.open()`** function is actually a shortcut to this classmethod.

TarFile.getmember(*name*)

Return a **`TarInfo`** object for member *name*. If *name* can not be found in the archive, **`KeyError`** is raised.

Note

If a member occurs more than once in the archive, its last occurrence is assumed to be the most up-to-date version.

TarFile.getmembers()

Return the members of the archive as a list of **`TarInfo`** objects. The list has the same order as the members in the archive.

TarFile.getnames()

Return the members as a list of their names. It has the same order as the list returned by **`getmembers()`**.

TarFile.list(*verbose=True*, ***, *members=None*)

Print a table of contents to `sys.stdout`. If *verbose* is **`False`**, only the names of the members are printed. If it is **`True`**, output similar to that of **ls -l** is produced. If optional *members* is given, it must be a subset of the list returned by `getmembers()`.

Changed in version 3.5: Added the *members* parameter.

TarFile.next()

Return the next member of the archive as a `TarInfo` object, when `TarFile` is opened for reading. Return `None` if there is no more available.

TarFile.extractall(*path='.'*, *members=None*, ***, *numeric_owner=False*, *filter=None*)

Extract all members from the archive to the current working directory or directory *path*. If optional *members* is given, it must be a subset of the list returned by `getmembers()`. Directory information like owner, modification time and permissions are set after all members have been extracted. This is done to work around two problems: A directory’s modification time is reset each time a file is created in it. And, if a directory’s permissions do not allow writing, extracting files to it will fail.

If *numeric_owner* is `True`, the uid and gid numbers from the tarfile are used to set the owner/group for the extracted files. Otherwise, the named values from the tarfile are used.

The *filter* argument specifies how `members` are modified or rejected before extraction. See Extraction filters for details. It is recommended to set this explicitly only if specific *tar* features are required, or as `filter='data'` to support Python versions with a less secure default (3.13 and lower).

Warning

Never extract archives from untrusted sources without prior inspection.

Since Python 3.14, the default (**`data`**) will prevent the most dangerous security issues. However, it will not prevent *all* unintended or insecure behavior. Read the Extraction filters section for details.

Changed in version 3.5: Added the *numeric_owner* parameter.

Changed in version 3.6: The *path* parameter accepts a path-like object.

Changed in version 3.12: Added the *filter* parameter.

Changed in version 3.14: The *filter* parameter now defaults to `'data'`.

TarFile.extract(*member*, *path=''*, *set_attrs=True*, ***, *numeric_owner=False*, *filter=None*)

Extract a member from the archive to the current working directory, using its full name. Its file information is extracted as accurately as possible. *member* may be a filename or a **`TarInfo`** object. You can specify a different directory using *path*. *path* may be a path-like object. File attributes (owner, mtime, mode) are set unless *set_attrs* is false.

The *numeric_owner* and *filter* arguments are the same as for **`extractall()`**.

Note

The **`extract()`** method does not take care of several extraction issues. In most cases you should consider using the `extractall()` method.

Warning

Never extract archives from untrusted sources without prior inspection. See the warning for `extractall()` for details.

Changed in version 3.2: Added the *set_attrs* parameter.

Changed in version 3.5: Added the *numeric_owner* parameter.

Changed in version 3.6: The *path* parameter accepts a path-like object.

Changed in version 3.12: Added the *filter* parameter.

TarFile.extractfile(*member*)

Extract a member from the archive as a file object. *member* may be a filename or a `TarInfo` object. If *member* is a regular file or a link, an `io.BufferedReader` object is returned. For all other existing members, `None` is returned. If *member* does not appear in the archive, `KeyError` is raised.

Changed in version 3.3: Return an `io.BufferedReader` object.

Changed in version 3.13: The returned `io.BufferedReader` object has the **`mode`** attribute which is always equal to `'rb'`.

TarFile.errorlevel*: int*

If *errorlevel* is `0`, errors are ignored when using `TarFile.extract()` and `TarFile.extractall()`. Nevertheless, they appear as error messages in the debug output when *debug* is greater than 0. If `1` (the default), all *fatal* errors are raised as **`OSError`** or **`FilterError`** exceptions. If `2`, all *non-fatal* errors are raised as **`TarError`** exceptions as well.

Some exceptions, e.g. ones caused by wrong argument types or data corruption, are always raised.

Custom extraction filters should raise **`FilterError`** for *fatal* errors and **`ExtractError`** for *non-fatal* ones.

Note that when an exception is raised, the archive may be partially extracted. It is the user's responsibility to clean up.

TarFile.extraction_filter

Added in version 3.12.

The extraction filter used as a default for the *filter* argument of **`extract()`** and **`extractall()`**.

The attribute may be `None` or a callable. String names are not allowed for this attribute, unlike the *filter* argument to **`extract()`**.

If `extraction_filter` is `None` (the default), extraction methods will use the **`data`** filter by default.

The attribute may be set on instances or overridden in subclasses. It also is possible to set it on the `TarFile` class itself to set a global default, although, since it affects all uses of *tarfile*, it is best practice to only do so in top-level applications or **`site configuration`**. To set a global default this way, a filter function needs to be wrapped in **`staticmethod()`** to prevent injection of a `self` argument.

Changed in version 3.14: The default filter is set to **`data`**, which disallows some dangerous features such as links to absolute paths or paths outside of the destination. Previously, the default was equivalent to **`fully_trusted`**.

TarFile.add(*name*, *arcname=None*, *recursive=True*, *, *filter=None*)

Add the file *name* to the archive. *name* may be any type of file (directory, fifo, symbolic link, etc.). If given, *arcname* specifies an alternative name for the file in the archive. Directories are added recursively by default. This can be avoided by setting *recursive* to **`False`**. Recursion adds entries in sorted order. If *filter* is given, it should be a function that takes a **`TarInfo`** object argument and returns the changed **`TarInfo`** object. If it instead returns **`None`** the **`TarInfo`** object will be excluded from the archive. See Examples for an example.

Changed in version 3.2: Added the *filter* parameter.

Changed in version 3.7: Recursion adds entries in sorted order.

TarFile.addfile(*tarinfo*, *fileobj=None*)

Add the **`TarInfo`** object *tarinfo* to the archive. If *tarinfo* represents a non zero-size regular file, the *fileobj* argument should be a binary file, and `tarinfo.size` bytes are read from it and added to the archive. You can create **`TarInfo`** objects directly, or by using **`gettarinfo()`**.

Changed in version 3.13: *fileobj* must be given for non-zero-sized regular files.

TarFile.gettarinfo(*name=None*, *arcname=None*, *fileobj=None*)

Create a **`TarInfo`** object from the result of **`os.stat()`** or equivalent on an existing file. The file is either named by *name*, or specified as a file object *fileobj* with a file descriptor. *name* may be a path-like object. If given, *arcname* specifies an alternative name for the file in the archive, otherwise, the name is taken from *fileobj*’s **`name`** attribute, or the *name* argument. The name should be a text string.

You can modify some of the **`TarInfo`**’s attributes before you add it using **`addfile()`**. If the file object is not an ordinary file object positioned at the beginning of the file, attributes such as **`size`** may need modifying. This is the case for objects such as **`GzipFile`**. The **`name`** may also be modified, in which case *arcname* could be a dummy string.

Changed in version 3.6: The *name* parameter accepts a path-like object.

TarFile.close()

Close the **`TarFile`**. In write mode, two finishing zero blocks are appended to the archive.

TarFile.pax_headers*: dict*

A dictionary containing key-value pairs of pax global headers.

TarInfo Objects

A **`TarInfo`** object represents one member in a **`TarFile`**. Aside from storing all required attributes of a file (like file type, size, time, permissions, owner etc.), it provides some useful methods to determine its type. It does *not* contain the file’s data itself.

`TarInfo` objects are returned by `TarFile`’s methods `getmember()`, `getmembers()` and `gettarinfo()`.

Modifying the objects returned by `getmember()` or `getmembers()` will affect all subsequent operations on the archive. For cases where this is unwanted, you can use `copy.copy()` or call the `replace()` method to create a modified copy in one step.

Several attributes can be set to `None` to indicate that a piece of metadata is unused or unknown. Different `TarInfo` methods handle `None` differently:

- The `extract()` or `extractall()` methods will ignore the corresponding metadata, leaving it set to a default.
- `addfile()` will fail.
- `list()` will print a placeholder string.

class tarfile.TarInfo(*name*='')

Create a `TarInfo` object.

classmethod TarInfo.frombuf(*buf*, *encoding*, *errors*)

Create and return a `TarInfo` object from string buffer *buf*.

Raises `HeaderError` if the buffer is invalid.

classmethod TarInfo.fromtarfile(*tarfile*)

Read the next member from the `TarFile` object *tarfile* and return it as a `TarInfo` object.

TarInfo.tobuf(*format=DEFAULT_FORMAT*, *encoding=ENCODING*, *errors='surrogateescape'*)

Create a string buffer from a `TarInfo` object. For information on the arguments see the constructor of the `TarFile` class.

Changed in version 3.2: Use `'surrogateescape'` as the default for the *errors* argument.

A `TarInfo` object has the following public data attributes:

TarInfo.name*: str*

Name of the archive member.

TarInfo.size*: int*

Size in bytes.

TarInfo.mtime*: int | float*

Time of last modification in seconds since the epoch, as in **`os.stat_result.st_mtime`**.

Changed in version 3.12: Can be set to `None` for **`extract()`** and **`extractall()`**, causing extraction to skip applying this attribute.

TarInfo.mode*: int*

Permission bits, as for **`os.chmod()`**.

Changed in version 3.12: Can be set to `None` for **`extract()`** and **`extractall()`**, causing extraction to skip applying this attribute.

TarInfo.type

File type. *type* is usually one of these constants: **`REGTYPE`**, **`AREGTYPE`**, **`LNKTYPE`**, **`SYMTYPE`**, **`DIRTYPE`**, **`FIFOTYPE`**, **`CONTTYPE`**, **`CHRTYPE`**, **`BLKTYPE`**, **`GNUTYPE_SPARSE`**. To determine the type of a **`TarInfo`** object more conveniently, use the `is*()` methods below.

TarInfo.linkname*: str*

Name of the target file name, which is only present in **`TarInfo`** objects of type **`LNKTYPE`** and **`SYMTYPE`**.

For symbolic links (`SYMTYPE`), the *linkname* is relative to the directory that contains the link. For hard links (`LNKTYPE`), the *linkname* is relative to the root of the archive.

TarInfo.uid*: int*

User ID of the user who originally stored this member.

Changed in version 3.12: Can be set to `None` for **`extract()`** and **`extractall()`**, causing extraction to skip applying this attribute.

TarInfo.gid*: int*

Group ID of the user who originally stored this member.

Changed in version 3.12: Can be set to `None` for **`extract()`** and **`extractall()`**, causing extraction to skip applying this attribute.

TarInfo.uname*: str*

User name.

Changed in version 3.12: Can be set to `None` for **`extract()`** and **`extractall()`**, causing extraction to skip applying this attribute.

TarInfo.gname*: str*

Group name.

Changed in version 3.12: Can be set to `None` for **`extract()`** and **`extractall()`**, causing extraction to skip applying this attribute.

TarInfo.chksum*: int*

Header checksum.

TarInfo.devmajor*: int*

Device major number.

TarInfo.devminor*: int*

Device minor number.

TarInfo.offset*: int*

The tar header starts here.

TarInfo.offset_data*: int*

The file’s data starts here.

TarInfo.sparse

Sparse member information.

TarInfo.pax_headers*: dict*

A dictionary containing key-value pairs of an associated pax extended header.

TarInfo.replace(*name=..., mtime=..., mode=..., linkname=..., uid=..., gid=..., uname=..., gname=..., deep=True*)

Added in version 3.12.

Return a *new* copy of the **`TarInfo`** object with the given attributes changed. For example, to return a `TarInfo` with the group name set to `'staff'`, use:

```
new_tarinfo = old_tarinfo.replace(gname='staff')
```

By default, a deep copy is made. If *deep* is false, the copy is shallow, i.e. `pax_headers` and any custom attributes are shared with the original `TarInfo` object.

A **`TarInfo`** object also provides some convenient query methods:

TarInfo.isfile()

Return `True` if the `TarInfo` object is a regular file.

TarInfo.isreg()

Same as `isfile()`.

TarInfo.isdir()

Return `True` if it is a directory.

TarInfo.issym()

Return `True` if it is a symbolic link.

TarInfo.islnk()

Return `True` if it is a hard link.

TarInfo.ischr()

Return `True` if it is a character device.

TarInfo.isblk()

Return `True` if it is a block device.

TarInfo.isfifo()

Return `True` if it is a FIFO.

TarInfo.isdev()

Return `True` if it is one of character device, block device or FIFO.

Extraction filters

Added in version 3.12.

The *tar* format is designed to capture all details of a UNIX-like filesystem, which makes it very powerful. Unfortunately, the features make it easy to create tar files that have unintended – and possibly malicious – effects when extracted. For example, extracting a tar file can overwrite arbitrary files in various ways (e.g. by using absolute paths, `..` path components, or symlinks that affect later members).

In most cases, the full functionality is not needed. Therefore, *tarfile* supports extraction filters: a mechanism to limit functionality, and thus mitigate some of the security issues.

Warning

None of the available filters blocks *all* dangerous archive features. Never extract archives from untrusted sources without prior inspection. See also Hints for further verification.

See also

PEP 706 **[https://peps.python.org/pep-0706/]**
Contains further motivation and rationale behind the design.

The *filter* argument to `TarFile.extract()` or `extractall()` can be:

- the string `'fully_trusted'`: Honor all metadata as specified in the archive. Should be used if the user trusts the archive completely, or implements their own complex verification.
- the string `'tar'`: Honor most *tar*-specific features (i.e. features of UNIX-like filesystems), but block features that are very likely to be surprising or malicious. See `tar_filter()` for details.

- the string `'data'`: Ignore or block most features specific to UNIX-like filesystems. Intended for extracting cross-platform data archives. See **`data_filter()`** for details.
- `None` (default): Use **`TarFile.extraction_filter`**.

  If that is also `None` (the default), the `'data'` filter will be used.

  > *Changed in version 3.14:* The default filter is set to **`data`**. Previously, the default was equivalent to **`fully_trusted`**.

- A callable which will be called for each extracted member with a TarInfo describing the member and the destination path to where the archive is extracted (i.e. the same path is used for all members):

  ```
  filter(member: TarInfo, path: str, /) -> TarInfo | None
  ```

  The callable is called just before each member is extracted, so it can take the current state of the disk into account. It can:

  - return a **`TarInfo`** object which will be used instead of the metadata in the archive, or
  - return `None`, in which case the member will be skipped, or
  - raise an exception to abort the operation or skip the member, depending on **`errorlevel`**. Note that when extraction is aborted, **`extractall()`** may leave the archive partially extracted. It does not attempt to clean up.

Default named filters

The pre-defined, named filters are available as functions, so they can be reused in custom filters:

tarfile.fully_trusted_filter(*member*, *path*)

Return *member* unchanged.

This implements the `'fully_trusted'` filter.

tarfile.tar_filter(*member*, *path*)

Implements the `'tar'` filter.

- Strip leading slashes (`/` and `os.sep`) from filenames.
- Refuse to extract files with absolute paths (in case the name is absolute even after stripping slashes, e.g. `C:/foo` on Windows). This raises `AbsolutePathError`.
- Refuse to extract files whose absolute path (after following symlinks) would end up outside the destination. This raises `OutsideDestinationError`.
- Clear high mode bits (setuid, setgid, sticky) and group/other write bits (`S_IWGRP` | `S_IWOTH`).

Return the modified `TarInfo` member.

tarfile.data_filter(*member*, *path*)

Implements the `'data'` filter. In addition to what `tar_filter` does:

- Normalize link targets (`TarInfo.linkname`) using `os.path.normpath()`. Note that this removes internal `..` components, which may change the meaning of the link if the path in **`TarInfo.linkname`** traverses symbolic links.

- Refuse to extract links (hard or soft) that link to absolute paths, or ones that link outside the destination.

  This raises `AbsoluteLinkError` or `LinkOutsideDestinationError`.

  Note that such files are refused even on platforms that do not support symbolic links.
- Refuse to extract device files (including pipes). This raises `SpecialFileError`.

- For regular files, including hard links:

  - Set the owner read and write permissions (`S_IRUSR` | `S_IWUSR`).
  - Remove the group & other executable permission (`S_IXGRP` | `S_IXOTH`) if the owner doesn't have it (`S_IXUSR`).

- For other files (directories), set `mode` to `None`, so that extraction methods skip applying permission bits.

- Set user and group info (`uid`, `gid`, `uname`, `gname`) to `None`, so that extraction methods skip setting it.

Return the modified `TarInfo` member.

Note that this filter does not block *all* dangerous archive features. See Hints for further verification for details.

Changed in version 3.14: Link targets are now normalized.

Filter errors

When a filter refuses to extract a file, it will raise an appropriate exception, a subclass of `FilterError`. This will abort the extraction if `TarFile.errorlevel` is 1 or more. With `errorlevel=0` the error will be logged and the member will be skipped, but extraction will continue.

Hints for further verification

Even with `filter='data'`, *tarfile* is not suited for extracting untrusted files without prior inspection. Among other issues, the pre-defined filters do not prevent denial-of-service attacks. Users should do additional checks.

Here is an incomplete list of things to consider:

- Extract to a `new temporary directory` to prevent e.g. exploiting pre-existing links, and to make it easier to clean up after a failed extraction.
- Disallow symbolic links if you do not need the functionality.
- When working with untrusted data, use external (e.g. OS-level) limits on disk, memory and CPU usage.
- Check filenames against an allow-list of characters (to filter out control characters, confusables, foreign path separators, and so on).
- Check that filenames have expected extensions (discouraging files that execute when you “click on them”, or extension-less files like Windows special device names).
- Limit the number of extracted files, total size of extracted data, filename length (including symlink length), and size of individual files.
- Check for files that would be shadowed on case-insensitive filesystems.

Also note that:

- Tar files may contain multiple versions of the same file. Later ones are expected to overwrite any earlier ones. This feature is crucial to allow updating tape archives, but can be abused maliciously.

- *tarfile* does not protect against issues with “live” data, e.g. an attacker tinkering with the destination (or source) directory while extraction (or archiving) is in progress.

Supporting older Python versions

Extraction filters were added to Python 3.12, but may be backported to older versions as security updates. To check whether the feature is available, use e.g. `hasattr(tarfile, 'data_filter')` rather than checking the Python version.

The following examples show how to support Python versions with and without the feature. Note that setting `extraction_filter` will affect any subsequent operations.

- Fully trusted archive:

  ```
  my_tarfile.extraction_filter = (lambda member, path:
  member)
  my_tarfile.extractall()
  ```

- Use the `'data'` filter if available, but revert to Python 3.11 behavior (`'fully_trusted'`) if this feature is not available:

  ```
  my_tarfile.extraction_filter = getattr(tarfile,
  'data_filter',
                                         (lambda member,
  path: member))
  my_tarfile.extractall()
  ```

- Use the `'data'` filter; *fail* if it is not available:

  ```
  my_tarfile.extractall(filter=tarfile.data_filter)
  ```

  or:

```
my_tarfile.extraction_filter = tarfile.data_filter
my_tarfile.extractall()
```

- Use the `'data'` filter; *warn* if it is not available:

  ```
  if hasattr(tarfile, 'data_filter'):
      my_tarfile.extractall(filter='data')
  else:
      # remove this when no longer needed
      warn_the_user('Extracting may be unsafe; consider
  updating Python')
      my_tarfile.extractall()
  ```

Stateful extraction filter example

While *tarfile*'s extraction methods take a simple *filter* callable, custom filters may be more complex objects with an internal state. It may be useful to write these as context managers, to be used like this:

```
with StatefulFilter() as filter_func:
    tar.extractall(path, filter=filter_func)
```

Such a filter can be written as, for example:

```
class StatefulFilter:
    def __init__(self):
        self.file_count = 0

    def __enter__(self):
        return self

    def __call__(self, member, path):
        self.file_count += 1
        return member

    def __exit__(self, *exc_info):
        print(f'{self.file_count} files extracted')
```

Command-Line Interface

Added in version 3.4.

The `tarfile` module provides a simple command-line interface to interact with tar archives.

If you want to create a new tar archive, specify its name after the `-c` option and then list the filename(s) that should be included:

```
$ python -m tarfile -c monty.tar  spam.txt eggs.txt
```

Passing a directory is also acceptable:

```
$ python -m tarfile -c monty.tar life-of-brian_1979/
```

If you want to extract a tar archive into the current directory, use the `-e` option:

```
$ python -m tarfile -e monty.tar
```

You can also extract a tar archive into a different directory by passing the directory’s name:

```
$ python -m tarfile -e monty.tar  other-dir/
```

For a list of the files in a tar archive, use the `-l` option:

```
$ python -m tarfile -l monty.tar
```

Command-line options

-l <tarfile>
--list <tarfile>

List files in a tarfile.

-c <tarfile> <source1> ... <sourceN>
--create <tarfile> <source1> ... <sourceN>

Create tarfile from source files.

-e <tarfile> [<output_dir>]
--extract <tarfile> [<output_dir>]

Extract tarfile into the current directory if *output_dir* is not specified.

-t <tarfile>
--test <tarfile>

Test whether the tarfile is valid or not.

-v, --verbose

Verbose output.

--filter <filtername>

Specifies the *filter* for `--extract`. See Extraction filters for details. Only string names are accepted (that is, `fully_trusted`, `tar`, and `data`).

Examples

Reading examples

How to extract an entire tar archive to the current working directory:

```
import tarfile
tar = tarfile.open("sample.tar.gz")
tar.extractall(filter='data')
tar.close()
```

How to extract a subset of a tar archive with **`TarFile.extractall()`** using a generator function instead of a list:

```
import os
import tarfile

def py_files(members):
    for tarinfo in members:
        if os.path.splitext(tarinfo.name)[1] == ".py":
            yield tarinfo

tar = tarfile.open("sample.tar.gz")
tar.extractall(members=py_files(tar))
tar.close()
```

How to read a gzip compressed tar archive and display some member information:

```
import tarfile
tar = tarfile.open("sample.tar.gz", "r:gz")
for tarinfo in tar:
    print(tarinfo.name, "is", tarinfo.size, "bytes in size and
is ", end="")
    if tarinfo.isreg():
        print("a regular file.")
    elif tarinfo.isdir():
        print("a directory.")
    else:
        print("something else.")
tar.close()
```

Writing examples

How to create an uncompressed tar archive from a list of filenames:

```
import tarfile
tar = tarfile.open("sample.tar", "w")
for name in ["foo", "bar", "quux"]:
    tar.add(name)
tar.close()
```

The same example using the `with` statement:

```
import tarfile
with tarfile.open("sample.tar", "w") as tar:
    for name in ["foo", "bar", "quux"]:
        tar.add(name)
```

How to create and write an archive to stdout using `sys.stdout.buffer` in the *fileobj* parameter in `TarFile.add()`:

```
import sys
import tarfile
with tarfile.open("sample.tar.gz", "w|gz",
fileobj=sys.stdout.buffer) as tar:
    for name in ["foo", "bar", "quux"]:
        tar.add(name)
```

How to create an archive and reset the user information using the *filter* parameter in `TarFile.add()`:

```
import tarfile
def reset(tarinfo):
    tarinfo.uid = tarinfo.gid = 0
    tarinfo.uname = tarinfo.gname = "root"
    return tarinfo
tar = tarfile.open("sample.tar.gz", "w:gz")
tar.add("foo", filter=reset)
tar.close()
```

Supported tar formats

There are three tar formats that can be created with the `tarfile` module:

- The POSIX.1-1988 ustar format (`USTAR_FORMAT`). It supports filenames up to a length of at best 256 characters and linknames up to 100 characters. The maximum file size is 8 GiB. This is an old and limited but widely supported format.

- The GNU tar format (`GNU_FORMAT`). It supports long filenames and linknames, files bigger than 8 GiB and sparse files. It is the de facto standard on GNU/Linux systems. `tarfile` fully supports the GNU tar extensions for long names, sparse file support is read-only.

- The POSIX.1-2001 pax format (`PAX_FORMAT`). It is the most flexible format with virtually no limits. It supports long filenames and linknames, large files and stores pathnames in a portable way. Modern tar implementations, including GNU tar, bsdtar/libarchive and star, fully support extended *pax* features; some old or unmaintained libraries may not, but should treat *pax* archives as if they were in the universally supported *ustar* format. It is the current default format for new archives.

  It extends the existing *ustar* format with extra headers for information that cannot be stored otherwise. There are two flavours of pax headers: Extended headers only affect the subsequent file header, global headers are valid for the complete archive and affect all following files. All the data in a pax header is encoded in *UTF-8* for portability reasons.

There are some more variants of the tar format which can be read, but not created:

- The ancient V7 format. This is the first tar format from Unix Seventh Edition, storing only regular files and directories. Names must not be longer than 100 characters, there is no user/group name information. Some archives have miscalculated header checksums in case of fields with non-ASCII characters.
- The SunOS tar extended format. This format is a variant of the POSIX.1-2001 pax format, but is not compatible.

Unicode issues

The tar format was originally conceived to make backups on tape drives with the main focus on preserving file system information. Nowadays tar archives are commonly used for file distribution and exchanging archives over networks. One problem of the original format (which is the basis of all other formats) is that there is no concept of supporting different character encodings. For example, an ordinary tar archive created on a *UTF-8* system cannot be read correctly on a *Latin-1* system if it contains non-*ASCII* characters. Textual metadata (like filenames, linknames, user/group names) will appear damaged. Unfortunately, there is no way to autodetect the encoding of an archive. The pax format was designed to solve this problem. It stores non-ASCII metadata using the universal character encoding *UTF-8*.

The details of character conversion in **`tarfile`** are controlled by the *encoding* and *errors* keyword arguments of the **`TarFile`** class.

encoding defines the character encoding to use for the metadata in the archive. The default value is **`sys.getfilesystemencoding()`** or `'ascii'` as a fallback. Depending on whether the archive is read or written, the metadata must be either decoded or encoded. If *encoding* is not set appropriately, this conversion may fail.

The *errors* argument defines how characters are treated that cannot be converted. Possible values are listed in section Error Handlers. The default scheme is `'surrogateescape'` which Python also uses for its file system calls, see File Names, Command Line Arguments, and Environment Variables.

For **`PAX_FORMAT`** archives (the default), *encoding* is generally not needed because all the metadata is stored using *UTF-8*. *encoding* is only used in the rare cases when binary pax headers are decoded or when strings with surrogate characters are stored.

File Formats

The modules described in this chapter parse various miscellaneous file formats that aren’t markup languages and are not related to e-mail.

- `csv` — CSV File Reading and Writing
    - Module Contents
    - Dialects and Formatting Parameters
    - Reader Objects
    - Writer Objects
    - Examples
- `configparser` — Configuration file parser
    - Quick Start
    - Supported Datatypes
    - Fallback Values
    - Supported INI File Structure
    - Unnamed Sections
    - Interpolation of values
    - Mapping Protocol Access
    - Customizing Parser Behaviour
    - Legacy API Examples
    - ConfigParser Objects
    - RawConfigParser Objects
    - Exceptions
- `tomllib` — Parse TOML files
    - Examples
    - Conversion Table
- `netrc` — netrc file processing
    - netrc Objects
- `plistlib` — Generate and parse Apple `.plist` files
    - Examples

`csv` — CSV File Reading and Writing

Source code: Lib/csv.py [https://github.com/python/cpython/tree/3.14/Lib/csv.py]

The so-called CSV (Comma Separated Values) format is the most common import and export format for spreadsheets and databases. CSV format was used for many years prior to attempts to describe the format in a standardized way in **RFC 4180** [https://datatracker.ietf.org/doc/html/rfc4180.html]. The lack of a well-defined standard means that subtle differences often exist in the data produced and consumed by different applications. These differences can make it annoying to process CSV files from multiple sources. Still, while the delimiters and quoting characters vary, the overall format is similar enough that it is possible to write a single module which can efficiently manipulate such data, hiding the details of reading and writing the data from the programmer.

The `csv` module implements classes to read and write tabular data in CSV format. It allows programmers to say, “write this data in the format preferred by Excel,” or “read data from this file which was generated by Excel,” without knowing the precise details of the CSV format used by Excel. Programmers can also describe the CSV formats understood by other applications or define their own special-purpose CSV formats.

The `csv` module’s `reader` and `writer` objects read and write sequences. Programmers can also read and write data in dictionary form using the `DictReader` and `DictWriter` classes.

See also

PEP 305 [https://peps.python.org/pep-0305/] - CSV File API
The Python Enhancement Proposal which proposed this addition to Python.

Module Contents

The `csv` module defines the following functions:

csv.reader(*csvfile*, */*, *dialect*='*excel*', ***fmtparams*)

Return a reader object that will process lines from the given *csvfile*. A csvfile must be an iterable of strings, each in the reader’s defined csv format. A csvfile is most commonly a file-like object or list. If *csvfile* is a file object, it should be opened with `newline=''`. [[1]] An optional *dialect* parameter can be given which is used to define a set of parameters specific to a particular CSV dialect. It may be an instance of a subclass of the `Dialect` class or one of the strings returned by the `list_dialects()` function. The other optional *fmtparams* keyword arguments can be given to override individual formatting parameters in the current dialect. For full details about the dialect and formatting parameters, see section Dialects and Formatting Parameters.

Each row read from the csv file is returned as a list of strings. No automatic data type conversion is performed unless the `QUOTE_NONNUMERIC` format option is specified (in which case unquoted fields are transformed into floats).

A short usage example:

```
>>> import csv
>>> with open('eggs.csv', newline='') as csvfile:
...     spamreader = csv.reader(csvfile, delimiter=' ',
quotechar='|')
...     for row in spamreader:
...         print(', '.join(row))
Spam, Spam, Spam, Spam, Spam, Baked Beans
Spam, Lovely Spam, Wonderful Spam
```

csv.writer(*csvfile*, /, *dialect*='*excel*', ***fmtparams*)

Return a writer object responsible for converting the user’s data into delimited strings on the given file-like object. *csvfile* can be any object with a **`write()`** method. If *csvfile* is a file object, it should be opened with `newline=''` [[1]]. An optional *dialect* parameter can be given which is used to define a set of parameters specific to a particular CSV dialect. It may be an instance of a subclass of the **`Dialect`** class or one of the strings returned by the **`list_dialects()`** function. The other optional *fmtparams* keyword arguments can be given to override individual formatting parameters in the current dialect. For full details about dialects and formatting parameters, see the Dialects and Formatting Parameters section. To make it as easy as possible to interface with modules which implement the DB API, the value **`None`** is written as the empty string. While this isn’t a reversible transformation, it makes it easier to dump SQL NULL data values to CSV files without preprocessing the data returned from a `cursor.fetch*` call. All other non-string data are stringified with **`str()`** before being written.

A short usage example:

```
import csv
with open('eggs.csv', 'w', newline='') as csvfile:
    spamwriter = csv.writer(csvfile, delimiter=' ',
                            quotechar='|',
quoting=csv.QUOTE_MINIMAL)
```

```
    spamwriter.writerow(['Spam'] * 5 + ['Baked Beans'])
    spamwriter.writerow(['Spam', 'Lovely Spam', 'Wonderful
Spam'])
```

csv.register_dialect(*name*, /, *dialect*='*excel*', **fmtparams)

Associate *dialect* with *name*. *name* must be a string. The dialect can be specified either by passing a sub-class of **`Dialect`**, or by *fmtparams* keyword arguments, or both, with keyword arguments overriding parameters of the dialect. For full details about dialects and formatting parameters, see section Dialects and Formatting Parameters.

csv.unregister_dialect(*name*)

Delete the dialect associated with *name* from the dialect registry. An **`Error`** is raised if *name* is not a registered dialect name.

csv.get_dialect(*name*)

Return the dialect associated with *name*. An **`Error`** is raised if *name* is not a registered dialect name. This function returns an immutable **`Dialect`**.

csv.list_dialects()

Return the names of all registered dialects.

csv.field_size_limit()
csv.field_size_limit(*new_limit*)

Returns the current maximum field size allowed by the parser. If *new_limit* is given, this becomes the new limit.

The **`csv`** module defines the following classes:

class csv.DictReader(*f*, *fieldnames*=*None*, *restkey*=*None*, *restval*=*None*, *dialect*='*excel*', *args, **kwds)

Create an object that operates like a regular reader but maps the information in each row to a `dict` whose keys are given by the optional *fieldnames* parameter.

The *fieldnames* parameter is a sequence. If *fieldnames* is omitted, the values in the first row of file *f* will be used as the fieldnames and will be omitted from the results. If *fieldnames* is provided, they will be used and the first row will be included in the results. Regardless of how the fieldnames are determined, the dictionary preserves their original ordering.

If a row has more fields than fieldnames, the remaining data is put in a list and stored with the fieldname specified by *restkey* (which defaults to `None`). If a non-blank row has fewer fields than fieldnames, the missing values are filled-in with the value of *restval* (which defaults to `None`).

All other optional or keyword arguments are passed to the underlying `reader` instance.

If the argument passed to *fieldnames* is an iterator, it will be coerced to a `list`.

Changed in version 3.6: Returned rows are now of type **`OrderedDict`**.

Changed in version 3.8: Returned rows are now of type `dict`.

A short usage example:

```
>>> import csv
>>> with open('names.csv', newline='') as csvfile:
...     reader = csv.DictReader(csvfile)
...     for row in reader:
...         print(row['first_name'], row['last_name'])
```

```
...
Eric Idle
John Cleese

>>> print(row)
{'first_name': 'John', 'last_name': 'Cleese'}
```

class csv.DictWriter(*f*, *fieldnames*, *restval*='', *extrasaction*='*raise*', *dialect*='*excel*', **args*, ***kwds*)

Create an object which operates like a regular writer but maps dictionaries onto output rows. The *fieldnames* parameter is a `sequence` of keys that identify the order in which values in the dictionary passed to the `writerow()` method are written to file *f*. The optional *restval* parameter specifies the value to be written if the dictionary is missing a key in *fieldnames*. If the dictionary passed to the `writerow()` method contains a key not found in *fieldnames*, the optional *extrasaction* parameter indicates what action to take. If it is set to `'raise'`, the default value, a `ValueError` is raised. If it is set to `'ignore'`, extra values in the dictionary are ignored. Any other optional or keyword arguments are passed to the underlying `writer` instance.

Note that unlike the `DictReader` class, the *fieldnames* parameter of the `DictWriter` class is not optional.

If the argument passed to *fieldnames* is an iterator, it will be coerced to a `list`.

A short usage example:

```
import csv

with open('names.csv', 'w', newline='') as csvfile:
    fieldnames = ['first_name', 'last_name']
    writer = csv.DictWriter(csvfile, fieldnames=fieldnames)
    writer.writeheader()
    writer.writerow({'first_name': 'Baked', 'last_name':
'Beans'})
    writer.writerow({'first_name': 'Lovely', 'last_name':
'Spam'})
    writer.writerow({'first_name': 'Wonderful', 'last_name':
'Spam'})
```

class csv.Dialect

The **`Dialect`** class is a container class whose attributes contain information for how to handle doublequotes, whitespace, delimiters, etc. Due to the lack of a strict CSV specification, different applications produce subtly different CSV data. **`Dialect`** instances define how **`reader`** and **`writer`** instances behave.

All available **`Dialect`** names are returned by **`list_dialects()`**, and they can be registered with specific **`reader`** and **`writer`** classes through their initializer (`__init__`) functions like this:

```
import csv

with open('students.csv', 'w', newline='') as csvfile:
    writer = csv.writer(csvfile, dialect='unix')
```

class csv.excel

The **`excel`** class defines the usual properties of an Excel-generated CSV file. It is registered with the dialect name `'excel'`.

class csv.excel_tab

The **`excel_tab`** class defines the usual properties of an Excel-generated TAB-delimited file. It is registered with the dialect name `'excel-tab'`.

class csv.unix_dialect

The `unix_dialect` class defines the usual properties of a CSV file generated on UNIX systems, i.e. using `'\n'` as line terminator and quoting all fields. It is registered with the dialect name `'unix'`.

Added in version 3.2.

class csv.Sniffer

The `Sniffer` class is used to deduce the format of a CSV file.

The `Sniffer` class provides two methods:

sniff(*sample*, *delimiters=None*)

Analyze the given *sample* and return a `Dialect` subclass reflecting the parameters found. If the optional *delimiters* parameter is given, it is interpreted as a string containing possible valid delimiter characters.

has_header(*sample*)

Analyze the sample text (presumed to be in CSV format) and return `True` if the first row appears to be a series of column headers. Inspecting each column, one of two key criteria will be considered to estimate if the sample contains a header:

- the second through n-th rows contain numeric values
- the second through n-th rows contain strings where at least one value’s length differs from that of the putative header of that column.

Twenty-one rows after the header are sampled; if more than half of the columns + rows meet the criteria, `True` is returned.

Note

This method is a rough heuristic and may produce both false positives and negatives.

An example for `Sniffer` use:

```
with open('example.csv', newline='') as csvfile:
    dialect = csv.Sniffer().sniff(csvfile.read(1024))
    csvfile.seek(0)
    reader = csv.reader(csvfile, dialect)
    # ... process CSV file contents here ...
```

The `csv` module defines the following constants:

csv.QUOTE_ALL

Instructs `writer` objects to quote all fields.

csv.QUOTE_MINIMAL

Instructs `writer` objects to only quote those fields which contain special characters such as *delimiter*, *quotechar*, `'\r'`, `'\n'` or any of the characters in *lineterminator*.

csv.QUOTE_NONNUMERIC

Instructs `writer` objects to quote all non-numeric fields.

Instructs `reader` objects to convert all non-quoted fields to type `float`.

Note

Some numeric types, such as `bool`, `Fraction`, or `IntEnum`, have a string representation that cannot be converted to `float`. They cannot be read in the `QUOTE_NONNUMERIC` and `QUOTE_STRINGS` modes.

csv.QUOTE_NONE

Instructs `writer` objects to never quote fields. When the current *delimiter*, *quotechar*, *escapechar*, `'\r'`, `'\n'` or any of the characters in *lineterminator* occurs in output data it is preceded by the current *escapechar* character. If *escapechar* is not set, the writer will raise `Error` if any characters that require escaping are encountered. Set *quotechar* to `None` to prevent its escaping.

Instructs `reader` objects to perform no special processing of quote characters.

csv.QUOTE_NOTNULL

Instructs `writer` objects to quote all fields which are not `None`. This is similar to `QUOTE_ALL`, except that if a field value is `None` an empty (unquoted) string is written.

Instructs `reader` objects to interpret an empty (unquoted) field as `None` and to otherwise behave as `QUOTE_ALL`.

Added in version 3.12.

csv.QUOTE_STRINGS

Instructs `writer` objects to always place quotes around fields which are strings. This is similar to `QUOTE_NONNUMERIC`, except that if a field value is `None` an empty (unquoted) string is written.

Instructs `reader` objects to interpret an empty (unquoted) string as `None` and to otherwise behave as `QUOTE_NONNUMERIC`.

Added in version 3.12.

The `csv` module defines the following exception:

exception csv.Error

Raised by any of the functions when an error is detected.

Dialects and Formatting Parameters

To make it easier to specify the format of input and output records, specific formatting parameters are grouped together into dialects. A dialect is a subclass of the `Dialect` class containing various attributes describing the format of the CSV file. When creating `reader` or `writer` objects, the programmer can specify a string or a subclass of the `Dialect` class as the dialect parameter. In addition to, or instead of, the *dialect* parameter, the programmer can also specify individual formatting parameters, which have the same names as the attributes defined below for the `Dialect` class.

Dialects support the following attributes:

Dialect.delimiter

A one-character string used to separate fields. It defaults to `','`.

Dialect.doublequote

Controls how instances of *quotechar* appearing inside a field should themselves be quoted. When `True`, the character is doubled. When `False`, the *escapechar* is used as a prefix to the *quotechar*. It defaults to `True`.

On output, if *doublequote* is `False` and no *escapechar* is set, `Error` is raised if a *quotechar* is found in a field.

Dialect.escapechar

A one-character string used by the writer to escape characters that require escaping:

- the *delimiter*, the *quotechar*, `'\r'`, `'\n'` and any of the characters in *lineterminator* are escaped if

quoting is set to `QUOTE_NONE`;
- the *quotechar* is escaped if *doublequote* is `False`;
- the *escapechar* itself.

On reading, the *escapechar* removes any special meaning from the following character. It defaults to `None`, which disables escaping.

Changed in version 3.11: An empty *escapechar* is not allowed.

Dialect.lineterminator

The string used to terminate lines produced by the `writer`. It defaults to `'\r\n'`.

Note

The `reader` is hard-coded to recognise either `'\r'` or `'\n'` as end-of-line, and ignores *lineterminator*. This behavior may change in the future.

Dialect.quotechar

A one-character string used to quote fields containing special characters, such as the *delimiter* or the *quotechar*, or which contain new-line characters (`'\r'`, `'\n'` or any of the characters in *lineterminator*). It defaults to `'"'`. Can be set to `None` to prevent escaping `'"'` if *quoting* is set to `QUOTE_NONE`.

Changed in version 3.11: An empty *quotechar* is not allowed.

Dialect.quoting

Controls when quotes should be generated by the writer and recognised by the reader. It can take on any of the

QUOTE_* constants and defaults to **`QUOTE_MINIMAL`** if *quotechar* is not `None`, and **`QUOTE_NONE`** otherwise.

Dialect.skipinitialspace

When **`True`**, spaces immediately following the *delimiter* are ignored. The default is **`False`**. When combining `delimiter=' '` with `skipinitialspace=True`, unquoted empty fields are not allowed.

Dialect.strict

When `True`, raise exception **`Error`** on bad CSV input. The default is `False`.

Reader Objects

Reader objects (**`DictReader`** instances and objects returned by the **`reader()`** function) have the following public methods:

csvreader.__next__()

Return the next row of the reader's iterable object as a list (if the object was returned from **`reader()`**) or a dict (if it is a **`DictReader`** instance), parsed according to the current **`Dialect`**. Usually you should call this as `next(reader)`.

Reader objects have the following public attributes:

csvreader.dialect

A read-only description of the dialect in use by the parser.

csvreader.line_num

The number of lines read from the source iterator. This is not the same as the number of records returned, as records can span multiple lines.

DictReader objects have the following public attribute:

DictReader.fieldnames

If not passed as a parameter when creating the object, this attribute is initialized upon first access or when the first record is read from the file.

Writer Objects

`writer` objects (`DictWriter` instances and objects returned by the `writer()` function) have the following public methods. A *row* must be an iterable of strings or numbers for `writer` objects and a dictionary mapping fieldnames to strings or numbers (by passing them through `str()` first) for `DictWriter` objects. Note that complex numbers are written out surrounded by parens. This may cause some problems for other programs which read CSV files (assuming they support complex numbers at all).

csvwriter.writerow(*row*, /)

Write the *row* parameter to the writer’s file object, formatted according to the current `Dialect`. Return the return value of the call to the *write* method of the underlying file object.

Changed in version 3.5: Added support of arbitrary iterables.

csvwriter.writerows(*rows*, /)

Write all elements in *rows* (an iterable of *row* objects as described above) to the writer’s file object, formatted according to the current dialect.

Writer objects have the following public attribute:

csvwriter.dialect

A read-only description of the dialect in use by the writer.

DictWriter objects have the following public method:

DictWriter.writeheader()

Write a row with the field names (as specified in the constructor) to the writer’s file object, formatted according to the current dialect. Return the return value of the `csvwriter.writerow()` call used internally.

Added in version 3.2.

Changed in version 3.8: `writeheader()` now also returns the value returned by the `csvwriter.writerow()` method it uses internally.

Examples

The simplest example of reading a CSV file:

```
import csv
with open('some.csv', newline='') as f:
    reader = csv.reader(f)
    for row in reader:
        print(row)
```

Reading a file with an alternate format:

```
import csv
with open('passwd', newline='') as f:
    reader = csv.reader(f, delimiter=':',
quoting=csv.QUOTE_NONE)
    for row in reader:
        print(row)
```

The corresponding simplest possible writing example is:

```
import csv
with open('some.csv', 'w', newline='') as f:
    writer = csv.writer(f)
    writer.writerows(someiterable)
```

Since **`open()`** is used to open a CSV file for reading, the file will by default be decoded into unicode using the system default encoding (see **`locale.getencoding()`**). To decode a file using a different encoding, use the `encoding` argument of open:

```
import csv
with open('some.csv', newline='', encoding='utf-8') as f:
    reader = csv.reader(f)
    for row in reader:
        print(row)
```

The same applies to writing in something other than the system default encoding: specify the encoding argument when opening the output file.

Registering a new dialect:

```
import csv
csv.register_dialect('unixpwd', delimiter=':',
quoting=csv.QUOTE_NONE)
with open('passwd', newline='') as f:
    reader = csv.reader(f, 'unixpwd')
```

A slightly more advanced use of the reader — catching and reporting errors:

```
import csv, sys
filename = 'some.csv'
with open(filename, newline='') as f:
    reader = csv.reader(f)
    try:
        for row in reader:
            print(row)
    except csv.Error as e:
        sys.exit(f'file {filename}, line {reader.line_num}:
{e}')
```

And while the module doesn’t directly support parsing strings, it can easily be done:

```
import csv
for row in csv.reader(['one,two,three']):
    print(row)
```

Footnotes

[1] (1,2)

If `newline=''` is not specified, newlines embedded inside quoted fields will not be interpreted correctly, and on platforms that use `\r\n` line endings on write an extra `\r` will be added. It should always be safe to specify `newline=''`, since the csv module does its own (universal) newline handling.

`configparser` — Configuration file parser

Source code: Lib/configparser.py [https://github.com/python/cpython/tree/3.14/Lib/configparser.py]

This module provides the `ConfigParser` class which implements a basic configuration language which provides a structure similar to what's found in Microsoft Windows INI files. You can use this to write Python programs which can be customized by end users easily.

Note

This library does *not* interpret or write the value-type prefixes used in the Windows Registry extended version of INI syntax.

See also

Module `tomllib`
TOML is a well-specified format for application configuration files. It is specifically designed to be an improved version of INI.

Module `shlex`
Support for creating Unix shell-like mini-languages which can also be used for application configuration files.

Module `json`

> The `json` module implements a subset of JavaScript syntax which is sometimes used for configuration, but does not support comments.

Quick Start

Let’s take a very basic configuration file that looks like this:

```
[DEFAULT]
ServerAliveInterval = 45
Compression = yes
CompressionLevel = 9
ForwardX11 = yes

[forge.example]
User = hg

[topsecret.server.example]
Port = 50022
ForwardX11 = no
```

The structure of INI files is described in the following section. Essentially, the file consists of sections, each of which contains keys with values. **`configparser`** classes can read and write such files. Let’s start by creating the above configuration file programmatically.

```
>>> import configparser
>>> config = configparser.ConfigParser()
>>> config['DEFAULT'] = {'ServerAliveInterval': '45',
...                      'Compression': 'yes',
...                      'CompressionLevel': '9'}
>>> config['forge.example'] = {}
>>> config['forge.example']['User'] = 'hg'
>>> config['topsecret.server.example'] = {}
>>> topsecret = config['topsecret.server.example']
>>> topsecret['Port'] = '50022'     # mutates the parser
>>> topsecret['ForwardX11'] = 'no'  # same here
>>> config['DEFAULT']['ForwardX11'] = 'yes'
>>> with open('example.ini', 'w') as configfile:
...   config.write(configfile)
...
```

As you can see, we can treat a config parser much like a dictionary. There are differences, outlined later, but the behavior is very close to what you would expect from a dictionary.

Now that we have created and saved a configuration file, let’s read it back and explore the data it holds.

```
>>> config = configparser.ConfigParser()
>>> config.sections()
[]
>>> config.read('example.ini')
['example.ini']
>>> config.sections()
['forge.example', 'topsecret.server.example']
>>> 'forge.example' in config
True
>>> 'python.org' in config
False
>>> config['forge.example']['User']
'hg'
>>> config['DEFAULT']['Compression']
'yes'
>>> topsecret = config['topsecret.server.example']
>>> topsecret['ForwardX11']
'no'
>>> topsecret['Port']
'50022'
>>> for key in config['forge.example']:
...     print(key)
user
compressionlevel
serveraliveinterval
compression
forwardx11
>>> config['forge.example']['ForwardX11']
'yes'
```

As we can see above, the API is pretty straightforward. The only bit of magic involves the `DEFAULT` section which provides default values for all other sections [[1]]. Note also that keys in sections are case-insensitive and stored in lowercase [[1]].

It is possible to read several configurations into a single **`ConfigParser`**, where the most recently added configuration has the highest priority. Any conflicting keys are taken from the more recent configuration while the previously existing keys are retained. The example below reads in an `override.ini` file, which will override any conflicting keys from the `example.ini` file.

```
[DEFAULT]
ServerAliveInterval = -1
```

```
>>> config_override = configparser.ConfigParser()
>>> config_override['DEFAULT'] = {'ServerAliveInterval': '-1'}
>>> with open('override.ini', 'w') as configfile:
...     config_override.write(configfile)
...
>>> config_override = configparser.ConfigParser()
>>> config_override.read(['example.ini', 'override.ini'])
['example.ini', 'override.ini']
>>> print(config_override.get('DEFAULT',
'ServerAliveInterval'))
-1
```

This behaviour is equivalent to a **`ConfigParser.read()`** call with several files passed to the *filenames* parameter.

Supported Datatypes

Config parsers do not guess datatypes of values in configuration files, always storing them internally as strings. This means that if you need other datatypes, you should convert on your own:

```
>>> int(topsecret['Port'])
50022
>>> float(topsecret['CompressionLevel'])
9.0
```

Since this task is so common, config parsers provide a range of handy getter methods to handle integers, floats and booleans. The last one is the most interesting because simply passing the value to `bool()` would do no good since `bool('False')` is still `True`. This is why config parsers also provide **`getboolean()`**. This method is case-insensitive and recognizes Boolean values from `'yes'`/`'no'`, `'on'`/`'off'`, `'true'`/`'false'` and `'1'`/`'0'` [[1]]. For example:

```
>>> topsecret.getboolean('ForwardX11')
False
>>> config['forge.example'].getboolean('ForwardX11')
True
>>> config.getboolean('forge.example', 'Compression')
True
```

Apart from **`getboolean()`**, config parsers also provide equivalent **`getint()`** and **`getfloat()`** methods. You can register your own converters and customize the provided ones. [[1]]

Fallback Values

As with a dictionary, you can use a section's **`get()`** method to provide fallback values:

```
>>> topsecret.get('Port')
'50022'
>>> topsecret.get('CompressionLevel')
'9'
>>> topsecret.get('Cipher')
>>> topsecret.get('Cipher', '3des-cbc')
'3des-cbc'
```

Please note that default values have precedence over fallback values. For instance, in our example the `'CompressionLevel'` key was specified only in the `'DEFAULT'` section. If we try to get it from the section `'topsecret.server.example'`, we will always get the default, even if we specify a fallback:

```
>>> topsecret.get('CompressionLevel', '3')
'9'
```

One more thing to be aware of is that the parser-level **`get()`** method provides a custom, more complex interface, maintained for backwards compatibility. When using this method, a fallback value can be provided via the `fallback` keyword-only argument:

```
>>> config.get('forge.example', 'monster',
...            fallback='No such things as monsters')
'No such things as monsters'
```

The same `fallback` argument can be used with the **`getint()`**, **`getfloat()`** and **`getboolean()`** methods, for example:

```
>>> 'BatchMode' in topsecret
False
>>> topsecret.getboolean('BatchMode', fallback=True)
True
>>> config['DEFAULT']['BatchMode'] = 'no'
>>> topsecret.getboolean('BatchMode', fallback=True)
False
```

Supported INI File Structure

A configuration file consists of sections, each led by a `[section]` header, followed by key/value entries separated by a specific string (`=` or `:` by default [[1]]). By default, section names are case sensitive but keys are not [[1]]. Leading and trailing whitespace is removed from keys and values. Values can be omitted if the parser is configured to allow it [[1]], in which case the key/value delimiter may also be left out. Values can also span multiple lines, as long as they are indented deeper than the first line of the value. Depending on the parser's mode, blank lines may be treated as parts of multiline values or ignored.

By default, a valid section name can be any string that does not contain '\n'. To change this, see **`ConfigParser.SECTCRE`**.

The first section name may be omitted if the parser is configured to allow an unnamed top level section with `allow_unnamed_section=True`. In this case, the keys/values may be retrieved by **`UNNAMED_SECTION`** as in `config[UNNAMED_SECTION]`.

Configuration files may include comments, prefixed by specific characters (`#` and `;` by default [[1]]). Comments may appear on their own on an otherwise empty line, possibly indented. [[1]]

For example:

```
[Simple Values]
key=value
spaces in keys=allowed
spaces in values=allowed as well
spaces around the delimiter = obviously
you can also use : to delimit keys from values

[All Values Are Strings]
values like this: 1000000
or this: 3.14159265359
are they treated as numbers? : no
integers, floats and booleans are held as: strings
can use the API to get converted values directly: true

[Multiline Values]
chorus: I'm a lumberjack, and I'm okay
    I sleep all night and I work all day
```

```
[No Values]
key_without_value
empty string value here =

[You can use comments]
# like this
; or this

# By default only in an empty line.
# Inline comments can be harmful because they prevent users
# from using the delimiting characters as parts of values.
# That being said, this can be customized.

    [Sections Can Be Indented]
        can_values_be_as_well = True
        does_that_mean_anything_special = False
        purpose = formatting for readability
        multiline_values = are
            handled just fine as
            long as they are indented
            deeper than the first line
            of a value
        # Did I mention we can indent comments, too?
```

Unnamed Sections

The name of the first section (or unique) may be omitted and values retrieved by the `UNNAMED_SECTION` attribute.

```
>>> config = """
... option = value
...
... [  Section 2  ]
... another = val
... """
>>> unnamed =
configparser.ConfigParser(allow_unnamed_section=True)
>>> unnamed.read_string(config)
>>> unnamed.get(configparser.UNNAMED_SECTION, 'option')
'value'
```

Interpolation of values

On top of the core functionality, **`ConfigParser`** supports interpolation. This means values can be preprocessed before returning them from `get()` calls.

class configparser.BasicInterpolation

The default implementation used by **`ConfigParser`**. It enables values to contain format strings which refer to other values in the same section, or values in the special default section [[1]]. Additional default values can be provided on initialization.

For example:

```
[Paths]
home_dir: /Users
my_dir: %(home_dir)s/lumberjack
my_pictures: %(my_dir)s/Pictures

[Escape]
# use a %% to escape the % sign (% is the only character 
that needs to be escaped):
gain: 80%%
```

In the example above, **`ConfigParser`** with *interpolation* set to `BasicInterpolation()` would resolve `%(home_dir)s` to the value of `home_dir` (`/Users` in this case). `%(my_dir)s` in effect would resolve to `/Users/lumberjack`. All interpolations are done on demand so keys used in the chain of references do not have to be specified in any specific order in the configuration file.

With `interpolation` set to `None`, the parser would simply return `%(my_dir)s/Pictures` as the value of `my_pictures` and `%(home_dir)s/lumberjack` as the value of `my_dir`.

class configparser.ExtendedInterpolation

An alternative handler for interpolation which implements a more advanced syntax, used for instance in `zc.buildout`. Extended interpolation is using `${section:option}` to denote a value from a foreign section. Interpolation can span multiple levels. For convenience, if the `section:` part is omitted, interpolation defaults to the current section (and possibly the default values from the special section).

For example, the configuration specified above with basic interpolation, would look like this with extended interpolation:

```
[Paths]
home_dir: /Users
my_dir: ${home_dir}/lumberjack
my_pictures: ${my_dir}/Pictures

[Escape]
# use a $$ to escape the $ sign ($ is the only character
that needs to be escaped):
cost: $$80
```

Values from other sections can be fetched as well:

```
[Common]
home_dir: /Users
library_dir: /Library
system_dir: /System
macports_dir: /opt/local

[Frameworks]
Python: 3.2
path: ${Common:system_dir}/Library/Frameworks/

[Arthur]
nickname: Two Sheds
last_name: Jackson
my_dir: ${Common:home_dir}/twosheds
my_pictures: ${my_dir}/Pictures
```

```
python_dir:
${Frameworks:path}/Python/Versions/${Frameworks:Python}
```

Mapping Protocol Access

Added in version 3.2.

Mapping protocol access is a generic name for functionality that enables using custom objects as if they were dictionaries. In case of **`configparser`**, the mapping interface implementation is using the `parser['section']['option']` notation.

`parser['section']` in particular returns a proxy for the section’s data in the parser. This means that the values are not copied but they are taken from the original parser on demand. What’s even more important is that when values are changed on a section proxy, they are actually mutated in the original parser.

`configparser` objects behave as close to actual dictionaries as possible. The mapping interface is complete and adheres to the **`MutableMapping`** ABC. However, there are a few differences that should be taken into account:

- By default, all keys in sections are accessible in a case-insensitive manner [[1]]. E.g. `for option in parser["section"]` yields only `optionxform`’ed option key names. This means lowercased keys by default. At the same time, for a section that holds the key `'a'`, both expressions return `True`:

  ```
  "a" in parser["section"]
  "A" in parser["section"]
  ```

- All sections include `DEFAULTSECT` values as well which means that `.clear()` on a section may not leave the section visibly empty. This is because default values cannot be deleted from the section (because technically they are not there). If they are overridden in the section, deleting causes the default value to be visible again. Trying to delete a default value causes a `KeyError`.

- `DEFAULTSECT` cannot be removed from the parser:
    - trying to delete it raises `ValueError`,
    - `parser.clear()` leaves it intact,
    - `parser.popitem()` never returns it.
- `parser.get(section, option, **kwargs)` - the second argument is **not** a fallback value. Note however that the section-level `get()` methods are compatible both with the mapping protocol and the classic configparser API.
- `parser.items()` is compatible with the mapping protocol (returns a list of *section_name*, *section_proxy* pairs including the DEFAULTSECT). However, this method can also be invoked with arguments: `parser.items(section, raw, vars)`. The latter call returns a list of *option*, *value* pairs for a specified `section`, with all interpolations expanded (unless `raw=True` is provided).

The mapping protocol is implemented on top of the existing legacy API so that subclasses overriding the original interface still should have mappings working as expected.

Customizing Parser Behaviour

There are nearly as many INI format variants as there are applications using it. `configparser` goes a long way to provide support for the largest sensible set of INI styles available. The default functionality is mainly dictated by historical background and it’s very likely that you will want to customize some of the features.

The most common way to change the way a specific config parser works is to use the **`__init__()`** options:

- *defaults*, default value: `None`

  This option accepts a dictionary of key-value pairs which will be initially put in the `DEFAULT` section. This makes for an elegant way to support concise configuration files that don’t specify values which are the same as the documented default.

  Hint: if you want to specify default values for a specific section, use **`read_dict()`** before you read the actual file.

- *dict_type*, default value: **`dict`**

  This option has a major impact on how the mapping protocol will behave and how the written configuration files look. With the standard dictionary, every section is stored in the order they were added to the parser. Same goes for options within sections.

  An alternative dictionary type can be used for example to sort sections and options on write-back.

  Please note: there are ways to add a set of key-value pairs in a single operation. When you use a regular dictionary in those operations, the order of the keys will be ordered. For example:

  ```
  >>> parser = configparser.ConfigParser()
  >>> parser.read_dict({'section1': {'key1': 'value1',
  ...                                'key2': 'value2',
  ...                                'key3': 'value3'},
  ...                   'section2': {'keyA': 'valueA',
  ```

```
...                                 'keyB': 'valueB',
...                                 'keyC': 'valueC'},
...                    'section3': {'foo': 'x',
...                                 'bar': 'y',
...                                 'baz': 'z'}
... })
>>> parser.sections()
['section1', 'section2', 'section3']
>>> [option for option in parser['section3']]
['foo', 'bar', 'baz']
```

- *allow_no_value*, default value: `False`

  Some configuration files are known to include settings without values, but which otherwise conform to the syntax supported by `configparser`. The *allow_no_value* parameter to the constructor can be used to indicate that such values should be accepted:

```
>>> import configparser

>>> sample_config = """
... [mysqld]
...   user = mysql
...   pid-file = /var/run/mysqld/mysqld.pid
...   skip-external-locking
...   old_passwords = 1
...   skip-bdb
...   # we don't need ACID today
...   skip-innodb
... """
>>> config = configparser.ConfigParser(allow_no_value=True)
>>> config.read_string(sample_config)

>>> # Settings with values are treated as before:
>>> config["mysqld"]["user"]
'mysql'

>>> # Settings without values provide None:
>>> config["mysqld"]["skip-bdb"]

>>> # Settings which aren't specified still raise an error:
```

```
>>> config["mysqld"]["does-not-exist"]
Traceback (most recent call last):
  ...
KeyError: 'does-not-exist'
```

- *delimiters*, default value: `('=', ':')`

  Delimiters are substrings that delimit keys from values within a section. The first occurrence of a delimiting substring on a line is considered a delimiter. This means values (but not keys) can contain the delimiters.

  See also the *space_around_delimiters* argument to **`ConfigParser.write()`**.

- *comment_prefixes*, default value: `('#', ';')`
- *inline_comment_prefixes*, default value: `None`

  Comment prefixes are strings that indicate the start of a valid comment within a config file. *comment_prefixes* are used only on otherwise empty lines (optionally indented) whereas *inline_comment_prefixes* can be used after every valid value (e.g. section names, options and empty lines as well). By default inline comments are disabled and `'#'` and `';'` are used as prefixes for whole line comments.

  Changed in version 3.2: In previous versions of **`configparser`** behaviour matched `comment_prefixes=('#',';')` and `inline_comment_prefixes=(';',)`.

  Please note that config parsers don’t support escaping of comment prefixes so using *inline_comment_prefixes* may prevent users from specifying option values with characters used as comment prefixes. When in doubt, avoid setting *inline_comment_prefixes*. In any circumstances, the only way of storing comment prefix characters at the beginning of a line in multiline values is to interpolate the prefix, for example:

```
>>> from configparser import ConfigParser,
ExtendedInterpolation
>>> parser =
ConfigParser(interpolation=ExtendedInterpolation())
>>> # the default BasicInterpolation could be used as well
>>> parser.read_string("""
... [DEFAULT]
... hash = #
...
... [hashes]
... shebang =
...   ${hash}!/usr/bin/env python
...   ${hash} -*- coding: utf-8 -*-
...
... extensions =
...   enabled_extension
...   another_extension
...   #disabled_by_comment
...   yet_another_extension
...
... interpolation not necessary = if # is not at line start
... even in multiline values = line #1
...   line #2
...   line #3
... """)
>>> print(parser['hashes']['shebang'])

#!/usr/bin/env python
# -*- coding: utf-8 -*-
>>> print(parser['hashes']['extensions'])

enabled_extension
another_extension
yet_another_extension
>>> print(parser['hashes']['interpolation not necessary'])
if # is not at line start
>>> print(parser['hashes']['even in multiline values'])
line #1
line #2
line #3
```

- *strict*, default value: `True`

  When set to `True`, the parser will not allow for any section or option duplicates while reading from a single source (using [`read_file()`](), [`read_string()`]() or [`read_dict()`]()). It is recommended to use strict parsers in new applications.

  Changed in version 3.2: In previous versions of [`configparser`]() behaviour matched `strict=False`.

- *empty_lines_in_values*, default value: `True`

  In config parsers, values can span multiple lines as long as they are indented more than the key that holds them. By default parsers also let empty lines to be parts of values. At the same time, keys can be arbitrarily indented themselves to improve readability. In consequence, when configuration files get big and complex, it is easy for the user to lose track of the file structure. Take for instance:

  ```
  [Section]
  key = multiline
    value with a gotcha

   this = is still a part of the multiline value of 'key'
  ```

  This can be especially problematic for the user to see if she’s using a proportional font to edit the file. That is why when your application does not need values with empty lines, you should consider disallowing them. This will make empty lines split keys every time. In the example above, it would produce two keys, `key` and `this`.

- *default_section*, default value: `configparser.DEFAULTSECT` (that is: `"DEFAULT"`)

  The convention of allowing a special section of default values for other sections or interpolation purposes is a powerful concept of this library, letting users create complex declarative configurations. This section is normally called `"DEFAULT"` but this can be customized to point to any other valid section name. Some typical values include: `"general"` or `"common"`. The name provided is used for recognizing default sections when reading from any source and is used when writing configuration back to a file. Its current value can be retrieved using the `parser_instance.default_section` attribute and may be modified at runtime (i.e. to convert files from one format to another).

- *interpolation*, default value: `configparser.BasicInterpolation`

  Interpolation behaviour may be customized by providing a custom handler through the *interpolation* argument. `None` can be used to turn off interpolation completely, `ExtendedInterpolation()` provides a more advanced variant inspired by `zc.buildout`. More on the subject in the dedicated documentation section. **`RawConfigParser`** has a default value of `None`.

- *converters*, default value: not set

  Config parsers provide option value getters that perform type conversion. By default **`getint()`**, **`getfloat()`**, and **`getboolean()`** are implemented. Should other getters be desirable, users may define them in a subclass or pass a dictionary where each key is a name of the converter and each value is a callable implementing said conversion. For instance, passing `{'decimal': decimal.Decimal}` would add **`getdecimal()`** on both the parser object and all section proxies. In other words, it will be possible to write both `parser_instance.getdecimal('section', 'key', fallback=0)` and `parser_instance['section'].getdecimal('key', 0)`.

If the converter needs to access the state of the parser, it can be implemented as a method on a config parser subclass. If the name of this method starts with `get`, it will be available on all section proxies, in the dict-compatible form (see the `getdecimal()` example above).

More advanced customization may be achieved by overriding default values of these parser attributes. The defaults are defined on the classes, so they may be overridden by subclasses or by attribute assignment.

ConfigParser.BOOLEAN_STATES

By default when using **`getboolean()`**, config parsers consider the following values `True`: `'1'`, `'yes'`, `'true'`, `'on'` and the following values `False`: `'0'`, `'no'`, `'false'`, `'off'`. You can override this by specifying a custom dictionary of strings and their Boolean outcomes. For example:

```
>>> custom = configparser.ConfigParser()
>>> custom['section1'] = {'funky': 'nope'}
>>> custom['section1'].getboolean('funky')
Traceback (most recent call last):
...
ValueError: Not a boolean: nope
>>> custom.BOOLEAN_STATES = {'sure': True, 'nope': False}
>>> custom['section1'].getboolean('funky')
False
```

Other typical Boolean pairs include `accept`/`reject` or `enabled`/`disabled`.

ConfigParser.optionxform(*option*)

This method transforms option names on every read, get, or set operation. The default converts the name to lowercase. This also means that when a configuration file gets written, all keys will be lowercase. Override this method if that’s unsuitable. For example:

```
>>> config = """
... [Section1]
... Key = Value
...
... [Section2]
... AnotherKey = Value
... """
>>> typical = configparser.ConfigParser()
>>> typical.read_string(config)
>>> list(typical['Section1'].keys())
['key']
>>> list(typical['Section2'].keys())
['anotherkey']
>>> custom = configparser.RawConfigParser()
>>> custom.optionxform = lambda option: option
>>> custom.read_string(config)
>>> list(custom['Section1'].keys())
['Key']
>>> list(custom['Section2'].keys())
['AnotherKey']
```

Note

The optionxform function transforms option names to a canonical form. This should be an idempotent function: if the name is already in canonical form, it should be returned unchanged.

ConfigParser.SECTCRE

A compiled regular expression used to parse section headers. The default matches `[section]` to the name `"section"`. Whitespace is considered part of the section name, thus `[  larch  ]` will be read as a section of name `"  larch  "`. Override this attribute if that’s unsuitable. For example:

```
>>> import re
>>> config = """
... [Section 1]
... option = value
...
... [  Section 2  ]
... another = val
... """
>>> typical = configparser.ConfigParser()
>>> typical.read_string(config)
>>> typical.sections()
['Section 1', '  Section 2  ']
>>> custom = configparser.ConfigParser()
>>> custom.SECTCRE = re.compile(r"\[ *(?P<header>[^]]+?)
*\]")
>>> custom.read_string(config)
>>> custom.sections()
['Section 1', 'Section 2']
```

Note

While ConfigParser objects also use an `OPTCRE` attribute for recognizing option lines, it’s not recommended to override it because that would interfere with constructor options *allow_no_value* and *delimiters*.

Legacy API Examples

Mainly because of backwards compatibility concerns, **`configparser`** provides also a legacy API with explicit `get`/`set` methods. While there are valid use cases for the methods outlined below, mapping protocol access is preferred for new projects. The legacy API is at times more advanced, low-level and downright counterintuitive.

An example of writing to a configuration file:

```
import configparser

config = configparser.RawConfigParser()

# Please note that using RawConfigParser's set functions, you
can assign
# non-string values to keys internally, but will receive an
error when
# attempting to write to a file or when you get it in non-raw
mode. Setting
# values using the mapping protocol or ConfigParser's set()
does not allow
# such assignments to take place.
config.add_section('Section1')
config.set('Section1', 'an_int', '15')
config.set('Section1', 'a_bool', 'true')
config.set('Section1', 'a_float', '3.1415')
config.set('Section1', 'baz', 'fun')
config.set('Section1', 'bar', 'Python')
config.set('Section1', 'foo', '%(bar)s is %(baz)s!')

# Writing our configuration file to 'example.cfg'
with open('example.cfg', 'w') as configfile:
    config.write(configfile)
```

An example of reading the configuration file again:

```
import configparser

config = configparser.RawConfigParser()
config.read('example.cfg')

# getfloat() raises an exception if the value is not a float
# getint() and getboolean() also do this for their respective
types
a_float = config.getfloat('Section1', 'a_float')
an_int = config.getint('Section1', 'an_int')
print(a_float + an_int)

# Notice that the next output does not interpolate '%(bar)s' or
```

```
'%(baz)s'.
# This is because we are using a RawConfigParser().
if config.getboolean('Section1', 'a_bool'):
    print(config.get('Section1', 'foo'))
```

To get interpolation, use [`ConfigParser`]:

```
import configparser

cfg = configparser.ConfigParser()
cfg.read('example.cfg')

# Set the optional *raw* argument of get() to True if you wish
to disable
# interpolation in a single get operation.
print(cfg.get('Section1', 'foo', raw=False))  # -> "Python is
fun!"
print(cfg.get('Section1', 'foo', raw=True))   # -> "%(bar)s is
%(baz)s!"

# The optional *vars* argument is a dict with members that will
take
# precedence in interpolation.
print(cfg.get('Section1', 'foo', vars={'bar': 'Documentation',
                                       'baz': 'evil'}))

# The optional *fallback* argument can be used to provide a
fallback value
print(cfg.get('Section1', 'foo'))
      # -> "Python is fun!"

print(cfg.get('Section1', 'foo', fallback='Monty is not.'))
      # -> "Python is fun!"

print(cfg.get('Section1', 'monster', fallback='No such things
as monsters.'))
      # -> "No such things as monsters."

# A bare print(cfg.get('Section1', 'monster')) would raise
NoOptionError
# but we can also use:
```

```
print(cfg.get('Section1', 'monster', fallback=None))
      # -> None
```

Default values are available in both types of ConfigParsers. They are used in interpolation if an option used is not defined elsewhere.

```
import configparser

# New instance with 'bar' and 'baz' defaulting to 'Life' and
'hard' each
config = configparser.ConfigParser({'bar': 'Life', 'baz':
'hard'})
config.read('example.cfg')

print(config.get('Section1', 'foo'))     # -> "Python is fun!"
config.remove_option('Section1', 'bar')
config.remove_option('Section1', 'baz')
print(config.get('Section1', 'foo'))     # -> "Life is hard!"
```

ConfigParser Objects

class configparser.ConfigParser(*defaults=None*, *dict_type=dict*, *allow_no_value=False*, ***, *delimiters=('=', ':')*, *comment_prefixes=('#', ';')*, *inline_comment_prefixes=None*, *strict=True*, *empty_lines_in_values=True*, *default_section=configparser.DEFAULTSECT*, *interpolation=BasicInterpolation()*, *converters={}*, *allow_unnamed_section=False*)

> The main configuration parser. When *defaults* is given, it is initialized into the dictionary of intrinsic defaults. When *dict_type* is given, it will be used to create the dictionary objects for the list of sections, for the options within a section, and for the default values.
>
> When *delimiters* is given, it is used as the set of substrings that divide keys from values. When

comment_prefixes is given, it will be used as the set of substrings that prefix comments in otherwise empty lines. Comments can be indented. When *inline_comment_prefixes* is given, it will be used as the set of substrings that prefix comments in non-empty lines.

When *strict* is `True` (the default), the parser won't allow for any section or option duplicates while reading from a single source (file, string or dictionary), raising **`DuplicateSectionError`** or **`DuplicateOptionError`**. When *empty_lines_in_values* is `False` (default: `True`), each empty line marks the end of an option. Otherwise, internal empty lines of a multiline option are kept as part of the value. When *allow_no_value* is `True` (default: `False`), options without values are accepted; the value held for these is `None` and they are serialized without the trailing delimiter.

When *default_section* is given, it specifies the name for the special section holding default values for other sections and interpolation purposes (normally named `"DEFAULT"`). This value can be retrieved and changed at runtime using the `default_section` instance attribute. This won't re-evaluate an already parsed config file, but will be used when writing parsed settings to a new config file.

Interpolation behaviour may be customized by providing a custom handler through the *interpolation* argument. `None` can be used to turn off interpolation completely, `ExtendedInterpolation()` provides a more advanced variant inspired by `zc.buildout`. More on the subject in the dedicated documentation section.

All option names used in interpolation will be passed through the **`optionxform()`** method just like any other option name reference. For example, using the default implementation of **`optionxform()`** (which converts option names to lower case), the values `foo %(bar)s` and `foo %(BAR)s` are equivalent.

When *converters* is given, it should be a dictionary where each key represents the name of a type converter and each value is a callable implementing the conversion from string to the desired datatype. Every converter gets its own corresponding **`get*()`** method on the parser object and section proxies.

When *allow_unnamed_section* is `True` (default: `False`), the first section name can be omitted. See the “Unnamed Sections” section.

It is possible to read several configurations into a single **`ConfigParser`**, where the most recently added configuration has the highest priority. Any conflicting keys are taken from the more recent configuration while the previously existing keys are retained. The example below reads in an `override.ini` file, which will override any conflicting keys from the `example.ini` file.

```
[DEFAULT]
ServerAliveInterval = -1
```

```
>>> config_override = configparser.ConfigParser()
>>> config_override['DEFAULT'] = {'ServerAliveInterval':
'-1'}
>>> with open('override.ini', 'w') as configfile:
...     config_override.write(configfile)
...
>>> config_override = configparser.ConfigParser()
>>> config_override.read(['example.ini', 'override.ini'])
['example.ini', 'override.ini']
>>> print(config_override.get('DEFAULT',
'ServerAliveInterval'))
-1
```

Changed in version 3.1: The default *dict_type* is `collections.OrderedDict`.

Changed in version 3.2: *allow_no_value*, *delimiters*, *comment_prefixes*, *strict*, *empty_lines_in_values*, *default_section* and *interpolation* were added.

Changed in version 3.5: The *converters* argument was added.

Changed in version 3.7: The *defaults* argument is read with `read_dict()`, providing consistent behavior across the parser: non-string keys and values are implicitly converted to strings.

Changed in version 3.8: The default *dict_type* is `dict`, since it now preserves insertion order.

Changed in version 3.13: Raise a `MultilineContinuationError` when *allow_no_value* is `True`, and a key without a value is continued with an indented line.

Changed in version 3.13: The *allow_unnamed_section* argument was added.

defaults()

Return a dictionary containing the instance-wide defaults.

sections()

Return a list of the sections available; the *default section* is not included in the list.

add_section(*section*)

Add a section named *section* to the instance. If a section by the given name already exists, `DuplicateSectionError` is raised. If the *default section* name is passed, `ValueError` is raised. The name of the section must be a string; if not, `TypeError` is raised.

Changed in version 3.2: Non-string section names raise `TypeError`.

has_section(*section*)

Indicates whether the named *section* is present in the configuration. The *default section* is not acknowledged.

options(*section*)

Return a list of options available in the specified *section*.

has_option(*section*, *option*)

If the given *section* exists, and contains the given *option*, return `True`; otherwise return `False`. If the specified *section* is `None` or an empty string, DEFAULT is assumed.

read(*filenames*, *encoding=None*)

Attempt to read and parse an iterable of filenames, returning a list of filenames which were successfully parsed.

If *filenames* is a string, a `bytes` object or a path-like object, it is treated as a single filename. If a file named in *filenames* cannot be opened, that file will be ignored. This is designed so that you can specify an iterable of potential configuration file locations (for example, the current directory, the user's home directory, and some system-wide directory), and all existing configuration files in the iterable will be read.

If none of the named files exist, the `ConfigParser` instance will contain an empty dataset. An application which requires initial values to be loaded from a file should load the required file or files using `read_file()` before calling `read()` for any optional files:

```
import configparser, os

config = configparser.ConfigParser()
config.read_file(open('defaults.cfg'))
config.read(['site.cfg',
os.path.expanduser('~/.myapp.cfg')],
            encoding='cp1250')
```

Changed in version 3.2: Added the *encoding* parameter. Previously, all files were read using the default encoding for `open()`.

Changed in version 3.6.1: The *filenames* parameter accepts a path-like object.

Changed in version 3.7: The *filenames* parameter accepts a `bytes` object.

read_file(*f*, *source=None*)

Read and parse configuration data from *f* which must be an iterable yielding Unicode strings (for example files opened in text mode).

Optional argument *source* specifies the name of the file being read. If not given and *f* has a **`name`** attribute, that is used for *source*; the default is `'<???>'`.

Added in version 3.2: Replaces **`readfp()`**.

read_string(*string*, *source*='<*string*>')

Parse configuration data from a string.

Optional argument *source* specifies a context-specific name of the string passed. If not given, `'<string>'` is used. This should commonly be a filesystem path or a URL.

Added in version 3.2.

read_dict(*dictionary*, *source*='<*dict*>')

Load configuration from any object that provides a dict-like `items()` method. Keys are section names, values are dictionaries with keys and values that should be present in the section. If the used dictionary type preserves order, sections and their keys will be added in order. Values are automatically converted to strings.

Optional argument *source* specifies a context-specific name of the dictionary passed. If not given, `<dict>` is used.

This method can be used to copy state between parsers.

Added in version 3.2.

get(*section*, *option*, *, *raw=False*, *vars=None*[, *fallback*])

Get an *option* value for the named *section*. If *vars* is provided, it must be a dictionary. The *option* is looked up in *vars* (if provided), *section*, and in *DEFAULTSECT* in that order. If the key is not found and *fallback* is provided, it is used as a fallback value. `None` can be provided as a *fallback* value.

All the `'%'` interpolations are expanded in the return values, unless the *raw* argument is true. Values for interpolation keys are looked up in the same manner as the option.

Changed in version 3.2: Arguments *raw*, *vars* and *fallback* are keyword only to protect users from trying to use the third argument as the *fallback* fallback (especially when using the mapping protocol).

getint(*section*, *option*, *, *raw=False*, *vars=None*[, *fallback*])

A convenience method which coerces the *option* in the specified *section* to an integer. See `get()` for explanation of *raw*, *vars* and *fallback*.

getfloat(*section*, *option*, *, *raw=False*, *vars=None*[, *fallback*])

A convenience method which coerces the *option* in the specified *section* to a floating-point number. See `get()` for explanation of *raw*, *vars* and *fallback*.

getboolean(*section*, *option*, *, *raw=False*, *vars=None*[, *fallback*])

A convenience method which coerces the *option* in the specified *section* to a Boolean value. Note that the accepted values for the option are `'1'`, `'yes'`, `'true'`, and `'on'`, which cause this method to return `True`, and `'0'`, `'no'`, `'false'`, and `'off'`, which cause it to return `False`. These string values are checked in a case-insensitive manner. Any other value will cause it to raise `ValueError`. See `get()` for explanation of *raw*, *vars* and *fallback*.

items(*raw=False*, *vars=None*)
items(*section*, *raw=False*, *vars=None*)

When *section* is not given, return a list of *section_name*, *section_proxy* pairs, including DEFAULTSECT.

Otherwise, return a list of *name*, *value* pairs for the options in the given *section*. Optional arguments have the same meaning as for the `get()` method.

Changed in version 3.8: Items present in *vars* no longer appear in the result. The previous behaviour mixed actual parser options with variables provided for interpolation.

set(*section*, *option*, *value*)

If the given section exists, set the given option to the specified value; otherwise raise `NoSectionError`. *option* and *value* must be strings; if not, `TypeError` is raised.

write(*fileobject*, *space_around_delimiters=True*)

Write a representation of the configuration to the specified file object, which must be opened in text mode (accepting strings). This representation can be parsed by a future `read()` call. If *space_around_delimiters* is true, delimiters between keys and values are surrounded by spaces.

Changed in version 3.14: Raises InvalidWriteError if this would write a representation which cannot be accurately parsed by a future `read()` call from this parser.

Note

Comments in the original configuration file are not preserved when writing the configuration back. What is considered a comment, depends on the given values for *comment_prefix* and *inline_comment_prefix*.

remove_option(*section*, *option*)

Remove the specified *option* from the specified *section*. If the section does not exist, raise `NoSectionError`. If the option existed to be removed, return `True`; otherwise return `False`.

remove_section(*section*)

Remove the specified *section* from the configuration. If the section in fact existed, return `True`. Otherwise return `False`.

optionxform(*option*)

Transforms the option name *option* as found in an input file or as passed in by client code to the form that should be used in the internal structures. The default implementation returns a lower-case version of *option*; subclasses may override this or client code can set an attribute of this name on instances to affect this behavior.

You don’t need to subclass the parser to use this method, you can also set it on an instance, to a function that takes a string argument and returns a string. Setting it to `str`, for example, would make option names case sensitive:

```
cfgparser = ConfigParser()
cfgparser.optionxform = str
```

Note that when reading configuration files, whitespace around the option names is stripped before `optionxform()` is called.

configparser.UNNAMED_SECTION

A special object representing a section name used to reference the unnamed section (see Unnamed Sections).

configparser.MAX_INTERPOLATION_DEPTH

The maximum depth for recursive interpolation for `get()` when the *raw* parameter is false. This is relevant only when the default *interpolation* is used.

RawConfigParser Objects

class configparser.RawConfigParser(*defaults=None*, *dict_type=dict*, *allow_no_value=False*, ***, *delimiters=('=', ':')*, *comment_prefixes=('#', ';')*, *inline_comment_prefixes=None*, *strict=True*, *empty_lines_in_values=True*, *default_section=configparser.DEFAULTSECT*, *interpolation=BasicInterpolation()*, *converters={}*, *allow_unnamed_section=False*)

Legacy variant of the `ConfigParser`. It has interpolation disabled by default and allows for non-string section names, option names, and values via its unsafe `add_section` and `set` methods, as well as the legacy `defaults=` keyword argument handling.

Changed in version 3.2: allow_no_value, *delimiters*, *comment_prefixes*, *strict*, *empty_lines_in_values*, *default_section* and *interpolation* were added.

Changed in version 3.5: The *converters* argument was added.

Changed in version 3.8: The default *dict_type* is `dict`, since it now preserves insertion order.

Changed in version 3.13: The *allow_unnamed_section* argument was added.

Note

Consider using `ConfigParser` instead which checks types of the values to be stored internally. If you don't want interpolation, you can use `ConfigParser(interpolation=None)`.

add_section(*section*)

Add a section named *section* or `UNNAMED_SECTION` to the instance.

If the given section already exists, `DuplicateSectionError` is raised. If the *default section* name is passed, `ValueError` is raised. If `UNNAMED_SECTION` is passed and support is disabled, `UnnamedSectionDisabledError` is raised.

Type of *section* is not checked which lets users create non-string named sections. This behaviour is unsupported and may cause internal errors.

Changed in version 3.14: Added support for `UNNAMED_SECTION`.

set(*section*, *option*, *value*)

If the given section exists, set the given option to the specified value; otherwise raise `NoSectionError`. While it is possible to use `RawConfigParser` (or `ConfigParser` with *raw* parameters set to true) for *internal* storage of non-string values, full functionality (including interpolation and output to files) can only be achieved using string values.

This method lets users assign non-string values to keys internally. This behaviour is unsupported and will cause errors when attempting to write to a file or get it in non-raw mode. **Use the mapping protocol API** which does not allow such assignments to take place.

Exceptions

exception configparser.Error

Base class for all other `configparser` exceptions.

exception configparser.NoSectionError

Exception raised when a specified section is not found.

exception configparser.DuplicateSectionError

Exception raised if `add_section()` is called with the name of a section that is already present or in strict parsers when a section if found more than once in a single input file, string or dictionary.

Changed in version 3.2: Added the optional *source* and *lineno* attributes and parameters to `__init__()`.

exception configparser.DuplicateOptionError

Exception raised by strict parsers if a single option appears twice during reading from a single file, string or dictionary. This catches misspellings and case sensitivity-related errors, e.g. a dictionary may have two keys representing the same case-insensitive configuration key.

exception configparser.NoOptionError

Exception raised when a specified option is not found in the specified section.

exception configparser.InterpolationError

Base class for exceptions raised when problems occur performing string interpolation.

exception configparser.InterpolationDepthError

Exception raised when string interpolation cannot be completed because the number of iterations exceeds `MAX_INTERPOLATION_DEPTH`. Subclass of `InterpolationError`.

exception configparser.InterpolationMissingOptionError

Exception raised when an option referenced from a value does not exist. Subclass of `InterpolationError`.

exception configparser.InterpolationSyntaxError

Exception raised when the source text into which substitutions are made does not conform to the required syntax. Subclass of `InterpolationError`.

exception configparser.MissingSectionHeaderError

Exception raised when attempting to parse a file which has no section headers.

exception configparser.ParsingError

Exception raised when errors occur attempting to parse a file.

Changed in version 3.12: The `filename` attribute and **`__init__()`** constructor argument were removed. They have been available using the name `source` since 3.2.

exception configparser.MultilineContinuationError

Exception raised when a key without a corresponding value is continued with an indented line.

Added in version 3.13.

exception configparser.UnnamedSectionDisabledError

Exception raised when attempting to use the **`UNNAMED_SECTION`** without enabling it.

Added in version 3.14.

exception configparser.InvalidWriteError

Exception raised when an attempted **`ConfigParser.write()`** would not be parsed accurately with a future **`ConfigParser.read()`** call.

Ex: Writing a key beginning with the **`ConfigParser.SECTCRE`** pattern would parse as a section header when read. Attempting to write this will raise this exception.

Added in version 3.14.

Footnotes

[1] (1,2,3,4,5,6,7,8,9,10,11)

Config parsers allow for heavy customization. If you are interested in changing the behaviour outlined by the footnote reference, consult the Customizing Parser Behaviour section.

`tomllib` — Parse TOML files

Added in version 3.11.

Source code: Lib/tomllib [https://github.com/python/cpython/tree/3.14/Lib/tomllib]

This module provides an interface for parsing TOML 1.0.0 (Tom’s Obvious Minimal Language, https://toml.io [https://toml.io/en/]). This module does not support writing TOML.

See also

The Tomli-W package [https://pypi.org/project/tomli-w/] is a TOML writer that can be used in conjunction with this module, providing a write API familiar to users of the standard library **`marshal`** and **`pickle`** modules.

See also

The TOML Kit package [https://pypi.org/project/tomlkit/] is a style-preserving TOML library with both read and write capability. It is a recommended replacement for this module for editing already existing TOML files.

This module defines the following functions:

tomllib.load(*fp*, */*, ***, *parse_float=float*)

Read a TOML file. The first argument should be a readable and binary file object. Return a **`dict`**. Convert TOML types to Python using this conversion table.

parse_float will be called with the string of every TOML float to be decoded. By default, this is equivalent to `float(num_str)`. This can be used to use another datatype or parser for TOML floats (e.g. **`decimal.Decimal`**). The callable must not return a **`dict`** or a **`list`**, else a **`ValueError`** is raised.

A **`TOMLDecodeError`** will be raised on an invalid TOML document.

tomllib.loads(*s*, */*, ***, *parse_float=float*)

Load TOML from a **`str`** object. Return a **`dict`**. Convert TOML types to Python using this conversion table. The *parse_float* argument has the same meaning as in **`load()`**.

A **`TOMLDecodeError`** will be raised on an invalid TOML document.

The following exceptions are available:

exception tomllib.TOMLDecodeError(*msg*, *doc*, *pos*)

Subclass of **`ValueError`** with the following additional attributes:

msg

The unformatted error message.

doc

The TOML document being parsed.

pos

The index of *doc* where parsing failed.

lineno

The line corresponding to *pos*.

colno

The column corresponding to *pos*.

Changed in version 3.14: Added the *msg*, *doc* and *pos* parameters. Added the `msg`, `doc`, `pos`, `lineno` and `colno` attributes.

Deprecated since version 3.14: Passing free-form positional arguments is deprecated.

Examples

Parsing a TOML file:

```
import tomllib

with open("pyproject.toml", "rb") as f:
    data = tomllib.load(f)
```

Parsing a TOML string:

```
import tomllib

toml_str = """
python-version = "3.11.0"
python-implementation = "CPython"
"""

data = tomllib.loads(toml_str)
```

Conversion Table

TOML	Python
TOML document	dict

TOML	Python
string	str
integer	int
float	float (configurable with *parse_float*)
boolean	bool
offset date-time	datetime.datetime (`tzinfo` attribute set to an instance of `datetime.timezone`)
local date-time	datetime.datetime (`tzinfo` attribute set to `None`)
local date	datetime.date
local time	datetime.time
array	list
table	dict
inline table	dict

TOML	Python
array of tables	list of dicts

`netrc` — netrc file processing

Source code: Lib/netrc.py
[https://github.com/python/cpython/tree/3.14/Lib/netrc.py]

The `netrc` class parses and encapsulates the netrc file format used by the Unix **ftp** program and other FTP clients.

class netrc.netrc([*file*])

A `netrc` instance or subclass instance encapsulates data from a netrc file. The initialization argument, if present, specifies the file to parse. If no argument is given, the file `.netrc` in the user's home directory – as determined by `os.path.expanduser()` – will be read. Otherwise, a `FileNotFoundError` exception will be raised. Parse errors will raise `NetrcParseError` with diagnostic information including the file name, line number, and terminating token.

If no argument is specified on a POSIX system, the presence of passwords in the `.netrc` file will raise a `NetrcParseError` if the file ownership or permissions are insecure (owned by a user other than the user running the process, or accessible for read or write by any other user). This implements security behavior equivalent to that of ftp and other programs that use `.netrc`. Such security checks are not available on platforms that do not support `os.getuid()`.

Changed in version 3.4: Added the POSIX permission check.

Changed in version 3.7: `os.path.expanduser()` is used to find the location of the `.netrc` file when *file* is not passed as argument.

Changed in version 3.10: `netrc` try UTF-8 encoding before using locale specific encoding. The entry in the netrc file no longer needs to contain all tokens. The missing tokens' value default to an empty string. All the tokens and their values now can contain arbitrary characters, like whitespace and non-ASCII characters. If the login name is anonymous, it won't trigger the security check.

exception netrc.NetrcParseError

Exception raised by the `netrc` class when syntactical errors are encountered in source text. Instances of this exception provide three interesting attributes:

msg

Textual explanation of the error.

filename

The name of the source file.

lineno

The line number on which the error was found.

netrc Objects

A `netrc` instance has the following methods:

netrc.authenticators(*host*)

Return a 3-tuple `(login, account, password)` of authenticators for *host*. If the netrc file did not contain an entry for the given host, return the tuple associated with the ‘default’ entry. If neither matching host nor default entry is available, return `None`.

netrc.__repr__()

Dump the class data as a string in the format of a netrc file. (This discards comments and may reorder the entries.)

Instances of **`netrc`** have public instance variables:

netrc.hosts

Dictionary mapping host names to `(login, account, password)` tuples. The ‘default’ entry, if any, is represented as a pseudo-host by that name.

netrc.macros

Dictionary mapping macro names to string lists.

`plistlib` — Generate and parse Apple `.plist` files

Source code: Lib/plistlib.py
[https://github.com/python/cpython/tree/3.14/Lib/plistlib.py]

This module provides an interface for reading and writing the “property list” files used by Apple, primarily on macOS and iOS. This module supports both binary and XML plist files.

The property list (`.plist`) file format is a simple serialization supporting basic object types, like dictionaries, lists, numbers and strings. Usually the top level object is a dictionary.

To write out and to parse a plist file, use the **`dump()`** and **`load()`** functions.

To work with plist data in bytes or string objects, use **`dumps()`** and **`loads()`**.

Values can be strings, integers, floats, booleans, tuples, lists, dictionaries (but only with string keys), **`bytes`**, **`bytearray`** or **`datetime.datetime`** objects.

Changed in version 3.4: New API, old API deprecated. Support for binary format plists added.

Changed in version 3.8: Support added for reading and writing **`UID`** tokens in binary plists as used by NSKeyedArchiver and NSKeyedUnarchiver.

Changed in version 3.9: Old API removed.

See also

PList manual page [https://developer.apple.com/library/archive/documentation/Cocoa/Conceptual/PropertyLists/]
Apple's documentation of the file format.

This module defines the following functions:

plistlib.load(*fp*, *, *fmt=None*, *dict_type=dict*, *aware_datetime=False*)

Read a plist file. *fp* should be a readable and binary file object. Return the unpacked root object (which usually is a dictionary).

The *fmt* is the format of the file and the following values are valid:

- `None`: Autodetect the file format
- `FMT_XML`: XML file format
- `FMT_BINARY`: Binary plist format

The *dict_type* is the type used for dictionaries that are read from the plist file.

When *aware_datetime* is true, fields with type `datetime.datetime` will be created as aware object, with **`tzinfo`** as `datetime.UTC`.

XML data for the `FMT_XML` format is parsed using the Expat parser from `xml.parsers.expat` – see its documentation for possible exceptions on ill-formed XML. Unknown elements will simply be ignored by the plist parser.

The parser raises `InvalidFileException` when the file cannot be parsed.

Added in version 3.4.

Changed in version 3.13: The keyword-only parameter *aware_datetime* has been added.

plistlib.loads(*data*, *, *fmt=None*, *dict_type=dict*, *aware_datetime=False*)

Load a plist from a bytes or string object. See `load()` for an explanation of the keyword arguments.

Added in version 3.4.

Changed in version 3.13: data can be a string when *fmt* equals `FMT_XML`.

plistlib.dump(*value*, *fp*, *, *fmt=FMT_XML*, *sort_keys=True*, *skipkeys=False*, *aware_datetime=False*)

Write *value* to a plist file. *fp* should be a writable, binary file object.

The *fmt* argument specifies the format of the plist file and can be one of the following values:

- `FMT_XML`: XML formatted plist file
- `FMT_BINARY`: Binary formatted plist file

When *sort_keys* is true (the default) the keys for dictionaries will be written to the plist in sorted order, otherwise they will be written in the iteration order of the dictionary.

When *skipkeys* is false (the default) the function raises `TypeError` when a key of a dictionary is not a string, otherwise such keys are skipped.

When *aware_datetime* is true and any field with type `datetime.datetime` is set as an aware object, it will convert to UTC timezone before writing it.

A **`TypeError`** will be raised if the object is of an unsupported type or a container that contains objects of unsupported types.

An **`OverflowError`** will be raised for integer values that cannot be represented in (binary) plist files.

Added in version 3.4.

Changed in version 3.13: The keyword-only parameter *aware_datetime* has been added.

plistlib.dumps(*value*, *, *fmt=FMT_XML*, *sort_keys=True*, *skipkeys=False*, *aware_datetime=False*)

Return *value* as a plist-formatted bytes object. See the documentation for **`dump()`** for an explanation of the keyword arguments of this function.

Added in version 3.4.

The following classes are available:

class plistlib.UID(*data*)

Wraps an **`int`**. This is used when reading or writing NSKeyedArchiver encoded data, which contains UID (see PList manual).

data

Int value of the UID. It must be in the range `0 <= data < 2**64`.

Added in version 3.8.

The following constants are available:

plistlib.FMT_XML
  The XML format for plist files.

  Added in version 3.4.

plistlib.FMT_BINARY
  The binary format for plist files

  Added in version 3.4.

The module defines the following exceptions:

exception plistlib.InvalidFileException
  Raised when a file cannot be parsed.

  Added in version 3.4.

Examples

Generating a plist:

```
import datetime
import plistlib

pl = dict(
    aString = "Doodah",
    aList = ["A", "B", 12, 32.1, [1, 2, 3]],
    aFloat = 0.1,
    anInt = 728,
    aDict = dict(
        anotherString = "<hello & hi there!>",
        aThirdString = "M\xe4ssig, Ma\xdf",
        aTrueValue = True,
        aFalseValue = False,
    ),
    someData = b"<binary gunk>",
    someMoreData = b"<lots of binary gunk>" * 10,
    aDate = datetime.datetime.now()
)
print(plistlib.dumps(pl).decode())
```

Parsing a plist:

```
import plistlib

plist = b"""<plist version="1.0">
<dict>
    <key>foo</key>
    <string>bar</string>
</dict>
</plist>"""
pl = plistlib.loads(plist)
print(pl["foo"])
```

Cryptographic Services

The modules described in this chapter implement various algorithms of a cryptographic nature. They are available at the discretion of the installation. Here’s an overview:

- `hashlib` — Secure hashes and message digests
  - Hash algorithms
  - Usage
  - Constructors
  - Attributes
  - Hash Objects
  - SHAKE variable length digests
  - File hashing
  - Key derivation
  - BLAKE2
    - Creating hash objects
    - Constants
    - Examples
      - Simple hashing
      - Using different digest sizes
      - Keyed hashing
      - Randomized hashing
      - Personalization
      - Tree mode
    - Credits
- `hmac` — Keyed-Hashing for Message Authentication
- `secrets` — Generate secure random numbers for managing secrets
  - Random numbers
  - Generating tokens
    - How many bytes should tokens use?
  - Other functions

    - Recipes and best practices

`hashlib` — Secure hashes and message digests

Source code: Lib/hashlib.py [https://github.com/python/cpython/tree/3.14/Lib/hashlib.py]

This module implements a common interface to many different hash algorithms. Included are the FIPS secure hash algorithms SHA224, SHA256, SHA384, SHA512, (defined in the FIPS 180-4 standard [https://csrc.nist.gov/pubs/fips/180-4/upd1/final]), the SHA-3 series (defined in the FIPS 202 standard [https://csrc.nist.gov/pubs/fips/202/final]) as well as the legacy algorithms SHA1 (formerly part of FIPS [https://csrc.nist.gov/news/2023/decision-to-revise-fips-180-4]) and the MD5 algorithm (defined in internet **RFC 1321** [https://datatracker.ietf.org/doc/html/rfc1321.html]).

Note

If you want the adler32 or crc32 hash functions, they are available in the **`zlib`** module.

Hash algorithms

There is one constructor method named for each type of *hash*. All return a hash object with the same simple interface. For example: use **`sha256()`** to create a SHA-256 hash object. You can now feed this object with bytes-like objects (normally **`bytes`**) using the **`update`** method. At any point you can ask it for the *digest* of the concatenation of the data fed to it so far using the `digest()` or `hexdigest()` methods.

To allow multithreading, the Python GIL is released while computing a hash supplied more than 2047 bytes of data at once in its constructor or `.update` method.

Constructors for hash algorithms that are always present in this module are `sha1()`, `sha224()`, `sha256()`, `sha384()`, `sha512()`, `sha3_224()`, `sha3_256()`, `sha3_384()`, `sha3_512()`, `shake_128()`, `shake_256()`, `blake2b()`, and `blake2s()`. `md5()` is normally available as well, though it may be missing or blocked if you are using a rare “FIPS compliant” build of Python. These correspond to `algorithms_guaranteed`.

Additional algorithms may also be available if your Python distribution’s `hashlib` was linked against a build of OpenSSL that provides others. Others *are not guaranteed available* on all installations and will only be accessible by name via `new()`. See `algorithms_available`.

Warning

Some algorithms have known hash collision weaknesses (including MD5 and SHA1). Refer to Attacks on cryptographic hash algorithms [https://en.wikipedia.org/wiki/Cryptographic_hash_function#Attacks_on_cryptographic_hash_algorithms] and the hashlib-seealso section at the end of this document.

Added in version 3.6: SHA3 (Keccak) and SHAKE constructors `sha3_224()`, `sha3_256()`, `sha3_384()`, `sha3_512()`, `shake_128()`, `shake_256()` were added. `blake2b()` and `blake2s()` were added.

Changed in version 3.9: All hashlib constructors take a keyword-only argument *usedforsecurity* with default value `True`. A false value allows the use of insecure and blocked hashing algorithms in restricted environments. `False` indicates that the hashing algorithm is not used in a security context, e.g. as a non-cryptographic one-way compression function.

Changed in version 3.9: Hashlib now uses SHA3 and SHAKE from OpenSSL if it provides it.

Changed in version 3.12: For any of the MD5, SHA1, SHA2, or SHA3 algorithms that the linked OpenSSL does not provide we fall back to a verified implementation from the HACL* project [https://github.com/hacl-star/hacl-star].

Usage

To obtain the digest of the byte string `b"Nobody inspects the spammish repetition"`:

```
>>> import hashlib
>>> m = hashlib.sha256()
>>> m.update(b"Nobody inspects")
>>> m.update(b" the spammish repetition")
>>> m.digest()
b'\x03\x1e\xdd}Ae\x15\x93\xc5\xfe\\\x00o\xa5u+7\xfd\xdf\xf7\xbc
N\x84:\xa6\xaf\x0c\x95\x0fK\x94\x06'
>>> m.hexdigest()
'031edd7d41651593c5fe5c006fa5752b37fddff7bc4e843aa6af0c950f4b94
06'
```

More condensed:

```
>>> hashlib.sha256(b"Nobody inspects the spammish
repetition").hexdigest()
'031edd7d41651593c5fe5c006fa5752b37fddff7bc4e843aa6af0c950f4b94
06'
```

Constructors

hashlib.new(*name*, [*data*,]*, *usedforsecurity=True*)

Is a generic constructor that takes the string *name* of the desired algorithm as its first parameter. It also exists to allow access to the above listed hashes as well as any other algorithms that your OpenSSL library may offer.

Using `new()` with an algorithm name:

```
>>> h = hashlib.new('sha256')
>>> h.update(b"Nobody inspects the spammish repetition")
>>> h.hexdigest()
'031edd7d41651593c5fe5c006fa5752b37fddff7bc4e843aa6af0c950f4b94
06'
```

hashlib.md5([*data*,]*, *usedforsecurity=True*)

hashlib.sha1([*data*,]*, *usedforsecurity=True*)

hashlib.sha224([*data*,]*, *usedforsecurity=True*)

hashlib.sha256([*data*,]*, *usedforsecurity=True*)

hashlib.sha384([*data*,]*, *usedforsecurity=True*)

hashlib.sha512([*data*,]*, *usedforsecurity=True*)

hashlib.sha3_224([*data*,]*, *usedforsecurity=True*)

hashlib.sha3_256([*data*,]*, *usedforsecurity=True*)

hashlib.sha3_384([*data*,]*, *usedforsecurity=True*) hashlib.sha3_512([*data*,]*, *usedforsecurity=True*)

Named constructors such as these are faster than passing an algorithm name to `new()`.

Attributes

Hashlib provides the following constant module attributes:

hashlib.algorithms_guaranteed

A set containing the names of the hash algorithms guaranteed to be supported by this module on all platforms. Note that ‘md5’ is in this list despite some upstream vendors offering an odd “FIPS compliant” Python build that excludes it.

Added in version 3.2.

hashlib.algorithms_available

A set containing the names of the hash algorithms that are available in the running Python interpreter. These names will be recognized when passed to `new()`. `algorithms_guaranteed` will always be a subset. The same algorithm may appear multiple times in this set under different names (thanks to OpenSSL).

Added in version 3.2.

Hash Objects

The following values are provided as constant attributes of the hash objects returned by the constructors:

hash.digest_size

The size of the resulting hash in bytes.

hash.block_size

The internal block size of the hash algorithm in bytes.

A hash object has the following attributes:

hash.name

The canonical name of this hash, always lowercase and always suitable as a parameter to **`new()`** to create another hash of this type.

Changed in version 3.4: The name attribute has been present in CPython since its inception, but until Python 3.4 was not formally specified, so may not exist on some platforms.

A hash object has the following methods:

hash.update(*data*)

Update the hash object with the bytes-like object. Repeated calls are equivalent to a single call with the concatenation of all the arguments: `m.update(a); m.update(b)` is equivalent to `m.update(a+b)`.

hash.digest()

Return the digest of the data passed to the **`update()`** method so far. This is a bytes object of size **`digest_size`** which may contain bytes in the whole range from 0 to 255.

hash.hexdigest()

Like **`digest()`** except the digest is returned as a string object of double length, containing only hexadecimal digits. This may be used to exchange the value safely in email or other non-binary environments.

hash.copy()

Return a copy (“clone”) of the hash object. This can be used to efficiently compute the digests of data sharing a common initial substring.

SHAKE variable length digests

hashlib.shake_128([*data*,]*, *usedforsecurity=True*)

hashlib.shake_256([*data*,]*, *usedforsecurity=True*)

The `shake_128()` and `shake_256()` algorithms provide variable length digests with length_in_bits//2 up to 128 or 256 bits of security. As such, their digest methods require a length. Maximum length is not limited by the SHAKE algorithm.

shake.digest(*length*)

Return the digest of the data passed to the `update()` method so far. This is a bytes object of size *length* which may contain bytes in the whole range from 0 to 255.

shake.hexdigest(*length*)

Like `digest()` except the digest is returned as a string object of double length, containing only hexadecimal digits. This may be used to exchange the value in email or other non-binary environments.

Example use:

```
>>> h = hashlib.shake_256(b'Nobody inspects the spammish
repetition')
>>> h.hexdigest(20)
'44709d6fcb83d92a76dcb0b668c98e1b1d3dafe7'
```

File hashing

The hashlib module provides a helper function for efficient hashing of a file or file-like object.

hashlib.file_digest(*fileobj*, *digest*, */*)

Return a digest object that has been updated with contents of file object.

fileobj must be a file-like object opened for reading in binary mode. It accepts file objects from builtin `open()`, `BytesIO` instances, SocketIO objects from `socket.socket.makefile()`, and similar. *fileobj* must be opened in blocking mode, otherwise a `BlockingIOError` may be raised.

The function may bypass Python’s I/O and use the file descriptor from `fileno()` directly. *fileobj* must be assumed to be in an unknown state after this function returns or raises. It is up to the caller to close *fileobj*.

digest must either be a hash algorithm name as a *str*, a hash constructor, or a callable that returns a hash object.

Example:

```
>>> import io, hashlib, hmac
>>> with open("library/hashlib.rst", "rb") as f:
...     digest = hashlib.file_digest(f, "sha256")
...
>>> digest.hexdigest()
'...'

>>> buf = io.BytesIO(b"somedata")
>>> mac1 = hmac.HMAC(b"key", digestmod=hashlib.sha512)
>>> digest = hashlib.file_digest(buf, lambda: mac1)

>>> digest is mac1
True
>>> mac2 = hmac.HMAC(b"key", b"somedata",
digestmod=hashlib.sha512)
```

```
>>> mac1.digest() == mac2.digest()
True
```

Added in version 3.11.

Changed in version 3.14: Now raises a `BlockingIOError` if the file is opened in non-blocking mode. Previously, spurious null bytes were added to the digest.

Key derivation

Key derivation and key stretching algorithms are designed for secure password hashing. Naive algorithms such as `sha1(password)` are not resistant against brute-force attacks. A good password hashing function must be tunable, slow, and include a salt [https://en.wikipedia.org/wiki/Salt_%28cryptography%29].

hashlib.pbkdf2_hmac(*hash_name*, *password*, *salt*, *iterations*, *dklen=None*)

The function provides PKCS#5 password-based key derivation function 2. It uses HMAC as pseudorandom function.

The string *hash_name* is the desired name of the hash digest algorithm for HMAC, e.g. ‘sha1’ or ‘sha256’. *password* and *salt* are interpreted as buffers of bytes. Applications and libraries should limit *password* to a sensible length (e.g. 1024). *salt* should be about 16 or more bytes from a proper source, e.g. `os.urandom()`.

The number of *iterations* should be chosen based on the hash algorithm and computing power. As of 2022, hundreds of thousands of iterations of SHA-256 are suggested. For rationale as to why and how to choose what is best for your application, read *Appendix A.2.2* of NIST-SP-800-132 [https://nvlpubs.nist.gov/nistpubs/Legacy/SP/nistspecialpublication800-

132.pdf]. The answers on the stackexchange pbkdf2 iterations question [https://security.stackexchange.com/questions/3959/recommended-of-iterations-when-using-pbkdf2-sha256/] explain in detail.

dklen is the length of the derived key in bytes. If *dklen* is `None` then the digest size of the hash algorithm *hash_name* is used, e.g. 64 for SHA-512.

```
>>> from hashlib import pbkdf2_hmac
>>> our_app_iters = 500_000  # Application specific, read
above.
>>> dk = pbkdf2_hmac('sha256', b'password', b'bad salt' * 2,
our_app_iters)
>>> dk.hex()
'15530bba69924174860db778f2c6f8104d3aaf9d26241840c8c4a641c8d
000a9'
```

Function only available when Python is compiled with OpenSSL.

Added in version 3.4.

Changed in version 3.12: Function now only available when Python is built with OpenSSL. The slow pure Python implementation has been removed.

hashlib.scrypt(*password*, *, *salt*, *n*, *r*, *p*, *maxmem=0*, *dklen=64*)

The function provides scrypt password-based key derivation function as defined in **RFC 7914** [https://datatracker.ietf.org/doc/html/rfc7914.html].

password and *salt* must be bytes-like objects. Applications and libraries should limit *password* to a sensible length (e.g. 1024). *salt* should be about 16 or more bytes from a proper source, e.g. `os.urandom()`.

n is the CPU/Memory cost factor, *r* the block size, *p* parallelization factor and *maxmem* limits memory (OpenSSL 1.1.0 defaults to 32 MiB). *dklen* is the length of the derived key in bytes.

Added in version 3.6.

BLAKE2

BLAKE2 [https://www.blake2.net] is a cryptographic hash function defined in **RFC 7693** [https://datatracker.ietf.org/doc/html/rfc7693.html] that comes in two flavors:

- **BLAKE2b**, optimized for 64-bit platforms and produces digests of any size between 1 and 64 bytes,
- **BLAKE2s**, optimized for 8- to 32-bit platforms and produces digests of any size between 1 and 32 bytes.

BLAKE2 supports **keyed mode** (a faster and simpler replacement for HMAC [https://en.wikipedia.org/wiki/Hash-based_message_authentication_code]), **salted hashing**, **personalization**, and **tree hashing**.

Hash objects from this module follow the API of standard library's `hashlib` objects.

Creating hash objects

New hash objects are created by calling constructor functions:

hashlib.blake2b(*data=b''*, *, *digest_size=64*, *key=b''*, *salt=b''*, *person=b''*, *fanout=1*, *depth=1*, *leaf_size=0*, *node_offset=0*, *node_depth=0*, *inner_size=0*, *last_node=False*, *usedforsecurity=True*) hashlib.blake2s(*data=b''*, *, *digest_size=32*, *key=b''*, *salt=b''*, *person=b''*, *fanout=1*, *depth=1*, *leaf_size=0*, *node_offset=0*, *node_depth=0*, *inner_size=0*, *last_node=False*, *usedforsecurity=True*)

These functions return the corresponding hash objects for calculating BLAKE2b or BLAKE2s. They optionally take these general parameters:

- *data*: initial chunk of data to hash, which must be bytes-like object. It can be passed only as positional argument.
- *digest_size*: size of output digest in bytes.
- *key*: key for keyed hashing (up to 64 bytes for BLAKE2b, up to 32 bytes for BLAKE2s).
- *salt*: salt for randomized hashing (up to 16 bytes for BLAKE2b, up to 8 bytes for BLAKE2s).
- *person*: personalization string (up to 16 bytes for BLAKE2b, up to 8 bytes for BLAKE2s).

The following table shows limits for general parameters (in bytes):

Hash	digest_size	len(key)	len(salt)	len(person)
BLAKE2b	64	64	16	16
BLAKE2s	32	32	8	8

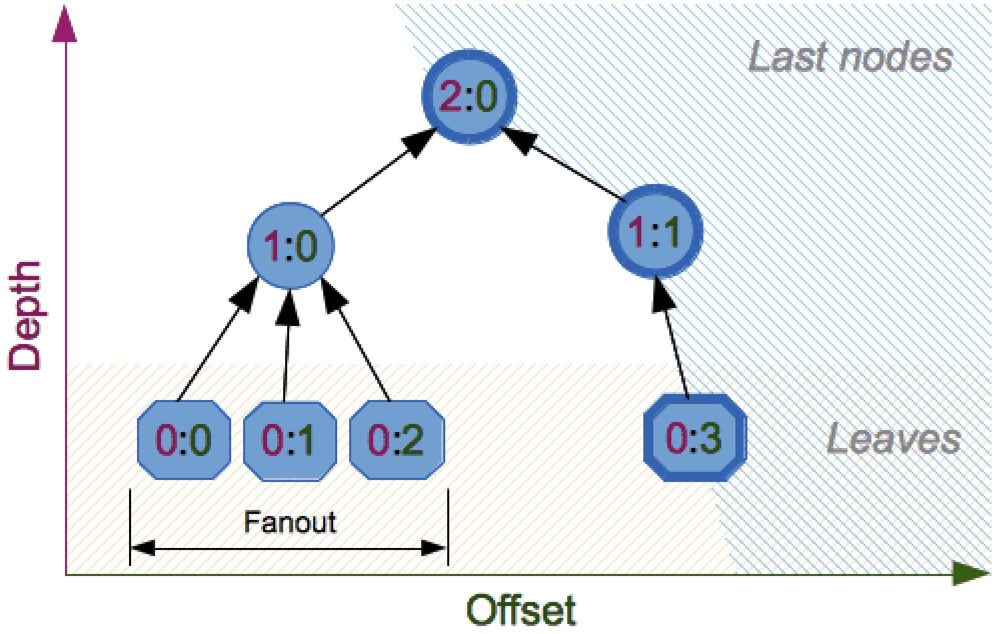
Note

BLAKE2 specification defines constant lengths for salt and personalization parameters, however, for convenience, this implementation accepts byte strings of any size up to the specified length. If the length of the parameter is less than specified, it is padded with zeros, thus, for example, `b'salt'` and `b'salt\x00'` is the same value. (This is not the case for *key*.)

These sizes are available as module constants described below.

Constructor functions also accept the following tree hashing parameters:

- *fanout*: fanout (0 to 255, 0 if unlimited, 1 in sequential mode).
- *depth*: maximal depth of tree (1 to 255, 255 if unlimited, 1 in sequential mode).
- *leaf_size*: maximal byte length of leaf (0 to `2**32-1`, 0 if unlimited or in sequential mode).
- *node_offset*: node offset (0 to `2**64-1` for BLAKE2b, 0 to `2**48-1` for BLAKE2s, 0 for the first, leftmost, leaf, or in sequential mode).
- *node_depth*: node depth (0 to 255, 0 for leaves, or in sequential mode).
- *inner_size*: inner digest size (0 to 64 for BLAKE2b, 0 to 32 for BLAKE2s, 0 in sequential mode).
- *last_node*: boolean indicating whether the processed node is the last one (`False` for sequential mode).


Last nodes
2:0
1:0
1:1
Depth
0:0
0:1
0:2
0:3
Leaves
Fanout
Offset

See section 2.10 in [BLAKE2 specification](https://www.blake2.net/blake2_20130129.pdf) [https://www.blake2.net/blake2_20130129.pdf] for comprehensive review of tree hashing.

Constants

blake2b.SALT_SIZE

blake2s.SALT_SIZE

Salt length (maximum length accepted by constructors).

blake2b.PERSON_SIZE

blake2s.PERSON_SIZE

Personalization string length (maximum length accepted by constructors).

blake2b.MAX_KEY_SIZE

blake2s.MAX_KEY_SIZE

Maximum key size.

blake2b.MAX_DIGEST_SIZE

blake2s.MAX_DIGEST_SIZE

Maximum digest size that the hash function can output.

Examples

Simple hashing

To calculate hash of some data, you should first construct a hash object by calling the appropriate constructor function (`blake2b()` or `blake2s()`), then update it with the data by calling `update()` on the object, and, finally, get the digest out of the object by calling `digest()` (or `hexdigest()` for hex-encoded string).

```
>>> from hashlib import blake2b
>>> h = blake2b()
>>> h.update(b'Hello world')
>>> h.hexdigest()
'6ff843ba685842aa82031d3f53c48b66326df7639a63d128974c5c14f31a0f
33343a8c65551134ed1ae0f2b0dd2bb495dc81039e3eeb0aa1bb0388bbeac29
183'
```

As a shortcut, you can pass the first chunk of data to update directly to the constructor as the positional argument:

```
>>> from hashlib import blake2b
>>> blake2b(b'Hello world').hexdigest()
'6ff843ba685842aa82031d3f53c48b66326df7639a63d128974c5c14f31a0f
33343a8c65551134ed1ae0f2b0dd2bb495dc81039e3eeb0aa1bb0388bbeac29
183'
```

You can call `hash.update()` as many times as you need to iteratively update the hash:

```
>>> from hashlib import blake2b
>>> items = [b'Hello', b' ', b'world']
>>> h = blake2b()
>>> for item in items:
...     h.update(item)
...
>>> h.hexdigest()
'6ff843ba685842aa82031d3f53c48b66326df7639a63d128974c5c14f31a0f
33343a8c65551134ed1ae0f2b0dd2bb495dc81039e3eeb0aa1bb0388bbeac29
183'
```

Using different digest sizes

BLAKE2 has configurable size of digests up to 64 bytes for BLAKE2b and up to 32 bytes for BLAKE2s. For example, to replace SHA-1 with BLAKE2b without changing the size of output, we can tell BLAKE2b to produce 20-byte digests:

```
>>> from hashlib import blake2b
>>> h = blake2b(digest_size=20)
>>> h.update(b'Replacing SHA1 with the more secure function')
>>> h.hexdigest()
'd24f26cf8de66472d58d4e1b1774b4c9158b1f4c'
>>> h.digest_size
20
>>> len(h.digest())
20
```

Hash objects with different digest sizes have completely different outputs (shorter hashes are *not* prefixes of longer hashes); BLAKE2b and BLAKE2s produce different outputs even if the output length is the same:

```
>>> from hashlib import blake2b, blake2s
>>> blake2b(digest_size=10).hexdigest()
'6fa1d8fcfd719046d762'
>>> blake2b(digest_size=11).hexdigest()
'eb6ec15daf9546254f0809'
>>> blake2s(digest_size=10).hexdigest()
'1bf21a98c78a1c376ae9'
```

```
>>> blake2s(digest_size=11).hexdigest()
'567004bf96e4a25773ebf4'
```

Keyed hashing

Keyed hashing can be used for authentication as a faster and simpler replacement for Hash-based message authentication code [https://en.wikipedia.org/wiki/HMAC] (HMAC). BLAKE2 can be securely used in prefix-MAC mode thanks to the indifferentiability property inherited from BLAKE.

This example shows how to get a (hex-encoded) 128-bit authentication code for message `b'message data'` with key `b'pseudorandom key'`:

```
>>> from hashlib import blake2b
>>> h = blake2b(key=b'pseudorandom key', digest_size=16)
>>> h.update(b'message data')
>>> h.hexdigest()
'3d363ff7401e02026f4a4687d4863ced'
```

As a practical example, a web application can symmetrically sign cookies sent to users and later verify them to make sure they weren’t tampered with:

```
>>> from hashlib import blake2b
>>> from hmac import compare_digest
>>>
>>> SECRET_KEY = b'pseudorandomly generated server secret key'
>>> AUTH_SIZE = 16
>>>
>>> def sign(cookie):
...     h = blake2b(digest_size=AUTH_SIZE, key=SECRET_KEY)
...     h.update(cookie)
...     return h.hexdigest().encode('utf-8')
>>>
>>> def verify(cookie, sig):
...     good_sig = sign(cookie)
...     return compare_digest(good_sig, sig)
>>>
```

```
>>> cookie = b'user-alice'
>>> sig = sign(cookie)
>>> print("{0},{1}".format(cookie.decode('utf-8'), sig))
user-alice,b'43b3c982cf697e0c5ab22172d1ca7421'
>>> verify(cookie, sig)
True
>>> verify(b'user-bob', sig)
False
>>> verify(cookie, b'0102030405060708090a0b0c0d0e0f00')
False
```

Even though there’s a native keyed hashing mode, BLAKE2 can, of course, be used in HMAC construction with `hmac` module:

```
>>> import hmac, hashlib
>>> m = hmac.new(b'secret key', digestmod=hashlib.blake2s)
>>> m.update(b'message')
>>> m.hexdigest()
'e3c8102868d28b5ff85fc35dda07329970d1a01e273c37481326fe0c861c81
42'
```

Randomized hashing

By setting *salt* parameter users can introduce randomization to the hash function. Randomized hashing is useful for protecting against collision attacks on the hash function used in digital signatures.

> Randomized hashing is designed for situations where one party, the message preparer, generates all or part of a message to be signed by a second party, the message signer. If the message preparer is able to find cryptographic hash function collisions (i.e., two messages producing the same hash value), then they might prepare meaningful versions of the message that would produce the same hash value and digital signature, but with different results (e.g., transferring

> $1,000,000 to an account, rather than $10). Cryptographic hash functions have been designed with collision resistance as a major goal, but the current concentration on attacking cryptographic hash functions may result in a given cryptographic hash function providing less collision resistance than expected. Randomized hashing offers the signer additional protection by reducing the likelihood that a preparer can generate two or more messages that ultimately yield the same hash value during the digital signature generation process — even if it is practical to find collisions for the hash function. However, the use of randomized hashing may reduce the amount of security provided by a digital signature when all portions of the message are prepared by the signer.
>
> (NIST SP-800-106 “Randomized Hashing for Digital Signatures” [https://csrc.nist.gov/pubs/sp/800/106/final])

In BLAKE2 the salt is processed as a one-time input to the hash function during initialization, rather than as an input to each compression function.

Warning

Salted hashing (or just hashing) with BLAKE2 or any other general-purpose cryptographic hash function, such as SHA-256, is not suitable for hashing passwords. See BLAKE2 FAQ [https://www.blake2.net/#qa] for more information.

```
>>> import os
>>> from hashlib import blake2b
>>> msg = b'some message'
>>> # Calculate the first hash with a random salt.
>>> salt1 = os.urandom(blake2b.SALT_SIZE)
>>> h1 = blake2b(salt=salt1)
>>> h1.update(msg)
>>> # Calculate the second hash with a different random salt.
>>> salt2 = os.urandom(blake2b.SALT_SIZE)
>>> h2 = blake2b(salt=salt2)
>>> h2.update(msg)
>>> # The digests are different.
>>> h1.digest() != h2.digest()
True
```

Personalization

Sometimes it is useful to force hash function to produce different digests for the same input for different purposes. Quoting the authors of the Skein hash function:

> We recommend that all application designers seriously consider doing this; we have seen many protocols where a hash that is computed in one part of the protocol can be used in an entirely different part because two hash computations were done on similar or related data, and the attacker can force the application to make the hash inputs the same. Personalizing each hash function used in the protocol summarily stops this type of attack.
>
> (The Skein Hash Function Family [https://www.schneier.com/wp-content/uploads/2016/02/skein.pdf], p. 21)

BLAKE2 can be personalized by passing bytes to the *person* argument:

```
>>> from hashlib import blake2b
>>> FILES_HASH_PERSON = b'MyApp Files Hash'
>>> BLOCK_HASH_PERSON = b'MyApp Block Hash'
>>> h = blake2b(digest_size=32, person=FILES_HASH_PERSON)
>>> h.update(b'the same content')
>>> h.hexdigest()
'20d9cd024d4fb086aae819a1432dd2466de12947831b75c5a30cf2676095d3
b4'
>>> h = blake2b(digest_size=32, person=BLOCK_HASH_PERSON)
>>> h.update(b'the same content')
>>> h.hexdigest()
'cf68fb5761b9c44e7878bfb2c4c9aea52264a80b75005e65619778de59f383
a3'
```

Personalization together with the keyed mode can also be used to derive different keys from a single one.

```
>>> from hashlib import blake2s
>>> from base64 import b64decode, b64encode
>>> orig_key =
b64decode(b'Rm5EPJai72qcK3RGBpW3vPNfZy5OZothY+kHY6h21KM=')
>>> enc_key = blake2s(key=orig_key,
person=b'kEncrypt').digest()
>>> mac_key = blake2s(key=orig_key, person=b'kMAC').digest()
>>> print(b64encode(enc_key).decode('utf-8'))
rbPb15S/Z9t+agffno5wuhB77VbRi6F9Iv2qIxU7WHw=
>>> print(b64encode(mac_key).decode('utf-8'))
G9GtHFE1YluXY1zWPlYk1e/nWfu0WSEb0KRcjhDeP/o=
```

Tree mode

Here’s an example of hashing a minimal tree with two leaf nodes:

```
  10
 /  \
00  01
```

This example uses 64-byte internal digests, and returns the 32-byte final digest:

```
>>> from hashlib import blake2b
>>>
>>> FANOUT = 2
>>> DEPTH = 2
>>> LEAF_SIZE = 4096
>>> INNER_SIZE = 64
>>>
>>> buf = bytearray(6000)
>>>
```

```
>>> # Left leaf
... h00 = blake2b(buf[0:LEAF_SIZE], fanout=FANOUT, depth=DEPTH,
...               leaf_size=LEAF_SIZE, inner_size=INNER_SIZE,
...               node_offset=0, node_depth=0, last_node=False)
>>> # Right leaf
... h01 = blake2b(buf[LEAF_SIZE:], fanout=FANOUT, depth=DEPTH,
...               leaf_size=LEAF_SIZE, inner_size=INNER_SIZE,
...               node_offset=1, node_depth=0, last_node=True)
>>> # Root node
... h10 = blake2b(digest_size=32, fanout=FANOUT, depth=DEPTH,
...               leaf_size=LEAF_SIZE, inner_size=INNER_SIZE,
...               node_offset=0, node_depth=1, last_node=True)
>>> h10.update(h00.digest())
>>> h10.update(h01.digest())
>>> h10.hexdigest()
'3ad2a9b37c6070e374c7a8c508fe20ca86b6ed54e286e93a0318e95e881db5
aa'
```

Credits

BLAKE2 [https://www.blake2.net] was designed by *Jean-Philippe Aumasson*, *Samuel Neves*, *Zooko Wilcox-O'Hearn*, and *Christian Winnerlein* based on SHA-3 [https://en.wikipedia.org/wiki/Secure_Hash_Algorithms] finalist BLAKE [https://web.archive.org/web/20200918190133/https://131002.net/blake/] created by *Jean-Philippe Aumasson*, *Luca Henzen*, *Willi Meier*, and *Raphael C.-W. Phan*.

It uses core algorithm from ChaCha [https://cr.yp.to/chacha.html] cipher designed by *Daniel J. Bernstein*.

The stdlib implementation is based on pyblake2 [https://pythonhosted.org/pyblake2/] module. It was written by *Dmitry Chestnykh* based on C implementation written by *Samuel Neves*. The documentation was copied from pyblake2 [https://pythonhosted.org/pyblake2/] and written by *Dmitry Chestnykh*.

The C code was partly rewritten for Python by *Christian Heimes*.

The following public domain dedication applies for both C hash function implementation, extension code, and this documentation:

> To the extent possible under law, the author(s) have dedicated all copyright and related and neighboring rights to this software to the public domain worldwide. This software is distributed without any warranty.
>
> You should have received a copy of the CC0 Public Domain Dedication along with this software. If not, see https://creativecommons.org/publicdomain/zero/1.0/.

The following people have helped with development or contributed their changes to the project and the public domain according to the Creative Commons Public Domain Dedication 1.0 Universal:

- *Alexandr Sokolovskiy*

See also

Module `hmac`
A module to generate message authentication codes using hashes.

Module `base64`
Another way to encode binary hashes for non-binary environments.

https://nvlpubs.nist.gov/nistpubs/fips/nist.fips.180-4.pdf
The FIPS 180-4 publication on Secure Hash Algorithms.

https://csrc.nist.gov/pubs/fips/202/final

The FIPS 202 publication on the SHA-3 Standard.

https://www.blake2.net/

Official BLAKE2 website.

https://en.wikipedia.org/wiki/Cryptographic_hash_function

Wikipedia article with information on which algorithms have known issues and what that means regarding their use.

https://www.ietf.org/rfc/rfc8018.txt

PKCS #5: Password-Based Cryptography Specification Version 2.1

https://nvlpubs.nist.gov/nistpubs/Legacy/SP/nistspecialpublication800-132.pdf

NIST Recommendation for Password-Based Key Derivation.

`hmac` — Keyed-Hashing for Message Authentication

Source code: Lib/hmac.py [https://github.com/python/cpython/tree/3.14/Lib/hmac.py]

This module implements the HMAC algorithm as described by **RFC 2104** [https://datatracker.ietf.org/doc/html/rfc2104.html]. The interface allows to use any hash function with a *fixed* digest size. In particular, extendable output functions such as SHAKE-128 or SHAKE-256 cannot be used with HMAC.

hmac.new(*key*, *msg=None*, *digestmod*)

Return a new hmac object. *key* is a bytes or bytearray object giving the secret key. If *msg* is present, the method call `update(msg)` is made. *digestmod* is the digest name, digest constructor or module for the HMAC object to use. It may be any name suitable to **`hashlib.new()`**. Despite its argument position, it is required.

Changed in version 3.4: Parameter *key* can be a bytes or bytearray object. Parameter *msg* can be of any type supported by **`hashlib`**. Parameter *digestmod* can be the name of a hash algorithm.

Changed in version 3.8: The *digestmod* argument is now required. Pass it as a keyword argument to avoid awkwardness when you do not have an initial *msg*.

hmac.digest(*key*, *msg*, *digest*)

Return digest of *msg* for given secret *key* and *digest*. The function is equivalent to `HMAC(key, msg, digest).digest()`, but uses an optimized C or inline implementation, which is faster for messages that fit into memory. The parameters *key*, *msg*, and *digest* have the same meaning as in `new()`.

CPython implementation detail, the optimized C implementation is only used when *digest* is a string and name of a digest algorithm, which is supported by OpenSSL.

Added in version 3.7.

class hmac.HMAC

An HMAC object has the following methods:

HMAC.update(*msg*)

Update the hmac object with *msg*. Repeated calls are equivalent to a single call with the concatenation of all the arguments: `m.update(a); m.update(b)` is equivalent to `m.update(a + b)`.

Changed in version 3.4: Parameter *msg* can be of any type supported by `hashlib`.

HMAC.digest()

Return the digest of the bytes passed to the `update()` method so far. This bytes object will be the same length as the *digest_size* of the digest given to the constructor. It may contain non-ASCII bytes, including NUL bytes.

Warning

When comparing the output of `digest()` to an externally supplied digest during a verification routine, it is recommended to use the `compare_digest()` function instead of the `==` operator to reduce the vulnerability to timing attacks.

HMAC.hexdigest()

Like `digest()` except the digest is returned as a string twice the length containing only hexadecimal digits. This may be used to exchange the value safely in email or other non-binary environments.

Warning

When comparing the output of `hexdigest()` to an externally supplied digest during a verification routine, it is recommended to use the `compare_digest()` function instead of the `==` operator to reduce the vulnerability to timing attacks.

HMAC.copy()

Return a copy (“clone”) of the hmac object. This can be used to efficiently compute the digests of strings that share a common initial substring.

A hash object has the following attributes:

HMAC.digest_size

The size of the resulting HMAC digest in bytes.

HMAC.block_size

The internal block size of the hash algorithm in bytes.

Added in version 3.4.

HMAC.name

The canonical name of this HMAC, always lowercase, e.g. `hmac-md5`.

Added in version 3.4.

Changed in version 3.10: Removed the undocumented attributes `HMAC.digest_cons`, `HMAC.inner`, and `HMAC.outer`.

This module also provides the following helper function:

hmac.compare_digest(*a*, *b*)

Return `a == b`. This function uses an approach designed to prevent timing analysis by avoiding content-based short circuiting behaviour, making it appropriate for cryptography. *a* and *b* must both be of the same type: either **`str`** (ASCII only, as e.g. returned by **`HMAC.hexdigest()`**), or a bytes-like object.

Note

If *a* and *b* are of different lengths, or if an error occurs, a timing attack could theoretically reveal information about the types and lengths of *a* and *b*—but not their values.

Added in version 3.3.

Changed in version 3.10: The function uses OpenSSL's `CRYPTO_memcmp()` internally when available.

See also

Module `hashlib`

The Python module providing secure hash functions.

`secrets` — Generate secure random numbers for managing secrets

Added in version 3.6.

Source code: Lib/secrets.py [https://github.com/python/cpython/tree/3.14/Lib/secrets.py]

The **`secrets`** module is used for generating cryptographically strong random numbers suitable for managing data such as passwords, account authentication, security tokens, and related secrets.

In particular, **`secrets`** should be used in preference to the default pseudo-random number generator in the **`random`** module, which is designed for modelling and simulation, not security or cryptography.

See also

PEP 506 [https://peps.python.org/pep-0506/]

Random numbers

The **`secrets`** module provides access to the most secure source of randomness that your operating system provides.

class secrets.SystemRandom

A class for generating random numbers using the highest-quality sources provided by the operating system. See `random.SystemRandom` for additional details.

secrets.choice(*seq*)

Return a randomly chosen element from a non-empty sequence.

secrets.randbelow(*exclusive_upper_bound*)

Return a random int in the range [0, *exclusive_upper_bound*).

secrets.randbits(*k*)

Return a non-negative int with *k* random bits.

Generating tokens

The `secrets` module provides functions for generating secure tokens, suitable for applications such as password resets, hard-to-guess URLs, and similar.

secrets.token_bytes([*nbytes=None*])

Return a random byte string containing *nbytes* number of bytes. If *nbytes* is `None` or not supplied, a reasonable default is used.

```
>>> token_bytes(16)
b'\xebr\x17D*t\xae\xd4\xe3S\xb6\xe2\xebP1\x8b'
```

secrets.token_hex([*nbytes=None*])

Return a random text string, in hexadecimal. The string has *nbytes* random bytes, each byte converted to two hex digits. If *nbytes* is `None` or not supplied, a reasonable default is used.

```
>>> token_hex(16)
'f9bf78b9a18ce6d46a0cd2b0b86df9da'
```

secrets.token_urlsafe([*nbytes=None*])

Return a random URL-safe text string, containing *nbytes* random bytes. The text is Base64 encoded, so on average each byte results in approximately 1.3 characters. If *nbytes* is `None` or not supplied, a reasonable default is used.

```
>>> token_urlsafe(16)
'Drmhze6EPcv0fN_81Bj-nA'
```

How many bytes should tokens use?

To be secure against brute-force attacks [https://en.wikipedia.org/wiki/Brute-force_attack], tokens need to have sufficient randomness. Unfortunately, what is considered sufficient will necessarily increase as computers get more powerful and able to make more guesses in a shorter period. As of 2015, it is believed that 32 bytes (256 bits) of randomness is sufficient for the typical use-case expected for the **`secrets`** module.

For those who want to manage their own token length, you can explicitly specify how much randomness is used for tokens by giving an **`int`** argument to the various `token_*` functions. That argument is taken as the number of bytes of randomness to use.

Otherwise, if no argument is provided, or if the argument is `None`, the `token_*` functions will use a reasonable default instead.

Note

That default is subject to change at any time, including during maintenance releases.

Other functions

secrets.compare_digest(*a*, *b*)

Return `True` if strings or bytes-like objects *a* and *b* are equal, otherwise `False`, using a “constant-time compare” to reduce the risk of timing attacks [https://codahale.com/a-lesson-in-timing-attacks/]. See **`hmac.compare_digest()`** for additional details.

Recipes and best practices

This section shows recipes and best practices for using **`secrets`** to manage a basic level of security.

Generate an eight-character alphanumeric password:

```
import string
import secrets
alphabet = string.ascii_letters + string.digits
password = ''.join(secrets.choice(alphabet) for i in range(8))
```

Note

Applications should not **store passwords in a recoverable format** [https://cwe.mitre.org/data/definitions/257.html], whether plain text or encrypted. They should be salted and hashed using a cryptographically strong one-way (irreversible) hash function.

Generate a ten-character alphanumeric password with at least one lowercase character, at least one uppercase character, and at least three digits:

```
import string
import secrets
alphabet = string.ascii_letters + string.digits
while True:
    password = ''.join(secrets.choice(alphabet) for i in
range(10))
    if (any(c.islower() for c in password)
            and any(c.isupper() for c in password)
            and sum(c.isdigit() for c in password) >= 3):
        break
```

Generate an XKCD-style passphrase [https://xkcd.com/936/]:

```
import secrets
# On standard Linux systems, use a convenient dictionary file.
# Other platforms may need to provide their own word-list.
with open('/usr/share/dict/words') as f:
    words = [word.strip() for word in f]
    password = ' '.join(secrets.choice(words) for i in
range(4))
```

Generate a hard-to-guess temporary URL containing a security token suitable for password recovery applications:

```
import secrets
url = 'https://example.com/reset=' + secrets.token_urlsafe()
```

Generic Operating System Services

The modules described in this chapter provide interfaces to operating system features that are available on (almost) all operating systems, such as files and a clock. The interfaces are generally modeled after the Unix or C interfaces, but they are available on most other systems as well. Here’s an overview:

- **`os`** — Miscellaneous operating system interfaces
  - File Names, Command Line Arguments, and Environment Variables
  - Python UTF-8 Mode
  - Process Parameters
  - File Object Creation
  - File Descriptor Operations
    - Querying the size of a terminal
    - Inheritance of File Descriptors
  - Files and Directories
    - Timer File Descriptors
    - Linux extended attributes
  - Process Management
  - Interface to the scheduler
  - Miscellaneous System Information
  - Random numbers
- **`io`** — Core tools for working with streams
  - Overview
    - Text I/O
    - Binary I/O
    - Raw I/O
  - Text Encoding

        - Opt-in EncodingWarning
    - High-level Module Interface
    - Class hierarchy
        - I/O Base Classes
        - Raw File I/O
        - Buffered Streams
        - Text I/O
    - Static Typing
    - Performance
        - Binary I/O
        - Text I/O
        - Multi-threading
        - Reentrancy
- `time` — Time access and conversions
    - Functions
    - Clock ID Constants
    - Timezone Constants
- `logging` — Logging facility for Python
    - Logger Objects
    - Logging Levels
    - Handler Objects
    - Formatter Objects
    - Filter Objects
    - LogRecord Objects
    - LogRecord attributes
    - LoggerAdapter Objects
    - Thread Safety
    - Module-Level Functions
    - Module-Level Attributes
    - Integration with the warnings module
- `logging.config` — Logging configuration
    - Configuration functions
    - Security considerations
    - Configuration dictionary schema
        - Dictionary Schema Details
        - Incremental Configuration

        - Object connections
        - User-defined objects
        - Handler configuration order
        - Access to external objects
        - Access to internal objects
        - Import resolution and custom importers
        - Configuring QueueHandler and QueueListener
    - Configuration file format
- `logging.handlers` — Logging handlers
    - StreamHandler
    - FileHandler
    - NullHandler
    - WatchedFileHandler
    - BaseRotatingHandler
    - RotatingFileHandler
    - TimedRotatingFileHandler
    - SocketHandler
    - DatagramHandler
    - SysLogHandler
    - NTEventLogHandler
    - SMTPHandler
    - MemoryHandler
    - HTTPHandler
    - QueueHandler
    - QueueListener
- `platform` — Access to underlying platform’s identifying data
    - Cross platform
    - Java platform
    - Windows platform
    - macOS platform
    - iOS platform
    - Unix platforms
    - Linux platforms
    - Android platform
    - Command-line usage

- `errno` — Standard errno system symbols
- `ctypes` — A foreign function library for Python
  - ctypes tutorial
    - Loading dynamic link libraries
    - Accessing functions from loaded dlls
    - Calling functions
    - Fundamental data types
    - Calling functions, continued
    - Calling variadic functions
    - Calling functions with your own custom data types
    - Specifying the required argument types (function prototypes)
    - Return types
    - Passing pointers (or: passing parameters by reference)
    - Structures and unions
    - Structure/union layout, alignment and byte order
    - Bit fields in structures and unions
    - Arrays
    - Pointers
    - Thread safety without the GIL
    - Type conversions
    - Incomplete Types
    - Callback functions
    - Accessing values exported from dlls
    - Surprises
    - Variable-sized data types
  - ctypes reference
    - Finding shared libraries
    - Listing loaded shared libraries
    - Loading shared libraries
    - Foreign functions
    - Function prototypes
    - Utility functions

- Data types
- Fundamental data types
- Structured data types
- Arrays and pointers
- Exceptions

`os` — Miscellaneous operating system interfaces

Source code: Lib/os.py [https://github.com/python/cpython/tree/3.14/Lib/os.py]

This module provides a portable way of using operating system dependent functionality. If you just want to read or write a file see `open()`, if you want to manipulate paths, see the `os.path` module, and if you want to read all the lines in all the files on the command line see the `fileinput` module. For creating temporary files and directories see the `tempfile` module, and for high-level file and directory handling see the `shutil` module.

Notes on the availability of these functions:

- The design of all built-in operating system dependent modules of Python is such that as long as the same functionality is available, it uses the same interface; for example, the function `os.stat(path)` returns stat information about *path* in the same format (which happens to have originated with the POSIX interface).
- Extensions peculiar to a particular operating system are also available through the `os` module, but using them is of course a threat to portability.
- All functions accepting path or file names accept both bytes and string objects, and result in an object of the same type, if a path or file name is returned.

- On VxWorks, os.popen, os.fork, os.execv and os.spawn*p* are not supported.
- On WebAssembly platforms, Android and iOS, large parts of the `os` module are not available or behave differently. APIs related to processes (e.g. `fork()`, `execve()`) and resources (e.g. `nice()`) are not available. Others like `getuid()` and `getpid()` are emulated or stubs. WebAssembly platforms also lack support for signals (e.g. `kill()`, `wait()`).

Note

All functions in this module raise `OSError` (or subclasses thereof) in the case of invalid or inaccessible file names and paths, or other arguments that have the correct type, but are not accepted by the operating system.

exception os.error

An alias for the built-in `OSError` exception.

os.name

The name of the operating system dependent module imported. The following names have currently been registered: `'posix'`, `'nt'`, `'java'`.

See also

`sys.platform` has a finer granularity. `os.uname()` gives system-dependent version information.

The `platform` module provides detailed checks for the system’s identity.

File Names, Command Line Arguments, and Environment Variables

In Python, file names, command line arguments, and environment variables are represented using the string type. On some systems, decoding these strings to and from bytes is necessary before passing them to the operating system. Python uses the filesystem encoding and error handler to perform this conversion (see `sys.getfilesystemencoding()`).

The filesystem encoding and error handler are configured at Python startup by the `PyConfig_Read()` function: see `filesystem_encoding` and `filesystem_errors` members of `PyConfig`.

Changed in version 3.1: On some systems, conversion using the file system encoding may fail. In this case, Python uses the surrogateescape encoding error handler, which means that undecodable bytes are replaced by a Unicode character U+DC*xx* on decoding, and these are again translated to the original byte on encoding.

The file system encoding must guarantee to successfully decode all bytes below 128. If the file system encoding fails to provide this guarantee, API functions can raise `UnicodeError`.

See also the locale encoding.

Python UTF-8 Mode

Added in version 3.7: See **PEP 540** [https://peps.python.org/pep-0540/] for more details.

The Python UTF-8 Mode ignores the locale encoding and forces the usage of the UTF-8 encoding:

- Use UTF-8 as the filesystem encoding.
- `sys.getfilesystemencoding()` returns `'utf-8'`.
- `locale.getpreferredencoding()` returns `'utf-8'` (the *do_setlocale* argument has no effect).
- `sys.stdin`, `sys.stdout`, and `sys.stderr` all use UTF-8 as their text encoding, with the `surrogateescape` error handler being enabled for `sys.stdin` and `sys.stdout` (`sys.stderr` continues to use `backslashreplace` as it does in the default locale-aware mode)
- On Unix, `os.device_encoding()` returns `'utf-8'` rather than the device encoding.

Note that the standard stream settings in UTF-8 mode can be overridden by `PYTHONIOENCODING` (just as they can be in the default locale-aware mode).

As a consequence of the changes in those lower level APIs, other higher level APIs also exhibit different default behaviours:

- Command line arguments, environment variables and filenames are decoded to text using the UTF-8 encoding.
- `os.fsdecode()` and `os.fsencode()` use the UTF-8 encoding.
- `open()`, `io.open()`, and `codecs.open()` use the UTF-8 encoding by default. However, they still use the strict error handler by default so that attempting to open a binary file in text mode is likely to raise an exception rather than producing nonsense data.

The Python UTF-8 Mode is enabled if the LC_CTYPE locale is `C` or `POSIX` at Python startup (see the `PyConfig_Read()` function).

It can be enabled or disabled using the `-X utf8` command line option and the `PYTHONUTF8` environment variable.

If the `PYTHONUTF8` environment variable is not set at all, then the interpreter defaults to using the current locale settings, *unless* the current locale is identified as a legacy ASCII-based locale (as described for `PYTHONCOERCECLOCALE`), and locale coercion is either disabled or fails. In such legacy locales, the interpreter will default to enabling UTF-8 mode unless explicitly instructed not to do so.

The Python UTF-8 Mode can only be enabled at the Python startup. Its value can be read from `sys.flags.utf8_mode`.

See also the UTF-8 mode on Windows and the filesystem encoding and error handler.

See also

PEP 686 **[https://peps.python.org/pep-0686/]**
Python 3.15 will make Python UTF-8 Mode default.

Process Parameters

These functions and data items provide information and operate on the current process and user.

os.ctermid()

Return the filename corresponding to the controlling terminal of the process.

Availability: Unix, not WASI.

os.environ

A mapping object where keys and values are strings that represent the process environment. For example, `environ['HOME']` is the pathname of your home directory (on some platforms), and is equivalent to `getenv("HOME")` in C.

This mapping is captured the first time the **`os`** module is imported, typically during Python startup as part of processing `site.py`. Changes to the environment made after this time are not reflected in **`os.environ`**, except for changes made by modifying **`os.environ`** directly.

This mapping may be used to modify the environment as well as query the environment. **`putenv()`** will be called automatically when the mapping is modified.

On Unix, keys and values use **`sys.getfilesystemencoding()`** and `'surrogateescape'` error handler. Use **`environb`** if you would like to use a different encoding.

On Windows, the keys are converted to uppercase. This also applies when getting, setting, or deleting an item. For example, `environ['monty'] = 'python'` maps the key `'MONTY'` to the value `'python'`.

Note

Calling **`putenv()`** directly does not change **`os.environ`**, so it’s better to modify **`os.environ`**.

Note

On some platforms, including FreeBSD and macOS, setting `environ` may cause memory leaks. Refer to the system documentation for **`putenv()`**.

You can delete items in this mapping to unset environment variables. `unsetenv()` will be called automatically when an item is deleted from `os.environ`, and when one of the **`pop()`** or **`clear()`** methods is called.

See also

The `os.reload_environ()` function.

Changed in version 3.9: Updated to support **PEP 584** [https://peps.python.org/pep-0584/]'s merge (`|`) and update (`|=`) operators.

os.environb

Bytes version of `environ`: a mapping object where both keys and values are `bytes` objects representing the process environment. `environ` and `environb` are synchronized (modifying `environb` updates `environ`, and vice versa).

`environb` is only available if `supports_bytes_environ` is `True`.

Added in version 3.2.

Changed in version 3.9: Updated to support **PEP 584** [https://peps.python.org/pep-0584/]'s merge (`|`) and update (`|=`) operators.

os.reload_environ()

The `os.environ` and `os.environb` mappings are a cache of environment variables at the time that Python started. As such, changes to the current process environment are not reflected if made outside Python, or by `os.putenv()` or `os.unsetenv()`. Use **`os.reload_environ()`** to update `os.environ` and `os.environb` with any such changes to the current process environment.

Warning

This function is not thread-safe. Calling it while the environment is being modified in an other thread is an undefined behavior. Reading from `os.environ` or `os.environb`, or calling `os.getenv()` while reloading, may return an empty result.

Added in version 3.14.

os.chdir(*path*)
os.fchdir(*fd*)
os.getcwd()

These functions are described in Files and Directories.

os.fsencode(*filename*)

Encode path-like *filename* to the filesystem encoding and error handler; return `bytes` unchanged.

`fsdecode()` is the reverse function.

Added in version 3.2.

Changed in version 3.6: Support added to accept objects implementing the `os.PathLike` interface.

os.fsdecode(*filename*)

Decode the path-like *filename* from the filesystem encoding and error handler; return `str` unchanged.

`fsencode()` is the reverse function.

Added in version 3.2.

Changed in version 3.6: Support added to accept objects implementing the `os.PathLike` interface.

os.fspath(*path*)

Return the file system representation of the path.

If **`str`** or **`bytes`** is passed in, it is returned unchanged. Otherwise **`__fspath__()`** is called and its value is returned as long as it is a **`str`** or **`bytes`** object. In all other cases, **`TypeError`** is raised.

Added in version 3.6.

class os.PathLike

An abstract base class for objects representing a file system path, e.g. **`pathlib.PurePath`**.

Added in version 3.6.

abstractmethod __fspath__()

Return the file system path representation of the object.

The method should only return a **`str`** or **`bytes`** object, with the preference being for **`str`**.

os.getenv(*key*, *default=None*)

Return the value of the environment variable *key* as a string if it exists, or *default* if it doesn’t. *key* is a string. Note that since **`getenv()`** uses **`os.environ`**, the mapping of **`getenv()`** is similarly also captured on import, and the function may not reflect future environment changes.

On Unix, keys and values are decoded with **`sys.getfilesystemencoding()`** and `'surrogateescape'` error handler. Use **`os.getenvb()`** if you would like to use a different encoding.

Availability: Unix, Windows.

os.getenvb(*key*, *default=None*)

Return the value of the environment variable *key* as bytes if it exists, or *default* if it doesn't. *key* must be bytes. Note that since `getenvb()` uses `os.environb`, the mapping of `getenvb()` is similarly also captured on import, and the function may not reflect future environment changes.

`getenvb()` is only available if `supports_bytes_environ` is `True`.

Availability: Unix.

Added in version 3.2.

os.get_exec_path(*env=None*)

Returns the list of directories that will be searched for a named executable, similar to a shell, when launching a process. *env*, when specified, should be an environment variable dictionary to lookup the PATH in. By default, when *env* is `None`, `environ` is used.

Added in version 3.2.

os.getegid()

Return the effective group id of the current process. This corresponds to the "set id" bit on the file being executed in the current process.

Availability: Unix, not WASI.

os.geteuid()

Return the current process's effective user id.

Availability: Unix, not WASI.

os.getgid()

Return the real group id of the current process.

Availability: Unix.

The function is a stub on WASI, see WebAssembly platforms for more information.

os.getgrouplist(*user*, *group*, /)

Return list of group ids that *user* belongs to. If *group* is not in the list, it is included; typically, *group* is specified as the group ID field from the password record for *user*, because that group ID will otherwise be potentially omitted.

Availability: Unix, not WASI.

Added in version 3.3.

os.getgroups()

Return list of supplemental group ids associated with the current process.

Availability: Unix, not WASI.

Note

On macOS, **`getgroups()`** behavior differs somewhat from other Unix platforms. If the Python interpreter was built with a deployment target of `10.5` or earlier, **`getgroups()`** returns the list of effective group ids associated with the current user process; this list is limited to a system-defined number of entries, typically 16, and may be modified by calls to **`setgroups()`** if suitably privileged. If built with a deployment target greater than `10.5`, **`getgroups()`** returns the current group access list for the user associated with the effective user id of the process; the group access list may change over the lifetime of the process, it is not affected by calls to `setgroups()`, and its length is not limited to 16. The deployment target value, **`MACOSX_DEPLOYMENT_TARGET`**, can be obtained with `sysconfig.get_config_var()`.

os.getlogin()

Return the name of the user logged in on the controlling terminal of the process. For most purposes, it is more useful to use `getpass.getuser()` since the latter checks the environment variables **`LOGNAME`** or **`USERNAME`** to find out who the user is, and falls back to `pwd.getpwuid(os.getuid())[0]` to get the login name of the current real user id.

Availability: Unix, Windows, not WASI.

os.getpgid(*pid*)

Return the process group id of the process with process id *pid*. If *pid* is 0, the process group id of the current process is returned.

Availability: Unix, not WASI.

os.getpgrp()

Return the id of the current process group.

Availability: Unix, not WASI.

os.getpid()

Return the current process id.

The function is a stub on WASI, see WebAssembly platforms for more information.

os.getppid()

Return the parent's process id. When the parent process has exited, on Unix the id returned is the one of the init process (1), on Windows it is still the same id, which may be already reused by another process.

Availability: Unix, Windows, not WASI.

Changed in version 3.2: Added support for Windows.

os.getpriority(*which*, *who*)

Get program scheduling priority. The value *which* is one of `PRIO_PROCESS`, `PRIO_PGRP`, or `PRIO_USER`, and *who* is interpreted relative to *which* (a process identifier for `PRIO_PROCESS`, process group identifier for `PRIO_PGRP`, and a user ID for `PRIO_USER`). A zero value for *who* denotes (respectively) the calling process, the process group of the calling process, or the real user ID of the calling process.

Availability: Unix, not WASI.

Added in version 3.3.

os.PRIO_PROCESS
os.PRIO_PGRP
os.PRIO_USER

Parameters for the `getpriority()` and `setpriority()` functions.

Availability: Unix, not WASI.

Added in version 3.3.

os.PRIO_DARWIN_THREAD
os.PRIO_DARWIN_PROCESS
os.PRIO_DARWIN_BG
os.PRIO_DARWIN_NONUI

Parameters for the **`getpriority()`** and **`setpriority()`** functions.

Availability: macOS

Added in version 3.12.

os.getresuid()

Return a tuple (ruid, euid, suid) denoting the current process’s real, effective, and saved user ids.

Availability: Unix, not WASI.

Added in version 3.2.

os.getresgid()

Return a tuple (rgid, egid, sgid) denoting the current process’s real, effective, and saved group ids.

Availability: Unix, not WASI.

Added in version 3.2.

os.getuid()

Return the current process’s real user id.

Availability: Unix.

The function is a stub on WASI, see WebAssembly platforms for more information.

os.initgroups(*username*, *gid*, */*)

Call the system `initgroups()` to initialize the group access list with all of the groups of which the specified username is a member, plus the specified group id.

Availability: Unix, not WASI, not Android.

Added in version 3.2.

os.putenv(*key*, *value*, /)

Set the environment variable named *key* to the string *value*. Such changes to the environment affect subprocesses started with `os.system()`, `popen()` or `fork()` and `execv()`.

Assignments to items in `os.environ` are automatically translated into corresponding calls to `putenv()`; however, calls to `putenv()` don't update `os.environ`, so it is actually preferable to assign to items of `os.environ`. This also applies to `getenv()` and `getenvb()`, which respectively use `os.environ` and `os.environb` in their implementations.

See also the `os.reload_environ()` function.

Note

On some platforms, including FreeBSD and macOS, setting `environ` may cause memory leaks. Refer to the system documentation for **`putenv()`**.

Raises an auditing event `os.putenv` with arguments `key`, `value`.

Changed in version 3.9: The function is now always available.

os.setegid(*egid*, /)

Set the current process's effective group id.

Availability: Unix, not WASI, not Android.

os.seteuid(*euid*, /)

Set the current process's effective user id.

Availability: Unix, not WASI, not Android.

os.setgid(*gid*, /)

Set the current process’ group id.

Availability: Unix, not WASI, not Android.

os.setgroups(*groups*, /)

Set the list of supplemental group ids associated with the current process to *groups*. *groups* must be a sequence, and each element must be an integer identifying a group. This operation is typically available only to the superuser.

Availability: Unix, not WASI.

Note

On macOS, the length of *groups* may not exceed the system-defined maximum number of effective group ids, typically 16. See the documentation for **`getgroups()`** for cases where it may not return the same group list set by calling setgroups().

os.setns(*fd*, *nstype=0*)

Reassociate the current thread with a Linux namespace. See the *setns(2)* *[https://manpages.debian.org/setns(2)]* and *namespaces(7)* *[https://manpages.debian.org/namespaces(7)]* man pages for more details.

If *fd* refers to a `/proc/pid/ns/` link, `setns()` reassociates the calling thread with the namespace associated with that link, and *nstype* may be set to one of the CLONE_NEW* constants to impose constraints on the operation (`0` means no constraints).

Since Linux 5.8, *fd* may refer to a PID file descriptor obtained from `pidfd_open()`. In this case, `setns()` reassociates the calling thread into one or more of the same namespaces as the thread referred to by *fd*. This is subject to any constraints imposed by *nstype*, which is a bit mask combining one or more of the CLONE_NEW* constants, e.g. `setns(fd, os.CLONE_NEWUTS | os.CLONE_NEWPID)`. The caller's memberships in unspecified namespaces are left unchanged.

fd can be any object with a `fileno()` method, or a raw file descriptor.

This example reassociates the thread with the `init` process's network namespace:

```
fd = os.open("/proc/1/ns/net", os.O_RDONLY)
os.setns(fd, os.CLONE_NEWNET)
os.close(fd)
```

Availability: Linux >= 3.0 with glibc >= 2.14.

Added in version 3.12.

See also

The `unshare()` function.

os.setpgrp()

Call the system call **`setpgrp()`** or `setpgrp(0, 0)` depending on which version is implemented (if any). See the Unix manual for the semantics.

Availability: Unix, not WASI.

os.setpgid(*pid*, *pgrp*, */*)

Call the system call **`setpgid()`** to set the process group id of the process with id *pid* to the process group with id *pgrp*. See the Unix manual for the semantics.

Availability: Unix, not WASI.

os.setpriority(*which*, *who*, *priority*)

Set program scheduling priority. The value *which* is one of **`PRIO_PROCESS`**, **`PRIO_PGRP`**, or **`PRIO_USER`**, and *who* is interpreted relative to *which* (a process identifier for **`PRIO_PROCESS`**, process group identifier for **`PRIO_PGRP`**, and a user ID for **`PRIO_USER`**). A zero value for *who* denotes (respectively) the calling process, the process group of the calling process, or the real user ID of the calling process. *priority* is a value in the range -20 to 19. The default priority is 0; lower priorities cause more favorable scheduling.

Availability: Unix, not WASI.

Added in version 3.3.

os.setregid(*rgid*, *egid*, */*)

Set the current process’s real and effective group ids.

Availability: Unix, not WASI, not Android.

os.setresgid(*rgid*, *egid*, *sgid*, */*)

Set the current process’s real, effective, and saved group ids.

Availability: Unix, not WASI, not Android.

Added in version 3.2.

os.setresuid(*ruid*, *euid*, *suid*, */*)

Set the current process’s real, effective, and saved user ids.

Availability: Unix, not WASI, not Android.

Added in version 3.2.

os.setreuid(*ruid*, *euid*, /)

Set the current process’s real and effective user ids.

Availability: Unix, not WASI, not Android.

os.getsid(*pid*, /)

Call the system call **`getsid()`**. See the Unix manual for the semantics.

Availability: Unix, not WASI.

os.setsid()

Call the system call **`setsid()`**. See the Unix manual for the semantics.

Availability: Unix, not WASI.

os.setuid(*uid*, /)

Set the current process’s user id.

Availability: Unix, not WASI, not Android.

os.strerror(*code*, /)

Return the error message corresponding to the error code in *code*. On platforms where **`strerror()`** returns `NULL` when given an unknown error number, **`ValueError`** is raised.

os.supports_bytes_environ

`True` if the native OS type of the environment is bytes (eg. `False` on Windows).

Added in version 3.2.

os.umask(*mask*, /)

Set the current numeric umask and return the previous umask.

The function is a stub on WASI, see WebAssembly platforms for more information.

os.uname()

Returns information identifying the current operating system. The return value is an object with five attributes:

- **`sysname`** - operating system name
- **`nodename`** - name of machine on network (implementation-defined)
- **`release`** - operating system release
- **`version`** - operating system version
- **`machine`** - hardware identifier

For backwards compatibility, this object is also iterable, behaving like a five-tuple containing **`sysname`**, **`nodename`**, **`release`**, **`version`**, and **`machine`** in that order.

Some systems truncate **`nodename`** to 8 characters or to the leading component; a better way to get the hostname is **`socket.gethostname()`** or even `socket.gethostbyaddr(socket.gethostname())`.

On macOS, iOS and Android, this returns the *kernel* name and version (i.e., `'Darwin'` on macOS and iOS; `'Linux'` on Android). **`platform.uname()`** can be used to get the user-facing operating system name and version on iOS and Android.

Availability: Unix.

Changed in version 3.3: Return type changed from a tuple to a tuple-like object with named attributes.

os.unsetenv(*key*, /)

Unset (delete) the environment variable named *key*. Such changes to the environment affect subprocesses started with **`os.system()`**, **`popen()`** or **`fork()`** and **`execv()`**.

Deletion of items in **`os.environ`** is automatically translated into a corresponding call to **`unsetenv()`**; however, calls to **`unsetenv()`** don't update **`os.environ`**, so it is actually preferable to delete items of **`os.environ`**.

See also the **`os.reload_environ()`** function.

Raises an auditing event `os.unsetenv` with argument `key`.

Changed in version 3.9: The function is now always available and is also available on Windows.

os.unshare(*flags*)

Disassociate parts of the process execution context, and move them into a newly created namespace. See the *unshare(2)* *[https://manpages.debian.org/unshare(2)]* man page for more details. The *flags* argument is a bit mask, combining zero or more of the CLONE_* constants, that specifies which parts of the execution context should be unshared from their existing associations and moved to a new namespace. If the *flags* argument is `0`, no changes are made to the calling process's execution context.

Availability: Linux >= 2.6.16.

Added in version 3.12.

See also

The `setns()` function.

Flags to the `unshare()` function, if the implementation supports them. See *unshare(2)* *[https://manpages.debian.org/unshare(2)]* in the Linux manual for their exact effect and availability.

os.CLONE_FILES
os.CLONE_FS
os.CLONE_NEWCGROUP
os.CLONE_NEWIPC
os.CLONE_NEWNET
os.CLONE_NEWNS
os.CLONE_NEWPID
os.CLONE_NEWTIME
os.CLONE_NEWUSER
os.CLONE_NEWUTS
os.CLONE_SIGHAND
os.CLONE_SYSVSEM
os.CLONE_THREAD
os.CLONE_VM

File Object Creation

These functions create new file objects. (See also `open()` for opening file descriptors.)

os.fdopen(*fd*, **args*, ***kwargs*)

Return an open file object connected to the file descriptor *fd*. This is an alias of the `open()` built-in function and accepts the same arguments. The only difference is that the first argument of `fdopen()` must always be an integer.

File Descriptor Operations

These functions operate on I/O streams referenced using file descriptors.

File descriptors are small integers corresponding to a file that has been opened by the current process. For example, standard input is usually file descriptor 0, standard output is 1, and standard error is 2. Further files opened by a process will then be assigned 3, 4, 5, and so forth. The name “file descriptor” is slightly deceptive; on Unix platforms, sockets and pipes are also referenced by file descriptors.

The `fileno()` method can be used to obtain the file descriptor associated with a file object when required. Note that using the file descriptor directly will bypass the file object methods, ignoring aspects such as internal buffering of data.

os.close(*fd*)

Close file descriptor *fd*.

Note

This function is intended for low-level I/O and must be applied to a file descriptor as returned by `os.open()` or `pipe()`. To close a “file object” returned by the built-in function `open()` or by `popen()` or `fdopen()`, use its `close()` method.

os.closerange(*fd_low*, *fd_high*, */*)

Close all file descriptors from *fd_low* (inclusive) to *fd_high* (exclusive), ignoring errors. Equivalent to (but much faster than):

```
for fd in range(fd_low, fd_high):
    try:
        os.close(fd)
    except OSError:
        pass
```

os.copy_file_range(*src*, *dst*, *count*, *offset_src=None*, *offset_dst=None*)

Copy *count* bytes from file descriptor *src*, starting from offset *offset_src*, to file descriptor *dst*, starting from offset *offset_dst*. If *offset_src* is `None`, then *src* is read from the current position; respectively for *offset_dst*.

In Linux kernel older than 5.3, the files pointed to by *src* and *dst* must reside in the same filesystem, otherwise an `OSError` is raised with `errno` set to `errno.EXDEV`.

This copy is done without the additional cost of transferring data from the kernel to user space and then back into the kernel. Additionally, some filesystems could implement extra optimizations, such as the use of reflinks (i.e., two or more inodes that share pointers to the same copy-on-write disk blocks; supported file systems include btrfs and XFS) and server-side copy (in the case of NFS).

The function copies bytes between two file descriptors. Text options, like the encoding and the line ending, are ignored.

The return value is the amount of bytes copied. This could be less than the amount requested.

Note

On Linux, `os.copy_file_range()` should not be used for copying a range of a pseudo file from a special filesystem like procfs and sysfs. It will always copy no bytes and return 0 as if the file was empty because of a known Linux kernel issue.

Availability: Linux >= 4.5 with glibc >= 2.27.

Added in version 3.8.

os.device_encoding(*fd*)

Return a string describing the encoding of the device associated with *fd* if it is connected to a terminal; else return **`None`**.

On Unix, if the Python UTF-8 Mode is enabled, return `'UTF-8'` rather than the device encoding.

Changed in version 3.10: On Unix, the function now implements the Python UTF-8 Mode.

os.dup(*fd*, */*)

Return a duplicate of file descriptor *fd*. The new file descriptor is non-inheritable.

On Windows, when duplicating a standard stream (0: stdin, 1: stdout, 2: stderr), the new file descriptor is inheritable.

Availability: not WASI.

Changed in version 3.4: The new file descriptor is now non-inheritable.

os.dup2(*fd*, *fd2*, *inheritable=True*)

Duplicate file descriptor *fd* to *fd2*, closing the latter first if necessary. Return *fd2*. The new file descriptor is inheritable by default or non-inheritable if *inheritable* is `False`.

Availability: not WASI.

Changed in version 3.4: Add the optional *inheritable* parameter.

Changed in version 3.7: Return *fd2* on success. Previously, `None` was always returned.

os.fchmod(*fd*, *mode*)

Change the mode of the file given by *fd* to the numeric *mode*. See the docs for **`chmod()`** for possible values of *mode*. As of Python 3.3, this is equivalent to `os.chmod(fd, mode)`.

Raises an auditing event `os.chmod` with arguments `path`, `mode`, `dir_fd`.

Availability: Unix, Windows.

The function is limited on WASI, see WebAssembly platforms for more information.

Changed in version 3.13: Added support on Windows.

os.fchown(*fd*, *uid*, *gid*)

Change the owner and group id of the file given by *fd* to the numeric *uid* and *gid*. To leave one of the ids unchanged, set it to -1. See **`chown()`**. As of Python 3.3, this is equivalent to `os.chown(fd, uid, gid)`.

Raises an auditing event `os.chown` with arguments `path`, `uid`, `gid`, `dir_fd`.

Availability: Unix.

The function is limited on WASI, see WebAssembly platforms for more information.

os.fdatasync(*fd*)

Force write of file with filedescriptor *fd* to disk. Does not force update of metadata.

Availability: Unix.

Note

This function is not available on MacOS.

os.fpathconf(*fd*, *name*, */*)

Return system configuration information relevant to an open file. *name* specifies the configuration value to retrieve; it may be a string which is the name of a defined system value; these names are specified in a number of standards (POSIX.1, Unix 95, Unix 98, and others). Some platforms define additional names as well. The names known to the host operating system are given in the `pathconf_names` dictionary. For configuration variables not included in that mapping, passing an integer for *name* is also accepted.

If *name* is a string and is not known, **`ValueError`** is raised. If a specific value for *name* is not supported by the host system, even if it is included in `pathconf_names`, an **`OSError`** is raised with **`errno.EINVAL`** for the error number.

As of Python 3.3, this is equivalent to `os.pathconf(fd, name)`.

Availability: Unix.

os.fstat(*fd*)

Get the status of the file descriptor *fd*. Return a **`stat_result`** object.

As of Python 3.3, this is equivalent to `os.stat(fd)`.

See also

The **`stat()`** function.

os.fstatvfs(*fd*, */*)

Return information about the filesystem containing the file associated with file descriptor *fd*, like **`statvfs()`**. As of Python 3.3, this is equivalent to `os.statvfs(fd)`.

Availability: Unix.

os.fsync(*fd*)

Force write of file with filedescriptor *fd* to disk. On Unix, this calls the native **`fsync()`** function; on Windows, the MS **`_commit()`** function.

If you’re starting with a buffered Python file object *f*, first do `f.flush()`, and then do `os.fsync(f.fileno())`, to ensure that all internal buffers associated with *f* are written to disk.

Availability: Unix, Windows.

os.ftruncate(*fd*, *length*, */*)

Truncate the file corresponding to file descriptor *fd*, so that it is at most *length* bytes in size. As of Python 3.3, this is equivalent to `os.truncate(fd, length)`.

Raises an auditing event `os.truncate` with arguments `fd`, `length`.

Availability: Unix, Windows.

Changed in version 3.5: Added support for Windows

os.get_blocking(*fd*, /)

Get the blocking mode of the file descriptor: `False` if the **`O_NONBLOCK`** flag is set, `True` if the flag is cleared.

See also **`set_blocking()`** and **`socket.socket.setblocking()`**.

Availability: Unix, Windows.

The function is limited on WASI, see WebAssembly platforms for more information.

On Windows, this function is limited to pipes.

Added in version 3.5.

Changed in version 3.12: Added support for pipes on Windows.

os.grantpt(*fd*, /)

Grant access to the slave pseudo-terminal device associated with the master pseudo-terminal device to which the file descriptor *fd* refers. The file descriptor *fd* is not closed upon failure.

Calls the C standard library function **`grantpt()`**.

Availability: Unix, not WASI.

Added in version 3.13.

os.isatty(*fd*, /)

Return `True` if the file descriptor *fd* is open and connected to a tty(-like) device, else `False`.

os.lockf(*fd*, *cmd*, *len*, */*)

Apply, test or remove a POSIX lock on an open file descriptor. *fd* is an open file descriptor. *cmd* specifies the command to use - one of **`F_LOCK`**, **`F_TLOCK`**, **`F_ULOCK`** or **`F_TEST`**. *len* specifies the section of the file to lock.

Raises an auditing event `os.lockf` with arguments `fd`, `cmd`, `len`.

Availability: Unix.

Added in version 3.3.

os.F_LOCK
os.F_TLOCK
os.F_ULOCK
os.F_TEST

Flags that specify what action **`lockf()`** will take.

Availability: Unix.

Added in version 3.3.

os.login_tty(*fd*, */*)

Prepare the tty of which fd is a file descriptor for a new login session. Make the calling process a session leader; make the tty the controlling tty, the stdin, the stdout, and the stderr of the calling process; close fd.

Availability: Unix, not WASI.

Added in version 3.11.

os.lseek(*fd*, *pos*, *whence*, */*)

Set the current position of file descriptor *fd* to position *pos*, modified by *whence*, and return the new position in bytes relative to the start of the file. Valid values for *whence* are:

- **`SEEK_SET`** or `0` – set *pos* relative to the beginning of the file
- **`SEEK_CUR`** or `1` – set *pos* relative to the current file position
- **`SEEK_END`** or `2` – set *pos* relative to the end of the file
- **`SEEK_HOLE`** – set *pos* to the next data location, relative to *pos*
- **`SEEK_DATA`** – set *pos* to the next data hole, relative to *pos*

Changed in version 3.3: Add support for **`SEEK_HOLE`** and **`SEEK_DATA`**.

os.SEEK_SET
os.SEEK_CUR
os.SEEK_END

Parameters to the `lseek()` function and the `seek()` method on file-like objects, for whence to adjust the file position indicator.

`SEEK_SET`

Adjust the file position relative to the beginning of the file.

`SEEK_CUR`

Adjust the file position relative to the current file position.

`SEEK_END`

Adjust the file position relative to the end of the file.

Their values are 0, 1, and 2, respectively.

os.SEEK_HOLE

os.SEEK_DATA

Parameters to the `lseek()` function and the `seek()` method on file-like objects, for seeking file data and holes on sparsely allocated files.

`SEEK_DATA`

Adjust the file offset to the next location containing data, relative to the seek position.

`SEEK_HOLE`

Adjust the file offset to the next location containing a hole, relative to the seek position. A hole is defined as a sequence of zeros.

Note

These operations only make sense for filesystems that support them.

Availability: Linux >= 3.1, macOS, Unix

Added in version 3.3.

os.open(*path*, *flags*, *mode=0o777*, *, *dir_fd=None*)

Open the file *path* and set various flags according to *flags* and possibly its mode according to *mode*. When computing *mode*, the current umask value is first masked out. Return the file descriptor for the newly opened file. The new file descriptor is non-inheritable.

For a description of the flag and mode values, see the C run-time documentation; flag constants (like `O_RDONLY` and `O_WRONLY`) are defined in the `os` module. In particular, on Windows adding `O_BINARY` is needed to open files in binary mode.

This function can support paths relative to directory descriptors with the *dir_fd* parameter.

Raises an auditing event `open` with arguments `path`, `mode`, `flags`.

Changed in version 3.4: The new file descriptor is now non-inheritable.

Note

This function is intended for low-level I/O. For normal usage, use the built-in function **`open()`**, which returns a file object with **`read()`** and **`write()`** methods (and many more). To wrap a file descriptor in a file object, use **`fdopen()`**.

Changed in version 3.3: Added the *dir_fd* parameter.

Changed in version 3.5: If the system call is interrupted and the signal handler does not raise an exception, the function now retries the system call instead of raising an **`InterruptedError`** exception (see **PEP 475** [https://peps.python.org/pep-0475/] for the rationale).

Changed in version 3.6: Accepts a path-like object.

The following constants are options for the *flags* parameter to the **`open()`** function. They can be combined using the bitwise OR operator `|`. Some of them are not available on all platforms. For descriptions of their availability and use, consult the *open(2)* *[https://manpages.debian.org/open(2)]* manual page on Unix or the MSDN [https://msdn.microsoft.com/en-us/library/z0kc8e3z.aspx] on Windows.

os.O_RDONLY
os.O_WRONLY

os.O_RDWR
os.O_APPEND
os.O_CREAT
os.O_EXCL
os.O_TRUNC

The above constants are available on Unix and Windows.

os.O_DSYNC
os.O_RSYNC
os.O_SYNC
os.O_NDELAY
os.O_NONBLOCK
os.O_NOCTTY
os.O_CLOEXEC

The above constants are only available on Unix.

Changed in version 3.3: Add `O_CLOEXEC` constant.

os.O_BINARY
os.O_NOINHERIT
os.O_SHORT_LIVED
os.O_TEMPORARY
os.O_RANDOM
os.O_SEQUENTIAL
os.O_TEXT

The above constants are only available on Windows.

os.O_EVTONLY
os.O_FSYNC
os.O_SYMLINK
os.O_NOFOLLOW_ANY

The above constants are only available on macOS.

Changed in version 3.10: Add `O_EVTONLY`, `O_FSYNC`, `O_SYMLINK` and `O_NOFOLLOW_ANY` constants.

os.O_ASYNC
os.O_DIRECT
os.O_DIRECTORY
os.O_NOFOLLOW
os.O_NOATIME
os.O_PATH
os.O_TMPFILE
os.O_SHLOCK
os.O_EXLOCK

The above constants are extensions and not present if they are not defined by the C library.

Changed in version 3.4: Add **`O_PATH`** on systems that support it. Add **`O_TMPFILE`**, only available on Linux Kernel 3.11 or newer.

os.openpty()

Open a new pseudo-terminal pair. Return a pair of file descriptors `(master, slave)` for the pty and the tty, respectively. The new file descriptors are non-inheritable. For a (slightly) more portable approach, use the **`pty`** module.

Availability: Unix, not WASI.

Changed in version 3.4: The new file descriptors are now non-inheritable.

os.pipe()

Create a pipe. Return a pair of file descriptors `(r, w)` usable for reading and writing, respectively. The new file descriptor is non-inheritable.

Availability: Unix, Windows.

Changed in version 3.4: The new file descriptors are now non-inheritable.

os.pipe2(*flags*, /)

Create a pipe with *flags* set atomically. *flags* can be constructed by ORing together one or more of these values: **`O_NONBLOCK`**, **`O_CLOEXEC`**. Return a pair of file descriptors `(r, w)` usable for reading and writing, respectively.

Availability: Unix, not WASI.

Added in version 3.3.

os.posix_fallocate(*fd*, *offset*, *len*, /)

Ensures that enough disk space is allocated for the file specified by *fd* starting from *offset* and continuing for *len* bytes.

Availability: Unix.

Added in version 3.3.

os.posix_fadvise(*fd*, *offset*, *len*, *advice*, /)

Announces an intention to access data in a specific pattern thus allowing the kernel to make optimizations. The advice applies to the region of the file specified by *fd* starting at *offset* and continuing for *len* bytes. *advice* is one of **`POSIX_FADV_NORMAL`**, **`POSIX_FADV_SEQUENTIAL`**, **`POSIX_FADV_RANDOM`**, **`POSIX_FADV_NOREUSE`**, **`POSIX_FADV_WILLNEED`** or **`POSIX_FADV_DONTNEED`**.

Availability: Unix.

Added in version 3.3.

os.POSIX_FADV_NORMAL
os.POSIX_FADV_SEQUENTIAL
os.POSIX_FADV_RANDOM
os.POSIX_FADV_NOREUSE
os.POSIX_FADV_WILLNEED
os.POSIX_FADV_DONTNEED

Flags that can be used in *advice* in **`posix_fadvise()`** that specify the access pattern that is likely to be used.

Availability: Unix.

Added in version 3.3.

os.pread(*fd*, *n*, *offset*, /)

Read at most *n* bytes from file descriptor *fd* at a position of *offset*, leaving the file offset unchanged.

Return a bytestring containing the bytes read. If the end of the file referred to by *fd* has been reached, an empty bytes object is returned.

Availability: Unix.

Added in version 3.3.

os.posix_openpt(*oflag*, /)

Open and return a file descriptor for a master pseudo-terminal device.

Calls the C standard library function **`posix_openpt()`**. The *oflag* argument is used to set file status flags and file access modes as specified in the manual page of **`posix_openpt()`** of your system.

The returned file descriptor is non-inheritable. If the value **`O_CLOEXEC`** is available on the system, it is added to *oflag*.

Availability: Unix, not WASI.

Added in version 3.13.

os.preadv(*fd*, *buffers*, *offset*, *flags=0*, */*)

Read from a file descriptor *fd* at a position of *offset* into mutable bytes-like objects *buffers*, leaving the file offset unchanged. Transfer data into each buffer until it is full and then move on to the next buffer in the sequence to hold the rest of the data.

The flags argument contains a bitwise OR of zero or more of the following flags:

- **`RWF_HIPRI`**
- **`RWF_NOWAIT`**

Return the total number of bytes actually read which can be less than the total capacity of all the objects.

The operating system may set a limit (**`sysconf()`** value `'SC_IOV_MAX'`) on the number of buffers that can be used.

Combine the functionality of **`os.readv()`** and **`os.pread()`**.

Availability: Linux >= 2.6.30, FreeBSD >= 6.0, OpenBSD >= 2.7, AIX >= 7.1.

Using flags requires Linux >= 4.6.

Added in version 3.7.

os.RWF_NOWAIT

Do not wait for data which is not immediately available. If this flag is specified, the system call will return instantly if it would have to read data from the backing storage or wait for a lock.

If some data was successfully read, it will return the number of bytes read. If no bytes were read, it will return `-1` and set errno to **`errno.EAGAIN`**.

Availability: Linux >= 4.14.

Added in version 3.7.

os.RWF_HIPRI

High priority read/write. Allows block-based filesystems to use polling of the device, which provides lower latency, but may use additional resources.

Currently, on Linux, this feature is usable only on a file descriptor opened using the **`O_DIRECT`** flag.

Availability: Linux >= 4.6.

Added in version 3.7.

os.ptsname(*fd*, */*)

Return the name of the slave pseudo-terminal device associated with the master pseudo-terminal device to which the file descriptor *fd* refers. The file descriptor *fd* is not closed upon failure.

Calls the reentrant C standard library function **`ptsname_r()`** if it is available; otherwise, the C standard library function **`ptsname()`**, which is not guaranteed to be thread-safe, is called.

Availability: Unix, not WASI.

Added in version 3.13.

os.pwrite(*fd*, *str*, *offset*, /)

Write the bytestring in *str* to file descriptor *fd* at position of *offset*, leaving the file offset unchanged.

Return the number of bytes actually written.

Availability: Unix.

Added in version 3.3.

os.pwritev(*fd*, *buffers*, *offset*, *flags=0*, /)

Write the *buffers* contents to file descriptor *fd* at an offset *offset*, leaving the file offset unchanged. *buffers* must be a sequence of bytes-like objects. Buffers are processed in array order. Entire contents of the first buffer is written before proceeding to the second, and so on.

The flags argument contains a bitwise OR of zero or more of the following flags:

- **`RWF_DSYNC`**
- **`RWF_SYNC`**
- **`RWF_APPEND`**

Return the total number of bytes actually written.

The operating system may set a limit (**`sysconf()`** value `'SC_IOV_MAX'`) on the number of buffers that can be used.

Combine the functionality of **`os.writev()`** and **`os.pwrite()`**.

Availability: Linux >= 2.6.30, FreeBSD >= 6.0, OpenBSD >= 2.7, AIX >= 7.1.

Using flags requires Linux >= 4.6.

Added in version 3.7.

os.RWF_DSYNC

Provide a per-write equivalent of the `O_DSYNC` `os.open()` flag. This flag effect applies only to the data range written by the system call.

Availability: Linux >= 4.7.

Added in version 3.7.

os.RWF_SYNC

Provide a per-write equivalent of the `O_SYNC` `os.open()` flag. This flag effect applies only to the data range written by the system call.

Availability: Linux >= 4.7.

Added in version 3.7.

os.RWF_APPEND

Provide a per-write equivalent of the `O_APPEND` `os.open()` flag. This flag is meaningful only for `os.pwritev()`, and its effect applies only to the data range written by the system call. The *offset* argument does not affect the write operation; the data is always appended to the end of the file. However, if the *offset* argument is `-1`, the current file *offset* is updated.

Availability: Linux >= 4.16.

Added in version 3.10.

os.read(*fd*, *n*, /)

Read at most *n* bytes from file descriptor *fd*.

Return a bytestring containing the bytes read. If the end of the file referred to by *fd* has been reached, an empty bytes object is returned.

Note

This function is intended for low-level I/O and must be applied to a file descriptor as returned by `os.open()` or `pipe()`. To read a “file object” returned by the built-in function `open()` or by `popen()` or `fdopen()`, or `sys.stdin`, use its **`read()`** or **`readline()`** methods.

Changed in version 3.5: If the system call is interrupted and the signal handler does not raise an exception, the function now retries the system call instead of raising an `InterruptedError` exception (see **PEP 475** [https://peps.python.org/pep-0475/] for the rationale).

os.readinto(*fd*, *buffer*, */*)

Read from a file descriptor *fd* into a mutable buffer object *buffer*.

The *buffer* should be mutable and bytes-like. On success, returns the number of bytes read. Less bytes may be read than the size of the buffer. The underlying system call will be retried when interrupted by a signal, unless the signal handler raises an exception. Other errors will not be retried and an error will be raised.

Returns 0 if *fd* is at end of file or if the provided *buffer* has length 0 (which can be used to check for errors without reading data). Never returns negative.

Note

This function is intended for low-level I/O and must be applied to a file descriptor as returned by `os.open()` or `os.pipe()`. To read a “file object” returned by the built-in function `open()`, or `sys.stdin`, use its member functions, for example `io.BufferedIOBase.readinto()`, `io.BufferedIOBase.read()`, or `io.TextIOBase.read()`

Added in version 3.14.

os.sendfile(*out_fd*, *in_fd*, *offset*, *count*)
os.sendfile(*out_fd*, *in_fd*, *offset*, *count*, *headers=()*, *trailers=()*, *flags=0*)

Copy *count* bytes from file descriptor *in_fd* to file descriptor *out_fd* starting at *offset*. Return the number of bytes sent. When EOF is reached return `0`.

The first function notation is supported by all platforms that define `sendfile()`.

On Linux, if *offset* is given as `None`, the bytes are read from the current position of *in_fd* and the position of *in_fd* is updated.

The second case may be used on macOS and FreeBSD where *headers* and *trailers* are arbitrary sequences of buffers that are written before and after the data from *in_fd* is written. It returns the same as the first case.

On macOS and FreeBSD, a value of `0` for *count* specifies to send until the end of *in_fd* is reached.

All platforms support sockets as *out_fd* file descriptor, and some platforms allow other types (e.g. regular file, pipe) as well.

Cross-platform applications should not use *headers*, *trailers* and *flags* arguments.

Availability: Unix, not WASI.

Note

For a higher-level wrapper of `sendfile()`, see `socket.socket.sendfile()`.

Added in version 3.3.

Changed in version 3.9: Parameters *out* and *in* was renamed to *out_fd* and *in_fd*.

os.SF_NODISKIO
os.SF_MNOWAIT
os.SF_SYNC

Parameters to the `sendfile()` function, if the implementation supports them.

Availability: Unix, not WASI.

Added in version 3.3.

os.SF_NOCACHE

Parameter to the `sendfile()` function, if the implementation supports it. The data won't be cached in the virtual memory and will be freed afterwards.

Availability: Unix, not WASI.

Added in version 3.11.

os.set_blocking(*fd*, *blocking*, */*)

Set the blocking mode of the specified file descriptor. Set the `O_NONBLOCK` flag if blocking is `False`, clear the flag otherwise.

See also `get_blocking()` and `socket.socket.setblocking()`.

Availability: Unix, Windows.

The function is limited on WASI, see WebAssembly platforms for more information.

On Windows, this function is limited to pipes.
Added in version 3.5.

Changed in version 3.12: Added support for pipes on Windows.

os.splice(*src*, *dst*, *count*, *offset_src=None*, *offset_dst=None*, *flags=0*)

Transfer *count* bytes from file descriptor *src*, starting from offset *offset_src*, to file descriptor *dst*, starting from offset *offset_dst*.

The splicing behaviour can be modified by specifying a *flags* value. Any of the following variables may used, combined using bitwise OR (the `|` operator):

- If **`SPLICE_F_MOVE`** is specified, the kernel is asked to move pages instead of copying, but pages may still be copied if the kernel cannot move the pages from the pipe.
- If **`SPLICE_F_NONBLOCK`** is specified, the kernel is asked to not block on I/O. This makes the splice pipe operations nonblocking, but splice may nevertheless block because the spliced file descriptors may block.
- If **`SPLICE_F_MORE`** is specified, it hints to the kernel that more data will be coming in a subsequent splice.

At least one of the file descriptors must refer to a pipe. If *offset_src* is `None`, then *src* is read from the current position; respectively for *offset_dst*. The offset associated to the file descriptor that refers to a pipe must be `None`. The files pointed to by *src* and *dst* must reside in the same filesystem, otherwise an **`OSError`** is raised with **`errno`** set to **`errno.EXDEV`**.

This copy is done without the additional cost of transferring data from the kernel to user space and then back into the kernel. Additionally, some filesystems could implement extra optimizations. The copy is done as if both files are opened as binary.

Upon successful completion, returns the number of bytes spliced to or from the pipe. A return value of 0 means end of input. If *src* refers to a pipe, then this means that there was no data to transfer, and it would not make sense to block because there are no writers connected to the write end of the pipe.

See also

The *splice(2)* *[https://manpages.debian.org/splice(2)]* man page.

Availability: Linux >= 2.6.17 with glibc >= 2.5

Added in version 3.10.

os.SPLICE_F_MOVE
os.SPLICE_F_NONBLOCK
os.SPLICE_F_MORE

Added in version 3.10.

os.readv(*fd*, *buffers*, */*)

Read from a file descriptor *fd* into a number of mutable bytes-like objects *buffers*. Transfer data into each buffer until it is full and then move on to the next buffer in the sequence to hold the rest of the data.

Return the total number of bytes actually read which can be less than the total capacity of all the objects.

The operating system may set a limit (`sysconf()` value `'SC_IOV_MAX'`) on the number of buffers that can be used.

Availability: Unix.

Added in version 3.3.

os.tcgetpgrp(*fd*, /)

Return the process group associated with the terminal given by *fd* (an open file descriptor as returned by `os.open()`).

Availability: Unix, not WASI.

os.tcsetpgrp(*fd*, *pg*, /)

Set the process group associated with the terminal given by *fd* (an open file descriptor as returned by `os.open()`) to *pg*.

Availability: Unix, not WASI.

os.ttyname(*fd*, /)

Return a string which specifies the terminal device associated with file descriptor *fd*. If *fd* is not associated with a terminal device, an exception is raised.

Availability: Unix.

os.unlockpt(*fd*, /)

Unlock the slave pseudo-terminal device associated with the master pseudo-terminal device to which the file descriptor *fd* refers. The file descriptor *fd* is not closed upon failure.

Calls the C standard library function **`unlockpt()`**.

Availability: Unix, not WASI.

Added in version 3.13.

os.write(*fd*, *str*, /)

Write the bytestring in *str* to file descriptor *fd*.

Return the number of bytes actually written.

Note

This function is intended for low-level I/O and must be applied to a file descriptor as returned by `os.open()` or `pipe()`. To write a “file object” returned by the built-in function `open()` or by `popen()` or `fdopen()`, or `sys.stdout` or `sys.stderr`, use its `write()` method.

Changed in version 3.5: If the system call is interrupted and the signal handler does not raise an exception, the function now retries the system call instead of raising an `InterruptedError` exception (see **PEP 475** [https://peps.python.org/pep-0475/] for the rationale).

os.writev(*fd*, *buffers*, /)

Write the contents of *buffers* to file descriptor *fd*. *buffers* must be a sequence of bytes-like objects. Buffers are processed in array order. Entire contents of the first buffer is written before proceeding to the second, and so on.

Returns the total number of bytes actually written.

The operating system may set a limit (`sysconf()` value `'SC_IOV_MAX'`) on the number of buffers that can be used.

Availability: Unix.

Added in version 3.3.

Querying the size of a terminal

Added in version 3.3.

os.get_terminal_size(*fd=STDOUT_FILENO*, */*)

Return the size of the terminal window as `(columns, lines)`, tuple of type **`terminal_size`**.

The optional argument `fd` (default `STDOUT_FILENO`, or standard output) specifies which file descriptor should be queried.

If the file descriptor is not connected to a terminal, an **`OSError`** is raised.

`shutil.get_terminal_size()` is the high-level function which should normally be used, `os.get_terminal_size` is the low-level implementation.

Availability: Unix, Windows.

class os.terminal_size

A subclass of tuple, holding `(columns, lines)` of the terminal window size.

columns

Width of the terminal window in characters.

lines

Height of the terminal window in characters.

Inheritance of File Descriptors

Added in version 3.4.

A file descriptor has an “inheritable” flag which indicates if the file descriptor can be inherited by child processes. Since Python 3.4, file descriptors created by Python are non-inheritable by default.

On UNIX, non-inheritable file descriptors are closed in child processes at the execution of a new program, other file descriptors are inherited. Note that non-inheritable file descriptors are still *inherited* by child processes on `os.fork()`.

On Windows, non-inheritable handles and file descriptors are closed in child processes, except for standard streams (file descriptors 0, 1 and 2: stdin, stdout and stderr), which are always inherited. Using `spawn*` functions, all inheritable handles and all inheritable file descriptors are inherited. Using the `subprocess` module, all file descriptors except standard streams are closed, and inheritable handles are only inherited if the *close_fds* parameter is `False`.

On WebAssembly platforms, the file descriptor cannot be modified.

os.get_inheritable(*fd*, */*)

Get the “inheritable” flag of the specified file descriptor (a boolean).

os.set_inheritable(*fd*, *inheritable*, */*)

Set the “inheritable” flag of the specified file descriptor.

os.get_handle_inheritable(*handle*, */*)

Get the “inheritable” flag of the specified handle (a boolean).

Availability: Windows.

os.set_handle_inheritable(*handle*, *inheritable*, */*)

Set the “inheritable” flag of the specified handle.

Availability: Windows.

Files and Directories

On some Unix platforms, many of these functions support one or more of these features:

- **specifying a file descriptor:** Normally the *path* argument provided to functions in the `os` module must be a string specifying a file path. However, some functions now alternatively accept an open file descriptor for their *path* argument. The function will then operate on the file referred to by the descriptor. For POSIX systems, Python will call the variant of the function prefixed with `f` (e.g. call `fchdir` instead of `chdir`).

  You can check whether or not *path* can be specified as a file descriptor for a particular function on your platform using `os.supports_fd`. If this functionality is unavailable, using it will raise a `NotImplementedError`.

  If the function also supports *dir_fd* or *follow_symlinks* arguments, it’s an error to specify one of those when supplying *path* as a file descriptor.

- **paths relative to directory descriptors:** If *dir_fd* is not `None`, it should be a file descriptor referring to a directory, and the path to operate on should be relative; path will then be relative to that directory. If the path is absolute, *dir_fd* is ignored. For POSIX systems, Python will call the variant of the function with an `at` suffix and possibly prefixed with `f` (e.g. call `faccessat` instead of `access`).

You can check whether or not *dir_fd* is supported for a particular function on your platform using `os.supports_dir_fd`. If it’s unavailable, using it will raise a `NotImplementedError`.

- **not following symlinks:** If *follow_symlinks* is `False`, and the last element of the path to operate on is a symbolic link, the function will operate on the symbolic link itself rather than the file pointed to by the link. For POSIX systems, Python will call the `l...` variant of the function.

  You can check whether or not *follow_symlinks* is supported for a particular function on your platform using `os.supports_follow_symlinks`. If it’s unavailable, using it will raise a `NotImplementedError`.

os.access(*path*, *mode*, *, *dir_fd=None*, *effective_ids=False*, *follow_symlinks=True*)

Use the real uid/gid to test for access to *path*. Note that most operations will use the effective uid/gid, therefore this routine can be used in a suid/sgid environment to test if the invoking user has the specified access to *path*. *mode* should be `F_OK` to test the existence of *path*, or it can be the inclusive OR of one or more of `R_OK`, `W_OK`, and `X_OK` to test permissions. Return `True` if access is allowed, `False` if not. See the Unix man page *access(2)* *[https://manpages.debian.org/access(2)]* for more information.

This function can support specifying paths relative to directory descriptors and not following symlinks.

If *effective_ids* is `True`, `access()` will perform its access checks using the effective uid/gid instead of the real uid/gid. *effective_ids* may not be supported on your platform; you can check whether or not it is available using `os.supports_effective_ids`. If it is unavailable, using it will raise a `NotImplementedError`.

Note

Using `access()` to check if a user is authorized to e.g. open a file before actually doing so using `open()` creates a security hole, because the user might exploit the short time interval between checking and opening the file to manipulate it. It's preferable to use EAFP techniques. For example:

```
if os.access("myfile", os.R_OK):
    with open("myfile") as fp:
        return fp.read()
return "some default data"
```

is better written as:

```
try:
    fp = open("myfile")
except PermissionError:
    return "some default data"
else:
    with fp:
        return fp.read()
```

Note

I/O operations may fail even when `access()` indicates that they would succeed, particularly for operations on network filesystems which may have permissions semantics beyond the usual POSIX permission-bit model.

Changed in version 3.3: Added the *dir_fd*, *effective_ids*, and *follow_symlinks* parameters.

Changed in version 3.6: Accepts a path-like object.

os.F_OK
os.R_OK
os.W_OK
os.X_OK

Values to pass as the *mode* parameter of **`access()`** to test the existence, readability, writability and executability of *path*, respectively.

os.chdir(*path*)

Change the current working directory to *path*.

This function can support specifying a file descriptor. The descriptor must refer to an opened directory, not an open file.

This function can raise **`OSError`** and subclasses such as **`FileNotFoundError`**, **`PermissionError`**, and **`NotADirectoryError`**.

Raises an auditing event `os.chdir` with argument `path`.

Changed in version 3.3: Added support for specifying *path* as a file descriptor on some platforms.

Changed in version 3.6: Accepts a path-like object.

os.chflags(*path*, *flags*, *, *follow_symlinks=True*)

Set the flags of *path* to the numeric *flags*. *flags* may take a combination (bitwise OR) of the following values (as defined in the **`stat`** module):

- **`stat.UF_NODUMP`**
- **`stat.UF_IMMUTABLE`**
- **`stat.UF_APPEND`**
- **`stat.UF_OPAQUE`**

- `stat.UF_NOUNLINK`
- `stat.UF_COMPRESSED`
- `stat.UF_HIDDEN`
- `stat.SF_ARCHIVED`
- `stat.SF_IMMUTABLE`
- `stat.SF_APPEND`
- `stat.SF_NOUNLINK`
- `stat.SF_SNAPSHOT`

This function can support not following symlinks.

Raises an auditing event `os.chflags` with arguments `path`, `flags`.

Availability: Unix, not WASI.

Changed in version 3.3: Added the *follow_symlinks* parameter.

Changed in version 3.6: Accepts a path-like object.

os.chmod(*path*, *mode*, *, *dir_fd=None*, *follow_symlinks=True*)

Change the mode of *path* to the numeric *mode*. *mode* may take one of the following values (as defined in the `stat` module) or bitwise ORed combinations of them:

- `stat.S_ISUID`
- `stat.S_ISGID`
- `stat.S_ENFMT`
- `stat.S_ISVTX`
- `stat.S_IREAD`
- `stat.S_IWRITE`
- `stat.S_IEXEC`
- `stat.S_IRWXU`
- `stat.S_IRUSR`
- `stat.S_IWUSR`

- `stat.S_IXUSR`
- `stat.S_IRWXG`
- `stat.S_IRGRP`
- `stat.S_IWGRP`
- `stat.S_IXGRP`
- `stat.S_IRWXO`
- `stat.S_IROTH`
- `stat.S_IWOTH`
- `stat.S_IXOTH`

This function can support specifying a file descriptor, paths relative to directory descriptors and not following symlinks.

Note

Although Windows supports `chmod()`, you can only set the file's read-only flag with it (via the `stat.S_IWRITE` and `stat.S_IREAD` constants or a corresponding integer value). All other bits are ignored. The default value of *follow_symlinks* is `False` on Windows.

The function is limited on WASI, see WebAssembly platforms for more information.

Raises an auditing event `os.chmod` with arguments `path`, `mode`, `dir_fd`.

Changed in version 3.3: Added support for specifying *path* as an open file descriptor, and the *dir_fd* and *follow_symlinks* arguments.

Changed in version 3.6: Accepts a path-like object.

Changed in version 3.13: Added support for a file descriptor and the *follow_symlinks* argument on Windows.

os.chown(*path*, *uid*, *gid*, *, *dir_fd=None*, *follow_symlinks=True*)

Change the owner and group id of *path* to the numeric *uid* and *gid*. To leave one of the ids unchanged, set it to -1.

This function can support specifying a file descriptor, paths relative to directory descriptors and not following symlinks.

See **`shutil.chown()`** for a higher-level function that accepts names in addition to numeric ids.

Raises an auditing event `os.chown` with arguments `path`, `uid`, `gid`, `dir_fd`.

Availability: Unix.

The function is limited on WASI, see WebAssembly platforms for more information.

Changed in version 3.3: Added support for specifying *path* as an open file descriptor, and the *dir_fd* and *follow_symlinks* arguments.

Changed in version 3.6: Supports a path-like object.

os.chroot(*path*)

Change the root directory of the current process to *path*.

Availability: Unix, not WASI, not Android.

Changed in version 3.6: Accepts a path-like object.

os.fchdir(*fd*)

Change the current working directory to the directory represented by the file descriptor *fd*. The descriptor must refer to an opened directory, not an open file. As of Python 3.3, this is equivalent to `os.chdir(fd)`.

Raises an auditing event `os.chdir` with argument `path`.

Availability: Unix.

os.getcwd()

Return a string representing the current working directory.

os.getcwdb()

Return a bytestring representing the current working directory.

Changed in version 3.8: The function now uses the UTF-8 encoding on Windows, rather than the ANSI code page: see **PEP 529** [https://peps.python.org/pep-0529/] for the rationale. The function is no longer deprecated on Windows.

os.lchflags(*path*, *flags*)

Set the flags of *path* to the numeric *flags*, like **`chflags()`**, but do not follow symbolic links. As of Python 3.3, this is equivalent to `os.chflags(path, flags, follow_symlinks=False)`.

Raises an auditing event `os.chflags` with arguments `path`, `flags`.

Availability: Unix, not WASI.

Changed in version 3.6: Accepts a path-like object.

os.lchmod(*path*, *mode*)

Change the mode of *path* to the numeric *mode*. If path is a symlink, this affects the symlink rather than the target. See the docs for **`chmod()`** for possible values of *mode*. As of Python 3.3, this is equivalent to `os.chmod(path, mode, follow_symlinks=False)`.

`lchmod()` is not part of POSIX, but Unix implementations may have it if changing the mode of symbolic links is supported.

Raises an auditing event `os.chmod` with arguments `path`, `mode`, `dir_fd`.

Availability: Unix, Windows, not Linux, FreeBSD >= 1.3, NetBSD >= 1.3, not OpenBSD

Changed in version 3.6: Accepts a path-like object.

Changed in version 3.13: Added support on Windows.

os.lchown(*path*, *uid*, *gid*)

Change the owner and group id of *path* to the numeric *uid* and *gid*. This function will not follow symbolic links. As of Python 3.3, this is equivalent to `os.chown(path, uid, gid, follow_symlinks=False)`.

Raises an auditing event `os.chown` with arguments `path`, `uid`, `gid`, `dir_fd`.

Availability: Unix.

Changed in version 3.6: Accepts a path-like object.

os.link(*src*, *dst*, ***, *src_dir_fd=None*, *dst_dir_fd=None*, *follow_symlinks=True*)

Create a hard link pointing to *src* named *dst*.

This function can support specifying *src_dir_fd* and/or *dst_dir_fd* to supply paths relative to directory descriptors, and not following symlinks. The default value of *follow_symlinks* is `False` on Windows.

Raises an auditing event `os.link` with arguments `src`, `dst`, `src_dir_fd`, `dst_dir_fd`.

Availability: Unix, Windows.

Changed in version 3.2: Added Windows support.

Changed in version 3.3: Added the *src_dir_fd*, *dst_dir_fd*, and *follow_symlinks* parameters.

Changed in version 3.6: Accepts a path-like object for *src* and *dst*.

os.listdir(*path*='.')

Return a list containing the names of the entries in the directory given by *path*. The list is in arbitrary order, and does not include the special entries `'.'` and `'..'` even if they are present in the directory. If a file is removed from or added to the directory during the call of this function, whether a name for that file be included is unspecified.

path may be a path-like object. If *path* is of type `bytes` (directly or indirectly through the **`PathLike`** interface), the filenames returned will also be of type `bytes`; in all other circumstances, they will be of type `str`.

This function can also support specifying a file descriptor; the file descriptor must refer to a directory.

Raises an auditing event `os.listdir` with argument `path`.

Note

To encode `str` filenames to `bytes`, use **`fsencode()`**.

See also

The **`scandir()`** function returns directory entries along with file attribute information, giving better performance for many common use cases.

Changed in version 3.2: The *path* parameter became optional.

Changed in version 3.3: Added support for specifying *path* as an open file descriptor.

Changed in version 3.6: Accepts a path-like object.

os.listdrives()

Return a list containing the names of drives on a Windows system.

A drive name typically looks like `'C:\\'`. Not every drive name will be associated with a volume, and some may be inaccessible for a variety of reasons, including permissions, network connectivity or missing media. This function does not test for access.

May raise **`OSError`** if an error occurs collecting the drive names.

Raises an auditing event `os.listdrives` with no arguments.

Availability: Windows

Added in version 3.12.

os.listmounts(*volume*)

Return a list containing the mount points for a volume on a Windows system.

volume must be represented as a GUID path, like those returned by **`os.listvolumes()`**. Volumes may be mounted in multiple locations or not at all. In the latter case, the list will be empty. Mount points that are not associated with a volume will not be returned by this function.

The mount points return by this function will be absolute paths, and may be longer than the drive name.

Raises **`OSError`** if the volume is not recognized or if an error occurs collecting the paths.

Raises an auditing event `os.listmounts` with argument `volume`.

Availability: Windows

Added in version 3.12.

os.listvolumes()

Return a list containing the volumes in the system.

Volumes are typically represented as a GUID path that looks like `\\?\Volume{xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx}\`. Files can usually be accessed through a GUID path, permissions allowing. However, users are generally not familiar with them, and so the recommended use of this function is to retrieve mount points using **`os.listmounts()`**.

May raise **`OSError`** if an error occurs collecting the volumes.

Raises an auditing event `os.listvolumes` with no arguments.

Availability: Windows

Added in version 3.12.

os.lstat(*path*, *, *dir_fd=None*)

Perform the equivalent of an **`lstat()`** system call on the given path. Similar to **`stat()`**, but does not follow symbolic links. Return a **`stat_result`** object.

On platforms that do not support symbolic links, this is an alias for **`stat()`**.

As of Python 3.3, this is equivalent to `os.stat(path, dir_fd=dir_fd, follow_symlinks=False)`.

This function can also support paths relative to directory descriptors.

See also

The **`stat()`** function.

Changed in version 3.2: Added support for Windows 6.0 (Vista) symbolic links.

Changed in version 3.3: Added the *dir_fd* parameter.

Changed in version 3.6: Accepts a path-like object.

Changed in version 3.8: On Windows, now opens reparse points that represent another path (name surrogates), including symbolic links and directory junctions. Other kinds of reparse points are resolved by the operating system as for **`stat()`**.

os.mkdir(*path*, *mode=0o777*, ***, *dir_fd=None*)

Create a directory named *path* with numeric mode *mode*.

If the directory already exists, **`FileExistsError`** is raised. If a parent directory in the path does not exist, **`FileNotFoundError`** is raised.

On some systems, *mode* is ignored. Where it is used, the current umask value is first masked out. If bits other than the last 9 (i.e. the last 3 digits of the octal representation of the *mode*) are set, their meaning is platform-dependent. On some platforms, they are ignored and you should call **`chmod()`** explicitly to set them.

On Windows, a *mode* of `0o700` is specifically handled to apply access control to the new directory such that only the current user and administrators have access. Other values of *mode* are ignored.

This function can also support paths relative to directory descriptors.

It is also possible to create temporary directories; see the **`tempfile`** module's **`tempfile.mkdtemp()`** function.

Raises an auditing event `os.mkdir` with arguments `path`, `mode`, `dir_fd`.

Changed in version 3.3: Added the *dir_fd* parameter.

Changed in version 3.6: Accepts a path-like object.

Changed in version 3.13: Windows now handles a *mode* of `0o700`.

os.makedirs(*name*, *mode=0o777*, *exist_ok=False*)

Recursive directory creation function. Like **`mkdir()`**, but makes all intermediate-level directories needed to contain the leaf directory.

The *mode* parameter is passed to **`mkdir()`** for creating the leaf directory; see the mkdir() description for how it is interpreted. To set the file permission bits of any newly created parent directories you can set the umask before invoking **`makedirs()`**. The file permission bits of existing parent directories are not changed.

If *exist_ok* is `False` (the default), a **`FileExistsError`** is raised if the target directory already exists.

Note

`makedirs()` will become confused if the path elements to create include **`pardir`** (eg. ".." on UNIX systems).

This function handles UNC paths correctly.

Raises an auditing event `os.mkdir` with arguments `path`, `mode`, `dir_fd`.

Changed in version 3.2: Added the *exist_ok* parameter.

Changed in version 3.4.1: Before Python 3.4.1, if *exist_ok* was `True` and the directory existed, **`makedirs()`** would still raise an error if *mode* did not match the mode of the existing directory. Since this behavior was impossible to implement safely, it was removed in Python 3.4.1. See bpo-21082 [https://bugs.python.org/issue?@action=redirect&bpo=21082].

Changed in version 3.6: Accepts a path-like object.

Changed in version 3.7: The *mode* argument no longer affects the file permission bits of newly created intermediate-level directories.

os.mkfifo(*path*, *mode=0o666*, *, *dir_fd=None*)

Create a FIFO (a named pipe) named *path* with numeric mode *mode*. The current umask value is first masked out from the mode.

This function can also support paths relative to directory descriptors.

FIFOs are pipes that can be accessed like regular files. FIFOs exist until they are deleted (for example with **`os.unlink()`**). Generally, FIFOs are used as rendezvous between "client" and "server" type processes: the server opens the FIFO for reading, and the client opens it for writing. Note that **`mkfifo()`** doesn't open the FIFO — it just creates the rendezvous point.

Availability: Unix, not WASI.

Changed in version 3.3: Added the *dir_fd* parameter.

Changed in version 3.6: Accepts a path-like object.

os.mknod(*path*, *mode=0o600*, *device=0*, *, *dir_fd=None*)

Create a filesystem node (file, device special file or named pipe) named *path*. *mode* specifies both the permissions to use and the type of node to be created, being combined (bitwise OR) with one of `stat.S_IFREG`, `stat.S_IFCHR`, `stat.S_IFBLK`, and `stat.S_IFIFO` (those constants are available in **`stat`**). For `stat.S_IFCHR` and `stat.S_IFBLK`, *device* defines the newly created device special file (probably using **`os.makedev()`**), otherwise it is ignored.

This function can also support paths relative to directory descriptors.

Availability: Unix, not WASI.

Changed in version 3.3: Added the *dir_fd* parameter.

Changed in version 3.6: Accepts a path-like object.

os.major(*device*, */*)

Extract the device major number from a raw device number (usually the **`st_dev`** or **`st_rdev`** field from **`stat`**).

os.minor(*device*, */*)

Extract the device minor number from a raw device number (usually the **`st_dev`** or **`st_rdev`** field from **`stat`**).

os.makedev(*major*, *minor*, */*)

Compose a raw device number from the major and minor device numbers.

os.pathconf(*path*, *name*)

Return system configuration information relevant to a named file. *name* specifies the configuration value to retrieve; it may be a string which is the name of a defined system value; these names are specified in a number of standards (POSIX.1, Unix 95, Unix 98, and others). Some platforms define additional names as well. The names known to the host operating system are given in the `pathconf_names` dictionary. For configuration variables not included in that mapping, passing an integer for *name* is also accepted.

If *name* is a string and is not known, **`ValueError`** is raised. If a specific value for *name* is not supported by the host system, even if it is included in `pathconf_names`, an **`OSError`** is raised with **`errno.EINVAL`** for the error number.

This function can support specifying a file descriptor.

Availability: Unix.

Changed in version 3.6: Accepts a path-like object.

os.pathconf_names

Dictionary mapping names accepted by **`pathconf()`** and **`fpathconf()`** to the integer values defined for those names by the host operating system. This can be used to determine the set of names known to the system.

Availability: Unix.

os.readlink(*path*, *, *dir_fd=None*)

Return a string representing the path to which the symbolic link points. The result may be either an absolute or relative pathname; if it is relative, it may be converted to an absolute pathname using `os.path.join(os.path.dirname(path), result)`.

If the *path* is a string object (directly or indirectly through a **`PathLike`** interface), the result will also be a string object, and the call may raise a UnicodeDecodeError. If the *path* is a bytes object (direct or indirectly), the result will be a bytes object.

This function can also support paths relative to directory descriptors.

When trying to resolve a path that may contain links, use **`realpath()`** to properly handle recursion and platform differences.

Availability: Unix, Windows.

Changed in version 3.2: Added support for Windows 6.0 (Vista) symbolic links.

Changed in version 3.3: Added the *dir_fd* parameter.

Changed in version 3.6: Accepts a path-like object on Unix.

Changed in version 3.8: Accepts a path-like object and a bytes object on Windows.

Added support for directory junctions, and changed to return the substitution path (which typically includes `\\?\` prefix) rather than the optional “print name” field that was previously returned.

os.remove(*path*, ***, *dir_fd=None*)

Remove (delete) the file *path*. If *path* is a directory, an **`OSError`** is raised. Use **`rmdir()`** to remove directories. If the file does not exist, a **`FileNotFoundError`** is raised.

This function can support paths relative to directory descriptors.

On Windows, attempting to remove a file that is in use causes an exception to be raised; on Unix, the directory entry is removed but the storage allocated to the file is not made available until the original file is no longer in use.

This function is semantically identical to **`unlink()`**.

Raises an auditing event `os.remove` with arguments `path`, `dir_fd`.

Changed in version 3.3: Added the *dir_fd* parameter.

Changed in version 3.6: Accepts a path-like object.

os.removedirs(*name*)

Remove directories recursively. Works like **`rmdir()`** except that, if the leaf directory is successfully removed, **`removedirs()`** tries to successively remove every parent directory mentioned in *path* until an error is raised (which is ignored, because it generally means that a parent directory is not empty). For example, `os.removedirs('foo/bar/baz')` will first remove the directory `'foo/bar/baz'`, and then remove `'foo/bar'` and `'foo'` if they are empty. Raises **`OSError`** if the leaf directory could not be successfully removed.

Raises an auditing event `os.remove` with arguments `path`, `dir_fd`.

Changed in version 3.6: Accepts a path-like object.

os.rename(*src*, *dst*, ***, *src_dir_fd=None*, *dst_dir_fd=None*)

Rename the file or directory *src* to *dst*. If *dst* exists, the operation will fail with an **`OSError`** subclass in a number of cases:

On Windows, if *dst* exists a **`FileExistsError`** is always raised. The operation may fail if *src* and *dst* are on different filesystems. Use **`shutil.move()`** to support moves to a different filesystem.

On Unix, if *src* is a file and *dst* is a directory or vice-versa, an **`IsADirectoryError`** or a **`NotADirectoryError`** will be raised respectively. If both are directories and *dst* is empty, *dst* will be silently replaced. If *dst* is a non-empty directory, an **`OSError`** is raised. If both are files, *dst* will be replaced silently if the user has permission. The operation may fail on some Unix flavors if *src* and *dst* are on different filesystems. If successful, the renaming will be an atomic operation (this is a POSIX requirement).

This function can support specifying *src_dir_fd* and/or *dst_dir_fd* to supply paths relative to directory descriptors.

If you want cross-platform overwriting of the destination, use **`replace()`**.

Raises an auditing event `os.rename` with arguments `src`, `dst`, `src_dir_fd`, `dst_dir_fd`.

Changed in version 3.3: Added the *src_dir_fd* and *dst_dir_fd* parameters.

Changed in version 3.6: Accepts a path-like object for *src* and *dst*.

os.renames(*old*, *new*)

Recursive directory or file renaming function. Works like **`rename()`**, except creation of any intermediate directories needed to make the new pathname good is attempted first. After the rename, directories corresponding to rightmost path segments of the old name will be pruned away using **`removedirs()`**.

Note

This function can fail with the new directory structure made if you lack permissions needed to remove the leaf directory or file.

Raises an auditing event `os.rename` with arguments `src`, `dst`, `src_dir_fd`, `dst_dir_fd`.

Changed in version 3.6: Accepts a path-like object for *old* and *new*.

os.replace(*src*, *dst*, *, *src_dir_fd=None*, *dst_dir_fd=None*)

Rename the file or directory *src* to *dst*. If *dst* is a non-empty directory, **`OSError`** will be raised. If *dst* exists and is a file, it will be replaced silently if the user has permission. The operation may fail if *src* and *dst* are on different filesystems. If successful, the renaming will be an atomic operation (this is a POSIX requirement).

This function can support specifying *src_dir_fd* and/or *dst_dir_fd* to supply paths relative to directory descriptors.

Raises an auditing event `os.rename` with arguments `src`, `dst`, `src_dir_fd`, `dst_dir_fd`.

Added in version 3.3.

Changed in version 3.6: Accepts a path-like object for *src* and *dst*.

os.rmdir(*path*, *, *dir_fd=None*)

Remove (delete) the directory *path*. If the directory does not exist or is not empty, a **`FileNotFoundError`** or an **`OSError`** is raised respectively. In order to remove whole directory trees, **`shutil.rmtree()`** can be used.

This function can support paths relative to directory descriptors.

Raises an auditing event `os.rmdir` with arguments `path`, `dir_fd`.

Changed in version 3.3: Added the *dir_fd* parameter.

Changed in version 3.6: Accepts a path-like object.

os.scandir(*path*='.')

Return an iterator of **`os.DirEntry`** objects corresponding to the entries in the directory given by *path*. The entries are yielded in arbitrary order, and the special entries `'.'` and `'..'` are not included. If a file is removed from or added to the directory after creating the iterator, whether an entry for that file be included is unspecified.

Using **`scandir()`** instead of **`listdir()`** can significantly increase the performance of code that also needs file type or file attribute information, because **`os.DirEntry`** objects expose this information if the operating system provides it when scanning a directory. All **`os.DirEntry`** methods may perform a system call, but **`is_dir()`** and **`is_file()`** usually only require a system call for symbolic links; **`os.DirEntry.stat()`** always requires a system call on Unix but only requires one for symbolic links on Windows.

path may be a path-like object. If *path* is of type `bytes` (directly or indirectly through the **`PathLike`** interface), the type of the **`name`** and **`path`** attributes of each **`os.DirEntry`** will be `bytes`; in all other circumstances, they will be of type `str`.

This function can also support specifying a file descriptor; the file descriptor must refer to a directory.

Raises an auditing event `os.scandir` with argument `path`.

The **`scandir()`** iterator supports the context manager protocol and has the following method:

scandir.close()

Close the iterator and free acquired resources.

This is called automatically when the iterator is exhausted or garbage collected, or when an error happens during iterating. However it is advisable to call it explicitly or use the **`with`** statement.

Added in version 3.6.

The following example shows a simple use of **`scandir()`** to display all the files (excluding directories) in the given *path* that don't start with `'.'`. The `entry.is_file()` call will generally not make an additional system call:

```
with os.scandir(path) as it:
    for entry in it:
        if not entry.name.startswith('.') and
entry.is_file():
            print(entry.name)
```

Note

On Unix-based systems, **`scandir()`** uses the system's opendir() [https://pubs.opengroup.org/onlinepubs/009695399/functions/opendir.html] and readdir() [https://pubs.opengroup.org/onlinepubs/009695399/functions/readdir_r.html] functions. On Windows, it uses the Win32 FindFirstFileW [https://msdn.microsoft.com/en-us/library/windows/desktop/aa364418(v=vs.85).aspx] and FindNextFileW [https://msdn.microsoft.com/en-us/library/windows/desktop/aa364428(v=vs.85).aspx] functions.

Added in version 3.5.

Changed in version 3.6: Added support for the context manager protocol and the **`close()`** method. If a **`scandir()`** iterator is neither exhausted nor explicitly closed a **`ResourceWarning`** will be emitted in its destructor.

The function accepts a path-like object.

Changed in version 3.7: Added support for file descriptors on Unix.

class os.DirEntry

Object yielded by **`scandir()`** to expose the file path and other file attributes of a directory entry.

`scandir()` will provide as much of this information as possible without making additional system calls. When a `stat()` or `lstat()` system call is made, the `os.DirEntry` object will cache the result.

`os.DirEntry` instances are not intended to be stored in long-lived data structures; if you know the file metadata has changed or if a long time has elapsed since calling **`scandir()`**, call `os.stat(entry.path)` to fetch up-to-date information.

Because the `os.DirEntry` methods can make operating system calls, they may also raise **`OSError`**. If you need very fine-grained control over errors, you can catch **`OSError`** when calling one of the `os.DirEntry` methods and handle as appropriate.

To be directly usable as a path-like object, `os.DirEntry` implements the **`PathLike`** interface.

Attributes and methods on a `os.DirEntry` instance are as follows:

name

The entry’s base filename, relative to the `scandir()` *path* argument.

The `name` attribute will be `bytes` if the `scandir()` *path* argument is of type `bytes` and `str` otherwise. Use `fsdecode()` to decode byte filenames.

path

The entry’s full path name: equivalent to `os.path.join(scandir_path, entry.name)` where *scandir_path* is the `scandir()` *path* argument. The path is only absolute if the `scandir()` *path* argument was absolute. If the `scandir()` *path* argument was a file descriptor, the `path` attribute is the same as the `name` attribute.

The `path` attribute will be `bytes` if the `scandir()` *path* argument is of type `bytes` and `str` otherwise. Use `fsdecode()` to decode byte filenames.

inode()

Return the inode number of the entry.

The result is cached on the `os.DirEntry` object. Use `os.stat(entry.path, follow_symlinks=False).st_ino` to fetch up-to-date information.

On the first, uncached call, a system call is required on Windows but not on Unix.

is_dir(*, *follow_symlinks=True*)

Return `True` if this entry is a directory or a symbolic link pointing to a directory; return `False` if the entry is or points to any other kind of file, or if it doesn’t exist anymore.

If *follow_symlinks* is `False`, return `True` only if this entry is a directory (without following symlinks); return `False` if the entry is any other kind of file or if it doesn't exist anymore.

The result is cached on the `os.DirEntry` object, with a separate cache for *follow_symlinks* `True` and `False`. Call **`os.stat()`** along with **`stat.S_ISDIR()`** to fetch up-to-date information.

On the first, uncached call, no system call is required in most cases. Specifically, for non-symlinks, neither Windows or Unix require a system call, except on certain Unix file systems, such as network file systems, that return `dirent.d_type == DT_UNKNOWN`. If the entry is a symlink, a system call will be required to follow the symlink unless *follow_symlinks* is `False`.

This method can raise **`OSError`**, such as **`PermissionError`**, but **`FileNotFoundError`** is caught and not raised.

is_file(*, *follow_symlinks=True*)

Return `True` if this entry is a file or a symbolic link pointing to a file; return `False` if the entry is or points to a directory or other non-file entry, or if it doesn't exist anymore.

If *follow_symlinks* is `False`, return `True` only if this entry is a file (without following symlinks); return `False` if the entry is a directory or other non-file entry, or if it doesn't exist anymore.

The result is cached on the `os.DirEntry` object. Caching, system calls made, and exceptions raised are as per **`is_dir()`**.

is_symlink()

Return `True` if this entry is a symbolic link (even if broken); return `False` if the entry points to a directory or any kind of file, or if it doesn’t exist anymore.

The result is cached on the `os.DirEntry` object. Call **`os.path.islink()`** to fetch up-to-date information.

On the first, uncached call, no system call is required in most cases. Specifically, neither Windows or Unix require a system call, except on certain Unix file systems, such as network file systems, that return `dirent.d_type == DT_UNKNOWN`.

This method can raise **`OSError`**, such as **`PermissionError`**, but **`FileNotFoundError`** is caught and not raised.

is_junction()

Return `True` if this entry is a junction (even if broken); return `False` if the entry points to a regular directory, any kind of file, a symlink, or if it doesn’t exist anymore.

The result is cached on the `os.DirEntry` object. Call **`os.path.isjunction()`** to fetch up-to-date information.

Added in version 3.12.

stat(*, *follow_symlinks=True*)

Return a **`stat_result`** object for this entry. This method follows symbolic links by default; to stat a symbolic link add the `follow_symlinks=False` argument.

On Unix, this method always requires a system call. On Windows, it only requires a system call if *follow_symlinks* is `True` and the entry is a reparse point (for example, a symbolic link or directory junction).

> On Windows, the `st_ino`, `st_dev` and `st_nlink` attributes of the **`stat_result`** are always set to zero. Call **`os.stat()`** to get these attributes.
>
> The result is cached on the `os.DirEntry` object, with a separate cache for *follow_symlinks* `True` and `False`. Call **`os.stat()`** to fetch up-to-date information.

Note that there is a nice correspondence between several attributes and methods of `os.DirEntry` and of **`pathlib.Path`**. In particular, the `name` attribute has the same meaning, as do the `is_dir()`, `is_file()`, `is_symlink()`, `is_junction()`, and `stat()` methods.

Added in version 3.5.

Changed in version 3.6: Added support for the **`PathLike`** interface. Added support for **`bytes`** paths on Windows.

Changed in version 3.12: The `st_ctime` attribute of a stat result is deprecated on Windows. The file creation time is properly available as `st_birthtime`, and in the future `st_ctime` may be changed to return zero or the metadata change time, if available.

os.stat(*path*, *, *dir_fd=None*, *follow_symlinks=True*)

Get the status of a file or a file descriptor. Perform the equivalent of a **`stat()`** system call on the given path. *path* may be specified as either a string or bytes – directly or indirectly through the **`PathLike`** interface – or as an open file descriptor. Return a **`stat_result`** object.

This function normally follows symlinks; to stat a symlink add the argument `follow_symlinks=False`, or use **`lstat()`**.

This function can support specifying a file descriptor and not following symlinks.

On Windows, passing `follow_symlinks=False` will disable following all name-surrogate reparse points, which includes symlinks and directory junctions. Other types of reparse points that do not resemble links or that the operating system is unable to follow will be opened directly. When following a chain of multiple links, this may result in the original link being returned instead of the non-link that prevented full traversal. To obtain stat results for the final path in this case, use the **`os.path.realpath()`** function to resolve the path name as far as possible and call **`lstat()`** on the result. This does not apply to dangling symlinks or junction points, which will raise the usual exceptions.

Example:

```
>>> import os
>>> statinfo = os.stat('somefile.txt')
>>> statinfo
os.stat_result(st_mode=33188, st_ino=7876932,
st_dev=234881026,
st_nlink=1, st_uid=501, st_gid=501, st_size=264,
st_atime=1297230295,
st_mtime=1297230027, st_ctime=1297230027)
>>> statinfo.st_size
264
```

See also

`fstat()` and **`lstat()`** functions.

Changed in version 3.3: Added the *dir_fd* and *follow_symlinks* parameters, specifying a file descriptor instead of a path.

Changed in version 3.6: Accepts a path-like object.

Changed in version 3.8: On Windows, all reparse points that can be resolved by the operating system are now followed, and passing `follow_symlinks=False` disables following all name surrogate reparse points. If the operating system reaches a reparse point that it is not able to follow, *stat* now returns the information for the original path as if `follow_symlinks=False` had been specified instead of raising an error.

class os.stat_result

Object whose attributes correspond roughly to the members of the **`stat`** structure. It is used for the result of **`os.stat()`**, **`os.fstat()`** and **`os.lstat()`**.

Attributes:

st_mode

File mode: file type and file mode bits (permissions).

st_ino

Platform dependent, but if non-zero, uniquely identifies the file for a given value of `st_dev`. Typically:

- the inode number on Unix,
- the file index [https://msdn.microsoft.com/en-us/library/aa363788] on Windows

st_dev

Identifier of the device on which this file resides.

st_nlink

Number of hard links.

st_uid

User identifier of the file owner.

st_gid
Group identifier of the file owner.

st_size
Size of the file in bytes, if it is a regular file or a symbolic link. The size of a symbolic link is the length of the pathname it contains, without a terminating null byte.

Timestamps:

st_atime
Time of most recent access expressed in seconds.

st_mtime
Time of most recent content modification expressed in seconds.

st_ctime
Time of most recent metadata change expressed in seconds.

Changed in version 3.12: `st_ctime` is deprecated on Windows. Use `st_birthtime` for the file creation time. In the future, `st_ctime` will contain the time of the most recent metadata change, as for other platforms.

st_atime_ns
Time of most recent access expressed in nanoseconds as an integer.

Added in version 3.3.

st_mtime_ns
Time of most recent content modification expressed in nanoseconds as an integer.

Added in version 3.3.

st_ctime_ns

Time of most recent metadata change expressed in nanoseconds as an integer.

Added in version 3.3.

Changed in version 3.12: `st_ctime_ns` is deprecated on Windows. Use `st_birthtime_ns` for the file creation time. In the future, `st_ctime` will contain the time of the most recent metadata change, as for other platforms.

st_birthtime

Time of file creation expressed in seconds. This attribute is not always available, and may raise **`AttributeError`**.

Changed in version 3.12: `st_birthtime` is now available on Windows.

st_birthtime_ns

Time of file creation expressed in nanoseconds as an integer. This attribute is not always available, and may raise **`AttributeError`**.

Added in version 3.12.

Note

The exact meaning and resolution of the **`st_atime`**, **`st_mtime`**, **`st_ctime`** and **`st_birthtime`** attributes depend on the operating system and the file system. For example, on Windows systems using the FAT32 file systems, **`st_mtime`** has 2-second resolution, and **`st_atime`** has only 1-day resolution. See your operating system documentation for details.

Similarly, although `st_atime_ns`, `st_mtime_ns`, `st_ctime_ns` and `st_birthtime_ns` are always expressed in nanoseconds, many systems do not provide nanosecond precision. On systems that do provide nanosecond precision, the floating-point object used to store `st_atime`, `st_mtime`, `st_ctime` and `st_birthtime` cannot preserve all of it, and as such will be slightly inexact. If you need the exact timestamps you should always use `st_atime_ns`, `st_mtime_ns`, `st_ctime_ns` and `st_birthtime_ns`.

On some Unix systems (such as Linux), the following attributes may also be available:

st_blocks

Number of 512-byte blocks allocated for file. This may be smaller than `st_size`/512 when the file has holes.

st_blksize

“Preferred” blocksize for efficient file system I/O. Writing to a file in smaller chunks may cause an inefficient read-modify-rewrite.

st_rdev

Type of device if an inode device.

st_flags

User defined flags for file.

On other Unix systems (such as FreeBSD), the following attributes may be available (but may be only filled out if root tries to use them):

st_gen

File generation number.

On Solaris and derivatives, the following attributes may also be available:

st_fstype
: String that uniquely identifies the type of the filesystem that contains the file.

On macOS systems, the following attributes may also be available:

st_rsize
: Real size of the file.

st_creator
: Creator of the file.

st_type
: File type.

On Windows systems, the following attributes are also available:

st_file_attributes
: Windows file attributes: `dwFileAttributes` member of the `BY_HANDLE_FILE_INFORMATION` structure returned by **`GetFileInformationByHandle()`**. See the **`FILE_ATTRIBUTE_* <stat.FILE_ATTRIBUTE_ARCHIVE>`** constants in the `stat` module.

  Added in version 3.5.

st_reparse_tag
: When `st_file_attributes` has the `FILE_ATTRIBUTE_REPARSE_POINT` set, this field contains the tag identifying the type of reparse point. See the `IO_REPARSE_TAG_*` constants in the `stat` module.

The standard module `stat` defines functions and constants that are useful for extracting information from a **`stat`** structure. (On Windows, some items are filled with dummy values.)

For backward compatibility, a `stat_result` instance is also accessible as a tuple of at least 10 integers giving the most important (and portable) members of the **`stat`** structure, in the order `st_mode`, `st_ino`, `st_dev`, `st_nlink`, `st_uid`, `st_gid`, `st_size`, `st_atime`, `st_mtime`, `st_ctime`. More items may be added at the end by some implementations. For compatibility with older Python versions, accessing `stat_result` as a tuple always returns integers.

Changed in version 3.5: Windows now returns the file index as `st_ino` when available.

Changed in version 3.7: Added the `st_fstype` member to Solaris/derivatives.

Changed in version 3.8: Added the `st_reparse_tag` member on Windows.

Changed in version 3.8: On Windows, the `st_mode` member now identifies special files as **`S_IFCHR`**, **`S_IFIFO`** or **`S_IFBLK`** as appropriate.

Changed in version 3.12: On Windows, `st_ctime` is deprecated. Eventually, it will contain the last metadata change time, for consistency with other platforms, but for now still contains creation time. Use `st_birthtime` for the creation time.

On Windows, `st_ino` may now be up to 128 bits, depending on the file system. Previously it would not be above 64 bits, and larger file identifiers would be arbitrarily packed.

On Windows, `st_rdev` no longer returns a value. Previously it would contain the same as `st_dev`, which was incorrect.

Added the `st_birthtime` member on Windows.

os.statvfs(*path*)

Perform a **`statvfs()`** system call on the given path. The return value is an object whose attributes describe the filesystem on the given path, and correspond to the members of the **`statvfs`** structure, namely: **`f_bsize`**, **`f_frsize`**, **`f_blocks`**, **`f_bfree`**, **`f_bavail`**, **`f_files`**, **`f_ffree`**, **`f_favail`**, **`f_flag`**, **`f_namemax`**, **`f_fsid`**.

Two module-level constants are defined for the **`f_flag`** attribute’s bit-flags: if **`ST_RDONLY`** is set, the filesystem is mounted read-only, and if **`ST_NOSUID`** is set, the semantics of setuid/setgid bits are disabled or not supported.

Additional module-level constants are defined for GNU/glibc based systems. These are **`ST_NODEV`** (disallow access to device special files), **`ST_NOEXEC`** (disallow program execution), **`ST_SYNCHRONOUS`** (writes are synced at once), **`ST_MANDLOCK`** (allow mandatory locks on an FS), **`ST_WRITE`** (write on file/directory/symlink), **`ST_APPEND`** (append-only file), **`ST_IMMUTABLE`** (immutable file), **`ST_NOATIME`** (do not update access times), **`ST_NODIRATIME`** (do not update directory access times), **`ST_RELATIME`** (update atime relative to mtime/ctime).

This function can support specifying a file descriptor.

Availability: Unix.

Changed in version 3.2: The **`ST_RDONLY`** and **`ST_NOSUID`** constants were added.

Changed in version 3.3: Added support for specifying *path* as an open file descriptor.

Changed in version 3.4: The **`ST_NODEV`**, **`ST_NOEXEC`**, **`ST_SYNCHRONOUS`**, **`ST_MANDLOCK`**, **`ST_WRITE`**, **`ST_APPEND`**, **`ST_IMMUTABLE`**, **`ST_NOATIME`**, **`ST_NODIRATIME`**, and **`ST_RELATIME`** constants were added.

Changed in version 3.6: Accepts a path-like object.

Changed in version 3.7: Added the **`f_fsid`** attribute.

os.supports_dir_fd

A **`set`** object indicating which functions in the **`os`** module accept an open file descriptor for their *dir_fd* parameter. Different platforms provide different features, and the underlying functionality Python uses to implement the *dir_fd* parameter is not available on all platforms Python supports. For consistency’s sake, functions that may support *dir_fd* always allow specifying the parameter, but will throw an exception if the functionality is used when it’s not locally available. (Specifying `None` for *dir_fd* is always supported on all platforms.)

To check whether a particular function accepts an open file descriptor for its *dir_fd* parameter, use the `in` operator on `supports_dir_fd`. As an example, this expression evaluates to `True` if **`os.stat()`** accepts open file descriptors for *dir_fd* on the local platform:

```
os.stat in os.supports_dir_fd
```

Currently *dir_fd* parameters only work on Unix platforms; none of them work on Windows.

Added in version 3.3.

os.supports_effective_ids

A `set` object indicating whether `os.access()` permits specifying `True` for its *effective_ids* parameter on the local platform. (Specifying `False` for *effective_ids* is always supported on all platforms.) If the local platform supports it, the collection will contain `os.access()`; otherwise it will be empty.

This expression evaluates to `True` if `os.access()` supports `effective_ids=True` on the local platform:

```
os.access in os.supports_effective_ids
```

Currently *effective_ids* is only supported on Unix platforms; it does not work on Windows.

Added in version 3.3.

os.supports_fd

A `set` object indicating which functions in the `os` module permit specifying their *path* parameter as an open file descriptor on the local platform. Different platforms provide different features, and the underlying functionality Python uses to accept open file descriptors as *path* arguments is not available on all platforms Python supports.

To determine whether a particular function permits specifying an open file descriptor for its *path* parameter, use the `in` operator on `supports_fd`. As an example, this expression evaluates to `True` if `os.chdir()` accepts open file descriptors for *path* on your local platform:

```
os.chdir in os.supports_fd
```

Added in version 3.3.

os.supports_follow_symlinks

A `set` object indicating which functions in the `os` module accept `False` for their *follow_symlinks* parameter on the local platform. Different platforms provide different features, and the underlying functionality Python uses to implement *follow_symlinks* is not available on all platforms Python supports. For consistency's sake, functions that may support *follow_symlinks* always allow specifying the parameter, but will throw an exception if the functionality is used when it's not locally available. (Specifying `True` for *follow_symlinks* is always supported on all platforms.)

To check whether a particular function accepts `False` for its *follow_symlinks* parameter, use the `in` operator on `supports_follow_symlinks`. As an example, this expression evaluates to `True` if you may specify `follow_symlinks=False` when calling `os.stat()` on the local platform:

```
os.stat in os.supports_follow_symlinks
```

Added in version 3.3.

os.symlink(*src*, *dst*, *target_is_directory=False*, *, *dir_fd=None*)

Create a symbolic link pointing to *src* named *dst*.

The *src* parameter refers to the target of the link (the file or directory being linked to), and *dst* is the name of the link being created.

On Windows, a symlink represents either a file or a directory, and does not morph to the target dynamically. If the target is present, the type of the symlink will be created to match. Otherwise, the symlink will be created as a directory if *target_is_directory* is `True` or a file symlink (the default) otherwise. On non-Windows platforms, *target_is_directory* is ignored.

This function can support paths relative to directory descriptors.

Note

On newer versions of Windows 10, unprivileged accounts can create symlinks if Developer Mode is enabled. When Developer Mode is not available/enabled, the *SeCreateSymbolicLinkPrivilege* privilege is required, or the process must be run as an administrator.

`OSError` is raised when the function is called by an unprivileged user.

Raises an auditing event `os.symlink` with arguments `src`, `dst`, `dir_fd`.

Availability: Unix, Windows.

The function is limited on WASI, see WebAssembly platforms for more information.

Changed in version 3.2: Added support for Windows 6.0 (Vista) symbolic links.

Changed in version 3.3: Added the *dir_fd* parameter, and now allow *target_is_directory* on non-Windows platforms.

Changed in version 3.6: Accepts a path-like object for *src* and *dst*.

Changed in version 3.8: Added support for unelevated symlinks on Windows with Developer Mode.

os.sync()

Force write of everything to disk.

Availability: Unix.

Added in version 3.3.

os.truncate(*path*, *length*)

Truncate the file corresponding to *path*, so that it is at most *length* bytes in size.

This function can support specifying a file descriptor.

Raises an auditing event `os.truncate` with arguments `path`, `length`.

Availability: Unix, Windows.

Added in version 3.3.

Changed in version 3.5: Added support for Windows

Changed in version 3.6: Accepts a path-like object.

os.unlink(*path*, *, *dir_fd=None*)

Remove (delete) the file *path*. This function is semantically identical to **`remove()`**; the `unlink` name is its traditional Unix name. Please see the documentation for **`remove()`** for further information.

Raises an auditing event `os.remove` with arguments `path`, `dir_fd`.

Changed in version 3.3: Added the *dir_fd* parameter.

Changed in version 3.6: Accepts a path-like object.

os.utime(*path*, *times=None*, ***, [*ns*,]*dir_fd=None*, *follow_symlinks=True*)

Set the access and modified times of the file specified by *path*.

`utime()` takes two optional parameters, *times* and *ns*. These specify the times set on *path* and are used as follows:

- If *ns* is specified, it must be a 2-tuple of the form `(atime_ns, mtime_ns)` where each member is an int expressing nanoseconds.
- If *times* is not `None`, it must be a 2-tuple of the form `(atime, mtime)` where each member is an int or float expressing seconds.
- If *times* is `None` and *ns* is unspecified, this is equivalent to specifying `ns=(atime_ns, mtime_ns)` where both times are the current time.

It is an error to specify tuples for both *times* and *ns*.

Note that the exact times you set here may not be returned by a subsequent **`stat()`** call, depending on the resolution with which your operating system records access and modification times; see **`stat()`**. The best way to preserve exact times is to use the *st_atime_ns* and *st_mtime_ns* fields from the **`os.stat()`** result object with the *ns* parameter to **`utime()`**.

This function can support specifying a file descriptor, paths relative to directory descriptors and not following symlinks.

Raises an auditing event `os.utime` with arguments `path`, `times`, `ns`, `dir_fd`.

Changed in version 3.3: Added support for specifying *path* as an open file descriptor, and the *dir_fd*, *follow_symlinks*, and *ns* parameters.

Changed in version 3.6: Accepts a path-like object.

os.walk(*top*, *topdown=True*, *onerror=None*, *followlinks=False*)

Generate the file names in a directory tree by walking the tree either top-down or bottom-up. For each directory in the tree rooted at directory *top* (including *top* itself), it yields a 3-tuple `(dirpath, dirnames, filenames)`.

dirpath is a string, the path to the directory. *dirnames* is a list of the names of the subdirectories in *dirpath* (including symlinks to directories, and excluding `'.'` and `'..'`). *filenames* is a list of the names of the non-directory files in *dirpath*. Note that the names in the lists contain no path components. To get a full path (which begins with *top*) to a file or directory in *dirpath*, do `os.path.join(dirpath, name)`. Whether or not the lists are sorted depends on the file system. If a file is removed from or added to the *dirpath* directory during generating the lists, whether a name for that file be included is unspecified.

If optional argument *topdown* is `True` or not specified, the triple for a directory is generated before the triples for any of its subdirectories (directories are generated top-down). If *topdown* is `False`, the triple for a directory is generated after the triples for all of its subdirectories (directories are generated bottom-up). No matter the value of *topdown*, the list of subdirectories is retrieved before the tuples for the directory and its subdirectories are generated.

When *topdown* is `True`, the caller can modify the *dirnames* list in-place (perhaps using `del` or slice assignment), and `walk()` will only recurse into the subdirectories whose names remain in *dirnames*; this can be used to prune the search, impose a specific order of visiting, or even to inform `walk()` about directories the caller creates or renames before it resumes `walk()` again. Modifying *dirnames* when *topdown* is `False` has no effect on the behavior of the walk, because in bottom-up mode the directories in *dirnames* are generated before *dirpath* itself is generated.

By default, errors from the `scandir()` call are ignored. If optional argument *onerror* is specified, it should be a function; it will be called with one argument, an `OSError` instance. It can report the error to continue with the walk, or raise the exception to abort the walk. Note that the filename is available as the `filename` attribute of the exception object.

By default, `walk()` will not walk down into symbolic links that resolve to directories. Set *followlinks* to `True` to visit directories pointed to by symlinks, on systems that support them.

Note

Be aware that setting *followlinks* to `True` can lead to infinite recursion if a link points to a parent directory of itself. `walk()` does not keep track of the directories it visited already.

Note

If you pass a relative pathname, don’t change the current working directory between resumptions of `walk()`. `walk()` never changes the current directory, and assumes that its caller doesn’t either.

This example displays the number of bytes taken by non-directory files in each directory under the starting directory, except that it doesn’t look under any `__pycache__` subdirectory:

```
import os
from os.path import join, getsize
for root, dirs, files in os.walk('python/Lib/xml'):
    print(root, "consumes", end=" ")
    print(sum(getsize(join(root, name)) for name in files),
end=" ")
    print("bytes in", len(files), "non-directory files")
    if '__pycache__' in dirs:
        dirs.remove('__pycache__')  # don't visit
__pycache__ directories
```

In the next example (simple implementation of `shutil.rmtree()`), walking the tree bottom-up is essential, `rmdir()` doesn’t allow deleting a directory before the directory is empty:

```
# Delete everything reachable from the directory named in
"top",
# assuming there are no symbolic links.
# CAUTION:  This is dangerous!  For example, if top == '/',
it
# could delete all your disk files.
import os
for root, dirs, files in os.walk(top, topdown=False):
    for name in files:
        os.remove(os.path.join(root, name))
    for name in dirs:
        os.rmdir(os.path.join(root, name))
os.rmdir(top)
```

Raises an auditing event `os.walk` with arguments `top`, `topdown`, `onerror`, `followlinks`.

Changed in version 3.5: This function now calls **`os.scandir()`** instead of **`os.listdir()`**, making it faster by reducing the number of calls to **`os.stat()`**.

Changed in version 3.6: Accepts a path-like object.

os.fwalk(*top='.'*, *topdown=True*, *onerror=None*, ***, *follow_symlinks=False*, *dir_fd=None*)

This behaves exactly like **`walk()`**, except that it yields a 4-tuple `(dirpath, dirnames, filenames, dirfd)`, and it supports `dir_fd`.

dirpath, *dirnames* and *filenames* are identical to **`walk()`** output, and *dirfd* is a file descriptor referring to the directory *dirpath*.

This function always supports paths relative to directory descriptors and not following symlinks. Note however that, unlike other functions, the **`fwalk()`** default value for *follow_symlinks* is `False`.

Note

Since **`fwalk()`** yields file descriptors, those are only valid until the next iteration step, so you should duplicate them (e.g. with **`dup()`**) if you want to keep them longer.

This example displays the number of bytes taken by non-directory files in each directory under the starting directory, except that it doesn’t look under any `__pycache__` subdirectory:

```
import os
for root, dirs, files, rootfd in os.fwalk('python/Lib/xml'):
    print(root, "consumes", end=" ")
    print(sum([os.stat(name, dir_fd=rootfd).st_size for name
in files]),
          end=" ")
    print("bytes in", len(files), "non-directory files")
    if '__pycache__' in dirs:
        dirs.remove('__pycache__')  # don't visit
__pycache__ directories
```

In the next example, walking the tree bottom-up is essential: **`rmdir()`** doesn't allow deleting a directory before the directory is empty:

```
# Delete everything reachable from the directory named in
"top",
# assuming there are no symbolic links.
# CAUTION:  This is dangerous!  For example, if top == '/',
it
# could delete all your disk files.
import os
for root, dirs, files, rootfd in os.fwalk(top,
topdown=False):
    for name in files:
        os.unlink(name, dir_fd=rootfd)
    for name in dirs:
        os.rmdir(name, dir_fd=rootfd)
```

Raises an auditing event `os.fwalk` with arguments `top`, `topdown`, `onerror`, `follow_symlinks`, `dir_fd`.

Availability: Unix.

Added in version 3.3.

Changed in version 3.6: Accepts a path-like object.

Changed in version 3.7: Added support for **`bytes`** paths.

os.memfd_create(*name*[, *flags=os.MFD_CLOEXEC*])

Create an anonymous file and return a file descriptor that refers to it. *flags* must be one of the `os.MFD_*` constants available on the system (or a bitwise ORed combination of them). By default, the new file descriptor is non-inheritable.

The name supplied in *name* is used as a filename and will be displayed as the target of the corresponding symbolic link in the directory `/proc/self/fd/`. The displayed name is always prefixed with `memfd:` and serves only for debugging purposes. Names do not affect the behavior of the file descriptor, and as such multiple files can have the same name without any side effects.

Availability: Linux >= 3.17 with glibc >= 2.27.

Added in version 3.8.

os.MFD_CLOEXEC
os.MFD_ALLOW_SEALING
os.MFD_HUGETLB
os.MFD_HUGE_SHIFT
os.MFD_HUGE_MASK
os.MFD_HUGE_64KB
os.MFD_HUGE_512KB
os.MFD_HUGE_1MB
os.MFD_HUGE_2MB
os.MFD_HUGE_8MB
os.MFD_HUGE_16MB
os.MFD_HUGE_32MB
os.MFD_HUGE_256MB
os.MFD_HUGE_512MB
os.MFD_HUGE_1GB
os.MFD_HUGE_2GB
os.MFD_HUGE_16GB

These flags can be passed to **`memfd_create()`**.

Availability: Linux >= 3.17 with glibc >= 2.27

The `MFD_HUGE*` flags are only available since Linux 4.14.
Added in version 3.8.

os.eventfd(*initval*[, *flags=os.EFD_CLOEXEC*])

Create and return an event file descriptor. The file descriptors supports raw **`read()`** and **`write()`** with a buffer size of 8, **`select()`**, **`poll()`** and similar. See man page *eventfd(2)* *[https://manpages.debian.org/eventfd(2)]* for more information. By default, the new file descriptor is non-inheritable.

initval is the initial value of the event counter. The initial value must be a 32 bit unsigned integer. Please note that the initial value is limited to a 32 bit unsigned int although the event counter is an unsigned 64 bit integer with a maximum value of 2^{64}-2.

flags can be constructed from **`EFD_CLOEXEC`**, **`EFD_NONBLOCK`**, and **`EFD_SEMAPHORE`**.

If **`EFD_SEMAPHORE`** is specified and the event counter is non-zero, **`eventfd_read()`** returns 1 and decrements the counter by one.

If **`EFD_SEMAPHORE`** is not specified and the event counter is non-zero, **`eventfd_read()`** returns the current event counter value and resets the counter to zero.

If the event counter is zero and **`EFD_NONBLOCK`** is not specified, **`eventfd_read()`** blocks.

`eventfd_write()` increments the event counter. Write blocks if the write operation would increment the counter to a value larger than 2^{64}-2.

Example:

```
import os

# semaphore with start value '1'
fd = os.eventfd(1, os.EFD_SEMAPHORE | os.EFC_CLOEXEC)
try:
    # acquire semaphore
    v = os.eventfd_read(fd)
    try:
        do_work()
    finally:
        # release semaphore
        os.eventfd_write(fd, v)
finally:
    os.close(fd)
```

Availability: Linux >= 2.6.27 with glibc >= 2.8

Added in version 3.10.

os.eventfd_read(*fd*)

Read value from an **`eventfd()`** file descriptor and return a 64 bit unsigned int. The function does not verify that *fd* is an **`eventfd()`**.

Availability: Linux >= 2.6.27

Added in version 3.10.

os.eventfd_write(*fd*, *value*)

Add value to an **`eventfd()`** file descriptor. *value* must be a 64 bit unsigned int. The function does not verify that *fd* is an **`eventfd()`**.

Availability: Linux >= 2.6.27

Added in version 3.10.

os.EFD_CLOEXEC

Set close-on-exec flag for new **`eventfd()`** file descriptor.

Availability: Linux >= 2.6.27

Added in version 3.10.

os.EFD_NONBLOCK

Set **`O_NONBLOCK`** status flag for new **`eventfd()`** file descriptor.

Availability: Linux >= 2.6.27

Added in version 3.10.

os.EFD_SEMAPHORE

Provide semaphore-like semantics for reads from an **`eventfd()`** file descriptor. On read the internal counter is decremented by one.

Availability: Linux >= 2.6.30

Added in version 3.10.

Timer File Descriptors

Added in version 3.13.

These functions provide support for Linux’s *timer file descriptor* API. Naturally, they are all only available on Linux.

os.timerfd_create(*clockid*, /, *, *flags=0*)

Create and return a timer file descriptor (*timerfd*).

The file descriptor returned by **`timerfd_create()`** supports:

- **`read()`**
- **`select()`**
- **`poll()`**

The file descriptor’s `read()` method can be called with a buffer size of 8. If the timer has already expired one or more times, `read()` returns the number of expirations with the host’s endianness, which may be converted to an `int` by `int.from_bytes(x, byteorder=sys.byteorder)`.

`select()` and `poll()` can be used to wait until timer expires and the file descriptor is readable.

clockid must be a valid clock ID, as defined in the `time` module:

- `time.CLOCK_REALTIME`
- `time.CLOCK_MONOTONIC`
- `time.CLOCK_BOOTTIME` (Since Linux 3.15 for timerfd_create)

If *clockid* is `time.CLOCK_REALTIME`, a settable system-wide real-time clock is used. If system clock is changed, timer setting need to be updated. To cancel timer when system clock is changed, see `TFD_TIMER_CANCEL_ON_SET`.

If *clockid* is `time.CLOCK_MONOTONIC`, a non-settable monotonically increasing clock is used. Even if the system clock is changed, the timer setting will not be affected.

If *clockid* is `time.CLOCK_BOOTTIME`, same as `time.CLOCK_MONOTONIC` except it includes any time that the system is suspended.

The file descriptor’s behaviour can be modified by specifying a *flags* value. Any of the following variables may used, combined using bitwise OR (the `|` operator):

- `TFD_NONBLOCK`
- `TFD_CLOEXEC`

If `TFD_NONBLOCK` is not set as a flag, `read()` blocks until the timer expires. If it is set as a flag, `read()` doesn't block, but If there hasn't been an expiration since the last call to read, `read()` raises `OSError` with `errno` is set to `errno.EAGAIN`.

`TFD_CLOEXEC` is always set by Python automatically.

The file descriptor must be closed with `os.close()` when it is no longer needed, or else the file descriptor will be leaked.

See also

The *timerfd_create(2)* *[https://manpages.debian.org/timerfd_create(2)]* man page.

Availability: Linux >= 2.6.27 with glibc >= 2.8

Added in version 3.13.

os.timerfd_settime(*fd*, */*, ***, *flags=flags*, *initial=0.0*, *interval=0.0*)

Alter a timer file descriptor's internal timer. This function operates the same interval timer as `timerfd_settime_ns()`.

fd must be a valid timer file descriptor.

The timer's behaviour can be modified by specifying a *flags* value. Any of the following variables may used, combined using bitwise OR (the `|` operator):

- `TFD_TIMER_ABSTIME`
- `TFD_TIMER_CANCEL_ON_SET`

The timer is disabled by setting *initial* to zero (`0`). If *initial* is equal to or greater than zero, the timer is enabled. If *initial* is less than zero, it raises an `OSError` exception with `errno` set to `errno.EINVAL`

By default the timer will fire when *initial* seconds have elapsed. (If *initial* is zero, timer will fire immediately.)

However, if the `TFD_TIMER_ABSTIME` flag is set, the timer will fire when the timer's clock (set by *clockid* in `timerfd_create()`) reaches *initial* seconds.

The timer's interval is set by the *interval* `float`. If *interval* is zero, the timer only fires once, on the initial expiration. If *interval* is greater than zero, the timer fires every time *interval* seconds have elapsed since the previous expiration. If *interval* is less than zero, it raises `OSError` with `errno` set to `errno.EINVAL`

If the `TFD_TIMER_CANCEL_ON_SET` flag is set along with `TFD_TIMER_ABSTIME` and the clock for this timer is `time.CLOCK_REALTIME`, the timer is marked as cancelable if the real-time clock is changed discontinuously. Reading the descriptor is aborted with the error ECANCELED.

Linux manages system clock as UTC. A daylight-savings time transition is done by changing time offset only and doesn't cause discontinuous system clock change.

Discontinuous system clock change will be caused by the following events:

- `settimeofday`
- `clock_settime`
- set the system date and time by `date` command

Return a two-item tuple of (`next_expiration`, `interval`) from the previous timer state, before this function executed.

See also

timerfd_create(2) *[https://manpages.debian.org/timerfd_create(2)]*, *timerfd_settime(2)* *[https://manpages.debian.org/timerfd_settime(2)]*, *settimeofday(2)* *[https://manpages.debian.org/settimeofday(2)]*, *clock_settime(2)* *[https://manpages.debian.org/clock_settime(2)]*, and *date(1)* *[https://manpages.debian.org/date(1)]*.

Availability: Linux >= 2.6.27 with glibc >= 2.8

Added in version 3.13.

os.timerfd_settime_ns(*fd*, */*, ***, *flags=0*, *initial=0*, *interval=0*)

Similar to `timerfd_settime()`, but use time as nanoseconds. This function operates the same interval timer as `timerfd_settime()`.

Availability: Linux >= 2.6.27 with glibc >= 2.8

Added in version 3.13.

os.timerfd_gettime(*fd*, */*)

Return a two-item tuple of floats (`next_expiration`, `interval`).

`next_expiration` denotes the relative time until next the timer next fires, regardless of if the `TFD_TIMER_ABSTIME` flag is set.

`interval` denotes the timer’s interval. If zero, the timer will only fire once, after `next_expiration` seconds have elapsed.

See also

timerfd_gettime(2)
[https://manpages.debian.org/timerfd_gettime(2)]

Availability: Linux >= 2.6.27 with glibc >= 2.8

Added in version 3.13.

os.timerfd_gettime_ns(*fd*, */*)

Similar to `timerfd_gettime()`, but return time as nanoseconds.

Availability: Linux >= 2.6.27 with glibc >= 2.8

Added in version 3.13.

os.TFD_NONBLOCK

A flag for the `timerfd_create()` function, which sets the `O_NONBLOCK` status flag for the new timer file descriptor. If `TFD_NONBLOCK` is not set as a flag, `read()` blocks.

Availability: Linux >= 2.6.27 with glibc >= 2.8

Added in version 3.13.

os.TFD_CLOEXEC

A flag for the `timerfd_create()` function, If `TFD_CLOEXEC` is set as a flag, set close-on-exec flag for new file descriptor.

Availability: Linux >= 2.6.27 with glibc >= 2.8

Added in version 3.13.

os.TFD_TIMER_ABSTIME

A flag for the `timerfd_settime()` and `timerfd_settime_ns()` functions. If this flag is set, *initial* is interpreted as an absolute value on the timer's clock (in UTC seconds or nanoseconds since the Unix Epoch).

Availability: Linux >= 2.6.27 with glibc >= 2.8

Added in version 3.13.

os.TFD_TIMER_CANCEL_ON_SET

A flag for the **`timerfd_settime()`** and **`timerfd_settime_ns()`** functions along with **`TFD_TIMER_ABSTIME`**. The timer is cancelled when the time of the underlying clock changes discontinuously.

Availability: Linux >= 2.6.27 with glibc >= 2.8

Added in version 3.13.

Linux extended attributes

Added in version 3.3.

These functions are all available on Linux only.

os.getxattr(*path*, *attribute*, *, *follow_symlinks=True*)

Return the value of the extended filesystem attribute *attribute* for *path*. *attribute* can be bytes or str (directly or indirectly through the **`PathLike`** interface). If it is str, it is encoded with the filesystem encoding.

This function can support specifying a file descriptor and not following symlinks.

Raises an auditing event `os.getxattr` with arguments `path`, `attribute`.

Changed in version 3.6: Accepts a path-like object for *path* and *attribute*.

os.listxattr(*path=None*, *, *follow_symlinks=True*)

Return a list of the extended filesystem attributes on *path*. The attributes in the list are represented as strings decoded with the filesystem encoding. If *path* is `None`, **`listxattr()`** will examine the current directory.

This function can support specifying a file descriptor and not following symlinks.

Raises an auditing event `os.listxattr` with argument `path`.

Changed in version 3.6: Accepts a path-like object.

os.removexattr(*path*, *attribute*, *, *follow_symlinks=True*)

Removes the extended filesystem attribute *attribute* from *path*. *attribute* should be bytes or str (directly or indirectly through the **`PathLike`** interface). If it is a string, it is encoded with the filesystem encoding and error handler.

This function can support specifying a file descriptor and not following symlinks.

Raises an auditing event `os.removexattr` with arguments `path`, `attribute`.

Changed in version 3.6: Accepts a path-like object for *path* and *attribute*.

os.setxattr(*path*, *attribute*, *value*, *flags=0*, *, *follow_symlinks=True*)

Set the extended filesystem attribute *attribute* on *path* to *value*. *attribute* must be a bytes or str with no embedded NULs (directly or indirectly through the **`PathLike`** interface). If it is a str, it is encoded with the filesystem encoding and error handler. *flags* may be **`XATTR_REPLACE`** or

`XATTR_CREATE`. If `XATTR_REPLACE` is given and the attribute does not exist, `ENODATA` will be raised. If `XATTR_CREATE` is given and the attribute already exists, the attribute will not be created and `EEXISTS` will be raised.

This function can support specifying a file descriptor and not following symlinks.

Note

A bug in Linux kernel versions less than 2.6.39 caused the flags argument to be ignored on some filesystems.

Raises an auditing event `os.setxattr` with arguments `path`, `attribute`, `value`, `flags`.

Changed in version 3.6: Accepts a path-like object for *path* and *attribute*.

os.XATTR_SIZE_MAX

The maximum size the value of an extended attribute can be. Currently, this is 64 KiB on Linux.

os.XATTR_CREATE

This is a possible value for the flags argument in `setxattr()`. It indicates the operation must create an attribute.

os.XATTR_REPLACE

This is a possible value for the flags argument in `setxattr()`. It indicates the operation must replace an existing attribute.

Process Management

These functions may be used to create and manage processes.

The various **`exec*`** functions take a list of arguments for the new program loaded into the process. In each case, the first of these arguments is passed to the new program as its own name rather than as an argument a user may have typed on a command line. For the C programmer, this is the `argv[0]` passed to a program's **`main()`**. For example, `os.execv('/bin/echo', ['foo', 'bar'])` will only print `bar` on standard output; `foo` will seem to be ignored.

os.abort()

Generate a **`SIGABRT`** signal to the current process. On Unix, the default behavior is to produce a core dump; on Windows, the process immediately returns an exit code of `3`. Be aware that calling this function will not call the Python signal handler registered for **`SIGABRT`** with **`signal.signal()`**.

os.add_dll_directory(*path*)

Add a path to the DLL search path.

This search path is used when resolving dependencies for imported extension modules (the module itself is resolved through **`sys.path`**), and also by **`ctypes`**.

Remove the directory by calling **close()** on the returned object or using it in a **`with`** statement.

See the Microsoft documentation [https://msdn.microsoft.com/44228cf2-6306-466c-8f16-f513cd3ba8b5] for more information about how DLLs are loaded.

Raises an auditing event `os.add_dll_directory` with argument `path`.

Availability: Windows.

Added in version 3.8: Previous versions of CPython would resolve DLLs using the default behavior for the current process. This led to inconsistencies, such as only sometimes searching **`PATH`** or the current working directory, and OS functions such as `AddDllDirectory` having no effect.

In 3.8, the two primary ways DLLs are loaded now explicitly override the process-wide behavior to ensure consistency. See the porting notes for information on updating libraries.

os.execl(*path*, *arg0*, *arg1*, *...*)
os.execle(*path*, *arg0*, *arg1*, *...*, *env*)
os.execlp(*file*, *arg0*, *arg1*, *...*)
os.execlpe(*file*, *arg0*, *arg1*, *...*, *env*)
os.execv(*path*, *args*)
os.execve(*path*, *args*, *env*)
os.execvp(*file*, *args*)
os.execvpe(*file*, *args*, *env*)

These functions all execute a new program, replacing the current process; they do not return. On Unix, the new executable is loaded into the current process, and will have the same process id as the caller. Errors will be reported as `OSError` exceptions.

The current process is replaced immediately. Open file objects and descriptors are not flushed, so if there may be data buffered on these open files, you should flush them using **`sys.stdout.flush()`** or `os.fsync()` before calling an `exec*` function.

The “l” and “v” variants of the `exec*` functions differ in how command-line arguments are passed. The “l” variants are perhaps the easiest to work with if the number of parameters is fixed when the code is written; the individual parameters simply become additional parameters to the **`execl*()`** functions. The “v” variants are good when the number of parameters is variable, with the arguments being passed in a list or tuple as the *args* parameter. In either case, the arguments to the child process should start with the name of the command being run, but this is not enforced.

The variants which include a “p” near the end (`execlp()`, `execlpe()`, `execvp()`, and `execvpe()`) will use the **`PATH`** environment variable to locate the program *file*. When the environment is being replaced (using one of the `exec*e` variants, discussed in the next paragraph), the new environment is used as the source of the **`PATH`** variable. The other variants, `execl()`, `execle()`, `execv()`, and `execve()`, will not use the **`PATH`** variable to locate the executable; *path* must contain an appropriate absolute or relative path. Relative paths must include at least one slash, even on Windows, as plain names will not be resolved.

For `execle()`, `execlpe()`, `execve()`, and `execvpe()` (note that these all end in “e”), the *env* parameter must be a mapping which is used to define the environment variables for the new process (these are used instead of the current process’ environment); the functions `execl()`, `execlp()`, `execv()`, and `execvp()` all cause the new process to inherit the environment of the current process.

For `execve()` on some platforms, *path* may also be specified as an open file descriptor. This functionality may not be supported on your platform; you can check whether or not it is available using `os.supports_fd`. If it is unavailable, using it will raise a `NotImplementedError`.

Raises an auditing event `os.exec` with arguments `path`, `args`, `env`.

Availability: Unix, Windows, not WASI, not Android, not iOS.

Changed in version 3.3: Added support for specifying *path* as an open file descriptor for `execve()`.

Changed in version 3.6: Accepts a path-like object.

os._exit(*n*)

Exit the process with status *n*, without calling cleanup handlers, flushing stdio buffers, etc.

Note

The standard way to exit is `sys.exit(n)`. `_exit()` should normally only be used in the child process after a `fork()`.

The following exit codes are defined and can be used with `_exit()`, although they are not required. These are typically used for system programs written in Python, such as a mail server’s external command delivery program.

Note

Some of these may not be available on all Unix platforms, since there is some variation. These constants are defined where they are defined by the underlying platform.

os.EX_OK

Exit code that means no error occurred. May be taken from the defined value of `EXIT_SUCCESS` on some platforms. Generally has a value of zero.

Availability: Unix, Windows.

os.EX_USAGE

Exit code that means the command was used incorrectly, such as when the wrong number of arguments are given.

Availability: Unix, not WASI.

os.EX_DATAERR

Exit code that means the input data was incorrect.

Availability: Unix, not WASI.

os.EX_NOINPUT

Exit code that means an input file did not exist or was not readable.

Availability: Unix, not WASI.

os.EX_NOUSER

Exit code that means a specified user did not exist.

Availability: Unix, not WASI.

os.EX_NOHOST

Exit code that means a specified host did not exist.

Availability: Unix, not WASI.

os.EX_UNAVAILABLE

Exit code that means that a required service is unavailable.

Availability: Unix, not WASI.

os.EX_SOFTWARE

Exit code that means an internal software error was detected.

Availability: Unix, not WASI.

os.EX_OSERR

Exit code that means an operating system error was detected, such as the inability to fork or create a pipe.

Availability: Unix, not WASI.

os.EX_OSFILE

Exit code that means some system file did not exist, could not be opened, or had some other kind of error.

Availability: Unix, not WASI.

os.EX_CANTCREAT

Exit code that means a user specified output file could not be created.

Availability: Unix, not WASI.

os.EX_IOERR

Exit code that means that an error occurred while doing I/O on some file.

Availability: Unix, not WASI.

os.EX_TEMPFAIL

Exit code that means a temporary failure occurred. This indicates something that may not really be an error, such as a network connection that couldn’t be made during a retryable operation.

Availability: Unix, not WASI.

os.EX_PROTOCOL

Exit code that means that a protocol exchange was illegal, invalid, or not understood.

Availability: Unix, not WASI.

os.EX_NOPERM

Exit code that means that there were insufficient permissions to perform the operation (but not intended for file system problems).

Availability: Unix, not WASI.

os.EX_CONFIG

Exit code that means that some kind of configuration error occurred.

Availability: Unix, not WASI.

os.EX_NOTFOUND

Exit code that means something like “an entry was not found”.

Availability: Unix, not WASI.

os.fork()

Fork a child process. Return `0` in the child and the child’s process id in the parent. If an error occurs **`OSError`** is raised.

Note that some platforms including FreeBSD <= 6.3 and Cygwin have known issues when using `fork()` from a thread.

Raises an auditing event `os.fork` with no arguments.

Warning

If you use TLS sockets in an application calling `fork()`, see the warning in the `ssl` documentation.

Warning

On macOS the use of this function is unsafe when mixed with using higher-level system APIs, and that includes using `urllib.request`.

Changed in version 3.8: Calling `fork()` in a subinterpreter is no longer supported (`RuntimeError` is raised).

Changed in version 3.12: If Python is able to detect that your process has multiple threads, `os.fork()` now raises a `DeprecationWarning`.

We chose to surface this as a warning, when detectable, to better inform developers of a design problem that the POSIX platform specifically notes as not supported. Even in code that *appears* to work, it has never been safe to mix threading with `os.fork()` on POSIX platforms. The CPython runtime itself has always made API calls that are not safe for use in the child process when threads existed in the parent (such as `malloc` and `free`).

Users of macOS or users of libc or malloc implementations other than those typically found in glibc to date are among those already more likely to experience deadlocks running such code.

See this discussion on fork being incompatible with threads [https://discuss.python.org/t/33555] for technical details of why we're surfacing this longstanding platform compatibility problem to developers.

Availability: POSIX, not WASI, not Android, not iOS.

os.forkpty()

Fork a child process, using a new pseudo-terminal as the child’s controlling terminal. Return a pair of `(pid, fd)`, where *pid* is `0` in the child, the new child’s process id in the parent, and *fd* is the file descriptor of the master end of the pseudo-terminal. For a more portable approach, use the **`pty`** module. If an error occurs **`OSError`** is raised.

Raises an auditing event `os.forkpty` with no arguments.

Warning

On macOS the use of this function is unsafe when mixed with using higher-level system APIs, and that includes using **`urllib.request`**.

Changed in version 3.8: Calling `forkpty()` in a subinterpreter is no longer supported (**`RuntimeError`** is raised).

Changed in version 3.12: If Python is able to detect that your process has multiple threads, this now raises a **`DeprecationWarning`**. See the longer explanation on **`os.fork()`**.

Availability: Unix, not WASI, not Android, not iOS.

os.kill(*pid*, *sig*, */*)

Send signal *sig* to the process *pid*. Constants for the specific signals available on the host platform are defined in the **`signal`** module.

Windows: The **`signal.CTRL_C_EVENT`** and **`signal.CTRL_BREAK_EVENT`** signals are special signals which can only be sent to console processes which share a common console window, e.g., some subprocesses. Any other value for *sig* will cause the process to be unconditionally killed by the TerminateProcess API, and the exit code will be set to *sig*.

See also **`signal.pthread_kill()`**.

Raises an auditing event `os.kill` with arguments `pid`, `sig`.

Availability: Unix, Windows, not WASI, not iOS.

Changed in version 3.2: Added Windows support.

os.killpg(*pgid*, *sig*, */*)

Send the signal *sig* to the process group *pgid*.

Raises an auditing event `os.killpg` with arguments `pgid`, `sig`.

Availability: Unix, not WASI, not iOS.

os.nice(*increment*, */*)

Add *increment* to the process's "niceness". Return the new niceness.

Availability: Unix, not WASI.

os.pidfd_open(*pid*, *flags=0*)

Return a file descriptor referring to the process *pid* with *flags* set. This descriptor can be used to perform process management without races and signals.

See the *pidfd_open(2)* *[https://manpages.debian.org/pidfd_open(2)]* man page for more details.

Availability: Linux >= 5.3, Android >= **`build-time`** API level 31

Added in version 3.9.

os.PIDFD_NONBLOCK

This flag indicates that the file descriptor will be non-blocking. If the process referred to by the file descriptor has not yet terminated, then an attempt to wait on the file descriptor using *waitid(2)* *[https://manpages.debian.org/waitid(2)]* will immediately return the error **`EAGAIN`** rather than blocking.

Availability: Linux >= 5.10

Added in version 3.12.

os.plock(*op*, */*)

Lock program segments into memory. The value of *op* (defined in `<sys/lock.h>`) determines which segments are locked.

Availability: Unix, not WASI, not iOS.

os.popen(*cmd*, *mode='r'*, *buffering=-1*)

Open a pipe to or from command *cmd*. The return value is an open file object connected to the pipe, which can be read or written depending on whether *mode* is `'r'` (default) or `'w'`. The *buffering* argument have the same meaning as the corresponding argument to the built-in **`open()`** function. The returned file object reads or writes text strings rather than bytes.

The `close` method returns **`None`** if the subprocess exited successfully, or the subprocess’s return code if there was an error. On POSIX systems, if the return code is positive it represents the return value of the process left-shifted by one byte. If the return code is negative, the process was terminated by the signal given by the negated value of the return code. (For example, the return value might be `- signal.SIGKILL` if the subprocess was killed.) On Windows systems, the return value contains the signed integer return code from the child process.

On Unix, **`waitstatus_to_exitcode()`** can be used to convert the `close` method result (exit status) into an exit code if it is not `None`. On Windows, the `close` method result is directly the exit code (or `None`).

This is implemented using **`subprocess.Popen`**; see that class's documentation for more powerful ways to manage and communicate with subprocesses.

Availability: not WASI, not Android, not iOS.

Note

The Python UTF-8 Mode affects encodings used for *cmd* and pipe contents.

`popen()` is a simple wrapper around **`subprocess.Popen`**. Use **`subprocess.Popen`** or **`subprocess.run()`** to control options like encodings.

Deprecated since version 3.14: The function is soft deprecated and should no longer be used to write new code. The **`subprocess`** module is recommended instead.

os.posix_spawn(*path*, *argv*, *env*, ***, *file_actions=None*, *setpgroup=None*, *resetids=False*, *setsid=False*, *setsigmask=()*, *setsigdef=()*, *scheduler=None*)

Wraps the **`posix_spawn()`** C library API for use from Python.

Most users should use `subprocess.run()` instead of `posix_spawn()`.

The positional-only arguments *path*, *args*, and *env* are similar to `execve()`. *env* is allowed to be `None`, in which case current process' environment is used.

The *path* parameter is the path to the executable file. The *path* should contain a directory. Use `posix_spawnp()` to pass an executable file without directory.

The *file_actions* argument may be a sequence of tuples describing actions to take on specific file descriptors in the child process between the C library implementation's **`fork()`** and **`exec()`** steps. The first item in each tuple must be one of the three type indicator listed below describing the remaining tuple elements:

os.POSIX_SPAWN_OPEN

(`os.POSIX_SPAWN_OPEN`, *fd*, *path*, *flags*, *mode*)

Performs `os.dup2(os.open(path, flags, mode), fd)`.

os.POSIX_SPAWN_CLOSE

(`os.POSIX_SPAWN_CLOSE`, *fd*)

Performs `os.close(fd)`.

os.POSIX_SPAWN_DUP2

(`os.POSIX_SPAWN_DUP2`, *fd*, *new_fd*)

Performs `os.dup2(fd, new_fd)`.

os.POSIX_SPAWN_CLOSEFROM

(`os.POSIX_SPAWN_CLOSEFROM`, *fd*)

Performs `os.closerange(fd, INF)`.

These tuples correspond to the C library **`posix_spawn_file_actions_addopen()`**, **`posix_spawn_file_actions_addclose()`**, **`posix_spawn_file_actions_adddup2()`**, and **`posix_spawn_file_actions_addclosefrom_np()`** API calls used to prepare for the **`posix_spawn()`** call itself.

The *setpgroup* argument will set the process group of the child to the value specified. If the value specified is 0, the child’s process group ID will be made the same as its process ID. If the value of *setpgroup* is not set, the child will inherit the parent’s process group ID. This argument corresponds to the C library **`POSIX_SPAWN_SETPGROUP`** flag.

If the *resetids* argument is `True` it will reset the effective UID and GID of the child to the real UID and GID of the parent process. If the argument is `False`, then the child retains the effective UID and GID of the parent. In either case, if the set-user-ID and set-group-ID permission bits are enabled on the executable file, their effect will override the setting of the effective UID and GID. This argument corresponds to the C library **`POSIX_SPAWN_RESETIDS`** flag.

If the *setsid* argument is `True`, it will create a new session ID for `posix_spawn`. *setsid* requires **`POSIX_SPAWN_SETSID`** or **`POSIX_SPAWN_SETSID_NP`** flag. Otherwise, `NotImplementedError` is raised.

The *setsigmask* argument will set the signal mask to the signal set specified. If the parameter is not used, then the child inherits the parent’s signal mask. This argument corresponds to the C library **`POSIX_SPAWN_SETSIGMASK`** flag.

The *sigdef* argument will reset the disposition of all signals in the set specified. This argument corresponds to the C library **`POSIX_SPAWN_SETSIGDEF`** flag.

The *scheduler* argument must be a tuple containing the (optional) scheduler policy and an instance of [**`sched_param`**] with the scheduler parameters. A value of `None` in the place of the scheduler policy indicates that is not being provided. This argument is a combination of the C library **`POSIX_SPAWN_SETSCHEDPARAM`** and **`POSIX_SPAWN_SETSCHEDULER`** flags.

Raises an auditing event `os.posix_spawn` with arguments `path`, `argv`, `env`.

Added in version 3.8.

Changed in version 3.13: env parameter accepts `None`. `os.POSIX_SPAWN_CLOSEFROM` is available on platforms where **`posix_spawn_file_actions_addclosefrom_np()`** exists.

Availability: Unix, not WASI, not Android, not iOS.

os.posix_spawnp(*path*, *argv*, *env*, ***, *file_actions=None*, *setpgroup=None*, *resetids=False*, *setsid=False*, *setsigmask=()*, *setsigdef=()*, *scheduler=None*)

Wraps the **`posix_spawnp()`** C library API for use from Python.

Similar to **`posix_spawn()`** except that the system searches for the *executable* file in the list of directories specified by the **`PATH`** environment variable (in the same way as for `execvp(3)`).

Raises an auditing event `os.posix_spawn` with arguments `path`, `argv`, `env`.

Added in version 3.8.

Availability: POSIX, not WASI, not Android, not iOS.

See `posix_spawn()` documentation.

os.register_at_fork(*, *before=None*, *after_in_parent=None*, *after_in_child=None*)

Register callables to be executed when a new child process is forked using `os.fork()` or similar process cloning APIs. The parameters are optional and keyword-only. Each specifies a different call point.

- *before* is a function called before forking a child process.
- *after_in_parent* is a function called from the parent process after forking a child process.
- *after_in_child* is a function called from the child process.

These calls are only made if control is expected to return to the Python interpreter. A typical `subprocess` launch will not trigger them as the child is not going to re-enter the interpreter.

Functions registered for execution before forking are called in reverse registration order. Functions registered for execution after forking (either in the parent or in the child) are called in registration order.

Note that `fork()` calls made by third-party C code may not call those functions, unless it explicitly calls `PyOS_BeforeFork()`, `PyOS_AfterFork_Parent()` and `PyOS_AfterFork_Child()`.

There is no way to unregister a function.

Availability: Unix, not WASI, not Android, not iOS.

Added in version 3.7.

os.spawnl(*mode*, *path*, *...*)
os.spawnle(*mode*, *path*, *...*, *env*)
os.spawnlp(*mode*, *file*, *...*)
os.spawnlpe(*mode*, *file*, *...*, *env*)
os.spawnv(*mode*, *path*, *args*)
os.spawnve(*mode*, *path*, *args*, *env*)
os.spawnvp(*mode*, *file*, *args*)
os.spawnvpe(*mode*, *file*, *args*, *env*)

Execute the program *path* in a new process.

(Note that the **`subprocess`** module provides more powerful facilities for spawning new processes and retrieving their results; using that module is preferable to using these functions. Check especially the Replacing Older Functions with the subprocess Module section.)

If *mode* is **`P_NOWAIT`**, this function returns the process id of the new process; if *mode* is **`P_WAIT`**, returns the process's exit code if it exits normally, or `-signal`, where *signal* is the signal that killed the process. On Windows, the process id will actually be the process handle, so can be used with the **`waitpid()`** function.

Note on VxWorks, this function doesn't return `-signal` when the new process is killed. Instead it raises OSError exception.

The "l" and "v" variants of the **`spawn*`** functions differ in how command-line arguments are passed. The "l" variants are perhaps the easiest to work with if the number of parameters is fixed when the code is written; the individual parameters simply become additional parameters to the **`spawnl*()`** functions. The "v" variants are good when the number of parameters is variable, with the arguments being passed in a list or tuple as the *args* parameter. In either case, the arguments to the child process must start with the name of the command being run.

The variants which include a second “p” near the end (`spawnlp()`, `spawnlpe()`, `spawnvp()`, and `spawnvpe()`) will use the **`PATH`** environment variable to locate the program *file*. When the environment is being replaced (using one of the `spawn*e` variants, discussed in the next paragraph), the new environment is used as the source of the **`PATH`** variable. The other variants, `spawnl()`, `spawnle()`, `spawnv()`, and `spawnve()`, will not use the **`PATH`** variable to locate the executable; *path* must contain an appropriate absolute or relative path.

For `spawnle()`, `spawnlpe()`, `spawnve()`, and `spawnvpe()` (note that these all end in “e”), the *env* parameter must be a mapping which is used to define the environment variables for the new process (they are used instead of the current process’ environment); the functions `spawnl()`, `spawnlp()`, `spawnv()`, and `spawnvp()` all cause the new process to inherit the environment of the current process. Note that keys and values in the *env* dictionary must be strings; invalid keys or values will cause the function to fail, with a return value of `127`.

As an example, the following calls to `spawnlp()` and `spawnvpe()` are equivalent:

```
import os
os.spawnlp(os.P_WAIT, 'cp', 'cp', 'index.html', '/dev/null')

L = ['cp', 'index.html', '/dev/null']
os.spawnvpe(os.P_WAIT, 'cp', L, os.environ)
```

Raises an auditing event `os.spawn` with arguments `mode`, `path`, `args`, `env`.

Availability: Unix, Windows, not WASI, not Android, not iOS.

`spawnlp()`, **`spawnlpe()`**, **`spawnvp()`** and **`spawnvpe()`** are not available on Windows. **`spawnle()`** and **`spawnve()`** are not thread-safe on Windows; we advise you to use the **`subprocess`** module instead.

Changed in version 3.6: Accepts a path-like object.

Deprecated since version 3.14: These functions are soft deprecated and should no longer be used to write new code. The **`subprocess`** module is recommended instead.

os.P_NOWAIT
os.P_NOWAITO

Possible values for the *mode* parameter to the **`spawn*`** family of functions. If either of these values is given, the **`spawn*`** functions will return as soon as the new process has been created, with the process id as the return value.

Availability: Unix, Windows.

os.P_WAIT

Possible value for the *mode* parameter to the **`spawn*`** family of functions. If this is given as *mode*, the **`spawn*`** functions will not return until the new process has run to completion and will return the exit code of the process the run is successful, or `-signal` if a signal kills the process.

Availability: Unix, Windows.

os.P_DETACH

os.P_OVERLAY

Possible values for the *mode* parameter to the `spawn*` family of functions. These are less portable than those listed above. `P_DETACH` is similar to `P_NOWAIT`, but the new process is detached from the console of the calling process. If `P_OVERLAY` is used, the current process will be replaced; the `spawn*` function will not return.

Availability: Windows.

os.startfile(*path*[, *operation*][, *arguments*][, *cwd*][, *show_cmd*])

Start a file with its associated application.

When *operation* is not specified, this acts like double-clicking the file in Windows Explorer, or giving the file name as an argument to the **start** command from the interactive command shell: the file is opened with whatever application (if any) its extension is associated.

When another *operation* is given, it must be a “command verb” that specifies what should be done with the file. Common verbs documented by Microsoft are `'open'`, `'print'` and `'edit'` (to be used on files) as well as `'explore'` and `'find'` (to be used on directories).

When launching an application, specify *arguments* to be passed as a single string. This argument may have no effect when using this function to launch a document.

The default working directory is inherited, but may be overridden by the *cwd* argument. This should be an absolute path. A relative *path* will be resolved against this argument.

Use *show_cmd* to override the default window style. Whether this has any effect will depend on the application being launched. Values are integers as supported by the Win32 **`ShellExecute()`** function.

`startfile()` returns as soon as the associated application is launched. There is no option to wait for the application to close, and no way to retrieve the application’s exit status. The *path* parameter is relative to the current directory or *cwd*. If you want to use an absolute path, make sure the first character is not a slash (`'/'`) Use **`pathlib`** or the **`os.path.normpath()`** function to ensure that paths are properly encoded for Win32.

To reduce interpreter startup overhead, the Win32 **`ShellExecute()`** function is not resolved until this function is first called. If the function cannot be resolved, **`NotImplementedError`** will be raised.

Raises an auditing event `os.startfile` with arguments `path`, `operation`.

Raises an auditing event `os.startfile/2` with arguments `path`, `operation`, `arguments`, `cwd`, `show_cmd`.

Availability: Windows.

Changed in version 3.10: Added the *arguments*, *cwd* and *show_cmd* arguments, and the `os.startfile/2` audit event.

os.system(*command*)

Execute the command (a string) in a subshell. This is implemented by calling the Standard C function **`system()`**, and has the same limitations. Changes to **`sys.stdin`**, etc. are not reflected in the environment of the executed command. If *command* generates any output, it will be sent to the interpreter standard output stream. The C standard does not specify the meaning of the return value of the C function, so the return value of the Python function is system-dependent.

On Unix, the return value is the exit status of the process encoded in the format specified for `wait()`.

On Windows, the return value is that returned by the system shell after running *command*. The shell is given by the Windows environment variable **`COMSPEC`**: it is usually **cmd.exe**, which returns the exit status of the command run; on systems using a non-native shell, consult your shell documentation.

The `subprocess` module provides more powerful facilities for spawning new processes and retrieving their results; using that module is recommended to using this function. See the Replacing Older Functions with the subprocess Module section in the `subprocess` documentation for some helpful recipes.

On Unix, `waitstatus_to_exitcode()` can be used to convert the result (exit status) into an exit code. On Windows, the result is directly the exit code.

Raises an auditing event `os.system` with argument `command`.

Availability: Unix, Windows, not WASI, not Android, not iOS.

os.times()

Returns the current global process times. The return value is an object with five attributes:

- **`user`** - user time
- **`system`** - system time

- **`children_user`** - user time of all child processes
- **`children_system`** - system time of all child processes
- **`elapsed`** - elapsed real time since a fixed point in the past

For backwards compatibility, this object also behaves like a five-tuple containing **`user`**, **`system`**, **`children_user`**, **`children_system`**, and **`elapsed`** in that order.

See the Unix manual page *times(2)* *[https://manpages.debian.org/times(2)]* and times(3) [https://man.freebsd.org/cgi/man.cgi?time(3)] manual page on Unix or the GetProcessTimes MSDN [https://docs.microsoft.com/windows/win32/api/processthreadsapi/nf-processthreadsapi-getprocesstimes] on Windows. On Windows, only **`user`** and **`system`** are known; the other attributes are zero.

Availability: Unix, Windows.

Changed in version 3.3: Return type changed from a tuple to a tuple-like object with named attributes.

os.wait()

Wait for completion of a child process, and return a tuple containing its pid and exit status indication: a 16-bit number, whose low byte is the signal number that killed the process, and whose high byte is the exit status (if the signal number is zero); the high bit of the low byte is set if a core file was produced.

If there are no children that could be waited for, **`ChildProcessError`** is raised.

`waitstatus_to_exitcode()` can be used to convert the exit status into an exit code.

Availability: Unix, not WASI, not Android, not iOS.

See also

The other `wait*()` functions documented below can be used to wait for the completion of a specific child process and have more options. `waitpid()` is the only one also available on Windows.

os.waitid(*idtype*, *id*, *options*, /)

Wait for the completion of a child process.

idtype can be `P_PID`, `P_PGID`, `P_ALL`, or (on Linux) `P_PIDFD`. The interpretation of *id* depends on it; see their individual descriptions.

options is an OR combination of flags. At least one of `WEXITED`, `WSTOPPED` or `WCONTINUED` is required; `WNOHANG` and `WNOWAIT` are additional optional flags.

The return value is an object representing the data contained in the `siginfo_t` structure with the following attributes:

- `si_pid` (process ID)
- `si_uid` (real user ID of the child)
- `si_signo` (always `SIGCHLD`)
- `si_status` (the exit status or signal number, depending on `si_code`)
- `si_code` (see `CLD_EXITED` for possible values)

If `WNOHANG` is specified and there are no matching children in the requested state, `None` is returned. Otherwise, if there are no matching children that could be waited for, `ChildProcessError` is raised.

Availability: Unix, not WASI, not Android, not iOS.

Added in version 3.3.

Changed in version 3.13: This function is now available on macOS as well.

os.waitpid(*pid*, *options*, */*)

The details of this function differ on Unix and Windows.

On Unix: Wait for completion of a child process given by process id *pid*, and return a tuple containing its process id and exit status indication (encoded as for **`wait()`**). The semantics of the call are affected by the value of the integer *options*, which should be `0` for normal operation.

If *pid* is greater than `0`, **`waitpid()`** requests status information for that specific process. If *pid* is `0`, the request is for the status of any child in the process group of the current process. If *pid* is `-1`, the request pertains to any child of the current process. If *pid* is less than `-1`, status is requested for any process in the process group `-pid` (the absolute value of *pid*).

options is an OR combination of flags. If it contains **`WNOHANG`** and there are no matching children in the requested state, `(0, 0)` is returned. Otherwise, if there are no matching children that could be waited for, **`ChildProcessError`** is raised. Other options that can be used are **`WUNTRACED`** and **`WCONTINUED`**.

On Windows: Wait for completion of a process given by process handle *pid*, and return a tuple containing *pid*, and its exit status shifted left by 8 bits (shifting makes cross-platform use of the function easier). A *pid* less than or equal to `0` has no special meaning on Windows, and raises an exception. The value of integer *options* has no effect. *pid* can refer to any process whose id is known, not necessarily a child process. The **`spawn*`** functions called with **`P_NOWAIT`** return suitable process handles.

`waitstatus_to_exitcode()` can be used to convert the exit status into an exit code.

Availability: Unix, Windows, not WASI, not Android, not iOS.

Changed in version 3.5: If the system call is interrupted and the signal handler does not raise an exception, the function now retries the system call instead of raising an **`InterruptedError`** exception (see **PEP 475** [https://peps.python.org/pep-0475/] for the rationale).

os.wait3(*options*)

Similar to **`waitpid()`**, except no process id argument is given and a 3-element tuple containing the child’s process id, exit status indication, and resource usage information is returned. Refer to **`resource.getrusage()`** for details on resource usage information. The *options* argument is the same as that provided to **`waitpid()`** and **`wait4()`**.

`waitstatus_to_exitcode()` can be used to convert the exit status into an exitcode.

Availability: Unix, not WASI, not Android, not iOS.

os.wait4(*pid*, *options*)

Similar to **`waitpid()`**, except a 3-element tuple, containing the child’s process id, exit status indication, and resource usage information is returned. Refer to **`resource.getrusage()`** for details on resource usage information. The arguments to `wait4()` are the same as those provided to `waitpid()`.

`waitstatus_to_exitcode()` can be used to convert the exit status into an exitcode.

Availability: Unix, not WASI, not Android, not iOS.

os.P_PID
os.P_PGID
os.P_ALL
os.P_PIDFD

These are the possible values for *idtype* in `waitid()`. They affect how *id* is interpreted:

- **`P_PID`** - wait for the child whose PID is *id*.
- **`P_PGID`** - wait for any child whose progress group ID is *id*.
- **`P_ALL`** - wait for any child; *id* is ignored.
- **`P_PIDFD`** - wait for the child identified by the file descriptor *id* (a process file descriptor created with `pidfd_open()`).

Availability: Unix, not WASI, not Android, not iOS.

Note

`P_PIDFD` is only available on Linux >= 5.4.

Added in version 3.3.

Added in version 3.9: The **`P_PIDFD`** constant.

os.WCONTINUED

This *options* flag for `waitpid()`, `wait3()`, `wait4()`, and `waitid()` causes child processes to be reported if they have been continued from a job control stop since they were last reported.

Availability: Unix, not WASI, not Android, not iOS.

os.WEXITED

This *options* flag for **`waitid()`** causes child processes that have terminated to be reported.

The other `wait*` functions always report children that have terminated, so this option is not available for them.

Availability: Unix, not WASI, not Android, not iOS.

Added in version 3.3.

os.WSTOPPED

This *options* flag for **`waitid()`** causes child processes that have been stopped by the delivery of a signal to be reported.

This option is not available for the other `wait*` functions.

Availability: Unix, not WASI, not Android, not iOS.

Added in version 3.3.

os.WUNTRACED

This *options* flag for **`waitpid()`**, **`wait3()`**, and **`wait4()`** causes child processes to also be reported if they have been stopped but their current state has not been reported since they were stopped.

This option is not available for **`waitid()`**.

Availability: Unix, not WASI, not Android, not iOS.

os.WNOHANG

This *options* flag causes **`waitpid()`**, **`wait3()`**, **`wait4()`**, and **`waitid()`** to return right away if no child process status is available immediately.

Availability: Unix, not WASI, not Android, not iOS.

os.WNOWAIT

This *options* flag causes **`waitid()`** to leave the child in a waitable state, so that a later **`wait*()`** call can be used to retrieve the child status information again.

This option is not available for the other `wait*` functions.

Availability: Unix, not WASI, not Android, not iOS.

os.CLD_EXITED
os.CLD_KILLED
os.CLD_DUMPED
os.CLD_TRAPPED
os.CLD_STOPPED
os.CLD_CONTINUED

These are the possible values for **`si_code`** in the result returned by **`waitid()`**.

Availability: Unix, not WASI, not Android, not iOS.

Added in version 3.3.

Changed in version 3.9: Added **`CLD_KILLED`** and **`CLD_STOPPED`** values.

os.waitstatus_to_exitcode(*status*)

Convert a wait status to an exit code.

On Unix:

- If the process exited normally (if `WIFEXITED(status)` is true), return the process exit status (return `WEXITSTATUS(status)`): result greater than or equal to 0.
- If the process was terminated by a signal (if `WIFSIGNALED(status)` is true), return `-signum` where *signum* is the number of the signal that caused the process to terminate (return `-WTERMSIG(status)`): result less than 0.
- Otherwise, raise a **`ValueError`**.

On Windows, return *status* shifted right by 8 bits.

On Unix, if the process is being traced or if **`waitpid()`** was called with **`WUNTRACED`** option, the caller must first check if `WIFSTOPPED(status)` is true. This function must not be called if `WIFSTOPPED(status)` is true.

See also

`WIFEXITED()`, **`WEXITSTATUS()`**, **`WIFSIGNALED()`**, **`WTERMSIG()`**, **`WIFSTOPPED()`**, **`WSTOPSIG()`** functions.

Availability: Unix, Windows, not WASI, not Android, not iOS.

Added in version 3.9.

The following functions take a process status code as returned by **`system()`**, **`wait()`**, or **`waitpid()`** as a parameter. They may be used to determine the disposition of a process.

os.WCOREDUMP(*status*, */*)

Return `True` if a core dump was generated for the process, otherwise return `False`.

This function should be employed only if **`WIFSIGNALED()`** is true.

Availability: Unix, not WASI, not Android, not iOS.

os.WIFCONTINUED(*status*)

Return `True` if a stopped child has been resumed by delivery of **`SIGCONT`** (if the process has been continued from a job control stop), otherwise return `False`.

See **`WCONTINUED`** option.

Availability: Unix, not WASI, not Android, not iOS.

os.WIFSTOPPED(*status*)

Return `True` if the process was stopped by delivery of a signal, otherwise return `False`.

`WIFSTOPPED()` only returns `True` if the **`waitpid()`** call was done using **`WUNTRACED`** option or when the process is being traced (see *ptrace(2)* *[https://manpages.debian.org/ptrace(2)]*).

Availability: Unix, not WASI, not Android, not iOS.

os.WIFSIGNALED(*status*)

Return `True` if the process was terminated by a signal, otherwise return `False`.

Availability: Unix, not WASI, not Android, not iOS.

os.WIFEXITED(*status*)

Return `True` if the process exited terminated normally, that is, by calling `exit()` or `_exit()`, or by returning from `main()`; otherwise return `False`.

Availability: Unix, not WASI, not Android, not iOS.

os.WEXITSTATUS(*status*)

Return the process exit status.

This function should be employed only if **`WIFEXITED()`** is true.

Availability: Unix, not WASI, not Android, not iOS.

os.WSTOPSIG(*status*)

Return the signal which caused the process to stop.

This function should be employed only if **`WIFSTOPPED()`** is true.

Availability: Unix, not WASI, not Android, not iOS.

os.WTERMSIG(*status*)

Return the number of the signal that caused the process to terminate.

This function should be employed only if **`WIFSIGNALED()`** is true.

Availability: Unix, not WASI, not Android, not iOS.

Interface to the scheduler

These functions control how a process is allocated CPU time by the operating system. They are only available on some Unix platforms. For more detailed information, consult your Unix manpages.

Added in version 3.3.

The following scheduling policies are exposed if they are supported by the operating system.

os.SCHED_OTHER

The default scheduling policy.

os.SCHED_BATCH

Scheduling policy for CPU-intensive processes that tries to preserve interactivity on the rest of the computer.

os.SCHED_DEADLINE

Scheduling policy for tasks with deadline constraints.

Added in version 3.14.

os.SCHED_IDLE

Scheduling policy for extremely low priority background tasks.

os.SCHED_NORMAL

Alias for `SCHED_OTHER`.

Added in version 3.14.

os.SCHED_SPORADIC

Scheduling policy for sporadic server programs.

os.SCHED_FIFO

A First In First Out scheduling policy.

os.SCHED_RR

A round-robin scheduling policy.

os.SCHED_RESET_ON_FORK

This flag can be OR’ed with any other scheduling policy. When a process with this flag set forks, its child’s scheduling policy and priority are reset to the default.

class os.sched_param(*sched_priority*)

This class represents tunable scheduling parameters used in `sched_setparam()`, `sched_setscheduler()`, and `sched_getparam()`. It is immutable.

At the moment, there is only one possible parameter:

sched_priority
The scheduling priority for a scheduling policy.

os.sched_get_priority_min(*policy*)

Get the minimum priority value for *policy*. *policy* is one of the scheduling policy constants above.

os.sched_get_priority_max(*policy*)

Get the maximum priority value for *policy*. *policy* is one of the scheduling policy constants above.

os.sched_setscheduler(*pid*, *policy*, *param*, */*)

Set the scheduling policy for the process with PID *pid*. A *pid* of 0 means the calling process. *policy* is one of the scheduling policy constants above. *param* is a `sched_param` instance.

os.sched_getscheduler(*pid*, */*)

Return the scheduling policy for the process with PID *pid*. A *pid* of 0 means the calling process. The result is one of the scheduling policy constants above.

os.sched_setparam(*pid*, *param*, */*)

Set the scheduling parameters for the process with PID *pid*. A *pid* of 0 means the calling process. *param* is a `sched_param` instance.

os.sched_getparam(*pid*, */*)

Return the scheduling parameters as a `sched_param` instance for the process with PID *pid*. A *pid* of 0 means the calling process.

os.sched_rr_get_interval(*pid*, */*)

Return the round-robin quantum in seconds for the process with PID *pid*. A *pid* of 0 means the calling process.

os.sched_yield()

Voluntarily relinquish the CPU. See *sched_yield(2)* *[https://manpages.debian.org/sched_yield(2)]* for details.

os.sched_setaffinity(*pid*, *mask*, */*)

Restrict the process with PID *pid* (or the current process if zero) to a set of CPUs. *mask* is an iterable of integers representing the set of CPUs to which the process should be restricted.

os.sched_getaffinity(*pid*, */*)

Return the set of CPUs the process with PID *pid* is restricted to.

If *pid* is zero, return the set of CPUs the calling thread of the current process is restricted to.

See also the **`process_cpu_count()`** function.

Miscellaneous System Information

os.confstr(*name*, */*)

Return string-valued system configuration values. *name* specifies the configuration value to retrieve; it may be a string which is the name of a defined system value; these names are specified in a number of standards (POSIX, Unix 95, Unix 98, and others). Some platforms define additional names as well. The names known to the host operating system are given as the keys of the `confstr_names` dictionary. For configuration variables not included in that mapping, passing an integer for *name* is also accepted.

If the configuration value specified by *name* isn’t defined, `None` is returned.

If *name* is a string and is not known, **`ValueError`** is raised. If a specific value for *name* is not supported by the host system, even if it is included in `confstr_names`, an **`OSError`** is raised with **`errno.EINVAL`** for the error number.

Availability: Unix.

os.confstr_names

Dictionary mapping names accepted by **`confstr()`** to the integer values defined for those names by the host operating system. This can be used to determine the set of names known to the system.

Availability: Unix.

os.cpu_count()

Return the number of logical CPUs in the **system**. Returns `None` if undetermined.

The **`process_cpu_count()`** function can be used to get the number of logical CPUs usable by the calling thread of the **current process**.

Added in version 3.4.

Changed in version 3.13: If **`-X cpu_count`** is given or **`PYTHON_CPU_COUNT`** is set, **`cpu_count()`** returns the override value *n*.

os.getloadavg()

Return the number of processes in the system run queue averaged over the last 1, 5, and 15 minutes or raises `OSError` if the load average was unobtainable.

Availability: Unix.

os.process_cpu_count()

Get the number of logical CPUs usable by the calling thread of the **current process**. Returns `None` if undetermined. It can be less than `cpu_count()` depending on the CPU affinity.

The `cpu_count()` function can be used to get the number of logical CPUs in the **system**.

If `-X cpu_count` is given or `PYTHON_CPU_COUNT` is set, `process_cpu_count()` returns the override value *n*.

See also the `sched_getaffinity()` function.

Added in version 3.13.

os.sysconf(*name*, /)

Return integer-valued system configuration values. If the configuration value specified by *name* isn’t defined, `-1` is returned. The comments regarding the *name* parameter for `confstr()` apply here as well; the dictionary that provides information on the known names is given by `sysconf_names`.

Availability: Unix.

os.sysconf_names

Dictionary mapping names accepted by `sysconf()` to the integer values defined for those names by the host operating system. This can be used to determine the set of names known to the system.

Availability: Unix.

Changed in version 3.11: Add `'SC_MINSIGSTKSZ'` name.

The following data values are used to support path manipulation operations. These are defined for all platforms.

Higher-level operations on pathnames are defined in the **`os.path`** module.

os.curdir

The constant string used by the operating system to refer to the current directory. This is `'.'` for Windows and POSIX. Also available via **`os.path`**.

os.pardir

The constant string used by the operating system to refer to the parent directory. This is `'..'` for Windows and POSIX. Also available via **`os.path`**.

os.sep

The character used by the operating system to separate pathname components. This is `'/'` for POSIX and `'\\'` for Windows. Note that knowing this is not sufficient to be able to parse or concatenate pathnames — use **`os.path.split()`** and **`os.path.join()`** — but it is occasionally useful. Also available via **`os.path`**.

os.altsep

An alternative character used by the operating system to separate pathname components, or `None` if only one separator character exists. This is set to `'/'` on Windows systems where `sep` is a backslash. Also available via **`os.path`**.

os.extsep

The character which separates the base filename from the extension; for example, the `'.'` in `os.py`. Also available via **`os.path`**.

os.pathsep

The character conventionally used by the operating system to separate search path components (as in **`PATH`**), such as `':'` for POSIX or `';'` for Windows. Also available via **`os.path`**.

os.defpath

The default search path used by **`exec*p*`** and **`spawn*p*`** if the environment doesn't have a `'PATH'` key. Also available via **`os.path`**.

os.linesep

The string used to separate (or, rather, terminate) lines on the current platform. This may be a single character, such as `'\n'` for POSIX, or multiple characters, for example, `'\r\n'` for Windows. Do not use *os.linesep* as a line terminator when writing files opened in text mode (the default); use a single `'\n'` instead, on all platforms.

os.devnull

The file path of the null device. For example: `'/dev/null'` for POSIX, `'nul'` for Windows. Also available via **`os.path`**.

os.RTLD_LAZY
os.RTLD_NOW
os.RTLD_GLOBAL
os.RTLD_LOCAL
os.RTLD_NODELETE

os.RTLD_NOLOAD
os.RTLD_DEEPBIND

Flags for use with the **`setdlopenflags()`** and **`getdlopenflags()`** functions. See the Unix manual page *dlopen(3)* *[https://manpages.debian.org/dlopen(3)]* for what the different flags mean.

Added in version 3.3.

Random numbers

os.getrandom(*size*, *flags=0*)

Get up to *size* random bytes. The function can return less bytes than requested.

These bytes can be used to seed user-space random number generators or for cryptographic purposes.

`getrandom()` relies on entropy gathered from device drivers and other sources of environmental noise. Unnecessarily reading large quantities of data will have a negative impact on other users of the `/dev/random` and `/dev/urandom` devices.

The flags argument is a bit mask that can contain zero or more of the following values ORed together: **`os.GRND_RANDOM`** and **`GRND_NONBLOCK`**.

See also the Linux getrandom() manual page [https://man7.org/linux/man-pages/man2/getrandom.2.html].

Availability: Linux >= 3.17.

Added in version 3.6.

os.urandom(*size*, */*)

Return a bytestring of *size* random bytes suitable for cryptographic use.

This function returns random bytes from an OS-specific randomness source. The returned data should be unpredictable enough for cryptographic applications, though its exact quality depends on the OS implementation.

On Linux, if the `getrandom()` syscall is available, it is used in blocking mode: block until the system urandom entropy pool is initialized (128 bits of entropy are collected by the kernel). See the **PEP 524** [https://peps.python.org/pep-0524/] for the rationale. On Linux, the `getrandom()` function can be used to get random bytes in non-blocking mode (using the `GRND_NONBLOCK` flag) or to poll until the system urandom entropy pool is initialized.

On a Unix-like system, random bytes are read from the `/dev/urandom` device. If the `/dev/urandom` device is not available or not readable, the `NotImplementedError` exception is raised.

On Windows, it will use `BCryptGenRandom()`.

See also

The `secrets` module provides higher level functions. For an easy-to-use interface to the random number generator provided by your platform, please see `random.SystemRandom`.

Changed in version 3.5: On Linux 3.17 and newer, the `getrandom()` syscall is now used when available. On OpenBSD 5.6 and newer, the C `getentropy()` function is

now used. These functions avoid the usage of an internal file descriptor.
Changed in version 3.5.2: On Linux, if the `getrandom()` syscall blocks (the urandom entropy pool is not initialized yet), fall back on reading `/dev/urandom`.

Changed in version 3.6: On Linux, `getrandom()` is now used in blocking mode to increase the security.

Changed in version 3.11: On Windows, `BCryptGenRandom()` is used instead of `CryptGenRandom()` which is deprecated.

os.GRND_NONBLOCK

By default, when reading from `/dev/random`, **`getrandom()`** blocks if no random bytes are available, and when reading from `/dev/urandom`, it blocks if the entropy pool has not yet been initialized.

If the **`GRND_NONBLOCK`** flag is set, then **`getrandom()`** does not block in these cases, but instead immediately raises **`BlockingIOError`**.

Added in version 3.6.

os.GRND_RANDOM

If this bit is set, then random bytes are drawn from the `/dev/random` pool instead of the `/dev/urandom` pool.

Added in version 3.6.

`io` — Core tools for working with streams

Source code: Lib/io.py
[https://github.com/python/cpython/tree/3.14/Lib/io.py]

Overview

The **`io`** module provides Python’s main facilities for dealing with various types of I/O. There are three main types of I/O: *text I/O*, *binary I/O* and *raw I/O*. These are generic categories, and various backing stores can be used for each of them. A concrete object belonging to any of these categories is called a file object. Other common terms are *stream* and *file-like object*.

Independent of its category, each concrete stream object will also have various capabilities: it can be read-only, write-only, or read-write. It can also allow arbitrary random access (seeking forwards or backwards to any location), or only sequential access (for example in the case of a socket or pipe).

All streams are careful about the type of data you give to them. For example giving a **`str`** object to the **`write()`** method of a binary stream will raise a **`TypeError`**. So will giving a **`bytes`** object to the **`write()`** method of a text stream.

Changed in version 3.3: Operations that used to raise **`IOError`** now raise **`OSError`**, since **`IOError`** is now an alias of **`OSError`**.

Text I/O

Text I/O expects and produces **`str`** objects. This means that whenever the backing store is natively made of bytes (such as in the case of a file), encoding and decoding of data is made transparently as well as optional translation of platform-specific newline characters.

The easiest way to create a text stream is with **`open()`**, optionally specifying an encoding:

```
f = open("myfile.txt", "r", encoding="utf-8")
```

In-memory text streams are also available as **`StringIO`** objects:

```
f = io.StringIO("some initial text data")
```

Note

When working with a non-blocking stream, be aware that read operations on text I/O objects might raise a **`BlockingIOError`** if the stream cannot perform the operation immediately.

The text stream API is described in detail in the documentation of **`TextIOBase`**.

Binary I/O

Binary I/O (also called *buffered I/O*) expects bytes-like objects and produces **`bytes`** objects. No encoding, decoding, or newline translation is performed. This category of streams can be used for all kinds of non-text data, and also when manual control over the handling of text data is desired.

The easiest way to create a binary stream is with `open()` with `'b'` in the mode string:

```
f = open("myfile.jpg", "rb")
```

In-memory binary streams are also available as `BytesIO` objects:

```
f = io.BytesIO(b"some initial binary data: \x00\x01")
```

The binary stream API is described in detail in the docs of `BufferedIOBase`.

Other library modules may provide additional ways to create text or binary streams. See `socket.socket.makefile()` for example.

Raw I/O

Raw I/O (also called *unbuffered I/O*) is generally used as a low-level building-block for binary and text streams; it is rarely useful to directly manipulate a raw stream from user code. Nevertheless, you can create a raw stream by opening a file in binary mode with buffering disabled:

```
f = open("myfile.jpg", "rb", buffering=0)
```

The raw stream API is described in detail in the docs of `RawIOBase`.

Text Encoding

The default encoding of `TextIOWrapper` and `open()` is locale-specific (`locale.getencoding()`).

However, many developers forget to specify the encoding when opening text files encoded in UTF-8 (e.g. JSON, TOML, Markdown, etc…) since most Unix platforms use UTF-8 locale by default. This causes bugs because the locale encoding is not UTF-8 for most Windows users. For example:

```
# May not work on Windows when non-ASCII characters in the
file.
with open("README.md") as f:
    long_description = f.read()
```

Accordingly, it is highly recommended that you specify the encoding explicitly when opening text files. If you want to use UTF-8, pass `encoding="utf-8"`. To use the current locale encoding, `encoding="locale"` is supported since Python 3.10.

See also

Python UTF-8 Mode
Python UTF-8 Mode can be used to change the default encoding to UTF-8 from locale-specific encoding.

PEP 686 [https://peps.python.org/pep-0686/]
Python 3.15 will make Python UTF-8 Mode default.

Opt-in EncodingWarning

Added in version 3.10: See **PEP 597** [https://peps.python.org/pep-0597/] for more details.

To find where the default locale encoding is used, you can enable the **`-X warn_default_encoding`** command line option or set the **`PYTHONWARNDEFAULTENCODING`** environment variable, which will emit an **`EncodingWarning`** when the default encoding is used.

If you are providing an API that uses **`open()`** or **`TextIOWrapper`** and passes `encoding=None` as a parameter, you can use

`text_encoding()` so that callers of the API will emit an **`EncodingWarning`** if they don’t pass an `encoding`. However, please consider using UTF-8 by default (i.e. `encoding="utf-8"`) for new APIs.

High-level Module Interface

io.DEFAULT_BUFFER_SIZE

An int containing the default buffer size used by the module’s buffered I/O classes. **`open()`** uses the file’s blksize (as obtained by **`os.stat()`**) if possible.

io.open(*file*, *mode='r'*, *buffering=-1*, *encoding=None*, *errors=None*, *newline=None*, *closefd=True*, *opener=None*)

This is an alias for the builtin **`open()`** function.

This function raises an auditing event `open` with arguments *path*, *mode* and *flags*. The *mode* and *flags* arguments may have been modified or inferred from the original call.

io.open_code(*path*)

Opens the provided file with mode `'rb'`. This function should be used when the intent is to treat the contents as executable code.

path should be a **`str`** and an absolute path.

The behavior of this function may be overridden by an earlier call to the **`PyFile_SetOpenCodeHook()`**. However, assuming that *path* is a **`str`** and an absolute path, `open_code(path)` should always behave the same as `open(path, 'rb')`. Overriding the behavior is intended for additional validation or preprocessing of the file.

Added in version 3.8.

io.text_encoding(*encoding*, *stacklevel=2*, */*)

This is a helper function for callables that use **`open()`** or **`TextIOWrapper`** and have an `encoding=None` parameter.

This function returns *encoding* if it is not `None`. Otherwise, it returns `"locale"` or `"utf-8"` depending on UTF-8 Mode.

This function emits an **`EncodingWarning`** if **`sys.flags.warn_default_encoding`** is true and *encoding* is `None`. *stacklevel* specifies where the warning is emitted. For example:

```
def read_text(path, encoding=None):
    encoding = io.text_encoding(encoding)  # stacklevel=2
    with open(path, encoding) as f:
        return f.read()
```

In this example, an **`EncodingWarning`** is emitted for the caller of `read_text()`.

See Text Encoding for more information.

Added in version 3.10.

Changed in version 3.11: **`text_encoding()`** returns “utf-8” when UTF-8 mode is enabled and *encoding* is `None`.

exception io.BlockingIOError

This is a compatibility alias for the builtin **`BlockingIOError`** exception.

exception io.UnsupportedOperation

An exception inheriting **`OSError`** and **`ValueError`** that is raised when an unsupported operation is called on a stream.

See also

`sys`
contains the standard IO streams: `sys.stdin`, `sys.stdout`, and `sys.stderr`.

Class hierarchy

The implementation of I/O streams is organized as a hierarchy of classes. First abstract base classes (ABCs), which are used to specify the various categories of streams, then concrete classes providing the standard stream implementations.

Note

The abstract base classes also provide default implementations of some methods in order to help implementation of concrete stream classes. For example, `BufferedIOBase` provides unoptimized implementations of `readinto()` and `readline()`.

At the top of the I/O hierarchy is the abstract base class `IOBase`. It defines the basic interface to a stream. Note, however, that there is no separation between reading and writing to streams; implementations are allowed to raise `UnsupportedOperation` if they do not support a given operation.

The `RawIOBase` ABC extends `IOBase`. It deals with the reading and writing of bytes to a stream. `FileIO` subclasses `RawIOBase` to provide an interface to files in the machine’s file system.

The `BufferedIOBase` ABC extends `IOBase`. It deals with buffering on a raw binary stream (`RawIOBase`). Its subclasses, `BufferedWriter`, `BufferedReader`, and `BufferedRWPair` buffer raw binary streams that are writable, readable, and both readable and writable, respectively. `BufferedRandom` provides a buffered interface to seekable streams. Another `BufferedIOBase` subclass, `BytesIO`, is a stream of in-memory bytes.

The `TextIOBase` ABC extends `IOBase`. It deals with streams whose bytes represent text, and handles encoding and decoding to and from strings. `TextIOWrapper`, which extends `TextIOBase`, is a buffered text interface to a buffered raw stream (`BufferedIOBase`). Finally, `StringIO` is an in-memory stream for text.

Argument names are not part of the specification, and only the arguments of `open()` are intended to be used as keyword arguments.

The following table summarizes the ABCs provided by the `io` module:

ABC	Inherits	Stub Methods	Mixin Methods and Properties
`IOBase`		`fileno`, `seek`, and `truncate`	`close`, `closed`, `__enter__`, `__exit__`, `flush`, `isatty`, `__iter__`, `__next__`, `readable`, `readline`, `readlines`, `seekable`, `tell`, `writable`, and `writelines`

ABC	Inherits	Stub Methods	Mixin Methods and Properties
`RawIOBase`	`IOBase`	`readinto` and `write`	Inherited `IOBase` methods, `read`, and `readall`
`BufferedIOBase`	`IOBase`	`detach`, `read`, `read1`, and `write`	Inherited `IOBase` methods, `readinto`, and `readinto1`
`TextIOBase`	`IOBase`	`detach`, `read`, `readline`, and `write`	Inherited `IOBase` methods, `encoding`, `errors`, and `newlines`

I/O Base Classes

class io.IOBase

The abstract base class for all I/O classes.

This class provides empty abstract implementations for many methods that derived classes can override selectively; the default implementations represent a file that cannot be read, written or seeked.

Even though `IOBase` does not declare **`read()`** or **`write()`** because their signatures will vary, implementations and clients should consider those methods part of the interface. Also, implementations may raise a `ValueError` (or `UnsupportedOperation`) when operations they do not support are called.

The basic type used for binary data read from or written to a file is **`bytes`**. Other bytes-like objects are accepted as method arguments too. Text I/O classes work with **`str`** data.

Note that calling any method (even inquiries) on a closed stream is undefined. Implementations may raise **`ValueError`** in this case.

`IOBase` (and its subclasses) supports the iterator protocol, meaning that an **`IOBase`** object can be iterated over yielding the lines in a stream. Lines are defined slightly differently depending on whether the stream is a binary stream (yielding bytes), or a text stream (yielding character strings). See **`readline()`** below.

`IOBase` is also a context manager and therefore supports the **`with`** statement. In this example, *file* is closed after the **`with`** statement's suite is finished—even if an exception occurs:

```
with open('spam.txt', 'w') as file:
    file.write('Spam and eggs!')
```

`IOBase` provides these data attributes and methods:

close()

Flush and close this stream. This method has no effect if the file is already closed. Once the file is closed, any operation on the file (e.g. reading or writing) will raise a **`ValueError`**.

As a convenience, it is allowed to call this method more than once; only the first call, however, will have an effect.

closed

`True` if the stream is closed.

fileno()

Return the underlying file descriptor (an integer) of the stream if it exists. An `OSError` is raised if the IO object does not use a file descriptor.

flush()

Flush the write buffers of the stream if applicable. This does nothing for read-only and non-blocking streams.

isatty()

Return `True` if the stream is interactive (i.e., connected to a terminal/tty device).

readable()

Return `True` if the stream can be read from. If `False`, **`read()`** will raise `OSError`.

readline(*size=-1*, */*)

Read and return one line from the stream. If *size* is specified, at most *size* bytes will be read.

The line terminator is always `b'\n'` for binary files; for text files, the *newline* argument to `open()` can be used to select the line terminator(s) recognized.

readlines(*hint=-1*, */*)

Read and return a list of lines from the stream. *hint* can be specified to control the number of lines read: no more lines will be read if the total size (in bytes/characters) of all lines so far exceeds *hint*.

hint values of `0` or less, as well as `None`, are treated as no hint.

Note that it’s already possible to iterate on file objects using `for line in file: ...` without calling **`file.readlines()`**.

seek(*offset*, *whence=os.SEEK_SET*, */*)

Change the stream position to the given byte *offset*, interpreted relative to the position indicated by *whence*, and return the new absolute position. Values for *whence* are:

- `os.SEEK_SET` or `0` – start of the stream (the default); *offset* should be zero or positive
- `os.SEEK_CUR` or `1` – current stream position; *offset* may be negative
- `os.SEEK_END` or `2` – end of the stream; *offset* is usually negative

Added in version 3.1: The **`SEEK_*`** constants.

Added in version 3.3: Some operating systems could support additional values, like `os.SEEK_HOLE` or `os.SEEK_DATA`. The valid values for a file could depend on it being open in text or binary mode.

seekable()

Return `True` if the stream supports random access. If `False`, `seek()`, `tell()` and `truncate()` will raise `OSError`.

tell()

Return the current stream position.

truncate(*size=None*, */*)

Resize the stream to the given *size* in bytes (or the current position if *size* is not specified). The current stream position isn’t changed. This resizing can extend or reduce the current file size. In case of extension, the contents of the new file area depend on the platform (on most systems, additional bytes are zero-filled). The new file size is returned.

Changed in version 3.5: Windows will now zero-fill files when extending.

writable()

Return `True` if the stream supports writing. If `False`, **`write()`** and [`truncate()`] will raise [`OSError`].

writelines(*lines*, /)

Write a list of lines to the stream. Line separators are not added, so it is usual for each of the lines provided to have a line separator at the end.

__del__()

Prepare for object destruction. [`IOBase`] provides a default implementation of this method that calls the instance’s [`close()`] method.

class io.RawIOBase

Base class for raw binary streams. It inherits from [`IOBase`].

Raw binary streams typically provide low-level access to an underlying OS device or API, and do not try to encapsulate it in high-level primitives (this functionality is done at a higher-level in buffered binary streams and text streams, described later in this page).

[`RawIOBase`] provides these methods in addition to those from [`IOBase`]:

read(*size=-1*, /)

Read up to *size* bytes from the object and return them. As a convenience, if *size* is unspecified or -1, all bytes until EOF are returned. Otherwise, only one system call is ever made. Fewer than *size* bytes may be returned if the operating system call returns fewer than *size* bytes.

If 0 bytes are returned, and *size* was not 0, this indicates end of file. If the object is in non-blocking mode and no bytes are available, `None` is returned.

The default implementation defers to **`readall()`** and **`readinto()`**.

readall()

Read and return all the bytes from the stream until EOF, using multiple calls to the stream if necessary.

readinto(*b*, /)

Read bytes into a pre-allocated, writable bytes-like object *b*, and return the number of bytes read. For example, *b* might be a **`bytearray`**. If the object is in non-blocking mode and no bytes are available, `None` is returned.

write(*b*, /)

Write the given bytes-like object, *b*, to the underlying raw stream, and return the number of bytes written. This can be less than the length of *b* in bytes, depending on specifics of the underlying raw stream, and especially if it is in non-blocking mode. `None` is returned if the raw stream is set not to block and no single byte could be readily written to it. The caller may release or mutate *b* after this method returns, so the implementation should only access *b* during the method call.

class io.BufferedIOBase

Base class for binary streams that support some kind of buffering. It inherits from `IOBase`.

The main difference with `RawIOBase` is that methods `read()`, `readinto()` and `write()` will try (respectively) to read as much input as requested or to emit all provided data.

In addition, if the underlying raw stream is in non-blocking mode, when the system returns would block `write()` will raise `BlockingIOError` with `BlockingIOError.characters_written` and `read()` will return data read so far or `None` if no data is available.

Besides, the `read()` method does not have a default implementation that defers to `readinto()`.

A typical `BufferedIOBase` implementation should not inherit from a `RawIOBase` implementation, but wrap one, like `BufferedWriter` and `BufferedReader` do.

`BufferedIOBase` provides or overrides these data attributes and methods in addition to those from `IOBase`:

raw

> The underlying raw stream (a `RawIOBase` instance) that `BufferedIOBase` deals with. This is not part of the `BufferedIOBase` API and may not exist on some implementations.

detach()

> Separate the underlying raw stream from the buffer and return it.
>
> After the raw stream has been detached, the buffer is in an unusable state.

Some buffers, like `BytesIO`, do not have the concept of a single raw stream to return from this method. They raise `UnsupportedOperation`.

Added in version 3.1.

read(*size=-1*, */*)

Read and return up to *size* bytes. If the argument is omitted, `None`, or negative read as much as possible.

Fewer bytes may be returned than requested. An empty `bytes` object is returned if the stream is already at EOF. More than one read may be made and calls may be retried if specific errors are encountered, see `os.read()` and **PEP 475** [https://peps.python.org/pep-0475/] for more details. Less than size bytes being returned does not imply that EOF is imminent.

When reading as much as possible the default implementation will use `raw.readall` if available (which should implement `RawIOBase.readall()`), otherwise will read in a loop until read returns `None`, an empty `bytes`, or a non-retryable error. For most streams this is to EOF, but for non-blocking streams more data may become available.

Note

When the underlying raw stream is non-blocking, implementations may either raise `BlockingIOError` or return `None` if no data is available. `io` implementations return `None`.

read1(*size=-1*, */*)

Read and return up to *size* bytes, calling `readinto()` which may retry if `EINTR` is encountered per **PEP 475** [https://peps.python.org/pep-0475/]. If *size* is `-1` or not provided, the implementation will choose an arbitrary value for *size*.

Note

When the underlying raw stream is non-blocking, implementations may either raise **`BlockingIOError`** or return `None` if no data is available. **`io`** implementations return `None`.

readinto(*b*, /)

Read bytes into a pre-allocated, writable bytes-like object *b* and return the number of bytes read. For example, *b* might be a **`bytearray`**.

Like **`read()`**, multiple reads may be issued to the underlying raw stream, unless the latter is interactive.

A **`BlockingIOError`** is raised if the underlying raw stream is in non blocking-mode, and has no data available at the moment.

readinto1(*b*, /)

Read bytes into a pre-allocated, writable bytes-like object *b*, using at most one call to the underlying raw stream's **`read()`** (or **`readinto()`**) method. Return the number of bytes read.

A **`BlockingIOError`** is raised if the underlying raw stream is in non blocking-mode, and has no data available at the moment.

Added in version 3.5.

write(*b*, /)

Write the given bytes-like object, *b*, and return the number of bytes written (always equal to the length of *b* in bytes, since if the write fails an **`OSError`** will be raised). Depending on the actual implementation, these bytes may be readily written to the underlying stream, or held in a buffer for performance and latency reasons.

When in non-blocking mode, a **`BlockingIOError`** is raised if the data needed to be written to the raw stream but it couldn't accept all the data without blocking.

The caller may release or mutate *b* after this method returns, so the implementation should only access *b* during the method call.

Raw File I/O

class io.FileIO(*name*, *mode='r'*, *closefd=True*, *opener=None*)

A raw binary stream representing an OS-level file containing bytes data. It inherits from **`RawIOBase`**.

The *name* can be one of two things:

- a character string or **`bytes`** object representing the path to the file which will be opened. In this case closefd must be `True` (the default) otherwise an error will be raised.
- an integer representing the number of an existing OS-level file descriptor to which the resulting **`FileIO`** object will give access. When the FileIO object is closed this fd will be closed as well, unless *closefd* is set to `False`.

The *mode* can be `'r'`, `'w'`, `'x'` or `'a'` for reading (default), writing, exclusive creation or appending. The file will be created if it doesn’t exist when opened for writing or appending; it will be truncated when opened for writing. **`FileExistsError`** will be raised if it already exists when opened for creating. Opening a file for creating implies writing, so this mode behaves in a similar way to `'w'`. Add a `'+'` to the mode to allow simultaneous reading and writing.

The **`read()`** (when called with a positive argument), **`readinto()`** and **`write()`** methods on this class will only make one system call.

A custom opener can be used by passing a callable as *opener*. The underlying file descriptor for the file object is then obtained by calling *opener* with (*name*, *flags*). *opener* must return an open file descriptor (passing **`os.open`** as *opener* results in functionality similar to passing `None`).

The newly created file is non-inheritable.

See the **`open()`** built-in function for examples on using the *opener* parameter.

Changed in version 3.3: The *opener* parameter was added. The `'x'` mode was added.

Changed in version 3.4: The file is now non-inheritable.

`FileIO` provides these data attributes in addition to those from **`RawIOBase`** and **`IOBase`**:

mode

The mode as given in the constructor.

name

The file name. This is the file descriptor of the file when no name is given in the constructor.

Buffered Streams

Buffered I/O streams provide a higher-level interface to an I/O device than raw I/O does.

class io.BytesIO(*initial_bytes=b''*)

A binary stream using an in-memory bytes buffer. It inherits from **`BufferedIOBase`**. The buffer is discarded when the **`close()`** method is called.

The optional argument *initial_bytes* is a bytes-like object that contains initial data.

`BytesIO` provides or overrides these methods in addition to those from **`BufferedIOBase`** and **`IOBase`**:

getbuffer()

Return a readable and writable view over the contents of the buffer without copying them. Also, mutating the view will transparently update the contents of the buffer:

```
>>> b = io.BytesIO(b"abcdef")
>>> view = b.getbuffer()
>>> view[2:4] = b"56"
>>> b.getvalue()
b'ab56ef'
```

Note

As long as the view exists, the **`BytesIO`** object cannot be resized or closed.

Added in version 3.2.

getvalue()

Return `bytes` containing the entire contents of the buffer.

read1(*size=-1*, */*)

In `BytesIO`, this is the same as `read()`.

Changed in version 3.7: The *size* argument is now optional.

readinto1(*b*, */*)

In `BytesIO`, this is the same as `readinto()`.

Added in version 3.5.

class io.BufferedReader(*raw*, *buffer_size=DEFAULT_BUFFER_SIZE*)

A buffered binary stream providing higher-level access to a readable, non seekable `RawIOBase` raw binary stream. It inherits from `BufferedIOBase`.

When reading data from this object, a larger amount of data may be requested from the underlying raw stream, and kept in an internal buffer. The buffered data can then be returned directly on subsequent reads.

The constructor creates a `BufferedReader` for the given readable *raw* stream and *buffer_size*. If *buffer_size* is omitted, `DEFAULT_BUFFER_SIZE` is used.

`BufferedReader` provides or overrides these methods in addition to those from `BufferedIOBase` and `IOBase`:

peek(*size=0*, */*)

Return bytes from the stream without advancing the position. The number of bytes returned may be less or more than requested. If the underlying raw stream is non-blocking and the operation would block, returns empty bytes.

read(*size=-1*, */*)

In `BufferedReader` this is the same as `io.BufferedIOBase.read()`

read1(*size=-1*, */*)

In `BufferedReader` this is the same as `io.BufferedIOBase.read1()`

Changed in version 3.7: The *size* argument is now optional.

class io.BufferedWriter(*raw*, *buffer_size=DEFAULT_BUFFER_SIZE*)

A buffered binary stream providing higher-level access to a writeable, non seekable `RawIOBase` raw binary stream. It inherits from `BufferedIOBase`.

When writing to this object, data is normally placed into an internal buffer. The buffer will be written out to the underlying `RawIOBase` object under various conditions, including:

- when the buffer gets too small for all pending data;
- when `flush()` is called;
- when a `seek()` is requested (for `BufferedRandom` objects);
- when the `BufferedWriter` object is closed or destroyed.

The constructor creates a `BufferedWriter` for the given writeable *raw* stream. If the *buffer_size* is not given, it defaults to `DEFAULT_BUFFER_SIZE`.

`BufferedWriter` provides or overrides these methods in addition to those from `BufferedIOBase` and `IOBase`:

flush()

Force bytes held in the buffer into the raw stream. A `BlockingIOError` should be raised if the raw stream blocks.

write(*b*, /)

Write the bytes-like object, *b*, and return the number of bytes written. When in non-blocking mode, a `BlockingIOError` with `BlockingIOError.characters_written` set is raised if the buffer needs to be written out but the raw stream blocks.

class io.BufferedRandom(*raw*, *buffer_size=DEFAULT_BUFFER_SIZE*)

A buffered binary stream providing higher-level access to a seekable `RawIOBase` raw binary stream. It inherits from `BufferedReader` and `BufferedWriter`.

The constructor creates a reader and writer for a seekable raw stream, given in the first argument. If the *buffer_size* is omitted it defaults to `DEFAULT_BUFFER_SIZE`.

`BufferedRandom` is capable of anything `BufferedReader` or `BufferedWriter` can do. In addition, `seek()` and `tell()` are guaranteed to be implemented.

class io.BufferedRWPair(*reader*, *writer*, *buffer_size=DEFAULT_BUFFER_SIZE*, /)

A buffered binary stream providing higher-level access to two non seekable `RawIOBase` raw binary streams—one readable, the other writeable. It inherits from `BufferedIOBase`.

reader and *writer* are `RawIOBase` objects that are readable and writeable respectively. If the *buffer_size* is omitted it defaults to `DEFAULT_BUFFER_SIZE`.

`BufferedRWPair` implements all of `BufferedIOBase`'s methods except for `detach()`, which raises `UnsupportedOperation`.

Warning

`BufferedRWPair` does not attempt to synchronize accesses to its underlying raw streams. You should not pass it the same object as reader and writer; use `BufferedRandom` instead.

Text I/O

class io.TextIOBase

Base class for text streams. This class provides a character and line based interface to stream I/O. It inherits from `IOBase`.

`TextIOBase` provides or overrides these data attributes and methods in addition to those from `IOBase`:

encoding

The name of the encoding used to decode the stream’s bytes into strings, and to encode strings into bytes.

errors

The error setting of the decoder or encoder.

newlines

A string, a tuple of strings, or `None`, indicating the newlines translated so far. Depending on the implementation and the initial constructor flags, this may not be available.

buffer

The underlying binary buffer (a **`BufferedIOBase`** or **`RawIOBase`** instance) that **`TextIOBase`** deals with. This is not part of the **`TextIOBase`** API and may not exist in some implementations.

detach()

Separate the underlying binary buffer from the **`TextIOBase`** and return it.

After the underlying buffer has been detached, the **`TextIOBase`** is in an unusable state.

Some **`TextIOBase`** implementations, like **`StringIO`**, may not have the concept of an underlying buffer and calling this method will raise **`UnsupportedOperation`**.

Added in version 3.1.

read(*size=-1*, */*)

Read and return at most *size* characters from the stream as a single **`str`**. If *size* is negative or `None`, reads until EOF.

readline(*size=-1*, */*)

Read until newline or EOF and return a single **`str`**. If the stream is already at EOF, an empty string is returned.

If *size* is specified, at most *size* characters will be read.

seek(*offset*, *whence=SEEK_SET*, */*)

Change the stream position to the given *offset*. Behaviour depends on the *whence* parameter. The default value for *whence* is **`SEEK_SET`**.

- **`SEEK_SET`** or `0`: seek from the start of the stream (the default); *offset* must either be a number returned by [`TextIOBase.tell()`](), or zero. Any other *offset* value produces undefined behaviour.
- **`SEEK_CUR`** or `1`: “seek” to the current position; *offset* must be zero, which is a no-operation (all other values are unsupported).
- **`SEEK_END`** or `2`: seek to the end of the stream; *offset* must be zero (all other values are unsupported).

Return the new absolute position as an opaque number.

Added in version 3.1: The **`SEEK_*`** constants.

tell()

Return the current stream position as an opaque number. The number does not usually represent a number of bytes in the underlying binary storage.

write(*s*, */*)

Write the string *s* to the stream and return the number of characters written.

class io.TextIOWrapper(*buffer*, *encoding=None*, *errors=None*, *newline=None*, *line_buffering=False*, *write_through=False*)

A buffered text stream providing higher-level access to a `BufferedIOBase` buffered binary stream. It inherits from `TextIOBase`.

encoding gives the name of the encoding that the stream will be decoded or encoded with. In UTF-8 Mode, this defaults to UTF-8. Otherwise, it defaults to **`locale.getencoding()`**. `encoding="locale"` can be used to specify the current locale's encoding explicitly. See Text Encoding for more information.

errors is an optional string that specifies how encoding and decoding errors are to be handled. Pass `'strict'` to raise a **`ValueError`** exception if there is an encoding error (the default of `None` has the same effect), or pass `'ignore'` to ignore errors. (Note that ignoring encoding errors can lead to data loss.) `'replace'` causes a replacement marker (such as `'?'`) to be inserted where there is malformed data. `'backslashreplace'` causes malformed data to be replaced by a backslashed escape sequence. When writing, `'xmlcharrefreplace'` (replace with the appropriate XML character reference) or `'namereplace'` (replace with `\N{...}` escape sequences) can be used. Any other error handling name that has been registered with **`codecs.register_error()`** is also valid.

newline controls how line endings are handled. It can be `None`, `''`, `'\n'`, `'\r'`, and `'\r\n'`. It works as follows:

- When reading input from the stream, if *newline* is `None`, universal newlines mode is enabled. Lines in the input can end in `'\n'`, `'\r'`, or `'\r\n'`, and these are translated into `'\n'` before being returned to the caller. If *newline* is `''`, universal newlines mode is enabled, but line endings are returned to the caller untranslated. If *newline* has any of the other legal values, input lines are only terminated by the given string, and the line ending is returned to the caller untranslated.

- When writing output to the stream, if *newline* is `None`, any `'\n'` characters written are translated to the system default line separator, **`os.linesep`**. If *newline* is `''` or `'\n'`, no translation takes place. If *newline* is any of the other legal values, any `'\n'` characters written are translated to the given string.

If *line_buffering* is `True`, **`flush()`** is implied when a call to write contains a newline character or a carriage return.

If *write_through* is `True`, calls to **`write()`** are guaranteed not to be buffered: any data written on the **`TextIOWrapper`** object is immediately handled to its underlying binary *buffer*.

Changed in version 3.3: The *write_through* argument has been added.

Changed in version 3.3: The default *encoding* is now `locale.getpreferredencoding(False)` instead of `locale.getpreferredencoding()`. Don’t change temporary the locale encoding using **`locale.setlocale()`**, use the current locale encoding instead of the user preferred encoding.

Changed in version 3.10: The *encoding* argument now supports the `"locale"` dummy encoding name.

Note

When the underlying raw stream is non-blocking, a **`BlockingIOError`** may be raised if a read operation cannot be completed immediately.

`TextIOWrapper` provides these data attributes and methods in addition to those from **`TextIOBase`** and **`IOBase`**:

line_buffering

Whether line buffering is enabled.

write_through

Whether writes are passed immediately to the underlying binary buffer.

Added in version 3.7.

reconfigure(*, *encoding=None*, *errors=None*, *newline=None*, *line_buffering=None*, *write_through=None*)

Reconfigure this text stream using new settings for *encoding*, *errors*, *newline*, *line_buffering* and *write_through*.

Parameters not specified keep current settings, except `errors='strict'` is used when *encoding* is specified but *errors* is not specified.

It is not possible to change the encoding or newline if some data has already been read from the stream. On the other hand, changing encoding after write is possible.

This method does an implicit stream flush before setting the new parameters.

Added in version 3.7.

Changed in version 3.11: The method supports `encoding="locale"` option.

seek(*cookie*, *whence=os.SEEK_SET*, /)

Set the stream position. Return the new stream position as an `int`.

Four operations are supported, given by the following argument combinations:

- `seek(0, SEEK_SET)`: Rewind to the start of the stream.
- `seek(cookie, SEEK_SET)`: Restore a previous position; *cookie* **must be** a number returned by `tell()`.
- `seek(0, SEEK_END)`: Fast-forward to the end of the stream.
- `seek(0, SEEK_CUR)`: Leave the current stream position unchanged.

Any other argument combinations are invalid, and may raise exceptions.

See also

`os.SEEK_SET`, `os.SEEK_CUR`, and `os.SEEK_END`.

tell()

Return the stream position as an opaque number. The return value of **`tell()`** can be given as input to `seek()`, to restore a previous stream position.

class io.StringIO(*initial_value*='', *newline*='*n*')

A text stream using an in-memory text buffer. It inherits from `TextIOBase`.

The text buffer is discarded when the `close()` method is called.

The initial value of the buffer can be set by providing *initial_value*. If newline translation is enabled, newlines will be encoded as if by `write()`. The stream is positioned at the start of the buffer which emulates opening an existing file in a `w+` mode, making it ready for an immediate write from the beginning or for a write that would overwrite the initial value. To emulate opening a file in an `a+` mode ready for appending, use `f.seek(0, io.SEEK_END)` to reposition the stream at the end of the buffer.

The *newline* argument works like that of **`TextIOWrapper`**, except that when writing output to the stream, if *newline* is `None`, newlines are written as `\n` on all platforms.

`StringIO` provides this method in addition to those from **`TextIOBase`** and **`IOBase`**:

getvalue()

Return a **`str`** containing the entire contents of the buffer. Newlines are decoded as if by **`read()`**, although the stream position is not changed.

Example usage:

```
import io

output = io.StringIO()
output.write('First line.\n')
print('Second line.', file=output)

# Retrieve file contents -- this will be
# 'First line.\nSecond line.\n'
contents = output.getvalue()

# Close object and discard memory buffer --
# .getvalue() will now raise an exception.
output.close()
```

class io.IncrementalNewlineDecoder

A helper codec that decodes newlines for universal newlines mode. It inherits from **`codecs.IncrementalDecoder`**.

Static Typing

The following protocols can be used for annotating function and method arguments for simple stream reading or writing operations. They are decorated with `@typing.runtime_checkable`.

class io.Reader[*T*]

Generic protocol for reading from a file or other input stream. `T` will usually be `str` or `bytes`, but can be any type that is read from the stream.

Added in version 3.14.

read()

read(*size*, */*)

Read data from the input stream and return it. If *size* is specified, it should be an integer, and at most *size* items (bytes/characters) will be read.

For example:

```
def read_it(reader: Reader[str]):
    data = reader.read(11)
    assert isinstance(data, str)
```

class io.Writer[*T*]

Generic protocol for writing to a file or other output stream. `T` will usually be `str` or `bytes`, but can be any type that can be written to the stream.

Added in version 3.14.

write(*data*, */*)

Write *data* to the output stream and return the number of items (bytes/characters) written.

For example:

```
def write_binary(writer: Writer[bytes]):
    writer.write(b"Hello world!\n")
```

See ABCs and Protocols for working with I/O for other I/O related protocols and classes that can be used for static type checking.

Performance

This section discusses the performance of the provided concrete I/O implementations.

Binary I/O

By reading and writing only large chunks of data even when the user asks for a single byte, buffered I/O hides any inefficiency in calling and executing the operating system’s unbuffered I/O routines. The gain depends on the OS and the kind of I/O which is performed. For example, on some modern OSes such as Linux, unbuffered disk I/O can be as fast as buffered I/O. The bottom line, however, is that buffered I/O offers predictable performance regardless of the platform and the backing device. Therefore, it is almost always preferable to use buffered I/O rather than unbuffered I/O for binary data.

Text I/O

Text I/O over a binary storage (such as a file) is significantly slower than binary I/O over the same storage, because it requires conversions between unicode and binary data using a character codec. This can become noticeable handling huge amounts of text data like large log files. Also, `tell()` and `seek()` are both quite slow due to the reconstruction algorithm used.

`StringIO`, however, is a native in-memory unicode container and will exhibit similar speed to `BytesIO`.

Multi-threading

`FileIO` objects are thread-safe to the extent that the operating system calls (such as *read(2)* *[https://manpages.debian.org/read(2)]* under Unix) they wrap are thread-safe too.

Binary buffered objects (instances of `BufferedReader`, `BufferedWriter`, `BufferedRandom` and `BufferedRWPair`) protect their internal structures using a lock; it is therefore safe to call them from multiple threads at once.

`TextIOWrapper` objects are not thread-safe.

Reentrancy

Binary buffered objects (instances of `BufferedReader`, `BufferedWriter`, `BufferedRandom` and `BufferedRWPair`) are not reentrant. While reentrant calls will not happen in normal situations, they can arise from doing I/O in a `signal` handler. If a thread tries to re-enter a buffered object which it is already accessing, a `RuntimeError` is raised. Note this doesn’t prohibit a different thread from entering the buffered object.

The above implicitly extends to text files, since the `open()` function will wrap a buffered object inside a `TextIOWrapper`. This includes standard streams and therefore affects the built-in `print()` function as well.

`time` — Time access and conversions

This module provides various time-related functions. For related functionality, see also the **`datetime`** and **`calendar`** modules.

Although this module is always available, not all functions are available on all platforms. Most of the functions defined in this module call platform C library functions with the same name. It may sometimes be helpful to consult the platform documentation, because the semantics of these functions varies among platforms.

An explanation of some terminology and conventions is in order.

- The *epoch* is the point where the time starts, the return value of `time.gmtime(0)`. It is January 1, 1970, 00:00:00 (UTC) on all platforms.
- The term *seconds since the epoch* refers to the total number of elapsed seconds since the epoch, typically excluding leap seconds [https://en.wikipedia.org/wiki/Leap_second]. Leap seconds are excluded from this total on all POSIX-compliant platforms.
- The functions in this module may not handle dates and times before the epoch or far in the future. The cut-off point in the future is determined by the C library; for 32-bit systems, it is typically in 2038.

- Function `strptime()` can parse 2-digit years when given `%y` format code. When 2-digit years are parsed, they are converted according to the POSIX and ISO C standards: values 69–99 are mapped to 1969–1999, and values 0–68 are mapped to 2000–2068.

- UTC is Coordinated Universal Time [https://en.wikipedia.org/wiki/Coordinated_Universal_Time] and superseded Greenwich Mean Time [https://en.wikipedia.org/wiki/Greenwich_Mean_Time] or GMT as the basis of international timekeeping. The acronym UTC is not a mistake but conforms to an earlier, language-agnostic naming scheme for time standards such as UT0, UT1, and UT2.

- DST is Daylight Saving Time, an adjustment of the timezone by (usually) one hour during part of the year. DST rules are magic (determined by local law) and can change from year to year. The C library has a table containing the local rules (often it is read from a system file for flexibility) and is the only source of True Wisdom in this respect.

- The precision of the various real-time functions may be less than suggested by the units in which their value or argument is expressed. E.g. on most Unix systems, the clock “ticks” only 50 or 100 times a second.

- On the other hand, the precision of `time()` and `sleep()` is better than their Unix equivalents: times are expressed as floating-point numbers, `time()` returns the most accurate time available (using Unix **`gettimeofday()`** where available), and `sleep()` will accept a time with a nonzero fraction (Unix **`select()`** is used to implement this, where available).

- The time value as returned by `gmtime()`, `localtime()`, and `strptime()`, and accepted by `asctime()`, `mktime()` and `strftime()`, is a sequence of 9 integers. The return values of `gmtime()`, `localtime()`, and `strptime()` also offer attribute names for individual fields.

  See `struct_time` for a description of these objects.

  Changed in version 3.3: The `struct_time` type was extended to provide the `tm_gmtoff` and `tm_zone` attributes when platform supports corresponding `struct tm` members.

  Changed in version 3.6: The `struct_time` attributes `tm_gmtoff` and `tm_zone` are now available on all platforms.

- Use the following functions to convert between time representations:

From	To	Use
seconds since the epoch	`struct_time` in UTC	`gmtime()`
seconds since the epoch	`struct_time` in local time	`localtime()`
`struct_time` in UTC	seconds since the epoch	`calendar.timegm()`
`struct_time` in local time	seconds since the epoch	`mktime()`

From	To	Use

Functions

time.asctime([*t*])

Convert a tuple or `struct_time` representing a time as returned by `gmtime()` or `localtime()` to a string of the following form: `'Sun Jun 20 23:21:05 1993'`. The day field is two characters long and is space padded if the day is a single digit, e.g.: `'Wed Jun  9 04:26:40 1993'`.

If *t* is not provided, the current time as returned by `localtime()` is used. Locale information is not used by `asctime()`.

Note

Unlike the C function of the same name, `asctime()` does not add a trailing newline.

time.pthread_getcpuclockid(*thread_id*)

Return the *clk_id* of the thread-specific CPU-time clock for the specified *thread_id*.

Use `threading.get_ident()` or the `ident` attribute of `threading.Thread` objects to get a suitable value for *thread_id*.

Warning

Passing an invalid or expired *thread_id* may result in undefined behavior, such as segmentation fault.

Availability: Unix

See the man page for *pthread_getcpuclockid(3)* *[https://manpages.debian.org/pthread_getcpuclockid(3)]* for further information.

Added in version 3.7.

time.clock_getres(*clk_id*)

Return the resolution (precision) of the specified clock *clk_id*. Refer to Clock ID Constants for a list of accepted values for *clk_id*.

Availability: Unix.

Added in version 3.3.

time.clock_gettime(*clk_id*) → float

Return the time of the specified clock *clk_id*. Refer to Clock ID Constants for a list of accepted values for *clk_id*.

Use **`clock_gettime_ns()`** to avoid the precision loss caused by the **`float`** type.

Availability: Unix.

Added in version 3.3.

time.clock_gettime_ns(*clk_id*) → int

Similar to **`clock_gettime()`** but return time as nanoseconds.

Availability: Unix.

Added in version 3.7.

time.clock_settime(*clk_id*, *time: float*)

Set the time of the specified clock *clk_id*. Currently, **`CLOCK_REALTIME`** is the only accepted value for *clk_id*.

Use **`clock_settime_ns()`** to avoid the precision loss caused by the **`float`** type.

Availability: Unix, not Android, not iOS.

Added in version 3.3.

time.clock_settime_ns(*clk_id*, *time*: *int*)

Similar to **`clock_settime()`** but set time with nanoseconds.

Availability: Unix, not Android, not iOS.

Added in version 3.7.

time.ctime([*secs*])

Convert a time expressed in seconds since the epoch to a string of a form: `'Sun Jun 20 23:21:05 1993'` representing local time. The day field is two characters long and is space padded if the day is a single digit, e.g.: `'Wed Jun  9 04:26:40 1993'`.

If *secs* is not provided or **`None`**, the current time as returned by **`time()`** is used. `ctime(secs)` is equivalent to `asctime(localtime(secs))`. Locale information is not used by **`ctime()`**.

time.get_clock_info(*name*)

Get information on the specified clock as a namespace object. Supported clock names and the corresponding functions to read their value are:

- `'monotonic'`: **`time.monotonic()`**
- `'perf_counter'`: **`time.perf_counter()`**

- `'process_time'`: **`time.process_time()`**
- `'thread_time'`: **`time.thread_time()`**
- `'time'`: **`time.time()`**

The result has the following attributes:

- *adjustable*: `True` if the clock can be set to jump forward or backward in time, `False` otherwise. Does not refer to gradual NTP rate adjustments.
- *implementation*: The name of the underlying C function used to get the clock value. Refer to Clock ID Constants for possible values.
- *monotonic*: `True` if the clock cannot go backward, `False` otherwise
- *resolution*: The resolution of the clock in seconds (**`float`**)

Added in version 3.3.

time.gmtime([*secs*])

Convert a time expressed in seconds since the epoch to a **`struct_time`** in UTC in which the dst flag is always zero. If *secs* is not provided or **`None`**, the current time as returned by **`time()`** is used. Fractions of a second are ignored. See above for a description of the **`struct_time`** object. See **`calendar.timegm()`** for the inverse of this function.

time.localtime([*secs*])

Like **`gmtime()`** but converts to local time. If *secs* is not provided or **`None`**, the current time as returned by **`time()`** is used. The dst flag is set to `1` when DST applies to the given time.

`localtime()` may raise **`OverflowError`**, if the timestamp is outside the range of values supported by the platform C **`localtime()`** or **`gmtime()`** functions, and **`OSError`** on

`localtime()` or **`gmtime()`** failure. It's common for this to be restricted to years between 1970 and 2038.

time.mktime(*t*)

This is the inverse function of **`localtime()`**. Its argument is the **`struct_time`** or full 9-tuple (since the dst flag is needed; use `-1` as the dst flag if it is unknown) which expresses the time in *local* time, not UTC. It returns a floating-point number, for compatibility with **`time()`**. If the input value cannot be represented as a valid time, either **`OverflowError`** or **`ValueError`** will be raised (which depends on whether the invalid value is caught by Python or the underlying C libraries). The earliest date for which it can generate a time is platform-dependent.

time.monotonic() → float

Return the value (in fractional seconds) of a monotonic clock, i.e. a clock that cannot go backwards. The clock is not affected by system clock updates. The reference point of the returned value is undefined, so that only the difference between the results of two calls is valid.

Clock:

- On Windows, call `QueryPerformanceCounter()` and `QueryPerformanceFrequency()`.
- On macOS, call `mach_absolute_time()` and `mach_timebase_info()`.
- On HP-UX, call `gethrtime()`.
- Call `clock_gettime(CLOCK_HIGHRES)` if available.
- Otherwise, call `clock_gettime(CLOCK_MONOTONIC)`.

Use **`monotonic_ns()`** to avoid the precision loss caused by the **`float`** type.

Added in version 3.3.

Changed in version 3.5: The function is now always available and the clock is now the same for all processes.

Changed in version 3.10: On macOS, the clock is now the same for all processes.

time.monotonic_ns() → int

Similar to `monotonic()`, but return time as nanoseconds.

Added in version 3.7.

time.perf_counter() → float

Return the value (in fractional seconds) of a performance counter, i.e. a clock with the highest available resolution to measure a short duration. It does include time elapsed during sleep. The clock is the same for all processes. The reference point of the returned value is undefined, so that only the difference between the results of two calls is valid.

CPython implementation detail: On CPython, use the same clock as `time.monotonic()` and is a monotonic clock, i.e. a clock that cannot go backwards.

Use `perf_counter_ns()` to avoid the precision loss caused by the `float` type.

Added in version 3.3.

Changed in version 3.10: On Windows, the clock is now the same for all processes.

Changed in version 3.13: Use the same clock as `time.monotonic()`.

time.perf_counter_ns() → int

Similar to `perf_counter()`, but return time as nanoseconds.

Added in version 3.7.

time.process_time() → float

Return the value (in fractional seconds) of the sum of the system and user CPU time of the current process. It does not include time elapsed during sleep. It is process-wide by definition. The reference point of the returned value is undefined, so that only the difference between the results of two calls is valid.

Use `process_time_ns()` to avoid the precision loss caused by the `float` type.

Added in version 3.3.

time.process_time_ns() → int

Similar to `process_time()` but return time as nanoseconds.

Added in version 3.7.

time.sleep(*secs*)

Suspend execution of the calling thread for the given number of seconds. The argument may be a floating-point number to indicate a more precise sleep time.

If the sleep is interrupted by a signal and no exception is raised by the signal handler, the sleep is restarted with a recomputed timeout.

The suspension time may be longer than requested by an arbitrary amount, because of the scheduling of other activity in the system.

Windows implementation

On Windows, if *secs* is zero, the thread relinquishes the remainder of its time slice to any other thread that is ready to run. If there are no other threads ready to run, the function returns immediately, and the thread continues execution. On Windows 10 and newer the implementation uses a high-resolution timer [https://learn.microsoft.com/windows/win32/api/synchapi/nf-synchapi-createwaitabletimerexw] which provides resolution of 100 nanoseconds. If *secs* is zero, `Sleep(0)` is used.

Unix implementation

- Use `clock_nanosleep()` if available (resolution: 1 nanosecond);
- Or use `nanosleep()` if available (resolution: 1 nanosecond);
- Or use `select()` (resolution: 1 microsecond).

Note

To emulate a "no-op", use **`pass`** instead of `time.sleep(0)`.

To voluntarily relinquish the CPU, specify a real-time scheduling policy and use **`os.sched_yield()`** instead.

Raises an auditing event `time.sleep` with argument `secs`.

Changed in version 3.5: The function now sleeps at least *secs* even if the sleep is interrupted by a signal, except if the signal handler raises an exception (see **PEP 475** [https://peps.python.org/pep-0475/] for the rationale).

Changed in version 3.11: On Unix, the `clock_nanosleep()` and `nanosleep()` functions are now used if available. On Windows, a waitable timer is now used.

Changed in version 3.13: Raises an auditing event.

time.strftime(*format*[, *t*])

Convert a tuple or `struct_time` representing a time as returned by `gmtime()` or `localtime()` to a string as specified by the *format* argument. If *t* is not provided, the current time as returned by `localtime()` is used. *format* must be a string. `ValueError` is raised if any field in *t* is outside of the allowed range.

0 is a legal argument for any position in the time tuple; if it is normally illegal the value is forced to a correct one.

The following directives can be embedded in the *format* string. They are shown without the optional field width and precision specification, and are replaced by the indicated characters in the `strftime()` result:

Directive	Meaning	Notes
`%a`	Locale’s abbreviated weekday name.	
`%A`	Locale’s full weekday name.	
`%b`	Locale’s abbreviated month name.	
`%B`	Locale’s full month name.	
`%c`	Locale’s appropriate date and time representation.	
`%d`	Day of the month as a decimal number [01,31].	
`%f`	Microseconds as a decimal number [000000,999999].	(1)

Directive	Meaning	Notes
`%H`	Hour (24-hour clock) as a decimal number [00,23].	
`%I`	Hour (12-hour clock) as a decimal number [01,12].	
`%j`	Day of the year as a decimal number [001,366].	
`%m`	Month as a decimal number [01,12].	
`%M`	Minute as a decimal number [00,59].	
`%p`	Locale’s equivalent of either AM or PM.	(2)
`%S`	Second as a decimal number [00,61].	(3)
`%U`	Week number of the year (Sunday as the first day of the week) as a decimal number [00,53]. All days in a new year preceding the first Sunday are considered to be in week 0.	(4)
`%u`	Day of the week (Monday is 1; Sunday is 7) as a decimal number [1, 7].	
`%w`	Weekday as a decimal number [0(Sunday),6].	

Directive	Meaning	Notes
`%W`	Week number of the year (Monday as the first day of the week) as a decimal number [00,53]. All days in a new year preceding the first Monday are considered to be in week 0.	(4)
`%x`	Locale’s appropriate date representation.	
`%X`	Locale’s appropriate time representation.	
`%y`	Year without century as a decimal number [00,99].	
`%Y`	Year with century as a decimal number.	
`%z`	Time zone offset indicating a positive or negative time difference from UTC/GMT of the form +HHMM or -HHMM, where H represents decimal hour digits and M represents decimal minute digits [-23:59, +23:59]. [[1]]	
`%Z`	Time zone name (no characters if no time zone exists). Deprecated. [[1]]	
`%G`	ISO 8601 year (similar to `%Y` but follows the rules for the ISO 8601 calendar year). The year starts with the week that contains the first Thursday of the calendar year.	

Directive	Meaning	Notes
`%V`	ISO 8601 week number (as a decimal number [01,53]). The first week of the year is the one that contains the first Thursday of the year. Weeks start on Monday.	
`%%`	A literal `'%'` character.	

Notes:

1. The `%f` format directive only applies to **`strptime()`**, not to **`strftime()`**. However, see also **`datetime.datetime.strptime()`** and **`datetime.datetime.strftime()`** where the `%f` format directive applies to microseconds.
2. When used with the **`strptime()`** function, the `%p` directive only affects the output hour field if the `%I` directive is used to parse the hour.
3. The range really is `0` to `61`; value `60` is valid in timestamps representing leap seconds [https://en.wikipedia.org/wiki/Leap_second] and value `61` is supported for historical reasons.
4. When used with the **`strptime()`** function, `%U` and `%W` are only used in calculations when the day of the week and the year are specified.

Here is an example, a format for dates compatible with that specified in the **RFC 5322** [https://datatracker.ietf.org/doc/html/rfc5322.html] Internet email standard. [[1]]

```
>>> from time import gmtime, strftime
>>> strftime("%a, %d %b %Y %H:%M:%S +0000", gmtime())
'Thu, 28 Jun 2001 14:17:15 +0000'
```

Additional directives may be supported on certain platforms, but only the ones listed here have a meaning standardized by ANSI C. To see the full set of format codes supported on your platform, consult the *strftime(3)* *[https://manpages.debian.org/strftime(3)]* documentation.

On some platforms, an optional field width and precision specification can immediately follow the initial `'%'` of a directive in the following order; this is also not portable. The field width is normally 2 except for `%j` where it is 3.

time.strptime(*string*[, *format*])

Parse a string representing a time according to a format. The return value is a **`struct_time`** as returned by **`gmtime()`** or **`localtime()`**.

The *format* parameter uses the same directives as those used by **`strftime()`**; it defaults to `"%a %b %d %H:%M:%S %Y"` which matches the formatting returned by **`ctime()`**. If *string* cannot be parsed according to *format*, or if it has excess data after parsing, **`ValueError`** is raised. The default values used to fill in any missing data when more accurate values cannot be inferred are `(1900, 1, 1, 0, 0, 0, 0, 1, -1)`. Both *string* and *format* must be strings.

For example:

```
>>> import time
>>> time.strptime("30 Nov 00", "%d %b %y")
time.struct_time(tm_year=2000, tm_mon=11, tm_mday=30,
tm_hour=0, tm_min=0,
                 tm_sec=0, tm_wday=3, tm_yday=335,
tm_isdst=-1)
```

Support for the `%Z` directive is based on the values contained in `tzname` and whether `daylight` is true. Because of this, it is platform-specific except for recognizing UTC and GMT which are always known (and are considered to be non-daylight savings timezones).

Only the directives specified in the documentation are supported. Because `strftime()` is implemented per platform it can sometimes offer more directives than those listed. But `strptime()` is independent of any platform and thus does not necessarily support all directives available that are not documented as supported.

class time.struct_time

The type of the time value sequence returned by **`gmtime()`**, **`localtime()`**, and **`strptime()`**. It is an object with a named tuple interface: values can be accessed by index and by attribute name. The following values are present:

Index	Attribute	Values
0	tm_year	(for example, 1993)
1	tm_mon	range [1, 12]
2	tm_mday	range [1, 31]
3	tm_hour	range [0, 23]
4	tm_min	range [0, 59]
5	tm_sec	range [0, 61]; see Note (2) in **`strftime()`**
6	tm_wday	range [0, 6]; Monday is 0
7	tm_yday	range [1, 366]

8	tm_isdst	0, 1 or -1; see below
N/A	tm_zone	abbreviation of timezone name
N/A	tm_gmtoff	offset east of UTC in seconds

Note that unlike the C structure, the month value is a range of [1, 12], not [0, 11].

In calls to `mktime()`, `tm_isdst` may be set to 1 when daylight savings time is in effect, and 0 when it is not. A value of -1 indicates that this is not known, and will usually result in the correct state being filled in.

When a tuple with an incorrect length is passed to a function expecting a `struct_time`, or having elements of the wrong type, a `TypeError` is raised.

time.time() → float

Return the time in seconds since the epoch as a floating-point number. The handling of leap seconds [https://en.wikipedia.org/wiki/Leap_second] is platform dependent. On Windows and most Unix systems, the leap seconds are not counted towards the time in seconds since the epoch. This is commonly referred to as Unix time [https://en.wikipedia.org/wiki/Unix_time].

Note that even though the time is always returned as a floating-point number, not all systems provide time with a better precision than 1 second. While this function normally returns non-decreasing values, it can return a lower value than a previous call if the system clock has been set back between the two calls.

The number returned by `time()` may be converted into a more common time format (i.e. year, month, day, hour, etc…) in UTC by passing it to `gmtime()` function or in local time by passing it to the `localtime()` function. In both cases a `struct_time` object is returned, from which the components of the calendar date may be accessed as attributes.

Clock:

- On Windows, call `GetSystemTimePreciseAsFileTime()`.
- Call `clock_gettime(CLOCK_REALTIME)` if available.
- Otherwise, call `gettimeofday()`.

Use `time_ns()` to avoid the precision loss caused by the `float` type.

Changed in version 3.13: On Windows, calls `GetSystemTimePreciseAsFileTime()` instead of `GetSystemTimeAsFileTime()`.

time.time_ns() → int

Similar to `time()` but returns time as an integer number of nanoseconds since the epoch.

Added in version 3.7.

time.thread_time() → float

Return the value (in fractional seconds) of the sum of the system and user CPU time of the current thread. It does not include time elapsed during sleep. It is thread-specific by definition. The reference point of the returned value is undefined, so that only the difference between the results of two calls in the same thread is valid.

Use `thread_time_ns()` to avoid the precision loss caused by the `float` type.

Availability: Linux, Unix, Windows.

Unix systems supporting `CLOCK_THREAD_CPUTIME_ID`.
Added in version 3.7.

time.thread_time_ns() → int

Similar to **`thread_time()`** but return time as nanoseconds.

Added in version 3.7.

time.tzset()

Reset the time conversion rules used by the library routines. The environment variable **`TZ`** specifies how this is done. It will also set the variables `tzname` (from the **`TZ`** environment variable), `timezone` (non-DST seconds West of UTC), `altzone` (DST seconds west of UTC) and `daylight` (to 0 if this timezone does not have any daylight saving time rules, or to nonzero if there is a time, past, present or future when daylight saving time applies).

Availability: Unix.

Note

Although in many cases, changing the **`TZ`** environment variable may affect the output of functions like **`localtime()`** without calling **`tzset()`**, this behavior should not be relied on.

The **`TZ`** environment variable should contain no whitespace.

The standard format of the **`TZ`** environment variable is (whitespace added for clarity):

```
std offset [dst [offset [,start[/time], end[/time]]]]
```

Where the components are:

`std` and `dst`

Three or more alphanumerics giving the timezone abbreviations. These will be propagated into time.tzname

`offset`

The offset has the form: `± hh[:mm[:ss]]`. This indicates the value added the local time to arrive at UTC. If preceded by a ‘-’, the timezone is east of the Prime Meridian; otherwise, it is west. If no offset follows dst, summer time is assumed to be one hour ahead of standard time.

`start[/time], end[/time]`

Indicates when to change to and back from DST. The format of the start and end dates are one of the following:

J*n*

The Julian day *n* (1 <= *n* <= 365). Leap days are not counted, so in all years February 28 is day 59 and March 1 is day 60.

n

The zero-based Julian day (0 <= *n* <= 365). Leap days are counted, and it is possible to refer to February 29.

M*m*.*n*.*d*

The *d*’th day (0 <= *d* <= 6) of week *n* of month *m* of the year (1 <= *n* <= 5, 1 <= *m* <= 12, where week 5 means “the last *d* day in month *m*” which may occur in either the fourth or the fifth week). Week 1 is the first week in which the *d*’th day occurs. Day zero is a Sunday.

`time` has the same format as `offset` except that no leading sign (‘-’ or ‘+’) is allowed. The default, if time is not given, is 02:00:00.

```
>>> os.environ['TZ'] = 'EST+05EDT,M4.1.0,M10.5.0'
>>> time.tzset()
>>> time.strftime('%X %x %Z')
'02:07:36 05/08/03 EDT'
>>> os.environ['TZ'] = 'AEST-10AEDT-11,M10.5.0,M3.5.0'
>>> time.tzset()
>>> time.strftime('%X %x %Z')
'16:08:12 05/08/03 AEST'
```

On many Unix systems (including *BSD, Linux, Solaris, and Darwin), it is more convenient to use the system’s zoneinfo (*tzfile(5)* *[https://manpages.debian.org/tzfile(5)]*) database to specify the timezone rules. To do this, set the **TZ** environment variable to the path of the required timezone datafile, relative to the root of the systems ‘zoneinfo’ timezone database, usually located at `/usr/share/zoneinfo`. For example, `'US/Eastern'`, `'Australia/Melbourne'`, `'Egypt'` or `'Europe/Amsterdam'`.

```
>>> os.environ['TZ'] = 'US/Eastern'
>>> time.tzset()
>>> time.tzname
('EST', 'EDT')
>>> os.environ['TZ'] = 'Egypt'
>>> time.tzset()
>>> time.tzname
('EET', 'EEST')
```

Clock ID Constants

These constants are used as parameters for **`clock_getres()`** and **`clock_gettime()`**.

time.CLOCK_BOOTTIME

Identical to `CLOCK_MONOTONIC`, except it also includes any time that the system is suspended.

This allows applications to get a suspend-aware monotonic clock without having to deal with the complications of `CLOCK_REALTIME`, which may have discontinuities if the time is changed using `settimeofday()` or similar.

Availability: Linux >= 2.6.39.

Added in version 3.7.

time.CLOCK_HIGHRES

The Solaris OS has a `CLOCK_HIGHRES` timer that attempts to use an optimal hardware source, and may give close to nanosecond resolution. `CLOCK_HIGHRES` is the nonadjustable, high-resolution clock.

Availability: Solaris.

Added in version 3.3.

time.CLOCK_MONOTONIC

Clock that cannot be set and represents monotonic time since some unspecified starting point.

Availability: Unix.

Added in version 3.3.

time.CLOCK_MONOTONIC_RAW

Similar to `CLOCK_MONOTONIC`, but provides access to a raw hardware-based time that is not subject to NTP adjustments.

Availability: Linux >= 2.6.28, macOS >= 10.12.

Added in version 3.3.

time.CLOCK_MONOTONIC_RAW_APPROX

Similar to `CLOCK_MONOTONIC_RAW`, but reads a value cached by the system at context switch and hence has less accuracy.

Availability: macOS >= 10.12.

Added in version 3.13.

time.CLOCK_PROCESS_CPUTIME_ID

High-resolution per-process timer from the CPU.

Availability: Unix.

Added in version 3.3.

time.CLOCK_PROF

High-resolution per-process timer from the CPU.

Availability: FreeBSD, NetBSD >= 7, OpenBSD.

Added in version 3.7.

time.CLOCK_TAI

International Atomic Time [https://www.nist.gov/pml/time-and-frequency-division/how-utcnist-related-coordinated-universal-time-utc-international]

The system must have a current leap second table in order for this to give the correct answer. PTP or NTP software can maintain a leap second table.

Availability: Linux.

Added in version 3.9.

time.CLOCK_THREAD_CPUTIME_ID

Thread-specific CPU-time clock.

Availability: Unix.

Added in version 3.3.

time.CLOCK_UPTIME

Time whose absolute value is the time the system has been running and not suspended, providing accurate uptime measurement, both absolute and interval.

Availability: FreeBSD, OpenBSD >= 5.5.

Added in version 3.7.

time.CLOCK_UPTIME_RAW

Clock that increments monotonically, tracking the time since an arbitrary point, unaffected by frequency or time adjustments and not incremented while the system is asleep.

Availability: macOS >= 10.12.

Added in version 3.8.

time.CLOCK_UPTIME_RAW_APPROX

Like `CLOCK_UPTIME_RAW`, but the value is cached by the system at context switches and therefore has less accuracy.

Availability: macOS >= 10.12.

Added in version 3.13.

The following constant is the only parameter that can be sent to `clock_settime()`.

time.CLOCK_REALTIME

Real-time clock. Setting this clock requires appropriate privileges. The clock is the same for all processes.

Availability: Unix.

Added in version 3.3.

Timezone Constants

time.altzone

The offset of the local DST timezone, in seconds west of UTC, if one is defined. This is negative if the local DST timezone is east of UTC (as in Western Europe, including the UK). Only use this if `daylight` is nonzero. See note below.

time.daylight

Nonzero if a DST timezone is defined. See note below.

time.timezone

The offset of the local (non-DST) timezone, in seconds west of UTC (negative in most of Western Europe, positive in the US, zero in the UK). See note below.

time.tzname

A tuple of two strings: the first is the name of the local non-DST timezone, the second is the name of the local DST timezone. If no DST timezone is defined, the second string should not be used. See note below.

Note

For the above Timezone constants (**`altzone`**, **`daylight`**, **`timezone`**, and **`tzname`**), the value is determined by the timezone rules in effect at module load time or the last time `tzset()` is called and may be incorrect for times in the past. It is recommended to use the `tm_gmtoff` and `tm_zone` results from `localtime()` to obtain timezone information.

See also

Module `datetime`
More object-oriented interface to dates and times.

Module `locale`
Internationalization services. The locale setting affects the interpretation of many format specifiers in `strftime()` and `strptime()`.

Module `calendar`
General calendar-related functions. `timegm()` is the inverse of `gmtime()` from this module.

Footnotes

[1] (1,2,3)

The use of `%Z` is now deprecated, but the `%z` escape that expands to the preferred hour/minute offset is not supported by all ANSI C libraries. Also, a strict reading of the original 1982 **RFC 822** [https://datatracker.ietf.org/doc/html/rfc822.html] standard calls for a two-digit year (`%y` rather than `%Y`), but practice moved to 4-digit years long before the year 2000. After that, **RFC 822** [https://datatracker.ietf.org/doc/html/rfc822.html] became obsolete and the 4-digit year has been first recommended by **RFC 1123** [https://datatracker.ietf.org/doc/html/rfc1123.html] and then mandated by **RFC 2822** [https://datatracker.ietf.org/doc/html/rfc2822.html], with

RFC 5322 [https://datatracker.ietf.org/doc/html/rfc5322.html] continuing this requirement.

`logging` — Logging facility for Python

Source code: Lib/logging/__init__.py
[https://github.com/python/cpython/tree/3.14/Lib/logging/__init__.py]

This module defines functions and classes which implement a flexible event logging system for applications and libraries.

The key benefit of having the logging API provided by a standard library module is that all Python modules can participate in logging, so your application log can include your own messages integrated with messages from third-party modules.

Important

This page contains the API reference information. For tutorial information and discussion of more advanced topics, see

- Basic Tutorial
- Advanced Tutorial
- Logging Cookbook

Here’s a simple example of idiomatic usage:

```
# myapp.py
import logging
import mylib
logger = logging.getLogger(__name__)

def main():
    logging.basicConfig(filename='myapp.log',
level=logging.INFO)
    logger.info('Started')
    mylib.do_something()
    logger.info('Finished')
```

```
if __name__ == '__main__':
    main()
```

```
# mylib.py
import logging
logger = logging.getLogger(__name__)

def do_something():
    logger.info('Doing something')
```

If you run *myapp.py*, you should see this in *myapp.log*:

```
INFO:__main__:Started
INFO:mylib:Doing something
INFO:__main__:Finished
```

The key feature of this idiomatic usage is that the majority of code is simply creating a module level logger with `getLogger(__name__)`, and using that logger to do any needed logging. This is concise, while allowing downstream code fine-grained control if needed. Logged messages to the module-level logger get forwarded to handlers of loggers in higher-level modules, all the way up to the highest-level logger known as the root logger; this approach is known as hierarchical logging.

For logging to be useful, it needs to be configured: setting the levels and destinations for each logger, potentially changing how specific modules log, often based on command-line arguments or application configuration. In most cases, like the one above, only the root logger needs to be so configured, since all the lower level loggers at module level eventually forward their messages to its handlers. **`basicConfig()`** provides a quick way to configure the root logger that handles many use cases.

The module provides a lot of functionality and flexibility. If you are unfamiliar with logging, the best way to get to grips with it is to view the tutorials (**see the links above and on the right**).

The basic classes defined by the module, together with their attributes and methods, are listed in the sections below.

- Loggers expose the interface that application code directly uses.
- Handlers send the log records (created by loggers) to the appropriate destination.
- Filters provide a finer grained facility for determining which log records to output.
- Formatters specify the layout of log records in the final output.

Logger Objects

Loggers have the following attributes and methods. Note that Loggers should *NEVER* be instantiated directly, but always through the module-level function `logging.getLogger(name)`. Multiple calls to `getLogger()` with the same name will always return a reference to the same Logger object.

The `name` is potentially a period-separated hierarchical value, like `foo.bar.baz` (though it could also be just plain `foo`, for example). Loggers that are further down in the hierarchical list are children of loggers higher up in the list. For example, given a logger with a name of `foo`, loggers with names of `foo.bar`, `foo.bar.baz`, and `foo.bam` are all descendants of `foo`. In addition, all loggers are descendants of the root logger. The logger name hierarchy is analogous to the Python package hierarchy, and identical to it if you organise your loggers on a per-module basis using the recommended construction `logging.getLogger(__name__)`. That’s because in a module, `__name__` is the module’s name in the Python package namespace.

class logging.Logger

name

This is the logger’s name, and is the value that was passed to `getLogger()` to obtain the logger.

Note

This attribute should be treated as read-only.

level

The threshold of this logger, as set by the `setLevel()` method.

Note

Do not set this attribute directly - always use `setLevel()`, which has checks for the level passed to it.

parent

The parent logger of this logger. It may change based on later instantiation of loggers which are higher up in the namespace hierarchy.

Note

This value should be treated as read-only.

propagate

If this attribute evaluates to true, events logged to this logger will be passed to the handlers of higher level (ancestor) loggers, in addition to any handlers attached to this logger. Messages are passed directly to the ancestor loggers’ handlers - neither the level nor filters of the ancestor loggers in question are considered.

If this evaluates to false, logging messages are not passed to the handlers of ancestor loggers.

Spelling it out with an example: If the propagate attribute of the logger named `A.B.C` evaluates to true, any event logged to `A.B.C` via a method call such as `logging.getLogger('A.B.C').error(...)` will [subject to passing that logger’s level and filter settings] be passed in turn to any handlers attached to loggers named `A.B`, `A` and the root logger, after first being passed to any handlers attached to `A.B.C`. If any logger in the chain `A.B.C`, `A.B`, `A` has its `propagate` attribute set to false, then that is the last logger whose handlers are offered the event to handle, and propagation stops at that point.

The constructor sets this attribute to `True`.

Note

If you attach a handler to a logger *and* one or more of its ancestors, it may emit the same record multiple times. In general, you should not need to attach a handler to more than one logger - if you just attach it to the appropriate logger which is highest in the logger hierarchy, then it will see all events logged by all descendant loggers, provided that their propagate setting is left set to `True`. A common scenario is to attach handlers only to the root logger, and to let propagation take care of the rest.

handlers

The list of handlers directly attached to this logger instance.

Note

This attribute should be treated as read-only; it is normally changed via the `addHandler()` and `removeHandler()` methods, which use locks to ensure thread-safe operation.

disabled

This attribute disables handling of any events. It is set to `False` in the initializer, and only changed by logging configuration code.

Note

This attribute should be treated as read-only.

setLevel(*level*)

Sets the threshold for this logger to *level*. Logging messages which are less severe than *level* will be ignored; logging messages which have severity *level* or higher will be emitted by whichever handler or handlers service this logger, unless a handler’s level has been set to a higher severity level than *level*.

When a logger is created, the level is set to `NOTSET` (which causes all messages to be processed when the logger is the root logger, or delegation to the parent when the logger is a non-root logger). Note that the root logger is created with level `WARNING`.

The term ‘delegation to the parent’ means that if a logger has a level of NOTSET, its chain of ancestor loggers is traversed until either an ancestor with a level other than NOTSET is found, or the root is reached.

If an ancestor is found with a level other than NOTSET, then that ancestor’s level is treated as the effective level of the logger where the ancestor search began, and is used to determine how a logging event is handled.

If the root is reached, and it has a level of NOTSET, then all messages will be processed. Otherwise, the root's level will be used as the effective level.

See Logging Levels for a list of levels.

Changed in version 3.2: The *level* parameter now accepts a string representation of the level such as 'INFO' as an alternative to the integer constants such as **`INFO`**. Note, however, that levels are internally stored as integers, and methods such as e.g. **`getEffectiveLevel()`** and **`isEnabledFor()`** will return/expect to be passed integers.

isEnabledFor(*level*)

Indicates if a message of severity *level* would be processed by this logger. This method checks first the module-level level set by `logging.disable(level)` and then the logger's effective level as determined by **`getEffectiveLevel()`**.

getEffectiveLevel()

Indicates the effective level for this logger. If a value other than **`NOTSET`** has been set using **`setLevel()`**, it is returned. Otherwise, the hierarchy is traversed towards the root until a value other than **`NOTSET`** is found, and that value is returned. The value returned is an integer, typically one of **`logging.DEBUG`**, **`logging.INFO`** etc.

getChild(*suffix*)

Returns a logger which is a descendant to this logger, as determined by the suffix. Thus, `logging.getLogger('abc').getChild('def.ghi')` would return the same logger as would be returned by `logging.getLogger('abc.def.ghi')`. This is a convenience method, useful when the parent logger is named using e.g. `__name__` rather than a literal string.

Added in version 3.2.

getChildren()

Returns a set of loggers which are immediate children of this logger. So for example `logging.getLogger().getChildren()` might return a set containing loggers named `foo` and `bar`, but a logger named `foo.bar` wouldn't be included in the set. Likewise, `logging.getLogger('foo').getChildren()` might return a set including a logger named `foo.bar`, but it wouldn't include one named `foo.bar.baz`.

Added in version 3.12.

debug(*msg*, **args*, ***kwargs*)

Logs a message with level **`DEBUG`** on this logger. The *msg* is the message format string, and the *args* are the arguments which are merged into *msg* using the string formatting operator. (Note that this means that you can use keywords in the format string, together with a single dictionary argument.) No % formatting operation is performed on *msg* when no *args* are supplied.

There are four keyword arguments in *kwargs* which are inspected: *exc_info*, *stack_info*, *stacklevel* and *extra*.

If *exc_info* does not evaluate as false, it causes exception information to be added to the logging message. If an exception tuple (in the format returned by **`sys.exc_info()`**) or an exception instance is provided, it is used; otherwise, **`sys.exc_info()`** is called to get the exception information.

The second optional keyword argument is *stack_info*, which defaults to `False`. If true, stack information is added to the logging message, including the actual logging call. Note that this is not the same stack information as that displayed through specifying *exc_info*: The former is stack frames from the bottom of the stack up to the logging call in the current thread, whereas the latter is information about stack frames which have been unwound, following an exception, while searching for exception handlers.

You can specify *stack_info* independently of *exc_info*, e.g. to just show how you got to a certain point in your code, even when no exceptions were raised. The stack frames are printed following a header line which says:

```
Stack (most recent call last):
```

This mimics the `Traceback (most recent call last):` which is used when displaying exception frames.

The third optional keyword argument is *stacklevel*, which defaults to `1`. If greater than 1, the corresponding number of stack frames are skipped when computing the line number and function name set in the `LogRecord` created for the logging event. This can be used in logging helpers so that the function name, filename and line number recorded are not the information for the helper function/method, but rather its caller. The name of this parameter mirrors the equivalent one in the `warnings` module.

The fourth keyword argument is *extra* which can be used to pass a dictionary which is used to populate the `__dict__` of the `LogRecord` created for the logging event with user-defined attributes. These custom attributes can then be used as you like. For example, they could be incorporated into logged messages. For example:

```
FORMAT = '%(asctime)s %(clientip)-15s %(user)-8s %
(message)s'
logging.basicConfig(format=FORMAT)
d = {'clientip': '192.168.0.1', 'user': 'fbloggs'}
logger = logging.getLogger('tcpserver')
logger.warning('Protocol problem: %s', 'connection reset',
extra=d)
```

would print something like

```
2006-02-08 22:20:02,165 192.168.0.1 fbloggs  Protocol
problem: connection reset
```

The keys in the dictionary passed in *extra* should not clash with the keys used by the logging system. (See the section on LogRecord attributes for more information on which keys are used by the logging system.)

If you choose to use these attributes in logged messages, you need to exercise some care. In the above example, for instance, the **`Formatter`** has been set up with a format string which expects ‘clientip’ and ‘user’ in the attribute dictionary of the **`LogRecord`**. If these are missing, the message will not be logged because a string formatting exception will occur. So in this case, you always need to pass the *extra* dictionary with these keys.

While this might be annoying, this feature is intended for use in specialized circumstances, such as multi-threaded servers where the same code executes in many contexts, and interesting conditions which arise are dependent on this context (such as remote client IP address and authenticated user name, in the above example). In such circumstances, it is likely that specialized `Formatter`s would be used with particular `Handler`s.

If no handler is attached to this logger (or any of its ancestors, taking into account the relevant `Logger.propagate` attributes), the message will be sent to the handler set on `lastResort`.

Changed in version 3.2: The *stack_info* parameter was added.

Changed in version 3.5: The *exc_info* parameter can now accept exception instances.

Changed in version 3.8: The *stacklevel* parameter was added.

info(*msg*, **args*, ***kwargs*)

Logs a message with level `INFO` on this logger. The arguments are interpreted as for `debug()`.

warning(*msg*, **args*, ***kwargs*)

Logs a message with level `WARNING` on this logger. The arguments are interpreted as for `debug()`.

Note

There is an obsolete method `warn` which is functionally identical to `warning`. As `warn` is deprecated, please do not use it - use `warning` instead.

error(*msg*, **args*, ***kwargs*)

Logs a message with level `ERROR` on this logger. The arguments are interpreted as for `debug()`.

critical(*msg*, **args*, ***kwargs*)

Logs a message with level `CRITICAL` on this logger. The arguments are interpreted as for `debug()`.

log(*level*, *msg*, **args*, ***kwargs*)

Logs a message with integer level *level* on this logger. The other arguments are interpreted as for `debug()`.

exception(*msg*, **args*, ***kwargs*)

Logs a message with level `ERROR` on this logger. The arguments are interpreted as for `debug()`. Exception info is added to the logging message. This method should only be called from an exception handler.

addFilter(*filter*)

Adds the specified filter *filter* to this logger.

removeFilter(*filter*)

Removes the specified filter *filter* from this logger.

filter(*record*)

Apply this logger's filters to the record and return `True` if the record is to be processed. The filters are consulted in turn, until one of them returns a false value. If none of them return a false value, the record will be processed (passed to handlers). If one returns a false value, no further processing of the record occurs.

addHandler(*hdlr*)

Adds the specified handler *hdlr* to this logger.

removeHandler(*hdlr*)

Removes the specified handler *hdlr* from this logger.

findCaller(*stack_info=False*, *stacklevel=1*)

Finds the caller's source filename and line number. Returns the filename, line number, function name and stack information as a 4-element tuple. The stack information is returned as `None` unless *stack_info* is `True`.

The *stacklevel* parameter is passed from code calling the **`debug()`** and other APIs. If greater than 1, the excess is used to skip stack frames before determining the values to be returned. This will generally be useful when calling logging APIs from helper/wrapper code, so that the information in the event log refers not to the helper/wrapper code, but to the code that calls it.

handle(*record*)

Handles a record by passing it to all handlers associated with this logger and its ancestors (until a false value of *propagate* is found). This method is used for unpickled records received from a socket, as well as those created locally. Logger-level filtering is applied using **`filter()`**.

makeRecord(*name*, *level*, *fn*, *lno*, *msg*, *args*, *exc_info*, *func=None*, *extra=None*, *sinfo=None*)

This is a factory method which can be overridden in subclasses to create specialized **`LogRecord`** instances.

hasHandlers()

Checks to see if this logger has any handlers configured. This is done by looking for handlers in this logger and its parents in the logger hierarchy. Returns `True` if a handler was found, else `False`. The method stops searching up the hierarchy whenever a logger with the ‘propagate’ attribute set to false is found - that will be the last logger which is checked for the existence of handlers.

Added in version 3.2.

Changed in version 3.7: Loggers can now be pickled and unpickled.

Logging Levels

The numeric values of logging levels are given in the following table. These are primarily of interest if you want to define your own levels, and need them to have specific values relative to the predefined levels. If you define a level with the same numeric value, it overwrites the predefined value; the predefined name is lost.

Level	Numeric value	What it means / When to use it
logging.NOTSET	0	When set on a logger, indicates that ancestor loggers are to be consulted to determine the effective level. If that still resolves to `NOTSET`, then all events are logged. When set on a handler, all events are handled.
logging.DEBUG	10	Detailed information, typically only of interest to a developer trying to diagnose a problem.
logging.INFO	20	Confirmation that things are working as expected.

Level	Numeric value	What it means / When to use it
logging.WARNING	30	An indication that something unexpected happened, or that a problem might occur in the near future (e.g. ‘disk space low’). The software is still working as expected.
logging.ERROR	40	Due to a more serious problem, the software has not been able to perform some function.
logging.CRITICAL	50	A serious error, indicating that the program itself may be unable to continue running.

Handler Objects

Handlers have the following attributes and methods. Note that `Handler` is never instantiated directly; this class acts as a base for more useful subclasses. However, the `__init__()` method in subclasses needs to call `Handler.__init__()`.

class logging.Handler

__init__(*level=NOTSET*)

Initializes the `Handler` instance by setting its level, setting the list of filters to the empty list and creating a lock (using `createLock()`) for serializing access to an I/O mechanism.

createLock()

Initializes a thread lock which can be used to serialize access to underlying I/O functionality which may not be threadsafe.

acquire()

Acquires the thread lock created with `createLock()`.

release()

Releases the thread lock acquired with `acquire()`.

setLevel(*level*)

Sets the threshold for this handler to *level*. Logging messages which are less severe than *level* will be ignored. When a handler is created, the level is set to `NOTSET` (which causes all messages to be processed).

See Logging Levels for a list of levels.

Changed in version 3.2: The *level* parameter now accepts a string representation of the level such as ‘INFO’ as an alternative to the integer constants such as `INFO`.

setFormatter(*fmt*)

Sets the formatter for this handler to *fmt*. The *fmt* argument must be a `Formatter` instance or `None`.

addFilter(*filter*)

Adds the specified filter *filter* to this handler.

removeFilter(*filter*)

Removes the specified filter *filter* from this handler.

filter(*record*)

Apply this handler’s filters to the record and return `True` if the record is to be processed. The filters are consulted in turn, until one of them returns a false value. If none of them return a false value, the record will be emitted. If one returns a false value, the handler will not emit the record.

flush()

Ensure all logging output has been flushed. This version does nothing and is intended to be implemented by subclasses.

close()

Tidy up any resources used by the handler. This version does no output but removes the handler from an internal map of handlers, which is used for handler lookup by name.

Subclasses should ensure that this gets called from overridden **`close()`** methods.

handle(*record*)

Conditionally emits the specified logging record, depending on filters which may have been added to the handler. Wraps the actual emission of the record with acquisition/release of the I/O thread lock.

handleError(*record*)

This method should be called from handlers when an exception is encountered during an **`emit()`** call. If the module-level attribute **`raiseExceptions`** is `False`, exceptions get silently ignored. This is what is mostly wanted for a logging system - most users will not care about errors in the logging system, they are more interested in application errors. You could, however, replace this with a custom handler if you wish. The specified record is the one which was being processed when the exception occurred. (The default value of `raiseExceptions` is `True`, as that is more useful during development).

format(*record*)

Do formatting for a record - if a formatter is set, use it. Otherwise, use the default formatter for the module.

emit(*record*)

Do whatever it takes to actually log the specified logging record. This version is intended to be implemented by subclasses and so raises a `NotImplementedError`.

Warning

This method is called after a handler-level lock is acquired, which is released after this method returns. When you override this method, note that you should be careful when calling anything that invokes other parts of the logging API which might do locking, because that might result in a deadlock. Specifically:

- Logging configuration APIs acquire the module-level lock, and then individual handler-level locks as those handlers are configured.
- Many logging APIs lock the module-level lock. If such an API is called from this method, it could cause a deadlock if a configuration call is made on another thread, because that thread will try to acquire the module-level lock *before* the handler-level lock, whereas this thread tries to acquire the module-level lock *after* the handler-level lock

(because in this method, the handler-level lock has already been acquired).

For a list of handlers included as standard, see **`logging.handlers`**.

Formatter Objects

class logging.Formatter(*fmt=None*, *datefmt=None*, *style='%'*, *validate=True*, ***, *defaults=None*)

Responsible for converting a **`LogRecord`** to an output string to be interpreted by a human or external system.

Parameters::

- **fmt** (*str*) – A format string in the given *style* for the logged output as a whole. The possible mapping keys are drawn from the **`LogRecord`** object's LogRecord attributes. If not specified, `'%(message)s'` is used, which is just the logged message.
- **datefmt** (*str*) – A format string for the date/time portion of the logged output. not specified, the default described in **`formatTime()`** is used.
- **style** (*str*) – Can be one of `'%'`, `'{'` or `'$'` and determines how the format string will be merged with its data: using one of printf-style String Formatting (`%`), **`str.format()`** (`{`) or **`string.Template`** (`$`). Thi only applies to *fmt* (e.g. `'%(message)s'` versus `'{message}'`), not to the actual log messages passed to the logging methods. However, there are other way to use `{`- and `$`-formatting for log messages.
- **validate** (*bool*) – If `True` (the default), incorrect or mismatched *fmt* and *style* will raise a **`ValueError`**; for example, `logging.Formatter('%(asctime)s - %(message)s', style='{')`.
- **defaults** (*dict[str, Any]*) – A dictionary with default values to use in custom fields. For example, `logging.Formatter('%(ip)s %(message)s', defaults={"ip": None})`

Changed in version 3.2: Added the *style* parameter.

Changed in version 3.8: Added the *validate* parameter.

Changed in version 3.10: Added the *defaults* parameter.

format(*record*)

The record’s attribute dictionary is used as the operand to a string formatting operation. Returns the resulting string. Before formatting the dictionary, a couple of preparatory steps are carried out. The *message* attribute of the record is computed using *msg* % *args*. If the formatting string contains `'(asctime)'`, **`formatTime()`** is called to format the event time. If there is exception information, it is formatted using **`formatException()`** and appended to the message. Note that the formatted exception information is cached in attribute *exc_text*. This is useful because the exception information can be pickled and sent across the wire, but you should be careful if you have more than one **`Formatter`** subclass which customizes the formatting of exception information. In this case, you will have to clear the cached value (by setting the *exc_text* attribute to `None`) after a formatter has done its formatting, so that the next formatter to handle the event doesn’t use the cached value, but recalculates it afresh.

If stack information is available, it’s appended after the exception information, using **`formatStack()`** to transform it if necessary.

formatTime(*record*, *datefmt=None*)

This method should be called from **`format()`** by a formatter which wants to make use of a formatted time. This method can be overridden in formatters to provide for any specific requirement, but the basic behavior is as follows: if *datefmt* (a string) is specified, it is used with **`time.strftime()`** to format the creation time of the record. Otherwise, the format ‘%Y-%m-%d %H:%M:%S,uuu’ is used, where the uuu part is a millisecond value and the other letters are as per the **`time.strftime()`** documentation. An example time in this format is `2003-01-23 00:29:50,411`. The resulting string is returned.

This function uses a user-configurable function to convert the creation time to a tuple. By default, **`time.localtime()`** is used; to change this for a particular formatter instance, set the `converter` attribute to a function with the same signature as **`time.localtime()`** or **`time.gmtime()`**. To change it for all formatters, for example if you want all logging times to be shown in GMT, set the `converter` attribute in the `Formatter` class.

Changed in version 3.3: Previously, the default format was hard-coded as in this example: `2010-09-06 22:38:15,292` where the part before the comma is handled by a strptime format string (`'%Y-%m-%d %H:%M:%S'`), and the part after the comma is a millisecond value. Because strptime does not have a format placeholder for milliseconds, the millisecond value is appended using another format string, `'%s,%03d'` — and both of these format strings have been hardcoded into this method. With the change, these strings are defined as class-level attributes which can be overridden at the instance level when desired. The names of the attributes are `default_time_format` (for the strptime format string) and `default_msec_format` (for appending the millisecond value).

Changed in version 3.9: The `default_msec_format` can be `None`.

formatException(*exc_info*)

Formats the specified exception information (a standard exception tuple as returned by **`sys.exc_info()`**) as a string. This default implementation just uses **`traceback.print_exception()`**. The resulting string is returned.

formatStack(*stack_info*)

Formats the specified stack information (a string as returned by `traceback.print_stack()`, but with the last newline removed) as a string. This default implementation just returns the input value.

class logging.BufferingFormatter(*linefmt=None*)

A base formatter class suitable for subclassing when you want to format a number of records. You can pass a `Formatter` instance which you want to use to format each line (that corresponds to a single record). If not specified, the default formatter (which just outputs the event message) is used as the line formatter.

formatHeader(*records*)

Return a header for a list of *records*. The base implementation just returns the empty string. You will need to override this method if you want specific behaviour, e.g. to show the count of records, a title or a separator line.

formatFooter(*records*)

Return a footer for a list of *records*. The base implementation just returns the empty string. You will need to override this method if you want specific behaviour, e.g. to show the count of records or a separator line.

format(*records*)

Return formatted text for a list of *records*. The base implementation just returns the empty string if there are no records; otherwise, it returns the concatenation of the header, each record formatted with the line formatter, and the footer.

Filter Objects

`Filters` can be used by `Handlers` and `Loggers` for more sophisticated filtering than is provided by levels. The base filter class only allows events which are below a certain point in the logger hierarchy. For example, a filter initialized with 'A.B' will allow events logged by loggers 'A.B', 'A.B.C', 'A.B.C.D', 'A.B.D' etc. but not 'A.BB', 'B.A.B' etc. If initialized with the empty string, all events are passed.

class logging.Filter(*name*='')

Returns an instance of the **`Filter`** class. If *name* is specified, it names a logger which, together with its children, will have its events allowed through the filter. If *name* is the empty string, allows every event.

filter(*record*)

Is the specified record to be logged? Returns false for no, true for yes. Filters can either modify log records in-place or return a completely different record instance which will replace the original log record in any future processing of the event.

Note that filters attached to handlers are consulted before an event is emitted by the handler, whereas filters attached to loggers are consulted whenever an event is logged (using **`debug()`**, **`info()`**, etc.), before sending an event to handlers. This means that events which have been generated by descendant loggers will not be filtered by a logger's filter setting, unless the filter has also been applied to those descendant loggers.

You don't actually need to subclass `Filter`: you can pass any instance which has a `filter` method with the same semantics.

Changed in version 3.2: You don’t need to create specialized `Filter` classes, or use other classes with a `filter` method: you can use a function (or other callable) as a filter. The filtering logic will check to see if the filter object has a `filter` attribute: if it does, it’s assumed to be a `Filter` and its **`filter()`** method is called. Otherwise, it’s assumed to be a callable and called with the record as the single parameter. The returned value should conform to that returned by **`filter()`**.

Changed in version 3.12: You can now return a **`LogRecord`** instance from filters to replace the log record rather than modifying it in place. This allows filters attached to a **`Handler`** to modify the log record before it is emitted, without having side effects on other handlers.

Although filters are used primarily to filter records based on more sophisticated criteria than levels, they get to see every record which is processed by the handler or logger they’re attached to: this can be useful if you want to do things like counting how many records were processed by a particular logger or handler, or adding, changing or removing attributes in the **`LogRecord`** being processed. Obviously changing the LogRecord needs to be done with some care, but it does allow the injection of contextual information into logs (see Using Filters to impart contextual information).

LogRecord Objects

`LogRecord` instances are created automatically by the **`Logger`** every time something is logged, and can be created manually via **`makeLogRecord()`** (for example, from a pickled event received over the wire).

class logging.LogRecord(*name*, *level*, *pathname*, *lineno*, *msg*, *args*, *exc_info*, *func=None*, *sinfo=None*)

Contains all the information pertinent to the event being logged.

The primary information is passed in *msg* and *args*, which are combined using `msg % args` to create the **`message`** attribute of the record.

Parameters::

- **name** (*str*) – The name of the logger used to log the event represented by this **`LogRecord`**. Note that the logger nam in the **`LogRecord`** will always have this value, even though it may be emitted b a handler attached to a different (ancestor) logger.
- **level** (*int*) – The numeric level of the logging event (such as `10` for `DEBUG`, `20` fo `INFO`, etc). Note that this is converted to *two* attributes of the LogRecord: **`levelno`** for the numeric value and **`levelname`** for the corresponding level name.
- **pathname** (*str*) – The full string path of the source file where the logging call was made.
- **lineno** (*int*) – The line number in the source file where the logging call was made.
- **msg** (*Any*) – The event description message, which can be a %-format string with placeholders for variable data, or an arbitrary object (see Using arbitrary objects as messages).
- **args** (*tuple* | *dict[str, Any]*) – Variable data to merge into the *msg* argument t obtain the event description.
- **exc_info** (*tuple[type[BaseException], BaseException, types.TracebackType]* | *None*) – An exception tuple with the current exception information, as returned by **`sys.exc_info()`**, or `None` if no exception information is available.
- **func** (*str* | *None*) – The name of the function or method from which the logging call was invoked.
- **sinfo** (*str* | *None*) – A text string

representing stack information from the base of the stack in the current thread, up to the logging call.

getMessage()

Returns the message for this **LogRecord** instance after merging any user-supplied arguments with the message. If the user-supplied message argument to the logging call is not a string, **str()** is called on it to convert it to a string. This allows use of user-defined classes as messages, whose `__str__` method can return the actual format string to be used.

Changed in version 3.2: The creation of a **LogRecord** has been made more configurable by providing a factory which is used to create the record. The factory can be set using **getLogRecordFactory()** and **setLogRecordFactory()** (see this for the factory's signature).

This functionality can be used to inject your own values into a **LogRecord** at creation time. You can use the following pattern:

```
old_factory = logging.getLogRecordFactory()

def record_factory(*args, **kwargs):
    record = old_factory(*args, **kwargs)
    record.custom_attribute = 0xdecafbad
    return record

logging.setLogRecordFactory(record_factory)
```

With this pattern, multiple factories could be chained, and as long as they don't overwrite each other's attributes or unintentionally overwrite the standard attributes listed above, there should be no surprises.

LogRecord attributes

The LogRecord has a number of attributes, most of which are derived from the parameters to the constructor. (Note that the names do not always correspond exactly between the

LogRecord constructor parameters and the LogRecord attributes.) These attributes can be used to merge data from the record into the format string. The following table lists (in alphabetical order) the attribute names, their meanings and the corresponding placeholder in a %-style format string.

If you are using {}-formatting (`str.format()`), you can use `{attrname}` as the placeholder in the format string. If you are using $-formatting (`string.Template`), use the form `${attrname}`. In both cases, of course, replace `attrname` with the actual attribute name you want to use.

In the case of {}-formatting, you can specify formatting flags by placing them after the attribute name, separated from it with a colon. For example: a placeholder of `{msecs:03.0f}` would format a millisecond value of `4` as `004`. Refer to the `str.format()` documentation for full details on the options available to you.

Attribute name	Format	Description
args	You shouldn’t need to format this yourself.	The tuple of arguments merged into `msg` to produce `message`, or a dict whose values are used for the merge (when there is only one argument, and it is a dictionary).
asctime	`%(asctime)s`	Human-readable time when the `LogRecord` was created. By default this

Attribute name	Format	Description
		is of the form ‘2003-07-08 16:49:45,896’ (the numbers after the comma are millisecond portion of the time).
created	`%(created)f`	Time when the `LogRecord` was created (as returned by `time.time_ns()` / 1e9).
exc_info	You shouldn’t need to format this yourself.	Exception tuple (à la `sys.exc_info`) or, if no exception has occurred, `None`.
exc_text	You shouldn’t need to format this yourself.	Exception information formatted as a string. This is set when `Formatter.format()` is invoked, or `None` if no exception has occurred.
filename	`%(filename)s`	Filename portion of `pathname`.
funcName	`%(funcName)s`	Name of function containing the logging call.

Attribute name	Format	Description
levelname	`%(levelname)s`	Text logging level for the message (`'DEBUG'`, `'INFO'`, `'WARNING'`, `'ERROR'`, `'CRITICAL'`).
levelno	`%(levelno)s`	Numeric logging level for the message (**`DEBUG`**, **`INFO`**, **`WARNING`**, **`ERROR`**, **`CRITICAL`**).
lineno	`%(lineno)d`	Source line number where the logging call was issued (if available).
message	`%(message)s`	The logged message, computed as `msg % args`. This is set when **`Formatter.format()`** is invoked.
module	`%(module)s`	Module (name portion of `filename`).
msecs	`%(msecs)d`	Millisecond portion of the time when the **`LogRecord`** was created.

Attribute name	Format	Description
msg	You shouldn’t need to format this yourself.	The format string passed in the original logging call. Merged with `args` to produce `message`, or an arbitrary object (see Using arbitrary objects as messages).
name	`%(name)s`	Name of the logger used to log the call.
pathname	`%(pathname)s`	Full pathname of the source file where the logging call was issued (if available).
process	`%(process)d`	Process ID (if available).
processName	`%(processName)s`	Process name (if available).
relativeCreated	`%(relativeCreated)d`	Time in milliseconds when the LogRecord was created, relative to the time the logging module was loaded.

Attribute name	Format	Description
stack_info	You shouldn’t need to format this yourself.	Stack frame information (where available) from the bottom of the stack in the current thread, up to and including the stack frame of the logging call which resulted in the creation of this record.
thread	`%(thread)d`	Thread ID (if available).
threadName	`%(threadName)s`	Thread name (if available).
taskName	`%(taskName)s`	**`asyncio.Task`** name (if available).

Changed in version 3.1: processName was added.

Changed in version 3.12: taskName was added.

LoggerAdapter Objects

`LoggerAdapter` instances are used to conveniently pass contextual information into logging calls. For a usage example, see the section on adding contextual information to your logging output.

class logging.LoggerAdapter(*logger*, *extra=None*, *merge_extra=False*)

Returns an instance of `LoggerAdapter` initialized with an underlying `Logger` instance, an optional dict-like object (*extra*), and an optional boolean (*merge_extra*) indicating whether or not the *extra* argument of individual log calls should be merged with the `LoggerAdapter` extra. The default behavior is to ignore the *extra* argument of individual log calls and only use the one of the `LoggerAdapter` instance

process(*msg*, *kwargs*)

Modifies the message and/or keyword arguments passed to a logging call in order to insert contextual information. This implementation takes the object passed as *extra* to the constructor and adds it to *kwargs* using key 'extra'. The return value is a (*msg*, *kwargs*) tuple which has the (possibly modified) versions of the arguments passed in.

manager

Delegates to the underlying **`manager`** on *logger*.

_log

Delegates to the underlying **`_log()`** method on *logger*.

In addition to the above, `LoggerAdapter` supports the following methods of `Logger`: `debug()`, `info()`, `warning()`, `error()`, `exception()`, `critical()`, `log()`, `isEnabledFor()`, `getEffectiveLevel()`, `setLevel()` and `hasHandlers()`. These methods have the same signatures as their counterparts in `Logger`, so you can use the two types of instances interchangeably.

Changed in version 3.2: The `isEnabledFor()`, `getEffectiveLevel()`, `setLevel()` and `hasHandlers()` methods

were added to `LoggerAdapter`. These methods delegate to the underlying logger.
Changed in version 3.6: Attribute **`manager`** and method **`_log()`** were added, which delegate to the underlying logger and allow adapters to be nested.

Changed in version 3.10: The *extra* argument is now optional.

Changed in version 3.13: The *merge_extra* parameter was added.

Thread Safety

The logging module is intended to be thread-safe without any special work needing to be done by its clients. It achieves this through using threading locks; there is one lock to serialize access to the module's shared data, and each handler also creates a lock to serialize access to its underlying I/O.

If you are implementing asynchronous signal handlers using the `signal` module, you may not be able to use logging from within such handlers. This is because lock implementations in the `threading` module are not always re-entrant, and so cannot be invoked from such signal handlers.

Module-Level Functions

In addition to the classes described above, there are a number of module-level functions.

logging.getLogger(*name=None*)

Return a logger with the specified name or, if name is `None`, return the root logger of the hierarchy. If specified, the name is typically a dot-separated hierarchical name like *'a'*, *'a.b'* or *'a.b.c.d'*. Choice of these names is entirely up to the developer who is using logging, though it is recommended that `__name__` be used unless you have a specific reason for not doing that, as mentioned in Logger Objects.

All calls to this function with a given name return the same logger instance. This means that logger instances never need to be passed between different parts of an application.

logging.getLoggerClass()

Return either the standard **`Logger`** class, or the last class passed to **`setLoggerClass()`**. This function may be called from within a new class definition, to ensure that installing a customized **`Logger`** class will not undo customizations already applied by other code. For example:

```
class MyLogger(logging.getLoggerClass()):
    # ... override behaviour here
```

logging.getLogRecordFactory()

Return a callable which is used to create a **`LogRecord`**.

Added in version 3.2: This function has been provided, along with **`setLogRecordFactory()`**, to allow developers more control over how the **`LogRecord`** representing a logging event is constructed.

See **`setLogRecordFactory()`** for more information about the how the factory is called.

logging.debug(*msg*, **args*, ***kwargs*)

This is a convenience function that calls **`Logger.debug()`**, on the root logger. The handling of the arguments is in every way identical to what is described in that method.

The only difference is that if the root logger has no handlers, then `basicConfig()` is called, prior to calling `debug` on the root logger.

For very short scripts or quick demonstrations of `logging` facilities, `debug` and the other module-level functions may be convenient. However, most programs will want to carefully and explicitly control the logging configuration, and should therefore prefer creating a module-level logger and calling `Logger.debug()` (or other level-specific methods) on it, as described at the beginning of this documentation.

logging.info(*msg*, **args*, ***kwargs*)

Logs a message with level `INFO` on the root logger. The arguments and behavior are otherwise the same as for `debug()`.

logging.warning(*msg*, **args*, ***kwargs*)

Logs a message with level `WARNING` on the root logger. The arguments and behavior are otherwise the same as for `debug()`.

Note

There is an obsolete function `warn` which is functionally identical to `warning`. As `warn` is deprecated, please do not use it - use `warning` instead.

logging.error(*msg*, **args*, ***kwargs*)

Logs a message with level `ERROR` on the root logger. The arguments and behavior are otherwise the same as for `debug()`.

logging.critical(*msg*, **args*, ***kwargs*)

Logs a message with level `CRITICAL` on the root logger. The arguments and behavior are otherwise the same as for `debug()`.

logging.exception(*msg*, **args*, ***kwargs*)

Logs a message with level **`ERROR`** on the root logger. The arguments and behavior are otherwise the same as for **`debug()`**. Exception info is added to the logging message. This function should only be called from an exception handler.

logging.log(*level*, *msg*, **args*, ***kwargs*)

Logs a message with level *level* on the root logger. The arguments and behavior are otherwise the same as for **`debug()`**.

logging.disable(*level=CRITICAL*)

Provides an overriding level *level* for all loggers which takes precedence over the logger's own level. When the need arises to temporarily throttle logging output down across the whole application, this function can be useful. Its effect is to disable all logging calls of severity *level* and below, so that if you call it with a value of INFO, then all INFO and DEBUG events would be discarded, whereas those of severity WARNING and above would be processed according to the logger's effective level. If `logging.disable(logging.NOTSET)` is called, it effectively removes this overriding level, so that logging output again depends on the effective levels of individual loggers.

Note that if you have defined any custom logging level higher than `CRITICAL` (this is not recommended), you won't be able to rely on the default value for the *level* parameter, but will have to explicitly supply a suitable value.

Changed in version 3.7: The *level* parameter was defaulted to level `CRITICAL`. See bpo-28524 [https://bugs.python.org/issue?@action=redirect&bpo=28524] for more information about this change.

logging.addLevelName(*level*, *levelName*)

Associates level *level* with text *levelName* in an internal dictionary, which is used to map numeric levels to a textual representation, for example when a `Formatter` formats a message. This function can also be used to define your own levels. The only constraints are that all levels used must be registered using this function, levels should be positive integers and they should increase in increasing order of severity.

Note

If you are thinking of defining your own levels, please see the section on Custom Levels.

logging.getLevelNamesMapping()

Returns a mapping from level names to their corresponding logging levels. For example, the string “CRITICAL” maps to `CRITICAL`. The returned mapping is copied from an internal mapping on each call to this function.

Added in version 3.11.

logging.getLevelName(*level*)

Returns the textual or numeric representation of logging level *level*.

If *level* is one of the predefined levels `CRITICAL`, `ERROR`, `WARNING`, `INFO` or `DEBUG` then you get the corresponding string. If you have associated levels with names using `addLevelName()` then the name you have associated with *level* is returned. If a numeric value corresponding to one of the defined levels is passed in, the corresponding string representation is returned.

The *level* parameter also accepts a string representation of the level such as ‘INFO’. In such cases, this functions returns the corresponding numeric value of the level.

If no matching numeric or string value is passed in, the string ‘Level %s’ % level is returned.

Note

Levels are internally integers (as they need to be compared in the logging logic). This function is used to convert between an integer level and the level name displayed in the formatted log output by means of the `%(levelname)s` format specifier (see LogRecord attributes), and vice versa.

Changed in version 3.4: In Python versions earlier than 3.4, this function could also be passed a text level, and would return the corresponding numeric value of the level. This undocumented behaviour was considered a mistake, and was removed in Python 3.4, but reinstated in 3.4.2 due to retain backward compatibility.

logging.getHandlerByName(*name*)

Returns a handler with the specified *name*, or `None` if there is no handler with that name.

Added in version 3.12.

logging.getHandlerNames()

Returns an immutable set of all known handler names.

Added in version 3.12.

logging.makeLogRecord(*attrdict*)

Creates and returns a new `LogRecord` instance whose attributes are defined by *attrdict*. This function is useful for taking a pickled `LogRecord` attribute dictionary, sent over a socket, and reconstituting it as a `LogRecord` instance at the receiving end.

logging.basicConfig(***kwargs*)

Does basic configuration for the logging system by creating a `StreamHandler` with a default `Formatter` and adding it to the root logger. The functions `debug()`, `info()`, `warning()`, `error()` and `critical()` will call `basicConfig()` automatically if no handlers are defined for the root logger.

This function does nothing if the root logger already has handlers configured, unless the keyword argument *force* is set to `True`.

Note

This function should be called from the main thread before other threads are started. In versions of Python prior to 2.7.1 and 3.2, if this function is called from multiple threads, it is possible (in rare circumstances) that a handler will be added to the root logger more than once, leading to unexpected results such as messages being duplicated in the log.

The following keyword arguments are supported.

Format	Description
filename	Specifies that a `FileHandler` be created, using the specified filename, rather than a `StreamHandler`.

Format	**Description**
filemode	If *filename* is specified, open the file in this mode. Defaults to `'a'`.
format	Use the specified format string for the handler. Defaults to attributes `levelname`, `name` and `message` separated by colons.
datefmt	Use the specified date/time format, as accepted by **`time.strftime()`**.
style	If *format* is specified, use this style for the format string. One of `'%'`, `'{'` or `'$'` for printf-style, **`str.format()`** or **`string.Template`** respectively. Defaults to `'%'`.
level	Set the root logger level to the specified level.
stream	Use the specified stream to initialize the **`StreamHandler`**. Note that this argument is incompatible with *filename* - if both are present, a `ValueError` is raised.
handlers	If specified, this should be an iterable of already created handlers to add to the root logger. Any handlers which don’t already have a formatter set will be assigned the default formatter created in this function. Note that this argument is incompatible with *filename* or *stream* - if both are present, a `ValueError` is raised.
force	If this keyword argument is specified as true, any existing handlers attached to the root logger are removed and closed, before

Format	Description
	carrying out the configuration as specified by the other arguments.
encoding	If this keyword argument is specified along with *filename*, its value is used when the `FileHandler` is created, and thus used when opening the output file.
errors	If this keyword argument is specified along with *filename*, its value is used when the `FileHandler` is created, and thus used when opening the output file. If not specified, the value ‘backslashreplace’ is used. Note that if `None` is specified, it will be passed as such to `open()`, which means that it will be treated the same as passing ‘errors’.

Changed in version 3.2: The *style* argument was added.

Changed in version 3.3: The *handlers* argument was added. Additional checks were added to catch situations where incompatible arguments are specified (e.g. *handlers* together with *stream* or *filename*, or *stream* together with *filename*).

Changed in version 3.8: The *force* argument was added.

Changed in version 3.9: The *encoding* and *errors* arguments were added.

logging.shutdown()

Informs the logging system to perform an orderly shutdown by flushing and closing all handlers. This should be called at application exit and no further use of the logging system should be made after this call.

When the logging module is imported, it registers this function as an exit handler (see `atexit`), so normally there's no need to do that manually.

logging.setLoggerClass(*klass*)

Tells the logging system to use the class *klass* when instantiating a logger. The class should define `__init__()` such that only a name argument is required, and the `__init__()` should call `Logger.__init__()`. This function is typically called before any loggers are instantiated by applications which need to use custom logger behavior. After this call, as at any other time, do not instantiate loggers directly using the subclass: continue to use the `logging.getLogger()` API to get your loggers.

logging.setLogRecordFactory(*factory*)

Set a callable which is used to create a `LogRecord`.

Parameters:: **factory** – The factory callable to be used to instantiate a log record.

Added in version 3.2: This function has been provided, along with `getLogRecordFactory()`, to allow developers more control over how the `LogRecord` representing a logging event is constructed.

The factory has the following signature:

```
factory(name, level, fn, lno, msg, args, exc_info, func=None,
sinfo=None, **kwargs)
```

name::	The logger name.
level::	The logging level (numeric).
fn::	

	The full pathname of the file where the logging call was made.
lno::	The line number in the file where the logging call was made.
msg::	The logging message.
args::	The arguments for the logging message.
exc_info::	An exception tuple, or `None`.
func::	The name of the function or method which invoked the logging call.
sinfo::	A stack traceback such as is provided by [`traceback.print_stack()`], showing the call hierarchy.
kwargs::	Additional keyword arguments.

Module-Level Attributes

logging.lastResort

A “handler of last resort” is available through this attribute. This is a [`StreamHandler`] writing to `sys.stderr` with a level of `WARNING`, and is used to handle logging events in the absence of any logging configuration. The end result is to just print the message to `sys.stderr`. This replaces the earlier error message saying that “no handlers could be found for logger XYZ”. If you need the earlier behaviour for some reason, `lastResort` can be set to `None`.

Added in version 3.2.

logging.raiseExceptions

Used to see if exceptions during handling should be propagated.

Default: `True`.

If `raiseExceptions` is `False`, exceptions get silently ignored. This is what is mostly wanted for a logging system - most users will not care about errors in the logging system, they are more interested in application errors.

Integration with the warnings module

The `captureWarnings()` function can be used to integrate `logging` with the `warnings` module.

logging.captureWarnings(*capture*)

This function is used to turn the capture of warnings by logging on and off.

If *capture* is `True`, warnings issued by the `warnings` module will be redirected to the logging system. Specifically, a warning will be formatted using `warnings.formatwarning()` and the resulting string logged to a logger named `'py.warnings'` with a severity of `WARNING`.

If *capture* is `False`, the redirection of warnings to the logging system will stop, and warnings will be redirected to their original destinations (i.e. those in effect before `captureWarnings(True)` was called).

See also

Module `logging.config`

Configuration API for the logging module.

Module `logging.handlers`

Useful handlers included with the logging module.

PEP 282 [https://peps.python.org/pep-0282/] - A Logging System

The proposal which described this feature for inclusion in the Python standard library.

Original Python logging package [https://old.red-dove.com/python_logging.html]

This is the original source for the `logging` package. The version of the package available from this site is suitable for use with Python 1.5.2, 2.1.x and 2.2.x, which do not include the `logging` package in the standard library.

`logging.config` — Logging configuration

Source code: Lib/logging/config.py
[https://github.com/python/cpython/tree/3.14/Lib/logging/config.py]

This section describes the API for configuring the logging module.

Important

This page contains only reference information. For tutorials, please see

- Basic Tutorial
- Advanced Tutorial
- Logging Cookbook

Configuration functions

The following functions configure the logging module. They are located in the `logging.config` module. Their use is optional — you can configure the logging module using these functions or by making calls to the main API (defined in `logging` itself) and defining handlers which are declared either in `logging` or `logging.handlers`.

logging.config.dictConfig(*config*)

> Takes the logging configuration from a dictionary. The contents of this dictionary are described in Configuration dictionary schema below.
>
> If an error is encountered during configuration, this function will raise a `ValueError`, `TypeError`, `AttributeError` or `ImportError` with a suitably descriptive message. The following is a (possibly incomplete) list of conditions which will raise an error:

- A `level` which is not a string or which is a string not corresponding to an actual logging level.
- A `propagate` value which is not a boolean.
- An id which does not have a corresponding destination.
- A non-existent handler id found during an incremental call.
- An invalid logger name.
- Inability to resolve to an internal or external object.

Parsing is performed by the **`DictConfigurator`** class, whose constructor is passed the dictionary used for configuration, and has a **`configure()`** method. The [`logging.config`] module has a callable attribute **`dictConfigClass`** which is initially set to **`DictConfigurator`**. You can replace the value of **`dictConfigClass`** with a suitable implementation of your own.

[`dictConfig()`] calls **`dictConfigClass`** passing the specified dictionary, and then calls the **`configure()`** method on the returned object to put the configuration into effect:

```
def dictConfig(config):
    dictConfigClass(config).configure()
```

For example, a subclass of **`DictConfigurator`** could call `DictConfigurator.__init__()` in its own **`__init__()`**, then set up custom prefixes which would be usable in the subsequent **`configure()`** call. **`dictConfigClass`** would be bound to this new subclass, and then [`dictConfig()`] could be called exactly as in the default, uncustomized state.

Added in version 3.2.

logging.config.fileConfig(*fname*, *defaults=None*, *disable_existing_loggers=True*, *encoding=None*)

Reads the logging configuration from a [`configparser`]-format file. The format of the file should be as described in Configuration file format. This function can be called several times from an application, allowing an end user to select from various pre-canned configurations (if the developer provides a mechanism to present the choices and load the chosen configuration).

It will raise `FileNotFoundError` if the file doesn’t exist and `RuntimeError` if the file is invalid or empty.

Parameters::

- **fname** – A filename, or a file-like object or an instance derived from `RawConfigParser`. If a **`RawConfigParser`**-derived instance is passed, it is used as is. Otherwise, a `ConfigParser` is instantiated, and the configuration read by it from the object passed in `fname`. If that has a `readline()` method, it is assumed to be a file-like object and read using `read_file()`; otherwise, it is assumed to be a filename and passed to `read()`.
- **defaults** – Defaults to be passed to the **`ConfigParser`** can be specified in this argument.
- **disable_existing_loggers** – If specified as `False`, loggers which exist when this call is made are left enabled. The defaul is `True` because this enables old behaviour in a backward-compatible way. This behaviour is to disable any existing non-root loggers unless they or their ancestors are explicitly named in the logging configuration.
- **encoding** – The encoding used to open file when *fname* is filename.

Changed in version 3.4: An instance of a subclass of `RawConfigParser` is now accepted as a value for `fname`. This facilitates:

- Use of a configuration file where logging configuration is just part of the overall application configuration.
- Use of a configuration read from a file, and then modified by the using application (e.g. based on command-line parameters or other aspects of the runtime environment) before being passed to `fileConfig`.

Changed in version 3.10: Added the *encoding* parameter.

Changed in version 3.12: An exception will be thrown if the provided file doesn’t exist or is invalid or empty.

logging.config.listen(*port=DEFAULT_LOGGING_CONFIG_PORT*, *verify=None*)

Starts up a socket server on the specified port, and listens for new configurations. If no port is specified, the module’s default **`DEFAULT_LOGGING_CONFIG_PORT`** is used. Logging configurations will be sent as a file suitable for processing by **`dictConfig()`** or **`fileConfig()`**. Returns a **`Thread`** instance on which you can call **`start()`** to start the server, and which you can **`join()`** when appropriate. To stop the server, call **`stopListening()`**.

The `verify` argument, if specified, should be a callable which should verify whether bytes received across the socket are valid and should be processed. This could be done by encrypting and/or signing what is sent across the socket, such that the `verify` callable can perform signature verification and/or decryption. The `verify` callable is called with a single argument - the bytes received across the socket - and should return the bytes to be processed, or `None` to indicate that the bytes should be discarded. The returned bytes could be the same as the passed in bytes (e.g. when only verification is done), or they could be completely different (perhaps if decryption were performed).

To send a configuration to the socket, read in the configuration file and send it to the socket as a sequence of bytes preceded by a four-byte length string packed in binary using `struct.pack('>L', n)`.

Note

Because portions of the configuration are passed through `eval()`, use of this function may open its users to a security risk. While the function only binds to a socket on `localhost`, and so does not accept connections from remote machines, there are scenarios where untrusted code could be run under the account of the process which calls `listen()`. Specifically, if the process calling `listen()` runs on a multi-user machine where users cannot trust each other, then a malicious user could arrange to run essentially arbitrary code in a victim user's process, simply by connecting to the victim's `listen()` socket and sending a configuration which runs whatever code the attacker wants to have executed in the victim's process. This is especially easy to do if the default port is used, but not hard even if a different port is used. To avoid the risk of this happening, use the `verify` argument to `listen()` to prevent unrecognised configurations from being applied.

Changed in version 3.4: The `verify` argument was added.

Note

If you want to send configurations to the listener which don't disable existing loggers, you will need to use a JSON format for the configuration, which will use `dictConfig()` for configuration. This method allows you to specify `disable_existing_loggers` as `False` in the configuration you send.

logging.config.stopListening()

Stops the listening server which was created with a call to `listen()`. This is typically called before calling **`join()`** on the return value from `listen()`.

Security considerations

The logging configuration functionality tries to offer convenience, and in part this is done by offering the ability to convert text in configuration files into Python objects used in logging configuration - for example, as described in User-defined objects. However, these same mechanisms (importing callables from user-defined modules and calling them with parameters from the configuration) could be used to invoke any code you like, and for this reason you should treat configuration files from untrusted sources with *extreme caution* and satisfy yourself that nothing bad can happen if you load them, before actually loading them.

Configuration dictionary schema

Describing a logging configuration requires listing the various objects to create and the connections between them; for example, you may create a handler named ‘console’ and then say that the logger named ‘startup’ will send its messages to the ‘console’ handler. These objects aren’t limited to those provided by the `logging` module because you might write your own formatter or handler class. The parameters to these classes may also need to include external objects such as `sys.stderr`. The syntax for describing these objects and connections is defined in Object connections below.

Dictionary Schema Details

The dictionary passed to **`dictConfig()`** must contain the following keys:

- *version* - to be set to an integer value representing the schema version. The only valid value at present is 1, but having this key allows the schema to evolve while still preserving backwards compatibility.

All other keys are optional, but if present they will be interpreted as described below. In all cases below where a 'configuring dict' is mentioned, it will be checked for the special `'()'` key to see if a custom instantiation is required. If so, the mechanism described in User-defined objects below is used to create an instance; otherwise, the context is used to determine what to instantiate.

- *formatters* - the corresponding value will be a dict in which each key is a formatter id and each value is a dict describing how to configure the corresponding **`Formatter`** instance.

  The configuring dict is searched for the following optional keys which correspond to the arguments passed to create a **`Formatter`** object:

    - `format`
    - `datefmt`
    - `style`
    - `validate` (since version >=3.8)
    - `defaults` (since version >=3.12)

An optional `class` key indicates the name of the formatter's class (as a dotted module and class name). The instantiation arguments are as for **`Formatter`**, thus this key is most useful for instantiating a customised subclass of **`Formatter`**. For example, the alternative class might present exception tracebacks in an expanded or condensed format. If your formatter requires different or extra configuration keys, you should use User-defined objects.

- *filters* - the corresponding value will be a dict in which each key is a filter id and each value is a dict describing how to configure the corresponding Filter instance.

  The configuring dict is searched for the key `name` (defaulting to the empty string) and this is used to construct a **`logging.Filter`** instance.

- *handlers* - the corresponding value will be a dict in which each key is a handler id and each value is a dict describing how to configure the corresponding Handler instance.

  The configuring dict is searched for the following keys:

  - `class` (mandatory). This is the fully qualified name of the handler class.
  - `level` (optional). The level of the handler.
  - `formatter` (optional). The id of the formatter for this handler.
  - `filters` (optional). A list of ids of the filters for this handler.

    Changed in version 3.11: `filters` can take filter instances in addition to ids.

All *other* keys are passed through as keyword arguments to the handler’s constructor. For example, given the snippet:

```
handlers:
  console:
    class : logging.StreamHandler
    formatter: brief
    level   : INFO
    filters: [allow_foo]
    stream  : ext://sys.stdout
  file:
    class : logging.handlers.RotatingFileHandler
    formatter: precise
    filename: logconfig.log
    maxBytes: 1024
    backupCount: 3
```

the handler with id `console` is instantiated as a **`logging.StreamHandler`**, using `sys.stdout` as the underlying stream. The handler with id `file` is instantiated as a **`logging.handlers.RotatingFileHandler`** with the keyword arguments `filename='logconfig.log', maxBytes=1024, backupCount=3`.

- *loggers* - the corresponding value will be a dict in which each key is a logger name and each value is a dict describing how to configure the corresponding Logger instance.

  The configuring dict is searched for the following keys:

  - `level` (optional). The level of the logger.
  - `propagate` (optional). The propagation setting of the logger.
  - `filters` (optional). A list of ids of the filters for this logger.

    Changed in version 3.11: `filters` can take filter instances in addition to ids.

  - `handlers` (optional). A list of ids of the handlers for this logger.

  The specified loggers will be configured according to the level, propagation, filters and handlers specified.

- *root* - this will be the configuration for the root logger. Processing of the configuration will be as for any logger, except that the `propagate` setting will not be applicable.

- *incremental* - whether the configuration is to be interpreted as incremental to the existing configuration. This value defaults to `False`, which means that the specified configuration replaces the existing configuration with the same semantics as used by the existing **`fileConfig()`** API.

  If the specified value is `True`, the configuration is processed as described in the section on Incremental Configuration.

- *disable_existing_loggers* - whether any existing non-root loggers are to be disabled. This setting mirrors the parameter of the same name in **`fileConfig()`**. If absent, this parameter defaults to `True`. This value is ignored if *incremental* is `True`.

Incremental Configuration

It is difficult to provide complete flexibility for incremental configuration. For example, because objects such as filters and formatters are anonymous, once a configuration is set up, it is not possible to refer to such anonymous objects when augmenting a configuration.

Furthermore, there is not a compelling case for arbitrarily altering the object graph of loggers, handlers, filters, formatters at run-time, once a configuration is set up; the verbosity of loggers and handlers can be controlled just by setting levels (and, in the case of loggers, propagation flags). Changing the object graph arbitrarily in a safe way is problematic in a multi-threaded environment; while not impossible, the benefits are not worth the complexity it adds to the implementation.

Thus, when the `incremental` key of a configuration dict is present and is `True`, the system will completely ignore any `formatters` and `filters` entries, and process only the `level` settings in the `handlers` entries, and the `level` and `propagate` settings in the `loggers` and `root` entries.

Using a value in the configuration dict lets configurations to be sent over the wire as pickled dicts to a socket listener. Thus, the logging verbosity of a long-running application can be altered over time with no need to stop and restart the application.

Object connections

The schema describes a set of logging objects - loggers, handlers, formatters, filters - which are connected to each other in an object graph. Thus, the schema needs to represent connections between the objects. For example, say that, once configured, a particular logger has attached to it a particular handler. For the purposes of this discussion, we can say that the logger represents the source, and the handler the destination, of a connection between the two. Of course in the configured objects this is represented by the logger holding a reference to the handler. In the configuration dict, this is done by giving each destination object an id which identifies it unambiguously, and then using the id in the source object’s configuration to indicate that a connection exists between the source and the destination object with that id.

So, for example, consider the following YAML snippet:

```
formatters:
  brief:
    # configuration for formatter with id 'brief' goes here
  precise:
    # configuration for formatter with id 'precise' goes here
handlers:
  h1: #This is an id
   # configuration of handler with id 'h1' goes here
   formatter: brief
  h2: #This is another id
   # configuration of handler with id 'h2' goes here
   formatter: precise
loggers:
  foo.bar.baz:
    # other configuration for logger 'foo.bar.baz'
    handlers: [h1, h2]
```

(Note: YAML used here because it’s a little more readable than the equivalent Python source form for the dictionary.)

The ids for loggers are the logger names which would be used programmatically to obtain a reference to those loggers, e.g. `foo.bar.baz`. The ids for Formatters and Filters can be any string value (such as `brief`, `precise` above) and they are transient, in that they are only meaningful for processing the configuration dictionary and used to determine connections between objects, and are not persisted anywhere when the configuration call is complete.

The above snippet indicates that logger named `foo.bar.baz` should have two handlers attached to it, which are described by the handler ids `h1` and `h2`. The formatter for `h1` is that described by id `brief`, and the formatter for `h2` is that described by id `precise`.

User-defined objects

The schema supports user-defined objects for handlers, filters and formatters. (Loggers do not need to have different types for different instances, so there is no support in this configuration schema for user-defined logger classes.)

Objects to be configured are described by dictionaries which detail their configuration. In some places, the logging system will be able to infer from the context how an object is to be instantiated, but when a user-defined object is to be instantiated, the system will not know how to do this. In order to provide complete flexibility for user-defined object instantiation, the user needs to provide a ‘factory’ - a callable which is called with a configuration dictionary and which returns the instantiated object. This is signalled by an absolute import path to the factory being made available under the special key `'()'`. Here’s a concrete example:

```
formatters:
  brief:
    format: '%(message)s'
  default:
    format: '%(asctime)s %(levelname)-8s %(name)-15s %
(message)s'
    datefmt: '%Y-%m-%d %H:%M:%S'
  custom:
      (): my.package.customFormatterFactory
      bar: baz
      spam: 99.9
      answer: 42
```

The above YAML snippet defines three formatters. The first, with id `brief`, is a standard `logging.Formatter` instance with the specified format string. The second, with id `default`, has a longer format and also defines the time format explicitly, and will result in a `logging.Formatter` initialized with those two format strings. Shown in Python source form, the `brief` and `default` formatters have configuration sub-dictionaries:

```
{
  'format' : '%(message)s'
}
```

and:

```
{
  'format' : '%(asctime)s %(levelname)-8s %(name)-15s %
(message)s',
  'datefmt' : '%Y-%m-%d %H:%M:%S'
}
```

respectively, and as these dictionaries do not contain the special key `'()'`, the instantiation is inferred from the context: as a result, standard `logging.Formatter` instances are created. The configuration sub-dictionary for the third formatter, with id `custom`, is:

```
{
  '()' : 'my.package.customFormatterFactory',
  'bar' : 'baz',
  'spam' : 99.9,
  'answer' : 42
}
```

and this contains the special key `'()'`, which means that user-defined instantiation is wanted. In this case, the specified factory callable will be used. If it is an actual callable it will be used directly - otherwise, if you specify a string (as in the example) the actual callable will be located using normal import mechanisms. The callable will be called with the **remaining** items in the configuration sub-dictionary as keyword arguments. In the above example, the formatter with id `custom` will be assumed to be returned by the call:

```
my.package.customFormatterFactory(bar='baz', spam=99.9,
answer=42)
```

Warning

The values for keys such as `bar`, `spam` and `answer` in the above example should not be configuration dictionaries or references such as `cfg://foo` or `ext://bar`, because they will not be processed by the configuration machinery, but passed to the callable as-is.

The key `'()'` has been used as the special key because it is not a valid keyword parameter name, and so will not clash with the names of the keyword arguments used in the call. The `'()'` also serves as a mnemonic that the corresponding value is a callable.

Changed in version 3.11: The `filters` member of `handlers` and `loggers` can take filter instances in addition to ids.

You can also specify a special key `'.'` whose value is a mapping of attribute names to values. If found, the specified attributes will be set on the user-defined object before it is returned. Thus, with the following configuration:

```
{
  '()' : 'my.package.customFormatterFactory',
  'bar' : 'baz',
  'spam' : 99.9,
  'answer' : 42,
  '.' {
    'foo': 'bar',
    'baz': 'bozz'
  }
}
```

the returned formatter will have attribute `foo` set to `'bar'` and attribute `baz` set to `'bozz'`.

Warning

The values for attributes such as `foo` and `baz` in the above example should not be configuration dictionaries or references such as `cfg://foo` or `ext://bar`, because they will not be processed by the configuration machinery, but set as attribute values as-is.

Handler configuration order

Handlers are configured in alphabetical order of their keys, and a configured handler replaces the configuration dictionary in (a working copy of) the `handlers` dictionary in the schema. If you use a construct such as `cfg://handlers.foo`, then initially `handlers['foo']` points to the configuration dictionary for the handler named `foo`, and later (once that handler has been configured) it points to the configured handler instance. Thus, `cfg://handlers.foo` could resolve to either a dictionary or a handler instance. In general, it is wise to name handlers in a way such that dependent handlers are configured *after* any handlers they depend on; that allows something like `cfg://handlers.foo` to be used in configuring a handler that depends on handler `foo`. If that dependent handler were named `bar`, problems would result, because the configuration of `bar` would be attempted before that of `foo`, and `foo` would not yet have been configured. However, if the dependent handler were named `foobar`, it would be configured after `foo`, with the result that `cfg://handlers.foo` would resolve to configured handler `foo`, and not its configuration dictionary.

Access to external objects

There are times where a configuration needs to refer to objects external to the configuration, for example `sys.stderr`. If the configuration dict is constructed using Python code, this is straightforward, but a problem arises when the configuration is provided via a text file (e.g. JSON, YAML). In a text file, there is no standard way to distinguish `sys.stderr` from the literal string `'sys.stderr'`. To facilitate this distinction, the configuration system looks for certain special prefixes in string values and treat them specially. For example, if the literal string `'ext://sys.stderr'` is provided as a value in the configuration, then the `ext://` will be stripped off and the remainder of the value processed using normal import mechanisms.

The handling of such prefixes is done in a way analogous to protocol handling: there is a generic mechanism to look for prefixes which match the regular expression `^(?P<prefix>[a-z]+)://(?P<suffix>.*)$` whereby, if the `prefix` is recognised, the `suffix` is processed in a prefix-dependent manner and the result of the processing replaces the string value. If the prefix is not recognised, then the string value will be left as-is.

Access to internal objects

As well as external objects, there is sometimes also a need to refer to objects in the configuration. This will be done implicitly by the configuration system for things that it knows about. For example, the string value `'DEBUG'` for a `level` in a logger or handler will automatically be converted to the value `logging.DEBUG`, and the `handlers`, `filters` and `formatter` entries will take an object id and resolve to the appropriate destination object.

However, a more generic mechanism is needed for user-defined objects which are not known to the **`logging`** module. For example, consider **`logging.handlers.MemoryHandler`**, which takes a `target` argument which is another handler to delegate to. Since the system already knows about this class, then in the configuration, the given `target` just needs to be the object id of the relevant target handler, and the system will resolve to the handler from the id. If, however, a user defines a `my.package.MyHandler` which has an `alternate` handler, the configuration system would not know that the `alternate` referred to a handler. To cater for this, a generic resolution system allows the user to specify:

```
handlers:
  file:
    # configuration of file handler goes here

  custom:
    (): my.package.MyHandler
    alternate: cfg://handlers.file
```

The literal string `'cfg://handlers.file'` will be resolved in an analogous way to strings with the `ext://` prefix, but looking in the configuration itself rather than the import namespace. The mechanism allows access by dot or by index, in a similar way to that provided by `str.format`. Thus, given the following snippet:

```
handlers:
  email:
    class: logging.handlers.SMTPHandler
    mailhost: localhost
    fromaddr: my_app@domain.tld
    toaddrs:
      - support_team@domain.tld
      - dev_team@domain.tld
    subject: Houston, we have a problem.
```

in the configuration, the string `'cfg://handlers'` would resolve to the dict with key `handlers`, the string `'cfg://handlers.email` would resolve to the dict with key `email` in the `handlers` dict, and so on. The string `'cfg://handlers.email.toaddrs[1]` would resolve to `'dev_team@domain.tld'` and the string `'cfg://handlers.email.toaddrs[0]'` would resolve to the value `'support_team@domain.tld'`. The `subject` value could be accessed using either `'cfg://handlers.email.subject'` or, equivalently, `'cfg://handlers.email[subject]'`. The latter form only needs to be used if the key contains spaces or non-alphanumeric characters. Please note that the characters `[` and `]` are not allowed in the keys. If an index value consists only of decimal digits, access will be attempted using the corresponding integer value, falling back to the string value if needed.

Given a string `cfg://handlers.myhandler.mykey.123`, this will resolve to `config_dict['handlers']['myhandler']['mykey']['123']`. If the string is specified as `cfg://handlers.myhandler.mykey[123]`, the system will attempt to retrieve the value from `config_dict['handlers']['myhandler']['mykey'][123]`, and fall back to `config_dict['handlers']['myhandler']['mykey']['123']` if that fails.

Import resolution and custom importers

Import resolution, by default, uses the builtin `__import__()` function to do its importing. You may want to replace this with your own importing mechanism: if so, you can replace the **`importer`** attribute of the **`DictConfigurator`** or its superclass, the **`BaseConfigurator`** class. However, you need to be careful because of the way functions are accessed from classes via descriptors. If you are using a Python callable to do your imports, and you want to define it at class level rather than instance level, you need to wrap it with `staticmethod()`. For example:

```
from importlib import import_module
from logging.config import BaseConfigurator

BaseConfigurator.importer = staticmethod(import_module)
```

You don’t need to wrap with `staticmethod()` if you’re setting the import callable on a configurator *instance*.

Configuring QueueHandler and QueueListener

If you want to configure a `QueueHandler`, noting that this is normally used in conjunction with a `QueueListener`, you can configure both together. After the configuration, the

`QueueListener` instance will be available as the **`listener`** attribute of the created handler, and that in turn will be available to you using **`getHandlerByName()`** and passing the name you have used for the `QueueHandler` in your configuration. The dictionary schema for configuring the pair is shown in the example YAML snippet below.

```
handlers:
  qhand:
    class: logging.handlers.QueueHandler
    queue: my.module.queue_factory
    listener: my.package.CustomListener
    handlers:
      - hand_name_1
      - hand_name_2
      ...
```

The `queue` and `listener` keys are optional.

If the `queue` key is present, the corresponding value can be one of the following:

- An object implementing the **`Queue.put_nowait`** and **`Queue.get`** public API. For instance, this may be an actual instance of **`queue.Queue`** or a subclass thereof, or a proxy obtained by **`multiprocessing.managers.SyncManager.Queue()`**.

  This is of course only possible if you are constructing or modifying the configuration dictionary in code.

- A string that resolves to a callable which, when called with no arguments, returns the queue instance to use. That callable could be a **`queue.Queue`** subclass or a function which returns a suitable queue instance, such as `my.module.queue_factory()`.

- A dict with a `'()'` key which is constructed in the usual way as discussed in User-defined objects. The result of this construction should be a **`queue.Queue`** instance.

If the `queue` key is absent, a standard unbounded **`queue.Queue`** instance is created and used.

If the `listener` key is present, the corresponding value can be one of the following:

- A subclass of **`logging.handlers.QueueListener`**. This is of course only possible if you are constructing or modifying the configuration dictionary in code.
- A string which resolves to a class which is a subclass of `QueueListener`, such as `'my.package.CustomListener'`.
- A dict with a `'()'` key which is constructed in the usual way as discussed in User-defined objects. The result of this construction should be a callable with the same signature as the `QueueListener` initializer.

If the `listener` key is absent, **`logging.handlers.QueueListener`** is used.

The values under the `handlers` key are the names of other handlers in the configuration (not shown in the above snippet) which will be passed to the queue listener.

Any custom queue handler and listener classes will need to be defined with the same initialization signatures as **`QueueHandler`** and **`QueueListener`**.

Added in version 3.12.

Configuration file format

The configuration file format understood by **`fileConfig()`** is based on **`configparser`** functionality. The file must contain sections called `[loggers]`, `[handlers]` and `[formatters]` which identify by name the entities of each type which are defined in the file. For each such entity, there is a separate section which identifies how that entity is configured. Thus, for a logger named `log01` in the `[loggers]` section, the relevant configuration details are held in a section `[logger_log01]`. Similarly, a handler called `hand01` in the `[handlers]` section will have its configuration held in a section called `[handler_hand01]`, while a formatter called `form01` in the `[formatters]` section will have its configuration specified in a section called `[formatter_form01]`. The root logger configuration must be specified in a section called `[logger_root]`.

Note

The **`fileConfig()`** API is older than the **`dictConfig()`** API and does not provide functionality to cover certain aspects of logging. For example, you cannot configure **`Filter`** objects, which provide for filtering of messages beyond simple integer levels, using **`fileConfig()`**. If you need to have instances of **`Filter`** in your logging configuration, you will need to use **`dictConfig()`**. Note that future enhancements to configuration functionality will be added to **`dictConfig()`**, so it's worth considering transitioning to this newer API when it's convenient to do so.

Examples of these sections in the file are given below.

```
[loggers]
keys=root,log02,log03,log04,log05,log06,log07

[handlers]
keys=hand01,hand02,hand03,hand04,hand05,hand06,hand07,hand08,han
d09

[formatters]
keys=form01,form02,form03,form04,form05,form06,form07,form08,for
m09
```

The root logger must specify a level and a list of handlers. An example of a root logger section is given below.

```
[logger_root]
level=NOTSET
handlers=hand01
```

The `level` entry can be one of `DEBUG`, `INFO`, `WARNING`, `ERROR`, `CRITICAL` or `NOTSET`. For the root logger only, `NOTSET` means that all messages will be logged. Level values are evaluated in the context of the `logging` package's namespace.

The `handlers` entry is a comma-separated list of handler names, which must appear in the `[handlers]` section. These names must appear in the `[handlers]` section and have corresponding sections in the configuration file.

For loggers other than the root logger, some additional information is required. This is illustrated by the following example.

```
[logger_parser]
level=DEBUG
handlers=hand01
propagate=1
qualname=compiler.parser
```

The `level` and `handlers` entries are interpreted as for the root logger, except that if a non-root logger's level is specified as `NOTSET`, the system consults loggers higher up the hierarchy to determine the effective level of the logger. The `propagate` entry is set to 1 to indicate that messages must propagate to handlers higher up the logger hierarchy from this logger, or 0 to indicate that messages are **not** propagated to handlers up the hierarchy. The `qualname` entry is the hierarchical channel name of the logger, that is to say the name used by the application to get the logger.

Sections which specify handler configuration are exemplified by the following.

```
[handler_hand01]
class=StreamHandler
level=NOTSET
formatter=form01
args=(sys.stdout,)
```

The `class` entry indicates the handler’s class (as determined by `eval()` in the `logging` package’s namespace). The `level` is interpreted as for loggers, and `NOTSET` is taken to mean ‘log everything’.

The `formatter` entry indicates the key name of the formatter for this handler. If blank, a default formatter (`logging._defaultFormatter`) is used. If a name is specified, it must appear in the `[formatters]` section and have a corresponding section in the configuration file.

The `args` entry, when evaluated in the context of the `logging` package’s namespace, is the list of arguments to the constructor for the handler class. Refer to the constructors for the relevant handlers, or to the examples below, to see how typical entries are constructed. If not provided, it defaults to `()`.

The optional `kwargs` entry, when evaluated in the context of the `logging` package’s namespace, is the keyword argument dict to the constructor for the handler class. If not provided, it defaults to `{}`.

```
[handler_hand02]
class=FileHandler
level=DEBUG
formatter=form02
args=('python.log', 'w')

[handler_hand03]
class=handlers.SocketHandler
level=INFO
formatter=form03
args=('localhost', handlers.DEFAULT_TCP_LOGGING_PORT)
```

```
[handler_hand04]
class=handlers.DatagramHandler
level=WARN
formatter=form04
args=('localhost', handlers.DEFAULT_UDP_LOGGING_PORT)

[handler_hand05]
class=handlers.SysLogHandler
level=ERROR
formatter=form05
args=(('localhost', handlers.SYSLOG_UDP_PORT),
handlers.SysLogHandler.LOG_USER)

[handler_hand06]
class=handlers.NTEventLogHandler
level=CRITICAL
formatter=form06
args=('Python Application', '', 'Application')

[handler_hand07]
class=handlers.SMTPHandler
level=WARN
formatter=form07
args=('localhost', 'from@abc', ['user1@abc', 'user2@xyz'],
'Logger Subject')
kwargs={'timeout': 10.0}

[handler_hand08]
class=handlers.MemoryHandler
level=NOTSET
formatter=form08
target=
args=(10, ERROR)

[handler_hand09]
class=handlers.HTTPHandler
level=NOTSET
formatter=form09
args=('localhost:9022', '/log', 'GET')
kwargs={'secure': True}
```

Sections which specify formatter configuration are typified by the following.

```
[formatter_form01]
format=F1 %(asctime)s %(levelname)s %(message)s %(customfield)s
datefmt=
style=%
validate=True
defaults={'customfield': 'defaultvalue'}
class=logging.Formatter
```

The arguments for the formatter configuration are the same as the keys in the dictionary schema formatters section.

The `defaults` entry, when evaluated in the context of the `logging` package's namespace, is a dictionary of default values for custom formatting fields. If not provided, it defaults to `None`.

Note

Due to the use of `eval()` as described above, there are potential security risks which result from using the `listen()` to send and receive configurations via sockets. The risks are limited to where multiple users with no mutual trust run code on the same machine; see the `listen()` documentation for more information.

See also

Module `logging`
API reference for the logging module.

Module `logging.handlers`
Useful handlers included with the logging module.

`logging.handlers` — Logging handlers

Source code: Lib/logging/handlers.py
[https://github.com/python/cpython/tree/3.14/Lib/logging/handlers.py]

The following useful handlers are provided in the package. Note that three of the handlers (`StreamHandler`, `FileHandler` and `NullHandler`) are actually defined in the `logging` module itself, but have been documented here along with the other handlers.

Important

This page contains only reference information. For tutorials, please see

- Basic Tutorial
- Advanced Tutorial
- Logging Cookbook

StreamHandler

The `StreamHandler` class, located in the core `logging` package, sends logging output to streams such as *sys.stdout*, *sys.stderr* or any file-like object (or, more precisely, any object which supports `write()` and `flush()` methods).

class logging.StreamHandler(*stream=None*)

Returns a new instance of the `StreamHandler` class. If *stream* is specified, the instance will use it for logging output; otherwise, *sys.stderr* will be used.

emit(*record*)

If a formatter is specified, it is used to format the record. The record is then written to the stream followed by `terminator`. If exception information is present, it is formatted using `traceback.print_exception()` and appended to the stream.

flush()

Flushes the stream by calling its `flush()` method. Note that the **`close()`** method is inherited from `Handler` and so does no output, so an explicit `flush()` call may be needed at times.

setStream(*stream*)

Sets the instance’s stream to the specified value, if it is different. The old stream is flushed before the new stream is set.

Parameters::	**stream** – The stream that the handler should use.
Returns::	the old stream, if the stream was changed, or `None` if it wasn’t.

Added in version 3.7.

terminator

String used as the terminator when writing a formatted record to a stream. Default value is `'\n'`.

If you don’t want a newline termination, you can set the handler instance’s `terminator` attribute to the empty string.

In earlier versions, the terminator was hardcoded as `'\n'`.

Added in version 3.2.

FileHandler

The `FileHandler` class, located in the core `logging` package, sends logging output to a disk file. It inherits the output functionality from `StreamHandler`.

class logging.FileHandler(*filename*, *mode='a'*, *encoding=None*, *delay=False*, *errors=None*)

Returns a new instance of the `FileHandler` class. The specified file is opened and used as the stream for logging. If *mode* is not specified, `'a'` is used. If *encoding* is not `None`, it is used to open the file with that encoding. If *delay* is true, then file opening is deferred until the first call to `emit()`. By default, the file grows indefinitely. If *errors* is specified, it's used to determine how encoding errors are handled.

Changed in version 3.6: As well as string values, `Path` objects are also accepted for the *filename* argument.

Changed in version 3.9: The *errors* parameter was added.

close()

Closes the file.

emit(*record*)

Outputs the record to the file.

Note that if the file was closed due to logging shutdown at exit and the file mode is 'w', the record will not be emitted (see bpo-42378 [https://bugs.python.org/issue?@action=redirect&bpo=42378]).

NullHandler

Added in version 3.1.

The `NullHandler` class, located in the core `logging` package, does not do any formatting or output. It is essentially a ‘no-op’ handler for use by library developers.

class logging.NullHandler

Returns a new instance of the `NullHandler` class.

emit(*record*)

This method does nothing.

handle(*record*)

This method does nothing.

createLock()

This method returns `None` for the lock, since there is no underlying I/O to which access needs to be serialized.

See Configuring Logging for a Library for more information on how to use `NullHandler`.

WatchedFileHandler

The `WatchedFileHandler` class, located in the `logging.handlers` module, is a **`FileHandler`** which watches the file it is logging to. If the file changes, it is closed and reopened using the file name.

A file change can happen because of usage of programs such as *newsyslog* and *logrotate* which perform log file rotation. This handler, intended for use under Unix/Linux, watches the file to see if it has changed since the last emit. (A file is deemed to have changed if its device or inode have changed.) If the file has changed, the old file stream is closed, and the file opened to get a new stream.

This handler is not appropriate for use under Windows, because under Windows open log files cannot be moved or renamed - logging opens the files with exclusive locks - and so there is no need for such a handler. Furthermore, *ST_INO* is not supported under Windows; `stat()` always returns zero for this value.

class logging.handlers.WatchedFileHandler(*filename*, *mode='a'*, *encoding=None*, *delay=False*, *errors=None*)

Returns a new instance of the `WatchedFileHandler` class. The specified file is opened and used as the stream for logging. If *mode* is not specified, `'a'` is used. If *encoding* is not `None`, it is used to open the file with that encoding. If *delay* is true, then file opening is deferred until the first call to `emit()`. By default, the file grows indefinitely. If *errors* is provided, it determines how encoding errors are handled.

Changed in version 3.6: As well as string values, `Path` objects are also accepted for the *filename* argument.

Changed in version 3.9: The *errors* parameter was added.

reopenIfNeeded()

Checks to see if the file has changed. If it has, the existing stream is flushed and closed and the file opened again, typically as a precursor to outputting the record to the file.

Added in version 3.6.

emit(*record*)

Outputs the record to the file, but first calls `reopenIfNeeded()` to reopen the file if it has changed.

BaseRotatingHandler

The `BaseRotatingHandler` class, located in the `logging.handlers` module, is the base class for the rotating file handlers, `RotatingFileHandler` and `TimedRotatingFileHandler`. You should not need to instantiate this class, but it has attributes and methods you may need to override.

class logging.handlers.BaseRotatingHandler(*filename*, *mode*, *encoding=None*, *delay=False*, *errors=None*)

The parameters are as for **`FileHandler`**. The attributes are:

namer

If this attribute is set to a callable, the `rotation_filename()` method delegates to this callable. The parameters passed to the callable are those passed to `rotation_filename()`.

Note

The namer function is called quite a few times during rollover, so it should be as simple and as fast as possible. It should also return the same output every time for a given input, otherwise the rollover behaviour may not work as expected.

It’s also worth noting that care should be taken when using a namer to preserve certain attributes in the filename which are used during rotation. For example, `RotatingFileHandler` expects to have a set of log files whose names contain successive integers, so that rotation works as expected, and `TimedRotatingFileHandler` deletes old log files (based on the `backupCount` parameter passed to the handler’s initializer) by determining the oldest files to delete. For this to happen, the filenames should be sortable using the date/time portion of the filename, and a namer needs to respect this. (If a namer is wanted that doesn’t respect this scheme, it will need to be used in a subclass of `TimedRotatingFileHandler` which overrides the `getFilesToDelete()` method to fit in with the custom naming scheme.)

Added in version 3.3.

rotator

If this attribute is set to a callable, the `rotate()` method delegates to this callable. The parameters passed to the callable are those passed to `rotate()`.

Added in version 3.3.

rotation_filename(*default_name*)

Modify the filename of a log file when rotating.

This is provided so that a custom filename can be provided.

The default implementation calls the ‘namer’ attribute of the handler, if it’s callable, passing the default name to it. If the attribute isn’t callable (the default is `None`), the name is returned unchanged.

Parameters:: **default_name** – The default name for the log file.

Added in version 3.3.

rotate(*source*, *dest*)

When rotating, rotate the current log.

The default implementation calls the ‘rotator’ attribute of the handler, if it’s callable, passing the source and dest arguments to it. If the attribute isn’t callable (the default is `None`), the source is simply renamed to the destination.

Parameters::

- **source** – The source filename. This i normally the base filename, e.g. ‘test.log’.
- **dest** – The destination filename. This is normally what the source is rotate to, e.g. ‘test.log.1’.

Added in version 3.3.

The reason the attributes exist is to save you having to subclass - you can use the same callables for instances of `RotatingFileHandler` and `TimedRotatingFileHandler`. If either the namer or rotator callable raises an exception, this will be handled in the same way as any other exception during an **`emit()`** call, i.e. via the **`handleError()`** method of the handler.

If you need to make more significant changes to rotation processing, you can override the methods.

For an example, see Using a rotator and namer to customize log rotation processing.

RotatingFileHandler

The `RotatingFileHandler` class, located in the `logging.handlers` module, supports rotation of disk log files.

class logging.handlers.RotatingFileHandler(*filename*, *mode='a'*, *maxBytes=0*, *backupCount=0*, *encoding=None*, *delay=False*, *errors=None*)

Returns a new instance of the `RotatingFileHandler` class. The specified file is opened and used as the stream for logging. If *mode* is not specified, `'a'` is used. If *encoding* is not `None`, it is used to open the file with that encoding. If *delay* is true, then file opening is deferred until the first call to `emit()`. By default, the file grows indefinitely. If *errors* is provided, it determines how encoding errors are handled.

You can use the *maxBytes* and *backupCount* values to allow the file to *rollover* at a predetermined size. When the size is about to be exceeded, the file is closed and a new file is silently opened for output. Rollover occurs whenever the current log file is nearly *maxBytes* in length; but if either of *maxBytes* or *backupCount* is zero, rollover never occurs, so you generally want to set *backupCount* to at least 1, and have a non-zero *maxBytes*. When *backupCount* is non-zero, the system will save old log files by appending the extensions ‘.1’, ‘.2’ etc., to the filename. For example, with a *backupCount* of 5 and a base file name of `app.log`, you would get `app.log`, `app.log.1`, `app.log.2`, up to `app.log.5`. The file being written to is always `app.log`. When this file is filled, it is closed and renamed to `app.log.1`, and if files `app.log.1`, `app.log.2`, etc. exist, then they are renamed to `app.log.2`, `app.log.3` etc. respectively.

Changed in version 3.6: As well as string values, `Path` objects are also accepted for the *filename* argument.

Changed in version 3.9: The *errors* parameter was added.

doRollover()

Does a rollover, as described above.

emit(*record*)

Outputs the record to the file, catering for rollover as described previously.

shouldRollover(*record*)

See if the supplied record would cause the file to exceed the configured size limit.

TimedRotatingFileHandler

The `TimedRotatingFileHandler` class, located in the `logging.handlers` module, supports rotation of disk log files at certain timed intervals.

class logging.handlers.TimedRotatingFileHandler(*filename*, *when='h'*, *interval=1*, *backupCount=0*, *encoding=None*, *delay=False*, *utc=False*, *atTime=None*, *errors=None*)

> Returns a new instance of the `TimedRotatingFileHandler` class. The specified file is opened and used as the stream for logging. On rotating it also sets the filename suffix. Rotating happens based on the product of *when* and *interval*.
>
> You can use the *when* to specify the type of *interval*. The list of possible values is below. Note that they are not case sensitive.

Value	Type of interval	If/how *atTime* is used
`'S'`	Seconds	Ignored
`'M'`	Minutes	Ignored
`'H'`	Hours	Ignored
`'D'`	Days	Ignored
`'W0'-'W6'`	Weekday (0=Monday)	Used to compute initial rollover time
`'midnight'`	Roll over at midnight, if *atTime* not specified, else at	Used to compute initial rollover

Value	**Type of interval**	**If/how *atTime* is used**
	time *atTime*	time

When using weekday-based rotation, specify ‘W0’ for Monday, ‘W1’ for Tuesday, and so on up to ‘W6’ for Sunday. In this case, the value passed for *interval* isn’t used.

The system will save old log files by appending extensions to the filename. The extensions are date-and-time based, using the strftime format `%Y-%m-%d_%H-%M-%S` or a leading portion thereof, depending on the rollover interval.

When computing the next rollover time for the first time (when the handler is created), the last modification time of an existing log file, or else the current time, is used to compute when the next rotation will occur.

If the *utc* argument is true, times in UTC will be used; otherwise local time is used.

If *backupCount* is nonzero, at most *backupCount* files will be kept, and if more would be created when rollover occurs, the oldest one is deleted. The deletion logic uses the interval to determine which files to delete, so changing the interval may leave old files lying around.

If *delay* is true, then file opening is deferred until the first call to `emit()`.

If *atTime* is not `None`, it must be a `datetime.time` instance which specifies the time of day when rollover occurs, for the cases where rollover is set to happen “at midnight” or “on a particular weekday”. Note that in these cases, the *atTime* value is effectively used to compute the *initial* rollover, and subsequent rollovers would be calculated via the normal interval calculation.

If *errors* is specified, it’s used to determine how encoding errors are handled.

Note

Calculation of the initial rollover time is done when the handler is initialised. Calculation of subsequent rollover times is done only when rollover occurs, and rollover occurs only when emitting output. If this is not kept in mind, it might lead to some confusion. For example, if an interval of “every minute” is set, that does not mean you will always see log files with times (in the filename) separated by a minute; if, during application execution, logging output is generated more frequently than once a minute, *then* you can expect to see log files with times separated by a minute. If, on the other hand, logging messages are only output once every five minutes (say), then there will be gaps in the file times corresponding to the minutes where no output (and hence no rollover) occurred.

Changed in version 3.4: atTime parameter was added.

Changed in version 3.6: As well as string values, `Path` objects are also accepted for the *filename* argument.

Changed in version 3.9: The *errors* parameter was added.

doRollover()

Does a rollover, as described above.

emit(*record*)

Outputs the record to the file, catering for rollover as described above.

getFilesToDelete()

Returns a list of filenames which should be deleted as part of rollover. These are the absolute paths of the oldest backup log files written by the handler.

shouldRollover(*record*)

See if enough time has passed for a rollover to occur and if it has, compute the next rollover time.

SocketHandler

The `SocketHandler` class, located in the `logging.handlers` module, sends logging output to a network socket. The base class uses a TCP socket.

class logging.handlers.SocketHandler(*host*, *port*)

Returns a new instance of the `SocketHandler` class intended to communicate with a remote machine whose address is given by *host* and *port*.

Changed in version 3.4: If `port` is specified as `None`, a Unix domain socket is created using the value in `host` - otherwise, a TCP socket is created.

close()

Closes the socket.

emit()

Pickles the record’s attribute dictionary and writes it to the socket in binary format. If there is an error with the socket, silently drops the packet. If the connection was previously lost, re-establishes the connection. To unpickle the record at the receiving end into a `LogRecord`, use the `makeLogRecord()` function.

handleError()

Handles an error which has occurred during `emit()`. The most likely cause is a lost connection. Closes the socket so that we can retry on the next event.

makeSocket()

This is a factory method which allows subclasses to define the precise type of socket they want. The default implementation creates a TCP socket (`socket.SOCK_STREAM`).

makePickle(*record*)

Pickles the record’s attribute dictionary in binary format with a length prefix, and returns it ready for transmission across the socket. The details of this operation are equivalent to:

```
data = pickle.dumps(record_attr_dict, 1)
datalen = struct.pack('>L', len(data))
return datalen + data
```

Note that pickles aren’t completely secure. If you are concerned about security, you may want to override this method to implement a more secure mechanism. For example, you can sign pickles using HMAC and then verify them on the receiving end, or alternatively you can disable unpickling of global objects on the receiving end.

send(*packet*)

Send a pickled byte-string *packet* to the socket. The format of the sent byte-string is as described in the documentation for `makePickle()`.

This function allows for partial sends, which can happen when the network is busy.

createSocket()

Tries to create a socket; on failure, uses an exponential back-off algorithm. On initial failure, the handler will drop the message it was trying to send. When subsequent messages are handled by the same instance, it will not try connecting until some time has passed. The default parameters are such that the initial delay is one second, and if after that delay the connection still can’t be made, the handler will double the delay each time up to a maximum of 30 seconds.

This behaviour is controlled by the following handler attributes:

- `retryStart` (initial delay, defaulting to 1.0 seconds).
- `retryFactor` (multiplier, defaulting to 2.0).
- `retryMax` (maximum delay, defaulting to 30.0 seconds).

This means that if the remote listener starts up *after* the handler has been used, you could lose messages (since the handler won’t even attempt a connection until the delay has elapsed, but just silently drop messages during the delay period).

DatagramHandler

The **`DatagramHandler`** class, located in the **`logging.handlers`** module, inherits from **`SocketHandler`** to support sending logging messages over UDP sockets.

class logging.handlers.DatagramHandler(*host*, *port*)

Returns a new instance of the **`DatagramHandler`** class intended to communicate with a remote machine whose address is given by *host* and *port*.

Note

As UDP is not a streaming protocol, there is no persistent connection between an instance of this handler and *host*. For this reason, when using a network socket, a DNS lookup might have to be made each time an event is logged, which can introduce some latency into the system. If this affects you, you can do a lookup yourself and initialize this handler using the looked-up IP address rather than the hostname.

Changed in version 3.4: If `port` is specified as `None`, a Unix domain socket is created using the value in `host` - otherwise, a UDP socket is created.

emit()

Pickles the record’s attribute dictionary and writes it to the socket in binary format. If there is an error with the socket, silently drops the packet. To unpickle the record at the receiving end into a `LogRecord`, use the `makeLogRecord()` function.

makeSocket()

The factory method of `SocketHandler` is here overridden to create a UDP socket (`socket.SOCK_DGRAM`).

send(*s*)

Send a pickled byte-string to a socket. The format of the sent byte-string is as described in the documentation for `SocketHandler.makePickle()`.

SysLogHandler

The `SysLogHandler` class, located in the `logging.handlers` module, supports sending logging messages to a remote or local Unix syslog.

class logging.handlers.SysLogHandler(*address=('localhost', SYSLOG_UDP_PORT)*, *facility=LOG_USER*, *socktype=socket.SOCK_DGRAM*, *timeout=None*)

Returns a new instance of the `SysLogHandler` class intended to communicate with a remote Unix machine whose address is given by *address* in the form of a `(host, port)` tuple. If *address* is not specified, `('localhost', 514)` is used. The address is used to open a socket. An alternative to providing a `(host, port)` tuple is providing an address as a string, for example ‘/dev/log’. In this case, a Unix domain socket is used to send the message to the syslog. If *facility* is not specified, **`LOG_USER`** is used. The type of socket opened depends on the *socktype* argument, which defaults to `socket.SOCK_DGRAM` and thus opens a UDP socket. To open a TCP socket (for use with the newer syslog daemons such as rsyslog), specify a value of `socket.SOCK_STREAM`. If *timeout* is specified, it sets a timeout (in seconds) for the socket operations. This can help prevent the program from hanging indefinitely if the syslog server is unreachable. By default, *timeout* is `None`, meaning no timeout is applied.

Note that if your server is not listening on UDP port 514, `SysLogHandler` may appear not to work. In that case, check what address you should be using for a domain socket - it’s system dependent. For example, on Linux it’s usually ‘/dev/log’ but on OS/X it’s ‘/var/run/syslog’. You’ll need to check your platform and use the appropriate address (you may need to do this check at runtime if your application needs to run on several platforms). On Windows, you pretty much have to use the UDP option.

Note

On macOS 12.x (Monterey), Apple has changed the behaviour of their syslog daemon - it no longer listens on a domain socket. Therefore, you cannot expect `SysLogHandler` to work on this system.

See gh-91070 [https://github.com/python/cpython/issues/91070] for more information.

Changed in version 3.2: socktype was added.

Changed in version 3.14: timeout was added.

close()

Closes the socket to the remote host.

createSocket()

Tries to create a socket and, if it’s not a datagram socket, connect it to the other end. This method is called during handler initialization, but it’s not regarded as an error if the other end isn’t listening at this point - the method will be called again when emitting an event, if there is no socket at that point.

Added in version 3.11.

emit(*record*)

The record is formatted, and then sent to the syslog server. If exception information is present, it is *not* sent to the server.

Changed in version 3.2.1: (See: bpo-12168 [https://bugs.python.org/issue?@action=redirect&bpo=12168].) In earlier versions, the message sent to the syslog daemons was always terminated with a NUL byte, because early versions of these daemons expected a NUL terminated message - even though it’s not in the relevant specification (**RFC 5424** [https://datatracker.ietf.org/doc/html/rfc5424.html]). More recent versions of these daemons don’t expect the NUL byte but strip it off if it’s there, and even more recent daemons (which adhere more closely to RFC 5424) pass the NUL byte on as part of the message.

To enable easier handling of syslog messages in the face of all these differing daemon behaviours, the appending of the NUL byte has been made configurable, through the use of a class-level attribute, `append_nul`. This defaults to `True` (preserving the existing behaviour) but can be set to `False` on a `SysLogHandler` instance in order for that instance to *not* append the NUL terminator.
Changed in version 3.3: (See: bpo-12419 [https://bugs.python.org/issue?@action=redirect&bpo=12419].) In earlier versions, there was no facility for an “ident” or “tag” prefix to identify the source of the message. This can now be specified using a class-level attribute, defaulting to `""` to preserve existing behaviour, but which can be overridden on a `SysLogHandler` instance in order for that instance to prepend the ident to every message handled. Note that the provided ident must be text, not bytes, and is prepended to the message exactly as is.

encodePriority(*facility*, *priority*)

Encodes the facility and priority into an integer. You can pass in strings or integers - if strings are passed, internal mapping dictionaries are used to convert them to integers.

The symbolic `LOG_` values are defined in **`SysLogHandler`** and mirror the values defined in the `sys/syslog.h` header file.

Priorities

Name (string)	Symbolic value
`alert`	LOG_ALERT
`crit` or `critical`	LOG_CRIT
`debug`	LOG_DEBUG
`emerg` or `panic`	LOG_EMERG
`err` or `error`	LOG_ERR
`info`	LOG_INFO
`notice`	LOG_NOTICE
`warn` or `warning`	LOG_WARNING

Facilities

Name (string)	Symbolic value
`auth`	LOG_AUTH
`authpriv`	LOG_AUTHPRIV
`cron`	LOG_CRON
`daemon`	LOG_DAEMON
`ftp`	LOG_FTP
`kern`	LOG_KERN
`lpr`	LOG_LPR
`mail`	LOG_MAIL

Name (string)	Symbolic value
`news`	LOG_NEWS
`syslog`	LOG_SYSLOG
`user`	LOG_USER
`uucp`	LOG_UUCP
`local0`	LOG_LOCAL0
`local1`	LOG_LOCAL1
`local2`	LOG_LOCAL2
`local3`	LOG_LOCAL3
`local4`	LOG_LOCAL4
`local5`	LOG_LOCAL5
`local6`	LOG_LOCAL6
`local7`	LOG_LOCAL7

mapPriority(*levelname*)

Maps a logging level name to a syslog priority name. You may need to override this if you are using custom levels, or if the default algorithm is not suitable for your needs. The default algorithm maps `DEBUG`, `INFO`, `WARNING`, `ERROR` and `CRITICAL` to the equivalent syslog names, and all other level names to ‘warning’.

NTEventLogHandler

The `NTEventLogHandler` class, located in the `logging.handlers` module, supports sending logging messages to a local Windows NT, Windows 2000 or Windows XP event log. Before you can use it, you need Mark Hammond’s Win32 extensions for Python installed.

class logging.handlers.NTEventLogHandler(*appname*, *dllname=None*, *logtype='Application'*)

Returns a new instance of the `NTEventLogHandler` class. The *appname* is used to define the application name as it appears in the event log. An appropriate registry entry is created using this name. The *dllname* should give the fully qualified pathname of a .dll or .exe which contains message definitions to hold in the log (if not specified, `'win32service.pyd'` is used - this is installed with the Win32 extensions and contains some basic placeholder message definitions. Note that use of these placeholders will make your event logs big, as the entire message source is held in the log. If you want slimmer logs, you have to pass in the name of your own .dll or .exe which contains the message definitions you want to use in the event log). The *logtype* is one of `'Application'`, `'System'` or `'Security'`, and defaults to `'Application'`.

close()

At this point, you can remove the application name from the registry as a source of event log entries. However, if you do this, you will not be able to see the events as you intended in the Event Log Viewer - it needs to be able to access the registry to get the .dll name. The current version does not do this.

emit(*record*)

Determines the message ID, event category and event type, and then logs the message in the NT event log.

getEventCategory(*record*)

Returns the event category for the record. Override this if you want to specify your own categories. This version returns 0.

getEventType(*record*)

Returns the event type for the record. Override this if you want to specify your own types. This version does a mapping using the handler's typemap attribute, which is set up in **`__init__()`** to a dictionary which contains mappings for **`DEBUG`**, **`INFO`**, **`WARNING`**, **`ERROR`** and **`CRITICAL`**. If you are using your own levels, you will either need to override this method or place a suitable dictionary in the handler's *typemap* attribute.

getMessageID(*record*)

Returns the message ID for the record. If you are using your own messages, you could do this by having the *msg* passed to the logger being an ID rather than a format string. Then, in here, you could use a dictionary lookup to get the message ID. This version returns 1, which is the base message ID in `win32service.pyd`.

SMTPHandler

The **`SMTPHandler`** class, located in the **`logging.handlers`** module, supports sending logging messages to an email address via SMTP.

class logging.handlers.SMTPHandler(*mailhost*, *fromaddr*, *toaddrs*, *subject*, *credentials=None*, *secure=None*, *timeout=1.0*)

Returns a new instance of the **`SMTPHandler`** class. The instance is initialized with the from and to addresses and subject line of the email. The *toaddrs* should be a list of strings. To specify a non-standard SMTP port, use the (host, port) tuple format for the *mailhost* argument. If you use a string, the standard SMTP port is used. If your SMTP server requires authentication, you can specify a (username, password) tuple for the *credentials* argument.

To specify the use of a secure protocol (TLS), pass in a tuple to the *secure* argument. This will only be used when authentication credentials are supplied. The tuple should be either an empty tuple, or a single-value tuple with the name of a keyfile, or a 2-value tuple with the names of the keyfile and certificate file. (This tuple is passed to the `smtplib.SMTP.starttls()` method.)

A timeout can be specified for communication with the SMTP server using the *timeout* argument.

Changed in version 3.3: Added the *timeout* parameter.

emit(*record*)

Formats the record and sends it to the specified addressees.

getSubject(*record*)

If you want to specify a subject line which is record-dependent, override this method.

MemoryHandler

The `MemoryHandler` class, located in the `logging.handlers` module, supports buffering of logging records in memory, periodically flushing them to a *target* handler. Flushing occurs whenever the buffer is full, or when an event of a certain severity or greater is seen.

`MemoryHandler` is a subclass of the more general `BufferingHandler`, which is an abstract class. This buffers logging records in memory. Whenever each record is added to the buffer, a check is made by calling **`shouldFlush()`** to see if the buffer should be flushed. If it should, then **`flush()`** is expected to do the flushing.

class logging.handlers.BufferingHandler(*capacity*)

Initializes the handler with a buffer of the specified capacity. Here, *capacity* means the number of logging records buffered.

emit(*record*)

Append the record to the buffer. If `shouldFlush()` returns true, call `flush()` to process the buffer.

flush()

For a `BufferingHandler` instance, flushing means that it sets the buffer to an empty list. This method can be overwritten to implement more useful flushing behavior.

shouldFlush(*record*)

Return `True` if the buffer is up to capacity. This method can be overridden to implement custom flushing strategies.

class logging.handlers.MemoryHandler(*capacity*, *flushLevel=ERROR*, *target=None*, *flushOnClose=True*)

Returns a new instance of the `MemoryHandler` class. The instance is initialized with a buffer size of *capacity* (number of records buffered). If *flushLevel* is not specified, **`ERROR`** is used. If no *target* is specified, the target will need to be set using `setTarget()` before this handler does anything useful. If *flushOnClose* is specified as `False`, then the buffer is *not* flushed when the handler is closed. If not specified or specified as `True`, the previous behaviour of flushing the buffer will occur when the handler is closed.

Changed in version 3.6: The *flushOnClose* parameter was added.

close()

Calls `flush()`, sets the target to `None` and clears the buffer.

flush()

For a `MemoryHandler` instance, flushing means just sending the buffered records to the target, if there is one. The buffer is also cleared when buffered records are sent to the target. Override if you want different behavior.

setTarget(*target*)

Sets the target handler for this handler.

shouldFlush(*record*)

Checks for buffer full or a record at the *flushLevel* or higher.

HTTPHandler

The `HTTPHandler` class, located in the `logging.handlers` module, supports sending logging messages to a web server, using either `GET` or `POST` semantics.

class logging.handlers.HTTPHandler(*host*, *url*, *method='GET'*, *secure=False*, *credentials=None*, *context=None*)

Returns a new instance of the `HTTPHandler` class. The *host* can be of the form `host:port`, should you need to use a specific port number. If no *method* is specified, `GET` is used.

If *secure* is true, a HTTPS connection will be used. The *context* parameter may be set to a `ssl.SSLContext` instance to configure the SSL settings used for the HTTPS connection. If *credentials* is specified, it should be a 2-tuple consisting of userid and password, which will be placed in a HTTP ‘Authorization’ header using Basic authentication. If you specify credentials, you should also specify secure=True so that your userid and password are not passed in cleartext across the wire.

Changed in version 3.5: The *context* parameter was added.

mapLogRecord(*record*)

Provides a dictionary, based on `record`, which is to be URL-encoded and sent to the web server. The default implementation just returns `record.__dict__`. This method can be overridden if e.g. only a subset of `LogRecord` is to be sent to the web server, or if more specific customization of what’s sent to the server is required.

emit(*record*)

Sends the record to the web server as a URL-encoded dictionary. The `mapLogRecord()` method is used to convert the record to the dictionary to be sent.

Note

Since preparing a record for sending it to a web server is not the same as a generic formatting operation, using `setFormatter()` to specify a `Formatter` for a `HTTPHandler` has no effect. Instead of calling `format()`, this handler calls `mapLogRecord()` and then `urllib.parse.urlencode()` to encode the dictionary in a form suitable for sending to a web server.

QueueHandler

Added in version 3.2.

The `QueueHandler` class, located in the `logging.handlers` module, supports sending logging messages to a queue, such as those implemented in the `queue` or `multiprocessing` modules.

Along with the `QueueListener` class, `QueueHandler` can be used to let handlers do their work on a separate thread from the one which does the logging. This is important in web applications and also other service applications where threads servicing clients need to respond as quickly as possible, while any potentially slow operations (such as sending an email via `SMTPHandler`) are done on a separate thread.

class logging.handlers.QueueHandler(*queue*)

Returns a new instance of the `QueueHandler` class. The instance is initialized with the queue to send messages to. The *queue* can be any queue-like object; it’s used as-is by the `enqueue()` method, which needs to know how to send messages to it. The queue is not *required* to have the task tracking API, which means that you can use `SimpleQueue` instances for *queue*.

Note

If you are using `multiprocessing`, you should avoid using `SimpleQueue` and instead use `multiprocessing.Queue`.

Warning

The `multiprocessing` module uses an internal logger created and accessed via `get_logger()`. `multiprocessing.Queue` will log `DEBUG` level messages upon items being queued. If those log messages are processed by a `QueueHandler` using the same `multiprocessing.Queue` instance, it will cause a deadlock or infinite recursion.

emit(*record*)

Enqueues the result of preparing the LogRecord. Should an exception occur (e.g. because a bounded queue has filled up), the `handleError()` method is called to handle the error. This can result in the record silently being dropped (if `logging.raiseExceptions` is `False`) or a message printed to `sys.stderr` (if `logging.raiseExceptions` is `True`).

prepare(*record*)

Prepares a record for queuing. The object returned by this method is enqueued.

The base implementation formats the record to merge the message, arguments, exception and stack information, if present. It also removes unpickleable items from the record in-place. Specifically, it overwrites the record's **`msg`** and **`message`** attributes with the merged message (obtained by calling the handler's `format()` method), and sets the **`args`**, **`exc_info`** and **`exc_text`** attributes to `None`.

You might want to override this method if you want to convert the record to a dict or JSON string, or send a modified copy of the record while leaving the original intact.

Note

The base implementation formats the message with arguments, sets the `message` and `msg` attributes to the formatted message and sets the `args` and `exc_text` attributes to `None` to allow pickling and to prevent further attempts at formatting. This means that a handler on the `QueueListener` side won't have the information to do custom formatting, e.g. of exceptions. You may wish to subclass `QueueHandler` and override this method to e.g. avoid setting `exc_text` to `None`. Note that the `message` / `msg` / `args` changes are related to ensuring the record is pickleable, and you might or might not be able to avoid doing that depending on whether your `args` are pickleable. (Note that you may have to consider not only your own code but also code in any libraries that you use.)

enqueue(*record*)

Enqueues the record on the queue using `put_nowait()`; you may want to override this if you want to use blocking behaviour, or a timeout, or a customized queue implementation.

listener

When created via configuration using `dictConfig()`, this attribute will contain a `QueueListener` instance for use with this handler. Otherwise, it will be `None`.

Added in version 3.12.

QueueListener

Added in version 3.2.

The `QueueListener` class, located in the `logging.handlers` module, supports receiving logging messages from a queue, such as those implemented in the `queue` or `multiprocessing` modules. The messages are received from a queue in an internal thread and passed, on the same thread, to one or more handlers for processing. While `QueueListener` is not itself a handler, it is documented here because it works hand-in-hand with `QueueHandler`.

Along with the `QueueHandler` class, `QueueListener` can be used to let handlers do their work on a separate thread from the one which does the logging. This is important in web applications and also other service applications where threads servicing clients need to respond as quickly as possible, while any potentially slow operations (such as sending an email via `SMTPHandler`) are done on a separate thread.

class logging.handlers.QueueListener(*queue*, **handlers*, *respect_handler_level=False*)

Returns a new instance of the `QueueListener` class. The instance is initialized with the queue to send messages to and a list of handlers which will handle entries placed on the queue. The queue can be any queue-like object; it's passed as-is to the `dequeue()` method, which needs to know how to get messages from it. The queue is not *required* to have the task tracking API (though it's used if available), which means that you can use `SimpleQueue` instances for *queue*.

Note

If you are using `multiprocessing`, you should avoid using `SimpleQueue` and instead use `multiprocessing.Queue`.

If `respect_handler_level` is `True`, a handler's level is respected (compared with the level for the message) when deciding whether to pass messages to that handler; otherwise, the behaviour is as in previous Python versions - to always pass each message to each handler.

Changed in version 3.5: The `respect_handler_level` argument was added.

Changed in version 3.14: `QueueListener` can now be used as a context manager via `with`. When entering the context, the listener is started. When exiting the context, the listener is stopped. `__enter__()` returns the `QueueListener` object.

dequeue(*block*)

Dequeues a record and return it, optionally blocking.

The base implementation uses `get()`. You may want to override this method if you want to use timeouts or work with custom queue implementations.

prepare(*record*)

Prepare a record for handling.

This implementation just returns the passed-in record. You may want to override this method if you need to do any custom marshalling or manipulation of the record before passing it to the handlers.

handle(*record*)

Handle a record.

This just loops through the handlers offering them the record to handle. The actual object passed to the handlers is that which is returned from `prepare()`.

start()

Starts the listener.

This starts up a background thread to monitor the queue for LogRecords to process.

Changed in version 3.14: Raises `RuntimeError` if called and the listener is already running.

stop()

Stops the listener.

This asks the thread to terminate, and then waits for it to do so. Note that if you don’t call this before your application exits, there may be some records still left on the queue, which won’t be processed.

enqueue_sentinel()

Writes a sentinel to the queue to tell the listener to quit. This implementation uses `put_nowait()`. You may want to override this method if you want to use timeouts or work with custom queue implementations.

Added in version 3.3.

See also

Module `logging`

API reference for the logging module.

Module `logging.config`

Configuration API for the logging module.

`platform` — Access to underlying platform’s identifying data

Source code: Lib/platform.py
[https://github.com/python/cpython/tree/3.14/Lib/platform.py]

Note

Specific platforms listed alphabetically, with Linux included in the Unix section.

Cross platform

platform.architecture(*executable=sys.executable*, *bits=*'', *linkage=*'')

Queries the given executable (defaults to the Python interpreter binary) for various architecture information.

Returns a tuple `(bits, linkage)` which contain information about the bit architecture and the linkage format used for the executable. Both values are returned as strings.

Values that cannot be determined are returned as given by the parameter presets. If bits is given as `''`, the `sizeof(pointer)` (or `sizeof(long)` on Python version < 1.5.2) is used as indicator for the supported pointer size.

The function relies on the system's `file` command to do the actual work. This is available on most if not all Unix platforms and some non-Unix platforms and then only if the executable points to the Python interpreter. Reasonable defaults are used when the above needs are not met.

Note

On macOS (and perhaps other platforms), executable files may be universal files containing multiple architectures.

To get at the "64-bitness" of the current interpreter, it is more reliable to query the `sys.maxsize` attribute:

```
is_64bits = sys.maxsize > 2**32
```

platform.machine()

Returns the machine type, e.g. `'AMD64'`. An empty string is returned if the value cannot be determined.

The output is platform-dependent and may differ in casing and naming conventions.

platform.node()

Returns the computer's network name (may not be fully qualified!). An empty string is returned if the value cannot be determined.

platform.platform(*aliased=False*, *terse=False*)

Returns a single string identifying the underlying platform with as much useful information as possible.

The output is intended to be *human readable* rather than machine parseable. It may look different on different platforms and this is intended.

If *aliased* is true, the function will use aliases for various platforms that report system names which differ from their common names, for example SunOS will be reported as Solaris. The `system_alias()` function is used to implement this.

Setting *terse* to true causes the function to return only the absolute minimum information needed to identify the platform.

Changed in version 3.8: On macOS, the function now uses `mac_ver()`, if it returns a non-empty release string, to get the macOS version rather than the darwin version.

platform.processor()

Returns the (real) processor name, e.g. `'amdk6'`.

An empty string is returned if the value cannot be determined. Note that many platforms do not provide this information or simply return the same value as for `machine()`. NetBSD does this.

platform.python_build()

Returns a tuple `(buildno, builddate)` stating the Python build number and date as strings.

platform.python_compiler()

Returns a string identifying the compiler used for compiling Python.

platform.python_branch()

Returns a string identifying the Python implementation SCM branch.

platform.python_implementation()

Returns a string identifying the Python implementation. Possible return values are: ‘CPython’, ‘IronPython’, ‘Jython’, ‘PyPy’.

platform.python_revision()

Returns a string identifying the Python implementation SCM revision.

platform.python_version()

Returns the Python version as string `'major.minor.patchlevel'`.

Note that unlike the Python `sys.version`, the returned value will always include the patchlevel (it defaults to 0).

platform.python_version_tuple()

Returns the Python version as tuple `(major, minor, patchlevel)` of strings.

Note that unlike the Python `sys.version`, the returned value will always include the patchlevel (it defaults to `'0'`).

platform.release()

Returns the system’s release, e.g. `'2.2.0'` or `'NT'`. An empty string is returned if the value cannot be determined.

platform.system()

Returns the system/OS name, such as `'Linux'`, `'Darwin'`, `'Java'`, `'Windows'`. An empty string is returned if the value cannot be determined.

On iOS and Android, this returns the user-facing OS name (i.e, `'iOS`, `'iPadOS'` or `'Android'`). To obtain the kernel name (`'Darwin'` or `'Linux'`), use `os.uname()`.

platform.system_alias(*system*, *release*, *version*)

Returns `(system, release, version)` aliased to common marketing names used for some systems. It also does some reordering of the information in some cases where it would otherwise cause confusion.

platform.version()

Returns the system's release version, e.g. `'#3 on degas'`. An empty string is returned if the value cannot be determined.

On iOS and Android, this is the user-facing OS version. To obtain the Darwin or Linux kernel version, use `os.uname()`.

platform.uname()

Fairly portable uname interface. Returns a `namedtuple()` containing six attributes: `system`, `node`, `release`, `version`, `machine`, and `processor`.

`processor` is resolved late, on demand.

Note: the first two attribute names differ from the names presented by `os.uname()`, where they are named **`sysname`** and **`nodename`**.

Entries which cannot be determined are set to `''`.

Changed in version 3.3: Result changed from a tuple to a `namedtuple()`.

Changed in version 3.9: `processor` is resolved late instead of immediately.

platform.invalidate_caches()

Clear out the internal cache of information, such as the `uname()`. This is typically useful when the platform's `node()` is changed by an external process and one needs to retrieve the updated value.

Added in version 3.14.

Java platform

platform.java_ver(*release=''*, *vendor=''*, *vminfo=('', '', '')*, *osinfo=('', '', '')*)

Version interface for Jython.

Returns a tuple `(release, vendor, vminfo, osinfo)` with *vminfo* being a tuple `(vm_name, vm_release, vm_vendor)` and *osinfo* being a tuple `(os_name, os_version, os_arch)`. Values which cannot be determined are set to the defaults given as parameters (which all default to `''`).

Deprecated since version 3.13, will be removed in version 3.15: It was largely untested, had a confusing API, and was only useful for Jython support.

Windows platform

platform.win32_ver(*release=''*, *version=''*, *csd=''*, *ptype=''*)

Get additional version information from the Windows Registry and return a tuple `(release, version, csd, ptype)` referring to OS release, version number, CSD level (service pack) and OS type (multi/single processor). Values which cannot be determined are set to the defaults given as parameters (which all default to an empty string).

As a hint: *ptype* is `'Uniprocessor Free'` on single processor NT machines and `'Multiprocessor Free'` on multi processor machines. The `'Free'` refers to the OS version being free of debugging code. It could also state `'Checked'` which means the OS version uses debugging code, i.e. code that checks arguments, ranges, etc.

platform.win32_edition()

Returns a string representing the current Windows edition, or `None` if the value cannot be determined. Possible values include but are not limited to `'Enterprise'`, `'IoTUAP'`, `'ServerStandard'`, and `'nanoserver'`.

Added in version 3.8.

platform.win32_is_iot()

Return `True` if the Windows edition returned by `win32_edition()` is recognized as an IoT edition.

Added in version 3.8.

macOS platform

platform.mac_ver(*release*='', *versioninfo=('', '', '')*, *machine*='')

Get macOS version information and return it as tuple `(release, versioninfo, machine)` with *versioninfo* being a tuple `(version, dev_stage, non_release_version)`.

Entries which cannot be determined are set to `''`. All tuple entries are strings.

iOS platform

platform.ios_ver(*system*='', *release*='', *model*='', *is_simulator*=*False*)

Get iOS version information and return it as a `namedtuple()` with the following attributes:

- `system` is the OS name; either `'iOS'` or `'iPadOS'`.
- `release` is the iOS version number as a string (e.g., `'17.2'`).
- `model` is the device model identifier; this will be a string like `'iPhone13,2'` for a physical device, or `'iPhone'` on a simulator.
- `is_simulator` is a boolean describing if the app is running on a simulator or a physical device.

Entries which cannot be determined are set to the defaults given as parameters.

Unix platforms

platform.libc_ver(*executable*=*sys.executable*, *lib*='', *version*='', *chunksize*=*16384*)

Tries to determine the libc version against which the file executable (defaults to the Python interpreter) is linked. Returns a tuple of strings `(lib, version)` which default to the given parameters in case the lookup fails.

Note that this function has intimate knowledge of how different libc versions add symbols to the executable is probably only usable for executables compiled using **gcc**.

The file is read and scanned in chunks of *chunksize* bytes.

Linux platforms

platform.freedesktop_os_release()

Get operating system identification from `os-release` file and return it as a dict. The `os-release` file is a freedesktop.org standard [https://www.freedesktop.org/software/systemd/man/os-release.html] and is available in most Linux distributions. A noticeable exception is Android and Android-based distributions.

Raises **`OSError`** or subclass when neither `/etc/os-release` nor `/usr/lib/os-release` can be read.

On success, the function returns a dictionary where keys and values are strings. Values have their special characters like `"` and `$` unquoted. The fields `NAME`, `ID`, and `PRETTY_NAME` are always defined according to the standard. All other fields are optional. Vendors may include additional fields.

Note that fields like `NAME`, `VERSION`, and `VARIANT` are strings suitable for presentation to users. Programs should use fields like `ID`, `ID_LIKE`, `VERSION_ID`, or `VARIANT_ID` to identify Linux distributions.

Example:

```
def get_like_distro():
    info = platform.freedesktop_os_release()
    ids = [info["ID"]]
    if "ID_LIKE" in info:
        # ids are space separated and ordered by precedence
        ids.extend(info["ID_LIKE"].split())
    return ids
```

Added in version 3.10.

Android platform

platform.android_ver(*release=''*, *api_level=0*, *manufacturer=''*, *model=''*, *device=''*, *is_emulator=False*)

Get Android device information. Returns a `namedtuple()` with the following attributes. Values which cannot be determined are set to the defaults given as parameters.

- `release` - Android version, as a string (e.g. `"14"`).
- `api_level` - API level of the running device, as an integer (e.g. `34` for Android 14). To get the API level which Python was built against, see `sys.getandroidapilevel()`.
- `manufacturer` - Manufacturer name [https://developer.android.com/reference/android/os/Build#MANUFACTURER].
- `model` - Model name [https://developer.android.com/reference/android/os/Build#MODEL] – typically the marketing name or model number.
- `device` - Device name [https://developer.android.com/reference/android/os/Build#DEVICE] – typically the model number or a codename.
- `is_emulator` - `True` if the device is an emulator; `False` if it's a physical device.

Google maintains a list of known model and device names [https://storage.googleapis.com/play_public/supported_devices.html].

Added in version 3.13.

Command-line usage

`platform` can also be invoked directly using the `-m` switch of the interpreter:

```
python -m platform [--terse] [--nonaliased] [{nonaliased,terse}
...]
```

The following options are accepted:

--terse

Print terse information about the platform. This is equivalent to calling `platform.platform()` with the *terse* argument set to `True`.

--nonaliased

Print platform information without system/OS name aliasing. This is equivalent to calling `platform.platform()` with the *aliased* argument set to `True`.

You can also pass one or more positional arguments (`terse`, `nonaliased`) to explicitly control the output format. These behave similarly to their corresponding options.

`errno` — Standard errno system symbols

This module makes available standard `errno` system symbols. The value of each symbol is the corresponding integer value. The names and descriptions are borrowed from `linux/include/errno.h`, which should be all-inclusive.

errno.errorcode

Dictionary providing a mapping from the errno value to the string name in the underlying system. For instance, `errno.errorcode[errno.EPERM]` maps to `'EPERM'`.

To translate a numeric error code to an error message, use `os.strerror()`.

Of the following list, symbols that are not used on the current platform are not defined by the module. The specific list of defined symbols is available as `errno.errorcode.keys()`. Symbols available can include:

errno.EPERM

Operation not permitted. This error is mapped to the exception `PermissionError`.

errno.ENOENT

No such file or directory. This error is mapped to the exception `FileNotFoundError`.

errno.ESRCH

No such process. This error is mapped to the exception `ProcessLookupError`.

errno.EINTR
: Interrupted system call. This error is mapped to the exception `InterruptedError`.

errno.EIO
: I/O error

errno.ENXIO
: No such device or address

errno.E2BIG
: Arg list too long

errno.ENOEXEC
: Exec format error

errno.EBADF
: Bad file number

errno.ECHILD
: No child processes. This error is mapped to the exception `ChildProcessError`.

errno.EAGAIN
: Try again. This error is mapped to the exception `BlockingIOError`.

errno.ENOMEM
: Out of memory

errno.EACCES
: Permission denied. This error is mapped to the exception `PermissionError`.

errno.EFAULT
: Bad address

errno.ENOTBLK
Block device required

errno.EBUSY
Device or resource busy

errno.EEXIST
File exists. This error is mapped to the exception `FileExistsError`.

errno.EXDEV
Cross-device link

errno.ENODEV
No such device

errno.ENOTDIR
Not a directory. This error is mapped to the exception `NotADirectoryError`.

errno.EISDIR
Is a directory. This error is mapped to the exception `IsADirectoryError`.

errno.EINVAL
Invalid argument

errno.ENFILE
File table overflow

errno.EMFILE
Too many open files

errno.ENOTTY
Not a typewriter

errno.ETXTBSY
Text file busy

errno.EFBIG
File too large

errno.ENOSPC
No space left on device

errno.ESPIPE
Illegal seek

errno.EROFS
Read-only file system

errno.EMLINK
Too many links

errno.EPIPE
Broken pipe. This error is mapped to the exception `BrokenPipeError`.

errno.EDOM
Math argument out of domain of func

errno.ERANGE
Math result not representable

errno.EDEADLK
Resource deadlock would occur

errno.ENAMETOOLONG
File name too long

errno.ENOLCK
No record locks available

errno.ENOSYS
: Function not implemented

errno.ENOTEMPTY
: Directory not empty

errno.ELOOP
: Too many symbolic links encountered

errno.EWOULDBLOCK
: Operation would block. This error is mapped to the exception `BlockingIOError`.

errno.ENOMSG
: No message of desired type

errno.EIDRM
: Identifier removed

errno.ECHRNG
: Channel number out of range

errno.EL2NSYNC
: Level 2 not synchronized

errno.EL3HLT
: Level 3 halted

errno.EL3RST
: Level 3 reset

errno.ELNRNG
: Link number out of range

errno.EUNATCH
: Protocol driver not attached

errno.ENOCSI
No CSI structure available

errno.EL2HLT
Level 2 halted

errno.EBADE
Invalid exchange

errno.EBADR
Invalid request descriptor

errno.EXFULL
Exchange full

errno.ENOANO
No anode

errno.EBADRQC
Invalid request code

errno.EBADSLT
Invalid slot

errno.EDEADLOCK
File locking deadlock error

errno.EBFONT
Bad font file format

errno.ENOSTR
Device not a stream

errno.ENODATA
No data available

errno.ETIME
Timer expired

errno.ENOSR
Out of streams resources

errno.ENONET
Machine is not on the network

errno.ENOPKG
Package not installed

errno.EREMOTE
Object is remote

errno.ENOLINK
Link has been severed

errno.EADV
Advertise error

errno.ESRMNT
Srmount error

errno.ECOMM
Communication error on send

errno.EPROTO
Protocol error

errno.EMULTIHOP
Multihop attempted

errno.EDOTDOT
RFS specific error

errno.EBADMSG
Not a data message

errno.EOVERFLOW
Value too large for defined data type

errno.ENOTUNIQ
Name not unique on network

errno.EBADFD
File descriptor in bad state

errno.EREMCHG
Remote address changed

errno.ELIBACC
Can not access a needed shared library

errno.ELIBBAD
Accessing a corrupted shared library

errno.ELIBSCN
.lib section in a.out corrupted

errno.ELIBMAX
Attempting to link in too many shared libraries

errno.ELIBEXEC
Cannot exec a shared library directly

errno.EILSEQ
Illegal byte sequence

errno.ERESTART
Interrupted system call should be restarted

errno.ESTRPIPE
Streams pipe error

errno.EUSERS
Too many users

errno.ENOTSOCK
Socket operation on non-socket

errno.EDESTADDRREQ
Destination address required

errno.EMSGSIZE
Message too long

errno.EPROTOTYPE
Protocol wrong type for socket

errno.ENOPROTOOPT
Protocol not available

errno.EPROTONOSUPPORT
Protocol not supported

errno.ESOCKTNOSUPPORT
Socket type not supported

errno.EOPNOTSUPP
Operation not supported on transport endpoint

errno.ENOTSUP
Operation not supported

Added in version 3.2.

errno.EPFNOSUPPORT

Protocol family not supported

errno.EAFNOSUPPORT
Address family not supported by protocol

errno.EADDRINUSE
Address already in use

errno.EADDRNOTAVAIL
Cannot assign requested address

errno.ENETDOWN
Network is down

errno.ENETUNREACH
Network is unreachable

errno.ENETRESET
Network dropped connection because of reset

errno.ECONNABORTED
Software caused connection abort. This error is mapped to the exception `ConnectionAbortedError`.

errno.ECONNRESET
Connection reset by peer. This error is mapped to the exception `ConnectionResetError`.

errno.ENOBUFS
No buffer space available

errno.EISCONN
Transport endpoint is already connected

errno.ENOTCONN
Transport endpoint is not connected

errno.ESHUTDOWN

Cannot send after transport endpoint shutdown. This error is mapped to the exception `BrokenPipeError`.

errno.ETOOMANYREFS

Too many references: cannot splice

errno.ETIMEDOUT

Connection timed out. This error is mapped to the exception `TimeoutError`.

errno.ECONNREFUSED

Connection refused. This error is mapped to the exception `ConnectionRefusedError`.

errno.EHOSTDOWN

Host is down

errno.EHOSTUNREACH

No route to host

errno.EHWPOISON

Memory page has hardware error.

Added in version 3.14.

errno.EALREADY

Operation already in progress. This error is mapped to the exception `BlockingIOError`.

errno.EINPROGRESS

Operation now in progress. This error is mapped to the exception `BlockingIOError`.

errno.ESTALE

Stale NFS file handle

errno.EUCLEAN
Structure needs cleaning

errno.ENOTNAM
Not a XENIX named type file

errno.ENAVAIL
No XENIX semaphores available

errno.EISNAM
Is a named type file

errno.EREMOTEIO
Remote I/O error

errno.EDQUOT
Quota exceeded

errno.EQFULL
Interface output queue is full

Added in version 3.11.

errno.ENOMEDIUM
No medium found

errno.EMEDIUMTYPE
Wrong medium type

errno.ENOKEY
Required key not available

errno.EKEYEXPIRED
Key has expired

errno.EKEYREVOKED

Key has been revoked

errno.EKEYREJECTED
Key was rejected by service

errno.ERFKILL
Operation not possible due to RF-kill

errno.ELOCKUNMAPPED
Locked lock was unmapped

errno.ENOTACTIVE
Facility is not active

errno.EAUTH
Authentication error

Added in version 3.2.

errno.EBADARCH
Bad CPU type in executable

Added in version 3.2.

errno.EBADEXEC
Bad executable (or shared library)

Added in version 3.2.

errno.EBADMACHO
Malformed Mach-o file

Added in version 3.2.

errno.EDEVERR
Device error

Added in version 3.2.

errno.EFTYPE

Inappropriate file type or format

Added in version 3.2.

errno.ENEEDAUTH

Need authenticator

Added in version 3.2.

errno.ENOATTR

Attribute not found

Added in version 3.2.

errno.ENOPOLICY

Policy not found

Added in version 3.2.

errno.EPROCLIM

Too many processes

Added in version 3.2.

errno.EPROCUNAVAIL

Bad procedure for program

Added in version 3.2.

errno.EPROGMISMATCH

Program version wrong

Added in version 3.2.

errno.EPROGUNAVAIL

RPC prog. not avail

Added in version 3.2.

errno.EPWROFF

Device power is off

Added in version 3.2.

errno.EBADRPC

RPC struct is bad

Added in version 3.2.

errno.ERPCMISMATCH

RPC version wrong

Added in version 3.2.

errno.ESHLIBVERS

Shared library version mismatch

Added in version 3.2.

errno.ENOTCAPABLE

Capabilities insufficient. This error is mapped to the exception `PermissionError`.

Availability: WASI, FreeBSD

Added in version 3.11.1.

errno.ECANCELED

Operation canceled

Added in version 3.2.

errno.EOWNERDEAD

Owner died

Added in version 3.2.

errno.ENOTRECOVERABLE

State not recoverable

Added in version 3.2.

`ctypes` — A foreign function library for Python

Source code: Lib/ctypes [https://github.com/python/cpython/tree/3.14/Lib/ctypes]

`ctypes` is a foreign function library for Python. It provides C compatible data types, and allows calling functions in DLLs or shared libraries. It can be used to wrap these libraries in pure Python.

This is an optional module. If it is missing from your copy of CPython, look for documentation from your distributor (that is, whoever provided Python to you). If you are the distributor, see Requirements for optional modules.

ctypes tutorial

Note: The code samples in this tutorial use **`doctest`** to make sure that they actually work. Since some code samples behave differently under Linux, Windows, or macOS, they contain doctest directives in comments.

Note: Some code samples reference the ctypes **`c_int`** type. On platforms where `sizeof(long) == sizeof(int)` it is an alias to **`c_long`**. So, you should not be confused if **`c_long`** is printed if you would expect **`c_int`** — they are actually the same type.

Loading dynamic link libraries

`ctypes` exports the *cdll*, and on Windows *windll* and *oledll* objects, for loading dynamic link libraries.

You load libraries by accessing them as attributes of these objects. *cdll* loads libraries which export functions using the standard `cdecl` calling convention, while *windll* libraries call functions using the `stdcall` calling convention. *oledll* also uses the `stdcall` calling convention, and assumes the functions return a Windows **`HRESULT`** error code. The error code is used to automatically raise an `OSError` exception when the function call fails.

Changed in version 3.3: Windows errors used to raise `WindowsError`, which is now an alias of `OSError`.

Here are some examples for Windows. Note that `msvcrt` is the MS standard C library containing most standard C functions, and uses the `cdecl` calling convention:

```
>>> from ctypes import *
>>> print(windll.kernel32)
<WinDLL 'kernel32', handle ... at ...>
>>> print(cdll.msvcrt)
<CDLL 'msvcrt', handle ... at ...>
>>> libc = cdll.msvcrt
>>>
```

Windows appends the usual `.dll` file suffix automatically.

Note

Accessing the standard C library through `cdll.msvcrt` will use an outdated version of the library that may be incompatible with the one being used by Python. Where possible, use native Python functionality, or else import and use the `msvcrt` module.

On Linux, it is required to specify the filename *including* the extension to load a library, so attribute access can not be used to load libraries. Either the `LoadLibrary()` method of the dll loaders should be used, or you should load the library by creating an instance of CDLL by calling the constructor:

```
>>> cdll.LoadLibrary("libc.so.6")
<CDLL 'libc.so.6', handle ... at ...>
>>> libc = CDLL("libc.so.6")
>>> libc
<CDLL 'libc.so.6', handle ... at ...>
>>>
```

Accessing functions from loaded dlls

Functions are accessed as attributes of dll objects:

```
>>> libc.printf
<_FuncPtr object at 0x...>
>>> print(windll.kernel32.GetModuleHandleA)
<_FuncPtr object at 0x...>
>>> print(windll.kernel32.MyOwnFunction)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "ctypes.py", line 239, in __getattr__
    func = _StdcallFuncPtr(name, self)
AttributeError: function 'MyOwnFunction' not found
>>>
```

Note that win32 system dlls like `kernel32` and `user32` often export ANSI as well as UNICODE versions of a function. The UNICODE version is exported with a `W` appended to the name, while the ANSI version is exported with an `A` appended to the name. The win32 `GetModuleHandle` function, which returns a *module handle* for a given module name, has the following C prototype, and a macro is used to expose one of them as `GetModuleHandle` depending on whether UNICODE is defined or not:

```
/* ANSI version */
HMODULE GetModuleHandleA(LPCSTR lpModuleName);
/* UNICODE version */
HMODULE GetModuleHandleW(LPCWSTR lpModuleName);
```

windll does not try to select one of them by magic, you must access the version you need by specifying `GetModuleHandleA` or `GetModuleHandleW` explicitly, and then call it with bytes or string objects respectively.

Sometimes, dlls export functions with names which aren’t valid Python identifiers, like `"??2@YAPAXI@Z"`. In this case you have to use **`getattr()`** to retrieve the function:

```
>>> getattr(cdll.msvcrt, "??2@YAPAXI@Z")
<_FuncPtr object at 0x...>
>>>
```

On Windows, some dlls export functions not by name but by ordinal. These functions can be accessed by indexing the dll object with the ordinal number:

```
>>> cdll.kernel32[1]
<_FuncPtr object at 0x...>
>>> cdll.kernel32[0]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "ctypes.py", line 310, in __getitem__
    func = _StdcallFuncPtr(name, self)
AttributeError: function ordinal 0 not found
>>>
```

Calling functions

You can call these functions like any other Python callable. This example uses the `rand()` function, which takes no arguments and returns a pseudo-random integer:

```
>>> print(libc.rand())
1804289383
```

On Windows, you can call the `GetModuleHandleA()` function, which returns a win32 module handle (passing `None` as single argument to call it with a `NULL` pointer):

```
>>> print(hex(windll.kernel32.GetModuleHandleA(None)))
0x1d000000
>>>
```

`ValueError` is raised when you call an `stdcall` function with the `cdecl` calling convention, or vice versa:

```
>>> cdll.kernel32.GetModuleHandleA(None)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: Procedure probably called with not enough arguments
(4 bytes missing)
>>>

>>> windll.msvcrt.printf(b"spam")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: Procedure probably called with too many arguments
(4 bytes in excess)
>>>
```

To find out the correct calling convention you have to look into the C header file or the documentation for the function you want to call.

On Windows, **`ctypes`** uses win32 structured exception handling to prevent crashes from general protection faults when functions are called with invalid argument values:

```
>>> windll.kernel32.GetModuleHandleA(32)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
OSError: exception: access violation reading 0x00000020
>>>
```

There are, however, enough ways to crash Python with `ctypes`, so you should be careful anyway. The `faulthandler` module can be helpful in debugging crashes (e.g. from segmentation faults produced by erroneous C library calls).

`None`, integers, bytes objects and (unicode) strings are the only native Python objects that can directly be used as parameters in these function calls. `None` is passed as a C `NULL` pointer, bytes objects and strings are passed as pointer to the memory block that contains their data (char* or wchar_t*). Python integers are passed as the platform's default C int type, their value is masked to fit into the C type.

Before we move on calling functions with other parameter types, we have to learn more about `ctypes` data types.

Fundamental data types

`ctypes` defines a number of primitive C compatible data types:

ctypes type	C type	Python type
`c_bool`	_Bool	bool (1)
`c_char`	char	1-character bytes object
`c_wchar`	`wchar_t`	1-character string

ctypes type	C type	Python type
`c_byte`	char	int
`c_ubyte`	unsigned char	int
`c_short`	short	int
`c_ushort`	unsigned short	int
`c_int`	int	int
`c_int8`	`int8_t`	int
`c_int16`	`int16_t`	int
`c_int32`	`int32_t`	int
`c_int64`	`int64_t`	int
`c_uint`	unsigned int	int
`c_uint8`	`uint8_t`	int

ctypes type	C type	Python type
`c_uint16`	`uint16_t`	int
`c_uint32`	`uint32_t`	int
`c_uint64`	`uint64_t`	int
`c_long`	long	int
`c_ulong`	unsigned long	int
`c_longlong`	__int64 or long long	int
`c_ulonglong`	unsigned __int64 or unsigned long long	int
`c_size_t`	`size_t`	int
`c_ssize_t`	`ssize_t` or Py_ssize_t	int
`c_time_t`	`time_t`	int
`c_float`	float	float

ctypes type	C type	Python type
`c_double`	double	float
`c_longdouble`	long double	float
`c_char_p`	char* (NUL terminated)	bytes object or `None`
`c_wchar_p`	wchar_t* (NUL terminated)	string or `None`
`c_void_p`	void*	int or `None`

1. The constructor accepts any object with a truth value.

Additionally, if IEC 60559 compatible complex arithmetic (Annex G) is supported in both C and `libffi`, the following complex types are available:

ctypes type	C type	Python type
`c_float_complex`	float complex	complex
`c_double_complex`	double complex	complex
`c_longdouble_complex`	long double complex	complex

ctypes type	C type	Python type

All these types can be created by calling them with an optional initializer of the correct type and value:

```
>>> c_int()
c_long(0)
>>> c_wchar_p("Hello, World")
c_wchar_p(140018365411392)
>>> c_ushort(-3)
c_ushort(65533)
>>>
```

Since these types are mutable, their value can also be changed afterwards:

```
>>> i = c_int(42)
>>> print(i)
c_long(42)
>>> print(i.value)
42
>>> i.value = -99
>>> print(i.value)
-99
>>>
```

Assigning a new value to instances of the pointer types `c_char_p`, `c_wchar_p`, and `c_void_p` changes the *memory location* they point to, *not the contents* of the memory block (of course not, because Python string objects are immutable):

```
>>> s = "Hello, World"
>>> c_s = c_wchar_p(s)
>>> print(c_s)
c_wchar_p(139966785747344)
>>> print(c_s.value)
Hello World
```

```
>>> c_s.value = "Hi, there"
>>> print(c_s)              # the memory location has changed
c_wchar_p(139966783348904)
>>> print(c_s.value)
Hi, there
>>> print(s)                # first object is unchanged
Hello, World
>>>
```

You should be careful, however, not to pass them to functions expecting pointers to mutable memory. If you need mutable memory blocks, ctypes has a `create_string_buffer()` function which creates these in various ways. The current memory block contents can be accessed (or changed) with the `raw` property; if you want to access it as NUL terminated string, use the `value` property:

```
>>> from ctypes import *
>>> p = create_string_buffer(3)            # create a 3 byte
buffer, initialized to NUL bytes
>>> print(sizeof(p), repr(p.raw))
3 b'\x00\x00\x00'
>>> p = create_string_buffer(b"Hello")     # create a buffer
containing a NUL terminated string
>>> print(sizeof(p), repr(p.raw))
6 b'Hello\x00'
>>> print(repr(p.value))
b'Hello'
>>> p = create_string_buffer(b"Hello", 10) # create a 10 byte
buffer
>>> print(sizeof(p), repr(p.raw))
10 b'Hello\x00\x00\x00\x00\x00'
>>> p.value = b"Hi"
>>> print(sizeof(p), repr(p.raw))
10 b'Hi\x00lo\x00\x00\x00\x00\x00'
>>>
```

The `create_string_buffer()` function replaces the old **`c_buffer()`** function (which is still available as an alias). To create a mutable memory block containing unicode characters of the C type **`wchar_t`**, use the [`create_unicode_buffer()`] function.

Calling functions, continued

Note that printf prints to the real standard output channel, *not* to `sys.stdout`, so these examples will only work at the console prompt, not from within *IDLE* or *PythonWin*:

```
>>> printf = libc.printf
>>> printf(b"Hello, %s\n", b"World!")
Hello, World!
14
>>> printf(b"Hello, %S\n", "World!")
Hello, World!
14
>>> printf(b"%d bottles of beer\n", 42)
42 bottles of beer
19
>>> printf(b"%f bottles of beer\n", 42.5)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ctypes.ArgumentError: argument 2: TypeError: Don't know how to
convert parameter 2
>>>
```

As has been mentioned before, all Python types except integers, strings, and bytes objects have to be wrapped in their corresponding `ctypes` type, so that they can be converted to the required C data type:

```
>>> printf(b"An int %d, a double %f\n", 1234, c_double(3.14))
An int 1234, a double 3.140000
31
>>>
```

Calling variadic functions

On a lot of platforms calling variadic functions through ctypes is exactly the same as calling functions with a fixed number of parameters. On some platforms, and in particular ARM64 for Apple Platforms, the calling convention for variadic functions is different than that for regular functions.

On those platforms it is required to specify the `argtypes` attribute for the regular, non-variadic, function arguments:

```
libc.printf.argtypes = [ctypes.c_char_p]
```

Because specifying the attribute does not inhibit portability it is advised to always specify `argtypes` for all variadic functions.

Calling functions with your own custom data types

You can also customize `ctypes` argument conversion to allow instances of your own classes be used as function arguments. `ctypes` looks for an **`_as_parameter_`** attribute and uses this as the function argument. The attribute must be an integer, string, bytes, a `ctypes` instance, or an object with an **`_as_parameter_`** attribute:

```
>>> class Bottles:
...     def __init__(self, number):
...         self._as_parameter_ = number
...
>>> bottles = Bottles(42)
>>> printf(b"%d bottles of beer\n", bottles)
42 bottles of beer
19
>>>
```

If you don’t want to store the instance’s data in the **`_as_parameter_`** instance variable, you could define a `property` which makes the attribute available on request.

Specifying the required argument types (function prototypes)

It is possible to specify the required argument types of functions exported from DLLs by setting the `argtypes` attribute.

`argtypes` must be a sequence of C data types (the **`printf()`** function is probably not a good example here, because it takes a variable number and different types of parameters depending on the format string, on the other hand this is quite handy to experiment with this feature):

```
>>> printf.argtypes = [c_char_p, c_char_p, c_int, c_double]
>>> printf(b"String '%s', Int %d, Double %f\n", b"Hi", 10, 2.2)
String 'Hi', Int 10, Double 2.200000
37
>>>
```

Specifying a format protects against incompatible argument types (just as a prototype for a C function), and tries to convert the arguments to valid types:

```
>>> printf(b"%d %d %d", 1, 2, 3)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ctypes.ArgumentError: argument 2: TypeError: 'int' object
cannot be interpreted as ctypes.c_char_p
>>> printf(b"%s %d %f\n", b"X", 2, 3)
X 2 3.000000
13
>>>
```

If you have defined your own classes which you pass to function calls, you have to implement a `from_param()` class method for them to be able to use them in the `argtypes` sequence. The `from_param()` class method receives the Python object passed to the function call, it should do a typecheck or whatever is needed to make sure this object is acceptable, and then return the object itself, its **`_as_parameter_`** attribute, or whatever you want to pass as the C function argument in this case. Again, the result should be an integer, string, bytes, a **`ctypes`** instance, or an object with an **`_as_parameter_`** attribute.

Return types

By default functions are assumed to return the C int type. Other return types can be specified by setting the **`restype`** attribute of the function object.

The C prototype of **`time()`** is `time_t time(time_t *)`. Because **`time_t`** might be of a different type than the default return type int, you should specify the **`restype`** attribute:

```
>>> libc.time.restype = c_time_t
```

The argument types can be specified using **`argtypes`**:

```
>>> libc.time.argtypes = (POINTER(c_time_t),)
```

To call the function with a `NULL` pointer as first argument, use `None`:

```
>>> print(libc.time(None))
1150640792
```

Here is a more advanced example, it uses the **`strchr()`** function, which expects a string pointer and a char, and returns a pointer to a string:

```
>>> strchr = libc.strchr
>>> strchr(b"abcdef", ord("d"))
8059983
>>> strchr.restype = c_char_p    # c_char_p is a pointer to a
string
>>> strchr(b"abcdef", ord("d"))
```

```
b'def'
>>> print(strchr(b"abcdef", ord("x")))
None
>>>
```

If you want to avoid the `ord("x")` calls above, you can set the `argtypes` attribute, and the second argument will be converted from a single character Python bytes object into a C char:

```
>>> strchr.restype = c_char_p
>>> strchr.argtypes = [c_char_p, c_char]
>>> strchr(b"abcdef", b"d")
b'def'
>>> strchr(b"abcdef", b"def")
Traceback (most recent call last):
ctypes.ArgumentError: argument 2: TypeError: one character
bytes, bytearray or integer expected
>>> print(strchr(b"abcdef", b"x"))
None
>>> strchr(b"abcdef", b"d")
b'def'
>>>
```

You can also use a callable Python object (a function or a class for example) as the `restype` attribute, if the foreign function returns an integer. The callable will be called with the *integer* the C function returns, and the result of this call will be used as the result of your function call. This is useful to check for error return values and automatically raise an exception:

```
>>> GetModuleHandle = windll.kernel32.GetModuleHandleA
>>> def ValidHandle(value):
...     if value == 0:
...         raise WinError()
...     return value
...
>>>
>>> GetModuleHandle.restype = ValidHandle
>>> GetModuleHandle(None)
```

```
486539264
>>> GetModuleHandle("something silly")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 3, in ValidHandle
OSError: [Errno 126] The specified module could not be found.
>>>
```

`WinError` is a function which will call Windows `FormatMessage()` api to get the string representation of an error code, and *returns* an exception. `WinError` takes an optional error code parameter, if no one is used, it calls **`GetLastError()`** to retrieve it.

Please note that a much more powerful error checking mechanism is available through the **`errcheck`** attribute; see the reference manual for details.

Passing pointers (or: passing parameters by reference)

Sometimes a C api function expects a *pointer* to a data type as parameter, probably to write into the corresponding location, or if the data is too large to be passed by value. This is also known as *passing parameters by reference*.

`ctypes` exports the **`byref()`** function which is used to pass parameters by reference. The same effect can be achieved with the **`pointer()`** function, although **`pointer()`** does a lot more work since it constructs a real pointer object, so it is faster to use **`byref()`** if you don’t need the pointer object in Python itself:

```
>>> i = c_int()
>>> f = c_float()
>>> s = create_string_buffer(b'\000' * 32)
>>> print(i.value, f.value, repr(s.value))
0 0.0 b''
```

```
>>> libc.sscanf(b"1 3.14 Hello", b"%d %f %s",
...             byref(i), byref(f), s)
3
>>> print(i.value, f.value, repr(s.value))
1 3.1400001049 b'Hello'
>>>
```

Structures and unions

Structures and unions must derive from the `Structure` and `Union` base classes which are defined in the `ctypes` module. Each subclass must define a `_fields_` attribute. **`_fields_`** must be a list of *2-tuples*, containing a *field name* and a *field type*.

The field type must be a `ctypes` type like `c_int`, or any other derived `ctypes` type: structure, union, array, pointer.

Here is a simple example of a POINT structure, which contains two integers named *x* and *y*, and also shows how to initialize a structure in the constructor:

```
>>> from ctypes import *
>>> class POINT(Structure):
...     _fields_ = [("x", c_int),
...                 ("y", c_int)]
...
>>> point = POINT(10, 20)
>>> print(point.x, point.y)
10 20
>>> point = POINT(y=5)
>>> print(point.x, point.y)
0 5
>>> POINT(1, 2, 3)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: too many initializers
>>>
```

You can, however, build much more complicated structures. A structure can itself contain other structures by using a structure as a field type.

Here is a RECT structure which contains two POINTs named *upperleft* and *lowerright*:

```
>>> class RECT(Structure):
...     _fields_ = [("upperleft", POINT),
...                 ("lowerright", POINT)]
...
>>> rc = RECT(point)
>>> print(rc.upperleft.x, rc.upperleft.y)
0 5
>>> print(rc.lowerright.x, rc.lowerright.y)
0 0
>>>
```

Nested structures can also be initialized in the constructor in several ways:

```
>>> r = RECT(POINT(1, 2), POINT(3, 4))
>>> r = RECT((1, 2), (3, 4))
```

Field descriptors can be retrieved from the *class*, they are useful for debugging because they can provide useful information. See **`CField`**:

```
>>> POINT.x
<ctypes.CField 'x' type=c_int, ofs=0, size=4>
>>> POINT.y
<ctypes.CField 'y' type=c_int, ofs=4, size=4>
>>>
```

Warning

`ctypes` does not support passing unions or structures with bit-fields to functions by value. While this may work on 32-bit x86, it’s not guaranteed by the library to work in the general case. Unions and structures with bit-fields should always be passed to functions by pointer.

Structure/union layout, alignment and byte order

By default, Structure and Union fields are laid out in the same way the C compiler does it. It is possible to override this behavior entirely by specifying a `_layout_` class attribute in the subclass definition; see the attribute documentation for details.

It is possible to specify the maximum alignment for the fields and/or for the structure itself by setting the class attributes `_pack_` and/or `_align_`, respectively. See the attribute documentation for details.

`ctypes` uses the native byte order for Structures and Unions. To build structures with non-native byte order, you can use one of the `BigEndianStructure`, `LittleEndianStructure`, `BigEndianUnion`, and `LittleEndianUnion` base classes. These classes cannot contain pointer fields.

Bit fields in structures and unions

It is possible to create structures and unions containing bit fields. Bit fields are only possible for integer fields, the bit width is specified as the third item in the `_fields_` tuples:

```
>>> class Int(Structure):
...     _fields_ = [("first_16", c_int, 16),
...                 ("second_16", c_int, 16)]
...
>>> print(Int.first_16)
<ctypes.CField 'first_16' type=c_int, ofs=0, bit_size=16,
bit_offset=0>
```

```
>>> print(Int.second_16)
<ctypes.CField 'second_16' type=c_int, ofs=0, bit_size=16,
bit_offset=16>
```

It is important to note that bit field allocation and layout in memory are not defined as a C standard; their implementation is compiler-specific. By default, Python will attempt to match the behavior of a “native” compiler for the current platform. See the `_layout_` attribute for details on the default behavior and how to change it.

Arrays

Arrays are sequences, containing a fixed number of instances of the same type.

The recommended way to create array types is by multiplying a data type with a positive integer:

```
TenPointsArrayType = POINT * 10
```

Here is an example of a somewhat artificial data type, a structure containing 4 POINTs among other stuff:

```
>>> from ctypes import *
>>> class POINT(Structure):
...     _fields_ = ("x", c_int), ("y", c_int)
...
>>> class MyStruct(Structure):
...     _fields_ = [("a", c_int),
...                 ("b", c_float),
...                 ("point_array", POINT * 4)]
>>>
>>> print(len(MyStruct().point_array))
4
>>>
```

Instances are created in the usual way, by calling the class:

```
arr = TenPointsArrayType()
for pt in arr:
    print(pt.x, pt.y)
```

The above code print a series of `0 0` lines, because the array contents is initialized to zeros.

Initializers of the correct type can also be specified:

```
>>> from ctypes import *
>>> TenIntegers = c_int * 10
>>> ii = TenIntegers(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)
>>> print(ii)
<c_long_Array_10 object at 0x...>
>>> for i in ii: print(i, end=" ")
...
1 2 3 4 5 6 7 8 9 10
>>>
```

Pointers

Pointer instances are created by calling the **`pointer()`** function on a **`ctypes`** type:

```
>>> from ctypes import *
>>> i = c_int(42)
>>> pi = pointer(i)
>>>
```

Pointer instances have a **`contents`** attribute which returns the object to which the pointer points, the `i` object above:

```
>>> pi.contents
c_long(42)
>>>
```

Note that **`ctypes`** does not have OOR (original object return), it constructs a new, equivalent object each time you retrieve an attribute:

```
>>> pi.contents is i
False
>>> pi.contents is pi.contents
False
>>>
```

Assigning another `c_int` instance to the pointer’s contents attribute would cause the pointer to point to the memory location where this is stored:

```
>>> i = c_int(99)
>>> pi.contents = i
>>> pi.contents
c_long(99)
>>>
```

Pointer instances can also be indexed with integers:

```
>>> pi[0]
99
>>>
```

Assigning to an integer index changes the pointed to value:

```
>>> print(i)
c_long(99)
>>> pi[0] = 22
>>> print(i)
c_long(22)
>>>
```

It is also possible to use indexes different from 0, but you must know what you’re doing, just as in C: You can access or change arbitrary memory locations. Generally you only use this feature if you receive a pointer from a C function, and you *know* that the pointer actually points to an array instead of a single item.

Behind the scenes, the `pointer()` function does more than simply create pointer instances, it has to create pointer *types* first. This is done with the **`POINTER()`** function, which accepts any **`ctypes`** type, and returns a new type:

```
>>> PI = POINTER(c_int)
>>> PI
<class 'ctypes.LP_c_long'>
>>> PI(42)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: expected c_long instead of int
>>> PI(c_int(42))
<ctypes.LP_c_long object at 0x...>
>>>
```

Calling the pointer type without an argument creates a `NULL` pointer. `NULL` pointers have a `False` boolean value:

```
>>> null_ptr = POINTER(c_int)()
>>> print(bool(null_ptr))
False
>>>
```

`ctypes` checks for `NULL` when dereferencing pointers (but dereferencing invalid non-`NULL` pointers would crash Python):

```
>>> null_ptr[0]
Traceback (most recent call last):
    ....
ValueError: NULL pointer access
>>>

>>> null_ptr[0] = 1234
Traceback (most recent call last):
    ....
ValueError: NULL pointer access
>>>
```

Thread safety without the GIL

From Python 3.13 onward, the GIL can be disabled on free threaded builds. In ctypes, reads and writes to a single object concurrently is safe, but not across multiple objects:

```
>>> number = c_int(42)
>>> pointer_a = pointer(number)
>>> pointer_b = pointer(number)
```

In the above, it’s only safe for one object to read and write to the address at once if the GIL is disabled. So, `pointer_a` can be shared and written to across multiple threads, but only if `pointer_b` is not also attempting to do the same. If this is an issue, consider using a **`threading.Lock`** to synchronize access to memory:

```
>>> import threading
>>> lock = threading.Lock()
>>> # Thread 1
>>> with lock:
...    pointer_a.contents = 24
>>> # Thread 2
>>> with lock:
...    pointer_b.contents = 42
```

Type conversions

Usually, ctypes does strict type checking. This means, if you have `POINTER(c_int)` in the **`argtypes`** list of a function or as the type of a member field in a structure definition, only instances of exactly the same type are accepted. There are some exceptions to this rule, where ctypes accepts other objects. For example, you can pass compatible array instances instead of pointer types. So, for `POINTER(c_int)`, ctypes accepts an array of c_int:

```
>>> class Bar(Structure):
...     _fields_ = [("count", c_int), ("values",
POINTER(c_int))]
...
>>> bar = Bar()
>>> bar.values = (c_int * 3)(1, 2, 3)
>>> bar.count = 3
>>> for i in range(bar.count):
...     print(bar.values[i])
...
1
2
3
>>>
```

In addition, if a function argument is explicitly declared to be a pointer type (such as `POINTER(c_int)`) in **`argtypes`**, an object of the pointed type (`c_int` in this case) can be passed to the function. ctypes will apply the required **`byref()`** conversion in this case automatically.

To set a POINTER type field to `NULL`, you can assign `None`:

```
>>> bar.values = None
>>>
```

Sometimes you have instances of incompatible types. In C, you can cast one type into another type. **`ctypes`** provides a **`cast()`** function which can be used in the same way. The `Bar` structure defined above accepts `POINTER(c_int)` pointers or **`c_int`** arrays for its `values` field, but not instances of other types:

```
>>> bar.values = (c_byte * 4)()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: incompatible types, c_byte_Array_4 instance instead
of LP_c_long instance
>>>
```

For these cases, the **`cast()`** function is handy.

The **`cast()`** function can be used to cast a ctypes instance into a pointer to a different ctypes data type. **`cast()`** takes two parameters, a ctypes object that is or can be converted to a pointer of some kind, and a ctypes pointer type. It returns an instance of the second argument, which references the same memory block as the first argument:

```
>>> a = (c_byte * 4)()
>>> cast(a, POINTER(c_int))
<ctypes.LP_c_long object at ...>
>>>
```

So, `cast()` can be used to assign to the `values` field of `Bar` the structure:

```
>>> bar = Bar()
>>> bar.values = cast((c_byte * 4)(), POINTER(c_int))
>>> print(bar.values[0])
0
>>>
```

Incomplete Types

Incomplete Types are structures, unions or arrays whose members are not yet specified. In C, they are specified by forward declarations, which are defined later:

```
struct cell; /* forward declaration */

struct cell {
    char *name;
    struct cell *next;
};
```

The straightforward translation into ctypes code would be this, but it does not work:

```
>>> class cell(Structure):
...     _fields_ = [("name", c_char_p),
...                 ("next", POINTER(cell))]
...
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 2, in cell
NameError: name 'cell' is not defined
>>>
```

because the new `class cell` is not available in the class statement itself. In **`ctypes`**, we can define the `cell` class and set the **`_fields_`** attribute later, after the class statement:

```
>>> from ctypes import *
>>> class cell(Structure):
...     pass
...
>>> cell._fields_ = [("name", c_char_p),
...                  ("next", POINTER(cell))]
>>>
```

Let’s try it. We create two instances of `cell`, and let them point to each other, and finally follow the pointer chain a few times:

```
>>> c1 = cell()
>>> c1.name = b"foo"
>>> c2 = cell()
>>> c2.name = b"bar"
>>> c1.next = pointer(c2)
>>> c2.next = pointer(c1)
>>> p = c1
>>> for i in range(8):
...     print(p.name, end=" ")
...     p = p.next[0]
...
foo bar foo bar foo bar foo bar
>>>
```

Callback functions

`ctypes` allows creating C callable function pointers from Python callables. These are sometimes called *callback functions*.

First, you must create a class for the callback function. The class knows the calling convention, the return type, and the number and types of arguments this function will receive.

The **`CFUNCTYPE()`** factory function creates types for callback functions using the `cdecl` calling convention. On Windows, the **`WINFUNCTYPE()`** factory function creates types for callback functions using the `stdcall` calling convention.

Both of these factory functions are called with the result type as first argument, and the callback functions expected argument types as the remaining arguments.

I will present an example here which uses the standard C library’s **`qsort()`** function, that is used to sort items with the help of a callback function. **`qsort()`** will be used to sort an array of integers:

```
>>> IntArray5 = c_int * 5
>>> ia = IntArray5(5, 1, 7, 33, 99)
>>> qsort = libc.qsort
>>> qsort.restype = None
>>>
```

`qsort()` must be called with a pointer to the data to sort, the number of items in the data array, the size of one item, and a pointer to the comparison function, the callback. The callback will then be called with two pointers to items, and it must return a negative integer if the first item is smaller than the second, a zero if they are equal, and a positive integer otherwise.

So our callback function receives pointers to integers, and must return an integer. First we create the `type` for the callback function:

```
>>> CMPFUNC = CFUNCTYPE(c_int, POINTER(c_int), POINTER(c_int))
>>>
```

To get started, here is a simple callback that shows the values it gets passed:

```
>>> def py_cmp_func(a, b):
...     print("py_cmp_func", a[0], b[0])
...     return 0
...
>>> cmp_func = CMPFUNC(py_cmp_func)
>>>
```

The result:

```
>>> qsort(ia, len(ia), sizeof(c_int), cmp_func)
py_cmp_func 5 1
py_cmp_func 33 99
py_cmp_func 7 33
py_cmp_func 5 7
py_cmp_func 1 7
>>>
```

Now we can actually compare the two items and return a useful result:

```
>>> def py_cmp_func(a, b):
...     print("py_cmp_func", a[0], b[0])
...     return a[0] - b[0]
...
>>>
>>> qsort(ia, len(ia), sizeof(c_int), CMPFUNC(py_cmp_func))
py_cmp_func 5 1
py_cmp_func 33 99
py_cmp_func 7 33
py_cmp_func 1 7
py_cmp_func 5 7
>>>
```

As we can easily check, our array is sorted now:

```
>>> for i in ia: print(i, end=" ")
...
1 5 7 33 99
>>>
```

The function factories can be used as decorator factories, so we may as well write:

```
>>> @CFUNCTYPE(c_int, POINTER(c_int), POINTER(c_int))
... def py_cmp_func(a, b):
...     print("py_cmp_func", a[0], b[0])
...     return a[0] - b[0]
...
>>> qsort(ia, len(ia), sizeof(c_int), py_cmp_func)
py_cmp_func 5 1
py_cmp_func 33 99
py_cmp_func 7 33
py_cmp_func 1 7
py_cmp_func 5 7
>>>
```

Note

Make sure you keep references to `CFUNCTYPE()` objects as long as they are used from C code. `ctypes` doesn’t, and if you don’t, they may be garbage collected, crashing your program when a callback is made.

Also, note that if the callback function is called in a thread created outside of Python’s control (e.g. by the foreign code that calls the callback), ctypes creates a new dummy Python thread on every invocation. This behavior is correct for most purposes, but it means that values stored with `threading.local` will *not* survive across different callbacks, even when those calls are made from the same C thread.

Accessing values exported from dlls

Some shared libraries not only export functions, they also export variables. An example in the Python library itself is the `Py_Version`, Python runtime version number encoded in a single constant integer.

`ctypes` can access values like this with the **`in_dll()`** class methods of the type. *pythonapi* is a predefined symbol giving access to the Python C api:

```
>>> version = ctypes.c_int.in_dll(ctypes.pythonapi,
"Py_Version")
>>> print(hex(version.value))
0x30c00a0
```

An extended example which also demonstrates the use of pointers accesses the **`PyImport_FrozenModules`** pointer exported by Python.

Quoting the docs for that value:

> This pointer is initialized to point to an array of **`_frozen`** records, terminated by one whose members are all `NULL` or zero. When a frozen module is imported, it is searched in this table. Third-party code could play tricks with this to provide a dynamically created collection of frozen modules.

So manipulating this pointer could even prove useful. To restrict the example size, we show only how this table can be read with **`ctypes`**:

```
>>> from ctypes import *
>>>
>>> class struct_frozen(Structure):
...     _fields_ = [("name", c_char_p),
...                 ("code", POINTER(c_ubyte)),
...                 ("size", c_int),
...                 ("get_code", POINTER(c_ubyte)),  # Function
pointer
...                ]
...
>>>
```

We have defined the `_frozen` data type, so we can get the pointer to the table:

```
>>> FrozenTable = POINTER(struct_frozen)
>>> table = FrozenTable.in_dll(pythonapi,
"_PyImport_FrozenBootstrap")
>>>
```

Since `table` is a `pointer` to the array of `struct_frozen` records, we can iterate over it, but we just have to make sure that our loop terminates, because pointers have no size. Sooner or later it would probably crash with an access violation or whatever, so it's better to break out of the loop when we hit the `NULL` entry:

```
>>> for item in table:
...     if item.name is None:
...         break
...     print(item.name.decode("ascii"), item.size)
...
_frozen_importlib 31764
_frozen_importlib_external 41499
zipimport 12345
>>>
```

The fact that standard Python has a frozen module and a frozen package (indicated by the negative `size` member) is not well known, it is only used for testing. Try it out with `import __hello__` for example.

Surprises

There are some edges in `ctypes` where you might expect something other than what actually happens.

Consider the following example:

```
>>> from ctypes import *
>>> class POINT(Structure):
```

```
...     _fields_ = ("x", c_int), ("y", c_int)
...
>>> class RECT(Structure):
...     _fields_ = ("a", POINT), ("b", POINT)
...
>>> p1 = POINT(1, 2)
>>> p2 = POINT(3, 4)
>>> rc = RECT(p1, p2)
>>> print(rc.a.x, rc.a.y, rc.b.x, rc.b.y)
1 2 3 4
>>> # now swap the two points
>>> rc.a, rc.b = rc.b, rc.a
>>> print(rc.a.x, rc.a.y, rc.b.x, rc.b.y)
3 4 3 4
>>>
```

Hm. We certainly expected the last statement to print `3 4 1 2`. What happened? Here are the steps of the `rc.a, rc.b = rc.b, rc.a` line above:

```
>>> temp0, temp1 = rc.b, rc.a
>>> rc.a = temp0
>>> rc.b = temp1
>>>
```

Note that `temp0` and `temp1` are objects still using the internal buffer of the `rc` object above. So executing `rc.a = temp0` copies the buffer contents of `temp0` into `rc` ‘s buffer. This, in turn, changes the contents of `temp1`. So, the last assignment `rc.b = temp1`, doesn’t have the expected effect.

Keep in mind that retrieving sub-objects from Structure, Unions, and Arrays doesn’t *copy* the sub-object, instead it retrieves a wrapper object accessing the root-object’s underlying buffer.

Another example that may behave differently from what one would expect is this:

```
>>> s = c_char_p()
>>> s.value = b"abc def ghi"
>>> s.value
b'abc def ghi'
>>> s.value is s.value
False
>>>
```

Note

Objects instantiated from [`c_char_p`] can only have their value set to bytes or integers.

Why is it printing `False`? ctypes instances are objects containing a memory block plus some descriptors accessing the contents of the memory. Storing a Python object in the memory block does not store the object itself, instead the `contents` of the object is stored. Accessing the contents again constructs a new Python object each time!

Variable-sized data types

`ctypes` provides some support for variable-sized arrays and structures.

The **`resize()`** function can be used to resize the memory buffer of an existing ctypes object. The function takes the object as first argument, and the requested size in bytes as the second argument. The memory block cannot be made smaller than the natural memory block specified by the objects type, a **`ValueError`** is raised if this is tried:

```
>>> short_array = (c_short * 4)()
>>> print(sizeof(short_array))
8
>>> resize(short_array, 4)
Traceback (most recent call last):
    ...
ValueError: minimum size is 8
>>> resize(short_array, 32)
>>> sizeof(short_array)
32
>>> sizeof(type(short_array))
8
>>>
```

This is nice and fine, but how would one access the additional elements contained in this array? Since the type still only knows about 4 elements, we get errors accessing other elements:

```
>>> short_array[:]
[0, 0, 0, 0]
>>> short_array[7]
Traceback (most recent call last):
    ...
IndexError: invalid index
>>>
```

Another way to use variable-sized data types with `ctypes` is to use the dynamic nature of Python, and (re-)define the data type after the required size is already known, on a case by case basis.

ctypes reference

Finding shared libraries

When programming in a compiled language, shared libraries are accessed when compiling/linking a program, and when the program is run.

The purpose of the `find_library()` function is to locate a library in a way similar to what the compiler or runtime loader does (on platforms with several versions of a shared library the most recent should be loaded), while the ctypes library loaders act like when a program is run, and call the runtime loader directly.

The **`ctypes.util`** module provides a function which can help to determine the library to load.

ctypes.util.find_library(*name*)

> Try to find a library and return a pathname. *name* is the library name without any prefix like *lib*, suffix like `.so`, `.dylib` or version number (this is the form used for the posix linker option **`-l`**). If no library can be found, returns `None`.

The exact functionality is system dependent.

On Linux, **`find_library()`** tries to run external programs (`/sbin/ldconfig`, `gcc`, `objdump` and `ld`) to find the library file. It returns the filename of the library file.

Note that if the output of these programs does not correspond to the dynamic linker used by Python, the result of this function may be misleading.

Changed in version 3.6: On Linux, the value of the environment variable `LD_LIBRARY_PATH` is used when searching for libraries, if a library cannot be found by any other means.

Here are some examples:

```
>>> from ctypes.util import find_library
>>> find_library("m")
'libm.so.6'
>>> find_library("c")
'libc.so.6'
>>> find_library("bz2")
```

```
'libbz2.so.1.0'
>>>
```

On macOS and Android, `find_library()` uses the system's standard naming schemes and paths to locate the library, and returns a full pathname if successful:

```
>>> from ctypes.util import find_library
>>> find_library("c")
'/usr/lib/libc.dylib'
>>> find_library("m")
'/usr/lib/libm.dylib'
>>> find_library("bz2")
'/usr/lib/libbz2.dylib'
>>> find_library("AGL")
'/System/Library/Frameworks/AGL.framework/AGL'
>>>
```

On Windows, `find_library()` searches along the system search path, and returns the full pathname, but since there is no predefined naming scheme a call like `find_library("c")` will fail and return `None`.

If wrapping a shared library with `ctypes`, it *may* be better to determine the shared library name at development time, and hardcode that into the wrapper module instead of using `find_library()` to locate the library at runtime.

Listing loaded shared libraries

When writing code that relies on code loaded from shared libraries, it can be useful to know which shared libraries have already been loaded into the current process.

The **`ctypes.util`** module provides the `dllist()` function, which calls the different APIs provided by the various platforms to help determine which shared libraries have already been loaded into the current process.

The exact output of this function will be system dependent. On most platforms, the first entry of this list represents the current process itself, which may be an empty string. For example, on glibc-based Linux, the return may look like:

```
>>> from ctypes.util import dllist
>>> dllist()
['', 'linux-vdso.so.1', '/lib/x86_64-linux-gnu/libm.so.6',
'/lib/x86_64-linux-gnu/libc.so.6', ... ]
```

Loading shared libraries

There are several ways to load shared libraries into the Python process. One way is to instantiate one of the following classes:

class ctypes.CDLL(*name*, *mode=DEFAULT_MODE*, *handle=None*, *use_errno=False*, *use_last_error=False*, *winmode=None*)

Instances of this class represent loaded shared libraries. Functions in these libraries use the standard C calling convention, and are assumed to return int.

On Windows creating a `CDLL` instance may fail even if the DLL name exists. When a dependent DLL of the loaded DLL is not found, a `OSError` error is raised with the message *“[WinError 126] The specified module could not be found”.* This error message does not contain the name of the missing DLL because the Windows API does not return this information making this error hard to diagnose. To resolve this error and determine which DLL is not found, you need to find the list of dependent DLLs and determine which one is not found using Windows debugging and tracing tools.

Changed in version 3.12: The *name* parameter can now be a path-like object.

See also

Microsoft DUMPBIN tool [https://docs.microsoft.com/cpp/build/reference/dependents] – A tool to find DLL dependents.

class ctypes.OleDLL(*name*, *mode=DEFAULT_MODE*, *handle=None*, *use_errno=False*, *use_last_error=False*, *winmode=None*)

Instances of this class represent loaded shared libraries, functions in these libraries use the `stdcall` calling convention, and are assumed to return the windows specific **`HRESULT`** code. **`HRESULT`** values contain information specifying whether the function call failed or succeeded, together with additional error code. If the return value signals a failure, an **`OSError`** is automatically raised.

Availability: Windows

Changed in version 3.3: **`WindowsError`** used to be raised, which is now an alias of **`OSError`**.

Changed in version 3.12: The *name* parameter can now be a path-like object.

class ctypes.WinDLL(*name*, *mode=DEFAULT_MODE*, *handle=None*, *use_errno=False*, *use_last_error=False*, *winmode=None*)

Instances of this class represent loaded shared libraries, functions in these libraries use the `stdcall` calling convention, and are assumed to return int by default.

Availability: Windows

Changed in version 3.12: The *name* parameter can now be a path-like object.

The Python global interpreter lock is released before calling any function exported by these libraries, and reacquired afterwards.

class ctypes.PyDLL(*name*, *mode=DEFAULT_MODE*, *handle=None*)

Instances of this class behave like **`CDLL`** instances, except that the Python GIL is *not* released during the function call, and after the function execution the Python error flag is checked. If the error flag is set, a Python exception is raised.

Thus, this is only useful to call Python C api functions directly.

Changed in version 3.12: The *name* parameter can now be a path-like object.

All these classes can be instantiated by calling them with at least one argument, the pathname of the shared library. If you have an existing handle to an already loaded shared library, it can be passed as the `handle` named parameter, otherwise the underlying platform’s **`dlopen()`** or **`LoadLibrary()`** function is used to load the library into the process, and to get a handle to it.

The *mode* parameter can be used to specify how the library is loaded. For details, consult the *dlopen(3)* *[https://manpages.debian.org/dlopen(3)]* manpage. On Windows, *mode* is ignored. On posix systems, RTLD_NOW is always added, and is not configurable.

The *use_errno* parameter, when set to true, enables a ctypes mechanism that allows accessing the system **`errno`** error number in a safe way. **`ctypes`** maintains a thread-local copy of the system's **`errno`** variable; if you call foreign functions created with `use_errno=True` then the **`errno`** value before the function call is swapped with the ctypes private copy, the same happens immediately after the function call.

The function **`ctypes.get_errno()`** returns the value of the ctypes private copy, and the function **`ctypes.set_errno()`** changes the ctypes private copy to a new value and returns the former value.

The *use_last_error* parameter, when set to true, enables the same mechanism for the Windows error code which is managed by the **`GetLastError()`** and **`SetLastError()`** Windows API functions; **`ctypes.get_last_error()`** and **`ctypes.set_last_error()`** are used to request and change the ctypes private copy of the windows error code.

The *winmode* parameter is used on Windows to specify how the library is loaded (since *mode* is ignored). It takes any value that is valid for the Win32 API `LoadLibraryEx` flags parameter. When omitted, the default is to use the flags that result in the most secure DLL load, which avoids issues such as DLL hijacking. Passing the full path to the DLL is the safest way to ensure the correct library and dependencies are loaded.

Changed in version 3.8: Added *winmode* parameter.

ctypes.RTLD_GLOBAL

Flag to use as *mode* parameter. On platforms where this flag is not available, it is defined as the integer zero.

ctypes.RTLD_LOCAL

Flag to use as *mode* parameter. On platforms where this is not available, it is the same as *RTLD_GLOBAL*.

ctypes.DEFAULT_MODE

The default mode which is used to load shared libraries. On OSX 10.3, this is *RTLD_GLOBAL*, otherwise it is the same as *RTLD_LOCAL*.

Instances of these classes have no public methods. Functions exported by the shared library can be accessed as attributes or by index. Please note that accessing the function through an attribute caches the result and therefore accessing it repeatedly returns the same object each time. On the other hand, accessing it through an index returns a new object each time:

```
>>> from ctypes import CDLL
>>> libc = CDLL("libc.so.6")  # On Linux
>>> libc.time == libc.time
True
>>> libc['time'] == libc['time']
False
```

The following public attributes are available, their name starts with an underscore to not clash with exported function names:

PyDLL._handle

The system handle used to access the library.

PyDLL._name

The name of the library passed in the constructor.

Shared libraries can also be loaded by using one of the prefabricated objects, which are instances of the `LibraryLoader` class, either by calling the `LoadLibrary()`

method, or by retrieving the library as attribute of the loader instance.

class ctypes.LibraryLoader(*dlltype*)

Class which loads shared libraries. *dlltype* should be one of the **`CDLL`**, **`PyDLL`**, **`WinDLL`**, or **`OleDLL`** types.

`__getattr__()` has special behavior: It allows loading a shared library by accessing it as attribute of a library loader instance. The result is cached, so repeated attribute accesses return the same library each time.

LoadLibrary(*name*)

Load a shared library into the process and return it. This method always returns a new instance of the library.

These prefabricated library loaders are available:

ctypes.cdll

Creates **`CDLL`** instances.

ctypes.windll

Creates **`WinDLL`** instances.

Availability: Windows

ctypes.oledll

Creates **`OleDLL`** instances.

Availability: Windows

ctypes.pydll

Creates **`PyDLL`** instances.

For accessing the C Python api directly, a ready-to-use Python shared library object is available:

ctypes.pythonapi

> An instance of **`PyDLL`** that exposes Python C API functions as attributes. Note that all these functions are assumed to return C int, which is of course not always the truth, so you have to assign the correct **`restype`** attribute to use these functions.

Loading a library through any of these objects raises an auditing event `ctypes.dlopen` with string argument `name`, the name used to load the library.

Accessing a function on a loaded library raises an auditing event `ctypes.dlsym` with arguments `library` (the library object) and `name` (the symbol’s name as a string or integer).

In cases when only the library handle is available rather than the object, accessing a function raises an auditing event `ctypes.dlsym/handle` with arguments `handle` (the raw library handle) and `name`.

Foreign functions

As explained in the previous section, foreign functions can be accessed as attributes of loaded shared libraries. The function objects created in this way by default accept any number of arguments, accept any ctypes data instances as arguments, and return the default result type specified by the library loader.

They are instances of a private local class **`_FuncPtr`** (not exposed in **`ctypes`**) which inherits from the private **`_CFuncPtr`** class:

```
>>> import ctypes
>>> lib = ctypes.CDLL(None)
>>> issubclass(lib._FuncPtr, ctypes._CFuncPtr)
True
>>> lib._FuncPtr is ctypes._CFuncPtr
False
```

class ctypes._CFuncPtr

Base class for C callable foreign functions.

Instances of foreign functions are also C compatible data types; they represent C function pointers.

This behavior can be customized by assigning to special attributes of the foreign function object.

restype

Assign a ctypes type to specify the result type of the foreign function. Use `None` for void, a function not returning anything.

It is possible to assign a callable Python object that is not a ctypes type, in this case the function is assumed to return a C int, and the callable will be called with this integer, allowing further processing or error checking. Using this is deprecated, for more flexible post processing or error checking use a ctypes data type as **`restype`** and assign a callable to the **`errcheck`** attribute.

argtypes

Assign a tuple of ctypes types to specify the argument types that the function accepts. Functions using the `stdcall` calling convention can only be called with the same number of arguments as the length of this tuple; functions using the C calling convention accept additional, unspecified arguments as well.

When a foreign function is called, each actual argument is passed to the `from_param()` class method of the items in the `argtypes` tuple, this method allows adapting the actual argument to an object that the foreign function accepts. For example, a `c_char_p` item in the `argtypes` tuple will convert a string passed as argument into a bytes object using ctypes conversion rules.

New: It is now possible to put items in argtypes which are not ctypes types, but each item must have a `from_param()` method which returns a value usable as argument (integer, string, ctypes instance). This allows defining adapters that can adapt custom objects as function parameters.

errcheck

Assign a Python function or another callable to this attribute. The callable will be called with three or more arguments:

callable(*result*, *func*, *arguments*)

result is what the foreign function returns, as specified by the **`restype`** attribute.

func is the foreign function object itself, this allows reusing the same callable object to check or post process the results of several functions.

arguments is a tuple containing the parameters originally passed to the function call, this allows specializing the behavior on the arguments used.

The object that this function returns will be returned from the foreign function call, but it can also check the result value and raise an exception if the foreign function call failed.

On Windows, when a foreign function call raises a system exception (for example, due to an access violation), it will be captured and replaced with a suitable Python exception. Further, an auditing event `ctypes.set_exception` with argument `code` will be raised, allowing an audit hook to replace the exception with its own.

Some ways to invoke foreign function calls as well as some of the functions in this module may raise an auditing event `ctypes.call_function` with arguments `function pointer` and `arguments`.

Function prototypes

Foreign functions can also be created by instantiating function prototypes. Function prototypes are similar to function prototypes in C; they describe a function (return type, argument types, calling convention) without defining an implementation. The factory functions must be called with the desired result type and the argument types of the function, and can be used as decorator factories, and as such, be applied to functions through the `@wrapper` syntax. See Callback functions for examples.

ctypes.CFUNCTYPE(*restype*, **argtypes*, *use_errno=False*, *use_last_error=False*)

The returned function prototype creates functions that use the standard C calling convention. The function will release the GIL during the call. If *use_errno* is set to true, the ctypes private copy of the system **`errno`** variable is exchanged with the real **`errno`** value before and after the call; *use_last_error* does the same for the Windows error code.

ctypes.WINFUNCTYPE(*restype*, **argtypes*, *use_errno=False*, *use_last_error=False*)

The returned function prototype creates functions that use the `stdcall` calling convention. The function will release the GIL during the call. *use_errno* and *use_last_error* have the same meaning as above.

Availability: Windows

ctypes.PYFUNCTYPE(*restype*, **argtypes*)

The returned function prototype creates functions that use the Python calling convention. The function will *not* release the GIL during the call.

Function prototypes created by these factory functions can be instantiated in different ways, depending on the type and number of the parameters in the call:

prototype(*address*)

Returns a foreign function at the specified address which must be an integer.

prototype(*callable*)

Create a C callable function (a callback function) from a Python *callable*.

prototype(*func_spec*[, *paramflags*])

Returns a foreign function exported by a shared library. *func_spec* must be a 2-tuple `(name_or_ordinal, library)`. The first item is the name of the exported function as string, or the ordinal of the exported function as small integer. The second item is the shared library instance.

prototype(*vtbl_index*, *name*[, *paramflags*[, *iid*]])

Returns a foreign function that will call a COM method. *vtbl_index* is the index into the virtual function table, a small non-negative integer. *name* is name of the COM method. *iid* is an optional pointer to the interface identifier which is used in extended error reporting.

If *iid* is not specified, an `OSError` is raised if the COM method call fails. If *iid* is specified, a `COMError` is raised instead.

COM methods use a special calling convention: They require a pointer to the COM interface as first argument, in addition to those parameters that are specified in the **`argtypes`** tuple.

Availability: Windows

The optional *paramflags* parameter creates foreign function wrappers with much more functionality than the features described above.

paramflags must be a tuple of the same length as `argtypes`.

Each item in this tuple contains further information about a parameter, it must be a tuple containing one, two, or three items.

The first item is an integer containing a combination of direction flags for the parameter:

1
: Specifies an input parameter to the function.

2
: Output parameter. The foreign function fills in a value.

4
: Input parameter which defaults to the integer zero.

The optional second item is the parameter name as string. If this is specified, the foreign function can be called with named parameters.

The optional third item is the default value for this parameter.

The following example demonstrates how to wrap the Windows `MessageBoxW` function so that it supports default parameters and named arguments. The C declaration from the windows header file is this:

```
WINUSERAPI int WINAPI
MessageBoxW(
    HWND hWnd,
    LPCWSTR lpText,
    LPCWSTR lpCaption,
    UINT uType);
```

Here is the wrapping with **`ctypes`**:

```
>>> from ctypes import c_int, WINFUNCTYPE, windll
>>> from ctypes.wintypes import HWND, LPCWSTR, UINT
>>> prototype = WINFUNCTYPE(c_int, HWND, LPCWSTR, LPCWSTR,
UINT)
>>> paramflags = (1, "hwnd", 0), (1, "text", "Hi"), (1,
"caption", "Hello from ctypes"), (1, "flags", 0)
>>> MessageBox = prototype(("MessageBoxW", windll.user32),
paramflags)
```

The `MessageBox` foreign function can now be called in these ways:

```
>>> MessageBox()
>>> MessageBox(text="Spam, spam, spam")
>>> MessageBox(flags=2, text="foo bar")
```

A second example demonstrates output parameters. The win32 `GetWindowRect` function retrieves the dimensions of a specified window by copying them into `RECT` structure that the caller has to supply. Here is the C declaration:

```
WINUSERAPI BOOL WINAPI
GetWindowRect(
     HWND hWnd,
     LPRECT lpRect);
```

Here is the wrapping with **`ctypes`**:

```
>>> from ctypes import POINTER, WINFUNCTYPE, windll, WinError
>>> from ctypes.wintypes import BOOL, HWND, RECT
>>> prototype = WINFUNCTYPE(BOOL, HWND, POINTER(RECT))
>>> paramflags = (1, "hwnd"), (2, "lprect")
>>> GetWindowRect = prototype(("GetWindowRect", windll.user32),
paramflags)
>>>
```

Functions with output parameters will automatically return the output parameter value if there is a single one, or a tuple containing the output parameter values when there are more than one, so the GetWindowRect function now returns a RECT instance, when called.

Output parameters can be combined with the **`errcheck`** protocol to do further output processing and error checking. The win32 `GetWindowRect` api function returns a `BOOL` to signal success or failure, so this function could do the error checking, and raises an exception when the api call failed:

```
>>> def errcheck(result, func, args):
...     if not result:
...         raise WinError()
...     return args
...
>>> GetWindowRect.errcheck = errcheck
>>>
```

If the [`errcheck`]() function returns the argument tuple it receives unchanged, [`ctypes`]() continues the normal processing it does on the output parameters. If you want to return a tuple of window coordinates instead of a `RECT` instance, you can retrieve the fields in the function and return them instead, the normal processing will no longer take place:

```
>>> def errcheck(result, func, args):
...     if not result:
...         raise WinError()
...     rc = args[1]
...     return rc.left, rc.top, rc.bottom, rc.right
...
>>> GetWindowRect.errcheck = errcheck
>>>
```

Utility functions

ctypes.addressof(*obj*)

Returns the address of the memory buffer as integer. *obj* must be an instance of a ctypes type.

Raises an auditing event `ctypes.addressof` with argument `obj`.

ctypes.alignment(*obj_or_type*)

Returns the alignment requirements of a ctypes type. *obj_or_type* must be a ctypes type or instance.

ctypes.byref(*obj*[, *offset*])

Returns a light-weight pointer to *obj*, which must be an instance of a ctypes type. *offset* defaults to zero, and must be an integer that will be added to the internal pointer value.

`byref(obj, offset)` corresponds to this C code:

`(((char *)&obj) + offset)`

The returned object can only be used as a foreign function call parameter. It behaves similar to `pointer(obj)`, but the construction is a lot faster.

ctypes.CopyComPointer(*src*, *dst*)

Copies a COM pointer from *src* to *dst* and returns the Windows specific **`HRESULT`** value.

If *src* is not `NULL`, its `AddRef` method is called, incrementing the reference count.

In contrast, the reference count of *dst* will not be decremented before assigning the new value. Unless *dst* is `NULL`, the caller is responsible for decrementing the reference count by calling its `Release` method when necessary.

Availability: Windows

Added in version 3.14.

ctypes.cast(*obj*, *type*)

This function is similar to the cast operator in C. It returns a new instance of *type* which points to the same memory block as *obj*. *type* must be a pointer type, and *obj* must be an object that can be interpreted as a pointer.

ctypes.create_string_buffer(*init*, *size=None*)
ctypes.create_string_buffer(*size*)

This function creates a mutable character buffer. The returned object is a ctypes array of **`c_char`**.

If *size* is given (and not `None`), it must be an **`int`**. It specifies the size of the returned array.

If the *init* argument is given, it must be `bytes`. It is used to initialize the array items. Bytes not initialized this way are set to zero (NUL).

If *size* is not given (or if it is `None`), the buffer is made one element larger than *init*, effectively adding a NUL terminator.

If both arguments are given, *size* must not be less than `len(init)`.

Warning

If *size* is equal to `len(init)`, a NUL terminator is not added. Do not treat such a buffer as a C string.

For example:

```
>>> bytes(create_string_buffer(2))
b'\x00\x00'
>>> bytes(create_string_buffer(b'ab'))
b'ab\x00'
>>> bytes(create_string_buffer(b'ab', 2))
b'ab'
>>> bytes(create_string_buffer(b'ab', 4))
b'ab\x00\x00'
>>> bytes(create_string_buffer(b'abcdef', 2))
Traceback (most recent call last):
   ...
ValueError: byte string too long
```

Raises an auditing event `ctypes.create_string_buffer` with arguments `init`, `size`.

ctypes.create_unicode_buffer(*init*, *size=None*)
ctypes.create_unicode_buffer(*size*)

This function creates a mutable unicode character buffer. The returned object is a ctypes array of `c_wchar`.

The function takes the same arguments as `create_string_buffer()` except *init* must be a string and *size* counts `c_wchar`.

Raises an auditing event `ctypes.create_unicode_buffer` with arguments `init`, `size`.

ctypes.DllCanUnloadNow()

This function is a hook which allows implementing in-process COM servers with ctypes. It is called from the DllCanUnloadNow function that the _ctypes extension dll exports.

Availability: Windows

ctypes.DllGetClassObject()

This function is a hook which allows implementing in-process COM servers with ctypes. It is called from the DllGetClassObject function that the `_ctypes` extension dll exports.

Availability: Windows

ctypes.util.find_library(*name*)

Try to find a library and return a pathname. *name* is the library name without any prefix like `lib`, suffix like `.so`, `.dylib` or version number (this is the form used for the posix linker option **`-l`**). If no library can be found, returns `None`.

The exact functionality is system dependent.

See Finding shared libraries for complete documentation.

ctypes.util.find_msvcrt()

Returns the filename of the VC runtime library used by Python, and by the extension modules. If the name of the library cannot be determined, `None` is returned.

If you need to free memory, for example, allocated by an extension module with a call to the `free(void *)`, it is important that you use the function in the same library that allocated the memory.

Availability: Windows

ctypes.util.dllist()

Try to provide a list of paths of the shared libraries loaded into the current process. These paths are not normalized or processed in any way. The function can raise **`OSError`** if the underlying platform APIs fail. The exact functionality is system dependent.

On most platforms, the first element of the list represents the current executable file. It may be an empty string.

Availability: Windows, macOS, iOS, glibc, BSD libc, musl

Added in version 3.14.

ctypes.FormatError([*code*])

Returns a textual description of the error code *code*. If no error code is specified, the last error code is used by calling the Windows API function **`GetLastError()`**.

Availability: Windows

ctypes.GetLastError()

Returns the last error code set by Windows in the calling thread. This function calls the Windows `GetLastError()`

function directly, it does not return the ctypes-private copy of the error code.

Availability: Windows

ctypes.get_errno()

Returns the current value of the ctypes-private copy of the system **`errno`** variable in the calling thread.

Raises an auditing event `ctypes.get_errno` with no arguments.

ctypes.get_last_error()

Returns the current value of the ctypes-private copy of the system **`LastError`** variable in the calling thread.

Availability: Windows

Raises an auditing event `ctypes.get_last_error` with no arguments.

ctypes.memmove(*dst*, *src*, *count*)

Same as the standard C memmove library function: copies *count* bytes from *src* to *dst*. *dst* and *src* must be integers or ctypes instances that can be converted to pointers.

ctypes.memset(*dst*, *c*, *count*)

Same as the standard C memset library function: fills the memory block at address *dst* with *count* bytes of value *c*. *dst* must be an integer specifying an address, or a ctypes instance.

ctypes.POINTER(*type*, /)

Create or return a ctypes pointer type. Pointer types are cached and reused internally, so calling this function repeatedly is cheap. *type* must be a ctypes type.

CPython implementation detail: The resulting pointer type is cached in the `__pointer_type__` attribute of *type*. It is possible to set this attribute before the first call to `POINTER` in order to set a custom pointer type. However, doing this is discouraged: manually creating a suitable pointer type is difficult without relying on implementation details that may change in future Python versions.

ctypes.pointer(*obj*, */*)

Create a new pointer instance, pointing to *obj*. The returned object is of the type `POINTER(type(obj))`.

Note: If you just want to pass a pointer to an object to a foreign function call, you should use `byref(obj)` which is much faster.

ctypes.resize(*obj*, *size*)

This function resizes the internal memory buffer of *obj*, which must be an instance of a ctypes type. It is not possible to make the buffer smaller than the native size of the objects type, as given by `sizeof(type(obj))`, but it is possible to enlarge the buffer.

ctypes.set_errno(*value*)

Set the current value of the ctypes-private copy of the system **`errno`** variable in the calling thread to *value* and return the previous value.

Raises an auditing event `ctypes.set_errno` with argument `errno`.

ctypes.set_last_error(*value*)

Sets the current value of the ctypes-private copy of the system **`LastError`** variable in the calling thread to *value* and return the previous value.

Availability: Windows

Raises an auditing event `ctypes.set_last_error` with argument `error`.

ctypes.sizeof(*obj_or_type*)

Returns the size in bytes of a ctypes type or instance memory buffer. Does the same as the C `sizeof` operator.

ctypes.string_at(*ptr*, *size=-1*)

Return the byte string at *void *ptr*. If *size* is specified, it is used as size, otherwise the string is assumed to be zero-terminated.

Raises an auditing event `ctypes.string_at` with arguments `ptr`, `size`.

ctypes.WinError(*code=None*, *descr=None*)

Creates an instance of **`OSError`**. If *code* is not specified, **`GetLastError()`** is called to determine the error code. If *descr* is not specified, **`FormatError()`** is called to get a textual description of the error.

Availability: Windows

Changed in version 3.3: An instance of **`WindowsError`** used to be created, which is now an alias of **`OSError`**.

ctypes.wstring_at(*ptr*, *size=-1*)

Return the wide-character string at *void *ptr*. If *size* is specified, it is used as the number of characters of the string, otherwise the string is assumed to be zero-terminated.

Raises an auditing event `ctypes.wstring_at` with arguments `ptr`, `size`.

ctypes.memoryview_at(*ptr*, *size*, *readonly=False*)

Return a **`memoryview`** object of length *size* that references memory starting at *void *ptr*.

If *readonly* is true, the returned **`memoryview`** object can not be used to modify the underlying memory. (Changes made by other means will still be reflected in the returned object.)

This function is similar to **`string_at()`** with the key difference of not making a copy of the specified memory. It is a semantically equivalent (but more efficient) alternative to `memoryview((c_byte * size).from_address(ptr))`. (While **`from_address()`** only takes integers, *ptr* can also be given as a **`ctypes.POINTER`** or a **`byref()`** object.)

Raises an auditing event `ctypes.memoryview_at` with arguments `address`, `size`, `readonly`.

Added in version 3.14.

Data types

class ctypes._CData

This non-public class is the common base class of all ctypes data types. Among other things, all ctypes type instances contain a memory block that hold C compatible data; the address of the memory block is returned by the **`addressof()`** helper function. Another instance variable is exposed as **`_objects`**; this contains other Python objects that need to be kept alive in case the memory block contains pointers.

Common methods of ctypes data types, these are all class methods (to be exact, they are methods of the metaclass):

from_buffer(*source*[, *offset*])

This method returns a ctypes instance that shares the buffer of the *source* object. The *source* object must support the writeable buffer interface. The optional *offset* parameter specifies an offset into the source buffer in bytes; the default is zero. If the source buffer is not large enough a **`ValueError`** is raised.

Raises an auditing event `ctypes.cdata/buffer` with arguments `pointer`, `size`, `offset`.

from_buffer_copy(*source*[, *offset*])

This method creates a ctypes instance, copying the buffer from the *source* object buffer which must be readable. The optional *offset* parameter specifies an offset into the source buffer in bytes; the default is zero. If the source buffer is not large enough a **`ValueError`** is raised.

Raises an auditing event `ctypes.cdata/buffer` with arguments `pointer`, `size`, `offset`.

from_address(*address*)

This method returns a ctypes type instance using the memory specified by *address* which must be an integer.

This method, and others that indirectly call this method, raises an auditing event `ctypes.cdata` with argument `address`.

from_param(*obj*)

This method adapts *obj* to a ctypes type. It is called with the actual object used in a foreign function call when the type is present in the foreign function's `argtypes` tuple; it must return an object that can be used as a function call parameter.

All ctypes data types have a default implementation of this classmethod that normally returns *obj* if that is an instance of the type. Some types accept other objects as well.

in_dll(*library*, *name*)

This method returns a ctypes type instance exported by a shared library. *name* is the name of the symbol that exports the data, *library* is the loaded shared library.

Common class variables of ctypes data types:

__pointer_type__

The pointer type that was created by calling `POINTER()` for corresponding ctypes data type. If a pointer type was not yet created, the attribute is missing.

Added in version 3.14.

Common instance variables of ctypes data types:

_b_base_

Sometimes ctypes data instances do not own the memory block they contain, instead they share part of the memory block of a base object. The `_b_base_` read-only member is the root ctypes object that owns the memory block.

_b_needsfree_

This read-only variable is true when the ctypes data instance has allocated the memory block itself, false otherwise.

_objects

This member is either `None` or a dictionary containing Python objects that need to be kept alive so that the memory block contents is kept valid. This object is only exposed for debugging; never modify the contents of this dictionary.

Fundamental data types

class ctypes._SimpleCData

This non-public class is the base class of all fundamental ctypes data types. It is mentioned here because it contains the common attributes of the fundamental ctypes data types. `_SimpleCData` is a subclass of `_CData`, so it inherits their methods and attributes. ctypes data types that are not and do not contain pointers can now be pickled.

Instances have a single attribute:

value

This attribute contains the actual value of the instance. For integer and pointer types, it is an integer, for character types, it is a single character bytes object or string, for character pointer types it is a Python bytes object or string.

When the `value` attribute is retrieved from a ctypes instance, usually a new object is returned each time. `ctypes` does *not* implement original object return, always a new object is constructed. The same is true for all other ctypes object instances.

Fundamental data types, when returned as foreign function call results, or, for example, by retrieving structure field members or array items, are transparently converted to native Python types. In other words, if a foreign function has a **`restype`** of **`c_char_p`**, you will always receive a Python bytes object, *not* a **`c_char_p`** instance.

Subclasses of fundamental data types do *not* inherit this behavior. So, if a foreign functions **`restype`** is a subclass of **`c_void_p`**, you will receive an instance of this subclass from the function call. Of course, you can get the value of the pointer by accessing the `value` attribute.

These are the fundamental ctypes data types:

class ctypes.c_byte

Represents the C signed char datatype, and interprets the value as small integer. The constructor accepts an optional integer initializer; no overflow checking is done.

class ctypes.c_char

Represents the C char datatype, and interprets the value as a single character. The constructor accepts an optional string initializer, the length of the string must be exactly one character.

class ctypes.c_char_p

Represents the C char* datatype when it points to a zero-terminated string. For a general character pointer that may also point to binary data, `POINTER(c_char)` must be used. The constructor accepts an integer address, or a bytes object.

class ctypes.c_double

Represents the C double datatype. The constructor accepts an optional float initializer.

class ctypes.c_longdouble

Represents the C long double datatype. The constructor accepts an optional float initializer. On platforms where `sizeof(long double) == sizeof(double)` it is an alias to **`c_double`**.

class ctypes.c_float

Represents the C float datatype. The constructor accepts an optional float initializer.

class ctypes.c_double_complex

Represents the C double complex datatype, if available. The constructor accepts an optional **`complex`** initializer.

Added in version 3.14.

class ctypes.c_float_complex

Represents the C float complex datatype, if available. The constructor accepts an optional **`complex`** initializer.

Added in version 3.14.

class ctypes.c_longdouble_complex

Represents the C long double complex datatype, if available. The constructor accepts an optional **`complex`** initializer.

Added in version 3.14.

class ctypes.c_int

Represents the C signed int datatype. The constructor accepts an optional integer initializer; no overflow checking is done. On platforms where `sizeof(int) == sizeof(long)` it is an alias to **`c_long`**.

class ctypes.c_int8

Represents the C 8-bit signed int datatype. It is an alias for **`c_byte`**.

class ctypes.c_int16

Represents the C 16-bit signed int datatype. Usually an alias for **`c_short`**.

class ctypes.c_int32

Represents the C 32-bit signed int datatype. Usually an alias for **`c_int`**.

class ctypes.c_int64

Represents the C 64-bit signed int datatype. Usually an alias for **`c_longlong`**.

class ctypes.c_long

Represents the C signed long datatype. The constructor accepts an optional integer initializer; no overflow checking is done.

class ctypes.c_longlong

Represents the C signed long long datatype. The constructor accepts an optional integer initializer; no overflow checking is done.

class ctypes.c_short

Represents the C signed short datatype. The constructor accepts an optional integer initializer; no overflow checking is done.

class ctypes.c_size_t

Represents the C **`size_t`** datatype.

class ctypes.c_ssize_t

Represents the C **`ssize_t`** datatype.

Added in version 3.2.

class ctypes.c_time_t

Represents the C **`time_t`** datatype.

Added in version 3.12.

class ctypes.c_ubyte

Represents the C unsigned char datatype, it interprets the value as small integer. The constructor accepts an optional integer initializer; no overflow checking is done.

class ctypes.c_uint

Represents the C unsigned int datatype. The constructor accepts an optional integer initializer; no overflow checking is done. On platforms where `sizeof(int) == sizeof(long)` it is an alias for `c_ulong`.

class ctypes.c_uint8

Represents the C 8-bit unsigned int datatype. It is an alias for `c_ubyte`.

class ctypes.c_uint16

Represents the C 16-bit unsigned int datatype. Usually an alias for `c_ushort`.

class ctypes.c_uint32

Represents the C 32-bit unsigned int datatype. Usually an alias for `c_uint`.

class ctypes.c_uint64

Represents the C 64-bit unsigned int datatype. Usually an alias for `c_ulonglong`.

class ctypes.c_ulong

Represents the C unsigned long datatype. The constructor accepts an optional integer initializer; no overflow checking is done.

class ctypes.c_ulonglong

Represents the C unsigned long long datatype. The constructor accepts an optional integer initializer; no overflow checking is done.

class ctypes.c_ushort

Represents the C unsigned short datatype. The constructor accepts an optional integer initializer; no overflow checking is done.

class ctypes.c_void_p

Represents the C void* type. The value is represented as integer. The constructor accepts an optional integer initializer.

class ctypes.c_wchar

Represents the C **`wchar_t`** datatype, and interprets the value as a single character unicode string. The constructor accepts an optional string initializer, the length of the string must be exactly one character.

class ctypes.c_wchar_p

Represents the C wchar_t* datatype, which must be a pointer to a zero-terminated wide character string. The constructor accepts an integer address, or a string.

class ctypes.c_bool

Represent the C bool datatype (more accurately, _Bool from C99). Its value can be `True` or `False`, and the constructor accepts any object that has a truth value.

class ctypes.HRESULT

Represents a **`HRESULT`** value, which contains success or error information for a function or method call.

Availability: Windows

class ctypes.py_object

Represents the C PyObject* datatype. Calling this without an argument creates a `NULL` PyObject* pointer.

Changed in version 3.14: **`py_object`** is now a generic type.

The **`ctypes.wintypes`** module provides quite some other Windows specific data types, for example **`HWND`**, **`WPARAM`**, or **`DWORD`**. Some useful structures like **`MSG`** or **`RECT`** are also defined.

Structured data types

class ctypes.Union(**args*, ***kw*)

Abstract base class for unions in native byte order.

Unions share common attributes and behavior with structures; see **`Structure`** documentation for details.

class ctypes.BigEndianUnion(**args*, ***kw*)

Abstract base class for unions in *big endian* byte order.

Added in version 3.11.

class ctypes.LittleEndianUnion(**args*, ***kw*)

Abstract base class for unions in *little endian* byte order.

Added in version 3.11.

class ctypes.BigEndianStructure(**args*, ***kw*)

Abstract base class for structures in *big endian* byte order.

class ctypes.LittleEndianStructure(**args*, ***kw*)

Abstract base class for structures in *little endian* byte order.

Structures and unions with non-native byte order cannot contain pointer type fields, or any other data types containing pointer type fields.

class ctypes.Structure(**args*, ***kw*)

Abstract base class for structures in *native* byte order.

Concrete structure and union types must be created by subclassing one of these types, and at least define a `_fields_` class variable. `ctypes` will create descriptors which allow reading and writing the fields by direct attribute accesses. These are the

fields

A sequence defining the structure fields. The items must be 2-tuples or 3-tuples. The first item is the name of the field, the second item specifies the type of the field; it can be any ctypes data type.

For integer type fields like `c_int`, a third optional item can be given. It must be a small positive integer defining the bit width of the field.

Field names must be unique within one structure or union. This is not checked, only one field can be accessed when names are repeated.

It is possible to define the `_fields_` class variable *after* the class statement that defines the Structure subclass, this allows creating data types that directly or indirectly reference themselves:

```
class List(Structure):
    pass
List._fields_ = [("pnext", POINTER(List)),
                 ...
                ]
```

The **`_fields_`** class variable can only be set once. Later assignments will raise an `AttributeError`.

Additionally, the **`_fields_`** class variable must be defined before the structure or union type is first used: an instance or subclass is created, `sizeof()` is called on it, and so on. Later assignments to **`_fields_`** will raise an `AttributeError`. If **`_fields_`** has not been set before such use, the structure or union will have no own fields, as if **`_fields_`** was empty.

Sub-subclasses of structure types inherit the fields of the base class plus the `_fields_` defined in the sub-subclass, if any.

pack

An optional small integer that allows overriding the alignment of structure fields in the instance.

This is only implemented for the MSVC-compatible memory layout (see `_layout_`).

Setting **`_pack_`** to 0 is the same as not setting it at all. Otherwise, the value must be a positive power of two. The effect is equivalent to `#pragma pack(N)` in C, except `ctypes` may allow larger *n* than what the compiler accepts.

`_pack_` must already be defined when [`_fields_`]() is assigned, otherwise it will have no effect.

Deprecated since version 3.14, will be removed in version 3.19: For historical reasons, if **`_pack_`** is non-zero, the MSVC-compatible layout will be used by default. On non-Windows platforms, this default is deprecated and is slated to become an error in Python 3.19. If it is intended, set [`_layout_`]() to `'ms'` explicitly.

align

An optional small integer that allows increasing the alignment of the structure when being packed or unpacked to/from memory.

The value must not be negative. The effect is equivalent to `__attribute__((aligned(N)))` on GCC or `#pragma align(N)` on MSVC, except [`ctypes`]() may allow values that the compiler would reject.

`_align_` can only *increase* a structure’s alignment requirements. Setting it to 0 or 1 has no effect.

Using values that are not powers of two is discouraged and may lead to surprising behavior.

`_align_` must already be defined when [`_fields_`]() is assigned, otherwise it will have no effect.

Added in version 3.13.

layout

An optional string naming the struct/union layout. It can currently be set to:

- `"ms"`: the layout used by the Microsoft compiler (MSVC). On GCC and Clang, this layout can be

  selected with `__attribute__((ms_struct))`.
- `"gcc-sysv"`: the layout used by GCC with the System V or “SysV-like” data model, as used on Linux and macOS. With this layout, `_pack_` must be unset or zero.

If not set explicitly, `ctypes` will use a default that matches the platform conventions. This default may change in future Python releases (for example, when a new platform gains official support, or when a difference between similar platforms is found). Currently the default will be:

- On Windows: `"ms"`
- When `_pack_` is specified: `"ms"`. (This is deprecated; see `_pack_` documentation.)
- Otherwise: `"gcc-sysv"`

`_layout_` must already be defined when `_fields_` is assigned, otherwise it will have no effect.

Added in version 3.14.

anonymous

An optional sequence that lists the names of unnamed (anonymous) fields. `_anonymous_` must be already defined when `_fields_` is assigned, otherwise it will have no effect.

The fields listed in this variable must be structure or union type fields. `ctypes` will create descriptors in the structure type that allows accessing the nested fields directly, without the need to create the structure or union field.

Here is an example type (Windows):

```
class _U(Union):
    _fields_ = [("lptdesc", POINTER(TYPEDESC)),
                ("lpadesc", POINTER(ARRAYDESC)),
                ("hreftype", HREFTYPE)]

class TYPEDESC(Structure):
    _anonymous_ = ("u",)
    _fields_ = [("u", _U),
                ("vt", VARTYPE)]
```

The `TYPEDESC` structure describes a COM data type, the `vt` field specifies which one of the union fields is valid. Since the `u` field is defined as anonymous field, it is now possible to access the members directly off the TYPEDESC instance. `td.lptdesc` and `td.u.lptdesc` are equivalent, but the former is faster since it does not need to create a temporary union instance:

```
td = TYPEDESC()
td.vt = VT_PTR
td.lptdesc = POINTER(some_type)
td.u.lptdesc = POINTER(some_type)
```

It is possible to define sub-subclasses of structures, they inherit the fields of the base class. If the subclass definition has a separate **`_fields_`** variable, the fields specified in this are appended to the fields of the base class.

Structure and union constructors accept both positional and keyword arguments. Positional arguments are used to initialize member fields in the same order as they are appear in **`_fields_`**. Keyword arguments in the constructor are interpreted as attribute assignments, so they will initialize **`_fields_`** with the same name, or create new attributes for names not present in **`_fields_`**.

class ctypes.CField(**args*, ***kw*)

Descriptor for fields of a **`Structure`** and **`Union`**. For example:

```
>>> class Color(Structure):
...     _fields_ = (
...         ('red', c_uint8),
...         ('green', c_uint8),
...         ('blue', c_uint8),
...         ('intense', c_bool, 1),
...         ('blinking', c_bool, 1),
...    )
...
>>> Color.red
<ctypes.CField 'red' type=c_ubyte, ofs=0, size=1>
>>> Color.green.type
<class 'ctypes.c_ubyte'>
>>> Color.blue.byte_offset
2
>>> Color.intense
<ctypes.CField 'intense' type=c_bool, ofs=3, bit_size=1,
bit_offset=0>
>>> Color.blinking.bit_offset
1
```

All attributes are read-only.

`CField` objects are created via [**`_fields_`**]; do not instantiate the class directly.

Added in version 3.14: Previously, descriptors only had `offset` and `size` attributes and a readable string representation; the **`CField`** class was not available directly.

name

  Name of the field, as a string.

type

  Type of the field, as a ctypes class.

offset
byte_offset

  Offset of the field, in bytes.

For bitfields, this is the offset of the underlying byte-aligned *storage unit*; see **`bit_offset`**.

byte_size

Size of the field, in bytes.

For bitfields, this is the size of the underlying *storage unit*. Typically, it has the same size as the bitfield’s type.

size

For non-bitfields, equivalent to **`byte_size`**.

For bitfields, this contains a backwards-compatible bit-packed value that combines **`bit_size`** and **`bit_offset`**. Prefer using the explicit attributes instead.

is_bitfield

True if this is a bitfield.

bit_offset
bit_size

The location of a bitfield within its *storage unit*, that is, within **`byte_size`** bytes of memory starting at **`byte_offset`**.

To get the field’s value, read the storage unit as an integer, shift left by **`bit_offset`** and take the **`bit_size`** least significant bits.

For non-bitfields, **`bit_offset`** is zero and **`bit_size`** is equal to `byte_size * 8`.

is_anonymous

True if this field is anonymous, that is, it contains nested sub-fields that should be merged into a containing structure or union.

Arrays and pointers

class ctypes.Array(**args*)

Abstract base class for arrays.

The recommended way to create concrete array types is by multiplying any `ctypes` data type with a non-negative integer. Alternatively, you can subclass this type and define `_length_` and `_type_` class variables. Array elements can be read and written using standard subscript and slice accesses; for slice reads, the resulting object is *not* itself an `Array`.

length

A positive integer specifying the number of elements in the array. Out-of-range subscripts result in an `IndexError`. Will be returned by `len()`.

type

Specifies the type of each element in the array.

Array subclass constructors accept positional arguments, used to initialize the elements in order.

ctypes.ARRAY(*type*, *length*)

Create an array. Equivalent to `type * length`, where *type* is a `ctypes` data type and *length* an integer.

This function is soft deprecated in favor of multiplication. There are no plans to remove it.

class ctypes._Pointer

Private, abstract base class for pointers.

Concrete pointer types are created by calling `POINTER()` with the type that will be pointed to; this is done automatically by `pointer()`.

If a pointer points to an array, its elements can be read and written using standard subscript and slice accesses. Pointer objects have no size, so `len()` will raise `TypeError`. Negative subscripts will read from the memory *before* the pointer (as in C), and out-of-range subscripts will probably crash with an access violation (if you’re lucky).

type

Specifies the type pointed to.

contents

Returns the object to which to pointer points. Assigning to this attribute changes the pointer to point to the assigned object.

Exceptions

exception ctypes.ArgumentError

This exception is raised when a foreign function call cannot convert one of the passed arguments.

exception ctypes.COMError(*hresult*, *text*, *details*)

This exception is raised when a COM method call failed.

hresult

The integer value representing the error code.

text

The error message.

details

The 5-tuple `(descr, source, helpfile, helpcontext, progid)`.

descr is the textual description. *source* is the language-dependent `ProgID` for the class or application that raised the error. *helpfile* is the path of the help file. *helpcontext* is the help context identifier. *progid* is the `ProgID` of the interface that defined the error.

Availability: Windows

Added in version 3.14.

Command-line interface libraries

The modules described in this chapter assist with implementing command line and terminal interfaces for applications.

Here’s an overview:

- `argparse` — Parser for command-line options, arguments and subcommands
- `optparse` — Parser for command line options
- `getpass` — Portable password input
- `fileinput` — Iterate over lines from multiple input streams
- `curses` — Terminal handling for character-cell displays
- `curses.textpad` — Text input widget for curses programs
- `curses.ascii` — Utilities for ASCII characters
- `curses.panel` — A panel stack extension for curses
- `cmd` — Support for line-oriented command interpreters

`argparse` — Parser for command-line options, arguments and subcommands

Added in version 3.2.

Source code: Lib/argparse.py
[https://github.com/python/cpython/tree/3.14/Lib/argparse.py]

Note

While **`argparse`** is the default recommended standard library module for implementing basic command line applications, authors with more exacting requirements for exactly how their command line applications behave may find it doesn't provide the necessary level of control. Refer to Choosing an argument parsing library for alternatives to consider when `argparse` doesn't support behaviors that the application requires (such as entirely disabling support for interspersed options and positional arguments, or accepting option parameter values that start with `-` even when they correspond to another defined option).

Tutorial

This page contains the API reference

The **`argparse`** module makes it easy to write user-friendly command-line interfaces. The program defines what arguments it requires, and

`argparse` will figure out how to parse those out of [`sys.argv`](). The **`argparse`** module also automatically generates help and usage messages. The module will also issue errors when users give the program invalid arguments.

information. For a more gentle introduction to Python command-line parsing, have a look at the [argparse tutorial]().

The **`argparse`** module’s support for command-line interfaces is built around an instance of [`argparse.ArgumentParser`](). It is a container for argument specifications and has options that apply to the parser as whole:

```
parser = argparse.ArgumentParser(
                    prog='ProgramName',
                    description='What the program does',
                    epilog='Text at the bottom of help')
```

The [`ArgumentParser.add_argument()`]() method attaches individual argument specifications to the parser. It supports positional arguments, options that accept values, and on/off flags:

```
parser.add_argument('filename')           # positional argument
parser.add_argument('-c', '--count')      # option that takes a
value
parser.add_argument('-v', '--verbose',
                    action='store_true')  # on/off flag
```

The [`ArgumentParser.parse_args()`]() method runs the parser and places the extracted data in a [`argparse.Namespace`]() object:

```
args = parser.parse_args()
print(args.filename, args.count, args.verbose)
```

Note

If you’re looking for a guide about how to upgrade **`optparse`** code to **`argparse`**, see Upgrading Optparse Code.

ArgumentParser objects

class argparse.ArgumentParser(*prog=None*, *usage=None*, *description=None*, *epilog=None*, *parents=[]*, *formatter_class=argparse.HelpFormatter*, *prefix_chars='-'*, *fromfile_prefix_chars=None*, *argument_default=None*, *conflict_handler='error'*, *add_help=True*, *allow_abbrev=True*, *exit_on_error=True*, ***, *suggest_on_error=False*, *color=True*)

Create a new **`ArgumentParser`** object. All parameters should be passed as keyword arguments. Each parameter has its own more detailed description below, but in short they are:

- prog - The name of the program (default: generated from the `__main__` module attributes and `sys.argv[0]`)
- usage - The string describing the program usage (default: generated from arguments added to parser)
- description - Text to display before the argument help (by default, no text)
- epilog - Text to display after the argument help (by default, no text)
- parents - A list of **`ArgumentParser`** objects whose arguments should also be included
- formatter_class - A class for customizing the help output
- prefix_chars - The set of characters that prefix optional arguments (default: ‘-‘)
- fromfile_prefix_chars - The set of characters that prefix files from which additional arguments should be read (default: `None`)

- argument_default - The global default value for arguments (default: `None`)
- conflict_handler - The strategy for resolving conflicting optionals (usually unnecessary)
- add_help - Add a `-h/--help` option to the parser (default: `True`)
- allow_abbrev - Allows long options to be abbreviated if the abbreviation is unambiguous (default: `True`)
- exit_on_error - Determines whether or not **`ArgumentParser`** exits with error info when an error occurs. (default: `True`)
- suggest_on_error - Enables suggestions for mistyped argument choices and subparser names (default: `False`)
- color - Allow color output (default: `True`)

Changed in version 3.5: allow_abbrev parameter was added.

Changed in version 3.8: In previous versions, *allow_abbrev* also disabled grouping of short flags such as `-vv` to mean `-v -v`.

Changed in version 3.9: exit_on_error parameter was added.

Changed in version 3.14: suggest_on_error and *color* parameters were added.

The following sections describe how each of these are used.

prog

By default, **`ArgumentParser`** calculates the name of the program to display in help messages depending on the way the Python interpreter was run:

- The **base name** of `sys.argv[0]` if a file was passed as argument.
- The Python interpreter name followed by `sys.argv[0]` if a directory or a zipfile was passed as argument.
- The Python interpreter name followed by `-m` followed by the module or package name if the **`-m`** option was used.

This default is almost always desirable because it will make the help messages match the string that was used to invoke the program on the command line. However, to change this default behavior, another value can be supplied using the `prog=` argument to **`ArgumentParser`**:

```
>>> parser = argparse.ArgumentParser(prog='myprogram')
>>> parser.print_help()
usage: myprogram [-h]

options:
 -h, --help  show this help message and exit
```

Note that the program name, whether determined from `sys.argv[0]`, from the `__main__` module attributes or from the `prog=` argument, is available to help messages using the `%(prog)s` format specifier.

```
>>> parser = argparse.ArgumentParser(prog='myprogram')
>>> parser.add_argument('--foo', help='foo of the %(prog)s
program')
>>> parser.print_help()
usage: myprogram [-h] [--foo FOO]

options:
 -h, --help  show this help message and exit
 --foo FOO   foo of the myprogram program
```

Changed in version 3.14: The default `prog` value now reflects how `__main__` was actually executed, rather than always being `os.path.basename(sys.argv[0])`.

usage

By default, `ArgumentParser` calculates the usage message from the arguments it contains. The default message can be overridden with the `usage=` keyword argument:

```
>>> parser = argparse.ArgumentParser(prog='PROG', usage='%
(prog)s [options]')
>>> parser.add_argument('--foo', nargs='?', help='foo help')
>>> parser.add_argument('bar', nargs='+', help='bar help')
>>> parser.print_help()
usage: PROG [options]

positional arguments:
 bar          bar help

options:
 -h, --help   show this help message and exit
 --foo [FOO]  foo help
```

The `%(prog)s` format specifier is available to fill in the program name in your usage messages.

When a custom usage message is specified for the main parser, you may also want to consider passing the `prog` argument to `add_subparsers()` or the `prog` and the `usage` arguments to **`add_parser()`**, to ensure consistent command prefixes and usage information across subparsers.

description

Most calls to the `ArgumentParser` constructor will use the `description=` keyword argument. This argument gives a brief description of what the program does and how it works. In help messages, the description is displayed between the command-line usage string and the help messages for the various arguments.

By default, the description will be line-wrapped so that it fits within the given space. To change this behavior, see the formatter_class argument.

epilog

Some programs like to display additional description of the program after the description of the arguments. Such text can be specified using the `epilog=` argument to **`ArgumentParser`**:

```
>>> parser = argparse.ArgumentParser(
...     description='A foo that bars',
...     epilog="And that's how you'd foo a bar")
>>> parser.print_help()
usage: argparse.py [-h]

A foo that bars

options:
 -h, --help  show this help message and exit

And that's how you'd foo a bar
```

As with the description argument, the `epilog=` text is by default line-wrapped, but this behavior can be adjusted with the formatter_class argument to **`ArgumentParser`**.

parents

Sometimes, several parsers share a common set of arguments. Rather than repeating the definitions of these arguments, a single parser with all the shared arguments and passed to `parents=` argument to **`ArgumentParser`** can be used. The `parents=` argument takes a list of **`ArgumentParser`** objects, collects all the positional and optional actions from them, and adds these actions to the **`ArgumentParser`** object being constructed:

```
>>> parent_parser = argparse.ArgumentParser(add_help=False)
>>> parent_parser.add_argument('--parent', type=int)

>>> foo_parser = argparse.ArgumentParser(parents=
[parent_parser])
>>> foo_parser.add_argument('foo')
>>> foo_parser.parse_args(['--parent', '2', 'XXX'])
Namespace(foo='XXX', parent=2)

>>> bar_parser = argparse.ArgumentParser(parents=
[parent_parser])
>>> bar_parser.add_argument('--bar')
>>> bar_parser.parse_args(['--bar', 'YYY'])
Namespace(bar='YYY', parent=None)
```

Note that most parent parsers will specify `add_help=False`. Otherwise, the **`ArgumentParser`** will see two `-h/--help` options (one in the parent and one in the child) and raise an error.

Note

You must fully initialize the parsers before passing them via `parents=`. If you change the parent parsers after the child parser, those changes will not be reflected in the child.

formatter_class

`ArgumentParser` objects allow the help formatting to be customized by specifying an alternate formatting class. Currently, there are four such classes:

class argparse.RawDescriptionHelpFormatter
class argparse.RawTextHelpFormatter
class argparse.ArgumentDefaultsHelpFormatter
class argparse.MetavarTypeHelpFormatter

`RawDescriptionHelpFormatter` and **`RawTextHelpFormatter`** give more control over how textual descriptions are displayed. By default, **`ArgumentParser`** objects line-wrap the description and epilog texts in command-line help messages:

```
>>> parser = argparse.ArgumentParser(
...     prog='PROG',
...     description='''this description
...         was indented weird
...             but that is okay''',
...     epilog='''
...             likewise for this epilog whose whitespace will
...         be cleaned up and whose words will be wrapped
...         across a couple lines''')
>>> parser.print_help()
usage: PROG [-h]

this description was indented weird but that is okay

options:
 -h, --help  show this help message and exit

likewise for this epilog whose whitespace will be cleaned up
and whose words
will be wrapped across a couple lines
```

Passing **`RawDescriptionHelpFormatter`** as `formatter_class=` indicates that description and epilog are already correctly formatted and should not be line-wrapped:

```
>>> parser = argparse.ArgumentParser(
...     prog='PROG',
...     formatter_class=argparse.RawDescriptionHelpFormatter,
...     description=textwrap.dedent('''\
...         Please do not mess up this text!
...         --------------------------------
...             I have indented it
...             exactly the way
...             I want it
...         '''))
>>> parser.print_help()
```

```
usage: PROG [-h]

Please do not mess up this text!
--------------------------------
   I have indented it
   exactly the way
   I want it

options:
 -h, --help  show this help message and exit
```

`RawTextHelpFormatter` maintains whitespace for all sorts of help text, including argument descriptions. However, multiple newlines are replaced with one. If you wish to preserve multiple blank lines, add spaces between the newlines.

`ArgumentDefaultsHelpFormatter` automatically adds information about default values to each of the argument help messages:

```
>>> parser = argparse.ArgumentParser(
...     prog='PROG',
...     formatter_class=argparse.ArgumentDefaultsHelpFormatter)
>>> parser.add_argument('--foo', type=int, default=42,
help='FOO!')
>>> parser.add_argument('bar', nargs='*', default=[1, 2, 3],
help='BAR!')
>>> parser.print_help()
usage: PROG [-h] [--foo FOO] [bar ...]

positional arguments:
 bar         BAR! (default: [1, 2, 3])

options:
 -h, --help  show this help message and exit
 --foo FOO   FOO! (default: 42)
```

`MetavarTypeHelpFormatter` uses the name of the type argument for each argument as the display name for its values (rather than using the dest as the regular formatter does):

```
>>> parser = argparse.ArgumentParser(
...     prog='PROG',
...     formatter_class=argparse.MetavarTypeHelpFormatter)
>>> parser.add_argument('--foo', type=int)
>>> parser.add_argument('bar', type=float)
>>> parser.print_help()
usage: PROG [-h] [--foo int] float

positional arguments:
  float

options:
  -h, --help  show this help message and exit
  --foo int
```

prefix_chars

Most command-line options will use `-` as the prefix, e.g. `-f/--foo`. Parsers that need to support different or additional prefix characters, e.g. for options like `+f` or `/foo`, may specify them using the `prefix_chars=` argument to the `ArgumentParser` constructor:

```
>>> parser = argparse.ArgumentParser(prog='PROG',
prefix_chars='-+')
>>> parser.add_argument('+f')
>>> parser.add_argument('++bar')
>>> parser.parse_args('+f X ++bar Y'.split())
Namespace(bar='Y', f='X')
```

The `prefix_chars=` argument defaults to `'-'`. Supplying a set of characters that does not include `-` will cause `-f/--foo` options to be disallowed.

fromfile_prefix_chars

Sometimes, when dealing with a particularly long argument list, it may make sense to keep the list of arguments in a file rather than typing it out at the command line. If the `fromfile_prefix_chars=` argument is given to the **`ArgumentParser`** constructor, then arguments that start with any of the specified characters will be treated as files, and will be replaced by the arguments they contain. For example:

```
>>> with open('args.txt', 'w',
encoding=sys.getfilesystemencoding()) as fp:
...     fp.write('-f\nbar')
...
>>> parser = argparse.ArgumentParser(fromfile_prefix_chars='@')
>>> parser.add_argument('-f')
>>> parser.parse_args(['-f', 'foo', '@args.txt'])
Namespace(f='bar')
```

Arguments read from a file must be one per line by default (but see also **`convert_arg_line_to_args()`**) and are treated as if they were in the same place as the original file referencing argument on the command line. So in the example above, the expression `['-f', 'foo', '@args.txt']` is considered equivalent to the expression `['-f', 'foo', '-f', 'bar']`.

Note

Empty lines are treated as empty strings (`''`), which are allowed as values but not as arguments. Empty lines that are read as arguments will result in an “unrecognized arguments” error.

`ArgumentParser` uses filesystem encoding and error handler to read the file containing arguments.

The `fromfile_prefix_chars=` argument defaults to `None`, meaning that arguments will never be treated as file references.

Changed in version 3.12: **`ArgumentParser`** changed encoding and errors to read arguments files from default (e.g.

`locale.getpreferredencoding(False)` and `"strict"`) to the filesystem encoding and error handler. Arguments file should be encoded in UTF-8 instead of ANSI Codepage on Windows.

argument_default

Generally, argument defaults are specified either by passing a default to `add_argument()` or by calling the `set_defaults()` methods with a specific set of name-value pairs. Sometimes however, it may be useful to specify a single parser-wide default for arguments. This can be accomplished by passing the `argument_default=` keyword argument to `ArgumentParser`. For example, to globally suppress attribute creation on `parse_args()` calls, we supply `argument_default=SUPPRESS`:

```
>>> parser =
argparse.ArgumentParser(argument_default=argparse.SUPPRESS)
>>> parser.add_argument('--foo')
>>> parser.add_argument('bar', nargs='?')
>>> parser.parse_args(['--foo', '1', 'BAR'])
Namespace(bar='BAR', foo='1')
>>> parser.parse_args([])
Namespace()
```

allow_abbrev

Normally, when you pass an argument list to the `parse_args()` method of an `ArgumentParser`, it recognizes abbreviations of long options.

This feature can be disabled by setting `allow_abbrev` to `False`:

```
>>> parser = argparse.ArgumentParser(prog='PROG',
allow_abbrev=False)
>>> parser.add_argument('--foobar', action='store_true')
>>> parser.add_argument('--foonley', action='store_false')
>>> parser.parse_args(['--foon'])
```

```
usage: PROG [-h] [--foobar] [--foonley]
PROG: error: unrecognized arguments: --foon
```

Added in version 3.5.

conflict_handler

`ArgumentParser` objects do not allow two actions with the same option string. By default, **`ArgumentParser`** objects raise an exception if an attempt is made to create an argument with an option string that is already in use:

```
>>> parser = argparse.ArgumentParser(prog='PROG')
>>> parser.add_argument('-f', '--foo', help='old foo help')
>>> parser.add_argument('--foo', help='new foo help')
Traceback (most recent call last):
 ..
ArgumentError: argument --foo: conflicting option string(s): --
foo
```

Sometimes (e.g. when using parents) it may be useful to simply override any older arguments with the same option string. To get this behavior, the value `'resolve'` can be supplied to the `conflict_handler=` argument of **`ArgumentParser`**:

```
>>> parser = argparse.ArgumentParser(prog='PROG',
conflict_handler='resolve')
>>> parser.add_argument('-f', '--foo', help='old foo help')
>>> parser.add_argument('--foo', help='new foo help')
>>> parser.print_help()
usage: PROG [-h] [-f FOO] [--foo FOO]

options:
 -h, --help  show this help message and exit
 -f FOO      old foo help
 --foo FOO   new foo help
```

Note that **`ArgumentParser`** objects only remove an action if all of its option strings are overridden. So, in the example above, the old `-f/--foo` action is retained as the `-f` action, because only the `--foo` option string was overridden.

add_help

By default, **`ArgumentParser`** objects add an option which simply displays the parser's help message. If `-h` or `--help` is supplied at the command line, the **`ArgumentParser`** help will be printed.

Occasionally, it may be useful to disable the addition of this help option. This can be achieved by passing `False` as the `add_help=` argument to **`ArgumentParser`**:

```
>>> parser = argparse.ArgumentParser(prog='PROG',
add_help=False)
>>> parser.add_argument('--foo', help='foo help')
>>> parser.print_help()
usage: PROG [--foo FOO]

options:
 --foo FOO  foo help
```

The help option is typically `-h/--help`. The exception to this is if the `prefix_chars=` is specified and does not include `-`, in which case `-h` and `--help` are not valid options. In this case, the first character in `prefix_chars` is used to prefix the help options:

```
>>> parser = argparse.ArgumentParser(prog='PROG',
prefix_chars='+/')
>>> parser.print_help()
usage: PROG [+h]

options:
  +h, ++help  show this help message and exit
```

exit_on_error

Normally, when you pass an invalid argument list to the `parse_args()` method of an `ArgumentParser`, it will print a *message* to `sys.stderr` and exit with a status code of 2.

If the user would like to catch errors manually, the feature can be enabled by setting `exit_on_error` to `False`:

```
>>> parser = argparse.ArgumentParser(exit_on_error=False)
>>> parser.add_argument('--integers', type=int)
_StoreAction(option_strings=['--integers'], dest='integers',
nargs=None, const=None, default=None, type=<class 'int'>,
choices=None, help=None, metavar=None)
>>> try:
...     parser.parse_args('--integers a'.split())
... except argparse.ArgumentError:
...     print('Catching an argumentError')
...
Catching an argumentError
```

Added in version 3.9.

suggest_on_error

By default, when a user passes an invalid argument choice or subparser name, `ArgumentParser` will exit with error info and list the permissible argument choices (if specified) or subparser names as part of the error message.

If the user would like to enable suggestions for mistyped argument choices and subparser names, the feature can be enabled by setting `suggest_on_error` to `True`. Note that this only applies for arguments when the choices specified are strings:

```
>>> parser = argparse.ArgumentParser(description='Process some
integers.',
                                     suggest_on_error=True)
>>> parser.add_argument('--action', choices=['sum', 'max'])
>>> parser.add_argument('integers', metavar='N', type=int,
nargs='+',
...                     help='an integer for the accumulator')
>>> parser.parse_args(['--action', 'sumn', 1, 2, 3])
tester.py: error: argument --action: invalid choice: 'sumn',
maybe you meant 'sum'? (choose from 'sum', 'max')
```

If you’re writing code that needs to be compatible with older Python versions and want to opportunistically use `suggest_on_error` when it’s available, you can set it as an attribute after initializing the parser instead of using the keyword argument:

```
>>> parser = argparse.ArgumentParser(description='Process some
integers.')
>>> parser.suggest_on_error = True
```

Added in version 3.14.

color

By default, the help message is printed in color using ANSI escape sequences [https://en.wikipedia.org/wiki/ANSI_escape_code]. If you want plain text help messages, you can disable this in your local environment, or in the argument parser itself by setting `color` to `False`:

```
>>> parser = argparse.ArgumentParser(description='Process some
integers.',
...                                  color=False)
>>> parser.add_argument('--action', choices=['sum', 'max'])
>>> parser.add_argument('integers', metavar='N', type=int,
nargs='+',
...                     help='an integer for the accumulator')
>>> parser.parse_args(['--help'])
```

Note that when `color=True`, colored output depends on both environment variables and terminal capabilities. However, if `color=False`, colored output is always disabled, even if environment variables like `FORCE_COLOR` are set.

Note

Error messages will include color codes when redirecting stderr to a file. To avoid this, set the **`NO_COLOR`** [https://no-color.org/] or **`PYTHON_COLORS`** environment variable (for example, `NO_COLOR=1 python script.py 2> errors.txt`).

Added in version 3.14.

The add_argument() method

ArgumentParser.add_argument(*name or flags...*, *[, *action*][, *nargs*][, *const*][, *default*][, *type*][, *choices*][, *required*][, *help*][, *metavar*][, *dest*][, *deprecated*])

Define how a single command-line argument should be parsed. Each parameter has its own more detailed description below, but in short they are:

- name or flags - Either a name or a list of option strings, e.g. `'foo'` or `'-f'`, `'--foo'`.
- action - The basic type of action to be taken when this argument is encountered at the command line.
- nargs - The number of command-line arguments that should be consumed.
- const - A constant value required by some action and nargs selections.
- default - The value produced if the argument is absent from the command line and if it is absent from the namespace object.
- type - The type to which the command-line argument should be converted.
- choices - A sequence of the allowable values for the argument.

- required - Whether or not the command-line option may be omitted (optionals only).
- help - A brief description of what the argument does.
- metavar - A name for the argument in usage messages.
- dest - The name of the attribute to be added to the object returned by **`parse_args()`**.
- deprecated - Whether or not use of the argument is deprecated.

The following sections describe how each of these are used.

name or flags

The **`add_argument()`** method must know whether an optional argument, like `-f` or `--foo`, or a positional argument, like a list of filenames, is expected. The first arguments passed to **`add_argument()`** must therefore be either a series of flags, or a simple argument name.

For example, an optional argument could be created like:

```
>>> parser.add_argument('-f', '--foo')
```

while a positional argument could be created like:

```
>>> parser.add_argument('bar')
```

When **`parse_args()`** is called, optional arguments will be identified by the `-` prefix, and the remaining arguments will be assumed to be positional:

```
>>> parser = argparse.ArgumentParser(prog='PROG')
>>> parser.add_argument('-f', '--foo')
>>> parser.add_argument('bar')
>>> parser.parse_args(['BAR'])
Namespace(bar='BAR', foo=None)
>>> parser.parse_args(['BAR', '--foo', 'FOO'])
```

```
Namespace(bar='BAR', foo='FOO')
>>> parser.parse_args(['--foo', 'FOO'])
usage: PROG [-h] [-f FOO] bar
PROG: error: the following arguments are required: bar
```

By default, **`argparse`** automatically handles the internal naming and display names of arguments, simplifying the process without requiring additional configuration. As such, you do not need to specify the dest and metavar parameters. The dest parameter defaults to the argument name with underscores `_` replacing hyphens `-` . The metavar parameter defaults to the upper-cased name. For example:

```
>>> parser = argparse.ArgumentParser(prog='PROG')
>>> parser.add_argument('--foo-bar')
>>> parser.parse_args(['--foo-bar', 'FOO-BAR'])
Namespace(foo_bar='FOO-BAR')
>>> parser.print_help()
usage:  [-h] [--foo-bar FOO-BAR]

optional arguments:
 -h, --help  show this help message and exit
 --foo-bar FOO-BAR
```

action

`ArgumentParser` objects associate command-line arguments with actions. These actions can do just about anything with the command-line arguments associated with them, though most actions simply add an attribute to the object returned by **`parse_args()`**. The `action` keyword argument specifies how the command-line arguments should be handled. The supplied actions are:

- `'store'` - This just stores the argument’s value. This is the default action.

- `'store_const'` - This stores the value specified by the const keyword argument; note that the const keyword argument defaults to `None`. The `'store_const'` action is most commonly used with optional arguments that specify some sort of flag. For example:

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('--foo', action='store_const',
const=42)
>>> parser.parse_args(['--foo'])
Namespace(foo=42)
```

- `'store_true'` and `'store_false'` - These are special cases of `'store_const'` that respectively store the values `True` and `False` with default values of `False` and `True`:

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('--foo', action='store_true')
>>> parser.add_argument('--bar', action='store_false')
>>> parser.add_argument('--baz', action='store_false')
>>> parser.parse_args('--foo --bar'.split())
Namespace(foo=True, bar=False, baz=True)
```

- `'append'` - This appends each argument value to a list. It is useful for allowing an option to be specified multiple times. If the default value is a non-empty list, the parsed value will start with the default list’s elements and any values from the command line will be appended after those default values. Example usage:

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('--foo', action='append', default=
['0'])
>>> parser.parse_args('--foo 1 --foo 2'.split())
Namespace(foo=['0', '1', '2'])
```

- `'append_const'` - This appends the value specified by the const keyword argument to a list; note that the const keyword argument defaults to `None`. The `'append_const'` action is typically useful when multiple arguments need to store constants to the same list. For example:

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('--str', dest='types',
action='append_const', const=str)
>>> parser.add_argument('--int', dest='types',
action='append_const', const=int)
>>> parser.parse_args('--str --int'.split())
Namespace(types=[<class 'str'>, <class 'int'>])
```

- `'extend'` - This appends each item from a multi-value argument to a list. The `'extend'` action is typically used with the nargs keyword argument value `'+'` or `'*'`. Note that when nargs is `None` (the default) or `'?'`, each character of the argument string will be appended to the list. Example usage:

  ```
  >>> parser = argparse.ArgumentParser()
  >>> parser.add_argument("--foo", action="extend",
  nargs="+", type=str)
  >>> parser.parse_args(["--foo", "f1", "--foo", "f2", "f3",
  "f4"])
  Namespace(foo=['f1', 'f2', 'f3', 'f4'])
  ```

  Added in version 3.8.

- `'count'` - This counts the number of times an argument occurs. For example, this is useful for increasing verbosity levels:

  ```
  >>> parser = argparse.ArgumentParser()
  >>> parser.add_argument('--verbose', '-v', action='count',
  default=0)
  >>> parser.parse_args(['-vvv'])
  Namespace(verbose=3)
  ```

  Note, the *default* will be `None` unless explicitly set to *0*.

- `'help'` - This prints a complete help message for all the options in the current parser and then exits. By default a help action is automatically added to the parser. See **`ArgumentParser`** for details of how the output is created.

- `'version'` - This expects a `version=` keyword argument in the `add_argument()` call, and prints version information and exits when invoked:

  ```
  >>> import argparse
  >>> parser = argparse.ArgumentParser(prog='PROG')
  >>> parser.add_argument('--version', action='version',
  version='%(prog)s 2.0')
  >>> parser.parse_args(['--version'])
  PROG 2.0
  ```

You may also specify an arbitrary action by passing an `Action` subclass (e.g. `BooleanOptionalAction`) or other object that implements the same interface. Only actions that consume command-line arguments (e.g. `'store'`, `'append'`, `'extend'`, or custom actions with non-zero `nargs`) can be used with positional arguments.

The recommended way to create a custom action is to extend `Action`, overriding the **`__call__()`** method and optionally the **`__init__()`** and **`format_usage()`** methods. You can also register custom actions using the `register()` method and reference them by their registered name.

An example of a custom action:

```
>>> class FooAction(argparse.Action):
...     def __init__(self, option_strings, dest, nargs=None,
**kwargs):
...         if nargs is not None:
...             raise ValueError("nargs not allowed")
...         super().__init__(option_strings, dest, **kwargs)
...     def __call__(self, parser, namespace, values,
option_string=None):
...         print('%r %r %r' % (namespace, values,
option_string))
...         setattr(namespace, self.dest, values)
...
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('--foo', action=FooAction)
```

```
>>> parser.add_argument('bar', action=FooAction)
>>> args = parser.parse_args('1 --foo 2'.split())
Namespace(bar=None, foo=None) '1' None
Namespace(bar='1', foo=None) '2' '--foo'
>>> args
Namespace(bar='1', foo='2')
```

For more details, see **`Action`**.

nargs

`ArgumentParser` objects usually associate a single command-line argument with a single action to be taken. The `nargs` keyword argument associates a different number of command-line arguments with a single action. See also Specifying ambiguous arguments. The supported values are:

- `N` (an integer). `N` arguments from the command line will be gathered together into a list. For example:

  ```
  >>> parser = argparse.ArgumentParser()
  >>> parser.add_argument('--foo', nargs=2)
  >>> parser.add_argument('bar', nargs=1)
  >>> parser.parse_args('c --foo a b'.split())
  Namespace(bar=['c'], foo=['a', 'b'])
  ```

  Note that `nargs=1` produces a list of one item. This is different from the default, in which the item is produced by itself.

- `'?'`. One argument will be consumed from the command line if possible, and produced as a single item. If no command-line argument is present, the value from default will be produced. Note that for optional arguments, there is an additional case - the option string is present but not followed by a command-line argument. In this case the value from const will be produced. Some examples to illustrate this:

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('--foo', nargs='?', const='c',
default='d')
>>> parser.add_argument('bar', nargs='?', default='d')
>>> parser.parse_args(['XX', '--foo', 'YY'])
Namespace(bar='XX', foo='YY')
>>> parser.parse_args(['XX', '--foo'])
Namespace(bar='XX', foo='c')
>>> parser.parse_args([])
Namespace(bar='d', foo='d')
```

One of the more common uses of `nargs='?'` is to allow optional input and output files:

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('infile', nargs='?')
>>> parser.add_argument('outfile', nargs='?')
>>> parser.parse_args(['input.txt', 'output.txt'])
Namespace(infile='input.txt', outfile='output.txt')
>>> parser.parse_args(['input.txt'])
Namespace(infile='input.txt', outfile=None)
>>> parser.parse_args([])
Namespace(infile=None, outfile=None)
```

- `'*'`. All command-line arguments present are gathered into a list. Note that it generally doesn’t make much sense to have more than one positional argument with `nargs='*'`, but multiple optional arguments with `nargs='*'` is possible. For example:

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('--foo', nargs='*')
>>> parser.add_argument('--bar', nargs='*')
>>> parser.add_argument('baz', nargs='*')
>>> parser.parse_args('a b --foo x y --bar 1 2'.split())
Namespace(bar=['1', '2'], baz=['a', 'b'], foo=['x', 'y'])
```

- `'+'`. Just like `'*'`, all command-line arguments present are gathered into a list. Additionally, an error message will be generated if there wasn’t at least one command-line argument present. For example:

```
>>> parser = argparse.ArgumentParser(prog='PROG')
>>> parser.add_argument('foo', nargs='+')
>>> parser.parse_args(['a', 'b'])
Namespace(foo=['a', 'b'])
>>> parser.parse_args([])
usage: PROG [-h] foo [foo ...]
PROG: error: the following arguments are required: foo
```

If the `nargs` keyword argument is not provided, the number of arguments consumed is determined by the action. Generally this means a single command-line argument will be consumed and a single item (not a list) will be produced. Actions that do not consume command-line arguments (e.g. `'store_const'`) set `nargs=0`.

const

The `const` argument of **`add_argument()`** is used to hold constant values that are not read from the command line but are required for the various **`ArgumentParser`** actions. The two most common uses of it are:

- When **`add_argument()`** is called with `action='store_const'` or `action='append_const'`. These actions add the `const` value to one of the attributes of the object returned by **`parse_args()`**. See the action description for examples. If `const` is not provided to **`add_argument()`**, it will receive a default value of `None`.
- When **`add_argument()`** is called with option strings (like `-f` or `--foo`) and `nargs='?'`. This creates an optional argument that can be followed by zero or one command-line arguments. When parsing the command line, if the option string is encountered with no command-line argument following it, the value from `const` will be used. See the nargs description for examples.

Changed in version 3.11: `const=None` by default, including when `action='append_const'` or `action='store_const'`.

default

All optional arguments and some positional arguments may be omitted at the command line. The `default` keyword argument of **`add_argument()`**, whose value defaults to `None`, specifies what value should be used if the command-line argument is not present. For optional arguments, the `default` value is used when the option string was not present at the command line:

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('--foo', default=42)
>>> parser.parse_args(['--foo', '2'])
Namespace(foo='2')
>>> parser.parse_args([])
Namespace(foo=42)
```

If the target namespace already has an attribute set, the action *default* will not overwrite it:

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('--foo', default=42)
>>> parser.parse_args([],
namespace=argparse.Namespace(foo=101))
Namespace(foo=101)
```

If the `default` value is a string, the parser parses the value as if it were a command-line argument. In particular, the parser applies any type conversion argument, if provided, before setting the attribute on the **`Namespace`** return value. Otherwise, the parser uses the value as is:

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('--length', default='10', type=int)
>>> parser.add_argument('--width', default=10.5, type=int)
>>> parser.parse_args()
Namespace(length=10, width=10.5)
```

For positional arguments with nargs equal to `?` or `*`, the `default` value is used when no command-line argument was present:

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('foo', nargs='?', default=42)
>>> parser.parse_args(['a'])
Namespace(foo='a')
>>> parser.parse_args([])
Namespace(foo=42)
```

For required arguments, the `default` value is ignored. For example, this applies to positional arguments with nargs values other than `?` or `*`, or optional arguments marked as `required=True`.

Providing `default=argparse.SUPPRESS` causes no attribute to be added if the command-line argument was not present:

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('--foo', default=argparse.SUPPRESS)
>>> parser.parse_args([])
Namespace()
>>> parser.parse_args(['--foo', '1'])
Namespace(foo='1')
```

type

By default, the parser reads command-line arguments in as simple strings. However, quite often the command-line string should instead be interpreted as another type, such as a **`float`** or **`int`**. The `type` keyword for **`add_argument()`** allows any necessary type-checking and type conversions to be performed.

If the type keyword is used with the default keyword, the type converter is only applied if the default is a string.

The argument to `type` can be a callable that accepts a single string or the name of a registered type (see **`register()`**) If the function raises **`ArgumentTypeError`**, **`TypeError`**, or **`ValueError`**, the exception is caught and a nicely formatted error message is displayed. Other exception types are not handled.

Common built-in types and functions can be used as type converters:

```
import argparse
import pathlib

parser = argparse.ArgumentParser()
parser.add_argument('count', type=int)
parser.add_argument('distance', type=float)
parser.add_argument('street', type=ascii)
parser.add_argument('code_point', type=ord)
parser.add_argument('datapath', type=pathlib.Path)
```

User defined functions can be used as well:

```
>>> def hyphenated(string):
...     return '-'.join([word[:4] for word in
string.casefold().split()])
...
>>> parser = argparse.ArgumentParser()
>>> _ = parser.add_argument('short_title', type=hyphenated)
>>> parser.parse_args(['"The Tale of Two Cities"'])
Namespace(short_title='"the-tale-of-two-citi')
```

The **`bool()`** function is not recommended as a type converter. All it does is convert empty strings to `False` and non-empty strings to `True`. This is usually not what is desired.

In general, the `type` keyword is a convenience that should only be used for simple conversions that can only raise one of the three supported exceptions. Anything with more interesting error-handling or resource management should be done downstream after the arguments are parsed.

For example, JSON or YAML conversions have complex error cases that require better reporting than can be given by the `type` keyword. A **`JSONDecodeError`** would not be well formatted and a **`FileNotFoundError`** exception would not be handled at all.

Even **`FileType`** has its limitations for use with the `type` keyword. If one argument uses **`FileType`** and then a subsequent argument fails, an error is reported but the file is not automatically closed. In this case, it would be better to wait until after the parser has run and then use the **`with`**-statement to manage the files.

For type checkers that simply check against a fixed set of values, consider using the choices keyword instead.

choices

Some command-line arguments should be selected from a restricted set of values. These can be handled by passing a sequence object as the *choices* keyword argument to **`add_argument()`**. When the command line is parsed, argument values will be checked, and an error message will be displayed if the argument was not one of the acceptable values:

```
>>> parser = argparse.ArgumentParser(prog='game.py')
>>> parser.add_argument('move', choices=['rock', 'paper',
'scissors'])
>>> parser.parse_args(['rock'])
Namespace(move='rock')
>>> parser.parse_args(['fire'])
usage: game.py [-h] {rock,paper,scissors}
game.py: error: argument move: invalid choice: 'fire' (choose
from 'rock',
'paper', 'scissors')
```

Any sequence can be passed as the *choices* value, so **`list`** objects, **`tuple`** objects, and custom sequences are all supported.

Use of **`enum.Enum`** is not recommended because it is difficult to control its appearance in usage, help, and error messages.

Note that *choices* are checked after any type conversions have been performed, so objects in *choices* should match the type specified. This can make *choices* appear unfamiliar in usage, help, or error messages.

To keep *choices* user-friendly, consider a custom type wrapper that converts and formats values, or omit type and handle conversion in your application code.

Formatted choices override the default *metavar* which is normally derived from *dest*. This is usually what you want because the user never sees the *dest* parameter. If this display isn't desirable (perhaps because there are many choices), just specify an explicit metavar.

required

In general, the **`argparse`** module assumes that flags like `-f` and `--bar` indicate *optional* arguments, which can always be omitted at the command line. To make an option *required*, `True` can be specified for the `required=` keyword argument to **`add_argument()`**:

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('--foo', required=True)
>>> parser.parse_args(['--foo', 'BAR'])
Namespace(foo='BAR')
>>> parser.parse_args([])
```

```
usage: [-h] --foo FOO
: error: the following arguments are required: --foo
```

As the example shows, if an option is marked as `required`, **`parse_args()`** will report an error if that option is not present at the command line.

Note

Required options are generally considered bad form because users expect *options* to be *optional*, and thus they should be avoided when possible.

help

The `help` value is a string containing a brief description of the argument. When a user requests help (usually by using `-h` or `--help` at the command line), these `help` descriptions will be displayed with each argument.

The `help` strings can include various format specifiers to avoid repetition of things like the program name or the argument default. The available specifiers include the program name, `%(prog)s` and most keyword arguments to **`add_argument()`**, e.g. `%(default)s`, `%(type)s`, etc.:

```
>>> parser = argparse.ArgumentParser(prog='frobble')
>>> parser.add_argument('bar', nargs='?', type=int, default=42,
...                     help='the bar to %(prog)s (default: %
(default)s)')
>>> parser.print_help()
usage: frobble [-h] [bar]

positional arguments:
 bar     the bar to frobble (default: 42)
```

```
options:
 -h, --help  show this help message and exit
```

As the help string supports %-formatting, if you want a literal `%` to appear in the help string, you must escape it as `%%`.

`argparse` supports silencing the help entry for certain options, by setting the `help` value to `argparse.SUPPRESS`:

```
>>> parser = argparse.ArgumentParser(prog='frobble')
>>> parser.add_argument('--foo', help=argparse.SUPPRESS)
>>> parser.print_help()
usage: frobble [-h]

options:
  -h, --help  show this help message and exit
```

metavar

When **`ArgumentParser`** generates help messages, it needs some way to refer to each expected argument. By default, **`ArgumentParser`** objects use the dest value as the “name” of each object. By default, for positional argument actions, the dest value is used directly, and for optional argument actions, the dest value is uppercased. So, a single positional argument with `dest='bar'` will be referred to as `bar`. A single optional argument `--foo` that should be followed by a single command-line argument will be referred to as `FOO`. An example:

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('--foo')
>>> parser.add_argument('bar')
>>> parser.parse_args('X --foo Y'.split())
Namespace(bar='X', foo='Y')
>>> parser.print_help()
usage:  [-h] [--foo FOO] bar
```

```
positional arguments:
 bar

options:
 -h, --help  show this help message and exit
 --foo FOO
```

An alternative name can be specified with `metavar`:

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('--foo', metavar='YYY')
>>> parser.add_argument('bar', metavar='XXX')
>>> parser.parse_args('X --foo Y'.split())
Namespace(bar='X', foo='Y')
>>> parser.print_help()
usage:  [-h] [--foo YYY] XXX

positional arguments:
 XXX

options:
 -h, --help  show this help message and exit
 --foo YYY
```

Note that `metavar` only changes the *displayed* name - the name of the attribute on the **`parse_args()`** object is still determined by the dest value.

Different values of `nargs` may cause the metavar to be used multiple times. Providing a tuple to `metavar` specifies a different display for each of the arguments:

```
>>> parser = argparse.ArgumentParser(prog='PROG')
>>> parser.add_argument('-x', nargs=2)
>>> parser.add_argument('--foo', nargs=2, metavar=('bar',
'baz'))
>>> parser.print_help()
usage: PROG [-h] [-x X X] [--foo bar baz]

options:
 -h, --help     show this help message and exit
 -x X X
 --foo bar baz
```

dest

Most `ArgumentParser` actions add some value as an attribute of the object returned by `parse_args()`. The name of this attribute is determined by the `dest` keyword argument of `add_argument()`. For positional argument actions, `dest` is normally supplied as the first argument to `add_argument()`:

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('bar')
>>> parser.parse_args(['XXX'])
Namespace(bar='XXX')
```

For optional argument actions, the value of `dest` is normally inferred from the option strings. `ArgumentParser` generates the value of `dest` by taking the first long option string and stripping away the initial `--` string. If no long option strings were supplied, `dest` will be derived from the first short option string by stripping the initial `-` character. Any internal `-` characters will be converted to `_` characters to make sure the string is a valid attribute name. The examples below illustrate this behavior:

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('-f', '--foo-bar', '--foo')
>>> parser.add_argument('-x', '-y')
>>> parser.parse_args('-f 1 -x 2'.split())
Namespace(foo_bar='1', x='2')
>>> parser.parse_args('--foo 1 -y 2'.split())
Namespace(foo_bar='1', x='2')
```

`dest` allows a custom attribute name to be provided:

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('--foo', dest='bar')
>>> parser.parse_args('--foo XXX'.split())
Namespace(bar='XXX')
```

deprecated

During a project’s lifetime, some arguments may need to be removed from the command line. Before removing them, you should inform your users that the arguments are deprecated and will be removed. The `deprecated` keyword argument of **`add_argument()`**, which defaults to `False`, specifies if the argument is deprecated and will be removed in the future. For arguments, if `deprecated` is `True`, then a warning will be printed to **`sys.stderr`** when the argument is used:

```
>>> import argparse
>>> parser = argparse.ArgumentParser(prog='snake.py')
>>> parser.add_argument('--legs', default=0, type=int,
deprecated=True)
>>> parser.parse_args([])
Namespace(legs=0)
>>> parser.parse_args(['--legs', '4'])
snake.py: warning: option '--legs' is deprecated
Namespace(legs=4)
```

Added in version 3.13.

Action classes

`Action` classes implement the Action API, a callable which returns a callable which processes arguments from the command-line. Any object which follows this API may be passed as the `action` parameter to **`add_argument()`**.

class argparse.Action(*option_strings*, *dest*, *nargs=None*, *const=None*, *default=None*, *type=None*, *choices=None*, *required=False*, *help=None*, *metavar=None*)

`Action` objects are used by an [`ArgumentParser`] to represent the information needed to parse a single argument from one or more strings from the command line. The **`Action`** class must accept the two positional arguments plus any keyword arguments passed to [`ArgumentParser.add_argument()`] except for the `action` itself.

Instances of **`Action`** (or return value of any callable to the `action` parameter) should have attributes **`dest`**, **`option_strings`**, **`default`**, **`type`**, **`required`**, **`help`**, etc. defined. The easiest way to ensure these attributes are defined is to call **`Action.__init__()`**.

__call__(*parser*, *namespace*, *values*, *option_string=None*)

`Action` instances should be callable, so subclasses must override the **`__call__()`** method, which should accept four parameters:

- *parser* - The [`ArgumentParser`] object which contains this action.
- *namespace* - The [`Namespace`] object that will be returned by [`parse_args()`]. Most actions add an attribute to this object using [`setattr()`].
- *values* - The associated command-line arguments, with any type conversions applied. Type conversions are specified with the [type] keyword argument to [`add_argument()`].
- *option_string* - The option string that was used to invoke this action. The `option_string` argument is optional, and will be absent if the action is associated with a positional argument.

The **`__call__()`** method may perform arbitrary actions, but will typically set attributes on the `namespace` based on `dest` and `values`.

format_usage()

`Action` subclasses can define a **`format_usage()`** method that takes no argument and return a string which will be used when printing the usage of the program. If such method is not provided, a sensible default will be used.

class argparse.BooleanOptionalAction

A subclass of **`Action`** for handling boolean flags with positive and negative options. Adding a single argument such as `--foo` automatically creates both `--foo` and `--no-foo` options, storing `True` and `False` respectively:

```
>>> import argparse
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('--foo',
action=argparse.BooleanOptionalAction)
>>> parser.parse_args(['--no-foo'])
Namespace(foo=False)
```

Added in version 3.9.

The parse_args() method

ArgumentParser.parse_args(*args=None*, *namespace=None*)

Convert argument strings to objects and assign them as attributes of the namespace. Return the populated namespace.

Previous calls to **`add_argument()`** determine exactly what objects are created and how they are assigned. See the documentation for **`add_argument()`** for details.

- args - List of strings to parse. The default is taken from **`sys.argv`**.

- namespace - An object to take the attributes. The default is a new empty `Namespace` object.

Option value syntax

The `parse_args()` method supports several ways of specifying the value of an option (if it takes one). In the simplest case, the option and its value are passed as two separate arguments:

```
>>> parser = argparse.ArgumentParser(prog='PROG')
>>> parser.add_argument('-x')
>>> parser.add_argument('--foo')
>>> parser.parse_args(['-x', 'X'])
Namespace(foo=None, x='X')
>>> parser.parse_args(['--foo', 'FOO'])
Namespace(foo='FOO', x=None)
```

For long options (options with names longer than a single character), the option and value can also be passed as a single command-line argument, using `=` to separate them:

```
>>> parser.parse_args(['--foo=FOO'])
Namespace(foo='FOO', x=None)
```

For short options (options only one character long), the option and its value can be concatenated:

```
>>> parser.parse_args(['-xX'])
Namespace(foo=None, x='X')
```

Several short options can be joined together, using only a single `-` prefix, as long as only the last option (or none of them) requires a value:

```
>>> parser = argparse.ArgumentParser(prog='PROG')
>>> parser.add_argument('-x', action='store_true')
>>> parser.add_argument('-y', action='store_true')
>>> parser.add_argument('-z')
>>> parser.parse_args(['-xyzZ'])
Namespace(x=True, y=True, z='Z')
```

Invalid arguments

While parsing the command line, `parse_args()` checks for a variety of errors, including ambiguous options, invalid types, invalid options, wrong number of positional arguments, etc. When it encounters such an error, it exits and prints the error along with a usage message:

```
>>> parser = argparse.ArgumentParser(prog='PROG')
>>> parser.add_argument('--foo', type=int)
>>> parser.add_argument('bar', nargs='?')

>>> # invalid type
>>> parser.parse_args(['--foo', 'spam'])
usage: PROG [-h] [--foo FOO] [bar]
PROG: error: argument --foo: invalid int value: 'spam'

>>> # invalid option
>>> parser.parse_args(['--bar'])
usage: PROG [-h] [--foo FOO] [bar]
PROG: error: no such option: --bar

>>> # wrong number of arguments
>>> parser.parse_args(['spam', 'badger'])
usage: PROG [-h] [--foo FOO] [bar]
PROG: error: extra arguments found: badger
```

Arguments containing `-`

The `parse_args()` method attempts to give errors whenever the user has clearly made a mistake, but some situations are inherently ambiguous. For example, the command-line argument `-1` could either be an attempt to specify an option or an attempt to provide a positional argument. The `parse_args()` method is cautious here: positional arguments may only begin with `-` if they look like negative numbers and there are no options in the parser that look like negative numbers:

```
>>> parser = argparse.ArgumentParser(prog='PROG')
>>> parser.add_argument('-x')
>>> parser.add_argument('foo', nargs='?')

>>> # no negative number options, so -1 is a positional
argument
>>> parser.parse_args(['-x', '-1'])
Namespace(foo=None, x='-1')

>>> # no negative number options, so -1 and -5 are positional
arguments
>>> parser.parse_args(['-x', '-1', '-5'])
Namespace(foo='-5', x='-1')

>>> parser = argparse.ArgumentParser(prog='PROG')
>>> parser.add_argument('-1', dest='one')
>>> parser.add_argument('foo', nargs='?')

>>> # negative number options present, so -1 is an option
>>> parser.parse_args(['-1', 'X'])
Namespace(foo=None, one='X')

>>> # negative number options present, so -2 is an option
>>> parser.parse_args(['-2'])
usage: PROG [-h] [-1 ONE] [foo]
PROG: error: no such option: -2

>>> # negative number options present, so both -1s are options
>>> parser.parse_args(['-1', '-1'])
usage: PROG [-h] [-1 ONE] [foo]
PROG: error: argument -1: expected one argument
```

If you have positional arguments that must begin with `-` and don’t look like negative numbers, you can insert the pseudo-argument `'--'` which tells `parse_args()` that everything after that is a positional argument:

```
>>> parser.parse_args(['--', '-f'])
Namespace(foo='-f', one=None)
```

See also the argparse howto on ambiguous arguments for more details.

Argument abbreviations (prefix matching)

The `parse_args()` method by default allows long options to be abbreviated to a prefix, if the abbreviation is unambiguous (the prefix matches a unique option):

```
>>> parser = argparse.ArgumentParser(prog='PROG')
>>> parser.add_argument('-bacon')
>>> parser.add_argument('-badger')
>>> parser.parse_args('-bac MMM'.split())
Namespace(bacon='MMM', badger=None)
>>> parser.parse_args('-bad WOOD'.split())
Namespace(bacon=None, badger='WOOD')
>>> parser.parse_args('-ba BA'.split())
usage: PROG [-h] [-bacon BACON] [-badger BADGER]
PROG: error: ambiguous option: -ba could match -badger, -bacon
```

An error is produced for arguments that could produce more than one options. This feature can be disabled by setting allow_abbrev to `False`.

Beyond `sys.argv`

Sometimes it may be useful to have an `ArgumentParser` parse arguments other than those of `sys.argv`. This can be accomplished by passing a list of strings to `parse_args()`. This is useful for testing at the interactive prompt:

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument(
...     'integers', metavar='int', type=int, choices=range(10),
...     nargs='+', help='an integer in the range 0..9')
>>> parser.add_argument(
...     '--sum', dest='accumulate', action='store_const',
const=sum,
...     default=max, help='sum the integers (default: find the
max)')
>>> parser.parse_args(['1', '2', '3', '4'])
Namespace(accumulate=<built-in function max>, integers=[1, 2,
3, 4])
>>> parser.parse_args(['1', '2', '3', '4', '--sum'])
Namespace(accumulate=<built-in function sum>, integers=[1, 2,
3, 4])
```

The Namespace object

class argparse.Namespace

Simple class used by default by `parse_args()` to create an object holding attributes and return it.

This class is deliberately simple, just an `object` subclass with a readable string representation. If you prefer to have dict-like view of the attributes, you can use the standard Python idiom, `vars()`:

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('--foo')
>>> args = parser.parse_args(['--foo', 'BAR'])
>>> vars(args)
{'foo': 'BAR'}
```

It may also be useful to have an `ArgumentParser` assign attributes to an already existing object, rather than a new `Namespace` object. This can be achieved by specifying the `namespace=` keyword argument:

```
>>> class C:
...     pass
...
>>> c = C()
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('--foo')
>>> parser.parse_args(args=['--foo', 'BAR'], namespace=c)
>>> c.foo
'BAR'
```

Other utilities

Subcommands

ArgumentParser.add_subparsers(*[, *title*][, *description*][, *prog*][, *parser_class*][, *action*][, *dest*][, *required*][, *help*][, *metavar*])

Many programs split up their functionality into a number of subcommands, for example, the `svn` program can invoke subcommands like `svn checkout`, `svn update`, and `svn commit`. Splitting up functionality this way can be a particularly good idea when a program performs several different functions which require different kinds of command-line arguments. `ArgumentParser` supports the creation of such subcommands with the **`add_subparsers()`** method. The **`add_subparsers()`** method is normally called with no arguments and returns a special action object. This object has a single method, **`add_parser()`**, which takes a command name and any **`ArgumentParser`** constructor arguments, and returns an **`ArgumentParser`** object that can be modified as usual.

Description of parameters:

- *title* - title for the sub-parser group in help output; by default “subcommands” if description is provided, otherwise uses title for positional arguments
- *description* - description for the sub-parser group in help output, by default `None`
- *prog* - usage information that will be displayed with subcommand help, by default the name of the program and any positional arguments before the subparser argument

- *parser_class* - class which will be used to create sub-parser instances, by default the class of the current parser (e.g. `ArgumentParser`)
- action - the basic type of action to be taken when this argument is encountered at the command line
- dest - name of the attribute under which subcommand name will be stored; by default `None` and no value is stored
- required - Whether or not a subcommand must be provided, by default `False` (added in 3.7)
- help - help for sub-parser group in help output, by default `None`
- metavar - string presenting available subcommands in help; by default it is `None` and presents subcommands in form {cmd1, cmd2, ..}

Some example usage:

```
>>> # create the top-level parser
>>> parser = argparse.ArgumentParser(prog='PROG')
>>> parser.add_argument('--foo', action='store_true',
help='foo help')
>>> subparsers = parser.add_subparsers(help='subcommand
help')
>>>
>>> # create the parser for the "a" command
>>> parser_a = subparsers.add_parser('a', help='a help')
>>> parser_a.add_argument('bar', type=int, help='bar help')
>>>
>>> # create the parser for the "b" command
>>> parser_b = subparsers.add_parser('b', help='b help')
>>> parser_b.add_argument('--baz', choices=('X', 'Y', 'Z'),
help='baz help')
>>>
>>> # parse some argument lists
>>> parser.parse_args(['a', '12'])
Namespace(bar=12, foo=False)
>>> parser.parse_args(['--foo', 'b', '--baz', 'Z'])
Namespace(baz='Z', foo=True)
```

Note that the object returned by `parse_args()` will only contain attributes for the main parser and the subparser that was selected by the command line (and not any other subparsers). So in the example above, when the `a` command is specified, only the `foo` and `bar` attributes are present, and when the `b` command is specified, only the `foo` and `baz` attributes are present.

Similarly, when a help message is requested from a subparser, only the help for that particular parser will be printed. The help message will not include parent parser or sibling parser messages. (A help message for each subparser command, however, can be given by supplying the `help=` argument to **`add_parser()`** as above.)

```
>>> parser.parse_args(['--help'])
usage: PROG [-h] [--foo] {a,b} ...

positional arguments:
  {a,b}   subcommand help
    a     a help
    b     b help

options:
  -h, --help  show this help message and exit
  --foo   foo help

>>> parser.parse_args(['a', '--help'])
usage: PROG a [-h] bar

positional arguments:
  bar     bar help

options:
  -h, --help  show this help message and exit

>>> parser.parse_args(['b', '--help'])
usage: PROG b [-h] [--baz {X,Y,Z}]

options:
  -h, --help     show this help message and exit
  --baz {X,Y,Z}  baz help
```

The **add_subparsers()** method also supports `title` and `description` keyword arguments. When either is present, the subparser’s commands will appear in their own group in the help output. For example:

```
>>> parser = argparse.ArgumentParser()
>>> subparsers = parser.add_subparsers(title='subcommands',
...                                    description='valid
subcommands',
...                                    help='additional
help')
>>> subparsers.add_parser('foo')
>>> subparsers.add_parser('bar')
>>> parser.parse_args(['-h'])
usage:  [-h] {foo,bar} ...

options:
  -h, --help  show this help message and exit

subcommands:
  valid subcommands

  {foo,bar}   additional help
```

Furthermore, **add_parser()** supports an additional *aliases* argument, which allows multiple strings to refer to the same subparser. This example, like `svn`, aliases `co` as a shorthand for `checkout`:

```
>>> parser = argparse.ArgumentParser()
>>> subparsers = parser.add_subparsers()
>>> checkout = subparsers.add_parser('checkout', aliases=
['co'])
>>> checkout.add_argument('foo')
>>> parser.parse_args(['co', 'bar'])
Namespace(foo='bar')
```

add_parser() supports also an additional *deprecated* argument, which allows to deprecate the subparser.

```
>>> import argparse
>>> parser = argparse.ArgumentParser(prog='chicken.py')
>>> subparsers = parser.add_subparsers()
>>> run = subparsers.add_parser('run')
>>> fly = subparsers.add_parser('fly', deprecated=True)
>>> parser.parse_args(['fly'])
chicken.py: warning: command 'fly' is deprecated
Namespace()
```

Added in version 3.13.

One particularly effective way of handling subcommands is to combine the use of the `add_subparsers()` method with calls to `set_defaults()` so that each subparser knows which Python function it should execute. For example:

```
>>> # subcommand functions
>>> def foo(args):
...     print(args.x * args.y)
...
>>> def bar(args):
...     print('((%s))' % args.z)
...
>>> # create the top-level parser
>>> parser = argparse.ArgumentParser()
>>> subparsers = parser.add_subparsers(required=True)
>>>
>>> # create the parser for the "foo" command
>>> parser_foo = subparsers.add_parser('foo')
>>> parser_foo.add_argument('-x', type=int, default=1)
>>> parser_foo.add_argument('y', type=float)
>>> parser_foo.set_defaults(func=foo)
>>>
>>> # create the parser for the "bar" command
>>> parser_bar = subparsers.add_parser('bar')
>>> parser_bar.add_argument('z')
>>> parser_bar.set_defaults(func=bar)
>>>
>>> # parse the args and call whatever function was selected
>>> args = parser.parse_args('foo 1 -x 2'.split())
>>> args.func(args)
2.0
```

```
>>>
>>> # parse the args and call whatever function was selected
>>> args = parser.parse_args('bar XYZYX'.split())
>>> args.func(args)
((XYZYX))
```

This way, you can let `parse_args()` do the job of calling the appropriate function after argument parsing is complete. Associating functions with actions like this is typically the easiest way to handle the different actions for each of your subparsers. However, if it is necessary to check the name of the subparser that was invoked, the `dest` keyword argument to the `add_subparsers()` call will work:

```
>>> parser = argparse.ArgumentParser()
>>> subparsers =
parser.add_subparsers(dest='subparser_name')
>>> subparser1 = subparsers.add_parser('1')
>>> subparser1.add_argument('-x')
>>> subparser2 = subparsers.add_parser('2')
>>> subparser2.add_argument('y')
>>> parser.parse_args(['2', 'frobble'])
Namespace(subparser_name='2', y='frobble')
```

Changed in version 3.7: New *required* keyword-only parameter.

Changed in version 3.14: Subparser’s *prog* is no longer affected by a custom usage message in the main parser.

FileType objects

class argparse.FileType(*mode='r'*, *bufsize=-1*, *encoding=None*, *errors=None*)

The `FileType` factory creates objects that can be passed to the type argument of `ArgumentParser.add_argument()`. Arguments that have `FileType` objects as their type will open command-line arguments as files with the requested modes, buffer sizes, encodings and error handling (see the `open()` function for more details):

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('--raw',
type=argparse.FileType('wb', 0))
>>> parser.add_argument('out', type=argparse.FileType('w',
encoding='UTF-8'))
>>> parser.parse_args(['--raw', 'raw.dat', 'file.txt'])
Namespace(out=<_io.TextIOWrapper name='file.txt' mode='w'
encoding='UTF-8'>, raw=<_io.FileIO name='raw.dat'
mode='wb'>)
```

FileType objects understand the pseudo-argument `'-'` and automatically convert this into `sys.stdin` for readable `FileType` objects and `sys.stdout` for writable `FileType` objects:

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('infile',
type=argparse.FileType('r'))
>>> parser.parse_args(['-'])
Namespace(infile=<_io.TextIOWrapper name='<stdin>'
encoding='UTF-8'>)
```

Note

If one argument uses *FileType* and then a subsequent argument fails, an error is reported but the file is not automatically closed. This can also clobber the output files. In this case, it would be better to wait until after the parser has run and then use the `with`-statement to manage the files.

Changed in version 3.4: Added the *encodings* and *errors* parameters.

Deprecated since version 3.14.

Argument groups

ArgumentParser.add_argument_group(*title=None*, *description=None*, *[, *argument_default*][, *conflict_handler*])

By default, [`ArgumentParser`]() groups command-line arguments into “positional arguments” and “options” when displaying help messages. When there is a better conceptual grouping of arguments than this default one, appropriate groups can be created using the **`add_argument_group()`** method:

```
>>> parser = argparse.ArgumentParser(prog='PROG',
add_help=False)
>>> group = parser.add_argument_group('group')
>>> group.add_argument('--foo', help='foo help')
>>> group.add_argument('bar', help='bar help')
>>> parser.print_help()
usage: PROG [--foo FOO] bar

group:
  bar    bar help
  --foo FOO  foo help
```

The [`add_argument_group()`]() method returns an argument group object which has an [`add_argument()`]() method just like a regular [`ArgumentParser`](). When an argument is added to the group, the parser treats it just like a normal argument, but displays the argument in a separate group for help messages. The **`add_argument_group()`** method accepts *title* and *description* arguments which can be used to customize this display:

```
>>> parser = argparse.ArgumentParser(prog='PROG',
add_help=False)
>>> group1 = parser.add_argument_group('group1', 'group1
description')
>>> group1.add_argument('foo', help='foo help')
>>> group2 = parser.add_argument_group('group2', 'group2
description')
>>> group2.add_argument('--bar', help='bar help')
>>> parser.print_help()
usage: PROG [--bar BAR] foo

group1:
  group1 description

  foo    foo help

group2:
  group2 description

  --bar BAR  bar help
```

The optional, keyword-only parameters argument_default and conflict_handler allow for finer-grained control of the behavior of the argument group. These parameters have the same meaning as in the **`ArgumentParser`** constructor, but apply specifically to the argument group rather than the entire parser.

Note that any arguments not in your user-defined groups will end up back in the usual “positional arguments” and “optional arguments” sections.

Deprecated since version 3.11, removed in version 3.14: Calling **`add_argument_group()`** on an argument group now raises an exception. This nesting was never supported, often failed to work correctly, and was unintentionally exposed through inheritance.

Deprecated since version 3.14: Passing prefix_chars to **`add_argument_group()`** is now deprecated.

Mutual exclusion

ArgumentParser.add_mutually_exclusive_group(*required=False*)

Create a mutually exclusive group. **`argparse`** will make sure that only one of the arguments in the mutually exclusive group was present on the command line:

```
>>> parser = argparse.ArgumentParser(prog='PROG')
>>> group = parser.add_mutually_exclusive_group()
>>> group.add_argument('--foo', action='store_true')
>>> group.add_argument('--bar', action='store_false')
>>> parser.parse_args(['--foo'])
Namespace(bar=True, foo=True)
>>> parser.parse_args(['--bar'])
Namespace(bar=False, foo=False)
>>> parser.parse_args(['--foo', '--bar'])
usage: PROG [-h] [--foo | --bar]
PROG: error: argument --bar: not allowed with argument --foo
```

The **`add_mutually_exclusive_group()`** method also accepts a *required* argument, to indicate that at least one of the mutually exclusive arguments is required:

```
>>> parser = argparse.ArgumentParser(prog='PROG')
>>> group =
parser.add_mutually_exclusive_group(required=True)
>>> group.add_argument('--foo', action='store_true')
>>> group.add_argument('--bar', action='store_false')
>>> parser.parse_args([])
usage: PROG [-h] (--foo | --bar)
PROG: error: one of the arguments --foo --bar is required
```

Note that currently mutually exclusive argument groups do not support the *title* and *description* arguments of **`add_argument_group()`**. However, a mutually exclusive group can be added to an argument group that has a title and description. For example:

```
>>> parser = argparse.ArgumentParser(prog='PROG')
>>> group = parser.add_argument_group('Group title', 'Group
description')
```

```
>>> exclusive_group =
group.add_mutually_exclusive_group(required=True)
>>> exclusive_group.add_argument('--foo', help='foo help')
>>> exclusive_group.add_argument('--bar', help='bar help')
>>> parser.print_help()
usage: PROG [-h] (--foo FOO | --bar BAR)

options:
  -h, --help  show this help message and exit

Group title:
  Group description

  --foo FOO   foo help
  --bar BAR   bar help
```

Deprecated since version 3.11, removed in version 3.14: Calling `add_argument_group()` or `add_mutually_exclusive_group()` on a mutually exclusive group now raises an exception. This nesting was never supported, often failed to work correctly, and was unintentionally exposed through inheritance.

Parser defaults

ArgumentParser.set_defaults(***kwargs*)

Most of the time, the attributes of the object returned by `parse_args()` will be fully determined by inspecting the command-line arguments and the argument actions. `set_defaults()` allows some additional attributes that are determined without any inspection of the command line to be added:

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('foo', type=int)
>>> parser.set_defaults(bar=42, baz='badger')
>>> parser.parse_args(['736'])
Namespace(bar=42, baz='badger', foo=736)
```

Note that defaults can be set at both the parser level using `set_defaults()` and at the argument level using `add_argument()`. If both are called for the same argument, the last default set for an argument is used:

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('--foo', default='bar')
>>> parser.set_defaults(foo='spam')
>>> parser.parse_args([])
Namespace(foo='spam')
```

Parser-level defaults can be particularly useful when working with multiple parsers. See the `add_subparsers()` method for an example of this type.

ArgumentParser.get_default(*dest*)

Get the default value for a namespace attribute, as set by either `add_argument()` or by `set_defaults()`:

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('--foo', default='badger')
>>> parser.get_default('foo')
'badger'
```

Printing help

In most typical applications, `parse_args()` will take care of formatting and printing any usage or error messages. However, several formatting methods are available:

ArgumentParser.print_usage(*file=None*)

Print a brief description of how the `ArgumentParser` should be invoked on the command line. If *file* is `None`, `sys.stdout` is assumed.

ArgumentParser.print_help(*file=None*)

Print a help message, including the program usage and information about the arguments registered with the `ArgumentParser`. If *file* is `None`, `sys.stdout` is assumed.

There are also variants of these methods that simply return a string instead of printing it:

ArgumentParser.format_usage()

Return a string containing a brief description of how the `ArgumentParser` should be invoked on the command line.

ArgumentParser.format_help()

Return a string containing a help message, including the program usage and information about the arguments registered with the `ArgumentParser`.

Partial parsing

ArgumentParser.parse_known_args(*args=None*, *namespace=None*)

Sometimes a script only needs to handle a specific set of command-line arguments, leaving any unrecognized arguments for another script or program. In these cases, the `parse_known_args()` method can be useful.

This method works similarly to `parse_args()`, but it does not raise an error for extra, unrecognized arguments. Instead, it parses the known arguments and returns a two item tuple that contains the populated namespace and the list of any unrecognized arguments.

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('--foo', action='store_true')
>>> parser.add_argument('bar')
>>> parser.parse_known_args(['--foo', '--badger', 'BAR',
'spam'])
(Namespace(bar='BAR', foo=True), ['--badger', 'spam'])
```

Warning

Prefix matching rules apply to `parse_known_args()`. The parser may consume an option even if it's just a prefix of one of its known options, instead of leaving it in the remaining arguments list.

Customizing file parsing

ArgumentParser.convert_arg_line_to_args(*arg_line*)

Arguments that are read from a file (see the *fromfile_prefix_chars* keyword argument to the `ArgumentParser` constructor) are read one argument per line. `convert_arg_line_to_args()` can be overridden for fancier reading.

This method takes a single argument *arg_line* which is a string read from the argument file. It returns a list of arguments parsed from this string. The method is called once per line read from the argument file, in order.

A useful override of this method is one that treats each space-separated word as an argument. The following example demonstrates how to do this:

```
class MyArgumentParser(argparse.ArgumentParser):
    def convert_arg_line_to_args(self, arg_line):
        return arg_line.split()
```

Exiting methods

ArgumentParser.exit(*status=0*, *message=None*)

This method terminates the program, exiting with the specified *status* and, if given, it prints a *message* to `sys.stderr` before that. The user can override this method to handle these steps differently:

```
class ErrorCatchingArgumentParser(argparse.ArgumentParser):
    def exit(self, status=0, message=None):
        if status:
            raise Exception(f'Exiting because of an error:
{message}')
        exit(status)
```

ArgumentParser.error(*message*)

This method prints a usage message, including the *message*, to `sys.stderr` and terminates the program with a status code of 2.

Intermixed parsing

ArgumentParser.parse_intermixed_args(*args=None*, *namespace=None*)

ArgumentParser.parse_known_intermixed_args(*args=None*, *namespace=None*)

A number of Unix commands allow the user to intermix optional arguments with positional arguments. The `parse_intermixed_args()` and `parse_known_intermixed_args()` methods support this parsing style.

These parsers do not support all the **`argparse`** features, and will raise exceptions if unsupported features are used. In particular, subparsers, and mutually exclusive groups that include both optionals and positionals are not supported.

The following example shows the difference between **`parse_known_args()`** and **`parse_intermixed_args()`**: the former returns `['2', '3']` as unparsed arguments, while the latter collects all the positionals into `rest`.

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('--foo')
>>> parser.add_argument('cmd')
>>> parser.add_argument('rest', nargs='*', type=int)
>>> parser.parse_known_args('doit 1 --foo bar 2 3'.split())
(Namespace(cmd='doit', foo='bar', rest=[1]), ['2', '3'])
>>> parser.parse_intermixed_args('doit 1 --foo bar 2 
3'.split())
Namespace(cmd='doit', foo='bar', rest=[1, 2, 3])
```

`parse_known_intermixed_args()` returns a two item tuple containing the populated namespace and the list of remaining argument strings. **`parse_intermixed_args()`** raises an error if there are any remaining unparsed argument strings.

Added in version 3.7.

Registering custom types or actions

ArgumentParser.register(*registry_name*, *value*, *object*)

Sometimes it’s desirable to use a custom string in error messages to provide more user-friendly output. In these cases, **`register()`** can be used to register custom actions or types with a parser and allow you to reference the type by their registered name instead of their callable name.

The **`register()`** method accepts three arguments - a *registry_name*, specifying the internal registry where the object will be stored (e.g., `action`, `type`), *value*, which is the key under which the object will be registered, and object, the callable to be registered.

The following example shows how to register a custom type with a parser:

```
>>> import argparse
>>> parser = argparse.ArgumentParser()
>>> parser.register('type', 'hexadecimal integer', lambda s:
int(s, 16))
>>> parser.add_argument('--foo', type='hexadecimal integer')
_StoreAction(option_strings=['--foo'], dest='foo',
nargs=None, const=None, default=None, type='hexadecimal
integer', choices=None, required=False, help=None,
metavar=None, deprecated=False)
>>> parser.parse_args(['--foo', '0xFA'])
Namespace(foo=250)
>>> parser.parse_args(['--foo', '1.2'])
usage: PROG [-h] [--foo FOO]
PROG: error: argument --foo: invalid 'hexadecimal integer'
value: '1.2'
```

Exceptions

exception argparse.ArgumentError

An error from creating or using an argument (optional or positional).

The string value of this exception is the message, augmented with information about the argument that caused it.

exception argparse.ArgumentTypeError

Raised when something goes wrong converting a command line string to a type.

Guides and Tutorials

- Argparse Tutorial
- Migrating `optparse` code to `argparse`

Argparse Tutorial

author:: Tshepang Mbambo

This tutorial is intended to be a gentle introduction to `argparse`, the recommended command-line parsing module in the Python standard library.

Note

The standard library includes two other libraries directly related to command-line parameter processing: the lower level `optparse` module (which may require more code to configure for a given application, but also allows an application to request behaviors that `argparse` doesn’t support), and the very low level `getopt` (which specifically serves as an equivalent to the **`getopt()`** family of functions available to C programmers). While neither of those modules is covered directly in this guide, many of the core concepts in `argparse` first originated in `optparse`, so some aspects of this tutorial will also be relevant to `optparse` users.

Concepts

Let’s show the sort of functionality that we are going to explore in this introductory tutorial by making use of the **ls** command:

```
$ ls
cpython  devguide  prog.py  pypy  rm-unused-function.patch
$ ls pypy
ctypes_configure  demo  dotviewer  include  lib_pypy  lib-
```

```
python ...
$ ls -l
total 20
drwxr-xr-x 19 wena wena 4096 Feb 18 18:51 cpython
drwxr-xr-x  4 wena wena 4096 Feb  8 12:04 devguide
-rwxr-xr-x  1 wena wena  535 Feb 19 00:05 prog.py
drwxr-xr-x 14 wena wena 4096 Feb  7 00:59 pypy
-rw-r--r--  1 wena wena  741 Feb 18 01:01 rm-unused-
function.patch
$ ls --help
Usage: ls [OPTION]... [FILE]...
List information about the FILEs (the current directory by
default).
Sort entries alphabetically if none of -cftuvSUX nor --sort is
specified.
...
```

A few concepts we can learn from the four commands:

- The **ls** command is useful when run without any options at all. It defaults to displaying the contents of the current directory.
- If we want beyond what it provides by default, we tell it a bit more. In this case, we want it to display a different directory, `pypy`. What we did is specify what is known as a positional argument. It’s named so because the program should know what to do with the value, solely based on where it appears on the command line. This concept is more relevant to a command like **cp**, whose most basic usage is `cp SRC DEST`. The first position is *what you want copied,* and the second position is *where you want it copied to*.
- Now, say we want to change behaviour of the program. In our example, we display more info for each file instead of just showing the file names. The `-l` in that case is known as an optional argument.
- That’s a snippet of the help text. It’s very useful in that you can come across a program you have never used

before, and can figure out how it works simply by reading its help text.

The basics

Let us start with a very simple example which does (almost) nothing:

```
import argparse
parser = argparse.ArgumentParser()
parser.parse_args()
```

Following is a result of running the code:

```
$ python prog.py
$ python prog.py --help
usage: prog.py [-h]

options:
  -h, --help  show this help message and exit
$ python prog.py --verbose
usage: prog.py [-h]
prog.py: error: unrecognized arguments: --verbose
$ python prog.py foo
usage: prog.py [-h]
prog.py: error: unrecognized arguments: foo
```

Here is what is happening:

- Running the script without any options results in nothing displayed to stdout. Not so useful.
- The second one starts to display the usefulness of the `argparse` module. We have done almost nothing, but already we get a nice help message.
- The `--help` option, which can also be shortened to `-h`, is the only option we get for free (i.e. no need to specify it). Specifying anything else results in an error. But even then, we do get a useful usage message, also for free.

Introducing Positional arguments

An example:

```
import argparse
parser = argparse.ArgumentParser()
parser.add_argument("echo")
args = parser.parse_args()
print(args.echo)
```

And running the code:

```
$ python prog.py
usage: prog.py [-h] echo
prog.py: error: the following arguments are required: echo
$ python prog.py --help
usage: prog.py [-h] echo

positional arguments:
  echo

options:
  -h, --help  show this help message and exit
$ python prog.py foo
foo
```

Here is what's happening:

- We've added the `add_argument()` method, which is what we use to specify which command-line options the program is willing to accept. In this case, I've named it `echo` so that it's in line with its function.
- Calling our program now requires us to specify an option.
- The `parse_args()` method actually returns some data from the options specified, in this case, `echo`.
- The variable is some form of 'magic' that `argparse` performs for free (i.e. no need to specify which variable

that value is stored in). You will also notice that its name matches the string argument given to the method, `echo`.

Note however that, although the help display looks nice and all, it currently is not as helpful as it can be. For example we see that we got `echo` as a positional argument, but we don’t know what it does, other than by guessing or by reading the source code. So, let’s make it a bit more useful:

```
import argparse
parser = argparse.ArgumentParser()
parser.add_argument("echo", help="echo the string you use
here")
args = parser.parse_args()
print(args.echo)
```

And we get:

```
$ python prog.py -h
usage: prog.py [-h] echo

positional arguments:
  echo        echo the string you use here

options:
  -h, --help  show this help message and exit
```

Now, how about doing something even more useful:

```
import argparse
parser = argparse.ArgumentParser()
parser.add_argument("square", help="display a square of a given
number")
args = parser.parse_args()
print(args.square**2)
```

Following is a result of running the code:

```
$ python prog.py 4
Traceback (most recent call last):
  File "prog.py", line 5, in <module>
```

```
    print(args.square**2)
TypeError: unsupported operand type(s) for ** or pow(): 'str'
and 'int'
```

That didn’t go so well. That’s because `argparse` treats the options we give it as strings, unless we tell it otherwise. So, let’s tell `argparse` to treat that input as an integer:

```
import argparse
parser = argparse.ArgumentParser()
parser.add_argument("square", help="display a square of a given
number",
                    type=int)
args = parser.parse_args()
print(args.square**2)
```

Following is a result of running the code:

```
$ python prog.py 4
16
$ python prog.py four
usage: prog.py [-h] square
prog.py: error: argument square: invalid int value: 'four'
```

That went well. The program now even helpfully quits on bad illegal input before proceeding.

Introducing Optional arguments

So far we have been playing with positional arguments. Let us have a look on how to add optional ones:

```
import argparse
parser = argparse.ArgumentParser()
parser.add_argument("--verbosity", help="increase output
verbosity")
args = parser.parse_args()
if args.verbosity:
    print("verbosity turned on")
```

And the output:

```
$ python prog.py --verbosity 1
verbosity turned on
$ python prog.py
$ python prog.py --help
usage: prog.py [-h] [--verbosity VERBOSITY]

options:
  -h, --help            show this help message and exit
  --verbosity VERBOSITY
                        increase output verbosity
$ python prog.py --verbosity
usage: prog.py [-h] [--verbosity VERBOSITY]
prog.py: error: argument --verbosity: expected one argument
```

Here is what is happening:

- The program is written so as to display something when `--verbosity` is specified and display nothing when not.
- To show that the option is actually optional, there is no error when running the program without it. Note that by default, if an optional argument isn't used, the relevant variable, in this case `args.verbosity`, is given `None` as a value, which is the reason it fails the truth test of the **`if`** statement.
- The help message is a bit different.
- When using the `--verbosity` option, one must also specify some value, any value.

The above example accepts arbitrary integer values for `--verbosity`, but for our simple program, only two values are actually useful, `True` or `False`. Let's modify the code accordingly:

```
import argparse
parser = argparse.ArgumentParser()
parser.add_argument("--verbose", help="increase output
verbosity",
                    action="store_true")
args = parser.parse_args()
if args.verbose:
    print("verbosity turned on")
```

And the output:

```
$ python prog.py --verbose
verbosity turned on
$ python prog.py --verbose 1
usage: prog.py [-h] [--verbose]
prog.py: error: unrecognized arguments: 1
$ python prog.py --help
usage: prog.py [-h] [--verbose]

options:
  -h, --help  show this help message and exit
  --verbose   increase output verbosity
```

Here is what is happening:

- The option is now more of a flag than something that requires a value. We even changed the name of the option to match that idea. Note that we now specify a new keyword, `action`, and give it the value `"store_true"`. This means that, if the option is specified, assign the value `True` to `args.verbose`. Not specifying it implies `False`.
- It complains when you specify a value, in true spirit of what flags actually are.
- Notice the different help text.

Short options

If you are familiar with command line usage, you will notice that I haven’t yet touched on the topic of short versions of the options. It’s quite simple:

```
import argparse
parser = argparse.ArgumentParser()
```

```
parser.add_argument("-v", "--verbose", help="increase output
verbosity",
                    action="store_true")
args = parser.parse_args()
if args.verbose:
    print("verbosity turned on")
```

And here goes:

```
$ python prog.py -v
verbosity turned on
$ python prog.py --help
usage: prog.py [-h] [-v]

options:
  -h, --help     show this help message and exit
  -v, --verbose  increase output verbosity
```

Note that the new ability is also reflected in the help text.

Combining Positional and Optional arguments

Our program keeps growing in complexity:

```
import argparse
parser = argparse.ArgumentParser()
parser.add_argument("square", type=int,
                    help="display a square of a given number")
parser.add_argument("-v", "--verbose", action="store_true",
                    help="increase output verbosity")
args = parser.parse_args()
answer = args.square**2
if args.verbose:
    print(f"the square of {args.square} equals {answer}")
else:
    print(answer)
```

And now the output:

```
$ python prog.py
usage: prog.py [-h] [-v] square
prog.py: error: the following arguments are required: square
$ python prog.py 4
16
$ python prog.py 4 --verbose
the square of 4 equals 16
$ python prog.py --verbose 4
the square of 4 equals 16
```

- We’ve brought back a positional argument, hence the complaint.
- Note that the order does not matter.

How about we give this program of ours back the ability to have multiple verbosity values, and actually get to use them:

```
import argparse
parser = argparse.ArgumentParser()
parser.add_argument("square", type=int,
                    help="display a square of a given number")
parser.add_argument("-v", "--verbosity", type=int,
                    help="increase output verbosity")
args = parser.parse_args()
answer = args.square**2
if args.verbosity == 2:
    print(f"the square of {args.square} equals {answer}")
elif args.verbosity == 1:
    print(f"{args.square}^2 == {answer}")
else:
    print(answer)
```

And the output:

```
$ python prog.py 4
16
$ python prog.py 4 -v
usage: prog.py [-h] [-v VERBOSITY] square
prog.py: error: argument -v/--verbosity: expected one argument
$ python prog.py 4 -v 1
```

```
4^2 == 16
$ python prog.py 4 -v 2
the square of 4 equals 16
$ python prog.py 4 -v 3
16
```

These all look good except the last one, which exposes a bug in our program. Let’s fix it by restricting the values the `--verbosity` option can accept:

```
import argparse
parser = argparse.ArgumentParser()
parser.add_argument("square", type=int,
                    help="display a square of a given number")
parser.add_argument("-v", "--verbosity", type=int, choices=[0,
1, 2],
                    help="increase output verbosity")
args = parser.parse_args()
answer = args.square**2
if args.verbosity == 2:
    print(f"the square of {args.square} equals {answer}")
elif args.verbosity == 1:
    print(f"{args.square}^2 == {answer}")
else:
    print(answer)
```

And the output:

```
$ python prog.py 4 -v 3
usage: prog.py [-h] [-v {0,1,2}] square
prog.py: error: argument -v/--verbosity: invalid choice: 3
(choose from 0, 1, 2)
$ python prog.py 4 -h
usage: prog.py [-h] [-v {0,1,2}] square

positional arguments:
  square                display a square of a given number

options:
  -h, --help            show this help message and exit
  -v, --verbosity {0,1,2}
                        increase output verbosity
```

Note that the change also reflects both in the error message as well as the help string.

Now, let’s use a different approach of playing with verbosity, which is pretty common. It also matches the way the CPython executable handles its own verbosity argument (check the output of `python --help`):

```
import argparse
parser = argparse.ArgumentParser()
parser.add_argument("square", type=int,
                    help="display the square of a given
number")
parser.add_argument("-v", "--verbosity", action="count",
                    help="increase output verbosity")
args = parser.parse_args()
answer = args.square**2
if args.verbosity == 2:
    print(f"the square of {args.square} equals {answer}")
elif args.verbosity == 1:
    print(f"{args.square}^2 == {answer}")
else:
    print(answer)
```

We have introduced another action, “count”, to count the number of occurrences of specific options.

```
$ python prog.py 4
16
$ python prog.py 4 -v
4^2 == 16
$ python prog.py 4 -vv
the square of 4 equals 16
$ python prog.py 4 --verbosity --verbosity
the square of 4 equals 16
$ python prog.py 4 -v 1
usage: prog.py [-h] [-v] square
prog.py: error: unrecognized arguments: 1
$ python prog.py 4 -h
usage: prog.py [-h] [-v] square
```

```
positional arguments:
  square           display a square of a given number

options:
  -h, --help       show this help message and exit
  -v, --verbosity  increase output verbosity
$ python prog.py 4 -vvv
16
```

- Yes, it’s now more of a flag (similar to `action="store_true"`) in the previous version of our script. That should explain the complaint.
- It also behaves similar to “store_true” action.
- Now here’s a demonstration of what the “count” action gives. You’ve probably seen this sort of usage before.
- And if you don’t specify the `-v` flag, that flag is considered to have `None` value.
- As should be expected, specifying the long form of the flag, we should get the same output.
- Sadly, our help output isn’t very informative on the new ability our script has acquired, but that can always be fixed by improving the documentation for our script (e.g. via the `help` keyword argument).
- That last output exposes a bug in our program.

Let’s fix:

```
import argparse
parser = argparse.ArgumentParser()
parser.add_argument("square", type=int,
                    help="display a square of a given number")
parser.add_argument("-v", "--verbosity", action="count",
                    help="increase output verbosity")
args = parser.parse_args()
answer = args.square**2

# bugfix: replace == with >=
if args.verbosity >= 2:
    print(f"the square of {args.square} equals {answer}")
elif args.verbosity >= 1:
    print(f"{args.square}^2 == {answer}")
else:
    print(answer)
```

And this is what it gives:

```
$ python prog.py 4 -vvv
the square of 4 equals 16
$ python prog.py 4 -vvvv
the square of 4 equals 16
$ python prog.py 4
Traceback (most recent call last):
  File "prog.py", line 11, in <module>
    if args.verbosity >= 2:
TypeError: '>=' not supported between instances of 'NoneType'
and 'int'
```

- First output went well, and fixes the bug we had before. That is, we want any value >= 2 to be as verbose as possible.
- Third output not so good.

Let’s fix that bug:

```
import argparse
parser = argparse.ArgumentParser()
parser.add_argument("square", type=int,
                    help="display a square of a given number")
parser.add_argument("-v", "--verbosity", action="count",
default=0,
                    help="increase output verbosity")
args = parser.parse_args()
answer = args.square**2
if args.verbosity >= 2:
    print(f"the square of {args.square} equals {answer}")
elif args.verbosity >= 1:
    print(f"{args.square}^2 == {answer}")
else:
    print(answer)
```

We’ve just introduced yet another keyword, `default`. We’ve set it to `0` in order to make it comparable to the other int values. Remember that by default, if an optional argument isn’t specified, it gets the `None` value, and that cannot be compared to an int value (hence the **`TypeError`** exception).

And:

```
$ python prog.py 4
16
```

You can go quite far just with what we’ve learned so far, and we have only scratched the surface. The **`argparse`** module is very powerful, and we’ll explore a bit more of it before we end this tutorial.

Getting a little more advanced

What if we wanted to expand our tiny program to perform other powers, not just squares:

```
import argparse
parser = argparse.ArgumentParser()
parser.add_argument("x", type=int, help="the base")
parser.add_argument("y", type=int, help="the exponent")
parser.add_argument("-v", "--verbosity", action="count",
default=0)
args = parser.parse_args()
answer = args.x**args.y
if args.verbosity >= 2:
    print(f"{args.x} to the power {args.y} equals {answer}")
elif args.verbosity >= 1:
    print(f"{args.x}^{args.y} == {answer}")
else:
    print(answer)
```

Output:

```
$ python prog.py
usage: prog.py [-h] [-v] x y
prog.py: error: the following arguments are required: x, y
$ python prog.py -h
usage: prog.py [-h] [-v] x y

positional arguments:
  x                the base
  y                the exponent

options:
  -h, --help       show this help message and exit
  -v, --verbosity
$ python prog.py 4 2 -v
4^2 == 16
```

Notice that so far we’ve been using verbosity level to *change* the text that gets displayed. The following example instead uses verbosity level to display *more* text instead:

```
import argparse
parser = argparse.ArgumentParser()
parser.add_argument("x", type=int, help="the base")
parser.add_argument("y", type=int, help="the exponent")
parser.add_argument("-v", "--verbosity", action="count",
default=0)
args = parser.parse_args()
answer = args.x**args.y
if args.verbosity >= 2:
    print(f"Running '{__file__}'")
if args.verbosity >= 1:
    print(f"{args.x}^{args.y} == ", end="")
print(answer)
```

Output:

```
$ python prog.py 4 2
16
$ python prog.py 4 2 -v
4^2 == 16
$ python prog.py 4 2 -vv
```

```
Running 'prog.py'
4^2 == 16
```

Specifying ambiguous arguments

When there is ambiguity in deciding whether an argument is positional or for an argument, `--` can be used to tell `parse_args()` that everything after that is a positional argument:

```
>>> parser = argparse.ArgumentParser(prog='PROG')
>>> parser.add_argument('-n', nargs='+')
>>> parser.add_argument('args', nargs='*')

>>> # ambiguous, so parse_args assumes it's an option
>>> parser.parse_args(['-f'])
usage: PROG [-h] [-n N [N ...]] [args ...]
PROG: error: unrecognized arguments: -f

>>> parser.parse_args(['--', '-f'])
Namespace(args=['-f'], n=None)

>>> # ambiguous, so the -n option greedily accepts arguments
>>> parser.parse_args(['-n', '1', '2', '3'])
Namespace(args=[], n=['1', '2', '3'])

>>> parser.parse_args(['-n', '1', '--', '2', '3'])
Namespace(args=['2', '3'], n=['1'])
```

Conflicting options

So far, we have been working with two methods of an `argparse.ArgumentParser` instance. Let’s introduce a third one, `add_mutually_exclusive_group()`. It allows for us to specify options that conflict with each other. Let’s also change the rest of the program so that the new functionality makes more sense: we’ll introduce the `--quiet` option, which will be the opposite of the `--verbose` one:

```
import argparse

parser = argparse.ArgumentParser()
group = parser.add_mutually_exclusive_group()
group.add_argument("-v", "--verbose", action="store_true")
group.add_argument("-q", "--quiet", action="store_true")
parser.add_argument("x", type=int, help="the base")
parser.add_argument("y", type=int, help="the exponent")
args = parser.parse_args()
answer = args.x**args.y

if args.quiet:
    print(answer)
elif args.verbose:
    print(f"{args.x} to the power {args.y} equals {answer}")
else:
    print(f"{args.x}^{args.y} == {answer}")
```

Our program is now simpler, and we’ve lost some functionality for the sake of demonstration. Anyways, here’s the output:

```
$ python prog.py 4 2
4^2 == 16
$ python prog.py 4 2 -q
16
$ python prog.py 4 2 -v
4 to the power 2 equals 16
$ python prog.py 4 2 -vq
usage: prog.py [-h] [-v | -q] x y
prog.py: error: argument -q/--quiet: not allowed with argument
-v/--verbose
$ python prog.py 4 2 -v --quiet
usage: prog.py [-h] [-v | -q] x y
prog.py: error: argument -q/--quiet: not allowed with argument
-v/--verbose
```

That should be easy to follow. I’ve added that last output so you can see the sort of flexibility you get, i.e. mixing long form options with short form ones.

Before we conclude, you probably want to tell your users the main purpose of your program, just in case they don’t know:

```
import argparse

parser = argparse.ArgumentParser(description="calculate X to
the power of Y")
group = parser.add_mutually_exclusive_group()
group.add_argument("-v", "--verbose", action="store_true")
group.add_argument("-q", "--quiet", action="store_true")
parser.add_argument("x", type=int, help="the base")
parser.add_argument("y", type=int, help="the exponent")
args = parser.parse_args()
answer = args.x**args.y

if args.quiet:
    print(answer)
elif args.verbose:
    print(f"{args.x} to the power {args.y} equals {answer}")
else:
    print(f"{args.x}^{args.y} == {answer}")
```

Note that slight difference in the usage text. Note the `[-v | -q]`, which tells us that we can either use `-v` or `-q`, but not both at the same time:

```
$ python prog.py --help
usage: prog.py [-h] [-v | -q] x y

calculate X to the power of Y

positional arguments:
  x              the base
  y              the exponent

options:
  -h, --help     show this help message and exit
  -v, --verbose
  -q, --quiet
```

How to translate the argparse output

The output of the **`argparse`** module such as its help text and error messages are all made translatable using the **`gettext`** module. This allows applications to easily localize messages produced by **`argparse`**. See also Internationalizing your programs and modules.

For instance, in this **`argparse`** output:

```
$ python prog.py --help
usage: prog.py [-h] [-v | -q] x y

calculate X to the power of Y

positional arguments:
  x               the base
  y               the exponent

options:
  -h, --help      show this help message and exit
  -v, --verbose
  -q, --quiet
```

The strings `usage:`, `positional arguments:`, `options:` and `show this help message and exit` are all translatable.

In order to translate these strings, they must first be extracted into a `.po` file. For example, using Babel [https://babel.pocoo.org/], run this command:

```
$ pybabel extract -o messages.po
/usr/lib/python3.12/argparse.py
```

This command will extract all translatable strings from the **`argparse`** module and output them into a file named `messages.po`. This command assumes that your Python installation is in `/usr/lib`.

You can find out the location of the `argparse` module on your system using this script:

```
import argparse
print(argparse.__file__)
```

Once the messages in the `.po` file are translated and the translations are installed using `gettext`, `argparse` will be able to display the translated messages.

To translate your own strings in the `argparse` output, use `gettext`.

Custom type converters

The `argparse` module allows you to specify custom type converters for your command-line arguments. This allows you to modify user input before it's stored in the `argparse.Namespace`. This can be useful when you need to pre-process the input before it is used in your program.

When using a custom type converter, you can use any callable that takes a single string argument (the argument value) and returns the converted value. However, if you need to handle more complex scenarios, you can use a custom action class with the **action** parameter instead.

For example, let's say you want to handle arguments with different prefixes and process them accordingly:

```
import argparse

parser = argparse.ArgumentParser(prefix_chars='-+')

parser.add_argument('-a', metavar='<value>', action='append',
                    type=lambda x: ('-', x))
parser.add_argument('+a', metavar='<value>', action='append',
                    type=lambda x: ('+', x))
```

```
args = parser.parse_args()
print(args)
```

Output:

```
$ python prog.py -a value1 +a value2
Namespace(a=[('-', 'value1'), ('+', 'value2')])
```

In this example, we:

- Created a parser with custom prefix characters using the `prefix_chars` parameter.
- Defined two arguments, `-a` and `+a`, which used the `type` parameter to create custom type converters to store the value in a tuple with the prefix.

Without the custom type converters, the arguments would have treated the `-a` and `+a` as the same argument, which would have been undesirable. By using custom type converters, we were able to differentiate between the two arguments.

Conclusion

The **`argparse`** module offers a lot more than shown here. Its docs are quite detailed and thorough, and full of examples. Having gone through this tutorial, you should easily digest them without feeling overwhelmed.

Migrating `optparse` code to `argparse`

The **`argparse`** module offers several higher level features not natively provided by the **`optparse`** module, including:

- Handling positional arguments.
- Supporting subcommands.
- Allowing alternative option prefixes like `+` and `/`.
- Handling zero-or-more and one-or-more style arguments.
- Producing more informative usage messages.
- Providing a much simpler interface for custom `type` and `action`.

Originally, the **`argparse`** module attempted to maintain compatibility with **`optparse`**. However, the fundamental design differences between supporting declarative command line option processing (while leaving positional argument processing to application code), and supporting both named options and positional arguments in the declarative interface mean that the API has diverged from that of `optparse` over time.

As described in Choosing an argument parsing library, applications that are currently using **`optparse`** and are happy with the way it works can just continue to use `optparse`.

Application developers that are considering migrating should also review the list of intrinsic behavioural differences described in that section before deciding whether or not migration is desirable.

For applications that do choose to migrate from `optparse` to `argparse`, the following suggestions should be helpful:

- Replace all `optparse.OptionParser.add_option()` calls with `ArgumentParser.add_argument()` calls.
- Replace `(options, args) = parser.parse_args()` with `args = parser.parse_args()` and add additional `ArgumentParser.add_argument()` calls for the positional arguments. Keep in mind that what was previously called `options`, now in the `argparse` context is called `args`.
- Replace `optparse.OptionParser.disable_interspersed_args()` by using `parse_intermixed_args()` instead of `parse_args()`.
- Replace callback actions and the `callback_*` keyword arguments with `type` or `action` arguments.
- Replace string names for `type` keyword arguments with the corresponding type objects (e.g. int, float, complex, etc).
- Replace `optparse.Values` with `Namespace` and `optparse.OptionError` and `optparse.OptionValueError` with `ArgumentError`.
- Replace strings with implicit arguments such as `%default` or `%prog` with the standard Python syntax to use dictionaries to format strings, that is, `%(default)s` and `%(prog)s`.
- Replace the OptionParser constructor `version` argument with a call to `parser.add_argument('--version', action='version', version='<the version>')`.

`optparse` — Parser for command line options

Source code: Lib/optparse.py [https://github.com/python/cpython/tree/3.14/Lib/optparse.py]

Choosing an argument parsing library

The standard library includes three argument parsing libraries:

- **`getopt`**: a module that closely mirrors the procedural C `getopt` API. Included in the standard library since before the initial Python 1.0 release.
- **`optparse`**: a declarative replacement for `getopt` that provides equivalent functionality without requiring each application to implement its own procedural option parsing logic. Included in the standard library since the Python 2.3 release.
- **`argparse`**: a more opinionated alternative to `optparse` that provides more functionality by default, at the expense of reduced application flexibility in controlling exactly how arguments are processed. Included in the standard library since the Python 2.7 and Python 3.2 releases.

In the absence of more specific argument parsing design constraints, **`argparse`** is the recommended choice for implementing command line applications, as it offers the highest level of baseline functionality with the least application level code.

`getopt` is retained almost entirely for backwards compatibility reasons. However, it also serves a niche use case as a tool for prototyping and testing command line argument handling in `getopt`-based C applications.

`optparse` should be considered as an alternative to `argparse` in the following cases:

- an application is already using `optparse` and doesn’t want to risk the subtle behavioural changes that may arise when migrating to `argparse`
- the application requires additional control over the way options and positional parameters are interleaved on the command line (including the ability to disable the interleaving feature completely)
- the application requires additional control over the incremental parsing of command line elements (while `argparse` does support this, the exact way it works in practice is undesirable for some use cases)
- the application requires additional control over the handling of options which accept parameter values that may start with `-` (such as delegated options to be passed to invoked subprocesses)
- the application requires some other command line parameter processing behavior which `argparse` does not support, but which can be implemented in terms of the lower level interface offered by `optparse`

These considerations also mean that `optparse` is likely to provide a better foundation for library authors writing third party command line argument processing libraries.

As a concrete example, consider the following two command line argument parsing configurations, the first using `optparse`, and the second using `argparse`:

```
import optparse

if __name__ == '__main__':
    parser = optparse.OptionParser()
    parser.add_option('-o', '--output')
    parser.add_option('-v', dest='verbose',
action='store_true')
    opts, args = parser.parse_args()
    process(args, output=opts.output, verbose=opts.verbose)
```

```
import argparse

if __name__ == '__main__':
    parser = argparse.ArgumentParser()
    parser.add_argument('-o', '--output')
    parser.add_argument('-v', dest='verbose',
action='store_true')
    parser.add_argument('rest', nargs='*')
    args = parser.parse_args()
    process(args.rest, output=args.output,
verbose=args.verbose)
```

The most obvious difference is that in the `optparse` version, the non-option arguments are processed separately by the application after the option processing is complete. In the `argparse` version, positional arguments are declared and processed in the same way as the named options.

However, the `argparse` version will also handle some parameter combination differently from the way the `optparse` version would handle them. For example (amongst other differences):

- supplying `-o -v` gives `output="-v"` and `verbose=False` when using `optparse`, but a usage error with `argparse` (complaining that no value has been supplied for `-o/--output`, since `-v` is interpreted as meaning the verbosity flag)
- similarly, supplying `-o --` gives `output="--"` and `args=()` when using `optparse`, but a usage error with `argparse` (also

complaining that no value has been supplied for `-o/--output`, since `--` is interpreted as terminating the option processing and treating all remaining values as positional arguments)

- supplying `-o=foo` gives `output="=foo"` when using `optparse`, but gives `output="foo"` with `argparse` (since `=` is special cased as an alternative separator for option parameter values)

Whether these differing behaviors in the `argparse` version are considered desirable or a problem will depend on the specific command line application use case.

See also

click [https://pypi.org/project/click/] is a third party argument processing library (originally based on `optparse`), which allows command line applications to be developed as a set of decorated command implementation functions.

Other third party libraries, such as typer [https://pypi.org/project/typer/] or msgspec-click [https://pypi.org/project/msgspec-click/], allow command line interfaces to be specified in ways that more effectively integrate with static checking of Python type annotations.

Introduction

`optparse` is a more convenient, flexible, and powerful library for parsing command-line options than the minimalist **`getopt`** module. **`optparse`** uses a more declarative style of command-line parsing: you create an instance of **`OptionParser`**, populate it with options, and parse the command line. **`optparse`** allows users to specify options in the conventional GNU/POSIX syntax, and additionally generates usage and help messages for you.

Here’s an example of using `optparse` in a simple script:

```
from optparse import OptionParser
...
parser = OptionParser()
parser.add_option("-f", "--file", dest="filename",
                  help="write report to FILE", metavar="FILE")
parser.add_option("-q", "--quiet",
                  action="store_false", dest="verbose",
default=True,
                  help="don't print status messages to stdout")

(options, args) = parser.parse_args()
```

With these few lines of code, users of your script can now do the “usual thing” on the command-line, for example:

```
<yourscript> --file=outfile -q
```

As it parses the command line, `optparse` sets attributes of the `options` object returned by `parse_args()` based on user-supplied command-line values. When `parse_args()` returns from parsing this command line, `options.filename` will be `"outfile"` and `options.verbose` will be `False`. `optparse` supports both long and short options, allows short options to be merged together, and allows options to be associated with their arguments in a variety of ways. Thus, the following command lines are all equivalent to the above example:

```
<yourscript> -f outfile --quiet
<yourscript> --quiet --file outfile
<yourscript> -q -foutfile
<yourscript> -qfoutfile
```

Additionally, users can run one of the following

```
<yourscript> -h
<yourscript> --help
```

and `optparse` will print out a brief summary of your script’s options:

```
Usage: <yourscript> [options]

Options:
  -h, --help            show this help message and exit
  -f FILE, --file=FILE  write report to FILE
  -q, --quiet           don't print status messages to stdout
```

where the value of *yourscript* is determined at runtime (normally from `sys.argv[0]`).

Background

`optparse` was explicitly designed to encourage the creation of programs with straightforward command-line interfaces that follow the conventions established by the **`getopt()`** family of functions available to C developers. To that end, it supports only the most common command-line syntax and semantics conventionally used under Unix. If you are unfamiliar with these conventions, reading this section will allow you to acquaint yourself with them.

Terminology

argument

a string entered on the command-line, and passed by the shell to `execl()` or `execv()`. In Python, arguments are elements of `sys.argv[1:]` (`sys.argv[0]` is the name of the program being executed). Unix shells also use the term “word”.

It is occasionally desirable to substitute an argument list other than `sys.argv[1:]`, so you should read “argument” as “an element of `sys.argv[1:]`, or of some other list provided as a substitute for `sys.argv[1:]`”.

option

an argument used to supply extra information to guide or customize the execution of a program. There are many different syntaxes for options; the traditional Unix syntax is a hyphen (“-”) followed by a single letter, e.g. `-x` or `-F`. Also, traditional Unix syntax allows multiple options to be merged into a single argument, e.g. `-x -F` is equivalent to `-xF`. The GNU project introduced `--` followed by a series of hyphen-separated words, e.g. `--file` or `--dry-run`. These are the only two option syntaxes provided by **`optparse`**.

Some other option syntaxes that the world has seen include:

- a hyphen followed by a few letters, e.g. `-pf` (this is *not* the same as multiple options merged into a single argument)
- a hyphen followed by a whole word, e.g. `-file` (this is technically equivalent to the previous syntax, but they aren’t usually seen in the same program)
- a plus sign followed by a single letter, or a few letters, or a word, e.g. `+f`, `+rgb`
- a slash followed by a letter, or a few letters, or a word, e.g. `/f`, `/file`

These option syntaxes are not supported by **`optparse`**, and they never will be. This is deliberate: the first three are non-standard on any environment, and the last only makes sense if you’re exclusively targeting Windows or certain legacy platforms (e.g. VMS, MS-DOS).

option argument

an argument that follows an option, is closely associated with that option, and is consumed from the argument list when that option is. With `optparse`, option arguments may either be in a separate argument from their option:

```
-f foo
--file foo
```

or included in the same argument:

```
-ffoo
--file=foo
```

Typically, a given option either takes an argument or it doesn't. Lots of people want an "optional option arguments" feature, meaning that some options will take an argument if they see it, and won't if they don't. This is somewhat controversial, because it makes parsing ambiguous: if `-a` takes an optional argument and `-b` is another option entirely, how do we interpret `-ab`? Because of this ambiguity, `optparse` does not support this feature.

positional argument

something leftover in the argument list after options have been parsed, i.e. after options and their arguments have been parsed and removed from the argument list.

required option

an option that must be supplied on the command-line; note that the phrase "required option" is self-contradictory in English. `optparse` doesn't prevent you from implementing required options, but doesn't give you much help at it either.

For example, consider this hypothetical command-line:

```
prog -v --report report.txt foo bar
```

`-v` and `--report` are both options. Assuming that `--report` takes one argument, `report.txt` is an option argument. `foo` and `bar` are positional arguments.

What are options for?

Options are used to provide extra information to tune or customize the execution of a program. In case it wasn't clear, options are usually *optional*. A program should be able to run just fine with no options whatsoever. (Pick a random program from the Unix or GNU toolsets. Can it run without any options at all and still make sense? The main exceptions are `find`, `tar`, and `dd`—all of which are mutant oddballs that have been rightly criticized for their non-standard syntax and confusing interfaces.)

Lots of people want their programs to have "required options". Think about it. If it's required, then it's *not optional*! If there is a piece of information that your program absolutely requires in order to run successfully, that's what positional arguments are for.

As an example of good command-line interface design, consider the humble `cp` utility, for copying files. It doesn't make much sense to try to copy files without supplying a destination and at least one source. Hence, `cp` fails if you run it with no arguments. However, it has a flexible, useful syntax that does not require any options at all:

```
cp SOURCE DEST
cp SOURCE ... DEST-DIR
```

You can get pretty far with just that. Most `cp` implementations provide a bunch of options to tweak exactly how the files are copied: you can preserve mode and modification time, avoid following symlinks, ask before clobbering existing files, etc. But none of this distracts from the core mission of `cp`, which is to copy either one file to another, or several files to another directory.

What are positional arguments for?

Positional arguments are for those pieces of information that your program absolutely, positively requires to run.

A good user interface should have as few absolute requirements as possible. If your program requires 17 distinct pieces of information in order to run successfully, it doesn’t much matter *how* you get that information from the user—most people will give up and walk away before they successfully run the program. This applies whether the user interface is a command-line, a configuration file, or a GUI: if you make that many demands on your users, most of them will simply give up.

In short, try to minimize the amount of information that users are absolutely required to supply—use sensible defaults whenever possible. Of course, you also want to make your programs reasonably flexible. That’s what options are for. Again, it doesn’t matter if they are entries in a config file, widgets in the “Preferences” dialog of a GUI, or command-line options—the more options you implement, the more flexible your program is, and the more complicated its implementation becomes. Too much flexibility has drawbacks as well, of course; too many options can overwhelm users and make your code much harder to maintain.

Tutorial

While **`optparse`** is quite flexible and powerful, it’s also straightforward to use in most cases. This section covers the code patterns that are common to any **`optparse`**-based program.

First, you need to import the OptionParser class; then, early in the main program, create an OptionParser instance:

```
from optparse import OptionParser
...
parser = OptionParser()
```

Then you can start defining options. The basic syntax is:

```
parser.add_option(opt_str, ...,
                  attr=value, ...)
```

Each option has one or more option strings, such as `-f` or `--file`, and several option attributes that tell **`optparse`** what to expect and what to do when it encounters that option on the command line.

Typically, each option will have one short option string and one long option string, e.g.:

```
parser.add_option("-f", "--file", ...)
```

You’re free to define as many short option strings and as many long option strings as you like (including zero), as long as there is at least one option string overall.

The option strings passed to **`OptionParser.add_option()`** are effectively labels for the option defined by that call. For brevity, we will frequently refer to *encountering an option* on the command line; in reality, **`optparse`** encounters *option strings* and looks up options from them.

Once all of your options are defined, instruct **`optparse`** to parse your program’s command line:

```
(options, args) = parser.parse_args()
```

(If you like, you can pass a custom argument list to **`parse_args()`**, but that’s rarely necessary: by default it uses `sys.argv[1:]`.)

`parse_args()` returns two values:

- `options`, an object containing values for all of your options—e.g. if `--file` takes a single string argument, then `options.file` will be the filename supplied by the user, or `None` if the user did not supply that option
- `args`, the list of positional arguments leftover after parsing options

This tutorial section only covers the four most important option attributes: **`action`**, **`type`**, **`dest`** (destination), and **`help`**. Of these, **`action`** is the most fundamental.

Understanding option actions

Actions tell **`optparse`** what to do when it encounters an option on the command line. There is a fixed set of actions hard-coded into **`optparse`**; adding new actions is an advanced topic covered in section Extending optparse. Most actions tell **`optparse`** to store a value in some variable—for example, take a string from the command line and store it in an attribute of `options`.

If you don’t specify an option action, **`optparse`** defaults to `store`.

The store action

The most common option action is `store`, which tells **`optparse`** to take the next argument (or the remainder of the current argument), ensure that it is of the correct type, and store it to your chosen destination.

For example:

```
parser.add_option("-f", "--file",
                  action="store", type="string",
dest="filename")
```

Now let’s make up a fake command line and ask **`optparse`** to parse it:

```
args = ["-f", "foo.txt"]
(options, args) = parser.parse_args(args)
```

When **`optparse`** sees the option string `-f`, it consumes the next argument, `foo.txt`, and stores it in `options.filename`. So, after this call to **`parse_args()`**, `options.filename` is `"foo.txt"`.

Some other option types supported by **`optparse`** are `int` and `float`. Here’s an option that expects an integer argument:

```
parser.add_option("-n", type="int", dest="num")
```

Note that this option has no long option string, which is perfectly acceptable. Also, there’s no explicit action, since the default is `store`.

Let’s parse another fake command-line. This time, we’ll jam the option argument right up against the option: since `-n42` (one argument) is equivalent to `-n 42` (two arguments), the code

```
(options, args) = parser.parse_args(["-n42"])
print(options.num)
```

will print `42`.

If you don’t specify a type, **`optparse`** assumes `string`. Combined with the fact that the default action is `store`, that means our first example can be a lot shorter:

```
parser.add_option("-f", "--file", dest="filename")
```

If you don’t supply a destination, **`optparse`** figures out a sensible default from the option strings: if the first long option string is `--foo-bar`, then the default destination is `foo_bar`. If there are no long option strings, **`optparse`** looks at the first short option string: the default destination for `-f` is `f`.

`optparse` also includes the built-in `complex` type. Adding types is covered in section Extending optparse.

Handling boolean (flag) options

Flag options—set a variable to true or false when a particular option is seen—are quite common. **`optparse`** supports them with two separate actions, `store_true` and `store_false`. For example, you might have a `verbose` flag that is turned on with `-v` and off with `-q`:

```
parser.add_option("-v", action="store_true", dest="verbose")
parser.add_option("-q", action="store_false", dest="verbose")
```

Here we have two different options with the same destination, which is perfectly OK. (It just means you have to be a bit careful when setting default values—see below.)

When **`optparse`** encounters `-v` on the command line, it sets `options.verbose` to `True`; when it encounters `-q`, `options.verbose` is set to `False`.

Other actions

Some other actions supported by **`optparse`** are:

`"store_const"`
  store a constant value, pre-set via **`Option.const`**

`"append"`
: append this option’s argument to a list

`"count"`
: increment a counter by one

`"callback"`
: call a specified function

These are covered in section Reference Guide, and section Option Callbacks.

Default values

All of the above examples involve setting some variable (the “destination”) when certain command-line options are seen. What happens if those options are never seen? Since we didn’t supply any defaults, they are all set to `None`. This is usually fine, but sometimes you want more control. **`optparse`** lets you supply a default value for each destination, which is assigned before the command line is parsed.

First, consider the verbose/quiet example. If we want **`optparse`** to set `verbose` to `True` unless `-q` is seen, then we can do this:

```
parser.add_option("-v", action="store_true", dest="verbose", 
default=True)
parser.add_option("-q", action="store_false", dest="verbose")
```

Since default values apply to the *destination* rather than to any particular option, and these two options happen to have the same destination, this is exactly equivalent:

```
parser.add_option("-v", action="store_true", dest="verbose")
parser.add_option("-q", action="store_false", dest="verbose", 
default=True)
```

Consider this:

```
parser.add_option("-v", action="store_true", dest="verbose",
default=False)
parser.add_option("-q", action="store_false", dest="verbose",
default=True)
```

Again, the default value for `verbose` will be `True`: the last default value supplied for any particular destination is the one that counts.

A clearer way to specify default values is the **`set_defaults()`** method of OptionParser, which you can call at any time before calling **`parse_args()`**:

```
parser.set_defaults(verbose=True)
parser.add_option(...)
(options, args) = parser.parse_args()
```

As before, the last value specified for a given option destination is the one that counts. For clarity, try to use one method or the other of setting default values, not both.

Generating help

`optparse`'s ability to generate help and usage text automatically is useful for creating user-friendly command-line interfaces. All you have to do is supply a **`help`** value for each option, and optionally a short usage message for your whole program. Here's an OptionParser populated with user-friendly (documented) options:

```
usage = "usage: %prog [options] arg1 arg2"
parser = OptionParser(usage=usage)
parser.add_option("-v", "--verbose",
                  action="store_true", dest="verbose",
default=True,
                  help="make lots of noise [default]")
parser.add_option("-q", "--quiet",
                  action="store_false", dest="verbose",
                  help="be vewwy quiet (I'm hunting wabbits)")
parser.add_option("-f", "--filename",
                  metavar="FILE", help="write output to FILE")
parser.add_option("-m", "--mode",
                  default="intermediate",
                  help="interaction mode: novice, intermediate,
"
                       "or expert [default: %default]")
```

If **optparse** encounters either `-h` or `--help` on the command-line, or if you just call **`parser.print_help()`**, it prints the following to standard output:

```
Usage: <yourscript> [options] arg1 arg2

Options:
  -h, --help            show this help message and exit
  -v, --verbose         make lots of noise [default]
  -q, --quiet           be vewwy quiet (I'm hunting wabbits)
  -f FILE, --filename=FILE
                        write output to FILE
  -m MODE, --mode=MODE  interaction mode: novice, intermediate,
or
                        expert [default: intermediate]
```

(If the help output is triggered by a help option, **optparse** exits after printing the help text.)

There’s a lot going on here to help **optparse** generate the best possible help message:

- the script defines its own usage message:

  ```
  usage = "usage: %prog [options] arg1 arg2"
  ```

  optparse expands `%prog` in the usage string to the name of the current program, i.e. `os.path.basename(sys.argv[0])`. The expanded string is then printed before the detailed option help.

If you don’t supply a usage string, **`optparse`** uses a bland but sensible default: `"Usage: %prog [options]"`, which is fine if your script doesn’t take any positional arguments.

- every option defines a help string, and doesn’t worry about line-wrapping—**`optparse`** takes care of wrapping lines and making the help output look good.
- options that take a value indicate this fact in their automatically generated help message, e.g. for the “mode” option:

  ```
  -m MODE, --mode=MODE
  ```

  Here, “MODE” is called the meta-variable: it stands for the argument that the user is expected to supply to `-m`/`--mode`. By default, **`optparse`** converts the destination variable name to uppercase and uses that for the meta-variable. Sometimes, that’s not what you want—for example, the `--filename` option explicitly sets `metavar="FILE"`, resulting in this automatically generated option description:

  ```
  -f FILE, --filename=FILE
  ```

  This is important for more than just saving space, though: the manually written help text uses the meta-variable `FILE` to clue the user in that there’s a connection between the semi-formal syntax `-f FILE` and the informal semantic description “write output to FILE”. This is a simple but effective way to make your help text a lot clearer and more useful for end users.
- options that have a default value can include `%default` in the help string—**`optparse`** will replace it with **`str()`** of the option’s default value. If an option has no default value (or the default value is `None`), `%default` expands to `none`.

Grouping Options

When dealing with many options, it is convenient to group these options for better help output. An `OptionParser` can contain several option groups, each of which can contain several options.

An option group is obtained using the class `OptionGroup`:

class optparse.OptionGroup(*parser*, *title*, *description=None*)

where

- parser is the `OptionParser` instance the group will be inserted in to
- title is the group title
- description, optional, is a long description of the group

`OptionGroup` inherits from **`OptionContainer`** (like `OptionParser`) and so the **`add_option()`** method can be used to add an option to the group.

Once all the options are declared, using the `OptionParser` method **`add_option_group()`** the group is added to the previously defined parser.

Continuing with the parser defined in the previous section, adding an `OptionGroup` to a parser is easy:

```
group = OptionGroup(parser, "Dangerous Options",
                    "Caution: use these options at your own
risk.  "
                    "It is believed that some of them bite.")
group.add_option("-g", action="store_true", help="Group
option.")
parser.add_option_group(group)
```

This would result in the following help output:

```
Usage: <yourscript> [options] arg1 arg2

Options:
  -h, --help            show this help message and exit
  -v, --verbose         make lots of noise [default]
  -q, --quiet           be vewwy quiet (I'm hunting wabbits)
  -f FILE, --filename=FILE
                        write output to FILE
  -m MODE, --mode=MODE  interaction mode: novice, intermediate,
or
                        expert [default: intermediate]

  Dangerous Options:
    Caution: use these options at your own risk.  It is
believed that some
    of them bite.

    -g                  Group option.
```

A bit more complete example might involve using more than one group: still extending the previous example:

```
group = OptionGroup(parser, "Dangerous Options",
                    "Caution: use these options at your own
risk.  "
                    "It is believed that some of them bite.")
group.add_option("-g", action="store_true", help="Group
option.")
parser.add_option_group(group)

group = OptionGroup(parser, "Debug Options")
group.add_option("-d", "--debug", action="store_true",
                 help="Print debug information")
group.add_option("-s", "--sql", action="store_true",
                 help="Print all SQL statements executed")
group.add_option("-e", action="store_true", help="Print every
action done")
parser.add_option_group(group)
```

that results in the following output:

```
Usage: <yourscript> [options] arg1 arg2

Options:
  -h, --help            show this help message and exit
  -v, --verbose         make lots of noise [default]
  -q, --quiet           be vewwy quiet (I'm hunting wabbits)
  -f FILE, --filename=FILE
                        write output to FILE
  -m MODE, --mode=MODE  interaction mode: novice, intermediate,
or expert
                        [default: intermediate]

  Dangerous Options:
    Caution: use these options at your own risk.  It is
believed that some
    of them bite.

    -g                  Group option.

  Debug Options:
    -d, --debug         Print debug information
    -s, --sql           Print all SQL statements executed
    -e                  Print every action done
```

Another interesting method, in particular when working programmatically with option groups is:

OptionParser.get_option_group(*opt_str*)

Return the `OptionGroup` to which the short or long option string *opt_str* (e.g. `'-o'` or `'--option'`) belongs. If there’s no such `OptionGroup`, return `None`.

Printing a version string

Similar to the brief usage string, `optparse` can also print a version string for your program. You have to supply the string as the `version` argument to OptionParser:

```
parser = OptionParser(usage="%prog [-f] [-q]", version="%prog
1.0")
```

`%prog` is expanded just like it is in `usage`. Apart from that, `version` can contain anything you like. When you supply it, `optparse` automatically adds a `--version` option to your parser. If it encounters this option on the command line, it expands your `version` string (by replacing `%prog`), prints it to stdout, and exits.

For example, if your script is called `/usr/bin/foo`:

```
$ /usr/bin/foo --version
foo 1.0
```

The following two methods can be used to print and get the `version` string:

OptionParser.print_version(*file=None*)

> Print the version message for the current program (`self.version`) to *file* (default stdout). As with `print_usage()`, any occurrence of `%prog` in `self.version` is replaced with the name of the current program. Does nothing if `self.version` is empty or undefined.

OptionParser.get_version()

> Same as `print_version()` but returns the version string instead of printing it.

How `optparse` handles errors

There are two broad classes of errors that `optparse` has to worry about: programmer errors and user errors. Programmer errors are usually erroneous calls to `OptionParser.add_option()`, e.g. invalid option strings, unknown option attributes, missing option attributes, etc. These are dealt with in the usual way: raise an exception (either `optparse.OptionError` or `TypeError`) and let the program crash.

Handling user errors is much more important, since they are guaranteed to happen no matter how stable your code is. **`optparse`** can automatically detect some user errors, such as bad option arguments (passing `-n 4x` where `-n` takes an integer argument), missing arguments (`-n` at the end of the command line, where `-n` takes an argument of any type). Also, you can call **`OptionParser.error()`** to signal an application-defined error condition:

```
(options, args) = parser.parse_args()
...
if options.a and options.b:
    parser.error("options -a and -b are mutually exclusive")
```

In either case, **`optparse`** handles the error the same way: it prints the program’s usage message and an error message to standard error and exits with error status 2.

Consider the first example above, where the user passes `4x` to an option that takes an integer:

```
$ /usr/bin/foo -n 4x
Usage: foo [options]

foo: error: option -n: invalid integer value: '4x'
```

Or, where the user fails to pass a value at all:

```
$ /usr/bin/foo -n
Usage: foo [options]

foo: error: -n option requires an argument
```

`optparse`-generated error messages take care always to mention the option involved in the error; be sure to do the same when calling **`OptionParser.error()`** from your application code.

If `optparse`’s default error-handling behaviour does not suit your needs, you’ll need to subclass OptionParser and override its **`exit()`** and/or **`error()`** methods.

Putting it all together

Here’s what `optparse`-based scripts usually look like:

```
from optparse import OptionParser
...
def main():
    usage = "usage: %prog [options] arg"
    parser = OptionParser(usage)
    parser.add_option("-f", "--file", dest="filename",
                      help="read data from FILENAME")
    parser.add_option("-v", "--verbose",
                      action="store_true", dest="verbose")
    parser.add_option("-q", "--quiet",
                      action="store_false", dest="verbose")
    ...
    (options, args) = parser.parse_args()
    if len(args) != 1:
        parser.error("incorrect number of arguments")
    if options.verbose:
        print("reading %s..." % options.filename)
    ...

if __name__ == "__main__":
    main()
```

Reference Guide

Creating the parser

The first step in using `optparse` is to create an OptionParser instance.

class optparse.OptionParser(*...*)

The OptionParser constructor has no required arguments, but a number of optional keyword arguments. You should always pass them as keyword arguments, i.e. do not rely on the order in which the arguments are declared.

`usage` (default: `"%prog [options]"`)
The usage summary to print when your program is run incorrectly or with a help option. When **`optparse`** prints the usage string, it expands `%prog` to `os.path.basename(sys.argv[0])` (or to `prog` if you passed that keyword argument). To suppress a usage message, pass the special value **`optparse.SUPPRESS_USAGE`**.

`option_list` (default: `[]`)
A list of Option objects to populate the parser with. The options in `option_list` are added after any options in `standard_option_list` (a class attribute that may be set by OptionParser subclasses), but before any version or help options. Deprecated; use **`add_option()`** after creating the parser instead.

`option_class` (default: optparse.Option)
Class to use when adding options to the parser in **`add_option()`**.

`version` (default: `None`)
A version string to print when the user supplies a version option. If you supply a true value for `version`, **`optparse`** automatically adds a version option with the single option string `--version`. The substring `%prog` is expanded the same as for `usage`.

`conflict_handler` (default: `"error"`)
Specifies what to do when options with conflicting option strings are added to the parser; see section

Conflicts between options.

`description` (default: `None`)
A paragraph of text giving a brief overview of your program. **`optparse`** reformats this paragraph to fit the current terminal width and prints it when the user requests help (after `usage`, but before the list of options).

`formatter` (default: a new **`IndentedHelpFormatter`**)
An instance of optparse.HelpFormatter that will be used for printing help text. **`optparse`** provides two concrete classes for this purpose: IndentedHelpFormatter and TitledHelpFormatter.

`add_help_option` (default: `True`)
If true, **`optparse`** will add a help option (with option strings `-h` and `--help`) to the parser.

`prog`
The string to use when expanding `%prog` in `usage` and `version` instead of `os.path.basename(sys.argv[0])`.

`epilog` (default: `None`)
A paragraph of help text to print after the option help.

Populating the parser

There are several ways to populate the parser with options. The preferred way is by using **`OptionParser.add_option()`**, as shown in section Tutorial. **`add_option()`** can be called in one of two ways:

- pass it an Option instance (as returned by **`make_option()`**)
- pass it any combination of positional and keyword arguments that are acceptable to **`make_option()`** (i.e., to

the Option constructor), and it will create the Option instance for you

The other alternative is to pass a list of pre-constructed Option instances to the OptionParser constructor, as in:

```
option_list = [
    make_option("-f", "--filename",
                action="store", type="string",
dest="filename"),
    make_option("-q", "--quiet",
                action="store_false", dest="verbose"),
    ]
parser = OptionParser(option_list=option_list)
```

(**`make_option()`** is a factory function for creating Option instances; currently it is an alias for the Option constructor. A future version of **`optparse`** may split Option into several classes, and **`make_option()`** will pick the right class to instantiate. Do not instantiate Option directly.)

Defining options

Each Option instance represents a set of synonymous command-line option strings, e.g. `-f` and `--file`. You can specify any number of short or long option strings, but you must specify at least one overall option string.

The canonical way to create an **`Option`** instance is with the **`add_option()`** method of **`OptionParser`**.

OptionParser.add_option(*option*)
OptionParser.add_option(**opt_str*, *attr=value*, *...*)

To define an option with only a short option string:

```
parser.add_option("-f", attr=value, ...)
```

And to define an option with only a long option string:

```
parser.add_option("--foo", attr=value, ...)
```

The keyword arguments define attributes of the new Option object. The most important option attribute is `action`, and it largely determines which other attributes are relevant or required. If you pass irrelevant option attributes, or fail to pass required ones, `optparse` raises an `OptionError` exception explaining your mistake.

An option's *action* determines what `optparse` does when it encounters this option on the command-line. The standard option actions hard-coded into `optparse` are:

`"store"`
: store this option's argument (default)

`"store_const"`
: store a constant value, pre-set via `Option.const`

`"store_true"`
: store `True`

`"store_false"`
: store `False`

`"append"`
: append this option's argument to a list

`"append_const"`
: append a constant value to a list, pre-set via `Option.const`

`"count"`
: increment a counter by one

`"callback"`
: call a specified function

`"help"`
print a usage message including all options and the documentation for them

(If you don’t supply an action, the default is `"store"`. For this action, you may also supply **`type`** and **`dest`** option attributes; see Standard option actions.)

As you can see, most actions involve storing or updating a value somewhere. **`optparse`** always creates a special object for this, conventionally called `options`, which is an instance of **`optparse.Values`**.

class optparse.Values
An object holding parsed argument names and values as attributes. Normally created by calling when calling **`OptionParser.parse_args()`**, and can be overridden by a custom subclass passed to the *values* argument of **`OptionParser.parse_args()`** (as described in Parsing arguments).

Option arguments (and various other values) are stored as attributes of this object, according to the **`dest`** (destination) option attribute.

For example, when you call

```
parser.parse_args()
```

one of the first things **`optparse`** does is create the `options` object:

```
options = Values()
```

If one of the options in this parser is defined with

```
parser.add_option("-f", "--file", action="store",
type="string", dest="filename")
```

and the command-line being parsed includes any of the following:

```
-ffoo
-f foo
--file=foo
--file foo
```

then **`optparse`**, on seeing this option, will do the equivalent of

```
options.filename = "foo"
```

The **`type`** and **`dest`** option attributes are almost as important as **`action`**, but **`action`** is the only one that makes sense for *all* options.

Option attributes

class optparse.Option

A single command line argument, with various attributes passed by keyword to the constructor. Normally created with **`OptionParser.add_option()`** rather than directly, and can be overridden by a custom class via the *option_class* argument to **`OptionParser`**.

The following option attributes may be passed as keyword arguments to **`OptionParser.add_option()`**. If you pass an option attribute that is not relevant to a particular option, or fail to pass a required option attribute, **`optparse`** raises **`OptionError`**.

Option.action

(default: `"store"`)

Determines **`optparse`**'s behaviour when this option is seen on the command line; the available options are documented here.

Option.type

(default: `"string"`)

The argument type expected by this option (e.g., `"string"` or `"int"`); the available option types are documented here.

Option.dest

(default: derived from option strings)

If the option’s action implies writing or modifying a value somewhere, this tells **`optparse`** where to write it: **`dest`** names an attribute of the `options` object that **`optparse`** builds as it parses the command line.

Option.default

The value to use for this option’s destination if the option is not seen on the command line. See also **`OptionParser.set_defaults()`**.

Option.nargs

(default: 1)

How many arguments of type **`type`** should be consumed when this option is seen. If > 1, **`optparse`** will store a tuple of values to **`dest`**.

Option.const

For actions that store a constant value, the constant value to store.

Option.choices

For options of type `"choice"`, the list of strings the user may choose from.

Option.callback

For options with action `"callback"`, the callable to call when this option is seen. See section Option Callbacks for detail on the arguments passed to the callable.

Option.callback_args
Option.callback_kwargs

Additional positional and keyword arguments to pass to `callback` after the four standard callback arguments.

Option.help

Help text to print for this option when listing all available options after the user supplies a **`help`** option (such as `--help`). If no help text is supplied, the option will be listed without help text. To hide this option, use the special value **`optparse.SUPPRESS_HELP`**.

Option.metavar

(default: derived from option strings)

Stand-in for the option argument(s) to use when printing help text. See section Tutorial for an example.

Standard option actions

The various option actions all have slightly different requirements and effects. Most actions have several relevant option attributes which you may specify to guide **`optparse`**'s behaviour; a few have required attributes, which you must specify for any option using that action.

- `"store"` [relevant: **`type`**, **`dest`**, **`nargs`**, **`choices`**]

  The option must be followed by an argument, which is converted to a value according to **`type`** and stored in **`dest`**. If **`nargs`** > 1, multiple arguments will be consumed from the command line; all will be converted according to **`type`** and stored to **`dest`** as a tuple. See the Standard option types section.

If **`choices`** is supplied (a list or tuple of strings), the type defaults to `"choice"`.

If **`type`** is not supplied, it defaults to `"string"`.

If **`dest`** is not supplied, **`optparse`** derives a destination from the first long option string (e.g., `--foo-bar` implies `foo_bar`). If there are no long option strings, **`optparse`** derives a destination from the first short option string (e.g., `-f` implies `f`).

Example:

```
parser.add_option("-f")
parser.add_option("-p", type="float", nargs=3,
dest="point")
```

As it parses the command line

```
-f foo.txt -p 1 -3.5 4 -fbar.txt
```

`optparse` will set

```
options.f = "foo.txt"
options.point = (1.0, -3.5, 4.0)
options.f = "bar.txt"
```

- `"store_const"` [required: **`const`**; relevant: **`dest`**]

  The value **`const`** is stored in **`dest`**.

  Example:

```
parser.add_option("-q", "--quiet",
                  action="store_const", const=0,
dest="verbose")
parser.add_option("-v", "--verbose",
                        action="store_const", const=1,
dest="verbose")
parser.add_option("--noisy",
                        action="store_const", const=2,
dest="verbose")
```

If `--noisy` is seen, **optparse** will set

```
options.verbose = 2
```

- `"store_true"` [relevant: **dest**]

  A special case of `"store_const"` that stores `True` to **dest**.

- `"store_false"` [relevant: **dest**]

  Like `"store_true"`, but stores `False`.

  Example:

  ```
  parser.add_option("--clobber", action="store_true",
  dest="clobber")
  parser.add_option("--no-clobber", action="store_false",
  dest="clobber")
  ```

- `"append"` [relevant: **type**, **dest**, **nargs**, **choices**]

  The option must be followed by an argument, which is appended to the list in **dest**. If no default value for **dest** is supplied, an empty list is automatically created when **optparse** first encounters this option on the command-line. If **nargs** > 1, multiple arguments are consumed, and a tuple of length **nargs** is appended to **dest**.

  The defaults for **type** and **dest** are the same as for the `"store"` action.

  Example:

```
parser.add_option("-t", "--tracks", action="append", 
type="int")
```

If `-t3` is seen on the command-line, **`optparse`** does the equivalent of:

```
options.tracks = []
options.tracks.append(int("3"))
```

If, a little later on, `--tracks=4` is seen, it does:

```
options.tracks.append(int("4"))
```

The `append` action calls the `append` method on the current value of the option. This means that any default value specified must have an `append` method. It also means that if the default value is non-empty, the default elements will be present in the parsed value for the option, with any values from the command line appended after those default values:

```
>>> parser.add_option("--files", action="append", default=
['~/.mypkg/defaults'])
>>> opts, args = parser.parse_args(['--files', 
'overrides.mypkg'])
>>> opts.files
['~/.mypkg/defaults', 'overrides.mypkg']
```

- `"append_const"` [required: **`const`**; relevant: **`dest`**]

  Like `"store_const"`, but the value **`const`** is appended to **`dest`**; as with `"append"`, **`dest`** defaults to `None`, and an empty list is automatically created the first time the option is encountered.

- `"count"` [relevant: **`dest`**]

  Increment the integer stored at **`dest`**. If no default value is supplied, **`dest`** is set to zero before being incremented the first time.

Example:

```
parser.add_option("-v", action="count", dest="verbosity")
```

The first time `-v` is seen on the command line, **`optparse`** does the equivalent of:

```
options.verbosity = 0
options.verbosity += 1
```

Every subsequent occurrence of `-v` results in

```
options.verbosity += 1
```

- `"callback"` [required: **`callback`**; relevant: **`type`**, **`nargs`**, **`callback_args`**, **`callback_kwargs`**]

  Call the function specified by **`callback`**, which is called as

  ```
  func(option, opt_str, value, parser, *args, **kwargs)
  ```

  See section Option Callbacks for more detail.

- `"help"`

  Prints a complete help message for all the options in the current option parser. The help message is constructed from the `usage` string passed to OptionParser's constructor and the **`help`** string passed to every option.

  If no **`help`** string is supplied for an option, it will still be listed in the help message. To omit an option entirely, use the special value **`optparse.SUPPRESS_HELP`**.

  `optparse` automatically adds a **`help`** option to all OptionParsers, so you do not normally need to create one.

Example:

```
from optparse import OptionParser, SUPPRESS_HELP

# usually, a help option is added automatically, but that
can
# be suppressed using the add_help_option argument
parser = OptionParser(add_help_option=False)

parser.add_option("-h", "--help", action="help")
parser.add_option("-v", action="store_true",
dest="verbose",
                  help="Be moderately verbose")
parser.add_option("--file", dest="filename",
                  help="Input file to read data from")
parser.add_option("--secret", help=SUPPRESS_HELP)
```

If **`optparse`** sees either `-h` or `--help` on the command line, it will print something like the following help message to stdout (assuming `sys.argv[0]` is `"foo.py"`):

```
Usage: foo.py [options]

Options:
  -h, --help        Show this help message and exit
  -v                Be moderately verbose
  --file=FILENAME   Input file to read data from
```

After printing the help message, **`optparse`** terminates your process with `sys.exit(0)`.

- `"version"`

  Prints the version number supplied to the OptionParser to stdout and exits. The version number is actually formatted and printed by the `print_version()` method of OptionParser. Generally only relevant if the `version` argument is supplied to the OptionParser constructor. As with **`help`** options, you will rarely create `version` options, since **`optparse`** automatically adds them when needed.

Standard option types

`optparse` has five built-in option types: `"string"`, `"int"`, `"choice"`, `"float"` and `"complex"`. If you need to add new option types, see section Extending optparse.

Arguments to string options are not checked or converted in any way: the text on the command line is stored in the destination (or passed to the callback) as-is.

Integer arguments (type `"int"`) are parsed as follows:

- if the number starts with `0x`, it is parsed as a hexadecimal number
- if the number starts with `0`, it is parsed as an octal number
- if the number starts with `0b`, it is parsed as a binary number
- otherwise, the number is parsed as a decimal number

The conversion is done by calling **`int()`** with the appropriate base (2, 8, 10, or 16). If this fails, so will **`optparse`**, although with a more useful error message.

`"float"` and `"complex"` option arguments are converted directly with **`float()`** and **`complex()`**, with similar error-handling.

`"choice"` options are a subtype of `"string"` options. The **`choices`** option attribute (a sequence of strings) defines the set of allowed option arguments. **`optparse.check_choice()`** compares user-supplied option arguments against this master list and raises **`OptionValueError`** if an invalid string is given.

Parsing arguments

The whole point of creating and populating an OptionParser is to call its `parse_args()` method.

OptionParser.parse_args(*args=None*, *values=None*)

Parse the command-line options found in *args*.

The input parameters are

`args`

the list of arguments to process (default: `sys.argv[1:]`)

`values`

a `Values` object to store option arguments in (default: a new instance of `Values`) – if you give an existing object, the option defaults will not be initialized on it

and the return value is a pair `(options, args)` where

`options`

the same object that was passed in as *values*, or the `optparse.Values` instance created by `optparse`

`args`

the leftover positional arguments after all options have been processed

The most common usage is to supply neither keyword argument. If you supply `values`, it will be modified with repeated `setattr()` calls (roughly one for every option argument stored to an option destination) and returned by `parse_args()`.

If `parse_args()` encounters any errors in the argument list, it calls the OptionParser's **`error()`** method with an appropriate end-user error message. This ultimately terminates your process with an exit status of 2 (the traditional Unix exit status for command-line errors).

Querying and manipulating your option parser

The default behavior of the option parser can be customized slightly, and you can also poke around your option parser and see what’s there. OptionParser provides several methods to help you out:

OptionParser.disable_interspersed_args()

Set parsing to stop on the first non-option. For example, if `-a` and `-b` are both simple options that take no arguments, `optparse` normally accepts this syntax:

```
prog -a arg1 -b arg2
```

and treats it as equivalent to

```
prog -a -b arg1 arg2
```

To disable this feature, call `disable_interspersed_args()`. This restores traditional Unix syntax, where option parsing stops with the first non-option argument.

Use this if you have a command processor which runs another command which has options of its own and you want to make sure these options don’t get confused. For example, each command might have a different set of options.

OptionParser.enable_interspersed_args()

Set parsing to not stop on the first non-option, allowing interspersing switches with command arguments. This is the default behavior.

OptionParser.get_option(*opt_str*)

Returns the Option instance with the option string *opt_str*, or `None` if no options have that option string.

OptionParser.has_option(*opt_str*)

Return `True` if the OptionParser has an option with option string *opt_str* (e.g., `-q` or `--verbose`).

OptionParser.remove_option(*opt_str*)

If the `OptionParser` has an option corresponding to *opt_str*, that option is removed. If that option provided any other option strings, all of those option strings become invalid. If *opt_str* does not occur in any option belonging to this `OptionParser`, raises `ValueError`.

Conflicts between options

If you’re not careful, it’s easy to define options with conflicting option strings:

```
parser.add_option("-n", "--dry-run", ...)
...
parser.add_option("-n", "--noisy", ...)
```

(This is particularly true if you’ve defined your own OptionParser subclass with some standard options.)

Every time you add an option, `optparse` checks for conflicts with existing options. If it finds any, it invokes the current conflict-handling mechanism. You can set the conflict-handling mechanism either in the constructor:

```
parser = OptionParser(..., conflict_handler=handler)
```

or with a separate call:

```
parser.set_conflict_handler(handler)
```

The available conflict handlers are:

`"error"` (default)

assume option conflicts are a programming error and raise `OptionConflictError`

`"resolve"`
resolve option conflicts intelligently (see below)

As an example, let’s define an `OptionParser` that resolves conflicts intelligently and add conflicting options to it:

```
parser = OptionParser(conflict_handler="resolve")
parser.add_option("-n", "--dry-run", ..., help="do no harm")
parser.add_option("-n", "--noisy", ..., help="be noisy")
```

At this point, `optparse` detects that a previously added option is already using the `-n` option string. Since `conflict_handler` is `"resolve"`, it resolves the situation by removing `-n` from the earlier option’s list of option strings. Now `--dry-run` is the only way for the user to activate that option. If the user asks for help, the help message will reflect that:

```
Options:
  --dry-run     do no harm
  ...
  -n, --noisy   be noisy
```

It’s possible to whittle away the option strings for a previously added option until there are none left, and the user has no way of invoking that option from the command-line. In that case, `optparse` removes that option completely, so it doesn’t show up in help text or anywhere else. Carrying on with our existing OptionParser:

```
parser.add_option("--dry-run", ..., help="new dry-run option")
```

At this point, the original `-n`/`--dry-run` option is no longer accessible, so `optparse` removes it, leaving this help text:

```
Options:
  ...
  -n, --noisy   be noisy
  --dry-run     new dry-run option
```

Cleanup

OptionParser instances have several cyclic references. This should not be a problem for Python’s garbage collector, but you may wish to break the cyclic references explicitly by calling **`destroy()`** on your OptionParser once you are done with it. This is particularly useful in long-running applications where large object graphs are reachable from your OptionParser.

Other methods

OptionParser supports several other public methods:

OptionParser.set_usage(*usage*)

> Set the usage string according to the rules described above for the `usage` constructor keyword argument. Passing `None` sets the default usage string; use **`optparse.SUPPRESS_USAGE`** to suppress a usage message.

OptionParser.print_usage(*file=None*)

> Print the usage message for the current program (`self.usage`) to *file* (default stdout). Any occurrence of the string `%prog` in `self.usage` is replaced with the name of the current program. Does nothing if `self.usage` is empty or not defined.

OptionParser.get_usage()

> Same as **`print_usage()`** but returns the usage string instead of printing it.

OptionParser.set_defaults(*dest=value, ...*)

Set default values for several option destinations at once. Using `set_defaults()` is the preferred way to set default values for options, since multiple options can share the same destination. For example, if several “mode” options all set the same destination, any one of them can set the default, and the last one wins:

```
parser.add_option("--advanced", action="store_const",
                  dest="mode", const="advanced",
                  default="novice")    # overridden below
parser.add_option("--novice", action="store_const",
                  dest="mode", const="novice",
                  default="advanced")  # overrides above
setting
```

To avoid this confusion, use `set_defaults()`:

```
parser.set_defaults(mode="advanced")
parser.add_option("--advanced", action="store_const",
                  dest="mode", const="advanced")
parser.add_option("--novice", action="store_const",
                  dest="mode", const="novice")
```

Option Callbacks

When `optparse`’s built-in actions and types aren’t quite enough for your needs, you have two choices: extend `optparse` or define a callback option. Extending `optparse` is more general, but overkill for a lot of simple cases. Quite often a simple callback is all you need.

There are two steps to defining a callback option:

- define the option itself using the `"callback"` action
- write the callback; this is a function (or method) that takes at least four arguments, as described below

Defining a callback option

As always, the easiest way to define a callback option is by using the `OptionParser.add_option()` method. Apart from `action`, the only option attribute you must specify is `callback`, the function to call:

```
parser.add_option("-c", action="callback",
callback=my_callback)
```

`callback` is a function (or other callable object), so you must have already defined `my_callback()` when you create this callback option. In this simple case, `optparse` doesn't even know if `-c` takes any arguments, which usually means that the option takes no arguments—the mere presence of `-c` on the command-line is all it needs to know. In some circumstances, though, you might want your callback to consume an arbitrary number of command-line arguments. This is where writing callbacks gets tricky; it's covered later in this section.

`optparse` always passes four particular arguments to your callback, and it will only pass additional arguments if you specify them via `callback_args` and `callback_kwargs`. Thus, the minimal callback function signature is:

```
def my_callback(option, opt, value, parser):
```

The four arguments to a callback are described below.

There are several other option attributes that you can supply when you define a callback option:

`type`
: has its usual meaning: as with the `"store"` or `"append"` actions, it instructs `optparse` to consume one argument and convert it to `type`. Rather than storing the converted value(s) anywhere, though, `optparse` passes it to your callback function.

`nargs`

also has its usual meaning: if it is supplied and > 1, **`optparse`** will consume **`nargs`** arguments, each of which must be convertible to **`type`**. It then passes a tuple of converted values to your callback.

`callback_args`

a tuple of extra positional arguments to pass to the callback

`callback_kwargs`

a dictionary of extra keyword arguments to pass to the callback

How callbacks are called

All callbacks are called as follows:

```
func(option, opt_str, value, parser, *args, **kwargs)
```

where

`option`

is the Option instance that's calling the callback

`opt_str`

is the option string seen on the command-line that's triggering the callback. (If an abbreviated long option was used, `opt_str` will be the full, canonical option string —e.g. if the user puts `--foo` on the command-line as an abbreviation for `--foobar`, then `opt_str` will be `"--foobar"`.)

`value`

is the argument to this option seen on the command-line. **`optparse`** will only expect an argument if **`type`** is set; the type of `value` will be the type implied by the option's type.

If **`type`** for this option is `None` (no argument expected), then `value` will be `None`. If **`nargs`** > 1, `value` will be a tuple of values of the appropriate type.

`parser`

is the OptionParser instance driving the whole thing, mainly useful because you can access some other interesting data through its instance attributes:

`parser.largs`

the current list of leftover arguments, ie. arguments that have been consumed but are neither options nor option arguments. Feel free to modify `parser.largs`, e.g. by adding more arguments to it. (This list will become `args`, the second return value of **`parse_args()`**.)

`parser.rargs`

the current list of remaining arguments, ie. with `opt_str` and `value` (if applicable) removed, and only the arguments following them still there. Feel free to modify `parser.rargs`, e.g. by consuming more arguments.

`parser.values`

the object where option values are by default stored (an instance of optparse.OptionValues). This lets callbacks use the same mechanism as the rest of **`optparse`** for storing option values; you don't need to mess around with globals or closures. You can also access or modify the value(s) of any options already encountered on the command-line.

`args`

is a tuple of arbitrary positional arguments supplied via the **`callback_args`** option attribute.

`kwargs`

is a dictionary of arbitrary keyword arguments supplied via `callback_kwargs`.

Raising errors in a callback

The callback function should raise `OptionValueError` if there are any problems with the option or its argument(s). `optparse` catches this and terminates the program, printing the error message you supply to stderr. Your message should be clear, concise, accurate, and mention the option at fault. Otherwise, the user will have a hard time figuring out what they did wrong.

Callback example 1: trivial callback

Here’s an example of a callback option that takes no arguments, and simply records that the option was seen:

```
def record_foo_seen(option, opt_str, value, parser):
    parser.values.saw_foo = True

parser.add_option("--foo", action="callback",
callback=record_foo_seen)
```

Of course, you could do that with the `"store_true"` action.

Callback example 2: check option order

Here’s a slightly more interesting example: record the fact that `-a` is seen, but blow up if it comes after `-b` in the command-line.

```
def check_order(option, opt_str, value, parser):
    if parser.values.b:
        raise OptionValueError("can't use -a after -b")
    parser.values.a = 1
...
parser.add_option("-a", action="callback",
callback=check_order)
parser.add_option("-b", action="store_true", dest="b")
```

Callback example 3: check option order (generalized)

If you want to reuse this callback for several similar options (set a flag, but blow up if `-b` has already been seen), it needs a bit of work: the error message and the flag that it sets must be generalized.

```
def check_order(option, opt_str, value, parser):
    if parser.values.b:
        raise OptionValueError("can't use %s after -b" %
opt_str)
    setattr(parser.values, option.dest, 1)
...
parser.add_option("-a", action="callback",
callback=check_order, dest='a')
parser.add_option("-b", action="store_true", dest="b")
parser.add_option("-c", action="callback",
callback=check_order, dest='c')
```

Callback example 4: check arbitrary condition

Of course, you could put any condition in there—you’re not limited to checking the values of already-defined options. For example, if you have options that should not be called when the moon is full, all you have to do is this:

```
def check_moon(option, opt_str, value, parser):
    if is_moon_full():
        raise OptionValueError("%s option invalid when moon is
full"
                               % opt_str)
    setattr(parser.values, option.dest, 1)
...
parser.add_option("--foo",
                  action="callback", callback=check_moon,
dest="foo")
```

(The definition of `is_moon_full()` is left as an exercise for the reader.)

Callback example 5: fixed arguments

Things get slightly more interesting when you define callback options that take a fixed number of arguments. Specifying that a callback option takes arguments is similar to defining a `"store"` or `"append"` option: if you define **`type`**, then the option takes one argument that must be convertible to that type; if you further define **`nargs`**, then the option takes **`nargs`** arguments.

Here’s an example that just emulates the standard `"store"` action:

```
def store_value(option, opt_str, value, parser):
    setattr(parser.values, option.dest, value)
...
parser.add_option("--foo",
                  action="callback", callback=store_value,
                  type="int", nargs=3, dest="foo")
```

Note that **`optparse`** takes care of consuming 3 arguments and converting them to integers for you; all you have to do is store them. (Or whatever; obviously you don’t need a callback for this example.)

Callback example 6: variable arguments

Things get hairy when you want an option to take a variable number of arguments. For this case, you must write a callback, as **`optparse`** doesn’t provide any built-in capabilities for it. And you have to deal with certain intricacies of conventional Unix command-line parsing that **`optparse`** normally handles for you. In particular, callbacks should implement the conventional rules for bare `--` and `-` arguments:

- either `--` or `-` can be option arguments
- bare `--` (if not the argument to some option): halt command-line processing and discard the `--`
- bare `-` (if not the argument to some option): halt command-line processing but keep the `-` (append it to `parser.largs`)

If you want an option that takes a variable number of arguments, there are several subtle, tricky issues to worry about. The exact implementation you choose will be based on which trade-offs you’re willing to make for your application (which is why **`optparse`** doesn’t support this sort of thing directly).

Nevertheless, here’s a stab at a callback for an option with variable arguments:

```
def vararg_callback(option, opt_str, value, parser):
    assert value is None
    value = []

    def floatable(str):
        try:
            float(str)
            return True
        except ValueError:
            return False

    for arg in parser.rargs:
        # stop on --foo like options
        if arg[:2] == "--" and len(arg) > 2:
            break
        # stop on -a, but not on -3 or -3.0
        if arg[:1] == "-" and len(arg) > 1 and not 
floatable(arg):
            break
        value.append(arg)
    del parser.rargs[:len(value)]
    setattr(parser.values, option.dest, value)

...
parser.add_option("-c", "--callback", dest="vararg_attr",
                  action="callback", callback=vararg_callback)
```

Extending `optparse`

Since the two major controlling factors in how `optparse` interprets command-line options are the action and type of each option, the most likely direction of extension is to add new actions and new types.

Adding new types

To add new types, you need to define your own subclass of `optparse`’s `Option` class. This class has a couple of attributes that define `optparse`’s types: `TYPES` and `TYPE_CHECKER`.

Option.TYPES

A tuple of type names; in your subclass, simply define a new tuple `TYPES` that builds on the standard one.

Option.TYPE_CHECKER

A dictionary mapping type names to type-checking functions. A type-checking function has the following signature:

```
def check_mytype(option, opt, value)
```

where `option` is an `Option` instance, `opt` is an option string (e.g., `-f`), and `value` is the string from the command line that must be checked and converted to your desired type. `check_mytype()` should return an object of the hypothetical type `mytype`. The value returned by a type-checking function will wind up in the OptionValues instance returned by `OptionParser.parse_args()`, or be passed to a callback as the `value` parameter.

Your type-checking function should raise `OptionValueError` if it encounters any problems. `OptionValueError` takes a single string argument, which is passed as-is to `OptionParser`’s **`error()`** method, which in turn prepends the program name and the string `"error:"` and prints everything to stderr before terminating the process.

Here’s a silly example that demonstrates adding a `"complex"` option type to parse Python-style complex numbers on the command line. (This is even sillier than it used to be, because `optparse` 1.3 added built-in support for complex numbers, but never mind.)

First, the necessary imports:

```
from copy import copy
from optparse import Option, OptionValueError
```

You need to define your type-checker first, since it’s referred to later (in the `TYPE_CHECKER` class attribute of your Option subclass):

```
def check_complex(option, opt, value):
    try:
        return complex(value)
    except ValueError:
        raise OptionValueError(
            "option %s: invalid complex value: %r" % (opt,
value))
```

Finally, the Option subclass:

```
class MyOption (Option):
    TYPES = Option.TYPES + ("complex",)
    TYPE_CHECKER = copy(Option.TYPE_CHECKER)
    TYPE_CHECKER["complex"] = check_complex
```

(If we didn’t make a `copy()` of `Option.TYPE_CHECKER`, we would end up modifying the `TYPE_CHECKER` attribute of `optparse`’s Option class. This being Python, nothing stops you from doing that except good manners and common sense.)

That’s it! Now you can write a script that uses the new option type just like any other `optparse`-based script, except you have to instruct your OptionParser to use MyOption instead of Option:

```
parser = OptionParser(option_class=MyOption)
parser.add_option("-c", type="complex")
```

Alternately, you can build your own option list and pass it to OptionParser; if you don’t use **`add_option()`** in the above way, you don’t need to tell OptionParser which option class to use:

```
option_list = [MyOption("-c", action="store", type="complex",
dest="c")]
parser = OptionParser(option_list=option_list)
```

Adding new actions

Adding new actions is a bit trickier, because you have to understand that `optparse` has a couple of classifications for actions:

“store” actions
: actions that result in `optparse` storing a value to an attribute of the current OptionValues instance; these options require a `dest` attribute to be supplied to the Option constructor.

“typed” actions

actions that take a value from the command line and expect it to be of a certain type; or rather, a string that can be converted to a certain type. These options require a `type` attribute to the Option constructor.

These are overlapping sets: some default “store” actions are `"store"`, `"store_const"`, `"append"`, and `"count"`, while the default “typed” actions are `"store"`, `"append"`, and `"callback"`.

When you add an action, you need to categorize it by listing it in at least one of the following class attributes of Option (all are lists of strings):

Option.ACTIONS
: All actions must be listed in ACTIONS.

Option.STORE_ACTIONS
: “store” actions are additionally listed here.

Option.TYPED_ACTIONS
: “typed” actions are additionally listed here.

Option.ALWAYS_TYPED_ACTIONS
: Actions that always take a type (i.e. whose options always take a value) are additionally listed here. The only effect of this is that `optparse` assigns the default type, `"string"`, to options with no explicit type whose action is listed in `ALWAYS_TYPED_ACTIONS`.

In order to actually implement your new action, you must override Option’s **`take_action()`** method and add a case that recognizes your action.

For example, let’s add an `"extend"` action. This is similar to the standard `"append"` action, but instead of taking a single value from the command-line and appending it to an existing list, `"extend"` will take multiple values in a single comma-delimited string, and extend an existing list with them. That is, if `--names` is an `"extend"` option of type `"string"`, the command line

```
--names=foo,bar --names blah --names ding,dong
```

would result in a list

```
["foo", "bar", "blah", "ding", "dong"]
```

Again we define a subclass of Option:

```
class MyOption(Option):

    ACTIONS = Option.ACTIONS + ("extend",)
    STORE_ACTIONS = Option.STORE_ACTIONS + ("extend",)
    TYPED_ACTIONS = Option.TYPED_ACTIONS + ("extend",)
    ALWAYS_TYPED_ACTIONS = Option.ALWAYS_TYPED_ACTIONS +
("extend",)

    def take_action(self, action, dest, opt, value, values,
parser):
        if action == "extend":
            lvalue = value.split(",")
            values.ensure_value(dest, []).extend(lvalue)
        else:
            Option.take_action(
                self, action, dest, opt, value, values, parser)
```

Features of note:

- `"extend"` both expects a value on the command-line and stores that value somewhere, so it goes in both `STORE_ACTIONS` and `TYPED_ACTIONS`.
- to ensure that `optparse` assigns the default type of `"string"` to `"extend"` actions, we put the `"extend"` action in `ALWAYS_TYPED_ACTIONS` as well.

- **`MyOption.take_action()`** implements just this one new action, and passes control back to **`Option.take_action()`** for the standard `optparse` actions.

- `values` is an instance of the optparse_parser.Values class, which provides the very useful **`ensure_value()`** method. **`ensure_value()`** is essentially `getattr()` with a safety valve; it is called as

  ```
  values.ensure_value(attr, value)
  ```

  If the `attr` attribute of `values` doesn't exist or is `None`, then ensure_value() first sets it to `value`, and then returns `value`. This is very handy for actions like `"extend"`, `"append"`, and `"count"`, all of which accumulate data in a variable and expect that variable to be of a certain type (a list for the first two, an integer for the latter). Using **`ensure_value()`** means that scripts using your action don't have to worry about setting a default value for the option destinations in question; they can just leave the default as `None` and **`ensure_value()`** will take care of getting it right when it's needed.

Exceptions

exception optparse.OptionError

Raised if an `Option` instance is created with invalid or inconsistent arguments.

exception optparse.OptionConflictError

Raised if conflicting options are added to an `OptionParser`.

exception optparse.OptionValueError

Raised if an invalid option value is encountered on the command line.

exception optparse.BadOptionError

Raised if an invalid option is passed on the command line.

exception optparse.AmbiguousOptionError

Raised if an ambiguous option is passed on the command line.

`getpass` — Portable password input

Source code: Lib/getpass.py
[https://github.com/python/cpython/tree/3.14/Lib/getpass.py]

Availability: not WASI.

This module does not work or is not available on WebAssembly. See WebAssembly platforms for more information.

The **`getpass`** module provides two functions:

getpass.getpass(*prompt='Password: '*, *stream=None*, *, *echo_char=None*)

Prompt the user for a password without echoing. The user is prompted using the string *prompt*, which defaults to `'Password: '`. On Unix, the prompt is written to the file-like object *stream* using the replace error handler if needed. *stream* defaults to the controlling terminal (`/dev/tty`) or if that is unavailable to `sys.stderr` (this argument is ignored on Windows).

The *echo_char* argument controls how user input is displayed while typing. If *echo_char* is `None` (default), input remains hidden. Otherwise, *echo_char* must be a single printable ASCII character and each typed character is replaced by it. For example, `echo_char='*'` will display asterisks instead of the actual input.

If echo free input is unavailable getpass() falls back to printing a warning message to *stream* and reading from `sys.stdin` and issuing a **`GetPassWarning`**.

Note

If you call getpass from within IDLE, the input may be done in the terminal you launched IDLE from rather than the idle window itself.

Note

On Unix systems, when *echo_char* is set, the terminal will be configured to operate in *noncanonical mode* *[https://manpages.debian.org/termios(3)#Canonical_and_noncanonical_mode]*. In particular, this means that line editing shortcuts such as `Ctrl`+`U` will not work and may insert unexpected characters into the input.

Changed in version 3.14: Added the *echo_char* parameter for keyboard feedback.

exception getpass.GetPassWarning

A **`UserWarning`** subclass issued when password input may be echoed.

getpass.getuser()

Return the “login name” of the user.

This function checks the environment variables **`LOGNAME`**, **`USER`**, **`LNAME`** and **`USERNAME`**, in order, and returns the value of the first one which is set to a non-empty string. If none are set, the login name from the password database is returned on systems which support the **`pwd`** module, otherwise, an **`OSError`** is raised.

In general, this function should be preferred over `os.getlogin()`.

Changed in version 3.13: Previously, various exceptions beyond just `OSError` were raised.

`fileinput` — Iterate over lines from multiple input streams

Source code: Lib/fileinput.py
[https://github.com/python/cpython/tree/3.14/Lib/fileinput.py]

This module implements a helper class and functions to quickly write a loop over standard input or a list of files. If you just want to read or write one file see **`open()`**.

The typical use is:

```
import fileinput
for line in fileinput.input(encoding="utf-8"):
    process(line)
```

This iterates over the lines of all files listed in `sys.argv[1:]`, defaulting to `sys.stdin` if the list is empty. If a filename is `'-'`, it is also replaced by `sys.stdin` and the optional arguments *mode* and *openhook* are ignored. To specify an alternative list of filenames, pass it as the first argument to **`input()`**. A single file name is also allowed.

All files are opened in text mode by default, but you can override this by specifying the *mode* parameter in the call to **`input()`** or **`FileInput`**. If an I/O error occurs during opening or reading a file, **`OSError`** is raised.

Changed in version 3.3: **`IOError`** used to be raised; it is now an alias of **`OSError`**.

If `sys.stdin` is used more than once, the second and further use will return no lines, except perhaps for interactive use, or if it has been explicitly reset (e.g. using `sys.stdin.seek(0)`).

Empty files are opened and immediately closed; the only time their presence in the list of filenames is noticeable at all is when the last file opened is empty.

Lines are returned with any newlines intact, which means that the last line in a file may not have one.

You can control how files are opened by providing an opening hook via the *openhook* parameter to **`fileinput.input()`** or **`FileInput()`**. The hook must be a function that takes two arguments, *filename* and *mode*, and returns an accordingly opened file-like object. If *encoding* and/or *errors* are specified, they will be passed to the hook as additional keyword arguments. This module provides a **`hook_compressed()`** to support compressed files.

The following function is the primary interface of this module:

fileinput.input(*files=None*, *inplace=False*, *backup=''*, ***, *mode='r'*, *openhook=None*, *encoding=None*, *errors=None*)

> Create an instance of the **`FileInput`** class. The instance will be used as global state for the functions of this module, and is also returned to use during iteration. The parameters to this function will be passed along to the constructor of the **`FileInput`** class.
>
> The **`FileInput`** instance can be used as a context manager in the **`with`** statement. In this example, *input* is closed after the **`with`** statement is exited, even if an exception occurs:

```
with fileinput.input(files=('spam.txt', 'eggs.txt'),
encoding="utf-8") as f:
    for line in f:
        process(line)
```

Changed in version 3.2: Can be used as a context manager.

Changed in version 3.8: The keyword parameters *mode* and *openhook* are now keyword-only.

Changed in version 3.10: The keyword-only parameter *encoding* and *errors* are added.

The following functions use the global state created by **`fileinput.input()`**; if there is no active state, **`RuntimeError`** is raised.

fileinput.filename()

Return the name of the file currently being read. Before the first line has been read, returns `None`.

fileinput.fileno()

Return the integer “file descriptor” for the current file. When no file is opened (before the first line and between files), returns `-1`.

fileinput.lineno()

Return the cumulative line number of the line that has just been read. Before the first line has been read, returns `0`. After the last line of the last file has been read, returns the line number of that line.

fileinput.filelineno()

Return the line number in the current file. Before the first line has been read, returns `0`. After the last line of the last file has been read, returns the line number of that line within the file.

fileinput.isfirstline()

Return `True` if the line just read is the first line of its file, otherwise return `False`.

fileinput.isstdin()

Return `True` if the last line was read from `sys.stdin`, otherwise return `False`.

fileinput.nextfile()

Close the current file so that the next iteration will read the first line from the next file (if any); lines not read from the file will not count towards the cumulative line count. The filename is not changed until after the first line of the next file has been read. Before the first line has been read, this function has no effect; it cannot be used to skip the first file. After the last line of the last file has been read, this function has no effect.

fileinput.close()

Close the sequence.

The class which implements the sequence behavior provided by the module is available for subclassing as well:

class fileinput.FileInput(*files=None*, *inplace=False*, *backup=''*, *, *mode='r'*, *openhook=None*, *encoding=None*, *errors=None*)

Class **`FileInput`** is the implementation; its methods **`filename()`**, **`fileno()`**, **`lineno()`**, **`filelineno()`**, **`isfirstline()`**, **`isstdin()`**, **`nextfile()`** and **`close()`** correspond to the functions of the same name in the module. In addition it is iterable and has a **`readline()`** method which returns the next input line. The sequence must be accessed in strictly sequential order; random access and `readline()` cannot be mixed.

With *mode* you can specify which file mode will be passed to `open()`. It must be one of `'r'` and `'rb'`.

The *openhook*, when given, must be a function that takes two arguments, *filename* and *mode*, and returns an accordingly opened file-like object. You cannot use *inplace* and *openhook* together.

You can specify *encoding* and *errors* that is passed to `open()` or *openhook*.

A `FileInput` instance can be used as a context manager in the `with` statement. In this example, *input* is closed after the **`with`** statement is exited, even if an exception occurs:

```
with FileInput(files=('spam.txt', 'eggs.txt')) as input:
    process(input)
```

Changed in version 3.2: Can be used as a context manager.

Changed in version 3.8: The keyword parameter *mode* and *openhook* are now keyword-only.

Changed in version 3.10: The keyword-only parameter *encoding* and *errors* are added.

Changed in version 3.11: The `'rU'` and `'U'` modes and the **`__getitem__()`** method have been removed.

Optional in-place filtering: if the keyword argument `inplace=True` is passed to `fileinput.input()` or to the `FileInput` constructor, the file is moved to a backup file and standard output is directed to the input file (if a file of the same name as the backup file already exists, it will be replaced silently). This makes it possible to write a filter that rewrites its input file in place. If the *backup* parameter is given (typically as `backup='.<some extension>'`), it specifies the extension for the backup file, and the backup file remains around; by default, the extension is `'.bak'` and it is deleted when the output file is closed. In-place filtering is disabled when standard input is read.

The two following opening hooks are provided by this module:

fileinput.hook_compressed(*filename*, *mode*, *, *encoding=None*, *errors=None*)

Transparently opens files compressed with gzip and bzip2 (recognized by the extensions `'.gz'` and `'.bz2'`) using the **`gzip`** and **`bz2`** modules. If the filename extension is not `'.gz'` or `'.bz2'`, the file is opened normally (ie, using **`open()`** without any decompression).

The *encoding* and *errors* values are passed to **`io.TextIOWrapper`** for compressed files and open for normal files.

Usage example: `fi = fileinput.FileInput(openhook=fileinput.hook_compressed, encoding="utf-8")`

Changed in version 3.10: The keyword-only parameter *encoding* and *errors* are added.

fileinput.hook_encoded(*encoding*, *errors=None*)

Returns a hook which opens each file with **`open()`**, using the given *encoding* and *errors* to read the file.

Usage example: `fi = fileinput.FileInput(openhook=fileinput.hook_encoded("utf-8", "surrogateescape"))`

Changed in version 3.6: Added the optional *errors* parameter.

Deprecated since version 3.10: This function is deprecated since **`fileinput.input()`** and **`FileInput`** now have *encoding* and *errors* parameters.

`curses` — Terminal handling for character-cell displays

Source code: Lib/curses
[https://github.com/python/cpython/tree/3.14/Lib/curses]

The **`curses`** module provides an interface to the curses library, the de-facto standard for portable advanced terminal handling.

While curses is most widely used in the Unix environment, versions are available for Windows, DOS, and possibly other systems as well. This extension module is designed to match the API of ncurses, an open-source curses library hosted on Linux and the BSD variants of Unix.

Availability: not Android, not iOS, not WASI.

This module is not supported on mobile platforms or WebAssembly platforms.

This is an optional module. If it is missing from your copy of CPython, look for documentation from your distributor (that is, whoever provided Python to you). If you are the distributor, see Requirements for optional modules.

Note

Whenever the documentation mentions a *character* it can be specified as an integer, a one-character Unicode string or a one-byte byte string.

Whenever the documentation mentions a *character string* it can be specified as a Unicode string or a byte string.

See also

Module [`curses.ascii`]
Utilities for working with ASCII characters, regardless of your locale settings.

Module `curses.panel`
A panel stack extension that adds depth to curses windows.

Module `curses.textpad`
Editable text widget for curses supporting **Emacs**-like bindings.

Curses Programming with Python
Tutorial material on using curses with Python, by Andrew Kuchling and Eric Raymond.

Functions

The module `curses` defines the following exception:

exception curses.error
Exception raised when a curses library function returns an error.

Note

Whenever *x* or *y* arguments to a function or a method are optional, they default to the current cursor location. Whenever *attr* is optional, it defaults to **`A_NORMAL`**.

The module **`curses`** defines the following functions:

curses.assume_default_colors(*fg*, *bg*, */*)

Allow use of default values for colors on terminals supporting this feature. Use this to support transparency in your application.

- Assign terminal default foreground/background colors to color number `-1`. So `init_pair(x, COLOR_RED, -1)` will initialize pair *x* as red on default background and `init_pair(x, -1, COLOR_BLUE)` will initialize pair *x* as default foreground on blue.
- Change the definition of the color-pair `0` to `(fg, bg)`.

Added in version 3.14.

curses.baudrate()

Return the output speed of the terminal in bits per second. On software terminal emulators it will have a fixed high value. Included for historical reasons; in former times, it was used to write output loops for time delays and occasionally to change interfaces depending on the line speed.

curses.beep()

Emit a short attention sound.

curses.can_change_color()

Return `True` or `False`, depending on whether the programmer can change the colors displayed by the terminal.

curses.cbreak()

Enter cbreak mode. In cbreak mode (sometimes called “rare” mode) normal tty line buffering is turned off and characters are available to be read one by one. However, unlike raw mode, special characters (interrupt, quit, suspend, and flow control) retain their effects on the tty driver and calling program. Calling first `raw()` then `cbreak()` leaves the terminal in cbreak mode.

curses.color_content(*color_number*)

Return the intensity of the red, green, and blue (RGB) components in the color *color_number*, which must be between `0` and `COLORS - 1`. Return a 3-tuple, containing the R,G,B values for the given color, which will be between `0` (no component) and `1000` (maximum amount of component).

curses.color_pair(*pair_number*)

Return the attribute value for displaying text in the specified color pair. Only the first 256 color pairs are supported. This attribute value can be combined with `A_STANDOUT`, `A_REVERSE`, and the other **`A_*`** attributes. `pair_number()` is the counterpart to this function.

curses.curs_set(*visibility*)

Set the cursor state. *visibility* can be set to `0`, `1`, or `2`, for invisible, normal, or very visible. If the terminal supports the visibility requested, return the previous cursor state; otherwise raise an exception. On many terminals, the “visible” mode is an underline cursor and the “very visible” mode is a block cursor.

curses.def_prog_mode()

Save the current terminal mode as the “program” mode, the mode when the running program is using curses. (Its counterpart is the “shell” mode, for when the program is not in curses.) Subsequent calls to `reset_prog_mode()` will restore this mode.

curses.def_shell_mode()

Save the current terminal mode as the “shell” mode, the mode when the running program is not using curses. (Its counterpart is the “program” mode, when the program is using curses capabilities.) Subsequent calls to `reset_shell_mode()` will restore this mode.

curses.delay_output(*ms*)

Insert an *ms* millisecond pause in output.

curses.doupdate()

Update the physical screen. The curses library keeps two data structures, one representing the current physical screen contents and a virtual screen representing the desired next state. The `doupdate()` ground updates the physical screen to match the virtual screen.

The virtual screen may be updated by a `noutrefresh()` call after write operations such as `addstr()` have been performed on a window. The normal `refresh()` call is simply `noutrefresh()` followed by `doupdate()`; if you have to update multiple windows, you can speed performance and perhaps reduce screen flicker by issuing `noutrefresh()` calls on all windows, followed by a single `doupdate()`.

curses.echo()

Enter echo mode. In echo mode, each character input is echoed to the screen as it is entered.

curses.endwin()

De-initialize the library, and return terminal to normal status.

curses.erasechar()

Return the user’s current erase character as a one-byte bytes object. Under Unix operating systems this is a property of the controlling tty of the curses program, and is not set by the curses library itself.

curses.filter()

The `filter()` routine, if used, must be called before `initscr()` is called. The effect is that, during those calls, **`LINES`** is set to `1`; the capabilities `clear`, `cup`, `cud`, `cud1`, `cuu1`, `cuu`, `vpa` are disabled; and the `home` string is set to the value of `cr`. The effect is that the cursor is confined to the current line, and so are screen updates. This may be used for enabling character-at-a-time line editing without touching the rest of the screen.

curses.flash()

Flash the screen. That is, change it to reverse-video and then change it back in a short interval. Some people prefer such as ‘visible bell’ to the audible attention signal produced by `beep()`.

curses.flushinp()

Flush all input buffers. This throws away any typeahead that has been typed by the user and has not yet been processed by the program.

curses.getmouse()

After `getch()` returns `KEY_MOUSE` to signal a mouse event, this method should be called to retrieve the queued mouse event, represented as a 5-tuple `(id, x, y, z, bstate)`. *id* is an ID value used to distinguish multiple devices, and *x*, *y*, *z* are the event's coordinates. (*z* is currently unused.) *bstate* is an integer value whose bits will be set to indicate the type of event, and will be the bitwise OR of one or more of the following constants, where *n* is the button number from 1 to 5: `BUTTONn_PRESSED`, `BUTTONn_RELEASED`, `BUTTONn_CLICKED`, `BUTTONn_DOUBLE_CLICKED`, `BUTTONn_TRIPLE_CLICKED`, `BUTTON_SHIFT`, `BUTTON_CTRL`, `BUTTON_ALT`.

Changed in version 3.10: The `BUTTON5_*` constants are now exposed if they are provided by the underlying curses library.

curses.getsyx()

Return the current coordinates of the virtual screen cursor as a tuple `(y, x)`. If `leaveok` is currently `True`, then return `(-1, -1)`.

curses.getwin(*file*)

Read window related data stored in the file by an earlier `window.putwin()` call. The routine then creates and initializes a new window using that data, returning the new window object.

curses.has_colors()

Return `True` if the terminal can display colors; otherwise, return `False`.

curses.has_extended_color_support()

Return `True` if the module supports extended colors; otherwise, return `False`. Extended color support allows more than 256 color pairs for terminals that support more than 16 colors (e.g. xterm-256color).

Extended color support requires ncurses version 6.1 or later.

Added in version 3.10.

curses.has_ic()

Return `True` if the terminal has insert- and delete-character capabilities. This function is included for historical reasons only, as all modern software terminal emulators have such capabilities.

curses.has_il()

Return `True` if the terminal has insert- and delete-line capabilities, or can simulate them using scrolling regions. This function is included for historical reasons only, as all modern software terminal emulators have such capabilities.

curses.has_key(*ch*)

Take a key value *ch*, and return `True` if the current terminal type recognizes a key with that value.

curses.halfdelay(*tenths*)

Used for half-delay mode, which is similar to cbreak mode in that characters typed by the user are immediately available to the program. However, after blocking for *tenths* tenths of seconds, raise an exception if nothing has been typed. The value of *tenths* must be a number between `1` and `255`. Use **`nocbreak()`** to leave half-delay mode.

curses.init_color(*color_number*, *r*, *g*, *b*)

Change the definition of a color, taking the number of the color to be changed followed by three RGB values (for the amounts of red, green, and blue components). The value of *color_number* must be between `0` and `COLORS - 1`. Each of *r*, *g*, *b*, must be a value between `0` and `1000`. When **`init_color()`** is used, all occurrences of that color on the screen immediately change to the new definition. This function is a no-op on most terminals; it is active only if **`can_change_color()`** returns `True`.

curses.init_pair(*pair_number*, *fg*, *bg*)

Change the definition of a color-pair. It takes three arguments: the number of the color-pair to be changed, the foreground color number, and the background color number. The value of *pair_number* must be between `1` and `COLOR_PAIRS - 1` (the `0` color pair can only be changed by **`use_default_colors()`** and **`assume_default_colors()`**). The value of *fg* and *bg* arguments must be between `0` and `COLORS - 1`, or, after calling **`use_default_colors()`** or **`assume_default_colors()`**, `-1`. If the color-pair was previously initialized, the screen is refreshed and all occurrences of that color-pair are changed to the new definition.

curses.initscr()

Initialize the library. Return a window object which represents the whole screen.

Note

If there is an error opening the terminal, the underlying curses library may cause the interpreter to exit.

curses.is_term_resized(*nlines*, *ncols*)

Return `True` if **`resize_term()`** would modify the window structure, `False` otherwise.

curses.isendwin()

Return `True` if **`endwin()`** has been called (that is, the curses library has been deinitialized).

curses.keyname(*k*)

Return the name of the key numbered *k* as a bytes object. The name of a key generating printable ASCII character is the key’s character. The name of a control-key combination is a two-byte bytes object consisting of a caret (`b'^'`) followed by the corresponding printable ASCII character. The name of an alt-key combination (128–255) is a bytes object consisting of the prefix `b'M-'` followed by the name of the corresponding ASCII character.

curses.killchar()

Return the user’s current line kill character as a one-byte bytes object. Under Unix operating systems this is a property of the controlling tty of the curses program, and is not set by the curses library itself.

curses.longname()

Return a bytes object containing the terminfo long name field describing the current terminal. The maximum length of a verbose description is 128 characters. It is defined only after the call to `initscr()`.

curses.meta(*flag*)

If *flag* is `True`, allow 8-bit characters to be input. If *flag* is `False`, allow only 7-bit chars.

curses.mouseinterval(*interval*)

Set the maximum time in milliseconds that can elapse between press and release events in order for them to be recognized as a click, and return the previous interval value. The default value is 200 milliseconds, or one fifth of a second.

curses.mousemask(*mousemask*)

Set the mouse events to be reported, and return a tuple `(availmask, oldmask)`. *availmask* indicates which of the specified mouse events can be reported; on complete failure it returns `0`. *oldmask* is the previous value of the given window’s mouse event mask. If this function is never called, no mouse events are ever reported.

curses.napms(*ms*)

Sleep for *ms* milliseconds.

curses.newpad(*nlines*, *ncols*)

Create and return a pointer to a new pad data structure with the given number of lines and columns. Return a pad as a window object.

A pad is like a window, except that it is not restricted by the screen size, and is not necessarily associated with a particular part of the screen. Pads can be used when a large window is needed, and only a part of the window will be on the screen at one time. Automatic refreshes of pads (such as from scrolling or echoing of input) do not occur. The **`refresh()`** and **`noutrefresh()`** methods of a pad require 6 arguments to specify the part of the pad to be displayed and the location on the screen to be used for the display. The arguments are *pminrow*, *pmincol*, *sminrow*, *smincol*, *smaxrow*, *smaxcol*; the *p* arguments refer to the upper left corner of the pad region to be displayed and the *s* arguments define a clipping box on the screen within which the pad region is to be displayed.

curses.newwin(*nlines*, *ncols*)
curses.newwin(*nlines*, *ncols*, *begin_y*, *begin_x*)

Return a new window, whose left-upper corner is at `(begin_y, begin_x)`, and whose height/width is *nlines*/*ncols*.

By default, the window will extend from the specified position to the lower right corner of the screen.

curses.nl()

Enter newline mode. This mode translates the return key into newline on input, and translates newline into return and line-feed on output. Newline mode is initially on.

curses.nocbreak()

Leave cbreak mode. Return to normal “cooked” mode with line buffering.

curses.noecho()

Leave echo mode. Echoing of input characters is turned off.

curses.nonl()

Leave newline mode. Disable translation of return into newline on input, and disable low-level translation of newline into newline/return on output (but this does not change the behavior of `addch('\n')`, which always does the equivalent of return and line feed on the virtual screen). With translation off, curses can sometimes speed up vertical motion a little; also, it will be able to detect the return key on input.

curses.noqiflush()

When the **`noqiflush()`** routine is used, normal flush of input and output queues associated with the `INTR`, `QUIT` and `SUSP` characters will not be done. You may want to call **`noqiflush()`** in a signal handler if you want output to continue as though the interrupt had not occurred, after the handler exits.

curses.noraw()

Leave raw mode. Return to normal “cooked” mode with line buffering.

curses.pair_content(*pair_number*)

Return a tuple `(fg, bg)` containing the colors for the requested color pair. The value of *pair_number* must be between `0` and `COLOR_PAIRS - 1`.

curses.pair_number(*attr*)

Return the number of the color-pair set by the attribute value *attr*. **`color_pair()`** is the counterpart to this function.

curses.putp(*str*)

Equivalent to `tputs(str, 1, putchar)`; emit the value of a specified terminfo capability for the current terminal. Note that the output of **`putp()`** always goes to standard output.

curses.qiflush([*flag*])

If *flag* is `False`, the effect is the same as calling **`noqiflush()`**. If *flag* is `True`, or no argument is provided, the queues will be flushed when these control characters are read.

curses.raw()

Enter raw mode. In raw mode, normal line buffering and processing of interrupt, quit, suspend, and flow control keys are turned off; characters are presented to curses input functions one by one.

curses.reset_prog_mode()

Restore the terminal to “program” mode, as previously saved by **`def_prog_mode()`**.

curses.reset_shell_mode()

Restore the terminal to “shell” mode, as previously saved by `def_shell_mode()`.

curses.resetty()

Restore the state of the terminal modes to what it was at the last call to `savetty()`.

curses.resize_term(*nlines*, *ncols*)

Backend function used by `resizeterm()`, performing most of the work; when resizing the windows, `resize_term()` blank-fills the areas that are extended. The calling application should fill in these areas with appropriate data. The **`resize_term()`** function attempts to resize all windows. However, due to the calling convention of pads, it is not possible to resize these without additional interaction with the application.

curses.resizeterm(*nlines*, *ncols*)

Resize the standard and current windows to the specified dimensions, and adjusts other bookkeeping data used by the curses library that record the window dimensions (in particular the SIGWINCH handler).

curses.savetty()

Save the current state of the terminal modes in a buffer, usable by `resetty()`.

curses.get_escdelay()

Retrieves the value set by `set_escdelay()`.

Added in version 3.9.

curses.set_escdelay(*ms*)

Sets the number of milliseconds to wait after reading an escape character, to distinguish between an individual escape character entered on the keyboard from escape sequences sent by cursor and function keys.

Added in version 3.9.

curses.get_tabsize()

Retrieves the value set by `set_tabsize()`.

Added in version 3.9.

curses.set_tabsize(*size*)

Sets the number of columns used by the curses library when converting a tab character to spaces as it adds the tab to a window.

Added in version 3.9.

curses.setsyx(*y*, *x*)

Set the virtual screen cursor to *y*, *x*. If *y* and *x* are both `-1`, then `leaveok` is set `True`.

curses.setupterm(*term=None*, *fd=-1*)

Initialize the terminal. *term* is a string giving the terminal name, or `None`; if omitted or `None`, the value of the **`TERM`** environment variable will be used. *fd* is the file descriptor to which any initialization sequences will be sent; if not supplied or `-1`, the file descriptor for `sys.stdout` will be used.

curses.start_color()

Must be called if the programmer wants to use colors, and before any other color manipulation routine is called. It is good practice to call this routine right after `initscr()`.

`start_color()` initializes eight basic colors (black, red, green, yellow, blue, magenta, cyan, and white), and two global variables in the `curses` module, `COLORS` and `COLOR_PAIRS`, containing the maximum number of colors and color-pairs the terminal can support. It also restores the colors on the terminal to the values they had when the terminal was just turned on.

curses.termattrs()

Return a logical OR of all video attributes supported by the terminal. This information is useful when a curses program needs complete control over the appearance of the screen.

curses.termname()

Return the value of the environment variable **`TERM`**, as a bytes object, truncated to 14 characters.

curses.tigetflag(*capname*)

Return the value of the Boolean capability corresponding to the terminfo capability name *capname* as an integer. Return the value `-1` if *capname* is not a Boolean capability, or `0` if it is canceled or absent from the terminal description.

curses.tigetnum(*capname*)

Return the value of the numeric capability corresponding to the terminfo capability name *capname* as an integer. Return the value `-2` if *capname* is not a numeric capability, or `-1` if it is canceled or absent from the terminal description.

curses.tigetstr(*capname*)

Return the value of the string capability corresponding to the terminfo capability name *capname* as a bytes object. Return `None` if *capname* is not a terminfo “string capability”, or is canceled or absent from the terminal description.

curses.tparm(*str*[, *...*])

Instantiate the bytes object *str* with the supplied parameters, where *str* should be a parameterized string obtained from the terminfo database. E.g. `tparm(tigetstr("cup"), 5, 3)` could result in `b'\033[6;4H'`, the exact result depending on terminal type.

curses.typeahead(*fd*)

Specify that the file descriptor *fd* be used for typeahead checking. If *fd* is `-1`, then no typeahead checking is done.

The curses library does “line-breakout optimization” by looking for typeahead periodically while updating the screen. If input is found, and it is coming from a tty, the current update is postponed until refresh or doupdate is called again, allowing faster response to commands typed in advance. This function allows specifying a different file descriptor for typeahead checking.

curses.unctrl(*ch*)

Return a bytes object which is a printable representation of the character *ch*. Control characters are represented as a caret followed by the character, for example as `b'^C'`. Printing characters are left as they are.

curses.ungetch(*ch*)

Push *ch* so the next `getch()` will return it.

Note

Only one *ch* can be pushed before **`getch()`** is called.

curses.update_lines_cols()

Update the **`LINES`** and **`COLS`** module variables. Useful for detecting manual screen resize.

Added in version 3.5.

curses.unget_wch(*ch*)

Push *ch* so the next **`get_wch()`** will return it.

Note

Only one *ch* can be pushed before **`get_wch()`** is called.

Added in version 3.3.

curses.ungetmouse(*id*, *x*, *y*, *z*, *bstate*)

Push a **`KEY_MOUSE`** event onto the input queue, associating the given state data with it.

curses.use_env(*flag*)

If used, this function should be called before **`initscr()`** or newterm are called. When *flag* is `False`, the values of lines and columns specified in the terminfo database will be used, even if environment variables **`LINES`** and **`COLUMNS`** (used by default) are set, or if curses is running in a window (in which case default behavior would be to use the window size if **`LINES`** and **`COLUMNS`** are not set).

curses.use_default_colors()

Equivalent to `assume_default_colors(-1, -1)`.

curses.wrapper(*func*, /, **args*, ***kwargs*)

Initialize curses and call another callable object, *func*, which should be the rest of your curses-using application. If the application raises an exception, this function will restore the terminal to a sane state before re-raising the exception and generating a traceback. The callable object *func* is then passed the main window ‘stdscr’ as its first argument, followed by any other arguments passed to **`wrapper()`**. Before calling *func*, **`wrapper()`** turns on cbreak mode, turns off echo, enables the terminal keypad, and initializes colors if the terminal has color support. On exit (whether normally or by exception) it restores cooked mode, turns on echo, and disables the terminal keypad.

Window Objects

class curses.window

Window objects, as returned by `initscr()` and `newwin()` above, have the following methods and attributes:

window.addch(*ch*[, *attr*])

window.addch(*y*, *x*, *ch*[, *attr*])

Paint character *ch* at `(y, x)` with attributes *attr*, overwriting any character previously painted at that location. By default, the character position and attributes are the current settings for the window object.

Note

Writing outside the window, subwindow, or pad raises a `curses.error`. Attempting to write to the lower right corner of a window, subwindow, or pad will cause an exception to be raised after the character is printed.

window.addnstr(*str*, *n*[, *attr*])

window.addnstr(*y*, *x*, *str*, *n*[, *attr*])

Paint at most *n* characters of the character string *str* at `(y, x)` with attributes *attr*, overwriting anything previously on the display.

window.addstr(*str*[, *attr*])

window.addstr(*y*, *x*, *str*[, *attr*])

Paint the character string *str* at `(y, x)` with attributes *attr*, overwriting anything previously on the display.

Note

- Writing outside the window, subwindow, or pad raises **`curses.error`**. Attempting to write to the lower right corner of a window, subwindow, or pad will cause an exception to be raised after the string is printed.
- A bug in ncurses [https://bugs.python.org/issue35924], the backend for this Python module, can cause SegFaults when resizing windows. This is fixed in ncurses-6.1-20190511. If you are stuck with an earlier ncurses, you can avoid triggering this if you do not call **`addstr()`** with a *str* that has embedded newlines. Instead, call **`addstr()`** separately for each line.

window.attroff(*attr*)

Remove attribute *attr* from the “background” set applied to all writes to the current window.

window.attron(*attr*)

Add attribute *attr* to the “background” set applied to all writes to the current window.

window.attrset(*attr*)

Set the “background” set of attributes to *attr*. This set is initially `0` (no attributes).

window.bkgd(*ch*[, *attr*])

Set the background property of the window to the character *ch*, with attributes *attr*. The change is then applied to every character position in that window:

- The attribute of every character in the window is changed to the new background attribute.
- Wherever the former background character appears, it is changed to the new background character.

window.bkgdset(*ch*[, *attr*])

Set the window’s background. A window’s background consists of a character and any combination of attributes. The attribute part of the background is combined (OR’ed) with all non-blank characters that are written into the window. Both the character and attribute parts of the background are combined with the blank characters. The background becomes a property of the character and moves with the character through any scrolling and insert/delete line/character operations.

window.border([*ls*[, *rs*[, *ts*[, *bs*[, *tl*[, *tr*[, *bl*[, *br*]]]]]]]])

Draw a border around the edges of the window. Each parameter specifies the character to use for a specific part of the border; see the table below for more details.

Note

A `0` value for any parameter will cause the default character to be used for that parameter. Keyword parameters can *not* be used. The defaults are listed in this table:

Parameter	**Description**	**Default value**
ls	Left side	`ACS_VLINE`
rs	Right side	`ACS_VLINE`
ts	Top	`ACS_HLINE`
bs	Bottom	`ACS_HLINE`
tl	Upper-left corner	`ACS_ULCORNER`
tr	Upper-right corner	`ACS_URCORNER`
bl	Bottom-left corner	`ACS_LLCORNER`
br	Bottom-right corner	`ACS_LRCORNER`

window.box([*vertch*, *horch*])

Similar to `border()`, but both *ls* and *rs* are *vertch* and both *ts* and *bs* are *horch*. The default corner characters are always used by this function.

window.chgat(*attr*)
window.chgat(*num*, *attr*)
window.chgat(*y*, *x*, *attr*)
window.chgat(*y*, *x*, *num*, *attr*)

Set the attributes of *num* characters at the current cursor position, or at position `(y, x)` if supplied. If *num* is not given or is `-1`, the attribute will be set on all the characters to the end of the line. This function moves cursor to position `(y, x)` if supplied. The changed line will be touched using the **`touchline()`** method so that the contents will be redisplayed by the next window refresh.

window.clear()

Like **`erase()`**, but also cause the whole window to be repainted upon next call to **`refresh()`**.

window.clearok(*flag*)

If *flag* is `True`, the next call to **`refresh()`** will clear the window completely.

window.clrtobot()

Erase from cursor to the end of the window: all lines below the cursor are deleted, and then the equivalent of **`clrtoeol()`** is performed.

window.clrtoeol()

Erase from cursor to the end of the line.

window.cursyncup()

Update the current cursor position of all the ancestors of the window to reflect the current cursor position of the window.

window.delch([*y*, *x*])

Delete any character at `(y, x)`.

window.deleteln()

Delete the line under the cursor. All following lines are moved up by one line.

window.derwin(*begin_y*, *begin_x*)
window.derwin(*nlines*, *ncols*, *begin_y*, *begin_x*)

An abbreviation for “derive window”, `derwin()` is the same as calling `subwin()`, except that *begin_y* and *begin_x* are relative to the origin of the window, rather than relative to the entire screen. Return a window object for the derived window.

window.echochar(*ch*[, *attr*])

Add character *ch* with attribute *attr*, and immediately call `refresh()` on the window.

window.enclose(*y*, *x*)

Test whether the given pair of screen-relative character-cell coordinates are enclosed by the given window, returning `True` or `False`. It is useful for determining what subset of the screen windows enclose the location of a mouse event.

Changed in version 3.10: Previously it returned `1` or `0` instead of `True` or `False`.

window.encoding

Encoding used to encode method arguments (Unicode strings and characters). The encoding attribute is inherited from the parent window when a subwindow is created, for example with `window.subwin()`. By default, current locale encoding is used (see `locale.getencoding()`).

Added in version 3.3.

window.erase()

Clear the window.

window.getbegyx()

Return a tuple `(y, x)` of coordinates of upper-left corner.

window.getbkgd()

Return the given window’s current background character/attribute pair.

window.getch([*y*, *x*])

Get a character. Note that the integer returned does *not* have to be in ASCII range: function keys, keypad keys and so on are represented by numbers higher than 255. In no-delay mode, return `-1` if there is no input, otherwise wait until a key is pressed.

window.get_wch([*y*, *x*])

Get a wide character. Return a character for most keys, or an integer for function keys, keypad keys, and other special keys. In no-delay mode, raise an exception if there is no input.

Added in version 3.3.

window.getkey([*y*, *x*])

Get a character, returning a string instead of an integer, as `getch()` does. Function keys, keypad keys and other special keys return a multibyte string containing the key name. In no-delay mode, raise an exception if there is no input.

window.getmaxyx()

Return a tuple `(y, x)` of the height and width of the window.

window.getparyx()

Return the beginning coordinates of this window relative to its parent window as a tuple `(y, x)`. Return `(-1, -1)` if this window has no parent.

window.getstr()
window.getstr(*n*)
window.getstr(*y*, *x*)
window.getstr(*y*, *x*, *n*)

Read a bytes object from the user, with primitive line editing capacity. The maximum value for *n* is 2047.

Changed in version 3.14: The maximum value for *n* was increased from 1023 to 2047.

window.getyx()

Return a tuple `(y, x)` of current cursor position relative to the window's upper-left corner.

window.hline(*ch*, *n*)
window.hline(*y*, *x*, *ch*, *n*)

Display a horizontal line starting at `(y, x)` with length *n* consisting of the character *ch*.

window.idcok(*flag*)

If *flag* is `False`, curses no longer considers using the hardware insert/delete character feature of the terminal; if *flag* is `True`, use of character insertion and deletion is enabled. When curses is first initialized, use of character insert/delete is enabled by default.

window.idlok(*flag*)

If *flag* is `True`, **`curses`** will try and use hardware line editing facilities. Otherwise, line insertion/deletion are disabled.

window.immedok(*flag*)

If *flag* is `True`, any change in the window image automatically causes the window to be refreshed; you no longer have to call **`refresh()`** yourself. However, it may degrade performance considerably, due to repeated calls to wrefresh. This option is disabled by default.

window.inch([*y*, *x*])

Return the character at the given position in the window. The bottom 8 bits are the character proper, and upper bits are the attributes.

window.insch(*ch*[, *attr*])

window.insch(*y*, *x*, *ch*[, *attr*])

Paint character *ch* at `(y, x)` with attributes *attr*, moving the line from position *x* right by one character.

window.insdelln(*nlines*)

Insert *nlines* lines into the specified window above the current line. The *nlines* bottom lines are lost. For negative *nlines*, delete *nlines* lines starting with the one under the cursor, and move the remaining lines up. The bottom *nlines* lines are cleared. The current cursor position remains the same.

window.insertln()

Insert a blank line under the cursor. All following lines are moved down by one line.

window.insnstr(*str*, *n*[, *attr*])

window.insnstr(*y*, *x*, *str*, *n*[, *attr*])

Insert a character string (as many characters as will fit on the line) before the character under the cursor, up to *n* characters. If *n* is zero or negative, the entire string is inserted. All characters to the right of the cursor are shifted right, with the rightmost characters on the line being lost. The cursor position does not change (after moving to *y*, *x*, if specified).

window.insstr(*str*[, *attr*])

window.insstr(*y*, *x*, *str*[, *attr*])

Insert a character string (as many characters as will fit on the line) before the character under the cursor. All characters to the right of the cursor are shifted right, with the rightmost characters on the line being lost. The cursor position does not change (after moving to *y*, *x*, if specified).

window.instr([*n*])

window.instr(*y*, *x*[, *n*])

Return a bytes object of characters, extracted from the window starting at the current cursor position, or at *y*, *x* if specified. Attributes are stripped from the characters. If *n* is specified, `instr()` returns a string at most *n* characters long (exclusive of the trailing NUL). The maximum value for *n* is 2047.

Changed in version 3.14: The maximum value for *n* was increased from 1023 to 2047.

window.is_linetouched(*line*)

Return `True` if the specified line was modified since the last call to `refresh()`; otherwise return `False`. Raise a `curses.error` exception if *line* is not valid for the given window.

window.is_wintouched()

Return `True` if the specified window was modified since the last call to `refresh()`; otherwise return `False`.

window.keypad(*flag*)

If *flag* is `True`, escape sequences generated by some keys (keypad, function keys) will be interpreted by `curses`. If *flag* is `False`, escape sequences will be left as is in the input stream.

window.leaveok(*flag*)

If *flag* is `True`, cursor is left where it is on update, instead of being at “cursor position.” This reduces cursor movement where possible. If possible the cursor will be made invisible.

If *flag* is `False`, cursor will always be at “cursor position” after an update.

window.move(*new_y*, *new_x*)

Move cursor to `(new_y, new_x)`.

window.mvderwin(*y*, *x*)

Move the window inside its parent window. The screen-relative parameters of the window are not changed. This routine is used to display different parts of the parent window at the same physical position on the screen.

window.mvwin(*new_y*, *new_x*)

Move the window so its upper-left corner is at `(new_y, new_x)`.

window.nodelay(*flag*)

If *flag* is `True`, `getch()` will be non-blocking.

window.notimeout(*flag*)

If *flag* is `True`, escape sequences will not be timed out.

If *flag* is `False`, after a few milliseconds, an escape sequence will not be interpreted, and will be left in the input stream as is.

window.noutrefresh()

Mark for refresh but wait. This function updates the data structure representing the desired state of the window, but does not force an update of the physical screen. To accomplish that, call **`doupdate()`**.

window.overlay(*destwin*[, *sminrow*, *smincol*, *dminrow*, *dmincol*, *dmaxrow*, *dmaxcol*])

Overlay the window on top of *destwin*. The windows need not be the same size, only the overlapping region is copied. This copy is non-destructive, which means that the current background character does not overwrite the old contents of *destwin*.

To get fine-grained control over the copied region, the second form of **`overlay()`** can be used. *sminrow* and *smincol* are the upper-left coordinates of the source window, and the other variables mark a rectangle in the destination window.

window.overwrite(*destwin*[, *sminrow*, *smincol*, *dminrow*, *dmincol*, *dmaxrow*, *dmaxcol*])

Overwrite the window on top of *destwin*. The windows need not be the same size, in which case only the overlapping region is copied. This copy is destructive, which means that the current background character overwrites the old contents of *destwin*.

To get fine-grained control over the copied region, the second form of **`overwrite()`** can be used. *sminrow* and *smincol* are the upper-left coordinates of the source window, the other variables mark a rectangle in the destination window.

window.putwin(*file*)

Write all data associated with the window into the provided file object. This information can be later retrieved using the `getwin()` function.

window.redrawln(*beg*, *num*)

Indicate that the *num* screen lines, starting at line *beg*, are corrupted and should be completely redrawn on the next `refresh()` call.

window.redrawwin()

Touch the entire window, causing it to be completely redrawn on the next `refresh()` call.

window.refresh([*pminrow*, *pmincol*, *sminrow*, *smincol*, *smaxrow*, *smaxcol*])

Update the display immediately (sync actual screen with previous drawing/deleting methods).

The 6 optional arguments can only be specified when the window is a pad created with `newpad()`. The additional parameters are needed to indicate what part of the pad and screen are involved. *pminrow* and *pmincol* specify the upper left-hand corner of the rectangle to be displayed in the pad. *sminrow*, *smincol*, *smaxrow*, and *smaxcol* specify the edges of the rectangle to be displayed on the screen. The lower right-hand corner of the rectangle to be displayed in the pad is calculated from the screen coordinates, since the rectangles must be the same size. Both rectangles must be entirely contained within their respective structures. Negative values of *pminrow*, *pmincol*, *sminrow*, or *smincol* are treated as if they were zero.

window.resize(*nlines*, *ncols*)

Reallocate storage for a curses window to adjust its dimensions to the specified values. If either dimension is larger than the current values, the window’s data is filled with blanks that have the current background rendition (as set by **`bkgdset()`**) merged into them.

window.scroll([*lines=1*])

Scroll the screen or scrolling region upward by *lines* lines.

window.scrollok(*flag*)

Control what happens when the cursor of a window is moved off the edge of the window or scrolling region, either as a result of a newline action on the bottom line, or typing the last character of the last line. If *flag* is `False`, the cursor is left on the bottom line. If *flag* is `True`, the window is scrolled up one line. Note that in order to get the physical scrolling effect on the terminal, it is also necessary to call **`idlok()`**.

window.setscrreg(*top*, *bottom*)

Set the scrolling region from line *top* to line *bottom*. All scrolling actions will take place in this region.

window.standend()

Turn off the standout attribute. On some terminals this has the side effect of turning off all attributes.

window.standout()

Turn on attribute *A_STANDOUT*.

window.subpad(*begin_y*, *begin_x*)

window.subpad(*nlines*, *ncols*, *begin_y*, *begin_x*)

Return a sub-window, whose upper-left corner is at `(begin_y, begin_x)`, and whose width/height is *ncols*/*nlines*.

window.subwin(*begin_y*, *begin_x*)
window.subwin(*nlines*, *ncols*, *begin_y*, *begin_x*)

Return a sub-window, whose upper-left corner is at `(begin_y, begin_x)`, and whose width/height is *ncols*/*nlines*.

By default, the sub-window will extend from the specified position to the lower right corner of the window.

window.syncdown()

Touch each location in the window that has been touched in any of its ancestor windows. This routine is called by `refresh()`, so it should almost never be necessary to call it manually.

window.syncok(*flag*)

If *flag* is `True`, then `syncup()` is called automatically whenever there is a change in the window.

window.syncup()

Touch all locations in ancestors of the window that have been changed in the window.

window.timeout(*delay*)

Set blocking or non-blocking read behavior for the window. If *delay* is negative, blocking read is used (which will wait indefinitely for input). If *delay* is zero, then non-blocking read is used, and `getch()` will return `-1` if no input is waiting. If *delay* is positive, then `getch()` will block for *delay* milliseconds, and return `-1` if there is still no input at the end of that time.

window.touchline(*start*, *count*[, *changed*])

Pretend *count* lines have been changed, starting with line *start*. If *changed* is supplied, it specifies whether the affected lines are marked as having been changed (*changed*=`True`) or unchanged (*changed*=`False`).

window.touchwin()

Pretend the whole window has been changed, for purposes of drawing optimizations.

window.untouchwin()

Mark all lines in the window as unchanged since the last call to `refresh()`.

window.vline(*ch*, *n*[, *attr*])

window.vline(*y*, *x*, *ch*, *n*[, *attr*])

Display a vertical line starting at `(y, x)` with length *n* consisting of the character *ch* with attributes *attr*.

Constants

The `curses` module defines the following data members:

curses.ERR

Some curses routines that return an integer, such as `getch()`, return `ERR` upon failure.

curses.OK

Some curses routines that return an integer, such as `napms()`, return `OK` upon success.

curses.version

A bytes object representing the current version of the module.

curses.ncurses_version

A named tuple containing the three components of the ncurses library version: *major*, *minor*, and *patch*. All values are integers. The components can also be accessed by name, so `curses.ncurses_version[0]` is equivalent to `curses.ncurses_version.major` and so on.

Availability: if the ncurses library is used.

Added in version 3.8.

curses.COLORS

The maximum number of colors the terminal can support. It is defined only after the call to **`start_color()`**.

curses.COLOR_PAIRS

The maximum number of color pairs the terminal can support. It is defined only after the call to **`start_color()`**.

curses.COLS

The width of the screen, i.e., the number of columns. It is defined only after the call to **`initscr()`**. Updated by **`update_lines_cols()`**, **`resizeterm()`** and **`resize_term()`**.

curses.LINES

The height of the screen, i.e., the number of lines. It is defined only after the call to **`initscr()`**. Updated by **`update_lines_cols()`**, **`resizeterm()`** and **`resize_term()`**.

Some constants are available to specify character cell attributes. The exact constants available are system dependent.

Attribute	Meaning
curses.A_ALTCHARSET	Alternate character set mode
curses.A_BLINK	Blink mode
curses.A_BOLD	Bold mode
curses.A_DIM	Dim mode
curses.A_INVIS	Invisible or blank mode
curses.A_ITALIC	Italic mode
curses.A_NORMAL	Normal attribute
curses.A_PROTECT	Protected mode
curses.A_REVERSE	Reverse background and foreground colors
curses.A_STANDOUT	Standout mode
curses.A_UNDERLINE	Underline mode

Attribute	**Meaning**
curses.A_HORIZONTAL	Horizontal highlight
curses.A_LEFT	Left highlight
curses.A_LOW	Low highlight
curses.A_RIGHT	Right highlight
curses.A_TOP	Top highlight
curses.A_VERTICAL	Vertical highlight

Added in version 3.7: `A_ITALIC` was added.

Several constants are available to extract corresponding attributes returned by some methods.

Bit-mask	**Meaning**
curses.A_ATTRIBUTES	Bit-mask to extract attributes
curses.A_CHARTEXT	Bit-mask to extract a character

Bit-mask	Meaning
curses.A_COLOR	Bit-mask to extract color-pair field information

Keys are referred to by integer constants with names starting with KEY_. The exact keycaps available are system dependent.

Key constant	Key
curses.KEY_MIN	Minimum key value
curses.KEY_BREAK	Break key (unreliable)
curses.KEY_DOWN	Down-arrow
curses.KEY_UP	Up-arrow
curses.KEY_LEFT	Left-arrow
curses.KEY_RIGHT	Right-arrow
curses.KEY_HOME	Home key (upward+left arrow)
curses.KEY_BACKSPACE	Backspace (unreliable)

Key constant	**Key**
curses.KEY_F0	Function keys. Up to 64 function keys are supported.
curses.KEY_Fn	Value of function key *n*
curses.KEY_DL	Delete line
curses.KEY_IL	Insert line
curses.KEY_DC	Delete character
curses.KEY_IC	Insert char or enter insert mode
curses.KEY_EIC	Exit insert char mode
curses.KEY_CLEAR	Clear screen
curses.KEY_EOS	Clear to end of screen
curses.KEY_EOL	Clear to end of line
curses.KEY_SF	Scroll 1 line forward

Key constant	**Key**
curses.KEY_SR	Scroll 1 line backward (reverse)
curses.KEY_NPAGE	Next page
curses.KEY_PPAGE	Previous page
curses.KEY_STAB	Set tab
curses.KEY_CTAB	Clear tab
curses.KEY_CATAB	Clear all tabs
curses.KEY_ENTER	Enter or send (unreliable)
curses.KEY_SRESET	Soft (partial) reset (unreliable)
curses.KEY_RESET	Reset or hard reset (unreliable)
curses.KEY_PRINT	Print
curses.KEY_LL	Home down or bottom (lower left)
curses.KEY_A1	Upper left of keypad

Key constant	Key
curses.KEY_A3	Upper right of keypad
curses.KEY_B2	Center of keypad
curses.KEY_C1	Lower left of keypad
curses.KEY_C3	Lower right of keypad
curses.KEY_BTAB	Back tab
curses.KEY_BEG	Beg (beginning)
curses.KEY_CANCEL	Cancel
curses.KEY_CLOSE	Close
curses.KEY_COMMAND	Cmd (command)
curses.KEY_COPY	Copy
curses.KEY_CREATE	Create
curses.KEY_END	End

Key constant	Key
curses.KEY_EXIT	Exit
curses.KEY_FIND	Find
curses.KEY_HELP	Help
curses.KEY_MARK	Mark
curses.KEY_MESSAGE	Message
curses.KEY_MOVE	Move
curses.KEY_NEXT	Next
curses.KEY_OPEN	Open
curses.KEY_OPTIONS	Options
curses.KEY_PREVIOUS	Prev (previous)
curses.KEY_REDO	Redo
curses.KEY_REFERENCE	Ref (reference)

Key constant	Key
curses.KEY_REFRESH	Refresh
curses.KEY_REPLACE	Replace
curses.KEY_RESTART	Restart
curses.KEY_RESUME	Resume
curses.KEY_SAVE	Save
curses.KEY_SBEG	Shifted Beg (beginning)
curses.KEY_SCANCEL	Shifted Cancel
curses.KEY_SCOMMAND	Shifted Command
curses.KEY_SCOPY	Shifted Copy
curses.KEY_SCREATE	Shifted Create
curses.KEY_SDC	Shifted Delete char

Key constant	Key
curses.KEY_SDL	Shifted Delete line
curses.KEY_SELECT	Select
curses.KEY_SEND	Shifted End
curses.KEY_SEOL	Shifted Clear line
curses.KEY_SEXIT	Shifted Exit
curses.KEY_SFIND	Shifted Find
curses.KEY_SHELP	Shifted Help
curses.KEY_SHOME	Shifted Home
curses.KEY_SIC	Shifted Input
curses.KEY_SLEFT	Shifted Left arrow
curses.KEY_SMESSAGE	Shifted Message
curses.KEY_SMOVE	Shifted Move

Key constant	Key
curses.KEY_SNEXT	Shifted Next
curses.KEY_SOPTIONS	Shifted Options
curses.KEY_SPREVIOUS	Shifted Prev
curses.KEY_SPRINT	Shifted Print
curses.KEY_SREDO	Shifted Redo
curses.KEY_SREPLACE	Shifted Replace
curses.KEY_SRIGHT	Shifted Right arrow
curses.KEY_SRSUME	Shifted Resume
curses.KEY_SSAVE	Shifted Save
curses.KEY_SSUSPEND	Shifted Suspend
curses.KEY_SUNDO	Shifted Undo
curses.KEY_SUSPEND	Suspend

Key constant	Key
curses.KEY_UNDO	Undo
curses.KEY_MOUSE	Mouse event has occurred
curses.KEY_RESIZE	Terminal resize event
curses.KEY_MAX	Maximum key value

On VT100s and their software emulations, such as X terminal emulators, there are normally at least four function keys (**`KEY_F1`**, **`KEY_F2`**, **`KEY_F3`**, **`KEY_F4`**) available, and the arrow keys mapped to **`KEY_UP`**, **`KEY_DOWN`**, **`KEY_LEFT`** and **`KEY_RIGHT`** in the obvious way. If your machine has a PC keyboard, it is safe to expect arrow keys and twelve function keys (older PC keyboards may have only ten function keys); also, the following keypad mappings are standard:

Keycap	Constant
`Insert`	KEY_IC
`Delete`	KEY_DC
`Home`	KEY_HOME

Keycap	Constant
`End`	KEY_END
`Page Up`	KEY_PPAGE
`Page Down`	KEY_NPAGE

The following table lists characters from the alternate character set. These are inherited from the VT100 terminal, and will generally be available on software emulations such as X terminals. When there is no graphic available, curses falls back on a crude printable ASCII approximation.

Note

These are available only after `initscr()` has been called.

ACS code	Meaning
curses.ACS_BBSS	alternate name for upper right corner
curses.ACS_BLOCK	solid square block
curses.ACS_BOARD	board of squares

ACS code	Meaning
curses.ACS_BSBS	alternate name for horizontal line
curses.ACS_BSSB	alternate name for upper left corner
curses.ACS_BSSS	alternate name for top tee
curses.ACS_BTEE	bottom tee
curses.ACS_BULLET	bullet
curses.ACS_CKBOARD	checker board (stipple)
curses.ACS_DARROW	arrow pointing down
curses.ACS_DEGREE	degree symbol
curses.ACS_DIAMOND	diamond
curses.ACS_GEQUAL	greater-than-or-equal-to
curses.ACS_HLINE	horizontal line

ACS code	Meaning
curses.ACS_LANTERN	lantern symbol
curses.ACS_LARROW	left arrow
curses.ACS_LEQUAL	less-than-or-equal-to
curses.ACS_LLCORNER	lower left-hand corner
curses.ACS_LRCORNER	lower right-hand corner
curses.ACS_LTEE	left tee
curses.ACS_NEQUAL	not-equal sign
curses.ACS_PI	letter pi
curses.ACS_PLMINUS	plus-or-minus sign
curses.ACS_PLUS	big plus sign
curses.ACS_RARROW	right arrow
curses.ACS_RTEE	right tee

ACS code	Meaning
curses.ACS_S1	scan line 1
curses.ACS_S3	scan line 3
curses.ACS_S7	scan line 7
curses.ACS_S9	scan line 9
curses.ACS_SBBS	alternate name for lower right corner
curses.ACS_SBSB	alternate name for vertical line
curses.ACS_SBSS	alternate name for right tee
curses.ACS_SSBB	alternate name for lower left corner
curses.ACS_SSBS	alternate name for bottom tee
curses.ACS_SSSB	alternate name for left tee
curses.ACS_SSSS	alternate name for crossover or big plus

ACS code	Meaning
curses.ACS_STERLING	pound sterling
curses.ACS_TTEE	top tee
curses.ACS_UARROW	up arrow
curses.ACS_ULCORNER	upper left corner
curses.ACS_URCORNER	upper right corner
curses.ACS_VLINE	vertical line

The following table lists mouse button constants used by `getmouse()`:

Mouse button constant	Meaning
curses.BUTTONn_PRESSED	Mouse button *n* pressed
curses.BUTTONn_RELEASED	Mouse button *n* released

Mouse button constant	**Meaning**
curses.BUTTONn_CLICKED	Mouse button *n* clicked
curses.BUTTONn_DOUBLE_CLICKED	Mouse button *n* double clicked
curses.BUTTONn_TRIPLE_CLICKED	Mouse button *n* triple clicked
curses.BUTTON_SHIFT	Shift was down during button state change
curses.BUTTON_CTRL	Control was down during button state change
curses.BUTTON_ALT	Control was down during button state change

Changed in version 3.10: The `BUTTON5_*` constants are now exposed if they are provided by the underlying curses library.

The following table lists the predefined colors:

Constant	**Color**
curses.COLOR_BLACK	Black
curses.COLOR_BLUE	Blue
curses.COLOR_CYAN	Cyan (light greenish blue)
curses.COLOR_GREEN	Green
curses.COLOR_MAGENTA	Magenta (purplish red)
curses.COLOR_RED	Red
curses.COLOR_WHITE	White
curses.COLOR_YELLOW	Yellow

`curses.textpad` — Text input widget for curses programs

The `curses.textpad` module provides a `Textbox` class that handles elementary text editing in a curses window, supporting a set of keybindings resembling those of Emacs (thus, also of Netscape Navigator, BBedit 6.x, FrameMaker, and many other programs). The module also provides a rectangle-drawing function useful for framing text boxes or for other purposes.

The module `curses.textpad` defines the following function:

curses.textpad.rectangle(*win*, *uly*, *ulx*, *lry*, *lrx*)

Draw a rectangle. The first argument must be a window object; the remaining arguments are coordinates relative to that window. The second and third arguments are the y and x coordinates of the upper left hand corner of the rectangle to be drawn; the fourth and fifth arguments are the y and x coordinates of the lower right hand corner. The rectangle will be drawn using VT100/IBM PC forms characters on terminals that make this possible (including xterm and most other software terminal emulators). Otherwise it will be drawn with ASCII dashes, vertical bars, and plus signs.

Textbox objects

You can instantiate a `Textbox` object as follows:

class curses.textpad.Textbox(*win*)

Return a textbox widget object. The *win* argument should be a curses window object in which the textbox is to be contained. The edit cursor of the textbox is initially located at the upper left hand corner of the containing window, with coordinates `(0, 0)`. The instance's `stripspaces` flag is initially on.

`Textbox` objects have the following methods:

edit([*validator*])

This is the entry point you will normally use. It accepts editing keystrokes until one of the termination keystrokes is entered. If *validator* is supplied, it must be a function. It will be called for each keystroke entered with the keystroke as a parameter; command dispatch is done on the result. This method returns the window contents as a string; whether blanks in the window are included is affected by the `stripspaces` attribute.

do_command(*ch*)

Process a single command keystroke. Here are the supported special keystrokes:

Keystroke	**Action**
`Control-A`	Go to left edge of window.
`Control-B`	Cursor left, wrapping to previous line if appropriate.
`Control-D`	Delete character under cursor.
`Control-E`	Go to right edge (stripspaces off) or end of line (stripspaces on).
`Control-F`	Cursor right, wrapping to next line when appropriate.
`Control-G`	Terminate, returning the window contents.
`Control-H`	Delete character backward.

Keystroke	Action
`Control-J`	Terminate if the window is 1 line, otherwise insert newline.
`Control-K`	If line is blank, delete it, otherwise clear to end of line.
`Control-L`	Refresh screen.
`Control-N`	Cursor down; move down one line.
`Control-O`	Insert a blank line at cursor location.
`Control-P`	Cursor up; move up one line.

Move operations do nothing if the cursor is at an edge where the movement is not possible. The following synonyms are supported where possible:

Constant	Keystroke
`KEY_LEFT`	`Control-B`
`KEY_RIGHT`	`Control-F`
`KEY_UP`	`Control-P`
`KEY_DOWN`	`Control-N`
`KEY_BACKSPACE`	`Control-h`

All other keystrokes are treated as a command to insert the given character and move right (with line wrapping).

gather()

Return the window contents as a string; whether blanks in the window are included is affected by the `stripspaces` member.

stripspaces

This attribute is a flag which controls the interpretation of blanks in the window. When it is on, trailing blanks on each line are ignored; any cursor motion that would land the cursor on a trailing blank goes to the end of that line instead, and trailing blanks are stripped when the window contents are gathered.

`curses.ascii` — Utilities for ASCII characters

Source code: Lib/curses/ascii.py [https://github.com/python/cpython/tree/3.14/Lib/curses/ascii.py]

The `curses.ascii` module supplies name constants for ASCII characters and functions to test membership in various ASCII character classes. The constants supplied are names for control characters as follows:

Name	Meaning
curses.ascii.NUL	
curses.ascii.SOH	Start of heading, console interrupt
curses.ascii.STX	Start of text
curses.ascii.ETX	End of text
curses.ascii.EOT	End of transmission
curses.ascii.ENQ	Enquiry, goes with `ACK` flow control

Name	Meaning
curses.ascii.ACK	Acknowledgement
curses.ascii.BEL	Bell
curses.ascii.BS	Backspace
curses.ascii.TAB	Tab
curses.ascii.HT	Alias for `TAB`: “Horizontal tab”
curses.ascii.LF	Line feed
curses.ascii.NL	Alias for `LF`: “New line”
curses.ascii.VT	Vertical tab
curses.ascii.FF	Form feed
curses.ascii.CR	Carriage return
curses.ascii.SO	Shift-out, begin alternate character set
curses.ascii.SI	Shift-in, resume default character set

Name	**Meaning**
curses.ascii.DLE	Data-link escape
curses.ascii.DC1	XON, for flow control
curses.ascii.DC2	Device control 2, block-mode flow control
curses.ascii.DC3	XOFF, for flow control
curses.ascii.DC4	Device control 4
curses.ascii.NAK	Negative acknowledgement
curses.ascii.SYN	Synchronous idle
curses.ascii.ETB	End transmission block
curses.ascii.CAN	Cancel
curses.ascii.EM	End of medium
curses.ascii.SUB	Substitute
curses.ascii.ESC	Escape

Name	**Meaning**
curses.ascii.FS	File separator
curses.ascii.GS	Group separator
curses.ascii.RS	Record separator, block-mode terminator
curses.ascii.US	Unit separator
curses.ascii.SP	Space
curses.ascii.DEL	Delete

Note that many of these have little practical significance in modern usage. The mnemonics derive from teleprinter conventions that predate digital computers.

The module supplies the following functions, patterned on those in the standard C library:

curses.ascii.isalnum(*c*)

Checks for an ASCII alphanumeric character; it is equivalent to `isalpha(c) or isdigit(c)`.

curses.ascii.isalpha(*c*)

Checks for an ASCII alphabetic character; it is equivalent to `isupper(c) or islower(c)`.

curses.ascii.isascii(*c*)

Checks for a character value that fits in the 7-bit ASCII set.

curses.ascii.isblank(*c*)

Checks for an ASCII whitespace character; space or horizontal tab.

curses.ascii.iscntrl(*c*)

Checks for an ASCII control character (in the range 0x00 to 0x1f or 0x7f).

curses.ascii.isdigit(*c*)

Checks for an ASCII decimal digit, `'0'` through `'9'`. This is equivalent to `c in string.digits`.

curses.ascii.isgraph(*c*)

Checks for ASCII any printable character except space.

curses.ascii.islower(*c*)

Checks for an ASCII lower-case character.

curses.ascii.isprint(*c*)

Checks for any ASCII printable character including space.

curses.ascii.ispunct(*c*)

Checks for any printable ASCII character which is not a space or an alphanumeric character.

curses.ascii.isspace(*c*)

Checks for ASCII white-space characters; space, line feed, carriage return, form feed, horizontal tab, vertical tab.

curses.ascii.isupper(*c*)

Checks for an ASCII uppercase letter.

curses.ascii.isxdigit(*c*)

Checks for an ASCII hexadecimal digit. This is equivalent to `c in string.hexdigits`.

curses.ascii.isctrl(*c*)

Checks for an ASCII control character (ordinal values 0 to 31).

curses.ascii.ismeta(*c*)

Checks for a non-ASCII character (ordinal values 0x80 and above).

These functions accept either integers or single-character strings; when the argument is a string, it is first converted using the built-in function `ord()`.

Note that all these functions check ordinal bit values derived from the character of the string you pass in; they do not actually know anything about the host machine’s character encoding.

The following two functions take either a single-character string or integer byte value; they return a value of the same type.

curses.ascii.ascii(*c*)

Return the ASCII value corresponding to the low 7 bits of *c*.

curses.ascii.ctrl(*c*)

Return the control character corresponding to the given character (the character bit value is bitwise-anded with 0x1f).

curses.ascii.alt(*c*)

Return the 8-bit character corresponding to the given ASCII character (the character bit value is bitwise-ored with 0x80).

The following function takes either a single-character string or integer value; it returns a string.

curses.ascii.unctrl(*c*)

Return a string representation of the ASCII character *c*. If *c* is printable, this string is the character itself. If the character is a control character (0x00–0x1f) the string consists of a caret (`'^'`) followed by the corresponding uppercase letter. If the character is an ASCII delete (0x7f) the string is `'^?'`. If the character has its meta bit (0x80) set, the meta bit is stripped, the preceding rules applied, and `'!'` prepended to the result.

curses.ascii.controlnames

A 33-element string array that contains the ASCII mnemonics for the thirty-two ASCII control characters from 0 (NUL) to 0x1f (US), in order, plus the mnemonic SP for the space character.

`curses.panel` — A panel stack extension for curses

Panels are windows with the added feature of depth, so they can be stacked on top of each other, and only the visible portions of each window will be displayed. Panels can be added, moved up or down in the stack, and removed.

Functions

The module `curses.panel` defines the following functions:

curses.panel.bottom_panel()

Returns the bottom panel in the panel stack.

curses.panel.new_panel(*win*)

Returns a panel object, associating it with the given window *win*. Be aware that you need to keep the returned panel object referenced explicitly. If you don’t, the panel object is garbage collected and removed from the panel stack.

curses.panel.top_panel()

Returns the top panel in the panel stack.

curses.panel.update_panels()

Updates the virtual screen after changes in the panel stack. This does not call `curses.doupdate()`, so you’ll have to do this yourself.

Panel Objects

Panel objects, as returned by `new_panel()` above, are windows with a stacking order. There's always a window associated with a panel which determines the content, while the panel methods are responsible for the window's depth in the panel stack.

Panel objects have the following methods:

Panel.above()

Returns the panel above the current panel.

Panel.below()

Returns the panel below the current panel.

Panel.bottom()

Push the panel to the bottom of the stack.

Panel.hidden()

Returns `True` if the panel is hidden (not visible), `False` otherwise.

Panel.hide()

Hide the panel. This does not delete the object, it just makes the window on screen invisible.

Panel.move(*y*, *x*)

Move the panel to the screen coordinates `(y, x)`.

Panel.replace(*win*)

Change the window associated with the panel to the window *win*.

Panel.set_userptr(*obj*)

Set the panel’s user pointer to *obj*. This is used to associate an arbitrary piece of data with the panel, and can be any Python object.

Panel.show()

Display the panel (which might have been hidden).

Panel.top()

Push panel to the top of the stack.

Panel.userptr()

Returns the user pointer for the panel. This might be any Python object.

Panel.window()

Returns the window object associated with the panel.

`cmd` — Support for line-oriented command interpreters

Source code: Lib/cmd.py
[https://github.com/python/cpython/tree/3.14/Lib/cmd.py]

The **`Cmd`** class provides a simple framework for writing line-oriented command interpreters. These are often useful for test harnesses, administrative tools, and prototypes that will later be wrapped in a more sophisticated interface.

class cmd.Cmd(*completekey*='*tab*', *stdin=None*, *stdout=None*)

A **`Cmd`** instance or subclass instance is a line-oriented interpreter framework. There is no good reason to instantiate **`Cmd`** itself; rather, it's useful as a superclass of an interpreter class you define yourself in order to inherit **`Cmd`**'s methods and encapsulate action methods.

The optional argument *completekey* is the **`readline`** name of a completion key; it defaults to `Tab`. If *completekey* is not **`None`** and **`readline`** is available, command completion is done automatically.

The default, `'tab'`, is treated specially, so that it refers to the `Tab` key on every **`readline.backend`**. Specifically, if **`readline.backend`** is `editline`, `Cmd` will use `'^I'` instead of `'tab'`. Note that other values are not treated this way, and might only work with a specific backend.

The optional arguments *stdin* and *stdout* specify the input and output file objects that the Cmd instance or subclass instance will use for input and output. If not specified, they will default to `sys.stdin` and `sys.stdout`.

If you want a given *stdin* to be used, make sure to set the instance's `use_rawinput` attribute to `False`, otherwise *stdin* will be ignored.

Changed in version 3.13: `completekey='tab'` is replaced by `'^I'` for `editline`.

Cmd Objects

A `Cmd` instance has the following methods:

Cmd.cmdloop(*intro=None*)

Repeatedly issue a prompt, accept input, parse an initial prefix off the received input, and dispatch to action methods, passing them the remainder of the line as argument.

The optional argument is a banner or intro string to be issued before the first prompt (this overrides the `intro` class attribute).

If the `readline` module is loaded, input will automatically inherit **bash**-like history-list editing (e.g. `Control-P` scrolls back to the last command, `Control-N` forward to the next one, `Control-F` moves the cursor to the right non-destructively, `Control-B` moves the cursor to the left non-destructively, etc.).

An end-of-file on input is passed back as the string `'EOF'`.

An interpreter instance will recognize a command name `foo` if and only if it has a method **`do_foo()`**. As a special case, a line beginning with the character `'?'` is dispatched to the method **`do_help()`**. As another special case, a line beginning with the character `'!'` is dispatched to the method **`do_shell()`** (if such a method is defined).

This method will return when the **`postcmd()`** method returns a true value. The *stop* argument to **`postcmd()`** is the return value from the command's corresponding **`do_*()`** method.

If completion is enabled, completing commands will be done automatically, and completing of commands args is done by calling **`complete_foo()`** with arguments *text*, *line*, *begidx*, and *endidx*. *text* is the string prefix we are attempting to match: all returned matches must begin with it. *line* is the current input line with leading whitespace removed, *begidx* and *endidx* are the beginning and ending indexes of the prefix text, which could be used to provide different completion depending upon which position the argument is in.

Cmd.do_help(*arg*)

All subclasses of **`Cmd`** inherit a predefined **`do_help()`**. This method, called with an argument `'bar'`, invokes the corresponding method **`help_bar()`**, and if that is not present, prints the docstring of **`do_bar()`**, if available. With no argument, **`do_help()`** lists all available help topics (that is, all commands with corresponding **`help_*()`** methods or commands that have docstrings), and also lists any undocumented commands.

Cmd.onecmd(*str*)

Interpret the argument as though it had been typed in response to the prompt. This may be overridden, but should not normally need to be; see the `precmd()` and `postcmd()` methods for useful execution hooks. The return value is a flag indicating whether interpretation of commands by the interpreter should stop. If there is a **`do_*()`** method for the command *str*, the return value of that method is returned, otherwise the return value from the `default()` method is returned.

Cmd.emptyline()

Method called when an empty line is entered in response to the prompt. If this method is not overridden, it repeats the last nonempty command entered.

Cmd.default(*line*)

Method called on an input line when the command prefix is not recognized. If this method is not overridden, it prints an error message and returns.

Cmd.completedefault(*text*, *line*, *begidx*, *endidx*)

Method called to complete an input line when no command-specific **`complete_*()`** method is available. By default, it returns an empty list.

Cmd.columnize(*list*, *displaywidth=80*)

Method called to display a list of strings as a compact set of columns. Each column is only as wide as necessary. Columns are separated by two spaces for readability.

Cmd.precmd(*line*)

Hook method executed just before the command line *line* is interpreted, but after the input prompt is generated and issued. This method is a stub in `Cmd`; it exists to be overridden by subclasses. The return value is used as the command which will be executed by the `onecmd()` method; the `precmd()` implementation may re-write the command or simply return *line* unchanged.

Cmd.postcmd(*stop*, *line*)

Hook method executed just after a command dispatch is finished. This method is a stub in `Cmd`; it exists to be overridden by subclasses. *line* is the command line which was executed, and *stop* is a flag which indicates whether execution will be terminated after the call to `postcmd()`; this will be the return value of the `onecmd()` method. The return value of this method will be used as the new value for the internal flag which corresponds to *stop*; returning false will cause interpretation to continue.

Cmd.preloop()

Hook method executed once when `cmdloop()` is called. This method is a stub in `Cmd`; it exists to be overridden by subclasses.

Cmd.postloop()

Hook method executed once when `cmdloop()` is about to return. This method is a stub in `Cmd`; it exists to be overridden by subclasses.

Instances of `Cmd` subclasses have some public instance variables:

Cmd.prompt

The prompt issued to solicit input.

Cmd.identchars

The string of characters accepted for the command prefix.

Cmd.lastcmd

The last nonempty command prefix seen.

Cmd.cmdqueue

A list of queued input lines. The cmdqueue list is checked in `cmdloop()` when new input is needed; if it is nonempty, its elements will be processed in order, as if entered at the prompt.

Cmd.intro

A string to issue as an intro or banner. May be overridden by giving the `cmdloop()` method an argument.

Cmd.doc_header

The header to issue if the help output has a section for documented commands.

Cmd.misc_header

The header to issue if the help output has a section for miscellaneous help topics (that is, there are **`help_*()`** methods without corresponding **`do_*()`** methods).

Cmd.undoc_header

The header to issue if the help output has a section for undocumented commands (that is, there are **`do_*()`** methods without corresponding **`help_*()`** methods).

Cmd.ruler

The character used to draw separator lines under the help-message headers. If empty, no ruler line is drawn. It defaults to `'='`.

Cmd.use_rawinput

A flag, defaulting to true. If true, `cmdloop()` uses `input()` to display a prompt and read the next command; if false, `sys.stdout.write()` and `sys.stdin.readline()` are used. (This means that by importing `readline`, on systems that support it, the interpreter will automatically support **Emacs**-like line editing and command-history keystrokes.)

Cmd Example

The `cmd` module is mainly useful for building custom shells that let a user work with a program interactively.

This section presents a simple example of how to build a shell around a few of the commands in the `turtle` module.

Basic turtle commands such as `forward()` are added to a `Cmd` subclass with method named **`do_forward()`**. The argument is converted to a number and dispatched to the turtle module. The docstring is used in the help utility provided by the shell.

The example also includes a basic record and playback facility implemented with the `precmd()` method which is responsible for converting the input to lowercase and writing the commands to a file. The **`do_playback()`** method reads the file and adds the recorded commands to the `cmdqueue` for immediate playback:

```
import cmd, sys
from turtle import *

class TurtleShell(cmd.Cmd):
    intro = 'Welcome to the turtle shell.   Type help or ? to 
list commands.\n'
    prompt = '(turtle) '
    file = None

    # ----- basic turtle commands -----
    def do_forward(self, arg):
        'Move the turtle forward by the specified distance:  
FORWARD 10'
```

```
        forward(*parse(arg))
    def do_right(self, arg):
        'Turn turtle right by given number of degrees:  RIGHT 
20'
        right(*parse(arg))
    def do_left(self, arg):
        'Turn turtle left by given number of degrees:  LEFT 90'
        left(*parse(arg))
    def do_goto(self, arg):
        'Move turtle to an absolute position with changing 
orientation.  GOTO 100 200'
        goto(*parse(arg))
    def do_home(self, arg):
        'Return turtle to the home position:  HOME'
        home()
    def do_circle(self, arg):
        'Draw circle with given radius an options extent and 
steps:  CIRCLE 50'
        circle(*parse(arg))
    def do_position(self, arg):
        'Print the current turtle position:  POSITION'
        print('Current position is %d %d\n' % position())
    def do_heading(self, arg):
        'Print the current turtle heading in degrees:  HEADING'
        print('Current heading is %d\n' % (heading(),))
    def do_color(self, arg):
        'Set the color:  COLOR BLUE'
        color(arg.lower())
    def do_undo(self, arg):
        'Undo (repeatedly) the last turtle action(s):  UNDO'
    def do_reset(self, arg):
        'Clear the screen and return turtle to center:  RESET'
        reset()
    def do_bye(self, arg):
        'Stop recording, close the turtle window, and exit:  
BYE'
        print('Thank you for using Turtle')
        self.close()
        bye()
        return True

    # ----- record and playback -----
    def do_record(self, arg):
        'Save future commands to filename:  RECORD rose.cmd'
        self.file = open(arg, 'w')
    def do_playback(self, arg):
        'Playback commands from a file:  PLAYBACK rose.cmd'
        self.close()
        with open(arg) as f:
            self.cmdqueue.extend(f.read().splitlines())
    def precmd(self, line):
        line = line.lower()
        if self.file and 'playback' not in line:
            print(line, file=self.file)
        return line
    def close(self):
        if self.file:
            self.file.close()
            self.file = None

def parse(arg):
    'Convert a series of zero or more numbers to an argument
tuple'
    return tuple(map(int, arg.split()))

if __name__ == '__main__':
    TurtleShell().cmdloop()
```

Here is a sample session with the turtle shell showing the help functions, using blank lines to repeat commands, and the simple record and playback facility:

```
Welcome to the turtle shell.   Type help or ? to list commands.

(turtle) ?

Documented commands (type help <topic>):
========================================
bye     color    goto     home  playback  record  right
circle  forward  heading  left  position  reset   undo

(turtle) help forward
Move the turtle forward by the specified distance:  FORWARD 10
(turtle) record spiral.cmd
(turtle) position
```

```
Current position is 0 0

(turtle) heading
Current heading is 0

(turtle) reset
(turtle) circle 20
(turtle) right 30
(turtle) circle 40
(turtle) right 30
(turtle) circle 60
(turtle) right 30
(turtle) circle 80
(turtle) right 30
(turtle) circle 100
(turtle) right 30
(turtle) circle 120
(turtle) right 30
(turtle) circle 120
(turtle) heading
Current heading is 180

(turtle) forward 100
(turtle)
(turtle) right 90
(turtle) forward 100
(turtle)
(turtle) right 90
(turtle) forward 400
(turtle) right 90
(turtle) forward 500
(turtle) right 90
(turtle) forward 400
(turtle) right 90
(turtle) forward 300
(turtle) playback spiral.cmd
Current position is 0 0

Current heading is 0

Current heading is 180
```

```
(turtle) bye
Thank you for using Turtle
```

Concurrent Execution

The modules described in this chapter provide support for concurrent execution of code. The appropriate choice of tool will depend on the task to be executed (CPU bound vs IO bound) and preferred style of development (event driven cooperative multitasking vs preemptive multitasking). Here’s an overview:

- `threading` — Thread-based parallelism
  - Introduction
  - GIL and performance considerations
  - Reference
    - Thread-local data
    - Thread objects
    - Lock objects
    - RLock objects
    - Condition objects
    - Semaphore objects
    - `Semaphore` example
    - Event objects
    - Timer objects
    - Barrier objects
  - Using locks, conditions, and semaphores in the `with` statement
- `multiprocessing` — Process-based parallelism
  - Introduction
    - The `Process` class
    - Contexts and start methods
    - Exchanging objects between processes
    - Synchronization between processes
    - Sharing state between processes
    - Using a pool of workers

    - Reference
        - Global start method
        - `Process` and exceptions
        - Pipes and Queues
        - Miscellaneous
        - Connection Objects
        - Synchronization primitives
        - Shared `ctypes` Objects
            - The `multiprocessing.sharedctypes` module
        - Managers
            - Customized managers
            - Using a remote manager
        - Proxy Objects
            - Cleanup
        - Process Pools
        - Listeners and Clients
            - Address Formats
        - Authentication keys
        - Logging
        - The `multiprocessing.dummy` module
    - Programming guidelines
        - All start methods
        - The *spawn* and *forkserver* start methods
    - Examples
- `multiprocessing.shared_memory` — Shared memory for direct access across processes
- The `concurrent` package
- `concurrent.futures` — Launching parallel tasks
    - Executor Objects
    - ThreadPoolExecutor
        - ThreadPoolExecutor Example
    - InterpreterPoolExecutor
    - ProcessPoolExecutor
        - ProcessPoolExecutor Example
    - Future Objects
    - Module Functions

    - Exception classes
- `concurrent.interpreters` — Multiple interpreters in the same process
    - Key details
    - Introduction
        - Multiple Interpreters and Isolation
        - Running in an Interpreter
        - Concurrency and Parallelism
        - Communication Between Interpreters
        - “Sharing” Objects
    - Reference
        - Interpreter objects
        - Exceptions
        - Communicating Between Interpreters
    - Basic usage
- `subprocess` — Subprocess management
    - Using the `subprocess` Module
        - Frequently Used Arguments
        - Popen Constructor
        - Exceptions
    - Security Considerations
    - Popen Objects
    - Windows Popen Helpers
        - Windows Constants
    - Older high-level API
    - Replacing Older Functions with the `subprocess` Module
        - Replacing **/bin/sh** shell command substitution
        - Replacing shell pipeline
        - Replacing `os.system()`
        - Replacing the `os.spawn` family
        - Replacing `os.popen()`
    - Legacy Shell Invocation Functions
    - Notes
        - Timeout Behavior
        - Converting an argument sequence to a string on Windows

        - Disable use of `posix_spawn()`
- `sched` — Event scheduler
    - Scheduler Objects
- `queue` — A synchronized queue class
    - Queue Objects
        - Waiting for task completion
        - Terminating queues
    - SimpleQueue Objects
- `contextvars` — Context Variables
    - Context Variables
    - Manual Context Management
    - asyncio support

The following are support modules for some of the above services:

- `_thread` — Low-level threading API

`threading` — Thread-based parallelism

Source code: Lib/threading.py [https://github.com/python/cpython/tree/3.14/Lib/threading.py]

This module constructs higher-level threading interfaces on top of the lower level **`_thread`** module.

Availability: not WASI.

This module does not work or is not available on WebAssembly. See WebAssembly platforms for more information.

Introduction

The **`threading`** module provides a way to run multiple threads [https://en.wikipedia.org/wiki/Thread_(computing)] (smaller units of a process) concurrently within a single process. It allows for the creation and management of threads, making it possible to execute tasks in parallel, sharing memory space. Threads are particularly useful when tasks are I/O bound, such as file operations or making network requests, where much of the time is spent waiting for external resources.

A typical use case for **`threading`** includes managing a pool of worker threads that can process multiple tasks concurrently. Here's a basic example of creating and starting threads using **`Thread`**:

```
import threading
import time

def crawl(link, delay=3):
    print(f"crawl started for {link}")
    time.sleep(delay)  # Blocking I/O (simulating a network
request)
    print(f"crawl ended for {link}")

links = [
    "https://python.org",
    "https://docs.python.org",
    "https://peps.python.org",
]

# Start threads for each link
threads = []
for link in links:
    # Using `args` to pass positional arguments and `kwargs`
for keyword arguments
    t = threading.Thread(target=crawl, args=(link,), kwargs=
{"delay": 2})
    threads.append(t)

# Start each thread
for t in threads:
    t.start()

# Wait for all threads to finish
for t in threads:
    t.join()
```

Changed in version 3.7: This module used to be optional, it is now always available.

See also

`concurrent.futures.ThreadPoolExecutor` offers a higher level interface to push tasks to a background thread without blocking execution of the calling thread, while still being able to retrieve their results when needed.

`queue` provides a thread-safe interface for exchanging data between running threads.

`asyncio` offers an alternative approach to achieving task level concurrency without requiring the use of multiple operating system threads.

Note

In the Python 2.x series, this module contained `camelCase` names for some methods and functions. These are deprecated as of Python 3.10, but they are still supported for compatibility with Python 2.5 and lower.

CPython implementation detail: In CPython, due to the Global Interpreter Lock, only one thread can execute Python code at once (even though certain performance-oriented libraries might overcome this limitation). If you want your application to make better use of the computational resources of multi-core machines, you are advised to use `multiprocessing` or `concurrent.futures.ProcessPoolExecutor`. However, threading is still an appropriate model if you want to run multiple I/O-bound tasks simultaneously.

GIL and performance considerations

Unlike the `multiprocessing` module, which uses separate processes to bypass the global interpreter lock (GIL), the threading module operates within a single process, meaning that all threads share the same memory space. However, the GIL limits the performance gains of threading when it comes to CPU-bound tasks, as only one thread can execute Python bytecode at a time. Despite this, threads remain a useful tool for achieving concurrency in many scenarios.

As of Python 3.13, free-threaded builds can disable the GIL, enabling true parallel execution of threads, but this feature is not available by default (see **PEP 703** [https://peps.python.org/pep-0703/]).

Reference

This module defines the following functions:

threading.active_count()

Return the number of **`Thread`** objects currently alive. The returned count is equal to the length of the list returned by **`enumerate()`**.

The function `activeCount` is a deprecated alias for this function.

threading.current_thread()

Return the current **`Thread`** object, corresponding to the caller’s thread of control. If the caller’s thread of control was not created through the **`threading`** module, a dummy thread object with limited functionality is returned.

The function `currentThread` is a deprecated alias for this function.

threading.excepthook(*args*, */*)

Handle uncaught exception raised by **`Thread.run()`**.

The *args* argument has the following attributes:

- *exc_type*: Exception type.
- *exc_value*: Exception value, can be `None`.
- *exc_traceback*: Exception traceback, can be `None`.
- *thread*: Thread which raised the exception, can be `None`.

If *exc_type* is `SystemExit`, the exception is silently ignored. Otherwise, the exception is printed out on `sys.stderr`.

If this function raises an exception, `sys.excepthook()` is called to handle it.

`threading.excepthook()` can be overridden to control how uncaught exceptions raised by `Thread.run()` are handled.

Storing *exc_value* using a custom hook can create a reference cycle. It should be cleared explicitly to break the reference cycle when the exception is no longer needed.

Storing *thread* using a custom hook can resurrect it if it is set to an object which is being finalized. Avoid storing *thread* after the custom hook completes to avoid resurrecting objects.

See also

`sys.excepthook()` handles uncaught exceptions.

Added in version 3.8.

threading.__excepthook__

Holds the original value of `threading.excepthook()`. It is saved so that the original value can be restored in case they happen to get replaced with broken or alternative objects.

Added in version 3.10.

threading.get_ident()

Return the ‘thread identifier’ of the current thread. This is a nonzero integer. Its value has no direct meaning; it is intended as a magic cookie to be used e.g. to index a dictionary of thread-specific data. Thread identifiers may be recycled when a thread exits and another thread is created.

Added in version 3.3.

threading.get_native_id()

Return the native integral Thread ID of the current thread assigned by the kernel. This is a non-negative integer. Its value may be used to uniquely identify this particular thread system-wide (until the thread terminates, after which the value may be recycled by the OS).

Availability: Windows, FreeBSD, Linux, macOS, OpenBSD, NetBSD, AIX, DragonFlyBSD, GNU/kFreeBSD.

Added in version 3.8.

Changed in version 3.13: Added support for GNU/kFreeBSD.

threading.enumerate()

Return a list of all `Thread` objects currently active. The list includes daemonic threads and dummy thread objects created by `current_thread()`. It excludes terminated threads and threads that have not yet been started. However, the main thread is always part of the result, even when terminated.

threading.main_thread()

Return the main `Thread` object. In normal conditions, the main thread is the thread from which the Python interpreter was started.

Added in version 3.4.

threading.settrace(*func*)

Set a trace function for all threads started from the `threading` module. The *func* will be passed to `sys.settrace()` for each thread, before its `run()` method is called.

threading.settrace_all_threads(*func*)

Set a trace function for all threads started from the `threading` module and all Python threads that are currently executing.

The *func* will be passed to `sys.settrace()` for each thread, before its `run()` method is called.

Added in version 3.12.

threading.gettrace()

Get the trace function as set by `settrace()`.

Added in version 3.10.

threading.setprofile(*func*)

Set a profile function for all threads started from the `threading` module. The *func* will be passed to `sys.setprofile()` for each thread, before its `run()` method is called.

threading.setprofile_all_threads(*func*)

Set a profile function for all threads started from the `threading` module and all Python threads that are currently executing.

The *func* will be passed to `sys.setprofile()` for each thread, before its `run()` method is called.

Added in version 3.12.

threading.getprofile()

Get the profiler function as set by `setprofile()`.

Added in version 3.10.

threading.stack_size([*size*])

Return the thread stack size used when creating new threads. The optional *size* argument specifies the stack size to be used for subsequently created threads, and must be 0 (use platform or configured default) or a positive integer value of at least 32,768 (32 KiB). If *size* is not specified, 0 is used. If changing the thread stack size is unsupported, a `RuntimeError` is raised. If the specified stack size is invalid, a `ValueError` is raised and the stack size is unmodified. 32 KiB is currently the minimum supported stack size value to guarantee sufficient stack space for the interpreter itself. Note that some platforms may have particular restrictions on values for the stack size, such as requiring a minimum stack size > 32 KiB or requiring allocation in multiples of the system memory page size - platform documentation should be referred to for more information (4 KiB pages are common; using multiples of 4096 for the stack size is the suggested approach in the absence of more specific information).

Availability: Windows, pthreads.

Unix platforms with POSIX threads support.

This module also defines the following constant:

threading.TIMEOUT_MAX

The maximum value allowed for the *timeout* parameter of blocking functions (`Lock.acquire()`, `RLock.acquire()`, `Condition.wait()`, etc.). Specifying a timeout greater than this value will raise an `OverflowError`.

Added in version 3.2.

This module defines a number of classes, which are detailed in the sections below.

The design of this module is loosely based on Java’s threading model. However, where Java makes locks and condition variables basic behavior of every object, they are separate objects in Python. Python’s `Thread` class supports a subset of the behavior of Java’s Thread class; currently, there are no priorities, no thread groups, and threads cannot be destroyed, stopped, suspended, resumed, or interrupted. The static methods of Java’s Thread class, when implemented, are mapped to module-level functions.

All of the methods described below are executed atomically.

Thread-local data

Thread-local data is data whose values are thread specific. If you have data that you want to be local to a thread, create a `local` object and use its attributes:

```
>>> mydata = local()
>>> mydata.number = 42
>>> mydata.number
42
```

You can also access the `local`-object’s dictionary:

```
>>> mydata.__dict__
{'number': 42}
>>> mydata.__dict__.setdefault('widgets', [])
[]
>>> mydata.widgets
[]
```

If we access the data in a different thread:

```
>>> log = []
>>> def f():
...     items = sorted(mydata.__dict__.items())
...     log.append(items)
...     mydata.number = 11
...     log.append(mydata.number)

>>> import threading
>>> thread = threading.Thread(target=f)
>>> thread.start()
>>> thread.join()
>>> log
[[], 11]
```

we get different data. Furthermore, changes made in the other thread don’t affect data seen in this thread:

```
>>> mydata.number
42
```

Of course, values you get from a `local` object, including their `__dict__` attribute, are for whatever thread was current at the time the attribute was read. For that reason, you generally don’t want to save these values across threads, as they apply only to the thread they came from.

You can create custom `local` objects by subclassing the `local` class:

```
>>> class MyLocal(local):
...     number = 2
...     def __init__(self, /, **kw):
...         self.__dict__.update(kw)
...     def squared(self):
...         return self.number ** 2
```

This can be useful to support default values, methods and initialization. Note that if you define an `__init__()` method, it will be called each time the `local` object is used in a separate thread. This is necessary to initialize each thread’s dictionary.

Now if we create a `local` object:

```
>>> mydata = MyLocal(color='red')
```

we have a default number:

```
>>> mydata.number
2
```

an initial color:

```
>>> mydata.color
'red'
>>> del mydata.color
```

And a method that operates on the data:

```
>>> mydata.squared()
4
```

As before, we can access the data in a separate thread:

```
>>> log = []
>>> thread = threading.Thread(target=f)
>>> thread.start()
>>> thread.join()
>>> log
[[('color', 'red')], 11]
```

without affecting this thread’s data:

```
>>> mydata.number
2
>>> mydata.color
Traceback (most recent call last):
...
AttributeError: 'MyLocal' object has no attribute 'color'
```

Note that subclasses can define __slots__, but they are not thread local. They are shared across threads:

```
>>> class MyLocal(local):
...     __slots__ = 'number'

>>> mydata = MyLocal()
>>> mydata.number = 42
>>> mydata.color = 'red'
```

So, the separate thread:

```
>>> thread = threading.Thread(target=f)
>>> thread.start()
>>> thread.join()
```

affects what we see:

```
>>> mydata.number
11
```

class threading.local

A class that represents thread-local data.

Thread objects

The **`Thread`** class represents an activity that is run in a separate thread of control. There are two ways to specify the activity: by passing a callable object to the constructor, or by overriding the **`run()`** method in a subclass. No other methods (except for the constructor) should be overridden in a subclass. In other words, *only* override the `__init__()` and **`run()`** methods of this class.

Once a thread object is created, its activity must be started by calling the thread’s **`start()`** method. This invokes the **`run()`** method in a separate thread of control.

Once the thread's activity is started, the thread is considered 'alive'. It stops being alive when its `run()` method terminates – either normally, or by raising an unhandled exception. The `is_alive()` method tests whether the thread is alive.

Other threads can call a thread's `join()` method. This blocks the calling thread until the thread whose `join()` method is called is terminated.

A thread has a name. The name can be passed to the constructor, and read or changed through the `name` attribute.

If the `run()` method raises an exception, `threading.excepthook()` is called to handle it. By default, `threading.excepthook()` ignores silently `SystemExit`.

A thread can be flagged as a "daemon thread". The significance of this flag is that the entire Python program exits when only daemon threads are left. The initial value is inherited from the creating thread. The flag can be set through the `daemon` property or the *daemon* constructor argument.

Note

Daemon threads are abruptly stopped at shutdown. Their resources (such as open files, database transactions, etc.) may not be released properly. If you want your threads to stop gracefully, make them non-daemonic and use a suitable signalling mechanism such as an `Event`.

There is a "main thread" object; this corresponds to the initial thread of control in the Python program. It is not a daemon thread.

There is the possibility that “dummy thread objects” are created. These are thread objects corresponding to “alien threads”, which are threads of control started outside the threading module, such as directly from C code. Dummy thread objects have limited functionality; they are always considered alive and daemonic, and cannot be joined. They are never deleted, since it is impossible to detect the termination of alien threads.

class threading.Thread(*group=None*, *target=None*, *name=None*, *args=()*, *kwargs={}*, *, *daemon=None*, *context=None*)

This constructor should always be called with keyword arguments. Arguments are:

group should be `None`; reserved for future extension when a **`ThreadGroup`** class is implemented.

target is the callable object to be invoked by the **`run()`** method. Defaults to `None`, meaning nothing is called.

name is the thread name. By default, a unique name is constructed of the form “Thread-*N*” where *N* is a small decimal number, or “Thread-*N* (target)” where “target” is `target.__name__` if the *target* argument is specified.

args is a list or tuple of arguments for the target invocation. Defaults to `()`.

kwargs is a dictionary of keyword arguments for the target invocation. Defaults to `{}`.

If not `None`, *daemon* explicitly sets whether the thread is daemonic. If `None` (the default), the daemonic property is inherited from the current thread.

context is the `Context` value to use when starting the thread. The default value is `None` which indicates that the `sys.flags.thread_inherit_context` flag controls the behaviour. If the flag is true, threads will start with a copy of the context of the caller of `start()`. If false, they will start with an empty context. To explicitly start with an empty context, pass a new instance of `Context()`. To explicitly start with a copy of the current context, pass the value from `copy_context()`. The flag defaults true on free-threaded builds and false otherwise.

If the subclass overrides the constructor, it must make sure to invoke the base class constructor (`Thread.__init__()`) before doing anything else to the thread.

Changed in version 3.3: Added the *daemon* parameter.

Changed in version 3.10: Use the *target* name if *name* argument is omitted.

Changed in version 3.14: Added the *context* parameter.

start()

Start the thread’s activity.

It must be called at most once per thread object. It arranges for the object’s `run()` method to be invoked in a separate thread of control.

This method will raise a `RuntimeError` if called more than once on the same thread object.

If supported, set the operating system thread name to `threading.Thread.name`. The name can be truncated depending on the operating system thread name limits.

Changed in version 3.14: Set the operating system thread name.

run()

Method representing the thread’s activity.

You may override this method in a subclass. The standard **`run()`** method invokes the callable object passed to the object’s constructor as the *target* argument, if any, with positional and keyword arguments taken from the *args* and *kwargs* arguments, respectively.

Using list or tuple as the *args* argument which passed to the **`Thread`** could achieve the same effect.

Example:

```
>>> from threading import Thread
>>> t = Thread(target=print, args=[1])
>>> t.run()
1
>>> t = Thread(target=print, args=(1,))
>>> t.run()
1
```

join(*timeout=None*)

Wait until the thread terminates. This blocks the calling thread until the thread whose **`join()`** method is called terminates – either normally or through an unhandled exception – or until the optional timeout occurs.

When the *timeout* argument is present and not `None`, it should be a floating-point number specifying a timeout for the operation in seconds (or fractions thereof). As **`join()`** always returns `None`, you must call **`is_alive()`** after **`join()`** to decide whether a timeout happened – if the thread is still alive, the `join()` call timed out.

When the *timeout* argument is not present or `None`, the operation will block until the thread terminates.

A thread can be joined many times.

`join()` raises a `RuntimeError` if an attempt is made to join the current thread as that would cause a deadlock. It is also an error to `join()` a thread before it has been started and attempts to do so raise the same exception.

If an attempt is made to join a running daemonic thread in late stages of Python finalization **`join()`** raises a `PythonFinalizationError`.

Changed in version 3.14: May raise `PythonFinalizationError`.

name

A string used for identification purposes only. It has no semantics. Multiple threads may be given the same name. The initial name is set by the constructor.

On some platforms, the thread name is set at the operating system level when the thread starts, so that it is visible in task managers. This name may be truncated to fit in a system-specific limit (for example, 15 bytes on Linux or 63 bytes on macOS).

Changes to *name* are only reflected at the OS level when the currently running thread is renamed. (Setting the *name* attribute of a different thread only updates the Python Thread object.)

getName()
setName()

Deprecated getter/setter API for `name`; use it directly as a property instead.

Deprecated since version 3.10.

ident

The ‘thread identifier’ of this thread or `None` if the thread has not been started. This is a nonzero integer. See the `get_ident()` function. Thread identifiers may be recycled when a thread exits and another thread is created. The identifier is available even after the thread has exited.

native_id

The Thread ID (`TID`) of this thread, as assigned by the OS (kernel). This is a non-negative integer, or `None` if the thread has not been started. See the `get_native_id()` function. This value may be used to uniquely identify this particular thread system-wide (until the thread terminates, after which the value may be recycled by the OS).

Note

Similar to Process IDs, Thread IDs are only valid (guaranteed unique system-wide) from the time the thread is created until the thread has been terminated.

Availability: Windows, FreeBSD, Linux, macOS, OpenBSD, NetBSD, AIX, DragonFlyBSD.

Added in version 3.8.

is_alive()

Return whether the thread is alive.

This method returns `True` just before the **`run()`** method starts until just after the **`run()`** method terminates. The module function **`enumerate()`** returns a list of all alive threads.

daemon

A boolean value indicating whether this thread is a daemon thread (`True`) or not (`False`). This must be set before **`start()`** is called, otherwise **`RuntimeError`** is raised. Its initial value is inherited from the creating thread; the main thread is not a daemon thread and therefore all threads created in the main thread default to **`daemon`** = `False`.

The entire Python program exits when no alive non-daemon threads are left.

isDaemon()

setDaemon()

Deprecated getter/setter API for **`daemon`**; use it directly as a property instead.

Deprecated since version 3.10.

Lock objects

A primitive lock is a synchronization primitive that is not owned by a particular thread when locked. In Python, it is currently the lowest level synchronization primitive available, implemented directly by the **`_thread`** extension module.

A primitive lock is in one of two states, “locked” or “unlocked”. It is created in the unlocked state. It has two basic methods, `acquire()` and `release()`. When the state is unlocked, `acquire()` changes the state to locked and returns immediately. When the state is locked, `acquire()` blocks until a call to `release()` in another thread changes it to unlocked, then the `acquire()` call resets it to locked and returns. The `release()` method should only be called in the locked state; it changes the state to unlocked and returns immediately. If an attempt is made to release an unlocked lock, a `RuntimeError` will be raised.

Locks also support the context management protocol.

When more than one thread is blocked in `acquire()` waiting for the state to turn to unlocked, only one thread proceeds when a `release()` call resets the state to unlocked; which one of the waiting threads proceeds is not defined, and may vary across implementations.

All methods are executed atomically.

class threading.Lock

The class implementing primitive lock objects. Once a thread has acquired a lock, subsequent attempts to acquire it block, until it is released; any thread may release it.

Changed in version 3.13: `Lock` is now a class. In earlier Pythons, `Lock` was a factory function which returned an instance of the underlying private lock type.

acquire(*blocking=True*, *timeout=-1*)

Acquire a lock, blocking or non-blocking.

When invoked with the *blocking* argument set to `True` (the default), block until the lock is unlocked, then set it to locked and return `True`.

When invoked with the *blocking* argument set to `False`, do not block. If a call with *blocking* set to `True` would block, return `False` immediately; otherwise, set the lock to locked and return `True`.

When invoked with the floating-point *timeout* argument set to a positive value, block for at most the number of seconds specified by *timeout* and as long as the lock cannot be acquired. A *timeout* argument of `-1` specifies an unbounded wait. It is forbidden to specify a *timeout* when *blocking* is `False`.

The return value is `True` if the lock is acquired successfully, `False` if not (for example if the *timeout* expired).

Changed in version 3.2: The *timeout* parameter is new.

Changed in version 3.2: Lock acquisition can now be interrupted by signals on POSIX if the underlying threading implementation supports it.

Changed in version 3.14: Lock acquisition can now be interrupted by signals on Windows.

release()

Release a lock. This can be called from any thread, not only the thread which has acquired the lock.

When the lock is locked, reset it to unlocked, and return. If any other threads are blocked waiting for the lock to become unlocked, allow exactly one of them to proceed.

When invoked on an unlocked lock, a `RuntimeError` is raised.

There is no return value.

locked()

Return `True` if the lock is acquired.

RLock objects

A reentrant lock is a synchronization primitive that may be acquired multiple times by the same thread. Internally, it uses the concepts of “owning thread” and “recursion level” in addition to the locked/unlocked state used by primitive locks. In the locked state, some thread owns the lock; in the unlocked state, no thread owns it.

Threads call a lock’s `acquire()` method to lock it, and its `release()` method to unlock it.

Note

Reentrant locks support the context management protocol, so it is recommended to use `with` instead of manually calling `acquire()` and `release()` to handle acquiring and releasing the lock for a block of code.

RLock’s `acquire()`/`release()` call pairs may be nested, unlike Lock’s `acquire()`/`release()`. Only the final `release()` (the `release()` of the outermost pair) resets the lock to an unlocked state and allows another thread blocked in `acquire()` to proceed.

`acquire()`/`release()` must be used in pairs: each acquire must have a release in the thread that has acquired the lock. Failing to call release as many times the lock has been acquired can lead to deadlock.

class threading.RLock

This class implements reentrant lock objects. A reentrant lock must be released by the thread that acquired it. Once a thread has acquired a reentrant lock, the same thread may acquire it again without blocking; the thread must release it once for each time it has acquired it.

Note that `RLock` is actually a factory function which returns an instance of the most efficient version of the concrete RLock class that is supported by the platform.

acquire(*blocking=True*, *timeout=-1*)

Acquire a lock, blocking or non-blocking.

See also

Using RLock as a context manager

Recommended over manual **`acquire()`** and `release()` calls whenever practical.

When invoked with the *blocking* argument set to `True` (the default):

- If no thread owns the lock, acquire the lock and return immediately.
- If another thread owns the lock, block until we are able to acquire lock, or *timeout*, if set to a positive float value.
- If the same thread owns the lock, acquire the lock again, and return immediately. This is the difference between `Lock` and **`RLock`**; `Lock`

handles this case the same as the previous, blocking until the lock can be acquired.

When invoked with the *blocking* argument set to `False`:

- If no thread owns the lock, acquire the lock and return immediately.
- If another thread owns the lock, return immediately.
- If the same thread owns the lock, acquire the lock again and return immediately.

In all cases, if the thread was able to acquire the lock, return `True`. If the thread was unable to acquire the lock (i.e. if not blocking or the timeout was reached) return `False`.

If called multiple times, failing to call **`release()`** as many times may lead to deadlock. Consider using **`RLock`** as a context manager rather than calling acquire/release directly.

Changed in version 3.2: The *timeout* parameter is new.

release()

Release a lock, decrementing the recursion level. If after the decrement it is zero, reset the lock to unlocked (not owned by any thread), and if any other threads are blocked waiting for the lock to become unlocked, allow exactly one of them to proceed. If after the decrement the recursion level is still nonzero, the lock remains locked and owned by the calling thread.

Only call this method when the calling thread owns the lock. A **`RuntimeError`** is raised if this method is called when the lock is not acquired.

There is no return value.

locked()

Return a boolean indicating whether this object is locked right now.

Added in version 3.14.

Condition objects

A condition variable is always associated with some kind of lock; this can be passed in or one will be created by default. Passing one in is useful when several condition variables must share the same lock. The lock is part of the condition object: you don't have to track it separately.

A condition variable obeys the context management protocol: using the `with` statement acquires the associated lock for the duration of the enclosed block. The **`acquire()`** and **`release()`** methods also call the corresponding methods of the associated lock.

Other methods must be called with the associated lock held. The **`wait()`** method releases the lock, and then blocks until another thread awakens it by calling **`notify()`** or **`notify_all()`**. Once awakened, **`wait()`** re-acquires the lock and returns. It is also possible to specify a timeout.

The **`notify()`** method wakes up one of the threads waiting for the condition variable, if any are waiting. The **`notify_all()`** method wakes up all threads waiting for the condition variable.

Note: the **`notify()`** and **`notify_all()`** methods don't release the lock; this means that the thread or threads awakened will not return from their `wait()` call immediately, but only when the thread that called `notify()` or `notify_all()` finally relinquishes ownership of the lock.

The typical programming style using condition variables uses the lock to synchronize access to some shared state; threads that are interested in a particular change of state call `wait()` repeatedly until they see the desired state, while threads that modify the state call `notify()` or `notify_all()` when they change the state in such a way that it could possibly be a desired state for one of the waiters. For example, the following code is a generic producer-consumer situation with unlimited buffer capacity:

```
# Consume one item
with cv:
    while not an_item_is_available():
        cv.wait()
    get_an_available_item()

# Produce one item
with cv:
    make_an_item_available()
    cv.notify()
```

The `while` loop checking for the application’s condition is necessary because `wait()` can return after an arbitrary long time, and the condition which prompted the `notify()` call may no longer hold true. This is inherent to multi-threaded programming. The `wait_for()` method can be used to automate the condition checking, and eases the computation of timeouts:

```
# Consume an item
with cv:
    cv.wait_for(an_item_is_available)
    get_an_available_item()
```

To choose between `notify()` and `notify_all()`, consider whether one state change can be interesting for only one or several waiting threads. E.g. in a typical producer-consumer situation, adding one item to the buffer only needs to wake up one consumer thread.

class threading.Condition(*lock=None*)

This class implements condition variable objects. A condition variable allows one or more threads to wait until they are notified by another thread.

If the *lock* argument is given and not `None`, it must be a `Lock` or `RLock` object, and it is used as the underlying lock. Otherwise, a new `RLock` object is created and used as the underlying lock.

Changed in version 3.3: changed from a factory function to a class.

acquire(**args*)

Acquire the underlying lock. This method calls the corresponding method on the underlying lock; the return value is whatever that method returns.

release()

Release the underlying lock. This method calls the corresponding method on the underlying lock; there is no return value.

locked()

Return a boolean indicating whether this object is locked right now.

Added in version 3.14.

wait(*timeout=None*)

Wait until notified or until a timeout occurs. If the calling thread has not acquired the lock when this method is called, a `RuntimeError` is raised.

This method releases the underlying lock, and then blocks until it is awakened by a `notify()` or `notify_all()` call for the same condition variable in another thread, or until the optional timeout occurs. Once awakened or timed out, it re-acquires the lock and returns.

When the *timeout* argument is present and not `None`, it should be a floating-point number specifying a timeout for the operation in seconds (or fractions thereof).

When the underlying lock is an `RLock`, it is not released using its `release()` method, since this may not actually unlock the lock when it was acquired multiple times recursively. Instead, an internal interface of the `RLock` class is used, which really unlocks it even when it has been recursively acquired several times. Another internal interface is then used to restore the recursion level when the lock is reacquired.

The return value is `True` unless a given *timeout* expired, in which case it is `False`.

Changed in version 3.2: Previously, the method always returned `None`.

wait_for(*predicate*, *timeout=None*)

Wait until a condition evaluates to true. *predicate* should be a callable which result will be interpreted as a boolean value. A *timeout* may be provided giving the maximum time to wait.

This utility method may call `wait()` repeatedly until the predicate is satisfied, or until a timeout occurs. The return value is the last return value of the predicate and will evaluate to `False` if the method timed out.

Ignoring the timeout feature, calling this method is roughly equivalent to writing:

```
while not predicate():
    cv.wait()
```

Therefore, the same rules apply as with `wait()`: The lock must be held when called and is re-acquired on return. The predicate is evaluated with the lock held.

Added in version 3.2.

notify(*n=1*)

By default, wake up one thread waiting on this condition, if any. If the calling thread has not acquired the lock when this method is called, a `RuntimeError` is raised.

This method wakes up at most *n* of the threads waiting for the condition variable; it is a no-op if no threads are waiting.

The current implementation wakes up exactly *n* threads, if at least *n* threads are waiting. However, it’s not safe to rely on this behavior. A future, optimized implementation may occasionally wake up more than *n* threads.

Note: an awakened thread does not actually return from its `wait()` call until it can reacquire the lock. Since `notify()` does not release the lock, its caller should.

notify_all()

Wake up all threads waiting on this condition. This method acts like `notify()`, but wakes up all waiting threads instead of one. If the calling thread has not acquired the lock when this method is called, a `RuntimeError` is raised.

The method `notifyAll` is a deprecated alias for this method.

Semaphore objects

This is one of the oldest synchronization primitives in the history of computer science, invented by the early Dutch computer scientist Edsger W. Dijkstra (he used the names `P()` and `V()` instead of `acquire()` and `release()`).

A semaphore manages an internal counter which is decremented by each `acquire()` call and incremented by each `release()` call. The counter can never go below zero; when `acquire()` finds that it is zero, it blocks, waiting until some other thread calls `release()`.

Semaphores also support the context management protocol.

class threading.Semaphore(*value=1*)

This class implements semaphore objects. A semaphore manages an atomic counter representing the number of `release()` calls minus the number of `acquire()` calls, plus an initial value. The `acquire()` method blocks if necessary until it can return without making the counter negative. If not given, *value* defaults to 1.

The optional argument gives the initial *value* for the internal counter; it defaults to `1`. If the *value* given is less than 0, `ValueError` is raised.

Changed in version 3.3: changed from a factory function to a class.

acquire(*blocking=True*, *timeout=None*)

Acquire a semaphore.

When invoked without arguments:

- If the internal counter is larger than zero on entry, decrement it by one and return `True` immediately.
- If the internal counter is zero on entry, block until awoken by a call to `release()`. Once awoken (and the counter is greater than 0), decrement the counter by 1 and return `True`. Exactly one thread will be awoken by each call to `release()`. The order in which threads are awoken should not be relied on.

When invoked with *blocking* set to `False`, do not block. If a call without an argument would block, return `False` immediately; otherwise, do the same thing as when called without arguments, and return `True`.

When invoked with a *timeout* other than `None`, it will block for at most *timeout* seconds. If acquire does not complete successfully in that interval, return `False`. Return `True` otherwise.

Changed in version 3.2: The *timeout* parameter is new.

release(*n=1*)

Release a semaphore, incrementing the internal counter by *n*. When it was zero on entry and other threads are waiting for it to become larger than zero again, wake up *n* of those threads.

Changed in version 3.9: Added the *n* parameter to release multiple waiting threads at once.

class threading.BoundedSemaphore(*value=1*)

Class implementing bounded semaphore objects. A bounded semaphore checks to make sure its current value doesn't exceed its initial value. If it does, `ValueError` is raised. In most situations semaphores are used to guard resources with limited capacity. If the semaphore is released too many times it's a sign of a bug. If not given, *value* defaults to 1.

Changed in version 3.3: changed from a factory function to a class.

`Semaphore` example

Semaphores are often used to guard resources with limited capacity, for example, a database server. In any situation where the size of the resource is fixed, you should use a bounded semaphore. Before spawning any worker threads, your main thread would initialize the semaphore:

```
maxconnections = 5
# ...
pool_sema = BoundedSemaphore(value=maxconnections)
```

Once spawned, worker threads call the semaphore's acquire and release methods when they need to connect to the server:

```
with pool_sema:
    conn = connectdb()
    try:
        # ... use connection ...
    finally:
        conn.close()
```

The use of a bounded semaphore reduces the chance that a programming error which causes the semaphore to be released more than it’s acquired will go undetected.

Event objects

This is one of the simplest mechanisms for communication between threads: one thread signals an event and other threads wait for it.

An event object manages an internal flag that can be set to true with the `set()` method and reset to false with the `clear()` method. The `wait()` method blocks until the flag is true.

class threading.Event

Class implementing event objects. An event manages a flag that can be set to true with the `set()` method and reset to false with the `clear()` method. The `wait()` method blocks until the flag is true. The flag is initially false.

Changed in version 3.3: changed from a factory function to a class.

is_set()

Return `True` if and only if the internal flag is true.

The method `isSet` is a deprecated alias for this method.

set()

Set the internal flag to true. All threads waiting for it to become true are awakened. Threads that call `wait()` once the flag is true will not block at all.

clear()

Reset the internal flag to false. Subsequently, threads calling `wait()` will block until `set()` is called to set the internal flag to true again.

wait(*timeout=None*)

Block as long as the internal flag is false and the timeout, if given, has not expired. The return value represents the reason that this blocking method returned; `True` if returning because the internal flag is set to true, or `False` if a timeout is given and the internal flag did not become true within the given wait time.

When the timeout argument is present and not `None`, it should be a floating-point number specifying a timeout for the operation in seconds, or fractions thereof.

Changed in version 3.1: Previously, the method always returned `None`.

Timer objects

This class represents an action that should be run only after a certain amount of time has passed — a timer. `Timer` is a subclass of `Thread` and as such also functions as an example of creating custom threads.

Timers are started, as with threads, by calling their `Timer.start` method. The timer can be stopped (before its action has begun) by calling the `cancel()` method. The interval the timer will wait before executing its action may not be exactly the same as the interval specified by the user.

For example:

```
def hello():
    print("hello, world")

t = Timer(30.0, hello)
t.start()  # after 30 seconds, "hello, world" will be printed
```

class threading.Timer(*interval*, *function*, *args=None*, *kwargs=None*)

Create a timer that will run *function* with arguments *args* and keyword arguments *kwargs*, after *interval* seconds have passed. If *args* is `None` (the default) then an empty list will be used. If *kwargs* is `None` (the default) then an empty dict will be used.

Changed in version 3.3: changed from a factory function to a class.

cancel()

Stop the timer, and cancel the execution of the timer’s action. This will only work if the timer is still in its waiting stage.

Barrier objects

Added in version 3.2.

This class provides a simple synchronization primitive for use by a fixed number of threads that need to wait for each other. Each of the threads tries to pass the barrier by calling the `wait()` method and will block until all of the threads have made their `wait()` calls. At this point, the threads are released simultaneously.

The barrier can be reused any number of times for the same number of threads.

As an example, here is a simple way to synchronize a client and server thread:

```
b = Barrier(2, timeout=5)

def server():
    start_server()
    b.wait()
    while True:
        connection = accept_connection()
        process_server_connection(connection)

def client():
    b.wait()
    while True:
        connection = make_connection()
        process_client_connection(connection)
```

class threading.Barrier(*parties*, *action=None*, *timeout=None*)

Create a barrier object for *parties* number of threads. An *action*, when provided, is a callable to be called by one of the threads when they are released. *timeout* is the default timeout value if none is specified for the `wait()` method.

wait(*timeout=None*)

Pass the barrier. When all the threads party to the barrier have called this function, they are all released simultaneously. If a *timeout* is provided, it is used in preference to any that was supplied to the class constructor.

The return value is an integer in the range 0 to *parties* – 1, different for each thread. This can be used to select a thread to do some special housekeeping, e.g.:

```
i = barrier.wait()
if i == 0:
    # Only one thread needs to print this
    print("passed the barrier")
```

If an *action* was provided to the constructor, one of the threads will have called it prior to being released. Should this call raise an error, the barrier is put into the broken state.

If the call times out, the barrier is put into the broken state.

This method may raise a `BrokenBarrierError` exception if the barrier is broken or reset while a thread is waiting.

reset()

Return the barrier to the default, empty state. Any threads waiting on it will receive the `BrokenBarrierError` exception.

Note that using this function may require some external synchronization if there are other threads whose state is unknown. If a barrier is broken it may be better to just leave it and create a new one.

abort()

Put the barrier into a broken state. This causes any active or future calls to `wait()` to fail with the `BrokenBarrierError`. Use this for example if one of the threads needs to abort, to avoid deadlocking the application.

It may be preferable to simply create the barrier with a sensible *timeout* value to automatically guard against one of the threads going awry.

parties

The number of threads required to pass the barrier.

n_waiting

The number of threads currently waiting in the barrier.

broken

A boolean that is `True` if the barrier is in the broken state.

exception threading.BrokenBarrierError

This exception, a subclass of **`RuntimeError`**, is raised when the **`Barrier`** object is reset or broken.

Using locks, conditions, and semaphores in the `with` statement

All of the objects provided by this module that have `acquire` and `release` methods can be used as context managers for a **`with`** statement. The `acquire` method will be called when the block is entered, and `release` will be called when the block is exited. Hence, the following snippet:

```
with some_lock:
    # do something...
```

is equivalent to:

```
some_lock.acquire()
try:
    # do something...
finally:
    some_lock.release()
```

Currently, **`Lock`**, **`RLock`**, **`Condition`**, **`Semaphore`**, and **`BoundedSemaphore`** objects may be used as **`with`** statement context managers.

`multiprocessing` — Process-based parallelism

Source code: Lib/multiprocessing/ [https://github.com/python/cpython/tree/3.14/Lib/multiprocessing/]

Availability: not Android, not iOS, not WASI.

This module is not supported on mobile platforms or WebAssembly platforms.

Introduction

`multiprocessing` is a package that supports spawning processes using an API similar to the **`threading`** module. The **`multiprocessing`** package offers both local and remote concurrency, effectively side-stepping the Global Interpreter Lock by using subprocesses instead of threads. Due to this, the **`multiprocessing`** module allows the programmer to fully leverage multiple processors on a given machine. It runs on both POSIX and Windows.

The **`multiprocessing`** module also introduces the **`Pool`** object which offers a convenient means of parallelizing the execution of a function across multiple input values, distributing the input data across processes (data parallelism). The following example demonstrates the common practice of defining such functions in a module so that child processes can successfully import that module. This basic example of data parallelism using **`Pool`**,

```
from multiprocessing import Pool

def f(x):
    return x*x

if __name__ == '__main__':
    with Pool(5) as p:
        print(p.map(f, [1, 2, 3]))
```

will print to standard output

```
[1, 4, 9]
```

The `multiprocessing` module also introduces APIs which do not have analogs in the `threading` module, like the ability to `terminate`, `interrupt` or `kill` a running process.

See also

`concurrent.futures.ProcessPoolExecutor` offers a higher level interface to push tasks to a background process without blocking execution of the calling process. Compared to using the `Pool` interface directly, the `concurrent.futures` API more readily allows the submission of work to the underlying process pool to be separated from waiting for the results.

The `Process` class

In `multiprocessing`, processes are spawned by creating a `Process` object and then calling its `start()` method. `Process` follows the API of `threading.Thread`. A trivial example of a multiprocess program is

```
from multiprocessing import Process

def f(name):
    print('hello', name)

if __name__ == '__main__':
    p = Process(target=f, args=('bob',))
    p.start()
    p.join()
```

To show the individual process IDs involved, here is an expanded example:

```
from multiprocessing import Process
import os

def info(title):
    print(title)
    print('module name:', __name__)
    print('parent process:', os.getppid())
    print('process id:', os.getpid())

def f(name):
    info('function f')
    print('hello', name)

if __name__ == '__main__':
    info('main line')
    p = Process(target=f, args=('bob',))
    p.start()
    p.join()
```

For an explanation of why the `if __name__ == '__main__'` part is necessary, see Programming guidelines.

The arguments to **`Process`** usually need to be unpickleable from within the child process. If you tried typing the above example directly into a REPL it could lead to an **`AttributeError`** in the child process trying to locate the *f* function in the `__main__` module.

Contexts and start methods

Depending on the platform, `multiprocessing` supports three ways to start a process. These *start methods* are

spawn

The parent process starts a fresh Python interpreter process. The child process will only inherit those resources necessary to run the process object's `run()` method. In particular, unnecessary file descriptors and handles from the parent process will not be inherited. Starting a process using this method is rather slow compared to using *fork* or *forkserver*.

Available on POSIX and Windows platforms. The default on Windows and macOS.

fork

The parent process uses `os.fork()` to fork the Python interpreter. The child process, when it begins, is effectively identical to the parent process. All resources of the parent are inherited by the child process. Note that safely forking a multithreaded process is problematic.

Available on POSIX systems.

Changed in version 3.14: This is no longer the default start method on any platform. Code that requires *fork* must explicitly specify that via `get_context()` or `set_start_method()`.

Changed in version 3.12: If Python is able to detect that your process has multiple threads, the `os.fork()` function that this start method calls internally will raise a `DeprecationWarning`. Use a different start method. See the `os.fork()` documentation for further explanation.

forkserver

When the program starts and selects the *forkserver* start method, a server process is spawned. From then on, whenever a new process is needed, the parent process connects to the server and requests that it fork a new process. The fork server process is single threaded unless system libraries or preloaded imports spawn threads as a side-effect so it is generally safe for it to use `os.fork()`. No unnecessary resources are inherited.

Available on POSIX platforms which support passing file descriptors over Unix pipes such as Linux. The default on those.

Changed in version 3.14: This became the default start method on POSIX platforms.

Changed in version 3.4: spawn added on all POSIX platforms, and *forkserver* added for some POSIX platforms. Child processes no longer inherit all of the parents inheritable handles on Windows.

Changed in version 3.8: On macOS, the *spawn* start method is now the default. The *fork* start method should be considered unsafe as it can lead to crashes of the subprocess as macOS system libraries may start threads. See bpo-33725 [https://bugs.python.org/issue?@action=redirect&bpo=33725].

Changed in version 3.14: On POSIX platforms the default start method was changed from *fork* to *forkserver* to retain the performance but avoid common multithreaded process incompatibilities. See gh-84559 [https://github.com/python/cpython/issues/84559].

On POSIX using the *spawn* or *forkserver* start methods will also start a *resource tracker* process which tracks the unlinked named system resources (such as named semaphores or `SharedMemory` objects) created by processes of the program. When all processes have exited the resource tracker unlinks any remaining tracked object. Usually there should be none, but if a process was killed by a signal there may be some “leaked” resources. (Neither leaked semaphores nor shared memory segments will be automatically unlinked until the next reboot. This is problematic for both objects because the system allows only a limited number of named semaphores, and shared memory segments occupy some space in the main memory.)

To select a start method you use the `set_start_method()` in the `if __name__ == '__main__'` clause of the main module. For example:

```
import multiprocessing as mp

def foo(q):
    q.put('hello')

if __name__ == '__main__':
    mp.set_start_method('spawn')
    q = mp.Queue()
    p = mp.Process(target=foo, args=(q,))
    p.start()
    print(q.get())
    p.join()
```

`set_start_method()` should not be used more than once in the program.

Alternatively, you can use `get_context()` to obtain a context object. Context objects have the same API as the multiprocessing module, and allow one to use multiple start methods in the same program.

```
import multiprocessing as mp

def foo(q):
    q.put('hello')

if __name__ == '__main__':
    ctx = mp.get_context('spawn')
    q = ctx.Queue()
    p = ctx.Process(target=foo, args=(q,))
    p.start()
    print(q.get())
    p.join()
```

Note that objects related to one context may not be compatible with processes for a different context. In particular, locks created using the *fork* context cannot be passed to processes started using the *spawn* or *forkserver* start methods.

Libraries using `multiprocessing` or `ProcessPoolExecutor` should be designed to allow their users to provide their own multiprocessing context. Using a specific context of your own within a library can lead to incompatibilities with the rest of the library user’s application. Always document if your library requires a specific start method.

Warning

The `'spawn'` and `'forkserver'` start methods generally cannot be used with “frozen” executables (i.e., binaries produced by packages like **PyInstaller** and **cx_Freeze**) on POSIX systems. The `'fork'` start method may work if code does not use threads.

Exchanging objects between processes

`multiprocessing` supports two types of communication channel between processes:

Queues

The `Queue` class is a near clone of `queue.Queue`. For example:

```
from multiprocessing import Process, Queue

def f(q):
    q.put([42, None, 'hello'])

if __name__ == '__main__':
    q = Queue()
    p = Process(target=f, args=(q,))
    p.start()
    print(q.get())    # prints "[42, None, 'hello']"
    p.join()
```

Queues are thread and process safe. Any object put into a `multiprocessing` queue will be serialized.

Pipes

The `Pipe()` function returns a pair of connection objects connected by a pipe which by default is duplex (two-way). For example:

```
from multiprocessing import Process, Pipe

def f(conn):
    conn.send([42, None, 'hello'])
    conn.close()

if __name__ == '__main__':
    parent_conn, child_conn = Pipe()
    p = Process(target=f, args=(child_conn,))
    p.start()
    print(parent_conn.recv())   # prints "[42, None,
'hello']"
    p.join()
```

The two connection objects returned by **`Pipe()`** represent the two ends of the pipe. Each connection object has **`send()`** and **`recv()`** methods (among others). Note that data in a pipe may become corrupted if two processes (or threads) try to read from or write to the *same* end of the pipe at the same time. Of course there is no risk of corruption from processes using different ends of the pipe at the same time.

The **`send()`** method serializes the object and **`recv()`** re-creates the object.

Synchronization between processes

`multiprocessing` contains equivalents of all the synchronization primitives from **`threading`**. For instance one can use a lock to ensure that only one process prints to standard output at a time:

```
from multiprocessing import Process, Lock

def f(l, i):
    l.acquire()
    try:
        print('hello world', i)
    finally:
        l.release()

if __name__ == '__main__':
    lock = Lock()

    for num in range(10):
        Process(target=f, args=(lock, num)).start()
```

Without using the lock output from the different processes is liable to get all mixed up.

Sharing state between processes

As mentioned above, when doing concurrent programming it is usually best to avoid using shared state as far as possible. This is particularly true when using multiple processes.

However, if you really do need to use some shared data then `multiprocessing` provides a couple of ways of doing so.

Shared memory

Data can be stored in a shared memory map using `Value` or `Array`. For example, the following code

```
from multiprocessing import Process, Value, Array

def f(n, a):
    n.value = 3.1415927
    for i in range(len(a)):
        a[i] = -a[i]

if __name__ == '__main__':
    num = Value('d', 0.0)
    arr = Array('i', range(10))

    p = Process(target=f, args=(num, arr))
    p.start()
    p.join()

    print(num.value)
    print(arr[:])
```

will print

```
3.1415927
[0, -1, -2, -3, -4, -5, -6, -7, -8, -9]
```

The `'d'` and `'i'` arguments used when creating `num` and `arr` are typecodes of the kind used by the `array` module: `'d'` indicates a double precision float and `'i'` indicates a signed integer. These shared objects will be process and thread-safe.

For more flexibility in using shared memory one can use the `multiprocessing.sharedctypes` module which supports the creation of arbitrary ctypes objects allocated from shared memory.

Server process

A manager object returned by `Manager()` controls a server process which holds Python objects and allows other processes to manipulate them using proxies.

A manager returned by `Manager()` will support types `list`, `dict`, `set`, `Namespace`, `Lock`, `RLock`, `Semaphore`, `BoundedSemaphore`, `Condition`, `Event`, `Barrier`, `Queue`, `Value` and `Array`. For example,

```
from multiprocessing import Process, Manager

def f(d, l, s):
    d[1] = '1'
    d['2'] = 2
    d[0.25] = None
    l.reverse()
    s.add('a')
    s.add('b')

if __name__ == '__main__':
    with Manager() as manager:
        d = manager.dict()
        l = manager.list(range(10))
        s = manager.set()

        p = Process(target=f, args=(d, l, s))
        p.start()
        p.join()

        print(d)
        print(l)
        print(s)
```

will print

```
{0.25: None, 1: '1', '2': 2}
[9, 8, 7, 6, 5, 4, 3, 2, 1, 0]
{'a', 'b'}
```

Server process managers are more flexible than using shared memory objects because they can be made to support arbitrary object types. Also, a single manager can be shared by processes on different computers over a network. They are, however, slower than using shared memory.

Using a pool of workers

The **`Pool`** class represents a pool of worker processes. It has methods which allows tasks to be offloaded to the worker processes in a few different ways.

For example:

```
from multiprocessing import Pool, TimeoutError
import time
import os

def f(x):
    return x*x

if __name__ == '__main__':
    # start 4 worker processes
    with Pool(processes=4) as pool:

        # print "[0, 1, 4,..., 81]"
        print(pool.map(f, range(10)))

        # print same numbers in arbitrary order
        for i in pool.imap_unordered(f, range(10)):
            print(i)

        # evaluate "f(20)" asynchronously
        res = pool.apply_async(f, (20,))      # runs in *only*
one process
        print(res.get(timeout=1))             # prints "400"

        # evaluate "os.getpid()" asynchronously
        res = pool.apply_async(os.getpid, ()) # runs in *only*
one process
        print(res.get(timeout=1))             # prints the PID
of that process

        # launching multiple evaluations asynchronously *may*
use more processes
        multiple_results = [pool.apply_async(os.getpid, ()) for
i in range(4)]
        print([res.get(timeout=1) for res in multiple_results])

        # make a single worker sleep for 10 seconds
        res = pool.apply_async(time.sleep, (10,))
        try:
            print(res.get(timeout=1))
        except TimeoutError:
            print("We lacked patience and got a
multiprocessing.TimeoutError")

        print("For the moment, the pool remains available for
more work")

    # exiting the 'with'-block has stopped the pool
    print("Now the pool is closed and no longer available")
```

Note that the methods of a pool should only ever be used by the process which created it.

Note

Functionality within this package requires that the `__main__` module be importable by the children. This is covered in Programming guidelines however it is worth pointing out here. This means that some examples, such as the `multiprocessing.pool.Pool` examples will not work in the interactive interpreter. For example:

```
>>> from multiprocessing import Pool
>>> p = Pool(5)
>>> def f(x):
...     return x*x
...
>>> with p:
...     p.map(f, [1,2,3])
Process PoolWorker-1:
Process PoolWorker-2:
Process PoolWorker-3:
Traceback (most recent call last):
Traceback (most recent call last):
Traceback (most recent call last):
AttributeError: Can't get attribute 'f' on <module '__main__'
(<class '_frozen_importlib.BuiltinImporter'>)>
AttributeError: Can't get attribute 'f' on <module '__main__'
(<class '_frozen_importlib.BuiltinImporter'>)>
AttributeError: Can't get attribute 'f' on <module '__main__'
(<class '_frozen_importlib.BuiltinImporter'>)>
```

(If you try this it will actually output three full tracebacks interleaved in a semi-random fashion, and then you may have to stop the parent process somehow.)

Reference

The `multiprocessing` package mostly replicates the API of the `threading` module.

Global start method

Python supports several ways to create and initialize a process. The global start method sets the default mechanism for creating a process.

Several multiprocessing functions and methods that may also instantiate certain objects will implicitly set the global start method to the system's default, if it hasn't been set already. The global start method can only be set once. If you need to change the start method from the system default, you must proactively set the global start method before calling functions or methods, or creating these objects.

`Process` and exceptions

class multiprocessing.Process(*group=None*, *target=None*, *name=None*, *args=()*, *kwargs={}*, *, *daemon=None*)

Process objects represent activity that is run in a separate process. The `Process` class has equivalents of all the methods of `threading.Thread`.

The constructor should always be called with keyword arguments. *group* should always be `None`; it exists solely for compatibility with `threading.Thread`. *target* is the callable object to be invoked by the `run()` method. It defaults to `None`, meaning nothing is called. *name* is the process name (see `name` for more details). *args* is the argument tuple for the target invocation. *kwargs* is a dictionary of keyword arguments for the target invocation. If provided, the keyword-only *daemon* argument sets the process `daemon` flag to `True` or `False`. If `None` (the default), this flag will be inherited from the creating process.

By default, no arguments are passed to *target*. The *args* argument, which defaults to `()`, can be used to specify a list or tuple of the arguments to pass to *target*.

If a subclass overrides the constructor, it must make sure it invokes the base class constructor (`super().__init__()`) before doing anything else to the process.

Note

In general, all arguments to **`Process`** must be picklable. This is frequently observed when trying to create a **`Process`** or use a **`concurrent.futures.ProcessPoolExecutor`** from a REPL with a locally defined *target* function.

Passing a callable object defined in the current REPL session causes the child process to die via an uncaught **`AttributeError`** exception when starting as *target* must have been defined within an importable module in order to be loaded during unpickling.

Example of this uncatchable error from the child:

```
>>> import multiprocessing as mp
>>> def knigit():
...     print("Ni!")
...
>>> process = mp.Process(target=knigit)
>>> process.start()
>>> Traceback (most recent call last):
  File ".../multiprocessing/spawn.py", line ..., in
spawn_main
  File ".../multiprocessing/spawn.py", line ..., in _main
AttributeError: module '__main__' has no attribute
'knigit'
>>> process
<SpawnProcess name='SpawnProcess-1' pid=379473
parent=378707 stopped exitcode=1>
```

See The spawn and forkserver start methods. While this restriction is not true if using the `"fork"` start method, as of Python `3.14` that is no longer the default on any platform. See Contexts and start methods. See also gh-132898 [https://github.com/python/cpython/issues/132898].

Changed in version 3.3: Added the *daemon* parameter.

run()

Method representing the process’s activity.

You may override this method in a subclass. The standard **`run()`** method invokes the callable object passed to the object’s constructor as the target argument, if any, with sequential and keyword arguments taken from the *args* and *kwargs* arguments, respectively.

Using a list or tuple as the *args* argument passed to **`Process`** achieves the same effect.

Example:

```
>>> from multiprocessing import Process
>>> p = Process(target=print, args=[1])
>>> p.run()
1
>>> p = Process(target=print, args=(1,))
>>> p.run()
1
```

start()

Start the process’s activity.

This must be called at most once per process object. It arranges for the object’s **`run()`** method to be invoked in a separate process.

join([*timeout*])

If the optional argument *timeout* is `None` (the default), the method blocks until the process whose `join()` method is called terminates. If *timeout* is a positive number, it blocks at most *timeout* seconds. Note that the method returns `None` if its process terminates or if the method times out. Check the process's `exitcode` to determine if it terminated.

A process can be joined many times.

A process cannot join itself because this would cause a deadlock. It is an error to attempt to join a process before it has been started.

name

The process's name. The name is a string used for identification purposes only. It has no semantics. Multiple processes may be given the same name.

The initial name is set by the constructor. If no explicit name is provided to the constructor, a name of the form 'Process-N_1:N_2:...:N_k' is constructed, where each N_k is the N-th child of its parent.

is_alive()

Return whether the process is alive.

Roughly, a process object is alive from the moment the `start()` method returns until the child process terminates.

daemon

The process's daemon flag, a Boolean value. This must be set before `start()` is called.

The initial value is inherited from the creating process.

When a process exits, it attempts to terminate all of its daemonic child processes.

Note that a daemonic process is not allowed to create child processes. Otherwise a daemonic process would leave its children orphaned if it gets terminated when its parent process exits. Additionally, these are **not** Unix daemons or services, they are normal processes that will be terminated (and not joined) if non-daemonic processes have exited.

In addition to the `threading.Thread` API, `Process` objects also support the following attributes and methods:

pid

Return the process ID. Before the process is spawned, this will be `None`.

exitcode

The child’s exit code. This will be `None` if the process has not yet terminated.

If the child’s `run()` method returned normally, the exit code will be 0. If it terminated via `sys.exit()` with an integer argument *N*, the exit code will be *N*.

If the child terminated due to an exception not caught within `run()`, the exit code will be 1. If it was terminated by signal *N*, the exit code will be the negative value -*N*.

authkey

The process’s authentication key (a byte string).

When `multiprocessing` is initialized the main process is assigned a random string using `os.urandom()`.

When a `Process` object is created, it will inherit the authentication key of its parent process, although this may be changed by setting `authkey` to another byte string.

See Authentication keys.

sentinel

A numeric handle of a system object which will become “ready” when the process ends.

You can use this value if you want to wait on several events at once using `multiprocessing.connection.wait()`. Otherwise calling `join()` is simpler.

On Windows, this is an OS handle usable with the `WaitForSingleObject` and `WaitForMultipleObjects` family of API calls. On POSIX, this is a file descriptor usable with primitives from the `select` module.

Added in version 3.3.

interrupt()

Terminate the process. Works on POSIX using the `SIGINT` signal. Behavior on Windows is undefined.

By default, this terminates the child process by raising `KeyboardInterrupt`. This behavior can be altered by setting the respective signal handler in the child process `signal.signal()` for `SIGINT`.

Note: if the child process catches and discards `KeyboardInterrupt`, the process will not be terminated.

Note: the default behavior will also set `exitcode` to `1` as if an uncaught exception was raised in the child process. To have a different `exitcode` you may simply catch `KeyboardInterrupt` and call `exit(your_code)`.

Added in version 3.14.

terminate()

Terminate the process. On POSIX this is done using the `SIGTERM` signal; on Windows **`TerminateProcess()`** is used. Note that exit handlers and finally clauses, etc., will not be executed.

Note that descendant processes of the process will *not* be terminated – they will simply become orphaned.

Warning

If this method is used when the associated process is using a pipe or queue then the pipe or queue is liable to become corrupted and may become unusable by other process. Similarly, if the process has acquired a lock or semaphore etc. then terminating it is liable to cause other processes to deadlock.

kill()

Same as `terminate()` but using the `SIGKILL` signal on POSIX.

Added in version 3.7.

close()

Close the `Process` object, releasing all resources associated with it. `ValueError` is raised if the underlying process is still running. Once `close()` returns successfully, most other methods and attributes of the `Process` object will raise `ValueError`.

Added in version 3.7.

Note that the `start()`, `join()`, `is_alive()`, `terminate()` and `exitcode` methods should only be called by the process that created the process object.

Example usage of some of the methods of `Process`:

```
>>> import multiprocessing, time, signal
>>> mp_context = multiprocessing.get_context('spawn')
>>> p = mp_context.Process(target=time.sleep, args=(1000,))
>>> print(p, p.is_alive())
<...Process ... initial> False
>>> p.start()
>>> print(p, p.is_alive())
<...Process ... started> True
>>> p.terminate()
>>> time.sleep(0.1)
>>> print(p, p.is_alive())
<...Process ... stopped exitcode=-SIGTERM> False
>>> p.exitcode == -signal.SIGTERM
True
```

exception multiprocessing.ProcessError

The base class of all `multiprocessing` exceptions.

exception multiprocessing.BufferTooShort

Exception raised by **`Connection.recv_bytes_into()`** when the supplied buffer object is too small for the message read.

If `e` is an instance of `BufferTooShort` then `e.args[0]` will give the message as a byte string.

exception multiprocessing.AuthenticationError

Raised when there is an authentication error.

exception multiprocessing.TimeoutError

Raised by methods with a timeout when the timeout expires.

Pipes and Queues

When using multiple processes, one generally uses message passing for communication between processes and avoids having to use any synchronization primitives like locks.

For passing messages one can use `Pipe()` (for a connection between two processes) or a queue (which allows multiple producers and consumers).

The `Queue`, `SimpleQueue` and `JoinableQueue` types are multi-producer, multi-consumer FIFO queues modelled on the `queue.Queue` class in the standard library. They differ in that `Queue` lacks the `task_done()` and `join()` methods introduced into Python 2.5's `queue.Queue` class.

If you use `JoinableQueue` then you **must** call `JoinableQueue.task_done()` for each task removed from the queue or else the semaphore used to count the number of unfinished tasks may eventually overflow, raising an exception.

One difference from other Python queue implementations, is that `multiprocessing` queues serializes all objects that are put into them using `pickle`. The object returned by the get method is a re-created object that does not share memory with the original object.

Note that one can also create a shared queue by using a manager object – see Managers.

Note

`multiprocessing` uses the usual `queue.Empty` and `queue.Full` exceptions to signal a timeout. They are not available in the `multiprocessing` namespace so you need to import them from `queue`.

Note

When an object is put on a queue, the object is pickled and a background thread later flushes the pickled data to an underlying pipe. This has some consequences which are a little surprising, but should not cause any practical difficulties – if they really bother you then you can instead use a queue created with a manager.

1. After putting an object on an empty queue there may be an infinitesimal delay before the queue’s `empty()` method returns `False` and `get_nowait()` can return without raising `queue.Empty`.
2. If multiple processes are enqueuing objects, it is possible for the objects to be received at the other end out-of-order. However, objects enqueued by the same process will always be in the expected order with respect to each other.

Warning

If a process is killed using `Process.terminate()` or `os.kill()` while it is trying to use a `Queue`, then the data in the queue is likely to become corrupted. This may cause any other process to get an exception when it tries to use the queue later on.

Warning

As mentioned above, if a child process has put items on a queue (and it has not used

`JoinableQueue.cancel_join_thread`), then that process will not terminate until all buffered items have been flushed to the pipe.

This means that if you try joining that process you may get a deadlock unless you are sure that all items which have been put on the queue have been consumed. Similarly, if the child process is non-daemonic then the parent process may hang on exit when it tries to join all its non-daemonic children.

Note that a queue created using a manager does not have this issue. See Programming guidelines.

For an example of the usage of queues for interprocess communication see Examples.

multiprocessing.Pipe(*duplex=True*)

Returns a pair `(conn1, conn2)` of **`Connection`** objects representing the ends of a pipe.

If *duplex* is `True` (the default) then the pipe is bidirectional. If *duplex* is `False` then the pipe is unidirectional: `conn1` can only be used for receiving messages and `conn2` can only be used for sending messages.

The **`send()`** method serializes the object using **`pickle`** and the **`recv()`** re-creates the object.

class multiprocessing.Queue([*maxsize*])

Returns a process shared queue implemented using a pipe and a few locks/semaphores. When a process first puts an item on the queue a feeder thread is started which transfers objects from a buffer into the pipe.

Instantiating this class may set the global start method. See Global start method for more details.

The usual `queue.Empty` and `queue.Full` exceptions from the standard library's `queue` module are raised to signal timeouts.

`Queue` implements all the methods of `queue.Queue` except for `task_done()` and `join()`.

qsize()

> Return the approximate size of the queue. Because of multithreading/multiprocessing semantics, this number is not reliable.
>
> Note that this may raise `NotImplementedError` on platforms like macOS where `sem_getvalue()` is not implemented.

empty()

> Return `True` if the queue is empty, `False` otherwise. Because of multithreading/multiprocessing semantics, this is not reliable.
>
> May raise an `OSError` on closed queues. (not guaranteed)

full()

> Return `True` if the queue is full, `False` otherwise. Because of multithreading/multiprocessing semantics, this is not reliable.

put(*obj*[, *block*[, *timeout*]])

> Put obj into the queue. If the optional argument *block* is `True` (the default) and *timeout* is `None` (the default), block if necessary until a free slot is available. If

timeout is a positive number, it blocks at most *timeout* seconds and raises the `queue.Full` exception if no free slot was available within that time. Otherwise (*block* is `False`), put an item on the queue if a free slot is immediately available, else raise the `queue.Full` exception (*timeout* is ignored in that case).

Changed in version 3.8: If the queue is closed, `ValueError` is raised instead of `AssertionError`.

put_nowait(*obj*)

Equivalent to `put(obj, False)`.

get([*block*[, *timeout*]])

Remove and return an item from the queue. If optional args *block* is `True` (the default) and *timeout* is `None` (the default), block if necessary until an item is available. If *timeout* is a positive number, it blocks at most *timeout* seconds and raises the `queue.Empty` exception if no item was available within that time. Otherwise (block is `False`), return an item if one is immediately available, else raise the `queue.Empty` exception (*timeout* is ignored in that case).

Changed in version 3.8: If the queue is closed, `ValueError` is raised instead of `OSError`.

get_nowait()

Equivalent to `get(False)`.

`multiprocessing.Queue` has a few additional methods not found in `queue.Queue`. These methods are usually unnecessary for most code:

close()

Close the queue: release internal resources.

A queue must not be used anymore after it is closed. For example, `get()`, `put()` and `empty()` methods must no longer be called.

The background thread will quit once it has flushed all buffered data to the pipe. This is called automatically when the queue is garbage collected.

join_thread()

Join the background thread. This can only be used after `close()` has been called. It blocks until the background thread exits, ensuring that all data in the buffer has been flushed to the pipe.

By default if a process is not the creator of the queue then on exit it will attempt to join the queue’s background thread. The process can call `cancel_join_thread()` to make `join_thread()` do nothing.

cancel_join_thread()

Prevent `join_thread()` from blocking. In particular, this prevents the background thread from being joined automatically when the process exits – see `join_thread()`.

A better name for this method might be `allow_exit_without_flush()`. It is likely to cause enqueued data to be lost, and you almost certainly will not need to use it. It is really only there if you need the current process to exit immediately without waiting to flush enqueued data to the underlying pipe, and you don’t care about lost data.

Note

This class’s functionality requires a functioning shared semaphore implementation on the host operating system. Without one, the functionality in this class will be disabled, and attempts to instantiate a `Queue` will result in an `ImportError`. See bpo-3770 [https://bugs.python.org/issue?@action=redirect&bpo=3770] for additional information. The same holds true for any of the specialized queue types listed below.

class multiprocessing.SimpleQueue

It is a simplified `Queue` type, very close to a locked `Pipe`.

Instantiating this class may set the global start method. See Global start method for more details.

close()

Close the queue: release internal resources.

A queue must not be used anymore after it is closed. For example, `get()`, `put()` and `empty()` methods must no longer be called.

Added in version 3.9.

empty()

Return `True` if the queue is empty, `False` otherwise.

Always raises an `OSError` if the SimpleQueue is closed.

get()

Remove and return an item from the queue.

put(*item*)

Put *item* into the queue.

class multiprocessing.JoinableQueue([*maxsize*])

`JoinableQueue`, a **`Queue`** subclass, is a queue which additionally has **`task_done()`** and **`join()`** methods.

Instantiating this class may set the global start method. See Global start method for more details.

task_done()

Indicate that a formerly enqueued task is complete. Used by queue consumers. For each **`get()`** used to fetch a task, a subsequent call to **`task_done()`** tells the queue that the processing on the task is complete.

If a **`join()`** is currently blocking, it will resume when all items have been processed (meaning that a **`task_done()`** call was received for every item that had been **`put()`** into the queue).

Raises a **`ValueError`** if called more times than there were items placed in the queue.

join()

Block until all items in the queue have been gotten and processed.

The count of unfinished tasks goes up whenever an item is added to the queue. The count goes down whenever a consumer calls **`task_done()`** to indicate that the item was retrieved and all work on it is complete. When the count of unfinished tasks drops to zero, **`join()`** unblocks.

Miscellaneous

multiprocessing.active_children()

Return list of all live children of the current process.

Calling this has the side effect of “joining” any processes which have already finished.

multiprocessing.cpu_count()

Return the number of CPUs in the system.

This number is not equivalent to the number of CPUs the current process can use. The number of usable CPUs can be obtained with `os.process_cpu_count()` (or `len(os.sched_getaffinity(0))`).

When the number of CPUs cannot be determined a `NotImplementedError` is raised.

See also

`os.cpu_count()` `os.process_cpu_count()`

Changed in version 3.13: The return value can also be overridden using the `-X cpu_count` flag or `PYTHON_CPU_COUNT` as this is merely a wrapper around the `os` cpu count APIs.

multiprocessing.current_process()

Return the `Process` object corresponding to the current process.

An analogue of `threading.current_thread()`.

multiprocessing.parent_process()

Return the `Process` object corresponding to the parent process of the `current_process()`. For the main process, `parent_process` will be `None`.

Added in version 3.8.

multiprocessing.freeze_support()

Add support for when a program which uses `multiprocessing` has been frozen to produce an executable. (Has been tested with **py2exe**, **PyInstaller** and **cx_Freeze**.)

One needs to call this function straight after the `if __name__ == '__main__'` line of the main module. For example:

```
from multiprocessing import Process, freeze_support

def f():
    print('hello world!')

if __name__ == '__main__':
    freeze_support()
    Process(target=f).start()
```

If the `freeze_support()` line is omitted then trying to run the frozen executable will raise `RuntimeError`.

Calling `freeze_support()` has no effect when the start method is not *spawn*. In addition, if the module is being run normally by the Python interpreter (the program has not been frozen), then `freeze_support()` has no effect.

multiprocessing.get_all_start_methods()

Returns a list of the supported start methods, the first of which is the default. The possible start methods are `'fork'`, `'spawn'` and `'forkserver'`. Not all platforms support all methods. See Contexts and start methods.

Added in version 3.4.

multiprocessing.get_context(*method=None*)

Return a context object which has the same attributes as the `multiprocessing` module.

If *method* is `None` then the default context is returned. Note that if the global start method has not been set, this will set it to the system default See Global start method for more details. Otherwise *method* should be `'fork'`, `'spawn'`, `'forkserver'`. **`ValueError`** is raised if the specified start method is not available. See Contexts and start methods.

Added in version 3.4.

multiprocessing.get_start_method(*allow_none=False*)

Return the name of start method used for starting processes.

If the global start method is not set and *allow_none* is `False`, the global start method is set to the default, and its name is returned. See Global start method for more details.

The return value can be `'fork'`, `'spawn'`, `'forkserver'` or `None`. See Contexts and start methods.

Added in version 3.4.

Changed in version 3.8: On macOS, the *spawn* start method is now the default. The *fork* start method should be considered unsafe as it can lead to crashes of the subprocess. See bpo-33725 [https://bugs.python.org/issue?@action=redirect&bpo=33725].

multiprocessing.set_executable(*executable*)

Set the path of the Python interpreter to use when starting a child process. (By default **`sys.executable`** is used). Embedders will probably need to do some thing like

```
set_executable(os.path.join(sys.exec_prefix, 'pythonw.exe'))
```

before they can create child processes.

Changed in version 3.4: Now supported on POSIX when the `'spawn'` start method is used.

Changed in version 3.11: Accepts a path-like object.

multiprocessing.set_forkserver_preload(*module_names*)

Set a list of module names for the forkserver main process to attempt to import so that their already imported state is inherited by forked processes. Any **`ImportError`** when doing so is silently ignored. This can be used as a performance enhancement to avoid repeated work in every process.

For this to work, it must be called before the forkserver process has been launched (before creating a **`Pool`** or starting a **`Process`**).

Only meaningful when using the `'forkserver'` start method. See Contexts and start methods.

Added in version 3.4.

multiprocessing.set_start_method(*method*, *force=False*)

Set the method which should be used to start child processes. The *method* argument can be `'fork'`, `'spawn'` or `'forkserver'`. Raises **`RuntimeError`** if the start method has already been set and *force* is not `True`. If *method* is `None` and *force* is `True` then the start method is set to `None`. If *method* is `None` and *force* is `False` then the context is set to the default context.

Note that this should be called at most once, and it should be protected inside the `if __name__ == '__main__'` clause of the main module.

See Contexts and start methods.

Added in version 3.4.

Note

`multiprocessing` contains no analogues of `threading.active_count()`, `threading.enumerate()`, `threading.settrace()`, `threading.setprofile()`, `threading.Timer`, or `threading.local`.

Connection Objects

Connection objects allow the sending and receiving of picklable objects or strings. They can be thought of as message oriented connected sockets.

Connection objects are usually created using `Pipe` – see also Listeners and Clients.

class multiprocessing.connection.Connection

send(*obj*)

Send an object to the other end of the connection which should be read using `recv()`.

The object must be picklable. Very large pickles (approximately 32 MiB+, though it depends on the OS) may raise a `ValueError` exception.

recv()

Return an object sent from the other end of the connection using `send()`. Blocks until there is something to receive. Raises `EOFError` if there is nothing left to receive and the other end was closed.

fileno()

Return the file descriptor or handle used by the connection.

close()

Close the connection.

This is called automatically when the connection is garbage collected.

poll([*timeout*])

Return whether there is any data available to be read.

If *timeout* is not specified then it will return immediately. If *timeout* is a number then this specifies the maximum time in seconds to block. If *timeout* is `None` then an infinite timeout is used.

Note that multiple connection objects may be polled at once by using **`multiprocessing.connection.wait()`**.

send_bytes(*buffer*[, *offset*[, *size*]])

Send byte data from a bytes-like object as a complete message.

If *offset* is given then data is read from that position in *buffer*. If *size* is given then that many bytes will be read from buffer. Very large buffers (approximately 32 MiB+, though it depends on the OS) may raise a **`ValueError`** exception

recv_bytes([*maxlength*])

Return a complete message of byte data sent from the other end of the connection as a string. Blocks until there is something to receive. Raises **`EOFError`** if there is nothing left to receive and the other end has closed.

If *maxlength* is specified and the message is longer than *maxlength* then **`OSError`** is raised and the connection will no longer be readable.

Changed in version 3.3: This function used to raise **`IOError`**, which is now an alias of **`OSError`**.

recv_bytes_into(*buffer*[, *offset*])

Read into *buffer* a complete message of byte data sent from the other end of the connection and return the number of bytes in the message. Blocks until there is something to receive. Raises **`EOFError`** if there is nothing left to receive and the other end was closed.

buffer must be a writable bytes-like object. If *offset* is given then the message will be written into the buffer from that position. Offset must be a non-negative integer less than the length of *buffer* (in bytes).

If the buffer is too short then a **`BufferTooShort`** exception is raised and the complete message is available as `e.args[0]` where `e` is the exception instance.

Changed in version 3.3: Connection objects themselves can now be transferred between processes using **`Connection.send()`** and **`Connection.recv()`**.

Connection objects also now support the context management protocol – see Context Manager Types. **`__enter__()`** returns the connection object, and **`__exit__()`** calls **`close()`**.

For example:

```
>>> from multiprocessing import Pipe
>>> a, b = Pipe()
>>> a.send([1, 'hello', None])
>>> b.recv()
[1, 'hello', None]
>>> b.send_bytes(b'thank you')
>>> a.recv_bytes()
b'thank you'
>>> import array
>>> arr1 = array.array('i', range(5))
>>> arr2 = array.array('i', [0] * 10)
>>> a.send_bytes(arr1)
>>> count = b.recv_bytes_into(arr2)
>>> assert count == len(arr1) * arr1.itemsize
>>> arr2
array('i', [0, 1, 2, 3, 4, 0, 0, 0, 0, 0])
```

Warning

The `Connection.recv()` method automatically unpickles the data it receives, which can be a security risk unless you can trust the process which sent the message.

Therefore, unless the connection object was produced using **`Pipe()`** you should only use the `recv()` and `send()` methods after performing some sort of authentication. See Authentication keys.

Warning

If a process is killed while it is trying to read or write to a pipe then the data in the pipe is likely to become corrupted, because it may become impossible to be sure where the message boundaries lie.

Synchronization primitives

Generally synchronization primitives are not as necessary in a multiprocess program as they are in a multithreaded program. See the documentation for `threading` module.

Note that one can also create synchronization primitives by using a manager object – see Managers.

class multiprocessing.Barrier(*parties*[, *action*[, *timeout*]])

A barrier object: a clone of `threading.Barrier`.

Instantiating this class may set the global start method. See Global start method for more details.

Added in version 3.3.

class multiprocessing.BoundedSemaphore([*value*])

A bounded semaphore object: a close analog of `threading.BoundedSemaphore`.

Instantiating this class may set the global start method. See Global start method for more details.

A solitary difference from its close analog exists: its `acquire` method's first argument is named *block*, as is consistent with `Lock.acquire()`.

locked()

Return a boolean indicating whether this object is locked right now.

Added in version 3.14.

Note

On macOS, this is indistinguishable from **`Semaphore`** because `sem_getvalue()` is not implemented on that platform.

class multiprocessing.Condition([*lock*])

A condition variable: an alias for **`threading.Condition`**.

If *lock* is specified then it should be a **`Lock`** or **`RLock`** object from **`multiprocessing`**.

Instantiating this class may set the global start method. See Global start method for more details.

Changed in version 3.3: The **`wait_for()`** method was added.

class multiprocessing.Event

A clone of **`threading.Event`**.

Instantiating this class may set the global start method. See Global start method for more details.

class multiprocessing.Lock

A non-recursive lock object: a close analog of **`threading.Lock`**. Once a process or thread has acquired a lock, subsequent attempts to acquire it from any process or thread will block until it is released; any process or thread may release it. The concepts and behaviors of **`threading.Lock`** as it applies to threads are replicated here in **`multiprocessing.Lock`** as it applies to either processes or threads, except as noted.

Note that **`Lock`** is actually a factory function which returns an instance of `multiprocessing.synchronize.Lock` initialized with a default context.

Instantiating this class may set the global start method. See Global start method for more details.

`Lock` supports the context manager protocol and thus may be used in **`with`** statements.

acquire(*block=True*, *timeout=None*)

Acquire a lock, blocking or non-blocking.

With the *block* argument set to `True` (the default), the method call will block until the lock is in an unlocked state, then set it to locked and return `True`. Note that the name of this first argument differs from that in **`threading.Lock.acquire()`**.

With the *block* argument set to `False`, the method call does not block. If the lock is currently in a locked state, return `False`; otherwise set the lock to a locked state and return `True`.

When invoked with a positive, floating-point value for *timeout*, block for at most the number of seconds specified by *timeout* as long as the lock can not be acquired. Invocations with a negative value for *timeout* are equivalent to a *timeout* of zero. Invocations with a *timeout* value of `None` (the default) set the timeout period to infinite. Note that the treatment of negative or `None` values for *timeout* differs from the implemented behavior in **`threading.Lock.acquire()`**. The *timeout* argument has no practical implications if the *block* argument is set to `False` and is thus ignored. Returns `True` if the lock has been acquired or `False` if the timeout period has elapsed.

release()

Release a lock. This can be called from any process or thread, not only the process or thread which originally acquired the lock.

Behavior is the same as in **`threading.Lock.release()`** except that when invoked on an unlocked lock, a **`ValueError`** is raised.

locked()

Return a boolean indicating whether this object is locked right now.

Added in version 3.14.

class multiprocessing.RLock

A recursive lock object: a close analog of **`threading.RLock`**. A recursive lock must be released by the process or thread that acquired it. Once a process or thread has acquired a recursive lock, the same process or thread may acquire it again without blocking; that process or thread must release it once for each time it has been acquired.

Note that **`RLock`** is actually a factory function which returns an instance of `multiprocessing.synchronize.RLock` initialized with a default context.

Instantiating this class may set the global start method. See Global start method for more details.

`RLock` supports the context manager protocol and thus may be used in **`with`** statements.

acquire(*block=True*, *timeout=None*)

Acquire a lock, blocking or non-blocking.

When invoked with the *block* argument set to `True`, block until the lock is in an unlocked state (not owned by any process or thread) unless the lock is already owned by the current process or thread. The current process or thread then takes ownership of the lock (if it does not already have ownership) and the recursion level inside the lock increments by one, resulting in a return value of `True`. Note that there are several differences in this first argument’s behavior compared to the implementation of `threading.RLock.acquire()`, starting with the name of the argument itself.

When invoked with the *block* argument set to `False`, do not block. If the lock has already been acquired (and thus is owned) by another process or thread, the current process or thread does not take ownership and the recursion level within the lock is not changed, resulting in a return value of `False`. If the lock is in an unlocked state, the current process or thread takes ownership and the recursion level is incremented, resulting in a return value of `True`.

Use and behaviors of the *timeout* argument are the same as in `Lock.acquire()`. Note that some of these behaviors of *timeout* differ from the implemented behaviors in `threading.RLock.acquire()`.

release()

Release a lock, decrementing the recursion level. If after the decrement the recursion level is zero, reset the lock to unlocked (not owned by any process or thread) and if any other processes or threads are blocked waiting for the lock to become unlocked, allow exactly one of them to proceed. If after the decrement the recursion level is still nonzero, the lock remains locked and owned by the calling process or thread.

Only call this method when the calling process or thread owns the lock. An `AssertionError` is raised if this method is called by a process or thread other than the owner or if the lock is in an unlocked (unowned) state. Note that the type of exception raised in this situation differs from the implemented behavior in `threading.RLock.release()`.

locked()

Return a boolean indicating whether this object is locked right now.

Added in version 3.14.

class multiprocessing.Semaphore([*value*])

A semaphore object: a close analog of `threading.Semaphore`.

Instantiating this class may set the global start method. See Global start method for more details.

A solitary difference from its close analog exists: its `acquire` method's first argument is named *block*, as is consistent with `Lock.acquire()`.

get_value()

Return the current value of semaphore.

Note that this may raise `NotImplementedError` on platforms like macOS where `sem_getvalue()` is not implemented.

locked()

Return a boolean indicating whether this object is locked right now.

Added in version 3.14.

Note

On macOS, `sem_timedwait` is unsupported, so calling `acquire()` with a timeout will emulate that function's behavior using a sleeping loop.

Note

Some of this package's functionality requires a functioning shared semaphore implementation on the host operating system. Without one, the **`multiprocessing.synchronize`** module will be disabled, and attempts to import it will result in an **`ImportError`**. See bpo-3770 [https://bugs.python.org/issue?@action=redirect&bpo=3770] for additional information.

Shared `ctypes` Objects

It is possible to create shared objects using shared memory which can be inherited by child processes.

multiprocessing.Value(*typecode_or_type*, **args*, *lock=True*)

Return a **`ctypes`** object allocated from shared memory. By default the return value is actually a synchronized wrapper for the object. The object itself can be accessed via the *value* attribute of a **`Value`**.

typecode_or_type determines the type of the returned object: it is either a ctypes type or a one character typecode of the kind used by the **`array`** module. **args* is passed on to the constructor for the type.

If *lock* is `True` (the default) then a new recursive lock object is created to synchronize access to the value. If *lock* is a **`Lock`** or **`RLock`** object then that will be used to synchronize access to the value. If *lock* is `False` then access to the returned object will not be automatically protected by a lock, so it will not necessarily be “process-safe”.

Operations like `+=` which involve a read and write are not atomic. So if, for instance, you want to atomically increment a shared value it is insufficient to just do

```
counter.value += 1
```

Assuming the associated lock is recursive (which it is by default) you can instead do

```
with counter.get_lock():
    counter.value += 1
```

Note that *lock* is a keyword-only argument.

multiprocessing.Array(*typecode_or_type*, *size_or_initializer*, *, *lock=True*)

Return a ctypes array allocated from shared memory. By default the return value is actually a synchronized wrapper for the array.

typecode_or_type determines the type of the elements of the returned array: it is either a ctypes type or a one character typecode of the kind used by the **`array`** module with the exception of `'w'`, which is not supported. In addition, the `'c'` typecode is an alias for **`ctypes.c_char`**. If *size_or_initializer* is an integer, then it determines the length of the array, and the array will be initially zeroed. Otherwise, *size_or_initializer* is a sequence which is used to initialize the array and whose length determines the length of the array.

If *lock* is `True` (the default) then a new lock object is created to synchronize access to the value. If *lock* is a `Lock` or `RLock` object then that will be used to synchronize access to the value. If *lock* is `False` then access to the returned object will not be automatically protected by a lock, so it will not necessarily be “process-safe”.

Note that *lock* is a keyword only argument.

Note that an array of `ctypes.c_char` has *value* and *raw* attributes which allow one to use it to store and retrieve strings.

The `multiprocessing.sharedctypes` module

The `multiprocessing.sharedctypes` module provides functions for allocating `ctypes` objects from shared memory which can be inherited by child processes.

Note

Although it is possible to store a pointer in shared memory remember that this will refer to a location in the address space of a specific process. However, the pointer is quite likely to be invalid in the context of a second process and trying to dereference the pointer from the second process may cause a crash.

multiprocessing.sharedctypes.RawArray(*typecode_or_type*, *size_or_initializer*)

Return a ctypes array allocated from shared memory.

typecode_or_type determines the type of the elements of the returned array: it is either a ctypes type or a one character typecode of the kind used by the `array` module. If *size_or_initializer* is an integer then it determines the length of the array, and the array will be initially zeroed. Otherwise *size_or_initializer* is a sequence which is used to initialize the array and whose length determines the length of the array.

Note that setting and getting an element is potentially non-atomic – use `Array()` instead to make sure that access is automatically synchronized using a lock.

multiprocessing.sharedctypes.RawValue(*typecode_or_type*, **args*)

Return a ctypes object allocated from shared memory.

typecode_or_type determines the type of the returned object: it is either a ctypes type or a one character typecode of the kind used by the `array` module. **args* is passed on to the constructor for the type.

Note that setting and getting the value is potentially non-atomic – use `Value()` instead to make sure that access is automatically synchronized using a lock.

Note that an array of `ctypes.c_char` has `value` and `raw` attributes which allow one to use it to store and retrieve strings – see documentation for `ctypes`.

multiprocessing.sharedctypes.Array(*typecode_or_type*, *size_or_initializer*, *, *lock=True*, *ctx=None*)

The same as `RawArray()` except that depending on the value of *lock* a process-safe synchronization wrapper may be returned instead of a raw ctypes array.

If *lock* is `True` (the default) then a new lock object is created to synchronize access to the value. If *lock* is a

`Lock` or `RLock` object then that will be used to synchronize access to the value. If *lock* is `False` then access to the returned object will not be automatically protected by a lock, so it will not necessarily be “process-safe”.

ctx is a context object, or `None` (use the current context). If `None`, calling this may set the global start method. See Global start method for more details.

Note that *lock* and *ctx* are keyword-only parameters.

multiprocessing.sharedctypes.Value(*typecode_or_type*, **args*, *lock=True*, *ctx=None*)

The same as `RawValue()` except that depending on the value of *lock* a process-safe synchronization wrapper may be returned instead of a raw ctypes object.

If *lock* is `True` (the default) then a new lock object is created to synchronize access to the value. If *lock* is a `Lock` or `RLock` object then that will be used to synchronize access to the value. If *lock* is `False` then access to the returned object will not be automatically protected by a lock, so it will not necessarily be “process-safe”.

ctx is a context object, or `None` (use the current context). If `None`, calling this may set the global start method. See Global start method for more details.

Note that *lock* and *ctx* are keyword-only parameters.

multiprocessing.sharedctypes.copy(*obj*)

Return a ctypes object allocated from shared memory which is a copy of the ctypes object *obj*.

multiprocessing.sharedctypes.synchronized(*obj*, *lock=None*, *ctx=None*)

Return a process-safe wrapper object for a ctypes object which uses *lock* to synchronize access. If *lock* is `None` (the default) then a **`multiprocessing.RLock`** object is created automatically.

ctx is a context object, or `None` (use the current context). If `None`, calling this may set the global start method. See Global start method for more details.

A synchronized wrapper will have two methods in addition to those of the object it wraps: **`get_obj()`** returns the wrapped object and **`get_lock()`** returns the lock object used for synchronization.

Note that accessing the ctypes object through the wrapper can be a lot slower than accessing the raw ctypes object.

Changed in version 3.5: Synchronized objects support the context manager protocol.

The table below compares the syntax for creating shared ctypes objects from shared memory with the normal ctypes syntax. (In the table `MyStruct` is some subclass of **`ctypes.Structure`**.)

ctypes	sharedctypes using type	sharedctypes using typecode
c_double(2.4)	RawValue(c_double, 2.4)	RawValue('d', 2.4)
MyStruct(4, 6)	RawValue(MyStruct, 4, 6)	

ctypes	sharedctypes using type	sharedctypes using typecode
(c_short * 7)()	RawArray(c_short, 7)	RawArray(‘h’, 7)
(c_int * 3)(9, 2, 8)	RawArray(c_int, (9, 2, 8))	RawArray(‘i’, (9, 2, 8))

Below is an example where a number of ctypes objects are modified by a child process:

```
from multiprocessing import Process, Lock
from multiprocessing.sharedctypes import Value, Array
from ctypes import Structure, c_double

class Point(Structure):
    _fields_ = [('x', c_double), ('y', c_double)]

def modify(n, x, s, A):
    n.value **= 2
    x.value **= 2
    s.value = s.value.upper()
    for a in A:
        a.x **= 2
        a.y **= 2

if __name__ == '__main__':
    lock = Lock()

    n = Value('i', 7)
    x = Value(c_double, 1.0/3.0, lock=False)
    s = Array('c', b'hello world', lock=lock)
    A = Array(Point, [(1.875,-6.25), (-5.75,2.0), (2.375,9.5)],
lock=lock)

    p = Process(target=modify, args=(n, x, s, A))
    p.start()
    p.join()

    print(n.value)
    print(x.value)
    print(s.value)
    print([(a.x, a.y) for a in A])
```

The results printed are

```
49
0.1111111111111111
HELLO WORLD
[(3.515625, 39.0625), (33.0625, 4.0), (5.640625, 90.25)]
```

Managers

Managers provide a way to create data which can be shared between different processes, including sharing over a network between processes running on different machines. A manager object controls a server process which manages *shared objects*. Other processes can access the shared objects by using proxies.

multiprocessing.Manager()

> Returns a started `SyncManager` object which can be used for sharing objects between processes. The returned manager object corresponds to a spawned child process and has methods which will create shared objects and return corresponding proxies.

Manager processes will be shutdown as soon as they are garbage collected or their parent process exits. The manager classes are defined in the `multiprocessing.managers` module:

class
multiprocessing.managers.BaseManager(*address*=*None*,

authkey=None, *serializer='pickle'*, *ctx=None*, ***, *shutdown_timeout=1.0*)

Create a BaseManager object.

Once created one should call [`start()`](#) or `get_server().serve_forever()` to ensure that the manager object refers to a started manager process.

address is the address on which the manager process listens for new connections. If *address* is `None` then an arbitrary one is chosen.

authkey is the authentication key which will be used to check the validity of incoming connections to the server process. If *authkey* is `None` then `current_process().authkey` is used. Otherwise *authkey* is used and it must be a byte string.

serializer must be `'pickle'` (use [`pickle`](#) serialization) or `'xmlrpclib'` (use [`xmlrpc.client`](#) serialization).

ctx is a context object, or `None` (use the current context). If `None`, calling this may set the global start method. See [Global start method](#) for more details.

shutdown_timeout is a timeout in seconds used to wait until the process used by the manager completes in the [`shutdown()`](#) method. If the shutdown times out, the process is terminated. If terminating the process also times out, the process is killed.

Changed in version 3.11: Added the *shutdown_timeout* parameter.

start([*initializer*[, *initargs*]])

Start a subprocess to start the manager. If *initializer* is not `None` then the subprocess will call `initializer(*initargs)` when it starts.

get_server()

Returns a **`Server`** object which represents the actual server under the control of the Manager. The **`Server`** object supports the **`serve_forever()`** method:

```
>>> from multiprocessing.managers import BaseManager
>>> manager = BaseManager(address=('', 50000),
authkey=b'abc')
>>> server = manager.get_server()
>>> server.serve_forever()
```

`Server` additionally has an **`address`** attribute.

connect()

Connect a local manager object to a remote manager process:

```
>>> from multiprocessing.managers import BaseManager
>>> m = BaseManager(address=('127.0.0.1', 50000),
authkey=b'abc')
>>> m.connect()
```

shutdown()

Stop the process used by the manager. This is only available if **`start()`** has been used to start the server process.

This can be called multiple times.

register(*typeid*[, *callable*[, *proxytype*[, *exposed*[, *method_to_typeid*[, *create_method*]]]]])

A classmethod which can be used for registering a type or callable with the manager class.

typeid is a “type identifier” which is used to identify a particular type of shared object. This must be a string.

callable is a callable used for creating objects for this type identifier. If a manager instance will be connected to the server using the `connect()` method, or if the *create_method* argument is `False` then this can be left as `None`.

proxytype is a subclass of `BaseProxy` which is used to create proxies for shared objects with this *typeid*. If `None` then a proxy class is created automatically.

exposed is used to specify a sequence of method names which proxies for this typeid should be allowed to access using `BaseProxy._callmethod()`. (If *exposed* is `None` then **`proxytype._exposed_`** is used instead if it exists.) In the case where no exposed list is specified, all “public methods” of the shared object will be accessible. (Here a “public method” means any attribute which has a `__call__()` method and whose name does not begin with `'_'`.)

method_to_typeid is a mapping used to specify the return type of those exposed methods which should return a proxy. It maps method names to typeid strings. (If *method_to_typeid* is `None` then **`proxytype._method_to_typeid_`** is used instead if it exists.) If a method’s name is not a key of this mapping or if the mapping is `None` then the object returned by the method will be copied by value.

create_method determines whether a method should be created with name *typeid* which can be used to tell the server process to create a new shared object and return a proxy for it. By default it is `True`.

`BaseManager` instances also have one read-only property:

address

The address used by the manager.

Changed in version 3.3: Manager objects support the context management protocol – see Context Manager Types. `__enter__()` starts the server process (if it has not already started) and then returns the manager object. `__exit__()` calls `shutdown()`.

In previous versions `__enter__()` did not start the manager's server process if it was not already started.

class multiprocessing.managers.SyncManager

A subclass of `BaseManager` which can be used for the synchronization of processes. Objects of this type are returned by `multiprocessing.Manager()`.

Its methods create and return Proxy Objects for a number of commonly used data types to be synchronized across processes. This notably includes shared lists and dictionaries.

Barrier(*parties*[, *action*[, *timeout*]])

Create a shared `threading.Barrier` object and return a proxy for it.

Added in version 3.3.

BoundedSemaphore([*value*])

Create a shared `threading.BoundedSemaphore` object and return a proxy for it.

Condition([*lock*])

Create a shared `threading.Condition` object and return a proxy for it.

If *lock* is supplied then it should be a proxy for a `threading.Lock` or `threading.RLock` object.

Changed in version 3.3: The `wait_for()` method was added.

Event()

Create a shared `threading.Event` object and return a proxy for it.

Lock()

Create a shared `threading.Lock` object and return a proxy for it.

Namespace()

Create a shared `Namespace` object and return a proxy for it.

Queue([*maxsize*])

Create a shared `queue.Queue` object and return a proxy for it.

RLock()

Create a shared `threading.RLock` object and return a proxy for it.

Semaphore([*value*])

Create a shared `threading.Semaphore` object and return a proxy for it.

Array(*typecode*, *sequence*)

Create an array and return a proxy for it.

Value(*typecode*, *value*)

Create an object with a writable `value` attribute and return a proxy for it.

dict()
dict(*mapping*)
dict(*sequence*)

Create a shared **`dict`** object and return a proxy for it.

list()
list(*sequence*)

Create a shared **`list`** object and return a proxy for it.

set()
set(*sequence*)
set(*mapping*)

Create a shared **`set`** object and return a proxy for it.

Added in version 3.14: **`set`** support was added.

Changed in version 3.6: Shared objects are capable of being nested. For example, a shared container object such as a shared list can contain other shared objects which will all be managed and synchronized by the **`SyncManager`**.

class multiprocessing.managers.Namespace

A type that can register with **`SyncManager`**.

A namespace object has no public methods, but does have writable attributes. Its representation shows the values of its attributes.

However, when using a proxy for a namespace object, an attribute beginning with `'_'` will be an attribute of the proxy and not an attribute of the referent:

```
>>> mp_context = multiprocessing.get_context('spawn')
>>> manager = mp_context.Manager()
>>> Global = manager.Namespace()
>>> Global.x = 10
>>> Global.y = 'hello'
>>> Global._z = 12.3    # this is an attribute of the proxy
>>> print(Global)
Namespace(x=10, y='hello')
```

Customized managers

To create one's own manager, one creates a subclass of `BaseManager` and uses the `register()` classmethod to register new types or callables with the manager class. For example:

```
from multiprocessing.managers import BaseManager

class MathsClass:
    def add(self, x, y):
        return x + y
    def mul(self, x, y):
        return x * y

class MyManager(BaseManager):
    pass

MyManager.register('Maths', MathsClass)

if __name__ == '__main__':
    with MyManager() as manager:
        maths = manager.Maths()
        print(maths.add(4, 3))         # prints 7
        print(maths.mul(7, 8))         # prints 56
```

Using a remote manager

It is possible to run a manager server on one machine and have clients use it from other machines (assuming that the firewalls involved allow it).

Running the following commands creates a server for a single shared queue which remote clients can access:

```
>>> from multiprocessing.managers import BaseManager
>>> from queue import Queue
>>> queue = Queue()
>>> class QueueManager(BaseManager): pass
>>> QueueManager.register('get_queue', callable=lambda:queue)
>>> m = QueueManager(address=('', 50000),
authkey=b'abracadabra')
>>> s = m.get_server()
>>> s.serve_forever()
```

One client can access the server as follows:

```
>>> from multiprocessing.managers import BaseManager
>>> class QueueManager(BaseManager): pass
>>> QueueManager.register('get_queue')
>>> m = QueueManager(address=('foo.bar.org', 50000),
authkey=b'abracadabra')
>>> m.connect()
>>> queue = m.get_queue()
>>> queue.put('hello')
```

Another client can also use it:

```
>>> from multiprocessing.managers import BaseManager
>>> class QueueManager(BaseManager): pass
>>> QueueManager.register('get_queue')
>>> m = QueueManager(address=('foo.bar.org', 50000),
authkey=b'abracadabra')
>>> m.connect()
>>> queue = m.get_queue()
>>> queue.get()
'hello'
```

Local processes can also access that queue, using the code from above on the client to access it remotely:

```
>>> from multiprocessing import Process, Queue
>>> from multiprocessing.managers import BaseManager
>>> class Worker(Process):
...     def __init__(self, q):
...         self.q = q
...         super().__init__()
...     def run(self):
...         self.q.put('local hello')
...
>>> queue = Queue()
>>> w = Worker(queue)
>>> w.start()
>>> class QueueManager(BaseManager): pass
...
>>> QueueManager.register('get_queue', callable=lambda: queue)
>>> m = QueueManager(address=('', 50000),
authkey=b'abracadabra')
>>> s = m.get_server()
>>> s.serve_forever()
```

Proxy Objects

A proxy is an object which *refers* to a shared object which lives (presumably) in a different process. The shared object is said to be the *referent* of the proxy. Multiple proxy objects may have the same referent.

A proxy object has methods which invoke corresponding methods of its referent (although not every method of the referent will necessarily be available through the proxy). In this way, a proxy can be used just like its referent can:

```
>>> mp_context = multiprocessing.get_context('spawn')
>>> manager = mp_context.Manager()
>>> l = manager.list([i*i for i in range(10)])
>>> print(l)
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
>>> print(repr(l))
<ListProxy object, typeid 'list' at 0x...>
>>> l[4]
16
```

```
>>> l[2:5]
[4, 9, 16]
```

Notice that applying `str()` to a proxy will return the representation of the referent, whereas applying `repr()` will return the representation of the proxy.

An important feature of proxy objects is that they are picklable so they can be passed between processes. As such, a referent can contain Proxy Objects. This permits nesting of these managed lists, dicts, and other Proxy Objects:

```
>>> a = manager.list()
>>> b = manager.list()
>>> a.append(b)         # referent of a now contains referent
of b
>>> print(a, b)
[<ListProxy object, typeid 'list' at ...>] []
>>> b.append('hello')
>>> print(a[0], b)
['hello'] ['hello']
```

Similarly, dict and list proxies may be nested inside one another:

```
>>> l_outer = manager.list([ manager.dict() for i in range(2)
])
>>> d_first_inner = l_outer[0]
>>> d_first_inner['a'] = 1
>>> d_first_inner['b'] = 2
>>> l_outer[1]['c'] = 3
>>> l_outer[1]['z'] = 26
>>> print(l_outer[0])
{'a': 1, 'b': 2}
>>> print(l_outer[1])
{'c': 3, 'z': 26}
```

If standard (non-proxy) `list` or `dict` objects are contained in a referent, modifications to those mutable values will not be propagated through the manager because the proxy has no way of knowing when the values contained within are modified. However, storing a value in a container proxy (which triggers a `__setitem__` on the proxy object) does propagate through the manager and so to effectively modify such an item, one could re-assign the modified value to the container proxy:

```
# create a list proxy and append a mutable object (a
dictionary)
lproxy = manager.list()
lproxy.append({})
# now mutate the dictionary
d = lproxy[0]
d['a'] = 1
d['b'] = 2
# at this point, the changes to d are not yet synced, but by
# updating the dictionary, the proxy is notified of the change
lproxy[0] = d
```

This approach is perhaps less convenient than employing nested Proxy Objects for most use cases but also demonstrates a level of control over the synchronization.

Note

The proxy types in **`multiprocessing`** do nothing to support comparisons by value. So, for instance, we have:

```
>>> manager.list([1,2,3]) == [1,2,3]
False
```

One should just use a copy of the referent instead when making comparisons.

class multiprocessing.managers.BaseProxy

Proxy objects are instances of subclasses of **`BaseProxy`**.

_callmethod(*methodname*[, *args*[, *kwds*]])

Call and return the result of a method of the proxy's referent.

If `proxy` is a proxy whose referent is `obj` then the expression

```
proxy._callmethod(methodname, args, kwds)
```

will evaluate the expression

```
getattr(obj, methodname)(*args, **kwds)
```

in the manager's process.

The returned value will be a copy of the result of the call or a proxy to a new shared object – see documentation for the *method_to_typeid* argument of `BaseManager.register()`.

If an exception is raised by the call, then is re-raised by `_callmethod()`. If some other exception is raised in the manager's process then this is converted into a **`RemoteError`** exception and is raised by `_callmethod()`.

Note in particular that an exception will be raised if *methodname* has not been *exposed*.

An example of the usage of `_callmethod()`:

```
>>> l = manager.list(range(10))
>>> l._callmethod('__len__')
10
>>> l._callmethod('__getitem__', (slice(2, 7),)) #
equivalent to l[2:7]
[2, 3, 4, 5, 6]
>>> l._callmethod('__getitem__', (20,))          #
equivalent to l[20]
Traceback (most recent call last):
```

```
...
IndexError: list index out of range
```

_getvalue()

Return a copy of the referent.

If the referent is unpicklable then this will raise an exception.

__repr__()

Return a representation of the proxy object.

__str__()

Return the representation of the referent.

Cleanup

A proxy object uses a weakref callback so that when it gets garbage collected it deregisters itself from the manager which owns its referent.

A shared object gets deleted from the manager process when there are no longer any proxies referring to it.

Process Pools

One can create a pool of processes which will carry out tasks submitted to it with the `Pool` class.

class multiprocessing.pool.Pool([*processes*[, *initializer*[, *initargs*[, *maxtasksperchild*[, *context*]]]]])

A process pool object which controls a pool of worker processes to which jobs can be submitted. It supports asynchronous results with timeouts and callbacks and has a parallel map implementation.

processes is the number of worker processes to use. If *processes* is `None` then the number returned by `os.process_cpu_count()` is used.

If *initializer* is not `None` then each worker process will call `initializer(*initargs)` when it starts.

maxtasksperchild is the number of tasks a worker process can complete before it will exit and be replaced with a fresh worker process, to enable unused resources to be freed. The default *maxtasksperchild* is `None`, which means worker processes will live as long as the pool.

context can be used to specify the context used for starting the worker processes. Usually a pool is created using the function **`multiprocessing.Pool()`** or the `Pool()` method of a context object. In both cases *context* is set appropriately. If `None`, calling this function will have the side effect of setting the current global start method if it has not been set already. See the **`get_context()`** function.

Note that the methods of the pool object should only be called by the process which created the pool.

Warning

`multiprocessing.pool` objects have internal resources that need to be properly managed (like any other resource) by using the pool as a context manager or by calling `close()` and `terminate()` manually. Failure to do this can lead to the process hanging on finalization.

Note that it is **not correct** to rely on the garbage collector to destroy the pool as CPython does not assure that the finalizer of the pool will be called (see `object.__del__()` for more information).

Changed in version 3.2: Added the *maxtasksperchild* parameter.

Changed in version 3.4: Added the *context* parameter.

Changed in version 3.13: *processes* uses `os.process_cpu_count()` by default, instead of `os.cpu_count()`.

Note

Worker processes within a `Pool` typically live for the complete duration of the Pool's work queue. A frequent pattern found in other systems (such as Apache, mod_wsgi, etc) to free resources held by workers is to allow a worker within a pool to complete only a set amount of work before being exiting, being cleaned up and a new process spawned to replace the old one. The *maxtasksperchild* argument to the `Pool` exposes this ability to the end user.

apply(*func*[, *args*[, *kwds*]])

Call *func* with arguments *args* and keyword arguments *kwds*. It blocks until the result is ready. Given this blocks, `apply_async()` is better suited for performing work in parallel. Additionally, *func* is only executed in one of the workers of the pool.

apply_async(*func*[, *args*[, *kwds*[, *callback*[, *error_callback*]]]])

A variant of the `apply()` method which returns a `AsyncResult` object.

If *callback* is specified then it should be a callable which accepts a single argument. When the result becomes ready *callback* is applied to it, that is unless the call failed, in which case the *error_callback* is applied instead.

If *error_callback* is specified then it should be a callable which accepts a single argument. If the target function fails, then the *error_callback* is called with the exception instance.

Callbacks should complete immediately since otherwise the thread which handles the results will get blocked.

map(*func*, *iterable*[, *chunksize*])

A parallel equivalent of the `map()` built-in function (it supports only one *iterable* argument though, for multiple iterables see `starmap()`). It blocks until the result is ready.

This method chops the iterable into a number of chunks which it submits to the process pool as separate tasks. The (approximate) size of these chunks can be specified by setting *chunksize* to a positive integer.

Note that it may cause high memory usage for very long iterables. Consider using `imap()` or `imap_unordered()` with explicit *chunksize* option for better efficiency.

map_async(*func*, *iterable*[, *chunksize*[, *callback*[, *error_callback*]]])

A variant of the `map()` method which returns a `AsyncResult` object.

If *callback* is specified then it should be a callable which accepts a single argument. When the result becomes ready *callback* is applied to it, that is unless the call failed, in which case the *error_callback* is applied instead.

If *error_callback* is specified then it should be a callable which accepts a single argument. If the target function fails, then the *error_callback* is called with the exception instance.

Callbacks should complete immediately since otherwise the thread which handles the results will get blocked.

imap(*func*, *iterable*[, *chunksize*])

A lazier version of `map()`.

The *chunksize* argument is the same as the one used by the `map()` method. For very long iterables using a large value for *chunksize* can make the job complete **much** faster than using the default value of `1`.

Also if *chunksize* is `1` then the **`next()`** method of the iterator returned by the `imap()` method has an optional *timeout* parameter: `next(timeout)` will raise `multiprocessing.TimeoutError` if the result cannot be returned within *timeout* seconds.

imap_unordered(*func*, *iterable*[, *chunksize*])

The same as `imap()` except that the ordering of the results from the returned iterator should be considered arbitrary. (Only when there is only one worker process is the order guaranteed to be “correct”.)

starmap(*func*, *iterable*[, *chunksize*])

Like `map()` except that the elements of the *iterable* are expected to be iterables that are unpacked as arguments.

Hence an *iterable* of `[(1,2), (3, 4)]` results in `[func(1,2), func(3,4)]`.

Added in version 3.3.

starmap_async(*func*, *iterable*[, *chunksize*[, *callback*[, *error_callback*]]])

A combination of `starmap()` and `map_async()` that iterates over *iterable* of iterables and calls *func* with the iterables unpacked. Returns a result object.

Added in version 3.3.

close()

Prevents any more tasks from being submitted to the pool. Once all the tasks have been completed the worker processes will exit.

terminate()

Stops the worker processes immediately without completing outstanding work. When the pool object is garbage collected `terminate()` will be called immediately.

join()

Wait for the worker processes to exit. One must call `close()` or `terminate()` before using `join()`.

Changed in version 3.3: Pool objects now support the context management protocol – see Context Manager

Types. `__enter__()` returns the pool object, and `__exit__()` calls `terminate()`.

class multiprocessing.pool.AsyncResult

The class of the result returned by `Pool.apply_async()` and `Pool.map_async()`.

get([*timeout*])

Return the result when it arrives. If *timeout* is not `None` and the result does not arrive within *timeout* seconds then `multiprocessing.TimeoutError` is raised. If the remote call raised an exception then that exception will be reraised by `get()`.

wait([*timeout*])

Wait until the result is available or until *timeout* seconds pass.

ready()

Return whether the call has completed.

successful()

Return whether the call completed without raising an exception. Will raise `ValueError` if the result is not ready.

Changed in version 3.7: If the result is not ready, `ValueError` is raised instead of `AssertionError`.

The following example demonstrates the use of a pool:

```
from multiprocessing import Pool
import time

def f(x):
    return x*x
```

```
if __name__ == '__main__':
    with Pool(processes=4) as pool:         # start 4 worker
processes
        result = pool.apply_async(f, (10,)) # evaluate "f(10)"
asynchronously in a single process
        print(result.get(timeout=1))        # prints "100"
unless your computer is *very* slow

        print(pool.map(f, range(10)))       # prints "[0, 1,
4,..., 81]"

        it = pool.imap(f, range(10))
        print(next(it))                     # prints "0"
        print(next(it))                     # prints "1"
        print(it.next(timeout=1))           # prints "4" unless
your computer is *very* slow

        result = pool.apply_async(time.sleep, (10,))
        print(result.get(timeout=1))        # raises
multiprocessing.TimeoutError
```

Listeners and Clients

Usually message passing between processes is done using queues or by using `Connection` objects returned by `Pipe()`.

However, the `multiprocessing.connection` module allows some extra flexibility. It basically gives a high level message oriented API for dealing with sockets or Windows named pipes. It also has support for *digest authentication* using the `hmac` module, and for polling multiple connections at the same time.

multiprocessing.connection.deliver_challenge(*connection*, *authkey*)

> Send a randomly generated message to the other end of the connection and wait for a reply.

If the reply matches the digest of the message using *authkey* as the key then a welcome message is sent to the other end of the connection. Otherwise `AuthenticationError` is raised.

multiprocessing.connection.answer_challenge(*connection*, *authkey*)

Receive a message, calculate the digest of the message using *authkey* as the key, and then send the digest back.

If a welcome message is not received, then `AuthenticationError` is raised.

multiprocessing.connection.Client(*address*[, *family*[, *authkey*]])

Attempt to set up a connection to the listener which is using address *address*, returning a `Connection`.

The type of the connection is determined by *family* argument, but this can generally be omitted since it can usually be inferred from the format of *address*. (See Address Formats)

If *authkey* is given and not `None`, it should be a byte string and will be used as the secret key for an HMAC-based authentication challenge. No authentication is done if *authkey* is `None`. `AuthenticationError` is raised if authentication fails. See Authentication keys.

class multiprocessing.connection.Listener([*address*[, *family*[, *backlog*[, *authkey*]]]])

A wrapper for a bound socket or Windows named pipe which is ‘listening’ for connections.

address is the address to be used by the bound socket or named pipe of the listener object.

Note

If an address of ‘0.0.0.0’ is used, the address will not be a connectable end point on Windows. If you require a connectable end-point, you should use ‘127.0.0.1’.

family is the type of socket (or named pipe) to use. This can be one of the strings `'AF_INET'` (for a TCP socket), `'AF_UNIX'` (for a Unix domain socket) or `'AF_PIPE'` (for a Windows named pipe). Of these only the first is guaranteed to be available. If *family* is `None` then the family is inferred from the format of *address*. If *address* is also `None` then a default is chosen. This default is the family which is assumed to be the fastest available. See Address Formats. Note that if *family* is `'AF_UNIX'` and address is `None` then the socket will be created in a private temporary directory created using **`tempfile.mkstemp()`**.

If the listener object uses a socket then *backlog* (1 by default) is passed to the **`listen()`** method of the socket once it has been bound.

If *authkey* is given and not `None`, it should be a byte string and will be used as the secret key for an HMAC-based authentication challenge. No authentication is done if *authkey* is `None`. **`AuthenticationError`** is raised if authentication fails. See Authentication keys.

accept()

Accept a connection on the bound socket or named pipe of the listener object and return a **`Connection`** object. If authentication is attempted and fails, then **`AuthenticationError`** is raised.

close()

Close the bound socket or named pipe of the listener object. This is called automatically when the listener is garbage collected. However it is advisable to call it explicitly.

Listener objects have the following read-only properties:

address

The address which is being used by the Listener object.

last_accepted

The address from which the last accepted connection came. If this is unavailable then it is `None`.

Changed in version 3.3: Listener objects now support the context management protocol – see Context Manager Types. **`__enter__()`** returns the listener object, and **`__exit__()`** calls **`close()`**.

multiprocessing.connection.wait(*object_list*, *timeout=None*)

Wait till an object in *object_list* is ready. Returns the list of those objects in *object_list* which are ready. If *timeout* is a float then the call blocks for at most that many seconds. If *timeout* is `None` then it will block for an unlimited period. A negative timeout is equivalent to a zero timeout.

For both POSIX and Windows, an object can appear in *object_list* if it is

- a readable **`Connection`** object;
- a connected and readable **`socket.socket`** object; or

- the `sentinel` attribute of a `Process` object.

A connection or socket object is ready when there is data available to be read from it, or the other end has been closed.

POSIX: `wait(object_list, timeout)` almost equivalent `select.select(object_list, [], [], timeout)`. The difference is that, if `select.select()` is interrupted by a signal, it can raise `OSError` with an error number of `EINTR`, whereas `wait()` will not.

Windows: An item in *object_list* must either be an integer handle which is waitable (according to the definition used by the documentation of the Win32 function `WaitForMultipleObjects()`) or it can be an object with a `fileno()` method which returns a socket handle or pipe handle. (Note that pipe handles and socket handles are **not** waitable handles.)

Added in version 3.3.

Examples

The following server code creates a listener which uses `'secret password'` as an authentication key. It then waits for a connection and sends some data to the client:

```
from multiprocessing.connection import Listener
from array import array

address = ('localhost', 6000)     # family is deduced to be
'AF_INET'

with Listener(address, authkey=b'secret password') as listener:
    with listener.accept() as conn:
        print('connection accepted from',
listener.last_accepted)
```

```
        conn.send([2.25, None, 'junk', float])

        conn.send_bytes(b'hello')

        conn.send_bytes(array('i', [42, 1729]))
```

The following code connects to the server and receives some data from the server:

```
from multiprocessing.connection import Client
from array import array

address = ('localhost', 6000)

with Client(address, authkey=b'secret password') as conn:
    print(conn.recv())                  # => [2.25, None,
'junk', float]

    print(conn.recv_bytes())            # => 'hello'

    arr = array('i', [0, 0, 0, 0, 0])
    print(conn.recv_bytes_into(arr))    # => 8
    print(arr)                          # => array('i', [42,
1729, 0, 0, 0])
```

The following code uses `wait()` to wait for messages from multiple processes at once:

```
from multiprocessing import Process, Pipe, current_process
from multiprocessing.connection import wait

def foo(w):
    for i in range(10):
        w.send((i, current_process().name))
    w.close()

if __name__ == '__main__':
    readers = []

    for i in range(4):
        r, w = Pipe(duplex=False)
        readers.append(r)
        p = Process(target=foo, args=(w,))
        p.start()
        # We close the writable end of the pipe now to be sure
that
        # p is the only process which owns a handle for it.
This
        # ensures that when p closes its handle for the
writable end,
        # wait() will promptly report the readable end as being
ready.
        w.close()

    while readers:
        for r in wait(readers):
            try:
                msg = r.recv()
            except EOFError:
                readers.remove(r)
            else:
                print(msg)
```

Address Formats

- An `'AF_INET'` address is a tuple of the form `(hostname, port)` where *hostname* is a string and *port* is an integer.
- An `'AF_UNIX'` address is a string representing a filename on the filesystem.
- An `'AF_PIPE'` address is a string of the form `r'\\.\pipe\PipeName'`. To use **`Client()`** to connect to a named pipe on a remote computer called *ServerName* one should use an address of the form `r'\\ServerName\pipe\PipeName'` instead.

Note that any string beginning with two backslashes is assumed by default to be an `'AF_PIPE'` address rather than an `'AF_UNIX'` address.

Authentication keys

When one uses `Connection.recv`, the data received is automatically unpickled. Unfortunately unpickling data from an untrusted source is a security risk. Therefore `Listener` and `Client()` use the `hmac` module to provide digest authentication.

An authentication key is a byte string which can be thought of as a password: once a connection is established both ends will demand proof that the other knows the authentication key. (Demonstrating that both ends are using the same key does **not** involve sending the key over the connection.)

If authentication is requested but no authentication key is specified then the return value of `current_process().authkey` is used (see `Process`). This value will be automatically inherited by any `Process` object that the current process creates. This means that (by default) all processes of a multi-process program will share a single authentication key which can be used when setting up connections between themselves.

Suitable authentication keys can also be generated by using `os.urandom()`.

Logging

Some support for logging is available. Note, however, that the `logging` package does not use process shared locks so it is possible (depending on the handler type) for messages from different processes to get mixed up.

multiprocessing.get_logger()

Returns the logger used by `multiprocessing`. If necessary, a new one will be created.

When first created the logger has level `logging.NOTSET` and no default handler. Messages sent to this logger will not by default propagate to the root logger.

Note that on Windows child processes will only inherit the level of the parent process's logger – any other customization of the logger will not be inherited.

multiprocessing.log_to_stderr(*level=None*)

This function performs a call to `get_logger()` but in addition to returning the logger created by get_logger, it adds a handler which sends output to `sys.stderr` using format `'[%(levelname)s/%(processName)s] %(message)s'`. You can modify `levelname` of the logger by passing a `level` argument.

Below is an example session with logging turned on:

```
>>> import multiprocessing, logging
>>> logger = multiprocessing.log_to_stderr()
>>> logger.setLevel(logging.INFO)
>>> logger.warning('doomed')
[WARNING/MainProcess] doomed
>>> m = multiprocessing.Manager()
[INFO/SyncManager-...] child process calling self.run()
[INFO/SyncManager-...] created temp directory /.../pymp-...
[INFO/SyncManager-...] manager serving at '/.../listener-...'
>>> del m
[INFO/MainProcess] sending shutdown message to manager
[INFO/SyncManager-...] manager exiting with exitcode 0
```

For a full table of logging levels, see the `logging` module.

The `multiprocessing.dummy` module

`multiprocessing.dummy` replicates the API of `multiprocessing` but is no more than a wrapper around the `threading` module.

In particular, the `Pool` function provided by **`multiprocessing.dummy`** returns an instance of **`ThreadPool`**, which is a subclass of **`Pool`** that supports all the same method calls but uses a pool of worker threads rather than worker processes.

class multiprocessing.pool.ThreadPool([*processes*[, *initializer*[, *initargs*]]])

A thread pool object which controls a pool of worker threads to which jobs can be submitted. **`ThreadPool`** instances are fully interface compatible with **`Pool`** instances, and their resources must also be properly managed, either by using the pool as a context manager or by calling **`close()`** and **`terminate()`** manually.

processes is the number of worker threads to use. If *processes* is `None` then the number returned by **`os.process_cpu_count()`** is used.

If *initializer* is not `None` then each worker process will call `initializer(*initargs)` when it starts.

Unlike **`Pool`**, *maxtasksperchild* and *context* cannot be provided.

Note

A **`ThreadPool`** shares the same interface as **`Pool`**, which is designed around a pool of processes and predates the introduction of the **`concurrent.futures`** module. As such, it inherits some operations that don’t make sense for a pool backed by threads, and it has its own type for representing the status of asynchronous jobs, **`AsyncResult`**, that is not understood by any other libraries.

> Users should generally prefer to use `concurrent.futures.ThreadPoolExecutor`, which has a simpler interface that was designed around threads from the start, and which returns `concurrent.futures.Future` instances that are compatible with many other libraries, including `asyncio`.

Programming guidelines

There are certain guidelines and idioms which should be adhered to when using `multiprocessing`.

All start methods

The following applies to all start methods.

Avoid shared state

> As far as possible one should try to avoid shifting large amounts of data between processes.
>
> It is probably best to stick to using queues or pipes for communication between processes rather than using the lower level synchronization primitives.

Picklability

> Ensure that the arguments to the methods of proxies are picklable.

Thread safety of proxies

> Do not use a proxy object from more than one thread unless you protect it with a lock.

(There is never a problem with different processes using the *same* proxy.)

Joining zombie processes

On POSIX when a process finishes but has not been joined it becomes a zombie. There should never be very many because each time a new process starts (or `active_children()` is called) all completed processes which have not yet been joined will be joined. Also calling a finished process's `Process.is_alive` will join the process. Even so it is probably good practice to explicitly join all the processes that you start.

Better to inherit than pickle/unpickle

When using the *spawn* or *forkserver* start methods many types from `multiprocessing` need to be picklable so that child processes can use them. However, one should generally avoid sending shared objects to other processes using pipes or queues. Instead you should arrange the program so that a process which needs access to a shared resource created elsewhere can inherit it from an ancestor process.

Avoid terminating processes

Using the `Process.terminate` method to stop a process is liable to cause any shared resources (such as locks, semaphores, pipes and queues) currently being used by the process to become broken or unavailable to other processes.

Therefore it is probably best to only consider using `Process.terminate` on processes which never use any shared resources.

Joining processes that use queues

Bear in mind that a process that has put items in a queue will wait before terminating until all the buffered items are fed by the “feeder” thread to the underlying pipe. (The child process can call the `Queue.cancel_join_thread` method of the queue to avoid this behaviour.)

This means that whenever you use a queue you need to make sure that all items which have been put on the queue will eventually be removed before the process is joined. Otherwise you cannot be sure that processes which have put items on the queue will terminate. Remember also that non-daemonic processes will be joined automatically.

An example which will deadlock is the following:

```
from multiprocessing import Process, Queue

def f(q):
    q.put('X' * 1000000)

if __name__ == '__main__':
    queue = Queue()
    p = Process(target=f, args=(queue,))
    p.start()
    p.join()                    # this deadlocks
    obj = queue.get()
```

A fix here would be to swap the last two lines (or simply remove the `p.join()` line).

Explicitly pass resources to child processes

On POSIX using the *fork* start method, a child process can make use of a shared resource created in a parent process using a global resource. However, it is better to pass the object as an argument to the constructor for the child process.

Apart from making the code (potentially) compatible with Windows and the other start methods this also ensures that as long as the child process is still alive the object will not be garbage collected in the parent process. This might be important if some resource is freed when the object is garbage collected in the parent process.

So for instance

```
from multiprocessing import Process, Lock

def f():
    ... do something using "lock" ...

if __name__ == '__main__':
    lock = Lock()
    for i in range(10):
        Process(target=f).start()
```

should be rewritten as

```
from multiprocessing import Process, Lock

def f(l):
    ... do something using "l" ...

if __name__ == '__main__':
    lock = Lock()
    for i in range(10):
        Process(target=f, args=(lock,)).start()
```

Beware of replacing `sys.stdin` with a “file like object”

`multiprocessing` originally unconditionally called:

```
os.close(sys.stdin.fileno())
```

in the **`multiprocessing.Process._bootstrap()`** method — this resulted in issues with processes-in-processes. This has been changed to:

```
sys.stdin.close()
sys.stdin = open(os.open(os.devnull, os.O_RDONLY),
closefd=False)
```

Which solves the fundamental issue of processes colliding with each other resulting in a bad file descriptor error, but introduces a potential danger to applications which replace **`sys.stdin()`** with a “file-like object” with output buffering. This danger is that if multiple processes call **`close()`** on this file-like object, it could result in the same data being flushed to the object multiple times, resulting in corruption.

If you write a file-like object and implement your own caching, you can make it fork-safe by storing the pid whenever you append to the cache, and discarding the cache when the pid changes. For example:

```
@property
def cache(self):
    pid = os.getpid()
    if pid != self._pid:
        self._pid = pid
        self._cache = []
    return self._cache
```

For more information, see bpo-5155 [https://bugs.python.org/issue?@action=redirect&bpo=5155], bpo-5313 [https://bugs.python.org/issue?@action=redirect&bpo=5313] and bpo-5331 [https://bugs.python.org/issue?@action=redirect&bpo=5331]

The *spawn* and *forkserver* start methods

There are a few extra restrictions which don’t apply to the *fork* start method.

More picklability

Ensure that all arguments to **`Process`** are picklable. Also, if you subclass `Process.__init__`, you must make sure that instances will be picklable when the **`Process.start`** method is called.

Global variables

Bear in mind that if code run in a child process tries to access a global variable, then the value it sees (if any) may not be the same as the value in the parent process at the time that **`Process.start`** was called.

However, global variables which are just module level constants cause no problems.

Safe importing of main module

Make sure that the main module can be safely imported by a new Python interpreter without causing unintended side effects (such as starting a new process).

For example, using the *spawn* or *forkserver* start method running the following module would fail with a **`RuntimeError`**:

```
from multiprocessing import Process

def foo():
    print('hello')

p = Process(target=foo)
p.start()
```

Instead one should protect the “entry point” of the program by using `if __name__ == '__main__':` as follows:

```
from multiprocessing import Process, freeze_support,
set_start_method

def foo():
    print('hello')

if __name__ == '__main__':
    freeze_support()
    set_start_method('spawn')
    p = Process(target=foo)
    p.start()
```

(The `freeze_support()` line can be omitted if the program will be run normally instead of frozen.)

This allows the newly spawned Python interpreter to safely import the module and then run the module’s `foo()` function.

Similar restrictions apply if a pool or manager is created in the main module.

Examples

Demonstration of how to create and use customized managers and proxies:

```
from multiprocessing import freeze_support
from multiprocessing.managers import BaseManager, BaseProxy
import operator

##

class Foo:
    def f(self):
        print('you called Foo.f()')
    def g(self):
        print('you called Foo.g()')
    def _h(self):
        print('you called Foo._h()')

# A simple generator function
def baz():
    for i in range(10):
        yield i*i

# Proxy type for generator objects
class GeneratorProxy(BaseProxy):
    _exposed_ = ['__next__']
    def __iter__(self):
        return self
    def __next__(self):
        return self._callmethod('__next__')

# Function to return the operator module
def get_operator_module():
    return operator

##

class MyManager(BaseManager):
    pass

# register the Foo class; make `f()` and `g()` accessible via
proxy
MyManager.register('Foo1', Foo)

# register the Foo class; make `g()` and `_h()` accessible via
proxy
MyManager.register('Foo2', Foo, exposed=('g', '_h'))

# register the generator function baz; use `GeneratorProxy` to
make proxies
MyManager.register('baz', baz, proxytype=GeneratorProxy)

# register get_operator_module(); make public functions
accessible via proxy
MyManager.register('operator', get_operator_module)
```

```
##

def test():
    manager = MyManager()
    manager.start()

    print('-' * 20)

    f1 = manager.Foo1()
    f1.f()
    f1.g()
    assert not hasattr(f1, '_h')
    assert sorted(f1._exposed_) == sorted(['f', 'g'])

    print('-' * 20)

    f2 = manager.Foo2()
    f2.g()
    f2._h()
    assert not hasattr(f2, 'f')
    assert sorted(f2._exposed_) == sorted(['g', '_h'])

    print('-' * 20)

    it = manager.baz()
    for i in it:
        print('<%d>' % i, end=' ')
    print()

    print('-' * 20)

    op = manager.operator()
    print('op.add(23, 45) =', op.add(23, 45))
    print('op.pow(2, 94) =', op.pow(2, 94))
    print('op._exposed_ =', op._exposed_)

##

if __name__ == '__main__':
    freeze_support()
    test()
```

Using `Pool`:

```
import multiprocessing
import time
import random
import sys

#
# Functions used by test code
#

def calculate(func, args):
    result = func(*args)
    return '%s says that %s%s = %s' % (
        multiprocessing.current_process().name,
        func.__name__, args, result
        )

def calculatestar(args):
    return calculate(*args)

def mul(a, b):
    time.sleep(0.5 * random.random())
    return a * b

def plus(a, b):
    time.sleep(0.5 * random.random())
    return a + b

def f(x):
    return 1.0 / (x - 5.0)

def pow3(x):
    return x ** 3

def noop(x):
    pass

#
# Test code
#

def test():
    PROCESSES = 4
    print('Creating pool with %d processes\n' % PROCESSES)
    with multiprocessing.Pool(PROCESSES) as pool:
        #
        # Tests
        #

        TASKS = [(mul, (i, 7)) for i in range(10)] + \
                [(plus, (i, 8)) for i in range(10)]

        results = [pool.apply_async(calculate, t) for t in
TASKS]
        imap_it = pool.imap(calculatestar, TASKS)
        imap_unordered_it = pool.imap_unordered(calculatestar,
TASKS)

        print('Ordered results using pool.apply_async():')
        for r in results:
            print('\t', r.get())
        print()

        print('Ordered results using pool.imap():')
        for x in imap_it:
            print('\t', x)
        print()

        print('Unordered results using pool.imap_unordered():')
        for x in imap_unordered_it:
            print('\t', x)
        print()

        print('Ordered results using pool.map() --- will block
till complete:')
        for x in pool.map(calculatestar, TASKS):
            print('\t', x)
        print()

        #
        # Test error handling
        #

        print('Testing error handling:')

        try:
            print(pool.apply(f, (5,)))
        except ZeroDivisionError:
            print('\tGot ZeroDivisionError as expected from 
pool.apply()')
        else:
            raise AssertionError('expected ZeroDivisionError')

        try:
            print(pool.map(f, list(range(10))))
        except ZeroDivisionError:
            print('\tGot ZeroDivisionError as expected from 
pool.map()')
        else:
            raise AssertionError('expected ZeroDivisionError')

        try:
            print(list(pool.imap(f, list(range(10)))))
        except ZeroDivisionError:
            print('\tGot ZeroDivisionError as expected from 
list(pool.imap())')
        else:
            raise AssertionError('expected ZeroDivisionError')

        it = pool.imap(f, list(range(10)))
        for i in range(10):
            try:
                x = next(it)
            except ZeroDivisionError:
                if i == 5:
                    pass
            except StopIteration:
                break
            else:
                if i == 5:
                    raise AssertionError('expected 
ZeroDivisionError')

        assert i == 9
        print('\tGot ZeroDivisionError as expected from 
IMapIterator.next()')
        print()

        #
        # Testing timeouts
        #

        print('Testing ApplyResult.get() with timeout:', end=' ')
        res = pool.apply_async(calculate, TASKS[0])
        while 1:
            sys.stdout.flush()
            try:
                sys.stdout.write('\n\t%s' % res.get(0.02))
                break
            except multiprocessing.TimeoutError:
                sys.stdout.write('.')
        print()
        print()

        print('Testing IMapIterator.next() with timeout:', end=' ')
        it = pool.imap(calculatestar, TASKS)
        while 1:
            sys.stdout.flush()
            try:
                sys.stdout.write('\n\t%s' % it.next(0.02))
            except StopIteration:
                break
            except multiprocessing.TimeoutError:
                sys.stdout.write('.')
        print()
        print()


if __name__ == '__main__':
    multiprocessing.freeze_support()
    test()
```

An example showing how to use queues to feed tasks to a collection of worker processes and collect the results:

```
import time
import random

from multiprocessing import Process, Queue, current_process,
freeze_support

#
# Function run by worker processes
#

def worker(input, output):
    for func, args in iter(input.get, 'STOP'):
        result = calculate(func, args)
        output.put(result)

#
# Function used to calculate result
#

def calculate(func, args):
    result = func(*args)
    return '%s says that %s%s = %s' % \
        (current_process().name, func.__name__, args, result)

#
# Functions referenced by tasks
#

def mul(a, b):
    time.sleep(0.5*random.random())
    return a * b

def plus(a, b):
    time.sleep(0.5*random.random())
    return a + b

#
#
#

def test():
    NUMBER_OF_PROCESSES = 4
    TASKS1 = [(mul, (i, 7)) for i in range(20)]
    TASKS2 = [(plus, (i, 8)) for i in range(10)]

    # Create queues
    task_queue = Queue()
    done_queue = Queue()

    # Submit tasks
    for task in TASKS1:
        task_queue.put(task)

    # Start worker processes
    for i in range(NUMBER_OF_PROCESSES):
        Process(target=worker, args=(task_queue,
done_queue)).start()

    # Get and print results
    print('Unordered results:')
    for i in range(len(TASKS1)):
        print('\t', done_queue.get())

    # Add more tasks using `put()`
    for task in TASKS2:
        task_queue.put(task)

    # Get and print some more results
    for i in range(len(TASKS2)):
        print('\t', done_queue.get())

    # Tell child processes to stop
    for i in range(NUMBER_OF_PROCESSES):
        task_queue.put('STOP')


if __name__ == '__main__':
    freeze_support()
    test()
```

`multiprocessing.shared_memory` — Shared memory for direct access across processes

Source code: Lib/multiprocessing/shared_memory.py [https://github.com/python/cpython/tree/3.14/Lib/multiprocessing/shared_memory.py]

Added in version 3.8.

This module provides a class, **`SharedMemory`**, for the allocation and management of shared memory to be accessed by one or more processes on a multicore or symmetric multiprocessor (SMP) machine. To assist with the life-cycle management of shared memory especially across distinct processes, a **`BaseManager`** subclass, **`SharedMemoryManager`**, is also provided in the **`multiprocessing.managers`** module.

In this module, shared memory refers to “POSIX style” shared memory blocks (though is not necessarily implemented explicitly as such) and does not refer to “distributed shared memory”. This style of shared memory permits distinct processes to potentially read and write to a common (or shared) region of volatile memory. Processes are conventionally limited to only have access to their own process memory space but shared memory permits the sharing of data between processes, avoiding the need to instead send messages between processes containing that data. Sharing data directly via memory can provide significant performance benefits compared to sharing data via disk or socket or other communications requiring the serialization/deserialization and copying of data.

class multiprocessing.shared_memory.SharedMemory(*name=None*, *create=False*, *size=0*, ***, *track=True*)

Create an instance of the **`SharedMemory`** class for either creating a new shared memory block or attaching to an existing shared memory block. Each shared memory block is assigned a unique name. In this way, one process can create a shared memory block with a particular name and a different process can attach to that same shared memory block using that same name.

As a resource for sharing data across processes, shared memory blocks may outlive the original process that created them. When one process no longer needs access to a shared memory block that might still be needed by other processes, the `close()` method should be called. When a shared memory block is no longer needed by any process, the `unlink()` method should be called to ensure proper cleanup.

Parameters::

- **name** (*str* | *None*) – The unique name for the requested shared memory, specified as a string. When creating a new shared memory block, if `None` (the default) is supplied for the name, a nove name will be generated.
- **create** (*bool*) – Control whether a new shared memory block is created (`True`) o an existing shared memory block is attached (`False`).
- **size** (*int*) – The requested number of bytes when creating a new shared memory block. Because some platforms choose to allocate chunks of memory based upon that platform’s memory page size, the exact size of the shared memory block may be larger or equal to the size requested. When attaching to an existing shared memory block, the *size* parameter is ignored.
- **track** (*bool*) – When `True`, register the shared memory block with a resource tracker process on platforms where the OS does not do this automatically. The resource tracker ensures proper cleanup of the shared memory even if all other processes with access to the memory exit without doing so. Python processes created from a common ancestor using **`multiprocessing`** facilities share a single resource tracker process, and the lifetime of shared memory segments is handled automatically among these processes. Python processes created in any other way will receive their own resource tracker when accessing shared memory with *track* enabled. This will

memory with track enabled. This will cause the shared memory to be deleted by the resource tracker of the first process that terminates. To avoid this issue, users of `subprocess` or standalone Python processes should set *track* to `False` when there is already another process in place that does the bookkeeping. *track* is ignored on Windows, which has its own tracking and automatically deletes shared memory when all handles to it have been closed.

Changed in version 3.13: Added the *track* parameter.

close()

Close the file descriptor/handle to the shared memory from this instance. `close()` should be called once access to the shared memory block from this instance is no longer needed. Depending on operating system, the underlying memory may or may not be freed even if all handles to it have been closed. To ensure proper cleanup, use the `unlink()` method.

unlink()

Delete the underlying shared memory block. This should be called only once per shared memory block regardless of the number of handles to it, even in other processes. `unlink()` and `close()` can be called in any order, but trying to access data inside a shared memory block after `unlink()` may result in memory access errors, depending on platform.

This method has no effect on Windows, where the only way to delete a shared memory block is to close all handles.

buf

A memoryview of contents of the shared memory block.

name

Read-only access to the unique name of the shared memory block.

size

Read-only access to size in bytes of the shared memory block.

The following example demonstrates low-level use of `SharedMemory` instances:

```
>>> from multiprocessing import shared_memory
>>> shm_a = shared_memory.SharedMemory(create=True, size=10)
>>> type(shm_a.buf)
<class 'memoryview'>
>>> buffer = shm_a.buf
>>> len(buffer)
10
>>> buffer[:4] = bytearray([22, 33, 44, 55])  # Modify multiple
at once
>>> buffer[4] = 100                           # Modify single
byte at a time
>>> # Attach to an existing shared memory block
>>> shm_b = shared_memory.SharedMemory(shm_a.name)
>>> import array
>>> array.array('b', shm_b.buf[:5])  # Copy the data into a new
array.array
array('b', [22, 33, 44, 55, 100])
>>> shm_b.buf[:5] = b'howdy'  # Modify via shm_b using bytes
>>> bytes(shm_a.buf[:5])      # Access via shm_a
b'howdy'
>>> shm_b.close()   # Close each SharedMemory instance
>>> shm_a.close()
>>> shm_a.unlink()  # Call unlink only once to release the
shared memory
```

The following example demonstrates a practical use of the `SharedMemory` class with NumPy arrays [https://numpy.org/], accessing the same **`numpy.ndarray`** from two distinct Python shells:

```
>>> # In the first Python interactive shell
>>> import numpy as np
>>> a = np.array([1, 1, 2, 3, 5, 8])  # Start with an existing
NumPy array
>>> from multiprocessing import shared_memory
>>> shm = shared_memory.SharedMemory(create=True, size=a.nbytes)
>>> # Now create a NumPy array backed by shared memory
>>> b = np.ndarray(a.shape, dtype=a.dtype, buffer=shm.buf)
>>> b[:] = a[:]  # Copy the original data into shared memory
>>> b
array([1, 1, 2, 3, 5, 8])
>>> type(b)
<class 'numpy.ndarray'>
```

```
>>> type(a)
<class 'numpy.ndarray'>
>>> shm.name  # We did not specify a name so one was chosen for
us
'psm_21467_46075'

>>> # In either the same shell or a new Python shell on the same
machine
>>> import numpy as np
>>> from multiprocessing import shared_memory
>>> # Attach to the existing shared memory block
>>> existing_shm =
shared_memory.SharedMemory(name='psm_21467_46075')
>>> # Note that a.shape is (6,) and a.dtype is np.int64 in this
example
>>> c = np.ndarray((6,), dtype=np.int64,
buffer=existing_shm.buf)
>>> c
array([1, 1, 2, 3, 5, 8])
>>> c[-1] = 888
>>> c
array([  1,   1,   2,   3,   5, 888])

>>> # Back in the first Python interactive shell, b reflects
this change
>>> b
array([  1,   1,   2,   3,   5, 888])

>>> # Clean up from within the second Python shell
>>> del c  # Unnecessary; merely emphasizing the array is no
longer used
>>> existing_shm.close()

>>> # Clean up from within the first Python shell
>>> del b  # Unnecessary; merely emphasizing the array is no
longer used
>>> shm.close()
>>> shm.unlink()  # Free and release the shared memory block at
the very end
```

class multiprocessing.managers.SharedMemoryManager([*address*[

, *authkey*]])

A subclass of `multiprocessing.managers.BaseManager` which can be used for the management of shared memory blocks across processes.

A call to `start()` on a **`SharedMemoryManager`** instance causes a new process to be started. This new process's sole purpose is to manage the life cycle of all shared memory blocks created through it. To trigger the release of all shared memory blocks managed by that process, call `shutdown()` on the instance. This triggers a `unlink()` call on all of the `SharedMemory` objects managed by that process and then stops the process itself. By creating **`SharedMemory`** instances through a **`SharedMemoryManager`**, we avoid the need to manually track and trigger the freeing of shared memory resources.

This class provides methods for creating and returning `SharedMemory` instances and for creating a list-like object (`ShareableList`) backed by shared memory.

Refer to `BaseManager` for a description of the inherited *address* and *authkey* optional input arguments and how they may be used to connect to an existing **`SharedMemoryManager`** service from other processes.

SharedMemory(*size*)

Create and return a new `SharedMemory` object with the specified *size* in bytes.

ShareableList(*sequence*)

Create and return a new `ShareableList` object, initialized by the values from the input *sequence*.

The following example demonstrates the basic mechanisms of a `SharedMemoryManager`:

```
>>> from multiprocessing.managers import SharedMemoryManager
>>> smm = SharedMemoryManager()
>>> smm.start()  # Start the process that manages the shared
memory blocks
>>> sl = smm.ShareableList(range(4))
>>> sl
ShareableList([0, 1, 2, 3], name='psm_6572_7512')
>>> raw_shm = smm.SharedMemory(size=128)
>>> another_sl = smm.ShareableList('alpha')
>>> another_sl
ShareableList(['a', 'l', 'p', 'h', 'a'], name='psm_6572_12221')
>>> smm.shutdown()  # Calls unlink() on sl, raw_shm, and
another_sl
```

The following example depicts a potentially more convenient pattern for using `SharedMemoryManager` objects via the `with` statement to ensure that all shared memory blocks are released after they are no longer needed:

```
>>> with SharedMemoryManager() as smm:
...     sl = smm.ShareableList(range(2000))
...     # Divide the work among two processes, storing partial
results in sl
...     p1 = Process(target=do_work, args=(sl, 0, 1000))
...     p2 = Process(target=do_work, args=(sl, 1000, 2000))
...     p1.start()
...     p2.start()  # A multiprocessing.Pool might be more
efficient
...     p1.join()
...     p2.join()   # Wait for all work to complete in both
processes
...     total_result = sum(sl)  # Consolidate the partial
results now in sl
```

When using a `SharedMemoryManager` in a `with` statement, the shared memory blocks created using that manager are all released when the `with` statement’s code block finishes execution.

class
multiprocessing.shared_memory.ShareableList(*sequence=No*

ne, ***, *name=None*)

Provide a mutable list-like object where all values stored within are stored in a shared memory block. This constrains storable values to the following built-in data types:

- `int` (signed 64-bit)
- `float`
- `bool`
- `str` (less than 10M bytes each when encoded as UTF-8)
- `bytes` (less than 10M bytes each)
- `None`

It also notably differs from the built-in `list` type in that these lists can not change their overall length (i.e. no **`append()`**, **`insert()`**, etc.) and do not support the dynamic creation of new **`ShareableList`** instances via slicing.

sequence is used in populating a new **`ShareableList`** full of values. Set to `None` to instead attach to an already existing **`ShareableList`** by its unique shared memory name.

name is the unique name for the requested shared memory, as described in the definition for `SharedMemory`. When attaching to an existing **`ShareableList`**, specify its shared memory block's unique name while leaving *sequence* set to `None`.

Note

A known issue exists for `bytes` and `str` values. If they end with `\x00` nul bytes or characters, those may be *silently stripped* when fetching them by index from the **`ShareableList`**. This `.rstrip(b'\x00')` behavior is considered a bug and may go away in the future. See gh-106939 [https://github.com/python/cpython/issues/106939].

For applications where rstripping of trailing nulls is a problem, work around it by always unconditionally appending an extra non-0 byte to the end of such values when storing and unconditionally removing it when fetching:

```
>>> from multiprocessing import shared_memory
>>> nul_bug_demo = shared_memory.ShareableList(['?\x00',
b'\x03\x02\x01\x00\x00\x00'])
>>> nul_bug_demo[0]
'?'
>>> nul_bug_demo[1]
b'\x03\x02\x01'
>>> nul_bug_demo.shm.unlink()
>>> padded = shared_memory.ShareableList(['?\x00\x07',
b'\x03\x02\x01\x00\x00\x00\x07'])
>>> padded[0][:-1]
'?\x00'
>>> padded[1][:-1]
b'\x03\x02\x01\x00\x00\x00'
>>> padded.shm.unlink()
```

count(*value*)

Return the number of occurrences of *value*.

index(*value*)

Return first index position of *value*. Raise `ValueError` if *value* is not present.

format

Read-only attribute containing the `struct` packing format used by all currently stored values.

shm

The `SharedMemory` instance where the values are stored.

The following example demonstrates basic use of a `ShareableList` instance:

```
>>> from multiprocessing import shared_memory
>>> a = shared_memory.ShareableList(['howdy', b'HoWdY',
-273.154, 100, None, True, 42])
>>> [ type(entry) for entry in a ]
[<class 'str'>, <class 'bytes'>, <class 'float'>, <class 'int'>,
<class 'NoneType'>, <class 'bool'>, <class 'int'>]
>>> a[2]
-273.154
>>> a[2] = -78.5
>>> a[2]
-78.5
>>> a[2] = 'dry ice'  # Changing data types is supported as well
>>> a[2]
'dry ice'
>>> a[2] = 'larger than previously allocated storage space'
Traceback (most recent call last):
  ...
ValueError: exceeds available storage for existing str
>>> a[2]
'dry ice'
>>> len(a)
7
>>> a.index(42)
6
>>> a.count(b'howdy')
0
>>> a.count(b'HoWdY')
1
>>> a.shm.close()
>>> a.shm.unlink()
>>> del a  # Use of a ShareableList after call to unlink() is
unsupported
```

The following example depicts how one, two, or many processes may access the same `ShareableList` by supplying the name of the shared memory block behind it:

```
>>> b = shared_memory.ShareableList(range(5))         # In a
first process
>>> c = shared_memory.ShareableList(name=b.shm.name)  # In a
second process
>>> c
ShareableList([0, 1, 2, 3, 4], name='...')
```

```
>>> c[-1] = -999
>>> b[-1]
-999
>>> b.shm.close()
>>> c.shm.close()
>>> c.shm.unlink()
```

The following examples demonstrates that `ShareableList` (and underlying `SharedMemory`) objects can be pickled and unpickled if needed. Note, that it will still be the same shared object. This happens, because the deserialized object has the same unique name and is just attached to an existing object with the same name (if the object is still alive):

```
>>> import pickle
>>> from multiprocessing import shared_memory
>>> sl = shared_memory.ShareableList(range(10))
>>> list(sl)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

>>> deserialized_sl = pickle.loads(pickle.dumps(sl))
>>> list(deserialized_sl)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

>>> sl[0] = -1
>>> deserialized_sl[1] = -2
>>> list(sl)
[-1, -2, 2, 3, 4, 5, 6, 7, 8, 9]
>>> list(deserialized_sl)
[-1, -2, 2, 3, 4, 5, 6, 7, 8, 9]

>>> sl.shm.close()
>>> sl.shm.unlink()
```

The `concurrent` package

This package contains the following modules:

- `concurrent.futures` – Launching parallel tasks
- `concurrent.interpreters` – Multiple interpreters in the same process

`concurrent.futures` — Launching parallel tasks

Added in version 3.2.

Source code: Lib/concurrent/futures/thread.py [https://github.com/python/cpython/tree/3.14/Lib/concurrent/futures/thread.py], Lib/concurrent/futures/process.py [https://github.com/python/cpython/tree/3.14/Lib/concurrent/futures/process.py], and Lib/concurrent/futures/interpreter.py [https://github.com/python/cpython/tree/3.14/Lib/concurrent/futures/interpreter.py]

The **`concurrent.futures`** module provides a high-level interface for asynchronously executing callables.

The asynchronous execution can be performed with threads, using **`ThreadPoolExecutor`** or **`InterpreterPoolExecutor`**, or separate processes, using **`ProcessPoolExecutor`**. Each implements the same interface, which is defined by the abstract **`Executor`** class.

`concurrent.futures.Future` must not be confused with **`asyncio.Future`**, which is designed for use with **`asyncio`** tasks and coroutines. See the asyncio's Future documentation for a detailed comparison of the two.

Availability: not WASI.

This module does not work or is not available on WebAssembly. See WebAssembly platforms for more information.

Executor Objects

class concurrent.futures.Executor

An abstract class that provides methods to execute calls asynchronously. It should not be used directly, but through its concrete subclasses.

submit(*fn*, */*, **args*, ***kwargs*)

Schedules the callable, *fn*, to be executed as `fn(*args, **kwargs)` and returns a **`Future`** object representing the execution of the callable.

```
with ThreadPoolExecutor(max_workers=1) as executor:
    future = executor.submit(pow, 323, 1235)
    print(future.result())
```

map(*fn*, **iterables*, *timeout=None*, *chunksize=1*, *buffersize=None*)

Similar to **`map(fn, *iterables)`** except:

- The *iterables* are collected immediately rather than lazily, unless a *buffersize* is specified to limit the number of submitted tasks whose results have not yet been yielded. If the buffer is full, iteration over the *iterables* pauses until a result is yielded from the buffer.
- *fn* is executed asynchronously and several calls to *fn* may be made concurrently.

The returned iterator raises a **`TimeoutError`** if **`__next__()`** is called and the result isn't available after *timeout* seconds from the original call to **`Executor.map()`**. *timeout* can be an int or a float. If *timeout* is not specified or `None`, there is no limit to the wait time.

If a *fn* call raises an exception, then that exception will be raised when its value is retrieved from the iterator.

When using `ProcessPoolExecutor`, this method chops *iterables* into a number of chunks which it submits to the pool as separate tasks. The (approximate) size of these chunks can be specified by setting *chunksize* to a positive integer. For very long iterables, using a large value for *chunksize* can significantly improve performance compared to the default size of 1. With `ThreadPoolExecutor` and `InterpreterPoolExecutor`, *chunksize* has no effect.

Changed in version 3.5: Added the *chunksize* parameter.

Changed in version 3.14: Added the *buffersize* parameter.

shutdown(*wait=True*, *, *cancel_futures=False*)

Signal the executor that it should free any resources that it is using when the currently pending futures are done executing. Calls to `Executor.submit()` and `Executor.map()` made after shutdown will raise `RuntimeError`.

If *wait* is `True` then this method will not return until all the pending futures are done executing and the resources associated with the executor have been freed. If *wait* is `False` then this method will return immediately and the resources associated with the executor will be freed when all pending futures are done executing. Regardless of the value of *wait*, the entire Python program will not exit until all pending futures are done executing.

If *cancel_futures* is `True`, this method will cancel all pending futures that the executor has not started running. Any futures that are completed or running won’t be cancelled, regardless of the value of *cancel_futures*.

If both *cancel_futures* and *wait* are `True`, all futures that the executor has started running will be completed prior to this method returning. The remaining futures are cancelled.

You can avoid having to call this method explicitly if you use the executor as a context manager via the **`with`** statement, which will shutdown the **`Executor`** (waiting as if **`Executor.shutdown()`** were called with *wait* set to `True`):

```
import shutil
with ThreadPoolExecutor(max_workers=4) as e:
    e.submit(shutil.copy, 'src1.txt', 'dest1.txt')
    e.submit(shutil.copy, 'src2.txt', 'dest2.txt')
    e.submit(shutil.copy, 'src3.txt', 'dest3.txt')
    e.submit(shutil.copy, 'src4.txt', 'dest4.txt')
```

Changed in version 3.9: Added *cancel_futures*.

ThreadPoolExecutor

`ThreadPoolExecutor` is an **`Executor`** subclass that uses a pool of threads to execute calls asynchronously.

Deadlocks can occur when the callable associated with a **`Future`** waits on the results of another **`Future`**. For example:

```
import time
def wait_on_b():
    time.sleep(5)
    print(b.result())  # b will never complete because it is
```

```
waiting on a.
    return 5

def wait_on_a():
    time.sleep(5)
    print(a.result())  # a will never complete because it is
waiting on b.
    return 6


executor = ThreadPoolExecutor(max_workers=2)
a = executor.submit(wait_on_b)
b = executor.submit(wait_on_a)
```

And:

```
def wait_on_future():
    f = executor.submit(pow, 5, 2)
    # This will never complete because there is only one worker
thread and
    # it is executing this function.
    print(f.result())

executor = ThreadPoolExecutor(max_workers=1)
executor.submit(wait_on_future)
```

class concurrent.futures.ThreadPoolExecutor(*max_workers=None*, *thread_name_prefix=''*, *initializer=None*, *initargs=()*)

An `Executor` subclass that uses a pool of at most *max_workers* threads to execute calls asynchronously.

All threads enqueued to `ThreadPoolExecutor` will be joined before the interpreter can exit. Note that the exit handler which does this is executed *before* any exit handlers added using `atexit`. This means exceptions in the main thread must be caught and handled in order to signal threads to exit gracefully. For this reason, it is recommended that `ThreadPoolExecutor` not be used for long-running tasks.

initializer is an optional callable that is called at the start of each worker thread; *initargs* is a tuple of arguments passed to the initializer. Should *initializer* raise an exception, all currently pending jobs will raise a **`BrokenThreadPool`**, as well as any attempt to submit more jobs to the pool.

Changed in version 3.5: If *max_workers* is `None` or not given, it will default to the number of processors on the machine, multiplied by `5`, assuming that **`ThreadPoolExecutor`** is often used to overlap I/O instead of CPU work and the number of workers should be higher than the number of workers for **`ProcessPoolExecutor`**.

Changed in version 3.6: Added the *thread_name_prefix* parameter to allow users to control the **`threading.Thread`** names for worker threads created by the pool for easier debugging.

Changed in version 3.7: Added the *initializer* and *initargs* arguments.

Changed in version 3.8: Default value of *max_workers* is changed to `min(32, os.cpu_count() + 4)`. This default value preserves at least 5 workers for I/O bound tasks. It utilizes at most 32 CPU cores for CPU bound tasks which release the GIL. And it avoids using very large resources implicitly on many-core machines.

ThreadPoolExecutor now reuses idle worker threads before starting *max_workers* worker threads too.

Changed in version 3.13: Default value of *max_workers* is changed to `min(32, (os.process_cpu_count() or 1) + 4)`.

ThreadPoolExecutor Example

```
import concurrent.futures
import urllib.request

URLS = ['http://www.foxnews.com/',
        'http://www.cnn.com/',
        'http://europe.wsj.com/',
        'http://www.bbc.co.uk/',
        'http://nonexistent-subdomain.python.org/']

# Retrieve a single page and report the URL and contents
def load_url(url, timeout):
    with urllib.request.urlopen(url, timeout=timeout) as conn:
        return conn.read()

# We can use a with statement to ensure threads are cleaned up
promptly
with concurrent.futures.ThreadPoolExecutor(max_workers=5) as
executor:
    # Start the load operations and mark each future with its
URL
    future_to_url = {executor.submit(load_url, url, 60): url
for url in URLS}
    for future in
concurrent.futures.as_completed(future_to_url):
        url = future_to_url[future]
        try:
            data = future.result()
        except Exception as exc:
            print('%r generated an exception: %s' % (url, exc))
        else:
            print('%r page is %d bytes' % (url, len(data)))
```

InterpreterPoolExecutor

Added in version 3.14.

The `InterpreterPoolExecutor` class uses a pool of interpreters to execute calls asynchronously. It is a `ThreadPoolExecutor` subclass, which means each worker is running in its own thread. The difference here is that each worker has its own interpreter, and runs each task using that interpreter.

The biggest benefit to using interpreters instead of only threads is true multi-core parallelism. Each interpreter has its own Global Interpreter Lock, so code running in one interpreter can run on one CPU core, while code in another interpreter runs unblocked on a different core.

The tradeoff is that writing concurrent code for use with multiple interpreters can take extra effort. However, this is because it forces you to be deliberate about how and when interpreters interact, and to be explicit about what data is shared between interpreters. This results in several benefits that help balance the extra effort, including true multi-core parallelism, For example, code written this way can make it easier to reason about concurrency. Another major benefit is that you don’t have to deal with several of the big pain points of using threads, like race conditions.

Each worker’s interpreter is isolated from all the other interpreters. “Isolated” means each interpreter has its own runtime state and operates completely independently. For example, if you redirect `sys.stdout` in one interpreter, it will not be automatically redirected to any other interpreter. If you import a module in one interpreter, it is not automatically imported in any other. You would need to import the module separately in interpreter where you need it. In fact, each module imported in an interpreter is a completely separate object from the same module in a different interpreter, including `sys`, `builtins`, and even `__main__`.

Isolation means a mutable object, or other data, cannot be used by more than one interpreter at the same time. That effectively means interpreters cannot actually share such objects or data. Instead, each interpreter must have its own copy, and you will have to synchronize any changes between the copies manually. Immutable objects and data, like the builtin singletons, strings, and tuples of immutable objects, don't have these limitations.

Communicating and synchronizing between interpreters is most effectively done using dedicated tools, like those proposed in **PEP 734** [https://peps.python.org/pep-0734/]. One less efficient alternative is to serialize with `pickle` and then send the bytes over a shared `socket` or `pipe`.

class concurrent.futures.InterpreterPoolExecutor(*max_workers=None*, *thread_name_prefix=''*, *initializer=None*, *initargs=()*)

A `ThreadPoolExecutor` subclass that executes calls asynchronously using a pool of at most *max_workers* threads. Each thread runs tasks in its own interpreter. The worker interpreters are isolated from each other, which means each has its own runtime state and that they can't share any mutable objects or other data. Each interpreter has its own Global Interpreter Lock, which means code run with this executor has true multi-core parallelism.

The optional *initializer* and *initargs* arguments have the same meaning as for `ThreadPoolExecutor`: the initializer is run when each worker is created, though in this case it is run in the worker's interpreter. The executor serializes the *initializer* and *initargs* using `pickle` when sending them to the worker's interpreter.

Note

> The executor may replace uncaught exceptions from *initializer* with **`ExecutionFailed`**.

> Other caveats from parent **`ThreadPoolExecutor`** apply here.

`submit()` and **`map()`** work like normal, except the worker serializes the callable and arguments using **`pickle`** when sending them to its interpreter. The worker likewise serializes the return value when sending it back.

When a worker’s current task raises an uncaught exception, the worker always tries to preserve the exception as-is. If that is successful then it also sets the `__cause__` to a corresponding **`ExecutionFailed`** instance, which contains a summary of the original exception. In the uncommon case that the worker is not able to preserve the original as-is then it directly preserves the corresponding **`ExecutionFailed`** instance instead.

ProcessPoolExecutor

The **`ProcessPoolExecutor`** class is an **`Executor`** subclass that uses a pool of processes to execute calls asynchronously. **`ProcessPoolExecutor`** uses the **`multiprocessing`** module, which allows it to side-step the Global Interpreter Lock but also means that only picklable objects can be executed and returned.

The `__main__` module must be importable by worker subprocesses. This means that **`ProcessPoolExecutor`** will not work in the interactive interpreter.

Calling **`Executor`** or **`Future`** methods from a callable submitted to a **`ProcessPoolExecutor`** will result in deadlock.

Note that the restrictions on functions and arguments needing to picklable as per `multiprocessing.Process` apply when using `submit()` and `map()` on a `ProcessPoolExecutor`. A function defined in a REPL or a lambda should not be expected to work.

class concurrent.futures.ProcessPoolExecutor(*max_workers=None*, *mp_context=None*, *initializer=None*, *initargs=()*, *max_tasks_per_child=None*)

An `Executor` subclass that executes calls asynchronously using a pool of at most *max_workers* processes. If *max_workers* is `None` or not given, it will default to `os.process_cpu_count()`. If *max_workers* is less than or equal to `0`, then a `ValueError` will be raised. On Windows, *max_workers* must be less than or equal to `61`. If it is not then `ValueError` will be raised. If *max_workers* is `None`, then the default chosen will be at most `61`, even if more processors are available. *mp_context* can be a `multiprocessing` context or `None`. It will be used to launch the workers. If *mp_context* is `None` or not given, the default `multiprocessing` context is used. See Contexts and start methods.

initializer is an optional callable that is called at the start of each worker process; *initargs* is a tuple of arguments passed to the initializer. Should *initializer* raise an exception, all currently pending jobs will raise a `BrokenProcessPool`, as well as any attempt to submit more jobs to the pool.

max_tasks_per_child is an optional argument that specifies the maximum number of tasks a single process can execute before it will exit and be replaced with a fresh worker process. By default *max_tasks_per_child* is `None` which means worker processes will live as long as the pool. When a max is specified, the “spawn” multiprocessing start method will be used by default in absence of a *mp_context* parameter. This feature is incompatible with the “fork” start method.

Note

Bugs have been reported when using the *max_tasks_per_child* feature that can result in the `ProcessPoolExecutor` hanging in some circumstances. Follow its eventual resolution in gh-115634 [https://github.com/python/cpython/issues/115634].

Changed in version 3.3: When one of the worker processes terminates abruptly, a `BrokenProcessPool` error is now raised. Previously, behaviour was undefined but operations on the executor or its futures would often freeze or deadlock.

Changed in version 3.7: The *mp_context* argument was added to allow users to control the start_method for worker processes created by the pool.

Added the *initializer* and *initargs* arguments.

Changed in version 3.11: The *max_tasks_per_child* argument was added to allow users to control the lifetime of workers in the pool.

Changed in version 3.12: On POSIX systems, if your application has multiple threads and the `multiprocessing` context uses the `"fork"` start method: The `os.fork()` function called internally to spawn workers may raise a `DeprecationWarning`. Pass a *mp_context* configured to use a different start method. See the `os.fork()` documentation for further explanation.

Changed in version 3.13: max_workers uses `os.process_cpu_count()` by default, instead of `os.cpu_count()`.

Changed in version 3.14: The default process start method (see Contexts and start methods) changed away from *fork*. If you require the *fork* start method for `ProcessPoolExecutor` you must explicitly pass `mp_context=multiprocessing.get_context("fork")`.

terminate_workers()

Attempt to terminate all living worker processes immediately by calling `Process.terminate` on each of them. Internally, it will also call `Executor.shutdown()` to ensure that all other resources associated with the executor are freed.

After calling this method the caller should no longer submit tasks to the executor.

Added in version 3.14.

kill_workers()

Attempt to kill all living worker processes immediately by calling `Process.kill` on each of them. Internally, it will also call `Executor.shutdown()` to ensure that all other resources associated with the executor are freed.

After calling this method the caller should no longer submit tasks to the executor.

Added in version 3.14.

ProcessPoolExecutor Example

```
import concurrent.futures
import math
```

```
PRIMES = [
    112272535095293,
    112582705942171,
    112272535095293,
    115280095190773,
    115797848077099,
    1099726899285419]

def is_prime(n):
    if n < 2:
        return False
    if n == 2:
        return True
    if n % 2 == 0:
        return False

    sqrt_n = int(math.floor(math.sqrt(n)))
    for i in range(3, sqrt_n + 1, 2):
        if n % i == 0:
            return False
    return True

def main():
    with concurrent.futures.ProcessPoolExecutor() as executor:
        for number, prime in zip(PRIMES, executor.map(is_prime,
PRIMES)):
            print('%d is prime: %s' % (number, prime))

if __name__ == '__main__':
    main()
```

Future Objects

The `Future` class encapsulates the asynchronous execution of a callable. `Future` instances are created by `Executor.submit()`.

class concurrent.futures.Future

Encapsulates the asynchronous execution of a callable. `Future` instances are created by `Executor.submit()` and should not be created directly except for testing.

cancel()

Attempt to cancel the call. If the call is currently being executed or finished running and cannot be cancelled then the method will return `False`, otherwise the call will be cancelled and the method will return `True`.

cancelled()

Return `True` if the call was successfully cancelled.

running()

Return `True` if the call is currently being executed and cannot be cancelled.

done()

Return `True` if the call was successfully cancelled or finished running.

result(*timeout=None*)

Return the value returned by the call. If the call hasn't yet completed then this method will wait up to *timeout* seconds. If the call hasn't completed in *timeout* seconds, then a `TimeoutError` will be raised. *timeout* can be an int or float. If *timeout* is not specified or `None`, there is no limit to the wait time.

If the future is cancelled before completing then `CancelledError` will be raised.

If the call raised an exception, this method will raise the same exception.

exception(*timeout=None*)

Return the exception raised by the call. If the call hasn't yet completed then this method will wait up to *timeout* seconds. If the call hasn't completed in *timeout* seconds, then a `TimeoutError` will be raised. *timeout* can be an int or float. If *timeout* is not specified or `None`, there is no limit to the wait time.

If the future is cancelled before completing then `CancelledError` will be raised.

If the call completed without raising, `None` is returned.

add_done_callback(*fn*)

Attaches the callable *fn* to the future. *fn* will be called, with the future as its only argument, when the future is cancelled or finishes running.

Added callables are called in the order that they were added and are always called in a thread belonging to the process that added them. If the callable raises an `Exception` subclass, it will be logged and ignored. If the callable raises a `BaseException` subclass, the behavior is undefined.

If the future has already completed or been cancelled, *fn* will be called immediately.

The following `Future` methods are meant for use in unit tests and `Executor` implementations.

set_running_or_notify_cancel()

This method should only be called by `Executor` implementations before executing the work associated with the `Future` and by unit tests.

If the method returns `False` then the `Future` was cancelled, i.e. `Future.cancel()` was called and returned `True`. Any threads waiting on the `Future` completing (i.e. through `as_completed()` or `wait()`) will be woken up.

If the method returns `True` then the `Future` was not cancelled and has been put in the running state, i.e. calls to `Future.running()` will return `True`.

This method can only be called once and cannot be called after `Future.set_result()` or `Future.set_exception()` have been called.

set_result(*result*)

Sets the result of the work associated with the `Future` to *result*.

This method should only be used by `Executor` implementations and unit tests.

Changed in version 3.8: This method raises `concurrent.futures.InvalidStateError` if the `Future` is already done.

set_exception(*exception*)

Sets the result of the work associated with the `Future` to the `Exception` *exception*.

This method should only be used by `Executor` implementations and unit tests.

Changed in version 3.8: This method raises `concurrent.futures.InvalidStateError` if the `Future` is already done.

Module Functions

concurrent.futures.wait(*fs*, *timeout=None*, *return_when=ALL_COMPLETED*)

Wait for the `Future` instances (possibly created by different `Executor` instances) given by *fs* to complete.

Duplicate futures given to *fs* are removed and will be returned only once. Returns a named 2-tuple of sets. The first set, named `done`, contains the futures that completed (finished or cancelled futures) before the wait completed. The second set, named `not_done`, contains the futures that did not complete (pending or running futures).

timeout can be used to control the maximum number of seconds to wait before returning. *timeout* can be an int or float. If *timeout* is not specified or `None`, there is no limit to the wait time.

return_when indicates when this function should return. It must be one of the following constants:

Constant	Description
concurrent.futures.FIRST_COMPLETED	The function will return when any future finishes or is cancelled.
concurrent.futures.FIRST_EXCEPTION	The function will return when any future finishes by raising an exception. If no future raises an exception then it is equivalent to `ALL_COMPLETED`.
concurrent.futures.ALL_COMPLETED	The function will return when all futures finish or are cancelled.

Constant	Description

concurrent.futures.as_completed(*fs*, *timeout=None*)

Returns an iterator over the `Future` instances (possibly created by different `Executor` instances) given by *fs* that yields futures as they complete (finished or cancelled futures). Any futures given by *fs* that are duplicated will be returned once. Any futures that completed before `as_completed()` is called will be yielded first. The returned iterator raises a `TimeoutError` if `__next__()` is called and the result isn't available after *timeout* seconds from the original call to `as_completed()`. *timeout* can be an int or float. If *timeout* is not specified or `None`, there is no limit to the wait time.

See also

PEP 3148 [https://peeps.python.org/pep-3148/] – futures - execute computations asynchronously

The proposal which described this feature for inclusion in the Python standard library.

Exception classes

exception concurrent.futures.CancelledError

Raised when a future is cancelled.

exception concurrent.futures.TimeoutError

A deprecated alias of `TimeoutError`, raised when a future operation exceeds the given timeout.

Changed in version 3.11: This class was made an alias of `TimeoutError`.

exception concurrent.futures.BrokenExecutor

Derived from `RuntimeError`, this exception class is raised when an executor is broken for some reason, and cannot be used to submit or execute new tasks.

Added in version 3.7.

exception concurrent.futures.InvalidStateError

Raised when an operation is performed on a future that is not allowed in the current state.

Added in version 3.8.

exception concurrent.futures.thread.BrokenThreadPool

Derived from `BrokenExecutor`, this exception class is raised when one of the workers of a `ThreadPoolExecutor` has failed initializing.

Added in version 3.7.

exception concurrent.futures.interpreter.BrokenInterpreterPool

Derived from `BrokenThreadPool`, this exception class is raised when one of the workers of a `InterpreterPoolExecutor` has failed initializing.

Added in version 3.14.

exception concurrent.futures.process.BrokenProcessPool

Derived from `BrokenExecutor` (formerly `RuntimeError`), this exception class is raised when one of the workers of a `ProcessPoolExecutor` has terminated in a non-clean fashion (for example, if it was killed from the outside).

Added in version 3.3.

`concurrent.interpreters` — Multiple interpreters in the same process

Added in version 3.14.

Source code: Lib/concurrent/interpreters [https://github.com/python/cpython/tree/3.14/Lib/concurrent/interpreters]

The **`concurrent.interpreters`** module constructs higher-level interfaces on top of the lower level **`_interpreters`** module.

The module is primarily meant to provide a basic API for managing interpreters (AKA “subinterpreters”) and running things in them. Running mostly involves switching to an interpreter (in the current thread) and calling a function in that execution context.

For concurrency, interpreters themselves (and this module) don’t provide much more than isolation, which on its own isn’t useful. Actual concurrency is available separately through **`threads`** See below

See also

`InterpreterPoolExecutor`
Combines threads with interpreters in a familiar interface.

Isolating Extension Modules

How to update an extension module to support multiple interpreters.

PEP 554 [https://peps.python.org/pep-0554/]

PEP 734 [https://peps.python.org/pep-0734/]

PEP 684 [https://peps.python.org/pep-0684/]

Availability: not WASI.

This module does not work or is not available on WebAssembly. See WebAssembly platforms for more information.

Key details

Before we dive in further, there are a small number of details to keep in mind about using multiple interpreters:

- isolated, by default
- no implicit threads
- not all PyPI packages support use in multiple interpreters yet

Introduction

An “interpreter” is effectively the execution context of the Python runtime. It contains all of the state the runtime needs to execute a program. This includes things like the import state and builtins. (Each thread, even if there’s only the main thread, has some extra runtime state, in addition to the current interpreter, related to the current exception and the bytecode eval loop.)

The concept and functionality of the interpreter have been a part of Python since version 2.2, but the feature was only available through the C-API and not well known, and the isolation was relatively incomplete until version 3.12.

Multiple Interpreters and Isolation

A Python implementation may support using multiple interpreters in the same process. CPython has this support. Each interpreter is effectively isolated from the others (with a limited number of carefully managed process-global exceptions to the rule).

That isolation is primarily useful as a strong separation between distinct logical components of a program, where you want to have careful control of how those components interact.

Note

Interpreters in the same process can technically never be strictly isolated from one another since there are few restrictions on memory access within the same process. The Python runtime makes a best effort at isolation but extension modules may easily violate that. Therefore, do not use multiple interpreters in security-sensitive situations, where they shouldn’t have access to each other’s data.

Running in an Interpreter

Running in a different interpreter involves switching to it in the current thread and then calling some function. The runtime will execute the function using the current interpreter’s state. The `concurrent.interpreters` module provides a basic API for creating and managing interpreters, as well as the switch-and-call operation.

No other threads are automatically started for the operation. There is a helper for that though. There is another dedicated helper for calling the builtin `exec()` in an interpreter.

When `exec()` (or `eval()`) are called in an interpreter, they run using the interpreter's `__main__` module as the "globals" namespace. The same is true for functions that aren't associated with any module. This is the same as how scripts invoked from the command-line run in the `__main__` module.

Concurrency and Parallelism

As noted earlier, interpreters do not provide any concurrency on their own. They strictly represent the isolated execution context the runtime will use *in the current thread*. That isolation makes them similar to processes, but they still enjoy in-process efficiency, like threads.

All that said, interpreters do naturally support certain flavors of concurrency. There's a powerful side effect of that isolation. It enables a different approach to concurrency than you can take with async or threads. It's a similar concurrency model to CSP or the actor model, a model which is relatively easy to reason about.

You can take advantage of that concurrency model in a single thread, switching back and forth between interpreters, Stackless-style. However, this model is more useful when you combine interpreters with multiple threads. This mostly involves starting a new thread, where you switch to another interpreter and run what you want there.

Each actual thread in Python, even if you’re only running in the main thread, has its own *current* execution context. Multiple threads can use the same interpreter or different ones.

At a high level, you can think of the combination of threads and interpreters as threads with opt-in sharing.

As a significant bonus, interpreters are sufficiently isolated that they do not share the GIL, which means combining threads with multiple interpreters enables full multi-core parallelism. (This has been the case since Python 3.12.)

Communication Between Interpreters

In practice, multiple interpreters are useful only if we have a way to communicate between them. This usually involves some form of message passing, but can even mean sharing data in some carefully managed way.

With this in mind, the `concurrent.interpreters` module provides a `queue.Queue` implementation, available through `create_queue()`.

“Sharing” Objects

Any data actually shared between interpreters loses the thread-safety provided by the GIL. There are various options for dealing with this in extension modules. However, from Python code the lack of thread-safety means objects can’t actually be shared, with a few exceptions. Instead, a copy must be created, which means mutable objects won’t stay in sync.

By default, most objects are copied with `pickle` when they are passed to another interpreter. Nearly all of the immutable builtin objects are either directly shared or copied efficiently. For example:

- `None`
- `bool` (`True` and `False`)
- `bytes`
- `str`
- `int`
- `float`
- `tuple` (of similarly supported objects)

There is a small number of Python types that actually share mutable data between interpreters:

- `memoryview`
- `Queue`

Reference

This module defines the following functions:

concurrent.interpreters.list_all()

Return a `list` of `Interpreter` objects, one for each existing interpreter.

concurrent.interpreters.get_current()

Return an `Interpreter` object for the currently running interpreter.

concurrent.interpreters.get_main()

Return an `Interpreter` object for the main interpreter. This is the interpreter the runtime created to run the REPL or the script given at the command-line. It is usually the only one.

concurrent.interpreters.create()

Initialize a new (idle) Python interpreter and return a `Interpreter` object for it.

concurrent.interpreters.create_queue()

Initialize a new cross-interpreter queue and return a `Queue` object for it.

Interpreter objects

class concurrent.interpreters.Interpreter(*id*)

A single interpreter in the current process.

Generally, `Interpreter` shouldn't be called directly. Instead, use `create()` or one of the other module functions.

id

(read-only)

The underlying interpreter's ID.

whence

(read-only)

A string describing where the interpreter came from.

is_running()

Return `True` if the interpreter is currently executing code in its **`__main__`** module and `False` otherwise.

close()

Finalize and destroy the interpreter.

prepare_main(*ns=None*, ***kwargs*)

Bind objects in the interpreter’s `__main__` module.

Some objects are actually shared and some are copied efficiently, but most are copied via `pickle`. See “Sharing” Objects.

exec(*code*, /, *dedent=True*)

Run the given source code in the interpreter (in the current thread).

call(*callable*, /, **args*, ***kwargs*)

Return the result of calling running the given function in the interpreter (in the current thread).

call_in_thread(*callable*, /, **args*, ***kwargs*)

Run the given function in the interpreter (in a new thread).

Exceptions

exception concurrent.interpreters.InterpreterError

This exception, a subclass of `Exception`, is raised when an interpreter-related error happens.

exception concurrent.interpreters.InterpreterNotFoundError

This exception, a subclass of `InterpreterError`, is raised when the targeted interpreter no longer exists.

exception concurrent.interpreters.ExecutionFailed

This exception, a subclass of `InterpreterError`, is raised when the running code raised an uncaught exception.

excinfo

A basic snapshot of the exception raised in the other interpreter.

exception concurrent.interpreters.NotShareableError

This exception, a subclass of **`TypeError`**, is raised when an object cannot be sent to another interpreter.

Communicating Between Interpreters

class concurrent.interpreters.Queue(*id*)

A wrapper around a low-level, cross-interpreter queue, which implements the **`queue.Queue`** interface. The underlying queue can only be created through **`create_queue()`**.

Some objects are actually shared and some are copied efficiently, but most are copied via **`pickle`**. See “Sharing” Objects.

id

(read-only)

The queue’s ID.

exception concurrent.interpreters.QueueEmptyError

This exception, a subclass of **`queue.Empty`**, is raised from **`Queue.get()`** and **`Queue.get_nowait()`** when the queue is empty.

exception concurrent.interpreters.QueueFullError

This exception, a subclass of **`queue.Full`**, is raised from **`Queue.put()`** and **`Queue.put_nowait()`** when the queue is full.

Basic usage

Creating an interpreter and running code in it:

```
from concurrent import interpreters
```

```
interp = interpreters.create()

# Run in the current OS thread.

interp.exec('print("spam!")')

interp.exec("""if True:
    print('spam!')
    """)

from textwrap import dedent
interp.exec(dedent("""
    print('spam!')
    """))

def run(arg):
    return arg

res = interp.call(run, 'spam!')
print(res)

def run():
    print('spam!')

interp.call(run)

# Run in new OS thread.

t = interp.call_in_thread(run)
t.join()
```

`subprocess` — Subprocess management

Source code: Lib/subprocess.py [https://github.com/python/cpython/tree/3.14/Lib/subprocess.py]

The `subprocess` module allows you to spawn new processes, connect to their input/output/error pipes, and obtain their return codes. This module intends to replace several older modules and functions:

```
os.system
os.spawn*
```

Information about how the `subprocess` module can be used to replace these modules and functions can be found in the following sections.

See also

PEP 324 [https://peps.python.org/pep-0324/] – PEP proposing the subprocess module

Availability: not Android, not iOS, not WASI.

This module is not supported on mobile platforms or WebAssembly platforms.

Using the `subprocess` Module

The recommended approach to invoking subprocesses is to use the **`run()`** function for all use cases it can handle. For more advanced use cases, the underlying **`Popen`** interface can be used directly.

subprocess.run(*args*, *, *stdin=None*, *input=None*, *stdout=None*, *stderr=None*, *capture_output=False*, *shell=False*, *cwd=None*, *timeout=None*, *check=False*, *encoding=None*, *errors=None*, *text=None*, *env=None*, *universal_newlines=None*, ***other_popen_kwargs*)

Run the command described by *args*. Wait for command to complete, then return a **`CompletedProcess`** instance.

The arguments shown above are merely the most common ones, described below in Frequently Used Arguments (hence the use of keyword-only notation in the abbreviated signature). The full function signature is largely the same as that of the **`Popen`** constructor - most of the arguments to this function are passed through to that interface. (*timeout*, *input*, *check*, and *capture_output* are not.)

If *capture_output* is true, stdout and stderr will be captured. When used, the internal **`Popen`** object is automatically created with *stdout* and *stderr* both set to **`PIPE`**. The *stdout* and *stderr* arguments may not be supplied at the same time as *capture_output*. If you wish to capture and combine both streams into one, set *stdout* to **`PIPE`** and *stderr* to **`STDOUT`**, instead of using *capture_output*.

A *timeout* may be specified in seconds, it is internally passed on to **`Popen.communicate()`**. If the timeout expires, the child process will be killed and waited for. The **`TimeoutExpired`** exception will be re-raised after the child process has terminated. The initial process creation itself cannot be interrupted on many platform APIs so you are not guaranteed to see a timeout exception until at least after however long process creation takes.

The *input* argument is passed to **`Popen.communicate()`** and thus to the subprocess's stdin. If used it must be a byte sequence, or a string if *encoding* or *errors* is specified or *text* is true. When used, the internal **`Popen`** object is automatically created with *stdin* set to **`PIPE`**, and the *stdin* argument may not be used as well.

If *check* is true, and the process exits with a non-zero exit code, a **`CalledProcessError`** exception will be raised. Attributes of that exception hold the arguments, the exit code, and stdout and stderr if they were captured.

If *encoding* or *errors* are specified, or *text* is true, file objects for stdin, stdout and stderr are opened in text mode using the specified *encoding* and *errors* or the **`io.TextIOWrapper`** default. The *universal_newlines* argument is equivalent to *text* and is provided for backwards compatibility. By default, file objects are opened in binary mode.

If *env* is not `None`, it must be a mapping that defines the environment variables for the new process; these are used instead of the default behavior of inheriting the current process' environment. It is passed directly to **`Popen`**. This mapping can be str to str on any platform or bytes to bytes on POSIX platforms much like **`os.environ`** or **`os.environb`**.

Examples:

```
>>> subprocess.run(["ls", "-l"])  # doesn't capture output
CompletedProcess(args=['ls', '-l'], returncode=0)
```

```
>>> subprocess.run("exit 1", shell=True, check=True)
Traceback (most recent call last):
  ...
subprocess.CalledProcessError: Command 'exit 1' returned
non-zero exit status 1

>>> subprocess.run(["ls", "-l", "/dev/null"],
capture_output=True)
CompletedProcess(args=['ls', '-l', '/dev/null'],
returncode=0,
stdout=b'crw-rw-rw- 1 root root 1, 3 Jan 23 16:23
/dev/null\n', stderr=b'')
```

Added in version 3.5.

Changed in version 3.6: Added *encoding* and *errors* parameters

Changed in version 3.7: Added the *text* parameter, as a more understandable alias of *universal_newlines*. Added the *capture_output* parameter.

Changed in version 3.12: Changed Windows shell search order for `shell=True`. The current directory and `%PATH%` are replaced with `%COMSPEC%` and `%SystemRoot%\System32\cmd.exe`. As a result, dropping a malicious program named `cmd.exe` into a current directory no longer works.

class subprocess.CompletedProcess

The return value from `run()`, representing a process that has finished.

args

The arguments used to launch the process. This may be a list or a string.

returncode

Exit status of the child process. Typically, an exit status of 0 indicates that it ran successfully.

A negative value `-N` indicates that the child was terminated by signal `N` (POSIX only).

stdout

Captured stdout from the child process. A bytes sequence, or a string if `run()` was called with an encoding, errors, or text=True. `None` if stdout was not captured.

If you ran the process with `stderr=subprocess.STDOUT`, stdout and stderr will be combined in this attribute, and `stderr` will be `None`.

stderr

Captured stderr from the child process. A bytes sequence, or a string if `run()` was called with an encoding, errors, or text=True. `None` if stderr was not captured.

check_returncode()

If `returncode` is non-zero, raise a `CalledProcessError`.

Added in version 3.5.

subprocess.DEVNULL

Special value that can be used as the *stdin*, *stdout* or *stderr* argument to `Popen` and indicates that the special file `os.devnull` will be used.

Added in version 3.3.

subprocess.PIPE

Special value that can be used as the *stdin*, *stdout* or *stderr* argument to `Popen` and indicates that a pipe to the standard stream should be opened. Most useful with `Popen.communicate()`.

subprocess.STDOUT

Special value that can be used as the *stderr* argument to **`Popen`** and indicates that standard error should go into the same handle as standard output.

exception subprocess.SubprocessError

Base class for all other exceptions from this module.

Added in version 3.3.

exception subprocess.TimeoutExpired

Subclass of **`SubprocessError`**, raised when a timeout expires while waiting for a child process.

cmd

Command that was used to spawn the child process.

timeout

Timeout in seconds.

output

Output of the child process if it was captured by **`run()`** or **`check_output()`**. Otherwise, `None`. This is always **`bytes`** when any output was captured regardless of the `text=True` setting. It may remain `None` instead of `b''` when no output was observed.

stdout

Alias for output, for symmetry with **`stderr`**.

stderr

Stderr output of the child process if it was captured by **`run()`**. Otherwise, `None`. This is always **`bytes`** when stderr output was captured regardless of the `text=True` setting. It may remain `None` instead of `b''` when no stderr output was observed.

Added in version 3.3.

Changed in version 3.5: stdout and *stderr* attributes added

exception subprocess.CalledProcessError

Subclass of `SubprocessError`, raised when a process run by `check_call()`, `check_output()`, or `run()` (with `check=True`) returns a non-zero exit status.

returncode

Exit status of the child process. If the process exited due to a signal, this will be the negative signal number.

cmd

Command that was used to spawn the child process.

output

Output of the child process if it was captured by `run()` or `check_output()`. Otherwise, `None`.

stdout

Alias for output, for symmetry with `stderr`.

stderr

Stderr output of the child process if it was captured by `run()`. Otherwise, `None`.

Changed in version 3.5: stdout and *stderr* attributes added

Frequently Used Arguments

To support a wide variety of use cases, the `Popen` constructor (and the convenience functions) accept a large number of optional arguments. For most typical use cases, many of these arguments can be safely left at their default values. The arguments that are most commonly needed are:

> *args* is required for all calls and should be a string, or a sequence of program arguments. Providing a sequence of arguments is generally preferred, as it allows the module to take care of any required escaping and quoting of arguments (e.g. to permit spaces in file names). If passing a single string, either *shell* must be **`True`** (see below) or else the string must simply name the program to be executed without specifying any arguments.
>
> *stdin*, *stdout* and *stderr* specify the executed program’s standard input, standard output and standard error file handles, respectively. Valid values are `None`, **`PIPE`**, **`DEVNULL`**, an existing file descriptor (a positive integer), and an existing file object with a valid file descriptor. With the default settings of `None`, no redirection will occur. **`PIPE`** indicates that a new pipe to the child should be created. **`DEVNULL`** indicates that the special file **`os.devnull`** will be used. Additionally, *stderr* can be **`STDOUT`**, which indicates that the stderr data from the child process should be captured into the same file handle as for *stdout*.
>
> If *encoding* or *errors* are specified, or *text* (also known as *universal_newlines*) is true, the file objects *stdin*, *stdout* and *stderr* will be opened in text mode using the *encoding* and *errors* specified in the call or the defaults for **`io.TextIOWrapper`**.
>
> For *stdin*, line ending characters `'\n'` in the input will be converted to the default line separator **`os.linesep`**. For *stdout* and *stderr*, all line endings in the output will be converted to `'\n'`. For more information see the

documentation of the `io.TextIOWrapper` class when the *newline* argument to its constructor is `None`.

If text mode is not used, *stdin*, *stdout* and *stderr* will be opened as binary streams. No encoding or line ending conversion is performed.

Changed in version 3.6: Added the *encoding* and *errors* parameters.

Changed in version 3.7: Added the *text* parameter as an alias for *universal_newlines*.

Note

The newlines attribute of the file objects `Popen.stdin`, `Popen.stdout` and `Popen.stderr` are not updated by the `Popen.communicate()` method.

If *shell* is `True`, the specified command will be executed through the shell. This can be useful if you are using Python primarily for the enhanced control flow it offers over most system shells and still want convenient access to other shell features such as shell pipes, filename wildcards, environment variable expansion, and expansion of `~` to a user’s home directory. However, note that Python itself offers implementations of many shell-like features (in particular, `glob`, `fnmatch`, `os.walk()`, `os.path.expandvars()`, `os.path.expanduser()`, and `shutil`).

Changed in version 3.3: When *universal_newlines* is `True`, the class uses the encoding `locale.getpreferredencoding(False)` instead of `locale.getpreferredencoding()`. See the `io.TextIOWrapper` class for more information on this change.

Note

Read the Security Considerations section before using `shell=True`.

These options, along with all of the other options, are described in more detail in the **`Popen`** constructor documentation.

Popen Constructor

The underlying process creation and management in this module is handled by the **`Popen`** class. It offers a lot of flexibility so that developers are able to handle the less common cases not covered by the convenience functions.

class subprocess.Popen(*args*, *bufsize=-1*, *executable=None*, *stdin=None*, *stdout=None*, *stderr=None*, *preexec_fn=None*, *close_fds=True*, *shell=False*, *cwd=None*, *env=None*, *universal_newlines=None*, *startupinfo=None*, *creationflags=0*, *restore_signals=True*, *start_new_session=False*, *pass_fds=()*, *, *group=None*, *extra_groups=None*, *user=None*, *umask=-1*, *encoding=None*, *errors=None*, *text=None*, *pipesize=-1*, *process_group=None*)

Execute a child program in a new process. On POSIX, the class uses **`os.execvpe()`**-like behavior to execute the child program. On Windows, the class uses the Windows `CreateProcess()` function. The arguments to **`Popen`** are as follows.

args should be a sequence of program arguments or else a single string or path-like object. By default, the program to execute is the first item in *args* if *args* is a sequence. If *args* is a string, the interpretation is platform-dependent and described below. See the *shell* and *executable* arguments for additional differences from the default behavior. Unless otherwise stated, it is recommended to pass *args* as a sequence.

Warning

For maximum reliability, use a fully qualified path for the executable. To search for an unqualified name on **`PATH`**, use `shutil.which()`. On all platforms, passing `sys.executable` is the recommended way to launch the current Python interpreter again, and use the `-m` command-line format to launch an installed module.

Resolving the path of *executable* (or the first item of *args*) is platform dependent. For POSIX, see `os.execvpe()`, and note that when resolving or searching for the executable path, *cwd* overrides the current working directory and *env* can override the `PATH` environment variable. For Windows, see the documentation of the `lpApplicationName` and `lpCommandLine` parameters of WinAPI `CreateProcess`, and note that when resolving or searching for the executable path with `shell=False`, *cwd* does not override the current working directory and *env* cannot override the `PATH` environment variable. Using a full path avoids all of these variations.

An example of passing some arguments to an external program as a sequence is:

```
Popen(["/usr/bin/git", "commit", "-m", "Fixes a bug."])
```

On POSIX, if *args* is a string, the string is interpreted as the name or path of the program to execute. However, this can only be done if not passing arguments to the program.

Note

It may not be obvious how to break a shell command into a sequence of arguments, especially in complex cases. `shlex.split()` can illustrate how to determine the correct tokenization for *args*:

```
>>> import shlex, subprocess
>>> command_line = input()
/bin/vikings -input eggs.txt -output "spam spam.txt" -cmd
"echo '$MONEY'"
>>> args = shlex.split(command_line)
>>> print(args)
['/bin/vikings', '-input', 'eggs.txt', '-output', 'spam
spam.txt', '-cmd', "echo '$MONEY'"]
>>> p = subprocess.Popen(args) # Success!
```

Note in particular that options (such as *-input*) and arguments (such as *eggs.txt*) that are separated by whitespace in the shell go in separate list elements, while arguments that need quoting or backslash escaping when used in the shell (such as filenames containing spaces or the *echo* command shown above) are single list elements.

On Windows, if *args* is a sequence, it will be converted to a string in a manner described in Converting an argument sequence to a string on Windows. This is because the underlying `CreateProcess()` operates on strings.

Changed in version 3.6: args parameter accepts a path-like object if *shell* is `False` and a sequence containing path-like objects on POSIX.

Changed in version 3.8: args parameter accepts a path-like object if *shell* is `False` and a sequence containing bytes and path-like objects on Windows.

The *shell* argument (which defaults to `False`) specifies whether to use the shell as the program to execute. If *shell* is `True`, it is recommended to pass *args* as a string rather than as a sequence.

On POSIX with `shell=True`, the shell defaults to `/bin/sh`. If *args* is a string, the string specifies the command to execute through the shell. This means that the string must be formatted exactly as it would be when typed at the shell prompt. This includes, for example, quoting or backslash escaping filenames with spaces in them. If *args* is a sequence, the first item specifies the command string, and any additional items will be treated as additional arguments to the shell itself. That is to say, **`Popen`** does the equivalent of:

```
Popen(['/bin/sh', '-c', args[0], args[1], ...])
```

On Windows with `shell=True`, the **`COMSPEC`** environment variable specifies the default shell. The only time you need to specify `shell=True` on Windows is when the command you wish to execute is built into the shell (e.g. **dir** or **copy**). You do not need `shell=True` to run a batch file or console-based executable.

Note

Read the Security Considerations section before using `shell=True`.

bufsize will be supplied as the corresponding argument to the **`open()`** function when creating the stdin/stdout/stderr pipe file objects:

- `0` means unbuffered (read and write are one system call and can return short)
- `1` means line buffered (only usable if `text=True` or `universal_newlines=True`)
- any other positive value means use a buffer of approximately that size
- negative bufsize (the default) means the system default of io.DEFAULT_BUFFER_SIZE will be used.

Changed in version 3.3.1: bufsize now defaults to -1 to enable buffering by default to match the behavior that most code expects. In versions prior to Python 3.2.4 and 3.3.1 it incorrectly defaulted to `0` which was unbuffered and allowed short reads. This was unintentional and did not match the behavior of Python 2 as most code expected.

The *executable* argument specifies a replacement program to execute. It is very seldom needed. When `shell=False`, *executable* replaces the program to execute specified by *args*. However, the original *args* is still passed to the program. Most programs treat the program specified by *args* as the command name, which can then be different from the program actually executed. On POSIX, the *args* name becomes the display name for the executable in utilities such as **ps**. If `shell=True`, on POSIX the *executable* argument specifies a replacement shell for the default `/bin/sh`.

Changed in version 3.6: executable parameter accepts a path-like object on POSIX.

Changed in version 3.8: executable parameter accepts a bytes and path-like object on Windows.

Changed in version 3.12: Changed Windows shell search order for `shell=True`. The current directory and `%PATH%` are replaced with `%COMSPEC%` and `%SystemRoot%\System32\cmd.exe`. As a result, dropping a malicious program named `cmd.exe` into a current directory no longer works.

stdin, *stdout* and *stderr* specify the executed program's standard input, standard output and standard error file handles, respectively. Valid values are `None`, **`PIPE`**, **`DEVNULL`**, an existing file descriptor (a positive integer), and an existing file object with a valid file descriptor. With the default settings of `None`, no redirection will occur. **`PIPE`** indicates that a new pipe to the child should be created. **`DEVNULL`** indicates that the special file **`os.devnull`** will be used. Additionally, *stderr* can be **`STDOUT`**, which indicates that the stderr data from the applications should be captured into the same file handle as for *stdout*.

If *preexec_fn* is set to a callable object, this object will be called in the child process just before the child is executed. (POSIX only)

Warning

The *preexec_fn* parameter is NOT SAFE to use in the presence of threads in your application. The child process could deadlock before exec is called.

Note

If you need to modify the environment for the child use the *env* parameter rather than doing it in a *preexec_fn*. The *start_new_session* and *process_group* parameters should take the place of code using *preexec_fn* to call **`os.setsid()`** or **`os.setpgid()`** in the child.

Changed in version 3.8: The *preexec_fn* parameter is no longer supported in subinterpreters. The use of the parameter in a subinterpreter raises **`RuntimeError`**. The new restriction may affect applications that are deployed in mod_wsgi, uWSGI, and other embedded environments. If *close_fds* is true, all file descriptors except `0`, `1` and `2` will be closed before the child process is executed. Otherwise when *close_fds* is false, file descriptors obey their inheritable flag as described in Inheritance of File Descriptors.

On Windows, if *close_fds* is true then no handles will be inherited by the child process unless explicitly passed in the `handle_list` element of **`STARTUPINFO.lpAttributeList`**, or by standard handle redirection.

Changed in version 3.2: The default for *close_fds* was changed from **`False`** to what is described above.

Changed in version 3.7: On Windows the default for *close_fds* was changed from **`False`** to **`True`** when redirecting the standard handles. It’s now possible to set *close_fds* to **`True`** when redirecting the standard handles.

pass_fds is an optional sequence of file descriptors to keep open between the parent and child. Providing any *pass_fds* forces *close_fds* to be **`True`**. (POSIX only)

Changed in version 3.2: The *pass_fds* parameter was added.

If *cwd* is not `None`, the function changes the working directory to *cwd* before executing the child. *cwd* can be a string, bytes or path-like object. On POSIX, the function looks for *executable* (or for the first item in *args*) relative to *cwd* if the executable path is a relative path.

Changed in version 3.6: cwd parameter accepts a path-like object on POSIX.

Changed in version 3.7: cwd parameter accepts a path-like object on Windows.

Changed in version 3.8: cwd parameter accepts a bytes object on Windows.

If *restore_signals* is true (the default) all signals that Python has set to SIG_IGN are restored to SIG_DFL in the child process before the exec. Currently this includes the SIGPIPE, SIGXFZ and SIGXFSZ signals. (POSIX only)

Changed in version 3.2: restore_signals was added.

If *start_new_session* is true the `setsid()` system call will be made in the child process prior to the execution of the subprocess.

Availability: POSIX

Changed in version 3.2: start_new_session was added.

If *process_group* is a non-negative integer, the `setpgid(0, value)` system call will be made in the child process prior to the execution of the subprocess.

Availability: POSIX

Changed in version 3.11: process_group was added.

If *group* is not `None`, the setregid() system call will be made in the child process prior to the execution of the subprocess. If the provided value is a string, it will be looked up via **`grp.getgrnam()`** and the value in `gr_gid` will be used. If the value is an integer, it will be passed verbatim. (POSIX only)

Availability: POSIX

Added in version 3.9.

If *extra_groups* is not `None`, the setgroups() system call will be made in the child process prior to the execution of the subprocess. Strings provided in *extra_groups* will be looked up via **`grp.getgrnam()`** and the values in `gr_gid` will be used. Integer values will be passed verbatim. (POSIX only)

Availability: POSIX

Added in version 3.9.

If *user* is not `None`, the setreuid() system call will be made in the child process prior to the execution of the subprocess. If the provided value is a string, it will be looked up via **`pwd.getpwnam()`** and the value in `pw_uid` will be used. If the value is an integer, it will be passed verbatim. (POSIX only)

Availability: POSIX

Added in version 3.9.

If *umask* is not negative, the umask() system call will be made in the child process prior to the execution of the subprocess.

Availability: POSIX

Added in version 3.9.

If *env* is not `None`, it must be a mapping that defines the environment variables for the new process; these are used instead of the default behavior of inheriting the current process’ environment. This mapping can be str to str on any platform or bytes to bytes on POSIX platforms much like `os.environ` or `os.environb`.

Note

If specified, *env* must provide any variables required for the program to execute. On Windows, in order to run a side-by-side assembly [https://en.wikipedia.org/wiki/Side-by-Side_Assembly] the specified *env* **must** include a valid `%SystemRoot%`.

If *encoding* or *errors* are specified, or *text* is true, the file objects *stdin*, *stdout* and *stderr* are opened in text mode with the specified *encoding* and *errors*, as described above in Frequently Used Arguments. The *universal_newlines* argument is equivalent to *text* and is provided for backwards compatibility. By default, file objects are opened in binary mode.

Added in version 3.6: encoding and *errors* were added.

Added in version 3.7: text was added as a more readable alias for *universal_newlines*.

If given, *startupinfo* will be a `STARTUPINFO` object, which is passed to the underlying `CreateProcess` function.

If given, *creationflags*, can be one or more of the following flags:

- `CREATE_NEW_CONSOLE`
- `CREATE_NEW_PROCESS_GROUP`
- `ABOVE_NORMAL_PRIORITY_CLASS`
- `BELOW_NORMAL_PRIORITY_CLASS`
- `HIGH_PRIORITY_CLASS`
- `IDLE_PRIORITY_CLASS`

- **`NORMAL_PRIORITY_CLASS`**
- **`REALTIME_PRIORITY_CLASS`**
- **`CREATE_NO_WINDOW`**
- **`DETACHED_PROCESS`**
- **`CREATE_DEFAULT_ERROR_MODE`**
- **`CREATE_BREAKAWAY_FROM_JOB`**

pipesize can be used to change the size of the pipe when **`PIPE`** is used for *stdin*, *stdout* or *stderr*. The size of the pipe is only changed on platforms that support this (only Linux at this time of writing). Other platforms will ignore this parameter.

Changed in version 3.10: Added the *pipesize* parameter.

Popen objects are supported as context managers via the **`with`** statement: on exit, standard file descriptors are closed, and the process is waited for.

```
with Popen(["ifconfig"], stdout=PIPE) as proc:
    log.write(proc.stdout.read())
```

Popen and the other functions in this module that use it raise an auditing event `subprocess.Popen` with arguments `executable`, `args`, `cwd`, and `env`. The value for `args` may be a single string or a list of strings, depending on platform.

Changed in version 3.2: Added context manager support.

Changed in version 3.6: Popen destructor now emits a **`ResourceWarning`** warning if the child process is still running.

Changed in version 3.8: Popen can use **`os.posix_spawn()`** in some cases for better performance. On Windows Subsystem for Linux and QEMU User Emulation, Popen constructor using **`os.posix_spawn()`** no longer raise an exception on errors like missing program, but the child process fails with a non-zero **`returncode`**.

Exceptions

Exceptions raised in the child process, before the new program has started to execute, will be re-raised in the parent.

The most common exception raised is `OSError`. This occurs, for example, when trying to execute a non-existent file. Applications should prepare for `OSError` exceptions. Note that, when `shell=True`, `OSError` will be raised by the child only if the selected shell itself was not found. To determine if the shell failed to find the requested application, it is necessary to check the return code or output from the subprocess.

A `ValueError` will be raised if `Popen` is called with invalid arguments.

`check_call()` and `check_output()` will raise `CalledProcessError` if the called process returns a non-zero return code.

All of the functions and methods that accept a *timeout* parameter, such as `run()` and `Popen.communicate()` will raise `TimeoutExpired` if the timeout expires before the process exits.

Exceptions defined in this module all inherit from `SubprocessError`.

Added in version 3.3: The `SubprocessError` base class was added.

Security Considerations

Unlike some other popen functions, this library will not implicitly choose to call a system shell. This means that all characters, including shell metacharacters, can safely be passed to child processes. If the shell is invoked explicitly, via `shell=True`, it is the application's responsibility to ensure that all whitespace and metacharacters are quoted appropriately to avoid shell injection [https://en.wikipedia.org/wiki/Shell_injection#Shell_injection] vulnerabilities. On some platforms, it is possible to use **`shlex.quote()`** for this escaping.

On Windows, batch files (`*.bat` or `*.cmd`) may be launched by the operating system in a system shell regardless of the arguments passed to this library. This could result in arguments being parsed according to shell rules, but without any escaping added by Python. If you are intentionally launching a batch file with arguments from untrusted sources, consider passing `shell=True` to allow Python to escape special characters. See gh-114539 [https://github.com/python/cpython/issues/114539] for additional discussion.

Popen Objects

Instances of the **`Popen`** class have the following methods:

Popen.poll()

Check if child process has terminated. Set and return **`returncode`** attribute. Otherwise, returns `None`.

Popen.wait(*timeout=None*)

Wait for child process to terminate. Set and return **`returncode`** attribute.

If the process does not terminate after *timeout* seconds, raise a **`TimeoutExpired`** exception. It is safe to catch this exception and retry the wait.

Note

This will deadlock when using `stdout=PIPE` or `stderr=PIPE` and the child process generates enough output to a pipe such that it blocks waiting for the OS pipe buffer to accept more data. Use `Popen.communicate()` when using pipes to avoid that.

Note

When the `timeout` parameter is not `None`, then (on POSIX) the function is implemented using a busy loop (non-blocking call and short sleeps). Use the `asyncio` module for an asynchronous wait: see `asyncio.create_subprocess_exec`.

Changed in version 3.3: timeout was added.

Popen.communicate(*input=None*, *timeout=None*)

Interact with process: Send data to stdin. Read data from stdout and stderr, until end-of-file is reached. Wait for process to terminate and set the `returncode` attribute. The optional *input* argument should be data to be sent to the child process, or `None`, if no data should be sent to the child. If streams were opened in text mode, *input* must be a string. Otherwise, it must be bytes.

`communicate()` returns a tuple `(stdout_data, stderr_data)`. The data will be strings if streams were opened in text mode; otherwise, bytes.

Note that if you want to send data to the process’s stdin, you need to create the Popen object with `stdin=PIPE`. Similarly, to get anything other than `None` in the result tuple, you need to give `stdout=PIPE` and/or `stderr=PIPE` too.

If the process does not terminate after *timeout* seconds, a `TimeoutExpired` exception will be raised. Catching this exception and retrying communication will not lose any output. Supplying *input* to a subsequent post-timeout `communicate()` call is in undefined behavior and may become an error in the future.

The child process is not killed if the timeout expires, so in order to cleanup properly a well-behaved application should kill the child process and finish communication:

```
proc = subprocess.Popen(...)
try:
    outs, errs = proc.communicate(timeout=15)
except TimeoutExpired:
    proc.kill()
    outs, errs = proc.communicate()
```

After a call to `communicate()` raises `TimeoutExpired`, do not call `wait()`. Use an additional `communicate()` call to finish handling pipes and populate the `returncode` attribute.

Note

The data read is buffered in memory, so do not use this method if the data size is large or unlimited.

Changed in version 3.3: timeout was added.

Popen.send_signal(*signal*)

Sends the signal *signal* to the child.

Do nothing if the process completed.

Note

On Windows, SIGTERM is an alias for `terminate()`. CTRL_C_EVENT and CTRL_BREAK_EVENT can be sent to processes started with a *creationflags* parameter which includes `CREATE_NEW_PROCESS_GROUP`.

Popen.terminate()

Stop the child. On POSIX OSs the method sends **`SIGTERM`** to the child. On Windows the Win32 API function **`TerminateProcess()`** is called to stop the child.

Popen.kill()

Kills the child. On POSIX OSs the function sends SIGKILL to the child. On Windows **`kill()`** is an alias for **`terminate()`**.

The following attributes are also set by the class for you to access. Reassigning them to new values is unsupported:

Popen.args

The *args* argument as it was passed to **`Popen`** – a sequence of program arguments or else a single string.

Added in version 3.3.

Popen.stdin

If the *stdin* argument was **`PIPE`**, this attribute is a writeable stream object as returned by **`open()`**. If the *encoding* or *errors* arguments were specified or the *text* or *universal_newlines* argument was `True`, the stream is a text stream, otherwise it is a byte stream. If the *stdin* argument was not **`PIPE`**, this attribute is `None`.

Popen.stdout

If the *stdout* argument was **`PIPE`**, this attribute is a readable stream object as returned by **`open()`**. Reading from the stream provides output from the child process. If the *encoding* or *errors* arguments were specified or the *text* or *universal_newlines* argument was `True`, the stream is a text stream, otherwise it is a byte stream. If the *stdout* argument was not `PIPE`, this attribute is `None`.

Popen.stderr

If the *stderr* argument was `PIPE`, this attribute is a readable stream object as returned by `open()`. Reading from the stream provides error output from the child process. If the *encoding* or *errors* arguments were specified or the *text* or *universal_newlines* argument was `True`, the stream is a text stream, otherwise it is a byte stream. If the *stderr* argument was not `PIPE`, this attribute is `None`.

Warning

Use `communicate()` rather than `.stdin.write`, `.stdout.read` or `.stderr.read` to avoid deadlocks due to any of the other OS pipe buffers filling up and blocking the child process.

Popen.pid

The process ID of the child process.

Note that if you set the *shell* argument to `True`, this is the process ID of the spawned shell.

Popen.returncode

The child return code. Initially `None`, `returncode` is set by a call to the `poll()`, `wait()`, or `communicate()` methods if they detect that the process has terminated.

A `None` value indicates that the process hadn’t yet terminated at the time of the last method call.

A negative value `-N` indicates that the child was terminated by signal `N` (POSIX only).

Windows Popen Helpers

The `STARTUPINFO` class and following constants are only available on Windows.

class subprocess.STARTUPINFO(*, *dwFlags=0*, *hStdInput=None*, *hStdOutput=None*, *hStdError=None*, *wShowWindow=0*, *lpAttributeList=None*)

Partial support of the Windows STARTUPINFO [https://msdn.microsoft.com/en-us/library/ms686331(v=vs.85).aspx] structure is used for `Popen` creation. The following attributes can be set by passing them as keyword-only arguments.

Changed in version 3.7: Keyword-only argument support was added.

dwFlags

A bit field that determines whether certain `STARTUPINFO` attributes are used when the process creates a window.

```
si = subprocess.STARTUPINFO()
si.dwFlags = subprocess.STARTF_USESTDHANDLES |
subprocess.STARTF_USESHOWWINDOW
```

hStdInput

If `dwFlags` specifies `STARTF_USESTDHANDLES`, this attribute is the standard input handle for the process. If `STARTF_USESTDHANDLES` is not specified, the default for standard input is the keyboard buffer.

hStdOutput

If `dwFlags` specifies `STARTF_USESTDHANDLES`, this attribute is the standard output handle for the process.

Otherwise, this attribute is ignored and the default for standard output is the console window’s buffer.

hStdError

If **`dwFlags`** specifies **`STARTF_USESTDHANDLES`**, this attribute is the standard error handle for the process. Otherwise, this attribute is ignored and the default for standard error is the console window’s buffer.

wShowWindow

If **`dwFlags`** specifies **`STARTF_USESHOWWINDOW`**, this attribute can be any of the values that can be specified in the `nCmdShow` parameter for the ShowWindow [https://msdn.microsoft.com/en-us/library/ms633548(v=vs.85).aspx] function, except for `SW_SHOWDEFAULT`. Otherwise, this attribute is ignored.

`SW_HIDE` is provided for this attribute. It is used when **`Popen`** is called with `shell=True`.

lpAttributeList

A dictionary of additional attributes for process creation as given in `STARTUPINFOEX`, see UpdateProcThreadAttribute [https://msdn.microsoft.com/en-us/library/windows/desktop/ms686880(v=vs.85).aspx].

Supported attributes:

handle_list

Sequence of handles that will be inherited. *close_fds* must be true if non-empty.

The handles must be temporarily made inheritable by **`os.set_handle_inheritable()`** when passed to the **`Popen`** constructor, else **`OSError`** will be raised with Windows error `ERROR_INVALID_PARAMETER` (87).

Warning

In a multithreaded process, use caution to avoid leaking handles that are marked inheritable when combining this feature with concurrent calls to other process creation functions that inherit all handles such as `os.system()`. This also applies to standard handle redirection, which temporarily creates inheritable handles.

Added in version 3.7.

Windows Constants

The `subprocess` module exposes the following constants.

subprocess.STD_INPUT_HANDLE

The standard input device. Initially, this is the console input buffer, `CONIN$`.

subprocess.STD_OUTPUT_HANDLE

The standard output device. Initially, this is the active console screen buffer, `CONOUT$`.

subprocess.STD_ERROR_HANDLE

The standard error device. Initially, this is the active console screen buffer, `CONOUT$`.

subprocess.SW_HIDE

Hides the window. Another window will be activated.

subprocess.STARTF_USESTDHANDLES

Specifies that the `STARTUPINFO.hStdInput`, `STARTUPINFO.hStdOutput`, and `STARTUPINFO.hStdError` attributes contain additional information.

subprocess.STARTF_USESHOWWINDOW

Specifies that the `STARTUPINFO.wShowWindow` attribute contains additional information.

subprocess.STARTF_FORCEONFEEDBACK

A `STARTUPINFO.dwFlags` parameter to specify that the *Working in Background* mouse cursor will be displayed while a process is launching. This is the default behavior for GUI processes.

Added in version 3.13.

subprocess.STARTF_FORCEOFFFEEDBACK

A `STARTUPINFO.dwFlags` parameter to specify that the mouse cursor will not be changed when launching a process.

Added in version 3.13.

subprocess.CREATE_NEW_CONSOLE

The new process has a new console, instead of inheriting its parent’s console (the default).

subprocess.CREATE_NEW_PROCESS_GROUP

A `Popen` `creationflags` parameter to specify that a new process group will be created. This flag is necessary for using `os.kill()` on the subprocess.

This flag is ignored if `CREATE_NEW_CONSOLE` is specified.

subprocess.ABOVE_NORMAL_PRIORITY_CLASS

A `Popen` `creationflags` parameter to specify that a new process will have an above average priority.

Added in version 3.7.

subprocess.BELOW_NORMAL_PRIORITY_CLASS

A **Popen** `creationflags` parameter to specify that a new process will have a below average priority.

Added in version 3.7.

subprocess.HIGH_PRIORITY_CLASS

A **Popen** `creationflags` parameter to specify that a new process will have a high priority.

Added in version 3.7.

subprocess.IDLE_PRIORITY_CLASS

A **Popen** `creationflags` parameter to specify that a new process will have an idle (lowest) priority.

Added in version 3.7.

subprocess.NORMAL_PRIORITY_CLASS

A **Popen** `creationflags` parameter to specify that a new process will have a normal priority. (default)

Added in version 3.7.

subprocess.REALTIME_PRIORITY_CLASS

A **Popen** `creationflags` parameter to specify that a new process will have realtime priority. You should almost never use REALTIME_PRIORITY_CLASS, because this interrupts system threads that manage mouse input, keyboard input, and background disk flushing. This class can be appropriate for applications that “talk” directly to hardware or that perform brief tasks that should have limited interruptions.

Added in version 3.7.

subprocess.CREATE_NO_WINDOW

A **`Popen`** `creationflags` parameter to specify that a new process will not create a window.

Added in version 3.7.

subprocess.DETACHED_PROCESS

A **`Popen`** `creationflags` parameter to specify that a new process will not inherit its parent’s console. This value cannot be used with CREATE_NEW_CONSOLE.

Added in version 3.7.

subprocess.CREATE_DEFAULT_ERROR_MODE

A **`Popen`** `creationflags` parameter to specify that a new process does not inherit the error mode of the calling process. Instead, the new process gets the default error mode. This feature is particularly useful for multithreaded shell applications that run with hard errors disabled.

Added in version 3.7.

subprocess.CREATE_BREAKAWAY_FROM_JOB

A **`Popen`** `creationflags` parameter to specify that a new process is not associated with the job.

Added in version 3.7.

Older high-level API

Prior to Python 3.5, these three functions comprised the high level API to subprocess. You can now use **`run()`** in many cases, but lots of existing code calls these functions.

subprocess.call(*args*, *, *stdin=None*, *stdout=None*, *stderr=None*, *shell=False*, *cwd=None*, *timeout=None*, ***other_popen_kwargs*)

Run the command described by *args*. Wait for command to complete, then return the `returncode` attribute.

Code needing to capture stdout or stderr should use `run()` instead:

```
run(...).returncode
```

To suppress stdout or stderr, supply a value of `DEVNULL`.

The arguments shown above are merely some common ones. The full function signature is the same as that of the `Popen` constructor - this function passes all supplied arguments other than *timeout* directly through to that interface.

Note

Do not use `stdout=PIPE` or `stderr=PIPE` with this function. The child process will block if it generates enough output to a pipe to fill up the OS pipe buffer as the pipes are not being read from.

Changed in version 3.3: timeout was added.

Changed in version 3.12: Changed Windows shell search order for `shell=True`. The current directory and `%PATH%` are replaced with `%COMSPEC%` and `%SystemRoot%\System32\cmd.exe`. As a result, dropping a malicious program named `cmd.exe` into a current directory no longer works.

subprocess.check_call(*args*, ***, *stdin=None*, *stdout=None*, *stderr=None*, *shell=False*, *cwd=None*, *timeout=None*, ***other_popen_kwargs*)

Run command with arguments. Wait for command to complete. If the return code was zero then return, otherwise raise `CalledProcessError`. The `CalledProcessError` object will have the return code in the `returncode` attribute. If `check_call()` was unable to start the process it will propagate the exception that was raised.

Code needing to capture stdout or stderr should use `run()` instead:

```
run(..., check=True)
```

To suppress stdout or stderr, supply a value of `DEVNULL`.

The arguments shown above are merely some common ones. The full function signature is the same as that of the `Popen` constructor - this function passes all supplied arguments other than *timeout* directly through to that interface.

Note

Do not use `stdout=PIPE` or `stderr=PIPE` with this function. The child process will block if it generates enough output to a pipe to fill up the OS pipe buffer as the pipes are not being read from.

Changed in version 3.3: timeout was added.

Changed in version 3.12: Changed Windows shell search order for `shell=True`. The current directory and `%PATH%` are replaced with `%COMSPEC%` and `%SystemRoot%\System32\cmd.exe`. As a result, dropping a malicious program named `cmd.exe` into a current directory no longer works.

subprocess.check_output(*args*, *, *stdin=None*, *stderr=None*, *shell=False*, *cwd=None*, *encoding=None*, *errors=None*, *universal_newlines=None*, *timeout=None*, *text=None*, ***other_popen_kwargs*)

Run command with arguments and return its output.

If the return code was non-zero it raises a **`CalledProcessError`**. The **`CalledProcessError`** object will have the return code in the **`returncode`** attribute and any output in the **`output`** attribute.

This is equivalent to:

```
run(..., check=True, stdout=PIPE).stdout
```

The arguments shown above are merely some common ones. The full function signature is largely the same as that of **`run()`** - most arguments are passed directly through to that interface. One API deviation from **`run()`** behavior exists: passing `input=None` will behave the same as `input=b''` (or `input=''`, depending on other arguments) rather than using the parent’s standard input file handle.

By default, this function will return the data as encoded bytes. The actual encoding of the output data may depend on the command being invoked, so the decoding to text will often need to be handled at the application level.

This behaviour may be overridden by setting *text*, *encoding*, *errors*, or *universal_newlines* to `True` as described in Frequently Used Arguments and **`run()`**.

To also capture standard error in the result, use `stderr=subprocess.STDOUT`:

```
>>> subprocess.check_output(
...     "ls non_existent_file; exit 0",
...     stderr=subprocess.STDOUT,
...     shell=True)
'ls: non_existent_file: No such file or directory\n'
```

Added in version 3.1.

Changed in version 3.3: timeout was added.

Changed in version 3.4: Support for the *input* keyword argument was added.

Changed in version 3.6: encoding and *errors* were added. See `run()` for details.

Added in version 3.7: text was added as a more readable alias for *universal_newlines*.

Changed in version 3.12: Changed Windows shell search order for `shell=True`. The current directory and `%PATH%` are replaced with `%COMSPEC%` and `%SystemRoot%\System32\cmd.exe`. As a result, dropping a malicious program named `cmd.exe` into a current directory no longer works.

Replacing Older Functions with the `subprocess` Module

In this section, “a becomes b” means that b can be used as a replacement for a.

Note

All “a” functions in this section fail (more or less) silently if the executed program cannot be found; the “b” replacements raise `OSError` instead.

In addition, the replacements using `check_output()` will fail with a `CalledProcessError` if the requested operation produces a non-zero return code. The output is still available as the `output` attribute of the raised exception.

In the following examples, we assume that the relevant functions have already been imported from the `subprocess` module.

Replacing /bin/sh shell command substitution

```
output=$(mycmd myarg)
```

becomes:

```
output = check_output(["mycmd", "myarg"])
```

Replacing shell pipeline

```
output=$(dmesg | grep hda)
```

becomes:

```
p1 = Popen(["dmesg"], stdout=PIPE)
p2 = Popen(["grep", "hda"], stdin=p1.stdout, stdout=PIPE)
p1.stdout.close()  # Allow p1 to receive a SIGPIPE if p2 exits.
output = p2.communicate()[0]
```

The `p1.stdout.close()` call after starting the p2 is important in order for p1 to receive a SIGPIPE if p2 exits before p1.

Alternatively, for trusted input, the shell’s own pipeline support may still be used directly:

```
output=$(dmesg | grep hda)
```

becomes:

```
output = check_output("dmesg | grep hda", shell=True)
```

Replacing `os.system()`

```
sts = os.system("mycmd" + " myarg")
# becomes
```

```
retcode = call("mycmd" + " myarg", shell=True)
```

Notes:

- Calling the program through the shell is usually not required.
- The `call()` return value is encoded differently to that of `os.system()`.
- The `os.system()` function ignores SIGINT and SIGQUIT signals while the command is running, but the caller must do this separately when using the `subprocess` module.

A more realistic example would look like this:

```
try:
    retcode = call("mycmd" + " myarg", shell=True)
    if retcode < 0:
        print("Child was terminated by signal", -retcode,
file=sys.stderr)
    else:
        print("Child returned", retcode, file=sys.stderr)
except OSError as e:
    print("Execution failed:", e, file=sys.stderr)
```

Replacing the `os.spawn` family

P_NOWAIT example:

```
pid = os.spawnlp(os.P_NOWAIT, "/bin/mycmd", "mycmd", "myarg")
==>
pid = Popen(["/bin/mycmd", "myarg"]).pid
```

P_WAIT example:

```
retcode = os.spawnlp(os.P_WAIT, "/bin/mycmd", "mycmd", "myarg")
==>
retcode = call(["/bin/mycmd", "myarg"])
```

Vector example:

```
os.spawnvp(os.P_NOWAIT, path, args)
==>
Popen([path] + args[1:])
```

Environment example:

```
os.spawnlpe(os.P_NOWAIT, "/bin/mycmd", "mycmd", "myarg", env)
==>
Popen(["/bin/mycmd", "myarg"], env={"PATH": "/usr/bin"})
```

Replacing `os.popen()`

Return code handling translates as follows:

```
pipe = os.popen(cmd, 'w')
...
rc = pipe.close()
if rc is not None and rc >> 8:
    print("There were some errors")
==>
process = Popen(cmd, stdin=PIPE)
...
process.stdin.close()
if process.wait() != 0:
    print("There were some errors")
```

Legacy Shell Invocation Functions

This module also provides the following legacy functions from the 2.x `commands` module. These operations implicitly invoke the system shell and none of the guarantees described above regarding security and exception handling consistency are valid for these functions.

subprocess.getstatusoutput(*cmd*, *, *encoding=None*, *errors=None*)

Return `(exitcode, output)` of executing *cmd* in a shell.

Execute the string *cmd* in a shell with **`check_output()`** and return a 2-tuple `(exitcode, output)`. *encoding* and *errors* are used to decode output; see the notes on Frequently Used Arguments for more details.

A trailing newline is stripped from the output. The exit code for the command can be interpreted as the return code of subprocess. Example:

```
>>> subprocess.getstatusoutput('ls /bin/ls')
(0, '/bin/ls')
>>> subprocess.getstatusoutput('cat /bin/junk')
(1, 'cat: /bin/junk: No such file or directory')
>>> subprocess.getstatusoutput('/bin/junk')
(127, 'sh: /bin/junk: not found')
>>> subprocess.getstatusoutput('/bin/kill $$')
(-15, '')
```

Availability: Unix, Windows.

Changed in version 3.3.4: Windows support was added.

The function now returns (exitcode, output) instead of (status, output) as it did in Python 3.3.3 and earlier. exitcode has the same value as **`returncode`**.

Changed in version 3.11: Added the *encoding* and *errors* parameters.

subprocess.getoutput(*cmd*, *, *encoding=None*, *errors=None*)

Return output (stdout and stderr) of executing *cmd* in a shell.

Like **`getstatusoutput()`**, except the exit code is ignored and the return value is a string containing the command’s output. Example:

```
>>> subprocess.getoutput('ls /bin/ls')
'/bin/ls'
```

Availability: Unix, Windows.

Changed in version 3.3.4: Windows support added

Changed in version 3.11: Added the *encoding* and *errors* parameters.

Notes

Timeout Behavior

When using the `timeout` parameter in functions like `run()`, `Popen.wait()`, or `Popen.communicate()`, users should be aware of the following behaviors:

1. **Process Creation Delay**: The initial process creation itself cannot be interrupted on many platform APIs. This means that even when specifying a timeout, you are not guaranteed to see a timeout exception until at least after however long process creation takes.
2. **Extremely Small Timeout Values**: Setting very small timeout values (such as a few milliseconds) may result in almost immediate `TimeoutExpired` exceptions because process creation and system scheduling inherently require time.

Converting an argument sequence to a string on Windows

On Windows, an *args* sequence is converted to a string that can be parsed using the following rules (which correspond to the rules used by the MS C runtime):

1. Arguments are delimited by white space, which is either a space or a tab.
2. A string surrounded by double quotation marks is interpreted as a single argument, regardless of white space contained within. A quoted string can be embedded in an argument.
3. A double quotation mark preceded by a backslash is interpreted as a literal double quotation mark.
4. Backslashes are interpreted literally, unless they immediately precede a double quotation mark.
5. If backslashes immediately precede a double quotation mark, every pair of backslashes is interpreted as a literal backslash. If the number of backslashes is odd, the last backslash escapes the next double quotation mark as described in rule 3.

See also

`shlex`
  Module which provides function to parse and escape command lines.

Disable use of `posix_spawn()`

On Linux, `subprocess` defaults to using the `vfork()` system call internally when it is safe to do so rather than `fork()`. This greatly improves performance.

```
subprocess._USE_POSIX_SPAWN = False  # See CPython issue gh-
NNNNNN.
```

It is safe to set this to false on any Python version. It will have no effect on older or newer versions where unsupported. Do not assume the attribute is available to read. Despite the name, a true value does not indicate the corresponding function will be used, only that it may be.

Please file issues any time you have to use these private knobs with a way to reproduce the issue you were seeing. Link to that issue from a comment in your code.

Added in version 3.8: `_USE_POSIX_SPAWN`

`sched` — Event scheduler

Source code: Lib/sched.py
[https://github.com/python/cpython/tree/3.14/Lib/sched.py]

The **`sched`** module defines a class which implements a general purpose event scheduler:

class sched.scheduler(*timefunc=time.monotonic*, *delayfunc=time.sleep*)

The **`scheduler`** class defines a generic interface to scheduling events. It needs two functions to actually deal with the “outside world” — *timefunc* should be callable without arguments, and return a number (the “time”, in any units whatsoever). The *delayfunc* function should be callable with one argument, compatible with the output of *timefunc*, and should delay that many time units. *delayfunc* will also be called with the argument `0` after each event is run to allow other threads an opportunity to run in multi-threaded applications.

Changed in version 3.3: timefunc and *delayfunc* parameters are optional.

Changed in version 3.3: **`scheduler`** class can be safely used in multi-threaded environments.

Example:

```
>>> import sched, time
>>> s = sched.scheduler(time.time, time.sleep)
>>> def print_time(a='default'):
...     print("From print_time", time.time(), a)
...
```

```
>>> def print_some_times():
...     print(time.time())
...     s.enter(10, 1, print_time)
...     s.enter(5, 2, print_time, argument=('positional',))
...     # despite having higher priority, 'keyword' runs after
'positional' as enter() is relative
...     s.enter(5, 1, print_time, kwargs={'a': 'keyword'})
...     s.enterabs(1_650_000_000, 10, print_time, argument=
("first enterabs",))
...     s.enterabs(1_650_000_000, 5, print_time, argument=
("second enterabs",))
...     s.run()
...     print(time.time())
...
>>> print_some_times()
1652342830.3640375
From print_time 1652342830.3642538 second enterabs
From print_time 1652342830.3643398 first enterabs
From print_time 1652342835.3694863 positional
From print_time 1652342835.3696074 keyword
From print_time 1652342840.369612 default
1652342840.3697174
```

Scheduler Objects

`scheduler` instances have the following methods and attributes:

scheduler.enterabs(*time*, *priority*, *action*, *argument=()*, *kwargs={}*)

Schedule a new event. The *time* argument should be a numeric type compatible with the return value of the *timefunc* function passed to the constructor. Events scheduled for the same *time* will be executed in the order of their *priority*. A lower number represents a higher priority.

Executing the event means executing `action(*argument, **kwargs)`. *argument* is a sequence holding the positional arguments for *action*. *kwargs* is a dictionary holding the keyword arguments for *action*.

Return value is an event which may be used for later cancellation of the event (see `cancel()`).

Changed in version 3.3: argument parameter is optional.

Changed in version 3.3: kwargs parameter was added.

scheduler.enter(*delay*, *priority*, *action*, *argument=()*, *kwargs={}*)

Schedule an event for *delay* more time units. Other than the relative time, the other arguments, the effect and the return value are the same as those for `enterabs()`.

Changed in version 3.3: argument parameter is optional.

Changed in version 3.3: kwargs parameter was added.

scheduler.cancel(*event*)

Remove the event from the queue. If *event* is not an event currently in the queue, this method will raise a `ValueError`.

scheduler.empty()

Return `True` if the event queue is empty.

scheduler.run(*blocking=True*)

Run all scheduled events. This method will wait (using the *delayfunc* function passed to the constructor) for the next event, then execute it and so on until there are no more scheduled events.

If *blocking* is false executes the scheduled events due to expire soonest (if any) and then return the deadline of the next scheduled call in the scheduler (if any).

Either *action* or *delayfunc* can raise an exception. In either case, the scheduler will maintain a consistent state and propagate the exception. If an exception is raised by *action*, the event will not be attempted in future calls to `run()`.

If a sequence of events takes longer to run than the time available before the next event, the scheduler will simply fall behind. No events will be dropped; the calling code is responsible for canceling events which are no longer pertinent.

Changed in version 3.3: blocking parameter was added.

scheduler.queue

Read-only attribute returning a list of upcoming events in the order they will be run. Each event is shown as a named tuple with the following fields: time, priority, action, argument, kwargs.

`queue` — A synchronized queue class

Source code: Lib/queue.py
[https://github.com/python/cpython/tree/3.14/Lib/queue.py]

The **`queue`** module implements multi-producer, multi-consumer queues. It is especially useful in threaded programming when information must be exchanged safely between multiple threads. The **`Queue`** class in this module implements all the required locking semantics.

The module implements three types of queue, which differ only in the order in which the entries are retrieved. In a FIFO queue, the first tasks added are the first retrieved. In a LIFO queue, the most recently added entry is the first retrieved (operating like a stack). With a priority queue, the entries are kept sorted (using the **`heapq`** module) and the lowest valued entry is retrieved first.

Internally, those three types of queues use locks to temporarily block competing threads; however, they are not designed to handle reentrancy within a thread.

In addition, the module implements a “simple” FIFO queue type, **`SimpleQueue`**, whose specific implementation provides additional guarantees in exchange for the smaller functionality.

The **`queue`** module defines the following classes and exceptions:

class queue.Queue(*maxsize=0*)

Constructor for a FIFO queue. *maxsize* is an integer that sets the upperbound limit on the number of items that can be placed in the queue. Insertion will block once this size has been reached, until queue items are consumed. If *maxsize* is less than or equal to zero, the queue size is infinite.

class queue.LifoQueue(*maxsize=0*)

Constructor for a LIFO queue. *maxsize* is an integer that sets the upperbound limit on the number of items that can be placed in the queue. Insertion will block once this size has been reached, until queue items are consumed. If *maxsize* is less than or equal to zero, the queue size is infinite.

class queue.PriorityQueue(*maxsize=0*)

Constructor for a priority queue. *maxsize* is an integer that sets the upperbound limit on the number of items that can be placed in the queue. Insertion will block once this size has been reached, until queue items are consumed. If *maxsize* is less than or equal to zero, the queue size is infinite.

The lowest valued entries are retrieved first (the lowest valued entry is the one that would be returned by `min(entries)`). A typical pattern for entries is a tuple in the form: `(priority_number, data)`.

If the *data* elements are not comparable, the data can be wrapped in a class that ignores the data item and only compares the priority number:

```
from dataclasses import dataclass, field
from typing import Any
```

```
@dataclass(order=True)
class PrioritizedItem:
    priority: int
    item: Any=field(compare=False)
```

class queue.SimpleQueue

Constructor for an unbounded FIFO queue. Simple queues lack advanced functionality such as task tracking.

Added in version 3.7.

exception queue.Empty

Exception raised when non-blocking **`get()`** (or **`get_nowait()`**) is called on a **`Queue`** object which is empty.

exception queue.Full

Exception raised when non-blocking **`put()`** (or **`put_nowait()`**) is called on a **`Queue`** object which is full.

exception queue.ShutDown

Exception raised when **`put()`** or **`get()`** is called on a **`Queue`** object which has been shut down.

Added in version 3.13.

Queue Objects

Queue objects (**`Queue`**, **`LifoQueue`**, or **`PriorityQueue`**) provide the public methods described below.

Queue.qsize()

Return the approximate size of the queue. Note, qsize() > 0 doesn’t guarantee that a subsequent get() will not block, nor will qsize() < maxsize guarantee that put() will not block.

Queue.empty()

Return `True` if the queue is empty, `False` otherwise. If empty() returns `True` it doesn’t guarantee that a subsequent call to put() will not block. Similarly, if empty() returns `False` it doesn’t guarantee that a subsequent call to get() will not block.

Queue.full()

Return `True` if the queue is full, `False` otherwise. If full() returns `True` it doesn’t guarantee that a subsequent call to get() will not block. Similarly, if full() returns `False` it doesn’t guarantee that a subsequent call to put() will not block.

Queue.put(*item*, *block=True*, *timeout=None*)

Put *item* into the queue. If optional args *block* is true and *timeout* is `None` (the default), block if necessary until a free slot is available. If *timeout* is a positive number, it blocks at most *timeout* seconds and raises the **`Full`** exception if no free slot was available within that time. Otherwise (*block* is false), put an item on the queue if a free slot is immediately available, else raise the **`Full`** exception (*timeout* is ignored in that case).

Raises **`ShutDown`** if the queue has been shut down.

Queue.put_nowait(*item*)

Equivalent to `put(item, block=False)`.

Queue.get(*block=True*, *timeout=None*)

Remove and return an item from the queue. If optional args *block* is true and *timeout* is `None` (the default), block if necessary until an item is available. If *timeout* is a positive number, it blocks at most *timeout* seconds and raises the **`Empty`** exception if no item was available within that time. Otherwise (*block* is false), return an item if one is immediately available, else raise the `Empty` exception (*timeout* is ignored in that case).

Prior to 3.0 on POSIX systems, and for all versions on Windows, if *block* is true and *timeout* is `None`, this operation goes into an uninterruptible wait on an underlying lock. This means that no exceptions can occur, and in particular a SIGINT will not trigger a `KeyboardInterrupt`.

Raises `ShutDown` if the queue has been shut down and is empty, or if the queue has been shut down immediately.

Queue.get_nowait()

Equivalent to `get(False)`.

Two methods are offered to support tracking whether enqueued tasks have been fully processed by daemon consumer threads.

Queue.task_done()

Indicate that a formerly enqueued task is complete. Used by queue consumer threads. For each `get()` used to fetch a task, a subsequent call to `task_done()` tells the queue that the processing on the task is complete.

If a `join()` is currently blocking, it will resume when all items have been processed (meaning that a `task_done()` call was received for every item that had been `put()` into the queue).

Raises a `ValueError` if called more times than there were items placed in the queue.

Queue.join()

Blocks until all items in the queue have been gotten and processed.

The count of unfinished tasks goes up whenever an item is added to the queue. The count goes down whenever a consumer thread calls `task_done()` to indicate that the item was retrieved and all work on it is complete. When the count of unfinished tasks drops to zero, `join()` unblocks.

Waiting for task completion

Example of how to wait for enqueued tasks to be completed:

```
import threading
import queue

q = queue.Queue()

def worker():
    while True:
        item = q.get()
        print(f'Working on {item}')
        print(f'Finished {item}')
        q.task_done()

# Turn-on the worker thread.
threading.Thread(target=worker, daemon=True).start()

# Send thirty task requests to the worker.
for item in range(30):
    q.put(item)

# Block until all tasks are done.
q.join()
print('All work completed')
```

Terminating queues

When no longer needed, `Queue` objects can be wound down until empty or terminated immediately with a hard shutdown.

Queue.shutdown(*immediate=False*)

Put a `Queue` instance into a shutdown mode.

The queue can no longer grow. Future calls to `put()` raise `ShutDown`. Currently blocked callers of `put()` will be unblocked and will raise `ShutDown` in the formerly blocked thread.

If *immediate* is false (the default), the queue can be wound down normally with `get()` calls to extract tasks that have already been loaded.

And if `task_done()` is called for each remaining task, a pending `join()` will be unblocked normally.

Once the queue is empty, future calls to `get()` will raise `ShutDown`.

If *immediate* is true, the queue is terminated immediately. The queue is drained to be completely empty and the count of unfinished tasks is reduced by the number of tasks drained. If unfinished tasks is zero, callers of `join()` are unblocked. Also, blocked callers of `get()` are unblocked and will raise `ShutDown` because the queue is empty.

Use caution when using `join()` with *immediate* set to true. This unblocks the join even when no work has been done on the tasks, violating the usual invariant for joining a queue.

Added in version 3.13.

SimpleQueue Objects

`SimpleQueue` objects provide the public methods described below.

SimpleQueue.qsize()

Return the approximate size of the queue. Note, qsize() > 0 doesn’t guarantee that a subsequent get() will not block.

SimpleQueue.empty()

Return `True` if the queue is empty, `False` otherwise. If empty() returns `False` it doesn’t guarantee that a subsequent call to get() will not block.

SimpleQueue.put(*item*, *block=True*, *timeout=None*)

Put *item* into the queue. The method never blocks and always succeeds (except for potential low-level errors such as failure to allocate memory). The optional args *block* and *timeout* are ignored and only provided for compatibility with `Queue.put()`.

CPython implementation detail: This method has a C implementation which is reentrant. That is, a `put()` or `get()` call can be interrupted by another `put()` call in the same thread without deadlocking or corrupting internal state inside the queue. This makes it appropriate for use in destructors such as `__del__` methods or `weakref` callbacks.

SimpleQueue.put_nowait(*item*)

Equivalent to `put(item, block=False)`, provided for compatibility with `Queue.put_nowait()`.

SimpleQueue.get(*block=True*, *timeout=None*)

Remove and return an item from the queue. If optional args *block* is true and *timeout* is `None` (the default), block if necessary until an item is available. If *timeout* is a positive number, it blocks at most *timeout* seconds and raises the **`Empty`** exception if no item was available within that time. Otherwise (*block* is false), return an item if one is immediately available, else raise the **`Empty`** exception (*timeout* is ignored in that case).

SimpleQueue.get_nowait()

Equivalent to `get(False)`.

See also

Class **`multiprocessing.Queue`**

A queue class for use in a multi-processing (rather than multi-threading) context.

`collections.deque` is an alternative implementation of unbounded queues with fast atomic **`append()`** and **`popleft()`** operations that do not require locking and also support indexing.

`contextvars` — Context Variables

This module provides APIs to manage, store, and access context-local state. The `ContextVar` class is used to declare and work with *Context Variables*. The `copy_context()` function and the `Context` class should be used to manage the current context in asynchronous frameworks.

Context managers that have state should use Context Variables instead of `threading.local()` to prevent their state from bleeding to other code unexpectedly, when used in concurrent code.

See also **PEP 567** [https://peps.python.org/pep-0567/] for additional details.

Added in version 3.7.

Context Variables

class contextvars.ContextVar(*name*[, *, *default*])

This class is used to declare a new Context Variable, e.g.:

```
var: ContextVar[int] = ContextVar('var', default=42)
```

The required *name* parameter is used for introspection and debug purposes.

The optional keyword-only *default* parameter is returned by `ContextVar.get()` when no value for the variable is found in the current context.

Important: Context Variables should be created at the top module level and never in closures. `Context` objects hold strong references to context variables which prevents context variables from being properly garbage collected.

name

The name of the variable. This is a read-only property.

Added in version 3.7.1.

get([*default*])

Return a value for the context variable for the current context.

If there is no value for the variable in the current context, the method will:

- return the value of the *default* argument of the method, if provided; or
- return the default value for the context variable, if it was created with one; or
- raise a `LookupError`.

set(*value*)

Call to set a new value for the context variable in the current context.

The required *value* argument is the new value for the context variable.

Returns a `Token` object that can be used to restore the variable to its previous value via the `ContextVar.reset()` method.

reset(*token*)

Reset the context variable to the value it had before the **`ContextVar.set()`** that created the *token* was used.

For example:

```
var = ContextVar('var')

token = var.set('new value')
# code that uses 'var'; var.get() returns 'new value'.
var.reset(token)

# After the reset call the var has no value again, so
# var.get() would raise a LookupError.
```

class contextvars.Token

Token objects are returned by the **`ContextVar.set()`** method. They can be passed to the **`ContextVar.reset()`** method to revert the value of the variable to what it was before the corresponding *set*.

The token supports context manager protocol to restore the corresponding context variable value at the exit from **`with`** block:

```
var = ContextVar('var', default='default value')

with var.set('new value'):
    assert var.get() == 'new value'

assert var.get() == 'default value'
```

Added in version 3.14: Added support for usage as a context manager.

var

A read-only property. Points to the **`ContextVar`** object that created the token.

old_value

A read-only property. Set to the value the variable had before the `ContextVar.set()` method call that created the token. It points to `Token.MISSING` if the variable was not set before the call.

MISSING

A marker object used by `Token.old_value`.

Manual Context Management

contextvars.copy_context()

Returns a copy of the current `Context` object.

The following snippet gets a copy of the current context and prints all variables and their values that are set in it:

```
ctx: Context = copy_context()
print(list(ctx.items()))
```

The function has an $O(1)$ complexity, i.e. works equally fast for contexts with a few context variables and for contexts that have a lot of them.

class contextvars.Context

A mapping of `ContextVars` to their values.

`Context()` creates an empty context with no values in it. To get a copy of the current context use the `copy_context()` function.

Each thread has its own effective stack of **`Context`** objects. The current context is the **`Context`** object at the top of the current thread’s stack. All **`Context`** objects in the stacks are considered to be *entered*.

Entering a context, which can be done by calling its `run()` method, makes the context the current context by pushing it onto the top of the current thread’s context stack.

Exiting from the current context, which can be done by returning from the callback passed to the `run()` method, restores the current context to what it was before the context was entered by popping the context off the top of the context stack.

Since each thread has its own context stack, `ContextVar` objects behave in a similar fashion to `threading.local()` when values are assigned in different threads.

Attempting to enter an already entered context, including contexts entered in other threads, raises a `RuntimeError`.

After exiting a context, it can later be re-entered (from any thread).

Any changes to `ContextVar` values via the `ContextVar.set()` method are recorded in the current context. The `ContextVar.get()` method returns the value associated with the current context. Exiting a context effectively reverts any changes made to context variables while the context was entered (if needed, the values can be restored by re-entering the context).

Context implements the `collections.abc.Mapping` interface.

run(*callable*, **args*, ***kwargs*)

Enters the Context, executes `callable(*args, **kwargs)`, then exits the Context. Returns *callable*’s return value, or propagates an exception if one occurred.

Example:

```
import contextvars

var = contextvars.ContextVar('var')
var.set('spam')
print(var.get())  # 'spam'

ctx = contextvars.copy_context()

def main():
    # 'var' was set to 'spam' before
    # calling 'copy_context()' and 'ctx.run(main)', so:
    print(var.get())  # 'spam'
    print(ctx[var])  # 'spam'

    var.set('ham')

    # Now, after setting 'var' to 'ham':
    print(var.get())  # 'ham'
    print(ctx[var])  # 'ham'

# Any changes that the 'main' function makes to 'var'
# will be contained in 'ctx'.
ctx.run(main)

# The 'main()' function was run in the 'ctx' context,
# so changes to 'var' are contained in it:
print(ctx[var])  # 'ham'

# However, outside of 'ctx', 'var' is still set to
'spam':
print(var.get())  # 'spam'
```

copy()

Return a shallow copy of the context object.

var in context

Return `True` if the *context* has a value for *var* set; return `False` otherwise.

context[var]

Return the value of the *var* `ContextVar` variable. If the variable is not set in the context object, a `KeyError` is raised.

get(*var*[, *default*])

Return the value for *var* if *var* has the value in the context object. Return *default* otherwise. If *default* is not given, return `None`.

iter(context)

Return an iterator over the variables stored in the context object.

len(proxy)

Return the number of variables set in the context object.

keys()

Return a list of all variables in the context object.

values()

Return a list of all variables’ values in the context object.

items()

Return a list of 2-tuples containing all variables and their values in the context object.

asyncio support

Context variables are natively supported in `asyncio` and are ready to be used without any extra configuration. For example, here is a simple echo server, that uses a context variable to make the address of a remote client available in the Task that handles that client:

```
import asyncio
import contextvars

client_addr_var = contextvars.ContextVar('client_addr')

def render_goodbye():
    # The address of the currently handled client can be
accessed
    # without passing it explicitly to this function.

    client_addr = client_addr_var.get()
    return f'Good bye, client @ {client_addr}\r\n'.encode()

async def handle_request(reader, writer):
    addr =
writer.transport.get_extra_info('socket').getpeername()
    client_addr_var.set(addr)

    # In any code that we call is now possible to get
    # client's address by calling 'client_addr_var.get()'.

    while True:
        line = await reader.readline()
        print(line)
        if not line.strip():
            break

    writer.write(b'HTTP/1.1 200 OK\r\n')  # status line
    writer.write(b'\r\n')  # headers
    writer.write(render_goodbye())  # body
    writer.close()

async def main():
    srv = await asyncio.start_server(
        handle_request, '127.0.0.1', 8081)

    async with srv:
        await srv.serve_forever()

asyncio.run(main())

# To test it you can use telnet or curl:
```

```
#     telnet 127.0.0.1 8081
#     curl 127.0.0.1:8081
```

`_thread` — Low-level threading API

This module provides low-level primitives for working with multiple threads (also called *light-weight processes* or *tasks*) — multiple threads of control sharing their global data space. For synchronization, simple locks (also called *mutexes* or *binary semaphores*) are provided. The `threading` module provides an easier to use and higher-level threading API built on top of this module.

Changed in version 3.7: This module used to be optional, it is now always available.

This module defines the following constants and functions:

exception _thread.error

Raised on thread-specific errors.

Changed in version 3.3: This is now a synonym of the built-in `RuntimeError`.

_thread.LockType

This is the type of lock objects.

_thread.start_new_thread(*function*, *args*[, *kwargs*])

Start a new thread and return its identifier. The thread executes the function *function* with the argument list *args* (which must be a tuple). The optional *kwargs* argument specifies a dictionary of keyword arguments.

When the function returns, the thread silently exits.

When the function terminates with an unhandled exception, `sys.unraisablehook()` is called to handle the exception. The *object* attribute of the hook argument is *function*. By default, a stack trace is printed and then the thread exits (but other threads continue to run).

When the function raises a `SystemExit` exception, it is silently ignored.

Raises an auditing event `_thread.start_new_thread` with arguments `function`, `args`, `kwargs`.

Changed in version 3.8: `sys.unraisablehook()` is now used to handle unhandled exceptions.

_thread.interrupt_main(*signum=signal.SIGINT*, */*)

Simulate the effect of a signal arriving in the main thread. A thread can use this function to interrupt the main thread, though there is no guarantee that the interruption will happen immediately.

If given, *signum* is the number of the signal to simulate. If *signum* is not given, `signal.SIGINT` is simulated.

If the given signal isn’t handled by Python (it was set to `signal.SIG_DFL` or `signal.SIG_IGN`), this function does nothing.

Changed in version 3.10: The *signum* argument is added to customize the signal number.

Note

This does not emit the corresponding signal but schedules a call to the associated handler (if it exists). If you want to truly emit the signal, use `signal.raise_signal()`.

_thread.exit()

Raise the `SystemExit` exception. When not caught, this will cause the thread to exit silently.

_thread.allocate_lock()

Return a new lock object. Methods of locks are described below. The lock is initially unlocked.

_thread.get_ident()

Return the ‘thread identifier’ of the current thread. This is a nonzero integer. Its value has no direct meaning; it is intended as a magic cookie to be used e.g. to index a dictionary of thread-specific data. Thread identifiers may be recycled when a thread exits and another thread is created.

_thread.get_native_id()

Return the native integral Thread ID of the current thread assigned by the kernel. This is a non-negative integer. Its value may be used to uniquely identify this particular thread system-wide (until the thread terminates, after which the value may be recycled by the OS).

Availability: Windows, FreeBSD, Linux, macOS, OpenBSD, NetBSD, AIX, DragonFlyBSD, GNU/kFreeBSD.

Added in version 3.8.

Changed in version 3.13: Added support for GNU/kFreeBSD.

_thread.stack_size([*size*])

Return the thread stack size used when creating new threads. The optional *size* argument specifies the stack size to be used for subsequently created threads, and must be 0 (use platform or configured default) or a positive integer value of at least 32,768 (32 KiB). If *size* is not specified, 0 is used. If changing the thread stack size is unsupported, a `RuntimeError` is raised. If the specified stack size is invalid, a `ValueError` is raised and the stack size is unmodified. 32 KiB is currently the minimum supported stack size value to guarantee sufficient stack space for the interpreter itself. Note that some platforms may have particular restrictions on values for the stack size, such as requiring a minimum stack size > 32 KiB or requiring allocation in multiples of the system memory page size - platform documentation should be referred to for more information (4 KiB pages are common; using multiples of 4096 for the stack size is the suggested approach in the absence of more specific information).

Availability: Windows, pthreads.

Unix platforms with POSIX threads support.

_thread.TIMEOUT_MAX

The maximum value allowed for the *timeout* parameter of `Lock.acquire`. Specifying a timeout greater than this value will raise an `OverflowError`.

Added in version 3.2.

Lock objects have the following methods:

lock.acquire(*blocking=True*, *timeout=-1*)

Without any optional argument, this method acquires the lock unconditionally, if necessary waiting until it is released by another thread (only one thread at a time can acquire a lock — that’s their reason for existence).

If the *blocking* argument is present, the action depends on its value: if it is false, the lock is only acquired if it can be acquired immediately without waiting, while if it is true, the lock is acquired unconditionally as above.

If the floating-point *timeout* argument is present and positive, it specifies the maximum wait time in seconds before returning. A negative *timeout* argument specifies an unbounded wait. You cannot specify a *timeout* if *blocking* is false.

The return value is `True` if the lock is acquired successfully, `False` if not.

Changed in version 3.2: The *timeout* parameter is new.

Changed in version 3.2: Lock acquires can now be interrupted by signals on POSIX.

Changed in version 3.14: Lock acquires can now be interrupted by signals on Windows.

lock.release()

Releases the lock. The lock must have been acquired earlier, but not necessarily by the same thread.

lock.locked()

Return the status of the lock: `True` if it has been acquired by some thread, `False` if not.

In addition to these methods, lock objects can also be used via the `with` statement, e.g.:

```
import _thread

a_lock = _thread.allocate_lock()

with a_lock:
    print("a_lock is locked while this executes")
```

Caveats:

- Interrupts always go to the main thread (the `KeyboardInterrupt` exception will be received by that thread.)
- Calling `sys.exit()` or raising the `SystemExit` exception is equivalent to calling `_thread.exit()`.
- When the main thread exits, it is system defined whether the other threads survive. On most systems, they are killed without executing `try` … `finally` clauses or executing object destructors.

Networking and Interprocess Communication

The modules described in this chapter provide mechanisms for networking and inter-processes communication.

Some modules only work for two processes that are on the same machine, e.g. `signal` and `mmap`. Other modules support networking protocols that two or more processes can use to communicate across machines.

The list of modules described in this chapter is:

- `asyncio` — Asynchronous I/O
- `socket` — Low-level networking interface
- `ssl` — TLS/SSL wrapper for socket objects
- `select` — Waiting for I/O completion
- `selectors` — High-level I/O multiplexing
- `signal` — Set handlers for asynchronous events
- `mmap` — Memory-mapped file support

`asyncio` — Asynchronous I/O

asyncio is a library to write **concurrent** code using the **async/await** syntax.

asyncio is used as a foundation for multiple Python asynchronous frameworks that provide high-performance network and web-servers, database connection libraries, distributed task queues, etc.

Hello World!

```
import asyncio

async def main():
    print('Hello ...')
    await
asyncio.sleep(1)
    print('... World!')

asyncio.run(main())
```

asyncio is often a perfect fit for IO-bound and high-level **structured** network code.

See also

A Conceptual Overview of asyncio
Explanation of the fundamentals of asyncio.

asyncio provides a set of **high-level** APIs to:

- run Python coroutines concurrently and have full control over their execution;
- perform network IO and IPC;
- control subprocesses;
- distribute tasks via queues;
- synchronize concurrent code;

Additionally, there are **low-level** APIs for *library and framework developers* to:

- create and manage event loops, which provide asynchronous APIs for networking, running subprocesses, handling OS signals, etc;
- implement efficient protocols using transports;
- bridge callback-based libraries and code with async/await syntax.

Availability: not WASI.

This module does not work or is not available on WebAssembly. See WebAssembly platforms for more information.

asyncio REPL

You can experiment with an `asyncio` concurrent context in the REPL:

```
$ python -m asyncio
asyncio REPL ...
Use "await" directly instead of "asyncio.run()".
Type "help", "copyright", "credits" or "license" for more
information.
>>> import asyncio
>>> await asyncio.sleep(10, result='hello')
'hello'
```

This REPL provides limited compatibility with **`PYTHON_BASIC_REPL`**. It is recommended that the default REPL is used for full functionality and the latest features.

Raises an auditing event `cpython.run_stdin` with no arguments.

Changed in version 3.12.5: (also 3.11.10, 3.10.15, 3.9.20, and 3.8.20) Emits audit events.

Changed in version 3.13: Uses PyREPL if possible, in which case **`PYTHONSTARTUP`** is also executed. Emits audit events.

Reference

High-level APIs

- Runners
- Coroutines and Tasks
- Streams
- Synchronization Primitives
- Subprocesses
- Queues
- Exceptions
- Call Graph Introspection

Low-level APIs

- Event Loop
- Futures
- Transports and Protocols
- Policies
- Platform Support
- Extending

Guides and Tutorials

- High-level API Index
- Low-level API Index
- Developing with asyncio

Note

The source code for asyncio can be found in Lib/asyncio/ [https://github.com/python/cpython/tree/3.14/Lib/asyncio/].

Runners

Source code: Lib/asyncio/runners.py [https://github.com/python/cpython/tree/3.14/Lib/asyncio/runners.py]

This section outlines high-level asyncio primitives to run asyncio code.

They are built on top of an event loop with the aim to simplify async code usage for common wide-spread scenarios.

- Running an asyncio Program
- Runner context manager
- Handling Keyboard Interruption

Running an asyncio Program

asyncio.run(*coro*, *, *debug=None*, *loop_factory=None*)

Execute *coro* in an asyncio event loop and return the result.

The argument can be any awaitable object.

This function runs the awaitable, taking care of managing the asyncio event loop, *finalizing asynchronous generators*, and closing the executor.

This function cannot be called when another asyncio event loop is running in the same thread.

If *debug* is `True`, the event loop will be run in debug mode. `False` disables debug mode explicitly. `None` is used to respect the global Debug Mode settings.

If *loop_factory* is not `None`, it is used to create a new event loop; otherwise **`asyncio.new_event_loop()`** is used. The loop is closed at the end. This function should be used as a main entry point for asyncio programs, and should ideally only be called once. It is recommended to use *loop_factory* to configure the event loop instead of policies. Passing **`asyncio.EventLoop`** allows running asyncio without the policy system.

The executor is given a timeout duration of 5 minutes to shutdown. If the executor hasn't finished within that duration, a warning is emitted and the executor is closed.

Example:

```
async def main():
    await asyncio.sleep(1)
    print('hello')

asyncio.run(main())
```

Added in version 3.7.

Changed in version 3.9: Updated to use **`loop.shutdown_default_executor()`**.

Changed in version 3.10: debug is `None` by default to respect the global debug mode settings.

Changed in version 3.12: Added *loop_factory* parameter.

Changed in version 3.14: coro can be any awaitable object.

Note

The **`asyncio`** policy system is deprecated and will be removed in Python 3.16; from there on, an explicit *loop_factory* is needed to configure the event loop.

Runner context manager

class asyncio.Runner(*, *debug=None*, *loop_factory=None*)

A context manager that simplifies *multiple* async function calls in the same context.

Sometimes several top-level async functions should be called in the same event loop and `contextvars.Context`.

If *debug* is `True`, the event loop will be run in debug mode. `False` disables debug mode explicitly. `None` is used to respect the global Debug Mode settings.

loop_factory could be used for overriding the loop creation. It is the responsibility of the *loop_factory* to set the created loop as the current one. By default `asyncio.new_event_loop()` is used and set as current event loop with `asyncio.set_event_loop()` if *loop_factory* is `None`.

Basically, `asyncio.run()` example can be rewritten with the runner usage:

```
async def main():
    await asyncio.sleep(1)
    print('hello')

with asyncio.Runner() as runner:
    runner.run(main())
```

Added in version 3.11.

run(*coro*, *, *context=None*)

Execute *coro* in the embedded event loop.

The argument can be any awaitable object.

If the argument is a coroutine, it is wrapped in a Task.

An optional keyword-only *context* argument allows specifying a custom `contextvars.Context` for the code to run in. The runner's default context is used if context is `None`.

Returns the awaitable's result or raises an exception.

This function cannot be called when another asyncio event loop is running in the same thread.

Changed in version 3.14: coro can be any awaitable object.

close()

Close the runner.

Finalize asynchronous generators, shutdown default executor, close the event loop and release embedded `contextvars.Context`.

get_loop()

Return the event loop associated with the runner instance.

Note

`Runner` uses the lazy initialization strategy, its constructor doesn't initialize underlying low-level structures.

Embedded *loop* and *context* are created at the `with` body entering or the first call of `run()` or `get_loop()`.

Handling Keyboard Interruption

Added in version 3.11.

When `signal.SIGINT` is raised by `Ctrl-C`, `KeyboardInterrupt` exception is raised in the main thread by default. However this doesn't work with `asyncio` because it can interrupt asyncio internals and can hang the program from exiting.

To mitigate this issue, `asyncio` handles `signal.SIGINT` as follows:

1. `asyncio.Runner.run()` installs a custom `signal.SIGINT` handler before any user code is executed and removes it when exiting from the function.
2. The `Runner` creates the main task for the passed coroutine for its execution.
3. When `signal.SIGINT` is raised by `Ctrl-C`, the custom signal handler cancels the main task by calling `asyncio.Task.cancel()` which raises `asyncio.CancelledError` inside the main task. This causes the Python stack to unwind, `try/except` and `try/finally` blocks can be used for resource cleanup. After the main task is cancelled, `asyncio.Runner.run()` raises `KeyboardInterrupt`.
4. A user could write a tight loop which cannot be interrupted by `asyncio.Task.cancel()`, in which case the second following `Ctrl-C` immediately raises the `KeyboardInterrupt` without cancelling the main task.

Coroutines and Tasks

This section outlines high-level asyncio APIs to work with coroutines and Tasks.

- Coroutines
- Awaitables
- Creating Tasks
- Task Cancellation
- Task Groups
- Sleeping
- Running Tasks Concurrently
- Eager Task Factory
- Shielding From Cancellation
- Timeouts
- Waiting Primitives
- Running in Threads
- Scheduling From Other Threads
- Introspection
- Task Object

Coroutines

Source code: Lib/asyncio/coroutines.py [https://github.com/python/cpython/tree/3.14/Lib/asyncio/coroutines.py]

Coroutines declared with the async/await syntax is the preferred way of writing asyncio applications. For example, the following snippet of code prints “hello”, waits 1 second, and then prints “world”:

```
>>> import asyncio

>>> async def main():
...     print('hello')
...     await asyncio.sleep(1)
...     print('world')

>>> asyncio.run(main())
hello
world
```

Note that simply calling a coroutine will not schedule it to be executed:

```
>>> main()
<coroutine object main at 0x1053bb7c8>
```

To actually run a coroutine, asyncio provides the following mechanisms:

- The `asyncio.run()` function to run the top-level entry point “main()” function (see the above example.)

- Awaiting on a coroutine. The following snippet of code will print “hello” after waiting for 1 second, and then print “world” after waiting for *another* 2 seconds:

  ```
  import asyncio
  import time

  async def say_after(delay, what):
      await asyncio.sleep(delay)
      print(what)

  async def main():
      print(f"started at {time.strftime('%X')}")

      await say_after(1, 'hello')
      await say_after(2, 'world')

      print(f"finished at {time.strftime('%X')}")
  ```

```
asyncio.run(main())
```

Expected output:

```
started at 17:13:52
hello
world
finished at 17:13:55
```

- The `asyncio.create_task()` function to run coroutines concurrently as asyncio `Tasks`.

  Let’s modify the above example and run two `say_after` coroutines *concurrently*:

  ```
  async def main():
      task1 = asyncio.create_task(
          say_after(1, 'hello'))

      task2 = asyncio.create_task(
          say_after(2, 'world'))

      print(f"started at {time.strftime('%X')}")

      # Wait until both tasks are completed (should take
      # around 2 seconds.)
      await task1
      await task2

      print(f"finished at {time.strftime('%X')}")
  ```

  Note that expected output now shows that the snippet runs 1 second faster than before:

  ```
  started at 17:14:32
  hello
  world
  finished at 17:14:34
  ```

- The `asyncio.TaskGroup` class provides a more modern alternative to `create_task()`. Using this API, the last example becomes:

  ```
  async def main():
      async with asyncio.TaskGroup() as tg:
          task1 = tg.create_task(
              say_after(1, 'hello'))

          task2 = tg.create_task(
              say_after(2, 'world'))

          print(f"started at {time.strftime('%X')}")

      # The await is implicit when the context manager exits.

      print(f"finished at {time.strftime('%X')}")
  ```

  The timing and output should be the same as for the previous version.

  Added in version 3.11: `asyncio.TaskGroup`.

Awaitables

We say that an object is an **awaitable** object if it can be used in an `await` expression. Many asyncio APIs are designed to accept awaitables.

There are three main types of *awaitable* objects: **coroutines**, **Tasks**, and **Futures**.

Coroutines

Python coroutines are *awaitables* and therefore can be awaited from other coroutines:

```
import asyncio

async def nested():
    return 42

async def main():
    # Nothing happens if we just call "nested()".
    # A coroutine object is created but not awaited,
    # so it *won't run at all*.
    nested()  # will raise a "RuntimeWarning".

    # Let's do it differently now and await it:
    print(await nested())  # will print "42".

asyncio.run(main())
```

Important

In this documentation the term “coroutine” can be used for two closely related concepts:

- a *coroutine function*: an `async def` function;
- a *coroutine object*: an object returned by calling a *coroutine function*.

Tasks

Tasks are used to schedule coroutines *concurrently*.

When a coroutine is wrapped into a *Task* with functions like `asyncio.create_task()` the coroutine is automatically scheduled to run soon:

```
import asyncio

async def nested():
    return 42

async def main():
    # Schedule nested() to run soon concurrently
    # with "main()".
    task = asyncio.create_task(nested())

    # "task" can now be used to cancel "nested()", or
    # can simply be awaited to wait until it is complete:
    await task

asyncio.run(main())
```

Futures

A **`Future`** is a special **low-level** awaitable object that represents an **eventual result** of an asynchronous operation.

When a Future object is *awaited* it means that the coroutine will wait until the Future is resolved in some other place.

Future objects in asyncio are needed to allow callback-based code to be used with async/await.

Normally **there is no need** to create Future objects at the application level code.

Future objects, sometimes exposed by libraries and some asyncio APIs, can be awaited:

```
async def main():
    await function_that_returns_a_future_object()

    # this is also valid:
    await asyncio.gather(
        function_that_returns_a_future_object(),
        some_python_coroutine()
    )
```

A good example of a low-level function that returns a Future object is **`loop.run_in_executor()`**.

Creating Tasks

Source code: Lib/asyncio/tasks.py
[https://github.com/python/cpython/tree/3.14/Lib/asyncio/tasks.py]

---

asyncio.create_task(*coro*, *, *name=None*, *context=None*, *eager_start=None*, ***kwargs*)

Wrap the *coro* coroutine into a `Task` and schedule its execution. Return the Task object.

The full function signature is largely the same as that of the `Task` constructor (or factory) - all of the keyword arguments to this function are passed through to that interface.

An optional keyword-only *context* argument allows specifying a custom `contextvars.Context` for the *coro* to run in. The current context copy is created when no *context* is provided.

An optional keyword-only *eager_start* argument allows specifying if the task should execute eagerly during the call to create_task, or be scheduled later. If *eager_start* is not passed the mode set by `loop.set_task_factory()` will be used.

The task is executed in the loop returned by `get_running_loop()`, `RuntimeError` is raised if there is no running loop in current thread.

Note

`asyncio.TaskGroup.create_task()` is a new alternative leveraging structural concurrency; it allows for waiting for a group of related tasks with strong safety guarantees.

Important

Save a reference to the result of this function, to avoid a task disappearing mid-execution. The event loop only keeps weak references to tasks. A task that isn’t referenced elsewhere may get garbage collected at any time, even before it’s done. For reliable “fire-and-forget” background tasks, gather them in a collection:

```
background_tasks = set()

for i in range(10):
    task = asyncio.create_task(some_coro(param=i))

    # Add task to the set. This creates a strong
reference.
    background_tasks.add(task)

    # To prevent keeping references to finished tasks
forever,
    # make each task remove its own reference from the set
after
    # completion:
    task.add_done_callback(background_tasks.discard)
```

Added in version 3.7.

Changed in version 3.8: Added the *name* parameter.

Changed in version 3.11: Added the *context* parameter.

Changed in version 3.14: Added the *eager_start* parameter by passing on all *kwargs*.

Task Cancellation

Tasks can easily and safely be cancelled. When a task is cancelled, `asyncio.CancelledError` will be raised in the task at the next opportunity.

It is recommended that coroutines use `try/finally` blocks to robustly perform clean-up logic. In case `asyncio.CancelledError` is explicitly caught, it should generally be propagated when clean-up is complete. `asyncio.CancelledError` directly subclasses `BaseException` so most code will not need to be aware of it.

The asyncio components that enable structured concurrency, like `asyncio.TaskGroup` and `asyncio.timeout()`, are implemented using cancellation internally and might misbehave if a coroutine swallows `asyncio.CancelledError`. Similarly, user code should not generally call `uncancel`. However, in cases when suppressing `asyncio.CancelledError` is truly desired, it is necessary to also call `uncancel()` to completely remove the cancellation state.

Task Groups

Task groups combine a task creation API with a convenient and reliable way to wait for all tasks in the group to finish.

class asyncio.TaskGroup

An asynchronous context manager holding a group of tasks. Tasks can be added to the group using `create_task()`. All tasks are awaited when the context manager exits.

Added in version 3.11.

create_task(*coro*, *, *name=None*, *context=None*, *eager_start=None*, ***kwargs*)

Create a task in this task group. The signature matches that of **`asyncio.create_task()`**. If the task group is inactive (e.g. not yet entered, already finished, or in the process of shutting down), we will close the given `coro`.

Changed in version 3.13: Close the given coroutine if the task group is not active.

Changed in version 3.14: Passes on all *kwargs* to **`loop.create_task()`**

Example:

```
async def main():
    async with asyncio.TaskGroup() as tg:
        task1 = tg.create_task(some_coro(...))
        task2 = tg.create_task(another_coro(...))
    print(f"Both tasks have completed now: {task1.result()},
{task2.result()}")
```

The `async with` statement will wait for all tasks in the group to finish. While waiting, new tasks may still be added to the group (for example, by passing `tg` into one of the coroutines and calling `tg.create_task()` in that coroutine). Once the last task has finished and the `async with` block is exited, no new tasks may be added to the group.

The first time any of the tasks belonging to the group fails with an exception other than **`asyncio.CancelledError`**, the remaining tasks in the group are cancelled. No further tasks can then be added to the group. At this point, if the body of the `async with` statement is still active (i.e., **`__aexit__()`** hasn’t been called yet), the task directly containing the `async with` statement is also cancelled. The resulting **`asyncio.CancelledError`** will interrupt an `await`, but it will not bubble out of the containing `async with` statement.

Once all tasks have finished, if any tasks have failed with an exception other than `asyncio.CancelledError`, those exceptions are combined in an `ExceptionGroup` or `BaseExceptionGroup` (as appropriate; see their documentation) which is then raised.

Two base exceptions are treated specially: If any task fails with `KeyboardInterrupt` or `SystemExit`, the task group still cancels the remaining tasks and waits for them, but then the initial `KeyboardInterrupt` or `SystemExit` is re-raised instead of `ExceptionGroup` or `BaseExceptionGroup`.

If the body of the `async with` statement exits with an exception (so `__aexit__()` is called with an exception set), this is treated the same as if one of the tasks failed: the remaining tasks are cancelled and then waited for, and non-cancellation exceptions are grouped into an exception group and raised. The exception passed into `__aexit__()`, unless it is `asyncio.CancelledError`, is also included in the exception group. The same special case is made for `KeyboardInterrupt` and `SystemExit` as in the previous paragraph.

Task groups are careful not to mix up the internal cancellation used to “wake up” their `__aexit__()` with cancellation requests for the task in which they are running made by other parties. In particular, when one task group is syntactically nested in another, and both experience an exception in one of their child tasks simultaneously, the inner task group will process its exceptions, and then the outer task group will receive another cancellation and process its own exceptions.

In the case where a task group is cancelled externally and also must raise an `ExceptionGroup`, it will call the parent task’s `cancel()` method. This ensures that a `asyncio.CancelledError` will be raised at the next `await`, so the cancellation is not lost.

Task groups preserve the cancellation count reported by `asyncio.Task.cancelling()`.

Changed in version 3.13: Improved handling of simultaneous internal and external cancellations and correct preservation of cancellation counts.

Terminating a Task Group

While terminating a task group is not natively supported by the standard library, termination can be achieved by adding an exception-raising task to the task group and ignoring the raised exception:

```
import asyncio
from asyncio import TaskGroup

class TerminateTaskGroup(Exception):
    """Exception raised to terminate a task group."""

async def force_terminate_task_group():
    """Used to force termination of a task group."""
    raise TerminateTaskGroup()

async def job(task_id, sleep_time):
    print(f'Task {task_id}: start')
    await asyncio.sleep(sleep_time)
    print(f'Task {task_id}: done')

async def main():
    try:
        async with TaskGroup() as group:
            # spawn some tasks
            group.create_task(job(1, 0.5))
            group.create_task(job(2, 1.5))
            # sleep for 1 second
            await asyncio.sleep(1)
            # add an exception-raising task to force the group
to terminate
            group.create_task(force_terminate_task_group())
    except* TerminateTaskGroup:
        pass

asyncio.run(main())
```

Expected output:

```
Task 1: start
Task 2: start
Task 1: done
```

Sleeping

async asyncio.sleep(*delay*, *result=None*)

Block for *delay* seconds.

If *result* is provided, it is returned to the caller when the coroutine completes.

`sleep()` always suspends the current task, allowing other tasks to run.

Setting the delay to 0 provides an optimized path to allow other tasks to run. This can be used by long-running functions to avoid blocking the event loop for the full duration of the function call.

Example of coroutine displaying the current date every second for 5 seconds:

```
import asyncio
import datetime

async def display_date():
    loop = asyncio.get_running_loop()
    end_time = loop.time() + 5.0
    while True:
        print(datetime.datetime.now())
        if (loop.time() + 1.0) >= end_time:
            break
        await asyncio.sleep(1)

asyncio.run(display_date())
```

Changed in version 3.10: Removed the *loop* parameter.

Changed in version 3.13: Raises `ValueError` if *delay* is `nan`.

Running Tasks Concurrently

awaitable asyncio.gather(**aws*, *return_exceptions=False*)

Run awaitable objects in the *aws* sequence *concurrently*.

If any awaitable in *aws* is a coroutine, it is automatically scheduled as a Task.

If all awaitables are completed successfully, the result is an aggregate list of returned values. The order of result values corresponds to the order of awaitables in *aws*.

If *return_exceptions* is `False` (default), the first raised exception is immediately propagated to the task that awaits on `gather()`. Other awaitables in the *aws* sequence **won't be cancelled** and will continue to run.

If *return_exceptions* is `True`, exceptions are treated the same as successful results, and aggregated in the result list.

If `gather()` is *cancelled*, all submitted awaitables (that have not completed yet) are also *cancelled*.

If any Task or Future from the *aws* sequence is *cancelled*, it is treated as if it raised `CancelledError` – the `gather()` call is **not** cancelled in this case. This is to prevent the cancellation of one submitted Task/Future to cause other Tasks/Futures to be cancelled.

Note

A new alternative to create and run tasks concurrently and wait for their completion is `asyncio.TaskGroup`. *TaskGroup* provides stronger safety guarantees than *gather* for scheduling a nesting of subtasks: if a task (or a subtask, a task scheduled by a task) raises an exception, *TaskGroup* will, while *gather* will not, cancel the remaining scheduled tasks).

Example:

```
import asyncio

async def factorial(name, number):
    f = 1
    for i in range(2, number + 1):
        print(f"Task {name}: Compute factorial({number}),
currently i={i}...")
        await asyncio.sleep(1)
        f *= i
    print(f"Task {name}: factorial({number}) = {f}")
    return f

async def main():
    # Schedule three calls *concurrently*:
    L = await asyncio.gather(
        factorial("A", 2),
        factorial("B", 3),
        factorial("C", 4),
    )
    print(L)

asyncio.run(main())

# Expected output:
#
#     Task A: Compute factorial(2), currently i=2...
#     Task B: Compute factorial(3), currently i=2...
#     Task C: Compute factorial(4), currently i=2...
#     Task A: factorial(2) = 2
#     Task B: Compute factorial(3), currently i=3...
#     Task C: Compute factorial(4), currently i=3...
#     Task B: factorial(3) = 6
#     Task C: Compute factorial(4), currently i=4...
#     Task C: factorial(4) = 24
#     [2, 6, 24]
```

Note

If *return_exceptions* is false, cancelling gather() after it has been marked done won't cancel any submitted awaitables. For instance, gather can be marked done after propagating an exception to the caller, therefore, calling `gather.cancel()` after catching an exception (raised by one of the awaitables) from gather won't cancel any other awaitables.

Changed in version 3.7: If the *gather* itself is cancelled, the cancellation is propagated regardless of *return_exceptions*.

Changed in version 3.10: Removed the *loop* parameter.

Deprecated since version 3.10: Deprecation warning is emitted if no positional arguments are provided or not all positional arguments are Future-like objects and there is no running event loop.

Eager Task Factory

asyncio.eager_task_factory(*loop*, *coro*, *, *name=None*, *context=None*)

A task factory for eager task execution.

When using this factory (via `loop.set_task_factory(asyncio.eager_task_factory)`), coroutines begin execution synchronously during `Task` construction. Tasks are only scheduled on the event loop if they block. This can be a performance improvement as the overhead of loop scheduling is avoided for coroutines that complete synchronously.

A common example where this is beneficial is coroutines which employ caching or memoization to avoid actual I/O when possible.

Note

Immediate execution of the coroutine is a semantic change. If the coroutine returns or raises, the task is never scheduled to the event loop. If the coroutine execution blocks, the task is scheduled to the event loop. This change may introduce behavior changes to existing applications. For example, the application’s task execution order is likely to change.

Added in version 3.12.

asyncio.create_eager_task_factory(*custom_task_constructor*)

Create an eager task factory, similar to `eager_task_factory()`, using the provided *custom_task_constructor* when creating a new task instead of the default `Task`.

custom_task_constructor must be a *callable* with the signature matching the signature of `Task.__init__`. The callable must return a `asyncio.Task`-compatible object.

This function returns a *callable* intended to be used as a task factory of an event loop via `loop.set_task_factory(factory)`).

Added in version 3.12.

Shielding From Cancellation

awaitable asyncio.shield(*aw*)

Protect an awaitable object from being **`cancelled`**.

If *aw* is a coroutine it is automatically scheduled as a Task.

The statement:

```
task = asyncio.create_task(something())
res = await shield(task)
```

is equivalent to:

```
res = await something()
```

except that if the coroutine containing it is cancelled, the Task running in `something()` is not cancelled. From the point of view of `something()`, the cancellation did not happen. Although its caller is still cancelled, so the “await” expression still raises a **`CancelledError`**.

If `something()` is cancelled by other means (i.e. from within itself) that would also cancel `shield()`.

If it is desired to completely ignore cancellation (not recommended) the `shield()` function should be combined with a try/except clause, as follows:

```
task = asyncio.create_task(something())
try:
    res = await shield(task)
except CancelledError:
    res = None
```

Important

Save a reference to tasks passed to this function, to avoid a task disappearing mid-execution. The event loop only keeps weak references to tasks. A task that isn’t referenced elsewhere may get garbage collected at any time, even before it’s done.

Changed in version 3.10: Removed the *loop* parameter.

Deprecated since version 3.10: Deprecation warning is emitted if *aw* is not Future-like object and there is no running event loop.

Timeouts

asyncio.timeout(*delay*)

Return an asynchronous context manager that can be used to limit the amount of time spent waiting on something.

delay can either be `None`, or a float/int number of seconds to wait. If *delay* is `None`, no time limit will be applied; this can be useful if the delay is unknown when the context manager is created.

In either case, the context manager can be rescheduled after creation using **`Timeout.reschedule()`**.

Example:

```
async def main():
    async with asyncio.timeout(10):
        await long_running_task()
```

If `long_running_task` takes more than 10 seconds to complete, the context manager will cancel the current task and handle the resulting `asyncio.CancelledError` internally, transforming it into a `TimeoutError` which can be caught and handled.

Note

The `asyncio.timeout()` context manager is what transforms the `asyncio.CancelledError` into a `TimeoutError`, which means the `TimeoutError` can only be caught *outside* of the context manager.

Example of catching `TimeoutError`:

```
async def main():
    try:
        async with asyncio.timeout(10):
            await long_running_task()
    except TimeoutError:
        print("The long operation timed out, but we've
handled it.")

    print("This statement will run regardless.")
```

The context manager produced by `asyncio.timeout()` can be rescheduled to a different deadline and inspected.

class asyncio.Timeout(*when*)

An asynchronous context manager for cancelling overdue coroutines.

`when` should be an absolute time at which the context should time out, as measured by the event loop’s clock:

- If `when` is `None`, the timeout will never trigger.

- If `when < loop.time()`, the timeout will trigger on the next iteration of the event loop.

when() → float | None

Return the current deadline, or `None` if the current deadline is not set.

reschedule(*when: float | None*)

Reschedule the timeout.

expired() → bool

Return whether the context manager has exceeded its deadline (expired).

Example:

```
async def main():
    try:
        # We do not know the timeout when starting, so we
pass ``None``.
        async with asyncio.timeout(None) as cm:
            # We know the timeout now, so we reschedule it.
            new_deadline = get_running_loop().time() + 10
            cm.reschedule(new_deadline)

            await long_running_task()
    except TimeoutError:
        pass

    if cm.expired():
        print("Looks like we haven't finished on time.")
```

Timeout context managers can be safely nested.

Added in version 3.11.

asyncio.timeout_at(*when*)

Similar to **`asyncio.timeout()`**, except *when* is the absolute time to stop waiting, or `None`.

Example:

```
async def main():
    loop = get_running_loop()
    deadline = loop.time() + 20
    try:
        async with asyncio.timeout_at(deadline):
            await long_running_task()
    except TimeoutError:
        print("The long operation timed out, but we've
handled it.")

    print("This statement will run regardless.")
```

Added in version 3.11.

async asyncio.wait_for(*aw*, *timeout*)

Wait for the *aw* awaitable to complete with a timeout.

If *aw* is a coroutine it is automatically scheduled as a Task.

timeout can either be `None` or a float or int number of seconds to wait for. If *timeout* is `None`, block until the future completes.

If a timeout occurs, it cancels the task and raises **`TimeoutError`**.

To avoid the task **`cancellation`**, wrap it in **`shield()`**.

The function will wait until the future is actually cancelled, so the total wait time may exceed the *timeout*. If an exception happens during cancellation, it is propagated.

If the wait is cancelled, the future *aw* is also cancelled.

Example:

```
async def eternity():
    # Sleep for one hour
    await asyncio.sleep(3600)
    print('yay!')

async def main():
    # Wait for at most 1 second
    try:
        await asyncio.wait_for(eternity(), timeout=1.0)
    except TimeoutError:
        print('timeout!')

asyncio.run(main())

# Expected output:
#
#     timeout!
```

Changed in version 3.7: When *aw* is cancelled due to a timeout, `wait_for` waits for *aw* to be cancelled. Previously, it raised `TimeoutError` immediately.

Changed in version 3.10: Removed the *loop* parameter.

Changed in version 3.11: Raises `TimeoutError` instead of `asyncio.TimeoutError`.

Waiting Primitives

async asyncio.wait(*aws*, *, *timeout=None*, *return_when=ALL_COMPLETED*)

Run `Future` and `Task` instances in the *aws* iterable concurrently and block until the condition specified by *return_when*.

The *aws* iterable must not be empty.

Returns two sets of Tasks/Futures: `(done, pending)`.

Usage:

```
done, pending = await asyncio.wait(aws)
```

timeout (a float or int), if specified, can be used to control the maximum number of seconds to wait before returning.

Note that this function does not raise `TimeoutError`. Futures or Tasks that aren’t done when the timeout occurs are simply returned in the second set.

return_when indicates when this function should return. It must be one of the following constants:

Constant	**Description**
asyncio.FIRST_COMPLETED	The function will return when any future finishes or is cancelled.
asyncio.FIRST_EXCEPTION	The function will return when any future finishes by raising an exception. If no future raises an exception then it is equivalent to `ALL_COMPLETED`.
asyncio.ALL_COMPLETED	The function will return when all futures finish or are cancelled.

Unlike `wait_for()`, `wait()` does not cancel the futures when a timeout occurs.

Changed in version 3.10: Removed the *loop* parameter.

Changed in version 3.11: Passing coroutine objects to `wait()` directly is forbidden.

Changed in version 3.12: Added support for generators yielding tasks.

asyncio.as_completed(*aws*, *, *timeout=None*)

Run awaitable objects in the *aws* iterable concurrently. The returned object can be iterated to obtain the results of the awaitables as they finish.

The object returned by `as_completed()` can be iterated as an asynchronous iterator or a plain iterator. When asynchronous iteration is used, the originally-supplied awaitables are yielded if they are tasks or futures. This makes it easy to correlate previously-scheduled tasks with their results. Example:

```
ipv4_connect = create_task(open_connection("127.0.0.1", 80))
ipv6_connect = create_task(open_connection("::1", 80))
tasks = [ipv4_connect, ipv6_connect]

async for earliest_connect in as_completed(tasks):
    # earliest_connect is done. The result can be obtained
by
    # awaiting it or calling earliest_connect.result()
    reader, writer = await earliest_connect

    if earliest_connect is ipv6_connect:
        print("IPv6 connection established.")
    else:
        print("IPv4 connection established.")
```

During asynchronous iteration, implicitly-created tasks will be yielded for supplied awaitables that aren’t tasks or futures.

When used as a plain iterator, each iteration yields a new coroutine that returns the result or raises the exception of the next completed awaitable. This pattern is compatible with Python versions older than 3.13:

```
ipv4_connect = create_task(open_connection("127.0.0.1", 80))
ipv6_connect = create_task(open_connection("::1", 80))
tasks = [ipv4_connect, ipv6_connect]

for next_connect in as_completed(tasks):
    # next_connect is not one of the original task objects.
It must be
    # awaited to obtain the result value or raise the
exception of the
    # awaitable that finishes next.
    reader, writer = await next_connect
```

A **`TimeoutError`** is raised if the timeout occurs before all awaitables are done. This is raised by the `async for` loop during asynchronous iteration or by the coroutines yielded during plain iteration.

Changed in version 3.10: Removed the *loop* parameter.

Deprecated since version 3.10: Deprecation warning is emitted if not all awaitable objects in the *aws* iterable are Future-like objects and there is no running event loop.

Changed in version 3.12: Added support for generators yielding tasks.

Changed in version 3.13: The result can now be used as either an asynchronous iterator or as a plain iterator (previously it was only a plain iterator).

Running in Threads

async asyncio.to_thread(*func*, */*, **args*, ***kwargs*)

Asynchronously run function *func* in a separate thread.

Any *args and **kwargs supplied for this function are directly passed to *func*. Also, the current `contextvars.Context` is propagated, allowing context variables from the event loop thread to be accessed in the separate thread.

Return a coroutine that can be awaited to get the eventual result of *func*.

This coroutine function is primarily intended to be used for executing IO-bound functions/methods that would otherwise block the event loop if they were run in the main thread. For example:

```
def blocking_io():
    print(f"start blocking_io at {time.strftime('%X')}")
    # Note that time.sleep() can be replaced with any
blocking
    # IO-bound operation, such as file operations.
    time.sleep(1)
    print(f"blocking_io complete at {time.strftime('%X')}")

async def main():
    print(f"started main at {time.strftime('%X')}")

    await asyncio.gather(
        asyncio.to_thread(blocking_io),
        asyncio.sleep(1))

    print(f"finished main at {time.strftime('%X')}")


asyncio.run(main())

# Expected output:
#
# started main at 19:50:53
# start blocking_io at 19:50:53
# blocking_io complete at 19:50:54
# finished main at 19:50:54
```

Directly calling `blocking_io()` in any coroutine would block the event loop for its duration, resulting in an additional 1 second of run time. Instead, by using `asyncio.to_thread()`, we can run it in a separate thread without blocking the event loop.

Note

Due to the GIL, `asyncio.to_thread()` can typically only be used to make IO-bound functions non-blocking. However, for extension modules that release the GIL or alternative Python implementations that don't have one, `asyncio.to_thread()` can also be used for CPU-bound functions.

Added in version 3.9.

Scheduling From Other Threads

asyncio.run_coroutine_threadsafe(*coro*, *loop*)

Submit a coroutine to the given event loop. Thread-safe.

Return a **`concurrent.futures.Future`** to wait for the result from another OS thread.

This function is meant to be called from a different OS thread than the one where the event loop is running. Example:

```
def in_thread(loop: asyncio.AbstractEventLoop) -> None:
    # Run some blocking IO
    pathlib.Path("example.txt").write_text("hello world",
encoding="utf8")

    # Create a coroutine
    coro = asyncio.sleep(1, result=3)
    # Submit the coroutine to a given loop
    future = asyncio.run_coroutine_threadsafe(coro, loop)

    # Wait for the result with an optional timeout argument
    assert future.result(timeout=2) == 3

async def amain() -> None:
    # Get the running loop
    loop = asyncio.get_running_loop()

    # Run something in a thread
    await asyncio.to_thread(in_thread, loop)
```

It’s also possible to run the other way around. Example:

```
@contextlib.contextmanager
def loop_in_thread() ->
Generator[asyncio.AbstractEventLoop]:
    loop_fut =
concurrent.futures.Future[asyncio.AbstractEventLoop]()
    stop_event = asyncio.Event()

    async def main() -> None:
        loop_fut.set_result(asyncio.get_running_loop())
        await stop_event.wait()

    with concurrent.futures.ThreadPoolExecutor(1) as tpe:
        complete_fut = tpe.submit(asyncio.run, main())
        for fut in
concurrent.futures.as_completed((loop_fut, complete_fut)):
            if fut is loop_fut:
                loop = loop_fut.result()
                try:
                    yield loop
                finally:

loop.call_soon_threadsafe(stop_event.set)
            else:
                fut.result()

# Create a loop in another thread
with loop_in_thread() as loop:
    # Create a coroutine
    coro = asyncio.sleep(1, result=3)
    # Submit the coroutine to a given loop
    future = asyncio.run_coroutine_threadsafe(coro, loop)

    # Wait for the result with an optional timeout argument
    assert future.result(timeout=2) == 3
```

If an exception is raised in the coroutine, the returned Future will be notified. It can also be used to cancel the task in the event loop:

```
try:
    result = future.result(timeout)
except TimeoutError:
    print('The coroutine took too long, cancelling the
task...')
    future.cancel()
except Exception as exc:
    print(f'The coroutine raised an exception: {exc!r}')
else:
    print(f'The coroutine returned: {result!r}')
```

See the concurrency and multithreading section of the documentation.

Unlike other asyncio functions this function requires the *loop* argument to be passed explicitly.

Added in version 3.5.1.

Introspection

asyncio.current_task(*loop=None*)

Return the currently running **`Task`** instance, or `None` if no task is running.

If *loop* is `None` **`get_running_loop()`** is used to get the current loop.

Added in version 3.7.

asyncio.all_tasks(*loop=None*)

Return a set of not yet finished `Task` objects run by the loop.

If *loop* is `None`, `get_running_loop()` is used for getting current loop.

Added in version 3.7.

asyncio.iscoroutine(*obj*)

Return `True` if *obj* is a coroutine object.

Added in version 3.4.

Task Object

class asyncio.Task(*coro*, ***, *loop=None*, *name=None*, *context=None*, *eager_start=False*)

A `Future-like` object that runs a Python coroutine. Not thread-safe.

Tasks are used to run coroutines in event loops. If a coroutine awaits on a Future, the Task suspends the execution of the coroutine and waits for the completion of the Future. When the Future is *done*, the execution of the wrapped coroutine resumes.

Event loops use cooperative scheduling: an event loop runs one Task at a time. While a Task awaits for the completion of a Future, the event loop runs other Tasks, callbacks, or performs IO operations.

Use the high-level `asyncio.create_task()` function to create Tasks, or the low-level `loop.create_task()` or `ensure_future()` functions. Manual instantiation of Tasks is discouraged.

To cancel a running Task use the `cancel()` method. Calling it will cause the Task to throw a `CancelledError` exception into the wrapped coroutine. If a coroutine is awaiting on a Future object during cancellation, the Future object will be cancelled.

`cancelled()` can be used to check if the Task was cancelled. The method returns `True` if the wrapped coroutine did not suppress the `CancelledError` exception and was actually cancelled.

`asyncio.Task` inherits from `Future` all of its APIs except `Future.set_result()` and `Future.set_exception()`.

An optional keyword-only *context* argument allows specifying a custom `contextvars.Context` for the *coro* to run in. If no *context* is provided, the Task copies the current context and later runs its coroutine in the copied context.

An optional keyword-only *eager_start* argument allows eagerly starting the execution of the `asyncio.Task` at task creation time. If set to `True` and the event loop is running, the task will start executing the coroutine immediately, until the first time the coroutine blocks. If the coroutine returns or raises without blocking, the task will be finished eagerly and will skip scheduling to the event loop.

Changed in version 3.7: Added support for the `contextvars` module.

Changed in version 3.8: Added the *name* parameter.

Deprecated since version 3.10: Deprecation warning is emitted if *loop* is not specified and there is no running event loop.

Changed in version 3.11: Added the *context* parameter.

Changed in version 3.12: Added the *eager_start* parameter.

done()

Return `True` if the Task is *done*.

A Task is *done* when the wrapped coroutine either returned a value, raised an exception, or the Task was cancelled.

result()

Return the result of the Task.

If the Task is *done*, the result of the wrapped coroutine is returned (or if the coroutine raised an exception, that exception is re-raised.)

If the Task has been *cancelled*, this method raises a **`CancelledError`** exception.

If the Task's result isn't yet available, this method raises an **`InvalidStateError`** exception.

exception()

Return the exception of the Task.

If the wrapped coroutine raised an exception that exception is returned. If the wrapped coroutine returned normally this method returns `None`.

If the Task has been *cancelled*, this method raises a **`CancelledError`** exception.

If the Task isn't *done* yet, this method raises an **`InvalidStateError`** exception.

add_done_callback(*callback*, *, *context=None*)

Add a callback to be run when the Task is *done*.

This method should only be used in low-level callback-based code.

See the documentation of `Future.add_done_callback()` for more details.

remove_done_callback(*callback*)

Remove *callback* from the callbacks list.

This method should only be used in low-level callback-based code.

See the documentation of `Future.remove_done_callback()` for more details.

get_stack(*, *limit=None*)

Return the list of stack frames for this Task.

If the wrapped coroutine is not done, this returns the stack where it is suspended. If the coroutine has completed successfully or was cancelled, this returns an empty list. If the coroutine was terminated by an exception, this returns the list of traceback frames.

The frames are always ordered from oldest to newest.

Only one stack frame is returned for a suspended coroutine.

The optional *limit* argument sets the maximum number of frames to return; by default all available frames are returned. The ordering of the returned list differs depending on whether a stack or a traceback is returned: the newest frames of a stack are returned, but the oldest frames of a traceback are returned. (This matches the behavior of the traceback module.)

print_stack(*, *limit=None*, *file=None*)

Print the stack or traceback for this Task.

This produces output similar to that of the traceback module for the frames retrieved by `get_stack()`.

The *limit* argument is passed to `get_stack()` directly.

The *file* argument is an I/O stream to which the output is written; by default output is written to `sys.stdout`.

get_coro()

Return the coroutine object wrapped by the `Task`.

Note

This will return `None` for Tasks which have already completed eagerly. See the Eager Task Factory.

Added in version 3.8.

Changed in version 3.12: Newly added eager task execution means result may be `None`.

get_context()

Return the `contextvars.Context` object associated with the task.

Added in version 3.12.

get_name()

Return the name of the Task.

If no name has been explicitly assigned to the Task, the default asyncio Task implementation generates a default name during instantiation.

Added in version 3.8.

set_name(*value*)

Set the name of the Task.

The *value* argument can be any object, which is then converted to a string.

In the default Task implementation, the name will be visible in the `repr()` output of a task object.

Added in version 3.8.

cancel(*msg=None*)

Request the Task to be cancelled.

If the Task is already *done* or *cancelled*, return `False`, otherwise, return `True`.

The method arranges for a `CancelledError` exception to be thrown into the wrapped coroutine on the next cycle of the event loop.

The coroutine then has a chance to clean up or even deny the request by suppressing the exception with a `try` … … `except CancelledError` … `finally` block. Therefore, unlike `Future.cancel()`, `Task.cancel()` does not guarantee that the Task will be cancelled, although suppressing cancellation completely is not common and is actively discouraged. Should the coroutine nevertheless decide to suppress the cancellation, it needs to call `Task.uncancel()` in addition to catching the exception.

Changed in version 3.9: Added the *msg* parameter.

Changed in version 3.11: The `msg` parameter is propagated from cancelled task to its awaiter.

The following example illustrates how coroutines can intercept the cancellation request:

```
async def cancel_me():
    print('cancel_me(): before sleep')

    try:
        # Wait for 1 hour
        await asyncio.sleep(3600)
    except asyncio.CancelledError:
        print('cancel_me(): cancel sleep')
        raise
    finally:
        print('cancel_me(): after sleep')

async def main():
    # Create a "cancel_me" Task
    task = asyncio.create_task(cancel_me())

    # Wait for 1 second
    await asyncio.sleep(1)

    task.cancel()
    try:
        await task
    except asyncio.CancelledError:
        print("main(): cancel_me is cancelled now")

asyncio.run(main())

# Expected output:
#
#     cancel_me(): before sleep
#     cancel_me(): cancel sleep
#     cancel_me(): after sleep
#     main(): cancel_me is cancelled now
```

cancelled()

Return `True` if the Task is *cancelled*.

The Task is *cancelled* when the cancellation was requested with **`cancel()`** and the wrapped coroutine propagated the **`CancelledError`** exception thrown into it.

uncancel()

Decrement the count of cancellation requests to this Task.

Returns the remaining number of cancellation requests.

Note that once execution of a cancelled task completed, further calls to **`uncancel()`** are ineffective.

Added in version 3.11.

This method is used by asyncio’s internals and isn’t expected to be used by end-user code. In particular, if a Task gets successfully uncancelled, this allows for elements of structured concurrency like Task Groups and **`asyncio.timeout()`** to continue running, isolating cancellation to the respective structured block. For example:

```
async def make_request_with_timeout():
    try:
        async with asyncio.timeout(1):
            # Structured block affected by the timeout:
            await make_request()
            await make_another_request()
    except TimeoutError:
        log("There was a timeout")
    # Outer code not affected by the timeout:
    await unrelated_code()
```

While the block with `make_request()` and `make_another_request()` might get cancelled due to the timeout, `unrelated_code()` should continue running even in case of the timeout. This is implemented with `uncancel()`. `TaskGroup` context managers use `uncancel()` in a similar fashion.

If end-user code is, for some reason, suppressing cancellation by catching `CancelledError`, it needs to call this method to remove the cancellation state.

When this method decrements the cancellation count to zero, the method checks if a previous `cancel()` call had arranged for `CancelledError` to be thrown into the task. If it hasn't been thrown yet, that arrangement will be rescinded (by resetting the internal `_must_cancel` flag).

Changed in version 3.13: Changed to rescind pending cancellation requests upon reaching zero.

cancelling()

Return the number of pending cancellation requests to this Task, i.e., the number of calls to `cancel()` less the number of `uncancel()` calls.

Note that if this number is greater than zero but the Task is still executing, `cancelled()` will still return `False`. This is because this number can be lowered by calling `uncancel()`, which can lead to the task not being cancelled after all if the cancellation requests go down to zero.

This method is used by asyncio's internals and isn't expected to be used by end-user code. See `uncancel()` for more details.

Added in version 3.11.

Streams

Source code: Lib/asyncio/streams.py [https://github.com/python/cpython/tree/3.14/Lib/asyncio/streams.py]

Streams are high-level async/await-ready primitives to work with network connections. Streams allow sending and receiving data without using callbacks or low-level protocols and transports.

Here is an example of a TCP echo client written using asyncio streams:

```
import asyncio

async def tcp_echo_client(message):
    reader, writer = await asyncio.open_connection(
        '127.0.0.1', 8888)

    print(f'Send: {message!r}')
    writer.write(message.encode())
    await writer.drain()

    data = await reader.read(100)
    print(f'Received: {data.decode()!r}')

    print('Close the connection')
    writer.close()
    await writer.wait_closed()

asyncio.run(tcp_echo_client('Hello World!'))
```

See also the Examples section below.

Stream Functions

The following top-level asyncio functions can be used to create and work with streams:

async asyncio.open_connection(*host=None*, *port=None*, ***, *limit=None*, *ssl=None*, *family=0*, *proto=0*, *flags=0*, *sock=None*, *local_addr=None*, *server_hostname=None*, *ssl_handshake_timeout=None*, *ssl_shutdown_timeout=None*, *happy_eyeballs_delay=None*, *interleave=None*)

Establish a network connection and return a pair of `(reader, writer)` objects.

The returned *reader* and *writer* objects are instances of `StreamReader` and `StreamWriter` classes.

limit determines the buffer size limit used by the returned `StreamReader` instance. By default the *limit* is set to 64 KiB.

The rest of the arguments are passed directly to `loop.create_connection()`.

Note

The *sock* argument transfers ownership of the socket to the `StreamWriter` created. To close the socket, call its `close()` method.

Changed in version 3.7: Added the *ssl_handshake_timeout* parameter.

Changed in version 3.8: Added the *happy_eyeballs_delay* and *interleave* parameters.

Changed in version 3.10: Removed the *loop* parameter.

Changed in version 3.11: Added the *ssl_shutdown_timeout* parameter.

async asyncio.start_server(*client_connected_cb*, *host=None*, *port=None*, ***, *limit=None*, *family=socket.AF_UNSPEC*, *flags=socket.AI_PASSIVE*, *sock=None*, *backlog=100*, *ssl=None*, *reuse_address=None*, *reuse_port=None*, *keep_alive=None*, *ssl_handshake_timeout=None*, *ssl_shutdown_timeout=None*, *start_serving=True*)

Start a socket server.

The *client_connected_cb* callback is called whenever a new client connection is established. It receives a `(reader, writer)` pair as two arguments, instances of the **`StreamReader`** and **`StreamWriter`** classes.

client_connected_cb can be a plain callable or a coroutine function; if it is a coroutine function, it will be automatically scheduled as a **`Task`**.

limit determines the buffer size limit used by the returned **`StreamReader`** instance. By default the *limit* is set to 64 KiB.

The rest of the arguments are passed directly to **`loop.create_server()`**.

Note

The *sock* argument transfers ownership of the socket to the server created. To close the socket, call the server's **`close()`** method.

Changed in version 3.7: Added the *ssl_handshake_timeout* and *start_serving* parameters.

Changed in version 3.10: Removed the *loop* parameter.

Changed in version 3.11: Added the *ssl_shutdown_timeout* parameter.

Changed in version 3.13: Added the *keep_alive* parameter.

Unix Sockets

async asyncio.open_unix_connection(*path=None*, *, *limit=None*, *ssl=None*, *sock=None*, *server_hostname=None*, *ssl_handshake_timeout=None*, *ssl_shutdown_timeout=None*)

Establish a Unix socket connection and return a pair of `(reader, writer)`.

Similar to **`open_connection()`** but operates on Unix sockets.

See also the documentation of **`loop.create_unix_connection()`**.

Note

The *sock* argument transfers ownership of the socket to the **`StreamWriter`** created. To close the socket, call its **`close()`** method.

Availability: Unix.

Changed in version 3.7: Added the *ssl_handshake_timeout* parameter. The *path* parameter can now be a path-like object

Changed in version 3.10: Removed the *loop* parameter.

Changed in version 3.11: Added the *ssl_shutdown_timeout* parameter.

async asyncio.start_unix_server(*client_connected_cb*, *path=None*, ***, *limit=None*, *sock=None*, *backlog=100*, *ssl=None*, *ssl_handshake_timeout=None*, *ssl_shutdown_timeout=None*, *start_serving=True*, *cleanup_socket=True*)

Start a Unix socket server.

Similar to `start_server()` but works with Unix sockets.

If *cleanup_socket* is true then the Unix socket will automatically be removed from the filesystem when the server is closed, unless the socket has been replaced after the server has been created.

See also the documentation of `loop.create_unix_server()`.

Note

The *sock* argument transfers ownership of the socket to the server created. To close the socket, call the server's `close()` method.

Availability: Unix.

Changed in version 3.7: Added the *ssl_handshake_timeout* and *start_serving* parameters. The *path* parameter can now be a path-like object.

Changed in version 3.10: Removed the *loop* parameter.

Changed in version 3.11: Added the *ssl_shutdown_timeout* parameter.

Changed in version 3.13: Added the *cleanup_socket* parameter.

StreamReader

class asyncio.StreamReader

Represents a reader object that provides APIs to read data from the IO stream. As an asynchronous iterable, the object supports the **`async for`** statement.

It is not recommended to instantiate *StreamReader* objects directly; use **`open_connection()`** and **`start_server()`** instead.

feed_eof()

Acknowledge the EOF.

async read(*n=-1*)

Read up to *n* bytes from the stream.

If *n* is not provided or set to `-1`, read until EOF, then return all read **`bytes`**. If EOF was received and the internal buffer is empty, return an empty `bytes` object.

If *n* is `0`, return an empty `bytes` object immediately.

If *n* is positive, return at most *n* available `bytes` as soon as at least 1 byte is available in the internal buffer. If EOF is received before any byte is read, return an empty `bytes` object.

async readline()

Read one line, where “line” is a sequence of bytes ending with `\n`.

If EOF is received and `\n` was not found, the method returns partially read data.

If EOF is received and the internal buffer is empty, return an empty `bytes` object.

async readexactly(*n*)

Read exactly *n* bytes.

Raise an **`IncompleteReadError`** if EOF is reached before *n* can be read. Use the **`IncompleteReadError.partial`** attribute to get the partially read data.

async readuntil(*separator=b'\n'*)

Read data from the stream until *separator* is found.

On success, the data and separator will be removed from the internal buffer (consumed). Returned data will include the separator at the end.

If the amount of data read exceeds the configured stream limit, a **`LimitOverrunError`** exception is raised, and the data is left in the internal buffer and can be read again.

If EOF is reached before the complete separator is found, an **`IncompleteReadError`** exception is raised, and the internal buffer is reset. The **`IncompleteReadError.partial`** attribute may contain a portion of the separator.

The *separator* may also be a tuple of separators. In this case the return value will be the shortest possible that has any separator as the suffix. For the purposes of **`LimitOverrunError`**, the shortest possible separator is considered to be the one that matched.

Added in version 3.5.2.

Changed in version 3.13: The *separator* parameter may now be a `tuple` of separators.

at_eof()

Return `True` if the buffer is empty and `feed_eof()` was called.

StreamWriter

class asyncio.StreamWriter

Represents a writer object that provides APIs to write data to the IO stream.

It is not recommended to instantiate *StreamWriter* objects directly; use `open_connection()` and `start_server()` instead.

write(*data*)

The method attempts to write the *data* to the underlying socket immediately. If that fails, the data is queued in an internal write buffer until it can be sent.

The *data* buffer should be a bytes, bytearray, or C-contiguous one-dimensional memoryview object.

The method should be used along with the `drain()` method:

```
stream.write(data)
await stream.drain()
```

writelines(*data*)

The method writes a list (or any iterable) of bytes to the underlying socket immediately. If that fails, the data is queued in an internal write buffer until it can be sent.

The method should be used along with the `drain()` method:

```
stream.writelines(lines)
await stream.drain()
```

close()

The method closes the stream and the underlying socket.

The method should be used, though not mandatory, along with the `wait_closed()` method:

```
stream.close()
await stream.wait_closed()
```

can_write_eof()

Return `True` if the underlying transport supports the **`write_eof()`** method, `False` otherwise.

write_eof()

Close the write end of the stream after the buffered write data is flushed.

transport

Return the underlying asyncio transport.

get_extra_info(*name*, *default=None*)

Access optional transport information; see **`BaseTransport.get_extra_info()`** for details.

async drain()

Wait until it is appropriate to resume writing to the stream. Example:

```
writer.write(data)
await writer.drain()
```

This is a flow control method that interacts with the underlying IO write buffer. When the size of the buffer reaches the high watermark, *drain()* blocks until the size of the buffer is drained down to the low watermark and writing can be resumed. When there is nothing to wait for, the `drain()` returns immediately.

async start_tls(*sslcontext*, *, *server_hostname=None*, *ssl_handshake_timeout=None*, *ssl_shutdown_timeout=None*)

Upgrade an existing stream-based connection to TLS.

Parameters:

- *sslcontext*: a configured instance of `SSLContext`.
- *server_hostname*: sets or overrides the host name that the target server's certificate will be matched against.
- *ssl_handshake_timeout* is the time in seconds to wait for the TLS handshake to complete before aborting the connection. `60.0` seconds if `None` (default).
- *ssl_shutdown_timeout* is the time in seconds to wait for the SSL shutdown to complete before aborting the connection. `30.0` seconds if `None` (default).

Added in version 3.11.

Changed in version 3.12: Added the *ssl_shutdown_timeout* parameter.

is_closing()

Return `True` if the stream is closed or in the process of being closed.

Added in version 3.7.

async wait_closed()

Wait until the stream is closed.

Should be called after `close()` to wait until the underlying connection is closed, ensuring that all data has been flushed before e.g. exiting the program.

Added in version 3.7.

Examples

TCP echo client using streams

TCP echo client using the `asyncio.open_connection()` function:

```
import asyncio

async def tcp_echo_client(message):
    reader, writer = await asyncio.open_connection(
        '127.0.0.1', 8888)

    print(f'Send: {message!r}')
    writer.write(message.encode())
    await writer.drain()

    data = await reader.read(100)
    print(f'Received: {data.decode()!r}')

    print('Close the connection')
    writer.close()
    await writer.wait_closed()

asyncio.run(tcp_echo_client('Hello World!'))
```

See also

The TCP echo client protocol example uses the low-level `loop.create_connection()` method.

TCP echo server using streams

TCP echo server using the `asyncio.start_server()` function:

```
import asyncio

async def handle_echo(reader, writer):
    data = await reader.read(100)
    message = data.decode()
    addr = writer.get_extra_info('peername')

    print(f"Received {message!r} from {addr!r}")

    print(f"Send: {message!r}")
    writer.write(data)
    await writer.drain()

    print("Close the connection")
    writer.close()
    await writer.wait_closed()

async def main():
    server = await asyncio.start_server(
        handle_echo, '127.0.0.1', 8888)

    addrs = ', '.join(str(sock.getsockname()) for sock in
server.sockets)
    print(f'Serving on {addrs}')

    async with server:
        await server.serve_forever()

asyncio.run(main())
```

See also

The TCP echo server protocol example uses the `loop.create_server()` method.

Get HTTP headers

Simple example querying HTTP headers of the URL passed on the command line:

```
import asyncio
import urllib.parse
import sys

async def print_http_headers(url):
    url = urllib.parse.urlsplit(url)
    if url.scheme == 'https':
        reader, writer = await asyncio.open_connection(
            url.hostname, 443, ssl=True)
    else:
        reader, writer = await asyncio.open_connection(
            url.hostname, 80)

    query = (
        f"HEAD {url.path or '/'} HTTP/1.0\r\n"
        f"Host: {url.hostname}\r\n"
        f"\r\n"
    )

    writer.write(query.encode('latin-1'))
    while True:
        line = await reader.readline()
        if not line:
            break

        line = line.decode('latin1').rstrip()
        if line:
            print(f'HTTP header> {line}')
    # Ignore the body, close the socket
    writer.close()
    await writer.wait_closed()

url = sys.argv[1]
asyncio.run(print_http_headers(url))
```

Usage:

```
python example.py http://example.com/path/page.html
```

or with HTTPS:

```
python example.py https://example.com/path/page.html
```

Register an open socket to wait for data using streams

Coroutine waiting until a socket receives data using the `open_connection()` function:

```
import asyncio
import socket

async def wait_for_data():
    # Get a reference to the current event loop because
    # we want to access low-level APIs.
    loop = asyncio.get_running_loop()

    # Create a pair of connected sockets.
    rsock, wsock = socket.socketpair()

    # Register the open socket to wait for data.
    reader, writer = await asyncio.open_connection(sock=rsock)

    # Simulate the reception of data from the network
    loop.call_soon(wsock.send, 'abc'.encode())

    # Wait for data
    data = await reader.read(100)
    # Got data, we are done: close the socket
    print("Received:", data.decode())
    writer.close()
    await writer.wait_closed()

    # Close the second socket
    wsock.close()

asyncio.run(wait_for_data())
```

See also

The register an open socket to wait for data using a protocol example uses a low-level protocol and the **`loop.create_connection()`** method.

The watch a file descriptor for read events example uses the low-level **`loop.add_reader()`** method to watch a file descriptor.

Synchronization Primitives

Source code: Lib/asyncio/locks.py
[https://github.com/python/cpython/tree/3.14/Lib/asyncio/locks.py]

---

asyncio synchronization primitives are designed to be similar to those of the `threading` module with two important caveats:

- asyncio primitives are not thread-safe, therefore they should not be used for OS thread synchronization (use `threading` for that);
- methods of these synchronization primitives do not accept the *timeout* argument; use the `asyncio.wait_for()` function to perform operations with timeouts.

asyncio has the following basic synchronization primitives:

- `Lock`
- `Event`
- `Condition`
- `Semaphore`
- `BoundedSemaphore`
- `Barrier`

Lock

class asyncio.Lock

Implements a mutex lock for asyncio tasks. Not thread-safe.

An asyncio lock can be used to guarantee exclusive access to a shared resource.

The preferred way to use a Lock is an `async with` statement:

```
lock = asyncio.Lock()

# ... later
async with lock:
    # access shared state
```

which is equivalent to:

```
lock = asyncio.Lock()

# ... later
await lock.acquire()
try:
    # access shared state
finally:
    lock.release()
```

Changed in version 3.10: Removed the *loop* parameter.

async acquire()

Acquire the lock.

This method waits until the lock is *unlocked*, sets it to *locked* and returns `True`.

When more than one coroutine is blocked in `acquire()` waiting for the lock to be unlocked, only one coroutine eventually proceeds.

Acquiring a lock is *fair*: the coroutine that proceeds will be the first coroutine that started waiting on the lock.

release()

Release the lock.

When the lock is *locked*, reset it to *unlocked* and return.

If the lock is *unlocked*, a `RuntimeError` is raised.

locked()

Return `True` if the lock is *locked*.

Event

class asyncio.Event

An event object. Not thread-safe.

An asyncio event can be used to notify multiple asyncio tasks that some event has happened.

An Event object manages an internal flag that can be set to *true* with the `set()` method and reset to *false* with the `clear()` method. The `wait()` method blocks until the flag is set to *true*. The flag is set to *false* initially.

Changed in version 3.10: Removed the *loop* parameter.

Example:

```
async def waiter(event):
    print('waiting for it ...')
    await event.wait()
    print('... got it!')

async def main():
    # Create an Event object.
    event = asyncio.Event()
    # Spawn a Task to wait until 'event' is set.
    waiter_task = asyncio.create_task(waiter(event))

    # Sleep for 1 second and set the event.
    await asyncio.sleep(1)
    event.set()

    # Wait until the waiter task is finished.
    await waiter_task

asyncio.run(main())
```

async wait()

Wait until the event is set.

If the event is set, return `True` immediately. Otherwise block until another task calls `set()`.

set()

Set the event.

All tasks waiting for event to be set will be immediately awakened.

clear()

Clear (unset) the event.

Subsequent tasks awaiting on `wait()` will now block until the `set()` method is called again.

is_set()

Return `True` if the event is set.

Condition

class asyncio.Condition(*lock=None*)

A Condition object. Not thread-safe.

An asyncio condition primitive can be used by a task to wait for some event to happen and then get exclusive access to a shared resource.

In essence, a Condition object combines the functionality of an `Event` and a `Lock`. It is possible to have multiple Condition objects share one Lock, which allows coordinating exclusive access to a shared resource between different tasks interested in particular states of that shared resource.

The optional *lock* argument must be a `Lock` object or `None`. In the latter case a new Lock object is created automatically.

Changed in version 3.10: Removed the *loop* parameter.

The preferred way to use a Condition is an `async with` statement:

```
cond = asyncio.Condition()

# ... later
async with cond:
    await cond.wait()
```

which is equivalent to:

```
cond = asyncio.Condition()

# ... later
await cond.acquire()
try:
    await cond.wait()
finally:
    cond.release()
```

async acquire()

Acquire the underlying lock.

This method waits until the underlying lock is *unlocked*, sets it to *locked* and returns `True`.

notify(*n=1*)

Wake up *n* tasks (1 by default) waiting on this condition. If fewer than *n* tasks are waiting they are all awakened.

The lock must be acquired before this method is called and released shortly after. If called with an *unlocked* lock a **`RuntimeError`** error is raised.

locked()

Return `True` if the underlying lock is acquired.

notify_all()

Wake up all tasks waiting on this condition.

This method acts like **`notify()`**, but wakes up all waiting tasks.

The lock must be acquired before this method is called and released shortly after. If called with an *unlocked* lock a **`RuntimeError`** error is raised.

release()

Release the underlying lock.

When invoked on an unlocked lock, a **`RuntimeError`** is raised.

async wait()

Wait until notified.

If the calling task has not acquired the lock when this method is called, a **`RuntimeError`** is raised.

This method releases the underlying lock, and then blocks until it is awakened by a `notify()` or `notify_all()` call. Once awakened, the Condition re-acquires its lock and this method returns `True`.

Note that a task *may* return from this call spuriously, which is why the caller should always re-check the state and be prepared to `wait()` again. For this reason, you may prefer to use `wait_for()` instead.

async wait_for(*predicate*)

Wait until a predicate becomes *true*.

The predicate must be a callable which result will be interpreted as a boolean value. The method will repeatedly `wait()` until the predicate evaluates to *true*. The final value is the return value.

Semaphore

class asyncio.Semaphore(*value=1*)

A Semaphore object. Not thread-safe.

A semaphore manages an internal counter which is decremented by each `acquire()` call and incremented by each `release()` call. The counter can never go below zero; when `acquire()` finds that it is zero, it blocks, waiting until some task calls `release()`.

The optional *value* argument gives the initial value for the internal counter (`1` by default). If the given value is less than `0` a `ValueError` is raised.

Changed in version 3.10: Removed the *loop* parameter.

The preferred way to use a Semaphore is an **`async with`** statement:

```
sem = asyncio.Semaphore(10)

# ... later
async with sem:
    # work with shared resource
```

which is equivalent to:

```
sem = asyncio.Semaphore(10)

# ... later
await sem.acquire()
try:
    # work with shared resource
finally:
    sem.release()
```

async acquire()

Acquire a semaphore.

If the internal counter is greater than zero, decrement it by one and return `True` immediately. If it is zero, wait until a **`release()`** is called and return `True`.

locked()

Returns `True` if semaphore can not be acquired immediately.

release()

Release a semaphore, incrementing the internal counter by one. Can wake up a task waiting to acquire the semaphore.

Unlike **`BoundedSemaphore`**, **`Semaphore`** allows making more `release()` calls than `acquire()` calls.

BoundedSemaphore

class asyncio.BoundedSemaphore(*value=1*)

A bounded semaphore object. Not thread-safe.

Bounded Semaphore is a version of `Semaphore` that raises a `ValueError` in `release()` if it increases the internal counter above the initial *value*.

Changed in version 3.10: Removed the *loop* parameter.

Barrier

class asyncio.Barrier(*parties*)

A barrier object. Not thread-safe.

A barrier is a simple synchronization primitive that allows to block until *parties* number of tasks are waiting on it. Tasks can wait on the `wait()` method and would be blocked until the specified number of tasks end up waiting on `wait()`. At that point all of the waiting tasks would unblock simultaneously.

`async with` can be used as an alternative to awaiting on `wait()`.

The barrier can be reused any number of times.

Example:

```
async def example_barrier():
   # barrier with 3 parties
   b = asyncio.Barrier(3)

   # create 2 new waiting tasks
   asyncio.create_task(b.wait())
   asyncio.create_task(b.wait())
```

```
    await asyncio.sleep(0)
    print(b)

    # The third .wait() call passes the barrier
    await b.wait()
    print(b)
    print("barrier passed")

    await asyncio.sleep(0)
    print(b)

asyncio.run(example_barrier())
```

Result of this example is:

```
<asyncio.locks.Barrier object at 0x... [filling,
waiters:2/3]>
<asyncio.locks.Barrier object at 0x... [draining,
waiters:0/3]>
barrier passed
<asyncio.locks.Barrier object at 0x... [filling,
waiters:0/3]>
```

Added in version 3.11.

async wait()

Pass the barrier. When all the tasks party to the barrier have called this function, they are all unblocked simultaneously.

When a waiting or blocked task in the barrier is cancelled, this task exits the barrier which stays in the same state. If the state of the barrier is “filling”, the number of waiting task decreases by 1.

The return value is an integer in the range of 0 to `parties-1`, different for each task. This can be used to select a task to do some special housekeeping, e.g.:

```
...
async with barrier as position:
   if position == 0:
      # Only one task prints this
      print('End of *draining phase*')
```

This method may raise a `BrokenBarrierError` exception if the barrier is broken or reset while a task is waiting. It could raise a `CancelledError` if a task is cancelled.

async reset()

Return the barrier to the default, empty state. Any tasks waiting on it will receive the `BrokenBarrierError` exception.

If a barrier is broken it may be better to just leave it and create a new one.

async abort()

Put the barrier into a broken state. This causes any active or future calls to `wait()` to fail with the `BrokenBarrierError`. Use this for example if one of the tasks needs to abort, to avoid infinite waiting tasks.

parties

The number of tasks required to pass the barrier.

n_waiting

The number of tasks currently waiting in the barrier while filling.

broken

A boolean that is `True` if the barrier is in the broken state.

exception asyncio.BrokenBarrierError

This exception, a subclass of **`RuntimeError`**, is raised when the **`Barrier`** object is reset or broken.

Changed in version 3.9: Acquiring a lock using `await lock` or `yield from lock` and/or **`with`** statement (`with await lock`, `with (yield from lock)`) was removed. Use `async with lock` instead.

Subprocesses

Source code: Lib/asyncio/subprocess.py [https://github.com/python/cpython/tree/3.14/Lib/asyncio/subprocess.py], Lib/asyncio/base_subprocess.py [https://github.com/python/cpython/tree/3.14/Lib/asyncio/base_subprocess.py]

This section describes high-level async/await asyncio APIs to create and manage subprocesses.

Here’s an example of how asyncio can run a shell command and obtain its result:

```
import asyncio

async def run(cmd):
    proc = await asyncio.create_subprocess_shell(
        cmd,
        stdout=asyncio.subprocess.PIPE,
        stderr=asyncio.subprocess.PIPE)

    stdout, stderr = await proc.communicate()

    print(f'[{cmd!r} exited with {proc.returncode}]')
    if stdout:
        print(f'[stdout]\n{stdout.decode()}')
    if stderr:
        print(f'[stderr]\n{stderr.decode()}')

asyncio.run(run('ls /zzz'))
```

will print:

```
['ls /zzz' exited with 1]
[stderr]
ls: /zzz: No such file or directory
```

Because all asyncio subprocess functions are asynchronous and asyncio provides many tools to work with such functions, it is easy to execute and monitor multiple subprocesses in parallel. It is indeed trivial to modify the above example to run several commands simultaneously:

```
async def main():
    await asyncio.gather(
        run('ls /zzz'),
        run('sleep 1; echo "hello"'))

asyncio.run(main())
```

See also the Examples subsection.

Creating Subprocesses

async asyncio.create_subprocess_exec(*program*, **args*, *stdin=None*, *stdout=None*, *stderr=None*, *limit=None*, ***kwds*)

Create a subprocess.

The *limit* argument sets the buffer limit for **`StreamReader`** wrappers for **`stdout`** and **`stderr`** (if **`subprocess.PIPE`** is passed to *stdout* and *stderr* arguments).

Return a **`Process`** instance.

See the documentation of **`loop.subprocess_exec()`** for other parameters.

If the process object is garbage collected while the process is still running, the child process will be killed.

Changed in version 3.10: Removed the *loop* parameter.

async asyncio.create_subprocess_shell(*cmd*, *stdin=None*, *stdout=None*, *stderr=None*, *limit=None*, ***kwds*)

Run the *cmd* shell command.

The *limit* argument sets the buffer limit for `StreamReader` wrappers for `stdout` and `stderr` (if `subprocess.PIPE` is passed to *stdout* and *stderr* arguments).

Return a `Process` instance.

See the documentation of `loop.subprocess_shell()` for other parameters.

If the process object is garbage collected while the process is still running, the child process will be killed.

Important

It is the application's responsibility to ensure that all whitespace and special characters are quoted appropriately to avoid shell injection [https://en.wikipedia.org/wiki/Shell_injection#Shell_injection] vulnerabilities. The `shlex.quote()` function can be used to properly escape whitespace and special shell characters in strings that are going to be used to construct shell commands.

Changed in version 3.10: Removed the *loop* parameter.

Note

Subprocesses are available for Windows if a `ProactorEventLoop` is used. See Subprocess Support on Windows for details.

See also

asyncio also has the following *low-level* APIs to work with subprocesses: `loop.subprocess_exec()`, `loop.subprocess_shell()`, `loop.connect_read_pipe()`, `loop.connect_write_pipe()`, as well as the Subprocess Transports and Subprocess Protocols.

Constants

asyncio.subprocess.PIPE

Can be passed to the *stdin*, *stdout* or *stderr* parameters.

If *PIPE* is passed to *stdin* argument, the `Process.stdin` attribute will point to a `StreamWriter` instance.

If *PIPE* is passed to *stdout* or *stderr* arguments, the `Process.stdout` and `Process.stderr` attributes will point to `StreamReader` instances.

asyncio.subprocess.STDOUT

Special value that can be used as the *stderr* argument and indicates that standard error should be redirected into standard output.

asyncio.subprocess.DEVNULL

Special value that can be used as the *stdin*, *stdout* or *stderr* argument to process creation functions. It indicates that the special file `os.devnull` will be used for the corresponding subprocess stream.

Interacting with Subprocesses

Both `create_subprocess_exec()` and `create_subprocess_shell()` functions return instances of the *Process* class. *Process* is a high-level wrapper that allows communicating with subprocesses and watching for their completion.

class asyncio.subprocess.Process

An object that wraps OS processes created by the `create_subprocess_exec()` and `create_subprocess_shell()` functions.

This class is designed to have a similar API to the `subprocess.Popen` class, but there are some notable differences:

- unlike Popen, Process instances do not have an equivalent to the `poll()` method;
- the `communicate()` and `wait()` methods don't have a *timeout* parameter: use the `wait_for()` function;
- the `Process.wait()` method is asynchronous, whereas `subprocess.Popen.wait()` method is implemented as a blocking busy loop;
- the *universal_newlines* parameter is not supported.

This class is not thread safe.

See also the Subprocess and Threads section.

async wait()

Wait for the child process to terminate.

Set and return the `returncode` attribute.

Note

This method can deadlock when using `stdout=PIPE` or `stderr=PIPE` and the child process generates so much output that it blocks waiting for the OS pipe buffer to accept more data. Use the `communicate()` method when using pipes to avoid this condition.

async communicate(*input=None*)

Interact with process:

1. send data to *stdin* (if *input* is not `None`);
2. closes *stdin*;
3. read data from *stdout* and *stderr*, until EOF is reached;
4. wait for process to terminate.

The optional *input* argument is the data (`bytes` object) that will be sent to the child process.

Return a tuple `(stdout_data, stderr_data)`.

If either `BrokenPipeError` or `ConnectionResetError` exception is raised when writing *input* into *stdin*, the exception is ignored. This condition occurs when the process exits before all data are written into *stdin*.

If it is desired to send data to the process' *stdin*, the process needs to be created with `stdin=PIPE`. Similarly, to get anything other than `None` in the result tuple, the process has to be created with `stdout=PIPE` and/or `stderr=PIPE` arguments.

Note, that the data read is buffered in memory, so do not use this method if the data size is large or unlimited.

Changed in version 3.12: stdin gets closed when `input=None` too.

send_signal(*signal*)

Sends the signal *signal* to the child process.

Note

On Windows, `SIGTERM` is an alias for `terminate()`. `CTRL_C_EVENT` and `CTRL_BREAK_EVENT` can be sent to processes started with a *creationflags* parameter which includes `CREATE_NEW_PROCESS_GROUP`.

terminate()

Stop the child process.

On POSIX systems this method sends `SIGTERM` to the child process.

On Windows the Win32 API function **`TerminateProcess()`** is called to stop the child process.

kill()

Kill the child process.

On POSIX systems this method sends `SIGKILL` to the child process.

On Windows this method is an alias for `terminate()`.

stdin

Standard input stream (`StreamWriter`) or `None` if the process was created with `stdin=None`.

stdout

Standard output stream (`StreamReader`) or `None` if the process was created with `stdout=None`.

stderr

Standard error stream ([`StreamReader`]) or `None` if the process was created with `stderr=None`.

Warning

Use the `communicate()` method rather than `process.stdin.write()`, `await process.stdout.read()` or `await process.stderr.read()`. This avoids deadlocks due to streams pausing reading or writing and blocking the child process.

pid

Process identification number (PID).

Note that for processes created by the `create_subprocess_shell()` function, this attribute is the PID of the spawned shell.

returncode

Return code of the process when it exits.

A `None` value indicates that the process has not terminated yet.

A negative value `-N` indicates that the child was terminated by signal `N` (POSIX only).

Subprocess and Threads

Standard asyncio event loop supports running subprocesses from different threads by default.

On Windows subprocesses are provided by `ProactorEventLoop` only (default), `SelectorEventLoop` has no subprocess support.

Note that alternative event loop implementations might have own limitations; please refer to their documentation.

See also

The Concurrency and multithreading in asyncio section.

Examples

An example using the **`Process`** class to control a subprocess and the **`StreamReader`** class to read from its standard output.

The subprocess is created by the **`create_subprocess_exec()`** function:

```
import asyncio
import sys

async def get_date():
    code = 'import datetime; print(datetime.datetime.now())'

    # Create the subprocess; redirect the standard output
    # into a pipe.
    proc = await asyncio.create_subprocess_exec(
        sys.executable, '-c', code,
        stdout=asyncio.subprocess.PIPE)

    # Read one line of output.
    data = await proc.stdout.readline()
    line = data.decode('ascii').rstrip()

    # Wait for the subprocess exit.
    await proc.wait()
    return line

date = asyncio.run(get_date())
print(f"Current date: {date}")
```

See also the same example written using low-level APIs.

Queues

Source code: Lib/asyncio/queues.py
[https://github.com/python/cpython/tree/3.14/Lib/asyncio/queues.py]

---

asyncio queues are designed to be similar to classes of the **`queue`** module. Although asyncio queues are not thread-safe, they are designed to be used specifically in async/await code.

Note that methods of asyncio queues don’t have a *timeout* parameter; use **`asyncio.wait_for()`** function to do queue operations with a timeout.

See also the Examples section below.

Queue

class asyncio.Queue(*maxsize=0*)

A first in, first out (FIFO) queue.

If *maxsize* is less than or equal to zero, the queue size is infinite. If it is an integer greater than `0`, then `await put()` blocks when the queue reaches *maxsize* until an item is removed by **`get()`**.

Unlike the standard library threading **`queue`**, the size of the queue is always known and can be returned by calling the **`qsize()`** method.

Changed in version 3.10: Removed the *loop* parameter.

This class is not thread safe.

maxsize

Number of items allowed in the queue.

empty()

Return `True` if the queue is empty, `False` otherwise.

full()

Return `True` if there are **`maxsize`** items in the queue.

If the queue was initialized with `maxsize=0` (the default), then **`full()`** never returns `True`.

async get()

Remove and return an item from the queue. If queue is empty, wait until an item is available.

Raises **`QueueShutDown`** if the queue has been shut down and is empty, or if the queue has been shut down immediately.

get_nowait()

Return an item if one is immediately available, else raise **`QueueEmpty`**.

async join()

Block until all items in the queue have been received and processed.

The count of unfinished tasks goes up whenever an item is added to the queue. The count goes down whenever a consumer coroutine calls **`task_done()`** to indicate that the item was retrieved and all work on it is complete. When the count of unfinished tasks drops to zero, **`join()`** unblocks.

async put(*item*)

Put an item into the queue. If the queue is full, wait until a free slot is available before adding the item.

Raises `QueueShutDown` if the queue has been shut down.

put_nowait(*item*)

Put an item into the queue without blocking.

If no free slot is immediately available, raise `QueueFull`.

qsize()

Return the number of items in the queue.

shutdown(*immediate=False*)

Put a `Queue` instance into a shutdown mode.

The queue can no longer grow. Future calls to `put()` raise `QueueShutDown`. Currently blocked callers of `put()` will be unblocked and will raise `QueueShutDown` in the formerly awaiting task.

If *immediate* is false (the default), the queue can be wound down normally with `get()` calls to extract tasks that have already been loaded.

And if `task_done()` is called for each remaining task, a pending `join()` will be unblocked normally.

Once the queue is empty, future calls to `get()` will raise `QueueShutDown`.

If *immediate* is true, the queue is terminated immediately. The queue is drained to be completely empty and the count of unfinished tasks is reduced by the number of tasks drained. If unfinished tasks is zero, callers of `join()` are unblocked. Also, blocked callers of `get()` are unblocked and will raise `QueueShutDown` because the queue is empty.

Use caution when using `join()` with *immediate* set to true. This unblocks the join even when no work has been done on the tasks, violating the usual invariant for joining a queue.

Added in version 3.13.

task_done()

Indicate that a formerly enqueued work item is complete.

Used by queue consumers. For each `get()` used to fetch a work item, a subsequent call to `task_done()` tells the queue that the processing on the work item is complete.

If a `join()` is currently blocking, it will resume when all items have been processed (meaning that a `task_done()` call was received for every item that had been `put()` into the queue).

Raises `ValueError` if called more times than there were items placed in the queue.

Priority Queue

class asyncio.PriorityQueue

A variant of `Queue`; retrieves entries in priority order (lowest first).

Entries are typically tuples of the form `(priority_number, data)`.

LIFO Queue

class asyncio.LifoQueue

A variant of `Queue` that retrieves most recently added entries first (last in, first out).

Exceptions

exception asyncio.QueueEmpty

This exception is raised when the `get_nowait()` method is called on an empty queue.

exception asyncio.QueueFull

Exception raised when the `put_nowait()` method is called on a queue that has reached its *maxsize*.

exception asyncio.QueueShutDown

Exception raised when `put()` or `get()` is called on a queue which has been shut down.

Added in version 3.13.

Examples

Queues can be used to distribute workload between several concurrent tasks:

```
import asyncio
import random
import time


async def worker(name, queue):
    while True:
        # Get a "work item" out of the queue.
        sleep_for = await queue.get()
        # Sleep for the "sleep_for" seconds.
        await asyncio.sleep(sleep_for)

        # Notify the queue that the "work item" has been
processed.
        queue.task_done()

        print(f'{name} has slept for {sleep_for:.2f} seconds')


async def main():
    # Create a queue that we will use to store our "workload".
    queue = asyncio.Queue()

    # Generate random timings and put them into the queue.
    total_sleep_time = 0
    for _ in range(20):
        sleep_for = random.uniform(0.05, 1.0)
        total_sleep_time += sleep_for
        queue.put_nowait(sleep_for)

    # Create three worker tasks to process the queue
concurrently.
    tasks = []
    for i in range(3):
        task = asyncio.create_task(worker(f'worker-{i}',
queue))
        tasks.append(task)

    # Wait until the queue is fully processed.
    started_at = time.monotonic()
    await queue.join()
    total_slept_for = time.monotonic() - started_at

    # Cancel our worker tasks.
    for task in tasks:
        task.cancel()
    # Wait until all worker tasks are cancelled.
    await asyncio.gather(*tasks, return_exceptions=True)

    print('====')
    print(f'3 workers slept in parallel for
{total_slept_for:.2f} seconds')
    print(f'total expected sleep time: {total_sleep_time:.2f}
seconds')


asyncio.run(main())
```

Exceptions

Source code: Lib/asyncio/exceptions.py
[https://github.com/python/cpython/tree/3.14/Lib/asyncio/exceptions.py]

---

exception asyncio.TimeoutError

A deprecated alias of **`TimeoutError`**, raised when the operation has exceeded the given deadline.

Changed in version 3.11: This class was made an alias of **`TimeoutError`**.

exception asyncio.CancelledError

The operation has been cancelled.

This exception can be caught to perform custom operations when asyncio Tasks are cancelled. In almost all situations the exception must be re-raised.

Changed in version 3.8: **`CancelledError`** is now a subclass of **`BaseException`** rather than **`Exception`**.

exception asyncio.InvalidStateError

Invalid internal state of **`Task`** or **`Future`**.

Can be raised in situations like setting a result value for a *Future* object that already has a result value set.

exception asyncio.SendfileNotAvailableError

The “sendfile” syscall is not available for the given socket or file type.

A subclass of **`RuntimeError`**.

exception asyncio.IncompleteReadError

The requested read operation did not complete fully.

Raised by the asyncio stream APIs.

This exception is a subclass of **`EOFError`**.

expected

The total number (**`int`**) of expected bytes.

partial

A string of **`bytes`** read before the end of stream was reached.

exception asyncio.LimitOverrunError

Reached the buffer size limit while looking for a separator.

Raised by the asyncio stream APIs.

consumed

The total number of to be consumed bytes.

Call Graph Introspection

Source code: Lib/asyncio/graph.py
[https://github.com/python/cpython/tree/3.14/Lib/asyncio/graph.py]

---

asyncio has powerful runtime call graph introspection utilities to trace the entire call graph of a running *coroutine* or *task*, or a suspended *future*. These utilities and the underlying machinery can be used from within a Python program or by external profilers and debuggers.

Added in version 3.14.

asyncio.print_call_graph(*future=None*, */*, ***, *file=None*, *depth=1*, *limit=None*)

Print the async call graph for the current task or the provided **`Task`** or **`Future`**.

This function prints entries starting from the top frame and going down towards the invocation point.

The function receives an optional *future* argument. If not passed, the current running task will be used.

If the function is called on *the current task*, the optional keyword-only *depth* argument can be used to skip the specified number of frames from top of the stack.

If the optional keyword-only *limit* argument is provided, each call stack in the resulting graph is truncated to include at most `abs(limit)` entries. If *limit* is positive, the entries left are the closest to the invocation point. If *limit* is negative, the topmost entries are left. If *limit* is omitted or `None`, all entries are present. If *limit* is `0`, the call stack is not printed at all, only “awaited by” information is printed.

If *file* is omitted or `None`, the function will print to **`sys.stdout`**.

Example:

The following Python code:

```
import asyncio

async def test():
    asyncio.print_call_graph()

async def main():
    async with asyncio.TaskGroup() as g:
        g.create_task(test(), name='test')

asyncio.run(main())
```

will print:

```
* Task(name='test', id=0x1039f0fe0)
+ Call stack:
|   File 't2.py', line 4, in async test()
+ Awaited by:
   * Task(name='Task-1', id=0x103a5e060)
      + Call stack:
      |   File 'taskgroups.py', line 107, in async
TaskGroup.__aexit__()
      |   File 't2.py', line 7, in async main()
```

asyncio.format_call_graph(*future=None*, */*, ***, *depth=1*, *limit=None*)

Like **`print_call_graph()`**, but returns a string. If *future* is `None` and there’s no current task, the function returns an empty string.

asyncio.capture_call_graph(*future=None*, */*, ***, *depth=1*, *limit=None*)

Capture the async call graph for the current task or the provided `Task` or `Future`.

The function receives an optional *future* argument. If not passed, the current running task will be used. If there's no current task, the function returns `None`.

If the function is called on *the current task*, the optional keyword-only *depth* argument can be used to skip the specified number of frames from top of the stack.

Returns a `FutureCallGraph` data class object:

- `FutureCallGraph(future, call_stack, awaited_by)`

  Where *future* is a reference to a `Future` or a `Task` (or their subclasses.)

  `call_stack` is a tuple of `FrameCallGraphEntry` objects.

  `awaited_by` is a tuple of `FutureCallGraph` objects.

- `FrameCallGraphEntry(frame)`

  Where *frame* is a frame object of a regular Python function in the call stack.

Low level utility functions

To introspect an async call graph asyncio requires cooperation from control flow structures, such as `shield()` or `TaskGroup`. Any time an intermediate `Future` object with low-level APIs like `Future.add_done_callback()` is involved, the following two functions should be used to inform asyncio about how exactly such intermediate future objects are connected with the tasks they wrap or control.

asyncio.future_add_to_awaited_by(*future*, *waiter*, /)

Record that *future* is awaited on by *waiter*.

Both *future* and *waiter* must be instances of **`Future`** or **`Task`** or their subclasses, otherwise the call would have no effect.

A call to `future_add_to_awaited_by()` must be followed by an eventual call to the **`future_discard_from_awaited_by()`** function with the same arguments.

asyncio.future_discard_from_awaited_by(*future*, *waiter*, /)

Record that *future* is no longer awaited on by *waiter*.

Both *future* and *waiter* must be instances of **`Future`** or **`Task`** or their subclasses, otherwise the call would have no effect.

Event Loop

Source code: Lib/asyncio/events.py [https://github.com/python/cpython/tree/3.14/Lib/asyncio/events.py], Lib/asyncio/base_events.py [https://github.com/python/cpython/tree/3.14/Lib/asyncio/base_events.py]

Preface

The event loop is the core of every asyncio application. Event loops run asynchronous tasks and callbacks, perform network IO operations, and run subprocesses.

Application developers should typically use the high-level asyncio functions, such as `asyncio.run()`, and should rarely need to reference the loop object or call its methods. This section is intended mostly for authors of lower-level code, libraries, and frameworks, who need finer control over the event loop behavior.

Obtaining the Event Loop

The following low-level functions can be used to get, set, or create an event loop:

asyncio.get_running_loop()

Return the running event loop in the current OS thread.

Raise a `RuntimeError` if there is no running event loop.

This function can only be called from a coroutine or a callback.

Added in version 3.7.

asyncio.get_event_loop()

Get the current event loop.

When called from a coroutine or a callback (e.g. scheduled with call_soon or similar API), this function will always return the running event loop.

If there is no running event loop set, the function will return the result of the `get_event_loop_policy().get_event_loop()` call.

Because this function has rather complex behavior (especially when custom event loop policies are in use), using the **`get_running_loop()`** function is preferred to **`get_event_loop()`** in coroutines and callbacks.

As noted above, consider using the higher-level **`asyncio.run()`** function, instead of using these lower level functions to manually create and close an event loop.

Changed in version 3.14: Raises a **`RuntimeError`** if there is no current event loop.

Note

The **`asyncio`** policy system is deprecated and will be removed in Python 3.16; from there on, this function will return the current running event loop if present else it will return the loop set by **`set_event_loop()`**.

asyncio.set_event_loop(*loop*)

Set *loop* as the current event loop for the current OS thread.

asyncio.new_event_loop()

Create and return a new event loop object.

Note that the behaviour of `get_event_loop()`, `set_event_loop()`, and `new_event_loop()` functions can be altered by setting a custom event loop policy.

Contents

This documentation page contains the following sections:

- The Event Loop Methods section is the reference documentation of the event loop APIs;
- The Callback Handles section documents the `Handle` and `TimerHandle` instances which are returned from scheduling methods such as `loop.call_soon()` and `loop.call_later()`;
- The Server Objects section documents types returned from event loop methods like `loop.create_server()`;
- The Event Loop Implementations section documents the `SelectorEventLoop` and `ProactorEventLoop` classes;
- The Examples section showcases how to work with some event loop APIs.

Event Loop Methods

Event loops have **low-level** APIs for the following:

- Running and stopping the loop
- Scheduling callbacks
- Scheduling delayed callbacks
- Creating Futures and Tasks
- Opening network connections
- Creating network servers

- Transferring files
- TLS Upgrade
- Watching file descriptors
- Working with socket objects directly
- DNS
- Working with pipes
- Unix signals
- Executing code in thread or process pools
- Error Handling API
- Enabling debug mode
- Running Subprocesses

Running and stopping the loop

loop.run_until_complete(*future*)

Run until the *future* (an instance of `Future`) has completed.

If the argument is a coroutine object it is implicitly scheduled to run as a `asyncio.Task`.

Return the Future’s result or raise its exception.

loop.run_forever()

Run the event loop until `stop()` is called.

If `stop()` is called before `run_forever()` is called, the loop will poll the I/O selector once with a timeout of zero, run all callbacks scheduled in response to I/O events (and those that were already scheduled), and then exit.

If `stop()` is called while `run_forever()` is running, the loop will run the current batch of callbacks and then exit. Note that new callbacks scheduled by callbacks will not run in this case; instead, they will run the next time `run_forever()` or `run_until_complete()` is called.

loop.stop()

Stop the event loop.

loop.is_running()

Return `True` if the event loop is currently running.

loop.is_closed()

Return `True` if the event loop was closed.

loop.close()

Close the event loop.

The loop must not be running when this function is called. Any pending callbacks will be discarded.

This method clears all queues and shuts down the executor, but does not wait for the executor to finish.

This method is idempotent and irreversible. No other methods should be called after the event loop is closed.

async loop.shutdown_asyncgens()

Schedule all currently open asynchronous generator objects to close with an `aclose()` call. After calling this method, the event loop will issue a warning if a new asynchronous generator is iterated. This should be used to reliably finalize all scheduled asynchronous generators.

Note that there is no need to call this function when `asyncio.run()` is used.

Example:

```
try:
    loop.run_forever()
finally:
    loop.run_until_complete(loop.shutdown_asyncgens())
    loop.close()
```

Added in version 3.6.

async loop.shutdown_default_executor(*timeout=None*)

Schedule the closure of the default executor and wait for it to join all of the threads in the `ThreadPoolExecutor`. Once this method has been called, using the default executor with `loop.run_in_executor()` will raise a `RuntimeError`.

The *timeout* parameter specifies the amount of time (in `float` seconds) the executor will be given to finish joining. With the default, `None`, the executor is allowed an unlimited amount of time.

If the *timeout* is reached, a `RuntimeWarning` is emitted and the default executor is terminated without waiting for its threads to finish joining.

Note

Do not call this method when using `asyncio.run()`, as the latter handles default executor shutdown automatically.

Added in version 3.9.

Changed in version 3.12: Added the *timeout* parameter.

Scheduling callbacks

loop.call_soon(*callback*, **args*, *context=None*)

Schedule the *callback* callback to be called with *args* arguments at the next iteration of the event loop.

Return an instance of `asyncio.Handle`, which can be used later to cancel the callback.

Callbacks are called in the order in which they are registered. Each callback will be called exactly once.

The optional keyword-only *context* argument specifies a custom `contextvars.Context` for the *callback* to run in. Callbacks use the current context when no *context* is provided.

Unlike `call_soon_threadsafe()`, this method is not thread-safe.

loop.call_soon_threadsafe(*callback*, **args*, *context=None*)

A thread-safe variant of `call_soon()`. When scheduling callbacks from another thread, this function *must* be used, since `call_soon()` is not thread-safe.

This function is safe to be called from a reentrant context or signal handler, however, it is not safe or fruitful to use the returned handle in such contexts.

Raises `RuntimeError` if called on a loop that’s been closed. This can happen on a secondary thread when the main application is shutting down.

See the concurrency and multithreading section of the documentation.

Changed in version 3.7: The *context* keyword-only parameter was added. See **PEP 567** [https://peps.python.org/pep-0567/] for more details.

Note

Most `asyncio` scheduling functions don’t allow passing keyword arguments. To do that, use `functools.partial()`:

```
# will schedule "print("Hello", flush=True)"
loop.call_soon(
    functools.partial(print, "Hello", flush=True))
```

Using partial objects is usually more convenient than using lambdas, as asyncio can render partial objects better in debug and error messages.

Scheduling delayed callbacks

Event loop provides mechanisms to schedule callback functions to be called at some point in the future. Event loop uses monotonic clocks to track time.

loop.call_later(*delay*, *callback*, **args*, *context=None*)

Schedule *callback* to be called after the given *delay* number of seconds (can be either an int or a float).

An instance of `asyncio.TimerHandle` is returned which can be used to cancel the callback.

callback will be called exactly once. If two callbacks are scheduled for exactly the same time, the order in which they are called is undefined.

The optional positional *args* will be passed to the callback when it is called. If you want the callback to be called with keyword arguments use `functools.partial()`.

An optional keyword-only *context* argument allows specifying a custom `contextvars.Context` for the *callback* to run in. The current context is used when no *context* is provided.

Note

For performance, callbacks scheduled with `loop.call_later()` may run up to one clock-resolution early (see `time.get_clock_info('monotonic').resolution`).

Changed in version 3.7: The *context* keyword-only parameter was added. See **PEP 567** [https://peps.python.org/pep-0567/] for more details.

Changed in version 3.8: In Python 3.7 and earlier with the default event loop implementation, the *delay* could not exceed one day. This has been fixed in Python 3.8.

loop.call_at(*when*, *callback*, **args*, *context=None*)

Schedule *callback* to be called at the given absolute timestamp *when* (an int or a float), using the same time reference as `loop.time()`.

This method’s behavior is the same as `call_later()`.

An instance of `asyncio.TimerHandle` is returned which can be used to cancel the callback.

Note

For performance, callbacks scheduled with `loop.call_at()` may run up to one clock-resolution early (see `time.get_clock_info('monotonic').resolution`).

Changed in version 3.7: The *context* keyword-only parameter was added. See **PEP 567** [https://peps.python.org/pep-0567/] for more details.

Changed in version 3.8: In Python 3.7 and earlier with the default event loop implementation, the difference between *when* and the current time could not exceed one day. This has been fixed in Python 3.8.

loop.time()

Return the current time, as a `float` value, according to the event loop’s internal monotonic clock.

Note

Changed in version 3.8: In Python 3.7 and earlier timeouts (relative *delay* or absolute *when*) should not exceed one day. This has been fixed in Python 3.8.

See also

The `asyncio.sleep()` function.

Creating Futures and Tasks

loop.create_future()

Create an `asyncio.Future` object attached to the event loop.

This is the preferred way to create Futures in asyncio. This lets third-party event loops provide alternative implementations of the Future object (with better performance or instrumentation).

Added in version 3.5.2.

loop.create_task(*coro*, *, *name=None*, *context=None*, *eager_start=None*, ***kwargs*)

Schedule the execution of coroutine *coro*. Return a `Task` object.

Third-party event loops can use their own subclass of `Task` for interoperability. In this case, the result type is a subclass of `Task`.

The full function signature is largely the same as that of the `Task` constructor (or factory) - all of the keyword arguments to this function are passed through to that interface.

If the *name* argument is provided and not `None`, it is set as the name of the task using `Task.set_name()`.

An optional keyword-only *context* argument allows specifying a custom `contextvars.Context` for the *coro* to run in. The current context copy is created when no *context* is provided.

An optional keyword-only *eager_start* argument allows specifying if the task should execute eagerly during the call to create_task, or be scheduled later. If *eager_start* is not passed the mode set by `loop.set_task_factory()` will be used.

Changed in version 3.8: Added the *name* parameter.

Changed in version 3.11: Added the *context* parameter.

Changed in version 3.13.3: Added `kwargs` which passes on arbitrary extra parameters, including `name` and `context`.

Changed in version 3.13.4: Rolled back the change that passes on *name* and *context* (if it is None), while still passing on other arbitrary keyword arguments (to avoid breaking backwards compatibility with 3.13.3).

Changed in version 3.14: All *kwargs* are now passed on. The *eager_start* parameter works with eager task factories.

loop.set_task_factory(*factory*)

Set a task factory that will be used by `loop.create_task()`.

If *factory* is `None` the default task factory will be set. Otherwise, *factory* must be a *callable* with the signature matching `(loop, coro, **kwargs)`, where *loop* is a reference to the active event loop, and *coro* is a coroutine object. The callable must pass on all *kwargs*, and return a `asyncio.Task`-compatible object.

Changed in version 3.13.3: Required that all *kwargs* are passed on to `asyncio.Task`.

Changed in version 3.13.4: name is no longer passed to task factories. *context* is no longer passed to task factories if it is `None`.

Changed in version 3.14: name and *context* are now unconditionally passed on to task factories again.

loop.get_task_factory()

Return a task factory or `None` if the default one is in use.

Opening network connections

async loop.create_connection(*protocol_factory*, *host=None*, *port=None*, *, *ssl=None*, *family=0*, *proto=0*, *flags=0*, *sock=None*, *local_addr=None*, *server_hostname=None*, *ssl_handshake_timeout=None*, *ssl_shutdown_timeout=None*, *happy_eyeballs_delay=None*, *interleave=None*, *all_errors=False*)

Open a streaming transport connection to a given address specified by *host* and *port*.

The socket family can be either `AF_INET` or `AF_INET6` depending on *host* (or the *family* argument, if provided).

The socket type will be `SOCK_STREAM`.

protocol_factory must be a callable returning an asyncio protocol implementation.

This method will try to establish the connection in the background. When successful, it returns a `(transport, protocol)` pair.

The chronological synopsis of the underlying operation is as follows:

1. The connection is established and a transport is created for it.
2. *protocol_factory* is called without arguments and is expected to return a protocol instance.
3. The protocol instance is coupled with the transport by calling its `connection_made()` method.
4. A `(transport, protocol)` tuple is returned on success.

The created transport is an implementation-dependent bidirectional stream.

Other arguments:

- *ssl*: if given and not false, a SSL/TLS transport is created (by default a plain TCP transport is created). If *ssl* is a `ssl.SSLContext` object, this context is used to create the transport; if *ssl* is `True`, a default context returned from `ssl.create_default_context()` is used.

  See also

  SSL/TLS security considerations

- *server_hostname* sets or overrides the hostname that the target server’s certificate will be matched against. Should only be passed if *ssl* is not `None`. By default the value of the *host* argument is used. If *host* is empty, there is no default and you must pass a value for *server_hostname*. If *server_hostname* is an empty string, hostname matching is disabled (which is a serious security risk, allowing for potential man-in-the-middle attacks).

- *family*, *proto*, *flags* are the optional address family, protocol and flags to be passed through to getaddrinfo() for *host* resolution. If given, these should all be integers from the corresponding `socket` module constants.

- *happy_eyeballs_delay*, if given, enables Happy Eyeballs for this connection. It should be a floating-point number representing the amount of time in seconds to wait for a connection attempt to complete, before starting the next attempt in parallel. This is the “Connection Attempt Delay” as defined in **RFC 8305** [https://datatracker.ietf.org/doc/html/rfc8305.html]. A sensible default value recommended by the RFC is `0.25` (250 milliseconds).

- *interleave* controls address reordering when a host name resolves to multiple IP addresses. If `0` or unspecified, no reordering is done, and addresses are tried in the order returned by `getaddrinfo()`. If a positive integer is specified, the addresses are interleaved by address family, and the given integer is interpreted as “First Address Family Count” as defined in **RFC 8305** [https://datatracker.ietf.org/doc/html/rfc8305.html]. The default is

`0` if *happy_eyeballs_delay* is not specified, and `1` if it is.

- *sock*, if given, should be an existing, already connected `socket.socket` object to be used by the transport. If *sock* is given, none of *host*, *port*, *family*, *proto*, *flags*, *happy_eyeballs_delay*, *interleave* and *local_addr* should be specified.

  Note

  The *sock* argument transfers ownership of the socket to the transport created. To close the socket, call the transport's `close()` method.

- *local_addr*, if given, is a `(local_host, local_port)` tuple used to bind the socket locally. The *local_host* and *local_port* are looked up using `getaddrinfo()`, similarly to *host* and *port*.

- *ssl_handshake_timeout* is (for a TLS connection) the time in seconds to wait for the TLS handshake to complete before aborting the connection. `60.0` seconds if `None` (default).

- *ssl_shutdown_timeout* is the time in seconds to wait for the SSL shutdown to complete before aborting the connection. `30.0` seconds if `None` (default).

- *all_errors* determines what exceptions are raised when a connection cannot be created. By default, only a single `Exception` is raised: the first exception if there is only one or all errors have same message, or a single `OSError` with the error messages combined. When `all_errors` is `True`, an `ExceptionGroup` will be raised containing all exceptions (even if there is only one).

Changed in version 3.5: Added support for SSL/TLS in `ProactorEventLoop`.

Changed in version 3.6: The socket option socket.TCP_NODELAY is set by default for all TCP connections.

Changed in version 3.7: Added the *ssl_handshake_timeout* parameter.

Changed in version 3.8: Added the *happy_eyeballs_delay* and *interleave* parameters.

Happy Eyeballs Algorithm: Success with Dual-Stack Hosts. When a server’s IPv4 path and protocol are working, but the server’s IPv6 path and protocol are not working, a dual-stack client application experiences significant connection delay compared to an IPv4-only client. This is undesirable because it causes the dual-stack client to have a worse user experience. This document specifies requirements for algorithms that reduce this user-visible delay and provides an algorithm.

For more information: https://datatracker.ietf.org/doc/html/rfc6555

Changed in version 3.11: Added the *ssl_shutdown_timeout* parameter.

Changed in version 3.12: all_errors was added.

See also

The `open_connection()` function is a high-level alternative API. It returns a pair of (`StreamReader`, `StreamWriter`) that can be used directly in async/await code.

async loop.create_datagram_endpoint(*protocol_factory*, *local_addr=None*, *remote_addr=None*, ***, *family=0*, *proto=0*, *flags=0*, *reuse_port=None*, *allow_broadcast=None*, *sock=None*)

Create a datagram connection.

The socket family can be either **`AF_INET`**, **`AF_INET6`**, or **`AF_UNIX`**, depending on *host* (or the *family* argument, if provided).

The socket type will be **`SOCK_DGRAM`**.

protocol_factory must be a callable returning a protocol implementation.

A tuple of `(transport, protocol)` is returned on success.

Other arguments:

- *local_addr*, if given, is a `(local_host, local_port)` tuple used to bind the socket locally. The *local_host* and *local_port* are looked up using **`getaddrinfo()`**.

  Note

  On Windows, when using the proactor event loop with `local_addr=None`, an **`OSError`** with **`errno.WSAEINVAL`** will be raised when running it.

- *remote_addr*, if given, is a `(remote_host, remote_port)` tuple used to connect the socket to a remote address. The *remote_host* and *remote_port* are looked up using **`getaddrinfo()`**.
- *family*, *proto*, *flags* are the optional address family, protocol and flags to be passed through to

`getaddrinfo()` for *host* resolution. If given, these should all be integers from the corresponding `socket` module constants.

- *reuse_port* tells the kernel to allow this endpoint to be bound to the same port as other existing endpoints are bound to, so long as they all set this flag when being created. This option is not supported on Windows and some Unixes. If the socket.SO_REUSEPORT constant is not defined then this capability is unsupported.
- *allow_broadcast* tells the kernel to allow this endpoint to send messages to the broadcast address.
- *sock* can optionally be specified in order to use a preexisting, already connected, `socket.socket` object to be used by the transport. If specified, *local_addr* and *remote_addr* should be omitted (must be `None`).

  Note

  The *sock* argument transfers ownership of the socket to the transport created. To close the socket, call the transport's `close()` method.

See UDP echo client protocol and UDP echo server protocol examples.

Changed in version 3.4.4: The *family*, *proto*, *flags*, *reuse_address*, *reuse_port*, *allow_broadcast*, and *sock* parameters were added.

Changed in version 3.8: Added support for Windows.

Changed in version 3.8.1: The *reuse_address* parameter is no longer supported, as using socket.SO_REUSEADDR poses a significant security concern for UDP. Explicitly passing `reuse_address=True` will raise an exception.

When multiple processes with differing UIDs assign sockets to an identical UDP socket address with `SO_REUSEADDR`, incoming packets can become randomly distributed among the sockets.

For supported platforms, *reuse_port* can be used as a replacement for similar functionality. With *reuse_port*, socket.SO_REUSEPORT is used instead, which specifically prevents processes with differing UIDs from assigning sockets to the same socket address.
Changed in version 3.11: The *reuse_address* parameter, disabled since Python 3.8.1, 3.7.6 and 3.6.10, has been entirely removed.

async loop.create_unix_connection(*protocol_factory*, *path=None*, ***, *ssl=None*, *sock=None*, *server_hostname=None*, *ssl_handshake_timeout=None*, *ssl_shutdown_timeout=None*)

Create a Unix connection.

The socket family will be **`AF_UNIX`**; socket type will be **`SOCK_STREAM`**.

A tuple of `(transport, protocol)` is returned on success.

path is the name of a Unix domain socket and is required, unless a *sock* parameter is specified. Abstract Unix sockets, **`str`**, **`bytes`**, and **`Path`** paths are supported.

See the documentation of the **`loop.create_connection()`** method for information about arguments to this method.

Availability: Unix.

Changed in version 3.7: Added the *ssl_handshake_timeout* parameter. The *path* parameter can now be a path-like object.

Changed in version 3.11: Added the *ssl_shutdown_timeout* parameter.

Creating network servers

async loop.create_server(*protocol_factory*, *host=None*, *port=None*, *, *family=socket.AF_UNSPEC*, *flags=socket.AI_PASSIVE*, *sock=None*, *backlog=100*, *ssl=None*, *reuse_address=None*, *reuse_port=None*, *keep_alive=None*, *ssl_handshake_timeout=None*, *ssl_shutdown_timeout=None*, *start_serving=True*)

Create a TCP server (socket type **`SOCK_STREAM`**) listening on *port* of the *host* address.

Returns a **`Server`** object.

Arguments:

- *protocol_factory* must be a callable returning a protocol implementation.
- The *host* parameter can be set to several types which determine where the server would be listening:
    - If *host* is a string, the TCP server is bound to a single network interface specified by *host*.
    - If *host* is a sequence of strings, the TCP server is bound to all network interfaces specified by the sequence.
    - If *host* is an empty string or `None`, all interfaces are assumed and a list of multiple sockets will be

returned (most likely one for IPv4 and another one for IPv6).

- The *port* parameter can be set to specify which port the server should listen on. If `0` or `None` (the default), a random unused port will be selected (note that if *host* resolves to multiple network interfaces, a different random port will be selected for each interface).

- *family* can be set to either `socket.AF_INET` or `AF_INET6` to force the socket to use IPv4 or IPv6. If not set, the *family* will be determined from host name (defaults to `AF_UNSPEC`).

- *flags* is a bitmask for `getaddrinfo()`.

- *sock* can optionally be specified in order to use a preexisting socket object. If specified, *host* and *port* must not be specified.

  Note

  The *sock* argument transfers ownership of the socket to the server created. To close the socket, call the server's `close()` method.

- *backlog* is the maximum number of queued connections passed to `listen()` (defaults to 100).

- *ssl* can be set to an `SSLContext` instance to enable TLS over the accepted connections.

- *reuse_address* tells the kernel to reuse a local socket in `TIME_WAIT` state, without waiting for its natural timeout to expire. If not specified will automatically be set to `True` on Unix.

- *reuse_port* tells the kernel to allow this endpoint to be bound to the same port as other existing endpoints are bound to, so long as they all set this flag when being created. This option is not supported on Windows.

- *keep_alive* set to `True` keeps connections active by enabling the periodic transmission of messages.

Changed in version 3.13: Added the *keep_alive* parameter.

- *ssl_handshake_timeout* is (for a TLS server) the time in seconds to wait for the TLS handshake to complete before aborting the connection. `60.0` seconds if `None` (default).
- *ssl_shutdown_timeout* is the time in seconds to wait for the SSL shutdown to complete before aborting the connection. `30.0` seconds if `None` (default).
- *start_serving* set to `True` (the default) causes the created server to start accepting connections immediately. When set to `False`, the user should await on **`Server.start_serving()`** or **`Server.serve_forever()`** to make the server to start accepting connections.

Changed in version 3.5: Added support for SSL/TLS in **`ProactorEventLoop`**.

Changed in version 3.5.1: The *host* parameter can be a sequence of strings.

Changed in version 3.6: Added *ssl_handshake_timeout* and *start_serving* parameters. The socket option socket.TCP_NODELAY is set by default for all TCP connections.

Changed in version 3.11: Added the *ssl_shutdown_timeout* parameter.

See also

The `start_server()` function is a higher-level alternative API that returns a pair of `StreamReader` and `StreamWriter` that can be used in an async/await code.

async loop.create_unix_server(*protocol_factory*, *path=None*, ***, *sock=None*, *backlog=100*, *ssl=None*, *ssl_handshake_timeout=None*, *ssl_shutdown_timeout=None*, *start_serving=True*, *cleanup_socket=True*)

Similar to `loop.create_server()` but works with the `AF_UNIX` socket family.

path is the name of a Unix domain socket, and is required, unless a *sock* argument is provided. Abstract Unix sockets, `str`, `bytes`, and `Path` paths are supported.

If *cleanup_socket* is true then the Unix socket will automatically be removed from the filesystem when the server is closed, unless the socket has been replaced after the server has been created.

See the documentation of the `loop.create_server()` method for information about arguments to this method.

Availability: Unix.

Changed in version 3.7: Added the *ssl_handshake_timeout* and *start_serving* parameters. The *path* parameter can now be a `Path` object.

Changed in version 3.11: Added the *ssl_shutdown_timeout* parameter.

Changed in version 3.13: Added the *cleanup_socket* parameter.

async loop.connect_accepted_socket(*protocol_factory*, *sock*, ***, *ssl=None*, *ssl_handshake_timeout=None*, *ssl_shutdown_timeout=None*)

Wrap an already accepted connection into a transport/protocol pair.

This method can be used by servers that accept connections outside of asyncio but that use asyncio to handle them.

Parameters:

- *protocol_factory* must be a callable returning a protocol implementation.
- *sock* is a preexisting socket object returned from `socket.accept`.

  Note

  The *sock* argument transfers ownership of the socket to the transport created. To close the socket, call the transport's `close()` method.

- *ssl* can be set to an `SSLContext` to enable SSL over the accepted connections.
- *ssl_handshake_timeout* is (for an SSL connection) the time in seconds to wait for the SSL handshake to complete before aborting the connection. `60.0` seconds if `None` (default).

- *ssl_shutdown_timeout* is the time in seconds to wait for the SSL shutdown to complete before aborting the connection. `30.0` seconds if `None` (default).

Returns a `(transport, protocol)` pair.

Added in version 3.5.3.

Changed in version 3.7: Added the *ssl_handshake_timeout* parameter.

Changed in version 3.11: Added the *ssl_shutdown_timeout* parameter.

Transferring files

async loop.sendfile(*transport*, *file*, *offset=0*, *count=None*, ***, *fallback=True*)

Send a *file* over a *transport*. Return the total number of bytes sent.

The method uses high-performance `os.sendfile()` if available.

file must be a regular file object opened in binary mode.

offset tells from where to start reading the file. If specified, *count* is the total number of bytes to transmit as opposed to sending the file until EOF is reached. File position is always updated, even when this method raises an error, and `file.tell()` can be used to obtain the actual number of bytes sent.

fallback set to `True` makes asyncio to manually read and send the file when the platform does not support the sendfile system call (e.g. Windows or SSL socket on Unix).

Raise `SendfileNotAvailableError` if the system does not support the *sendfile* syscall and *fallback* is `False`.

Added in version 3.7.

TLS Upgrade

async loop.start_tls(*transport*, *protocol*, *sslcontext*, *, *server_side=False*, *server_hostname=None*, *ssl_handshake_timeout=None*, *ssl_shutdown_timeout=None*)

Upgrade an existing transport-based connection to TLS.

Create a TLS coder/decoder instance and insert it between the *transport* and the *protocol*. The coder/decoder implements both *transport*-facing protocol and *protocol*-facing transport.

Return the created two-interface instance. After *await*, the *protocol* must stop using the original *transport* and communicate with the returned object only because the coder caches *protocol*-side data and sporadically exchanges extra TLS session packets with *transport*.

In some situations (e.g. when the passed transport is already closing) this may return `None`.

Parameters:

- *transport* and *protocol* instances that methods like `create_server()` and `create_connection()` return.
- *sslcontext*: a configured instance of `SSLContext`.
- *server_side* pass `True` when a server-side connection is being upgraded (like the one created by `create_server()`).

- *server_hostname*: sets or overrides the host name that the target server’s certificate will be matched against.
- *ssl_handshake_timeout* is (for a TLS connection) the time in seconds to wait for the TLS handshake to complete before aborting the connection. `60.0` seconds if `None` (default).
- *ssl_shutdown_timeout* is the time in seconds to wait for the SSL shutdown to complete before aborting the connection. `30.0` seconds if `None` (default).

Added in version 3.7.

Changed in version 3.11: Added the *ssl_shutdown_timeout* parameter.

Watching file descriptors

loop.add_reader(*fd*, *callback*, **args*)

Start monitoring the *fd* file descriptor for read availability and invoke *callback* with the specified arguments once *fd* is available for reading.

Any preexisting callback registered for *fd* is cancelled and replaced by *callback*.

loop.remove_reader(*fd*)

Stop monitoring the *fd* file descriptor for read availability. Returns `True` if *fd* was previously being monitored for reads.

loop.add_writer(*fd*, *callback*, **args*)

Start monitoring the *fd* file descriptor for write availability and invoke *callback* with the specified arguments once *fd* is available for writing.

Any preexisting callback registered for *fd* is cancelled and replaced by *callback*.

Use **`functools.partial()`** to pass keyword arguments to *callback*.

loop.remove_writer(*fd*)

Stop monitoring the *fd* file descriptor for write availability. Returns `True` if *fd* was previously being monitored for writes.

See also Platform Support section for some limitations of these methods.

Working with socket objects directly

In general, protocol implementations that use transport-based APIs such as **`loop.create_connection()`** and **`loop.create_server()`** are faster than implementations that work with sockets directly. However, there are some use cases when performance is not critical, and working with **`socket`** objects directly is more convenient.

async loop.sock_recv(*sock*, *nbytes*)

Receive up to *nbytes* from *sock*. Asynchronous version of **`socket.recv()`**.

Return the received data as a bytes object.

sock must be a non-blocking socket.

Changed in version 3.7: Even though this method was always documented as a coroutine method, releases before Python 3.7 returned a **`Future`**. Since Python 3.7 this is an `async def` method.

async loop.sock_recv_into(*sock*, *buf*)

Receive data from *sock* into the *buf* buffer. Modeled after the blocking `socket.recv_into()` method.

Return the number of bytes written to the buffer.

sock must be a non-blocking socket.

Added in version 3.7.

async loop.sock_recvfrom(*sock*, *bufsize*)

Receive a datagram of up to *bufsize* from *sock*. Asynchronous version of `socket.recvfrom()`.

Return a tuple of (received data, remote address).

sock must be a non-blocking socket.

Added in version 3.11.

async loop.sock_recvfrom_into(*sock*, *buf*, *nbytes=0*)

Receive a datagram of up to *nbytes* from *sock* into *buf*. Asynchronous version of `socket.recvfrom_into()`.

Return a tuple of (number of bytes received, remote address).

sock must be a non-blocking socket.

Added in version 3.11.

async loop.sock_sendall(*sock*, *data*)

Send *data* to the *sock* socket. Asynchronous version of `socket.sendall()`.

This method continues to send to the socket until either all data in *data* has been sent or an error occurs. `None` is returned on success. On error, an exception is raised. Additionally, there is no way to determine how much data, if any, was successfully processed by the receiving end of the connection.

sock must be a non-blocking socket.

Changed in version 3.7: Even though the method was always documented as a coroutine method, before Python 3.7 it returned a `Future`. Since Python 3.7, this is an `async def` method.

async loop.sock_sendto(*sock*, *data*, *address*)

Send a datagram from *sock* to *address*. Asynchronous version of `socket.sendto()`.

Return the number of bytes sent.

sock must be a non-blocking socket.

Added in version 3.11.

async loop.sock_connect(*sock*, *address*)

Connect *sock* to a remote socket at *address*.

Asynchronous version of `socket.connect()`.

sock must be a non-blocking socket.

Changed in version 3.5.2: `address` no longer needs to be resolved. `sock_connect` will try to check if the *address* is already resolved by calling `socket.inet_pton()`. If not, `loop.getaddrinfo()` will be used to resolve the *address*.

See also

`loop.create_connection()` and `asyncio.open_connection()`.

async loop.sock_accept(*sock*)

Accept a connection. Modeled after the blocking `socket.accept()` method.

The socket must be bound to an address and listening for connections. The return value is a pair `(conn, address)` where *conn* is a *new* socket object usable to send and receive data on the connection, and *address* is the address bound to the socket on the other end of the connection.

sock must be a non-blocking socket.

Changed in version 3.7: Even though the method was always documented as a coroutine method, before Python 3.7 it returned a `Future`. Since Python 3.7, this is an `async def` method.

See also

`loop.create_server()` and `start_server()`.

async loop.sock_sendfile(*sock*, *file*, *offset=0*, *count=None*, *, *fallback=True*)

Send a file using high-performance `os.sendfile` if possible. Return the total number of bytes sent.

Asynchronous version of `socket.sendfile()`.

sock must be a non-blocking `socket.SOCK_STREAM` `socket`.

file must be a regular file object open in binary mode.

offset tells from where to start reading the file. If specified, *count* is the total number of bytes to transmit as opposed to sending the file until EOF is reached. File position is always updated, even when this method raises an error, and `file.tell()` can be used to obtain the actual number of bytes sent.

fallback, when set to `True`, makes asyncio manually read and send the file when the platform does not support the sendfile syscall (e.g. Windows or SSL socket on Unix).

Raise `SendfileNotAvailableError` if the system does not support *sendfile* syscall and *fallback* is `False`.

sock must be a non-blocking socket.

Added in version 3.7.

DNS

async loop.getaddrinfo(*host*, *port*, ***, *family=0*, *type=0*, *proto=0*, *flags=0*)

Asynchronous version of `socket.getaddrinfo()`.

async loop.getnameinfo(*sockaddr*, *flags=0*)

Asynchronous version of `socket.getnameinfo()`.

Note

Both *getaddrinfo* and *getnameinfo* internally utilize their synchronous versions through the loop's default thread pool executor. When this executor is saturated, these methods may experience delays, which higher-level networking libraries may report as increased timeouts. To mitigate this, consider using a custom executor for other user tasks, or setting a default executor with a larger number of workers.

Changed in version 3.7: Both *getaddrinfo* and *getnameinfo* methods were always documented to return a coroutine, but prior to Python 3.7 they were, in fact, returning `asyncio.Future` objects. Starting with Python 3.7 both methods are coroutines.

Working with pipes

async loop.connect_read_pipe(*protocol_factory*, *pipe*)

Register the read end of *pipe* in the event loop.

protocol_factory must be a callable returning an asyncio protocol implementation.

pipe is a file-like object.

Return pair `(transport, protocol)`, where *transport* supports the `ReadTransport` interface and *protocol* is an object instantiated by the *protocol_factory*.

With `SelectorEventLoop` event loop, the *pipe* is set to non-blocking mode.

async loop.connect_write_pipe(*protocol_factory*, *pipe*)

Register the write end of *pipe* in the event loop.

protocol_factory must be a callable returning an asyncio protocol implementation.

pipe is file-like object.

Return pair `(transport, protocol)`, where *transport* supports **`WriteTransport`** interface and *protocol* is an object instantiated by the *protocol_factory*.

With **`SelectorEventLoop`** event loop, the *pipe* is set to non-blocking mode.

Note

`SelectorEventLoop` does not support the above methods on Windows. Use **`ProactorEventLoop`** instead for Windows.

See also

The **`loop.subprocess_exec()`** and **`loop.subprocess_shell()`** methods.

Unix signals

loop.add_signal_handler(*signum*, *callback*, **args*)

Set *callback* as the handler for the *signum* signal.

The callback will be invoked by *loop*, along with other queued callbacks and runnable coroutines of that event loop. Unlike signal handlers registered using **`signal.signal()`**, a callback registered with this function is allowed to interact with the event loop.

Raise **`ValueError`** if the signal number is invalid or uncatchable. Raise **`RuntimeError`** if there is a problem setting up the handler.

Use **`functools.partial()`** to pass keyword arguments to *callback*.

Like `signal.signal()`, this function must be invoked in the main thread.

loop.remove_signal_handler(*sig*)

Remove the handler for the *sig* signal.

Return `True` if the signal handler was removed, or `False` if no handler was set for the given signal.

Availability: Unix.

See also

The `signal` module.

Executing code in thread or process pools

awaitable loop.run_in_executor(*executor*, *func*, **args*)

Arrange for *func* to be called in the specified executor.

The *executor* argument should be an `concurrent.futures.Executor` instance. The default executor is used if *executor* is `None`. The default executor can be set by `loop.set_default_executor()`, otherwise, a `concurrent.futures.ThreadPoolExecutor` will be lazy-initialized and used by `run_in_executor()` if needed.

Example:

```
import asyncio
import concurrent.futures

def blocking_io():
    # File operations (such as logging) can block the
    # event loop: run them in a thread pool.
    with open('/dev/urandom', 'rb') as f:
        return f.read(100)

def cpu_bound():
    # CPU-bound operations will block the event loop:
    # in general it is preferable to run them in a
    # process pool.
    return sum(i * i for i in range(10 ** 7))

async def main():
    loop = asyncio.get_running_loop()

    ## Options:

    # 1. Run in the default loop's executor:
    result = await loop.run_in_executor(
        None, blocking_io)
    print('default thread pool', result)

    # 2. Run in a custom thread pool:
    with concurrent.futures.ThreadPoolExecutor() as pool:
        result = await loop.run_in_executor(
            pool, blocking_io)
        print('custom thread pool', result)

    # 3. Run in a custom process pool:
    with concurrent.futures.ProcessPoolExecutor() as pool:
        result = await loop.run_in_executor(
            pool, cpu_bound)
        print('custom process pool', result)

    # 4. Run in a custom interpreter pool:
    with concurrent.futures.InterpreterPoolExecutor() as
pool:
        result = await loop.run_in_executor(
            pool, cpu_bound)
        print('custom interpreter pool', result)

if __name__ == '__main__':
    asyncio.run(main())
```

Note that the entry point guard (`if __name__ == '__main__'`) is required for option 3 due to the peculiarities of `multiprocessing`, which is used by `ProcessPoolExecutor`. See Safe importing of main module.

This method returns a `asyncio.Future` object.

Use `functools.partial()` to pass keyword arguments to *func*.

Changed in version 3.5.3: `loop.run_in_executor()` no longer configures the `max_workers` of the thread pool executor it creates, instead leaving it up to the thread pool executor (`ThreadPoolExecutor`) to set the default.

loop.set_default_executor(*executor*)

Set *executor* as the default executor used by `run_in_executor()`. *executor* must be an instance of `ThreadPoolExecutor`, which includes `InterpreterPoolExecutor`.

Changed in version 3.11: executor must be an instance of `ThreadPoolExecutor`.

Error Handling API

Allows customizing how exceptions are handled in the event loop.

loop.set_exception_handler(*handler*)

Set *handler* as the new event loop exception handler.

If *handler* is `None`, the default exception handler will be set. Otherwise, *handler* must be a callable with the signature matching `(loop, context)`, where `loop` is a reference to the active event loop, and `context` is a `dict` object containing the details of the exception (see `call_exception_handler()` documentation for details about context).

If the handler is called on behalf of a `Task` or `Handle`, it is run in the `contextvars.Context` of that task or callback handle.

Changed in version 3.12: The handler may be called in the `Context` of the task or handle where the exception originated.

loop.get_exception_handler()

Return the current exception handler, or `None` if no custom exception handler was set.

Added in version 3.5.2.

loop.default_exception_handler(*context*)

Default exception handler.

This is called when an exception occurs and no exception handler is set. This can be called by a custom exception handler that wants to defer to the default handler behavior.

context parameter has the same meaning as in `call_exception_handler()`.

loop.call_exception_handler(*context*)

Call the current event loop exception handler.

context is a `dict` object containing the following keys (new keys may be introduced in future Python versions):

- ‘message’: Error message;
- ‘exception’ (optional): Exception object;
- ‘future’ (optional): `asyncio.Future` instance;
- ‘task’ (optional): `asyncio.Task` instance;
- ‘handle’ (optional): `asyncio.Handle` instance;

- ‘protocol’ (optional): Protocol instance;
- ‘transport’ (optional): Transport instance;
- ‘socket’ (optional): `socket.socket` instance;
- ‘source_traceback’ (optional): Traceback of the source;
- ‘handle_traceback’ (optional): Traceback of the handle;
- ‘asyncgen’ (optional): Asynchronous generator that caused

    the exception.

Note

This method should not be overloaded in subclassed event loops. For custom exception handling, use the `set_exception_handler()` method.

Enabling debug mode

loop.get_debug()

Get the debug mode (`bool`) of the event loop.

The default value is `True` if the environment variable `PYTHONASYNCIODEBUG` is set to a non-empty string, `False` otherwise.

loop.set_debug(*enabled: bool*)

Set the debug mode of the event loop.

Changed in version 3.7: The new Python Development Mode can now also be used to enable the debug mode.

loop.slow_callback_duration

This attribute can be used to set the minimum execution duration in seconds that is considered “slow”. When debug mode is enabled, “slow” callbacks are logged.

Default value is 100 milliseconds.

See also

The debug mode of asyncio.

Running Subprocesses

Methods described in this subsections are low-level. In regular async/await code consider using the high-level `asyncio.create_subprocess_shell()` and `asyncio.create_subprocess_exec()` convenience functions instead.

Note

On Windows, the default event loop `ProactorEventLoop` supports subprocesses, whereas `SelectorEventLoop` does not. See Subprocess Support on Windows for details.

async loop.subprocess_exec(*protocol_factory*, **args*, *stdin=subprocess.PIPE*, *stdout=subprocess.PIPE*, *stderr=subprocess.PIPE*, ***kwargs*)

Create a subprocess from one or more string arguments specified by *args*.

args must be a list of strings represented by:

- `str`;
- or `bytes`, encoded to the filesystem encoding.

The first string specifies the program executable, and the remaining strings specify the arguments. Together, string arguments form the `argv` of the program.

This is similar to the standard library `subprocess.Popen` class called with `shell=False` and the list of strings passed as the first argument; however, where `Popen` takes a single argument which is list of strings, *subprocess_exec* takes multiple string arguments.

The *protocol_factory* must be a callable returning a subclass of the `asyncio.SubprocessProtocol` class.

Other parameters:

- *stdin* can be any of these:
  - a file-like object
  - an existing file descriptor (a positive integer), for example those created with `os.pipe()`
  - the `subprocess.PIPE` constant (default) which will create a new pipe and connect it,
  - the value `None` which will make the subprocess inherit the file descriptor from this process
  - the `subprocess.DEVNULL` constant which indicates that the special `os.devnull` file will be used
- *stdout* can be any of these:
  - a file-like object
  - the `subprocess.PIPE` constant (default) which will create a new pipe and connect it,
  - the value `None` which will make the subprocess inherit the file descriptor from this process
  - the `subprocess.DEVNULL` constant which indicates that the special `os.devnull` file will be used
- *stderr* can be any of these:
  - a file-like object
  - the `subprocess.PIPE` constant (default) which will create a new pipe and connect it,

  - the value `None` which will make the subprocess inherit the file descriptor from this process
  - the `subprocess.DEVNULL` constant which indicates that the special `os.devnull` file will be used
  - the `subprocess.STDOUT` constant which will connect the standard error stream to the process' standard output stream
- All other keyword arguments are passed to `subprocess.Popen` without interpretation, except for *bufsize*, *universal_newlines*, *shell*, *text*, *encoding* and *errors*, which should not be specified at all.

  The `asyncio` subprocess API does not support decoding the streams as text. `bytes.decode()` can be used to convert the bytes returned from the stream to text.

If a file-like object passed as *stdin*, *stdout* or *stderr* represents a pipe, then the other side of this pipe should be registered with `connect_write_pipe()` or `connect_read_pipe()` for use with the event loop.

See the constructor of the `subprocess.Popen` class for documentation on other arguments.

Returns a pair of `(transport, protocol)`, where *transport* conforms to the `asyncio.SubprocessTransport` base class and *protocol* is an object instantiated by the *protocol_factory*.

If the transport is closed or is garbage collected, the child process is killed if it is still running.

async loop.subprocess_shell(*protocol_factory*, *cmd*, *, *stdin=subprocess.PIPE*, *stdout=subprocess.PIPE*, *stderr=subprocess.PIPE*, ***kwargs*)

Create a subprocess from *cmd*, which can be a `str` or a `bytes` string encoded to the filesystem encoding, using the platform's "shell" syntax.

This is similar to the standard library `subprocess.Popen` class called with `shell=True`.

The *protocol_factory* must be a callable returning a subclass of the `SubprocessProtocol` class.

See `subprocess_exec()` for more details about the remaining arguments.

Returns a pair of `(transport, protocol)`, where *transport* conforms to the `SubprocessTransport` base class and *protocol* is an object instantiated by the *protocol_factory*.

If the transport is closed or is garbage collected, the child process is killed if it is still running.

Note

It is the application's responsibility to ensure that all whitespace and special characters are quoted appropriately to avoid shell injection [https://en.wikipedia.org/wiki/Shell_injection#Shell_injection] vulnerabilities. The `shlex.quote()` function can be used to properly escape whitespace and special characters in strings that are going to be used to construct shell commands.

Callback Handles

class asyncio.Handle

A callback wrapper object returned by `loop.call_soon()`, `loop.call_soon_threadsafe()`.

get_context()

Return the `contextvars.Context` object associated with the handle.

Added in version 3.12.

cancel()

Cancel the callback. If the callback has already been canceled or executed, this method has no effect.

cancelled()

Return `True` if the callback was cancelled.

Added in version 3.7.

class asyncio.TimerHandle

A callback wrapper object returned by `loop.call_later()`, and `loop.call_at()`.

This class is a subclass of `Handle`.

when()

Return a scheduled callback time as `float` seconds.

The time is an absolute timestamp, using the same time reference as `loop.time()`.

Added in version 3.7.

Server Objects

Server objects are created by `loop.create_server()`, `loop.create_unix_server()`, `start_server()`, and `start_unix_server()` functions.

Do not instantiate the `Server` class directly.

class asyncio.Server

Server objects are asynchronous context managers. When used in an `async with` statement, it’s guaranteed that the Server object is closed and not accepting new connections when the `async with` statement is completed:

```
srv = await loop.create_server(...)

async with srv:
    # some code

# At this point, srv is closed and no longer accepts new
connections.
```

Changed in version 3.7: Server object is an asynchronous context manager since Python 3.7.

Changed in version 3.11: This class was exposed publicly as `asyncio.Server` in Python 3.9.11, 3.10.3 and 3.11.

close()

Stop serving: close listening sockets and set the **`sockets`** attribute to `None`.

The sockets that represent existing incoming client connections are left open.

The server is closed asynchronously; use the **`wait_closed()`** coroutine to wait until the server is closed (and no more connections are active).

close_clients()

Close all existing incoming client connections.

Calls **`close()`** on all associated transports.

`close()` should be called before **`close_clients()`** when closing the server to avoid races with new clients connecting.

Added in version 3.13.

abort_clients()

Close all existing incoming client connections immediately, without waiting for pending operations to complete.

Calls `abort()` on all associated transports.

`close()` should be called before `abort_clients()` when closing the server to avoid races with new clients connecting.

Added in version 3.13.

get_loop()

Return the event loop associated with the server object.

Added in version 3.7.

async start_serving()

Start accepting connections.

This method is idempotent, so it can be called when the server is already serving.

The *start_serving* keyword-only parameter to `loop.create_server()` and `asyncio.start_server()` allows creating a Server object that is not accepting connections initially. In this case `Server.start_serving()`, or `Server.serve_forever()` can be used to make the Server start accepting connections.

Added in version 3.7.

async serve_forever()

Start accepting connections until the coroutine is cancelled. Cancellation of `serve_forever` task causes the server to be closed.

This method can be called if the server is already accepting connections. Only one `serve_forever` task can exist per one *Server* object.

Example:

```
async def client_connected(reader, writer):
    # Communicate with the client with
    # reader/writer streams.  For example:
    await reader.readline()

async def main(host, port):
    srv = await asyncio.start_server(
        client_connected, host, port)
    await srv.serve_forever()

asyncio.run(main('127.0.0.1', 0))
```

Added in version 3.7.

is_serving()

Return `True` if the server is accepting new connections.

Added in version 3.7.

async wait_closed()

Wait until the `close()` method completes and all active connections have finished.

sockets

List of socket-like objects, `asyncio.trsock.TransportSocket`, which the server is listening on.

Changed in version 3.7: Prior to Python 3.7 `Server.sockets` used to return an internal list of server sockets directly. In 3.7 a copy of that list is returned.

Event Loop Implementations

asyncio ships with two different event loop implementations: `SelectorEventLoop` and `ProactorEventLoop`.

By default asyncio is configured to use `EventLoop`.

class asyncio.SelectorEventLoop

A subclass of `AbstractEventLoop` based on the `selectors` module.

Uses the most efficient *selector* available for the given platform. It is also possible to manually configure the exact selector implementation to be used:

```
import asyncio
import selectors

async def main():
   ...

loop_factory = lambda:
asyncio.SelectorEventLoop(selectors.SelectSelector())
asyncio.run(main(), loop_factory=loop_factory)
```

Availability: Unix, Windows.

class asyncio.ProactorEventLoop

A subclass of `AbstractEventLoop` for Windows that uses “I/O Completion Ports” (IOCP).

Availability: Windows.

See also

MSDN documentation on I/O Completion Ports [https://learn.microsoft.com/windows/win32/fileio/i-o-completion-ports].

class asyncio.EventLoop

An alias to the most efficient available subclass of `AbstractEventLoop` for the given platform.

It is an alias to `SelectorEventLoop` on Unix and `ProactorEventLoop` on Windows.

Added in version 3.13.

class asyncio.AbstractEventLoop

Abstract base class for asyncio-compliant event loops.

The Event Loop Methods section lists all methods that an alternative implementation of `AbstractEventLoop` should have defined.

Examples

Note that all examples in this section **purposefully** show how to use the low-level event loop APIs, such as `loop.run_forever()` and `loop.call_soon()`. Modern asyncio applications rarely need to be written this way; consider using the high-level functions like `asyncio.run()`.

Hello World with call_soon()

An example using the `loop.call_soon()` method to schedule a callback. The callback displays `"Hello World"` and then stops the event loop:

```
import asyncio

def hello_world(loop):
    """A callback to print 'Hello World' and stop the event
loop"""
    print('Hello World')
    loop.stop()

loop = asyncio.new_event_loop()

# Schedule a call to hello_world()
loop.call_soon(hello_world, loop)

# Blocking call interrupted by loop.stop()
try:
    loop.run_forever()
finally:
    loop.close()
```

See also

A similar Hello World example created with a coroutine and the `run()` function.

Display the current date with call_later()

An example of a callback displaying the current date every second. The callback uses the `loop.call_later()` method to reschedule itself after 5 seconds, and then stops the event loop:

```
import asyncio
import datetime

def display_date(end_time, loop):
    print(datetime.datetime.now())
    if (loop.time() + 1.0) < end_time:
        loop.call_later(1, display_date, end_time, loop)
    else:
        loop.stop()

loop = asyncio.new_event_loop()

# Schedule the first call to display_date()
end_time = loop.time() + 5.0
loop.call_soon(display_date, end_time, loop)

# Blocking call interrupted by loop.stop()
try:
    loop.run_forever()
finally:
    loop.close()
```

See also

A similar current date example created with a coroutine and the `run()` function.

Watch a file descriptor for read events

Wait until a file descriptor received some data using the `loop.add_reader()` method and then close the event loop:

```
import asyncio
from socket import socketpair

# Create a pair of connected file descriptors
rsock, wsock = socketpair()

loop = asyncio.new_event_loop()

def reader():
    data = rsock.recv(100)
    print("Received:", data.decode())

    # We are done: unregister the file descriptor
    loop.remove_reader(rsock)

    # Stop the event loop
    loop.stop()

# Register the file descriptor for read event
loop.add_reader(rsock, reader)

# Simulate the reception of data from the network
loop.call_soon(wsock.send, 'abc'.encode())

try:
    # Run the event loop
    loop.run_forever()
finally:
    # We are done. Close sockets and the event loop.
    rsock.close()
    wsock.close()
    loop.close()
```

See also

- A similar example using transports, protocols, and the **`loop.create_connection()`** method.
- Another similar example using the high-level **`asyncio.open_connection()`** function and streams.

Set signal handlers for SIGINT and SIGTERM

(This `signals` example only works on Unix.)

Register handlers for signals **`SIGINT`** and **`SIGTERM`** using the **`loop.add_signal_handler()`** method:

```
import asyncio
import functools
import os
import signal

def ask_exit(signame, loop):
    print("got signal %s: exit" % signame)
    loop.stop()
```

```
async def main():
    loop = asyncio.get_running_loop()

    for signame in {'SIGINT', 'SIGTERM'}:
        loop.add_signal_handler(
            getattr(signal, signame),
            functools.partial(ask_exit, signame, loop))

    await asyncio.sleep(3600)

print("Event loop running for 1 hour, press Ctrl+C to
interrupt.")
print(f"pid {os.getpid()}: send SIGINT or SIGTERM to exit.")

asyncio.run(main())
```

Futures

Source code: Lib/asyncio/futures.py [https://github.com/python/cpython/tree/3.14/Lib/asyncio/futures.py], Lib/asyncio/base_futures.py [https://github.com/python/cpython/tree/3.14/Lib/asyncio/base_futures.py]

Future objects are used to bridge **low-level callback-based code** with high-level async/await code.

Future Functions

asyncio.isfuture(*obj*)

Return `True` if *obj* is either of:

- an instance of **`asyncio.Future`**,
- an instance of **`asyncio.Task`**,
- a Future-like object with a `_asyncio_future_blocking` attribute.

Added in version 3.5.

asyncio.ensure_future(*obj*, *, *loop=None*)

Return:

- *obj* argument as is, if *obj* is a **`Future`**, a **`Task`**, or a Future-like object (**`isfuture()`** is used for the test.)
- a **`Task`** object wrapping *obj*, if *obj* is a coroutine (**`iscoroutine()`** is used for the test); in this case the coroutine will be scheduled by `ensure_future()`.
- a **`Task`** object that would await on *obj*, if *obj* is an awaitable (**`inspect.isawaitable()`** is used for the test.)

If *obj* is neither of the above a `TypeError` is raised.

Important

Save a reference to the result of this function, to avoid a task disappearing mid-execution.

See also the `create_task()` function which is the preferred way for creating new tasks or use `asyncio.TaskGroup` which keeps reference to the task internally.

Changed in version 3.5.1: The function accepts any awaitable object.

Deprecated since version 3.10: Deprecation warning is emitted if *obj* is not a Future-like object and *loop* is not specified and there is no running event loop.

asyncio.wrap_future(*future*, *, *loop=None*)

Wrap a `concurrent.futures.Future` object in a `asyncio.Future` object.

Deprecated since version 3.10: Deprecation warning is emitted if *future* is not a Future-like object and *loop* is not specified and there is no running event loop.

Future Object

class asyncio.Future(*, *loop=None*)

A Future represents an eventual result of an asynchronous operation. Not thread-safe.

Future is an awaitable object. Coroutines can await on Future objects until they either have a result or an exception set, or until they are cancelled. A Future can be awaited multiple times and the result is same.

Typically Futures are used to enable low-level callback-based code (e.g. in protocols implemented using asyncio transports) to interoperate with high-level async/await code.

The rule of thumb is to never expose Future objects in user-facing APIs, and the recommended way to create a Future object is to call `loop.create_future()`. This way alternative event loop implementations can inject their own optimized implementations of a Future object.

Changed in version 3.7: Added support for the `contextvars` module.

Deprecated since version 3.10: Deprecation warning is emitted if *loop* is not specified and there is no running event loop.

result()

Return the result of the Future.

If the Future is *done* and has a result set by the `set_result()` method, the result value is returned.

If the Future is *done* and has an exception set by the `set_exception()` method, this method raises the exception.

If the Future has been *cancelled*, this method raises a `CancelledError` exception.

If the Future’s result isn’t yet available, this method raises an `InvalidStateError` exception.

set_result(*result*)

Mark the Future as *done* and set its result.

Raises an **`InvalidStateError`** error if the Future is already *done*.

set_exception(*exception*)

Mark the Future as *done* and set an exception.

Raises an **`InvalidStateError`** error if the Future is already *done*.

done()

Return `True` if the Future is *done*.

A Future is *done* if it was *cancelled* or if it has a result or an exception set with **`set_result()`** or **`set_exception()`** calls.

cancelled()

Return `True` if the Future was *cancelled*.

The method is usually used to check if a Future is not *cancelled* before setting a result or an exception for it:

```
if not fut.cancelled():
    fut.set_result(42)
```

add_done_callback(*callback*, *, *context=None*)

Add a callback to be run when the Future is *done*.

The *callback* is called with the Future object as its only argument.

If the Future is already *done* when this method is called, the callback is scheduled with **`loop.call_soon()`**.

An optional keyword-only *context* argument allows specifying a custom `contextvars.Context` for the *callback* to run in. The current context is used when no *context* is provided.

`functools.partial()` can be used to pass parameters to the callback, e.g.:

```
# Call 'print("Future:", fut)' when "fut" is done.
fut.add_done_callback(
    functools.partial(print, "Future:"))
```

Changed in version 3.7: The *context* keyword-only parameter was added. See **PEP 567** [https://peps.python.org/pep-0567/] for more details.

remove_done_callback(*callback*)

Remove *callback* from the callbacks list.

Returns the number of callbacks removed, which is typically 1, unless a callback was added more than once.

cancel(*msg=None*)

Cancel the Future and schedule callbacks.

If the Future is already *done* or *cancelled*, return `False`. Otherwise, change the Future’s state to *cancelled*, schedule the callbacks, and return `True`.

Changed in version 3.9: Added the *msg* parameter.

exception()

Return the exception that was set on this Future.

The exception (or `None` if no exception was set) is returned only if the Future is *done*.

If the Future has been *cancelled*, this method raises a `CancelledError` exception.

If the Future isn't *done* yet, this method raises an `InvalidStateError` exception.

get_loop()

Return the event loop the Future object is bound to.

Added in version 3.7.

This example creates a Future object, creates and schedules an asynchronous Task to set result for the Future, and waits until the Future has a result:

```
async def set_after(fut, delay, value):
    # Sleep for *delay* seconds.
    await asyncio.sleep(delay)

    # Set *value* as a result of *fut* Future.
    fut.set_result(value)

async def main():
    # Get the current event loop.
    loop = asyncio.get_running_loop()

    # Create a new Future object.
    fut = loop.create_future()

    # Run "set_after()" coroutine in a parallel Task.
    # We are using the low-level "loop.create_task()" API here 
because
    # we already have a reference to the event loop at hand.
    # Otherwise we could have just used 
"asyncio.create_task()".
    loop.create_task(
        set_after(fut, 1, '... world'))

    print('hello ...')

    # Wait until *fut* has a result (1 second) and print it.
    print(await fut)

asyncio.run(main())
```

Important

The Future object was designed to mimic `concurrent.futures.Future`. Key differences include:

- unlike asyncio Futures, `concurrent.futures.Future` instances cannot be awaited.
- `asyncio.Future.result()` and `asyncio.Future.exception()` do not accept the *timeout* argument.
- `asyncio.Future.result()` and `asyncio.Future.exception()` raise an `InvalidStateError` exception when the Future is not *done*.
- Callbacks registered with `asyncio.Future.add_done_callback()` are not called immediately. They are scheduled with `loop.call_soon()` instead.
- asyncio Future is not compatible with the `concurrent.futures.wait()` and `concurrent.futures.as_completed()` functions.
- `asyncio.Future.cancel()` accepts an optional `msg` argument, but `concurrent.futures.Future.cancel()` does not.

Transports and Protocols

Preface

Transports and Protocols are used by the **low-level** event loop APIs such as `loop.create_connection()`. They use callback-based programming style and enable high-performance implementations of network or IPC protocols (e.g. HTTP).

Essentially, transports and protocols should only be used in libraries and frameworks and never in high-level asyncio applications.

This documentation page covers both Transports and Protocols.

Introduction

At the highest level, the transport is concerned with *how* bytes are transmitted, while the protocol determines *which* bytes to transmit (and to some extent when).

A different way of saying the same thing: a transport is an abstraction for a socket (or similar I/O endpoint) while a protocol is an abstraction for an application, from the transport’s point of view.

Yet another view is the transport and protocol interfaces together define an abstract interface for using network I/O and interprocess I/O.

There is always a 1:1 relationship between transport and protocol objects: the protocol calls transport methods to send data, while the transport calls protocol methods to pass it data that has been received.

Most of connection oriented event loop methods (such as **`loop.create_connection()`**) usually accept a *protocol_factory* argument used to create a *Protocol* object for an accepted connection, represented by a *Transport* object. Such methods usually return a tuple of `(transport, protocol)`.

Contents

This documentation page contains the following sections:

- The Transports section documents asyncio **`BaseTransport`**, **`ReadTransport`**, **`WriteTransport`**, **`Transport`**, **`DatagramTransport`**, and **`SubprocessTransport`** classes.
- The Protocols section documents asyncio **`BaseProtocol`**, **`Protocol`**, **`BufferedProtocol`**, **`DatagramProtocol`**, and **`SubprocessProtocol`** classes.
- The Examples section showcases how to work with transports, protocols, and low-level event loop APIs.

Transports

Source code: Lib/asyncio/transports.py [https://github.com/python/cpython/tree/3.14/Lib/asyncio/transports.py]

Transports are classes provided by **`asyncio`** in order to abstract various kinds of communication channels.

Transport objects are always instantiated by an asyncio event loop.

asyncio implements transports for TCP, UDP, SSL, and subprocess pipes. The methods available on a transport depend on the transport’s kind.

The transport classes are not thread safe.

Transports Hierarchy

class asyncio.BaseTransport

Base class for all transports. Contains methods that all asyncio transports share.

class asyncio.WriteTransport(*BaseTransport*)

A base transport for write-only connections.

Instances of the *WriteTransport* class are returned from the `loop.connect_write_pipe()` event loop method and are also used by subprocess-related methods like `loop.subprocess_exec()`.

class asyncio.ReadTransport(*BaseTransport*)

A base transport for read-only connections.

Instances of the *ReadTransport* class are returned from the `loop.connect_read_pipe()` event loop method and are also used by subprocess-related methods like `loop.subprocess_exec()`.

class asyncio.Transport(*WriteTransport*, *ReadTransport*)

Interface representing a bidirectional transport, such as a TCP connection.

The user does not instantiate a transport directly; they call a utility function, passing it a protocol factory and other information necessary to create the transport and protocol.

Instances of the *Transport* class are returned from or used by event loop methods like `loop.create_connection()`, `loop.create_unix_connection()`, `loop.create_server()`, `loop.sendfile()`, etc.

class asyncio.DatagramTransport(*BaseTransport*)

A transport for datagram (UDP) connections.

Instances of the *DatagramTransport* class are returned from the `loop.create_datagram_endpoint()` event loop method.

class asyncio.SubprocessTransport(*BaseTransport*)

An abstraction to represent a connection between a parent and its child OS process.

Instances of the *SubprocessTransport* class are returned from event loop methods `loop.subprocess_shell()` and `loop.subprocess_exec()`.

Base Transport

BaseTransport.close()

Close the transport.

If the transport has a buffer for outgoing data, buffered data will be flushed asynchronously. No more data will be received. After all buffered data is flushed, the protocol's `protocol.connection_lost()` method will be called with `None` as its argument. The transport should not be used once it is closed.

BaseTransport.is_closing()

Return `True` if the transport is closing or is closed.

BaseTransport.get_extra_info(*name*, *default=None*)

Return information about the transport or underlying resources it uses.

name is a string representing the piece of transport-specific information to get.

default is the value to return if the information is not available, or if the transport does not support querying it with the given third-party event loop implementation or on the current platform.

For example, the following code attempts to get the underlying socket object of the transport:

```
sock = transport.get_extra_info('socket')
if sock is not None:
    print(sock.getsockopt(...))
```

Categories of information that can be queried on some transports:

- socket:
    - `'peername'`: the remote address to which the socket is connected, result of **`socket.socket.getpeername()`** (`None` on error)
    - `'socket'`: **`socket.socket`** instance
    - `'sockname'`: the socket’s own address, result of **`socket.socket.getsockname()`**
- SSL socket:
    - `'compression'`: the compression algorithm being used as a string, or `None` if the connection isn’t compressed; result of **`ssl.SSLSocket.compression()`**
    - `'cipher'`: a three-value tuple containing the name of the cipher being used, the version of the SSL protocol that defines its use, and the number of secret bits being used; result of **`ssl.SSLSocket.cipher()`**

    - `'peercert'`: peer certificate; result of `ssl.SSLSocket.getpeercert()`
    - `'sslcontext'`: `ssl.SSLContext` instance
    - `'ssl_object'`: `ssl.SSLObject` or `ssl.SSLSocket` instance
- pipe:
    - `'pipe'`: pipe object
- subprocess:
    - `'subprocess'`: `subprocess.Popen` instance

BaseTransport.set_protocol(*protocol*)

Set a new protocol.

Switching protocol should only be done when both protocols are documented to support the switch.

BaseTransport.get_protocol()

Return the current protocol.

Read-only Transports

ReadTransport.is_reading()

Return `True` if the transport is receiving new data.

Added in version 3.7.

ReadTransport.pause_reading()

Pause the receiving end of the transport. No data will be passed to the protocol’s `protocol.data_received()` method until `resume_reading()` is called.

Changed in version 3.7: The method is idempotent, i.e. it can be called when the transport is already paused or closed.

ReadTransport.resume_reading()

Resume the receiving end. The protocol’s `protocol.data_received()` method will be called once again if some data is available for reading.

Changed in version 3.7: The method is idempotent, i.e. it can be called when the transport is already reading.

Write-only Transports

WriteTransport.abort()

Close the transport immediately, without waiting for pending operations to complete. Buffered data will be lost. No more data will be received. The protocol’s `protocol.connection_lost()` method will eventually be called with `None` as its argument.

WriteTransport.can_write_eof()

Return `True` if the transport supports `write_eof()`, `False` if not.

WriteTransport.get_write_buffer_size()

Return the current size of the output buffer used by the transport.

WriteTransport.get_write_buffer_limits()

Get the *high* and *low* watermarks for write flow control. Return a tuple `(low, high)` where *low* and *high* are positive number of bytes.

Use `set_write_buffer_limits()` to set the limits.

Added in version 3.4.2.

WriteTransport.set_write_buffer_limits(*high=None*, *low=None*)

Set the *high* and *low* watermarks for write flow control.

These two values (measured in number of bytes) control when the protocol's `protocol.pause_writing()` and `protocol.resume_writing()` methods are called. If specified, the low watermark must be less than or equal to the high watermark. Neither *high* nor *low* can be negative.

`pause_writing()` is called when the buffer size becomes greater than or equal to the *high* value. If writing has been paused, `resume_writing()` is called when the buffer size becomes less than or equal to the *low* value.

The defaults are implementation-specific. If only the high watermark is given, the low watermark defaults to an implementation-specific value less than or equal to the high watermark. Setting *high* to zero forces *low* to zero as well, and causes `pause_writing()` to be called whenever the buffer becomes non-empty. Setting *low* to zero causes `resume_writing()` to be called only once the buffer is empty. Use of zero for either limit is generally sub-optimal as it reduces opportunities for doing I/O and computation concurrently.

Use `get_write_buffer_limits()` to get the limits.

WriteTransport.write(*data*)

Write some *data* bytes to the transport.

This method does not block; it buffers the data and arranges for it to be sent out asynchronously.

WriteTransport.writelines(*list_of_data*)

Write a list (or any iterable) of data bytes to the transport. This is functionally equivalent to calling `write()`

on each element yielded by the iterable, but may be implemented more efficiently.

WriteTransport.write_eof()

Close the write end of the transport after flushing all buffered data. Data may still be received.

This method can raise `NotImplementedError` if the transport (e.g. SSL) doesn’t support half-closed connections.

Datagram Transports

DatagramTransport.sendto(*data*, *addr=None*)

Send the *data* bytes to the remote peer given by *addr* (a transport-dependent target address). If *addr* is `None`, the data is sent to the target address given on transport creation.

This method does not block; it buffers the data and arranges for it to be sent out asynchronously.

Changed in version 3.13: This method can be called with an empty bytes object to send a zero-length datagram. The buffer size calculation used for flow control is also updated to account for the datagram header.

DatagramTransport.abort()

Close the transport immediately, without waiting for pending operations to complete. Buffered data will be lost. No more data will be received. The protocol’s `protocol.connection_lost()` method will eventually be called with `None` as its argument.

Subprocess Transports

SubprocessTransport.get_pid()

Return the subprocess process id as an integer.

SubprocessTransport.get_pipe_transport(*fd*)

Return the transport for the communication pipe corresponding to the integer file descriptor *fd*:

- `0`: writable streaming transport of the standard input (*stdin*), or **`None`** if the subprocess was not created with `stdin=PIPE`
- `1`: readable streaming transport of the standard output (*stdout*), or **`None`** if the subprocess was not created with `stdout=PIPE`
- `2`: readable streaming transport of the standard error (*stderr*), or **`None`** if the subprocess was not created with `stderr=PIPE`
- other *fd*: **`None`**

SubprocessTransport.get_returncode()

Return the subprocess return code as an integer or **`None`** if it hasn't returned, which is similar to the **`subprocess.Popen.returncode`** attribute.

SubprocessTransport.kill()

Kill the subprocess.

On POSIX systems, the function sends SIGKILL to the subprocess. On Windows, this method is an alias for **`terminate()`**.

See also **`subprocess.Popen.kill()`**.

SubprocessTransport.send_signal(*signal*)

Send the *signal* number to the subprocess, as in **`subprocess.Popen.send_signal()`**.

SubprocessTransport.terminate()

Stop the subprocess.

On POSIX systems, this method sends `SIGTERM` to the subprocess. On Windows, the Windows API function `TerminateProcess()` is called to stop the subprocess.

See also `subprocess.Popen.terminate()`.

SubprocessTransport.close()

Kill the subprocess by calling the `kill()` method.

If the subprocess hasn't returned yet, and close transports of *stdin*, *stdout*, and *stderr* pipes.

Protocols

Source code: Lib/asyncio/protocols.py
[https://github.com/python/cpython/tree/3.14/Lib/asyncio/protocols.py]

---

asyncio provides a set of abstract base classes that should be used to implement network protocols. Those classes are meant to be used together with transports.

Subclasses of abstract base protocol classes may implement some or all methods. All these methods are callbacks: they are called by transports on certain events, for example when some data is received. A base protocol method should be called by the corresponding transport.

Base Protocols

class asyncio.BaseProtocol

Base protocol with methods that all protocols share.

class asyncio.Protocol(*BaseProtocol*)

The base class for implementing streaming protocols (TCP, Unix sockets, etc).

class asyncio.BufferedProtocol(*BaseProtocol*)

A base class for implementing streaming protocols with manual control of the receive buffer.

class asyncio.DatagramProtocol(*BaseProtocol*)

The base class for implementing datagram (UDP) protocols.

class asyncio.SubprocessProtocol(*BaseProtocol*)

The base class for implementing protocols communicating with child processes (unidirectional pipes).

Base Protocol

All asyncio protocols can implement Base Protocol callbacks.

Connection Callbacks

Connection callbacks are called on all protocols, exactly once per a successful connection. All other protocol callbacks can only be called between those two methods.

BaseProtocol.connection_made(*transport*)

Called when a connection is made.

The *transport* argument is the transport representing the connection. The protocol is responsible for storing the reference to its transport.

BaseProtocol.connection_lost(*exc*)

Called when the connection is lost or closed.

The argument is either an exception object or `None`. The latter means a regular EOF is received, or the connection was aborted or closed by this side of the connection.

Flow Control Callbacks

Flow control callbacks can be called by transports to pause or resume writing performed by the protocol.

See the documentation of the `set_write_buffer_limits()` method for more details.

BaseProtocol.pause_writing()

Called when the transport’s buffer goes over the high watermark.

BaseProtocol.resume_writing()

Called when the transport’s buffer drains below the low watermark.

If the buffer size equals the high watermark, `pause_writing()` is not called: the buffer size must go strictly over.

Conversely, `resume_writing()` is called when the buffer size is equal or lower than the low watermark. These end conditions are important to ensure that things go as expected when either mark is zero.

Streaming Protocols

Event methods, such as `loop.create_server()`, `loop.create_unix_server()`, `loop.create_connection()`, `loop.create_unix_connection()`, `loop.connect_accepted_socket()`, `loop.connect_read_pipe()`, and `loop.connect_write_pipe()` accept factories that return streaming protocols.

Protocol.data_received(*data*)

Called when some data is received. *data* is a non-empty bytes object containing the incoming data.

Whether the data is buffered, chunked or reassembled depends on the transport. In general, you shouldn’t rely on specific semantics and instead make your parsing generic and flexible. However, data is always received in the correct order.

The method can be called an arbitrary number of times while a connection is open.

However, `protocol.eof_received()` is called at most once. Once `eof_received()` is called, `data_received()` is not called anymore.

Protocol.eof_received()

Called when the other end signals it won’t send any more data (for example by calling `transport.write_eof()`, if the other end also uses asyncio).

This method may return a false value (including `None`), in which case the transport will close itself. Conversely, if this method returns a true value, the protocol used determines whether to close the transport. Since the default implementation returns `None`, it implicitly closes the connection.

Some transports, including SSL, don’t support half-closed connections, in which case returning true from this method will result in the connection being closed.

State machine:

```
start -> connection_made
    [-> data_received]*
    [-> eof_received]?
-> connection_lost -> end
```

Buffered Streaming Protocols

Added in version 3.7.

Buffered Protocols can be used with any event loop method that supports Streaming Protocols.

`BufferedProtocol` implementations allow explicit manual allocation and control of the receive buffer. Event loops can then use the buffer provided by the protocol to avoid unnecessary data copies. This can result in noticeable performance improvement for protocols that receive big amounts of data. Sophisticated protocol implementations can significantly reduce the number of buffer allocations.

The following callbacks are called on **`BufferedProtocol`** instances:

BufferedProtocol.get_buffer(*sizehint*)

Called to allocate a new receive buffer.

sizehint is the recommended minimum size for the returned buffer. It is acceptable to return smaller or larger buffers than what *sizehint* suggests. When set to -1, the buffer size can be arbitrary. It is an error to return a buffer with a zero size.

`get_buffer()` must return an object implementing the buffer protocol.

BufferedProtocol.buffer_updated(*nbytes*)

Called when the buffer was updated with the received data.

nbytes is the total number of bytes that were written to the buffer.

BufferedProtocol.eof_received()

See the documentation of the `protocol.eof_received()` method.

`get_buffer()` can be called an arbitrary number of times during a connection. However, `protocol.eof_received()` is called at most once and, if called, `get_buffer()` and `buffer_updated()` won't be called after it.

State machine:

```
start -> connection_made
    [-> get_buffer
        [-> buffer_updated]?
    ]*
    [-> eof_received]?
-> connection_lost -> end
```

Datagram Protocols

Datagram Protocol instances should be constructed by protocol factories passed to the `loop.create_datagram_endpoint()` method.

DatagramProtocol.datagram_received(*data*, *addr*)

Called when a datagram is received. *data* is a bytes object containing the incoming data. *addr* is the address of the peer sending the data; the exact format depends on the transport.

DatagramProtocol.error_received(*exc*)

Called when a previous send or receive operation raises an `OSError`. *exc* is the `OSError` instance.

This method is called in rare conditions, when the transport (e.g. UDP) detects that a datagram could not be delivered to its recipient. In many conditions though, undeliverable datagrams will be silently dropped.

Note

On BSD systems (macOS, FreeBSD, etc.) flow control is not supported for datagram protocols, because there is no reliable way to detect send failures caused by writing too many packets.

The socket always appears ‘ready’ and excess packets are dropped. An `OSError` with `errno` set to `errno.ENOBUFS` may or may not be raised; if it is raised, it will be reported to `DatagramProtocol.error_received()` but otherwise ignored.

Subprocess Protocols

Subprocess Protocol instances should be constructed by protocol factories passed to the `loop.subprocess_exec()` and `loop.subprocess_shell()` methods.

SubprocessProtocol.pipe_data_received(*fd*, *data*)

Called when the child process writes data into its stdout or stderr pipe.

fd is the integer file descriptor of the pipe.

data is a non-empty bytes object containing the received data.

SubprocessProtocol.pipe_connection_lost(*fd*, *exc*)

Called when one of the pipes communicating with the child process is closed.

fd is the integer file descriptor that was closed.

SubprocessProtocol.process_exited()

Called when the child process has exited.

It can be called before `pipe_data_received()` and `pipe_connection_lost()` methods.

Examples

TCP Echo Server

Create a TCP echo server using the `loop.create_server()` method, send back received data, and close the connection:

```
import asyncio


class EchoServerProtocol(asyncio.Protocol):
    def connection_made(self, transport):
        peername = transport.get_extra_info('peername')
        print('Connection from {}'.format(peername))
        self.transport = transport

    def data_received(self, data):
        message = data.decode()
        print('Data received: {!r}'.format(message))

        print('Send: {!r}'.format(message))
        self.transport.write(data)

        print('Close the client socket')
        self.transport.close()
```

```
async def main():
    # Get a reference to the event loop as we plan to use
    # low-level APIs.
    loop = asyncio.get_running_loop()

    server = await loop.create_server(
        EchoServerProtocol,
        '127.0.0.1', 8888)

    async with server:
        await server.serve_forever()


asyncio.run(main())
```

See also

The TCP echo server using streams example uses the high-level `asyncio.start_server()` function.

TCP Echo Client

A TCP echo client using the `loop.create_connection()` method, sends data, and waits until the connection is closed:

```
import asyncio


class EchoClientProtocol(asyncio.Protocol):
    def __init__(self, message, on_con_lost):
        self.message = message
        self.on_con_lost = on_con_lost

    def connection_made(self, transport):
        transport.write(self.message.encode())
        print('Data sent: {!r}'.format(self.message))

    def data_received(self, data):
        print('Data received: {!r}'.format(data.decode()))
    def connection_lost(self, exc):
        print('The server closed the connection')
        self.on_con_lost.set_result(True)


async def main():
    # Get a reference to the event loop as we plan to use
    # low-level APIs.
    loop = asyncio.get_running_loop()

    on_con_lost = loop.create_future()
    message = 'Hello World!'

    transport, protocol = await loop.create_connection(
        lambda: EchoClientProtocol(message, on_con_lost),
        '127.0.0.1', 8888)

    # Wait until the protocol signals that the connection
    # is lost and close the transport.
    try:
        await on_con_lost
    finally:
        transport.close()


asyncio.run(main())
```

See also

The TCP echo client using streams example uses the high-level **`asyncio.open_connection()`** function.

UDP Echo Server

A UDP echo server, using the **`loop.create_datagram_endpoint()`** method, sends back received data:

```
import asyncio
```

```
class EchoServerProtocol:
    def connection_made(self, transport):
        self.transport = transport

    def datagram_received(self, data, addr):
        message = data.decode()
        print('Received %r from %s' % (message, addr))
        print('Send %r to %s' % (message, addr))
        self.transport.sendto(data, addr)


async def main():
    print("Starting UDP server")

    # Get a reference to the event loop as we plan to use
    # low-level APIs.
    loop = asyncio.get_running_loop()

    # One protocol instance will be created to serve all
    # client requests.
    transport, protocol = await loop.create_datagram_endpoint(
        EchoServerProtocol,
        local_addr=('127.0.0.1', 9999))

    try:
        await asyncio.sleep(3600)  # Serve for 1 hour.
    finally:
        transport.close()


asyncio.run(main())
```

UDP Echo Client

A UDP echo client, using the **`loop.create_datagram_endpoint()`** method, sends data and closes the transport when it receives the answer:

```
import asyncio


class EchoClientProtocol:
    def __init__(self, message, on_con_lost):
        self.message = message
        self.on_con_lost = on_con_lost
        self.transport = None

    def connection_made(self, transport):
        self.transport = transport
        print('Send:', self.message)
        self.transport.sendto(self.message.encode())

    def datagram_received(self, data, addr):
        print("Received:", data.decode())

        print("Close the socket")
        self.transport.close()

    def error_received(self, exc):
        print('Error received:', exc)

    def connection_lost(self, exc):
        print("Connection closed")
        self.on_con_lost.set_result(True)


async def main():
    # Get a reference to the event loop as we plan to use
    # low-level APIs.
    loop = asyncio.get_running_loop()

    on_con_lost = loop.create_future()
    message = "Hello World!"

    transport, protocol = await loop.create_datagram_endpoint(
        lambda: EchoClientProtocol(message, on_con_lost),
        remote_addr=('127.0.0.1', 9999))

    try:
        await on_con_lost
    finally:
        transport.close()


asyncio.run(main())
```

Connecting Existing Sockets

Wait until a socket receives data using the `loop.create_connection()` method with a protocol:

```
import asyncio
import socket


class MyProtocol(asyncio.Protocol):

    def __init__(self, on_con_lost):
        self.transport = None
        self.on_con_lost = on_con_lost

    def connection_made(self, transport):
        self.transport = transport

    def data_received(self, data):
        print("Received:", data.decode())

        # We are done: close the transport;
        # connection_lost() will be called automatically.
        self.transport.close()

    def connection_lost(self, exc):
        # The socket has been closed
        self.on_con_lost.set_result(True)


async def main():
    # Get a reference to the event loop as we plan to use
    # low-level APIs.
    loop = asyncio.get_running_loop()
    on_con_lost = loop.create_future()

    # Create a pair of connected sockets
    rsock, wsock = socket.socketpair()

    # Register the socket to wait for data.
    transport, protocol = await loop.create_connection(
        lambda: MyProtocol(on_con_lost), sock=rsock)
    # Simulate the reception of data from the network.
    loop.call_soon(wsock.send, 'abc'.encode())

    try:
        await protocol.on_con_lost
    finally:
        transport.close()
        wsock.close()

asyncio.run(main())
```

See also

The watch a file descriptor for read events example uses the low-level `loop.add_reader()` method to register an FD.

The register an open socket to wait for data using streams example uses high-level streams created by the `open_connection()` function in a coroutine.

loop.subprocess_exec() and SubprocessProtocol

An example of a subprocess protocol used to get the output of a subprocess and to wait for the subprocess exit.

The subprocess is created by the `loop.subprocess_exec()` method:

```
import asyncio
import sys

class DateProtocol(asyncio.SubprocessProtocol):
    def __init__(self, exit_future):
        self.exit_future = exit_future
        self.output = bytearray()
        self.pipe_closed = False
        self.exited = False
    def pipe_connection_lost(self, fd, exc):
        self.pipe_closed = True
        self.check_for_exit()

    def pipe_data_received(self, fd, data):
        self.output.extend(data)

    def process_exited(self):
        self.exited = True
        # process_exited() method can be called before
        # pipe_connection_lost() method: wait until both
methods are
        # called.
        self.check_for_exit()

    def check_for_exit(self):
        if self.pipe_closed and self.exited:
            self.exit_future.set_result(True)

async def get_date():
    # Get a reference to the event loop as we plan to use
    # low-level APIs.
    loop = asyncio.get_running_loop()

    code = 'import datetime; print(datetime.datetime.now())'
    exit_future = asyncio.Future(loop=loop)

    # Create the subprocess controlled by DateProtocol;
    # redirect the standard output into a pipe.
    transport, protocol = await loop.subprocess_exec(
        lambda: DateProtocol(exit_future),
        sys.executable, '-c', code,
        stdin=None, stderr=None)

    # Wait for the subprocess exit using the process_exited()
    # method of the protocol.
    await exit_future

    # Close the stdout pipe.
    transport.close()

    # Read the output which was collected by the
    # pipe_data_received() method of the protocol.
    data = bytes(protocol.output)
    return data.decode('ascii').rstrip()

date = asyncio.run(get_date())
print(f"Current date: {date}")
```

See also the same example written using high-level APIs.

Policies

Warning

Policies are deprecated and will be removed in Python 3.16. Users are encouraged to use the `asyncio.run()` function or the `asyncio.Runner` with *loop_factory* to use the desired loop implementation.

An event loop policy is a global object used to get and set the current event loop, as well as create new event loops. The default policy can be replaced with built-in alternatives to use different event loop implementations, or substituted by a custom policy that can override these behaviors.

The policy object gets and sets a separate event loop per *context*. This is per-thread by default, though custom policies could define *context* differently.

Custom event loop policies can control the behavior of `get_event_loop()`, `set_event_loop()`, and `new_event_loop()`.

Policy objects should implement the APIs defined in the `AbstractEventLoopPolicy` abstract base class.

Getting and Setting the Policy

The following functions can be used to get and set the policy for the current process:

asyncio.get_event_loop_policy()

Return the current process-wide policy.

Deprecated since version 3.14: The `get_event_loop_policy()` function is deprecated and will be removed in Python 3.16.

asyncio.set_event_loop_policy(*policy*)

Set the current process-wide policy to *policy*.

If *policy* is set to `None`, the default policy is restored.

Deprecated since version 3.14: The `set_event_loop_policy()` function is deprecated and will be removed in Python 3.16.

Policy Objects

The abstract event loop policy base class is defined as follows:

class asyncio.AbstractEventLoopPolicy

An abstract base class for asyncio policies.

get_event_loop()

Get the event loop for the current context.

Return an event loop object implementing the `AbstractEventLoop` interface.

This method should never return `None`.

Changed in version 3.6.

set_event_loop(*loop*)

Set the event loop for the current context to *loop*.

new_event_loop()

Create and return a new event loop object.

This method should never return `None`.

Deprecated since version 3.14: The `AbstractEventLoopPolicy` class is deprecated and will be removed in Python 3.16.

asyncio ships with the following built-in policies:

class asyncio.DefaultEventLoopPolicy

The default asyncio policy. Uses `SelectorEventLoop` on Unix and `ProactorEventLoop` on Windows.

There is no need to install the default policy manually. asyncio is configured to use the default policy automatically.

Changed in version 3.8: On Windows, `ProactorEventLoop` is now used by default.

Changed in version 3.14: The `get_event_loop()` method of the default asyncio policy now raises a `RuntimeError` if there is no set event loop.

Deprecated since version 3.14: The `DefaultEventLoopPolicy` class is deprecated and will be removed in Python 3.16.

class asyncio.WindowsSelectorEventLoopPolicy

An alternative event loop policy that uses the `SelectorEventLoop` event loop implementation.

Availability: Windows.

Deprecated since version 3.14: The `WindowsSelectorEventLoopPolicy` class is deprecated and will be removed in Python 3.16.

class asyncio.WindowsProactorEventLoopPolicy

An alternative event loop policy that uses the `ProactorEventLoop` event loop implementation.

Availability: Windows.

Deprecated since version 3.14: The `WindowsProactorEventLoopPolicy` class is deprecated and will be removed in Python 3.16.

Custom Policies

To implement a new event loop policy, it is recommended to subclass `DefaultEventLoopPolicy` and override the methods for which custom behavior is wanted, e.g.:

```
class MyEventLoopPolicy(asyncio.DefaultEventLoopPolicy):

    def get_event_loop(self):
        """Get the event loop.

        This may be None or an instance of EventLoop.
        """
        loop = super().get_event_loop()
        # Do something with loop ...
        return loop

asyncio.set_event_loop_policy(MyEventLoopPolicy())
```

Platform Support

The `asyncio` module is designed to be portable, but some platforms have subtle differences and limitations due to the platforms’ underlying architecture and capabilities.

All Platforms

- `loop.add_reader()` and `loop.add_writer()` cannot be used to monitor file I/O.

Windows

Source code: Lib/asyncio/proactor_events.py [https://github.com/python/cpython/tree/3.14/Lib/asyncio/proactor_events.py], Lib/asyncio/windows_events.py [https://github.com/python/cpython/tree/3.14/Lib/asyncio/windows_events.py], Lib/asyncio/windows_utils.py [https://github.com/python/cpython/tree/3.14/Lib/asyncio/windows_utils.py]

Changed in version 3.8: On Windows, `ProactorEventLoop` is now the default event loop.

All event loops on Windows do not support the following methods:

- `loop.create_unix_connection()` and `loop.create_unix_server()` are not supported. The `socket.AF_UNIX` socket family is specific to Unix.
- `loop.add_signal_handler()` and `loop.remove_signal_handler()` are not supported.

`SelectorEventLoop` has the following limitations:

- `SelectSelector` is used to wait on socket events: it supports sockets and is limited to 512 sockets.
- `loop.add_reader()` and `loop.add_writer()` only accept socket handles (e.g. pipe file descriptors are not supported).
- Pipes are not supported, so the `loop.connect_read_pipe()` and `loop.connect_write_pipe()` methods are not implemented.
- Subprocesses are not supported, i.e. `loop.subprocess_exec()` and `loop.subprocess_shell()` methods are not implemented.

`ProactorEventLoop` has the following limitations:

- The `loop.add_reader()` and `loop.add_writer()` methods are not supported.

The resolution of the monotonic clock on Windows is usually around 15.6 milliseconds. The best resolution is 0.5 milliseconds. The resolution depends on the hardware (availability of HPET [https://en.wikipedia.org/wiki/High_Precision_Event_Timer]) and on the Windows configuration.

Subprocess Support on Windows

On Windows, the default event loop `ProactorEventLoop` supports subprocesses, whereas `SelectorEventLoop` does not.

macOS

Modern macOS versions are fully supported.

macOS <= 10.8

On macOS 10.6, 10.7 and 10.8, the default event loop uses `selectors.KqueueSelector`, which does not support character devices on these versions. The `SelectorEventLoop` can be manually configured to use `SelectSelector` or `PollSelector` to support character devices on these older versions of macOS. Example:

```
import asyncio
import selectors

selector = selectors.SelectSelector()
loop = asyncio.SelectorEventLoop(selector)
asyncio.set_event_loop(loop)
```

Extending

The main direction for `asyncio` extending is writing custom *event loop* classes. Asyncio has helpers that could be used to simplify this task.

Note

Third-parties should reuse existing asyncio code with caution, a new Python version is free to break backward compatibility in *internal* part of API.

Writing a Custom Event Loop

`asyncio.AbstractEventLoop` declares very many methods. Implementing all them from scratch is a tedious job.

A loop can get many common methods implementation for free by inheriting from **`asyncio.BaseEventLoop`**.

In turn, the successor should implement a bunch of *private* methods declared but not implemented in **`asyncio.BaseEventLoop`**.

For example, `loop.create_connection()` checks arguments, resolves DNS addresses, and calls `loop._make_socket_transport()` that should be implemented by inherited class. The `_make_socket_transport()` method is not documented and is considered as an *internal* API.

Future and Task private constructors

`asyncio.Future` and `asyncio.Task` should be never created directly, please use corresponding `loop.create_future()` and `loop.create_task()`, or `asyncio.create_task()` factories instead.

However, third-party *event loops* may *reuse* built-in future and task implementations for the sake of getting a complex and highly optimized code for free.

For this purpose the following, *private* constructors are listed:

Future.__init__(*, *loop=None*)

Create a built-in future instance.

loop is an optional event loop instance.

Task.__init__(*coro*, *, *loop=None*, *name=None*, *context=None*)

Create a built-in task instance.

loop is an optional event loop instance. The rest of arguments are described in `loop.create_task()` description.

Changed in version 3.11: context argument is added.

Task lifetime support

A third party task implementation should call the following functions to keep a task visible by `asyncio.all_tasks()` and `asyncio.current_task()`:

asyncio._register_task(*task*)

Register a new *task* as managed by *asyncio*.

Call the function from a task constructor.

asyncio._unregister_task(*task*)

Unregister a *task* from *asyncio* internal structures.

The function should be called when a task is about to finish.

asyncio._enter_task(*loop*, *task*)

Switch the current task to the *task* argument.

Call the function just before executing a portion of embedded *coroutine* (**`coroutine.send()`** or **`coroutine.throw()`**).

asyncio._leave_task(*loop*, *task*)

Switch the current task back from *task* to `None`.

Call the function just after **`coroutine.send()`** or **`coroutine.throw()`** execution.

High-level API Index

This page lists all high-level async/await enabled asyncio APIs.

Tasks

Utilities to run asyncio programs, create Tasks, and await on multiple things with timeouts.

`run()`	Create event loop, run a coroutine, close the loop.
`Runner`	A context manager that simplifies multiple async function calls.
`Task`	Task object.
`TaskGroup`	A context manager that holds a group of tasks. Provides a convenient and reliable way to wait for all tasks in the group to finish.
`create_task()`	Start an asyncio Task, then returns it.

`current_task()`	Return the current Task.
`all_tasks()`	Return all tasks that are not yet finished for an event loop.
`await sleep()`	Sleep for a number of seconds.
`await gather()`	Schedule and wait for things concurrently.
`await wait_for()`	Run with a timeout.
`await shield()`	Shield from cancellation.
`await wait()`	Monitor for completion.
`timeout()`	Run with a timeout. Useful in cases when `wait_for` is not suitable.
`to_thread()`	Asynchronously run a function in a separate OS thread.

`run_coroutine_threadsafe()`	Schedule a coroutine from another OS thread.
`for in as_completed()`	Monitor for completion with a `for` loop.

Examples

- Using asyncio.gather() to run things in parallel.
- Using asyncio.wait_for() to enforce a timeout.
- Cancellation.
- Using asyncio.sleep().
- See also the main Tasks documentation page.

Queues

Queues should be used to distribute work amongst multiple asyncio Tasks, implement connection pools, and pub/sub patterns.

`Queue`	A FIFO queue.
`PriorityQueue`	A priority queue.
`LifoQueue`	A LIFO queue.

Examples

- Using asyncio.Queue to distribute workload between several Tasks.
- See also the Queues documentation page.

Subprocesses

Utilities to spawn subprocesses and run shell commands.

`await create_subprocess_exec()`	Create a subprocess.
`await create_subprocess_shell()`	Run a shell command.

Examples

- Executing a shell command.
- See also the subprocess APIs documentation.

Streams

High-level APIs to work with network IO.

`await open_connection()`	Establish a TCP connection.
`await open_unix_connection()`	Establish a Unix socket connection.
`await start_server()`	Start a TCP server.

await `start_unix_server()`	Start a Unix socket server.
`StreamReader`	High-level async/await object to receive network data.
`StreamWriter`	High-level async/await object to send network data.

Examples

- Example TCP client.
- See also the streams APIs documentation.

Synchronization

Threading-like synchronization primitives that can be used in Tasks.

`Lock`	A mutex lock.
`Event`	An event object.
`Condition`	A condition object.
`Semaphore`	A semaphore.
`BoundedSemaphore`	A bounded semaphore.

`Barrier`	A barrier object.

Examples

- Using asyncio.Event.
- Using asyncio.Barrier.
- See also the documentation of asyncio synchronization primitives.

Exceptions

`asyncio.CancelledError`	Raised when a Task is cancelled. See also **`Task.cancel()`**.
`asyncio.BrokenBarrierError`	Raised when a Barrier is broken. See also **`Barrier.wait()`**.

Examples

- Handling CancelledError to run code on cancellation request.
- See also the full list of asyncio-specific exceptions.

Low-level API Index

This page lists all low-level asyncio APIs.

Obtaining the Event Loop

`asyncio.get_running_loop()`	The **preferred** function to get the running event loop.
`asyncio.get_event_loop()`	Get an event loop instance (running or current via the current policy).
`asyncio.set_event_loop()`	Set the event loop as current via the current policy.
`asyncio.new_event_loop()`	Create a new event loop.

Examples

- Using asyncio.get_running_loop().

Event Loop Methods

See also the main documentation section about the Event Loop Methods.

Lifecycle

`loop.run_until_complete()`	Run a Future/Task/awaitable until complete.
`loop.run_forever()`	Run the event loop forever.
`loop.stop()`	Stop the event loop.
`loop.close()`	Close the event loop.
`loop.is_running()`	Return `True` if the event loop is running.
`loop.is_closed()`	Return `True` if the event loop is closed.
`await loop.shutdown_asyncgens()`	Close asynchronous generators.

Debugging

`loop.set_debug()`	Enable or disable the debug mode.
`loop.get_debug()`	Get the current debug mode.

Scheduling Callbacks

`loop.call_soon()`	Invoke a callback soon.
`loop.call_soon_threadsafe()`	A thread-safe variant of `loop.call_soon()`.
`loop.call_later()`	Invoke a callback *after* the given time.
`loop.call_at()`	Invoke a callback *at* the given time.

Thread/Interpreter/Process Pool

`await loop.run_in_executor()`	Run a CPU-bound or other blocking function in a `concurrent.futures` executor.
`loop.set_default_executor()`	Set the default executor for `loop.run_in_executor()`.

Tasks and Futures

`loop.create_future()`	Create a `Future` object.
`loop.create_task()`	Schedule coroutine as a `Task`.

`loop.set_task_factory()`	Set a factory used by `loop.create_task()` to create `Tasks`.
`loop.get_task_factory()`	Get the factory `loop.create_task()` uses to create `Tasks`.

DNS

`await loop.getaddrinfo()`	Asynchronous version of `socket.getaddrinfo()`.
`await loop.getnameinfo()`	Asynchronous version of `socket.getnameinfo()`.

Networking and IPC

`await loop.create_connection()`	Open a TCP connection.
`await loop.create_server()`	Create a TCP server.
`await loop.create_unix_connection()`	Open a Unix socket connection.
`await loop.create_unix_server()`	Create a Unix socket server.

`await loop.connect_accepted_socket()`	Wrap a `socket` into a `(transport, protocol)` pair.
`await loop.create_datagram_endpoint()`	Open a datagram (UDP) connection.
`await loop.sendfile()`	Send a file over a transport.
`await loop.start_tls()`	Upgrade an existing connection to TLS.
`await loop.connect_read_pipe()`	Wrap a read end of a pipe into a `(transport, protocol)` pair.
`await loop.connect_write_pipe()`	Wrap a write end of a pipe into a `(transport, protocol)` pair.

Sockets

`await loop.sock_recv()`	Receive data from the `socket`.
`await loop.sock_recv_into()`	Receive data from the `socket` into a buffer.

`await` `loop.sock_recvfrom()`	Receive a datagram from the `socket`.
`await` `loop.sock_recvfrom_into()`	Receive a datagram from the `socket` into a buffer.
`await` `loop.sock_sendall()`	Send data to the `socket`.
`await` `loop.sock_sendto()`	Send a datagram via the `socket` to the given address.
`await` `loop.sock_connect()`	Connect the `socket`.
`await` `loop.sock_accept()`	Accept a `socket` connection.
`await` `loop.sock_sendfile()`	Send a file over the `socket`.
`loop.add_reader()`	Start watching a file descriptor for read availability.
`loop.remove_reader()`	Stop watching a file descriptor for read availability.
`loop.add_writer()`	Start watching a file descriptor for write

	availability.
`loop.remove_writer()`	Stop watching a file descriptor for write availability.

Unix Signals

`loop.add_signal_handler()`	Add a handler for a `signal`.
`loop.remove_signal_handler()`	Remove a handler for a `signal`.

Subprocesses

`loop.subprocess_exec()`	Spawn a subprocess.
`loop.subprocess_shell()`	Spawn a subprocess from a shell command.

Error Handling

`loop.call_exception_handler()`	Call the exception handler.
`loop.set_exception_handler()`	Set a new exception handler.

`loop.get_exception_handler()`	Get the current exception handler.
`loop.default_exception_handler()`	The default exception handler implementation.

Examples

- Using asyncio.new_event_loop() and loop.run_forever().
- Using loop.call_later().
- Using `loop.create_connection()` to implement an echo-client.
- Using `loop.create_connection()` to connect a socket.
- Using add_reader() to watch an FD for read events.
- Using loop.add_signal_handler().
- Using loop.subprocess_exec().

Transports

All transports implement the following methods:

`transport.close()`	Close the transport.
`transport.is_closing()`	Return `True` if the transport is closing or is closed.
`transport.get_extra_info()`	Request for information about the transport.

`transport.set_protocol()`	Set a new protocol.
`transport.get_protocol()`	Return the current protocol.

Transports that can receive data (TCP and Unix connections, pipes, etc). Returned from methods like `loop.create_connection()`, `loop.create_unix_connection()`, `loop.connect_read_pipe()`, etc:

Read Transports

`transport.is_reading()`	Return `True` if the transport is receiving.
`transport.pause_reading()`	Pause receiving.
`transport.resume_reading()`	Resume receiving.

Transports that can Send data (TCP and Unix connections, pipes, etc). Returned from methods like `loop.create_connection()`, `loop.create_unix_connection()`, `loop.connect_write_pipe()`, etc:

Write Transports

`transport.write()`	Write data to the transport.

`transport.writelines()`	Write buffers to the transport.
`transport.can_write_eof()`	Return `True` if the transport supports sending EOF.
`transport.write_eof()`	Close and send EOF after flushing buffered data.
`transport.abort()`	Close the transport immediately.
`transport.get_write_buffer_size()`	Return the current size of the output buffer.
`transport.get_write_buffer_limits()`	Return high and low water marks for write flow control.
`transport.set_write_buffer_limits()`	Set new high and low water marks for write flow control.

Transports returned by `loop.create_datagram_endpoint()`:

Datagram Transports

`transport.sendto()`	Send data to the remote peer.
`transport.abort()`	Close the transport immediately.

Low-level transport abstraction over subprocesses. Returned by `loop.subprocess_exec()` and `loop.subprocess_shell()`:

Subprocess Transports

`transport.get_pid()`	Return the subprocess process id.
`transport.get_pipe_transport()`	Return the transport for the requested communication pipe (*stdin*, *stdout*, or *stderr*).
`transport.get_returncode()`	Return the subprocess return code.
`transport.kill()`	Kill the subprocess.
`transport.send_signal()`	Send a signal to the subprocess.
`transport.terminate()`	Stop the subprocess.

`transport.close()`	Kill the subprocess and close all pipes.

Protocols

Protocol classes can implement the following **callback methods**:

`callback` `connection_made()`	Called when a connection is made.
`callback` `connection_lost()`	Called when the connection is lost or closed.
`callback` `pause_writing()`	Called when the transport’s buffer goes over the high water mark.
`callback` `resume_writing()`	Called when the transport’s buffer drains below the low water mark.

Streaming Protocols (TCP, Unix Sockets, Pipes)

`callback` `data_received()`	Called when some data is received.

callback `eof_received()`	Called when an EOF is received.

Buffered Streaming Protocols

callback `get_buffer()`	Called to allocate a new receive buffer.
callback `buffer_updated()`	Called when the buffer was updated with the received data.
callback `eof_received()`	Called when an EOF is received.

Datagram Protocols

callback `datagram_received()`	Called when a datagram is received.
callback `error_received()`	Called when a previous send or receive operation raises an `OSError`.

Subprocess Protocols

`callback` **`pipe_data_received()`**	Called when the child process writes data into its *stdout* or *stderr* pipe.
`callback` **`pipe_connection_lost()`**	Called when one of the pipes communicating with the child process is closed.
`callback` **`process_exited()`**	Called when the child process has exited. It can be called before **`pipe_data_received()`** and **`pipe_connection_lost()`** methods.

Event Loop Policies

Policies is a low-level mechanism to alter the behavior of functions like **`asyncio.get_event_loop()`**. See also the main policies section for more details.

Accessing Policies

`asyncio.get_event_loop_policy()`	Return the current process-wide policy.
`asyncio.set_event_loop_policy()`	Set a new process-wide policy.

`AbstractEventLoopPolicy`	Base class for policy objects.

Developing with asyncio

Asynchronous programming is different from classic “sequential” programming.

This page lists common mistakes and traps and explains how to avoid them.

Debug Mode

By default asyncio runs in production mode. In order to ease the development asyncio has a *debug mode*.

There are several ways to enable asyncio debug mode:

- Setting the **`PYTHONASYNCIODEBUG`** environment variable to `1`.
- Using the Python Development Mode.
- Passing `debug=True` to **`asyncio.run()`**.
- Calling **`loop.set_debug()`**.

In addition to enabling the debug mode, consider also:

- setting the log level of the asyncio logger to **`logging.DEBUG`**, for example the following snippet of code can be run at startup of the application:

  ```
  logging.basicConfig(level=logging.DEBUG)
  ```

- configuring the **`warnings`** module to display **`ResourceWarning`** warnings. One way of doing that is by using the **`-W`** `default` command line option.

When the debug mode is enabled:

- Many non-threadsafe asyncio APIs (such as `loop.call_soon()` and `loop.call_at()` methods) raise an exception if they are called from a wrong thread.
- The execution time of the I/O selector is logged if it takes too long to perform an I/O operation.
- Callbacks taking longer than 100 milliseconds are logged. The `loop.slow_callback_duration` attribute can be used to set the minimum execution duration in seconds that is considered “slow”.

Concurrency and Multithreading

An event loop runs in a thread (typically the main thread) and executes all callbacks and Tasks in its thread. While a Task is running in the event loop, no other Tasks can run in the same thread. When a Task executes an `await` expression, the running Task gets suspended, and the event loop executes the next Task.

To schedule a callback from another OS thread, the `loop.call_soon_threadsafe()` method should be used. Example:

```
loop.call_soon_threadsafe(callback, *args)
```

Almost all asyncio objects are not thread safe, which is typically not a problem unless there is code that works with them from outside of a Task or a callback. If there’s a need for such code to call a low-level asyncio API, the `loop.call_soon_threadsafe()` method should be used, e.g.:

```
loop.call_soon_threadsafe(fut.cancel)
```

To schedule a coroutine object from a different OS thread, the `run_coroutine_threadsafe()` function should be used. It returns a `concurrent.futures.Future` to access the result:

```
async def coro_func():
     return await asyncio.sleep(1, 42)

# Later in another OS thread:

future = asyncio.run_coroutine_threadsafe(coro_func(), loop)
# Wait for the result:
result = future.result()
```

To handle signals the event loop must be run in the main thread.

The `loop.run_in_executor()` method can be used with a `concurrent.futures.ThreadPoolExecutor` or `InterpreterPoolExecutor` to execute blocking code in a different OS thread without blocking the OS thread that the event loop runs in.

There is currently no way to schedule coroutines or callbacks directly from a different process (such as one started with `multiprocessing`). The Event Loop Methods section lists APIs that can read from pipes and watch file descriptors without blocking the event loop. In addition, asyncio’s Subprocess APIs provide a way to start a process and communicate with it from the event loop. Lastly, the aforementioned `loop.run_in_executor()` method can also be used with a `concurrent.futures.ProcessPoolExecutor` to execute code in a different process.

Running Blocking Code

Blocking (CPU-bound) code should not be called directly. For example, if a function performs a CPU-intensive calculation for 1 second, all concurrent asyncio Tasks and IO operations would be delayed by 1 second.

An executor can be used to run a task in a different thread, including in a different interpreter, or even in a different process to avoid blocking the OS thread with the event loop. See the `loop.run_in_executor()` method for more details.

Logging

asyncio uses the `logging` module and all logging is performed via the `"asyncio"` logger.

The default log level is `logging.INFO`, which can be easily adjusted:

```
logging.getLogger("asyncio").setLevel(logging.WARNING)
```

Network logging can block the event loop. It is recommended to use a separate thread for handling logs or use non-blocking IO. For example, see Dealing with handlers that block.

Detect never-awaited coroutines

When a coroutine function is called, but not awaited (e.g. `coro()` instead of `await coro()`) or the coroutine is not scheduled with `asyncio.create_task()`, asyncio will emit a `RuntimeWarning`:

```
import asyncio

async def test():
    print("never scheduled")

async def main():
    test()

asyncio.run(main())
```

Output:

```
test.py:7: RuntimeWarning: coroutine 'test' was never awaited
  test()
```

Output in debug mode:

```
test.py:7: RuntimeWarning: coroutine 'test' was never awaited
Coroutine created at (most recent call last)
  File "../t.py", line 9, in <module>
    asyncio.run(main(), debug=True)

  < .. >

  File "../t.py", line 7, in main
    test()
  test()
```

The usual fix is to either await the coroutine or call the **`asyncio.create_task()`** function:

```
async def main():
    await test()
```

Detect never-retrieved exceptions

If a **`Future.set_exception()`** is called but the Future object is never awaited on, the exception would never be propagated to the user code. In this case, asyncio would emit a log message when the Future object is garbage collected.

Example of an unhandled exception:

```
import asyncio

async def bug():
    raise Exception("not consumed")

async def main():
    asyncio.create_task(bug())
asyncio.run(main())
```

Output:

```
Task exception was never retrieved
future: <Task finished coro=<bug() done, defined at test.py:3>
  exception=Exception('not consumed')>

Traceback (most recent call last):
  File "test.py", line 4, in bug
    raise Exception("not consumed")
Exception: not consumed
```

Enable the debug mode to get the traceback where the task was created:

```
asyncio.run(main(), debug=True)
```

Output in debug mode:

```
Task exception was never retrieved
future: <Task finished coro=<bug() done, defined at test.py:3>
    exception=Exception('not consumed') created at
asyncio/tasks.py:321>

source_traceback: Object created at (most recent call last):
  File "../t.py", line 9, in <module>
    asyncio.run(main(), debug=True)

< .. >

Traceback (most recent call last):
  File "../t.py", line 4, in bug
    raise Exception("not consumed")
Exception: not consumed
```

`socket` — Low-level networking interface

Source code: Lib/socket.py
[https://github.com/python/cpython/tree/3.14/Lib/socket.py]

This module provides access to the BSD *socket* interface. It is available on all modern Unix systems, Windows, MacOS, and probably additional platforms.

Note

Some behavior may be platform dependent, since calls are made to the operating system socket APIs.

Availability: not WASI.

This module does not work or is not available on WebAssembly. See WebAssembly platforms for more information.

The Python interface is a straightforward transliteration of the Unix system call and library interface for sockets to Python’s object-oriented style: the **`socket()`** function returns a *socket object* whose methods implement the various socket system calls. Parameter types are somewhat higher-level than in the C interface: as with **`read()`** and **`write()`** operations on Python files, buffer allocation on receive operations is automatic, and buffer length is implicit on send operations.

See also

Module `socketserver`
Classes that simplify writing network servers.

Module `ssl`
A TLS/SSL wrapper for socket objects.

Socket families

Depending on the system and the build options, various socket families are supported by this module.

The address format required by a particular socket object is automatically selected based on the address family specified when the socket object was created. Socket addresses are represented as follows:

- The address of an `AF_UNIX` socket bound to a file system node is represented as a string, using the file system encoding and the `'surrogateescape'` error handler (see **PEP 383** [https://peps.python.org/pep-0383/]). An address in Linux’s abstract namespace is returned as a bytes-like object with an initial null byte; note that sockets in this namespace can communicate with normal file system sockets, so programs intended to run on Linux may need to deal with both types of address. A string or bytes-like object can be used for either type of address when passing it as an argument.

  Changed in version 3.3: Previously, `AF_UNIX` socket paths were assumed to use UTF-8 encoding.

  Changed in version 3.5: Writable bytes-like object is now accepted.

- A pair `(host, port)` is used for the **`AF_INET`** address family, where *host* is a string representing either a hostname in internet domain notation like `'daring.cwi.nl'` or an IPv4 address like `'100.50.200.5'`, and *port* is an integer.
    - For IPv4 addresses, two special forms are accepted instead of a host address: `''` represents **`INADDR_ANY`**, which is used to bind to all interfaces, and the string `'<broadcast>'` represents **`INADDR_BROADCAST`**. This behavior is not compatible with IPv6, therefore, you may want to avoid these if you intend to support IPv6 with your Python programs.
- For **`AF_INET6`** address family, a four-tuple `(host, port, flowinfo, scope_id)` is used, where *flowinfo* and *scope_id* represent the `sin6_flowinfo` and `sin6_scope_id` members in **`struct sockaddr_in6`** in C. For **`socket`** module methods, *flowinfo* and *scope_id* can be omitted just for backward compatibility. Note, however, omission of *scope_id* can cause problems in manipulating scoped IPv6 addresses.

  Changed in version 3.7: For multicast addresses (with *scope_id* meaningful) *address* may not contain `%scope_id` (or `zone id`) part. This information is superfluous and may be safely omitted (recommended).
- **`AF_NETLINK`** sockets are represented as pairs `(pid, groups)`.
- Linux-only support for TIPC is available using the **`AF_TIPC`** address family. TIPC is an open, non-IP based networked protocol designed for use in clustered computer environments. Addresses are represented by a tuple, and the fields depend on the address type. The general tuple form is `(addr_type, v1, v2, v3 [, scope])`, where:
    - *addr_type* is one of **`TIPC_ADDR_NAMESEQ`**, **`TIPC_ADDR_NAME`**, or **`TIPC_ADDR_ID`**.

  - *scope* is one of **`TIPC_ZONE_SCOPE`**, **`TIPC_CLUSTER_SCOPE`**, and **`TIPC_NODE_SCOPE`**.
  - If *addr_type* is **`TIPC_ADDR_NAME`**, then *v1* is the server type, *v2* is the port identifier, and *v3* should be 0.

    If *addr_type* is **`TIPC_ADDR_NAMESEQ`**, then *v1* is the server type, *v2* is the lower port number, and *v3* is the upper port number.

    If *addr_type* is **`TIPC_ADDR_ID`**, then *v1* is the node, *v2* is the reference, and *v3* should be set to 0.

- A tuple `(interface, )` is used for the **`AF_CAN`** address family, where *interface* is a string representing a network interface name like `'can0'`. The network interface name `''` can be used to receive packets from all network interfaces of this family.
  - **`CAN_ISOTP`** protocol require a tuple `(interface, rx_addr, tx_addr)` where both additional parameters are unsigned long integer that represent a CAN identifier (standard or extended).
  - **`CAN_J1939`** protocol require a tuple `(interface, name, pgn, addr)` where additional parameters are 64-bit unsigned integer representing the ECU name, a 32-bit unsigned integer representing the Parameter Group Number (PGN), and an 8-bit integer representing the address.

- A string or a tuple `(id, unit)` is used for the **`SYSPROTO_CONTROL`** protocol of the **`PF_SYSTEM`** family. The string is the name of a kernel control using a dynamically assigned ID. The tuple can be used if ID and unit number of the kernel control are known or if a registered ID is used.

Added in version 3.3.

- **AF_BLUETOOTH** supports the following protocols and address formats:
    - **BTPROTO_L2CAP** accepts a tuple `(bdaddr, psm[, cid[, bdaddr_type]])` where:
        - `bdaddr` is a string specifying the Bluetooth address.
        - `psm` is an integer specifying the Protocol/Service Multiplexer.
        - `cid` is an optional integer specifying the Channel Identifier. If not given, defaults to zero.
        - `bdaddr_type` is an optional integer specifying the address type; one of **BDADDR_BREDR** (default), **BDADDR_LE_PUBLIC**, **BDADDR_LE_RANDOM**.

      Changed in version 3.14: Added `cid` and `bdaddr_type` fields.

    - **BTPROTO_RFCOMM** accepts `(bdaddr, channel)` where `bdaddr` is the Bluetooth address as a string and `channel` is an integer.
    - **BTPROTO_HCI** accepts a format that depends on your OS.
        - On Linux it accepts an integer `device_id` or a tuple `(device_id, [channel])` where `device_id` specifies the number of the Bluetooth device, and `channel` is an optional integer specifying the HCI channel (**HCI_CHANNEL_RAW** by default).
        - On FreeBSD, NetBSD and DragonFly BSD it accepts `bdaddr` where `bdaddr` is the Bluetooth address as a string.

    Changed in version 3.2: NetBSD and DragonFlyBSD support added.

    Changed in version 3.13.3: FreeBSD support added.

    Changed in version 3.14: Added `channel` field. `device_id` not packed in a tuple is now accepted.

  - **`BTPROTO_SCO`** accepts `bdaddr` where `bdaddr` is the Bluetooth address as a string or a **`bytes`** object. (ex. `'12:23:34:45:56:67'` or `b'12:23:34:45:56:67'`)

    Changed in version 3.14: FreeBSD support added.

- **`AF_ALG`** is a Linux-only socket based interface to Kernel cryptography. An algorithm socket is configured with a tuple of two to four elements `(type, name [, feat [, mask]])`, where:

  - *type* is the algorithm type as string, e.g. `aead`, `hash`, `skcipher` or `rng`.
  - *name* is the algorithm name and operation mode as string, e.g. `sha256`, `hmac(sha256)`, `cbc(aes)` or `drbg_nopr_ctr_aes256`.
  - *feat* and *mask* are unsigned 32bit integers.

  Availability: Linux >= 2.6.38.

  Some algorithm types require more recent Kernels.

  Added in version 3.6.

- **`AF_VSOCK`** allows communication between virtual machines and their hosts. The sockets are represented as a `(CID, port)` tuple where the context ID or CID and port are integers.

Availability: Linux >= 3.9

See *vsock(7)* *[https://manpages.debian.org/vsock(7)]*

Added in version 3.7.

- **`AF_PACKET`** is a low-level interface directly to network devices. The addresses are represented by the tuple `(ifname, proto[, pkttype[, hatype[, addr]]])` where:
    - *ifname* - String specifying the device name.
    - *proto* - The Ethernet protocol number. May be **`ETH_P_ALL`** to capture all protocols, one of the ETHERTYPE_* constants or any other Ethernet protocol number.
    - *pkttype* - Optional integer specifying the packet type:
        - `PACKET_HOST` (the default) - Packet addressed to the local host.
        - `PACKET_BROADCAST` - Physical-layer broadcast packet.
        - `PACKET_MULTICAST` - Packet sent to a physical-layer multicast address.
        - `PACKET_OTHERHOST` - Packet to some other host that has been caught by a device driver in promiscuous mode.
        - `PACKET_OUTGOING` - Packet originating from the local host that is looped back to a packet socket.
    - *hatype* - Optional integer specifying the ARP hardware address type.
    - *addr* - Optional bytes-like object specifying the hardware physical address, whose interpretation depends on the device.

  Availability: Linux >= 2.2.

- **`AF_QIPCRTR`** is a Linux-only socket based interface for communicating with services running on co-processors in Qualcomm platforms. The address family is represented as a `(node, port)` tuple where the *node* and *port* are non-negative integers.

  Availability: Linux >= 4.7.

  Added in version 3.8.

- **`IPPROTO_UDPLITE`** is a variant of UDP which allows you to specify what portion of a packet is covered with the checksum. It adds two socket options that you can change. `self.setsockopt(IPPROTO_UDPLITE, UDPLITE_SEND_CSCOV, length)` will change what portion of outgoing packets are covered by the checksum and `self.setsockopt(IPPROTO_UDPLITE, UDPLITE_RECV_CSCOV, length)` will filter out packets which cover too little of their data. In both cases `length` should be in `range(8, 2**16, 8)`.

  Such a socket should be constructed with `socket(AF_INET, SOCK_DGRAM, IPPROTO_UDPLITE)` for IPv4 or `socket(AF_INET6, SOCK_DGRAM, IPPROTO_UDPLITE)` for IPv6.

  Availability: Linux >= 2.6.20, FreeBSD >= 10.1

  Added in version 3.9.

- **`AF_HYPERV`** is a Windows-only socket based interface for communicating with Hyper-V hosts and guests. The address family is represented as a `(vm_id, service_id)` tuple where the `vm_id` and `service_id` are UUID strings.

  The `vm_id` is the virtual machine identifier or a set of known VMID values if the target is not a specific virtual machine. Known VMID constants defined on `socket` are:

    - `HV_GUID_ZERO`
    - `HV_GUID_BROADCAST`
    - `HV_GUID_WILDCARD` - Used to bind on itself and accept connections from all partitions.
    - `HV_GUID_CHILDREN` - Used to bind on itself and accept connection from child partitions.
    - `HV_GUID_LOOPBACK` - Used as a target to itself.
    - `HV_GUID_PARENT` - When used as a bind accepts connection from the parent partition. When used as an address target it will connect to the parent partition.

  The `service_id` is the service identifier of the registered service.

  Added in version 3.12.

If you use a hostname in the *host* portion of IPv4/v6 socket address, the program may show a nondeterministic behavior, as Python uses the first address returned from the DNS resolution. The socket address will be resolved differently into an actual IPv4/v6 address, depending on the results from DNS resolution and/or the host configuration. For deterministic behavior use a numeric address in *host* portion.

All errors raise exceptions. The normal exceptions for invalid argument types and out-of-memory conditions can be raised. Errors related to socket or address semantics raise **`OSError`** or one of its subclasses.

Non-blocking mode is supported through **`setblocking()`**. A generalization of this based on timeouts is supported through **`settimeout()`**.

Module contents

The module `socket` exports the following elements.

Exceptions

exception socket.error

A deprecated alias of `OSError`.

Changed in version 3.3: Following **PEP 3151** [https://peps.python.org/pep-3151/], this class was made an alias of `OSError`.

exception socket.herror

A subclass of `OSError`, this exception is raised for address-related errors, i.e. for functions that use *h_errno* in the POSIX C API, including `gethostbyname_ex()` and `gethostbyaddr()`. The accompanying value is a pair `(h_errno, string)` representing an error returned by a library call. *h_errno* is a numeric value, while *string* represents the description of *h_errno*, as returned by the **`hstrerror()`** C function.

Changed in version 3.3: This class was made a subclass of `OSError`.

exception socket.gaierror

A subclass of `OSError`, this exception is raised for address-related errors by `getaddrinfo()` and `getnameinfo()`. The accompanying value is a pair `(error, string)` representing an error returned by a library call. *string* represents the description of *error*, as returned by the **`gai_strerror()`** C function. The numeric *error* value will match one of the **`EAI_*`** constants defined in this module.

Changed in version 3.3: This class was made a subclass of `OSError`.

exception socket.timeout

A deprecated alias of `TimeoutError`.

A subclass of `OSError`, this exception is raised when a timeout occurs on a socket which has had timeouts enabled via a prior call to `settimeout()` (or implicitly through `setdefaulttimeout()`). The accompanying value is a string whose value is currently always “timed out”.

Changed in version 3.3: This class was made a subclass of `OSError`.

Changed in version 3.10: This class was made an alias of `TimeoutError`.

Constants

The AF_* and SOCK_* constants are now **`AddressFamily`** and **`SocketKind`** `IntEnum` collections.

Added in version 3.4.

socket.AF_UNIX
socket.AF_INET
socket.AF_INET6

These constants represent the address (and protocol) families, used for the first argument to `socket()`. If the `AF_UNIX` constant is not defined then this protocol is unsupported. More constants may be available depending on the system.

socket.AF_UNSPEC

`AF_UNSPEC` means that `getaddrinfo()` should return socket addresses for any address family (either IPv4, IPv6, or any other) that can be used.

socket.SOCK_STREAM
socket.SOCK_DGRAM
socket.SOCK_RAW
socket.SOCK_RDM
socket.SOCK_SEQPACKET

These constants represent the socket types, used for the second argument to `socket()`. More constants may be available depending on the system. (Only `SOCK_STREAM` and `SOCK_DGRAM` appear to be generally useful.)

socket.SOCK_CLOEXEC
socket.SOCK_NONBLOCK

These two constants, if defined, can be combined with the socket types and allow you to set some flags atomically (thus avoiding possible race conditions and the need for separate calls).

See also

Secure File Descriptor Handling [https://udrepper.livejournal.com/20407.html] for a more thorough explanation.

Availability: Linux >= 2.6.27.

Added in version 3.2.

SO_*
socket.SOMAXCONN
MSG_*
SOL_*
SCM_*
IPPROTO_*
IPPORT_*
INADDR_*
IP_*

IPV6_*
EAI_*
AI_*
NI_*
TCP_*

Many constants of these forms, documented in the Unix documentation on sockets and/or the IP protocol, are also defined in the socket module. They are generally used in arguments to the `setsockopt()` and `getsockopt()` methods of socket objects. In most cases, only those symbols that are defined in the Unix header files are defined; for a few symbols, default values are provided.

Changed in version 3.6: `SO_DOMAIN`, `SO_PROTOCOL`, `SO_PEERSEC`, `SO_PASSSEC`, `TCP_USER_TIMEOUT`, `TCP_CONGESTION` were added.

Changed in version 3.6.5: Added support for `TCP_FASTOPEN`, `TCP_KEEPCNT` on Windows platforms when available.

Changed in version 3.7: `TCP_NOTSENT_LOWAT` was added.

Added support for `TCP_KEEPIDLE`, `TCP_KEEPINTVL` on Windows platforms when available.

Changed in version 3.10: `IP_RECVTOS` was added. Added `TCP_KEEPALIVE`. On MacOS this constant can be used in the same way that `TCP_KEEPIDLE` is used on Linux.

Changed in version 3.11: Added `TCP_CONNECTION_INFO`. On MacOS this constant can be used in the same way that `TCP_INFO` is used on Linux and BSD.

Changed in version 3.12: Added `SO_RTABLE` and `SO_USER_COOKIE`. On OpenBSD and FreeBSD respectively those constants can be used in the same way that `SO_MARK` is used on Linux. Also added missing TCP socket options from Linux: `TCP_MD5SIG`, `TCP_THIN_LINEAR_TIMEOUTS`, `TCP_THIN_DUPACK`, `TCP_REPAIR`, `TCP_REPAIR_QUEUE`, `TCP_QUEUE_SEQ`, `TCP_REPAIR_OPTIONS`, `TCP_TIMESTAMP`, `TCP_CC_INFO`, `TCP_SAVE_SYN`, `TCP_SAVED_SYN`, `TCP_REPAIR_WINDOW`, `TCP_FASTOPEN_CONNECT`, `TCP_ULP`, `TCP_MD5SIG_EXT`, `TCP_FASTOPEN_KEY`, `TCP_FASTOPEN_NO_COOKIE`, `TCP_ZEROCOPY_RECEIVE`, `TCP_INQ`, `TCP_TX_DELAY`. Added `IP_PKTINFO`, `IP_UNBLOCK_SOURCE`, `IP_BLOCK_SOURCE`, `IP_ADD_SOURCE_MEMBERSHIP`, `IP_DROP_SOURCE_MEMBERSHIP`.

Changed in version 3.13: Added `SO_BINDTOIFINDEX`. On Linux this constant can be used in the same way that `SO_BINDTODEVICE` is used, but with the index of a network interface instead of its name.

Changed in version 3.14: Added missing `IP_FREEBIND`, `IP_RECVERR`, `IPV6_RECVERR`, `IP_RECVTTL`, and `IP_RECVORIGDSTADDR` on Linux.

Changed in version 3.14: Added support for `TCP_QUICKACK` on Windows platforms when available.

socket.AF_CAN
socket.PF_CAN
SOL_CAN_*
CAN_*

Many constants of these forms, documented in the Linux documentation, are also defined in the socket module.

Availability: Linux >= 2.6.25, NetBSD >= 8.

Added in version 3.3.

Changed in version 3.11: NetBSD support was added.

Changed in version 3.14: Restored missing `CAN_RAW_ERR_FILTER` on Linux.

socket.CAN_BCM

CAN_BCM_*

CAN_BCM, in the CAN protocol family, is the broadcast manager (BCM) protocol. Broadcast manager constants, documented in the Linux documentation, are also defined in the socket module.

Availability: Linux >= 2.6.25.

Note

The **`CAN_BCM_CAN_FD_FRAME`** flag is only available on Linux >= 4.8.

Added in version 3.4.

socket.CAN_RAW_FD_FRAMES

Enables CAN FD support in a CAN_RAW socket. This is disabled by default. This allows your application to send both CAN and CAN FD frames; however, you must accept both CAN and CAN FD frames when reading from the socket.

This constant is documented in the Linux documentation.

Availability: Linux >= 3.6.

Added in version 3.5.

socket.CAN_RAW_JOIN_FILTERS

Joins the applied CAN filters such that only CAN frames that match all given CAN filters are passed to user space.

This constant is documented in the Linux documentation.

Availability: Linux >= 4.1.

Added in version 3.9.

socket.CAN_ISOTP

CAN_ISOTP, in the CAN protocol family, is the ISO-TP (ISO 15765-2) protocol. ISO-TP constants, documented in the Linux documentation.

Availability: Linux >= 2.6.25.

Added in version 3.7.

socket.CAN_J1939

CAN_J1939, in the CAN protocol family, is the SAE J1939 protocol. J1939 constants, documented in the Linux documentation.

Availability: Linux >= 5.4.

Added in version 3.9.

socket.AF_DIVERT
socket.PF_DIVERT

These two constants, documented in the FreeBSD divert(4) manual page, are also defined in the socket module.

Availability: FreeBSD >= 14.0.

Added in version 3.12.

socket.AF_PACKET
socket.PF_PACKET
PACKET_*

Many constants of these forms, documented in the Linux documentation, are also defined in the socket module.

Availability: Linux >= 2.2.

socket.ETH_P_ALL

`ETH_P_ALL` can be used in the **`socket`** constructor as *proto* for the **`AF_PACKET`** family in order to capture every packet, regardless of protocol.

For more information, see the *packet(7)* *[https://manpages.debian.org/packet(7)]* manpage.

Availability: Linux.

Added in version 3.12.

socket.AF_RDS
socket.PF_RDS
socket.SOL_RDS
RDS_*

Many constants of these forms, documented in the Linux documentation, are also defined in the socket module.

Availability: Linux >= 2.6.30.

Added in version 3.3.

socket.SIO_RCVALL
socket.SIO_KEEPALIVE_VALS
socket.SIO_LOOPBACK_FAST_PATH
RCVALL_*

Constants for Windows' WSAIoctl(). The constants are used as arguments to the **`ioctl()`** method of socket objects.

Changed in version 3.6: `SIO_LOOPBACK_FAST_PATH` was added.

TIPC_*

TIPC related constants, matching the ones exported by the C socket API. See the TIPC documentation for more information.

socket.AF_ALG
socket.SOL_ALG
ALG_*

Constants for Linux Kernel cryptography.

Availability: Linux >= 2.6.38.

Added in version 3.6.

socket.AF_VSOCK
socket.IOCTL_VM_SOCKETS_GET_LOCAL_CID
VMADDR*
SO_VM*

Constants for Linux host/guest communication.

Availability: Linux >= 4.8.

Added in version 3.7.

socket.AF_LINK

Availability: BSD, macOS.

Added in version 3.4.

socket.has_ipv6

This constant contains a boolean value which indicates if IPv6 is supported on this platform.

socket.AF_BLUETOOTH
socket.BTPROTO_L2CAP
socket.BTPROTO_RFCOMM
socket.BTPROTO_HCI
socket.BTPROTO_SCO

Integer constants for use with Bluetooth addresses.

socket.BDADDR_ANY
socket.BDADDR_LOCAL

These are string constants containing Bluetooth addresses with special meanings. For example, `BDADDR_ANY` can be used to indicate any address when specifying the binding socket with `BTPROTO_RFCOMM`.

socket.BDADDR_BREDR
socket.BDADDR_LE_PUBLIC
socket.BDADDR_LE_RANDOM

These constants describe the Bluetooth address type when binding or connecting a `BTPROTO_L2CAP` socket.

Availability: Linux, FreeBSD

Added in version 3.14.

socket.SOL_RFCOMM
socket.SOL_L2CAP
socket.SOL_HCI
socket.SOL_SCO
socket.SOL_BLUETOOTH

Used in the level argument to the `setsockopt()` and `getsockopt()` methods of Bluetooth socket objects.

`SOL_BLUETOOTH` is only available on Linux. Other constants are available if the corresponding protocol is supported.

SO_L2CAP_*
socket.L2CAP_LM
L2CAP_LM_*
SO_RFCOMM_*
RFCOMM_LM_*
SO_SCO_*
SO_BTH_*
BT_*

Used in the option name and value argument to the `setsockopt()` and `getsockopt()` methods of Bluetooth socket objects.

`BT_*` and **`L2CAP_LM`** are only available on Linux. **`SO_BTH_*`** are only available on Windows. Other constants may be available on Linux and various BSD platforms.

Added in version 3.14.

socket.HCI_FILTER
socket.HCI_TIME_STAMP
socket.HCI_DATA_DIR
socket.SO_HCI_EVT_FILTER
socket.SO_HCI_PKT_FILTER

Option names for use with **`BTPROTO_HCI`**. Availability and format of the option values depend on platform.

Changed in version 3.14: Added **`SO_HCI_EVT_FILTER`** and **`SO_HCI_PKT_FILTER`** on NetBSD and DragonFly BSD. Added **`HCI_DATA_DIR`** on FreeBSD, NetBSD and DragonFly BSD.

socket.HCI_DEV_NONE

The `device_id` value used to create an HCI socket that isn’t specific to a single Bluetooth adapter.

Availability: Linux

Added in version 3.14.

socket.HCI_CHANNEL_RAW
socket.HCI_CHANNEL_USER
socket.HCI_CHANNEL_MONITOR
socket.HCI_CHANNEL_CONTROL
socket.HCI_CHANNEL_LOGGING

Possible values for `channel` field in the **`BTPROTO_HCI`** address.

Availability: Linux

Added in version 3.14.

socket.AF_QIPCRTR

Constant for Qualcomm’s IPC router protocol, used to communicate with service providing remote processors.

Availability: Linux >= 4.7.

socket.SCM_CREDS2
socket.LOCAL_CREDS
socket.LOCAL_CREDS_PERSISTENT

LOCAL_CREDS and LOCAL_CREDS_PERSISTENT can be used with SOCK_DGRAM, SOCK_STREAM sockets, equivalent to Linux/DragonFlyBSD SO_PASSCRED, while LOCAL_CREDS sends the credentials at first read, LOCAL_CREDS_PERSISTENT sends for each read, SCM_CREDS2 must be then used for the latter for the message type.

Added in version 3.11.

Availability: FreeBSD.

socket.SO_INCOMING_CPU

Constant to optimize CPU locality, to be used in conjunction with **`SO_REUSEPORT`**.

Added in version 3.11.

Availability: Linux >= 3.9

socket.SO_REUSEPORT_LB

Constant to enable duplicate address and port bindings with load balancing.

Added in version 3.14.

Availability: FreeBSD >= 12.0

socket.AF_HYPERV
socket.HV_PROTOCOL_RAW
socket.HVSOCKET_CONNECT_TIMEOUT
socket.HVSOCKET_CONNECT_TIMEOUT_MAX
socket.HVSOCKET_CONNECTED_SUSPEND
socket.HVSOCKET_ADDRESS_FLAG_PASSTHRU
socket.HV_GUID_ZERO
socket.HV_GUID_WILDCARD
socket.HV_GUID_BROADCAST
socket.HV_GUID_CHILDREN
socket.HV_GUID_LOOPBACK
socket.HV_GUID_PARENT

Constants for Windows Hyper-V sockets for host/guest communications.

Availability: Windows.

Added in version 3.12.

socket.ETHERTYPE_ARP
socket.ETHERTYPE_IP
socket.ETHERTYPE_IPV6
socket.ETHERTYPE_VLAN

IEEE 802.3 protocol number [https://www.iana.org/assignments/ieee-802-numbers/ieee-802-numbers.txt]. constants.

Availability: Linux, FreeBSD, macOS.

Added in version 3.12.

socket.SHUT_RD
socket.SHUT_WR
socket.SHUT_RDWR

These constants are used by the `shutdown()` method of socket objects.

Availability: not WASI.

Functions

Creating sockets

The following functions all create socket objects.

class socket.socket(*family=AF_INET*, *type=SOCK_STREAM*, *proto=0*, *fileno=None*)

Create a new socket using the given address family, socket type and protocol number. The address family should be `AF_INET` (the default), `AF_INET6`, `AF_UNIX`, `AF_CAN`, `AF_PACKET`, or `AF_RDS`. The socket type should be `SOCK_STREAM` (the default), `SOCK_DGRAM`, `SOCK_RAW` or perhaps one of the other `SOCK_` constants. The protocol number is usually zero and may be omitted or in the case where the address family is `AF_CAN` the protocol should be one of **`CAN_RAW`**, `CAN_BCM`, `CAN_ISOTP` or `CAN_J1939`.

If *fileno* is specified, the values for *family*, *type*, and *proto* are auto-detected from the specified file descriptor. Auto-detection can be overruled by calling the function with explicit *family*, *type*, or *proto* arguments. This only affects how Python represents e.g. the return value of `socket.getpeername()` but not the actual OS resource. Unlike `socket.fromfd()`, *fileno* will return the same socket and not a duplicate. This may help close a detached socket using `socket.close()`.

The newly created socket is non-inheritable.

Raises an auditing event `socket.__new__` with arguments `self`, `family`, `type`, `protocol`.

Changed in version 3.3: The AF_CAN family was added. The AF_RDS family was added.

Changed in version 3.4: The CAN_BCM protocol was added.

Changed in version 3.4: The returned socket is now non-inheritable.

Changed in version 3.7: The CAN_ISOTP protocol was added.

Changed in version 3.7: When **`SOCK_NONBLOCK`** or **`SOCK_CLOEXEC`** bit flags are applied to *type* they are cleared, and **`socket.type`** will not reflect them. They are still passed to the underlying system `socket()` call. Therefore,

```
sock = socket.socket(
    socket.AF_INET,
    socket.SOCK_STREAM | socket.SOCK_NONBLOCK)
```

will still create a non-blocking socket on OSes that support `SOCK_NONBLOCK`, but `sock.type` will be set to `socket.SOCK_STREAM`.

Changed in version 3.9: The CAN_J1939 protocol was added.

Changed in version 3.10: The IPPROTO_MPTCP protocol was added.

socket.socketpair([*family*[, *type*[, *proto*]]])

Build a pair of connected socket objects using the given address family, socket type, and protocol number.

Address family, socket type, and protocol number are as for the [`socket()`] function above. The default family is [`AF_UNIX`] if defined on the platform; otherwise, the default is [`AF_INET`].

The newly created sockets are non-inheritable.

Changed in version 3.2: The returned socket objects now support the whole socket API, rather than a subset.

Changed in version 3.4: The returned sockets are now non-inheritable.

Changed in version 3.5: Windows support added.

socket.create_connection(*address*, *timeout=GLOBAL_DEFAULT*, *source_address=None*, ***, *all_errors=False*)

Connect to a TCP service listening on the internet *address* (a 2-tuple `(host, port)`), and return the socket object. This is a higher-level function than **`socket.connect()`**: if *host* is a non-numeric hostname, it will try to resolve it for both **`AF_INET`** and **`AF_INET6`**, and then try to connect to all possible addresses in turn until a connection succeeds. This makes it easy to write clients that are compatible to both IPv4 and IPv6.

Passing the optional *timeout* parameter will set the timeout on the socket instance before attempting to connect. If no *timeout* is supplied, the global default timeout setting returned by **`getdefaulttimeout()`** is used.

If supplied, *source_address* must be a 2-tuple `(host, port)` for the socket to bind to as its source address before connecting. If host or port are ‘’ or 0 respectively the OS default behavior will be used.

When a connection cannot be created, an exception is raised. By default, it is the exception from the last address in the list. If *all_errors* is `True`, it is an **`ExceptionGroup`** containing the errors of all attempts.

Changed in version 3.2: source_address was added.

Changed in version 3.11: all_errors was added.

socket.create_server(*address*, *, *family=AF_INET*, *backlog=None*, *reuse_port=False*, *dualstack_ipv6=False*)

Convenience function which creates a TCP socket bound to *address* (a 2-tuple `(host, port)`) and returns the socket object.

family should be either **`AF_INET`** or **`AF_INET6`**. *backlog* is the queue size passed to **`socket.listen()`**; if not specified , a default reasonable value is chosen. *reuse_port* dictates whether to set the **`SO_REUSEPORT`** socket option.

If *dualstack_ipv6* is true, *family* is **`AF_INET6`** and the platform supports it the socket will be able to accept both IPv4 and IPv6 connections, else it will raise **`ValueError`**. Most POSIX platforms and Windows are supposed to support this functionality. When this functionality is enabled the address returned by **`socket.getpeername()`** when an IPv4 connection occurs will be an IPv6 address represented as an IPv4-mapped IPv6 address. If *dualstack_ipv6* is false it will explicitly disable this functionality on platforms that enable it by default (e.g. Linux). This parameter can be used in conjunction with **`has_dualstack_ipv6()`**:

```
import socket

addr = ("", 8080)  # all interfaces, port 8080
if socket.has_dualstack_ipv6():
    s = socket.create_server(addr, family=socket.AF_INET6,
dualstack_ipv6=True)
else:
    s = socket.create_server(addr)
```

Note

On POSIX platforms the **`SO_REUSEADDR`** socket option is set in order to immediately reuse previous sockets which were bound on the same *address* and remained in TIME_WAIT state.

Added in version 3.8.

socket.has_dualstack_ipv6()

Return `True` if the platform supports creating a TCP socket which can handle both IPv4 and IPv6 connections.

Added in version 3.8.

socket.fromfd(*fd*, *family*, *type*, *proto=0*)

Duplicate the file descriptor *fd* (an integer as returned by a file object's **`fileno()`** method) and build a socket object from the result. Address family, socket type and protocol number are as for the **`socket()`** function above. The file descriptor should refer to a socket, but this is not checked — subsequent operations on the object may fail if the file descriptor is invalid. This function is rarely needed, but can be used to get or set socket options on a socket passed to a program as standard input or output (such as a server started by the Unix inet daemon). The socket is assumed to be in blocking mode.

The newly created socket is non-inheritable.

Changed in version 3.4: The returned socket is now non-inheritable.

socket.fromshare(*data*)

Instantiate a socket from data obtained from the `socket.share()` method. The socket is assumed to be in blocking mode.

Availability: Windows.

Added in version 3.3.

socket.SocketType

This is a Python type object that represents the socket object type. It is the same as `type(socket(...))`.

Other functions

The `socket` module also offers various network-related services:

socket.close(*fd*)

Close a socket file descriptor. This is like `os.close()`, but for sockets. On some platforms (most noticeable Windows) `os.close()` does not work for socket file descriptors.

Added in version 3.7.

socket.getaddrinfo(*host*, *port*, *family=AF_UNSPEC*, *type=0*, *proto=0*, *flags=0*)

This function wraps the C function `getaddrinfo` of the underlying system.

Translate the *host*/*port* argument into a sequence of 5-tuples that contain all the necessary arguments for creating a socket connected to that service. *host* is a domain name, a string representation of an IPv4/v6 address or `None`. *port* is a string service name such as `'http'`, a numeric port number or `None`. By passing `None` as the value of *host* and *port*, you can pass `NULL` to the underlying C API.

The *family*, *type* and *proto* arguments can be optionally specified in order to provide options and limit the list of addresses returned. Pass their default values (**`AF_UNSPEC`**, 0, and 0, respectively) to not limit the results. See the note below for details.

The *flags* argument can be one or several of the `AI_*` constants, and will influence how results are computed and returned. For example, **`AI_NUMERICHOST`** will disable domain name resolution and will raise an error if *host* is a domain name.

The function returns a list of 5-tuples with the following structure:

```
(family, type, proto, canonname, sockaddr)
```

In these tuples, *family*, *type*, *proto* are all integers and are meant to be passed to the **`socket()`** function. *canonname* will be a string representing the canonical name of the *host* if **`AI_CANONNAME`** is part of the *flags* argument; else *canonname* will be empty. *sockaddr* is a tuple describing a socket address, whose format depends on the returned *family* (a `(address, port)` 2-tuple for **`AF_INET`**, a `(address, port, flowinfo, scope_id)` 4-tuple for **`AF_INET6`**), and is meant to be passed to the **`socket.connect()`** method.

Note

If you intend to use results from **`getaddrinfo()`** to create a socket (rather than, for example, retrieve *canonname*), consider limiting the results by *type* (e.g. `SOCK_STREAM` or `SOCK_DGRAM`) and/or *proto* (e.g. **`IPPROTO_TCP`** or **`IPPROTO_UDP`**) that your application can handle.

The behavior with default values of *family*, *type*, *proto* and *flags* is system-specific.

Many systems (for example, most Linux configurations) will return a sorted list of all matching addresses. These addresses should generally be tried in order until a connection succeeds (possibly tried in parallel, for example, using a Happy Eyeballs [https://en.wikipedia.org/wiki/Happy_Eyeballs] algorithm). In these cases, limiting the *type* and/or *proto* can help eliminate unsuccessful or unusable connection attempts.

Some systems will, however, only return a single address. (For example, this was reported on Solaris and AIX configurations.) On these systems, limiting the *type* and/or *proto* helps ensure that this address is usable.

Raises an auditing event `socket.getaddrinfo` with arguments `host`, `port`, `family`, `type`, `protocol`.

The following example fetches address information for a hypothetical TCP connection to `example.org` on port 80 (results may differ on your system if IPv6 isn’t enabled):

```
>>> socket.getaddrinfo("example.org", 80,
proto=socket.IPPROTO_TCP)
[(socket.AF_INET6, socket.SOCK_STREAM,
 6, '', ('2606:2800:220:1:248:1893:25c8:1946', 80, 0, 0)),
 (socket.AF_INET, socket.SOCK_STREAM,
 6, '', ('93.184.216.34', 80))]
```

Changed in version 3.2: parameters can now be passed using keyword arguments.

Changed in version 3.7: for IPv6 multicast addresses, string representing an address will not contain `%scope_id` part.

socket.getfqdn([*name*])

Return a fully qualified domain name for *name*. If *name* is omitted or empty, it is interpreted as the local host. To find the fully qualified name, the hostname returned by **`gethostbyaddr()`** is checked, followed by aliases for the host, if available. The first name which includes a period is selected. In case no fully qualified domain name is available and *name* was provided, it is returned unchanged. If *name* was empty or equal to `'0.0.0.0'`, the hostname from **`gethostname()`** is returned.

socket.gethostbyname(*hostname*)

Translate a host name to IPv4 address format. The IPv4 address is returned as a string, such as `'100.50.200.5'`. If the host name is an IPv4 address itself it is returned unchanged. See **`gethostbyname_ex()`** for a more complete interface. **`gethostbyname()`** does not support IPv6 name resolution, and **`getaddrinfo()`** should be used instead for IPv4/v6 dual stack support.

Raises an auditing event `socket.gethostbyname` with argument `hostname`.

Availability: not WASI.

socket.gethostbyname_ex(*hostname*)

Translate a host name to IPv4 address format, extended interface. Return a 3-tuple `(hostname, aliaslist, ipaddrlist)` where *hostname* is the host’s primary host name, *aliaslist* is a (possibly empty) list of alternative host names for the same address, and *ipaddrlist* is a list of IPv4 addresses for the same interface on the same host (often but not always a single address). **`gethostbyname_ex()`** does not support IPv6 name resolution, and **`getaddrinfo()`** should be used instead for IPv4/v6 dual stack support.

Raises an auditing event `socket.gethostbyname` with argument `hostname`.

Availability: not WASI.

socket.gethostname()

Return a string containing the hostname of the machine where the Python interpreter is currently executing.

Raises an auditing event `socket.gethostname` with no arguments.

Note: **`gethostname()`** doesn’t always return the fully qualified domain name; use **`getfqdn()`** for that.

Availability: not WASI.

socket.gethostbyaddr(*ip_address*)

Return a 3-tuple `(hostname, aliaslist, ipaddrlist)` where *hostname* is the primary host name responding to the given *ip_address*, *aliaslist* is a (possibly empty) list of alternative host names for the same address, and *ipaddrlist* is a list of IPv4/v6 addresses for the same interface on the same host (most likely containing only a single address). To find the fully qualified domain name, use the function **`getfqdn()`**. **`gethostbyaddr()`** supports both IPv4 and IPv6.

Raises an auditing event `socket.gethostbyaddr` with argument `ip_address`.

Availability: not WASI.

socket.getnameinfo(*sockaddr*, *flags*)

Translate a socket address *sockaddr* into a 2-tuple `(host, port)`. Depending on the settings of *flags*, the result can contain a fully qualified domain name or numeric address representation in *host*. Similarly, *port* can contain a string port name or a numeric port number.

For IPv6 addresses, `%scope_id` is appended to the host part if *sockaddr* contains meaningful *scope_id*. Usually this happens for multicast addresses.

For more information about *flags* you can consult *getnameinfo(3)* *[https://manpages.debian.org/getnameinfo(3)]*.

Raises an auditing event `socket.getnameinfo` with argument `sockaddr`.

Availability: not WASI.

socket.getprotobyname(*protocolname*)

Translate an internet protocol name (for example, `'icmp'`) to a constant suitable for passing as the (optional) third argument to the **`socket()`** function. This is usually only needed for sockets opened in “raw” mode (**`SOCK_RAW`**); for the normal socket modes, the correct protocol is chosen automatically if the protocol is omitted or zero.

Availability: not WASI.

socket.getservbyname(*servicename*[, *protocolname*])

Translate an internet service name and protocol name to a port number for that service. The optional protocol name, if given, should be `'tcp'` or `'udp'`, otherwise any protocol will match.

Raises an auditing event `socket.getservbyname` with arguments `servicename`, `protocolname`.

Availability: not WASI.

socket.getservbyport(*port*[, *protocolname*])

Translate an internet port number and protocol name to a service name for that service. The optional protocol name, if given, should be `'tcp'` or `'udp'`, otherwise any protocol will match.

Raises an auditing event `socket.getservbyport` with arguments `port`, `protocolname`.

Availability: not WASI.

socket.ntohl(*x*)

Convert 32-bit positive integers from network to host byte order. On machines where the host byte order is the same as network byte order, this is a no-op; otherwise, it performs a 4-byte swap operation.

socket.ntohs(*x*)

Convert 16-bit positive integers from network to host byte order. On machines where the host byte order is the same as network byte order, this is a no-op; otherwise, it performs a 2-byte swap operation.

Changed in version 3.10: Raises **`OverflowError`** if *x* does not fit in a 16-bit unsigned integer.

socket.htonl(*x*)

Convert 32-bit positive integers from host to network byte order. On machines where the host byte order is the same as network byte order, this is a no-op; otherwise, it performs a 4-byte swap operation.

socket.htons(*x*)

Convert 16-bit positive integers from host to network byte order. On machines where the host byte order is the same as network byte order, this is a no-op; otherwise, it performs a 2-byte swap operation.

Changed in version 3.10: Raises `OverflowError` if *x* does not fit in a 16-bit unsigned integer.

socket.inet_aton(*ip_string*)

Convert an IPv4 address from dotted-quad string format (for example, ‘123.45.67.89’) to 32-bit packed binary format, as a bytes object four characters in length. This is useful when conversing with a program that uses the standard C library and needs objects of type `in_addr`, which is the C type for the 32-bit packed binary this function returns.

`inet_aton()` also accepts strings with less than three dots; see the Unix manual page *inet(3)* *[https://manpages.debian.org/inet(3)]* for details.

If the IPv4 address string passed to this function is invalid, `OSError` will be raised. Note that exactly what is valid depends on the underlying C implementation of `inet_aton()`.

`inet_aton()` does not support IPv6, and `inet_pton()` should be used instead for IPv4/v6 dual stack support.

socket.inet_ntoa(*packed_ip*)

Convert a 32-bit packed IPv4 address (a bytes-like object four bytes in length) to its standard dotted-quad string representation (for example, ‘123.45.67.89’). This is useful when conversing with a program that uses the standard C library and needs objects of type `in_addr`, which is the C type for the 32-bit packed binary data this function takes as an argument.

If the byte sequence passed to this function is not exactly 4 bytes in length, **`OSError`** will be raised. **`inet_ntoa()`** does not support IPv6, and **`inet_ntop()`** should be used instead for IPv4/v6 dual stack support.

Changed in version 3.5: Writable bytes-like object is now accepted.

socket.inet_pton(*address_family*, *ip_string*)

Convert an IP address from its family-specific string format to a packed, binary format. **`inet_pton()`** is useful when a library or network protocol calls for an object of type **`in_addr`** (similar to **`inet_aton()`**) or **`in6_addr`**.

Supported values for *address_family* are currently **`AF_INET`** and **`AF_INET6`**. If the IP address string *ip_string* is invalid, **`OSError`** will be raised. Note that exactly what is valid depends on both the value of *address_family* and the underlying implementation of **`inet_pton()`**.

Availability: Unix, Windows.

Changed in version 3.4: Windows support added

socket.inet_ntop(*address_family*, *packed_ip*)

Convert a packed IP address (a bytes-like object of some number of bytes) to its standard, family-specific string representation (for example, `'7.10.0.5'` or `'5aef:2b::8'`). **`inet_ntop()`** is useful when a library or network protocol returns an object of type **`in_addr`** (similar to **`inet_ntoa()`**) or **`in6_addr`**.

Supported values for *address_family* are currently `AF_INET` and `AF_INET6`. If the bytes object *packed_ip* is not the correct length for the specified address family, `ValueError` will be raised. `OSError` is raised for errors from the call to `inet_ntop()`.

Availability: Unix, Windows.

Changed in version 3.4: Windows support added

Changed in version 3.5: Writable bytes-like object is now accepted.

socket.CMSG_LEN(*length*)

Return the total length, without trailing padding, of an ancillary data item with associated data of the given *length*. This value can often be used as the buffer size for `recvmsg()` to receive a single item of ancillary data, but **RFC 3542** [https://datatracker.ietf.org/doc/html/rfc3542.html] requires portable applications to use `CMSG_SPACE()` and thus include space for padding, even when the item will be the last in the buffer. Raises `OverflowError` if *length* is outside the permissible range of values.

Availability: Unix, not WASI.

Most Unix platforms.

Added in version 3.3.

socket.CMSG_SPACE(*length*)

Return the buffer size needed for `recvmsg()` to receive an ancillary data item with associated data of the given *length*, along with any trailing padding. The buffer space needed to receive multiple items is the sum of the `CMSG_SPACE()` values for their associated data lengths.

Raises **`OverflowError`** if *length* is outside the permissible range of values.

Note that some systems might support ancillary data without providing this function. Also note that setting the buffer size using the results of this function may not precisely limit the amount of ancillary data that can be received, since additional data may be able to fit into the padding area.

Availability: Unix, not WASI.

most Unix platforms.

Added in version 3.3.

socket.getdefaulttimeout()

Return the default timeout in seconds (float) for new socket objects. A value of `None` indicates that new socket objects have no timeout. When the socket module is first imported, the default is `None`.

socket.setdefaulttimeout(*timeout*)

Set the default timeout in seconds (float) for new socket objects. When the socket module is first imported, the default is `None`. See **`settimeout()`** for possible values and their respective meanings.

socket.sethostname(*name*)

Set the machine’s hostname to *name*. This will raise an **`OSError`** if you don’t have enough rights.

Raises an auditing event `socket.sethostname` with argument `name`.

Availability: Unix, not Android.

Added in version 3.3.

socket.if_nameindex()

Return a list of network interface information (index int, name string) tuples. `OSError` if the system call fails.

Availability: Unix, Windows, not WASI.

Added in version 3.3.

Changed in version 3.8: Windows support was added.

Note

On Windows network interfaces have different names in different contexts (all names are examples):

- UUID: `{FB605B73-AAC2-49A6-9A2F-25416AEA0573}`
- name: `ethernet_32770`
- friendly name: `vEthernet (nat)`
- description: `Hyper-V Virtual Ethernet Adapter`

This function returns names of the second form from the list, `ethernet_32770` in this example case.

socket.if_nametoindex(*if_name*)

Return a network interface index number corresponding to an interface name. `OSError` if no interface with the given name exists.

Availability: Unix, Windows, not WASI.

Added in version 3.3.

Changed in version 3.8: Windows support was added.

See also

“Interface name” is a name as documented in `if_nameindex()`.

socket.if_indextoname(*if_index*)

Return a network interface name corresponding to an interface index number. `OSError` if no interface with the given index exists.

Availability: Unix, Windows, not WASI.

Added in version 3.3.

Changed in version 3.8: Windows support was added.

See also

“Interface name” is a name as documented in `if_nameindex()`.

socket.send_fds(*sock*, *buffers*, *fds*[, *flags*[, *address*]])

Send the list of file descriptors *fds* over an `AF_UNIX` socket *sock*. The *fds* parameter is a sequence of file descriptors. Consult `sendmsg()` for the documentation of these parameters.

Availability: Unix, not WASI.

Unix platforms supporting `sendmsg()` and **`SCM_RIGHTS`** mechanism.

Added in version 3.9.

socket.recv_fds(*sock*, *bufsize*, *maxfds*[, *flags*])

Receive up to *maxfds* file descriptors from an `AF_UNIX` socket *sock*. Return `(msg, list(fds), flags, addr)`. Consult `recvmsg()` for the documentation of these parameters.

Availability: Unix, not WASI.

Unix platforms supporting `recvmsg()` and **`SCM_RIGHTS`** mechanism.

Added in version 3.9.

Note

Any truncated integers at the end of the list of file descriptors.

Socket Objects

Socket objects have the following methods. Except for `makefile()`, these correspond to Unix system calls applicable to sockets.

Changed in version 3.2: Support for the context manager protocol was added. Exiting the context manager is equivalent to calling `close()`.

socket.accept()

Accept a connection. The socket must be bound to an address and listening for connections. The return value is a pair `(conn, address)` where *conn* is a *new* socket object usable to send and receive data on the connection, and *address* is the address bound to the socket on the other end of the connection.

The newly created socket is non-inheritable.

Changed in version 3.4: The socket is now non-inheritable.

Changed in version 3.5: If the system call is interrupted and the signal handler does not raise an exception, the method now retries the system call instead of raising an **`InterruptedError`** exception (see **PEP 475** [https://peps.python.org/pep-0475/] for the rationale).

socket.bind(*address*)

Bind the socket to *address*. The socket must not already be bound. (The format of *address* depends on the address family — see above.)

Raises an auditing event `socket.bind` with arguments `self`, `address`.

Availability: not WASI.

socket.close()

Mark the socket closed. The underlying system resource (e.g. a file descriptor) is also closed when all file objects from **`makefile()`** are closed. Once that happens, all future operations on the socket object will fail. The remote end will receive no more data (after queued data is flushed).

Sockets are automatically closed when they are garbage-collected, but it is recommended to **`close()`** them explicitly, or to use a **`with`** statement around them.

Changed in version 3.6: **`OSError`** is now raised if an error occurs when the underlying **`close()`** call is made.

Note

`close()` releases the resource associated with a connection but does not necessarily close the connection immediately. If you want to close the connection in a timely fashion, call `shutdown()` before `close()`.

socket.connect(*address*)

Connect to a remote socket at *address*. (The format of *address* depends on the address family — see above.)

If the connection is interrupted by a signal, the method waits until the connection completes, or raise a `TimeoutError` on timeout, if the signal handler doesn't raise an exception and the socket is blocking or has a timeout. For non-blocking sockets, the method raises an `InterruptedError` exception if the connection is interrupted by a signal (or the exception raised by the signal handler).

Raises an auditing event `socket.connect` with arguments `self`, `address`.

Changed in version 3.5: The method now waits until the connection completes instead of raising an `InterruptedError` exception if the connection is interrupted by a signal, the signal handler doesn't raise an exception and the socket is blocking or has a timeout (see the **PEP 475** [https://peps.python.org/pep-0475/] for the rationale).

Availability: not WASI.

socket.connect_ex(*address*)

Like `connect(address)`, but return an error indicator instead of raising an exception for errors returned by the C-level **`connect()`** call (other problems, such as "host not found," can still raise exceptions). The error indicator is `0` if the operation succeeded, otherwise the value of the **`errno`** variable. This is useful to support, for example, asynchronous connects.

Raises an auditing event `socket.connect` with arguments `self`, `address`.

Availability: not WASI.

socket.detach()

Put the socket object into closed state without actually closing the underlying file descriptor. The file descriptor is returned, and can be reused for other purposes.

Added in version 3.2.

socket.dup()

Duplicate the socket.

The newly created socket is non-inheritable.

Changed in version 3.4: The socket is now non-inheritable.

Availability: not WASI.

socket.fileno()

Return the socket’s file descriptor (a small integer), or -1 on failure. This is useful with **`select.select()`**.

Under Windows the small integer returned by this method cannot be used where a file descriptor can be used (such as **`os.fdopen()`**). Unix does not have this limitation.

socket.get_inheritable()

Get the inheritable flag of the socket’s file descriptor or socket’s handle: `True` if the socket can be inherited in child processes, `False` if it cannot.

Added in version 3.4.

socket.getpeername()

Return the remote address to which the socket is connected. This is useful to find out the port number of a remote IPv4/v6 socket, for instance. (The format of the address returned depends on the address family — see above.) On some systems this function is not supported.

socket.getsockname()

Return the socket’s own address. This is useful to find out the port number of an IPv4/v6 socket, for instance. (The format of the address returned depends on the address family — see above.)

socket.getsockopt(*level*, *optname*[, *buflen*])

Return the value of the given socket option (see the Unix man page *getsockopt(2)* *[https://manpages.debian.org/getsockopt(2)]*). The needed symbolic constants (SO_* etc.) are defined in this module. If *buflen* is absent, an integer option is assumed and its integer value is returned by the function. If *buflen* is present, it specifies the maximum length of the buffer used to receive the option in, and this buffer is returned as a bytes object. It is up to the caller to decode the contents of the buffer (see the optional built-in module **`struct`** for a way to decode C structures encoded as byte strings).

Availability: not WASI.

socket.getblocking()

Return `True` if socket is in blocking mode, `False` if in non-blocking.

This is equivalent to checking `socket.gettimeout() != 0`.

Added in version 3.7.

socket.gettimeout()

Return the timeout in seconds (float) associated with socket operations, or `None` if no timeout is set. This reflects the last call to **`setblocking()`** or **`settimeout()`**.

socket.ioctl(*control*, *option*)

The **`ioctl()`** method is a limited interface to the WSAIoctl system interface. Please refer to the Win32 documentation [https://msdn.microsoft.com/en-us/library/ms741621%28VS.85%29.aspx] for more information.

On other platforms, the generic **`fcntl.fcntl()`** and **`fcntl.ioctl()`** functions may be used; they accept a socket object as their first argument.

Currently only the following control codes are supported: `SIO_RCVALL`, `SIO_KEEPALIVE_VALS`, and `SIO_LOOPBACK_FAST_PATH`.

Availability: Windows

Changed in version 3.6: `SIO_LOOPBACK_FAST_PATH` was added.

socket.listen([*backlog*])

Enable a server to accept connections. If *backlog* is specified, it must be at least 0 (if it is lower, it is set to 0); it specifies the number of unaccepted connections that the system will allow before refusing new connections. If not specified, a default reasonable value is chosen.

Availability: not WASI.

Changed in version 3.5: The *backlog* parameter is now optional.

socket.makefile(*mode='r'*, *buffering=None*, ***, *encoding=None*, *errors=None*, *newline=None*)

Return a file object associated with the socket. The exact returned type depends on the arguments given to **`makefile()`**. These arguments are interpreted the same way as by the built-in **`open()`** function, except the only supported *mode* values are `'r'` (default), `'w'`, `'b'`, or a combination of those.

The socket must be in blocking mode; it can have a timeout, but the file object's internal buffer may end up in an inconsistent state if a timeout occurs.

Closing the file object returned by **`makefile()`** won't close the original socket unless all other file objects have been closed and **`socket.close()`** has been called on the socket object.

Note

On Windows, the file-like object created by **`makefile()`** cannot be used where a file object with a file descriptor is expected, such as the stream arguments of **`subprocess.Popen()`**.

socket.recv(*bufsize*[, *flags*])

Receive data from the socket. The return value is a bytes object representing the data received. The maximum amount of data to be received at once is specified by *bufsize*. A returned empty bytes object indicates that the client has disconnected. See the Unix manual page *recv(2)* *[https://manpages.debian.org/recv(2)]* for the meaning of the optional argument *flags*; it defaults to zero.

Changed in version 3.5: If the system call is interrupted and the signal handler does not raise an exception, the method now retries the system call instead of raising an **`InterruptedError`** exception (see **PEP 475** [https://peps.python.org/pep-0475/] for the rationale).

socket.recvfrom(*bufsize*[, *flags*])

Receive data from the socket. The return value is a pair `(bytes, address)` where *bytes* is a bytes object representing the data received and *address* is the address of the socket sending the data. See the Unix manual page *recv(2)* *[https://manpages.debian.org/recv(2)]* for the meaning of the optional argument *flags*; it defaults to zero. (The format of *address* depends on the address family — see above.)

Changed in version 3.5: If the system call is interrupted and the signal handler does not raise an exception, the method now retries the system call instead of raising an **`InterruptedError`** exception (see **PEP 475** [https://peps.python.org/pep-0475/] for the rationale).

Changed in version 3.7: For multicast IPv6 address, first item of *address* does not contain `%scope_id` part anymore. In order to get full IPv6 address use **`getnameinfo()`**.

socket.recvmsg(*bufsize*[, *ancbufsize*[, *flags*]])

Receive normal data (up to *bufsize* bytes) and ancillary data from the socket. The *ancbufsize* argument sets the size in bytes of the internal buffer used to receive the ancillary data; it defaults to 0, meaning that no ancillary data will be received. Appropriate buffer sizes for ancillary data can be calculated using `CMSG_SPACE()` or `CMSG_LEN()`, and items which do not fit into the buffer might be truncated or discarded. The *flags* argument defaults to 0 and has the same meaning as for `recv()`.

The return value is a 4-tuple: `(data, ancdata, msg_flags, address)`. The *data* item is a `bytes` object holding the non-ancillary data received. The *ancdata* item is a list of zero or more tuples `(cmsg_level, cmsg_type, cmsg_data)` representing the ancillary data (control messages) received: *cmsg_level* and *cmsg_type* are integers specifying the protocol level and protocol-specific type respectively, and *cmsg_data* is a `bytes` object holding the associated data. The *msg_flags* item is the bitwise OR of various flags indicating conditions on the received message; see your system documentation for details. If the receiving socket is unconnected, *address* is the address of the sending socket, if available; otherwise, its value is unspecified.

On some systems, `sendmsg()` and `recvmsg()` can be used to pass file descriptors between processes over an `AF_UNIX` socket. When this facility is used (it is often restricted to `SOCK_STREAM` sockets), `recvmsg()` will return, in its ancillary data, items of the form `(socket.SOL_SOCKET, socket.SCM_RIGHTS, fds)`, where *fds* is a `bytes` object representing the new file descriptors as a binary array of the native C int type. If `recvmsg()` raises an exception after the system call returns, it will first attempt to close any file descriptors received via this mechanism.

Some systems do not indicate the truncated length of ancillary data items which have been only partially received. If an item appears to extend beyond the end of the buffer, `recvmsg()` will issue a `RuntimeWarning`, and will return the part of it which is inside the buffer provided it has not been truncated before the start of its associated data.

On systems which support the **`SCM_RIGHTS`** mechanism, the following function will receive up to *maxfds* file descriptors, returning the message data and a list containing the descriptors (while ignoring unexpected conditions such as unrelated control messages being received). See also `sendmsg()`.

```
import socket, array

def recv_fds(sock, msglen, maxfds):
    fds = array.array("i")   # Array of ints
    msg, ancdata, flags, addr = sock.recvmsg(msglen,
socket.CMSG_LEN(maxfds * fds.itemsize))
    for cmsg_level, cmsg_type, cmsg_data in ancdata:
        if cmsg_level == socket.SOL_SOCKET and cmsg_type ==
socket.SCM_RIGHTS:
            # Append data, ignoring any truncated integers
at the end.
            fds.frombytes(cmsg_data[:len(cmsg_data) -
(len(cmsg_data) % fds.itemsize)])
    return msg, list(fds)
```

Availability: Unix.

Most Unix platforms.

Added in version 3.3.

Changed in version 3.5: If the system call is interrupted and the signal handler does not raise an exception, the method now retries the system call instead of raising an `InterruptedError` exception (see **PEP 475** [https://peps.python.org/pep-0475/] for the rationale).

socket.recvmsg_into(*buffers*[, *ancbufsize*[, *flags*]])

Receive normal data and ancillary data from the socket, behaving as `recvmsg()` would, but scatter the non-ancillary data into a series of buffers instead of returning a new bytes object. The *buffers* argument must be an iterable of objects that export writable buffers (e.g. `bytearray` objects); these will be filled with successive chunks of the non-ancillary data until it has all been written or there are no more buffers. The operating system may set a limit (`sysconf()` value `SC_IOV_MAX`) on the number of buffers that can be used. The *ancbufsize* and *flags* arguments have the same meaning as for `recvmsg()`.

The return value is a 4-tuple: `(nbytes, ancdata, msg_flags, address)`, where *nbytes* is the total number of bytes of non-ancillary data written into the buffers, and *ancdata*, *msg_flags* and *address* are the same as for `recvmsg()`.

Example:

```
>>> import socket
>>> s1, s2 = socket.socketpair()
>>> b1 = bytearray(b'----')
>>> b2 = bytearray(b'0123456789')
>>> b3 = bytearray(b'--------------')
>>> s1.send(b'Mary had a little lamb')
22
>>> s2.recvmsg_into([b1, memoryview(b2)[2:9], b3])
(22, [], 0, None)
>>> [b1, b2, b3]
[bytearray(b'Mary'), bytearray(b'01 had a 9'),
bytearray(b'little lamb---')]
```

Availability: Unix.

Most Unix platforms.

Added in version 3.3.

socket.recvfrom_into(*buffer*[, *nbytes*[, *flags*]])

Receive data from the socket, writing it into *buffer* instead of creating a new bytestring. The return value is a pair `(nbytes, address)` where *nbytes* is the number of bytes received and *address* is the address of the socket sending the data. See the Unix manual page *recv(2)* *[https://manpages.debian.org/recv(2)]* for the meaning of the optional argument *flags*; it defaults to zero. (The format of *address* depends on the address family — see above.)

socket.recv_into(*buffer*[, *nbytes*[, *flags*]])

Receive up to *nbytes* bytes from the socket, storing the data into a buffer rather than creating a new bytestring. If *nbytes* is not specified (or 0), receive up to the size available in the given buffer. Returns the number of bytes received. See the Unix manual page *recv(2)* *[https://manpages.debian.org/recv(2)]* for the meaning of the optional argument *flags*; it defaults to zero.

socket.send(*bytes*[, *flags*])

Send data to the socket. The socket must be connected to a remote socket. The optional *flags* argument has the same meaning as for **`recv()`** above. Returns the number of bytes sent. Applications are responsible for checking that all data has been sent; if only some of the data was transmitted, the application needs to attempt delivery of the remaining data. For further information on this topic, consult the Socket Programming HOWTO.

Changed in version 3.5: If the system call is interrupted and the signal handler does not raise an exception, the method now retries the system call instead of raising an **`InterruptedError`** exception (see **PEP 475** [https://peps.python.org/pep-0475/] for the rationale).

socket.sendall(*bytes*[, *flags*])

Send data to the socket. The socket must be connected to a remote socket. The optional *flags* argument has the same meaning as for **`recv()`** above. Unlike **`send()`**, this method continues to send data from *bytes* until either all data has been sent or an error occurs. `None` is returned on success. On error, an exception is raised, and there is no way to determine how much data, if any, was successfully sent.

Changed in version 3.5: The socket timeout is no longer reset each time data is sent successfully. The socket timeout is now the maximum total duration to send all data.

Changed in version 3.5: If the system call is interrupted and the signal handler does not raise an exception, the method now retries the system call instead of raising an **`InterruptedError`** exception (see **PEP 475** [https://peps.python.org/pep-0475/] for the rationale).

socket.sendto(*bytes*, *address*)
socket.sendto(*bytes*, *flags*, *address*)

Send data to the socket. The socket should not be connected to a remote socket, since the destination socket is specified by *address*. The optional *flags* argument has the same meaning as for **`recv()`** above. Return the number of bytes sent. (The format of *address* depends on the address family — see above.)

Raises an auditing event `socket.sendto` with arguments `self`, `address`.

Changed in version 3.5: If the system call is interrupted and the signal handler does not raise an exception, the method now retries the system call instead of raising an `InterruptedError` exception (see **PEP 475** [https://peps.python.org/pep-0475/] for the rationale).

socket.sendmsg(*buffers*[, *ancdata*[, *flags*[, *address*]]])

Send normal and ancillary data to the socket, gathering the non-ancillary data from a series of buffers and concatenating it into a single message. The *buffers* argument specifies the non-ancillary data as an iterable of bytes-like objects (e.g. `bytes` objects); the operating system may set a limit (`sysconf()` value `SC_IOV_MAX`) on the number of buffers that can be used. The *ancdata* argument specifies the ancillary data (control messages) as an iterable of zero or more tuples `(cmsg_level, cmsg_type, cmsg_data)`, where *cmsg_level* and *cmsg_type* are integers specifying the protocol level and protocol-specific type respectively, and *cmsg_data* is a bytes-like object holding the associated data. Note that some systems (in particular, systems without `CMSG_SPACE()`) might support sending only one control message per call. The *flags* argument defaults to 0 and has the same meaning as for `send()`. If *address* is supplied and not `None`, it sets a destination address for the message. The return value is the number of bytes of non-ancillary data sent.

The following function sends the list of file descriptors *fds* over an `AF_UNIX` socket, on systems which support the **`SCM_RIGHTS`** mechanism. See also `recvmsg()`.

```
import socket, array

def send_fds(sock, msg, fds):
    return sock.sendmsg([msg], [(socket.SOL_SOCKET,
socket.SCM_RIGHTS, array.array("i", fds))])
```

Availability: Unix, not WASI.

Most Unix platforms.
Raises an auditing event `socket.sendmsg` with arguments `self`, `address`.

Added in version 3.3.

Changed in version 3.5: If the system call is interrupted and the signal handler does not raise an exception, the method now retries the system call instead of raising an **`InterruptedError`** exception (see **PEP 475** [https://peps.python.org/pep-0475/] for the rationale).

socket.sendmsg_afalg([*msg*,]*, *op*[, *iv*[, *assoclen*[, *flags*]]])

Specialized version of **`sendmsg()`** for **`AF_ALG`** socket. Set mode, IV, AEAD associated data length and flags for **`AF_ALG`** socket.

Availability: Linux >= 2.6.38.

Added in version 3.6.

socket.sendfile(*file*, *offset=0*, *count=None*)

Send a file until EOF is reached by using high-performance **`os.sendfile`** and return the total number of bytes which were sent. *file* must be a regular file object opened in binary mode. If **`os.sendfile`** is not available (e.g. Windows) or *file* is not a regular file **`send()`** will be used instead. *offset* tells from where to start reading the file. If specified, *count* is the total number of bytes to transmit as opposed to sending the file until EOF is reached. File position is updated on return or also in case of error in which case **`file.tell()`** can be used to figure out the number of bytes which were sent. The socket must be of **`SOCK_STREAM`** type. Non-blocking sockets are not supported.

Added in version 3.5.

socket.set_inheritable(*inheritable*)

Set the inheritable flag of the socket’s file descriptor or socket’s handle.

Added in version 3.4.

socket.setblocking(*flag*)

Set blocking or non-blocking mode of the socket: if *flag* is false, the socket is set to non-blocking, else to blocking mode.

This method is a shorthand for certain **`settimeout()`** calls:

- `sock.setblocking(True)` is equivalent to `sock.settimeout(None)`
- `sock.setblocking(False)` is equivalent to `sock.settimeout(0.0)`

Changed in version 3.7: The method no longer applies **`SOCK_NONBLOCK`** flag on **`socket.type`**.

socket.settimeout(*value*)

Set a timeout on blocking socket operations. The *value* argument can be a nonnegative floating-point number expressing seconds, or `None`. If a non-zero value is given, subsequent socket operations will raise a **`timeout`** exception if the timeout period *value* has elapsed before the operation has completed. If zero is given, the socket is put in non-blocking mode. If `None` is given, the socket is put in blocking mode.

For further information, please consult the notes on socket timeouts.

Changed in version 3.7: The method no longer toggles **`SOCK_NONBLOCK`** flag on **`socket.type`**.

socket.setsockopt(*level*, *optname*, *value: int* | *Buffer*)
socket.setsockopt(*level*, *optname*, *None*, *optlen: int*)

Set the value of the given socket option (see the Unix manual page *setsockopt(2)* *[https://manpages.debian.org/setsockopt(2)]*). The needed symbolic constants are defined in this module (SO_* etc. <socket-unix-constants>). The value can be an integer, `None` or a bytes-like object representing a buffer. In the later case it is up to the caller to ensure that the bytestring contains the proper bits (see the optional built-in module **`struct`** for a way to encode C structures as bytestrings). When *value* is set to `None`, *optlen* argument is required. It's equivalent to call **`setsockopt()`** C function with `optval=NULL` and `optlen=optlen`.

Changed in version 3.5: Writable bytes-like object is now accepted.

Changed in version 3.6: setsockopt(level, optname, None, optlen: int) form added.

Availability: not WASI.

socket.shutdown(*how*)

Shut down one or both halves of the connection. If *how* is **`SHUT_RD`**, further receives are disallowed. If *how* is **`SHUT_WR`**, further sends are disallowed. If *how* is **`SHUT_RDWR`**, further sends and receives are disallowed.

Availability: not WASI.

socket.share(*process_id*)

Duplicate a socket and prepare it for sharing with a target process. The target process must be provided with *process_id*. The resulting bytes object can then be passed to the target process using some form of interprocess communication and the socket can be recreated there using `fromshare()`. Once this method has been called, it is safe to close the socket since the operating system has already duplicated it for the target process.

Availability: Windows.

Added in version 3.3.

Note that there are no methods **`read()`** or **`write()`**; use `recv()` and `send()` without *flags* argument instead.

Socket objects also have these (read-only) attributes that correspond to the values given to the `socket` constructor.

socket.family

The socket family.

socket.type

The socket type.

socket.proto

The socket protocol.

Notes on socket timeouts

A socket object can be in one of three modes: blocking, non-blocking, or timeout. Sockets are by default always created in blocking mode, but this can be changed by calling `setdefaulttimeout()`.

- In *blocking mode*, operations block until complete or the system returns an error (such as connection timed out).
- In *non-blocking mode*, operations fail (with an error that is unfortunately system-dependent) if they cannot be completed immediately: functions from the `select` module can be used to know when and whether a socket is available for reading or writing.
- In *timeout mode*, operations fail if they cannot be completed within the timeout specified for the socket (they raise a `timeout` exception) or if the system returns an error.

Note

At the operating system level, sockets in *timeout mode* are internally set in non-blocking mode. Also, the blocking and timeout modes are shared between file descriptors and socket objects that refer to the same network endpoint. This implementation detail can have visible consequences if e.g. you decide to use the `fileno()` of a socket.

Timeouts and the `connect` method

The `connect()` operation is also subject to the timeout setting, and in general it is recommended to call `settimeout()` before calling `connect()` or pass a timeout parameter to `create_connection()`. However, the system network stack may also return a connection timeout error of its own regardless of any Python socket timeout setting.

Timeouts and the `accept` method

If `getdefaulttimeout()` is not `None`, sockets returned by the `accept()` method inherit that timeout. Otherwise, the behaviour depends on settings of the listening socket:

- if the listening socket is in *blocking mode* or in *timeout mode*, the socket returned by `accept()` is in *blocking mode*;
- if the listening socket is in *non-blocking mode*, whether the socket returned by `accept()` is in blocking or non-blocking mode is operating system-dependent. If you want to ensure cross-platform behaviour, it is recommended you manually override this setting.

Example

Here are four minimal example programs using the TCP/IP protocol: a server that echoes all data that it receives back (servicing only one client), and a client using it. Note that a server must perform the sequence `socket()`, `bind()`, `listen()`, `accept()` (possibly repeating the `accept()` to service more than one client), while a client only needs the sequence `socket()`, `connect()`. Also note that the server does not `sendall()`/`recv()` on the socket it is listening on but on the new socket returned by `accept()`.

The first two examples support IPv4 only.

```
# Echo server program
import socket

HOST = ''                 # Symbolic name meaning all available
interfaces
PORT = 50007              # Arbitrary non-privileged port
with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as s:
    s.bind((HOST, PORT))
    s.listen(1)
    conn, addr = s.accept()
    with conn:
        print('Connected by', addr)
        while True:
            data = conn.recv(1024)
            if not data: break
            conn.sendall(data)
```

```
# Echo client program
import socket

HOST = 'daring.cwi.nl'    # The remote host
PORT = 50007              # The same port as used by the server
with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as s:
    s.connect((HOST, PORT))
    s.sendall(b'Hello, world')
    data = s.recv(1024)
print('Received', repr(data))
```

The next two examples are identical to the above two, but support both IPv4 and IPv6. The server side will listen to the first address family available (it should listen to both instead). On most of IPv6-ready systems, IPv6 will take precedence and the server may not accept IPv4 traffic. The client side will try to connect to all the addresses returned as a result of the name resolution, and sends traffic to the first one connected successfully.

```
# Echo server program
import socket
import sys

HOST = None               # Symbolic name meaning all available
interfaces
PORT = 50007              # Arbitrary non-privileged port
s = None
for res in socket.getaddrinfo(HOST, PORT, socket.AF_UNSPEC,
                              socket.SOCK_STREAM, 0,
socket.AI_PASSIVE):
    af, socktype, proto, canonname, sa = res
    try:
        s = socket.socket(af, socktype, proto)
    except OSError as msg:
        s = None
        continue
    try:
        s.bind(sa)
        s.listen(1)
    except OSError as msg:
        s.close()
        s = None
        continue
    break
if s is None:
    print('could not open socket')
    sys.exit(1)
conn, addr = s.accept()
with conn:
    print('Connected by', addr)
    while True:
        data = conn.recv(1024)
        if not data: break
        conn.send(data)
```

```
# Echo client program
import socket
import sys

HOST = 'daring.cwi.nl'    # The remote host
PORT = 50007              # The same port as used by the server
s = None
for res in socket.getaddrinfo(HOST, PORT, socket.AF_UNSPEC,
socket.SOCK_STREAM):
    af, socktype, proto, canonname, sa = res
    try:
        s = socket.socket(af, socktype, proto)
    except OSError as msg:
        s = None
        continue
    try:
        s.connect(sa)
    except OSError as msg:
        s.close()
        s = None
        continue
    break
if s is None:
    print('could not open socket')
    sys.exit(1)
with s:
    s.sendall(b'Hello, world')
    data = s.recv(1024)
print('Received', repr(data))
```

The next example shows how to write a very simple network sniffer with raw sockets on Windows. The example requires administrator privileges to modify the interface:

```
import socket

# the public network interface
HOST = socket.gethostbyname(socket.gethostname())

# create a raw socket and bind it to the public interface
s = socket.socket(socket.AF_INET, socket.SOCK_RAW,
socket.IPPROTO_IP)
s.bind((HOST, 0))

# Include IP headers
s.setsockopt(socket.IPPROTO_IP, socket.IP_HDRINCL, 1)

# receive all packets
s.ioctl(socket.SIO_RCVALL, socket.RCVALL_ON)

# receive a packet
print(s.recvfrom(65565))

# disabled promiscuous mode
s.ioctl(socket.SIO_RCVALL, socket.RCVALL_OFF)
```

The next example shows how to use the socket interface to communicate to a CAN network using the raw socket protocol. To use CAN with the broadcast manager protocol instead, open a socket with:

```
socket.socket(socket.AF_CAN, socket.SOCK_DGRAM, socket.CAN_BCM)
```

After binding (**`CAN_RAW`**) or connecting (**`CAN_BCM`**) the socket, you can use the **`socket.send()`** and **`socket.recv()`** operations (and their counterparts) on the socket object as usual.

This last example might require special privileges:

```
import socket
import struct


# CAN frame packing/unpacking (see 'struct can_frame' in
<linux/can.h>)

can_frame_fmt = "=IB3x8s"
can_frame_size = struct.calcsize(can_frame_fmt)

def build_can_frame(can_id, data):
    can_dlc = len(data)
    data = data.ljust(8, b'\x00')
    return struct.pack(can_frame_fmt, can_id, can_dlc, data)

def dissect_can_frame(frame):
    can_id, can_dlc, data = struct.unpack(can_frame_fmt, frame)
    return (can_id, can_dlc, data[:can_dlc])


# create a raw socket and bind it to the 'vcan0' interface
s = socket.socket(socket.AF_CAN, socket.SOCK_RAW,
socket.CAN_RAW)
s.bind(('vcan0',))

while True:
    cf, addr = s.recvfrom(can_frame_size)

    print('Received: can_id=%x, can_dlc=%x, data=%s' %
dissect_can_frame(cf))

    try:
        s.send(cf)
    except OSError:
        print('Error sending CAN frame')

    try:
        s.send(build_can_frame(0x01, b'\x01\x02\x03'))
    except OSError:
        print('Error sending CAN frame')
```

Running an example several times with too small delay between executions, could lead to this error:

```
OSError: [Errno 98] Address already in use
```

This is because the previous execution has left the socket in a `TIME_WAIT` state, and can’t be immediately reused.

There is a `socket` flag to set, in order to prevent this, **`socket.SO_REUSEADDR`**:

```
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
s.bind((HOST, PORT))
```

the **`SO_REUSEADDR`** flag tells the kernel to reuse a local socket in `TIME_WAIT` state, without waiting for its natural timeout to expire.

See also

For an introduction to socket programming (in C), see the following papers:

- *An Introductory 4.3BSD Interprocess Communication Tutorial*, by Stuart Sechrest
- *An Advanced 4.3BSD Interprocess Communication Tutorial*, by Samuel J. Leffler et al,

both in the UNIX Programmer’s Manual, Supplementary Documents 1 (sections PS1:7 and PS1:8). The platform-specific reference material for the various socket-related system calls are also a valuable source of information on the details of socket semantics. For Unix, refer to the manual pages; for Windows, see the WinSock (or Winsock

2) specification. For IPv6-ready APIs, readers may want to refer to **RFC 3493** [https://datatracker.ietf.org/doc/html/rfc3493.html] titled Basic Socket Interface Extensions for IPv6.

`ssl` — TLS/SSL wrapper for socket objects

Source code: Lib/ssl.py
[https://github.com/python/cpython/tree/3.14/Lib/ssl.py]

This module provides access to Transport Layer Security (often known as “Secure Sockets Layer”) encryption and peer authentication facilities for network sockets, both client-side and server-side. This module uses the OpenSSL library.

This is an optional module. If it is missing from your copy of CPython, look for documentation from your distributor (that is, whoever provided Python to you). If you are the distributor, see Requirements for optional modules.

Note

Some behavior may be platform dependent, since calls are made to the operating system socket APIs. The installed version of OpenSSL may also cause variations in behavior. For example, TLSv1.3 comes with OpenSSL version 1.1.1.

Warning

Don’t use this module without reading the Security considerations. Doing so may lead to a false sense of security, as the default settings of the ssl module are not necessarily appropriate for your application.

Availability: not WASI.

This module does not work or is not available on WebAssembly. See WebAssembly platforms for more information.

This section documents the objects and functions in the `ssl` module; for more general information about TLS, SSL, and certificates, the reader is referred to the documents in the “See Also” section at the bottom.

This module provides a class, **`ssl.SSLSocket`**, which is derived from the **`socket.socket`** type, and provides a socket-like wrapper that also encrypts and decrypts the data going over the socket with SSL. It supports additional methods such as **`getpeercert()`**, which retrieves the certificate of the other side of the connection, **`cipher()`**, which retrieves the cipher being used for the secure connection or **`get_verified_chain()`**, **`get_unverified_chain()`** which retrieves certificate chain.

For more sophisticated applications, the **`ssl.SSLContext`** class helps manage settings and certificates, which can then be inherited by SSL sockets created through the **`SSLContext.wrap_socket()`** method.

Changed in version 3.5.3: Updated to support linking with OpenSSL 1.1.0

Changed in version 3.6: OpenSSL 0.9.8, 1.0.0 and 1.0.1 are deprecated and no longer supported. In the future the ssl module will require at least OpenSSL 1.0.2 or 1.1.0.

Changed in version 3.10: **PEP 644** [https://peps.python.org/pep-0644/] has been implemented. The ssl module requires OpenSSL 1.1.1 or newer.

Use of deprecated constants and functions result in deprecation warnings.

Functions, Constants, and Exceptions

Socket creation

Instances of `SSLSocket` must be created using the `SSLContext.wrap_socket()` method. The helper function `create_default_context()` returns a new context with secure default settings.

Client socket example with default context and IPv4/IPv6 dual stack:

```
import socket
import ssl

hostname = 'www.python.org'
context = ssl.create_default_context()

with socket.create_connection((hostname, 443)) as sock:
    with context.wrap_socket(sock, server_hostname=hostname) as
ssock:
        print(ssock.version())
```

Client socket example with custom context and IPv4:

```
hostname = 'www.python.org'
# PROTOCOL_TLS_CLIENT requires valid cert chain and hostname
context = ssl.SSLContext(ssl.PROTOCOL_TLS_CLIENT)
context.load_verify_locations('path/to/cabundle.pem')

with socket.socket(socket.AF_INET, socket.SOCK_STREAM, 0) as
sock:
    with context.wrap_socket(sock, server_hostname=hostname) as
ssock:
        print(ssock.version())
```

Server socket example listening on localhost IPv4:

```
context = ssl.SSLContext(ssl.PROTOCOL_TLS_SERVER)
context.load_cert_chain('/path/to/certchain.pem',
'/path/to/private.key')
```

```
with socket.socket(socket.AF_INET, socket.SOCK_STREAM, 0) as
sock:
    sock.bind(('127.0.0.1', 8443))
    sock.listen(5)
    with context.wrap_socket(sock, server_side=True) as ssock:
        conn, addr = ssock.accept()
        ...
```

Context creation

A convenience function helps create `SSLContext` objects for common purposes.

ssl.create_default_context(*purpose=Purpose.SERVER_AUTH*, *, *cafile=None*, *capath=None*, *cadata=None*)

Return a new `SSLContext` object with default settings for the given *purpose*. The settings are chosen by the `ssl` module, and usually represent a higher security level than when calling the `SSLContext` constructor directly.

cafile, *capath*, *cadata* represent optional CA certificates to trust for certificate verification, as in `SSLContext.load_verify_locations()`. If all three are `None`, this function can choose to trust the system's default CA certificates instead.

The settings are: `PROTOCOL_TLS_CLIENT` or `PROTOCOL_TLS_SERVER`, `OP_NO_SSLv2`, and `OP_NO_SSLv3` with high encryption cipher suites without RC4 and without unauthenticated cipher suites. Passing `SERVER_AUTH` as *purpose* sets `verify_mode` to `CERT_REQUIRED` and either loads CA certificates (when at least one of *cafile*, *capath* or *cadata* is given) or uses `SSLContext.load_default_certs()` to load default CA certificates.

When `keylog_filename` is supported and the environment variable **`SSLKEYLOGFILE`** is set, `create_default_context()`

enables key logging.

The default settings for this context include `VERIFY_X509_PARTIAL_CHAIN` and `VERIFY_X509_STRICT`. These make the underlying OpenSSL implementation behave more like a conforming implementation of **RFC 5280** [https://datatracker.ietf.org/doc/html/rfc5280.html], in exchange for a small amount of incompatibility with older X.509 certificates.

Note

The protocol, options, cipher and other settings may change to more restrictive values anytime without prior deprecation. The values represent a fair balance between compatibility and security.

If your application needs specific settings, you should create a `SSLContext` and apply the settings yourself.

Note

If you find that when certain older clients or servers attempt to connect with a `SSLContext` created by this function that they get an error stating “Protocol or cipher suite mismatch”, it may be that they only support SSL3.0 which this function excludes using the `OP_NO_SSLv3`. SSL3.0 is widely considered to be completely broken [https://en.wikipedia.org/wiki/POODLE]. If you still wish to continue to use this function but still allow SSL 3.0 connections you can re-enable them using:

```
ctx = ssl.create_default_context(Purpose.CLIENT_AUTH)
ctx.options &= ~ssl.OP_NO_SSLv3
```

Note

This context enables `VERIFY_X509_STRICT` by default, which may reject pre-**RFC 5280** [https://datatracker.ietf.org/doc/html/rfc5280.html] or malformed certificates that the underlying OpenSSL implementation otherwise would accept. While disabling this is not recommended, you can do so using:

```
ctx = ssl.create_default_context()
ctx.verify_flags &= ~ssl.VERIFY_X509_STRICT
```

Added in version 3.4.

Changed in version 3.4.4: RC4 was dropped from the default cipher string.

Changed in version 3.6: ChaCha20/Poly1305 was added to the default cipher string.

3DES was dropped from the default cipher string.

Changed in version 3.8: Support for key logging to **SSLKEYLOGFILE** was added.

Changed in version 3.10: The context now uses `PROTOCOL_TLS_CLIENT` or `PROTOCOL_TLS_SERVER` protocol instead of generic `PROTOCOL_TLS`.

Changed in version 3.13: The context now uses `VERIFY_X509_PARTIAL_CHAIN` and `VERIFY_X509_STRICT` in its default verify flags.

Exceptions

exception ssl.SSLError

Raised to signal an error from the underlying SSL implementation (currently provided by the OpenSSL library). This signifies some problem in the higher-level encryption and authentication layer that’s superimposed on the underlying network connection. This error is a subtype of `OSError`. The error code and message of `SSLError` instances are provided by the OpenSSL library.

Changed in version 3.3: `SSLError` used to be a subtype of `socket.error`.

library

A string mnemonic designating the OpenSSL submodule in which the error occurred, such as `SSL`, `PEM` or `X509`. The range of possible values depends on the OpenSSL version.

Added in version 3.3.

reason

A string mnemonic designating the reason this error occurred, for example `CERTIFICATE_VERIFY_FAILED`. The range of possible values depends on the OpenSSL version.

Added in version 3.3.

exception ssl.SSLZeroReturnError

A subclass of `SSLError` raised when trying to read or write and the SSL connection has been closed cleanly. Note that this doesn’t mean that the underlying transport (read TCP) has been closed.

Added in version 3.3.

exception ssl.SSLWantReadError

A subclass of `SSLError` raised by a non-blocking SSL socket when trying to read or write data, but more data needs to be received on the underlying TCP transport before the request can be fulfilled.

Added in version 3.3.

exception ssl.SSLWantWriteError

A subclass of `SSLError` raised by a non-blocking SSL socket when trying to read or write data, but more data needs to be sent on the underlying TCP transport before the request can be fulfilled.

Added in version 3.3.

exception ssl.SSLSyscallError

A subclass of `SSLError` raised when a system error was encountered while trying to fulfill an operation on a SSL socket. Unfortunately, there is no easy way to inspect the original errno number.

Added in version 3.3.

exception ssl.SSLEOFError

A subclass of `SSLError` raised when the SSL connection has been terminated abruptly. Generally, you shouldn’t try to reuse the underlying transport when this error is encountered.

Added in version 3.3.

exception ssl.SSLCertVerificationError

A subclass of `SSLError` raised when certificate validation has failed.

Added in version 3.7.

verify_code

A numeric error number that denotes the verification error.

verify_message

A human readable string of the verification error.

exception ssl.CertificateError

An alias for `SSLCertVerificationError`.

Changed in version 3.7: The exception is now an alias for `SSLCertVerificationError`.

Random generation

ssl.RAND_bytes(*num*, /)

Return *num* cryptographically strong pseudo-random bytes. Raises an `SSLError` if the PRNG has not been seeded with enough data or if the operation is not supported by the current RAND method. `RAND_status()` can be used to check the status of the PRNG and `RAND_add()` can be used to seed the PRNG.

For almost all applications `os.urandom()` is preferable.

Read the Wikipedia article, Cryptographically secure pseudorandom number generator (CSPRNG) [https://en.wikipedia.org/wiki/Cryptographically_secure_pseudorandom_number_generator], to get the requirements of a cryptographically strong generator.

Added in version 3.3.

ssl.RAND_status()

Return `True` if the SSL pseudo-random number generator has been seeded with ‘enough’ randomness, and `False` otherwise. You can use **`ssl.RAND_egd()`** and `ssl.RAND_add()` to increase the randomness of the pseudo-random number generator.

ssl.RAND_add(*bytes*, *entropy*, /)

Mix the given *bytes* into the SSL pseudo-random number generator. The parameter *entropy* (a float) is a lower bound on the entropy contained in string (so you can always use `0.0`). See **RFC 1750** [https://datatracker.ietf.org/doc/html/rfc1750.html] for more information on sources of entropy.

Changed in version 3.5: Writable bytes-like object is now accepted.

Certificate handling

ssl.cert_time_to_seconds(*cert_time*)

Return the time in seconds since the Epoch, given the `cert_time` string representing the “notBefore” or “notAfter” date from a certificate in `"%b %d %H:%M:%S %Y %Z"` strptime format (C locale).

Here’s an example:

```
>>> import ssl
>>> timestamp = ssl.cert_time_to_seconds("Jan  5 09:34:43 2018
GMT")
>>> timestamp
1515144883
>>> from datetime import datetime
>>> print(datetime.utcfromtimestamp(timestamp))
2018-01-05 09:34:43
```

“notBefore” or “notAfter” dates must use GMT (**RFC 5280** [https://datatracker.ietf.org/doc/html/rfc5280.html]).

Changed in version 3.5: Interpret the input time as a time in UTC as specified by ‘GMT’ timezone in the input string. Local timezone was used previously. Return an integer (no fractions of a second in the input format)

ssl.get_server_certificate(*addr*, *ssl_version=PROTOCOL_TLS_CLIENT*, *ca_certs=None*[, *timeout*])

Given the address `addr` of an SSL-protected server, as a (*hostname*, *port-number*) pair, fetches the server's certificate, and returns it as a PEM-encoded string. If `ssl_version` is specified, uses that version of the SSL protocol to attempt to connect to the server. If *ca_certs* is specified, it should be a file containing a list of root certificates, the same format as used for the *cafile* parameter in **`SSLContext.load_verify_locations()`**. The call will attempt to validate the server certificate against that set of root certificates, and will fail if the validation attempt fails. A timeout can be specified with the `timeout` parameter.

Changed in version 3.3: This function is now IPv6-compatible.

Changed in version 3.5: The default *ssl_version* is changed from **`PROTOCOL_SSLv3`** to **`PROTOCOL_TLS`** for maximum compatibility with modern servers.

Changed in version 3.10: The *timeout* parameter was added.

ssl.DER_cert_to_PEM_cert(*der_cert_bytes*)

Given a certificate as a DER-encoded blob of bytes, returns a PEM-encoded string version of the same certificate.

ssl.PEM_cert_to_DER_cert(*pem_cert_string*)

Given a certificate as an ASCII PEM string, returns a DER-encoded sequence of bytes for that same certificate.

ssl.get_default_verify_paths()

Returns a named tuple with paths to OpenSSL's default cafile and capath. The paths are the same as used by **`SSLContext.set_default_verify_paths()`**. The return value is a named tuple `DefaultVerifyPaths`:

- **`cafile`** - resolved path to cafile or `None` if the file doesn't exist,
- **`capath`** - resolved path to capath or `None` if the directory doesn't exist,
- **`openssl_cafile_env`** - OpenSSL's environment key that points to a cafile,
- **`openssl_cafile`** - hard coded path to a cafile,
- **`openssl_capath_env`** - OpenSSL's environment key that points to a capath,
- **`openssl_capath`** - hard coded path to a capath directory

Added in version 3.4.

ssl.enum_certificates(*store_name*)

Retrieve certificates from Windows' system cert store. *store_name* may be one of `CA`, `ROOT` or `MY`. Windows may provide additional cert stores, too.

The function returns a list of (cert_bytes, encoding_type, trust) tuples. The encoding_type specifies the encoding of cert_bytes. It is either **`x509_asn`** for X.509 ASN.1 data or **`pkcs_7_asn`** for PKCS#7 ASN.1 data. Trust specifies the purpose of the certificate as a set of OIDS or exactly `True` if the certificate is trustworthy for all purposes.

Example:

```
>>> ssl.enum_certificates("CA")
[(b'data...', 'x509_asn', {'1.3.6.1.5.5.7.3.1',
'1.3.6.1.5.5.7.3.2'}),
 (b'data...', 'x509_asn', True)]
```

Availability: Windows.

Added in version 3.4.

ssl.enum_crls(*store_name*)

Retrieve CRLs from Windows' system cert store. *store_name* may be one of `CA`, `ROOT` or `MY`. Windows may provide additional cert stores, too.

The function returns a list of (cert_bytes, encoding_type, trust) tuples. The encoding_type specifies the encoding of cert_bytes. It is either `x509_asn` for X.509 ASN.1 data or `pkcs_7_asn` for PKCS#7 ASN.1 data.

Availability: Windows.

Added in version 3.4.

Constants

All constants are now `enum.IntEnum` or `enum.IntFlag` collections.

Added in version 3.6.

ssl.CERT_NONE

Possible value for `SSLContext.verify_mode`. Except for `PROTOCOL_TLS_CLIENT`, it is the default mode. With client-side sockets, just about any cert is accepted. Validation errors, such as untrusted or expired cert, are ignored and do not abort the TLS/SSL handshake.

In server mode, no certificate is requested from the client, so the client does not send any for client cert authentication.

See the discussion of Security considerations below.

ssl.CERT_OPTIONAL

Possible value for `SSLContext.verify_mode`. In client mode, `CERT_OPTIONAL` has the same meaning as `CERT_REQUIRED`. It is recommended to use `CERT_REQUIRED` for client-side sockets instead.

In server mode, a client certificate request is sent to the client. The client may either ignore the request or send a certificate in order perform TLS client cert authentication. If the client chooses to send a certificate, it is verified. Any verification error immediately aborts the TLS handshake.

Use of this setting requires a valid set of CA certificates to be passed to `SSLContext.load_verify_locations()`.

ssl.CERT_REQUIRED

Possible value for `SSLContext.verify_mode`. In this mode, certificates are required from the other side of the socket connection; an `SSLError` will be raised if no certificate is provided, or if its validation fails. This mode is **not** sufficient to verify a certificate in client mode as it does not match hostnames. `check_hostname` must be enabled as well to verify the authenticity of a cert. `PROTOCOL_TLS_CLIENT` uses `CERT_REQUIRED` and enables `check_hostname` by default.

With server socket, this mode provides mandatory TLS client cert authentication. A client certificate request is sent to the client and the client must provide a valid and trusted certificate.

Use of this setting requires a valid set of CA certificates to be passed to `SSLContext.load_verify_locations()`.

class ssl.VerifyMode

`enum.IntEnum` collection of CERT_* constants.

Added in version 3.6.

ssl.VERIFY_DEFAULT

Possible value for `SSLContext.verify_flags`. In this mode, certificate revocation lists (CRLs) are not checked. By default OpenSSL does neither require nor verify CRLs.

Added in version 3.4.

ssl.VERIFY_CRL_CHECK_LEAF

Possible value for `SSLContext.verify_flags`. In this mode, only the peer cert is checked but none of the intermediate CA certificates. The mode requires a valid CRL that is signed by the peer cert's issuer (its direct ancestor CA). If no proper CRL has been loaded with `SSLContext.load_verify_locations`, validation will fail.

Added in version 3.4.

ssl.VERIFY_CRL_CHECK_CHAIN

Possible value for `SSLContext.verify_flags`. In this mode, CRLs of all certificates in the peer cert chain are checked.

Added in version 3.4.

ssl.VERIFY_X509_STRICT

Possible value for `SSLContext.verify_flags` to disable workarounds for broken X.509 certificates.

Added in version 3.4.

ssl.VERIFY_ALLOW_PROXY_CERTS

Possible value for `SSLContext.verify_flags` to enables proxy certificate verification.

Added in version 3.10.

ssl.VERIFY_X509_TRUSTED_FIRST

Possible value for `SSLContext.verify_flags`. It instructs OpenSSL to prefer trusted certificates when building the trust chain to validate a certificate. This flag is enabled by default.

Added in version 3.4.4.

ssl.VERIFY_X509_PARTIAL_CHAIN

Possible value for `SSLContext.verify_flags`. It instructs OpenSSL to accept intermediate CAs in the trust store to be treated as trust-anchors, in the same way as the self-signed root CA certificates. This makes it possible to trust certificates issued by an intermediate CA without having to trust its ancestor root CA.

Added in version 3.10.

class ssl.VerifyFlags

`enum.IntFlag` collection of VERIFY_* constants.

Added in version 3.6.

ssl.PROTOCOL_TLS

Selects the highest protocol version that both the client and server support. Despite the name, this option can select both “SSL” and “TLS” protocols.

Added in version 3.6.

Deprecated since version 3.10: TLS clients and servers require different default settings for secure communication. The generic TLS protocol constant is deprecated in favor of `PROTOCOL_TLS_CLIENT` and `PROTOCOL_TLS_SERVER`.

ssl.PROTOCOL_TLS_CLIENT

Auto-negotiate the highest protocol version that both the client and server support, and configure the context client-side connections. The protocol enables `CERT_REQUIRED` and `check_hostname` by default.

Added in version 3.6.

ssl.PROTOCOL_TLS_SERVER

Auto-negotiate the highest protocol version that both the client and server support, and configure the context server-side connections.

Added in version 3.6.

ssl.PROTOCOL_SSLv23

Alias for `PROTOCOL_TLS`.

Deprecated since version 3.6: Use `PROTOCOL_TLS` instead.

ssl.PROTOCOL_SSLv3

Selects SSL version 3 as the channel encryption protocol.

This protocol is not available if OpenSSL is compiled with the `no-ssl3` option.

Warning

SSL version 3 is insecure. Its use is highly discouraged.

Deprecated since version 3.6: OpenSSL has deprecated all version specific protocols. Use the default protocol `PROTOCOL_TLS_SERVER` or `PROTOCOL_TLS_CLIENT` with `SSLContext.minimum_version` and `SSLContext.maximum_version` instead.

ssl.PROTOCOL_TLSv1

Selects TLS version 1.0 as the channel encryption protocol.

Deprecated since version 3.6: OpenSSL has deprecated all version specific protocols.

ssl.PROTOCOL_TLSv1_1

Selects TLS version 1.1 as the channel encryption protocol. Available only with openssl version 1.0.1+.

Added in version 3.4.

Deprecated since version 3.6: OpenSSL has deprecated all version specific protocols.

ssl.PROTOCOL_TLSv1_2

Selects TLS version 1.2 as the channel encryption protocol. Available only with openssl version 1.0.1+.

Added in version 3.4.

Deprecated since version 3.6: OpenSSL has deprecated all version specific protocols.

ssl.OP_ALL

Enables workarounds for various bugs present in other SSL implementations. This option is set by default. It does not necessarily set the same flags as OpenSSL's `SSL_OP_ALL` constant.

Added in version 3.2.

ssl.OP_NO_SSLv2

Prevents an SSLv2 connection. This option is only applicable in conjunction with **`PROTOCOL_TLS`**. It prevents the peers from choosing SSLv2 as the protocol version.

Added in version 3.2.

Deprecated since version 3.6: SSLv2 is deprecated

ssl.OP_NO_SSLv3

Prevents an SSLv3 connection. This option is only applicable in conjunction with **`PROTOCOL_TLS`**. It prevents the peers from choosing SSLv3 as the protocol version.

Added in version 3.2.

Deprecated since version 3.6: SSLv3 is deprecated ssl.OP_NO_TLSv1

Prevents a TLSv1 connection. This option is only applicable in conjunction with `PROTOCOL_TLS`. It prevents the peers from choosing TLSv1 as the protocol version.

Added in version 3.2.

Deprecated since version 3.7: The option is deprecated since OpenSSL 1.1.0, use the new `SSLContext.minimum_version` and `SSLContext.maximum_version` instead.

ssl.OP_NO_TLSv1_1

Prevents a TLSv1.1 connection. This option is only applicable in conjunction with `PROTOCOL_TLS`. It prevents the peers from choosing TLSv1.1 as the protocol version. Available only with openssl version 1.0.1+.

Added in version 3.4.

Deprecated since version 3.7: The option is deprecated since OpenSSL 1.1.0.

ssl.OP_NO_TLSv1_2

Prevents a TLSv1.2 connection. This option is only applicable in conjunction with `PROTOCOL_TLS`. It prevents the peers from choosing TLSv1.2 as the protocol version. Available only with openssl version 1.0.1+.

Added in version 3.4.

Deprecated since version 3.7: The option is deprecated since OpenSSL 1.1.0.

ssl.OP_NO_TLSv1_3

Prevents a TLSv1.3 connection. This option is only applicable in conjunction with `PROTOCOL_TLS`. It prevents the peers from choosing TLSv1.3 as the protocol version. TLS 1.3 is available with OpenSSL 1.1.1 or later. When Python has been compiled against an older version of OpenSSL, the flag defaults to *0*.

Added in version 3.6.3.

Deprecated since version 3.7: The option is deprecated since OpenSSL 1.1.0. It was added to 2.7.15 and 3.6.3 for backwards compatibility with OpenSSL 1.0.2.

ssl.OP_NO_RENEGOTIATION

Disable all renegotiation in TLSv1.2 and earlier. Do not send HelloRequest messages, and ignore renegotiation requests via ClientHello.

This option is only available with OpenSSL 1.1.0h and later.

Added in version 3.7.

ssl.OP_CIPHER_SERVER_PREFERENCE

Use the server's cipher ordering preference, rather than the client's. This option has no effect on client sockets and SSLv2 server sockets.

Added in version 3.3.

ssl.OP_SINGLE_DH_USE

Prevents reuse of the same DH key for distinct SSL sessions. This improves forward secrecy but requires more computational resources. This option only applies to server sockets.

Added in version 3.3.

ssl.OP_SINGLE_ECDH_USE

Prevents reuse of the same ECDH key for distinct SSL sessions. This improves forward secrecy but requires more computational resources. This option only applies to server sockets.

Added in version 3.3.

ssl.OP_ENABLE_MIDDLEBOX_COMPAT

Send dummy Change Cipher Spec (CCS) messages in TLS 1.3 handshake to make a TLS 1.3 connection look more like a TLS 1.2 connection.

This option is only available with OpenSSL 1.1.1 and later.

Added in version 3.8.

ssl.OP_NO_COMPRESSION

Disable compression on the SSL channel. This is useful if the application protocol supports its own compression scheme.

Added in version 3.3.

class ssl.Options

`enum.IntFlag` collection of OP_* constants.

ssl.OP_NO_TICKET

Prevent client side from requesting a session ticket.

Added in version 3.6.

ssl.OP_IGNORE_UNEXPECTED_EOF

Ignore unexpected shutdown of TLS connections.

This option is only available with OpenSSL 3.0.0 and later.

Added in version 3.10.

ssl.OP_ENABLE_KTLS

Enable the use of the kernel TLS. To benefit from the feature, OpenSSL must have been compiled with support for it, and the negotiated cipher suites and extensions must be supported by it (a list of supported ones may vary by platform and kernel version).

Note that with enabled kernel TLS some cryptographic operations are performed by the kernel directly and not via any available OpenSSL Providers. This might be undesirable if, for example, the application requires all cryptographic operations to be performed by the FIPS provider.

This option is only available with OpenSSL 3.0.0 and later.

Added in version 3.12.

ssl.OP_LEGACY_SERVER_CONNECT

Allow legacy insecure renegotiation between OpenSSL and unpatched servers only.

Added in version 3.12.

ssl.HAS_ALPN

Whether the OpenSSL library has built-in support for the *Application-Layer Protocol Negotiation* TLS extension as described in **RFC 7301** [https://datatracker.ietf.org/doc/html/rfc7301.html].

Added in version 3.5.

ssl.HAS_NEVER_CHECK_COMMON_NAME

Whether the OpenSSL library has built-in support not checking subject common name and `SSLContext.hostname_checks_common_name` is writeable.

Added in version 3.7.

ssl.HAS_ECDH

Whether the OpenSSL library has built-in support for the Elliptic Curve-based Diffie-Hellman key exchange. This should be true unless the feature was explicitly disabled by the distributor.

Added in version 3.3.

ssl.HAS_SNI

Whether the OpenSSL library has built-in support for the *Server Name Indication* extension (as defined in **RFC 6066** [https://datatracker.ietf.org/doc/html/rfc6066.html]).

Added in version 3.2.

ssl.HAS_NPN

Whether the OpenSSL library has built-in support for the *Next Protocol Negotiation* as described in the Application Layer Protocol Negotiation [https://en.wikipedia.org/wiki/Application-Layer_Protocol_Negotiation]. When true, you can use the `SSLContext.set_npn_protocols()` method to advertise which protocols you want to support.

Added in version 3.3.

ssl.HAS_SSLv2

Whether the OpenSSL library has built-in support for the SSL 2.0 protocol.

Added in version 3.7.

ssl.HAS_SSLv3

Whether the OpenSSL library has built-in support for the SSL 3.0 protocol.

Added in version 3.7.

ssl.HAS_TLSv1

Whether the OpenSSL library has built-in support for the TLS 1.0 protocol.

Added in version 3.7.

ssl.HAS_TLSv1_1

Whether the OpenSSL library has built-in support for the TLS 1.1 protocol.

Added in version 3.7.

ssl.HAS_TLSv1_2

Whether the OpenSSL library has built-in support for the TLS 1.2 protocol.

Added in version 3.7.

ssl.HAS_TLSv1_3

Whether the OpenSSL library has built-in support for the TLS 1.3 protocol.

Added in version 3.7.

ssl.HAS_PSK

Whether the OpenSSL library has built-in support for TLS-PSK.

Added in version 3.13.

ssl.HAS_PHA

Whether the OpenSSL library has built-in support for TLS-PHA.

Added in version 3.14.

ssl.CHANNEL_BINDING_TYPES

List of supported TLS channel binding types. Strings in this list can be used as arguments to `SSLSocket.get_channel_binding()`.

Added in version 3.3.

ssl.OPENSSL_VERSION

The version string of the OpenSSL library loaded by the interpreter:

```
>>> ssl.OPENSSL_VERSION
'OpenSSL 1.0.2k  26 Jan 2017'
```

Added in version 3.2.

ssl.OPENSSL_VERSION_INFO

A tuple of five integers representing version information about the OpenSSL library:

```
>>> ssl.OPENSSL_VERSION_INFO
(1, 0, 2, 11, 15)
```

Added in version 3.2.

ssl.OPENSSL_VERSION_NUMBER

The raw version number of the OpenSSL library, as a single integer:

```
>>> ssl.OPENSSL_VERSION_NUMBER
268443839
>>> hex(ssl.OPENSSL_VERSION_NUMBER)
'0x100020bf'
```

Added in version 3.2.

ssl.ALERT_DESCRIPTION_HANDSHAKE_FAILURE
ssl.ALERT_DESCRIPTION_INTERNAL_ERROR
ALERT_DESCRIPTION_*

Alert Descriptions from **RFC 5246** [https://datatracker.ietf.org/doc/html/rfc5246.html] and others. The IANA TLS Alert Registry [https://www.iana.org/assignments/tls-parameters/tls-parameters.xml#tls-parameters-6] contains this list and references to the RFCs where their meaning is defined.

Used as the return value of the callback function in **`SSLContext.set_servername_callback()`**.

Added in version 3.4.

class ssl.AlertDescription

`enum.IntEnum` collection of ALERT_DESCRIPTION_* constants.

Added in version 3.6.

Purpose.SERVER_AUTH

Option for `create_default_context()` and `SSLContext.load_default_certs()`. This value indicates that the context may be used to authenticate web servers (therefore, it will be used to create client-side sockets).

Added in version 3.4.

Purpose.CLIENT_AUTH

Option for `create_default_context()` and `SSLContext.load_default_certs()`. This value indicates that the context may be used to authenticate web clients (therefore, it will be used to create server-side sockets).

Added in version 3.4.

class ssl.SSLErrorNumber

`enum.IntEnum` collection of SSL_ERROR_* constants.

Added in version 3.6.

class ssl.TLSVersion

`enum.IntEnum` collection of SSL and TLS versions for `SSLContext.maximum_version` and `SSLContext.minimum_version`.

Added in version 3.7.

TLSVersion.MINIMUM_SUPPORTED

TLSVersion.MAXIMUM_SUPPORTED

The minimum or maximum supported SSL or TLS version. These are magic constants. Their values don’t reflect the lowest and highest available TLS/SSL versions.

TLSVersion.SSLv3

TLSVersion.TLSv1

TLSVersion.TLSv1_1

TLSVersion.TLSv1_2

TLSVersion.TLSv1_3

SSL 3.0 to TLS 1.3.

Deprecated since version 3.10: All `TLSVersion` members except `TLSVersion.TLSv1_2` and `TLSVersion.TLSv1_3` are deprecated.

SSL Sockets

class ssl.SSLSocket(*socket.socket*)

SSL sockets provide the following methods of Socket Objects:

- `accept()`
- `bind()`
- `close()`
- `connect()`
- `detach()`
- `fileno()`
- `getpeername()`, `getsockname()`
- `getsockopt()`, `setsockopt()`
- `gettimeout()`, `settimeout()`, `setblocking()`
- `listen()`
- `makefile()`

- `recv()`, `recv_into()` (but passing a non-zero `flags` argument is not allowed)
- `send()`, `sendall()` (with the same limitation)
- `sendfile()` (but `os.sendfile` will be used for plain-text sockets only, else `send()` will be used)
- `shutdown()`

However, since the SSL (and TLS) protocol has its own framing atop of TCP, the SSL sockets abstraction can, in certain respects, diverge from the specification of normal, OS-level sockets. See especially the notes on non-blocking sockets.

Instances of `SSLSocket` must be created using the `SSLContext.wrap_socket()` method.

Changed in version 3.5: The **`sendfile()`** method was added.

Changed in version 3.5: The **`shutdown()`** does not reset the socket timeout each time bytes are received or sent. The socket timeout is now the maximum total duration of the shutdown.

Deprecated since version 3.6: It is deprecated to create a `SSLSocket` instance directly, use `SSLContext.wrap_socket()` to wrap a socket.

Changed in version 3.7: `SSLSocket` instances must to created with `wrap_socket()`. In earlier versions, it was possible to create instances directly. This was never documented or officially supported.

Changed in version 3.10: Python now uses `SSL_read_ex` and `SSL_write_ex` internally. The functions support reading and writing of data larger than 2 GB. Writing zero-length data no longer fails with a protocol violation error.

SSL sockets also have the following additional methods and attributes:

SSLSocket.read(*len=1024*, *buffer=None*)

Read up to *len* bytes of data from the SSL socket and return the result as a `bytes` instance. If *buffer* is specified, then read into the buffer instead, and return the number of bytes read.

Raise **`SSLWantReadError`** or **`SSLWantWriteError`** if the socket is non-blocking and the read would block.

As at any time a re-negotiation is possible, a call to **`read()`** can also cause write operations.

Changed in version 3.5: The socket timeout is no longer reset each time bytes are received or sent. The socket timeout is now the maximum total duration to read up to *len* bytes.

Deprecated since version 3.6: Use **`recv()`** instead of **`read()`**.

SSLSocket.write(*data*)

Write *data* to the SSL socket and return the number of bytes written. The *data* argument must be an object supporting the buffer interface.

Raise **`SSLWantReadError`** or **`SSLWantWriteError`** if the socket is non-blocking and the write would block.

As at any time a re-negotiation is possible, a call to **`write()`** can also cause read operations.

Changed in version 3.5: The socket timeout is no longer reset each time bytes are received or sent. The socket timeout is now the maximum total duration to write *data*.

Deprecated since version 3.6: Use **`send()`** instead of **`write()`**.

Note

The `read()` and `write()` methods are the low-level methods that read and write unencrypted, application-level data and decrypt/encrypt it to encrypted, wire-level data. These methods require an active SSL connection, i.e. the handshake was completed and `SSLSocket.unwrap()` was not called.

Normally you should use the socket API methods like `recv()` and `send()` instead of these methods.

SSLSocket.do_handshake(*block=False*)

Perform the SSL setup handshake.

If *block* is true and the timeout obtained by `gettimeout()` is zero, the socket is set in blocking mode until the handshake is performed.

Changed in version 3.4: The handshake method also performs `match_hostname()` when the `check_hostname` attribute of the socket's `context` is true.

Changed in version 3.5: The socket timeout is no longer reset each time bytes are received or sent. The socket timeout is now the maximum total duration of the handshake.

Changed in version 3.7: Hostname or IP address is matched by OpenSSL during handshake. The function `match_hostname()` is no longer used. In case OpenSSL refuses a hostname or IP address, the handshake is aborted early and a TLS alert message is sent to the peer.

SSLSocket.getpeercert(*binary_form=False*)

If there is no certificate for the peer on the other end of the connection, return `None`. If the SSL handshake hasn't been done yet, raise **`ValueError`**.

If the `binary_form` parameter is **`False`**, and a certificate was received from the peer, this method returns a **`dict`** instance. If the certificate was not validated, the dict is empty. If the certificate was validated, it returns a dict with several keys, amongst them `subject` (the principal for which the certificate was issued) and `issuer` (the principal issuing the certificate). If a certificate contains an instance of the *Subject Alternative Name* extension (see **RFC 3280** [https://datatracker.ietf.org/doc/html/rfc3280.html]), there will also be a `subjectAltName` key in the dictionary.

The `subject` and `issuer` fields are tuples containing the sequence of relative distinguished names (RDNs) given in the certificate's data structure for the respective fields, and each RDN is a sequence of name-value pairs. Here is a real-world example:

```
{'issuer': ((('countryName', 'IL'),),
            (('organizationName', 'StartCom Ltd.'),),
            (('organizationalUnitName',
              'Secure Digital Certificate Signing'),),
            (('commonName',
              'StartCom Class 2 Primary Intermediate Server
CA'),)),
 'notAfter': 'Nov 22 08:15:19 2013 GMT',
 'notBefore': 'Nov 21 03:09:52 2011 GMT',
 'serialNumber': '95F0',
 'subject': ((('description', '571208-SLe257oHY9fVQ07Z'),),
             (('countryName', 'US'),),
             (('stateOrProvinceName', 'California'),),
             (('localityName', 'San Francisco'),),
             (('organizationName', 'Electronic Frontier
Foundation, Inc.'),),
             (('commonName', '*.eff.org'),),
             (('emailAddress', 'hostmaster@eff.org'),)),
 'subjectAltName': (('DNS', '*.eff.org'), ('DNS', 'eff.org')),
 'version': 3}
```

If the `binary_form` parameter is `True`, and a certificate was provided, this method returns the DER-encoded form of the entire certificate as a sequence of bytes, or `None` if the peer did not provide a certificate. Whether the peer provides a certificate depends on the SSL socket's role:

- for a client SSL socket, the server will always provide a certificate, regardless of whether validation was required;
- for a server SSL socket, the client will only provide a certificate when requested by the server; therefore `getpeercert()` will return `None` if you used `CERT_NONE` (rather than `CERT_OPTIONAL` or `CERT_REQUIRED`).

See also `SSLContext.check_hostname`.

Changed in version 3.2: The returned dictionary includes additional items such as `issuer` and `notBefore`.

Changed in version 3.4: `ValueError` is raised when the handshake isn't done. The returned dictionary includes additional X509v3 extension items such as `crlDistributionPoints`, `caIssuers` and `OCSP` URIs.

Changed in version 3.9: IPv6 address strings no longer have a trailing new line.

SSLSocket.get_verified_chain()

Returns verified certificate chain provided by the other end of the SSL channel as a list of DER-encoded bytes. If certificate verification was disabled method acts the same as `get_unverified_chain()`.

Added in version 3.13.

SSLSocket.get_unverified_chain()

Returns raw certificate chain provided by the other end of the SSL channel as a list of DER-encoded bytes.

Added in version 3.13.

SSLSocket.cipher()

Returns a three-value tuple containing the name of the cipher being used, the version of the SSL protocol that defines its use, and the number of secret bits being used. If no connection has been established, returns `None`.

SSLSocket.shared_ciphers()

Return the list of ciphers available in both the client and server. Each entry of the returned list is a three-value tuple containing the name of the cipher, the version of the SSL protocol that defines its use, and the number of secret bits the cipher uses. **`shared_ciphers()`** returns `None` if no connection has been established or the socket is a client socket.

Added in version 3.5.

SSLSocket.compression()

Return the compression algorithm being used as a string, or `None` if the connection isn’t compressed.

If the higher-level protocol supports its own compression mechanism, you can use **`OP_NO_COMPRESSION`** to disable SSL-level compression.

Added in version 3.3.

SSLSocket.get_channel_binding(*cb_type*='*tls-unique*')

Get channel binding data for current connection, as a bytes object. Returns `None` if not connected or the handshake has not been completed.

The *cb_type* parameter allow selection of the desired channel binding type. Valid channel binding types are listed in the `CHANNEL_BINDING_TYPES` list. Currently only the ‘tls-unique’ channel binding, defined by **RFC 5929** [https://datatracker.ietf.org/doc/html/rfc5929.html], is supported. `ValueError` will be raised if an unsupported channel binding type is requested.

Added in version 3.3.

SSLSocket.selected_alpn_protocol()

Return the protocol that was selected during the TLS handshake. If `SSLContext.set_alpn_protocols()` was not called, if the other party does not support ALPN, if this socket does not support any of the client’s proposed protocols, or if the handshake has not happened yet, `None` is returned.

Added in version 3.5.

SSLSocket.selected_npn_protocol()

Return the higher-level protocol that was selected during the TLS/SSL handshake. If `SSLContext.set_npn_protocols()` was not called, or if the other party does not support NPN, or if the handshake has not yet happened, this will return `None`.

Added in version 3.3.

Deprecated since version 3.10: NPN has been superseded by ALPN

SSLSocket.unwrap()

Performs the SSL shutdown handshake, which removes the TLS layer from the underlying socket, and returns the underlying socket object. This can be used to go from encrypted operation over a connection to unencrypted. The returned socket should always be used for further communication with the other side of the connection, rather than the original socket.

SSLSocket.verify_client_post_handshake()

Requests post-handshake authentication (PHA) from a TLS 1.3 client. PHA can only be initiated for a TLS 1.3 connection from a server-side socket, after the initial TLS handshake and with PHA enabled on both sides, see `SSLContext.post_handshake_auth`.

The method does not perform a cert exchange immediately. The server-side sends a CertificateRequest during the next write event and expects the client to respond with a certificate on the next read event.

If any precondition isn't met (e.g. not TLS 1.3, PHA not enabled), an `SSLError` is raised.

Note

Only available with OpenSSL 1.1.1 and TLS 1.3 enabled. Without TLS 1.3 support, the method raises `NotImplementedError`.

Added in version 3.8.

SSLSocket.version()

Return the actual SSL protocol version negotiated by the connection as a string, or `None` if no secure connection is established. As of this writing, possible return values include `"SSLv2"`, `"SSLv3"`, `"TLSv1"`, `"TLSv1.1"` and `"TLSv1.2"`. Recent OpenSSL versions may define more return values.

Added in version 3.5.

SSLSocket.pending()

Returns the number of already decrypted bytes available for read, pending on the connection.

SSLSocket.context

The `SSLContext` object this SSL socket is tied to.

Added in version 3.2.

SSLSocket.server_side

A boolean which is `True` for server-side sockets and `False` for client-side sockets.

Added in version 3.2.

SSLSocket.server_hostname

Hostname of the server: `str` type, or `None` for server-side socket or if the hostname was not specified in the constructor.

Added in version 3.2.

Changed in version 3.7: The attribute is now always ASCII text. When `server_hostname` is an internationalized domain name (IDN), this attribute now stores the A-label form (`"xn--pythn-mua.org"`), rather than the U-label form (`"pythön.org"`).

SSLSocket.session

The `SSLSession` for this SSL connection. The session is available for client and server side sockets after the TLS handshake has been performed. For client sockets the session can be set before `do_handshake()` has been called to reuse a session.

Added in version 3.6.

SSLSocket.session_reused

Added in version 3.6.

SSL Contexts

Added in version 3.2.

An SSL context holds various data longer-lived than single SSL connections, such as SSL configuration options, certificate(s) and private key(s). It also manages a cache of SSL sessions for server-side sockets, in order to speed up repeated connections from the same clients.

class ssl.SSLContext(*protocol=None*)

Create a new SSL context. You may pass *protocol* which must be one of the `PROTOCOL_*` constants defined in this module. The parameter specifies which version of the SSL protocol to use. Typically, the server chooses a particular protocol version, and the client must adapt to the server’s choice. Most of the versions are not interoperable with the other versions. If not specified, the default is **`PROTOCOL_TLS`**; it provides the most compatibility with other versions.

Here’s a table showing which versions in a client (down the side) can connect to which versions in a server (along the top):

client / **server**	**SSLv2**	**SSLv3**	**TLS** [[3]]	**TLSv1**	**TLSv1.1**	**TLSv1.2**
SSLv2	yes	no	no [[1]]	no	no	no
SSLv3	no	yes	no [[2]]	no	no	no
TLS (*SSLv23*) [[3]]	no [[1]]	no [[2]]	yes	yes	yes	yes

TLSv1	no	no	yes	yes	no	no
TLSv1.1	no	no	yes	no	yes	no
TLSv1.2	no	no	yes	no	no	yes

Footnotes

[1] (1,2)
`SSLContext` disables SSLv2 with `OP_NO_SSLv2` by default.

[2] (1,2)
`SSLContext` disables SSLv3 with `OP_NO_SSLv3` by default.

[3] (1,2)
TLS 1.3 protocol will be available with `PROTOCOL_TLS` in OpenSSL >= 1.1.1. There is no dedicated PROTOCOL constant for just TLS 1.3.

See also

`create_default_context()` lets the `ssl` module choose security settings for a given purpose.

Changed in version 3.6: The context is created with secure default values. The options `OP_NO_COMPRESSION`, `OP_CIPHER_SERVER_PREFERENCE`, `OP_SINGLE_DH_USE`, `OP_SINGLE_ECDH_USE`, `OP_NO_SSLv2`, and `OP_NO_SSLv3` (except for `PROTOCOL_SSLv3`) are set by default. The initial cipher suite list contains only `HIGH` ciphers, no `NULL` ciphers and no `MD5` ciphers.

Deprecated since version 3.10: `SSLContext` without protocol argument is deprecated. The context class will either require `PROTOCOL_TLS_CLIENT` or `PROTOCOL_TLS_SERVER` protocol in the future.

Changed in version 3.10: The default cipher suites now include only secure AES and ChaCha20 ciphers with forward secrecy and security level 2. RSA and DH keys with less than 2048 bits and ECC keys with less than 224 bits are prohibited. `PROTOCOL_TLS`, `PROTOCOL_TLS_CLIENT`, and `PROTOCOL_TLS_SERVER` use TLS 1.2 as minimum TLS version.

Note

`SSLContext` only supports limited mutation once it has been used by a connection. Adding new certificates to the internal trust store is allowed, but changing ciphers, verification settings, or mTLS certificates may result in surprising behavior.

Note

`SSLContext` is designed to be shared and used by multiple connections. Thus, it is thread-safe as long as it is not reconfigured after being used by a connection.

`SSLContext` objects have the following methods and attributes:

SSLContext.cert_store_stats()

Get statistics about quantities of loaded X.509 certificates, count of X.509 certificates flagged as CA certificates and certificate revocation lists as dictionary.

Example for a context with one CA cert and one other cert:

```
>>> context.cert_store_stats()
{'crl': 0, 'x509_ca': 1, 'x509': 2}
```

Added in version 3.4.

SSLContext.load_cert_chain(*certfile*, *keyfile=None*, *password=None*)

Load a private key and the corresponding certificate. The *certfile* string must be the path to a single file in PEM format containing the certificate as well as any number of CA certificates needed to establish the certificate's authenticity. The *keyfile* string, if present, must point to a file containing the private key. Otherwise the private key will be taken from *certfile* as well. See the discussion of Certificates for more information on how the certificate is stored in the *certfile*.

The *password* argument may be a function to call to get the password for decrypting the private key. It will only be called if the private key is encrypted and a password is necessary. It will be called with no arguments, and it should return a string, bytes, or bytearray. If the return value is a string it will be encoded as UTF-8 before using it to decrypt the key. Alternatively a string, bytes, or bytearray value may be supplied directly as the *password* argument. It will be ignored if the private key is not encrypted and no password is needed.

If the *password* argument is not specified and a password is required, OpenSSL's built-in password prompting mechanism will be used to interactively prompt the user for a password.

An **`SSLError`** is raised if the private key doesn't match with the certificate.

Changed in version 3.3: New optional argument *password*.

SSLContext.load_default_certs(*purpose=Purpose.SERVER_AUTH*)

Load a set of default "certification authority" (CA) certificates from default locations. On Windows it loads CA certs from the `CA` and `ROOT` system stores. On all systems it calls `SSLContext.set_default_verify_paths()`. In the future the method may load CA certificates from other locations, too.

The *purpose* flag specifies what kind of CA certificates are loaded. The default settings `Purpose.SERVER_AUTH` loads certificates, that are flagged and trusted for TLS web server authentication (client side sockets). `Purpose.CLIENT_AUTH` loads CA certificates for client certificate verification on the server side.

Added in version 3.4.

SSLContext.load_verify_locations(*cafile=None*, *capath=None*, *cadata=None*)

Load a set of “certification authority” (CA) certificates used to validate other peers’ certificates when `verify_mode` is other than `CERT_NONE`. At least one of *cafile* or *capath* must be specified.

This method can also load certification revocation lists (CRLs) in PEM or DER format. In order to make use of CRLs, `SSLContext.verify_flags` must be configured properly.

The *cafile* string, if present, is the path to a file of concatenated CA certificates in PEM format. See the discussion of Certificates for more information about how to arrange the certificates in this file.

The *capath* string, if present, is the path to a directory containing several CA certificates in PEM format, following an OpenSSL specific layout [https://docs.openssl.org/master/man3/SSL_CTX_load_verify_locations/].

The *cadata* object, if present, is either an ASCII string of one or more PEM-encoded certificates or a bytes-like object of DER-encoded certificates. Like with *capath* extra lines around PEM-encoded certificates are ignored but at least one certificate must be present.

Changed in version 3.4: New optional argument *cadata*

SSLContext.get_ca_certs(*binary_form=False*)

Get a list of loaded “certification authority” (CA) certificates. If the `binary_form` parameter is `False` each list entry is a dict like the output of `SSLSocket.getpeercert()`. Otherwise the method returns a list of DER-encoded certificates. The returned list does not contain certificates from *capath* unless a certificate was requested and loaded by a SSL connection.

Note

Certificates in a capath directory aren’t loaded unless they have been used at least once.

Added in version 3.4.

SSLContext.get_ciphers()

Get a list of enabled ciphers. The list is in order of cipher priority. See `SSLContext.set_ciphers()`.

Example:

```
>>> ctx = ssl.SSLContext(ssl.PROTOCOL_SSLv23)
>>> ctx.set_ciphers('ECDHE+AESGCM:!ECDSA')
>>> ctx.get_ciphers()
[{'aead': True,
  'alg_bits': 256,
  'auth': 'auth-rsa',
  'description': 'ECDHE-RSA-AES256-GCM-SHA384 TLSv1.2 Kx=ECDH
Au=RSA  '
                 'Enc=AESGCM(256) Mac=AEAD',
  'digest': None,
  'id': 50380848,
  'kea': 'kx-ecdhe',
  'name': 'ECDHE-RSA-AES256-GCM-SHA384',
  'protocol': 'TLSv1.2',
  'strength_bits': 256,
  'symmetric': 'aes-256-gcm'},
 {'aead': True,
  'alg_bits': 128,
  'auth': 'auth-rsa',
  'description': 'ECDHE-RSA-AES128-GCM-SHA256 TLSv1.2 Kx=ECDH
Au=RSA  '
                 'Enc=AESGCM(128) Mac=AEAD',
  'digest': None,
  'id': 50380847,
  'kea': 'kx-ecdhe',
  'name': 'ECDHE-RSA-AES128-GCM-SHA256',
  'protocol': 'TLSv1.2',
  'strength_bits': 128,
  'symmetric': 'aes-128-gcm'}]
```

Added in version 3.6.

SSLContext.set_default_verify_paths()

Load a set of default “certification authority” (CA) certificates from a filesystem path defined when building the OpenSSL library. Unfortunately, there’s no easy way to know whether this method succeeds: no error is returned if no certificates are to be found. When the OpenSSL library is provided as part of the operating system, though, it is likely to be configured properly.

SSLContext.set_ciphers(*ciphers*, */*)

Set the available ciphers for sockets created with this context. It should be a string in the OpenSSL cipher list format [https://docs.openssl.org/master/man1/ciphers/]. If no cipher can be selected (because compile-time options or other configuration forbids use of all the specified ciphers), an `SSLError` will be raised.

Note

when connected, the `SSLSocket.cipher()` method of SSL sockets will give the currently selected cipher.

TLS 1.3 cipher suites cannot be disabled with `set_ciphers()`.

SSLContext.set_alpn_protocols(*alpn_protocols*)

Specify which protocols the socket should advertise during the SSL/TLS handshake. It should be a list of ASCII strings, like `['http/1.1', 'spdy/2']`, ordered by preference. The selection of a protocol will happen during the handshake, and will play out according to **RFC 7301** [https://datatracker.ietf.org/doc/html/rfc7301.html]. After a successful handshake, the `SSLSocket.selected_alpn_protocol()` method will return the agreed-upon protocol.

This method will raise `NotImplementedError` if `HAS_ALPN` is `False`.

Added in version 3.5.

SSLContext.set_npn_protocols(*npn_protocols*)

Specify which protocols the socket should advertise during the SSL/TLS handshake. It should be a list of strings, like `['http/1.1', 'spdy/2']`, ordered by preference. The selection of a protocol will happen during the handshake, and will play out according to the Application Layer Protocol Negotiation [https://en.wikipedia.org/wiki/Application-Layer_Protocol_Negotiation]. After a successful handshake, the `SSLSocket.selected_npn_protocol()` method will return the agreed-upon protocol.

This method will raise `NotImplementedError` if `HAS_NPN` is `False`.

Added in version 3.3.

Deprecated since version 3.10: NPN has been superseded by ALPN

SSLContext.sni_callback

Register a callback function that will be called after the TLS Client Hello handshake message has been received by the SSL/TLS server when the TLS client specifies a server name indication. The server name indication mechanism is specified in **RFC 6066** [https://datatracker.ietf.org/doc/html/rfc6066.html] section 3 - Server Name Indication.

Only one callback can be set per `SSLContext`. If *sni_callback* is set to `None` then the callback is disabled. Calling this function a subsequent time will disable the previously registered callback.

The callback function will be called with three arguments; the first being the `ssl.SSLSocket`, the second is a string that represents the server name that the client is intending to communicate (or `None` if the TLS Client Hello does not contain a server name) and the third argument is the original `SSLContext`. The server name argument is text. For internationalized domain name, the server name is an IDN A-label (`"xn--pythn-mua.org"`).

A typical use of this callback is to change the `ssl.SSLSocket`'s `SSLSocket.context` attribute to a new object of type `SSLContext` representing a certificate chain that matches the server name.

Due to the early negotiation phase of the TLS connection, only limited methods and attributes are usable like `SSLSocket.selected_alpn_protocol()` and `SSLSocket.context`. The `SSLSocket.getpeercert()`, `SSLSocket.get_verified_chain()`, `SSLSocket.get_unverified_chain()` `SSLSocket.cipher()` and `SSLSocket.compression()` methods require that the TLS connection has progressed beyond the TLS Client Hello and therefore will not return meaningful values nor can they be called safely.

The *sni_callback* function must return `None` to allow the TLS negotiation to continue. If a TLS failure is required, a constant **`ALERT_DESCRIPTION_*`** can be returned. Other return values will result in a TLS fatal error with **`ALERT_DESCRIPTION_INTERNAL_ERROR`**.

If an exception is raised from the *sni_callback* function the TLS connection will terminate with a fatal TLS alert message **`ALERT_DESCRIPTION_HANDSHAKE_FAILURE`**.

This method will raise **`NotImplementedError`** if the OpenSSL library had OPENSSL_NO_TLSEXT defined when it was built.

Added in version 3.7.

SSLContext.set_servername_callback(*server_name_callback*)

This is a legacy API retained for backwards compatibility. When possible, you should use **`sni_callback`** instead. The given *server_name_callback* is similar to *sni_callback*, except that when the server hostname is an IDN-encoded internationalized domain name, the *server_name_callback* receives a decoded U-label (`"pythön.org"`).

If there is a decoding error on the server name, the TLS connection will terminate with an **`ALERT_DESCRIPTION_INTERNAL_ERROR`** fatal TLS alert message to the client.

Added in version 3.4.

SSLContext.load_dh_params(*dhfile*, */*)

Load the key generation parameters for Diffie-Hellman (DH) key exchange. Using DH key exchange improves forward secrecy at the expense of computational resources (both on the server and on the client). The *dhfile* parameter should be the path to a file containing DH parameters in PEM format.

This setting doesn’t apply to client sockets. You can also use the `OP_SINGLE_DH_USE` option to further improve security.

Added in version 3.3.

SSLContext.set_ecdh_curve(*curve_name*, /)

Set the curve name for Elliptic Curve-based Diffie-Hellman (ECDH) key exchange. ECDH is significantly faster than regular DH while arguably as secure. The *curve_name* parameter should be a string describing a well-known elliptic curve, for example `prime256v1` for a widely supported curve.

This setting doesn’t apply to client sockets. You can also use the `OP_SINGLE_ECDH_USE` option to further improve security.

This method is not available if `HAS_ECDH` is `False`.

Added in version 3.3.

See also

SSL/TLS & Perfect Forward Secrecy

[https://vincent.bernat.ch/en/blog/2011-ssl-perfect-forward-secrecy]

Vincent Bernat.

SSLContext.wrap_socket(*sock*, *server_side=False*, *do_handshake_on_connect=True*, *suppress_ragged_eofs=True*, *server_hostname=None*, *session=None*)

Wrap an existing Python socket *sock* and return an instance of `SSLContext.sslsocket_class` (default `SSLSocket`). The returned SSL socket is tied to the context, its settings and certificates. *sock* must be a `SOCK_STREAM` socket; other socket types are unsupported.

The parameter `server_side` is a boolean which identifies whether server-side or client-side behavior is desired from this socket.

For client-side sockets, the context construction is lazy; if the underlying socket isn’t connected yet, the context construction will be performed after **`connect()`** is called on the socket. For server-side sockets, if the socket has no remote peer, it is assumed to be a listening socket, and the server-side SSL wrapping is automatically performed on client connections accepted via the **`accept()`** method. The method may raise `SSLError`.

On client connections, the optional parameter *server_hostname* specifies the hostname of the service which we are connecting to. This allows a single server to host multiple SSL-based services with distinct certificates, quite similarly to HTTP virtual hosts. Specifying *server_hostname* will raise a `ValueError` if *server_side* is true.

The parameter `do_handshake_on_connect` specifies whether to do the SSL handshake automatically after doing a **`socket.connect()`**, or whether the application program will call it explicitly, by invoking the `SSLSocket.do_handshake()` method. Calling `SSLSocket.do_handshake()` explicitly gives the program control over the blocking behavior of the socket I/O involved in the handshake.

The parameter `suppress_ragged_eofs` specifies how the **`SSLSocket.recv()`** method should signal unexpected EOF from the other end of the connection. If specified as `True` (the default), it returns a normal EOF (an empty bytes object) in response to unexpected EOF errors raised from the underlying socket; if `False`, it will raise the exceptions back to the caller.

session, see `session`.

To wrap an `SSLSocket` in another `SSLSocket`, use `SSLContext.wrap_bio()`.

Changed in version 3.5: Always allow a server_hostname to be passed, even if OpenSSL does not have SNI.

Changed in version 3.6: session argument was added.

Changed in version 3.7: The method returns an instance of `SSLContext.sslsocket_class` instead of hard-coded `SSLSocket`.

SSLContext.sslsocket_class

The return type of `SSLContext.wrap_socket()`, defaults to `SSLSocket`. The attribute can be assigned to on instances of `SSLContext` in order to return a custom subclass of `SSLSocket`.

Added in version 3.7.

SSLContext.wrap_bio(*incoming*, *outgoing*, *server_side=False*, *server_hostname=None*, *session=None*)

Wrap the BIO objects *incoming* and *outgoing* and return an instance of `SSLContext.sslobject_class` (default `SSLObject`). The SSL routines will read input data from the incoming BIO and write data to the outgoing BIO.

The *server_side*, *server_hostname* and *session* parameters have the same meaning as in `SSLContext.wrap_socket()`.

Changed in version 3.6: session argument was added.

Changed in version 3.7: The method returns an instance of `SSLContext.sslobject_class` instead of hard-coded `SSLObject`.

SSLContext.sslobject_class

The return type of `SSLContext.wrap_bio()`, defaults to `SSLObject`. The attribute can be overridden on instance of class in order to return a custom subclass of **`SSLObject`**.

Added in version 3.7.

SSLContext.session_stats()

Get statistics about the SSL sessions created or managed by this context. A dictionary is returned which maps the names of each piece of information [https://docs.openssl.org/1.1.1/man3/SSL_CTX_sess_number/] to their numeric values. For example, here is the total number of hits and misses in the session cache since the context was created:

```
>>> stats = context.session_stats()
>>> stats['hits'], stats['misses']
(0, 0)
```

SSLContext.check_hostname

Whether to match the peer cert's hostname in **`SSLSocket.do_handshake()`**. The context's **`verify_mode`** must be set to **`CERT_OPTIONAL`** or **`CERT_REQUIRED`**, and you must pass *server_hostname* to **`wrap_socket()`** in order to match the hostname. Enabling hostname checking automatically sets **`verify_mode`** from **`CERT_NONE`** to **`CERT_REQUIRED`**. It cannot be set back to **`CERT_NONE`** as long as hostname checking is enabled. The **`PROTOCOL_TLS_CLIENT`** protocol enables hostname checking by default. With other protocols, hostname checking must be enabled explicitly.

Example:

```
import socket, ssl

context = ssl.SSLContext(ssl.PROTOCOL_TLSv1_2)
context.verify_mode = ssl.CERT_REQUIRED
context.check_hostname = True
context.load_default_certs()

s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
```

```
ssl_sock = context.wrap_socket(s,
server_hostname='www.verisign.com')
ssl_sock.connect(('www.verisign.com', 443))
```

Added in version 3.4.

Changed in version 3.7: `verify_mode` is now automatically changed to `CERT_REQUIRED` when hostname checking is enabled and `verify_mode` is `CERT_NONE`. Previously the same operation would have failed with a `ValueError`.

SSLContext.keylog_filename

Write TLS keys to a keylog file, whenever key material is generated or received. The keylog file is designed for debugging purposes only. The file format is specified by NSS and used by many traffic analyzers such as Wireshark. The log file is opened in append-only mode. Writes are synchronized between threads, but not between processes.

Added in version 3.8.

SSLContext.maximum_version

A `TLSVersion` enum member representing the highest supported TLS version. The value defaults to `TLSVersion.MAXIMUM_SUPPORTED`. The attribute is read-only for protocols other than `PROTOCOL_TLS`, `PROTOCOL_TLS_CLIENT`, and `PROTOCOL_TLS_SERVER`.

The attributes `maximum_version`, `minimum_version` and `SSLContext.options` all affect the supported SSL and TLS versions of the context. The implementation does not prevent invalid combination. For example a context with `OP_NO_TLSv1_2` in `options` and `maximum_version` set to `TLSVersion.TLSv1_2` will not be able to establish a TLS 1.2 connection.

Added in version 3.7.

SSLContext.minimum_version

Like `SSLContext.maximum_version` except it is the lowest supported version or `TLSVersion.MINIMUM_SUPPORTED`.

Added in version 3.7.

SSLContext.num_tickets

Control the number of TLS 1.3 session tickets of a `PROTOCOL_TLS_SERVER` context. The setting has no impact on TLS 1.0 to 1.2 connections.

Added in version 3.8.

SSLContext.options

An integer representing the set of SSL options enabled on this context. The default value is `OP_ALL`, but you can specify other options such as `OP_NO_SSLv2` by ORing them together.

Changed in version 3.6: `SSLContext.options` returns `Options` flags:

```
>>> ssl.create_default_context().options
<Options.OP_ALL|OP_NO_SSLv3|OP_NO_SSLv2|OP_NO_COMPRESSION:
2197947391>
```

Deprecated since version 3.7: All `OP_NO_SSL*` and `OP_NO_TLS*` options have been deprecated since Python 3.7. Use `SSLContext.minimum_version` and `SSLContext.maximum_version` instead.

SSLContext.post_handshake_auth

Enable TLS 1.3 post-handshake client authentication. Post-handshake auth is disabled by default and a server can only request a TLS client certificate during the initial handshake. When enabled, a server may request a TLS client certificate at any time after the handshake.

When enabled on client-side sockets, the client signals the server that it supports post-handshake authentication.

When enabled on server-side sockets, `SSLContext.verify_mode` must be set to `CERT_OPTIONAL` or `CERT_REQUIRED`, too. The actual client cert exchange is delayed until `SSLSocket.verify_client_post_handshake()` is called and some I/O is performed.

Added in version 3.8.

SSLContext.protocol

The protocol version chosen when constructing the context. This attribute is read-only.

SSLContext.hostname_checks_common_name

Whether `check_hostname` falls back to verify the cert's subject common name in the absence of a subject alternative name extension (default: true).

Added in version 3.7.

Changed in version 3.10: The flag had no effect with OpenSSL before version 1.1.1l. Python 3.8.9, 3.9.3, and 3.10 include workarounds for previous versions.

SSLContext.security_level

An integer representing the security level [https://docs.openssl.org/master/man3/SSL_CTX_get_security_level/] for the context. This attribute is read-only.

Added in version 3.10.

SSLContext.verify_flags

The flags for certificate verification operations. You can set flags like `VERIFY_CRL_CHECK_LEAF` by ORing them together. By default OpenSSL does neither require nor verify certificate revocation lists (CRLs).

Added in version 3.4.

Changed in version 3.6: **`SSLContext.verify_flags`** returns **`VerifyFlags`** flags:

```
>>> ssl.create_default_context().verify_flags
<VerifyFlags.VERIFY_X509_TRUSTED_FIRST: 32768>
```

SSLContext.verify_mode

Whether to try to verify other peers’ certificates and how to behave if verification fails. This attribute must be one of **`CERT_NONE`**, **`CERT_OPTIONAL`** or **`CERT_REQUIRED`**.

Changed in version 3.6: **`SSLContext.verify_mode`** returns **`VerifyMode`** enum:

```
>>> ssl.create_default_context().verify_mode
<VerifyMode.CERT_REQUIRED: 2>
```

SSLContext.set_psk_client_callback(*callback*)

Enables TLS-PSK (pre-shared key) authentication on a client-side connection.

In general, certificate based authentication should be preferred over this method.

The parameter `callback` is a callable object with the signature: `def callback(hint: str | None) -> tuple[str | None, bytes]`. The `hint` parameter is an optional identity hint sent by the server. The return value is a tuple in the form (client-identity, psk). Client-identity is an optional string which may be used by the server to select a corresponding PSK for the client. The string must be less than or equal to `256` octets when UTF-8 encoded. PSK is a bytes-like object representing the pre-shared key. Return a zero length PSK to reject the connection.

Setting `callback` to **`None`** removes any existing callback.

Note

When using TLS 1.3:

- the `hint` parameter is always `None`.
- client-identity must be a non-empty string.

Example usage:

```
context = ssl.SSLContext(ssl.PROTOCOL_TLS_CLIENT)
context.check_hostname = False
context.verify_mode = ssl.CERT_NONE
context.maximum_version = ssl.TLSVersion.TLSv1_2
context.set_ciphers('PSK')

# A simple lambda:
psk = bytes.fromhex('c0ffee')
context.set_psk_client_callback(lambda hint: (None, psk))

# A table using the hint from the server:
psk_table = { 'ServerId_1': bytes.fromhex('c0ffee'),
              'ServerId_2': bytes.fromhex('facade')
}
def callback(hint):
    return 'ClientId_1', psk_table.get(hint, b'')
context.set_psk_client_callback(callback)
```

This method will raise `NotImplementedError` if `HAS_PSK` is `False`.

Added in version 3.13.

SSLContext.set_psk_server_callback(*callback*, *identity_hint=None*)

Enables TLS-PSK (pre-shared key) authentication on a server-side connection.

In general, certificate based authentication should be preferred over this method.

The parameter `callback` is a callable object with the signature: `def callback(identity: str | None) -> bytes`. The `identity` parameter is an optional identity sent by the client which can be used to select a corresponding PSK. The return value is a bytes-like object representing the pre-shared key. Return a zero length PSK to reject the connection.

Setting `callback` to **`None`** removes any existing callback.

The parameter `identity_hint` is an optional identity hint string sent to the client. The string must be less than or equal to `256` octets when UTF-8 encoded.

Note

When using TLS 1.3 the `identity_hint` parameter is not sent to the client.

Example usage:

```
context = ssl.SSLContext(ssl.PROTOCOL_TLS_SERVER)
context.maximum_version = ssl.TLSVersion.TLSv1_2
context.set_ciphers('PSK')

# A simple lambda:
psk = bytes.fromhex('c0ffee')
context.set_psk_server_callback(lambda identity: psk)

# A table using the identity of the client:
psk_table = { 'ClientId_1': bytes.fromhex('c0ffee'),
              'ClientId_2': bytes.fromhex('facade')
}
def callback(identity):
    return psk_table.get(identity, b'')
context.set_psk_server_callback(callback, 'ServerId_1')
```

This method will raise **`NotImplementedError`** if **`HAS_PSK`** is `False`.

Added in version 3.13.

Certificates

Certificates in general are part of a public-key / private-key system. In this system, each *principal*, (which may be a machine, or a person, or an organization) is assigned a unique two-part encryption key. One part of the key is public, and is called the *public key*; the other part is kept secret, and is called the *private key*. The two parts are related, in that if you encrypt a message with one of the parts, you can decrypt it with the other part, and **only** with the other part.

A certificate contains information about two principals. It contains the name of a *subject*, and the subject's public key. It also contains a statement by a second principal, the *issuer*, that the subject is who they claim to be, and that this is indeed the subject's public key. The issuer's statement is signed with the issuer's private key, which only the issuer knows. However, anyone can verify the issuer's statement by finding the issuer's public key, decrypting the statement with it, and comparing it to the other information in the certificate. The certificate also contains information about the time period over which it is valid. This is expressed as two fields, called "notBefore" and "notAfter".

In the Python use of certificates, a client or server can use a certificate to prove who they are. The other side of a network connection can also be required to produce a certificate, and that certificate can be validated to the satisfaction of the client or server that requires such validation. The connection attempt can be set to raise an exception if the validation fails. Validation is done automatically, by the underlying OpenSSL framework; the application need not concern itself with its mechanics. But the application does usually need to provide sets of certificates to allow this process to take place.

Python uses files to contain certificates. They should be formatted as "PEM" (see **RFC 1422** [https://datatracker.ietf.org/doc/html/rfc1422.html]), which is a base-64 encoded form wrapped with a header line and a footer line:

```
-----BEGIN CERTIFICATE-----
... (certificate in base64 PEM encoding) ...
-----END CERTIFICATE-----
```

Certificate chains

The Python files which contain certificates can contain a sequence of certificates, sometimes called a *certificate chain*. This chain should start with the specific certificate for the principal who “is” the client or server, and then the certificate for the issuer of that certificate, and then the certificate for the issuer of *that* certificate, and so on up the chain till you get to a certificate which is *self-signed*, that is, a certificate which has the same subject and issuer, sometimes called a *root certificate*. The certificates should just be concatenated together in the certificate file. For example, suppose we had a three certificate chain, from our server certificate to the certificate of the certification authority that signed our server certificate, to the root certificate of the agency which issued the certification authority’s certificate:

```
-----BEGIN CERTIFICATE-----
... (certificate for your server)...
-----END CERTIFICATE-----
-----BEGIN CERTIFICATE-----
... (the certificate for the CA)...
-----END CERTIFICATE-----
-----BEGIN CERTIFICATE-----
... (the root certificate for the CA's issuer)...
-----END CERTIFICATE-----
```

CA certificates

If you are going to require validation of the other side of the connection’s certificate, you need to provide a “CA certs” file, filled with the certificate chains for each issuer you are willing to trust. Again, this file just contains these chains concatenated together. For validation, Python will use the first chain it finds in the file which matches. The platform’s certificates file can be used by calling `SSLContext.load_default_certs()`, this is done automatically with `create_default_context()`.

Combined key and certificate

Often the private key is stored in the same file as the certificate; in this case, only the `certfile` parameter to `SSLContext.load_cert_chain()` needs to be passed. If the private key is stored with the certificate, it should come before the first certificate in the certificate chain:

```
-----BEGIN RSA PRIVATE KEY-----
... (private key in base64 encoding) ...
-----END RSA PRIVATE KEY-----
-----BEGIN CERTIFICATE-----
... (certificate in base64 PEM encoding) ...
-----END CERTIFICATE-----
```

Self-signed certificates

If you are going to create a server that provides SSL-encrypted connection services, you will need to acquire a certificate for that service. There are many ways of acquiring appropriate certificates, such as buying one from a certification authority. Another common practice is to generate a self-signed certificate. The simplest way to do this is with the OpenSSL package, using something like the following:

```
% openssl req -new -x509 -days 365 -nodes -out cert.pem -keyout
cert.pem
Generating a 1024 bit RSA private key
.......++++++
.............................++++++
writing new private key to 'cert.pem'
-----
You are about to be asked to enter information that will be
incorporated
into your certificate request.
```

```
What you are about to enter is what is called a Distinguished
Name or a DN.
There are quite a few fields but you can leave some blank
For some fields there will be a default value,
If you enter '.', the field will be left blank.
-----
Country Name (2 letter code) [AU]:US
State or Province Name (full name) [Some-State]:MyState
Locality Name (eg, city) []:Some City
Organization Name (eg, company) [Internet Widgits Pty Ltd]:My
Organization, Inc.
Organizational Unit Name (eg, section) []:My Group
Common Name (eg, YOUR name)
[]:myserver.mygroup.myorganization.com
Email Address []:ops@myserver.mygroup.myorganization.com
%
```

The disadvantage of a self-signed certificate is that it is its own root certificate, and no one else will have it in their cache of known (and trusted) root certificates.

Examples

Testing for SSL support

To test for the presence of SSL support in a Python installation, user code should use the following idiom:

```
try:
    import ssl
except ImportError:
    pass
else:
    ...  # do something that requires SSL support
```

Client-side operation

This example creates a SSL context with the recommended security settings for client sockets, including automatic certificate verification:

```
>>> context = ssl.create_default_context()
```

If you prefer to tune security settings yourself, you might create a context from scratch (but beware that you might not get the settings right):

```
>>> context = ssl.SSLContext(ssl.PROTOCOL_TLS_CLIENT)
>>> context.load_verify_locations("/etc/ssl/certs/ca-bundle.crt")
```

(this snippet assumes your operating system places a bundle of all CA certificates in `/etc/ssl/certs/ca-bundle.crt`; if not, you'll get an error and have to adjust the location)

The **`PROTOCOL_TLS_CLIENT`** protocol configures the context for cert validation and hostname verification. **`verify_mode`** is set to **`CERT_REQUIRED`** and **`check_hostname`** is set to `True`. All other protocols create SSL contexts with insecure defaults.

When you use the context to connect to a server, **`CERT_REQUIRED`** and **`check_hostname`** validate the server certificate: it ensures that the server certificate was signed with one of the CA certificates, checks the signature for correctness, and verifies other properties like validity and identity of the hostname:

```
>>> conn = context.wrap_socket(socket.socket(socket.AF_INET),
...                            server_hostname="www.python.org")
>>> conn.connect(("www.python.org", 443))
```

You may then fetch the certificate:

```
>>> cert = conn.getpeercert()
```

Visual inspection shows that the certificate does identify the desired service (that is, the HTTPS host `www.python.org`):

```
>>> pprint.pprint(cert)
{'OCSP': ('http://ocsp.digicert.com',),
 'caIssuers':
('http://cacerts.digicert.com/DigiCertSHA2ExtendedValidationServe
rCA.crt',),
 'crlDistributionPoints': ('http://crl3.digicert.com/sha2-ev-
server-g1.crl',
                           'http://crl4.digicert.com/sha2-ev-
server-g1.crl'),
 'issuer': ((('countryName', 'US'),),
            (('organizationName', 'DigiCert Inc'),),
            (('organizationalUnitName', 'www.digicert.com'),),
            (('commonName', 'DigiCert SHA2 Extended Validation
Server CA'),)),
 'notAfter': 'Sep  9 12:00:00 2016 GMT',
 'notBefore': 'Sep  5 00:00:00 2014 GMT',
 'serialNumber': '01BB6F00122B177F36CAB49CEA8B6B26',
 'subject': ((('businessCategory', 'Private Organization'),),
             (('1.3.6.1.4.1.311.60.2.1.3', 'US'),),
             (('1.3.6.1.4.1.311.60.2.1.2', 'Delaware'),),
             (('serialNumber', '3359300'),),
             (('streetAddress', '16 Allen Rd'),),
             (('postalCode', '03894-4801'),),
             (('countryName', 'US'),),
             (('stateOrProvinceName', 'NH'),),
             (('localityName', 'Wolfeboro'),),
             (('organizationName', 'Python Software
Foundation'),),
             (('commonName', 'www.python.org'),)),
 'subjectAltName': (('DNS', 'www.python.org'),
                    ('DNS', 'python.org'),
                    ('DNS', 'pypi.org'),
                    ('DNS', 'docs.python.org'),
                    ('DNS', 'testpypi.org'),
                    ('DNS', 'bugs.python.org'),
                    ('DNS', 'wiki.python.org'),
                    ('DNS', 'hg.python.org'),
                    ('DNS', 'mail.python.org'),
                    ('DNS', 'packaging.python.org'),
                    ('DNS', 'pythonhosted.org'),
                    ('DNS', 'www.pythonhosted.org'),
                    ('DNS', 'test.pythonhosted.org'),
                    ('DNS', 'us.pycon.org'),
                    ('DNS', 'id.python.org')),
 'version': 3}
```

Now the SSL channel is established and the certificate verified, you can proceed to talk with the server:

```
>>> conn.sendall(b"HEAD / HTTP/1.0\r\nHost: linuxfr.org\r\n\r\n")
>>> pprint.pprint(conn.recv(1024).split(b"\r\n"))
[b'HTTP/1.1 200 OK',
 b'Date: Sat, 18 Oct 2014 18:27:20 GMT',
 b'Server: nginx',
 b'Content-Type: text/html; charset=utf-8',
 b'X-Frame-Options: SAMEORIGIN',
 b'Content-Length: 45679',
 b'Accept-Ranges: bytes',
 b'Via: 1.1 varnish',
 b'Age: 2188',
 b'X-Served-By: cache-lcy1134-LCY',
 b'X-Cache: HIT',
 b'X-Cache-Hits: 11',
 b'Vary: Cookie',
 b'Strict-Transport-Security: max-age=63072000;
includeSubDomains',
 b'Connection: close',
 b'',
 b'']
```

See the discussion of Security considerations below.

Server-side operation

For server operation, typically you’ll need to have a server certificate, and private key, each in a file. You’ll first create a context holding the key and the certificate, so that clients can check your authenticity. Then you’ll open a socket, bind it to a port, call **`listen()`** on it, and start waiting for clients to connect:

```
import socket, ssl

context = ssl.create_default_context(ssl.Purpose.CLIENT_AUTH)
context.load_cert_chain(certfile="mycertfile",
keyfile="mykeyfile")

bindsocket = socket.socket()
bindsocket.bind(('myaddr.example.com', 10023))
bindsocket.listen(5)
```

When a client connects, you’ll call **`accept()`** on the socket to get the new socket from the other end, and use the context’s **`SSLContext.wrap_socket()`** method to create a server-side SSL socket for the connection:

```
while True:
    newsocket, fromaddr = bindsocket.accept()
    connstream = context.wrap_socket(newsocket, server_side=True)
    try:
        deal_with_client(connstream)
    finally:
        connstream.shutdown(socket.SHUT_RDWR)
        connstream.close()
```

Then you’ll read data from the `connstream` and do something with it till you are finished with the client (or the client is finished with you):

```
def deal_with_client(connstream):
    data = connstream.recv(1024)
    # empty data means the client is finished with us
    while data:
        if not do_something(connstream, data):
            # we'll assume do_something returns False
            # when we're finished with client
            break
        data = connstream.recv(1024)
    # finished with client
```

And go back to listening for new client connections (of course, a real server would probably handle each client connection in a separate thread, or put the sockets in non-blocking mode and use an event loop).

Notes on non-blocking sockets

SSL sockets behave slightly different than regular sockets in non-blocking mode. When working with non-blocking sockets, there are thus several things you need to be aware of:

- Most `SSLSocket` methods will raise either `SSLWantWriteError` or `SSLWantReadError` instead of `BlockingIOError` if an I/O operation would block. `SSLWantReadError` will be raised if a read operation on the underlying socket is necessary, and `SSLWantWriteError` for a write operation on the underlying socket. Note that attempts to *write* to an SSL socket may require *reading* from the underlying socket first, and attempts to *read* from the SSL socket may require a prior *write* to the underlying socket.

  Changed in version 3.5: In earlier Python versions, the **`SSLSocket.send()`** method returned zero instead of raising `SSLWantWriteError` or `SSLWantReadError`.

- Calling `select()` tells you that the OS-level socket can be read from (or written to), but it does not imply that there is sufficient data at the upper SSL layer. For example, only part of an SSL frame might have arrived. Therefore, you must be ready to handle **`SSLSocket.recv()`** and **`SSLSocket.send()`** failures, and retry after another call to `select()`.

- Conversely, since the SSL layer has its own framing, a SSL socket may still have data available for reading without `select()` being aware of it. Therefore, you should first call **`SSLSocket.recv()`** to drain any potentially available data, and then only block on a `select()` call if still necessary.

  (of course, similar provisions apply when using other primitives such as `poll()`, or those in the `selectors` module)

- The SSL handshake itself will be non-blocking: the `SSLSocket.do_handshake()` method has to be retried until it returns successfully. Here is a synopsis using `select()` to wait for the socket’s readiness:

  ```
  while True:
      try:
  ```

```
            sock.do_handshake()
            break
        except ssl.SSLWantReadError:
            select.select([sock], [], [])
        except ssl.SSLWantWriteError:
            select.select([], [sock], [])
```

See also

The `asyncio` module supports non-blocking SSL sockets and provides a higher level Streams API. It polls for events using the `selectors` module and handles `SSLWantWriteError`, `SSLWantReadError` and `BlockingIOError` exceptions. It runs the SSL handshake asynchronously as well.

Memory BIO Support

Added in version 3.5.

Ever since the SSL module was introduced in Python 2.6, the `SSLSocket` class has provided two related but distinct areas of functionality:

- SSL protocol handling
- Network IO

The network IO API is identical to that provided by `socket.socket`, from which `SSLSocket` also inherits. This allows an SSL socket to be used as a drop-in replacement for a regular socket, making it very easy to add SSL support to an existing application.

Combining SSL protocol handling and network IO usually works well, but there are some cases where it doesn’t. An example is async IO frameworks that want to use a different IO multiplexing model than the “select/poll on a file descriptor” (readiness based) model that is assumed by `socket.socket` and by the internal OpenSSL socket IO routines. This is mostly relevant for platforms like Windows where this model is not efficient. For this purpose, a reduced scope variant of `SSLSocket` called `SSLObject` is provided.

class ssl.SSLObject

A reduced-scope variant of `SSLSocket` representing an SSL protocol instance that does not contain any network IO methods. This class is typically used by framework authors that want to implement asynchronous IO for SSL through memory buffers.

This class implements an interface on top of a low-level SSL object as implemented by OpenSSL. This object captures the state of an SSL connection but does not provide any network IO itself. IO needs to be performed through separate “BIO” objects which are OpenSSL’s IO abstraction layer.

This class has no public constructor. An `SSLObject` instance must be created using the `wrap_bio()` method. This method will create the `SSLObject` instance and bind it to a pair of BIOs. The *incoming* BIO is used to pass data from Python to the SSL protocol instance, while the *outgoing* BIO is used to pass data the other way around.

The following methods are available:

- `context`
- `server_side`
- `server_hostname`
- `session`
- `session_reused`
- `read()`
- `write()`
- `getpeercert()`
- `get_verified_chain()`

- `get_unverified_chain()`
- `selected_alpn_protocol()`
- `selected_npn_protocol()`
- `cipher()`
- `shared_ciphers()`
- `compression()`
- `pending()`
- `do_handshake()`
- `verify_client_post_handshake()`
- `unwrap()`
- `get_channel_binding()`
- `version()`

When compared to `SSLSocket`, this object lacks the following features:

- Any form of network IO; `recv()` and `send()` read and write only to the underlying `MemoryBIO` buffers.
- There is no *do_handshake_on_connect* machinery. You must always manually call `do_handshake()` to start the handshake.
- There is no handling of *suppress_ragged_eofs*. All end-of-file conditions that are in violation of the protocol are reported via the `SSLEOFError` exception.
- The method `unwrap()` call does not return anything, unlike for an SSL socket where it returns the underlying socket.
- The *server_name_callback* callback passed to `SSLContext.set_servername_callback()` will get an `SSLObject` instance instead of a `SSLSocket` instance as its first parameter.

Some notes related to the use of `SSLObject`:

- All IO on an `SSLObject` is non-blocking. This means that for example `read()` will raise an `SSLWantReadError` if it needs more data than the incoming BIO has available.

Changed in version 3.7: `SSLObject` instances must be created with `wrap_bio()`. In earlier versions, it was possible to create instances directly. This was never documented or officially supported.

An SSLObject communicates with the outside world using memory buffers. The class `MemoryBIO` provides a memory buffer that can be used for this purpose. It wraps an OpenSSL memory BIO (Basic IO) object:

class ssl.MemoryBIO

A memory buffer that can be used to pass data between Python and an SSL protocol instance.

pending

Return the number of bytes currently in the memory buffer.

eof

A boolean indicating whether the memory BIO is current at the end-of-file position.

read(*n=-1*, */*)

Read up to *n* bytes from the memory buffer. If *n* is not specified or negative, all bytes are returned.

write(*buf*, */*)

Write the bytes from *buf* to the memory BIO. The *buf* argument must be an object supporting the buffer protocol.

The return value is the number of bytes written, which is always equal to the length of *buf*.

write_eof()

Write an EOF marker to the memory BIO. After this method has been called, it is illegal to call `write()`. The attribute `eof` will become true after all data currently in the buffer has been read.

SSL session

Added in version 3.6.

class ssl.SSLSession

Session object used by `session`.

id

time

timeout

ticket_lifetime_hint

has_ticket

Security considerations

Best defaults

For **client use**, if you don’t have any special requirements for your security policy, it is highly recommended that you use the `create_default_context()` function to create your SSL context. It will load the system’s trusted CA certificates, enable certificate validation and hostname checking, and try to choose reasonably secure protocol and cipher settings.

For example, here is how you would use the `smtplib.SMTP` class to create a trusted, secure connection to a SMTP server:

```
>>> import ssl, smtplib
>>> smtp = smtplib.SMTP("mail.python.org", port=587)
>>> context = ssl.create_default_context()
>>> smtp.starttls(context=context)
(220, b'2.0.0 Ready to start TLS')
```

If a client certificate is needed for the connection, it can be added with `SSLContext.load_cert_chain()`.

By contrast, if you create the SSL context by calling the `SSLContext` constructor yourself, it will not have certificate validation nor hostname checking enabled by default. If you do so, please read the paragraphs below to achieve a good security level.

Manual settings

Verifying certificates

When calling the `SSLContext` constructor directly, `CERT_NONE` is the default. Since it does not authenticate the other peer, it can be insecure, especially in client mode where most of the time you would like to ensure the authenticity of the server you're talking to. Therefore, when in client mode, it is highly recommended to use `CERT_REQUIRED`. However, it is in itself not sufficient; you also have to check that the server certificate, which can be obtained by calling `SSLSocket.getpeercert()`, matches the desired service. For many protocols and applications, the service can be identified by the hostname. This common check is automatically performed when `SSLContext.check_hostname` is enabled.

Changed in version 3.7: Hostname matchings is now performed by OpenSSL. Python no longer uses `match_hostname()`.

In server mode, if you want to authenticate your clients using the SSL layer (rather than using a higher-level authentication mechanism), you'll also have to specify `CERT_REQUIRED` and similarly check the client certificate.

Protocol versions

SSL versions 2 and 3 are considered insecure and are therefore dangerous to use. If you want maximum compatibility between clients and servers, it is recommended to use `PROTOCOL_TLS_CLIENT` or `PROTOCOL_TLS_SERVER` as the protocol version. SSLv2 and SSLv3 are disabled by default.

```
>>> client_context = ssl.SSLContext(ssl.PROTOCOL_TLS_CLIENT)
>>> client_context.minimum_version = ssl.TLSVersion.TLSv1_3
>>> client_context.maximum_version = ssl.TLSVersion.TLSv1_3
```

The SSL context created above will only allow TLSv1.3 and later (if supported by your system) connections to a server. `PROTOCOL_TLS_CLIENT` implies certificate validation and hostname checks by default. You have to load certificates into the context.

Cipher selection

If you have advanced security requirements, fine-tuning of the ciphers enabled when negotiating a SSL session is possible through the `SSLContext.set_ciphers()` method. Starting from Python 3.2.3, the ssl module disables certain weak ciphers by default, but you may want to further restrict the cipher choice. Be sure to read OpenSSL’s documentation about the cipher list format [https://docs.openssl.org/1.1.1/man1/ciphers/#cipher-list-format]. If you want to check which ciphers are enabled by a given cipher list, use `SSLContext.get_ciphers()` or the `openssl ciphers` command on your system.

Multi-processing

If using this module as part of a multi-processed application (using, for example the `multiprocessing` or `concurrent.futures` modules), be aware that OpenSSL’s internal random number generator does not properly handle forked processes. Applications must change the PRNG state of the parent process if they use any SSL feature with `os.fork()`. Any successful call of `RAND_add()` or `RAND_bytes()` is sufficient.

TLS 1.3

Added in version 3.7.

The TLS 1.3 protocol behaves slightly differently than previous version of TLS/SSL. Some new TLS 1.3 features are not yet available.

- TLS 1.3 uses a disjunct set of cipher suites. All AES-GCM and ChaCha20 cipher suites are enabled by default. The method `SSLContext.set_ciphers()` cannot enable or disable any TLS 1.3 ciphers yet, but `SSLContext.get_ciphers()` returns them.
- Session tickets are no longer sent as part of the initial handshake and are handled differently. `SSLSocket.session` and `SSLSession` are not compatible with TLS 1.3.
- Client-side certificates are also no longer verified during the initial handshake. A server can request a certificate at any time. Clients process certificate requests while they send or receive application data from the server.
- TLS 1.3 features like early data, deferred TLS client cert request, signature algorithm configuration, and rekeying are not supported yet.

See also

Class `socket.socket`
Documentation of underlying `socket` class

SSL/TLS Strong Encryption: An Introduction
[https://httpd.apache.org/docs/trunk/en/ssl/ssl_intro.html]
Intro from the Apache HTTP Server documentation

RFC 1422: Privacy Enhancement for Internet Electronic Mail: Part II: Certificate-Based Key Management **[https://datatracker.ietf.org/doc/html/rfc1422.html]**
Steve Kent

RFC 4086: Randomness Requirements for Security **[https://datatracker.ietf.org/doc/html/rfc4086.html]**
Donald E. Eastlake, Jeffrey I. Schiller, Steve Crocker

RFC 5280: Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile **[https://datatracker.ietf.org/doc/html/rfc5280.html]**
David Cooper et al.

RFC 5246: The Transport Layer Security (TLS) Protocol Version 1.2 **[https://datatracker.ietf.org/doc/html/rfc5246.html]**
Tim Dierks and Eric Rescorla.

RFC 6066: Transport Layer Security (TLS) Extensions **[https://datatracker.ietf.org/doc/html/rfc6066.html]**
Donald E. Eastlake

IANA TLS: Transport Layer Security (TLS) Parameters **[https://www.iana.org/assignments/tls-parameters/tls-parameters.xml]**
IANA

RFC 7525: Recommendations for Secure Use of Transport Layer Security (TLS) and Datagram Transport Layer Security (DTLS) **[https://datatracker.ietf.org/doc/html/rfc7525.html]**
IETF

Mozilla’s Server Side TLS recommendations **[https://wiki.mozilla.org/Security/Server_Side_TLS]**
Mozilla

`select` — Waiting for I/O completion

This module provides access to the **`select()`** and **`poll()`** functions available in most operating systems, **`devpoll()`** available on Solaris and derivatives, **`epoll()`** available on Linux 2.5+ and **`kqueue()`** available on most BSD. Note that on Windows, it only works for sockets; on other operating systems, it also works for other file types (in particular, on Unix, it works on pipes). It cannot be used on regular files to determine whether a file has grown since it was last read.

Note

The **`selectors`** module allows high-level and efficient I/O multiplexing, built upon the **`select`** module primitives. Users are encouraged to use the **`selectors`** module instead, unless they want precise control over the OS-level primitives used.

Availability: not WASI.

This module does not work or is not available on WebAssembly. See WebAssembly platforms for more information.

The module defines the following:

exception select.error

A deprecated alias of **`OSError`**.

Changed in version 3.3: Following **PEP 3151** [https://peps.python.org/pep-3151/], this class was made an alias of **`OSError`**.

select.devpoll()

(Only supported on Solaris and derivatives.) Returns a `/dev/poll` polling object; see section /dev/poll Polling Objects below for the methods supported by devpoll objects.

`devpoll()` objects are linked to the number of file descriptors allowed at the time of instantiation. If your program reduces this value, **`devpoll()`** will fail. If your program increases this value, **`devpoll()`** may return an incomplete list of active file descriptors.

The new file descriptor is non-inheritable.

Added in version 3.3.

Changed in version 3.4: The new file descriptor is now non-inheritable.

select.epoll(*sizehint=-1*, *flags=0*)

(Only supported on Linux 2.5.44 and newer.) Return an edge polling object, which can be used as Edge or Level Triggered interface for I/O events.

sizehint informs epoll about the expected number of events to be registered. It must be positive, or `-1` to use the default. It is only used on older systems where **`epoll_create1()`** is not available; otherwise it has no effect (though its value is still checked).

flags is deprecated and completely ignored. However, when supplied, its value must be `0` or `select.EPOLL_CLOEXEC`, otherwise `OSError` is raised.

See the Edge and Level Trigger Polling (epoll) Objects section below for the methods supported by epolling objects.

`epoll` objects support the context management protocol: when used in a **`with`** statement, the new file descriptor is automatically closed at the end of the block.

The new file descriptor is non-inheritable.

Changed in version 3.3: Added the *flags* parameter.

Changed in version 3.4: Support for the **`with`** statement was added. The new file descriptor is now non-inheritable.

Deprecated since version 3.4: The *flags* parameter. `select.EPOLL_CLOEXEC` is used by default now. Use **`os.set_inheritable()`** to make the file descriptor inheritable.

select.poll()

(Not supported by all operating systems.) Returns a polling object, which supports registering and unregistering file descriptors, and then polling them for I/O events; see section Polling Objects below for the methods supported by polling objects.

select.kqueue()

(Only supported on BSD.) Returns a kernel queue object; see section Kqueue Objects below for the methods supported by kqueue objects.

The new file descriptor is non-inheritable.

Changed in version 3.4: The new file descriptor is now non-inheritable.

select.kevent(*ident*, *filter=KQ_FILTER_READ*, *flags=KQ_EV_ADD*, *fflags=0*, *data=0*, *udata=0*)

(Only supported on BSD.) Returns a kernel event object; see section Kevent Objects below for the methods supported by kevent objects.

select.select(*rlist*, *wlist*, *xlist*, *timeout=None*)

This is a straightforward interface to the Unix **`select()`** system call. The first three arguments are iterables of ‘waitable objects’: either integers representing file descriptors or objects with a parameterless method named **`fileno()`** returning such an integer:

- *rlist*: wait until ready for reading
- *wlist*: wait until ready for writing
- *xlist*: wait for an “exceptional condition” (see the manual page for what your system considers such a condition)

Empty iterables are allowed, but acceptance of three empty iterables is platform-dependent. (It is known to work on Unix but not on Windows.) The optional *timeout* argument specifies a time-out as a floating-point number in seconds. When the *timeout* argument is omitted or `None`, the function blocks until at least one file descriptor is ready. A time-out value of zero specifies a poll and never blocks.

The return value is a triple of lists of objects that are ready: subsets of the first three arguments. When the time-out is reached without a file descriptor becoming ready, three empty lists are returned.

Among the acceptable object types in the iterables are Python file objects (e.g. `sys.stdin`, or objects returned by **`open()`** or **`os.popen()`**), socket objects returned by **`socket.socket()`**. You may also define a *wrapper* class yourself, as long as it has an appropriate **`fileno()`** method (that really returns a file descriptor, not just a random integer).

Note

File objects on Windows are not acceptable, but sockets are. On Windows, the underlying **`select()`** function is provided by the WinSock library, and does not handle file descriptors that don't originate from WinSock.

Changed in version 3.5: The function is now retried with a recomputed timeout when interrupted by a signal, except if the signal handler raises an exception (see **PEP 475** [https://peps.python.org/pep-0475/] for the rationale), instead of raising **`InterruptedError`**.

select.PIPE_BUF

The minimum number of bytes which can be written without blocking to a pipe when the pipe has been reported as ready for writing by **`select()`**, **`poll()`** or another interface in this module. This doesn't apply to other kind of file-like objects such as sockets.

This value is guaranteed by POSIX to be at least 512.

Availability: Unix

Added in version 3.2.

`/dev/poll` Polling Objects

Solaris and derivatives have `/dev/poll`. While **`select()`** is *O*(*highest file descriptor*) and **`poll()`** is *O*(*number of file descriptors*), `/dev/poll` is *O*(*active file descriptors*).

`/dev/poll` behaviour is very close to the standard **`poll()`** object.

devpoll.close()

Close the file descriptor of the polling object.

Added in version 3.4.

devpoll.closed

`True` if the polling object is closed.

Added in version 3.4.

devpoll.fileno()

Return the file descriptor number of the polling object.

Added in version 3.4.

devpoll.register(*fd*[, *eventmask*])

Register a file descriptor with the polling object. Future calls to the **`poll()`** method will then check whether the file descriptor has any pending I/O events. *fd* can be either an integer, or an object with a **`fileno()`** method that returns an integer. File objects implement **`fileno()`**, so they can also be used as the argument.

eventmask is an optional bitmask describing the type of events you want to check for. The constants are the same that with **`poll()`** object. The default value is a combination of the constants **`POLLIN`**, **`POLLPRI`**, and **`POLLOUT`**.

Warning

Registering a file descriptor that's already registered is not an error, but the result is undefined. The appropriate action is to unregister or modify it first. This is an important difference compared with **`poll()`**.

devpoll.modify(*fd*[, *eventmask*])

This method does an [**`unregister()`**] followed by a [**`register()`**]. It is (a bit) more efficient that doing the same explicitly.

devpoll.unregister(*fd*)

Remove a file descriptor being tracked by a polling object. Just like the [**`register()`**] method, *fd* can be an integer or an object with a [**`fileno()`**] method that returns an integer.

Attempting to remove a file descriptor that was never registered is safely ignored.

devpoll.poll([*timeout*])

Polls the set of registered file descriptors, and returns a possibly empty list containing `(fd, event)` 2-tuples for the descriptors that have events or errors to report. *fd* is the file descriptor, and *event* is a bitmask with bits set for the reported events for that descriptor — **`POLLIN`** for waiting input, **`POLLOUT`** to indicate that the descriptor can be written to, and so forth. An empty list indicates that the call timed out and no file descriptors had any events to report. If *timeout* is given, it specifies the length of time in milliseconds which the system will wait for events before returning. If *timeout* is omitted, -1, or [**`None`**], the call will block until there is an event for this poll object.

Changed in version 3.5: The function is now retried with a recomputed timeout when interrupted by a signal, except if the signal handler raises an exception (see **PEP 475** [https://peps.python.org/pep-0475/] for the rationale), instead of raising `InterruptedError`.

Edge and Level Trigger Polling (epoll) Objects

https://linux.die.net/man/4/epoll

eventmask

Constant	Meaning
`EPOLLIN`	Available for read
`EPOLLOUT`	Available for write
`EPOLLPRI`	Urgent data for read
`EPOLLERR`	Error condition happened on the assoc. fd
`EPOLLHUP`	Hang up happened on the assoc. fd
`EPOLLET`	Set Edge Trigger behavior, the default is Level Trigger behavior
`EPOLLONESHOT`	Set one-shot behavior. After one event is pulled out, the fd is internally disabled

Constant	Meaning
EPOLLEXCLUSIVE	Wake only one epoll object when the associated fd has an event. The default (if this flag is not set) is to wake all epoll objects polling on a fd.
EPOLLRDHUP	Stream socket peer closed connection or shut down writing half of connection.
EPOLLRDNORM	Equivalent to **EPOLLIN**
EPOLLRDBAND	Priority data band can be read.
EPOLLWRNORM	Equivalent to **EPOLLOUT**
EPOLLWRBAND	Priority data may be written.
EPOLLMSG	Ignored.
EPOLLWAKEUP	Prevents sleep during event waiting.

Added in version 3.6: **EPOLLEXCLUSIVE** was added. It’s only supported by Linux Kernel 4.5 or later.

Added in version 3.14: **EPOLLWAKEUP** was added. It’s only supported by Linux Kernel 3.5 or later.

epoll.close()

Close the control file descriptor of the epoll object.

epoll.closed

`True` if the epoll object is closed.

epoll.fileno()

Return the file descriptor number of the control fd.

epoll.fromfd(*fd*)

Create an epoll object from a given file descriptor.

epoll.register(*fd*[, *eventmask*])

Register a fd descriptor with the epoll object.

epoll.modify(*fd*, *eventmask*)

Modify a registered file descriptor.

epoll.unregister(*fd*)

Remove a registered file descriptor from the epoll object.

Changed in version 3.9: The method no longer ignores the `EBADF` error.

epoll.poll(*timeout=None*, *maxevents=-1*)

Wait for events. timeout in seconds (float)

Changed in version 3.5: The function is now retried with a recomputed timeout when interrupted by a signal, except if the signal handler raises an exception (see **PEP 475** [https://peps.python.org/pep-0475/] for the rationale), instead of raising `InterruptedError`.

Polling Objects

The `poll()` system call, supported on most Unix systems, provides better scalability for network servers that service many, many clients at the same time. `poll()` scales better because the system call only requires listing the file descriptors of interest, while `select()` builds a bitmap, turns on bits for the fds of interest, and then afterward the whole bitmap has to be linearly scanned again. `select()` is *O*(*highest file descriptor*), while `poll()` is *O*(*number of file descriptors*).

poll.register(*fd*[, *eventmask*])

Register a file descriptor with the polling object. Future calls to the `poll()` method will then check whether the file descriptor has any pending I/O events. *fd* can be either an integer, or an object with a `fileno()` method that returns an integer. File objects implement `fileno()`, so they can also be used as the argument.

eventmask is an optional bitmask describing the type of events you want to check for, and can be a combination of the constants `POLLIN`, `POLLPRI`, and `POLLOUT`, described in the table below. If not specified, the default value used will check for all 3 types of events.

Constant	Meaning
`POLLIN`	There is data to read
`POLLPRI`	There is urgent data to read
`POLLOUT`	Ready for output: writing will not block
`POLLERR`	Error condition of some sort
`POLLHUP`	Hung up

Constant	Meaning
`POLLRDHUP`	Stream socket peer closed connection, or shut down writing half of connection
`POLLNVAL`	Invalid request: descriptor not open

Registering a file descriptor that's already registered is not an error, and has the same effect as registering the descriptor exactly once.

poll.modify(*fd*, *eventmask*)

Modifies an already registered fd. This has the same effect as `register(fd, eventmask)`. Attempting to modify a file descriptor that was never registered causes an `OSError` exception with errno **`ENOENT`** to be raised.

poll.unregister(*fd*)

Remove a file descriptor being tracked by a polling object. Just like the `register()` method, *fd* can be an integer or an object with a `fileno()` method that returns an integer.

Attempting to remove a file descriptor that was never registered causes a `KeyError` exception to be raised.

poll.poll([*timeout*])

Polls the set of registered file descriptors, and returns a possibly empty list containing `(fd, event)` 2-tuples for the descriptors that have events or errors to report. *fd* is the file descriptor, and *event* is a bitmask with bits set for the reported events for that descriptor — **`POLLIN`** for waiting input, **`POLLOUT`** to indicate that the descriptor can be written to, and so forth. An empty list indicates that the call timed out and no file descriptors had any events to report. If *timeout* is given, it specifies the length of time in milliseconds which the system will wait for events before returning. If *timeout* is omitted, negative, or `None`, the call will block until there is an event for this poll object.

Changed in version 3.5: The function is now retried with a recomputed timeout when interrupted by a signal, except if the signal handler raises an exception (see **PEP 475** [https://peps.python.org/pep-0475/] for the rationale), instead of raising `InterruptedError`.

Kqueue Objects

kqueue.close()

Close the control file descriptor of the kqueue object.

kqueue.closed

`True` if the kqueue object is closed.

kqueue.fileno()

Return the file descriptor number of the control fd.

kqueue.fromfd(*fd*)

Create a kqueue object from a given file descriptor.

kqueue.control(*changelist*, *max_events*[, *timeout*]) → eventlist

Low level interface to kevent

- changelist must be an iterable of kevent objects or `None`
- max_events must be 0 or a positive integer
- timeout in seconds (floats possible); the default is `None`, to wait forever

Changed in version 3.5: The function is now retried with a recomputed timeout when interrupted by a signal, except if the signal handler raises an exception (see **PEP 475** [https://peps.python.org/pep-0475/] for the rationale), instead of raising **`InterruptedError`**.

Kevent Objects

https://man.freebsd.org/cgi/man.cgi?query=kqueue&sektion=2

kevent.ident

Value used to identify the event. The interpretation depends on the filter but it’s usually the file descriptor. In the constructor ident can either be an int or an object with a **`fileno()`** method. kevent stores the integer internally.

kevent.filter

Name of the kernel filter.

Constant	**Meaning**
`KQ_FILTER_READ`	Takes a descriptor and returns whenever there is data available to read
`KQ_FILTER_WRITE`	Takes a descriptor and returns whenever there is data available to write
`KQ_FILTER_AIO`	AIO requests
`KQ_FILTER_VNODE`	Returns when one or more of the requested events watched in *fflag*

Constant	Meaning
	occurs
`KQ_FILTER_PROC`	Watch for events on a process id
`KQ_FILTER_NETDEV`	Watch for events on a network device [not available on macOS]
`KQ_FILTER_SIGNAL`	Returns whenever the watched signal is delivered to the process
`KQ_FILTER_TIMER`	Establishes an arbitrary timer

kevent.flags

Filter action.

Constant	Meaning
`KQ_EV_ADD`	Adds or modifies an event
`KQ_EV_DELETE`	Removes an event from the queue
`KQ_EV_ENABLE`	Permitscontrol() to returns the event
`KQ_EV_DISABLE`	Disablesevent
`KQ_EV_ONESHOT`	Removes event after first occurrence
`KQ_EV_CLEAR`	Reset the state after an event is retrieved
`KQ_EV_SYSFLAGS`	internal event
`KQ_EV_FLAG1`	internal event

Constant	Meaning
`KQ_EV_EOF`	Filter specific EOF condition
`KQ_EV_ERROR`	See return values

kevent.fflags
Filter specific flags.

`KQ_FILTER_READ` and **`KQ_FILTER_WRITE`** filter flags:

Constant	Meaning
`KQ_NOTE_LOWAT`	low water mark of a socket buffer

`KQ_FILTER_VNODE` filter flags:

Constant	Meaning
`KQ_NOTE_DELETE`	*unlink()* was called
`KQ_NOTE_WRITE`	a write occurred
`KQ_NOTE_EXTEND`	the file was extended
`KQ_NOTE_ATTRIB`	an attribute was changed
`KQ_NOTE_LINK`	the link count has changed
`KQ_NOTE_RENAME`	the file was renamed
`KQ_NOTE_REVOKE`	access to the file was revoked

`KQ_FILTER_PROC` filter flags:

Constant	Meaning
`KQ_NOTE_EXIT`	the process has exited
`KQ_NOTE_FORK`	the process has called *fork()*
`KQ_NOTE_EXEC`	the process has executed a new process
`KQ_NOTE_PCTRLMASK`	internal filter flag
`KQ_NOTE_PDATAMASK`	internal filter flag
`KQ_NOTE_TRACK`	follow a process across *fork()*
`KQ_NOTE_CHILD`	returned on the child process for *NOTE_TRACK*
`KQ_NOTE_TRACKERR`	unable to attach to a child

`KQ_FILTER_NETDEV` filter flags (not available on macOS):

Constant	Meaning
`KQ_NOTE_LINKUP`	link is up
`KQ_NOTE_LINKDOWN`	link is down
`KQ_NOTE_LINKINV`	link state is invalid

kevent.data

Filter specific data.

kevent.udata

User defined value.

`selectors` — High-level I/O multiplexing

Added in version 3.4.

Source code: Lib/selectors.py
[https://github.com/python/cpython/tree/3.14/Lib/selectors.py]

Introduction

This module allows high-level and efficient I/O multiplexing, built upon the **`select`** module primitives. Users are encouraged to use this module instead, unless they want precise control over the OS-level primitives used.

It defines a **`BaseSelector`** abstract base class, along with several concrete implementations (**`KqueueSelector`**, **`EpollSelector`**…), that can be used to wait for I/O readiness notification on multiple file objects. In the following, “file object” refers to any object with a **`fileno()`** method, or a raw file descriptor. See file object.

`DefaultSelector` is an alias to the most efficient implementation available on the current platform: this should be the default choice for most users.

Note

The type of file objects supported depends on the platform: on Windows, sockets are supported, but not pipes, whereas on Unix, both are supported (some other types may be supported as well, such as fifos or special file devices).

See also

`select`
Low-level I/O multiplexing module.

Availability: not WASI.

This module does not work or is not available on WebAssembly. See WebAssembly platforms for more information.

Classes

Classes hierarchy:

```
BaseSelector
+-- SelectSelector
+-- PollSelector
+-- EpollSelector
+-- DevpollSelector
+-- KqueueSelector
```

In the following, *events* is a bitwise mask indicating which I/O events should be waited for on a given file object. It can be a combination of the modules constants below:

Constant	Meaning
selectors.EVENT_READ	Available for read

Constant	**Meaning**
selectors.EVENT_WRITE	Available for write

class selectors.SelectorKey

A `SelectorKey` is a `namedtuple` used to associate a file object to its underlying file descriptor, selected event mask and attached data. It is returned by several `BaseSelector` methods.

fileobj

File object registered.

fd

Underlying file descriptor.

events

Events that must be waited for on this file object.

data

Optional opaque data associated to this file object: for example, this could be used to store a per-client session ID.

class selectors.BaseSelector

A `BaseSelector` is used to wait for I/O event readiness on multiple file objects. It supports file stream registration, unregistration, and a method to wait for I/O events on those streams, with an optional timeout. It’s an abstract base class, so cannot be instantiated. Use `DefaultSelector` instead, or one of `SelectSelector`, `KqueueSelector` etc. if you want to specifically use an implementation, and your platform supports it. **`BaseSelector`** and its concrete implementations support the context manager protocol.

abstractmethod register(*fileobj*, *events*, *data=None*)

Register a file object for selection, monitoring it for I/O events.

fileobj is the file object to monitor. It may either be an integer file descriptor or an object with a `fileno()` method. *events* is a bitwise mask of events to monitor. *data* is an opaque object.

This returns a new **`SelectorKey`** instance, or raises a **`ValueError`** in case of invalid event mask or file descriptor, or **`KeyError`** if the file object is already registered.

abstractmethod unregister(*fileobj*)

Unregister a file object from selection, removing it from monitoring. A file object shall be unregistered prior to being closed.

fileobj must be a file object previously registered.

This returns the associated **`SelectorKey`** instance, or raises a **`KeyError`** if *fileobj* is not registered. It will raise **`ValueError`** if *fileobj* is invalid (e.g. it has no `fileno()` method or its `fileno()` method has an invalid return value).

modify(*fileobj*, *events*, *data=None*)

Change a registered file object's monitored events or attached data.

This is equivalent to `BaseSelector.unregister(fileobj)` followed by `BaseSelector.register(fileobj, events,` `data)`, except that it can be implemented more efficiently.

This returns a new **`SelectorKey`** instance, or raises a **`ValueError`** in case of invalid event mask or file descriptor, or **`KeyError`** if the file object is not registered.

abstractmethod select(*timeout=None*)

Wait until some registered file objects become ready, or the timeout expires.

If `timeout > 0`, this specifies the maximum wait time, in seconds. If `timeout <= 0`, the call won't block, and will report the currently ready file objects. If *timeout* is `None`, the call will block until a monitored file object becomes ready.

This returns a list of `(key, events)` tuples, one for each ready file object.

key is the **`SelectorKey`** instance corresponding to a ready file object. *events* is a bitmask of events ready on this file object.

Note

This method can return before any file object becomes ready or the timeout has elapsed if the current process receives a signal: in this case, an empty list will be returned.

Changed in version 3.5: The selector is now retried with a recomputed timeout when interrupted by a signal if the signal handler did not raise an exception (see **PEP 475** [https://peps.python.org/pep-0475/] for the rationale), instead of returning an empty list of events before the timeout.

close()

Close the selector.

This must be called to make sure that any underlying resource is freed. The selector shall not be used once it has been closed.

get_key(*fileobj*)

Return the key associated with a registered file object.

This returns the `SelectorKey` instance associated to this file object, or raises `KeyError` if the file object is not registered.

abstractmethod get_map()

Return a mapping of file objects to selector keys.

This returns a `Mapping` instance mapping registered file objects to their associated `SelectorKey` instance.

class selectors.DefaultSelector

The default selector class, using the most efficient implementation available on the current platform. This should be the default choice for most users.

class selectors.SelectSelector

`select.select()`-based selector.

class selectors.PollSelector

`select.poll()`-based selector.

class selectors.EpollSelector

`select.epoll()`-based selector.

fileno()

This returns the file descriptor used by the underlying `select.epoll()` object.

class selectors.DevpollSelector

`select.devpoll()`-based selector.

fileno()

This returns the file descriptor used by the underlying `select.devpoll()` object.

Added in version 3.5.

class selectors.KqueueSelector

`select.kqueue()`-based selector.

fileno()

This returns the file descriptor used by the underlying `select.kqueue()` object.

Examples

Here is a simple echo server implementation:

```
import selectors
import socket

sel = selectors.DefaultSelector()

def accept(sock, mask):
    conn, addr = sock.accept()  # Should be ready
    print('accepted', conn, 'from', addr)
    conn.setblocking(False)
    sel.register(conn, selectors.EVENT_READ, read)

def read(conn, mask):
    data = conn.recv(1000)  # Should be ready
    if data:
        print('echoing', repr(data), 'to', conn)
        conn.send(data)  # Hope it won't block
    else:
        print('closing', conn)
        sel.unregister(conn)
        conn.close()

sock = socket.socket()
sock.bind(('localhost', 1234))
sock.listen(100)
sock.setblocking(False)
sel.register(sock, selectors.EVENT_READ, accept)

while True:
    events = sel.select()
    for key, mask in events:
        callback = key.data
        callback(key.fileobj, mask)
```

`signal` — Set handlers for asynchronous events

Source code: Lib/signal.py
[https://github.com/python/cpython/tree/3.14/Lib/signal.py]

This module provides mechanisms to use signal handlers in Python.

General rules

The **`signal.signal()`** function allows defining custom handlers to be executed when a signal is received. A small number of default handlers are installed: **`SIGPIPE`** is ignored (so write errors on pipes and sockets can be reported as ordinary Python exceptions) and **`SIGINT`** is translated into a **`KeyboardInterrupt`** exception if the parent process has not changed it.

A handler for a particular signal, once set, remains installed until it is explicitly reset (Python emulates the BSD style interface regardless of the underlying implementation), with the exception of the handler for **`SIGCHLD`**, which follows the underlying implementation.

On WebAssembly platforms, signals are emulated and therefore behave differently. Several functions and signals are not available on these platforms.

Execution of Python signal handlers

A Python signal handler does not get executed inside the low-level (C) signal handler. Instead, the low-level signal handler sets a flag which tells the virtual machine to execute the corresponding Python signal handler at a later point(for example at the next bytecode instruction). This has consequences:

- It makes little sense to catch synchronous errors like **`SIGFPE`** or **`SIGSEGV`** that are caused by an invalid operation in C code. Python will return from the signal handler to the C code, which is likely to raise the same signal again, causing Python to apparently hang. From Python 3.3 onwards, you can use the **`faulthandler`** module to report on synchronous errors.
- A long-running calculation implemented purely in C (such as regular expression matching on a large body of text) may run uninterrupted for an arbitrary amount of time, regardless of any signals received. The Python signal handlers will be called when the calculation finishes.
- If the handler raises an exception, it will be raised “out of thin air” in the main thread. See the note below for a discussion.

Signals and threads

Python signal handlers are always executed in the main Python thread of the main interpreter, even if the signal was received in another thread. This means that signals can’t be used as a means of inter-thread communication. You can use the synchronization primitives from the **`threading`** module instead.

Besides, only the main thread of the main interpreter is allowed to set a new signal handler.

Module contents

Changed in version 3.5: signal (SIG*), handler (`SIG_DFL`, `SIG_IGN`) and sigmask (`SIG_BLOCK`, `SIG_UNBLOCK`, `SIG_SETMASK`) related constants listed below were turned into `enums` (`Signals`, `Handlers` and `Sigmasks` respectively). `getsignal()`, `pthread_sigmask()`, `sigpending()` and `sigwait()` functions return human-readable `enums` as `Signals` objects.

The signal module defines three enums:

class signal.Signals

`enum.IntEnum` collection of SIG* constants and the CTRL_* constants.

Added in version 3.5.

class signal.Handlers

`enum.IntEnum` collection the constants `SIG_DFL` and `SIG_IGN`.

Added in version 3.5.

class signal.Sigmasks

`enum.IntEnum` collection the constants `SIG_BLOCK`, `SIG_UNBLOCK` and `SIG_SETMASK`.

Availability: Unix.

See the man page *sigprocmask(2)* *[https://manpages.debian.org/sigprocmask(2)]* and *pthread_sigmask(3)* *[https://manpages.debian.org/pthread_sigmask(3)]* for further information.

Added in version 3.5.

The variables defined in the `signal` module are:

signal.SIG_DFL

This is one of two standard signal handling options; it will simply perform the default function for the signal. For example, on most systems the default action for `SIGQUIT` is to dump core and exit, while the default action for `SIGCHLD` is to simply ignore it.

signal.SIG_IGN

This is another standard signal handler, which will simply ignore the given signal.

signal.SIGABRT

Abort signal from *abort(3)* *[https://manpages.debian.org/abort(3)]*.

signal.SIGALRM

Timer signal from *alarm(2)* *[https://manpages.debian.org/alarm(2)]*.

Availability: Unix.

signal.SIGBREAK

Interrupt from keyboard (CTRL + BREAK).

Availability: Windows.

signal.SIGBUS

Bus error (bad memory access).

Availability: Unix.

signal.SIGCHLD

Child process stopped or terminated.

Availability: Unix.

signal.SIGCLD

Alias to **`SIGCHLD`**.

Availability: not macOS.

signal.SIGCONT

Continue the process if it is currently stopped

Availability: Unix.

signal.SIGFPE

Floating-point exception. For example, division by zero.

See also

`ZeroDivisionError` is raised when the second argument of a division or modulo operation is zero.

signal.SIGHUP

Hangup detected on controlling terminal or death of controlling process.

Availability: Unix.

signal.SIGILL

Illegal instruction.

signal.SIGINT

Interrupt from keyboard (CTRL + C).

Default action is to raise **`KeyboardInterrupt`**.

signal.SIGKILL

Kill signal.

It cannot be caught, blocked, or ignored.

Availability: Unix.

signal.SIGPIPE

Broken pipe: write to pipe with no readers.

Default action is to ignore the signal.

Availability: Unix.

signal.SIGPROF

Profiling timer expired.

Availability: Unix.

signal.SIGQUIT

Terminal quit signal.

Availability: Unix.

signal.SIGSEGV

Segmentation fault: invalid memory reference.

signal.SIGSTOP

Stop executing (cannot be caught or ignored).

signal.SIGSTKFLT

Stack fault on coprocessor. The Linux kernel does not raise this signal: it can only be raised in user space.

Availability: Linux.

On architectures where the signal is available. See the man page *signal(7)* *[https://manpages.debian.org/signal(7)]* for further information.

Added in version 3.11.

signal.SIGTERM

Termination signal.

signal.SIGUSR1

User-defined signal 1.

Availability: Unix.

signal.SIGUSR2

User-defined signal 2.

Availability: Unix.

signal.SIGVTALRM

Virtual timer expired.

Availability: Unix.

signal.SIGWINCH

Window resize signal.

Availability: Unix.

signal.SIGXCPU

CPU time limit exceeded.

Availability: Unix.

SIG*

All the signal numbers are defined symbolically. For example, the hangup signal is defined as **`signal.SIGHUP`**; the variable names are identical to the names used in C programs, as found in `<signal.h>`. The Unix man page for ‘`signal`’ lists the existing signals (on some systems this is *signal(2)* *[https://manpages.debian.org/signal(2)]*, on others the list is in *signal(7)* *[https://manpages.debian.org/signal(7)]*). Note that not all systems define the same set of signal names; only those names defined by the system are defined by this module.

signal.CTRL_C_EVENT

The signal corresponding to the `Ctrl+C` keystroke event. This signal can only be used with **`os.kill()`**.

Availability: Windows.

Added in version 3.2.

signal.CTRL_BREAK_EVENT

The signal corresponding to the `Ctrl+Break` keystroke event. This signal can only be used with **`os.kill()`**.

Availability: Windows.

Added in version 3.2.

signal.NSIG

One more than the number of the highest signal number. Use **`valid_signals()`** to get valid signal numbers.

signal.ITIMER_REAL

Decrements interval timer in real time, and delivers **`SIGALRM`** upon expiration.

signal.ITIMER_VIRTUAL

Decrements interval timer only when the process is executing, and delivers SIGVTALRM upon expiration.

signal.ITIMER_PROF

Decrements interval timer both when the process executes and when the system is executing on behalf of the process. Coupled with ITIMER_VIRTUAL, this timer is usually used to profile the time spent by the application in user and kernel space. SIGPROF is delivered upon expiration.

signal.SIG_BLOCK

A possible value for the *how* parameter to `pthread_sigmask()` indicating that signals are to be blocked.

Added in version 3.3.

signal.SIG_UNBLOCK

A possible value for the *how* parameter to `pthread_sigmask()` indicating that signals are to be unblocked.

Added in version 3.3.

signal.SIG_SETMASK

A possible value for the *how* parameter to `pthread_sigmask()` indicating that the signal mask is to be replaced.

Added in version 3.3.

The `signal` module defines one exception:

exception signal.ItimerError

Raised to signal an error from the underlying `setitimer()` or `getitimer()` implementation. Expect this error if an invalid interval timer or a negative time is passed to `setitimer()`. This error is a subtype of `OSError`.

Added in version 3.3: This error used to be a subtype of `IOError`, which is now an alias of `OSError`.

The `signal` module defines the following functions:

signal.alarm(*time*)

If *time* is non-zero, this function requests that a `SIGALRM` signal be sent to the process in *time* seconds. Any previously scheduled alarm is canceled (only one alarm can be scheduled at any time). The returned value is then the number of seconds before any previously set alarm was to have been delivered. If *time* is zero, no alarm is scheduled, and any scheduled alarm is canceled. If the return value is zero, no alarm is currently scheduled.

Availability: Unix.

See the man page *alarm(2)* *[https://manpages.debian.org/alarm(2)]* for further information.

signal.getsignal(*signalnum*)

Return the current signal handler for the signal *signalnum*. The returned value may be a callable Python object, or one of the special values `signal.SIG_IGN`, `signal.SIG_DFL` or `None`. Here, `signal.SIG_IGN` means that the signal was previously ignored, `signal.SIG_DFL` means that the default way of handling the signal was previously in use, and `None` means that the previous signal handler was not installed from Python.

signal.strsignal(*signalnum*)

Returns the description of signal *signalnum*, such as “Interrupt” for `SIGINT`. Returns `None` if *signalnum* has no description. Raises `ValueError` if *signalnum* is invalid.

Added in version 3.8.

signal.valid_signals()

Return the set of valid signal numbers on this platform. This can be less than `range(1, NSIG)` if some signals are reserved by the system for internal use.

Added in version 3.8.

signal.pause()

Cause the process to sleep until a signal is received; the appropriate handler will then be called. Returns nothing.

Availability: Unix.

See the man page *signal(2)* *[https://manpages.debian.org/signal(2)]* for further information.

See also **`sigwait()`**, **`sigwaitinfo()`**, **`sigtimedwait()`** and **`sigpending()`**.

signal.raise_signal(*signum*)

Sends a signal to the calling process. Returns nothing.

Added in version 3.8.

signal.pidfd_send_signal(*pidfd*, *sig*, *siginfo=None*, *flags=0*)

Send signal *sig* to the process referred to by file descriptor *pidfd*. Python does not currently support the *siginfo* parameter; it must be `None`. The *flags* argument is provided for future extensions; no flag values are currently defined.

See the *pidfd_send_signal(2)* *[https://manpages.debian.org/pidfd_send_signal(2)]* man page for more information.

Availability: Linux >= 5.1, Android >= **`build-time`** API level 31

Added in version 3.9.

signal.pthread_kill(*thread_id*, *signalnum*)

Send the signal *signalnum* to the thread *thread_id*, another thread in the same process as the caller. The target thread can be executing any code (Python or not). However, if the target thread is executing the Python interpreter, the Python signal handlers will be executed by the main thread of the main interpreter. Therefore, the only point of sending a signal to a particular Python thread would be to force a running system call to fail with **`InterruptedError`**.

Use **`threading.get_ident()`** or the **`ident`** attribute of **`threading.Thread`** objects to get a suitable value for *thread_id*.

If *signalnum* is 0, then no signal is sent, but error checking is still performed; this can be used to check if the target thread is still running.

Raises an auditing event `signal.pthread_kill` with arguments `thread_id`, `signalnum`.

Availability: Unix.

See the man page *pthread_kill(3)* *[https://manpages.debian.org/pthread_kill(3)]* for further information.

See also **`os.kill()`**.

Added in version 3.3.

signal.pthread_sigmask(*how*, *mask*)

Fetch and/or change the signal mask of the calling thread. The signal mask is the set of signals whose delivery is currently blocked for the caller. Return the old signal mask as a set of signals.

The behavior of the call is dependent on the value of *how*, as follows.

- **`SIG_BLOCK`**: The set of blocked signals is the union of the current set and the *mask* argument.
- **`SIG_UNBLOCK`**: The signals in *mask* are removed from the current set of blocked signals. It is permissible to attempt to unblock a signal which is not blocked.
- **`SIG_SETMASK`**: The set of blocked signals is set to the *mask* argument.

mask is a set of signal numbers (e.g. {**`signal.SIGINT`**, **`signal.SIGTERM`**}). Use **`valid_signals()`** for a full mask including all signals.

For example, `signal.pthread_sigmask(signal.SIG_BLOCK, [])` reads the signal mask of the calling thread.

`SIGKILL` and **`SIGSTOP`** cannot be blocked.

Availability: Unix.

See the man page *sigprocmask(2)* *[https://manpages.debian.org/sigprocmask(2)]* and *pthread_sigmask(3)* *[https://manpages.debian.org/pthread_sigmask(3)]* for further information.

See also **`pause()`**, **`sigpending()`** and **`sigwait()`**.

Added in version 3.3.

signal.setitimer(*which*, *seconds*, *interval=0.0*)

Sets given interval timer (one of `signal.ITIMER_REAL`, `signal.ITIMER_VIRTUAL` or `signal.ITIMER_PROF`) specified by *which* to fire after *seconds* (float is accepted, different from `alarm()`) and after that every *interval* seconds (if *interval* is non-zero). The interval timer specified by *which* can be cleared by setting *seconds* to zero.

When an interval timer fires, a signal is sent to the process. The signal sent is dependent on the timer being used; `signal.ITIMER_REAL` will deliver `SIGALRM`, `signal.ITIMER_VIRTUAL` sends `SIGVTALRM`, and `signal.ITIMER_PROF` will deliver `SIGPROF`.

The old values are returned as a tuple: (delay, interval).

Attempting to pass an invalid interval timer will cause an `ItimerError`.

Availability: Unix.

signal.getitimer(*which*)

Returns current value of a given interval timer specified by *which*.

Availability: Unix.

signal.set_wakeup_fd(*fd*, *, *warn_on_full_buffer=True*)

Set the wakeup file descriptor to *fd*. When a signal your program has registered a signal handler for is received, the signal number is written as a single byte into the fd. If you haven’t registered a signal handler for the signals you care about, then nothing will be written to the wakeup fd. This can be used by a library to wakeup a poll or select call, allowing the signal to be fully processed.

The old wakeup fd is returned (or -1 if file descriptor wakeup was not enabled). If *fd* is -1, file descriptor wakeup is disabled. If not -1, *fd* must be non-blocking. It is up to the library to remove any bytes from *fd* before calling poll or select again.

When threads are enabled, this function can only be called from the main thread of the main interpreter; attempting to call it from other threads will cause a **`ValueError`** exception to be raised.

There are two common ways to use this function. In both approaches, you use the fd to wake up when a signal arrives, but then they differ in how they determine *which* signal or signals have arrived.

In the first approach, we read the data out of the fd’s buffer, and the byte values give you the signal numbers. This is simple, but in rare cases it can run into a problem: generally the fd will have a limited amount of buffer space, and if too many signals arrive too quickly, then the buffer may become full, and some signals may be lost. If you use this approach, then you should set `warn_on_full_buffer=True`, which will at least cause a warning to be printed to stderr when signals are lost.

In the second approach, we use the wakeup fd *only* for wakeups, and ignore the actual byte values. In this case, all we care about is whether the fd’s buffer is empty or non-empty; a full buffer doesn’t indicate a problem at all. If you use this approach, then you should set `warn_on_full_buffer=False`, so that your users are not confused by spurious warning messages.

Changed in version 3.5: On Windows, the function now also supports socket handles.

Changed in version 3.7: Added `warn_on_full_buffer` parameter.

signal.siginterrupt(*signalnum*, *flag*)

Change system call restart behaviour: if *flag* is **`False`**, system calls will be restarted when interrupted by signal *signalnum*, otherwise system calls will be interrupted. Returns nothing.

Availability: Unix.

See the man page *siginterrupt(3)* *[https://manpages.debian.org/siginterrupt(3)]* for further information.

Note that installing a signal handler with **`signal()`** will reset the restart behaviour to interruptible by implicitly calling **`siginterrupt()`** with a true *flag* value for the given signal.

signal.signal(*signalnum*, *handler*)

Set the handler for signal *signalnum* to the function *handler*. *handler* can be a callable Python object taking two arguments (see below), or one of the special values **`signal.SIG_IGN`** or **`signal.SIG_DFL`**. The previous signal handler will be returned (see the description of **`getsignal()`** above). (See the Unix man page *signal(2)* *[https://manpages.debian.org/signal(2)]* for further information.)

When threads are enabled, this function can only be called from the main thread of the main interpreter; attempting to call it from other threads will cause a **`ValueError`** exception to be raised.

The *handler* is called with two arguments: the signal number and the current stack frame (`None` or a frame object; for a description of frame objects, see the description in the type hierarchy or see the attribute descriptions in the `inspect` module).

On Windows, `signal()` can only be called with `SIGABRT`, `SIGFPE`, `SIGILL`, `SIGINT`, `SIGSEGV`, `SIGTERM`, or `SIGBREAK`. A `ValueError` will be raised in any other case. Note that not all systems define the same set of signal names; an `AttributeError` will be raised if a signal name is not defined as `SIG*` module level constant.

signal.sigpending()

Examine the set of signals that are pending for delivery to the calling thread (i.e., the signals which have been raised while blocked). Return the set of the pending signals.

Availability: Unix.

See the man page *sigpending(2)* *[https://manpages.debian.org/sigpending(2)]* for further information.

See also `pause()`, `pthread_sigmask()` and `sigwait()`.

Added in version 3.3.

signal.sigwait(*sigset*)

Suspend execution of the calling thread until the delivery of one of the signals specified in the signal set *sigset*. The function accepts the signal (removes it from the pending list of signals), and returns the signal number.

Availability: Unix.

See the man page *sigwait(3)* *[https://manpages.debian.org/sigwait(3)]* for further information.

See also **`pause()`**, **`pthread_sigmask()`**, **`sigpending()`**, **`sigwaitinfo()`** and **`sigtimedwait()`**.

Added in version 3.3.

signal.sigwaitinfo(*sigset*)

Suspend execution of the calling thread until the delivery of one of the signals specified in the signal set *sigset*. The function accepts the signal and removes it from the pending list of signals. If one of the signals in *sigset* is already pending for the calling thread, the function will return immediately with information about that signal. The signal handler is not called for the delivered signal. The function raises an **`InterruptedError`** if it is interrupted by a signal that is not in *sigset*.

The return value is an object representing the data contained in the `siginfo_t` structure, namely: `si_signo`, `si_code`, `si_errno`, `si_pid`, `si_uid`, `si_status`, `si_band`.

Availability: Unix.

See the man page *sigwaitinfo(2)* *[https://manpages.debian.org/sigwaitinfo(2)]* for further information.

See also **`pause()`**, **`sigwait()`** and **`sigtimedwait()`**.

Added in version 3.3.

Changed in version 3.5: The function is now retried if interrupted by a signal not in *sigset* and the signal handler does not raise an exception (see **PEP 475** [https://peps.python.org/pep-0475/] for the rationale).

signal.sigtimedwait(*sigset*, *timeout*)

Like **`sigwaitinfo()`**, but takes an additional *timeout* argument specifying a timeout. If *timeout* is specified as

`0`, a poll is performed. Returns **`None`** if a timeout occurs.

Availability: Unix.

See the man page *sigtimedwait(2)* *[https://manpages.debian.org/sigtimedwait(2)]* for further information.

See also **`pause()`**, **`sigwait()`** and **`sigwaitinfo()`**.

Added in version 3.3.

Changed in version 3.5: The function is now retried with the recomputed *timeout* if interrupted by a signal not in *sigset* and the signal handler does not raise an exception (see **PEP 475** [https://peps.python.org/pep-0475/] for the rationale).

Examples

Here is a minimal example program. It uses the **`alarm()`** function to limit the time spent waiting to open a file; this is useful if the file is for a serial device that may not be turned on, which would normally cause the **`os.open()`** to hang indefinitely. The solution is to set a 5-second alarm before opening the file; if the operation takes too long, the alarm signal will be sent, and the handler raises an exception.

```
import signal, os

def handler(signum, frame):
    signame = signal.Signals(signum).name
    print(f'Signal handler called with signal {signame}
({signum})')
    raise OSError("Couldn't open device!")

# Set the signal handler and a 5-second alarm
signal.signal(signal.SIGALRM, handler)
signal.alarm(5)

# This open() may hang indefinitely
fd = os.open('/dev/ttyS0', os.O_RDWR)

signal.alarm(0)          # Disable the alarm
```

Note on SIGPIPE

Piping output of your program to tools like *head(1)* *[https://manpages.debian.org/head(1)]* will cause a **`SIGPIPE`** signal to be sent to your process when the receiver of its standard output closes early. This results in an exception like `BrokenPipeError: [Errno 32] Broken pipe`. To handle this case, wrap your entry point to catch this exception as follows:

```
import os
import sys

def main():
    try:
        # simulate large output (your code replaces this loop)
        for x in range(10000):
            print("y")
        # flush output here to force SIGPIPE to be triggered
        # while inside this try block.
        sys.stdout.flush()
    except BrokenPipeError:
        # Python flushes standard streams on exit; redirect
remaining output
        # to devnull to avoid another BrokenPipeError at
shutdown
        devnull = os.open(os.devnull, os.O_WRONLY)
        os.dup2(devnull, sys.stdout.fileno())
        sys.exit(1)  # Python exits with error code 1 on EPIPE

if __name__ == '__main__':
    main()
```

Do not set `SIGPIPE`’s disposition to `SIG_DFL` in order to avoid `BrokenPipeError`. Doing that would cause your program to exit unexpectedly whenever any socket connection is interrupted while your program is still writing to it.

Note on Signal Handlers and Exceptions

If a signal handler raises an exception, the exception will be propagated to the main thread and may be raised after any bytecode instruction. Most notably, a `KeyboardInterrupt` may appear at any point during execution. Most Python code, including the standard library, cannot be made robust against this, and so a `KeyboardInterrupt` (or any other exception resulting from a signal handler) may on rare occasions put the program in an unexpected state.

To illustrate this issue, consider the following code:

```
class SpamContext:
    def __init__(self):
        self.lock = threading.Lock()

    def __enter__(self):
        # If KeyboardInterrupt occurs here, everything is fine
        self.lock.acquire()
        # If KeyboardInterrupt occurs here, __exit__ will not
be called
        ...
        # KeyboardInterrupt could occur just before the
function returns

    def __exit__(self, exc_type, exc_val, exc_tb):
        ...
        self.lock.release()
```

For many programs, especially those that merely want to exit on `KeyboardInterrupt`, this is not a problem, but applications that are complex or require high reliability should avoid raising exceptions from signal handlers. They should also avoid catching `KeyboardInterrupt` as a means of gracefully shutting down. Instead, they should install their own `SIGINT` handler. Below is an example of an HTTP server that avoids `KeyboardInterrupt`:

```
import signal
import socket
from selectors import DefaultSelector, EVENT_READ
from http.server import HTTPServer, SimpleHTTPRequestHandler

interrupt_read, interrupt_write = socket.socketpair()

def handler(signum, frame):
    print('Signal handler called with signal', signum)
    interrupt_write.send(b'\0')
signal.signal(signal.SIGINT, handler)

def serve_forever(httpd):
    sel = DefaultSelector()
    sel.register(interrupt_read, EVENT_READ)
    sel.register(httpd, EVENT_READ)

    while True:
        for key, _ in sel.select():
            if key.fileobj == interrupt_read:
                interrupt_read.recv(1)
                return
            if key.fileobj == httpd:
                httpd.handle_request()

print("Serving on port 8000")
httpd = HTTPServer(('', 8000), SimpleHTTPRequestHandler)
serve_forever(httpd)
print("Shutdown...")
```

`mmap` — Memory-mapped file support

Availability: not WASI.

This module does not work or is not available on WebAssembly. See WebAssembly platforms for more information.

Memory-mapped file objects behave like both **`bytearray`** and like file objects. You can use mmap objects in most places where **`bytearray`** are expected; for example, you can use the **`re`** module to search through a memory-mapped file. You can also change a single byte by doing `obj[index] = 97`, or change a subsequence by assigning to a slice: `obj[i1:i2] = b'...'`. You can also read and write data starting at the current file position, and **`seek()`** through the file to different positions.

A memory-mapped file is created by the **`mmap`** constructor, which is different on Unix and on Windows. In either case you must provide a file descriptor for a file opened for update. If you wish to map an existing Python file object, use its **`fileno()`** method to obtain the correct value for the *fileno* parameter. Otherwise, you can open the file using the **`os.open()`** function, which returns a file descriptor directly (the file still needs to be closed when done).

Note

If you want to create a memory-mapping for a writable, buffered file, you should **`flush()`** the file first. This is necessary to ensure that local modifications to the buffers are actually available to the mapping.

For both the Unix and Windows versions of the constructor, *access* may be specified as an optional keyword parameter. *access* accepts one of four values: `ACCESS_READ`, `ACCESS_WRITE`, or `ACCESS_COPY` to specify read-only, write-through or copy-on-write memory respectively, or `ACCESS_DEFAULT` to defer to *prot*. *access* can be used on both Unix and Windows. If *access* is not specified, Windows mmap returns a write-through mapping. The initial memory values for all three access types are taken from the specified file. Assignment to an `ACCESS_READ` memory map raises a `TypeError` exception. Assignment to an `ACCESS_WRITE` memory map affects both memory and the underlying file. Assignment to an `ACCESS_COPY` memory map affects memory but does not update the underlying file.

Changed in version 3.7: Added `ACCESS_DEFAULT` constant.

To map anonymous memory, -1 should be passed as the fileno along with the length.

class mmap.mmap(*fileno*, *length*, *tagname=None*, *access=ACCESS_DEFAULT*, *offset=0*)

(Windows version) Maps *length* bytes from the file specified by the file handle *fileno*, and creates a mmap object. If *length* is larger than the current size of the file, the file is extended to contain *length* bytes. If *length* is `0`, the maximum length of the map is the current size of the file, except that if the file is empty Windows raises an exception (you cannot create an empty mapping on Windows).

tagname, if specified and not `None`, is a string giving a tag name for the mapping. Windows allows you to have many different mappings against the same file. If you specify the name of an existing tag, that tag is opened, otherwise a new tag of this name is created. If this parameter is omitted or `None`, the mapping is created without a name. Avoiding the use of the *tagname* parameter will assist in keeping your code portable between Unix and Windows.

offset may be specified as a non-negative integer offset. mmap references will be relative to the offset from the beginning of the file. *offset* defaults to 0. *offset* must be a multiple of the **`ALLOCATIONGRANULARITY`**.

Raises an auditing event `mmap.__new__` with arguments `fileno`, `length`, `access`, `offset`.

class mmap.mmap(*fileno*, *length*, *flags=MAP_SHARED*, *prot=PROT_WRITE | PROT_READ*, *access=ACCESS_DEFAULT*, *offset=0*, *, *trackfd=True*)

(Unix version) Maps *length* bytes from the file specified by the file descriptor *fileno*, and returns a mmap object. If *length* is `0`, the maximum length of the map will be the current size of the file when **`mmap`** is called.

flags specifies the nature of the mapping. **`MAP_PRIVATE`** creates a private copy-on-write mapping, so changes to the contents of the mmap object will be private to this process, and **`MAP_SHARED`** creates a mapping that's shared with all other processes mapping the same areas of the file. The default value is **`MAP_SHARED`**. Some systems have additional possible flags with the full list specified in MAP_* constants.

prot, if specified, gives the desired memory protection; the two most useful values are **`PROT_READ`** and **`PROT_WRITE`**, to specify that the pages may be read or written. *prot* defaults to **`PROT_READ | PROT_WRITE`**.

access may be specified in lieu of *flags* and *prot* as an optional keyword parameter. It is an error to specify both *flags*, *prot* and *access*. See the description of *access* above for information on how to use this parameter.

offset may be specified as a non-negative integer offset. mmap references will be relative to the offset from the beginning of the file. *offset* defaults to 0. *offset* must be a multiple of **`ALLOCATIONGRANULARITY`** which is equal to **`PAGESIZE`** on Unix systems.

If *trackfd* is `False`, the file descriptor specified by *fileno* will not be duplicated, and the resulting **`mmap`** object will not be associated with the map's underlying file. This means that the **`size()`** and **`resize()`** methods will fail. This mode is useful to limit the number of open file descriptors.

To ensure validity of the created memory mapping the file specified by the descriptor *fileno* is internally automatically synchronized with the physical backing store on macOS.

Changed in version 3.13: The *trackfd* parameter was added.

This example shows a simple way of using **`mmap`**:

```
import mmap

# write a simple example file
with open("hello.txt", "wb") as f:
    f.write(b"Hello Python!\n")
```

```
with open("hello.txt", "r+b") as f:
    # memory-map the file, size 0 means whole file
    mm = mmap.mmap(f.fileno(), 0)
    # read content via standard file methods
    print(mm.readline())  # prints b"Hello Python!\n"
    # read content via slice notation
    print(mm[:5])  # prints b"Hello"
    # update content using slice notation;
    # note that new content must have same size
    mm[6:] = b" world!\n"
    # ... and read again using standard file methods
    mm.seek(0)
    print(mm.readline())  # prints b"Hello  world!\n"
    # close the map
    mm.close()
```

`mmap` can also be used as a context manager in a **`with`** statement:

```
import mmap

with mmap.mmap(-1, 13) as mm:
    mm.write(b"Hello world!")
```

Added in version 3.2: Context manager support.

The next example demonstrates how to create an anonymous map and exchange data between the parent and child processes:

```
import mmap
import os

mm = mmap.mmap(-1, 13)
mm.write(b"Hello world!")

pid = os.fork()

if pid == 0:  # In a child process
    mm.seek(0)
    print(mm.readline())
    mm.close()
```

Raises an auditing event `mmap.__new__` with arguments `fileno`, `length`, `access`, `offset`.

Memory-mapped file objects support the following methods:

close()

Closes the mmap. Subsequent calls to other methods of the object will result in a ValueError exception being raised. This will not close the open file.

closed

`True` if the file is closed.

Added in version 3.2.

find(*sub*[, *start*[, *end*]])

Returns the lowest index in the object where the subsequence *sub* is found, such that *sub* is contained in the range [*start*, *end*]. Optional arguments *start* and *end* are interpreted as in slice notation. Returns `-1` on failure.

Changed in version 3.5: Writable bytes-like object is now accepted.

flush()
flush(*offset*, *size*, */*)

Flushes changes made to the in-memory copy of a file back to disk. Without use of this call there is no guarantee that changes are written back before the object is destroyed. If *offset* and *size* are specified, only changes to the given range of bytes will be flushed to disk; otherwise, the whole extent of the mapping is flushed. *offset* must be a multiple of the **`PAGESIZE`** or **`ALLOCATIONGRANULARITY`**.

`None` is returned to indicate success. An exception is raised when the call failed.

Changed in version 3.8: Previously, a nonzero value was returned on success; zero was returned on error under Windows. A zero value was returned on success; an exception was raised on error under Unix.

madvise(*option*[, *start*[, *length*]])

Send advice *option* to the kernel about the memory region beginning at *start* and extending *length* bytes. *option* must be one of the MADV_* constants available on the system. If *start* and *length* are omitted, the entire mapping is spanned. On some systems (including Linux), *start* must be a multiple of the **`PAGESIZE`**.

Availability: Systems with the `madvise()` system call.

Added in version 3.8.

move(*dest*, *src*, *count*)

Copy the *count* bytes starting at offset *src* to the destination index *dest*. If the mmap was created with **`ACCESS_READ`**, then calls to move will raise a **`TypeError`** exception.

read([*n*])

Return a **`bytes`** containing up to *n* bytes starting from the current file position. If the argument is omitted, `None` or negative, return all bytes from the current file position to the end of the mapping. The file position is updated to point after the bytes that were returned.

Changed in version 3.3: Argument can be omitted or `None`.

read_byte()

Returns a byte at the current file position as an integer, and advances the file position by 1.

readline()

Returns a single line, starting at the current file position and up to the next newline. The file position is updated to point after the bytes that were returned.

resize(*newsize*)

Resizes the map and the underlying file, if any.

Resizing a map created with *access* of **`ACCESS_READ`** or **`ACCESS_COPY`**, will raise a **`TypeError`** exception. Resizing a map created with *trackfd* set to `False`, will raise a **`ValueError`** exception.

On Windows: Resizing the map will raise an **`OSError`** if there are other maps against the same named file. Resizing an anonymous map (ie against the pagefile) will silently create a new map with the original data copied over up to the length of the new size.

Changed in version 3.11: Correctly fails if attempting to resize when another map is held Allows resize against an anonymous map on Windows

rfind(*sub*[, *start*[, *end*]])

Returns the highest index in the object where the subsequence *sub* is found, such that *sub* is contained in the range [*start*, *end*]. Optional arguments *start* and *end* are interpreted as in slice notation. Returns `-1` on failure.

Changed in version 3.5: Writable bytes-like object is now accepted.

seek(*pos*[, *whence*])

Set the file’s current position. *whence* argument is optional and defaults to `os.SEEK_SET` or `0` (absolute file positioning); other values are `os.SEEK_CUR` or `1` (seek relative to the current position) and `os.SEEK_END` or `2` (seek relative to the file’s end).

Changed in version 3.13: Return the new absolute position instead of `None`.

seekable()

Return whether the file supports seeking, and the return value is always `True`.

Added in version 3.13.

size()

Return the length of the file, which can be larger than the size of the memory-mapped area.

tell()

Returns the current position of the file pointer.

write(*bytes*)

Write the bytes in *bytes* into memory at the current position of the file pointer and return the number of bytes written (never less than `len(bytes)`, since if the write fails, a **`ValueError`** will be raised). The file position is updated to point after the bytes that were written. If the mmap was created with **`ACCESS_READ`**, then writing to it will raise a **`TypeError`** exception.

Changed in version 3.5: Writable bytes-like object is now accepted.

Changed in version 3.6: The number of bytes written is now returned.

write_byte(*byte*)

Write the integer *byte* into memory at the current position of the file pointer; the file position is advanced by `1`. If the mmap was created with **`ACCESS_READ`**, then writing to it will raise a **`TypeError`** exception.

MADV_* Constants

mmap.MADV_NORMAL
mmap.MADV_RANDOM
mmap.MADV_SEQUENTIAL
mmap.MADV_WILLNEED
mmap.MADV_DONTNEED
mmap.MADV_REMOVE
mmap.MADV_DONTFORK
mmap.MADV_DOFORK
mmap.MADV_HWPOISON
mmap.MADV_MERGEABLE
mmap.MADV_UNMERGEABLE
mmap.MADV_SOFT_OFFLINE
mmap.MADV_HUGEPAGE
mmap.MADV_NOHUGEPAGE
mmap.MADV_DONTDUMP
mmap.MADV_DODUMP
mmap.MADV_FREE
mmap.MADV_NOSYNC
mmap.MADV_AUTOSYNC
mmap.MADV_NOCORE

mmap.MADV_CORE
mmap.MADV_PROTECT
mmap.MADV_FREE_REUSABLE
mmap.MADV_FREE_REUSE

These options can be passed to `mmap.madvise()`. Not every option will be present on every system.

Availability: Systems with the madvise() system call.

Added in version 3.8.

MAP_* Constants

mmap.MAP_SHARED
mmap.MAP_PRIVATE
mmap.MAP_32BIT
mmap.MAP_ALIGNED_SUPER
mmap.MAP_ANON
mmap.MAP_ANONYMOUS
mmap.MAP_CONCEAL
mmap.MAP_DENYWRITE
mmap.MAP_EXECUTABLE
mmap.MAP_HASSEMAPHORE
mmap.MAP_JIT
mmap.MAP_NOCACHE
mmap.MAP_NOEXTEND
mmap.MAP_NORESERVE
mmap.MAP_POPULATE
mmap.MAP_RESILIENT_CODESIGN
mmap.MAP_RESILIENT_MEDIA
mmap.MAP_STACK
mmap.MAP_TPRO
mmap.MAP_TRANSLATED_ALLOW_EXECUTE
mmap.MAP_UNIX03

These are the various flags that can be passed to `mmap.mmap()`. `MAP_ALIGNED_SUPER` is only available at FreeBSD and `MAP_CONCEAL` is only available at OpenBSD. Note that some options might not be present on some systems.

Changed in version 3.10: Added `MAP_POPULATE` constant.

Added in version 3.11: Added `MAP_STACK` constant.

Added in version 3.12: Added `MAP_ALIGNED_SUPER` and `MAP_CONCEAL` constants.

Added in version 3.13: Added `MAP_32BIT`, `MAP_HASSEMAPHORE`, `MAP_JIT`, `MAP_NOCACHE`, `MAP_NOEXTEND`, `MAP_NORESERVE`, `MAP_RESILIENT_CODESIGN`, `MAP_RESILIENT_MEDIA`, `MAP_TPRO`, `MAP_TRANSLATED_ALLOW_EXECUTE`, and `MAP_UNIX03` constants.

Internet Data Handling

This chapter describes modules which support handling data formats commonly used on the internet.

- `email` — An email and MIME handling package
    - `email.message`: Representing an email message
    - `email.parser`: Parsing email messages
        - FeedParser API
        - Parser API
        - Additional notes
    - `email.generator`: Generating MIME documents
    - `email.policy`: Policy Objects
    - `email.errors`: Exception and Defect classes
    - `email.headerregistry`: Custom Header Objects
    - `email.contentmanager`: Managing MIME Content
        - Content Manager Instances
    - `email`: Examples
    - `email.message.Message`: Representing an email message using the `compat32` API
    - `email.mime`: Creating email and MIME objects from scratch
    - `email.header`: Internationalized headers
    - `email.charset`: Representing character sets
    - `email.encoders`: Encoders
    - `email.utils`: Miscellaneous utilities
    - `email.iterators`: Iterators
- `json` — JSON encoder and decoder
    - Basic Usage
    - Encoders and Decoders
    - Exceptions
    - Standard Compliance and Interoperability
        - Character Encodings

        - Infinite and NaN Number Values
        - Repeated Names Within an Object
        - Top-level Non-Object, Non-Array Values
        - Implementation Limitations
    - Command-line interface
        - Command-line options
- **`mailbox`** — Manipulate mailboxes in various formats
    - **`Mailbox`** objects
        - **`Maildir`** objects
        - **`mbox`** objects
        - **`MH`** objects
        - **`Babyl`** objects
        - **`MMDF`** objects
    - **`Message`** objects
        - **`MaildirMessage`** objects
        - **`mboxMessage`** objects
        - **`MHMessage`** objects
        - **`BabylMessage`** objects
        - **`MMDFMessage`** objects
    - Exceptions
    - Examples
- **`mimetypes`** — Map filenames to MIME types
    - MimeTypes objects
    - Command-line usage
    - Command-line example
- **`base64`** — Base16, Base32, Base64, Base85 Data Encodings
    - RFC 4648 Encodings
    - Base85 Encodings
    - Legacy Interface
    - Security Considerations
- **`binascii`** — Convert between binary and ASCII
- **`quopri`** — Encode and decode MIME quoted-printable data

`email` — An email and MIME handling package

Source code: Lib/email/__init__.py [https://github.com/python/cpython/tree/3.14/Lib/email/__init__.py]

The **`email`** package is a library for managing email messages. It is specifically *not* designed to do any sending of email messages to SMTP (**RFC 2821** [https://datatracker.ietf.org/doc/html/rfc2821.html]), NNTP, or other servers; those are functions of modules such as **`smtplib`**. The **`email`** package attempts to be as RFC-compliant as possible, supporting **RFC 5322** [https://datatracker.ietf.org/doc/html/rfc5322.html] and **RFC 6532** [https://datatracker.ietf.org/doc/html/rfc6532.html], as well as such MIME-related RFCs as **RFC 2045** [https://datatracker.ietf.org/doc/html/rfc2045.html], **RFC 2046** [https://datatracker.ietf.org/doc/html/rfc2046.html], **RFC 2047** [https://datatracker.ietf.org/doc/html/rfc2047.html], **RFC 2183** [https://datatracker.ietf.org/doc/html/rfc2183.html], and **RFC 2231** [https://datatracker.ietf.org/doc/html/rfc2231.html].

The overall structure of the email package can be divided into three major components, plus a fourth component that controls the behavior of the other components.

The central component of the package is an “object model” that represents email messages. An application interacts with the package primarily through the object model interface defined in the **`message`** sub-module. The application can use this API to ask questions about an existing email, to construct a new email, or to add or remove email subcomponents that themselves use the same object model interface. That is, following the nature of email messages and their MIME subcomponents, the email object model is a tree structure of objects that all provide the `EmailMessage` API.

The other two major components of the package are the `parser` and the `generator`. The parser takes the serialized version of an email message (a stream of bytes) and converts it into a tree of `EmailMessage` objects. The generator takes an `EmailMessage` and turns it back into a serialized byte stream. (The parser and generator also handle streams of text characters, but this usage is discouraged as it is too easy to end up with messages that are not valid in one way or another.)

The control component is the `policy` module. Every `EmailMessage`, every `generator`, and every `parser` has an associated `policy` object that controls its behavior. Usually an application only needs to specify the policy when an `EmailMessage` is created, either by directly instantiating an `EmailMessage` to create a new email, or by parsing an input stream using a `parser`. But the policy can be changed when the message is serialized using a `generator`. This allows, for example, a generic email message to be parsed from disk, but to serialize it using standard SMTP settings when sending it to an email server.

The email package does its best to hide the details of the various governing RFCs from the application. Conceptually the application should be able to treat the email message as a structured tree of unicode text and binary attachments, without having to worry about how these are represented when serialized. In practice, however, it is often necessary to be aware of at least some of the rules governing MIME messages and their structure, specifically the names and nature of the MIME “content types” and how they identify multipart documents. For the most part this knowledge should only be required for more complex applications, and even then it should only be the high level structure in question, and not the details of how those structures are represented. Since MIME content types are used widely in modern internet software (not just email), this will be a familiar concept to many programmers.

The following sections describe the functionality of the `email` package. We start with the `message` object model, which is the primary interface an application will use, and follow that with the `parser` and `generator` components. Then we cover the `policy` controls, which completes the treatment of the main components of the library.

The next three sections cover the exceptions the package may raise and the defects (non-compliance with the RFCs) that the `parser` may detect. Then we cover the `headerregistry` and the `contentmanager` sub-components, which provide tools for doing more detailed manipulation of headers and payloads, respectively. Both of these components contain features relevant to consuming and producing non-trivial messages, but also document their extensibility APIs, which will be of interest to advanced applications.

Following those is a set of examples of using the fundamental parts of the APIs covered in the preceding sections.

The foregoing represent the modern (unicode friendly) API of the email package. The remaining sections, starting with the `Message` class, cover the legacy `compat32` API that deals much more directly with the details of how email messages are represented. The `compat32` API does *not* hide the details of the RFCs from the application, but for applications that need to operate at that level, they can be useful tools. This documentation is also relevant for applications that are still using the `compat32` API for backward compatibility reasons.

Changed in version 3.6: Docs reorganized and rewritten to promote the new `EmailMessage`/`EmailPolicy` API.

Contents of the `email` package documentation:

- `email.message`: Representing an email message
- `email.parser`: Parsing email messages
  - FeedParser API
  - Parser API
  - Additional notes
- `email.generator`: Generating MIME documents
- `email.policy`: Policy Objects
- `email.errors`: Exception and Defect classes
- `email.headerregistry`: Custom Header Objects
- `email.contentmanager`: Managing MIME Content
  - Content Manager Instances
- `email`: Examples

Legacy API:

- `email.message.Message`: Representing an email message using the `compat32` API
- `email.mime`: Creating email and MIME objects from scratch
- `email.header`: Internationalized headers
- `email.charset`: Representing character sets
- `email.encoders`: Encoders
- `email.utils`: Miscellaneous utilities
- `email.iterators`: Iterators

See also

Module `smtplib`
SMTP (Simple Mail Transport Protocol) client

Module [`poplib`]
POP (Post Office Protocol) client

Module `imaplib`
IMAP (Internet Message Access Protocol) client

Module `mailbox`
Tools for creating, reading, and managing collections of messages on disk using a variety standard formats.

`email.message`: Representing an email message

Source code: Lib/email/message.py [https://github.com/python/cpython/tree/3.14/Lib/email/message.py]

Added in version 3.6: [[1]]

The central class in the `email` package is the `EmailMessage` class, imported from the `email.message` module. It is the base class for the `email` object model. `EmailMessage` provides the core functionality for setting and querying header fields, for accessing message bodies, and for creating or modifying structured messages.

An email message consists of *headers* and a *payload* (which is also referred to as the *content*). Headers are **RFC 5322** [https://datatracker.ietf.org/doc/html/rfc5322.html] or **RFC 6532** [https://datatracker.ietf.org/doc/html/rfc6532.html] style field names and values, where the field name and value are separated by a colon. The colon is not part of either the field name or the field value. The payload may be a simple text message, or a binary object, or a structured sequence of sub-messages each with their own set of headers and their own payload. The latter type of payload is indicated by the message having a MIME type such as *multipart/** or *message/rfc822*.

The conceptual model provided by an `EmailMessage` object is that of an ordered dictionary of headers coupled with a *payload* that represents the **RFC 5322** [https://datatracker.ietf.org/doc/html/rfc5322.html] body of the message, which might be a list of sub-`EmailMessage` objects. In addition to the normal dictionary methods for accessing the header names and values, there are methods for accessing specialized information from the headers (for example the MIME content type), for operating on the payload, for generating a serialized version of the message, and for recursively walking over the object tree.

The `EmailMessage` dictionary-like interface is indexed by the header names, which must be ASCII values. The values of the dictionary are strings with some extra methods. Headers are stored and returned in case-preserving form, but field names are matched case-insensitively. The keys are ordered, but unlike a real dict, there can be duplicates. Additional methods are provided for working with headers that have duplicate keys.

The *payload* is either a string or bytes object, in the case of simple message objects, or a list of `EmailMessage` objects, for MIME container documents such as *multipart/** and *message/rfc822* message objects.

class email.message.EmailMessage(*policy=default*)

If *policy* is specified use the rules it specifies to update and serialize the representation of the message. If *policy* is not set, use the `default` policy, which follows the rules of the email RFCs except for line endings (instead of the RFC mandated `\r\n`, it uses the Python standard `\n` line endings). For more information see the `policy` documentation. [[2]]

as_string(*unixfrom=False*, *maxheaderlen=None*, *policy=None*)

Return the entire message flattened as a string. When optional *unixfrom* is true, the envelope header is included in the returned string. *unixfrom* defaults to `False`. For backward compatibility with the base `Message` class *maxheaderlen* is accepted, but defaults to `None`, which means that by default the line length is controlled by the **`max_line_length`** of the policy. The *policy* argument may be used to override the default policy obtained from the message instance. This can be used to control some of the formatting produced by the method, since the specified *policy* will be passed to the **`Generator`**.

Flattening the message may trigger changes to the **`EmailMessage`** if defaults need to be filled in to complete the transformation to a string (for example, MIME boundaries may be generated or modified).

Note that this method is provided as a convenience and may not be the most useful way to serialize messages in your application, especially if you are dealing with multiple messages. See **`email.generator.Generator`** for a more flexible API for serializing messages. Note also that this method is restricted to producing messages serialized as “7 bit clean” when **`utf8`** is `False`, which is the default.

Changed in version 3.6: the default behavior when *maxheaderlen* is not specified was changed from defaulting to 0 to defaulting to the value of *max_line_length* from the policy.

__str__()

Equivalent to `as_string(policy=self.policy.clone(utf8=True))`. Allows `str(msg)` to produce a string containing the serialized message in a readable format.

Changed in version 3.4: the method was changed to use `utf8=True`, thus producing an **RFC 6531** [https://datatracker.ietf.org/doc/html/rfc6531.html]-like message representation, instead of being a direct alias for `as_string()`.

as_bytes(*unixfrom=False*, *policy=None*)

Return the entire message flattened as a bytes object. When optional *unixfrom* is true, the envelope header is included in the returned string. *unixfrom* defaults to `False`. The *policy* argument may be used to override the default policy obtained from the message instance. This can be used to control some of the formatting produced by the method, since the specified *policy* will be passed to the `BytesGenerator`.

Flattening the message may trigger changes to the `EmailMessage` if defaults need to be filled in to complete the transformation to a string (for example, MIME boundaries may be generated or modified).

Note that this method is provided as a convenience and may not be the most useful way to serialize messages in your application, especially if you are dealing with multiple messages. See `email.generator.BytesGenerator` for a more flexible API for serializing messages.

__bytes__()

Equivalent to `as_bytes()`. Allows `bytes(msg)` to produce a bytes object containing the serialized message.

is_multipart()

Return `True` if the message's payload is a list of sub-`EmailMessage` objects, otherwise return `False`. When `is_multipart()` returns `False`, the payload should be a string object (which might be a CTE encoded binary payload). Note that `is_multipart()` returning `True` does not necessarily mean that “msg.get_content_maintype() == ‘multipart’” will return the `True`. For example, `is_multipart` will return `True` when the **`EmailMessage`** is of type `message/rfc822`.

set_unixfrom(*unixfrom*)

Set the message’s envelope header to *unixfrom*, which should be a string. (See **`mboxMessage`** for a brief description of this header.)

get_unixfrom()

Return the message’s envelope header. Defaults to `None` if the envelope header was never set.

The following methods implement the mapping-like interface for accessing the message’s headers. Note that there are some semantic differences between these methods and a normal mapping (i.e. dictionary) interface. For example, in a dictionary there are no duplicate keys, but here there may be duplicate message headers. Also, in dictionaries there is no guaranteed order to the keys returned by **`keys()`**, but in an **`EmailMessage`** object, headers are always returned in the order they appeared in the original message, or in which they were added to the message later. Any header deleted and then re-added is always appended to the end of the header list.

These semantic differences are intentional and are biased toward convenience in the most common use cases.

Note that in all cases, any envelope header present in the message is not included in the mapping interface.

__len__()

Return the total number of headers, including duplicates.

__contains__(*name*)

Return `True` if the message object has a field named *name*. Matching is done without regard to case and *name* does not include the trailing colon. Used for the `in` operator. For example:

```
if 'message-id' in myMessage:
   print('Message-ID:', myMessage['message-id'])
```

__getitem__(*name*)

Return the value of the named header field. *name* does not include the colon field separator. If the header is missing, `None` is returned; a **`KeyError`** is never raised.

Note that if the named field appears more than once in the message's headers, exactly which of those field values will be returned is undefined. Use the **`get_all()`** method to get the values of all the extant headers named *name*.

Using the standard (non-`compat32`) policies, the returned value is an instance of a subclass of **`email.headerregistry.BaseHeader`**.

__setitem__(*name*, *val*)

Add a header to the message with field name *name* and value *val*. The field is appended to the end of the message's existing headers.

Note that this does *not* overwrite or delete any existing header with the same name. If you want to ensure that the new header is the only one present in the message with field name *name*, delete the field first, e.g.:

```
del msg['subject']
msg['subject'] = 'Python roolz!'
```

If the `policy` defines certain headers to be unique (as the standard policies do), this method may raise a `ValueError` when an attempt is made to assign a value to such a header when one already exists. This behavior is intentional for consistency's sake, but do not depend on it as we may choose to make such assignments do an automatic deletion of the existing header in the future.

__delitem__(*name*)

Delete all occurrences of the field with name *name* from the message's headers. No exception is raised if the named field isn't present in the headers.

keys()

Return a list of all the message's header field names.

values()

Return a list of all the message's field values.

items()

Return a list of 2-tuples containing all the message's field headers and values.

get(*name*, *failobj=None*)

Return the value of the named header field. This is identical to `__getitem__()` except that optional *failobj* is returned if the named header is missing (*failobj* defaults to `None`).

Here are some additional useful header related methods:

get_all(*name*, *failobj=None*)

Return a list of all the values for the field named *name*. If there are no such named headers in the message, *failobj* is returned (defaults to `None`).

add_header(*_name*, *_value*, ***_params*)

Extended header setting. This method is similar to `__setitem__()` except that additional header parameters can be provided as keyword arguments. *_name* is the header field to add and *_value* is the *primary* value for the header.

For each item in the keyword argument dictionary *_params*, the key is taken as the parameter name, with underscores converted to dashes (since dashes are illegal in Python identifiers). Normally, the parameter will be added as `key="value"` unless the value is `None`, in which case only the key will be added.

If the value contains non-ASCII characters, the charset and language may be explicitly controlled by specifying the value as a three tuple in the format `(CHARSET, LANGUAGE, VALUE)`, where `CHARSET` is a string naming the charset to be used to encode the value, `LANGUAGE` can usually be set to `None` or the empty string (see **RFC 2231** [https://datatracker.ietf.org/doc/html/rfc2231.html] for other possibilities), and `VALUE` is the string value containing non-ASCII code points. If a three tuple is not passed and the value contains non-ASCII characters, it is automatically encoded in **RFC 2231** [https://datatracker.ietf.org/doc/html/rfc2231.html] format using a `CHARSET` of `utf-8` and a `LANGUAGE` of `None`.

Here is an example:

```
msg.add_header('Content-Disposition', 'attachment',
filename='bud.gif')
```

This will add a header that looks like

```
Content-Disposition: attachment; filename="bud.gif"
```

An example of the extended interface with non-ASCII characters:

```
msg.add_header('Content-Disposition', 'attachment',
               filename=('iso-8859-1', '',
'Fußballer.ppt'))
```

replace_header(*_name*, *_value*)

Replace a header. Replace the first header found in the message that matches *_name*, retaining header order and field name case of the original header. If no matching header is found, raise a **`KeyError`**.

get_content_type()

Return the message's content type, coerced to lower case of the form *maintype/subtype*. If there is no *Content-Type* header in the message return the value returned by **`get_default_type()`**. If the *Content-Type* header is invalid, return `text/plain`.

(According to **RFC 2045** [https://datatracker.ietf.org/doc/html/rfc2045.html], messages always have a default type, **`get_content_type()`** will always return a value. **RFC 2045** [https://datatracker.ietf.org/doc/html/rfc2045.html] defines a message's default type to be *text/plain* unless it appears inside a *multipart/digest* container, in which case it would be *message/rfc822*. If the *Content-Type* header has an invalid type specification, **RFC 2045** [https://datatracker.ietf.org/doc/html/rfc2045.html] mandates that the default type be *text/plain*.)

get_content_maintype()

Return the message’s main content type. This is the *maintype* part of the string returned by [`get_content_type()`](#).

get_content_subtype()

Return the message’s sub-content type. This is the *subtype* part of the string returned by [`get_content_type()`](#).

get_default_type()

Return the default content type. Most messages have a default content type of *text/plain*, except for messages that are subparts of *multipart/digest* containers. Such subparts have a default content type of *message/rfc822*.

set_default_type(*ctype*)

Set the default content type. *ctype* should either be *text/plain* or *message/rfc822*, although this is not enforced. The default content type is not stored in the *Content-Type* header, so it only affects the return value of the `get_content_type` methods when no *Content-Type* header is present in the message.

set_param(*param*, *value*, *header='Content-Type'*, *requote=True*, *charset=None*, *language=''*, *replace=False*)

Set a parameter in the *Content-Type* header. If the parameter already exists in the header, replace its value with *value*. When *header* is `Content-Type` (the default) and the header does not yet exist in the message, add it, set its value to *text/plain*, and append the new parameter value. Optional *header* specifies an alternative header to *Content-Type*.

If the value contains non-ASCII characters, the charset and language may be explicitly specified using the optional *charset* and *language* parameters. Optional *language* specifies the **RFC 2231** [https://datatracker.ietf.org/doc/html/rfc2231.html] language, defaulting to the empty string. Both *charset* and *language* should be strings. The default is to use the `utf8` *charset* and `None` for the *language*.

If *replace* is `False` (the default) the header is moved to the end of the list of headers. If *replace* is `True`, the header will be updated in place.

Use of the *requote* parameter with `EmailMessage` objects is deprecated.

Note that existing parameter values of headers may be accessed through the `params` attribute of the header value (for example, `msg['Content-Type'].params['charset']`).

Changed in version 3.4: `replace` keyword was added.

del_param(*param*, *header='content-type'*, *requote=True*)

Remove the given parameter completely from the *Content-Type* header. The header will be re-written in place without the parameter or its value. Optional *header* specifies an alternative to *Content-Type*.

Use of the *requote* parameter with `EmailMessage` objects is deprecated.

get_filename(*failobj=None*)

Return the value of the `filename` parameter of the *Content-Disposition* header of the message. If the header does not have a `filename` parameter, this method falls back to looking for the `name` parameter on the *Content-Type* header. If neither is found, or the header is missing, then *failobj* is returned. The returned string will always be unquoted as per **`email.utils.unquote()`**.

get_boundary(*failobj=None*)

Return the value of the `boundary` parameter of the *Content-Type* header of the message, or *failobj* if either the header is missing, or has no `boundary` parameter. The returned string will always be unquoted as per **`email.utils.unquote()`**.

set_boundary(*boundary*)

Set the `boundary` parameter of the *Content-Type* header to *boundary*. **`set_boundary()`** will always quote *boundary* if necessary. A **`HeaderParseError`** is raised if the message object has no *Content-Type* header.

Note that using this method is subtly different from deleting the old *Content-Type* header and adding a new one with the new boundary via **`add_header()`**, because **`set_boundary()`** preserves the order of the *Content-Type* header in the list of headers.

get_content_charset(*failobj=None*)

Return the `charset` parameter of the *Content-Type* header, coerced to lower case. If there is no *Content-Type* header, or if that header has no `charset` parameter, *failobj* is returned.

get_charsets(*failobj=None*)

Return a list containing the character set names in the message. If the message is a *multipart*, then the list will contain one element for each subpart in the payload, otherwise, it will be a list of length 1.

Each item in the list will be a string which is the value of the `charset` parameter in the *Content-Type* header for the represented subpart. If the subpart has no *Content-Type* header, no `charset` parameter, or is not of the *text* main MIME type, then that item in the returned list will be *failobj*.

is_attachment()

Return `True` if there is a *Content-Disposition* header and its (case insensitive) value is `attachment`, `False` otherwise.

Changed in version 3.4.2: is_attachment is now a method instead of a property, for consistency with **`is_multipart()`**.

get_content_disposition()

Return the lowercased value (without parameters) of the message's *Content-Disposition* header if it has one, or `None`. The possible values for this method are *inline*, *attachment* or `None` if the message follows **RFC 2183** [https://datatracker.ietf.org/doc/html/rfc2183.html].

Added in version 3.5.

The following methods relate to interrogating and manipulating the content (payload) of the message.

walk()

The **`walk()`** method is an all-purpose generator which can be used to iterate over all the parts and subparts of a message object tree, in depth-first traversal order. You will typically use **`walk()`** as the iterator in a `for` loop; each iteration returns the next subpart.

Here’s an example that prints the MIME type of every part of a multipart message structure:

```
>>> for part in msg.walk():
...     print(part.get_content_type())
multipart/report
text/plain
message/delivery-status
text/plain
text/plain
message/rfc822
text/plain
```

`walk` iterates over the subparts of any part where `is_multipart()` returns `True`, even though `msg.get_content_maintype() == 'multipart'` may return `False`. We can see this in our example by making use of the `_structure` debug helper function:

```
>>> from email.iterators import _structure
>>> for part in msg.walk():
...     print(part.get_content_maintype() == 'multipart',
...           part.is_multipart())
True True
False False
False True
False False
False False
False True
False False
>>> _structure(msg)
multipart/report
    text/plain
    message/delivery-status
        text/plain
        text/plain
    message/rfc822
        text/plain
```

Here the `message` parts are not `multiparts`, but they do contain subparts. `is_multipart()` returns `True` and `walk` descends into the subparts.

get_body(*preferencelist=('related', 'html', 'plain')*)

Return the MIME part that is the best candidate to be the “body” of the message.

preferencelist must be a sequence of strings from the set `related`, `html`, and `plain`, and indicates the order of preference for the content type of the part returned.

Start looking for candidate matches with the object on which the `get_body` method is called.

If `related` is not included in *preferencelist*, consider the root part (or subpart of the root part) of any related encountered as a candidate if the (sub-)part matches a preference.

When encountering a `multipart/related`, check the `start` parameter and if a part with a matching *Content-ID* is found, consider only it when looking for candidate matches. Otherwise consider only the first (default root) part of the `multipart/related`.

If a part has a *Content-Disposition* header, only consider the part a candidate match if the value of the header is `inline`.

If none of the candidates matches any of the preferences in *preferencelist*, return `None`.

Notes: (1) For most applications the only *preferencelist* combinations that really make sense are `('plain',)`, `('html', 'plain')`, and the default `('related', 'html', 'plain')`. (2) Because matching starts with the object on which `get_body` is called, calling `get_body` on a `multipart/related` will return the object itself unless *preferencelist* has a non-default value. (3) Messages (or message parts) that do not specify a *Content-Type* or whose *Content-Type* header is invalid will be treated as if they are of type `text/plain`, which may occasionally cause `get_body` to return unexpected results.

iter_attachments()

Return an iterator over all of the immediate sub-parts of the message that are not candidate “body” parts. That is, skip the first occurrence of each of `text/plain`, `text/html`, `multipart/related`, or `multipart/alternative` (unless they are explicitly marked as attachments via *Content-Disposition: attachment*), and return all remaining parts. When applied directly to a `multipart/related`, return an iterator over the all the related parts except the root part (ie: the part pointed to by the `start` parameter, or the first part if there is no `start` parameter or the `start` parameter doesn’t match the *Content-ID* of any of the parts). When applied directly to a `multipart/alternative` or a non-`multipart`, return an empty iterator.

iter_parts()

Return an iterator over all of the immediate sub-parts of the message, which will be empty for a non-`multipart`. (See also `walk()`.)

get_content(**args*, *content_manager=None*, ***kw*)

Call the `get_content()` method of the *content_manager*, passing self as the message object, and passing along any other arguments or keywords as additional arguments. If *content_manager* is not specified, use the `content_manager` specified by the current `policy`.

set_content(**args*, *content_manager=None*, ***kw*)

Call the `set_content()` method of the *content_manager*, passing self as the message object, and passing along any other arguments or keywords as additional arguments. If *content_manager* is not specified, use the `content_manager` specified by the current `policy`.

make_related(*boundary=None*)

Convert a non-`multipart` message into a `multipart/related` message, moving any existing *Content-* headers and payload into a (new) first part of the `multipart`. If *boundary* is specified, use it as the boundary string in the multipart, otherwise leave the boundary to be automatically created when it is needed (for example, when the message is serialized).

make_alternative(*boundary=None*)

Convert a non-`multipart` or a `multipart/related` into a `multipart/alternative`, moving any existing *Content-* headers and payload into a (new) first part of the `multipart`. If *boundary* is specified, use it as the boundary string in the multipart, otherwise leave the boundary to be automatically created when it is needed (for example, when the message is serialized).

make_mixed(*boundary=None*)

Convert a non-`multipart`, a `multipart/related`, or a `multipart-alternative` into a `multipart/mixed`, moving any existing *Content-* headers and payload into a (new) first part of the `multipart`. If *boundary* is specified, use it as the boundary string in the multipart, otherwise leave the boundary to be automatically created when it is needed (for example, when the message is serialized).

add_related(**args*, *content_manager=None*, ***kw*)

If the message is a `multipart/related`, create a new message object, pass all of the arguments to its [`set_content()`] method, and [`attach()`] it to the `multipart`. If the message is a non-`multipart`, call [`make_related()`] and then proceed as above. If the message is any other type of `multipart`, raise a [`TypeError`]. If *content_manager* is not specified, use the `content_manager` specified by the current [`policy`]. If the added part has no *Content-Disposition* header, add one with the value `inline`.

add_alternative(**args*, *content_manager=None*, ***kw*)

If the message is a `multipart/alternative`, create a new message object, pass all of the arguments to its [`set_content()`] method, and [`attach()`] it to the `multipart`. If the message is a non-`multipart` or `multipart/related`, call [`make_alternative()`] and then proceed as above. If the message is any other type of `multipart`, raise a [`TypeError`]. If *content_manager* is not specified, use the `content_manager` specified by the current [`policy`].

add_attachment(**args*, *content_manager=None*, ***kw*)

If the message is a `multipart/mixed`, create a new message object, pass all of the arguments to its [`set_content()`] method, and [`attach()`] it to the `multipart`. If the message is a non-`multipart`, `multipart/related`, or `multipart/alternative`, call [`make_mixed()`] and then proceed as above. If *content_manager* is not specified, use the `content_manager` specified by the current [`policy`]. If the added part has no *Content-Disposition* header, add one with the value `attachment`. This method can be used both for explicit attachments (*Content-Disposition: attachment*) and `inline` attachments (*Content-Disposition: inline*), by passing appropriate options to the `content_manager`.

clear()

Remove the payload and all of the headers.

clear_content()

Remove the payload and all of the *!Content-* headers, leaving all other headers intact and in their original order.

`EmailMessage` objects have the following instance attributes:

preamble

The format of a MIME document allows for some text between the blank line following the headers, and the first multipart boundary string. Normally, this text is never visible in a MIME-aware mail reader because it falls outside the standard MIME armor. However, when viewing the raw text of the message, or when viewing the message in a non-MIME aware reader, this text can become visible.

The *preamble* attribute contains this leading extra-armor text for MIME documents. When the `Parser` discovers some text after the headers but before the first boundary string, it assigns this text to the message's *preamble* attribute. When the `Generator` is writing out the plain text representation of a MIME message, and it finds the message has a *preamble* attribute, it will write this text in the area between the headers and the first boundary. See `email.parser` and `email.generator` for details.

Note that if the message object has no preamble, the *preamble* attribute will be `None`.

epilogue

The *epilogue* attribute acts the same way as the *preamble* attribute, except that it contains text that appears between the last boundary and the end of the message. As with the **`preamble`**, if there is no epilog text this attribute will be `None`.

defects

The *defects* attribute contains a list of all the problems found when parsing this message. See **`email.errors`** for a detailed description of the possible parsing defects.

class email.message.MIMEPart(*policy=default*)

This class represents a subpart of a MIME message. It is identical to **`EmailMessage`**, except that no *MIME-Version* headers are added when **`set_content()`** is called, since sub-parts do not need their own *MIME-Version* headers.

Footnotes

[1]

Originally added in 3.4 as a provisional module. Docs for legacy message class moved to email.message.Message: Representing an email message using the compat32 API.

[2]

The **`EmailMessage`** class requires a policy that provides a `content_manager` attribute for content management methods like `set_content()` and `get_content()` to work. The legacy **`compat32`** policy does not support these methods and should not be used with **`EmailMessage`**.

`email.parser`: Parsing email messages

Source code: Lib/email/parser.py
[https://github.com/python/cpython/tree/3.14/Lib/email/parser.py]

Message object structures can be created in one of two ways: they can be created from whole cloth by creating an **`EmailMessage`** object, adding headers using the dictionary interface, and adding payload(s) using **`set_content()`** and related methods, or they can be created by parsing a serialized representation of the email message.

The **`email`** package provides a standard parser that understands most email document structures, including MIME documents. You can pass the parser a bytes, string or file object, and the parser will return to you the root **`EmailMessage`** instance of the object structure. For simple, non-MIME messages the payload of this root object will likely be a string containing the text of the message. For MIME messages, the root object will return `True` from its **`is_multipart()`** method, and the subparts can be accessed via the payload manipulation methods, such as **`get_body()`**, **`iter_parts()`**, and **`walk()`**.

There are actually two parser interfaces available for use, the **`Parser`** API and the incremental **`FeedParser`** API. The **`Parser`** API is most useful if you have the entire text of the message in memory, or if the entire message lives in a file on the file system. **`FeedParser`** is more appropriate when you are reading the message from a stream which might block waiting for more input (such as reading an email message from a socket). The `FeedParser` can consume and parse the message incrementally, and only returns the root object when you close the parser.

Note that the parser can be extended in limited ways, and of course you can implement your own parser completely from scratch. All of the logic that connects the `email` package's bundled parser and the `EmailMessage` class is embodied in the `Policy` class, so a custom parser can create message object trees any way it finds necessary by implementing custom versions of the appropriate `Policy` methods.

FeedParser API

The `BytesFeedParser`, imported from the `email.feedparser` module, provides an API that is conducive to incremental parsing of email messages, such as would be necessary when reading the text of an email message from a source that can block (such as a socket). The `BytesFeedParser` can of course be used to parse an email message fully contained in a bytes-like object, string, or file, but the `BytesParser` API may be more convenient for such use cases. The semantics and results of the two parser APIs are identical.

The `BytesFeedParser`'s API is simple; you create an instance, feed it a bunch of bytes until there's no more to feed it, then close the parser to retrieve the root message object. The `BytesFeedParser` is extremely accurate when parsing standards-compliant messages, and it does a very good job of parsing non-compliant messages, providing information about how a message was deemed broken. It will populate a message object's `defects` attribute with a list of any problems it found in a message. See the `email.errors` module for the list of defects that it can find.

Here is the API for the `BytesFeedParser`:

class email.parser.BytesFeedParser(*_factory=None*, *, *policy=policy.compat32*)

Create a `BytesFeedParser` instance. Optional *_factory* is a no-argument callable; if not specified use the `message_factory` from the *policy*. Call *_factory* whenever a new message object is needed.

If *policy* is specified use the rules it specifies to update the representation of the message. If *policy* is not set, use the `compat32` policy, which maintains backward compatibility with the Python 3.2 version of the email package and provides `Message` as the default factory. All other policies provide `EmailMessage` as the default *_factory*. For more information on what else *policy* controls, see the `policy` documentation.

Note: **The policy keyword should always be specified**; The default will change to `email.policy.default` in a future version of Python.

Added in version 3.2.

Changed in version 3.3: Added the *policy* keyword.

Changed in version 3.6: *_factory* defaults to the policy `message_factory`.

feed(*data*)

Feed the parser some more data. *data* should be a bytes-like object containing one or more lines. The lines can be partial and the parser will stitch such partial lines together properly. The lines can have any of the three common line endings: carriage return, newline, or carriage return and newline (they can even be mixed).

close()

Complete the parsing of all previously fed data and return the root message object. It is undefined what happens if `feed()` is called after this method has been called.

class email.parser.FeedParser(*_factory=None*, *, *policy=policy.compat32*)

Works like `BytesFeedParser` except that the input to the `feed()` method must be a string. This is of limited utility, since the only way for such a message to be valid is for it to contain only ASCII text or, if `utf8` is `True`, no binary attachments.

Changed in version 3.3: Added the *policy* keyword.

Parser API

The `BytesParser` class, imported from the `email.parser` module, provides an API that can be used to parse a message when the complete contents of the message are available in a bytes-like object or file. The `email.parser` module also provides `Parser` for parsing strings, and header-only parsers, `BytesHeaderParser` and `HeaderParser`, which can be used if you're only interested in the headers of the message. `BytesHeaderParser` and `HeaderParser` can be much faster in these situations, since they do not attempt to parse the message body, instead setting the payload to the raw body.

class email.parser.BytesParser(*_class=None*, *, *policy=policy.compat32*)

Create a `BytesParser` instance. The *_class* and *policy* arguments have the same meaning and semantics as the *_factory* and *policy* arguments of `BytesFeedParser`.

Note: **The policy keyword should always be specified**; The default will change to `email.policy.default` in a future version of Python.

Changed in version 3.3: Removed the *strict* argument that was deprecated in 2.4. Added the *policy* keyword.

Changed in version 3.6: *_class* defaults to the policy `message_factory`.

parse(*fp*, *headersonly=False*)

Read all the data from the binary file-like object *fp*, parse the resulting bytes, and return the message object. *fp* must support both the `readline()` and the **`read()`** methods.

The bytes contained in *fp* must be formatted as a block of **RFC 5322** [https://datatracker.ietf.org/doc/html/rfc5322.html] (or, if `utf8` is `True`, **RFC 6532** [https://datatracker.ietf.org/doc/html/rfc6532.html]) style headers and header continuation lines, optionally preceded by an envelope header. The header block is terminated either by the end of the data or by a blank line. Following the header block is the body of the message (which may contain MIME-encoded subparts, including subparts with a *Content-Transfer-Encoding* of `8bit`).

Optional *headersonly* is a flag specifying whether to stop parsing after reading the headers or not. The default is `False`, meaning it parses the entire contents of the file.

parsebytes(*bytes*, *headersonly=False*)

Similar to the **`parse()`** method, except it takes a bytes-like object instead of a file-like object. Calling this method on a bytes-like object is equivalent to wrapping *bytes* in a **`BytesIO`** instance first and calling **`parse()`**.

Optional *headersonly* is as with the **`parse()`** method.

Added in version 3.2.

class email.parser.BytesHeaderParser(*_class=None*, *, *policy=policy.compat32*)

Exactly like **`BytesParser`**, except that *headersonly* defaults to `True`.

Added in version 3.3.

class email.parser.Parser(*_class=None*, *, *policy=policy.compat32*)

This class is parallel to **`BytesParser`**, but handles string input.

Changed in version 3.3: Removed the *strict* argument. Added the *policy* keyword.

Changed in version 3.6: *_class* defaults to the policy `message_factory`.

parse(*fp*, *headersonly=False*)

Read all the data from the text-mode file-like object *fp*, parse the resulting text, and return the root message object. *fp* must support both the **`readline()`** and the **`read()`** methods on file-like objects.

Other than the text mode requirement, this method operates like `BytesParser.parse()`.

parsestr(*text*, *headersonly=False*)

Similar to the `parse()` method, except it takes a string object instead of a file-like object. Calling this method on a string is equivalent to wrapping *text* in a `StringIO` instance first and calling `parse()`.

Optional *headersonly* is as with the `parse()` method.

class email.parser.HeaderParser(*_class=None*, *, *policy=policy.compat32*)

Exactly like `Parser`, except that *headersonly* defaults to `True`.

Since creating a message object structure from a string or a file object is such a common task, four functions are provided as a convenience. They are available in the top-level `email` package namespace.

email.message_from_bytes(*s*, *_class=None*, *, *policy=policy.compat32*)

Return a message object structure from a bytes-like object. This is equivalent to `BytesParser().parsebytes(s)`. Optional *_class* and *policy* are interpreted as with the `BytesParser` class constructor.

Added in version 3.2.

Changed in version 3.3: Removed the *strict* argument. Added the *policy* keyword.

email.message_from_binary_file(*fp*, *_class=None*, *, *policy=policy.compat32*)

Return a message object structure tree from an open binary file object. This is equivalent to `BytesParser().parse(fp)`. *_class* and *policy* are interpreted as with the **`BytesParser`** class constructor.

Added in version 3.2.

Changed in version 3.3: Removed the *strict* argument. Added the *policy* keyword.

email.message_from_string(*s*, *_class=None*, *, *policy=policy.compat32*)

Return a message object structure from a string. This is equivalent to `Parser().parsestr(s)`. *_class* and *policy* are interpreted as with the **`Parser`** class constructor.

Changed in version 3.3: Removed the *strict* argument. Added the *policy* keyword.

email.message_from_file(*fp*, *_class=None*, *, *policy=policy.compat32*)

Return a message object structure tree from an open file object. This is equivalent to `Parser().parse(fp)`. *_class* and *policy* are interpreted as with the **`Parser`** class constructor.

Changed in version 3.3: Removed the *strict* argument. Added the *policy* keyword.

Changed in version 3.6: *_class* defaults to the policy `message_factory`.

Here’s an example of how you might use **`message_from_bytes()`** at an interactive Python prompt:

```
>>> import email
>>> msg = email.message_from_bytes(myBytes)
```

Additional notes

Here are some notes on the parsing semantics:

- Most non-*multipart* type messages are parsed as a single message object with a string payload. These objects will return `False` for `is_multipart()`, and `iter_parts()` will yield an empty list.
- All *multipart* type messages will be parsed as a container message object with a list of sub-message objects for their payload. The outer container message will return `True` for `is_multipart()`, and `iter_parts()` will yield a list of subparts.
- Most messages with a content type of *message/** (such as *message/delivery-status* and *message/rfc822*) will also be parsed as container object containing a list payload of length 1. Their `is_multipart()` method will return `True`. The single element yielded by `iter_parts()` will be a sub-message object.
- Some non-standards-compliant messages may not be internally consistent about their *multipart*-edness. Such messages may have a *Content-Type* header of type *multipart*, but their `is_multipart()` method may return `False`. If such messages were parsed with the `FeedParser`, they will have an instance of the `MultipartInvariantViolationDefect` class in their *defects* attribute list. See `email.errors` for details.

`email.generator`: Generating MIME documents

Source code: Lib/email/generator.py
[https://github.com/python/cpython/tree/3.14/Lib/email/generator.py]

One of the most common tasks is to generate the flat (serialized) version of the email message represented by a message object structure. You will need to do this if you want to send your message via `smtplib.SMTP.sendmail()`, or print the message on the console. Taking a message object structure and producing a serialized representation is the job of the generator classes.

As with the `email.parser` module, you aren't limited to the functionality of the bundled generator; you could write one from scratch yourself. However the bundled generator knows how to generate most email in a standards-compliant way, should handle MIME and non-MIME email messages just fine, and is designed so that the bytes-oriented parsing and generation operations are inverses, assuming the same non-transforming `policy` is used for both. That is, parsing the serialized byte stream via the `BytesParser` class and then regenerating the serialized byte stream using `BytesGenerator` should produce output identical to the input [[1]]. (On the other hand, using the generator on an `EmailMessage` constructed by program may result in changes to the `EmailMessage` object as defaults are filled in.)

The `Generator` class can be used to flatten a message into a text (as opposed to binary) serialized representation, but since Unicode cannot represent binary data directly, the message is of necessity transformed into something that contains only ASCII characters, using the standard email RFC Content Transfer Encoding techniques for encoding email messages for transport over channels that are not “8 bit clean”.

To accommodate reproducible processing of SMIME-signed messages **`Generator`** disables header folding for message parts of type `multipart/signed` and all subparts.

class email.generator.BytesGenerator(*outfp*, *mangle_from_=None*, *maxheaderlen=None*, *, *policy=None*)

Return a **`BytesGenerator`** object that will write any message provided to the **`flatten()`** method, or any surrogateescape encoded text provided to the **`write()`** method, to the file-like object *outfp*. *outfp* must support a `write` method that accepts binary data.

If optional *mangle_from_* is `True`, put a `>` character in front of any line in the body that starts with the exact string `"From "`, that is `From` followed by a space at the beginning of a line. *mangle_from_* defaults to the value of the **`mangle_from_`** setting of the *policy* (which is `True` for the **`compat32`** policy and `False` for all others). *mangle_from_* is intended for use when messages are stored in Unix mbox format (see **`mailbox`** and WHY THE CONTENT-LENGTH FORMAT IS BAD [https://www.jwz.org/doc/content-length.html]).

If *maxheaderlen* is not `None`, refold any header lines that are longer than *maxheaderlen*, or if `0`, do not rewrap any headers. If *manheaderlen* is `None` (the default), wrap headers and other message lines according to the *policy* settings.

If *policy* is specified, use that policy to control message generation. If *policy* is `None` (the default), use the policy associated with the **`Message`** or **`EmailMessage`** object passed to `flatten` to control the message generation. See **`email.policy`** for details on what *policy* controls.

Added in version 3.2.

Changed in version 3.3: Added the *policy* keyword.

Changed in version 3.6: The default behavior of the *mangle_from_* and *maxheaderlen* parameters is to follow the policy.

flatten(*msg*, *unixfrom=False*, *linesep=None*)

> Print the textual representation of the message object structure rooted at *msg* to the output file specified when the **`BytesGenerator`** instance was created.
>
> If the **`policy`** option **`cte_type`** is `8bit` (the default), copy any headers in the original parsed message that have not been modified to the output with any bytes with the high bit set reproduced as in the original, and preserve the non-ASCII *Content-Transfer-Encoding* of any body parts that have them. If `cte_type` is `7bit`, convert the bytes with the high bit set as needed using an ASCII-compatible *Content-Transfer-Encoding*. That is, transform parts with non-ASCII *Content-Transfer-Encoding* (*Content-Transfer-Encoding: 8bit*) to an ASCII compatible *Content-Transfer-Encoding*, and encode RFC-invalid non-ASCII bytes in headers using the MIME `unknown-8bit` character set, thus rendering them RFC-compliant.
>
> If *unixfrom* is `True`, print the envelope header delimiter used by the Unix mailbox format (see **`mailbox`**) before the first of the **RFC 5322** [https://datatracker.ietf.org/doc/html/rfc5322.html] headers of the

root message object. If the root object has no envelope header, craft a standard one. The default is `False`. Note that for subparts, no envelope header is ever printed.

If *linesep* is not `None`, use it as the separator character between all the lines of the flattened message. If *linesep* is `None` (the default), use the value specified in the *policy*.

clone(*fp*)

Return an independent clone of this `BytesGenerator` instance with the exact same option settings, and *fp* as the new *outfp*.

write(*s*)

Encode *s* using the `ASCII` codec and the `surrogateescape` error handler, and pass it to the *write* method of the *outfp* passed to the `BytesGenerator`'s constructor.

As a convenience, `EmailMessage` provides the methods `as_bytes()` and `bytes(aMessage)` (a.k.a. `__bytes__()`), which simplify the generation of a serialized binary representation of a message object. For more detail, see `email.message`.

Because strings cannot represent binary data, the `Generator` class must convert any binary data in any message it flattens to an ASCII compatible format, by converting them to an ASCII compatible *Content-Transfer_Encoding*. Using the terminology of the email RFCs, you can think of this as `Generator` serializing to an I/O stream that is not “8 bit clean”. In other words, most applications will want to be using `BytesGenerator`, and not `Generator`.

class email.generator.Generator(*outfp*, *mangle_from_=None*, *maxheaderlen=None*, *, *policy=None*)

Return a **`Generator`** object that will write any message provided to the **`flatten()`** method, or any text provided to the **`write()`** method, to the file-like object *outfp*. *outfp* must support a `write` method that accepts string data.

If optional *mangle_from_* is `True`, put a `>` character in front of any line in the body that starts with the exact string `"From "`, that is `From` followed by a space at the beginning of a line. *mangle_from_* defaults to the value of the **`mangle_from_`** setting of the *policy* (which is `True` for the **`compat32`** policy and `False` for all others). *mangle_from_* is intended for use when messages are stored in Unix mbox format (see **`mailbox`** and WHY THE CONTENT-LENGTH FORMAT IS BAD [https://www.jwz.org/doc/content-length.html]).

If *maxheaderlen* is not `None`, refold any header lines that are longer than *maxheaderlen*, or if `0`, do not rewrap any headers. If *manheaderlen* is `None` (the default), wrap headers and other message lines according to the *policy* settings.

If *policy* is specified, use that policy to control message generation. If *policy* is `None` (the default), use the policy associated with the **`Message`** or **`EmailMessage`** object passed to `flatten` to control the message generation. See **`email.policy`** for details on what *policy* controls.

Changed in version 3.3: Added the *policy* keyword.

Changed in version 3.6: The default behavior of the *mangle_from_* and *maxheaderlen* parameters is to follow the policy.

flatten(*msg*, *unixfrom=False*, *linesep=None*)

Print the textual representation of the message object structure rooted at *msg* to the output file specified when the `Generator` instance was created.

If the `policy` option `cte_type` is `8bit`, generate the message as if the option were set to `7bit`. (This is required because strings cannot represent non-ASCII bytes.) Convert any bytes with the high bit set as needed using an ASCII-compatible *Content-Transfer-Encoding*. That is, transform parts with non-ASCII *Content-Transfer-Encoding* (*Content-Transfer-Encoding: 8bit*) to an ASCII compatible *Content-Transfer-Encoding*, and encode RFC-invalid non-ASCII bytes in headers using the MIME `unknown-8bit` character set, thus rendering them RFC-compliant.

If *unixfrom* is `True`, print the envelope header delimiter used by the Unix mailbox format (see `mailbox`) before the first of the **RFC 5322** [https://datatracker.ietf.org/doc/html/rfc5322.html] headers of the root message object. If the root object has no envelope header, craft a standard one. The default is `False`. Note that for subparts, no envelope header is ever printed.

If *linesep* is not `None`, use it as the separator character between all the lines of the flattened message. If *linesep* is `None` (the default), use the value specified in the *policy*.

Changed in version 3.2: Added support for re-encoding `8bit` message bodies, and the *linesep* argument.

clone(*fp*)

Return an independent clone of this `Generator` instance with the exact same options, and *fp* as the new *outfp*.

write(*s*)

Write *s* to the *write* method of the *outfp* passed to the `Generator`'s constructor. This provides just enough file-like API for `Generator` instances to be used in the `print()` function.

As a convenience, `EmailMessage` provides the methods `as_string()` and `str(aMessage)` (a.k.a. `__str__()`), which simplify the generation of a formatted string representation of a message object. For more detail, see `email.message`.

The `email.generator` module also provides a derived class, `DecodedGenerator`, which is like the `Generator` base class, except that non-*text* parts are not serialized, but are instead represented in the output stream by a string derived from a template filled in with information about the part.

class email.generator.DecodedGenerator(*outfp*, *mangle_from_=None*, *maxheaderlen=None*, *fmt=None*, *, *policy=None*)

Act like `Generator`, except that for any subpart of the message passed to `Generator.flatten()`, if the subpart is of main type *text*, print the decoded payload of the subpart, and if the main type is not *text*, instead of printing it fill in the string *fmt* using information from the part and print the resulting filled-in string.

To fill in *fmt*, execute `fmt % part_info`, where `part_info` is a dictionary composed of the following keys and values:

- `type` – Full MIME type of the non-*text* part
- `maintype` – Main MIME type of the non-*text* part
- `subtype` – Sub-MIME type of the non-*text* part
- `filename` – Filename of the non-*text* part
- `description` – Description associated with the non-*text* part

- `encoding` – Content transfer encoding of the non-*text* part

If *fmt* is `None`, use the following default *fmt*:

> “[Non-text (%(type)s) part of message omitted, filename %(filename)s]”

Optional *_mangle_from_* and *maxheaderlen* are as with the **`Generator`** base class.

Footnotes

[1]

This statement assumes that you use the appropriate setting for `unixfrom`, and that there are no **`email.policy`** settings calling for automatic adjustments (for example, **`refold_source`** must be `none`, which is *not* the default). It is also not 100% true, since if the message does not conform to the RFC standards occasionally information about the exact original text is lost during parsing error recovery. It is a goal to fix these latter edge cases when possible.

`email.policy`: Policy Objects

Added in version 3.3.

Source code: Lib/email/policy.py
[https://github.com/python/cpython/tree/3.14/Lib/email/policy.py]

The **`email`** package's prime focus is the handling of email messages as described by the various email and MIME RFCs. However, the general format of email messages (a block of header fields each consisting of a name followed by a colon followed by a value, the whole block followed by a blank line and an arbitrary 'body'), is a format that has found utility outside of the realm of email. Some of these uses conform fairly closely to the main email RFCs, some do not. Even when working with email, there are times when it is desirable to break strict compliance with the RFCs, such as generating emails that interoperate with email servers that do not themselves follow the standards, or that implement extensions you want to use in ways that violate the standards.

Policy objects give the email package the flexibility to handle all these disparate use cases.

A **`Policy`** object encapsulates a set of attributes and methods that control the behavior of various components of the email package during use. **`Policy`** instances can be passed to various classes and methods in the email package to alter the default behavior. The settable values and their defaults are described below.

There is a default policy used by all classes in the email package. For all of the `parser` classes and the related convenience functions, and for the `Message` class, this is the `Compat32` policy, via its corresponding pre-defined instance `compat32`. This policy provides for complete backward compatibility (in some cases, including bug compatibility) with the pre-Python3.3 version of the email package.

This default value for the *policy* keyword to `EmailMessage` is the `EmailPolicy` policy, via its pre-defined instance `default`.

When a `Message` or `EmailMessage` object is created, it acquires a policy. If the message is created by a `parser`, a policy passed to the parser will be the policy used by the message it creates. If the message is created by the program, then the policy can be specified when it is created. When a message is passed to a `generator`, the generator uses the policy from the message by default, but you can also pass a specific policy to the generator that will override the one stored on the message object.

The default value for the *policy* keyword for the `email.parser` classes and the parser convenience functions **will be changing** in a future version of Python. Therefore you should **always specify explicitly which policy you want to use** when calling any of the classes and functions described in the `parser` module.

The first part of this documentation covers the features of `Policy`, an abstract base class that defines the features that are common to all policy objects, including `compat32`. This includes certain hook methods that are called internally by the email package, which a custom policy could override to obtain different behavior. The second part describes the concrete classes `EmailPolicy` and `Compat32`, which implement the hooks that provide the standard behavior and the backward compatible behavior and features, respectively.

`Policy` instances are immutable, but they can be cloned, accepting the same keyword arguments as the class constructor and returning a new **`Policy`** instance that is a copy of the original but with the specified attributes values changed.

As an example, the following code could be used to read an email message from a file on disk and pass it to the system `sendmail` program on a Unix system:

```
>>> from email import message_from_binary_file
>>> from email.generator import BytesGenerator
>>> from email import policy
>>> from subprocess import Popen, PIPE
>>> with open('mymsg.txt', 'rb') as f:
...     msg = message_from_binary_file(f,
policy=policy.default)
...
>>> p = Popen(['sendmail', msg['To'].addresses[0]], stdin=PIPE)
>>> g = BytesGenerator(p.stdin,
policy=msg.policy.clone(linesep='\r\n'))
>>> g.flatten(msg)
>>> p.stdin.close()
>>> rc = p.wait()
```

Here we are telling **`BytesGenerator`** to use the RFC correct line separator characters when creating the binary string to feed into `sendmail's` `stdin`, where the default policy would use `\n` line separators.

Some email package methods accept a *policy* keyword argument, allowing the policy to be overridden for that method. For example, the following code uses the **`as_bytes()`** method of the *msg* object from the previous example and writes the message to a file using the native line separators for the platform on which it is running:

```
>>> import os
>>> with open('converted.txt', 'wb') as f:
...
f.write(msg.as_bytes(policy=msg.policy.clone(linesep=os.linesep
)))
17
```

Policy objects can also be combined using the addition operator, producing a policy object whose settings are a combination of the non-default values of the summed objects:

```
>>> compat_SMTP = policy.compat32.clone(linesep='\r\n')
>>> compat_strict = policy.compat32.clone(raise_on_defect=True)
>>> compat_strict_SMTP = compat_SMTP + compat_strict
```

This operation is not commutative; that is, the order in which the objects are added matters. To illustrate:

```
>>> policy100 = policy.compat32.clone(max_line_length=100)
>>> policy80 = policy.compat32.clone(max_line_length=80)
>>> apolicy = policy100 + policy80
>>> apolicy.max_line_length
80
>>> apolicy = policy80 + policy100
>>> apolicy.max_line_length
100
```

class email.policy.Policy(***kw*)

This is the abstract base class for all policy classes. It provides default implementations for a couple of trivial methods, as well as the implementation of the immutability property, the **`clone()`** method, and the constructor semantics.

The constructor of a policy class can be passed various keyword arguments. The arguments that may be specified are any non-method properties on this class, plus any additional non-method properties on the concrete class. A value specified in the constructor will override the default value for the corresponding attribute.

This class defines the following properties, and thus values for the following may be passed in the constructor of any policy class:

max_line_length

The maximum length of any line in the serialized output, not counting the end of line character(s). Default is 78, per **RFC 5322** [https://datatracker.ietf.org/doc/html/rfc5322.html]. A value of `0` or `None` indicates that no line wrapping should be done at all.

linesep

The string to be used to terminate lines in serialized output. The default is `\n` because that's the internal end-of-line discipline used by Python, though `\r\n` is required by the RFCs.

cte_type

Controls the type of Content Transfer Encodings that may be or are required to be used. The possible values are:

`7bit`	all data must be "7 bit clean" (ASCII-only). This means that where necessary data will be encoded using either quoted-printable or base64 encoding.
`8bit`	data is not constrained to be 7 bit clean. Data in headers is still required to be ASCII-only and so will be encoded (see `fold_binary()` and `utf8`

below for exceptions), but body parts may use the `8bit` CTE.

A `cte_type` value of `8bit` only works with `BytesGenerator`, not `Generator`, because strings cannot contain binary data. If a `Generator` is operating under a policy that specifies `cte_type=8bit`, it will act as if `cte_type` is `7bit`.

raise_on_defect

If **`True`**, any defects encountered will be raised as errors. If **`False`** (the default), defects will be passed to the **`register_defect()`** method.

mangle_from_

If **`True`**, lines starting with *"From "* in the body are escaped by putting a `>` in front of them. This parameter is used when the message is being serialized by a generator. Default: **`False`**.

Added in version 3.5.

message_factory

A factory function for constructing a new empty message object. Used by the parser when building messages. Defaults to `None`, in which case **`Message`** is used.

Added in version 3.6.

verify_generated_headers

If `True` (the default), the generator will raise **`HeaderWriteError`** instead of writing a header that is improperly folded or delimited, such that it would be parsed as multiple headers or joined with adjacent data. Such headers can be generated by custom header classes or bugs in the `email` module.

As it’s a security feature, this defaults to `True` even in the **`Compat32`** policy. For backwards compatible, but unsafe, behavior, it must be set to `False` explicitly.

Added in version 3.13.

The following **`Policy`** method is intended to be called by code using the email library to create policy instances with custom settings:

clone(***kw*)

Return a new **`Policy`** instance whose attributes have the same values as the current instance, except where those attributes are given new values by the keyword arguments.

The remaining **`Policy`** methods are called by the email package code, and are not intended to be called by an application using the email package. A custom policy must implement all of these methods.

handle_defect(*obj*, *defect*)

Handle a *defect* found on *obj*. When the email package calls this method, *defect* will always be a subclass of **`MessageDefect`**.

The default implementation checks the **`raise_on_defect`** flag. If it is `True`, *defect* is raised as an exception. If it is `False` (the default), *obj* and *defect* are passed to **`register_defect()`**.

register_defect(*obj*, *defect*)

Register a *defect* on *obj*. In the email package, *defect* will always be a subclass of **`MessageDefect`**.

The default implementation calls the `append` method of the `defects` attribute of *obj*. When the email package calls **`handle_defect`**, *obj* will normally have a `defects` attribute that has an `append` method. Custom object types used with the email package (for example, custom `Message` objects) should also provide such an attribute, otherwise defects in parsed messages will raise unexpected errors.

header_max_count(*name*)

Return the maximum allowed number of headers named *name*.

Called when a header is added to an **`EmailMessage`** or **`Message`** object. If the returned value is not `0` or `None`, and there are already a number of headers with the name *name* greater than or equal to the value returned, a **`ValueError`** is raised.

Because the default behavior of `Message.__setitem__` is to append the value to the list of headers, it is easy to create duplicate headers without realizing it. This method allows certain headers to be limited in the number of instances of that header that may be added to a `Message` programmatically. (The limit is not observed by the parser, which will faithfully produce as many headers as exist in the message being parsed.)

The default implementation returns `None` for all header names.

header_source_parse(*sourcelines*)

The email package calls this method with a list of strings, each string ending with the line separation characters found in the source being parsed. The first line includes the field header name and separator. All whitespace in the source is preserved. The method should return the `(name, value)` tuple that is to be stored in the `Message` to represent the parsed header.

If an implementation wishes to retain compatibility with the existing email package policies, *name* should be the case preserved name (all characters up to the '`:`' separator), while *value* should be the unfolded value (all line separator characters removed, but whitespace kept intact), stripped of leading whitespace.

sourcelines may contain surrogateescaped binary data.

There is no default implementation

header_store_parse(*name*, *value*)

The email package calls this method with the name and value provided by the application program when the application program is modifying a `Message` programmatically (as opposed to a `Message` created by a parser). The method should return the `(name, value)` tuple that is to be stored in the `Message` to represent the header.

If an implementation wishes to retain compatibility with the existing email package policies, the *name* and *value* should be strings or string subclasses that do not change the content of the passed in arguments.

There is no default implementation

header_fetch_parse(*name*, *value*)

The email package calls this method with the *name* and *value* currently stored in the `Message` when that header is requested by the application program, and whatever the method returns is what is passed back to the application as the value of the header being retrieved. Note that there may be more than one header with the same name stored in the `Message`; the method is passed the specific name and value of the header destined to be returned to the application.

value may contain surrogateescaped binary data. There should be no surrogateescaped binary data in the value returned by the method.

There is no default implementation

fold(*name*, *value*)

The email package calls this method with the *name* and *value* currently stored in the `Message` for a given header. The method should return a string that represents that header “folded” correctly (according to the policy settings) by composing the *name* with the *value* and inserting **`linesep`** characters at the appropriate places. See **RFC 5322** [https://datatracker.ietf.org/doc/html/rfc5322.html] for a discussion of the rules for folding email headers.

value may contain surrogateescaped binary data. There should be no surrogateescaped binary data in the string returned by the method.

fold_binary(*name*, *value*)

The same as **`fold()`**, except that the returned value should be a bytes object rather than a string.

value may contain surrogateescaped binary data. These could be converted back into binary data in the returned bytes object.

class email.policy.EmailPolicy(***kw*)

This concrete **`Policy`** provides behavior that is intended to be fully compliant with the current email RFCs. These include (but are not limited to) **RFC 5322** [https://datatracker.ietf.org/doc/html/rfc5322.html], **RFC 2047** [https://datatracker.ietf.org/doc/html/rfc2047.html], and the current MIME RFCs.

This policy adds new header parsing and folding algorithms. Instead of simple strings, headers are `str` subclasses with attributes that depend on the type of the field. The parsing and folding algorithm fully implement **RFC 2047** [https://datatracker.ietf.org/doc/html/rfc2047.html] and **RFC 5322** [https://datatracker.ietf.org/doc/html/rfc5322.html].

The default value for the **`message_factory`** attribute is **`EmailMessage`**.

In addition to the settable attributes listed above that apply to all policies, this policy adds the following additional attributes:

Added in version 3.6: [[1]]

utf8

If `False`, follow **RFC 5322** [https://datatracker.ietf.org/doc/html/rfc5322.html], supporting non-ASCII characters in headers by encoding them as “encoded words”. If `True`, follow **RFC 6532** [https://datatracker.ietf.org/doc/html/rfc6532.html] and use `utf-8` encoding for headers. Messages formatted in this way may be passed to SMTP servers that support the

`SMTPUTF8` extension (**RFC 6531** [https://datatracker.ietf.org/doc/html/rfc6531.html]).

refold_source

If the value for a header in the `Message` object originated from a **`parser`** (as opposed to being set by a program), this attribute indicates whether or not a generator should refold that value when transforming the message back into serialized form. The possible values are:

`none`	all source values use original folding
`long`	source values that have any line that is longer than `max_line_length` will be refolded
`all`	all values are refolded.

The default is `long`.

header_factory

A callable that takes two arguments, `name` and `value`, where `name` is a header field name and `value` is an unfolded header field value, and returns a string subclass that represents that header. A default `header_factory` (see **`headerregistry`**) is provided that supports custom parsing for the various address and date **RFC 5322** [https://datatracker.ietf.org/doc/html/rfc5322.html] header field types, and the major MIME header field stypes. Support for additional custom parsing will be added in the future.

content_manager

An object with at least two methods: get_content and set_content. When the **`get_content()`** or **`set_content()`** method of an **`EmailMessage`** object is called, it calls the corresponding method of this object, passing it the message object as its first argument, and any arguments or keywords that were passed to it as additional arguments. By default `content_manager` is set to **`raw_data_manager`**.

Added in version 3.4.

The class provides the following concrete implementations of the abstract methods of **`Policy`**:

header_max_count(*name*)

Returns the value of the **`max_count`** attribute of the specialized class used to represent the header with the given name.

header_source_parse(*sourcelines*)

The name is parsed as everything up to the '`:`' and returned unmodified. The value is determined by stripping leading whitespace off the remainder of the first line, joining all subsequent lines together, and stripping any trailing carriage return or linefeed characters.

header_store_parse(*name*, *value*)

The name is returned unchanged. If the input value has a `name` attribute and it matches *name* ignoring case, the value is returned unchanged. Otherwise the *name* and *value* are passed to `header_factory`, and the resulting header object is returned as the value. In this case a `ValueError` is raised if the input value contains CR or LF characters.

header_fetch_parse(*name*, *value*)

If the value has a `name` attribute, it is returned to unmodified. Otherwise the *name*, and the *value* with any CR or LF characters removed, are passed to the `header_factory`, and the resulting header object is returned. Any surrogateescaped bytes get turned into the unicode unknown-character glyph.

fold(*name*, *value*)

Header folding is controlled by the **`refold_source`** policy setting. A value is considered to be a 'source value' if and only if it does not have a `name` attribute (having a `name` attribute means it is a header object of some sort). If a source value needs to be refolded according to the policy, it is converted into a header object by passing the *name* and the *value* with any CR and LF characters removed to the `header_factory`. Folding of a header object is done by calling its `fold` method with the current policy.

Source values are split into lines using **`splitlines()`**. If the value is not to be refolded, the lines are rejoined using the `linesep` from the policy and returned. The exception is lines containing non-ascii binary data. In that case the value is refolded regardless of the `refold_source` setting, which causes the binary data to be CTE encoded using the `unknown-8bit` charset.

fold_binary(*name*, *value*)

The same as **`fold()`** if **`cte_type`** is `7bit`, except that the returned value is bytes.

If **`cte_type`** is `8bit`, non-ASCII binary data is converted back into bytes. Headers with binary data are not refolded, regardless of the `refold_header` setting, since there is no way to know whether the binary data consists of single byte characters or multibyte characters.

The following instances of `EmailPolicy` provide defaults suitable for specific application domains. Note that in the future the behavior of these instances (in particular the `HTTP` instance) may be adjusted to conform even more closely to the RFCs relevant to their domains.

email.policy.default

An instance of `EmailPolicy` with all defaults unchanged. This policy uses the standard Python `\n` line endings rather than the RFC-correct `\r\n`.

email.policy.SMTP

Suitable for serializing messages in conformance with the email RFCs. Like `default`, but with `linesep` set to `\r\n`, which is RFC compliant.

email.policy.SMTPUTF8

The same as `SMTP` except that `utf8` is `True`. Useful for serializing messages to a message store without using encoded words in the headers. Should only be used for SMTP transmission if the sender or recipient addresses have non-ASCII characters (the `smtplib.SMTP.send_message()` method handles this automatically).

email.policy.HTTP

Suitable for serializing headers with for use in HTTP traffic. Like `SMTP` except that `max_line_length` is set to `None` (unlimited).

email.policy.strict

Convenience instance. The same as `default` except that `raise_on_defect` is set to `True`. This allows any policy to be made strict by writing:

```
somepolicy + policy.strict
```

With all of these `EmailPolicies`, the effective API of the email package is changed from the Python 3.2 API in the following ways:

- Setting a header on a `Message` results in that header being parsed and a header object created.
- Fetching a header value from a `Message` results in that header being parsed and a header object created and returned.
- Any header object, or any header that is refolded due to the policy settings, is folded using an algorithm that fully implements the RFC folding algorithms, including knowing where encoded words are required and allowed.

From the application view, this means that any header obtained through the `EmailMessage` is a header object with extra attributes, whose string value is the fully decoded unicode value of the header. Likewise, a header may be assigned a new value, or a new header created, using a unicode string, and the policy will take care of converting the unicode string into the correct RFC encoded form.

The header objects and their attributes are described in `headerregistry`.

class email.policy.Compat32(***kw*)

This concrete `Policy` is the backward compatibility policy. It replicates the behavior of the email package in Python 3.2. The `policy` module also defines an instance of this class, `compat32`, that is used as the default policy. Thus the default behavior of the email package is to maintain compatibility with Python 3.2.

The following attributes have values that are different from the `Policy` default:

mangle_from_

The default is `True`.

The class provides the following concrete implementations of the abstract methods of `Policy`:

header_source_parse(*sourcelines*)

The name is parsed as everything up to the ‘`:`’ and returned unmodified. The value is determined by stripping leading whitespace off the remainder of the first line, joining all subsequent lines together, and stripping any trailing carriage return or linefeed characters.

header_store_parse(*name*, *value*)

The name and value are returned unmodified.

header_fetch_parse(*name*, *value*)

If the value contains binary data, it is converted into a `Header` object using the `unknown-8bit` charset. Otherwise it is returned unmodified.

fold(*name*, *value*)

Headers are folded using the `Header` folding algorithm, which preserves existing line breaks in the value, and wraps each resulting line to the `max_line_length`. Non-ASCII binary data are CTE encoded using the `unknown-8bit` charset.

fold_binary(*name*, *value*)

Headers are folded using the `Header` folding algorithm, which preserves existing line breaks in the value, and wraps each resulting line to the `max_line_length`. If `cte_type` is `7bit`, non-ascii binary data is CTE encoded using the `unknown-8bit` charset. Otherwise the original source header is used, with its existing line breaks and any (RFC invalid) binary data it may contain.

email.policy.compat32

An instance of `Compat32`, providing backward compatibility with the behavior of the email package in Python 3.2.

Note

The `compat32` policy should not be used as a policy for `EmailMessage` objects, and should only be used to serialize messages that were created using the `compat32` policy.

Footnotes

[1]

Originally added in 3.3 as a provisional feature.

`email.errors`: Exception and Defect classes

Source code: Lib/email/errors.py [https://github.com/python/cpython/tree/3.14/Lib/email/errors.py]

The following exception classes are defined in the `email.errors` module:

exception email.errors.MessageError

This is the base class for all exceptions that the `email` package can raise. It is derived from the standard `Exception` class and defines no additional methods.

exception email.errors.MessageParseError

This is the base class for exceptions raised by the `Parser` class. It is derived from `MessageError`. This class is also used internally by the parser used by `headerregistry`.

exception email.errors.HeaderParseError

Raised under some error conditions when parsing the **RFC 5322** [https://datatracker.ietf.org/doc/html/rfc5322.html] headers of a message, this class is derived from `MessageParseError`. The `set_boundary()` method will raise this error if the content type is unknown when the method is called. `Header` may raise this error for certain base64 decoding errors, and when an attempt is made to create a header that appears to contain an embedded header (that is, there is what is supposed to be a continuation line that has no leading whitespace and looks like a header).

exception email.errors.BoundaryError

Deprecated and no longer used.

exception email.errors.MultipartConversionError

Raised if the `attach()` method is called on an instance of a class derived from `MIMENonMultipart` (e.g. `MIMEImage`). `MultipartConversionError` multiply inherits from `MessageError` and the built-in `TypeError`.

exception email.errors.HeaderWriteError

Raised when an error occurs when the `generator` outputs headers.

exception email.errors.MessageDefect

This is the base class for all defects found when parsing email messages. It is derived from `ValueError`.

exception email.errors.HeaderDefect

This is the base class for all defects found when parsing email headers. It is derived from `MessageDefect`.

Here is the list of the defects that the `FeedParser` can find while parsing messages. Note that the defects are added to the message where the problem was found, so for example, if a message nested inside a *multipart/alternative* had a malformed header, that nested message object would have a defect, but the containing messages would not.

All defect classes are subclassed from `email.errors.MessageDefect`.

exception email.errors.NoBoundaryInMultipartDefect

A message claimed to be a multipart, but had no *boundary* parameter.

exception email.errors.StartBoundaryNotFoundDefect

The start boundary claimed in the *Content-Type* header was never found.

exception email.errors.CloseBoundaryNotFoundDefect

A start boundary was found, but no corresponding close boundary was ever found.

Added in version 3.3.

exception email.errors.FirstHeaderLineIsContinuationDefect

The message had a continuation line as its first header line.

exception email.errors.MisplacedEnvelopeHeaderDefect

A “Unix From” header was found in the middle of a header block.

exception email.errors.MissingHeaderBodySeparatorDefect

A line was found while parsing headers that had no leading white space but contained no ‘:’. Parsing continues assuming that the line represents the first line of the body.

Added in version 3.3.

exception email.errors.MalformedHeaderDefect

A header was found that was missing a colon, or was otherwise malformed.

Deprecated since version 3.3: This defect has not been used for several Python versions.

exception email.errors.MultipartInvariantViolationDefect

A message claimed to be a *multipart*, but no subparts were found. Note that when a message has this defect, its `is_multipart()` method may return `False` even though its content type claims to be *multipart*.

exception email.errors.InvalidBase64PaddingDefect

When decoding a block of base64 encoded bytes, the padding was not correct. Enough padding is added to perform the decode, but the resulting decoded bytes may be invalid.

exception email.errors.InvalidBase64CharactersDefect

When decoding a block of base64 encoded bytes, characters outside the base64 alphabet were encountered. The characters are ignored, but the resulting decoded bytes may be invalid.

exception email.errors.InvalidBase64LengthDefect

When decoding a block of base64 encoded bytes, the number of non-padding base64 characters was invalid (1 more than a multiple of 4). The encoded block was kept as-is.

exception email.errors.InvalidDateDefect

When decoding an invalid or unparsable date field. The original value is kept as-is.

`email.headerregistry`: Custom Header Objects

Source code: Lib/email/headerregistry.py [https://github.com/python/cpython/tree/3.14/Lib/email/headerregistry.py]

Added in version 3.6: [[1]]

Headers are represented by customized subclasses of **`str`**. The particular class used to represent a given header is determined by the **`header_factory`** of the **`policy`** in effect when the headers are created. This section documents the particular `header_factory` implemented by the email package for handling **RFC 5322** [https://datatracker.ietf.org/doc/html/rfc5322.html] compliant email messages, which not only provides customized header objects for various header types, but also provides an extension mechanism for applications to add their own custom header types.

When using any of the policy objects derived from **`EmailPolicy`**, all headers are produced by **`HeaderRegistry`** and have **`BaseHeader`** as their last base class. Each header class has an additional base class that is determined by the type of the header. For example, many headers have the class **`UnstructuredHeader`** as their other base class. The specialized second class for a header is determined by the name of the header, using a lookup table stored in the **`HeaderRegistry`**. All of this is managed transparently for the typical application program, but interfaces are provided for modifying the default behavior for use by more complex applications.

The sections below first document the header base classes and their attributes, followed by the API for modifying the behavior of `HeaderRegistry`, and finally the support classes used to represent the data parsed from structured headers.

class email.headerregistry.BaseHeader(*name*, *value*)

name and *value* are passed to `BaseHeader` from the `header_factory` call. The string value of any header object is the *value* fully decoded to unicode.

This base class defines the following read-only properties:

name

The name of the header (the portion of the field before the ':'). This is exactly the value passed in the `header_factory` call for *name*; that is, case is preserved.

defects

A tuple of `HeaderDefect` instances reporting any RFC compliance problems found during parsing. The email package tries to be complete about detecting compliance issues. See the `errors` module for a discussion of the types of defects that may be reported.

max_count

The maximum number of headers of this type that can have the same `name`. A value of `None` means unlimited. The `BaseHeader` value for this attribute is `None`; it is expected that specialized header classes will override this value as needed.

`BaseHeader` also provides the following method, which is called by the email library code and should not in general be called by application programs:

fold(*, *policy*)

Return a string containing **`linesep`** characters as required to correctly fold the header according to *policy*. A **`cte_type`** of `8bit` will be treated as if it were `7bit`, since headers may not contain arbitrary binary data. If **`utf8`** is `False`, non-ASCII data will be **RFC 2047** [https://datatracker.ietf.org/doc/html/rfc2047.html] encoded.

`BaseHeader` by itself cannot be used to create a header object. It defines a protocol that each specialized header cooperates with in order to produce the header object. Specifically, `BaseHeader` requires that the specialized class provide a **`classmethod()`** named `parse`. This method is called as follows:

```
parse(string, kwds)
```

`kwds` is a dictionary containing one pre-initialized key, `defects`. `defects` is an empty list. The parse method should append any detected defects to this list. On return, the `kwds` dictionary *must* contain values for at least the keys `decoded` and `defects`. `decoded` should be the string value for the header (that is, the header value fully decoded to unicode). The parse method should assume that *string* may contain content-transfer-encoded parts, but should correctly handle all valid unicode characters as well so that it can parse un-encoded header values.

`BaseHeader`’s `__new__` then creates the header instance, and calls its `init` method. The specialized class only needs to provide an `init` method if it wishes to set additional attributes beyond those provided by `BaseHeader` itself. Such an `init` method should look like this:

```
def init(self, /, *args, **kw):
    self._myattr = kw.pop('myattr')
    super().init(*args, **kw)
```

That is, anything extra that the specialized class puts in to the `kwds` dictionary should be removed and handled, and the remaining contents of `kw` (and `args`) passed to the `BaseHeader` `init` method.

class email.headerregistry.UnstructuredHeader

An “unstructured” header is the default type of header in **RFC 5322** [https://datatracker.ietf.org/doc/html/rfc5322.html]. Any header that does not have a specified syntax is treated as unstructured. The classic example of an unstructured header is the *Subject* header.

In **RFC 5322** [https://datatracker.ietf.org/doc/html/rfc5322.html], an unstructured header is a run of arbitrary text in the ASCII character set. **RFC 2047** [https://datatracker.ietf.org/doc/html/rfc2047.html], however, has an **RFC 5322** [https://datatracker.ietf.org/doc/html/rfc5322.html] compatible mechanism for encoding non-ASCII text as ASCII characters within a header value. When a *value* containing encoded words is passed to the constructor, the `UnstructuredHeader` parser converts such encoded words into unicode, following the **RFC 2047** [https://datatracker.ietf.org/doc/html/rfc2047.html] rules for unstructured text. The parser uses heuristics to attempt to decode certain non-compliant encoded words. Defects are registered in such cases, as well as defects for issues such as invalid characters within the encoded words or the non-encoded text.

This header type provides no additional attributes.

class email.headerregistry.DateHeader

RFC 5322 [https://datatracker.ietf.org/doc/html/rfc5322.html] specifies a very specific format for dates within email headers. The `DateHeader` parser recognizes that date format, as well as recognizing a number of variant forms that are sometimes found “in the wild”.

This header type provides the following additional attributes:

datetime

> If the header value can be recognized as a valid date of one form or another, this attribute will contain a **`datetime`** instance representing that date. If the timezone of the input date is specified as `-0000` (indicating it is in UTC but contains no information about the source timezone), then **`datetime`** will be a naive **`datetime`**. If a specific timezone offset is found (including `+0000`), then **`datetime`** will contain an aware `datetime` that uses **`datetime.timezone`** to record the timezone offset.

The `decoded` value of the header is determined by formatting the `datetime` according to the **RFC 5322** [https://datatracker.ietf.org/doc/html/rfc5322.html] rules; that is, it is set to:

```
email.utils.format_datetime(self.datetime)
```

When creating a `DateHeader`, *value* may be **`datetime`** instance. This means, for example, that the following code is valid and does what one would expect:

```
msg['Date'] = datetime(2011, 7, 15, 21)
```

Because this is a naive `datetime` it will be interpreted as a UTC timestamp, and the resulting value will have a timezone of `-0000`. Much more useful is to use the **`localtime()`** function from the **`utils`** module:

```
msg['Date'] = utils.localtime()
```

This example sets the date header to the current time and date using the current timezone offset.

class email.headerregistry.AddressHeader

Address headers are one of the most complex structured header types. The `AddressHeader` class provides a generic interface to any address header.

This header type provides the following additional attributes:

groups

A tuple of **`Group`** objects encoding the addresses and groups found in the header value. Addresses that are not part of a group are represented in this list as single-address `Groups` whose **`display_name`** is `None`.

addresses

A tuple of **`Address`** objects encoding all of the individual addresses from the header value. If the header value contains any groups, the individual addresses from the group are included in the list at the point where the group occurs in the value (that is, the list of addresses is “flattened” into a one dimensional list).

The `decoded` value of the header will have all encoded words decoded to unicode. **`idna`** encoded domain names are also decoded to unicode. The `decoded` value is set by joining the **`str`** value of the elements of the `groups` attribute with `', '`.

A list of **`Address`** and **`Group`** objects in any combination may be used to set the value of an address header. `Group` objects whose `display_name` is `None` will be interpreted as single addresses, which allows an address list to be copied with groups intact by using the list obtained from the `groups` attribute of the source header.

class email.headerregistry.SingleAddressHeader

A subclass of [`AddressHeader`] that adds one additional attribute:

address

The single address encoded by the header value. If the header value actually contains more than one address (which would be a violation of the RFC under the default [`policy`]), accessing this attribute will result in a [`ValueError`].

Many of the above classes also have a `Unique` variant (for example, `UniqueUnstructuredHeader`). The only difference is that in the `Unique` variant, [`max_count`] is set to 1.

class email.headerregistry.MIMEVersionHeader

There is really only one valid value for the *MIME-Version* header, and that is `1.0`. For future proofing, this header class supports other valid version numbers. If a version number has a valid value per **RFC 2045** [https://datatracker.ietf.org/doc/html/rfc2045.html], then the header object will have non-`None` values for the following attributes:

version

The version number as a string, with any whitespace and/or comments removed.

major

The major version number as an integer

minor

The minor version number as an integer

class email.headerregistry.ParameterizedMIMEHeader

MIME headers all start with the prefix ‘Content-’. Each specific header has a certain value, described under the class for that header. Some can also take a list of supplemental parameters, which have a common format. This class serves as a base for all the MIME headers that take parameters.

params

A dictionary mapping parameter names to parameter values.

class email.headerregistry.ContentTypeHeader

A `ParameterizedMIMEHeader` class that handles the *Content-Type* header.

content_type

The content type string, in the form `maintype/subtype`.

maintype

subtype

class email.headerregistry.ContentDispositionHeader

A `ParameterizedMIMEHeader` class that handles the *Content-Disposition* header.

content_disposition

`inline` and `attachment` are the only valid values in common use.

class email.headerregistry.ContentTransferEncodingHeader

Handles the *Content-Transfer-Encoding* header.

cte

Valid values are `7bit`, `8bit`, `base64`, and `quoted-printable`. See **RFC 2045** [https://datatracker.ietf.org/doc/html/rfc2045.html] for more information.

class email.headerregistry.HeaderRegistry(*base_class=BaseHeader*, *default_class=UnstructuredHeader*, *use_default_map=True*)

This is the factory used by `EmailPolicy` by default. `HeaderRegistry` builds the class used to create a header instance dynamically, using *base_class* and a specialized class retrieved from a registry that it holds. When a given header name does not appear in the registry, the class specified by *default_class* is used as the specialized class. When *use_default_map* is `True` (the default), the standard mapping of header names to classes is copied in to the registry during initialization. *base_class* is always the last class in the generated class's `__bases__` list.

The default mappings are:

subject::	UniqueUnstructuredHeader
date::	UniqueDateHeader
resent-date::	DateHeader
orig-date::	UniqueDateHeader
sender::	UniqueSingleAddressHeader
resent-sender::	SingleAddressHeader
to::	UniqueAddressHeader
resent-to::	AddressHeader
cc::	UniqueAddressHeader
resent-cc::	AddressHeader
bcc::	UniqueAddressHeader
resent-bcc::	AddressHeader
from::	UniqueAddressHeader

resent-from:: AddressHeader
reply-to:: UniqueAddressHeader
mime-version:: MIMEVersionHeader
content-type:: ContentTypeHeader
content-disposition:: ContentDispositionHeader
content-transfer-encoding:: ContentTransferEncodingHeader
message-id:: MessageIDHeader

`HeaderRegistry` has the following methods:

map_to_type(*self*, *name*, *cls*)

name is the name of the header to be mapped. It will be converted to lower case in the registry. *cls* is the specialized class to be used, along with *base_class*, to create the class used to instantiate headers that match *name*.

__getitem__(*name*)

Construct and return a class to handle creating a *name* header.

__call__(*name*, *value*)

Retrieves the specialized header associated with *name* from the registry (using *default_class* if *name* does not appear in the registry) and composes it with *base_class* to produce a class, calls the constructed class’s constructor, passing it the same argument list, and finally returns the class instance created thereby.

The following classes are the classes used to represent data parsed from structured headers and can, in general, be used by an application program to construct structured values to assign to specific headers.

class email.headerregistry.Address(*display_name=''*, *username=''*, *domain=''*, *addr_spec=None*)

The class used to represent an email address. The general form of an address is:

```
[display_name] <username@domain>
```

or:

```
username@domain
```

where each part must conform to specific syntax rules spelled out in **RFC 5322** [https://datatracker.ietf.org/doc/html/rfc5322.html].

As a convenience *addr_spec* can be specified instead of *username* and *domain*, in which case *username* and *domain* will be parsed from the *addr_spec*. An *addr_spec* must be a properly RFC quoted string; if it is not `Address` will raise an error. Unicode characters are allowed and will be property encoded when serialized. However, per the RFCs, unicode is *not* allowed in the username portion of the address.

display_name

The display name portion of the address, if any, with all quoting removed. If the address does not have a display name, this attribute will be an empty string.

username

The `username` portion of the address, with all quoting removed.

domain

The `domain` portion of the address.

addr_spec

The `username@domain` portion of the address, correctly quoted for use as a bare address (the second form shown above). This attribute is not mutable.

__str__()

The `str` value of the object is the address quoted according to **RFC 5322** [https://datatracker.ietf.org/doc/html/rfc5322.html] rules, but with no Content Transfer Encoding of any non-ASCII characters.

To support SMTP (**RFC 5321** [https://datatracker.ietf.org/doc/html/rfc5321.html]), `Address` handles one special case: if `username` and `domain` are both the empty string (or `None`), then the string value of the `Address` is `<>`.

class email.headerregistry.Group(*display_name=None*, *addresses=None*)

The class used to represent an address group. The general form of an address group is:

```
display_name: [address-list];
```

As a convenience for processing lists of addresses that consist of a mixture of groups and single addresses, a `Group` may also be used to represent single addresses that are not part of a group by setting *display_name* to `None` and providing a list of the single address as *addresses*.

display_name

The `display_name` of the group. If it is `None` and there is exactly one `Address` in `addresses`, then the `Group` represents a single address that is not in a group.

addresses

A possibly empty tuple of **`Address`** objects representing the addresses in the group.

__str__()

The `str` value of a `Group` is formatted according to **RFC 5322** [https://datatracker.ietf.org/doc/html/rfc5322.html], but with no Content Transfer Encoding of any non-ASCII characters. If `display_name` is none and there is a single `Address` in the `addresses` list, the `str` value will be the same as the `str` of that single `Address`.

Footnotes

[1]

Originally added in 3.3 as a provisional module

`email.contentmanager`: Managing MIME Content

Source code: Lib/email/contentmanager.py [https://github.com/python/cpython/tree/3.14/Lib/email/contentmanager.py]

Added in version 3.6: [[1]]

class email.contentmanager.ContentManager

Base class for content managers. Provides the standard registry mechanisms to register converters between MIME content and other representations, as well as the `get_content` and `set_content` dispatch methods.

get_content(*msg*, **args*, ***kw*)

Look up a handler function based on the `mimetype` of *msg* (see next paragraph), call it, passing through all arguments, and return the result of the call. The expectation is that the handler will extract the payload from *msg* and return an object that encodes information about the extracted data.

To find the handler, look for the following keys in the registry, stopping with the first one found:

- the string representing the full MIME type (`maintype/subtype`)
- the string representing the `maintype`
- the empty string

If none of these keys produce a handler, raise a **`KeyError`** for the full MIME type.

set_content(*msg*, *obj*, **args*, ***kw*)

If the `maintype` is `multipart`, raise a **`TypeError`**; otherwise look up a handler function based on the type of *obj* (see next paragraph), call **`clear_content()`** on the *msg*, and call the handler function, passing through all arguments. The expectation is that the handler will transform and store *obj* into *msg*, possibly making other changes to *msg* as well, such as adding various MIME headers to encode information needed to interpret the stored data.

To find the handler, obtain the type of *obj* (`typ = type(obj)`), and look for the following keys in the registry, stopping with the first one found:

- the type itself (`typ`)
- the type’s fully qualified name (`typ.__module__ + '.' + typ.__qualname__`).
- the type’s **`qualname`** (`typ.__qualname__`)
- the type’s **`name`** (`typ.__name__`).

If none of the above match, repeat all of the checks above for each of the types in the MRO (**`typ.__mro__`**). Finally, if no other key yields a handler, check for a handler for the key `None`. If there is no handler for `None`, raise a **`KeyError`** for the fully qualified name of the type.

Also add a *MIME-Version* header if one is not present (see also **`MIMEPart`**).

add_get_handler(*key*, *handler*)

Record the function *handler* as the handler for *key*. For the possible values of *key*, see **`get_content()`**.

add_set_handler(*typekey*, *handler*)

Record *handler* as the function to call when an object of a type matching *typekey* is passed to `set_content()`. For the possible values of *typekey*, see `set_content()`.

Content Manager Instances

Currently the email package provides only one concrete content manager, `raw_data_manager`, although more may be added in the future. `raw_data_manager` is the `content_manager` provided by `EmailPolicy` and its derivatives.

email.contentmanager.raw_data_manager

This content manager provides only a minimum interface beyond that provided by `Message` itself: it deals only with text, raw byte strings, and `Message` objects. Nevertheless, it provides significant advantages compared to the base API: `get_content` on a text part will return a unicode string without the application needing to manually decode it, `set_content` provides a rich set of options for controlling the headers added to a part and controlling the content transfer encoding, and it enables the use of the various `add_` methods, thereby simplifying the creation of multipart messages.

email.contentmanager.get_content(*msg*, *errors='replace'*)

Return the payload of the part as either a string (for `text` parts), an `EmailMessage` object (for `message/rfc822` parts), or a `bytes` object (for all other non-multipart types). Raise a `KeyError` if called on a `multipart`. If the part is a `text` part and *errors* is specified, use it as the error handler when decoding the payload to unicode. The default error handler is `replace`.

email.contentmanager.set_content(*msg*, *<'str'>*, *subtype="plain"*, *charset='utf-8'*, *cte=None*, *disposition=None*, *filename=None*, *cid=None*, *params=None*, *headers=None*)
email.contentmanager.set_content(*msg*, *<'bytes'>*, *maintype*, *subtype*, *cte="base64"*, *disposition=None*, *filename=None*, *cid=None*, *params=None*, *headers=None*)
email.contentmanager.set_content(*msg*, *<'EmailMessage'>*, *cte=None*, *disposition=None*, *filename=None*, *cid=None*, *params=None*, *headers=None*)

Add headers and payload to *msg*:

Add a *Content-Type* header with a `maintype/subtype` value.

- For `str`, set the MIME `maintype` to `text`, and set the subtype to *subtype* if it is specified, or `plain` if it is not.
- For `bytes`, use the specified *maintype* and *subtype*, or raise a `TypeError` if they are not specified.
- For `EmailMessage` objects, set the maintype to `message`, and set the subtype to *subtype* if it is specified or `rfc822` if it is not. If *subtype* is `partial`, raise an error (`bytes` objects must be used to construct `message/partial` parts).

If *charset* is provided (which is valid only for `str`), encode the string to bytes using the specified character set. The default is `utf-8`. If the specified *charset* is a known alias for a standard MIME charset name, use the standard charset instead.

If *cte* is set, encode the payload using the specified content transfer encoding, and set the *Content-*

Transfer-Encoding header to that value. Possible values for *cte* are `quoted-printable`, `base64`, `7bit`, `8bit`, and `binary`. If the input cannot be encoded in the specified encoding (for example, specifying a *cte* of `7bit` for an input that contains non-ASCII values), raise a `ValueError`.

- For `str` objects, if *cte* is not set use heuristics to determine the most compact encoding. Prior to encoding, `str.splitlines()` is used to normalize all line boundaries, ensuring that each line of the payload is terminated by the current policy's `linesep` property (even if the original string did not end with one).
- For `bytes` objects, *cte* is taken to be base64 if not set, and the aforementioned newline translation is not performed.
- For `EmailMessage`, per **RFC 2046** [https://datatracker.ietf.org/doc/html/rfc2046.html], raise an error if a *cte* of `quoted-printable` or `base64` is requested for *subtype* `rfc822`, and for any *cte* other than `7bit` for *subtype* `external-body`. For `message/rfc822`, use `8bit` if *cte* is not specified. For all other values of *subtype*, use `7bit`.

Note

A *cte* of `binary` does not actually work correctly yet. The `EmailMessage` object as modified by `set_content` is correct, but `BytesGenerator` does not serialize it correctly.

If *disposition* is set, use it as the value of the *Content-Disposition* header. If not specified, and *filename* is specified, add the header with the value `attachment`. If *disposition* is not specified and *filename* is also not specified, do not add the header. The only valid values for *disposition* are `attachment` and `inline`.

If *filename* is specified, use it as the value of the `filename` parameter of the *Content-Disposition* header.

If *cid* is specified, add a *Content-ID* header with *cid* as its value.

If *params* is specified, iterate its `items` method and use the resulting `(key, value)` pairs to set additional parameters on the *Content-Type* header.

If *headers* is specified and is a list of strings of the form `headername: headervalue` or a list of `header` objects (distinguished from strings by having a `name` attribute), add the headers to *msg*.

Footnotes

[1]

Originally added in 3.4 as a provisional module

`email`: Examples

Here are a few examples of how to use the `email` package to read, write, and send simple email messages, as well as more complex MIME messages.

First, let’s see how to create and send a simple text message (both the text content and the addresses may contain unicode characters):

```
# Import smtplib for the actual sending function
import smtplib

# Import the email modules we'll need
from email.message import EmailMessage

# Open the plain text file whose name is in textfile for
reading.
with open(textfile) as fp:
    # Create a text/plain message
    msg = EmailMessage()
    msg.set_content(fp.read())

# me == the sender's email address
# you == the recipient's email address
msg['Subject'] = f'The contents of {textfile}'
msg['From'] = me
msg['To'] = you

# Send the message via our own SMTP server.
s = smtplib.SMTP('localhost')
s.send_message(msg)
s.quit()
```

Parsing **RFC 822** [https://datatracker.ietf.org/doc/html/rfc822.html] headers can easily be done by the using the classes from the `parser` module:

```
# Import the email modules we'll need
#from email.parser import BytesParser
from email.parser import Parser
from email.policy import default

# If the e-mail headers are in a file, uncomment these two
lines:
# with open(messagefile, 'rb') as fp:
#     headers = BytesParser(policy=default).parse(fp)

#  Or for parsing headers in a string (this is an uncommon
operation), use:
headers = Parser(policy=default).parsestr(
        'From: Foo Bar <user@example.com>\n'
        'To: <someone_else@example.com>\n'
        'Subject: Test message\n'
        '\n'
        'Body would go here\n')

#  Now the header items can be accessed as a dictionary:
print('To: {}'.format(headers['to']))
print('From: {}'.format(headers['from']))
print('Subject: {}'.format(headers['subject']))

# You can also access the parts of the addresses:
print('Recipient username:
{}'.format(headers['to'].addresses[0].username))
print('Sender name:
{}'.format(headers['from'].addresses[0].display_name))
```

Here’s an example of how to send a MIME message containing a bunch of family pictures that may be residing in a directory:

```
# Import smtplib for the actual sending function.
import smtplib

# Here are the email package modules we'll need.
from email.message import EmailMessage

# Create the container email message.
msg = EmailMessage()
```

```
msg['Subject'] = 'Our family reunion'
# me == the sender's email address
# family = the list of all recipients' email addresses
msg['From'] = me
msg['To'] = ', '.join(family)
msg.preamble = 'You will not see this in a MIME-aware mail
reader.\n'

# Open the files in binary mode.  You can also omit the subtype
# if you want MIMEImage to guess it.
for file in pngfiles:
    with open(file, 'rb') as fp:
        img_data = fp.read()
    msg.add_attachment(img_data, maintype='image',
                                 subtype='png')

# Send the email via our own SMTP server.
with smtplib.SMTP('localhost') as s:
    s.send_message(msg)
```

Here’s an example of how to send the entire contents of a directory as an email message: [[1]]

```
#!/usr/bin/env python3

"""Send the contents of a directory as a MIME message."""

import os
import smtplib
# For guessing MIME type based on file name extension
import mimetypes

from argparse import ArgumentParser

from email.message import EmailMessage
from email.policy import SMTP


def main():
    parser = ArgumentParser(description="""\
Send the contents of a directory as a MIME message.
Unless the -o option is given, the email is sent by forwarding
to your local
SMTP server, which then does the normal delivery process.  Your
local machine
must be running an SMTP server.
""")
    parser.add_argument('-d', '--directory',
                        help="""Mail the contents of the
specified directory,
                        otherwise use the current directory.
Only the regular
                        files in the directory are sent, and we
don't recurse to
                        subdirectories.""")
    parser.add_argument('-o', '--output',
                        metavar='FILE',
                        help="""Print the composed message to
FILE instead of
                        sending the message to the SMTP
server.""")
    parser.add_argument('-s', '--sender', required=True,
                        help='The value of the From: header
(required)')
    parser.add_argument('-r', '--recipient', required=True,
                        action='append', metavar='RECIPIENT',
                        default=[], dest='recipients',
                        help='A To: header value (at least one
required)')
    args = parser.parse_args()
    directory = args.directory
    if not directory:
        directory = '.'
    # Create the message
    msg = EmailMessage()
    msg['Subject'] = f'Contents of directory
{os.path.abspath(directory)}'
    msg['To'] = ', '.join(args.recipients)
    msg['From'] = args.sender
    msg.preamble = 'You will not see this in a MIME-aware mail
reader.\n'

    for filename in os.listdir(directory):
        path = os.path.join(directory, filename)
        if not os.path.isfile(path):
            continue
        # Guess the content type based on the file's extension.
Encoding
        # will be ignored, although we should check for simple
things like
        # gzip'd or compressed files.
        ctype, encoding = mimetypes.guess_file_type(path)
        if ctype is None or encoding is not None:
            # No guess could be made, or the file is encoded
(compressed), so
            # use a generic bag-of-bits type.
            ctype = 'application/octet-stream'
        maintype, subtype = ctype.split('/', 1)
        with open(path, 'rb') as fp:
            msg.add_attachment(fp.read(),
                               maintype=maintype,
                               subtype=subtype,
                               filename=filename)
    # Now send or store the message
    if args.output:
        with open(args.output, 'wb') as fp:
            fp.write(msg.as_bytes(policy=SMTP))
    else:
        with smtplib.SMTP('localhost') as s:
            s.send_message(msg)


if __name__ == '__main__':
    main()
```

Here’s an example of how to unpack a MIME message like the one above, into a directory of files:

```
#!/usr/bin/env python3

"""Unpack a MIME message into a directory of files."""

import os
import email
import mimetypes

from email.policy import default
from argparse import ArgumentParser


def main():
    parser = ArgumentParser(description="""\
Unpack a MIME message into a directory of files.
""")
    parser.add_argument('-d', '--directory', required=True,
                        help="""Unpack the MIME message into
the named
                        directory, which will be created if it
doesn't already
                        exist.""")
    parser.add_argument('msgfile')
    args = parser.parse_args()

    with open(args.msgfile, 'rb') as fp:
        msg = email.message_from_binary_file(fp,
policy=default)

    try:
        os.mkdir(args.directory)
    except FileExistsError:
        pass

    counter = 1
    for part in msg.walk():
        # multipart/* are just containers
        if part.get_content_maintype() == 'multipart':
            continue
        # Applications should really sanitize the given
filename so that an
        # email message can't be used to overwrite important
files
        filename = part.get_filename()
        if not filename:
            ext =
mimetypes.guess_extension(part.get_content_type())
            if not ext:
                # Use a generic bag-of-bits extension
                ext = '.bin'
            filename = f'part-{counter:03d}{ext}'
        counter += 1
        with open(os.path.join(args.directory, filename), 'wb')
as fp:
            fp.write(part.get_payload(decode=True))


if __name__ == '__main__':
    main()
```

Here’s an example of how to create an HTML message with an alternative plain text version. To make things a bit more interesting, we include a related image in the html part, and we save a copy of what we are going to send to disk, as well as sending it.

```
#!/usr/bin/env python3

import smtplib

from email.message import EmailMessage
from email.headerregistry import Address
from email.utils import make_msgid

# Create the base text message.
msg = EmailMessage()
msg['Subject'] = "Pourquoi pas des asperges pour ce midi ?"
msg['From'] = Address("Pepé Le Pew", "pepe", "example.com")
msg['To'] = (Address("Penelope Pussycat", "penelope",
"example.com"),
             Address("Fabrette Pussycat", "fabrette",
"example.com"))
msg.set_content("""\
Salut!

Cette recette [1] sera sûrement un très bon repas.

[1] http://www.yummly.com/recipe/Roasted-Asparagus-Epicurious-
203718

--Pepé
""")
```

```
# Add the html version.  This converts the message into a
multipart/alternative
# container, with the original text message as the first part
and the new html
# message as the second part.
asparagus_cid = make_msgid()
msg.add_alternative("""\
<html>
  <head></head>
  <body>
    <p>Salut!</p>
    <p>Cette
        <a href="http://www.yummly.com/recipe/Roasted-
Asparagus-Epicurious-203718">
            recette
        </a> sera sûrement un très bon repas.
    </p>
    <img src="cid:{asparagus_cid}">
  </body>
</html>
""".format(asparagus_cid=asparagus_cid[1:-1]), subtype='html')
# note that we needed to peel the <> off the msgid for use in
the html.

# Now add the related image to the html part.
with open("roasted-asparagus.jpg", 'rb') as img:
    msg.get_payload()[1].add_related(img.read(), 'image',
'jpeg',
                                     cid=asparagus_cid)

# Make a local copy of what we are going to send.
with open('outgoing.msg', 'wb') as f:
    f.write(bytes(msg))

# Send the message via local SMTP server.
with smtplib.SMTP('localhost') as s:
    s.send_message(msg)
```

If we were sent the message from the last example, here is one way we could process it:

```
import os
import sys
import tempfile
import mimetypes
import webbrowser

# Import the email modules we'll need
from email import policy
from email.parser import BytesParser


def magic_html_parser(html_text, partfiles):
    """Return safety-sanitized html linked to partfiles.

    Rewrite the href="cid:...." attributes to point to the
filenames in partfiles.
    Though not trivial, this should be possible using
html.parser.
    """
    raise NotImplementedError("Add the magic needed")


# In a real program you'd get the filename from the arguments.
with open('outgoing.msg', 'rb') as fp:
    msg = BytesParser(policy=policy.default).parse(fp)

# Now the header items can be accessed as a dictionary, and any
non-ASCII will
# be converted to unicode:
print('To:', msg['to'])
print('From:', msg['from'])
print('Subject:', msg['subject'])

# If we want to print a preview of the message content, we can
extract whatever
# the least formatted payload is and print the first three
lines.  Of course,
# if the message has no plain text part printing the first
three lines of html
# is probably useless, but this is just a conceptual example.
simplest = msg.get_body(preferencelist=('plain', 'html'))
print()
print(''.join(simplest.get_content().splitlines(keepends=True)
[:3]))

ans = input("View full message?")
if ans.lower()[0] == 'n':
    sys.exit()

# We can extract the richest alternative in order to display
it:
richest = msg.get_body()
partfiles = {}
if richest['content-type'].maintype == 'text':
    if richest['content-type'].subtype == 'plain':
        for line in richest.get_content().splitlines():
            print(line)
        sys.exit()
    elif richest['content-type'].subtype == 'html':
        body = richest
    else:
        print("Don't know how to display
{}".format(richest.get_content_type()))
        sys.exit()
elif richest['content-type'].content_type ==
'multipart/related':
    body = richest.get_body(preferencelist=('html'))
    for part in richest.iter_attachments():
        fn = part.get_filename()
        if fn:
            extension = os.path.splitext(part.get_filename())
[1]
        else:
            extension =
mimetypes.guess_extension(part.get_content_type())
        with tempfile.NamedTemporaryFile(suffix=extension,
delete=False) as f:
            f.write(part.get_content())
            # again strip the <> to go from email form of cid
to html form.
            partfiles[part['content-id'][1:-1]] = f.name
else:
    print("Don't know how to display
{}".format(richest.get_content_type()))
    sys.exit()
with tempfile.NamedTemporaryFile(mode='w', delete=False) as f:
    f.write(magic_html_parser(body.get_content(), partfiles))
webbrowser.open(f.name)
os.remove(f.name)
for fn in partfiles.values():
    os.remove(fn)

# Of course, there are lots of email messages that could break
this simple
# minded program, but it will handle the most common ones.
```

Up to the prompt, the output from the above is:

```
To: Penelope Pussycat <penelope@example.com>, Fabrette Pussycat
<fabrette@example.com>
From: Pepé Le Pew <pepe@example.com>
Subject: Pourquoi pas des asperges pour ce midi ?

Salut!

Cette recette [1] sera sûrement un très bon repas.
```

Footnotes

[1]

Thanks to Matthew Dixon Cowles for the original inspiration and examples.

`email.message.Message`: Representing an email message using the `compat32` API

The `Message` class is very similar to the `EmailMessage` class, without the methods added by that class, and with the default behavior of certain other methods being slightly different. We also document here some methods that, while supported by the `EmailMessage` class, are not recommended unless you are dealing with legacy code.

The philosophy and structure of the two classes is otherwise the same.

This document describes the behavior under the default (for `Message`) policy `Compat32`. If you are going to use another policy, you should be using the `EmailMessage` class instead.

An email message consists of *headers* and a *payload*. Headers must be **RFC 5322** [https://datatracker.ietf.org/doc/html/rfc5322.html] style names and values, where the field name and value are separated by a colon. The colon is not part of either the field name or the field value. The payload may be a simple text message, or a binary object, or a structured sequence of sub-messages each with their own set of headers and their own payload. The latter type of payload is indicated by the message having a MIME type such as *multipart/** or *message/rfc822*.

The conceptual model provided by a `Message` object is that of an ordered dictionary of headers with additional methods for accessing both specialized information from the headers, for accessing the payload, for generating a serialized version of the message, and for recursively walking over the object tree. Note that duplicate headers are supported but special methods must be used to access them.

The `Message` pseudo-dictionary is indexed by the header names, which must be ASCII values. The values of the dictionary are strings that are supposed to contain only ASCII characters; there is some special handling for non-ASCII input, but it doesn’t always produce the correct results. Headers are stored and returned in case-preserving form, but field names are matched case-insensitively. There may also be a single envelope header, also known as the *Unix-From* header or the `From_` header. The *payload* is either a string or bytes, in the case of simple message objects, or a list of `Message` objects, for MIME container documents (e.g. *multipart/** and *message/rfc822*).

Here are the methods of the `Message` class:

class email.message.Message(*policy=compat32*)

If *policy* is specified (it must be an instance of a `policy` class) use the rules it specifies to update and serialize the representation of the message. If *policy* is not set, use the `compat32` policy, which maintains backward compatibility with the Python 3.2 version of the email package. For more information see the `policy` documentation.

Changed in version 3.3: The *policy* keyword argument was added.

as_string(*unixfrom=False*, *maxheaderlen=0*, *policy=None*)

Return the entire message flattened as a string. When optional *unixfrom* is true, the envelope header is included in the returned string. *unixfrom* defaults to `False`. For backward compatibility reasons, *maxheaderlen* defaults to `0`, so if you want a different value you must override it explicitly (the value specified for *max_line_length* in the policy will be ignored by this method). The *policy* argument may be used to override the default policy obtained from the message instance. This can be used to control some of the formatting produced by the method, since the specified *policy* will be passed to the `Generator`.

Flattening the message may trigger changes to the **`Message`** if defaults need to be filled in to complete the transformation to a string (for example, MIME boundaries may be generated or modified).

Note that this method is provided as a convenience and may not always format the message the way you want. For example, by default it does not do the mangling of lines that begin with `From` that is required by the Unix mbox format. For more flexibility, instantiate a **`Generator`** instance and use its **`flatten()`** method directly. For example:

```
from io import StringIO
from email.generator import Generator
fp = StringIO()
g = Generator(fp, mangle_from_=True, maxheaderlen=60)
g.flatten(msg)
text = fp.getvalue()
```

If the message object contains binary data that is not encoded according to RFC standards, the non-compliant data will be replaced by unicode “unknown character” code points. (See also `as_bytes()` and `BytesGenerator`.)

Changed in version 3.4: the *policy* keyword argument was added.

__str__()

Equivalent to `as_string()`. Allows `str(msg)` to produce a string containing the formatted message.

as_bytes(*unixfrom=False*, *policy=None*)

Return the entire message flattened as a bytes object. When optional *unixfrom* is true, the envelope header is included in the returned string. *unixfrom* defaults to `False`. The *policy* argument may be used to override the default policy obtained from the message instance. This can be used to control some of the formatting produced by the method, since the specified *policy* will be passed to the `BytesGenerator`.

Flattening the message may trigger changes to the `Message` if defaults need to be filled in to complete the transformation to a string (for example, MIME boundaries may be generated or modified).

Note that this method is provided as a convenience and may not always format the message the way you want. For example, by default it does not do the mangling of lines that begin with `From` that is required by the Unix mbox format. For more flexibility, instantiate a `BytesGenerator` instance and use its `flatten()` method directly. For example:

```
from io import BytesIO
from email.generator import BytesGenerator
fp = BytesIO()
```

```
g = BytesGenerator(fp, mangle_from_=True,
maxheaderlen=60)
g.flatten(msg)
text = fp.getvalue()
```

Added in version 3.4.

__bytes__()

Equivalent to `as_bytes()`. Allows `bytes(msg)` to produce a bytes object containing the formatted message.

Added in version 3.4.

is_multipart()

Return `True` if the message’s payload is a list of sub-`Message` objects, otherwise return `False`. When `is_multipart()` returns `False`, the payload should be a string object (which might be a CTE encoded binary payload). (Note that `is_multipart()` returning `True` does not necessarily mean that “msg.get_content_maintype() == ‘multipart’” will return the `True`. For example, `is_multipart` will return `True` when the `Message` is of type `message/rfc822`.)

set_unixfrom(*unixfrom*)

Set the message’s envelope header to *unixfrom*, which should be a string.

get_unixfrom()

Return the message’s envelope header. Defaults to `None` if the envelope header was never set.

attach(*payload*)

Add the given *payload* to the current payload, which must be `None` or a list of `Message` objects before the call. After the call, the payload will always be a list of

`Message` objects. If you want to set the payload to a scalar object (e.g. a string), use `set_payload()` instead.

This is a legacy method. On the `EmailMessage` class its functionality is replaced by `set_content()` and the related `make` and `add` methods.

get_payload(*i=None*, *decode=False*)

Return the current payload, which will be a list of `Message` objects when `is_multipart()` is `True`, or a string when `is_multipart()` is `False`. If the payload is a list and you mutate the list object, you modify the message's payload in place.

With optional argument *i*, `get_payload()` will return the *i*-th element of the payload, counting from zero, if `is_multipart()` is `True`. An `IndexError` will be raised if *i* is less than 0 or greater than or equal to the number of items in the payload. If the payload is a string (i.e. `is_multipart()` is `False`) and *i* is given, a `TypeError` is raised.

Optional *decode* is a flag indicating whether the payload should be decoded or not, according to the *Content-Transfer-Encoding* header. When `True` and the message is not a multipart, the payload will be decoded if this header's value is `quoted-printable` or `base64`. If some other encoding is used, or *Content-Transfer-Encoding* header is missing, the payload is returned as-is (undecoded). In all cases the returned value is binary data. If the message is a multipart and the *decode* flag is `True`, then `None` is returned. If the payload is base64 and it was not perfectly formed (missing padding, characters outside the base64 alphabet), then an appropriate defect will be added to the message's defect property

(`InvalidBase64PaddingDefect` or `InvalidBase64CharactersDefect`, respectively).

When *decode* is `False` (the default) the body is returned as a string without decoding the *Content-Transfer-Encoding*. However, for a *Content-Transfer-Encoding* of 8bit, an attempt is made to decode the original bytes using the `charset` specified by the *Content-Type* header, using the `replace` error handler. If no `charset` is specified, or if the `charset` given is not recognized by the email package, the body is decoded using the default ASCII charset.

This is a legacy method. On the `EmailMessage` class its functionality is replaced by `get_content()` and `iter_parts()`.

set_payload(*payload*, *charset=None*)

Set the entire message object's payload to *payload*. It is the client's responsibility to ensure the payload invariants. Optional *charset* sets the message's default character set; see `set_charset()` for details.

This is a legacy method. On the `EmailMessage` class its functionality is replaced by `set_content()`.

set_charset(*charset*)

Set the character set of the payload to *charset*, which can either be a `Charset` instance (see `email.charset`), a string naming a character set, or `None`. If it is a string, it will be converted to a `Charset` instance. If *charset* is `None`, the `charset` parameter will be removed from the *Content-Type* header (the message will not be otherwise modified). Anything else will generate a `TypeError`.

If there is no existing *MIME-Version* header one will be added. If there is no existing *Content-Type* header, one will be added with a value of *text/plain*. Whether the *Content-Type* header already exists or not, its `charset` parameter will be set to *charset.output_charset*. If *charset.input_charset* and *charset.output_charset* differ, the payload will be re-encoded to the *output_charset*. If there is no existing *Content-Transfer-Encoding* header, then the payload will be transfer-encoded, if needed, using the specified `Charset`, and a header with the appropriate value will be added. If a *Content-Transfer-Encoding* header already exists, the payload is assumed to already be correctly encoded using that *Content-Transfer-Encoding* and is not modified.

This is a legacy method. On the `EmailMessage` class its functionality is replaced by the *charset* parameter of the `email.message.EmailMessage.set_content()` method.

get_charset()

Return the `Charset` instance associated with the message’s payload.

This is a legacy method. On the `EmailMessage` class it always returns `None`.

The following methods implement a mapping-like interface for accessing the message’s **RFC 2822** [https://datatracker.ietf.org/doc/html/rfc2822.html] headers. Note that there are some semantic differences between these methods and a normal mapping (i.e. dictionary) interface. For example, in a dictionary there are no duplicate keys, but here there may be duplicate message headers. Also, in dictionaries there is no guaranteed order to the keys returned by `keys()`, but in a `Message` object, headers are always returned in the order they appeared in the original message, or were added to the message later. Any header deleted and then re-added are always appended to the end of the header list.

These semantic differences are intentional and are biased toward maximal convenience.

Note that in all cases, any envelope header present in the message is not included in the mapping interface.

In a model generated from bytes, any header values that (in contravention of the RFCs) contain non-ASCII bytes will, when retrieved through this interface, be represented as **`Header`** objects with a charset of `unknown-8bit`.

__len__()

Return the total number of headers, including duplicates.

__contains__(*name*)

Return `True` if the message object has a field named *name*. Matching is done case-insensitively and *name* should not include the trailing colon. Used for the `in` operator, e.g.:

```
if 'message-id' in myMessage:
   print('Message-ID:', myMessage['message-id'])
```

__getitem__(*name*)

Return the value of the named header field. *name* should not include the colon field separator. If the header is missing, `None` is returned; a **`KeyError`** is never raised.

Note that if the named field appears more than once in the message’s headers, exactly which of those field values will be returned is undefined. Use the `get_all()` method to get the values of all the extant named headers.

__setitem__(*name*, *val*)

Add a header to the message with field name *name* and value *val*. The field is appended to the end of the message’s existing fields.

Note that this does *not* overwrite or delete any existing header with the same name. If you want to ensure that the new header is the only one present in the message with field name *name*, delete the field first, e.g.:

```
del msg['subject']
msg['subject'] = 'Python roolz!'
```

__delitem__(*name*)

Delete all occurrences of the field with name *name* from the message’s headers. No exception is raised if the named field isn’t present in the headers.

keys()

Return a list of all the message’s header field names.

values()

Return a list of all the message’s field values.

items()

Return a list of 2-tuples containing all the message’s field headers and values.

get(*name*, *failobj=None*)

Return the value of the named header field. This is identical to `__getitem__()` except that optional *failobj* is returned if the named header is missing (defaults to `None`).

Here are some additional useful methods:

get_all(*name*, *failobj=None*)

Return a list of all the values for the field named *name*. If there are no such named headers in the message, *failobj* is returned (defaults to `None`).

add_header(*_name*, *_value*, ***_params*)

Extended header setting. This method is similar to `__setitem__()` except that additional header parameters can be provided as keyword arguments. *_name* is the header field to add and *_value* is the *primary* value for the header.

For each item in the keyword argument dictionary *_params*, the key is taken as the parameter name, with underscores converted to dashes (since dashes are illegal in Python identifiers). Normally, the parameter will be added as `key="value"` unless the value is `None`, in which case only the key will be added. If the value contains non-ASCII characters, it can be specified as a three tuple in the format `(CHARSET, LANGUAGE, VALUE)`, where `CHARSET` is a string naming the charset to be used to encode the value, `LANGUAGE` can usually be set to `None` or the empty string (see **RFC 2231** [https://datatracker.ietf.org/doc/html/rfc2231.html] for other possibilities), and `VALUE` is the string value containing non-ASCII code points. If a three tuple is not passed and the value contains non-ASCII characters, it is automatically encoded in **RFC 2231** [https://datatracker.ietf.org/doc/html/rfc2231.html] format using a `CHARSET` of `utf-8` and a `LANGUAGE` of `None`.

Here’s an example:

```
msg.add_header('Content-Disposition', 'attachment',
filename='bud.gif')
```

This will add a header that looks like

```
Content-Disposition: attachment; filename="bud.gif"
```

An example with non-ASCII characters:

```
msg.add_header('Content-Disposition', 'attachment',
               filename=('iso-8859-1', '',
'Fußballer.ppt'))
```

Which produces

```
Content-Disposition: attachment; filename*="iso-8859-
1''Fu%DFballer.ppt"
```

replace_header(*_name*, *_value*)

Replace a header. Replace the first header found in the message that matches *_name*, retaining header order and field name case. If no matching header was found, a `KeyError` is raised.

get_content_type()

Return the message’s content type. The returned string is coerced to lower case of the form *maintype/subtype*. If there was no *Content-Type* header in the message the default type as given by `get_default_type()` will be returned. Since according to **RFC 2045** [https://datatracker.ietf.org/doc/html/rfc2045.html], messages always have a default type, `get_content_type()` will always return a value.

RFC 2045 [https://datatracker.ietf.org/doc/html/rfc2045.html] defines a message’s default type to be *text/plain* unless it appears inside a *multipart/digest* container, in which case it would be *message/rfc822*. If the *Content-Type* header has an invalid type specification, **RFC 2045** [https://datatracker.ietf.org/doc/html/rfc2045.html] mandates that the default type be *text/plain*.

get_content_maintype()

Return the message’s main content type. This is the *maintype* part of the string returned by `get_content_type()`.

get_content_subtype()

Return the message’s sub-content type. This is the *subtype* part of the string returned by `get_content_type()`.

get_default_type()

Return the default content type. Most messages have a default content type of *text/plain*, except for messages that are subparts of *multipart/digest* containers. Such subparts have a default content type of *message/rfc822*.

set_default_type(*ctype*)

Set the default content type. *ctype* should either be *text/plain* or *message/rfc822*, although this is not enforced. The default content type is not stored in the *Content-Type* header.

get_params(*failobj=None*, *header='content-type'*, *unquote=True*)

Return the message’s *Content-Type* parameters, as a list. The elements of the returned list are 2-tuples of key/value pairs, as split on the `'='` sign. The left hand side of the `'='` is the key, while the right hand side is the value. If there is no `'='` sign in the parameter the value is the empty string, otherwise the value is as described in `get_param()` and is unquoted if optional *unquote* is `True` (the default).

Optional *failobj* is the object to return if there is no *Content-Type* header. Optional *header* is the header to search instead of *Content-Type*.

This is a legacy method. On the **`EmailMessage`** class its functionality is replaced by the *params* property of the individual header objects returned by the header access methods.

get_param(*param*, *failobj=None*, *header='content-type'*, *unquote=True*)

Return the value of the *Content-Type* header's parameter *param* as a string. If the message has no *Content-Type* header or if there is no such parameter, then *failobj* is returned (defaults to `None`).

Optional *header* if given, specifies the message header to use instead of *Content-Type*.

Parameter keys are always compared case insensitively. The return value can either be a string, or a 3-tuple if the parameter was **RFC 2231** [https://datatracker.ietf.org/doc/html/rfc2231.html] encoded. When it's a 3-tuple, the elements of the value are of the form `(CHARSET, LANGUAGE, VALUE)`. Note that both `CHARSET` and `LANGUAGE` can be `None`, in which case you should consider `VALUE` to be encoded in the `us-ascii` charset. You can usually ignore `LANGUAGE`.

If your application doesn’t care whether the parameter was encoded as in **RFC 2231** [https://datatracker.ietf.org/doc/html/rfc2231.html], you can collapse the parameter value by calling `email.utils.collapse_rfc2231_value()`, passing in the return value from `get_param()`. This will return a suitably decoded Unicode string when the value is a tuple, or the original string unquoted if it isn’t. For example:

```
rawparam = msg.get_param('foo')
param = email.utils.collapse_rfc2231_value(rawparam)
```

In any case, the parameter value (either the returned string, or the `VALUE` item in the 3-tuple) is always unquoted, unless *unquote* is set to `False`.

This is a legacy method. On the `EmailMessage` class its functionality is replaced by the *params* property of the individual header objects returned by the header access methods.

set_param(*param*, *value*, *header='Content-Type'*, *requote=True*, *charset=None*, *language=''*, *replace=False*)

Set a parameter in the *Content-Type* header. If the parameter already exists in the header, its value will be replaced with *value*. If the *Content-Type* header as not yet been defined for this message, it will be set to *text/plain* and the new parameter value will be appended as per **RFC 2045** [https://datatracker.ietf.org/doc/html/rfc2045.html].

Optional *header* specifies an alternative header to *Content-Type*, and all parameters will be quoted as necessary unless optional *requote* is `False` (the default is `True`).

If optional *charset* is specified, the parameter will be encoded according to **RFC 2231** [https://datatracker.ietf.org/doc/html/rfc2231.html]. Optional *language* specifies the RFC 2231 language, defaulting to the empty string. Both *charset* and *language* should be strings.

If *replace* is `False` (the default) the header is moved to the end of the list of headers. If *replace* is `True`, the header will be updated in place.

Changed in version 3.4: `replace` keyword was added.

del_param(*param*, *header='content-type'*, *requote=True*)

Remove the given parameter completely from the *Content-Type* header. The header will be re-written in place without the parameter or its value. All values will be quoted as necessary unless *requote* is `False` (the default is `True`). Optional *header* specifies an alternative to *Content-Type*.

set_type(*type*, *header='Content-Type'*, *requote=True*)

Set the main type and subtype for the *Content-Type* header. *type* must be a string in the form *maintype/subtype*, otherwise a `ValueError` is raised.

This method replaces the *Content-Type* header, keeping all the parameters in place. If *requote* is `False`, this leaves the existing header's quoting as is, otherwise the parameters will be quoted (the default).

An alternative header can be specified in the *header* argument. When the *Content-Type* header is set a *MIME-Version* header is also added.

This is a legacy method. On the `EmailMessage` class its functionality is replaced by the `make_` and `add_` methods.

get_filename(*failobj=None*)

Return the value of the `filename` parameter of the *Content-Disposition* header of the message. If the header does not have a `filename` parameter, this method falls back to looking for the `name` parameter on the *Content-Type* header. If neither is found, or the header is missing, then *failobj* is returned. The returned string will always be unquoted as per `email.utils.unquote()`.

get_boundary(*failobj=None*)

Return the value of the `boundary` parameter of the *Content-Type* header of the message, or *failobj* if either the header is missing, or has no `boundary` parameter. The returned string will always be unquoted as per `email.utils.unquote()`.

set_boundary(*boundary*)

Set the `boundary` parameter of the *Content-Type* header to *boundary*. `set_boundary()` will always quote *boundary* if necessary. A `HeaderParseError` is raised if the message object has no *Content-Type* header.

Note that using this method is subtly different than deleting the old *Content-Type* header and adding a new one with the new boundary via `add_header()`, because `set_boundary()` preserves the order of the *Content-Type* header in the list of headers. However, it does *not* preserve any continuation lines which may have been present in the original *Content-Type* header.

get_content_charset(*failobj=None*)

Return the `charset` parameter of the *Content-Type* header, coerced to lower case. If there is no *Content-Type* header, or if that header has no `charset` parameter, *failobj* is returned.

Note that this method differs from **`get_charset()`** which returns the **`Charset`** instance for the default encoding of the message body.

get_charsets(*failobj=None*)

Return a list containing the character set names in the message. If the message is a *multipart*, then the list will contain one element for each subpart in the payload, otherwise, it will be a list of length 1.

Each item in the list will be a string which is the value of the `charset` parameter in the *Content-Type* header for the represented subpart. However, if the subpart has no *Content-Type* header, no `charset` parameter, or is not of the *text* main MIME type, then that item in the returned list will be *failobj*.

get_content_disposition()

Return the lowercased value (without parameters) of the message's *Content-Disposition* header if it has one, or `None`. The possible values for this method are *inline*, *attachment* or `None` if the message follows **RFC 2183** [https://datatracker.ietf.org/doc/html/rfc2183.html].

Added in version 3.5.

walk()

The **`walk()`** method is an all-purpose generator which can be used to iterate over all the parts and subparts of a message object tree, in depth-first traversal order.

You will typically use `walk()` as the iterator in a `for` loop; each iteration returns the next subpart.

Here’s an example that prints the MIME type of every part of a multipart message structure:

```
>>> for part in msg.walk():
...     print(part.get_content_type())
multipart/report
text/plain
message/delivery-status
text/plain
text/plain
message/rfc822
text/plain
```

`walk` iterates over the subparts of any part where `is_multipart()` returns `True`, even though `msg.get_content_maintype() == 'multipart'` may return `False`. We can see this in our example by making use of the `_structure` debug helper function:

```
>>> for part in msg.walk():
...     print(part.get_content_maintype() == 'multipart',
...           part.is_multipart())
True True
False False
False True
False False
False False
False True
False False
>>> _structure(msg)
multipart/report
    text/plain
    message/delivery-status
        text/plain
        text/plain
    message/rfc822
        text/plain
```

Here the `message` parts are not `multiparts`, but they do contain subparts. `is_multipart()` returns `True` and `walk` descends into the subparts.

`Message` objects can also optionally contain two instance attributes, which can be used when generating the plain text of a MIME message.

preamble

The format of a MIME document allows for some text between the blank line following the headers, and the first multipart boundary string. Normally, this text is never visible in a MIME-aware mail reader because it falls outside the standard MIME armor. However, when viewing the raw text of the message, or when viewing the message in a non-MIME aware reader, this text can become visible.

The *preamble* attribute contains this leading extra-armor text for MIME documents. When the **`Parser`** discovers some text after the headers but before the first boundary string, it assigns this text to the message's *preamble* attribute. When the **`Generator`** is writing out the plain text representation of a MIME message, and it finds the message has a *preamble* attribute, it will write this text in the area between the headers and the first boundary. See **`email.parser`** and **`email.generator`** for details.

Note that if the message object has no preamble, the *preamble* attribute will be `None`.

epilogue

The *epilogue* attribute acts the same way as the *preamble* attribute, except that it contains text that appears between the last boundary and the end of the message.

You do not need to set the epilogue to the empty string in order for the `Generator` to print a newline at the end of the file.

defects

The *defects* attribute contains a list of all the problems found when parsing this message. See `email.errors` for a detailed description of the possible parsing defects.

`email.mime`: Creating email and MIME objects from scratch

Source code: Lib/email/mime/ [https://github.com/python/cpython/tree/3.14/Lib/email/mime/]

This module is part of the legacy (`Compat32`) email API. Its functionality is partially replaced by the **`contentmanager`** in the new API, but in certain applications these classes may still be useful, even in non-legacy code.

Ordinarily, you get a message object structure by passing a file or some text to a parser, which parses the text and returns the root message object. However you can also build a complete message structure from scratch, or even individual **`Message`** objects by hand. In fact, you can also take an existing structure and add new **`Message`** objects, move them around, etc. This makes a very convenient interface for slicing-and-dicing MIME messages.

You can create a new object structure by creating **`Message`** instances, adding attachments and all the appropriate headers manually. For MIME messages though, the **`email`** package provides some convenient subclasses to make things easier.

Here are the classes:

class email.mime.base.MIMEBase(*_maintype*, *_subtype*, *, *policy=compat32*, ***_params*)

Module: `email.mime.base`

This is the base class for all the MIME-specific subclasses of `Message`. Ordinarily you won’t create instances specifically of `MIMEBase`, although you could. `MIMEBase` is provided primarily as a convenient base class for more specific MIME-aware subclasses.

_maintype is the *Content-Type* major type (e.g. *text* or *image*), and *_subtype* is the *Content-Type* minor type (e.g. *plain* or *gif*). *_params* is a parameter key/value dictionary and is passed directly to `Message.add_header`.

If *policy* is specified, (defaults to the `compat32` policy) it will be passed to `Message`.

The `MIMEBase` class always adds a *Content-Type* header (based on *_maintype*, *_subtype*, and *_params*), and a *MIME-Version* header (always set to `1.0`).

Changed in version 3.6: Added *policy* keyword-only parameter.

class email.mime.nonmultipart.MIMENonMultipart

Module: `email.mime.nonmultipart`

A subclass of `MIMEBase`, this is an intermediate base class for MIME messages that are not *multipart*. The primary purpose of this class is to prevent the use of the `attach()` method, which only makes sense for *multipart* messages. If `attach()` is called, a `MultipartConversionError` exception is raised.

class email.mime.multipart.MIMEMultipart(*_subtype*='*mixed*', *boundary=None*, *_subparts=None*, ***, *policy=compat32*, ***_params*)

Module: [`email.mime.multipart`]

A subclass of [`MIMEBase`], this is an intermediate base class for MIME messages that are *multipart*. Optional *_subtype* defaults to *mixed*, but can be used to specify the subtype of the message. A *Content-Type* header of *multipart/_subtype* will be added to the message object. A *MIME-Version* header will also be added.

Optional *boundary* is the multipart boundary string. When `None` (the default), the boundary is calculated when needed (for example, when the message is serialized).

_subparts is a sequence of initial subparts for the payload. It must be possible to convert this sequence to a list. You can always attach new subparts to the message by using the [`Message.attach`] method.

Optional *policy* argument defaults to [`compat32`].

Additional parameters for the *Content-Type* header are taken from the keyword arguments, or passed into the *_params* argument, which is a keyword dictionary.

Changed in version 3.6: Added *policy* keyword-only parameter.

class email.mime.application.MIMEApplication(*_data*, *_subtype='octet-stream'*, *_encoder=email.encoders.encode_base64*, ***, *policy=compat32*, ***_params*)

Module: [`email.mime.application`]

A subclass of `MIMENonMultipart`, the `MIMEApplication` class is used to represent MIME message objects of major type *application*. *_data* contains the bytes for the raw application data. Optional *_subtype* specifies the MIME subtype and defaults to *octet-stream*.

Optional *_encoder* is a callable (i.e. function) which will perform the actual encoding of the data for transport. This callable takes one argument, which is the `MIMEApplication` instance. It should use `get_payload()` and `set_payload()` to change the payload to encoded form. It should also add any *Content-Transfer-Encoding* or other headers to the message object as necessary. The default encoding is base64. See the `email.encoders` module for a list of the built-in encoders.

Optional *policy* argument defaults to `compat32`.

_params are passed straight through to the base class constructor.

Changed in version 3.6: Added *policy* keyword-only parameter.

class email.mime.audio.MIMEAudio(*_audiodata*, *_subtype=None*, *_encoder=email.encoders.encode_base64*, *, *policy=compat32*, ***_params*)

Module: `email.mime.audio`

A subclass of `MIMENonMultipart`, the `MIMEAudio` class is used to create MIME message objects of major type *audio*. *_audiodata* contains the bytes for the raw audio data. If this data can be decoded as au, wav, aiff, or aifc, then the subtype will be automatically included in the *Content-Type* header. Otherwise you can explicitly specify the audio subtype via the *_subtype* argument. If the minor type could not be guessed and *_subtype* was not given, then `TypeError` is raised.

Optional *_encoder* is a callable (i.e. function) which will perform the actual encoding of the audio data for transport. This callable takes one argument, which is the `MIMEAudio` instance. It should use `get_payload()` and `set_payload()` to change the payload to encoded form. It should also add any *Content-Transfer-Encoding* or other headers to the message object as necessary. The default encoding is base64. See the `email.encoders` module for a list of the built-in encoders.

Optional *policy* argument defaults to `compat32`.

_params are passed straight through to the base class constructor.

Changed in version 3.6: Added *policy* keyword-only parameter.

class email.mime.image.MIMEImage(*_imagedata*, *_subtype=None*, *_encoder=email.encoders.encode_base64*, ***, *policy=compat32*, ***_params*)

Module: `email.mime.image`

A subclass of `MIMENonMultipart`, the `MIMEImage` class is used to create MIME message objects of major type *image*. *_imagedata* contains the bytes for the raw image data. If this data type can be detected (jpeg, png, gif, tiff, rgb, pbm, pgm, ppm, rast, xbm, bmp, webp, and exr attempted), then the subtype will be automatically included in the *Content-Type* header. Otherwise you can explicitly specify the image subtype via the *_subtype* argument. If the minor type could not be guessed and *_subtype* was not given, then `TypeError` is raised.

Optional *_encoder* is a callable (i.e. function) which will perform the actual encoding of the image data for transport. This callable takes one argument, which is the `MIMEImage` instance. It should use `get_payload()` and `set_payload()` to change the payload to encoded form. It should also add any *Content-Transfer-Encoding* or other headers to the message object as necessary. The default encoding is base64. See the `email.encoders` module for a list of the built-in encoders.

Optional *policy* argument defaults to `compat32`.

_params are passed straight through to the `MIMEBase` constructor.

Changed in version 3.6: Added *policy* keyword-only parameter.

class email.mime.message.MIMEMessage(*_msg*, *_subtype='rfc822'*, ***, *policy=compat32*)

Module: `email.mime.message`

A subclass of `MIMENonMultipart`, the `MIMEMessage` class is used to create MIME objects of main type *message*. *_msg* is used as the payload, and must be an instance of class `Message` (or a subclass thereof), otherwise a `TypeError` is raised.

Optional *_subtype* sets the subtype of the message; it defaults to *rfc822*.

Optional *policy* argument defaults to `compat32`.

Changed in version 3.6: Added *policy* keyword-only parameter.

class email.mime.text.MIMEText(*_text*, *_subtype='plain'*, *_charset=None*, *, *policy=compat32*)

Module: `email.mime.text`

A subclass of `MIMENonMultipart`, the `MIMEText` class is used to create MIME objects of major type *text*. *_text* is the string for the payload. *_subtype* is the minor type and defaults to *plain*. *_charset* is the character set of the text and is passed as an argument to the `MIMENonMultipart` constructor; it defaults to `us-ascii` if the string contains only `ascii` code points, and `utf-8` otherwise. The *_charset* parameter accepts either a string or a `Charset` instance.

Unless the *_charset* argument is explicitly set to `None`, the MIMEText object created will have both a *Content-Type* header with a `charset` parameter, and a *Content-Transfer-Encoding* header. This means that a subsequent `set_payload` call will not result in an encoded payload, even if a charset is passed in the `set_payload` command. You can “reset” this behavior by deleting the `Content-Transfer-Encoding` header, after which a `set_payload` call will automatically encode the new payload (and add a new *Content-Transfer-Encoding* header).

Optional *policy* argument defaults to `compat32`.

Changed in version 3.5: *_charset* also accepts `Charset` instances.

Changed in version 3.6: Added *policy* keyword-only parameter.

`email.header`: Internationalized headers

Source code: Lib/email/header.py [https://github.com/python/cpython/tree/3.14/Lib/email/header.py]

This module is part of the legacy (`Compat32`) email API. In the current API encoding and decoding of headers is handled transparently by the dictionary-like API of the **`EmailMessage`** class. In addition to uses in legacy code, this module can be useful in applications that need to completely control the character sets used when encoding headers.

The remaining text in this section is the original documentation of the module.

RFC 2822 [https://datatracker.ietf.org/doc/html/rfc2822.html] is the base standard that describes the format of email messages. It derives from the older **RFC 822** [https://datatracker.ietf.org/doc/html/rfc822.html] standard which came into widespread use at a time when most email was composed of ASCII characters only. **RFC 2822** [https://datatracker.ietf.org/doc/html/rfc2822.html] is a specification written assuming email contains only 7-bit ASCII characters.

Of course, as email has been deployed worldwide, it has become internationalized, such that language specific character sets can now be used in email messages. The base standard still requires email messages to be transferred using only 7-bit ASCII characters, so a slew of RFCs have been written describing how to encode email containing non-ASCII characters into **RFC 2822** [https://datatracker.ietf.org/doc/html/rfc2822.html]-compliant format. These RFCs include **RFC 2045** [https://datatracker.ietf.org/doc/html/rfc2045.html], **RFC 2046** [https://datatracker.ietf.org/doc/html/rfc2046.html], **RFC 2047** [https://datatracker.ietf.org/doc/html/rfc2047.html], and **RFC 2231** [https://datatracker.ietf.org/doc/html/rfc2231.html]. The `email` package supports these standards in its `email.header` and `email.charset` modules.

If you want to include non-ASCII characters in your email headers, say in the *Subject* or *To* fields, you should use the `Header` class and assign the field in the `Message` object to an instance of `Header` instead of using a string for the header value. Import the `Header` class from the `email.header` module. For example:

```
>>> from email.message import Message
>>> from email.header import Header
>>> msg = Message()
>>> h = Header('p\xf6stal', 'iso-8859-1')
>>> msg['Subject'] = h
>>> msg.as_string()
'Subject: =?iso-8859-1?q?p=F6stal?=\n\n'
```

Notice here how we wanted the *Subject* field to contain a non-ASCII character? We did this by creating a `Header` instance and passing in the character set that the byte string was encoded in. When the subsequent `Message` instance was flattened, the *Subject* field was properly **RFC 2047** [https://datatracker.ietf.org/doc/html/rfc2047.html] encoded. MIME-aware mail readers would show this header using the embedded ISO-8859-1 character.

Here is the `Header` class description:

class email.header.Header(*s=None*, *charset=None*, *maxlinelen=None*, *header_name=None*, *continuation_ws='*

', *errors='strict'*)

Create a MIME-compliant header that can contain strings in different character sets.

Optional *s* is the initial header value. If `None` (the default), the initial header value is not set. You can later append to the header with `append()` method calls. *s* may be an instance of `bytes` or `str`, but see the `append()` documentation for semantics.

Optional *charset* serves two purposes: it has the same meaning as the *charset* argument to the `append()` method. It also sets the default character set for all subsequent `append()` calls that omit the *charset* argument. If *charset* is not provided in the constructor (the default), the `us-ascii` character set is used both as *s*'s initial charset and as the default for subsequent `append()` calls.

The maximum line length can be specified explicitly via *maxlinelen*. For splitting the first line to a shorter value (to account for the field header which isn't included in *s*, e.g. *Subject*) pass in the name of the field in *header_name*. The default *maxlinelen* is 78, and the default value for *header_name* is `None`, meaning it is not taken into account for the first line of a long, split header.

Optional *continuation_ws* must be **RFC 2822** [https://datatracker.ietf.org/doc/html/rfc2822.html]-compliant folding whitespace, and is usually either a space or a hard tab character. This character will be prepended to continuation lines. *continuation_ws* defaults to a single space character.

Optional *errors* is passed straight through to the `append()` method.

append(*s*, *charset=None*, *errors='strict'*)

Append the string *s* to the MIME header.

Optional *charset*, if given, should be a `Charset` instance (see `email.charset`) or the name of a character set, which will be converted to a `Charset` instance. A value of `None` (the default) means that the *charset* given in the constructor is used.

s may be an instance of `bytes` or `str`. If it is an instance of `bytes`, then *charset* is the encoding of that byte string, and a `UnicodeError` will be raised if the string cannot be decoded with that character set.

If *s* is an instance of `str`, then *charset* is a hint specifying the character set of the characters in the string.

In either case, when producing an **RFC 2822** [https://datatracker.ietf.org/doc/html/rfc2822.html]-compliant header using **RFC 2047** [https://datatracker.ietf.org/doc/html/rfc2047.html] rules, the string will be encoded using the output codec of the charset. If the string cannot be encoded using the output codec, a UnicodeError will be raised.

Optional *errors* is passed as the errors argument to the decode call if *s* is a byte string.

encode(*splitchars=';, \t'*, *maxlinelen=None*, *linesep='\n'*)

Encode a message header into an RFC-compliant format, possibly wrapping long lines and encapsulating non-ASCII parts in base64 or quoted-printable encodings.

Optional *splitchars* is a string containing characters which should be given extra weight by the splitting algorithm during normal header wrapping. This is in very rough support of **RFC 2822** [https://datatracker.ietf.org/doc/html/rfc2822.html]'s ‘higher level syntactic breaks’: split points preceded by a splitchar are preferred during line splitting, with the characters preferred in the order in which they appear in the string. Space and tab may be included in the string to indicate whether preference should be given to one over the other as a split point when other split chars do not appear in the line being split. Splitchars does not affect **RFC 2047** [https://datatracker.ietf.org/doc/html/rfc2047.html] encoded lines.

maxlinelen, if given, overrides the instance’s value for the maximum line length.

linesep specifies the characters used to separate the lines of the folded header. It defaults to the most useful value for Python application code (`\n`), but `\r\n` can be specified in order to produce headers with RFC-compliant line separators.

Changed in version 3.2: Added the *linesep* argument.

The **`Header`** class also provides a number of methods to support standard operators and built-in functions.

__str__()

Returns an approximation of the **`Header`** as a string, using an unlimited line length. All pieces are converted to unicode using the specified encoding and joined together appropriately. Any pieces with a charset of `'unknown-8bit'` are decoded as ASCII using the `'replace'` error handler.

Changed in version 3.2: Added handling for the `'unknown-8bit'` charset.

__eq__(*other*)

This method allows you to compare two **`Header`** instances for equality.

__ne__(*other*)

This method allows you to compare two **`Header`** instances for inequality.

The **`email.header`** module also provides the following convenient functions.

email.header.decode_header(*header*)

Decode a message header value without converting the character set. The header value is in *header*.

For historical reasons, this function may return either:

1. A list of pairs containing each of the decoded parts of the header, `(decoded_bytes, charset)`, where *decoded_bytes* is always an instance of **`bytes`**, and *charset* is either:
    - A lower case string containing the name of the character set specified.
    - `None` for non-encoded parts of the header.
2. A list of length 1 containing a pair `(string, None)`, where *string* is always an instance of **`str`**.

An **`email.errors.HeaderParseError`** may be raised when certain decoding errors occur (e.g. a base64 decoding exception).

Here are examples:

```
>>> from email.header import decode_header
>>> decode_header('=?iso-8859-1?q?p=F6stal?=')
[(b'p\xf6stal', 'iso-8859-1')]
>>> decode_header('unencoded_string')
[('unencoded_string', None)]
>>> decode_header('bar =?utf-8?B?ZsOzbw==?=')
[(b'bar ', None), (b'f\xc3\xb3o', 'utf-8')]
```

Note

This function exists for backwards compatibility only. For new code, we recommend using `email.headerregistry.HeaderRegistry`.

email.header.make_header(*decoded_seq*, *maxlinelen=None*, *header_name=None*, *continuation_ws=' '*)

Create a `Header` instance from a sequence of pairs as returned by `decode_header()`.

`decode_header()` takes a header value string and returns a sequence of pairs of the format `(decoded_string, charset)` where *charset* is the name of the character set.

This function takes one of those sequence of pairs and returns a `Header` instance. Optional *maxlinelen*, *header_name*, and *continuation_ws* are as in the `Header` constructor.

Note

This function exists for backwards compatibility only, and is not recommended for use in new code.

`email.charset`: Representing character sets

Source code: Lib/email/charset.py
[https://github.com/python/cpython/tree/3.14/Lib/email/charset.py]

This module is part of the legacy (`Compat32`) email API. In the new API only the aliases table is used.

The remaining text in this section is the original documentation of the module.

This module provides a class **`Charset`** for representing character sets and character set conversions in email messages, as well as a character set registry and several convenience methods for manipulating this registry. Instances of **`Charset`** are used in several other modules within the **`email`** package.

Import this class from the **`email.charset`** module.

class
email.charset.Charset(*input_charset=DEFAULT_CHARSET*)

Map character sets to their email properties.

This class provides information about the requirements imposed on email for a specific character set. It also provides convenience routines for converting between character sets, given the availability of the applicable codecs. Given a character set, it will do its best to provide information on how to use that character set in an email message in an RFC-compliant way.

Certain character sets must be encoded with quoted-printable or base64 when used in email headers or bodies. Certain character sets must be converted outright, and are not allowed in email.

Optional *input_charset* is as described below; it is always coerced to lower case. After being alias normalized it is also used as a lookup into the registry of character sets to find out the header encoding, body encoding, and output conversion codec to be used for the character set. For example, if *input_charset* is `iso-8859-1`, then headers and bodies will be encoded using quoted-printable and no output conversion codec is necessary. If *input_charset* is `euc-jp`, then headers will be encoded with base64, bodies will not be encoded, but output text will be converted from the `euc-jp` character set to the `iso-2022-jp` character set.

`Charset` instances have the following data attributes:

input_charset

The initial character set specified. Common aliases are converted to their *official* email names (e.g. `latin_1` is converted to `iso-8859-1`). Defaults to 7-bit `us-ascii`.

header_encoding

If the character set must be encoded before it can be used in an email header, this attribute will be set to `charset.QP` (for quoted-printable), `charset.BASE64` (for base64 encoding), or `charset.SHORTEST` for the shortest of QP or BASE64 encoding. Otherwise, it will be `None`.

body_encoding

Same as *header_encoding*, but describes the encoding for the mail message’s body, which indeed may be different than the header encoding. `charset.SHORTEST` is not allowed for *body_encoding*.

output_charset

Some character sets must be converted before they can be used in email headers or bodies. If the *input_charset* is one of them, this attribute will contain the name of the character set output will be converted to. Otherwise, it will be `None`.

input_codec

The name of the Python codec used to convert the *input_charset* to Unicode. If no conversion codec is necessary, this attribute will be `None`.

output_codec

The name of the Python codec used to convert Unicode to the *output_charset*. If no conversion codec is necessary, this attribute will have the same value as the *input_codec*.

`Charset` instances also have the following methods:

get_body_encoding()

Return the content transfer encoding used for body encoding.

This is either the string `quoted-printable` or `base64` depending on the encoding used, or it is a function, in which case you should call the function with a single argument, the Message object being encoded. The function should then set the *Content-Transfer-Encoding* header itself to whatever is appropriate.

Returns the string `quoted-printable` if *body_encoding* is `QP`, returns the string `base64` if *body_encoding* is `BASE64`, and returns the string `7bit` otherwise.

get_output_charset()

Return the output character set.

This is the *output_charset* attribute if that is not `None`, otherwise it is *input_charset*.

header_encode(*string*)

Header-encode the string *string*.

The type of encoding (base64 or quoted-printable) will be based on the *header_encoding* attribute.

header_encode_lines(*string*, *maxlengths*)

Header-encode a *string* by converting it first to bytes.

This is similar to **`header_encode()`** except that the string is fit into maximum line lengths as given by the argument *maxlengths*, which must be an iterator: each element returned from this iterator will provide the next maximum line length.

body_encode(*string*)

Body-encode the string *string*.

The type of encoding (base64 or quoted-printable) will be based on the *body_encoding* attribute.

The **`Charset`** class also provides a number of methods to support standard operations and built-in functions.

__str__()

Returns *input_charset* as a string coerced to lower case. **`__repr__()`** is an alias for **`__str__()`**.

__eq__(*other*)

This method allows you to compare two `Charset` instances for equality.

__ne__(*other*)

This method allows you to compare two `Charset` instances for inequality.

The `email.charset` module also provides the following functions for adding new entries to the global character set, alias, and codec registries:

email.charset.add_charset(*charset*, *header_enc=None*, *body_enc=None*, *output_charset=None*)

Add character properties to the global registry.

charset is the input character set, and must be the canonical name of a character set.

Optional *header_enc* and *body_enc* is either `charset.QP` for quoted-printable, `charset.BASE64` for base64 encoding, `charset.SHORTEST` for the shortest of quoted-printable or base64 encoding, or `None` for no encoding. `SHORTEST` is only valid for *header_enc*. The default is `None` for no encoding.

Optional *output_charset* is the character set that the output should be in. Conversions will proceed from input charset, to Unicode, to the output charset when the method **`Charset.convert()`** is called. The default is to output in the same character set as the input.

Both *input_charset* and *output_charset* must have Unicode codec entries in the module’s character set-to-codec mapping; use `add_codec()` to add codecs the module does not know about. See the `codecs` module’s documentation for more information.

The global character set registry is kept in the module global dictionary `CHARSETS`.

email.charset.add_alias(*alias*, *canonical*)

Add a character set alias. *alias* is the alias name, e.g. `latin-1`. *canonical* is the character set's canonical name, e.g. `iso-8859-1`.

The global charset alias registry is kept in the module global dictionary `ALIASES`.

email.charset.add_codec(*charset*, *codecname*)

Add a codec that map characters in the given character set to and from Unicode.

charset is the canonical name of a character set. *codecname* is the name of a Python codec, as appropriate for the second argument to the **`str`**'s **`encode()`** method.

`email.encoders`: Encoders

Source code: Lib/email/encoders.py
[https://github.com/python/cpython/tree/3.14/Lib/email/encoders.py]

This module is part of the legacy (`Compat32`) email API. In the new API the functionality is provided by the *cte* parameter of the **`set_content()`** method.

This module is deprecated in Python 3. The functions provided here should not be called explicitly since the **`MIMEText`** class sets the content type and CTE header using the *_subtype* and *_charset* values passed during the instantiation of that class.

The remaining text in this section is the original documentation of the module.

When creating **`Message`** objects from scratch, you often need to encode the payloads for transport through compliant mail servers. This is especially true for *image/** and *text/** type messages containing binary data.

The **`email`** package provides some convenient encoders in its **`encoders`** module. These encoders are actually used by the **`MIMEAudio`** and **`MIMEImage`** class constructors to provide default encodings. All encoder functions take exactly one argument, the message object to encode. They usually extract the payload, encode it, and reset the payload to this newly encoded value. They should also set the *Content-Transfer-Encoding* header as appropriate.

Note that these functions are not meaningful for a multipart message. They must be applied to individual subparts instead, and will raise a `TypeError` if passed a message whose type is multipart.

Here are the encoding functions provided:

email.encoders.encode_quopri(*msg*)

Encodes the payload into quoted-printable form and sets the *Content-Transfer-Encoding* header to `quoted-printable` [[1]]. This is a good encoding to use when most of your payload is normal printable data, but contains a few unprintable characters.

email.encoders.encode_base64(*msg*)

Encodes the payload into base64 form and sets the *Content-Transfer-Encoding* header to `base64`. This is a good encoding to use when most of your payload is unprintable data since it is a more compact form than quoted-printable. The drawback of base64 encoding is that it renders the text non-human readable.

email.encoders.encode_7or8bit(*msg*)

This doesn’t actually modify the message’s payload, but it does set the *Content-Transfer-Encoding* header to either `7bit` or `8bit` as appropriate, based on the payload data.

email.encoders.encode_noop(*msg*)

This does nothing; it doesn’t even set the *Content-Transfer-Encoding* header.

Footnotes

[1]

Note that encoding with `encode_quopri()` also encodes all tabs and space characters in the data.

`email.utils`: Miscellaneous utilities

Source code: Lib/email/utils.py
[https://github.com/python/cpython/tree/3.14/Lib/email/utils.py]

There are a couple of useful utilities provided in the **`email.utils`** module:

email.utils.localtime(*dt=None*)

Return local time as an aware datetime object. If called without arguments, return current time. Otherwise *dt* argument should be a **`datetime`** instance, and it is converted to the local time zone according to the system time zone database. If *dt* is naive (that is, `dt.tzinfo` is `None`), it is assumed to be in local time.

Added in version 3.3.

Deprecated since version 3.12, removed in version 3.14: The *isdst* parameter.

email.utils.make_msgid(*idstring=None*, *domain=None*)

Returns a string suitable for an **RFC 2822** [https://datatracker.ietf.org/doc/html/rfc2822.html]-compliant *Message-ID* header. Optional *idstring* if given, is a string used to strengthen the uniqueness of the message id. Optional *domain* if given provides the portion of the msgid after the ‘@’. The default is the local hostname. It is not normally necessary to override this default, but may be useful certain cases, such as a constructing distributed system that uses a consistent domain name across multiple hosts.

Changed in version 3.2: Added the *domain* keyword.

The remaining functions are part of the legacy (`Compat32`) email API. There is no need to directly use these with the new API, since the parsing and formatting they provide is done automatically by the header parsing machinery of the new API.

email.utils.quote(*str*)

Return a new string with backslashes in *str* replaced by two backslashes, and double quotes replaced by backslash-double quote.

email.utils.unquote(*str*)

Return a new string which is an *unquoted* version of *str*. If *str* ends and begins with double quotes, they are stripped off. Likewise if *str* ends and begins with angle brackets, they are stripped off.

email.utils.parseaddr(*address*, *, *strict=True*)

Parse address – which should be the value of some address-containing field such as *To* or *Cc* – into its constituent *realname* and *email address* parts. Returns a tuple of that information, unless the parse fails, in which case a 2-tuple of `('', '')` is returned.

If *strict* is true, use a strict parser which rejects malformed inputs.

Changed in version 3.13: Add *strict* optional parameter and reject malformed inputs by default.

email.utils.formataddr(*pair*, *charset='utf-8'*)

The inverse of `parseaddr()`, this takes a 2-tuple of the form `(realname, email_address)` and returns the string value suitable for a *To* or *Cc* header. If the first element of *pair* is false, then the second element is returned unmodified.

Optional *charset* is the character set that will be used in the **RFC 2047** [https://datatracker.ietf.org/doc/html/rfc2047.html] encoding of the `realname` if the `realname` contains non-ASCII characters. Can be an instance of `str` or a `Charset`. Defaults to `utf-8`.

Changed in version 3.3: Added the *charset* option.

email.utils.getaddresses(*fieldvalues*, *, *strict=True*)

This method returns a list of 2-tuples of the form returned by `parseaddr()`. *fieldvalues* is a sequence of header field values as might be returned by `Message.get_all`.

If *strict* is true, use a strict parser which rejects malformed inputs.

Here’s a simple example that gets all the recipients of a message:

```
from email.utils import getaddresses

tos = msg.get_all('to', [])
ccs = msg.get_all('cc', [])
resent_tos = msg.get_all('resent-to', [])
resent_ccs = msg.get_all('resent-cc', [])
all_recipients = getaddresses(tos + ccs + resent_tos +
resent_ccs)
```

Changed in version 3.13: Add *strict* optional parameter and reject malformed inputs by default.

email.utils.parsedate(*date*)

Attempts to parse a date according to the rules in **RFC 2822** [https://datatracker.ietf.org/doc/html/rfc2822.html]. however, some mailers don't follow that format as specified, so `parsedate()` tries to guess correctly in such cases. *date* is a string containing an **RFC 2822** [https://datatracker.ietf.org/doc/html/rfc2822.html] date, such as `"Mon, 20 Nov 1995 19:12:08 -0500"`. If it succeeds in parsing the date, `parsedate()` returns a 9-tuple that can be passed directly to `time.mktime()`; otherwise `None` will be returned. Note that indexes 6, 7, and 8 of the result tuple are not usable.

email.utils.parsedate_tz(*date*)

Performs the same function as `parsedate()`, but returns either `None` or a 10-tuple; the first 9 elements make up a tuple that can be passed directly to `time.mktime()`, and the tenth is the offset of the date's timezone from UTC (which is the official term for Greenwich Mean Time) [[1]]. If the input string has no timezone, the last element of the tuple returned is `0`, which represents UTC. Note that indexes 6, 7, and 8 of the result tuple are not usable.

email.utils.parsedate_to_datetime(*date*)

The inverse of `format_datetime()`. Performs the same function as `parsedate()`, but on success returns a `datetime`; otherwise `ValueError` is raised if *date* contains an invalid value such as an hour greater than 23 or a timezone offset not between -24 and 24 hours. If the input date has a timezone of `-0000`, the `datetime` will be a naive `datetime`, and if the date is conforming to the RFCs it will represent a time in UTC but with no indication of the actual source timezone of the message the date comes from. If the input date has any other valid timezone offset, the `datetime` will be an aware `datetime` with the corresponding a `timezone` `tzinfo`.

Added in version 3.3.

email.utils.mktime_tz(*tuple*)

Turn a 10-tuple as returned by [`parsedate_tz()`] into a UTC timestamp (seconds since the Epoch). If the timezone item in the tuple is `None`, assume local time.

email.utils.formatdate(*timeval=None*, *localtime=False*, *usegmt=False*)

Returns a date string as per **RFC 2822** [https://datatracker.ietf.org/doc/html/rfc2822.html], e.g.:

```
Fri, 09 Nov 2001 01:08:47 -0000
```

Optional *timeval* if given is a floating-point time value as accepted by `time.gmtime()` and `time.localtime()`, otherwise the current time is used.

Optional *localtime* is a flag that when `True`, interprets *timeval*, and returns a date relative to the local timezone instead of UTC, properly taking daylight savings time into account. The default is `False` meaning UTC is used.

Optional *usegmt* is a flag that when `True`, outputs a date string with the timezone as an ascii string `GMT`, rather than a numeric `-0000`. This is needed for some protocols (such as HTTP). This only applies when *localtime* is `False`. The default is `False`.

email.utils.format_datetime(*dt*, *usegmt=False*)

Like `formatdate`, but the input is a `datetime` instance. If it is a naive datetime, it is assumed to be “UTC with no information about the source timezone”, and the conventional `-0000` is used for the timezone. If it is an aware `datetime`, then the numeric timezone offset is used. If it is an aware timezone with offset zero, then *usegmt* may be set to `True`, in which case the string `GMT` is used instead of the numeric timezone offset. This provides a way to generate standards conformant HTTP date headers.

Added in version 3.3.

email.utils.decode_rfc2231(*s*)

Decode the string *s* according to **RFC 2231** [https://datatracker.ietf.org/doc/html/rfc2231.html].

email.utils.encode_rfc2231(*s*, *charset=None*, *language=None*)

Encode the string *s* according to **RFC 2231** [https://datatracker.ietf.org/doc/html/rfc2231.html]. Optional *charset* and *language*, if given is the character set name and language name to use. If neither is given, *s* is returned as-is. If *charset* is given but *language* is not, the string is encoded using the empty string for *language*.

email.utils.collapse_rfc2231_value(*value*, *errors='replace'*, *fallback_charset='us-ascii'*)

When a header parameter is encoded in **RFC 2231** [https://datatracker.ietf.org/doc/html/rfc2231.html] format, `Message.get_param` may return a 3-tuple containing the character set, language, and value. `collapse_rfc2231_value()` turns this into a unicode string. Optional *errors* is passed to the *errors* argument of `str`’s `encode()` method; it defaults to `'replace'`. Optional *fallback_charset* specifies the character set to use if the one in the **RFC 2231** [https://datatracker.ietf.org/doc/html/rfc2231.html] header is not known by Python; it defaults to `'us-ascii'`.

For convenience, if the *value* passed to `collapse_rfc2231_value()` is not a tuple, it should be a string and it is returned unquoted.

email.utils.decode_params(*params*)

Decode parameters list according to **RFC 2231** [https://datatracker.ietf.org/doc/html/rfc2231.html]. *params* is a sequence of 2-tuples containing elements of the form `(content-type, string-value)`.

Footnotes

---

[1] Note that the sign of the timezone offset is the opposite of the sign of the `time.timezone` variable for the same timezone; the latter variable follows the POSIX standard while this module follows **RFC 2822** [https://datatracker.ietf.org/doc/html/rfc2822.html].

`email.iterators`: Iterators

Source code: Lib/email/iterators.py
[https://github.com/python/cpython/tree/3.14/Lib/email/iterators.py]

Iterating over a message object tree is fairly easy with the `Message.walk` method. The `email.iterators` module provides some useful higher level iterations over message object trees.

email.iterators.body_line_iterator(*msg*, *decode=False*)

This iterates over all the payloads in all the subparts of *msg*, returning the string payloads line-by-line. It skips over all the subpart headers, and it skips over any subpart with a payload that isn’t a Python string. This is somewhat equivalent to reading the flat text representation of the message from a file using `readline()`, skipping over all the intervening headers.

Optional *decode* is passed through to `Message.get_payload`.

email.iterators.typed_subpart_iterator(*msg*, *maintype='text'*, *subtype=None*)

This iterates over all the subparts of *msg*, returning only those subparts that match the MIME type specified by *maintype* and *subtype*.

Note that *subtype* is optional; if omitted, then subpart MIME type matching is done only with the main type. *maintype* is optional too; it defaults to *text*.

Thus, by default `typed_subpart_iterator()` returns each subpart that has a MIME type of *text/**.

The following function has been added as a useful debugging tool. It should *not* be considered part of the supported public interface for the package.

email.iterators._structure(*msg*, *fp=None*, *level=0*, *include_default=False*)

Prints an indented representation of the content types of the message object structure. For example:

```
>>> msg = email.message_from_file(somefile)
>>> _structure(msg)
multipart/mixed
    text/plain
    text/plain
    multipart/digest
        message/rfc822
            text/plain
        message/rfc822
            text/plain
        message/rfc822
            text/plain
        message/rfc822
            text/plain
        message/rfc822
            text/plain
    text/plain
```

Optional *fp* is a file-like object to print the output to. It must be suitable for Python’s `print()` function. *level* is used internally. *include_default*, if true, prints the default type as well.

`json` — JSON encoder and decoder

Source code: Lib/json/__init__.py [https://github.com/python/cpython/tree/3.14/Lib/json/__init__.py]

JSON (JavaScript Object Notation) [https://json.org], specified by **RFC 7159** [https://datatracker.ietf.org/doc/html/rfc7159.html] (which obsoletes **RFC 4627** [https://datatracker.ietf.org/doc/html/rfc4627.html]) and by ECMA-404 [https://ecma-international.org/publications-and-standards/standards/ecma-404/], is a lightweight data interchange format inspired by JavaScript [https://en.wikipedia.org/wiki/JavaScript] object literal syntax (although it is not a strict subset of JavaScript [[1]]).

Note

The term “object” in the context of JSON processing in Python can be ambiguous. All values in Python are objects. In JSON, an object refers to any data wrapped in curly braces, similar to a Python dictionary.

Warning

Be cautious when parsing JSON data from untrusted sources. A malicious JSON string may cause the decoder to consume considerable CPU and memory resources. Limiting the size of data to be parsed is recommended.

This module exposes an API familiar to users of the standard library **`marshal`** and **`pickle`** modules.

Encoding basic Python object hierarchies:

```
>>> import json
>>> json.dumps(['foo', {'bar': ('baz', None, 1.0, 2)}])
'["foo", {"bar": ["baz", null, 1.0, 2]}]'
>>> print(json.dumps("\"foo\bar"))
"\"foo\bar"
>>> print(json.dumps('\u1234'))
"\u1234"
>>> print(json.dumps('\\'))
"\\"
>>> print(json.dumps({"c": 0, "b": 0, "a": 0}, sort_keys=True))
{"a": 0, "b": 0, "c": 0}
>>> from io import StringIO
>>> io = StringIO()
>>> json.dump(['streaming API'], io)
>>> io.getvalue()
'["streaming API"]'
```

Compact encoding:

```
>>> import json
>>> json.dumps([1, 2, 3, {'4': 5, '6': 7}], separators=(',',
':'))
'[1,2,3,{"4":5,"6":7}]'
```

Pretty printing:

```
>>> import json
>>> print(json.dumps({'6': 7, '4': 5}, sort_keys=True,
indent=4))
{
    "4": 5,
    "6": 7
}
```

Customizing JSON object encoding:

```
>>> import json
>>> def custom_json(obj):
...     if isinstance(obj, complex):
...         return {'__complex__': True, 'real': obj.real,
'imag': obj.imag}
```

```
...     raise TypeError(f'Cannot serialize object of
{type(obj)}')
...
>>> json.dumps(1 + 2j, default=custom_json)
'{"__complex__": true, "real": 1.0, "imag": 2.0}'
```

Decoding JSON:

```
>>> import json
>>> json.loads('["foo", {"bar":["baz", null, 1.0, 2]}]')
['foo', {'bar': ['baz', None, 1.0, 2]}]
>>> json.loads('"\\"foo\\bar"')
'"foo\x08ar'
>>> from io import StringIO
>>> io = StringIO('["streaming API"]')
>>> json.load(io)
['streaming API']
```

Customizing JSON object decoding:

```
>>> import json
>>> def as_complex(dct):
...     if '__complex__' in dct:
...         return complex(dct['real'], dct['imag'])
...     return dct
...
>>> json.loads('{"__complex__": true, "real": 1, "imag": 2}',
...     object_hook=as_complex)
(1+2j)
>>> import decimal
>>> json.loads('1.1', parse_float=decimal.Decimal)
Decimal('1.1')
```

Extending `JSONEncoder`:

```
>>> import json
>>> class ComplexEncoder(json.JSONEncoder):
...     def default(self, obj):
...         if isinstance(obj, complex):
...             return [obj.real, obj.imag]
...         # Let the base class default method raise the
TypeError
...         return super().default(obj)
```

```
...
>>> json.dumps(2 + 1j, cls=ComplexEncoder)
'[2.0, 1.0]'
>>> ComplexEncoder().encode(2 + 1j)
'[2.0, 1.0]'
>>> list(ComplexEncoder().iterencode(2 + 1j))
['[2.0', ', 1.0', ']']
```

Using **`json`** from the shell to validate and pretty-print:

```
$ echo '{"json":"obj"}' | python -m json
{
    "json": "obj"
}
$ echo '{1.2:3.4}' | python -m json
Expecting property name enclosed in double quotes: line 1 column
2 (char 1)
```

See Command-line interface for detailed documentation.

Note

JSON is a subset of YAML [https://yaml.org/] 1.2. The JSON produced by this module's default settings (in particular, the default *separators* value) is also a subset of YAML 1.0 and 1.1. This module can thus also be used as a YAML serializer.

Note

This module's encoders and decoders preserve input and output order by default. Order is only lost if the underlying containers are unordered.

Basic Usage

json.dump(*obj*, *fp*, ***, *skipkeys=False*, *ensure_ascii=True*, *check_circular=True*, *allow_nan=True*, *cls=None*, *indent=None*, *separators=None*, *default=None*, *sort_keys=False*, ***kw*)

Serialize *obj* as a JSON formatted stream to *fp* (a `.write()`-supporting file-like object) using this Python-to-JSON conversion table.

Note

Unlike **`pickle`** and **`marshal`**, JSON is not a framed protocol, so trying to serialize multiple objects with repeated calls to **`dump()`** using the same *fp* will result in an invalid JSON file.

Parameters::

- **obj** (*object*) – The Python object to be serialized.
- **fp** (file-like object) – The file-like object *obj* will be serialized to. The **`json`** module always produces **`str`** objects, not **`bytes`** objects, therefore `fp.write()` must support **`str`** input.
- **skipkeys** (*bool*) – If `True`, keys that are not of a basic type (**`str`**, **`int`**, **`float`**, **`bool`**, `None`) will be skipped instead of raising a **`TypeError`**. Default `False`.
- **ensure_ascii** (*bool*) – If `True` (the default), the output is guaranteed to have all incoming non-ASCII and non-printable characters escaped. If `False`, al characters will be outputted as-is, except for the characters that must be escaped: quotation mark, reverse solidus, and the control characters U+0000 through U+001F.
- **check_circular** (*bool*) – If `False`, the circular reference check for container types is skipped and a circular reference will result in a **`RecursionError`** (or worse). Default `True`.
- **allow_nan** (*bool*) – If `False`, serialization of out-of-range **`float`** values (`nan`, `inf`, `-inf`) will result in a **`ValueError`**, in strict compliance with the JSON specification. If `True` (the default), their JavaScript equivalents (`NaN`, `Infinity`, `-Infinity`) are used.
- **cls** (a **`JSONEncoder`** subclass) – If set, a custom JSON encoder with the **`default()`** method overridden, for serializing into custom datatypes. If `None` (the default), **`JSONEncoder`** is used.

- **indent** (*int* | *str* | *None*) – If a positive integer or string, JSON array elements and object members will be pretty-printed with that indent level. A positive integer indents that many spaces per level; a string (such as `"\t"`) is used to indent each level. If zero, negative, or `""` (the empty string), only newlines are inserted. If `None` (the default), the most compact representation is used.
- **separators** (*tuple* | *None*) – A two-tuple `(item_separator, key_separator)`. If `None` (the default), *separators* defaults to `(', ', ': ')` if *indent* is `None`, and `(',', ': ')` otherwise. For the most compact JSON, specify `(',', ':')` to eliminate whitespace.
- **default** (*callable* | *None*) – A function that is called for objects that can't otherwise be serialized. It should return a JSON encodable version of the object or raise a `TypeError`. If `None` (the default), `TypeError` is raised.
- **sort_keys** (*bool*) – If `True`, dictionaries will be outputted sorted by key. Default `False`.

Changed in version 3.2: Allow strings for *indent* in addition to integers.

Changed in version 3.4: Use `(',', ': ')` as default if *indent* is not `None`.

Changed in version 3.6: All optional parameters are now keyword-only.

json.dumps(*obj*, *, *skipkeys=False*, *ensure_ascii=True*, *check_circular=True*, *allow_nan=True*, *cls=None*, *indent=None*, *separators=None*, *default=None*, *sort_keys=False*, ***kw*)

Serialize *obj* to a JSON formatted `str` using this conversion table. The arguments have the same meaning as in `dump()`.

Note

Keys in key/value pairs of JSON are always of the type `str`. When a dictionary is converted into JSON, all the keys of the dictionary are coerced to strings. As a result of this, if a dictionary is converted into JSON and then back into a dictionary, the dictionary may not equal the original one. That is, `loads(dumps(x)) != x` if x has non-string keys.

json.load(*fp*, *, *cls=None*, *object_hook=None*, *parse_float=None*, *parse_int=None*, *parse_constant=None*, *object_pairs_hook=None*, ***kw*)

Deserialize *fp* to a Python object using the JSON-to-Python conversion table.

Parameters::

- **fp** (file-like object) – A `.read()`-supportin text file or binary file containing the JSON document to be deserialized.
- **cls** (a **`JSONDecoder`** subclass) – If set, a custom JSON decoder. Additional keyword arguments to **`load()`** will be passed to the constructor of *cls*. If `None` (the default), **`JSONDecoder`** is used.
- **object_hook** (callable | None) – If set, a function that is called with the result of any JSON object literal decoded (a **`dict`**). The return value of this function will be used instead of the **`dict`**. This feature ca be used to implement custom decoders for example JSON-RPC [https://www.jsonrpc.org] class hinting. Defaul `None`.
- **object_pairs_hook** (callable | None) – I set, a function that is called with the result of any JSON object literal decode with an ordered list of pairs. The return value of this function will be used instead of the **`dict`**. This feature can be used to implement custom decoders. If *object_hook* is also set, *object_pairs_hook* takes priority. Default `None`.
- **parse_float** (callable | None) – If set, a function that is called with the string of every JSON float to be decoded. If `None` (the default), it is equivalent to `float(num_str)`. This can be used to parse JSON floats into custom datatypes, for example **`decimal.Decimal`**.
- **parse_int** (callable | None) – If set, a function that is called with the string of every JSON int to be decoded. If `None` (th

default), it is equivalent to `int(num_str)`. This can be used to parse JSON integers into custom datatypes, for example [`float`](#).

- **parse_constant** ([callable](#) | None) – If set, a function that is called with one of the following strings: `'-Infinity'`, `'Infinity'`, or `'NaN'`. This can be used to raise an exception if invalid JSON numbers are encountered. Default `None`.

Raises::

- **JSONDecodeError** – When the data being deserialized is not a valid JSON document.
- **UnicodeDecodeError** – When the data being deserialized does not contain UTF-8, UTF-16 or UTF-32 encoded data.

Changed in version 3.1:

- Added the optional *object_pairs_hook* parameter.
- *parse_constant* doesn't get called on 'null', 'true', 'false' anymore.

Changed in version 3.6:

- All optional parameters are now keyword-only.
- *fp* can now be a binary file. The input encoding should be UTF-8, UTF-16 or UTF-32.

Changed in version 3.11: The default *parse_int* of `int()` now limits the maximum length of the integer string via the interpreter's integer string conversion length limitation to help avoid denial of service attacks.

json.loads(*s*, *, *cls=None*, *object_hook=None*, *parse_float=None*, *parse_int=None*, *parse_constant=None*, *object_pairs_hook=None*, ***kw*)

Identical to `load()`, but instead of a file-like object, deserialize *s* (a `str`, `bytes` or `bytearray` instance containing a JSON document) to a Python object using this conversion table.

Changed in version 3.6: *s* can now be of type `bytes` or `bytearray`. The input encoding should be UTF-8, UTF-16 or UTF-32.

Changed in version 3.9: The keyword argument *encoding* has been removed.

Encoders and Decoders

class json.JSONDecoder(***, *object_hook=None*, *parse_float=None*, *parse_int=None*, *parse_constant=None*, *strict=True*, *object_pairs_hook=None*)

Simple JSON decoder.

Performs the following translations in decoding by default:

JSON	Python
object	dict
array	list
string	str
number (int)	int
number (real)	float
true	True
false	False
null	None

It also understands `NaN`, `Infinity`, and `-Infinity` as their corresponding `float` values, which is outside the JSON spec.

object_hook is an optional function that will be called with the result of every JSON object decoded and its return value will be used in place of the given **`dict`**. This can be used to provide custom deserializations (e.g. to support JSON-RPC [https://www.jsonrpc.org] class hinting).

object_pairs_hook is an optional function that will be called with the result of every JSON object decoded with an ordered list of pairs. The return value of *object_pairs_hook* will be used instead of the `dict`. This feature can be used to implement custom decoders. If *object_hook* is also defined, the *object_pairs_hook* takes priority.

Changed in version 3.1: Added support for *object_pairs_hook*.

parse_float is an optional function that will be called with the string of every JSON float to be decoded. By default, this is equivalent to `float(num_str)`. This can be used to use another datatype or parser for JSON floats (e.g. `decimal.Decimal`).

parse_int is an optional function that will be called with the string of every JSON int to be decoded. By default, this is equivalent to `int(num_str)`. This can be used to use another datatype or parser for JSON integers (e.g. `float`).

parse_constant is an optional function that will be called with one of the following strings: `'-Infinity'`, `'Infinity'`, `'NaN'`. This can be used to raise an exception if invalid JSON numbers are encountered.

If *strict* is false (`True` is the default), then control characters will be allowed inside strings. Control characters in this context are those with character codes in the 0–31 range, including `'\t'` (tab), `'\n'`, `'\r'` and `'\0'`.

If the data being deserialized is not a valid JSON document, a `JSONDecodeError` will be raised.

Changed in version 3.6: All parameters are now keyword-only.

decode(*s*)

Return the Python representation of *s* (a `str` instance containing a JSON document).

`JSONDecodeError` will be raised if the given JSON document is not valid.

raw_decode(*s*)

Decode a JSON document from *s* (a `str` beginning with a JSON document) and return a 2-tuple of the Python representation and the index in *s* where the document ended.

This can be used to decode a JSON document from a string that may have extraneous data at the end.

class json.JSONEncoder(***, *skipkeys=False*, *ensure_ascii=True*, *check_circular=True*, *allow_nan=True*, *sort_keys=False*, *indent=None*, *separators=None*, *default=None*)

Extensible JSON encoder for Python data structures.

Supports the following objects and types by default:

Python	**JSON**
dict	object
list, tuple	array
str	string
int, float, int- & float-derived Enums	number
True	true
False	false

Python	**JSON**
None	null

Changed in version 3.4: Added support for int- and float-derived Enum classes.

To extend this to recognize other objects, subclass and implement a `default()` method with another method that returns a serializable object for `o` if possible, otherwise it should call the superclass implementation (to raise `TypeError`).

If *skipkeys* is false (the default), a `TypeError` will be raised when trying to encode keys that are not `str`, `int`, `float`, `bool` or `None`. If *skipkeys* is true, such items are simply skipped.

If *ensure_ascii* is true (the default), the output is guaranteed to have all incoming non-ASCII and non-printable characters escaped. If *ensure_ascii* is false, all characters will be output as-is, except for the characters that must be escaped: quotation mark, reverse solidus, and the control characters U+0000 through U+001F.

If *check_circular* is true (the default), then lists, dicts, and custom encoded objects will be checked for circular references during encoding to prevent an infinite recursion (which would cause a `RecursionError`). Otherwise, no such check takes place.

If *allow_nan* is true (the default), then `NaN`, `Infinity`, and `-Infinity` will be encoded as such. This behavior is not JSON specification compliant, but is consistent with most JavaScript based encoders and decoders. Otherwise, it will be a `ValueError` to encode such floats.

If *sort_keys* is true (default: `False`), then the output of dictionaries will be sorted by key; this is useful for regression tests to ensure that JSON serializations can be compared on a day-to-day basis.

If *indent* is a non-negative integer or string, then JSON array elements and object members will be pretty-printed with that indent level. An indent level of 0, negative, or `""` will only insert newlines. `None` (the default) selects the most compact representation. Using a positive integer indent indents that many spaces per level. If *indent* is a string (such as `"\t"`), that string is used to indent each level.

Changed in version 3.2: Allow strings for *indent* in addition to integers.

If specified, *separators* should be an `(item_separator, key_separator)` tuple. The default is `(', ', ': ')` if *indent* is `None` and `(',', ': ')` otherwise. To get the most compact JSON representation, you should specify `(',', ':')` to eliminate whitespace.

Changed in version 3.4: Use `(',', ': ')` as default if *indent* is not `None`.

If specified, *default* should be a function that gets called for objects that can't otherwise be serialized. It should return a JSON encodable version of the object or raise a **`TypeError`**. If not specified, **`TypeError`** is raised.

Changed in version 3.6: All parameters are now keyword-only.

default(*o*)

Implement this method in a subclass such that it returns a serializable object for *o*, or calls the base implementation (to raise a [`TypeError`]()).

For example, to support arbitrary iterators, you could implement [`default()`]() like this:

```
def default(self, o):
   try:
       iterable = iter(o)
   except TypeError:
       pass
   else:
       return list(iterable)
   # Let the base class default method raise the TypeError
   return super().default(o)
```

encode(*o*)

Return a JSON string representation of a Python data structure, *o*. For example:

```
>>> json.JSONEncoder().encode({"foo": ["bar", "baz"]})
'{"foo": ["bar", "baz"]}'
```

iterencode(*o*)

Encode the given object, *o*, and yield each string representation as available. For example:

```
for chunk in json.JSONEncoder().iterencode(bigobject):
    mysocket.write(chunk)
```

Exceptions

exception json.JSONDecodeError(*msg*, *doc*, *pos*)

Subclass of [`ValueError`]() with the following additional attributes:

msg

The unformatted error message.

doc

The JSON document being parsed.

pos

The start index of *doc* where parsing failed.

lineno

The line corresponding to *pos*.

colno

The column corresponding to *pos*.

Added in version 3.5.

Standard Compliance and Interoperability

The JSON format is specified by **RFC 7159** [https://datatracker.ietf.org/doc/html/rfc7159.html] and by ECMA-404 [https://ecma-international.org/publications-and-standards/standards/ecma-404/]. This section details this module's level of compliance with the RFC. For simplicity, **`JSONEncoder`** and **`JSONDecoder`** subclasses, and parameters other than those explicitly mentioned, are not considered.

This module does not comply with the RFC in a strict fashion, implementing some extensions that are valid JavaScript but not valid JSON. In particular:

- Infinite and NaN number values are accepted and output;
- Repeated names within an object are accepted, and only the value of the last name-value pair is used.

Since the RFC permits RFC-compliant parsers to accept input texts that are not RFC-compliant, this module's deserializer is technically RFC-compliant under default settings.

Character Encodings

The RFC requires that JSON be represented using either UTF-8, UTF-16, or UTF-32, with UTF-8 being the recommended default for maximum interoperability.

As permitted, though not required, by the RFC, this module's serializer sets *ensure_ascii=True* by default, thus escaping the output so that the resulting strings only contain printable ASCII characters.

Other than the *ensure_ascii* parameter, this module is defined strictly in terms of conversion between Python objects and `Unicode strings`, and thus does not otherwise directly address the issue of character encodings.

The RFC prohibits adding a byte order mark (BOM) to the start of a JSON text, and this module's serializer does not add a BOM to its output. The RFC permits, but does not require, JSON deserializers to ignore an initial BOM in their input. This module's deserializer raises a `ValueError` when an initial BOM is present.

The RFC does not explicitly forbid JSON strings which contain byte sequences that don't correspond to valid Unicode characters (e.g. unpaired UTF-16 surrogates), but it does note that they may cause interoperability problems. By default, this module accepts and outputs (when present in the original `str`) code points for such sequences.

Infinite and NaN Number Values

The RFC does not permit the representation of infinite or NaN number values. Despite that, by default, this module accepts and outputs `Infinity`, `-Infinity`, and `NaN` as if they were valid JSON number literal values:

```
>>> # Neither of these calls raises an exception, but the
results are not valid JSON
>>> json.dumps(float('-inf'))
'-Infinity'
>>> json.dumps(float('nan'))
'NaN'
>>> # Same when deserializing
>>> json.loads('-Infinity')
-inf
>>> json.loads('NaN')
nan
```

In the serializer, the *allow_nan* parameter can be used to alter this behavior. In the deserializer, the *parse_constant* parameter can be used to alter this behavior.

Repeated Names Within an Object

The RFC specifies that the names within a JSON object should be unique, but does not mandate how repeated names in JSON objects should be handled. By default, this module does not raise an exception; instead, it ignores all but the last name-value pair for a given name:

```
>>> weird_json = '{"x": 1, "x": 2, "x": 3}'
>>> json.loads(weird_json)
{'x': 3}
```

The *object_pairs_hook* parameter can be used to alter this behavior.

Top-level Non-Object, Non-Array Values

The old version of JSON specified by the obsolete **RFC 4627** [https://datatracker.ietf.org/doc/html/rfc4627.html] required that the top-level value of a JSON text must be either a JSON object or array (Python `dict` or `list`), and could not be a JSON null, boolean, number, or string value. **RFC 7159** [https://datatracker.ietf.org/doc/html/rfc7159.html] removed that restriction, and this module does not and has never implemented that restriction in either its serializer or its deserializer.

Regardless, for maximum interoperability, you may wish to voluntarily adhere to the restriction yourself.

Implementation Limitations

Some JSON deserializer implementations may set limits on:

- the size of accepted JSON texts
- the maximum level of nesting of JSON objects and arrays
- the range and precision of JSON numbers
- the content and maximum length of JSON strings

This module does not impose any such limits beyond those of the relevant Python datatypes themselves or the Python interpreter itself.

When serializing to JSON, beware any such limitations in applications that may consume your JSON. In particular, it is common for JSON numbers to be deserialized into IEEE 754 double precision numbers and thus subject to that representation's range and precision limitations. This is especially relevant when serializing Python **`int`** values of extremely large magnitude, or when serializing instances of "exotic" numerical types such as **`decimal.Decimal`**.

Command-line interface

Source code: Lib/json/tool.py
[https://github.com/python/cpython/tree/3.14/Lib/json/tool.py]

The **`json`** module can be invoked as a script via `python -m json` to validate and pretty-print JSON objects. The **`json.tool`** submodule implements this interface.

If the optional `infile` and `outfile` arguments are not specified, **`sys.stdin`** and **`sys.stdout`** will be used respectively:

```
$ echo '{"json": "obj"}' | python -m json
{
    "json": "obj"
}
$ echo '{1.2:3.4}' | python -m json
Expecting property name enclosed in double quotes: line 1 column
2 (char 1)
```

Changed in version 3.5: The output is now in the same order as the input. Use the **`--sort-keys`** option to sort the output of dictionaries alphabetically by key.

Changed in version 3.14: The **`json`** module may now be directly executed as `python -m json`. For backwards compatibility, invoking the CLI as `python -m json.tool` remains supported.

Command-line options

infile

The JSON file to be validated or pretty-printed:

```
$ python -m json mp_films.json
[
    {
        "title": "And Now for Something Completely
Different",
        "year": 1971
    },
    {
        "title": "Monty Python and the Holy Grail",
        "year": 1975
    }
]
```

If *infile* is not specified, read from `sys.stdin`.

outfile
Write the output of the *infile* to the given *outfile*. Otherwise, write it to `sys.stdout`.

--sort-keys
Sort the output of dictionaries alphabetically by key.

Added in version 3.5.

--no-ensure-ascii
Disable escaping of non-ascii characters, see `json.dumps()` for more information.

Added in version 3.9.

--json-lines
Parse every input line as separate JSON object.

Added in version 3.8.

--indent, --tab, --no-indent, --compact
Mutually exclusive options for whitespace control.

Added in version 3.9.

-h, --help
Show the help message.

Footnotes

[1]

As noted in the errata for RFC 7159 [https://www.rfc-editor.org/errata_search.php?rfc=7159], JSON permits literal U+2028 (LINE SEPARATOR) and U+2029 (PARAGRAPH SEPARATOR) characters in strings, whereas JavaScript (as of ECMAScript Edition 5.1) does not.

`mailbox` — Manipulate mailboxes in various formats

Source code: Lib/mailbox.py
[https://github.com/python/cpython/tree/3.14/Lib/mailbox.py]

This module defines two classes, `Mailbox` and `Message`, for accessing and manipulating on-disk mailboxes and the messages they contain. `Mailbox` offers a dictionary-like mapping from keys to messages. `Message` extends the `email.message` module's `Message` class with format-specific state and behavior. Supported mailbox formats are Maildir, mbox, MH, Babyl, and MMDF.

See also

Module `email`
Represent and manipulate messages.

`Mailbox` objects

class mailbox.Mailbox

A mailbox, which may be inspected and modified.

The `Mailbox` class defines an interface and is not intended to be instantiated. Instead, format-specific subclasses should inherit from `Mailbox` and your code should instantiate a particular subclass.

The `Mailbox` interface is dictionary-like, with small keys corresponding to messages. Keys are issued by the `Mailbox` instance with which they will be used and are only meaningful to that `Mailbox` instance. A key continues to identify a message even if the corresponding message is modified, such as by replacing it with another message.

Messages may be added to a `Mailbox` instance using the set-like method `add()` and removed using a `del` statement or the set-like methods `remove()` and `discard()`.

`Mailbox` interface semantics differ from dictionary semantics in some noteworthy ways. Each time a message is requested, a new representation (typically a `Message` instance) is generated based upon the current state of the mailbox. Similarly, when a message is added to a `Mailbox` instance, the provided message representation's contents are copied. In neither case is a reference to the message representation kept by the `Mailbox` instance.

The default `Mailbox` iterator iterates over message representations, not keys as the default `dictionary` iterator does. Moreover, modification of a mailbox during iteration is safe and well-defined. Messages added to the mailbox after an iterator is created will not be seen by the iterator. Messages removed from the mailbox before the iterator yields them will be silently skipped, though using a key from an iterator may result in a `KeyError` exception if the corresponding message is subsequently removed.

Warning

Be very cautious when modifying mailboxes that might be simultaneously changed by some other process. The safest mailbox format to use for such tasks is `Maildir`;

try to avoid using single-file formats such as `mbox` for concurrent writing. If you're modifying a mailbox, you *must* lock it by calling the `lock()` and `unlock()` methods *before* reading any messages in the file or making any changes by adding or deleting a message. Failing to lock the mailbox runs the risk of losing messages or corrupting the entire mailbox.

`Mailbox` instances have the following methods:

add(*message*)

Add *message* to the mailbox and return the key that has been assigned to it.

Parameter *message* may be a `Message` instance, an `email.message.Message` instance, a string, a byte string, or a file-like object (which should be open in binary mode). If *message* is an instance of the appropriate format-specific `Message` subclass (e.g., if it's an `mboxMessage` instance and this is an `mbox` instance), its format-specific information is used. Otherwise, reasonable defaults for format-specific information are used.

Changed in version 3.2: Support for binary input was added.

remove(*key*)
__delitem__(*key*)
discard(*key*)

Delete the message corresponding to *key* from the mailbox.

If no such message exists, a `KeyError` exception is raised if the method was called as `remove()` or `__delitem__()` but no exception is raised if the method was called as `discard()`. The behavior of `discard()` may be preferred if the underlying mailbox format supports concurrent modification by other processes.

__setitem__(*key*, *message*)

Replace the message corresponding to *key* with *message*. Raise a `KeyError` exception if no message already corresponds to *key*.

As with `add()`, parameter *message* may be a `Message` instance, an `email.message.Message` instance, a string, a byte string, or a file-like object (which should be open in binary mode). If *message* is an instance of the appropriate format-specific `Message` subclass (e.g., if it's an `mboxMessage` instance and this is an `mbox` instance), its format-specific information is used. Otherwise, the format-specific information of the message that currently corresponds to *key* is left unchanged.

iterkeys()

Return an iterator over all keys

keys()

The same as `iterkeys()`, except that a `list` is returned rather than an iterator

itervalues()
__iter__()

Return an iterator over representations of all messages. The messages are represented as instances of the appropriate format-specific `Message` subclass unless a custom message factory was specified when the **`Mailbox`** instance was initialized.

Note

The behavior of `__iter__()` is unlike that of dictionaries, which iterate over keys.

values()

The same as `itervalues()`, except that a `list` is returned rather than an iterator

iteritems()

Return an iterator over (*key*, *message*) pairs, where *key* is a key and *message* is a message representation. The messages are represented as instances of the appropriate format-specific `Message` subclass unless a custom message factory was specified when the **`Mailbox`** instance was initialized.

items()

The same as `iteritems()`, except that a `list` of pairs is returned rather than an iterator of pairs.

get(*key*, *default=None*)
__getitem__(*key*)

Return a representation of the message corresponding to *key*. If no such message exists, *default* is returned if the method was called as `get()` and a `KeyError` exception is raised if the method was called as **`__getitem__()`**. The message is represented as an instance of the appropriate format-specific `Message` subclass unless a custom message factory was specified when the **`Mailbox`** instance was initialized.

get_message(*key*)

Return a representation of the message corresponding to *key* as an instance of the appropriate format-specific **`Message`** subclass, or raise a **`KeyError`** exception if no such message exists.

get_bytes(*key*)

Return a byte representation of the message corresponding to *key*, or raise a **`KeyError`** exception if no such message exists.

Added in version 3.2.

get_string(*key*)

Return a string representation of the message corresponding to *key*, or raise a **`KeyError`** exception if no such message exists. The message is processed through **`email.message.Message`** to convert it to a 7bit clean representation.

get_file(*key*)

Return a file-like representation of the message corresponding to *key*, or raise a **`KeyError`** exception if no such message exists. The file-like object behaves as if open in binary mode. This file should be closed once it is no longer needed.

Changed in version 3.2: The file object really is a binary file; previously it was incorrectly returned in text mode. Also, the file-like object now supports the context manager protocol: you can use a **`with`** statement to automatically close it.

Note

Unlike other representations of messages, file-like representations are not necessarily independent of the **`Mailbox`** instance that created them or of the underlying mailbox. More specific documentation is provided by each subclass.

__contains__(*key*)

Return `True` if *key* corresponds to a message, `False` otherwise.

__len__()

Return a count of messages in the mailbox.

clear()

Delete all messages from the mailbox.

pop(*key*, *default=None*)

Return a representation of the message corresponding to *key* and delete the message. If no such message exists, return *default*. The message is represented as an instance of the appropriate format-specific `Message` subclass unless a custom message factory was specified when the **`Mailbox`** instance was initialized.

popitem()

Return an arbitrary (*key*, *message*) pair, where *key* is a key and *message* is a message representation, and delete the corresponding message. If the mailbox is empty, raise a `KeyError` exception. The message is represented as an instance of the appropriate format-specific `Message` subclass unless a custom message factory was specified when the **`Mailbox`** instance was initialized.

update(*arg*)

Parameter *arg* should be a *key*-to-*message* mapping or an iterable of (*key*, *message*) pairs. Updates the mailbox so that, for each given *key* and *message*, the message corresponding to *key* is set to *message* as if by using `__setitem__()`. As with `__setitem__()`, each *key* must already correspond to a message in the mailbox or else a `KeyError` exception will be raised, so in general it is incorrect for *arg* to be a **`Mailbox`** instance.

Note

Unlike with dictionaries, keyword arguments are not supported.

flush()

Write any pending changes to the filesystem. For some `Mailbox` subclasses, changes are always written immediately and **`flush()`** does nothing, but you should still make a habit of calling this method.

lock()

Acquire an exclusive advisory lock on the mailbox so that other processes know not to modify it. An `ExternalClashError` is raised if the lock is not available. The particular locking mechanisms used depend upon the mailbox format. You should *always* lock the mailbox before making any modifications to its contents.

unlock()

Release the lock on the mailbox, if any.

close()

Flush the mailbox, unlock it if necessary, and close any open files. For some **`Mailbox`** subclasses, this method does nothing.

`Maildir` objects

class mailbox.Maildir(*dirname*, *factory=None*, *create=True*)

A subclass of [`Mailbox`] for mailboxes in Maildir format. Parameter *factory* is a callable object that accepts a file-like message representation (which behaves as if opened in binary mode) and returns a custom representation. If *factory* is `None`, [`MaildirMessage`] is used as the default message representation. If *create* is `True`, the mailbox is created if it does not exist.

If *create* is `True` and the *dirname* path exists, it will be treated as an existing maildir without attempting to verify its directory layout.

It is for historical reasons that *dirname* is named as such rather than *path*.

Maildir is a directory-based mailbox format invented for the qmail mail transfer agent and now widely supported by other programs. Messages in a Maildir mailbox are stored in separate files within a common directory structure. This design allows Maildir mailboxes to be accessed and modified by multiple unrelated programs without data corruption, so file locking is unnecessary.

Maildir mailboxes contain three subdirectories, namely: `tmp`, `new`, and `cur`. Messages are created momentarily in the `tmp` subdirectory and then moved to the `new` subdirectory to finalize delivery. A mail user agent may subsequently move the message to the `cur` subdirectory and store information about the state of the message in a special “info” section appended to its file name.

Folders of the style introduced by the Courier mail transfer agent are also supported. Any subdirectory of the main mailbox is considered a folder if `'.'` is the first character in its name. Folder names are represented by **`Maildir`** without the leading `'.'`. Each folder is itself a Maildir mailbox but should not contain other folders. Instead, a logical nesting is indicated using `'.'` to delimit levels, e.g., “Archived.2005.07”.

colon

The Maildir specification requires the use of a colon (`':'`) in certain message file names. However, some operating systems do not permit this character in file names, If you wish to use a Maildir-like format on such an operating system, you should specify another character to use instead. The exclamation point (`'!'`) is a popular choice. For example:

```
import mailbox
mailbox.Maildir.colon = '!'
```

The **`colon`** attribute may also be set on a per-instance basis.

Changed in version 3.13: `Maildir` now ignores files with a leading dot.

`Maildir` instances have all of the methods of `Mailbox` in addition to the following:

list_folders()

Return a list of the names of all folders.

get_folder(*folder*)

Return a **`Maildir`** instance representing the folder whose name is *folder*. A `NoSuchMailboxError` exception is raised if the folder does not exist.

add_folder(*folder*)

Create a folder whose name is *folder* and return a **`Maildir`** instance representing it.

remove_folder(*folder*)

Delete the folder whose name is *folder*. If the folder contains any messages, a **`NotEmptyError`** exception will be raised and the folder will not be deleted.

clean()

Delete temporary files from the mailbox that have not been accessed in the last 36 hours. The Maildir specification says that mail-reading programs should do this occasionally.

get_flags(*key*)

Return as a string the flags that are set on the message corresponding to *key*. This is the same as `get_message(key).get_flags()` but much faster, because it does not open the message file. Use this method when iterating over the keys to determine which messages are interesting to get.

If you do have a **`MaildirMessage`** object, use its **`get_flags()`** method instead, because changes made by the message's **`set_flags()`**, **`add_flag()`** and **`remove_flag()`** methods are not reflected here until the mailbox's **`__setitem__()`** method is called.

Added in version 3.13.

set_flags(*key*, *flags*)

On the message corresponding to *key*, set the flags specified by *flags* and unset all others. Calling `some_mailbox.set_flags(key, flags)` is similar to

```
one_message = some_mailbox.get_message(key)
one_message.set_flags(flags)
some_mailbox[key] = one_message
```

but faster, because it does not open the message file.

If you do have a **`MaildirMessage`** object, use its **`set_flags()`** method instead, because changes made with this mailbox method will not be visible to the message object's method, **`get_flags()`**.

Added in version 3.13.

add_flag(*key*, *flag*)

On the message corresponding to *key*, set the flags specified by *flag* without changing other flags. To add more than one flag at a time, *flag* may be a string of more than one character.

Considerations for using this method versus the message object's **`add_flag()`** method are similar to those for **`set_flags()`**; see the discussion there.

Added in version 3.13.

remove_flag(*key*, *flag*)

On the message corresponding to *key*, unset the flags specified by *flag* without changing other flags. To remove more than one flag at a time, *flag* may be a string of more than one character.

Considerations for using this method versus the message object's **`remove_flag()`** method are similar to those for **`set_flags()`**; see the discussion there.

Added in version 3.13.

get_info(*key*)

Return a string containing the info for the message corresponding to *key*. This is the same as `get_message(key).get_info()` but much faster, because it does not open the message file. Use this method when iterating over the keys to determine which messages are interesting to get.

If you do have a `MaildirMessage` object, use its `get_info()` method instead, because changes made by the message's `set_info()` method are not reflected here until the mailbox's `__setitem__()` method is called.

Added in version 3.13.

set_info(*key*, *info*)

Set the info of the message corresponding to *key* to *info*. Calling `some_mailbox.set_info(key, flags)` is similar to

```
one_message = some_mailbox.get_message(key)
one_message.set_info(info)
some_mailbox[key] = one_message
```

but faster, because it does not open the message file.

If you do have a `MaildirMessage` object, use its `set_info()` method instead, because changes made with this mailbox method will not be visible to the message object's method, `get_info()`.

Added in version 3.13.

Some `Mailbox` methods implemented by **`Maildir`** deserve special remarks:

add(*message*)

__setitem__(*key*, *message*)
update(*arg*)

Warning

These methods generate unique file names based upon the current process ID. When using multiple threads, undetected name clashes may occur and cause corruption of the mailbox unless threads are coordinated to avoid using these methods to manipulate the same mailbox simultaneously.

flush()

All changes to Maildir mailboxes are immediately applied, so this method does nothing.

lock()
unlock()

Maildir mailboxes do not support (or require) locking, so these methods do nothing.

close()

`Maildir` instances do not keep any open files and the underlying mailboxes do not support locking, so this method does nothing.

get_file(*key*)

Depending upon the host platform, it may not be possible to modify or remove the underlying message while the returned file remains open.

See also

maildir man page from Courier **[https://www.courier-mta.org/maildir.html]**

A specification of the format. Describes a common extension for supporting folders.

Using maildir format [https://cr.yp.to/proto/maildir.html]
Notes on Maildir by its inventor. Includes an updated name-creation scheme and details on “info” semantics.

`mbox` objects

class mailbox.mbox(*path*, *factory=None*, *create=True*)

A subclass of `Mailbox` for mailboxes in mbox format. Parameter *factory* is a callable object that accepts a file-like message representation (which behaves as if opened in binary mode) and returns a custom representation. If *factory* is `None`, `mboxMessage` is used as the default message representation. If *create* is `True`, the mailbox is created if it does not exist.

The mbox format is the classic format for storing mail on Unix systems. All messages in an mbox mailbox are stored in a single file with the beginning of each message indicated by a line whose first five characters are “From “.

Several variations of the mbox format exist to address perceived shortcomings in the original. In the interest of compatibility, **`mbox`** implements the original format, which is sometimes referred to as *mboxo*. This means that the *Content-Length* header, if present, is ignored and that any occurrences of “From “ at the beginning of a line in a message body are transformed to “>From “ when storing the message, although occurrences of “>From “ are not transformed to “From “ when reading the message.

Some `Mailbox` methods implemented by **`mbox`** deserve special remarks:

get_bytes(*key*, *from_=False*)

Note: This method has an extra parameter (*from_*) compared with other classes. The first line of an mbox file entry is the Unix “From “ line. If *from_* is False, the first line of the file is dropped.

get_file(*key*, *from_=False*)

Using the file after calling `flush()` or `close()` on the `mbox` instance may yield unpredictable results or raise an exception.

Note: This method has an extra parameter (*from_*) compared with other classes. The first line of an mbox file entry is the Unix “From “ line. If *from_* is False, the first line of the file is dropped.

get_string(*key*, *from_=False*)

Note: This method has an extra parameter (*from_*) compared with other classes. The first line of an mbox file entry is the Unix “From “ line. If *from_* is False, the first line of the file is dropped.

lock()
unlock()

Three locking mechanisms are used—dot locking and, if available, the `flock()` and `lockf()` system calls.

See also

mbox man page from tin [http://www.tin.org/bin/man.cgi?section=5&topic=mbox]

A specification of the format, with details on locking.

Configuring Netscape Mail on Unix: Why The Content-Length Format is Bad

[https://www.jwz.org/doc/content-length.html]

An argument for using the original mbox format rather than a variation.

“mbox” is a family of several mutually incompatible mailbox formats
[https://www.loc.gov/preservation/digital/formats/fdd/fdd000383.shtml]

A history of mbox variations.

`MH` objects

class mailbox.MH(*path*, *factory=None*, *create=True*)

A subclass of `Mailbox` for mailboxes in MH format. Parameter *factory* is a callable object that accepts a file-like message representation (which behaves as if opened in binary mode) and returns a custom representation. If *factory* is `None`, `MHMessage` is used as the default message representation. If *create* is `True`, the mailbox is created if it does not exist.

MH is a directory-based mailbox format invented for the MH Message Handling System, a mail user agent. Each message in an MH mailbox resides in its own file. An MH mailbox may contain other MH mailboxes (called *folders*) in addition to messages. Folders may be nested indefinitely. MH mailboxes also support *sequences*, which are named lists used to logically group messages without moving them to sub-folders. Sequences are defined in a file called `.mh_sequences` in each folder.

The **`MH`** class manipulates MH mailboxes, but it does not attempt to emulate all of **mh**’s behaviors. In particular, it does not modify and is not affected by the `context` or

`.mh_profile` files that are used by **mh** to store its state and configuration.

`MH` instances have all of the methods of **`Mailbox`** in addition to the following:

Changed in version 3.13: Supported folders that don’t contain a `.mh_sequences` file.

list_folders()

Return a list of the names of all folders.

get_folder(*folder*)

Return an **`MH`** instance representing the folder whose name is *folder*. A **`NoSuchMailboxError`** exception is raised if the folder does not exist.

add_folder(*folder*)

Create a folder whose name is *folder* and return an **`MH`** instance representing it.

remove_folder(*folder*)

Delete the folder whose name is *folder*. If the folder contains any messages, a **`NotEmptyError`** exception will be raised and the folder will not be deleted.

get_sequences()

Return a dictionary of sequence names mapped to key lists. If there are no sequences, the empty dictionary is returned.

set_sequences(*sequences*)

Re-define the sequences that exist in the mailbox based upon *sequences*, a dictionary of names mapped to key lists, like returned by **`get_sequences()`**.

pack()

Rename messages in the mailbox as necessary to eliminate gaps in numbering. Entries in the sequences list are updated correspondingly.

Note

Already-issued keys are invalidated by this operation and should not be subsequently used.

Some `Mailbox` methods implemented by **`MH`** deserve special remarks:

remove(*key*)
__delitem__(*key*)
discard(*key*)

These methods immediately delete the message. The MH convention of marking a message for deletion by prepending a comma to its name is not used.

lock()
unlock()

Three locking mechanisms are used—dot locking and, if available, the **`flock()`** and **`lockf()`** system calls. For MH mailboxes, locking the mailbox means locking the `.mh_sequences` file and, only for the duration of any operations that affect them, locking individual message files.

get_file(*key*)

Depending upon the host platform, it may not be possible to remove the underlying message while the returned file remains open.

flush()

All changes to MH mailboxes are immediately applied, so this method does nothing.

close()

`MH` instances do not keep any open files, so this method is equivalent to `unlock()`.

See also

nmh - Message Handling System [https://www.nongnu.org/nmh/]

Home page of **nmh**, an updated version of the original **mh**.

MH & nmh: Email for Users & Programmers [https://rand-mh.sourceforge.io/book/]

A GPL-licensed book on **mh** and **nmh**, with some information on the mailbox format.

`Babyl` objects

class mailbox.Babyl(*path*, *factory=None*, *create=True*)

A subclass of `Mailbox` for mailboxes in Babyl format. Parameter *factory* is a callable object that accepts a file-like message representation (which behaves as if opened in binary mode) and returns a custom representation. If *factory* is `None`, `BabylMessage` is used as the default message representation. If *create* is `True`, the mailbox is created if it does not exist.

Babyl is a single-file mailbox format used by the Rmail mail user agent included with Emacs. The beginning of a message is indicated by a line containing the two characters Control-Underscore (`'\037'`) and Control-L (`'\014'`). The end of a message is indicated by the start of the next message or, in the case of the last message, a line containing a Control-Underscore (`'\037'`) character.

Messages in a Babyl mailbox have two sets of headers, original headers and so-called visible headers. Visible headers are typically a subset of the original headers that have been reformatted or abridged to be more attractive. Each message in a Babyl mailbox also has an accompanying list of *labels*, or short strings that record extra information about the message, and a list of all user-defined labels found in the mailbox is kept in the Babyl options section.

`Babyl` instances have all of the methods of [`Mailbox`] in addition to the following:

get_labels()

Return a list of the names of all user-defined labels used in the mailbox.

Note

The actual messages are inspected to determine which labels exist in the mailbox rather than consulting the list of labels in the Babyl options section, but the Babyl section is updated whenever the mailbox is modified.

Some [`Mailbox`] methods implemented by **`Babyl`** deserve special remarks:

get_file(*key*)

In Babyl mailboxes, the headers of a message are not stored contiguously with the body of the message. To generate a file-like representation, the headers and body are copied together into an [`io.BytesIO`] instance, which has an API identical to that of a file. As a result, the file-like object is truly independent of the underlying mailbox but does not save memory compared to a string representation.

lock()

unlock()

Three locking mechanisms are used—dot locking and, if available, the `flock()` and `lockf()` system calls.

See also

Format of Version 5 Babyl Files **[https://quimby.gnus.org/notes/BABYL]**

A specification of the Babyl format.

Reading Mail with Rmail **[https://www.gnu.org/software/emacs/manual/html_node/emacs/Rmail.html]**

The Rmail manual, with some information on Babyl semantics.

`MMDF` objects

class mailbox.MMDF(*path*, *factory=None*, *create=True*)

A subclass of `Mailbox` for mailboxes in MMDF format. Parameter *factory* is a callable object that accepts a file-like message representation (which behaves as if opened in binary mode) and returns a custom representation. If *factory* is `None`, `MMDFMessage` is used as the default message representation. If *create* is `True`, the mailbox is created if it does not exist.

MMDF is a single-file mailbox format invented for the Multichannel Memorandum Distribution Facility, a mail transfer agent. Each message is in the same form as an mbox message but is bracketed before and after by lines containing four Control-A (`'\001'`) characters. As with the mbox format, the beginning of each message is indicated by a line whose first five characters are “From “, but additional occurrences of “From “ are not transformed to “>From “ when storing messages because the extra message separator lines prevent mistaking such occurrences for the starts of subsequent messages.

Some **`Mailbox`** methods implemented by **`MMDF`** deserve special remarks:

get_bytes(*key*, *from_=False*)

Note: This method has an extra parameter (*from_*) compared with other classes. The first line of an mbox file entry is the Unix “From “ line. If *from_* is False, the first line of the file is dropped.

get_file(*key*, *from_=False*)

Using the file after calling **`flush()`** or **`close()`** on the **`MMDF`** instance may yield unpredictable results or raise an exception.

Note: This method has an extra parameter (*from_*) compared with other classes. The first line of an mbox file entry is the Unix “From “ line. If *from_* is False, the first line of the file is dropped.

lock()
unlock()

Three locking mechanisms are used—dot locking and, if available, the **`flock()`** and **`lockf()`** system calls.

See also

mmdf man page from tin **[http://www.tin.org/bin/man.cgi?section=5&topic=mmdf]**

A specification of MMDF format from the documentation of tin, a newsreader.

MMDF **[https://en.wikipedia.org/wiki/MMDF]**

A Wikipedia article describing the Multichannel Memorandum Distribution Facility.

`Message` objects

class mailbox.Message(*message=None*)

A subclass of the `email.message` module’s `Message`. Subclasses of **`mailbox.Message`** add mailbox-format-specific state and behavior.

If *message* is omitted, the new instance is created in a default, empty state. If *message* is an `email.message.Message` instance, its contents are copied; furthermore, any format-specific information is converted insofar as possible if *message* is a **`Message`** instance. If *message* is a string, a byte string, or a file, it should contain an **RFC 5322** [https://datatracker.ietf.org/doc/html/rfc5322.html]-compliant message, which is read and parsed. Files should be open in binary mode, but text mode files are accepted for backward compatibility.

The format-specific state and behaviors offered by subclasses vary, but in general it is only the properties that are not specific to a particular mailbox that are supported (although presumably the properties are specific to a particular mailbox format). For example, file offsets for single-file mailbox formats and file names for directory-based mailbox formats are not retained, because they are only applicable to the original mailbox. But state such as whether a message has been read by the user or marked as important is retained, because it applies to the message itself.

There is no requirement that **`Message`** instances be used to represent messages retrieved using `Mailbox` instances. In some situations, the time and memory required to generate **`Message`** representations might not be acceptable. For such situations, **`Mailbox`** instances also offer string and file-like representations, and a custom message factory may be specified when a **`Mailbox`** instance is initialized.

`MaildirMessage` objects

class mailbox.MaildirMessage(*message=None*)

A message with Maildir-specific behaviors. Parameter *message* has the same meaning as with the `Message` constructor.

Typically, a mail user agent application moves all of the messages in the `new` subdirectory to the `cur` subdirectory after the first time the user opens and closes the mailbox, recording that the messages are old whether or not they’ve actually been read. Each message in `cur` has an “info” section added to its file name to store information about its state. (Some mail readers may also add an “info” section to messages in `new`.) The “info” section may take one of two forms: it may contain “2,” followed by a list of standardized flags (e.g., “2,FR”) or it may contain “1,” followed by so-called experimental information. Standard flags for Maildir messages are as follows:

Flag	Meaning	Explanation
D	Draft	Under composition
F	Flagged	Marked as important
P	Passed	Forwarded, resent, or bounced
R	Replied	Replied to
S	Seen	Read
T	Trashed	Marked for subsequent deletion

`MaildirMessage` instances offer the following methods:

get_subdir()

Return either “new” (if the message should be stored in the `new` subdirectory) or “cur” (if the message should be stored in the `cur` subdirectory).

Note

A message is typically moved from `new` to `cur` after its mailbox has been accessed, whether or not the message has been read. A message `msg` has been read if `"S" in msg.get_flags()` is `True`.

set_subdir(*subdir*)

Set the subdirectory the message should be stored in. Parameter *subdir* must be either “new” or “cur”.

get_flags()

Return a string specifying the flags that are currently set. If the message complies with the standard Maildir format, the result is the concatenation in alphabetical order of zero or one occurrence of each of `'D'`, `'F'`, `'P'`, `'R'`, `'S'`, and `'T'`. The empty string is returned if no flags are set or if “info” contains experimental semantics.

set_flags(*flags*)

Set the flags specified by *flags* and unset all others.

add_flag(*flag*)

Set the flag(s) specified by *flag* without changing other flags. To add more than one flag at a time, *flag* may be a string of more than one character. The current “info” is overwritten whether or not it contains experimental information rather than flags.

remove_flag(*flag*)

Unset the flag(s) specified by *flag* without changing other flags. To remove more than one flag at a time, *flag* maybe a string of more than one character. If “info” contains experimental information rather than flags, the current “info” is not modified.

get_date()

Return the delivery date of the message as a floating-point number representing seconds since the epoch.

set_date(*date*)

Set the delivery date of the message to *date*, a floating-point number representing seconds since the epoch.

get_info()

Return a string containing the “info” for a message. This is useful for accessing and modifying “info” that is experimental (i.e., not a list of flags).

set_info(*info*)
Set “info” to *info*, which should be a string.

When a `MaildirMessage` instance is created based upon an `mboxMessage` or `MMDFMessage` instance, the *Status* and *X-Status* headers are omitted and the following conversions take place:

Resulting state	`mboxMessage` **or** `MMDFMessage` **state**
“cur” subdirectory	O flag
F flag	F flag
R flag	A flag
S flag	R flag
T flag	D flag

When a `MaildirMessage` instance is created based upon an `MHMessage` instance, the following conversions take place:

Resulting state	`MHMessage` **state**
“cur” subdirectory	“unseen” sequence

Resulting state	`MHMessage` state
"cur" subdirectory and S flag	no "unseen" sequence
F flag	"flagged" sequence
R flag	"replied" sequence

When a **`MaildirMessage`** instance is created based upon a `BabylMessage` instance, the following conversions take place:

Resulting state	`BabylMessage` state
"cur" subdirectory	"unseen" label
"cur" subdirectory and S flag	no "unseen" label
P flag	"forwarded" or "resent" label
R flag	"answered" label
T flag	"deleted" label

`mboxMessage` objects

class mailbox.mboxMessage(*message=None*)

A message with mbox-specific behaviors. Parameter *message* has the same meaning as with the `Message` constructor.

Messages in an mbox mailbox are stored together in a single file. The sender’s envelope address and the time of delivery are typically stored in a line beginning with “From “ that is used to indicate the start of a message, though there is considerable variation in the exact format of this data among mbox implementations. Flags that indicate the state of the message, such as whether it has been read or marked as important, are typically stored in *Status* and *X-Status* headers.

Conventional flags for mbox messages are as follows:

Flag	Meaning	Explanation
R	Read	Read
O	Old	Previously detected by MUA
D	Deleted	Marked for subsequent deletion
F	Flagged	Marked as important
A	Answered	Replied to

The “R” and “O” flags are stored in the *Status* header, and the “D”, “F”, and “A” flags are stored in the *X-Status* header. The flags and headers typically appear in the order mentioned.

`mboxMessage` instances offer the following methods:

get_from()

Return a string representing the “From “ line that marks the start of the message in an mbox mailbox. The leading “From “ and the trailing newline are excluded.

set_from(*from_*, *time_=None*)

Set the “From “ line to *from_*, which should be specified without a leading “From “ or trailing newline. For convenience, *time_* may be specified and will be formatted appropriately and appended to *from_*. If *time_* is specified, it should be a **`time.struct_time`** instance, a tuple suitable for passing to **`time.strftime()`**, or `True` (to use **`time.gmtime()`**).

get_flags()

Return a string specifying the flags that are currently set. If the message complies with the conventional format, the result is the concatenation in the following order of zero or one occurrence of each of `'R'`, `'O'`, `'D'`, `'F'`, and `'A'`.

set_flags(*flags*)

Set the flags specified by *flags* and unset all others. Parameter *flags* should be the concatenation in any order of zero or more occurrences of each of `'R'`, `'O'`, `'D'`, `'F'`, and `'A'`.

add_flag(*flag*)

Set the flag(s) specified by *flag* without changing other flags. To add more than one flag at a time, *flag* may be a string of more than one character.

remove_flag(*flag*)

Unset the flag(s) specified by *flag* without changing other flags. To remove more than one flag at a time, *flag* maybe a string of more than one character.

When an `mboxMessage` instance is created based upon a `MaildirMessage` instance, a “From “ line is generated based upon the `MaildirMessage` instance’s delivery date, and the following conversions take place:

Resulting state	`MaildirMessage` **state**
R flag	S flag
O flag	“cur” subdirectory
D flag	T flag
F flag	F flag
A flag	R flag

When an `mboxMessage` instance is created based upon an `MHMessage` instance, the following conversions take place:

Resulting state	`MHMessage` **state**
R flag and O flag	no “unseen” sequence
O flag	“unseen” sequence

Resulting state	`MHMessage` state
F flag	“flagged” sequence
A flag	“replied” sequence

When an **`mboxMessage`** instance is created based upon a `BabylMessage` instance, the following conversions take place:

Resulting state	`BabylMessage` state
R flag and O flag	no “unseen” label
O flag	“unseen” label
D flag	“deleted” label
A flag	“answered” label

When a **`mboxMessage`** instance is created based upon an `MMDFMessage` instance, the “From “ line is copied and all flags directly correspond:

Resulting state	`MMDFMessage` state
R flag	R flag

Resulting state	`MMDFMessage` state
O flag	O flag
D flag	D flag
F flag	F flag
A flag	A flag

`MHMessage` objects

class mailbox.MHMessage(*message=None*)

A message with MH-specific behaviors. Parameter *message* has the same meaning as with the `Message` constructor.

MH messages do not support marks or flags in the traditional sense, but they do support sequences, which are logical groupings of arbitrary messages. Some mail reading programs (although not the standard **mh** and **nmh**) use sequences in much the same way flags are used with other formats, as follows:

Sequence	Explanation
unseen	Not read, but previously detected by MUA
replied	Replied to

Sequence	Explanation
flagged	Marked as important

`MHMessage` instances offer the following methods:

get_sequences()
: Return a list of the names of sequences that include this message.

set_sequences(*sequences*)
: Set the list of sequences that include this message.

add_sequence(*sequence*)
: Add *sequence* to the list of sequences that include this message.

remove_sequence(*sequence*)
: Remove *sequence* from the list of sequences that include this message.

When an `MHMessage` instance is created based upon a `MaildirMessage` instance, the following conversions take place:

Resulting state	`MaildirMessage` state
“unseen” sequence	no S flag
“replied” sequence	R flag
“flagged” sequence	F flag

When an `MHMessage` instance is created based upon an `mboxMessage` or `MMDFMessage` instance, the *Status* and *X-Status* headers are omitted and the following conversions take place:

Resulting state	`mboxMessage` **or** `MMDFMessage` **state**
“unseen” sequence	no R flag
“replied” sequence	A flag
“flagged” sequence	F flag

When an `MHMessage` instance is created based upon a `BabylMessage` instance, the following conversions take place:

Resulting state	`BabylMessage` **state**
“unseen” sequence	“unseen” label
“replied” sequence	“answered” label

`BabylMessage` objects

class mailbox.BabylMessage(*message=None*)

A message with Babyl-specific behaviors. Parameter *message* has the same meaning as with the `Message` constructor.

Certain message labels, called *attributes*, are defined by convention to have special meanings. The attributes are as follows:

Label	**Explanation**
unseen	Not read, but previously detected by MUA
deleted	Marked for subsequent deletion
filed	Copied to another file or mailbox
answered	Replied to
forwarded	Forwarded
edited	Modified by the user
resent	Resent

By default, Rmail displays only visible headers. The **`BabylMessage`** class, though, uses the original headers because they are more complete. Visible headers may be accessed explicitly if desired.

`BabylMessage` instances offer the following methods:

get_labels()

Return a list of labels on the message.

set_labels(*labels*)

Set the list of labels on the message to *labels*.

add_label(*label*)

Add *label* to the list of labels on the message.

remove_label(*label*)

Remove *label* from the list of labels on the message.

get_visible()

Return a `Message` instance whose headers are the message's visible headers and whose body is empty.

set_visible(*visible*)

Set the message's visible headers to be the same as the headers in *message*. Parameter *visible* should be a `Message` instance, an `email.message.Message` instance, a string, or a file-like object (which should be open in text mode).

update_visible()

When a **`BabylMessage`** instance's original headers are modified, the visible headers are not automatically modified to correspond. This method updates the visible headers as follows: each visible header with a corresponding original header is set to the value of the original header, each visible header without a corresponding original header is removed, and any of *Date*, *From*, *Reply-To*, *To*, *CC*, and *Subject* that are present in the original headers but not the visible headers are added to the visible headers.

When a **`BabylMessage`** instance is created based upon a `MaildirMessage` instance, the following conversions take place:

Resulting state	`MaildirMessage` **state**
"unseen" label	no S flag

Resulting state	`MaildirMessage` **state**
“deleted” label	T flag
“answered” label	R flag
“forwarded” label	P flag

When a **`BabylMessage`** instance is created based upon an `mboxMessage` or `MMDFMessage` instance, the *Status* and *X-Status* headers are omitted and the following conversions take place:

Resulting state	`mboxMessage` **or** `MMDFMessage` **state**
“unseen” label	no R flag
“deleted” label	D flag
“answered” label	A flag

When a **`BabylMessage`** instance is created based upon an `MHMessage` instance, the following conversions take place:

Resulting state	`MHMessage` state
“unseen” label	“unseen” sequence
“answered” label	“replied” sequence

`MMDFMessage` objects

class mailbox.MMDFMessage(*message=None*)

A message with MMDF-specific behaviors. Parameter *message* has the same meaning as with the `Message` constructor.

As with message in an mbox mailbox, MMDF messages are stored with the sender’s address and the delivery date in an initial line beginning with “From “. Likewise, flags that indicate the state of the message are typically stored in *Status* and *X-Status* headers.

Conventional flags for MMDF messages are identical to those of mbox message and are as follows:

Flag	Meaning	Explanation
R	Read	Read
O	Old	Previously detected by MUA
D	Deleted	Marked for subsequent deletion
F	Flagged	Marked as important
A	Answered	Replied to

Flag	Meaning	Explanation

The “R” and “O” flags are stored in the *Status* header, and the “D”, “F”, and “A” flags are stored in the *X-Status* header. The flags and headers typically appear in the order mentioned.

`MMDFMessage` instances offer the following methods, which are identical to those offered by **`mboxMessage`**:

get_from()

> Return a string representing the “From “ line that marks the start of the message in an mbox mailbox. The leading “From “ and the trailing newline are excluded.

set_from(*from_*, *time_=None*)

> Set the “From “ line to *from_*, which should be specified without a leading “From “ or trailing newline. For convenience, *time_* may be specified and will be formatted appropriately and appended to *from_*. If *time_* is specified, it should be a **`time.struct_time`** instance, a tuple suitable for passing to **`time.strftime()`**, or `True` (to use **`time.gmtime()`**).

get_flags()

> Return a string specifying the flags that are currently set. If the message complies with the conventional format, the result is the concatenation in the following order of zero or one occurrence of each of `'R'`, `'O'`, `'D'`, `'F'`, and `'A'`.

set_flags(*flags*)

Set the flags specified by *flags* and unset all others. Parameter *flags* should be the concatenation in any order of zero or more occurrences of each of `'R'`, `'O'`, `'D'`, `'F'`, and `'A'`.

add_flag(*flag*)

Set the flag(s) specified by *flag* without changing other flags. To add more than one flag at a time, *flag* may be a string of more than one character.

remove_flag(*flag*)

Unset the flag(s) specified by *flag* without changing other flags. To remove more than one flag at a time, *flag* maybe a string of more than one character.

When an **`MMDFMessage`** instance is created based upon a `MaildirMessage` instance, a “From “ line is generated based upon the `MaildirMessage` instance’s delivery date, and the following conversions take place:

Resulting state	`MaildirMessage` **state**
R flag	S flag
O flag	“cur” subdirectory
D flag	T flag
F flag	F flag
A flag	R flag

Resulting state	`MaildirMessage` state

When an **`MMDFMessage`** instance is created based upon an `MHMessage` instance, the following conversions take place:

Resulting state	`MHMessage` state
R flag and O flag	no “unseen” sequence
O flag	“unseen” sequence
F flag	“flagged” sequence
A flag	“replied” sequence

When an **`MMDFMessage`** instance is created based upon a `BabylMessage` instance, the following conversions take place:

Resulting state	`BabylMessage` state
R flag and O flag	no “unseen” label
O flag	“unseen” label
D flag	“deleted” label

Resulting state	`BabylMessage` state
A flag	“answered” label

When an **`MMDFMessage`** instance is created based upon an `mboxMessage` instance, the “From “ line is copied and all flags directly correspond:

Resulting state	`mboxMessage` state
R flag	R flag
O flag	O flag
D flag	D flag
F flag	F flag
A flag	A flag

Exceptions

The following exception classes are defined in the **`mailbox`** module:

exception mailbox.Error

The based class for all other module-specific exceptions.

exception mailbox.NoSuchMailboxError

Raised when a mailbox is expected but is not found, such as when instantiating a `Mailbox` subclass with a path that does not exist (and with the *create* parameter set to `False`), or when opening a folder that does not exist.

exception mailbox.NotEmptyError

Raised when a mailbox is not empty but is expected to be, such as when deleting a folder that contains messages.

exception mailbox.ExternalClashError

Raised when some mailbox-related condition beyond the control of the program causes it to be unable to proceed, such as when failing to acquire a lock that another program already holds a lock, or when a uniquely generated file name already exists.

exception mailbox.FormatError

Raised when the data in a file cannot be parsed, such as when an `MH` instance attempts to read a corrupted `.mh_sequences` file.

Examples

A simple example of printing the subjects of all messages in a mailbox that seem interesting:

```
import mailbox
for message in mailbox.mbox('~/mbox'):
    subject = message['subject']       # Could possibly be
None.
    if subject and 'python' in subject.lower():
        print(subject)
```

To copy all mail from a Babyl mailbox to an MH mailbox, converting all of the format-specific information that can be converted:

```
import mailbox
destination = mailbox.MH('~/Mail')
destination.lock()
for message in mailbox.Babyl('~/RMAIL'):
    destination.add(mailbox.MHMessage(message))
destination.flush()
destination.unlock()
```

This example sorts mail from several mailing lists into different mailboxes, being careful to avoid mail corruption due to concurrent modification by other programs, mail loss due to interruption of the program, or premature termination due to malformed messages in the mailbox:

```
import mailbox
import email.errors

list_names = ('python-list', 'python-dev', 'python-bugs')

boxes = {name: mailbox.mbox('~/email/%s' % name) for name in
list_names}
inbox = mailbox.Maildir('~/Maildir', factory=None)

for key in inbox.iterkeys():
    try:
        message = inbox[key]
    except email.errors.MessageParseError:
        continue                # The message is malformed.
Just leave it.

    for name in list_names:
        list_id = message['list-id']
        if list_id and name in list_id:
            # Get mailbox to use
            box = boxes[name]

            # Write copy to disk before removing original.
            # If there's a crash, you might duplicate a
message, but
            # that's better than losing a message completely.
            box.lock()
            box.add(message)
            box.flush()
            box.unlock()

            # Remove original message
            inbox.lock()
            inbox.discard(key)
            inbox.flush()
            inbox.unlock()
            break               # Found destination, so stop
looking.

for box in boxes.itervalues():
    box.close()
```

`mimetypes` — Map filenames to MIME types

Source code: Lib/mimetypes.py [https://github.com/python/cpython/tree/3.14/Lib/mimetypes.py]

The **`mimetypes`** module converts between a filename or URL and the MIME type associated with the filename extension. Conversions are provided from filename to MIME type and from MIME type to filename extension; encodings are not supported for the latter conversion.

The module provides one class and a number of convenience functions. The functions are the normal interface to this module, but some applications may be interested in the class as well.

The functions described below provide the primary interface for this module. If the module has not been initialized, they will call **`init()`** if they rely on the information **`init()`** sets up.

mimetypes.guess_type(*url*, *strict=True*)

Guess the type of a file based on its filename, path or URL, given by *url*. URL can be a string or a path-like object.

The return value is a tuple `(type, encoding)` where *type* is `None` if the type can't be guessed (missing or unknown suffix) or a string of the form `'type/subtype'`, usable for a MIME *content-type* header.

encoding is `None` for no encoding or the name of the program used to encode (e.g. **compress** or **gzip**). The encoding is suitable for use as a *Content-Encoding* header, **not** as a *Content-Transfer-Encoding* header. The mappings are table driven. Encoding suffixes are case sensitive; type suffixes are first tried case sensitively, then case insensitively.

The optional *strict* argument is a flag specifying whether the list of known MIME types is limited to only the official types registered with IANA [https://www.iana.org/assignments/media-types/media-types.xhtml]. However, the behavior of this module also depends on the underlying operating system. Only file types recognized by the OS or explicitly registered with Python's internal database can be identified. When *strict* is `True` (the default), only the IANA types are supported; when *strict* is `False`, some additional non-standard but commonly used MIME types are also recognized.

Changed in version 3.8: Added support for *url* being a path-like object.

Deprecated since version 3.13: Passing a file path instead of URL is soft deprecated. Use **`guess_file_type()`** for this.

mimetypes.guess_file_type(*path*, *, *strict=True*)

Guess the type of a file based on its path, given by *path*. Similar to the **`guess_type()`** function, but accepts a path instead of URL. Path can be a string, a bytes object or a path-like object.

Added in version 3.13.

mimetypes.guess_all_extensions(*type*, *strict=True*)

Guess the extensions for a file based on its MIME type, given by *type*. The return value is a list of strings giving all possible filename extensions, including the leading dot (`'.'`). The extensions are not guaranteed to have been associated with any particular data stream, but would be mapped to the MIME type *type* by **`guess_type()`** and **`guess_file_type()`**.

The optional *strict* argument has the same meaning as with the **`guess_type()`** function.

mimetypes.guess_extension(*type*, *strict=True*)

Guess the extension for a file based on its MIME type, given by *type*. The return value is a string giving a filename extension, including the leading dot (`'.'`). The extension is not guaranteed to have been associated with any particular data stream, but would be mapped to the MIME type *type* by **`guess_type()`** and **`guess_file_type()`**. If no extension can be guessed for *type*, `None` is returned.

The optional *strict* argument has the same meaning as with the **`guess_type()`** function.

Some additional functions and data items are available for controlling the behavior of the module.

mimetypes.init(*files=None*)

Initialize the internal data structures. If given, *files* must be a sequence of file names which should be used to augment the default type map. If omitted, the file names to use are taken from **`knownfiles`**; on Windows, the current registry settings are loaded. Each file named in *files* or **`knownfiles`** takes precedence over those named before it. Calling **`init()`** repeatedly is allowed.

Specifying an empty list for *files* will prevent the system defaults from being applied: only the well-known values will be present from a built-in list.

If *files* is `None` the internal data structure is completely rebuilt to its initial default value. This is a stable operation and will produce the same results when called multiple times.

Changed in version 3.2: Previously, Windows registry settings were ignored.

mimetypes.read_mime_types(*filename*)

Load the type map given in the file *filename*, if it exists. The type map is returned as a dictionary mapping filename extensions, including the leading dot (`'.'`), to strings of the form `'type/subtype'`. If the file *filename* does not exist or cannot be read, `None` is returned.

mimetypes.add_type(*type*, *ext*, *strict=True*)

Add a mapping from the MIME type *type* to the extension *ext*. When the extension is already known, the new type will replace the old one. When the type is already known the extension will be added to the list of known extensions.

When *strict* is `True` (the default), the mapping will be added to the official MIME types, otherwise to the non-standard ones.

mimetypes.inited

Flag indicating whether or not the global data structures have been initialized. This is set to `True` by **`init()`**.

mimetypes.knownfiles

List of type map file names commonly installed. These files are typically named `mime.types` and are installed in different locations by different packages.

mimetypes.suffix_map

Dictionary mapping suffixes to suffixes. This is used to allow recognition of encoded files for which the encoding and the type are indicated by the same extension. For example, the `.tgz` extension is mapped to `.tar.gz` to allow the encoding and type to be recognized separately.

mimetypes.encodings_map

Dictionary mapping filename extensions to encoding types.

mimetypes.types_map

Dictionary mapping filename extensions to MIME types.

mimetypes.common_types

Dictionary mapping filename extensions to non-standard, but commonly found MIME types.

An example usage of the module:

```
>>> import mimetypes
>>> mimetypes.init()
>>> mimetypes.knownfiles
['/etc/mime.types', '/etc/httpd/mime.types', ... ]
>>> mimetypes.suffix_map['.tgz']
'.tar.gz'
>>> mimetypes.encodings_map['.gz']
'gzip'
>>> mimetypes.types_map['.tgz']
'application/x-tar-gz'
```

MimeTypes objects

The `MimeTypes` class may be useful for applications which may want more than one MIME-type database; it provides an interface similar to the one of the `mimetypes` module.

class mimetypes.MimeTypes(*filenames=()*, *strict=True*)

This class represents a MIME-types database. By default, it provides access to the same database as the rest of this module. The initial database is a copy of that provided by the module, and may be extended by loading additional `mime.types`-style files into the database using the `read()` or `readfp()` methods. The mapping dictionaries may also be cleared before loading additional data if the default data is not desired.

The optional *filenames* parameter can be used to cause additional files to be loaded “on top” of the default database.

suffix_map

Dictionary mapping suffixes to suffixes. This is used to allow recognition of encoded files for which the encoding and the type are indicated by the same extension. For example, the `.tgz` extension is mapped to `.tar.gz` to allow the encoding and type to be recognized separately. This is initially a copy of the global `suffix_map` defined in the module.

encodings_map

Dictionary mapping filename extensions to encoding types. This is initially a copy of the global `encodings_map` defined in the module.

types_map

Tuple containing two dictionaries, mapping filename extensions to MIME types: the first dictionary is for the non-standards types and the second one is for the standard types. They are initialized by `common_types` and `types_map`.

types_map_inv

Tuple containing two dictionaries, mapping MIME types to a list of filename extensions: the first dictionary is for the non-standards types and the second one is for the standard types. They are initialized by `common_types` and `types_map`.

guess_extension(*type*, *strict=True*)

Similar to the `guess_extension()` function, using the tables stored as part of the object.

guess_type(*url*, *strict=True*)

Similar to the `guess_type()` function, using the tables stored as part of the object.

guess_file_type(*path*, *, *strict=True*)

Similar to the `guess_file_type()` function, using the tables stored as part of the object.

Added in version 3.13.

guess_all_extensions(*type*, *strict=True*)

Similar to the `guess_all_extensions()` function, using the tables stored as part of the object.

read(*filename*, *strict=True*)

Load MIME information from a file named *filename*. This uses `readfp()` to parse the file.

If *strict* is `True`, information will be added to list of standard types, else to the list of non-standard types.

readfp(*fp*, *strict=True*)

Load MIME type information from an open file *fp*. The file must have the format of the standard `mime.types` files.

If *strict* is `True`, information will be added to the list of standard types, else to the list of non-standard types.

read_windows_registry(*strict=True*)

Load MIME type information from the Windows registry.

Availability: Windows.

If *strict* is `True`, information will be added to the list of standard types, else to the list of non-standard types.

Added in version 3.2.

add_type(*type*, *ext*, *strict=True*)

Add a mapping from the MIME type *type* to the extension *ext*. Valid extensions start with a ‘.’ or are empty. When the extension is already known, the new type will replace the old one. When the type is already known the extension will be added to the list of known extensions.

When *strict* is `True` (the default), the mapping will be added to the official MIME types, otherwise to the non-standard ones.

Deprecated since version 3.14, will be removed in version 3.16: Invalid, undotted extensions will raise a `ValueError` in Python 3.16.

Command-line usage

The **`mimetypes`** module can be executed as a script from the command line.

```
python -m mimetypes [-h] [-e] [-l] type [type ...]
```

The following options are accepted:

-h
--help
    Show the help message and exit.

-e
--extension
    Guess extension instead of type.

-l
--lenient
    Additionally search for some common, but non-standard types.

By default the script converts MIME types to file extensions. However, if `--extension` is specified, it converts file extensions to MIME types.

For each `type` entry, the script writes a line into the standard output stream. If an unknown type occurs, it writes an error message into the standard error stream and exits with the return code `1`.

Command-line example

Here are some examples of typical usage of the **`mimetypes`** command-line interface:

```
$ # get a MIME type by a file name
$ python -m mimetypes filename.png
type: image/png encoding: None
```

```
$ # get a MIME type by a URL
$ python -m mimetypes https://example.com/filename.txt
type: text/plain encoding: None

$ # get a complex MIME type
$ python -m mimetypes filename.tar.gz
type: application/x-tar encoding: gzip

$ # get a MIME type for a rare file extension
$ python -m mimetypes filename.pict
error: unknown extension of filename.pict

$ # now look in the extended database built into Python
$ python -m mimetypes --lenient filename.pict
type: image/pict encoding: None

$ # get a file extension by a MIME type
$ python -m mimetypes --extension text/javascript
.js

$ # get a file extension by a rare MIME type
$ python -m mimetypes --extension text/xul
error: unknown type text/xul

$ # now look in the extended database again
$ python -m mimetypes --extension --lenient text/xul
.xul

$ # try to feed an unknown file extension
$ python -m mimetypes filename.sh filename.nc filename.xxx
filename.txt
type: application/x-sh encoding: None
type: application/x-netcdf encoding: None
error: unknown extension of filename.xxx

$ # try to feed an unknown MIME type
$ python -m mimetypes --extension audio/aac audio/opus
audio/future audio/x-wav
.aac
.opus
error: unknown type audio/future
```

`base64` — Base16, Base32, Base64, Base85 Data Encodings

Source code: Lib/base64.py [https://github.com/python/cpython/tree/3.14/Lib/base64.py]

This module provides functions for encoding binary data to printable ASCII characters and decoding such encodings back to binary data. This includes the encodings specified in **RFC 4648** [https://datatracker.ietf.org/doc/html/rfc4648.html] (Base64, Base32 and Base16) and the non-standard Base85 encodings.

There are two interfaces provided by this module. The modern interface supports encoding bytes-like objects to ASCII `bytes`, and decoding bytes-like objects or strings containing ASCII to `bytes`. Both base-64 alphabets defined in **RFC 4648** [https://datatracker.ietf.org/doc/html/rfc4648.html] (normal, and URL- and filesystem-safe) are supported.

The legacy interface does not support decoding from strings, but it does provide functions for encoding and decoding to and from file objects. It only supports the Base64 standard alphabet, and it adds newlines every 76 characters as per **RFC 2045** [https://datatracker.ietf.org/doc/html/rfc2045.html]. Note that if you are looking for **RFC 2045** [https://datatracker.ietf.org/doc/html/rfc2045.html] support you probably want to be looking at the `email` package instead.

Changed in version 3.3: ASCII-only Unicode strings are now accepted by the decoding functions of the modern interface.

Changed in version 3.4: Any bytes-like objects are now accepted by all encoding and decoding functions in this module. Ascii85/Base85 support added.

RFC 4648 Encodings

The **RFC 4648** [https://datatracker.ietf.org/doc/html/rfc4648.html] encodings are suitable for encoding binary data so that it can be safely sent by email, used as parts of URLs, or included as part of an HTTP POST request.

base64.b64encode(*s*, *altchars=None*)

Encode the bytes-like object *s* using Base64 and return the encoded **`bytes`**.

Optional *altchars* must be a bytes-like object of length 2 which specifies an alternative alphabet for the `+` and `/` characters. This allows an application to e.g. generate URL or filesystem safe Base64 strings. The default is `None`, for which the standard Base64 alphabet is used.

May assert or raise a **`ValueError`** if the length of *altchars* is not 2. Raises a **`TypeError`** if *altchars* is not a bytes-like object.

base64.b64decode(*s*, *altchars=None*, *validate=False*)

Decode the Base64 encoded bytes-like object or ASCII string *s* and return the decoded **`bytes`**.

Optional *altchars* must be a bytes-like object or ASCII string of length 2 which specifies the alternative alphabet used instead of the `+` and `/` characters.

A **`binascii.Error`** exception is raised if *s* is incorrectly padded.

If *validate* is `False` (the default), characters that are neither in the normal base-64 alphabet nor the alternative alphabet are discarded prior to the padding check. If *validate* is `True`, these non-alphabet characters in the input result in a **`binascii.Error`**.

For more information about the strict base64 check, see **`binascii.a2b_base64()`**

May assert or raise a **`ValueError`** if the length of *altchars* is not 2.

base64.standard_b64encode(*s*)

Encode bytes-like object *s* using the standard Base64 alphabet and return the encoded **`bytes`**.

base64.standard_b64decode(*s*)

Decode bytes-like object or ASCII string *s* using the standard Base64 alphabet and return the decoded **`bytes`**.

base64.urlsafe_b64encode(*s*)

Encode bytes-like object *s* using the URL- and filesystem-safe alphabet, which substitutes `-` instead of `+` and `_` instead of `/` in the standard Base64 alphabet, and return the encoded **`bytes`**. The result can still contain `=`.

base64.urlsafe_b64decode(*s*)

Decode bytes-like object or ASCII string *s* using the URL- and filesystem-safe alphabet, which substitutes `-` instead of `+` and `_` instead of `/` in the standard Base64 alphabet, and return the decoded **`bytes`**.

base64.b32encode(*s*)

Encode the bytes-like object *s* using Base32 and return the encoded `bytes`.

base64.b32decode(*s*, *casefold=False*, *map01=None*)

Decode the Base32 encoded bytes-like object or ASCII string *s* and return the decoded `bytes`.

Optional *casefold* is a flag specifying whether a lowercase alphabet is acceptable as input. For security purposes, the default is `False`.

RFC 4648 [https://datatracker.ietf.org/doc/html/rfc4648.html] allows for optional mapping of the digit 0 (zero) to the letter O (oh), and for optional mapping of the digit 1 (one) to either the letter I (eye) or letter L (el). The optional argument *map01* when not `None`, specifies which letter the digit 1 should be mapped to (when *map01* is not `None`, the digit 0 is always mapped to the letter O). For security purposes the default is `None`, so that 0 and 1 are not allowed in the input.

A `binascii.Error` is raised if *s* is incorrectly padded or if there are non-alphabet characters present in the input.

base64.b32hexencode(*s*)

Similar to `b32encode()` but uses the Extended Hex Alphabet, as defined in **RFC 4648** [https://datatracker.ietf.org/doc/html/rfc4648.html].

Added in version 3.10.

base64.b32hexdecode(*s*, *casefold=False*)

Similar to `b32decode()` but uses the Extended Hex Alphabet, as defined in **RFC 4648** [https://datatracker.ietf.org/doc/html/rfc4648.html].

This version does not allow the digit 0 (zero) to the letter O (oh) and digit 1 (one) to either the letter I (eye) or letter L (el) mappings, all these characters are included in the Extended Hex Alphabet and are not interchangeable.

Added in version 3.10.

base64.b16encode(*s*)

Encode the bytes-like object *s* using Base16 and return the encoded **`bytes`**.

base64.b16decode(*s*, *casefold=False*)

Decode the Base16 encoded bytes-like object or ASCII string *s* and return the decoded **`bytes`**.

Optional *casefold* is a flag specifying whether a lowercase alphabet is acceptable as input. For security purposes, the default is `False`.

A **`binascii.Error`** is raised if *s* is incorrectly padded or if there are non-alphabet characters present in the input.

Base85 Encodings

Base85 encoding is not formally specified but rather a de facto standard, thus different systems perform the encoding differently.

The **`a85encode()`** and **`b85encode()`** functions in this module are two implementations of the de facto standard. You should call the function with the Base85 implementation used by the software you intend to work with.

The two functions present in this module differ in how they handle the following:

- Whether to include enclosing `<~` and `~>` markers
- Whether to include newline characters
- The set of ASCII characters used for encoding
- Handling of null bytes

Refer to the documentation of the individual functions for more information.

base64.a85encode(*b*, *, *foldspaces=False*, *wrapcol=0*, *pad=False*, *adobe=False*)

Encode the bytes-like object *b* using Ascii85 and return the encoded **`bytes`**.

foldspaces is an optional flag that uses the special short sequence ‘y’ instead of 4 consecutive spaces (ASCII 0x20) as supported by ‘btoa’. This feature is not supported by the “standard” Ascii85 encoding.

wrapcol controls whether the output should have newline (`b'\n'`) characters added to it. If this is non-zero, each output line will be at most this many characters long, excluding the trailing newline.

pad controls whether the input is padded to a multiple of 4 before encoding. Note that the `btoa` implementation always pads.

adobe controls whether the encoded byte sequence is framed with `<~` and `~>`, which is used by the Adobe implementation.

Added in version 3.4.

base64.a85decode(*b*, *, *foldspaces=False*, *adobe=False*, *ignorechars=b' \t\n\r\x0b'*)

Decode the Ascii85 encoded bytes-like object or ASCII string *b* and return the decoded **`bytes`**.

foldspaces is a flag that specifies whether the ‘y’ short sequence should be accepted as shorthand for 4 consecutive spaces (ASCII 0x20). This feature is not supported by the “standard” Ascii85 encoding.

adobe controls whether the input sequence is in Adobe Ascii85 format (i.e. is framed with <~ and ~>).

ignorechars should be a bytes-like object or ASCII string containing characters to ignore from the input. This should only contain whitespace characters, and by default contains all whitespace characters in ASCII.

Added in version 3.4.

base64.b85encode(*b*, *pad=False*)

Encode the bytes-like object *b* using base85 (as used in e.g. git-style binary diffs) and return the encoded **`bytes`**.

If *pad* is true, the input is padded with `b'\0'` so its length is a multiple of 4 bytes before encoding.

Added in version 3.4.

base64.b85decode(*b*)

Decode the base85-encoded bytes-like object or ASCII string *b* and return the decoded **`bytes`**. Padding is implicitly removed, if necessary.

Added in version 3.4.

base64.z85encode(*s*)

Encode the bytes-like object *s* using Z85 (as used in ZeroMQ) and return the encoded **`bytes`**. See Z85 specification [https://rfc.zeromq.org/spec/32/] for more information.

Added in version 3.13.

base64.z85decode(*s*)

Decode the Z85-encoded bytes-like object or ASCII string *s* and return the decoded **`bytes`**. See Z85 specification [https://rfc.zeromq.org/spec/32/] for more information.

Added in version 3.13.

Legacy Interface

base64.decode(*input*, *output*)

Decode the contents of the binary *input* file and write the resulting binary data to the *output* file. *input* and *output* must be file objects. *input* will be read until `input.readline()` returns an empty bytes object.

base64.decodebytes(*s*)

Decode the bytes-like object *s*, which must contain one or more lines of base64 encoded data, and return the decoded **`bytes`**.

Added in version 3.1.

base64.encode(*input*, *output*)

Encode the contents of the binary *input* file and write the resulting base64 encoded data to the *output* file. *input* and *output* must be file objects. *input* will be read until `input.read()` returns an empty bytes object. **`encode()`** inserts a newline character (`b'\n'`) after every 76 bytes of the output, as well as ensuring that the output always ends with a newline, as per **RFC 2045** [https://datatracker.ietf.org/doc/html/rfc2045.html] (MIME).

base64.encodebytes(*s*)

Encode the bytes-like object *s*, which can contain arbitrary binary data, and return **`bytes`** containing the base64-encoded data, with newlines (`b'\n'`) inserted after every 76 bytes of output, and ensuring that there is a trailing newline, as per **RFC 2045** [https://datatracker.ietf.org/doc/html/rfc2045.html] (MIME).

Added in version 3.1.

An example usage of the module:

```
>>> import base64
>>> encoded = base64.b64encode(b'data to be encoded')
>>> encoded
b'ZGF0YSB0byBiZSBlbmNvZGVk'
>>> data = base64.b64decode(encoded)
>>> data
b'data to be encoded'
```

Security Considerations

A new security considerations section was added to **RFC 4648** [https://datatracker.ietf.org/doc/html/rfc4648.html] (section 12); it’s recommended to review the security section for any code deployed to production.

See also

Module `binascii`

Support module containing ASCII-to-binary and binary-to-ASCII conversions.

RFC 1521 [https://datatracker.ietf.org/doc/html/rfc1521.html] - MIME (Multipurpose Internet Mail Extensions) Part One: Mechanisms for Specifying and Describing the Format of Internet Message Bodies

Section 5.2, “Base64 Content-Transfer-Encoding,” provides the definition of the base64 encoding.

`binascii` — Convert between binary and ASCII

The **`binascii`** module contains a number of methods to convert between binary and various ASCII-encoded binary representations. Normally, you will not use these functions directly but use wrapper modules like **`base64`** instead. The **`binascii`** module contains low-level functions written in C for greater speed that are used by the higher-level modules.

Note

`a2b_*` functions accept Unicode strings containing only ASCII characters. Other functions only accept bytes-like objects (such as **`bytes`**, **`bytearray`** and other objects that support the buffer protocol).

Changed in version 3.3: ASCII-only unicode strings are now accepted by the `a2b_*` functions.

The **`binascii`** module defines the following functions:

binascii.a2b_uu(*string*)

Convert a single line of uuencoded data back to binary and return the binary data. Lines normally contain 45 (binary) bytes, except for the last line. Line data may be followed by whitespace.

binascii.b2a_uu(*data*, *, *backtick=False*)

Convert binary data to a line of ASCII characters, the return value is the converted line, including a newline char. The length of *data* should be at most 45. If *backtick* is true, zeros are represented by `` '`' `` instead of spaces.

Changed in version 3.7: Added the *backtick* parameter.

binascii.a2b_base64(*string*, */*, ***, *strict_mode=False*)

Convert a block of base64 data back to binary and return the binary data. More than one line may be passed at a time.

If *strict_mode* is true, only valid base64 data will be converted. Invalid base64 data will raise `binascii.Error`.

Valid base64:

- Conforms to **RFC 3548** [https://datatracker.ietf.org/doc/html/rfc3548.html].
- Contains only characters from the base64 alphabet.
- Contains no excess data after padding (including excess padding, newlines, etc.).
- Does not start with a padding.

Changed in version 3.11: Added the *strict_mode* parameter.

binascii.b2a_base64(*data*, ***, *newline=True*)

Convert binary data to a line of ASCII characters in base64 coding. The return value is the converted line, including a newline char if *newline* is true. The output of this function conforms to **RFC 3548** [https://datatracker.ietf.org/doc/html/rfc3548.html].

Changed in version 3.6: Added the *newline* parameter.

binascii.a2b_qp(*data*, *header=False*)

Convert a block of quoted-printable data back to binary and return the binary data. More than one line may be passed at a time. If the optional argument *header* is present and true, underscores will be decoded as spaces.

binascii.b2a_qp(*data*, *quotetabs=False*, *istext=True*, *header=False*)

Convert binary data to a line(s) of ASCII characters in quoted-printable encoding. The return value is the converted line(s). If the optional argument *quotetabs* is present and true, all tabs and spaces will be encoded. If the optional argument *istext* is present and true, newlines are not encoded but trailing whitespace will be encoded. If the optional argument *header* is present and true, spaces will be encoded as underscores per **RFC 1522** [https://datatracker.ietf.org/doc/html/rfc1522.html]. If the optional argument *header* is present and false, newline characters will be encoded as well; otherwise linefeed conversion might corrupt the binary data stream.

binascii.crc_hqx(*data*, *value*)

Compute a 16-bit CRC value of *data*, starting with *value* as the initial CRC, and return the result. This uses the CRC-CCITT polynomial $x^{16} + x^{12} + x^5 + 1$, often represented as 0x1021. This CRC is used in the binhex4 format.

binascii.crc32(*data*[, *value*])

Compute CRC-32, the unsigned 32-bit checksum of *data*, starting with an initial CRC of *value*. The default initial CRC is zero. The algorithm is consistent with the ZIP file checksum. Since the algorithm is designed for use as a checksum algorithm, it is not suitable for use as a general hash algorithm. Use as follows:

```
print(binascii.crc32(b"hello world"))
# Or, in two pieces:
crc = binascii.crc32(b"hello")
crc = binascii.crc32(b" world", crc)
print('crc32 = {:#010x}'.format(crc))
```

Changed in version 3.0: The result is always unsigned.

binascii.b2a_hex(*data*[, *sep*[, *bytes_per_sep=1*]])

binascii.hexlify(*data*[, *sep*[, *bytes_per_sep=1*]])

Return the hexadecimal representation of the binary *data*. Every byte of *data* is converted into the corresponding 2-digit hex representation. The returned bytes object is therefore twice as long as the length of *data*.

Similar functionality (but returning a text string) is also conveniently accessible using the `bytes.hex()` method.

If *sep* is specified, it must be a single character str or bytes object. It will be inserted in the output after every *bytes_per_sep* input bytes. Separator placement is counted from the right end of the output by default, if you wish to count from the left, supply a negative *bytes_per_sep* value.

```
>>> import binascii
>>> binascii.b2a_hex(b'\xb9\x01\xef')
b'b901ef'
>>> binascii.hexlify(b'\xb9\x01\xef', '-')
b'b9-01-ef'
>>> binascii.b2a_hex(b'\xb9\x01\xef', b'_', 2)
b'b9_01ef'
>>> binascii.b2a_hex(b'\xb9\x01\xef', b' ', -2)
b'b901 ef'
```

Changed in version 3.8: The *sep* and *bytes_per_sep* parameters were added.

binascii.a2b_hex(*hexstr*)
binascii.unhexlify(*hexstr*)

Return the binary data represented by the hexadecimal string *hexstr*. This function is the inverse of `b2a_hex()`. *hexstr* must contain an even number of hexadecimal digits (which can be upper or lower case), otherwise an `Error` exception is raised.

Similar functionality (accepting only text string arguments, but more liberal towards whitespace) is also accessible using the `bytes.fromhex()` class method.

exception binascii.Error

Exception raised on errors. These are usually programming errors.

exception binascii.Incomplete

Exception raised on incomplete data. These are usually not programming errors, but may be handled by reading a little more data and trying again.

See also

Module `base64`

Support for RFC compliant base64-style encoding in base 16, 32, 64, and 85.

Module `quopri`

Support for quoted-printable encoding used in MIME email messages.

`quopri` — Encode and decode MIME quoted-printable data

Source code: Lib/quopri.py [https://github.com/python/cpython/tree/3.14/Lib/quopri.py]

This module performs quoted-printable transport encoding and decoding, as defined in **RFC 1521** [https://datatracker.ietf.org/doc/html/rfc1521.html]: “MIME (Multipurpose Internet Mail Extensions) Part One: Mechanisms for Specifying and Describing the Format of Internet Message Bodies”. The quoted-printable encoding is designed for data where there are relatively few nonprintable characters; the base64 encoding scheme available via the **`base64`** module is more compact if there are many such characters, as when sending a graphics file.

quopri.decode(*input*, *output*, *header=False*)

Decode the contents of the *input* file and write the resulting decoded binary data to the *output* file. *input* and *output* must be binary file objects. If the optional argument *header* is present and true, underscore will be decoded as space. This is used to decode “Q”-encoded headers as described in **RFC 1522** [https://datatracker.ietf.org/doc/html/rfc1522.html]: “MIME (Multipurpose Internet Mail Extensions) Part Two: Message Header Extensions for Non-ASCII Text”.

quopri.encode(*input*, *output*, *quotetabs*, *header=False*)

Encode the contents of the *input* file and write the resulting quoted-printable data to the *output* file. *input* and *output* must be binary file objects. *quotetabs*, a non-optional flag which controls whether to encode embedded spaces and tabs; when true it encodes such embedded whitespace, and when false it leaves them unencoded. Note that spaces and tabs appearing at the end of lines are always encoded, as per **RFC 1521** [https://datatracker.ietf.org/doc/html/rfc1521.html]. *header* is a flag which controls if spaces are encoded as underscores as per **RFC 1522** [https://datatracker.ietf.org/doc/html/rfc1522.html].

quopri.decodestring(*s*, *header=False*)

Like **`decode()`**, except that it accepts a source **`bytes`** and returns the corresponding decoded **`bytes`**.

quopri.encodestring(*s*, *quotetabs=False*, *header=False*)

Like **`encode()`**, except that it accepts a source **`bytes`** and returns the corresponding encoded **`bytes`**. By default, it sends a `False` value to *quotetabs* parameter of the **`encode()`** function.

See also

Module **`base64`**

Encode and decode MIME base64 data

Structured Markup Processing Tools

Python supports a variety of modules to work with various forms of structured data markup. This includes modules to work with the Standard Generalized Markup Language (SGML) and the Hypertext Markup Language (HTML), and several interfaces for working with the Extensible Markup Language (XML).

- `html` — HyperText Markup Language support
- `html.parser` — Simple HTML and XHTML parser
    - Example HTML Parser Application
    - `HTMLParser` Methods
    - Examples
- `html.entities` — Definitions of HTML general entities
- XML Processing Modules
    - XML security
- `xml.etree.ElementTree` — The ElementTree XML API
    - Tutorial
        - XML tree and elements
        - Parsing XML
        - Pull API for non-blocking parsing
        - Finding interesting elements
        - Modifying an XML File
        - Building XML documents
        - Parsing XML with Namespaces
    - XPath support
        - Example
        - Supported XPath syntax
    - Reference
        - Functions

    - XInclude support
        - Example
    - Reference
        - Functions
        - Element Objects
        - ElementTree Objects
        - QName Objects
        - TreeBuilder Objects
        - XMLParser Objects
        - XMLPullParser Objects
        - Exceptions
- `xml.dom` — The Document Object Model API
    - Module Contents
    - Objects in the DOM
        - DOMImplementation Objects
        - Node Objects
        - NodeList Objects
        - DocumentType Objects
        - Document Objects
        - Element Objects
        - Attr Objects
        - NamedNodeMap Objects
        - Comment Objects
        - Text and CDATASection Objects
        - ProcessingInstruction Objects
        - Exceptions
    - Conformance
        - Type Mapping
        - Accessor Methods
- `xml.dom.minidom` — Minimal DOM implementation
    - DOM Objects
    - DOM Example
    - minidom and the DOM standard
- `xml.dom.pulldom` — Support for building partial DOM trees
    - DOMEventStream Objects
- `xml.sax` — Support for SAX2 parsers

    - SAXException Objects
- `xml.sax.handler` — Base classes for SAX handlers
    - ContentHandler Objects
    - DTDHandler Objects
    - EntityResolver Objects
    - ErrorHandler Objects
    - LexicalHandler Objects
- `xml.sax.saxutils` — SAX Utilities
- `xml.sax.xmlreader` — Interface for XML parsers
    - XMLReader Objects
    - IncrementalParser Objects
    - Locator Objects
    - InputSource Objects
    - The `Attributes` Interface
    - The `AttributesNS` Interface
- `xml.parsers.expat` — Fast XML parsing using Expat
    - XMLParser Objects
    - ExpatError Exceptions
    - Example
    - Content Model Descriptions
    - Expat error constants

`html` — HyperText Markup Language support

Source code: Lib/html/__init__.py
[https://github.com/python/cpython/tree/3.14/Lib/html/__init__.py]

This module defines utilities to manipulate HTML.

html.escape(*s*, *quote=True*)

Convert the characters `&`, `<` and `>` in string *s* to HTML-safe sequences. Use this if you need to display text that might contain such characters in HTML. If the optional flag *quote* is true (the default), the characters (`"`) and (`'`) are also translated; this helps for inclusion in an HTML attribute value delimited by quotes, as in `<a href="...">`. If *quote* is set to false, the characters (`"`) and (`'`) are not translated.

Added in version 3.2.

html.unescape(*s*)

Convert all named and numeric character references (e.g. `&gt;`, `&#62;`, `&#x3e;`) in the string *s* to the corresponding Unicode characters. This function uses the rules defined by the HTML 5 standard for both valid and invalid character references, and the **list of HTML 5 named character references**.

Added in version 3.4.

Submodules in the `html` package are:

- `html.parser` – HTML/XHTML parser with lenient parsing mode
- `html.entities` – HTML entity definitions

`html.parser` — Simple HTML and XHTML parser

Source code: Lib/html/parser.py [https://github.com/python/cpython/tree/3.14/Lib/html/parser.py]

This module defines a class **`HTMLParser`** which serves as the basis for parsing text files formatted in HTML (HyperText Mark-up Language) and XHTML.

class html.parser.HTMLParser(*, *convert_charrefs=True*, *scripting=False*)

Create a parser instance able to parse invalid markup.

If *convert_charrefs* is true (the default), all character references (except the ones in elements like `script` and `style`) are automatically converted to the corresponding Unicode characters.

If *scripting* is false (the default), the content of the `noscript` element is parsed normally; if it’s true, it’s returned as is without being parsed.

An **`HTMLParser`** instance is fed HTML data and calls handler methods when start tags, end tags, text, comments, and other markup elements are encountered. The user should subclass **`HTMLParser`** and override its methods to implement the desired behavior.

This parser does not check that end tags match start tags or call the end-tag handler for elements which are closed implicitly by closing an outer element.

Changed in version 3.4: convert_charrefs keyword argument added.

Changed in version 3.5: The default value for argument *convert_charrefs* is now `True`.

Changed in version 3.14.1: Added the *scripting* parameter.

Example HTML Parser Application

As a basic example, below is a simple HTML parser that uses the `HTMLParser` class to print out start tags, end tags, and data as they are encountered:

```
from html.parser import HTMLParser

class MyHTMLParser(HTMLParser):
    def handle_starttag(self, tag, attrs):
        print("Encountered a start tag:", tag)

    def handle_endtag(self, tag):
        print("Encountered an end tag :", tag)

    def handle_data(self, data):
        print("Encountered some data  :", data)

parser = MyHTMLParser()
parser.feed('<html><head><title>Test</title></head>'
            '<body><h1>Parse me!</h1></body></html>')
```

The output will then be:

```
Encountered a start tag: html
Encountered a start tag: head
Encountered a start tag: title
Encountered some data  : Test
Encountered an end tag : title
Encountered an end tag : head
Encountered a start tag: body
Encountered a start tag: h1
Encountered some data  : Parse me!
Encountered an end tag : h1
Encountered an end tag : body
Encountered an end tag : html
```

`HTMLParser` Methods

`HTMLParser` instances have the following methods:

HTMLParser.feed(*data*)

Feed some text to the parser. It is processed insofar as it consists of complete elements; incomplete data is buffered until more data is fed or `close()` is called. *data* must be `str`.

HTMLParser.close()

Force processing of all buffered data as if it were followed by an end-of-file mark. This method may be redefined by a derived class to define additional processing at the end of the input, but the redefined version should always call the `HTMLParser` base class method `close()`.

HTMLParser.reset()

Reset the instance. Loses all unprocessed data. This is called implicitly at instantiation time.

HTMLParser.getpos()

Return current line number and offset.

HTMLParser.get_starttag_text()

Return the text of the most recently opened start tag. This should not normally be needed for structured processing, but may be useful in dealing with HTML “as deployed” or for re-generating input with minimal changes (whitespace between attributes can be preserved, etc.).

The following methods are called when data or markup elements are encountered and they are meant to be overridden in a subclass. The base class implementations do nothing (except for **`handle_startendtag()`**):

HTMLParser.handle_starttag(*tag*, *attrs*)

This method is called to handle the start tag of an element (e.g. `<div id="main">`).

The *tag* argument is the name of the tag converted to lower case. The *attrs* argument is a list of `(name, value)` pairs containing the attributes found inside the tag's `<>` brackets. The *name* will be translated to lower case, and quotes in the *value* have been removed, and character and entity references have been replaced.

For instance, for the tag `<A HREF="https://www.cwi.nl/">`, this method would be called as `handle_starttag('a', [('href', 'https://www.cwi.nl/')])`.

All entity references from **`html.entities`** are replaced in the attribute values.

HTMLParser.handle_endtag(*tag*)

This method is called to handle the end tag of an element (e.g. `</div>`).

The *tag* argument is the name of the tag converted to lower case.

HTMLParser.handle_startendtag(*tag*, *attrs*)

Similar to **`handle_starttag()`**, but called when the parser encounters an XHTML-style empty tag (`<img ... />`). This method may be overridden by subclasses which require this particular lexical information; the default implementation simply calls **`handle_starttag()`** and **`handle_endtag()`**.

HTMLParser.handle_data(*data*)

This method is called to process arbitrary data (e.g. text nodes and the content of elements like `script` and `style`).

HTMLParser.handle_entityref(*name*)

This method is called to process a named character reference of the form `&name;` (e.g. `&gt;`), where *name* is a general entity reference (e.g. `'gt'`). This method is only called if *convert_charrefs* is false.

HTMLParser.handle_charref(*name*)

This method is called to process decimal and hexadecimal numeric character references of the form `&#NNN;` and `&#xNNN;`. For example, the decimal equivalent for `&gt;` is `&#62;`, whereas the hexadecimal is `&#x3E;`; in this case the method will receive `'62'` or `'x3E'`. This method is only called if *convert_charrefs* is false.

HTMLParser.handle_comment(*data*)

This method is called when a comment is encountered (e.g. `<!--comment-->`).

For example, the comment `<!-- comment -->` will cause this method to be called with the argument `' comment '`.

The content of Internet Explorer conditional comments (condcoms) will also be sent to this method, so, for `<!--[if IE 9]>IE9-specific content<![endif]-->`, this method will receive `'[if IE 9]>IE9-specific content<![endif]'`.

HTMLParser.handle_decl(*decl*)

This method is called to handle an HTML doctype declaration (e.g. `<!DOCTYPE html>`).

The *decl* parameter will be the entire contents of the declaration inside the `<!...>` markup (e.g. `'DOCTYPE html'`).

HTMLParser.handle_pi(*data*)

Method called when a processing instruction is encountered. The *data* parameter will contain the entire processing instruction. For example, for the processing instruction `<?proc color='red'>`, this method would be called as `handle_pi("proc color='red'")`. It is intended to be overridden by a derived class; the base class implementation does nothing.

Note

The **`HTMLParser`** class uses the SGML syntactic rules for processing instructions. An XHTML processing instruction using the trailing `'?'` will cause the `'?'` to be included in *data*.

HTMLParser.unknown_decl(*data*)

This method is called when an unrecognized declaration is read by the parser.

The *data* parameter will be the entire contents of the declaration inside the `<![...]>` markup. It is sometimes useful to be overridden by a derived class. The base class implementation does nothing.

Examples

The following class implements a parser that will be used to illustrate more examples:

```
from html.parser import HTMLParser
from html.entities import name2codepoint

class MyHTMLParser(HTMLParser):
    def handle_starttag(self, tag, attrs):
        print("Start tag:", tag)
        for attr in attrs:
            print("     attr:", attr)

    def handle_endtag(self, tag):
        print("End tag  :", tag)

    def handle_data(self, data):
        print("Data     :", data)

    def handle_comment(self, data):
        print("Comment  :", data)

    def handle_entityref(self, name):
        c = chr(name2codepoint[name])
        print("Named ent:", c)

    def handle_charref(self, name):
        if name.startswith('x'):
            c = chr(int(name[1:], 16))
        else:
            c = chr(int(name))
        print("Num ent  :", c)

    def handle_decl(self, data):
        print("Decl     :", data)

parser = MyHTMLParser()
```

Parsing a doctype:

```
>>> parser.feed('<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML
4.01//EN" '
...             '"http://www.w3.org/TR/html4/strict.dtd">')
Decl     : DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN"
"http://www.w3.org/TR/html4/strict.dtd"
```

Parsing an element with a few attributes and a title:

```
>>> parser.feed('<img src="python-logo.png" alt="The Python
logo">')
Start tag: img
     attr: ('src', 'python-logo.png')
     attr: ('alt', 'The Python logo')
>>>
>>> parser.feed('<h1>Python</h1>')
Start tag: h1
Data     : Python
End tag  : h1
```

The content of elements like `script` and `style` is returned as is, without further parsing:

```
>>> parser.feed('<style type="text/css">#python { color: green
}</style>')
Start tag: style
     attr: ('type', 'text/css')
Data     : #python { color: green }
End tag  : style

>>> parser.feed('<script type="text/javascript">'
...             'alert("<strong>hello! &#9786;</strong>");
</script>')
Start tag: script
     attr: ('type', 'text/javascript')
Data     : alert("<strong>hello! &#9786;</strong>");
End tag  : script
```

Parsing comments:

```
>>> parser.feed('<!--a comment-->'
...             '<!--[if IE 9]>IE-specific content<![endif]--
>')
Comment  : a comment
Comment  : [if IE 9]>IE-specific content<![endif]
```

Parsing named and numeric character references and converting them to the correct char (note: these 3 references are all equivalent to `'>'`):

```
>>> parser = MyHTMLParser()
>>> parser.feed('&gt;&#62;&#x3E;')
Data     : >>>

>>> parser = MyHTMLParser(convert_charrefs=False)
>>> parser.feed('&gt;&#62;&#x3E;')
Named ent: >
Num ent  : >
Num ent  : >
```

Feeding incomplete chunks to `feed()` works, but `handle_data()` might be called more than once if *convert_charrefs* is false:

```
>>> for chunk in ['<sp', 'an>buff', 'ered', ' text</s',
'pan>']:
...     parser.feed(chunk)
...
Start tag: span
Data     : buff
Data     : ered
Data     :  text
End tag  : span
```

Parsing invalid HTML (e.g. unquoted attributes) also works:

```
>>> parser.feed('<p><a class=link href=#main>tag soup</p >
</a>')
Start tag: p
Start tag: a
     attr: ('class', 'link')
     attr: ('href', '#main')
Data     : tag soup
End tag  : p
End tag  : a
```

`html.entities` — Definitions of HTML general entities

Source code: Lib/html/entities.py [https://github.com/python/cpython/tree/3.14/Lib/html/entities.py]

This module defines four dictionaries, **`html5`**, **`name2codepoint`**, **`codepoint2name`**, and **`entitydefs`**.

html.entities.html5

A dictionary that maps HTML5 named character references [[1]] to the equivalent Unicode character(s), e.g. `html5['gt;'] == '>'`. Note that the trailing semicolon is included in the name (e.g. `'gt;'`), however some of the names are accepted by the standard even without the semicolon: in this case the name is present with and without the `';'`. See also **`html.unescape()`**.

Added in version 3.3.

html.entities.entitydefs

A dictionary mapping XHTML 1.0 entity definitions to their replacement text in ISO Latin-1.

html.entities.name2codepoint

A dictionary that maps HTML4 entity names to the Unicode code points.

html.entities.codepoint2name

A dictionary that maps Unicode code points to HTML4 entity names.

Footnotes

[1]

See https://html.spec.whatwg.org/multipage/named-characters.html#named-character-references

XML Processing Modules

Source code: Lib/xml/
[https://github.com/python/cpython/tree/3.14/Lib/xml/]

Python’s interfaces for processing XML are grouped in the `xml` package.

Note

If you need to parse untrusted or unauthenticated data, see XML security.

It is important to note that modules in the **`xml`** package require that there be at least one SAX-compliant XML parser available. The Expat parser is included with Python, so the **`xml.parsers.expat`** module will always be available.

The documentation for the **`xml.dom`** and **`xml.sax`** packages are the definition of the Python bindings for the DOM and SAX interfaces.

The XML handling submodules are:

- **`xml.etree.ElementTree`**: the ElementTree API, a simple and lightweight XML processor

- **`xml.dom`**: the DOM API definition
- **`xml.dom.minidom`**: a minimal DOM implementation
- **`xml.dom.pulldom`**: support for building partial DOM trees

- **`xml.sax`**: SAX2 base classes and convenience functions

- `xml.parsers.expat`: the Expat parser binding

XML security

An attacker can abuse XML features to carry out denial of service attacks, access local files, generate network connections to other machines, or circumvent firewalls when attacker-controlled XML is being parsed, in Python or elsewhere.

The built-in XML parsers of Python rely on the library libexpat [https://github.com/libexpat/libexpat], commonly called Expat, for parsing XML.

By default, Expat itself does not access local files or create network connections.

Expat versions lower than 2.7.2 may be vulnerable to the “billion laughs”, “quadratic blowup” and “large tokens” vulnerabilities, or to disproportional use of dynamic memory. Python bundles a copy of Expat, and whether Python uses the bundled or a system-wide Expat, depends on how the Python interpreter `has been configured` in your environment. Python may be vulnerable if it uses such older versions of Expat. Check **`pyexpat.EXPAT_VERSION`**.

`xmlrpc` is **vulnerable** to the “decompression bomb” attack.

billion laughs / exponential entity expansion

> The Billion Laughs [https://en.wikipedia.org/wiki/Billion_laughs] attack – also known as exponential entity expansion – uses multiple levels of nested entities. Each entity refers to another entity several times, and the final entity definition contains a small string. The exponential expansion results in several gigabytes of text and consumes lots of memory and CPU time.

quadratic blowup entity expansion

A quadratic blowup attack is similar to a Billion Laughs [https://en.wikipedia.org/wiki/Billion_laughs] attack; it abuses entity expansion, too. Instead of nested entities it repeats one large entity with a couple of thousand chars over and over again. The attack isn't as efficient as the exponential case but it avoids triggering parser countermeasures that forbid deeply nested entities.

decompression bomb

Decompression bombs (aka ZIP bomb [https://en.wikipedia.org/wiki/Zip_bomb]) apply to all XML libraries that can parse compressed XML streams such as gzipped HTTP streams or LZMA-compressed files. For an attacker it can reduce the amount of transmitted data by three magnitudes or more.

large tokens

Expat needs to re-parse unfinished tokens; without the protection introduced in Expat 2.6.0, this can lead to quadratic runtime that can be used to cause denial of service in the application parsing XML. The issue is known as **CVE 2023-52425** [https://www.cve.org/CVERecord?id=CVE-2023-52425].

`xml.etree.ElementTree` — The ElementTree XML API

Source code: Lib/xml/etree/ElementTree.py [https://github.com/python/cpython/tree/3.14/Lib/xml/etree/ElementTree.py]

The **`xml.etree.ElementTree`** module implements a simple and efficient API for parsing and creating XML data.

Changed in version 3.3: This module will use a fast implementation whenever available.

Deprecated since version 3.3: The **`xml.etree.cElementTree`** module is deprecated.

Note

If you need to parse untrusted or unauthenticated data, see XML security.

Tutorial

This is a short tutorial for using **`xml.etree.ElementTree`** (ET in short). The goal is to demonstrate some of the building blocks and basic concepts of the module.

XML tree and elements

XML is an inherently hierarchical data format, and the most natural way to represent it is with a tree. ET has two classes for this purpose - `ElementTree` represents the whole XML document as a tree, and `Element` represents a single node in this tree. Interactions with the whole document (reading and writing to/from files) are usually done on the `ElementTree` level. Interactions with a single XML element and its sub-elements are done on the `Element` level.

Parsing XML

We'll be using the fictive `country_data.xml` XML document as the sample data for this section:

```
<?xml version="1.0"?>
<data>
    <country name="Liechtenstein">
        <rank>1</rank>
        <year>2008</year>
        <gdppc>141100</gdppc>
        <neighbor name="Austria" direction="E"/>
        <neighbor name="Switzerland" direction="W"/>
    </country>
    <country name="Singapore">
        <rank>4</rank>
        <year>2011</year>
        <gdppc>59900</gdppc>
        <neighbor name="Malaysia" direction="N"/>
    </country>
    <country name="Panama">
        <rank>68</rank>
        <year>2011</year>
        <gdppc>13600</gdppc>
        <neighbor name="Costa Rica" direction="W"/>
        <neighbor name="Colombia" direction="E"/>
    </country>
</data>
```

We can import this data by reading from a file:

```
import xml.etree.ElementTree as ET
tree = ET.parse('country_data.xml')
```

```
root = tree.getroot()
```

Or directly from a string:

```
root = ET.fromstring(country_data_as_string)
```

`fromstring()` parses XML from a string directly into an `Element`, which is the root element of the parsed tree. Other parsing functions may create an `ElementTree`. Check the documentation to be sure.

As an `Element`, `root` has a tag and a dictionary of attributes:

```
>>> root.tag
'data'
>>> root.attrib
{}
```

It also has children nodes over which we can iterate:

```
>>> for child in root:
...     print(child.tag, child.attrib)
...
country {'name': 'Liechtenstein'}
country {'name': 'Singapore'}
country {'name': 'Panama'}
```

Children are nested, and we can access specific child nodes by index:

```
>>> root[0][1].text
'2008'
```

Note

Not all elements of the XML input will end up as elements of the parsed tree. Currently, this module skips over any XML comments, processing instructions, and document type declarations in the input. Nevertheless, trees built using this module’s API rather than parsing from XML text can have comments and processing instructions in them; they will be included when generating XML output. A document type declaration may be accessed by passing a custom `TreeBuilder` instance to the `XMLParser` constructor.

Pull API for non-blocking parsing

Most parsing functions provided by this module require the whole document to be read at once before returning any result. It is possible to use an `XMLParser` and feed data into it incrementally, but it is a push API that calls methods on a callback target, which is too low-level and inconvenient for most needs. Sometimes what the user really wants is to be able to parse XML incrementally, without blocking operations, while enjoying the convenience of fully constructed `Element` objects.

The most powerful tool for doing this is `XMLPullParser`. It does not require a blocking read to obtain the XML data, and is instead fed with data incrementally with `XMLPullParser.feed()` calls. To get the parsed XML elements, call `XMLPullParser.read_events()`. Here is an example:

```
>>> parser = ET.XMLPullParser(['start', 'end'])
>>> parser.feed('<mytag>sometext')
>>> list(parser.read_events())
[('start', <Element 'mytag' at 0x7fa66db2be58>)]
>>> parser.feed(' more text</mytag>')
>>> for event, elem in parser.read_events():
...     print(event)
...     print(elem.tag, 'text=', elem.text)
...
end
mytag text= sometext more text
```

The obvious use case is applications that operate in a non-blocking fashion where the XML data is being received from a socket or read incrementally from some storage device. In such cases, blocking reads are unacceptable.

Because it’s so flexible, **`XMLPullParser`** can be inconvenient to use for simpler use-cases. If you don’t mind your application blocking on reading XML data but would still like to have incremental parsing capabilities, take a look at **`iterparse()`**. It can be useful when you’re reading a large XML document and don’t want to hold it wholly in memory.

Where *immediate* feedback through events is wanted, calling method **`XMLPullParser.flush()`** can help reduce delay; please make sure to study the related security notes.

Finding interesting elements

`Element` has some useful methods that help iterate recursively over all the sub-tree below it (its children, their children, and so on). For example, **`Element.iter()`**:

```
>>> for neighbor in root.iter('neighbor'):
...     print(neighbor.attrib)
...
{'name': 'Austria', 'direction': 'E'}
{'name': 'Switzerland', 'direction': 'W'}
{'name': 'Malaysia', 'direction': 'N'}
{'name': 'Costa Rica', 'direction': 'W'}
{'name': 'Colombia', 'direction': 'E'}
```

`Element.findall()` finds only elements with a tag which are direct children of the current element. **`Element.find()`** finds the *first* child with a particular tag, and **`Element.text`** accesses the element’s text content. **`Element.get()`** accesses the element’s attributes:

```
>>> for country in root.findall('country'):
...     rank = country.find('rank').text
...     name = country.get('name')
...     print(name, rank)
...
Liechtenstein 1
Singapore 4
Panama 68
```

More sophisticated specification of which elements to look for is possible by using XPath.

Modifying an XML File

`ElementTree` provides a simple way to build XML documents and write them to files. The **`ElementTree.write()`** method serves this purpose.

Once created, an **`Element`** object may be manipulated by directly changing its fields (such as **`Element.text`**), adding and modifying attributes (**`Element.set()`** method), as well as adding new children (for example with **`Element.append()`**).

Let's say we want to add one to each country's rank, and add an `updated` attribute to the rank element:

```
>>> for rank in root.iter('rank'):
...     new_rank = int(rank.text) + 1
...     rank.text = str(new_rank)
...     rank.set('updated', 'yes')
...
>>> tree.write('output.xml')
```

Our XML now looks like this:

```
<?xml version="1.0"?>
<data>
    <country name="Liechtenstein">
        <rank updated="yes">2</rank>
        <year>2008</year>
        <gdppc>141100</gdppc>
        <neighbor name="Austria" direction="E"/>
        <neighbor name="Switzerland" direction="W"/>
    </country>
    <country name="Singapore">
        <rank updated="yes">5</rank>
        <year>2011</year>
        <gdppc>59900</gdppc>
        <neighbor name="Malaysia" direction="N"/>
    </country>
    <country name="Panama">
        <rank updated="yes">69</rank>
        <year>2011</year>
        <gdppc>13600</gdppc>
        <neighbor name="Costa Rica" direction="W"/>
        <neighbor name="Colombia" direction="E"/>
    </country>
</data>
```

We can remove elements using `Element.remove()`. Let’s say we want to remove all countries with a rank higher than 50:

```
>>> for country in root.findall('country'):
...     # using root.findall() to avoid removal during
traversal
...     rank = int(country.find('rank').text)
...     if rank > 50:
...         root.remove(country)
...
>>> tree.write('output.xml')
```

Note that concurrent modification while iterating can lead to problems, just like when iterating and modifying Python lists or dicts. Therefore, the example first collects all matching elements with `root.findall()`, and only then iterates over the list of matches.

Our XML now looks like this:

```
<?xml version="1.0"?>
<data>
    <country name="Liechtenstein">
        <rank updated="yes">2</rank>
        <year>2008</year>
        <gdppc>141100</gdppc>
        <neighbor name="Austria" direction="E"/>
        <neighbor name="Switzerland" direction="W"/>
    </country>
    <country name="Singapore">
        <rank updated="yes">5</rank>
        <year>2011</year>
        <gdppc>59900</gdppc>
        <neighbor name="Malaysia" direction="N"/>
    </country>
</data>
```

Building XML documents

The `SubElement()` function also provides a convenient way to create new sub-elements for a given element:

```
>>> a = ET.Element('a')
>>> b = ET.SubElement(a, 'b')
>>> c = ET.SubElement(a, 'c')
>>> d = ET.SubElement(c, 'd')
>>> ET.dump(a)
<a><b /><c><d /></c></a>
```

Parsing XML with Namespaces

If the XML input has namespaces [https://en.wikipedia.org/wiki/XML_namespace], tags and attributes with prefixes in the form `prefix:sometag` get expanded to `{uri}sometag` where the *prefix* is replaced by the full *URI*. Also, if there is a default namespace [https://www.w3.org/TR/xml-names/#defaulting], that full URI gets prepended to all of the non-prefixed tags.

Here is an XML example that incorporates two namespaces, one with the prefix “fictional” and the other serving as the default namespace:

```
<?xml version="1.0"?>
<actors xmlns:fictional="http://characters.example.com"
        xmlns="http://people.example.com">
    <actor>
        <name>John Cleese</name>
        <fictional:character>Lancelot</fictional:character>
        <fictional:character>Archie Leach</fictional:character>
    </actor>
    <actor>
        <name>Eric Idle</name>
        <fictional:character>Sir Robin</fictional:character>
        <fictional:character>Gunther</fictional:character>
        <fictional:character>Commander
Clement</fictional:character>
    </actor>
</actors>
```

One way to search and explore this XML example is to manually add the URI to every tag or attribute in the xpath of a `find()` or `findall()`:

```
root = fromstring(xml_text)
for actor in root.findall('{http://people.example.com}actor'):
    name = actor.find('{http://people.example.com}name')
    print(name.text)
    for char in
actor.findall('{http://characters.example.com}character'):
        print(' |-->', char.text)
```

A better way to search the namespaced XML example is to create a dictionary with your own prefixes and use those in the search functions:

```
ns = {'real_person': 'http://people.example.com',
      'role': 'http://characters.example.com'}

for actor in root.findall('real_person:actor', ns):
    name = actor.find('real_person:name', ns)
    print(name.text)
    for char in actor.findall('role:character', ns):
        print(' |-->', char.text)
```

These two approaches both output:

```
John Cleese
 |--> Lancelot
 |--> Archie Leach
Eric Idle
 |--> Sir Robin
 |--> Gunther
 |--> Commander Clement
```

XPath support

This module provides limited support for XPath expressions [https://www.w3.org/TR/xpath] for locating elements in a tree. The goal is to support a small subset of the abbreviated syntax; a full XPath engine is outside the scope of the module.

Example

Here's an example that demonstrates some of the XPath capabilities of the module. We'll be using the `countrydata` XML document from the Parsing XML section:

```
import xml.etree.ElementTree as ET

root = ET.fromstring(countrydata)

# Top-level elements
root.findall(".")

# All 'neighbor' grand-children of 'country' children of the
top-level
# elements
root.findall("./country/neighbor")

# Nodes with name='Singapore' that have a 'year' child
root.findall(".//year/..[@name='Singapore']")

# 'year' nodes that are children of nodes with name='Singapore'
```

```
root.findall(".//*[@name='Singapore']/year")

# All 'neighbor' nodes that are the second child of their
parent
root.findall(".//neighbor[2]")
```

For XML with namespaces, use the usual qualified `{namespace}tag` notation:

```
# All dublin-core "title" tags in the document
root.findall(".//{http://purl.org/dc/elements/1.1/}title")
```

Supported XPath syntax

Syntax	Meaning
`tag`	Selects all child elements with the given tag. For example, `spam` selects all child elements named `spam`, and `spam/egg` selects all grandchildren named `egg` in all children named `spam`. `{namespace}*` selects all tags in the given namespace, `{*}spam` selects tags named `spam` in any (or no) namespace, and `{}*` only selects tags that are not in a namespace. *Changed in version 3.8:* Support for star-wildcards was added.
`*`	Selects all child elements, including comments and processing instructions. For example, `*/egg` selects all grandchildren named `egg`.

Syntax	Meaning
`.`	Selects the current node. This is mostly useful at the beginning of the path, to indicate that it’s a relative path.
`//`	Selects all subelements, on all levels beneath the current element. For example, `.//egg` selects all `egg` elements in the entire tree.
`..`	Selects the parent element. Returns `None` if the path attempts to reach the ancestors of the start element (the element `find` was called on).
`[@attrib]`	Selects all elements that have the given attribute.
`[@attrib='value']`	Selects all elements for which the given attribute has the given value. The value cannot contain quotes.
`[@attrib!='value']`	Selects all elements for which the given attribute does not have the given value. The value cannot contain quotes. *Added in version 3.10.*

Syntax	Meaning
`[tag]`	Selects all elements that have a child named `tag`. Only immediate children are supported.
`[.='text']`	Selects all elements whose complete text content, including descendants, equals the given `text`. *Added in version 3.7.*
`[.!='text']`	Selects all elements whose complete text content, including descendants, does not equal the given `text`. *Added in version 3.10.*
`[tag='text']`	Selects all elements that have a child named `tag` whose complete text content, including descendants, equals the given `text`.
`[tag!='text']`	Selects all elements that have a child named `tag` whose complete text content, including descendants, does not equal the given `text`. *Added in version 3.10.*

Syntax	Meaning
`[position]`	Selects all elements that are located at the given position. The position can be either an integer (1 is the first position), the expression `last()` (for the last position), or a position relative to the last position (e.g. `last()-1`).

Predicates (expressions within square brackets) must be preceded by a tag name, an asterisk, or another predicate. `position` predicates must be preceded by a tag name.

Reference

Functions

xml.etree.ElementTree.canonicalize(*xml_data=None*, *, *out=None*, *from_file=None*, ***options*)

C14N 2.0 [https://www.w3.org/TR/xml-c14n2/] transformation function.

Canonicalization is a way to normalise XML output in a way that allows byte-by-byte comparisons and digital signatures. It reduces the freedom that XML serializers have and instead generates a more constrained XML representation. The main restrictions regard the placement of namespace declarations, the ordering of attributes, and ignorable whitespace.

This function takes an XML data string (*xml_data*) or a file path or file-like object (*from_file*) as input, converts it to the canonical form, and writes it out using the *out* file(-

like) object, if provided, or returns it as a text string if not. The output file receives text, not bytes. It should therefore be opened in text mode with `utf-8` encoding.

Typical uses:

```
xml_data = "<root>...</root>"
print(canonicalize(xml_data))

with open("c14n_output.xml", mode='w', encoding='utf-8') as
out_file:
    canonicalize(xml_data, out=out_file)

with open("c14n_output.xml", mode='w', encoding='utf-8') as
out_file:
    canonicalize(from_file="inputfile.xml", out=out_file)
```

The configuration *options* are as follows:

- *with_comments*: set to true to include comments (default: false)
- *strip_text*: set to true to strip whitespace before and after text content
  (default: false)
- *rewrite_prefixes*: set to true to replace namespace prefixes by “n{number}”
  (default: false)
- *qname_aware_tags*: a set of qname aware tag names in which prefixes
  should be replaced in text content (default: empty)
- *qname_aware_attrs*: a set of qname aware attribute names in which prefixes
  should be replaced in text content (default: empty)

- *exclude_attrs*: a set of attribute names that should not be serialised
- *exclude_tags*: a set of tag names that should not be serialised

In the option list above, “a set” refers to any collection or iterable of strings, no ordering is expected.

Added in version 3.8.

xml.etree.ElementTree.Comment(*text=None*)

Comment element factory. This factory function creates a special element that will be serialized as an XML comment by the standard serializer. The comment string can be either a bytestring or a Unicode string. *text* is a string containing the comment string. Returns an element instance representing a comment.

Note that **`XMLParser`** skips over comments in the input instead of creating comment objects for them. An **`ElementTree`** will only contain comment nodes if they have been inserted into to the tree using one of the **`Element`** methods.

xml.etree.ElementTree.dump(*elem*)

Writes an element tree or element structure to sys.stdout. This function should be used for debugging only.

The exact output format is implementation dependent. In this version, it’s written as an ordinary XML file.

elem is an element tree or an individual element.

Changed in version 3.8: The **`dump()`** function now preserves the attribute order specified by the user.

xml.etree.ElementTree.fromstring(*text*, *parser=None*)

Parses an XML section from a string constant. Same as `XML()`. *text* is a string containing XML data. *parser* is an optional parser instance. If not given, the standard `XMLParser` parser is used. Returns an `Element` instance.

xml.etree.ElementTree.fromstringlist(*sequence*, *parser=None*)

Parses an XML document from a sequence of string fragments. *sequence* is a list or other sequence containing XML data fragments. *parser* is an optional parser instance. If not given, the standard `XMLParser` parser is used. Returns an `Element` instance.

Added in version 3.2.

xml.etree.ElementTree.indent(*tree*, *space=' '*, *level=0*)

Appends whitespace to the subtree to indent the tree visually. This can be used to generate pretty-printed XML output. *tree* can be an Element or ElementTree. *space* is the whitespace string that will be inserted for each indentation level, two space characters by default. For indenting partial subtrees inside of an already indented tree, pass the initial indentation level as *level*.

Added in version 3.9.

xml.etree.ElementTree.iselement(*element*)

Check if an object appears to be a valid element object. *element* is an element instance. Return `True` if this is an element object.

xml.etree.ElementTree.iterparse(*source*, *events=None*, *parser=None*)

Parses an XML section into an element tree incrementally, and reports what's going on to the user. *source* is a filename or file object containing XML data. *events* is a sequence of events to report back. The supported events are the strings `"start"`, `"end"`, `"comment"`, `"pi"`, `"start-ns"` and `"end-ns"` (the “ns” events are used to get detailed namespace information). If *events* is omitted, only `"end"` events are reported. *parser* is an optional parser instance. If not given, the standard **`XMLParser`** parser is used. *parser* must be a subclass of **`XMLParser`** and can only use the default **`TreeBuilder`** as a target. Returns an iterator providing `(event, elem)` pairs; it has a `root` attribute that references the root element of the resulting XML tree once *source* is fully read. The iterator has the **`close()`** method that closes the internal file object if *source* is a filename.

Note that while **`iterparse()`** builds the tree incrementally, it issues blocking reads on *source* (or the file it names). As such, it's unsuitable for applications where blocking reads can't be made. For fully non-blocking parsing, see **`XMLPullParser`**.

Note

`iterparse()` only guarantees that it has seen the “>” character of a starting tag when it emits a “start” event, so the attributes are defined, but the contents of the text and tail attributes are undefined at that point. The same applies to the element children; they may or may not be present.

If you need a fully populated element, look for “end” events instead.

Deprecated since version 3.4: The *parser* argument.

Changed in version 3.8: The `comment` and `pi` events were added.

Changed in version 3.13: Added the **`close()`** method.

xml.etree.ElementTree.parse(*source*, *parser=None*)

Parses an XML section into an element tree. *source* is a filename or file object containing XML data. *parser* is an optional parser instance. If not given, the standard **`XMLParser`** parser is used. Returns an **`ElementTree`** instance.

xml.etree.ElementTree.ProcessingInstruction(*target*, *text=None*)

PI element factory. This factory function creates a special element that will be serialized as an XML processing instruction. *target* is a string containing the PI target. *text* is a string containing the PI contents, if given. Returns an element instance, representing a processing instruction.

Note that **`XMLParser`** skips over processing instructions in the input instead of creating PI objects for them. An **`ElementTree`** will only contain processing instruction nodes if they have been inserted into to the tree using one of the **`Element`** methods.

xml.etree.ElementTree.register_namespace(*prefix*, *uri*)

Registers a namespace prefix. The registry is global, and any existing mapping for either the given prefix or the namespace URI will be removed. *prefix* is a namespace prefix. *uri* is a namespace uri. Tags and attributes in this namespace will be serialized with the given prefix, if at all possible.

Added in version 3.2.

xml.etree.ElementTree.SubElement(*parent*, *tag*, *attrib={}*, ***extra*)

Subelement factory. This function creates an element instance, and appends it to an existing element.

The element name, attribute names, and attribute values can be either bytestrings or Unicode strings. *parent* is the parent element. *tag* is the subelement name. *attrib* is an optional dictionary, containing element attributes. *extra* contains additional attributes, given as keyword arguments. Returns an element instance.

xml.etree.ElementTree.tostring(*element*, *encoding='us-ascii'*, *method='xml'*, ***, *xml_declaration=None*, *default_namespace=None*, *short_empty_elements=True*)

Generates a string representation of an XML element, including all subelements. *element* is an **`Element`** instance. *encoding* [[1]] is the output encoding (default is US-ASCII). Use `encoding="unicode"` to generate a Unicode string (otherwise, a bytestring is generated). *method* is either `"xml"`, `"html"` or `"text"` (default is `"xml"`). *xml_declaration*, *default_namespace* and *short_empty_elements* has the same meaning as in **`ElementTree.write()`**. Returns an (optionally) encoded string containing the XML data.

Changed in version 3.4: Added the *short_empty_elements* parameter.

Changed in version 3.8: Added the *xml_declaration* and *default_namespace* parameters.

Changed in version 3.8: The **`tostring()`** function now preserves the attribute order specified by the user.

xml.etree.ElementTree.tostringlist(*element*, *encoding='us-ascii'*, *method='xml'*, *, *xml_declaration=None*, *default_namespace=None*, *short_empty_elements=True*)

Generates a string representation of an XML element, including all subelements. *element* is an **`Element`** instance. *encoding* [[1]] is the output encoding (default is US-ASCII). Use `encoding="unicode"` to generate a Unicode string (otherwise, a bytestring is generated). *method* is either `"xml"`, `"html"` or `"text"` (default is `"xml"`). *xml_declaration*, *default_namespace* and *short_empty_elements* has the same meaning as in **`ElementTree.write()`**. Returns a list of (optionally) encoded strings containing the XML data. It does not guarantee any specific sequence, except that `b"".join(tostringlist(element)) == tostring(element)`.

Added in version 3.2.

Changed in version 3.4: Added the *short_empty_elements* parameter.

Changed in version 3.8: Added the *xml_declaration* and *default_namespace* parameters.

Changed in version 3.8: The **`tostringlist()`** function now preserves the attribute order specified by the user.

xml.etree.ElementTree.XML(*text*, *parser=None*)

Parses an XML section from a string constant. This function can be used to embed “XML literals” in Python code. *text* is a string containing XML data. *parser* is an optional parser instance. If not given, the standard **`XMLParser`** parser is used. Returns an **`Element`** instance.

xml.etree.ElementTree.XMLID(*text*, *parser=None*)

Parses an XML section from a string constant, and also returns a dictionary which maps from element id:s to elements. *text* is a string containing XML data. *parser* is an optional parser instance. If not given, the standard `XMLParser` parser is used. Returns a tuple containing an `Element` instance and a dictionary.

XInclude support

This module provides limited support for XInclude directives [https://www.w3.org/TR/xinclude/], via the `xml.etree.ElementInclude` helper module. This module can be used to insert subtrees and text strings into element trees, based on information in the tree.

Example

Here's an example that demonstrates use of the XInclude module. To include an XML document in the current document, use the `{http://www.w3.org/2001/XInclude}include` element and set the **parse** attribute to `"xml"`, and use the **href** attribute to specify the document to include.

```
<?xml version="1.0"?>
<document xmlns:xi="http://www.w3.org/2001/XInclude">
  <xi:include href="source.xml" parse="xml" />
</document>
```

By default, the **href** attribute is treated as a file name. You can use custom loaders to override this behaviour. Also note that the standard helper does not support XPointer syntax.

To process this file, load it as usual, and pass the root element to the `xml.etree.ElementTree` module:

```
from xml.etree import ElementTree, ElementInclude
```

```
tree = ElementTree.parse("document.xml")
root = tree.getroot()

ElementInclude.include(root)
```

The ElementInclude module replaces the `{http://www.w3.org/2001/XInclude}include` element with the root element from the **source.xml** document. The result might look something like this:

```
<document xmlns:xi="http://www.w3.org/2001/XInclude">
  <para>This is a paragraph.</para>
</document>
```

If the **parse** attribute is omitted, it defaults to “xml”. The href attribute is required.

To include a text document, use the `{http://www.w3.org/2001/XInclude}include` element, and set the **parse** attribute to “text”:

```
<?xml version="1.0"?>
<document xmlns:xi="http://www.w3.org/2001/XInclude">
  Copyright (c) <xi:include href="year.txt" parse="text" />.
</document>
```

The result might look something like:

```
<document xmlns:xi="http://www.w3.org/2001/XInclude">
  Copyright (c) 2003.
</document>
```

Reference

Functions

xml.etree.ElementInclude.default_loader(*href*, *parse*, *encoding=None*)

Default loader. This default loader reads an included resource from disk. *href* is a URL. *parse* is for parse mode either “xml” or “text”. *encoding* is an optional text encoding. If not given, encoding is `utf-8`. Returns the expanded resource. If the parse mode is `"xml"`, this is an **`Element`** instance. If the parse mode is `"text"`, this is a string. If the loader fails, it can return `None` or raise an exception.

xml.etree.ElementInclude.include(*elem*, *loader=None*, *base_url=None*, *max_depth=6*)

This function expands XInclude directives in-place in tree pointed by *elem*. *elem* is either the root **`Element`** or an **`ElementTree`** instance to find such element. *loader* is an optional resource loader. If omitted, it defaults to **`default_loader()`**. If given, it should be a callable that implements the same interface as **`default_loader()`**. *base_url* is base URL of the original file, to resolve relative include file references. *max_depth* is the maximum number of recursive inclusions. Limited to reduce the risk of malicious content explosion. Pass `None` to disable the limitation.

Changed in version 3.9: Added the *base_url* and *max_depth* parameters.

Element Objects

class xml.etree.ElementTree.Element(*tag*, *attrib={}*, ***extra*)

Element class. This class defines the Element interface, and provides a reference implementation of this interface.

The element name, attribute names, and attribute values can be either bytestrings or Unicode strings. *tag* is the element name. *attrib* is an optional dictionary, containing element attributes. *extra* contains additional attributes, given as keyword arguments.

tag

A string identifying what kind of data this element represents (the element type, in other words).

text
tail

These attributes can be used to hold additional data associated with the element. Their values are usually strings but may be any application-specific object. If the element is created from an XML file, the *text* attribute holds either the text between the element's start tag and its first child or end tag, or `None`, and the *tail* attribute holds either the text between the element's end tag and the next tag, or `None`. For the XML data

```
<a><b>1<c>2<d/>3</c></b>4</a>
```

the *a* element has `None` for both *text* and *tail* attributes, the *b* element has *text* `"1"` and *tail* `"4"`, the *c* element has *text* `"2"` and *tail* `None`, and the *d* element has *text* `None` and *tail* `"3"`.

To collect the inner text of an element, see **`itertext()`**, for example `"".join(element.itertext())`.

Applications may store arbitrary objects in these attributes.

attrib

A dictionary containing the element’s attributes. Note that while the *attrib* value is always a real mutable Python dictionary, an ElementTree implementation may choose to use another internal representation, and create the dictionary only if someone asks for it. To take advantage of such implementations, use the dictionary methods below whenever possible.

The following dictionary-like methods work on the element attributes.

clear()

Resets an element. This function removes all subelements, clears all attributes, and sets the text and tail attributes to `None`.

get(*key*, *default=None*)

Gets the element attribute named *key*.

Returns the attribute value, or *default* if the attribute was not found.

items()

Returns the element attributes as a sequence of (name, value) pairs. The attributes are returned in an arbitrary order.

keys()

Returns the elements attribute names as a list. The names are returned in an arbitrary order.

set(*key*, *value*)

Set the attribute *key* on the element to *value*.

The following methods work on the element’s children (subelements).

append(*subelement*)

Adds the element *subelement* to the end of this element's internal list of subelements. Raises **`TypeError`** if *subelement* is not an **`Element`**.

extend(*subelements*)

Appends *subelements* from an iterable of elements. Raises **`TypeError`** if a subelement is not an **`Element`**.

Added in version 3.2.

find(*match*, *namespaces=None*)

Finds the first subelement matching *match*. *match* may be a tag name or a path. Returns an element instance or `None`. *namespaces* is an optional mapping from namespace prefix to full name. Pass `''` as prefix to move all unprefixed tag names in the expression into the given namespace.

findall(*match*, *namespaces=None*)

Finds all matching subelements, by tag name or path. Returns a list containing all matching elements in document order. *namespaces* is an optional mapping from namespace prefix to full name. Pass `''` as prefix to move all unprefixed tag names in the expression into the given namespace.

findtext(*match*, *default=None*, *namespaces=None*)

Finds text for the first subelement matching *match*. *match* may be a tag name or a path. Returns the text content of the first matching element, or *default* if no element was found. Note that if the matching element has no text content an empty string is returned. *namespaces* is an optional mapping from namespace prefix to full name. Pass `''` as prefix to move all unprefixed tag names in the expression into the given namespace.

insert(*index*, *subelement*)

Inserts *subelement* at the given position in this element. Raises **`TypeError`** if *subelement* is not an **`Element`**.

iter(*tag=None*)

Creates a tree iterator with the current element as the root. The iterator iterates over this element and all elements below it, in document (depth first) order. If *tag* is not `None` or `'*'`, only elements whose tag equals *tag* are returned from the iterator. If the tree structure is modified during iteration, the result is undefined.

Added in version 3.2.

iterfind(*match*, *namespaces=None*)

Finds all matching subelements, by tag name or path. Returns an iterable yielding all matching elements in document order. *namespaces* is an optional mapping from namespace prefix to full name.

Added in version 3.2.

itertext()

Creates a text iterator. The iterator loops over this element and all subelements, in document order, and returns all inner text.

Added in version 3.2.

makeelement(*tag*, *attrib*)

Creates a new element object of the same type as this element. Do not call this method, use the **`SubElement()`**

factory function instead.

remove(*subelement*)

Removes *subelement* from the element. Unlike the find* methods this method compares elements based on the instance identity, not on tag value or contents.

`Element` objects also support the following sequence type methods for working with subelements: **`__delitem__()`**, **`__getitem__()`**, **`__setitem__()`**, **`__len__()`**.

Caution: Elements with no subelements will test as `False`. In a future release of Python, all elements will test as `True` regardless of whether subelements exist. Instead, prefer explicit `len(elem)` or `elem is not None` tests.:

```
element = root.find('foo')

if not element:  # careful!
    print("element not found, or element has no
subelements")

if element is None:
    print("element not found")
```

Changed in version 3.12: Testing the truth value of an Element emits **`DeprecationWarning`**.

Prior to Python 3.8, the serialisation order of the XML attributes of elements was artificially made predictable by sorting the attributes by their name. Based on the now guaranteed ordering of dicts, this arbitrary reordering was removed in Python 3.8 to preserve the order in which attributes were originally parsed or created by user code.

In general, user code should try not to depend on a specific ordering of attributes, given that the XML

Information Set [https://www.w3.org/TR/xml-infoset/] explicitly excludes the attribute order from conveying information. Code should be prepared to deal with any ordering on input. In cases where deterministic XML output is required, e.g. for cryptographic signing or test data sets, canonical serialisation is available with the **`canonicalize()`** function.

In cases where canonical output is not applicable but a specific attribute order is still desirable on output, code should aim for creating the attributes directly in the desired order, to avoid perceptual mismatches for readers of the code. In cases where this is difficult to achieve, a recipe like the following can be applied prior to serialisation to enforce an order independently from the Element creation:

```
def reorder_attributes(root):
    for el in root.iter():
        attrib = el.attrib
        if len(attrib) > 1:
            # adjust attribute order, e.g. by sorting
            attribs = sorted(attrib.items())
            attrib.clear()
            attrib.update(attribs)
```

ElementTree Objects

class xml.etree.ElementTree.ElementTree(*element=None*, *file=None*)

ElementTree wrapper class. This class represents an entire element hierarchy, and adds some extra support for serialization to and from standard XML.

element is the root element. The tree is initialized with the contents of the XML *file* if given.

_setroot(*element*)

Replaces the root element for this tree. This discards the current contents of the tree, and replaces it with the given element. Use with care. *element* is an element instance.

find(*match*, *namespaces=None*)

Same as `Element.find()`, starting at the root of the tree.

findall(*match*, *namespaces=None*)

Same as `Element.findall()`, starting at the root of the tree.

findtext(*match*, *default=None*, *namespaces=None*)

Same as `Element.findtext()`, starting at the root of the tree.

getroot()

Returns the root element for this tree.

iter(*tag=None*)

Creates and returns a tree iterator for the root element. The iterator loops over all elements in this tree, in section order. *tag* is the tag to look for (default is to return all elements).

iterfind(*match*, *namespaces=None*)

Same as `Element.iterfind()`, starting at the root of the tree.

Added in version 3.2.

parse(*source*, *parser=None*)

Loads an external XML section into this element tree. *source* is a file name or file object. *parser* is an optional parser instance. If not given, the standard `XMLParser` parser is used. Returns the section root element.

write(*file*, *encoding='us-ascii'*, *xml_declaration=None*, *default_namespace=None*, *method='xml'*, ***, *short_empty_elements=True*)

Writes the element tree to a file, as XML. *file* is a file name, or a file object opened for writing. *encoding* [[1]] is the output encoding (default is US-ASCII). *xml_declaration* controls if an XML declaration should be added to the file. Use `False` for never, `True` for always, `None` for only if not US-ASCII or UTF-8 or Unicode (default is `None`). *default_namespace* sets the default XML namespace (for “xmlns”). *method* is either `"xml"`, `"html"` or `"text"` (default is `"xml"`). The keyword-only *short_empty_elements* parameter controls the formatting of elements that contain no content. If `True` (the default), they are emitted as a single self-closed tag, otherwise they are emitted as a pair of start/end tags.

The output is either a string (`str`) or binary (`bytes`). This is controlled by the *encoding* argument. If *encoding* is `"unicode"`, the output is a string; otherwise, it’s binary. Note that this may conflict with the type of *file* if it’s an open file object; make sure you do not try to write a string to a binary stream and vice versa.

Changed in version 3.4: Added the *short_empty_elements* parameter.

Changed in version 3.8: The `write()` method now preserves the attribute order specified by the user.

This is the XML file that is going to be manipulated:

```
<html>
    <head>
        <title>Example page</title>
    </head>
    <body>
        <p>Moved to <a
href="http://example.org/">example.org</a>
        or <a href="http://example.com/">example.com</a>.</p>
    </body>
</html>
```

Example of changing the attribute “target” of every link in first paragraph:

```
>>> from xml.etree.ElementTree import ElementTree
>>> tree = ElementTree()
>>> tree.parse("index.xhtml")
<Element 'html' at 0xb77e6fac>
>>> p = tree.find("body/p")     # Finds first occurrence of tag
p in body
>>> p
<Element 'p' at 0xb77ec26c>
>>> links = list(p.iter("a"))   # Returns list of all links
>>> links
[<Element 'a' at 0xb77ec2ac>, <Element 'a' at 0xb77ec1cc>]
>>> for i in links:             # Iterates through all found
links
...     i.attrib["target"] = "blank"
...
>>> tree.write("output.xhtml")
```

QName Objects

class xml.etree.ElementTree.QName(*text_or_uri*, *tag=None*)

QName wrapper. This can be used to wrap a QName attribute value, in order to get proper namespace handling on output. *text_or_uri* is a string containing the QName value, in the form {uri}local, or, if the tag argument is given, the URI part of a QName. If *tag* is given, the first argument is interpreted as a URI, and this argument is interpreted as a local name. `QName` instances are opaque.

TreeBuilder Objects

class xml.etree.ElementTree.TreeBuilder(*element_factory=None*, *, *comment_factory=None*, *pi_factory=None*, *insert_comments=False*, *insert_pis=False*)

Generic element structure builder. This builder converts a sequence of start, data, end, comment and pi method calls to a well-formed element structure. You can use this class to build an element structure using a custom XML parser, or a parser for some other XML-like format.

element_factory, when given, must be a callable accepting two positional arguments: a tag and a dict of attributes. It is expected to return a new element instance.

The *comment_factory* and *pi_factory* functions, when given, should behave like the `Comment()` and `ProcessingInstruction()` functions to create comments and processing instructions. When not given, the default factories will be used. When *insert_comments* and/or *insert_pis* is true, comments/pis will be inserted into the tree if they appear within the root element (but not outside of it).

close()

Flushes the builder buffers, and returns the toplevel document element. Returns an `Element` instance.

data(*data*)

Adds text to the current element. *data* is a string. This should be either a bytestring, or a Unicode string.

end(*tag*)

Closes the current element. *tag* is the element name. Returns the closed element.

start(*tag*, *attrs*)

Opens a new element. *tag* is the element name. *attrs* is a dictionary containing element attributes. Returns the opened element.

comment(*text*)

Creates a comment with the given *text*. If `insert_comments` is true, this will also add it to the tree.

Added in version 3.8.

pi(*target*, *text*)

Creates a process instruction with the given *target* name and *text*. If `insert_pis` is true, this will also add it to the tree.

Added in version 3.8.

In addition, a custom **`TreeBuilder`** object can provide the following methods:

doctype(*name*, *pubid*, *system*)

Handles a doctype declaration. *name* is the doctype name. *pubid* is the public identifier. *system* is the system identifier. This method does not exist on the default **`TreeBuilder`** class.

Added in version 3.2.

start_ns(*prefix*, *uri*)

Is called whenever the parser encounters a new namespace declaration, before the `start()` callback for the opening element that defines it. *prefix* is `''` for the default namespace and the declared namespace prefix name otherwise. *uri* is the namespace URI.

Added in version 3.8.

end_ns(*prefix*)

Is called after the `end()` callback of an element that declared a namespace prefix mapping, with the name of the *prefix* that went out of scope.

Added in version 3.8.

class xml.etree.ElementTree.C14NWriterTarget(*write*, *, *with_comments=False*, *strip_text=False*, *rewrite_prefixes=False*, *qname_aware_tags=None*, *qname_aware_attrs=None*, *exclude_attrs=None*, *exclude_tags=None*)

A C14N 2.0 [https://www.w3.org/TR/xml-c14n2/] writer. Arguments are the same as for the `canonicalize()` function. This class does not build a tree but translates the callback events directly into a serialised form using the *write* function.

Added in version 3.8.

XMLParser Objects

class xml.etree.ElementTree.XMLParser(*, *target=None*, *encoding=None*)

This class is the low-level building block of the module. It uses `xml.parsers.expat` for efficient, event-based parsing of XML. It can be fed XML data incrementally with the `feed()` method, and parsing events are translated to a push API - by invoking callbacks on the *target* object. If *target* is omitted, the standard `TreeBuilder` is used. If *encoding* [[1]] is given, the value overrides the encoding specified in the XML file.

Changed in version 3.8: Parameters are now keyword-only. The *html* argument is no longer supported.

close()

Finishes feeding data to the parser. Returns the result of calling the `close()` method of the *target* passed during construction; by default, this is the toplevel document element.

feed(*data*)

Feeds data to the parser. *data* is encoded data.

flush()

Triggers parsing of any previously fed unparsed data, which can be used to ensure more immediate feedback, in particular with Expat >=2.6.0. The implementation of `flush()` temporarily disables reparse deferral with Expat (if currently enabled) and triggers a reparse. Disabling reparse deferral has security consequences; please see `xml.parsers.expat.xmlparser.SetReparseDeferralEnabled()` for details.

Note that `flush()` has been backported to some prior releases of CPython as a security fix. Check for availability of `flush()` using `hasattr()` if used in code running across a variety of Python versions.

Added in version 3.13.

`XMLParser.feed()` calls *target*'s `start(tag, attrs_dict)` method for each opening tag, its `end(tag)` method for each closing tag, and data is processed by method `data(data)`. For further supported callback methods, see the **`TreeBuilder`** class. **`XMLParser.close()`** calls *target*'s method `close()`. **`XMLParser`** can be used not only for building a tree structure. This is an example of counting the maximum depth of an XML file:

```
>>> from xml.etree.ElementTree import XMLParser
>>> class MaxDepth:                     # The target object
of the parser
...     maxDepth = 0
...     depth = 0
...     def start(self, tag, attrib):   # Called for each
opening tag.
...         self.depth += 1
...         if self.depth > self.maxDepth:
...             self.maxDepth = self.depth
...     def end(self, tag):             # Called for each
closing tag.
...         self.depth -= 1
...     def data(self, data):
...         pass            # We do not need to do anything
with data.
...     def close(self):    # Called when all data has been
parsed.
...         return self.maxDepth
...
>>> target = MaxDepth()
>>> parser = XMLParser(target=target)
>>> exampleXml = """
... <a>
...   <b>
...   </b>
...   <b>
...     <c>
...       <d>
...       </d>
...     </c>
...   </b>
... </a>"""
>>> parser.feed(exampleXml)
>>> parser.close()
4
```

XMLPullParser Objects

class xml.etree.ElementTree.XMLPullParser(*events=None*)

A pull parser suitable for non-blocking applications. Its input-side API is similar to that of `XMLParser`, but instead of pushing calls to a callback target, `XMLPullParser` collects an internal list of parsing events and lets the user read from it. *events* is a sequence of events to report back. The supported events are the strings `"start"`, `"end"`, `"comment"`, `"pi"`, `"start-ns"` and `"end-ns"` (the “ns” events are used to get detailed namespace information). If *events* is omitted, only `"end"` events are reported.

feed(*data*)

Feed the given bytes data to the parser.

flush()

Triggers parsing of any previously fed unparsed data, which can be used to ensure more immediate feedback, in particular with Expat >=2.6.0. The implementation of `flush()` temporarily disables reparse deferral with Expat (if currently enabled) and triggers a reparse. Disabling reparse deferral has security consequences; please see `xml.parsers.expat.xmlparser.SetReparseDeferralEnabled()` for details.

Note that `flush()` has been backported to some prior releases of CPython as a security fix. Check for availability of `flush()` using `hasattr()` if used in code running across a variety of Python versions.

Added in version 3.13.

close()

Signal the parser that the data stream is terminated. Unlike `XMLParser.close()`, this method always returns `None`. Any events not yet retrieved when the parser is closed can still be read with `read_events()`.

read_events()

Return an iterator over the events which have been encountered in the data fed to the parser. The iterator yields `(event, elem)` pairs, where *event* is a string representing the type of event (e.g. `"end"`) and *elem* is the encountered `Element` object, or other context value as follows.

- `start`, `end`: the current Element.
- `comment`, `pi`: the current comment / processing instruction
- `start-ns`: a tuple `(prefix, uri)` naming the declared namespace mapping.
- `end-ns`: `None` (this may change in a future version)

Events provided in a previous call to `read_events()` will not be yielded again. Events are consumed from the internal queue only when they are retrieved from the iterator, so multiple readers iterating in parallel over iterators obtained from `read_events()` will have unpredictable results.

Note

`XMLPullParser` only guarantees that it has seen the “>” character of a starting tag when it emits a “start” event, so the attributes are defined, but the contents of the text and tail attributes are undefined at that point.

The same applies to the element children; they may or may not be present.

If you need a fully populated element, look for “end” events instead.

Added in version 3.4.

Changed in version 3.8: The `comment` and `pi` events were added.

Exceptions

class xml.etree.ElementTree.ParseError

XML parse error, raised by the various parsing methods in this module when parsing fails. The string representation of an instance of this exception will contain a user-friendly error message. In addition, it will have the following attributes available:

code

A numeric error code from the expat parser. See the documentation of `xml.parsers.expat` for the list of error codes and their meanings.

position

A tuple of *line*, *column* numbers, specifying where the error occurred.

Footnotes

[1] (1,2,3,4)

The encoding string included in XML output should conform to the appropriate standards. For example, “UTF-8” is valid, but “UTF8” is not. See https://www.w3.org/TR/2006/REC-xml11-20060816/#NT-EncodingDecl and https://www.iana.org/assignments/character-sets/character-sets.xhtml.

`xml.dom` — The Document Object Model API

Source code: Lib/xml/dom/__init__.py [https://github.com/python/cpython/tree/3.14/Lib/xml/dom/__init__.py]

The Document Object Model, or “DOM,” is a cross-language API from the World Wide Web Consortium (W3C) for accessing and modifying XML documents. A DOM implementation presents an XML document as a tree structure, or allows client code to build such a structure from scratch. It then gives access to the structure through a set of objects which provided well-known interfaces.

The DOM is extremely useful for random-access applications. SAX only allows you a view of one bit of the document at a time. If you are looking at one SAX element, you have no access to another. If you are looking at a text node, you have no access to a containing element. When you write a SAX application, you need to keep track of your program’s position in the document somewhere in your own code. SAX does not do it for you. Also, if you need to look ahead in the XML document, you are just out of luck.

Some applications are simply impossible in an event driven model with no access to a tree. Of course you could build some sort of tree yourself in SAX events, but the DOM allows you to avoid writing that code. The DOM is a standard tree representation for XML data.

The Document Object Model is being defined by the W3C in stages, or “levels” in their terminology. The Python mapping of the API is substantially based on the DOM Level 2 recommendation.

DOM applications typically start by parsing some XML into a DOM. How this is accomplished is not covered at all by DOM Level 1, and Level 2 provides only limited improvements: There is a `DOMImplementation` object class which provides access to `Document` creation methods, but no way to access an XML reader/parser/Document builder in an implementation-independent way. There is also no well-defined way to access these methods without an existing `Document` object. In Python, each DOM implementation will provide a function `getDOMImplementation()`. DOM Level 3 adds a Load/Store specification, which defines an interface to the reader, but this is not yet available in the Python standard library.

Once you have a DOM document object, you can access the parts of your XML document through its properties and methods. These properties are defined in the DOM specification; this portion of the reference manual describes the interpretation of the specification in Python.

The specification provided by the W3C defines the DOM API for Java, ECMAScript, and OMG IDL. The Python mapping defined here is based in large part on the IDL version of the specification, but strict compliance is not required (though implementations are free to support the strict mapping from IDL). See section Conformance for a detailed discussion of mapping requirements.

See also

Document Object Model (DOM) Level 2 Specification
[https://www.w3.org/TR/2000/REC-DOM-Level-2-Core-20001113/]

The W3C recommendation upon which the Python DOM API is based.

Document Object Model (DOM) Level 1 Specification [https://www.w3.org/TR/REC-DOM-Level-1/]
The W3C recommendation for the DOM supported by `xml.dom.minidom`.

Python Language Mapping Specification [https://www.omg.org/spec/PYTH/1.2/PDF]
This specifies the mapping from OMG IDL to Python.

Module Contents

The `xml.dom` contains the following functions:

xml.dom.registerDOMImplementation(*name*, *factory*)

Register the *factory* function with the name *name*. The factory function should return an object which implements the **`DOMImplementation`** interface. The factory function can return the same object every time, or a new one for each call, as appropriate for the specific implementation (e.g. if that implementation supports some customization).

xml.dom.getDOMImplementation(*name=None*, *features=()*)

Return a suitable DOM implementation. The *name* is either well-known, the module name of a DOM implementation, or `None`. If it is not `None`, imports the corresponding module and returns a **`DOMImplementation`** object if the import succeeds. If no name is given, and if the environment variable **`PYTHON_DOM`** is set, this variable is used to find the implementation.

If name is not given, this examines the available implementations to find one with the required feature set. If no implementation can be found, raise an `ImportError`. The features list must be a sequence of `(feature, version)` pairs which are passed to the **`hasFeature()`** method on available **`DOMImplementation`** objects.

Some convenience constants are also provided:

xml.dom.EMPTY_NAMESPACE

The value used to indicate that no namespace is associated with a node in the DOM. This is typically found as the **`namespaceURI`** of a node, or used as the *namespaceURI* parameter to a namespaces-specific method.

xml.dom.XML_NAMESPACE

The namespace URI associated with the reserved prefix `xml`, as defined by Namespaces in XML [https://www.w3.org/TR/REC-xml-names/] (section 4).

xml.dom.XMLNS_NAMESPACE

The namespace URI for namespace declarations, as defined by Document Object Model (DOM) Level 2 Core Specification [https://www.w3.org/TR/DOM-Level-2-Core/core.html] (section 1.1.8).

xml.dom.XHTML_NAMESPACE

The URI of the XHTML namespace as defined by XHTML 1.0: The Extensible HyperText Markup Language [https://www.w3.org/TR/xhtml1/] (section 3.1.1).

In addition, `xml.dom` contains a base **`Node`** class and the DOM exception classes. The **`Node`** class provided by this module does not implement any of the methods or attributes defined by the DOM specification; concrete DOM implementations must provide those. The **`Node`** class provided as part of this module does provide the constants used for the **`nodeType`** attribute on concrete **`Node`** objects; they are located within the class rather than at the module level to conform with the DOM specifications.

Objects in the DOM

The definitive documentation for the DOM is the DOM specification from the W3C.

Note that DOM attributes may also be manipulated as nodes instead of as simple strings. It is fairly rare that you must do this, however, so this usage is not yet documented.

Interface	Section	Purpose
`DOMImplementation`	DOMImplementation Objects	Interface to the underlying implementation.
`Node`	Node Objects	Base interface for most objects in a document.
`NodeList`	NodeList Objects	Interface for a sequence of nodes.
`DocumentType`	DocumentType Objects	Information about the declarations

Interface	Section	Purpose
		needed to process a document.
`Document`	Document Objects	Object which represents an entire document.
`Element`	Element Objects	Element nodes in the document hierarchy.
`Attr`	Attr Objects	Attribute value nodes on element nodes.
`Comment`	Comment Objects	Representation of comments in the source document.
`Text`	Text and CDATASection Objects	Nodes containing textual content from the document.

Interface	Section	Purpose
`ProcessingInstruction`	ProcessingInstruction Objects	Processing instruction representation.

An additional section describes the exceptions defined for working with the DOM in Python.

DOMImplementation Objects

The **`DOMImplementation`** interface provides a way for applications to determine the availability of particular features in the DOM they are using. DOM Level 2 added the ability to create new **`Document`** and **`DocumentType`** objects using the **`DOMImplementation`** as well.

DOMImplementation.hasFeature(*feature*, *version*)

Return `True` if the feature identified by the pair of strings *feature* and *version* is implemented.

DOMImplementation.createDocument(*namespaceUri*, *qualifiedName*, *doctype*)

Return a new **`Document`** object (the root of the DOM), with a child **`Element`** object having the given *namespaceUri* and *qualifiedName*. The *doctype* must be a **`DocumentType`** object created by **`createDocumentType()`**, or `None`. In the Python DOM API, the first two arguments can also be `None` in order to indicate that no **`Element`** child is to be created.

DOMImplementation.createDocumentType(*qualifiedName*, *publicId*, *systemId*)

Return a new **`DocumentType`** object that encapsulates the given *qualifiedName*, *publicId*, and *systemId* strings, representing the information contained in an XML document type declaration.

Node Objects

All of the components of an XML document are subclasses of **`Node`**.

Node.nodeType

An integer representing the node type. Symbolic constants for the types are on the **`Node`** object: **`ELEMENT_NODE`**, **`ATTRIBUTE_NODE`**, **`TEXT_NODE`**, **`CDATA_SECTION_NODE`**, **`ENTITY_NODE`**, **`PROCESSING_INSTRUCTION_NODE`**, **`COMMENT_NODE`**, **`DOCUMENT_NODE`**, **`DOCUMENT_TYPE_NODE`**, **`NOTATION_NODE`**. This is a read-only attribute.

Node.parentNode

The parent of the current node, or `None` for the document node. The value is always a **`Node`** object or `None`. For **`Element`** nodes, this will be the parent element, except for the root element, in which case it will be the **`Document`** object. For **`Attr`** nodes, this is always `None`. This is a read-only attribute.

Node.attributes

A **`NamedNodeMap`** of attribute objects. Only elements have actual values for this; others provide `None` for this attribute. This is a read-only attribute.

Node.previousSibling

The node that immediately precedes this one with the same parent. For instance the element with an end-tag that comes just before the *self* element's start-tag. Of course, XML documents are made up of more than just elements so the previous sibling could be text, a comment, or something else. If this node is the first child of the parent, this attribute will be `None`. This is a read-only attribute.

Node.nextSibling

The node that immediately follows this one with the same parent. See also **`previousSibling`**. If this is the last child of the parent, this attribute will be `None`. This is a read-only attribute.

Node.childNodes

A list of nodes contained within this node. This is a read-only attribute.

Node.firstChild

The first child of the node, if there are any, or `None`. This is a read-only attribute.

Node.lastChild

The last child of the node, if there are any, or `None`. This is a read-only attribute.

Node.localName

The part of the **`tagName`** following the colon if there is one, else the entire **`tagName`**. The value is a string.

Node.prefix

The part of the **`tagName`** preceding the colon if there is one, else the empty string. The value is a string, or `None`.

Node.namespaceURI

The namespace associated with the element name. This will be a string or `None`. This is a read-only attribute.

Node.nodeName

This has a different meaning for each node type; see the DOM specification for details. You can always get the information you would get here from another property such as the **`tagName`** property for elements or the **`name`** property for attributes. For all node types, the value of this attribute will be either a string or `None`. This is a read-only attribute.

Node.nodeValue

This has a different meaning for each node type; see the DOM specification for details. The situation is similar to that with **`nodeName`**. The value is a string or `None`.

Node.hasAttributes()

Return `True` if the node has any attributes.

Node.hasChildNodes()

Return `True` if the node has any child nodes.

Node.isSameNode(*other*)

Return `True` if *other* refers to the same node as this node. This is especially useful for DOM implementations which use any sort of proxy architecture (because more than one object can refer to the same node).

Note

This is based on a proposed DOM Level 3 API which is still in the “working draft” stage, but this particular interface appears uncontroversial. Changes from the W3C will not necessarily affect this method in the Python DOM interface (though any new W3C API for this would also be supported).

Node.appendChild(*newChild*)

Add a new child node to this node at the end of the list of children, returning *newChild*. If the node was already in the tree, it is removed first.

Node.insertBefore(*newChild*, *refChild*)

Insert a new child node before an existing child. It must be the case that *refChild* is a child of this node; if not, `ValueError` is raised. *newChild* is returned. If *refChild* is `None`, it inserts *newChild* at the end of the children's list.

Node.removeChild(*oldChild*)

Remove a child node. *oldChild* must be a child of this node; if not, `ValueError` is raised. *oldChild* is returned on success. If *oldChild* will not be used further, its **`unlink()`** method should be called.

Node.replaceChild(*newChild*, *oldChild*)

Replace an existing node with a new node. It must be the case that *oldChild* is a child of this node; if not, `ValueError` is raised.

Node.normalize()

Join adjacent text nodes so that all stretches of text are stored as single **`Text`** instances. This simplifies processing text from a DOM tree for many applications.

Node.cloneNode(*deep*)

Clone this node. Setting *deep* means to clone all child nodes as well. This returns the clone.

NodeList Objects

A **`NodeList`** represents a sequence of nodes. These objects are used in two ways in the DOM Core recommendation: an **`Element`** object provides one as its list of child nodes, and the `getElementsByTagName()` and `getElementsByTagNameNS()` methods of `Node` return objects with this interface to represent query results.

The DOM Level 2 recommendation defines one method and one attribute for these objects:

NodeList.item(*i*)

Return the *i*’th item from the sequence, if there is one, or `None`. The index *i* is not allowed to be less than zero or greater than or equal to the length of the sequence.

NodeList.length

The number of nodes in the sequence.

In addition, the Python DOM interface requires that some additional support is provided to allow `NodeList` objects to be used as Python sequences. All `NodeList` implementations must include support for `__len__()` and `__getitem__()`; this allows iteration over the `NodeList` in `for` statements and proper support for the `len()` built-in function.

If a DOM implementation supports modification of the document, the `NodeList` implementation must also support the `__setitem__()` and `__delitem__()` methods.

DocumentType Objects

Information about the notations and entities declared by a document (including the external subset if the parser uses it and can provide the information) is available from a `DocumentType` object. The `DocumentType` for a document is available from the `Document` object’s `doctype` attribute; if there is no `DOCTYPE` declaration for the document, the document’s `doctype` attribute will be set to `None` instead of an instance of this interface.

`DocumentType` is a specialization of **`Node`**, and adds the following attributes:

DocumentType.publicId

The public identifier for the external subset of the document type definition. This will be a string or `None`.

DocumentType.systemId

The system identifier for the external subset of the document type definition. This will be a URI as a string, or `None`.

DocumentType.internalSubset

A string giving the complete internal subset from the document. This does not include the brackets which enclose the subset. If the document has no internal subset, this should be `None`.

DocumentType.name

The name of the root element as given in the `DOCTYPE` declaration, if present.

DocumentType.entities

This is a **`NamedNodeMap`** giving the definitions of external entities. For entity names defined more than once, only the first definition is provided (others are ignored as required by the XML recommendation). This may be `None` if the information is not provided by the parser, or if no entities are defined.

DocumentType.notations

This is a **`NamedNodeMap`** giving the definitions of notations. For notation names defined more than once, only the first definition is provided (others are ignored as required by the XML recommendation). This may be `None` if the information is not provided by the parser, or if no notations are defined.

Document Objects

A `Document` represents an entire XML document, including its constituent elements, attributes, processing instructions, comments etc. Remember that it inherits properties from `Node`.

Document.documentElement
: The one and only root element of the document.

Document.createElement(*tagName*)
: Create and return a new element node. The element is not inserted into the document when it is created. You need to explicitly insert it with one of the other methods such as `insertBefore()` or `appendChild()`.

Document.createElementNS(*namespaceURI*, *tagName*)
: Create and return a new element with a namespace. The *tagName* may have a prefix. The element is not inserted into the document when it is created. You need to explicitly insert it with one of the other methods such as `insertBefore()` or `appendChild()`.

Document.createTextNode(*data*)
: Create and return a text node containing the data passed as a parameter. As with the other creation methods, this one does not insert the node into the tree.

Document.createComment(*data*)
: Create and return a comment node containing the data passed as a parameter. As with the other creation methods, this one does not insert the node into the tree.

Document.createProcessingInstruction(*target*, *data*)

Create and return a processing instruction node containing the *target* and *data* passed as parameters. As with the other creation methods, this one does not insert the node into the tree.

Document.createAttribute(*name*)

Create and return an attribute node. This method does not associate the attribute node with any particular element. You must use **`setAttributeNode()`** on the appropriate **`Element`** object to use the newly created attribute instance.

Document.createAttributeNS(*namespaceURI*, *qualifiedName*)

Create and return an attribute node with a namespace. The *tagName* may have a prefix. This method does not associate the attribute node with any particular element. You must use **`setAttributeNode()`** on the appropriate **`Element`** object to use the newly created attribute instance.

Document.getElementsByTagName(*tagName*)

Search for all descendants (direct children, children’s children, etc.) with a particular element type name.

Document.getElementsByTagNameNS(*namespaceURI*, *localName*)

Search for all descendants (direct children, children’s children, etc.) with a particular namespace URI and localname. The localname is the part of the namespace after the prefix.

Element Objects

`Element` is a subclass of **`Node`**, so inherits all the attributes of that class.

Element.tagName

The element type name. In a namespace-using document it may have colons in it. The value is a string.

Element.getElementsByTagName(*tagName*)

Same as equivalent method in the **`Document`** class.

Element.getElementsByTagNameNS(*namespaceURI*, *localName*)

Same as equivalent method in the **`Document`** class.

Element.hasAttribute(*name*)

Return `True` if the element has an attribute named by *name*.

Element.hasAttributeNS(*namespaceURI*, *localName*)

Return `True` if the element has an attribute named by *namespaceURI* and *localName*.

Element.getAttribute(*name*)

Return the value of the attribute named by *name* as a string. If no such attribute exists, an empty string is returned, as if the attribute had no value.

Element.getAttributeNode(*attrname*)

Return the **`Attr`** node for the attribute named by *attrname*.

Element.getAttributeNS(*namespaceURI*, *localName*)

Return the value of the attribute named by *namespaceURI* and *localName* as a string. If no such attribute exists, an empty string is returned, as if the attribute had no value.

Element.getAttributeNodeNS(*namespaceURI*, *localName*)

Return an attribute value as a node, given a *namespaceURI* and *localName*.

Element.removeAttribute(*name*)

Remove an attribute by name. If there is no matching attribute, a `NotFoundErr` is raised.

Element.removeAttributeNode(*oldAttr*)

Remove and return *oldAttr* from the attribute list, if present. If *oldAttr* is not present, `NotFoundErr` is raised.

Element.removeAttributeNS(*namespaceURI*, *localName*)

Remove an attribute by name. Note that it uses a localName, not a qname. No exception is raised if there is no matching attribute.

Element.setAttribute(*name*, *value*)

Set an attribute value from a string.

Element.setAttributeNode(*newAttr*)

Add a new attribute node to the element, replacing an existing attribute if necessary if the **`name`** attribute matches. If a replacement occurs, the old attribute node will be returned. If *newAttr* is already in use, `InuseAttributeErr` will be raised.

Element.setAttributeNodeNS(*newAttr*)

Add a new attribute node to the element, replacing an existing attribute if necessary if the **`namespaceURI`** and **`localName`** attributes match. If a replacement occurs, the old attribute node will be returned. If *newAttr* is already in use, `InuseAttributeErr` will be raised.

Element.setAttributeNS(*namespaceURI*, *qname*, *value*)

Set an attribute value from a string, given a *namespaceURI* and a *qname*. Note that a qname is the whole attribute name. This is different than above.

Attr Objects

`Attr` inherits from **`Node`**, so inherits all its attributes.

Attr.name

The attribute name. In a namespace-using document it may include a colon.

Attr.localName

The part of the name following the colon if there is one, else the entire name. This is a read-only attribute.

Attr.prefix

The part of the name preceding the colon if there is one, else the empty string.

Attr.value

The text value of the attribute. This is a synonym for the **`nodeValue`** attribute.

NamedNodeMap Objects

`NamedNodeMap` does *not* inherit from **`Node`**.

NamedNodeMap.length

The length of the attribute list.

NamedNodeMap.item(*index*)

Return an attribute with a particular index. The order you get the attributes in is arbitrary but will be consistent for the life of a DOM. Each item is an attribute node. Get its value with the **`value`** attribute.

There are also experimental methods that give this class more mapping behavior. You can use them or you can use the standardized **`getAttribute*()`** family of methods on the **`Element`** objects.

Comment Objects

`Comment` represents a comment in the XML document. It is a subclass of **`Node`**, but cannot have child nodes.

Comment.data

The content of the comment as a string. The attribute contains all characters between the leading `<!--` and trailing `-->`, but does not include them.

Text and CDATASection Objects

The **`Text`** interface represents text in the XML document. If the parser and DOM implementation support the DOM’s XML extension, portions of the text enclosed in CDATA marked sections are stored in **`CDATASection`** objects. These two interfaces are identical, but provide different values for the **`nodeType`** attribute.

These interfaces extend the **`Node`** interface. They cannot have child nodes.

Text.data

The content of the text node as a string.

Note

The use of a `CDATASection` node does not indicate that the node represents a complete CDATA marked section, only that the content of the node was part of a CDATA section. A single CDATA section may be represented by more than one node in the document tree. There is no way to determine whether two adjacent `CDATASection` nodes represent different CDATA marked sections.

ProcessingInstruction Objects

Represents a processing instruction in the XML document; this inherits from the `Node` interface and cannot have child nodes.

ProcessingInstruction.target

The content of the processing instruction up to the first whitespace character. This is a read-only attribute.

ProcessingInstruction.data

The content of the processing instruction following the first whitespace character.

Exceptions

The DOM Level 2 recommendation defines a single exception, `DOMException`, and a number of constants that allow applications to determine what sort of error occurred. `DOMException` instances carry a `code` attribute that provides the appropriate value for the specific exception.

The Python DOM interface provides the constants, but also expands the set of exceptions so that a specific exception exists for each of the exception codes defined by the DOM. The implementations must raise the appropriate specific exception, each of which carries the appropriate value for the `code` attribute.

exception xml.dom.DOMException

Base exception class used for all specific DOM exceptions. This exception class cannot be directly instantiated.

exception xml.dom.DomstringSizeErr

Raised when a specified range of text does not fit into a string. This is not known to be used in the Python DOM implementations, but may be received from DOM implementations not written in Python.

exception xml.dom.HierarchyRequestErr

Raised when an attempt is made to insert a node where the node type is not allowed.

exception xml.dom.IndexSizeErr

Raised when an index or size parameter to a method is negative or exceeds the allowed values.

exception xml.dom.InuseAttributeErr

Raised when an attempt is made to insert an **`Attr`** node that is already present elsewhere in the document.

exception xml.dom.InvalidAccessErr

Raised if a parameter or an operation is not supported on the underlying object.

exception xml.dom.InvalidCharacterErr

This exception is raised when a string parameter contains a character that is not permitted in the context it’s being used in by the XML 1.0 recommendation. For example, attempting to create an **`Element`** node with a space in the element type name will cause this error to be raised.

exception xml.dom.InvalidModificationErr

Raised when an attempt is made to modify the type of a node.

exception xml.dom.InvalidStateErr

Raised when an attempt is made to use an object that is not defined or is no longer usable.

exception xml.dom.NamespaceErr

If an attempt is made to change any object in a way that is not permitted with regard to the Namespaces in XML [https://www.w3.org/TR/REC-xml-names/] recommendation, this exception is raised.

exception xml.dom.NotFoundErr

Exception when a node does not exist in the referenced context. For example, **`NamedNodeMap.removeNamedItem()`** will raise this if the node passed in does not exist in the map.

exception xml.dom.NotSupportedErr

Raised when the implementation does not support the requested type of object or operation.

exception xml.dom.NoDataAllowedErr

This is raised if data is specified for a node which does not support data.

exception xml.dom.NoModificationAllowedErr

Raised on attempts to modify an object where modifications are not allowed (such as for read-only nodes).

exception xml.dom.SyntaxErr

Raised when an invalid or illegal string is specified.

exception xml.dom.WrongDocumentErr

Raised when a node is inserted in a different document than it currently belongs to, and the implementation does not support migrating the node from one document to the other.

The exception codes defined in the DOM recommendation map to the exceptions described above according to this table:

Constant	Exception
`DOMSTRING_SIZE_ERR`	`DomstringSizeErr`
`HIERARCHY_REQUEST_ERR`	`HierarchyRequestErr`
`INDEX_SIZE_ERR`	`IndexSizeErr`
`INUSE_ATTRIBUTE_ERR`	`InuseAttributeErr`
`INVALID_ACCESS_ERR`	`InvalidAccessErr`
`INVALID_CHARACTER_ERR`	`InvalidCharacterErr`
`INVALID_MODIFICATION_ERR`	`InvalidModificationErr`
`INVALID_STATE_ERR`	`InvalidStateErr`
`NAMESPACE_ERR`	`NamespaceErr`

Constant	Exception
`NOT_FOUND_ERR`	`NotFoundErr`
`NOT_SUPPORTED_ERR`	`NotSupportedErr`
`NO_DATA_ALLOWED_ERR`	`NoDataAllowedErr`
`NO_MODIFICATION_ALLOWED_ERR`	`NoModificationAllowedErr`
`SYNTAX_ERR`	`SyntaxErr`
`WRONG_DOCUMENT_ERR`	`WrongDocumentErr`

Conformance

This section describes the conformance requirements and relationships between the Python DOM API, the W3C DOM recommendations, and the OMG IDL mapping for Python.

Type Mapping

The IDL types used in the DOM specification are mapped to Python types according to the following table.

IDL Type	Python Type
`boolean`	`bool` or `int`
`int`	`int`
`long int`	`int`
`unsigned int`	`int`
`DOMString`	`str` or `bytes`
`null`	`None`

Accessor Methods

The mapping from OMG IDL to Python defines accessor functions for IDL `attribute` declarations in much the way the Java mapping does. Mapping the IDL declarations

```
readonly attribute string someValue;
         attribute string anotherValue;
```

yields three accessor functions: a “get” method for **`someValue`** (**`_get_someValue()`**), and “get” and “set” methods for **`anotherValue`** (**`_get_anotherValue()`** and **`_set_anotherValue()`**). The mapping, in particular, does not require that the IDL attributes are accessible as normal Python attributes: `object.someValue` is *not* required to work, and may raise an `AttributeError`.

The Python DOM API, however, *does* require that normal attribute access work. This means that the typical surrogates generated by Python IDL compilers are not likely to work, and wrapper objects may be needed on the client if the DOM objects are accessed via CORBA. While this does require some additional consideration for CORBA DOM clients, the implementers with experience using DOM over CORBA from Python do not consider this a problem. Attributes that are declared `readonly` may not restrict write access in all DOM implementations.

In the Python DOM API, accessor functions are not required. If provided, they should take the form defined by the Python IDL mapping, but these methods are considered unnecessary since the attributes are accessible directly from Python. “Set” accessors should never be provided for `readonly` attributes.

The IDL definitions do not fully embody the requirements of the W3C DOM API, such as the notion of certain objects, such as the return value of **`getElementsByTagName()`**, being “live”. The Python DOM API does not require implementations to enforce such requirements.

`xml.dom.minidom` — Minimal DOM implementation

Source code: Lib/xml/dom/minidom.py [https://github.com/python/cpython/tree/3.14/Lib/xml/dom/minidom.py]

`xml.dom.minidom` is a minimal implementation of the Document Object Model interface, with an API similar to that in other languages. It is intended to be simpler than the full DOM and also significantly smaller. Users who are not already proficient with the DOM should consider using the **`xml.etree.ElementTree`** module for their XML processing instead.

Note

If you need to parse untrusted or unauthenticated data, see XML security.

DOM applications typically start by parsing some XML into a DOM. With **`xml.dom.minidom`**, this is done through the parse functions:

```
from xml.dom.minidom import parse, parseString

dom1 = parse('c:\\temp\\mydata.xml')  # parse an XML file by
name

datasource = open('c:\\temp\\mydata.xml')
dom2 = parse(datasource)  # parse an open file
```

```
dom3 = parseString('<myxml>Some data<empty/> some more
data</myxml>')
```

The `parse()` function can take either a filename or an open file object.

xml.dom.minidom.parse(*filename_or_file*, *parser=None*, *bufsize=None*)

> Return a **`Document`** from the given input. *filename_or_file* may be either a file name, or a file-like object. *parser*, if given, must be a SAX2 parser object. This function will change the document handler of the parser and activate namespace support; other parser configuration (like setting an entity resolver) must have been done in advance.

If you have XML in a string, you can use the `parseString()` function instead:

xml.dom.minidom.parseString(*string*, *parser=None*)

> Return a **`Document`** that represents the *string*. This method creates an `io.StringIO` object for the string and passes that on to `parse()`.

Both functions return a **`Document`** object representing the content of the document.

What the `parse()` and `parseString()` functions do is connect an XML parser with a “DOM builder” that can accept parse events from any SAX parser and convert them into a DOM tree. The name of the functions are perhaps misleading, but are easy to grasp when learning the interfaces. The parsing of the document will be completed before these functions return; it’s simply that these functions do not provide a parser implementation themselves.

You can also create a **`Document`** by calling a method on a “DOM Implementation” object. You can get this object either by calling the **`getDOMImplementation()`** function in the [`xml.dom`](#) package or the [`xml.dom.minidom`](#) module. Once you have a **`Document`**, you can add child nodes to it to populate the DOM:

```
from xml.dom.minidom import getDOMImplementation

impl = getDOMImplementation()

newdoc = impl.createDocument(None, "some_tag", None)
top_element = newdoc.documentElement
text = newdoc.createTextNode('Some textual content.')
top_element.appendChild(text)
```

Once you have a DOM document object, you can access the parts of your XML document through its properties and methods. These properties are defined in the DOM specification. The main property of the document object is the **`documentElement`** property. It gives you the main element in the XML document: the one that holds all others. Here is an example program:

```
dom3 = parseString("<myxml>Some data</myxml>")
assert dom3.documentElement.tagName == "myxml"
```

When you are finished with a DOM tree, you may optionally call the **`unlink()`** method to encourage early cleanup of the now-unneeded objects. **`unlink()`** is an [`xml.dom.minidom`](#)-specific extension to the DOM API that renders the node and its descendants essentially useless. Otherwise, Python’s garbage collector will eventually take care of the objects in the tree.

See also

Document Object Model (DOM) Level 1 Specification

[https://www.w3.org/TR/REC-DOM-Level-1/]

The W3C recommendation for the DOM supported by `xml.dom.minidom`.

DOM Objects

The definition of the DOM API for Python is given as part of the `xml.dom` module documentation. This section lists the differences between the API and `xml.dom.minidom`.

Node.unlink()

Break internal references within the DOM so that it will be garbage collected on versions of Python without cyclic GC. Even when cyclic GC is available, using this can make large amounts of memory available sooner, so calling this on DOM objects as soon as they are no longer needed is good practice. This only needs to be called on the **`Document`** object, but may be called on child nodes to discard children of that node.

You can avoid calling this method explicitly by using the `with` statement. The following code will automatically unlink *dom* when the **`with`** block is exited:

```
with xml.dom.minidom.parse(datasource) as dom:
    ... # Work with dom.
```

Node.writexml(*writer*, *indent*='', *addindent*='', *newl*='', *encoding=None*, *standalone=None*)

Write XML to the writer object. The writer receives texts but not bytes as input, it should have a **`write()`** method which matches that of the file object interface. The *indent* parameter is the indentation of the current node. The *addindent* parameter is the incremental indentation to use for subnodes of the current one. The *newl* parameter specifies the string to use to terminate newlines.

For the **`Document`** node, an additional keyword argument *encoding* can be used to specify the encoding field of the XML header.

Similarly, explicitly stating the *standalone* argument causes the standalone document declarations to be added to the prologue of the XML document. If the value is set to `True`, `standalone="yes"` is added, otherwise it is set to `"no"`. Not stating the argument will omit the declaration from the document.

Changed in version 3.8: The **`writexml()`** method now preserves the attribute order specified by the user.

Changed in version 3.9: The *standalone* parameter was added.

Node.toxml(*encoding=None*, *standalone=None*)

Return a string or byte string containing the XML represented by the DOM node.

With an explicit *encoding* [[1]] argument, the result is a byte string in the specified encoding. With no *encoding* argument, the result is a Unicode string, and the XML declaration in the resulting string does not specify an encoding. Encoding this string in an encoding other than UTF-8 is likely incorrect, since UTF-8 is the default encoding of XML.

The *standalone* argument behaves exactly as in **`writexml()`**.

Changed in version 3.8: The **`toxml()`** method now preserves the attribute order specified by the user.

Changed in version 3.9: The *standalone* parameter was added.

Node.toprettyxml(*indent='\t'*, *newl='\n'*, *encoding=None*, *standalone=None*)

Return a pretty-printed version of the document. *indent* specifies the indentation string and defaults to a tabulator; *newl* specifies the string emitted at the end of each line and defaults to `\n`.

The *encoding* argument behaves like the corresponding argument of **`toxml()`**.

The *standalone* argument behaves exactly as in **`writexml()`**.

Changed in version 3.8: The **`toprettyxml()`** method now preserves the attribute order specified by the user.

Changed in version 3.9: The *standalone* parameter was added.

DOM Example

This example program is a fairly realistic example of a simple program. In this particular case, we do not take much advantage of the flexibility of the DOM.

```
import xml.dom.minidom

document = """\
<slideshow>
<title>Demo slideshow</title>
<slide><title>Slide title</title>
<point>This is a demo</point>
<point>Of a program for processing slides</point>
</slide>
```

```
<slide><title>Another demo slide</title>
<point>It is important</point>
<point>To have more than</point>
<point>one slide</point>
</slide>
</slideshow>
"""

dom = xml.dom.minidom.parseString(document)

def getText(nodelist):
    rc = []
    for node in nodelist:
        if node.nodeType == node.TEXT_NODE:
            rc.append(node.data)
    return ''.join(rc)

def handleSlideshow(slideshow):
    print("<html>")

handleSlideshowTitle(slideshow.getElementsByTagName("title")
[0])
    slides = slideshow.getElementsByTagName("slide")
    handleToc(slides)
    handleSlides(slides)
    print("</html>")

def handleSlides(slides):
    for slide in slides:
        handleSlide(slide)

def handleSlide(slide):
    handleSlideTitle(slide.getElementsByTagName("title")[0])
    handlePoints(slide.getElementsByTagName("point"))

def handleSlideshowTitle(title):
    print(f"<title>{getText(title.childNodes)}</title>")

def handleSlideTitle(title):
    print(f"<h2>{getText(title.childNodes)}</h2>")

def handlePoints(points):
    print("<ul>")
    for point in points:
        handlePoint(point)
    print("</ul>")

def handlePoint(point):
    print(f"<li>{getText(point.childNodes)}</li>")

def handleToc(slides):
    for slide in slides:
        title = slide.getElementsByTagName("title")[0]
        print(f"<p>{getText(title.childNodes)}</p>")

handleSlideshow(dom)
```

minidom and the DOM standard

The `xml.dom.minidom` module is essentially a DOM 1.0-compatible DOM with some DOM 2 features (primarily namespace features).

Usage of the DOM interface in Python is straight-forward. The following mapping rules apply:

- Interfaces are accessed through instance objects. Applications should not instantiate the classes themselves; they should use the creator functions available on the **`Document`** object. Derived interfaces support all operations (and attributes) from the base interfaces, plus any new operations.
- Operations are used as methods. Since the DOM uses only `in` parameters, the arguments are passed in normal order (from left to right). There are no optional arguments. `void` operations return `None`.
- IDL attributes map to instance attributes. For compatibility with the OMG IDL language mapping for Python, an attribute `foo` can also be accessed through accessor methods **`_get_foo()`** and **`_set_foo()`**. `readonly`

attributes must not be changed; this is not enforced at runtime.
- The types `short int`, `unsigned int`, `unsigned long long`, and `boolean` all map to Python integer objects.
- The type `DOMString` maps to Python strings. **`xml.dom.minidom`** supports either bytes or strings, but will normally produce strings. Values of type `DOMString` may also be `None` where allowed to have the IDL `null` value by the DOM specification from the W3C.
- `const` declarations map to variables in their respective scope (e.g. `xml.dom.minidom.Node.PROCESSING_INSTRUCTION_NODE`); they must not be changed.
- `DOMException` is currently not supported in **`xml.dom.minidom`**. Instead, **`xml.dom.minidom`** uses standard Python exceptions such as **`TypeError`** and **`AttributeError`**.
- **`NodeList`** objects are implemented using Python’s built-in list type. These objects provide the interface defined in the DOM specification, but with earlier versions of Python they do not support the official API. They are, however, much more “Pythonic” than the interface defined in the W3C recommendations.

The following interfaces have no implementation in **`xml.dom.minidom`**:

- **`DOMTimeStamp`**
- **`EntityReference`**

Most of these reflect information in the XML document that is not of general utility to most DOM users.

Footnotes

[1]

The encoding name included in the XML output should conform to the appropriate standards. For example, “UTF-8” is valid, but “UTF8” is not valid in an XML document’s declaration, even though Python accepts it as an encoding name. See https://www.w3.org/TR/2006/REC-xml11-20060816/#NT-EncodingDecl and https://www.iana.org/assignments/character-sets/character-sets.xhtml.

`xml.dom.pulldom` — Support for building partial DOM trees

Source code: Lib/xml/dom/pulldom.py
[https://github.com/python/cpython/tree/3.14/Lib/xml/dom/pulldom.py]

The **`xml.dom.pulldom`** module provides a “pull parser” which can also be asked to produce DOM-accessible fragments of the document where necessary. The basic concept involves pulling “events” from a stream of incoming XML and processing them. In contrast to SAX which also employs an event-driven processing model together with callbacks, the user of a pull parser is responsible for explicitly pulling events from the stream, looping over those events until either processing is finished or an error condition occurs.

Note

If you need to parse untrusted or unauthenticated data, see XML security.

Changed in version 3.7.1: The SAX parser no longer processes general external entities by default to increase security by default. To enable processing of external entities, pass a custom parser instance in:

```
from xml.dom.pulldom import parse
from xml.sax import make_parser
from xml.sax.handler import feature_external_ges
```

```
parser = make_parser()
parser.setFeature(feature_external_ges, True)
parse(filename, parser=parser)
```

Example:

```
from xml.dom import pulldom

doc = pulldom.parse('sales_items.xml')
for event, node in doc:
    if event == pulldom.START_ELEMENT and node.tagName ==
'item':
        if int(node.getAttribute('price')) > 50:
            doc.expandNode(node)
            print(node.toxml())
```

`event` is a constant and can be one of:

- **`START_ELEMENT`**
- **`END_ELEMENT`**
- **`COMMENT`**
- **`START_DOCUMENT`**
- **`END_DOCUMENT`**
- **`CHARACTERS`**
- **`PROCESSING_INSTRUCTION`**
- **`IGNORABLE_WHITESPACE`**

`node` is an object of type **`xml.dom.minidom.Document`**, **`xml.dom.minidom.Element`** or **`xml.dom.minidom.Text`**.

Since the document is treated as a “flat” stream of events, the document “tree” is implicitly traversed and the desired elements are found regardless of their depth in the tree. In other words, one does not need to consider hierarchical issues such as recursive searching of the document nodes, although if the context of elements were important, one would either need to maintain some context-related state (i.e. remembering where one is in the document at any given point) or to make use of the `DOMEventStream.expandNode()` method and switch to DOM-related processing.

class xml.dom.pulldom.PullDOM(*documentFactory=None*)

Subclass of `xml.sax.handler.ContentHandler`.

class xml.dom.pulldom.SAX2DOM(*documentFactory=None*)

Subclass of `xml.sax.handler.ContentHandler`.

xml.dom.pulldom.parse(*stream_or_string*, *parser=None*, *bufsize=None*)

Return a `DOMEventStream` from the given input. *stream_or_string* may be either a file name, or a file-like object. *parser*, if given, must be an `XMLReader` object. This function will change the document handler of the parser and activate namespace support; other parser configuration (like setting an entity resolver) must have been done in advance.

If you have XML in a string, you can use the `parseString()` function instead:

xml.dom.pulldom.parseString(*string*, *parser=None*)

Return a `DOMEventStream` that represents the (Unicode) *string*.

xml.dom.pulldom.default_bufsize

Default value for the *bufsize* parameter to `parse()`.

The value of this variable can be changed before calling `parse()` and the new value will take effect.

DOMEventStream Objects

class xml.dom.pulldom.DOMEventStream(*stream*, *parser*, *bufsize*)

Changed in version 3.11: Support for `__getitem__()` method has been removed.

getEvent()

Return a tuple containing *event* and the current *node* as **`xml.dom.minidom.Document`** if event equals **`START_DOCUMENT`**, **`xml.dom.minidom.Element`** if event equals **`START_ELEMENT`** or **`END_ELEMENT`** or **`xml.dom.minidom.Text`** if event equals **`CHARACTERS`**. The current node does not contain information about its children, unless `expandNode()` is called.

expandNode(*node*)

Expands all children of *node* into *node*. Example:

```
from xml.dom import pulldom

xml = '<html><title>Foo</title> <p>Some text <div>and
more</div></p> </html>'
doc = pulldom.parseString(xml)
for event, node in doc:
    if event == pulldom.START_ELEMENT and node.tagName ==
'p':
        # Following statement only prints '<p/>'
        print(node.toxml())
        doc.expandNode(node)
        # Following statement prints node with all its
children '<p>Some text <div>and more</div></p>'
        print(node.toxml())
```

reset()

`xml.sax` — Support for SAX2 parsers

Source code: Lib/xml/sax/__init__.py
[https://github.com/python/cpython/tree/3.14/Lib/xml/sax/__init__.py]

The **`xml.sax`** package provides a number of modules which implement the Simple API for XML (SAX) interface for Python. The package itself provides the SAX exceptions and the convenience functions which will be most used by users of the SAX API.

Note

If you need to parse untrusted or unauthenticated data, see XML security.

Changed in version 3.7.1: The SAX parser no longer processes general external entities by default to increase security. Before, the parser created network connections to fetch remote files or loaded local files from the file system for DTD and entities. The feature can be enabled again with method **`setFeature()`** on the parser object and argument **`feature_external_ges`**.

The convenience functions are:

xml.sax.make_parser(*parser_list=[]*)

Create and return a SAX **`XMLReader`** object. The first parser found will be used. If *parser_list* is provided, it must be an iterable of strings which name modules that have a function named `create_parser()`. Modules listed in *parser_list* will be used before modules in the default list of parsers.

Changed in version 3.8: The *parser_list* argument can be any iterable, not just a list.

xml.sax.parse(*filename_or_stream*, *handler*, *error_handler=handler.ErrorHandler()*)

Create a SAX parser and use it to parse a document. The document, passed in as *filename_or_stream*, can be a filename or a file object. The *handler* parameter needs to be a SAX `ContentHandler` instance. If *error_handler* is given, it must be a SAX `ErrorHandler` instance; if omitted, `SAXParseException` will be raised on all errors. There is no return value; all work must be done by the *handler* passed in.

xml.sax.parseString(*string*, *handler*, *error_handler=handler.ErrorHandler()*)

Similar to `parse()`, but parses from a buffer *string* received as a parameter. *string* must be a `str` instance or a bytes-like object.

Changed in version 3.5: Added support of `str` instances.

A typical SAX application uses three kinds of objects: readers, handlers and input sources. “Reader” in this context is another term for parser, i.e. some piece of code that reads the bytes or characters from the input source, and produces a sequence of events. The events then get distributed to the handler objects, i.e. the reader invokes a method on the handler. A SAX application must therefore obtain a reader object, create or open the input sources, create the handlers, and connect these objects all together. As the final step of preparation, the reader is called to parse the input. During parsing, methods on the handler objects are called based on structural and syntactic events from the input data.

For these objects, only the interfaces are relevant; they are normally not instantiated by the application itself. Since Python does not have an explicit notion of interface, they are formally introduced as classes, but applications may use implementations which do not inherit from the provided classes. The `InputSource`, `Locator`, **`Attributes`**, **`AttributesNS`**, and `XMLReader` interfaces are defined in the module `xml.sax.xmlreader`. The handler interfaces are defined in `xml.sax.handler`. For convenience, `InputSource` (which is often instantiated directly) and the handler classes are also available from `xml.sax`. These interfaces are described below.

In addition to these classes, `xml.sax` provides the following exception classes.

exception xml.sax.SAXException(*msg*, *exception=None*)

> Encapsulate an XML error or warning. This class can contain basic error or warning information from either the XML parser or the application: it can be subclassed to provide additional functionality or to add localization. Note that although the handlers defined in the `ErrorHandler` interface receive instances of this exception, it is not required to actually raise the exception — it is also useful as a container for information.
>
> When instantiated, *msg* should be a human-readable description of the error. The optional *exception* parameter, if given, should be `None` or an exception that was caught by the parsing code and is being passed along as information.

This is the base class for the other SAX exception classes.

exception xml.sax.SAXParseException(*msg*, *exception*, *locator*)

Subclass of `SAXException` raised on parse errors. Instances of this class are passed to the methods of the SAX `ErrorHandler` interface to provide information about the parse error. This class supports the SAX `Locator` interface as well as the `SAXException` interface.

exception xml.sax.SAXNotRecognizedException(*msg*, *exception=None*)

Subclass of `SAXException` raised when a SAX `XMLReader` is confronted with an unrecognized feature or property. SAX applications and extensions may use this class for similar purposes.

exception xml.sax.SAXNotSupportedException(*msg*, *exception=None*)

Subclass of `SAXException` raised when a SAX `XMLReader` is asked to enable a feature that is not supported, or to set a property to a value that the implementation does not support. SAX applications and extensions may use this class for similar purposes.

See also

SAX: The Simple API for XML **[http://www.saxproject.org/]**

This site is the focal point for the definition of the SAX API. It provides a Java implementation and online documentation. Links to implementations and historical information are also available.

Module `xml.sax.handler`

Definitions of the interfaces for application-provided objects.

Module `xml.sax.saxutils`
Convenience functions for use in SAX applications.

Module `xml.sax.xmlreader`
Definitions of the interfaces for parser-provided objects.

SAXException Objects

The `SAXException` exception class supports the following methods:

SAXException.getMessage()
Return a human-readable message describing the error condition.

SAXException.getException()
Return an encapsulated exception object, or `None`.

`xml.sax.handler` — Base classes for SAX handlers

Source code: Lib/xml/sax/handler.py [https://github.com/python/cpython/tree/3.14/Lib/xml/sax/handler.py]

The SAX API defines five kinds of handlers: content handlers, DTD handlers, error handlers, entity resolvers and lexical handlers. Applications normally only need to implement those interfaces whose events they are interested in; they can implement the interfaces in a single object or in multiple objects. Handler implementations should inherit from the base classes provided in the module `xml.sax.handler`, so that all methods get default implementations.

class xml.sax.handler.ContentHandler

This is the main callback interface in SAX, and the one most important to applications. The order of events in this interface mirrors the order of the information in the document.

class xml.sax.handler.DTDHandler

Handle DTD events.

This interface specifies only those DTD events required for basic parsing (unparsed entities and attributes).

class xml.sax.handler.EntityResolver

Basic interface for resolving entities. If you create an object implementing this interface, then register the object with your Parser, the parser will call the method in your object to resolve all external entities.

class xml.sax.handler.ErrorHandler

Interface used by the parser to present error and warning messages to the application. The methods of this object control whether errors are immediately converted to exceptions or are handled in some other way.

class xml.sax.handler.LexicalHandler

Interface used by the parser to represent low frequency events which may not be of interest to many applications.

In addition to these classes, `xml.sax.handler` provides symbolic constants for the feature and property names.

xml.sax.handler.feature_namespaces

value: `"http://xml.org/sax/features/namespaces"`
true: Perform Namespace processing.
false: Optionally do not perform Namespace processing (implies namespace-prefixes; default).
access: (parsing) read-only; (not parsing) read/write

xml.sax.handler.feature_namespace_prefixes

value: `"http://xml.org/sax/features/namespace-prefixes"`
true: Report the original prefixed names and attributes used for Namespace declarations.
false: Do not report attributes used for Namespace declarations, and optionally do not report original prefixed names (default).
access: (parsing) read-only; (not parsing) read/write

xml.sax.handler.feature_string_interning

value: `"http://xml.org/sax/features/string-interning"`
true: All element names, prefixes, attribute names, Namespace URIs, and local names are interned using the built-in intern function.
false: Names are not necessarily interned, although they may be (default).
access: (parsing) read-only; (not parsing) read/write

xml.sax.handler.feature_validation

value: `"http://xml.org/sax/features/validation"`
true: Report all validation errors (implies external-general-entities and external-parameter-entities).
false: Do not report validation errors.
access: (parsing) read-only; (not parsing) read/write

xml.sax.handler.feature_external_ges

Warning

Enabling opens a vulnerability to external entity attacks [https://en.wikipedia.org/wiki/XML_external_entity_attack] if the parser is used with user-provided XML content. Please reflect on your threat model [https://en.wikipedia.org/wiki/Threat_model] before enabling this feature.

value: `"http://xml.org/sax/features/external-general-entities"`
true: Include all external general (text) entities.
false: Do not include external general entities.
access: (parsing) read-only; (not parsing) read/write

xml.sax.handler.feature_external_pes

value: `"http://xml.org/sax/features/external-parameter-entities"`

true: Include all external parameter entities, including the external DTD subset.
false: Do not include any external parameter entities, even the external DTD subset.
access: (parsing) read-only; (not parsing) read/write

xml.sax.handler.all_features

List of all features.

xml.sax.handler.property_lexical_handler

value: `"http://xml.org/sax/properties/lexical-handler"`
data type: xml.sax.handler.LexicalHandler (not supported in Python 2)
description: An optional extension handler for lexical events like comments.
access: read/write

xml.sax.handler.property_declaration_handler

value: `"http://xml.org/sax/properties/declaration-handler"`
data type: xml.sax.sax2lib.DeclHandler (not supported in Python 2)
description: An optional extension handler for DTD-related events other than notations and unparsed entities.
access: read/write

xml.sax.handler.property_dom_node

value: `"http://xml.org/sax/properties/dom-node"`
data type: org.w3c.dom.Node (not supported in Python 2)
description: When parsing, the current DOM node being visited if this is a DOM iterator; when not parsing, the root DOM node for iteration.
access: (parsing) read-only; (not parsing) read/write

xml.sax.handler.property_xml_string

value: `"http://xml.org/sax/properties/xml-string"`
data type: Bytes
description: The literal string of characters that was the source for the current event.
access: read-only

xml.sax.handler.all_properties

List of all known property names.

ContentHandler Objects

Users are expected to subclass `ContentHandler` to support their application. The following methods are called by the parser on the appropriate events in the input document:

ContentHandler.setDocumentLocator(*locator*)

Called by the parser to give the application a locator for locating the origin of document events.

SAX parsers are strongly encouraged (though not absolutely required) to supply a locator: if it does so, it must supply the locator to the application by invoking this method before invoking any of the other methods in the DocumentHandler interface.

The locator allows the application to determine the end position of any document-related event, even if the parser is not reporting an error. Typically, the application will use this information for reporting its own errors (such as character content that does not match an application’s business rules). The information returned by the locator is probably not sufficient for use with a search engine.

Note that the locator will return correct information only during the invocation of the events in this interface. The application should not attempt to use it at any other time.

ContentHandler.startDocument()

Receive notification of the beginning of a document.

The SAX parser will invoke this method only once, before any other methods in this interface or in DTDHandler (except for `setDocumentLocator()`).

ContentHandler.endDocument()

Receive notification of the end of a document.

The SAX parser will invoke this method only once, and it will be the last method invoked during the parse. The parser shall not invoke this method until it has either abandoned parsing (because of an unrecoverable error) or reached the end of input.

ContentHandler.startPrefixMapping(*prefix*, *uri*)

Begin the scope of a prefix-URI Namespace mapping.

The information from this event is not necessary for normal Namespace processing: the SAX XML reader will automatically replace prefixes for element and attribute names when the `feature_namespaces` feature is enabled (the default).

There are cases, however, when applications need to use prefixes in character data or in attribute values, where they cannot safely be expanded automatically; the `startPrefixMapping()` and `endPrefixMapping()` events supply the information to the application to expand prefixes in those contexts itself, if necessary.

Note that `startPrefixMapping()` and `endPrefixMapping()` events are not guaranteed to be properly nested relative to each-other: all `startPrefixMapping()` events will occur before the corresponding `startElement()` event, and all `endPrefixMapping()` events will occur after the corresponding `endElement()` event, but their order is not guaranteed.

ContentHandler.endPrefixMapping(*prefix*)

End the scope of a prefix-URI mapping.

See `startPrefixMapping()` for details. This event will always occur after the corresponding `endElement()` event, but the order of `endPrefixMapping()` events is not otherwise guaranteed.

ContentHandler.startElement(*name*, *attrs*)

Signals the start of an element in non-namespace mode.

The *name* parameter contains the raw XML 1.0 name of the element type as a string and the *attrs* parameter holds an object of the Attributes interface containing the attributes of the element. The object passed as *attrs* may be re-used by the parser; holding on to a reference to it is not a reliable way to keep a copy of the attributes. To keep a copy of the attributes, use the `copy()` method of the *attrs* object.

ContentHandler.endElement(*name*)

Signals the end of an element in non-namespace mode.

The *name* parameter contains the name of the element type, just as with the `startElement()` event.

ContentHandler.startElementNS(*name*, *qname*, *attrs*)

Signals the start of an element in namespace mode.

The *name* parameter contains the name of the element type as a `(uri, localname)` tuple, the *qname* parameter contains the raw XML 1.0 name used in the source document, and the *attrs* parameter holds an instance of the AttributesNS interface containing the attributes of the element. If no namespace is associated with the element, the *uri* component of *name* will be `None`. The object passed as *attrs* may be re-used by the parser; holding on to a reference to it is not a reliable way to keep a copy of the attributes. To keep a copy of the attributes, use the **`copy()`** method of the *attrs* object.

Parsers may set the *qname* parameter to `None`, unless the `feature_namespace_prefixes` feature is activated.

ContentHandler.endElementNS(*name*, *qname*)

Signals the end of an element in namespace mode.

The *name* parameter contains the name of the element type, just as with the **`startElementNS()`** method, likewise the *qname* parameter.

ContentHandler.characters(*content*)

Receive notification of character data.

The Parser will call this method to report each chunk of character data. SAX parsers may return all contiguous character data in a single chunk, or they may split it into several chunks; however, all of the characters in any single event must come from the same external entity so that the Locator provides useful information.

content may be a string or bytes instance; the `expat` reader module always produces strings.

Note

The earlier SAX 1 interface provided by the Python XML Special Interest Group used a more Java-like interface for this method. Since most parsers used from Python did not take advantage of the older interface, the simpler signature was chosen to replace it. To convert old code to the new interface, use *content* instead of slicing content with the old *offset* and *length* parameters.

ContentHandler.ignorableWhitespace(*whitespace*)

Receive notification of ignorable whitespace in element content.

Validating Parsers must use this method to report each chunk of ignorable whitespace (see the W3C XML 1.0 recommendation, section 2.10): non-validating parsers may also use this method if they are capable of parsing and using content models.

SAX parsers may return all contiguous whitespace in a single chunk, or they may split it into several chunks; however, all of the characters in any single event must come from the same external entity, so that the Locator provides useful information.

ContentHandler.processingInstruction(*target*, *data*)

Receive notification of a processing instruction.

The Parser will invoke this method once for each processing instruction found: note that processing instructions may occur before or after the main document element.

A SAX parser should never report an XML declaration (XML 1.0, section 2.8) or a text declaration (XML 1.0, section 4.3.1) using this method.

ContentHandler.skippedEntity(*name*)

Receive notification of a skipped entity.

The Parser will invoke this method once for each entity skipped. Non-validating processors may skip entities if they have not seen the declarations (because, for example, the entity was declared in an external DTD subset). All processors may skip external entities, depending on the values of the `feature_external_ges` and the `feature_external_pes` properties.

DTDHandler Objects

`DTDHandler` instances provide the following methods:

DTDHandler.notationDecl(*name*, *publicId*, *systemId*)

Handle a notation declaration event.

DTDHandler.unparsedEntityDecl(*name*, *publicId*, *systemId*, *ndata*)

Handle an unparsed entity declaration event.

EntityResolver Objects

EntityResolver.resolveEntity(*publicId*, *systemId*)

Resolve the system identifier of an entity and return either the system identifier to read from as a string, or an InputSource to read from. The default implementation returns *systemId*.

ErrorHandler Objects

Objects with this interface are used to receive error and warning information from the `XMLReader`. If you create an object that implements this interface, then register the object with your `XMLReader`, the parser will call the methods in your object to report all warnings and errors. There are three levels of errors available: warnings, (possibly) recoverable errors, and unrecoverable errors. All methods take a `SAXParseException` as the only parameter. Errors and warnings may be converted to an exception by raising the passed-in exception object.

ErrorHandler.error(*exception*)

> Called when the parser encounters a recoverable error. If this method does not raise an exception, parsing may continue, but further document information should not be expected by the application. Allowing the parser to continue may allow additional errors to be discovered in the input document.

ErrorHandler.fatalError(*exception*)

> Called when the parser encounters an error it cannot recover from; parsing is expected to terminate when this method returns.

ErrorHandler.warning(*exception*)

> Called when the parser presents minor warning information to the application. Parsing is expected to continue when this method returns, and document information will continue to be passed to the application. Raising an exception in this method will cause parsing to end.

LexicalHandler Objects

Optional SAX2 handler for lexical events.

This handler is used to obtain lexical information about an XML document. Lexical information includes information describing the document encoding used and XML comments embedded in the document, as well as section boundaries for the DTD and for any CDATA sections. The lexical handlers are used in the same manner as content handlers.

Set the LexicalHandler of an XMLReader by using the setProperty method with the property identifier `'http://xml.org/sax/properties/lexical-handler'`.

LexicalHandler.comment(*content*)

> Reports a comment anywhere in the document (including the DTD and outside the document element).

LexicalHandler.startDTD(*name*, *public_id*, *system_id*)

> Reports the start of the DTD declarations if the document has an associated DTD.

LexicalHandler.endDTD()

> Reports the end of DTD declaration.

LexicalHandler.startCDATA()

> Reports the start of a CDATA marked section.
>
> The contents of the CDATA marked section will be reported through the characters handler.

LexicalHandler.endCDATA()

> Reports the end of a CDATA marked section.

`xml.sax.saxutils` — SAX Utilities

Source code: Lib/xml/sax/saxutils.py
[https://github.com/python/cpython/tree/3.14/Lib/xml/sax/saxutils.py]

The module `xml.sax.saxutils` contains a number of classes and functions that are commonly useful when creating SAX applications, either in direct use, or as base classes.

xml.sax.saxutils.escape(*data*, *entities={}*)

Escape `'&'`, `'<'`, and `'>'` in a string of data.

You can escape other strings of data by passing a dictionary as the optional *entities* parameter. The keys and values must all be strings; each key will be replaced with its corresponding value. The characters `'&'`, `'<'` and `'>'` are always escaped, even if *entities* is provided.

Note

This function should only be used to escape characters that can't be used directly in XML. Do not use this function as a general string translation function.

xml.sax.saxutils.unescape(*data*, *entities={}*)

Unescape `'&amp;'`, `'&lt;'`, and `'&gt;'` in a string of data.

You can unescape other strings of data by passing a dictionary as the optional *entities* parameter. The keys and values must all be strings; each key will be replaced with its corresponding value. `'&amp;'`, `'&lt;'`, and `'&gt;'` are always unescaped, even if *entities* is provided.

xml.sax.saxutils.quoteattr(*data*, *entities={}*)

Similar to **`escape()`**, but also prepares *data* to be used as an attribute value. The return value is a quoted version of *data* with any additional required replacements. **`quoteattr()`** will select a quote character based on the content of *data*, attempting to avoid encoding any quote characters in the string. If both single- and double-quote characters are already in *data*, the double-quote characters will be encoded and *data* will be wrapped in double-quotes. The resulting string can be used directly as an attribute value:

```
>>> print("<element attr=%s>" % quoteattr("ab ' cd \" ef"))
<element attr="ab ' cd &quot; ef">
```

This function is useful when generating attribute values for HTML or any SGML using the reference concrete syntax.

class xml.sax.saxutils.XMLGenerator(*out=None*, *encoding='iso-8859-1'*, *short_empty_elements=False*)

This class implements the **`ContentHandler`** interface by writing SAX events back into an XML document. In other words, using an **`XMLGenerator`** as the content handler will reproduce the original document being parsed. *out* should be a file-like object which will default to *sys.stdout*. *encoding* is the encoding of the output stream which defaults to `'iso-8859-1'`. *short_empty_elements* controls the formatting of elements that contain no content: if `False` (the default) they are emitted as a pair of start/end tags, if set to `True` they are emitted as a single self-closed tag.

Changed in version 3.2: Added the *short_empty_elements* parameter.

class xml.sax.saxutils.XMLFilterBase(*base*)

This class is designed to sit between an `XMLReader` and the client application's event handlers. By default, it does nothing but pass requests up to the reader and events on to the handlers unmodified, but subclasses can override specific methods to modify the event stream or the configuration requests as they pass through.

xml.sax.saxutils.prepare_input_source(*source*, *base*='')

This function takes an input source and an optional base URL and returns a fully resolved `InputSource` object ready for reading. The input source can be given as a string, a file-like object, or an `InputSource` object; parsers will use this function to implement the polymorphic *source* argument to their `parse()` method.

`xml.sax.xmlreader` — Interface for XML parsers

Source code: Lib/xml/sax/xmlreader.py
[https://github.com/python/cpython/tree/3.14/Lib/xml/sax/xmlreader.py]

SAX parsers implement the `XMLReader` interface. They are implemented in a Python module, which must provide a function `create_parser()`. This function is invoked by `xml.sax.make_parser()` with no arguments to create a new parser object.

class xml.sax.xmlreader.XMLReader

Base class which can be inherited by SAX parsers.

class xml.sax.xmlreader.IncrementalParser

In some cases, it is desirable not to parse an input source at once, but to feed chunks of the document as they get available. Note that the reader will normally not read the entire file, but read it in chunks as well; still `parse()` won’t return until the entire document is processed. So these interfaces should be used if the blocking behaviour of `parse()` is not desirable.

When the parser is instantiated it is ready to begin accepting data from the feed method immediately. After parsing has been finished with a call to close the reset method must be called to make the parser ready to accept new data, either from feed or using the parse method.

Note that these methods must *not* be called during parsing, that is, after parse has been called and before it returns.

By default, the class also implements the parse method of the XMLReader interface using the feed, close and reset methods of the IncrementalParser interface as a convenience to SAX 2.0 driver writers.

class xml.sax.xmlreader.Locator

Interface for associating a SAX event with a document location. A locator object will return valid results only during calls to DocumentHandler methods; at any other time, the results are unpredictable. If information is not available, methods may return `None`.

class xml.sax.xmlreader.InputSource(*system_id=None*)

Encapsulation of the information needed by the `XMLReader` to read entities.

This class may include information about the public identifier, system identifier, byte stream (possibly with character encoding information) and/or the character stream of an entity.

Applications will create objects of this class for use in the `XMLReader.parse()` method and for returning from EntityResolver.resolveEntity.

An `InputSource` belongs to the application, the `XMLReader` is not allowed to modify `InputSource` objects passed to it from the application, although it may make copies and modify those.

class xml.sax.xmlreader.AttributesImpl(*attrs*)

This is an implementation of the **`Attributes`** interface (see section The Attributes Interface). This is a dictionary-like object which represents the element attributes in a **`startElement()`** call. In addition to the most useful dictionary operations, it supports a number of other methods as described by the interface. Objects of this class should be instantiated by readers; *attrs* must be a dictionary-like object containing a mapping from attribute names to attribute values.

class xml.sax.xmlreader.AttributesNSImpl(*attrs*, *qnames*)

Namespace-aware variant of **`AttributesImpl`**, which will be passed to **`startElementNS()`**. It is derived from **`AttributesImpl`**, but understands attribute names as two-tuples of *namespaceURI* and *localname*. In addition, it provides a number of methods expecting qualified names as they appear in the original document. This class implements the **`AttributesNS`** interface (see section The AttributesNS Interface).

XMLReader Objects

The **`XMLReader`** interface supports the following methods:

XMLReader.parse(*source*)

Process an input source, producing SAX events. The *source* object can be a system identifier (a string identifying the input source – typically a file name or a URL), a **`pathlib.Path`** or path-like object, or an **`InputSource`** object. When **`parse()`** returns, the input is completely processed, and the parser object can be discarded or reset.

Changed in version 3.5: Added support of character streams.

Changed in version 3.8: Added support of path-like objects.

XMLReader.getContentHandler()

Return the current `ContentHandler`.

XMLReader.setContentHandler(*handler*)

Set the current `ContentHandler`. If no `ContentHandler` is set, content events will be discarded.

XMLReader.getDTDHandler()

Return the current `DTDHandler`.

XMLReader.setDTDHandler(*handler*)

Set the current `DTDHandler`. If no `DTDHandler` is set, DTD events will be discarded.

XMLReader.getEntityResolver()

Return the current `EntityResolver`.

XMLReader.setEntityResolver(*handler*)

Set the current `EntityResolver`. If no `EntityResolver` is set, attempts to resolve an external entity will result in opening the system identifier for the entity, and fail if it is not available.

XMLReader.getErrorHandler()

Return the current `ErrorHandler`.

XMLReader.setErrorHandler(*handler*)

Set the current error handler. If no `ErrorHandler` is set, errors will be raised as exceptions, and warnings will be printed.

XMLReader.setLocale(*locale*)

Allow an application to set the locale for errors and warnings.

SAX parsers are not required to provide localization for errors and warnings; if they cannot support the requested locale, however, they must raise a SAX exception. Applications may request a locale change in the middle of a parse.

XMLReader.getFeature(*featurename*)

Return the current setting for feature *featurename*. If the feature is not recognized, `SAXNotRecognizedException` is raised. The well-known featurenames are listed in the module `xml.sax.handler`.

XMLReader.setFeature(*featurename*, *value*)

Set the *featurename* to *value*. If the feature is not recognized, `SAXNotRecognizedException` is raised. If the feature or its setting is not supported by the parser, *SAXNotSupportedException* is raised.

XMLReader.getProperty(*propertyname*)

Return the current setting for property *propertyname*. If the property is not recognized, a `SAXNotRecognizedException` is raised. The well-known propertynames are listed in the module `xml.sax.handler`.

XMLReader.setProperty(*propertyname*, *value*)

Set the *propertyname* to *value*. If the property is not recognized, `SAXNotRecognizedException` is raised. If the property or its setting is not supported by the parser, *SAXNotSupportedException* is raised.

IncrementalParser Objects

Instances of [`IncrementalParser`] offer the following additional methods:

IncrementalParser.feed(*data*)

Process a chunk of *data*.

IncrementalParser.close()

Assume the end of the document. That will check well-formedness conditions that can be checked only at the end, invoke handlers, and may clean up resources allocated during parsing.

IncrementalParser.reset()

This method is called after close has been called to reset the parser so that it is ready to parse new documents. The results of calling parse or feed after close without calling reset are undefined.

Locator Objects

Instances of `Locator` provide these methods:

Locator.getColumnNumber()

Return the column number where the current event begins.

Locator.getLineNumber()

Return the line number where the current event begins.

Locator.getPublicId()

Return the public identifier for the current event.

Locator.getSystemId()

Return the system identifier for the current event.

InputSource Objects

InputSource.setPublicId(*id*)

Sets the public identifier of this `InputSource`.

InputSource.getPublicId()

Returns the public identifier of this `InputSource`.

InputSource.setSystemId(*id*)

Sets the system identifier of this `InputSource`.

InputSource.getSystemId()

Returns the system identifier of this `InputSource`.

InputSource.setEncoding(*encoding*)

Sets the character encoding of this `InputSource`.

The encoding must be a string acceptable for an XML encoding declaration (see section 4.3.3 of the XML recommendation).

The encoding attribute of the `InputSource` is ignored if the `InputSource` also contains a character stream.

InputSource.getEncoding()

Get the character encoding of this InputSource.

InputSource.setByteStream(*bytefile*)

Set the byte stream (a binary file) for this input source.

The SAX parser will ignore this if there is also a character stream specified, but it will use a byte stream in preference to opening a URI connection itself.

If the application knows the character encoding of the byte stream, it should set it with the setEncoding method.

InputSource.getByteStream()

Get the byte stream for this input source.

The getEncoding method will return the character encoding for this byte stream, or `None` if unknown.

InputSource.setCharacterStream(*charfile*)

Set the character stream (a text file) for this input source.

If there is a character stream specified, the SAX parser will ignore any byte stream and will not attempt to open a URI connection to the system identifier.

InputSource.getCharacterStream()

Get the character stream for this input source.

The `Attributes` Interface

`Attributes` objects implement a portion of the mapping protocol, including the methods **`copy()`**, **`get()`**, **`__contains__()`**, **`items()`**, **`keys()`**, and **`values()`**. The following methods are also provided:

Attributes.getLength()

Return the number of attributes.

Attributes.getNames()

Return the names of the attributes.

Attributes.getType(*name*)

Returns the type of the attribute *name*, which is normally `'CDATA'`.

Attributes.getValue(*name*)

Return the value of attribute *name*.

The `AttributesNS` Interface

This interface is a subtype of the **`Attributes`** interface (see section The Attributes Interface). All methods supported by that interface are also available on **`AttributesNS`** objects.

The following methods are also available:

AttributesNS.getValueByQName(*name*)

Return the value for a qualified name.

AttributesNS.getNameByQName(*name*)

Return the `(namespace, localname)` pair for a qualified *name*.

AttributesNS.getQNameByName(*name*)

Return the qualified name for a `(namespace, localname)` pair.

AttributesNS.getQNames()

Return the qualified names of all attributes.

`xml.parsers.expat` — Fast XML parsing using Expat

Note

If you need to parse untrusted or unauthenticated data, see XML security.

The `xml.parsers.expat` module is a Python interface to the Expat non-validating XML parser. The module provides a single extension type, `xmlparser`, that represents the current state of an XML parser. After an `xmlparser` object has been created, various attributes of the object can be set to handler functions. When an XML document is then fed to the parser, the handler functions are called for the character data and markup in the XML document.

This module uses the `pyexpat` module to provide access to the Expat parser. Direct use of the `pyexpat` module is deprecated.

This module provides one exception and one type object:

exception xml.parsers.expat.ExpatError

The exception raised when Expat reports an error. See section ExpatError Exceptions for more information on interpreting Expat errors.

exception xml.parsers.expat.error

Alias for `ExpatError`.

xml.parsers.expat.XMLParserType

The type of the return values from the `ParserCreate()` function.

The `xml.parsers.expat` module contains two functions:

xml.parsers.expat.ErrorString(*errno*)

Returns an explanatory string for a given error number *errno*.

xml.parsers.expat.ParserCreate(*encoding=None*, *namespace_separator=None*)

Creates and returns a new **`xmlparser`** object. *encoding*, if specified, must be a string naming the encoding used by the XML data. Expat doesn't support as many encodings as Python does, and its repertoire of encodings can't be extended; it supports UTF-8, UTF-16, ISO-8859-1 (Latin1), and ASCII. If *encoding* [[1]] is given it will override the implicit or explicit encoding of the document.

Parsers created through **`ParserCreate()`** are called "root" parsers, in the sense that they do not have any parent parser attached. Non-root parsers are created by `parser.ExternalEntityParserCreate`.

Expat can optionally do XML namespace processing for you, enabled by providing a value for *namespace_separator*. The value must be a one-character string; a `ValueError` will be raised if the string has an illegal length (`None` is considered the same as omission). When namespace processing is enabled, element type names and attribute names that belong to a namespace will be expanded. The element name passed to the element handlers **`StartElementHandler`** and **`EndElementHandler`** will be the concatenation of the namespace URI, the namespace separator character, and the local part of the name. If the namespace separator is a zero byte (`chr(0)`) then the namespace URI and the local part will be concatenated without any separator.

For example, if *namespace_separator* is set to a space character (`' '`) and the following document is parsed:

```
<?xml version="1.0"?>
<root xmlns    = "http://default-namespace.org/"
      xmlns:py = "http://www.python.org/ns/">
  <py:elem1 />
  <elem2 xmlns="" />
</root>
```

`StartElementHandler` will receive the following strings for each element:

```
http://default-namespace.org/ root
http://www.python.org/ns/ elem1
elem2
```

Due to limitations in the `Expat` library used by **`pyexpat`**, the **`xmlparser`** instance returned can only be used to parse a single XML document. Call `ParserCreate` for each document to provide unique parser instances.

See also

The Expat XML Parser [http://www.libexpat.org/]
Home page of the Expat project.

XMLParser Objects

`xmlparser` objects have the following methods:

xmlparser.Parse(*data*[, *isfinal*])

Parses the contents of the string *data*, calling the appropriate handler functions to process the parsed data. *isfinal* must be true on the final call to this method; it allows the parsing of a single file in fragments, not the submission of multiple files. *data* can be the empty string at any time.

xmlparser.ParseFile(*file*)

Parse XML data reading from the object *file*. *file* only needs to provide the `read(nbytes)` method, returning the empty string when there’s no more data.

xmlparser.SetBase(*base*)

Sets the base to be used for resolving relative URIs in system identifiers in declarations. Resolving relative identifiers is left to the application: this value will be passed through as the *base* argument to the **`ExternalEntityRefHandler()`**, **`NotationDeclHandler()`**, and **`UnparsedEntityDeclHandler()`** functions.

xmlparser.GetBase()

Returns a string containing the base set by a previous call to **`SetBase()`**, or `None` if **`SetBase()`** hasn’t been called.

xmlparser.GetInputContext()

Returns the input data that generated the current event as a string. The data is in the encoding of the entity which contains the text. When called while an event handler is not active, the return value is `None`.

xmlparser.ExternalEntityParserCreate(*context*[, *encoding*])

Create a “child” parser which can be used to parse an external parsed entity referred to by content parsed by the parent parser. The *context* parameter should be the string passed to the **`ExternalEntityRefHandler()`** handler function, described below. The child parser is created with the `ordered_attributes` and `specified_attributes` set to the values of this parser.

xmlparser.SetParamEntityParsing(*flag*)

Control parsing of parameter entities (including the external DTD subset). Possible *flag* values are **`XML_PARAM_ENTITY_PARSING_NEVER`**, **`XML_PARAM_ENTITY_PARSING_UNLESS_STANDALONE`** and **`XML_PARAM_ENTITY_PARSING_ALWAYS`**. Return true if setting the flag was successful.

xmlparser.UseForeignDTD([*flag*])

Calling this with a true value for *flag* (the default) will cause Expat to call the `ExternalEntityRefHandler` with `None` for all arguments to allow an alternate DTD to be loaded. If the document does not contain a document type declaration, the `ExternalEntityRefHandler` will still be called, but the `StartDoctypeDeclHandler` and `EndDoctypeDeclHandler` will not be called.

Passing a false value for *flag* will cancel a previous call that passed a true value, but otherwise has no effect.

This method can only be called before the `Parse()` or `ParseFile()` methods are called; calling it after either of those have been called causes `ExpatError` to be raised with the `code` attribute set to `errors.codes[errors.XML_ERROR_CANT_CHANGE_FEATURE_ONCE_PARSING]`.

xmlparser.SetReparseDeferralEnabled(*enabled*)

Warning

Calling `SetReparseDeferralEnabled(False)` has security implications, as detailed below; please make sure to understand these consequences prior to using the `SetReparseDeferralEnabled` method.

Expat 2.6.0 introduced a security mechanism called “reparse deferral” where instead of causing denial of service through quadratic runtime from reparsing large tokens, reparsing of unfinished tokens is now delayed by default until a sufficient amount of input is reached. Due to this delay, registered handlers may — depending of the sizing of input chunks pushed to Expat — no longer be called right after pushing new input to the parser. Where immediate feedback and taking over responsibility of protecting against denial of service from large tokens are both wanted, calling `SetReparseDeferralEnabled(False)` disables reparse deferral for the current Expat parser instance, temporarily or altogether. Calling `SetReparseDeferralEnabled(True)` allows re-enabling reparse deferral.

Note that **`SetReparseDeferralEnabled()`** has been backported to some prior releases of CPython as a security fix. Check for availability of **`SetReparseDeferralEnabled()`** using **`hasattr()`** if used in code running across a variety of Python versions.

Added in version 3.13.

xmlparser.GetReparseDeferralEnabled()

Returns whether reparse deferral is currently enabled for the given Expat parser instance.

Added in version 3.13.

`xmlparser` objects have the following methods to mitigate some common XML vulnerabilities.

xmlparser.SetAllocTrackerActivationThreshold(*threshold*, /)

Sets the number of allocated bytes of dynamic memory needed to activate protection against disproportionate use of RAM.

By default, parser objects have an allocation activation threshold of 64 MiB, or equivalently 67,108,864 bytes.

An **`ExpatError`** is raised if this method is called on a non-root parser. The corresponding **`lineno`** and **`offset`** should not be used as they may have no special meaning.

Added in version 3.14.1.

xmlparser.SetAllocTrackerMaximumAmplification(*max_factor*, /)

Sets the maximum amplification factor between direct input and bytes of dynamic memory allocated.

The amplification factor is calculated as `allocated / direct` while parsing, where `direct` is the number of bytes read from the primary document in parsing and `allocated` is the number of bytes of dynamic memory allocated in the parser hierarchy.

The *max_factor* value must be a non-NaN **`float`** value greater than or equal to 1.0. Amplification factors greater than 100.0 can be observed near the start of parsing even with benign files in practice. In particular, the activation threshold should be carefully chosen to avoid false positives.

By default, parser objects have a maximum amplification factor of 100.0.

An `ExpatError` is raised if this method is called on a non-root parser or if *max_factor* is outside the valid range. The corresponding `lineno` and `offset` should not be used as they may have no special meaning.

Note

The maximum amplification factor is only considered if the threshold that can be adjusted by `SetAllocTrackerActivationThreshold()` is exceeded.

Added in version 3.14.1.

`xmlparser` objects have the following attributes:

xmlparser.buffer_size

The size of the buffer used when `buffer_text` is true. A new buffer size can be set by assigning a new integer value to this attribute. When the size is changed, the buffer will be flushed.

xmlparser.buffer_text

Setting this to true causes the `xmlparser` object to buffer textual content returned by Expat to avoid multiple calls to the `CharacterDataHandler()` callback whenever possible. This can improve performance substantially since Expat normally breaks character data into chunks at every line ending. This attribute is false by default, and may be changed at any time. Note that when it is false, data that does not contain newlines may be chunked too.

xmlparser.buffer_used

If `buffer_text` is enabled, the number of bytes stored in the buffer. These bytes represent UTF-8 encoded text. This attribute has no meaningful interpretation when `buffer_text` is false.

xmlparser.ordered_attributes

Setting this attribute to a non-zero integer causes the attributes to be reported as a list rather than a dictionary. The attributes are presented in the order found in the document text. For each attribute, two list entries are presented: the attribute name and the attribute value. (Older versions of this module also used this format.) By default, this attribute is false; it may be changed at any time.

xmlparser.specified_attributes

If set to a non-zero integer, the parser will report only those attributes which were specified in the document instance and not those which were derived from attribute declarations. Applications which set this need to be especially careful to use what additional information is available from the declarations as needed to comply with the standards for the behavior of XML processors. By default, this attribute is false; it may be changed at any time.

The following attributes contain values relating to the most recent error encountered by an **`xmlparser`** object, and will only have correct values once a call to **`Parse()`** or **`ParseFile()`** has raised an `xml.parsers.expat.ExpatError` exception.

xmlparser.ErrorByteIndex

Byte index at which an error occurred.

xmlparser.ErrorCode

Numeric code specifying the problem. This value can be passed to the `ErrorString()` function, or compared to one of the constants defined in the `errors` object.

xmlparser.ErrorColumnNumber

Column number at which an error occurred.

xmlparser.ErrorLineNumber

Line number at which an error occurred.

The following attributes contain values relating to the current parse location in an **`xmlparser`** object. During a callback reporting a parse event they indicate the location of the first of the sequence of characters that generated the event. When called outside of a callback, the position indicated will be just past the last parse event (regardless of whether there was an associated callback).

xmlparser.CurrentByteIndex

Current byte index in the parser input.

xmlparser.CurrentColumnNumber

Current column number in the parser input.

xmlparser.CurrentLineNumber

Current line number in the parser input.

Here is the list of handlers that can be set. To set a handler on an **`xmlparser`** object *o*, use `o.handlername = func`. *handlername* must be taken from the following list, and *func* must be a callable object accepting the correct number of arguments. The arguments are all strings, unless otherwise stated.

xmlparser.XmlDeclHandler(*version*, *encoding*, *standalone*)

Called when the XML declaration is parsed. The XML declaration is the (optional) declaration of the applicable version of the XML recommendation, the encoding of the document text, and an optional “standalone” declaration. *version* and *encoding* will be strings, and *standalone* will be `1` if the document is declared standalone, `0` if it is declared not to be standalone, or `-1` if the standalone clause was omitted. This is only available with Expat version 1.95.0 or newer.

xmlparser.StartDoctypeDeclHandler(*doctypeName*, *systemId*, *publicId*, *has_internal_subset*)

Called when Expat begins parsing the document type declaration (`<!DOCTYPE ...`). The *doctypeName* is provided exactly as presented. The *systemId* and *publicId* parameters give the system and public identifiers if specified, or `None` if omitted. *has_internal_subset* will be true if the document contains and internal document declaration subset. This requires Expat version 1.2 or newer.

xmlparser.EndDoctypeDeclHandler()

Called when Expat is done parsing the document type declaration. This requires Expat version 1.2 or newer.

xmlparser.ElementDeclHandler(*name*, *model*)

Called once for each element type declaration. *name* is the name of the element type, and *model* is a representation of the content model.

xmlparser.AttlistDeclHandler(*elname*, *attname*, *type*, *default*, *required*)

Called for each declared attribute for an element type. If an attribute list declaration declares three attributes, this handler is called three times, once for each attribute. *elname* is the name of the element to which the declaration applies and *attname* is the name of the attribute declared. The attribute type is a string passed as *type*; the possible values are `'CDATA'`, `'ID'`, `'IDREF'`, … *default* gives the default value for the attribute used when the attribute is not specified by the document instance, or `None` if there is no default value (`#IMPLIED` values). If the attribute is required to be given in the document instance, *required* will be true. This requires Expat version 1.95.0 or newer.

xmlparser.StartElementHandler(*name*, *attributes*)

Called for the start of every element. *name* is a string containing the element name, and *attributes* is the element attributes. If `ordered_attributes` is true, this is a list (see `ordered_attributes` for a full description). Otherwise it’s a dictionary mapping names to values.

xmlparser.EndElementHandler(*name*)

Called for the end of every element.

xmlparser.ProcessingInstructionHandler(*target*, *data*)

Called for every processing instruction.

xmlparser.CharacterDataHandler(*data*)

Called for character data. This will be called for normal character data, CDATA marked content, and ignorable whitespace. Applications which must distinguish these cases can use the `StartCdataSectionHandler`, `EndCdataSectionHandler`, and `ElementDeclHandler` callbacks to collect the required information. Note that the character data may be chunked even if it is short and so you may receive more than one call to `CharacterDataHandler()`. Set the `buffer_text` instance attribute to `True` to avoid that.

xmlparser.UnparsedEntityDeclHandler(*entityName*, *base*, *systemId*, *publicId*, *notationName*)

Called for unparsed (NDATA) entity declarations. This is only present for version 1.2 of the Expat library; for more recent versions, use `EntityDeclHandler` instead. (The underlying function in the Expat library has been declared obsolete.)

xmlparser.EntityDeclHandler(*entityName*, *is_parameter_entity*, *value*, *base*, *systemId*, *publicId*, *notationName*)

Called for all entity declarations. For parameter and internal entities, *value* will be a string giving the declared contents of the entity; this will be `None` for external entities. The *notationName* parameter will be `None` for parsed entities, and the name of the notation for unparsed entities. *is_parameter_entity* will be true if the entity is a parameter entity or false for general entities (most applications only need to be concerned with general entities). This is only available starting with version 1.95.0 of the Expat library.

xmlparser.NotationDeclHandler(*notationName*, *base*, *systemId*, *publicId*)

Called for notation declarations. *notationName*, *base*, and *systemId*, and *publicId* are strings if given. If the public identifier is omitted, *publicId* will be `None`.

xmlparser.StartNamespaceDeclHandler(*prefix*, *uri*)

Called when an element contains a namespace declaration. Namespace declarations are processed before the **`StartElementHandler`** is called for the element on which declarations are placed.

xmlparser.EndNamespaceDeclHandler(*prefix*)

Called when the closing tag is reached for an element that contained a namespace declaration. This is called once for each namespace declaration on the element in the reverse of the order for which the **`StartNamespaceDeclHandler`** was called to indicate the start of each namespace declaration's scope. Calls to this handler are made after the corresponding **`EndElementHandler`** for the end of the element.

xmlparser.CommentHandler(*data*)

Called for comments. *data* is the text of the comment, excluding the leading `'<!--'` and trailing `'-->'`.

xmlparser.StartCdataSectionHandler()

Called at the start of a CDATA section. This and **`EndCdataSectionHandler`** are needed to be able to identify the syntactical start and end for CDATA sections.

xmlparser.EndCdataSectionHandler()

Called at the end of a CDATA section.

xmlparser.DefaultHandler(*data*)

Called for any characters in the XML document for which no applicable handler has been specified. This means characters that are part of a construct which could be reported, but for which no handler has been supplied.

xmlparser.DefaultHandlerExpand(*data*)

This is the same as the **`DefaultHandler()`**, but doesn’t inhibit expansion of internal entities. The entity reference will not be passed to the default handler.

xmlparser.NotStandaloneHandler()

Called if the XML document hasn’t been declared as being a standalone document. This happens when there is an external subset or a reference to a parameter entity, but the XML declaration does not set standalone to `yes` in an XML declaration. If this handler returns `0`, then the parser will raise an **`XML_ERROR_NOT_STANDALONE`** error. If this handler is not set, no exception is raised by the parser for this condition.

xmlparser.ExternalEntityRefHandler(*context*, *base*, *systemId*, *publicId*)

Warning

Implementing a handler that accesses local files and/or the network may create a vulnerability to external entity attacks [https://en.wikipedia.org/wiki/XML_external_entity_attack] if **`xmlparser`** is used with user-provided XML content. Please reflect on your threat model [https://en.wikipedia.org/wiki/Threat_model] before implementing this handler.

Called for references to external entities. *base* is the current base, as set by a previous call to **`SetBase()`**. The public and system identifiers, *systemId* and *publicId*, are strings if given; if the public identifier is not given, *publicId* will be `None`. The *context* value is opaque and should only be used as described below.

For external entities to be parsed, this handler must be implemented. It is responsible for creating the sub-parser using `ExternalEntityParserCreate(context)`, initializing it with the appropriate callbacks, and parsing the entity. This handler should return an integer; if it returns `0`, the parser will raise an **`XML_ERROR_EXTERNAL_ENTITY_HANDLING`** error, otherwise parsing will continue.

If this handler is not provided, external entities are reported by the **`DefaultHandler`** callback, if provided.

ExpatError Exceptions

`ExpatError` exceptions have a number of interesting attributes:

ExpatError.code

Expat’s internal error number for the specific error. The `errors.messages` dictionary maps these error numbers to Expat’s error messages. For example:

```
from xml.parsers.expat import ParserCreate, ExpatError,
errors

p = ParserCreate()
try:
    p.Parse(some_xml_document)
except ExpatError as err:
    print("Error:", errors.messages[err.code])
```

The `errors` module also provides error message constants and a dictionary `codes` mapping these messages back to the error codes, see below.

ExpatError.lineno

Line number on which the error was detected. The first line is numbered `1`.

ExpatError.offset

Character offset into the line where the error occurred. The first column is numbered `0`.

Example

The following program defines three handlers that just print out their arguments.

```
import xml.parsers.expat

# 3 handler functions
def start_element(name, attrs):
    print('Start element:', name, attrs)
def end_element(name):
    print('End element:', name)
def char_data(data):
    print('Character data:', repr(data))
```

```
p = xml.parsers.expat.ParserCreate()

p.StartElementHandler = start_element
p.EndElementHandler = end_element
p.CharacterDataHandler = char_data

p.Parse("""<?xml version="1.0"?>
<parent id="top"><child1 name="paul">Text goes here</child1>
<child2 name="fred">More text</child2>
</parent>""", 1)
```

The output from this program is:

```
Start element: parent {'id': 'top'}
Start element: child1 {'name': 'paul'}
Character data: 'Text goes here'
End element: child1
Character data: '\n'
Start element: child2 {'name': 'fred'}
Character data: 'More text'
End element: child2
Character data: '\n'
End element: parent
```

Content Model Descriptions

Content models are described using nested tuples. Each tuple contains four values: the type, the quantifier, the name, and a tuple of children. Children are simply additional content model descriptions.

The values of the first two fields are constants defined in the `xml.parsers.expat.model` module. These constants can be collected in two groups: the model type group and the quantifier group.

The constants in the model type group are:

xml.parsers.expat.model.XML_CTYPE_ANY

The element named by the model name was declared to have a content model of `ANY`.

xml.parsers.expat.model.XML_CTYPE_CHOICE

The named element allows a choice from a number of options; this is used for content models such as `(A | B | C)`.

xml.parsers.expat.model.XML_CTYPE_EMPTY

Elements which are declared to be `EMPTY` have this model type.

xml.parsers.expat.model.XML_CTYPE_MIXED

xml.parsers.expat.model.XML_CTYPE_NAME

xml.parsers.expat.model.XML_CTYPE_SEQ

Models which represent a series of models which follow one after the other are indicated with this model type. This is used for models such as `(A, B, C)`.

The constants in the quantifier group are:

xml.parsers.expat.model.XML_CQUANT_NONE

No modifier is given, so it can appear exactly once, as for `A`.

xml.parsers.expat.model.XML_CQUANT_OPT

The model is optional: it can appear once or not at all, as for `A?`.

xml.parsers.expat.model.XML_CQUANT_PLUS

The model must occur one or more times (like `A+`).

xml.parsers.expat.model.XML_CQUANT_REP

The model must occur zero or more times, as for `A*`.

Expat error constants

The following constants are provided in the `xml.parsers.expat.errors` module. These constants are useful in interpreting some of the attributes of the **`ExpatError`** exception objects raised when an error has occurred. Since for backwards compatibility reasons, the constants' value is the error *message* and not the numeric error *code*, you do this by comparing its `code` attribute with `errors.codes[errors.XML_ERROR_CONSTANT_NAME]`.

The `errors` module has the following attributes:

xml.parsers.expat.errors.codes

A dictionary mapping string descriptions to their error codes.

Added in version 3.2.

xml.parsers.expat.errors.messages

A dictionary mapping numeric error codes to their string descriptions.

Added in version 3.2.

xml.parsers.expat.errors.XML_ERROR_ASYNC_ENTITY

xml.parsers.expat.errors.XML_ERROR_ATTRIBUTE_EXTERNAL_ENTITY_REF

An entity reference in an attribute value referred to an external entity instead of an internal entity.

xml.parsers.expat.errors.XML_ERROR_BAD_CHAR_REF

A character reference referred to a character which is illegal in XML (for example, character `0`, or ‘`&#0;`’).

xml.parsers.expat.errors.XML_ERROR_BINARY_ENTITY_REF

An entity reference referred to an entity which was declared with a notation, so cannot be parsed.

xml.parsers.expat.errors.XML_ERROR_DUPLICATE_ATTRIBUTE

An attribute was used more than once in a start tag.

xml.parsers.expat.errors.XML_ERROR_INCORRECT_ENCODING

xml.parsers.expat.errors.XML_ERROR_INVALID_TOKEN

Raised when an input byte could not properly be assigned to a character; for example, a NUL byte (value `0`) in a UTF-8 input stream.

xml.parsers.expat.errors.XML_ERROR_JUNK_AFTER_DOC_ELEMENT

Something other than whitespace occurred after the document element.

xml.parsers.expat.errors.XML_ERROR_MISPLACED_XML_PI

An XML declaration was found somewhere other than the start of the input data.

xml.parsers.expat.errors.XML_ERROR_NO_ELEMENTS

The document contains no elements (XML requires all documents to contain exactly one top-level element)..

xml.parsers.expat.errors.XML_ERROR_NO_MEMORY

Expat was not able to allocate memory internally.

xml.parsers.expat.errors.XML_ERROR_PARAM_ENTITY_REF

A parameter entity reference was found where it was not allowed.

xml.parsers.expat.errors.XML_ERROR_PARTIAL_CHAR

An incomplete character was found in the input.

xml.parsers.expat.errors.XML_ERROR_RECURSIVE_ENTITY_REF

An entity reference contained another reference to the same entity; possibly via a different name, and possibly indirectly.

xml.parsers.expat.errors.XML_ERROR_SYNTAX

Some unspecified syntax error was encountered.

xml.parsers.expat.errors.XML_ERROR_TAG_MISMATCH

An end tag did not match the innermost open start tag.

xml.parsers.expat.errors.XML_ERROR_UNCLOSED_TOKEN

Some token (such as a start tag) was not closed before the end of the stream or the next token was encountered.

xml.parsers.expat.errors.XML_ERROR_UNDEFINED_ENTITY

A reference was made to an entity which was not defined.

xml.parsers.expat.errors.XML_ERROR_UNKNOWN_ENCODING

The document encoding is not supported by Expat.

xml.parsers.expat.errors.XML_ERROR_UNCLOSED_CDATA_SECTION

A CDATA marked section was not closed.

xml.parsers.expat.errors.XML_ERROR_EXTERNAL_ENTITY_HANDLING

xml.parsers.expat.errors.XML_ERROR_NOT_STANDALONE

The parser determined that the document was not “standalone” though it declared itself to be in the XML declaration, and the **`NotStandaloneHandler`** was set and returned `0`.

xml.parsers.expat.errors.XML_ERROR_UNEXPECTED_STATE

xml.parsers.expat.errors.XML_ERROR_ENTITY_DECLARED_IN_PE

xml.parsers.expat.errors.XML_ERROR_FEATURE_REQUIRES_XML_DTD

An operation was requested that requires DTD support to be compiled in, but Expat was configured without DTD support. This should never be reported by a standard build of the `xml.parsers.expat` module.

xml.parsers.expat.errors.XML_ERROR_CANT_CHANGE_FEATURE_ONCE_PARSING

A behavioral change was requested after parsing started that can only be changed before parsing has started. This is (currently) only raised by **`UseForeignDTD()`**.

xml.parsers.expat.errors.XML_ERROR_UNBOUND_PREFIX

An undeclared prefix was found when namespace processing was enabled.

xml.parsers.expat.errors.XML_ERROR_UNDECLARING_PREFIX

The document attempted to remove the namespace declaration associated with a prefix.

xml.parsers.expat.errors.XML_ERROR_INCOMPLETE_PE

A parameter entity contained incomplete markup.

xml.parsers.expat.errors.XML_ERROR_XML_DECL

The document contained no document element at all.

xml.parsers.expat.errors.XML_ERROR_TEXT_DECL

There was an error parsing a text declaration in an external entity.

xml.parsers.expat.errors.XML_ERROR_PUBLICID

Characters were found in the public id that are not allowed.

xml.parsers.expat.errors.XML_ERROR_SUSPENDED

The requested operation was made on a suspended parser, but isn’t allowed. This includes attempts to provide additional input or to stop the parser.

xml.parsers.expat.errors.XML_ERROR_NOT_SUSPENDED

An attempt to resume the parser was made when the parser had not been suspended.

xml.parsers.expat.errors.XML_ERROR_ABORTED

This should not be reported to Python applications.

xml.parsers.expat.errors.XML_ERROR_FINISHED

The requested operation was made on a parser which was finished parsing input, but isn’t allowed. This includes attempts to provide additional input or to stop the parser.

xml.parsers.expat.errors.XML_ERROR_SUSPEND_PE

xml.parsers.expat.errors.XML_ERROR_RESERVED_PREFIX_XML

An attempt was made to undeclare reserved namespace prefix `xml` or to bind it to another namespace URI.

xml.parsers.expat.errors.XML_ERROR_RESERVED_PREFIX_XMLNS

An attempt was made to declare or undeclare reserved namespace prefix `xmlns`.

xml.parsers.expat.errors.XML_ERROR_RESERVED_NAMESPACE_URI

An attempt was made to bind the URI of one the reserved namespace prefixes `xml` and `xmlns` to another namespace prefix.

xml.parsers.expat.errors.XML_ERROR_INVALID_ARGUMENT

This should not be reported to Python applications.

xml.parsers.expat.errors.XML_ERROR_NO_BUFFER

This should not be reported to Python applications.

xml.parsers.expat.errors.XML_ERROR_AMPLIFICATION_LIMIT_BREACH

The limit on input amplification factor (from DTD and entities) has been breached.

xml.parsers.expat.errors.XML_ERROR_NOT_STARTED

The parser was tried to be stopped or suspended before it started.

Added in version 3.14.

Footnotes

[1]

The encoding string included in XML output should conform to the appropriate standards. For example, “UTF-8” is valid, but “UTF8” is not. See https://www.w3.org/TR/2006/REC-xml11-20060816/#NT-EncodingDecl and https://www.iana.org/assignments/character-sets/character-sets.xhtml.

Internet Protocols and Support

The modules described in this chapter implement internet protocols and support for related technology. They are all implemented in Python. Most of these modules require the presence of the system-dependent module **`socket`**, which is currently supported on most popular platforms. Here is an overview:

- **`webbrowser`** — Convenient web-browser controller
    - Command-line interface
    - Browser controller objects
- **`wsgiref`** — WSGI Utilities and Reference Implementation
    - **`wsgiref.util`** – WSGI environment utilities
    - **`wsgiref.headers`** – WSGI response header tools
    - **`wsgiref.simple_server`** – a simple WSGI HTTP server
    - **`wsgiref.validate`** — WSGI conformance checker
    - **`wsgiref.handlers`** – server/gateway base classes
    - **`wsgiref.types`** – WSGI types for static type checking
    - Examples
- **`urllib`** — URL handling modules
- **`urllib.request`** — Extensible library for opening URLs
    - Request Objects
    - OpenerDirector Objects
    - BaseHandler Objects
    - HTTPRedirectHandler Objects
    - HTTPCookieProcessor Objects
    - ProxyHandler Objects
    - HTTPPasswordMgr Objects
    - HTTPPasswordMgrWithPriorAuth Objects
    - AbstractBasicAuthHandler Objects

  - HTTPBasicAuthHandler Objects
  - ProxyBasicAuthHandler Objects
  - AbstractDigestAuthHandler Objects
  - HTTPDigestAuthHandler Objects
  - ProxyDigestAuthHandler Objects
  - HTTPHandler Objects
  - HTTPSHandler Objects
  - FileHandler Objects
  - DataHandler Objects
  - FTPHandler Objects
  - CacheFTPHandler Objects
  - UnknownHandler Objects
  - HTTPErrorProcessor Objects
  - Examples
  - Legacy interface
  - `urllib.request` Restrictions
- `urllib.response` — Response classes used by urllib
- `urllib.parse` — Parse URLs into components
  - URL Parsing
  - URL parsing security
  - Parsing ASCII Encoded Bytes
  - Structured Parse Results
  - URL Quoting
- `urllib.error` — Exception classes raised by urllib.request
- `urllib.robotparser` — Parser for robots.txt
- `http` — HTTP modules
  - HTTP status codes
  - HTTP status category
  - HTTP methods
- `http.client` — HTTP protocol client
  - HTTPConnection Objects
  - HTTPResponse Objects
  - Examples
  - HTTPMessage Objects
- `ftplib` — FTP protocol client
  - Reference

        - FTP objects
        - FTP_TLS objects
        - Module variables
- `poplib` — POP3 protocol client
    - POP3 Objects
    - POP3 Example
- `imaplib` — IMAP4 protocol client
    - IMAP4 Objects
    - IMAP4 Example
- `smtplib` — SMTP protocol client
    - SMTP Objects
    - SMTP Example
- `uuid` — UUID objects according to **RFC 9562**
    - Command-Line Usage
    - Example
    - Command-Line Example
- `socketserver` — A framework for network servers
    - Server Creation Notes
    - Server Objects
    - Request Handler Objects
    - Examples
        - `socketserver.TCPServer` Example
        - `socketserver.UDPServer` Example
        - Asynchronous Mixins
- `http.server` — HTTP servers
    - Command-line interface
    - Security considerations
- `http.cookies` — HTTP state management
    - Cookie Objects
    - Morsel Objects
    - Example
- `http.cookiejar` — Cookie handling for HTTP clients
    - CookieJar and FileCookieJar Objects
    - FileCookieJar subclasses and co-operation with web browsers
    - CookiePolicy Objects

  - DefaultCookiePolicy Objects
  - Cookie Objects
  - Examples
- `xmlrpc` — XMLRPC server and client modules
- `xmlrpc.client` — XML-RPC client access
  - ServerProxy Objects
  - DateTime Objects
  - Binary Objects
  - Fault Objects
  - ProtocolError Objects
  - MultiCall Objects
  - Convenience Functions
  - Example of Client Usage
  - Example of Client and Server Usage
- `xmlrpc.server` — Basic XML-RPC servers
  - SimpleXMLRPCServer Objects
    - SimpleXMLRPCServer Example
  - CGIXMLRPCRequestHandler
  - Documenting XMLRPC server
  - DocXMLRPCServer Objects
  - DocCGIXMLRPCRequestHandler
- `ipaddress` — IPv4/IPv6 manipulation library
  - Convenience factory functions
  - IP Addresses
    - Address objects
    - Conversion to Strings and Integers
    - Operators
      - Comparison operators
      - Arithmetic operators
  - IP Network definitions
    - Prefix, net mask and host mask
    - Network objects
    - Operators
      - Logical operators
      - Iteration
      - Networks as containers of addresses

- Interface objects
    - Operators
        - Logical operators
- Other Module Level Functions
- Custom Exceptions

`webbrowser` — Convenient web-browser controller

Source code: Lib/webbrowser.py [https://github.com/python/cpython/tree/3.14/Lib/webbrowser.py]

The **`webbrowser`** module provides a high-level interface to allow displaying web-based documents to users. Under most circumstances, simply calling the **`open()`** function from this module will do the right thing.

Under Unix, graphical browsers are preferred under X11, but text-mode browsers will be used if graphical browsers are not available or an X11 display isn't available. If text-mode browsers are used, the calling process will block until the user exits the browser.

If the environment variable **`BROWSER`** exists, it is interpreted as the **`os.pathsep`**-separated list of browsers to try ahead of the platform defaults. When the value of a list part contains the string `%s`, then it is interpreted as a literal browser command line to be used with the argument URL substituted for `%s`; if the value is a single word that refers to one of the already registered browsers this browser is added to the front of the search list; if the part does not contain `%s`, it is simply interpreted as the name of the browser to launch. [[1]]

Changed in version 3.14: The **`BROWSER`** variable can now also be used to reorder the list of platform defaults. This is particularly useful on macOS where the platform defaults do not refer to command-line tools on **`PATH`**.

For non-Unix platforms, or when a remote browser is available on Unix, the controlling process will not wait for the user to finish with the browser, but allow the remote browser to maintain its own windows on the display. If remote browsers are not available on Unix, the controlling process will launch a new browser and wait.

On iOS, the **`BROWSER`** environment variable, as well as any arguments controlling autoraise, browser preference, and new tab/window creation will be ignored. Web pages will *always* be opened in the user's preferred browser, in a new tab, with the browser being brought to the foreground. The use of the `webbrowser` module on iOS requires the `ctypes` module. If `ctypes` isn't available, calls to `open()` will fail.

Command-line interface

The script **webbrowser** can be used as a command-line interface for the module. It accepts a URL as the argument. It accepts the following optional parameters:

-n, --new-window
    Opens the URL in a new browser window, if possible.

-t, --new-tab
    Opens the URL in a new browser tab.

The options are, naturally, mutually exclusive. Usage example:

```
python -m webbrowser -t "https://www.python.org"
```

Availability: not WASI, not Android.

The following exception is defined:

exception webbrowser.Error

Exception raised when a browser control error occurs.

The following functions are defined:

webbrowser.open(*url*, *new=0*, *autoraise=True*)

Display *url* using the default browser. If *new* is 0, the *url* is opened in the same browser window if possible. If *new* is 1, a new browser window is opened if possible. If *new* is 2, a new browser page ("tab") is opened if possible. If *autoraise* is `True`, the window is raised if possible (note that under many window managers this will occur regardless of the setting of this variable).

Returns `True` if a browser was successfully launched, `False` otherwise.

Note that on some platforms, trying to open a filename using this function, may work and start the operating system's associated program. However, this is neither supported nor portable.

Raises an auditing event `webbrowser.open` with argument `url`.

webbrowser.open_new(*url*)

Open *url* in a new window of the default browser, if possible, otherwise, open *url* in the only browser window.

Returns `True` if a browser was successfully launched, `False` otherwise.

webbrowser.open_new_tab(*url*)

Open *url* in a new page ("tab") of the default browser, if possible, otherwise equivalent to **`open_new()`**.

Returns `True` if a browser was successfully launched, `False` otherwise.

webbrowser.get(*using=None*)

Return a controller object for the browser type *using*. If *using* is `None`, return a controller for a default browser appropriate to the caller’s environment.

webbrowser.register(*name*, *constructor*, *instance=None*, *, *preferred=False*)

Register the browser type *name*. Once a browser type is registered, the **`get()`** function can return a controller for that browser type. If *instance* is not provided, or is `None`, *constructor* will be called without parameters to create an instance when needed. If *instance* is provided, *constructor* will never be called, and may be `None`.

Setting *preferred* to `True` makes this browser a preferred result for a **`get()`** call with no argument. Otherwise, this entry point is only useful if you plan to either set the **`BROWSER`** variable or call **`get()`** with a nonempty argument matching the name of a handler you declare.

Changed in version 3.7: preferred keyword-only parameter was added.

A number of browser types are predefined. This table gives the type names that may be passed to the **`get()`** function and the corresponding instantiations for the controller classes, all defined in this module.

Type Name	Class Name	Notes
`'mozilla'`	`Mozilla('mozilla')`	

Type Name	Class Name	Notes
`'firefox'`	`Mozilla('mozilla')`	
`'epiphany'`	`Epiphany('epiphany')`	
`'kfmclient'`	`Konqueror()`	(1)
`'konqueror'`	`Konqueror()`	(1)
`'kfm'`	`Konqueror()`	(1)
`'opera'`	`Opera()`	
`'links'`	`GenericBrowser('links')`	
`'elinks'`	`Elinks('elinks')`	
`'lynx'`	`GenericBrowser('lynx')`	
`'w3m'`	`GenericBrowser('w3m')`	
`'windows-default'`	`WindowsDefault`	(2)
`'macosx'`	`MacOSXOSAScript('default')`	(3)

Type Name	Class Name	Notes
`'safari'`	`MacOSXOSAScript('safari')`	(3)
`'google-chrome'`	`Chrome('google-chrome')`	
`'chrome'`	`Chrome('chrome')`	
`'chromium'`	`Chromium('chromium')`	
`'chromium-browser'`	`Chromium('chromium-browser')`	
`'iosbrowser'`	`IOSBrowser`	(4)

Notes:

1. “Konqueror” is the file manager for the KDE desktop environment for Unix, and only makes sense to use if KDE is running. Some way of reliably detecting KDE would be nice; the **`KDEDIR`** variable is not sufficient. Note also that the name “kfm” is used even when using the **konqueror** command with KDE 2 — the implementation selects the best strategy for running Konqueror.
2. Only on Windows platforms.
3. Only on macOS.
4. Only on iOS.

Added in version 3.2: A new **`MacOSXOSAScript`** class has been added and is used on Mac instead of the previous **`MacOSX`** class. This adds support for opening browsers not currently set as the OS default.

Added in version 3.3: Support for Chrome/Chromium has been added.

Changed in version 3.12: Support for several obsolete browsers has been removed. Removed browsers include Grail, Mosaic, Netscape, Galeon, Skipstone, Iceape, and Firefox versions 35 and below.

Changed in version 3.13: Support for iOS has been added.

Here are some simple examples:

```
url = 'https://docs.python.org/'

# Open URL in a new tab, if a browser window is already open.
webbrowser.open_new_tab(url)

# Open URL in new window, raising the window if possible.
webbrowser.open_new(url)
```

Browser controller objects

Browser controllers provide the `name` attribute, and the following three methods which parallel module-level convenience functions:

controller.name

System-dependent name for the browser.

controller.open(*url*, *new=0*, *autoraise=True*)

Display *url* using the browser handled by this controller. If *new* is 1, a new browser window is opened if possible. If *new* is 2, a new browser page (“tab”) is opened if possible.

controller.open_new(*url*)

Open *url* in a new window of the browser handled by this controller, if possible, otherwise, open *url* in the only browser window. Alias `open_new()`.

controller.open_new_tab(*url*)

Open *url* in a new page (“tab”) of the browser handled by this controller, if possible, otherwise equivalent to `open_new()`.

Footnotes

[1]

Executables named here without a full path will be searched in the directories given in the **`PATH`** environment variable.

`wsgiref` — WSGI Utilities and Reference Implementation

Source code: Lib/wsgiref [https://github.com/python/cpython/tree/3.14/Lib/wsgiref]

The Web Server Gateway Interface (WSGI) is a standard interface between web server software and web applications written in Python. Having a standard interface makes it easy to use an application that supports WSGI with a number of different web servers.

Only authors of web servers and programming frameworks need to know every detail and corner case of the WSGI design. You don’t need to understand every detail of WSGI just to install a WSGI application or to write a web application using an existing framework.

`wsgiref` is a reference implementation of the WSGI specification that can be used to add WSGI support to a web server or framework. It provides utilities for manipulating WSGI environment variables and response headers, base classes for implementing WSGI servers, a demo HTTP server that serves WSGI applications, types for static type checking, and a validation tool that checks WSGI servers and applications for conformance to the WSGI specification (**PEP 3333** [https://peps.python.org/pep-3333/]).

See wsgi.readthedocs.io [https://wsgi.readthedocs.io/] for more information about WSGI, and links to tutorials and other resources.

`wsgiref.util` – WSGI environment utilities

This module provides a variety of utility functions for working with WSGI environments. A WSGI environment is a dictionary containing HTTP request variables as described in **PEP 3333** [https://peps.python.org/pep-3333/]. All of the functions taking an *environ* parameter expect a WSGI-compliant dictionary to be supplied; please see **PEP 3333** [https://peps.python.org/pep-3333/] for a detailed specification and `WSGIEnvironment` for a type alias that can be used in type annotations.

wsgiref.util.guess_scheme(*environ*)

Return a guess for whether `wsgi.url_scheme` should be "http" or "https", by checking for a `HTTPS` environment variable in the *environ* dictionary. The return value is a string.

This function is useful when creating a gateway that wraps CGI or a CGI-like protocol such as FastCGI. Typically, servers providing such protocols will include a `HTTPS` variable with a value of "1", "yes", or "on" when a request is received via SSL. So, this function returns "https" if such a value is found, and "http" otherwise.

wsgiref.util.request_uri(*environ*, *include_query=True*)

Return the full request URI, optionally including the query string, using the algorithm found in the "URL Reconstruction" section of **PEP 3333** [https://peps.python.org/pep-3333/]. If *include_query* is false, the query string is not included in the resulting URI.

wsgiref.util.application_uri(*environ*)

Similar to **`request_uri()`**, except that the `PATH_INFO` and `QUERY_STRING` variables are ignored. The result is the base URI of the application object addressed by the request.

wsgiref.util.shift_path_info(*environ*)

Shift a single name from `PATH_INFO` to `SCRIPT_NAME` and return the name. The *environ* dictionary is *modified* in-place; use a copy if you need to keep the original `PATH_INFO` or `SCRIPT_NAME` intact.

If there are no remaining path segments in `PATH_INFO`, `None` is returned.

Typically, this routine is used to process each portion of a request URI path, for example to treat the path as a series of dictionary keys. This routine modifies the passed-in environment to make it suitable for invoking another WSGI application that is located at the target URI. For example, if there is a WSGI application at `/foo`, and the request URI path is `/foo/bar/baz`, and the WSGI application at `/foo` calls **`shift_path_info()`**, it will receive the string “bar”, and the environment will be updated to be suitable for passing to a WSGI application at `/foo/bar`. That is, `SCRIPT_NAME` will change from `/foo` to `/foo/bar`, and `PATH_INFO` will change from `/bar/baz` to `/baz`.

When `PATH_INFO` is just a “/”, this routine returns an empty string and appends a trailing slash to `SCRIPT_NAME`, even though empty path segments are normally ignored, and `SCRIPT_NAME` doesn’t normally end in a slash. This is intentional behavior, to ensure that an application can tell the difference between URIs ending in `/x` from ones ending in `/x/` when using this routine to do object traversal.

wsgiref.util.setup_testing_defaults(*environ*)

Update *environ* with trivial defaults for testing purposes.

This routine adds various parameters required for WSGI, including `HTTP_HOST`, `SERVER_NAME`, `SERVER_PORT`, `REQUEST_METHOD`, `SCRIPT_NAME`, `PATH_INFO`, and all of the **PEP 3333** [https://peps.python.org/pep-3333/]-defined `wsgi.*` variables. It only supplies default values, and does not replace any existing settings for these variables.

This routine is intended to make it easier for unit tests of WSGI servers and applications to set up dummy environments. It should NOT be used by actual WSGI servers or applications, since the data is fake!

Example usage (see also **`demo_app()`** for another example):

```
from wsgiref.util import setup_testing_defaults
from wsgiref.simple_server import make_server

# A relatively simple WSGI application. It's going to print
out the
# environment dictionary after being updated by
setup_testing_defaults
def simple_app(environ, start_response):
    setup_testing_defaults(environ)

    status = '200 OK'
    headers = [('Content-type', 'text/plain; charset=utf-
8')]

    start_response(status, headers)

    ret = [("%s: %s\n" % (key, value)).encode("utf-8")
           for key, value in environ.items()]
    return ret

with make_server('', 8000, simple_app) as httpd:
    print("Serving on port 8000...")
    httpd.serve_forever()
```

In addition to the environment functions above, the **`wsgiref.util`** module also provides these miscellaneous utilities:

wsgiref.util.is_hop_by_hop(*header_name*)

Return `True` if 'header_name' is an HTTP/1.1 "Hop-by-Hop" header, as defined by **RFC 2616** [https://datatracker.ietf.org/doc/html/rfc2616.html].

class wsgiref.util.FileWrapper(*filelike*, *blksize=8192*)

A concrete implementation of the **`wsgiref.types.FileWrapper`** protocol used to convert a file-like object to an iterator. The resulting objects are iterables. As the object is iterated over, the optional *blksize* parameter will be repeatedly passed to the *filelike* object's **`read()`** method to obtain bytestrings to yield. When **`read()`** returns an empty bytestring, iteration is ended and is not resumable.

If *filelike* has a **`close()`** method, the returned object will also have a **`close()`** method, and it will invoke the *filelike* object's **`close()`** method when called.

Example usage:

```
from io import StringIO
from wsgiref.util import FileWrapper

# We're using a StringIO-buffer for as the file-like object
filelike = StringIO("This is an example file-like
object"*10)
wrapper = FileWrapper(filelike, blksize=5)

for chunk in wrapper:
    print(chunk)
```

Changed in version 3.11: Support for **`__getitem__()`** method has been removed.

`wsgiref.headers` – WSGI response header tools

This module provides a single class, `Headers`, for convenient manipulation of WSGI response headers using a mapping-like interface.

class wsgiref.headers.Headers([*headers*])

Create a mapping-like object wrapping *headers*, which must be a list of header name/value tuples as described in **PEP 3333** [https://peps.python.org/pep-3333/]. The default value of *headers* is an empty list.

`Headers` objects support typical mapping operations including `__getitem__()`, `get()`, `__setitem__()`, `setdefault()`, `__delitem__()` and `__contains__()`. For each of these methods, the key is the header name (treated case-insensitively), and the value is the first value associated with that header name. Setting a header deletes any existing values for that header, then adds a new value at the end of the wrapped header list. Headers' existing order is generally maintained, with new headers added to the end of the wrapped list.

Unlike a dictionary, `Headers` objects do not raise an error when you try to get or delete a key that isn't in the wrapped header list. Getting a nonexistent header just returns `None`, and deleting a nonexistent header does nothing.

`Headers` objects also support **`keys()`**, **`values()`**, and **`items()`** methods. The lists returned by **`keys()`** and **`items()`** can include the same key more than once if there is a multi-valued header. The `len()` of a `Headers` object is the same as the length of its **`items()`**, which is the same as the length of the wrapped header list. In fact, the **`items()`** method just returns a copy of the wrapped header list.

Calling `bytes()` on a **`Headers`** object returns a formatted bytestring suitable for transmission as HTTP response headers. Each header is placed on a line with its value, separated by a colon and a space. Each line is terminated by a carriage return and line feed, and the bytestring is terminated with a blank line.

In addition to their mapping interface and formatting features, **`Headers`** objects also have the following methods for querying and adding multi-valued headers, and for adding headers with MIME parameters:

get_all(*name*)

> Return a list of all the values for the named header.
>
> The returned list will be sorted in the order they appeared in the original header list or were added to this instance, and may contain duplicates. Any fields deleted and re-inserted are always appended to the header list. If no fields exist with the given name, returns an empty list.

add_header(*name*, *value*, ***_params*)

> Add a (possibly multi-valued) header, with optional MIME parameters specified via keyword arguments.
>
> *name* is the header field to add. Keyword arguments can be used to set MIME parameters for the header field. Each parameter must be a string or `None`. Underscores in parameter names are converted to dashes, since dashes are illegal in Python identifiers, but many MIME parameter names include dashes. If the parameter value is a string, it is added to the

header value parameters in the form `name="value"`. If it is `None`, only the parameter name is added. (This is used for MIME parameters without a value.) Example usage:

```
h.add_header('content-disposition', 'attachment',
filename='bud.gif')
```

The above will add a header that looks like this:

```
Content-Disposition: attachment; filename="bud.gif"
```

Changed in version 3.5: headers parameter is optional.

`wsgiref.simple_server` – a simple WSGI HTTP server

This module implements a simple HTTP server (based on `http.server`) that serves WSGI applications. Each server instance serves a single WSGI application on a given host and port. If you want to serve multiple applications on a single host and port, you should create a WSGI application that parses `PATH_INFO` to select which application to invoke for each request. (E.g., using the **`shift_path_info()`** function from `wsgiref.util`.)

wsgiref.simple_server.make_server(*host*, *port*, *app*, *server_class=WSGIServer*, *handler_class=WSGIRequestHandler*)

Create a new WSGI server listening on *host* and *port*, accepting connections for *app*. The return value is an instance of the supplied *server_class*, and will process requests using the specified *handler_class*. *app* must be a WSGI application object, as defined by **PEP 3333** [https://peps.python.org/pep-3333/].

Example usage:

```
from wsgiref.simple_server import make_server, demo_app

with make_server('', 8000, demo_app) as httpd:
    print("Serving HTTP on port 8000...")

    # Respond to requests until process is killed
    httpd.serve_forever()

    # Alternative: serve one request, then exit
    httpd.handle_request()
```

wsgiref.simple_server.demo_app(*environ*, *start_response*)

This function is a small but complete WSGI application that returns a text page containing the message "Hello world!" and a list of the key/value pairs provided in the *environ* parameter. It's useful for verifying that a WSGI server (such as `wsgiref.simple_server`) is able to run a simple WSGI application correctly.

The *start_response* callable should follow the `StartResponse` protocol.

class wsgiref.simple_server.WSGIServer(*server_address*, *RequestHandlerClass*)

Create a `WSGIServer` instance. *server_address* should be a `(host,port)` tuple, and *RequestHandlerClass* should be the subclass of `http.server.BaseHTTPRequestHandler` that will be used to process requests.

You do not normally need to call this constructor, as the `make_server()` function can handle all the details for you.

`WSGIServer` is a subclass of `http.server.HTTPServer`, so all of its methods (such as **`serve_forever()`** and **`handle_request()`**) are available. `WSGIServer` also provides these WSGI-specific methods:

set_app(*application*)

Sets the callable *application* as the WSGI application that will receive requests.

get_app()

Returns the currently set application callable.

Normally, however, you do not need to use these additional methods, as `set_app()` is normally called by `make_server()`, and the `get_app()` exists mainly for the benefit of request handler instances.

class wsgiref.simple_server.WSGIRequestHandler(*request*, *client_address*, *server*)

Create an HTTP handler for the given *request* (i.e. a socket), *client_address* (a `(host,port)` tuple), and *server* (`WSGIServer` instance).

You do not need to create instances of this class directly; they are automatically created as needed by `WSGIServer` objects. You can, however, subclass this class and supply it as a *handler_class* to the `make_server()` function. Some possibly relevant methods for overriding in subclasses:

get_environ()

Return a `WSGIEnvironment` dictionary for a request. The default implementation copies the contents of the `WSGIServer` object's **`base_environ`** dictionary attribute and then adds various headers derived from the HTTP request. Each call to this method should return a new dictionary containing all of the relevant CGI environment variables as specified in **PEP 3333** [https://peps.python.org/pep-3333/].

get_stderr()

Return the object that should be used as the `wsgi.errors` stream. The default implementation just returns `sys.stderr`.

handle()

Process the HTTP request. The default implementation creates a handler instance using a **`wsgiref.handlers`** class to implement the actual WSGI application interface.

`wsgiref.validate` — WSGI conformance checker

When creating new WSGI application objects, frameworks, servers, or middleware, it can be useful to validate the new code’s conformance using **`wsgiref.validate`**. This module provides a function that creates WSGI application objects that validate communications between a WSGI server or gateway and a WSGI application object, to check both sides for protocol conformance.

Note that this utility does not guarantee complete **PEP 3333** [https://peps.python.org/pep-3333/] compliance; an absence of errors from this module does not necessarily mean that errors do not exist. However, if this module does produce an error, then it is virtually certain that either the server or application is not 100% compliant.

This module is based on the **`paste.lint`** module from Ian Bicking’s “Python Paste” library.

wsgiref.validate.validator(*application*)

Wrap *application* and return a new WSGI application object. The returned application will forward all requests to the original *application*, and will check that both the *application* and the server invoking it are conforming to the WSGI specification and to **RFC 2616** [https://datatracker.ietf.org/doc/html/rfc2616.html].

Any detected nonconformance results in an `AssertionError` being raised; note, however, that how these errors are handled is server-dependent. For example, `wsgiref.simple_server` and other servers based on `wsgiref.handlers` (that don’t override the error handling methods to do something else) will simply output a message that an error has occurred, and dump the traceback to `sys.stderr` or some other error stream.

This wrapper may also generate output using the `warnings` module to indicate behaviors that are questionable but which may not actually be prohibited by **PEP 3333** [https://peps.python.org/pep-3333/]. Unless they are suppressed using Python command-line options or the `warnings` API, any such warnings will be written to `sys.stderr` (*not* `wsgi.errors`, unless they happen to be the same object).

Example usage:

```
from wsgiref.validate import validator
from wsgiref.simple_server import make_server

# Our callable object which is intentionally not compliant
to the
# standard, so the validator is going to break
def simple_app(environ, start_response):
    status = '200 OK'  # HTTP Status
    headers = [('Content-type', 'text/plain')]  # HTTP
Headers
    start_response(status, headers)

    # This is going to break because we need to return a
list, and
    # the validator is going to inform us
    return b"Hello World"
```

```
# This is the application wrapped in a validator
validator_app = validator(simple_app)

with make_server('', 8000, validator_app) as httpd:
    print("Listening on port 8000....")
    httpd.serve_forever()
```

`wsgiref.handlers` – server/gateway base classes

This module provides base handler classes for implementing WSGI servers and gateways. These base classes handle most of the work of communicating with a WSGI application, as long as they are given a CGI-like environment, along with input, output, and error streams.

class wsgiref.handlers.CGIHandler

CGI-based invocation via `sys.stdin`, `sys.stdout`, `sys.stderr` and `os.environ`. This is useful when you have a WSGI application and want to run it as a CGI script. Simply invoke `CGIHandler().run(app)`, where `app` is the WSGI application object you wish to invoke.

This class is a subclass of **`BaseCGIHandler`** that sets `wsgi.run_once` to true, `wsgi.multithread` to false, and `wsgi.multiprocess` to true, and always uses **`sys`** and **`os`** to obtain the necessary CGI streams and environment.

class wsgiref.handlers.IISCGIHandler

A specialized alternative to **`CGIHandler`**, for use when deploying on Microsoft’s IIS web server, without having set the config allowPathInfo option (IIS>=7) or metabase allowPathInfoForScriptMappings (IIS<7).

By default, IIS gives a `PATH_INFO` that duplicates the `SCRIPT_NAME` at the front, causing problems for WSGI applications that wish to implement routing. This handler strips any such duplicated path.

IIS can be configured to pass the correct `PATH_INFO`, but this causes another bug where `PATH_TRANSLATED` is wrong. Luckily this variable is rarely used and is not guaranteed by WSGI. On IIS<7, though, the setting can only be made on a vhost level, affecting all other script mappings, many of which break when exposed to the `PATH_TRANSLATED` bug. For this reason IIS<7 is almost never deployed with the fix (Even IIS7 rarely uses it because there is still no UI for it.).

There is no way for CGI code to tell whether the option was set, so a separate handler class is provided. It is used in the same way as **`CGIHandler`**, i.e., by calling `IISCGIHandler().run(app)`, where `app` is the WSGI application object you wish to invoke.

Added in version 3.2.

class wsgiref.handlers.BaseCGIHandler(*stdin*, *stdout*, *stderr*, *environ*, *multithread=True*, *multiprocess=False*)

Similar to **`CGIHandler`**, but instead of using the **`sys`** and **`os`** modules, the CGI environment and I/O streams are specified explicitly. The *multithread* and *multiprocess* values are used to set the `wsgi.multithread` and `wsgi.multiprocess` flags for any applications run by the handler instance.

This class is a subclass of **`SimpleHandler`** intended for use with software other than HTTP “origin servers”. If you are writing a gateway protocol implementation (such as CGI, FastCGI, SCGI, etc.) that uses a `Status:` header to send an

HTTP status, you probably want to subclass this instead of `SimpleHandler`.

class wsgiref.handlers.SimpleHandler(*stdin*, *stdout*, *stderr*, *environ*, *multithread=True*, *multiprocess=False*)

Similar to `BaseCGIHandler`, but designed for use with HTTP origin servers. If you are writing an HTTP server implementation, you will probably want to subclass this instead of `BaseCGIHandler`.

This class is a subclass of `BaseHandler`. It overrides the **`__init__()`**, `get_stdin()`, `get_stderr()`, `add_cgi_vars()`, `_write()`, and `_flush()` methods to support explicitly setting the environment and streams via the constructor. The supplied environment and streams are stored in the **`stdin`**, **`stdout`**, **`stderr`**, and **`environ`** attributes.

The `write()` method of *stdout* should write each chunk in full, like `io.BufferedIOBase`.

class wsgiref.handlers.BaseHandler

This is an abstract base class for running WSGI applications. Each instance will handle a single HTTP request, although in principle you could create a subclass that was reusable for multiple requests.

`BaseHandler` instances have only one method intended for external use:

run(*app*)

Run the specified WSGI application, *app*.

All of the other `BaseHandler` methods are invoked by this method in the process of running the application, and thus exist primarily to allow customizing the process.

The following methods MUST be overridden in a subclass:

_write(*data*)

Buffer the bytes *data* for transmission to the client. It’s okay if this method actually transmits the data; `BaseHandler` just separates write and flush operations for greater efficiency when the underlying system actually has such a distinction.

_flush()

Force buffered data to be transmitted to the client. It’s okay if this method is a no-op (i.e., if `_write()` actually sends the data).

get_stdin()

Return an object compatible with `InputStream` suitable for use as the `wsgi.input` of the request currently being processed.

get_stderr()

Return an object compatible with `ErrorStream` suitable for use as the `wsgi.errors` of the request currently being processed.

add_cgi_vars()

Insert CGI variables for the current request into the **`environ`** attribute.

Here are some other methods and attributes you may wish to override. This list is only a summary, however, and does not include every method that can be overridden. You should consult the docstrings and source code for additional information before attempting to create a customized `BaseHandler` subclass.

Attributes and methods for customizing the WSGI environment:

wsgi_multithread

The value to be used for the `wsgi.multithread` environment variable. It defaults to true in **`BaseHandler`**, but may have a different default (or be set by the constructor) in the other subclasses.

wsgi_multiprocess

The value to be used for the `wsgi.multiprocess` environment variable. It defaults to true in **`BaseHandler`**, but may have a different default (or be set by the constructor) in the other subclasses.

wsgi_run_once

The value to be used for the `wsgi.run_once` environment variable. It defaults to false in **`BaseHandler`**, but **`CGIHandler`** sets it to true by default.

os_environ

The default environment variables to be included in every request’s WSGI environment. By default, this is a copy of `os.environ` at the time that **`wsgiref.handlers`** was imported, but subclasses can either create their own at the class or instance level. Note that the dictionary should be considered read-only, since the default value is shared between multiple classes and instances.

server_software

If the **`origin_server`** attribute is set, this attribute’s value is used to set the default `SERVER_SOFTWARE` WSGI environment variable, and also to set a default `Server:` header in HTTP responses. It is ignored for handlers (such as `BaseCGIHandler` and `CGIHandler`) that are not HTTP origin servers.

Changed in version 3.3: The term “Python” is replaced with implementation specific term like “CPython”, “Jython” etc.

get_scheme()

Return the URL scheme being used for the current request. The default implementation uses the **`guess_scheme()`** function from `wsgiref.util` to guess whether the scheme should be “http” or “https”, based on the current request’s **`environ`** variables.

setup_environ()

Set the **`environ`** attribute to a fully populated WSGI environment. The default implementation uses all of the above methods and attributes, plus the `get_stdin()`, `get_stderr()`, and `add_cgi_vars()` methods and the `wsgi_file_wrapper` attribute. It also inserts a `SERVER_SOFTWARE` key if not present, as long as the `origin_server` attribute is a true value and the `server_software` attribute is set.

Methods and attributes for customizing exception handling:

log_exception(*exc_info*)

Log the *exc_info* tuple in the server log. *exc_info* is a `(type, value, traceback)` tuple. The default implementation simply writes the traceback to the request’s `wsgi.errors` stream and flushes it. Subclasses can override this method to change the format or retarget the output, mail the traceback to an administrator, or whatever other action may be deemed suitable.

traceback_limit

The maximum number of frames to include in tracebacks output by the default `log_exception()` method. If `None`, all frames are included.

error_output(*environ*, *start_response*)

This method is a WSGI application to generate an error page for the user. It is only invoked if an error occurs before headers are sent to the client.

This method can access the current error using `sys.exception()`, and should pass that information to *start_response* when calling it (as described in the “Error Handling” section of **PEP 3333** [https://peps.python.org/pep-3333/]). In particular, the *start_response* callable should follow the `StartResponse` protocol.

The default implementation just uses the `error_status`, `error_headers`, and `error_body` attributes to generate an output page. Subclasses can override this to produce more dynamic error output.

Note, however, that it’s not recommended from a security perspective to spit out diagnostics to any old user; ideally, you should have to do something special to enable diagnostic output, which is why the default implementation doesn’t include any.

error_status

The HTTP status used for error responses. This should be a status string as defined in **PEP 3333** [https://peps.python.org/pep-3333/]; it defaults to a 500 code and message.

error_headers

The HTTP headers used for error responses. This should be a list of WSGI response headers (`(name, value)` tuples), as described in **PEP 3333** [https://peps.python.org/pep-3333/]. The default list just sets the content type to `text/plain`.

error_body

The error response body. This should be an HTTP response body bytestring. It defaults to the plain text, “A server error occurred. Please contact the administrator.”

Methods and attributes for **PEP 3333** [https://peps.python.org/pep-3333/]’s “Optional Platform-Specific File Handling” feature:

wsgi_file_wrapper

A `wsgi.file_wrapper` factory, compatible with `wsgiref.types.FileWrapper`, or `None`. The default value of this attribute is the `wsgiref.util.FileWrapper` class.

sendfile()

Override to implement platform-specific file transmission. This method is called only if the application’s return value is an instance of the class specified by the `wsgi_file_wrapper` attribute. It should return a true value if it was able to successfully transmit the file, so that the default transmission code will not be executed. The default implementation of this method just returns a false value.

Miscellaneous methods and attributes:

origin_server

This attribute should be set to a true value if the handler’s `_write()` and `_flush()` are being used to communicate directly to the client, rather than via a CGI-like gateway protocol that wants the HTTP status in a special `Status:` header.

This attribute's default value is true in `BaseHandler`, but false in `BaseCGIHandler` and `CGIHandler`.

http_version

If `origin_server` is true, this string attribute is used to set the HTTP version of the response set to the client. It defaults to `"1.0"`.

wsgiref.handlers.read_environ()

Transcode CGI variables from `os.environ` to **PEP 3333** [https://peps.python.org/pep-3333/] "bytes in unicode" strings, returning a new dictionary. This function is used by `CGIHandler` and `IISCGIHandler` in place of directly using `os.environ`, which is not necessarily WSGI-compliant on all platforms and web servers using Python 3 – specifically, ones where the OS's actual environment is Unicode (i.e. Windows), or ones where the environment is bytes, but the system encoding used by Python to decode it is anything other than ISO-8859-1 (e.g. Unix systems using UTF-8).

If you are implementing a CGI-based handler of your own, you probably want to use this routine instead of just copying values out of `os.environ` directly.

Added in version 3.2.

`wsgiref.types` – WSGI types for static type checking

This module provides various types for static type checking as described in **PEP 3333** [https://peps.python.org/pep-3333/].

Added in version 3.11.

class wsgiref.types.StartResponse

A `typing.Protocol` describing **start_response()** [https://peps.python.org/pep-3333/#the-start-response-callable] callables (**PEP 3333** [https://peps.python.org/pep-3333/]).

wsgiref.types.WSGIEnvironment

A type alias describing a WSGI environment dictionary.

wsgiref.types.WSGIApplication

A type alias describing a WSGI application callable.

class wsgiref.types.InputStream

A `typing.Protocol` describing a **WSGI Input Stream** [https://peps.python.org/pep-3333/#input-and-error-streams].

class wsgiref.types.ErrorStream

A `typing.Protocol` describing a **WSGI Error Stream** [https://peps.python.org/pep-3333/#input-and-error-streams].

class wsgiref.types.FileWrapper

A `typing.Protocol` describing a **file wrapper** [https://peps.python.org/pep-3333/#optional-platform-specific-file-handling]. See `wsgiref.util.FileWrapper` for a concrete implementation of this protocol.

Examples

This is a working “Hello World” WSGI application, where the *start_response* callable should follow the `StartResponse` protocol:

```
"""
Every WSGI application must have an application object - a
callable
object that accepts two arguments. For that purpose, we're
going to
use a function (note that you're not limited to a function, you
can
use a class for example). The first argument passed to the
function
is a dictionary containing CGI-style environment variables and
the
second variable is the callable object.
"""
from wsgiref.simple_server import make_server


def hello_world_app(environ, start_response):
    status = "200 OK"  # HTTP Status
    headers = [("Content-type", "text/plain; charset=utf-8")]
# HTTP Headers
    start_response(status, headers)

    # The returned object is going to be printed
    return [b"Hello World"]

with make_server("", 8000, hello_world_app) as httpd:
    print("Serving on port 8000...")

    # Serve until process is killed
    httpd.serve_forever()
```

Example of a WSGI application serving the current directory, accept optional directory and port number (default: 8000) on the command line:

```
"""
Small wsgiref based web server. Takes a path to serve from and
an
optional port number (defaults to 8000), then tries to serve
files.
MIME types are guessed from the file names, 404 errors are
raised
```

```
if the file is not found.
"""
import mimetypes
import os
import sys
from wsgiref import simple_server, util


def app(environ, respond):
    # Get the file name and MIME type
    fn = os.path.join(path, environ["PATH_INFO"][1:])
    if "." not in fn.split(os.path.sep)[-1]:
        fn = os.path.join(fn, "index.html")
    mime_type = mimetypes.guess_file_type(fn)[0]

    # Return 200 OK if file exists, otherwise 404 Not Found
    if os.path.exists(fn):
        respond("200 OK", [("Content-Type", mime_type)])
        return util.FileWrapper(open(fn, "rb"))
    else:
        respond("404 Not Found", [("Content-Type",
"text/plain")])
        return [b"not found"]


if __name__ == "__main__":
    # Get the path and port from command-line arguments
    path = sys.argv[1] if len(sys.argv) > 1 else os.getcwd()
    port = int(sys.argv[2]) if len(sys.argv) > 2 else 8000

    # Make and start the server until control-c
    httpd = simple_server.make_server("", port, app)
    print(f"Serving {path} on port {port}, control-C to stop")
    try:
        httpd.serve_forever()
    except KeyboardInterrupt:
        print("Shutting down.")
        httpd.server_close()
```

`urllib` — URL handling modules

Source code: Lib/urllib/
[https://github.com/python/cpython/tree/3.14/Lib/urllib/]

`urllib` is a package that collects several modules for working with URLs:

- **`urllib.request`** for opening and reading URLs
- **`urllib.error`** containing the exceptions raised by **`urllib.request`**
- **`urllib.parse`** for parsing URLs
- **`urllib.robotparser`** for parsing `robots.txt` files

`urllib.request` — Extensible library for opening URLs

Source code: Lib/urllib/request.py [https://github.com/python/cpython/tree/3.14/Lib/urllib/request.py]

The `urllib.request` module defines functions and classes which help in opening URLs (mostly HTTP) in a complex world — basic and digest authentication, redirections, cookies and more.

See also

The Requests package [https://requests.readthedocs.io/en/master/] is recommended for a higher-level HTTP client interface.

Warning

On macOS it is unsafe to use this module in programs using `os.fork()` because the `getproxies()` implementation for macOS uses a higher-level system API. Set the environment variable `no_proxy` to `*` to avoid this problem (e.g. `os.environ["no_proxy"] = "*"`).

Availability: not WASI.

This module does not work or is not available on WebAssembly. See WebAssembly platforms for more information.

The `urllib.request` module defines the following functions:

urllib.request.urlopen(*url*, *data=None*, [*timeout*,]*, *context=None*)

Open *url*, which can be either a string containing a valid, properly encoded URL, or a `Request` object.

data must be an object specifying additional data to be sent to the server, or `None` if no such data is needed. See `Request` for details.

urllib.request module uses HTTP/1.1 and includes `Connection:close` header in its HTTP requests.

The optional *timeout* parameter specifies a timeout in seconds for blocking operations like the connection attempt (if not specified, the global default timeout setting will be used). This actually only works for HTTP, HTTPS and FTP connections.

If *context* is specified, it must be a `ssl.SSLContext` instance describing the various SSL options. See `HTTPSConnection` for more details.

This function always returns an object which can work as a context manager and has the properties *url*, *headers*, and *status*. See `urllib.response.addinfourl` for more detail on these properties.

For HTTP and HTTPS URLs, this function returns a `http.client.HTTPResponse` object slightly modified. In addition to the three new methods above, the msg attribute contains the same information as the `reason` attribute — the reason phrase returned by server — instead of the response headers as it is specified in the documentation for `HTTPResponse`.

For FTP, file, and data URLs, this function returns a `urllib.response.addinfourl` object.

Raises `URLError` on protocol errors.

Note that `None` may be returned if no handler handles the request (though the default installed global `OpenerDirector` uses `UnknownHandler` to ensure this never happens).

In addition, if proxy settings are detected (for example, when a `*_proxy` environment variable like **`http_proxy`** is set), `ProxyHandler` is default installed and makes sure the requests are handled through the proxy.

The legacy `urllib.urlopen` function from Python 2.6 and earlier has been discontinued; `urllib.request.urlopen()` corresponds to the old `urllib2.urlopen`. Proxy handling, which was done by passing a dictionary parameter to `urllib.urlopen`, can be obtained by using `ProxyHandler` objects.

The default opener raises an auditing event `urllib.Request` with arguments `fullurl`, `data`, `headers`, `method` taken from the request object.

Changed in version 3.2: cafile and *capath* were added.

HTTPS virtual hosts are now supported if possible (that is, if `ssl.HAS_SNI` is true).

data can be an iterable object.

Changed in version 3.3: cadefault was added.

Changed in version 3.4.3: context was added.

Changed in version 3.10: HTTPS connection now send an ALPN extension with protocol indicator `http/1.1` when no *context* is given. Custom *context* should set ALPN protocols with `set_alpn_protocols()`.

Changed in version 3.13: Remove *cafile*, *capath* and *cadefault* parameters: use the *context* parameter instead.

urllib.request.install_opener(*opener*)

Install an `OpenerDirector` instance as the default global opener. Installing an opener is only necessary if you want urlopen to use that opener; otherwise, simply call `OpenerDirector.open()` instead of `urlopen()`. The code does not check for a real `OpenerDirector`, and any class with the appropriate interface will work.

urllib.request.build_opener([*handler*, ...])

Return an `OpenerDirector` instance, which chains the handlers in the order given. *handler*s can be either instances of `BaseHandler`, or subclasses of `BaseHandler` (in which case it must be possible to call the constructor without any parameters). Instances of the following classes will be in front of the *handler*s, unless the *handler*s contain them, instances of them or subclasses of them: `ProxyHandler` (if proxy settings are detected), `UnknownHandler`, `HTTPHandler`, `HTTPDefaultErrorHandler`, `HTTPRedirectHandler`, `FTPHandler`, `FileHandler`, `HTTPErrorProcessor`.

If the Python installation has SSL support (i.e., if the `ssl` module can be imported), `HTTPSHandler` will also be added.

A `BaseHandler` subclass may also change its **`handler_order`** attribute to modify its position in the handlers list.

urllib.request.pathname2url(*path*, *, *add_scheme=False*)

Convert the given local path to a `file:` URL. This function uses `quote()` function to encode the path.

If *add_scheme* is false (the default), the return value omits the `file:` scheme prefix. Set *add_scheme* to true to return a complete URL.

This example shows the function being used on Windows:

```
>>> from urllib.request import pathname2url
>>> path = 'C:\\Program Files'
>>> pathname2url(path, add_scheme=True)
'file:///C:/Program%20Files'
```

Changed in version 3.14: Windows drive letters are no longer converted to uppercase, and `:` characters not following a drive letter no longer cause an `OSError` exception to be raised on Windows.

Changed in version 3.14: Paths beginning with a slash are converted to URLs with authority sections. For example, the path `/etc/hosts` is converted to the URL `///etc/hosts`.

Changed in version 3.14: The *add_scheme* parameter was added.

urllib.request.url2pathname(*url*, *, *require_scheme=False*, *resolve_host=False*)

Convert the given `file:` URL to a local path. This function uses `unquote()` to decode the URL.

If *require_scheme* is false (the default), the given value should omit a `file:` scheme prefix. If *require_scheme* is set to true, the given value should include the prefix; a `URLError` is raised if it doesn’t.

The URL authority is discarded if it is empty, `localhost`, or the local hostname. Otherwise, if *resolve_host* is set to true, the authority is resolved using **`socket.gethostbyname()`** and discarded if it matches a local IP address (as per **RFC 8089 §3** [https://datatracker.ietf.org/doc/html/rfc8089.html#section-3]). If the authority is still unhandled, then on Windows a UNC path is returned, and on other platforms a **`URLError`** is raised.

This example shows the function being used on Windows:

```
>>> from urllib.request import url2pathname
>>> url = 'file:///C:/Program%20Files'
>>> url2pathname(url, require_scheme=True)
'C:\\Program Files'
```

Changed in version 3.14: Windows drive letters are no longer converted to uppercase, and `:` characters not following a drive letter no longer cause an **`OSError`** exception to be raised on Windows.

Changed in version 3.14: The URL authority is discarded if it matches the local hostname. Otherwise, if the authority isn't empty or `localhost`, then on Windows a UNC path is returned (as before), and on other platforms a **`URLError`** is raised.

Changed in version 3.14: The URL query and fragment components are discarded if present.

Changed in version 3.14: The *require_scheme* and *resolve_host* parameters were added.

urllib.request.getproxies()

This helper function returns a dictionary of scheme to proxy server URL mappings. It scans the environment for variables named `<scheme>_proxy`, in a case insensitive approach, for all operating systems first, and when it cannot find it, looks for proxy information from System Configuration for macOS and Windows Systems Registry for Windows. If both lowercase and uppercase environment variables exist (and disagree), lowercase is preferred.

> **Note**
>
> If the environment variable `REQUEST_METHOD` is set, which usually indicates your script is running in a CGI environment, the environment variable `HTTP_PROXY` (uppercase `_PROXY`) will be ignored. This is because that variable can be injected by a client using the "Proxy:" HTTP header. If you need to use an HTTP proxy in a CGI environment, either use `ProxyHandler` explicitly, or make sure the variable name is in lowercase (or at least the `_proxy` suffix).

The following classes are provided:

class urllib.request.Request(*url*, *data=None*, *headers={}*, *origin_req_host=None*, *unverifiable=False*, *method=None*)

This class is an abstraction of a URL request.

url should be a string containing a valid, properly encoded URL.

data must be an object specifying additional data to send to the server, or `None` if no such data is needed. Currently HTTP requests are the only ones that use *data*. The supported object types include bytes, file-like objects, and iterables of bytes-like objects. If no `Content-Length` nor `Transfer-Encoding` header field has been provided, **`HTTPHandler`** will set these headers according to the type of *data*. `Content-Length` will be used to send bytes objects, while `Transfer-Encoding: chunked` as specified in **RFC 7230** [https://datatracker.ietf.org/doc/html/rfc7230.html], Section 3.3.1 will be used to send files and other iterables.

For an HTTP POST request method, *data* should be a buffer in the standard *application/x-www-form-urlencoded* format. The **`urllib.parse.urlencode()`** function takes a mapping or sequence of 2-tuples and returns an ASCII string in this format. It should be encoded to bytes before being used as the *data* parameter.

headers should be a dictionary, and will be treated as if **`add_header()`** was called with each key and value as arguments. This is often used to "spoof" the `User-Agent` header value, which is used by a browser to identify itself – some HTTP servers only allow requests coming from common browsers as opposed to scripts. For example, Mozilla Firefox may identify itself as `"Mozilla/5.0 (X11; U; Linux i686) Gecko/20071127 Firefox/2.0.0.11"`, while **`urllib`**'s default user agent string is `"Python-urllib/2.6"` (on Python 2.6). All header keys are sent in camel case.

An appropriate `Content-Type` header should be included if the *data* argument is present. If this header has not been provided and *data* is not `None`, `Content-Type: application/x-www-form-urlencoded` will be added as a default.

The next two arguments are only of interest for correct handling of third-party HTTP cookies:

origin_req_host should be the request-host of the origin transaction, as defined by **RFC 2965** [https://datatracker.ietf.org/doc/html/rfc2965.html]. It defaults to `http.cookiejar.request_host(self)`. This is the host name or IP address of the original request that was initiated by the user. For example, if the request is for an image in an HTML document, this should be the request-host of the request for the page containing the image.

unverifiable should indicate whether the request is unverifiable, as defined by **RFC 2965** [https://datatracker.ietf.org/doc/html/rfc2965.html]. It defaults to `False`. An unverifiable request is one whose URL the user did not have the option to approve. For example, if the request is for an image in an HTML document, and the user had no option to approve the automatic fetching of the image, this should be true.

method should be a string that indicates the HTTP request method that will be used (e.g. `'HEAD'`). If provided, its value is stored in the `method` attribute and is used by `get_method()`. The default is `'GET'` if *data* is `None` or `'POST'` otherwise. Subclasses may indicate a different default method by setting the `method` attribute in the class itself.

Note

The request will not work as expected if the data object is unable to deliver its content more than once (e.g. a file or an iterable that can produce the content only once) and the request is retried for HTTP redirects or authentication. The *data* is sent to the HTTP server right away after the headers. There is no support for a 100-continue expectation in the library.

Changed in version 3.3: `Request.method` argument is added to the Request class.

Changed in version 3.4: Default `Request.method` may be indicated at the class level.

Changed in version 3.6: Do not raise an error if the `Content-Length` has not been provided and *data* is neither `None` nor a bytes object. Fall back to use chunked transfer encoding instead.

class urllib.request.OpenerDirector

The **`OpenerDirector`** class opens URLs via **`BaseHandler`**s chained together. It manages the chaining of handlers, and recovery from errors.

class urllib.request.BaseHandler

This is the base class for all registered handlers — and handles only the simple mechanics of registration.

class urllib.request.HTTPDefaultErrorHandler

A class which defines a default handler for HTTP error responses; all responses are turned into **`HTTPError`** exceptions.

class urllib.request.HTTPRedirectHandler

A class to handle redirections.

class urllib.request.HTTPCookieProcessor(*cookiejar=None*)

A class to handle HTTP Cookies.

class urllib.request.ProxyHandler(*proxies=None*)

Cause requests to go through a proxy. If *proxies* is given, it must be a dictionary mapping protocol names to URLs of proxies. The default is to read the list of proxies from the environment variables `<protocol>_proxy`. If no proxy environment variables are set, then in a Windows environment proxy settings are obtained from the registry’s Internet Settings section, and in a macOS environment proxy information is retrieved from the System Configuration Framework.

To disable autodetected proxy pass an empty dictionary.

The **`no_proxy`** environment variable can be used to specify hosts which shouldn't be reached via proxy; if set, it should be a comma-separated list of hostname suffixes, optionally with `:port` appended, for example `cern.ch,ncsa.uiuc.edu,some.host:8080`.

Note

`HTTP_PROXY` will be ignored if a variable `REQUEST_METHOD` is set; see the documentation on `getproxies()`.

class urllib.request.HTTPPasswordMgr

Keep a database of `(realm, uri) -> (user, password)` mappings.

class urllib.request.HTTPPasswordMgrWithDefaultRealm

Keep a database of `(realm, uri) -> (user, password)` mappings. A realm of `None` is considered a catch-all realm, which is searched if no other realm fits.

class urllib.request.HTTPPasswordMgrWithPriorAuth

A variant of `HTTPPasswordMgrWithDefaultRealm` that also has a database of `uri -> is_authenticated` mappings. Can be used by a BasicAuth handler to determine when to send authentication credentials immediately instead of waiting for a `401` response first.

Added in version 3.5.

class urllib.request.AbstractBasicAuthHandler(*password_mgr=None*)

This is a mixin class that helps with HTTP authentication, both to the remote host and to a proxy. *password_mgr*, if given, should be something that is compatible with **`HTTPPasswordMgr`**; refer to section HTTPPasswordMgr Objects for information on the interface that must be supported. If *passwd_mgr* also provides `is_authenticated` and `update_authenticated` methods (see HTTPPasswordMgrWithPriorAuth Objects), then the handler will use the `is_authenticated` result for a given URI to determine whether or not to send authentication credentials with the request. If `is_authenticated` returns `True` for the URI, credentials are sent. If `is_authenticated` is `False`, credentials are not sent, and then if a `401` response is received the request is re-sent with the authentication credentials. If authentication succeeds, `update_authenticated` is called to set `is_authenticated` `True` for the URI, so that subsequent requests to the URI or any of its super-URIs will automatically include the authentication credentials.

Added in version 3.5: Added `is_authenticated` support.

class
urllib.request.HTTPBasicAuthHandler(*password_mgr=None*)

Handle authentication with the remote host. *password_mgr*, if given, should be something that is compatible with **`HTTPPasswordMgr`**; refer to section HTTPPasswordMgr Objects for information on the interface that must be supported. HTTPBasicAuthHandler will raise a **`ValueError`** when presented with a wrong Authentication scheme.

class
urllib.request.ProxyBasicAuthHandler(*password_mgr=None*)

Handle authentication with the proxy. *password_mgr*, if given, should be something that is compatible with

`HTTPPasswordMgr`; refer to section HTTPPasswordMgr Objects for information on the interface that must be supported.

class urllib.request.AbstractDigestAuthHandler(*password_mgr=None*)

This is a mixin class that helps with HTTP authentication, both to the remote host and to a proxy. *password_mgr*, if given, should be something that is compatible with **`HTTPPasswordMgr`**; refer to section HTTPPasswordMgr Objects for information on the interface that must be supported.

Changed in version 3.14: Added support for HTTP digest authentication algorithm `SHA-256`.

class urllib.request.HTTPDigestAuthHandler(*password_mgr=None*)

Handle authentication with the remote host. *password_mgr*, if given, should be something that is compatible with **`HTTPPasswordMgr`**; refer to section HTTPPasswordMgr Objects for information on the interface that must be supported. When both Digest Authentication Handler and Basic Authentication Handler are both added, Digest Authentication is always tried first. If the Digest Authentication returns a 40x response again, it is sent to Basic Authentication handler to Handle. This Handler method will raise a **`ValueError`** when presented with an authentication scheme other than Digest or Basic.

Changed in version 3.3: Raise **`ValueError`** on unsupported Authentication Scheme.

class urllib.request.ProxyDigestAuthHandler(*password_mgr=None*)

Handle authentication with the proxy. *password_mgr*, if given, should be something that is compatible with `HTTPPasswordMgr`; refer to section HTTPPasswordMgr Objects for information on the interface that must be supported.

class urllib.request.HTTPHandler

A class to handle opening of HTTP URLs.

class urllib.request.HTTPSHandler(*debuglevel=0*, *context=None*, *check_hostname=None*)

A class to handle opening of HTTPS URLs. *context* and *check_hostname* have the same meaning as in `http.client.HTTPSConnection`.

Changed in version 3.2: context and *check_hostname* were added.

class urllib.request.FileHandler

Open local files.

class urllib.request.DataHandler

Open data URLs.

Added in version 3.4.

class urllib.request.FTPHandler

Open FTP URLs.

class urllib.request.CacheFTPHandler

Open FTP URLs, keeping a cache of open FTP connections to minimize delays.

class urllib.request.UnknownHandler

A catch-all class to handle unknown URLs.

class urllib.request.HTTPErrorProcessor

Process HTTP error responses.

Request Objects

The following methods describe `Request`'s public interface, and so all may be overridden in subclasses. It also defines several public attributes that can be used by clients to inspect the parsed request.

Request.full_url

The original URL passed to the constructor.

Changed in version 3.4.

Request.full_url is a property with setter, getter and a deleter. Getting `full_url` returns the original request URL with the fragment, if it was present.

Request.type

The URI scheme.

Request.host

The URI authority, typically a host, but may also contain a port separated by a colon.

Request.origin_req_host

The original host for the request, without port.

Request.selector

The URI path. If the `Request` uses a proxy, then selector will be the full URL that is passed to the proxy.

Request.data

The entity body for the request, or `None` if not specified.

Changed in version 3.4: Changing value of `Request.data` now deletes “Content-Length” header if it was previously set or calculated.

Request.unverifiable

boolean, indicates whether the request is unverifiable as defined by **RFC 2965** [https://datatracker.ietf.org/doc/html/rfc2965.html].

Request.method

The HTTP request method to use. By default its value is `None`, which means that `get_method()` will do its normal computation of the method to be used. Its value can be set (thus overriding the default computation in `get_method()`) either by providing a default value by setting it at the class level in a `Request` subclass, or by passing a value in to the `Request` constructor via the *method* argument.

Added in version 3.3.

Changed in version 3.4: A default value can now be set in subclasses; previously it could only be set via the constructor argument.

Request.get_method()

Return a string indicating the HTTP request method. If `Request.method` is not `None`, return its value, otherwise return `'GET'` if `Request.data` is `None`, or `'POST'` if it’s not. This is only meaningful for HTTP requests.

Changed in version 3.3: get_method now looks at the value of `Request.method`.

Request.add_header(*key*, *val*)

Add another header to the request. Headers are currently ignored by all handlers except HTTP handlers, where they are added to the list of headers sent to the server. Note that there cannot be more than one header with the same name, and later calls will overwrite previous calls in case the *key* collides. Currently, this is no loss of HTTP functionality, since all headers which have meaning when used more than once have a (header-specific) way of gaining the same functionality using only one header. Note that headers added using this method are also added to redirected requests.

Request.add_unredirected_header(*key*, *header*)

Add a header that will not be added to a redirected request.

Request.has_header(*header*)

Return whether the instance has the named header (checks both regular and unredirected).

Request.remove_header(*header*)

Remove named header from the request instance (both from regular and unredirected headers).

Added in version 3.4.

Request.get_full_url()

Return the URL given in the constructor.

Changed in version 3.4.

Returns **`Request.full_url`**

Request.set_proxy(*host*, *type*)

Prepare the request by connecting to a proxy server. The *host* and *type* will replace those of the instance, and the instance's selector will be the original URL given in the constructor.

Request.get_header(*header_name*, *default=None*)

Return the value of the given header. If the header is not present, return the default value.

Request.header_items()

Return a list of tuples (header_name, header_value) of the Request headers.

Changed in version 3.4: The request methods add_data, has_data, get_data, get_type, get_host, get_selector, get_origin_req_host and is_unverifiable that were deprecated since 3.3 have been removed.

OpenerDirector Objects

`OpenerDirector` instances have the following methods:

OpenerDirector.add_handler(*handler*)

handler should be an instance of `BaseHandler`. The following methods are searched, and added to the possible chains (note that HTTP errors are a special case). Note that, in the following, *protocol* should be replaced with the actual protocol to handle, for example **`http_response()`** would be the HTTP protocol response handler. Also *type* should be replaced with the actual HTTP code, for example **`http_error_404()`** would handle HTTP 404 errors.

- **`<protocol>_open()`** — signal that the handler knows how to open *protocol* URLs.

See `BaseHandler.<protocol>_open()` for more information.

- **`http_error_<type>()`** — signal that the handler knows how to handle HTTP errors with HTTP error code *type*.

  See `BaseHandler.http_error_<nnn>()` for more information.

- **`<protocol>_error()`** — signal that the handler knows how to handle errors from (non-`http`) *protocol*.

- **`<protocol>_request()`** — signal that the handler knows how to pre-process *protocol* requests.

  See `BaseHandler.<protocol>_request()` for more information.

- **`<protocol>_response()`** — signal that the handler knows how to post-process *protocol* responses.

  See `BaseHandler.<protocol>_response()` for more information.

OpenerDirector.open(*url*, *data=None*[, *timeout*])

Open the given *url* (which can be a request object or a string), optionally passing the given *data*. Arguments, return values and exceptions raised are the same as those of `urlopen()` (which simply calls the `open()` method on the currently installed global `OpenerDirector`). The optional *timeout* parameter specifies a timeout in seconds for blocking operations like the connection attempt (if not specified, the global default timeout setting will be used). The timeout feature actually works only for HTTP, HTTPS and FTP connections.

OpenerDirector.error(*proto*, **args*)

Handle an error of the given protocol. This will call the registered error handlers for the given protocol with the given arguments (which are protocol specific). The HTTP protocol is a special case which uses the HTTP response code to determine the specific error handler; refer to the **`http_error_<type>()`** methods of the handler classes.

Return values and exceptions raised are the same as those of `urlopen()`.

OpenerDirector objects open URLs in three stages:

The order in which these methods are called within each stage is determined by sorting the handler instances.

1. Every handler with a method named like **`<protocol>_request()`** has that method called to pre-process the request.

2. Handlers with a method named like **`<protocol>_open()`** are called to handle the request. This stage ends when a handler either returns a non-`None` value (ie. a response), or raises an exception (usually `URLError`). Exceptions are allowed to propagate.

   In fact, the above algorithm is first tried for methods named `default_open()`. If all such methods return `None`, the algorithm is repeated for methods named like **`<protocol>_open()`**. If all such methods return `None`, the algorithm is repeated for methods named `unknown_open()`.

   Note that the implementation of these methods may involve calls of the parent `OpenerDirector` instance’s `open()` and `error()` methods.

3. Every handler with a method named like **`<protocol>_response()`** has that method called to post-

process the response.

BaseHandler Objects

`BaseHandler` objects provide a couple of methods that are directly useful, and others that are meant to be used by derived classes. These are intended for direct use:

BaseHandler.add_parent(*director*)

Add a director as parent.

BaseHandler.close()

Remove any parents.

The following attribute and methods should only be used by classes derived from `BaseHandler`.

Note

The convention has been adopted that subclasses defining `<protocol>_request()` or `<protocol>_response()` methods are named `*Processor`; all others are named `*Handler`.

BaseHandler.parent

A valid `OpenerDirector`, which can be used to open using a different protocol, or handle errors.

BaseHandler.default_open(*req*)

This method is *not* defined in `BaseHandler`, but subclasses should define it if they want to catch all URLs.

This method, if implemented, will be called by the parent `OpenerDirector`. It should return a file-like object as described in the return value of the `open()` method of

`OpenerDirector`, or `None`. It should raise `URLError`, unless a truly exceptional thing happens (for example, `MemoryError` should not be mapped to `URLError`).

This method will be called before any protocol-specific open method.

BaseHandler.<protocol>_open(req)

This method is *not* defined in `BaseHandler`, but subclasses should define it if they want to handle URLs with the given protocol.

This method, if defined, will be called by the parent `OpenerDirector`. Return values should be the same as for `default_open()`.

BaseHandler.unknown_open(*req*)

This method is *not* defined in `BaseHandler`, but subclasses should define it if they want to catch all URLs with no specific registered handler to open it.

This method, if implemented, will be called by the `parent` `OpenerDirector`. Return values should be the same as for `default_open()`.

BaseHandler.http_error_default(*req*, *fp*, *code*, *msg*, *hdrs*)

This method is *not* defined in `BaseHandler`, but subclasses should override it if they intend to provide a catch-all for otherwise unhandled HTTP errors. It will be called automatically by the `OpenerDirector` getting the error, and should not normally be called in other circumstances.

`OpenerDirector` will call this method with five positional arguments:

1. a `Request` object,

2. a file-like object with the HTTP error body,
3. the three-digit code of the error, as a string,
4. the user-visible explanation of the code, as a string, and
5. the headers of the error, as a mapping object.

Return values and exceptions raised should be the same as those of `urlopen()`.

BaseHandler.http_error_<nnn>(req, fp, code, msg, hdrs)

nnn should be a three-digit HTTP error code. This method is also not defined in `BaseHandler`, but will be called, if it exists, on an instance of a subclass, when an HTTP error with code *nnn* occurs.

Subclasses should override this method to handle specific HTTP errors.

Arguments, return values and exceptions raised should be the same as for `http_error_default()`.

BaseHandler.<protocol>_request(req)

This method is *not* defined in `BaseHandler`, but subclasses should define it if they want to pre-process requests of the given protocol.

This method, if defined, will be called by the parent `OpenerDirector`. *req* will be a `Request` object. The return value should be a `Request` object.

BaseHandler.<protocol>_response(req, response)

This method is *not* defined in `BaseHandler`, but subclasses should define it if they want to post-process responses of the given protocol.

This method, if defined, will be called by the parent `OpenerDirector`. *req* will be a `Request` object. *response* will be an object implementing the same interface as the return value of `urlopen()`. The return value should implement the same interface as the return value of `urlopen()`.

HTTPRedirectHandler Objects

Note

Some HTTP redirections require action from this module's client code. If this is the case, `HTTPError` is raised. See **RFC 2616** [https://datatracker.ietf.org/doc/html/rfc2616.html] for details of the precise meanings of the various redirection codes.

An `HTTPError` exception raised as a security consideration if the HTTPRedirectHandler is presented with a redirected URL which is not an HTTP, HTTPS or FTP URL.

HTTPRedirectHandler.redirect_request(*req*, *fp*, *code*, *msg*, *hdrs*, *newurl*)

Return a `Request` or `None` in response to a redirect. This is called by the default implementations of the **`http_error_30*()`** methods when a redirection is received from the server. If a redirection should take place, return a new `Request` to allow **`http_error_30*()`** to perform the redirect to *newurl*. Otherwise, raise `HTTPError` if no other handler should try to handle this URL, or return `None` if you can't but another handler might.

Note

The default implementation of this method does not strictly follow **RFC 2616** [https://datatracker.ietf.org/doc/html/rfc2616.html], which says that

301 and 302 responses to `POST` requests must not be automatically redirected without confirmation by the user. In reality, browsers do allow automatic redirection of these responses, changing the POST to a `GET`, and the default implementation reproduces this behavior.

HTTPRedirectHandler.http_error_301(*req*, *fp*, *code*, *msg*, *hdrs*)

Redirect to the `Location:` or `URI:` URL. This method is called by the parent **`OpenerDirector`** when getting an HTTP 'moved permanently' response.

HTTPRedirectHandler.http_error_302(*req*, *fp*, *code*, *msg*, *hdrs*)

The same as **`http_error_301()`**, but called for the 'found' response.

HTTPRedirectHandler.http_error_303(*req*, *fp*, *code*, *msg*, *hdrs*)

The same as **`http_error_301()`**, but called for the 'see other' response.

HTTPRedirectHandler.http_error_307(*req*, *fp*, *code*, *msg*, *hdrs*)

The same as **`http_error_301()`**, but called for the 'temporary redirect' response. It does not allow changing the request method from `POST` to `GET`.

HTTPRedirectHandler.http_error_308(*req*, *fp*, *code*, *msg*, *hdrs*)

The same as **`http_error_301()`**, but called for the 'permanent redirect' response. It does not allow changing the request method from `POST` to `GET`.

Added in version 3.11.

HTTPCookieProcessor Objects

`HTTPCookieProcessor` instances have one attribute:

HTTPCookieProcessor.cookiejar

The `http.cookiejar.CookieJar` in which cookies are stored.

ProxyHandler Objects

ProxyHandler.<protocol>_open(request)

The `ProxyHandler` will have a method **`<protocol>_open()`** for every *protocol* which has a proxy in the *proxies* dictionary given in the constructor. The method will modify requests to go through the proxy, by calling `request.set_proxy()`, and call the next handler in the chain to actually execute the protocol.

HTTPPasswordMgr Objects

These methods are available on `HTTPPasswordMgr` and `HTTPPasswordMgrWithDefaultRealm` objects.

HTTPPasswordMgr.add_password(*realm*, *uri*, *user*, *passwd*)

uri can be either a single URI, or a sequence of URIs. *realm*, *user* and *passwd* must be strings. This causes `(user, passwd)` to be used as authentication tokens when authentication for *realm* and a super-URI of any of the given URIs is given.

HTTPPasswordMgr.find_user_password(*realm*, *authuri*)

Get user/password for given realm and URI, if any. This method will return `(None, None)` if there is no matching user/password.

For `HTTPPasswordMgrWithDefaultRealm` objects, the realm `None` will be searched if the given *realm* has no matching user/password.

HTTPPasswordMgrWithPriorAuth Objects

This password manager extends `HTTPPasswordMgrWithDefaultRealm` to support tracking URIs for which authentication credentials should always be sent.

HTTPPasswordMgrWithPriorAuth.add_password(*realm*, *uri*, *user*, *passwd*, *is_authenticated=False*)

realm, *uri*, *user*, *passwd* are as for `HTTPPasswordMgr.add_password()`. *is_authenticated* sets the initial value of the `is_authenticated` flag for the given URI or list of URIs. If *is_authenticated* is specified as `True`, *realm* is ignored.

HTTPPasswordMgrWithPriorAuth.find_user_password(*realm*, *authuri*)

Same as for `HTTPPasswordMgrWithDefaultRealm` objects

HTTPPasswordMgrWithPriorAuth.update_authenticated(*self*, *uri*, *is_authenticated=False*)

Update the `is_authenticated` flag for the given *uri* or list of URIs.

HTTPPasswordMgrWithPriorAuth.is_authenticated(*self*, *authuri*)

Returns the current state of the `is_authenticated` flag for the given URI.

AbstractBasicAuthHandler Objects

AbstractBasicAuthHandler.http_error_auth_reqed(*authreq*, *host*, *req*, *headers*)

Handle an authentication request by getting a user/password pair, and re-trying the request. *authreq* should be the name of the header where the information about the realm is included in the request, *host* specifies the URL and path to authenticate for, *req* should be the (failed) `Request` object, and *headers* should be the error headers.

host is either an authority (e.g. `"python.org"`) or a URL containing an authority component (e.g. `"http://python.org/"`). In either case, the authority must not contain a userinfo component (so, `"python.org"` and `"python.org:80"` are fine, `"joe:password@python.org"` is not).

HTTPBasicAuthHandler Objects

HTTPBasicAuthHandler.http_error_401(*req*, *fp*, *code*, *msg*, *hdrs*)

Retry the request with authentication information, if available.

ProxyBasicAuthHandler Objects

ProxyBasicAuthHandler.http_error_407(*req*, *fp*, *code*, *msg*, *hdrs*)

Retry the request with authentication information, if available.

AbstractDigestAuthHandler Objects

AbstractDigestAuthHandler.http_error_auth_reqed(*authreq*, *host*, *req*, *headers*)

authreq should be the name of the header where the information about the realm is included in the request, *host* should be the host to authenticate to, *req* should be the (failed) **`Request`** object, and *headers* should be the error headers.

HTTPDigestAuthHandler Objects

HTTPDigestAuthHandler.http_error_401(*req*, *fp*, *code*, *msg*, *hdrs*)

Retry the request with authentication information, if available.

ProxyDigestAuthHandler Objects

ProxyDigestAuthHandler.http_error_407(*req*, *fp*, *code*, *msg*, *hdrs*)

Retry the request with authentication information, if available.

HTTPHandler Objects

HTTPHandler.http_open(*req*)

Send an HTTP request, which can be either GET or POST, depending on `req.data`.

HTTPSHandler Objects

HTTPSHandler.https_open(*req*)

Send an HTTPS request, which can be either GET or POST, depending on `req.data`.

FileHandler Objects

FileHandler.file_open(*req*)

Open the file locally, if there is no host name, or the host name is `'localhost'`.

Changed in version 3.2: This method is applicable only for local hostnames. When a remote hostname is given, a `URLError` is raised.

DataHandler Objects

DataHandler.data_open(*req*)

Read a data URL. This kind of URL contains the content encoded in the URL itself. The data URL syntax is specified in **RFC 2397** [https://datatracker.ietf.org/doc/html/rfc2397.html]. This implementation ignores white spaces in base64 encoded data URLs so the URL may be wrapped in whatever source file it comes from. But even though some browsers don’t mind about a missing padding at the end of a base64 encoded data URL, this implementation will raise a `ValueError` in that case.

FTPHandler Objects

FTPHandler.ftp_open(*req*)

Open the FTP file indicated by *req*. The login is always done with empty username and password.

CacheFTPHandler Objects

`CacheFTPHandler` objects are `FTPHandler` objects with the following additional methods:

CacheFTPHandler.setTimeout(*t*)

Set timeout of connections to *t* seconds.

CacheFTPHandler.setMaxConns(*m*)

Set maximum number of cached connections to *m*.

UnknownHandler Objects

UnknownHandler.unknown_open()

Raise a `URLError` exception.

HTTPErrorProcessor Objects

HTTPErrorProcessor.http_response(*request*, *response*)

Process HTTP error responses.

For 200 error codes, the response object is returned immediately.

For non-200 error codes, this simply passes the job on to the **`http_error_<type>()`** handler methods, via `OpenerDirector.error()`. Eventually, `HTTPDefaultErrorHandler` will raise an `HTTPError` if no other handler handles the error.

HTTPErrorProcessor.https_response(*request*, *response*)

Process HTTPS error responses.

The behavior is same as `http_response()`.

Examples

In addition to the examples below, more examples are given in [HOWTO Fetch Internet Resources Using The urllib Package](#).

This example gets the python.org main page and displays the first 300 bytes of it:

```
>>> import urllib.request
>>> with urllib.request.urlopen('http://www.python.org/') as f:
...     print(f.read(300))
...
b'<!doctype html>\n<!--[if lt IE 7]>   <html class="no-js ie6
lt-ie7 lt-ie8 lt-ie9">   <![endif]-->\n<!--[if IE 7]>
<html class="no-js ie7 lt-ie8 lt-ie9">          <![endif]--
>\n<!--[if IE 8]>      <html class="no-js ie8 lt-ie9">
```

Note that urlopen returns a bytes object. This is because there is no way for urlopen to automatically determine the encoding of the byte stream it receives from the HTTP server. In general, a program will decode the returned bytes object to string once it determines or guesses the appropriate encoding.

The following HTML spec document, https://html.spec.whatwg.org/#charset, lists the various ways in which an HTML or an XML document could have specified its encoding information.

For additional information, see the W3C document: https://www.w3.org/International/questions/qa-html-encoding-declarations.

As the python.org website uses *utf-8* encoding as specified in its meta tag, we will use the same for decoding the bytes object:

```
>>> with urllib.request.urlopen('http://www.python.org/') as f:
...     print(f.read(100).decode('utf-8'))
...
<!doctype html>
<!--[if lt IE 7]>   <html class="no-js ie6 lt-ie7 lt-ie8 lt-
ie9">   <![endif]-->
<!-
```

It is also possible to achieve the same result without using the context manager approach:

```
>>> import urllib.request
>>> f = urllib.request.urlopen('http://www.python.org/')
>>> try:
...     print(f.read(100).decode('utf-8'))
... finally:
...     f.close()
...
<!doctype html>
<!--[if lt IE 7]>   <html class="no-js ie6 lt-ie7 lt-ie8 lt-
ie9">   <![endif]-->
<!--
```

In the following example, we are sending a data-stream to the stdin of a CGI and reading the data it returns to us. Note that this example will only work when the Python installation supports SSL.

```
>>> import urllib.request
>>> req = urllib.request.Request(url='https://localhost/cgi-
bin/test.cgi',
...                       data=b'This data is passed to stdin
of the CGI')
>>> with urllib.request.urlopen(req) as f:
...     print(f.read().decode('utf-8'))
...
Got Data: "This data is passed to stdin of the CGI"
```

The code for the sample CGI used in the above example is:

```
#!/usr/bin/env python
import sys
data = sys.stdin.read()
print('Content-type: text/plain\n\nGot Data: "%s"' % data)
```

Here is an example of doing a `PUT` request using **`Request`**:

```
import urllib.request
DATA = b'some data'
req = urllib.request.Request(url='http://localhost:8080',
data=DATA, method='PUT')
with urllib.request.urlopen(req) as f:
    pass
print(f.status)
print(f.reason)
```

Use of Basic HTTP Authentication:

```
import urllib.request
# Create an OpenerDirector with support for Basic HTTP
Authentication...
auth_handler = urllib.request.HTTPBasicAuthHandler()
auth_handler.add_password(realm='PDQ Application',
                          uri='https://mahler:8092/site-
updates.py',
                          user='klem',
                          passwd='kadidd!ehopper')
opener = urllib.request.build_opener(auth_handler)
# ...and install it globally so it can be used with urlopen.
urllib.request.install_opener(opener)
with
urllib.request.urlopen('http://www.example.com/login.html') as
f:
    print(f.read().decode('utf-8'))
```

`build_opener()` provides many handlers by default, including a **`ProxyHandler`**. By default, **`ProxyHandler`** uses the environment variables named `<scheme>_proxy`, where `<scheme>` is the URL scheme involved. For example, the **`http_proxy`** environment variable is read to obtain the HTTP proxy’s URL.

This example replaces the default `ProxyHandler` with one that uses programmatically supplied proxy URLs, and adds proxy authorization support with `ProxyBasicAuthHandler`.

```
proxy_handler = urllib.request.ProxyHandler({'http':
'http://www.example.com:3128/'})
proxy_auth_handler = urllib.request.ProxyBasicAuthHandler()
proxy_auth_handler.add_password('realm', 'host', 'username',
'password')

opener = urllib.request.build_opener(proxy_handler,
proxy_auth_handler)
# This time, rather than install the OpenerDirector, we use it
directly:
with opener.open('http://www.example.com/login.html') as f:
   print(f.read().decode('utf-8'))
```

Adding HTTP headers:

Use the *headers* argument to the `Request` constructor, or:

```
import urllib.request
req = urllib.request.Request('http://www.example.com/')
req.add_header('Referer', 'http://www.python.org/')
# Customize the default User-Agent header value:
req.add_header('User-Agent', 'urllib-example/0.1 (Contact: . .
.)')
with urllib.request.urlopen(req) as f:
    print(f.read().decode('utf-8'))
```

`OpenerDirector` automatically adds a *User-Agent* header to every `Request`. To change this:

```
import urllib.request
opener = urllib.request.build_opener()
opener.addheaders = [('User-agent', 'Mozilla/5.0')]
with opener.open('http://www.example.com/') as f:
   print(f.read().decode('utf-8'))
```

Also, remember that a few standard headers (*Content-Length*, *Content-Type* and *Host*) are added when the `Request` is passed to [`urlopen()`] (or [`OpenerDirector.open()`]).

Here is an example session that uses the `GET` method to retrieve a URL containing parameters:

```
>>> import urllib.request
>>> import urllib.parse
>>> params = urllib.parse.urlencode({'spam': 1, 'eggs': 2,
'bacon': 0})
>>> url = "http://www.musi-cal.com/cgi-bin/query?%s" % params
>>> with urllib.request.urlopen(url) as f:
...     print(f.read().decode('utf-8'))
...
```

The following example uses the `POST` method instead. Note that params output from urlencode is encoded to bytes before it is sent to urlopen as data:

```
>>> import urllib.request
>>> import urllib.parse
>>> data = urllib.parse.urlencode({'spam': 1, 'eggs': 2,
'bacon': 0})
>>> data = data.encode('ascii')
>>> with urllib.request.urlopen("http://requestb.in/xrbl82xr",
data) as f:
...     print(f.read().decode('utf-8'))
...
```

The following example uses an explicitly specified HTTP proxy, overriding environment settings:

```
>>> import urllib.request
>>> proxies = {'http': 'http://proxy.example.com:8080/'}
>>> opener =
urllib.request.build_opener(urllib.request.ProxyHandler(proxies
))
>>> with opener.open("http://www.python.org") as f:
...     f.read().decode('utf-8')
...
```

The following example uses no proxies at all, overriding environment settings:

```
>>> import urllib.request
>>> opener =
urllib.request.build_opener(urllib.request.ProxyHandler({}))
>>> with opener.open("http://www.python.org/") as f:
...     f.read().decode('utf-8')
...
```

Legacy interface

The following functions and classes are ported from the Python 2 module `urllib` (as opposed to `urllib2`). They might become deprecated at some point in the future.

urllib.request.urlretrieve(*url*, *filename=None*, *reporthook=None*, *data=None*)

Copy a network object denoted by a URL to a local file. If the URL points to a local file, the object will not be copied unless filename is supplied. Return a tuple `(filename, headers)` where *filename* is the local file name under which the object can be found, and *headers* is whatever the **`info()`** method of the object returned by **`urlopen()`** returned (for a remote object). Exceptions are the same as for **`urlopen()`**.

The second argument, if present, specifies the file location to copy to (if absent, the location will be a tempfile with a generated name). The third argument, if present, is a callable that will be called once on establishment of the network connection and once after each block read thereafter. The callable will be passed three arguments; a count of blocks transferred so far, a block size in bytes, and the total size of the file. The third argument may be `-1` on older FTP servers which do not return a file size in response to a retrieval request.

The following example illustrates the most common usage scenario:

```
>>> import urllib.request
>>> local_filename, headers =
urllib.request.urlretrieve('http://python.org/')
>>> html = open(local_filename)
>>> html.close()
```

If the *url* uses the `http:` scheme identifier, the optional *data* argument may be given to specify a `POST` request (normally the request type is `GET`). The *data* argument must be a bytes object in standard *application/x-www-form-urlencoded* format; see the `urllib.parse.urlencode()` function.

`urlretrieve()` will raise `ContentTooShortError` when it detects that the amount of data available was less than the expected amount (which is the size reported by a *Content-Length* header). This can occur, for example, when the download is interrupted.

The *Content-Length* is treated as a lower bound: if there’s more data to read, urlretrieve reads more data, but if less data is available, it raises the exception.

You can still retrieve the downloaded data in this case, it is stored in the **`content`** attribute of the exception instance.

If no *Content-Length* header was supplied, urlretrieve can not check the size of the data it has downloaded, and just returns it. In this case you just have to assume that the download was successful.

urllib.request.urlcleanup()

Cleans up temporary files that may have been left behind by previous calls to `urlretrieve()`.

`urllib.request` Restrictions

- Currently, only the following protocols are supported: HTTP (versions 0.9 and 1.0), FTP, local files, and data URLs.

  Changed in version 3.4: Added support for data URLs.

- The caching feature of `urlretrieve()` has been disabled until someone finds the time to hack proper processing of Expiration time headers.

- There should be a function to query whether a particular URL is in the cache.

- For backward compatibility, if a URL appears to point to a local file but the file can’t be opened, the URL is re-interpreted using the FTP protocol. This can sometimes cause confusing error messages.

- The `urlopen()` and `urlretrieve()` functions can cause arbitrarily long delays while waiting for a network connection to be set up. This means that it is difficult to build an interactive web client using these functions without using threads.

- The data returned by `urlopen()` or `urlretrieve()` is the raw data returned by the server. This may be binary data (such as an image), plain text or (for example) HTML. The HTTP protocol provides type information in the reply header, which can be inspected by looking at the

Content-Type header. If the returned data is HTML, you can use the module `html.parser` to parse it.

- The code handling the FTP protocol cannot differentiate between a file and a directory. This can lead to unexpected behavior when attempting to read a URL that points to a file that is not accessible. If the URL ends in a `/`, it is assumed to refer to a directory and will be handled accordingly. But if an attempt to read a file leads to a 550 error (meaning the URL cannot be found or is not accessible, often for permission reasons), then the path is treated as a directory in order to handle the case when a directory is specified by a URL but the trailing `/` has been left off. This can cause misleading results when you try to fetch a file whose read permissions make it inaccessible; the FTP code will try to read it, fail with a 550 error, and then perform a directory listing for the unreadable file. If fine-grained control is needed, consider using the `ftplib` module.

`urllib.response` — Response classes used by urllib

The `urllib.response` module defines functions and classes which define a minimal file-like interface, including `read()` and `readline()`. Functions defined by this module are used internally by the `urllib.request` module. The typical response object is a `urllib.response.addinfourl` instance:

class urllib.response.addinfourl

url

URL of the resource retrieved, commonly used to determine if a redirect was followed.

headers

Returns the headers of the response in the form of an `EmailMessage` instance.

status

Added in version 3.9.

Status code returned by server.

geturl()

Deprecated since version 3.9: Deprecated in favor of `url`.

info()

Deprecated since version 3.9: Deprecated in favor of `headers`.

code

Deprecated since version 3.9: Deprecated in favor of `status`.

getcode()

Deprecated since version 3.9: Deprecated in favor of `status`.

`urllib.parse` — Parse URLs into components

Source code: Lib/urllib/parse.py
[https://github.com/python/cpython/tree/3.14/Lib/urllib/parse.py]

This module defines a standard interface to break Uniform Resource Locator (URL) strings up in components (addressing scheme, network location, path etc.), to combine the components back into a URL string, and to convert a “relative URL” to an absolute URL given a “base URL.”

The module has been designed to match the internet RFC on Relative Uniform Resource Locators. It supports the following URL schemes: `file`, `ftp`, `gopher`, `hdl`, `http`, `https`, `imap`, `itms-services`, `mailto`, `mms`, `news`, `nntp`, `prospero`, `rsync`, `rtsp`, `rtsps`, `rtspu`, `sftp`, `shttp`, `sip`, `sips`, `snews`, `svn`, `svn+ssh`, `telnet`, `wais`, `ws`, `wss`.

CPython implementation detail: The inclusion of the `itms-services` URL scheme can prevent an app from passing Apple’s App Store review process for the macOS and iOS App Stores. Handling for the `itms-services` scheme is always removed on iOS; on macOS, it *may* be removed if CPython has been built with the **`--with-app-store-compliance`** option.

The **`urllib.parse`** module defines functions that fall into two broad categories: URL parsing and URL quoting. These are covered in detail in the following sections.

This module’s functions use the deprecated term `netloc` (or `net_loc`), which was introduced in **RFC 1808** [https://datatracker.ietf.org/doc/html/rfc1808.html]. However, this term has been obsoleted by **RFC 3986** [https://datatracker.ietf.org/doc/html/rfc3986.html], which introduced the term `authority` as its replacement. The use of `netloc` is continued for backward compatibility.

URL Parsing

The URL parsing functions focus on splitting a URL string into its components, or on combining URL components into a URL string.

urllib.parse.urlparse(*urlstring*, *scheme=''*, *allow_fragments=True*)

Parse a URL into six components, returning a 6-item named tuple. This corresponds to the general structure of a URL: `scheme://netloc/path;parameters?query#fragment`. Each tuple item is a string, possibly empty. The components are not broken up into smaller parts (for example, the network location is a single string), and % escapes are not expanded. The delimiters as shown above are not part of the result, except for a leading slash in the *path* component, which is retained if present. For example:

```
>>> from urllib.parse import urlparse
>>> urlparse("scheme://netloc/path;parameters?
query#fragment")
ParseResult(scheme='scheme', netloc='netloc',
path='/path;parameters', params='',
            query='query', fragment='fragment')
>>> o =
urlparse("http://docs.python.org:80/3/library/urllib.parse.h
tml?"
...              "highlight=params#url-parsing")
>>> o
```

```
ParseResult(scheme='http', netloc='docs.python.org:80',
            path='/3/library/urllib.parse.html', params='',
            query='highlight=params', fragment='url-
parsing')
>>> o.scheme
'http'
>>> o.netloc
'docs.python.org:80'
>>> o.hostname
'docs.python.org'
>>> o.port
80
>>> o._replace(fragment="").geturl()
'http://docs.python.org:80/3/library/urllib.parse.html?
highlight=params'
```

Following the syntax specifications in **RFC 1808** [https://datatracker.ietf.org/doc/html/rfc1808.html], urlparse recognizes a netloc only if it is properly introduced by '//'. Otherwise the input is presumed to be a relative URL and thus to start with a path component.

```
>>> from urllib.parse import urlparse
>>> urlparse('//www.cwi.nl:80/%7Eguido/Python.html')
ParseResult(scheme='', netloc='www.cwi.nl:80',
path='/%7Eguido/Python.html',
            params='', query='', fragment='')
>>> urlparse('www.cwi.nl/%7Eguido/Python.html')
ParseResult(scheme='', netloc='',
path='www.cwi.nl/%7Eguido/Python.html',
            params='', query='', fragment='')
>>> urlparse('help/Python.html')
ParseResult(scheme='', netloc='', path='help/Python.html',
params='',
            query='', fragment='')
```

The *scheme* argument gives the default addressing scheme, to be used only if the URL does not specify one. It should be the same type (text or bytes) as *urlstring*, except that the default value `''` is always allowed, and is automatically converted to `b''` if appropriate.

If the *allow_fragments* argument is false, fragment identifiers are not recognized. Instead, they are parsed as part of the path, parameters or query component, and `fragment` is set to the empty string in the return value.

The return value is a named tuple, which means that its items can be accessed by index or as named attributes, which are:

Attribute	**Index**	**Value**	**Value if not present**
`scheme`	0	URL scheme specifier	*scheme* parameter
`netloc`	1	Network location part	empty string
`path`	2	Hierarchical path	empty string
`params`	3	Parameters for last path element	empty string
`query`	4	Query component	empty string
`fragment`	5	Fragment identifier	empty string
`username`		User name	`None`
`password`		Password	`None`
`hostname`		Host name (lower case)	`None`
`port`		Port number as integer, if present	`None`

Attribute	Index	Value	Value if not present

Reading the **`port`** attribute will raise a **`ValueError`** if an invalid port is specified in the URL. See section Structured Parse Results for more information on the result object.

Unmatched square brackets in the **`netloc`** attribute will raise a **`ValueError`**.

Characters in the **`netloc`** attribute that decompose under NFKC normalization (as used by the IDNA encoding) into any of `/`, `?`, `#`, `@`, or `:` will raise a **`ValueError`**. If the URL is decomposed before parsing, no error will be raised.

As is the case with all named tuples, the subclass has a few additional methods and attributes that are particularly useful. One such method is **`_replace()`**. The **`_replace()`** method will return a new ParseResult object replacing specified fields with new values.

```
>>> from urllib.parse import urlparse
>>> u = urlparse('//www.cwi.nl:80/%7Eguido/Python.html')
>>> u
ParseResult(scheme='', netloc='www.cwi.nl:80',
path='/%7Eguido/Python.html',
            params='', query='', fragment='')
>>> u._replace(scheme='http')
ParseResult(scheme='http', netloc='www.cwi.nl:80',
path='/%7Eguido/Python.html',
            params='', query='', fragment='')
```

Warning

`urlparse()` does not perform validation. See URL parsing security for details.

Changed in version 3.2: Added IPv6 URL parsing capabilities.

Changed in version 3.3: The fragment is now parsed for all URL schemes (unless *allow_fragments* is false), in accordance with **RFC 3986** [https://datatracker.ietf.org/doc/html/rfc3986.html]. Previously, an allowlist of schemes that support fragments existed.

Changed in version 3.6: Out-of-range port numbers now raise `ValueError`, instead of returning `None`.

Changed in version 3.8: Characters that affect netloc parsing under NFKC normalization will now raise `ValueError`.

urllib.parse.parse_qs(*qs*, *keep_blank_values=False*, *strict_parsing=False*, *encoding='utf-8'*, *errors='replace'*, *max_num_fields=None*, *separator='&'*)

Parse a query string given as a string argument (data of type *application/x-www-form-urlencoded*). Data are returned as a dictionary. The dictionary keys are the unique query variable names and the values are lists of values for each name.

The optional argument *keep_blank_values* is a flag indicating whether blank values in percent-encoded queries should be treated as blank strings. A true value indicates that blanks should be retained as blank strings. The default false value indicates that blank values are to be ignored and treated as if they were not included.

The optional argument *strict_parsing* is a flag indicating what to do with parsing errors. If false (the default), errors are silently ignored. If true, errors raise a `ValueError` exception.

The optional *encoding* and *errors* parameters specify how to decode percent-encoded sequences into Unicode characters, as accepted by the **`bytes.decode()`** method.

The optional argument *max_num_fields* is the maximum number of fields to read. If set, then throws a **`ValueError`** if there are more than *max_num_fields* fields read.

The optional argument *separator* is the symbol to use for separating the query arguments. It defaults to `&`.

Use the **`urllib.parse.urlencode()`** function (with the `doseq` parameter set to `True`) to convert such dictionaries into query strings.

Changed in version 3.2: Add *encoding* and *errors* parameters.

Changed in version 3.8: Added *max_num_fields* parameter.

Changed in version 3.10: Added *separator* parameter with the default value of `&`. Python versions earlier than Python 3.10 allowed using both `;` and `&` as query parameter separator. This has been changed to allow only a single separator key, with `&` as the default separator.

Deprecated since version 3.14: Accepting objects with false values (like `0` and `[]`) except empty strings and byte-like objects and `None` is now deprecated.

urllib.parse.parse_qsl(*qs*, *keep_blank_values=False*, *strict_parsing=False*, *encoding='utf-8'*, *errors='replace'*, *max_num_fields=None*, *separator='&'*)

Parse a query string given as a string argument (data of type *application/x-www-form-urlencoded*). Data are returned as a list of name, value pairs.

The optional argument *keep_blank_values* is a flag indicating whether blank values in percent-encoded queries should be treated as blank strings. A true value indicates that blanks should be retained as blank strings. The default false value indicates that blank values are to be ignored and treated as if they were not included.

The optional argument *strict_parsing* is a flag indicating what to do with parsing errors. If false (the default), errors are silently ignored. If true, errors raise a `ValueError` exception.

The optional *encoding* and *errors* parameters specify how to decode percent-encoded sequences into Unicode characters, as accepted by the `bytes.decode()` method.

The optional argument *max_num_fields* is the maximum number of fields to read. If set, then throws a `ValueError` if there are more than *max_num_fields* fields read.

The optional argument *separator* is the symbol to use for separating the query arguments. It defaults to `&`.

Use the `urllib.parse.urlencode()` function to convert such lists of pairs into query strings.

Changed in version 3.2: Add *encoding* and *errors* parameters.

Changed in version 3.8: Added *max_num_fields* parameter.

Changed in version 3.10: Added *separator* parameter with the default value of `&`. Python versions earlier than Python 3.10 allowed using both `;` and `&` as query parameter separator. This has been changed to allow only a single separator key, with `&` as the default separator.

urllib.parse.urlunparse(*parts*)

Construct a URL from a tuple as returned by `urlparse()`. The *parts* argument can be any six-item iterable. This may result in a slightly different, but equivalent URL, if the URL that was parsed originally had unnecessary delimiters (for example, a `?` with an empty query; the RFC states that these are equivalent).

urllib.parse.urlsplit(*urlstring*, *scheme*='', *allow_fragments*=*True*)

This is similar to **`urlparse()`**, but does not split the params from the URL. This should generally be used instead of **`urlparse()`** if the more recent URL syntax allowing parameters to be applied to each segment of the *path* portion of the URL (see **RFC 2396** [https://datatracker.ietf.org/doc/html/rfc2396.html]) is wanted. A separate function is needed to separate the path segments and parameters. This function returns a 5-item named tuple:

```
(addressing scheme, network location, path, query, fragment
identifier).
```

The return value is a named tuple, its items can be accessed by index or as named attributes:

Attribute	Index	Value	Value if not present
`scheme`	0	URL scheme specifier	*scheme* parameter
`netloc`	1	Network location part	empty string
`path`	2	Hierarchical path	empty string
`query`	3	Query component	empty string
`fragment`	4	Fragment identifier	empty string
`username`		User name	`None`
`password`		Password	`None`
`hostname`		Host name (lower case)	`None`
`port`		Port number as integer, if present	`None`

Reading the `port` attribute will raise a `ValueError` if an invalid port is specified in the URL. See section Structured Parse Results for more information on the result object.

Unmatched square brackets in the `netloc` attribute will raise a `ValueError`.

Characters in the `netloc` attribute that decompose under NFKC normalization (as used by the IDNA encoding) into any of `/`, `?`, `#`, `@`, or `:` will raise a `ValueError`. If the URL is decomposed before parsing, no error will be raised.

Following some of the WHATWG spec [https://url.spec.whatwg.org/#concept-basic-url-parser] that updates RFC 3986, leading C0 control and space characters are stripped from the URL. `\n`, `\r` and tab `\t` characters are removed from the URL at any position.

Warning

`urlsplit()` does not perform validation. See URL parsing security for details.

Changed in version 3.6: Out-of-range port numbers now raise `ValueError`, instead of returning `None`.

Changed in version 3.8: Characters that affect netloc parsing under NFKC normalization will now raise `ValueError`.

Changed in version 3.10: ASCII newline and tab characters are stripped from the URL.

Changed in version 3.12: Leading WHATWG C0 control and space characters are stripped from the URL.

urllib.parse.urlunsplit(*parts*)

Combine the elements of a tuple as returned by `urlsplit()` into a complete URL as a string. The *parts* argument can be any five-item iterable. This may result in a slightly different, but equivalent URL, if the URL that was parsed originally had unnecessary delimiters (for example, a ? with an empty query; the RFC states that these are equivalent).

urllib.parse.urljoin(*base*, *url*, *allow_fragments=True*)

Construct a full (“absolute”) URL by combining a “base URL” (*base*) with another URL (*url*). Informally, this uses components of the base URL, in particular the addressing scheme, the network location and (part of) the path, to provide missing components in the relative URL. For example:

```
>>> from urllib.parse import urljoin
>>> urljoin('http://www.cwi.nl/%7Eguido/Python.html',
'FAQ.html')
'http://www.cwi.nl/%7Eguido/FAQ.html'
```

The *allow_fragments* argument has the same meaning and default as for `urlparse()`.

Note

If *url* is an absolute URL (that is, it starts with `//` or `scheme://`), the *url*’s hostname and/or scheme will be present in the result. For example:

```
>>> urljoin('http://www.cwi.nl/%7Eguido/Python.html',
...         '//www.python.org/%7Eguido')
'http://www.python.org/%7Eguido'
```

If you do not want that behavior, preprocess the *url* with `urlsplit()` and `urlunsplit()`, removing possible *scheme* and *netloc* parts.

Warning

Because an absolute URL may be passed as the `url` parameter, it is generally **not secure** to use `urljoin` with an attacker-controlled `url`. For example in, `urljoin("https://website.com/users/", username)`, if `username` can contain an absolute URL, the result of `urljoin` will be the absolute URL.

Changed in version 3.5: Behavior updated to match the semantics defined in **RFC 3986** [https://datatracker.ietf.org/doc/html/rfc3986.html].

urllib.parse.urldefrag(*url*)

If *url* contains a fragment identifier, return a modified version of *url* with no fragment identifier, and the fragment identifier as a separate string. If there is no fragment identifier in *url*, return *url* unmodified and an empty string.

The return value is a named tuple, its items can be accessed by index or as named attributes:

Attribute	Index	Value	Value if not present
`url`	0	URL with no fragment	empty string
`fragment`	1	Fragment identifier	empty string

See section Structured Parse Results for more information on the result object.

Changed in version 3.2: Result is a structured object rather than a simple 2-tuple.

urllib.parse.unwrap(*url*)

Extract the url from a wrapped URL (that is, a string formatted as `<URL:scheme://host/path>`, `<scheme://host/path>`, `URL:scheme://host/path` or `scheme://host/path`). If *url* is not a wrapped URL, it is returned without changes.

URL parsing security

The `urlsplit()` and `urlparse()` APIs do not perform **validation** of inputs. They may not raise errors on inputs that other applications consider invalid. They may also succeed on some inputs that might not be considered URLs elsewhere. Their purpose is for practical functionality rather than purity.

Instead of raising an exception on unusual input, they may instead return some component parts as empty strings. Or components may contain more than perhaps they should.

We recommend that users of these APIs where the values may be used anywhere with security implications code defensively. Do some verification within your code before trusting a returned component part. Does that `scheme` make sense? Is that a sensible `path`? Is there anything strange about that `hostname`? etc.

What constitutes a URL is not universally well defined. Different applications have different needs and desired constraints. For instance the living WHATWG spec [https://url.spec.whatwg.org/#concept-basic-url-parser] describes what user facing web clients such as a web browser require. While **RFC 3986** [https://datatracker.ietf.org/doc/html/rfc3986.html] is more general. These functions incorporate some aspects of both, but cannot be claimed compliant with either. The APIs and existing user code with expectations on specific behaviors predate both standards leading us to be very cautious about making API behavior changes.

Parsing ASCII Encoded Bytes

The URL parsing functions were originally designed to operate on character strings only. In practice, it is useful to be able to manipulate properly quoted and encoded URLs as sequences of ASCII bytes. Accordingly, the URL parsing functions in this module all operate on `bytes` and `bytearray` objects in addition to `str` objects.

If `str` data is passed in, the result will also contain only `str` data. If `bytes` or `bytearray` data is passed in, the result will contain only `bytes` data.

Attempting to mix `str` data with `bytes` or `bytearray` in a single function call will result in a `TypeError` being raised, while attempting to pass in non-ASCII byte values will trigger `UnicodeDecodeError`.

To support easier conversion of result objects between `str` and `bytes`, all return values from URL parsing functions provide either an **`encode()`** method (when the result contains `str` data) or a **`decode()`** method (when the result contains `bytes` data). The signatures of these methods match those of the corresponding `str` and `bytes` methods (except that the default encoding is `'ascii'` rather than `'utf-8'`). Each produces a value of a corresponding type that contains either `bytes` data (for **`encode()`** methods) or `str` data (for **`decode()`** methods).

Applications that need to operate on potentially improperly quoted URLs that may contain non-ASCII data will need to do their own decoding from bytes to characters before invoking the URL parsing methods.

The behaviour described in this section applies only to the URL parsing functions. The URL quoting functions use their own rules when producing or consuming byte sequences as detailed in the documentation of the individual URL quoting functions.

Changed in version 3.2: URL parsing functions now accept ASCII encoded byte sequences

Structured Parse Results

The result objects from the `urlparse()`, `urlsplit()` and `urldefrag()` functions are subclasses of the `tuple` type. These subclasses add the attributes listed in the documentation for those functions, the encoding and decoding support described in the previous section, as well as an additional method:

urllib.parse.SplitResult.geturl()

Return the re-combined version of the original URL as a string. This may differ from the original URL in that the scheme may be normalized to lower case and empty components may be dropped. Specifically, empty parameters, queries, and fragment identifiers will be removed.

For `urldefrag()` results, only empty fragment identifiers will be removed. For `urlsplit()` and `urlparse()` results, all noted changes will be made to the URL returned by this method.

The result of this method remains unchanged if passed back through the original parsing function:

```
>>> from urllib.parse import urlsplit
>>> url = 'HTTP://www.Python.org/doc/#'
>>> r1 = urlsplit(url)
>>> r1.geturl()
'http://www.Python.org/doc/'
>>> r2 = urlsplit(r1.geturl())
>>> r2.geturl()
'http://www.Python.org/doc/'
```

The following classes provide the implementations of the structured parse results when operating on `str` objects:

class urllib.parse.DefragResult(*url*, *fragment*)

Concrete class for `urldefrag()` results containing `str` data. The **`encode()`** method returns a `DefragResultBytes` instance.

Added in version 3.2.

class urllib.parse.ParseResult(*scheme*, *netloc*, *path*, *params*, *query*, *fragment*)

Concrete class for `urlparse()` results containing `str` data. The **`encode()`** method returns a `ParseResultBytes` instance.

class urllib.parse.SplitResult(*scheme*, *netloc*, *path*, *query*, *fragment*)

Concrete class for `urlsplit()` results containing `str` data. The **`encode()`** method returns a `SplitResultBytes` instance.

The following classes provide the implementations of the parse results when operating on `bytes` or `bytearray` objects:

class urllib.parse.DefragResultBytes(*url*, *fragment*)

Concrete class for `urldefrag()` results containing `bytes` data. The **`decode()`** method returns a `DefragResult` instance.

Added in version 3.2.

class urllib.parse.ParseResultBytes(*scheme*, *netloc*, *path*, *params*, *query*, *fragment*)

Concrete class for `urlparse()` results containing `bytes` data. The **`decode()`** method returns a `ParseResult` instance.

Added in version 3.2.

class urllib.parse.SplitResultBytes(*scheme*, *netloc*, *path*, *query*, *fragment*)

Concrete class for `urlsplit()` results containing `bytes` data. The **`decode()`** method returns a `SplitResult` instance.

Added in version 3.2.

URL Quoting

The URL quoting functions focus on taking program data and making it safe for use as URL components by quoting special characters and appropriately encoding non-ASCII text. They also support reversing these operations to recreate the original data from the contents of a URL component if that task isn't already covered by the URL parsing functions above.

urllib.parse.quote(*string*, *safe='/'*, *encoding=None*, *errors=None*)

Replace special characters in *string* using the `%xx` escape. Letters, digits, and the characters `'_.-~'` are never quoted. By default, this function is intended for quoting the path section of a URL. The optional *safe* parameter specifies additional ASCII characters that should not be quoted — its default value is `'/'`.

string may be either a `str` or a `bytes` object.

Changed in version 3.7: Moved from **RFC 2396** [https://datatracker.ietf.org/doc/html/rfc2396.html] to **RFC 3986** [https://datatracker.ietf.org/doc/html/rfc3986.html] for quoting URL strings. "~" is now included in the set of unreserved characters.

The optional *encoding* and *errors* parameters specify how to deal with non-ASCII characters, as accepted by the `str.encode()` method. *encoding* defaults to `'utf-8'`. *errors* defaults to `'strict'`, meaning unsupported characters raise a `UnicodeEncodeError`. *encoding* and *errors* must not be supplied if *string* is a `bytes`, or a `TypeError` is raised.

Note that `quote(string, safe, encoding, errors)` is equivalent to `quote_from_bytes(string.encode(encoding, errors), safe)`.

Example: `quote('/El Niño/')` yields `'/El%20Ni%C3%B1o/'`.

urllib.parse.quote_plus(*string*, *safe*='', *encoding*=*None*, *errors*=*None*)

Like `quote()`, but also replace spaces with plus signs, as required for quoting HTML form values when building up a query string to go into a URL. Plus signs in the original string are escaped unless they are included in *safe*. It also does not have *safe* default to `'/'`.

Example: `quote_plus('/El Niño/')` yields `'%2FEl+Ni%C3%B1o%2F'`.

urllib.parse.quote_from_bytes(*bytes*, *safe*='/')

Like `quote()`, but accepts a `bytes` object rather than a `str`, and does not perform string-to-bytes encoding.

Example: `quote_from_bytes(b'a&\xef')` yields `'a%26%EF'`.

urllib.parse.unquote(*string*, *encoding*='*utf-8*', *errors*='*replace*')

Replace `%xx` escapes with their single-character equivalent. The optional *encoding* and *errors* parameters specify how to decode percent-encoded sequences into

Unicode characters, as accepted by the **`bytes.decode()`** method.

string may be either a **`str`** or a **`bytes`** object.

encoding defaults to `'utf-8'`. *errors* defaults to `'replace'`, meaning invalid sequences are replaced by a placeholder character.

Example: `unquote('/El%20Ni%C3%B1o/')` yields `'/El Niño/'`.

Changed in version 3.9: string parameter supports bytes and str objects (previously only str).

urllib.parse.unquote_plus(*string*, *encoding='utf-8'*, *errors='replace'*)

Like **`unquote()`**, but also replace plus signs with spaces, as required for unquoting HTML form values.

string must be a **`str`**.

Example: `unquote_plus('/El+Ni%C3%B1o/')` yields `'/El Niño/'`.

urllib.parse.unquote_to_bytes(*string*)

Replace `%xx` escapes with their single-octet equivalent, and return a **`bytes`** object.

string may be either a **`str`** or a **`bytes`** object.

If it is a **`str`**, unescaped non-ASCII characters in *string* are encoded into UTF-8 bytes.

Example: `unquote_to_bytes('a%26%EF')` yields `b'a&\xef'`.

urllib.parse.urlencode(*query*, *doseq=False*, *safe=''*, *encoding=None*, *errors=None*, *quote_via=quote_plus*)

Convert a mapping object or a sequence of two-element tuples, which may contain `str` or `bytes` objects, to a percent-encoded ASCII text string. If the resultant string is to be used as a *data* for POST operation with the `urlopen()` function, then it should be encoded to bytes, otherwise it would result in a `TypeError`.

The resulting string is a series of `key=value` pairs separated by `'&'` characters, where both *key* and *value* are quoted using the *quote_via* function. By default, `quote_plus()` is used to quote the values, which means spaces are quoted as a `'+'` character and '/' characters are encoded as `%2F`, which follows the standard for GET requests (`application/x-www-form-urlencoded`). An alternate function that can be passed as *quote_via* is `quote()`, which will encode spaces as `%20` and not encode '/' characters. For maximum control of what is quoted, use `quote` and specify a value for *safe*.

When a sequence of two-element tuples is used as the *query* argument, the first element of each tuple is a key and the second is a value. The value element in itself can be a sequence and in that case, if the optional parameter *doseq* evaluates to `True`, individual `key=value` pairs separated by `'&'` are generated for each element of the value sequence for the key. The order of parameters in the encoded string will match the order of parameter tuples in the sequence.

The *safe*, *encoding*, and *errors* parameters are passed down to *quote_via* (the *encoding* and *errors* parameters are only passed when a query element is a `str`).

To reverse this encoding process, `parse_qs()` and `parse_qsl()` are provided in this module to parse query strings into Python data structures.

Refer to urllib examples to find out how the `urllib.parse.urlencode()` method can be used for generating the query string of a URL or data for a POST request.

Changed in version 3.2: query supports bytes and string objects.

Changed in version 3.5: Added the *quote_via* parameter.

Deprecated since version 3.14: Accepting objects with false values (like `0` and `[]`) except empty strings and byte-like objects and `None` is now deprecated.

See also

WHATWG [https://url.spec.whatwg.org/] - URL Living standard

Working Group for the URL Standard that defines URLs, domains, IP addresses, the application/x-www-form-urlencoded format, and their API.

RFC 3986 [https://datatracker.ietf.org/doc/html/rfc3986.html] - Uniform Resource Identifiers

This is the current standard (STD66). Any changes to urllib.parse module should conform to this. Certain deviations could be observed, which are mostly for backward compatibility purposes and for certain de-facto parsing requirements as commonly observed in major browsers.

RFC 2732 [https://datatracker.ietf.org/doc/html/rfc2732.html] - Format for Literal IPv6 Addresses in URL's.

This specifies the parsing requirements of IPv6 URLs.

RFC 2396 [https://datatracker.ietf.org/doc/html/rfc2396.html] - Uniform Resource Identifiers (URI): Generic Syntax

Document describing the generic syntactic requirements for both Uniform Resource Names (URNs) and Uniform Resource Locators (URLs).

RFC 2368 [https://datatracker.ietf.org/doc/html/rfc2368.html] - The mailto URL scheme.

Parsing requirements for mailto URL schemes.

RFC 1808 [https://datatracker.ietf.org/doc/html/rfc1808.html] - Relative Uniform Resource Locators

This Request For Comments includes the rules for joining an absolute and a relative URL, including a fair number of “Abnormal Examples” which govern the treatment of border cases.

RFC 1738 [https://datatracker.ietf.org/doc/html/rfc1738.html] - Uniform Resource Locators (URL)

This specifies the formal syntax and semantics of absolute URLs.

`urllib.error` — Exception classes raised by urllib.request

Source code: Lib/urllib/error.py
[https://github.com/python/cpython/tree/3.14/Lib/urllib/error.py]

The `urllib.error` module defines the exception classes for exceptions raised by `urllib.request`. The base exception class is `URLError`.

The following exceptions are raised by `urllib.error` as appropriate:

exception urllib.error.URLError

The handlers raise this exception (or derived exceptions) when they run into a problem. It is a subclass of `OSError`.

reason

The reason for this error. It can be a message string or another exception instance.

Changed in version 3.3: `URLError` used to be a subtype of `IOError`, which is now an alias of `OSError`.

exception urllib.error.HTTPError(*url*, *code*, *msg*, *hdrs*, *fp*)

Though being an exception (a subclass of `URLError`), an `HTTPError` can also function as a non-exceptional file-like return value (the same thing that `urlopen()` returns). This is useful when handling exotic HTTP errors, such as requests for authentication.

url

Contains the request URL. An alias for *filename* attribute.

code

An HTTP status code as defined in **RFC 2616** [https://datatracker.ietf.org/doc/html/rfc2616.html]. This numeric value corresponds to a value found in the dictionary of codes as found in `http.server.BaseHTTPRequestHandler.responses`.

reason

This is usually a string explaining the reason for this error. An alias for *msg* attribute.

headers

The HTTP response headers for the HTTP request that caused the `HTTPError`. An alias for *hdrs* attribute.

Added in version 3.4.

fp

A file-like object where the HTTP error body can be read from.

exception urllib.error.ContentTooShortError(*msg*, *content*)

This exception is raised when the `urlretrieve()` function detects that the amount of the downloaded data is less than the expected amount (given by the *Content-Length* header).

content

The downloaded (and supposedly truncated) data.

`urllib.robotparser` — Parser for robots.txt

Source code: Lib/urllib/robotparser.py [https://github.com/python/cpython/tree/3.14/Lib/urllib/robotparser.py]

This module provides a single class, **`RobotFileParser`**, which answers questions about whether or not a particular user agent can fetch a URL on the website that published the `robots.txt` file. For more details on the structure of `robots.txt` files, see http://www.robotstxt.org/orig.html.

class urllib.robotparser.RobotFileParser(*url*='')

This class provides methods to read, parse and answer questions about the `robots.txt` file at *url*.

set_url(*url*)

Sets the URL referring to a `robots.txt` file.

read()

Reads the `robots.txt` URL and feeds it to the parser.

parse(*lines*)

Parses the lines argument.

can_fetch(*useragent*, *url*)

Returns `True` if the *useragent* is allowed to fetch the *url* according to the rules contained in the parsed `robots.txt` file.

mtime()

Returns the time the `robots.txt` file was last fetched. This is useful for long-running web spiders that need to check for new `robots.txt` files periodically.

modified()

Sets the time the `robots.txt` file was last fetched to the current time.

crawl_delay(*useragent*)

Returns the value of the `Crawl-delay` parameter from `robots.txt` for the *useragent* in question. If there is no such parameter or it doesn't apply to the *useragent* specified or the `robots.txt` entry for this parameter has invalid syntax, return `None`.

Added in version 3.6.

request_rate(*useragent*)

Returns the contents of the `Request-rate` parameter from `robots.txt` as a named tuple `RequestRate(requests, seconds)`. If there is no such parameter or it doesn't apply to the *useragent* specified or the `robots.txt` entry for this parameter has invalid syntax, return `None`.

Added in version 3.6.

site_maps()

Returns the contents of the `Sitemap` parameter from `robots.txt` in the form of a **`list()`**. If there is no such parameter or the `robots.txt` entry for this parameter has invalid syntax, return `None`.

Added in version 3.8.

The following example demonstrates basic use of the `RobotFileParser` class:

```
>>> import urllib.robotparser
>>> rp = urllib.robotparser.RobotFileParser()
>>> rp.set_url("http://www.musi-cal.com/robots.txt")
>>> rp.read()
>>> rrate = rp.request_rate("*")
>>> rrate.requests
3
>>> rrate.seconds
20
>>> rp.crawl_delay("*")
6
>>> rp.can_fetch("*", "http://www.musi-cal.com/cgi-bin/search?
city=San+Francisco")
False
>>> rp.can_fetch("*", "http://www.musi-cal.com/")
True
```

`http` — HTTP modules

Source code: Lib/http/__init__.py
[https://github.com/python/cpython/tree/3.14/Lib/http/__init__.py]

`http` is a package that collects several modules for working with the HyperText Transfer Protocol:

- **`http.client`** is a low-level HTTP protocol client; for high-level URL opening use **`urllib.request`**
- **`http.server`** contains basic HTTP server classes based on **`socketserver`**
- **`http.cookies`** has utilities for implementing state management with cookies
- **`http.cookiejar`** provides persistence of cookies

The **`http`** module also defines the following enums that help you work with http related code:

class http.HTTPStatus

Added in version 3.5.

A subclass of **`enum.IntEnum`** that defines a set of HTTP status codes, reason phrases and long descriptions written in English.

Usage:

```
>>> from http import HTTPStatus
>>> HTTPStatus.OK
HTTPStatus.OK
>>> HTTPStatus.OK == 200
True
>>> HTTPStatus.OK.value
```

```
200
>>> HTTPStatus.OK.phrase
'OK'
>>> HTTPStatus.OK.description
'Request fulfilled, document follows'
>>> list(HTTPStatus)
[HTTPStatus.CONTINUE, HTTPStatus.SWITCHING_PROTOCOLS, ...]
```

HTTP status codes

Supported, IANA-registered status codes [https://www.iana.org/assignments/http-status-codes/http-status-codes.xhtml] available in `http.HTTPStatus` are:

Code	Enum Name	Details
`100`	`CONTINUE`	HTTP Semantics **RFC 9110**, Section 15.2.1
`101`	`SWITCHING_PROTOCOLS`	HTTP Semantics **RFC 9110**, Section 15.2.2
`102`	`PROCESSING`	WebDAV **RFC 2518**, Section 10.1
`103`	`EARLY_HINTS`	An HTTP Status Code for Indicating Hints **RFC 8297**
`200`	`OK`	HTTP Semantics **RFC 9110**, Section 15.3.1

Code	Enum Name	Details
201	`CREATED`	HTTP Semantics **RFC 9110**, Section 15.3.2
202	`ACCEPTED`	HTTP Semantics **RFC 9110**, Section 15.3.3
203	`NON_AUTHORITATIVE_INFORMATION`	HTTP Semantics **RFC 9110**, Section 15.3.4
204	`NO_CONTENT`	HTTP Semantics **RFC 9110**, Section 15.3.5
205	`RESET_CONTENT`	HTTP Semantics **RFC 9110**, Section 15.3.6
206	`PARTIAL_CONTENT`	HTTP Semantics **RFC 9110**, Section 15.3.7
207	`MULTI_STATUS`	WebDAV **RFC 4918**, Section 11.1
208	`ALREADY_REPORTED`	WebDAV Binding Extensions **RFC 5842**, Section 7.1 (Experimental)

Code	Enum Name	Details
226	`IM_USED`	Delta Encoding in HTTP **RFC 3229**, Section 10.4.1
300	`MULTIPLE_CHOICES`	HTTP Semantics **RFC 9110**, Section 15.4.1
301	`MOVED_PERMANENTLY`	HTTP Semantics **RFC 9110**, Section 15.4.2
302	`FOUND`	HTTP Semantics **RFC 9110**, Section 15.4.3
303	`SEE_OTHER`	HTTP Semantics **RFC 9110**, Section 15.4.4
304	`NOT_MODIFIED`	HTTP Semantics **RFC 9110**, Section 15.4.5
305	`USE_PROXY`	HTTP Semantics **RFC 9110**, Section 15.4.6
307	`TEMPORARY_REDIRECT`	HTTP Semantics **RFC 9110**, Section 15.4.8

Code	Enum Name	Details
308	`PERMANENT_REDIRECT`	HTTP Semantics **RFC 9110**, Section 15.4.9
400	`BAD_REQUEST`	HTTP Semantics **RFC 9110**, Section 15.5.1
401	`UNAUTHORIZED`	HTTP Semantics **RFC 9110**, Section 15.5.2
402	`PAYMENT_REQUIRED`	HTTP Semantics **RFC 9110**, Section 15.5.3
403	`FORBIDDEN`	HTTP Semantics **RFC 9110**, Section 15.5.4
404	`NOT_FOUND`	HTTP Semantics **RFC 9110**, Section 15.5.5
405	`METHOD_NOT_ALLOWED`	HTTP Semantics **RFC 9110**, Section 15.5.6
406	`NOT_ACCEPTABLE`	HTTP Semantics **RFC 9110**, Section 15.5.7
407	`PROXY_AUTHENTICATION_REQUIRED`	HTTP Semantics **RFC 9110**, Section 15.5.8

Code	Enum Name	Details
408	`REQUEST_TIMEOUT`	HTTP Semantics **RFC 9110**, Section 15.5.9
409	`CONFLICT`	HTTP Semantics **RFC 9110**, Section 15.5.10
410	`GONE`	HTTP Semantics **RFC 9110**, Section 15.5.11
411	`LENGTH_REQUIRED`	HTTP Semantics **RFC 9110**, Section 15.5.12
412	`PRECONDITION_FAILED`	HTTP Semantics **RFC 9110**, Section 15.5.13
413	`CONTENT_TOO_LARGE`	HTTP Semantics **RFC 9110**, Section 15.5.14
414	`URI_TOO_LONG`	HTTP Semantics **RFC 9110**, Section 15.5.15

Code	Enum Name	Details
415	`UNSUPPORTED_MEDIA_TYPE`	HTTP Semantics **RFC 9110**, Section 15.5.16
416	`RANGE_NOT_SATISFIABLE`	HTTP Semantics **RFC 9110**, Section 15.5.17
417	`EXPECTATION_FAILED`	HTTP Semantics **RFC 9110**, Section 15.5.18
418	`IM_A_TEAPOT`	HTCPCP/1.0 **RFC 2324**, Section 2.3.2
421	`MISDIRECTED_REQUEST`	HTTP Semantics **RFC 9110**, Section 15.5.20
422	`UNPROCESSABLE_CONTENT`	HTTP Semantics **RFC 9110**, Section 15.5.21
423	`LOCKED`	WebDAV **RFC 4918**, Section 11.3

Code	Enum Name	Details
424	`FAILED_DEPENDENCY`	WebDAV **RFC 4918**, Section 11.4
425	`TOO_EARLY`	Using Early Data in HTTP **RFC 8470**
426	`UPGRADE_REQUIRED`	HTTP Semantics **RFC 9110**, Section 15.5.22
428	`PRECONDITION_REQUIRED`	Additional HTTP Status Codes **RFC 6585**
429	`TOO_MANY_REQUESTS`	Additional HTTP Status Codes **RFC 6585**
431	`REQUEST_HEADER_FIELDS_TOO_LARGE`	Additional HTTP Status Codes **RFC 6585**
451	`UNAVAILABLE_FOR_LEGAL_REASONS`	An HTTP Status Code to Report Legal Obstacles **RFC 7725**

Code	Enum Name	Details
500	`INTERNAL_SERVER_ERROR`	HTTP Semantics **RFC 9110**, Section 15.6.1
501	`NOT_IMPLEMENTED`	HTTP Semantics **RFC 9110**, Section 15.6.2
502	`BAD_GATEWAY`	HTTP Semantics **RFC 9110**, Section 15.6.3
503	`SERVICE_UNAVAILABLE`	HTTP Semantics **RFC 9110**, Section 15.6.4
504	`GATEWAY_TIMEOUT`	HTTP Semantics **RFC 9110**, Section 15.6.5
505	`HTTP_VERSION_NOT_SUPPORTED`	HTTP Semantics **RFC 9110**, Section 15.6.6
506	`VARIANT_ALSO_NEGOTIATES`	Transparent Content Negotiation in HTTP **RFC 2295**, Section 8.1 (Experimental)
507	`INSUFFICIENT_STORAGE`	WebDAV **RFC 4918**, Section 11.5

Code	Enum Name	Details
`508`	`LOOP_DETECTED`	WebDAV Binding Extensions **RFC 5842**, Section 7.2 (Experimental)
`510`	`NOT_EXTENDED`	An HTTP Extension Framework **RFC 2774**, Section 7 (Experimental)
`511`	`NETWORK_AUTHENTICATION_REQUIRED`	Additional HTTP Status Codes **RFC 6585**, Section 6

In order to preserve backwards compatibility, enum values are also present in the **`http.client`** module in the form of constants. The enum name is equal to the constant name (i.e. `http.HTTPStatus.OK` is also available as `http.client.OK`).

Changed in version 3.7: Added `421 MISDIRECTED_REQUEST` status code.

Added in version 3.8: Added `451 UNAVAILABLE_FOR_LEGAL_REASONS` status code.

Added in version 3.9: Added `103 EARLY_HINTS`, `418 IM_A_TEAPOT` and `425 TOO_EARLY` status codes.

Changed in version 3.13: Implemented RFC9110 naming for status constants. Old constant names are preserved for backwards compatibility: `413 REQUEST_ENTITY_TOO_LARGE`, `414 REQUEST_URI_TOO_LONG`, `416 REQUESTED_RANGE_NOT_SATISFIABLE` and `422 UNPROCESSABLE_ENTITY`.

HTTP status category

Added in version 3.12.

The enum values have several properties to indicate the HTTP status category:

Property	**Indicates that**	**Details**
`is_informational`	`100 <= status <= 199`	HTTP Semantics **RFC 9110**, Section 15
`is_success`	`200 <= status <= 299`	HTTP Semantics **RFC 9110**, Section 15
`is_redirection`	`300 <= status <= 399`	HTTP Semantics **RFC 9110**, Section 15
`is_client_error`	`400 <= status <= 499`	HTTP Semantics **RFC 9110**, Section 15
`is_server_error`	`500 <= status <= 599`	HTTP Semantics **RFC 9110**, Section 15

Usage:

```
>>> from http import HTTPStatus
>>> HTTPStatus.OK.is_success
True
>>> HTTPStatus.OK.is_client_error
False
```

class http.HTTPMethod

Added in version 3.11.

A subclass of [`enum.StrEnum`] that defines a set of HTTP methods and descriptions written in English.

Usage:

```
>>> from http import HTTPMethod
>>>
>>> HTTPMethod.GET
<HTTPMethod.GET>
>>> HTTPMethod.GET == 'GET'
True
>>> HTTPMethod.GET.value
'GET'
>>> HTTPMethod.GET.description
'Retrieve the target.'
>>> list(HTTPMethod)
[<HTTPMethod.CONNECT>,
 <HTTPMethod.DELETE>,
 <HTTPMethod.GET>,
 <HTTPMethod.HEAD>,
 <HTTPMethod.OPTIONS>,
 <HTTPMethod.PATCH>,
 <HTTPMethod.POST>,
 <HTTPMethod.PUT>,
 <HTTPMethod.TRACE>]
```

HTTP methods

Supported, IANA-registered methods [https://www.iana.org/assignments/http-methods/http-methods.xhtml] available in **`http.HTTPMethod`** are:

Method	Enum Name	Details
`GET`	`GET`	HTTP Semantics **RFC 9110**, Section 9.3.1
`HEAD`	`HEAD`	HTTP Semantics **RFC 9110**, Section 9.3.2
`POST`	`POST`	HTTP Semantics **RFC 9110**, Section 9.3.3
`PUT`	`PUT`	HTTP Semantics **RFC 9110**, Section 9.3.4
`DELETE`	`DELETE`	HTTP Semantics **RFC 9110**, Section 9.3.5
`CONNECT`	`CONNECT`	HTTP Semantics **RFC 9110**, Section 9.3.6
`OPTIONS`	`OPTIONS`	HTTP Semantics **RFC 9110**, Section 9.3.7
`TRACE`	`TRACE`	HTTP Semantics **RFC 9110**, Section 9.3.8
`PATCH`	`PATCH`	HTTP/1.1 **RFC 5789**

Method	Enum Name	Details

`http.client` — HTTP protocol client

Source code: Lib/http/client.py [https://github.com/python/cpython/tree/3.14/Lib/http/client.py]

This module defines classes that implement the client side of the HTTP and HTTPS protocols. It is normally not used directly — the module **`urllib.request`** uses it to handle URLs that use HTTP and HTTPS.

See also

The Requests package [https://requests.readthedocs.io/en/latest/] is recommended for a higher-level HTTP client interface.

Note

HTTPS support is only available if Python was compiled with SSL support (through the **`ssl`** module).

Availability: not WASI.

This module does not work or is not available on WebAssembly. See WebAssembly platforms for more information.

The module provides the following classes:

class http.client.HTTPConnection(*host*, *port=None*, [*timeout*,]*source_address=None*, *blocksize=8192*)

An `HTTPConnection` instance represents one transaction with an HTTP server. It should be instantiated by passing it a host and optional port number. If no port number is passed, the port is extracted from the host string if it has the form `host:port`, else the default HTTP port (80) is used. If the optional *timeout* parameter is given, blocking operations (like connection attempts) will timeout after that many seconds (if it is not given, the global default timeout setting is used). The optional *source_address* parameter may be a tuple of a (host, port) to use as the source address the HTTP connection is made from. The optional *blocksize* parameter sets the buffer size in bytes for sending a file-like message body.

For example, the following calls all create instances that connect to the server at the same host and port:

```
>>> h1 = http.client.HTTPConnection('www.python.org')
>>> h2 = http.client.HTTPConnection('www.python.org:80')
>>> h3 = http.client.HTTPConnection('www.python.org', 80)
>>> h4 = http.client.HTTPConnection('www.python.org', 80,
timeout=10)
```

Changed in version 3.2: source_address was added.

Changed in version 3.4: The *strict* parameter was removed. HTTP 0.9-style “Simple Responses” are no longer supported.

Changed in version 3.7: blocksize parameter was added.

class http.client.HTTPSConnection(*host*, *port=None*, ***, [*timeout*,]*source_address=None*, *context=None*, *blocksize=8192*)

A subclass of `HTTPConnection` that uses SSL for communication with secure servers. Default port is `443`. If *context* is specified, it must be a `ssl.SSLContext` instance describing the various SSL options.

Please read Security considerations for more information on best practices.

Changed in version 3.2: source_address, *context* and *check_hostname* were added.

Changed in version 3.2: This class now supports HTTPS virtual hosts if possible (that is, if `ssl.HAS_SNI` is true).

Changed in version 3.4: The *strict* parameter was removed. HTTP 0.9-style “Simple Responses” are no longer supported.

Changed in version 3.4.3: This class now performs all the necessary certificate and hostname checks by default. To revert to the previous, unverified, behavior **`ssl._create_unverified_context()`** can be passed to the *context* parameter.

Changed in version 3.8: This class now enables TLS 1.3 `ssl.SSLContext.post_handshake_auth` for the default *context* or when *cert_file* is passed with a custom *context*.

Changed in version 3.10: This class now sends an ALPN extension with protocol indicator `http/1.1` when no *context* is given. Custom *context* should set ALPN protocols with `set_alpn_protocols()`.

Changed in version 3.12: The deprecated *key_file*, *cert_file* and *check_hostname* parameters have been removed.

class http.client.HTTPResponse(*sock*, *debuglevel=0*, *method=None*, *url=None*)

Class whose instances are returned upon successful connection. Not instantiated directly by user.

Changed in version 3.4: The *strict* parameter was removed. HTTP 0.9 style “Simple Responses” are no longer supported.

This module provides the following function:

http.client.parse_headers(*fp*)

Parse the headers from a file pointer *fp* representing a HTTP request/response. The file has to be a `BufferedIOBase` reader (i.e. not text) and must provide a valid **RFC 5322** [https://datatracker.ietf.org/doc/html/rfc5322.html] style header.

This function returns an instance of `http.client.HTTPMessage` that holds the header fields, but no payload (the same as `HTTPResponse.msg` and `http.server.BaseHTTPRequestHandler.headers`). After returning, the file pointer *fp* is ready to read the HTTP body.

Note

`parse_headers()` does not parse the start-line of a HTTP message; it only parses the `Name: value` lines. The file has to be ready to read these field lines, so the first line should already be consumed before calling the function.

The following exceptions are raised as appropriate:

exception http.client.HTTPException

The base class of the other exceptions in this module. It is a subclass of `Exception`.

exception http.client.NotConnected

A subclass of `HTTPException`.

exception http.client.InvalidURL

A subclass of `HTTPException`, raised if a port is given and is either non-numeric or empty.

exception http.client.UnknownProtocol

A subclass of `HTTPException`.

exception http.client.UnknownTransferEncoding

A subclass of `HTTPException`.

exception http.client.UnimplementedFileMode

A subclass of `HTTPException`.

exception http.client.IncompleteRead

A subclass of `HTTPException`.

exception http.client.ImproperConnectionState

A subclass of `HTTPException`.

exception http.client.CannotSendRequest

A subclass of `ImproperConnectionState`.

exception http.client.CannotSendHeader

A subclass of `ImproperConnectionState`.

exception http.client.ResponseNotReady

A subclass of `ImproperConnectionState`.

exception http.client.BadStatusLine

A subclass of `HTTPException`. Raised if a server responds with a HTTP status code that we don’t understand.

exception http.client.LineTooLong

A subclass of `HTTPException`. Raised if an excessively long line is received in the HTTP protocol from the server.

exception http.client.RemoteDisconnected

A subclass of `ConnectionResetError` and `BadStatusLine`. Raised by `HTTPConnection.getresponse()` when the attempt to read the response results in no data read from the connection, indicating that the remote end has closed the connection.

Added in version 3.5: Previously, `BadStatusLine('')` was raised.

The constants defined in this module are:

http.client.HTTP_PORT

The default port for the HTTP protocol (always `80`).

http.client.HTTPS_PORT

The default port for the HTTPS protocol (always `443`).

http.client.responses

This dictionary maps the HTTP 1.1 status codes to the W3C names.

Example: `http.client.responses[http.client.NOT_FOUND]` is `'Not Found'`.

See HTTP status codes for a list of HTTP status codes that are available in this module as constants.

HTTPConnection Objects

`HTTPConnection` instances have the following methods:

HTTPConnection.request(*method*, *url*, *body=None*, *headers={}*, *, *encode_chunked=False*)

This will send a request to the server using the HTTP request method *method* and the request URI *url*. The provided *url* must be an absolute path to conform with **RFC 2616 §5.1.2** [https://datatracker.ietf.org/doc/html/rfc2616.html#section-5.1.2] (unless connecting to an HTTP proxy server or using the `OPTIONS` or `CONNECT` methods).

If *body* is specified, the specified data is sent after the headers are finished. It may be a **`str`**, a bytes-like object, an open file object, or an iterable of **`bytes`**. If *body* is a string, it is encoded as ISO-8859-1, the default for HTTP. If it is a bytes-like object, the bytes are sent as is. If it is a file object, the contents of the file is sent; this file object should support at least the `read()` method. If the file object is an instance of **`io.TextIOBase`**, the data returned by the `read()` method will be encoded as ISO-8859-1, otherwise the data returned by `read()` is sent as is. If *body* is an iterable, the elements of the iterable are sent as is until the iterable is exhausted.

The *headers* argument should be a mapping of extra HTTP headers to send with the request. A **Host header** [https://datatracker.ietf.org/doc/html/rfc2616.html#section-14.23] must be provided to conform with **RFC 2616 §5.1.2** [https://datatracker.ietf.org/doc/html/rfc2616.html#section-5.1.2] (unless connecting to an HTTP proxy server or using the `OPTIONS` or `CONNECT` methods).

If *headers* contains neither Content-Length nor Transfer-Encoding, but there is a request body, one of those header fields will be added automatically. If *body* is `None`, the Content-Length header is set to `0` for methods that expect a body (`PUT`, `POST`, and `PATCH`). If *body* is a string or a bytes-like object that is not also a file, the Content-Length header is set to its length. Any other type of *body* (files and iterables in general) will be chunk-encoded, and the Transfer-Encoding header will automatically be set instead of Content-Length.

The *encode_chunked* argument is only relevant if Transfer-Encoding is specified in *headers*. If *encode_chunked* is `False`, the HTTPConnection object assumes that all encoding is handled by the calling code. If it is `True`, the body will be chunk-encoded.

For example, to perform a `GET` request to `https://docs.python.org/3/`:

```
>>> import http.client
>>> host = "docs.python.org"
>>> conn = http.client.HTTPSConnection(host)
>>> conn.request("GET", "/3/", headers={"Host": host})
>>> response = conn.getresponse()
>>> print(response.status, response.reason)
200 OK
```

Note

Chunked transfer encoding has been added to the HTTP protocol version 1.1. Unless the HTTP server is known to handle HTTP 1.1, the caller must either specify the Content-Length, or must pass a `str` or bytes-like object that is not also a file as the body representation.

Note

Note that you must have read the whole response or call `close()` if `getresponse()` raised an non-`ConnectionError` exception before you can send a new request to the server.

Changed in version 3.2: body can now be an iterable.

Changed in version 3.6: If neither Content-Length nor Transfer-Encoding are set in *headers*, file and iterable *body* objects are now chunk-encoded. The *encode_chunked* argument was added. No attempt is made to determine the Content-Length for file objects.

HTTPConnection.getresponse()

Should be called after a request is sent to get the response from the server. Returns an `HTTPResponse` instance.

Changed in version 3.5: If a `ConnectionError` or subclass is raised, the `HTTPConnection` object will be ready to reconnect when a new request is sent.

Note that this does not apply to `OSError`s raised by the underlying socket. Instead the caller is responsible to call `close()` on the existing connection.

HTTPConnection.set_debuglevel(*level*)

Set the debugging level. The default debug level is `0`, meaning no debugging output is printed. Any value greater than `0` will cause all currently defined debug output to be printed to stdout. The `debuglevel` is passed to any new `HTTPResponse` objects that are created.

Added in version 3.1.

HTTPConnection.set_tunnel(*host*, *port=None*, *headers=None*)

Set the host and the port for HTTP Connect Tunnelling. This allows running the connection through a proxy server.

The *host* and *port* arguments specify the endpoint of the tunneled connection (i.e. the address included in the CONNECT request, *not* the address of the proxy server).

The *headers* argument should be a mapping of extra HTTP headers to send with the CONNECT request.

As HTTP/1.1 is used for HTTP CONNECT tunnelling request, as per the RFC [https://datatracker.ietf.org/doc/html/rfc7231#section-4.3.6], a HTTP `Host:` header must be provided, matching the authority-form of the request target provided as the destination for the CONNECT request. If a HTTP `Host:` header is not provided via the headers argument, one is generated and transmitted automatically.

For example, to tunnel through a HTTPS proxy server running locally on port 8080, we would pass the address of the proxy to the **`HTTPSConnection`** constructor, and the address of the host that we eventually want to reach to the **`set_tunnel()`** method:

```
>>> import http.client
>>> conn = http.client.HTTPSConnection("localhost", 8080)
>>> conn.set_tunnel("www.python.org")
>>> conn.request("HEAD","/index.html")
```

Added in version 3.2.

Changed in version 3.12: HTTP CONNECT tunnelling requests use protocol HTTP/1.1, upgraded from protocol HTTP/1.0. `Host:` HTTP headers are mandatory for HTTP/1.1, so one will be automatically generated and transmitted if not provided in the headers argument.

HTTPConnection.get_proxy_response_headers()

Returns a dictionary with the headers of the response received from the proxy server to the CONNECT request.

If the CONNECT request was not sent, the method returns `None`.

Added in version 3.12.

HTTPConnection.connect()

Connect to the server specified when the object was created. By default, this is called automatically when making a request if the client does not already have a connection.

Raises an auditing event `http.client.connect` with arguments `self`, `host`, `port`.

HTTPConnection.close()

Close the connection to the server.

HTTPConnection.blocksize

Buffer size in bytes for sending a file-like message body.

Added in version 3.7.

As an alternative to using the **`request()`** method described above, you can also send your request step by step, by using the four functions below.

HTTPConnection.putrequest(*method*, *url*, *skip_host=False*, *skip_accept_encoding=False*)

This should be the first call after the connection to the server has been made. It sends a line to the server consisting of the *method* string, the *url* string, and the HTTP version (`HTTP/1.1`). To disable automatic sending of `Host:` or `Accept-Encoding:` headers (for example to accept additional content encodings), specify *skip_host* or *skip_accept_encoding* with non-False values.

HTTPConnection.putheader(*header*, *argument*[, *...*])

Send an **RFC 822** [https://datatracker.ietf.org/doc/html/rfc822.html]-style header to the server. It sends a line to the server consisting of the header, a colon and a space, and the first argument. If more arguments are given, continuation lines are sent, each consisting of a tab and an argument.

HTTPConnection.endheaders(*message_body=None*, ***, *encode_chunked=False*)

Send a blank line to the server, signalling the end of the headers. The optional *message_body* argument can be used to pass a message body associated with the request.

If *encode_chunked* is `True`, the result of each iteration of *message_body* will be chunk-encoded as specified in **RFC 7230** [https://datatracker.ietf.org/doc/html/rfc7230.html], Section 3.3.1. How the data is encoded is dependent on the type of *message_body*. If *message_body* implements the buffer interface the encoding will result in a single chunk. If *message_body* is a **`collections.abc.Iterable`**, each iteration of *message_body* will result in a chunk. If *message_body* is a file object, each call to `.read()` will result in a chunk. The method automatically signals the end of the chunk-encoded data immediately after *message_body*.

Note

Due to the chunked encoding specification, empty chunks yielded by an iterator body will be ignored by the chunk-encoder. This is to avoid premature termination of the read of the request by the target server due to malformed encoding.

Changed in version 3.6: Added chunked encoding support and the *encode_chunked* parameter.

HTTPConnection.send(*data*)

Send data to the server. This should be used directly only after the **`endheaders()`** method has been called and before **`getresponse()`** is called.

Raises an auditing event `http.client.send` with arguments `self`, `data`.

HTTPResponse Objects

An **`HTTPResponse`** instance wraps the HTTP response from the server. It provides access to the request headers and the entity body. The response is an iterable object and can be used in a with statement.

Changed in version 3.5: The **`io.BufferedIOBase`** interface is now implemented and all of its reader operations are supported.

HTTPResponse.read([*amt*])

Reads and returns the response body, or up to the next *amt* bytes.

HTTPResponse.readinto(*b*)

Reads up to the next len(b) bytes of the response body into the buffer *b*. Returns the number of bytes read.

Added in version 3.3.

HTTPResponse.getheader(*name*, *default=None*)

Return the value of the header *name*, or *default* if there is no header matching *name*. If there is more than one header with the name *name*, return all of the values joined by ‘, ‘. If *default* is any iterable other than a single string, its elements are similarly returned joined by commas.

HTTPResponse.getheaders()

Return a list of (header, value) tuples.

HTTPResponse.fileno()

Return the `fileno` of the underlying socket.

HTTPResponse.msg

A **`http.client.HTTPMessage`** instance containing the response headers. **`http.client.HTTPMessage`** is a subclass of **`email.message.Message`**.

HTTPResponse.version

HTTP protocol version used by server. 10 for HTTP/1.0, 11 for HTTP/1.1.

HTTPResponse.url

URL of the resource retrieved, commonly used to determine if a redirect was followed.

HTTPResponse.headers

Headers of the response in the form of an **`email.message.EmailMessage`** instance.

HTTPResponse.status

Status code returned by server.

HTTPResponse.reason

Reason phrase returned by server.

HTTPResponse.debuglevel

A debugging hook. If `debuglevel` is greater than zero, messages will be printed to stdout as the response is read and parsed.

HTTPResponse.closed

Is `True` if the stream is closed.

HTTPResponse.geturl()

Deprecated since version 3.9: Deprecated in favor of `url`.

HTTPResponse.info()

Deprecated since version 3.9: Deprecated in favor of `headers`.

HTTPResponse.getcode()

Deprecated since version 3.9: Deprecated in favor of `status`.

Examples

Here is an example session that uses the `GET` method:

```
>>> import http.client
>>> conn = http.client.HTTPSConnection("www.python.org")
>>> conn.request("GET", "/")
>>> r1 = conn.getresponse()
>>> print(r1.status, r1.reason)
200 OK
>>> data1 = r1.read()  # This will return entire content.
>>> # The following example demonstrates reading data in
chunks.
>>> conn.request("GET", "/")
>>> r1 = conn.getresponse()
>>> while chunk := r1.read(200):
```

```
...     print(repr(chunk))
b'<!doctype html>\n<!--[if"...
...
>>> # Example of an invalid request
>>> conn = http.client.HTTPSConnection("docs.python.org")
>>> conn.request("GET", "/parrot.spam")
>>> r2 = conn.getresponse()
>>> print(r2.status, r2.reason)
404 Not Found
>>> data2 = r2.read()
>>> conn.close()
```

Here is an example session that uses the `HEAD` method. Note that the `HEAD` method never returns any data.

```
>>> import http.client
>>> conn = http.client.HTTPSConnection("www.python.org")
>>> conn.request("HEAD", "/")
>>> res = conn.getresponse()
>>> print(res.status, res.reason)
200 OK
>>> data = res.read()
>>> print(len(data))
0
>>> data == b''
True
```

Here is an example session that uses the `POST` method:

```
>>> import http.client, urllib.parse
>>> params = urllib.parse.urlencode({'@number': 12524, '@type':
'issue', '@action': 'show'})
>>> headers = {"Content-type": "application/x-www-form-
urlencoded",
...            "Accept": "text/plain"}
>>> conn = http.client.HTTPConnection("bugs.python.org")
>>> conn.request("POST", "", params, headers)
>>> response = conn.getresponse()
>>> print(response.status, response.reason)
302 Found
>>> data = response.read()
>>> data
```

```
b'Redirecting to <a
href="https://bugs.python.org/issue12524">https://bugs.python.o
rg/issue12524</a>'
>>> conn.close()
```

Client side HTTP `PUT` requests are very similar to `POST` requests. The difference lies only on the server side where HTTP servers will allow resources to be created via `PUT` requests. It should be noted that custom HTTP methods are also handled in **`urllib.request.Request`** by setting the appropriate method attribute. Here is an example session that uses the `PUT` method:

```
>>> # This creates an HTTP request
>>> # with the content of BODY as the enclosed representation
>>> # for the resource http://localhost:8080/file
...
>>> import http.client
>>> BODY = "***filecontents***"
>>> conn = http.client.HTTPConnection("localhost", 8080)
>>> conn.request("PUT", "/file", BODY)
>>> response = conn.getresponse()
>>> print(response.status, response.reason)
200, OK
```

HTTPMessage Objects

class http.client.HTTPMessage(*email.message.Message*)

An **`http.client.HTTPMessage`** instance holds the headers from an HTTP response. It is implemented using the **`email.message.Message`** class.

`ftplib` — FTP protocol client

Source code: Lib/ftplib.py
[https://github.com/python/cpython/tree/3.14/Lib/ftplib.py]

This module defines the class **`FTP`** and a few related items. The **`FTP`** class implements the client side of the FTP protocol. You can use this to write Python programs that perform a variety of automated FTP jobs, such as mirroring other FTP servers. It is also used by the module **`urllib.request`** to handle URLs that use FTP. For more information on FTP (File Transfer Protocol), see internet **RFC 959**
[https://datatracker.ietf.org/doc/html/rfc959.html].

The default encoding is UTF-8, following **RFC 2640**
[https://datatracker.ietf.org/doc/html/rfc2640.html].

Availability: not WASI.

This module does not work or is not available on WebAssembly. See WebAssembly platforms for more information.

Here’s a sample session using the **`ftplib`** module:

```
>>> from ftplib import FTP
>>> ftp = FTP('ftp.us.debian.org')  # connect to host, default
port
>>> ftp.login()                     # user anonymous, passwd
anonymous@
'230 Login successful.'
>>> ftp.cwd('debian')               # change into "debian"
directory
'250 Directory successfully changed.'
>>> ftp.retrlines('LIST')           # list directory contents
-rw-rw-r--    1 1176     1176         1063 Jun 15 10:18 README
```

```
...
drwxr-sr-x    5 1176     1176         4096 Dec 19  2000 pool
drwxr-sr-x    4 1176     1176         4096 Nov 17  2008 project
drwxr-xr-x    3 1176     1176         4096 Oct 10  2012 tools
'226 Directory send OK.'
>>> with open('README', 'wb') as fp:
>>>     ftp.retrbinary('RETR README', fp.write)
'226 Transfer complete.'
>>> ftp.quit()
'221 Goodbye.'
```

Reference

FTP objects

class ftplib.FTP(*host*='', *user*='', *passwd*='', *acct*='', *timeout=None*, *source_address=None*, *, *encoding*='*utf-8*')

Return a new instance of the `FTP` class.

Parameters::

- **host** (*str*) – The hostname to connect to. If given, `connect(host)` is implicitly called by the constructor.
- **user** (*str*) – The username to log in with (default: `'anonymous'`). If given, `login(host` `passwd, acct)` is implicitly called by the constructor.
- **passwd** (*str*) – The password to use when logging in. If not given, and if *passwd* is the empty string or `"-"`, a password will be automatically generated.
- **acct** (*str*) – Account information to be used for the `ACCT` FTP command. Few systems implement this. See RFC-959 [https://datatracker.ietf.org/doc/html/rfc959.html] fo more details.
- **timeout** (*float* | *None*) – A timeout in seconds for blocking operations like `connect()` (default: the global default timeout setting).
- **source_address** (*tuple* | *None*) – A 2-tuple `(host, port)` for the socket to bind to as its source address before connecting.
- **encoding** (*str*) – The encoding for directories and filenames (default: `'utf-8'`).

The `FTP` class supports the `with` statement, e.g.:

```
>>> from ftplib import FTP
>>> with FTP("ftp1.at.proftpd.org") as ftp:
...     ftp.login()
...     ftp.dir()
...
'230 Anonymous login ok, restrictions apply.'
dr-xr-xr-x   9 ftp      ftp           154 May  6 10:43 .
```

```
dr-xr-xr-x   9 ftp      ftp           154 May  6 10:43 ..
dr-xr-xr-x   5 ftp      ftp          4096 May  6 10:43 CentOS
dr-xr-xr-x   3 ftp      ftp            18 Jul 10  2008 Fedora
>>>
```

Changed in version 3.2: Support for the `with` statement was added.

Changed in version 3.3: *source_address* parameter was added.

Changed in version 3.9: If the *timeout* parameter is set to be zero, it will raise a `ValueError` to prevent the creation of a non-blocking socket. The *encoding* parameter was added, and the default was changed from Latin-1 to UTF-8 to follow **RFC 2640** [https://datatracker.ietf.org/doc/html/rfc2640.html].

Several **`FTP`** methods are available in two flavors: one for handling text files and another for binary files. The methods are named for the command which is used followed by `lines` for the text version or `binary` for the binary version.

`FTP` instances have the following methods:

set_debuglevel(*level*)

Set the instance's debugging level as an `int`. This controls the amount of debugging output printed. The debug levels are:

- `0` (default): No debug output.
- `1`: Produce a moderate amount of debug output, generally a single line per request.
- `2` or higher: Produce the maximum amount of debugging output, logging each line sent and received on the control connection.

connect(*host=''*, *port=0*, *timeout=None*, *source_address=None*)

Connect to the given host and port. This function should be called only once for each instance; it should not be called if a *host* argument was given when the **FTP** instance was created. All other **FTP** methods can only be called after a connection has successfully been made.

Parameters::

- **host** (*str*) – The host to connect to.
- **port** (*int*) – The TCP port to connect to (default: `21`, as specified by the FT protocol specification). It is rarely needed to specify a different port number.
- **timeout** (*float* | *None*) – A timeout in seconds for the connection attempt (default: the global default timeout setting).
- **source_address** (*tuple* | *None*) – A 2 tuple `(host, port)` for the socket to bind to as its source address before connecting.

Raises an auditing event `ftplib.connect` with arguments `self`, `host`, `port`.

Changed in version 3.3: source_address parameter was added.

getwelcome()

Return the welcome message sent by the server in reply to the initial connection. (This message sometimes contains disclaimers or help information that may be relevant to the user.)

login(*user='anonymous'*, *passwd=''*, *acct=''*)

Log on to the connected FTP server. This function should be called only once for each instance, after a connection has been established; it should not be called if the *host* and *user* arguments were given when the **`FTP`** instance was created. Most FTP commands are only allowed after the client has logged in.

Parameters::

- **user** (*str*) – The username to log in with (default: `'anonymous'`).
- **passwd** (*str*) – The password to use when logging in. If not given, and if *passwd* is the empty string or `"-"`, a password will be automatically generated.
- **acct** (*str*) – Account information to be used for the ACCT FTP command. Few systems implement this. See RFC-959 [https://datatracker.ietf.org/doc/html/rfc959.html] for more details.

abort()

Abort a file transfer that is in progress. Using this does not always work, but it’s worth a try.

sendcmd(*cmd*)

Send a simple command string to the server and return the response string.

Raises an auditing event `ftplib.sendcmd` with arguments `self`, `cmd`.

voidcmd(*cmd*)

Send a simple command string to the server and handle the response. Return the response string if the response code corresponds to success (codes in the range 200–299). Raise **`error_reply`** otherwise.

Raises an auditing event `ftplib.sendcmd` with arguments `self`, `cmd`.

retrbinary(*cmd*, *callback*, *blocksize=8192*, *rest=None*)

Retrieve a file in binary transfer mode.

Parameters::

- **cmd** (*str*) – An appropriate `RETR` command: `"RETR` *`filename`*`"`.
- **callback** (callable) – A single parameter callable that is called for each block of data received, with its single argument being the data as `bytes`.
- **blocksize** (*int*) – The maximum chunk size to read on the low-level `socket` object created to do the actual transfer. This also corresponds to the largest size of data that will be passed to *callback*. Defaults to `8192`.
- **rest** (*int*) – A `REST` command to be sent to the server. See the documentation for the *rest* paramete of the `transfercmd()` method.

retrlines(*cmd*, *callback=None*)

Retrieve a file or directory listing in the encoding specified by the *encoding* parameter at initialization. *cmd* should be an appropriate `RETR` command (see `retrbinary()`) or a command such as `LIST` or `NLST` (usually just the string `'LIST'`). `LIST` retrieves a list of files and information about those files. `NLST` retrieves a list of file names. The *callback* function is called for each line with a string argument containing the line with the trailing CRLF stripped. The default *callback* prints the line to `sys.stdout`.

set_pasv(*val*)

Enable “passive” mode if *val* is true, otherwise disable passive mode. Passive mode is on by default.

storbinary(*cmd*, *fp*, *blocksize=8192*, *callback=None*, *rest=None*)

Store a file in binary transfer mode.

Parameters::

- **cmd** (*str*) – An appropriate `STOR` command: `"STOR filename"`.
- **fp** (file object) – A file object (opened in binary mode) which is read until EOF, using its **`read()`** method in blocks of size *blocksize* to provide the data to be stored.
- **blocksize** (*int*) – The read block size. Defaults to `8192`.
- **callback** (callable) – A single parameter callable that is called for each block of data sent, with its single argument being the data as **`bytes`**.
- **rest** (*int*) – A `REST` command to be sent to the server. See the documentation for the *rest* paramete of the **`transfercmd()`** method.

Changed in version 3.2: The *rest* parameter was added.

storlines(*cmd*, *fp*, *callback=None*)

Store a file in line mode. *cmd* should be an appropriate `STOR` command (see **`storbinary()`**). Lines are read until EOF from the file object *fp* (opened in binary mode) using its **`readline()`** method to provide the data to be stored. *callback* is an optional single parameter callable that is called on each line after it is sent.

transfercmd(*cmd*, *rest=None*)

Initiate a transfer over the data connection. If the transfer is active, send an `EPRT` or `PORT` command and the transfer command specified by *cmd*, and accept the connection. If the server is passive, send an `EPSV` or `PASV` command, connect to it, and start the transfer command. Either way, return the socket for the connection.

If optional *rest* is given, a `REST` command is sent to the server, passing *rest* as an argument. *rest* is usually a byte offset into the requested file, telling the server to restart sending the file's bytes at the requested offset, skipping over the initial bytes. Note however that the `transfercmd()` method converts *rest* to a string with the *encoding* parameter specified at initialization, but no check is performed on the string's contents. If the server does not recognize the `REST` command, an `error_reply` exception will be raised. If this happens, simply call `transfercmd()` without a *rest* argument.

ntransfercmd(*cmd*, *rest=None*)

Like `transfercmd()`, but returns a tuple of the data connection and the expected size of the data. If the expected size could not be computed, `None` will be returned as the expected size. *cmd* and *rest* means the same thing as in `transfercmd()`.

mlsd(*path=''*, *facts=[]*)

List a directory in a standardized format by using `MLSD` command (**RFC 3659** [https://datatracker.ietf.org/doc/html/rfc3659.html]). If *path* is omitted the current directory is assumed. *facts* is a list of strings representing the type of information desired (e.g. `["type", "size", "perm"]`). Return a generator object yielding a tuple of two elements for every file found in path. First element is the file name, the second one is a dictionary containing facts about the file name. Content of this dictionary might be limited by the *facts* argument but server is not guaranteed to return all requested facts.

Added in version 3.3.

nlst(*argument*[, ...])

Return a list of file names as returned by the `NLST` command. The optional *argument* is a directory to list (default is the current server directory). Multiple arguments can be used to pass non-standard options to the `NLST` command.

Note

If your server supports the command, `mlsd()` offers a better API.

dir(*argument*[, ...])

Produce a directory listing as returned by the `LIST` command, printing it to standard output. The optional *argument* is a directory to list (default is the current server directory). Multiple arguments can be used to pass non-standard options to the `LIST` command. If the last argument is a function, it is used as a *callback* function as for `retrlines()`; the default prints to `sys.stdout`. This method returns `None`.

Note

If your server supports the command, `mlsd()` offers a better API.

rename(*fromname*, *toname*)

Rename file *fromname* on the server to *toname*.

delete(*filename*)

Remove the file named *filename* from the server. If successful, returns the text of the response, otherwise raises `error_perm` on permission errors or `error_reply` on other errors.

cwd(*pathname*)

Set the current directory on the server.

mkd(*pathname*)

Create a new directory on the server.

pwd()

Return the pathname of the current directory on the server.

rmd(*dirname*)

Remove the directory named *dirname* on the server.

size(*filename*)

Request the size of the file named *filename* on the server. On success, the size of the file is returned as an integer, otherwise `None` is returned. Note that the `SIZE` command is not standardized, but is supported by many common server implementations.

quit()

Send a `QUIT` command to the server and close the connection. This is the “polite” way to close a connection, but it may raise an exception if the server responds with an error to the `QUIT` command. This implies a call to the `close()` method which renders the `FTP` instance useless for subsequent calls (see below).

close()

Close the connection unilaterally. This should not be applied to an already closed connection such as after a successful call to `quit()`. After this call the `FTP` instance should not be used any more (after a call to `close()` or `quit()` you cannot reopen the connection by issuing another `login()` method).

FTP_TLS objects

class ftplib.FTP_TLS(*host=''*, *user=''*, *passwd=''*, *acct=''*, *, *context=None*, *timeout=None*, *source_address=None*, *encoding='utf-8'*)

An `FTP` subclass which adds TLS support to FTP as described in **RFC 4217** [https://datatracker.ietf.org/doc/html/rfc4217.html]. Connect to port 21 implicitly securing the FTP control connection before authenticating.

Note

The user must explicitly secure the data connection by calling the `prot_p()` method.

Parameters::

- **host** (*str*) – The hostname to connect to. If given, `connect(host)` is implicitly called by the constructor.
- **user** (*str*) – The username to log in with (default: `'anonymous'`). If given, `login(host passwd, acct)` is implicitly called by the constructor.
- **passwd** (*str*) – The password to use when logging in. If not given, and if *passwd* is the empty string or `"-"`, a password will be automatically generated.
- **acct** (*str*) – Account information to be used for the ACCT FTP command. Few systems implement this. See RFC-959 [https://datatracker.ietf.org/doc/html/rfc959.html] for more details.
- **context** (`ssl.SSLContext`) – An SSL context object which allows bundling SS configuration options, certificates and private keys into a single, potentially long-lived, structure. Please read Security considerations for best practices.
- **timeout** (*float* | *None*) – A timeout in seconds for blocking operations like `connect()` (default: the global default timeout setting).
- **source_address** (*tuple* | *None*) – A 2-tuple `(host, port)` for the socket to bind to as its source address before connecting.
- **encoding** (*str*) – The encoding for directories and filenames (default: `'utf-8'`).

Added in version 3.2.

Changed in version 3.3: Added the *source_address* parameter.

Changed in version 3.4: The class now supports hostname check with `ssl.SSLContext.check_hostname` and *Server Name Indication* (see `ssl.HAS_SNI`).

Changed in version 3.9: If the *timeout* parameter is set to be zero, it will raise a `ValueError` to prevent the creation of a non-blocking socket. The *encoding* parameter was added, and the default was changed from Latin-1 to UTF-8 to follow **RFC 2640** [https://datatracker.ietf.org/doc/html/rfc2640.html].

Changed in version 3.12: The deprecated *keyfile* and *certfile* parameters have been removed.

Here's a sample session using the `FTP_TLS` class:

```
>>> ftps = FTP_TLS('ftp.pureftpd.org')
>>> ftps.login()
'230 Anonymous user logged in'
>>> ftps.prot_p()
'200 Data protection level set to "private"'
>>> ftps.nlst()
['6jack', 'OpenBSD', 'antilink', 'blogbench', 'bsdcam',
'clockspeed', 'djbdns-jedi', 'docs', 'eaccelerator-jedi',
'favicon.ico', 'francotone', 'fugu', 'ignore', 'libpuzzle',
'metalog', 'minidentd', 'misc', 'mysql-udf-global-user-
variables', 'php-jenkins-hash', 'php-skein-hash', 'php-
webdav', 'phpaudit', 'phpbench', 'pincaster', 'ping',
'posto', 'pub', 'public', 'public_keys', 'pure-ftpd',
'qscan', 'qtc', 'sharedance', 'skycache', 'sound', 'tmp',
'ucarp']
```

FTP_TLS class inherits from `FTP`, defining these additional methods and attributes:

ssl_version

The SSL version to use (defaults to `ssl.PROTOCOL_SSLv23`).

auth()

Set up a secure control connection by using TLS or SSL, depending on what is specified in the `ssl_version` attribute.

Changed in version 3.4: The method now supports hostname check with `ssl.SSLContext.check_hostname` and *Server Name Indication* (see `ssl.HAS_SNI`).

ccc()

Revert control channel back to plaintext. This can be useful to take advantage of firewalls that know how to handle NAT with non-secure FTP without opening fixed ports.

Added in version 3.3.

prot_p()

Set up secure data connection.

prot_c()

Set up clear text data connection.

Module variables

exception ftplib.error_reply

Exception raised when an unexpected reply is received from the server.

exception ftplib.error_temp

Exception raised when an error code signifying a temporary error (response codes in the range 400–499) is received.

exception ftplib.error_perm

Exception raised when an error code signifying a permanent error (response codes in the range 500–599) is received.

exception ftplib.error_proto

Exception raised when a reply is received from the server that does not fit the response specifications of the File Transfer Protocol, i.e. begin with a digit in the range 1–5.

ftplib.all_errors

The set of all exceptions (as a tuple) that methods of **`FTP`** instances may raise as a result of problems with the FTP connection (as opposed to programming errors made by the caller). This set includes the four exceptions listed above as well as **`OSError`** and **`EOFError`**.

See also

Module `netrc`

Parser for the `.netrc` file format. The file `.netrc` is typically used by FTP clients to load user authentication information before prompting the user.

`poplib` — POP3 protocol client

Source code: Lib/poplib.py [https://github.com/python/cpython/tree/3.14/Lib/poplib.py]

This module defines a class, **`POP3`**, which encapsulates a connection to a POP3 server and implements the protocol as defined in **RFC 1939** [https://datatracker.ietf.org/doc/html/rfc1939.html]. The **`POP3`** class supports both the minimal and optional command sets from **RFC 1939** [https://datatracker.ietf.org/doc/html/rfc1939.html]. The **`POP3`** class also supports the STLS command introduced in **RFC 2595** [https://datatracker.ietf.org/doc/html/rfc2595.html] to enable encrypted communication on an already established connection.

Additionally, this module provides a class **`POP3_SSL`**, which provides support for connecting to POP3 servers that use SSL as an underlying protocol layer.

Note that POP3, though widely supported, is obsolescent. The implementation quality of POP3 servers varies widely, and too many are quite poor. If your mailserver supports IMAP, you would be better off using the **`imaplib.IMAP4`** class, as IMAP servers tend to be better implemented.

Availability: not WASI.

This module does not work or is not available on WebAssembly. See WebAssembly platforms for more information.

The `poplib` module provides two classes:

class poplib.POP3(*host*, *port=POP3_PORT*[, *timeout*])

This class implements the actual POP3 protocol. The connection is created when the instance is initialized. If *port* is omitted, the standard POP3 port (110) is used. The optional *timeout* parameter specifies a timeout in seconds for the connection attempt (if not specified, the global default timeout setting will be used).

Raises an auditing event `poplib.connect` with arguments `self`, `host`, `port`.

All commands will raise an auditing event `poplib.putline` with arguments `self` and `line`, where `line` is the bytes about to be sent to the remote host.

Changed in version 3.9: If the *timeout* parameter is set to be zero, it will raise a `ValueError` to prevent the creation of a non-blocking socket.

class poplib.POP3_SSL(*host*, *port=POP3_SSL_PORT*, ***, *timeout=None*, *context=None*)

This is a subclass of `POP3` that connects to the server over an SSL encrypted socket. If *port* is not specified, 995, the standard POP3-over-SSL port is used. *timeout* works as in the `POP3` constructor. *context* is an optional `ssl.SSLContext` object which allows bundling SSL configuration options, certificates and private keys into a single (potentially long-lived) structure. Please read Security considerations for best practices.

Raises an auditing event `poplib.connect` with arguments `self`, `host`, `port`.

All commands will raise an auditing event `poplib.putline` with arguments `self` and `line`, where `line` is the bytes about to be sent to the remote host.

Changed in version 3.2: context parameter added.

Changed in version 3.4: The class now supports hostname check with **`ssl.SSLContext.check_hostname`** and *Server Name Indication* (see **`ssl.HAS_SNI`**).

Changed in version 3.9: If the *timeout* parameter is set to be zero, it will raise a **`ValueError`** to prevent the creation of a non-blocking socket.

Changed in version 3.12: The deprecated *keyfile* and *certfile* parameters have been removed.

One exception is defined as an attribute of the **`poplib`** module:

exception poplib.error_proto

Exception raised on any errors from this module (errors from **`socket`** module are not caught). The reason for the exception is passed to the constructor as a string.

See also

Module `imaplib`

The standard Python IMAP module.

Frequently Asked Questions About Fetchmail

[http://www.catb.org/~esr/fetchmail/fetchmail-FAQ.html]

The FAQ for the **fetchmail** POP/IMAP client collects information on POP3 server variations and RFC noncompliance that may be useful if you need to write an application based on the POP protocol.

POP3 Objects

All POP3 commands are represented by methods of the same name, in lowercase; most return the response text sent by the server.

A `POP3` instance has the following methods:

POP3.set_debuglevel(*level*)

Set the instance’s debugging level. This controls the amount of debugging output printed. The default, `0`, produces no debugging output. A value of `1` produces a moderate amount of debugging output, generally a single line per request. A value of `2` or higher produces the maximum amount of debugging output, logging each line sent and received on the control connection.

POP3.getwelcome()

Returns the greeting string sent by the POP3 server.

POP3.capa()

Query the server’s capabilities as specified in **RFC 2449** [https://datatracker.ietf.org/doc/html/rfc2449.html]. Returns a dictionary in the form `{'name': ['param'...]}`.

Added in version 3.4.

POP3.user(*username*)

Send user command, response should indicate that a password is required.

POP3.pass_(*password*)

Send password, response includes message count and mailbox size. Note: the mailbox on the server is locked until `quit()` is called.

POP3.apop(*user*, *secret*)

Use the more secure APOP authentication to log into the POP3 server.

POP3.rpop(*user*)

Use RPOP authentication (similar to UNIX r-commands) to log into POP3 server.

POP3.stat()

Get mailbox status. The result is a tuple of 2 integers: `(message count, mailbox size)`.

POP3.list([*which*])

Request message list, result is in the form `(response, ['mesg_num octets', ...], octets)`. If *which* is set, it is the message to list.

POP3.retr(*which*)

Retrieve whole message number *which*, and set its seen flag. Result is in form `(response, ['line', ...], octets)`.

POP3.dele(*which*)

Flag message number *which* for deletion. On most servers deletions are not actually performed until QUIT (the major exception is Eudora QPOP, which deliberately violates the RFCs by doing pending deletes on any disconnect).

POP3.rset()

Remove any deletion marks for the mailbox.

POP3.noop()

Do nothing. Might be used as a keep-alive.

POP3.quit()

Signoff: commit changes, unlock mailbox, drop connection.

POP3.top(*which*, *howmuch*)

Retrieves the message header plus *howmuch* lines of the message after the header of message number *which*. Result is in form `(response, ['line', ...], octets)`.

The POP3 TOP command this method uses, unlike the RETR command, doesn't set the message's seen flag; unfortunately, TOP is poorly specified in the RFCs and is frequently broken in off-brand servers. Test this method by hand against the POP3 servers you will use before trusting it.

POP3.uidl(*which=None*)

Return message digest (unique id) list. If *which* is specified, result contains the unique id for that message in the form `'response mesgnum uid`, otherwise result is list `(response, ['mesgnum uid', ...], octets)`.

POP3.utf8()

Try to switch to UTF-8 mode. Returns the server response if successful, raises **`error_proto`** if not. Specified in **RFC 6856** [https://datatracker.ietf.org/doc/html/rfc6856.html].

Added in version 3.5.

POP3.stls(*context=None*)

Start a TLS session on the active connection as specified in **RFC 2595** [https://datatracker.ietf.org/doc/html/rfc2595.html]. This is only allowed before user authentication

context parameter is a **`ssl.SSLContext`** object which allows bundling SSL configuration options, certificates and private keys into a single (potentially long-lived) structure. Please read Security considerations for best practices.

This method supports hostname checking via **`ssl.SSLContext.check_hostname`** and *Server Name Indication* (see **`ssl.HAS_SNI`**).

Added in version 3.4.

Instances of **`POP3_SSL`** have no additional methods. The interface of this subclass is identical to its parent.

POP3 Example

Here is a minimal example (without error checking) that opens a mailbox and retrieves and prints all messages:

```
import getpass, poplib

M = poplib.POP3('localhost')
M.user(getpass.getuser())
M.pass_(getpass.getpass())
numMessages = len(M.list()[1])
for i in range(numMessages):
    for j in M.retr(i+1)[1]:
        print(j)
```

At the end of the module, there is a test section that contains a more extensive example of usage.

`imaplib` — IMAP4 protocol client

Source code: Lib/imaplib.py [https://github.com/python/cpython/tree/3.14/Lib/imaplib.py]

This module defines three classes, **`IMAP4`**, **`IMAP4_SSL`** and **`IMAP4_stream`**, which encapsulate a connection to an IMAP4 server and implement a large subset of the IMAP4rev1 client protocol as defined in **RFC 2060** [https://datatracker.ietf.org/doc/html/rfc2060.html]. It is backward compatible with IMAP4 (**RFC 1730** [https://datatracker.ietf.org/doc/html/rfc1730.html]) servers, but note that the `STATUS` command is not supported in IMAP4.

Availability: not WASI.

This module does not work or is not available on WebAssembly. See WebAssembly platforms for more information.

Three classes are provided by the **`imaplib`** module, **`IMAP4`** is the base class:

class imaplib.IMAP4(*host=''*, *port=IMAP4_PORT*, *timeout=None*)

This class implements the actual IMAP4 protocol. The connection is created and protocol version (IMAP4 or IMAP4rev1) is determined when the instance is initialized. If *host* is not specified, `''` (the local host) is used. If *port* is omitted, the standard IMAP4 port (143) is used. The optional *timeout* parameter specifies a timeout in seconds for the connection attempt. If timeout is not given or is `None`, the global default socket timeout is used.

The **`IMAP4`** class supports the **`with`** statement. When used like this, the IMAP4 `LOGOUT` command is issued automatically when the **`with`** statement exits. E.g.:

```
>>> from imaplib import IMAP4
>>> with IMAP4("domain.org") as M:
...     M.noop()
...
('OK', [b'Nothing Accomplished. d25if65hy903weo.87'])
```

Changed in version 3.5: Support for the **`with`** statement was added.

Changed in version 3.9: The optional *timeout* parameter was added.

Three exceptions are defined as attributes of the **`IMAP4`** class:

exception IMAP4.error

Exception raised on any errors. The reason for the exception is passed to the constructor as a string.

exception IMAP4.abort

IMAP4 server errors cause this exception to be raised. This is a sub-class of **`IMAP4.error`**. Note that closing the instance and instantiating a new one will usually allow recovery from this exception.

exception IMAP4.readonly

This exception is raised when a writable mailbox has its status changed by the server. This is a sub-class of **`IMAP4.error`**. Some other client now has write permission, and the mailbox will need to be re-opened to re-obtain write permission.

There’s also a subclass for secure connections:

class imaplib.IMAP4_SSL(*host=''*, *port=IMAP4_SSL_PORT*, ***, *ssl_context=None*, *timeout=None*)

This is a subclass derived from **`IMAP4`** that connects over an SSL encrypted socket (to use this class you need a socket module that was compiled with SSL support). If *host* is not specified, `''` (the local host) is used. If *port* is omitted, the standard IMAP4-over-SSL port (993) is used. *ssl_context* is a **`ssl.SSLContext`** object which allows bundling SSL configuration options, certificates and private keys into a single (potentially long-lived) structure. Please read Security considerations for best practices.

The optional *timeout* parameter specifies a timeout in seconds for the connection attempt. If timeout is not given or is `None`, the global default socket timeout is used.

Changed in version 3.3: ssl_context parameter was added.

Changed in version 3.4: The class now supports hostname check with **`ssl.SSLContext.check_hostname`** and *Server Name Indication* (see **`ssl.HAS_SNI`**).

Changed in version 3.9: The optional *timeout* parameter was added.

Changed in version 3.12: The deprecated *keyfile* and *certfile* parameters have been removed.

The second subclass allows for connections created by a child process:

class imaplib.IMAP4_stream(*command*)

This is a subclass derived from `IMAP4` that connects to the `stdin/stdout` file descriptors created by passing *command* to `subprocess.Popen()`.

The following utility functions are defined:

imaplib.Internaldate2tuple(*datestr*)

Parse an IMAP4 `INTERNALDATE` string and return corresponding local time. The return value is a `time.struct_time` tuple or `None` if the string has wrong format.

imaplib.Int2AP(*num*)

Converts an integer into a bytes representation using characters from the set [`A` .. `P`].

imaplib.ParseFlags(*flagstr*)

Converts an IMAP4 `FLAGS` response to a tuple of individual flags.

imaplib.Time2Internaldate(*date_time*)

Convert *date_time* to an IMAP4 `INTERNALDATE` representation. The return value is a string in the form: `"DD-Mmm-YYYY HH:MM:SS +HHMM"` (including double-quotes). The *date_time* argument can be a number (int or float) representing seconds since epoch (as returned by `time.time()`), a 9-tuple representing local time an instance of `time.struct_time` (as returned by `time.localtime()`), an aware instance of `datetime.datetime`, or a double-quoted string. In the last case, it is assumed to already be in the correct format.

Note that IMAP4 message numbers change as the mailbox changes; in particular, after an `EXPUNGE` command performs deletions the remaining messages are renumbered. So it is highly advisable to use UIDs instead, with the UID command.

At the end of the module, there is a test section that contains a more extensive example of usage.

See also

Documents describing the protocol, sources for servers implementing it, by the University of Washington's IMAP Information Center can all be found at (**Source Code**) https://github.com/uw-imap/imap (**Not Maintained**).

IMAP4 Objects

All IMAP4rev1 commands are represented by methods of the same name, either uppercase or lowercase.

All arguments to commands are converted to strings, except for `AUTHENTICATE`, and the last argument to `APPEND` which is passed as an IMAP4 literal. If necessary (the string contains IMAP4 protocol-sensitive characters and isn't enclosed with either parentheses or double quotes) each string is quoted. However, the *password* argument to the `LOGIN` command is always quoted. If you want to avoid having an argument string quoted (eg: the *flags* argument to `STORE`) then enclose the string in parentheses (eg: `r'(\Deleted)'`).

Most commands return a tuple: `(type, [data, ...])` where *type* is usually `'OK'` or `'NO'`, and *data* is either the text from the command response, or mandated results from the command. Each *data* is either a `bytes`, or a tuple. If a tuple, then the first part is the header of the response, and the second part contains the data (ie: 'literal' value).

The *message_set* options to commands below is a string specifying one or more messages to be acted upon. It may be a simple message number (`'1'`), a range of message numbers (`'2:4'`), or a group of non-contiguous ranges separated by commas (`'1:3,6:9'`). A range can contain an asterisk to indicate an infinite upper bound (`'3:*'`).

An **`IMAP4`** instance has the following methods:

IMAP4.append(*mailbox*, *flags*, *date_time*, *message*)

Append *message* to named mailbox.

IMAP4.authenticate(*mechanism*, *authobject*)

Authenticate command — requires response processing.

mechanism specifies which authentication mechanism is to be used - it should appear in the instance variable `capabilities` in the form `AUTH=mechanism`.

authobject must be a callable object:

```
data = authobject(response)
```

It will be called to process server continuation responses; the *response* argument it is passed will be `bytes`. It should return `bytes` *data* that will be base64 encoded and sent to the server. It should return `None` if the client abort response `*` should be sent instead.

Changed in version 3.5: string usernames and passwords are now encoded to `utf-8` instead of being limited to ASCII.

IMAP4.check()

Checkpoint mailbox on server.

IMAP4.close()

Close currently selected mailbox. Deleted messages are removed from writable mailbox. This is the recommended command before `LOGOUT`.

IMAP4.copy(*message_set*, *new_mailbox*)

Copy *message_set* messages onto end of *new_mailbox*.

IMAP4.create(*mailbox*)

Create new mailbox named *mailbox*.

IMAP4.delete(*mailbox*)

Delete old mailbox named *mailbox*.

IMAP4.deleteacl(*mailbox*, *who*)

Delete the ACLs (remove any rights) set for who on mailbox.

IMAP4.enable(*capability*)

Enable *capability* (see **RFC 5161** [https://datatracker.ietf.org/doc/html/rfc5161.html]). Most capabilities do not need to be enabled. Currently only the `UTF8=ACCEPT` capability is supported (see **RFC 6855** [https://datatracker.ietf.org/doc/html/rfc6855.html]).

Added in version 3.5: The `enable()` method itself, and **RFC 6855** [https://datatracker.ietf.org/doc/html/rfc6855.html] support.

IMAP4.expunge()

Permanently remove deleted items from selected mailbox. Generates an `EXPUNGE` response for each deleted message. Returned data contains a list of `EXPUNGE` message numbers in order received.

IMAP4.fetch(*message_set*, *message_parts*)

Fetch (parts of) messages. *message_parts* should be a string of message part names enclosed within parentheses, eg: `"(UID BODY[TEXT])"`. Returned data are tuples of message part envelope and data.

IMAP4.getacl(*mailbox*)

Get the `ACL`s for *mailbox*. The method is non-standard, but is supported by the `Cyrus` server.

IMAP4.getannotation(*mailbox*, *entry*, *attribute*)

Retrieve the specified `ANNOTATION`s for *mailbox*. The method is non-standard, but is supported by the `Cyrus` server.

IMAP4.getquota(*root*)

Get the `quota` *root*'s resource usage and limits. This method is part of the IMAP4 QUOTA extension defined in rfc2087.

IMAP4.getquotaroot(*mailbox*)

Get the list of `quota` `roots` for the named *mailbox*. This method is part of the IMAP4 QUOTA extension defined in rfc2087.

IMAP4.idle(*duration=None*)

Return an **`Idler`**: an iterable context manager implementing the IMAP4 `IDLE` command as defined in **RFC 2177** [https://datatracker.ietf.org/doc/html/rfc2177.html].

The returned object sends the `IDLE` command when activated by the **`with`** statement, produces IMAP untagged responses via the iterator protocol, and sends `DONE` upon context exit.

All untagged responses that arrive after sending the `IDLE` command (including any that arrive before the server acknowledges the command) will be available via iteration. Any leftover responses (those not iterated in the **`with`** context) can be retrieved in the usual way after `IDLE` ends, using **`IMAP4.response()`**.

Responses are represented as `(type, [data, ...])` tuples, as described in IMAP4 Objects.

The *duration* argument sets a maximum duration (in seconds) to keep idling, after which any ongoing iteration will stop. It can be an **`int`** or **`float`**, or `None` for no time limit. Callers wishing to avoid inactivity timeouts on servers that impose them should keep this at most 29 minutes (1740 seconds). Requires a socket connection; *duration* must be `None` on **`IMAP4_stream`** connections.

```
>>> with M.idle(duration=29 * 60) as idler:
...     for typ, data in idler:
...         print(typ, data)
...
EXISTS [b'1']
RECENT [b'1']
```

Idler.burst(*interval=0.1*)

Yield a burst of responses no more than *interval* seconds apart (expressed as an **`int`** or **`float`**).

This generator is an alternative to iterating one response at a time, intended to aid in efficient batch processing. It retrieves the next response along with any immediately available subsequent responses. (For example, a rapid series of `EXPUNGE` responses after a bulk delete.)

Requires a socket connection; does not work on `IMAP4_stream` connections.

```
>>> with M.idle() as idler:
...     # get a response and any others following by <
0.1 seconds
...     batch = list(idler.burst())
...     print(f'processing {len(batch)} responses...')
...     print(batch)
...
processing 3 responses...
[('EXPUNGE', [b'2']), ('EXPUNGE', [b'1']), ('RECENT',
[b'0'])]
```

Tip

The IDLE context's maximum duration, as passed to `IMAP4.idle()`, is respected when waiting for the first response in a burst. Therefore, an expired **`Idler`** will cause this generator to return immediately without producing anything. Callers should consider this if using it in a loop.

Note

The iterator returned by `IMAP4.idle()` is usable only within a `with` statement. Before or after that context, unsolicited responses are collected internally whenever a command finishes, and can be retrieved with `IMAP4.response()`.

Note

The **`Idler`** class name and structure are internal interfaces, subject to change. Calling code can rely on its context management, iteration, and public method to remain stable, but should not subclass, instantiate, compare, or otherwise directly reference the class.

Added in version 3.14.

IMAP4.list([*directory*[, *pattern*]])

List mailbox names in *directory* matching *pattern*. *directory* defaults to the top-level mail folder, and *pattern* defaults to match anything. Returned data contains a list of `LIST` responses.

IMAP4.login(*user*, *password*)

Identify the client using a plaintext password. The *password* will be quoted.

IMAP4.login_cram_md5(*user*, *password*)

Force use of `CRAM-MD5` authentication when identifying the client to protect the password. Will only work if the server `CAPABILITY` response includes the phrase `AUTH=CRAM-MD5`.

Changed in version 3.14: An **`IMAP4.error`** is raised if MD5 support is not available.

IMAP4.logout()

Shutdown connection to server. Returns server `BYE` response.

Changed in version 3.8: The method no longer ignores silently arbitrary exceptions.

IMAP4.lsub(*directory='""'*, *pattern='*'*)

List subscribed mailbox names in directory matching pattern. *directory* defaults to the top level directory and *pattern* defaults to match any mailbox. Returned data are tuples of message part envelope and data.

IMAP4.myrights(*mailbox*)

Show my ACLs for a mailbox (i.e. the rights that I have on mailbox).

IMAP4.namespace()

Returns IMAP namespaces as defined in **RFC 2342** [https://datatracker.ietf.org/doc/html/rfc2342.html].

IMAP4.noop()

Send `NOOP` to server.

IMAP4.open(*host*, *port*, *timeout=None*)

Opens socket to *port* at *host*. The optional *timeout* parameter specifies a timeout in seconds for the connection attempt. If timeout is not given or is `None`, the global default socket timeout is used. Also note that if the *timeout* parameter is set to be zero, it will raise a **`ValueError`** to reject creating a non-blocking socket. This method is implicitly called by the **`IMAP4`** constructor. The connection objects established by this method will be used in the **`IMAP4.read()`**, **`IMAP4.readline()`**, **`IMAP4.send()`**, and **`IMAP4.shutdown()`** methods. You may override this method.

Raises an auditing event `imaplib.open` with arguments `self`, `host`, `port`.

Changed in version 3.9: The *timeout* parameter was added.

IMAP4.partial(*message_num*, *message_part*, *start*, *length*)

Fetch truncated part of a message. Returned data is a tuple of message part envelope and data.

IMAP4.proxyauth(*user*)

Assume authentication as *user*. Allows an authorised administrator to proxy into any user’s mailbox.

IMAP4.read(*size*)

Reads *size* bytes from the remote server. You may override this method.

IMAP4.readline()

Reads one line from the remote server. You may override this method.

IMAP4.recent()

Prompt server for an update. Returned data is `None` if no new messages, else value of `RECENT` response.

IMAP4.rename(*oldmailbox*, *newmailbox*)

Rename mailbox named *oldmailbox* to *newmailbox*.

IMAP4.response(*code*)

Return data for response *code* if received, or `None`. Returns the given code, instead of the usual type.

IMAP4.search(*charset*, *criterion*[, *...*])

Search mailbox for matching messages. *charset* may be `None`, in which case no `CHARSET` will be specified in the request to the server. The IMAP protocol requires that at least one criterion be specified; an exception will be raised when the server returns an error. *charset* must be `None` if the `UTF8=ACCEPT` capability was enabled using the `enable()` command.

Example:

```
# M is a connected IMAP4 instance...
typ, msgnums = M.search(None, 'FROM', '"LDJ"')
```

```
# or:
typ, msgnums = M.search(None, '(FROM "LDJ")')
```

IMAP4.select(*mailbox='INBOX'*, *readonly=False*)

Select a mailbox. Returned data is the count of messages in *mailbox* (`EXISTS` response). The default *mailbox* is `'INBOX'`. If the *readonly* flag is set, modifications to the mailbox are not allowed.

IMAP4.send(*data*)

Sends `data` to the remote server. You may override this method.

Raises an auditing event `imaplib.send` with arguments `self`, `data`.

IMAP4.setacl(*mailbox*, *who*, *what*)

Set an `ACL` for *mailbox*. The method is non-standard, but is supported by the `Cyrus` server.

IMAP4.setannotation(*mailbox*, *entry*, *attribute*[, *...*])

Set `ANNOTATIONS` for *mailbox*. The method is non-standard, but is supported by the `Cyrus` server.

IMAP4.setquota(*root*, *limits*)

Set the `quota` *root*’s resource *limits*. This method is part of the IMAP4 QUOTA extension defined in rfc2087.

IMAP4.shutdown()

Close connection established in `open`. This method is implicitly called by **`IMAP4.logout()`**. You may override this method.

IMAP4.socket()

Returns socket instance used to connect to server.

IMAP4.sort(*sort_criteria*, *charset*, *search_criterion*[, ...])

The `sort` command is a variant of `search` with sorting semantics for the results. Returned data contains a space separated list of matching message numbers.

Sort has two arguments before the *search_criterion* argument(s); a parenthesized list of *sort_criteria*, and the searching *charset*. Note that unlike `search`, the searching *charset* argument is mandatory. There is also a `uid sort` command which corresponds to `sort` the way that `uid search` corresponds to `search`. The `sort` command first searches the mailbox for messages that match the given searching criteria using the charset argument for the interpretation of strings in the searching criteria. It then returns the numbers of matching messages.

This is an `IMAP4rev1` extension command.

IMAP4.starttls(*ssl_context=None*)

Send a STARTTLS command. The *ssl_context* argument is optional and should be a **`ssl.SSLContext`** object. This will enable encryption on the IMAP connection. Please read Security considerations for best practices.

Added in version 3.2.

Changed in version 3.4: The method now supports hostname check with **`ssl.SSLContext.check_hostname`** and *Server Name Indication* (see **`ssl.HAS_SNI`**).

IMAP4.status(*mailbox*, *names*)

Request named status conditions for *mailbox*.

IMAP4.store(*message_set*, *command*, *flag_list*)

Alters flag dispositions for messages in mailbox. *command* is specified by section 6.4.6 of **RFC 2060** [https://datatracker.ietf.org/doc/html/rfc2060.html] as being one of "FLAGS", "+FLAGS", or "-FLAGS", optionally with a suffix of ".SILENT".

For example, to set the delete flag on all messages:

```
typ, data = M.search(None, 'ALL')
for num in data[0].split():
   M.store(num, '+FLAGS', '\\Deleted')
M.expunge()
```

Note

Creating flags containing ']' (for example: "[test]") violates **RFC 3501** [https://datatracker.ietf.org/doc/html/rfc3501.html] (the IMAP protocol). However, imaplib has historically allowed creation of such tags, and popular IMAP servers, such as Gmail, accept and produce such flags. There are non-Python programs which also create such tags. Although it is an RFC violation and IMAP clients and servers are supposed to be strict, imaplib still continues to allow such tags to be created for backward compatibility reasons, and as of Python 3.6, handles them if they are sent from the server, since this improves real-world compatibility.

IMAP4.subscribe(*mailbox*)

Subscribe to new mailbox.

IMAP4.thread(*threading_algorithm*, *charset*, *search_criterion*[, ...])

The `thread` command is a variant of `search` with threading semantics for the results. Returned data contains a space separated list of thread members.

Thread members consist of zero or more messages numbers, delimited by spaces, indicating successive parent and child.

Thread has two arguments before the *search_criterion* argument(s); a *threading_algorithm*, and the searching *charset*. Note that unlike `search`, the searching *charset* argument is mandatory. There is also a `uid thread` command which corresponds to `thread` the way that `uid search` corresponds to `search`. The `thread` command first searches the mailbox for messages that match the given searching criteria using the *charset* argument for the interpretation of strings in the searching criteria. It then returns the matching messages threaded according to the specified threading algorithm.

This is an `IMAP4rev1` extension command.

IMAP4.uid(*command*, *arg*[, *...*])

Execute command args with messages identified by UID, rather than message number. Returns response appropriate to command. At least one argument must be supplied; if none are provided, the server will return an error and an exception will be raised.

IMAP4.unsubscribe(*mailbox*)

Unsubscribe from old mailbox.

IMAP4.unselect()

`imaplib.IMAP4.unselect()` frees server’s resources associated with the selected mailbox and returns the server to the authenticated state. This command performs the same actions as `imaplib.IMAP4.close()`, except that no messages are permanently removed from the currently selected mailbox.

Added in version 3.9.

IMAP4.xatom(*name*[, *...*])

Allow simple extension commands notified by server in `CAPABILITY` response.

The following attributes are defined on instances of **`IMAP4`**:

IMAP4.PROTOCOL_VERSION

The most recent supported protocol in the `CAPABILITY` response from the server.

IMAP4.debug

Integer value to control debugging output. The initialize value is taken from the module variable `Debug`. Values greater than three trace each command.

IMAP4.utf8_enabled

Boolean value that is normally `False`, but is set to `True` if an **`enable()`** command is successfully issued for the `UTF8=ACCEPT` capability.

Added in version 3.5.

IMAP4 Example

Here is a minimal example (without error checking) that opens a mailbox and retrieves and prints all messages:

```
import getpass, imaplib

M = imaplib.IMAP4(host='example.org')
M.login(getpass.getuser(), getpass.getpass())
```

```
M.select()
typ, data = M.search(None, 'ALL')
for num in data[0].split():
    typ, data = M.fetch(num, '(RFC822)')
    print('Message %s\n%s\n' % (num, data[0][1]))
M.close()
M.logout()
```

`smtplib` — SMTP protocol client

Source code: Lib/smtplib.py [https://github.com/python/cpython/tree/3.14/Lib/smtplib.py]

The **`smtplib`** module defines an SMTP client session object that can be used to send mail to any internet machine with an SMTP or ESMTP listener daemon. For details of SMTP and ESMTP operation, consult **RFC 821** [https://datatracker.ietf.org/doc/html/rfc821.html] (Simple Mail Transfer Protocol) and **RFC 1869** [https://datatracker.ietf.org/doc/html/rfc1869.html] (SMTP Service Extensions).

Availability: not WASI.

This module does not work or is not available on WebAssembly. See WebAssembly platforms for more information.

class smtplib.SMTP(*host=''*, *port=0*, *local_hostname=None*, [*timeout*,]*source_address=None*)

An **`SMTP`** instance encapsulates an SMTP connection. It has methods that support a full repertoire of SMTP and ESMTP operations. If the optional *host* and *port* parameters are given, the SMTP **`connect()`** method is called with those parameters during initialization. If specified, *local_hostname* is used as the FQDN of the local host in the HELO/EHLO command. Otherwise, the local hostname is found using **`socket.getfqdn()`**. If the `connect()` call returns anything other than a success code, an `SMTPConnectError` is raised. The optional *timeout* parameter specifies a timeout in seconds for blocking operations like the connection attempt (if not specified, the global default timeout setting will be used). If the timeout expires, `TimeoutError` is raised. The optional *source_address* parameter allows binding to some specific source address in a machine with multiple network interfaces, and/or to some specific source TCP port. It takes a 2-tuple `(host, port)`, for the socket to bind to as its source address before connecting. If omitted (or if *host* or *port* are `''` and/or `0` respectively) the OS default behavior will be used.

For normal use, you should only require the initialization/connect, `sendmail()`, and `SMTP.quit()` methods. An example is included below.

The `SMTP` class supports the `with` statement. When used like this, the SMTP `QUIT` command is issued automatically when the **`with`** statement exits. E.g.:

```
>>> from smtplib import SMTP
>>> with SMTP("domain.org") as smtp:
...     smtp.noop()
...
(250, b'Ok')
>>>
```

All commands will raise an auditing event `smtplib.SMTP.send` with arguments `self` and `data`, where `data` is the bytes about to be sent to the remote host.

Changed in version 3.3: Support for the `with` statement was added.

Changed in version 3.3: *source_address* argument was added.

Added in version 3.5: The SMTPUTF8 extension (**RFC 6531** [https://datatracker.ietf.org/doc/html/rfc6531.html]) is now supported.

Changed in version 3.9: If the *timeout* parameter is set to be zero, it will raise a `ValueError` to prevent the creation of a non-blocking socket.

class smtplib.SMTP_SSL(*host=''*, *port=0*, *local_hostname=None*, *, [*timeout*,]*context=None*, *source_address=None*)

An `SMTP_SSL` instance behaves exactly the same as instances of `SMTP`. `SMTP_SSL` should be used for situations where SSL is required from the beginning of the connection and using `starttls()` is not appropriate. If *host* is not specified, the local host is used. If *port* is zero, the standard SMTP-over-SSL port (465) is used. The optional arguments *local_hostname*, *timeout* and *source_address* have the same meaning as they do in the `SMTP` class. *context*, also optional, can contain a `SSLContext` and allows configuring various aspects of the secure connection. Please read Security considerations for best practices.

Changed in version 3.3: context was added.

Changed in version 3.3: The *source_address* argument was added.

Changed in version 3.4: The class now supports hostname check with `ssl.SSLContext.check_hostname` and *Server Name Indication* (see `ssl.HAS_SNI`).

Changed in version 3.9: If the *timeout* parameter is set to be zero, it will raise a `ValueError` to prevent the creation of a non-blocking socket

Changed in version 3.12: The deprecated *keyfile* and *certfile* parameters have been removed.

class smtplib.LMTP(*host=''*, *port=LMTP_PORT*, *local_hostname=None*, *source_address=None*[, *timeout*])

The LMTP protocol, which is very similar to ESMTP, is heavily based on the standard SMTP client. It’s common to use Unix sockets for LMTP, so our `connect()` method must support that as well as a regular host:port server. The optional arguments *local_hostname* and *source_address* have the same meaning as they do in the `SMTP` class. To specify a Unix socket, you must use an absolute path for *host*, starting with a ‘/’.

Authentication is supported, using the regular SMTP mechanism. When using a Unix socket, LMTP generally don’t support or require any authentication, but your mileage might vary.

Changed in version 3.9: The optional *timeout* parameter was added.

A nice selection of exceptions is defined as well:

exception smtplib.SMTPException

Subclass of `OSError` that is the base exception class for all the other exceptions provided by this module.

Changed in version 3.4: SMTPException became subclass of `OSError`

exception smtplib.SMTPServerDisconnected

This exception is raised when the server unexpectedly disconnects, or when an attempt is made to use the `SMTP` instance before connecting it to a server.

exception smtplib.SMTPResponseException

Base class for all exceptions that include an SMTP error code. These exceptions are generated in some instances when the SMTP server returns an error code.

smtp_code

The error code.

smtp_error

The error message.

exception smtplib.SMTPSenderRefused

Sender address refused. In addition to the attributes set by on all **`SMTPResponseException`** exceptions, this sets 'sender' to the string that the SMTP server refused.

exception smtplib.SMTPRecipientsRefused

All recipient addresses refused.

recipients

A dictionary of exactly the same sort as returned by **`SMTP.sendmail()`** containing the errors for each recipient.

exception smtplib.SMTPDataError

The SMTP server refused to accept the message data.

exception smtplib.SMTPConnectError

Error occurred during establishment of a connection with the server.

exception smtplib.SMTPHeloError

The server refused our `HELO` message.

exception smtplib.SMTPNotSupportedError

The command or option attempted is not supported by the server.

Added in version 3.5.

exception smtplib.SMTPAuthenticationError

SMTP authentication went wrong. Most probably the server didn’t accept the username/password combination provided.

See also

RFC 821 [https://datatracker.ietf.org/doc/html/rfc821.html] - Simple Mail Transfer Protocol

Protocol definition for SMTP. This document covers the model, operating procedure, and protocol details for SMTP.

RFC 1869 [https://datatracker.ietf.org/doc/html/rfc1869.html] - SMTP Service Extensions

Definition of the ESMTP extensions for SMTP. This describes a framework for extending SMTP with new commands, supporting dynamic discovery of the commands provided by the server, and defines a few additional commands.

SMTP Objects

An `SMTP` instance has the following methods:

SMTP.set_debuglevel(*level*)

Set the debug output level. A value of 1 or `True` for *level* results in debug messages for connection and for all messages sent to and received from the server. A value of 2 for *level* results in these messages being timestamped.

Changed in version 3.5: Added debuglevel 2.

SMTP.docmd(*cmd*, *args=''*)

Send a command *cmd* to the server. The optional argument *args* is simply concatenated to the command, separated by a space.

This returns a 2-tuple composed of a numeric response code and the actual response line (multiline responses are joined into one long line.)

In normal operation it should not be necessary to call this method explicitly. It is used to implement other methods and may be useful for testing private extensions.

If the connection to the server is lost while waiting for the reply, **`SMTPServerDisconnected`** will be raised.

SMTP.connect(*host='localhost'*, *port=0*)

Connect to a host on a given port. The defaults are to connect to the local host at the standard SMTP port (25). If the hostname ends with a colon (`':'`) followed by a number, that suffix will be stripped off and the number interpreted as the port number to use. This method is automatically invoked by the constructor if a host is specified during instantiation. Returns a 2-tuple of the response code and message sent by the server in its connection response.

Raises an auditing event `smtplib.connect` with arguments `self`, `host`, `port`.

SMTP.helo(*name=''*)

Identify yourself to the SMTP server using `HELO`. The hostname argument defaults to the fully qualified domain name of the local host. The message returned by the server is stored as the `helo_resp` attribute of the object.

In normal operation it should not be necessary to call this method explicitly. It will be implicitly called by the `sendmail()` when necessary.

SMTP.ehlo(*name*='')

Identify yourself to an ESMTP server using `EHLO`. The hostname argument defaults to the fully qualified domain name of the local host. Examine the response for ESMTP option and store them for use by `has_extn()`. Also sets several informational attributes: the message returned by the server is stored as the `ehlo_resp` attribute, `does_esmtp` is set to `True` or `False` depending on whether the server supports ESMTP, and `esmtp_features` will be a dictionary containing the names of the SMTP service extensions this server supports, and their parameters (if any).

Unless you wish to use `has_extn()` before sending mail, it should not be necessary to call this method explicitly. It will be implicitly called by `sendmail()` when necessary.

SMTP.ehlo_or_helo_if_needed()

This method calls `ehlo()` and/or `helo()` if there has been no previous `EHLO` or `HELO` command this session. It tries ESMTP `EHLO` first.

`SMTPHeloError`

The server didn’t reply properly to the `HELO` greeting.

SMTP.has_extn(*name*)

Return `True` if *name* is in the set of SMTP service extensions returned by the server, `False` otherwise. Case is ignored.

SMTP.verify(*address*)

Check the validity of an address on this server using SMTP `VRFY`. Returns a tuple consisting of code 250 and a full **RFC 822** [https://datatracker.ietf.org/doc/html/rfc822.html] address (including human name) if the user address is valid. Otherwise returns an SMTP error code of 400 or greater and an error string.

Note

Many sites disable SMTP `VRFY` in order to foil spammers.

SMTP.login(*user*, *password*, *, *initial_response_ok=True*)

Log in on an SMTP server that requires authentication. The arguments are the username and the password to authenticate with. If there has been no previous `EHLO` or `HELO` command this session, this method tries ESMTP `EHLO` first. This method will return normally if the authentication was successful, or may raise the following exceptions:

`SMTPHeloError`
The server didn’t reply properly to the `HELO` greeting.

`SMTPAuthenticationError`
The server didn’t accept the username/password combination.

`SMTPNotSupportedError`
The `AUTH` command is not supported by the server.

`SMTPException`
No suitable authentication method was found.

Each of the authentication methods supported by `smtplib` are tried in turn if they are advertised as supported by the server. See `auth()` for a list of supported authentication methods. *initial_response_ok* is passed through to `auth()`.

Optional keyword argument *initial_response_ok* specifies whether, for authentication methods that support it, an "initial response" as specified in **RFC 4954** [https://datatracker.ietf.org/doc/html/rfc4954.html] can be sent along with the `AUTH` command, rather than requiring a challenge/response.

Changed in version 3.5: `SMTPNotSupportedError` may be raised, and the *initial_response_ok* parameter was added.

SMTP.auth(*mechanism*, *authobject*, *, *initial_response_ok=True*)

Issue an `SMTP` `AUTH` command for the specified authentication *mechanism*, and handle the challenge response via *authobject*.

mechanism specifies which authentication mechanism is to be used as argument to the `AUTH` command; the valid values are those listed in the `auth` element of `esmtp_features`.

authobject must be a callable object taking an optional single argument:

```
data = authobject(challenge=None)
```

If optional keyword argument *initial_response_ok* is true, `authobject()` will be called first with no argument. It can return the **RFC 4954** [https://datatracker.ietf.org/doc/html/rfc4954.html] "initial response" ASCII `str` which will be encoded and sent with the `AUTH` command as below. If the `authobject()` does not support an initial response (e.g. because it requires a challenge), it should return `None` when called with `challenge=None`. If *initial_response_ok* is false, then `authobject()` will not be called first with `None`.

If the initial response check returns `None`, or if *initial_response_ok* is false, `authobject()` will be called to process the server's challenge response; the *challenge* argument it is passed will be a `bytes`. It should return ASCII `str` *data* that will be base64 encoded and sent to the server.

The `SMTP` class provides `authobjects` for the `CRAM-MD5`, `PLAIN`, and `LOGIN` mechanisms; they are named `SMTP.auth_cram_md5`, `SMTP.auth_plain`, and `SMTP.auth_login` respectively. They all require that the `user` and `password` properties of the `SMTP` instance are set to appropriate values.

User code does not normally need to call `auth` directly, but can instead call the **`login()`** method, which will try each of the above mechanisms in turn, in the order listed. `auth` is exposed to facilitate the implementation of authentication methods not (or not yet) supported directly by **`smtplib`**.

Added in version 3.5.

SMTP.starttls(*, *context=None*)

Put the SMTP connection in TLS (Transport Layer Security) mode. All SMTP commands that follow will be encrypted. You should then call **`ehlo()`** again.

If *keyfile* and *certfile* are provided, they are used to create an **`ssl.SSLContext`**.

Optional *context* parameter is an `ssl.SSLContext` object; This is an alternative to using a keyfile and a certfile and if specified both *keyfile* and *certfile* should be `None`.

If there has been no previous `EHLO` or `HELO` command this session, this method tries ESMTP `EHLO` first.

Changed in version 3.12: The deprecated *keyfile* and *certfile* parameters have been removed.

`SMTPHeloError`
The server didn't reply properly to the `HELO` greeting.

`SMTPNotSupportedError`
The server does not support the STARTTLS extension.

`RuntimeError`
SSL/TLS support is not available to your Python interpreter.

Changed in version 3.3: *context* was added.

Changed in version 3.4: The method now supports hostname check with `ssl.SSLContext.check_hostname` and *Server Name Indicator* (see `HAS_SNI`).

Changed in version 3.5: The error raised for lack of STARTTLS support is now the `SMTPNotSupportedError` subclass instead of the base `SMTPException`.

SMTP.sendmail(*from_addr*, *to_addrs*, *msg*, *mail_options=()*, *rcpt_options=()*)

Send mail. The required arguments are an **RFC 822** [https://datatracker.ietf.org/doc/html/rfc822.html] from-address string, a list of **RFC 822** [https://datatracker.ietf.org/doc/html/rfc822.html] to-address strings (a bare string will be treated as a list with 1 address), and a message string. The caller may pass a list of ESMTP options (such as `"8bitmime"`) to be used in `MAIL FROM` commands as *mail_options*. ESMTP options (such as `DSN` commands) that should be used with all `RCPT` commands can be passed as *rcpt_options*. Each option should be passed as a string containing the full text of the option, including any potential key (for instance, `"NOTIFY=SUCCESS,FAILURE"`). (If you need to use different ESMTP options to different recipients you have to use the low-level methods such as **`mail()`**, **`rcpt()`** and **`data()`** to send the message.)

Note

The *from_addr* and *to_addrs* parameters are used to construct the message envelope used by the transport agents. `sendmail` does not modify the message headers in any way.

msg may be a string containing characters in the ASCII range, or a byte string. A string is encoded to bytes using the ascii codec, and lone `\r` and `\n` characters are converted to `\r\n` characters. A byte string is not modified.

If there has been no previous `EHLO` or `HELO` command this session, this method tries ESMTP `EHLO` first. If the server does ESMTP, message size and each of the specified options will be passed to it (if the option is in the feature set the server advertises). If `EHLO` fails, `HELO` will be tried and ESMTP options suppressed.

This method will return normally if the mail is accepted for at least one recipient. Otherwise it will raise an exception. That is, if this method does not raise an exception, then someone should get your mail. If this method does not raise an exception, it returns a dictionary, with one entry for each recipient that was refused. Each entry contains a tuple of the SMTP error code and the accompanying error message sent by the server.

If `SMTPUTF8` is included in *mail_options*, and the server supports it, *from_addr* and *to_addrs* may contain non-ASCII characters.

This method may raise the following exceptions:

`SMTPRecipientsRefused`
: All recipients were refused. Nobody got the mail.

`SMTPHeloError`
: The server didn’t reply properly to the `HELO` greeting.

`SMTPSenderRefused`
: The server didn’t accept the *from_addr*.

`SMTPDataError`
: The server replied with an unexpected error code (other than a refusal of a recipient).

`SMTPNotSupportedError`
: `SMTPUTF8` was given in the *mail_options* but is not supported by the server.

Unless otherwise noted, the connection will be open even after an exception is raised.

Changed in version 3.2: msg may be a byte string.

Changed in version 3.5: `SMTPUTF8` support added, and **`SMTPNotSupportedError`** may be raised if `SMTPUTF8` is specified but the server does not support it.

SMTP.send_message(*msg*, *from_addr=None*, *to_addrs=None*, *mail_options=()*, *rcpt_options=()*)

This is a convenience method for calling **`sendmail()`** with the message represented by an **`email.message.Message`** object. The arguments have the same meaning as for **`sendmail()`**, except that *msg* is a `Message` object.

If *from_addr* is `None` or *to_addrs* is `None`, `send_message` fills those arguments with addresses extracted from the headers of *msg* as specified in **RFC 5322** [https://datatracker.ietf.org/doc/html/rfc5322.html]: *from_addr* is set to the *Sender* field if it is present, and otherwise to the *From* field. *to_addrs* combines the values (if any) of the *To*, *Cc*, and *Bcc* fields from *msg*. If exactly one set of *Resent-** headers appear in the message, the regular headers are ignored and the *Resent-** headers are used instead. If the message contains more than one set of *Resent-** headers, a **`ValueError`** is raised, since there is no way to unambiguously detect the most recent set of *Resent-* headers.

`send_message` serializes *msg* using **`BytesGenerator`** with `\r\n` as the *linesep*, and calls **`sendmail()`** to transmit the resulting message. Regardless of the values of *from_addr* and *to_addrs*, `send_message` does not transmit any *Bcc* or *Resent-Bcc* headers that may appear in *msg*. If any of the addresses in *from_addr* and *to_addrs* contain non-ASCII characters and the server does not advertise `SMTPUTF8` support, an **`SMTPNotSupportedError`** is raised. Otherwise the `Message` is serialized with a clone of its **`policy`** with the **`utf8`** attribute set to `True`, and `SMTPUTF8` and `BODY=8BITMIME` are added to *mail_options*.

Added in version 3.2.

Added in version 3.5: Support for internationalized addresses (`SMTPUTF8`).

SMTP.quit()

Terminate the SMTP session and close the connection. Return the result of the SMTP `QUIT` command.

Low-level methods corresponding to the standard SMTP/ESMTP commands `HELP`, `RSET`, `NOOP`, `MAIL`, `RCPT`, and `DATA` are also supported. Normally these do not need to be called directly, so they are not documented here. For details, consult the module code.

Additionally, an SMTP instance has the following attributes:

SMTP.helo_resp

The response to the `HELO` command, see **`helo()`**.

SMTP.ehlo_resp

The response to the `EHLO` command, see **`ehlo()`**.

SMTP.does_esmtp

A boolean value indicating whether the server supports ESMTP, see **`ehlo()`**.

SMTP.esmtp_features

A dictionary of the names of SMTP service extensions supported by the server, see **`ehlo()`**.

SMTP Example

This example prompts the user for addresses needed in the message envelope (‘To’ and ‘From’ addresses), and the message to be delivered. Note that the headers to be included with the message must be included in the message as entered; this example doesn’t do any processing of the

RFC 822 [https://datatracker.ietf.org/doc/html/rfc822.html] headers. In particular, the ‘To’ and ‘From’ addresses must be included in the message headers explicitly:

```
import smtplib

def prompt(title):
    return input(title).strip()

from_addr = prompt("From: ")
to_addrs  = prompt("To: ").split()
print("Enter message, end with ^D (Unix) or ^Z (Windows):")

# Add the From: and To: headers at the start!
lines = [f"From: {from_addr}", f"To: {', '.join(to_addrs)}",
""]
while True:
    try:
        line = input()
    except EOFError:
        break
    else:
        lines.append(line)

msg = "\r\n".join(lines)
print("Message length is", len(msg))

server = smtplib.SMTP("localhost")
server.set_debuglevel(1)
server.sendmail(from_addr, to_addrs, msg)
server.quit()
```

Note

In general, you will want to use the **`email`** package’s features to construct an email message, which you can then send via **`send_message()`**; see email: Examples.

`uuid` — UUID objects according to RFC 9562 [https://datatracker.ietf.org/doc/html/rfc9562.html]

Source code: Lib/uuid.py [https://github.com/python/cpython/tree/3.14/Lib/uuid.py]

This module provides immutable `UUID` objects (the `UUID` class) and functions for generating UUIDs corresponding to a specific UUID version as specified in **RFC 9562** [https://datatracker.ietf.org/doc/html/rfc9562.html] (which supersedes **RFC 4122** [https://datatracker.ietf.org/doc/html/rfc4122.html]), for example, `uuid1()` for UUID version 1, `uuid3()` for UUID version 3, and so on. Note that UUID version 2 is deliberately omitted as it is outside the scope of the RFC.

If all you want is a unique ID, you should probably call `uuid1()` or `uuid4()`. Note that `uuid1()` may compromise privacy since it creates a UUID containing the computer's network address. `uuid4()` creates a random UUID.

Depending on support from the underlying platform, `uuid1()` may or may not return a "safe" UUID. A safe UUID is one which is generated using synchronization methods that ensure no two processes can obtain the same UUID. All instances of `UUID` have an `is_safe` attribute which relays any information about the UUID's safety, using this enumeration:

class uuid.SafeUUID

Added in version 3.7.

safe

The UUID was generated by the platform in a multiprocessing-safe way.

unsafe

The UUID was not generated in a multiprocessing-safe way.

unknown

The platform does not provide information on whether the UUID was generated safely or not.

class uuid.UUID(*hex=None*, *bytes=None*, *bytes_le=None*, *fields=None*, *int=None*, *version=None*, *, *is_safe=SafeUUID.unknown*)

Create a UUID from either a string of 32 hexadecimal digits, a string of 16 bytes in big-endian order as the *bytes* argument, a string of 16 bytes in little-endian order as the *bytes_le* argument, a tuple of six integers (32-bit *time_low*, 16-bit *time_mid*, 16-bit *time_hi_version*, 8-bit *clock_seq_hi_variant*, 8-bit *clock_seq_low*, 48-bit *node*) as the *fields* argument, or a single 128-bit integer as the *int* argument. When a string of hex digits is given, curly braces, hyphens, and a URN prefix are all optional. For example, these expressions all yield the same UUID:

```
UUID('{12345678-1234-5678-1234-567812345678}')
UUID('12345678123456781234567812345678')
UUID('urn:uuid:12345678-1234-5678-1234-567812345678')
UUID(bytes=b'\x12\x34\x56\x78'*4)
UUID(bytes_le=b'\x78\x56\x34\x12\x34\x12\x78\x56' +
              b'\x12\x34\x56\x78\x12\x34\x56\x78')
UUID(fields=(0x12345678, 0x1234, 0x5678, 0x12, 0x34,
0x567812345678))
UUID(int=0x12345678123456781234567812345678)
```

Exactly one of *hex*, *bytes*, *bytes_le*, *fields*, or *int* must be given. The *version* argument is optional; if given, the resulting UUID will have its variant and version number set according to **RFC 9562** [https://datatracker.ietf.org/doc/html/rfc9562.html], overriding bits in the given *hex*, *bytes*, *bytes_le*, *fields*, or *int*.

Comparison of UUID objects are made by way of comparing their **`UUID.int`** attributes. Comparison with a non-UUID object raises a **`TypeError`**.

`str(uuid)` returns a string in the form `12345678-1234-5678-1234-567812345678` where the 32 hexadecimal digits represent the UUID.

`UUID` instances have these read-only attributes:

UUID.bytes

The UUID as a 16-byte string (containing the six integer fields in big-endian byte order).

UUID.bytes_le

The UUID as a 16-byte string (with *time_low*, *time_mid*, and *time_hi_version* in little-endian byte order).

UUID.fields

A tuple of the six integer fields of the UUID, which are also available as six individual attributes and two derived attributes:

Field	Meaning

UUID.time_low	The first 32 bits of the UUID. Only relevant to version 1.
UUID.time_mid	The next 16 bits of the UUID. Only relevant to version 1.
UUID.time_hi_version	The next 16 bits of the UUID. Only relevant to version 1.
UUID.clock_seq_hi_variant	The next 8 bits of the UUID. Only relevant to versions 1 and 6.
UUID.clock_seq_low	The next 8 bits of the UUID. Only relevant to versions 1 and 6.
UUID.node	The last 48 bits of the UUID. Only relevant to version 1.
UUID.time	The 60-bit timestamp as a count of 100-nanosecond intervals since Gregorian epoch (1582-10-15 00:00:00) for versions 1 and 6, or the 48-bit timestamp in milliseconds since Unix epoch (1970-01-01 00:00:00) for version 7.

UUID.clock_seq	The 14-bit sequence number. Only relevant to versions 1 and 6.

UUID.hex

The UUID as a 32-character lowercase hexadecimal string.

UUID.int

The UUID as a 128-bit integer.

UUID.urn

The UUID as a URN as specified in **RFC 9562** [https://datatracker.ietf.org/doc/html/rfc9562.html].

UUID.variant

The UUID variant, which determines the internal layout of the UUID. This will be one of the constants **`RESERVED_NCS`**, **`RFC_4122`**, **`RESERVED_MICROSOFT`**, or **`RESERVED_FUTURE`**.

UUID.version

The UUID version number (1 through 8, meaningful only when the variant is **`RFC_4122`**).

Changed in version 3.14: Added UUID versions 6, 7 and 8.

UUID.is_safe

An enumeration of **`SafeUUID`** which indicates whether the platform generated the UUID in a multiprocessing-safe way.

Added in version 3.7.

The `uuid` module defines the following functions:

uuid.getnode()

Get the hardware address as a 48-bit positive integer. The first time this runs, it may launch a separate program, which could be quite slow. If all attempts to obtain the hardware address fail, we choose a random 48-bit number with the multicast bit (least significant bit of the first octet) set to 1 as recommended in **RFC 4122** [https://datatracker.ietf.org/doc/html/rfc4122.html]. “Hardware address” means the MAC address of a network interface. On a machine with multiple network interfaces, universally administered MAC addresses (i.e. where the second least significant bit of the first octet is *unset*) will be preferred over locally administered MAC addresses, but with no other ordering guarantees.

Changed in version 3.7: Universally administered MAC addresses are preferred over locally administered MAC addresses, since the former are guaranteed to be globally unique, while the latter are not.

uuid.uuid1(*node=None*, *clock_seq=None*)

Generate a UUID from a host ID, sequence number, and the current time according to **RFC 9562, §5.1** [https://datatracker.ietf.org/doc/html/rfc9562.html#section-5.1].

When *node* is not specified, `getnode()` is used to obtain the hardware address as a 48-bit positive integer. When a sequence number *clock_seq* is not specified, a pseudo-random 14-bit positive integer is generated.

If *node* or *clock_seq* exceed their expected bit count, only their least significant bits are kept.

uuid.uuid3(*namespace*, *name*)

Generate a UUID based on the MD5 hash of a namespace identifier (which is a UUID) and a name (which is a `bytes` object or a string that will be encoded using UTF-8) according to **RFC 9562, §5.3** [https://datatracker.ietf.org/doc/html/rfc9562.html#section-5.3].

uuid.uuid4()

Generate a random UUID in a cryptographically-secure method according to **RFC 9562, §5.4** [https://datatracker.ietf.org/doc/html/rfc9562.html#section-5.4].

uuid.uuid5(*namespace*, *name*)

Generate a UUID based on the SHA-1 hash of a namespace identifier (which is a UUID) and a name (which is a `bytes` object or a string that will be encoded using UTF-8) according to **RFC 9562, §5.5** [https://datatracker.ietf.org/doc/html/rfc9562.html#section-5.5].

uuid.uuid6(*node=None*, *clock_seq=None*)

Generate a UUID from a sequence number and the current time according to **RFC 9562, §5.6** [https://datatracker.ietf.org/doc/html/rfc9562.html#section-5.6].

This is an alternative to `uuid1()` to improve database locality.

When *node* is not specified, `getnode()` is used to obtain the hardware address as a 48-bit positive integer. When a sequence number *clock_seq* is not specified, a pseudo-random 14-bit positive integer is generated.

If *node* or *clock_seq* exceed their expected bit count, only their least significant bits are kept.

Added in version 3.14.

uuid.uuid7()

Generate a time-based UUID according to **RFC 9562, §5.7** [https://datatracker.ietf.org/doc/html/rfc9562.html#section-5.7].

For portability across platforms lacking sub-millisecond precision, UUIDs produced by this function embed a 48-bit timestamp and use a 42-bit counter to guarantee monotonicity within a millisecond.

Added in version 3.14.

uuid.uuid8(*a=None*, *b=None*, *c=None*)

Generate a pseudo-random UUID according to **RFC 9562, §5.8** [https://datatracker.ietf.org/doc/html/rfc9562.html#section-5.8].

When specified, the parameters *a*, *b* and *c* are expected to be positive integers of 48, 12 and 62 bits respectively. If they exceed their expected bit count, only their least significant bits are kept; non-specified arguments are substituted for a pseudo-random integer of appropriate size.

By default, *a*, *b* and *c* are not generated by a cryptographically secure pseudo-random number generator (CSPRNG). Use `uuid4()` when a UUID needs to be used in a security-sensitive context.

Added in version 3.14.

The `uuid` module defines the following namespace identifiers for use with `uuid3()` or `uuid5()`.

uuid.NAMESPACE_DNS

When this namespace is specified, the *name* string is a fully qualified domain name.

uuid.NAMESPACE_URL

When this namespace is specified, the *name* string is a URL.

uuid.NAMESPACE_OID

When this namespace is specified, the *name* string is an ISO OID.

uuid.NAMESPACE_X500

When this namespace is specified, the *name* string is an X.500 DN in DER or a text output format.

The `uuid` module defines the following constants for the possible values of the `variant` attribute:

uuid.RESERVED_NCS

Reserved for NCS compatibility.

uuid.RFC_4122

Specifies the UUID layout given in **RFC 4122** [https://datatracker.ietf.org/doc/html/rfc4122.html]. This constant is kept for backward compatibility even though **RFC 4122** [https://datatracker.ietf.org/doc/html/rfc4122.html] has been superseded by **RFC 9562** [https://datatracker.ietf.org/doc/html/rfc9562.html].

uuid.RESERVED_MICROSOFT

Reserved for Microsoft compatibility.

uuid.RESERVED_FUTURE

Reserved for future definition.

The `uuid` module defines the special Nil and Max UUID values:

uuid.NIL

A special form of UUID that is specified to have all 128 bits set to zero according to RFC 9562, §5.9 [https://datatracker.ietf.org/doc/html/rfc9562.html#section-5.9].

Added in version 3.14.

uuid.MAX

A special form of UUID that is specified to have all 128 bits set to one according to RFC 9562, §5.10 [https://datatracker.ietf.org/doc/html/rfc9562.html#section-5.10].

Added in version 3.14.

See also

RFC 9562 [https://datatracker.ietf.org/doc/html/rfc9562.html] - A Universally Unique IDentifier (UUID) URN Namespace

This specification defines a Uniform Resource Name namespace for UUIDs, the internal format of UUIDs, and methods of generating UUIDs.

Command-Line Usage

Added in version 3.12.

The `uuid` module can be executed as a script from the command line.

```
python -m uuid [-h] [-u
{uuid1,uuid3,uuid4,uuid5,uuid6,uuid7,uuid8}] [-n NAMESPACE] [-N
NAME]
```

The following options are accepted:

-h, --help

Show the help message and exit.

-u <uuid>
--uuid <uuid>

Specify the function name to use to generate the uuid. By default `uuid4()` is used.

Changed in version 3.14: Allow generating UUID versions 6, 7 and 8.

-n <namespace>
--namespace <namespace>

The namespace is a `UUID`, or `@ns` where `ns` is a well-known predefined UUID addressed by namespace name. Such as `@dns`, `@url`, `@oid`, and `@x500`. Only required for `uuid3()` / `uuid5()` functions.

-N <name>
--name <name>

The name used as part of generating the uuid. Only required for `uuid3()` / `uuid5()` functions.

-C <num>
--count <num>

Generate *num* fresh UUIDs.

Added in version 3.14.

Example

Here are some examples of typical usage of the `uuid` module:

```
>>> import uuid

>>> # make a UUID based on the host ID and current time
>>> uuid.uuid1()
UUID('a8098c1a-f86e-11da-bd1a-00112444be1e')

>>> # make a UUID using an MD5 hash of a namespace UUID and a
name
>>> uuid.uuid3(uuid.NAMESPACE_DNS, 'python.org')
UUID('6fa459ea-ee8a-3ca4-894e-db77e160355e')

>>> # make a random UUID
>>> uuid.uuid4()
UUID('16fd2706-8baf-433b-82eb-8c7fada847da')

>>> # make a UUID using a SHA-1 hash of a namespace UUID and a
name
>>> uuid.uuid5(uuid.NAMESPACE_DNS, 'python.org')
UUID('886313e1-3b8a-5372-9b90-0c9aee199e5d')

>>> # make a UUID from a string of hex digits (braces and
hyphens ignored)
>>> x = uuid.UUID('{00010203-0405-0607-0809-0a0b0c0d0e0f}')

>>> # convert a UUID to a string of hex digits in standard form
>>> str(x)
'00010203-0405-0607-0809-0a0b0c0d0e0f'

>>> # get the raw 16 bytes of the UUID
>>> x.bytes
b'\x00\x01\x02\x03\x04\x05\x06\x07\x08\t\n\x0b\x0c\r\x0e\x0f'

>>> # make a UUID from a 16-byte string
>>> uuid.UUID(bytes=x.bytes)
UUID('00010203-0405-0607-0809-0a0b0c0d0e0f')

>>> # get the Nil UUID
>>> uuid.NIL
UUID('00000000-0000-0000-0000-000000000000')

>>> # get the Max UUID
>>> uuid.MAX
UUID('ffffffff-ffff-ffff-ffff-ffffffffffff')

>>> # same as UUIDv1 but with fields reordered to improve DB
locality
```

```
>>> uuid.uuid6()
UUID('1f0799c0-98b9-62db-92c6-a0d365b91053')

>>> # get UUIDv7 creation (local) time as a timestamp in
milliseconds
>>> u = uuid.uuid7()
>>> u.time
1743936859822

>>> # get UUIDv7 creation (local) time as a datetime object
>>> import datetime as dt
>>> dt.datetime.fromtimestamp(u.time / 1000)
datetime.datetime(...)

>>> # make a UUID with custom blocks
>>> uuid.uuid8(0x12345678, 0x9abcdef0, 0x11223344)
UUID('00001234-5678-8ef0-8000-000011223344')
```

Command-Line Example

Here are some examples of typical usage of the `uuid` command-line interface:

```
# generate a random UUID - by default uuid4() is used
$ python -m uuid

# generate a UUID using uuid1()
$ python -m uuid -u uuid1

# generate a UUID using uuid5
$ python -m uuid -u uuid5 -n @url -N example.com

# generate 42 random UUIDs
$ python -m uuid -C 42
```

`socketserver` — A framework for network servers

Source code: Lib/socketserver.py [https://github.com/python/cpython/tree/3.14/Lib/socketserver.py]

The **`socketserver`** module simplifies the task of writing network servers.

Availability: not WASI.

This module does not work or is not available on WebAssembly. See WebAssembly platforms for more information.

There are four basic concrete server classes:

class socketserver.TCPServer(*server_address*, *RequestHandlerClass*, *bind_and_activate=True*)

This uses the internet TCP protocol, which provides for continuous streams of data between the client and server. If *bind_and_activate* is true, the constructor automatically attempts to invoke **`server_bind()`** and **`server_activate()`**. The other parameters are passed to the **`BaseServer`** base class.

class socketserver.UDPServer(*server_address*, *RequestHandlerClass*, *bind_and_activate=True*)

This uses datagrams, which are discrete packets of information that may arrive out of order or be lost while in transit. The parameters are the same as for **`TCPServer`**.

class socketserver.UnixStreamServer(*server_address*, *RequestHandlerClass*, *bind_and_activate=True*)
class socketserver.UnixDatagramServer(*server_address*, *RequestHandlerClass*, *bind_and_activate=True*)

> These more infrequently used classes are similar to the TCP and UDP classes, but use Unix domain sockets; they’re not available on non-Unix platforms. The parameters are the same as for `TCPServer`.

These four classes process requests *synchronously*; each request must be completed before the next request can be started. This isn’t suitable if each request takes a long time to complete, because it requires a lot of computation, or because it returns a lot of data which the client is slow to process. The solution is to create a separate process or thread to handle each request; the `ForkingMixIn` and `ThreadingMixIn` mix-in classes can be used to support asynchronous behaviour.

Creating a server requires several steps. First, you must create a request handler class by subclassing the `BaseRequestHandler` class and overriding its `handle()` method; this method will process incoming requests. Second, you must instantiate one of the server classes, passing it the server’s address and the request handler class. It is recommended to use the server in a `with` statement. Then call the `handle_request()` or `serve_forever()` method of the server object to process one or many requests. Finally, call `server_close()` to close the socket (unless you used a **`with`** statement).

When inheriting from `ThreadingMixIn` for threaded connection behavior, you should explicitly declare how you want your threads to behave on an abrupt shutdown. The `ThreadingMixIn` class defines an attribute *daemon_threads*, which indicates whether or not the server should wait for thread termination. You should set the flag explicitly if you would like threads to behave autonomously; the default is `False`, meaning that Python will not exit until all threads created by `ThreadingMixIn` have exited.

Server classes have the same external methods and attributes, no matter what network protocol they use.

Server Creation Notes

There are five classes in an inheritance diagram, four of which represent synchronous servers of four types:

```
+------------+
| BaseServer |
+------------+
      |
      v
+-----------+        +------------------+
| TCPServer |------->| UnixStreamServer |
+-----------+        +------------------+
      |
      v
+-----------+        +--------------------+
| UDPServer |------->| UnixDatagramServer |
+-----------+        +--------------------+
```

Note that `UnixDatagramServer` derives from `UDPServer`, not from `UnixStreamServer` — the only difference between an IP and a Unix server is the address family.

class socketserver.ForkingMixIn
class socketserver.ThreadingMixIn

Forking and threading versions of each type of server can be created using these mix-in classes. For instance, `ThreadingUDPServer` is created as follows:

```
class ThreadingUDPServer(ThreadingMixIn, UDPServer):
    pass
```

The mix-in class comes first, since it overrides a method defined in `UDPServer`. Setting the various attributes also changes the behavior of the underlying server mechanism.

`ForkingMixIn` and the Forking classes mentioned below are only available on POSIX platforms that support `fork()`.

block_on_close

`ForkingMixIn.server_close` waits until all child processes complete, except if `block_on_close` attribute is `False`.

`ThreadingMixIn.server_close` waits until all non-daemon threads complete, except if `block_on_close` attribute is `False`.

max_children

Specify how many child processes will exist to handle requests at a time for `ForkingMixIn`. If the limit is reached, new requests will wait until one child process has finished.

daemon_threads

For `ThreadingMixIn` use daemonic threads by setting `ThreadingMixIn.daemon_threads` to `True` to not wait until threads complete.

Changed in version 3.7: `ForkingMixIn.server_close` and `ThreadingMixIn.server_close` now waits until all child processes and non-daemonic threads complete. Add a new `ForkingMixIn.block_on_close` class attribute to opt-in for the pre-3.7 behaviour.

class socketserver.ForkingTCPServer

class socketserver.ForkingUDPServer
class socketserver.ThreadingTCPServer
class socketserver.ThreadingUDPServer
class socketserver.ForkingUnixStreamServer
class socketserver.ForkingUnixDatagramServer
class socketserver.ThreadingUnixStreamServer
class socketserver.ThreadingUnixDatagramServer

These classes are pre-defined using the mix-in classes.

Added in version 3.12: The `ForkingUnixStreamServer` and `ForkingUnixDatagramServer` classes were added.

To implement a service, you must derive a class from `BaseRequestHandler` and redefine its `handle()` method. You can then run various versions of the service by combining one of the server classes with your request handler class. The request handler class must be different for datagram or stream services. This can be hidden by using the handler subclasses `StreamRequestHandler` or `DatagramRequestHandler`.

Of course, you still have to use your head! For instance, it makes no sense to use a forking server if the service contains state in memory that can be modified by different requests, since the modifications in the child process would never reach the initial state kept in the parent process and passed to each child. In this case, you can use a threading server, but you will probably have to use locks to protect the integrity of the shared data.

On the other hand, if you are building an HTTP server where all data is stored externally (for instance, in the file system), a synchronous class will essentially render the service “deaf” while one request is being handled – which may be for a very long time if a client is slow to receive all the data it has requested. Here a threading or forking server is appropriate.

In some cases, it may be appropriate to process part of a request synchronously, but to finish processing in a forked child depending on the request data. This can be implemented by using a synchronous server and doing an explicit fork in the request handler class `handle()` method.

Another approach to handling multiple simultaneous requests in an environment that supports neither threads nor `fork()` (or where these are too expensive or inappropriate for the service) is to maintain an explicit table of partially finished requests and to use `selectors` to decide which request to work on next (or whether to handle a new incoming request). This is particularly important for stream services where each client can potentially be connected for a long time (if threads or subprocesses cannot be used).

Server Objects

class socketserver.BaseServer(*server_address*, *RequestHandlerClass*)

This is the superclass of all Server objects in the module. It defines the interface, given below, but does not implement most of the methods, which is done in subclasses. The two parameters are stored in the respective `server_address` and `RequestHandlerClass` attributes.

fileno()

Return an integer file descriptor for the socket on which the server is listening. This function is most commonly passed to `selectors`, to allow monitoring multiple servers in the same process.

handle_request()

Process a single request. This function calls the following methods in order: `get_request()`, `verify_request()`, and `process_request()`. If the user-provided `handle()` method of the handler class raises an exception, the server's `handle_error()` method will be called. If no request is received within `timeout` seconds, `handle_timeout()` will be called and `handle_request()` will return.

serve_forever(*poll_interval=0.5*)

Handle requests until an explicit `shutdown()` request. Poll for shutdown every *poll_interval* seconds. Ignores the `timeout` attribute. It also calls `service_actions()`, which may be used by a subclass or mixin to provide actions specific to a given service. For example, the `ForkingMixIn` class uses `service_actions()` to clean up zombie child processes.

Changed in version 3.3: Added `service_actions` call to the `serve_forever` method.

service_actions()

This is called in the `serve_forever()` loop. This method can be overridden by subclasses or mixin classes to perform actions specific to a given service, such as cleanup actions.

Added in version 3.3.

shutdown()

Tell the `serve_forever()` loop to stop and wait until it does. `shutdown()` must be called while `serve_forever()` is running in a different thread otherwise it will deadlock.

server_close()

Clean up the server. May be overridden.

address_family

The family of protocols to which the server's socket belongs. Common examples are `socket.AF_INET`, `socket.AF_INET6`, and `socket.AF_UNIX`. Subclass the TCP or UDP server classes in this module with class attribute `address_family = AF_INET6` set if you want IPv6 server classes.

RequestHandlerClass

The user-provided request handler class; an instance of this class is created for each request.

server_address

The address on which the server is listening. The format of addresses varies depending on the protocol family; see the documentation for the `socket` module for details. For internet protocols, this is a tuple containing a string giving the address, and an integer port number: `('127.0.0.1', 80)`, for example.

socket

The socket object on which the server will listen for incoming requests.

The server classes support the following class variables:

allow_reuse_address

Whether the server will allow the reuse of an address. This defaults to `False`, and can be set in subclasses to change the policy.

request_queue_size

The size of the request queue. If it takes a long time to process a single request, any requests that arrive while the server is busy are placed into a queue, up to `request_queue_size` requests. Once the queue is full, further requests from clients will get a “Connection denied” error. The default value is usually 5, but this can be overridden by subclasses.

socket_type

The type of socket used by the server; `socket.SOCK_STREAM` and `socket.SOCK_DGRAM` are two common values.

timeout

Timeout duration, measured in seconds, or `None` if no timeout is desired. If `handle_request()` receives no incoming requests within the timeout period, the `handle_timeout()` method is called.

There are various server methods that can be overridden by subclasses of base server classes like `TCPServer`; these methods aren’t useful to external users of the server object.

finish_request(*request*, *client_address*)

Actually processes the request by instantiating `RequestHandlerClass` and calling its `handle()` method.

get_request()

Must accept a request from the socket, and return a 2-tuple containing the *new* socket object to be used to communicate with the client, and the client’s address.

handle_error(*request*, *client_address*)

This function is called if the `handle()` method of a `RequestHandlerClass` instance raises an exception. The default action is to print the traceback to standard error and continue handling further requests.

Changed in version 3.6: Now only called for exceptions derived from the `Exception` class.

handle_timeout()

This function is called when the `timeout` attribute has been set to a value other than `None` and the timeout period has passed with no requests being received. The default action for forking servers is to collect the status of any child processes that have exited, while in threading servers this method does nothing.

process_request(*request*, *client_address*)

Calls `finish_request()` to create an instance of the `RequestHandlerClass`. If desired, this function can create a new process or thread to handle the request; the `ForkingMixIn` and `ThreadingMixIn` classes do this.

server_activate()

Called by the server's constructor to activate the server. The default behavior for a TCP server just invokes `listen()` on the server's socket. May be overridden.

server_bind()

Called by the server's constructor to bind the socket to the desired address. May be overridden.

verify_request(*request*, *client_address*)

Must return a Boolean value; if the value is `True`, the request will be processed, and if it's `False`, the request will be denied. This function can be overridden to implement access controls for a server. The default implementation always returns `True`.

Changed in version 3.6: Support for the context manager protocol was added. Exiting the context manager is equivalent to calling `server_close()`.

Request Handler Objects

class socketserver.BaseRequestHandler

This is the superclass of all request handler objects. It defines the interface, given below. A concrete request handler subclass must define a new `handle()` method, and can override any of the other methods. A new instance of the subclass is created for each request.

setup()

Called before the `handle()` method to perform any initialization actions required. The default implementation does nothing.

handle()

This function must do all the work required to service a request. The default implementation does nothing. Several instance attributes are available to it; the request is available as `request`; the client address as `client_address`; and the server instance as `server`, in case it needs access to per-server information.

The type of `request` is different for datagram or stream services. For stream services, `request` is a socket object; for datagram services, `request` is a pair of string and socket.

finish()

Called after the `handle()` method to perform any clean-up actions required. The default implementation does nothing. If `setup()` raises an exception, this function will not be called.

request

The *new* `socket.socket` object to be used to communicate with the client.

client_address

Client address returned by `BaseServer.get_request()`.

server

`BaseServer` object used for handling the request.

class socketserver.StreamRequestHandler
class socketserver.DatagramRequestHandler

These `BaseRequestHandler` subclasses override the `setup()` and `finish()` methods, and provide `rfile` and `wfile` attributes.

rfile

A file object from which receives the request is read. Support the `io.BufferedIOBase` readable interface.

wfile

A file object to which the reply is written. Support the `io.BufferedIOBase` writable interface

Changed in version 3.6: `wfile` also supports the `io.BufferedIOBase` writable interface.

Examples

`socketserver.TCPServer` Example

This is the server side:

```
import socketserver

class MyTCPHandler(socketserver.BaseRequestHandler):
    """
    The request handler class for our server.

    It is instantiated once per connection to the server, and
must
    override the handle() method to implement communication to
the
    client.
    """

    def handle(self):
        # self.request is the TCP socket connected to the
client
        pieces = [b'']
        total = 0
        while b'\n' not in pieces[-1] and total < 10_000:
            pieces.append(self.request.recv(2000))
            total += len(pieces[-1])
        self.data = b''.join(pieces)
        print(f"Received from {self.client_address[0]}:")
        print(self.data.decode("utf-8"))
        # just send back the same data, but upper-cased
        self.request.sendall(self.data.upper())
        # after we return, the socket will be closed.

if __name__ == "__main__":
    HOST, PORT = "localhost", 9999

    # Create the server, binding to localhost on port 9999
    with socketserver.TCPServer((HOST, PORT), MyTCPHandler) as
server:
        # Activate the server; this will keep running until you
        # interrupt the program with Ctrl-C
        server.serve_forever()
```

An alternative request handler class that makes use of streams (file-like objects that simplify communication by providing the standard file interface):

```
class MyTCPHandler(socketserver.StreamRequestHandler):

    def handle(self):
        # self.rfile is a file-like object created by the
handler.
        # We can now use e.g. readline() instead of raw recv()
calls.
        # We limit ourselves to 10000 bytes to avoid abuse by
the sender.
        self.data = self.rfile.readline(10000).rstrip()
        print(f"{self.client_address[0]} wrote:")
        print(self.data.decode("utf-8"))
        # Likewise, self.wfile is a file-like object used to
write back
        # to the client
        self.wfile.write(self.data.upper())
```

The difference is that the `readline()` call in the second handler will call `recv()` multiple times until it encounters a newline character, while the first handler had to use a `recv()` loop to accumulate data until a newline itself. If it had just used a single `recv()` without the loop it would just have returned what has been received so far from the client. TCP is stream based: data arrives in the order it was sent, but there is no correlation between client `send()` or `sendall()` calls and the number of `recv()` calls on the server required to receive it.

This is the client side:

```
import socket
import sys

HOST, PORT = "localhost", 9999
data = " ".join(sys.argv[1:])

# Create a socket (SOCK_STREAM means a TCP socket)
with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as sock:
    # Connect to server and send data
    sock.connect((HOST, PORT))
    sock.sendall(bytes(data, "utf-8"))
    sock.sendall(b"\n")

    # Receive data from the server and shut down
    received = str(sock.recv(1024), "utf-8")

print("Sent:    ", data)
print("Received:", received)
```

The output of the example should look something like this:

Server:

```
$ python TCPServer.py
127.0.0.1 wrote:
b'hello world with TCP'
127.0.0.1 wrote:
b'python is nice'
```

Client:

```
$ python TCPClient.py hello world with TCP
Sent:     hello world with TCP
Received: HELLO WORLD WITH TCP
$ python TCPClient.py python is nice
Sent:     python is nice
Received: PYTHON IS NICE
```

`socketserver.UDPServer` Example

This is the server side:

```
import socketserver

class MyUDPHandler(socketserver.BaseRequestHandler):
    """
    This class works similar to the TCP handler class, except
that
    self.request consists of a pair of data and client socket,
and since
    there is no connection the client address must be given
explicitly
```

```
    when sending data back via sendto().
    """

    def handle(self):
        data = self.request[0].strip()
        socket = self.request[1]
        print(f"{self.client_address[0]} wrote:")
        print(data)
        socket.sendto(data.upper(), self.client_address)

if __name__ == "__main__":
    HOST, PORT = "localhost", 9999
    with socketserver.UDPServer((HOST, PORT), MyUDPHandler) as 
server:
        server.serve_forever()
```

This is the client side:

```
import socket
import sys

HOST, PORT = "localhost", 9999
data = " ".join(sys.argv[1:])

# SOCK_DGRAM is the socket type to use for UDP sockets
sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)

# As you can see, there is no connect() call; UDP has no 
connections.
# Instead, data is directly sent to the recipient via sendto().
sock.sendto(bytes(data + "\n", "utf-8"), (HOST, PORT))
received = str(sock.recv(1024), "utf-8")

print("Sent:    ", data)
print("Received:", received)
```

The output of the example should look exactly like for the TCP server example.

Asynchronous Mixins

To build asynchronous handlers, use the `ThreadingMixIn` and `ForkingMixIn` classes.

An example for the `ThreadingMixIn` class:

```
import socket
import threading
import socketserver

class
ThreadedTCPRequestHandler(socketserver.BaseRequestHandler):

    def handle(self):
        data = str(self.request.recv(1024), 'ascii')
        cur_thread = threading.current_thread()
        response = bytes("{}: {}".format(cur_thread.name,
data), 'ascii')
        self.request.sendall(response)

class ThreadedTCPServer(socketserver.ThreadingMixIn,
socketserver.TCPServer):
    pass

def client(ip, port, message):
    with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as
sock:
        sock.connect((ip, port))
        sock.sendall(bytes(message, 'ascii'))
        response = str(sock.recv(1024), 'ascii')
        print("Received: {}".format(response))

if __name__ == "__main__":
    # Port 0 means to select an arbitrary unused port
    HOST, PORT = "localhost", 0

    server = ThreadedTCPServer((HOST, PORT),
ThreadedTCPRequestHandler)
    with server:
        ip, port = server.server_address

        # Start a thread with the server -- that thread will
then start one
```

```
        # more thread for each request
        server_thread =
threading.Thread(target=server.serve_forever)
        # Exit the server thread when the main thread
terminates
        server_thread.daemon = True
        server_thread.start()
        print("Server loop running in thread:",
server_thread.name)

        client(ip, port, "Hello World 1")
        client(ip, port, "Hello World 2")
        client(ip, port, "Hello World 3")

        server.shutdown()
```

The output of the example should look something like this:

```
$ python ThreadedTCPServer.py
Server loop running in thread: Thread-1
Received: Thread-2: Hello World 1
Received: Thread-3: Hello World 2
Received: Thread-4: Hello World 3
```

The **`ForkingMixIn`** class is used in the same way, except that the server will spawn a new process for each request. Available only on POSIX platforms that support **`fork()`**.

`http.server` — HTTP servers

Source code: Lib/http/server.py
[https://github.com/python/cpython/tree/3.14/Lib/http/server.py]

This module defines classes for implementing HTTP servers.

Warning

`http.server` is not recommended for production. It only implements basic security checks.

Availability: not WASI.

This module does not work or is not available on WebAssembly. See WebAssembly platforms for more information.

One class, `HTTPServer`, is a `socketserver.TCPServer` subclass. It creates and listens at the HTTP socket, dispatching the requests to a handler. Code to create and run the server looks like this:

```
def run(server_class=HTTPServer,
handler_class=BaseHTTPRequestHandler):
    server_address = ('', 8000)
    httpd = server_class(server_address, handler_class)
    httpd.serve_forever()
```

class http.server.HTTPServer(*server_address*, *RequestHandlerClass*)

This class builds on the `TCPServer` class by storing the server address as instance variables named **`server_name`** and `server_port`. The server is accessible by the handler, typically through the handler's `server` instance variable.

class http.server.ThreadingHTTPServer(*server_address*, *RequestHandlerClass*)

This class is identical to HTTPServer but uses threads to handle requests by using the `ThreadingMixIn`. This is useful to handle web browsers pre-opening sockets, on which `HTTPServer` would wait indefinitely.

Added in version 3.7.

class http.server.HTTPSServer(*server_address*, *RequestHandlerClass*, *bind_and_activate=True*, *, *certfile*, *keyfile=None*, *password=None*, *alpn_protocols=None*)

Subclass of `HTTPServer` with a wrapped socket using the `ssl` module. If the `ssl` module is not available, instantiating a `HTTPSServer` object fails with a `RuntimeError`.

The *certfile* argument is the path to the SSL certificate chain file, and the *keyfile* is the path to file containing the private key.

A *password* can be specified for files protected and wrapped with PKCS#8, but beware that this could possibly expose hardcoded passwords in clear.

See also

See `ssl.SSLContext.load_cert_chain()` for additional information on the accepted values for *certfile*, *keyfile* and *password*.

When specified, the *alpn_protocols* argument must be a sequence of strings specifying the "Application-Layer Protocol Negotiation" (ALPN) protocols supported by the server. ALPN allows the server and the client to negotiate the application protocol during the TLS handshake.

By default, it is set to `["http/1.1"]`, meaning the server supports HTTP/1.1.

Added in version 3.14.

class http.server.ThreadingHTTPSServer(*server_address*, *RequestHandlerClass*, *bind_and_activate=True*, ***, *certfile*, *keyfile=None*, *password=None*, *alpn_protocols=None*)

This class is identical to `HTTPSServer` but uses threads to handle requests by inheriting from `ThreadingMixIn`. This is analogous to `ThreadingHTTPServer` only using `HTTPSServer`.

Added in version 3.14.

The `HTTPServer`, `ThreadingHTTPServer`, `HTTPSServer` and `ThreadingHTTPSServer` must be given a *RequestHandlerClass* on instantiation, of which this module provides three different variants:

class http.server.BaseHTTPRequestHandler(*request*, *client_address*, *server*)

This class is used to handle the HTTP requests that arrive at the server. By itself, it cannot respond to any actual HTTP requests; it must be subclassed to handle each request method (e.g. GET or POST). `BaseHTTPRequestHandler` provides a number of class and instance variables, and methods for use by subclasses.

The handler will parse the request and the headers, then call a method specific to the request type. The method name is constructed from the request. For example, for the request method `SPAM`, the **`do_SPAM()`** method will be called with no arguments. All of the relevant information is stored in instance variables of the handler. Subclasses should not need to override or extend the **`__init__()`** method.

[`BaseHTTPRequestHandler`] has the following instance variables:

client_address

Contains a tuple of the form `(host, port)` referring to the client's address.

server

Contains the server instance.

close_connection

Boolean that should be set before [`handle_one_request()`] returns, indicating if another request may be expected, or if the connection should be shut down.

requestline

Contains the string representation of the HTTP request line. The terminating CRLF is stripped. This attribute should be set by [`handle_one_request()`]. If no valid request line was processed, it should be set to the empty string.

command

Contains the command (request type). For example, `'GET'`.

path

Contains the request path. If query component of the URL is present, then `path` includes the query. Using the terminology of **RFC 3986** [https://datatracker.ietf.org/doc/html/rfc3986.html], `path` here includes `hier-part` and the `query`.

request_version

Contains the version string from the request. For example, `'HTTP/1.0'`.

headers

Holds an instance of the class specified by the **`MessageClass`** class variable. This instance parses and manages the headers in the HTTP request. The **`parse_headers()`** function from **`http.client`** is used to parse the headers and it requires that the HTTP request provide a valid **RFC 5322** [https://datatracker.ietf.org/doc/html/rfc5322.html] style header.

rfile

An **`io.BufferedIOBase`** input stream, ready to read from the start of the optional input data.

wfile

Contains the output stream for writing a response back to the client. Proper adherence to the HTTP protocol must be used when writing to this stream in order to achieve successful interoperation with HTTP clients.

Changed in version 3.6: This is an **`io.BufferedIOBase`** stream.

`BaseHTTPRequestHandler` has the following attributes:

server_version

Specifies the server software version. You may want to override this. The format is multiple whitespace-separated strings, where each string is of the form name[/version]. For example, `'BaseHTTP/0.2'`.

sys_version

Contains the Python system version, in a form usable by the `version_string` method and the `server_version` class variable. For example, `'Python/1.4'`.

error_message_format

Specifies a format string that should be used by `send_error()` method for building an error response to the client. The string is filled by default with variables from `responses` based on the status code that passed to `send_error()`.

error_content_type

Specifies the Content-Type HTTP header of error responses sent to the client. The default value is `'text/html'`.

protocol_version

Specifies the HTTP version to which the server is conformant. It is sent in responses to let the client know the server’s communication capabilities for future requests. If set to `'HTTP/1.1'`, the server will permit HTTP persistent connections; however, your server *must* then include an accurate `Content-Length` header (using `send_header()`) in all of its responses to clients. For backwards compatibility, the setting defaults to `'HTTP/1.0'`.

MessageClass

Specifies an `email.message.Message`-like class to parse HTTP headers. Typically, this is not overridden, and it defaults to `http.client.HTTPMessage`.

responses

This attribute contains a mapping of error code integers to two-element tuples containing a short and long message. For example, `{code: (shortmessage,` `longmessage)}`. The *shortmessage* is usually used as the *message* key in an error response, and *longmessage* as the *explain* key. It is used by `send_response_only()` and `send_error()` methods.

A `BaseHTTPRequestHandler` instance has the following methods:

handle()

Calls `handle_one_request()` once (or, if persistent connections are enabled, multiple times) to handle incoming HTTP requests. You should never need to override it; instead, implement appropriate **`do_*()`** methods.

handle_one_request()

This method will parse and dispatch the request to the appropriate **`do_*()`** method. You should never need to override it.

handle_expect_100()

When an HTTP/1.1 conformant server receives an `Expect: 100-continue` request header it responds back with a `100 Continue` followed by `200 OK` headers. This method can be overridden to raise an error if the server does not want the client to continue. For e.g. server can choose to send `417 Expectation Failed` as a response header and `return False`.

Added in version 3.2.

send_error(*code*, *message=None*, *explain=None*)

Sends and logs a complete error reply to the client. The numeric *code* specifies the HTTP error code, with *message* as an optional, short, human readable description of the error. The *explain* argument can be used to provide more detailed information about the error; it will be formatted using the **`error_message_format`** attribute and emitted, after a complete set of headers, as the response body. The **`responses`** attribute holds the default values for *message* and *explain* that will be used if no value is provided; for unknown codes the default value for both is the string `???`. The body will be empty if the method is HEAD or the response code is one of the following: `1xx`, `204 No Content`, `205 Reset Content`, `304 Not Modified`.

Changed in version 3.4: The error response includes a Content-Length header. Added the *explain* argument.

send_response(*code*, *message=None*)

Adds a response header to the headers buffer and logs the accepted request. The HTTP response line is written to the internal buffer, followed by *Server* and *Date* headers. The values for these two headers are picked up from the **`version_string()`** and **`date_time_string()`** methods, respectively. If the server does not intend to send any other headers using the **`send_header()`** method, then **`send_response()`** should be followed by an **`end_headers()`** call.

Changed in version 3.3: Headers are stored to an internal buffer and **`end_headers()`** needs to be called explicitly.

send_header(*keyword*, *value*)

Adds the HTTP header to an internal buffer which will be written to the output stream when either **`end_headers()`** or **`flush_headers()`** is invoked. *keyword* should specify the header keyword, with *value* specifying its value. Note that, after the send_header calls are done, [`end_headers()`] MUST BE called in order to complete the operation.

Changed in version 3.2: Headers are stored in an internal buffer.

send_response_only(*code*, *message=None*)

Sends the response header only, used for the purposes when `100 Continue` response is sent by the server to the client. The headers not buffered and sent directly the output stream.If the *message* is not specified, the HTTP message corresponding the response *code* is sent.

Added in version 3.2.

end_headers()

Adds a blank line (indicating the end of the HTTP headers in the response) to the headers buffer and calls [`flush_headers()`].

Changed in version 3.2: The buffered headers are written to the output stream.

flush_headers()

Finally send the headers to the output stream and flush the internal headers buffer.

Added in version 3.3.

log_request(*code='-'*, *size='-'*)

Logs an accepted (successful) request. *code* should specify the numeric HTTP code associated with the response. If a size of the response is available, then it should be passed as the *size* parameter.

log_error(*...*)

Logs an error when a request cannot be fulfilled. By default, it passes the message to `log_message()`, so it takes the same arguments (*format* and additional values).

log_message(*format*, *...*)

Logs an arbitrary message to `sys.stderr`. This is typically overridden to create custom error logging mechanisms. The *format* argument is a standard printf-style format string, where the additional arguments to `log_message()` are applied as inputs to the formatting. The client ip address and current date and time are prefixed to every message logged.

version_string()

Returns the server software's version string. This is a combination of the `server_version` and `sys_version` attributes.

date_time_string(*timestamp=None*)

Returns the date and time given by *timestamp* (which must be `None` or in the format returned by `time.time()`), formatted for a message header. If *timestamp* is omitted, it uses the current date and time.

The result looks like `'Sun, 06 Nov 1994 08:49:37 GMT'`.

log_date_time_string()

Returns the current date and time, formatted for logging.

address_string()

Returns the client address.

Changed in version 3.3: Previously, a name lookup was performed. To avoid name resolution delays, it now always returns the IP address.

class http.server.SimpleHTTPRequestHandler(*request*, *client_address*, *server*, *directory=None*)

This class serves files from the directory *directory* and below, or the current directory if *directory* is not provided, directly mapping the directory structure to HTTP requests.

Changed in version 3.7: Added the *directory* parameter.

Changed in version 3.9: The *directory* parameter accepts a path-like object.

A lot of the work, such as parsing the request, is done by the base class **`BaseHTTPRequestHandler`**. This class implements the **`do_GET()`** and **`do_HEAD()`** functions.

The following are defined as class-level attributes of **`SimpleHTTPRequestHandler`**:

server_version

This will be `"SimpleHTTP/" + __version__`, where `__version__` is defined at the module level.

extensions_map

A dictionary mapping suffixes into MIME types, contains custom overrides for the default system mappings. The mapping is used case-insensitively, and so should contain only lower-cased keys.

Changed in version 3.9: This dictionary is no longer filled with the default system mappings, but only contains overrides.

The `SimpleHTTPRequestHandler` class defines the following methods:

do_HEAD()

This method serves the `'HEAD'` request type: it sends the headers it would send for the equivalent `GET` request. See the `do_GET()` method for a more complete explanation of the possible headers.

do_GET()

The request is mapped to a local file by interpreting the request as a path relative to the current working directory.

If the request was mapped to a directory, the directory is checked for a file named `index.html` or `index.htm` (in that order). If found, the file's contents are returned; otherwise a directory listing is generated by calling the **`list_directory()`** method. This method uses `os.listdir()` to scan the directory, and returns a `404` error response if the `listdir()` fails.

If the request was mapped to a file, it is opened. Any `OSError` exception in opening the requested file is mapped to a `404`, `'File not found'` error. If there was an `'If-Modified-Since'` header in the request, and the file was not modified after this time, a `304`, `'Not Modified'` response is sent. Otherwise, the content type is guessed by calling the **`guess_type()`** method, which in turn uses the *extensions_map* variable, and the file contents are returned.

A `'Content-type:'` header with the guessed content type is output, followed by a `'Content-Length:'` header with the file's size and a `'Last-Modified:'` header with the file's modification time.

Then follows a blank line signifying the end of the headers, and then the contents of the file are output.

For example usage, see the implementation of the `test` function in Lib/http/server.py [https://github.com/python/cpython/tree/3.14/Lib/http/server.py].

Changed in version 3.7: Support of the `'If-Modified-Since'` header.

The **`SimpleHTTPRequestHandler`** class can be used in the following manner in order to create a very basic webserver serving files relative to the current directory:

```
import http.server
import socketserver

PORT = 8000

Handler = http.server.SimpleHTTPRequestHandler

with socketserver.TCPServer(("", PORT), Handler) as httpd:
    print("serving at port", PORT)
    httpd.serve_forever()
```

`SimpleHTTPRequestHandler` can also be subclassed to enhance behavior, such as using different index file names by overriding the class attribute **`index_pages`**.

class http.server.CGIHTTPRequestHandler(*request*, *client_address*, *server*)

This class is used to serve either files or output of CGI scripts from the current directory and below. Note that mapping HTTP hierarchic structure to local directory structure is exactly as in **`SimpleHTTPRequestHandler`**.

Note

CGI scripts run by the `CGIHTTPRequestHandler` class cannot execute redirects (HTTP code 302), because code 200 (script output follows) is sent prior to execution of the CGI script. This pre-empts the status code.

The class will however, run the CGI script, instead of serving it as a file, if it guesses it to be a CGI script. Only directory-based CGI are used — the other common server configuration is to treat special extensions as denoting CGI scripts.

The **`do_GET()`** and **`do_HEAD()`** functions are modified to run CGI scripts and serve the output, instead of serving files, if the request leads to somewhere below the `cgi_directories` path.

The `CGIHTTPRequestHandler` defines the following data member:

cgi_directories

This defaults to `['/cgi-bin', '/htbin']` and describes directories to treat as containing CGI scripts.

The `CGIHTTPRequestHandler` defines the following method:

do_POST()

This method serves the `'POST'` request type, only allowed for CGI scripts. Error 501, “Can only POST to CGI scripts”, is output when trying to POST to a non-CGI url.

Note that CGI scripts will be run with UID of user nobody, for security reasons. Problems with the CGI script will be translated to error 403.

Deprecated since version 3.13, will be removed in version 3.15: `CGIHTTPRequestHandler` is being removed in 3.15. CGI has not been considered a good way to do things for well over a decade. This code has been unmaintained for a while now and sees very little practical use. Retaining it could lead to further security considerations.

Command-line interface

`http.server` can also be invoked directly using the `-m` switch of the interpreter. The following example illustrates how to serve files relative to the current directory:

```
python -m http.server [OPTIONS] [port]
```

The following options are accepted:

port

The server listens to port 8000 by default. The default can be overridden by passing the desired port number as an argument:

```
python -m http.server 9000
```

-b, --bind <address>

Specifies a specific address to which it should bind. Both IPv4 and IPv6 addresses are supported. By default, the server binds itself to all interfaces. For example, the following command causes the server to bind to localhost only:

```
python -m http.server --bind 127.0.0.1
```

Added in version 3.4.

Changed in version 3.8: Support IPv6 in the `--bind` option.

-d, --directory <dir>

Specifies a directory to which it should serve the files. By default, the server uses the current directory. For example, the following command uses a specific directory:

```
python -m http.server --directory /tmp/
```

Added in version 3.7.

-p, --protocol <version>

Specifies the HTTP version to which the server is conformant. By default, the server is conformant to HTTP/1.0. For example, the following command runs an HTTP/1.1 conformant server:

```
python -m http.server --protocol HTTP/1.1
```

Added in version 3.11.

--cgi

`CGIHTTPRequestHandler` can be enabled in the command line by passing the `--cgi` option:

```
python -m http.server --cgi
```

Deprecated since version 3.13, will be removed in version 3.15: `http.server` command line `--cgi` support is being removed because `CGIHTTPRequestHandler` is being removed.

Warning

`CGIHTTPRequestHandler` and the `--cgi` command-line option are not intended for use by untrusted clients and may be vulnerable to exploitation. Always use within a secure environment.

--tls-cert

Specifies a TLS certificate chain for HTTPS connections:

```
python -m http.server --tls-cert fullchain.pem
```

Added in version 3.14.

--tls-key

Specifies a private key file for HTTPS connections.

This option requires `--tls-cert` to be specified.

Added in version 3.14.

--tls-password-file

Specifies the password file for password-protected private keys:

```
python -m http.server \
       --tls-cert cert.pem \
       --tls-key key.pem \
       --tls-password-file password.txt
```

This option requires *–tls-cert*` to be specified.

Added in version 3.14.

Security considerations

`SimpleHTTPRequestHandler` will follow symbolic links when handling requests, this makes it possible for files outside of the specified directory to be served.

Earlier versions of Python did not scrub control characters from the log messages emitted to stderr from `python -m http.server` or the default **`BaseHTTPRequestHandler`** `.log_message` implementation. This could allow remote clients connecting to your server to send nefarious control codes to your terminal.

Changed in version 3.12: Control characters are scrubbed in stderr logs.

`http.cookies` — HTTP state management

Source code: Lib/http/cookies.py
[https://github.com/python/cpython/tree/3.14/Lib/http/cookies.py]

The **`http.cookies`** module defines classes for abstracting the concept of cookies, an HTTP state management mechanism. It supports both simple string-only cookies, and provides an abstraction for having any serializable data-type as cookie value.

The module formerly strictly applied the parsing rules described in the **RFC 2109** [https://datatracker.ietf.org/doc/html/rfc2109.html] and **RFC 2068** [https://datatracker.ietf.org/doc/html/rfc2068.html] specifications. It has since been discovered that MSIE 3.0x didn’t follow the character rules outlined in those specs; many current-day browsers and servers have also relaxed parsing rules when it comes to cookie handling. As a result, this module now uses parsing rules that are a bit less strict than they once were.

The character set, `string.ascii_letters`, `string.digits` and `!#$%&'*+-.^_`|~:` denote the set of valid characters allowed by this module in a cookie name (as `key`).

Changed in version 3.3: Allowed ‘:’ as a valid cookie name character.

Note

On encountering an invalid cookie, `CookieError` is raised, so if your cookie data comes from a browser you should always prepare for invalid data and catch `CookieError` on parsing.

exception http.cookies.CookieError

Exception failing because of **RFC 2109** [https://datatracker.ietf.org/doc/html/rfc2109.html] invalidity: incorrect attributes, incorrect *Set-Cookie* header, etc.

class http.cookies.BaseCookie([*input*])

This class is a dictionary-like object whose keys are strings and whose values are `Morsel` instances. Note that upon setting a key to a value, the value is first converted to a `Morsel` containing the key and the value.

If *input* is given, it is passed to the `load()` method.

class http.cookies.SimpleCookie([*input*])

This class derives from `BaseCookie` and overrides `value_decode()` and `value_encode()`. **`SimpleCookie`** supports strings as cookie values. When setting the value, **`SimpleCookie`** calls the builtin `str()` to convert the value to a string. Values received from HTTP are kept as strings.

See also

Module `http.cookiejar`

HTTP cookie handling for web *clients*. The `http.cookiejar` and `http.cookies` modules do not depend on each other.

RFC 2109 [https://datatracker.ietf.org/doc/html/rfc2109.html] - HTTP State Management Mechanism

This is the state management specification implemented by this module.

Cookie Objects

BaseCookie.value_decode(*val*)

Return a tuple `(real_value, coded_value)` from a string representation. `real_value` can be any type. This method does no decoding in **`BaseCookie`** — it exists so it can be overridden.

BaseCookie.value_encode(*val*)

Return a tuple `(real_value, coded_value)`. *val* can be any type, but `coded_value` will always be converted to a string. This method does no encoding in **`BaseCookie`** — it exists so it can be overridden.

In general, it should be the case that **`value_encode()`** and **`value_decode()`** are inverses on the range of *value_decode*.

BaseCookie.output(*attrs=None*, *header='Set-Cookie:'*, *sep='\r\n'*)

Return a string representation suitable to be sent as HTTP headers. *attrs* and *header* are sent to each **`Morsel`**’s **`output()`** method. *sep* is used to join the headers together, and is by default the combination `'\r\n'` (CRLF).

BaseCookie.js_output(*attrs=None*)

Return an embeddable JavaScript snippet, which, if run on a browser which supports JavaScript, will act the same as if the HTTP headers was sent.

The meaning for *attrs* is the same as in **`output()`**.

BaseCookie.load(*rawdata*)

If *rawdata* is a string, parse it as an `HTTP_COOKIE` and add the values found there as **`Morsel`**s. If it is a dictionary, it is equivalent to:

```
for k, v in rawdata.items():
    cookie[k] = v
```

Morsel Objects

class http.cookies.Morsel

Abstract a key/value pair, which has some **RFC 2109** [https://datatracker.ietf.org/doc/html/rfc2109.html] attributes.

Morsels are dictionary-like objects, whose set of keys is constant — the valid **RFC 2109** [https://datatracker.ietf.org/doc/html/rfc2109.html] attributes, which are:

expires
path
comment
domain
max-age
secure
version
httponly
samesite
partitioned

The attribute **`httponly`** specifies that the cookie is only transferred in HTTP requests, and is not accessible through JavaScript. This is intended to mitigate some forms of cross-site scripting.

The attribute **`samesite`** controls when the browser sends the cookie with cross-site requests. This helps to mitigate

CSRF attacks. Valid values are “Strict” (only sent with same-site requests), “Lax” (sent with same-site requests and top-level navigations), and “None” (sent with same-site and cross-site requests). When using “None”, the “secure” attribute must also be set, as required by modern browsers.

The attribute **`partitioned`** indicates to user agents that these cross-site cookies *should* only be available in the same top-level context that the cookie was first set in. For this to be accepted by the user agent, you **must** also set `Secure`.

In addition, it is recommended to use the `__Host` prefix when setting partitioned cookies to make them bound to the hostname and not the registrable domain. Read CHIPS (Cookies Having Independent Partitioned State) [https://github.com/privacycg/CHIPS/blob/main/README.md] for full details and examples.

The keys are case-insensitive and their default value is `''`.

Changed in version 3.5: **`__eq__()`** now takes **`key`** and **`value`** into account.

Changed in version 3.7: Attributes **`key`**, **`value`** and **`coded_value`** are read-only. Use **`set()`** for setting them.

Changed in version 3.8: Added support for the **`samesite`** attribute.

Changed in version 3.14: Added support for the **`partitioned`** attribute.

Morsel.value

The value of the cookie.

Morsel.coded_value

The encoded value of the cookie — this is what should be sent.

Morsel.key

The name of the cookie.

Morsel.set(*key*, *value*, *coded_value*)

Set the *key*, *value* and *coded_value* attributes.

Morsel.isReservedKey(*K*)

Whether *K* is a member of the set of keys of a `Morsel`.

Morsel.output(*attrs=None*, *header='Set-Cookie:'*)

Return a string representation of the Morsel, suitable to be sent as an HTTP header. By default, all the attributes are included, unless *attrs* is given, in which case it should be a list of attributes to use. *header* is by default `"Set-Cookie:"`.

Morsel.js_output(*attrs=None*)

Return an embeddable JavaScript snippet, which, if run on a browser which supports JavaScript, will act the same as if the HTTP header was sent.

The meaning for *attrs* is the same as in `output()`.

Morsel.OutputString(*attrs=None*)

Return a string representing the Morsel, without any surrounding HTTP or JavaScript.

The meaning for *attrs* is the same as in `output()`.

Morsel.update(*values*)

Update the values in the Morsel dictionary with the values in the dictionary *values*. Raise an error if any of the keys in the *values* dict is not a valid **RFC 2109** [https://datatracker.ietf.org/doc/html/rfc2109.html] attribute.

Changed in version 3.5: an error is raised for invalid keys.

Morsel.copy(*value*)

Return a shallow copy of the Morsel object.

Changed in version 3.5: return a Morsel object instead of a dict.

Morsel.setdefault(*key*, *value=None*)

Raise an error if key is not a valid **RFC 2109** [https://datatracker.ietf.org/doc/html/rfc2109.html] attribute, otherwise behave the same as `dict.setdefault()`.

Example

The following example demonstrates how to use the `http.cookies` module.

```
>>> from http import cookies
>>> C = cookies.SimpleCookie()
>>> C["fig"] = "newton"
>>> C["sugar"] = "wafer"
>>> print(C) # generate HTTP headers
Set-Cookie: fig=newton
Set-Cookie: sugar=wafer
>>> print(C.output()) # same thing
Set-Cookie: fig=newton
Set-Cookie: sugar=wafer
>>> C = cookies.SimpleCookie()
>>> C["rocky"] = "road"
>>> C["rocky"]["path"] = "/cookie"
>>> print(C.output(header="Cookie:"))
Cookie: rocky=road; Path=/cookie
```

```
>>> print(C.output(attrs=[], header="Cookie:"))
Cookie: rocky=road
>>> C = cookies.SimpleCookie()
>>> C.load("chips=ahoy; vienna=finger") # load from a string
(HTTP header)
>>> print(C)
Set-Cookie: chips=ahoy
Set-Cookie: vienna=finger
>>> C = cookies.SimpleCookie()
>>> C.load('keebler="E=everybody; L=\\"Loves\\";
fudge=\\012;";')
>>> print(C)
Set-Cookie: keebler="E=everybody; L=\"Loves\"; fudge=\012;"
>>> C = cookies.SimpleCookie()
>>> C["oreo"] = "doublestuff"
>>> C["oreo"]["path"] = "/"
>>> print(C)
Set-Cookie: oreo=doublestuff; Path=/
>>> C = cookies.SimpleCookie()
>>> C["twix"] = "none for you"
>>> C["twix"].value
'none for you'
>>> C = cookies.SimpleCookie()
>>> C["number"] = 7 # equivalent to C["number"] = str(7)
>>> C["string"] = "seven"
>>> C["number"].value
'7'
>>> C["string"].value
'seven'
>>> print(C)
Set-Cookie: number=7
Set-Cookie: string=seven
```

`http.cookiejar` — Cookie handling for HTTP clients

Source code: Lib/http/cookiejar.py [https://github.com/python/cpython/tree/3.14/Lib/http/cookiejar.py]

The `http.cookiejar` module defines classes for automatic handling of HTTP cookies. It is useful for accessing websites that require small pieces of data – *cookies* – to be set on the client machine by an HTTP response from a web server, and then returned to the server in later HTTP requests.

Both the regular Netscape cookie protocol and the protocol defined by **RFC 2965** [https://datatracker.ietf.org/doc/html/rfc2965.html] are handled. RFC 2965 handling is switched off by default. **RFC 2109** [https://datatracker.ietf.org/doc/html/rfc2109.html] cookies are parsed as Netscape cookies and subsequently treated either as Netscape or RFC 2965 cookies according to the 'policy' in effect. Note that the great majority of cookies on the internet are Netscape cookies. `http.cookiejar` attempts to follow the de-facto Netscape cookie protocol (which differs substantially from that set out in the original Netscape specification), including taking note of the `max-age` and `port` cookie-attributes introduced with RFC 2965.

Note

The various named parameters found in *Set-Cookie* and *Set-Cookie2* headers (eg. `domain` and `expires`) are conventionally referred to as *attributes*. To distinguish them from Python attributes, the documentation for this module uses the term *cookie-attribute* instead.

The module defines the following exception:

exception http.cookiejar.LoadError

Instances of `FileCookieJar` raise this exception on failure to load cookies from a file. `LoadError` is a subclass of `OSError`.

Changed in version 3.3: `LoadError` used to be a subtype of `IOError`, which is now an alias of `OSError`.

The following classes are provided:

class http.cookiejar.CookieJar(*policy=None*)

policy is an object implementing the `CookiePolicy` interface.

The `CookieJar` class stores HTTP cookies. It extracts cookies from HTTP requests, and returns them in HTTP responses. `CookieJar` instances automatically expire contained cookies when necessary. Subclasses are also responsible for storing and retrieving cookies from a file or database.

class http.cookiejar.FileCookieJar(*filename=None*, *delayload=None*, *policy=None*)

policy is an object implementing the `CookiePolicy` interface. For the other arguments, see the documentation for the corresponding attributes.

A `CookieJar` which can load cookies from, and perhaps save cookies to, a file on disk. Cookies are **NOT** loaded from the named file until either the `load()` or `revert()`

method is called. Subclasses of this class are documented in section FileCookieJar subclasses and co-operation with web browsers.

This should not be initialized directly – use its subclasses below instead.

Changed in version 3.8: The filename parameter supports a path-like object.

class http.cookiejar.CookiePolicy

This class is responsible for deciding whether each cookie should be accepted from / returned to the server.

class http.cookiejar.DefaultCookiePolicy(*blocked_domains=None*, *allowed_domains=None*, *netscape=True*, *rfc2965=False*, *rfc2109_as_netscape=None*, *hide_cookie2=False*, *strict_domain=False*, *strict_rfc2965_unverifiable=True*, *strict_ns_unverifiable=False*, *strict_ns_domain=DefaultCookiePolicy.DomainLiberal*, *strict_ns_set_initial_dollar=False*, *strict_ns_set_path=False*, *secure_protocols=('https', 'wss')*)

Constructor arguments should be passed as keyword arguments only. *blocked_domains* is a sequence of domain names that we never accept cookies from, nor return cookies to. *allowed_domains* if not `None`, this is a sequence of the only domains for which we accept and return cookies. *secure_protocols* is a sequence of protocols for which secure cookies can be added to. By default *https* and *wss* (secure websocket) are considered secure protocols. For all other arguments, see the documentation for `CookiePolicy` and `DefaultCookiePolicy` objects.

`DefaultCookiePolicy` implements the standard accept / reject rules for Netscape and **RFC 2965** [https://datatracker.ietf.org/doc/html/rfc2965.html] cookies. By default, **RFC 2109** [https://datatracker.ietf.org/doc/html/rfc2109.html] cookies (ie. cookies received in a *Set-Cookie* header with a version cookie-attribute of 1) are treated according to the RFC 2965 rules. However, if RFC 2965 handling is turned off or `rfc2109_as_netscape` is `True`, RFC 2109 cookies are 'downgraded' by the `CookieJar` instance to Netscape cookies, by setting the **`version`** attribute of the `Cookie` instance to 0. `DefaultCookiePolicy` also provides some parameters to allow some fine-tuning of policy.

class http.cookiejar.Cookie

This class represents Netscape, **RFC 2109** [https://datatracker.ietf.org/doc/html/rfc2109.html] and **RFC 2965** [https://datatracker.ietf.org/doc/html/rfc2965.html] cookies. It is not expected that users of `http.cookiejar` construct their own `Cookie` instances. Instead, if necessary, call **`make_cookies()`** on a `CookieJar` instance.

See also

Module `urllib.request`

URL opening with automatic cookie handling.

Module `http.cookies`

HTTP cookie classes, principally useful for server-side code. The `http.cookiejar` and `http.cookies` modules do not depend on each other.

https://curl.se/rfc/cookie_spec.html

The specification of the original Netscape cookie protocol. Though this is still the dominant protocol, the 'Netscape cookie protocol' implemented by all the major browsers (and `http.cookiejar`) only bears a passing resemblance to the one sketched out in `cookie_spec.html`.

RFC 2109 [https://datatracker.ietf.org/doc/html/rfc2109.html] - HTTP State Management Mechanism

Obsoleted by **RFC 2965** [https://datatracker.ietf.org/doc/html/rfc2965.html]. Uses *Set-Cookie* with version=1.

RFC 2965 [https://datatracker.ietf.org/doc/html/rfc2965.html] - HTTP State Management Mechanism

The Netscape protocol with the bugs fixed. Uses *Set-Cookie2* in place of *Set-Cookie*. Not widely used.

https://kristol.org/cookie/errata.html

Unfinished errata to **RFC 2965** [https://datatracker.ietf.org/doc/html/rfc2965.html].

RFC 2964 [https://datatracker.ietf.org/doc/html/rfc2964.html] - Use of HTTP State Management

CookieJar and FileCookieJar Objects

`CookieJar` objects support the iterator protocol for iterating over contained `Cookie` objects.

`CookieJar` has the following methods:

CookieJar.add_cookie_header(*request*)

Add correct *Cookie* header to *request*.

If policy allows (ie. the **`rfc2965`** and **`hide_cookie2`** attributes of the `CookieJar`’s `CookiePolicy` instance are true and false respectively), the *Cookie2* header is also added when appropriate.

The *request* object (usually a `urllib.request.Request` instance) must support the methods **`get_full_url()`**, **`has_header()`**, **`get_header()`**, **`header_items()`**, **`add_unredirected_header()`** and the attributes **`host`**, **`type`**, **`unverifiable`** and **`origin_req_host`** as documented by `urllib.request`.

Changed in version 3.3: request object needs **`origin_req_host`** attribute. Dependency on a deprecated method **`get_origin_req_host()`** has been removed.

CookieJar.extract_cookies(*response*, *request*)

Extract cookies from HTTP *response* and store them in the `CookieJar`, where allowed by policy.

The `CookieJar` will look for allowable *Set-Cookie* and *Set-Cookie2* headers in the *response* argument, and store cookies as appropriate (subject to the `CookiePolicy.set_ok()` method's approval).

The *response* object (usually the result of a call to `urllib.request.urlopen()`, or similar) should support an **`info()`** method, which returns an `email.message.Message` instance.

The *request* object (usually a `urllib.request.Request` instance) must support the method **`get_full_url()`** and the attributes **`host`**, **`unverifiable`** and **`origin_req_host`**, as documented by `urllib.request`. The request is used to set default values for cookie-attributes as well as for checking that the cookie is allowed to be set.

Changed in version 3.3: request object needs **`origin_req_host`** attribute. Dependency on a deprecated method **`get_origin_req_host()`** has been removed.

CookieJar.set_policy(*policy*)

Set the `CookiePolicy` instance to be used.

CookieJar.make_cookies(*response*, *request*)

Return sequence of `Cookie` objects extracted from *response* object.

See the documentation for `extract_cookies()` for the interfaces required of the *response* and *request* arguments.

CookieJar.set_cookie_if_ok(*cookie*, *request*)

Set a `Cookie` if policy says it's OK to do so.

CookieJar.set_cookie(*cookie*)

Set a `Cookie`, without checking with policy to see whether or not it should be set.

CookieJar.clear([*domain*[, *path*[, *name*]]])

Clear some cookies.

If invoked without arguments, clear all cookies. If given a single argument, only cookies belonging to that *domain* will be removed. If given two arguments, cookies belonging to the specified *domain* and URL *path* are removed. If given three arguments, then the cookie with the specified *domain*, *path* and *name* is removed.

Raises `KeyError` if no matching cookie exists.

CookieJar.clear_session_cookies()

Discard all session cookies.

Discards all contained cookies that have a true **`discard`** attribute (usually because they had either no `max-age` or `expires` cookie-attribute, or an explicit `discard` cookie-attribute). For interactive browsers, the end of a session usually corresponds to closing the browser window.

Note that the **`save()`** method won’t save session cookies anyway, unless you ask otherwise by passing a true *ignore_discard* argument.

`FileCookieJar` implements the following additional methods:

FileCookieJar.save(*filename=None*, *ignore_discard=False*, *ignore_expires=False*)

Save cookies to a file.

This base class raises **`NotImplementedError`**. Subclasses may leave this method unimplemented.

filename is the name of file in which to save cookies. If *filename* is not specified, **`self.filename`** is used (whose default is the value passed to the constructor, if any); if **`self.filename`** is **`None`**, **`ValueError`** is raised.

ignore_discard: save even cookies set to be discarded. *ignore_expires*: save even cookies that have expired

The file is overwritten if it already exists, thus wiping all the cookies it contains. Saved cookies can be restored later using the **`load()`** or **`revert()`** methods.

FileCookieJar.load(*filename=None*, *ignore_discard=False*, *ignore_expires=False*)

Load cookies from a file.

Old cookies are kept unless overwritten by newly loaded ones.

Arguments are as for `save()`.

The named file must be in the format understood by the class, or `LoadError` will be raised. Also, `OSError` may be raised, for example if the file does not exist.

Changed in version 3.3: `IOError` used to be raised, it is now an alias of `OSError`.

FileCookieJar.revert(*filename=None*, *ignore_discard=False*, *ignore_expires=False*)

Clear all cookies and reload cookies from a saved file.

`revert()` can raise the same exceptions as `load()`. If there is a failure, the object's state will not be altered.

`FileCookieJar` instances have the following public attributes:

FileCookieJar.filename

Filename of default file in which to keep cookies. This attribute may be assigned to.

FileCookieJar.delayload

If true, load cookies lazily from disk. This attribute should not be assigned to. This is only a hint, since this only affects performance, not behaviour (unless the cookies on disk are changing). A `CookieJar` object may ignore it. None of the `FileCookieJar` classes included in the standard library lazily loads cookies.

FileCookieJar subclasses and co-operation with web browsers

The following `CookieJar` subclasses are provided for reading and writing.

class http.cookiejar.MozillaCookieJar(*filename=None*, *delayload=None*, *policy=None*)

A **`FileCookieJar`** that can load from and save cookies to disk in the Mozilla `cookies.txt` file format (which is also used by curl and the Lynx and Netscape browsers).

Note

This loses information about **RFC 2965** [https://datatracker.ietf.org/doc/html/rfc2965.html] cookies, and also about newer or non-standard cookie-attributes such as `port`.

Warning

Back up your cookies before saving if you have cookies whose loss / corruption would be inconvenient (there are some subtleties which may lead to slight changes in the file over a load / save round-trip).

Also note that cookies saved while Mozilla is running will get clobbered by Mozilla.

class http.cookiejar.LWPCookieJar(*filename=None*, *delayload=None*, *policy=None*)

A **`FileCookieJar`** that can load from and save cookies to disk in format compatible with the libwww-perl library's `Set-Cookie3` file format. This is convenient if you want to store cookies in a human-readable file.

Changed in version 3.8: The filename parameter supports a path-like object.

CookiePolicy Objects

Objects implementing the `CookiePolicy` interface have the following methods:

CookiePolicy.set_ok(*cookie*, *request*)

Return boolean value indicating whether cookie should be accepted from server.

cookie is a `Cookie` instance. *request* is an object implementing the interface defined by the documentation for `CookieJar.extract_cookies()`.

CookiePolicy.return_ok(*cookie*, *request*)

Return boolean value indicating whether cookie should be returned to server.

cookie is a `Cookie` instance. *request* is an object implementing the interface defined by the documentation for `CookieJar.add_cookie_header()`.

CookiePolicy.domain_return_ok(*domain*, *request*)

Return `False` if cookies should not be returned, given cookie domain.

This method is an optimization. It removes the need for checking every cookie with a particular domain (which might involve reading many files). Returning true from `domain_return_ok()` and `path_return_ok()` leaves all the work to `return_ok()`.

If `domain_return_ok()` returns true for the cookie domain, `path_return_ok()` is called for the cookie path. Otherwise, `path_return_ok()` and `return_ok()` are never called for that cookie domain. If `path_return_ok()` returns true, `return_ok()` is called with the `Cookie` object itself for a full check. Otherwise, `return_ok()` is never called for that cookie path.

Note that `domain_return_ok()` is called for every *cookie* domain, not just for the *request* domain. For example, the function might be called with both `".example.com"` and `"www.example.com"` if the request domain is `"www.example.com"`. The same goes for `path_return_ok()`.

The *request* argument is as documented for `return_ok()`.

CookiePolicy.path_return_ok(*path*, *request*)

Return `False` if cookies should not be returned, given cookie path.

See the documentation for `domain_return_ok()`.

In addition to implementing the methods above, implementations of the `CookiePolicy` interface must also supply the following attributes, indicating which protocols should be used, and how. All of these attributes may be assigned to.

CookiePolicy.netscape

Implement Netscape protocol.

CookiePolicy.rfc2965

Implement **RFC 2965** [https://datatracker.ietf.org/doc/html/rfc2965.html] protocol.

CookiePolicy.hide_cookie2

Don’t add *Cookie2* header to requests (the presence of this header indicates to the server that we understand **RFC 2965** [https://datatracker.ietf.org/doc/html/rfc2965.html] cookies).

The most useful way to define a `CookiePolicy` class is by subclassing from `DefaultCookiePolicy` and overriding some or all of the methods above. `CookiePolicy` itself may be used as a ‘null policy’ to allow setting and receiving any and all cookies (this is unlikely to be useful).

DefaultCookiePolicy Objects

Implements the standard rules for accepting and returning cookies.

Both **RFC 2965** [https://datatracker.ietf.org/doc/html/rfc2965.html] and Netscape cookies are covered. RFC 2965 handling is switched off by default.

The easiest way to provide your own policy is to override this class and call its methods in your overridden implementations before adding your own additional checks:

```
import http.cookiejar
class MyCookiePolicy(http.cookiejar.DefaultCookiePolicy):
    def set_ok(self, cookie, request):
        if not http.cookiejar.DefaultCookiePolicy.set_ok(self,
cookie, request):
            return False
        if i_dont_want_to_store_this_cookie(cookie):
            return False
        return True
```

In addition to the features required to implement the `CookiePolicy` interface, this class allows you to block and allow domains from setting and receiving cookies. There are also some strictness switches that allow you to tighten up the rather loose Netscape protocol rules a little bit (at the cost of blocking some benign cookies).

A domain blocklist and allowlist is provided (both off by default). Only domains not in the blocklist and present in the allowlist (if the allowlist is active) participate in cookie setting and returning. Use the *blocked_domains* constructor argument, and **`blocked_domains()`** and **`set_blocked_domains()`** methods (and the corresponding argument and methods for *allowed_domains*). If you set an allowlist, you can turn it off again by setting it to `None`.

Domains in block or allow lists that do not start with a dot must equal the cookie domain to be matched. For example, `"example.com"` matches a blocklist entry of `"example.com"`, but `"www.example.com"` does not. Domains that do start with a dot are matched by more specific domains too. For example, both `"www.example.com"` and `"www.coyote.example.com"` match `".example.com"` (but `"example.com"` itself does not). IP addresses are an exception, and must match exactly. For example, if blocked_domains contains `"192.168.1.2"` and `".168.1.2"`, 192.168.1.2 is blocked, but 193.168.1.2 is not.

`DefaultCookiePolicy` implements the following additional methods:

DefaultCookiePolicy.blocked_domains()

Return the sequence of blocked domains (as a tuple).

DefaultCookiePolicy.set_blocked_domains(*blocked_domains*)

Set the sequence of blocked domains.

DefaultCookiePolicy.is_blocked(*domain*)

Return `True` if *domain* is on the blocklist for setting or receiving cookies.

DefaultCookiePolicy.allowed_domains()

Return `None`, or the sequence of allowed domains (as a tuple).

DefaultCookiePolicy.set_allowed_domains(*allowed_domains*)

Set the sequence of allowed domains, or `None`.

DefaultCookiePolicy.is_not_allowed(*domain*)

Return `True` if *domain* is not on the allowlist for setting or receiving cookies.

`DefaultCookiePolicy` instances have the following attributes, which are all initialised from the constructor arguments of the same name, and which may all be assigned to.

DefaultCookiePolicy.rfc2109_as_netscape

If true, request that the `CookieJar` instance downgrade **RFC 2109** [https://datatracker.ietf.org/doc/html/rfc2109.html] cookies (ie. cookies received in a *Set-Cookie* header with a version cookie-attribute of 1) to Netscape cookies by setting the version attribute of the `Cookie` instance to 0. The default value is `None`, in which case RFC 2109 cookies are downgraded if and only if **RFC 2965** [https://datatracker.ietf.org/doc/html/rfc2965.html] handling is turned off. Therefore, RFC 2109 cookies are downgraded by default.

General strictness switches:

DefaultCookiePolicy.strict_domain

Don't allow sites to set two-component domains with country-code top-level domains like `.co.uk`, `.gov.uk`, `.co.nz`.etc. This is far from perfect and isn't guaranteed to work!

RFC 2965 [https://datatracker.ietf.org/doc/html/rfc2965.html] protocol strictness switches:

DefaultCookiePolicy.strict_rfc2965_unverifiable

Follow **RFC 2965** [https://datatracker.ietf.org/doc/html/rfc2965.html] rules on unverifiable transactions (usually, an unverifiable transaction is one resulting from a redirect or a request for an image hosted on another site). If this is false, cookies are *never* blocked on the basis of verifiability

Netscape protocol strictness switches:

DefaultCookiePolicy.strict_ns_unverifiable

Apply **RFC 2965** [https://datatracker.ietf.org/doc/html/rfc2965.html] rules on unverifiable transactions even to Netscape cookies.

DefaultCookiePolicy.strict_ns_domain

Flags indicating how strict to be with domain-matching rules for Netscape cookies. See below for acceptable values.

DefaultCookiePolicy.strict_ns_set_initial_dollar

Ignore cookies in Set-Cookie: headers that have names starting with `'$'`.

DefaultCookiePolicy.strict_ns_set_path

Don’t allow setting cookies whose path doesn’t path-match request URI.

`strict_ns_domain` is a collection of flags. Its value is constructed by or-ing together (for example, `DomainStrictNoDots|DomainStrictNonDomain` means both flags are set).

DefaultCookiePolicy.DomainStrictNoDots

When setting cookies, the ‘host prefix’ must not contain a dot (eg. `www.foo.bar.com` can’t set a cookie for `.bar.com`, because `www.foo` contains a dot).

DefaultCookiePolicy.DomainStrictNonDomain

Cookies that did not explicitly specify a `domain` cookie-attribute can only be returned to a domain equal to the domain that set the cookie (eg. `spam.example.com` won’t be returned cookies from `example.com` that had no `domain` cookie-attribute).

DefaultCookiePolicy.DomainRFC2965Match
When setting cookies, require a full **RFC 2965** [https://datatracker.ietf.org/doc/html/rfc2965.html] domain-match.

The following attributes are provided for convenience, and are the most useful combinations of the above flags:

DefaultCookiePolicy.DomainLiberal
Equivalent to 0 (ie. all of the above Netscape domain strictness flags switched off).

DefaultCookiePolicy.DomainStrict
Equivalent to `DomainStrictNoDots|DomainStrictNonDomain`.

Cookie Objects

`Cookie` instances have Python attributes roughly corresponding to the standard cookie-attributes specified in the various cookie standards. The correspondence is not one-to-one, because there are complicated rules for assigning default values, because the `max-age` and `expires` cookie-attributes contain equivalent information, and because **RFC 2109** [https://datatracker.ietf.org/doc/html/rfc2109.html] cookies may be ‘downgraded’ by `http.cookiejar` from version 1 to version 0 (Netscape) cookies.

Assignment to these attributes should not be necessary other than in rare circumstances in a `CookiePolicy` method. The class does not enforce internal consistency, so you should know what you’re doing if you do that.

Cookie.version

Integer or `None`. Netscape cookies have `version` 0. **RFC 2965** [https://datatracker.ietf.org/doc/html/rfc2965.html] and **RFC 2109** [https://datatracker.ietf.org/doc/html/rfc2109.html] cookies have a `version` cookie-attribute of 1. However, note that `http.cookiejar` may ‘downgrade’ RFC 2109 cookies to Netscape cookies, in which case `version` is 0.

Cookie.name

Cookie name (a string).

Cookie.value

Cookie value (a string), or `None`.

Cookie.port

String representing a port or a set of ports (eg. ‘80’, or ‘80,8080’), or `None`.

Cookie.domain

Cookie domain (a string).

Cookie.path

Cookie path (a string, eg. `'/acme/rocket_launchers'`).

Cookie.secure

`True` if cookie should only be returned over a secure connection.

Cookie.expires

Integer expiry date in seconds since epoch, or `None`. See also the `is_expired()` method.

Cookie.discard

`True` if this is a session cookie.

Cookie.comment

String comment from the server explaining the function of this cookie, or `None`.

Cookie.comment_url

URL linking to a comment from the server explaining the function of this cookie, or `None`.

Cookie.rfc2109

`True` if this cookie was received as an **RFC 2109** [https://datatracker.ietf.org/doc/html/rfc2109.html] cookie (ie. the cookie arrived in a *Set-Cookie* header, and the value of the Version cookie-attribute in that header was 1). This attribute is provided because `http.cookiejar` may ‘downgrade’ RFC 2109 cookies to Netscape cookies, in which case `version` is 0.

Cookie.port_specified

`True` if a port or set of ports was explicitly specified by the server (in the *Set-Cookie* / *Set-Cookie2* header).

Cookie.domain_specified

`True` if a domain was explicitly specified by the server.

Cookie.domain_initial_dot

`True` if the domain explicitly specified by the server began with a dot (`'.'`).

Cookies may have additional non-standard cookie-attributes. These may be accessed using the following methods:

Cookie.has_nonstandard_attr(*name*)

Return `True` if cookie has the named cookie-attribute.

Cookie.get_nonstandard_attr(*name*, *default=None*)

If cookie has the named cookie-attribute, return its value. Otherwise, return *default*.

Cookie.set_nonstandard_attr(*name*, *value*)

Set the value of the named cookie-attribute.

The `Cookie` class also defines the following method:

Cookie.is_expired(*now=None*)

`True` if cookie has passed the time at which the server requested it should expire. If *now* is given (in seconds since the epoch), return whether the cookie has expired at the specified time.

Examples

The first example shows the most common usage of `http.cookiejar`:

```
import http.cookiejar, urllib.request
cj = http.cookiejar.CookieJar()
opener =
urllib.request.build_opener(urllib.request.HTTPCookieProcessor(
cj))
r = opener.open("http://example.com/")
```

This example illustrates how to open a URL using your Netscape, Mozilla, or Lynx cookies (assumes Unix/Netscape convention for location of the cookies file):

```
import os, http.cookiejar, urllib.request
cj = http.cookiejar.MozillaCookieJar()
cj.load(os.path.join(os.path.expanduser("~"), ".netscape",
"cookies.txt"))
opener =
urllib.request.build_opener(urllib.request.HTTPCookieProcessor(
cj))
r = opener.open("http://example.com/")
```

The next example illustrates the use of `DefaultCookiePolicy`. Turn on **RFC 2965** [https://datatracker.ietf.org/doc/html/rfc2965.html] cookies, be more strict about domains when setting and returning Netscape cookies, and block some domains from setting cookies or having them returned:

```
import urllib.request
from http.cookiejar import CookieJar, DefaultCookiePolicy
policy = DefaultCookiePolicy(
    rfc2965=True, strict_ns_domain=Policy.DomainStrict,
    blocked_domains=["ads.net", ".ads.net"])
cj = CookieJar(policy)
opener =
urllib.request.build_opener(urllib.request.HTTPCookieProcessor(
cj))
r = opener.open("http://example.com/")
```

`xmlrpc` — XMLRPC server and client modules

XML-RPC is a Remote Procedure Call method that uses XML passed via HTTP as a transport. With it, a client can call methods with parameters on a remote server (the server is named by a URI) and get back structured data.

`xmlrpc` is a package that collects server and client modules implementing XML-RPC. The modules are:

- `xmlrpc.client`
- `xmlrpc.server`

`xmlrpc.client` — XML-RPC client access

Source code: Lib/xmlrpc/client.py [https://github.com/python/cpython/tree/3.14/Lib/xmlrpc/client.py]

XML-RPC is a Remote Procedure Call method that uses XML passed via HTTP(S) as a transport. With it, a client can call methods with parameters on a remote server (the server is named by a URI) and get back structured data. This module supports writing XML-RPC client code; it handles all the details of translating between conformable Python objects and XML on the wire.

Warning

The **`xmlrpc.client`** module is not secure against maliciously constructed data. If you need to parse untrusted or unauthenticated data, see XML security.

Changed in version 3.5: For HTTPS URIs, **`xmlrpc.client`** now performs all the necessary certificate and hostname checks by default.

Availability: not WASI.

This module does not work or is not available on WebAssembly. See WebAssembly platforms for more information.

class xmlrpc.client.ServerProxy(*uri*, *transport=None*, *encoding=None*, *verbose=False*, *allow_none=False*, *use_datetime=False*, *use_builtin_types=False*, *, *headers=()*, *context=None*)

A `ServerProxy` instance is an object that manages communication with a remote XML-RPC server. The required first argument is a URI (Uniform Resource Indicator), and will normally be the URL of the server. The optional second argument is a transport factory instance; by default it is an internal **`SafeTransport`** instance for https: URLs and an internal HTTP **`Transport`** instance otherwise. The optional third argument is an encoding, by default UTF-8. The optional fourth argument is a debugging flag.

The following parameters govern the use of the returned proxy instance. If *allow_none* is true, the Python constant `None` will be translated into XML; the default behaviour is for `None` to raise a `TypeError`. This is a commonly used extension to the XML-RPC specification, but isn’t supported by all clients and servers; see http://ontosys.com/xml-rpc/extensions.php [https://web.archive.org/web/20130120074804/http://ontosys.com/xml-rpc/extensions.php] for a description. The *use_builtin_types* flag can be used to cause date/time values to be presented as `datetime.datetime` objects and binary data to be presented as `bytes` objects; this flag is false by default. `datetime.datetime`, `bytes` and `bytearray` objects may be passed to calls. The *headers* parameter is an optional sequence of HTTP headers to send with each request, expressed as a sequence of 2-tuples representing the header name and value. (e.g. `[('Header-Name', 'value')]`). If an HTTPS URL is provided, *context* may be `ssl.SSLContext` and configures the SSL settings of the underlying HTTPS connection. The obsolete *use_datetime* flag is similar to *use_builtin_types* but it applies only to date/time values.

Changed in version 3.3: The *use_builtin_types* flag was added.

Changed in version 3.8: The *headers* parameter was added.

Both the HTTP and HTTPS transports support the URL syntax extension for HTTP Basic Authentication: `http://user:pass@host:port/path`. The `user:pass` portion will be base64-encoded as an HTTP ‘Authorization’ header, and sent to the remote server as part of the connection process when invoking an XML-RPC method. You only need to use this if the remote server requires a Basic Authentication user and password.

The returned instance is a proxy object with methods that can be used to invoke corresponding RPC calls on the remote server. If the remote server supports the introspection API, the proxy can also be used to query the remote server for the methods it supports (service discovery) and fetch other server-associated metadata.

Types that are conformable (e.g. that can be marshalled through XML), include the following (and except where noted, they are unmarshalled as the same Python type):

XML-RPC type	Python type
`boolean`	**`bool`**
`int`, `i1`, `i2`, `i4`, `i8` or `biginteger`	**`int`** in range from -2147483648 to 2147483647. Values get the `<int>` tag.

XML-RPC type	Python type
`double` or `float`	`float`. Values get the `<double>` tag.
`string`	`str`
`array`	`list` or `tuple` containing conformable elements. Arrays are returned as `lists`.
`struct`	`dict`. Keys must be strings, values may be any conformable type. Objects of user-defined classes can be passed in; only their `__dict__` attribute is transmitted.
`dateTime.iso8601`	`DateTime` or `datetime.datetime`. Returned type depends on values of *use_builtin_types* and *use_datetime* flags.
`base64`	`Binary`, `bytes` or `bytearray`. Returned type depends on the value of the *use_builtin_types* flag.
`nil`	The `None` constant. Passing is allowed only if *allow_none* is true.
`bigdecimal`	`decimal.Decimal`. Returned type only.

This is the full set of data types supported by XML-RPC. Method calls may also raise a special `Fault` instance, used to signal XML-RPC server errors, or `ProtocolError` used to signal an error in the HTTP/HTTPS transport layer. Both `Fault` and `ProtocolError` derive from a base class called

`Error`. Note that the xmlrpc client module currently does not marshal instances of subclasses of built-in types.

When passing strings, characters special to XML such as `<`, `>`, and `&` will be automatically escaped. However, it’s the caller’s responsibility to ensure that the string is free of characters that aren’t allowed in XML, such as the control characters with ASCII values between 0 and 31 (except, of course, tab, newline and carriage return); failing to do this will result in an XML-RPC request that isn’t well-formed XML. If you have to pass arbitrary bytes via XML-RPC, use **`bytes`** or **`bytearray`** classes or the **`Binary`** wrapper class described below.

`Server` is retained as an alias for **`ServerProxy`** for backwards compatibility. New code should use **`ServerProxy`**.

Changed in version 3.5: Added the *context* argument.

Changed in version 3.6: Added support of type tags with prefixes (e.g. `ex:nil`). Added support of unmarshalling additional types used by Apache XML-RPC implementation for numerics: `i1`, `i2`, `i8`, `biginteger`, `float` and `bigdecimal`. See https://ws.apache.org/xmlrpc/types.html for a description.

See also

XML-RPC HOWTO [https://tldp.org/HOWTO/XML-RPC-HOWTO/index.html]

A good description of XML-RPC operation and client software in several languages. Contains pretty much everything an XML-RPC client developer needs to know.

XML-RPC Introspection [https://xmlrpc-c.sourceforge.io/introspection.html]

Describes the XML-RPC protocol extension for introspection.

XML-RPC Specification [http://xmlrpc.scripting.com/spec.html]
The official specification.

ServerProxy Objects

A `ServerProxy` instance has a method corresponding to each remote procedure call accepted by the XML-RPC server. Calling the method performs an RPC, dispatched by both name and argument signature (e.g. the same method name can be overloaded with multiple argument signatures). The RPC finishes by returning a value, which may be either returned data in a conformant type or a `Fault` or `ProtocolError` object indicating an error.

Servers that support the XML introspection API support some common methods grouped under the reserved **`system`** attribute:

ServerProxy.system.listMethods()

This method returns a list of strings, one for each (non-system) method supported by the XML-RPC server.

ServerProxy.system.methodSignature(*name*)

This method takes one parameter, the name of a method implemented by the XML-RPC server. It returns an array of possible signatures for this method. A signature is an array of types. The first of these types is the return type of the method, the rest are parameters.

Because multiple signatures (ie. overloading) is permitted, this method returns a list of signatures rather than a singleton.

Signatures themselves are restricted to the top level parameters expected by a method. For instance if a method expects one array of structs as a parameter, and it returns a string, its signature is simply “string, array”. If it expects three integers and returns a string, its signature is “string, int, int, int”.

If no signature is defined for the method, a non-array value is returned. In Python this means that the type of the returned value will be something other than list.

ServerProxy.system.methodHelp(*name*)

This method takes one parameter, the name of a method implemented by the XML-RPC server. It returns a documentation string describing the use of that method. If no such string is available, an empty string is returned. The documentation string may contain HTML markup.

Changed in version 3.5: Instances of `ServerProxy` support the context manager protocol for closing the underlying transport.

A working example follows. The server code:

```
from xmlrpc.server import SimpleXMLRPCServer

def is_even(n):
    return n % 2 == 0

server = SimpleXMLRPCServer(("localhost", 8000))
print("Listening on port 8000...")
server.register_function(is_even, "is_even")
server.serve_forever()
```

The client code for the preceding server:

```
import xmlrpc.client
```

```
with xmlrpc.client.ServerProxy("http://localhost:8000/") as
proxy:
    print("3 is even: %s" % str(proxy.is_even(3)))
    print("100 is even: %s" % str(proxy.is_even(100)))
```

DateTime Objects

class xmlrpc.client.DateTime

This class may be initialized with seconds since the epoch, a time tuple, an ISO 8601 time/date string, or a `datetime.datetime` instance. It has the following methods, supported mainly for internal use by the marshalling/unmarshalling code:

decode(*string*)

Accept a string as the instance's new time value.

encode(*out*)

Write the XML-RPC encoding of this `DateTime` item to the *out* stream object.

It also supports certain of Python's built-in operators through `rich comparison` and `__repr__()` methods.

A working example follows. The server code:

```
import datetime
from xmlrpc.server import SimpleXMLRPCServer
import xmlrpc.client

def today():
    today = datetime.datetime.today()
    return xmlrpc.client.DateTime(today)

server = SimpleXMLRPCServer(("localhost", 8000))
print("Listening on port 8000...")
server.register_function(today, "today")
server.serve_forever()
```

The client code for the preceding server:

```
import xmlrpc.client
import datetime

proxy = xmlrpc.client.ServerProxy("http://localhost:8000/")

today = proxy.today()
# convert the ISO8601 string to a datetime object
converted = datetime.datetime.strptime(today.value,
"%Y%m%dT%H:%M:%S")
print("Today: %s" % converted.strftime("%d.%m.%Y, %H:%M"))
```

Binary Objects

class xmlrpc.client.Binary

This class may be initialized from bytes data (which may include NULs). The primary access to the content of a `Binary` object is provided by an attribute:

data

The binary data encapsulated by the `Binary` instance. The data is provided as a `bytes` object.

`Binary` objects have the following methods, supported mainly for internal use by the marshalling/unmarshalling code:

decode(*bytes*)

Accept a base64 `bytes` object and decode it as the instance’s new data.

encode(*out*)

Write the XML-RPC base 64 encoding of this binary item to the *out* stream object.

The encoded data will have newlines every 76 characters as per RFC 2045 section 6.8 [https://datatracker.ietf.org/doc/html/rfc2045.html#section-6.8], which was the de facto standard base64 specification when the XML-RPC spec was written.

It also supports certain of Python’s built-in operators through `__eq__()` and `__ne__()` methods.

Example usage of the binary objects. We’re going to transfer an image over XMLRPC:

```
from xmlrpc.server import SimpleXMLRPCServer
import xmlrpc.client

def python_logo():
    with open("python_logo.jpg", "rb") as handle:
        return xmlrpc.client.Binary(handle.read())

server = SimpleXMLRPCServer(("localhost", 8000))
print("Listening on port 8000...")
server.register_function(python_logo, 'python_logo')

server.serve_forever()
```

The client gets the image and saves it to a file:

```
import xmlrpc.client

proxy = xmlrpc.client.ServerProxy("http://localhost:8000/")
with open("fetched_python_logo.jpg", "wb") as handle:
    handle.write(proxy.python_logo().data)
```

Fault Objects

class xmlrpc.client.Fault

A `Fault` object encapsulates the content of an XML-RPC fault tag. Fault objects have the following attributes:

faultCode

An int indicating the fault type.

faultString

A string containing a diagnostic message associated with the fault.

In the following example we’re going to intentionally cause a `Fault` by returning a complex type object. The server code:

```
from xmlrpc.server import SimpleXMLRPCServer

# A marshalling error is going to occur because we're returning
a
# complex number
def add(x, y):
    return x+y+0j

server = SimpleXMLRPCServer(("localhost", 8000))
print("Listening on port 8000...")
server.register_function(add, 'add')

server.serve_forever()
```

The client code for the preceding server:

```
import xmlrpc.client

proxy = xmlrpc.client.ServerProxy("http://localhost:8000/")
try:
    proxy.add(2, 5)
except xmlrpc.client.Fault as err:
    print("A fault occurred")
    print("Fault code: %d" % err.faultCode)
    print("Fault string: %s" % err.faultString)
```

ProtocolError Objects

class xmlrpc.client.ProtocolError

A `ProtocolError` object describes a protocol error in the underlying transport layer (such as a 404 ‘not found’ error if the server named by the URI does not exist). It has the following attributes:

url
The URI or URL that triggered the error.

errcode
The error code.

errmsg
The error message or diagnostic string.

headers
A dict containing the headers of the HTTP/HTTPS request that triggered the error.

In the following example we’re going to intentionally cause a `ProtocolError` by providing an invalid URI:

```
import xmlrpc.client

# create a ServerProxy with a URI that doesn't respond to
XMLRPC requests
proxy = xmlrpc.client.ServerProxy("http://google.com/")

try:
    proxy.some_method()
except xmlrpc.client.ProtocolError as err:
    print("A protocol error occurred")
    print("URL: %s" % err.url)
    print("HTTP/HTTPS headers: %s" % err.headers)
    print("Error code: %d" % err.errcode)
    print("Error message: %s" % err.errmsg)
```

MultiCall Objects

The **`MultiCall`** object provides a way to encapsulate multiple calls to a remote server into a single request [[1]].

class xmlrpc.client.MultiCall(*server*)

> Create an object used to boxcar method calls. *server* is the eventual target of the call. Calls can be made to the result object, but they will immediately return `None`, and only store the call name and parameters in the **`MultiCall`** object. Calling the object itself causes all stored calls to be transmitted as a single `system.multicall` request. The result of this call is a generator; iterating over this generator yields the individual results.

A usage example of this class follows. The server code:

```
from xmlrpc.server import SimpleXMLRPCServer

def add(x, y):
    return x + y

def subtract(x, y):
    return x - y

def multiply(x, y):
    return x * y

def divide(x, y):
    return x // y

# A simple server with simple arithmetic functions
server = SimpleXMLRPCServer(("localhost", 8000))
print("Listening on port 8000...")
server.register_multicall_functions()
server.register_function(add, 'add')
server.register_function(subtract, 'subtract')
server.register_function(multiply, 'multiply')
server.register_function(divide, 'divide')
server.serve_forever()
```

The client code for the preceding server:

```
import xmlrpc.client

proxy = xmlrpc.client.ServerProxy("http://localhost:8000/")
multicall = xmlrpc.client.MultiCall(proxy)
multicall.add(7, 3)
multicall.subtract(7, 3)
multicall.multiply(7, 3)
multicall.divide(7, 3)
result = multicall()

print("7+3=%d, 7-3=%d, 7*3=%d, 7//3=%d" % tuple(result))
```

Convenience Functions

xmlrpc.client.dumps(*params*, *methodname=None*, *methodresponse=None*, *encoding=None*, *allow_none=False*)

> Convert *params* into an XML-RPC request, or into a response if *methodresponse* is true. *params* can be either a tuple of arguments or an instance of the `Fault` exception class. If *methodresponse* is true, only a single value can be returned, meaning that *params* must be of length 1. *encoding*, if supplied, is the encoding to use in the generated XML; the default is UTF-8. Python's `None` value cannot be used in standard XML-RPC; to allow using it via an extension, provide a true value for *allow_none*.

xmlrpc.client.loads(*data*, *use_datetime=False*, *use_builtin_types=False*)

> Convert an XML-RPC request or response into Python objects, a `(params, methodname)`. *params* is a tuple of argument; *methodname* is a string, or `None` if no method name is present in the packet. If the XML-RPC packet represents a fault condition, this function will raise a `Fault` exception. The *use_builtin_types* flag can be used to

cause date/time values to be presented as `datetime.datetime` objects and binary data to be presented as `bytes` objects; this flag is false by default.

The obsolete *use_datetime* flag is similar to *use_builtin_types* but it applies only to date/time values.

Changed in version 3.3: The *use_builtin_types* flag was added.

Example of Client Usage

```
# simple test program (from the XML-RPC specification)
from xmlrpc.client import ServerProxy, Error

# server = ServerProxy("http://localhost:8000") # local server
with ServerProxy("http://betty.userland.com") as proxy:

    print(proxy)

    try:
        print(proxy.examples.getStateName(41))
    except Error as v:
        print("ERROR", v)
```

To access an XML-RPC server through a HTTP proxy, you need to define a custom transport. The following example shows how:

```
import http.client
import xmlrpc.client

class ProxiedTransport(xmlrpc.client.Transport):

    def set_proxy(self, host, port=None, headers=None):
        self.proxy = host, port
        self.proxy_headers = headers

    def make_connection(self, host):
        connection = http.client.HTTPConnection(*self.proxy)
        connection.set_tunnel(host, headers=self.proxy_headers)
        self._connection = host, connection
        return connection

transport = ProxiedTransport()
transport.set_proxy('proxy-server', 8080)
server = xmlrpc.client.ServerProxy('http://betty.userland.com',
transport=transport)
print(server.examples.getStateName(41))
```

Example of Client and Server Usage

See SimpleXMLRPCServer Example.

Footnotes

[1]

This approach has been first presented in a discussion on xmlrpc.com [https://web.archive.org/web/20060624230303/http://www.xmlrpc.com/discuss/msgReader$1208?mode=topic].

`xmlrpc.server` — Basic XML-RPC servers

Source code: Lib/xmlrpc/server.py [https://github.com/python/cpython/tree/3.14/Lib/xmlrpc/server.py]

The **`xmlrpc.server`** module provides a basic server framework for XML-RPC servers written in Python. Servers can either be free standing, using **`SimpleXMLRPCServer`**, or embedded in a CGI environment, using **`CGIXMLRPCRequestHandler`**.

Warning

The **`xmlrpc.server`** module is not secure against maliciously constructed data. If you need to parse untrusted or unauthenticated data, see XML security.

Availability: not WASI.

This module does not work or is not available on WebAssembly. See WebAssembly platforms for more information.

class xmlrpc.server.SimpleXMLRPCServer(*addr*, *requestHandler=SimpleXMLRPCRequestHandler*, *logRequests=True*, *allow_none=False*, *encoding=None*, *bind_and_activate=True*, *use_builtin_types=False*)

Create a new server instance. This class provides methods for registration of functions that can be called by the XML-RPC protocol. The *requestHandler* parameter should be a factory for request handler instances; it defaults to `SimpleXMLRPCRequestHandler`. The *addr* and *requestHandler* parameters are passed to the `socketserver.TCPServer` constructor. If *logRequests* is true (the default), requests will be logged; setting this parameter to false will turn off logging. The *allow_none* and *encoding* parameters are passed on to `xmlrpc.client` and control the XML-RPC responses that will be returned from the server. The *bind_and_activate* parameter controls whether **`server_bind()`** and **`server_activate()`** are called immediately by the constructor; it defaults to true. Setting it to false allows code to manipulate the *allow_reuse_address* class variable before the address is bound. The *use_builtin_types* parameter is passed to the `loads()` function and controls which types are processed when date/times values or binary data are received; it defaults to false.

Changed in version 3.3: The *use_builtin_types* flag was added.

class xmlrpc.server.CGIXMLRPCRequestHandler(*allow_none=False*, *encoding=None*, *use_builtin_types=False*)

Create a new instance to handle XML-RPC requests in a CGI environment. The *allow_none* and *encoding* parameters are passed on to `xmlrpc.client` and control the XML-RPC responses that will be returned from the server. The *use_builtin_types* parameter is passed to the `loads()` function and controls which types are processed when date/times values or binary data are received; it defaults to false.

Changed in version 3.3: The *use_builtin_types* flag was added.

class xmlrpc.server.SimpleXMLRPCRequestHandler

Create a new request handler instance. This request handler supports `POST` requests and modifies logging so that the *logRequests* parameter to the `SimpleXMLRPCServer` constructor parameter is honored.

SimpleXMLRPCServer Objects

The `SimpleXMLRPCServer` class is based on `socketserver.TCPServer` and provides a means of creating simple, stand alone XML-RPC servers.

SimpleXMLRPCServer.register_function(*function=None*, *name=None*)

Register a function that can respond to XML-RPC requests. If *name* is given, it will be the method name associated with *function*, otherwise `function.__name__` will be used. *name* is a string, and may contain characters not legal in Python identifiers, including the period character.

This method can also be used as a decorator. When used as a decorator, *name* can only be given as a keyword argument to register *function* under *name*. If no *name* is given, `function.__name__` will be used.

Changed in version 3.7: `register_function()` can be used as a decorator.

SimpleXMLRPCServer.register_instance(*instance*, *allow_dotted_names=False*)

Register an object which is used to expose method names which have not been registered using `register_function()`. If *instance* contains a **`_dispatch()`** method, it is called with the requested method name and the parameters from the request. Its API is `def _dispatch(self, method, params)` (note that *params* does not represent a variable argument list). If it calls an underlying function to perform its task, that function is called as `func(*params)`, expanding the parameter list. The return value from **`_dispatch()`** is returned to the client as the result. If *instance* does not have a **`_dispatch()`** method, it is searched for an attribute matching the name of the requested method.

If the optional *allow_dotted_names* argument is true and the instance does not have a **`_dispatch()`** method, then if the requested method name contains periods, each component of the method name is searched for individually, with the effect that a simple hierarchical search is performed. The value found from this search is then called with the parameters from the request, and the return value is passed back to the client.

Warning

Enabling the *allow_dotted_names* option allows intruders to access your module’s global variables and may allow intruders to execute arbitrary code on your machine. Only use this option on a secure, closed network.

SimpleXMLRPCServer.register_introspection_functions()

Registers the XML-RPC introspection functions `system.listMethods`, `system.methodHelp` and `system.methodSignature`.

SimpleXMLRPCServer.register_multicall_functions()

Registers the XML-RPC multicall function system.multicall.

SimpleXMLRPCRequestHandler.rpc_paths

An attribute value that must be a tuple listing valid path portions of the URL for receiving XML-RPC requests. Requests posted to other paths will result in a 404 “no such page” HTTP error. If this tuple is empty, all paths will be considered valid. The default value is `('/', '/RPC2')`.

SimpleXMLRPCServer Example

Server code:

```
from xmlrpc.server import SimpleXMLRPCServer
from xmlrpc.server import SimpleXMLRPCRequestHandler

# Restrict to a particular path.
class RequestHandler(SimpleXMLRPCRequestHandler):
    rpc_paths = ('/RPC2',)

# Create server
with SimpleXMLRPCServer(('localhost', 8000),
                        requestHandler=RequestHandler) as
server:
    server.register_introspection_functions()

    # Register pow() function; this will use the value of
    # pow.__name__ as the name, which is just 'pow'.
    server.register_function(pow)

    # Register a function under a different name
    def adder_function(x, y):
        return x + y
    server.register_function(adder_function, 'add')

    # Register an instance; all the methods of the instance are
    # published as XML-RPC methods (in this case, just 'mul').
    class MyFuncs:
        def mul(self, x, y):
            return x * y

    server.register_instance(MyFuncs())
    # Run the server's main loop
    server.serve_forever()
```

The following client code will call the methods made available by the preceding server:

```
import xmlrpc.client

s = xmlrpc.client.ServerProxy('http://localhost:8000')
print(s.pow(2,3))  # Returns 2**3 = 8
print(s.add(2,3))  # Returns 5
print(s.mul(5,2))  # Returns 5*2 = 10

# Print list of available methods
print(s.system.listMethods())
```

`register_function()` can also be used as a decorator. The previous server example can register functions in a decorator way:

```
from xmlrpc.server import SimpleXMLRPCServer
from xmlrpc.server import SimpleXMLRPCRequestHandler

class RequestHandler(SimpleXMLRPCRequestHandler):
    rpc_paths = ('/RPC2',)

with SimpleXMLRPCServer(('localhost', 8000),
                        requestHandler=RequestHandler) as
server:
    server.register_introspection_functions()

    # Register pow() function; this will use the value of
    # pow.__name__ as the name, which is just 'pow'.
    server.register_function(pow)

    # Register a function under a different name, using
    # register_function as a decorator. *name* can only be
given
    # as a keyword argument.
    @server.register_function(name='add')
    def adder_function(x, y):
        return x + y
    # Register a function under function.__name__.
    @server.register_function
    def mul(x, y):
        return x * y

    server.serve_forever()
```

The following example included in the `Lib/xmlrpc/server.py` module shows a server allowing dotted names and registering a multicall function.

Warning

Enabling the *allow_dotted_names* option allows intruders to access your module’s global variables and may allow intruders to execute arbitrary code on your machine. Only use this example only within a secure, closed network.

```
import datetime

class ExampleService:
    def getData(self):
        return '42'

    class currentTime:
        @staticmethod
        def getCurrentTime():
            return datetime.datetime.now()

with SimpleXMLRPCServer(("localhost", 8000)) as server:
    server.register_function(pow)
    server.register_function(lambda x,y: x+y, 'add')
    server.register_instance(ExampleService(),
allow_dotted_names=True)
    server.register_multicall_functions()
    print('Serving XML-RPC on localhost port 8000')
    try:
        server.serve_forever()
    except KeyboardInterrupt:
        print("\nKeyboard interrupt received, exiting.")
        sys.exit(0)
```

This ExampleService demo can be invoked from the command line:

```
python -m xmlrpc.server
```

The client that interacts with the above server is included in `Lib/xmlrpc/client.py`:

```
server = ServerProxy("http://localhost:8000")

try:
    print(server.currentTime.getCurrentTime())
except Error as v:
    print("ERROR", v)

multi = MultiCall(server)
multi.getData()
multi.pow(2,9)
multi.add(1,2)
try:
    for response in multi():
        print(response)
except Error as v:
    print("ERROR", v)
```

This client which interacts with the demo XMLRPC server can be invoked as:

```
python -m xmlrpc.client
```

CGIXMLRPCRequestHandler

The **`CGIXMLRPCRequestHandler`** class can be used to handle XML-RPC requests sent to Python CGI scripts.

CGIXMLRPCRequestHandler.register_function(*function=None*, *name=None*)

Register a function that can respond to XML-RPC requests. If *name* is given, it will be the method name associated with *function*, otherwise `function.__name__` will be used. *name* is a string, and may contain characters not legal in Python identifiers, including the period character.

This method can also be used as a decorator. When used as a decorator, *name* can only be given as a keyword argument to register *function* under *name*. If no *name* is given, `function.__name__` will be used.

Changed in version 3.7: `register_function()` can be used as a decorator.

CGIXMLRPCRequestHandler.register_instance(*instance*)

Register an object which is used to expose method names which have not been registered using `register_function()`. If instance contains a `_dispatch()` method, it is called with the requested method name and the parameters from the request; the return value is returned to the client as the result. If instance does not have a `_dispatch()` method, it is searched for an attribute matching the name of the requested method; if the requested method name contains periods, each component of the method name is searched for individually, with the effect that a simple hierarchical search is performed. The value found from this search is then called with the parameters from the request, and the return value is passed back to the client.

CGIXMLRPCRequestHandler.register_introspection_functions()

Register the XML-RPC introspection functions `system.listMethods`, `system.methodHelp` and `system.methodSignature`.

CGIXMLRPCRequestHandler.register_multicall_functions()

Register the XML-RPC multicall function `system.multicall`.

CGIXMLRPCRequestHandler.handle_request(*request_text=None*)

Handle an XML-RPC request. If *request_text* is given, it should be the POST data provided by the HTTP server, otherwise the contents of stdin will be used.

Example:

```
class MyFuncs:
    def mul(self, x, y):
        return x * y


handler = CGIXMLRPCRequestHandler()
handler.register_function(pow)
handler.register_function(lambda x,y: x+y, 'add')
handler.register_introspection_functions()
handler.register_instance(MyFuncs())
handler.handle_request()
```

Documenting XMLRPC server

These classes extend the above classes to serve HTML documentation in response to HTTP GET requests. Servers can either be free standing, using **`DocXMLRPCServer`**, or embedded in a CGI environment, using **`DocCGIXMLRPCRequestHandler`**.

class xmlrpc.server.DocXMLRPCServer(*addr*, *requestHandler=DocXMLRPCRequestHandler*, *logRequests=True*, *allow_none=False*, *encoding=None*, *bind_and_activate=True*, *use_builtin_types=True*)

Create a new server instance. All parameters have the same meaning as for `SimpleXMLRPCServer`; *requestHandler* defaults to `DocXMLRPCRequestHandler`.

Changed in version 3.3: The *use_builtin_types* flag was added.

class xmlrpc.server.DocCGIXMLRPCRequestHandler

Create a new instance to handle XML-RPC requests in a CGI environment.

class xmlrpc.server.DocXMLRPCRequestHandler

Create a new request handler instance. This request handler supports XML-RPC POST requests, documentation GET requests, and modifies logging so that the *logRequests* parameter to the `DocXMLRPCServer` constructor parameter is honored.

DocXMLRPCServer Objects

The `DocXMLRPCServer` class is derived from `SimpleXMLRPCServer` and provides a means of creating self-documenting, stand alone XML-RPC servers. HTTP POST requests are handled as XML-RPC method calls. HTTP GET requests are handled by generating pydoc-style HTML documentation. This allows a server to provide its own web-based documentation.

DocXMLRPCServer.set_server_title(*server_title*)

Set the title used in the generated HTML documentation. This title will be used inside the HTML “title” element.

DocXMLRPCServer.set_server_name(*server_name*)

Set the name used in the generated HTML documentation. This name will appear at the top of the generated documentation inside a “h1” element.

DocXMLRPCServer.set_server_documentation(*server_documentation*)

Set the description used in the generated HTML documentation. This description will appear as a paragraph, below the server name, in the documentation.

DocCGIXMLRPCRequestHandler

The `DocCGIXMLRPCRequestHandler` class is derived from `CGIXMLRPCRequestHandler` and provides a means of creating self-documenting, XML-RPC CGI scripts. HTTP POST requests are handled as XML-RPC method calls. HTTP GET requests are handled by generating pydoc-style HTML documentation. This allows a server to provide its own web-based documentation.

DocCGIXMLRPCRequestHandler.set_server_title(*server_title*)

Set the title used in the generated HTML documentation. This title will be used inside the HTML “title” element.

DocCGIXMLRPCRequestHandler.set_server_name(*server_name*)

Set the name used in the generated HTML documentation. This name will appear at the top of the generated documentation inside a “h1” element.

DocCGIXMLRPCRequestHandler.set_server_documentation(*server_documentation*)

Set the description used in the generated HTML documentation. This description will appear as a paragraph, below the server name, in the documentation.

`ipaddress` — IPv4/IPv6 manipulation library

Source code: Lib/ipaddress.py [https://github.com/python/cpython/tree/3.14/Lib/ipaddress.py]

`ipaddress` provides the capabilities to create, manipulate and operate on IPv4 and IPv6 addresses and networks.

The functions and classes in this module make it straightforward to handle various tasks related to IP addresses, including checking whether or not two hosts are on the same subnet, iterating over all hosts in a particular subnet, checking whether or not a string represents a valid IP address or network definition, and so on.

This is the full module API reference—for an overview and introduction, see An introduction to the ipaddress module.

Added in version 3.3.

Convenience factory functions

The `ipaddress` module provides factory functions to conveniently create IP addresses, networks and interfaces:

ipaddress.ip_address(*address*)

Return an `IPv4Address` or `IPv6Address` object depending on the IP address passed as argument. Either IPv4 or IPv6 addresses may be supplied; integers less than `2**32` will be considered to be IPv4 by default. A `ValueError` is raised if *address* does not represent a valid IPv4 or IPv6 address.

```
>>> ipaddress.ip_address('192.168.0.1')
IPv4Address('192.168.0.1')
>>> ipaddress.ip_address('2001:db8::')
IPv6Address('2001:db8::')
```

ipaddress.ip_network(*address*, *strict=True*)

Return an **`IPv4Network`** or **`IPv6Network`** object depending on the IP address passed as argument. *address* is a string or integer representing the IP network. Either IPv4 or IPv6 networks may be supplied; integers less than `2**32` will be considered to be IPv4 by default. *strict* is passed to **`IPv4Network`** or **`IPv6Network`** constructor. A **`ValueError`** is raised if *address* does not represent a valid IPv4 or IPv6 address, or if the network has host bits set.

```
>>> ipaddress.ip_network('192.168.0.0/28')
IPv4Network('192.168.0.0/28')
```

ipaddress.ip_interface(*address*)

Return an **`IPv4Interface`** or **`IPv6Interface`** object depending on the IP address passed as argument. *address* is a string or integer representing the IP address. Either IPv4 or IPv6 addresses may be supplied; integers less than `2**32` will be considered to be IPv4 by default. A **`ValueError`** is raised if *address* does not represent a valid IPv4 or IPv6 address.

One downside of these convenience functions is that the need to handle both IPv4 and IPv6 formats means that error messages provide minimal information on the precise error, as the functions don’t know whether the IPv4 or IPv6 format was intended. More detailed error reporting can be obtained by calling the appropriate version specific class constructors directly.

IP Addresses

Address objects

The **`IPv4Address`** and **`IPv6Address`** objects share a lot of common attributes. Some attributes that are only meaningful for IPv6 addresses are also implemented by **`IPv4Address`** objects, in order to make it easier to write code that handles both IP versions correctly. Address objects are hashable, so they can be used as keys in dictionaries.

class ipaddress.IPv4Address(*address*)

Construct an IPv4 address. An **`AddressValueError`** is raised if *address* is not a valid IPv4 address.

The following constitutes a valid IPv4 address:

1. A string in decimal-dot notation, consisting of four decimal integers in the inclusive range 0–255, separated by dots (e.g. `192.168.0.1`). Each integer represents an octet (byte) in the address. Leading zeroes are not tolerated to prevent confusion with octal notation.
2. An integer that fits into 32 bits.
3. An integer packed into a **`bytes`** object of length 4 (most significant octet first).

```
>>> ipaddress.IPv4Address('192.168.0.1')
IPv4Address('192.168.0.1')
>>> ipaddress.IPv4Address(3232235521)
IPv4Address('192.168.0.1')
>>> ipaddress.IPv4Address(b'\xC0\xA8\x00\x01')
IPv4Address('192.168.0.1')
```

Changed in version 3.8: Leading zeros are tolerated, even in ambiguous cases that look like octal notation.

Changed in version 3.9.5: Leading zeros are no longer tolerated and are treated as an error. IPv4 address strings are now parsed as strict as glibc `inet_pton()`.

version

The appropriate version number: `4` for IPv4, `6` for IPv6.

Changed in version 3.14: Made available on the class.

max_prefixlen

The total number of bits in the address representation for this version: `32` for IPv4, `128` for IPv6.

The prefix defines the number of leading bits in an address that are compared to determine whether or not an address is part of a network.

Changed in version 3.14: Made available on the class.

compressed

exploded

The string representation in dotted decimal notation. Leading zeroes are never included in the representation.

As IPv4 does not define a shorthand notation for addresses with octets set to zero, these two attributes are always the same as `str(addr)` for IPv4 addresses. Exposing these attributes makes it easier to write display code that can handle both IPv4 and IPv6 addresses.

packed

The binary representation of this address - a `bytes` object of the appropriate length (most significant octet first). This is 4 bytes for IPv4 and 16 bytes for IPv6.

reverse_pointer

The name of the reverse DNS PTR record for the IP address, e.g.:

```
>>> ipaddress.ip_address("127.0.0.1").reverse_pointer
'1.0.0.127.in-addr.arpa'
>>> ipaddress.ip_address("2001:db8::1").reverse_pointer
'1.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.8.b.d.0.
1.0.0.2.ip6.arpa'
```

This is the name that could be used for performing a PTR lookup, not the resolved hostname itself.

Added in version 3.5.

is_multicast

`True` if the address is reserved for multicast use. See **RFC 3171** [https://datatracker.ietf.org/doc/html/rfc3171.html] (for IPv4) or **RFC 2373** [https://datatracker.ietf.org/doc/html/rfc2373.html] (for IPv6).

is_private

`True` if the address is defined as not globally reachable by iana-ipv4-special-registry [https://www.iana.org/assignments/iana-ipv4-special-registry/iana-ipv4-special-registry.xhtml] (for IPv4) or iana-ipv6-special-registry [https://www.iana.org/assignments/iana-ipv6-special-registry/iana-ipv6-special-registry.xhtml] (for IPv6) with the following exceptions:

- `is_private` is `False` for the shared address space (`100.64.0.0/10`)
- For IPv4-mapped IPv6-addresses the `is_private` value is determined by the semantics of the

underlying IPv4 addresses and the following condition holds (see **`IPv6Address.ipv4_mapped`**):

```
address.is_private == address.ipv4_mapped.is_private
```

`is_private` has value opposite to **`is_global`**, except for the shared address space (`100.64.0.0/10` range) where they are both `False`.

Changed in version 3.13: Fixed some false positives and false negatives.

- `192.0.0.0/24` is considered private with the exception of `192.0.0.9/32` and `192.0.0.10/32` (previously: only the `192.0.0.0/29` sub-range was considered private).
- `64:ff9b:1::/48` is considered private.
- `2002::/16` is considered private.
- There are exceptions within `2001::/23` (otherwise considered private): `2001:1::1/128`, `2001:1::2/128`, `2001:3::/32`, `2001:4:112::/48`, `2001:20::/28`, `2001:30::/28`. The exceptions are not considered private.

is_global

`True` if the address is defined as globally reachable by iana-ipv4-special-registry [https://www.iana.org/assignments/iana-ipv4-special-registry/iana-ipv4-special-registry.xhtml] (for IPv4) or iana-ipv6-special-registry [https://www.iana.org/assignments/iana-ipv6-special-registry/iana-ipv6-special-registry.xhtml] (for IPv6) with the following exception:

For IPv4-mapped IPv6-addresses the `is_private` value is determined by the semantics of the underlying IPv4 addresses and the following condition holds (see **`IPv6Address.ipv4_mapped`**):

```
address.is_global == address.ipv4_mapped.is_global
```

`is_global` has value opposite to **`is_private`**, except for the shared address space (`100.64.0.0/10` range) where they are both `False`.

Added in version 3.4.

Changed in version 3.13: Fixed some false positives and false negatives, see **`is_private`** for details.

is_unspecified

`True` if the address is unspecified. See **RFC 5735** [https://datatracker.ietf.org/doc/html/rfc5735.html] (for IPv4) or **RFC 2373** [https://datatracker.ietf.org/doc/html/rfc2373.html] (for IPv6).

is_reserved

`True` if the address is noted as reserved by the IETF. For IPv4, this is only `240.0.0.0/4`, the `Reserved` address block. For IPv6, this is all addresses allocated [https://www.iana.org/assignments/ipv6-address-space/ipv6-address-space.xhtml] as `Reserved by IETF` for future use.

Note

For IPv4, `is_reserved` is not related to the address block value of the `Reserved-by-Protocol` column in iana-ipv4-special-registry [https://www.iana.org/assignments/iana-ipv4-special-registry/iana-ipv4-special-registry.xhtml].

Caution

For IPv6, `fec0::/10` a former Site-Local scoped address prefix is currently excluded from that list (see **`is_site_local`** & **RFC 3879** [https://datatracker.ietf.org/doc/html/rfc3879.html]).

is_loopback

`True` if this is a loopback address. See **RFC 3330** [https://datatracker.ietf.org/doc/html/rfc3330.html] (for IPv4) or **RFC 2373** [https://datatracker.ietf.org/doc/html/rfc2373.html] (for IPv6).

is_link_local

`True` if the address is reserved for link-local usage. See **RFC 3927** [https://datatracker.ietf.org/doc/html/rfc3927.html].

ipv6_mapped

`IPv4Address` object representing the IPv4-mapped IPv6 address. See **RFC 4291** [https://datatracker.ietf.org/doc/html/rfc4291.html].

Added in version 3.13.

IPv4Address.__format__(*fmt*)

Returns a string representation of the IP address, controlled by an explicit format string. *fmt* can be one of the following: `'s'`, the default option, equivalent to **`str()`**, `'b'` for a zero-padded binary string, `'X'` or `'x'` for an uppercase or lowercase hexadecimal representation, or `'n'`, which is equivalent to `'b'` for IPv4 addresses and `'x'` for IPv6. For binary and hexadecimal representations, the form specifier `'#'` and the grouping option `'_'` are available. `__format__` is used by `format`, `str.format` and f-strings.

```
>>> format(ipaddress.IPv4Address('192.168.0.1'))
'192.168.0.1'
>>> '{:#b}'.format(ipaddress.IPv4Address('192.168.0.1'))
'0b11000000101010000000000000000001'
```

```
>>> f'{ipaddress.IPv6Address("2001:db8::1000"):s}'
'2001:db8::1000'
>>> format(ipaddress.IPv6Address('2001:db8::1000'), '_X')
'2001_0DB8_0000_0000_0000_0000_0000_1000'
>>> '{:#_n}'.format(ipaddress.IPv6Address('2001:db8::1000'))
'0x2001_0db8_0000_0000_0000_0000_0000_1000'
```

Added in version 3.9.

class ipaddress.IPv6Address(*address*)

Construct an IPv6 address. An `AddressValueError` is raised if *address* is not a valid IPv6 address.

The following constitutes a valid IPv6 address:

1. A string consisting of eight groups of four hexadecimal digits, each group representing 16 bits. The groups are separated by colons. This describes an *exploded* (longhand) notation. The string can also be *compressed* (shorthand notation) by various means. See **RFC 4291** [https://datatracker.ietf.org/doc/html/rfc4291.html] for details. For example, `"0000:0000:0000:0000:0000:0abc:0007:0def"` can be compressed to `"::abc:7:def"`.

   Optionally, the string may also have a scope zone ID, expressed with a suffix `%scope_id`. If present, the scope ID must be non-empty, and may not contain `%`. See **RFC 4007** [https://datatracker.ietf.org/doc/html/rfc4007.html] for details. For example, `fe80::1234%1` might identify address `fe80::1234` on the first link of the node.

2. An integer that fits into 128 bits.

3. An integer packed into a `bytes` object of length 16, big-endian.

```
>>> ipaddress.IPv6Address('2001:db8::1000')
IPv6Address('2001:db8::1000')
```

```
>>> ipaddress.IPv6Address('ff02::5678%1')
IPv6Address('ff02::5678%1')
```

compressed

The short form of the address representation, with leading zeroes in groups omitted and the longest sequence of groups consisting entirely of zeroes collapsed to a single empty group.

This is also the value returned by `str(addr)` for IPv6 addresses.

exploded

The long form of the address representation, with all leading zeroes and groups consisting entirely of zeroes included.

For the following attributes and methods, see the corresponding documentation of the **`IPv4Address`** class:

packed

reverse_pointer

version

max_prefixlen

is_multicast

is_private

is_global

Added in version 3.4.

is_unspecified

is_reserved

is_loopback

is_link_local

is_site_local

`True` if the address is reserved for site-local usage. Note that the site-local address space has been deprecated by **RFC 3879** [https://datatracker.ietf.org/doc/html/rfc3879.html]. Use `is_private` to test if this address is in the space of unique local addresses as defined by **RFC 4193** [https://datatracker.ietf.org/doc/html/rfc4193.html].

ipv4_mapped

For addresses that appear to be IPv4 mapped addresses (starting with `::FFFF/96`), this property will report the embedded IPv4 address. For any other address, this property will be `None`.

scope_id

For scoped addresses as defined by **RFC 4007** [https://datatracker.ietf.org/doc/html/rfc4007.html], this property identifies the particular zone of the address's scope that the address belongs to, as a string. When no scope zone is specified, this property will be `None`.

sixtofour

For addresses that appear to be 6to4 addresses (starting with `2002::/16`) as defined by **RFC 3056** [https://datatracker.ietf.org/doc/html/rfc3056.html], this property will report the embedded IPv4 address. For any other address, this property will be `None`.

teredo

For addresses that appear to be Teredo addresses (starting with `2001::/32`) as defined by **RFC 4380** [https://datatracker.ietf.org/doc/html/rfc4380.html], this property will report the embedded `(server, client)` IP address pair. For any other address, this property will be `None`.

IPv6Address.__format__(*fmt*)

Refer to the corresponding method documentation in `IPv4Address`.

Added in version 3.9.

Conversion to Strings and Integers

To interoperate with networking interfaces such as the socket module, addresses must be converted to strings or integers. This is handled using the `str()` and `int()` builtin functions:

```
>>> str(ipaddress.IPv4Address('192.168.0.1'))
'192.168.0.1'
>>> int(ipaddress.IPv4Address('192.168.0.1'))
3232235521
>>> str(ipaddress.IPv6Address('::1'))
'::1'
>>> int(ipaddress.IPv6Address('::1'))
1
```

Note that IPv6 scoped addresses are converted to integers without scope zone ID.

Operators

Address objects support some operators. Unless stated otherwise, operators can only be applied between compatible objects (i.e. IPv4 with IPv4, IPv6 with IPv6).

Comparison operators

Address objects can be compared with the usual set of comparison operators. Same IPv6 addresses with different scope zone IDs are not equal. Some examples:

```
>>> IPv4Address('127.0.0.2') > IPv4Address('127.0.0.1')
True
>>> IPv4Address('127.0.0.2') == IPv4Address('127.0.0.1')
False
>>> IPv4Address('127.0.0.2') != IPv4Address('127.0.0.1')
True
>>> IPv6Address('fe80::1234') == IPv6Address('fe80::1234%1')
False
>>> IPv6Address('fe80::1234%1') != IPv6Address('fe80::1234%2')
True
```

Arithmetic operators

Integers can be added to or subtracted from address objects. Some examples:

```
>>> IPv4Address('127.0.0.2') + 3
IPv4Address('127.0.0.5')
>>> IPv4Address('127.0.0.2') - 3
IPv4Address('126.255.255.255')
>>> IPv4Address('255.255.255.255') + 1
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ipaddress.AddressValueError: 4294967296 (>= 2**32) is not
permitted as an IPv4 address
```

IP Network definitions

The **`IPv4Network`** and **`IPv6Network`** objects provide a mechanism for defining and inspecting IP network definitions. A network definition consists of a *mask* and a *network address*, and as such defines a range of IP addresses that equal the network address when masked (binary AND) with the mask. For example, a network definition with the mask `255.255.255.0` and the network address `192.168.1.0` consists of IP addresses in the inclusive range `192.168.1.0` to `192.168.1.255`.

Prefix, net mask and host mask

There are several equivalent ways to specify IP network masks. A *prefix* `/<nbits>` is a notation that denotes how many high-order bits are set in the network mask. A *net mask* is an IP address with some number of high-order bits set. Thus the prefix `/24` is equivalent to the net mask `255.255.255.0` in IPv4, or `ffff:ff00::` in IPv6. In addition, a *host mask* is the logical inverse of a *net mask*, and is sometimes used (for example in Cisco access control lists) to denote a network mask. The host mask equivalent to `/24` in IPv4 is `0.0.0.255`.

Network objects

All attributes implemented by address objects are implemented by network objects as well. In addition, network objects implement additional attributes. All of these are common between **`IPv4Network`** and **`IPv6Network`**, so to avoid duplication they are only documented for **`IPv4Network`**. Network objects are hashable, so they can be used as keys in dictionaries.

class ipaddress.IPv4Network(*address*, *strict=True*)

Construct an IPv4 network definition. *address* can be one of the following:

1. A string consisting of an IP address and an optional mask, separated by a slash (`/`). The IP address is the network address, and the mask can be either a single number, which means it’s a *prefix*, or a string representation of an IPv4 address. If it’s the latter, the mask is interpreted as a *net mask* if it starts with

a non-zero field, or as a *host mask* if it starts with a zero field, with the single exception of an all-zero mask which is treated as a *net mask*. If no mask is provided, it's considered to be `/32`.

   For example, the following *address* specifications are equivalent: `192.168.1.0/24`, `192.168.1.0/255.255.255.0` and `192.168.1.0/0.0.0.255`.

2. An integer that fits into 32 bits. This is equivalent to a single-address network, with the network address being *address* and the mask being `/32`.

3. An integer packed into a `bytes` object of length 4, big-endian. The interpretation is similar to an integer *address*.

4. A two-tuple of an address description and a netmask, where the address description is either a string, a 32-bits integer, a 4-bytes packed integer, or an existing `IPv4Address` object; and the netmask is either an integer representing the prefix length (e.g. `24`) or a string representing the prefix mask (e.g. `255.255.255.0`).

An `AddressValueError` is raised if *address* is not a valid IPv4 address. A `NetmaskValueError` is raised if the mask is not valid for an IPv4 address.

If *strict* is `True` and host bits are set in the supplied address, then `ValueError` is raised. Otherwise, the host bits are masked out to determine the appropriate network address.

Unless stated otherwise, all network methods accepting other network/address objects will raise `TypeError` if the argument's IP version is incompatible to `self`.

Changed in version 3.5: Added the two-tuple form for the *address* constructor parameter.

version

max_prefixlen

Refer to the corresponding attribute documentation in `IPv4Address`.

is_multicast

is_private

is_unspecified

is_reserved

is_loopback

is_link_local

These attributes are true for the network as a whole if they are true for both the network address and the broadcast address.

network_address

The network address for the network. The network address and the prefix length together uniquely define a network.

broadcast_address

The broadcast address for the network. Packets sent to the broadcast address should be received by every host on the network.

hostmask

The host mask, as an `IPv4Address` object.

netmask

The net mask, as an `IPv4Address` object.

with_prefixlen

compressed

exploded

A string representation of the network, with the mask in prefix notation.

`with_prefixlen` and `compressed` are always the same as `str(network)`. `exploded` uses the exploded form the network address.

with_netmask

A string representation of the network, with the mask in net mask notation.

with_hostmask

A string representation of the network, with the mask in host mask notation.

num_addresses

The total number of addresses in the network.

prefixlen

Length of the network prefix, in bits.

hosts()

Returns an iterator over the usable hosts in the network. The usable hosts are all the IP addresses that belong to the network, except the network address itself and the network broadcast address. For networks with a mask length of 31, the network address and network broadcast address are also included in the result. Networks with a mask of 32 will return a list containing the single host address.

```
>>> list(ip_network('192.0.2.0/29').hosts())
[IPv4Address('192.0.2.1'), IPv4Address('192.0.2.2'),
 IPv4Address('192.0.2.3'), IPv4Address('192.0.2.4'),
 IPv4Address('192.0.2.5'), IPv4Address('192.0.2.6')]
>>> list(ip_network('192.0.2.0/31').hosts())
[IPv4Address('192.0.2.0'), IPv4Address('192.0.2.1')]
>>> list(ip_network('192.0.2.1/32').hosts())
[IPv4Address('192.0.2.1')]
```

overlaps(*other*)

`True` if this network is partly or wholly contained in *other* or *other* is wholly contained in this network.

address_exclude(*network*)

Computes the network definitions resulting from removing the given *network* from this one. Returns an iterator of network objects. Raises `ValueError` if *network* is not completely contained in this network.

```
>>> n1 = ip_network('192.0.2.0/28')
>>> n2 = ip_network('192.0.2.1/32')
>>> list(n1.address_exclude(n2))
[IPv4Network('192.0.2.8/29'),
IPv4Network('192.0.2.4/30'),
 IPv4Network('192.0.2.2/31'),
IPv4Network('192.0.2.0/32')]
```

subnets(*prefixlen_diff=1*, *new_prefix=None*)

The subnets that join to make the current network definition, depending on the argument values. *prefixlen_diff* is the amount our prefix length should be increased by. *new_prefix* is the desired new prefix of the subnets; it must be larger than our prefix. One and only one of *prefixlen_diff* and *new_prefix* must be set. Returns an iterator of network objects.

```
>>> list(ip_network('192.0.2.0/24').subnets())
[IPv4Network('192.0.2.0/25'),
IPv4Network('192.0.2.128/25')]
```

```
>>>
list(ip_network('192.0.2.0/24').subnets(prefixlen_diff=2)
)
[IPv4Network('192.0.2.0/26'),
IPv4Network('192.0.2.64/26'),
 IPv4Network('192.0.2.128/26'),
IPv4Network('192.0.2.192/26')]
>>>
list(ip_network('192.0.2.0/24').subnets(new_prefix=26))
[IPv4Network('192.0.2.0/26'),
IPv4Network('192.0.2.64/26'),
 IPv4Network('192.0.2.128/26'),
IPv4Network('192.0.2.192/26')]
>>>
list(ip_network('192.0.2.0/24').subnets(new_prefix=23))
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
    raise ValueError('new prefix must be longer')
ValueError: new prefix must be longer
>>>
list(ip_network('192.0.2.0/24').subnets(new_prefix=25))
[IPv4Network('192.0.2.0/25'),
IPv4Network('192.0.2.128/25')]
```

supernet(*prefixlen_diff=1*, *new_prefix=None*)

The supernet containing this network definition, depending on the argument values. *prefixlen_diff* is the amount our prefix length should be decreased by. *new_prefix* is the desired new prefix of the supernet; it must be smaller than our prefix. One and only one of *prefixlen_diff* and *new_prefix* must be set. Returns a single network object.

```
>>> ip_network('192.0.2.0/24').supernet()
IPv4Network('192.0.2.0/23')
>>> ip_network('192.0.2.0/24').supernet(prefixlen_diff=2)
IPv4Network('192.0.0.0/22')
>>> ip_network('192.0.2.0/24').supernet(new_prefix=20)
IPv4Network('192.0.0.0/20')
```

subnet_of(*other*)

Return `True` if this network is a subnet of *other*.

```
>>> a = ip_network('192.168.1.0/24')
>>> b = ip_network('192.168.1.128/30')
>>> b.subnet_of(a)
True
```

Added in version 3.7.

supernet_of(*other*)

Return `True` if this network is a supernet of *other*.

```
>>> a = ip_network('192.168.1.0/24')
>>> b = ip_network('192.168.1.128/30')
>>> a.supernet_of(b)
True
```

Added in version 3.7.

compare_networks(*other*)

Compare this network to *other*. In this comparison only the network addresses are considered; host bits aren’t. Returns either `-1`, `0` or `1`.

```
>>>
ip_network('192.0.2.1/32').compare_networks(ip_network('1
92.0.2.2/32'))
-1
>>>
ip_network('192.0.2.1/32').compare_networks(ip_network('1
92.0.2.0/32'))
1
>>>
ip_network('192.0.2.1/32').compare_networks(ip_network('1
92.0.2.1/32'))
0
```

Deprecated since version 3.7: It uses the same ordering and comparison algorithm as “<”, “==”, and “>”

class ipaddress.IPv6Network(*address*, *strict=True*)

Construct an IPv6 network definition. *address* can be one of the following:

1. A string consisting of an IP address and an optional prefix length, separated by a slash (`/`). The IP address is the network address, and the prefix length must be a single number, the *prefix*. If no prefix length is provided, it’s considered to be `/128`.

   Note that currently expanded netmasks are not supported. That means `2001:db00::0/24` is a valid argument while `2001:db00::0/ffff:ff00::` is not.

2. An integer that fits into 128 bits. This is equivalent to a single-address network, with the network address being *address* and the mask being `/128`.

3. An integer packed into a `bytes` object of length 16, big-endian. The interpretation is similar to an integer *address*.

4. A two-tuple of an address description and a netmask, where the address description is either a string, a 128-bits integer, a 16-bytes packed integer, or an existing `IPv6Address` object; and the netmask is an integer representing the prefix length.

An `AddressValueError` is raised if *address* is not a valid IPv6 address. A `NetmaskValueError` is raised if the mask is not valid for an IPv6 address.

If *strict* is `True` and host bits are set in the supplied address, then `ValueError` is raised. Otherwise, the host bits are masked out to determine the appropriate network address.

Changed in version 3.5: Added the two-tuple form for the *address* constructor parameter.

version

max_prefixlen

is_multicast

is_private

is_unspecified

is_reserved

is_loopback

is_link_local

network_address

broadcast_address

hostmask

netmask

with_prefixlen

compressed

exploded

with_netmask

with_hostmask

num_addresses

prefixlen

hosts()

Returns an iterator over the usable hosts in the network. The usable hosts are all the IP addresses that belong to the network, except the Subnet-Router anycast address. For networks with a mask length of 127, the Subnet-Router anycast address is also included in the result. Networks with a mask of 128 will return a list containing the single host address.

overlaps(*other*)

address_exclude(*network*)

subnets(*prefixlen_diff=1*, *new_prefix=None*)

supernet(*prefixlen_diff=1*, *new_prefix=None*)

subnet_of(*other*)

supernet_of(*other*)

compare_networks(*other*)

Refer to the corresponding attribute documentation in `IPv4Network`.

is_site_local

This attribute is true for the network as a whole if it is true for both the network address and the broadcast address.

Operators

Network objects support some operators. Unless stated otherwise, operators can only be applied between compatible objects (i.e. IPv4 with IPv4, IPv6 with IPv6).

Logical operators

Network objects can be compared with the usual set of logical operators. Network objects are ordered first by network address, then by net mask.

Iteration

Network objects can be iterated to list all the addresses belonging to the network. For iteration, *all* hosts are returned, including unusable hosts (for usable hosts, use the `hosts()` method). An example:

```
>>> for addr in IPv4Network('192.0.2.0/28'):
...     addr
...
IPv4Address('192.0.2.0')
IPv4Address('192.0.2.1')
IPv4Address('192.0.2.2')
IPv4Address('192.0.2.3')
IPv4Address('192.0.2.4')
IPv4Address('192.0.2.5')
IPv4Address('192.0.2.6')
IPv4Address('192.0.2.7')
IPv4Address('192.0.2.8')
IPv4Address('192.0.2.9')
IPv4Address('192.0.2.10')
IPv4Address('192.0.2.11')
IPv4Address('192.0.2.12')
IPv4Address('192.0.2.13')
IPv4Address('192.0.2.14')
IPv4Address('192.0.2.15')
```

Networks as containers of addresses

Network objects can act as containers of addresses. Some examples:

```
>>> IPv4Network('192.0.2.0/28')[0]
IPv4Address('192.0.2.0')
>>> IPv4Network('192.0.2.0/28')[15]
```

```
IPv4Address('192.0.2.15')
>>> IPv4Address('192.0.2.6') in IPv4Network('192.0.2.0/28')
True
>>> IPv4Address('192.0.3.6') in IPv4Network('192.0.2.0/28')
False
```

Interface objects

Interface objects are hashable, so they can be used as keys in dictionaries.

class ipaddress.IPv4Interface(*address*)

Construct an IPv4 interface. The meaning of *address* is as in the constructor of **`IPv4Network`**, except that arbitrary host addresses are always accepted.

`IPv4Interface` is a subclass of **`IPv4Address`**, so it inherits all the attributes from that class. In addition, the following attributes are available:

ip

The address (**`IPv4Address`**) without network information.

```
>>> interface = IPv4Interface('192.0.2.5/24')
>>> interface.ip
IPv4Address('192.0.2.5')
```

network

The network (**`IPv4Network`**) this interface belongs to.

```
>>> interface = IPv4Interface('192.0.2.5/24')
>>> interface.network
IPv4Network('192.0.2.0/24')
```

with_prefixlen

A string representation of the interface with the mask in prefix notation.

```
>>> interface = IPv4Interface('192.0.2.5/24')
>>> interface.with_prefixlen
'192.0.2.5/24'
```

with_netmask

A string representation of the interface with the network as a net mask.

```
>>> interface = IPv4Interface('192.0.2.5/24')
>>> interface.with_netmask
'192.0.2.5/255.255.255.0'
```

with_hostmask

A string representation of the interface with the network as a host mask.

```
>>> interface = IPv4Interface('192.0.2.5/24')
>>> interface.with_hostmask
'192.0.2.5/0.0.0.255'
```

class ipaddress.IPv6Interface(*address*)

Construct an IPv6 interface. The meaning of *address* is as in the constructor of **`IPv6Network`**, except that arbitrary host addresses are always accepted.

`IPv6Interface` is a subclass of **`IPv6Address`**, so it inherits all the attributes from that class. In addition, the following attributes are available:

ip

network

with_prefixlen

with_netmask

with_hostmask

Refer to the corresponding attribute documentation in `IPv4Interface`.

Operators

Interface objects support some operators. Unless stated otherwise, operators can only be applied between compatible objects (i.e. IPv4 with IPv4, IPv6 with IPv6).

Logical operators

Interface objects can be compared with the usual set of logical operators.

For equality comparison (`==` and `!=`), both the IP address and network must be the same for the objects to be equal. An interface will not compare equal to any address or network object.

For ordering (`<`, `>`, etc) the rules are different. Interface and address objects with the same IP version can be compared, and the address objects will always sort before the interface objects. Two interface objects are first compared by their networks and, if those are the same, then by their IP addresses.

Other Module Level Functions

The module also provides the following module level functions:

ipaddress.v4_int_to_packed(*address*)

Represent an address as 4 packed bytes in network (big-endian) order. *address* is an integer representation of an

IPv4 IP address. A `ValueError` is raised if the integer is negative or too large to be an IPv4 IP address.

```
>>> ipaddress.ip_address(3221225985)
IPv4Address('192.0.2.1')
>>> ipaddress.v4_int_to_packed(3221225985)
b'\xc0\x00\x02\x01'
```

ipaddress.v6_int_to_packed(*address*)

Represent an address as 16 packed bytes in network (big-endian) order. *address* is an integer representation of an IPv6 IP address. A `ValueError` is raised if the integer is negative or too large to be an IPv6 IP address.

ipaddress.summarize_address_range(*first*, *last*)

Return an iterator of the summarized network range given the first and last IP addresses. *first* is the first `IPv4Address` or `IPv6Address` in the range and *last* is the last `IPv4Address` or `IPv6Address` in the range. A `TypeError` is raised if *first* or *last* are not IP addresses or are not of the same version. A `ValueError` is raised if *last* is not greater than *first* or if *first* address version is not 4 or 6.

```
>>> [ipaddr for ipaddr in ipaddress.summarize_address_range(
...    ipaddress.IPv4Address('192.0.2.0'),
...    ipaddress.IPv4Address('192.0.2.130'))]
[IPv4Network('192.0.2.0/25'), IPv4Network('192.0.2.128/31'),
IPv4Network('192.0.2.130/32')]
```

ipaddress.collapse_addresses(*addresses*)

Return an iterator of the collapsed `IPv4Network` or `IPv6Network` objects. *addresses* is an iterable of `IPv4Network` or `IPv6Network` objects. A `TypeError` is raised if *addresses* contains mixed version objects.

```
>>> [ipaddr for ipaddr in
...
ipaddress.collapse_addresses([ipaddress.IPv4Network('192.0.2
.0/25'),
... ipaddress.IPv4Network('192.0.2.128/25')]))]
[IPv4Network('192.0.2.0/24')]
```

ipaddress.get_mixed_type_key(*obj*)

Return a key suitable for sorting between networks and addresses. Address and Network objects are not sortable by default; they're fundamentally different, so the expression:

```
IPv4Address('192.0.2.0') <= IPv4Network('192.0.2.0/24')
```

doesn't make sense. There are some times however, where you may wish to have **`ipaddress`** sort these anyway. If you need to do this, you can use this function as the *key* argument to **`sorted()`**.

obj is either a network or address object.

Custom Exceptions

To support more specific error reporting from class constructors, the module defines the following exceptions:

exception ipaddress.AddressValueError(*ValueError*)

Any value error related to the address.

exception ipaddress.NetmaskValueError(*ValueError*)

Any value error related to the net mask.

Multimedia Services

The modules described in this chapter implement various algorithms or interfaces that are mainly useful for multimedia applications. They are available at the discretion of the installation. Here’s an overview:

- **`wave`** — Read and write WAV files
  - Wave_read Objects
  - Wave_write Objects
- **`colorsys`** — Conversions between color systems

`wave` — Read and write WAV files

Source code: Lib/wave.py
[https://github.com/python/cpython/tree/3.14/Lib/wave.py]

The **`wave`** module provides a convenient interface to the Waveform Audio “WAVE” (or “WAV”) file format. Only uncompressed PCM encoded wave files are supported.

Changed in version 3.12: Support for `WAVE_FORMAT_EXTENSIBLE` headers was added, provided that the extended format is `KSDATAFORMAT_SUBTYPE_PCM`.

The **`wave`** module defines the following function and exception:

wave.open(*file*, *mode=None*)

If *file* is a string, open the file by that name, otherwise treat it as a file-like object. *mode* can be:

`'rb'`
Read only mode.

`'wb'`
Write only mode.

Note that it does not allow read/write WAV files.

A *mode* of `'rb'` returns a **`Wave_read`** object, while a *mode* of `'wb'` returns a **`Wave_write`** object. If *mode* is omitted and a file-like object is passed as *file*, `file.mode` is used as the default value for *mode*.

If you pass in a file-like object, the wave object will not close it when its `close()` method is called; it is the caller’s responsibility to close the file object.

The `open()` function may be used in a `with` statement. When the **`with`** block completes, the `Wave_read.close()` or `Wave_write.close()` method is called.

Changed in version 3.4: Added support for unseekable files.

exception wave.Error

An error raised when something is impossible because it violates the WAV specification or hits an implementation deficiency.

Wave_read Objects

class wave.Wave_read

Read a WAV file.

Wave_read objects, as returned by `open()`, have the following methods:

close()

Close the stream if it was opened by `wave`, and make the instance unusable. This is called automatically on object collection.

getnchannels()

Returns number of audio channels (`1` for mono, `2` for stereo).

getsampwidth()

Returns sample width in bytes.

getframerate()

Returns sampling frequency.

getnframes()

Returns number of audio frames.

getcomptype()

Returns compression type (`'NONE'` is the only supported type).

getcompname()

Human-readable version of [`getcomptype()`](). Usually `'not compressed'` parallels `'NONE'`.

getparams()

Returns a [`namedtuple()`]() `(nchannels, sampwidth, framerate, nframes, comptype, compname)`, equivalent to output of the `get*()` methods.

readframes(*n*)

Reads and returns at most *n* frames of audio, as a [`bytes`]() object.

rewind()

Rewind the file pointer to the beginning of the audio stream.

The following two methods are defined for compatibility with the old **`aifc`** module, and don’t do anything interesting.

getmarkers()

Returns `None`.

Deprecated since version 3.13, will be removed in version 3.15: The method only existed for compatibility with the **`aifc`** module which has been removed in Python 3.13.

getmark(*id*)

Raise an error.

Deprecated since version 3.13, will be removed in version 3.15: The method only existed for compatibility with the **`aifc`** module which has been removed in Python 3.13.

The following two methods define a term “position” which is compatible between them, and is otherwise implementation dependent.

setpos(*pos*)

Set the file pointer to the specified position.

tell()

Return current file pointer position.

Wave_write Objects

class wave.Wave_write

Write a WAV file.

Wave_write objects, as returned by **`open()`**.

For seekable output streams, the `wave` header will automatically be updated to reflect the number of frames actually written. For unseekable streams, the *nframes* value must be accurate when the first frame data is written. An accurate *nframes* value can be achieved either by calling `setnframes()` or `setparams()` with the number of frames that will be written before `close()` is called and then using `writeframesraw()` to write the frame data, or by calling `writeframes()` with all of the frame data to be written. In the latter case `writeframes()` will calculate the number of frames in the data and set *nframes* accordingly before writing the frame data.

Changed in version 3.4: Added support for unseekable files.

Wave_write objects have the following methods:

close()

Make sure *nframes* is correct, and close the file if it was opened by `wave`. This method is called upon object collection. It will raise an exception if the output stream is not seekable and *nframes* does not match the number of frames actually written.

setnchannels(*n*)

Set the number of channels.

setsampwidth(*n*)

Set the sample width to *n* bytes.

setframerate(*n*)

Set the frame rate to *n*.

Changed in version 3.2: A non-integral input to this method is rounded to the nearest integer.

setnframes(*n*)

Set the number of frames to *n*. This will be changed later if the number of frames actually written is different (this update attempt will raise an error if the output stream is not seekable).

setcomptype(*type*, *name*)

Set the compression type and description. At the moment, only compression type `NONE` is supported, meaning no compression.

setparams(*tuple*)

The *tuple* should be `(nchannels, sampwidth, framerate, nframes, comptype, compname)`, with values valid for the `set*()` methods. Sets all parameters.

tell()

Return current position in the file, with the same disclaimer for the **`Wave_read.tell()`** and **`Wave_read.setpos()`** methods.

writeframesraw(*data*)

Write audio frames, without correcting *nframes*.

Changed in version 3.4: Any bytes-like object is now accepted.

writeframes(*data*)

Write audio frames and make sure *nframes* is correct. It will raise an error if the output stream is not seekable and the total number of frames that have been written after *data* has been written does not match the previously set value for *nframes*.

Changed in version 3.4: Any bytes-like object is now accepted.

Note that it is invalid to set any parameters after calling **`writeframes()`** or **`writeframesraw()`**, and any attempt to do so will raise `wave.Error`.

`colorsys` — Conversions between color systems

Source code: Lib/colorsys.py [https://github.com/python/cpython/tree/3.14/Lib/colorsys.py]

The **`colorsys`** module defines bidirectional conversions of color values between colors expressed in the RGB (Red Green Blue) color space used in computer monitors and three other coordinate systems: YIQ, HLS (Hue Lightness Saturation) and HSV (Hue Saturation Value). Coordinates in all of these color spaces are floating-point values. In the YIQ space, the Y coordinate is between 0 and 1, but the I and Q coordinates can be positive or negative. In all other spaces, the coordinates are all between 0 and 1.

See also

More information about color spaces can be found at https://poynton.ca/ColorFAQ.html and https://www.cambridgeincolour.com/tutorials/color-spaces.htm.

The **`colorsys`** module defines the following functions:

colorsys.rgb_to_yiq(*r*, *g*, *b*)

Convert the color from RGB coordinates to YIQ coordinates.

colorsys.yiq_to_rgb(*y*, *i*, *q*)

Convert the color from YIQ coordinates to RGB coordinates.

colorsys.rgb_to_hls(*r*, *g*, *b*)

Convert the color from RGB coordinates to HLS coordinates.

colorsys.hls_to_rgb(*h*, *l*, *s*)

Convert the color from HLS coordinates to RGB coordinates.

colorsys.rgb_to_hsv(*r*, *g*, *b*)

Convert the color from RGB coordinates to HSV coordinates.

colorsys.hsv_to_rgb(*h*, *s*, *v*)

Convert the color from HSV coordinates to RGB coordinates.

Example:

```
>>> import colorsys
>>> colorsys.rgb_to_hsv(0.2, 0.4, 0.4)
(0.5, 0.5, 0.4)
>>> colorsys.hsv_to_rgb(0.5, 0.5, 0.4)
(0.2, 0.4, 0.4)
```

Internationalization

The modules described in this chapter help you write software that is independent of language and locale by providing mechanisms for selecting a language to be used in program messages or by tailoring output to match local conventions.

The list of modules described in this chapter is:

- `gettext` — Multilingual internationalization services
    - GNU **gettext** API
    - Class-based API
        - The `NullTranslations` class
        - The `GNUTranslations` class
        - Solaris message catalog support
        - The Catalog constructor
    - Internationalizing your programs and modules
        - Localizing your module
        - Localizing your application
        - Changing languages on the fly
        - Deferred translations
    - Acknowledgements
- `locale` — Internationalization services
    - Background, details, hints, tips and caveats
    - Locale names
    - For extension writers and programs that embed Python
    - Access to message catalogs

`gettext` — Multilingual internationalization services

Source code: Lib/gettext.py [https://github.com/python/cpython/tree/3.14/Lib/gettext.py]

The `gettext` module provides internationalization (I18N) and localization (L10N) services for your Python modules and applications. It supports both the GNU **gettext** message catalog API and a higher level, class-based API that may be more appropriate for Python files. The interface described below allows you to write your module and application messages in one natural language, and provide a catalog of translated messages for running under different natural languages.

Some hints on localizing your Python modules and applications are also given.

GNU gettext API

The `gettext` module defines the following API, which is very similar to the GNU **gettext** API. If you use this API you will affect the translation of your entire application globally. Often this is what you want if your application is monolingual, with the choice of language dependent on the locale of your user. If you are localizing a Python module, or if your application needs to switch languages on the fly, you probably want to use the class-based API instead.

gettext.bindtextdomain(*domain*, *localedir=None*)

Bind the *domain* to the locale directory *localedir*. More concretely, `gettext` will look for binary `.mo` files for the given domain using the path (on Unix): `localedir/language/LC_MESSAGES/domain.mo`, where *language* is searched for in the environment variables **`LANGUAGE`**, **`LC_ALL`**, **`LC_MESSAGES`**, and **`LANG`** respectively.

If *localedir* is omitted or `None`, then the current binding for *domain* is returned. [[1]]

gettext.textdomain(*domain=None*)

Change or query the current global domain. If *domain* is `None`, then the current global domain is returned, otherwise the global domain is set to *domain*, which is returned.

gettext.gettext(*message*)

Return the localized translation of *message*, based on the current global domain, language, and locale directory. This function is usually aliased as `_()` in the local namespace (see examples below).

gettext.dgettext(*domain*, *message*)

Like `gettext()`, but look the message up in the specified *domain*.

gettext.ngettext(*singular*, *plural*, *n*)

Like `gettext()`, but consider plural forms. If a translation is found, apply the plural formula to *n*, and return the resulting message (some languages have more than two plural forms). If no translation is found, return *singular* if *n* is 1; return *plural* otherwise.

The Plural formula is taken from the catalog header. It is a C or Python expression that has a free variable *n*; the expression evaluates to the index of the plural in the catalog. See the GNU gettext documentation [https://www.gnu.org/software/gettext/manual/gettext.html] for the precise syntax to be used in `.po` files and the formulas for a variety of languages.

gettext.dngettext(*domain*, *singular*, *plural*, *n*)

Like **`ngettext()`**, but look the message up in the specified *domain*.

gettext.pgettext(*context*, *message*)

gettext.dpgettext(*domain*, *context*, *message*)

gettext.npgettext(*context*, *singular*, *plural*, *n*)

gettext.dnpgettext(*domain*, *context*, *singular*, *plural*, *n*)

Similar to the corresponding functions without the `p` in the prefix (that is, **`gettext()`**, **`dgettext()`**, **`ngettext()`**, **`dngettext()`**), but the translation is restricted to the given message *context*.

Added in version 3.8.

Note that GNU **gettext** also defines a **`dcgettext()`** method, but this was deemed not useful and so it is currently unimplemented.

Here’s an example of typical usage for this API:

```
import gettext
gettext.bindtextdomain('myapplication',
'/path/to/my/language/directory')
gettext.textdomain('myapplication')
_ = gettext.gettext
```

```
# ...
print(_('This is a translatable string.'))
```

Class-based API

The class-based API of the `gettext` module gives you more flexibility and greater convenience than the GNU **gettext** API. It is the recommended way of localizing your Python applications and modules. **`gettext`** defines a `GNUTranslations` class which implements the parsing of GNU `.mo` format files, and has methods for returning strings. Instances of this class can also install themselves in the built-in namespace as the function **`_()`**.

gettext.find(*domain*, *localedir=None*, *languages=None*, *all=False*)

This function implements the standard `.mo` file search algorithm. It takes a *domain*, identical to what `textdomain()` takes. Optional *localedir* is as in `bindtextdomain()`. Optional *languages* is a list of strings, where each string is a language code.

If *localedir* is not given, then the default system locale directory is used. [[2]] If *languages* is not given, then the following environment variables are searched: **`LANGUAGE`**, **`LC_ALL`**, **`LC_MESSAGES`**, and **`LANG`**. The first one returning a non-empty value is used for the *languages* variable. The environment variables should contain a colon separated list of languages, which will be split on the colon to produce the expected list of language code strings.

`find()` then expands and normalizes the languages, and then iterates through them, searching for an existing file built of these components:

`localedir/language/LC_MESSAGES/domain.mo`

The first such file name that exists is returned by `find()`. If no such file is found, then `None` is returned. If *all* is given, it returns a list of all file names, in the order in which they appear in the languages list or the environment variables.

gettext.translation(*domain*, *localedir=None*, *languages=None*, *class_=None*, *fallback=False*)

Return a `*Translations` instance based on the *domain*, *localedir*, and *languages*, which are first passed to `find()` to get a list of the associated `.mo` file paths. Instances with identical `.mo` file names are cached. The actual class instantiated is *class_* if provided, otherwise `GNUTranslations`. The class's constructor must take a single file object argument.

If multiple files are found, later files are used as fallbacks for earlier ones. To allow setting the fallback, `copy.copy()` is used to clone each translation object from the cache; the actual instance data is still shared with the cache.

If no `.mo` file is found, this function raises `OSError` if *fallback* is false (which is the default), and returns a `NullTranslations` instance if *fallback* is true.

Changed in version 3.3: `IOError` used to be raised, it is now an alias of `OSError`.

Changed in version 3.11: codeset parameter is removed.

gettext.install(*domain*, *localedir=None*, *, *names=None*)

This installs the function `_()` in Python's builtins namespace, based on *domain* and *localedir* which are passed to the function `translation()`.

For the *names* parameter, please see the description of the translation object's `install()` method.

As seen below, you usually mark the strings in your application that are candidates for translation, by wrapping them in a call to the **`_()`** function, like this:

```
print(_('This string will be translated.'))
```

For convenience, you want the **`_()`** function to be installed in Python's builtins namespace, so it is easily accessible in all modules of your application.

Changed in version 3.11: names is now a keyword-only parameter.

The `NullTranslations` class

Translation classes are what actually implement the translation of original source file message strings to translated message strings. The base class used by all translation classes is `NullTranslations`; this provides the basic interface you can use to write your own specialized translation classes. Here are the methods of **`NullTranslations`**:

class gettext.NullTranslations(*fp=None*)

Takes an optional file object *fp*, which is ignored by the base class. Initializes "protected" instance variables *_info* and *_charset* which are set by derived classes, as well as *_fallback*, which is set through `add_fallback()`. It then calls `self._parse(fp)` if *fp* is not `None`.

_parse(*fp*)

No-op in the base class, this method takes file object *fp*, and reads the data from the file, initializing its message catalog. If you have an unsupported message catalog file format, you should override this method to parse your format.

add_fallback(*fallback*)

Add *fallback* as the fallback object for the current translation object. A translation object should consult the fallback if it cannot provide a translation for a given message.

gettext(*message*)

If a fallback has been set, forward **`gettext()`** to the fallback. Otherwise, return *message*. Overridden in derived classes.

ngettext(*singular*, *plural*, *n*)

If a fallback has been set, forward **`ngettext()`** to the fallback. Otherwise, return *singular* if *n* is 1; return *plural* otherwise. Overridden in derived classes.

pgettext(*context*, *message*)

If a fallback has been set, forward `pgettext()` to the fallback. Otherwise, return the translated message. Overridden in derived classes.

Added in version 3.8.

npgettext(*context*, *singular*, *plural*, *n*)

If a fallback has been set, forward `npgettext()` to the fallback. Otherwise, return the translated message. Overridden in derived classes.

Added in version 3.8.

info()

Return a dictionary containing the metadata found in the message catalog file.

charset()

Return the encoding of the message catalog file.

install(*names=None*)

This method installs `gettext()` into the built-in namespace, binding it to _.

If the *names* parameter is given, it must be a sequence containing the names of functions you want to install in the builtins namespace in addition to `_()`. Supported names are `'gettext'`, `'ngettext'`, `'pgettext'`, and `'npgettext'`.

Note that this is only one way, albeit the most convenient way, to make the `_()` function available to your application. Because it affects the entire application globally, and specifically the built-in namespace, localized modules should never install `_()`. Instead, they should use this code to make `_()` available to their module:

```
import gettext
t = gettext.translation('mymodule', ...)
_ = t.gettext
```

This puts `_()` only in the module’s global namespace and so only affects calls within this module.

Changed in version 3.8: Added `'pgettext'` and `'npgettext'`.

The `GNUTranslations` class

The **`gettext`** module provides one additional class derived from [`NullTranslations`](): [`GNUTranslations`](). This class overrides **`_parse()`** to enable reading GNU **gettext** format `.mo` files in both big-endian and little-endian format.

[`GNUTranslations`]() parses optional metadata out of the translation catalog. It is convention with GNU **gettext** to include metadata as the translation for the empty string. This metadata is in **RFC 822** [https://datatracker.ietf.org/doc/html/rfc822.html]-style `key: value` pairs, and should contain the `Project-Id-Version` key. If the key `Content-Type` is found, then the `charset` property is used to initialize the “protected” **`_charset`** instance variable, defaulting to `None` if not found. If the charset encoding is specified, then all message ids and message strings read from the catalog are converted to Unicode using this encoding, else ASCII is assumed.

Since message ids are read as Unicode strings too, all `*gettext()` methods will assume message ids as Unicode strings, not byte strings.

The entire set of key/value pairs are placed into a dictionary and set as the “protected” **`_info`** instance variable.

If the `.mo` file’s magic number is invalid, the major version number is unexpected, or if other problems occur while reading the file, instantiating a **`GNUTranslations`** class can raise **`OSError`**.

class gettext.GNUTranslations

The following methods are overridden from the base class implementation:

gettext(*message*)

Look up the *message* id in the catalog and return the corresponding message string, as a Unicode string. If there is no entry in the catalog for the *message* id, and a fallback has been set, the look up is forwarded to the fallback's `gettext()` method. Otherwise, the *message* id is returned.

ngettext(*singular*, *plural*, *n*)

Do a plural-forms lookup of a message id. *singular* is used as the message id for purposes of lookup in the catalog, while *n* is used to determine which plural form to use. The returned message string is a Unicode string.

If the message id is not found in the catalog, and a fallback is specified, the request is forwarded to the fallback's `ngettext()` method. Otherwise, when *n* is 1 *singular* is returned, and *plural* is returned in all other cases.

Here is an example:

```
n = len(os.listdir('.'))
cat = GNUTranslations(somefile)
message = cat.ngettext(
    'There is %(num)d file in this directory',
    'There are %(num)d files in this directory',
    n) % {'num': n}
```

pgettext(*context*, *message*)

Look up the *context* and *message* id in the catalog and return the corresponding message string, as a Unicode string. If there is no entry in the catalog for the *message* id and *context*, and a fallback has been set, the look up is forwarded to the fallback's `pgettext()` method. Otherwise, the *message* id is returned.

Added in version 3.8.

npgettext(*context*, *singular*, *plural*, *n*)

Do a plural-forms lookup of a message id. *singular* is used as the message id for purposes of lookup in the catalog, while *n* is used to determine which plural form to use.

If the message id for *context* is not found in the catalog, and a fallback is specified, the request is forwarded to the fallback's `npgettext()` method. Otherwise, when *n* is 1 *singular* is returned, and *plural* is returned in all other cases.

Added in version 3.8.

Solaris message catalog support

The Solaris operating system defines its own binary `.mo` file format, but since no documentation can be found on this format, it is not supported at this time.

The Catalog constructor

GNOME uses a version of the `gettext` module by James Henstridge, but this version has a slightly different API. Its documented usage was:

```
import gettext
cat = gettext.Catalog(domain, localedir)
_ = cat.gettext
print(_('hello world'))
```

For compatibility with this older module, the function **`Catalog()`** is an alias for the `translation()` function described above.

One difference between this module and Henstridge’s: his catalog objects supported access through a mapping API, but this appears to be unused and so is not currently supported.

Internationalizing your programs and modules

Internationalization (I18N) refers to the operation by which a program is made aware of multiple languages. Localization (L10N) refers to the adaptation of your program, once internationalized, to the local language and cultural habits. In order to provide multilingual messages for your Python programs, you need to take the following steps:

1. prepare your program or module by specially marking translatable strings
2. run a suite of tools over your marked files to generate raw messages catalogs
3. create language-specific translations of the message catalogs
4. use the `gettext` module so that message strings are properly translated

In order to prepare your code for I18N, you need to look at all the strings in your files. Any string that needs to be translated should be marked by wrapping it in `_('...')` — that is, a call to the function `_`. For example:

```
filename = 'mylog.txt'
message = _('writing a log message')
with open(filename, 'w') as fp:
    fp.write(message)
```

In this example, the string `'writing a log message'` is marked as a candidate for translation, while the strings `'mylog.txt'` and `'w'` are not.

There are a few tools to extract the strings meant for translation. The original GNU **gettext** only supported C or C++ source code but its extended version **xgettext** scans code written in a number of languages, including Python, to find strings marked as translatable. Babel [https://babel.pocoo.org/] is a Python internationalization library that includes a `pybabel` script to extract and compile message catalogs. François Pinard’s program called **xpot** does a similar job and is available as part of his po-utils package [https://github.com/pinard/po-utils].

(Python also includes pure-Python versions of these programs, called **pygettext.py** and **msgfmt.py**; some Python distributions will install them for you. **pygettext.py** is similar to **xgettext**, but only understands Python source code and cannot handle other programming languages such as C or C++. **pygettext.py** supports a command-line interface similar to **xgettext**; for details on its use, run `pygettext.py --help`. **msgfmt.py** is binary compatible with GNU **msgfmt**. With these two programs, you may not need the GNU **gettext** package to internationalize your Python applications.)

xgettext, **pygettext**, and similar tools generate `.po` files that are message catalogs. They are structured human-readable files that contain every marked string in the source code, along with a placeholder for the translated versions of these strings.

Copies of these `.po` files are then handed over to the individual human translators who write translations for every supported natural language. They send back the completed language-specific versions as a `<language-name>.po` file that’s compiled into a machine-readable `.mo` binary catalog file using the **msgfmt** program. The `.mo` files are used by the `gettext` module for the actual translation processing at run-time.

How you use the `gettext` module in your code depends on whether you are internationalizing a single module or your entire application. The next two sections will discuss each case.

Localizing your module

If you are localizing your module, you must take care not to make global changes, e.g. to the built-in namespace. You should not use the GNU **gettext** API but instead the class-based API.

Let’s say your module is called “spam” and the module’s various natural language translation `.mo` files reside in `/usr/share/locale` in GNU **gettext** format. Here’s what you would put at the top of your module:

```
import gettext
t = gettext.translation('spam', '/usr/share/locale')
_ = t.gettext
```

Localizing your application

If you are localizing your application, you can install the `_()` function globally into the built-in namespace, usually in the main driver file of your application. This will let all your application-specific files just use `_('...')` without having to explicitly install it in each file.

In the simple case then, you need only add the following bit of code to the main driver file of your application:

```
import gettext
gettext.install('myapplication')
```

If you need to set the locale directory, you can pass it into the `install()` function:

```
import gettext
gettext.install('myapplication', '/usr/share/locale')
```

Changing languages on the fly

If your program needs to support many languages at the same time, you may want to create multiple translation instances and then switch between them explicitly, like so:

```
import gettext

lang1 = gettext.translation('myapplication', languages=['en'])
lang2 = gettext.translation('myapplication', languages=['fr'])
lang3 = gettext.translation('myapplication', languages=['de'])

# start by using language1
lang1.install()

# ... time goes by, user selects language 2
lang2.install()

# ... more time goes by, user selects language 3
lang3.install()
```

Deferred translations

In most coding situations, strings are translated where they are coded. Occasionally however, you need to mark strings for translation, but defer actual translation until later. A classic example is:

```
animals = ['mollusk',
           'albatross',
           'rat',
           'penguin',
           'python', ]
# ...
for a in animals:
    print(a)
```

Here, you want to mark the strings in the `animals` list as being translatable, but you don’t actually want to translate them until they are printed.

Here is one way you can handle this situation:

```
def _(message): return message

animals = [_('mollusk'),
           _('albatross'),
           _('rat'),
           _('penguin'),
           _('python'), ]

del _

# ...
for a in animals:
    print(_(a))
```

This works because the dummy definition of `_()` simply returns the string unchanged. And this dummy definition will temporarily override any definition of `_()` in the built-in namespace (until the `del` command). Take care, though if you have a previous definition of `_()` in the local namespace.

Note that the second use of `_()` will not identify “a” as being translatable to the **gettext** program, because the parameter is not a string literal.

Another way to handle this is with the following example:

```
def N_(message): return message
```

```
animals = [N_('mollusk'),
           N_('albatross'),
           N_('rat'),
           N_('penguin'),
           N_('python'), ]

# ...
for a in animals:
    print(_(a))
```

In this case, you are marking translatable strings with the function **`N_()`**, which won't conflict with any definition of **`_()`**. However, you will need to teach your message extraction program to look for translatable strings marked with **`N_()`**. **xgettext**, **pygettext**, `pybabel extract`, and **xpot** all support this through the use of the **`-k`** command-line switch. The choice of **`N_()`** here is totally arbitrary; it could have just as easily been **`MarkThisStringForTranslation()`**.

Acknowledgements

The following people contributed code, feedback, design suggestions, previous implementations, and valuable experience to the creation of this module:

- Peter Funk
- James Henstridge
- Juan David Ibáñez Palomar
- Marc-André Lemburg
- Martin von Löwis
- François Pinard
- Barry Warsaw
- Gustavo Niemeyer

Footnotes

[1]

The default locale directory is system dependent; for example, on Red Hat Linux it is `/usr/share/locale`, but on Solaris it is `/usr/lib/locale`. The **`gettext`** module does not try to support these system dependent defaults; instead its default is `*sys.base_prefix*/share/locale` (see **`sys.base_prefix`**). For this reason, it is always best to call **`bindtextdomain()`** with an explicit absolute path at the start of your application. [2]

See the footnote for **`bindtextdomain()`** above.

`locale` — Internationalization services

Source code: Lib/locale.py
[https://github.com/python/cpython/tree/3.14/Lib/locale.py]

The **`locale`** module opens access to the POSIX locale database and functionality. The POSIX locale mechanism allows programmers to deal with certain cultural issues in an application, without requiring the programmer to know all the specifics of each country where the software is executed.

The **`locale`** module is implemented on top of the **`_locale`** module, which in turn uses an ANSI C locale implementation if available.

The **`locale`** module defines the following exception and functions:

exception locale.Error

Exception raised when the locale passed to **`setlocale()`** is not recognized.

locale.setlocale(*category*, *locale=None*)

If *locale* is given and not `None`, **`setlocale()`** modifies the locale setting for the *category*. The available categories are listed in the data description below. *locale* may be a string, or a pair, language code and encoding. An empty string specifies the user’s default settings. If the modification of the locale fails, the exception **`Error`** is raised. If successful, the new locale setting is returned.

If *locale* is a pair, it is converted to a locale name using the locale aliasing engine. The language code has the same format as a locale name, but without encoding and `@`-modifier. The language code and encoding can be `None`.

If *locale* is omitted or `None`, the current setting for *category* is returned.

Example:

```
>>> import locale
>>> loc = locale.setlocale(locale.LC_ALL)  # get current
locale
# use German locale; name and availability varies with
platform
>>> locale.setlocale(locale.LC_ALL, 'de_DE.UTF-8')
>>> locale.strcoll('f\xe4n', 'foo')  # compare a string
containing an umlaut
>>> locale.setlocale(locale.LC_ALL, '')   # use user's
preferred locale
>>> locale.setlocale(locale.LC_ALL, 'C')  # use default (C)
locale
>>> locale.setlocale(locale.LC_ALL, loc)  # restore saved
locale
```

`setlocale()` is not thread-safe on most systems. Applications typically start with a call of:

```
import locale
locale.setlocale(locale.LC_ALL, '')
```

This sets the locale for all categories to the user’s default setting (typically specified in the **`LANG`** environment variable). If the locale is not changed thereafter, using multithreading should not cause problems.

locale.localeconv()

Returns the database of the local conventions as a dictionary. This dictionary has the following strings as keys:

Category	Key	Meaning
`LC_NUMERIC`	`'decimal_point'`	Decimal point character.
	`'grouping'`	Sequence of numbers specifying which relative positions the `'thousands_sep'` is expected. If the sequence is terminated with `CHAR_MAX`, no further grouping is performed. If the sequence terminates with a `0`, the last group size is repeatedly used.
	`'thousands_sep'`	Character used between groups.
`LC_MONETARY`	`'int_curr_symbol'`	International currency

Category	Key	Meaning
		symbol.
	`'currency_symbol'`	Local currency symbol.
	`'p_cs_precedes/n_cs_precedes'`	Whether the currency symbol precedes the value (for positive resp. negative values).
	`'p_sep_by_space/n_sep_by_space'`	Whether the currency symbol is separated from the value by a space (for positive resp. negative values).
	`'mon_decimal_point'`	Decimal point used for monetary values.
	`'frac_digits'`	Number of fractional digits used in

Category	Key	Meaning
		local formatting of monetary values.
	`'int_frac_digits'`	Number of fractional digits used in international formatting of monetary values.
	`'mon_thousands_sep'`	Group separator used for monetary values.
	`'mon_grouping'`	Equivalent to `'grouping'`, used for monetary values.
	`'positive_sign'`	Symbol used to annotate a positive monetary value.
	`'negative_sign'`	Symbol used to annotate a negative

Category	Key	Meaning
		monetary value.
	`'p_sign_posn/n_sign_posn'`	The position of the sign (for positive resp. negative values), see below.

All numeric values can be set to `CHAR_MAX` to indicate that there is no value specified in this locale.

The possible values for `'p_sign_posn'` and `'n_sign_posn'` are given below.

Value	Explanation
`0`	Currency and value are surrounded by parentheses.
`1`	The sign should precede the value and currency symbol.
`2`	The sign should follow the value and currency symbol.
`3`	The sign should immediately precede the value.
`4`	The sign should immediately follow the value.
`CHAR_MAX`	Nothing is specified in this locale.

The function temporarily sets the `LC_CTYPE` locale to the `LC_NUMERIC` locale or the `LC_MONETARY` locale if locales are different and numeric or monetary strings are non-ASCII. This temporary change affects other threads.

Changed in version 3.7: The function now temporarily sets the `LC_CTYPE` locale to the `LC_NUMERIC` locale in some cases.

locale.nl_langinfo(*option*)

Return some locale-specific information as a string. This function is not available on all systems, and the set of possible options might also vary across platforms. The possible argument values are numbers, for which symbolic constants are available in the locale module.

The **`nl_langinfo()`** function accepts one of the following keys. Most descriptions are taken from the corresponding description in the GNU C library.

locale.CODESET

Get a string with the name of the character encoding used in the selected locale.

locale.D_T_FMT

Get a string that can be used as a format string for **`time.strftime()`** to represent date and time in a locale-specific way.

locale.D_FMT

Get a string that can be used as a format string for **`time.strftime()`** to represent a date in a locale-specific way.

locale.T_FMT

Get a string that can be used as a format string for `time.strftime()` to represent a time in a locale-specific way.

locale.T_FMT_AMPM

Get a format string for `time.strftime()` to represent time in the am/pm format.

locale.DAY_1
locale.DAY_2
locale.DAY_3
locale.DAY_4
locale.DAY_5
locale.DAY_6
locale.DAY_7

Get the name of the n-th day of the week.

Note

This follows the US convention of `DAY_1` being Sunday, not the international convention (ISO 8601) that Monday is the first day of the week.

locale.ABDAY_1
locale.ABDAY_2
locale.ABDAY_3
locale.ABDAY_4
locale.ABDAY_5
locale.ABDAY_6
locale.ABDAY_7

Get the abbreviated name of the n-th day of the week.

locale.MON_1
locale.MON_2
locale.MON_3
locale.MON_4

locale.MON_5
locale.MON_6
locale.MON_7
locale.MON_8
locale.MON_9
locale.MON_10
locale.MON_11
locale.MON_12

Get the name of the n-th month.

locale.ABMON_1
locale.ABMON_2
locale.ABMON_3
locale.ABMON_4
locale.ABMON_5
locale.ABMON_6
locale.ABMON_7
locale.ABMON_8
locale.ABMON_9
locale.ABMON_10
locale.ABMON_11
locale.ABMON_12

Get the abbreviated name of the n-th month.

locale.RADIXCHAR

Get the radix character (decimal dot, decimal comma, etc.).

locale.THOUSEP

Get the separator character for thousands (groups of three digits).

locale.YESEXPR

Get a regular expression that can be used with the regex function to recognize a positive response to a yes/no question.

locale.NOEXPR

Get a regular expression that can be used with the `regex(3)` function to recognize a negative response to a yes/no question.

Note

The regular expressions for **`YESEXPR`** and **`NOEXPR`** use syntax suitable for the `regex` function from the C library, which might differ from the syntax used in **`re`**.

locale.CRNCYSTR

Get the currency symbol, preceded by “-” if the symbol should appear before the value, “+” if the symbol should appear after the value, or “.” if the symbol should replace the radix character.

locale.ERA

Get a string which describes how years are counted and displayed for each era in a locale.

Most locales do not define this value. An example of a locale which does define this value is the Japanese one. In Japan, the traditional representation of dates includes the name of the era corresponding to the then-emperor’s reign.

Normally it should not be necessary to use this value directly. Specifying the `E` modifier in their format strings causes the **`time.strftime()`** function to use this information. The format of the returned string is specified in *The Open Group Base Specifications Issue 8*, paragraph 7.3.5.2 LC_TIME C-Language Access [https://pubs.opengroup.org/onlinepubs/9799919799/basedefs/V1_chap07.html#tag_07_03_05_02].

locale.ERA_D_T_FMT

Get a format string for `time.strftime()` to represent date and time in a locale-specific era-based way.

locale.ERA_D_FMT

Get a format string for `time.strftime()` to represent a date in a locale-specific era-based way.

locale.ERA_T_FMT

Get a format string for `time.strftime()` to represent a time in a locale-specific era-based way.

locale.ALT_DIGITS

Get a string consisting of up to 100 semicolon-separated symbols used to represent the values 0 to 99 in a locale-specific way. In most locales this is an empty string.

The function temporarily sets the `LC_CTYPE` locale to the locale of the category that determines the requested value (`LC_TIME`, `LC_NUMERIC`, `LC_MONETARY` or `LC_MESSAGES`) if locales are different and the resulting string is non-ASCII. This temporary change affects other threads.

Changed in version 3.14: The function now temporarily sets the `LC_CTYPE` locale in some cases.

locale.getdefaultlocale([*envvars*])

Tries to determine the default locale settings and returns them as a tuple of the form `(language code, encoding)`.

According to POSIX, a program which has not called `setlocale(LC_ALL, '')` runs using the portable `'C'` locale. Calling `setlocale(LC_ALL, '')` lets it use the default locale as defined by the **`LANG`** variable. Since we do not want to interfere with the current locale setting we thus emulate the behavior in the way described above.

To maintain compatibility with other platforms, not only the **`LANG`** variable is tested, but a list of variables given as envvars parameter. The first found to be defined will be used. *envvars* defaults to the search path used in GNU gettext; it must always contain the variable name `'LANG'`. The GNU gettext search path contains `'LC_ALL'`, `'LC_CTYPE'`, `'LANG'` and `'LANGUAGE'`, in that order.

The language code has the same format as a locale name, but without encoding and `@`-modifier. The language code and encoding may be `None` if their values cannot be determined. The “C” locale is represented as `(None, None)`.

Deprecated since version 3.11, will be removed in version 3.15.

locale.getlocale(*category=LC_CTYPE*)

Returns the current setting for the given locale category as a tuple containing the language code and encoding. *category* may be one of the **`LC_*`** values except **`LC_ALL`**. It defaults to **`LC_CTYPE`**.

The language code has the same format as a locale name, but without encoding and `@`-modifier. The language code and encoding may be `None` if their values cannot be determined. The “C” locale is represented as `(None, None)`.

locale.getpreferredencoding(*do_setlocale=True*)

Return the locale encoding used for text data, according to user preferences. User preferences are expressed differently on different systems, and might not be available programmatically on some systems, so this function only returns a guess.

On some systems, it is necessary to invoke **`setlocale()`** to obtain the user preferences, so this function is not thread-safe. If invoking setlocale is not necessary or desired, *do_setlocale* should be set to `False`.

On Android or if the Python UTF-8 Mode is enabled, always return `'utf-8'`, the locale encoding and the *do_setlocale* argument are ignored.

The Python preinitialization configures the LC_CTYPE locale. See also the filesystem encoding and error handler.

Changed in version 3.7: The function now always returns `"utf-8"` on Android or if the Python UTF-8 Mode is enabled.

locale.getencoding()

Get the current locale encoding:

- On Android and VxWorks, return `"utf-8"`.
- On Unix, return the encoding of the current **`LC_CTYPE`** locale. Return `"utf-8"` if `nl_langinfo(CODESET)` returns an empty string: for example, if the current LC_CTYPE locale is not supported.
- On Windows, return the ANSI code page.

The Python preinitialization configures the LC_CTYPE locale. See also the filesystem encoding and error handler.

This function is similar to **`getpreferredencoding(False)`** except this function ignores the Python UTF-8 Mode.

Added in version 3.11.

locale.normalize(*localename*)

Returns a normalized locale code for the given locale name. The returned locale code is formatted for use with `setlocale()`. If normalization fails, the original name is returned unchanged.

If the given encoding is not known, the function defaults to the default encoding for the locale code just like `setlocale()`.

locale.strcoll(*string1*, *string2*)

Compares two strings according to the current `LC_COLLATE` setting. As any other compare function, returns a negative, or a positive value, or `0`, depending on whether *string1* collates before or after *string2* or is equal to it.

locale.strxfrm(*string*)

Transforms a string to one that can be used in locale-aware comparisons. For example, `strxfrm(s1) < strxfrm(s2)` is equivalent to `strcoll(s1, s2) < 0`. This function can be used when the same string is compared repeatedly, e.g. when collating a sequence of strings.

locale.format_string(*format*, *val*, *grouping=False*, *monetary=False*)

Formats a number *val* according to the current `LC_NUMERIC` setting. The format follows the conventions of the `%` operator. For floating-point values, the decimal point is modified if appropriate. If *grouping* is `True`, also takes the grouping into account.

If *monetary* is true, the conversion uses monetary thousands separator and grouping strings.

Processes formatting specifiers as in `format % val`, but takes the current locale settings into account.

Changed in version 3.7: The *monetary* keyword parameter was added.

locale.currency(*val*, *symbol=True*, *grouping=False*, *international=False*)

Formats a number *val* according to the current `LC_MONETARY` settings.

The returned string includes the currency symbol if *symbol* is true, which is the default. If *grouping* is `True` (which is not the default), grouping is done with the value. If *international* is `True` (which is not the default), the international currency symbol is used.

Note

This function will not work with the ‘C’ locale, so you have to set a locale via `setlocale()` first.

locale.str(*float*)

Formats a floating-point number using the same format as the built-in function `str(float)`, but takes the decimal point into account.

locale.delocalize(*string*)

Converts a string into a normalized number string, following the `LC_NUMERIC` settings.

Added in version 3.5.

locale.localize(*string*, *grouping=False*, *monetary=False*)

Converts a normalized number string into a formatted string following the `LC_NUMERIC` settings.

Added in version 3.10.

locale.atof(*string*, *func=float*)

Converts a string to a number, following the `LC_NUMERIC` settings, by calling *func* on the result of calling `delocalize()` on *string*.

locale.atoi(*string*)

Converts a string to an integer, following the `LC_NUMERIC` conventions.

locale.LC_CTYPE

Locale category for the character type functions. Most importantly, this category defines the text encoding, i.e. how bytes are interpreted as Unicode codepoints. See **PEP 538** [https://peps.python.org/pep-0538/] and **PEP 540** [https://peps.python.org/pep-0540/] for how this variable might be automatically coerced to `C.UTF-8` to avoid issues created by invalid settings in containers or incompatible settings passed over remote SSH connections.

Python doesn’t internally use locale-dependent character transformation functions from `ctype.h`. Instead, `pyctype.h` provides locale-independent equivalents like `Py_TOLOWER`.

locale.LC_COLLATE

Locale category for sorting strings. The functions `strcoll()` and `strxfrm()` of the `locale` module are affected.

locale.LC_TIME

Locale category for the formatting of time. The function `time.strftime()` follows these conventions.

locale.LC_MONETARY

Locale category for formatting of monetary values. The available options are available from the `localeconv()` function.

locale.LC_MESSAGES

Locale category for message display. Python currently does not support application specific locale-aware messages. Messages displayed by the operating system, like those returned by `os.strerror()` might be affected by this category.

This value may not be available on operating systems not conforming to the POSIX standard, most notably Windows.

locale.LC_NUMERIC

Locale category for formatting numbers. The functions `format_string()`, `atoi()`, `atof()` and `str()` of the `locale` module are affected by that category. All other numeric formatting operations are not affected.

locale.LC_ALL

Combination of all locale settings. If this flag is used when the locale is changed, setting the locale for all categories is attempted. If that fails for any category, no category is changed at all. When the locale is retrieved using this flag, a string indicating the setting for all categories is returned. This string can be later used to restore the settings.

locale.CHAR_MAX

This is a symbolic constant used for different values returned by `localeconv()`.

Background, details, hints, tips and caveats

The C standard defines the locale as a program-wide property that may be relatively expensive to change. On top of that, some implementations are broken in such a way that frequent locale changes may cause core dumps. This makes the locale somewhat painful to use correctly.

Initially, when a program is started, the locale is the `C` locale, no matter what the user’s preferred locale is. There is one exception: the **`LC_CTYPE`** category is changed at startup to set the current locale encoding to the user’s preferred locale encoding. The program must explicitly say that it wants the user’s preferred locale settings for other categories by calling `setlocale(LC_ALL, '')`.

It is generally a bad idea to call **`setlocale()`** in some library routine, since as a side effect it affects the entire program. Saving and restoring it is almost as bad: it is expensive and affects other threads that happen to run before the settings have been restored.

If, when coding a module for general use, you need a locale independent version of an operation that is affected by the locale (such as certain formats used with **`time.strftime()`**), you will have to find a way to do it without using the standard library routine. Even better is convincing yourself that using locale settings is okay. Only as a last resort should you document that your module is not compatible with non-`C` locale settings.

The only way to perform numeric operations according to the locale is to use the special functions defined by this module: **`atof()`**, **`atoi()`**, **`format_string()`**, **`str()`**.

There is no way to perform case conversions and character classifications according to the locale. For (Unicode) text strings these are done according to the character value only, while for byte strings, the conversions and classifications are done according to the ASCII value of the byte, and bytes whose high bit is set (i.e., non-ASCII bytes) are never converted or considered part of a character class such as letter or whitespace.

Locale names

The format of the locale name is platform dependent, and the set of supported locales can depend on the system configuration.

On Posix platforms, it usually has the format [[1]]:

```
language ["_" territory] ["." charset] ["@" modifier]
```

where *language* is a two- or three-letter language code from ISO 639 [https://www.iso.org/iso-639-language-code], *territory* is a two-letter country or region code from ISO 3166 [https://www.iso.org/iso-3166-country-codes.html], *charset* is a locale encoding, and *modifier* is a script name, a language subtag, a sort order identifier, or other locale modifier (for example, “latin”, “valencia”, “stroke” and “euro”).

On Windows, several formats are supported. [[2]] [[3]] A subset of IETF BCP 47 [https://www.rfc-editor.org/info/bcp47] tags:

```
language ["-" script] ["-" territory] ["." charset]
language ["-" script] "-" territory "-" modifier
```

where *language* and *territory* have the same meaning as in Posix, *script* is a four-letter script code from ISO 15924 [https://www.unicode.org/iso15924/], and *modifier* is a language subtag, a sort order identifier or custom modifier (for example, “valencia”, “stroke” or “x-python”). Both hyphen (`'-'`) and underscore (`'_'`) separators are supported. Only UTF-8 encoding is allowed for BCP 47 tags.

Windows also supports locale names in the format:

```
language ["_" territory] ["." charset]
```

where *language* and *territory* are full names, such as “English” and “United States”, and *charset* is either a code page number (for example, “1252”) or UTF-8. Only the underscore separator is supported in this format.

The “C” locale is supported on all platforms.

[1]

IEEE Std 1003.1-2024; 8.2 Internationalization Variables [https://pubs.opengroup.org/onlinepubs/9799919799/basedefs/V1_chap08.html#tag_08_02]

[2]

UCRT Locale names, Languages, and Country/Region strings [https://learn.microsoft.com/en-us/cpp/c-runtime-library/locale-names-languages-and-country-region-strings]

[3]

Locale Names [https://learn.microsoft.com/en-us/windows/win32/intl/locale-names]

For extension writers and programs that embed Python

Extension modules should never call **`setlocale()`**, except to find out what the current locale is. But since the return value can only be used portably to restore it, that is not very useful (except perhaps to find out whether or not the locale is `C`).

When Python code uses the **`locale`** module to change the locale, this also affects the embedding application. If the embedding application doesn’t want this to happen, it should remove the **`_locale`** extension module (which does all the work) from the table of built-in modules in the `config.c` file, and make sure that the **`_locale`** module is not accessible as a shared library.

Access to message catalogs

locale.gettext(*msg*)

locale.dgettext(*domain*, *msg*)

locale.dcgettext(*domain*, *msg*, *category*)

locale.textdomain(*domain*)

locale.bindtextdomain(*domain*, *dir*)

locale.bind_textdomain_codeset(*domain*, *codeset*)

The locale module exposes the C library’s gettext interface on systems that provide this interface. It consists of the functions **`gettext()`**, **`dgettext()`**, **`dcgettext()`**, **`textdomain()`**, **`bindtextdomain()`**, and **`bind_textdomain_codeset()`**. These are similar to the same functions in the **`gettext`** module, but use the C library’s binary format for message catalogs, and the C library’s search algorithms for locating message catalogs.

Python applications should normally find no need to invoke these functions, and should use **`gettext`** instead. A known exception to this rule are applications that link with additional C libraries which internally invoke C functions `gettext` or `dcgettext`. For these applications, it may be necessary to bind the text domain, so that the libraries can properly locate their message catalogs.

Graphical user interfaces with Tk

Tk/Tcl has long been an integral part of Python. It provides a robust and platform independent windowing toolkit, that is available to Python programmers using the `tkinter` package, and its extension, the `tkinter.ttk` module.

The `tkinter` package is a thin object-oriented layer on top of Tcl/Tk. To use `tkinter`, you don’t need to write Tcl code, but you will need to consult the Tk documentation, and occasionally the Tcl documentation. `tkinter` is a set of wrappers that implement the Tk widgets as Python classes.

`tkinter`’s chief virtues are that it is fast, and that it usually comes bundled with Python. Although its standard documentation is weak, good material is available, which includes: references, tutorials, a book and others. `tkinter` is also famous for having an outdated look and feel, which has been vastly improved in Tk 8.5. Nevertheless, there are many other GUI libraries that you could be interested in. The Python wiki lists several alternative GUI frameworks and tools [https://wiki.python.org/moin/GuiProgramming].

- `tkinter` — Python interface to Tcl/Tk
  - Architecture
  - Tkinter Modules
  - Tkinter Life Preserver
    - A Hello World Program
    - Important Tk Concepts
    - Understanding How Tkinter Wraps Tcl/Tk
    - How do I…? What option does…?
    - Navigating the Tcl/Tk Reference Manual

    - Threading model
    - Handy Reference
        - Setting Options
        - The Packer
        - Packer Options
        - Coupling Widget Variables
        - The Window Manager
        - Tk Option Data Types
        - Bindings and Events
        - The index Parameter
        - Images
    - File Handlers
- **`tkinter.colorchooser`** — Color choosing dialog
- **`tkinter.font`** — Tkinter font wrapper
- Tkinter Dialogs
    - **`tkinter.simpledialog`** — Standard Tkinter input dialogs
    - **`tkinter.filedialog`** — File selection dialogs
        - Native Load/Save Dialogs
    - **`tkinter.commondialog`** — Dialog window templates
- **`tkinter.messagebox`** — Tkinter message prompts
- **`tkinter.scrolledtext`** — Scrolled Text Widget
- **`tkinter.dnd`** — Drag and drop support
- **`tkinter.ttk`** — Tk themed widgets
    - Using Ttk
    - Ttk Widgets
    - Widget
        - Standard Options
        - Scrollable Widget Options
        - Label Options
        - Compatibility Options
        - Widget States
        - ttk.Widget
    - Combobox
        - Options
        - Virtual events
        - ttk.Combobox

  - Spinbox
    - Options
    - Virtual events
    - ttk.Spinbox
  - Notebook
    - Options
    - Tab Options
    - Tab Identifiers
    - Virtual Events
    - ttk.Notebook
  - Progressbar
    - Options
    - ttk.Progressbar
  - Separator
    - Options
  - Sizegrip
    - Platform-specific notes
    - Bugs
  - Treeview
    - Options
    - Item Options
    - Tag Options
    - Column Identifiers
    - Virtual Events
    - ttk.Treeview
  - Ttk Styling
    - Layouts
- IDLE — Python editor and shell
  - Menus
    - File menu (Shell and Editor)
    - Edit menu (Shell and Editor)
    - Format menu (Editor window only)
    - Run menu (Editor window only)
    - Shell menu (Shell window only)
    - Debug menu (Shell window only)
    - Options menu (Shell and Editor)

        - Window menu (Shell and Editor)
        - Help menu (Shell and Editor)
        - Context menus
    - Editing and Navigation
        - Editor windows
        - Key bindings
        - Automatic indentation
        - Search and Replace
        - Completions
        - Calltips
        - Code Context
        - Shell window
        - Text colors
    - Startup and Code Execution
        - Command-line usage
        - Startup failure
        - Running user code
        - User output in Shell
        - Developing tkinter applications
        - Running without a subprocess
    - Help and Preferences
        - Help sources
        - Setting preferences
        - IDLE on macOS
        - Extensions
    - idlelib — implementation of IDLE application
- `turtle` — Turtle graphics
    - Introduction
    - Get started
    - Tutorial
        - Starting a turtle environment
        - Basic drawing
            - Pen control
            - The turtle’s position
        - Making algorithmic patterns
    - How to…

    - Get started as quickly as possible
    - Automatically begin and end filling
    - Use the **`turtle`** module namespace
    - Use turtle graphics in a script
    - Use object-oriented turtle graphics
- Turtle graphics reference
    - Turtle methods
    - Methods of TurtleScreen/Screen
- Methods of RawTurtle/Turtle and corresponding functions
    - Turtle motion
    - Tell Turtle’s state
    - Settings for measurement
    - Pen control
        - Drawing state
        - Color control
        - Filling
        - More drawing control
    - Turtle state
        - Visibility
        - Appearance
    - Using events
    - Special Turtle methods
    - Compound shapes
- Methods of TurtleScreen/Screen and corresponding functions
    - Window control
    - Animation control
    - Using screen events
    - Input methods
    - Settings and special methods
    - Methods specific to Screen, not inherited from TurtleScreen
- Public classes
- Explanation
- Help and configuration

    - How to use help
    - Translation of docstrings into different languages
    - How to configure Screen and Turtles
- `turtledemo` — Demo scripts
- Changes since Python 2.6
- Changes since Python 3.0

`tkinter` — Python interface to Tcl/Tk

Source code: Lib/tkinter/__init__.py
[https://github.com/python/cpython/tree/3.14/Lib/tkinter/__init__.py]

The **`tkinter`** package (“Tk interface”) is the standard Python interface to the Tcl/Tk GUI toolkit. Both Tk and **`tkinter`** are available on most Unix platforms, including macOS, as well as on Windows systems.

Running `python -m tkinter` from the command line should open a window demonstrating a simple Tk interface, letting you know that **`tkinter`** is properly installed on your system, and also showing what version of Tcl/Tk is installed, so you can read the Tcl/Tk documentation specific to that version.

Tkinter supports a range of Tcl/Tk versions, built either with or without thread support. The official Python binary release bundles Tcl/Tk 8.6 threaded. See the source code for the **`_tkinter`** module for more information about supported versions.

Tkinter is not a thin wrapper, but adds a fair amount of its own logic to make the experience more pythonic. This documentation will concentrate on these additions and changes, and refer to the official Tcl/Tk documentation for details that are unchanged.

Note

Tcl/Tk 8.5 (2007) introduced a modern set of themed user interface components along with a new API to use them. Both old and new APIs are still available. Most documentation you will find online still uses the old API and can be woefully outdated.

This is an optional module. If it is missing from your copy of CPython, look for documentation from your distributor (that is, whoever provided Python to you). If you are the distributor, see Requirements for optional modules.

See also

- **TkDocs** **[https://tkdocs.com/]**
  Extensive tutorial on creating user interfaces with Tkinter. Explains key concepts, and illustrates recommended approaches using the modern API.
- **Tkinter 8.5 reference: a GUI for Python** **[https://www.tkdocs.com/shipman/]**
  Reference documentation for Tkinter 8.5 detailing available classes, methods, and options.

Tcl/Tk Resources:

- **Tk commands** **[https://www.tcl.tk/man/tcl8.6/TkCmd/contents.htm]**
  Comprehensive reference to each of the underlying Tcl/Tk commands used by Tkinter.
- **Tcl/Tk Home Page** **[https://www.tcl.tk]**
  Additional documentation, and links to Tcl/Tk core development.

Books:

- **Modern Tkinter for Busy Python Developers** **[https://tkdocs.com/book.html]**

  By Mark Roseman. (ISBN 978-1999149567)
- **Python GUI programming with Tkinter** **[https://www.packtpub.com/en-us/product/python-gui-programming-with-tkinter-9781788835886]**

  By Alan D. Moore. (ISBN 978-1788835886)
- **Programming Python** **[https://learning-python.com/about-pp4e.html]**

  By Mark Lutz; has excellent coverage of Tkinter. (ISBN 978-0596158101)
- **Tcl and the Tk Toolkit (2nd edition)** **[https://www.amazon.com/exec/obidos/ASIN/032133633X]**

  By John Ousterhout, inventor of Tcl/Tk, and Ken Jones; does not cover Tkinter. (ISBN 978-0321336330)

Architecture

Tcl/Tk is not a single library but rather consists of a few distinct modules, each with separate functionality and its own official documentation. Python’s binary releases also ship an add-on module together with it.

Tcl

Tcl is a dynamic interpreted programming language, just like Python. Though it can be used on its own as a general-purpose programming language, it is most commonly embedded into C applications as a scripting engine or an interface to the Tk toolkit. The Tcl library has a C interface to create and manage one or more instances of a Tcl interpreter, run Tcl commands and scripts in those instances, and add custom commands implemented in either Tcl or C. Each interpreter has an event queue, and there are facilities to send events to it and process them. Unlike Python, Tcl's execution model is designed around cooperative multitasking, and Tkinter bridges this difference (see Threading model for details).

Tk

Tk is a Tcl package [https://wiki.tcl-lang.org/37432] implemented in C that adds custom commands to create and manipulate GUI widgets. Each `Tk` object embeds its own Tcl interpreter instance with Tk loaded into it. Tk's widgets are very customizable, though at the cost of a dated appearance. Tk uses Tcl's event queue to generate and process GUI events.

Ttk

Themed Tk (Ttk) is a newer family of Tk widgets that provide a much better appearance on different platforms than many of the classic Tk widgets. Ttk is distributed as part of Tk, starting with Tk version 8.5. Python bindings are provided in a separate module, `tkinter.ttk`.

Internally, Tk and Ttk use facilities of the underlying operating system, i.e., Xlib on Unix/X11, Cocoa on macOS, GDI on Windows.

When your Python application uses a class in Tkinter, e.g., to create a widget, the `tkinter` module first assembles a Tcl/Tk command string. It passes that Tcl command string to an internal `_tkinter` binary module, which then calls the Tcl interpreter to evaluate it. The Tcl interpreter will then call into the Tk and/or Ttk packages, which will in turn make calls to Xlib, Cocoa, or GDI.

Tkinter Modules

Support for Tkinter is spread across several modules. Most applications will need the main `tkinter` module, as well as the **`tkinter.ttk`** module, which provides the modern themed widget set and API:

```
from tkinter import *
from tkinter import ttk
```

class tkinter.Tk(*screenName=None*, *baseName=None*, *className='Tk'*, *useTk=True*, *sync=False*, *use=None*)

Construct a toplevel Tk widget, which is usually the main window of an application, and initialize a Tcl interpreter for this widget. Each instance has its own associated Tcl interpreter.

The **`Tk`** class is typically instantiated using all default values. However, the following keyword arguments are currently recognized:

screenName

When given (as a string), sets the **`DISPLAY`** environment variable. (X11 only)

baseName

Name of the profile file. By default, *baseName* is derived from the program name (`sys.argv[0]`).

className

Name of the widget class. Used as a profile file and also as the name with which Tcl is invoked (*argv0* in *interp*).

useTk

If `True`, initialize the Tk subsystem. The **`tkinter.Tcl()`** function sets this to `False`.

sync

If `True`, execute all X server commands synchronously, so that errors are reported immediately. Can be used for debugging. (X11 only)

use

Specifies the *id* of the window in which to embed the application, instead of it being created as an independent toplevel window. *id* must be specified in the same way as the value for the -use option for toplevel widgets (that is, it has a form like that returned by **`winfo_id()`**).

Note that on some platforms this will only work correctly if *id* refers to a Tk frame or toplevel that has its -container option enabled.

`Tk` reads and interprets profile files, named `.*className*.tcl` and `.*baseName*.tcl`, into the Tcl interpreter and calls **`exec()`** on the contents of `.*className*.py` and `.*baseName*.py`. The path for the profile files is the **`HOME`** environment variable or, if that isn't defined, then **`os.curdir`**.

tk

The Tk application object created by instantiating **`Tk`**. This provides access to the Tcl interpreter. Each widget that is attached the same instance of **`Tk`** has the same value for its **`tk`** attribute.

master

The widget object that contains this widget. For **`Tk`**, the *master* is **`None`** because it is the main window. The terms *master* and *parent* are similar and sometimes used interchangeably as argument names; however, calling **`winfo_parent()`** returns a string of the widget name whereas **`master`** returns the object. *parent*/*child* reflects the tree-like relationship while *master*/*slave* reflects the container structure.

children

The immediate descendants of this widget as a `dict` with the child widget names as the keys and the child instance objects as the values.

tkinter.Tcl(*screenName=None*, *baseName=None*, *className='Tk'*, *useTk=False*)

The `Tcl()` function is a factory function which creates an object much like that created by the `Tk` class, except that it does not initialize the Tk subsystem. This is most often useful when driving the Tcl interpreter in an environment where one doesn’t want to create extraneous toplevel windows, or where one cannot (such as Unix/Linux systems without an X server). An object created by the `Tcl()` object can have a Toplevel window created (and the Tk subsystem initialized) by calling its **`loadtk()`** method.

The modules that provide Tk support include:

`tkinter`

Main Tkinter module.

`tkinter.colorchooser`

Dialog to let the user choose a color.

`tkinter.commondialog`

Base class for the dialogs defined in the other modules listed here.

`tkinter.filedialog`

Common dialogs to allow the user to specify a file to open or save.

`tkinter.font`

Utilities to help work with fonts.

`tkinter.messagebox`
Access to standard Tk dialog boxes.

`tkinter.scrolledtext`
Text widget with a vertical scroll bar built in.

`tkinter.simpledialog`
Basic dialogs and convenience functions.

`tkinter.ttk`
Themed widget set introduced in Tk 8.5, providing modern alternatives for many of the classic widgets in the main `tkinter` module.

Additional modules:

`_tkinter`
A binary module that contains the low-level interface to Tcl/Tk. It is automatically imported by the main `tkinter` module, and should never be used directly by application programmers. It is usually a shared library (or DLL), but might in some cases be statically linked with the Python interpreter.

`idlelib`
Python’s Integrated Development and Learning Environment (IDLE). Based on `tkinter`.

`tkinter.constants`
Symbolic constants that can be used in place of strings when passing various parameters to Tkinter calls. Automatically imported by the main `tkinter` module.

`tkinter.dnd`

(experimental) Drag-and-drop support for `tkinter`. This will become deprecated when it is replaced with the Tk DND.

`turtle`
Turtle graphics in a Tk window.

Tkinter Life Preserver

This section is not designed to be an exhaustive tutorial on either Tk or Tkinter. For that, refer to one of the external resources noted earlier. Instead, this section provides a very quick orientation to what a Tkinter application looks like, identifies foundational Tk concepts, and explains how the Tkinter wrapper is structured.

The remainder of this section will help you to identify the classes, methods, and options you'll need in your Tkinter application, and where to find more detailed documentation on them, including in the official Tcl/Tk reference manual.

A Hello World Program

We'll start by walking through a “Hello World” application in Tkinter. This isn't the smallest one we could write, but has enough to illustrate some key concepts you'll need to know.

```
from tkinter import *
from tkinter import ttk
root = Tk()
frm = ttk.Frame(root, padding=10)
frm.grid()
ttk.Label(frm, text="Hello World!").grid(column=0, row=0)
ttk.Button(frm, text="Quit",
command=root.destroy).grid(column=1, row=0)
root.mainloop()
```

After the imports, the next line creates an instance of the `Tk` class, which initializes Tk and creates its associated Tcl interpreter. It also creates a toplevel window, known as the root window, which serves as the main window of the application.

The following line creates a frame widget, which in this case will contain a label and a button we'll create next. The frame is fit inside the root window.

The next line creates a label widget holding a static text string. The `grid()` method is used to specify the relative layout (position) of the label within its containing frame widget, similar to how tables in HTML work.

A button widget is then created, and placed to the right of the label. When pressed, it will call the `destroy()` method of the root window.

Finally, the `mainloop()` method puts everything on the display, and responds to user input until the program terminates.

Important Tk Concepts

Even this simple program illustrates the following key Tk concepts:

widgets

A Tkinter user interface is made up of individual *widgets*. Each widget is represented as a Python object, instantiated from classes like `ttk.Frame`, `ttk.Label`, and `ttk.Button`.

widget hierarchy

Widgets are arranged in a *hierarchy*. The label and button were contained within a frame, which in turn was contained within the root window. When creating each *child* widget, its *parent* widget is passed as the first argument to the widget constructor.

configuration options

Widgets have *configuration options*, which modify their appearance and behavior, such as the text to display in a label or button. Different classes of widgets will have different sets of options.

geometry management

Widgets aren't automatically added to the user interface when they are created. A *geometry manager* like `grid` controls where in the user interface they are placed.

event loop

Tkinter reacts to user input, changes from your program, and even refreshes the display only when actively running an *event loop*. If your program isn't running the event loop, your user interface won't update.

Understanding How Tkinter Wraps Tcl/Tk

When your application uses Tkinter's classes and methods, internally Tkinter is assembling strings representing Tcl/Tk commands, and executing those commands in the Tcl interpreter attached to your application's **Tk** instance.

Whether it's trying to navigate reference documentation, trying to find the right method or option, adapting some existing code, or debugging your Tkinter application, there are times that it will be useful to understand what those underlying Tcl/Tk commands look like.

To illustrate, here is the Tcl/Tk equivalent of the main part of the Tkinter script above.

```
ttk::frame .frm -padding 10
grid .frm
grid [ttk::label .frm.lbl -text "Hello World!"] -column 0 -row
0
grid [ttk::button .frm.btn -text "Quit" -command "destroy ."] -
column 1 -row 0
```

Tcl’s syntax is similar to many shell languages, where the first word is the command to be executed, with arguments to that command following it, separated by spaces. Without getting into too many details, notice the following:

- The commands used to create widgets (like `ttk::frame`) correspond to widget classes in Tkinter.
- Tcl widget options (like `-text`) correspond to keyword arguments in Tkinter.
- Widgets are referred to by a *pathname* in Tcl (like `.frm.btn`), whereas Tkinter doesn’t use names but object references.
- A widget’s place in the widget hierarchy is encoded in its (hierarchical) pathname, which uses a `.` (dot) as a path separator. The pathname for the root window is just `.` (dot). In Tkinter, the hierarchy is defined not by pathname but by specifying the parent widget when creating each child widget.
- Operations which are implemented as separate *commands* in Tcl (like `grid` or `destroy`) are represented as *methods* on Tkinter widget objects. As you’ll see shortly, at other times Tcl uses what appear to be method calls on widget objects, which more closely mirror what is used in Tkinter.

How do I…? What option does…?

If you’re not sure how to do something in Tkinter, and you can’t immediately find it in the tutorial or reference documentation you’re using, there are a few strategies that can be helpful.

First, remember that the details of how individual widgets work may vary across different versions of both Tkinter and Tcl/Tk. If you’re searching documentation, make sure it corresponds to the Python and Tcl/Tk versions installed on your system.

When searching for how to use an API, it helps to know the exact name of the class, option, or method that you’re using. Introspection, either in an interactive Python shell or with `print()`, can help you identify what you need.

To find out what configuration options are available on any widget, call its **`configure()`** method, which returns a dictionary containing a variety of information about each object, including its default and current values. Use **`keys()`** to get just the names of each option.

```
btn = ttk.Button(frm, ...)
print(btn.configure().keys())
```

As most widgets have many configuration options in common, it can be useful to find out which are specific to a particular widget class. Comparing the list of options to that of a simpler widget, like a frame, is one way to do that.

```
print(set(btn.configure().keys()) -
set(frm.configure().keys()))
```

Similarly, you can find the available methods for a widget object using the standard `dir()` function. If you try it, you’ll see there are over 200 common widget methods, so again identifying those specific to a widget class is helpful.

```
print(dir(btn))
print(set(dir(btn)) - set(dir(frm)))
```

Navigating the Tcl/Tk Reference Manual

As noted, the official Tk commands [https://www.tcl.tk/man/tcl8.6/TkCmd/contents.htm] reference manual (man pages) is often the most accurate description of what specific operations on widgets do. Even when you know the name of the option or method that you need, you may still have a few places to look.

While all operations in Tkinter are implemented as method calls on widget objects, you've seen that many Tcl/Tk operations appear as commands that take a widget pathname as its first parameter, followed by optional parameters, e.g.

```
destroy .
grid .frm.btn -column 0 -row 0
```

Others, however, look more like methods called on a widget object (in fact, when you create a widget in Tcl/Tk, it creates a Tcl command with the name of the widget pathname, with the first parameter to that command being the name of a method to call).

```
.frm.btn invoke
.frm.lbl configure -text "Goodbye"
```

In the official Tcl/Tk reference documentation, you'll find most operations that look like method calls on the man page for a specific widget (e.g., you'll find the **`invoke()`** method on the ttk::button [https://www.tcl.tk/man/tcl8.6/TkCmd/ttk_button.htm] man page), while functions that take a widget as a parameter often have their own man page (e.g., grid [https://www.tcl.tk/man/tcl8.6/TkCmd/grid.htm]).

You'll find many common options and methods in the options [https://www.tcl.tk/man/tcl8.6/TkCmd/options.htm] or ttk::widget [https://www.tcl.tk/man/tcl8.6/TkCmd/ttk_widget.htm] man pages, while others are found in the man page for a specific widget class.

You’ll also find that many Tkinter methods have compound names, e.g., **`winfo_x()`**, **`winfo_height()`**, **`winfo_viewable()`**. You’d find documentation for all of these in the winfo [https://www.tcl.tk/man/tcl8.6/TkCmd/winfo.htm] man page.

Note

Somewhat confusingly, there are also methods on all Tkinter widgets that don’t actually operate on the widget, but operate at a global scope, independent of any widget. Examples are methods for accessing the clipboard or the system bell. (They happen to be implemented as methods in the base **`Widget`** class that all Tkinter widgets inherit from).

Threading model

Python and Tcl/Tk have very different threading models, which **`tkinter`** tries to bridge. If you use threads, you may need to be aware of this.

A Python interpreter may have many threads associated with it. In Tcl, multiple threads can be created, but each thread has a separate Tcl interpreter instance associated with it. Threads can also create more than one interpreter instance, though each interpreter instance can be used only by the one thread that created it.

Each **`Tk`** object created by **`tkinter`** contains a Tcl interpreter. It also keeps track of which thread created that interpreter. Calls to **`tkinter`** can be made from any Python thread. Internally, if a call comes from a thread other than the one that created the `Tk` object, an event is posted to the interpreter's event queue, and when executed, the result is returned to the calling Python thread.

Tcl/Tk applications are normally event-driven, meaning that after initialization, the interpreter runs an event loop (i.e. `Tk.mainloop()`) and responds to events. Because it is single-threaded, event handlers must respond quickly, otherwise they will block other events from being processed. To avoid this, any long-running computations should not run in an event handler, but are either broken into smaller pieces using timers, or run in another thread. This is different from many GUI toolkits where the GUI runs in a completely separate thread from all application code including event handlers.

If the Tcl interpreter is not running the event loop and processing events, any `tkinter` calls made from threads other than the one running the Tcl interpreter will fail.

A number of special cases exist:

- Tcl/Tk libraries can be built so they are not thread-aware. In this case, `tkinter` calls the library from the originating Python thread, even if this is different than the thread that created the Tcl interpreter. A global lock ensures only one call occurs at a time.
- While `tkinter` allows you to create more than one instance of a `Tk` object (with its own interpreter), all interpreters that are part of the same thread share a common event queue, which gets ugly fast. In practice, don't create more than one instance of `Tk` at a time. Otherwise, it's best to create them in separate threads and ensure you're running a thread-aware Tcl/Tk build.
- Blocking event handlers are not the only way to prevent the Tcl interpreter from reentering the event loop. It is

even possible to run multiple nested event loops or abandon the event loop entirely. If you’re doing anything tricky when it comes to events or threads, be aware of these possibilities.
- There are a few select `tkinter` functions that presently work only when called from the thread that created the Tcl interpreter.

Handy Reference

Setting Options

Options control things like the color and border width of a widget. Options can be set in three ways:

At object creation time, using keyword arguments

```
fred = Button(self, fg="red", bg="blue")
```

After object creation, treating the option name like a dictionary index

```
fred["fg"] = "red"
fred["bg"] = "blue"
```

Use the config() method to update multiple attrs subsequent to object creation

```
fred.config(fg="red", bg="blue")
```

For a complete explanation of a given option and its behavior, see the Tk man pages for the widget in question.

Note that the man pages list “STANDARD OPTIONS” and “WIDGET SPECIFIC OPTIONS” for each widget. The former is a list of options that are common to many widgets, the latter are the options that are idiosyncratic to that particular widget. The Standard Options are documented on the *options(3)* *[https://manpages.debian.org/options(3)]* man page.

No distinction between standard and widget-specific options is made in this document. Some options don’t apply to some kinds of widgets. Whether a given widget responds to a particular option depends on the class of the widget; buttons have a `command` option, labels do not.

The options supported by a given widget are listed in that widget’s man page, or can be queried at runtime by calling the **`config()`** method without arguments, or by calling the **`keys()`** method on that widget. The return value of these calls is a dictionary whose key is the name of the option as a string (for example, `'relief'`) and whose values are 5-tuples.

Some options, like `bg` are synonyms for common options with long names (`bg` is shorthand for “background”). Passing the `config()` method the name of a shorthand option will return a 2-tuple, not 5-tuple. The 2-tuple passed back will contain the name of the synonym and the “real” option (such as `('bg', 'background')`).

Index	Meaning	Example
0	option name	`'relief'`
1	option name for database lookup	`'relief'`
2	option class for database lookup	`'Relief'`
3	default value	`'raised'`

Index	Meaning	Example
4	current value	`'groove'`

Example:

```
>>> print(fred.config())
{'relief': ('relief', 'relief', 'Relief', 'raised', 'groove')}
```

Of course, the dictionary printed will include all the options available and their values. This is meant only as an example.

The Packer

The packer is one of Tk’s geometry-management mechanisms. Geometry managers are used to specify the relative positioning of widgets within their container - their mutual *master*. In contrast to the more cumbersome *placer* (which is used less commonly, and we do not cover here), the packer takes qualitative relationship specification - *above*, *to the left of*, *filling*, etc - and works everything out to determine the exact placement coordinates for you.

The size of any *master* widget is determined by the size of the “slave widgets” inside. The packer is used to control where slave widgets appear inside the master into which they are packed. You can pack widgets into frames, and frames into other frames, in order to achieve the kind of layout you desire. Additionally, the arrangement is dynamically adjusted to accommodate incremental changes to the configuration, once it is packed.

Note that widgets do not appear until they have had their geometry specified with a geometry manager. It's a common early mistake to leave out the geometry specification, and then be surprised when the widget is created but nothing appears. A widget will appear only after it has had, for example, the packer's **`pack()`** method applied to it.

The pack() method can be called with keyword-option/value pairs that control where the widget is to appear within its container, and how it is to behave when the main application window is resized. Here are some examples:

```
fred.pack()                     # defaults to side = "top"
fred.pack(side="left")
fred.pack(expand=1)
```

Packer Options

For more extensive information on the packer and the options that it can take, see the man pages and page 183 of John Ousterhout's book.

anchor
: Anchor type. Denotes where the packer is to place each slave in its parcel.

expand
: Boolean, `0` or `1`.

fill
: Legal values: `'x'`, `'y'`, `'both'`, `'none'`.

ipadx and ipady
: A distance - designating internal padding on each side of the slave widget.

padx and pady
: A distance - designating external padding on each side of the slave widget.

side
: Legal values are: `'left'`, `'right'`, `'top'`, `'bottom'`.

Coupling Widget Variables

The current-value setting of some widgets (like text entry widgets) can be connected directly to application variables by using special options. These options are `variable`, `textvariable`, `onvalue`, `offvalue`, and `value`. This connection works both ways: if the variable changes for any reason, the widget it’s connected to will be updated to reflect the new value.

Unfortunately, in the current implementation of **`tkinter`** it is not possible to hand over an arbitrary Python variable to a widget through a `variable` or `textvariable` option. The only kinds of variables for which this works are variables that are subclassed from a class called Variable, defined in **`tkinter`**.

There are many useful subclasses of Variable already defined: **`StringVar`**, **`IntVar`**, **`DoubleVar`**, and **`BooleanVar`**. To read the current value of such a variable, call the **`get()`** method on it, and to change its value you call the **`set()`** method. If you follow this protocol, the widget will always track the value of the variable, with no further intervention on your part.

For example:

```
import tkinter as tk

class App(tk.Frame):
    def __init__(self, master):
        super().__init__(master)
        self.pack()

        self.entrythingy = tk.Entry()
        self.entrythingy.pack()

        # Create the application variable.
        self.contents = tk.StringVar()
        # Set it to some value.
        self.contents.set("this is a variable")
        # Tell the entry widget to watch this variable.
        self.entrythingy["textvariable"] = self.contents

        # Define a callback for when the user hits return.
        # It prints the current value of the variable.
        self.entrythingy.bind('<Key-Return>',
                              self.print_contents)

    def print_contents(self, event):
        print("Hi. The current entry content is:",
              self.contents.get())

root = tk.Tk()
myapp = App(root)
myapp.mainloop()
```

The Window Manager

In Tk, there is a utility command, `wm`, for interacting with the window manager. Options to the `wm` command allow you to control things like titles, placement, icon bitmaps, and the like. In `tkinter`, these commands have been implemented as methods on the **`Wm`** class. Toplevel widgets are subclassed from the **`Wm`** class, and so can call the **`Wm`** methods directly.

To get at the toplevel window that contains a given widget, you can often just refer to the widget’s master. Of course if the widget has been packed inside of a frame, the master won’t represent a toplevel window. To get at the toplevel window that contains an arbitrary widget, you can call the

`_root()` method. This method begins with an underscore to denote the fact that this function is part of the implementation, and not an interface to Tk functionality.

Here are some examples of typical usage:

```
import tkinter as tk

class App(tk.Frame):
    def __init__(self, master=None):
        super().__init__(master)
        self.pack()

# create the application
myapp = App()

#
# here are method calls to the window manager class
#
myapp.master.title("My Do-Nothing Application")
myapp.master.maxsize(1000, 400)

# start the program
myapp.mainloop()
```

Tk Option Data Types

anchor

Legal values are points of the compass: `"n"`, `"ne"`, `"e"`, `"se"`, `"s"`, `"sw"`, `"w"`, `"nw"`, and also `"center"`.

bitmap

There are eight built-in, named bitmaps: `'error'`, `'gray25'`, `'gray50'`, `'hourglass'`, `'info'`, `'questhead'`, `'question'`, `'warning'`. To specify an X bitmap filename, give the full path to the file, preceded with an `@`, as in `"@/usr/contrib/bitmap/gumby.bit"`.

boolean

You can pass integers 0 or 1 or the strings `"yes"` or `"no"`.

callback

This is any Python function that takes no arguments. For example:

```
def print_it():
    print("hi there")
fred["command"] = print_it
```

color

Colors can be given as the names of X colors in the rgb.txt file, or as strings representing RGB values in 4 bit: `"#RGB"`, 8 bit: `"#RRGGBB"`, 12 bit: `"#RRRGGGBBB"`, or 16 bit: `"#RRRRGGGGBBBB"` ranges, where R,G,B here represent any legal hex digit. See page 160 of Ousterhout’s book for details.

cursor

The standard X cursor names from `cursorfont.h` can be used, without the `XC_` prefix. For example to get a hand cursor (**`XC_hand2`**), use the string `"hand2"`. You can also specify a bitmap and mask file of your own. See page 179 of Ousterhout’s book.

distance

Screen distances can be specified in either pixels or absolute distances. Pixels are given as numbers and absolute distances as strings, with the trailing character denoting units: `c` for centimetres, `i` for inches, `m` for millimetres, `p` for printer’s points. For example, 3.5 inches is expressed as `"3.5i"`.

font

Tk uses a list font name format, such as `{courier 10 bold}`. Font sizes with positive numbers are measured in points; sizes with negative numbers are measured in pixels.

geometry

This is a string of the form `widthxheight`, where width and height are measured in pixels for most widgets (in characters for widgets displaying text). For example: `fred["geometry"] = "200x100"`.

justify

Legal values are the strings: `"left"`, `"center"`, and `"right"`.

region

This is a string with four space-delimited elements, each of which is a legal distance (see above). For example: `"2 3 4 5"` and `"3i 2i 4.5i 2i"` and `"3c 2c 4c 10.43c"` are all legal regions.

relief

Determines what the border style of a widget will be. Legal values are: `"raised"`, `"sunken"`, `"flat"`, `"groove"`, and `"ridge"`.

scrollcommand

This is almost always the **`set()`** method of some scrollbar widget, but can be any widget method that takes a single argument.

wrap

Must be one of: `"none"`, `"char"`, or `"word"`.

Bindings and Events

The bind method from the widget command allows you to watch for certain events and to have a callback function trigger when that event type occurs. The form of the bind method is:

```
def bind(self, sequence, func, add=''):
```

where:

sequence
: is a string that denotes the target kind of event. (See the bind(3tk) *[https://manpages.debian.org/bind(3tk)]* man page, and page 201 of John Ousterhout’s book, *Tcl and the Tk Toolkit (2nd edition)*, for details).

func
: is a Python function, taking one argument, to be invoked when the event occurs. An Event instance will be passed as the argument. (Functions deployed this way are commonly known as *callbacks*.)

add
: is optional, either `''` or `'+'`. Passing an empty string denotes that this binding is to replace any other bindings that this event is associated with. Passing a `'+'` means that this function is to be added to the list of functions bound to this event type.

For example:

```
def turn_red(self, event):
    event.widget["activeforeground"] = "red"

self.button.bind("<Enter>", self.turn_red)
```

Notice how the widget field of the event is being accessed in the `turn_red()` callback. This field contains the widget that caught the X event. The following table lists the other event fields you can access, and how they are denoted in Tk, which can be useful when referring to the Tk man pages.

Tk	Tkinter Event Field	Tk	Tkinter Event Field
%f	focus	%A	char
%h	height	%E	send_event
%k	keycode	%K	keysym
%s	state	%N	keysym_num
%t	time	%T	type
%w	width	%W	widget
%x	x	%X	x_root
%y	y	%Y	y_root

The index Parameter

A number of widgets require “index” parameters to be passed. These are used to point at a specific place in a Text widget, or to particular characters in an Entry widget, or to particular menu items in a Menu widget.

Entry widget indexes (index, view index, etc.)

Entry widgets have options that refer to character positions in the text being displayed. You can use these `tkinter` functions to access these special points in text widgets:

Text widget indexes

The index notation for Text widgets is very rich and is best described in the Tk man pages.

Menu indexes (menu.invoke(), menu.entryconfig(), etc.)

Some options and methods for menus manipulate specific menu entries. Anytime a menu index is needed for an option or a parameter, you may pass in:

- an integer which refers to the numeric position of the entry in the widget, counted from the top, starting with 0;
- the string `"active"`, which refers to the menu position that is currently under the cursor;
- the string `"last"` which refers to the last menu item;
- An integer preceded by `@`, as in `@6`, where the integer is interpreted as a y pixel coordinate in the menu’s coordinate system;
- the string `"none"`, which indicates no menu entry at all, most often used with menu.activate() to deactivate all entries, and finally,
- a text string that is pattern matched against the label of the menu entry, as scanned from the top of the menu to the bottom. Note that this index type is considered after all the others, which means that matches for menu items labelled `last`, `active`, or `none` may be interpreted as the above literals, instead.

Images

Images of different formats can be created through the corresponding subclass of **`tkinter.Image`**:

- **`BitmapImage`** for images in XBM format.

- **`PhotoImage`** for images in PGM, PPM, GIF and PNG formats. The latter is supported starting with Tk 8.6.

Either type of image is created through either the `file` or the `data` option (other options are available as well).

Changed in version 3.13: Added the **`PhotoImage`** method **`copy_replace()`** to copy a region from one image to other image, possibly with pixel zooming and/or subsampling. Add *from_coords* parameter to **`PhotoImage`** methods **`copy()`**, **`zoom()`** and **`subsample()`**. Add *zoom* and *subsample* parameters to **`PhotoImage`** method **`copy()`**.

The image object can then be used wherever an `image` option is supported by some widget (e.g. labels, buttons, menus). In these cases, Tk will not keep a reference to the image. When the last Python reference to the image object is deleted, the image data is deleted as well, and Tk will display an empty box wherever the image was used.

See also

The Pillow [https://python-pillow.org/] package adds support for formats such as BMP, JPEG, TIFF, and WebP, among others.

File Handlers

Tk allows you to register and unregister a callback function which will be called from the Tk mainloop when I/O is possible on a file descriptor. Only one handler may be registered per file descriptor. Example code:

```
import tkinter
widget = tkinter.Tk()
mask = tkinter.READABLE | tkinter.WRITABLE
```

```
widget.tk.createfilehandler(file, mask, callback)
...
widget.tk.deletefilehandler(file)
```

This feature is not available on Windows.

Since you don’t know how many bytes are available for reading, you may not want to use the `BufferedIOBase` or `TextIOBase` `read()` or `readline()` methods, since these will insist on reading a predefined number of bytes. For sockets, the `recv()` or `recvfrom()` methods will work fine; for other files, use raw reads or `os.read(file.fileno(), maxbytecount)`.

Widget.tk.createfilehandler(*file*, *mask*, *func*)

Registers the file handler callback function *func*. The *file* argument may either be an object with a `fileno()` method (such as a file or socket object), or an integer file descriptor. The *mask* argument is an ORed combination of any of the three constants below. The callback is called as follows:

```
callback(file, mask)
```

Widget.tk.deletefilehandler(*file*)

Unregisters a file handler.

_tkinter.READABLE
_tkinter.WRITABLE
_tkinter.EXCEPTION

Constants used in the *mask* arguments.

`tkinter.colorchooser` — Color choosing dialog

Source code: Lib/tkinter/colorchooser.py
[https://github.com/python/cpython/tree/3.14/Lib/tkinter/colorchooser.py]

The **`tkinter.colorchooser`** module provides the **`Chooser`** class as an interface to the native color picker dialog. `Chooser` implements a modal color choosing dialog window. The `Chooser` class inherits from the **`Dialog`** class.

class tkinter.colorchooser.Chooser(*master=None*, ***options*)

tkinter.colorchooser.askcolor(*color=None*, ***options*)

Create a color choosing dialog. A call to this method will show the window, wait for the user to make a selection, and return the selected color (or `None`) to the caller.

See also

Module **`tkinter.commondialog`**

Tkinter standard dialog module

`tkinter.font` — Tkinter font wrapper

Source code: Lib/tkinter/font.py [https://github.com/python/cpython/tree/3.14/Lib/tkinter/font.py]

The **`tkinter.font`** module provides the **`Font`** class for creating and using named fonts.

The different font weights and slants are:

tkinter.font.NORMAL
tkinter.font.BOLD
tkinter.font.ITALIC
tkinter.font.ROMAN

class tkinter.font.Font(*root=None*, *font=None*, *name=None*, *exists=False*, ***options*)

The **`Font`** class represents a named font. *Font* instances are given unique names and can be specified by their family, size, and style configuration. Named fonts are Tk's method of creating and identifying fonts as a single object, rather than specifying a font by its attributes with each occurrence.

arguments:

font - font specifier tuple (family, size, options)
name - unique font name
exists - self points to existing named font if true

additional keyword options (ignored if *font* is specified):

family - font family i.e. Courier, Times
size - font size
If *size* is positive it is interpreted as size in points.
If *size* is a negative number its absolute value is treated
as size in pixels.
weight - font emphasis (NORMAL, BOLD)
slant - ROMAN, ITALIC
underline - font underlining (0 - none, 1 - underline)
overstrike - font strikeout (0 - none, 1 - strikeout)

actual(*option=None*, *displayof=None*)

Return the attributes of the font.

cget(*option*)

Retrieve an attribute of the font.

config(***options*)

Modify attributes of the font.

copy()

Return new instance of the current font.

measure(*text*, *displayof=None*)

Return amount of space the text would occupy on the specified display when formatted in the current font. If no display is specified then the main application window is assumed.

metrics(**options*, ***kw*)

Return font-specific data. Options include:

ascent - distance between baseline and highest point that a
character of the font can occupy

descent - distance between baseline and lowest point that a
character of the font can occupy

linespace - minimum vertical separation necessary between any two
characters of the font that ensures no vertical overlap between lines.

fixed - 1 if font is fixed-width else 0

tkinter.font.families(*root=None*, *displayof=None*)

Return the different font families.

tkinter.font.names(*root=None*)

Return the names of defined fonts.

tkinter.font.nametofont(*name*, *root=None*)

Return a `Font` representation of a tk named font.

Changed in version 3.10: The *root* parameter was added.

Tkinter Dialogs

`tkinter.simpledialog` — Standard Tkinter input dialogs

Source code: Lib/tkinter/simpledialog.py [https://github.com/python/cpython/tree/3.14/Lib/tkinter/simpledialog.py]

The `tkinter.simpledialog` module contains convenience classes and functions for creating simple modal dialogs to get a value from the user.

tkinter.simpledialog.askfloat(*title*, *prompt*, ***kw*)
tkinter.simpledialog.askinteger(*title*, *prompt*, ***kw*)
tkinter.simpledialog.askstring(*title*, *prompt*, ***kw*)

The above three functions provide dialogs that prompt the user to enter a value of the desired type.

class tkinter.simpledialog.Dialog(*parent*, *title=None*)

The base class for custom dialogs.

body(*master*)

Override to construct the dialog’s interface and return the widget that should have initial focus.

buttonbox()

Default behaviour adds OK and Cancel buttons. Override for custom button layouts.

`tkinter.filedialog` — File selection dialogs

Source code: Lib/tkinter/filedialog.py
[https://github.com/python/cpython/tree/3.14/Lib/tkinter/filedialog.py]

The `tkinter.filedialog` module provides classes and factory functions for creating file/directory selection windows.

Native Load/Save Dialogs

The following classes and functions provide file dialog windows that combine a native look-and-feel with configuration options to customize behaviour. The following keyword arguments are applicable to the classes and functions listed below:

> *parent* - the window to place the dialog on top of
>
> *title* - the title of the window
>
> *initialdir* - the directory that the dialog starts in
>
> *initialfile* - the file selected upon opening of the dialog
>
> *filetypes* - a sequence of (label, pattern) tuples, ‘*’ wildcard is allowed
>
> *defaultextension* - default extension to append to file (save dialogs)
>
> *multiple* - when true, selection of multiple items is allowed

Static factory functions

The below functions when called create a modal, native look-and-feel dialog, wait for the user's selection, then return the selected value(s) or `None` to the caller.

tkinter.filedialog.askopenfile(*mode*='*r*', ***options*)
tkinter.filedialog.askopenfiles(*mode*='*r*', ***options*)

The above two functions create an `Open` dialog and return the opened file object(s) in read-only mode.

tkinter.filedialog.asksaveasfile(*mode*='*w*', ***options*)

Create a `SaveAs` dialog and return a file object opened in write-only mode.

tkinter.filedialog.askopenfilename(***options*)
tkinter.filedialog.askopenfilenames(***options*)

The above two functions create an `Open` dialog and return the selected filename(s) that correspond to existing file(s).

tkinter.filedialog.asksaveasfilename(***options*)

Create a `SaveAs` dialog and return the selected filename.

tkinter.filedialog.askdirectory(***options*)

Prompt user to select a directory.
Additional keyword option:
mustexist - determines if selection must be an existing directory.

class tkinter.filedialog.Open(*master=None*, ***options*)
class tkinter.filedialog.SaveAs(*master=None*, ***options*)

The above two classes provide native dialog windows for saving and loading files.

Convenience classes

The below classes are used for creating file/directory windows from scratch. These do not emulate the native look-and-feel of the platform.

class tkinter.filedialog.Directory(*master=None*, ***options*)

Create a dialog prompting the user to select a directory.

Note

The *FileDialog* class should be subclassed for custom event handling and behaviour.

class tkinter.filedialog.FileDialog(*master*, *title=None*)

Create a basic file selection dialog.

cancel_command(*event=None*)

Trigger the termination of the dialog window.

dirs_double_event(*event*)

Event handler for double-click event on directory.

dirs_select_event(*event*)

Event handler for click event on directory.

files_double_event(*event*)

Event handler for double-click event on file.

files_select_event(*event*)

Event handler for single-click event on file.

filter_command(*event=None*)

Filter the files by directory.

get_filter()

Retrieve the file filter currently in use.

get_selection()

Retrieve the currently selected item.

go(*dir_or_file=os.curdir*, *pattern*='*', *default*='', *key=None*)

Render dialog and start event loop.

ok_event(*event*)

Exit dialog returning current selection.

quit(*how=None*)

Exit dialog returning filename, if any.

set_filter(*dir*, *pat*)

Set the file filter.

set_selection(*file*)

Update the current file selection to *file*.

class tkinter.filedialog.LoadFileDialog(*master*, *title=None*)

A subclass of FileDialog that creates a dialog window for selecting an existing file.

ok_command()

Test that a file is provided and that the selection indicates an already existing file.

class tkinter.filedialog.SaveFileDialog(*master*, *title=None*)

A subclass of FileDialog that creates a dialog window for selecting a destination file.

ok_command()

Test whether or not the selection points to a valid file that is not a directory. Confirmation is required if an already existing file is selected.

`tkinter.commondialog` — Dialog window templates

Source code: Lib/tkinter/commondialog.py [https://github.com/python/cpython/tree/3.14/Lib/tkinter/commondialog.py]

The **`tkinter.commondialog`** module provides the **`Dialog`** class that is the base class for dialogs defined in other supporting modules.

class tkinter.commondialog.Dialog(*master=None*, ***options*)

show(***options*)

Render the Dialog window.

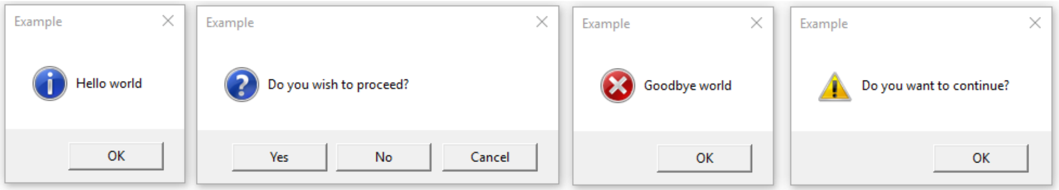
See also

Modules **`tkinter.messagebox`**, Reading and Writing Files

`tkinter.messagebox` — Tkinter message prompts

Source code: Lib/tkinter/messagebox.py
[https://github.com/python/cpython/tree/3.14/Lib/tkinter/messagebox.py]

The **`tkinter.messagebox`** module provides a template base class as well as a variety of convenience methods for commonly used configurations. The message boxes are modal and will return a subset of (`True`, `False`, `None`, **`OK`**, **`CANCEL`**, **`YES`**, **`NO`**) based on the user's selection. Common message box styles and layouts include but are not limited to:


Hello world

Do you wish to proceed?

Goodbye world

Do you want to continue?

class tkinter.messagebox.Message(*master=None*, ***options*)

Create a message window with an application-specified message, an icon and a set of buttons. Each of the buttons in the message window is identified by a unique symbolic name (see the *type* options).

The following options are supported:

command

Specifies the function to invoke when the user closes the dialog. The name of the button clicked by the user to close the dialog is passed as argument. This is only available on macOS.

default
Gives the symbolic name of the default button for this message window (**`OK`**, **`CANCEL`**, and so on). If this option is not specified, the first button in the dialog will be made the default.

detail
Specifies an auxiliary message to the main message given by the *message* option. The message detail will be presented beneath the main message and, where supported by the OS, in a less emphasized font than the main message.

icon
Specifies an icon to display. If this option is not specified, then the **`INFO`** icon will be displayed.

message
Specifies the message to display in this message box. The default value is an empty string.

parent
Makes the specified window the logical parent of the message box. The message box is displayed on top of its parent window.

title
Specifies a string to display as the title of the message box. This option is ignored on macOS, where platform guidelines forbid the use of a title on this kind of dialog.

type
Arranges for a predefined set of buttons to be displayed.

show(***options*)

Display a message window and wait for the user to select one of the buttons. Then return the symbolic name of the selected button. Keyword arguments can override options specified in the constructor.

Information message box

tkinter.messagebox.showinfo(*title=None*, *message=None*, ***options*)

Creates and displays an information message box with the specified title and message.

Warning message boxes

tkinter.messagebox.showwarning(*title=None*, *message=None*, ***options*)

Creates and displays a warning message box with the specified title and message.

tkinter.messagebox.showerror(*title=None*, *message=None*, ***options*)

Creates and displays an error message box with the specified title and message.

Question message boxes

tkinter.messagebox.askquestion(*title=None*, *message=None*, ***, *type=YESNO*, ***options*)

Ask a question. By default shows buttons `YES` and `NO`. Returns the symbolic name of the selected button.

tkinter.messagebox.askokcancel(*title=None*, *message=None*, ***options*)

Ask if operation should proceed. Shows buttons `OK` and `CANCEL`. Returns `True` if the answer is ok and `False` otherwise.

tkinter.messagebox.askretrycancel(*title=None*, *message=None*, ***options*)

Ask if operation should be retried. Shows buttons **`RETRY`** and **`CANCEL`**. Return `True` if the answer is yes and `False` otherwise.

tkinter.messagebox.askyesno(*title=None*, *message=None*, ***options*)

Ask a question. Shows buttons **`YES`** and **`NO`**. Returns `True` if the answer is yes and `False` otherwise.

tkinter.messagebox.askyesnocancel(*title=None*, *message=None*, ***options*)

Ask a question. Shows buttons **`YES`**, **`NO`** and **`CANCEL`**. Return `True` if the answer is yes, `None` if cancelled, and `False` otherwise.

Symbolic names of buttons:

tkinter.messagebox.ABORT = '*abort*'

tkinter.messagebox.RETRY = '*retry*'

tkinter.messagebox.IGNORE = '*ignore*'

tkinter.messagebox.OK = '*ok*'

tkinter.messagebox.CANCEL = '*cancel*'

tkinter.messagebox.YES = '*yes*'

tkinter.messagebox.NO = '*no*'

Predefined sets of buttons:

tkinter.messagebox.ABORTRETRYIGNORE = '*abortretryignore*'

Displays three buttons whose symbolic names are `ABORT`, `RETRY` and `IGNORE`.

tkinter.messagebox.OK = '*ok*'

Displays one button whose symbolic name is `OK`.

tkinter.messagebox.OKCANCEL = '*okcancel*'

Displays two buttons whose symbolic names are `OK` and `CANCEL`.

tkinter.messagebox.RETRYCANCEL = '*retrycancel*'

Displays two buttons whose symbolic names are `RETRY` and `CANCEL`.

tkinter.messagebox.YESNO = '*yesno*'

Displays two buttons whose symbolic names are `YES` and `NO`.

tkinter.messagebox.YESNOCANCEL = '*yesnocancel*'

Displays three buttons whose symbolic names are `YES`, `NO` and `CANCEL`.

Icon images:

tkinter.messagebox.ERROR = '*error*'

tkinter.messagebox.INFO = '*info*'

tkinter.messagebox.QUESTION = '*question*'

tkinter.messagebox.WARNING = '*warning*'

`tkinter.scrolledtext` — Scrolled Text Widget

Source code: Lib/tkinter/scrolledtext.py [https://github.com/python/cpython/tree/3.14/Lib/tkinter/scrolledtext.py]

The `tkinter.scrolledtext` module provides a class of the same name which implements a basic text widget which has a vertical scroll bar configured to do the “right thing.” Using the `ScrolledText` class is a lot easier than setting up a text widget and scroll bar directly.

The text widget and scrollbar are packed together in a `Frame`, and the methods of the `Grid` and `Pack` geometry managers are acquired from the `Frame` object. This allows the `ScrolledText` widget to be used directly to achieve most normal geometry management behavior.

Should more specific control be necessary, the following attributes are available:

class tkinter.scrolledtext.ScrolledText(*master=None*, ***kw*)

frame

The frame which surrounds the text and scroll bar widgets.

vbar

The scroll bar widget.

`tkinter.dnd` — Drag and drop support

Source code: Lib/tkinter/dnd.py [https://github.com/python/cpython/tree/3.14/Lib/tkinter/dnd.py]

Note

This is experimental and due to be deprecated when it is replaced with the Tk DND.

The **`tkinter.dnd`** module provides drag-and-drop support for objects within a single application, within the same window or between windows. To enable an object to be dragged, you must create an event binding for it that starts the drag-and-drop process. Typically, you bind a ButtonPress event to a callback function that you write (see Bindings and Events). The function should call **`dnd_start()`**, where 'source' is the object to be dragged, and 'event' is the event that invoked the call (the argument to your callback function).

Selection of a target object occurs as follows:

1. Top-down search of area under mouse for target widget
   - Target widget should have a callable *dnd_accept* attribute
   - If *dnd_accept* is not present or returns `None`, search moves to parent widget
   - If no target widget is found, then the target object is `None`

2. Call to *<old_target>.dnd_leave(source, event)*
3. Call to *<new_target>.dnd_enter(source, event)*
4. Call to *<target>.dnd_commit(source, event)* to notify of drop
5. Call to *<source>.dnd_end(target, event)* to signal end of drag-and-drop

class tkinter.dnd.DndHandler(*source*, *event*)

The *DndHandler* class handles drag-and-drop events tracking Motion and ButtonRelease events on the root of the event widget.

cancel(*event=None*)

Cancel the drag-and-drop process.

finish(*event*, *commit=0*)

Execute end of drag-and-drop functions.

on_motion(*event*)

Inspect area below mouse for target objects while drag is performed.

on_release(*event*)

Signal end of drag when the release pattern is triggered.

tkinter.dnd.dnd_start(*source*, *event*)

Factory function for drag-and-drop process.

See also

Bindings and Events

`tkinter.ttk` — Tk themed widgets

Source code: Lib/tkinter/ttk.py [https://github.com/python/cpython/tree/3.14/Lib/tkinter/ttk.py]

The **`tkinter.ttk`** module provides access to the Tk themed widget set, introduced in Tk 8.5. It provides additional benefits including anti-aliased font rendering under X11 and window transparency (requiring a composition window manager on X11).

The basic idea for **`tkinter.ttk`** is to separate, to the extent possible, the code implementing a widget’s behavior from the code implementing its appearance.

See also

Tk Widget Styling Support [https://core.tcl.tk/tips/doc/trunk/tip/48.md]
A document introducing theming support for Tk

Using Ttk

To start using Ttk, import its module:

```
from tkinter import ttk
```

To override the basic Tk widgets, the import should follow the Tk import:

```
from tkinter import *
from tkinter.ttk import *
```

That code causes several `tkinter.ttk` widgets (**`Button`**, **`Checkbutton`**, **`Entry`**, **`Frame`**, **`Label`**, **`LabelFrame`**, **`Menubutton`**, **`PanedWindow`**, **`Radiobutton`**, **`Scale`** and **`Scrollbar`**) to automatically replace the Tk widgets.

This has the direct benefit of using the new widgets which gives a better look and feel across platforms; however, the replacement widgets are not completely compatible. The main difference is that widget options such as “fg”, “bg” and others related to widget styling are no longer present in Ttk widgets. Instead, use the **`ttk.Style`** class for improved styling effects.

See also

Converting existing applications to use Tile widgets [https://tktable.sourceforge.net/tile/doc/converting.txt]

A monograph (using Tcl terminology) about differences typically encountered when moving applications to use the new widgets.

Ttk Widgets

Ttk comes with 18 widgets, twelve of which already existed in tkinter: **`Button`**, **`Checkbutton`**, **`Entry`**, **`Frame`**, **`Label`**, **`LabelFrame`**, **`Menubutton`**, **`PanedWindow`**, **`Radiobutton`**, **`Scale`**, **`Scrollbar`**, and `Spinbox`. The other six are new: `Combobox`, `Notebook`, `Progressbar`, **`Separator`**, **`Sizegrip`** and `Treeview`. And all them are subclasses of `Widget`.

Using the Ttk widgets gives the application an improved look and feel. As discussed above, there are differences in how the styling is coded.

Tk code:

```
l1 = tkinter.Label(text="Test", fg="black", bg="white")
l2 = tkinter.Label(text="Test", fg="black", bg="white")
```

Ttk code:

```
style = ttk.Style()
style.configure("BW.TLabel", foreground="black",
background="white")

l1 = ttk.Label(text="Test", style="BW.TLabel")
l2 = ttk.Label(text="Test", style="BW.TLabel")
```

For more information about TtkStyling, see the **`Style`** class documentation.

Widget

`ttk.Widget` defines standard options and methods supported by Tk themed widgets and is not supposed to be directly instantiated.

Standard Options

All the **`ttk`** Widgets accept the following options:

Option	Description
class	Specifies the window class. The class is used when querying the option database for the window’s other options, to determine the default bindtags for the window, and to select the widget’s default layout and style. This

Option	Description
	option is read-only, and may only be specified when the window is created.
cursor	Specifies the mouse cursor to be used for the widget. If set to the empty string (the default), the cursor is inherited for the parent widget.
takefocus	Determines whether the window accepts the focus during keyboard traversal. 0, 1 or an empty string is returned. If 0 is returned, it means that the window should be skipped entirely during keyboard traversal. If 1, it means that the window should receive the input focus as long as it is viewable. And an empty string means that the traversal scripts make the decision about whether or not to focus on the window.
style	May be used to specify a custom widget style.

Scrollable Widget Options

The following options are supported by widgets that are controlled by a scrollbar.

Option	Description
xscrollcommand	Used to communicate with horizontal scrollbars. When the view in the widget’s window change, the widget will generate a Tcl command based on the scrollcommand. Usually this option consists of the method `Scrollbar.set()` of some scrollbar. This will cause the scrollbar to be updated whenever the view in the window changes.
yscrollcommand	Used to communicate with vertical scrollbars. For some more information, see above.

Label Options

The following options are supported by labels, buttons and other button-like widgets.

Option	Description
text	Specifies a text string to be displayed inside the widget.

Option	Description
textvariable	Specifies a name whose value will be used in place of the text option resource.
underline	If set, specifies the index (0-based) of a character to underline in the text string. The underline character is used for mnemonic activation.
image	Specifies an image to display. This is a list of 1 or more elements. The first element is the default image name. The rest of the list if a sequence of statespec/value pairs as defined by `Style.map()`, specifying different images to use when the widget is in a particular state or a combination of states. All images in the list should have the same size.
compound	Specifies how to display the image relative to the text, in the case both text and images options are present. Valid values are: • text: display text only • image: display image only • top, bottom, left, right: display image above, below, left of, or right of the text, respectively. • none: the default. display the image if present, otherwise the text.

Option	Description
width	If greater than zero, specifies how much space, in character widths, to allocate for the text label, if less than zero, specifies a minimum width. If zero or unspecified, the natural width of the text label is used.

Compatibility Options

Option	Description
state	May be set to “normal” or “disabled” to control the “disabled” state bit. This is a write-only option: setting it changes the widget state, but the `Widget.state()` method does not affect this option.

Widget States

The widget state is a bitmap of independent state flags.

Flag	Description
active	The mouse cursor is over the widget and pressing a mouse button will cause some action to occur

Flag	Description
disabled	Widget is disabled under program control
focus	Widget has keyboard focus
pressed	Widget is being pressed
selected	“On”, “true”, or “current” for things like Checkbuttons and radiobuttons
background	Windows and Mac have a notion of an “active” or foreground window. The *background* state is set for widgets in a background window, and cleared for those in the foreground window
readonly	Widget should not allow user modification
alternate	A widget-specific alternate display format
invalid	The widget’s value is invalid

A state specification is a sequence of state names, optionally prefixed with an exclamation point indicating that the bit is off.

ttk.Widget

Besides the methods described below, the **`ttk.Widget`** supports the methods **`tkinter.Widget.cget()`** and **`tkinter.Widget.configure()`**.

class tkinter.ttk.Widget

identify(*x*, *y*)

Returns the name of the element at position *x* *y*, or the empty string if the point does not lie within any element.

x and *y* are pixel coordinates relative to the widget.

instate(*statespec*, *callback=None*, **args*, ***kw*)

Test the widget's state. If a callback is not specified, returns `True` if the widget state matches *statespec* and `False` otherwise. If callback is specified then it is called with args if widget state matches *statespec*.

state(*statespec=None*)

Modify or inquire widget state. If *statespec* is specified, sets the widget state according to it and return a new *statespec* indicating which flags were changed. If *statespec* is not specified, returns the currently enabled state flags.

statespec will usually be a list or a tuple.

Combobox

The **`ttk.Combobox`** widget combines a text field with a pop-down list of values. This widget is a subclass of **`Entry`**.

Besides the methods inherited from `Widget`: `Widget.cget()`, `Widget.configure()`, `Widget.identify()`, `Widget.instate()` and `Widget.state()`, and the following inherited from `Entry`: `Entry.bbox()`, `Entry.delete()`, `Entry.icursor()`, `Entry.index()`, `Entry.insert()`, `Entry.selection()`, `Entry.xview()`, it has some other methods, described at `ttk.Combobox`.

Options

This widget accepts the following specific options:

Option	Description
exportselection	Boolean value. If set, the widget selection is linked to the Window Manager selection (which can be returned by invoking Misc.selection_get, for example).
justify	Specifies how the text is aligned within the widget. One of “left”, “center”, or “right”.
height	Specifies the height of the pop-down listbox, in rows.
postcommand	A script (possibly registered with Misc.register) that is called immediately before displaying the values. It may specify which values to display.

Option	**Description**
state	One of “normal”, “readonly”, or “disabled”. In the “readonly” state, the value may not be edited directly, and the user can only selection of the values from the dropdown list. In the “normal” state, the text field is directly editable. In the “disabled” state, no interaction is possible.
textvariable	Specifies a name whose value is linked to the widget value. Whenever the value associated with that name changes, the widget value is updated, and vice versa. See `tkinter.StringVar`.
values	Specifies the list of values to display in the drop-down listbox.
width	Specifies an integer value indicating the desired width of the entry window, in average-size characters of the widget’s font.

Virtual events

The combobox widgets generates a **<<ComboboxSelected>>** virtual event when the user selects an element from the list of values.

ttk.Combobox

class tkinter.ttk.Combobox

current(*newindex=None*)

If *newindex* is specified, sets the combobox value to the element position *newindex*. Otherwise, returns the index of the current value or -1 if the current value is not in the values list.

get()

Returns the current value of the combobox.

set(*value*)

Sets the value of the combobox to *value*.

Spinbox

The **`ttk.Spinbox`** widget is a **`ttk.Entry`** enhanced with increment and decrement arrows. It can be used for numbers or lists of string values. This widget is a subclass of **`Entry`**.

Besides the methods inherited from **`Widget`**: **`Widget.cget()`**, **`Widget.configure()`**, **`Widget.identify()`**, **`Widget.instate()`** and **`Widget.state()`**, and the following inherited from **`Entry`**: **`Entry.bbox()`**, **`Entry.delete()`**, **`Entry.icursor()`**, **`Entry.index()`**, **`Entry.insert()`**, **`Entry.xview()`**, it has some other methods, described at **`ttk.Spinbox`**.

Options

This widget accepts the following specific options:

Option	Description
from	Float value. If set, this is the minimum value to which the decrement button will decrement. Must be spelled as `from_` when used as an argument, since `from` is a Python keyword.
to	Float value. If set, this is the maximum value to which the increment button will increment.
increment	Float value. Specifies the amount which the increment/decrement buttons change the value. Defaults to 1.0.
values	Sequence of string or float values. If specified, the increment/decrement buttons will cycle through the items in this sequence rather than incrementing or decrementing numbers.
wrap	Boolean value. If `True`, increment and decrement buttons will cycle from the `to` value to the `from` value or the `from` value to the `to` value, respectively.
format	String value. This specifies the format of numbers set by the increment/decrement buttons. It must be in the form “%W.Pf”, where W is the padded width of the value, P is the precision, and ‘%’ and ‘f’ are literal.

Option	Description
command	Python callable. Will be called with no arguments whenever either of the increment or decrement buttons are pressed.

Virtual events

The spinbox widget generates an **<<Increment>>** virtual event when the user presses <Up>, and a **<<Decrement>>** virtual event when the user presses <Down>.

ttk.Spinbox

class tkinter.ttk.Spinbox

get()

Returns the current value of the spinbox.

set(*value*)

Sets the value of the spinbox to *value*.

Notebook

Ttk Notebook widget manages a collection of windows and displays a single one at a time. Each child window is associated with a tab, which the user may select to change the currently displayed window.

Options

This widget accepts the following specific options:

Option	Description
height	If present and greater than zero, specifies the desired height of the pane area (not including internal padding or tabs). Otherwise, the maximum height of all panes is used.
padding	Specifies the amount of extra space to add around the outside of the notebook. The padding is a list up to four length specifications left top right bottom. If fewer than four elements are specified, bottom defaults to top, right defaults to left, and top defaults to left.
width	If present and greater than zero, specified the desired width of the pane area (not including internal padding). Otherwise, the maximum width of all panes is used.

Tab Options

There are also specific options for tabs:

Option	Description
state	Either “normal”, “disabled” or “hidden”. If “disabled”, then the tab is not selectable. If “hidden”, then the tab is not shown.

Option	Description
sticky	Specifies how the child window is positioned within the pane area. Value is a string containing zero or more of the characters “n”, “s”, “e” or “w”. Each letter refers to a side (north, south, east or west) that the child window will stick to, as per the `grid()` geometry manager.
padding	Specifies the amount of extra space to add between the notebook and this pane. Syntax is the same as for the option padding used by this widget.
text	Specifies a text to be displayed in the tab.
image	Specifies an image to display in the tab. See the option image described in `Widget`.
compound	Specifies how to display the image relative to the text, in the case both options text and image are present. See Label Options for legal values.
underline	Specifies the index (0-based) of a character to underline in the text string. The underlined character is used for mnemonic activation if `Notebook.enable_traversal()` is called.

Tab Identifiers

The tab_id present in several methods of **`ttk.Notebook`** may take any of the following forms:

- An integer between zero and the number of tabs
- The name of a child window
- A positional specification of the form “@x,y”, which identifies the tab
- The literal string “current”, which identifies the currently selected tab
- The literal string “end”, which returns the number of tabs (only valid for **`Notebook.index()`**)

Virtual Events

This widget generates a **<<NotebookTabChanged>>** virtual event after a new tab is selected.

ttk.Notebook

class tkinter.ttk.Notebook

add(*child*, ***kw*)

Adds a new tab to the notebook.

If window is currently managed by the notebook but hidden, it is restored to its previous position.

See Tab Options for the list of available options.

forget(*tab_id*)

Removes the tab specified by *tab_id*, unmaps and unmanages the associated window.

hide(*tab_id*)

Hides the tab specified by *tab_id*.

The tab will not be displayed, but the associated window remains managed by the notebook and its configuration remembered. Hidden tabs may be restored with the `add()` command.

identify(*x*, *y*)

Returns the name of the tab element at position *x*, *y*, or the empty string if none.

index(*tab_id*)

Returns the numeric index of the tab specified by *tab_id*, or the total number of tabs if *tab_id* is the string “end”.

insert(*pos*, *child*, ***kw*)

Inserts a pane at the specified position.

pos is either the string “end”, an integer index, or the name of a managed child. If *child* is already managed by the notebook, moves it to the specified position.

See Tab Options for the list of available options.

select(*tab_id=None*)

Selects the specified *tab_id*.

The associated child window will be displayed, and the previously selected window (if different) is unmapped. If *tab_id* is omitted, returns the widget name of the currently selected pane.

tab(*tab_id*, *option=None*, ***kw*)

Query or modify the options of the specific *tab_id*.

If *kw* is not given, returns a dictionary of the tab option values. If *option* is specified, returns the value of that *option*. Otherwise, sets the options to the corresponding values.

tabs()

Returns a list of windows managed by the notebook.

enable_traversal()

Enable keyboard traversal for a toplevel window containing this notebook.

This will extend the bindings for the toplevel window containing the notebook as follows:

- `Control-Tab`: selects the tab following the currently selected one.
- `Shift-Control-Tab`: selects the tab preceding the currently selected one.
- `Alt-K`: where *K* is the mnemonic (underlined) character of any tab, will select that tab.

Multiple notebooks in a single toplevel may be enabled for traversal, including nested notebooks. However, notebook traversal only works properly if all panes have the notebook they are in as master.

Progressbar

The **`ttk.Progressbar`** widget shows the status of a long-running operation. It can operate in two modes: 1) the determinate mode which shows the amount completed relative to the total amount of work to be done and 2) the indeterminate mode which provides an animated display to let the user know that work is progressing.

Options

This widget accepts the following specific options:

Option	Description
orient	One of “horizontal” or “vertical”. Specifies the orientation of the progress bar.
length	Specifies the length of the long axis of the progress bar (width if horizontal, height if vertical).
mode	One of “determinate” or “indeterminate”.
maximum	A number specifying the maximum value. Defaults to 100.
value	The current value of the progress bar. In “determinate” mode, this represents the amount of work completed. In “indeterminate” mode, it is interpreted as modulo *maximum*; that is, the progress bar completes one “cycle” when its value increases by *maximum*.
variable	A name which is linked to the option value. If specified, the value of the progress bar is automatically set to the value of this name whenever the latter is modified.

Option	Description
phase	Read-only option. The widget periodically increments the value of this option whenever its value is greater than 0 and, in determinate mode, less than maximum. This option may be used by the current theme to provide additional animation effects.

ttk.Progressbar

class tkinter.ttk.Progressbar

start(*interval=None*)

Begin autoincrement mode: schedules a recurring timer event that calls `Progressbar.step()` every *interval* milliseconds. If omitted, *interval* defaults to 50 milliseconds.

step(*amount=None*)

Increments the progress bar’s value by *amount*.

amount defaults to 1.0 if omitted.

stop()

Stop autoincrement mode: cancels any recurring timer event initiated by `Progressbar.start()` for this progress bar.

Separator

The **`ttk.Separator`** widget displays a horizontal or vertical separator bar.

It has no other methods besides the ones inherited from **`ttk.Widget`**.

Options

This widget accepts the following specific option:

Option	Description
orient	One of “horizontal” or “vertical”. Specifies the orientation of the separator.

Sizegrip

The **`ttk.Sizegrip`** widget (also known as a grow box) allows the user to resize the containing toplevel window by pressing and dragging the grip.

This widget has neither specific options nor specific methods, besides the ones inherited from **`ttk.Widget`**.

Platform-specific notes

- On macOS, toplevel windows automatically include a built-in size grip by default. Adding a **`Sizegrip`** is harmless, since the built-in grip will just mask the widget.

Bugs

- If the containing toplevel’s position was specified relative to the right or bottom of the screen (e.g. ….), the **`Sizegrip`** widget will not resize the window.

- This widget supports only “southeast” resizing.

Treeview

The **`ttk.Treeview`** widget displays a hierarchical collection of items. Each item has a textual label, an optional image, and an optional list of data values. The data values are displayed in successive columns after the tree label.

The order in which data values are displayed may be controlled by setting the widget option `displaycolumns`. The tree widget can also display column headings. Columns may be accessed by number or symbolic names listed in the widget option columns. See Column Identifiers.

Each item is identified by a unique name. The widget will generate item IDs if they are not supplied by the caller. There is a distinguished root item, named `{}`. The root item itself is not displayed; its children appear at the top level of the hierarchy.

Each item also has a list of tags, which can be used to associate event bindings with individual items and control the appearance of the item.

The Treeview widget supports horizontal and vertical scrolling, according to the options described in Scrollable Widget Options and the methods **`Treeview.xview()`** and **`Treeview.yview()`**.

Options

This widget accepts the following specific options:

Option	Description
columns	A list of column identifiers, specifying the number of columns and their names.
displaycolumns	A list of column identifiers (either symbolic or integer indices) specifying which data columns are displayed and the order in which they appear, or the string “#all”.
height	Specifies the number of rows which should be visible. Note: the requested width is determined from the sum of the column widths.
padding	Specifies the internal padding for the widget. The padding is a list of up to four length specifications.
selectmode	Controls how the built-in class bindings manage the selection. One of “extended”, “browse” or “none”. If set to “extended” (the default), multiple items may be selected. If “browse”, only a single item will be selected at a time. If “none”, the selection will not be changed. Note that the application code and tag bindings can set the selection however

Option	Description
	they wish, regardless of the value of this option.
show	A list containing zero or more of the following values, specifying which elements of the tree to display. • tree: display tree labels in column #0. • headings: display the heading row. The default is “tree headings”, i.e., show all elements. **Note**: Column #0 always refers to the tree column, even if show=”tree” is not specified.

Item Options

The following item options may be specified for items in the insert and item widget commands.

Option	Description
text	The textual label to display for the item.
image	A Tk Image, displayed to the left of the label.

Option	Description
values	The list of values associated with the item. Each item should have the same number of values as the widget option columns. If there are fewer values than columns, the remaining values are assumed empty. If there are more values than columns, the extra values are ignored.
open	`True`/`False` value indicating whether the item's children should be displayed or hidden.
tags	A list of tags associated with this item.

Tag Options

The following options may be specified on tags:

Option	Description
foreground	Specifies the text foreground color.
background	Specifies the cell or item background color.
font	Specifies the font to use when drawing text.

Option	Description
image	Specifies the item image, in case the item’s image option is empty.

Column Identifiers

Column identifiers take any of the following forms:

- A symbolic name from the list of columns option.
- An integer n, specifying the nth data column.
- A string of the form #n, where n is an integer, specifying the nth display column.

Notes:

- Item’s option values may be displayed in a different order than the order in which they are stored.
- Column #0 always refers to the tree column, even if show=”tree” is not specified.

A data column number is an index into an item’s option values list; a display column number is the column number in the tree where the values are displayed. Tree labels are displayed in column #0. If option displaycolumns is not set, then data column n is displayed in column #n+1. Again, **column #0 always refers to the tree column**.

Virtual Events

The Treeview widget generates the following virtual events.

Event	**Description**
<<TreeviewSelect>>	Generated whenever the selection changes.
<<TreeviewOpen>>	Generated just before settings the focus item to open=True.
<<TreeviewClose>>	Generated just after setting the focus item to open=False.

The `Treeview.focus()` and `Treeview.selection()` methods can be used to determine the affected item or items.

ttk.Treeview

class tkinter.ttk.Treeview

bbox(*item*, *column=None*)

Returns the bounding box (relative to the treeview widget's window) of the specified *item* in the form (x, y, width, height).

If *column* is specified, returns the bounding box of that cell. If the *item* is not visible (i.e., if it is a descendant of a closed item or is scrolled offscreen), returns an empty string.

get_children(*item=None*)

Returns the list of children belonging to *item*.

If *item* is not specified, returns root children.

set_children(*item*, **newchildren*)

Replaces *item*’s child with *newchildren*.

Children present in *item* that are not present in *newchildren* are detached from the tree. No items in *newchildren* may be an ancestor of *item*. Note that not specifying *newchildren* results in detaching *item*’s children.

column(*column*, *option=None*, ***kw*)

Query or modify the options for the specified *column*.

If *kw* is not given, returns a dict of the column option values. If *option* is specified then the value for that *option* is returned. Otherwise, sets the options to the corresponding values.

The valid options/values are:

id

Returns the column name. This is a read-only option.

anchor: One of the standard Tk anchor values.

Specifies how the text in this column should be aligned with respect to the cell.

minwidth: width

The minimum width of the column in pixels. The treeview widget will not make the column any smaller than specified by this option when the widget is resized or the user drags a column.

stretch: `True`/`False`

Specifies whether the column’s width should be adjusted when the widget is resized.

width: width

The width of the column in pixels.

To configure the tree column, call this with column = “#0”

delete(**items*)

Delete all specified *items* and all their descendants.

The root item may not be deleted.

detach(**items*)

Unlinks all of the specified *items* from the tree.

The items and all of their descendants are still present, and may be reinserted at another point in the tree, but will not be displayed.

The root item may not be detached.

exists(*item*)

Returns `True` if the specified *item* is present in the tree.

focus(*item=None*)

If *item* is specified, sets the focus item to *item*. Otherwise, returns the current focus item, or ‘’ if there is none.

heading(*column*, *option=None*, ***kw*)

Query or modify the heading options for the specified *column*.

If *kw* is not given, returns a dict of the heading option values. If *option* is specified then the value for that *option* is returned. Otherwise, sets the options to the corresponding values.

The valid options/values are:

text: text
The text to display in the column heading.

image: imageName
Specifies an image to display to the right of the column heading.

anchor: anchor
Specifies how the heading text should be aligned. One of the standard Tk anchor values.

command: callback
A callback to be invoked when the heading label is pressed.

To configure the tree column heading, call this with column = “#0”.

identify(*component*, *x*, *y*)

Returns a description of the specified *component* under the point given by *x* and *y*, or the empty string if no such *component* is present at that position.

identify_row(*y*)

Returns the item ID of the item at position *y*.

identify_column(*x*)

Returns the data column identifier of the cell at position *x*.

The tree column has ID #0.

identify_region(*x*, *y*)

Returns one of:

region	**meaning**
heading	Tree heading area.
separator	Space between two columns headings.
tree	The tree area.
cell	A data cell.

Availability: Tk 8.6.

identify_element(*x*, *y*)

Returns the element at position *x*, *y*.

Availability: Tk 8.6.

index(*item*)

Returns the integer index of *item* within its parent's list of children.

insert(*parent*, *index*, *iid=None*, ***kw*)

Creates a new item and returns the item identifier of the newly created item.

parent is the item ID of the parent item, or the empty string to create a new top-level item. *index* is an integer, or the value “end”, specifying where in the list of parent's children to insert the new item. If *index* is less than or equal to zero, the new node is inserted at the beginning; if *index* is greater than or equal to the current number of children, it is inserted at the end. If *iid* is specified, it is used as the item identifier; *iid* must not already exist in the tree. Otherwise, a new unique identifier is generated.

See Item Options for the list of available options.

item(*item*, *option=None*, ***kw*)

Query or modify the options for the specified *item*.

If no options are given, a dict with options/values for the item is returned. If *option* is specified then the value for that option is returned. Otherwise, sets the options to the corresponding values as given by *kw*.

move(*item*, *parent*, *index*)

Moves *item* to position *index* in *parent*’s list of children.

It is illegal to move an item under one of its descendants. If *index* is less than or equal to zero, *item* is moved to the beginning; if greater than or equal to the number of children, it is moved to the end. If *item* was detached it is reattached.

next(*item*)

Returns the identifier of *item*’s next sibling, or ‘’ if *item* is the last child of its parent.

parent(*item*)

Returns the ID of the parent of *item*, or ‘’ if *item* is at the top level of the hierarchy.

prev(*item*)

Returns the identifier of *item*’s previous sibling, or ‘’ if *item* is the first child of its parent.

reattach(*item*, *parent*, *index*)

An alias for `Treeview.move()`.

see(*item*)

Ensure that *item* is visible.

Sets all of *item*’s ancestors open option to `True`, and scrolls the widget if necessary so that *item* is within the visible portion of the tree.

selection()

Returns a tuple of selected items.

Changed in version 3.8: `selection()` no longer takes arguments. For changing the selection state use the following selection methods.

selection_set(**items*)

items becomes the new selection.

Changed in version 3.6: items can be passed as separate arguments, not just as a single tuple.

selection_add(**items*)

Add *items* to the selection.

Changed in version 3.6: items can be passed as separate arguments, not just as a single tuple.

selection_remove(**items*)

Remove *items* from the selection.

Changed in version 3.6: items can be passed as separate arguments, not just as a single tuple.

selection_toggle(**items*)

Toggle the selection state of each item in *items*.

Changed in version 3.6: items can be passed as separate arguments, not just as a single tuple.

set(*item*, *column=None*, *value=None*)

With one argument, returns a dictionary of column/value pairs for the specified *item*. With two arguments, returns the current value of the specified *column*. With three arguments, sets the value of given *column* in given *item* to the specified *value*.

tag_bind(*tagname*, *sequence=None*, *callback=None*)

Bind a callback for the given event *sequence* to the tag *tagname*. When an event is delivered to an item, the callbacks for each of the item's tags option are called.

tag_configure(*tagname*, *option=None*, ***kw*)

Query or modify the options for the specified *tagname*.

If *kw* is not given, returns a dict of the option settings for *tagname*. If *option* is specified, returns the value for that *option* for the specified *tagname*. Otherwise, sets the options to the corresponding values for the given *tagname*.

tag_has(*tagname*, *item=None*)

If *item* is specified, returns 1 or 0 depending on whether the specified *item* has the given *tagname*. Otherwise, returns a list of all items that have the specified tag.

Availability: Tk 8.6

xview(**args*)

Query or modify horizontal position of the treeview.

yview(**args*)

Query or modify vertical position of the treeview.

Ttk Styling

Each widget in `ttk` is assigned a style, which specifies the set of elements making up the widget and how they are arranged, along with dynamic and default settings for element options. By default the style name is the same as the widget’s class name, but it may be overridden by the widget’s style option. If you don’t know the class name of a widget, use the method `Misc.winfo_class()` (somewidget.winfo_class()).

See also

[Tcl’2004 conference presentation](https://tktable.sourceforge.net/tile/tile-tcl2004.pdf)
[https://tktable.sourceforge.net/tile/tile-tcl2004.pdf]
This document explains how the theme engine works

class tkinter.ttk.Style

This class is used to manipulate the style database.

configure(*style*, *query_opt=None*, ***kw*)

Query or set the default value of the specified option(s) in *style*.

Each key in *kw* is an option and each value is a string identifying the value for that option.

For example, to change every default button to be a flat button with some padding and a different background color:

```
from tkinter import ttk
import tkinter
```

```
root = tkinter.Tk()

ttk.Style().configure("TButton", padding=6,
relief="flat",
   background="#ccc")

btn = ttk.Button(text="Sample")
btn.pack()

root.mainloop()
```

map(*style*, *query_opt=None*, ***kw*)

Query or sets dynamic values of the specified option(s) in *style*.

Each key in *kw* is an option and each value should be a list or a tuple (usually) containing statespecs grouped in tuples, lists, or some other preference. A statespec is a compound of one or more states and then a value.

An example may make it more understandable:

```
import tkinter
from tkinter import ttk

root = tkinter.Tk()

style = ttk.Style()
style.map("C.TButton",
    foreground=[('pressed', 'red'), ('active', 'blue')],
    background=[('pressed', '!disabled', 'black'),
('active', 'white')]
    )

colored_btn = ttk.Button(text="Test",
style="C.TButton").pack()

root.mainloop()
```

Note that the order of the (states, value) sequences for an option does matter, if the order is changed to `[('active', 'blue'), ('pressed', 'red')]` in the foreground option, for example, the result would be a blue foreground when the widget were in active or pressed states.

lookup(*style*, *option*, *state=None*, *default=None*)

Returns the value specified for *option* in *style*.

If *state* is specified, it is expected to be a sequence of one or more states. If the *default* argument is set, it is used as a fallback value in case no specification for option is found.

To check what font a Button uses by default:

```
from tkinter import ttk

print(ttk.Style().lookup("TButton", "font"))
```

layout(*style*, *layoutspec=None*)

Define the widget layout for given *style*. If *layoutspec* is omitted, return the layout specification for given style.

layoutspec, if specified, is expected to be a list or some other sequence type (excluding strings), where each item should be a tuple and the first item is the layout name and the second item should have the format described in Layouts.

To understand the format, see the following example (it is not intended to do anything useful):

```
from tkinter import ttk
import tkinter
```

```
root = tkinter.Tk()

style = ttk.Style()
style.layout("TMenubutton", [
   ("Menubutton.background", None),
   ("Menubutton.button", {"children":
       [("Menubutton.focus", {"children":
           [("Menubutton.padding", {"children":
               [("Menubutton.label", {"side": "left",
"expand": 1})]
           })]
       })]
   }),
])

mbtn = ttk.Menubutton(text='Text')
mbtn.pack()
root.mainloop()
```

element_create(*elementname*, *etype*, **args*, ***kw*)

Create a new element in the current theme, of the given *etype* which is expected to be either “image”, “from” or “vsapi”. The latter is only available in Tk 8.6 on Windows.

If “image” is used, *args* should contain the default image name followed by statespec/value pairs (this is the imagespec), and *kw* may have the following options:

border=padding

padding is a list of up to four integers, specifying the left, top, right, and bottom borders, respectively.

height=height

Specifies a minimum height for the element. If less than zero, the base image’s height is used as a default.

padding=padding

Specifies the element’s interior padding. Defaults to border’s value if not specified.

sticky=spec

Specifies how the image is placed within the final parcel. spec contains zero or more characters “n”, “s”, “w”, or “e”.

width=width

Specifies a minimum width for the element. If less than zero, the base image’s width is used as a default.

Example:

```
img1 = tkinter.PhotoImage(master=root, file='button.png')
img1 = tkinter.PhotoImage(master=root, file='button-
pressed.png')
img1 = tkinter.PhotoImage(master=root, file='button-
active.png')
style = ttk.Style(root)
style.element_create('Button.button', 'image',
                     img1, ('pressed', img2), ('active',
img3),
                     border=(2, 4), sticky='we')
```

If “from” is used as the value of *etype*, `element_create()` will clone an existing element. *args* is expected to contain a themename, from which the element will be cloned, and optionally an element to clone from. If this element to clone from is not specified, an empty element will be used. *kw* is discarded.

Example:

```
style = ttk.Style(root)
style.element_create('plain.background', 'from',
'default')
```

If “vsapi” is used as the value of *etype*, `element_create()` will create a new element in the current theme whose visual appearance is drawn using the Microsoft Visual Styles API which is responsible for the themed styles on Windows XP and Vista. *args* is expected to contain the Visual Styles class and part as given in the Microsoft documentation followed by an optional sequence of tuples of ttk states and the corresponding Visual Styles API state value. *kw* may have the following options:

padding=padding
: Specify the element’s interior padding. *padding* is a list of up to four integers specifying the left, top, right and bottom padding quantities respectively. If fewer than four elements are specified, bottom defaults to top, right defaults to left, and top defaults to left. In other words, a list of three numbers specify the left, vertical, and right padding; a list of two numbers specify the horizontal and the vertical padding; a single number specifies the same padding all the way around the widget. This option may not be mixed with any other options.

margins=padding
: Specifies the elements exterior padding. *padding* is a list of up to four integers specifying the left, top, right and bottom padding quantities respectively. This option may not be mixed with any other options.

width=width
: Specifies the width for the element. If this option is set then the Visual Styles API will not be queried for the recommended size or the part. If this option is

set then *height* should also be set. The *width* and *height* options cannot be mixed with the *padding* or *margins* options.

height=height

Specifies the height of the element. See the comments for *width*.

Example:

```
style = ttk.Style(root)
style.element_create('pin', 'vsapi', 'EXPLORERBAR', 3, [
                     ('pressed', '!selected', 3),
                     ('active', '!selected', 2),
                     ('pressed', 'selected', 6),
                     ('active', 'selected', 5),
                     ('selected', 4),
                     ('', 1)])
style.layout('Explorer.Pin',
             [('Explorer.Pin.pin', {'sticky': 'news'})])
pin = ttk.Checkbutton(style='Explorer.Pin')
pin.pack(expand=True, fill='both')
```

Changed in version 3.13: Added support of the “vsapi” element factory.

element_names()

Returns the list of elements defined in the current theme.

element_options(*elementname*)

Returns the list of *elementname*’s options.

theme_create(*themename*, *parent=None*, *settings=None*)

Create a new theme.

It is an error if *themename* already exists. If *parent* is specified, the new theme will inherit styles, elements and layouts from the parent theme. If *settings* are present they are expected to have the same syntax used for `theme_settings()`.

theme_settings(*themename*, *settings*)

Temporarily sets the current theme to *themename*, apply specified *settings* and then restore the previous theme.

Each key in *settings* is a style and each value may contain the keys ‘configure’, ‘map’, ‘layout’ and ‘element create’ and they are expected to have the same format as specified by the methods `Style.configure()`, `Style.map()`, `Style.layout()` and `Style.element_create()` respectively.

As an example, let’s change the Combobox for the default theme a bit:

```
from tkinter import ttk
import tkinter

root = tkinter.Tk()

style = ttk.Style()
style.theme_settings("default", {
   "TCombobox": {
       "configure": {"padding": 5},
       "map": {
           "background": [("active", "green2"),
                          ("!disabled", "green4")],
           "fieldbackground": [("!disabled", "green3")],
           "foreground": [("focus", "OliveDrab1"),
                          ("!disabled", "OliveDrab2")]
       }
   }
})

combo = ttk.Combobox().pack()
```

```
root.mainloop()
```

theme_names()

Returns a list of all known themes.

theme_use(*themename=None*)

If *themename* is not given, returns the theme in use. Otherwise, sets the current theme to *themename*, refreshes all widgets and emits a <<ThemeChanged>> event.

Layouts

A layout can be just `None`, if it takes no options, or a dict of options specifying how to arrange the element. The layout mechanism uses a simplified version of the pack geometry manager: given an initial cavity, each element is allocated a parcel.

The valid options/values are:

side: whichside

Specifies which side of the cavity to place the element; one of top, right, bottom or left. If omitted, the element occupies the entire cavity.

sticky: nswe

Specifies where the element is placed inside its allocated parcel.

unit: 0 or 1

If set to 1, causes the element and all of its descendants to be treated as a single element for the purposes of **`Widget.identify()`** et al. It's used for things like scrollbar thumbs with grips.

children: [sublayout…]

Specifies a list of elements to place inside the element. Each element is a tuple (or other sequence type) where the first item is the layout name, and the other is a Layout.

IDLE — Python editor and shell

Source code: Lib/idlelib/
[https://github.com/python/cpython/tree/3.14/Lib/idlelib/]

IDLE is Python’s Integrated Development and Learning Environment.

IDLE has the following features:

- cross-platform: works mostly the same on Windows, Unix, and macOS
- Python shell window (interactive interpreter) with colorizing of code input, output, and error messages
- multi-window text editor with multiple undo, Python colorizing, smart indent, call tips, auto completion, and other features
- search within any window, replace within editor windows, and search through multiple files (grep)
- debugger with persistent breakpoints, stepping, and viewing of global and local namespaces
- configuration, browsers, and other dialogs

The IDLE application is implemented in the **`idlelib`** package.

This is an optional module. If it is missing from your copy of CPython, look for documentation from your distributor (that is, whoever provided Python to you). If you are the distributor, see Requirements for optional modules.

Menus

IDLE has two main window types, the Shell window and the Editor window. It is possible to have multiple editor windows simultaneously. On Windows and Linux, each has its own top menu. Each menu documented below indicates which window type it is associated with.

Output windows, such as used for Edit => Find in Files, are a subtype of editor window. They currently have the same top menu but a different default title and context menu.

On macOS, there is one application menu. It dynamically changes according to the window currently selected. It has an IDLE menu, and some entries described below are moved around to conform to Apple guidelines.

File menu (Shell and Editor)

New File
Create a new file editing window.

Open…
Open an existing file with an Open dialog.

Open Module…
Open an existing module (searches sys.path).

Recent Files
Open a list of recent files. Click one to open it.

Module Browser
Show functions, classes, and methods in the current Editor file in a tree structure. In the shell, open a module first.

Path Browser
Show sys.path directories, modules, functions, classes and methods in a tree structure.

Save
: Save the current window to the associated file, if there is one. Windows that have been changed since being opened or last saved have a * before and after the window title. If there is no associated file, do Save As instead.

Save As…
: Save the current window with a Save As dialog. The file saved becomes the new associated file for the window. (If your file manager is set to hide extensions, the current extension will be omitted in the file name box. If the new filename has no ‘.’, ‘.py’ and ‘.txt’ will be added for Python and text files, except that on macOS Aqua,’.py’ is added for all files.)

Save Copy As…
: Save the current window to different file without changing the associated file. (See Save As note above about filename extensions.)

Print Window
: Print the current window to the default printer.

Close Window
: Close the current window (if an unsaved editor, ask to save; if an unsaved Shell, ask to quit execution). Calling `exit()` or `close()` in the Shell window also closes Shell. If this is the only window, also exit IDLE.

Exit IDLE
: Close all windows and quit IDLE (ask to save unsaved edit windows).

Edit menu (Shell and Editor)

Undo
Undo the last change to the current window. A maximum of 1000 changes may be undone.

Redo
Redo the last undone change to the current window.

Select All
Select the entire contents of the current window.

Cut
Copy selection into the system-wide clipboard; then delete the selection.

Copy
Copy selection into the system-wide clipboard.

Paste
Insert contents of the system-wide clipboard into the current window.

The clipboard functions are also available in context menus.

Find…
Open a search dialog with many options

Find Again
Repeat the last search, if there is one.

Find Selection
Search for the currently selected string, if there is one.

Find in Files…
Open a file search dialog. Put results in a new output window.

Replace…

Open a search-and-replace dialog.

Go to Line

Move the cursor to the beginning of the line requested and make that line visible. A request past the end of the file goes to the end. Clear any selection and update the line and column status.

Show Completions

Open a scrollable list allowing selection of existing names. See Completions in the Editing and navigation section below.

Expand Word

Expand a prefix you have typed to match a full word in the same window; repeat to get a different expansion.

Show Call Tip

After an unclosed parenthesis for a function, open a small window with function parameter hints. See Calltips in the Editing and navigation section below.

Show Surrounding Parens

Highlight the surrounding parenthesis.

Format menu (Editor window only)

Format Paragraph

Reformat the current blank-line-delimited paragraph in comment block or multiline string or selected line in a string. All lines in the paragraph will be formatted to less than N columns, where N defaults to 72.

Indent Region

Shift selected lines right by the indent width (default 4 spaces).

Dedent Region
Shift selected lines left by the indent width (default 4 spaces).

Comment Out Region
Insert ## in front of selected lines.

Uncomment Region
Remove leading # or ## from selected lines.

Tabify Region
Turn *leading* stretches of spaces into tabs. (Note: We recommend using 4 space blocks to indent Python code.)

Untabify Region
Turn *all* tabs into the correct number of spaces.

Toggle Tabs
Open a dialog to switch between indenting with spaces and tabs.

New Indent Width
Open a dialog to change indent width. The accepted default by the Python community is 4 spaces.

Strip Trailing Whitespace
Remove trailing space and other whitespace characters after the last non-whitespace character of a line by applying `str.rstrip()` to each line, including lines within multiline strings. Except for Shell windows, remove extra newlines at the end of the file.

Run menu (Editor window only)

Run Module
Do Check Module. If no error, restart the shell to clean the environment, then execute the module. Output is

displayed in the Shell window. Note that output requires use of `print` or `write`. When execution is complete, the Shell retains focus and displays a prompt. At this point, one may interactively explore the result of execution. This is similar to executing a file with `python -i file` at a command line.

Run... Customized

Same as Run Module, but run the module with customized settings. *Command Line Arguments* extend `sys.argv` as if passed on a command line. The module can be run in the Shell without restarting.

Check Module

Check the syntax of the module currently open in the Editor window. If the module has not been saved IDLE will either prompt the user to save or autosave, as selected in the General tab of the Idle Settings dialog. If there is a syntax error, the approximate location is indicated in the Editor window.

Python Shell

Open or wake up the Python Shell window.

Shell menu (Shell window only)

View Last Restart

Scroll the shell window to the last Shell restart.

Restart Shell

Restart the shell to clean the environment and reset display and exception handling.

Previous History

Cycle through earlier commands in history which match the current entry.

Next History
Cycle through later commands in history which match the current entry.

Interrupt Execution
Stop a running program.

Debug menu (Shell window only)

Go to File/Line
Look on the current line. with the cursor, and the line above for a filename and line number. If found, open the file if not already open, and show the line. Use this to view source lines referenced in an exception traceback and lines found by Find in Files. Also available in the context menu of the Shell window and Output windows.

Debugger (toggle)
When activated, code entered in the Shell or run from an Editor will run under the debugger. In the Editor, breakpoints can be set with the context menu. This feature is still incomplete and somewhat experimental.

Stack Viewer
Show the stack traceback of the last exception in a tree widget, with access to locals and globals.

Auto-open Stack Viewer
Toggle automatically opening the stack viewer on an unhandled exception.

Options menu (Shell and Editor)

Configure IDLE
Open a configuration dialog and change preferences for the following: fonts, indentation, keybindings, text color

themes, startup windows and size, additional help sources, and extensions. On macOS, open the configuration dialog by selecting Preferences in the application menu. For more details, see Setting preferences under Help and preferences.

Most configuration options apply to all windows or all future windows. The option items below only apply to the active window.

Show/Hide Code Context (Editor Window only)
Open a pane at the top of the edit window which shows the block context of the code which has scrolled above the top of the window. See Code Context in the Editing and Navigation section below.

Show/Hide Line Numbers (Editor Window only)
Open a column to the left of the edit window which shows the number of each line of text. The default is off, which may be changed in the preferences (see Setting preferences).

Zoom/Restore Height
Toggles the window between normal size and maximum height. The initial size defaults to 40 lines by 80 chars unless changed on the General tab of the Configure IDLE dialog. The maximum height for a screen is determined by momentarily maximizing a window the first time one is zoomed on the screen. Changing screen settings may invalidate the saved height. This toggle has no effect when a window is maximized.

Window menu (Shell and Editor)

Lists the names of all open windows; select one to bring it to the foreground (deiconifying it if necessary).

Help menu (Shell and Editor)

About IDLE
: Display version, copyright, license, credits, and more.

IDLE Help
: Display this IDLE document, detailing the menu options, basic editing and navigation, and other tips.

Python Docs
: Access local Python documentation, if installed, or start a web browser and open docs.python.org showing the latest Python documentation.

Turtle Demo
: Run the turtledemo module with example Python code and turtle drawings.

Additional help sources may be added here with the Configure IDLE dialog under the General tab. See the Help sources subsection below for more on Help menu choices.

Context menus

Open a context menu by right-clicking in a window (Control-click on macOS). Context menus have the standard clipboard functions also on the Edit menu.

Cut
: Copy selection into the system-wide clipboard; then delete the selection.

Copy
: Copy selection into the system-wide clipboard.

Paste

Insert contents of the system-wide clipboard into the current window.

Editor windows also have breakpoint functions. Lines with a breakpoint set are specially marked. Breakpoints only have an effect when running under the debugger. Breakpoints for a file are saved in the user's `.idlerc` directory.

Set Breakpoint
Set a breakpoint on the current line.

Clear Breakpoint
Clear the breakpoint on that line.

Shell and Output windows also have the following.

Go to file/line
Same as in Debug menu.

The Shell window also has an output squeezing facility explained in the *Python Shell window* subsection below.

Squeeze
If the cursor is over an output line, squeeze all the output between the code above and the prompt below down to a 'Squeezed text' label.

Editing and Navigation

Editor windows

IDLE may open editor windows when it starts, depending on settings and how you start IDLE. Thereafter, use the File menu. There can be only one open editor window for a given file.

The title bar contains the name of the file, the full path, and the version of Python and IDLE running the window. The status bar contains the line number (‘Ln’) and column number (‘Col’). Line numbers start with 1; column numbers with 0.

IDLE assumes that files with a known .py* extension contain Python code and that other files do not. Run Python code with the Run menu.

Key bindings

The IDLE insertion cursor is a thin vertical bar between character positions. When characters are entered, the insertion cursor and everything to its right moves right one character and the new character is entered in the new space.

Several non-character keys move the cursor and possibly delete characters. Deletion does not puts text on the clipboard, but IDLE has an undo list. Wherever this doc discusses keys, ‘C’ refers to the `Control` key on Windows and Unix and the `Command` key on macOS. (And all such discussions assume that the keys have not been re-bound to something else.)

- Arrow keys move the cursor one character or line.
- `C-LeftArrow` and `C-RightArrow` moves left or right one word.
- `Home` and `End` go to the beginning or end of the line.
- `Page Up` and `Page Down` go up or down one screen.
- `C-Home` and `C-End` go to beginning or end of the file.
- `Backspace` and `Del` (or `C-d`) delete the previous or next character.
- `C-Backspace` and `C-Del` delete one word left or right.
- `C-k` deletes (‘kills’) everything to the right.

Standard keybindings (like `C-c` to copy and `C-v` to paste) may work. Keybindings are selected in the Configure IDLE dialog.

Automatic indentation

After a block-opening statement, the next line is indented by 4 spaces (in the Python Shell window by one tab). After certain keywords (break, return etc.) the next line is dedented. In leading indentation, `Backspace` deletes up to 4 spaces if they are there. `Tab` inserts spaces (in the Python Shell window one tab), number depends on Indent width. Currently, tabs are restricted to four spaces due to Tcl/Tk limitations.

See also the indent/dedent region commands on the Format menu.

Search and Replace

Any selection becomes a search target. However, only selections within a line work because searches are only performed within lines with the terminal newline removed. If `[x] Regular expression` is checked, the target is interpreted according to the Python re module.

Completions

Completions are supplied, when requested and available, for module names, attributes of classes or functions, or filenames. Each request method displays a completion box with existing names. (See tab completions below for an exception.) For any box, change the name being completed and the item highlighted in the box by typing and deleting characters; by hitting `Up`, `Down`, `PageUp`, `PageDown`, `Home`, and `End` keys; and by a single click within the box. Close the box with

`Escape`, `Enter`, and double `Tab` keys or clicks outside the box. A double click within the box selects and closes.

One way to open a box is to type a key character and wait for a predefined interval. This defaults to 2 seconds; customize it in the settings dialog. (To prevent auto popups, set the delay to a large number of milliseconds, such as 100000000.) For imported module names or class or function attributes, type '.'. For filenames in the root directory, type `os.sep` or `os.altsep` immediately after an opening quote. (On Windows, one can specify a drive first.) Move into subdirectories by typing a directory name and a separator.

Instead of waiting, or after a box is closed, open a completion box immediately with Show Completions on the Edit menu. The default hot key is `C-space`. If one types a prefix for the desired name before opening the box, the first match or near miss is made visible. The result is the same as if one enters a prefix after the box is displayed. Show Completions after a quote completes filenames in the current directory instead of a root directory.

Hitting `Tab` after a prefix usually has the same effect as Show Completions. (With no prefix, it indents.) However, if there is only one match to the prefix, that match is immediately added to the editor text without opening a box.

Invoking 'Show Completions', or hitting `Tab` after a prefix, outside of a string and without a preceding '.' opens a box with keywords, builtin names, and available module-level names.

When editing code in an editor (as oppose to Shell), increase the available module-level names by running your code and not restarting the Shell thereafter. This is especially useful after adding imports at the top of a file. This also increases possible attribute completions.

Completion boxes initially exclude names beginning with ‘_’ or, for modules, not included in ‘__all__’. The hidden names can be accessed by typing ‘_’ after ‘.’, either before or after the box is opened.

Calltips

A calltip is shown automatically when one types `(` after the name of an *accessible* function. A function name expression may include dots and subscripts. A calltip remains until it is clicked, the cursor is moved out of the argument area, or `)` is typed. Whenever the cursor is in the argument part of a definition, select Edit and “Show Call Tip” on the menu or enter its shortcut to display a calltip.

The calltip consists of the function’s signature and docstring up to the latter’s first blank line or the fifth non-blank line. (Some builtin functions lack an accessible signature.) A ‘/’ or ‘*’ in the signature indicates that the preceding or following arguments are passed by position or name (keyword) only. Details are subject to change.

In Shell, the accessible functions depends on what modules have been imported into the user process, including those imported by Idle itself, and which definitions have been run, all since the last restart.

For example, restart the Shell and enter `itertools.count(`. A calltip appears because Idle imports itertools into the user process for its own use. (This could change.) Enter `turtle.write(` and nothing appears. Idle does not itself import turtle. The menu entry and shortcut also do nothing. Enter `import turtle`. Thereafter, `turtle.write(` will display a calltip.

In an editor, import statements have no effect until one runs the file. One might want to run a file after writing import statements, after adding function definitions, or after opening an existing file.

Code Context

Within an editor window containing Python code, code context can be toggled in order to show or hide a pane at the top of the window. When shown, this pane freezes the opening lines for block code, such as those beginning with `class`, `def`, or `if` keywords, that would have otherwise scrolled out of view. The size of the pane will be expanded and contracted as needed to show the all current levels of context, up to the maximum number of lines defined in the Configure IDLE dialog (which defaults to 15). If there are no current context lines and the feature is toggled on, a single blank line will display. Clicking on a line in the context pane will move that line to the top of the editor.

The text and background colors for the context pane can be configured under the Highlights tab in the Configure IDLE dialog.

Shell window

In IDLE’s Shell, enter, edit, and recall complete statements. (Most consoles and terminals only work with a single physical line at a time).

Submit a single-line statement for execution by hitting `Return` with the cursor anywhere on the line. If a line is extended with Backslash (`\`), the cursor must be on the last physical line. Submit a multi-line compound statement by entering a blank line after the statement.

When one pastes code into Shell, it is not compiled and possibly executed until one hits `Return`, as specified above. One may edit pasted code first. If one pastes more than one statement into Shell, the result will be a `SyntaxError` when multiple statements are compiled as if they were one.

Lines containing `RESTART` mean that the user execution process has been re-started. This occurs when the user execution process has crashed, when one requests a restart on the Shell menu, or when one runs code in an editor window.

The editing features described in previous subsections work when entering code interactively. IDLE's Shell window also responds to the following:

- `C-c` attempts to interrupt statement execution (but may fail).
- `C-d` closes Shell if typed at a `>>>` prompt.
- `Alt-p` and `Alt-n` (`C-p` and `C-n` on macOS) retrieve to the current prompt the previous or next previously entered statement that matches anything already typed.
- `Return` while the cursor is on any previous statement appends the latter to anything already typed at the prompt.

Text colors

Idle defaults to black on white text, but colors text with special meanings. For the shell, these are shell output, shell error, user output, and user error. For Python code, at the shell prompt or in an editor, these are keywords, builtin class and function names, names following `class` and `def`, strings, and comments. For any text window, these are the cursor (when present), found text (when possible), and selected text.

IDLE also highlights the soft keywords **`match`**, **`case`**, and **`_`** in pattern-matching statements. However, this highlighting is not perfect and will be incorrect in some rare cases, including some `_`-s in `case` patterns.

Text coloring is done in the background, so uncolorized text is occasionally visible. To change the color scheme, use the Configure IDLE dialog Highlighting tab. The marking of debugger breakpoint lines in the editor and text in popups and dialogs is not user-configurable.

Startup and Code Execution

Upon startup with the `-s` option, IDLE will execute the file referenced by the environment variables **`IDLESTARTUP`** or **`PYTHONSTARTUP`**. IDLE first checks for `IDLESTARTUP`; if `IDLESTARTUP` is present the file referenced is run. If `IDLESTARTUP` is not present, IDLE checks for `PYTHONSTARTUP`. Files referenced by these environment variables are convenient places to store functions that are used frequently from the IDLE shell, or for executing import statements to import common modules.

In addition, `Tk` also loads a startup file if it is present. Note that the Tk file is loaded unconditionally. This additional file is `.Idle.py` and is looked for in the user's home directory. Statements in this file will be executed in the Tk namespace, so this file is not useful for importing functions to be used from IDLE's Python shell.

Command-line usage

IDLE can be invoked from the command line with various options. The general syntax is:

```
python -m idlelib [options] [file ...]
```

The following options are available:

-c <command>
: Run the specified Python command in the shell window. For example, pass `-c "print('Hello, World!')"`. On Windows, the outer quotes must be double quotes as shown.

-d
: Enable the debugger and open the shell window.

-e
: Open an editor window.

-h
: Print a help message with legal combinations of options and exit.

-i
: Open a shell window.

-r <file>
: Run the specified file in the shell window.

-s
: Run the startup file (as defined by the environment variables **`IDLESTARTUP`** or `PYTHONSTARTUP`) before opening the shell window.

-t <title>
: Set the title of the shell window.

-
: Read and execute standard input in the shell window. This option must be the last one before any arguments.

If arguments are provided:

- If `-`, `-c`, or `-r` is used, all arguments are placed in `sys.argv[1:]`, and `sys.argv[0]` is set to `''`, `'-c'`, or `'-r'` respectively. No editor window is opened, even if that is the default set in the *Options* dialog.
- Otherwise, arguments are treated as files to be opened for editing, and `sys.argv` reflects the arguments passed to IDLE itself.

Startup failure

IDLE uses a socket to communicate between the IDLE GUI process and the user code execution process. A connection must be established whenever the Shell starts or restarts. (The latter is indicated by a divider line that says 'RESTART'). If the user process fails to connect to the GUI process, it usually displays a `Tk` error box with a 'cannot connect' message that directs the user here. It then exits.

One specific connection failure on Unix systems results from misconfigured masquerading rules somewhere in a system's network setup. When IDLE is started from a terminal, one will see a message starting with `** Invalid host:`. The valid value is `127.0.0.1 (idlelib.rpc.LOCALHOST)`. One can diagnose with `tcpconnect -irv 127.0.0.1 6543` in one terminal window and `tcplisten <same args>` in another.

A common cause of failure is a user-written file with the same name as a standard library module, such as *random.py* and *tkinter.py*. When such a file is located in the same directory as a file that is about to be run, IDLE cannot import the stdlib file. The current fix is to rename the user file.

Though less common than in the past, an antivirus or firewall program may stop the connection. If the program cannot be taught to allow the connection, then it must be turned off for IDLE to work. It is safe to allow this internal connection because no data is visible on external ports. A similar problem is a network mis-configuration that blocks connections.

Python installation issues occasionally stop IDLE: multiple versions can clash, or a single installation might need admin access. If one undo the clash, or cannot or does not want to run as admin, it might be easiest to completely remove Python and start over.

A zombie pythonw.exe process could be a problem. On Windows, use Task Manager to check for one and stop it if there is. Sometimes a restart initiated by a program crash or Keyboard Interrupt (control-C) may fail to connect. Dismissing the error box or using Restart Shell on the Shell menu may fix a temporary problem.

When IDLE first starts, it attempts to read user configuration files in `~/.idlerc/` (~ is one's home directory). If there is a problem, an error message should be displayed. Leaving aside random disk glitches, this can be prevented by never editing the files by hand. Instead, use the configuration dialog, under Options. Once there is an error in a user configuration file, the best solution may be to delete it and start over with the settings dialog.

If IDLE quits with no message, and it was not started from a console, try starting it from a console or terminal (`python -m idlelib`) and see if this results in an error message.

On Unix-based systems with tcl/tk older than `8.6.11` (see `About IDLE`) certain characters of certain fonts can cause a tk failure with a message to the terminal. This can happen either if one starts IDLE to edit a file with such a character or later when entering such a character. If one cannot upgrade tcl/tk, then re-configure IDLE to use a font that works better.

Running user code

With rare exceptions, the result of executing Python code with IDLE is intended to be the same as executing the same code by the default method, directly with Python in a text-mode system console or terminal window. However, the different interface and operation occasionally affect visible results. For instance, `sys.modules` starts with more entries, and `threading.active_count()` returns 2 instead of 1.

By default, IDLE runs user code in a separate OS process rather than in the user interface process that runs the shell and editor. In the execution process, it replaces `sys.stdin`, `sys.stdout`, and `sys.stderr` with objects that get input from and send output to the Shell window. The original values stored in `sys.__stdin__`, `sys.__stdout__`, and `sys.__stderr__` are not touched, but may be `None`.

Sending print output from one process to a text widget in another is slower than printing to a system terminal in the same process. This has the most effect when printing multiple arguments, as the string for each argument, each separator, the newline are sent separately. For development, this is usually not a problem, but if one wants to print faster in IDLE, format and join together everything one wants displayed together and then print a single string. Both format strings and **`str.join()`** can help combine fields and lines.

IDLE’s standard stream replacements are not inherited by subprocesses created in the execution process, whether directly by user code or by modules such as multiprocessing. If such subprocess use `input` from sys.stdin or `print` or `write` to sys.stdout or sys.stderr, IDLE should be started in a command line window. (On Windows, use `python` or `py` rather than `pythonw` or `pyw`.) The secondary subprocess will then be attached to that window for input and output.

If `sys` is reset by user code, such as with `importlib.reload(sys)`, IDLE’s changes are lost and input from the keyboard and output to the screen will not work correctly.

When Shell has the focus, it controls the keyboard and screen. This is normally transparent, but functions that directly access the keyboard and screen will not work. These include system-specific functions that determine whether a key has been pressed and if so, which.

The IDLE code running in the execution process adds frames to the call stack that would not be there otherwise. IDLE wraps `sys.getrecursionlimit` and `sys.setrecursionlimit` to reduce the effect of the additional stack frames.

When user code raises SystemExit either directly or by calling sys.exit, IDLE returns to a Shell prompt instead of exiting.

User output in Shell

When a program outputs text, the result is determined by the corresponding output device. When IDLE executes user code, `sys.stdout` and `sys.stderr` are connected to the display area of IDLE’s Shell. Some of its features are inherited from the underlying Tk Text widget. Others are programmed additions. Where it matters, Shell is designed for development rather than production runs.

For instance, Shell never throws away output. A program that sends unlimited output to Shell will eventually fill memory, resulting in a memory error. In contrast, some system text windows only keep the last n lines of output. A Windows console, for instance, keeps a user-settable 1 to 9999 lines, with 300 the default.

A Tk Text widget, and hence IDLE’s Shell, displays characters (codepoints) in the BMP (Basic Multilingual Plane) subset of Unicode. Which characters are displayed with a proper glyph and which with a replacement box depends on the operating system and installed fonts. Tab characters cause the following text to begin after the next tab stop. (They occur every 8 ‘characters’). Newline characters cause following text to appear on a new line. Other control characters are ignored or displayed as a space, box, or something else, depending on the operating system and font. (Moving the text cursor through such output with arrow keys may exhibit some surprising spacing behavior.)

```
>>> s = 'a\tb\a<\x02><\r>\bc\nd'  # Enter 22 chars.
>>> len(s)
14
>>> s  # Display repr(s)
'a\tb\x07<\x02><\r>\x08c\nd'
>>> print(s, end='')  # Display s as is.
# Result varies by OS and font.  Try it.
```

The `repr` function is used for interactive echo of expression values. It returns an altered version of the input string in which control codes, some BMP codepoints, and all non-BMP codepoints are replaced with escape codes. As demonstrated above, it allows one to identify the characters in a string, regardless of how they are displayed.

Normal and error output are generally kept separate (on separate lines) from code input and each other. They each get different highlight colors.

For SyntaxError tracebacks, the normal ‘^’ marking where the error was detected is replaced by coloring the text with an error highlight. When code run from a file causes other exceptions, one may right click on a traceback line to jump to the corresponding line in an IDLE editor. The file will be opened if necessary.

Shell has a special facility for squeezing output lines down to a ‘Squeezed text’ label. This is done automatically for output over N lines (N = 50 by default). N can be changed in the PyShell section of the General page of the Settings dialog. Output with fewer lines can be squeezed by right clicking on the output. This can be useful lines long enough to slow down scrolling.

Squeezed output is expanded in place by double-clicking the label. It can also be sent to the clipboard or a separate view window by right-clicking the label.

Developing tkinter applications

IDLE is intentionally different from standard Python in order to facilitate development of tkinter programs. Enter `import tkinter as tk; root = tk.Tk()` in standard Python and nothing appears. Enter the same in IDLE and a tk window appears. In standard Python, one must also enter `root.update()` to see the window. IDLE does the equivalent in the background, about 20 times a second, which is about every 50 milliseconds. Next enter `b = tk.Button(root, text='button'); b.pack()`. Again, nothing visibly changes in standard Python until one enters `root.update()`.

Most tkinter programs run `root.mainloop()`, which usually does not return until the tk app is destroyed. If the program is run with `python -i` or from an IDLE editor, a `>>>` shell prompt does not appear until `mainloop()` returns, at which time there is nothing left to interact with.

When running a tkinter program from an IDLE editor, one can comment out the mainloop call. One then gets a shell prompt immediately and can interact with the live application. One just has to remember to re-enable the mainloop call when running in standard Python.

Running without a subprocess

By default, IDLE executes user code in a separate subprocess via a socket, which uses the internal loopback interface. This connection is not externally visible and no data is sent to or received from the internet. If firewall software complains anyway, you can ignore it.

If the attempt to make the socket connection fails, Idle will notify you. Such failures are sometimes transient, but if persistent, the problem may be either a firewall blocking the connection or misconfiguration of a particular system. Until the problem is fixed, one can run Idle with the -n command line switch.

If IDLE is started with the -n command line switch it will run in a single process and will not create the subprocess which runs the RPC Python execution server. This can be useful if Python cannot create the subprocess or the RPC socket interface on your platform. However, in this mode user code is not isolated from IDLE itself. Also, the environment is not restarted when Run/Run Module (F5) is selected. If your code has been modified, you must reload() the affected modules and re-import any specific items (e.g. from foo import baz) if the changes are to take effect. For these reasons, it is preferable to run IDLE with the default subprocess if at all possible.

Deprecated since version 3.4.

Help and Preferences

Help sources

Help menu entry “IDLE Help” displays a formatted html version of the IDLE chapter of the Library Reference. The result, in a read-only tkinter text window, is close to what one sees in a web browser. Navigate through the text with a mousewheel, the scrollbar, or up and down arrow keys held down. Or click the TOC (Table of Contents) button and select a section header in the opened box.

Help menu entry “Python Docs” opens the extensive sources of help, including tutorials, available at `docs.python.org/x.y`, where ‘x.y’ is the currently running Python version. If your system has an off-line copy of the docs (this may be an installation option), that will be opened instead.

Selected URLs can be added or removed from the help menu at any time using the General tab of the Configure IDLE dialog.

Setting preferences

The font preferences, highlighting, keys, and general preferences can be changed via Configure IDLE on the Option menu. Non-default user settings are saved in a `.idlerc` directory in the user’s home directory. Problems caused by bad user configuration files are solved by editing or deleting one or more of the files in `.idlerc`.

On the Font tab, see the text sample for the effect of font face and size on multiple characters in multiple languages. Edit the sample to add other characters of personal interest. Use the sample to select monospaced fonts. If particular characters have problems in Shell or an editor, add them to the top of the sample and try changing first size and then font.

On the Highlights and Keys tab, select a built-in or custom color theme and key set. To use a newer built-in color theme or key set with older IDLEs, save it as a new custom theme or key set and it well be accessible to older IDLEs.

IDLE on macOS

Under System Preferences: Dock, one can set “Prefer tabs when opening documents” to “Always”. This setting is not compatible with the tk/tkinter GUI framework used by IDLE, and it breaks a few IDLE features.

Extensions

IDLE contains an extension facility. Preferences for extensions can be changed with the Extensions tab of the preferences dialog. See the beginning of config-extensions.def in the idlelib directory for further information. The only current default extension is zzdummy, an example also used for testing.

idlelib — implementation of IDLE application

Source code: Lib/idlelib
[https://github.com/python/cpython/tree/3.14/Lib/idlelib]

The Lib/idlelib package implements the IDLE application. See the rest of this page for how to use IDLE.

The files in idlelib are described in idlelib/README.txt. Access it either in idlelib or click Help => About IDLE on the IDLE menu. This file also maps IDLE menu items to the code that implements the item. Except for files listed under ‘Startup’, the idlelib code is ‘private’ in sense that feature changes can be backported (see **PEP 434** [https://peps.python.org/pep-0434/]).

`turtle` — Turtle graphics

Source code: Lib/turtle.py
[https://github.com/python/cpython/tree/3.14/Lib/turtle.py]

Introduction

Turtle graphics is an implementation of the popular geometric drawing tools introduced in Logo [https://en.wikipedia.org/wiki/Turtle_(robot)], developed by Wally Feurzeig, Seymour Papert and Cynthia Solomon in 1967.

This is an optional module. If it is missing from your copy of CPython, look for documentation from your distributor (that is, whoever provided Python to you). If you are the distributor, see Requirements for optional modules.

Get started

Imagine a robotic turtle starting at (0, 0) in the x-y plane. After an `import turtle`, give it the command `turtle.forward(15)`, and it moves (on-screen!) 15 pixels in the direction it is facing, drawing a line as it moves. Give it the command `turtle.right(25)`, and it rotates in-place 25 degrees clockwise.

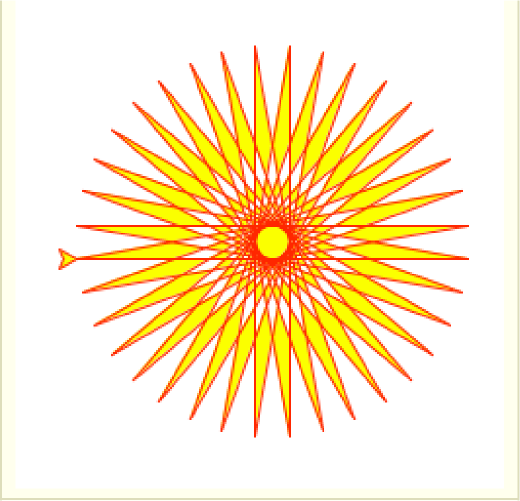
In Python, turtle graphics provides a representation of a physical “turtle” (a little robot with a pen) that draws on a sheet of paper on the floor.

It’s an effective and well-proven way for learners to encounter

Turtle star

Turtle can draw intricate shapes using programs that repeat simple moves.

programming concepts and interaction with software, as it provides instant, visible feedback. It also provides convenient access to graphical output in general.



Turtle drawing was originally created as an educational tool, to be used by teachers in the classroom. For the programmer who needs to produce some graphical output it can be a way to do that without the overhead of introducing more complex or external libraries into their work.

Tutorial

New users should start here. In this tutorial we’ll explore some of the basics of turtle drawing.

Starting a turtle environment

In a Python shell, import all the objects of the `turtle` module:

```
from turtle import *
```

If you run into a `No module named '_tkinter'` error, you’ll have to install the [`Tk interface package`] on your system.

Basic drawing

Send the turtle forward 100 steps:

```
forward(100)
```

You should see (most likely, in a new window on your display) a line drawn by the turtle, heading East. Change the direction of the turtle, so that it turns 120 degrees left (anti-clockwise):

```
left(120)
```

Let’s continue by drawing a triangle:

```
forward(100)
left(120)
forward(100)
```

Notice how the turtle, represented by an arrow, points in different directions as you steer it.

Experiment with those commands, and also with `backward()` and `right()`.

Pen control

Try changing the color - for example, `color('blue')` - and width of the line - for example, `width(3)` - and then drawing again.

You can also move the turtle around without drawing, by lifting up the pen: `up()` before moving. To start drawing again, use `down()`.

The turtle’s position

Send your turtle back to its starting-point (useful if it has disappeared off-screen):

```
home()
```

The home position is at the center of the turtle’s screen. If you ever need to know them, get the turtle’s x-y coordinates with:

```
pos()
```

Home is at `(0, 0)`.

And after a while, it will probably help to clear the window so we can start anew:

```
clearscreen()
```

Making algorithmic patterns

Using loops, it’s possible to build up geometric patterns:

```
for steps in range(100):
    for c in ('blue', 'red', 'green'):
        color(c)
        forward(steps)
        right(30)
```

- which of course, are limited only by the imagination!

Let’s draw the star shape at the top of this page. We want red lines, filled in with yellow:

```
color('red')
fillcolor('yellow')
```

Just as `up()` and `down()` determine whether lines will be drawn, filling can be turned on and off:

```
begin_fill()
```

Next we’ll create a loop:

```
while True:
    forward(200)
    left(170)
    if abs(pos()) < 1:
        break
```

`abs(pos()) < 1` is a good way to know when the turtle is back at its home position.

Finally, complete the filling:

```
end_fill()
```

(Note that filling only actually takes place when you give the `end_fill()` command.)

How to…

This section covers some typical turtle use-cases and approaches.

Get started as quickly as possible

One of the joys of turtle graphics is the immediate, visual feedback that's available from simple commands - it's an excellent way to introduce children to programming ideas, with a minimum of overhead (not just children, of course).

The turtle module makes this possible by exposing all its basic functionality as functions, available with `from turtle import *`. The turtle graphics tutorial covers this approach.

It's worth noting that many of the turtle commands also have even more terse equivalents, such as `fd()` for **`forward()`**. These are especially useful when working with learners for whom typing is not a skill.

> You'll need to have the **`Tk interface package`** installed on your system for turtle graphics to work. Be warned that this is not always straightforward, so check this in advance if you're planning to use turtle graphics with a learner.

Automatically begin and end filling

Starting with Python 3.14, you can use the `fill()` context manager instead of `begin_fill()` and `end_fill()` to automatically begin and end fill. Here is an example:

```
with fill():
    for i in range(4):
        forward(100)
        right(90)

forward(200)
```

The code above is equivalent to:

```
begin_fill()
for i in range(4):
    forward(100)
    right(90)
end_fill()

forward(200)
```

Use the `turtle` module namespace

Using `from turtle import *` is convenient - but be warned that it imports a rather large collection of objects, and if you’re doing anything but turtle graphics you run the risk of a name conflict (this becomes even more an issue if you’re using turtle graphics in a script where other modules might be imported).

The solution is to use `import turtle` - `fd()` becomes `turtle.fd()`, `width()` becomes `turtle.width()` and so on. (If typing “turtle” over and over again becomes tedious, use for example `import turtle as t` instead.)

Use turtle graphics in a script

It’s recommended to use the `turtle` module namespace as described immediately above, for example:

```
import turtle as t
from random import random

for i in range(100):
    steps = int(random() * 100)
    angle = int(random() * 360)
    t.right(angle)
    t.fd(steps)
```

Another step is also required though - as soon as the script ends, Python will also close the turtle’s window. Add:

```
t.mainloop()
```

to the end of the script. The script will now wait to be dismissed and will not exit until it is terminated, for example by closing the turtle graphics window.

Use object-oriented turtle graphics

See also

Explanation of the object-oriented interface

Other than for very basic introductory purposes, or for trying things out as quickly as possible, it’s more usual and much more powerful to use the object-oriented approach to turtle graphics. For example, this allows multiple turtles on screen at once.

In this approach, the various turtle commands are methods of objects (mostly of `Turtle` objects). You *can* use the object-oriented approach in the shell, but it would be more typical in a Python script.

The example above then becomes:

```
from turtle import Turtle
from random import random

t = Turtle()
for i in range(100):
    steps = int(random() * 100)
    angle = int(random() * 360)
    t.right(angle)
    t.fd(steps)

t.screen.mainloop()
```

Note the last line. `t.screen` is an instance of the **`Screen`** that a Turtle instance exists on; it’s created automatically along with the turtle.

The turtle’s screen can be customised, for example:

```
t.screen.title('Object-oriented turtle demo')
t.screen.bgcolor("orange")
```

Turtle graphics reference

Note

In the following documentation the argument list for functions is given. Methods, of course, have the additional first argument *self* which is omitted here.

Turtle methods

Turtle motion

Move and draw

`forward()` | **`fd()`**

- `backward()` | `bk()` | `back()`
- `right()` | `rt()`
- `left()` | `lt()`
- `goto()` | `setpos()` | `setposition()`
- `teleport()`
- `setx()`
- `sety()`
- `setheading()` | `seth()`
- `home()`
- `circle()`
- `dot()`
- `stamp()`
- `clearstamp()`
- `clearstamps()`
- `undo()`
- `speed()`

Tell Turtle’s state

- `position()` | `pos()`
- `towards()`
- `xcor()`
- `ycor()`
- `heading()`
- `distance()`

Setting and measurement

- `degrees()`
- `radians()`

Pen control

Drawing state

- `pendown()` | `pd()` | `down()`
- `penup()` | `pu()` | `up()`
- `pensize()` | `width()`
- `pen()`

- `isdown()`

Color control

- `color()`
- `pencolor()`
- `fillcolor()`

Filling

- `filling()`
- `fill()`
- `begin_fill()`
- `end_fill()`

More drawing control

- `reset()`
- `clear()`
- `write()`

Turtle state

Visibility

- `showturtle()` | `st()`
- `hideturtle()` | `ht()`
- `isvisible()`

Appearance

- `shape()`
- `resizemode()`
- `shapesize()` | `turtlesize()`
- `shearfactor()`
- `tiltangle()`
- `tilt()`
- `shapetransform()`
- `get_shapepoly()`

Using events

`onclick()`
`onrelease()`
`ondrag()`

Special Turtle methods

`poly()`
`begin_poly()`
`end_poly()`
`get_poly()`
`clone()`
`getturtle()` | `getpen()`
`getscreen()`
`setundobuffer()`
`undobufferentries()`

Methods of TurtleScreen/Screen

Window control

`bgcolor()`
`bgpic()`
`clearscreen()`
`resetscreen()`
`screensize()`
`setworldcoordinates()`

Animation control

`no_animation()`
`delay()`
`tracer()`
`update()`

Using screen events

- `listen()`
- `onkey()` | `onkeyrelease()`
- `onkeypress()`
- `onclick()` | `onscreenclick()`
- `ontimer()`
- `mainloop()` | `done()`

Settings and special methods

- `mode()`
- `colormode()`
- `getcanvas()`
- `getshapes()`
- `register_shape()` | `addshape()`
- `turtles()`
- `window_height()`
- `window_width()`

Input methods

- `textinput()`
- `numinput()`

Methods specific to Screen

- `bye()`
- `exitonclick()`
- `save()`
- `setup()`
- `title()`

Methods of RawTurtle/Turtle and corresponding functions

Most of the examples in this section refer to a Turtle instance called `turtle`.

Turtle motion

turtle.forward(*distance*)
turtle.fd(*distance*)

Parameters:: **distance** – a number (integer or float)

Move the turtle forward by the specified *distance*, in the direction the turtle is headed.

```
>>> turtle.position()
(0.00,0.00)
>>> turtle.forward(25)
>>> turtle.position()
(25.00,0.00)
>>> turtle.forward(-75)
>>> turtle.position()
(-50.00,0.00)
```

turtle.back(*distance*)
turtle.bk(*distance*)
turtle.backward(*distance*)

Parameters:: **distance** – a number

Move the turtle backward by *distance*, opposite to the direction the turtle is headed. Do not change the turtle's heading.

```
>>> turtle.position()
(0.00,0.00)
>>> turtle.backward(30)
>>> turtle.position()
(-30.00,0.00)
```

turtle.right(*angle*)
turtle.rt(*angle*)

Parameters:: **angle** – a number (integer or float)

Turn turtle right by *angle* units. (Units are by default degrees, but can be set via the `degrees()` and `radians()`

functions.) Angle orientation depends on the turtle mode, see `mode()`.

```
>>> turtle.heading()
22.0
>>> turtle.right(45)
>>> turtle.heading()
337.0
```

turtle.left(*angle*)
turtle.lt(*angle*)

Parameters:: **angle** – a number (integer or float)

Turn turtle left by *angle* units. (Units are by default degrees, but can be set via the `degrees()` and `radians()` functions.) Angle orientation depends on the turtle mode, see `mode()`.

```
>>> turtle.heading()
22.0
>>> turtle.left(45)
>>> turtle.heading()
67.0
```

turtle.goto(*x*, *y=None*)
turtle.setpos(*x*, *y=None*)
turtle.setposition(*x*, *y=None*)

Parameters::
- **x** – a number or a pair/vector of number
- **y** – a number or `None`

If *y* is `None`, *x* must be a pair of coordinates or a `Vec2D` (e.g. as returned by `pos()`).

Move turtle to an absolute position. If the pen is down, draw line. Do not change the turtle’s orientation.

```
>>> tp = turtle.pos()
>>> tp
(0.00,0.00)
```

```
>>> turtle.setpos(60,30)
>>> turtle.pos()
(60.00,30.00)
>>> turtle.setpos((20,80))
>>> turtle.pos()
(20.00,80.00)
>>> turtle.setpos(tp)
>>> turtle.pos()
(0.00,0.00)
```

turtle.teleport(*x*, *y=None*, *, *fill_gap=False*)

Parameters::
- **x** – a number or `None`
- **y** – a number or `None`
- **fill_gap** – a boolean

Move turtle to an absolute position. Unlike goto(x, y), a line will not be drawn. The turtle's orientation does not change. If currently filling, the polygon(s) teleported from will be filled after leaving, and filling will begin again after teleporting. This can be disabled with fill_gap=True, which makes the imaginary line traveled during teleporting act as a fill barrier like in goto(x, y).

```
>>> tp = turtle.pos()
>>> tp
(0.00,0.00)
>>> turtle.teleport(60)
>>> turtle.pos()
(60.00,0.00)
>>> turtle.teleport(y=10)
>>> turtle.pos()
(60.00,10.00)
>>> turtle.teleport(20, 30)
>>> turtle.pos()
(20.00,30.00)
```

Added in version 3.12.

turtle.setx(*x*)

Parameters:: **x** – a number (integer or float)

Set the turtle’s first coordinate to *x*, leave second coordinate unchanged.

```
>>> turtle.position()
(0.00,240.00)
>>> turtle.setx(10)
>>> turtle.position()
(10.00,240.00)
```

turtle.sety(*y*)

Parameters:: **y** – a number (integer or float)

Set the turtle’s second coordinate to *y*, leave first coordinate unchanged.

```
>>> turtle.position()
(0.00,40.00)
>>> turtle.sety(-10)
>>> turtle.position()
(0.00,-10.00)
```

turtle.setheading(*to_angle*)
turtle.seth(*to_angle*)

Parameters:: **to_angle** – a number (integer or float)

Set the orientation of the turtle to *to_angle*. Here are some common directions in degrees:

standard mode	logo mode
0 - east	0 - north
90 - north	90 - east
180 - west	180 - south
270 - south	270 - west

```
>>> turtle.setheading(90)
>>> turtle.heading()
90.0
```

turtle.home()

Move turtle to the origin – coordinates (0,0) – and set its heading to its start-orientation (which depends on the mode, see `mode()`).

```
>>> turtle.heading()
90.0
>>> turtle.position()
(0.00,-10.00)
>>> turtle.home()
>>> turtle.position()
(0.00,0.00)
>>> turtle.heading()
0.0
```

turtle.circle(*radius*, *extent=None*, *steps=None*)

Parameters::

- **radius** – a number
- **extent** – a number (or `None`)
- **steps** – an integer (or `None`)

Draw a circle with given *radius*. The center is *radius* units left of the turtle; *extent* – an angle – determines which part of the circle is drawn. If *extent* is not given, draw the entire circle. If *extent* is not a full circle, one endpoint of the arc is the current pen position. Draw the arc in counterclockwise direction if *radius* is positive, otherwise in clockwise direction. Finally the direction of the turtle is changed by the amount of *extent*.

As the circle is approximated by an inscribed regular polygon, *steps* determines the number of steps to use. If not given, it will be calculated automatically. May be used to draw regular polygons.

```
>>> turtle.home()
>>> turtle.position()
(0.00,0.00)
>>> turtle.heading()
0.0
>>> turtle.circle(50)
>>> turtle.position()
(-0.00,0.00)
>>> turtle.heading()
0.0
>>> turtle.circle(120, 180)  # draw a semicircle
>>> turtle.position()
(0.00,240.00)
>>> turtle.heading()
180.0
```

turtle.dot()
turtle.dot(*size*)
turtle.dot(*color*, */*)
turtle.dot(*size*, *color*, */*)
turtle.dot(*size*, *r*, *g*, *b*, */*)

Parameters::

- **size** – an integer >= 1 (if given)
- **color** – a colorstring or a numeric color tuple

Draw a circular dot with diameter *size*, using *color*. If *size* is not given, the maximum of `pensize+4` and `2*pensize` is used.

```
>>> turtle.home()
>>> turtle.dot()
>>> turtle.fd(50); turtle.dot(20, "blue"); turtle.fd(50)
>>> turtle.position()
(100.00,-0.00)
>>> turtle.heading()
0.0
```

turtle.stamp()

Stamp a copy of the turtle shape onto the canvas at the current turtle position. Return a stamp_id for that stamp, which can be used to delete it by calling `clearstamp(stamp_id)`.

```
>>> turtle.color("blue")
>>> stamp_id = turtle.stamp()
>>> turtle.fd(50)
```

turtle.clearstamp(*stampid*)

Parameters:: **stampid** – an integer, must be return value of previous `stamp()` call

Delete stamp with given *stampid*.

```
>>> turtle.position()
(150.00,-0.00)
>>> turtle.color("blue")
>>> astamp = turtle.stamp()
>>> turtle.fd(50)
>>> turtle.position()
(200.00,-0.00)
>>> turtle.clearstamp(astamp)
>>> turtle.position()
(200.00,-0.00)
```

turtle.clearstamps(*n=None*)

Parameters:: **n** – an integer (or `None`)

Delete all or first/last *n* of turtle’s stamps. If *n* is `None`, delete all stamps, if *n* > 0 delete first *n* stamps, else if *n* < 0 delete last *n* stamps.

```
>>> for i in range(8):
...     unused_stamp_id = turtle.stamp()
...     turtle.fd(30)
>>> turtle.clearstamps(2)
>>> turtle.clearstamps(-2)
>>> turtle.clearstamps()
```

turtle.undo()

Undo (repeatedly) the last turtle action(s). Number of available undo actions is determined by the size of the undobuffer.

```
>>> for i in range(4):
...     turtle.fd(50); turtle.lt(80)
...
>>> for i in range(8):
...     turtle.undo()
```

turtle.speed(*speed=None*)

Parameters:: **speed** – an integer in the range 0..10 or a speedstring (see below)

Set the turtle’s speed to an integer value in the range 0..10. If no argument is given, return current speed.

If input is a number greater than 10 or smaller than 0.5, speed is set to 0. Speedstrings are mapped to speedvalues as follows:

- “fastest”: 0
- “fast”: 10
- “normal”: 6
- “slow”: 3
- “slowest”: 1

Speeds from 1 to 10 enforce increasingly faster animation of line drawing and turtle turning.

Attention: *speed* = 0 means that *no* animation takes place. forward/back makes turtle jump and likewise left/right make the turtle turn instantly.

```
>>> turtle.speed()
3
>>> turtle.speed('normal')
>>> turtle.speed()
6
>>> turtle.speed(9)
>>> turtle.speed()
9
```

Tell Turtle’s state

turtle.position()
turtle.pos()

Return the turtle’s current location (x,y) (as a `Vec2D` vector).

```
>>> turtle.pos()
(440.00,-0.00)
```

turtle.towards(*x*, *y=None*)

Parameters::
- **x** – a number or a pair/vector of number or a turtle instance
- **y** – a number if *x* is a number, else `None`

Return the angle between the line from turtle position to position specified by (x,y), the vector or the other turtle. This depends on the turtle’s start orientation which depends on the mode - “standard”/”world” or “logo”.

```
>>> turtle.goto(10, 10)
>>> turtle.towards(0,0)
225.0
```

turtle.xcor()

Return the turtle’s x coordinate.

```
>>> turtle.home()
>>> turtle.left(50)
>>> turtle.forward(100)
>>> turtle.pos()
(64.28,76.60)
>>> print(round(turtle.xcor(), 5))
64.27876
```

turtle.ycor()

Return the turtle’s y coordinate.

```
>>> turtle.home()
>>> turtle.left(60)
>>> turtle.forward(100)
>>> print(turtle.pos())
(50.00,86.60)
>>> print(round(turtle.ycor(), 5))
86.60254
```

turtle.heading()

Return the turtle’s current heading (value depends on the turtle mode, see `mode()`).

```
>>> turtle.home()
>>> turtle.left(67)
>>> turtle.heading()
67.0
```

turtle.distance(*x*, *y=None*)

Parameters::
- **x** – a number or a pair/vector of number or a turtle instance
- **y** – a number if *x* is a number, else `None`

Return the distance from the turtle to (x,y), the given vector, or the given other turtle, in turtle step units.

```
>>> turtle.home()
>>> turtle.distance(30,40)
50.0
>>> turtle.distance((30,40))
50.0
>>> joe = Turtle()
>>> joe.forward(77)
>>> turtle.distance(joe)
77.0
```

Settings for measurement

turtle.degrees(*fullcircle=360.0*)

Parameters:: **fullcircle** – a number

Set angle measurement units, i.e. set number of “degrees” for a full circle. Default value is 360 degrees.

```
>>> turtle.home()
>>> turtle.left(90)
>>> turtle.heading()
90.0

>>> # Change angle measurement unit to grad (also known as
gon,
>>> # grade, or gradian and equals 1/100-th of the right
angle.)
>>> turtle.degrees(400.0)
>>> turtle.heading()
100.0
>>> turtle.degrees(360)
>>> turtle.heading()
90.0
```

turtle.radians()

Set the angle measurement units to radians. Equivalent to `degrees(2*math.pi)`.

```
>>> turtle.home()
>>> turtle.left(90)
>>> turtle.heading()
90.0
>>> turtle.radians()
>>> turtle.heading()
1.5707963267948966
```

Pen control

Drawing state

turtle.pendown()
turtle.pd()
turtle.down()

Pull the pen down – drawing when moving.

turtle.penup()
turtle.pu()
turtle.up()

Pull the pen up – no drawing when moving.

turtle.pensize(*width=None*)
turtle.width(*width=None*)

Parameters:: **width** – a positive number

Set the line thickness to *width* or return it. If resizemode is set to “auto” and turtleshape is a polygon, that polygon is drawn with the same line thickness. If no argument is given, the current pensize is returned.

```
>>> turtle.pensize()
1
>>> turtle.pensize(10)   # from here on lines of width 10 are
drawn
```

turtle.pen(*pen=None*, ***pendict*)

Parameters::
- **pen** – a dictionary with some or all of the below listed keys
- **pendict** – one or more keyword-arguments with the below listed keys as keywords

Return or set the pen’s attributes in a “pen-dictionary” with the following key/value pairs:

- “shown”: True/False
- “pendown”: True/False
- “pencolor”: color-string or color-tuple
- “fillcolor”: color-string or color-tuple
- “pensize”: positive number
- “speed”: number in range 0..10
- “resizemode”: “auto” or “user” or “noresize”
- “stretchfactor”: (positive number, positive number)

- “outline”: positive number
- “tilt”: number

This dictionary can be used as argument for a subsequent call to `pen()` to restore the former pen-state. Moreover one or more of these attributes can be provided as keyword-arguments. This can be used to set several pen attributes in one statement.

```
>>> turtle.pen(fillcolor="black", pencolor="red", pensize=10)
>>> sorted(turtle.pen().items())
[('fillcolor', 'black'), ('outline', 1), ('pencolor', 'red'),
 ('pendown', True), ('pensize', 10), ('resizemode',
'noresize'),
 ('shearfactor', 0.0), ('shown', True), ('speed', 9),
 ('stretchfactor', (1.0, 1.0)), ('tilt', 0.0)]
>>> penstate=turtle.pen()
>>> turtle.color("yellow", "")
>>> turtle.penup()
>>> sorted(turtle.pen().items())[:3]
[('fillcolor', ''), ('outline', 1), ('pencolor', 'yellow')]
>>> turtle.pen(penstate, fillcolor="green")
>>> sorted(turtle.pen().items())[:3]
[('fillcolor', 'green'), ('outline', 1), ('pencolor', 'red')]
```

turtle.isdown()

Return `True` if pen is down, `False` if it’s up.

```
>>> turtle.penup()
>>> turtle.isdown()
False
>>> turtle.pendown()
>>> turtle.isdown()
True
```

Color control

turtle.pencolor()
turtle.pencolor(*color*, /)
turtle.pencolor(*r*, *g*, *b*, /)

Return or set the pencolor.

Four input formats are allowed:

`pencolor()`
: Return the current pencolor as color specification string or as a tuple (see example). May be used as input to another color/pencolor/fillcolor/bgcolor call.

`pencolor(colorstring)`
: Set pencolor to *colorstring*, which is a Tk color specification string, such as `"red"`, `"yellow"`, or `"#33cc8c"`.

`pencolor((r, g, b))`
: Set pencolor to the RGB color represented by the tuple of *r*, *g*, and *b*. Each of *r*, *g*, and *b* must be in the range 0..colormode, where colormode is either 1.0 or 255 (see **`colormode()`**).

`pencolor(r, g, b)`
: Set pencolor to the RGB color represented by *r*, *g*, and *b*. Each of *r*, *g*, and *b* must be in the range 0..colormode.

If turtleshape is a polygon, the outline of that polygon is drawn with the newly set pencolor.

```
>>> colormode()
1.0
>>> turtle.pencolor()
'red'
>>> turtle.pencolor("brown")
>>> turtle.pencolor()
'brown'
>>> tup = (0.2, 0.8, 0.55)
>>> turtle.pencolor(tup)
>>> turtle.pencolor()
(0.2, 0.8, 0.5490196078431373)
>>> colormode(255)
>>> turtle.pencolor()
```

```
(51.0, 204.0, 140.0)
>>> turtle.pencolor('#32c18f')
>>> turtle.pencolor()
(50.0, 193.0, 143.0)
```

turtle.fillcolor()
turtle.fillcolor(*color*, */*)
turtle.fillcolor(*r*, *g*, *b*, */*)

Return or set the fillcolor.

Four input formats are allowed:

`fillcolor()`

Return the current fillcolor as color specification string, possibly in tuple format (see example). May be used as input to another color/pencolor/fillcolor/bgcolor call.

`fillcolor(colorstring)`

Set fillcolor to *colorstring*, which is a Tk color specification string, such as `"red"`, `"yellow"`, or `"#33cc8c"`.

`fillcolor((r, g, b))`

Set fillcolor to the RGB color represented by the tuple of *r*, *g*, and *b*. Each of *r*, *g*, and *b* must be in the range 0..colormode, where colormode is either 1.0 or 255 (see **`colormode()`**).

`fillcolor(r, g, b)`

Set fillcolor to the RGB color represented by *r*, *g*, and *b*. Each of *r*, *g*, and *b* must be in the range 0..colormode.

If turtleshape is a polygon, the interior of that polygon is drawn with the newly set fillcolor.

```
>>> turtle.fillcolor("violet")
>>> turtle.fillcolor()
'violet'
>>> turtle.pencolor()
(50.0, 193.0, 143.0)
```

```
>>> turtle.fillcolor((50, 193, 143))  # Integers, not floats
>>> turtle.fillcolor()
(50.0, 193.0, 143.0)
>>> turtle.fillcolor('#ffffff')
>>> turtle.fillcolor()
(255.0, 255.0, 255.0)
```

turtle.color()
turtle.color(*color*, */*)
turtle.color(*r*, *g*, *b*, */*)
turtle.color(*pencolor*, *fillcolor*, */*)

Return or set pencolor and fillcolor.

Several input formats are allowed. They use 0 to 3 arguments as follows:

`color()`

Return the current pencolor and the current fillcolor as a pair of color specification strings or tuples as returned by `pencolor()` and `fillcolor()`.

`color(colorstring)`, `color((r,g,b))`, `color(r,g,b)`

Inputs as in `pencolor()`, set both, fillcolor and pencolor, to the given value.

`color(colorstring1, colorstring2)`, `color((r1,g1,b1), (r2,g2,b2))`

Equivalent to `pencolor(colorstring1)` and `fillcolor(colorstring2)` and analogously if the other input format is used.

If turtleshape is a polygon, outline and interior of that polygon is drawn with the newly set colors.

```
>>> turtle.color("red", "green")
>>> turtle.color()
('red', 'green')
>>> color("#285078", "#a0c8f0")
>>> color()
((40.0, 80.0, 120.0), (160.0, 200.0, 240.0))
```

See also: Screen method `colormode()`.

Filling

turtle.filling()

Return fillstate (`True` if filling, `False` else).

```
>>> turtle.begin_fill()
>>> if turtle.filling():
...     turtle.pensize(5)
... else:
...     turtle.pensize(3)
```

turtle.fill()

Fill the shape drawn in the `with turtle.fill():` block.

```
>>> turtle.color("black", "red")
>>> with turtle.fill():
...     turtle.circle(80)
```

Using **`fill()`** is equivalent to adding the `begin_fill()` before the fill-block and `end_fill()` after the fill-block:

```
>>> turtle.color("black", "red")
>>> turtle.begin_fill()
>>> turtle.circle(80)
>>> turtle.end_fill()
```

Added in version 3.14.

turtle.begin_fill()

To be called just before drawing a shape to be filled.

turtle.end_fill()

Fill the shape drawn after the last call to `begin_fill()`.

Whether or not overlap regions for self-intersecting polygons or multiple shapes are filled depends on the operating system graphics, type of overlap, and number of overlaps. For example, the Turtle star above may be either all yellow or have some white regions.

```
>>> turtle.color("black", "red")
>>> turtle.begin_fill()
>>> turtle.circle(80)
>>> turtle.end_fill()
```

More drawing control

turtle.reset()

Delete the turtle’s drawings from the screen, re-center the turtle and set variables to the default values.

```
>>> turtle.goto(0,-22)
>>> turtle.left(100)
>>> turtle.position()
(0.00,-22.00)
>>> turtle.heading()
100.0
>>> turtle.reset()
>>> turtle.position()
(0.00,0.00)
>>> turtle.heading()
0.0
```

turtle.clear()

Delete the turtle’s drawings from the screen. Do not move turtle. State and position of the turtle as well as drawings of other turtles are not affected.

turtle.write(*arg*, *move=False*, *align='left'*, *font=('Arial', 8, 'normal')*)

Parameters::

- **arg** – object to be written to the TurtleScreen
- **move** – True/False
- **align** – one of the strings “left”, “center or right”
- **font** – a triple (fontname, fontsize, fonttype)

Write text - the string representation of *arg* - at the current turtle position according to *align* (“left”, “center” or “right”) and with the given font. If *move* is true, the pen is moved to the bottom-right corner of the text. By default, *move* is `False`.

```
>>> turtle.write("Home = ", True, align="center")
>>> turtle.write((0,0), True)
```

Turtle state

Visibility

turtle.hideturtle()
turtle.ht()

Make the turtle invisible. It’s a good idea to do this while you’re in the middle of doing some complex drawing, because hiding the turtle speeds up the drawing observably.

```
>>> turtle.hideturtle()
```

turtle.showturtle()
turtle.st()

Make the turtle visible.

```
>>> turtle.showturtle()
```

turtle.isvisible()

Return `True` if the Turtle is shown, `False` if it’s hidden.

```
>>> turtle.hideturtle()
>>> turtle.isvisible()
False
>>> turtle.showturtle()
>>> turtle.isvisible()
True
```

Appearance

turtle.shape(*name=None*)

Parameters:: **name** – a string which is a valid shapename

Set turtle shape to shape with given *name* or, if name is not given, return name of current shape. Shape with *name* must exist in the TurtleScreen’s shape dictionary. Initially there are the following polygon shapes: “arrow”, “turtle”, “circle”, “square”, “triangle”, “classic”. To learn about how to deal with shapes see Screen method `register_shape()`.

```
>>> turtle.shape()
'classic'
>>> turtle.shape("turtle")
>>> turtle.shape()
'turtle'
```

turtle.resizemode(*rmode=None*)

Parameters:: **rmode** – one of the strings “auto”, “user”, “noresize”

Set resizemode to one of the values: “auto”, “user”, “noresize”. If *rmode* is not given, return current resizemode. Different resizemodes have the following effects:

- “auto”: adapts the appearance of the turtle corresponding to the value of pensize.

- “user”: adapts the appearance of the turtle according to the values of stretchfactor and outlinewidth (outline), which are set by `shapesize()`.
- “noresize”: no adaption of the turtle’s appearance takes place.

`resizemode("user")` is called by `shapesize()` when used with arguments.

```
>>> turtle.resizemode()
'noresize'
>>> turtle.resizemode("auto")
>>> turtle.resizemode()
'auto'
```

turtle.shapesize(*stretch_wid=None*, *stretch_len=None*, *outline=None*)
turtle.turtlesize(*stretch_wid=None*, *stretch_len=None*, *outline=None*)

Parameters::

- **stretch_wid** – positive number
- **stretch_len** – positive number
- **outline** – positive number

Return or set the pen’s attributes x/y-stretchfactors and/or outline. Set resizemode to “user”. If and only if resizemode is set to “user”, the turtle will be displayed stretched according to its stretchfactors: *stretch_wid* is stretchfactor perpendicular to its orientation, *stretch_len* is stretchfactor in direction of its orientation, *outline* determines the width of the shape’s outline.

```
>>> turtle.shapesize()
(1.0, 1.0, 1)
>>> turtle.resizemode("user")
>>> turtle.shapesize(5, 5, 12)
>>> turtle.shapesize()
(5, 5, 12)
>>> turtle.shapesize(outline=8)
>>> turtle.shapesize()
(5, 5, 8)
```

turtle.shearfactor(*shear=None*)

Parameters:: **shear** – number (optional)

Set or return the current shearfactor. Shear the turtleshape according to the given shearfactor shear, which is the tangent of the shear angle. Do *not* change the turtle’s heading (direction of movement). If shear is not given: return the current shearfactor, i. e. the tangent of the shear angle, by which lines parallel to the heading of the turtle are sheared.

```
>>> turtle.shape("circle")
>>> turtle.shapesize(5,2)
>>> turtle.shearfactor(0.5)
>>> turtle.shearfactor()
0.5
```

turtle.tilt(*angle*)

Parameters:: **angle** – a number

Rotate the turtleshape by *angle* from its current tilt-angle, but do *not* change the turtle’s heading (direction of movement).

```
>>> turtle.reset()
>>> turtle.shape("circle")
>>> turtle.shapesize(5,2)
>>> turtle.tilt(30)
>>> turtle.fd(50)
>>> turtle.tilt(30)
>>> turtle.fd(50)
```

turtle.tiltangle(*angle=None*)

Parameters:: **angle** – a number (optional)

Set or return the current tilt-angle. If angle is given, rotate the turtleshape to point in the direction specified by angle, regardless of its current tilt-angle. Do *not* change the turtle’s heading (direction of movement). If angle is not given: return the current tilt-angle, i. e. the angle between the orientation of the turtleshape and the heading of the turtle (its direction of movement).

```
>>> turtle.reset()
>>> turtle.shape("circle")
>>> turtle.shapesize(5,2)
>>> turtle.tilt(45)
>>> turtle.tiltangle()
45.0
```

turtle.shapetransform(*t11=None*, *t12=None*, *t21=None*, *t22=None*)

Parameters::

- **t11** – a number (optional)
- **t12** – a number (optional)
- **t21** – a number (optional)
- **t12** – a number (optional)

Set or return the current transformation matrix of the turtle shape.

If none of the matrix elements are given, return the transformation matrix as a tuple of 4 elements. Otherwise set the given elements and transform the turtleshape according to the matrix consisting of first row t11, t12 and second row t21, t22. The determinant t11 * t22 - t12 * t21 must not be zero, otherwise an error is raised. Modify stretchfactor, shearfactor and tiltangle according to the given matrix.

```
>>> turtle = Turtle()
>>> turtle.shape("square")
>>> turtle.shapesize(4,2)
>>> turtle.shearfactor(-0.5)
>>> turtle.shapetransform()
(4.0, -1.0, -0.0, 2.0)
```

turtle.get_shapepoly()

Return the current shape polygon as tuple of coordinate pairs. This can be used to define a new shape or components of a compound shape.

```
>>> turtle.shape("square")
>>> turtle.shapetransform(4, -1, 0, 2)
>>> turtle.get_shapepoly()
((50, -20), (30, 20), (-50, 20), (-30, -20))
```

Using events

turtle.onclick(*fun*, *btn=1*, *add=None*)

Parameters::

- **fun** – a function with two arguments which will be called with the coordinate of the clicked point on the canvas
- **btn** – number of the mouse-button, defaults to 1 (left mouse button)
- **add** – `True` or `False` – if `True`, a new binding will be added, otherwise it will replace a former binding

Bind *fun* to mouse-click events on this turtle. If *fun* is `None`, existing bindings are removed. Example for the anonymous turtle, i.e. the procedural way:

```
>>> def turn(x, y):
...     left(180)
...
>>> onclick(turn)  # Now clicking into the turtle will turn
it.
>>> onclick(None)  # event-binding will be removed
```

turtle.onrelease(*fun*, *btn=1*, *add=None*)

Parameters::

- **fun** – a function with two arguments which will be called with the coordinates of the clicked point on the canvas
- **btn** – number of the mouse-button, defaults to 1 (left mouse button)
- **add** – `True` or `False` – if `True`, a new binding will be added, otherwise it will replace a former binding

Bind *fun* to mouse-button-release events on this turtle. If *fun* is `None`, existing bindings are removed.

```
>>> class MyTurtle(Turtle):
...     def glow(self,x,y):
...         self.fillcolor("red")
...     def unglow(self,x,y):
...         self.fillcolor("")
...
>>> turtle = MyTurtle()
>>> turtle.onclick(turtle.glow)     # clicking on turtle
turns fillcolor red,
>>> turtle.onrelease(turtle.unglow) # releasing turns it to
transparent.
```

turtle.ondrag(*fun*, *btn=1*, *add=None*)

Parameters::

- **fun** – a function with two arguments which will be called with the coordinates of the clicked point on the canvas
- **btn** – number of the mouse-button, defaults to 1 (left mouse button)
- **add** – `True` or `False` – if `True`, a new binding will be added, otherwise it will replace a former binding

Bind *fun* to mouse-move events on this turtle. If *fun* is `None`, existing bindings are removed.

Remark: Every sequence of mouse-move-events on a turtle is preceded by a mouse-click event on that turtle.

```
>>> turtle.ondrag(turtle.goto)
```

Subsequently, clicking and dragging the Turtle will move it across the screen thereby producing handdrawings (if pen is down).

Special Turtle methods

turtle.poly()

Record the vertices of a polygon drawn in the `with turtle.poly():` block. The first and last vertices will be connected.

```
>>> with turtle.poly():
...     turtle.forward(100)
...     turtle.right(60)
...     turtle.forward(100)
```

Added in version 3.14.

turtle.begin_poly()

Start recording the vertices of a polygon. Current turtle position is first vertex of polygon.

turtle.end_poly()

Stop recording the vertices of a polygon. Current turtle position is last vertex of polygon. This will be connected with the first vertex.

turtle.get_poly()

Return the last recorded polygon.

```
>>> turtle.home()
>>> turtle.begin_poly()
>>> turtle.fd(100)
>>> turtle.left(20)
>>> turtle.fd(30)
>>> turtle.left(60)
>>> turtle.fd(50)
>>> turtle.end_poly()
>>> p = turtle.get_poly()
>>> register_shape("myFavouriteShape", p)
```

turtle.clone()

Create and return a clone of the turtle with same position, heading and turtle properties.

```
>>> mick = Turtle()
>>> joe = mick.clone()
```

turtle.getturtle()

turtle.getpen()

Return the Turtle object itself. Only reasonable use: as a function to return the “anonymous turtle”:

```
>>> pet = getturtle()
>>> pet.fd(50)
>>> pet
<turtle.Turtle object at 0x...>
```

turtle.getscreen()

Return the `TurtleScreen` object the turtle is drawing on. TurtleScreen methods can then be called for that object.

```
>>> ts = turtle.getscreen()
>>> ts
<turtle._Screen object at 0x...>
>>> ts.bgcolor("pink")
```

turtle.setundobuffer(*size*)

Parameters:: **size** – an integer or `None`

Set or disable undobuffer. If *size* is an integer, an empty undobuffer of given size is installed. *size* gives the maximum number of turtle actions that can be undone by the `undo()` method/function. If *size* is `None`, the undobuffer is disabled.

```
>>> turtle.setundobuffer(42)
```

turtle.undobufferentries()

Return number of entries in the undobuffer.

```
>>> while undobufferentries():
...     undo()
```

Compound shapes

To use compound turtle shapes, which consist of several polygons of different color, you must use the helper class `Shape` explicitly as described below:

1. Create an empty Shape object of type “compound”.
2. Add as many components to this object as desired, using the `addcomponent()` method.

   For example:

   ```
   >>> s = Shape("compound")
   >>> poly1 = ((0,0),(10,-5),(0,10),(-10,-5))
   >>> s.addcomponent(poly1, "red", "blue")
   >>> poly2 = ((0,0),(10,-5),(-10,-5))
   >>> s.addcomponent(poly2, "blue", "red")
   ```

3. Now add the Shape to the Screen’s shapelist and use it:

   ```
   >>> register_shape("myshape", s)
   >>> shape("myshape")
   ```

Note

The `Shape` class is used internally by the `register_shape()` method in different ways. The application programmer has to deal with the Shape class *only* when using compound shapes like shown above!

Methods of TurtleScreen/Screen and corresponding functions

Most of the examples in this section refer to a TurtleScreen instance called `screen`.

Window control

turtle.bgcolor()
turtle.bgcolor(*color*, */*)
turtle.bgcolor(*r*, *g*, *b*, */*)

Return or set the background color of the TurtleScreen.

Four input formats are allowed:

`bgcolor()`

Return the current background color as color specification string or as a tuple (see example). May be used as input to another color/pencolor/fillcolor/bgcolor call.

`bgcolor(colorstring)`

Set the background color to *colorstring*, which is a Tk color specification string, such as `"red"`, `"yellow"`, or `"#33cc8c"`.

`bgcolor((r, g, b))`

Set the background color to the RGB color represented by the tuple of *r*, *g*, and *b*. Each of *r*, *g*, and *b* must be in the range 0..colormode, where colormode is either 1.0 or 255 (see **`colormode()`**).

`bgcolor(r, g, b)`

Set the background color to the RGB color represented by *r*, *g*, and *b*. Each of *r*, *g*, and *b* must be in the range 0..colormode.

```
>>> screen.bgcolor("orange")
>>> screen.bgcolor()
'orange'
>>> screen.bgcolor("#800080")
>>> screen.bgcolor()
(128.0, 0.0, 128.0)
```

turtle.bgpic(*picname=None*)

Parameters:: **picname** – a string, name of an image file (PNG, GIF, PGM, and PPM) or `"nopic"`, or `None`

Set background image or return name of current backgroundimage. If *picname* is a filename, set the corresponding image as background. If *picname* is `"nopic"`, delete background image, if present. If *picname* is `None`, return the filename of the current backgroundimage.

```
>>> screen.bgpic()
'nopic'
>>> screen.bgpic("landscape.gif")
>>> screen.bgpic()
"landscape.gif"
```

turtle.clear()

Note

This TurtleScreen method is available as a global function only under the name `clearscreen`. The global function `clear` is a different one derived from the Turtle method `clear`.

turtle.clearscreen()

Delete all drawings and all turtles from the TurtleScreen. Reset the now empty TurtleScreen to its initial state: white background, no background image, no event bindings and tracing on.

turtle.reset()

Note

This TurtleScreen method is available as a global function only under the name `resetscreen`. The global function `reset` is another one derived from the Turtle method `reset`.

turtle.resetscreen()

Reset all Turtles on the Screen to their initial state.

turtle.screensize(*canvwidth=None*, *canvheight=None*, *bg=None*)

Parameters::

- **canvwidth** – positive integer, new widt of canvas in pixels
- **canvheight** – positive integer, new height of canvas in pixels
- **bg** – colorstring or color-tuple, new background color

If no arguments are given, return current (canvaswidth, canvasheight). Else resize the canvas the turtles are drawing on. Do not alter the drawing window. To observe hidden parts of the canvas, use the scrollbars. With this method, one can make visible those parts of a drawing which were outside the canvas before.

```
>>> screen.screensize()
(400, 300)
>>> screen.screensize(2000,1500)
>>> screen.screensize()
(2000, 1500)
```

e.g. to search for an erroneously escaped turtle ;-)

turtle.setworldcoordinates(*llx*, *lly*, *urx*, *ury*)

Parameters::

- **llx** – a number, x-coordinate of lower lef corner of canvas
- **lly** – a number, y-coordinate of lower lef corner of canvas
- **urx** – a number, x-coordinate of upper right corner of canvas
- **ury** – a number, y-coordinate of upper right corner of canvas

Set up user-defined coordinate system and switch to mode “world” if necessary. This performs a `screen.reset()`. If mode “world” is already active, all drawings are redrawn according to the new coordinates.

ATTENTION: in user-defined coordinate systems angles may appear distorted.

```
>>> screen.reset()
>>> screen.setworldcoordinates(-50,-7.5,50,7.5)
>>> for _ in range(72):
...     left(10)
...
>>> for _ in range(8):
...     left(45); fd(2)   # a regular octagon
```

Animation control

turtle.no_animation()

Temporarily disable turtle animation. The code written inside the `no_animation` block will not be animated; once the code block is exited, the drawing will appear.

```
>>> with screen.no_animation():
...     for dist in range(2, 400, 2):
...         fd(dist)
...         rt(90)
```

Added in version 3.14.

turtle.delay(*delay=None*)

Parameters:: **delay** – positive integer

Set or return the drawing *delay* in milliseconds. (This is approximately the time interval between two consecutive canvas updates.) The longer the drawing delay, the slower the animation.

Optional argument:

```
>>> screen.delay()
10
>>> screen.delay(5)
>>> screen.delay()
5
```

turtle.tracer(*n=None*, *delay=None*)

Parameters::
- **n** – nonnegative integer
- **delay** – nonnegative integer

Turn turtle animation on/off and set delay for update drawings. If *n* is given, only each n-th regular screen update is really performed. (Can be used to accelerate the drawing of complex graphics.) When called without arguments, returns the currently stored value of n. Second argument sets delay value (see `delay()`).

```
>>> screen.tracer(8, 25)
>>> dist = 2
>>> for i in range(200):
...     fd(dist)
...     rt(90)
...     dist += 2
```

turtle.update()

Perform a TurtleScreen update. To be used when tracer is turned off.

See also the RawTurtle/Turtle method `speed()`.

Using screen events

turtle.listen(*xdummy=None*, *ydummy=None*)

Set focus on TurtleScreen (in order to collect key-events). Dummy arguments are provided in order to be able to pass `listen()` to the onclick method.

turtle.onkey(*fun*, *key*)
turtle.onkeyrelease(*fun*, *key*)

Parameters::

- **fun** – a function with no arguments or `None`
- **key** – a string: key (e.g. “a”) or key-symbol (e.g. “space”)

Bind *fun* to key-release event of key. If *fun* is `None`, event bindings are removed. Remark: in order to be able to register key-events, TurtleScreen must have the focus. (See method `listen()`.)

```
>>> def f():
...     fd(50)
...     lt(60)
...
>>> screen.onkey(f, "Up")
>>> screen.listen()
```

turtle.onkeypress(*fun*, *key=None*)

Parameters::

- **fun** – a function with no arguments or `None`
- **key** – a string: key (e.g. “a”) or key-symbol (e.g. “space”)

Bind *fun* to key-press event of key if key is given, or to any key-press-event if no key is given. Remark: in order to be able to register key-events, TurtleScreen must have focus. (See method `listen()`.)

```
>>> def f():
...     fd(50)
...
>>> screen.onkey(f, "Up")
>>> screen.listen()
```

turtle.onclick(*fun*, *btn=1*, *add=None*)
turtle.onscreenclick(*fun*, *btn=1*, *add=None*)

Parameters::

- **fun** – a function with two arguments which will be called with the coordinates of the clicked point on the canvas
- **btn** – number of the mouse-button, defaults to 1 (left mouse button)
- **add** – `True` or `False` – if `True`, a new binding will be added, otherwise it will replace a former binding

Bind *fun* to mouse-click events on this screen. If *fun* is `None`, existing bindings are removed.

Example for a TurtleScreen instance named `screen` and a Turtle instance named `turtle`:

```
>>> screen.onclick(turtle.goto) # Subsequently clicking into
the TurtleScreen will
>>>                             # make the turtle move to the
clicked point.
>>> screen.onclick(None)        # remove event binding again
```

Note

This TurtleScreen method is available as a global function only under the name `onscreenclick`. The global function `onclick` is another one derived from the Turtle method `onclick`.

turtle.ontimer(*fun*, *t=0*)

Parameters::
- **fun** – a function with no arguments
- **t** – a number >= 0

Install a timer that calls *fun* after *t* milliseconds.

```
>>> running = True
>>> def f():
...     if running:
...         fd(50)
...         lt(60)
...         screen.ontimer(f, 250)
>>> f()   ### makes the turtle march around
>>> running = False
```

turtle.mainloop()

turtle.done()

Starts event loop - calling Tkinter’s mainloop function. Must be the last statement in a turtle graphics program. Must *not* be used if a script is run from within IDLE in -n mode (No subprocess) - for interactive use of turtle graphics.

```
>>> screen.mainloop()
```

Input methods

turtle.textinput(*title*, *prompt*)

Parameters::
- **title** – string
- **prompt** – string

Pop up a dialog window for input of a string. Parameter title is the title of the dialog window, prompt is a text mostly describing what information to input. Return the string input. If the dialog is canceled, return `None`.

```
>>> screen.textinput("NIM", "Name of first player:")
```

turtle.numinput(*title*, *prompt*, *default=None*, *minval=None*, *maxval=None*)

Parameters::
- **title** – string
- **prompt** – string
- **default** – number (optional)
- **minval** – number (optional)
- **maxval** – number (optional)

Pop up a dialog window for input of a number. title is the title of the dialog window, prompt is a text mostly describing what numerical information to input. default: default value, minval: minimum value for input, maxval: maximum value for input. The number input must be in the range minval .. maxval if these are given. If not, a hint is issued and the dialog remains open for correction. Return the number input. If the dialog is canceled, return `None`.

```
>>> screen.numinput("Poker", "Your stakes:", 1000, minval=10,
maxval=10000)
```

Settings and special methods

turtle.mode(*mode=None*)

Parameters:: **mode** – one of the strings "standard", "logo" or "world"

Set turtle mode ("standard", "logo" or "world") and perform reset. If mode is not given, current mode is returned.

Mode "standard" is compatible with old `turtle`. Mode "logo" is compatible with most Logo turtle graphics. Mode "world" uses user-defined "world coordinates". **Attention**: in this mode angles appear distorted if `x/y` unit-ratio doesn't equal 1.

Mode	Initial turtle heading	positive angles
"standard"	to the right (east)	counterclockwise

Mode	Initial turtle heading	positive angles
“logo”	upward (north)	clockwise

```
>>> mode("logo")   # resets turtle heading to north
>>> mode()
'logo'
```

turtle.colormode(*cmode=None*)

Parameters:: **cmode** – one of the values 1.0 or 255

Return the colormode or set it to 1.0 or 255. Subsequently *r*, *g*, *b* values of color triples have to be in the range 0..*cmode*.

```
>>> screen.colormode(1)
>>> turtle.pencolor(240, 160, 80)
Traceback (most recent call last):
     ...
TurtleGraphicsError: bad color sequence: (240, 160, 80)
>>> screen.colormode()
1.0
>>> screen.colormode(255)
>>> screen.colormode()
255
>>> turtle.pencolor(240,160,80)
```

turtle.getcanvas()

Return the Canvas of this TurtleScreen. Useful for insiders who know what to do with a Tkinter Canvas.

```
>>> cv = screen.getcanvas()
>>> cv
<turtle.ScrolledCanvas object ...>
```

turtle.getshapes()

Return a list of names of all currently available turtle shapes.

```
>>> screen.getshapes()
['arrow', 'blank', 'circle', ..., 'turtle']
```

turtle.register_shape(*name*, *shape=None*)
turtle.addshape(*name*, *shape=None*)

There are four different ways to call this function:

1. *name* is the name of an image file (PNG, GIF, PGM, and PPM) and *shape* is `None`: Install the corresponding image shape.

   ```
   >>> screen.register_shape("turtle.gif")
   ```

   Note

   Image shapes *do not* rotate when turning the turtle, so they do not display the heading of the turtle!

2. *name* is an arbitrary string and *shape* is the name of an image file (PNG, GIF, PGM, and PPM): Install the corresponding image shape.

   ```
   >>> screen.register_shape("turtle", "turtle.gif")
   ```

   Note

   Image shapes *do not* rotate when turning the turtle, so they do not display the heading of the turtle!

3. *name* is an arbitrary string and *shape* is a tuple of pairs of coordinates: Install the corresponding polygon shape.

   ```
   >>> screen.register_shape("triangle", ((5,-3), (0,5),
   (-5,-3)))
   ```

4. *name* is an arbitrary string and *shape* is a (compound) `Shape` object: Install the corresponding compound shape.

Add a turtle shape to TurtleScreen’s shapelist. Only thusly registered shapes can be used by issuing the command `shape(shapename)`.

Changed in version 3.14: Added support for PNG, PGM, and PPM image formats. Both a shape name and an image file name can be specified.

turtle.turtles()

Return the list of turtles on the screen.

```
>>> for turtle in screen.turtles():
...     turtle.color("red")
```

turtle.window_height()

Return the height of the turtle window.

```
>>> screen.window_height()
480
```

turtle.window_width()

Return the width of the turtle window.

```
>>> screen.window_width()
640
```

Methods specific to Screen, not inherited from TurtleScreen

turtle.bye()

Shut the turtlegraphics window.

turtle.exitonclick()

Bind `bye()` method to mouse clicks on the Screen.

If the value “using_IDLE” in the configuration dictionary is `False` (default value), also enter mainloop. Remark: If IDLE with the `-n` switch (no subprocess) is used, this value should be set to `True` in `turtle.cfg`. In this case IDLE's own mainloop is active also for the client script.

turtle.save(*filename*, *overwrite=False*)

Save the current turtle drawing (and turtles) as a PostScript file.

Parameters::

- **filename** – the path of the saved PostScript file
- **overwrite** – if `False` and there already exists a file with the given filename, then the function will raise a `FileExistsError`. If it is `True`, the file will b overwritten.

```
>>> screen.save("my_drawing.ps")
>>> screen.save("my_drawing.ps", overwrite=True)
```

Added in version 3.14.

turtle.setup(*width=_CFG['width']*, *height=_CFG['height']*, *startx=_CFG['leftright']*, *starty=_CFG['topbottom']*)

Set the size and position of the main window. Default values of arguments are stored in the configuration dictionary and can be changed via a `turtle.cfg` file.

Parameters::

- **width** – if an integer, a size in pixels, if float, a fraction of the screen; default is 50% of screen
- **height** – if an integer, the height in pixels, if a float, a fraction of the screen default is 75% of screen
- **startx** – if positive, starting position in pixels from the left edge of the screen, i negative from the right edge, if `None`, center window horizontally
- **starty** – if positive, starting position in pixels from the top edge of the screen, i negative from the bottom edge, if `None`, center window vertically

```
>>> screen.setup (width=200, height=200, startx=0, starty=0)
>>>              # sets window to 200x200 pixels, in upper
left of screen
>>> screen.setup(width=.75, height=0.5, startx=None,
starty=None)
>>>              # sets window to 75% of screen by 50% of
screen and centers
```

turtle.title(*titlestring*)

Parameters:: **titlestring** – a string that is shown in the titlebar of the turtle graphics window

Set title of turtle window to *titlestring*.

```
>>> screen.title("Welcome to the turtle zoo!")
```

Public classes

class turtle.RawTurtle(*canvas*)
class turtle.RawPen(*canvas*)

Parameters:: **canvas** – a **`tkinter.Canvas`**, a `ScrolledCanvas` or a `TurtleScreen`

Create a turtle. The turtle has all methods described above as “methods of Turtle/RawTurtle”.

class turtle.Turtle

Subclass of RawTurtle, has the same interface but draws on a default `Screen` object created automatically when needed for the first time.

class turtle.TurtleScreen(*cv*)

Parameters:: **cv** – a `tkinter.Canvas`

Provides screen oriented methods like `bgcolor()` etc. that are described above.

class turtle.Screen

Subclass of TurtleScreen, with four methods added.

class turtle.ScrolledCanvas(*master*)

Parameters:: **master** – some Tkinter widget to contain the ScrolledCanvas, i.e. a Tkinter-canvas with scrollbars added

Used by class Screen, which thus automatically provides a ScrolledCanvas as playground for the turtles.

class turtle.Shape(*type_*, *data*)

Parameters:: **type_** – one of the strings “polygon”, “image”, “compound”

Data structure modeling shapes. The pair `(type_, data)` must follow this specification:

type_	***data***
“polygon”	a polygon-tuple, i.e. a tuple of pairs of coordinates

type_	***data***
"image"	an image (in this form only used internally!)
"compound"	`None` (a compound shape has to be constructed using the [`addcomponent()`]() method)

addcomponent(*poly*, *fill*, *outline=None*)

Parameters::
- **poly** – a polygon, i.e. a tuple of pairs of numbers
- **fill** – a color the *poly* will be filled wit
- **outline** – a color for the poly's outlin (if given)

Example:

```
>>> poly = ((0,0),(10,-5),(0,10),(-10,-5))
>>> s = Shape("compound")
>>> s.addcomponent(poly, "red", "blue")
>>> # ... add more components and then use
register_shape()
```

See [Compound shapes]().

class turtle.Vec2D(*x*, *y*)

A two-dimensional vector class, used as a helper class for implementing turtle graphics. May be useful for turtle graphics programs too. Derived from tuple, so a vector is a tuple!

Provides (for *a*, *b* vectors, *k* number):

- `a + b` vector addition
- `a - b` vector subtraction
- `a * b` inner product
- `k * a` and `a * k` multiplication with scalar
- `abs(a)` absolute value of a

- `a.rotate(angle)` rotation

Explanation

A turtle object draws on a screen object, and there a number of key classes in the turtle object-oriented interface that can be used to create them and relate them to each other.

A **`Turtle`** instance will automatically create a **`Screen`** instance if one is not already present.

`Turtle` is a subclass of **`RawTurtle`**, which *doesn't* automatically create a drawing surface - a *canvas* will need to be provided or created for it. The *canvas* can be a **`tkinter.Canvas`**, **`ScrolledCanvas`** or **`TurtleScreen`**.

`TurtleScreen` is the basic drawing surface for a turtle. **`Screen`** is a subclass of `TurtleScreen`, and includes some additional methods for managing its appearance (including size and title) and behaviour. `TurtleScreen`'s constructor needs a **`tkinter.Canvas`** or a **`ScrolledCanvas`** as an argument.

The functional interface for turtle graphics uses the various methods of `Turtle` and `TurtleScreen`/`Screen`. Behind the scenes, a screen object is automatically created whenever a function derived from a `Screen` method is called. Similarly, a turtle object is automatically created whenever any of the functions derived from a Turtle method is called.

To use multiple turtles on a screen, the object-oriented interface must be used.

Help and configuration

How to use help

The public methods of the Screen and Turtle classes are documented extensively via docstrings. So these can be used as online-help via the Python help facilities:

- When using IDLE, tooltips show the signatures and first lines of the docstrings of typed in function-/method calls.
- Calling `help()` on methods or functions displays the docstrings:

  ```
  >>> help(Screen.bgcolor)
  Help on method bgcolor in module turtle:

  bgcolor(self, *args) unbound turtle.Screen method
      Set or return backgroundcolor of the TurtleScreen.

      Arguments (if given): a color string or three numbers
      in the range 0..colormode or a 3-tuple of such numbers.


      >>> screen.bgcolor("orange")
      >>> screen.bgcolor()
      "orange"
      >>> screen.bgcolor(0.5,0,0.5)
      >>> screen.bgcolor()
      "#800080"

  >>> help(Turtle.penup)
  Help on method penup in module turtle:

  penup(self) unbound turtle.Turtle method
      Pull the pen up -- no drawing when moving.

      Aliases: penup | pu | up

      No argument

      >>> turtle.penup()
  ```

- The docstrings of the functions which are derived from methods have a modified form:

```
>>> help(bgcolor)
Help on function bgcolor in module turtle:

bgcolor(*args)
    Set or return backgroundcolor of the TurtleScreen.

    Arguments (if given): a color string or three numbers
    in the range 0..colormode or a 3-tuple of such numbers.

    Example::

      >>> bgcolor("orange")
      >>> bgcolor()
      "orange"
      >>> bgcolor(0.5,0,0.5)
      >>> bgcolor()
      "#800080"

>>> help(penup)
Help on function penup in module turtle:

penup()
    Pull the pen up -- no drawing when moving.

    Aliases: penup | pu | up

    No argument

    Example:
    >>> penup()
```

These modified docstrings are created automatically together with the function definitions that are derived from the methods at import time.

Translation of docstrings into different languages

There is a utility to create a dictionary the keys of which are the method names and the values of which are the docstrings of the public methods of the classes Screen and Turtle.

turtle.write_docstringdict(*filename='turtle_docstringdict'*)

Parameters:: **filename** – a string, used as filename

Create and write docstring-dictionary to a Python script with the given filename. This function has to be called explicitly (it is not used by the turtle graphics classes). The docstring dictionary will be written to the Python script `filename.py`. It is intended to serve as a template for translation of the docstrings into different languages.

If you (or your students) want to use **`turtle`** with online help in your native language, you have to translate the docstrings and save the resulting file as e.g. `turtle_docstringdict_german.py`.

If you have an appropriate entry in your `turtle.cfg` file this dictionary will be read in at import time and will replace the original English docstrings.

At the time of this writing there are docstring dictionaries in German and in Italian. (Requests please to glingl@aon.at.)

How to configure Screen and Turtles

The built-in default configuration mimics the appearance and behaviour of the old turtle module in order to retain best possible compatibility with it.

If you want to use a different configuration which better reflects the features of this module or which better fits to your needs, e.g. for use in a classroom, you can prepare a configuration file `turtle.cfg` which will be read at import time and modify the configuration according to its settings.

The built in configuration would correspond to the following `turtle.cfg`:

```
width = 0.5
height = 0.75
leftright = None
topbottom = None
canvwidth = 400
canvheight = 300
mode = standard
colormode = 1.0
delay = 10
undobuffersize = 1000
shape = classic
pencolor = black
fillcolor = black
resizemode = noresize
visible = True
language = english
exampleturtle = turtle
examplescreen = screen
title = Python Turtle Graphics
using_IDLE = False
```

Short explanation of selected entries:

- The first four lines correspond to the arguments of the `Screen.setup` method.
- Line 5 and 6 correspond to the arguments of the method `Screen.screensize`.
- *shape* can be any of the built-in shapes, e.g: arrow, turtle, etc. For more info try `help(shape)`.
- If you want to use no fill color (i.e. make the turtle transparent), you have to write `fillcolor = ""` (but all nonempty strings must not have quotes in the cfg file).
- If you want to reflect the turtle its state, you have to use `resizemode = auto`.
- If you set e.g. `language = italian` the docstringdict `turtle_docstringdict_italian.py` will be loaded at import time (if present on the import path, e.g. in the same directory as `turtle`).
- The entries *exampleturtle* and *examplescreen* define the names of these objects as they occur in the docstrings.

The transformation of method-docstrings to function-docstrings will delete these names from the docstrings.
- *using_IDLE*: Set this to `True` if you regularly work with IDLE and its `-n` switch ("no subprocess"). This will prevent **`exitonclick()`** to enter the mainloop.

There can be a `turtle.cfg` file in the directory where **`turtle`** is stored and an additional one in the current working directory. The latter will override the settings of the first one.

The `Lib/turtledemo` directory contains a `turtle.cfg` file. You can study it as an example and see its effects when running the demos (preferably not from within the demo-viewer).

`turtledemo` — Demo scripts

The **`turtledemo`** package includes a set of demo scripts. These scripts can be run and viewed using the supplied demo viewer as follows:

```
python -m turtledemo
```

Alternatively, you can run the demo scripts individually. For example,

```
python -m turtledemo.bytedesign
```

The **`turtledemo`** package directory contains:

- A demo viewer `__main__.py` which can be used to view the sourcecode of the scripts and run them at the same time.
- Multiple scripts demonstrating different features of the **`turtle`** module. Examples can be accessed via the Examples menu. They can also be run standalone.
- A `turtle.cfg` file which serves as an example of how to write and use such files.

The demo scripts are:

Name	**Description**	**Features**
`bytedesign`	complex classical turtle graphics pattern	`tracer()`, `delay()`, `update()`
`chaos`	graphs Verhulst dynamics, shows that computer’s computations can generate results sometimes against the common sense expectations	world coordinates
`clock`	analog clock showing time of your computer	turtles as clock’s hands, `ontimer()`
`colormixer`	experiment with r, g, b	`ondrag()`
`forest`	3 breadth-first trees	randomization
`fractalcurves`	Hilbert & Koch curves	recursion
`lindenmayer`	ethnomathematics (indian kolams)	L-System
`minimal_hanoi`	Towers of Hanoi	Rectangular Turtles as Hanoi discs (`shape()`, `shapesize()`)

Name	Description	Features
`nim`	play the classical nim game with three heaps of sticks against the computer.	turtles as nimsticks, event driven (mouse, keyboard)
`paint`	super minimalistic drawing program	`onclick()`
`peace`	elementary	turtle: appearance and animation
`penrose`	aperiodic tiling with kites and darts	`stamp()`
`planet_and_moon`	simulation of gravitational system	compound shapes, `Vec2D`
`rosette`	a pattern from the wikipedia article on turtle graphics	`clone()`, `undo()`
`round_dance`	dancing turtles rotating pairwise in opposite direction	compound shapes, `clone()` `shapesize()`, `tilt()`,

Name	Description	Features
		`get_shapepoly()`, `update()`
`sorting_animate`	visual demonstration of different sorting methods	simple alignment, randomization
`tree`	a (graphical) breadth first tree (using generators)	`clone()`
`two_canvases`	simple design	turtles on two canvases
`yinyang`	another elementary example	`circle()`

Have fun!

Changes since Python 2.6

- The methods `Turtle.tracer`, `Turtle.window_width` and `Turtle.window_height` have been eliminated. Methods with these names and functionality are now available only as methods of `Screen`. The functions derived from these remain available. (In fact already in Python 2.6 these methods were merely duplications of the corresponding `TurtleScreen`/`Screen` methods.)
- The method **`Turtle.fill()`** has been eliminated. The behaviour of `begin_fill()` and `end_fill()` have changed

slightly: now every filling process must be completed with an `end_fill()` call.
- A method **`Turtle.filling`** has been added. It returns a boolean value: `True` if a filling process is under way, `False` otherwise. This behaviour corresponds to a `fill()` call without arguments in Python 2.6.

Changes since Python 3.0

- The **`Turtle`** methods **`shearfactor()`**, **`shapetransform()`** and **`get_shapepoly()`** have been added. Thus the full range of regular linear transforms is now available for transforming turtle shapes. **`tiltangle()`** has been enhanced in functionality: it now can be used to get or set the tilt angle.
- The **`Screen`** method **`onkeypress()`** has been added as a complement to **`onkey()`**. As the latter binds actions to the key release event, an alias: **`onkeyrelease()`** was also added for it.
- The method **`Screen.mainloop`** has been added, so there is no longer a need to use the standalone **`mainloop()`** function when working with **`Screen`** and **`Turtle`** objects.
- Two input methods have been added: **`Screen.textinput`** and **`Screen.numinput`**. These pop up input dialogs and return strings and numbers respectively.

Development Tools

The modules described in this chapter help you write software. For example, the `pydoc` module takes a module and generates documentation based on the module’s contents. The `doctest` and `unittest` modules contains frameworks for writing unit tests that automatically exercise code and verify that the expected output is produced.

The list of modules described in this chapter is:

- `typing` — Support for type hints
  - Specification for the Python Type System
  - Type aliases
  - NewType
  - Annotating callable objects
  - Generics
  - Annotating tuples
  - The type of class objects
  - Annotating generators and coroutines
  - User-defined generic types
  - The `Any` type
  - Nominal vs structural subtyping
  - Module contents
    - Special typing primitives
      - Special types
      - Special forms
      - Building generic types and type aliases
      - Other special directives
    - Protocols
    - ABCs and Protocols for working with I/O
    - Functions and decorators
    - Introspection helpers
    - Constant

        - Deprecated aliases
            - Aliases to built-in types
            - Aliases to types in **`collections`**
            - Aliases to other concrete types
            - Aliases to container ABCs in **`collections.abc`**
            - Aliases to asynchronous ABCs in **`collections.abc`**
            - Aliases to other ABCs in **`collections.abc`**
            - Aliases to **`contextlib`** ABCs
    - Deprecation Timeline of Major Features
- **`pydoc`** — Documentation generator and online help system
- Python Development Mode
    - Effects of the Python Development Mode
    - ResourceWarning Example
    - Bad file descriptor error example
- **`doctest`** — Test interactive Python examples
    - Simple Usage: Checking Examples in Docstrings
    - Simple Usage: Checking Examples in a Text File
    - Command-line Usage
    - How It Works
        - Which Docstrings Are Examined?
        - How are Docstring Examples Recognized?
        - What’s the Execution Context?
        - What About Exceptions?
        - Option Flags
        - Directives
        - Warnings
    - Basic API
    - Unittest API
    - Advanced API
        - DocTest Objects
        - Example Objects
        - DocTestFinder objects
        - DocTestParser objects
        - TestResults objects

        - DocTestRunner objects
        - OutputChecker objects
    - Debugging
    - Soapbox
- `unittest` — Unit testing framework
    - Basic example
    - Command-Line Interface
        - Command-line options
    - Test Discovery
    - Organizing test code
    - Re-using old test code
    - Skipping tests and expected failures
    - Distinguishing test iterations using subtests
    - Classes and functions
        - Test cases
        - Grouping tests
        - Loading and running tests
            - load_tests Protocol
    - Class and Module Fixtures
        - setUpClass and tearDownClass
        - setUpModule and tearDownModule
    - Signal Handling
- `unittest.mock` — mock object library
    - Quick Guide
    - The Mock Class
        - Calling
        - Deleting Attributes
        - Mock names and the name attribute
        - Attaching Mocks as Attributes
    - The patchers
        - patch
        - patch.object
        - patch.dict
        - patch.multiple
        - patch methods: start and stop
        - patch builtins

        - TEST_PREFIX
        - Nesting Patch Decorators
        - Where to patch
        - Patching Descriptors and Proxy Objects
    - MagicMock and magic method support
        - Mocking Magic Methods
        - Magic Mock
    - Helpers
        - sentinel
        - DEFAULT
        - call
        - create_autospec
        - ANY
        - FILTER_DIR
        - mock_open
        - Autospeccing
        - Sealing mocks
    - Order of precedence of **`side_effect`**, **`return_value`** and *wraps*
- **`unittest.mock`** — getting started
    - Using Mock
        - Mock Patching Methods
        - Mock for Method Calls on an Object
        - Mocking Classes
        - Naming your mocks
        - Tracking all Calls
        - Setting Return Values and Attributes
        - Raising exceptions with mocks
        - Side effect functions and iterables
        - Mocking asynchronous iterators
        - Mocking asynchronous context manager
        - Creating a Mock from an Existing Object
        - Using side_effect to return per file content
    - Patch Decorators
    - Further Examples
        - Mocking chained calls

        - Partial mocking
        - Mocking a Generator Method
        - Applying the same patch to every test method
        - Mocking Unbound Methods
        - Checking multiple calls with mock
        - Coping with mutable arguments
        - Nesting Patches
        - Mocking a dictionary with MagicMock
        - Mock subclasses and their attributes
        - Mocking imports with patch.dict
        - Tracking order of calls and less verbose call assertions
        - More complex argument matching
- `test` — Regression tests package for Python
    - Writing Unit Tests for the `test` package
    - Running tests using the command-line interface
- `test.support` — Utilities for the Python test suite
- `test.support.socket_helper` — Utilities for socket tests
- `test.support.script_helper` — Utilities for the Python execution tests
- `test.support.bytecode_helper` — Support tools for testing correct bytecode generation
- `test.support.threading_helper` — Utilities for threading tests
- `test.support.os_helper` — Utilities for os tests
- `test.support.import_helper` — Utilities for import tests
- `test.support.warnings_helper` — Utilities for warnings tests

`typing` — Support for type hints

Added in version 3.5.

Source code: Lib/typing.py [https://github.com/python/cpython/tree/3.14/Lib/typing.py]

Note

The Python runtime does not enforce function and variable type annotations. They can be used by third party tools such as type checkers, IDEs, linters, etc.

This module provides runtime support for type hints.

Consider the function below:

```
def surface_area_of_cube(edge_length: float) -> str:
    return f"The surface area of the cube is {6 * edge_length ** 2}."
```

The function `surface_area_of_cube` takes an argument expected to be an instance of **`float`**, as indicated by the type hint `edge_length: float`. The function is expected to return an instance of **`str`**, as indicated by the `-> str` hint.

While type hints can be simple classes like **`float`** or **`str`**, they can also be more complex. The **`typing`** module provides a vocabulary of more advanced type hints.

New features are frequently added to the `typing` module. The typing_extensions [https://pypi.org/project/typing_extensions/] package provides backports of these new features to older versions of Python.

See also

Typing cheat sheet [https://mypy.readthedocs.io/en/stable/cheat_sheet_py3.html]
A quick overview of type hints (hosted at the mypy docs)

Type System Reference section of the mypy docs [https://mypy.readthedocs.io/en/stable/index.html]

The Python typing system is standardised via PEPs, so this reference should broadly apply to most Python type checkers. (Some parts may still be specific to mypy.)

Static Typing with Python [https://typing.python.org/en/latest/]

Type-checker-agnostic documentation written by the community detailing type system features, useful typing related tools and typing best practices.

Specification for the Python Type System

The canonical, up-to-date specification of the Python type system can be found at Specification for the Python type system [https://typing.python.org/en/latest/spec/index.html].

Type aliases

A type alias is defined using the **`type`** statement, which creates an instance of **`TypeAliasType`**. In this example, `Vector` and `list[float]` will be treated equivalently by static type checkers:

```
type Vector = list[float]

def scale(scalar: float, vector: Vector) -> Vector:
    return [scalar * num for num in vector]

# passes type checking; a list of floats qualifies as a Vector.
new_vector = scale(2.0, [1.0, -4.2, 5.4])
```

Type aliases are useful for simplifying complex type signatures. For example:

```
from collections.abc import Sequence

type ConnectionOptions = dict[str, str]
type Address = tuple[str, int]
type Server = tuple[Address, ConnectionOptions]

def broadcast_message(message: str, servers: Sequence[Server]) -> None:
    ...
```

```
# The static type checker will treat the previous type signature as
# being exactly equivalent to this one.
def broadcast_message(
    message: str,
    servers: Sequence[tuple[tuple[str, int], dict[str, str]]]
) -> None:
    ...
```

The **`type`** statement is new in Python 3.12. For backwards compatibility, type aliases can also be created through simple assignment:

```
Vector = list[float]
```

Or marked with **`TypeAlias`** to make it explicit that this is a type alias, not a normal variable assignment:

```
from typing import TypeAlias

Vector: TypeAlias = list[float]
```

NewType

Use the **`NewType`** helper to create distinct types:

```
from typing import NewType

UserId = NewType('UserId', int)
some_id = UserId(524313)
```

The static type checker will treat the new type as if it were a subclass of the original type. This is useful in helping catch logical errors:

```
def get_user_name(user_id: UserId) -> str:
    ...

# passes type checking
user_a = get_user_name(UserId(42351))

# fails type checking; an int is not a UserId
user_b = get_user_name(-1)
```

You may still perform all `int` operations on a variable of type `UserId`, but the result will always be of type `int`. This lets you pass in a `UserId` wherever an `int` might be expected, but will prevent you from accidentally creating a `UserId` in an invalid way:

```
# 'output' is of type 'int', not 'UserId'
output = UserId(23413) + UserId(54341)
```

Note that these checks are enforced only by the static type checker. At runtime, the statement `Derived = NewType('Derived', Base)` will make `Derived` a callable that immediately returns whatever parameter you pass it. That means the expression `Derived(some_value)` does not create a new class or introduce much overhead beyond that of a regular function call.

More precisely, the expression `some_value is Derived(some_value)` is always true at runtime.

It is invalid to create a subtype of `Derived`:

```
from typing import NewType

UserId = NewType('UserId', int)

# Fails at runtime and does not pass type checking
class AdminUserId(UserId): pass
```

However, it is possible to create a **`NewType`** based on a ‘derived’ `NewType`:

```
from typing import NewType

UserId = NewType('UserId', int)

ProUserId = NewType('ProUserId', UserId)
```

and typechecking for `ProUserId` will work as expected.

See **PEP 484** [https://peps.python.org/pep-0484/] for more details.

Note

Recall that the use of a type alias declares two types to be *equivalent* to one another. Doing `type Alias = Original` will make the static type checker treat `Alias` as being *exactly equivalent* to `Original` in all cases. This is useful when you want to simplify complex type signatures.

In contrast, `NewType` declares one type to be a *subtype* of another. Doing `Derived = NewType('Derived', Original)` will make the static type checker treat `Derived` as a *subclass* of `Original`, which means a value of type `Original` cannot be used in places where a value of type `Derived` is expected. This is useful when you want to prevent logic errors with minimal runtime cost.

Added in version 3.5.2.

Changed in version 3.10: `NewType` is now a class rather than a function. As a result, there is some additional runtime cost when calling `NewType` over a regular function.

Changed in version 3.11: The performance of calling `NewType` has been restored to its level in Python 3.9.

Annotating callable objects

Functions – or other callable objects – can be annotated using **`collections.abc.Callable`** or deprecated **`typing.Callable`**. `Callable[[int], str]` signifies a function that takes a single parameter of type **`int`** and returns a **`str`**.

For example:

```
from collections.abc import Callable, Awaitable

def feeder(get_next_item: Callable[[], str]) -> None:
    ...  # Body

def async_query(on_success: Callable[[int], None],
                on_error: Callable[[int, Exception], None]) -> None:
    ...  # Body

async def on_update(value: str) -> None:
    ...  # Body

callback: Callable[[str], Awaitable[None]] = on_update
```

The subscription syntax must always be used with exactly two values: the argument list and the return type. The argument list must be a list of types, a **`ParamSpec`**, **`Concatenate`**, or an ellipsis (`...`). The return type must be a single type.

If a literal ellipsis `...` is given as the argument list, it indicates that a callable with any arbitrary parameter list would be acceptable:

```
def concat(x: str, y: str) -> str:
    return x + y

x: Callable[..., str]
x = str     # OK
x = concat  # Also OK
```

`Callable` cannot express complex signatures such as functions that take a variadic number of arguments, overloaded functions, or functions that have keyword-only parameters. However, these signatures can be expressed by defining a **`Protocol`** class with a **`__call__()`** method:

```
from collections.abc import Iterable
from typing import Protocol

class Combiner(Protocol):
    def __call__(self, *vals: bytes, maxlen: int | None = None) ->
list[bytes]: ...

def batch_proc(data: Iterable[bytes], cb_results: Combiner) -> bytes:
    for item in data:
        ...

def good_cb(*vals: bytes, maxlen: int | None = None) -> list[bytes]:
    ...
def bad_cb(*vals: bytes, maxitems: int | None) -> list[bytes]:
    ...

batch_proc([], good_cb)  # OK
batch_proc([], bad_cb)   # Error! Argument 2 has incompatible type
because of
                         # different name and kind in the callback
```

Callables which take other callables as arguments may indicate that their parameter types are dependent on each other using **`ParamSpec`**. Additionally, if that callable adds or removes arguments from other callables, the **`Concatenate`** operator may be used. They take the form `Callable[ParamSpecVariable, ReturnType]` and `Callable[Concatenate[Arg1Type, Arg2Type, ..., ParamSpecVariable], ReturnType]` respectively.

Changed in version 3.10: `Callable` now supports **`ParamSpec`** and **`Concatenate`**. See **PEP 612** [https://peps.python.org/pep-0612/] for more details.

See also

The documentation for **`ParamSpec`** and **`Concatenate`** provides examples of usage in `Callable`.

Generics

Since type information about objects kept in containers cannot be statically inferred in a generic way, many container classes in the standard library support subscription to denote the expected types of container elements.

```
from collections.abc import Mapping, Sequence

class Employee: ...

# Sequence[Employee] indicates that all elements in the sequence
# must be instances of "Employee".
# Mapping[str, str] indicates that all keys and all values in the mapping
# must be strings.
def notify_by_email(employees: Sequence[Employee],
                    overrides: Mapping[str, str]) -> None: ...
```

Generic functions and classes can be parameterized by using type parameter syntax:

```
from collections.abc import Sequence

def first[T](l: Sequence[T]) -> T:  # Function is generic over the
TypeVar "T"
    return l[0]
```

Or by using the **`TypeVar`** factory directly:

```
from collections.abc import Sequence
from typing import TypeVar

U = TypeVar('U')                  # Declare type variable "U"

def second(l: Sequence[U]) -> U:  # Function is generic over the TypeVar
"U"
    return l[1]
```

Changed in version 3.12: Syntactic support for generics is new in Python 3.12.

Annotating tuples

For most containers in Python, the typing system assumes that all elements in the container will be of the same type. For example:

```
from collections.abc import Mapping

# Type checker will infer that all elements in ``x`` are meant to be ints
x: list[int] = []

# Type checker error: ``list`` only accepts a single type argument:
y: list[int, str] = [1, 'foo']

# Type checker will infer that all keys in ``z`` are meant to be strings,
# and that all values in ``z`` are meant to be either strings or ints
z: Mapping[str, str | int] = {}
```

`list` only accepts one type argument, so a type checker would emit an error on the `y` assignment above. Similarly, `Mapping` only accepts two type arguments: the first indicates the type of the keys, and the second indicates the type of the values.

Unlike most other Python containers, however, it is common in idiomatic Python code for tuples to have elements which are not all of the same type. For this reason, tuples are special-cased in Python’s typing system. `tuple` accepts *any number* of type arguments:

```
# OK: ``x`` is assigned to a tuple of length 1 where the sole element is
an int
x: tuple[int] = (5,)

# OK: ``y`` is assigned to a tuple of length 2;
# element 1 is an int, element 2 is a str
y: tuple[int, str] = (5, "foo")

# Error: the type annotation indicates a tuple of length 1,
# but ``z`` has been assigned to a tuple of length 3
z: tuple[int] = (1, 2, 3)
```

To denote a tuple which could be of *any* length, and in which all elements are of the same type `T`, use the literal ellipsis `...`: `tuple[T, ...]`. To denote an empty tuple, use `tuple[()]`. Using plain `tuple` as an annotation is equivalent to using `tuple[Any, ...]`:

```
x: tuple[int, ...] = (1, 2)
# These reassignments are OK: ``tuple[int, ...]`` indicates x can be of
```

```
any length
x = (1, 2, 3)
x = ()
# This reassignment is an error: all elements in ``x`` must be ints
x = ("foo", "bar")

# ``y`` can only ever be assigned to an empty tuple
y: tuple[()] = ()

z: tuple = ("foo", "bar")
# These reassignments are OK: plain ``tuple`` is equivalent to
``tuple[Any, ...]``
z = (1, 2, 3)
z = ()
```

The type of class objects

A variable annotated with `C` may accept a value of type `C`. In contrast, a variable annotated with `type[C]` (or deprecated **`typing.Type[C]`**) may accept values that are classes themselves – specifically, it will accept the *class object* of `C`. For example:

```
a = 3         # Has type ``int``
b = int       # Has type ``type[int]``
c = type(a)   # Also has type ``type[int]``
```

Note that `type[C]` is covariant:

```
class User: ...
class ProUser(User): ...
class TeamUser(User): ...

def make_new_user(user_class: type[User]) -> User:
    # ...
    return user_class()

make_new_user(User)      # OK
make_new_user(ProUser)   # Also OK: ``type[ProUser]`` is a subtype of
``type[User]``
make_new_user(TeamUser)  # Still fine
make_new_user(User())    # Error: expected ``type[User]`` but got
``User``
make_new_user(int)       # Error: ``type[int]`` is not a subtype of
``type[User]``
```

The only legal parameters for **`type`** are classes, **`Any`**, type variables, and unions of any of these types. For example:

```
def new_non_team_user(user_class: type[BasicUser | ProUser]): ...

new_non_team_user(BasicUser)  # OK
new_non_team_user(ProUser)    # OK
new_non_team_user(TeamUser)   # Error: ``type[TeamUser]`` is not a
subtype
                              # of ``type[BasicUser | ProUser]``
new_non_team_user(User)       # Also an error
```

`type[Any]` is equivalent to **`type`**, which is the root of Python's metaclass hierarchy.

Annotating generators and coroutines

A generator can be annotated using the generic type **`Generator[YieldType, SendType, ReturnType]`**. For example:

```
def echo_round() -> Generator[int, float, str]:
    sent = yield 0
    while sent >= 0:
        sent = yield round(sent)
    return 'Done'
```

Note that unlike many other generic classes in the standard library, the `SendType` of **`Generator`** behaves contravariantly, not covariantly or invariantly.

The `SendType` and `ReturnType` parameters default to **`None`**:

```
def infinite_stream(start: int) -> Generator[int]:
    while True:
        yield start
        start += 1
```

It is also possible to set these types explicitly:

```
def infinite_stream(start: int) -> Generator[int, None, None]:
    while True:
        yield start
        start += 1
```

Simple generators that only ever yield values can also be annotated as having a return type of either **`Iterable[YieldType]`** or **`Iterator[YieldType]`**:

```
def infinite_stream(start: int) -> Iterator[int]:
    while True:
        yield start
        start += 1
```

Async generators are handled in a similar fashion, but don’t expect a `ReturnType` type argument (**`AsyncGenerator[YieldType, SendType]`**). The `SendType` argument defaults to **`None`**, so the following definitions are equivalent:

```
async def infinite_stream(start: int) -> AsyncGenerator[int]:
    while True:
        yield start
        start = await increment(start)

async def infinite_stream(start: int) -> AsyncGenerator[int, None]:
    while True:
        yield start
        start = await increment(start)
```

As in the synchronous case, **`AsyncIterable[YieldType]`** and **`AsyncIterator[YieldType]`** are available as well:

```
async def infinite_stream(start: int) -> AsyncIterator[int]:
    while True:
        yield start
        start = await increment(start)
```

Coroutines can be annotated using **`Coroutine[YieldType, SendType, ReturnType]`**. Generic arguments correspond to those of **`Generator`**, for example:

```
from collections.abc import Coroutine
c: Coroutine[list[str], str, int]  # Some coroutine defined elsewhere
x = c.send('hi')                   # Inferred type of 'x' is list[str]
async def bar() -> None:
    y = await c                    # Inferred type of 'y' is int
```

User-defined generic types

A user-defined class can be defined as a generic class.

```
from logging import Logger

class LoggedVar[T]:
    def __init__(self, value: T, name: str, logger: Logger) -> None:
        self.name = name
        self.logger = logger
        self.value = value

    def set(self, new: T) -> None:
        self.log('Set ' + repr(self.value))
        self.value = new

    def get(self) -> T:
        self.log('Get ' + repr(self.value))
        return self.value

    def log(self, message: str) -> None:
        self.logger.info('%s: %s', self.name, message)
```

This syntax indicates that the class `LoggedVar` is parameterised around a single type variable `T` . This also makes `T` valid as a type within the class body.

Generic classes implicitly inherit from **`Generic`**. For compatibility with Python 3.11 and lower, it is also possible to inherit explicitly from **`Generic`** to indicate a generic class:

```
from typing import TypeVar, Generic

T = TypeVar('T')

class LoggedVar(Generic[T]):
    ...
```

Generic classes have **`__class_getitem__()`** methods, meaning they can be parameterised at runtime (e.g. `LoggedVar[int]` below):

```
from collections.abc import Iterable

def zero_all_vars(vars: Iterable[LoggedVar[int]]) -> None:
    for var in vars:
        var.set(0)
```

A generic type can have any number of type variables. All varieties of **`TypeVar`** are permissible as parameters for a generic type:

```
from typing import TypeVar, Generic, Sequence

class WeirdTrio[T, B: Sequence[bytes], S: (int, str)]:
    ...
```

```
OldT = TypeVar('OldT', contravariant=True)
OldB = TypeVar('OldB', bound=Sequence[bytes], covariant=True)
OldS = TypeVar('OldS', int, str)

class OldWeirdTrio(Generic[OldT, OldB, OldS]):
    ...
```

Each type variable argument to **`Generic`** must be distinct. This is thus invalid:

```
from typing import TypeVar, Generic
...

class Pair[M, M]:  # SyntaxError
    ...

T = TypeVar('T')

class Pair(Generic[T, T]):   # INVALID
    ...
```

Generic classes can also inherit from other classes:

```
from collections.abc import Sized

class LinkedList[T](Sized):
    ...
```

When inheriting from generic classes, some type parameters could be fixed:

```
from collections.abc import Mapping

class MyDict[T](Mapping[str, T]):
    ...
```

In this case `MyDict` has a single parameter, `T`.

Using a generic class without specifying type parameters assumes **`Any`** for each position. In the following example, `MyIterable` is not generic but implicitly inherits from `Iterable[Any]`:

```
from collections.abc import Iterable

class MyIterable(Iterable): # Same as Iterable[Any]
    ...
```

User-defined generic type aliases are also supported. Examples:

```
from collections.abc import Iterable

type Response[S] = Iterable[S] | int

# Return type here is same as Iterable[str] | int
def response(query: str) -> Response[str]:
    ...

type Vec[T] = Iterable[tuple[T, T]]

def inproduct[T: (int, float, complex)](v: Vec[T]) -> T: # Same as
Iterable[tuple[T, T]]
    return sum(x*y for x, y in v)
```

For backward compatibility, generic type aliases can also be created through a simple assignment:

```
from collections.abc import Iterable
from typing import TypeVar

S = TypeVar("S")
Response = Iterable[S] | int
```

Changed in version 3.7: `Generic` no longer has a custom metaclass.

Changed in version 3.12: Syntactic support for generics and type aliases is new in version 3.12. Previously, generic classes had to explicitly inherit from `Generic` or contain a type variable in one of their bases.

User-defined generics for parameter expressions are also supported via parameter specification variables in the form `[**P]`. The behavior is consistent with type variables' described above as parameter specification variables are treated by the **`typing`** module as a specialized type variable. The one exception to this is that a list of types can be used to substitute a `ParamSpec`:

```
>>> class Z[T, **P]: ...  # T is a TypeVar; P is a ParamSpec
...
>>> Z[int, [dict, float]]
__main__.Z[int, [dict, float]]
```

Classes generic over a `ParamSpec` can also be created using explicit inheritance from `Generic`. In this case, `**` is not used:

```
from typing import ParamSpec, Generic

P = ParamSpec('P')

class Z(Generic[P]):
    ...
```

Another difference between `TypeVar` and `ParamSpec` is that a generic with only one parameter specification variable will accept parameter lists in the forms `X[[Type1, Type2, ...]]` and also `X[Type1, Type2, ...]` for aesthetic reasons. Internally, the latter is converted to the former, so the following are equivalent:

```
>>> class X[**P]: ...
...
>>> X[int, str]
__main__.X[[int, str]]
>>> X[[int, str]]
__main__.X[[int, str]]
```

Note that generics with `ParamSpec` may not have correct `__parameters__` after substitution in some cases because they are intended primarily for static type checking.

Changed in version 3.10: `Generic` can now be parameterized over parameter expressions. See `ParamSpec` and **PEP 612** [https://peps.python.org/pep-0612/] for more details.

A user-defined generic class can have ABCs as base classes without a metaclass conflict. Generic metaclasses are not supported. The outcome of parameterizing generics is cached, and most types in the **`typing`** module are hashable and comparable for equality.

The `Any` type

A special kind of type is `Any`. A static type checker will treat every type as being compatible with `Any` and `Any` as being compatible with every type.

This means that it is possible to perform any operation or method call on a value of type `Any` and assign it to any variable:

```
from typing import Any
```

```
a: Any = None
a = []          # OK
a = 2           # OK

s: str = ''
s = a           # OK

def foo(item: Any) -> int:
    # Passes type checking; 'item' could be any type,
    # and that type might have a 'bar' method
    item.bar()
    ...
```

Notice that no type checking is performed when assigning a value of type `Any` to a more precise type. For example, the static type checker did not report an error when assigning `a` to `s` even though `s` was declared to be of type `str` and receives an `int` value at runtime!

Furthermore, all functions without a return type or parameter types will implicitly default to using `Any`:

```
def legacy_parser(text):
    ...
    return data

# A static type checker will treat the above
# as having the same signature as:
def legacy_parser(text: Any) -> Any:
    ...
    return data
```

This behavior allows `Any` to be used as an *escape hatch* when you need to mix dynamically and statically typed code.

Contrast the behavior of `Any` with the behavior of `object`. Similar to `Any`, every type is a subtype of `object`. However, unlike `Any`, the reverse is not true: `object` is *not* a subtype of every other type.

That means when the type of a value is `object`, a type checker will reject almost all operations on it, and assigning it to a variable (or using it as a return value) of a more specialized type is a type error. For example:

```
def hash_a(item: object) -> int:
    # Fails type checking; an object does not have a 'magic' method.
    item.magic()
    ...

def hash_b(item: Any) -> int:
    # Passes type checking
    item.magic()
    ...

# Passes type checking, since ints and strs are subclasses of object
hash_a(42)
hash_a("foo")

# Passes type checking, since Any is compatible with all types
hash_b(42)
hash_b("foo")
```

Use `object` to indicate that a value could be any type in a typesafe manner. Use `Any` to indicate that a value is dynamically typed.

Nominal vs structural subtyping

Initially **PEP 484** [https://peps.python.org/pep-0484/] defined the Python static type system as using *nominal subtyping*. This means that a class `A` is allowed where a class `B` is expected if and only if `A` is a subclass of `B`.

This requirement previously also applied to abstract base classes, such as `Iterable`. The problem with this approach is that a class had to be explicitly marked to support them, which is unpythonic and unlike what one would normally do in idiomatic dynamically typed Python code. For example, this conforms to **PEP 484** [https://peps.python.org/pep-0484/]:

```
from collections.abc import Sized, Iterable, Iterator

class Bucket(Sized, Iterable[int]):
    ...
    def __len__(self) -> int: ...
    def __iter__(self) -> Iterator[int]: ...
```

PEP 544 [https://peps.python.org/pep-0544/] allows to solve this problem by allowing users to write the above code without explicit base classes in the class definition, allowing `Bucket` to be implicitly considered a subtype of both `Sized` and `Iterable[int]` by static type checkers. This is known as *structural subtyping* (or static duck-typing):

```
from collections.abc import Iterator, Iterable

class Bucket:  # Note: no base classes
    ...
    def __len__(self) -> int: ...
    def __iter__(self) -> Iterator[int]: ...

def collect(items: Iterable[int]) -> int: ...
result = collect(Bucket())  # Passes type check
```

Moreover, by subclassing a special class **`Protocol`**, a user can define new custom protocols to fully enjoy structural subtyping (see examples below).

Module contents

The `typing` module defines the following classes, functions and decorators.

Special typing primitives

Special types

These can be used as types in annotations. They do not support subscription using `[]`.

typing.Any

Special type indicating an unconstrained type.

- Every type is compatible with **`Any`**.
- **`Any`** is compatible with every type.

Changed in version 3.11: **`Any`** can now be used as a base class. This can be useful for avoiding type checker errors with classes that can duck type anywhere or are highly dynamic.

typing.AnyStr

A constrained type variable.

Definition:

```
AnyStr = TypeVar('AnyStr', str, bytes)
```

`AnyStr` is meant to be used for functions that may accept **`str`** or **`bytes`** arguments but cannot allow the two to mix.

For example:

```
def concat(a: AnyStr, b: AnyStr) -> AnyStr:
    return a + b

concat("foo", "bar")    # OK, output has type 'str'
concat(b"foo", b"bar")  # OK, output has type 'bytes'
concat("foo", b"bar")   # Error, cannot mix str and bytes
```

Note that, despite its name, `AnyStr` has nothing to do with the **`Any`** type, nor does it mean "any string". In particular, `AnyStr` and `str | bytes` are different from each other and have different use cases:

```
# Invalid use of AnyStr:
# The type variable is used only once in the function signature,
# so cannot be "solved" by the type checker
def greet_bad(cond: bool) -> AnyStr:
    return "hi there!" if cond else b"greetings!"

# The better way of annotating this function:
def greet_proper(cond: bool) -> str | bytes:
    return "hi there!" if cond else b"greetings!"
```

Deprecated since version 3.13, will be removed in version 3.18: Deprecated in favor of the new type parameter syntax. Use `class A[T: (str, bytes)]: ...` instead of importing `AnyStr`. See **PEP 695** [https://peps.python.org/pep-0695/] for more details.

In Python 3.16, `AnyStr` will be removed from `typing.__all__`, and deprecation warnings will be emitted at runtime when it is accessed or imported from `typing`. `AnyStr` will be removed from `typing` in Python 3.18.

typing.LiteralString

Special type that includes only literal strings.

Any string literal is compatible with `LiteralString`, as is another `LiteralString`. However, an object typed as just `str` is not. A string created by composing `LiteralString`-typed objects is also acceptable as a `LiteralString`.

Example:

```
def run_query(sql: LiteralString) -> None:
    ...

def caller(arbitrary_string: str, literal_string: LiteralString) ->
None:
    run_query("SELECT * FROM students")  # OK
    run_query(literal_string)  # OK
    run_query("SELECT * FROM " + literal_string)  # OK
    run_query(arbitrary_string)  # type checker error
    run_query(  # type checker error
        f"SELECT * FROM students WHERE name = {arbitrary_string}"
    )
```

`LiteralString` is useful for sensitive APIs where arbitrary user-generated strings could generate problems. For example, the two cases above that generate type checker errors could be vulnerable to an SQL injection attack.

See **PEP 675** [https://peps.python.org/pep-0675/] for more details.

Added in version 3.11.

typing.Never
typing.NoReturn

`Never` and **`NoReturn`** represent the bottom type [https://en.wikipedia.org/wiki/Bottom_type], a type that has no members.

They can be used to indicate that a function never returns, such as `sys.exit()`:

```
from typing import Never  # or NoReturn

def stop() -> Never:
    raise RuntimeError('no way')
```

Or to define a function that should never be called, as there are no valid arguments, such as `assert_never()`:

```
from typing import Never  # or NoReturn

def never_call_me(arg: Never) -> None:
    pass

def int_or_str(arg: int | str) -> None:
    never_call_me(arg)  # type checker error
    match arg:
        case int():
            print("It's an int")
        case str():
            print("It's a str")
        case _:
            never_call_me(arg)  # OK, arg is of type Never (or
NoReturn)
```

`Never` and **`NoReturn`** have the same meaning in the type system and static type checkers treat both equivalently.

Added in version 3.6.2: Added `NoReturn`.

Added in version 3.11: Added `Never`.

typing.Self

Special type to represent the current enclosed class.

For example:

```
from typing import Self, reveal_type

class Foo:
    def return_self(self) -> Self:
        ...
        return self

class SubclassOfFoo(Foo): pass

reveal_type(Foo().return_self())  # Revealed type is "Foo"
reveal_type(SubclassOfFoo().return_self())  # Revealed type is
"SubclassOfFoo"
```

This annotation is semantically equivalent to the following, albeit in a more succinct fashion:

```
from typing import TypeVar

Self = TypeVar("Self", bound="Foo")

class Foo:
    def return_self(self: Self) -> Self:
        ...
        return self
```

In general, if something returns `self`, as in the above examples, you should use `Self` as the return annotation. If `Foo.return_self` was annotated as returning `"Foo"`, then the type checker would infer the object returned from `SubclassOfFoo.return_self` as being of type `Foo` rather than `SubclassOfFoo`.

Other common use cases include:

- **`classmethod`**s that are used as alternative constructors and return instances of the `cls` parameter.
- Annotating an **`__enter__()`** method which returns self.

You should not use `Self` as the return annotation if the method is not guaranteed to return an instance of a subclass when the class is subclassed:

```
class Eggs:
    # Self would be an incorrect return annotation here,
    # as the object returned is always an instance of Eggs,
    # even in subclasses
    def returns_eggs(self) -> "Eggs":
        return Eggs()
```

See **PEP 673** [https://peps.python.org/pep-0673/] for more details.

Added in version 3.11.

typing.TypeAlias

Special annotation for explicitly declaring a type alias.

For example:

```
from typing import TypeAlias

Factors: TypeAlias = list[int]
```

`TypeAlias` is particularly useful on older Python versions for annotating aliases that make use of forward references, as it can be hard for type checkers to distinguish these from normal variable assignments:

```
from typing import Generic, TypeAlias, TypeVar

T = TypeVar("T")

# "Box" does not exist yet,
# so we have to use quotes for the forward reference on Python <3.12.
# Using ``TypeAlias`` tells the type checker that this is a type alias
declaration,
# not a variable assignment to a string.
BoxOfStrings: TypeAlias = "Box[str]"

class Box(Generic[T]):
    @classmethod
    def make_box_of_strings(cls) -> BoxOfStrings: ...
```

See **PEP 613** [https://peps.python.org/pep-0613/] for more details.

Added in version 3.10.

Deprecated since version 3.12: `TypeAlias` is deprecated in favor of the `type` statement, which creates instances of `TypeAliasType` and which natively supports forward references. Note that while `TypeAlias` and `TypeAliasType` serve similar purposes and have similar names, they are distinct and the latter is not the type of the former. Removal of `TypeAlias` is not currently planned, but users are encouraged to migrate to `type` statements.

Special forms

These can be used as types in annotations. They all support subscription using `[]`, but each has a unique syntax.

class typing.Union

Union type; `Union[X, Y]` is equivalent to `X | Y` and means either X or Y.

To define a union, use e.g. `Union[int, str]` or the shorthand `int | str`. Using that shorthand is recommended. Details:

- The arguments must be types and there must be at least one.
- Unions of unions are flattened, e.g.:

  ```
  Union[Union[int, str], float] == Union[int, str, float]
  ```

  However, this does not apply to unions referenced through a type alias, to avoid forcing evaluation of the underlying `TypeAliasType`:

  ```
  type A = Union[int, str]
  Union[A, float] != Union[int, str, float]
  ```

- Unions of a single argument vanish, e.g.:

```
Union[int] == int  # The constructor actually returns int
```

- Redundant arguments are skipped, e.g.:

  ```
  Union[int, str, int] == Union[int, str] == int | str
  ```

- When comparing unions, the argument order is ignored, e.g.:

  ```
  Union[int, str] == Union[str, int]
  ```

- You cannot subclass or instantiate a `Union`.

- You cannot write `Union[X][Y]`.

Changed in version 3.7: Don't remove explicit subclasses from unions at runtime.

Changed in version 3.10: Unions can now be written as `X | Y`. See union type expressions.

Changed in version 3.14: **`types.UnionType`** is now an alias for **`Union`**, and both `Union[int, str]` and `int | str` create instances of the same class. To check whether an object is a `Union` at runtime, use `isinstance(obj, Union)`. For compatibility with earlier versions of Python, use `get_origin(obj) is typing.Union or get_origin(obj) is types.UnionType`.

typing.Optional

`Optional[X]` is equivalent to `X | None` (or `Union[X, None]`).

Note that this is not the same concept as an optional argument, which is one that has a default. An optional argument with a default does not require the `Optional` qualifier on its type annotation just because it is optional. For example:

```
def foo(arg: int = 0) -> None:
    ...
```

On the other hand, if an explicit value of `None` is allowed, the use of `Optional` is appropriate, whether the argument is optional or not. For example:

```
def foo(arg: Optional[int] = None) -> None:
    ...
```

Changed in version 3.10: Optional can now be written as `X | None`. See union type expressions.

typing.Concatenate

Special form for annotating higher-order functions.

`Concatenate` can be used in conjunction with Callable and **`ParamSpec`** to annotate a higher-order callable which adds, removes, or transforms parameters of another callable. Usage is in the form `Concatenate[Arg1Type, Arg2Type, ..., ParamSpecVariable]`. `Concatenate` is currently only valid when used as the first argument to a Callable. The last parameter to `Concatenate` must be a **`ParamSpec`** or ellipsis (`...`).

For example, to annotate a decorator `with_lock` which provides a **`threading.Lock`** to the decorated function, `Concatenate` can be used to indicate that `with_lock` expects a callable which takes in a `Lock` as the first argument, and returns a callable with a different type signature. In this case, the **`ParamSpec`** indicates that the returned callable's parameter types are dependent on the parameter types of the callable being passed in:

```
from collections.abc import Callable
from threading import Lock
from typing import Concatenate

# Use this lock to ensure that only one thread is executing a function
# at any time.
my_lock = Lock()

def with_lock[**P, R](f: Callable[Concatenate[Lock, P], R]) ->
Callable[P, R]:
    '''A type-safe decorator which provides a lock.'''
    def inner(*args: P.args, **kwargs: P.kwargs) -> R:
        # Provide the lock as the first argument.
        return f(my_lock, *args, **kwargs)
    return inner

@with_lock
def sum_threadsafe(lock: Lock, numbers: list[float]) -> float:
    '''Add a list of numbers together in a thread-safe manner.'''
    with lock:
        return sum(numbers)

# We don't need to pass in the lock ourselves thanks to the decorator.
sum_threadsafe([1.1, 2.2, 3.3])
```

Added in version 3.10.

See also

- **PEP 612** [https://peps.python.org/pep-0612/] – Parameter Specification Variables (the PEP which introduced `ParamSpec` and `Concatenate`)
- **`ParamSpec`**
- Annotating callable objects

typing.Literal

Special typing form to define “literal types”.

`Literal` can be used to indicate to type checkers that the annotated object has a value equivalent to one of the provided literals.

For example:

```
def validate_simple(data: Any) -> Literal[True]:  # always returns
True
    ...

type Mode = Literal['r', 'rb', 'w', 'wb']
def open_helper(file: str, mode: Mode) -> str:
    ...

open_helper('/some/path', 'r')      # Passes type check
open_helper('/other/path', 'typo')  # Error in type checker
```

`Literal[...]` cannot be subclassed. At runtime, an arbitrary value is allowed as type argument to `Literal[...]`, but type checkers may impose restrictions. See **PEP 586** [https://peps.python.org/pep-0586/] for more details about literal types.

Additional details:

- The arguments must be literal values and there must be at least one.

- Nested `Literal` types are flattened, e.g.:

  ```
  assert Literal[Literal[1, 2], 3] == Literal[1, 2, 3]
  ```

  However, this does not apply to `Literal` types referenced through a type alias, to avoid forcing evaluation of the underlying **`TypeAliasType`**:

```
type A = Literal[1, 2]
assert Literal[A, 3] != Literal[1, 2, 3]
```

- Redundant arguments are skipped, e.g.:

  ```
  assert Literal[1, 2, 1] == Literal[1, 2]
  ```

- When comparing literals, the argument order is ignored, e.g.:

  ```
  assert Literal[1, 2] == Literal[2, 1]
  ```

- You cannot subclass or instantiate a `Literal`.

- You cannot write `Literal[X][Y]`.

Added in version 3.8.

Changed in version 3.9.1: `Literal` now de-duplicates parameters. Equality comparisons of `Literal` objects are no longer order dependent. `Literal` objects will now raise a **`TypeError`** exception during equality comparisons if one of their parameters are not hashable.

typing.ClassVar

Special type construct to mark class variables.

As introduced in **PEP 526** [https://peps.python.org/pep-0526/], a variable annotation wrapped in ClassVar indicates that a given attribute is intended to be used as a class variable and should not be set on instances of that class. Usage:

```
class Starship:
    stats: ClassVar[dict[str, int]] = {} # class variable
    damage: int = 10                     # instance variable
```

`ClassVar` accepts only types and cannot be further subscribed.

`ClassVar` is not a class itself, and should not be used with **`isinstance()`** or **`issubclass()`**. **`ClassVar`** does not change Python runtime behavior, but it can be used by third-party type checkers. For example, a type checker might flag the following code as an error:

```
enterprise_d = Starship(3000)
enterprise_d.stats = {} # Error, setting class variable on instance
```

```
Starship.stats = {}     # This is OK
```

Added in version 3.5.3.

Changed in version 3.13: **`ClassVar`** can now be nested in **`Final`** and vice versa.

typing.Final

Special typing construct to indicate final names to type checkers.

Final names cannot be reassigned in any scope. Final names declared in class scopes cannot be overridden in subclasses.

For example:

```
MAX_SIZE: Final = 9000
MAX_SIZE += 1  # Error reported by type checker

class Connection:
    TIMEOUT: Final[int] = 10

class FastConnector(Connection):
    TIMEOUT = 1  # Error reported by type checker
```

There is no runtime checking of these properties. See **PEP 591** [https://peps.python.org/pep-0591/] for more details.

Added in version 3.8.

Changed in version 3.13: **`Final`** can now be nested in **`ClassVar`** and vice versa.

typing.Required

Special typing construct to mark a **`TypedDict`** key as required.

This is mainly useful for `total=False` TypedDicts. See **`TypedDict`** and **PEP 655** [https://peps.python.org/pep-0655/] for more details.

Added in version 3.11.

typing.NotRequired

Special typing construct to mark a **`TypedDict`** key as potentially missing.

See **`TypedDict`** and **PEP 655** [https://peps.python.org/pep-0655/] for more details.

Added in version 3.11.

typing.ReadOnly

A special typing construct to mark an item of a **`TypedDict`** as read-only.

For example:

```
class Movie(TypedDict):
   title: ReadOnly[str]
   year: int

def mutate_movie(m: Movie) -> None:
   m["year"] = 1999  # allowed
   m["title"] = "The Matrix"  # typechecker error
```

There is no runtime checking for this property.

See **`TypedDict`** and **PEP 705** [https://peps.python.org/pep-0705/] for more details.

Added in version 3.13.

typing.Annotated

Special typing form to add context-specific metadata to an annotation.

Add metadata `x` to a given type `T` by using the annotation `Annotated[T, x]`. Metadata added using `Annotated` can be used by static analysis tools or at runtime. At runtime, the metadata is stored in a **`__metadata__`** attribute.

If a library or tool encounters an annotation `Annotated[T, x]` and has no special logic for the metadata, it should ignore the metadata and simply treat the annotation as `T`. As such, `Annotated` can be useful for code that wants to use annotations for purposes outside Python’s static typing system.

Using `Annotated[T, x]` as an annotation still allows for static typechecking of `T`, as type checkers will simply ignore the metadata `x`. In this way, `Annotated` differs from the **`@no_type_check`** decorator, which can also be used for adding annotations outside the scope of the typing system, but completely disables typechecking for a function or class.

The responsibility of how to interpret the metadata lies with the tool or library encountering an `Annotated` annotation. A tool or library encountering an `Annotated` type can scan through the metadata elements to determine if they are of interest (e.g., using **`isinstance()`**).

Annotated[<type>, <metadata>]

Here is an example of how you might use `Annotated` to add metadata to type annotations if you were doing range analysis:

```
@dataclass
class ValueRange:
    lo: int
    hi: int

T1 = Annotated[int, ValueRange(-10, 5)]
T2 = Annotated[T1, ValueRange(-20, 3)]
```

The first argument to `Annotated` must be a valid type. Multiple metadata elements can be supplied as `Annotated` supports variadic arguments. The order of the metadata elements is preserved and matters for equality checks:

```
@dataclass
class ctype:
     kind: str

a1 = Annotated[int, ValueRange(3, 10), ctype("char")]
a2 = Annotated[int, ctype("char"), ValueRange(3, 10)]

assert a1 != a2  # Order matters
```

It is up to the tool consuming the annotations to decide whether the client is allowed to add multiple metadata elements to one annotation and how to merge those annotations.

Nested `Annotated` types are flattened. The order of the metadata elements starts with the innermost annotation:

```
assert Annotated[Annotated[int, ValueRange(3, 10)], ctype("char")] ==
Annotated[
    int, ValueRange(3, 10), ctype("char")
]
```

However, this does not apply to `Annotated` types referenced through a type alias, to avoid forcing evaluation of the underlying **TypeAliasType**:

```
type From3To10[T] = Annotated[T, ValueRange(3, 10)]
assert Annotated[From3To10[int], ctype("char")] != Annotated[
   int, ValueRange(3, 10), ctype("char")
]
```

Duplicated metadata elements are not removed:

```
assert Annotated[int, ValueRange(3, 10)] != Annotated[
    int, ValueRange(3, 10), ValueRange(3, 10)
]
```

`Annotated` can be used with nested and generic aliases:

```
@dataclass
class MaxLen:
    value: int

type Vec[T] = Annotated[list[tuple[T, T]], MaxLen(10)]

# When used in a type annotation, a type checker will treat "V"
the same as
# ``Annotated[list[tuple[int, int]], MaxLen(10)]``:
type V = Vec[int]
```

`Annotated` cannot be used with an unpacked **TypeVarTuple**:

```
type Variadic[*Ts] = Annotated[*Ts, Ann1] = Annotated[T1, T2, T3, ...,
Ann1]  # NOT valid
```

where `T1`, `T2`, ... are **TypeVars**. This is invalid as only one type should be passed to Annotated.

By default, **get_type_hints()** strips the metadata from annotations. Pass `include_extras=True` to have the metadata preserved:

```
>>> from typing import Annotated, get_type_hints
>>> def func(x: Annotated[int, "metadata"]) -> None: pass
...
>>> get_type_hints(func)
{'x': <class 'int'>, 'return': <class 'NoneType'>}
>>> get_type_hints(func, include_extras=True)
```

```
{'x': typing.Annotated[int, 'metadata'], 'return': <class
'NoneType'>}
```

At runtime, the metadata associated with an `Annotated` type can be retrieved via the **`__metadata__`** attribute:

```
>>> from typing import Annotated
>>> X = Annotated[int, "very", "important", "metadata"]
>>> X
typing.Annotated[int, 'very', 'important', 'metadata']
>>> X.__metadata__
('very', 'important', 'metadata')
```

If you want to retrieve the original type wrapped by `Annotated`, use the **`__origin__`** attribute:

```
>>> from typing import Annotated, get_origin
>>> Password = Annotated[str, "secret"]
>>> Password.__origin__
<class 'str'>
```

Note that using **`get_origin()`** will return `Annotated` itself:

```
>>> get_origin(Password)
typing.Annotated
```

See also

PEP 593 [https://peps.python.org/pep-0593/] - Flexible function and variable annotations

The PEP introducing `Annotated` to the standard library.

Added in version 3.9.

typing.TypeIs

Special typing construct for marking user-defined type predicate functions.

`TypeIs` can be used to annotate the return type of a user-defined type predicate function. `TypeIs` only accepts a single type argument. At runtime, functions marked this way should return a boolean and take at least one positional argument.

`TypeIs` aims to benefit *type narrowing* – a technique used by static type checkers to determine a more precise type of an expression within a program’s code flow. Usually type narrowing is done by analyzing conditional code flow and applying the narrowing to a block of code. The conditional expression here is sometimes referred to as a “type predicate”:

```
def is_str(val: str | float):
    # "isinstance" type predicate
    if isinstance(val, str):
        # Type of ``val`` is narrowed to ``str``
        ...
    else:
        # Else, type of ``val`` is narrowed to ``float``.
        ...
```

Sometimes it would be convenient to use a user-defined boolean function as a type predicate. Such a function should use `TypeIs[...]` or **`TypeGuard`** as its return type to alert static type checkers to this intention. `TypeIs` usually has more intuitive behavior than `TypeGuard`, but it cannot be used when the input and output types are incompatible (e.g., `list[object]` to `list[int]`) or when the function does not return `True` for all instances of the narrowed type.

Using `-> TypeIs[NarrowedType]` tells the static type checker that for a given function:

1. The return value is a boolean.
2. If the return value is `True`, the type of its argument is the intersection of the argument’s original type and `NarrowedType`.
3. If the return value is `False`, the type of its argument is narrowed to exclude `NarrowedType`.

For example:

```
from typing import assert_type, final, TypeIs

class Parent: pass
class Child(Parent): pass
@final
class Unrelated: pass

def is_parent(val: object) -> TypeIs[Parent]:
    return isinstance(val, Parent)

def run(arg: Child | Unrelated):
    if is_parent(arg):
        # Type of ``arg`` is narrowed to the intersection
        # of ``Parent`` and ``Child``, which is equivalent to
        # ``Child``.
        assert_type(arg, Child)
    else:
        # Type of ``arg`` is narrowed to exclude ``Parent``,
        # so only ``Unrelated`` is left.
        assert_type(arg, Unrelated)
```

The type inside `TypeIs` must be consistent with the type of the function's argument; if it is not, static type checkers will raise an error. An incorrectly written `TypeIs` function can lead to unsound behavior in the type system; it is the user's responsibility to write such functions in a type-safe manner.

If a `TypeIs` function is a class or instance method, then the type in `TypeIs` maps to the type of the second parameter (after `cls` or `self`).

In short, the form `def foo(arg: TypeA) -> TypeIs[TypeB]: ...`, means that if `foo(arg)` returns `True`, then `arg` is an instance of `TypeB`, and if it returns `False`, it is not an instance of `TypeB`.

`TypeIs` also works with type variables. For more information, see **PEP 742** [https://peps.python.org/pep-0742/] (Narrowing types with `TypeIs`).

Added in version 3.13.

typing.TypeGuard

Special typing construct for marking user-defined type predicate functions.

Type predicate functions are user-defined functions that return whether their argument is an instance of a particular type. `TypeGuard` works similarly to `TypeIs`, but has subtly different effects on type checking behavior (see below).

Using `-> TypeGuard` tells the static type checker that for a given function:

1. The return value is a boolean.
2. If the return value is `True`, the type of its argument is the type inside `TypeGuard`.

`TypeGuard` also works with type variables. See **PEP 647** [https://peps.python.org/pep-0647/] for more details.

For example:

```
def is_str_list(val: list[object]) -> TypeGuard[list[str]]:
    '''Determines whether all objects in the list are strings'''
    return all(isinstance(x, str) for x in val)

def func1(val: list[object]):
    if is_str_list(val):
        # Type of ``val`` is narrowed to ``list[str]``.
        print(" ".join(val))
    else:
        # Type of ``val`` remains as ``list[object]``.
        print("Not a list of strings!")
```

`TypeIs` and `TypeGuard` differ in the following ways:

- `TypeIs` requires the narrowed type to be a subtype of the input type, while `TypeGuard` does not. The main reason is to allow for things like narrowing `list[object]` to `list[str]` even though the latter is not a subtype of the former, since `list` is invariant.
- When a `TypeGuard` function returns `True`, type checkers narrow the type of the variable to exactly the `TypeGuard` type. When a `TypeIs` function returns `True`, type checkers can infer a more precise type combining the previously known type of the variable with the `TypeIs` type. (Technically, this is known as an intersection type.)
- When a `TypeGuard` function returns `False`, type checkers cannot narrow the type of the variable at all. When a `TypeIs` function returns `False`, type checkers can narrow the type of the variable to exclude the `TypeIs` type.

Added in version 3.10.

typing.Unpack

Typing operator to conceptually mark an object as having been unpacked.

For example, using the unpack operator `*` on a type variable tuple is equivalent to using `Unpack` to mark the type variable tuple as having been unpacked:

```
Ts = TypeVarTuple('Ts')
tup: tuple[*Ts]
# Effectively does:
tup: tuple[Unpack[Ts]]
```

In fact, `Unpack` can be used interchangeably with `*` in the context of [`typing.TypeVarTuple`](#) and [`builtins.tuple`](#) types. You might see `Unpack` being used explicitly in older versions of Python, where `*` couldn't be used in certain places:

```
# In older versions of Python, TypeVarTuple and Unpack
# are located in the `typing_extensions` backports package.
from typing_extensions import TypeVarTuple, Unpack

Ts = TypeVarTuple('Ts')
tup: tuple[*Ts]         # Syntax error on Python <= 3.10!
tup: tuple[Unpack[Ts]]  # Semantically equivalent, and backwards-
compatible
```

`Unpack` can also be used along with **`typing.TypedDict`** for typing `**kwargs` in a function signature:

```
from typing import TypedDict, Unpack

class Movie(TypedDict):
    name: str
    year: int

# This function expects two keyword arguments - `name` of type `str`
# and `year` of type `int`.
def foo(**kwargs: Unpack[Movie]): ...
```

See **PEP 692** [https://peps.python.org/pep-0692/] for more details on using `Unpack` for `**kwargs` typing.

Added in version 3.11.

Building generic types and type aliases

The following classes should not be used directly as annotations. Their intended purpose is to be building blocks for creating generic types and type aliases.

These objects can be created through special syntax (type parameter lists and the **`type`** statement). For compatibility with Python 3.11 and earlier, they can also be created without the dedicated syntax, as documented below.

class typing.Generic

Abstract base class for generic types.

A generic type is typically declared by adding a list of type parameters after the class name:

```
class Mapping[KT, VT]:
    def __getitem__(self, key: KT) -> VT:
        ...
        # Etc.
```

Such a class implicitly inherits from `Generic`. The runtime semantics of this syntax are discussed in the Language Reference.

This class can then be used as follows:

```
def lookup_name[X, Y](mapping: Mapping[X, Y], key: X, default: Y) ->
Y:
    try:
        return mapping[key]
    except KeyError:
        return default
```

Here the brackets after the function name indicate a generic function.

For backwards compatibility, generic classes can also be declared by explicitly inheriting from `Generic`. In this case, the type parameters must be declared separately:

```
KT = TypeVar('KT')
VT = TypeVar('VT')

class Mapping(Generic[KT, VT]):
    def __getitem__(self, key: KT) -> VT:
        ...
        # Etc.
```

class typing.TypeVar(*name*, **constraints*, *bound=None*, *covariant=False*, *contravariant=False*, *infer_variance=False*, *default=typing.NoDefault*)

Type variable.

The preferred way to construct a type variable is via the dedicated syntax for generic functions, generic classes, and generic type aliases:

```
class Sequence[T]:  # T is a TypeVar
    ...
```

This syntax can also be used to create bounded and constrained type variables:

```
class StrSequence[S: str]:  # S is a TypeVar with a `str` upper bound;
    ...                     # we can say that S is "bounded by `str`"


class StrOrBytesSequence[A: (str, bytes)]:  # A is a TypeVar 
constrained to str or bytes
    ...
```

However, if desired, reusable type variables can also be constructed manually, like so:

```
T = TypeVar('T')  # Can be anything
S = TypeVar('S', bound=str)  # Can be any subtype of str
A = TypeVar('A', str, bytes)  # Must be exactly str or bytes
```

Type variables exist primarily for the benefit of static type checkers. They serve as the parameters for generic types as well as for generic function and type alias definitions. See **`Generic`** for more information on generic types. Generic functions work as follows:

```
def repeat[T](x: T, n: int) -> Sequence[T]:
    """Return a list containing n references to x."""
    return [x]*n


def print_capitalized[S: str](x: S) -> S:
    """Print x capitalized, and return x."""
    print(x.capitalize())
    return x


def concatenate[A: (str, bytes)](x: A, y: A) -> A:
    """Add two strings or bytes objects together."""
    return x + y
```

Note that type variables can be *bounded*, *constrained*, or neither, but cannot be both bounded *and* constrained.

The variance of type variables is inferred by type checkers when they are created through the type parameter syntax or when `infer_variance=True` is passed. Manually created type variables may be explicitly marked covariant or contravariant by passing `covariant=True` or `contravariant=True`. By default, manually created type variables are invariant. See **PEP 484** [https://peps.python.org/pep-0484/] and **PEP 695** [https://peps.python.org/pep-0695/] for more details.

Bounded type variables and constrained type variables have different semantics in several important ways. Using a *bounded* type variable means that the `TypeVar` will be solved using the most specific type possible:

```
x = print_capitalized('a string')
reveal_type(x)  # revealed type is str

class StringSubclass(str):
    pass

y = print_capitalized(StringSubclass('another string'))
reveal_type(y)  # revealed type is StringSubclass

z = print_capitalized(45)  # error: int is not a subtype of str
```

The upper bound of a type variable can be a concrete type, abstract type (ABC or Protocol), or even a union of types:

```
# Can be anything with an __abs__ method
def print_abs[T: SupportsAbs](arg: T) -> None:
    print("Absolute value:", abs(arg))

U = TypeVar('U', bound=str|bytes)  # Can be any subtype of the union
str|bytes
V = TypeVar('V', bound=SupportsAbs)  # Can be anything with an __abs__
method
```

Using a *constrained* type variable, however, means that the `TypeVar` can only ever be solved as being exactly one of the constraints given:

```
a = concatenate('one', 'two')
reveal_type(a)  # revealed type is str

b = concatenate(StringSubclass('one'), StringSubclass('two'))
reveal_type(b)  # revealed type is str, despite StringSubclass being
passed in

c = concatenate('one', b'two')  # error: type variable 'A' can be
either str or bytes in a function call, but not both
```

At runtime, `isinstance(x, T)` will raise `TypeError`.

__name__
: The name of the type variable.

__covariant__
: Whether the type var has been explicitly marked as covariant.

__contravariant__
: Whether the type var has been explicitly marked as contravariant.

__infer_variance__
: Whether the type variable’s variance should be inferred by type checkers.

  Added in version 3.12.

__bound__
: The upper bound of the type variable, if any.

  Changed in version 3.12: For type variables created through type parameter syntax, the bound is evaluated only when the attribute is accessed, not when the type variable is created (see Lazy evaluation).

evaluate_bound()
: An evaluate function corresponding to the **`__bound__`** attribute. When called directly, this method supports only the **`VALUE`** format, which is equivalent to accessing the **`__bound__`** attribute directly, but the method object can be passed to **`annotationlib.call_evaluate_function()`** to evaluate the value in a different format.

  Added in version 3.14.

__constraints__
: A tuple containing the constraints of the type variable, if any.

  Changed in version 3.12: For type variables created through type parameter syntax, the constraints are evaluated only when the attribute is accessed, not when the type variable is created (see Lazy evaluation).

evaluate_constraints()

An evaluate function corresponding to the **`__constraints__`** attribute. When called directly, this method supports only the **`VALUE`** format, which is equivalent to accessing the **`__constraints__`** attribute directly, but the method object can be passed to **`annotationlib.call_evaluate_function()`** to evaluate the value in a different format.

Added in version 3.14.

__default__

The default value of the type variable, or **`typing.NoDefault`** if it has no default.

Added in version 3.13.

evaluate_default()

An evaluate function corresponding to the **`__default__`** attribute. When called directly, this method supports only the **`VALUE`** format, which is equivalent to accessing the **`__default__`** attribute directly, but the method object can be passed to **`annotationlib.call_evaluate_function()`** to evaluate the value in a different format.

Added in version 3.14.

has_default()

Return whether or not the type variable has a default value. This is equivalent to checking whether **`__default__`** is not the **`typing.NoDefault`** singleton, except that it does not force evaluation of the lazily evaluated default value.

Added in version 3.13.

Changed in version 3.12: Type variables can now be declared using the type parameter syntax introduced by **PEP 695** [https://peps.python.org/pep-0695/]. The `infer_variance` parameter was added.

Changed in version 3.13: Support for default values was added.

class typing.TypeVarTuple(*name*, ***, *default=typing.NoDefault*)

Type variable tuple. A specialized form of type variable that enables *variadic* generics.

Type variable tuples can be declared in type parameter lists using a single asterisk (`*`) before the name:

```
def move_first_element_to_last[T, *Ts](tup: tuple[T, *Ts]) ->
tuple[*Ts, T]:
    return (*tup[1:], tup[0])
```

Or by explicitly invoking the `TypeVarTuple` constructor:

```
T = TypeVar("T")
Ts = TypeVarTuple("Ts")

def move_first_element_to_last(tup: tuple[T, *Ts]) -> tuple[*Ts, T]:
    return (*tup[1:], tup[0])
```

A normal type variable enables parameterization with a single type. A type variable tuple, in contrast, allows parameterization with an *arbitrary* number of types by acting like an *arbitrary* number of type variables wrapped in a tuple. For example:

```
# T is bound to int, Ts is bound to ()
# Return value is (1,), which has type tuple[int]
move_first_element_to_last(tup=(1,))

# T is bound to int, Ts is bound to (str,)
# Return value is ('spam', 1), which has type tuple[str, int]
move_first_element_to_last(tup=(1, 'spam'))

# T is bound to int, Ts is bound to (str, float)
# Return value is ('spam', 3.0, 1), which has type tuple[str, float,
int]
move_first_element_to_last(tup=(1, 'spam', 3.0))

# This fails to type check (and fails at runtime)
# because tuple[()] is not compatible with tuple[T, *Ts]
# (at least one element is required)
move_first_element_to_last(tup=())
```

Note the use of the unpacking operator `*` in `tuple[T, *Ts]`. Conceptually, you can think of `Ts` as a tuple of type variables `(T1, T2, ...)`. `tuple[T, *Ts]` would then become `tuple[T, *(T1, T2, ...)]`, which is equivalent to `tuple[T, T1, T2, ...]`. (Note that in older versions of Python, you might see this written using **`Unpack`** instead, as `Unpack[Ts]`.)

Type variable tuples must *always* be unpacked. This helps distinguish type variable tuples from normal type variables:

```
x: Ts          # Not valid
x: tuple[Ts]   # Not valid
x: tuple[*Ts]  # The correct way to do it
```

Type variable tuples can be used in the same contexts as normal type variables. For example, in class definitions, arguments, and return types:

```
class Array[*Shape]:
    def __getitem__(self, key: tuple[*Shape]) -> float: ...
    def __abs__(self) -> "Array[*Shape]": ...
    def get_shape(self) -> tuple[*Shape]: ...
```

Type variable tuples can be happily combined with normal type variables:

```
class Array[DType, *Shape]:  # This is fine
    pass

class Array2[*Shape, DType]:  # This would also be fine
    pass

class Height: ...
class Width: ...

float_array_1d: Array[float, Height] = Array()     # Totally fine
int_array_2d: Array[int, Height, Width] = Array()  # Yup, fine too
```

However, note that at most one type variable tuple may appear in a single list of type arguments or type parameters:

```
x: tuple[*Ts, *Ts]            # Not valid
class Array[*Shape, *Shape]:  # Not valid
    pass
```

Finally, an unpacked type variable tuple can be used as the type annotation of `*args`:

```
def call_soon[*Ts](
    callback: Callable[[*Ts], None],
    *args: *Ts
) -> None:
    ...
    callback(*args)
```

In contrast to non-unpacked annotations of `*args` - e.g. `*args: int`, which would specify that *all* arguments are `int` - `*args: *Ts` enables reference to the types of the *individual* arguments in `*args`. Here, this allows us to ensure the types of the `*args` passed to `call_soon` match the types of the (positional) arguments of `callback`.

See **PEP 646** [https://peps.python.org/pep-0646/] for more details on type variable tuples.

__name__

The name of the type variable tuple.

__default__

The default value of the type variable tuple, or **`typing.NoDefault`** if it has no default.

Added in version 3.13.

evaluate_default()

An evaluate function corresponding to the **`__default__`** attribute. When called directly, this method supports only the **`VALUE`** format, which is equivalent to accessing the **`__default__`** attribute directly, but the method object can be passed to **`annotationlib.call_evaluate_function()`** to evaluate the value in a different format.

Added in version 3.14.

has_default()

Return whether or not the type variable tuple has a default value. This is equivalent to checking whether **`__default__`** is not the **`typing.NoDefault`** singleton, except that it does not force evaluation of the lazily evaluated default value.

Added in version 3.13.

Added in version 3.11.

Changed in version 3.12: Type variable tuples can now be declared using the type parameter syntax introduced by **PEP 695** [https://peps.python.org/pep-0695/].

Changed in version 3.13: Support for default values was added.

class typing.ParamSpec(*name*, ***, *bound=None*, *covariant=False*, *contravariant=False*, *default=typing.NoDefault*)

Parameter specification variable. A specialized version of type variables.

In type parameter lists, parameter specifications can be declared with two asterisks (`**`):

```
type IntFunc[**P] = Callable[P, int]
```

For compatibility with Python 3.11 and earlier, `ParamSpec` objects can also be created as follows:

```
P = ParamSpec('P')
```

Parameter specification variables exist primarily for the benefit of static type checkers. They are used to forward the parameter types of one callable to another callable – a pattern commonly found in higher order functions and decorators. They are only valid when used in `Concatenate`, or as the first argument to `Callable`, or as parameters for user-defined Generics. See **`Generic`** for more information on generic types.

For example, to add basic logging to a function, one can create a decorator `add_logging` to log function calls. The parameter specification variable tells the type checker that the callable passed into the decorator and the new callable returned by it have inter-dependent type parameters:

```
from collections.abc import Callable
import logging

def add_logging[T, **P](f: Callable[P, T]) -> Callable[P, T]:
    '''A type-safe decorator to add logging to a function.'''
    def inner(*args: P.args, **kwargs: P.kwargs) -> T:
        logging.info(f'{f.__name__} was called')
        return f(*args, **kwargs)
    return inner

@add_logging
def add_two(x: float, y: float) -> float:
    '''Add two numbers together.'''
    return x + y
```

Without `ParamSpec`, the simplest way to annotate this previously was to use a **`TypeVar`** with upper bound `Callable[..., Any]`. However this causes two problems:

1. The type checker can't type check the `inner` function because `*args` and `**kwargs` have to be typed **`Any`**.
2. **`cast()`** may be required in the body of the `add_logging` decorator when returning the `inner` function, or the static type checker must be told to ignore the `return inner`.

args

kwargs

Since `ParamSpec` captures both positional and keyword parameters, `P.args` and `P.kwargs` can be used to split a `ParamSpec` into its components. `P.args` represents the tuple of positional parameters in a given call and should only be used to annotate `*args`. `P.kwargs` represents the mapping of keyword parameters to their values in a given call, and should be only be used to annotate `**kwargs`. Both attributes require the annotated parameter to be in scope. At runtime, `P.args` and `P.kwargs` are instances respectively of **`ParamSpecArgs`** and **`ParamSpecKwargs`**.

__name__

The name of the parameter specification.

__default__

The default value of the parameter specification, or **`typing.NoDefault`** if it has no default.

Added in version 3.13.

evaluate_default()

An evaluate function corresponding to the **`__default__`** attribute. When called directly, this method supports only the **`VALUE`** format, which is equivalent to accessing the **`__default__`** attribute directly, but the method object can be passed to **`annotationlib.call_evaluate_function()`** to evaluate the value in a different format.

Added in version 3.14.

has_default()

Return whether or not the parameter specification has a default value. This is equivalent to checking whether **`__default__`** is not the **`typing.NoDefault`** singleton, except that it does not force evaluation of the lazily evaluated default value.

Added in version 3.13.

Parameter specification variables created with `covariant=True` or `contravariant=True` can be used to declare covariant or contravariant generic types. The `bound` argument is also accepted, similar to **`TypeVar`**. However the actual semantics of these keywords are yet to be decided.

Added in version 3.10.

Changed in version 3.12: Parameter specifications can now be declared using the type parameter syntax introduced by **PEP 695** [https://peps.python.org/pep-0695/].

Changed in version 3.13: Support for default values was added.

Note

Only parameter specification variables defined in global scope can be pickled.

See also

- **PEP 612** [https://peps.python.org/pep-0612/] – Parameter Specification Variables (the PEP which introduced `ParamSpec` and `Concatenate`)
- **`Concatenate`**
- Annotating callable objects

typing.ParamSpecArgs
typing.ParamSpecKwargs

Arguments and keyword arguments attributes of a **`ParamSpec`**. The `P.args` attribute of a `ParamSpec` is an instance of `ParamSpecArgs`, and `P.kwargs` is an instance of `ParamSpecKwargs`. They are intended for runtime introspection and have no special meaning to static type checkers.

Calling **`get_origin()`** on either of these objects will return the original `ParamSpec`:

```
>>> from typing import ParamSpec, get_origin
>>> P = ParamSpec("P")
>>> get_origin(P.args) is P
True
>>> get_origin(P.kwargs) is P
True
```

Added in version 3.10.

class typing.TypeAliasType(*name*, *value*, ***, *type_params=()*)

The type of type aliases created through the **`type`** statement.

Example:

```
>>> type Alias = int
>>> type(Alias)
<class 'typing.TypeAliasType'>
```

Added in version 3.12.

__name__

The name of the type alias:

```
>>> type Alias = int
>>> Alias.__name__
'Alias'
```

__module__

The name of the module in which the type alias was defined:

```
>>> type Alias = int
>>> Alias.__module__
'__main__'
```

__type_params__

The type parameters of the type alias, or an empty tuple if the alias is not generic:

```
>>> type ListOrSet[T] = list[T] | set[T]
>>> ListOrSet.__type_params__
(T,)
>>> type NotGeneric = int
>>> NotGeneric.__type_params__
()
```

__value__

The type alias’s value. This is lazily evaluated, so names used in the definition of the alias are not resolved until the `__value__` attribute is accessed:

```
>>> type Mutually = Recursive
>>> type Recursive = Mutually
>>> Mutually
Mutually
>>> Recursive
Recursive
>>> Mutually.__value__
Recursive
>>> Recursive.__value__
Mutually
```

evaluate_value()

An evaluate function corresponding to the **`__value__`** attribute. When called directly, this method supports only the **`VALUE`** format, which is equivalent to accessing the **`__value__`** attribute directly, but the method object can be passed to **`annotationlib.call_evaluate_function()`** to evaluate the value in a different format:

```
>>> type Alias = undefined
>>> Alias.__value__
Traceback (most recent call last):
...
NameError: name 'undefined' is not defined
>>> from annotationlib import Format, call_evaluate_function
>>> Alias.evaluate_value(Format.VALUE)
Traceback (most recent call last):
...
NameError: name 'undefined' is not defined
>>> call_evaluate_function(Alias.evaluate_value, Format.FORWARDREF)
ForwardRef('undefined')
```

Added in version 3.14.

Unpacking

Type aliases support star unpacking using the `*Alias` syntax. This is equivalent to using `Unpack[Alias]` directly:

```
>>> type Alias = tuple[int, str]
>>> type Unpacked = tuple[bool, *Alias]
>>> Unpacked.__value__
tuple[bool, typing.Unpack[Alias]]
```

Added in version 3.14.

Other special directives

These functions and classes should not be used directly as annotations. Their intended purpose is to be building blocks for creating and declaring types.

class typing.NamedTuple

Typed version of [`collections.namedtuple()`](collections.namedtuple()).

Usage:

```
class Employee(NamedTuple):
    name: str
    id: int
```

This is equivalent to:

```
Employee = collections.namedtuple('Employee', ['name', 'id'])
```

To give a field a default value, you can assign to it in the class body:

```
class Employee(NamedTuple):
    name: str
    id: int = 3

employee = Employee('Guido')
assert employee.id == 3
```

Fields with a default value must come after any fields without a default.

The resulting class has an extra attribute `__annotations__` giving a dict that maps the field names to the field types. (The field names are in the `_fields` attribute and the default values are in the `_field_defaults` attribute, both of which are part of the **`namedtuple()`** API.)

`NamedTuple` subclasses can also have docstrings and methods:

```
class Employee(NamedTuple):
    """Represents an employee."""
    name: str
    id: int = 3

    def __repr__(self) -> str:
        return f'<Employee {self.name}, id={self.id}>'
```

`NamedTuple` subclasses can be generic:

```
class Group[T](NamedTuple):
    key: T
    group: list[T]
```

Backward-compatible usage:

```
# For creating a generic NamedTuple on Python 3.11
T = TypeVar("T")

class Group(NamedTuple, Generic[T]):
    key: T
    group: list[T]

# A functional syntax is also supported
Employee = NamedTuple('Employee', [('name', str), ('id', int)])
```

Changed in version 3.6: Added support for **PEP 526** [https://peps.python.org/pep-0526/] variable annotation syntax.

Changed in version 3.6.1: Added support for default values, methods, and docstrings.

Changed in version 3.8: The `_field_types` and `__annotations__` attributes are now regular dictionaries instead of instances of `OrderedDict`.

Changed in version 3.9: Removed the `_field_types` attribute in favor of the more standard `__annotations__` attribute which has the same information.

Changed in version 3.9: `NamedTuple` is now a function rather than a class. It can still be used as a class base, as described above.

Changed in version 3.11: Added support for generic namedtuples.

Changed in version 3.14: Using **`super()`** (and the `__class__` closure variable) in methods of `NamedTuple` subclasses is unsupported and causes a `TypeError`.

Deprecated since version 3.13, will be removed in version 3.15: The undocumented keyword argument syntax for creating NamedTuple classes (`NT = NamedTuple("NT", x=int)`) is deprecated, and will be disallowed in 3.15. Use the class-based syntax or the functional syntax instead.

Deprecated since version 3.13, will be removed in version 3.15: When using the functional syntax to create a NamedTuple class, failing to pass a value to the 'fields' parameter (`NT = NamedTuple("NT")`) is deprecated. Passing `None` to the 'fields' parameter (`NT = NamedTuple("NT", None)`) is also deprecated. Both will be disallowed in Python 3.15. To create a NamedTuple class with 0 fields, use `class NT(NamedTuple): pass` or `NT = NamedTuple("NT", [])`.

class typing.NewType(*name*, *tp*)

Helper class to create low-overhead distinct types.

A `NewType` is considered a distinct type by a typechecker. At runtime, however, calling a `NewType` returns its argument unchanged.

Usage:

```
UserId = NewType('UserId', int)  # Declare the NewType "UserId"
first_user = UserId(1)  # "UserId" returns the argument unchanged at
runtime
```

__module__

The name of the module in which the new type is defined.

__name__

The name of the new type.

__supertype__

The type that the new type is based on.

Added in version 3.5.2.

Changed in version 3.10: `NewType` is now a class rather than a function.

class typing.Protocol(*Generic*)

Base class for protocol classes.

Protocol classes are defined like this:

```
class Proto(Protocol):
    def meth(self) -> int:
        ...
```

Such classes are primarily used with static type checkers that recognize structural subtyping (static duck-typing), for example:

```
class C:
    def meth(self) -> int:
        return 0

def func(x: Proto) -> int:
    return x.meth()

func(C())  # Passes static type check
```

See **PEP 544** [https://peps.python.org/pep-0544/] for more details. Protocol classes decorated with `runtime_checkable()` (described later) act as simple-minded runtime protocols that check only the presence of given attributes, ignoring their type signatures. Protocol classes without this decorator cannot be used as the second argument to `isinstance()` or `issubclass()`.

Protocol classes can be generic, for example:

```
class GenProto[T](Protocol):
    def meth(self) -> T:
        ...
```

In code that needs to be compatible with Python 3.11 or older, generic Protocols can be written as follows:

```
T = TypeVar("T")

class GenProto(Protocol[T]):
    def meth(self) -> T:
        ...
```

Added in version 3.8.

@typing.runtime_checkable

Mark a protocol class as a runtime protocol.

Such a protocol can be used with `isinstance()` and `issubclass()`. This allows a simple-minded structural check, very similar to “one trick ponies” in `collections.abc` such as `Iterable`. For example:

```
@runtime_checkable
class Closable(Protocol):
    def close(self): ...

assert isinstance(open('/some/file'), Closable)

@runtime_checkable
class Named(Protocol):
    name: str

import threading
assert isinstance(threading.Thread(name='Bob'), Named)
```

This decorator raises `TypeError` when applied to a non-protocol class.

Note

`runtime_checkable()` will check only the presence of the required methods or attributes, not their type signatures or types. For example, `ssl.SSLObject` is a class, therefore it passes an `issubclass()` check against Callable. However, the `ssl.SSLObject.__init__` method exists only to raise a `TypeError` with a more informative message, therefore making it impossible to call (instantiate) `ssl.SSLObject`.

Note

An `isinstance()` check against a runtime-checkable protocol can be surprisingly slow compared to an `isinstance()` check against a non-protocol class. Consider using alternative idioms such as `hasattr()` calls for structural checks in performance-sensitive code.

Added in version 3.8.

Changed in version 3.12: The internal implementation of `isinstance()` checks against runtime-checkable protocols now uses `inspect.getattr_static()` to look up attributes (previously, `hasattr()` was used). As a result, some objects which used to be considered

instances of a runtime-checkable protocol may no longer be considered instances of that protocol on Python 3.12+, and vice versa. Most users are unlikely to be affected by this change.
Changed in version 3.12: The members of a runtime-checkable protocol are now considered “frozen” at runtime as soon as the class has been created. Monkey-patching attributes onto a runtime-checkable protocol will still work, but will have no impact on **`isinstance()`** checks comparing objects to the protocol. See What’s new in Python 3.12 for more details.

class typing.TypedDict(*dict*)

Special construct to add type hints to a dictionary. At runtime “**`TypedDict`** instances” are simply **`dicts`**.

`TypedDict` declares a dictionary type that expects all of its instances to have a certain set of keys, where each key is associated with a value of a consistent type. This expectation is not checked at runtime but is only enforced by type checkers. Usage:

```
class Point2D(TypedDict):
    x: int
    y: int
    label: str

a: Point2D = {'x': 1, 'y': 2, 'label': 'good'}  # OK
b: Point2D = {'z': 3, 'label': 'bad'}           # Fails type check

assert Point2D(x=1, y=2, label='first') == dict(x=1, y=2,
label='first')
```

An alternative way to create a `TypedDict` is by using function-call syntax. The second argument must be a literal **`dict`**:

```
Point2D = TypedDict('Point2D', {'x': int, 'y': int, 'label': str})
```

This functional syntax allows defining keys which are not valid identifiers, for example because they are keywords or contain hyphens, or when key names must not be mangled like regular private names:

```
# raises SyntaxError
class Point2D(TypedDict):
    in: int  # 'in' is a keyword
    x-y: int  # name with hyphens
```

```
class Definition(TypedDict):
    __schema: str  # mangled to `_Definition__schema`

# OK, functional syntax
Point2D = TypedDict('Point2D', {'in': int, 'x-y': int})
Definition = TypedDict('Definition', {'__schema': str})  # not mangled
```

By default, all keys must be present in a `TypedDict`. It is possible to mark individual keys as non-required using **`NotRequired`**:

```
class Point2D(TypedDict):
    x: int
    y: int
    label: NotRequired[str]

# Alternative syntax
Point2D = TypedDict('Point2D', {'x': int, 'y': int, 'label':
NotRequired[str]})
```

This means that a `Point2D` `TypedDict` can have the `label` key omitted.

It is also possible to mark all keys as non-required by default by specifying a totality of `False`:

```
class Point2D(TypedDict, total=False):
    x: int
    y: int

# Alternative syntax
Point2D = TypedDict('Point2D', {'x': int, 'y': int}, total=False)
```

This means that a `Point2D` `TypedDict` can have any of the keys omitted. A type checker is only expected to support a literal `False` or `True` as the value of the `total` argument. `True` is the default, and makes all items defined in the class body required.

Individual keys of a `total=False` `TypedDict` can be marked as required using **`Required`**:

```
class Point2D(TypedDict, total=False):
    x: Required[int]
    y: Required[int]
    label: str

# Alternative syntax
Point2D = TypedDict('Point2D', {
    'x': Required[int],
    'y': Required[int],
    'label': str
}, total=False)
```

It is possible for a `TypedDict` type to inherit from one or more other `TypedDict` types using the class-based syntax. Usage:

```
class Point3D(Point2D):
    z: int
```

`Point3D` has three items: `x`, `y` and `z`. It is equivalent to this definition:

```
class Point3D(TypedDict):
    x: int
    y: int
    z: int
```

A `TypedDict` cannot inherit from a non-`TypedDict` class, except for **`Generic`**. For example:

```
class X(TypedDict):
    x: int

class Y(TypedDict):
    y: int

class Z(object): pass  # A non-TypedDict class

class XY(X, Y): pass  # OK

class XZ(X, Z): pass  # raises TypeError
```

A `TypedDict` can be generic:

```
class Group[T](TypedDict):
    key: T
    group: list[T]
```

To create a generic `TypedDict` that is compatible with Python 3.11 or lower, inherit from **`Generic`** explicitly:

```
T = TypeVar("T")

class Group(TypedDict, Generic[T]):
    key: T
    group: list[T]
```

A `TypedDict` can be introspected via annotations dicts (see Annotations Best Practices for more information on annotations best practices), **`__total__`**, **`__required_keys__`**, and **`__optional_keys__`**.

__total__

`Point2D.__total__` gives the value of the `total` argument. Example:

```
>>> from typing import TypedDict
>>> class Point2D(TypedDict): pass
>>> Point2D.__total__
True
>>> class Point2D(TypedDict, total=False): pass
>>> Point2D.__total__
False
>>> class Point3D(Point2D): pass
>>> Point3D.__total__
True
```

This attribute reflects *only* the value of the `total` argument to the current `TypedDict` class, not whether the class is semantically total. For example, a `TypedDict` with `__total__` set to `True` may have keys marked with **`NotRequired`**, or it may inherit from another `TypedDict` with `total=False`. Therefore, it is generally better to use **`__required_keys__`** and **`__optional_keys__`** for introspection.

__required_keys__

Added in version 3.9.

__optional_keys__

`Point2D.__required_keys__` and `Point2D.__optional_keys__` return **`frozenset`** objects containing required and non-required keys, respectively.

Keys marked with **`Required`** will always appear in `__required_keys__` and keys marked with **`NotRequired`** will always appear in `__optional_keys__`.

For backwards compatibility with Python 3.10 and below, it is also possible to use inheritance to declare both required and non-required keys in the same `TypedDict` . This is done by declaring a `TypedDict` with one value for the `total` argument and then inheriting from it in another `TypedDict` with a different value for `total`:

```
>>> class Point2D(TypedDict, total=False):
...     x: int
...     y: int
...
>>> class Point3D(Point2D):
...     z: int
...
>>> Point3D.__required_keys__ == frozenset({'z'})
True
>>> Point3D.__optional_keys__ == frozenset({'x', 'y'})
True
```

Added in version 3.9.

> **Note**
>
> If `from __future__ import annotations` is used or if annotations are given as strings, annotations are not evaluated when the `TypedDict` is defined. Therefore, the runtime introspection that `__required_keys__` and `__optional_keys__` rely on may not work properly, and the values of the attributes may be incorrect.

Support for **`ReadOnly`** is reflected in the following attributes:

__readonly_keys__

A **`frozenset`** containing the names of all read-only keys. Keys are read-only if they carry the **`ReadOnly`** qualifier.

Added in version 3.13.

__mutable_keys__

A **`frozenset`** containing the names of all mutable keys. Keys are mutable if they do not carry the **`ReadOnly`** qualifier.

Added in version 3.13.

See the TypedDict [https://typing.python.org/en/latest/spec/typeddict.html#typeddict] section in the typing documentation for more examples and detailed rules.

Added in version 3.8.

Changed in version 3.9: `TypedDict` is now a function rather than a class. It can still be used as a class base, as described above.

Changed in version 3.11: Added support for marking individual keys as **`Required`** or **`NotRequired`**. See **PEP 655** [https://peps.python.org/pep-

0655/].

Changed in version 3.11: Added support for generic `TypedDict`s.

Changed in version 3.13: Removed support for the keyword-argument method of creating `TypedDict`s.

Changed in version 3.13: Support for the **`ReadOnly`** qualifier was added.

Deprecated since version 3.13, will be removed in version 3.15: When using the functional syntax to create a TypedDict class, failing to pass a value to the 'fields' parameter (`TD = TypedDict("TD")`) is deprecated. Passing `None` to the 'fields' parameter (`TD = TypedDict("TD", None)`) is also deprecated. Both will be disallowed in Python 3.15. To create a TypedDict class with 0 fields, use `class TD(TypedDict): pass` or `TD = TypedDict("TD", {})`.

Protocols

The following protocols are provided by the **`typing`** module. All are decorated with **`@runtime_checkable`**.

class typing.SupportsAbs

An ABC with one abstract method `__abs__` that is covariant in its return type.

class typing.SupportsBytes

An ABC with one abstract method `__bytes__`.

class typing.SupportsComplex

An ABC with one abstract method `__complex__`.

class typing.SupportsFloat

An ABC with one abstract method `__float__`.

class typing.SupportsIndex

An ABC with one abstract method `__index__`.

Added in version 3.8.

class typing.SupportsInt

An ABC with one abstract method `__int__`.

class typing.SupportsRound

An ABC with one abstract method `__round__` that is covariant in its return type.

ABCs and Protocols for working with I/O

class typing.IO[*AnyStr*]
class typing.TextIO
class typing.BinaryIO

Generic class `IO[AnyStr]` and its subclasses `TextIO(IO[str])` and `BinaryIO(IO[bytes])` represent the types of I/O streams such as returned by `open()`. Please note that these classes are not protocols, and their interface is fairly broad.

The protocols `io.Reader` and `io.Writer` offer a simpler alternative for argument types, when only the `read()` or `write()` methods are accessed, respectively:

```
def read_and_write(reader: Reader[str], writer: Writer[bytes]):
    data = reader.read()
    writer.write(data.encode())
```

Also consider using `collections.abc.Iterable` for iterating over the lines of an input stream:

```
def read_config(stream: Iterable[str]):
    for line in stream:
        ...
```

Functions and decorators

typing.cast(*typ*, *val*)

Cast a value to a type.

This returns the value unchanged. To the type checker this signals that the return value has the designated type, but at runtime we intentionally don’t check anything (we want this to be as fast as possible).

typing.assert_type(*val*, *typ*, */*)

Ask a static type checker to confirm that *val* has an inferred type of *typ*.

At runtime this does nothing: it returns the first argument unchanged with no checks or side effects, no matter the actual type of the argument.

When a static type checker encounters a call to `assert_type()`, it emits an error if the value is not of the specified type:

```
def greet(name: str) -> None:
    assert_type(name, str)  # OK, inferred type of `name` is `str`
    assert_type(name, int)  # type checker error
```

This function is useful for ensuring the type checker’s understanding of a script is in line with the developer’s intentions:

```
def complex_function(arg: object):
    # Do some complex type-narrowing logic,
    # after which we hope the inferred type will be `int`
    ...
    # Test whether the type checker correctly understands our function
    assert_type(arg, int)
```

Added in version 3.11.

typing.assert_never(*arg*, */*)

Ask a static type checker to confirm that a line of code is unreachable.

Example:

```
def int_or_str(arg: int | str) -> None:
    match arg:
        case int():
            print("It's an int")
        case str():
            print("It's a str")
        case _ as unreachable:
            assert_never(unreachable)
```

Here, the annotations allow the type checker to infer that the last case can never execute, because `arg` is either an **`int`** or a **`str`**, and both options are covered by earlier cases.

If a type checker finds that a call to `assert_never()` is reachable, it will emit an error. For example, if the type annotation for `arg` was instead `int | str | float`, the type checker would emit an error pointing out that `unreachable` is of type **`float`**. For a call to `assert_never` to pass type checking, the inferred type of the argument passed in must be the bottom type, **`Never`**, and nothing else.

At runtime, this throws an exception when called.

See also

Unreachable Code and Exhaustiveness Checking [https://typing.python.org/en/latest/guides/unreachable.html] has more information about exhaustiveness checking with static typing.

Added in version 3.11.

typing.reveal_type(*obj*, */*)

Ask a static type checker to reveal the inferred type of an expression.

When a static type checker encounters a call to this function, it emits a diagnostic with the inferred type of the argument. For example:

```
x: int = 1
reveal_type(x)  # Revealed type is "builtins.int"
```

This can be useful when you want to debug how your type checker handles a particular piece of code.

At runtime, this function prints the runtime type of its argument to **`sys.stderr`** and returns the argument unchanged (allowing the call to be used within an expression):

```
x = reveal_type(1)  # prints "Runtime type is int"
print(x)  # prints "1"
```

Note that the runtime type may be different from (more or less specific than) the type statically inferred by a type checker.

Most type checkers support `reveal_type()` anywhere, even if the name is not imported from `typing`. Importing the name from `typing`, however, allows your code to run without runtime errors and communicates intent more clearly.

Added in version 3.11.

@typing.dataclass_transform(*, *eq_default=True*, *order_default=False*, *kw_only_default=False*, *frozen_default=False*, *field_specifiers=()*, ***kwargs*)

Decorator to mark an object as providing `dataclass`-like behavior.

`dataclass_transform` may be used to decorate a class, metaclass, or a function that is itself a decorator. The presence of `@dataclass_transform()` tells a static type checker that the decorated object performs runtime "magic" that transforms a class in a similar way to `@dataclasses.dataclass`.

Example usage with a decorator function:

```
@dataclass_transform()
def create_model[T](cls: type[T]) -> type[T]:
    ...
    return cls

@create_model
class CustomerModel:
    id: int
    name: str
```

On a base class:

```
@dataclass_transform()
class ModelBase: ...

class CustomerModel(ModelBase):
    id: int
    name: str
```

On a metaclass:

```
@dataclass_transform()
class ModelMeta(type): ...

class ModelBase(metaclass=ModelMeta): ...

class CustomerModel(ModelBase):
    id: int
    name: str
```

The `CustomerModel` classes defined above will be treated by type checkers similarly to classes created with `@dataclasses.dataclass`.

For example, type checkers will assume these classes have `__init__` methods that accept `id` and `name`.

The decorated class, metaclass, or function may accept the following bool arguments which type checkers will assume have the same effect as they would have on the **`@dataclasses.dataclass`** decorator: `init`, `eq`, `order`, `unsafe_hash`, `frozen`, `match_args`, `kw_only`, and `slots`. It must be possible for the value of these arguments (`True` or `False`) to be statically evaluated.

The arguments to the `dataclass_transform` decorator can be used to customize the default behaviors of the decorated class, metaclass, or function:

Parameters::

- **eq_default** (*bool*) – Indicates whether the `eq` parameter is assumed to be `True` or `False` if it is omitted by the caller. Defaults to `True`.
- **order_default** (*bool*) – Indicates whether the `order` parameter is assumed to be `True` or `False` if it is omitted by the caller. Defaults to `False`.
- **kw_only_default** (*bool*) – Indicates whether the `kw_only` parameter is assumed to be `True` or `False` if it is omitted by the caller. Defaults to `False`.
- **frozen_default** (*bool*) – Indicates whether the `frozen` parameter is assumed to be `True` or `False` if it is omitted by the caller. Defaults to `False`.
  Added in version 3.12.
- **field_specifiers** (*tuple[Callable[..., Any], ...]*) – Specifies a static list of supported classes or functions that describe fields, similar to **`dataclasses.field()`**. Defaults to `()`.
- ****kwargs** (*Any*) – Arbitrary other keyword arguments are accepted in order to allow for possible future extensions.

Type checkers recognize the following optional parameters on field specifiers:

Recognised parameters for field specifiers

Parameter name	**Description**
`init`	Indicates whether the field should be included in the synthesized `__init__` method. If unspecified, `init` defaults to `True`.
`default`	Provides the default value for the field.
`default_factory`	Provides a runtime callback that returns the default value for the field. If neither `default` nor `default_factory` are specified, the field is assumed to have no default value and must be provided a value when the class is instantiated.
`factory`	An alias for the `default_factory` parameter on field specifiers.
`kw_only`	Indicates whether the field should be marked as keyword-only. If `True`, the field will be keyword-only. If `False`, it will not be keyword-only. If unspecified, the value of the `kw_only` parameter on the object decorated with `dataclass_transform` will be used, or if that is unspecified, the value of `kw_only_default` on `dataclass_transform` will be used.
`alias`	Provides an alternative name for the field. This alternative name is used in the synthesized `__init__` method.

At runtime, this decorator records its arguments in the `__dataclass_transform__` attribute on the decorated object. It has no other runtime effect.

See **PEP 681** [https://peps.python.org/pep-0681/] for more details.

Added in version 3.11.

@typing.overload

Decorator for creating overloaded functions and methods.

The `@overload` decorator allows describing functions and methods that support multiple different combinations of argument types. A series of `@overload`-decorated definitions must be followed by exactly one non-`@overload`-decorated definition (for the same function/method).

`@overload`-decorated definitions are for the benefit of the type checker only, since they will be overwritten by the non-`@overload`-decorated definition. The non-`@overload`-decorated definition, meanwhile, will be used at runtime but should be ignored by a type checker. At runtime, calling an `@overload`-decorated function directly will raise **`NotImplementedError`**.

An example of overload that gives a more precise type than can be expressed using a union or a type variable:

```
@overload
def process(response: None) -> None:
    ...
@overload
def process(response: int) -> tuple[int, str]:
    ...
@overload
def process(response: bytes) -> str:
    ...
def process(response):
    ...  # actual implementation goes here
```

See **PEP 484** [https://peps.python.org/pep-0484/] for more details and comparison with other typing semantics.

Changed in version 3.11: Overloaded functions can now be introspected at runtime using **`get_overloads()`**.

typing.get_overloads(*func*)

Return a sequence of **`@overload`**-decorated definitions for *func*.

func is the function object for the implementation of the overloaded function. For example, given the definition of `process` in the documentation for **`@overload`**, `get_overloads(process)` will return a sequence of three function objects for the three defined overloads. If called on a function with no overloads, `get_overloads()` returns an empty sequence.

`get_overloads()` can be used for introspecting an overloaded function at runtime.

Added in version 3.11.

typing.clear_overloads()

Clear all registered overloads in the internal registry.

This can be used to reclaim the memory used by the registry.

Added in version 3.11.

@typing.final

Decorator to indicate final methods and final classes.

Decorating a method with `@final` indicates to a type checker that the method cannot be overridden in a subclass. Decorating a class with `@final` indicates that it cannot be subclassed.

For example:

```
class Base:
    @final
    def done(self) -> None:
        ...
class Sub(Base):
    def done(self) -> None:  # Error reported by type checker
        ...

@final
class Leaf:
    ...
class Other(Leaf):  # Error reported by type checker
    ...
```

There is no runtime checking of these properties. See **PEP 591** [https://peps.python.org/pep-0591/] for more details.

Added in version 3.8.

Changed in version 3.11: The decorator will now attempt to set a `__final__` attribute to `True` on the decorated object. Thus, a check like `if getattr(obj, "__final__", False)` can be used at runtime to determine whether an object `obj` has been marked as final. If the decorated object does not support setting attributes, the decorator returns the object unchanged without raising an exception.

@typing.no_type_check

Decorator to indicate that annotations are not type hints.

This works as a class or function decorator. With a class, it applies recursively to all methods and classes defined in that class (but not to methods defined in its superclasses or subclasses). Type checkers will ignore all annotations in a function or class with this decorator.

`@no_type_check` mutates the decorated object in place.

@typing.no_type_check_decorator

Decorator to give another decorator the **`no_type_check()`** effect.

This wraps the decorator with something that wraps the decorated function in **`no_type_check()`**.

Deprecated since version 3.13, will be removed in version 3.15: No type checker ever added support for `@no_type_check_decorator`. It is therefore deprecated, and will be removed in Python 3.15.

@typing.override

Decorator to indicate that a method in a subclass is intended to override a method or attribute in a superclass.

Type checkers should emit an error if a method decorated with `@override` does not, in fact, override anything. This helps prevent bugs that may occur when a base class is changed without an equivalent change to a child class.

For example:

```
class Base:
    def log_status(self) -> None:
        ...

class Sub(Base):
    @override
    def log_status(self) -> None:  # Okay: overrides Base.log_status
        ...
    @override
    def done(self) -> None:  # Error reported by type checker
        ...
```

There is no runtime checking of this property.

The decorator will attempt to set an `__override__` attribute to `True` on the decorated object. Thus, a check like `if getattr(obj, "__override__", False)` can be used at runtime to determine whether an object `obj` has been marked as an override. If the decorated object does not support setting attributes, the decorator returns the object unchanged without raising an exception.

See **PEP 698** [https://peps.python.org/pep-0698/] for more details.

Added in version 3.12.

@typing.type_check_only

Decorator to mark a class or function as unavailable at runtime.

This decorator is itself not available at runtime. It is mainly intended to mark classes that are defined in type stub files if an implementation returns an instance of a private class:

```
@type_check_only
class Response:  # private or not available at runtime
    code: int
    def get_header(self, name: str) -> str: ...

def fetch_response() -> Response: ...
```

Note that returning instances of private classes is not recommended. It is usually preferable to make such classes public.

Introspection helpers

typing.get_type_hints(*obj*, *globalns=None*, *localns=None*, *include_extras=False*)

Return a dictionary containing type hints for a function, method, module or class object.

This is often the same as `obj.__annotations__`, but this function makes the following changes to the annotations dictionary:

- Forward references encoded as string literals or **`ForwardRef`** objects are handled by evaluating them in *globalns*, *localns*, and (where applicable) *obj*'s type parameter namespace. If *globalns* or *localns* is not given, appropriate namespace dictionaries are inferred from *obj*.
- `None` is replaced with **`types.NoneType`**.
- If **`@no_type_check`** has been applied to *obj*, an empty dictionary is returned.
- If *obj* is a class `C`, the function returns a dictionary that merges annotations from `C`'s base classes with those on `C` directly. This is done by traversing **`C.__mro__`** and iteratively combining `__annotations__` dictionaries. Annotations on classes appearing earlier in the method resolution order always take precedence over annotations on classes appearing later in the method resolution order.
- The function recursively replaces all occurrences of `Annotated[T, ...]`, `Required[T]`, `NotRequired[T]`, and `ReadOnly[T]` with `T`, unless *include_extras* is set to `True` (see **`Annotated`** for more information).

See also **`annotationlib.get_annotations()`**, a lower-level function that returns annotations more directly.

Caution

This function may execute arbitrary code contained in annotations. See Security implications of introspecting annotations for more information.

Note

If any forward references in the annotations of *obj* are not resolvable or are not valid Python code, this function will raise an exception such as **`NameError`**. For example, this can happen with imported type aliases that include forward references, or with names imported under **`if TYPE_CHECKING`**.

Changed in version 3.9: Added `include_extras` parameter as part of **PEP 593** [https://peps.python.org/pep-0593/]. See the documentation on **`Annotated`** for more information.

Changed in version 3.11: Previously, `Optional[t]` was added for function and method annotations if a default value equal to `None` was set. Now the annotation is returned unchanged.

typing.get_origin(*tp*)

Get the unsubscripted version of a type: for a typing object of the form `X[Y, Z, ...]` return `X`.

If `X` is a typing-module alias for a builtin or **`collections`** class, it will be normalized to the original class. If `X` is an instance of **`ParamSpecArgs`** or **`ParamSpecKwargs`**, return the underlying **`ParamSpec`**. Return `None` for unsupported objects.

Examples:

```
assert get_origin(str) is None
assert get_origin(Dict[str, int]) is dict
assert get_origin(Union[int, str]) is Union
assert get_origin(Annotated[str, "metadata"]) is Annotated
P = ParamSpec('P')
assert get_origin(P.args) is P
assert get_origin(P.kwargs) is P
```

Added in version 3.8.

typing.get_args(*tp*)

Get type arguments with all substitutions performed: for a typing object of the form `X[Y, Z, ...]` return `(Y, Z, ...)`.

If `X` is a union or **`Literal`** contained in another generic type, the order of `(Y, Z, ...)` may be different from the order of the original arguments `[Y, Z, ...]` due to type caching. Return `()` for unsupported objects.

Examples:

```
assert get_args(int) == ()
assert get_args(Dict[int, str]) == (int, str)
assert get_args(Union[int, str]) == (int, str)
```

Added in version 3.8.

typing.get_protocol_members(*tp*)

Return the set of members defined in a **`Protocol`**.

```
>>> from typing import Protocol, get_protocol_members
>>> class P(Protocol):
...     def a(self) -> str: ...
...     b: int
>>> get_protocol_members(P) == frozenset({'a', 'b'})
True
```

Raise `TypeError` for arguments that are not Protocols.

Added in version 3.13.

typing.is_protocol(*tp*)

Determine if a type is a `Protocol`.

For example:

```
class P(Protocol):
    def a(self) -> str: ...
    b: int

is_protocol(P)    # => True
is_protocol(int)  # => False
```

Added in version 3.13.

typing.is_typeddict(*tp*)

Check if a type is a `TypedDict`.

For example:

```
class Film(TypedDict):
    title: str
    year: int

assert is_typeddict(Film)
assert not is_typeddict(list | str)

# TypedDict is a factory for creating typed dicts,
# not a typed dict itself
assert not is_typeddict(TypedDict)
```

Added in version 3.10.

class typing.ForwardRef

Class used for internal typing representation of string forward references.

For example, `List["SomeClass"]` is implicitly transformed into `List[ForwardRef("SomeClass")]`. **`ForwardRef`** should not be instantiated by a user, but may be used by introspection tools.

Note

PEP 585 [https://peps.python.org/pep-0585/] generic types such as `list["SomeClass"]` will not be implicitly transformed into `list[ForwardRef("SomeClass")]` and thus will not automatically resolve to `list[SomeClass]`.

Added in version 3.7.4.

Changed in version 3.14: This is now an alias for **`annotationlib.ForwardRef`**. Several undocumented behaviors of this class have been changed; for example, after a `ForwardRef` has been evaluated, the evaluated value is no longer cached.

typing.evaluate_forward_ref(*forward_ref*, *, *owner=None*, *globals=None*, *locals=None*, *type_params=None*, *format=annotationlib.Format.VALUE*)

Evaluate an **`annotationlib.ForwardRef`** as a type hint.

This is similar to calling **`annotationlib.ForwardRef.evaluate()`**, but unlike that method, **`evaluate_forward_ref()`** also recursively evaluates forward references nested within the type hint.

See the documentation for **`annotationlib.ForwardRef.evaluate()`** for the meaning of the *owner*, *globals*, *locals*, *type_params*, and *format* parameters.

Caution

This function may execute arbitrary code contained in annotations. See Security implications of introspecting annotations for more information.

Added in version 3.14.

typing.NoDefault

A sentinel object used to indicate that a type parameter has no default value. For example:

```
>>> T = TypeVar("T")
>>> T.__default__ is typing.NoDefault
True
>>> S = TypeVar("S", default=None)
>>> S.__default__ is None
True
```

Added in version 3.13.

Constant

typing.TYPE_CHECKING

A special constant that is assumed to be `True` by 3rd party static type checkers. It’s `False` at runtime.

A module which is expensive to import, and which only contain types used for typing annotations, can be safely imported inside an `if TYPE_CHECKING:` block. This prevents the module from actually being imported at runtime; annotations aren’t eagerly evaluated (see **PEP 649** [https://peps.python.org/pep-0649/]) so using undefined symbols in annotations is harmless–as long as you don’t later examine them. Your static type analysis tool will set `TYPE_CHECKING` to `True` during static type analysis, which means the module will be imported and the types will be checked properly during such analysis.

Usage:

```
if TYPE_CHECKING:
    import expensive_mod

def fun(arg: expensive_mod.SomeType) -> None:
    local_var: expensive_mod.AnotherType = other_fun()
```

If you occasionally need to examine type annotations at runtime which may contain undefined symbols, use `annotationlib.get_annotations()` with a `format` parameter of `annotationlib.Format.STRING` or `annotationlib.Format.FORWARDREF` to safely retrieve the annotations without raising `NameError`.

Added in version 3.5.2.

Deprecated aliases

This module defines several deprecated aliases to pre-existing standard library classes. These were originally included in the `typing` module in order to support parameterizing these generic classes using `[]`. However, the aliases became redundant in Python 3.9 when the corresponding pre-existing classes were enhanced to support `[]` (see **PEP 585** [https://peps.python.org/pep-0585/]).

The redundant types are deprecated as of Python 3.9. However, while the aliases may be removed at some point, removal of these aliases is not currently planned. As such, no deprecation warnings are currently issued by the interpreter for these aliases.

If at some point it is decided to remove these deprecated aliases, a deprecation warning will be issued by the interpreter for at least two releases prior to removal. The aliases are guaranteed to remain in the `typing` module without deprecation warnings until at least Python 3.14.

Type checkers are encouraged to flag uses of the deprecated types if the program they are checking targets a minimum Python version of 3.9 or newer.

Aliases to built-in types

class typing.Dict(*dict, MutableMapping[KT, VT]*)

Deprecated alias to `dict`.

Note that to annotate arguments, it is preferred to use an abstract collection type such as `Mapping` rather than to use `dict` or `typing.Dict`.

Deprecated since version 3.9: `builtins.dict` now supports subscripting (`[]`). See **PEP 585** [https://peps.python.org/pep-0585/] and Generic Alias Type.

class typing.List(*list, MutableSequence[T]*)

Deprecated alias to `list`.

Note that to annotate arguments, it is preferred to use an abstract collection type such as `Sequence` or `Iterable` rather than to use `list` or `typing.List`.

Deprecated since version 3.9: **`builtins.list`** now supports subscripting (`[]`). See **PEP 585** [https://peps.python.org/pep-0585/] and Generic Alias Type.

class typing.Set(*set, MutableSet[T]*)

Deprecated alias to **`builtins.set`**.

Note that to annotate arguments, it is preferred to use an abstract collection type such as **`collections.abc.Set`** rather than to use **`set`** or **`typing.Set`**.

Deprecated since version 3.9: **`builtins.set`** now supports subscripting (`[]`). See **PEP 585** [https://peps.python.org/pep-0585/] and Generic Alias Type.

class typing.FrozenSet(*frozenset, AbstractSet[T_co]*)

Deprecated alias to **`builtins.frozenset`**.

Deprecated since version 3.9: **`builtins.frozenset`** now supports subscripting (`[]`). See **PEP 585** [https://peps.python.org/pep-0585/] and Generic Alias Type.

typing.Tuple

Deprecated alias for **`tuple`**.

`tuple` and `Tuple` are special-cased in the type system; see Annotating tuples for more details.

Deprecated since version 3.9: **`builtins.tuple`** now supports subscripting (`[]`). See **PEP 585** [https://peps.python.org/pep-0585/] and Generic Alias Type.

class typing.Type(*Generic[CT_co]*)

Deprecated alias to **`type`**.

See The type of class objects for details on using **`type`** or `typing.Type` in type annotations.

Added in version 3.5.2.

Deprecated since version 3.9: **`builtins.type`** now supports subscripting (`[]`). See **PEP 585** [https://peps.python.org/pep-0585/] and

Generic Alias Type.

Aliases to types in `collections`

class typing.DefaultDict(*collections.defaultdict, MutableMapping[KT, VT]*)

Deprecated alias to `collections.defaultdict`.

Added in version 3.5.2.

Deprecated since version 3.9: `collections.defaultdict` now supports subscripting (`[]`). See **PEP 585** [https://peps.python.org/pep-0585/] and Generic Alias Type.

class typing.OrderedDict(*collections.OrderedDict, MutableMapping[KT, VT]*)

Deprecated alias to `collections.OrderedDict`.

Added in version 3.7.2.

Deprecated since version 3.9: `collections.OrderedDict` now supports subscripting (`[]`). See **PEP 585** [https://peps.python.org/pep-0585/] and Generic Alias Type.

class typing.ChainMap(*collections.ChainMap, MutableMapping[KT, VT]*)

Deprecated alias to `collections.ChainMap`.

Added in version 3.6.1.

Deprecated since version 3.9: `collections.ChainMap` now supports subscripting (`[]`). See **PEP 585** [https://peps.python.org/pep-0585/] and Generic Alias Type.

class typing.Counter(*collections.Counter, Dict[T, int]*)

Deprecated alias to `collections.Counter`.

Added in version 3.6.1.

Deprecated since version 3.9: `collections.Counter` now supports subscripting (`[]`). See **PEP 585** [https://peps.python.org/pep-0585/] and

Generic Alias Type.

class typing.Deque(*deque, MutableSequence[T]*)

Deprecated alias to `collections.deque`.

Added in version 3.6.1.

Deprecated since version 3.9: `collections.deque` now supports subscripting (`[]`). See **PEP 585** [https://peps.python.org/pep-0585/] and Generic Alias Type.

Aliases to other concrete types

class typing.Pattern
class typing.Match

Deprecated aliases corresponding to the return types from `re.compile()` and `re.match()`.

These types (and the corresponding functions) are generic over `AnyStr`. `Pattern` can be specialised as `Pattern[str]` or `Pattern[bytes]`; `Match` can be specialised as `Match[str]` or `Match[bytes]`.

Deprecated since version 3.9: Classes `Pattern` and `Match` from `re` now support `[]`. See **PEP 585** [https://peps.python.org/pep-0585/] and Generic Alias Type.

class typing.Text

Deprecated alias for `str`.

`Text` is provided to supply a forward compatible path for Python 2 code: in Python 2, `Text` is an alias for `unicode`.

Use `Text` to indicate that a value must contain a unicode string in a manner that is compatible with both Python 2 and Python 3:

```
def add_unicode_checkmark(text: Text) -> Text:
    return text + u' \u2713'
```

Added in version 3.5.2.

Deprecated since version 3.11: Python 2 is no longer supported, and most type checkers also no longer support type checking

Python 2 code. Removal of the alias is not currently planned, but users are encouraged to use `str` instead of `Text`.

Aliases to container ABCs in `collections.abc`

class typing.AbstractSet(*Collection[T_co]*)

Deprecated alias to `collections.abc.Set`.

Deprecated since version 3.9: `collections.abc.Set` now supports subscripting (`[]`). See **PEP 585** [https://peps.python.org/pep-0585/] and Generic Alias Type.

class typing.ByteString(*Sequence[int]*)

Deprecated alias to `collections.abc.ByteString`.

Use `isinstance(obj, collections.abc.Buffer)` to test if `obj` implements the buffer protocol at runtime. For use in type annotations, either use `Buffer` or a union that explicitly specifies the types your code supports (e.g., `bytes | bytearray | memoryview`).

`ByteString` was originally intended to be an abstract class that would serve as a supertype of both `bytes` and `bytearray`. However, since the ABC never had any methods, knowing that an object was an instance of **`ByteString`** never actually told you anything useful about the object. Other common buffer types such as `memoryview` were also never understood as subtypes of **`ByteString`** (either at runtime or by static type checkers).

See **PEP 688** [https://peps.python.org/pep-0688/#current-options] for more details.

Deprecated since version 3.9, will be removed in version 3.17.

class typing.Collection(*Sized, Iterable[T_co], Container[T_co]*)

Deprecated alias to `collections.abc.Collection`.

Added in version 3.6.

Deprecated since version 3.9: `collections.abc.Collection` now supports subscripting (`[]`). See **PEP 585** [https://peps.python.org/pep-0585/] and Generic Alias Type.

class typing.Container(*Generic[T_co]*)

Deprecated alias to `collections.abc.Container`.

Deprecated since version 3.9: `collections.abc.Container` now supports subscripting (`[]`). See **PEP 585** [https://peps.python.org/pep-0585/] and Generic Alias Type.

class typing.ItemsView(*MappingView, AbstractSet[tuple[KT_co, VT_co]]*)

Deprecated alias to `collections.abc.ItemsView`.

Deprecated since version 3.9: `collections.abc.ItemsView` now supports subscripting (`[]`). See **PEP 585** [https://peps.python.org/pep-0585/] and Generic Alias Type.

class typing.KeysView(*MappingView, AbstractSet[KT_co]*)

Deprecated alias to `collections.abc.KeysView`.

Deprecated since version 3.9: `collections.abc.KeysView` now supports subscripting (`[]`). See **PEP 585** [https://peps.python.org/pep-0585/] and Generic Alias Type.

class typing.Mapping(*Collection[KT], Generic[KT, VT_co]*)

Deprecated alias to `collections.abc.Mapping`.

Deprecated since version 3.9: `collections.abc.Mapping` now supports subscripting (`[]`). See **PEP 585** [https://peps.python.org/pep-0585/] and Generic Alias Type.

class typing.MappingView(*Sized*)

Deprecated alias to `collections.abc.MappingView`.

Deprecated since version 3.9: `collections.abc.MappingView` now supports subscripting (`[]`). See **PEP 585** [https://peps.python.org/pep-0585/] and Generic Alias Type.

class typing.MutableMapping(*Mapping[KT, VT]*)

Deprecated alias to `collections.abc.MutableMapping`.

Deprecated since version 3.9: `collections.abc.MutableMapping` now supports subscripting (`[]`). See **PEP 585** [https://peps.python.org/pep-

0585/] and Generic Alias Type.

class typing.MutableSequence(*Sequence[T]*)

Deprecated alias to **`collections.abc.MutableSequence`**.

Deprecated since version 3.9: **`collections.abc.MutableSequence`** now supports subscripting (`[]`). See **PEP 585** [https://peps.python.org/pep-0585/] and Generic Alias Type.

class typing.MutableSet(*AbstractSet[T]*)

Deprecated alias to **`collections.abc.MutableSet`**.

Deprecated since version 3.9: **`collections.abc.MutableSet`** now supports subscripting (`[]`). See **PEP 585** [https://peps.python.org/pep-0585/] and Generic Alias Type.

class typing.Sequence(*Reversible[T_co], Collection[T_co]*)

Deprecated alias to **`collections.abc.Sequence`**.

Deprecated since version 3.9: **`collections.abc.Sequence`** now supports subscripting (`[]`). See **PEP 585** [https://peps.python.org/pep-0585/] and Generic Alias Type.

class typing.ValuesView(*MappingView, Collection[_VT_co]*)

Deprecated alias to **`collections.abc.ValuesView`**.

Deprecated since version 3.9: **`collections.abc.ValuesView`** now supports subscripting (`[]`). See **PEP 585** [https://peps.python.org/pep-0585/] and Generic Alias Type.

Aliases to asynchronous ABCs in `collections.abc`

class typing.Coroutine(*Awaitable[ReturnType], Generic[YieldType, SendType, ReturnType]*)

Deprecated alias to **`collections.abc.Coroutine`**.

See Annotating generators and coroutines for details on using **`collections.abc.Coroutine`** and `typing.Coroutine` in type annotations.

Added in version 3.5.3.

Deprecated since version 3.9: **`collections.abc.Coroutine`** now supports subscripting (`[]`). See **PEP 585** [https://peps.python.org/pep-0585/] and Generic Alias Type.

class typing.AsyncGenerator(*AsyncIterator[YieldType], Generic[YieldType, SendType]*)

Deprecated alias to **`collections.abc.AsyncGenerator`**.

See Annotating generators and coroutines for details on using **`collections.abc.AsyncGenerator`** and `typing.AsyncGenerator` in type annotations.

Added in version 3.6.1.

Deprecated since version 3.9: **`collections.abc.AsyncGenerator`** now supports subscripting (`[]`). See **PEP 585** [https://peps.python.org/pep-0585/] and Generic Alias Type.

Changed in version 3.13: The `SendType` parameter now has a default.

class typing.AsyncIterable(*Generic[T_co]*)

Deprecated alias to **`collections.abc.AsyncIterable`**.

Added in version 3.5.2.

Deprecated since version 3.9: **`collections.abc.AsyncIterable`** now supports subscripting (`[]`). See **PEP 585** [https://peps.python.org/pep-0585/] and Generic Alias Type.

class typing.AsyncIterator(*AsyncIterable[T_co]*)

Deprecated alias to **`collections.abc.AsyncIterator`**.

Added in version 3.5.2.

Deprecated since version 3.9: **`collections.abc.AsyncIterator`** now supports subscripting (`[]`). See **PEP 585** [https://peps.python.org/pep-0585/] and Generic Alias Type.

class typing.Awaitable(*Generic[T_co]*)

Deprecated alias to **`collections.abc.Awaitable`**.

Added in version 3.5.2.

Deprecated since version 3.9: **`collections.abc.Awaitable`** now supports subscripting (`[]`). See **PEP 585** [https://peps.python.org/pep-0585/] and Generic Alias Type.

Aliases to other ABCs in `collections.abc`

class typing.Iterable(*Generic[T_co]*)

Deprecated alias to **`collections.abc.Iterable`**.

Deprecated since version 3.9: **`collections.abc.Iterable`** now supports subscripting (`[]`). See **PEP 585** [https://peps.python.org/pep-0585/] and Generic Alias Type.

class typing.Iterator(*Iterable[T_co]*)

Deprecated alias to **`collections.abc.Iterator`**.

Deprecated since version 3.9: **`collections.abc.Iterator`** now supports subscripting (`[]`). See **PEP 585** [https://peps.python.org/pep-0585/] and Generic Alias Type.

typing.Callable

Deprecated alias to **`collections.abc.Callable`**.

See Annotating callable objects for details on how to use **`collections.abc.Callable`** and `typing.Callable` in type annotations.

Deprecated since version 3.9: **`collections.abc.Callable`** now supports subscripting (`[]`). See **PEP 585** [https://peps.python.org/pep-0585/] and Generic Alias Type.

Changed in version 3.10: `Callable` now supports **`ParamSpec`** and **`Concatenate`**. See **PEP 612** [https://peps.python.org/pep-0612/] for more details.

class typing.Generator(*Iterator[YieldType], Generic[YieldType, SendType, ReturnType]*)

Deprecated alias to **`collections.abc.Generator`**.

See Annotating generators and coroutines for details on using **`collections.abc.Generator`** and `typing.Generator` in type annotations.

Deprecated since version 3.9: **`collections.abc.Generator`** now supports subscripting (`[]`). See **PEP 585** [https://peps.python.org/pep-0585/] and Generic Alias Type.

Changed in version 3.13: Default values for the send and return types were added.

class typing.Hashable

Deprecated alias to **`collections.abc.Hashable`**.

Deprecated since version 3.12: Use **`collections.abc.Hashable`** directly instead.

class typing.Reversible(*Iterable[T_co]*)

Deprecated alias to **`collections.abc.Reversible`**.

Deprecated since version 3.9: **`collections.abc.Reversible`** now supports subscripting (`[]`). See **PEP 585** [https://peps.python.org/pep-0585/] and Generic Alias Type.

class typing.Sized

Deprecated alias to **`collections.abc.Sized`**.

Deprecated since version 3.12: Use **`collections.abc.Sized`** directly instead.

Aliases to `contextlib` ABCs

class typing.ContextManager(*Generic[T_co, ExitT_co]*)

Deprecated alias to **`contextlib.AbstractContextManager`**.

The first type parameter, `T_co`, represents the type returned by the **`__enter__()`** method. The optional second type parameter, `ExitT_co`, which defaults to `bool | None`, represents the type returned by the **`__exit__()`** method.

Added in version 3.5.4.

Deprecated since version 3.9: `contextlib.AbstractContextManager` now supports subscripting (`[]`). See **PEP 585** [https://peps.python.org/pep-0585/] and Generic Alias Type.

Changed in version 3.13: Added the optional second type parameter, `ExitT_co`.

class typing.AsyncContextManager(*Generic[T_co, AExitT_co]*)

Deprecated alias to `contextlib.AbstractAsyncContextManager`.

The first type parameter, `T_co`, represents the type returned by the `__aenter__()` method. The optional second type parameter, `AExitT_co`, which defaults to `bool | None`, represents the type returned by the `__aexit__()` method.

Added in version 3.6.2.

Deprecated since version 3.9: `contextlib.AbstractAsyncContextManager` now supports subscripting (`[]`). See **PEP 585** [https://peps.python.org/pep-0585/] and Generic Alias Type.

Changed in version 3.13: Added the optional second type parameter, `AExitT_co`.

Deprecation Timeline of Major Features

Certain features in `typing` are deprecated and may be removed in a future version of Python. The following table summarizes major deprecations for your convenience. This is subject to change, and not all deprecations are listed.

Feature	Deprecated in	Projected removal	PEP/issue
`typing` versions of standard collections	3.9	Undecided (see Deprecated aliases for more information)	**PEP 585**

`pydoc` — Documentation generator and online help system

Source code: Lib/pydoc.py
[https://github.com/python/cpython/tree/3.14/Lib/pydoc.py]

The **`pydoc`** module automatically generates documentation from Python modules. The documentation can be presented as pages of text on the console, served to a web browser, or saved to HTML files.

For modules, classes, functions and methods, the displayed documentation is derived from the docstring (i.e. the **`__doc__`** attribute) of the object, and recursively of its documentable members. If there is no docstring, **`pydoc`** tries to obtain a description from the block of comment lines just above the definition of the class, function or method in the source file, or at the top of the module (see **`inspect.getcomments()`**).

The built-in function **`help()`** invokes the online help system in the interactive interpreter, which uses **`pydoc`** to generate its documentation as text on the console. The same text documentation can also be viewed from outside the Python interpreter by running **pydoc** as a script at the operating system’s command prompt. For example, running

```
python -m pydoc sys
```

at a shell prompt will display documentation on the **`sys`** module, in a style similar to the manual pages shown by the

Unix **man** command. The argument to **pydoc** can be the name of a function, module, or package, or a dotted reference to a class, method, or function within a module or module in a package. If the argument to **pydoc** looks like a path (that is, it contains the path separator for your operating system, such as a slash in Unix), and refers to an existing Python source file, then documentation is produced for that file.

Note

In order to find objects and their documentation, **`pydoc`** imports the module(s) to be documented. Therefore, any code on module level will be executed on that occasion. Use an `if __name__ == '__main__':` guard to only execute code when a file is invoked as a script and not just imported.

When printing output to the console, **pydoc** attempts to paginate the output for easier reading. If either the **`MANPAGER`** or the **`PAGER`** environment variable is set, **pydoc** will use its value as a pagination program. When both are set, **`MANPAGER`** is used.

Specifying a `-w` flag before the argument will cause HTML documentation to be written out to a file in the current directory, instead of displaying text on the console.

Specifying a `-k` flag before the argument will search the synopsis lines of all available modules for the keyword given as the argument, again in a manner similar to the Unix **man** command. The synopsis line of a module is the first line of its documentation string.

You can also use **pydoc** to start an HTTP server on the local machine that will serve documentation to visiting web browsers. **python -m pydoc -p 1234** will start a HTTP server on port 1234, allowing you to browse the documentation at `http://localhost:1234/` in your preferred web browser. Specifying `0` as the port number will select an arbitrary unused port.

python -m pydoc -n <hostname> will start the server listening at the given hostname. By default the hostname is ‘localhost’ but if you want the server to be reached from other machines, you may want to change the host name that the server responds to. During development this is especially useful if you want to run pydoc from within a container.

python -m pydoc -b will start the server and additionally open a web browser to a module index page. Each served page has a navigation bar at the top where you can *Get* help on an individual item, *Search* all modules with a keyword in their synopsis line, and go to the *Module index*, *Topics* and *Keywords* pages.

When **pydoc** generates documentation, it uses the current environment and path to locate modules. Thus, invoking **pydoc spam** documents precisely the version of the module you would get if you started the Python interpreter and typed `import spam`.

Module docs for core modules are assumed to reside in `https://docs.python.org/X.Y/library/` where `X` and `Y` are the major and minor version numbers of the Python interpreter. This can be overridden by setting the **`PYTHONDOCS`** environment variable to a different URL or to a local directory containing the Library Reference Manual pages.

Changed in version 3.2: Added the `-b` option.

Changed in version 3.3: The `-g` command line option was removed.

Changed in version 3.4: **`pydoc`** now uses `inspect.signature()` rather than `inspect.getfullargspec()` to extract signature information from callables.

Changed in version 3.7: Added the `-n` option.

Python Development Mode

Added in version 3.7.

The Python Development Mode introduces additional runtime checks that are too expensive to be enabled by default. It should not be more verbose than the default if the code is correct; new warnings are only emitted when an issue is detected.

It can be enabled using the **`-X dev`** command line option or by setting the **`PYTHONDEVMODE`** environment variable to `1`.

See also Python debug build.

Effects of the Python Development Mode

Enabling the Python Development Mode is similar to the following command, but with additional effects described below:

```
PYTHONMALLOC=debug PYTHONASYNCIODEBUG=1 python -W default -X
faulthandler
```

Effects of the Python Development Mode:

- Add `default` warning filter. The following warnings are shown:
  - **`DeprecationWarning`**
  - **`ImportWarning`**
  - **`PendingDeprecationWarning`**
  - **`ResourceWarning`**

  Normally, the above warnings are filtered by the default warning filters.

  It behaves as if the `-W default` command line option is used.

  Use the `-W error` command line option or set the `PYTHONWARNINGS` environment variable to `error` to treat warnings as errors.

- Install debug hooks on memory allocators to check for:

  - Buffer underflow
  - Buffer overflow
  - Memory allocator API violation
  - Unsafe usage of the GIL

  See the `PyMem_SetupDebugHooks()` C function.

  It behaves as if the `PYTHONMALLOC` environment variable is set to `debug`.

  To enable the Python Development Mode without installing debug hooks on memory allocators, set the `PYTHONMALLOC` environment variable to `default`.

- Call `faulthandler.enable()` at Python startup to install handlers for the `SIGSEGV`, `SIGFPE`, `SIGABRT`, `SIGBUS` and `SIGILL` signals to dump the Python traceback on a crash.

  It behaves as if the `-X faulthandler` command line option is used or if the `PYTHONFAULTHANDLER` environment variable is set to `1`.

- Enable asyncio debug mode. For example, `asyncio` checks for coroutines that were not awaited and logs them.

It behaves as if the **`PYTHONASYNCIODEBUG`** environment variable is set to `1`.

- Check the *encoding* and *errors* arguments for string encoding and decoding operations. Examples: **`open()`**, **`str.encode()`** and **`bytes.decode()`**.

  By default, for best performance, the *errors* argument is only checked at the first encoding/decoding error and the *encoding* argument is sometimes ignored for empty strings.

- The **`io.IOBase`** destructor logs `close()` exceptions.

- Set the **`dev_mode`** attribute of **`sys.flags`** to `True`.

The Python Development Mode does not enable the **`tracemalloc`** module by default, because the overhead cost (to performance and memory) would be too large. Enabling the **`tracemalloc`** module provides additional information on the origin of some errors. For example, **`ResourceWarning`** logs the traceback where the resource was allocated, and a buffer overflow error logs the traceback where the memory block was allocated.

The Python Development Mode does not prevent the **`-O`** command line option from removing **`assert`** statements nor from setting **`__debug__`** to `False`.

The Python Development Mode can only be enabled at the Python startup. Its value can be read from **`sys.flags.dev_mode`**.

Changed in version 3.8: The **`io.IOBase`** destructor now logs `close()` exceptions.

Changed in version 3.9: The *encoding* and *errors* arguments are now checked for string encoding and decoding operations.

ResourceWarning Example

Example of a script counting the number of lines of the text file specified in the command line:

```
import sys

def main():
    fp = open(sys.argv[1])
    nlines = len(fp.readlines())
    print(nlines)
    # The file is closed implicitly

if __name__ == "__main__":
    main()
```

The script does not close the file explicitly. By default, Python does not emit any warning. Example using README.txt, which has 269 lines:

```
$ python script.py README.txt
269
```

Enabling the Python Development Mode displays a `ResourceWarning` warning:

```
$ python -X dev script.py README.txt
269
script.py:10: ResourceWarning: unclosed file <_io.TextIOWrapper
name='README.rst' mode='r' encoding='UTF-8'>
  main()
ResourceWarning: Enable tracemalloc to get the object
allocation traceback
```

In addition, enabling `tracemalloc` shows the line where the file was opened:

```
$ python -X dev -X tracemalloc=5 script.py README.rst
269
script.py:10: ResourceWarning: unclosed file <_io.TextIOWrapper
name='README.rst' mode='r' encoding='UTF-8'>
  main()
Object allocated at (most recent call last):
  File "script.py", lineno 10
    main()
  File "script.py", lineno 4
    fp = open(sys.argv[1])
```

The fix is to close explicitly the file. Example using a context manager:

```
def main():
    # Close the file explicitly when exiting the with block
    with open(sys.argv[1]) as fp:
        nlines = len(fp.readlines())
    print(nlines)
```

Not closing a resource explicitly can leave a resource open for way longer than expected; it can cause severe issues upon exiting Python. It is bad in CPython, but it is even worse in PyPy. Closing resources explicitly makes an application more deterministic and more reliable.

Bad file descriptor error example

Script displaying the first line of itself:

```
import os

def main():
    fp = open(__file__)
    firstline = fp.readline()
    print(firstline.rstrip())
    os.close(fp.fileno())
    # The file is closed implicitly

main()
```

By default, Python does not emit any warning:

```
$ python script.py
import os
```

The Python Development Mode shows a **`ResourceWarning`** and logs a “Bad file descriptor” error when finalizing the file object:

```
$ python -X dev script.py
import os
script.py:10: ResourceWarning: unclosed file <_io.TextIOWrapper
name='script.py' mode='r' encoding='UTF-8'>
  main()
ResourceWarning: Enable tracemalloc to get the object
allocation traceback
Exception ignored in: <_io.TextIOWrapper name='script.py'
mode='r' encoding='UTF-8'>
Traceback (most recent call last):
  File "script.py", line 10, in <module>
    main()
OSError: [Errno 9] Bad file descriptor
```

`os.close(fp.fileno())` closes the file descriptor. When the file object finalizer tries to close the file descriptor again, it fails with the `Bad file descriptor` error. A file descriptor must be closed only once. In the worst case scenario, closing it twice can lead to a crash (see bpo-18748 [https://bugs.python.org/issue?@action=redirect&bpo=18748] for an example).

The fix is to remove the `os.close(fp.fileno())` line, or open the file with `closefd=False`.

`doctest` — Test interactive Python examples

Source code: Lib/doctest.py [https://github.com/python/cpython/tree/3.14/Lib/doctest.py]

The **`doctest`** module searches for pieces of text that look like interactive Python sessions, and then executes those sessions to verify that they work exactly as shown. There are several common ways to use doctest:

- To check that a module's docstrings are up-to-date by verifying that all interactive examples still work as documented.
- To perform regression testing by verifying that interactive examples from a test file or a test object work as expected.
- To write tutorial documentation for a package, liberally illustrated with input-output examples. Depending on whether the examples or the expository text are emphasized, this has the flavor of "literate testing" or "executable documentation".

Here's a complete but small example module:

```
"""
This is the "example" module.

The example module supplies one function, factorial().  For
example,

>>> factorial(5)
120
```

```
"""

def factorial(n):
    """Return the factorial of n, an exact integer >= 0.

    >>> [factorial(n) for n in range(6)]
    [1, 1, 2, 6, 24, 120]
    >>> factorial(30)
    265252859812191058636308480000000
    >>> factorial(-1)
    Traceback (most recent call last):
        ...
    ValueError: n must be >= 0

    Factorials of floats are OK, but the float must be an exact
integer:
    >>> factorial(30.1)
    Traceback (most recent call last):
        ...
    ValueError: n must be exact integer
    >>> factorial(30.0)
    265252859812191058636308480000000

    It must also not be ridiculously large:
    >>> factorial(1e100)
    Traceback (most recent call last):
        ...
    OverflowError: n too large
    """

    import math
    if not n >= 0:
        raise ValueError("n must be >= 0")
    if math.floor(n) != n:
        raise ValueError("n must be exact integer")
    if n+1 == n:  # catch a value like 1e300
        raise OverflowError("n too large")
    result = 1
    factor = 2
    while factor <= n:
        result *= factor
        factor += 1
    return result
```

```
if __name__ == "__main__":
    import doctest
    doctest.testmod()
```

If you run `example.py` directly from the command line, **`doctest`** works its magic:

```
$ python example.py
$
```

There's no output! That's normal, and it means all the examples worked. Pass `-v` to the script, and **`doctest`** prints a detailed log of what it's trying, and prints a summary at the end:

```
$ python example.py -v
Trying:
    factorial(5)
Expecting:
    120
ok
Trying:
    [factorial(n) for n in range(6)]
Expecting:
    [1, 1, 2, 6, 24, 120]
ok
```

And so on, eventually ending with:

```
Trying:
    factorial(1e100)
Expecting:
    Traceback (most recent call last):
        ...
    OverflowError: n too large
ok
2 items passed all tests:
   1 test in __main__
   6 tests in __main__.factorial
```

```
7 tests in 2 items.
7 passed.
Test passed.
$
```

That’s all you need to know to start making productive use of **`doctest`**! Jump in. The following sections provide full details. Note that there are many examples of doctests in the standard Python test suite and libraries. Especially useful examples can be found in the standard test file `Lib/test/test_doctest/test_doctest.py`.

Added in version 3.13: Output is colorized by default and can be controlled using environment variables.

Simple Usage: Checking Examples in Docstrings

The simplest way to start using doctest (but not necessarily the way you’ll continue to do it) is to end each module **`M`** with:

```
if __name__ == "__main__":
    import doctest
    doctest.testmod()
```

`doctest` then examines docstrings in module **`M`**.

Running the module as a script causes the examples in the docstrings to get executed and verified:

```
python M.py
```

This won’t display anything unless an example fails, in which case the failing example(s) and the cause(s) of the failure(s) are printed to stdout, and the final line of output is

`***Test Failed*** N failures.`, where *N* is the number of examples that failed.

Run it with the `-v` switch instead:

```
python M.py -v
```

and a detailed report of all examples tried is printed to standard output, along with assorted summaries at the end.

You can force verbose mode by passing `verbose=True` to **`testmod()`**, or prohibit it by passing `verbose=False`. In either of those cases, **`sys.argv`** is not examined by **`testmod()`** (so passing `-v` or not has no effect).

There is also a command line shortcut for running **`testmod()`**, see section Command-line Usage.

For more information on **`testmod()`**, see section Basic API.

Simple Usage: Checking Examples in a Text File

Another simple application of doctest is testing interactive examples in a text file. This can be done with the **`testfile()`** function:

```
import doctest
doctest.testfile("example.txt")
```

That short script executes and verifies any interactive Python examples contained in the file `example.txt`. The file content is treated as if it were a single giant docstring; the file doesn't need to contain a Python program! For example, perhaps `example.txt` contains this:

```
The ``example`` module
======================

Using ``factorial``
-------------------

This is an example text file in reStructuredText format.  First
import
``factorial`` from the ``example`` module:

    >>> from example import factorial

Now use it:

    >>> factorial(6)
    120
```

Running `doctest.testfile("example.txt")` then finds the error in this documentation:

```
File "./example.txt", line 14, in example.txt
Failed example:
    factorial(6)
Expected:
    120
Got:
    720
```

As with **`testmod()`**, **`testfile()`** won't display anything unless an example fails. If an example does fail, then the failing example(s) and the cause(s) of the failure(s) are printed to stdout, using the same format as **`testmod()`**.

By default, **`testfile()`** looks for files in the calling module's directory. See section Basic API for a description of the optional arguments that can be used to tell it to look for files in other locations.

Like **`testmod()`**, **`testfile()`**'s verbosity can be set with the `-v` command-line switch or with the optional keyword argument *verbose*.

There is also a command line shortcut for running **`testfile()`**, see section Command-line Usage.

For more information on **`testfile()`**, see section Basic API.

Command-line Usage

The **`doctest`** module can be invoked as a script from the command line:

```
python -m doctest [-v] [-o OPTION] [-f] file [file ...]
```

-v, --verbose

Detailed report of all examples tried is printed to standard output, along with assorted summaries at the end:

```
python -m doctest -v example.py
```

This will import `example.py` as a standalone module and run **`testmod()`** on it. Note that this may not work correctly if the file is part of a package and imports other submodules from that package.

If the file name does not end with `.py`, **`doctest`** infers that it must be run with **`testfile()`** instead:

```
python -m doctest -v example.txt
```

-o, --option <option>

Option flags control various aspects of doctest’s behavior, see section Option Flags.

Added in version 3.4.

-f, --fail-fast

This is shorthand for `-o FAIL_FAST`.

Added in version 3.4.

How It Works

This section examines in detail how doctest works: which docstrings it looks at, how it finds interactive examples, what execution context it uses, how it handles exceptions, and how option flags can be used to control its behavior. This is the information that you need to know to write doctest examples; for information about actually running doctest on these examples, see the following sections.

Which Docstrings Are Examined?

The module docstring, and all function, class and method docstrings are searched. Objects imported into the module are not searched.

In addition, there are cases when you want tests to be part of a module but not part of the help text, which requires that the tests not be included in the docstring. Doctest looks for a module-level variable called `__test__` and uses it to locate other tests. If `M.__test__` exists, it must be a dict, and each entry maps a (string) name to a function object, class object, or string. Function and class object docstrings found from `M.__test__` are searched, and strings are treated as if they were docstrings. In output, a key `K` in `M.__test__` appears with name `M.__test__.K`.

For example, place this block of code at the top of `example.py`:

```
__test__ = {
    'numbers': """
>>> factorial(6)
720

>>> [factorial(n) for n in range(6)]
[1, 1, 2, 6, 24, 120]
"""
}
```

The value of `example.__test__["numbers"]` will be treated as a docstring and all the tests inside it will be run. It is important to note that the value can be mapped to a function, class object, or module; if so, **`doctest`** searches them recursively for docstrings, which are then scanned for tests.

Any classes found are recursively searched similarly, to test docstrings in their contained methods and nested classes.

Note

`doctest` can only automatically discover classes and functions that are defined at the module level or inside other classes.

Since nested classes and functions only exist when an outer function is called, they cannot be discovered. Define them outside to make them visible.

How are Docstring Examples Recognized?

In most cases a copy-and-paste of an interactive console session works fine, but doctest isn’t trying to do an exact emulation of any specific Python shell.

```
>>> # comments are ignored
>>> x = 12
>>> x
12
>>> if x == 13:
...     print("yes")
... else:
...     print("no")
...     print("NO")
...     print("NO!!!")
...
no
NO
NO!!!
>>>
```

Any expected output must immediately follow the final `'>>> '` or `'... '` line containing the code, and the expected output (if any) extends to the next `'>>> '` or all-whitespace line.

The fine print:

- Expected output cannot contain an all-whitespace line, since such a line is taken to signal the end of expected output. If expected output does contain a blank line, put `<BLANKLINE>` in your doctest example each place a blank line is expected.

- All hard tab characters are expanded to spaces, using 8-column tab stops. Tabs in output generated by the tested code are not modified. Because any hard tabs in the sample output *are* expanded, this means that if the code output includes hard tabs, the only way the doctest can pass is if the **`NORMALIZE_WHITESPACE`** option or directive is in effect. Alternatively, the test can be rewritten to capture the output and compare it to an expected value as part of the test. This handling of tabs in the source was arrived at through trial and error, and has proven to be the least error prone way of handling them. It is possible to use a different algorithm for handling tabs by writing a custom `DocTestParser` class.

- Output to stdout is captured, but not output to stderr (exception tracebacks are captured via a different means).
- If you continue a line via backslashing in an interactive session, or for any other reason use a backslash, you should use a raw docstring, which will preserve your backslashes exactly as you type them:

  ```
  >>> def f(x):
  ...     r'''Backslashes in a raw docstring: m\n'''
  ...
  >>> print(f.__doc__)
  Backslashes in a raw docstring: m\n
  ```

  Otherwise, the backslash will be interpreted as part of the string. For example, the `\n` above would be interpreted as a newline character. Alternatively, you can double each backslash in the doctest version (and not use a raw string):

  ```
  >>> def f(x):
  ...     '''Backslashes in a raw docstring: m\\n'''
  ...
  >>> print(f.__doc__)
  Backslashes in a raw docstring: m\n
  ```

- The starting column doesn't matter:

  ```
  >>> assert "Easy!"
        >>> import math
            >>> math.floor(1.9)
            1
  ```

  and as many leading whitespace characters are stripped from the expected output as appeared in the initial `'>>>

' line that started the example.

What's the Execution Context?

By default, each time **`doctest`** finds a docstring to test, it uses a *shallow copy* of **`M`**'s globals, so that running tests doesn't change the module's real globals, and so that one test in **`M`** can't leave behind crumbs that accidentally allow another test to work. This means examples can freely use any names defined at top-level in **`M`**, and names defined earlier in the docstring being run. Examples cannot see names defined in other docstrings.

You can force use of your own dict as the execution context by passing `globs=your_dict` to **`testmod()`** or **`testfile()`** instead.

What About Exceptions?

No problem, provided that the traceback is the only output produced by the example: just paste in the traceback. [[1]] Since tracebacks contain details that are likely to change rapidly (for example, exact file paths and line numbers), this is one case where doctest works hard to be flexible in what it accepts.

Simple example:

```
>>> [1, 2, 3].remove(42)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: list.remove(x): x not in list
```

That doctest succeeds if **`ValueError`** is raised, with the `list.remove(x): x not in list` detail as shown.

The expected output for an exception must start with a traceback header, which may be either of the following two lines, indented the same as the first line of the example:

```
Traceback (most recent call last):
Traceback (innermost last):
```

The traceback header is followed by an optional traceback stack, whose contents are ignored by doctest. The traceback stack is typically omitted, or copied verbatim from an interactive session.

The traceback stack is followed by the most interesting part: the line(s) containing the exception type and detail. This is usually the last line of a traceback, but can extend across multiple lines if the exception has a multi-line detail:

```
>>> raise ValueError('multi\n    line\ndetail')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: multi
    line
detail
```

The last three lines (starting with `ValueError`) are compared against the exception's type and detail, and the rest are ignored.

Best practice is to omit the traceback stack, unless it adds significant documentation value to the example. So the last example is probably better as:

```
>>> raise ValueError('multi\n    line\ndetail')
Traceback (most recent call last):
    ...
ValueError: multi
    line
detail
```

Note that tracebacks are treated very specially. In particular, in the rewritten example, the use of `...` is independent of doctest's `ELLIPSIS` option. The ellipsis in that example could be left out, or could just as well be three (or three hundred) commas or digits, or an indented transcript of a Monty Python skit.

Some details you should read once, but won't need to remember:

- Doctest can't guess whether your expected output came from an exception traceback or from ordinary printing. So, e.g., an example that expects `ValueError: 42 is prime` will pass whether `ValueError` is actually raised or if the example merely prints that traceback text. In practice, ordinary output rarely begins with a traceback header line, so this doesn't create real problems.
- Each line of the traceback stack (if present) must be indented further than the first line of the example, *or* start with a non-alphanumeric character. The first line following the traceback header indented the same and starting with an alphanumeric is taken to be the start of the exception detail. Of course this does the right thing for genuine tracebacks.
- When the `IGNORE_EXCEPTION_DETAIL` doctest option is specified, everything following the leftmost colon and any module information in the exception name is ignored.
- The interactive shell omits the traceback header line for some `SyntaxError`s. But doctest uses the traceback header line to distinguish exceptions from non-exceptions. So in the rare case where you need to test a **`SyntaxError`** that omits the traceback header, you will need to manually add the traceback header line to your test example.

- For some exceptions, Python displays the position of the error using ^ markers and tildes:

```
>>> 1 + None
  File "<stdin>", line 1
    1 + None
    ~~^~~~~~
TypeError: unsupported operand type(s) for +: 'int' and
'NoneType'
```

Since the lines showing the position of the error come before the exception type and detail, they are not checked by doctest. For example, the following test would pass, even though it puts the `^` marker in the wrong location:

```
>>> 1 + None
  File "<stdin>", line 1
    1 + None
    ^~~~~~~~
TypeError: unsupported operand type(s) for +: 'int' and
'NoneType'
```

Option Flags

A number of option flags control various aspects of doctest's behavior. Symbolic names for the flags are supplied as module constants, which can be bitwise ORed together and passed to various functions. The names can also be used in doctest directives, and may be passed to the doctest command line interface via the `-o` option.

The first group of options define test semantics, controlling aspects of how doctest decides whether actual output matches an example's expected output:

doctest.DONT_ACCEPT_TRUE_FOR_1

By default, if an expected output block contains just `1`, an actual output block containing just `1` or just `True` is considered to be a match, and similarly for `0` versus `False`. When **`DONT_ACCEPT_TRUE_FOR_1`** is specified, neither substitution is allowed. The default behavior caters to that Python changed the return type of many functions from integer to boolean; doctests expecting “little integer” output still work in these cases. This option will probably go away, but not for several years.

doctest.DONT_ACCEPT_BLANKLINE

By default, if an expected output block contains a line containing only the string `<BLANKLINE>`, then that line will match a blank line in the actual output. Because a genuinely blank line delimits the expected output, this is the only way to communicate that a blank line is expected. When **`DONT_ACCEPT_BLANKLINE`** is specified, this substitution is not allowed.

doctest.NORMALIZE_WHITESPACE

When specified, all sequences of whitespace (blanks and newlines) are treated as equal. Any sequence of whitespace within the expected output will match any sequence of whitespace within the actual output. By default, whitespace must match exactly. **`NORMALIZE_WHITESPACE`** is especially useful when a line of expected output is very long, and you want to wrap it across multiple lines in your source.

doctest.ELLIPSIS

When specified, an ellipsis marker (`...`) in the expected output can match any substring in the actual output. This includes substrings that span line boundaries, and empty substrings, so it’s best to keep usage of this simple. Complicated uses can lead to the same kinds of “oops, it matched too much!” surprises that `.*` is prone to in regular expressions.

doctest.IGNORE_EXCEPTION_DETAIL

When specified, doctests expecting exceptions pass so long as an exception of the expected type is raised, even if the details (message and fully qualified exception name) don’t match.

For example, an example expecting `ValueError: 42` will pass if the actual exception raised is `ValueError: 3*14`, but will fail if, say, a `TypeError` is raised instead. It will also ignore any fully qualified name included before the exception class, which can vary between implementations and versions of Python and the code/libraries in use. Hence, all three of these variations will work with the flag specified:

```
>>> raise Exception('message')
Traceback (most recent call last):
Exception: message

>>> raise Exception('message')
Traceback (most recent call last):
builtins.Exception: message

>>> raise Exception('message')
Traceback (most recent call last):
__main__.Exception: message
```

Note that `ELLIPSIS` can also be used to ignore the details of the exception message, but such a test may still fail based on whether the module name is present or matches exactly.

Changed in version 3.2: `IGNORE_EXCEPTION_DETAIL` now also ignores any information relating to the module containing the exception under test.

doctest.SKIP

When specified, do not run the example at all. This can be useful in contexts where doctest examples serve as both documentation and test cases, and an example should be included for documentation purposes, but should not be checked. E.g., the example’s output might be random; or the example might depend on resources which would be unavailable to the test driver.

The SKIP flag can also be used for temporarily “commenting out” examples.

doctest.COMPARISON_FLAGS

A bitmask or’ing together all the comparison flags above.

The second group of options controls how test failures are reported:

doctest.REPORT_UDIFF

When specified, failures that involve multi-line expected and actual outputs are displayed using a unified diff.

doctest.REPORT_CDIFF

When specified, failures that involve multi-line expected and actual outputs will be displayed using a context diff.

doctest.REPORT_NDIFF

When specified, differences are computed by `difflib.Differ`, using the same algorithm as the popular `ndiff.py` utility. This is the only method that marks differences within lines as well as across lines. For example, if a line of expected output contains digit `1` where actual output contains letter `l`, a line is inserted with a caret marking the mismatching column positions.

doctest.REPORT_ONLY_FIRST_FAILURE

When specified, display the first failing example in each doctest, but suppress output for all remaining examples. This will prevent doctest from reporting correct examples that break because of earlier failures; but it might also hide incorrect examples that fail independently of the first failure. When `REPORT_ONLY_FIRST_FAILURE` is specified, the remaining examples are still run, and still count towards the total number of failures reported; only the output is suppressed.

doctest.FAIL_FAST

When specified, exit after the first failing example and don’t attempt to run the remaining examples. Thus, the number of failures reported will be at most 1. This flag may be useful during debugging, since examples after the first failure won’t even produce debugging output.

doctest.REPORTING_FLAGS

A bitmask or’ing together all the reporting flags above.

There is also a way to register new option flag names, though this isn’t useful unless you intend to extend `doctest` internals via subclassing:

doctest.register_optionflag(*name*)

Create a new option flag with a given name, and return the new flag’s integer value. `register_optionflag()` can be used when subclassing `OutputChecker` or `DocTestRunner` to create new options that are supported by your subclasses. `register_optionflag()` should always be called using the following idiom:

```
MY_FLAG = register_optionflag('MY_FLAG')
```

Directives

Doctest directives may be used to modify the option flags for an individual example. Doctest directives are special Python comments following an example’s source code:

```
directive:             "#" "doctest:" directive_options
directive_options:     directive_option ("," directive_option)*
directive_option:      on_or_off directive_option_name
on_or_off:             "+" | "-"
directive_option_name: "DONT_ACCEPT_BLANKLINE" |
"NORMALIZE_WHITESPACE" | ...
```

Whitespace is not allowed between the `+` or `-` and the directive option name. The directive option name can be any of the option flag names explained above.

An example’s doctest directives modify doctest’s behavior for that single example. Use `+` to enable the named behavior, or `-` to disable it.

For example, this test passes:

```
>>> print(list(range(20)))  # doctest: +NORMALIZE_WHITESPACE
[0,   1,  2,  3,  4,  5,  6,  7,  8,  9,
10,  11, 12, 13, 14, 15, 16, 17, 18, 19]
```

Without the directive it would fail, both because the actual output doesn’t have two blanks before the single-digit list elements, and because the actual output is on a single line. This test also passes, and also requires a directive to do so:

```
>>> print(list(range(20)))  # doctest: +ELLIPSIS
[0, 1, ..., 18, 19]
```

Multiple directives can be used on a single physical line, separated by commas:

```
>>> print(list(range(20)))  # doctest: +ELLIPSIS,
+NORMALIZE_WHITESPACE
[0,    1, ...,   18,    19]
```

If multiple directive comments are used for a single example, then they are combined:

```
>>> print(list(range(20)))  # doctest: +ELLIPSIS
...                         # doctest: +NORMALIZE_WHITESPACE
[0,    1, ...,   18,    19]
```

As the previous example shows, you can add `...` lines to your example containing only directives. This can be useful when an example is too long for a directive to comfortably fit on the same line:

```
>>> print(list(range(5)) + list(range(10, 20)) + list(range(30,
40)))
... # doctest: +ELLIPSIS
[0, ..., 4, 10, ..., 19, 30, ..., 39]
```

Note that since all options are disabled by default, and directives apply only to the example they appear in, enabling options (via `+` in a directive) is usually the only meaningful choice. However, option flags can also be passed to functions that run doctests, establishing different defaults. In such cases, disabling an option via `-` in a directive can be useful.

Warnings

`doctest` is serious about requiring exact matches in expected output. If even a single character doesn’t match, the test fails. This will probably surprise you a few times, as you learn exactly what Python does and doesn’t guarantee about output. For example, when printing a set, Python doesn’t guarantee that the element is printed in any particular order, so a test like

```
>>> foo()
{"spam", "eggs"}
```

is vulnerable! One workaround is to do

```
>>> foo() == {"spam", "eggs"}
True
```

instead. Another is to do

```
>>> d = sorted(foo())
>>> d
['eggs', 'spam']
```

There are others, but you get the idea.

Another bad idea is to print things that embed an object address, like

```
>>> id(1.0)  # certain to fail some of the time
7948648
>>> class C: pass
>>> C()  # the default repr() for instances embeds an address
<C object at 0x00AC18F0>
```

The `ELLIPSIS` directive gives a nice approach for the last example:

```
>>> C()  # doctest: +ELLIPSIS
<C object at 0x...>
```

Floating-point numbers are also subject to small output variations across platforms, because Python defers to the platform C library for some floating-point calculations, and C libraries vary widely in quality here.

```
>>> 1000**0.1  # risky
1.9952623149688797
>>> round(1000**0.1, 9) # safer
1.995262315
>>> print(f'{1000**0.1:.4f}') # much safer
1.9953
```

Numbers of the form `I/2.**J` are safe across all platforms, and I often contrive doctest examples to produce numbers of that form:

```
>>> 3./4  # utterly safe
0.75
```

Simple fractions are also easier for people to understand, and that makes for better documentation.

Basic API

The functions **`testmod()`** and **`testfile()`** provide a simple interface to doctest that should be sufficient for most basic uses. For a less formal introduction to these two functions, see sections Simple Usage: Checking Examples in Docstrings and Simple Usage: Checking Examples in a Text File.

doctest.testfile(*filename*, *module_relative=True*, *name=None*, *package=None*, *globs=None*, *verbose=None*, *report=True*, *optionflags=0*, *extraglobs=None*, *raise_on_error=False*, *parser=DocTestParser()*, *encoding=None*)

> All arguments except *filename* are optional, and should be specified in keyword form.
>
> Test examples in the file named *filename*. Return `(failure_count, test_count)`.
>
> Optional argument *module_relative* specifies how the filename should be interpreted:
>
> - If *module_relative* is `True` (the default), then *filename* specifies an OS-independent module-relative path. By default, this path is relative to the calling module’s directory; but if the *package* argument is specified, then it is relative to that package. To

ensure OS-independence, *filename* should use `/` characters to separate path segments, and may not be an absolute path (i.e., it may not begin with `/`).
- If *module_relative* is `False`, then *filename* specifies an OS-specific path. The path may be absolute or relative; relative paths are resolved with respect to the current working directory.

Optional argument *name* gives the name of the test; by default, or if `None`, `os.path.basename(filename)` is used.

Optional argument *package* is a Python package or the name of a Python package whose directory should be used as the base directory for a module-relative filename. If no package is specified, then the calling module’s directory is used as the base directory for module-relative filenames. It is an error to specify *package* if *module_relative* is `False`.

Optional argument *globs* gives a dict to be used as the globals when executing examples. A new shallow copy of this dict is created for the doctest, so its examples start with a clean slate. By default, or if `None`, a new empty dict is used.

Optional argument *extraglobs* gives a dict merged into the globals used to execute examples. This works like **`dict.update()`**: if *globs* and *extraglobs* have a common key, the associated value in *extraglobs* appears in the combined dict. By default, or if `None`, no extra globals are used. This is an advanced feature that allows parameterization of doctests. For example, a doctest can be written for a base class, using a generic name for the class, then reused to test any number of subclasses by passing an *extraglobs* dict mapping the generic name to the subclass to be tested.

Optional argument *verbose* prints lots of stuff if true, and prints only failures if false; by default, or if `None`, it’s true if and only if `'-v'` is in **`sys.argv`**.

Optional argument *report* prints a summary at the end when true, else prints nothing at the end. In verbose mode, the summary is detailed, else the summary is very brief (in fact, empty if all tests passed).

Optional argument *optionflags* (default value `0`) takes the bitwise OR of option flags. See section Option Flags.

Optional argument *raise_on_error* defaults to false. If true, an exception is raised upon the first failure or unexpected exception in an example. This allows failures to be post-mortem debugged. Default behavior is to continue running examples.

Optional argument *parser* specifies a **`DocTestParser`** (or subclass) that should be used to extract tests from the files. It defaults to a normal parser (i.e., `DocTestParser()`).

Optional argument *encoding* specifies an encoding that should be used to convert the file to unicode.

doctest.testmod(*m=None*, *name=None*, *globs=None*, *verbose=None*, *report=True*, *optionflags=0*, *extraglobs=None*, *raise_on_error=False*, *exclude_empty=False*)

All arguments are optional, and all except for *m* should be specified in keyword form.

Test examples in docstrings in functions and classes reachable from module *m* (or module **`__main__`** if *m* is not supplied or is `None`), starting with `m.__doc__`.

Also test examples reachable from dict `m.__test__`, if it exists. `m.__test__` maps names (strings) to functions, classes and strings; function and class docstrings are searched for examples; strings are searched directly, as if they were docstrings.

Only docstrings attached to objects belonging to module *m* are searched.

Return `(failure_count, test_count)`.

Optional argument *name* gives the name of the module; by default, or if `None`, `m.__name__` is used.

Optional argument *exclude_empty* defaults to false. If true, objects for which no doctests are found are excluded from consideration. The default is a backward compatibility hack, so that code still using **`doctest.master.summarize`** in conjunction with **`testmod()`** continues to get output for objects with no tests. The *exclude_empty* argument to the newer **`DocTestFinder`** constructor defaults to true.

Optional arguments *extraglobs*, *verbose*, *report*, *optionflags*, *raise_on_error*, and *globs* are the same as for function **`testfile()`** above, except that *globs* defaults to `m.__dict__`.

doctest.run_docstring_examples(*f*, *globs*, *verbose=False*, *name='NoName'*, *compileflags=None*, *optionflags=0*)

Test examples associated with object *f*; for example, *f* may be a string, a module, a function, or a class object.

A shallow copy of dictionary argument *globs* is used for the execution context.

Optional argument *name* is used in failure messages, and defaults to `"NoName"`.

If optional argument *verbose* is true, output is generated even if there are no failures. By default, output is generated only in case of an example failure.

Optional argument *compileflags* gives the set of flags that should be used by the Python compiler when running the examples. By default, or if `None`, flags are deduced corresponding to the set of future features found in *globs*.

Optional argument *optionflags* works as for function **`testfile()`** above.

Unittest API

As your collection of doctest'ed modules grows, you'll want a way to run all their doctests systematically. **`doctest`** provides two functions that can be used to create **`unittest`** test suites from modules and text files containing doctests. To integrate with **`unittest`** test discovery, include a load_tests function in your test module:

```
import unittest
import doctest
import my_module_with_doctests

def load_tests(loader, tests, ignore):

tests.addTests(doctest.DocTestSuite(my_module_with_doctests))
    return tests
```

There are two main functions for creating **`unittest.TestSuite`** instances from text files and modules with doctests:

doctest.DocFileSuite(**paths*, *module_relative=True*, *package=None*, *setUp=None*, *tearDown=None*, *globs=None*, *optionflags=0*, *parser=DocTestParser()*, *encoding=None*)

Convert doctest tests from one or more text files to a `unittest.TestSuite`.

The returned `unittest.TestSuite` is to be run by the unittest framework and runs the interactive examples in each file. If an example in any file fails, then the synthesized unit test fails, and a `failureException` exception is raised showing the name of the file containing the test and a (sometimes approximate) line number. If all the examples in a file are skipped, then the synthesized unit test is also marked as skipped.

Pass one or more paths (as strings) to text files to be examined.

Options may be provided as keyword arguments:

Optional argument *module_relative* specifies how the filenames in *paths* should be interpreted:

- If *module_relative* is `True` (the default), then each filename in *paths* specifies an OS-independent module-relative path. By default, this path is relative to the calling module’s directory; but if the *package* argument is specified, then it is relative to that package. To ensure OS-independence, each filename should use `/` characters to separate path segments, and may not be an absolute path (i.e., it may not begin with `/`).
- If *module_relative* is `False`, then each filename in *paths* specifies an OS-specific path. The path may be

absolute or relative; relative paths are resolved with respect to the current working directory.

Optional argument *package* is a Python package or the name of a Python package whose directory should be used as the base directory for module-relative filenames in *paths*. If no package is specified, then the calling module's directory is used as the base directory for module-relative filenames. It is an error to specify *package* if *module_relative* is `False`.

Optional argument *setUp* specifies a set-up function for the test suite. This is called before running the tests in each file. The *setUp* function will be passed a **`DocTest`** object. The *setUp* function can access the test globals as the **`globs`** attribute of the test passed.

Optional argument *tearDown* specifies a tear-down function for the test suite. This is called after running the tests in each file. The *tearDown* function will be passed a **`DocTest`** object. The *tearDown* function can access the test globals as the **`globs`** attribute of the test passed.

Optional argument *globs* is a dictionary containing the initial global variables for the tests. A new copy of this dictionary is created for each test. By default, *globs* is a new empty dictionary.

Optional argument *optionflags* specifies the default doctest options for the tests, created by or-ing together individual option flags. See section Option Flags. See function **`set_unittest_reportflags()`** below for a better way to set reporting options.

Optional argument *parser* specifies a **`DocTestParser`** (or subclass) that should be used to extract tests from the files. It defaults to a normal parser (i.e., `DocTestParser()`).

Optional argument *encoding* specifies an encoding that should be used to convert the file to unicode.

The global `__file__` is added to the globals provided to doctests loaded from a text file using `DocFileSuite()`.

doctest.DocTestSuite(*module=None*, *globs=None*, *extraglobs=None*, *test_finder=None*, *setUp=None*, *tearDown=None*, *optionflags=0*, *checker=None*)

Convert doctest tests for a module to a `unittest.TestSuite`.

The returned `unittest.TestSuite` is to be run by the unittest framework and runs each doctest in the module. Each docstring is run as a separate unit test. If any of the doctests fail, then the synthesized unit test fails, and a `unittest.TestCase.failureException` exception is raised showing the name of the file containing the test and a (sometimes approximate) line number. If all the examples in a docstring are skipped, then the

Optional argument *module* provides the module to be tested. It can be a module object or a (possibly dotted) module name. If not specified, the module calling this function is used.

Optional argument *globs* is a dictionary containing the initial global variables for the tests. A new copy of this dictionary is created for each test. By default, *globs* is the module’s `__dict__`.

Optional argument *extraglobs* specifies an extra set of global variables, which is merged into *globs*. By default, no extra globals are used.

Optional argument *test_finder* is the `DocTestFinder` object (or a drop-in replacement) that is used to extract doctests from the module.

Optional arguments *setUp*, *tearDown*, and *optionflags* are the same as for function `DocFileSuite()` above, but they are called for each docstring.

This function uses the same search technique as `testmod()`.

Changed in version 3.5: `DocTestSuite()` returns an empty `unittest.TestSuite` if *module* contains no docstrings instead of raising `ValueError`.

Under the covers, `DocTestSuite()` creates a `unittest.TestSuite` out of `doctest.DocTestCase` instances, and `DocTestCase` is a subclass of `unittest.TestCase`. `DocTestCase` isn't documented here (it's an internal detail), but studying its code can answer questions about the exact details of `unittest` integration.

Similarly, `DocFileSuite()` creates a `unittest.TestSuite` out of `doctest.DocFileCase` instances, and `DocFileCase` is a subclass of `DocTestCase`.

So both ways of creating a `unittest.TestSuite` run instances of `DocTestCase`. This is important for a subtle reason: when you run `doctest` functions yourself, you can control the `doctest` options in use directly, by passing option flags to `doctest` functions. However, if you're writing a `unittest` framework, `unittest` ultimately controls when and how tests get run. The framework author typically wants to control `doctest` reporting options (perhaps, e.g., specified by command line options), but there's no way to pass options through `unittest` to `doctest` test runners.

For this reason, `doctest` also supports a notion of `doctest` reporting flags specific to `unittest` support, via this function:

doctest.set_unittest_reportflags(*flags*)

Set the **`doctest`** reporting flags to use.

Argument *flags* takes the bitwise OR of option flags. See section Option Flags. Only “reporting flags” can be used.

This is a module-global setting, and affects all future doctests run by module **`unittest`**: the **`runTest()`** method of **`DocTestCase`** looks at the option flags specified for the test case when the **`DocTestCase`** instance was constructed. If no reporting flags were specified (which is the typical and expected case), **`doctest`**’s **`unittest`** reporting flags are bitwise ORed into the option flags, and the option flags so augmented are passed to the **`DocTestRunner`** instance created to run the doctest. If any reporting flags were specified when the **`DocTestCase`** instance was constructed, **`doctest`**’s **`unittest`** reporting flags are ignored.

The value of the **`unittest`** reporting flags in effect before the function was called is returned by the function.

Advanced API

The basic API is a simple wrapper that’s intended to make doctest easy to use. It is fairly flexible, and should meet most users’ needs; however, if you require more fine-grained control over testing, or wish to extend doctest’s capabilities, then you should use the advanced API.

The advanced API revolves around two container classes, which are used to store the interactive examples extracted from doctest cases:

- **`Example`**: A single Python statement, paired with its expected output.

- `DocTest`: A collection of `Example`s, typically extracted from a single docstring or text file.

Additional processing classes are defined to find, parse, and run, and check doctest examples:

- `DocTestFinder`: Finds all docstrings in a given module, and uses a `DocTestParser` to create a `DocTest` from every docstring that contains interactive examples.
- `DocTestParser`: Creates a `DocTest` object from a string (such as an object's docstring).
- `DocTestRunner`: Executes the examples in a `DocTest`, and uses an `OutputChecker` to verify their output.
- `OutputChecker`: Compares the actual output from a doctest example with the expected output, and decides whether they match.

The relationships among these processing classes are summarized in the following diagram:

```
                            list of:
+------+                   +---------+
|module| --DocTestFinder-> | DocTest | --DocTestRunner->
results
+------+    |        ^     +---------+     |       ^
(printed)
            |        |     | Example |     |       |
            v        |     |   ...   |     v       |
           DocTestParser   | Example |   OutputChecker
                           +---------+
```

DocTest Objects

class doctest.DocTest(*examples*, *globs*, *name*, *filename*, *lineno*, *docstring*)

A collection of doctest examples that should be run in a single namespace. The constructor arguments are used to initialize the attributes of the same names.

`DocTest` defines the following attributes. They are initialized by the constructor, and should not be modified directly.

examples

A list of `Example` objects encoding the individual interactive Python examples that should be run by this test.

globs

The namespace (aka globals) that the examples should be run in. This is a dictionary mapping names to values. Any changes to the namespace made by the examples (such as binding new variables) will be reflected in `globs` after the test is run.

name

A string name identifying the `DocTest`. Typically, this is the name of the object or file that the test was extracted from.

filename

The name of the file that this `DocTest` was extracted from; or `None` if the filename is unknown, or if the **`DocTest`** was not extracted from a file.

lineno

The line number within `filename` where this `DocTest` begins, or `None` if the line number is unavailable. This line number is zero-based with respect to the beginning of the file.

docstring

The string that the test was extracted from, or `None` if the string is unavailable, or if the test was not extracted from a string.

Example Objects

class doctest.Example(*source*, *want*, *exc_msg=None*, *lineno=0*, *indent=0*, *options=None*)

A single interactive example, consisting of a Python statement and its expected output. The constructor arguments are used to initialize the attributes of the same names.

`Example` defines the following attributes. They are initialized by the constructor, and should not be modified directly.

source

A string containing the example's source code. This source code consists of a single Python statement, and always ends with a newline; the constructor adds a newline when necessary.

want

The expected output from running the example's source code (either from stdout, or a traceback in case of exception). `want` ends with a newline unless no output is expected, in which case it's an empty string. The constructor adds a newline when necessary.

exc_msg

The exception message generated by the example, if the example is expected to generate an exception; or `None` if it is not expected to generate an exception. This exception message is compared against the return value of `traceback.format_exception_only()`. `exc_msg` ends with a newline unless it's `None`. The constructor adds a newline if needed.

lineno

The line number within the string containing this example where the example begins. This line number is zero-based with respect to the beginning of the containing string.

indent

The example's indentation in the containing string, i.e., the number of space characters that precede the example's first prompt.

options

A dictionary mapping from option flags to `True` or `False`, which is used to override default options for this example. Any option flags not contained in this dictionary are left at their default value (as specified by the `DocTestRunner`'s optionflags). By default, no options are set.

DocTestFinder objects

class doctest.DocTestFinder(*verbose=False*, *parser=DocTestParser()*, *recurse=True*, *exclude_empty=True*)

A processing class used to extract the `DocTest`s that are relevant to a given object, from its docstring and the docstrings of its contained objects. `DocTest`s can be extracted from modules, classes, functions, methods, staticmethods, classmethods, and properties.

The optional argument *verbose* can be used to display the objects searched by the finder. It defaults to `False` (no output).

The optional argument *parser* specifies the `DocTestParser` object (or a drop-in replacement) that is used to extract doctests from docstrings.

If the optional argument *recurse* is false, then `DocTestFinder.find()` will only examine the given object, and not any contained objects.

If the optional argument *exclude_empty* is false, then `DocTestFinder.find()` will include tests for objects with empty docstrings.

`DocTestFinder` defines the following method:

find(*obj[, name][, module][, globs][, extraglobs]*)

Return a list of the `DocTest`s that are defined by *obj*'s docstring, or by any of its contained objects' docstrings.

The optional argument *name* specifies the object's name; this name will be used to construct names for the returned `DocTest`s. If *name* is not specified, then `obj.__name__` is used.

The optional parameter *module* is the module that contains the given object. If the module is not specified or is `None`, then the test finder will attempt to automatically determine the correct module. The object's module is used:

- As a default namespace, if *globs* is not specified.
- To prevent the DocTestFinder from extracting DocTests from objects that are imported from other modules. (Contained objects with modules other than *module* are ignored.)

- To find the name of the file containing the object.
- To help find the line number of the object within its file.

If *module* is `False`, no attempt to find the module will be made. This is obscure, of use mostly in testing doctest itself: if *module* is `False`, or is `None` but cannot be found automatically, then all objects are considered to belong to the (non-existent) module, so all contained objects will (recursively) be searched for doctests.

The globals for each `DocTest` is formed by combining *globs* and *extraglobs* (bindings in *extraglobs* override bindings in *globs*). A new shallow copy of the globals dictionary is created for each **`DocTest`**. If *globs* is not specified, then it defaults to the module's `__dict__`, if specified, or `{}` otherwise. If *extraglobs* is not specified, then it defaults to `{}`.

DocTestParser objects

class doctest.DocTestParser

A processing class used to extract interactive examples from a string, and use them to create a `DocTest` object.

`DocTestParser` defines the following methods:

get_doctest(*string*, *globs*, *name*, *filename*, *lineno*)

Extract all doctest examples from the given string, and collect them into a `DocTest` object.

globs, *name*, *filename*, and *lineno* are attributes for the new **`DocTest`** object. See the documentation for `DocTest` for more information.

get_examples(*string*, *name='<string>'*)

Extract all doctest examples from the given string, and return them as a list of `Example` objects. Line numbers are 0-based. The optional argument *name* is a name identifying this string, and is only used for error messages.

parse(*string*, *name='<string>'*)

Divide the given string into examples and intervening text, and return them as a list of alternating `Example`s and strings. Line numbers for the **`Example`**s are 0-based. The optional argument *name* is a name identifying this string, and is only used for error messages.

TestResults objects

class doctest.TestResults(*failed*, *attempted*)

failed

Number of failed tests.

attempted

Number of attempted tests.

skipped

Number of skipped tests.

Added in version 3.13.

DocTestRunner objects

class doctest.DocTestRunner(*checker=None*, *verbose=None*, *optionflags=0*)

A processing class used to execute and verify the interactive examples in a `DocTest`.

The comparison between expected outputs and actual outputs is done by an **`OutputChecker`**. This comparison may be customized with a number of option flags; see section Option Flags for more information. If the option flags are insufficient, then the comparison may also be customized by passing a subclass of **`OutputChecker`** to the constructor.

The test runner's display output can be controlled in two ways. First, an output function can be passed to **`run()`**; this function will be called with strings that should be displayed. It defaults to `sys.stdout.write`. If capturing the output is not sufficient, then the display output can be also customized by subclassing DocTestRunner, and overriding the methods **`report_start()`**, **`report_success()`**, **`report_unexpected_exception()`**, and **`report_failure()`**.

The optional keyword argument *checker* specifies the **`OutputChecker`** object (or drop-in replacement) that should be used to compare the expected outputs to the actual outputs of doctest examples.

The optional keyword argument *verbose* controls the **`DocTestRunner`**'s verbosity. If *verbose* is `True`, then information is printed about each example, as it is run. If *verbose* is `False`, then only failures are printed. If *verbose* is unspecified, or `None`, then verbose output is used iff the command-line switch `-v` is used.

The optional keyword argument *optionflags* can be used to control how the test runner compares expected output to actual output, and how it displays failures. For more information, see section Option Flags.

The test runner accumulates statistics. The aggregated number of attempted, failed and skipped examples is also available via the **`tries`**, **`failures`** and **`skips`** attributes.

The `run()` and `summarize()` methods return a `TestResults` instance.

`DocTestRunner` defines the following methods:

report_start(*out*, *test*, *example*)

Report that the test runner is about to process the given example. This method is provided to allow subclasses of `DocTestRunner` to customize their output; it should not be called directly.

example is the example about to be processed. *test* is the test containing *example*. *out* is the output function that was passed to `DocTestRunner.run()`.

report_success(*out*, *test*, *example*, *got*)

Report that the given example ran successfully. This method is provided to allow subclasses of `DocTestRunner` to customize their output; it should not be called directly.

example is the example about to be processed. *got* is the actual output from the example. *test* is the test containing *example*. *out* is the output function that was passed to `DocTestRunner.run()`.

report_failure(*out*, *test*, *example*, *got*)

Report that the given example failed. This method is provided to allow subclasses of `DocTestRunner` to customize their output; it should not be called directly.

example is the example about to be processed. *got* is the actual output from the example. *test* is the test containing *example*. *out* is the output function that was passed to `DocTestRunner.run()`.

report_unexpected_exception(*out*, *test*, *example*, *exc_info*)

Report that the given example raised an unexpected exception. This method is provided to allow subclasses of `DocTestRunner` to customize their output; it should not be called directly.

example is the example about to be processed. *exc_info* is a tuple containing information about the unexpected exception (as returned by `sys.exc_info()`). *test* is the test containing *example*. *out* is the output function that was passed to `DocTestRunner.run()`.

run(*test*, *compileflags=None*, *out=None*, *clear_globs=True*)

Run the examples in *test* (a `DocTest` object), and display the results using the writer function *out*. Return a `TestResults` instance.

The examples are run in the namespace `test.globs`. If *clear_globs* is true (the default), then this namespace will be cleared after the test runs, to help with garbage collection. If you would like to examine the namespace after the test completes, then use *clear_globs=False*.

compileflags gives the set of flags that should be used by the Python compiler when running the examples. If not specified, then it will default to the set of future-import flags that apply to *globs*.

The output of each example is checked using the `DocTestRunner`'s output checker, and the results are formatted by the **`DocTestRunner.report_*()`** methods.

summarize(*verbose=None*)

Print a summary of all the test cases that have been run by this DocTestRunner, and return a `TestResults` instance.

The optional *verbose* argument controls how detailed the summary is. If the verbosity is not specified, then the `DocTestRunner`'s verbosity is used.

`DocTestParser` has the following attributes:

tries

Number of attempted examples.

failures

Number of failed examples.

skips

Number of skipped examples.

Added in version 3.13.

OutputChecker objects

class doctest.OutputChecker

A class used to check the whether the actual output from a doctest example matches the expected output. `OutputChecker` defines two methods: `check_output()`, which compares a given pair of outputs, and returns `True` if they match; and `output_difference()`, which returns a string describing the differences between two outputs.

`OutputChecker` defines the following methods:

check_output(*want*, *got*, *optionflags*)

Return `True` iff the actual output from an example (*got*) matches the expected output (*want*). These strings are always considered to match if they are identical; but depending on what option flags the test runner is using, several non-exact match types are also possible. See section Option Flags for more information about option flags.

output_difference(*example*, *got*, *optionflags*)

Return a string describing the differences between the expected output for a given example (*example*) and the actual output (*got*). *optionflags* is the set of option flags used to compare *want* and *got*.

Debugging

Doctest provides several mechanisms for debugging doctest examples:

- Several functions convert doctests to executable Python programs, which can be run under the Python debugger, **`pdb`**.
- The **`DebugRunner`** class is a subclass of **`DocTestRunner`** that raises an exception for the first failing example, containing information about that example. This information can be used to perform post-mortem debugging on the example.
- The **`unittest`** cases generated by **`DocTestSuite()`** support the **`debug()`** method defined by **`unittest.TestCase`**.
- You can add a call to **`pdb.set_trace()`** in a doctest example, and you’ll drop into the Python debugger when that line is executed. Then you can inspect current values of variables, and so on. For example, suppose `a.py` contains just this module docstring:

```
"""
>>> def f(x):
...     g(x*2)
>>> def g(x):
...     print(x+3)
...     import pdb; pdb.set_trace()
>>> f(3)
9
"""
```

Then an interactive Python session may look like this:

```
>>> import a, doctest
>>> doctest.testmod(a)
--Return--
> <doctest a[1]>(3)g()->None
-> import pdb; pdb.set_trace()
(Pdb) list
  1     def g(x):
  2         print(x+3)
  3  ->     import pdb; pdb.set_trace()
[EOF]
(Pdb) p x
6
(Pdb) step
--Return--
> <doctest a[0]>(2)f()->None
-> g(x*2)
(Pdb) list
  1     def f(x):
  2  ->     g(x*2)
[EOF]
(Pdb) p x
3
(Pdb) step
--Return--
> <doctest a[2]>(1)?()->None
-> f(3)
(Pdb) cont
(0, 3)
>>>
```

Functions that convert doctests to Python code, and possibly run the synthesized code under the debugger:

doctest.script_from_examples(*s*)

Convert text with examples to a script.

Argument *s* is a string containing doctest examples. The string is converted to a Python script, where doctest examples in *s* are converted to regular code, and everything else is converted to Python comments. The generated script is returned as a string. For example,

```
import doctest
print(doctest.script_from_examples(r"""
    Set x and y to 1 and 2.
    >>> x, y = 1, 2

    Print their sum:
    >>> print(x+y)
    3
"""))
```

displays:

```
# Set x and y to 1 and 2.
x, y = 1, 2
#
# Print their sum:
print(x+y)
# Expected:
## 3
```

This function is used internally by other functions (see below), but can also be useful when you want to transform an interactive Python session into a Python script.

doctest.testsource(*module*, *name*)

Convert the doctest for an object to a script.

Argument *module* is a module object, or dotted name of a module, containing the object whose doctests are of interest. Argument *name* is the name (within the module) of the object with the doctests of interest. The result is a string, containing the object's docstring converted to a Python script, as described for `script_from_examples()` above. For example, if module `a.py` contains a top-level function **`f()`**, then

```
import a, doctest
print(doctest.testsource(a, "a.f"))
```

prints a script version of function **`f()`**'s docstring, with doctests converted to code, and the rest placed in comments.

doctest.debug(*module*, *name*, *pm=False*)

Debug the doctests for an object.

The *module* and *name* arguments are the same as for function `testsource()` above. The synthesized Python script for the named object's docstring is written to a temporary file, and then that file is run under the control of the Python debugger, `pdb`.

A shallow copy of `module.__dict__` is used for both local and global execution context.

Optional argument *pm* controls whether post-mortem debugging is used. If *pm* has a true value, the script file is run directly, and the debugger gets involved only if the script terminates via raising an unhandled exception. If it does, then post-mortem debugging is invoked, via `pdb.post_mortem()`, passing the traceback object from the unhandled exception. If *pm* is not specified, or is false, the script is run under the debugger from the start, via passing an appropriate `exec()` call to `pdb.run()`.

doctest.debug_src(*src*, *pm=False*, *globs=None*)

Debug the doctests in a string.

This is like function **`debug()`** above, except that a string containing doctest examples is specified directly, via the *src* argument.

Optional argument *pm* has the same meaning as in function **`debug()`** above.

Optional argument *globs* gives a dictionary to use as both local and global execution context. If not specified, or `None`, an empty dictionary is used. If specified, a shallow copy of the dictionary is used.

The **`DebugRunner`** class, and the special exceptions it may raise, are of most interest to testing framework authors, and will only be sketched here. See the source code, and especially **`DebugRunner`**’s docstring (which is a doctest!) for more details:

class doctest.DebugRunner(*checker=None*, *verbose=None*, *optionflags=0*)

A subclass of **`DocTestRunner`** that raises an exception as soon as a failure is encountered. If an unexpected exception occurs, an **`UnexpectedException`** exception is raised, containing the test, the example, and the original exception. If the output doesn’t match, then a **`DocTestFailure`** exception is raised, containing the test, the example, and the actual output.

For information about the constructor parameters and methods, see the documentation for **`DocTestRunner`** in section Advanced API.

There are two exceptions that may be raised by `DebugRunner` instances:

exception doctest.DocTestFailure(*test*, *example*, *got*)

An exception raised by `DocTestRunner` to signal that a doctest example's actual output did not match its expected output. The constructor arguments are used to initialize the attributes of the same names.

`DocTestFailure` defines the following attributes:

DocTestFailure.test

The `DocTest` object that was being run when the example failed.

DocTestFailure.example

The `Example` that failed.

DocTestFailure.got

The example's actual output.

exception doctest.UnexpectedException(*test*, *example*, *exc_info*)

An exception raised by `DocTestRunner` to signal that a doctest example raised an unexpected exception. The constructor arguments are used to initialize the attributes of the same names.

`UnexpectedException` defines the following attributes:

UnexpectedException.test

The `DocTest` object that was being run when the example failed.

UnexpectedException.example

The `Example` that failed.

UnexpectedException.exc_info

A tuple containing information about the unexpected exception, as returned by `sys.exc_info()`.

Soapbox

As mentioned in the introduction, `doctest` has grown to have three primary uses:

1. Checking examples in docstrings.
2. Regression testing.
3. Executable documentation / literate testing.

These uses have different requirements, and it is important to distinguish them. In particular, filling your docstrings with obscure test cases makes for bad documentation.

When writing a docstring, choose docstring examples with care. There’s an art to this that needs to be learned—it may not be natural at first. Examples should add genuine value to the documentation. A good example can often be worth many words. If done with care, the examples will be invaluable for your users, and will pay back the time it takes to collect them many times over as the years go by and things change. I’m still amazed at how often one of my `doctest` examples stops working after a “harmless” change.

Doctest also makes an excellent tool for regression testing, especially if you don’t skimp on explanatory text. By interleaving prose and examples, it becomes much easier to keep track of what’s actually being tested, and why. When a test fails, good prose can make it much easier to figure out what the problem is, and how it should be fixed. It’s true that you could write extensive comments in code-based testing, but few programmers do. Many have found that using doctest approaches instead leads to much clearer tests. Perhaps this is simply because doctest makes writing prose a little easier than writing code, while writing comments in code is a little harder. I think it goes deeper than just that: the natural attitude when writing a doctest-based test is that you want to explain the fine points of your software, and illustrate them with examples. This in turn naturally leads to test files that start with the simplest features, and logically progress to complications and edge cases. A coherent narrative is the result, instead of a collection of isolated functions that test isolated bits of functionality seemingly at random. It’s a different attitude, and produces different results, blurring the distinction between testing and explaining.

Regression testing is best confined to dedicated objects or files. There are several options for organizing tests:

- Write text files containing test cases as interactive examples, and test the files using `testfile()` or `DocFileSuite()`. This is recommended, although is easiest to do for new projects, designed from the start to use doctest.
- Define functions named `_regrtest_topic` that consist of single docstrings, containing test cases for the named topics. These functions can be included in the same file as the module, or separated out into a separate test file.
- Define a `__test__` dictionary mapping from regression test topics to docstrings containing test cases.

When you have placed your tests in a module, the module can itself be the test runner. When a test fails, you can arrange for your test runner to re-run only the failing doctest while you debug the problem. Here is a minimal example of such a test runner:

```
if __name__ == '__main__':
    import doctest
    flags = doctest.REPORT_NDIFF|doctest.FAIL_FAST
    if len(sys.argv) > 1:
        name = sys.argv[1]
        if name in globals():
            obj = globals()[name]
        else:
            obj = __test__[name]
        doctest.run_docstring_examples(obj, globals(), 
name=name,
                                       optionflags=flags)
    else:
        fail, total = doctest.testmod(optionflags=flags)
        print(f"{fail} failures out of {total} tests")
```

Footnotes

[1] Examples containing both expected output and an exception are not supported. Trying to guess where one ends and the other begins is too error-prone, and that also makes for a confusing test.

`unittest` — Unit testing framework

Source code: Lib/unittest/__init__.py
[https://github.com/python/cpython/tree/3.14/Lib/unittest/__init__.py]

(If you are already familiar with the basic concepts of testing, you might want to skip to the list of assert methods.)

The **`unittest`** unit testing framework was originally inspired by JUnit and has a similar flavor as major unit testing frameworks in other languages. It supports test automation, sharing of setup and shutdown code for tests, aggregation of tests into collections, and independence of the tests from the reporting framework.

To achieve this, **`unittest`** supports some important concepts in an object-oriented way:

test fixture
: A *test fixture* represents the preparation needed to perform one or more tests, and any associated cleanup actions. This may involve, for example, creating temporary or proxy databases, directories, or starting a server process.

test case
: A *test case* is the individual unit of testing. It checks for a specific response to a particular set of inputs. **`unittest`** provides a base class, **`TestCase`**, which may be used to create new test cases.

test suite
: A *test suite* is a collection of test cases, test suites, or both. It is used to aggregate tests that should be executed together.

test runner
: A *test runner* is a component which orchestrates the execution of tests and provides the outcome to the user. The runner may use a graphical interface, a textual interface, or return a special value to indicate the results of executing the tests.

See also

Module `doctest`

Another test-support module with a very different flavor.

Simple Smalltalk Testing: With Patterns [https://web.archive.org/web/20150315073817/http://www.xprogramming.com/testfram.htm]

Kent Beck's original paper on testing frameworks using the pattern shared by `unittest`.

pytest [https://docs.pytest.org/]

Third-party unittest framework with a lighter-weight syntax for writing tests. For example, `assert func(10) == 42`.

The Python Testing Tools Taxonomy [https://wiki.python.org/moin/PythonTestingToolsTaxonomy]

An extensive list of Python testing tools including functional testing frameworks and mock object libraries.

Testing in Python Mailing List [http://lists.idyll.org/listinfo/testing-in-python]

> A special-interest-group for discussion of testing, and testing tools, in Python.

The script `Tools/unittestgui/unittestgui.py` in the Python source distribution is a GUI tool for test discovery and execution. This is intended largely for ease of use for those new to unit testing. For production environments it is recommended that tests be driven by a continuous integration system such as Buildbot [https://buildbot.net/], Jenkins [https://www.jenkins.io/], GitHub Actions [https://github.com/features/actions], or AppVeyor [https://www.appveyor.com/].

Basic example

The **`unittest`** module provides a rich set of tools for constructing and running tests. This section demonstrates that a small subset of the tools suffice to meet the needs of most users.

Here is a short script to test three string methods:

```
import unittest

class TestStringMethods(unittest.TestCase):

    def test_upper(self):
        self.assertEqual('foo'.upper(), 'FOO')

    def test_isupper(self):
        self.assertTrue('FOO'.isupper())
        self.assertFalse('Foo'.isupper())

    def test_split(self):
        s = 'hello world'
        self.assertEqual(s.split(), ['hello', 'world'])
        # check that s.split fails when the separator is not a
```

```
string
        with self.assertRaises(TypeError):
            s.split(2)

if __name__ == '__main__':
    unittest.main()
```

A test case is created by subclassing **`unittest.TestCase`**. The three individual tests are defined with methods whose names start with the letters `test`. This naming convention informs the test runner about which methods represent tests.

The crux of each test is a call to **`assertEqual()`** to check for an expected result; **`assertTrue()`** or **`assertFalse()`** to verify a condition; or **`assertRaises()`** to verify that a specific exception gets raised. These methods are used instead of the **`assert`** statement so the test runner can accumulate all test results and produce a report.

The **`setUp()`** and **`tearDown()`** methods allow you to define instructions that will be executed before and after each test method. They are covered in more detail in the section Organizing test code.

The final block shows a simple way to run the tests. **`unittest.main()`** provides a command-line interface to the test script. When run from the command line, the above script produces an output that looks like this:

```
...
---------------------------------------------------------------
-------
Ran 3 tests in 0.000s

OK
```

Passing the `-v` option to your test script will instruct **`unittest.main()`** to enable a higher level of verbosity, and produce the following output:

```
test_isupper (__main__.TestStringMethods.test_isupper) ... ok
test_split (__main__.TestStringMethods.test_split) ... ok
test_upper (__main__.TestStringMethods.test_upper) ... ok

---------------------------------------------------------------
-------
Ran 3 tests in 0.001s

OK
```

The above examples show the most commonly used **`unittest`** features which are sufficient to meet many everyday testing needs. The remainder of the documentation explores the full feature set from first principles.

Changed in version 3.11: The behavior of returning a value from a test method (other than the default `None` value), is now deprecated.

Command-Line Interface

The unittest module can be used from the command line to run tests from modules, classes or even individual test methods:

```
python -m unittest test_module1 test_module2
python -m unittest test_module.TestClass
python -m unittest test_module.TestClass.test_method
```

You can pass in a list with any combination of module names, and fully qualified class or method names.

Test modules can be specified by file path as well:

```
python -m unittest tests/test_something.py
```

This allows you to use the shell filename completion to specify the test module. The file specified must still be importable as a module. The path is converted to a module name by removing the ‘.py’ and converting path separators into ‘.’. If you want to execute a test file that isn’t importable as a module you should execute the file directly instead.

You can run tests with more detail (higher verbosity) by passing in the -v flag:

```
python -m unittest -v test_module
```

When executed without arguments Test Discovery is started:

```
python -m unittest
```

For a list of all the command-line options:

```
python -m unittest -h
```

Changed in version 3.2: In earlier versions it was only possible to run individual test methods and not modules or classes.

Added in version 3.14: Output is colorized by default and can be controlled using environment variables.

Command-line options

unittest supports these command-line options:

-b, --buffer

The standard output and standard error streams are buffered during the test run. Output during a passing test is discarded. Output is echoed normally on test fail or error and is added to the failure messages.

-c, --catch

`Control-C` during the test run waits for the current test to end and then reports all the results so far. A second `Control-C` raises the normal **`KeyboardInterrupt`** exception.

See Signal Handling for the functions that provide this functionality.

-f, --failfast

Stop the test run on the first error or failure.

-k

Only run test methods and classes that match the pattern or substring. This option may be used multiple times, in which case all test cases that match any of the given patterns are included.

Patterns that contain a wildcard character (`*`) are matched against the test name using **`fnmatch.fnmatchcase()`**; otherwise simple case-sensitive substring matching is used.

Patterns are matched against the fully qualified test method name as imported by the test loader.

For example, `-k foo` matches `foo_tests.SomeTest.test_something`, `bar_tests.SomeTest.test_foo`, but not `bar_tests.FooTest.test_something`.

--locals

Show local variables in tracebacks.

--durations N

Show the N slowest test cases (N=0 for all).

Added in version 3.2: The command-line options `-b`, `-c` and `-f` were added.

Added in version 3.5: The command-line option `--locals`.

Added in version 3.7: The command-line option `-k`.

Added in version 3.12: The command-line option `--durations`.

The command line can also be used for test discovery, for running all of the tests in a project or just a subset.

Test Discovery

Added in version 3.2.

Unittest supports simple test discovery. In order to be compatible with test discovery, all of the test files must be modules or packages importable from the top-level directory of the project (this means that their filenames must be valid identifiers).

Test discovery is implemented in **`TestLoader.discover()`**, but can also be used from the command line. The basic command-line usage is:

```
cd project_directory
python -m unittest discover
```

Note

As a shortcut, `python -m unittest` is the equivalent of `python -m unittest discover`. If you want to pass arguments to test discovery the `discover` sub-command must be used explicitly.

The `discover` sub-command has the following options:

-v, --verbose
    Verbose output

-s, --start-directory directory
    Directory to start discovery (. default)

-p, --pattern pattern
    Pattern to match test files (`test*.py` default)

-t, --top-level-directory directory
    Top level directory of project (defaults to start directory)

The `-s`, `-p`, and `-t` options can be passed in as positional arguments in that order. The following two command lines are equivalent:

```
python -m unittest discover -s project_directory -p "*_test.py"
python -m unittest discover project_directory "*_test.py"
```

As well as being a path it is possible to pass a package name, for example `myproject.subpackage.test`, as the start directory. The package name you supply will then be imported and its location on the filesystem will be used as the start directory.

Caution

Test discovery loads tests by importing them. Once test discovery has found all the test files from the start directory you specify it turns the paths into package names to import. For example `foo/bar/baz.py` will be imported as `foo.bar.baz`.

If you have a package installed globally and attempt test discovery on a different copy of the package then the import *could* happen from the wrong place. If this happens test discovery will warn you and exit.

If you supply the start directory as a package name rather than a path to a directory then discover assumes that whichever location it imports from is the location you intended, so you will not get the warning.

Test modules and packages can customize test loading and discovery by through the load_tests protocol.

Changed in version 3.4: Test discovery supports namespace packages.

Changed in version 3.11: Test discovery dropped the namespace packages support. It has been broken since Python 3.7. Start directory and its subdirectories containing tests must be regular package that have `__init__.py` file.

If the start directory is the dotted name of the package, the ancestor packages can be namespace packages.

Changed in version 3.14: Test discovery supports namespace package as start directory again. To avoid scanning directories unrelated to Python, tests are not searched in subdirectories that do not contain `__init__.py`.

Organizing test code

The basic building blocks of unit testing are *test cases* — single scenarios that must be set up and checked for correctness. In **`unittest`**, test cases are represented by **`unittest.TestCase`** instances. To make your own test cases you must write subclasses of **`TestCase`** or use **`FunctionTestCase`**.

The testing code of a **`TestCase`** instance should be entirely self contained, such that it can be run either in isolation or in arbitrary combination with any number of other test cases.

The simplest **`TestCase`** subclass will simply implement a test method (i.e. a method whose name starts with `test`) in order to perform specific testing code:

```
import unittest

class DefaultWidgetSizeTestCase(unittest.TestCase):
    def test_default_widget_size(self):
        widget = Widget('The widget')
        self.assertEqual(widget.size(), (50, 50))
```

Note that in order to test something, we use one of the assert* methods provided by the **`TestCase`** base class. If the test fails, an exception will be raised with an explanatory message, and **`unittest`** will identify the test case as a *failure*. Any other exceptions will be treated as *errors*.

Tests can be numerous, and their set-up can be repetitive. Luckily, we can factor out set-up code by implementing a method called **`setUp()`**, which the testing framework will automatically call for every single test we run:

```
import unittest

class WidgetTestCase(unittest.TestCase):
    def setUp(self):
        self.widget = Widget('The widget')

    def test_default_widget_size(self):
        self.assertEqual(self.widget.size(), (50,50),
                         'incorrect default size')

    def test_widget_resize(self):
        self.widget.resize(100,150)
        self.assertEqual(self.widget.size(), (100,150),
                         'wrong size after resize')
```

Note

The order in which the various tests will be run is determined by sorting the test method names with respect to the built-in ordering for strings.

If the `setUp()` method raises an exception while the test is running, the framework will consider the test to have suffered an error, and the test method will not be executed.

Similarly, we can provide a `tearDown()` method that tidies up after the test method has been run:

```
import unittest

class WidgetTestCase(unittest.TestCase):
    def setUp(self):
        self.widget = Widget('The widget')

    def tearDown(self):
        self.widget.dispose()
```

If `setUp()` succeeded, `tearDown()` will be run whether the test method succeeded or not.

Such a working environment for the testing code is called a *test fixture*. A new TestCase instance is created as a unique test fixture used to execute each individual test method. Thus `setUp()`, `tearDown()`, and **`TestCase.__init__()`** will be called once per test.

It is recommended that you use TestCase implementations to group tests together according to the features they test. `unittest` provides a mechanism for this: the *test suite*, represented by `unittest`’s `TestSuite` class. In most cases, calling `unittest.main()` will do the right thing and collect all the module’s test cases for you and execute them.

However, should you want to customize the building of your test suite, you can do it yourself:

```
def suite():
    suite = unittest.TestSuite()
    suite.addTest(WidgetTestCase('test_default_widget_size'))
    suite.addTest(WidgetTestCase('test_widget_resize'))
    return suite

if __name__ == '__main__':
    runner = unittest.TextTestRunner()
    runner.run(suite())
```

You can place the definitions of test cases and test suites in the same modules as the code they are to test (such as `widget.py`), but there are several advantages to placing the test code in a separate module, such as `test_widget.py`:

- The test module can be run standalone from the command line.
- The test code can more easily be separated from shipped code.
- There is less temptation to change test code to fit the code it tests without a good reason.
- Test code should be modified much less frequently than the code it tests.
- Tested code can be refactored more easily.
- Tests for modules written in C must be in separate modules anyway, so why not be consistent?
- If the testing strategy changes, there is no need to change the source code.

Re-using old test code

Some users will find that they have existing test code that they would like to run from `unittest`, without converting every old test function to a `TestCase` subclass.

For this reason, `unittest` provides a `FunctionTestCase` class. This subclass of `TestCase` can be used to wrap an existing test function. Set-up and tear-down functions can also be provided.

Given the following test function:

```
def testSomething():
    something = makeSomething()
    assert something.name is not None
    # ...
```

one can create an equivalent test case instance as follows, with optional set-up and tear-down methods:

```
testcase = unittest.FunctionTestCase(testSomething,
                                     setUp=makeSomethingDB,

tearDown=deleteSomethingDB)
```

Note

Even though `FunctionTestCase` can be used to quickly convert an existing test base over to a `unittest`-based system, this approach is not recommended. Taking the time to set up proper `TestCase` subclasses will make future test refactorings infinitely easier.

In some cases, the existing tests may have been written using the `doctest` module. If so, `doctest` provides a `DocTestSuite` class that can automatically build `unittest.TestSuite` instances from the existing `doctest`-based tests.

Skipping tests and expected failures

Added in version 3.1.

Unittest supports skipping individual test methods and even whole classes of tests. In addition, it supports marking a test as an “expected failure,” a test that is broken and will fail, but shouldn’t be counted as a failure on a **`TestResult`**.

Skipping a test is simply a matter of using the **`skip()`** decorator or one of its conditional variants, calling **`TestCase.skipTest()`** within a **`setUp()`** or test method, or raising **`SkipTest`** directly.

Basic skipping looks like this:

```
class MyTestCase(unittest.TestCase):

    @unittest.skip("demonstrating skipping")
    def test_nothing(self):
        self.fail("shouldn't happen")

    @unittest.skipIf(mylib.__version__ < (1, 3),
                     "not supported in this library version")
    def test_format(self):
        # Tests that work for only a certain version of the
library.
        pass

    @unittest.skipUnless(sys.platform.startswith("win"),
"requires Windows")
    def test_windows_support(self):
        # windows specific testing code
        pass

    def test_maybe_skipped(self):
        if not external_resource_available():
            self.skipTest("external resource not available")
        # test code that depends on the external resource
        pass
```

This is the output of running the example above in verbose mode:

```
test_format (__main__.MyTestCase.test_format) ... skipped 'not
supported in this library version'
test_nothing (__main__.MyTestCase.test_nothing) ... skipped
'demonstrating skipping'
test_maybe_skipped (__main__.MyTestCase.test_maybe_skipped) ...
skipped 'external resource not available'
test_windows_support (__main__.MyTestCase.test_windows_support)
... skipped 'requires Windows'

---------------------------------------------------------------
-------
Ran 4 tests in 0.005s

OK (skipped=4)
```

Classes can be skipped just like methods:

```
@unittest.skip("showing class skipping")
class MySkippedTestCase(unittest.TestCase):
    def test_not_run(self):
        pass
```

`TestCase.setUp()` can also skip the test. This is useful when a resource that needs to be set up is not available.

Expected failures use the **`expectedFailure()`** decorator.

```
class ExpectedFailureTestCase(unittest.TestCase):
    @unittest.expectedFailure
    def test_fail(self):
        self.assertEqual(1, 0, "broken")
```

It’s easy to roll your own skipping decorators by making a decorator that calls **`skip()`** on the test when it wants it to be skipped. This decorator skips the test unless the passed object has a certain attribute:

```
def skipUnlessHasattr(obj, attr):
    if hasattr(obj, attr):
        return lambda func: func
    return unittest.skip("{!r} doesn't have {!r}".format(obj,
attr))
```

The following decorators and exception implement test skipping and expected failures:

@unittest.skip(*reason*)

Unconditionally skip the decorated test. *reason* should describe why the test is being skipped.

@unittest.skipIf(*condition*, *reason*)

Skip the decorated test if *condition* is true.

@unittest.skipUnless(*condition*, *reason*)

Skip the decorated test unless *condition* is true.

@unittest.expectedFailure

Mark the test as an expected failure or error. If the test fails or errors in the test function itself (rather than in one of the *test fixture* methods) then it will be considered a success. If the test passes, it will be considered a failure.

exception unittest.SkipTest(*reason*)

This exception is raised to skip a test.

Usually you can use `TestCase.skipTest()` or one of the skipping decorators instead of raising this directly.

Skipped tests will not have `setUp()` or `tearDown()` run around them. Skipped classes will not have `setUpClass()` or `tearDownClass()` run. Skipped modules will not have `setUpModule()` or `tearDownModule()` run.

Distinguishing test iterations using subtests

Added in version 3.4.

When there are very small differences among your tests, for instance some parameters, unittest allows you to distinguish them inside the body of a test method using the `subTest()` context manager.

For example, the following test:

```
class NumbersTest(unittest.TestCase):

    def test_even(self):
        """
        Test that numbers between 0 and 5 are all even.
        """
        for i in range(0, 6):
            with self.subTest(i=i):
                self.assertEqual(i % 2, 0)
```

will produce the following output:

```
===============================================================
=======
FAIL: test_even (__main__.NumbersTest.test_even) (i=1)
Test that numbers between 0 and 5 are all even.
---------------------------------------------------------------
-------
Traceback (most recent call last):
  File "subtests.py", line 11, in test_even
    self.assertEqual(i % 2, 0)
    ^^^^^^^^^^^^^^^^^^^^^^^^^^
AssertionError: 1 != 0

===============================================================
=======
FAIL: test_even (__main__.NumbersTest.test_even) (i=3)
Test that numbers between 0 and 5 are all even.
```

```
---------------------------------------------------------------
-------
Traceback (most recent call last):
  File "subtests.py", line 11, in test_even
    self.assertEqual(i % 2, 0)
    ^^^^^^^^^^^^^^^^^^^^^^^^^^
AssertionError: 1 != 0

===============================================================
=======
FAIL: test_even (__main__.NumbersTest.test_even) (i=5)
Test that numbers between 0 and 5 are all even.
---------------------------------------------------------------
-------
Traceback (most recent call last):
  File "subtests.py", line 11, in test_even
    self.assertEqual(i % 2, 0)
    ^^^^^^^^^^^^^^^^^^^^^^^^^^
AssertionError: 1 != 0
```

Without using a subtest, execution would stop after the first failure, and the error would be less easy to diagnose because the value of `i` wouldn’t be displayed:

```
===============================================================
=======
FAIL: test_even (__main__.NumbersTest.test_even)
---------------------------------------------------------------
-------
Traceback (most recent call last):
  File "subtests.py", line 32, in test_even
    self.assertEqual(i % 2, 0)
AssertionError: 1 != 0
```

Classes and functions

This section describes in depth the API of `unittest`.

Test cases

class unittest.TestCase(*methodName='runTest'*)

Instances of the `TestCase` class represent the logical test units in the `unittest` universe. This class is intended to be used as a base class, with specific tests being implemented by concrete subclasses. This class implements the interface needed by the test runner to allow it to drive the tests, and methods that the test code can use to check for and report various kinds of failure.

Each instance of `TestCase` will run a single base method: the method named *methodName*. In most uses of `TestCase`, you will neither change the *methodName* nor reimplement the default `runTest()` method.

Changed in version 3.2: `TestCase` can be instantiated successfully without providing a *methodName*. This makes it easier to experiment with `TestCase` from the interactive interpreter.

`TestCase` instances provide three groups of methods: one group used to run the test, another used by the test implementation to check conditions and report failures, and some inquiry methods allowing information about the test itself to be gathered.

Methods in the first group (running the test) are:

setUp()

Method called to prepare the test fixture. This is called immediately before calling the test method; other than `AssertionError` or `SkipTest`, any exception raised by this method will be considered an error rather than a test failure. The default implementation does nothing.

tearDown()

Method called immediately after the test method has been called and the result recorded. This is called even if the test method raised an exception, so the implementation in subclasses may need to be particularly careful about checking internal state. Any exception, other than **`AssertionError`** or **`SkipTest`**, raised by this method will be considered an additional error rather than a test failure (thus increasing the total number of reported errors). This method will only be called if the **`setUp()`** succeeds, regardless of the outcome of the test method. The default implementation does nothing.

setUpClass()

A class method called before tests in an individual class are run. `setUpClass` is called with the class as the only argument and must be decorated as a **`classmethod()`**:

```
@classmethod
def setUpClass(cls):
    ...
```

See Class and Module Fixtures for more details.

Added in version 3.2.

tearDownClass()

A class method called after tests in an individual class have run. `tearDownClass` is called with the class as the only argument and must be decorated as a **`classmethod()`**:

```
@classmethod
def tearDownClass(cls):
    ...
```

See Class and Module Fixtures for more details.

Added in version 3.2.

run(*result=None*)

Run the test, collecting the result into the `TestResult` object passed as *result*. If *result* is omitted or `None`, a temporary result object is created (by calling the `defaultTestResult()` method) and used. The result object is returned to `run()`'s caller.

The same effect may be had by simply calling the `TestCase` instance.

Changed in version 3.3: Previous versions of `run` did not return the result. Neither did calling an instance.

skipTest(*reason*)

Calling this during a test method or `setUp()` skips the current test. See Skipping tests and expected failures for more information.

Added in version 3.1.

subTest(*msg=None*, ***params*)

Return a context manager which executes the enclosed code block as a subtest. *msg* and *params* are optional, arbitrary values which are displayed whenever a subtest fails, allowing you to identify them clearly.

A test case can contain any number of subtest declarations, and they can be arbitrarily nested.

See Distinguishing test iterations using subtests for more information.

Added in version 3.4.

debug()

Run the test without collecting the result. This allows exceptions raised by the test to be propagated to the caller, and can be used to support running tests under a debugger.

The `TestCase` class provides several assert methods to check for and report failures. The following table lists the most commonly used methods (see the tables below for more assert methods):

Method	Checks that	New in
`assertEqual(a, b)`	`a == b`	
`assertNotEqual(a, b)`	`a != b`	
`assertTrue(x)`	`bool(x) is True`	
`assertFalse(x)`	`bool(x) is False`	
`assertIs(a, b)`	`a is b`	3.1
`assertIsNot(a, b)`	`a is not b`	3.1
`assertIsNone(x)`	`x is None`	3.1
`assertIsNotNone(x)`	`x is not None`	3.1
`assertIn(a, b)`	`a in b`	3.1
`assertNotIn(a, b)`	`a not in b`	3.1
`assertIsInstance(a, b)`	`isinstance(a, b)`	3.2
`assertNotIsInstance(a, b)`	`not isinstance(a, b)`	3.2

Method	Checks that	New in
`assertIsSubclass(a, b)`	`issubclass(a, b)`	3.14
`assertNotIsSubclass(a, b)`	`not issubclass(a, b)`	3.14

All the assert methods accept a *msg* argument that, if specified, is used as the error message on failure (see also `longMessage`). Note that the *msg* keyword argument can be passed to `assertRaises()`, `assertRaisesRegex()`, `assertWarns()`, `assertWarnsRegex()` only when they are used as a context manager.

assertEqual(*first*, *second*, *msg=None*)

Test that *first* and *second* are equal. If the values do not compare equal, the test will fail.

In addition, if *first* and *second* are the exact same type and one of list, tuple, dict, set, frozenset or str or any type that a subclass registers with `addTypeEqualityFunc()` the type-specific equality function will be called in order to generate a more useful default error message (see also the list of type-specific methods).

Changed in version 3.1: Added the automatic calling of type-specific equality function.

Changed in version 3.2: `assertMultiLineEqual()` added as the default type equality function for comparing strings.

assertNotEqual(*first*, *second*, *msg=None*)

Test that *first* and *second* are not equal. If the values do compare equal, the test will fail.

assertTrue(*expr*, *msg=None*)
assertFalse(*expr*, *msg=None*)

Test that *expr* is true (or false).

Note that this is equivalent to `bool(expr) is True` and not to `expr is True` (use `assertIs(expr, True)` for the latter). This method should also be avoided when more specific methods are available (e.g. `assertEqual(a, b)` instead of `assertTrue(a == b)`), because they provide a better error message in case of failure.

assertIs(*first*, *second*, *msg=None*)
assertIsNot(*first*, *second*, *msg=None*)

Test that *first* and *second* are (or are not) the same object.

Added in version 3.1.

assertIsNone(*expr*, *msg=None*)
assertIsNotNone(*expr*, *msg=None*)

Test that *expr* is (or is not) `None`.

Added in version 3.1.

assertIn(*member*, *container*, *msg=None*)
assertNotIn(*member*, *container*, *msg=None*)

Test that *member* is (or is not) in *container*.

Added in version 3.1.

assertIsInstance(*obj*, *cls*, *msg=None*)
assertNotIsInstance(*obj*, *cls*, *msg=None*)

Test that *obj* is (or is not) an instance of *cls* (which can be a class or a tuple of classes, as supported by `isinstance()`). To check for the exact type, use `assertIs(type(obj), cls)`.

Added in version 3.2.

assertIsSubclass(*cls*, *superclass*, *msg=None*)
assertNotIsSubclass(*cls*, *superclass*, *msg=None*)

Test that *cls* is (or is not) a subclass of *superclass* (which can be a class or a tuple of classes, as supported by `issubclass()`). To check for the exact type, use `assertIs(cls, superclass)`.

Added in version 3.14.

It is also possible to check the production of exceptions, warnings, and log messages using the following methods:

Method	**Checks that**	**New in**
`assertRaises(exc, fun, *args, **kwds)`	`fun(*args, **kwds)` raises *exc*	
`assertRaisesRegex(exc, r, fun, *args, **kwds)`	`fun(*args, **kwds)` raises *exc* and the message matches regex *r*	3.1
`assertWarns(warn, fun, *args, **kwds)`	`fun(*args, **kwds)` raises *warn*	3.2
`assertWarnsRegex(warn, r, fun, *args, **kwds)`	`fun(*args, **kwds)` raises *warn* and the message matches regex *r*	3.2

Method	**Checks that**	**New in**
`assertLogs(logger, level)`	The `with` block logs on *logger* with minimum *level*	3.4
`assertNoLogs(logger, level)`	The `with` block does not log on *logger* with minimum *level*	3.10

assertRaises(*exception*, *callable*, **args*, ***kwds*)
assertRaises(*exception*, *, *msg=None*)

Test that an exception is raised when *callable* is called with any positional or keyword arguments that are also passed to `assertRaises()`. The test passes if *exception* is raised, is an error if another exception is raised, or fails if no exception is raised. To catch any of a group of exceptions, a tuple containing the exception classes may be passed as *exception*.

If only the *exception* and possibly the *msg* arguments are given, return a context manager so that the code under test can be written inline rather than as a function:

```
with self.assertRaises(SomeException):
    do_something()
```

When used as a context manager, `assertRaises()` accepts the additional keyword argument *msg*.

The context manager will store the caught exception object in its **`exception`** attribute. This can be useful if the intention is to perform additional checks on the exception raised:

```
with self.assertRaises(SomeException) as cm:
    do_something()

the_exception = cm.exception
self.assertEqual(the_exception.error_code, 3)
```

Changed in version 3.1: Added the ability to use `assertRaises()` as a context manager.

Changed in version 3.2: Added the **`exception`** attribute.

Changed in version 3.3: Added the *msg* keyword argument when used as a context manager.

assertRaisesRegex(*exception*, *regex*, *callable*, **args*, ***kwds*)
assertRaisesRegex(*exception*, *regex*, ***, *msg=None*)

Like `assertRaises()` but also tests that *regex* matches on the string representation of the raised exception. *regex* may be a regular expression object or a string containing a regular expression suitable for use by `re.search()`. Examples:

```
self.assertRaisesRegex(ValueError, "invalid literal
for.*XYZ'$",
                       int, 'XYZ')
```

or:

```
with self.assertRaisesRegex(ValueError, 'literal'):
   int('XYZ')
```

Added in version 3.1: Added under the name `assertRaisesRegexp`.

Changed in version 3.2: Renamed to `assertRaisesRegex()`.

Changed in version 3.3: Added the *msg* keyword argument when used as a context manager.

assertWarns(*warning*, *callable*, **args*, ***kwds*)
assertWarns(*warning*, *, *msg=None*)

Test that a warning is triggered when *callable* is called with any positional or keyword arguments that are also passed to `assertWarns()`. The test passes if *warning* is triggered and fails if it isn't. Any exception is an error. To catch any of a group of warnings, a tuple containing the warning classes may be passed as *warnings*.

If only the *warning* and possibly the *msg* arguments are given, return a context manager so that the code under test can be written inline rather than as a function:

```
with self.assertWarns(SomeWarning):
    do_something()
```

When used as a context manager, `assertWarns()` accepts the additional keyword argument *msg*.

The context manager will store the caught warning object in its **`warning`** attribute, and the source line which triggered the warnings in the **`filename`** and **`lineno`** attributes. This can be useful if the intention is to perform additional checks on the warning caught:

```
with self.assertWarns(SomeWarning) as cm:
    do_something()

self.assertIn('myfile.py', cm.filename)
self.assertEqual(320, cm.lineno)
```

This method works regardless of the warning filters in place when it is called.

Added in version 3.2.

Changed in version 3.3: Added the *msg* keyword argument when used as a context manager.

assertWarnsRegex(*warning*, *regex*, *callable*, **args*, ***kwds*)
assertWarnsRegex(*warning*, *regex*, *, *msg=None*)

Like `assertWarns()` but also tests that *regex* matches on the message of the triggered warning. *regex* may be a regular expression object or a string containing a regular expression suitable for use by `re.search()`. Example:

```
self.assertWarnsRegex(DeprecationWarning,
                      r'legacy_function\(\) is
deprecated',
                      legacy_function, 'XYZ')
```

or:

```
with self.assertWarnsRegex(RuntimeWarning, 'unsafe
frobnicating'):
    frobnicate('/etc/passwd')
```

Added in version 3.2.

Changed in version 3.3: Added the *msg* keyword argument when used as a context manager.

assertLogs(*logger=None*, *level=None*)

A context manager to test that at least one message is logged on the *logger* or one of its children, with at least the given *level*.

If given, *logger* should be a `logging.Logger` object or a `str` giving the name of a logger. The default is the root logger, which will catch all messages that were not blocked by a non-propagating descendent logger.

If given, *level* should be either a numeric logging level or its string equivalent (for example either `"ERROR"` or `logging.ERROR`). The default is `logging.INFO`.

The test passes if at least one message emitted inside the `with` block matches the *logger* and *level* conditions, otherwise it fails.

The object returned by the context manager is a recording helper which keeps tracks of the matching log messages. It has two attributes:

records

A list of `logging.LogRecord` objects of the matching log messages.

output

A list of `str` objects with the formatted output of matching messages.

Example:

```
with self.assertLogs('foo', level='INFO') as cm:
    logging.getLogger('foo').info('first message')
    logging.getLogger('foo.bar').error('second message')
self.assertEqual(cm.output, ['INFO:foo:first message',
                             'ERROR:foo.bar:second 
message'])
```

Added in version 3.4.

assertNoLogs(*logger=None*, *level=None*)

A context manager to test that no messages are logged on the *logger* or one of its children, with at least the given *level*.

If given, *logger* should be a `logging.Logger` object or a `str` giving the name of a logger. The default is the root logger, which will catch all messages.

If given, *level* should be either a numeric logging level or its string equivalent (for example either `"ERROR"` or `logging.ERROR`). The default is `logging.INFO`.

Unlike `assertLogs()`, nothing will be returned by the context manager.

Added in version 3.10.

There are also other methods used to perform more specific checks, such as:

Method	**Checks that**	**New in**
`assertAlmostEqual(a, b)`	`round(a-b, 7) == 0`	
`assertNotAlmostEqual(a, b)`	`round(a-b, 7) != 0`	
`assertGreater(a, b)`	`a > b`	3.1
`assertGreaterEqual(a, b)`	`a >= b`	3.1
`assertLess(a, b)`	`a < b`	3.1
`assertLessEqual(a, b)`	`a <= b`	3.1

Method	Checks that	New in
`assertRegex(s, r)`	`r.search(s)`	3.1
`assertNotRegex(s, r)`	`not r.search(s)`	3.2
`assertCountEqual(a, b)`	*a* and *b* have the same elements in the same number, regardless of their order.	3.2
`assertStartsWith(a, b)`	`a.startswith(b)`	3.14
`assertNotStartsWith(a, b)`	`not a.startswith(b)`	3.14
`assertEndsWith(a, b)`	`a.endswith(b)`	3.14
`assertNotEndsWith(a, b)`	`not a.endswith(b)`	3.14
`assertHasAttr(a, b)`	`hastattr(a, b)`	3.14
`assertNotHasAttr(a, b)`	`not hastattr(a, b)`	3.14

assertAlmostEqual(*first*, *second*, *places=7*, *msg=None*, *delta=None*)

assertNotAlmostEqual(*first*, *second*, *places=7*, *msg=None*, *delta=None*)

Test that *first* and *second* are approximately (or not approximately) equal by computing the difference, rounding to the given number of decimal *places* (default 7), and comparing to zero. Note that these methods round the values to the given number of *decimal places* (i.e. like the `round()` function) and not *significant digits*.

If *delta* is supplied instead of *places* then the difference between *first* and *second* must be less or equal to (or greater than) *delta*.

Supplying both *delta* and *places* raises a `TypeError`.

Changed in version 3.2: `assertAlmostEqual()` automatically considers almost equal objects that compare equal. `assertNotAlmostEqual()` automatically fails if the objects compare equal. Added the *delta* keyword argument.

assertGreater(*first*, *second*, *msg=None*)
assertGreaterEqual(*first*, *second*, *msg=None*)
assertLess(*first*, *second*, *msg=None*)
assertLessEqual(*first*, *second*, *msg=None*)

Test that *first* is respectively >, >=, < or <= than *second* depending on the method name. If not, the test will fail:

```
>>> self.assertGreaterEqual(3, 4)
AssertionError: "3" unexpectedly not greater than or
equal to "4"
```

Added in version 3.1.

assertRegex(*text*, *regex*, *msg=None*)
assertNotRegex(*text*, *regex*, *msg=None*)

Test that a *regex* search matches (or does not match) *text*. In case of failure, the error message will include the pattern and the *text* (or the pattern and the part of *text* that unexpectedly matched). *regex* may be a regular expression object or a string containing a regular expression suitable for use by `re.search()`.

Added in version 3.1: Added under the name `assertRegexpMatches`.

Changed in version 3.2: The method `assertRegexpMatches()` has been renamed to **`assertRegex()`**.

Added in version 3.2: **`assertNotRegex()`**.

assertCountEqual(*first*, *second*, *msg=None*)

Test that sequence *first* contains the same elements as *second*, regardless of their order. When they don't, an error message listing the differences between the sequences will be generated.

Duplicate elements are *not* ignored when comparing *first* and *second*. It verifies whether each element has the same count in both sequences. Equivalent to: `assertEqual(Counter(list(first)), Counter(list(second)))` but works with sequences of unhashable objects as well.

Added in version 3.2.

assertStartsWith(*s*, *prefix*, *msg=None*)

assertNotStartsWith(*s*, *prefix*, *msg=None*)

Test that the Unicode or byte string *s* starts (or does not start) with a *prefix*. *prefix* can also be a tuple of strings to try.

Added in version 3.14.

assertEndsWith(*s*, *suffix*, *msg=None*)

assertNotEndsWith(*s*, *suffix*, *msg=None*)

Test that the Unicode or byte string *s* ends (or does not end) with a *suffix*. *suffix* can also be a tuple of strings to try.

Added in version 3.14.

assertHasAttr(*obj*, *name*, *msg=None*)

assertNotHasAttr(*obj*, *name*, *msg=None*)

Test that the object *obj* has (or has not) an attribute *name*.

Added in version 3.14.

The `assertEqual()` method dispatches the equality check for objects of the same type to different type-specific methods. These methods are already implemented for most of the built-in types, but it's also possible to register new methods using `addTypeEqualityFunc()`:

addTypeEqualityFunc(*typeobj*, *function*)

Registers a type-specific method called by `assertEqual()` to check if two objects of exactly the same *typeobj* (not subclasses) compare equal. *function* must take two positional arguments and a third msg=None keyword argument just as `assertEqual()` does. It must raise `self.failureException(msg)` when inequality between the first two parameters is detected – possibly providing useful information and explaining the inequalities in details in the error message.

Added in version 3.1.

The list of type-specific methods automatically used by `assertEqual()` are summarized in the following table. Note that it’s usually not necessary to invoke these methods directly.

Method	Used to compare	New in
`assertMultiLineEqual(a, b)`	strings	3.1
`assertSequenceEqual(a, b)`	sequences	3.1
`assertListEqual(a, b)`	lists	3.1
`assertTupleEqual(a, b)`	tuples	3.1
`assertSetEqual(a, b)`	sets or frozensets	3.1
`assertDictEqual(a, b)`	dicts	3.1

assertMultiLineEqual(*first*, *second*, *msg=None*)

Test that the multiline string *first* is equal to the string *second*. When not equal a diff of the two strings highlighting the differences will be included in the error message. This method is used by default when comparing strings with `assertEqual()`.

Added in version 3.1.

assertSequenceEqual(*first*, *second*, *msg=None*, *seq_type=None*)

Tests that two sequences are equal. If a *seq_type* is supplied, both *first* and *second* must be instances of *seq_type* or a failure will be raised. If the sequences are different an error message is constructed that shows the difference between the two.

This method is not called directly by `assertEqual()`, but it's used to implement `assertListEqual()` and `assertTupleEqual()`.

Added in version 3.1.

assertListEqual(*first*, *second*, *msg=None*)
assertTupleEqual(*first*, *second*, *msg=None*)

Tests that two lists or tuples are equal. If not, an error message is constructed that shows only the differences between the two. An error is also raised if either of the parameters are of the wrong type. These methods are used by default when comparing lists or tuples with `assertEqual()`.

Added in version 3.1.

assertSetEqual(*first*, *second*, *msg=None*)

Tests that two sets are equal. If not, an error message is constructed that lists the differences between the sets. This method is used by default when comparing sets or frozensets with `assertEqual()`.

Fails if either of *first* or *second* does not have a `difference()` method.

Added in version 3.1.

assertDictEqual(*first*, *second*, *msg=None*)

Test that two dictionaries are equal. If not, an error message is constructed that shows the differences in the dictionaries. This method will be used by default to compare dictionaries in calls to `assertEqual()`.

Added in version 3.1.

Finally the `TestCase` provides the following methods and attributes:

fail(*msg=None*)

Signals a test failure unconditionally, with *msg* or `None` for the error message.

failureException

This class attribute gives the exception raised by the test method. If a test framework needs to use a specialized exception, possibly to carry additional information, it must subclass this exception in order to "play fair" with the framework. The initial value of this attribute is `AssertionError`.

longMessage

This class attribute determines what happens when a custom failure message is passed as the msg argument to an assertXYY call that fails. `True` is the default value. In this case, the custom message is appended to the end of the standard failure message. When set to `False`, the custom message replaces the standard message.

The class setting can be overridden in individual test methods by assigning an instance attribute, self.longMessage, to `True` or `False` before calling the assert methods.

The class setting gets reset before each test call.

Added in version 3.1.

maxDiff

This attribute controls the maximum length of diffs output by assert methods that report diffs on failure. It defaults to 80*8 characters. Assert methods affected by this attribute are `assertSequenceEqual()` (including all the sequence comparison methods that delegate to it), `assertDictEqual()` and `assertMultiLineEqual()`.

Setting `maxDiff` to `None` means that there is no maximum length of diffs.

Added in version 3.2.

Testing frameworks can use the following methods to collect information on the test:

countTestCases()

Return the number of tests represented by this test object. For `TestCase` instances, this will always be `1`.

defaultTestResult()

Return an instance of the test result class that should be used for this test case class (if no other result instance is provided to the `run()` method).

For `TestCase` instances, this will always be an instance of `TestResult`; subclasses of `TestCase` should override this as necessary.

id()

Return a string identifying the specific test case. This is usually the full name of the test method, including the module and class name.

shortDescription()

Returns a description of the test, or `None` if no description has been provided. The default implementation of this method returns the first line of the test method’s docstring, if available, or `None`.

Changed in version 3.1: In 3.1 this was changed to add the test name to the short description even in the presence of a docstring. This caused compatibility issues with unittest extensions and adding the test name was moved to the `TextTestResult` in Python 3.2.

addCleanup(*function*, /, **args*, ***kwargs*)

Add a function to be called after `tearDown()` to cleanup resources used during the test. Functions will be called in reverse order to the order they are added (LIFO). They are called with any arguments and keyword arguments passed into `addCleanup()` when they are added.

If `setUp()` fails, meaning that `tearDown()` is not called, then any cleanup functions added will still be called.

Added in version 3.1.

enterContext(*cm*)

Enter the supplied context manager. If successful, also add its `__exit__()` method as a cleanup function by `addCleanup()` and return the result of the `__enter__()` method.

Added in version 3.11.

doCleanups()

This method is called unconditionally after `tearDown()`, or after `setUp()` if `setUp()` raises an exception.

It is responsible for calling all the cleanup functions added by `addCleanup()`. If you need cleanup functions to be called *prior* to `tearDown()` then you can call `doCleanups()` yourself.

`doCleanups()` pops methods off the stack of cleanup functions one at a time, so it can be called at any time.

Added in version 3.1.

classmethod addClassCleanup(*function*, /, **args*, ***kwargs*)

Add a function to be called after `tearDownClass()` to cleanup resources used during the test class. Functions will be called in reverse order to the order they are added (LIFO). They are called with any arguments and keyword arguments passed into `addClassCleanup()` when they are added.

If `setUpClass()` fails, meaning that `tearDownClass()` is not called, then any cleanup functions added will still be called.

Added in version 3.8.

classmethod enterClassContext(*cm*)

Enter the supplied context manager. If successful, also add its `__exit__()` method as a cleanup function by `addClassCleanup()` and return the result of the `__enter__()` method.

Added in version 3.11.

classmethod doClassCleanups()

This method is called unconditionally after `tearDownClass()`, or after `setUpClass()` if `setUpClass()` raises an exception.

It is responsible for calling all the cleanup functions added by `addClassCleanup()`. If you need cleanup functions to be called *prior* to `tearDownClass()` then you can call `doClassCleanups()` yourself.

`doClassCleanups()` pops methods off the stack of cleanup functions one at a time, so it can be called at any time.

Added in version 3.8.

class unittest.IsolatedAsyncioTestCase(*methodName*='*runTest*')

This class provides an API similar to `TestCase` and also accepts coroutines as test functions.

Added in version 3.8.

loop_factory

The *loop_factory* passed to `asyncio.Runner`. Override in subclasses with `asyncio.EventLoop` to avoid using the asyncio policy system.

Added in version 3.13.

async asyncSetUp()

Method called to prepare the test fixture. This is called after `TestCase.setUp()`. This is called immediately before calling the test method; other than `AssertionError` or `SkipTest`, any exception raised by this method will be considered an error rather than a test failure. The default implementation does nothing.

async asyncTearDown()

Method called immediately after the test method has been called and the result recorded. This is called before `tearDown()`. This is called even if the test method raised an exception, so the implementation in subclasses may need to be particularly careful about checking internal state. Any exception, other than `AssertionError` or `SkipTest`, raised by this method will be considered an additional error rather than a test failure (thus increasing the total number of reported errors). This method will only be called if the `asyncSetUp()` succeeds, regardless of the outcome of the test method. The default implementation does nothing.

addAsyncCleanup(*function*, */*, **args*, ***kwargs*)

This method accepts a coroutine that can be used as a cleanup function.

async enterAsyncContext(*cm*)

Enter the supplied asynchronous context manager. If successful, also add its `__aexit__()` method as a cleanup function by `addAsyncCleanup()` and return the result of the `__aenter__()` method.

Added in version 3.11.

run(*result=None*)

Sets up a new event loop to run the test, collecting the result into the `TestResult` object passed as *result*. If *result* is omitted or `None`, a temporary result object is created (by calling the `defaultTestResult()` method) and used. The result object is returned to `run()`'s caller. At the end of the test all the tasks in the event loop are cancelled.

An example illustrating the order:

```
from unittest import IsolatedAsyncioTestCase

events = []
```

```
class Test(IsolatedAsyncioTestCase):


    def setUp(self):
        events.append("setUp")

    async def asyncSetUp(self):
        self._async_connection = await AsyncConnection()
        events.append("asyncSetUp")

    async def test_response(self):
        events.append("test_response")
        response = await
self._async_connection.get("https://example.com")
        self.assertEqual(response.status_code, 200)
        self.addAsyncCleanup(self.on_cleanup)

    def tearDown(self):
        events.append("tearDown")

    async def asyncTearDown(self):
        await self._async_connection.close()
        events.append("asyncTearDown")

    async def on_cleanup(self):
        events.append("cleanup")

if __name__ == "__main__":
    unittest.main()
```

After running the test, `events` would contain `["setUp", "asyncSetUp", "test_response", "asyncTearDown", "tearDown", "cleanup"]`.

class unittest.FunctionTestCase(*testFunc*, *setUp=None*, *tearDown=None*, *description=None*)

This class implements the portion of the [`TestCase`](#) interface which allows the test runner to drive the test, but does not provide the methods which test code can use to check and report errors. This is used to create test cases using legacy test code, allowing it to be integrated into a `unittest`-based test framework.

Grouping tests

class unittest.TestSuite(*tests=()*)

This class represents an aggregation of individual test cases and test suites. The class presents the interface needed by the test runner to allow it to be run as any other test case. Running a `TestSuite` instance is the same as iterating over the suite, running each test individually.

If *tests* is given, it must be an iterable of individual test cases or other test suites that will be used to build the suite initially. Additional methods are provided to add test cases and suites to the collection later on.

`TestSuite` objects behave much like `TestCase` objects, except they do not actually implement a test. Instead, they are used to aggregate tests into groups of tests that should be run together. Some additional methods are available to add tests to `TestSuite` instances:

addTest(*test*)

Add a `TestCase` or `TestSuite` to the suite.

addTests(*tests*)

Add all the tests from an iterable of `TestCase` and `TestSuite` instances to this test suite.

This is equivalent to iterating over *tests*, calling `addTest()` for each element.

`TestSuite` shares the following methods with `TestCase`:

run(*result*)

Run the tests associated with this suite, collecting the result into the test result object passed as *result*. Note that unlike `TestCase.run()`, `TestSuite.run()` requires the result object to be passed in.

debug()

Run the tests associated with this suite without collecting the result. This allows exceptions raised by the test to be propagated to the caller and can be used to support running tests under a debugger.

countTestCases()

Return the number of tests represented by this test object, including all individual tests and sub-suites.

__iter__()

Tests grouped by a `TestSuite` are always accessed by iteration. Subclasses can lazily provide tests by overriding `__iter__()`. Note that this method may be called several times on a single suite (for example when counting tests or comparing for equality) so the tests returned by repeated iterations before `TestSuite.run()` must be the same for each call iteration. After `TestSuite.run()`, callers should not rely on the tests returned by this method unless the caller uses a subclass that overrides `TestSuite._removeTestAtIndex()` to preserve test references.

Changed in version 3.2: In earlier versions the `TestSuite` accessed tests directly rather than through iteration, so overriding `__iter__()` wasn't sufficient for providing tests.

Changed in version 3.4: In earlier versions the `TestSuite` held references to each `TestCase` after

`TestSuite.run()`. Subclasses can restore that behavior by overriding **`TestSuite._removeTestAtIndex()`**.

In the typical usage of a `TestSuite` object, the `run()` method is invoked by a **`TestRunner`** rather than by the end-user test harness.

Loading and running tests

class unittest.TestLoader

The `TestLoader` class is used to create test suites from classes and modules. Normally, there is no need to create an instance of this class; the `unittest` module provides an instance that can be shared as `unittest.defaultTestLoader`. Using a subclass or instance, however, allows customization of some configurable properties.

`TestLoader` objects have the following attributes:

errors

A list of the non-fatal errors encountered while loading tests. Not reset by the loader at any point. Fatal errors are signalled by the relevant method raising an exception to the caller. Non-fatal errors are also indicated by a synthetic test that will raise the original error when run.

Added in version 3.5.

`TestLoader` objects have the following methods:

loadTestsFromTestCase(*testCaseClass*)

Return a suite of all test cases contained in the `TestCase`-derived **`testCaseClass`**.

A test case instance is created for each method named by `getTestCaseNames()`. By default these are the method names beginning with `test`. If `getTestCaseNames()` returns no methods, but the **`runTest()`** method is implemented, a single test case is created for that method instead.

loadTestsFromModule(*module*, *, *pattern=None*)

Return a suite of all test cases contained in the given module. This method searches *module* for classes derived from `TestCase` and creates an instance of the class for each test method defined for the class.

Note

While using a hierarchy of `TestCase`-derived classes can be convenient in sharing fixtures and helper functions, defining test methods on base classes that are not intended to be instantiated directly does not play well with this method. Doing so, however, can be useful when the fixtures are different and defined in subclasses.

If a module provides a `load_tests` function it will be called to load the tests. This allows modules to customize test loading. This is the load_tests protocol. The *pattern* argument is passed as the third argument to `load_tests`.

Changed in version 3.2: Support for `load_tests` added.

Changed in version 3.5: Support for a keyword-only argument *pattern* has been added.

Changed in version 3.12: The undocumented and unofficial *use_load_tests* parameter has been removed.

loadTestsFromName(*name*, *module=None*)

Return a suite of all test cases given a string specifier.

The specifier *name* is a “dotted name” that may resolve either to a module, a test case class, a test method within a test case class, a `TestSuite` instance, or a callable object which returns a `TestCase` or `TestSuite` instance. These checks are applied in the order listed here; that is, a method on a possible test case class will be picked up as “a test method within a test case class”, rather than “a callable object”.

For example, if you have a module **`SampleTests`** containing a `TestCase`-derived class **`SampleTestCase`** with three test methods (**`test_one()`**, **`test_two()`**, and **`test_three()`**), the specifier `'SampleTests.SampleTestCase'` would cause this method to return a suite which will run all three test methods. Using the specifier `'SampleTests.SampleTestCase.test_two'` would cause it to return a test suite which will run only the **`test_two()`** test method. The specifier can refer to modules and packages which have not been imported; they will be imported as a side-effect.

The method optionally resolves *name* relative to the given *module*.

Changed in version 3.5: If an `ImportError` or `AttributeError` occurs while traversing *name* then a synthetic test that raises that error when run will be returned. These errors are included in the errors accumulated by self.errors.

loadTestsFromNames(*names*, *module=None*)

Similar to `loadTestsFromName()`, but takes a sequence of names rather than a single name. The return value is a test suite which supports all the tests defined for each name.

getTestCaseNames(*testCaseClass*)

Return a sorted sequence of method names found within *testCaseClass*; this should be a subclass of `TestCase`.

discover(*start_dir*, *pattern='test*.py'*, *top_level_dir=None*)

Find all the test modules by recursing into subdirectories from the specified start directory, and return a TestSuite object containing them. Only test files that match *pattern* will be loaded. (Using shell style pattern matching.) Only module names that are importable (i.e. are valid Python identifiers) will be loaded.

All test modules must be importable from the top level of the project. If the start directory is not the top level directory then *top_level_dir* must be specified separately.

If importing a module fails, for example due to a syntax error, then this will be recorded as a single error and discovery will continue. If the import failure is due to `SkipTest` being raised, it will be recorded as a skip instead of an error.

If a package (a directory containing a file named `__init__.py`) is found, the package will be checked for a `load_tests` function. If this exists then it will be called `package.load_tests(loader, tests, pattern)`. Test discovery takes care to ensure that a package is only checked for tests once during an invocation, even if the load_tests function itself calls `loader.discover`.

If `load_tests` exists then discovery does *not* recurse into the package, `load_tests` is responsible for loading all tests in the package.

The pattern is deliberately not stored as a loader attribute so that packages can continue discovery themselves.

top_level_dir is stored internally, and used as a default to any nested calls to `discover()`. That is, if a package's `load_tests` calls `loader.discover()`, it does not need to pass this argument.

start_dir can be a dotted module name as well as a directory.

Added in version 3.2.

Changed in version 3.4: Modules that raise **`SkipTest`** on import are recorded as skips, not errors.

start_dir can be a namespace packages.

Paths are sorted before being imported so that execution order is the same even if the underlying file system's ordering is not dependent on file name.

Changed in version 3.5: Found packages are now checked for `load_tests` regardless of whether their path matches *pattern*, because it is impossible for a package name to match the default pattern.

Changed in version 3.11: start_dir can not be a namespace packages. It has been broken since Python

3.7, and Python 3.11 officially removes it.

Changed in version 3.13: top_level_dir is only stored for the duration of *discover* call.

Changed in version 3.14: start_dir can once again be a namespace package.

The following attributes of a **`TestLoader`** can be configured either by subclassing or assignment on an instance:

testMethodPrefix

String giving the prefix of method names which will be interpreted as test methods. The default value is `'test'`.

This affects **`getTestCaseNames()`** and all the `loadTestsFrom*` methods.

sortTestMethodsUsing

Function to be used to compare method names when sorting them in **`getTestCaseNames()`** and all the `loadTestsFrom*` methods.

suiteClass

Callable object that constructs a test suite from a list of tests. No methods on the resulting object are needed. The default value is the **`TestSuite`** class.

This affects all the `loadTestsFrom*` methods.

testNamePatterns

List of Unix shell-style wildcard test name patterns that test methods have to match to be included in test suites (see `-k` option).

If this attribute is not `None` (the default), all test methods to be included in test suites must match one of the patterns in this list. Note that matches are always performed using [`fnmatch.fnmatchcase()`], so unlike patterns passed to the `-k` option, simple substring patterns will have to be converted using `*` wildcards.

This affects all the `loadTestsFrom*` methods.

Added in version 3.7.

class unittest.TestResult

This class is used to compile information about which tests have succeeded and which have failed.

A **`TestResult`** object stores the results of a set of tests. The **`TestCase`** and **`TestSuite`** classes ensure that results are properly recorded; test authors do not need to worry about recording the outcome of tests.

Testing frameworks built on top of **`unittest`** may want access to the **`TestResult`** object generated by running a set of tests for reporting purposes; a **`TestResult`** instance is returned by the **`TestRunner.run()`** method for this purpose.

`TestResult` instances have the following attributes that will be of interest when inspecting the results of running a set of tests:

errors

A list containing 2-tuples of **`TestCase`** instances and strings holding formatted tracebacks. Each tuple represents a test which raised an unexpected exception.

failures

A list containing 2-tuples of **`TestCase`** instances and strings holding formatted tracebacks. Each tuple represents a test where a failure was explicitly signalled using the assert* methods.

skipped

A list containing 2-tuples of **`TestCase`** instances and strings holding the reason for skipping the test.

Added in version 3.1.

expectedFailures

A list containing 2-tuples of **`TestCase`** instances and strings holding formatted tracebacks. Each tuple represents an expected failure or error of the test case.

unexpectedSuccesses

A list containing **`TestCase`** instances that were marked as expected failures, but succeeded.

collectedDurations

A list containing 2-tuples of test case names and floats representing the elapsed time of each test which was run.

Added in version 3.12.

shouldStop

Set to `True` when the execution of tests should stop by **`stop()`**.

testsRun

The total number of tests run so far.

buffer

If set to true, `sys.stdout` and `sys.stderr` will be buffered in between `startTest()` and `stopTest()` being called. Collected output will only be echoed onto the real `sys.stdout` and `sys.stderr` if the test fails or errors. Any output is also attached to the failure / error message.

Added in version 3.2.

failfast

If set to true `stop()` will be called on the first failure or error, halting the test run.

Added in version 3.2.

tb_locals

If set to true then local variables will be shown in tracebacks.

Added in version 3.5.

wasSuccessful()

Return `True` if all tests run so far have passed, otherwise returns `False`.

Changed in version 3.4: Returns `False` if there were any `unexpectedSuccesses` from tests marked with the `expectedFailure()` decorator.

stop()

This method can be called to signal that the set of tests being run should be aborted by setting the `shouldStop` attribute to `True`. **`TestRunner`** objects should respect this flag and return without running any additional tests.

For example, this feature is used by the `TextTestRunner` class to stop the test framework when the user signals an interrupt from the keyboard. Interactive tools which provide **`TestRunner`** implementations can use this in a similar manner.

The following methods of the `TestResult` class are used to maintain the internal data structures, and may be extended in subclasses to support additional reporting requirements. This is particularly useful in building tools which support interactive reporting while tests are being run.

startTest(*test*)

Called when the test case *test* is about to be run.

stopTest(*test*)

Called after the test case *test* has been executed, regardless of the outcome.

startTestRun()

Called once before any tests are executed.

Added in version 3.1.

stopTestRun()

Called once after all tests are executed.

Added in version 3.1.

addError(*test*, *err*)

Called when the test case *test* raises an unexpected exception. *err* is a tuple of the form returned by `sys.exc_info()`: `(type, value, traceback)`.

The default implementation appends a tuple `(test, formatted_err)` to the instance’s **`errors`** attribute, where *formatted_err* is a formatted traceback derived from *err*.

addFailure(*test*, *err*)

Called when the test case *test* signals a failure. *err* is a tuple of the form returned by **`sys.exc_info()`**: `(type, value, traceback)`.

The default implementation appends a tuple `(test, formatted_err)` to the instance’s **`failures`** attribute, where *formatted_err* is a formatted traceback derived from *err*.

addSuccess(*test*)

Called when the test case *test* succeeds.

The default implementation does nothing.

addSkip(*test*, *reason*)

Called when the test case *test* is skipped. *reason* is the reason the test gave for skipping.

The default implementation appends a tuple `(test, reason)` to the instance’s **`skipped`** attribute.

addExpectedFailure(*test*, *err*)

Called when the test case *test* fails or errors, but was marked with the **`expectedFailure()`** decorator.

The default implementation appends a tuple `(test, formatted_err)` to the instance’s **`expectedFailures`** attribute, where *formatted_err* is a formatted traceback derived from *err*.

addUnexpectedSuccess(*test*)

Called when the test case *test* was marked with the `expectedFailure()` decorator, but succeeded.

The default implementation appends the test to the instance's `unexpectedSuccesses` attribute.

addSubTest(*test*, *subtest*, *outcome*)

Called when a subtest finishes. *test* is the test case corresponding to the test method. *subtest* is a custom `TestCase` instance describing the subtest.

If *outcome* is `None`, the subtest succeeded. Otherwise, it failed with an exception where *outcome* is a tuple of the form returned by `sys.exc_info()`: `(type, value, traceback)`.

The default implementation does nothing when the outcome is a success, and records subtest failures as normal failures.

Added in version 3.4.

addDuration(*test*, *elapsed*)

Called when the test case finishes. *elapsed* is the time represented in seconds, and it includes the execution of cleanup functions.

Added in version 3.12.

class unittest.TextTestResult(*stream*, *descriptions*, *verbosity*, ***, *durations=None*)

A concrete implementation of `TestResult` used by the `TextTestRunner`. Subclasses should accept `**kwargs` to ensure compatibility as the interface changes.

Added in version 3.2.

Changed in version 3.12: Added the *durations* keyword parameter.

unittest.defaultTestLoader

Instance of the **`TestLoader`** class intended to be shared. If no customization of the **`TestLoader`** is needed, this instance can be used instead of repeatedly creating new instances.

class unittest.TextTestRunner(*stream=None*, *descriptions=True*, *verbosity=1*, *failfast=False*, *buffer=False*, *resultclass=None*, *warnings=None*, *, *tb_locals=False*, *durations=None*)

A basic test runner implementation that outputs results to a stream. If *stream* is `None`, the default, **`sys.stderr`** is used as the output stream. This class has a few configurable parameters, but is essentially very simple. Graphical applications which run test suites should provide alternate implementations. Such implementations should accept `**kwargs` as the interface to construct runners changes when features are added to unittest.

By default this runner shows **`DeprecationWarning`**, **`PendingDeprecationWarning`**, **`ResourceWarning`** and **`ImportWarning`** even if they are ignored by default. This behavior can be overridden using Python's **`-Wd`** or **`-Wa`** options (see Warning control) and leaving *warnings* to `None`.

Changed in version 3.2: Added the *warnings* parameter.

Changed in version 3.2: The default stream is set to **`sys.stderr`** at instantiation time rather than import time.

Changed in version 3.5: Added the *tb_locals* parameter.

Changed in version 3.12: Added the *durations* parameter.

_makeResult()

This method returns the instance of `TestResult` used by **`run()`**. It is not intended to be called directly, but can be overridden in subclasses to provide a custom `TestResult`.

`_makeResult()` instantiates the class or callable passed in the `TextTestRunner` constructor as the `resultclass` argument. It defaults to **`TextTestResult`** if no `resultclass` is provided. The result class is instantiated with the following arguments:

```
stream, descriptions, verbosity
```

run(*test*)

This method is the main public interface to the `TextTestRunner`. This method takes a **`TestSuite`** or **`TestCase`** instance. A **`TestResult`** is created by calling **`_makeResult()`** and the test(s) are run and the results printed to stdout.

unittest.main(*module='__main__'*, *defaultTest=None*, *argv=None*, *testRunner=None*, *testLoader=unittest.defaultTestLoader*, *exit=True*, *verbosity=1*, *failfast=None*, *catchbreak=None*, *buffer=None*, *warnings=None*)

A command-line program that loads a set of tests from *module* and runs them; this is primarily for making test modules conveniently executable. The simplest use for this function is to include the following line at the end of a test script:

```
if __name__ == '__main__':
    unittest.main()
```

You can run tests with more detailed information by passing in the verbosity argument:

```
if __name__ == '__main__':
    unittest.main(verbosity=2)
```

The *defaultTest* argument is either the name of a single test or an iterable of test names to run if no test names are specified via *argv*. If not specified or `None` and no test names are provided via *argv*, all tests found in *module* are run.

The *argv* argument can be a list of options passed to the program, with the first element being the program name. If not specified or `None`, the values of **`sys.argv`** are used.

The *testRunner* argument can either be a test runner class or an already created instance of it. By default `main` calls **`sys.exit()`** with an exit code indicating success (0) or failure (1) of the tests run. An exit code of 5 indicates that no tests were run or skipped.

The *testLoader* argument has to be a **`TestLoader`** instance, and defaults to **`defaultTestLoader`**.

`main` supports being used from the interactive interpreter by passing in the argument `exit=False`. This displays the result on standard output without calling **`sys.exit()`**:

```
>>> from unittest import main
>>> main(module='test_module', exit=False)
```

The *failfast*, *catchbreak* and *buffer* parameters have the same effect as the same-name command-line options.

The *warnings* argument specifies the warning filter that should be used while running the tests. If it’s not specified, it will remain `None` if a **`-W`** option is passed to **python** (see Warning control), otherwise it will be set to `'default'`.

Calling `main` returns an object with the `result` attribute that contains the result of the tests run as a **`unittest.TestResult`**.

Changed in version 3.1: The *exit* parameter was added.

Changed in version 3.2: The *verbosity*, *failfast*, *catchbreak*, *buffer* and *warnings* parameters were added.

Changed in version 3.4: The *defaultTest* parameter was changed to also accept an iterable of test names.

load_tests Protocol

Added in version 3.2.

Modules or packages can customize how tests are loaded from them during normal test runs or test discovery by implementing a function called `load_tests`.

If a test module defines `load_tests` it will be called by **`TestLoader.loadTestsFromModule()`** with the following arguments:

```
load_tests(loader, standard_tests, pattern)
```

where *pattern* is passed straight through from `loadTestsFromModule`. It defaults to `None`.

It should return a **`TestSuite`**.

loader is the instance of **`TestLoader`** doing the loading. *standard_tests* are the tests that would be loaded by default from the module. It is common for test modules to only want to add or remove tests from the standard set of tests. The third argument is used when loading packages as part of test discovery.

A typical `load_tests` function that loads tests from a specific set of **`TestCase`** classes may look like:

```
test_cases = (TestCase1, TestCase2, TestCase3)

def load_tests(loader, tests, pattern):
    suite = TestSuite()
    for test_class in test_cases:
        tests = loader.loadTestsFromTestCase(test_class)
        suite.addTests(tests)
    return suite
```

If discovery is started in a directory containing a package, either from the command line or by calling **`TestLoader.discover()`**, then the package `__init__.py` will be checked for `load_tests`. If that function does not exist, discovery will recurse into the package as though it were just another directory. Otherwise, discovery of the package’s tests will be left up to `load_tests` which is called with the following arguments:

```
load_tests(loader, standard_tests, pattern)
```

This should return a **`TestSuite`** representing all the tests from the package. (`standard_tests` will only contain tests collected from `__init__.py`.)

Because the pattern is passed into `load_tests` the package is free to continue (and potentially modify) test discovery. A ‘do nothing’ `load_tests` function for a test package would look like:

```
def load_tests(loader, standard_tests, pattern):
    # top level directory cached on loader instance
    this_dir = os.path.dirname(__file__)
    package_tests = loader.discover(start_dir=this_dir,
pattern=pattern)
    standard_tests.addTests(package_tests)
    return standard_tests
```

Changed in version 3.5: Discovery no longer checks package names for matching *pattern* due to the impossibility of package names matching the default pattern.

Class and Module Fixtures

Class and module level fixtures are implemented in **`TestSuite`**. When the test suite encounters a test from a new class then **`tearDownClass()`** from the previous class (if there is one) is called, followed by **`setUpClass()`** from the new class.

Similarly if a test is from a different module from the previous test then `tearDownModule` from the previous module is run, followed by `setUpModule` from the new module.

After all the tests have run the final `tearDownClass` and `tearDownModule` are run.

Note that shared fixtures do not play well with [potential] features like test parallelization and they break test isolation. They should be used with care.

The default ordering of tests created by the unittest test loaders is to group all tests from the same modules and classes together. This will lead to `setUpClass` / `setUpModule` (etc) being called exactly once per class and module. If you randomize the order, so that tests from different modules and classes are adjacent to each other, then these shared fixture functions may be called multiple times in a single test run.

Shared fixtures are not intended to work with suites with non-standard ordering. A `BaseTestSuite` still exists for frameworks that don't want to support shared fixtures.

If there are any exceptions raised during one of the shared fixture functions the test is reported as an error. Because there is no corresponding test instance an `_ErrorHolder` object (that has the same interface as a **`TestCase`**) is created to represent the error. If you are just using the standard unittest test runner then this detail doesn't matter, but if you are a framework author it may be relevant.

setUpClass and tearDownClass

These must be implemented as class methods:

```
import unittest

class Test(unittest.TestCase):
    @classmethod
    def setUpClass(cls):
        cls._connection = createExpensiveConnectionObject()

    @classmethod
    def tearDownClass(cls):
        cls._connection.destroy()
```

If you want the `setUpClass` and `tearDownClass` on base classes called then you must call up to them yourself. The implementations in **`TestCase`** are empty.

If an exception is raised during a `setUpClass` then the tests in the class are not run and the `tearDownClass` is not run. Skipped classes will not have `setUpClass` or `tearDownClass` run.

If the exception is a **`SkipTest`** exception then the class will be reported as having been skipped instead of as an error.

setUpModule and tearDownModule

These should be implemented as functions:

```
def setUpModule():
    createConnection()

def tearDownModule():
    closeConnection()
```

If an exception is raised in a `setUpModule` then none of the tests in the module will be run and the `tearDownModule` will not be run. If the exception is a **`SkipTest`** exception then the module will be reported as having been skipped instead of as an error.

To add cleanup code that must be run even in the case of an exception, use `addModuleCleanup`:

unittest.addModuleCleanup(*function*, */*, **args*, ***kwargs*)

Add a function to be called after **`tearDownModule()`** to cleanup resources used during the test class. Functions will be called in reverse order to the order they are added (LIFO). They are called with any arguments and keyword arguments passed into **`addModuleCleanup()`** when they are added.

If **`setUpModule()`** fails, meaning that **`tearDownModule()`** is not called, then any cleanup functions added will still be called.

Added in version 3.8.

unittest.enterModuleContext(*cm*)

Enter the supplied context manager. If successful, also add its `__exit__()` method as a cleanup function by `addModuleCleanup()` and return the result of the `__enter__()` method.

Added in version 3.11.

unittest.doModuleCleanups()

This function is called unconditionally after `tearDownModule()`, or after `setUpModule()` if `setUpModule()` raises an exception.

It is responsible for calling all the cleanup functions added by `addModuleCleanup()`. If you need cleanup functions to be called *prior* to `tearDownModule()` then you can call `doModuleCleanups()` yourself.

`doModuleCleanups()` pops methods off the stack of cleanup functions one at a time, so it can be called at any time.

Added in version 3.8.

Signal Handling

Added in version 3.2.

The `-c/--catch` command-line option to unittest, along with the `catchbreak` parameter to `unittest.main()`, provide more friendly handling of control-C during a test run. With catch break behavior enabled control-C will allow the currently running test to complete, and the test run will then end and report all the results so far. A second control-c will raise a `KeyboardInterrupt` in the usual way.

The control-c handling signal handler attempts to remain compatible with code or tests that install their own `signal.SIGINT` handler. If the `unittest` handler is called but *isn’t* the installed `signal.SIGINT` handler, i.e. it has been replaced by the system under test and delegated to, then it calls the default handler. This will normally be the expected behavior by code that replaces an installed handler and delegates to it. For individual tests that need `unittest` control-c handling disabled the `removeHandler()` decorator can be used.

There are a few utility functions for framework authors to enable control-c handling functionality within test frameworks.

unittest.installHandler()

Install the control-c handler. When a `signal.SIGINT` is received (usually in response to the user pressing control-c) all registered results have `stop()` called.

unittest.registerResult(*result*)

Register a `TestResult` object for control-c handling. Registering a result stores a weak reference to it, so it doesn’t prevent the result from being garbage collected.

Registering a `TestResult` object has no side-effects if control-c handling is not enabled, so test frameworks can unconditionally register all results they create independently of whether or not handling is enabled.

unittest.removeResult(*result*)

Remove a registered result. Once a result has been removed then `stop()` will no longer be called on that result object in response to a control-c.

unittest.removeHandler(*function=None*)

When called without arguments this function removes the control-c handler if it has been installed. This function can also be used as a test decorator to temporarily remove the handler while the test is being executed:

```
@unittest.removeHandler
def test_signal_handling(self):
    ...
```

`unittest.mock` — mock object library

Added in version 3.3.

Source code: Lib/unittest/mock.py [https://github.com/python/cpython/tree/3.14/Lib/unittest/mock.py]

`unittest.mock` is a library for testing in Python. It allows you to replace parts of your system under test with mock objects and make assertions about how they have been used.

`unittest.mock` provides a core `Mock` class removing the need to create a host of stubs throughout your test suite. After performing an action, you can make assertions about which methods / attributes were used and arguments they were called with. You can also specify return values and set needed attributes in the normal way.

Additionally, mock provides a `patch()` decorator that handles patching module and class level attributes within the scope of a test, along with `sentinel` for creating unique objects. See the quick guide for some examples of how to use `Mock`, `MagicMock` and `patch()`.

Mock is designed for use with `unittest` and is based on the 'action -> assertion' pattern instead of 'record -> replay' used by many mocking frameworks.

There is a backport of `unittest.mock` for earlier versions of Python, available as mock [https://pypi.org/project/mock/] on PyPI.

Quick Guide

`Mock` and `MagicMock` objects create all attributes and methods as you access them and store details of how they have been used. You can configure them, to specify return values or limit what attributes are available, and then make assertions about how they have been used:

```
>>> from unittest.mock import MagicMock
>>> thing = ProductionClass()
>>> thing.method = MagicMock(return_value=3)
>>> thing.method(3, 4, 5, key='value')
3
>>> thing.method.assert_called_with(3, 4, 5, key='value')
```

`side_effect` allows you to perform side effects, including raising an exception when a mock is called:

```
>>> from unittest.mock import Mock
>>> mock = Mock(side_effect=KeyError('foo'))
>>> mock()
Traceback (most recent call last):
 ...
KeyError: 'foo'
```

```
>>> values = {'a': 1, 'b': 2, 'c': 3}
>>> def side_effect(arg):
...     return values[arg]
...
>>> mock.side_effect = side_effect
>>> mock('a'), mock('b'), mock('c')
(1, 2, 3)
>>> mock.side_effect = [5, 4, 3, 2, 1]
>>> mock(), mock(), mock()
(5, 4, 3)
```

Mock has many other ways you can configure it and control its behaviour. For example the *spec* argument configures the mock to take its specification from another object.

Attempting to access attributes or methods on the mock that don’t exist on the spec will fail with an `AttributeError`.

The `patch()` decorator / context manager makes it easy to mock classes or objects in a module under test. The object you specify will be replaced with a mock (or other object) during the test and restored when the test ends:

```
>>> from unittest.mock import patch
>>> @patch('module.ClassName2')
... @patch('module.ClassName1')
... def test(MockClass1, MockClass2):
...     module.ClassName1()
...     module.ClassName2()
...     assert MockClass1 is module.ClassName1
...     assert MockClass2 is module.ClassName2
...     assert MockClass1.called
...     assert MockClass2.called
...
>>> test()
```

Note

When you nest patch decorators the mocks are passed in to the decorated function in the same order they applied (the normal *Python* order that decorators are applied). This means from the bottom up, so in the example above the mock for `module.ClassName1` is passed in first.

With `patch()` it matters that you patch objects in the namespace where they are looked up. This is normally straightforward, but for a quick guide read where to patch.

As well as a decorator `patch()` can be used as a context manager in a with statement:

```
>>> with patch.object(ProductionClass, 'method',
return_value=None) as mock_method:
```

```
...     thing = ProductionClass()
...     thing.method(1, 2, 3)
...
>>> mock_method.assert_called_once_with(1, 2, 3)
```

There is also **`patch.dict()`** for setting values in a dictionary just during a scope and restoring the dictionary to its original state when the test ends:

```
>>> foo = {'key': 'value'}
>>> original = foo.copy()
>>> with patch.dict(foo, {'newkey': 'newvalue'}, clear=True):
...     assert foo == {'newkey': 'newvalue'}
...
>>> assert foo == original
```

Mock supports the mocking of Python magic methods. The easiest way of using magic methods is with the **`MagicMock`** class. It allows you to do things like:

```
>>> mock = MagicMock()
>>> mock.__str__.return_value = 'foobarbaz'
>>> str(mock)
'foobarbaz'
>>> mock.__str__.assert_called_with()
```

Mock allows you to assign functions (or other Mock instances) to magic methods and they will be called appropriately. The **`MagicMock`** class is just a Mock variant that has all of the magic methods pre-created for you (well, all the useful ones anyway).

The following is an example of using magic methods with the ordinary Mock class:

```
>>> mock = Mock()
>>> mock.__str__ = Mock(return_value='wheeeeee')
>>> str(mock)
'wheeeeee'
```

For ensuring that the mock objects in your tests have the same api as the objects they are replacing, you can use auto-speccing. Auto-speccing can be done through the *autospec* argument to patch, or the **`create_autospec()`** function. Auto-speccing creates mock objects that have the same attributes and methods as the objects they are replacing, and any functions and methods (including constructors) have the same call signature as the real object.

This ensures that your mocks will fail in the same way as your production code if they are used incorrectly:

```
>>> from unittest.mock import create_autospec
>>> def function(a, b, c):
...     pass
...
>>> mock_function = create_autospec(function,
return_value='fishy')
>>> mock_function(1, 2, 3)
'fishy'
>>> mock_function.assert_called_once_with(1, 2, 3)
>>> mock_function('wrong arguments')
Traceback (most recent call last):
 ...
TypeError: missing a required argument: 'b'
```

`create_autospec()` can also be used on classes, where it copies the signature of the `__init__` method, and on callable objects where it copies the signature of the `__call__` method.

The Mock Class

`Mock` is a flexible mock object intended to replace the use of stubs and test doubles throughout your code. Mocks are callable and create attributes as new mocks when you access them [[1]]. Accessing the same attribute will always return the same mock. Mocks record how you use them, allowing you to make assertions about what your code has done to them.

`MagicMock` is a subclass of `Mock` with all the magic methods pre-created and ready to use. There are also non-callable variants, useful when you are mocking out objects that aren’t callable: `NonCallableMock` and `NonCallableMagicMock`

The `patch()` decorators makes it easy to temporarily replace classes in a particular module with a `Mock` object. By default `patch()` will create a `MagicMock` for you. You can specify an alternative class of `Mock` using the *new_callable* argument to `patch()`.

class unittest.mock.Mock(*spec=None*, *side_effect=None*, *return_value=DEFAULT*, *wraps=None*, *name=None*, *spec_set=None*, *unsafe=False*, ***kwargs*)

Create a new `Mock` object. `Mock` takes several optional arguments that specify the behaviour of the Mock object:

- *spec*: This can be either a list of strings or an existing object (a class or instance) that acts as the specification for the mock object. If you pass in an object then a list of strings is formed by calling dir on the object (excluding unsupported magic attributes and methods). Accessing any attribute not in this list will raise an `AttributeError`.

  If *spec* is an object (rather than a list of strings) then `__class__` returns the class of the spec object. This allows mocks to pass `isinstance()` tests.

- *spec_set*: A stricter variant of *spec*. If used, attempting to *set* or get an attribute on the mock that isn’t on the object passed as *spec_set* will raise an `AttributeError`.

- *side_effect*: A function to be called whenever the Mock is called. See the [`side_effect`] attribute. Useful for raising exceptions or dynamically changing return values. The function is called with the same arguments as the mock, and unless it returns [`DEFAULT`], the return value of this function is used as the return value.

  Alternatively *side_effect* can be an exception class or instance. In this case the exception will be raised when the mock is called.

  If *side_effect* is an iterable then each call to the mock will return the next value from the iterable.

  A *side_effect* can be cleared by setting it to `None`.

- *return_value*: The value returned when the mock is called. By default this is a new Mock (created on first access). See the [`return_value`] attribute.

- *unsafe*: By default, accessing any attribute whose name starts with *assert*, *assret*, *asert*, *aseert* or *assrt* will raise an [`AttributeError`]. Passing `unsafe=True` will allow access to these attributes.

  Added in version 3.5.

- *wraps*: Item for the mock object to wrap. If *wraps* is not `None` then calling the Mock will pass the call through to the wrapped object (returning the real result). Attribute access on the mock will return a Mock object that wraps the corresponding attribute of the wrapped object (so attempting to access an attribute that doesn’t exist will raise an [`AttributeError`]).

If the mock has an explicit *return_value* set then calls are not passed to the wrapped object and the *return_value* is returned instead.

- *name*: If the mock has a name then it will be used in the repr of the mock. This can be useful for debugging. The name is propagated to child mocks.

Mocks can also be called with arbitrary keyword arguments. These will be used to set attributes on the mock after it is created. See the `configure_mock()` method for details.

assert_called()

Assert that the mock was called at least once.

```
>>> mock = Mock()
>>> mock.method()
<Mock name='mock.method()' id='...'>
>>> mock.method.assert_called()
```

Added in version 3.6.

assert_called_once()

Assert that the mock was called exactly once.

```
>>> mock = Mock()
>>> mock.method()
<Mock name='mock.method()' id='...'>
>>> mock.method.assert_called_once()
>>> mock.method()
<Mock name='mock.method()' id='...'>
>>> mock.method.assert_called_once()
Traceback (most recent call last):
...
AssertionError: Expected 'method' to have been called
once. Called 2 times.
Calls: [call(), call()].
```

Added in version 3.6.

assert_called_with(*args*, *kwargs*)

This method is a convenient way of asserting that the last call has been made in a particular way:

```
>>> mock = Mock()
>>> mock.method(1, 2, 3, test='wow')
<Mock name='mock.method()' id='...'>
>>> mock.method.assert_called_with(1, 2, 3, test='wow')
```

assert_called_once_with(*args*, *kwargs*)

Assert that the mock was called exactly once and that call was with the specified arguments.

```
>>> mock = Mock(return_value=None)
>>> mock('foo', bar='baz')
>>> mock.assert_called_once_with('foo', bar='baz')
>>> mock('other', bar='values')
>>> mock.assert_called_once_with('other', bar='values')
Traceback (most recent call last):
  ...
AssertionError: Expected 'mock' to be called once. Called
2 times.
Calls: [call('foo', bar='baz'), call('other',
bar='values')].
```

assert_any_call(*args*, *kwargs*)

assert the mock has been called with the specified arguments.

The assert passes if the mock has *ever* been called, unlike `assert_called_with()` and `assert_called_once_with()` that only pass if the call is the most recent one, and in the case of `assert_called_once_with()` it must also be the only call.

```
>>> mock = Mock(return_value=None)
>>> mock(1, 2, arg='thing')
>>> mock('some', 'thing', 'else')
>>> mock.assert_any_call(1, 2, arg='thing')
```

assert_has_calls(*calls*, *any_order=False*)

assert the mock has been called with the specified calls. The `mock_calls` list is checked for the calls.

If *any_order* is false then the calls must be sequential. There can be extra calls before or after the specified calls.

If *any_order* is true then the calls can be in any order, but they must all appear in `mock_calls`.

```
>>> mock = Mock(return_value=None)
>>> mock(1)
>>> mock(2)
>>> mock(3)
>>> mock(4)
>>> calls = [call(2), call(3)]
>>> mock.assert_has_calls(calls)
>>> calls = [call(4), call(2), call(3)]
>>> mock.assert_has_calls(calls, any_order=True)
```

assert_not_called()

Assert the mock was never called.

```
>>> m = Mock()
>>> m.hello.assert_not_called()
>>> obj = m.hello()
>>> m.hello.assert_not_called()
Traceback (most recent call last):
  ...
AssertionError: Expected 'hello' to not have been called.
Called 1 times.
Calls: [call()].
```

Added in version 3.5.

reset_mock(*, *return_value=False*, *side_effect=False*)

The reset_mock method resets all the call attributes on a mock object:

```
>>> mock = Mock(return_value=None)
>>> mock('hello')
>>> mock.called
True
>>> mock.reset_mock()
>>> mock.called
False
```

This can be useful where you want to make a series of assertions that reuse the same object.

return_value parameter when set to `True` resets `return_value`:

```
>>> mock = Mock(return_value=5)
>>> mock('hello')
5
>>> mock.reset_mock(return_value=True)
>>> mock('hello')
<Mock name='mock()' id='...'>
```

side_effect parameter when set to `True` resets `side_effect`:

```
>>> mock = Mock(side_effect=ValueError)
>>> mock('hello')
Traceback (most recent call last):
  ...
ValueError
>>> mock.reset_mock(side_effect=True)
>>> mock('hello')
<Mock name='mock()' id='...'>
```

Note that `reset_mock()` *doesn’t* clear the `return_value`, `side_effect` or any child attributes you have set using normal assignment by default.

Child mocks are reset as well.

Changed in version 3.6: Added two keyword-only arguments to the reset_mock function.

mock_add_spec(*spec*, *spec_set=False*)

Add a spec to a mock. *spec* can either be an object or a list of strings. Only attributes on the *spec* can be fetched as attributes from the mock.

If *spec_set* is true then only attributes on the spec can be set.

attach_mock(*mock*, *attribute*)

Attach a mock as an attribute of this one, replacing its name and parent. Calls to the attached mock will be recorded in the **`method_calls`** and **`mock_calls`** attributes of this one.

configure_mock(***kwargs*)

Set attributes on the mock through keyword arguments.

Attributes plus return values and side effects can be set on child mocks using standard dot notation and unpacking a dictionary in the method call:

```
>>> mock = Mock()
>>> attrs = {'method.return_value': 3,
'other.side_effect': KeyError}
>>> mock.configure_mock(**attrs)
>>> mock.method()
3
>>> mock.other()
Traceback (most recent call last):
  ...
KeyError
```

The same thing can be achieved in the constructor call to mocks:

```
>>> attrs = {'method.return_value': 3,
'other.side_effect': KeyError}
```

```
>>> mock = Mock(some_attribute='eggs', **attrs)
>>> mock.some_attribute
'eggs'
>>> mock.method()
3
>>> mock.other()
Traceback (most recent call last):
  ...
KeyError
```

`configure_mock()` exists to make it easier to do configuration after the mock has been created.

__dir__()

`Mock` objects limit the results of `dir(some_mock)` to useful results. For mocks with a *spec* this includes all the permitted attributes for the mock.

See `FILTER_DIR` for what this filtering does, and how to switch it off.

_get_child_mock(***kw*)

Create the child mocks for attributes and return value. By default child mocks will be the same type as the parent. Subclasses of Mock may want to override this to customize the way child mocks are made.

For non-callable mocks the callable variant will be used (rather than any custom subclass).

called

A boolean representing whether or not the mock object has been called:

```
>>> mock = Mock(return_value=None)
>>> mock.called
False
>>> mock()
>>> mock.called
True
```

call_count

An integer telling you how many times the mock object has been called:

```
>>> mock = Mock(return_value=None)
>>> mock.call_count
0
>>> mock()
>>> mock()
>>> mock.call_count
2
```

return_value

Set this to configure the value returned by calling the mock:

```
>>> mock = Mock()
>>> mock.return_value = 'fish'
>>> mock()
'fish'
```

The default return value is a mock object and you can configure it in the normal way:

```
>>> mock = Mock()
>>> mock.return_value.attribute = sentinel.Attribute
>>> mock.return_value()
<Mock name='mock()()' id='...'>
>>> mock.return_value.assert_called_with()
```

`return_value` can also be set in the constructor:

```
>>> mock = Mock(return_value=3)
>>> mock.return_value
3
>>> mock()
3
```

side_effect

This can either be a function to be called when the mock is called, an iterable or an exception (class or instance) to be raised.

If you pass in a function it will be called with same arguments as the mock and unless the function returns the `DEFAULT` singleton the call to the mock will then return whatever the function returns. If the function returns `DEFAULT` then the mock will return its normal value (from the `return_value`).

If you pass in an iterable, it is used to retrieve an iterator which must yield a value on every call. This value can either be an exception instance to be raised, or a value to be returned from the call to the mock (`DEFAULT` handling is identical to the function case).

An example of a mock that raises an exception (to test exception handling of an API):

```
>>> mock = Mock()
>>> mock.side_effect = Exception('Boom!')
>>> mock()
Traceback (most recent call last):
  ...
Exception: Boom!
```

Using `side_effect` to return a sequence of values:

```
>>> mock = Mock()
>>> mock.side_effect = [3, 2, 1]
>>> mock(), mock(), mock()
(3, 2, 1)
```

Using a callable:

```
>>> mock = Mock(return_value=3)
>>> def side_effect(*args, **kwargs):
...     return DEFAULT
...
>>> mock.side_effect = side_effect
>>> mock()
3
```

`side_effect` can be set in the constructor. Here’s an example that adds one to the value the mock is called with and returns it:

```
>>> side_effect = lambda value: value + 1
>>> mock = Mock(side_effect=side_effect)
>>> mock(3)
4
>>> mock(-8)
-7
```

Setting **`side_effect`** to `None` clears it:

```
>>> m = Mock(side_effect=KeyError, return_value=3)
>>> m()
Traceback (most recent call last):
 ...
KeyError
>>> m.side_effect = None
>>> m()
3
```

call_args

This is either `None` (if the mock hasn’t been called), or the arguments that the mock was last called with. This will be in the form of a tuple: the first member, which can also be accessed through the `args` property, is any positional arguments the mock was called with (or an empty tuple) and the second member, which can also be accessed through the `kwargs` property, is any keyword arguments (or an empty dictionary).

```
>>> mock = Mock(return_value=None)
>>> print(mock.call_args)
None
>>> mock()
>>> mock.call_args
call()
```

```
>>> mock.call_args == ()
True
>>> mock(3, 4)
>>> mock.call_args
call(3, 4)
>>> mock.call_args == ((3, 4),)
True
>>> mock.call_args.args
(3, 4)
>>> mock.call_args.kwargs
{}
>>> mock(3, 4, 5, key='fish', next='w00t!')
>>> mock.call_args
call(3, 4, 5, key='fish', next='w00t!')
>>> mock.call_args.args
(3, 4, 5)
>>> mock.call_args.kwargs
{'key': 'fish', 'next': 'w00t!'}
```

`call_args`, along with members of the lists **`call_args_list`**, **`method_calls`** and **`mock_calls`** are **`call`** objects. These are tuples, so they can be unpacked to get at the individual arguments and make more complex assertions. See calls as tuples.

Changed in version 3.8: Added `args` and `kwargs` properties.

call_args_list

This is a list of all the calls made to the mock object in sequence (so the length of the list is the number of times it has been called). Before any calls have been made it is an empty list. The **`call`** object can be used for conveniently constructing lists of calls to compare with **`call_args_list`**.

```
>>> mock = Mock(return_value=None)
>>> mock()
>>> mock(3, 4)
>>> mock(key='fish', next='w00t!')
>>> mock.call_args_list
```

```
[call(), call(3, 4), call(key='fish', next='w00t!')]
>>> expected = [(), ((3, 4),), ({'key': 'fish', 'next':
'w00t!'},)]
>>> mock.call_args_list == expected
True
```

Members of [`call_args_list`](#) are [`call`](#) objects. These can be unpacked as tuples to get at the individual arguments. See [calls as tuples](#).

method_calls

As well as tracking calls to themselves, mocks also track calls to methods and attributes, and *their* methods and attributes:

```
>>> mock = Mock()
>>> mock.method()
<Mock name='mock.method()' id='...'>
>>> mock.property.method.attribute()
<Mock name='mock.property.method.attribute()' id='...'>
>>> mock.method_calls
[call.method(), call.property.method.attribute()]
```

Members of [`method_calls`](#) are [`call`](#) objects. These can be unpacked as tuples to get at the individual arguments. See [calls as tuples](#).

mock_calls

[`mock_calls`](#) records *all* calls to the mock object, its methods, magic methods *and* return value mocks.

```
>>> mock = MagicMock()
>>> result = mock(1, 2, 3)
>>> mock.first(a=3)
<MagicMock name='mock.first()' id='...'>
>>> mock.second()
<MagicMock name='mock.second()' id='...'>
>>> int(mock)
1
>>> result(1)
<MagicMock name='mock()()' id='...'>
```

```
>>> expected = [call(1, 2, 3), call.first(a=3),
call.second(),
... call.__int__(), call()(1)]
>>> mock.mock_calls == expected
True
```

Members of **`mock_calls`** are **`call`** objects. These can be unpacked as tuples to get at the individual arguments. See calls as tuples.

Note

The way **`mock_calls`** are recorded means that where nested calls are made, the parameters of ancestor calls are not recorded and so will always compare equal:

```
>>> mock = MagicMock()
>>> mock.top(a=3).bottom()
<MagicMock name='mock.top().bottom()' id='...'>
>>> mock.mock_calls
[call.top(a=3), call.top().bottom()]
>>> mock.mock_calls[-1] == call.top(a=-1).bottom()
True
```

__class__

Normally the **`__class__`** attribute of an object will return its type. For a mock object with a **`spec`**, **`__class__`** returns the spec class instead. This allows mock objects to pass **`isinstance()`** tests for the object they are replacing / masquerading as:

```
>>> mock = Mock(spec=3)
>>> isinstance(mock, int)
True
```

`__class__` is assignable to, this allows a mock to pass an **`isinstance()`** check without forcing you to use a spec:

```
>>> mock = Mock()
>>> mock.__class__ = dict
>>> isinstance(mock, dict)
True
```

class unittest.mock.NonCallableMock(*spec=None*, *wraps=None*, *name=None*, *spec_set=None*, ***kwargs*)

A non-callable version of **`Mock`**. The constructor parameters have the same meaning of **`Mock`**, with the exception of *return_value* and *side_effect* which have no meaning on a non-callable mock.

Mock objects that use a class or an instance as a **`spec`** or **`spec_set`** are able to pass **`isinstance()`** tests:

```
>>> mock = Mock(spec=SomeClass)
>>> isinstance(mock, SomeClass)
True
>>> mock = Mock(spec_set=SomeClass())
>>> isinstance(mock, SomeClass)
True
```

The **`Mock`** classes have support for mocking magic methods. See magic methods for the full details.

The mock classes and the **`patch()`** decorators all take arbitrary keyword arguments for configuration. For the **`patch()`** decorators the keywords are passed to the constructor of the mock being created. The keyword arguments are for configuring attributes of the mock:

```
>>> m = MagicMock(attribute=3, other='fish')
>>> m.attribute
3
>>> m.other
'fish'
```

The return value and side effect of child mocks can be set in the same way, using dotted notation. As you can’t use dotted names directly in a call you have to create a dictionary and unpack it using `**`:

```
>>> attrs = {'method.return_value': 3, 'other.side_effect':
KeyError}
>>> mock = Mock(some_attribute='eggs', **attrs)
>>> mock.some_attribute
'eggs'
>>> mock.method()
3
>>> mock.other()
Traceback (most recent call last):
  ...
KeyError
```

A callable mock which was created with a *spec* (or a *spec_set*) will introspect the specification object's signature when matching calls to the mock. Therefore, it can match the actual call's arguments regardless of whether they were passed positionally or by name:

```
>>> def f(a, b, c): pass
...
>>> mock = Mock(spec=f)
>>> mock(1, 2, c=3)
<Mock name='mock()' id='140161580456576'>
>>> mock.assert_called_with(1, 2, 3)
>>> mock.assert_called_with(a=1, b=2, c=3)
```

This applies to **`assert_called_with()`**, **`assert_called_once_with()`**, **`assert_has_calls()`** and **`assert_any_call()`**. When Autospeccing, it will also apply to method calls on the mock object.

Changed in version 3.4: Added signature introspection on specced and autospecced mock objects.

class unittest.mock.PropertyMock(**args*, ***kwargs*)

A mock intended to be used as a **`property`**, or other descriptor, on a class. **`PropertyMock`** provides **`__get__()`** and `__set__()` methods so you can specify a return value when it is fetched.

Fetching a `PropertyMock` instance from an object calls the mock, with no args. Setting it calls the mock with the value being set.

```
>>> class Foo:
...     @property
...     def foo(self):
...         return 'something'
...     @foo.setter
...     def foo(self, value):
...         pass
...
>>> with patch('__main__.Foo.foo',
new_callable=PropertyMock) as mock_foo:
...     mock_foo.return_value = 'mockity-mock'
...     this_foo = Foo()
...     print(this_foo.foo)
...     this_foo.foo = 6
...
mockity-mock
>>> mock_foo.mock_calls
[call(), call(6)]
```

Because of the way mock attributes are stored you can’t directly attach a `PropertyMock` to a mock object. Instead you can attach it to the mock type object:

```
>>> m = MagicMock()
>>> p = PropertyMock(return_value=3)
>>> type(m).foo = p
>>> m.foo
3
>>> p.assert_called_once_with()
```

Caution

If an `AttributeError` is raised by `PropertyMock`, it will be interpreted as a missing descriptor and `__getattr__()` will be called on the parent mock:

```
>>> m = MagicMock()
>>> no_attribute = PropertyMock(side_effect=AttributeError)
>>> type(m).my_property = no_attribute
>>> m.my_property
<MagicMock name='mock.my_property' id='140165240345424'>
```

See `__getattr__()` for details.

class unittest.mock.AsyncMock(*spec=None*, *side_effect=None*, *return_value=DEFAULT*, *wraps=None*, *name=None*, *spec_set=None*, *unsafe=False*, ***kwargs*)

An asynchronous version of `MagicMock`. The `AsyncMock` object will behave so the object is recognized as an async function, and the result of a call is an awaitable.

```
>>> mock = AsyncMock()
>>> inspect.iscoroutinefunction(mock)
True
>>> inspect.isawaitable(mock())
True
```

The result of `mock()` is an async function which will have the outcome of `side_effect` or `return_value` after it has been awaited:

- if `side_effect` is a function, the async function will return the result of that function,
- if `side_effect` is an exception, the async function will raise the exception,
- if `side_effect` is an iterable, the async function will return the next value of the iterable, however, if the sequence of result is exhausted, `StopAsyncIteration` is raised immediately,

- if `side_effect` is not defined, the async function will return the value defined by `return_value`, hence, by default, the async function returns a new **`AsyncMock`** object.

Setting the *spec* of a **`Mock`** or **`MagicMock`** to an async function will result in a coroutine object being returned after calling.

```
>>> async def async_func(): pass
...
>>> mock = MagicMock(async_func)
>>> mock
<MagicMock spec='function' id='...'>
>>> mock()
<coroutine object AsyncMockMixin._mock_call at ...>
```

Setting the *spec* of a **`Mock`**, **`MagicMock`**, or **`AsyncMock`** to a class with asynchronous and synchronous functions will automatically detect the synchronous functions and set them as **`MagicMock`** (if the parent mock is **`AsyncMock`** or **`MagicMock`**) or **`Mock`** (if the parent mock is **`Mock`**). All asynchronous functions will be **`AsyncMock`**.

```
>>> class ExampleClass:
...     def sync_foo():
...         pass
...     async def async_foo():
...         pass
...
>>> a_mock = AsyncMock(ExampleClass)
>>> a_mock.sync_foo
<MagicMock name='mock.sync_foo' id='...'>
>>> a_mock.async_foo
<AsyncMock name='mock.async_foo' id='...'>
>>> mock = Mock(ExampleClass)
>>> mock.sync_foo
<Mock name='mock.sync_foo' id='...'>
>>> mock.async_foo
<AsyncMock name='mock.async_foo' id='...'>
```

Added in version 3.8.

assert_awaited()

Assert that the mock was awaited at least once. Note that this is separate from the object having been called, the `await` keyword must be used:

```
>>> mock = AsyncMock()
>>> async def main(coroutine_mock):
...     await coroutine_mock
...
>>> coroutine_mock = mock()
>>> mock.called
True
>>> mock.assert_awaited()
Traceback (most recent call last):
...
AssertionError: Expected mock to have been awaited.
>>> asyncio.run(main(coroutine_mock))
>>> mock.assert_awaited()
```

assert_awaited_once()

Assert that the mock was awaited exactly once.

```
>>> mock = AsyncMock()
>>> async def main():
...     await mock()
...
>>> asyncio.run(main())
>>> mock.assert_awaited_once()
>>> asyncio.run(main())
>>> mock.assert_awaited_once()
Traceback (most recent call last):
...
AssertionError: Expected mock to have been awaited once.
Awaited 2 times.
```

assert_awaited_with(**args*, ***kwargs*)

Assert that the last await was with the specified arguments.

```
>>> mock = AsyncMock()
>>> async def main(*args, **kwargs):
...     await mock(*args, **kwargs)
...
>>> asyncio.run(main('foo', bar='bar'))
>>> mock.assert_awaited_with('foo', bar='bar')
>>> mock.assert_awaited_with('other')
Traceback (most recent call last):
...
AssertionError: expected await not found.
Expected: mock('other')
Actual: mock('foo', bar='bar')
```

assert_awaited_once_with(**args*, ***kwargs*)

Assert that the mock was awaited exactly once and with the specified arguments.

```
>>> mock = AsyncMock()
>>> async def main(*args, **kwargs):
...     await mock(*args, **kwargs)
...
>>> asyncio.run(main('foo', bar='bar'))
>>> mock.assert_awaited_once_with('foo', bar='bar')
>>> asyncio.run(main('foo', bar='bar'))
>>> mock.assert_awaited_once_with('foo', bar='bar')
Traceback (most recent call last):
...
AssertionError: Expected mock to have been awaited once.
Awaited 2 times.
```

assert_any_await(**args*, ***kwargs*)

Assert the mock has ever been awaited with the specified arguments.

```
>>> mock = AsyncMock()
>>> async def main(*args, **kwargs):
...     await mock(*args, **kwargs)
...
>>> asyncio.run(main('foo', bar='bar'))
>>> asyncio.run(main('hello'))
>>> mock.assert_any_await('foo', bar='bar')
>>> mock.assert_any_await('other')
Traceback (most recent call last):
...
AssertionError: mock('other') await not found
```

assert_has_awaits(*calls*, *any_order=False*)

Assert the mock has been awaited with the specified calls. The `await_args_list` list is checked for the awaits.

If *any_order* is false then the awaits must be sequential. There can be extra calls before or after the specified awaits.

If *any_order* is true then the awaits can be in any order, but they must all appear in `await_args_list`.

```
>>> mock = AsyncMock()
>>> async def main(*args, **kwargs):
...     await mock(*args, **kwargs)
...
>>> calls = [call("foo"), call("bar")]
>>> mock.assert_has_awaits(calls)
Traceback (most recent call last):
...
AssertionError: Awaits not found.
Expected: [call('foo'), call('bar')]
Actual: []
>>> asyncio.run(main('foo'))
>>> asyncio.run(main('bar'))
>>> mock.assert_has_awaits(calls)
```

assert_not_awaited()

Assert that the mock was never awaited.

```
>>> mock = AsyncMock()
>>> mock.assert_not_awaited()
```

reset_mock(**args*, ***kwargs*)

See `Mock.reset_mock()`. Also sets `await_count` to 0, `await_args` to None, and clears the `await_args_list`.

await_count

An integer keeping track of how many times the mock object has been awaited.

```
>>> mock = AsyncMock()
>>> async def main():
...     await mock()
...
>>> asyncio.run(main())
>>> mock.await_count
1
>>> asyncio.run(main())
>>> mock.await_count
2
```

await_args

This is either `None` (if the mock hasn't been awaited), or the arguments that the mock was last awaited with. Functions the same as `Mock.call_args`.

```
>>> mock = AsyncMock()
>>> async def main(*args):
...     await mock(*args)
...
>>> mock.await_args
>>> asyncio.run(main('foo'))
>>> mock.await_args
call('foo')
>>> asyncio.run(main('bar'))
>>> mock.await_args
call('bar')
```

await_args_list

This is a list of all the awaits made to the mock object in sequence (so the length of the list is the number of times it has been awaited). Before any awaits have been made it is an empty list.

```
>>> mock = AsyncMock()
>>> async def main(*args):
```

```
...     await mock(*args)
...
>>> mock.await_args_list
[]
>>> asyncio.run(main('foo'))
>>> mock.await_args_list
[call('foo')]
>>> asyncio.run(main('bar'))
>>> mock.await_args_list
[call('foo'), call('bar')]
```

class unittest.mock.ThreadingMock(*spec=None*, *side_effect=None*, *return_value=DEFAULT*, *wraps=None*, *name=None*, *spec_set=None*, *unsafe=False*, *, *timeout=UNSET*, ***kwargs*)

A version of `MagicMock` for multithreading tests. The `ThreadingMock` object provides extra methods to wait for a call to be invoked, rather than assert on it immediately.

The default timeout is specified by the `timeout` argument, or if unset by the `ThreadingMock.DEFAULT_TIMEOUT` attribute, which defaults to blocking (`None`).

You can configure the global default timeout by setting `ThreadingMock.DEFAULT_TIMEOUT`.

wait_until_called(*, *timeout=UNSET*)

Waits until the mock is called.

If a timeout was passed at the creation of the mock or if a timeout argument is passed to this function, the function raises an `AssertionError` if the call is not performed in time.

```
>>> mock = ThreadingMock()
>>> thread = threading.Thread(target=mock)
>>> thread.start()
>>> mock.wait_until_called(timeout=1)
>>> thread.join()
```

wait_until_any_call_with(**args*, ***kwargs*)

Waits until the mock is called with the specified arguments.

If a timeout was passed at the creation of the mock the function raises an `AssertionError` if the call is not performed in time.

```
>>> mock = ThreadingMock()
>>> thread = threading.Thread(target=mock, args=("arg1",
"arg2",), kwargs={"arg": "thing"})
>>> thread.start()
>>> mock.wait_until_any_call_with("arg1", "arg2",
arg="thing")
>>> thread.join()
```

DEFAULT_TIMEOUT

Global default timeout in seconds to create instances of `ThreadingMock`.

Added in version 3.13.

Calling

Mock objects are callable. The call will return the value set as the `return_value` attribute. The default return value is a new Mock object; it is created the first time the return value is accessed (either explicitly or by calling the Mock) - but it is stored and the same one returned each time.

Calls made to the object will be recorded in the attributes like `call_args` and `call_args_list`.

If `side_effect` is set then it will be called after the call has been recorded, so if **`side_effect`** raises an exception the call is still recorded.

The simplest way to make a mock raise an exception when called is to make `side_effect` an exception class or instance:

```
>>> m = MagicMock(side_effect=IndexError)
>>> m(1, 2, 3)
Traceback (most recent call last):
  ...
IndexError
>>> m.mock_calls
[call(1, 2, 3)]
>>> m.side_effect = KeyError('Bang!')
>>> m('two', 'three', 'four')
Traceback (most recent call last):
  ...
KeyError: 'Bang!'
>>> m.mock_calls
[call(1, 2, 3), call('two', 'three', 'four')]
```

If `side_effect` is a function then whatever that function returns is what calls to the mock return. The **`side_effect`** function is called with the same arguments as the mock. This allows you to vary the return value of the call dynamically, based on the input:

```
>>> def side_effect(value):
...     return value + 1
...
>>> m = MagicMock(side_effect=side_effect)
>>> m(1)
2
>>> m(2)
3
>>> m.mock_calls
[call(1), call(2)]
```

If you want the mock to still return the default return value (a new mock), or any set return value, then there are two ways of doing this. Either return `return_value` from inside `side_effect`, or return `DEFAULT`:

```
>>> m = MagicMock()
>>> def side_effect(*args, **kwargs):
...     return m.return_value
...
>>> m.side_effect = side_effect
>>> m.return_value = 3
>>> m()
3
>>> def side_effect(*args, **kwargs):
...     return DEFAULT
...
>>> m.side_effect = side_effect
>>> m()
3
```

To remove a `side_effect`, and return to the default behaviour, set the **`side_effect`** to `None`:

```
>>> m = MagicMock(return_value=6)
>>> def side_effect(*args, **kwargs):
...     return 3
...
>>> m.side_effect = side_effect
>>> m()
3
>>> m.side_effect = None
>>> m()
6
```

The `side_effect` can also be any iterable object. Repeated calls to the mock will return values from the iterable (until the iterable is exhausted and a `StopIteration` is raised):

```
>>> m = MagicMock(side_effect=[1, 2, 3])
>>> m()
1
>>> m()
2
```

```
>>> m()
3
>>> m()
Traceback (most recent call last):
  ...
StopIteration
```

If any members of the iterable are exceptions they will be raised instead of returned:

```
>>> iterable = (33, ValueError, 66)
>>> m = MagicMock(side_effect=iterable)
>>> m()
33
>>> m()
Traceback (most recent call last):
 ...
ValueError
>>> m()
66
```

Deleting Attributes

Mock objects create attributes on demand. This allows them to pretend to be objects of any type.

You may want a mock object to return `False` to a `hasattr()` call, or raise an `AttributeError` when an attribute is fetched. You can do this by providing an object as a **`spec`** for a mock, but that isn't always convenient.

You “block” attributes by deleting them. Once deleted, accessing an attribute will raise an `AttributeError`.

```
>>> mock = MagicMock()
>>> hasattr(mock, 'm')
True
>>> del mock.m
>>> hasattr(mock, 'm')
False
```

```
>>> del mock.f
>>> mock.f
Traceback (most recent call last):
    ...
AttributeError: f
```

Mock names and the name attribute

Since “name” is an argument to the `Mock` constructor, if you want your mock object to have a “name” attribute you can’t just pass it in at creation time. There are two alternatives. One option is to use `configure_mock()`:

```
>>> mock = MagicMock()
>>> mock.configure_mock(name='my_name')
>>> mock.name
'my_name'
```

A simpler option is to simply set the “name” attribute after mock creation:

```
>>> mock = MagicMock()
>>> mock.name = "foo"
```

Attaching Mocks as Attributes

When you attach a mock as an attribute of another mock (or as the return value) it becomes a “child” of that mock. Calls to the child are recorded in the `method_calls` and `mock_calls` attributes of the parent. This is useful for configuring child mocks and then attaching them to the parent, or for attaching mocks to a parent that records all calls to the children and allows you to make assertions about the order of calls between mocks:

```
>>> parent = MagicMock()
>>> child1 = MagicMock(return_value=None)
>>> child2 = MagicMock(return_value=None)
>>> parent.child1 = child1
>>> parent.child2 = child2
>>> child1(1)
>>> child2(2)
>>> parent.mock_calls
[call.child1(1), call.child2(2)]
```

The exception to this is if the mock has a name. This allows you to prevent the “parenting” if for some reason you don’t want it to happen.

```
>>> mock = MagicMock()
>>> not_a_child = MagicMock(name='not-a-child')
>>> mock.attribute = not_a_child
>>> mock.attribute()
<MagicMock name='not-a-child()' id='...'>
>>> mock.mock_calls
[]
```

Mocks created for you by `patch()` are automatically given names. To attach mocks that have names to a parent you use the `attach_mock()` method:

```
>>> thing1 = object()
>>> thing2 = object()
>>> parent = MagicMock()
>>> with patch('__main__.thing1', return_value=None) as child1:
...     with patch('__main__.thing2', return_value=None) as
child2:
...         parent.attach_mock(child1, 'child1')
...         parent.attach_mock(child2, 'child2')
...         child1('one')
...         child2('two')
...
>>> parent.mock_calls
[call.child1('one'), call.child2('two')]
```

[1]

The only exceptions are magic methods and attributes (those that have leading and trailing double underscores). Mock doesn’t create these but instead raises an

`AttributeError`. This is because the interpreter will often implicitly request these methods, and gets *very* confused to get a new Mock object when it expects a magic method. If you need magic method support see magic methods.

The patchers

The patch decorators are used for patching objects only within the scope of the function they decorate. They automatically handle the unpatching for you, even if exceptions are raised. All of these functions can also be used in with statements or as class decorators.

patch

Note

The key is to do the patching in the right namespace. See the section where to patch.

unittest.mock.patch(*target*, *new=DEFAULT*, *spec=None*, *create=False*, *spec_set=None*, *autospec=None*, *new_callable=None*, ***kwargs*)

`patch()` acts as a function decorator, class decorator or a context manager. Inside the body of the function or with statement, the *target* is patched with a *new* object. When the function/with statement exits the patch is undone.

If *new* is omitted, then the target is replaced with an `AsyncMock` if the patched object is an async function or a `MagicMock` otherwise. If `patch()` is used as a decorator and *new* is omitted, the created mock is passed in as an extra argument to the decorated function. If `patch()` is used as a context manager the created mock is returned by the context manager.

target should be a string in the form `'package.module.ClassName'`. The *target* is imported and the specified object replaced with the *new* object, so the *target* must be importable from the environment you are calling **`patch()`** from. The target is imported when the decorated function is executed, not at decoration time.

The *spec* and *spec_set* keyword arguments are passed to the **`MagicMock`** if patch is creating one for you.

In addition you can pass `spec=True` or `spec_set=True`, which causes patch to pass in the object being mocked as the spec/spec_set object.

new_callable allows you to specify a different class, or callable object, that will be called to create the *new* object. By default **`AsyncMock`** is used for async functions and **`MagicMock`** for the rest.

A more powerful form of *spec* is *autospec*. If you set `autospec=True` then the mock will be created with a spec from the object being replaced. All attributes of the mock will also have the spec of the corresponding attribute of the object being replaced. Methods and functions being mocked will have their arguments checked and will raise a **`TypeError`** if they are called with the wrong signature. For mocks replacing a class, their return value (the ‘instance’) will have the same spec as the class. See the **`create_autospec()`** function and Autospeccing.

Instead of `autospec=True` you can pass `autospec=some_object` to use an arbitrary object as the spec instead of the one being replaced.

By default `patch()` will fail to replace attributes that don't exist. If you pass in `create=True`, and the attribute doesn't exist, patch will create the attribute for you when the patched function is called, and delete it again after the patched function has exited. This is useful for writing tests against attributes that your production code creates at runtime. It is off by default because it can be dangerous. With it switched on you can write passing tests against APIs that don't actually exist!

Note

Changed in version 3.5: If you are patching builtins in a module then you don't need to pass `create=True`, it will be added by default.

Patch can be used as a `TestCase` class decorator. It works by decorating each test method in the class. This reduces the boilerplate code when your test methods share a common patchings set. `patch()` finds tests by looking for method names that start with `patch.TEST_PREFIX`. By default this is `'test'`, which matches the way `unittest` finds tests. You can specify an alternative prefix by setting `patch.TEST_PREFIX`.

Patch can be used as a context manager, with the with statement. Here the patching applies to the indented block after the with statement. If you use "as" then the patched object will be bound to the name after the "as"; very useful if `patch()` is creating a mock object for you.

`patch()` takes arbitrary keyword arguments. These will be passed to `AsyncMock` if the patched object is asynchronous, to `MagicMock` otherwise or to *new_callable* if specified.

`patch.dict(...)`, `patch.multiple(...)` and `patch.object(...)` are available for alternate use-cases.

`patch()` as function decorator, creating the mock for you and passing it into the decorated function:

```
>>> @patch('__main__.SomeClass')
... def function(normal_argument, mock_class):
...     print(mock_class is SomeClass)
...
>>> function(None)
True
```

Patching a class replaces the class with a **`MagicMock`** *instance*. If the class is instantiated in the code under test then it will be the **`return_value`** of the mock that will be used.

If the class is instantiated multiple times you could use **`side_effect`** to return a new mock each time. Alternatively you can set the *return_value* to be anything you want.

To configure return values on methods of *instances* on the patched class you must do this on the **`return_value`**. For example:

```
>>> class Class:
...     def method(self):
...         pass
...
>>> with patch('__main__.Class') as MockClass:
...     instance = MockClass.return_value
...     instance.method.return_value = 'foo'
...     assert Class() is instance
...     assert Class().method() == 'foo'
...
```

If you use *spec* or *spec_set* and **`patch()`** is replacing a *class*, then the return value of the created mock will have the same spec.

```
>>> Original = Class
>>> patcher = patch('__main__.Class', spec=True)
>>> MockClass = patcher.start()
>>> instance = MockClass()
>>> assert isinstance(instance, Original)
>>> patcher.stop()
```

The *new_callable* argument is useful where you want to use an alternative class to the default **`MagicMock`** for the created mock. For example, if you wanted a **`NonCallableMock`** to be used:

```
>>> thing = object()
>>> with patch('__main__.thing', new_callable=NonCallableMock)
as mock_thing:
...     assert thing is mock_thing
...     thing()
...
Traceback (most recent call last):
  ...
TypeError: 'NonCallableMock' object is not callable
```

Another use case might be to replace an object with an **`io.StringIO`** instance:

```
>>> from io import StringIO
>>> def foo():
...     print('Something')
...
>>> @patch('sys.stdout', new_callable=StringIO)
... def test(mock_stdout):
...     foo()
...     assert mock_stdout.getvalue() == 'Something\n'
...
>>> test()
```

When **`patch()`** is creating a mock for you, it is common that the first thing you need to do is to configure the mock. Some of that configuration can be done in the call to patch. Any arbitrary keywords you pass into the call will be used to set attributes on the created mock:

```
>>> patcher = patch('__main__.thing', first='one',
second='two')
>>> mock_thing = patcher.start()
>>> mock_thing.first
'one'
>>> mock_thing.second
'two'
```

As well as attributes on the created mock attributes, like the `return_value` and `side_effect`, of child mocks can also be configured. These aren’t syntactically valid to pass in directly as keyword arguments, but a dictionary with these as keys can still be expanded into a `patch()` call using `**`:

```
>>> config = {'method.return_value': 3, 'other.side_effect':
KeyError}
>>> patcher = patch('__main__.thing', **config)
>>> mock_thing = patcher.start()
>>> mock_thing.method()
3
>>> mock_thing.other()
Traceback (most recent call last):
  ...
KeyError
```

By default, attempting to patch a function in a module (or a method or an attribute in a class) that does not exist will fail with `AttributeError`:

```
>>> @patch('sys.non_existing_attribute', 42)
... def test():
...     assert sys.non_existing_attribute == 42
...
>>> test()
Traceback (most recent call last):
  ...
AttributeError: <module 'sys' (built-in)> does not have the
attribute 'non_existing_attribute'
```

but adding `create=True` in the call to `patch()` will make the previous example work as expected:

```
>>> @patch('sys.non_existing_attribute', 42, create=True)
... def test(mock_stdout):
...     assert sys.non_existing_attribute == 42
...
>>> test()
```

Changed in version 3.8: `patch()` now returns an `AsyncMock` if the target is an async function.

patch.object

patch.object(*target*, *attribute*, *new=DEFAULT*, *spec=None*, *create=False*, *spec_set=None*, *autospec=None*, *new_callable=None*, ***kwargs*)

patch the named member (*attribute*) on an object (*target*) with a mock object.

`patch.object()` can be used as a decorator, class decorator or a context manager. Arguments *new*, *spec*, *create*, *spec_set*, *autospec* and *new_callable* have the same meaning as for `patch()`. Like `patch()`, `patch.object()` takes arbitrary keyword arguments for configuring the mock object it creates.

When used as a class decorator `patch.object()` honours `patch.TEST_PREFIX` for choosing which methods to wrap.

You can either call `patch.object()` with three arguments or two arguments. The three argument form takes the object to be patched, the attribute name and the object to replace the attribute with.

When calling with the two argument form you omit the replacement object, and a mock is created for you and passed in as an extra argument to the decorated function:

```
>>> @patch.object(SomeClass, 'class_method')
... def test(mock_method):
...     SomeClass.class_method(3)
...     mock_method.assert_called_with(3)
...
>>> test()
```

spec, *create* and the other arguments to `patch.object()` have the same meaning as they do for `patch()`.

patch.dict

patch.dict(*in_dict*, *values=()*, *clear=False*, ***kwargs*)

Patch a dictionary, or dictionary like object, and restore the dictionary to its original state after the test, where the restored dictionary is a copy of the dictionary as it was before the test.

in_dict can be a dictionary or a mapping like container. If it is a mapping then it must at least support getting, setting and deleting items plus iterating over keys.

in_dict can also be a string specifying the name of the dictionary, which will then be fetched by importing it.

values can be a dictionary of values to set in the dictionary. *values* can also be an iterable of `(key, value)` pairs.

If *clear* is true then the dictionary will be cleared before the new values are set.

`patch.dict()` can also be called with arbitrary keyword arguments to set values in the dictionary.

Changed in version 3.8: `patch.dict()` now returns the patched dictionary when used as a context manager.

`patch.dict()` can be used as a context manager, decorator or class decorator:

```
>>> foo = {}
>>> @patch.dict(foo, {'newkey': 'newvalue'})
... def test():
...     assert foo == {'newkey': 'newvalue'}
...
>>> test()
>>> assert foo == {}
```

When used as a class decorator **`patch.dict()`** honours `patch.TEST_PREFIX` (default to `'test'`) for choosing which methods to wrap:

```
>>> import os
>>> import unittest
>>> from unittest.mock import patch
>>> @patch.dict('os.environ', {'newkey': 'newvalue'})
... class TestSample(unittest.TestCase):
...     def test_sample(self):
...         self.assertEqual(os.environ['newkey'], 'newvalue')
```

If you want to use a different prefix for your test, you can inform the patchers of the different prefix by setting `patch.TEST_PREFIX`. For more details about how to change the value of see TEST_PREFIX.

`patch.dict()` can be used to add members to a dictionary, or simply let a test change a dictionary, and ensure the dictionary is restored when the test ends.

```
>>> foo = {}
>>> with patch.dict(foo, {'newkey': 'newvalue'}) as
patched_foo:
...     assert foo == {'newkey': 'newvalue'}
...     assert patched_foo == {'newkey': 'newvalue'}
...     # You can add, update or delete keys of foo (or
patched_foo, it's the same dict)
...     patched_foo['spam'] = 'eggs'
```

```
...
>>> assert foo == {}
>>> assert patched_foo == {}
```

```
>>> import os
>>> with patch.dict('os.environ', {'newkey': 'newvalue'}):
...     print(os.environ['newkey'])
...
newvalue
>>> assert 'newkey' not in os.environ
```

Keywords can be used in the `patch.dict()` call to set values in the dictionary:

```
>>> mymodule = MagicMock()
>>> mymodule.function.return_value = 'fish'
>>> with patch.dict('sys.modules', mymodule=mymodule):
...     import mymodule
...     mymodule.function('some', 'args')
...
'fish'
```

`patch.dict()` can be used with dictionary like objects that aren’t actually dictionaries. At the very minimum they must support item getting, setting, deleting and either iteration or membership test. This corresponds to the magic methods `__getitem__()`, `__setitem__()`, `__delitem__()` and either `__iter__()` or `__contains__()`.

```
>>> class Container:
...     def __init__(self):
...         self.values = {}
...     def __getitem__(self, name):
...         return self.values[name]
...     def __setitem__(self, name, value):
...         self.values[name] = value
...     def __delitem__(self, name):
...         del self.values[name]
...     def __iter__(self):
...         return iter(self.values)
...
```

```
>>> thing = Container()
>>> thing['one'] = 1
>>> with patch.dict(thing, one=2, two=3):
...     assert thing['one'] == 2
...     assert thing['two'] == 3
...
>>> assert thing['one'] == 1
>>> assert list(thing) == ['one']
```

patch.multiple

patch.multiple(*target*, *spec=None*, *create=False*, *spec_set=None*, *autospec=None*, *new_callable=None*, ***kwargs*)

Perform multiple patches in a single call. It takes the object to be patched (either as an object or a string to fetch the object by importing) and keyword arguments for the patches:

```
with patch.multiple(settings, FIRST_PATCH='one',
SECOND_PATCH='two'):
    ...
```

Use **`DEFAULT`** as the value if you want **`patch.multiple()`** to create mocks for you. In this case the created mocks are passed into a decorated function by keyword, and a dictionary is returned when **`patch.multiple()`** is used as a context manager.

`patch.multiple()` can be used as a decorator, class decorator or a context manager. The arguments *spec*, *spec_set*, *create*, *autospec* and *new_callable* have the same meaning as for **`patch()`**. These arguments will be applied to *all* patches done by **`patch.multiple()`**.

When used as a class decorator **`patch.multiple()`** honours `patch.TEST_PREFIX` for choosing which methods to wrap.

If you want `patch.multiple()` to create mocks for you, then you can use `DEFAULT` as the value. If you use `patch.multiple()` as a decorator then the created mocks are passed into the decorated function by keyword.

```
>>> thing = object()
>>> other = object()

>>> @patch.multiple('__main__', thing=DEFAULT, other=DEFAULT)
... def test_function(thing, other):
...     assert isinstance(thing, MagicMock)
...     assert isinstance(other, MagicMock)
...
>>> test_function()
```

`patch.multiple()` can be nested with other `patch` decorators, but put arguments passed by keyword *after* any of the standard arguments created by `patch()`:

```
>>> @patch('sys.exit')
... @patch.multiple('__main__', thing=DEFAULT, other=DEFAULT)
... def test_function(mock_exit, other, thing):
...     assert 'other' in repr(other)
...     assert 'thing' in repr(thing)
...     assert 'exit' in repr(mock_exit)
...
>>> test_function()
```

If `patch.multiple()` is used as a context manager, the value returned by the context manager is a dictionary where created mocks are keyed by name:

```
>>> with patch.multiple('__main__', thing=DEFAULT,
other=DEFAULT) as values:
...     assert 'other' in repr(values['other'])
...     assert 'thing' in repr(values['thing'])
...     assert values['thing'] is thing
...     assert values['other'] is other
...
```

patch methods: start and stop

All the patchers have **`start()`** and **`stop()`** methods. These make it simpler to do patching in `setUp` methods or where you want to do multiple patches without nesting decorators or with statements.

To use them call **`patch()`**, **`patch.object()`** or **`patch.dict()`** as normal and keep a reference to the returned `patcher` object. You can then call **`start()`** to put the patch in place and **`stop()`** to undo it.

If you are using **`patch()`** to create a mock for you then it will be returned by the call to `patcher.start`.

```
>>> patcher = patch('package.module.ClassName')
>>> from package import module
>>> original = module.ClassName
>>> new_mock = patcher.start()
>>> assert module.ClassName is not original
>>> assert module.ClassName is new_mock
>>> patcher.stop()
>>> assert module.ClassName is original
>>> assert module.ClassName is not new_mock
```

A typical use case for this might be for doing multiple patches in the `setUp` method of a **`TestCase`**:

```
>>> class MyTest(unittest.TestCase):
...     def setUp(self):
...         self.patcher1 = patch('package.module.Class1')
...         self.patcher2 = patch('package.module.Class2')
...         self.MockClass1 = self.patcher1.start()
...         self.MockClass2 = self.patcher2.start()
...
...     def tearDown(self):
...         self.patcher1.stop()
...         self.patcher2.stop()
...
...     def test_something(self):
```

```
...         assert package.module.Class1 is self.MockClass1
...         assert package.module.Class2 is self.MockClass2
...
>>> MyTest('test_something').run()
```

Caution

If you use this technique you must ensure that the patching is "undone" by calling `stop`. This can be fiddlier than you might think, because if an exception is raised in the `setUp` then `tearDown` is not called. **`unittest.TestCase.addCleanup()`** makes this easier:

```
>>> class MyTest(unittest.TestCase):
...     def setUp(self):
...         patcher = patch('package.module.Class')
...         self.MockClass = patcher.start()
...         self.addCleanup(patcher.stop)
...
...     def test_something(self):
...         assert package.module.Class is self.MockClass
...
```

As an added bonus you no longer need to keep a reference to the `patcher` object.

It is also possible to stop all patches which have been started by using **`patch.stopall()`**.

patch.stopall()

Stop all active patches. Only stops patches started with `start`.

patch builtins

You can patch any builtins within a module. The following example patches builtin **`ord()`**:

```
>>> @patch('__main__.ord')
... def test(mock_ord):
...     mock_ord.return_value = 101
...     print(ord('c'))
...
>>> test()
101
```

TEST_PREFIX

All of the patchers can be used as class decorators. When used in this way they wrap every test method on the class. The patchers recognise methods that start with `'test'` as being test methods. This is the same way that the **`unittest.TestLoader`** finds test methods by default.

It is possible that you want to use a different prefix for your tests. You can inform the patchers of the different prefix by setting `patch.TEST_PREFIX`:

```
>>> patch.TEST_PREFIX = 'foo'
>>> value = 3
>>>
>>> @patch('__main__.value', 'not three')
... class Thing:
...     def foo_one(self):
...         print(value)
...     def foo_two(self):
...         print(value)
...
>>>
>>> Thing().foo_one()
not three
>>> Thing().foo_two()
not three
>>> value
3
```

Nesting Patch Decorators

If you want to perform multiple patches then you can simply stack up the decorators.

You can stack up multiple patch decorators using this pattern:

```
>>> @patch.object(SomeClass, 'class_method')
... @patch.object(SomeClass, 'static_method')
... def test(mock1, mock2):
...     assert SomeClass.static_method is mock1
...     assert SomeClass.class_method is mock2
...     SomeClass.static_method('foo')
...     SomeClass.class_method('bar')
...     return mock1, mock2
...
>>> mock1, mock2 = test()
>>> mock1.assert_called_once_with('foo')
>>> mock2.assert_called_once_with('bar')
```

Note that the decorators are applied from the bottom upwards. This is the standard way that Python applies decorators. The order of the created mocks passed into your test function matches this order.

Where to patch

`patch()` works by (temporarily) changing the object that a *name* points to with another one. There can be many names pointing to any individual object, so for patching to work you must ensure that you patch the name used by the system under test.

The basic principle is that you patch where an object is *looked up*, which is not necessarily the same place as where it is defined. A couple of examples will help to clarify this.

Imagine we have a project that we want to test with the following structure:

```
a.py
    -> Defines SomeClass

b.py
    -> from a import SomeClass
    -> some_function instantiates SomeClass
```

Now we want to test `some_function` but we want to mock out `SomeClass` using **`patch()`**. The problem is that when we import module b, which we will have to do when it imports `SomeClass` from module a. If we use **`patch()`** to mock out `a.SomeClass` then it will have no effect on our test; module b already has a reference to the *real* `SomeClass` and it looks like our patching had no effect.

The key is to patch out `SomeClass` where it is used (or where it is looked up). In this case `some_function` will actually look up `SomeClass` in module b, where we have imported it. The patching should look like:

```
@patch('b.SomeClass')
```

However, consider the alternative scenario where instead of `from a import SomeClass` module b does `import a` and `some_function` uses `a.SomeClass`. Both of these import forms are common. In this case the class we want to patch is being looked up in the module and so we have to patch `a.SomeClass` instead:

```
@patch('a.SomeClass')
```

Patching Descriptors and Proxy Objects

Both patch and patch.object correctly patch and restore descriptors: class methods, static methods and properties. You should patch these on the *class* rather than an instance. They also work with *some* objects that proxy attribute access, like the django settings object [https://web.archive.org/web/20200603181648/http://www.voidspace.org.uk/python/weblog/arch_d7_2010_12_04.shtml#e1198].

MagicMock and magic method support

Mocking Magic Methods

`Mock` supports mocking the Python protocol methods, also known as “magic methods”. This allows mock objects to replace containers or other objects that implement Python protocols.

Because magic methods are looked up differently from normal methods [[2]], this support has been specially implemented. This means that only specific magic methods are supported. The supported list includes *almost* all of them. If there are any missing that you need please let us know.

You mock magic methods by setting the method you are interested in to a function or a mock instance. If you are using a function then it *must* take `self` as the first argument [[3]].

```
>>> def __str__(self):
...     return 'fooble'
...
>>> mock = Mock()
>>> mock.__str__ = __str__
>>> str(mock)
'fooble'
```

```
>>> mock = Mock()
>>> mock.__str__ = Mock()
>>> mock.__str__.return_value = 'fooble'
>>> str(mock)
'fooble'
```

```
>>> mock = Mock()
>>> mock.__iter__ = Mock(return_value=iter([]))
>>> list(mock)
[]
```

One use case for this is for mocking objects used as context managers in a `with` statement:

```
>>> mock = Mock()
>>> mock.__enter__ = Mock(return_value='foo')
>>> mock.__exit__ = Mock(return_value=False)
>>> with mock as m:
...     assert m == 'foo'
...
>>> mock.__enter__.assert_called_with()
>>> mock.__exit__.assert_called_with(None, None, None)
```

Calls to magic methods do not appear in `method_calls`, but they are recorded in `mock_calls`.

Note

If you use the *spec* keyword argument to create a mock then attempting to set a magic method that isn't in the spec will raise an `AttributeError`.

The full list of supported magic methods is:

- `__hash__`, `__sizeof__`, `__repr__` and `__str__`
- `__dir__`, `__format__` and `__subclasses__`
- `__round__`, `__floor__`, `__trunc__` and `__ceil__`
- Comparisons: `__lt__`, `__gt__`, `__le__`, `__ge__`, `__eq__` and `__ne__`
- Container methods: `__getitem__`, `__setitem__`, `__delitem__`, `__contains__`, `__len__`, `__iter__`, `__reversed__` and `__missing__`

- Context manager: `__enter__`, `__exit__`, `__aenter__` and `__aexit__`
- Unary numeric methods: `__neg__`, `__pos__` and `__invert__`
- The numeric methods (including right hand and in-place variants): `__add__`, `__sub__`, `__mul__`, `__matmul__`, `__truediv__`, `__floordiv__`, `__mod__`, `__divmod__`, `__lshift__`, `__rshift__`, `__and__`, `__xor__`, `__or__`, and `__pow__`
- Numeric conversion methods: `__complex__`, `__int__`, `__float__` and `__index__`
- Descriptor methods: `__get__`, `__set__` and `__delete__`
- Pickling: `__reduce__`, `__reduce_ex__`, `__getinitargs__`, `__getnewargs__`, `__getstate__` and `__setstate__`
- File system path representation: `__fspath__`
- Asynchronous iteration methods: `__aiter__` and `__anext__`

Changed in version 3.8: Added support for **`os.PathLike.__fspath__()`**.

Changed in version 3.8: Added support for `__aenter__`, `__aexit__`, `__aiter__` and `__anext__`.

The following methods exist but are *not* supported as they are either in use by mock, can’t be set dynamically, or can cause problems:

- `__getattr__`, `__setattr__`, `__init__` and `__new__`
- `__prepare__`, `__instancecheck__`, `__subclasscheck__`, `__del__`

Magic Mock

There are two `MagicMock` variants: **`MagicMock`** and **`NonCallableMagicMock`**.

class unittest.mock.MagicMock(**args*, ***kw*)

`MagicMock` is a subclass of **`Mock`** with default implementations of most of the magic methods. You can use `MagicMock` without having to configure the magic methods yourself.

The constructor parameters have the same meaning as for **`Mock`**.

If you use the *spec* or *spec_set* arguments then *only* magic methods that exist in the spec will be created.

class unittest.mock.NonCallableMagicMock(**args*, ***kw*)

A non-callable version of **`MagicMock`**.

The constructor parameters have the same meaning as for **`MagicMock`**, with the exception of *return_value* and *side_effect* which have no meaning on a non-callable mock.

The magic methods are setup with **`MagicMock`** objects, so you can configure them and use them in the usual way:

```
>>> mock = MagicMock()
>>> mock[3] = 'fish'
>>> mock.__setitem__.assert_called_with(3, 'fish')
>>> mock.__getitem__.return_value = 'result'
>>> mock[2]
'result'
```

By default many of the protocol methods are required to return objects of a specific type. These methods are preconfigured with a default return value, so that they can be used without you having to do anything if you aren’t interested in the return value. You can still *set* the return value manually if you want to change the default.

Methods and their defaults:

- `__lt__`: **`NotImplemented`**
- `__gt__`: **`NotImplemented`**

- `__le__`: **`NotImplemented`**
- `__ge__`: **`NotImplemented`**
- `__int__`: `1`
- `__contains__`: `False`
- `__len__`: `0`
- `__iter__`: `iter([])`
- `__exit__`: `False`
- `__aexit__`: `False`
- `__complex__`: `1j`
- `__float__`: `1.0`
- `__bool__`: `True`
- `__index__`: `1`
- `__hash__`: default hash for the mock
- `__str__`: default str for the mock
- `__sizeof__`: default sizeof for the mock

For example:

```
>>> mock = MagicMock()
>>> int(mock)
1
>>> len(mock)
0
>>> list(mock)
[]
>>> object() in mock
False
```

The two equality methods, **`__eq__()`** and **`__ne__()`**, are special. They do the default equality comparison on identity, using the **`side_effect`** attribute, unless you change their return value to return something else:

```
>>> MagicMock() == 3
False
>>> MagicMock() != 3
True
>>> mock = MagicMock()
>>> mock.__eq__.return_value = True
>>> mock == 3
True
```

The return value of **`MagicMock.__iter__()`** can be any iterable object and isn’t required to be an iterator:

```
>>> mock = MagicMock()
>>> mock.__iter__.return_value = ['a', 'b', 'c']
>>> list(mock)
['a', 'b', 'c']
>>> list(mock)
['a', 'b', 'c']
```

If the return value *is* an iterator, then iterating over it once will consume it and subsequent iterations will result in an empty list:

```
>>> mock.__iter__.return_value = iter(['a', 'b', 'c'])
>>> list(mock)
['a', 'b', 'c']
>>> list(mock)
[]
```

`MagicMock` has all of the supported magic methods configured except for some of the obscure and obsolete ones. You can still set these up if you want.

Magic methods that are supported but not setup by default in `MagicMock` are:

- `__subclasses__`
- `__dir__`
- `__format__`
- `__get__`, `__set__` and `__delete__`
- `__reversed__` and `__missing__`
- `__reduce__`, `__reduce_ex__`, `__getinitargs__`, `__getnewargs__`, `__getstate__` and `__setstate__`
- `__getformat__`

[2]

Magic methods *should* be looked up on the class rather than the instance. Different versions of Python are inconsistent about applying this rule. The supported protocol methods should work with all supported versions of Python.

[3]

The function is basically hooked up to the class, but each `Mock` instance is kept isolated from the others.

Helpers

sentinel

unittest.mock.sentinel

> The `sentinel` object provides a convenient way of providing unique objects for your tests.
>
> Attributes are created on demand when you access them by name. Accessing the same attribute will always return the same object. The objects returned have a sensible repr so that test failure messages are readable.
>
> *Changed in version 3.7:* The `sentinel` attributes now preserve their identity when they are **`copied`** or **`pickled`**.

Sometimes when testing you need to test that a specific object is passed as an argument to another method, or returned. It can be common to create named sentinel objects to test this. **`sentinel`** provides a convenient way of creating and testing the identity of objects like this.

In this example we monkey patch `method` to return `sentinel.some_object`:

```
>>> real = ProductionClass()
>>> real.method = Mock(name="method")
>>> real.method.return_value = sentinel.some_object
>>> result = real.method()
>>> assert result is sentinel.some_object
>>> result
sentinel.some_object
```

DEFAULT

unittest.mock.DEFAULT

The **`DEFAULT`** object is a pre-created sentinel (actually `sentinel.DEFAULT`). It can be used by **`side_effect`** functions to indicate that the normal return value should be used.

call

unittest.mock.call(**args*, ***kwargs*)

`call()` is a helper object for making simpler assertions, for comparing with **`call_args`**, **`call_args_list`**, **`mock_calls`** and **`method_calls`**. **`call()`** can also be used with **`assert_has_calls()`**.

```
>>> m = MagicMock(return_value=None)
>>> m(1, 2, a='foo', b='bar')
>>> m()
>>> m.call_args_list == [call(1, 2, a='foo', b='bar'),
call()]
True
```

call.call_list()

For a call object that represents multiple calls, **`call_list()`** returns a list of all the intermediate calls as well as the final call.

`call_list` is particularly useful for making assertions on “chained calls”. A chained call is multiple calls on a single line of code. This results in multiple entries in **`mock_calls`** on a mock. Manually constructing the sequence of calls can be tedious.

`call_list()` can construct the sequence of calls from the same chained call:

```
>>> m = MagicMock()
>>> m(1).method(arg='foo').other('bar')(2.0)
<MagicMock name='mock().method().other()()' id='...'>
>>> kall = call(1).method(arg='foo').other('bar')(2.0)
>>> kall.call_list()
[call(1),
 call().method(arg='foo'),
 call().method().other('bar'),
 call().method().other()(2.0)]
>>> m.mock_calls == kall.call_list()
True
```

A `call` object is either a tuple of (positional args, keyword args) or (name, positional args, keyword args) depending on how it was constructed. When you construct them yourself this isn’t particularly interesting, but the `call` objects that are in the `Mock.call_args`, `Mock.call_args_list` and `Mock.mock_calls` attributes can be introspected to get at the individual arguments they contain.

The `call` objects in `Mock.call_args` and `Mock.call_args_list` are two-tuples of (positional args, keyword args) whereas the `call` objects in `Mock.mock_calls`, along with ones you construct yourself, are three-tuples of (name, positional args, keyword args).

You can use their “tupleness” to pull out the individual arguments for more complex introspection and assertions. The positional arguments are a tuple (an empty tuple if there are no positional arguments) and the keyword arguments are a dictionary:

```
>>> m = MagicMock(return_value=None)
>>> m(1, 2, 3, arg='one', arg2='two')
>>> kall = m.call_args
>>> kall.args
(1, 2, 3)
>>> kall.kwargs
{'arg': 'one', 'arg2': 'two'}
>>> kall.args is kall[0]
True
>>> kall.kwargs is kall[1]
True
```

```
>>> m = MagicMock()
>>> m.foo(4, 5, 6, arg='two', arg2='three')
<MagicMock name='mock.foo()' id='...'>
>>> kall = m.mock_calls[0]
>>> name, args, kwargs = kall
>>> name
'foo'
>>> args
(4, 5, 6)
>>> kwargs
{'arg': 'two', 'arg2': 'three'}
>>> name is m.mock_calls[0][0]
True
```

create_autospec

unittest.mock.create_autospec(*spec*, *spec_set=False*, *instance=False*, ***kwargs*)

Create a mock object using another object as a spec. Attributes on the mock will use the corresponding attribute on the *spec* object as their spec.

Functions or methods being mocked will have their arguments checked to ensure that they are called with the correct signature.

If *spec_set* is `True` then attempting to set attributes that don’t exist on the spec object will raise an `AttributeError`.

> If a class is used as a spec then the return value of the mock (the instance of the class) will have the same spec. You can use a class as the spec for an instance object by passing `instance=True`. The returned mock will only be callable if instances of the mock are callable.
>
> **`create_autospec()`** also takes arbitrary keyword arguments that are passed to the constructor of the created mock.

See Autospeccing for examples of how to use auto-speccing with **`create_autospec()`** and the *autospec* argument to **`patch()`**.

Changed in version 3.8: **`create_autospec()`** now returns an **`AsyncMock`** if the target is an async function.

ANY

unittest.mock.ANY

Sometimes you may need to make assertions about *some* of the arguments in a call to mock, but either not care about some of the arguments or want to pull them individually out of **`call_args`** and make more complex assertions on them.

To ignore certain arguments you can pass in objects that compare equal to *everything*. Calls to **`assert_called_with()`** and **`assert_called_once_with()`** will then succeed no matter what was passed in.

```
>>> mock = Mock(return_value=None)
>>> mock('foo', bar=object())
>>> mock.assert_called_once_with('foo', bar=ANY)
```

`ANY` can also be used in comparisons with call lists like **`mock_calls`**:

```
>>> m = MagicMock(return_value=None)
>>> m(1)
>>> m(1, 2)
>>> m(object())
>>> m.mock_calls == [call(1), call(1, 2), ANY]
True
```

`ANY` is not limited to comparisons with call objects and so can also be used in test assertions:

```
class TestStringMethods(unittest.TestCase):

    def test_split(self):
        s = 'hello world'
        self.assertEqual(s.split(), ['hello', ANY])
```

FILTER_DIR

unittest.mock.FILTER_DIR

`FILTER_DIR` is a module level variable that controls the way mock objects respond to `dir()`. The default is `True`, which uses the filtering described below, to only show useful members. If you dislike this filtering, or need to switch it off for diagnostic purposes, then set `mock.FILTER_DIR = False`.

With filtering on, `dir(some_mock)` shows only useful attributes and will include any dynamically created attributes that wouldn’t normally be shown. If the mock was created with a *spec* (or *autospec* of course) then all the attributes from the original are shown, even if they haven’t been accessed yet:

```
>>> dir(Mock())
['assert_any_call',
 'assert_called',
 'assert_called_once',
 'assert_called_once_with',
 'assert_called_with',
 'assert_has_calls',
 'assert_not_called',
 'attach_mock',
 ...
```

```
>>> from urllib import request
>>> dir(Mock(spec=request))
['AbstractBasicAuthHandler',
 'AbstractDigestAuthHandler',
 'AbstractHTTPHandler',
 'BaseHandler',
 ...
```

Many of the not-very-useful (private to `Mock` rather than the thing being mocked) underscore and double underscore prefixed attributes have been filtered from the result of calling `dir()` on a `Mock`. If you dislike this behaviour you can switch it off by setting the module level switch `FILTER_DIR`:

```
>>> from unittest import mock
>>> mock.FILTER_DIR = False
>>> dir(mock.Mock())
['_NonCallableMock__get_return_value',
 '_NonCallableMock__get_side_effect',
 '_NonCallableMock__return_value_doc',
 '_NonCallableMock__set_return_value',
 '_NonCallableMock__set_side_effect',
 '__call__',
 '__class__',
 ...
```

Alternatively you can just use `vars(my_mock)` (instance members) and `dir(type(my_mock))` (type members) to bypass the filtering irrespective of `FILTER_DIR`.

mock_open

unittest.mock.mock_open(*mock=None*, *read_data=None*)

A helper function to create a mock to replace the use of `open()`. It works for `open()` called directly or used as a context manager.

The *mock* argument is the mock object to configure. If `None` (the default) then a `MagicMock` will be created for you, with the API limited to methods or attributes available on standard file handles.

read_data is a string for the **`read()`**, **`readline()`**, and **`readlines()`** methods of the file handle to return. Calls to those methods will take data from *read_data* until it is depleted. The mock of these methods is pretty simplistic: every time the *mock* is called, the *read_data* is rewound to the start. If you need more control over the data that you are feeding to the tested code you will need to customize this mock for yourself. When that is insufficient, one of the in-memory filesystem packages on PyPI [https://pypi.org] can offer a realistic filesystem for testing.

Changed in version 3.4: Added **`readline()`** and **`readlines()`** support. The mock of **`read()`** changed to consume *read_data* rather than returning it on each call.

Changed in version 3.5: read_data is now reset on each call to the *mock*.

Changed in version 3.8: Added **`__iter__()`** to implementation so that iteration (such as in for loops) correctly consumes *read_data*.

Using **`open()`** as a context manager is a great way to ensure your file handles are closed properly and is becoming common:

```
with open('/some/path', 'w') as f:
    f.write('something')
```

The issue is that even if you mock out the call to **`open()`** it is the *returned object* that is used as a context manager (and has **`__enter__()`** and **`__exit__()`** called).

Mocking context managers with a `MagicMock` is common enough and fiddly enough that a helper function is useful.

```
>>> m = mock_open()
>>> with patch('__main__.open', m):
...     with open('foo', 'w') as h:
...         h.write('some stuff')
...
>>> m.mock_calls
[call('foo', 'w'),
 call().__enter__(),
 call().write('some stuff'),
 call().__exit__(None, None, None)]
>>> m.assert_called_once_with('foo', 'w')
>>> handle = m()
>>> handle.write.assert_called_once_with('some stuff')
```

And for reading files:

```
>>> with patch('__main__.open', mock_open(read_data='bibble'))
as m:
...     with open('foo') as h:
...         result = h.read()
...
>>> m.assert_called_once_with('foo')
>>> assert result == 'bibble'
```

Autospeccing

Autospeccing is based on the existing **`spec`** feature of mock. It limits the api of mocks to the api of an original object (the spec), but it is recursive (implemented lazily) so that attributes of mocks only have the same api as the attributes of the spec. In addition mocked functions / methods have the same call signature as the original so they raise a `TypeError` if they are called incorrectly.

Before I explain how auto-speccing works, here’s why it is needed.

`Mock` is a very powerful and flexible object, but it suffers from a flaw which is general to mocking. If you refactor some of your code, rename members and so on, any tests for code that is still using the *old api* but uses mocks instead of the real objects will still pass. This means your tests can all pass even though your code is broken.

Changed in version 3.5: Before 3.5, tests with a typo in the word assert would silently pass when they should raise an error. You can still achieve this behavior by passing `unsafe=True` to Mock.

Note that this is another reason why you need integration tests as well as unit tests. Testing everything in isolation is all fine and dandy, but if you don’t test how your units are “wired together” there is still lots of room for bugs that tests might have caught.

`unittest.mock` already provides a feature to help with this, called speccing. If you use a class or instance as the **`spec`** for a mock then you can only access attributes on the mock that exist on the real class:

```
>>> from urllib import request
>>> mock = Mock(spec=request.Request)
>>> mock.assret_called_with  # Intentional typo!
Traceback (most recent call last):
 ...
AttributeError: Mock object has no attribute
'assret_called_with'
```

The spec only applies to the mock itself, so we still have the same issue with any methods on the mock:

```
>>> mock.header_items()
<mock.Mock object at 0x...>
>>> mock.header_items.assret_called_with()  # Intentional typo!
```

Auto-speccing solves this problem. You can either pass `autospec=True` to **`patch()`** / **`patch.object()`** or use the **`create_autospec()`** function to create a mock with a spec. If you use the `autospec=True` argument to **`patch()`** then the object that is being replaced will be used as the spec object. Because the speccing is done “lazily” (the spec is created as attributes on the mock are accessed) you can use it with very complex or deeply nested objects (like modules that import modules that import modules) without a big performance hit.

Here’s an example of it in use:

```
>>> from urllib import request
>>> patcher = patch('__main__.request', autospec=True)
>>> mock_request = patcher.start()
>>> request is mock_request
True
>>> mock_request.Request
<MagicMock name='request.Request' spec='Request' id='...'>
```

You can see that **`request.Request`** has a spec. **`request.Request`** takes two arguments in the constructor (one of which is *self*). Here’s what happens if we try to call it incorrectly:

```
>>> req = request.Request()
Traceback (most recent call last):
 ...
TypeError: <lambda>() takes at least 2 arguments (1 given)
```

The spec also applies to instantiated classes (i.e. the return value of specced mocks):

```
>>> req = request.Request('foo')
>>> req
<NonCallableMagicMock name='request.Request()' spec='Request'
id='...'>
```

`Request` objects are not callable, so the return value of instantiating our mocked out `request.Request` is a non-callable mock. With the spec in place any typos in our asserts will raise the correct error:

```
>>> req.add_header('spam', 'eggs')
<MagicMock name='request.Request().add_header()' id='...'>
>>> req.add_header.assret_called_with  # Intentional typo!
Traceback (most recent call last):
 ...
AttributeError: Mock object has no attribute
'assret_called_with'
>>> req.add_header.assert_called_with('spam', 'eggs')
```

In many cases you will just be able to add `autospec=True` to your existing `patch()` calls and then be protected against bugs due to typos and api changes.

As well as using *autospec* through `patch()` there is a `create_autospec()` for creating autospecced mocks directly:

```
>>> from urllib import request
>>> mock_request = create_autospec(request)
>>> mock_request.Request('foo', 'bar')
<NonCallableMagicMock name='mock.Request()' spec='Request'
id='...'>
```

This isn’t without caveats and limitations however, which is why it is not the default behaviour. In order to know what attributes are available on the spec object, autospec has to introspect (access attributes) the spec. As you traverse attributes on the mock a corresponding traversal of the original object is happening under the hood. If any of your specced objects have properties or descriptors that can trigger code execution then you may not be able to use autospec. On the other hand it is much better to design your objects so that introspection is safe [[4]].

A more serious problem is that it is common for instance attributes to be created in the `__init__()` method and not to exist on the class at all. *autospec* can’t know about any dynamically created attributes and restricts the api to visible attributes.

```
>>> class Something:
...   def __init__(self):
...     self.a = 33
...
>>> with patch('__main__.Something', autospec=True):
...   thing = Something()
...   thing.a
...
Traceback (most recent call last):
  ...
AttributeError: Mock object has no attribute 'a'
```

There are a few different ways of resolving this problem. The easiest, but not necessarily the least annoying, way is to simply set the required attributes on the mock after creation. Just because *autospec* doesn’t allow you to fetch attributes that don’t exist on the spec it doesn’t prevent you setting them:

```
>>> with patch('__main__.Something', autospec=True):
...   thing = Something()
...   thing.a = 33
...
```

There is a more aggressive version of both *spec* and *autospec* that *does* prevent you setting non-existent attributes. This is useful if you want to ensure your code only *sets* valid attributes too, but obviously it prevents this particular scenario:

```
>>> with patch('__main__.Something', autospec=True,
spec_set=True):
...   thing = Something()
...   thing.a = 33
```

```
...
Traceback (most recent call last):
 ...
AttributeError: Mock object has no attribute 'a'
```

Probably the best way of solving the problem is to add class attributes as default values for instance members initialised in `__init__()`. Note that if you are only setting default attributes in **`__init__()`** then providing them via class attributes (shared between instances of course) is faster too. e.g.

```
class Something:
    a = 33
```

This brings up another issue. It is relatively common to provide a default value of `None` for members that will later be an object of a different type. `None` would be useless as a spec because it wouldn’t let you access *any* attributes or methods on it. As `None` is *never* going to be useful as a spec, and probably indicates a member that will normally of some other type, autospec doesn’t use a spec for members that are set to `None`. These will just be ordinary mocks (well - MagicMocks):

```
>>> class Something:
...     member = None
...
>>> mock = create_autospec(Something)
>>> mock.member.foo.bar.baz()
<MagicMock name='mock.member.foo.bar.baz()' id='...'>
```

If modifying your production classes to add defaults isn’t to your liking then there are more options. One of these is simply to use an instance as the spec rather than the class. The other is to create a subclass of the production class and add the defaults to the subclass without affecting the production class. Both of these require you to use an alternative object as the spec. Thankfully `patch()` supports this - you can simply pass the alternative object as the *autospec* argument:

```
>>> class Something:
...   def __init__(self):
...     self.a = 33
...
>>> class SomethingForTest(Something):
...   a = 33
...
>>> p = patch('__main__.Something', autospec=SomethingForTest)
>>> mock = p.start()
>>> mock.a
<NonCallableMagicMock name='Something.a' spec='int' id='...'>
```

[4]

This only applies to classes or already instantiated objects. Calling a mocked class to create a mock instance *does not* create a real instance. It is only attribute lookups - along with calls to `dir()` - that are done.

Sealing mocks

unittest.mock.seal(*mock*)

Seal will disable the automatic creation of mocks when accessing an attribute of the mock being sealed or any of its attributes that are already mocks recursively.

If a mock instance with a name or a spec is assigned to an attribute it won’t be considered in the sealing chain. This allows one to prevent seal from fixing part of the mock object.

```
>>> mock = Mock()
>>> mock.submock.attribute1 = 2
>>> mock.not_submock = mock.Mock(name="sample_name")
>>> seal(mock)
>>> mock.new_attribute  # This will raise AttributeError.
>>> mock.submock.attribute2  # This will raise
AttributeError.
>>> mock.not_submock.attribute2  # This won't raise.
```

Added in version 3.7.

Order of precedence of `side_effect`, `return_value` and *wraps*

The order of their precedence is:

1. `side_effect`
2. `return_value`
3. *wraps*

If all three are set, mock will return the value from `side_effect`, ignoring `return_value` and the wrapped object altogether. If any two are set, the one with the higher precedence will return the value. Regardless of the order of which was set first, the order of precedence remains unchanged.

```
>>> from unittest.mock import Mock
>>> class Order:
...     @staticmethod
...     def get_value():
...         return "third"
...
>>> order_mock = Mock(spec=Order, wraps=Order)
>>> order_mock.get_value.side_effect = ["first"]
>>> order_mock.get_value.return_value = "second"
>>> order_mock.get_value()
'first'
```

As `None` is the default value of `side_effect`, if you reassign its value back to `None`, the order of precedence will be checked between `return_value` and the wrapped object, ignoring `side_effect`.

```
>>> order_mock.get_value.side_effect = None
>>> order_mock.get_value()
'second'
```

If the value being returned by `side_effect` is `DEFAULT`, it is ignored and the order of precedence moves to the successor to obtain the value to return.

```
>>> from unittest.mock import DEFAULT
>>> order_mock.get_value.side_effect = [DEFAULT]
>>> order_mock.get_value()
'second'
```

When `Mock` wraps an object, the default value of `return_value` will be `DEFAULT`.

```
>>> order_mock = Mock(spec=Order, wraps=Order)
>>> order_mock.return_value
sentinel.DEFAULT
>>> order_mock.get_value.return_value
sentinel.DEFAULT
```

The order of precedence will ignore this value and it will move to the last successor which is the wrapped object.

As the real call is being made to the wrapped object, creating an instance of this mock will return the real instance of the class. The positional arguments, if any, required by the wrapped object must be passed.

```
>>> order_mock_instance = order_mock()
>>> isinstance(order_mock_instance, Order)
True
>>> order_mock_instance.get_value()
'third'
```

```
>>> order_mock.get_value.return_value = DEFAULT
>>> order_mock.get_value()
'third'
```

```
>>> order_mock.get_value.return_value = "second"
>>> order_mock.get_value()
'second'
```

But if you assign `None` to it, this will not be ignored as it is an explicit assignment. So, the order of precedence will not move to the wrapped object.

```
>>> order_mock.get_value.return_value = None
>>> order_mock.get_value() is None
True
```

Even if you set all three at once when initializing the mock, the order of precedence remains the same:

```
>>> order_mock = Mock(spec=Order, wraps=Order,
...                   **{"get_value.side_effect": ["first"],
...                      "get_value.return_value": "second"}
...                   )
...
>>> order_mock.get_value()
'first'
>>> order_mock.get_value.side_effect = None
>>> order_mock.get_value()
'second'
>>> order_mock.get_value.return_value = DEFAULT
>>> order_mock.get_value()
'third'
```

If `side_effect` is exhausted, the order of precedence will not cause a value to be obtained from the successors. Instead, `StopIteration` exception is raised.

```
>>> order_mock = Mock(spec=Order, wraps=Order)
>>> order_mock.get_value.side_effect = ["first side effect
value",
...                                     "another side effect
value"]
>>> order_mock.get_value.return_value = "second"
>>> order_mock.get_value()
'first side effect value'
>>> order_mock.get_value()
'another side effect value'

>>> order_mock.get_value()
Traceback (most recent call last):
 ...
StopIteration
```

`unittest.mock` — getting started

Added in version 3.3.

Using Mock

Mock Patching Methods

Common uses for `Mock` objects include:

- Patching methods
- Recording method calls on objects

You might want to replace a method on an object to check that it is called with the correct arguments by another part of the system:

```
>>> real = SomeClass()
>>> real.method = MagicMock(name='method')
>>> real.method(3, 4, 5, key='value')
<MagicMock name='method()' id='...'>
```

Once our mock has been used (`real.method` in this example) it has methods and attributes that allow you to make assertions about how it has been used.

Note

In most of these examples the `Mock` and `MagicMock` classes are interchangeable. As the `MagicMock` is the more capable class it makes a sensible one to use by default.

Once the mock has been called its `called` attribute is set to `True`. More importantly we can use the `assert_called_with()` or `assert_called_once_with()` method to check that it was called with the correct arguments.

This example tests that calling `ProductionClass().method` results in a call to the `something` method:

```
>>> class ProductionClass:
...     def method(self):
...         self.something(1, 2, 3)
...     def something(self, a, b, c):
...         pass
...
>>> real = ProductionClass()
>>> real.something = MagicMock()
>>> real.method()
>>> real.something.assert_called_once_with(1, 2, 3)
```

Mock for Method Calls on an Object

In the last example we patched a method directly on an object to check that it was called correctly. Another common use case is to pass an object into a method (or some part of the system under test) and then check that it is used in the correct way.

The simple `ProductionClass` below has a `closer` method. If it is called with an object then it calls `close` on it.

```
>>> class ProductionClass:
...     def closer(self, something):
...         something.close()
...
```

So to test it we need to pass in an object with a `close` method and check that it was called correctly.

```
>>> real = ProductionClass()
>>> mock = Mock()
>>> real.closer(mock)
>>> mock.close.assert_called_with()
```

We don’t have to do any work to provide the ‘close’ method on our mock. Accessing close creates it. So, if ‘close’ hasn’t already been called then accessing it in the test will create it, but **`assert_called_with()`** will raise a failure exception.

Mocking Classes

A common use case is to mock out classes instantiated by your code under test. When you patch a class, then that class is replaced with a mock. Instances are created by *calling the class*. This means you access the “mock instance” by looking at the return value of the mocked class.

In the example below we have a function `some_function` that instantiates `Foo` and calls a method on it. The call to **`patch()`** replaces the class `Foo` with a mock. The `Foo` instance is the result of calling the mock, so it is configured by modifying the mock **`return_value`**.

```
>>> def some_function():
...     instance = module.Foo()
...     return instance.method()
...
>>> with patch('module.Foo') as mock:
...     instance = mock.return_value
...     instance.method.return_value = 'the result'
...     result = some_function()
...     assert result == 'the result'
```

Naming your mocks

It can be useful to give your mocks a name. The name is shown in the repr of the mock and can be helpful when the mock appears in test failure messages. The name is also propagated to attributes or methods of the mock:

```
>>> mock = MagicMock(name='foo')
>>> mock
<MagicMock name='foo' id='...'>
>>> mock.method
<MagicMock name='foo.method' id='...'>
```

Tracking all Calls

Often you want to track more than a single call to a method. The `mock_calls` attribute records all calls to child attributes of the mock - and also to their children.

```
>>> mock = MagicMock()
>>> mock.method()
<MagicMock name='mock.method()' id='...'>
>>> mock.attribute.method(10, x=53)
<MagicMock name='mock.attribute.method()' id='...'>
>>> mock.mock_calls
[call.method(), call.attribute.method(10, x=53)]
```

If you make an assertion about `mock_calls` and any unexpected methods have been called, then the assertion will fail. This is useful because as well as asserting that the calls you expected have been made, you are also checking that they were made in the right order and with no additional calls:

You use the `call` object to construct lists for comparing with `mock_calls`:

```
>>> expected = [call.method(), call.attribute.method(10, x=53)]
>>> mock.mock_calls == expected
True
```

However, parameters to calls that return mocks are not recorded, which means it is not possible to track nested calls where the parameters used to create ancestors are important:

```
>>> m = Mock()
>>> m.factory(important=True).deliver()
<Mock name='mock.factory().deliver()' id='...'>
>>> m.mock_calls[-1] == call.factory(important=False).deliver()
True
```

Setting Return Values and Attributes

Setting the return values on a mock object is trivially easy:

```
>>> mock = Mock()
>>> mock.return_value = 3
>>> mock()
3
```

Of course you can do the same for methods on the mock:

```
>>> mock = Mock()
>>> mock.method.return_value = 3
>>> mock.method()
3
```

The return value can also be set in the constructor:

```
>>> mock = Mock(return_value=3)
>>> mock()
3
```

If you need an attribute setting on your mock, just do it:

```
>>> mock = Mock()
>>> mock.x = 3
>>> mock.x
3
```

Sometimes you want to mock up a more complex situation, like for example `mock.connection.cursor().execute("SELECT 1")`. If we wanted this call to return a list, then we have to configure the result of the nested call.

We can use `call` to construct the set of calls in a “chained call” like this for easy assertion afterwards:

```
>>> mock = Mock()
>>> cursor = mock.connection.cursor.return_value
>>> cursor.execute.return_value = ['foo']
>>> mock.connection.cursor().execute("SELECT 1")
['foo']
>>> expected = call.connection.cursor().execute("SELECT
1").call_list()
>>> mock.mock_calls
[call.connection.cursor(),
call.connection.cursor().execute('SELECT 1')]
>>> mock.mock_calls == expected
True
```

It is the call to `.call_list()` that turns our call object into a list of calls representing the chained calls.

Raising exceptions with mocks

A useful attribute is `side_effect`. If you set this to an exception class or instance then the exception will be raised when the mock is called.

```
>>> mock = Mock(side_effect=Exception('Boom!'))
>>> mock()
Traceback (most recent call last):
  ...
Exception: Boom!
```

Side effect functions and iterables

`side_effect` can also be set to a function or an iterable. The use case for `side_effect` as an iterable is where your mock is going to be called several times, and you want each call to return a different value. When you set `side_effect` to an iterable every call to the mock returns the next value from the iterable:

```
>>> mock = MagicMock(side_effect=[4, 5, 6])
>>> mock()
4
>>> mock()
5
>>> mock()
6
```

For more advanced use cases, like dynamically varying the return values depending on what the mock is called with, `side_effect` can be a function. The function will be called with the same arguments as the mock. Whatever the function returns is what the call returns:

```
>>> vals = {(1, 2): 1, (2, 3): 2}
>>> def side_effect(*args):
...     return vals[args]
...
>>> mock = MagicMock(side_effect=side_effect)
>>> mock(1, 2)
1
>>> mock(2, 3)
2
```

Mocking asynchronous iterators

Since Python 3.8, `AsyncMock` and `MagicMock` have support to mock Asynchronous Iterators through `__aiter__`. The **`return_value`** attribute of `__aiter__` can be used to set the return values to be used for iteration.

```
>>> mock = MagicMock()  # AsyncMock also works here
>>> mock.__aiter__.return_value = [1, 2, 3]
>>> async def main():
...     return [i async for i in mock]
...
>>> asyncio.run(main())
[1, 2, 3]
```

Mocking asynchronous context manager

Since Python 3.8, `AsyncMock` and `MagicMock` have support to mock Asynchronous Context Managers through `__aenter__` and `__aexit__`. By default, `__aenter__` and `__aexit__` are `AsyncMock` instances that return an async function.

```
>>> class AsyncContextManager:
...     async def __aenter__(self):
...         return self
...     async def __aexit__(self, exc_type, exc, tb):
...         pass
...
>>> mock_instance = MagicMock(AsyncContextManager())  #
AsyncMock also works here
>>> async def main():
...     async with mock_instance as result:
...         pass
...
>>> asyncio.run(main())
>>> mock_instance.__aenter__.assert_awaited_once()
>>> mock_instance.__aexit__.assert_awaited_once()
```

Creating a Mock from an Existing Object

One problem with over use of mocking is that it couples your tests to the implementation of your mocks rather than your real code. Suppose you have a class that implements `some_method`. In a test for another class, you provide a mock of this object that *also* provides `some_method`. If later you refactor the first class, so that it no longer has `some_method` -

then your tests will continue to pass even though your code is now broken!

`Mock` allows you to provide an object as a specification for the mock, using the *spec* keyword argument. Accessing methods / attributes on the mock that don't exist on your specification object will immediately raise an attribute error. If you change the implementation of your specification, then tests that use that class will start failing immediately without you having to instantiate the class in those tests.

```
>>> mock = Mock(spec=SomeClass)
>>> mock.old_method()
Traceback (most recent call last):
   ...
AttributeError: Mock object has no attribute 'old_method'. Did
you mean: 'class_method'?
```

Using a specification also enables a smarter matching of calls made to the mock, regardless of whether some parameters were passed as positional or named arguments:

```
>>> def f(a, b, c): pass
...
>>> mock = Mock(spec=f)
>>> mock(1, 2, 3)
<Mock name='mock()' id='140161580456576'>
>>> mock.assert_called_with(a=1, b=2, c=3)
```

If you want this smarter matching to also work with method calls on the mock, you can use auto-speccing.

If you want a stronger form of specification that prevents the setting of arbitrary attributes as well as the getting of them then you can use *spec_set* instead of *spec*.

Using side_effect to return per file content

`mock_open()` is used to patch `open()` method. `side_effect` can be used to return a new Mock object per call. This can be used to return different contents per file stored in a dictionary:

```
DEFAULT = "default"
data_dict = {"file1": "data1",
             "file2": "data2"}

def open_side_effect(name):
    return mock_open(read_data=data_dict.get(name, DEFAULT))()

with patch("builtins.open", side_effect=open_side_effect):
    with open("file1") as file1:
        assert file1.read() == "data1"

    with open("file2") as file2:
        assert file2.read() == "data2"

    with open("file3") as file2:
        assert file2.read() == "default"
```

Patch Decorators

Note

With `patch()` it matters that you patch objects in the namespace where they are looked up. This is normally straightforward, but for a quick guide read where to patch.

A common need in tests is to patch a class attribute or a module attribute, for example patching a builtin or patching a class in a module to test that it is instantiated. Modules and classes are effectively global, so patching on them has to be undone after the test or the patch will persist into other tests and cause hard to diagnose problems.

mock provides three convenient decorators for this: `patch()`, `patch.object()` and `patch.dict()`. `patch` takes a single string, of the form `package.module.Class.attribute` to specify the attribute you are patching. It also optionally takes a value that you want the attribute (or class or whatever) to be replaced with. ‘patch.object’ takes an object and the name of the attribute you would like patched, plus optionally the value to patch it with.

`patch.object`:

```
>>> original = SomeClass.attribute
>>> @patch.object(SomeClass, 'attribute', sentinel.attribute)
... def test():
...     assert SomeClass.attribute == sentinel.attribute
...
>>> test()
>>> assert SomeClass.attribute == original

>>> @patch('package.module.attribute', sentinel.attribute)
... def test():
...     from package.module import attribute
...     assert attribute is sentinel.attribute
...
>>> test()
```

If you are patching a module (including `builtins`) then use `patch()` instead of `patch.object()`:

```
>>> mock = MagicMock(return_value=sentinel.file_handle)
>>> with patch('builtins.open', mock):
...     handle = open('filename', 'r')
...
>>> mock.assert_called_with('filename', 'r')
>>> assert handle == sentinel.file_handle, "incorrect file
handle returned"
```

The module name can be ‘dotted’, in the form `package.module` if needed:

```
>>> @patch('package.module.ClassName.attribute',
sentinel.attribute)
... def test():
...     from package.module import ClassName
...     assert ClassName.attribute == sentinel.attribute
...
>>> test()
```

A nice pattern is to actually decorate test methods themselves:

```
>>> class MyTest(unittest.TestCase):
...     @patch.object(SomeClass, 'attribute',
sentinel.attribute)
...     def test_something(self):
...         self.assertEqual(SomeClass.attribute,
sentinel.attribute)
...
>>> original = SomeClass.attribute
>>> MyTest('test_something').test_something()
>>> assert SomeClass.attribute == original
```

If you want to patch with a Mock, you can use **`patch()`** with only one argument (or **`patch.object()`** with two arguments). The mock will be created for you and passed into the test function / method:

```
>>> class MyTest(unittest.TestCase):
...     @patch.object(SomeClass, 'static_method')
...     def test_something(self, mock_method):
...         SomeClass.static_method()
...         mock_method.assert_called_with()
...
>>> MyTest('test_something').test_something()
```

You can stack up multiple patch decorators using this pattern:

```
>>> class MyTest(unittest.TestCase):
...     @patch('package.module.ClassName1')
...     @patch('package.module.ClassName2')
...     def test_something(self, MockClass2, MockClass1):
...         self.assertIs(package.module.ClassName1,
MockClass1)
...         self.assertIs(package.module.ClassName2,
MockClass2)
...
>>> MyTest('test_something').test_something()
```

When you nest patch decorators the mocks are passed in to the decorated function in the same order they applied (the normal *Python* order that decorators are applied). This means from the bottom up, so in the example above the mock for `test_module.ClassName2` is passed in first.

There is also **`patch.dict()`** for setting values in a dictionary just during a scope and restoring the dictionary to its original state when the test ends:

```
>>> foo = {'key': 'value'}
>>> original = foo.copy()
>>> with patch.dict(foo, {'newkey': 'newvalue'}, clear=True):
...     assert foo == {'newkey': 'newvalue'}
...
>>> assert foo == original
```

`patch`, `patch.object` and `patch.dict` can all be used as context managers.

Where you use **`patch()`** to create a mock for you, you can get a reference to the mock using the “as” form of the with statement:

```
>>> class ProductionClass:
...     def method(self):
...         pass
...
>>> with patch.object(ProductionClass, 'method') as
mock_method:
...     mock_method.return_value = None
...     real = ProductionClass()
...     real.method(1, 2, 3)
...
>>> mock_method.assert_called_with(1, 2, 3)
```

As an alternative `patch`, `patch.object` and `patch.dict` can be used as class decorators. When used in this way it is the same as applying the decorator individually to every method whose name starts with “test”.

Further Examples

Here are some more examples for some slightly more advanced scenarios.

Mocking chained calls

Mocking chained calls is actually straightforward with mock once you understand the **`return_value`** attribute. When a mock is called for the first time, or you fetch its `return_value` before it has been called, a new **`Mock`** is created.

This means that you can see how the object returned from a call to a mocked object has been used by interrogating the `return_value` mock:

```
>>> mock = Mock()
>>> mock().foo(a=2, b=3)
<Mock name='mock().foo()' id='...'>
>>> mock.return_value.foo.assert_called_with(a=2, b=3)
```

From here it is a simple step to configure and then make assertions about chained calls. Of course another alternative is writing your code in a more testable way in the first place…

So, suppose we have some code that looks a little bit like this:

```
>>> class Something:
...     def __init__(self):
...         self.backend = BackendProvider()
...     def method(self):
...         response =
self.backend.get_endpoint('foobar').create_call('spam',
'eggs').start_call()
...         # more code
```

Assuming that `BackendProvider` is already well tested, how do we test `method()`? Specifically, we want to test that the code section `# more code` uses the response object in the correct way.

As this chain of calls is made from an instance attribute we can monkey patch the `backend` attribute on a `Something` instance. In this particular case we are only interested in the return value from the final call to `start_call` so we don't have much configuration to do. Let's assume the object it returns is 'file-like', so we'll ensure that our response object uses the builtin **`open()`** as its `spec`.

To do this we create a mock instance as our mock backend and create a mock response object for it. To set the response as the return value for that final `start_call` we could do this:

```
mock_backend.get_endpoint.return_value.create_call.return_value
.start_call.return_value = mock_response
```

We can do that in a slightly nicer way using the **`configure_mock()`** method to directly set the return value for us:

```
>>> something = Something()
>>> mock_response = Mock(spec=open)
>>> mock_backend = Mock()
>>> config =
{'get_endpoint.return_value.create_call.return_value.start_call
.return_value': mock_response}
>>> mock_backend.configure_mock(**config)
```

With these we monkey patch the “mock backend” in place and can make the real call:

```
>>> something.backend = mock_backend
>>> something.method()
```

Using `mock_calls` we can check the chained call with a single assert. A chained call is several calls in one line of code, so there will be several entries in `mock_calls`. We can use `call.call_list()` to create this list of calls for us:

```
>>> chained = call.get_endpoint('foobar').create_call('spam',
'eggs').start_call()
>>> call_list = chained.call_list()
>>> assert mock_backend.mock_calls == call_list
```

Partial mocking

For some tests, you may want to mock out a call to `datetime.date.today()` to return a known date, but don’t want to prevent the code under test from creating new date objects. Unfortunately `datetime.date` is written in C, so you cannot just monkey-patch out the static `datetime.date.today()` method.

Instead, you can effectively wrap the date class with a mock, while passing through calls to the constructor to the real class (and returning real instances).

The `patch decorator` is used here to mock out the `date` class in the module under test. The `side_effect` attribute on the mock date class is then set to a lambda function that returns a real date. When the mock date class is called a real date will be constructed and returned by `side_effect`.

```
>>> from datetime import date
>>> with patch('mymodule.date') as mock_date:
...     mock_date.today.return_value = date(2010, 10, 8)
...     mock_date.side_effect = lambda *args, **kw: date(*args,
**kw)
...
...     assert mymodule.date.today() == date(2010, 10, 8)
...     assert mymodule.date(2009, 6, 8) == date(2009, 6, 8)
```

Note that we don’t patch **`datetime.date`** globally, we patch `date` in the module that *uses* it. See where to patch.

When `date.today()` is called a known date is returned, but calls to the `date(...)` constructor still return normal dates. Without this you can find yourself having to calculate an expected result using exactly the same algorithm as the code under test, which is a classic testing anti-pattern.

Calls to the date constructor are recorded in the `mock_date` attributes (`call_count` and friends) which may also be useful for your tests.

An alternative way of dealing with mocking dates, or other builtin classes, is discussed in this blog entry [https://williambert.online/2011/07/how-to-unit-testing-in-django-with-mocking-and-patching/].

Mocking a Generator Method

A Python generator is a function or method that uses the **`yield`** statement to return a series of values when iterated over [[1]].

A generator method / function is called to return the generator object. It is the generator object that is then iterated over. The protocol method for iteration is **`__iter__()`**, so we can mock this using a **`MagicMock`**.

Here’s an example class with an “iter” method implemented as a generator:

```
>>> class Foo:
...     def iter(self):
...         for i in [1, 2, 3]:
...             yield i
...
>>> foo = Foo()
>>> list(foo.iter())
[1, 2, 3]
```

How would we mock this class, and in particular its “iter” method?

To configure the values returned from the iteration (implicit in the call to **`list`**), we need to configure the object returned by the call to `foo.iter()`.

```
>>> mock_foo = MagicMock()
>>> mock_foo.iter.return_value = iter([1, 2, 3])
>>> list(mock_foo.iter())
[1, 2, 3]
```

[1]

There are also generator expressions and more advanced uses [http://www.dabeaz.com/coroutines/index.html] of generators, but we aren’t concerned about them here. A very good introduction to generators and how powerful they are is: Generator Tricks for Systems Programmers [http://www.dabeaz.com/generators/].

Applying the same patch to every test method

If you want several patches in place for multiple test methods the obvious way is to apply the patch decorators to every method. This can feel like unnecessary repetition. Instead, you can use **`patch()`** (in all its various forms) as a class decorator. This applies the patches to all test methods on the class. A test method is identified by methods whose names start with `test`:

```
>>> @patch('mymodule.SomeClass')
... class MyTest(unittest.TestCase):
...
...     def test_one(self, MockSomeClass):
...         self.assertIs(mymodule.SomeClass, MockSomeClass)
...
...     def test_two(self, MockSomeClass):
...         self.assertIs(mymodule.SomeClass, MockSomeClass)
...
...     def not_a_test(self):
...         return 'something'
...
>>> MyTest('test_one').test_one()
>>> MyTest('test_two').test_two()
>>> MyTest('test_two').not_a_test()
'something'
```

An alternative way of managing patches is to use the patch methods: start and stop. These allow you to move the patching into your `setUp` and `tearDown` methods.

```
>>> class MyTest(unittest.TestCase):
...     def setUp(self):
...         self.patcher = patch('mymodule.foo')
...         self.mock_foo = self.patcher.start()
...
...     def test_foo(self):
...         self.assertIs(mymodule.foo, self.mock_foo)
...
...     def tearDown(self):
...         self.patcher.stop()
...
>>> MyTest('test_foo').run()
```

If you use this technique you must ensure that the patching is “undone” by calling `stop`. This can be fiddlier than you might think, because if an exception is raised in the setUp then tearDown is not called. `unittest.TestCase.addCleanup()` makes this easier:

```
>>> class MyTest(unittest.TestCase):
...     def setUp(self):
...         patcher = patch('mymodule.foo')
...         self.addCleanup(patcher.stop)
...         self.mock_foo = patcher.start()
...
...     def test_foo(self):
...         self.assertIs(mymodule.foo, self.mock_foo)
...
>>> MyTest('test_foo').run()
```

Mocking Unbound Methods

Sometimes a test needs to patch an *unbound method*, which means patching the method on the class rather than on the instance. In order to make assertions about which objects were calling this particular method, you need to pass `self` as the first argument. The issue is that you can't patch with a mock for this, because if you replace an unbound method with a mock it doesn't become a bound method when fetched from the instance, and so it doesn't get `self` passed in. The workaround is to patch the unbound method with a real function instead. The `patch()` decorator makes it so simple to patch out methods with a mock that having to create a real function becomes a nuisance.

If you pass `autospec=True` to patch then it does the patching with a *real* function object. This function object has the same signature as the one it is replacing, but delegates to a mock under the hood. You still get your mock auto-created in exactly the same way as before. What it means though, is that if you use it to patch out an unbound method on a class the mocked function will be turned into a bound method if it is fetched from an instance. It will have `self` passed in as the first argument, which is exactly what was needed:

```
>>> class Foo:
...   def foo(self):
...     pass
...
>>> with patch.object(Foo, 'foo', autospec=True) as mock_foo:
...   mock_foo.return_value = 'foo'
...   foo = Foo()
...   foo.foo()
...
'foo'
>>> mock_foo.assert_called_once_with(foo)
```

If we don’t use `autospec=True` then the unbound method is patched out with a Mock instance instead, and isn’t called with `self`.

Checking multiple calls with mock

mock has a nice API for making assertions about how your mock objects are used.

```
>>> mock = Mock()
>>> mock.foo_bar.return_value = None
>>> mock.foo_bar('baz', spam='eggs')
>>> mock.foo_bar.assert_called_with('baz', spam='eggs')
```

If your mock is only being called once you can use the `assert_called_once_with()` method that also asserts that the `call_count` is one.

```
>>> mock.foo_bar.assert_called_once_with('baz', spam='eggs')
>>> mock.foo_bar()
>>> mock.foo_bar.assert_called_once_with('baz', spam='eggs')
Traceback (most recent call last):
    ...
AssertionError: Expected 'foo_bar' to be called once. Called 2
times.
Calls: [call('baz', spam='eggs'), call()].
```

Both `assert_called_with` and `assert_called_once_with` make assertions about the *most recent* call. If your mock is going to be called several times, and you want to make assertions about *all* those calls you can use **`call_args_list`**:

```
>>> mock = Mock(return_value=None)
>>> mock(1, 2, 3)
>>> mock(4, 5, 6)
>>> mock()
>>> mock.call_args_list
[call(1, 2, 3), call(4, 5, 6), call()]
```

The **`call`** helper makes it easy to make assertions about these calls. You can build up a list of expected calls and compare it to `call_args_list`. This looks remarkably similar to the repr of the `call_args_list`:

```
>>> expected = [call(1, 2, 3), call(4, 5, 6), call()]
>>> mock.call_args_list == expected
True
```

Coping with mutable arguments

Another situation is rare, but can bite you, is when your mock is called with mutable arguments. `call_args` and `call_args_list` store *references* to the arguments. If the arguments are mutated by the code under test then you can no longer make assertions about what the values were when the mock was called.

Here’s some example code that shows the problem. Imagine the following functions defined in ‘mymodule’:

```
def frob(val):
    pass

def grob(val):
    "First frob and then clear val"
    frob(val)
    val.clear()
```

When we try to test that `grob` calls `frob` with the correct argument look what happens:

```
>>> with patch('mymodule.frob') as mock_frob:
...     val = {6}
...     mymodule.grob(val)
...
>>> val
set()
>>> mock_frob.assert_called_with({6})
Traceback (most recent call last):
    ...
AssertionError: Expected: (({6},), {})
Called with: ((set(),), {})
```

One possibility would be for mock to copy the arguments you pass in. This could then cause problems if you do assertions that rely on object identity for equality.

Here's one solution that uses the **`side_effect`** functionality. If you provide a `side_effect` function for a mock then `side_effect` will be called with the same args as the mock. This gives us an opportunity to copy the arguments and store them for later assertions. In this example I'm using *another* mock to store the arguments so that I can use the mock methods for doing the assertion. Again a helper function sets this up for me.

```
>>> from copy import deepcopy
>>> from unittest.mock import Mock, patch, DEFAULT
>>> def copy_call_args(mock):
...     new_mock = Mock()
...     def side_effect(*args, **kwargs):
...         args = deepcopy(args)
...         kwargs = deepcopy(kwargs)
...         new_mock(*args, **kwargs)
...         return DEFAULT
...     mock.side_effect = side_effect
...     return new_mock
...
>>> with patch('mymodule.frob') as mock_frob:
...     new_mock = copy_call_args(mock_frob)
...     val = {6}
...     mymodule.grob(val)
...
>>> new_mock.assert_called_with({6})
>>> new_mock.call_args
call({6})
```

`copy_call_args` is called with the mock that will be called. It returns a new mock that we do the assertion on. The `side_effect` function makes a copy of the args and calls our `new_mock` with the copy.

Note

If your mock is only going to be used once there is an easier way of checking arguments at the point they are called. You can simply do the checking inside a `side_effect` function.

```
>>> def side_effect(arg):
...     assert arg == {6}
...
>>> mock = Mock(side_effect=side_effect)
>>> mock({6})
>>> mock(set())
Traceback (most recent call last):
    ...
AssertionError
```

An alternative approach is to create a subclass of **`Mock`** or **`MagicMock`** that copies (using **`copy.deepcopy()`**) the arguments. Here’s an example implementation:

```
>>> from copy import deepcopy
>>> class CopyingMock(MagicMock):
...     def __call__(self, /, *args, **kwargs):
...         args = deepcopy(args)
...         kwargs = deepcopy(kwargs)
...         return super().__call__(*args, **kwargs)
...
>>> c = CopyingMock(return_value=None)
>>> arg = set()
>>> c(arg)
>>> arg.add(1)
>>> c.assert_called_with(set())
>>> c.assert_called_with(arg)
Traceback (most recent call last):
    ...
AssertionError: expected call not found.
Expected: mock({1})
Actual: mock(set())
>>> c.foo
<CopyingMock name='mock.foo' id='...'>
```

When you subclass `Mock` or `MagicMock` all dynamically created attributes, and the `return_value` will use your subclass automatically. That means all children of a `CopyingMock` will also have the type `CopyingMock`.

Nesting Patches

Using patch as a context manager is nice, but if you do multiple patches you can end up with nested with statements indenting further and further to the right:

```
>>> class MyTest(unittest.TestCase):
...
...     def test_foo(self):
...         with patch('mymodule.Foo') as mock_foo:
...             with patch('mymodule.Bar') as mock_bar:
...                 with patch('mymodule.Spam') as mock_spam:
...                     assert mymodule.Foo is mock_foo
...                     assert mymodule.Bar is mock_bar
...                     assert mymodule.Spam is mock_spam
```

```
...
>>> original = mymodule.Foo
>>> MyTest('test_foo').test_foo()
>>> assert mymodule.Foo is original
```

With unittest `cleanup` functions and the patch methods: start and stop we can achieve the same effect without the nested indentation. A simple helper method, `create_patch`, puts the patch in place and returns the created mock for us:

```
>>> class MyTest(unittest.TestCase):
...
...     def create_patch(self, name):
...         patcher = patch(name)
...         thing = patcher.start()
...         self.addCleanup(patcher.stop)
...         return thing
...
...     def test_foo(self):
...         mock_foo = self.create_patch('mymodule.Foo')
...         mock_bar = self.create_patch('mymodule.Bar')
...         mock_spam = self.create_patch('mymodule.Spam')
...
...         assert mymodule.Foo is mock_foo
...         assert mymodule.Bar is mock_bar
...         assert mymodule.Spam is mock_spam
...
>>> original = mymodule.Foo
>>> MyTest('test_foo').run()
>>> assert mymodule.Foo is original
```

Mocking a dictionary with MagicMock

You may want to mock a dictionary, or other container object, recording all access to it whilst having it still behave like a dictionary.

We can do this with **`MagicMock`**, which will behave like a dictionary, and using **`side_effect`** to delegate dictionary access to a real underlying dictionary that is under our control.

When the `__getitem__()` and `__setitem__()` methods of our `MagicMock` are called (normal dictionary access) then `side_effect` is called with the key (and in the case of `__setitem__` the value too). We can also control what is returned.

After the `MagicMock` has been used we can use attributes like `call_args_list` to assert about how the dictionary was used:

```
>>> my_dict = {'a': 1, 'b': 2, 'c': 3}
>>> def getitem(name):
...      return my_dict[name]
...
>>> def setitem(name, val):
...     my_dict[name] = val
...
>>> mock = MagicMock()
>>> mock.__getitem__.side_effect = getitem
>>> mock.__setitem__.side_effect = setitem
```

Note

An alternative to using `MagicMock` is to use `Mock` and *only* provide the magic methods you specifically want:

```
>>> mock = Mock()
>>> mock.__getitem__ = Mock(side_effect=getitem)
>>> mock.__setitem__ = Mock(side_effect=setitem)
```

A *third* option is to use `MagicMock` but passing in `dict` as the *spec* (or *spec_set*) argument so that the `MagicMock` created only has dictionary magic methods available:

```
>>> mock = MagicMock(spec_set=dict)
>>> mock.__getitem__.side_effect = getitem
>>> mock.__setitem__.side_effect = setitem
```

With these side effect functions in place, the `mock` will behave like a normal dictionary but recording the access. It even raises a `KeyError` if you try to access a key that doesn’t exist.

```
>>> mock['a']
1
>>> mock['c']
3
>>> mock['d']
Traceback (most recent call last):
    ...
KeyError: 'd'
>>> mock['b'] = 'fish'
>>> mock['d'] = 'eggs'
>>> mock['b']
'fish'
>>> mock['d']
'eggs'
```

After it has been used you can make assertions about the access using the normal mock methods and attributes:

```
>>> mock.__getitem__.call_args_list
[call('a'), call('c'), call('d'), call('b'), call('d')]
>>> mock.__setitem__.call_args_list
[call('b', 'fish'), call('d', 'eggs')]
>>> my_dict
{'a': 1, 'b': 'fish', 'c': 3, 'd': 'eggs'}
```

Mock subclasses and their attributes

There are various reasons why you might want to subclass `Mock`. One reason might be to add helper methods. Here’s a silly example:

```
>>> class MyMock(MagicMock):
...     def has_been_called(self):
...         return self.called
...
>>> mymock = MyMock(return_value=None)
>>> mymock
<MyMock id='...'>
>>> mymock.has_been_called()
False
>>> mymock()
>>> mymock.has_been_called()
True
```

The standard behaviour for `Mock` instances is that attributes and the return value mocks are of the same type as the mock they are accessed on. This ensures that `Mock` attributes are `Mocks` and `MagicMock` attributes are `MagicMocks` [[2]]. So if you're subclassing to add helper methods then they'll also be available on the attributes and return value mock of instances of your subclass.

```
>>> mymock.foo
<MyMock name='mock.foo' id='...'>
>>> mymock.foo.has_been_called()
False
>>> mymock.foo()
<MyMock name='mock.foo()' id='...'>
>>> mymock.foo.has_been_called()
True
```

Sometimes this is inconvenient. For example, one user [https://code.google.com/archive/p/mock/issues/105] is subclassing mock to created a Twisted adaptor [https://twisted.org/documents/11.0.0/api/twisted.python.components.html]. Having this applied to attributes too actually causes errors.

`Mock` (in all its flavours) uses a method called `_get_child_mock` to create these "sub-mocks" for attributes and return values. You can prevent your subclass being used for attributes by overriding this method. The signature is that it takes arbitrary keyword arguments (`**kwargs`) which are then passed onto the mock constructor:

```
>>> class Subclass(MagicMock):
...     def _get_child_mock(self, /, **kwargs):
...         return MagicMock(**kwargs)
...
>>> mymock = Subclass()
>>> mymock.foo
<MagicMock name='mock.foo' id='...'>
>>> assert isinstance(mymock, Subclass)
>>> assert not isinstance(mymock.foo, Subclass)
>>> assert not isinstance(mymock(), Subclass)
```

[2]

An exception to this rule are the non-callable mocks. Attributes use the callable variant because otherwise non-callable mocks couldn’t have callable methods.

Mocking imports with patch.dict

One situation where mocking can be hard is where you have a local import inside a function. These are harder to mock because they aren’t using an object from the module namespace that we can patch out.

Generally local imports are to be avoided. They are sometimes done to prevent circular dependencies, for which there is *usually* a much better way to solve the problem (refactor the code) or to prevent “up front costs” by delaying the import. This can also be solved in better ways than an unconditional local import (store the module as a class or module attribute and only do the import on first use).

That aside there is a way to use `mock` to affect the results of an import. Importing fetches an *object* from the **`sys.modules`** dictionary. Note that it fetches an *object*, which need not be a module. Importing a module for the first time results in a module object being put in `sys.modules`, so usually when you import something you get a module back. This need not be the case however.

This means you can use `patch.dict()` to *temporarily* put a mock in place in `sys.modules`. Any imports whilst this patch is active will fetch the mock. When the patch is complete (the decorated function exits, the with statement body is complete or `patcher.stop()` is called) then whatever was there previously will be restored safely.

Here's an example that mocks out the 'fooble' module.

```
>>> import sys
>>> mock = Mock()
>>> with patch.dict('sys.modules', {'fooble': mock}):
...    import fooble
...    fooble.blob()
...
<Mock name='mock.blob()' id='...'>
>>> assert 'fooble' not in sys.modules
>>> mock.blob.assert_called_once_with()
```

As you can see the `import fooble` succeeds, but on exit there is no 'fooble' left in `sys.modules`.

This also works for the `from module import name` form:

```
>>> mock = Mock()
>>> with patch.dict('sys.modules', {'fooble': mock}):
...    from fooble import blob
...    blob.blip()
...
<Mock name='mock.blob.blip()' id='...'>
>>> mock.blob.blip.assert_called_once_with()
```

With slightly more work you can also mock package imports:

```
>>> mock = Mock()
>>> modules = {'package': mock, 'package.module': mock.module}
>>> with patch.dict('sys.modules', modules):
...     from package.module import fooble
...     fooble()
...
<Mock name='mock.module.fooble()' id='...'>
>>> mock.module.fooble.assert_called_once_with()
```

Tracking order of calls and less verbose call assertions

The `Mock` class allows you to track the *order* of method calls on your mock objects through the `method_calls` attribute. This doesn't allow you to track the order of calls between separate mock objects, however we can use `mock_calls` to achieve the same effect.

Because mocks track calls to child mocks in `mock_calls`, and accessing an arbitrary attribute of a mock creates a child mock, we can create our separate mocks from a parent one. Calls to those child mock will then all be recorded, in order, in the `mock_calls` of the parent:

```
>>> manager = Mock()
>>> mock_foo = manager.foo
>>> mock_bar = manager.bar

>>> mock_foo.something()
<Mock name='mock.foo.something()' id='...'>
>>> mock_bar.other.thing()
<Mock name='mock.bar.other.thing()' id='...'>

>>> manager.mock_calls
[call.foo.something(), call.bar.other.thing()]
```

We can then assert about the calls, including the order, by comparing with the `mock_calls` attribute on the manager mock:

```
>>> expected_calls = [call.foo.something(),
call.bar.other.thing()]
>>> manager.mock_calls == expected_calls
True
```

If `patch` is creating, and putting in place, your mocks then you can attach them to a manager mock using the **`attach_mock()`** method. After attaching calls will be recorded in `mock_calls` of the manager.

```
>>> manager = MagicMock()
>>> with patch('mymodule.Class1') as MockClass1:
...     with patch('mymodule.Class2') as MockClass2:
...         manager.attach_mock(MockClass1, 'MockClass1')
...         manager.attach_mock(MockClass2, 'MockClass2')
...         MockClass1().foo()
...         MockClass2().bar()
<MagicMock name='mock.MockClass1().foo()' id='...'>
<MagicMock name='mock.MockClass2().bar()' id='...'>
>>> manager.mock_calls
[call.MockClass1(),
call.MockClass1().foo(),
call.MockClass2(),
call.MockClass2().bar()]
```

If many calls have been made, but you’re only interested in a particular sequence of them then an alternative is to use the **`assert_has_calls()`** method. This takes a list of calls (constructed with the **`call`** object). If that sequence of calls are in **`mock_calls`** then the assert succeeds.

```
>>> m = MagicMock()
>>> m().foo().bar().baz()
<MagicMock name='mock().foo().bar().baz()' id='...'>
>>> m.one().two().three()
<MagicMock name='mock.one().two().three()' id='...'>
>>> calls = call.one().two().three().call_list()
>>> m.assert_has_calls(calls)
```

Even though the chained call `m.one().two().three()` aren’t the only calls that have been made to the mock, the assert still succeeds.

Sometimes a mock may have several calls made to it, and you are only interested in asserting about *some* of those calls. You may not even care about the order. In this case you can pass `any_order=True` to `assert_has_calls`:

```
>>> m = MagicMock()
>>> m(1), m.two(2, 3), m.seven(7), m.fifty('50')
(...)
>>> calls = [call.fifty('50'), call(1), call.seven(7)]
>>> m.assert_has_calls(calls, any_order=True)
```

More complex argument matching

Using the same basic concept as **`ANY`** we can implement matchers to do more complex assertions on objects used as arguments to mocks.

Suppose we expect some object to be passed to a mock that by default compares equal based on object identity (which is the Python default for user defined classes). To use **`assert_called_with()`** we would need to pass in the exact same object. If we are only interested in some of the attributes of this object then we can create a matcher that will check these attributes for us.

You can see in this example how a ‘standard’ call to `assert_called_with` isn’t sufficient:

```
>>> class Foo:
...     def __init__(self, a, b):
...         self.a, self.b = a, b
...
>>> mock = Mock(return_value=None)
>>> mock(Foo(1, 2))
>>> mock.assert_called_with(Foo(1, 2))
Traceback (most recent call last):
    ...
```

```
AssertionError: expected call not found.
Expected: mock(<__main__.Foo object at 0x...>)
Actual: mock(<__main__.Foo object at 0x...>)
```

A comparison function for our `Foo` class might look something like this:

```
>>> def compare(self, other):
...     if not type(self) == type(other):
...         return False
...     if self.a != other.a:
...         return False
...     if self.b != other.b:
...         return False
...     return True
...
```

And a matcher object that can use comparison functions like this for its equality operation would look something like this:

```
>>> class Matcher:
...     def __init__(self, compare, some_obj):
...         self.compare = compare
...         self.some_obj = some_obj
...     def __eq__(self, other):
...         return self.compare(self.some_obj, other)
...
```

Putting all this together:

```
>>> match_foo = Matcher(compare, Foo(1, 2))
>>> mock.assert_called_with(match_foo)
```

The `Matcher` is instantiated with our compare function and the `Foo` object we want to compare against. In `assert_called_with` the `Matcher` equality method will be called, which compares the object the mock was called with against the one we created our matcher with. If they match then `assert_called_with` passes, and if they don’t an `AssertionError` is raised:

```
>>> match_wrong = Matcher(compare, Foo(3, 4))
>>> mock.assert_called_with(match_wrong)
Traceback (most recent call last):
    ...
AssertionError: Expected: ((<Matcher object at 0x...>,), {})
Called with: ((<Foo object at 0x...>,), {})
```

With a bit of tweaking you could have the comparison function raise the `AssertionError` directly and provide a more useful failure message.

As of version 1.5, the Python testing library PyHamcrest [https://pyhamcrest.readthedocs.io/] provides similar functionality, that may be useful here, in the form of its equality matcher (hamcrest.library.integration.match_equality [https://pyhamcrest.readthedocs.io/en/release-1.8/integration/#module-hamcrest.library.integration.match_equality]).

`test` — Regression tests package for Python

Note

The `test` package is meant for internal use by Python only. It is documented for the benefit of the core developers of Python. Any use of this package outside of Python's standard library is discouraged as code mentioned here can change or be removed without notice between releases of Python.

The `test` package contains all regression tests for Python as well as the modules `test.support` and `test.regrtest`. `test.support` is used to enhance your tests while `test.regrtest` drives the testing suite.

Each module in the `test` package whose name starts with `test_` is a testing suite for a specific module or feature. All new tests should be written using the `unittest` or `doctest` module. Some older tests are written using a "traditional" testing style that compares output printed to `sys.stdout`; this style of test is considered deprecated.

See also

Module `unittest`
Writing PyUnit regression tests.

Module `doctest`

Tests embedded in documentation strings.

Writing Unit Tests for the `test` package

It is preferred that tests that use the `unittest` module follow a few guidelines. One is to name the test module by starting it with `test_` and end it with the name of the module being tested. The test methods in the test module should start with `test_` and end with a description of what the method is testing. This is needed so that the methods are recognized by the test driver as test methods. Also, no documentation string for the method should be included. A comment (such as `# Tests function returns only True or False`) should be used to provide documentation for test methods. This is done because documentation strings get printed out if they exist and thus what test is being run is not stated.

A basic boilerplate is often used:

```
import unittest
from test import support

class MyTestCase1(unittest.TestCase):

    # Only use setUp() and tearDown() if necessary

    def setUp(self):
        ... code to execute in preparation for tests ...

    def tearDown(self):
        ... code to execute to clean up after tests ...

    def test_feature_one(self):
        # Test feature one.
        ... testing code ...

    def test_feature_two(self):
        # Test feature two.
        ... testing code ...

    ... more test methods ...

class MyTestCase2(unittest.TestCase):
    ... same structure as MyTestCase1 ...

... more test classes ...

if __name__ == '__main__':
    unittest.main()
```

This code pattern allows the testing suite to be run by **`test.regrtest`**, on its own as a script that supports the **`unittest`** CLI, or via the `python -m unittest` CLI.

The goal for regression testing is to try to break code. This leads to a few guidelines to be followed:

- The testing suite should exercise all classes, functions, and constants. This includes not just the external API that is to be presented to the outside world but also “private” code.

- Whitebox testing (examining the code being tested when the tests are being written) is preferred. Blackbox testing (testing only the published user interface) is not complete enough to make sure all boundary and edge cases are tested.

- Make sure all possible values are tested including invalid ones. This makes sure that not only all valid values are acceptable but also that improper values are handled correctly.

- Exhaust as many code paths as possible. Test where branching occurs and thus tailor input to make sure as many different paths through the code are taken.

- Add an explicit test for any bugs discovered for the tested code. This will make sure that the error does not crop up again if the code is changed in the future.

- Make sure to clean up after your tests (such as close and remove all temporary files).

- If a test is dependent on a specific condition of the operating system then verify the condition already exists before attempting the test.

- Import as few modules as possible and do it as soon as possible. This minimizes external dependencies of tests and also minimizes possible anomalous behavior from side-effects of importing a module.

- Try to maximize code reuse. On occasion, tests will vary by something as small as what type of input is used. Minimize code duplication by subclassing a basic test class with a class that specifies the input:

```
class TestFuncAcceptsSequencesMixin:

    func = mySuperWhammyFunction

    def test_func(self):
        self.func(self.arg)

class AcceptLists(TestFuncAcceptsSequencesMixin,
unittest.TestCase):
    arg = [1, 2, 3]

class AcceptStrings(TestFuncAcceptsSequencesMixin,
unittest.TestCase):
    arg = 'abc'

class AcceptTuples(TestFuncAcceptsSequencesMixin,
unittest.TestCase):
    arg = (1, 2, 3)
```

> When using this pattern, remember that all classes that inherit from `unittest.TestCase` are run as tests. The **`TestFuncAcceptsSequencesMixin`** class in the example above does not have any data and so can't be run by itself, thus it does not inherit from `unittest.TestCase`.

See also

Test Driven Development
A book by Kent Beck on writing tests before code.

Running tests using the command-line interface

The `test` package can be run as a script to drive Python's regression test suite, thanks to the `-m` option: **python -m test**. Under the hood, it uses `test.regrtest`; the call **python -m test.regrtest** used in previous Python versions still works. Running the script by itself automatically starts running all regression tests in the `test` package. It does this by finding all modules in the package whose name starts with `test_`, importing them, and executing the function **`test_main()`** if present or loading the tests via unittest.TestLoader.loadTestsFromModule if `test_main` does not exist. The names of tests to execute may also be passed to the script. Specifying a single regression test (**python -m test test_spam**) will minimize output and only print whether the test passed or failed.

Running `test` directly allows what resources are available for tests to use to be set. You do this by using the `-u` command-line option. Specifying `all` as the value for the `-u` option enables all possible resources: **python -m test -uall**. If all but one resource is desired (a more common case), a comma-separated list of resources that are not desired may be listed after `all`. The command **python -m test -uall,-audio,-largefile** will run `test` with all resources except the `audio` and `largefile` resources. For a list of all resources and more command-line options, run **python -m test -h**.

Some other ways to execute the regression tests depend on what platform the tests are being executed on. On Unix, you can run **make test** at the top-level directory where Python was built. On Windows, executing **rt.bat** from your `PCbuild` directory will run all regression tests.

Added in version 3.14: Output is colorized by default and can be controlled using environment variables.

`test.support` — Utilities for the Python test suite

The `test.support` module provides support for Python’s regression test suite.

Note

`test.support` is not a public module. It is documented here to help Python developers write tests. The API of this module is subject to change without backwards compatibility concerns between releases.

This module defines the following exceptions:

exception test.support.TestFailed

Exception to be raised when a test fails. This is deprecated in favor of `unittest`-based tests and `unittest.TestCase`’s assertion methods.

exception test.support.ResourceDenied

Subclass of `unittest.SkipTest`. Raised when a resource (such as a network connection) is not available. Raised by the `requires()` function.

The `test.support` module defines the following constants:

test.support.verbose

`True` when verbose output is enabled. Should be checked when more detailed information is desired about a running test. *verbose* is set by `test.regrtest`.

test.support.is_jython

`True` if the running interpreter is Jython.

test.support.is_android

`True` if `sys.platform` is `android`.

test.support.is_emscripten

`True` if `sys.platform` is `emscripten`.

test.support.is_wasi

`True` if `sys.platform` is `wasi`.

test.support.is_apple_mobile

`True` if `sys.platform` is `ios`, `tvos`, or `watchos`.

test.support.is_apple

`True` if `sys.platform` is `darwin` or `is_apple_mobile` is `True`.

test.support.unix_shell

Path for shell if not on Windows; otherwise `None`.

test.support.LOOPBACK_TIMEOUT

Timeout in seconds for tests using a network server listening on the network local loopback interface like `127.0.0.1`.

The timeout is long enough to prevent test failure: it takes into account that the client and the server can run in different threads or even different processes.

The timeout should be long enough for `connect()`, `recv()` and `send()` methods of `socket.socket`.

Its default value is 5 seconds.

See also `INTERNET_TIMEOUT`.

test.support.INTERNET_TIMEOUT

Timeout in seconds for network requests going to the internet.

The timeout is short enough to prevent a test to wait for too long if the internet request is blocked for whatever reason.

Usually, a timeout using `INTERNET_TIMEOUT` should not mark a test as failed, but skip the test instead: see `transient_internet()`.

Its default value is 1 minute.

See also `LOOPBACK_TIMEOUT`.

test.support.SHORT_TIMEOUT

Timeout in seconds to mark a test as failed if the test takes “too long”.

The timeout value depends on the regrtest `--timeout` command line option.

If a test using **`SHORT_TIMEOUT`** starts to fail randomly on slow buildbots, use **`LONG_TIMEOUT`** instead.

Its default value is 30 seconds.

test.support.LONG_TIMEOUT

Timeout in seconds to detect when a test hangs.

It is long enough to reduce the risk of test failure on the slowest Python buildbots. It should not be used to mark a test as failed if the test takes “too long”. The timeout value depends on the regrtest `--timeout` command line option.

Its default value is 5 minutes.

See also **`LOOPBACK_TIMEOUT`**, **`INTERNET_TIMEOUT`** and **`SHORT_TIMEOUT`**.

test.support.PGO

Set when tests can be skipped when they are not useful for PGO.

test.support.PIPE_MAX_SIZE

A constant that is likely larger than the underlying OS pipe buffer size, to make writes blocking.

test.support.Py_DEBUG

`True` if Python was built with the **`Py_DEBUG`** macro defined, that is, if Python was built in debug mode.

Added in version 3.12.

test.support.SOCK_MAX_SIZE

A constant that is likely larger than the underlying OS socket buffer size, to make writes blocking.

test.support.TEST_SUPPORT_DIR

Set to the top level directory that contains `test.support`.

test.support.TEST_HOME_DIR

Set to the top level directory for the test package.

test.support.TEST_DATA_DIR

Set to the `data` directory within the test package.

test.support.MAX_Py_ssize_t

Set to `sys.maxsize` for big memory tests.

test.support.max_memuse

Set by `set_memlimit()` as the memory limit for big memory tests. Limited by `MAX_Py_ssize_t`.

test.support.real_max_memuse

Set by `set_memlimit()` as the memory limit for big memory tests. Not limited by `MAX_Py_ssize_t`.

test.support.MISSING_C_DOCSTRINGS

Set to `True` if Python is built without docstrings (the **`WITH_DOC_STRINGS`** macro is not defined). See the `configure --without-doc-strings` option.

See also the `HAVE_DOCSTRINGS` variable.

test.support.HAVE_DOCSTRINGS

Set to `True` if function docstrings are available. See the `python -OO` option, which strips docstrings of functions implemented in Python.

See also the `MISSING_C_DOCSTRINGS` variable.

test.support.TEST_HTTP_URL

Define the URL of a dedicated HTTP server for the network tests.

test.support.ALWAYS_EQ

Object that is equal to anything. Used to test mixed type comparison.

test.support.NEVER_EQ

Object that is not equal to anything (even to `ALWAYS_EQ`). Used to test mixed type comparison.

test.support.LARGEST

Object that is greater than anything (except itself). Used to test mixed type comparison.

test.support.SMALLEST

Object that is less than anything (except itself). Used to test mixed type comparison.

The `test.support` module defines the following functions:

test.support.busy_retry(*timeout*, *err_msg=None*, */*, ***, *error=True*)

Run the loop body until `break` stops the loop.

After *timeout* seconds, raise an `AssertionError` if *error* is true, or just stop the loop if *error* is false.

Example:

```
for _ in support.busy_retry(support.SHORT_TIMEOUT):
    if check():
        break
```

Example of error=False usage:

```
for _ in support.busy_retry(support.SHORT_TIMEOUT,
error=False):
    if check():
        break
else:
    raise RuntimeError('my custom error')
```

test.support.sleeping_retry(*timeout*, *err_msg=None*, */*, ***, *init_delay=0.010*, *max_delay=1.0*, *error=True*)

Wait strategy that applies exponential backoff.

Run the loop body until `break` stops the loop. Sleep at each loop iteration, but not at the first iteration. The sleep delay is doubled at each iteration (up to *max_delay* seconds).

See **`busy_retry()`** documentation for the parameters usage.

Example raising an exception after SHORT_TIMEOUT seconds:

```
for _ in support.sleeping_retry(support.SHORT_TIMEOUT):
    if check():
        break
```

Example of error=False usage:

```
for _ in support.sleeping_retry(support.SHORT_TIMEOUT,
error=False):
    if check():
        break
else:
    raise RuntimeError('my custom error')
```

test.support.is_resource_enabled(*resource*)

Return `True` if *resource* is enabled and available. The list of available resources is only set when **`test.regrtest`** is executing the tests.

test.support.get_resource_value(*resource*)

Return the value specified for *resource* (as `-u` *`resource`*`=`*`value`*). Return `None` if *resource* is disabled or no value is specified.

test.support.python_is_optimized()

Return `True` if Python was not built with `-O0` or `-Og`.

test.support.with_pymalloc()

Return **`_testcapi.WITH_PYMALLOC`**.

test.support.requires(*resource*, *msg=None*)

Raise **`ResourceDenied`** if *resource* is not available. *msg* is the argument to **`ResourceDenied`** if it is raised. Always returns `True` if called by a function whose `__name__` is `'__main__'`. Used when tests are executed by **`test.regrtest`**.

test.support.sortdict(*dict*)

Return a repr of *dict* with keys sorted.

test.support.findfile(*filename*, *subdir=None*)

Return the path to the file named *filename*. If no match is found *filename* is returned. This does not equal a failure since it could be the path to the file.

Setting *subdir* indicates a relative path to use to find the file rather than looking directly in the path directories.

test.support.get_pagesize()

Get size of a page in bytes.

Added in version 3.12.

test.support.setswitchinterval(*interval*)

Set the `sys.setswitchinterval()` to the given *interval*. Defines a minimum interval for Android systems to prevent the system from hanging.

test.support.check_impl_detail(***guards*)

Use this check to guard CPython's implementation-specific tests or to run them only on the implementations guarded by the arguments. This function returns `True` or `False` depending on the host platform. Example usage:

```
check_impl_detail()               # Only on CPython
(default).
check_impl_detail(jython=True)    # Only on Jython.
check_impl_detail(cpython=False)  # Everywhere except
CPython.
```

test.support.set_memlimit(*limit*)

Set the values for `max_memuse` and `real_max_memuse` for big memory tests.

test.support.record_original_stdout(*stdout*)

Store the value from *stdout*. It is meant to hold the stdout at the time the regrtest began.

test.support.get_original_stdout()

Return the original stdout set by `record_original_stdout()` or `sys.stdout` if it's not set.

test.support.args_from_interpreter_flags()

Return a list of command line arguments reproducing the current settings in `sys.flags` and `sys.warnoptions`.

test.support.optim_args_from_interpreter_flags()

Return a list of command line arguments reproducing the current optimization settings in `sys.flags`.

test.support.captured_stdin()
test.support.captured_stdout()
test.support.captured_stderr()

A context managers that temporarily replaces the named stream with `io.StringIO` object.

Example use with output streams:

```
with captured_stdout() as stdout, captured_stderr() as
stderr:
    print("hello")
    print("error", file=sys.stderr)
assert stdout.getvalue() == "hello\n"
assert stderr.getvalue() == "error\n"
```

Example use with input stream:

```
with captured_stdin() as stdin:
    stdin.write('hello\n')
    stdin.seek(0)
    # call test code that consumes from sys.stdin
    captured = input()
self.assertEqual(captured, "hello")
```

test.support.disable_faulthandler()

A context manager that temporary disables `faulthandler`.

test.support.gc_collect()

Force as many objects as possible to be collected. This is needed because timely deallocation is not guaranteed by the garbage collector. This means that `__del__` methods may be called later than expected and weakrefs may remain alive for longer than expected.

test.support.disable_gc()

A context manager that disables the garbage collector on entry. On exit, the garbage collector is restored to its prior state.

test.support.swap_attr(*obj*, *attr*, *new_val*)

Context manager to swap out an attribute with a new object.

Usage:

```
with swap_attr(obj, "attr", 5):
    ...
```

This will set `obj.attr` to 5 for the duration of the `with` block, restoring the old value at the end of the block. If `attr` doesn’t exist on `obj`, it will be created and then deleted at the end of the block.

The old value (or `None` if it doesn’t exist) will be assigned to the target of the “as” clause, if there is one.

test.support.swap_item(*obj*, *attr*, *new_val*)

Context manager to swap out an item with a new object.

Usage:

```
with swap_item(obj, "item", 5):
    ...
```

This will set `obj["item"]` to 5 for the duration of the `with` block, restoring the old value at the end of the block. If `item` doesn’t exist on `obj`, it will be created and then deleted at the end of the block.

The old value (or `None` if it doesn’t exist) will be assigned to the target of the “as” clause, if there is one.

test.support.flush_std_streams()

Call the `flush()` method on `sys.stdout` and then on `sys.stderr`. It can be used to make sure that the logs order is consistent before writing into stderr.

Added in version 3.11.

test.support.print_warning(*msg*)

Print a warning into `sys.__stderr__`. Format the message as: `f"Warning -- {msg}"`. If *msg* is made of multiple lines, add `"Warning -- "` prefix to each line.

Added in version 3.9.

test.support.wait_process(*pid*, *, *exitcode*, *timeout=None*)

Wait until process *pid* completes and check that the process exit code is *exitcode*.

Raise an `AssertionError` if the process exit code is not equal to *exitcode*.

If the process runs longer than *timeout* seconds (`SHORT_TIMEOUT` by default), kill the process and raise an `AssertionError`. The timeout feature is not available on Windows.

Added in version 3.9.

test.support.calcobjsize(*fmt*)

Return the size of the `PyObject` whose structure members are defined by *fmt*. The returned value includes the size of the Python object header and alignment.

test.support.calcvobjsize(*fmt*)

Return the size of the `PyVarObject` whose structure members are defined by *fmt*. The returned value includes the size of the Python object header and alignment.

test.support.checksizeof(*test*, *o*, *size*)

For testcase *test*, assert that the `sys.getsizeof` for *o* plus the GC header size equals *size*.

@test.support.anticipate_failure(*condition*)

A decorator to conditionally mark tests with `unittest.expectedFailure()`. Any use of this decorator should have an associated comment identifying the relevant tracker issue.

test.support.system_must_validate_cert(*f*)

A decorator that skips the decorated test on TLS certification validation failures.

@test.support.run_with_locale(*catstr*, **locales*)

A decorator for running a function in a different locale, correctly resetting it after it has finished. *catstr* is the locale category as a string (for example `"LC_ALL"`). The *locales* passed will be tried sequentially, and the first valid locale will be used.

@test.support.run_with_tz(*tz*)

A decorator for running a function in a specific timezone, correctly resetting it after it has finished.

@test.support.requires_freebsd_version(**min_version*)

Decorator for the minimum version when running test on FreeBSD. If the FreeBSD version is less than the minimum, the test is skipped.

@test.support.requires_linux_version(**min_version*)

Decorator for the minimum version when running test on Linux. If the Linux version is less than the minimum, the test is skipped.

@test.support.requires_mac_version(**min_version*)

Decorator for the minimum version when running test on macOS. If the macOS version is less than the minimum, the test is skipped.

@test.support.requires_gil_enabled

Decorator for skipping tests on the free-threaded build. If the GIL is disabled, the test is skipped.

@test.support.requires_IEEE_754

Decorator for skipping tests on non-IEEE 754 platforms.

@test.support.requires_zlib

Decorator for skipping tests if `zlib` doesn’t exist.

@test.support.requires_gzip

Decorator for skipping tests if `gzip` doesn’t exist.

@test.support.requires_bz2

Decorator for skipping tests if `bz2` doesn’t exist.

@test.support.requires_lzma

Decorator for skipping tests if `lzma` doesn’t exist.

@test.support.requires_resource(*resource*)

Decorator for skipping tests if *resource* is not available.

@test.support.requires_docstrings

Decorator for only running the test if `HAVE_DOCSTRINGS`.

@test.support.requires_limited_api

Decorator for only running the test if Limited C API is available.

@test.support.cpython_only

Decorator for tests only applicable to CPython.

@test.support.impl_detail(*msg=None*, ***guards*)

Decorator for invoking `check_impl_detail()` on *guards*. If that returns `False`, then uses *msg* as the reason for skipping the test.

@test.support.thread_unsafe(*reason=None*)

Decorator for marking tests as thread-unsafe. This test always runs in one thread even when invoked with `--parallel-threads`.

@test.support.no_tracing

Decorator to temporarily turn off tracing for the duration of the test.

@test.support.refcount_test

Decorator for tests which involve reference counting. The decorator does not run the test if it is not run by CPython. Any trace function is unset for the duration of the test to prevent unexpected refcounts caused by the trace function.

@test.support.bigmemtest(*size*, *memuse*, *dry_run=True*)

Decorator for bigmem tests.

size is a requested size for the test (in arbitrary, test-interpreted units.) *memuse* is the number of bytes per unit for the test, or a good estimate of it. For example, a test that needs two byte buffers, of 4 GiB each, could be decorated with `@bigmemtest(size=_4G, memuse=2)`.

The *size* argument is normally passed to the decorated test method as an extra argument. If *dry_run* is `True`, the value passed to the test method may be less than the requested value. If *dry_run* is `False`, it means the test doesn’t support dummy runs when `-M` is not specified.

@test.support.bigaddrspacetest

Decorator for tests that fill the address space.

test.support.linked_to_musl()

Return `False` if there is no evidence the interpreter was compiled with `musl`, otherwise return a version triple, either `(0, 0, 0)` if the version is unknown, or the actual version if it is known. Intended for use in `skip` decorators. `emscripten` and `wasi` are assumed to be compiled with `musl`; otherwise `platform.libc_ver` is checked.

test.support.check_syntax_error(*testcase*, *statement*, *errtext*='', *, *lineno*=*None*, *offset*=*None*)

Test for syntax errors in *statement* by attempting to compile *statement*. *testcase* is the **`unittest`** instance for the test. *errtext* is the regular expression which should match the string representation of the raised **`SyntaxError`**. If *lineno* is not `None`, compares to the line of the exception. If *offset* is not `None`, compares to the offset of the exception.

test.support.open_urlresource(*url*, **args*, ***kw*)

Open *url*. If open fails, raises **`TestFailed`**.

test.support.reap_children()

Use this at the end of `test_main` whenever sub-processes are started. This will help ensure that no extra children (zombies) stick around to hog resources and create problems when looking for refleaks.

test.support.get_attribute(*obj*, *name*)

Get an attribute, raising **`unittest.SkipTest`** if **`AttributeError`** is raised.

test.support.catch_unraisable_exception()

Context manager catching unraisable exception using **`sys.unraisablehook()`**.

Storing the exception value (`cm.unraisable.exc_value`) creates a reference cycle. The reference cycle is broken explicitly when the context manager exits.

Storing the object (`cm.unraisable.object`) can resurrect it if it is set to an object which is being finalized. Exiting the context manager clears the stored object.

Usage:

```
with support.catch_unraisable_exception() as cm:
    # code creating an "unraisable exception"
    ...

    # check the unraisable exception: use cm.unraisable
    ...

# cm.unraisable attribute no longer exists at this point
# (to break a reference cycle)
```

Added in version 3.8.

test.support.load_package_tests(*pkg_dir*, *loader*, *standard_tests*, *pattern*)

Generic implementation of the **`unittest`** `load_tests` protocol for use in test packages. *pkg_dir* is the root directory of the package; *loader*, *standard_tests*, and *pattern* are the arguments expected by `load_tests`. In simple cases, the test package's `__init__.py` can be the following:

```
import os
from test.support import load_package_tests

def load_tests(*args):
    return load_package_tests(os.path.dirname(__file__),
*args)
```

test.support.detect_api_mismatch(*ref_api*, *other_api*, *, *ignore=()*)

Returns the set of attributes, functions or methods of *ref_api* not found on *other_api*, except for a defined list of items to be ignored in this check specified in *ignore*.

By default this skips private attributes beginning with '_' but includes all magic methods, i.e. those starting and ending in '__'.

Added in version 3.5.

test.support.patch(*test_instance*, *object_to_patch*, *attr_name*, *new_value*)

Override *object_to_patch.attr_name* with *new_value*. Also add cleanup procedure to *test_instance* to restore *object_to_patch* for *attr_name*. The *attr_name* should be a valid attribute for *object_to_patch*.

test.support.run_in_subinterp(*code*)

Run *code* in subinterpreter. Raise `unittest.SkipTest` if `tracemalloc` is enabled.

test.support.check_free_after_iterating(*test*, *iter*, *cls*, *args=()*)

Assert instances of *cls* are deallocated after iterating.

test.support.missing_compiler_executable(*cmd_names=[]*)

Check for the existence of the compiler executables whose names are listed in *cmd_names* or all the compiler executables when *cmd_names* is empty and return the first missing executable or `None` when none is found missing.

test.support.check__all__(*test_case*, *module*, *name_of_module=None*, *extra=()*, *not_exported=()*)

Assert that the `__all__` variable of *module* contains all public names.

The module's public names (its API) are detected automatically based on whether they match the public name convention and were defined in *module*.

The *name_of_module* argument can specify (as a string or tuple thereof) what module(s) an API could be defined in order to be detected as a public API. One case for this is when *module* imports part of its public API from other modules, possibly a C backend (like `csv` and its `_csv`).

The *extra* argument can be a set of names that wouldn't otherwise be automatically detected as "public", like objects without a proper `__module__` attribute. If provided, it will be added to the automatically detected ones.

The *not_exported* argument can be a set of names that must not be treated as part of the public API even though their names indicate otherwise.

Example use:

```
import bar
import foo
import unittest
from test import support

class MiscTestCase(unittest.TestCase):
    def test__all__(self):
        support.check__all__(self, foo)

class OtherTestCase(unittest.TestCase):
    def test__all__(self):
        extra = {'BAR_CONST', 'FOO_CONST'}
        not_exported = {'baz'}  # Undocumented name.
        # bar imports part of its API from _bar.
        support.check__all__(self, bar, ('bar', '_bar'),
                             extra=extra,
not_exported=not_exported)
```

Added in version 3.6.

test.support.skip_if_broken_multiprocessing_synchronize()

Skip tests if the **`multiprocessing.synchronize`** module is missing, if there is no available semaphore implementation, or if creating a lock raises an `OSError`.

Added in version 3.10.

test.support.check_disallow_instantiation(*test_case*, *tp*, **args*, ***kwds*)

Assert that type *tp* cannot be instantiated using *args* and *kwds*.

Added in version 3.10.

test.support.adjust_int_max_str_digits(*max_digits*)

This function returns a context manager that will change the global `sys.set_int_max_str_digits()` setting for the duration of the context to allow execution of test code that needs a different limit on the number of digits when converting between an integer and string.

Added in version 3.11.

The `test.support` module defines the following classes:

class test.support.SuppressCrashReport

A context manager used to try to prevent crash dialog popups on tests that are expected to crash a subprocess.

On Windows, it disables Windows Error Reporting dialogs using SetErrorMode [https://msdn.microsoft.com/en-us/library/windows/desktop/ms680621.aspx].

On UNIX, `resource.setrlimit()` is used to set `resource.RLIMIT_CORE`'s soft limit to 0 to prevent coredump file creation.

On both platforms, the old value is restored by `__exit__()`.

class test.support.SaveSignals

Class to save and restore signal handlers registered by the Python signal handler.

save(*self*)

Save the signal handlers to a dictionary mapping signal numbers to the current signal handler.

restore(*self*)

Set the signal numbers from the `save()` dictionary to the saved handler.

class test.support.Matcher

matches(*self*, *d*, ***kwargs*)

Try to match a single dict with the supplied arguments.

match_value(*self*, *k*, *dv*, *v*)

Try to match a single stored value (*dv*) with a supplied value (*v*).

`test.support.socket_helper` — Utilities for socket tests

The `test.support.socket_helper` module provides support for socket tests.

Added in version 3.9.

test.support.socket_helper.IPV6_ENABLED

Set to `True` if IPv6 is enabled on this host, `False` otherwise.

test.support.socket_helper.find_unused_port(*family=socket.AF_INET*, *socktype=socket.SOCK_STREAM*)

Returns an unused port that should be suitable for binding. This is achieved by creating a temporary socket with the same family and type as the `sock` parameter (default is `AF_INET`, `SOCK_STREAM`), and binding it to the specified host address (defaults to `0.0.0.0`) with the port set to 0, eliciting an unused ephemeral port from the OS. The temporary socket is then closed and deleted, and the ephemeral port is returned.

Either this method or `bind_port()` should be used for any tests where a server socket needs to be bound to a particular port for the duration of the test. Which one to use depends on whether the calling code is creating a Python socket, or if an unused port needs to be provided in a constructor or passed to an external program (i.e. the `-accept` argument to openssl's s_server mode). Always prefer `bind_port()` over `find_unused_port()` where possible. Using a hard coded port is discouraged since it can make multiple instances of the test impossible to run simultaneously, which is a problem for buildbots.

test.support.socket_helper.bind_port(*sock*, *host=HOST*)

Bind the socket to a free port and return the port number. Relies on ephemeral ports in order to ensure we are using an unbound port. This is important as many tests may be running simultaneously, especially in a buildbot environment. This method raises an exception if the `sock.family` is `AF_INET` and `sock.type` is `SOCK_STREAM`, and the socket has **`SO_REUSEADDR`** or **`SO_REUSEPORT`** set on it. Tests should never set these socket options for TCP/IP sockets. The only case for setting these options is testing multicasting via multiple UDP sockets.

Additionally, if the **`SO_EXCLUSIVEADDRUSE`** socket option is available (i.e. on Windows), it will be set on the socket. This will prevent anyone else from binding to our host/port for the duration of the test.

test.support.socket_helper.bind_unix_socket(*sock*, *addr*)

Bind a Unix socket, raising `unittest.SkipTest` if `PermissionError` is raised.

@test.support.socket_helper.skip_unless_bind_unix_socket

A decorator for running tests that require a functional `bind()` for Unix sockets.

test.support.socket_helper.transient_internet(*resource_name*, *, *timeout=30.0*, *errnos=()*)

A context manager that raises `ResourceDenied` when various issues with the internet connection manifest themselves as exceptions.

`test.support.script_helper` — Utilities for the Python execution tests

The `test.support.script_helper` module provides support for Python’s script execution tests.

test.support.script_helper.interpreter_requires_environment()

Return `True` if `sys.executable interpreter` requires environment variables in order to be able to run at all.

This is designed to be used with `@unittest.skipIf()` to annotate tests that need to use an `assert_python*()` function to launch an isolated mode (`-I`) or no environment mode (`-E`) sub-interpreter process.

A normal build & test does not run into this situation but it can happen when trying to run the standard library test suite from an interpreter that doesn’t have an obvious home with Python’s current home finding logic.

Setting **`PYTHONHOME`** is one way to get most of the testsuite to run in that situation. **`PYTHONPATH`** or **`PYTHONUSERSITE`** are other common environment variables that might impact whether or not the interpreter can start.

test.support.script_helper.run_python_until_end(**args*, ***env_vars*)

Set up the environment based on *env_vars* for running the interpreter in a subprocess. The values can include `__isolated`, `__cleanenv`, `__cwd`, and `TERM`.

Changed in version 3.9: The function no longer strips whitespaces from *stderr*.

test.support.script_helper.assert_python_ok(**args*, ***env_vars*)

Assert that running the interpreter with *args* and optional environment variables *env_vars* succeeds (`rc == 0`) and return a `(return code, stdout, stderr)` tuple.

If the *__cleanenv* keyword-only parameter is set, *env_vars* is used as a fresh environment.

Python is started in isolated mode (command line option `-I`), except if the *__isolated* keyword-only parameter is set to `False`.

Changed in version 3.9: The function no longer strips whitespaces from *stderr*.

test.support.script_helper.assert_python_failure(**args*, ***env_vars*)

Assert that running the interpreter with *args* and optional environment variables *env_vars* fails (`rc != 0`) and return a `(return code, stdout, stderr)` tuple.

See **`assert_python_ok()`** for more options.

Changed in version 3.9: The function no longer strips whitespaces from *stderr*.

test.support.script_helper.spawn_python(**args*, *stdout=subprocess.PIPE*, *stderr=subprocess.STDOUT*, ***kw*)

Run a Python subprocess with the given arguments.

kw is extra keyword args to pass to **`subprocess.Popen()`**. Returns a **`subprocess.Popen`** object.

test.support.script_helper.kill_python(*p*)

Run the given **`subprocess.Popen`** process until completion and return stdout.

test.support.script_helper.make_script(*script_dir*, *script_basename*, *source*, *omit_suffix=False*)

Create script containing *source* in path *script_dir* and *script_basename*. If *omit_suffix* is `False`, append `.py` to the name. Return the full script path.

test.support.script_helper.make_zip_script(*zip_dir*, *zip_basename*, *script_name*, *name_in_zip=None*)

Create zip file at *zip_dir* and *zip_basename* with extension `zip` which contains the files in *script_name*. *name_in_zip* is the archive name. Return a tuple containing `(full path, full path of archive name)`.

test.support.script_helper.make_pkg(*pkg_dir*, *init_source=''*)

Create a directory named *pkg_dir* containing an `__init__` file with *init_source* as its contents.

test.support.script_helper.make_zip_pkg(*zip_dir*, *zip_basename*, *pkg_name*, *script_basename*, *source*, *depth=1*, *compiled=False*)

Create a zip package directory with a path of *zip_dir* and *zip_basename* containing an empty `__init__` file and a file *script_basename* containing the *source*. If *compiled* is `True`, both source files will be compiled and added to the zip package. Return a tuple of the full zip path and the archive name for the zip file.

`test.support.bytecode_helper` — Support tools for testing correct bytecode generation

The `test.support.bytecode_helper` module provides support for testing and inspecting bytecode generation.

Added in version 3.9.

The module defines the following class:

class test.support.bytecode_helper.BytecodeTestCase(*unittest.TestCase*)

This class has custom assertion methods for inspecting bytecode.

BytecodeTestCase.get_disassembly_as_string(*co*)

Return the disassembly of *co* as string.

BytecodeTestCase.assertInBytecode(*x*, *opname*, *argval=_UNSPECIFIED*)

Return instr if *opname* is found, otherwise throws `AssertionError`.

BytecodeTestCase.assertNotInBytecode(*x*, *opname*, *argval=_UNSPECIFIED*)

Throws `AssertionError` if *opname* is found.

`test.support.threading_helper` — Utilities for threading tests

The `test.support.threading_helper` module provides support for threading tests.

Added in version 3.10.

test.support.threading_helper.join_thread(*thread*, *timeout=None*)

Join a *thread* within *timeout*. Raise an `AssertionError` if thread is still alive after *timeout* seconds.

@test.support.threading_helper.reap_threads

Decorator to ensure the threads are cleaned up even if the test fails.

test.support.threading_helper.start_threads(*threads*, *unlock=None*)

Context manager to start *threads*, which is a sequence of threads. *unlock* is a function called after the threads are started, even if an exception was raised; an example would be `threading.Event.set()`. `start_threads` will attempt to join the started threads upon exit.

test.support.threading_helper.threading_cleanup(**original_values*)

Cleanup up threads not specified in *original_values*. Designed to emit a warning if a test leaves running threads in the background.

test.support.threading_helper.threading_setup()

Return current thread count and copy of dangling threads.

test.support.threading_helper.wait_threads_exit(*timeout=None*)

Context manager to wait until all threads created in the `with` statement exit.

test.support.threading_helper.catch_threading_exception()

Context manager catching `threading.Thread` exception using `threading.excepthook()`.

Attributes set when an exception is caught:

- `exc_type`
- `exc_value`
- `exc_traceback`
- `thread`

See `threading.excepthook()` documentation.

These attributes are deleted at the context manager exit.

Usage:

```
with threading_helper.catch_threading_exception() as cm:
    # code spawning a thread which raises an exception
    ...

    # check the thread exception, use cm attributes:
    # exc_type, exc_value, exc_traceback, thread
    ...

# exc_type, exc_value, exc_traceback, thread attributes of
cm no longer
# exists at this point
# (to avoid reference cycles)
```

Added in version 3.8.

test.support.threading_helper.run_concurrently(*worker_func*, *nthreads*, *args=()*, *kwargs={}*)

Run the worker function concurrently in multiple threads. Re-raises an exception if any thread raises one, after all threads have finished.

`test.support.os_helper` — Utilities for os tests

The `test.support.os_helper` module provides support for os tests.

Added in version 3.10.

test.support.os_helper.FS_NONASCII

A non-ASCII character encodable by `os.fsencode()`.

test.support.os_helper.SAVEDCWD

Set to `os.getcwd()`.

test.support.os_helper.TESTFN

Set to a name that is safe to use as the name of a temporary file. Any temporary file that is created should be closed and unlinked (removed).

test.support.os_helper.TESTFN_NONASCII

Set to a filename containing the `FS_NONASCII` character, if it exists. This guarantees that if the filename exists, it can be encoded and decoded with the default filesystem encoding. This allows tests that require a non-ASCII filename to be easily skipped on platforms where they can’t work.

test.support.os_helper.TESTFN_UNENCODABLE

Set to a filename (str type) that should not be able to be encoded by file system encoding in strict mode. It may be `None` if it’s not possible to generate such a filename.

test.support.os_helper.TESTFN_UNDECODABLE

Set to a filename (bytes type) that should not be able to be decoded by file system encoding in strict mode. It may be `None` if it’s not possible to generate such a filename.

test.support.os_helper.TESTFN_UNICODE

Set to a non-ASCII name for a temporary file.

class test.support.os_helper.EnvironmentVarGuard

Class used to temporarily set or unset environment variables. Instances can be used as a context manager and have a complete dictionary interface for querying/modifying the underlying `os.environ`. After exit from the context manager all changes to environment variables done through this instance will be rolled back.

Changed in version 3.1: Added dictionary interface.

class test.support.os_helper.FakePath(*path*)

Simple path-like object. It implements the **`__fspath__()`** method which just returns the *path* argument. If *path* is an exception, it will be raised in **`__fspath__()`**.

EnvironmentVarGuard.set(*envvar*, *value*)

Temporarily set the environment variable `envvar` to the value of `value`.

EnvironmentVarGuard.unset(*envvar*, **others*)

Temporarily unset one or more environment variables.

Changed in version 3.14: More than one environment variable can be unset.

test.support.os_helper.can_symlink()

Return `True` if the OS supports symbolic links, `False` otherwise.

test.support.os_helper.can_xattr()

Return `True` if the OS supports xattr, `False` otherwise.

test.support.os_helper.change_cwd(*path*, *quiet=False*)

A context manager that temporarily changes the current working directory to *path* and yields the directory.

If *quiet* is `False`, the context manager raises an exception on error. Otherwise, it issues only a warning and keeps the current working directory the same.

test.support.os_helper.create_empty_file(*filename*)

Create an empty file with *filename*. If it already exists, truncate it.

test.support.os_helper.fd_count()

Count the number of open file descriptors.

test.support.os_helper.fs_is_case_insensitive(*directory*)

Return `True` if the file system for *directory* is case-insensitive.

test.support.os_helper.make_bad_fd()

Create an invalid file descriptor by opening and closing a temporary file, and returning its descriptor.

test.support.os_helper.rmdir(*filename*)

Call `os.rmdir()` on *filename*. On Windows platforms, this is wrapped with a wait loop that checks for the existence of the file, which is needed due to antivirus programs that can hold files open and prevent deletion.

test.support.os_helper.rmtree(*path*)

Call `shutil.rmtree()` on *path* or call `os.lstat()` and `os.rmdir()` to remove a path and its contents. As with `rmdir()`, on Windows platforms this is wrapped with a wait loop that checks for the existence of the files.

@test.support.os_helper.skip_unless_symlink

A decorator for running tests that require support for symbolic links.

@test.support.os_helper.skip_unless_xattr

A decorator for running tests that require support for xattr.

test.support.os_helper.temp_cwd(*name='tempcwd'*, *quiet=False*)

A context manager that temporarily creates a new directory and changes the current working directory (CWD).

The context manager creates a temporary directory in the current directory with name *name* before temporarily changing the current working directory. If *name* is `None`, the temporary directory is created using `tempfile.mkdtemp()`.

If *quiet* is `False` and it is not possible to create or change the CWD, an error is raised. Otherwise, only a warning is raised and the original CWD is used.

test.support.os_helper.temp_dir(*path=None*, *quiet=False*)

A context manager that creates a temporary directory at *path* and yields the directory.

If *path* is `None`, the temporary directory is created using `tempfile.mkdtemp()`. If *quiet* is `False`, the context manager raises an exception on error. Otherwise, if *path* is specified and cannot be created, only a warning is issued.

test.support.os_helper.temp_umask(*umask*)

A context manager that temporarily sets the process umask.

test.support.os_helper.unlink(*filename*)

Call `os.unlink()` on *filename*. As with `rmdir()`, on Windows platforms, this is wrapped with a wait loop that checks for the existence of the file.

`test.support.import_helper` — Utilities for import tests

The `test.support.import_helper` module provides support for import tests.

Added in version 3.10.

test.support.import_helper.forget(*module_name*)

Remove the module named *module_name* from `sys.modules` and delete any byte-compiled files of the module.

test.support.import_helper.import_fresh_module(*name*, *fresh=()*, *blocked=()*, *deprecated=False*)

This function imports and returns a fresh copy of the named Python module by removing the named module from `sys.modules` before doing the import. Note that unlike **`reload()`**, the original module is not affected by this operation.

fresh is an iterable of additional module names that are also removed from the `sys.modules` cache before doing the import.

blocked is an iterable of module names that are replaced with `None` in the module cache during the import to ensure that attempts to import them raise `ImportError`.

The named module and any modules named in the *fresh* and *blocked* parameters are saved before starting the import and then reinserted into `sys.modules` when the fresh import is complete.

Module and package deprecation messages are suppressed during this import if *deprecated* is `True`.

This function will raise `ImportError` if the named module cannot be imported.

Example use:

```
# Get copies of the warnings module for testing without affecting the
# version being used by the rest of the test suite. One copy uses the
# C implementation, the other is forced to use the pure Python fallback
# implementation
py_warnings = import_fresh_module('warnings', blocked=['_warnings'])
c_warnings = import_fresh_module('warnings', fresh=['_warnings'])
```

Added in version 3.1.

test.support.import_helper.import_module(*name*, *deprecated=False*, *, *required_on=()*)

This function imports and returns the named module. Unlike a normal import, this function raises `unittest.SkipTest` if the module cannot be imported.

Module and package deprecation messages are suppressed during this import if *deprecated* is `True`. If a module is required on a platform but optional for others, set *required_on* to an iterable of platform prefixes which will be compared against `sys.platform`.

Added in version 3.1.

test.support.import_helper.modules_setup()

Return a copy of `sys.modules`.

test.support.import_helper.modules_cleanup(*oldmodules*)

Remove modules except for *oldmodules* and `encodings` in order to preserve internal cache.

test.support.import_helper.unload(*name*)

Delete *name* from `sys.modules`.

test.support.import_helper.make_legacy_pyc(*source*)

Move a **PEP 3147** [https://peps.python.org/pep-3147/]/**PEP 488** [https://peps.python.org/pep-0488/] pyc file to its legacy pyc location and return the file system path to the legacy pyc file. The *source* value is the file system path to the source file. It does not need to exist, however the PEP 3147/488 pyc file must exist.

class test.support.import_helper.CleanImport(**module_names*)

A context manager to force import to return a new module reference. This is useful for testing module-level behaviors, such as the emission of a `DeprecationWarning` on import. Example usage:

```
with CleanImport('foo'):
    importlib.import_module('foo')  # New reference.
```

class test.support.import_helper.DirsOnSysPath(**paths*)

A context manager to temporarily add directories to `sys.path`.

This makes a copy of `sys.path`, appends any directories given as positional arguments, then reverts `sys.path` to the copied settings when the context ends.

Note that *all* `sys.path` modifications in the body of the context manager, including replacement of the object, will be reverted at the end of the block.

`test.support.warnings_helper` — Utilities for warnings tests

The `test.support.warnings_helper` module provides support for warnings tests.

Added in version 3.10.

test.support.warnings_helper.ignore_warnings(*, *category*)

Suppress warnings that are instances of *category*, which must be `Warning` or a subclass. Roughly equivalent to `warnings.catch_warnings()` with `warnings.simplefilter('ignore', category=category)`. For example:

```
@warning_helper.ignore_warnings(category=DeprecationWarning)
def test_suppress_warning():
    # do something
```

Added in version 3.8.

test.support.warnings_helper.check_no_resource_warning(*testcase*)

Context manager to check that no `ResourceWarning` was raised. You must remove the object which may emit `ResourceWarning` before the end of the context manager.

test.support.warnings_helper.check_syntax_warning(*testcase*, *statement*, *errtext=''*, ***, *lineno=1*, *offset=None*)

Test for syntax warning in *statement* by attempting to compile *statement*. Test also that the `SyntaxWarning` is emitted only once, and that it will be converted to a `SyntaxError` when turned into error. *testcase* is the `unittest` instance for the test. *errtext* is the regular expression which should match the string representation of the emitted `SyntaxWarning` and raised `SyntaxError`. If *lineno* is not `None`, compares to the line of the warning and exception. If *offset* is not `None`, compares to the offset of the exception.

Added in version 3.8.

test.support.warnings_helper.check_warnings(**filters*, *quiet=True*)

A convenience wrapper for `warnings.catch_warnings()` that makes it easier to test that a warning was correctly raised. It is approximately equivalent to calling `warnings.catch_warnings(record=True)` with `warnings.simplefilter()` set to `always` and with the option to automatically validate the results that are recorded.

`check_warnings` accepts 2-tuples of the form `("message regexp", WarningCategory)` as positional arguments. If one or more *filters* are provided, or if the optional keyword argument *quiet* is `False`, it checks to make sure the warnings are as expected: each specified filter must match at least one of the warnings raised by the enclosed code or the test fails, and if any warnings are raised that do not match any of the specified filters the test fails. To disable the first of these checks, set *quiet* to `True`.

If no arguments are specified, it defaults to:

```
check_warnings(("", Warning), quiet=True)
```

In this case all warnings are caught and no errors are raised.

On entry to the context manager, a **`WarningRecorder`** instance is returned. The underlying warnings list from **`catch_warnings()`** is available via the recorder object's **`warnings`** attribute. As a convenience, the attributes of the object representing the most recent warning can also be accessed directly through the recorder object (see example below). If no warning has been raised, then any of the attributes that would otherwise be expected on an object representing a warning will return `None`.

The recorder object also has a **`reset()`** method, which clears the warnings list.

The context manager is designed to be used like this:

```
with check_warnings(("assertion is always true",
SyntaxWarning),
                    ("", UserWarning)):
    exec('assert(False, "Hey!")')
    warnings.warn(UserWarning("Hide me!"))
```

In this case if either warning was not raised, or some other warning was raised, **`check_warnings()`** would raise an error.

When a test needs to look more deeply into the warnings, rather than just checking whether or not they occurred, code like this can be used:

```
with check_warnings(quiet=True) as w:
    warnings.warn("foo")
    assert str(w.args[0]) == "foo"
    warnings.warn("bar")
    assert str(w.args[0]) == "bar"
    assert str(w.warnings[0].args[0]) == "foo"
    assert str(w.warnings[1].args[0]) == "bar"
    w.reset()
    assert len(w.warnings) == 0
```

Here all warnings will be caught, and the test code tests the captured warnings directly.

Changed in version 3.2: New optional arguments *filters* and *quiet*.

class test.support.warnings_helper.WarningsRecorder

Class used to record warnings for unit tests. See documentation of `check_warnings()` above for more details.

Debugging and Profiling

These libraries help you with Python development: the debugger enables you to step through code, analyze stack frames and set breakpoints etc., and the profilers run code and give you a detailed breakdown of execution times, allowing you to identify bottlenecks in your programs. Auditing events provide visibility into runtime behaviors that would otherwise require intrusive debugging or patching.

- Audit events table
- `bdb` — Debugger framework
- `faulthandler` — Dump the Python traceback
  - Dumping the traceback
  - Dumping the C stack
    - C Stack Compatibility
  - Fault handler state
  - Dumping the tracebacks after a timeout
  - Dumping the traceback on a user signal
  - Issue with file descriptors
  - Example
- `pdb` — The Python Debugger
  - Command-line interface
  - Debugger commands
- The Python Profilers
  - Introduction to the profilers
  - Instant User’s Manual
  - `profile` and `cProfile` Module Reference
  - The `Stats` Class
  - What Is Deterministic Profiling?
  - Limitations
  - Calibration
  - Using a custom timer

- `timeit` — Measure execution time of small code snippets
  - Basic Examples
  - Python Interface
  - Command-Line Interface
  - Examples
- `trace` — Trace or track Python statement execution
  - Command-Line Usage
    - Main options
    - Modifiers
    - Filters
  - Programmatic Interface
- `tracemalloc` — Trace memory allocations
  - Examples
    - Display the top 10
    - Compute differences
    - Get the traceback of a memory block
    - Pretty top
      - Record the current and peak size of all traced memory blocks
  - API
    - Functions
    - DomainFilter
    - Filter
    - Frame
    - Snapshot
    - Statistic
    - StatisticDiff
    - Trace
    - Traceback

Audit events table

This table contains all events raised by `sys.audit()` or `PySys_Audit()` calls throughout the CPython runtime and the standard library. These calls were added in 3.8 or later (see **PEP 578** [https://peps.python.org/pep-0578/]).

See `sys.addaudithook()` and `PySys_AddAuditHook()` for information on handling these events.

CPython implementation detail: This table is generated from the CPython documentation, and may not represent events raised by other implementations. See your runtime specific documentation for actual events raised.

Audit event	Arguments	References
_thread.start_new_thread	`function`, `args`, `kwargs`	[1]
array.__new__	`typecode`, `initializer`	[1]
builtins.breakpoint	`breakpointhook`	[1]
builtins.id	`id`	[1]
builtins.input	`prompt`	[1]
builtins.input/result	`result`	[1]

Audit event	Arguments	References
code.__new__	`code`, `filename`, `name`, `argcount`, `posonlyargcount`, `kwonlyargcount`, `nlocals`, `stacksize`, `flags`	[1]
compile	`source`, `filename`	[1]
cpython.PyConfig_Set	`name`, `value`	[1]
cpython.PyInterpreterState_Clear		[1]
cpython.PyInterpreterState_New		[1]
cpython._PySys_ClearAuditHooks		[1]
cpython.remote_debugger_script	`script_path`	[1]
cpython.run_command	`command`	[1]
cpython.run_file	`filename`	[1]
cpython.run_interactivehook	`hook`	[1]
cpython.run_module	`module-name`	[1]

Audit event	Arguments	References
cpython.run_startup	`filename`	[1]
cpython.run_stdin		[1][2][3]
ctypes.addressof	`obj`	[1]
ctypes.call_function	`func_pointer`, `arguments`	[1]
ctypes.cdata	`address`	[1]
ctypes.cdata/buffer	`pointer`, `size`, `offset`	[1][2]
ctypes.create_string_buffer	`init`, `size`	[1]
ctypes.create_unicode_buffer	`init`, `size`	[1]
ctypes.dlopen	`name`	[1]
ctypes.dlsym	`library`, `name`	[1]
ctypes.dlsym/handle	`handle`, `name`	[1]
ctypes.get_errno		[1]

Audit event	Arguments	References
ctypes.get_last_error		[1]
ctypes.memoryview_at	`address`, `size`, `readonly`	[1]
ctypes.set_errno	`errno`	[1]
ctypes.set_exception	`code`	[1]
ctypes.set_last_error	`error`	[1]
ctypes.string_at	`ptr`, `size`	[1]
ctypes.wstring_at	`ptr`, `size`	[1]
ensurepip.bootstrap	`root`	[1]
exec	`code_object`	[1][2]
fcntl.fcntl	`fd`, `cmd`, `arg`	[1]
fcntl.flock	`fd`, `operation`	[1]
fcntl.ioctl	`fd`, `request`, `arg`	[1]

Audit event	Arguments	References
fcntl.lockf	`fd`, `cmd`, `len`, `start`, `whence`	[1]
ftplib.connect	`self`, `host`, `port`	[1]
ftplib.sendcmd	`self`, `cmd`	[1][2]
function.__new__	`code`	[1]
gc.get_objects	`generation`	[1]
gc.get_referents	`objs`	[1]
gc.get_referrers	`objs`	[1]
glob.glob	`pathname`, `recursive`	[1][2]
glob.glob/2	`pathname`, `recursive`, `root_dir`, `dir_fd`	[1][2]
http.client.connect	`self`, `host`, `port`	[1]
http.client.send	`self`, `data`	[1]

Audit event	Arguments	References
imaplib.open	`self`, `host`, `port`	[1]
imaplib.send	`self`, `data`	[1]
import	`module`, `filename`, `sys.path`, `sys.meta_path`, `sys.path_hooks`	[1]
marshal.dumps	`value`, `version`	[1]
marshal.load		[1]
marshal.loads	`bytes`	[1]
mmap.__new__	`fileno`, `length`, `access`, `offset`	[1]
msvcrt.get_osfhandle	`fd`	[1]
msvcrt.locking	`fd`, `mode`, `nbytes`	[1]
msvcrt.open_osfhandle	`handle`, `flags`	[1]
object.__delattr__	`obj`, `name`	[1]

Audit event	Arguments	References
object.__getattr__	`obj`, `name`	[1]
object.__setattr__	`obj`, `name`, `value`	[1]
open	`path`, `mode`, `flags`	[1][2][3]
os.add_dll_directory	`path`	[1]
os.chdir	`path`	[1][2]
os.chflags	`path`, `flags`	[1][2]
os.chmod	`path`, `mode`, `dir_fd`	[1][2][3]
os.chown	`path`, `uid`, `gid`, `dir_fd`	[1][2][3]
os.exec	`path`, `args`, `env`	[1]
os.fork		[1]
os.forkpty		[1]
os.fwalk	`top`, `topdown`, `onerror`,	[1]

Audit event	Arguments	References
	`follow_symlinks`, `dir_fd`	
os.getxattr	`path`, `attribute`	[1]
os.kill	`pid`, `sig`	[1]
os.killpg	`pgid`, `sig`	[1]
os.link	`src`, `dst`, `src_dir_fd`, `dst_dir_fd`	[1]
os.listdir	`path`	[1]
os.listdrives		[1]
os.listmounts	`volume`	[1]
os.listvolumes		[1]
os.listxattr	`path`	[1]
os.lockf	`fd`, `cmd`, `len`	[1]
os.mkdir	`path`, `mode`, `dir_fd`	[1][2]

Audit event	**Arguments**	**References**
os.posix_spawn	`path`, `argv`, `env`	[1][2]
os.putenv	`key`, `value`	[1]
os.remove	`path`, `dir_fd`	[1][2][3]
os.removexattr	`path`, `attribute`	[1]
os.rename	`src`, `dst`, `src_dir_fd`, `dst_dir_fd`	[1][2][3]
os.rmdir	`path`, `dir_fd`	[1]
os.scandir	`path`	[1]
os.setxattr	`path`, `attribute`, `value`, `flags`	[1]
os.spawn	`mode`, `path`, `args`, `env`	[1]
os.startfile	`path`, `operation`	[1]
os.startfile/2	`path`, `operation`, `arguments`, `cwd`, `show_cmd`	[1]

Audit event	Arguments	References
os.symlink	`src`, `dst`, `dir_fd`	[1]
os.system	`command`	[1]
os.truncate	`fd`, `length`	[1][2]
os.unsetenv	`key`	[1]
os.utime	`path`, `times`, `ns`, `dir_fd`	[1]
os.walk	`top`, `topdown`, `onerror`, `followlinks`	[1]
pathlib.Path.glob	`self`, `pattern`	[1]
pathlib.Path.rglob	`self`, `pattern`	[1]
pdb.Pdb		[1]
pickle.find_class	`module`, `name`	[1]
poplib.connect	`self`, `host`, `port`	[1][2]
poplib.putline	`self`, `line`	[1][2]

Audit event	Arguments	References
pty.spawn	`argv`	[1]
resource.prlimit	`pid`, `resource`, `limits`	[1]
resource.setrlimit	`resource`, `limits`	[1]
setopencodehook		[1]
shutil.chown	`path`, `user`, `group`	[1]
shutil.copyfile	`src`, `dst`	[1][2][3]
shutil.copymode	`src`, `dst`	[1][2]
shutil.copystat	`src`, `dst`	[1][2]
shutil.copytree	`src`, `dst`	[1]
shutil.make_archive	`base_name`, `format`, `root_dir`, `base_dir`	[1]
shutil.move	`src`, `dst`	[1]

Audit event	Arguments	References
shutil.rmtree	`path`, `dir_fd`	[1]
shutil.unpack_archive	`filename`, `extract_dir`, `format`	[1]
signal.pthread_kill	`thread_id`, `signalnum`	[1]
smtplib.connect	`self`, `host`, `port`	[1]
smtplib.send	`self`, `data`	[1]
socket.__new__	`self`, `family`, `type`, `protocol`	[1]
socket.bind	`self`, `address`	[1]
socket.connect	`self`, `address`	[1][2]
socket.getaddrinfo	`host`, `port`, `family`, `type`, `protocol`	[1]
socket.gethostbyaddr	`ip_address`	[1]
socket.gethostbyname	`hostname`	[1][2]

Audit event	**Arguments**	**References**
socket.gethostname		[1]
socket.getnameinfo	`sockaddr`	[1]
socket.getservbyname	`servicename`, `protocolname`	[1]
socket.getservbyport	`port`, `protocolname`	[1]
socket.sendmsg	`self`, `address`	[1]
socket.sendto	`self`, `address`	[1]
socket.sethostname	`name`	[1]
sqlite3.connect	`database`	[1]
sqlite3.connect/handle	`connection_handle`	[1]
sqlite3.enable_load_extension	`connection`, `enabled`	[1]
sqlite3.load_extension	`connection`, `path`	[1]

Audit event	**Arguments**	**References**
subprocess.Popen	`executable`, `args`, `cwd`, `env`	[1]
sys._current_exceptions		[1]
sys._current_frames		[1]
sys._getframe	`frame`	[1]
sys._getframemodulename	`depth`	[1]
sys.addaudithook		[1][2]
sys.excepthook	`hook`, `type`, `value`, `traceback`	[1]
sys.monitoring.register_callback	`func`	[1]
sys.remote_exec	`pid`	[1]
sys.set_asyncgen_hooks_finalizer		[1]
sys.set_asyncgen_hooks_firstiter		[1]
sys.setprofile		[1]

Audit event	Arguments	References
sys.settrace		[1]
sys.unraisablehook	`hook`, `unraisable`	[1]
syslog.closelog		[1]
syslog.openlog	`ident`, `logoption`, `facility`	[1]
syslog.setlogmask	`maskpri`	[1]
syslog.syslog	`priority`, `message`	[1]
tempfile.mkdtemp	`fullpath`	[1][2]
tempfile.mkstemp	`fullpath`	[1][2][3]
time.sleep	`secs`	[1]
urllib.Request	`fullurl`, `data`, `headers`, `method`	[1]
webbrowser.open	`url`	[1]
winreg.ConnectRegistry	`computer_name`, `key`	[1]

Audit event	Arguments	References
winreg.CreateKey	`key`, `sub_key`, `access`	[1][2]
winreg.DeleteKey	`key`, `sub_key`, `access`	[1][2]
winreg.DeleteValue	`key`, `value`	[1]
winreg.DisableReflectionKey	`key`	[1]
winreg.EnableReflectionKey	`key`	[1]
winreg.EnumKey	`key`, `index`	[1]
winreg.EnumValue	`key`, `index`	[1]
winreg.ExpandEnvironmentStrings	`str`	[1]
winreg.LoadKey	`key`, `sub_key`, `file_name`	[1]
winreg.OpenKey	`key`, `sub_key`, `access`	[1]
winreg.OpenKey/result	`key`	[1][2][3]

Audit event	Arguments	References
winreg.PyHKEY.Detach	`key`	[1]
winreg.QueryInfoKey	`key`	[1]
winreg.QueryReflectionKey	`key`	[1]
winreg.QueryValue	`key`, `sub_key`, `value_name`	[1][2]
winreg.SaveKey	`key`, `file_name`	[1]
winreg.SetValue	`key`, `sub_key`, `type`, `value`	[1][2]

The following events are raised internally and do not correspond to any public API of CPython:

Audit event	Arguments
_winapi.CreateFile	`file_name`, `desired_access`, `share_mode`, `creation_disposition`, `flags_and_attributes`
_winapi.CreateJunction	`src_path`, `dst_path`
_winapi.CreateNamedPipe	`name`, `open_mode`, `pipe_mode`

Audit event	**Arguments**
_winapi.CreatePipe	
_winapi.CreateProcess	`application_name`, `command_line`, `current_directory`
_winapi.OpenProcess	`process_id`, `desired_access`
_winapi.TerminateProcess	`handle`, `exit_code`
_posixsubprocess.fork_exec	`exec_list`, `args`, `env`
ctypes.PyObj_FromPtr	`obj`

Added in version 3.14: The `_posixsubprocess.fork_exec` internal audit event.

`bdb` — Debugger framework

Source code: Lib/bdb.py
[https://github.com/python/cpython/tree/3.14/Lib/bdb.py]

The **`bdb`** module handles basic debugger functions, like setting breakpoints or managing execution via the debugger.

The following exception is defined:

exception bdb.BdbQuit

Exception raised by the **`Bdb`** class for quitting the debugger.

The **`bdb`** module also defines two classes:

class bdb.Breakpoint(*self*, *file*, *line*, *temporary=False*, *cond=None*, *funcname=None*)

This class implements temporary breakpoints, ignore counts, disabling and (re-)enabling, and conditionals.

Breakpoints are indexed by number through a list called **`bpbynumber`** and by `(file, line)` pairs through **`bplist`**. The former points to a single instance of class **`Breakpoint`**. The latter points to a list of such instances since there may be more than one breakpoint per line.

When creating a breakpoint, its associated **`file name`** should be in canonical form. If a **`funcname`** is defined, a breakpoint **`hit`** will be counted when the first line of that function is executed. A **`conditional`** breakpoint always counts a **`hit`**.

`Breakpoint` instances have the following methods:

deleteMe()

Delete the breakpoint from the list associated to a file/line. If it is the last breakpoint in that position, it also deletes the entry for the file/line.

enable()

Mark the breakpoint as enabled.

disable()

Mark the breakpoint as disabled.

bpformat()

Return a string with all the information about the breakpoint, nicely formatted:

- Breakpoint number.
- Temporary status (del or keep).
- File/line position.
- Break condition.
- Number of times to ignore.
- Number of times hit.

Added in version 3.2.

bpprint(*out=None*)

Print the output of `bpformat()` to the file *out*, or if it is `None`, to standard output.

`Breakpoint` instances have the following attributes:

file

File name of the `Breakpoint`.

line

Line number of the **`Breakpoint`** within **`file`**.

temporary

`True` if a **`Breakpoint`** at (file, line) is temporary.

cond

Condition for evaluating a **`Breakpoint`** at (file, line).

funcname

Function name that defines whether a **`Breakpoint`** is hit upon entering the function.

enabled

`True` if **`Breakpoint`** is enabled.

bpbynumber

Numeric index for a single instance of a **`Breakpoint`**.

bplist

Dictionary of **`Breakpoint`** instances indexed by (**`file`**, **`line`**) tuples.

ignore

Number of times to ignore a **`Breakpoint`**.

hits

Count of the number of times a **`Breakpoint`** has been hit.

class bdb.Bdb(*skip=None*, *backend='settrace'*)

The **`Bdb`** class acts as a generic Python debugger base class.

This class takes care of the details of the trace facility; a derived class should implement user interaction. The standard debugger class (**`pdb.Pdb`**) is an example.

The *skip* argument, if given, must be an iterable of glob-style module name patterns. The debugger will not step into frames that originate in a module that matches one of these patterns. Whether a frame is considered to originate in a certain module is determined by the `__name__` in the frame globals.

The *backend* argument specifies the backend to use for **`Bdb`**. It can be either `'settrace'` or `'monitoring'`. `'settrace'` uses **`sys.settrace()`** which has the best backward compatibility. The `'monitoring'` backend uses the new **`sys.monitoring`** that was introduced in Python 3.12, which can be much more efficient because it can disable unused events. We are trying to keep the exact interfaces for both backends, but there are some differences. The debugger developers are encouraged to use the `'monitoring'` backend to achieve better performance.

Changed in version 3.1: Added the *skip* parameter.

Changed in version 3.14: Added the *backend* parameter.

The following methods of **`Bdb`** normally don't need to be overridden.

canonic(*filename*)

Return canonical form of *filename*.

For real file names, the canonical form is an operating-system-dependent, **`case-normalized`** **`absolute path`**. A *filename* with angle brackets, such as `"<stdin>"` generated in interactive mode, is returned unchanged.

start_trace(*self*)

Start tracing. For `'settrace'` backend, this method is equivalent to `sys.settrace(self.trace_dispatch)`

Added in version 3.14.

stop_trace(*self*)

Stop tracing. For `'settrace'` backend, this method is equivalent to `sys.settrace(None)`

Added in version 3.14.

reset()

Set the **`botframe`**, **`stopframe`**, **`returnframe`** and **`quitting`** attributes with values ready to start debugging.

trace_dispatch(*frame*, *event*, *arg*)

This function is installed as the trace function of debugged frames. Its return value is the new trace function (in most cases, that is, itself).

The default implementation decides how to dispatch a frame, depending on the type of event (passed as a string) that is about to be executed. *event* can be one of the following:

- `"line"`: A new line of code is going to be executed.
- `"call"`: A function is about to be called, or another code block entered.
- `"return"`: A function or other code block is about to return.
- `"exception"`: An exception has occurred.
- `"c_call"`: A C function is about to be called.
- `"c_return"`: A C function has returned.
- `"c_exception"`: A C function has raised an exception.

For the Python events, specialized functions (see below) are called. For the C events, no action is taken.

The *arg* parameter depends on the previous event.

See the documentation for `sys.settrace()` for more information on the trace function. For more information on code and frame objects, refer to The standard type hierarchy.

dispatch_line(*frame*)

If the debugger should stop on the current line, invoke the `user_line()` method (which should be overridden in subclasses). Raise a `BdbQuit` exception if the `quitting` flag is set (which can be set from `user_line()`). Return a reference to the `trace_dispatch()` method for further tracing in that scope.

dispatch_call(*frame*, *arg*)

If the debugger should stop on this function call, invoke the `user_call()` method (which should be overridden in subclasses). Raise a `BdbQuit` exception if the `quitting` flag is set (which can be set from `user_call()`). Return a reference to the `trace_dispatch()` method for further tracing in that scope.

dispatch_return(*frame*, *arg*)

If the debugger should stop on this function return, invoke the `user_return()` method (which should be overridden in subclasses). Raise a `BdbQuit` exception if the `quitting` flag is set (which can be set from `user_return()`). Return a reference to the `trace_dispatch()` method for further tracing in that scope.

dispatch_exception(*frame*, *arg*)

If the debugger should stop at this exception, invokes the `user_exception()` method (which should be overridden in subclasses). Raise a `BdbQuit` exception if the `quitting` flag is set (which can be set from `user_exception()`). Return a reference to the `trace_dispatch()` method for further tracing in that scope.

Normally derived classes don’t override the following methods, but they may if they want to redefine the definition of stopping and breakpoints.

is_skipped_module(*module_name*)

Return `True` if *module_name* matches any skip pattern.

stop_here(*frame*)

Return `True` if *frame* is below the starting frame in the stack.

break_here(*frame*)

Return `True` if there is an effective breakpoint for this line.

Check whether a line or function breakpoint exists and is in effect. Delete temporary breakpoints based on information from `effective()`.

break_anywhere(*frame*)

Return `True` if any breakpoint exists for *frame*’s filename.

Derived classes should override these methods to gain control over debugger operation.

user_call(*frame*, *argument_list*)

Called from `dispatch_call()` if a break might stop inside the called function.

argument_list is not used anymore and will always be `None`. The argument is kept for backwards compatibility.

user_line(*frame*)

Called from `dispatch_line()` when either `stop_here()` or `break_here()` returns `True`.

user_return(*frame*, *return_value*)

Called from `dispatch_return()` when `stop_here()` returns `True`.

user_exception(*frame*, *exc_info*)

Called from `dispatch_exception()` when `stop_here()` returns `True`.

do_clear(*arg*)

Handle how a breakpoint must be removed when it is a temporary one.

This method must be implemented by derived classes.

Derived classes and clients can call the following methods to affect the stepping state.

set_step()

Stop after one line of code.

set_next(*frame*)

Stop on the next line in or below the given frame.

set_return(*frame*)

Stop when returning from the given frame.

set_until(*frame*, *lineno=None*)

Stop when the line with the *lineno* greater than the current one is reached or when returning from current frame.

set_trace([*frame*])

Start debugging from *frame*. If *frame* is not specified, debugging starts from caller’s frame.

Changed in version 3.13: `set_trace()` will enter the debugger immediately, rather than on the next line of code to be executed.

set_continue()

Stop only at breakpoints or when finished. If there are no breakpoints, set the system trace function to `None`.

set_quit()

Set the **`quitting`** attribute to `True`. This raises `BdbQuit` in the next call to one of the **`dispatch_*()`** methods.

Derived classes and clients can call the following methods to manipulate breakpoints. These methods return a string containing an error message if something went wrong, or `None` if all is well.

set_break(*filename*, *lineno*, *temporary=False*, *cond=None*, *funcname=None*)

Set a new breakpoint. If the *lineno* line doesn’t exist for the *filename* passed as argument, return an error message. The *filename* should be in canonical form, as described in the `canonic()` method.

clear_break(*filename*, *lineno*)

Delete the breakpoints in *filename* and *lineno*. If none were set, return an error message.

clear_bpbynumber(*arg*)

Delete the breakpoint which has the index *arg* in the `Breakpoint.bpbynumber`. If *arg* is not numeric or out of range, return an error message.

clear_all_file_breaks(*filename*)

Delete all breakpoints in *filename*. If none were set, return an error message.

clear_all_breaks()

Delete all existing breakpoints. If none were set, return an error message.

get_bpbynumber(*arg*)

Return a breakpoint specified by the given number. If *arg* is a string, it will be converted to a number. If *arg* is a non-numeric string, if the given breakpoint never existed or has been deleted, a `ValueError` is raised.

Added in version 3.2.

get_break(*filename*, *lineno*)

Return `True` if there is a breakpoint for *lineno* in *filename*.

get_breaks(*filename*, *lineno*)

Return all breakpoints for *lineno* in *filename*, or an empty list if none are set.

get_file_breaks(*filename*)

Return all breakpoints in *filename*, or an empty list if none are set.

get_all_breaks()

Return all breakpoints that are set.

Derived classes and clients can call the following methods to disable and restart events to achieve better performance. These methods only work when using the `'monitoring'` backend.

disable_current_event()

Disable the current event until the next time `restart_events()` is called. This is helpful when the debugger is not interested in the current line.

Added in version 3.14.

restart_events()

Restart all the disabled events. This function is automatically called in `dispatch_*` methods after `user_*` methods are called. If the `dispatch_*` methods are not overridden, the disabled events will be restarted after each user interaction.

Added in version 3.14.

Derived classes and clients can call the following methods to get a data structure representing a stack trace.

get_stack(*f*, *t*)

Return a list of (frame, lineno) tuples in a stack trace, and a size.

The most recently called frame is last in the list. The size is the number of frames below the frame where the debugger was invoked.

format_stack_entry(*frame_lineno*, *lprefix=': '*)

Return a string with information about a stack entry, which is a `(frame, lineno)` tuple. The return string contains:

- The canonical filename which contains the frame.
- The function name or `"<lambda>"`.
- The input arguments.
- The return value.
- The line of code (if it exists).

The following two methods can be called by clients to use a debugger to debug a statement, given as a string.

run(*cmd*, *globals=None*, *locals=None*)

Debug a statement executed via the `exec()` function. *globals* defaults to **`__main__.__dict__`**, *locals* defaults to *globals*.

runeval(*expr*, *globals=None*, *locals=None*)

Debug an expression executed via the `eval()` function. *globals* and *locals* have the same meaning as in `run()`.

runctx(*cmd*, *globals*, *locals*)

For backwards compatibility. Calls the `run()` method.

runcall(*func*, */*, **args*, ***kwds*)

Debug a single function call, and return its result.

Finally, the module defines the following functions:

bdb.checkfuncname(*b*, *frame*)

Return `True` if we should break here, depending on the way the `Breakpoint` *b* was set.

If it was set via line number, it checks if **`b.line`** is the same as the one in *frame*. If the breakpoint was set via **`function name`**, we have to check we are in the right *frame* (the right function) and if we are on its first executable line.

bdb.effective(*file*, *line*, *frame*)

Return `(active breakpoint, delete temporary flag)` or `(None, None)` as the breakpoint to act upon.

The *active breakpoint* is the first entry in **`bplist`** for the (**`file`**, **`line`**) (which must exist) that is **`enabled`**, for which **`checkfuncname()`** is true, and that has neither a false **`condition`** nor positive **`ignore`** count. The *flag*, meaning that a temporary breakpoint should be deleted, is `False` only when the **`cond`** cannot be evaluated (in which case, **`ignore`** count is ignored).

If no such entry exists, then `(None, None)` is returned.

bdb.set_trace()

Start debugging with a **`Bdb`** instance from caller’s frame.

`faulthandler` — Dump the Python traceback

Added in version 3.3.

This module contains functions to dump Python tracebacks explicitly, on a fault, after a timeout, or on a user signal. Call **`faulthandler.enable()`** to install fault handlers for the **`SIGSEGV`**, **`SIGFPE`**, **`SIGABRT`**, **`SIGBUS`**, and **`SIGILL`** signals. You can also enable them at startup by setting the **`PYTHONFAULTHANDLER`** environment variable or by using the **`-X`** `faulthandler` command line option.

The fault handler is compatible with system fault handlers like Apport or the Windows fault handler. The module uses an alternative stack for signal handlers if the **`sigaltstack()`** function is available. This allows it to dump the traceback even on a stack overflow.

The fault handler is called on catastrophic cases and therefore can only use signal-safe functions (e.g. it cannot allocate memory on the heap). Because of this limitation traceback dumping is minimal compared to normal Python tracebacks:

- Only ASCII is supported. The `backslashreplace` error handler is used on encoding.
- Each string is limited to 500 characters.
- Only the filename, the function name and the line number are displayed. (no source code)
- It is limited to 100 frames and 100 threads.

- The order is reversed: the most recent call is shown first.

By default, the Python traceback is written to `sys.stderr`. To see tracebacks, applications must be run in the terminal. A log file can alternatively be passed to `faulthandler.enable()`.

The module is implemented in C, so tracebacks can be dumped on a crash or when Python is deadlocked.

The Python Development Mode calls `faulthandler.enable()` at Python startup.

See also

Module `pdb`
Interactive source code debugger for Python programs.

Module `traceback`
Standard interface to extract, format and print stack traces of Python programs.

Dumping the traceback

faulthandler.dump_traceback(*file=sys.stderr*, *all_threads=True*)

Dump the tracebacks of all threads into *file*. If *all_threads* is `False`, dump only the current thread.

See also

`traceback.print_tb()`, which can be used to print a traceback object.

Changed in version 3.5: Added support for passing file descriptor to this function.

Dumping the C stack

Added in version 3.14.

faulthandler.dump_c_stack(*file=sys.stderr*)

Dump the C stack trace of the current thread into *file*.

If the Python build does not support it or the operating system does not provide a stack trace, then this prints an error in place of a dumped C stack.

C Stack Compatibility

If the system does not support the C-level *backtrace(3)* *[https://manpages.debian.org/backtrace(3)]* or *dladdr1(3)* *[https://manpages.debian.org/dladdr1(3)]*, then C stack dumps will not work. An error will be printed instead of the stack.

Additionally, some compilers do not support CPython's implementation of C stack dumps. As a result, a different error may be printed instead of the stack, even if the operating system supports dumping stacks.

Note

Dumping C stacks can be arbitrarily slow, depending on the DWARF level of the binaries in the call stack.

Fault handler state

faulthandler.enable(*file=sys.stderr*, *all_threads=True*, *c_stack=True*)

Enable the fault handler: install handlers for the **`SIGSEGV`**, **`SIGFPE`**, **`SIGABRT`**, **`SIGBUS`** and **`SIGILL`** signals to dump the Python traceback. If *all_threads* is `True`, produce tracebacks for every running thread. Otherwise, dump only the current thread.

The *file* must be kept open until the fault handler is disabled: see issue with file descriptors.

If *c_stack* is `True`, then the C stack trace is printed after the Python traceback, unless the system does not support it. See **`dump_c_stack()`** for more information on compatibility.

Changed in version 3.5: Added support for passing file descriptor to this function.

Changed in version 3.6: On Windows, a handler for Windows exception is also installed.

Changed in version 3.10: The dump now mentions if a garbage collector collection is running if *all_threads* is true.

Changed in version 3.14: Only the current thread is dumped if the GIL is disabled to prevent the risk of data races.

Changed in version 3.14: The dump now displays the C stack trace if *c_stack* is true.

faulthandler.disable()

Disable the fault handler: uninstall the signal handlers installed by **`enable()`**.

faulthandler.is_enabled()

Check if the fault handler is enabled.

Dumping the tracebacks after a timeout

faulthandler.dump_traceback_later(*timeout*, *repeat=False*, *file=sys.stderr*, *exit=False*)

Dump the tracebacks of all threads, after a timeout of *timeout* seconds, or every *timeout* seconds if *repeat* is `True`. If *exit* is `True`, call **`_exit()`** with status=1 after dumping the tracebacks. (Note **`_exit()`** exits the process immediately, which means it doesn't do any cleanup like flushing file buffers.) If the function is called twice, the new call replaces previous parameters and resets the timeout. The timer has a sub-second resolution.

The *file* must be kept open until the traceback is dumped or **`cancel_dump_traceback_later()`** is called: see issue with file descriptors.

This function is implemented using a watchdog thread.

Changed in version 3.5: Added support for passing file descriptor to this function.

Changed in version 3.7: This function is now always available.

faulthandler.cancel_dump_traceback_later()

Cancel the last call to **`dump_traceback_later()`**.

Dumping the traceback on a user signal

faulthandler.register(*signum*, *file=sys.stderr*, *all_threads=True*, *chain=False*)

Register a user signal: install a handler for the *signum* signal to dump the traceback of all threads, or of the current thread if *all_threads* is `False`, into *file*. Call the previous handler if chain is `True`.

The *file* must be kept open until the signal is unregistered by **`unregister()`**: see issue with file descriptors.

Not available on Windows.

Changed in version 3.5: Added support for passing file descriptor to this function.

faulthandler.unregister(*signum*)

Unregister a user signal: uninstall the handler of the *signum* signal installed by **`register()`**. Return `True` if the signal was registered, `False` otherwise.

Not available on Windows.

Issue with file descriptors

`enable()`, **`dump_traceback_later()`** and **`register()`** keep the file descriptor of their *file* argument. If the file is closed and its file descriptor is reused by a new file, or if **`os.dup2()`** is used to replace the file descriptor, the traceback will be written into a different file. Call these functions again each time that the file is replaced.

Example

Example of a segmentation fault on Linux with and without enabling the fault handler:

```
$ python -c "import ctypes; ctypes.string_at(0)"
Segmentation fault

$ python -q -X faulthandler
>>> import ctypes
>>> ctypes.string_at(0)
Fatal Python error: Segmentation fault

Current thread 0x00007fb899f39700 (most recent call first):
  File "/opt/python/Lib/ctypes/__init__.py", line 486 in
string_at
  File "<stdin>", line 1 in <module>

Current thread's C stack trace (most recent call first):
  Binary file "/opt/python/python", at _Py_DumpStack+0x42
[0x5b27f7d7147e]
  Binary file "/opt/python/python", at +0x32dcbd
[0x5b27f7d85cbd]
  Binary file "/opt/python/python", at +0x32df8a
[0x5b27f7d85f8a]
  Binary file "/usr/lib/libc.so.6", at +0x3def0
[0x77b73226bef0]
  Binary file "/usr/lib/libc.so.6", at +0x17ef9c
[0x77b7323acf9c]
  Binary file "/opt/python/build/lib.linux-x86_64-
3.14/_ctypes.cpython-314d-x86_64-linux-gnu.so", at +0xcdf6
[0x77b7315dddf6]
  Binary file "/usr/lib/libffi.so.8", at +0x7976
[0x77b73158f976]
  Binary file "/usr/lib/libffi.so.8", at +0x413c
[0x77b73158c13c]
  Binary file "/usr/lib/libffi.so.8", at ffi_call+0x12e
[0x77b73158ef0e]
  Binary file "/opt/python/build/lib.linux-x86_64-
3.14/_ctypes.cpython-314d-x86_64-linux-gnu.so", at +0x15a33
[0x77b7315e6a33]
  Binary file "/opt/python/build/lib.linux-x86_64-
3.14/_ctypes.cpython-314d-x86_64-linux-gnu.so", at +0x164fa
[0x77b7315e74fa]
  Binary file "/opt/python/build/lib.linux-x86_64-
3.14/_ctypes.cpython-314d-x86_64-linux-gnu.so", at +0xc624
[0x77b7315dd624]
  Binary file "/opt/python/python", at
```

```
_PyObject_MakeTpCall+0xce [0x5b27f7b73883]
  Binary file "/opt/python/python", at +0x11bab6
[0x5b27f7b73ab6]
  Binary file "/opt/python/python", at PyObject_Vectorcall+0x23
[0x5b27f7b73b04]
  Binary file "/opt/python/python", at
_PyEval_EvalFrameDefault+0x490c [0x5b27f7cbb302]
  Binary file "/opt/python/python", at +0x2818e6
[0x5b27f7cd98e6]
  Binary file "/opt/python/python", at +0x281aab
[0x5b27f7cd9aab]
  Binary file "/opt/python/python", at PyEval_EvalCode+0xc5
[0x5b27f7cd9ba3]
  Binary file "/opt/python/python", at +0x255957
[0x5b27f7cad957]
  Binary file "/opt/python/python", at +0x255ab4
[0x5b27f7cadab4]
  Binary file "/opt/python/python", at
_PyEval_EvalFrameDefault+0x6c3e [0x5b27f7cbd634]
  Binary file "/opt/python/python", at +0x2818e6
[0x5b27f7cd98e6]
  Binary file "/opt/python/python", at +0x281aab
[0x5b27f7cd9aab]
  Binary file "/opt/python/python", at +0x11b6e1
[0x5b27f7b736e1]
  Binary file "/opt/python/python", at +0x11d348
[0x5b27f7b75348]
  Binary file "/opt/python/python", at +0x11d626
[0x5b27f7b75626]
  Binary file "/opt/python/python", at PyObject_Call+0x20
[0x5b27f7b7565e]
  Binary file "/opt/python/python", at +0x32a67a
[0x5b27f7d8267a]
  Binary file "/opt/python/python", at +0x32a7f8
[0x5b27f7d827f8]
  Binary file "/opt/python/python", at +0x32ac1b
[0x5b27f7d82c1b]
  Binary file "/opt/python/python", at Py_RunMain+0x31
[0x5b27f7d82ebe]
  <truncated rest of calls>
Segmentation fault
```

pdb — The Python Debugger

Source code: Lib/pdb.py [https://github.com/python/cpython/tree/3.14/Lib/pdb.py]

The module `pdb` defines an interactive source code debugger for Python programs. It supports setting (conditional) breakpoints and single stepping at the source line level, inspection of stack frames, source code listing, and evaluation of arbitrary Python code in the context of any stack frame. It also supports post-mortem debugging and can be called under program control.

The debugger is extensible – it is actually defined as the class `Pdb`. This is currently undocumented but easily understood by reading the source. The extension interface uses the modules `bdb` and `cmd`.

See also

Module `faulthandler`
Used to dump Python tracebacks explicitly, on a fault, after a timeout, or on a user signal.

Module `traceback`
Standard interface to extract, format and print stack traces of Python programs.

The typical usage to break into the debugger is to insert:

```
import pdb; pdb.set_trace()
```

Or:

```
breakpoint()
```

at the location you want to break into the debugger, and then run the program. You can then step through the code following this statement, and continue running without the debugger using the [`continue`](#) command.

Changed in version 3.7: The built-in [`breakpoint()`](#), when called with defaults, can be used instead of `import pdb; pdb.set_trace()`.

```
def double(x):
   breakpoint()
   return x * 2
val = 3
print(f"{val} * 2 is {double(val)}")
```

The debugger’s prompt is `(Pdb)`, which is the indicator that you are in debug mode:

```
> ...(2)double()
-> breakpoint()
(Pdb) p x
3
(Pdb) continue
3 * 2 is 6
```

Changed in version 3.3: Tab-completion via the [`readline`](#) module is available for commands and command arguments, e.g. the current global and local names are offered as arguments of the `p` command.

Command-line interface

You can also invoke **`pdb`** from the command line to debug other scripts. For example:

```
python -m pdb [-c command] (-m module | -p pid | pyfile) [args
...]
```

When invoked as a module, pdb will automatically enter post-mortem debugging if the program being debugged exits abnormally. After post-mortem debugging (or after normal exit of the program), pdb will restart the program. Automatic restarting preserves pdb’s state (such as breakpoints) and in most cases is more useful than quitting the debugger upon program’s exit.

-c, --command <command>

To execute commands as if given in a `.pdbrc` file; see Debugger commands.

Changed in version 3.2: Added the `-c` option.

-m <module>

To execute modules similar to the way `python -m` does. As with a script, the debugger will pause execution just before the first line of the module.

Changed in version 3.7: Added the `-m` option.

-p, --pid <pid>

Attach to the process with the specified PID.

Added in version 3.14.

To attach to a running Python process for remote debugging, use the `-p` or `--pid` option with the target process’s PID:

```
python -m pdb -p 1234
```

Note

Attaching to a process that is blocked in a system call or waiting for I/O will only work once the next bytecode instruction is executed or when the process receives a signal.

Typical usage to execute a statement under control of the debugger is:

```
>>> import pdb
>>> def f(x):
...     print(1 / x)
>>> pdb.run("f(2)")
> <string>(1)<module>()
(Pdb) continue
0.5
>>>
```

The typical usage to inspect a crashed program is:

```
>>> import pdb
>>> def f(x):
...     print(1 / x)
...
>>> f(0)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 2, in f
ZeroDivisionError: division by zero
>>> pdb.pm()
> <stdin>(2)f()
(Pdb) p x
0
(Pdb)
```

Changed in version 3.13: The implementation of **PEP 667** [https://peps.python.org/pep-0667/] means that name assignments made via `pdb` will immediately affect the active scope, even when running inside an optimized scope.

The module defines the following functions; each enters the debugger in a slightly different way:

pdb.run(*statement*, *globals=None*, *locals=None*)

Execute the *statement* (given as a string or a code object) under debugger control. The debugger prompt appears before any code is executed; you can set breakpoints and type **`continue`**, or you can step through the statement using **`step`** or **`next`** (all these commands are explained below). The optional *globals* and *locals* arguments specify the environment in which the code is executed; by default the dictionary of the module **`__main__`** is used. (See the explanation of the built-in **`exec()`** or **`eval()`** functions.)

pdb.runeval(*expression*, *globals=None*, *locals=None*)

Evaluate the *expression* (given as a string or a code object) under debugger control. When **`runeval()`** returns, it returns the value of the *expression*. Otherwise this function is similar to **`run()`**.

pdb.runcall(*function*, **args*, ***kwds*)

Call the *function* (a function or method object, not a string) with the given arguments. When **`runcall()`** returns, it returns whatever the function call returned. The debugger prompt appears as soon as the function is entered.

pdb.set_trace(*, *header=None*, *commands=None*)

Enter the debugger at the calling stack frame. This is useful to hard-code a breakpoint at a given point in a program, even if the code is not otherwise being debugged (e.g. when an assertion fails). If given, *header* is printed to the console just before debugging begins. The *commands* argument, if given, is a list of commands to execute when the debugger starts.

Changed in version 3.7: The keyword-only argument *header*.

Changed in version 3.13: `set_trace()` will enter the debugger immediately, rather than on the next line of code to be executed.

Added in version 3.14: The *commands* argument.

awaitable pdb.set_trace_async(*, *header=None*, *commands=None*)

async version of `set_trace()`. This function should be used inside an async function with `await`.

```
async def f():
    await pdb.set_trace_async()
```

`await` statements are supported if the debugger is invoked by this function.

Added in version 3.14.

pdb.post_mortem(*t=None*)

Enter post-mortem debugging of the given exception or traceback object. If no value is given, it uses the exception that is currently being handled, or raises `ValueError` if there isn’t one.

Changed in version 3.13: Support for exception objects was added.

pdb.pm()

Enter post-mortem debugging of the exception found in `sys.last_exc`.

pdb.set_default_backend(*backend*)

There are two supported backends for pdb: `'settrace'` and `'monitoring'`. See `bdb.Bdb` for details. The user can set the default backend to use if none is specified when instantiating `Pdb`. If no backend is specified, the default is `'settrace'`.

Note

`breakpoint()` and `set_trace()` will not be affected by this function. They always use `'monitoring'` backend.

Added in version 3.14.

pdb.get_default_backend()

Returns the default backend for pdb.

Added in version 3.14.

The `run*` functions and `set_trace()` are aliases for instantiating the `Pdb` class and calling the method of the same name. If you want to access further features, you have to do this yourself:

class pdb.Pdb(*completekey='tab'*, *stdin=None*, *stdout=None*, *skip=None*, *nosigint=False*, *readrc=True*, *mode=None*, *backend=None*, *colorize=False*)

`Pdb` is the debugger class.

The *completekey*, *stdin* and *stdout* arguments are passed to the underlying `cmd.Cmd` class; see the description there.

The *skip* argument, if given, must be an iterable of glob-style module name patterns. The debugger will not step into frames that originate in a module that matches one of these patterns. [[1]]

By default, Pdb sets a handler for the SIGINT signal (which is sent when the user presses `Ctrl-C` on the console) when you give a **`continue`** command. This allows you to break into the debugger again by pressing `Ctrl-C`. If you want Pdb not to touch the SIGINT handler, set *nosigint* to true.

The *readrc* argument defaults to true and controls whether Pdb will load .pdbrc files from the filesystem.

The *mode* argument specifies how the debugger was invoked. It impacts the workings of some debugger commands. Valid values are `'inline'` (used by the breakpoint() builtin), `'cli'` (used by the command line invocation) or `None` (for backwards compatible behaviour, as before the *mode* argument was added).

The *backend* argument specifies the backend to use for the debugger. If `None` is passed, the default backend will be used. See **`set_default_backend()`**. Otherwise the supported backends are `'settrace'` and `'monitoring'`.

The *colorize* argument, if set to `True`, will enable colorized output in the debugger, if color is supported. This will highlight source code displayed in pdb.

Example call to enable tracing with *skip*:

```
import pdb; pdb.Pdb(skip=['django.*']).set_trace()
```

Raises an auditing event `pdb.Pdb` with no arguments.

Changed in version 3.1: Added the *skip* parameter.

Changed in version 3.2: Added the *nosigint* parameter. Previously, a SIGINT handler was never set by Pdb.

Changed in version 3.6: The *readrc* argument.

Added in version 3.14: Added the *mode* argument.

Added in version 3.14: Added the *backend* argument.

Added in version 3.14: Added the *colorize* argument.

Changed in version 3.14: Inline breakpoints like `breakpoint()` or `pdb.set_trace()` will always stop the program at calling frame, ignoring the *skip* pattern (if any).

run(*statement*, *globals=None*, *locals=None*)
runeval(*expression*, *globals=None*, *locals=None*)
runcall(*function*, **args*, ***kwds*)
set_trace()

See the documentation for the functions explained above.

Debugger commands

The commands recognized by the debugger are listed below. Most commands can be abbreviated to one or two letters as indicated; e.g. `h(elp)` means that either `h` or `help` can be used to enter the help command (but not `he` or `hel`, nor `H` or `Help` or `HELP`). Arguments to commands must be separated by whitespace (spaces or tabs). Optional arguments are enclosed in square brackets (`[]`) in the command syntax; the square brackets must not be typed.

Alternatives in the command syntax are separated by a vertical bar (`|`).

Entering a blank line repeats the last command entered. Exception: if the last command was a **`list`** command, the next 11 lines are listed.

Commands that the debugger doesn't recognize are assumed to be Python statements and are executed in the context of the program being debugged. Python statements can also be prefixed with an exclamation point (`!`). This is a powerful way to inspect the program being debugged; it is even possible to change a variable or call a function. When an exception occurs in such a statement, the exception name is printed but the debugger's state is not changed.

Changed in version 3.13: Expressions/Statements whose prefix is a pdb command are now correctly identified and executed.

The debugger supports aliases. Aliases can have parameters which allows one a certain level of adaptability to the context under examination.

Multiple commands may be entered on a single line, separated by `;;`. (A single `;` is not used as it is the separator for multiple commands in a line that is passed to the Python parser.) No intelligence is applied to separating the commands; the input is split at the first `;;` pair, even if it is in the middle of a quoted string. A workaround for strings with double semicolons is to use implicit string concatenation `';'';'` or `";"";"`.

To set a temporary global variable, use a *convenience variable*. A *convenience variable* is a variable whose name starts with `$`. For example, `$foo = 1` sets a global variable `$foo` which you can use in the debugger session. The *convenience variables* are cleared when the program resumes execution so it’s less likely to interfere with your program compared to using normal variables like `foo = 1`.

There are four preset *convenience variables*:

- `$_frame`: the current frame you are debugging
- `$_retval`: the return value if the frame is returning
- `$_exception`: the exception if the frame is raising an exception
- `$_asynctask`: the asyncio task if pdb stops in an async function

Added in version 3.12: Added the *convenience variable* feature.

Added in version 3.14: Added the `$_asynctask` convenience variable.

If a file `.pdbrc` exists in the user’s home directory or in the current directory, it is read with `'utf-8'` encoding and executed as if it had been typed at the debugger prompt, with the exception that empty lines and lines starting with `#` are ignored. This is particularly useful for aliases. If both files exist, the one in the home directory is read first and aliases defined there can be overridden by the local file.

Changed in version 3.2: `.pdbrc` can now contain commands that continue debugging, such as **`continue`** or **`next`**. Previously, these commands had no effect.

Changed in version 3.11: `.pdbrc` is now read with `'utf-8'` encoding. Previously, it was read with the system locale encoding.

h(elp) [command]

Without argument, print the list of available commands. With a *command* as argument, print help about that command. `help pdb` displays the full documentation (the docstring of the `pdb` module). Since the *command* argument must be an identifier, `help exec` must be entered to get help on the `!` command.

w(here) [count]

Print a stack trace, with the most recent frame at the bottom. if *count* is 0, print the current frame entry. If *count* is negative, print the least recent - *count* frames. If *count* is positive, print the most recent *count* frames. An arrow (`>`) indicates the current frame, which determines the context of most commands.

Changed in version 3.14: count argument is added.

d(own) [count]

Move the current frame *count* (default one) levels down in the stack trace (to a newer frame).

u(p) [count]

Move the current frame *count* (default one) levels up in the stack trace (to an older frame).

b(reak) [([filename:]lineno | function) [, condition]]

With a *lineno* argument, set a break at line *lineno* in the current file. The line number may be prefixed with a *filename* and a colon, to specify a breakpoint in another file (possibly one that hasn't been loaded yet). The file is searched on `sys.path`. Acceptable forms of *filename* are `/abspath/to/file.py`, `relpath/file.py`, `module` and `package.module`.

With a *function* argument, set a break at the first executable statement within that function. *function* can be any expression that evaluates to a function in the current namespace.

If a second argument is present, it is an expression which must evaluate to true before the breakpoint is honored.

Without argument, list all breaks, including for each breakpoint, the number of times that breakpoint has been hit, the current ignore count, and the associated condition if any.

Each breakpoint is assigned a number to which all the other breakpoint commands refer.

tbreak [([filename:]lineno | function) [, condition]]

Temporary breakpoint, which is removed automatically when it is first hit. The arguments are the same as for `break`.

cl(ear) [filename:lineno | bpnumber ...]

With a *filename:lineno* argument, clear all the breakpoints at this line. With a space separated list of breakpoint numbers, clear those breakpoints. Without argument, clear all breaks (but first ask confirmation).

disable bpnumber [bpnumber ...]

Disable the breakpoints given as a space separated list of breakpoint numbers. Disabling a breakpoint means it cannot cause the program to stop execution, but unlike clearing a breakpoint, it remains in the list of breakpoints and can be (re-)enabled.

enable bpnumber [bpnumber ...]

Enable the breakpoints specified.

ignore bpnumber [count]

Set the ignore count for the given breakpoint number. If *count* is omitted, the ignore count is set to 0. A breakpoint becomes active when the ignore count is zero. When non-zero, the *count* is decremented each time the breakpoint is reached and the breakpoint is not disabled and any associated condition evaluates to true.

condition bpnumber [condition]

Set a new *condition* for the breakpoint, an expression which must evaluate to true before the breakpoint is honored. If *condition* is absent, any existing condition is removed; i.e., the breakpoint is made unconditional.

commands [bpnumber]

Specify a list of commands for breakpoint number *bpnumber*. The commands themselves appear on the following lines. Type a line containing just `end` to terminate the commands. An example:

```
(Pdb) commands 1
(com) p some_variable
(com) end
(Pdb)
```

To remove all commands from a breakpoint, type `commands` and follow it immediately with `end`; that is, give no commands.

With no *bpnumber* argument, `commands` refers to the last breakpoint set.

You can use breakpoint commands to start your program up again. Simply use the **`continue`** command, or **`step`**, or any other command that resumes execution.

Specifying any command resuming execution (currently **`continue`**, **`step`**, **`next`**, **`return`**, **`until`**, **`jump`**, **`quit`** and their abbreviations) terminates the command list (as if that command was immediately followed by end). This is because any time you resume execution (even with a simple next or step), you may encounter another breakpoint—which could have its own command list, leading to ambiguities about which list to execute.

If the list of commands contains the `silent` command, or a command that resumes execution, then the breakpoint message containing information about the frame is not displayed.

Changed in version 3.14: Frame information will not be displayed if a command that resumes execution is present in the command list.

s(tep)

Execute the current line, stop at the first possible occasion (either in a function that is called or on the next line in the current function).

n(ext)

Continue execution until the next line in the current function is reached or it returns. (The difference between **`next`** and **`step`** is that **`step`** stops inside a called function, while **`next`** executes called functions at (nearly) full speed, only stopping at the next line in the current function.)

unt(il) [lineno]

Without argument, continue execution until the line with a number greater than the current one is reached.

With *lineno*, continue execution until a line with a number greater or equal to *lineno* is reached. In both cases, also stop when the current frame returns.

Changed in version 3.2: Allow giving an explicit line number.

r(eturn)

Continue execution until the current function returns.

c(ont(inue))

Continue execution, only stop when a breakpoint is encountered.

j(ump) lineno

Set the next line that will be executed. Only available in the bottom-most frame. This lets you jump back and execute code again, or jump forward to skip code that you don’t want to run.

It should be noted that not all jumps are allowed – for instance it is not possible to jump into the middle of a **`for`** loop or out of a **`finally`** clause.

l(ist) [first[, last]]

List source code for the current file. Without arguments, list 11 lines around the current line or continue the previous listing. With `.` as argument, list 11 lines around the current line. With one argument, list 11 lines around at that line. With two arguments, list the given range; if the second argument is less than the first, it is interpreted as a count.

The current line in the current frame is indicated by `->`. If an exception is being debugged, the line where the exception was originally raised or propagated is indicated by `>>`, if it differs from the current line.

Changed in version 3.2: Added the `>>` marker.

ll | longlist

List all source code for the current function or frame. Interesting lines are marked as for `list`.

Added in version 3.2.

a(rgs)

Print the arguments of the current function and their current values.

p expression

Evaluate *expression* in the current context and print its value.

Note

`print()` can also be used, but is not a debugger command — this executes the Python `print()` function.

pp expression

Like the `p` command, except the value of *expression* is pretty-printed using the `pprint` module.

whatis expression

Print the type of *expression*.

source expression

Try to get source code of *expression* and display it.

Added in version 3.2.

display [expression]

Display the value of *expression* if it changed, each time execution stops in the current frame.

Without *expression*, list all display expressions for the current frame.

Note

Display evaluates *expression* and compares to the result of the previous evaluation of *expression*, so when the result is mutable, display may not be able to pick up the changes.

Example:

```
lst = []
breakpoint()
pass
lst.append(1)
print(lst)
```

Display won't realize `lst` has been changed because the result of evaluation is modified in place by `lst.append(1)` before being compared:

```
> example.py(3)<module>()
-> pass
(Pdb) display lst
display lst: []
(Pdb) n
> example.py(4)<module>()
-> lst.append(1)
(Pdb) n
> example.py(5)<module>()
-> print(lst)
(Pdb)
```

You can do some tricks with copy mechanism to make it work:

```
> example.py(3)<module>()
-> pass
(Pdb) display lst[:]
```

```
display lst[:]: []
(Pdb) n
> example.py(4)<module>()
-> lst.append(1)
(Pdb) n
> example.py(5)<module>()
-> print(lst)
display lst[:]: [1]  [old: []]
(Pdb)
```

Added in version 3.2.

undisplay [expression]

Do not display *expression* anymore in the current frame. Without *expression*, clear all display expressions for the current frame.

Added in version 3.2.

interact

Start an interactive interpreter (using the `code` module) in a new global namespace initialised from the local and global namespaces for the current scope. Use `exit()` or `quit()` to exit the interpreter and return to the debugger.

Note

As `interact` creates a new dedicated namespace for code execution, assignments to variables will not affect the original namespaces. However, modifications to any referenced mutable objects will be reflected in the original namespaces as usual.

Added in version 3.2.

Changed in version 3.13: `exit()` and `quit()` can be used to exit the `interact` command.

Changed in version 3.13: **`interact`** directs its output to the debugger's output channel rather than **`sys.stderr`**.

alias [name [command]]

Create an alias called *name* that executes *command*. The *command* must *not* be enclosed in quotes. Replaceable parameters can be indicated by `%1`, `%2`, … and `%9`, while `%*` is replaced by all the parameters. If *command* is omitted, the current alias for *name* is shown. If no arguments are given, all aliases are listed.

Aliases may be nested and can contain anything that can be legally typed at the pdb prompt. Note that internal pdb commands *can* be overridden by aliases. Such a command is then hidden until the alias is removed. Aliasing is recursively applied to the first word of the command line; all other words in the line are left alone.

As an example, here are two useful aliases (especially when placed in the `.pdbrc` file):

```
# Print instance variables (usage "pi classInst")
alias pi for k in %1.__dict__.keys(): print(f"%1.{k} =
{%1.__dict__[k]}")
# Print instance variables in self
alias ps pi self
```

unalias name

Delete the specified alias *name*.

! statement

Execute the (one-line) *statement* in the context of the current stack frame. The exclamation point can be omitted unless the first word of the statement resembles a debugger command, e.g.:

```
(Pdb) ! n=42
(Pdb)
```

To set a global variable, you can prefix the assignment command with a `global` statement on the same line, e.g.:

```
(Pdb) global list_options; list_options = ['-l']
(Pdb)
```

run [args ...]
restart [args ...]

Restart the debugged Python program. If *args* is supplied, it is split with `shlex` and the result is used as the new `sys.argv`. History, breakpoints, actions and debugger options are preserved. `restart` is an alias for `run`.

Changed in version 3.14: `run` and `restart` commands are disabled when the debugger is invoked in `'inline'` mode.

q(uit)

Quit from the debugger. The program being executed is aborted. An end-of-file input is equivalent to `quit`.

A confirmation prompt will be shown if the debugger is invoked in `'inline'` mode. Either `y`, `Y`, `<Enter>` or `EOF` will confirm the quit.

Changed in version 3.14: A confirmation prompt will be shown if the debugger is invoked in `'inline'` mode. After the confirmation, the debugger will call `sys.exit()` immediately, instead of raising `bdb.BdbQuit` in the next trace event.

debug code

Enter a recursive debugger that steps through *code* (which is an arbitrary expression or statement to be executed in the current environment).

retval

Print the return value for the last return of the current function.

exceptions [excnumber]

List or jump between chained exceptions.

When using `pdb.pm()` or `Pdb.post_mortem(...)` with a chained exception instead of a traceback, it allows the user to move between the chained exceptions using `exceptions` command to list exceptions, and `exceptions <number>` to switch to that exception.

Example:

```
def out():
    try:
        middle()
    except Exception as e:
        raise ValueError("reraise middle() error") from e

def middle():
    try:
        return inner(0)
    except Exception as e:
        raise ValueError("Middle fail")

def inner(x):
    1 / x

 out()
```

calling `pdb.pm()` will allow to move between exceptions:

```
> example.py(5)out()
-> raise ValueError("reraise middle() error") from e

(Pdb) exceptions
  0 ZeroDivisionError('division by zero')
  1 ValueError('Middle fail')
```

```
> 2 ValueError('reraise middle() error')

(Pdb) exceptions 0
> example.py(16)inner()
-> 1 / x

(Pdb) up
> example.py(10)middle()
-> return inner(0)
```

Added in version 3.13.

Footnotes

[1]

Whether a frame is considered to originate in a certain module is determined by the `__name__` in the frame globals.

The Python Profilers

Source code: Lib/profile.py [https://github.com/python/cpython/tree/3.14/Lib/profile.py] and Lib/pstats.py [https://github.com/python/cpython/tree/3.14/Lib/pstats.py]

Introduction to the profilers

`cProfile` and **`profile`** provide *deterministic profiling* of Python programs. A *profile* is a set of statistics that describes how often and for how long various parts of the program executed. These statistics can be formatted into reports via the **`pstats`** module.

The Python standard library provides two different implementations of the same profiling interface:

1. **`cProfile`** is recommended for most users; it's a C extension with reasonable overhead that makes it suitable for profiling long-running programs. Based on **`lsprof`**, contributed by Brett Rosen and Ted Czotter.
2. **`profile`**, a pure Python module whose interface is imitated by **`cProfile`**, but which adds significant overhead to profiled programs. If you're trying to extend the profiler in some way, the task might be easier with this module. Originally designed and written by Jim Roskind.

Note

The profiler modules are designed to provide an execution profile for a given program, not for benchmarking purposes (for that, there is `timeit` for reasonably accurate results). This particularly applies to benchmarking Python code against C code: the profilers introduce overhead for Python code, but not for C-level functions, and so the C code would seem faster than any Python one.

Instant User’s Manual

This section is provided for users that “don’t want to read the manual.” It provides a very brief overview, and allows a user to rapidly perform profiling on an existing application.

To profile a function that takes a single argument, you can do:

```
import cProfile
import re
cProfile.run('re.compile("foo|bar")')
```

(Use `profile` instead of `cProfile` if the latter is not available on your system.)

The above action would run `re.compile()` and print profile results like the following:

```
      214 function calls (207 primitive calls) in 0.002 seconds

Ordered by: cumulative time

ncalls  tottime  percall  cumtime  percall
filename:lineno(function)
     1    0.000    0.000    0.002    0.002 {built-in method
builtins.exec}
     1    0.000    0.000    0.001    0.001 <string>:1(<module>)
     1    0.000    0.000    0.001    0.001
__init__.py:250(compile)
     1    0.000    0.000    0.001    0.001
```

```
__init__.py:289(_compile)
     1    0.000    0.000    0.000    0.000
_compiler.py:759(compile)
     1    0.000    0.000    0.000    0.000
_parser.py:937(parse)
     1    0.000    0.000    0.000    0.000
_compiler.py:598(_code)
     1    0.000    0.000    0.000    0.000
_parser.py:435(_parse_sub)
```

The first line indicates that 214 calls were monitored. Of those calls, 207 were *primitive*, meaning that the call was not induced via recursion. The next line: `Ordered by: cumulative time` indicates the output is sorted by the `cumtime` values. The column headings include:

ncalls
: for the number of calls.

tottime
: for the total time spent in the given function (and excluding time made in calls to sub-functions)

percall
: is the quotient of `tottime` divided by `ncalls`

cumtime
: is the cumulative time spent in this and all subfunctions (from invocation till exit). This figure is accurate *even* for recursive functions.

percall
: is the quotient of `cumtime` divided by primitive calls

filename:lineno(function)
: provides the respective data of each function

When there are two numbers in the first column (for example `3/1`), it means that the function recursed. The second value is the number of primitive calls and the former is the total number of calls. Note that when the function does not recurse, these two values are the same, and only the single figure is printed.

Instead of printing the output at the end of the profile run, you can save the results to a file by specifying a filename to the **`run()`** function:

```
import cProfile
import re
cProfile.run('re.compile("foo|bar")', 'restats')
```

The **`pstats.Stats`** class reads profile results from a file and formats them in various ways.

The files **`cProfile`** and **`profile`** can also be invoked as a script to profile another script. For example:

```
python -m cProfile [-o output_file] [-s sort_order] (-m module
| myscript.py)
```

-o <output_file>

Writes the profile results to a file instead of to stdout.

-s <sort_order>

Specifies one of the **`sort_stats()`** sort values to sort the output by. This only applies when **`-o`** is not supplied.

-m <module>

Specifies that a module is being profiled instead of a script.

Added in version 3.7: Added the `-m` option to **`cProfile`**.

Added in version 3.8: Added the `-m` option to **`profile`**.

The `pstats` module’s `Stats` class has a variety of methods for manipulating and printing the data saved into a profile results file:

```
import pstats
from pstats import SortKey
p = pstats.Stats('restats')
p.strip_dirs().sort_stats(-1).print_stats()
```

The `strip_dirs()` method removed the extraneous path from all the module names. The `sort_stats()` method sorted all the entries according to the standard module/line/name string that is printed. The `print_stats()` method printed out all the statistics. You might try the following sort calls:

```
p.sort_stats(SortKey.NAME)
p.print_stats()
```

The first call will actually sort the list by function name, and the second call will print out the statistics. The following are some interesting calls to experiment with:

```
p.sort_stats(SortKey.CUMULATIVE).print_stats(10)
```

This sorts the profile by cumulative time in a function, and then only prints the ten most significant lines. If you want to understand what algorithms are taking time, the above line is what you would use.

If you were looking to see what functions were looping a lot, and taking a lot of time, you would do:

```
p.sort_stats(SortKey.TIME).print_stats(10)
```

to sort according to time spent within each function, and then print the statistics for the top ten functions.

You might also try:

```
p.sort_stats(SortKey.FILENAME).print_stats('__init__')
```

This will sort all the statistics by file name, and then print out statistics for only the class init methods (since they are spelled with `__init__` in them). As one final example, you could try:

```
p.sort_stats(SortKey.TIME, SortKey.CUMULATIVE).print_stats(.5,
'init')
```

This line sorts statistics with a primary key of time, and a secondary key of cumulative time, and then prints out some of the statistics. To be specific, the list is first culled down to 50% (re: `.5`) of its original size, then only lines containing `init` are maintained, and that sub-sub-list is printed.

If you wondered what functions called the above functions, you could now (`p` is still sorted according to the last criteria) do:

```
p.print_callers(.5, 'init')
```

and you would get a list of callers for each of the listed functions.

If you want more functionality, you’re going to have to read the manual, or guess what the following functions do:

```
p.print_callees()
p.add('restats')
```

Invoked as a script, the **`pstats`** module is a statistics browser for reading and examining profile dumps. It has a simple line-oriented interface (implemented using **`cmd`**) and interactive help.

`profile` and `cProfile` Module Reference

Both the `profile` and `cProfile` modules provide the following functions:

profile.run(*command*, *filename=None*, *sort=-1*)

This function takes a single argument that can be passed to the `exec()` function, and an optional file name. In all cases this routine executes:

```
exec(command, __main__.__dict__, __main__.__dict__)
```

and gathers profiling statistics from the execution. If no file name is present, then this function automatically creates a `Stats` instance and prints a simple profiling report. If the sort value is specified, it is passed to this `Stats` instance to control how the results are sorted.

profile.runctx(*command*, *globals*, *locals*, *filename=None*, *sort=-1*)

This function is similar to `run()`, with added arguments to supply the globals and locals mappings for the *command* string. This routine executes:

```
exec(command, globals, locals)
```

and gathers profiling statistics as in the `run()` function above.

class profile.Profile(*timer=None*, *timeunit=0.0*, *subcalls=True*, *builtins=True*)

This class is normally only used if more precise control over profiling is needed than what the **`cProfile.run()`** function provides.

A custom timer can be supplied for measuring how long code takes to run via the *timer* argument. This must be a function that returns a single number representing the current time. If the number is an integer, the *timeunit* specifies a multiplier that specifies the duration of each unit of time. For example, if the timer returns times measured in thousands of seconds, the time unit would be `.001`.

Directly using the **`Profile`** class allows formatting profile results without writing the profile data to a file:

```
import cProfile, pstats, io
from pstats import SortKey
pr = cProfile.Profile()
pr.enable()
# ... do something ...
pr.disable()
s = io.StringIO()
sortby = SortKey.CUMULATIVE
ps = pstats.Stats(pr, stream=s).sort_stats(sortby)
ps.print_stats()
print(s.getvalue())
```

The **`Profile`** class can also be used as a context manager (supported only in **`cProfile`** module. see Context Manager Types):

```
import cProfile

with cProfile.Profile() as pr:
    # ... do something ...

    pr.print_stats()
```

Changed in version 3.8: Added context manager support.

enable()

Start collecting profiling data. Only in **`cProfile`**.

disable()

Stop collecting profiling data. Only in **`cProfile`**.

create_stats()

Stop collecting profiling data and record the results internally as the current profile.

print_stats(*sort=-1*)

Create a [`Stats`] object based on the current profile and print the results to stdout.

The *sort* parameter specifies the sorting order of the displayed statistics. It accepts a single key or a tuple of keys to enable multi-level sorting, as in `Stats.sort_stats`.

Added in version 3.13: `print_stats()` now accepts a tuple of keys.

dump_stats(*filename*)

Write the results of the current profile to *filename*.

run(*cmd*)

Profile the cmd via `exec()`.

runctx(*cmd*, *globals*, *locals*)

Profile the cmd via `exec()` with the specified global and local environment.

runcall(*func*, */*, **args*, ***kwargs*)

Profile `func(*args, **kwargs)`

Note that profiling will only work if the called command/function actually returns. If the interpreter is terminated (e.g. via a `sys.exit()` call during the called command/function execution) no profiling results will be printed.

The `Stats` Class

Analysis of the profiler data is done using the `Stats` class.

class pstats.Stats(**filenames or profile*, *stream=sys.stdout*)

This class constructor creates an instance of a "statistics object" from a *filename* (or list of filenames) or from a **`Profile`** instance. Output will be printed to the stream specified by *stream*.

The file selected by the above constructor must have been created by the corresponding version of `profile` or `cProfile`. To be specific, there is *no* file compatibility guaranteed with future versions of this profiler, and there is no compatibility with files produced by other profilers, or the same profiler run on a different operating system. If several files are provided, all the statistics for identical functions will be coalesced, so that an overall view of several processes can be considered in a single report. If additional files need to be combined with data in an existing `Stats` object, the `add()` method can be used.

Instead of reading the profile data from a file, a **`cProfile.Profile`** or `profile.Profile` object can be used as the profile data source.

`Stats` objects have the following methods:

strip_dirs()

This method for the `Stats` class removes all leading path information from file names. It is very useful in reducing the size of the printout to fit within (close to) 80 columns. This method modifies the object, and the stripped information is lost. After performing a strip operation, the object is considered to have its entries in a “random” order, as it was just after object initialization and loading. If **`strip_dirs()`** causes two function names to be indistinguishable (they are on the same line of the same filename, and have the same function name), then the statistics for these two entries are accumulated into a single entry.

add(**filenames*)

This method of the **`Stats`** class accumulates additional profiling information into the current profiling object. Its arguments should refer to filenames created by the corresponding version of **`profile.run()`** or **`cProfile.run()`**. Statistics for identically named (re: file, line, name) functions are automatically accumulated into single function statistics.

dump_stats(*filename*)

Save the data loaded into the **`Stats`** object to a file named *filename*. The file is created if it does not exist, and is overwritten if it already exists. This is equivalent to the method of the same name on the **`profile.Profile`** and **`cProfile.Profile`** classes.

sort_stats(**keys*)

This method modifies the **`Stats`** object by sorting it according to the supplied criteria. The argument can be either a string or a SortKey enum identifying the basis of a sort (example: `'time'`, `'name'`, `SortKey.TIME` or `SortKey.NAME`). The SortKey enums argument have advantage over the string argument in that it is more robust and less error prone.

When more than one key is provided, then additional keys are used as secondary criteria when there is equality in all keys selected before them. For example, `sort_stats(SortKey.NAME, SortKey.FILE)` will sort all the entries according to their function name, and resolve all ties (identical function names) by sorting by file name.

For the string argument, abbreviations can be used for any key names, as long as the abbreviation is unambiguous.

The following are the valid string and SortKey:

Valid String Arg	**Valid enum Arg**	**Meaning**
`'calls'`	SortKey.CALLS	call count
`'cumulative'`	SortKey.CUMULATIVE	cumulative time
`'cumtime'`	N/A	cumulative time
`'file'`	N/A	file name
`'filename'`	SortKey.FILENAME	file name
`'module'`	N/A	file name
`'ncalls'`	N/A	call count
`'pcalls'`	SortKey.PCALLS	primitive call count
`'line'`	SortKey.LINE	line number
`'name'`	SortKey.NAME	function name

Valid String Arg	Valid enum Arg	Meaning
`'nfl'`	SortKey.NFL	name/file/line
`'stdname'`	SortKey.STDNAME	standard name
`'time'`	SortKey.TIME	internal time
`'tottime'`	N/A	internal time

Note that all sorts on statistics are in descending order (placing most time consuming items first), where as name, file, and line number searches are in ascending order (alphabetical). The subtle distinction between `SortKey.NFL` and `SortKey.STDNAME` is that the standard name is a sort of the name as printed, which means that the embedded line numbers get compared in an odd way. For example, lines 3, 20, and 40 would (if the file names were the same) appear in the string order 20, 3 and 40. In contrast, `SortKey.NFL` does a numeric compare of the line numbers. In fact, `sort_stats(SortKey.NFL)` is the same as `sort_stats(SortKey.NAME, SortKey.FILENAME, SortKey.LINE)`.

For backward-compatibility reasons, the numeric arguments `-1`, `0`, `1`, and `2` are permitted. They are interpreted as `'stdname'`, `'calls'`, `'time'`, and `'cumulative'` respectively. If this old style format (numeric) is used, only one sort key (the numeric key) will be used, and additional arguments will be silently ignored.

Added in version 3.7: Added the SortKey enum.

reverse_order()

This method for the `Stats` class reverses the ordering of the basic list within the object. Note that by default ascending vs descending order is properly selected based on the sort key of choice.

print_stats(**restrictions*)

This method for the `Stats` class prints out a report as described in the `profile.run()` definition.

The order of the printing is based on the last `sort_stats()` operation done on the object (subject to caveats in `add()` and `strip_dirs()`).

The arguments provided (if any) can be used to limit the list down to the significant entries. Initially, the list is taken to be the complete set of profiled functions. Each restriction is either an integer (to select a count of lines), or a decimal fraction between 0.0 and 1.0 inclusive (to select a percentage of lines), or a string that will be interpreted as a regular expression (to pattern match the standard name that is printed). If several restrictions are provided, then they are applied sequentially. For example:

```
print_stats(.1, 'foo:')
```

would first limit the printing to first 10% of list, and then only print functions that were part of filename `.*foo:`. In contrast, the command:

```
print_stats('foo:', .1)
```

would limit the list to all functions having file names `.*foo:`, and then proceed to only print the first 10% of them.

print_callers(**restrictions*)

This method for the `Stats` class prints a list of all functions that called each function in the profiled database. The ordering is identical to that provided by `print_stats()`, and the definition of the restricting argument is also identical. Each caller is reported on its own line. The format differs slightly depending on the profiler that produced the stats:

- With `profile`, a number is shown in parentheses after each caller to show how many times this specific call was made. For convenience, a second non-parenthesized number repeats the cumulative time spent in the function at the right.
- With `cProfile`, each caller is preceded by three numbers: the number of times this specific call was made, and the total and cumulative times spent in the current function while it was invoked by this specific caller.

print_callees(**restrictions*)

This method for the `Stats` class prints a list of all function that were called by the indicated function. Aside from this reversal of direction of calls (re: called vs was called by), the arguments and ordering are identical to the `print_callers()` method.

get_stats_profile()

This method returns an instance of StatsProfile, which contains a mapping of function names to instances of FunctionProfile. Each FunctionProfile instance holds information related to the function’s profile such as how long the function took to run, how many times it was called, etc…

Added in version 3.9: Added the following dataclasses: StatsProfile, FunctionProfile. Added the following function: get_stats_profile.

What Is Deterministic Profiling?

Deterministic profiling is meant to reflect the fact that all *function call*, *function return*, and *exception* events are monitored, and precise timings are made for the intervals between these events (during which time the user’s code is executing). In contrast, *statistical profiling* (which is not done by this module) randomly samples the effective instruction pointer, and deduces where time is being spent. The latter technique traditionally involves less overhead (as the code does not need to be instrumented), but provides only relative indications of where time is being spent.

In Python, since there is an interpreter active during execution, the presence of instrumented code is not required in order to do deterministic profiling. Python automatically provides a *hook* (optional callback) for each event. In addition, the interpreted nature of Python tends to add so much overhead to execution, that deterministic profiling tends to only add small processing overhead in typical applications. The result is that deterministic profiling is not that expensive, yet provides extensive run time statistics about the execution of a Python program.

Call count statistics can be used to identify bugs in code (surprising counts), and to identify possible inline-expansion points (high call counts). Internal time statistics can be used to identify “hot loops” that should be carefully optimized. Cumulative time statistics should be used to identify high level errors in the selection of algorithms. Note that the unusual handling of cumulative times in this profiler allows statistics for recursive implementations of algorithms to be directly compared to iterative implementations.

Limitations

One limitation has to do with accuracy of timing information. There is a fundamental problem with deterministic profilers involving accuracy. The most obvious restriction is that the underlying “clock” is only ticking at a rate (typically) of about .001 seconds. Hence no measurements will be more accurate than the underlying clock. If enough measurements are taken, then the “error” will tend to average out. Unfortunately, removing this first error induces a second source of error.

The second problem is that it “takes a while” from when an event is dispatched until the profiler’s call to get the time actually *gets* the state of the clock. Similarly, there is a certain lag when exiting the profiler event handler from the time that the clock’s value was obtained (and then squirreled away), until the user’s code is once again executing. As a result, functions that are called many times, or call many functions, will typically accumulate this error. The error that accumulates in this fashion is typically less than the accuracy of the clock (less than one clock tick), but it *can* accumulate and become very significant.

The problem is more important with `profile` than with the lower-overhead `cProfile`. For this reason, `profile` provides a means of calibrating itself for a given platform so that this error can be probabilistically (on the average) removed. After the profiler is calibrated, it will be more accurate (in a least square sense), but it will sometimes produce negative numbers (when call counts are exceptionally low, and the gods of probability work against you :-).) Do *not* be alarmed by negative numbers in the profile. They should *only* appear if you have calibrated your profiler, and the results are actually better than without calibration.

Calibration

The profiler of the `profile` module subtracts a constant from each event handling time to compensate for the overhead of calling the time function, and socking away the results. By default, the constant is 0. The following procedure can be used to obtain a better constant for a given platform (see Limitations).

```
import profile
pr = profile.Profile()
for i in range(5):
    print(pr.calibrate(10000))
```

The method executes the number of Python calls given by the argument, directly and again under the profiler, measuring the time for both. It then computes the hidden overhead per profiler event, and returns that as a float. For example, on a 1.8Ghz Intel Core i5 running macOS, and using Python’s time.process_time() as the timer, the magical number is about 4.04e-6.

The object of this exercise is to get a fairly consistent result. If your computer is *very* fast, or your timer function has poor resolution, you might have to pass 100000, or even 1000000, to get consistent results.

When you have a consistent answer, there are three ways you can use it:

```
import profile

# 1. Apply computed bias to all Profile instances created
hereafter.
profile.Profile.bias = your_computed_bias

# 2. Apply computed bias to a specific Profile instance.
pr = profile.Profile()
```

```
pr.bias = your_computed_bias

# 3. Specify computed bias in instance constructor.
pr = profile.Profile(bias=your_computed_bias)
```

If you have a choice, you are better off choosing a smaller constant, and then your results will “less often” show up as negative in profile statistics.

Using a custom timer

If you want to change how current time is determined (for example, to force use of wall-clock time or elapsed process time), pass the timing function you want to the **`Profile`** class constructor:

```
pr = profile.Profile(your_time_func)
```

The resulting profiler will then call `your_time_func`. Depending on whether you are using **`profile.Profile`** or **`cProfile.Profile`**, `your_time_func`’s return value will be interpreted differently:

`profile.Profile`

`your_time_func` should return a single number, or a list of numbers whose sum is the current time (like what **`os.times()`** returns). If the function returns a single time number, or the list of returned numbers has length 2, then you will get an especially fast version of the dispatch routine.

Be warned that you should calibrate the profiler class for the timer function that you choose (see Calibration). For most machines, a timer that returns a lone integer value will provide the best results in terms of low overhead during profiling. (**`os.times()`** is *pretty* bad, as it returns a tuple of floating-point values). If you want to substitute a better timer in the cleanest fashion, derive a class and hardwire a replacement dispatch method that best handles your timer call, along with the appropriate calibration constant.

`cProfile.Profile`

`your_time_func` should return a single number. If it returns integers, you can also invoke the class constructor with a second argument specifying the real duration of one unit of time. For example, if `your_integer_time_func` returns times measured in thousands of seconds, you would construct the **`Profile`** instance as follows:

```
pr = cProfile.Profile(your_integer_time_func, 0.001)
```

As the **`cProfile.Profile`** class cannot be calibrated, custom timer functions should be used with care and should be as fast as possible. For the best results with a custom timer, it might be necessary to hard-code it in the C source of the internal **`_lsprof`** module.

Python 3.3 adds several new functions in **`time`** that can be used to make precise measurements of process or wall-clock time. For example, see **`time.perf_counter()`**.

`timeit` — Measure execution time of small code snippets

Source code: Lib/timeit.py
[https://github.com/python/cpython/tree/3.14/Lib/timeit.py]

This module provides a simple way to time small bits of Python code. It has both a Command-Line Interface as well as a callable one. It avoids a number of common traps for measuring execution times. See also Tim Peters' introduction to the "Algorithms" chapter in the second edition of *Python Cookbook*, published by O'Reilly.

Basic Examples

The following example shows how the Command-Line Interface can be used to compare three different expressions:

```
$ python -m timeit "'-'.join(str(n) for n in range(100))"
10000 loops, best of 5: 30.2 usec per loop
$ python -m timeit "'-'.join([str(n) for n in range(100)])"
10000 loops, best of 5: 27.5 usec per loop
$ python -m timeit "'-'.join(map(str, range(100)))"
10000 loops, best of 5: 23.2 usec per loop
```

This can be achieved from the Python Interface with:

```
>>> import timeit
>>> timeit.timeit('"-".join(str(n) for n in range(100))',
number=10000)
```

```
0.3018611848820001
>>> timeit.timeit('"-".join([str(n) for n in range(100)])',
number=10000)
0.2727368790656328
>>> timeit.timeit('"-".join(map(str, range(100)))',
number=10000)
0.23702679807320237
```

A callable can also be passed from the Python Interface:

```
>>> timeit.timeit(lambda: "-".join(map(str, range(100))),
number=10000)
0.19665591977536678
```

Note however that **`timeit()`** will automatically determine the number of repetitions only when the command-line interface is used. In the Examples section you can find more advanced examples.

Python Interface

The module defines three convenience functions and a public class:

timeit.timeit(*stmt='pass'*, *setup='pass'*, *timer=<default timer>*, *number=1000000*, *globals=None*)

> Create a **`Timer`** instance with the given statement, *setup* code and *timer* function and run its **`timeit()`** method with *number* executions. The optional *globals* argument specifies a namespace in which to execute the code.
>
> *Changed in version 3.5:* The optional *globals* parameter was added.

timeit.repeat(*stmt='pass'*, *setup='pass'*, *timer=<default timer>*, *repeat=5*, *number=1000000*, *globals=None*)

Create a `Timer` instance with the given statement, *setup* code and *timer* function and run its `repeat()` method with the given *repeat* count and *number* executions. The optional *globals* argument specifies a namespace in which to execute the code.

Changed in version 3.5: The optional *globals* parameter was added.

Changed in version 3.7: Default value of *repeat* changed from 3 to 5.

timeit.default_timer()

The default timer, which is always time.perf_counter(), returns float seconds. An alternative, time.perf_counter_ns, returns integer nanoseconds.

Changed in version 3.3: `time.perf_counter()` is now the default timer.

class timeit.Timer(*stmt='pass'*, *setup='pass'*, *timer=<timer function>*, *globals=None*)

Class for timing execution speed of small code snippets.

The constructor takes a statement to be timed, an additional statement used for setup, and a timer function. Both statements default to `'pass'`; the timer function is platform-dependent (see the module doc string). *stmt* and *setup* may also contain multiple statements separated by `;` or newlines, as long as they don’t contain multi-line string literals. The statement will by default be executed within timeit’s namespace; this behavior can be controlled by passing a namespace to *globals*.

To measure the execution time of the first statement, use the `timeit()` method. The `repeat()` and `autorange()` methods are convenience methods to call `timeit()` multiple times.

The execution time of *setup* is excluded from the overall timed execution run.

The *stmt* and *setup* parameters can also take objects that are callable without arguments. This will embed calls to them in a timer function that will then be executed by `timeit()`. Note that the timing overhead is a little larger in this case because of the extra function calls.

Changed in version 3.5: The optional *globals* parameter was added.

timeit(*number=1000000*)

Time *number* executions of the main statement. This executes the setup statement once, and then returns the time it takes to execute the main statement a number of times. The default timer returns seconds as a float. The argument is the number of times through the loop, defaulting to one million. The main statement, the setup statement and the timer function to be used are passed to the constructor.

Note

By default, `timeit()` temporarily turns off garbage collection during the timing. The advantage of this approach is that it makes independent timings more comparable. The disadvantage is that GC may be an important component of the performance of the function being measured. If so, GC can be re-enabled as the first statement in the *setup* string. For example:

```
timeit.Timer('for i in range(10): oct(i)',
'gc.enable()').timeit()
```

autorange(*callback=None*)

Automatically determine how many times to call `timeit()`.

This is a convenience function that calls `timeit()` repeatedly so that the total time >= 0.2 second, returning the eventual (number of loops, time taken for that number of loops). It calls `timeit()` with increasing numbers from the sequence 1, 2, 5, 10, 20, 50, … until the time taken is at least 0.2 seconds.

If *callback* is given and is not `None`, it will be called after each trial with two arguments: `callback(number, time_taken)`.

Added in version 3.6.

repeat(*repeat=5*, *number=1000000*)

Call `timeit()` a few times.

This is a convenience function that calls the `timeit()` repeatedly, returning a list of results. The first argument specifies how many times to call `timeit()`. The second argument specifies the *number* argument for `timeit()`.

Note

It’s tempting to calculate mean and standard deviation from the result vector and report these. However, this is not very useful. In a typical case, the lowest value gives a lower bound for how fast your machine can run the given code snippet; higher values in the result vector are typically not caused by variability in Python’s speed, but by other processes interfering with your timing accuracy. So the `min()` of the result is probably the only number you should be interested in. After that, you should look at the entire vector and apply common sense rather than statistics.

Changed in version 3.7: Default value of *repeat* changed from 3 to 5.

print_exc(*file=None*)

Helper to print a traceback from the timed code.

Typical use:

```
t = Timer(...)       # outside the try/except
try:
    t.timeit(...)    # or t.repeat(...)
except Exception:
    t.print_exc()
```

The advantage over the standard traceback is that source lines in the compiled template will be displayed. The optional *file* argument directs where the traceback is sent; it defaults to `sys.stderr`.

Command-Line Interface

When called as a program from the command line, the following form is used:

```
python -m timeit [-n N] [-r N] [-u U] [-s S] [-p] [-v] [-h]
[statement ...]
```

Where the following options are understood:

-n N, --number=N
: how many times to execute ‘statement’

-r N, --repeat=N
: how many times to repeat the timer (default 5)

-s S, --setup=S
: statement to be executed once initially (default `pass`)

-p, --process
: measure process time, not wallclock time, using `time.process_time()` instead of `time.perf_counter()`, which is the default

  Added in version 3.3.

-u, --unit=U
: specify a time unit for timer output; can select `nsec`, `usec`, `msec`, or `sec`

  Added in version 3.5.

-v, --verbose
: print raw timing results; repeat for more digits precision

-h, --help
: print a short usage message and exit

A multi-line statement may be given by specifying each line as a separate statement argument; indented lines are possible by enclosing an argument in quotes and using leading spaces. Multiple `-s` options are treated similarly.

If `-n` is not given, a suitable number of loops is calculated by trying increasing numbers from the sequence 1, 2, 5, 10, 20, 50, … until the total time is at least 0.2 seconds.

`default_timer()` measurements can be affected by other programs running on the same machine, so the best thing to do when accurate timing is necessary is to repeat the timing a few times and use the best time. The `-r` option is good for this; the default of 5 repetitions is probably enough in most cases. You can use `time.process_time()` to measure CPU time.

Note

There is a certain baseline overhead associated with executing a pass statement. The code here doesn't try to hide it, but you should be aware of it. The baseline overhead can be measured by invoking the program without arguments, and it might differ between Python versions.

Examples

It is possible to provide a setup statement that is executed only once at the beginning:

```
$ python -m timeit -s "text = 'sample string'; char = 'g'"
"char in text"
5000000 loops, best of 5: 0.0877 usec per loop
$ python -m timeit -s "text = 'sample string'; char = 'g'"
"text.find(char)"
1000000 loops, best of 5: 0.342 usec per loop
```

In the output, there are three fields. The loop count, which tells you how many times the statement body was run per timing loop repetition. The repetition count ('best of 5') which tells you how many times the timing loop was repeated, and finally the time the statement body took on average within the best repetition of the timing loop. That is, the time the fastest repetition took divided by the loop count.

```
>>> import timeit
>>> timeit.timeit('char in text', setup='text = "sample
string"; char = "g"')
0.41440500499993504
>>> timeit.timeit('text.find(char)', setup='text = "sample
string"; char = "g"')
1.7246671520006203
```

The same can be done using the `Timer` class and its methods:

```
>>> import timeit
>>> t = timeit.Timer('char in text', setup='text = "sample
string"; char = "g"')
>>> t.timeit()
0.3955516149999312
>>> t.repeat()
[0.40183617287970225, 0.37027556854118704, 0.38344867356679524,
0.3712595970846668, 0.37866875250654886]
```

The following examples show how to time expressions that contain multiple lines. Here we compare the cost of using `hasattr()` vs. `try`/`except` to test for missing and present object attributes:

```
$ python -m timeit "try:" "  str.__bool__" "except
AttributeError:" "  pass"
20000 loops, best of 5: 15.7 usec per loop
$ python -m timeit "if hasattr(str, '__bool__'): pass"
50000 loops, best of 5: 4.26 usec per loop

$ python -m timeit "try:" "  int.__bool__" "except
AttributeError:" "  pass"
200000 loops, best of 5: 1.43 usec per loop
$ python -m timeit "if hasattr(int, '__bool__'): pass"
100000 loops, best of 5: 2.23 usec per loop
```

```
>>> import timeit
>>> # attribute is missing
>>> s = """\
... try:
...     str.__bool__
... except AttributeError:
...     pass
... """
>>> timeit.timeit(stmt=s, number=100000)
0.9138244460009446
>>> s = "if hasattr(str, '__bool__'): pass"
>>> timeit.timeit(stmt=s, number=100000)
0.5829014980008651
>>>
>>> # attribute is present
>>> s = """\
... try:
...     int.__bool__
... except AttributeError:
...     pass
... """
>>> timeit.timeit(stmt=s, number=100000)
0.04215312199994514
>>> s = "if hasattr(int, '__bool__'): pass"
>>> timeit.timeit(stmt=s, number=100000)
0.08588060699912603
```

To give the `timeit` module access to functions you define, you can pass a *setup* parameter which contains an import statement:

```
def test():
    """Stupid test function"""
    L = [i for i in range(100)]

if __name__ == '__main__':
    import timeit
    print(timeit.timeit("test()", setup="from __main__ import
test"))
```

Another option is to pass `globals()` to the *globals* parameter, which will cause the code to be executed within your current global namespace. This can be more convenient than individually specifying imports:

```
def f(x):
    return x**2
def g(x):
    return x**4
def h(x):
    return x**8

import timeit
print(timeit.timeit('[func(42) for func in (f,g,h)]',
globals=globals()))
```

`trace` — Trace or track Python statement execution

Source code: Lib/trace.py [https://github.com/python/cpython/tree/3.14/Lib/trace.py]

The `trace` module allows you to trace program execution, generate annotated statement coverage listings, print caller/callee relationships and list functions executed during a program run. It can be used in another program or from the command line.

See also

Coverage.py [https://coverage.readthedocs.io/]
A popular third-party coverage tool that provides HTML output along with advanced features such as branch coverage.

Command-Line Usage

The `trace` module can be invoked from the command line. It can be as simple as

```
python -m trace --count -C . somefile.py ...
```

The above will execute `somefile.py` and generate annotated listings of all Python modules imported during the execution into the current directory.

--help
: Display usage and exit.

--version
: Display the version of the module and exit.

Added in version 3.8: Added `--module` option that allows to run an executable module.

Main options

At least one of the following options must be specified when invoking **`trace`**. The **`--listfuncs`** option is mutually exclusive with the **`--trace`** and **`--count`** options. When **`--listfuncs`** is provided, neither **`--count`** nor **`--trace`** are accepted, and vice versa.

-c, --count
: Produce a set of annotated listing files upon program completion that shows how many times each statement was executed. See also **`--coverdir`**, **`--file`** and **`--no-report`** below.

-t, --trace
: Display lines as they are executed.

-l, --listfuncs
: Display the functions executed by running the program.

-r, --report
: Produce an annotated list from an earlier program run that used the **`--count`** and **`--file`** option. This does not execute any code.

-T, --trackcalls

Display the calling relationships exposed by running the program.

Modifiers

-f, --file=<file>

Name of a file to accumulate counts over several tracing runs. Should be used with the `--count` option.

-C, --coverdir=<dir>

Directory where the report files go. The coverage report for `package.module` is written to file `dir/package/module.cover`.

-m, --missing

When generating annotated listings, mark lines which were not executed with `>>>>>>`.

-s, --summary

When using `--count` or `--report`, write a brief summary to stdout for each file processed.

-R, --no-report

Do not generate annotated listings. This is useful if you intend to make several runs with `--count`, and then produce a single set of annotated listings at the end.

-g, --timing

Prefix each line with the time since the program started. Only used while tracing.

Filters

These options may be repeated multiple times.

--ignore-module=<mod>

Ignore each of the given module names and its submodules (if it is a package). The argument can be a list of names separated by a comma.

--ignore-dir=<dir>

Ignore all modules and packages in the named directory and subdirectories. The argument can be a list of directories separated by `os.pathsep`.

Programmatic Interface

class trace.Trace(*count=1*, *trace=1*, *countfuncs=0*, *countcallers=0*, *ignoremods=()*, *ignoredirs=()*, *infile=None*, *outfile=None*, *timing=False*)

Create an object to trace execution of a single statement or expression. All parameters are optional. *count* enables counting of line numbers. *trace* enables line execution tracing. *countfuncs* enables listing of the functions called during the run. *countcallers* enables call relationship tracking. *ignoremods* is a list of modules or packages to ignore. *ignoredirs* is a list of directories whose modules or packages should be ignored. *infile* is the name of the file from which to read stored count information. *outfile* is the name of the file in which to write updated count information. *timing* enables a timestamp relative to when tracing was started to be displayed.

run(*cmd*)

Execute the command and gather statistics from the execution with the current tracing parameters. *cmd* must be a string or code object, suitable for passing into `exec()`.

runctx(*cmd*, *globals=None*, *locals=None*)

Execute the command and gather statistics from the execution with the current tracing parameters, in the defined global and local environments. If not defined, *globals* and *locals* default to empty dictionaries.

runfunc(*func*, */*, **args*, ***kwds*)

Call *func* with the given arguments under control of the `Trace` object with the current tracing parameters.

results()

Return a `CoverageResults` object that contains the cumulative results of all previous calls to `run`, `runctx` and `runfunc` for the given `Trace` instance. Does not reset the accumulated trace results.

class trace.CoverageResults

A container for coverage results, created by `Trace.results()`. Should not be created directly by the user.

update(*other*)

Merge in data from another `CoverageResults` object.

write_results(*show_missing=True*, *summary=False*, *coverdir=None*, ***, *ignore_missing_files=False*)

Write coverage results. Set *show_missing* to show lines that had no hits. Set *summary* to include in the output the coverage summary per module. *coverdir* specifies the directory into which the coverage result files will be output. If `None`, the results for each source file are placed in its directory.

If *ignore_missing_files* is `True`, coverage counts for files that no longer exist are silently ignored. Otherwise, a missing file will raise a `FileNotFoundError`.

Changed in version 3.13: Added *ignore_missing_files* parameter.

A simple example demonstrating the use of the programmatic interface:

```
import sys
import trace

# create a Trace object, telling it what to ignore, and whether
to
# do tracing or line-counting or both.
tracer = trace.Trace(
    ignoredirs=[sys.prefix, sys.exec_prefix],
    trace=0,
    count=1)

# run the new command using the given tracer
tracer.run('main()')

# make a report, placing output in the current directory
r = tracer.results()
r.write_results(show_missing=True, coverdir=".")
```

`tracemalloc` — Trace memory allocations

Added in version 3.4.

Source code: Lib/tracemalloc.py
[https://github.com/python/cpython/tree/3.14/Lib/tracemalloc.py]

The tracemalloc module is a debug tool to trace memory blocks allocated by Python. It provides the following information:

- Traceback where an object was allocated
- Statistics on allocated memory blocks per filename and per line number: total size, number and average size of allocated memory blocks
- Compute the differences between two snapshots to detect memory leaks

To trace most memory blocks allocated by Python, the module should be started as early as possible by setting the **`PYTHONTRACEMALLOC`** environment variable to `1`, or by using **`-X`** `tracemalloc` command line option. The **`tracemalloc.start()`** function can be called at runtime to start tracing Python memory allocations.

By default, a trace of an allocated memory block only stores the most recent frame (1 frame). To store 25 frames at startup: set the **`PYTHONTRACEMALLOC`** environment variable to `25`, or use the **`-X`** `tracemalloc=25` command line option.

Examples

Display the top 10

Display the 10 files allocating the most memory:

```
import tracemalloc

tracemalloc.start()

# ... run your application ...

snapshot = tracemalloc.take_snapshot()
top_stats = snapshot.statistics('lineno')

print("[ Top 10 ]")
for stat in top_stats[:10]:
    print(stat)
```

Example of output of the Python test suite:

```
[ Top 10 ]
<frozen importlib._bootstrap>:716: size=4855 KiB, count=39328,
average=126 B
<frozen importlib._bootstrap>:284: size=521 KiB, count=3199,
average=167 B
/usr/lib/python3.4/collections/__init__.py:368: size=244 KiB,
count=2315, average=108 B
/usr/lib/python3.4/unittest/case.py:381: size=185 KiB,
count=779, average=243 B
/usr/lib/python3.4/unittest/case.py:402: size=154 KiB,
count=378, average=416 B
/usr/lib/python3.4/abc.py:133: size=88.7 KiB, count=347,
average=262 B
<frozen importlib._bootstrap>:1446: size=70.4 KiB, count=911,
average=79 B
<frozen importlib._bootstrap>:1454: size=52.0 KiB, count=25,
average=2131 B
<string>:5: size=49.7 KiB, count=148, average=344 B
/usr/lib/python3.4/sysconfig.py:411: size=48.0 KiB, count=1,
average=48.0 KiB
```

We can see that Python loaded `4855 KiB` data (bytecode and constants) from modules and that the `collections` module allocated `244 KiB` to build `namedtuple` types.

See `Snapshot.statistics()` for more options.

Compute differences

Take two snapshots and display the differences:

```
import tracemalloc
tracemalloc.start()
# ... start your application ...

snapshot1 = tracemalloc.take_snapshot()
# ... call the function leaking memory ...
snapshot2 = tracemalloc.take_snapshot()

top_stats = snapshot2.compare_to(snapshot1, 'lineno')

print("[ Top 10 differences ]")
for stat in top_stats[:10]:
    print(stat)
```

Example of output before/after running some tests of the Python test suite:

```
[ Top 10 differences ]
<frozen importlib._bootstrap>:716: size=8173 KiB (+4428 KiB),
count=71332 (+39369), average=117 B
/usr/lib/python3.4/linecache.py:127: size=940 KiB (+940 KiB),
count=8106 (+8106), average=119 B
/usr/lib/python3.4/unittest/case.py:571: size=298 KiB (+298
KiB), count=589 (+589), average=519 B
<frozen importlib._bootstrap>:284: size=1005 KiB (+166 KiB),
count=7423 (+1526), average=139 B
/usr/lib/python3.4/mimetypes.py:217: size=112 KiB (+112 KiB),
count=1334 (+1334), average=86 B
/usr/lib/python3.4/http/server.py:848: size=96.0 KiB (+96.0
KiB), count=1 (+1), average=96.0 KiB
/usr/lib/python3.4/inspect.py:1465: size=83.5 KiB (+83.5 KiB),
count=109 (+109), average=784 B
/usr/lib/python3.4/unittest/mock.py:491: size=77.7 KiB (+77.7
KiB), count=143 (+143), average=557 B
/usr/lib/python3.4/urllib/parse.py:476: size=71.8 KiB (+71.8
KiB), count=969 (+969), average=76 B
/usr/lib/python3.4/contextlib.py:38: size=67.2 KiB (+67.2 KiB),
count=126 (+126), average=546 B
```

We can see that Python has loaded `8173 KiB` of module data (bytecode and constants), and that this is `4428 KiB` more than had been loaded before the tests, when the previous snapshot was taken. Similarly, the **`linecache`** module has cached `940 KiB` of Python source code to format tracebacks, all of it since the previous snapshot.

If the system has little free memory, snapshots can be written on disk using the **`Snapshot.dump()`** method to analyze the snapshot offline. Then use the **`Snapshot.load()`** method reload the snapshot.

Get the traceback of a memory block

Code to display the traceback of the biggest memory block:

```
import tracemalloc

# Store 25 frames
tracemalloc.start(25)

# ... run your application ...

snapshot = tracemalloc.take_snapshot()
top_stats = snapshot.statistics('traceback')

# pick the biggest memory block
stat = top_stats[0]
print("%s memory blocks: %.1f KiB" % (stat.count, stat.size /
1024))
for line in stat.traceback.format():
    print(line)
```

Example of output of the Python test suite (traceback limited to 25 frames):

```
903 memory blocks: 870.1 KiB
  File "<frozen importlib._bootstrap>", line 716
  File "<frozen importlib._bootstrap>", line 1036
  File "<frozen importlib._bootstrap>", line 934
  File "<frozen importlib._bootstrap>", line 1068
  File "<frozen importlib._bootstrap>", line 619
  File "<frozen importlib._bootstrap>", line 1581
  File "<frozen importlib._bootstrap>", line 1614
  File "/usr/lib/python3.4/doctest.py", line 101
    import pdb
  File "<frozen importlib._bootstrap>", line 284
  File "<frozen importlib._bootstrap>", line 938
  File "<frozen importlib._bootstrap>", line 1068
  File "<frozen importlib._bootstrap>", line 619
  File "<frozen importlib._bootstrap>", line 1581
  File "<frozen importlib._bootstrap>", line 1614
  File "/usr/lib/python3.4/test/support/__init__.py", line 1728
    import doctest
  File "/usr/lib/python3.4/test/test_pickletools.py", line 21
    support.run_doctest(pickletools)
  File "/usr/lib/python3.4/test/regrtest.py", line 1276
    test_runner()
  File "/usr/lib/python3.4/test/regrtest.py", line 976
    display_failure=not verbose)
  File "/usr/lib/python3.4/test/regrtest.py", line 761
    match_tests=ns.match_tests)
  File "/usr/lib/python3.4/test/regrtest.py", line 1563
    main()
  File "/usr/lib/python3.4/test/__main__.py", line 3
    regrtest.main_in_temp_cwd()
  File "/usr/lib/python3.4/runpy.py", line 73
    exec(code, run_globals)
  File "/usr/lib/python3.4/runpy.py", line 160
    "__main__", fname, loader, pkg_name)
```

We can see that the most memory was allocated in the **`importlib`** module to load data (bytecode and constants) from modules: `870.1 KiB`. The traceback is where the **`importlib`** loaded data most recently: on the `import pdb` line of the `doctest` module. The traceback may change if a new module is loaded.

Pretty top

Code to display the 10 lines allocating the most memory with a pretty output, ignoring `<frozen importlib._bootstrap>` and `<unknown>` files:

```
import linecache
import os
import tracemalloc

def display_top(snapshot, key_type='lineno', limit=10):
    snapshot = snapshot.filter_traces((
        tracemalloc.Filter(False, "<frozen
importlib._bootstrap>"),
        tracemalloc.Filter(False, "<unknown>"),
    ))
    top_stats = snapshot.statistics(key_type)

    print("Top %s lines" % limit)
    for index, stat in enumerate(top_stats[:limit], 1):
        frame = stat.traceback[0]
        print("#%s: %s:%s: %.1f KiB"
              % (index, frame.filename, frame.lineno, stat.size
/ 1024))
        line = linecache.getline(frame.filename,
frame.lineno).strip()
        if line:
            print('    %s' % line)

    other = top_stats[limit:]
    if other:
        size = sum(stat.size for stat in other)
        print("%s other: %.1f KiB" % (len(other), size / 1024))
    total = sum(stat.size for stat in top_stats)
    print("Total allocated size: %.1f KiB" % (total / 1024))

tracemalloc.start()
```

```
# ... run your application ...

snapshot = tracemalloc.take_snapshot()
display_top(snapshot)
```

Example of output of the Python test suite:

```
Top 10 lines
#1: Lib/base64.py:414: 419.8 KiB
    _b85chars2 = [(a + b) for a in _b85chars for b in
_b85chars]
#2: Lib/base64.py:306: 419.8 KiB
    _a85chars2 = [(a + b) for a in _a85chars for b in
_a85chars]
#3: collections/__init__.py:368: 293.6 KiB
    exec(class_definition, namespace)
#4: Lib/abc.py:133: 115.2 KiB
    cls = super().__new__(mcls, name, bases, namespace)
#5: unittest/case.py:574: 103.1 KiB
    testMethod()
#6: Lib/linecache.py:127: 95.4 KiB
    lines = fp.readlines()
#7: urllib/parse.py:476: 71.8 KiB
    for a in _hexdig for b in _hexdig}
#8: <string>:5: 62.0 KiB
#9: Lib/_weakrefset.py:37: 60.0 KiB
    self.data = set()
#10: Lib/base64.py:142: 59.8 KiB
    _b32tab2 = [a + b for a in _b32tab for b in _b32tab]
6220 other: 3602.8 KiB
Total allocated size: 5303.1 KiB
```

See `Snapshot.statistics()` for more options.

Record the current and peak size of all traced memory blocks

The following code computes two sums like `0 + 1 + 2 + ...` inefficiently, by creating a list of those numbers. This list consumes a lot of memory temporarily. We can use [`get_traced_memory()`] and [`reset_peak()`] to observe the small memory usage after the sum is computed as well as the peak memory usage during the computations:

```
import tracemalloc

tracemalloc.start()

# Example code: compute a sum with a large temporary list
large_sum = sum(list(range(100000)))

first_size, first_peak = tracemalloc.get_traced_memory()

tracemalloc.reset_peak()

# Example code: compute a sum with a small temporary list
small_sum = sum(list(range(1000)))

second_size, second_peak = tracemalloc.get_traced_memory()

print(f"{first_size=}, {first_peak=}")
print(f"{second_size=}, {second_peak=}")
```

Output:

```
first_size=664, first_peak=3592984
second_size=804, second_peak=29704
```

Using `reset_peak()` ensured we could accurately record the peak during the computation of `small_sum`, even though it is much smaller than the overall peak size of memory blocks since the `start()` call. Without the call to `reset_peak()`, `second_peak` would still be the peak from the computation `large_sum` (that is, equal to `first_peak`). In this case, both peaks are much higher than the final memory usage, and which suggests we could optimise (by removing the unnecessary call to `list`, and writing `sum(range(...))`).

API

Functions

tracemalloc.clear_traces()

Clear traces of memory blocks allocated by Python.

See also `stop()`.

tracemalloc.get_object_traceback(*obj*)

Get the traceback where the Python object *obj* was allocated. Return a `Traceback` instance, or `None` if the `tracemalloc` module is not tracing memory allocations or did not trace the allocation of the object.

See also `gc.get_referrers()` and `sys.getsizeof()` functions.

tracemalloc.get_traceback_limit()

Get the maximum number of frames stored in the traceback of a trace.

The `tracemalloc` module must be tracing memory allocations to get the limit, otherwise an exception is raised.

The limit is set by the `start()` function.

tracemalloc.get_traced_memory()

Get the current size and peak size of memory blocks traced by the `tracemalloc` module as a tuple: `(current: int, peak: int)`.

tracemalloc.reset_peak()

Set the peak size of memory blocks traced by the `tracemalloc` module to the current size.

Do nothing if the `tracemalloc` module is not tracing memory allocations.

This function only modifies the recorded peak size, and does not modify or clear any traces, unlike `clear_traces()`. Snapshots taken with `take_snapshot()` before a call to `reset_peak()` can be meaningfully compared to snapshots taken after the call.

See also `get_traced_memory()`.

Added in version 3.9.

tracemalloc.get_tracemalloc_memory()

Get the memory usage in bytes of the `tracemalloc` module used to store traces of memory blocks. Return an `int`.

tracemalloc.is_tracing()

`True` if the `tracemalloc` module is tracing Python memory allocations, `False` otherwise.

See also `start()` and `stop()` functions.

tracemalloc.start(*nframe*: *int* = *1*)

Start tracing Python memory allocations: install hooks on Python memory allocators. Collected tracebacks of traces will be limited to *nframe* frames. By default, a trace of a memory block only stores the most recent frame: the limit is `1`. *nframe* must be greater or equal to `1`.

You can still read the original number of total frames that composed the traceback by looking at the `Traceback.total_nframe` attribute.

Storing more than `1` frame is only useful to compute statistics grouped by `'traceback'` or to compute cumulative statistics: see the `Snapshot.compare_to()` and `Snapshot.statistics()` methods.

Storing more frames increases the memory and CPU overhead of the `tracemalloc` module. Use the `get_tracemalloc_memory()` function to measure how much memory is used by the `tracemalloc` module.

The `PYTHONTRACEMALLOC` environment variable (`PYTHONTRACEMALLOC=NFRAME`) and the `-X` `tracemalloc=NFRAME` command line option can be used to start tracing at startup.

See also `stop()`, `is_tracing()` and `get_traceback_limit()` functions.

tracemalloc.stop()

Stop tracing Python memory allocations: uninstall hooks on Python memory allocators. Also clears all previously collected traces of memory blocks allocated by Python.

Call `take_snapshot()` function to take a snapshot of traces before clearing them.

See also `start()`, `is_tracing()` and `clear_traces()` functions.

tracemalloc.take_snapshot()

Take a snapshot of traces of memory blocks allocated by Python. Return a new `Snapshot` instance.

The snapshot does not include memory blocks allocated before the `tracemalloc` module started to trace memory allocations.

Tracebacks of traces are limited to `get_traceback_limit()` frames. Use the *nframe* parameter of the `start()` function to store more frames.

The `tracemalloc` module must be tracing memory allocations to take a snapshot, see the `start()` function.

See also the `get_object_traceback()` function.

DomainFilter

class tracemalloc.DomainFilter(*inclusive: bool*, *domain: int*)

Filter traces of memory blocks by their address space (domain).

Added in version 3.6.

inclusive

If *inclusive* is `True` (include), match memory blocks allocated in the address space `domain`.

If *inclusive* is `False` (exclude), match memory blocks not allocated in the address space `domain`.

domain

Address space of a memory block (`int`). Read-only property.

Filter

class tracemalloc.Filter(*inclusive: bool*, *filename_pattern: str*, *lineno: int = None*, *all_frames: bool = False*, *domain: int = None*)

Filter on traces of memory blocks.

See the `fnmatch.fnmatch()` function for the syntax of *filename_pattern*. The `'.pyc'` file extension is replaced with `'.py'`.

Examples:

- `Filter(True, subprocess.__file__)` only includes traces of the `subprocess` module

- `Filter(False, tracemalloc.__file__)` excludes traces of the **`tracemalloc`** module
- `Filter(False, "<unknown>")` excludes empty tracebacks

Changed in version 3.5: The `'.pyo'` file extension is no longer replaced with `'.py'`.

Changed in version 3.6: Added the **`domain`** attribute.

domain

Address space of a memory block (`int` or `None`).

tracemalloc uses the domain `0` to trace memory allocations made by Python. C extensions can use other domains to trace other resources.

inclusive

If *inclusive* is `True` (include), only match memory blocks allocated in a file with a name matching **`filename_pattern`** at line number **`lineno`**.

If *inclusive* is `False` (exclude), ignore memory blocks allocated in a file with a name matching **`filename_pattern`** at line number **`lineno`**.

lineno

Line number (`int`) of the filter. If *lineno* is `None`, the filter matches any line number.

filename_pattern

Filename pattern of the filter (`str`). Read-only property.

all_frames

If *all_frames* is `True`, all frames of the traceback are checked. If *all_frames* is `False`, only the most recent frame is checked.

This attribute has no effect if the traceback limit is `1`. See the `get_traceback_limit()` function and `Snapshot.traceback_limit` attribute.

Frame

class tracemalloc.Frame

Frame of a traceback.

The `Traceback` class is a sequence of `Frame` instances.

filename

Filename (`str`).

lineno

Line number (`int`).

Snapshot

class tracemalloc.Snapshot

Snapshot of traces of memory blocks allocated by Python.

The `take_snapshot()` function creates a snapshot instance.

compare_to(*old_snapshot: Snapshot*, *key_type: str*, *cumulative: bool = False*)

Compute the differences with an old snapshot. Get statistics as a sorted list of `StatisticDiff` instances grouped by *key_type*.

See the `Snapshot.statistics()` method for *key_type* and *cumulative* parameters.

The result is sorted from the biggest to the smallest by: absolute value of `StatisticDiff.size_diff`, `StatisticDiff.size`, absolute value of `StatisticDiff.count_diff`, `Statistic.count` and then by `StatisticDiff.traceback`.

dump(*filename*)

Write the snapshot into a file.

Use `load()` to reload the snapshot.

filter_traces(*filters*)

Create a new `Snapshot` instance with a filtered `traces` sequence, *filters* is a list of `DomainFilter` and `Filter` instances. If *filters* is an empty list, return a new `Snapshot` instance with a copy of the traces.

All inclusive filters are applied at once, a trace is ignored if no inclusive filters match it. A trace is ignored if at least one exclusive filter matches it.

Changed in version 3.6: `DomainFilter` instances are now also accepted in *filters*.

classmethod load(*filename*)

Load a snapshot from a file.

See also `dump()`.

statistics(*key_type: str*, *cumulative: bool = False*)

Get statistics as a sorted list of `Statistic` instances grouped by *key_type*:

key_type	description
`'filename'`	filename
`'lineno'`	filename and line number

key_type	description
`'traceback'`	traceback

If *cumulative* is `True`, cumulate size and count of memory blocks of all frames of the traceback of a trace, not only the most recent frame. The cumulative mode can only be used with *key_type* equals to `'filename'` and `'lineno'`.

The result is sorted from the biggest to the smallest by: **`Statistic.size`**, **`Statistic.count`** and then by **`Statistic.traceback`**.

traceback_limit

Maximum number of frames stored in the traceback of **`traces`**: result of the **`get_traceback_limit()`** when the snapshot was taken.

traces

Traces of all memory blocks allocated by Python: sequence of **`Trace`** instances.

The sequence has an undefined order. Use the **`Snapshot.statistics()`** method to get a sorted list of statistics.

Statistic

class tracemalloc.Statistic

Statistic on memory allocations.

`Snapshot.statistics()` returns a list of **`Statistic`** instances.

See also the **`StatisticDiff`** class.

count

Number of memory blocks (`int`).

size

Total size of memory blocks in bytes (`int`).

traceback

Traceback where the memory block was allocated, `Traceback` instance.

StatisticDiff

class tracemalloc.StatisticDiff

Statistic difference on memory allocations between an old and a new `Snapshot` instance.

`Snapshot.compare_to()` returns a list of `StatisticDiff` instances. See also the `Statistic` class.

count

Number of memory blocks in the new snapshot (`int`): `0` if the memory blocks have been released in the new snapshot.

count_diff

Difference of number of memory blocks between the old and the new snapshots (`int`): `0` if the memory blocks have been allocated in the new snapshot.

size

Total size of memory blocks in bytes in the new snapshot (`int`): `0` if the memory blocks have been released in the new snapshot.

size_diff

Difference of total size of memory blocks in bytes between the old and the new snapshots (`int`): `0` if the memory blocks have been allocated in the new snapshot.

traceback

Traceback where the memory blocks were allocated, `Traceback` instance.

Trace

class tracemalloc.Trace

Trace of a memory block.

The `Snapshot.traces` attribute is a sequence of `Trace` instances.

Changed in version 3.6: Added the `domain` attribute.

domain

Address space of a memory block (`int`). Read-only property.

tracemalloc uses the domain `0` to trace memory allocations made by Python. C extensions can use other domains to trace other resources.

size

Size of the memory block in bytes (`int`).

traceback

Traceback where the memory block was allocated, `Traceback` instance.

Traceback

class tracemalloc.Traceback

Sequence of `Frame` instances sorted from the oldest frame to the most recent frame.

A traceback contains at least `1` frame. If the `tracemalloc` module failed to get a frame, the filename `"<unknown>"` at line number `0` is used.

When a snapshot is taken, tracebacks of traces are limited to `get_traceback_limit()` frames. See the `take_snapshot()` function. The original number of frames of the traceback is stored in the `Traceback.total_nframe` attribute. That allows to know if a traceback has been truncated by the traceback limit.

The `Trace.traceback` attribute is an instance of `Traceback` instance.

Changed in version 3.7: Frames are now sorted from the oldest to the most recent, instead of most recent to oldest.

total_nframe

Total number of frames that composed the traceback before truncation. This attribute can be set to `None` if the information is not available.

Changed in version 3.9: The `Traceback.total_nframe` attribute was added.

format(*limit=None*, *most_recent_first=False*)

Format the traceback as a list of lines. Use the `linecache` module to retrieve lines from the source code. If *limit* is set, format the *limit* most recent frames if *limit* is positive. Otherwise, format the `abs(limit)` oldest frames. If *most_recent_first* is `True`, the order of the formatted frames is reversed, returning the most recent frame first instead of last.

Similar to the `traceback.format_tb()` function, except that `format()` does not include newlines.

Example:

```
print("Traceback (most recent call first):")
for line in traceback:
    print(line)
```

Output:

```
Traceback (most recent call first):
  File "test.py", line 9
    obj = Object()
  File "test.py", line 12
    tb = tracemalloc.get_object_traceback(f())
```

Software Packaging and Distribution

These libraries help you with publishing and installing Python software. While these modules are designed to work in conjunction with the Python Package Index [https://pypi.org], they can also be used with a local index server, or without any index server at all.

- `ensurepip` — Bootstrapping the `pip` installer
    - Command-line interface
    - Module API
- `venv` — Creation of virtual environments
    - Creating virtual environments
    - How venvs work
    - API
    - An example of extending `EnvBuilder`
- `zipapp` — Manage executable Python zip archives
    - Basic Example
    - Command-Line Interface
    - Python API
    - Examples
    - Specifying the Interpreter
    - Creating Standalone Applications with zipapp
        - Caveats
    - The Python Zip Application Archive Format

`ensurepip` — Bootstrapping the `pip` installer

Added in version 3.4.

Source code: Lib/ensurepip [https://github.com/python/cpython/tree/3.14/Lib/ensurepip]

The **`ensurepip`** package provides support for bootstrapping the `pip` installer into an existing Python installation or virtual environment. This bootstrapping approach reflects the fact that `pip` is an independent project with its own release cycle, and the latest available stable version is bundled with maintenance and feature releases of the CPython reference interpreter.

In most cases, end users of Python shouldn't need to invoke this module directly (as `pip` should be bootstrapped by default), but it may be needed if installing `pip` was skipped when installing Python (or when creating a virtual environment) or after explicitly uninstalling `pip`.

Note

This module *does not* access the internet. All of the components needed to bootstrap `pip` are included as internal parts of the package.

This is an optional module. If it is missing from your copy of CPython, look for documentation from your distributor (that is, whoever provided Python to you). If you are the distributor, see Requirements for optional modules.

See also

Installing Python Modules
The end user guide for installing Python packages

PEP 453 [https://peps.python.org/pep-0453/]: Explicit bootstrapping of pip in Python installations
The original rationale and specification for this module.

Availability: not Android, not iOS, not WASI.

This module is not supported on mobile platforms or WebAssembly platforms.

Command-line interface

The command line interface is invoked using the interpreter’s `-m` switch.

The simplest possible invocation is:

```
python -m ensurepip
```

This invocation will install `pip` if it is not already installed, but otherwise does nothing. To ensure the installed version of `pip` is at least as recent as the one available in `ensurepip`, pass the `--upgrade` option:

```
python -m ensurepip --upgrade
```

By default, `pip` is installed into the current virtual environment (if one is active) or into the system site packages (if there is no active virtual environment). The installation location can be controlled through two additional command line options:

--root <dir>
: Installs `pip` relative to the given root directory rather than the root of the currently active virtual environment (if any) or the default root for the current Python installation.

--user
: Installs `pip` into the user site packages directory rather than globally for the current Python installation (this option is not permitted inside an active virtual environment).

By default, the scripts `pipX` and `pipX.Y` will be installed (where X.Y stands for the version of Python used to invoke `ensurepip`). The scripts installed can be controlled through two additional command line options:

--altinstall
: If an alternate installation is requested, the `pipX` script will *not* be installed.

--default-pip
: If a “default pip” installation is requested, the `pip` script will be installed in addition to the two regular scripts.

Providing both of the script selection options will trigger an exception.

Module API

`ensurepip` exposes two functions for programmatic use:

ensurepip.version()

Returns a string specifying the available version of pip that will be installed when bootstrapping an environment.

ensurepip.bootstrap(*root=None*, *upgrade=False*, *user=False*, *altinstall=False*, *default_pip=False*, *verbosity=0*)

Bootstraps `pip` into the current or designated environment.

root specifies an alternative root directory to install relative to. If *root* is `None`, then installation uses the default install location for the current environment.

upgrade indicates whether or not to upgrade an existing installation of an earlier version of `pip` to the available version.

user indicates whether to use the user scheme rather than installing globally.

By default, the scripts `pipX` and `pipX.Y` will be installed (where X.Y stands for the current version of Python).

If *altinstall* is set, then `pipX` will *not* be installed.

If *default_pip* is set, then `pip` will be installed in addition to the two regular scripts.

Setting both *altinstall* and *default_pip* will trigger **`ValueError`**.

verbosity controls the level of output to **`sys.stdout`** from the bootstrapping operation.

Raises an auditing event `ensurepip.bootstrap` with argument `root`.

Note

The bootstrapping process has side effects on both `sys.path` and `os.environ`. Invoking the command line interface in a subprocess instead allows these side effects to be avoided.

Note

The bootstrapping process may install additional modules required by `pip`, but other software should not assume those dependencies will always be present by default (as the dependencies may be removed in a future version of `pip`).

`venv` — Creation of virtual environments

Added in version 3.3.

Source code: Lib/venv/
[https://github.com/python/cpython/tree/3.14/Lib/venv/]

The **`venv`** module supports creating lightweight “virtual environments”, each with their own independent set of Python packages installed in their **`site`** directories. A virtual environment is created on top of an existing Python installation, known as the virtual environment’s “base” Python, and by default is isolated from the packages in the base environment, so that only those explicitly installed in the virtual environment are available. See Virtual Environments and **`site`**’s virtual environments documentation for more information.

When used from within a virtual environment, common installation tools such as pip [https://pypi.org/project/pip/] will install Python packages into a virtual environment without needing to be told to do so explicitly.

A virtual environment is (amongst other things):

- Used to contain a specific Python interpreter and software libraries and binaries which are needed to support a project (library or application). These are by default isolated from software in other virtual environments and Python interpreters and libraries installed in the operating system.

- Contained in a directory, conventionally named `.venv` or `venv` in the project directory, or under a container directory for lots of virtual environments, such as `~/.virtualenvs`.
- Not checked into source control systems such as Git.
- Considered as disposable – it should be simple to delete and recreate it from scratch. You don’t place any project code in the environment.
- Not considered as movable or copyable – you just recreate the same environment in the target location.

See **PEP 405** [https://peps.python.org/pep-0405/] for more background on Python virtual environments.

See also

Python Packaging User Guide: Creating and using virtual environments [https://packaging.python.org/guides/installing-using-pip-and-virtual-environments/#create-and-use-virtual-environments]

Availability: not Android, not iOS, not WASI.

This module is not supported on mobile platforms or WebAssembly platforms.

Creating virtual environments

Virtual environments are created by executing the `venv` module:

```
python -m venv /path/to/new/virtual/environment
```

This creates the target directory (including parent directories as needed) and places a `pyvenv.cfg` file in it with a `home` key pointing to the Python installation from which the command was run. It also creates a `bin` (or `Scripts` on Windows) subdirectory containing a copy or symlink of the Python executable (as appropriate for the platform or arguments used at environment creation time). It also creates a `lib/pythonX.Y/site-packages` subdirectory (on Windows, this is `Lib\site-packages`). If an existing directory is specified, it will be re-used.

Changed in version 3.5: The use of `venv` is now recommended for creating virtual environments.

Deprecated since version 3.6, removed in version 3.8: **pyvenv** was the recommended tool for creating virtual environments for Python 3.3 and 3.4, and replaced in 3.5 by executing `venv` directly.

On Windows, invoke the `venv` command as follows:

```
PS> python -m venv C:\path\to\new\virtual\environment
```

The command, if run with `-h`, will show the available options:

```
usage: venv [-h] [--system-site-packages] [--symlinks | --
copies] [--clear]
            [--upgrade] [--without-pip] [--prompt PROMPT] [--
upgrade-deps]
            [--without-scm-ignore-files]
            ENV_DIR [ENV_DIR ...]

Creates virtual Python environments in one or more target
directories.

Once an environment has been created, you may wish to activate
it, e.g. by
sourcing an activate script in its bin directory.
```

ENV_DIR

A required argument specifying the directory to create the environment in.

--system-site-packages

Give the virtual environment access to the system site-packages directory.

--symlinks

Try to use symlinks rather than copies, when symlinks are not the default for the platform.

--copies

Try to use copies rather than symlinks, even when symlinks are the default for the platform.

--clear

Delete the contents of the environment directory if it already exists, before environment creation.

--upgrade

Upgrade the environment directory to use this version of Python, assuming Python has been upgraded in-place.

--without-pip

Skips installing or upgrading pip in the virtual environment (pip is bootstrapped by default).

--prompt <PROMPT>

Provides an alternative prompt prefix for this environment.

--upgrade-deps

Upgrade core dependencies (pip) to the latest version in PyPI.

--without-scm-ignore-files

Skips adding SCM ignore files to the environment directory (Git is supported by default).

Changed in version 3.4: Installs pip by default, added the `--without-pip` and `--copies` options.

Changed in version 3.4: In earlier versions, if the target directory already existed, an error was raised, unless the `--clear` or `--upgrade` option was provided.

Changed in version 3.9: Add `--upgrade-deps` option to upgrade pip + setuptools to the latest on PyPI.

Changed in version 3.12: `setuptools` is no longer a core venv dependency.

Changed in version 3.13: Added the `--without-scm-ignore-files` option.

Changed in version 3.13: `venv` now creates a `.gitignore` file for Git by default.

Note

While symlinks are supported on Windows, they are not recommended. Of particular note is that double-clicking `python.exe` in File Explorer will resolve the symlink eagerly and ignore the virtual environment.

Note

On Microsoft Windows, it may be required to enable the `Activate.ps1` script by setting the execution policy for the user. You can do this by issuing the following PowerShell command:

```
PS C:\> Set-ExecutionPolicy -ExecutionPolicy RemoteSigned -Scope CurrentUser
```

See About Execution Policies [https://go.microsoft.com/fwlink/?LinkID=135170] for more information.

The created `pyvenv.cfg` file also includes the `include-system-site-packages` key, set to `true` if `venv` is run with the `--system-site-packages` option, `false` otherwise.

Unless the `--without-pip` option is given, **`ensurepip`** will be invoked to bootstrap `pip` into the virtual environment.

Multiple paths can be given to `venv`, in which case an identical virtual environment will be created, according to the given options, at each provided path.

How venvs work

When a Python interpreter is running from a virtual environment, **`sys.prefix`** and **`sys.exec_prefix`** point to the directories of the virtual environment, whereas **`sys.base_prefix`** and **`sys.base_exec_prefix`** point to those of the base Python used to create the environment. It is sufficient to check `sys.prefix != sys.base_prefix` to determine if the current interpreter is running from a virtual environment.

A virtual environment may be “activated” using a script in its binary directory (`bin` on POSIX; `Scripts` on Windows). This will prepend that directory to your **`PATH`**, so that running **python** will invoke the environment’s Python interpreter and you can run installed scripts without having to use their full path. The invocation of the activation script is platform-specific (*<venv>* must be replaced by the path to the directory containing the virtual environment):

Platform	Shell	Command to activate virtual environment
POSIX	bash/zsh	`$ source <venv>/bin/activate`
	fish	`$ source <venv>/bin/activate.fish`
	csh/tcsh	`$ source <venv>/bin/activate.csh`
	pwsh	`$ <venv>/bin/Activate.ps1`
Windows	cmd.exe	`C:\> <venv>\Scripts\activate.bat`
	PowerShell	`PS C:\> <venv>\Scripts\Activate.ps1`

Added in version 3.4: **fish** and **csh** activation scripts.

Added in version 3.8: PowerShell activation scripts installed under POSIX for PowerShell Core support.

You don't specifically *need* to activate a virtual environment, as you can just specify the full path to that environment's Python interpreter when invoking Python. Furthermore, all scripts installed in the environment should be runnable without activating it.

In order to achieve this, scripts installed into virtual environments have a "shebang" line which points to the environment's Python interpreter, `#!/<path-to-venv>/bin/python`. This means that the script will run with that interpreter regardless of the value of **`PATH`**. On Windows, “shebang” line processing is supported if you have the Python install manager installed. Thus, double-clicking an installed script in a Windows Explorer window should run it with the correct interpreter without the environment needing to be activated or on the **`PATH`**.

When a virtual environment has been activated, the **`VIRTUAL_ENV`** environment variable is set to the path of the environment. Since explicitly activating a virtual environment is not required to use it, **`VIRTUAL_ENV`** cannot be relied upon to determine whether a virtual environment is being used.

Warning

Because scripts installed in environments should not expect the environment to be activated, their shebang lines contain the absolute paths to their environment’s interpreters. Because of this, environments are inherently non-portable, in the general case. You should always have a simple means of recreating an environment (for example, if you have a requirements file `requirements.txt`, you can invoke `pip install -r requirements.txt` using the environment’s `pip` to install all of the packages needed by the environment). If for any reason you need to move the environment to a new location, you should recreate it at the desired location and delete the one at the old location. If you move an environment because you moved a parent directory of it, you should recreate the environment in its new location. Otherwise, software installed into the environment may not work as expected.

You can deactivate a virtual environment by typing `deactivate` in your shell. The exact mechanism is platform-specific and is an internal implementation detail (typically, a script or shell function will be used).

API

The high-level method described above makes use of a simple API which provides mechanisms for third-party virtual environment creators to customize environment creation according to their needs, the `EnvBuilder` class.

class venv.EnvBuilder(*system_site_packages=False*, *clear=False*, *symlinks=False*, *upgrade=False*, *with_pip=False*, *prompt=None*, *upgrade_deps=False*, ***, *scm_ignore_files=frozenset()*)

The `EnvBuilder` class accepts the following keyword arguments on instantiation:

- *system_site_packages* – a boolean value indicating that the system Python site-packages should be available to the environment (defaults to `False`).
- *clear* – a boolean value which, if true, will delete the contents of any existing target directory, before creating the environment.
- *symlinks* – a boolean value indicating whether to attempt to symlink the Python binary rather than copying.
- *upgrade* – a boolean value which, if true, will upgrade an existing environment with the running Python - for use when that Python has been upgraded in-place (defaults to `False`).
- *with_pip* – a boolean value which, if true, ensures pip is installed in the virtual environment. This uses `ensurepip` with the `--default-pip` option.
- *prompt* – a string to be used after virtual environment is activated (defaults to `None` which

means directory name of the environment would be used). If the special string `"."` is provided, the basename of the current directory is used as the prompt.
- *upgrade_deps* – Update the base venv modules to the latest on PyPI
- *scm_ignore_files* – Create ignore files based for the specified source control managers (SCM) in the iterable. Support is defined by having a method named `create_{scm}_ignore_file`. The only value supported by default is `"git"` via **`create_git_ignore_file()`**.

Changed in version 3.4: Added the `with_pip` parameter

Changed in version 3.6: Added the `prompt` parameter

Changed in version 3.9: Added the `upgrade_deps` parameter

Changed in version 3.13: Added the `scm_ignore_files` parameter

`EnvBuilder` may be used as a base class.

create(*env_dir*)

Create a virtual environment by specifying the target directory (absolute or relative to the current directory) which is to contain the virtual environment. The `create` method will either create the environment in the specified directory, or raise an appropriate exception.

The `create` method of the **`EnvBuilder`** class illustrates the hooks available for subclass customization:

```
def create(self, env_dir):
    """
    Create a virtualized Python environment in a
```

```
directory.
    env_dir is the target directory to create an
environment in.
    """
    env_dir = os.path.abspath(env_dir)
    context = self.ensure_directories(env_dir)
    self.create_configuration(context)
    self.setup_python(context)
    self.setup_scripts(context)
    self.post_setup(context)
```

Each of the methods `ensure_directories()`, `create_configuration()`, `setup_python()`, `setup_scripts()` and `post_setup()` can be overridden.

ensure_directories(*env_dir*)

Creates the environment directory and all necessary subdirectories that don’t already exist, and returns a context object. This context object is just a holder for attributes (such as paths) for use by the other methods. If the `EnvBuilder` is created with the arg `clear=True`, contents of the environment directory will be cleared and then all necessary subdirectories will be recreated.

The returned context object is a `types.SimpleNamespace` with the following attributes:

- `env_dir` - The location of the virtual environment. Used for `__VENV_DIR__` in activation scripts (see `install_scripts()`).
- `env_name` - The name of the virtual environment. Used for `__VENV_NAME__` in activation scripts (see `install_scripts()`).
- `prompt` - The prompt to be used by the activation scripts. Used for `__VENV_PROMPT__` in activation scripts (see `install_scripts()`).
- `executable` - The underlying Python executable used by the virtual environment. This takes into

account the case where a virtual environment is created from another virtual environment.

- `inc_path` - The include path for the virtual environment.
- `lib_path` - The purelib path for the virtual environment.
- `bin_path` - The script path for the virtual environment.
- `bin_name` - The name of the script path relative to the virtual environment location. Used for `__VENV_BIN_NAME__` in activation scripts (see **`install_scripts()`**).
- `env_exe` - The name of the Python interpreter in the virtual environment. Used for `__VENV_PYTHON__` in activation scripts (see **`install_scripts()`**).
- `env_exec_cmd` - The name of the Python interpreter, taking into account filesystem redirections. This can be used to run Python in the virtual environment.

Changed in version 3.11: The *venv* sysconfig installation scheme is used to construct the paths of the created directories.

Changed in version 3.12: The attribute `lib_path` was added to the context, and the context object was documented.

create_configuration(*context*)

Creates the `pyvenv.cfg` configuration file in the environment.

setup_python(*context*)

Creates a copy or symlink to the Python executable in the environment. On POSIX systems, if a specific executable `python3.x` was used, symlinks to `python` and `python3` will be created pointing to that executable, unless files with those names already exist.

setup_scripts(*context*)

Installs activation scripts appropriate to the platform into the virtual environment.

upgrade_dependencies(*context*)

Upgrades the core venv dependency packages (currently pip [https://pypi.org/project/pip/]) in the environment. This is done by shelling out to the `pip` executable in the environment.

Added in version 3.9.

Changed in version 3.12: setuptools [https://pypi.org/project/setuptools/] is no longer a core venv dependency.

post_setup(*context*)

A placeholder method which can be overridden in third party implementations to pre-install packages in the virtual environment or perform other post-creation steps.

install_scripts(*context*, *path*)

This method can be called from **`setup_scripts()`** or **`post_setup()`** in subclasses to assist in installing custom scripts into the virtual environment.

path is the path to a directory that should contain subdirectories `common`, `posix`, `nt`; each containing scripts destined for the `bin` directory in the environment. The contents of `common` and the directory corresponding to **`os.name`** are copied after some text replacement of placeholders:

- `__VENV_DIR__` is replaced with the absolute path of the environment directory.
- `__VENV_NAME__` is replaced with the environment name (final path segment of environment directory).
- `__VENV_PROMPT__` is replaced with the prompt (the environment name surrounded by parentheses and with a following space)
- `__VENV_BIN_NAME__` is replaced with the name of the bin directory (either `bin` or `Scripts`).
- `__VENV_PYTHON__` is replaced with the absolute path of the environment's executable.

The directories are allowed to exist (for when an existing environment is being upgraded).

create_git_ignore_file(*context*)

Creates a `.gitignore` file within the virtual environment that causes the entire directory to be ignored by the Git source control manager.

Added in version 3.13.

Changed in version 3.7.2: Windows now uses redirector scripts for `python[w].exe` instead of copying the actual binaries. In 3.7.2 only `setup_python()` does nothing unless running from a build in the source tree.

Changed in version 3.7.3: Windows copies the redirector scripts as part of `setup_python()` instead of `setup_scripts()`. This was not the case in 3.7.2. When using symlinks, the original executables will be linked.

There is also a module-level convenience function:

venv.create(*env_dir*, *system_site_packages=False*, *clear=False*, *symlinks=False*, *with_pip=False*, *prompt=None*, *upgrade_deps=False*, ***, *scm_ignore_files=frozenset()*)

Create an `EnvBuilder` with the given keyword arguments, and call its `create()` method with the *env_dir* argument.

Added in version 3.3.

Changed in version 3.4: Added the *with_pip* parameter

Changed in version 3.6: Added the *prompt* parameter

Changed in version 3.9: Added the *upgrade_deps* parameter

Changed in version 3.13: Added the *scm_ignore_files* parameter

An example of extending `EnvBuilder`

The following script shows how to extend `EnvBuilder` by implementing a subclass which installs setuptools and pip into a created virtual environment:

```
import os
import os.path
from subprocess import Popen, PIPE
import sys
from threading import Thread
from urllib.parse import urlparse
from urllib.request import urlretrieve
import venv

class ExtendedEnvBuilder(venv.EnvBuilder):
    """
    This builder installs setuptools and pip so that you can
pip or
    easy_install other packages into the created virtual
environment.
```

```
    :param nodist: If true, setuptools and pip are not
installed into the
                   created virtual environment.
    :param nopip: If true, pip is not installed into the
created
                  virtual environment.
    :param progress: If setuptools or pip are installed, the
progress of the
                     installation can be monitored by passing a
progress
                     callable. If specified, it is called with
two
                     arguments: a string indicating some
progress, and a
                     context indicating where the string is
coming from.
                     The context argument can have one of three
values:
                     'main', indicating that it is called from
virtualize()
                     itself, and 'stdout' and 'stderr', which
are obtained
                     by reading lines from the output streams
of a subprocess
                     which is used to install the app.

                     If a callable is not specified, default
progress
                     information is output to sys.stderr.
    """

    def __init__(self, *args, **kwargs):
        self.nodist = kwargs.pop('nodist', False)
        self.nopip = kwargs.pop('nopip', False)
        self.progress = kwargs.pop('progress', None)
        self.verbose = kwargs.pop('verbose', False)
        super().__init__(*args, **kwargs)

    def post_setup(self, context):
        """
        Set up any packages which need to be pre-installed into
the
        virtual environment being created.
        :param context: The information for the virtual
environment
                        creation request being processed.
        """
        os.environ['VIRTUAL_ENV'] = context.env_dir
        if not self.nodist:
            self.install_setuptools(context)
        # Can't install pip without setuptools
        if not self.nopip and not self.nodist:
            self.install_pip(context)

    def reader(self, stream, context):
        """
        Read lines from a subprocess' output stream and either
pass to a progress
        callable (if specified) or write progress information
to sys.stderr.
        """
        progress = self.progress
        while True:
            s = stream.readline()
            if not s:
                break
            if progress is not None:
                progress(s, context)
            else:
                if not self.verbose:
                    sys.stderr.write('.')
                else:
                    sys.stderr.write(s.decode('utf-8'))
                sys.stderr.flush()
        stream.close()

    def install_script(self, context, name, url):
        _, _, path, _, _, _ = urlparse(url)
        fn = os.path.split(path)[-1]
        binpath = context.bin_path
        distpath = os.path.join(binpath, fn)
        # Download script into the virtual environment's
binaries folder
        urlretrieve(url, distpath)
        progress = self.progress
        if self.verbose:
            term = '\n'
        else:
            term = ''
        if progress is not None:
            progress('Installing %s ...%s' % (name, term),
'main')
        else:
            sys.stderr.write('Installing %s ...%s' % (name,
term))
            sys.stderr.flush()
        # Install in the virtual environment
        args = [context.env_exe, fn]
        p = Popen(args, stdout=PIPE, stderr=PIPE, cwd=binpath)
        t1 = Thread(target=self.reader, args=(p.stdout,
'stdout'))
        t1.start()
        t2 = Thread(target=self.reader, args=(p.stderr,
'stderr'))
        t2.start()
        p.wait()
        t1.join()
        t2.join()
        if progress is not None:
            progress('done.', 'main')
        else:
            sys.stderr.write('done.\n')
        # Clean up - no longer needed
        os.unlink(distpath)

    def install_setuptools(self, context):
        """
        Install setuptools in the virtual environment.

        :param context: The information for the virtual
environment
                        creation request being processed.
        """
        url = "https://bootstrap.pypa.io/ez_setup.py"
        self.install_script(context, 'setuptools', url)
        # clear up the setuptools archive which gets downloaded
        pred = lambda o: o.startswith('setuptools-') and
o.endswith('.tar.gz')
```

```
        files = filter(pred, os.listdir(context.bin_path))
        for f in files:
            f = os.path.join(context.bin_path, f)
            os.unlink(f)

    def install_pip(self, context):
        """
        Install pip in the virtual environment.

        :param context: The information for the virtual
environment
                        creation request being processed.
        """
        url = 'https://bootstrap.pypa.io/get-pip.py'
        self.install_script(context, 'pip', url)


def main(args=None):
    import argparse

    parser = argparse.ArgumentParser(prog=__name__,
                                     description='Creates
virtual Python '
                                                 'environments
in one or '
                                                 'more target '

'directories.')
    parser.add_argument('dirs', metavar='ENV_DIR', nargs='+',
                        help='A directory in which to create
the '
                             'virtual environment.')
    parser.add_argument('--no-setuptools', default=False,
                        action='store_true', dest='nodist',
                        help="Don't install setuptools or pip
in the "
                             "virtual environment.")
    parser.add_argument('--no-pip', default=False,
                        action='store_true', dest='nopip',
                        help="Don't install pip in the virtual
"
                             "environment.")
    parser.add_argument('--system-site-packages',
default=False,
                        action='store_true',
dest='system_site',
                        help='Give the virtual environment
access to the '
                             'system site-packages dir.')
    if os.name == 'nt':
        use_symlinks = False
    else:
        use_symlinks = True
    parser.add_argument('--symlinks', default=use_symlinks,
                        action='store_true', dest='symlinks',
                        help='Try to use symlinks rather than
copies, '
                             'when symlinks are not the default
for '
                             'the platform.')
    parser.add_argument('--clear', default=False,
action='store_true',
                        dest='clear', help='Delete the contents
of the '
                                           'virtual environment
'
                                           'directory if it
already '
                                           'exists, before
virtual '
                                           'environment
creation.')
    parser.add_argument('--upgrade', default=False,
action='store_true',
                        dest='upgrade', help='Upgrade the
virtual '
                                             'environment
directory to '
                                             'use this version
of '
                                             'Python, assuming
Python '
                                             'has been upgraded
'
                                             'in-place.')
    parser.add_argument('--verbose', default=False,
action='store_true',
                        dest='verbose', help='Display the
output '
                                             'from the scripts
which '
                                             'install
setuptools and pip.')
    options = parser.parse_args(args)
    if options.upgrade and options.clear:
        raise ValueError('you cannot supply --upgrade and --
clear together.')
    builder =
ExtendedEnvBuilder(system_site_packages=options.system_site,
                                   clear=options.clear,
                                   symlinks=options.symlinks,
                                   upgrade=options.upgrade,
                                   nodist=options.nodist,
                                   nopip=options.nopip,
                                   verbose=options.verbose)
    for d in options.dirs:
        builder.create(d)

if __name__ == '__main__':
    rc = 1
    try:
        main()
        rc = 0
    except Exception as e:
        print('Error: %s' % e, file=sys.stderr)
    sys.exit(rc)
```

This script is also available for download online [https://gist.github.com/vsajip/4673395].

`zipapp` — Manage executable Python zip archives

Added in version 3.5.

Source code: Lib/zipapp.py
[https://github.com/python/cpython/tree/3.14/Lib/zipapp.py]

This module provides tools to manage the creation of zip files containing Python code, which can be executed directly by the Python interpreter. The module provides both a Command-Line Interface and a Python API.

Basic Example

The following example shows how the Command-Line Interface can be used to create an executable archive from a directory containing Python code. When run, the archive will execute the `main` function from the module `myapp` in the archive.

```
$ python -m zipapp myapp -m "myapp:main"
$ python myapp.pyz
<output from myapp>
```

Command-Line Interface

When called as a program from the command line, the following form is used:

```
$ python -m zipapp source [options]
```

If *source* is a directory, this will create an archive from the contents of *source*. If *source* is a file, it should be an archive, and it will be copied to the target archive (or the contents of its shebang line will be displayed if the –info option is specified).

The following options are understood:

-o <output>, --output=<output>

Write the output to a file named *output*. If this option is not specified, the output filename will be the same as the input *source*, with the extension `.pyz` added. If an explicit filename is given, it is used as is (so a `.pyz` extension should be included if required).

An output filename must be specified if the *source* is an archive (and in that case, *output* must not be the same as *source*).

-p <interpreter>, --python=<interpreter>

Add a `#!` line to the archive specifying *interpreter* as the command to run. Also, on POSIX, make the archive executable. The default is to write no `#!` line, and not make the file executable.

-m <mainfn>, --main=<mainfn>

Write a `__main__.py` file to the archive that executes *mainfn*. The *mainfn* argument should have the form “pkg.mod:fn”, where “pkg.mod” is a package/module in the archive, and “fn” is a callable in the given module. The `__main__.py` file will execute that callable.

`--main` cannot be specified when copying an archive.

-c, --compress

Compress files with the deflate method, reducing the size of the output file. By default, files are stored uncompressed in the archive.

`--compress` has no effect when copying an archive.

Added in version 3.7.

--info

Display the interpreter embedded in the archive, for diagnostic purposes. In this case, any other options are ignored and SOURCE must be an archive, not a directory.

-h, --help

Print a short usage message and exit.

Python API

The module defines two convenience functions:

zipapp.create_archive(*source*, *target=None*, *interpreter=None*, *main=None*, *filter=None*, *compressed=False*)

Create an application archive from *source*. The source can be any of the following:

- The name of a directory, or a path-like object referring to a directory, in which case a new application archive will be created from the content of that directory.
- The name of an existing application archive file, or a path-like object referring to such a file, in which case the file is copied to the target (modifying it to reflect the value given for the *interpreter* argument). The

file name should include the `.pyz` extension, if required.
- A file object open for reading in bytes mode. The content of the file should be an application archive, and the file object is assumed to be positioned at the start of the archive.

The *target* argument determines where the resulting archive will be written:

- If it is the name of a file, or a path-like object, the archive will be written to that file.
- If it is an open file object, the archive will be written to that file object, which must be open for writing in bytes mode.
- If the target is omitted (or `None`), the source must be a directory and the target will be a file with the same name as the source, with a `.pyz` extension added.

The *interpreter* argument specifies the name of the Python interpreter with which the archive will be executed. It is written as a “shebang” line at the start of the archive. On POSIX, this will be interpreted by the OS, and on Windows it will be handled by the Python launcher. Omitting the *interpreter* results in no shebang line being written. If an interpreter is specified, and the target is a filename, the executable bit of the target file will be set.

The *main* argument specifies the name of a callable which will be used as the main program for the archive. It can only be specified if the source is a directory, and the source does not already contain a `__main__.py` file. The *main* argument should take the form “pkg.module:callable” and the archive will be run by importing “pkg.module” and executing the given callable with no arguments. It is an error to omit *main* if the source is a directory and does not contain a `__main__.py` file, as otherwise the resulting archive would not be executable.

The optional *filter* argument specifies a callback function that is passed a Path object representing the path to the file being added (relative to the source directory). It should return `True` if the file is to be added.

The optional *compressed* argument determines whether files are compressed. If set to `True`, files in the archive are compressed with the deflate method; otherwise, files are stored uncompressed. This argument has no effect when copying an existing archive.

If a file object is specified for *source* or *target*, it is the caller's responsibility to close it after calling create_archive.

When copying an existing archive, file objects supplied only need `read` and `readline`, or `write` methods. When creating an archive from a directory, if the target is a file object it will be passed to the `zipfile.ZipFile` class, and must supply the methods needed by that class.

Changed in version 3.7: Added the *filter* and *compressed* parameters.

zipapp.get_interpreter(*archive*)

Return the interpreter specified in the `#!` line at the start of the archive. If there is no `#!` line, return `None`. The *archive* argument can be a filename or a file-like object open for reading in bytes mode. It is assumed to be at the start of the archive.

Examples

Pack up a directory into an archive, and run it.

```
$ python -m zipapp myapp
$ python myapp.pyz
<output from myapp>
```

The same can be done using the `create_archive()` function:

```
>>> import zipapp
>>> zipapp.create_archive('myapp', 'myapp.pyz')
```

To make the application directly executable on POSIX, specify an interpreter to use.

```
$ python -m zipapp myapp -p "/usr/bin/env python"
$ ./myapp.pyz
<output from myapp>
```

To replace the shebang line on an existing archive, create a modified archive using the `create_archive()` function:

```
>>> import zipapp
>>> zipapp.create_archive('old_archive.pyz', 'new_archive.pyz',
'/usr/bin/python3')
```

To update the file in place, do the replacement in memory using a `BytesIO` object, and then overwrite the source afterwards. Note that there is a risk when overwriting a file in place that an error will result in the loss of the original file. This code does not protect against such errors, but production code should do so. Also, this method will only work if the archive fits in memory:

```
>>> import zipapp
>>> import io
>>> temp = io.BytesIO()
>>> zipapp.create_archive('myapp.pyz', temp,
'/usr/bin/python2')
>>> with open('myapp.pyz', 'wb') as f:
>>>     f.write(temp.getvalue())
```

Specifying the Interpreter

Note that if you specify an interpreter and then distribute your application archive, you need to ensure that the interpreter used is portable. The Python launcher for Windows supports most common forms of POSIX `#!` line, but there are other issues to consider:

- If you use “/usr/bin/env python” (or other forms of the “python” command, such as “/usr/bin/python”), you need to consider that your users may have either Python 2 or Python 3 as their default, and write your code to work under both versions.
- If you use an explicit version, for example “/usr/bin/env python3” your application will not work for users who do not have that version. (This may be what you want if you have not made your code Python 2 compatible).
- There is no way to say “python X.Y or later”, so be careful of using an exact version like “/usr/bin/env python3.4” as you will need to change your shebang line for users of Python 3.5, for example.

Typically, you should use an “/usr/bin/env python2” or “/usr/bin/env python3”, depending on whether your code is written for Python 2 or 3.

Creating Standalone Applications with zipapp

Using the `zipapp` module, it is possible to create self-contained Python programs, which can be distributed to end users who only need to have a suitable version of Python installed on their system. The key to doing this is to bundle all of the application’s dependencies into the archive, along with the application code.

The steps to create a standalone archive are as follows:

1. Create your application in a directory as normal, so you have a `myapp` directory containing a `__main__.py` file, and any supporting application code.

2. Install all of your application's dependencies into the `myapp` directory, using pip:

   ```
   $ python -m pip install -r requirements.txt --target myapp
   ```

   (this assumes you have your project requirements in a `requirements.txt` file - if not, you can just list the dependencies manually on the pip command line).

3. Package the application using:

   ```
   $ python -m zipapp -p "interpreter" myapp
   ```

This will produce a standalone executable, which can be run on any machine with the appropriate interpreter available. See Specifying the Interpreter for details. It can be shipped to users as a single file.

On Unix, the `myapp.pyz` file is executable as it stands. You can rename the file to remove the `.pyz` extension if you prefer a "plain" command name. On Windows, the `myapp.pyz[w]` file is executable by virtue of the fact that the Python interpreter registers the `.pyz` and `.pyzw` file extensions when installed.

Caveats

If your application depends on a package that includes a C extension, that package cannot be run from a zip file (this is an OS limitation, as executable code must be present in the filesystem for the OS loader to load it). In this case, you can exclude that dependency from the zipfile, and either require your users to have it installed, or ship it alongside your zipfile and add code to your `__main__.py` to include the directory containing the unzipped module in `sys.path`. In this case, you will need to make sure to ship appropriate binaries for your target architecture(s) (and potentially pick the correct version to add to `sys.path` at runtime, based on the user’s machine).

The Python Zip Application Archive Format

Python has been able to execute zip files which contain a `__main__.py` file since version 2.6. In order to be executed by Python, an application archive simply has to be a standard zip file containing a `__main__.py` file which will be run as the entry point for the application. As usual for any Python script, the parent of the script (in this case the zip file) will be placed on `sys.path` and thus further modules can be imported from the zip file.

The zip file format allows arbitrary data to be prepended to a zip file. The zip application format uses this ability to prepend a standard POSIX “shebang” line to the file (`#!/path/to/interpreter`).

Formally, the Python zip application format is therefore:

1. An optional shebang line, containing the characters `b'#!'` followed by an interpreter name, and then a newline (`b'\n'`) character. The interpreter name can be anything acceptable to the OS “shebang” processing, or the Python launcher on Windows. The interpreter should be encoded in UTF-8 on Windows, and in `sys.getfilesystemencoding()` on POSIX.

2. Standard zipfile data, as generated by the `zipfile` module. The zipfile content *must* include a file called `__main__.py` (which must be in the “root” of the zipfile - i.e., it cannot be in a subdirectory). The zipfile data can be compressed or uncompressed.

If an application archive has a shebang line, it may have the executable bit set on POSIX systems, to allow it to be executed directly.

There is no requirement that the tools in this module are used to create application archives - the module is a convenience, but archives in the above format created by any means are acceptable to Python.

Python Runtime Services

The modules described in this chapter provide a wide range of services related to the Python interpreter and its interaction with its environment. Here’s an overview:

- `sys` — System-specific parameters and functions
- `sys.monitoring` — Execution event monitoring
  - Tool identifiers
    - Registering and using tools
  - Events
    - Local events
    - Deprecated event
    - Ancillary events
    - Other events
    - The STOP_ITERATION event
  - Turning events on and off
    - Setting events globally
    - Per code object events
    - Disabling events
  - Registering callback functions
    - Callback function arguments
- `sysconfig` — Provide access to Python’s configuration information
  - Configuration variables
  - Installation paths
  - User scheme
    - `posix_user`
    - `nt_user`
    - `osx_framework_user`
  - Home scheme
    - `posix_home`
  - Prefix scheme

        - `posix_prefix`
        - `nt`
    - Installation path functions
    - Other functions
    - Command-line usage
- `builtins` — Built-in objects
- `__main__` — Top-level code environment
    - `__name__ == '__main__'`
        - What is the “top-level code environment”?
        - Idiomatic Usage
        - Packaging Considerations
    - `__main__.py` in Python Packages
        - Idiomatic Usage
    - `import __main__`
- `warnings` — Warning control
    - Warning Categories
    - The Warnings Filter
        - Repeated Warning Suppression Criteria
        - Describing Warning Filters
        - Default Warning Filter
        - Overriding the default filter
    - Temporarily Suppressing Warnings
    - Testing Warnings
    - Updating Code For New Versions of Dependencies
    - Available Functions
    - Available Context Managers
    - Concurrent safety of Context Managers
- `dataclasses` — Data Classes
    - Module contents
    - Post-init processing
    - Class variables
    - Init-only variables
    - Frozen instances
    - Inheritance
    - Re-ordering of keyword-only parameters in `__init__()`

    - Default factory functions
    - Mutable default values
    - Descriptor-typed fields
- `contextlib` — Utilities for `with`-statement contexts
    - Utilities
    - Examples and Recipes
        - Supporting a variable number of context managers
        - Catching exceptions from `__enter__` methods
        - Cleaning up in an `__enter__` implementation
        - Replacing any use of `try-finally` and flag variables
        - Using a context manager as a function decorator
    - Single use, reusable and reentrant context managers
        - Reentrant context managers
        - Reusable context managers
- `abc` — Abstract Base Classes
- `atexit` — Exit handlers
    - `atexit` Example
- `traceback` — Print or retrieve a stack traceback
    - Module-Level Functions
    - `TracebackException` Objects
    - `StackSummary` Objects
    - `FrameSummary` Objects
    - Examples of Using the Module-Level Functions
    - Examples of Using `TracebackException`
- `__future__` — Future statement definitions
    - Module Contents
- `gc` — Garbage Collector interface
- `inspect` — Inspect live objects
    - Types and members
    - Retrieving source code
    - Introspecting callables with the Signature object
    - Classes and functions

  - The interpreter stack
  - Fetching attributes statically
  - Current State of Generators, Coroutines, and Asynchronous Generators
  - Code Objects Bit Flags
  - Buffer flags
  - Command-line interface
- `annotationlib` — Functionality for introspecting annotations
  - Annotation semantics
  - Classes
  - Functions
  - Recipes
    - Using annotations in a metaclass
  - Limitations of the `STRING` format
  - Limitations of the `FORWARDREF` format
  - Security implications of introspecting annotations
- `site` — Site-specific configuration hook
  - `sitecustomize`
  - `usercustomize`
  - Readline configuration
  - Module contents
  - Command-line interface

See also

- See the `concurrent.interpreters` module, which similarly exposes core runtime functionality.

`sys` — System-specific parameters and functions

This module provides access to some variables used or maintained by the interpreter and to functions that interact strongly with the interpreter. It is always available. Unless explicitly noted otherwise, all variables are read-only.

sys.abiflags

On POSIX systems where Python was built with the standard `configure` script, this contains the ABI flags as specified by **PEP 3149** [https://peps.python.org/pep-3149/].

Added in version 3.2.

Changed in version 3.8: Default flags became an empty string (`m` flag for pymalloc has been removed).

Availability: Unix.

sys.addaudithook(*hook*)

Append the callable *hook* to the list of active auditing hooks for the current (sub)interpreter.

When an auditing event is raised through the **`sys.audit()`** function, each hook will be called in the order it was added with the event name and the tuple of arguments. Native hooks added by **`PySys_AddAuditHook()`** are called first, followed by hooks added in the current (sub)interpreter. Hooks can then log the event, raise an exception to abort the operation, or terminate the process entirely.

Note that audit hooks are primarily for collecting information about internal or otherwise unobservable actions, whether by Python or libraries written in Python. They are not suitable for implementing a “sandbox”. In particular, malicious code can trivially disable or bypass hooks added using this function. At a minimum, any security-sensitive hooks must be added using the C API `PySys_AddAuditHook()` before initialising the runtime, and any modules allowing arbitrary memory modification (such as `ctypes`) should be completely removed or closely monitored.

Calling `sys.addaudithook()` will itself raise an auditing event named `sys.addaudithook` with no arguments. If any existing hooks raise an exception derived from `RuntimeError`, the new hook will not be added and the exception suppressed. As a result, callers cannot assume that their hook has been added unless they control all existing hooks.

See the audit events table for all events raised by CPython, and **PEP 578** [https://peps.python.org/pep-0578/] for the original design discussion.

Added in version 3.8.

Changed in version 3.8.1: Exceptions derived from `Exception` but not `RuntimeError` are no longer suppressed.

CPython implementation detail: When tracing is enabled (see `settrace()`), Python hooks are only traced if the callable has a `__cantrace__` member that is set to a true value. Otherwise, trace functions will skip the hook.

sys.argv

The list of command line arguments passed to a Python script. `argv[0]` is the script name (it is operating system dependent whether this is a full pathname or not). If the command was executed using the `-c` command line option to the interpreter, `argv[0]` is set to the string `'-c'`. If no script name was passed to the Python interpreter, `argv[0]` is the empty string.

To loop over the standard input, or the list of files given on the command line, see the **`fileinput`** module.

See also **`sys.orig_argv`**.

Note

On Unix, command line arguments are passed by bytes from OS. Python decodes them with filesystem encoding and “surrogateescape” error handler. When you need original bytes, you can get it by `[os.fsencode(arg) for arg in sys.argv]`.

sys.audit(*event*, **args*)

Raise an auditing event and trigger any active auditing hooks. *event* is a string identifying the event, and *args* may contain optional arguments with more information about the event. The number and types of arguments for a given event are considered a public and stable API and should not be modified between releases.

For example, one auditing event is named `os.chdir`. This event has one argument called *path* that will contain the requested new working directory.

`sys.audit()` will call the existing auditing hooks, passing the event name and arguments, and will re-raise the first exception from any hook. In general, if an exception is raised, it should not be handled and the process should be terminated as quickly as possible. This allows hook implementations to decide how to respond to particular events: they can merely log the event or abort the operation by raising an exception.

Hooks are added using the `sys.addaudithook()` or `PySys_AddAuditHook()` functions.

The native equivalent of this function is `PySys_Audit()`. Using the native function is preferred when possible.

See the audit events table for all events raised by CPython.

Added in version 3.8.

sys.base_exec_prefix

Equivalent to `exec_prefix`, but referring to the base Python installation.

When running under Virtual Environments, `exec_prefix` gets overwritten to the virtual environment prefix. `base_exec_prefix`, conversely, does not change, and always points to the base Python installation. Refer to Virtual Environments for more information.

Added in version 3.3.

sys.base_prefix

Equivalent to `prefix`, but referring to the base Python installation.

When running under virtual environment, `prefix` gets overwritten to the virtual environment prefix. `base_prefix`, conversely, does not change, and always points to the base Python installation. Refer to Virtual Environments for more information.

Added in version 3.3.

sys.byteorder

An indicator of the native byte order. This will have the value `'big'` on big-endian (most-significant byte first) platforms, and `'little'` on little-endian (least-significant byte first) platforms.

sys.builtin_module_names

A tuple of strings containing the names of all modules that are compiled into this Python interpreter. (This information is not available in any other way — `modules.keys()` only lists the imported modules.)

See also the **`sys.stdlib_module_names`** list.

sys.call_tracing(*func*, *args*)

Call `func(*args)`, while tracing is enabled. The tracing state is saved, and restored afterwards. This is intended to be called from a debugger from a checkpoint, to recursively debug or profile some other code.

Tracing is suspended while calling a tracing function set by **`settrace()`** or **`setprofile()`** to avoid infinite recursion. **`call_tracing()`** enables explicit recursion of the tracing function.

sys.copyright

A string containing the copyright pertaining to the Python interpreter.

sys._clear_type_cache()

Clear the internal type cache. The type cache is used to speed up attribute and method lookups. Use the function *only* to drop unnecessary references during reference leak debugging.

This function should be used for internal and specialized purposes only.

Deprecated since version 3.13: Use the more general **`_clear_internal_caches()`** function instead.

sys._clear_internal_caches()

Clear all internal performance-related caches. Use this function *only* to release unnecessary references and memory blocks when hunting for leaks.

Added in version 3.13.

sys._current_frames()

Return a dictionary mapping each thread’s identifier to the topmost stack frame currently active in that thread at the time the function is called. Note that functions in the **`traceback`** module can build the call stack given such a frame.

This is most useful for debugging deadlock: this function does not require the deadlocked threads’ cooperation, and such threads’ call stacks are frozen for as long as they remain deadlocked. The frame returned for a non-deadlocked thread may bear no relationship to that thread’s current activity by the time calling code examines the frame.

This function should be used for internal and specialized purposes only.

Raises an auditing event `sys._current_frames` with no arguments.

sys._current_exceptions()

Return a dictionary mapping each thread’s identifier to the topmost exception currently active in that thread at the time the function is called. If a thread is not currently handling an exception, it is not included in the result dictionary.

This is most useful for statistical profiling.

This function should be used for internal and specialized purposes only.

Raises an auditing event `sys._current_exceptions` with no arguments.

Changed in version 3.12: Each value in the dictionary is now a single exception instance, rather than a 3-tuple as returned from `sys.exc_info()`.

sys.breakpointhook()

This hook function is called by built-in **`breakpoint()`**. By default, it drops you into the **`pdb`** debugger, but it can be set to any other function so that you can choose which debugger gets used.

The signature of this function is dependent on what it calls. For example, the default binding (e.g. `pdb.set_trace()`) expects no arguments, but you might bind it to a function that expects additional arguments (positional and/or keyword). The built-in `breakpoint()` function passes its `*args` and `**kws` straight through. Whatever `breakpointhooks()` returns is returned from `breakpoint()`.

The default implementation first consults the environment variable **`PYTHONBREAKPOINT`**. If that is set to `"0"` then this function returns immediately; i.e. it is a no-op. If the environment variable is not set, or is set to the empty string, `pdb.set_trace()` is called. Otherwise this variable should name a function to run, using Python’s dotted-import nomenclature, e.g. `package.subpackage.module.function`. In this case, `package.subpackage.module` would be imported and the resulting module must have a callable named `function()`. This is run, passing in `*args` and `**kws`, and whatever `function()` returns, `sys.breakpointhook()` returns to the built-in **`breakpoint()`** function.

Note that if anything goes wrong while importing the callable named by **`PYTHONBREAKPOINT`**, a **`RuntimeWarning`** is reported and the breakpoint is ignored.

Also note that if `sys.breakpointhook()` is overridden programmatically, **`PYTHONBREAKPOINT`** is *not* consulted.

Added in version 3.7.

sys._debugmallocstats()

Print low-level information to stderr about the state of CPython’s memory allocator.

If Python is built in debug mode (**`configure --with-pydebug option`**), it also performs some expensive internal consistency checks.

Added in version 3.3.

CPython implementation detail: This function is specific to CPython. The exact output format is not defined here, and may change.

sys.dllhandle

Integer specifying the handle of the Python DLL.

Availability: Windows.

sys.displayhook(*value*)

If *value* is not `None`, this function prints `repr(value)` to `sys.stdout`, and saves *value* in `builtins._`. If `repr(value)` is not encodable to `sys.stdout.encoding` with `sys.stdout.errors` error handler (which is probably `'strict'`), encode it to `sys.stdout.encoding` with `'backslashreplace'` error handler.

`sys.displayhook` is called on the result of evaluating an expression entered in an interactive Python session. The display of these values can be customized by assigning another one-argument function to `sys.displayhook`.

Pseudo-code:

```
def displayhook(value):
    if value is None:
        return
    # Set '_' to None to avoid recursion
    builtins._ = None
    text = repr(value)
    try:
        sys.stdout.write(text)
    except UnicodeEncodeError:
        bytes = text.encode(sys.stdout.encoding,
'backslashreplace')
        if hasattr(sys.stdout, 'buffer'):
            sys.stdout.buffer.write(bytes)
        else:
            text = bytes.decode(sys.stdout.encoding, 'strict')
            sys.stdout.write(text)
    sys.stdout.write("\n")
    builtins._ = value
```

Changed in version 3.2: Use `'backslashreplace'` error handler on **`UnicodeEncodeError`**.

sys.dont_write_bytecode

If this is true, Python won't try to write `.pyc` files on the import of source modules. This value is initially set to `True` or `False` depending on the **`-B`** command line option and the **`PYTHONDONTWRITEBYTECODE`** environment variable, but you can set it yourself to control bytecode file generation.

sys._emscripten_info

A named tuple holding information about the environment on the *wasm32-emscripten* platform. The named tuple is provisional and may change in the future.

_emscripten_info.emscripten_version

Emscripten version as tuple of ints (major, minor, micro), e.g. `(3, 1, 8)`.

_emscripten_info.runtime

Runtime string, e.g. browser user agent, `'Node.js v14.18.2'`, or `'UNKNOWN'`.

_emscripten_info.pthreads

`True` if Python is compiled with Emscripten pthreads support.

_emscripten_info.shared_memory

`True` if Python is compiled with shared memory support.

Availability: Emscripten.

Added in version 3.11.

sys.pycache_prefix

If this is set (not `None`), Python will write bytecode-cache `.pyc` files to (and read them from) a parallel directory tree rooted at this directory, rather than from `__pycache__` directories in the source code tree. Any `__pycache__` directories in the source code tree will be ignored and new `.pyc` files written within the pycache prefix. Thus if you use **`compileall`** as a pre-build step, you must ensure you run it with the same pycache prefix (if any) that you will use at runtime.

A relative path is interpreted relative to the current working directory.

This value is initially set based on the value of the **`-X`** `pycache_prefix=PATH` command-line option or the **`PYTHONPYCACHEPREFIX`** environment variable (command-line takes precedence). If neither are set, it is `None`.

Added in version 3.8.

sys.excepthook(*type*, *value*, *traceback*)

This function prints out a given traceback and exception to `sys.stderr`.

When an exception other than **`SystemExit`** is raised and uncaught, the interpreter calls `sys.excepthook` with three arguments, the exception class, exception instance, and a traceback object. In an interactive session this happens just before control is returned to the prompt; in a Python program this happens just before the program exits. The handling of such top-level exceptions can be customized by assigning another three-argument function to `sys.excepthook`.

Raise an auditing event `sys.excepthook` with arguments `hook`, `type`, `value`, `traceback` when an uncaught exception occurs. If no hook has been set, `hook` may be `None`. If any hook raises an exception derived from **`RuntimeError`** the call to the hook will be suppressed. Otherwise, the audit hook exception will be reported as unraisable and `sys.excepthook` will be called.

See also

The **`sys.unraisablehook()`** function handles unraisable exceptions and the **`threading.excepthook()`** function handles exception raised by **`threading.Thread.run()`**.

sys.__breakpointhook__
sys.__displayhook__
sys.__excepthook__
sys.__unraisablehook__

These objects contain the original values of `breakpointhook`, `displayhook`, `excepthook`, and `unraisablehook` at the start of the program. They are saved so that `breakpointhook`, `displayhook` and `excepthook`, `unraisablehook` can be restored in case they happen to get replaced with broken or alternative objects.

Added in version 3.7: __breakpointhook__

Added in version 3.8: __unraisablehook__

sys.exception()

This function, when called while an exception handler is executing (such as an `except` or `except*` clause), returns the exception instance that was caught by this handler. When exception handlers are nested within one another, only the exception handled by the innermost handler is accessible.

If no exception handler is executing, this function returns `None`.

Added in version 3.11.

sys.exc_info()

This function returns the old-style representation of the handled exception. If an exception `e` is currently handled (so **`exception()`** would return `e`), **`exc_info()`** returns the tuple `(type(e), e, e.__traceback__)`. That is, a tuple containing the type of the exception (a subclass of **`BaseException`**), the exception itself, and a traceback object which typically encapsulates the call stack at the point where the exception last occurred.

If no exception is being handled anywhere on the stack, this function return a tuple containing three `None` values.

Changed in version 3.11: The `type` and `traceback` fields are now derived from the `value` (the exception instance), so when an exception is modified while it is being handled, the changes are reflected in the results of subsequent calls to **`exc_info()`**.

sys.exec_prefix

A string giving the site-specific directory prefix where the platform-dependent Python files are installed; by default, this is also `'/usr/local'`. This can be set at build time with the `--exec-prefix` argument to the **configure** script. Specifically, all configuration files (e.g. the `pyconfig.h` header file) are installed in the directory `exec_prefix/lib/pythonX.Y/config`, and shared library modules are installed in `exec_prefix/lib/pythonX.Y/lib-dynload`, where *X.Y* is the version number of Python, for example `3.2`.

Note

If a virtual environment is in effect, this **`exec_prefix`** will point to the virtual environment. The value for the Python installation will still be available, via **`base_exec_prefix`**. Refer to Virtual Environments for more information.

Changed in version 3.14: When running under a virtual environment, **`prefix`** and **`exec_prefix`** are now set to the virtual environment prefix by the path initialization, instead of **`site`**. This means that **`prefix`** and **`exec_prefix`** always point to the virtual environment, even when **`site`** is disabled (**`-S`**).

sys.executable

A string giving the absolute path of the executable binary for the Python interpreter, on systems where this makes sense. If Python is unable to retrieve the real path to its executable, **`sys.executable`** will be an empty string or `None`.

sys.exit([*arg*])

Raise a **`SystemExit`** exception, signaling an intention to exit the interpreter.

The optional argument *arg* can be an integer giving the exit status (defaulting to zero), or another type of object. If it is an integer, zero is considered “successful termination” and any nonzero value is considered “abnormal termination” by shells and the like. Most systems require it to be in the range 0–127, and produce undefined results otherwise. Some systems have a convention for assigning specific meanings to specific exit codes, but these are generally underdeveloped; Unix programs generally use 2 for command line syntax errors and 1 for all other kind of errors. If another type of object is passed, `None` is equivalent to passing zero, and any other object is printed to **`stderr`** and results in an exit code of 1. In particular, `sys.exit("some error message")` is a quick way to exit a program when an error occurs.

Since **`exit()`** ultimately “only” raises an exception, it will only exit the process when called from the main thread, and the exception is not intercepted. Cleanup actions specified by finally clauses of **`try`** statements are honored, and it is possible to intercept the exit attempt at an outer level.

Changed in version 3.6: If an error occurs in the cleanup after the Python interpreter has caught **`SystemExit`** (such as an error flushing buffered data in the standard streams), the exit status is changed to 120.

sys.flags

The named tuple *flags* exposes the status of command line flags. Flags should only be accessed only by name and not by index. The attributes are read only.

flags.debug	**`-d`**
flags.inspect	**`-i`**
flags.interactive	**`-i`**

flags.isolated	`-I`
flags.optimize	`-O` or `-OO`
flags.dont_write_bytecode	`-B`
flags.no_user_site	`-s`
flags.no_site	`-S`
flags.ignore_environment	`-E`
flags.verbose	`-v`
flags.bytes_warning	`-b`
flags.quiet	`-q`
flags.hash_randomization	`-R`
flags.dev_mode	`-X dev` (Python Development Mode)
flags.utf8_mode	`-X utf8`
flags.safe_path	`-P`
flags.int_max_str_digits	`-X int_max_str_digits` (integer string conversion length limitation)
flags.warn_default_encoding	`-X warn_default_encoding`
flags.gil	`-X gil` and `PYTHON_GIL`
flags.thread_inherit_context	`-X thread_inherit_context` and

	`PYTHON_THREAD_INHERIT_CONTEXT`
flags.context_aware_warnings	`-X context_aware_warnings` and `PYTHON_CONTEXT_AWARE_WARNINGS`

Changed in version 3.2: Added `quiet` attribute for the new `-q` flag.

Added in version 3.2.3: The `hash_randomization` attribute.

Changed in version 3.3: Removed obsolete `division_warning` attribute.

Changed in version 3.4: Added `isolated` attribute for `-I` `isolated` flag.

Changed in version 3.7: Added the `dev_mode` attribute for the new Python Development Mode and the `utf8_mode` attribute for the new `-X` `utf8` flag.

Changed in version 3.10: Added `warn_default_encoding` attribute for `-X` `warn_default_encoding` flag.

Changed in version 3.11: Added the `safe_path` attribute for `-P` option.

Changed in version 3.11: Added the `int_max_str_digits` attribute.

Changed in version 3.13: Added the `gil` attribute.

Changed in version 3.14: Added the `thread_inherit_context` attribute.

Changed in version 3.14: Added the `context_aware_warnings` attribute.

sys.float_info

A named tuple holding information about the float type. It contains low level information about the precision and internal representation. The values correspond to the various floating-point constants defined in the standard header file `float.h` for the ‘C’ programming language; see section 5.2.4.2.2 of the 1999 ISO/IEC C standard [C99], ‘Characteristics of floating types’, for details.

Attributes of the **`float_info`** named tuple

attribute	**float.h macro**	**explanation**
float_info.epsilon	**`DBL_EPSILON`**	difference between 1.0 and the least value greater than 1.0 that is representable as a float. See also **`math.ulp()`**.
float_info.dig	**`DBL_DIG`**	The maximum number of decimal digits that can be faithfully represented in a float; see below.
float_info.mant_dig	**`DBL_MANT_DIG`**	Float precision: the number of base-`radix` digits in the significand of a float.
float_info.max	**`DBL_MAX`**	The maximum representable

attribute	**float.h macro**	**explanation**
		positive finite float.
float_info.max_exp	**`DBL_MAX_EXP`**	The maximum integer *e* such that `radix**(e-1)` is a representable finite float.
float_info.max_10_exp	**`DBL_MAX_10_EXP`**	The maximum integer *e* such that `10**e` is in the range of representable finite floats.
float_info.min	**`DBL_MIN`**	The minimum representable positive *normalized* float. Use [`math.ulp(0.0)`] to get the smallest positive *denormalized* representable float.
float_info.min_exp	**`DBL_MIN_EXP`**	The minimum integer *e* such that `radix**(e-1)` is a normalized float.
float_info.min_10_exp	**`DBL_MIN_10_EXP`**	The minimum integer *e* such that `10**e` is a normalized float.

attribute	float.h macro	explanation
float_info.radix	**`FLT_RADIX`**	The radix of exponent representation.
float_info.rounds	**`FLT_ROUNDS`**	An integer representing the rounding mode for floating-point arithmetic. This reflects the value of the system **`FLT_ROUNDS`** macro at interpreter startup time: • `-1`: indeterminable • `0`: toward zero • `1`: to nearest • `2`: toward positive infinity • `3`: toward negative infinity All other values for **`FLT_ROUNDS`** characterize implementation-defined rounding behavior.

The attribute `sys.float_info.dig` needs further explanation. If `s` is any string representing a decimal number with at most **`sys.float_info.dig`** significant digits, then converting `s` to a float and back again will recover a string representing the same decimal value:

```
>>> import sys
>>> sys.float_info.dig
15
>>> s = '3.14159265358979'    # decimal string with 15
significant digits
>>> format(float(s), '.15g')  # convert to float and back ->
same value
'3.14159265358979'
```

But for strings with more than **`sys.float_info.dig`** significant digits, this isn't always true:

```
>>> s = '9876543211234567'    # 16 significant digits is too
many!
>>> format(float(s), '.16g')  # conversion changes value
'9876543211234568'
```

sys.float_repr_style

A string indicating how the **`repr()`** function behaves for floats. If the string has value `'short'` then for a finite float `x`, `repr(x)` aims to produce a short string with the property that `float(repr(x)) == x`. This is the usual behaviour in Python 3.1 and later. Otherwise, `float_repr_style` has value `'legacy'` and `repr(x)` behaves in the same way as it did in versions of Python prior to 3.1.

Added in version 3.1.

sys.getallocatedblocks()

Return the number of memory blocks currently allocated by the interpreter, regardless of their size. This function is mainly useful for tracking and debugging memory leaks. Because of the interpreter's internal caches, the result can vary from call to call; you may have to call `_clear_internal_caches()` and `gc.collect()` to get more predictable results.

If a Python build or implementation cannot reasonably compute this information, `getallocatedblocks()` is allowed to return 0 instead.

Added in version 3.4.

sys.getunicodeinternedsize()

Return the number of unicode objects that have been interned.

Added in version 3.12.

sys.getandroidapilevel()

Return the build-time API level of Android as an integer. This represents the minimum version of Android this build of Python can run on. For runtime version information, see `platform.android_ver()`.

Availability: Android.

Added in version 3.7.

sys.getdefaultencoding()

Return `'utf-8'`. This is the name of the default string encoding, used in methods like `str.encode()`.

sys.getdlopenflags()

Return the current value of the flags that are used for **`dlopen()`** calls. Symbolic names for the flag values can be found in the `os` module (`RTLD_`*xxx* constants, e.g. `os.RTLD_LAZY`).

Availability: Unix.

sys.getfilesystemencoding()

Get the filesystem encoding: the encoding used with the filesystem error handler to convert between Unicode filenames and bytes filenames. The filesystem error handler is returned from **`getfilesystemencodeerrors()`**.

For best compatibility, str should be used for filenames in all cases, although representing filenames as bytes is also supported. Functions accepting or returning filenames should support either str or bytes and internally convert to the system’s preferred representation.

`os.fsencode()` and **`os.fsdecode()`** should be used to ensure that the correct encoding and errors mode are used.

The filesystem encoding and error handler are configured at Python startup by the **`PyConfig_Read()`** function: see **`filesystem_encoding`** and **`filesystem_errors`** members of **`PyConfig`**.

Changed in version 3.2: **`getfilesystemencoding()`** result cannot be `None` anymore.

Changed in version 3.6: Windows is no longer guaranteed to return `'mbcs'`. See **PEP 529** [https://peps.python.org/pep-0529/] and **`_enablelegacywindowsfsencoding()`** for more information.

Changed in version 3.7: Return `'utf-8'` if the Python UTF-8 Mode is enabled.

sys.getfilesystemencodeerrors()

Get the filesystem error handler: the error handler used with the filesystem encoding to convert between Unicode filenames and bytes filenames. The filesystem encoding is returned from **`getfilesystemencoding()`**.

`os.fsencode()` and `os.fsdecode()` should be used to ensure that the correct encoding and errors mode are used.

The filesystem encoding and error handler are configured at Python startup by the `PyConfig_Read()` function: see `filesystem_encoding` and `filesystem_errors` members of `PyConfig`.

Added in version 3.6.

sys.get_int_max_str_digits()

Returns the current value for the integer string conversion length limitation. See also `set_int_max_str_digits()`.

Added in version 3.11.

sys.getrefcount(*object*)

Return the reference count of the *object*. The count returned is generally one higher than you might expect, because it includes the (temporary) reference as an argument to `getrefcount()`.

Note that the returned value may not actually reflect how many references to the object are actually held. For example, some objects are immortal and have a very high refcount that does not reflect the actual number of references. Consequently, do not rely on the returned value to be accurate, other than a value of 0 or 1.

CPython implementation detail: Immortal objects with a large reference count can be identified via `_is_immortal()`.

Changed in version 3.12: Immortal objects have very large refcounts that do not match the actual number of references to the object.

sys.getrecursionlimit()

Return the current value of the recursion limit, the maximum depth of the Python interpreter stack. This limit prevents infinite recursion from causing an overflow of the C stack and crashing Python. It can be set by **`setrecursionlimit()`**.

sys.getsizeof(*object*[, *default*])

Return the size of an object in bytes. The object can be any type of object. All built-in objects will return correct results, but this does not have to hold true for third-party extensions as it is implementation specific.

Only the memory consumption directly attributed to the object is accounted for, not the memory consumption of objects it refers to.

If given, *default* will be returned if the object does not provide means to retrieve the size. Otherwise a **`TypeError`** will be raised.

`getsizeof()` calls the object’s `__sizeof__` method and adds an additional garbage collector overhead if the object is managed by the garbage collector.

See recursive sizeof recipe [https://code.activestate.com/recipes/577504-compute-memory-footprint-of-an-object-and-its-cont/] for an example of using **`getsizeof()`** recursively to find the size of containers and all their contents.

sys.getswitchinterval()

Return the interpreter’s “thread switch interval” in seconds; see **`setswitchinterval()`**.

Added in version 3.2.

sys._getframe([*depth*])

Return a frame object from the call stack. If optional integer *depth* is given, return the frame object that many calls below the top of the stack. If that is deeper than the call stack, `ValueError` is raised. The default for *depth* is zero, returning the frame at the top of the call stack.

Raises an auditing event `sys._getframe` with argument `frame`.

CPython implementation detail: This function should be used for internal and specialized purposes only. It is not guaranteed to exist in all implementations of Python.

sys._getframemodulename([*depth*])

Return the name of a module from the call stack. If optional integer *depth* is given, return the module that many calls below the top of the stack. If that is deeper than the call stack, or if the module is unidentifiable, `None` is returned. The default for *depth* is zero, returning the module at the top of the call stack.

Raises an auditing event `sys._getframemodulename` with argument `depth`.

CPython implementation detail: This function should be used for internal and specialized purposes only. It is not guaranteed to exist in all implementations of Python.

Added in version 3.12.

sys.getobjects(*limit*[, *type*])

This function only exists if CPython was built using the specialized configure option `--with-trace-refs`. It is intended only for debugging garbage-collection issues.

Return a list of up to *limit* dynamically allocated Python objects. If *type* is given, only objects of that exact type (not subtypes) are included.

Objects from the list are not safe to use. Specifically, the result will include objects from all interpreters that share their object allocator state (that is, ones created with `PyInterpreterConfig.use_main_obmalloc` set to 1 or using `Py_NewInterpreter()`, and the main interpreter). Mixing objects from different interpreters may lead to crashes or other unexpected behavior.

CPython implementation detail: This function should be used for specialized purposes only. It is not guaranteed to exist in all implementations of Python.

Changed in version 3.14: The result may include objects from other interpreters.

sys.getprofile()

Get the profiler function as set by `setprofile()`.

sys.gettrace()

Get the trace function as set by `settrace()`.

CPython implementation detail: The `gettrace()` function is intended only for implementing debuggers, profilers, coverage tools and the like. Its behavior is part of the implementation platform, rather than part of the language definition, and thus may not be available in all Python implementations.

sys.getwindowsversion()

Return a named tuple describing the Windows version currently running. The named elements are *major*, *minor*, *build*, *platform*, *service_pack*, *service_pack_minor*, *service_pack_major*, *suite_mask*, *product_type* and *platform_version*. *service_pack* contains a string, *platform_version* a 3-tuple and all other values are integers. The components can also be accessed by name, so `sys.getwindowsversion()[0]` is equivalent to `sys.getwindowsversion().major`. For compatibility with prior versions, only the first 5 elements are retrievable by indexing.

platform will be `2` (VER_PLATFORM_WIN32_NT).

product_type may be one of the following values:

Constant	Meaning
`1` (VER_NT_WORKSTATION)	The system is a workstation.
`2` (VER_NT_DOMAIN_CONTROLLER)	The system is a domain controller.
`3` (VER_NT_SERVER)	The system is a server, but not a domain controller.

This function wraps the Win32 **`GetVersionEx()`** function; see the Microsoft documentation on **`OSVERSIONINFOEX()`** for more information about these fields.

platform_version returns the major version, minor version and build number of the current operating system, rather than the version that is being emulated for the process. It is intended for use in logging rather than for feature detection.

Note

platform_version derives the version from kernel32.dll which can be of a different version than the OS version. Please use **`platform`** module for achieving accurate OS version.

Availability: Windows.

Changed in version 3.2: Changed to a named tuple and added *service_pack_minor*, *service_pack_major*, *suite_mask*, and *product_type*.

Changed in version 3.6: Added *platform_version*

sys.get_asyncgen_hooks()

Returns an *asyncgen_hooks* object, which is similar to a **`namedtuple`** of the form `(firstiter, finalizer)`, where *firstiter* and *finalizer* are expected to be either `None` or functions which take an asynchronous generator iterator as an argument, and are used to schedule finalization of an asynchronous generator by an event loop.

Added in version 3.6: See **PEP 525** [https://peps.python.org/pep-0525/] for more details.

Note

This function has been added on a provisional basis (see **PEP 411** [https://peps.python.org/pep-0411/] for details.)

sys.get_coroutine_origin_tracking_depth()

Get the current coroutine origin tracking depth, as set by **`set_coroutine_origin_tracking_depth()`**.

Added in version 3.7.

Note

This function has been added on a provisional basis (see **PEP 411** [https://peps.python.org/pep-0411/] for details.) Use it only for debugging purposes.

sys.hash_info

A named tuple giving parameters of the numeric hash implementation. For more details about hashing of numeric types, see Hashing of numeric types.

hash_info.width

The width in bits used for hash values

hash_info.modulus

The prime modulus P used for numeric hash scheme

hash_info.inf

The hash value returned for a positive infinity

hash_info.nan

(This attribute is no longer used)

hash_info.imag

The multiplier used for the imaginary part of a complex number

hash_info.algorithm

The name of the algorithm for hashing of str, bytes, and memoryview

hash_info.hash_bits

The internal output size of the hash algorithm

hash_info.seed_bits

The size of the seed key of the hash algorithm

hash_info.cutoff

Cutoff for small string DJBX33A optimization in range `[1, cutoff)`.

Added in version 3.2.

Changed in version 3.4: Added *algorithm*, *hash_bits*, *seed_bits*, and *cutoff*.

sys.hexversion

The version number encoded as a single integer. This is guaranteed to increase with each version, including proper support for non-production releases. For example, to test that the Python interpreter is at least version 1.5.2, use:

```
if sys.hexversion >= 0x010502F0:
    # use some advanced feature
    ...
else:
    # use an alternative implementation or warn the user
    ...
```

This is called `hexversion` since it only really looks meaningful when viewed as the result of passing it to the built-in **`hex()`** function. The named tuple **`sys.version_info`** may be used for a more human-friendly encoding of the same information.

More details of `hexversion` can be found at API and ABI Versioning.

sys.implementation

An object containing information about the implementation of the currently running Python interpreter. The following attributes are required to exist in all Python implementations.

name is the implementation’s identifier, e.g. `'cpython'`. The actual string is defined by the Python implementation, but it is guaranteed to be lower case.

version is a named tuple, in the same format as **`sys.version_info`**. It represents the version of the Python *implementation*. This has a distinct meaning from the specific version of the Python *language* to which the currently running interpreter conforms, which `sys.version_info` represents. For example, for PyPy 1.8 `sys.implementation.version` might be `sys.version_info(1, 8, 0, 'final', 0)`, whereas `sys.version_info` would be `sys.version_info(2, 7, 2, 'final', 0)`. For CPython they are the same value, since it is the reference implementation.

hexversion is the implementation version in hexadecimal format, like **`sys.hexversion`**.

cache_tag is the tag used by the import machinery in the filenames of cached modules. By convention, it would be a composite of the implementation's name and version, like `'cpython-33'`. However, a Python implementation may use some other value if appropriate. If `cache_tag` is set to `None`, it indicates that module caching should be disabled.

supports_isolated_interpreters is a boolean value, whether this implementation supports multiple isolated interpreters. It is `True` for CPython on most platforms. Platforms with this support implement the low-level **`_interpreters`** module.

See also

PEP 684 [https://peps.python.org/pep-0684/], **PEP 734** [https://peps.python.org/pep-0734/], and **`concurrent.interpreters`**.

`sys.implementation` may contain additional attributes specific to the Python implementation. These non-standard attributes must start with an underscore, and are not described here. Regardless of its contents, **`sys.implementation`** will not change during a run of the interpreter, nor between implementation versions. (It may change between Python language versions, however.) See **PEP 421** [https://peps.python.org/pep-0421/] for more information.

Added in version 3.3.

Changed in version 3.14: Added `supports_isolated_interpreters` field.

Note

The addition of new required attributes must go through the normal PEP process. See **PEP 421** [https://peps.python.org/pep-0421/] for more information.

sys.int_info

A named tuple that holds information about Python’s internal representation of integers. The attributes are read only.

int_info.bits_per_digit

The number of bits held in each digit. Python integers are stored internally in base `2**int_info.bits_per_digit`.

int_info.sizeof_digit

The size in bytes of the C type used to represent a digit.

int_info.default_max_str_digits

The default value for **`sys.get_int_max_str_digits()`** when it is not otherwise explicitly configured.

int_info.str_digits_check_threshold

The minimum non-zero value for **`sys.set_int_max_str_digits()`**, **`PYTHONINTMAXSTRDIGITS`**, or **`-X int_max_str_digits`**.

Added in version 3.1.

Changed in version 3.11: Added **`default_max_str_digits`** and **`str_digits_check_threshold`**.

sys.__interactivehook__

When this attribute exists, its value is automatically called (with no arguments) when the interpreter is launched in interactive mode. This is done after the **`PYTHONSTARTUP`** file is read, so that you can set this hook there. The **`site`** module sets this.

Raises an auditing event `cpython.run_interactivehook` with the hook object as the argument when the hook is called on startup.

Added in version 3.4.

sys.intern(*string*)

Enter *string* in the table of “interned” strings and return the interned string – which is *string* itself or a copy. Interning strings is useful to gain a little performance on dictionary lookup – if the keys in a dictionary are interned, and the lookup key is interned, the key comparisons (after hashing) can be done by a pointer compare instead of a string compare. Normally, the names used in Python programs are automatically interned, and the dictionaries used to hold module, class or instance attributes have interned keys.

Interned strings are not immortal; you must keep a reference to the return value of **`intern()`** around to benefit from it.

sys._is_gil_enabled()

Return **`True`** if the GIL is enabled and **`False`** if it is disabled.

Added in version 3.13.

CPython implementation detail: It is not guaranteed to exist in all implementations of Python.

sys.is_finalizing()

Return **`True`** if the main Python interpreter is shutting down. Return **`False`** otherwise.

See also the **`PythonFinalizationError`** exception.

Added in version 3.5.

sys._jit

Utilities for observing just-in-time compilation.

CPython implementation detail: JIT compilation is an *experimental implementation detail* of CPython. `sys._jit` is not guaranteed to exist or behave the same way in all Python implementations, versions, or build configurations.

Added in version 3.14.

_jit.is_available()

Return `True` if the current Python executable supports JIT compilation, and `False` otherwise. This can be controlled by building CPython with the `--experimental-jit` option on Windows, and the **`--enable-experimental-jit`** option on all other platforms.

_jit.is_enabled()

Return `True` if JIT compilation is enabled for the current Python process (implies **`sys._jit.is_available()`**), and `False` otherwise. If JIT compilation is available, this can be controlled by setting the **`PYTHON_JIT`** environment variable to `0` (disabled) or `1` (enabled) at interpreter startup.

_jit.is_active()

Return `True` if the topmost Python frame is currently executing JIT code (implies **`sys._jit.is_enabled()`**), and `False` otherwise.

Note

This function is intended for testing and debugging the JIT itself. It should be avoided for any other purpose.

Note

Due to the nature of tracing JIT compilers, repeated calls to this function may give surprising results. For example, branching on its return value will likely lead to unexpected behavior (if doing so causes JIT code to be entered or exited):

```
>>> for warmup in range(BIG_NUMBER):
...     # This line is "hot", and is eventually JIT-
compiled:
...     if sys._jit.is_active():
...         # This line is "cold", and is run in the
interpreter:
...         assert sys._jit.is_active()
...
Traceback (most recent call last):
  File "<stdin>", line 5, in <module>
    assert sys._jit.is_active()
           ~~~~~~~~~~~~~~~~~~^^
AssertionError
```

sys.last_exc

This variable is not always defined; it is set to the exception instance when an exception is not handled and the interpreter prints an error message and a stack traceback. Its intended use is to allow an interactive user to import a debugger module and engage in post-mortem debugging without having to re-execute the command that caused the error. (Typical use is `import pdb; pdb.pm()` to enter the post-mortem debugger; see `pdb` module for more information.)

Added in version 3.12.

sys._is_immortal(*op*)

Return `True` if the given object is immortal, `False` otherwise.

Note

Objects that are immortal (and thus return `True` upon being passed to this function) are not guaranteed to be immortal in future versions, and vice versa for mortal objects.

Added in version 3.14.

CPython implementation detail: This function should be used for specialized purposes only. It is not guaranteed to exist in all implementations of Python.

sys._is_interned(*string*)

Return `True` if the given string is “interned”, `False` otherwise.

Added in version 3.13.

CPython implementation detail: It is not guaranteed to exist in all implementations of Python.

sys.last_type
sys.last_value
sys.last_traceback

These three variables are deprecated; use `sys.last_exc` instead. They hold the legacy representation of `sys.last_exc`, as returned from `exc_info()` above.

sys.maxsize

An integer giving the maximum value a variable of type `Py_ssize_t` can take. It’s usually `2**31 - 1` on a 32-bit platform and `2**63 - 1` on a 64-bit platform.

sys.maxunicode

An integer giving the value of the largest Unicode code point, i.e. `1114111` (`0x10FFFF` in hexadecimal).

Changed in version 3.3: Before **PEP 393** [https://peps.python.org/pep-0393/], `sys.maxunicode` used to be either `0xFFFF` or `0x10FFFF`, depending on the configuration option that specified whether Unicode characters were stored as UCS-2 or UCS-4.

sys.meta_path

A list of meta path finder objects that have their **`find_spec()`** methods called to see if one of the objects can find the module to be imported. By default, it holds entries that implement Python's default import semantics. The **`find_spec()`** method is called with at least the absolute name of the module being imported. If the module to be imported is contained in a package, then the parent package's **`__path__`** attribute is passed in as a second argument. The method returns a module spec, or `None` if the module cannot be found.

See also

`importlib.abc.MetaPathFinder`
The abstract base class defining the interface of finder objects on **`meta_path`**.

`importlib.machinery.ModuleSpec`
The concrete class which **`find_spec()`** should return instances of.

Changed in version 3.4: Module specs were introduced in Python 3.4, by **PEP 451** [https://peps.python.org/pep-0451/].

Changed in version 3.12: Removed the fallback that looked for a **`find_module()`** method if a **`meta_path`** entry didn't have a **`find_spec()`** method.

sys.modules

This is a dictionary that maps module names to modules which have already been loaded. This can be manipulated to force reloading of modules and other tricks. However, replacing the dictionary will not necessarily work as expected and deleting essential items from the dictionary may cause Python to fail. If you want to iterate over this global dictionary always use `sys.modules.copy()` or `tuple(sys.modules)` to avoid exceptions as its size may change during iteration as a side effect of code or activity in other threads.

sys.orig_argv

The list of the original command line arguments passed to the Python executable.

The elements of **`sys.orig_argv`** are the arguments to the Python interpreter, while the elements of **`sys.argv`** are the arguments to the user's program. Arguments consumed by the interpreter itself will be present in **`sys.orig_argv`** and missing from **`sys.argv`**.

Added in version 3.10.

sys.path

A list of strings that specifies the search path for modules. Initialized from the environment variable **`PYTHONPATH`**, plus an installation-dependent default.

By default, as initialized upon program startup, a potentially unsafe path is prepended to **`sys.path`** (*before* the entries inserted as a result of **`PYTHONPATH`**):

- `python -m module` command line: prepend the current working directory.
- `python script.py` command line: prepend the script's directory. If it's a symbolic link, resolve symbolic links.

- `python -c code` and `python` (REPL) command lines: prepend an empty string, which means the current working directory.

To not prepend this potentially unsafe path, use the **`-P`** command line option or the **`PYTHONSAFEPATH`** environment variable.

A program is free to modify this list for its own purposes. Only strings should be added to **`sys.path`**; all other data types are ignored during import.

See also

- Module **`site`** This describes how to use .pth files to extend **`sys.path`**.

sys.path_hooks

A list of callables that take a path argument to try to create a finder for the path. If a finder can be created, it is to be returned by the callable, else raise **`ImportError`**.

Originally specified in **PEP 302** [https://peps.python.org/pep-0302/].

sys.path_importer_cache

A dictionary acting as a cache for finder objects. The keys are paths that have been passed to **`sys.path_hooks`** and the values are the finders that are found. If a path is a valid file system path but no finder is found on **`sys.path_hooks`** then `None` is stored.

Originally specified in **PEP 302** [https://peps.python.org/pep-0302/].

sys.platform

A string containing a platform identifier. Known values are:

System	**`platform` value**
AIX	`'aix'`
Android	`'android'`
Emscripten	`'emscripten'`
FreeBSD	`'freebsd'`
iOS	`'ios'`
Linux	`'linux'`
macOS	`'darwin'`
Windows	`'win32'`
Windows/Cygwin	`'cygwin'`
WASI	`'wasi'`

On Unix systems not listed in the table, the value is the lowercased OS name as returned by `uname -s`, with the first part of the version as returned by `uname -r` appended, e.g. `'sunos5'`, *at the time when Python was built*. Unless you want to test for a specific system version, it is therefore recommended to use the following idiom:

```
if sys.platform.startswith('sunos'):
    # SunOS-specific code here...
```

Changed in version 3.3: On Linux, `sys.platform` doesn't contain the major version anymore. It is always `'linux'`, instead of `'linux2'` or `'linux3'`.

Changed in version 3.8: On AIX, `sys.platform` doesn't contain the major version anymore. It is always `'aix'`,

instead of `'aix5'` or `'aix7'`.
Changed in version 3.13: On Android, **`sys.platform`** now returns `'android'` rather than `'linux'`.

Changed in version 3.14: On FreeBSD, **`sys.platform`** doesn’t contain the major version anymore. It is always `'freebsd'`, instead of `'freebsd13'` or `'freebsd14'`.

See also

`os.name` has a coarser granularity. **`os.uname()`** gives system-dependent version information.

The **`platform`** module provides detailed checks for the system’s identity.

sys.platlibdir

Name of the platform-specific library directory. It is used to build the path of standard library and the paths of installed extension modules.

It is equal to `"lib"` on most platforms. On Fedora and SuSE, it is equal to `"lib64"` on 64-bit platforms which gives the following `sys.path` paths (where `X.Y` is the Python `major.minor` version):

- `/usr/lib64/pythonX.Y/`: Standard library (like `os.py` of the **`os`** module)
- `/usr/lib64/pythonX.Y/lib-dynload/`: C extension modules of the standard library (like the **`errno`** module, the exact filename is platform specific)
- `/usr/lib/pythonX.Y/site-packages/` (always use `lib`, not **`sys.platlibdir`**): Third-party modules
- `/usr/lib64/pythonX.Y/site-packages/`: C extension modules of third-party packages

Added in version 3.9.

sys.prefix

A string giving the site-specific directory prefix where the platform independent Python files are installed; on Unix, the default is `/usr/local`. This can be set at build time with the **`--prefix`** argument to the **configure** script. See Installation paths for derived paths.

> **Note**
>
> If a virtual environment is in effect, this **`prefix`** will point to the virtual environment. The value for the Python installation will still be available, via **`base_prefix`**. Refer to Virtual Environments for more information.

Changed in version 3.14: When running under a virtual environment, **`prefix`** and **`exec_prefix`** are now set to the virtual environment prefix by the path initialization, instead of **`site`**. This means that **`prefix`** and **`exec_prefix`** always point to the virtual environment, even when **`site`** is disabled (**`-S`**).

sys.ps1
sys.ps2

Strings specifying the primary and secondary prompt of the interpreter. These are only defined if the interpreter is in interactive mode. Their initial values in this case are `'>>> '` and `'... '`. If a non-string object is assigned to either variable, its **`str()`** is re-evaluated each time the interpreter prepares to read a new interactive command; this can be used to implement a dynamic prompt.

sys.setdlopenflags(*n*)

Set the flags used by the interpreter for **`dlopen()`** calls, such as when the interpreter loads extension modules. Among other things, this will enable a lazy resolving of symbols when importing a module, if called as `sys.setdlopenflags(0)`. To share symbols across extension modules, call as `sys.setdlopenflags(os.RTLD_GLOBAL)`. Symbolic names for the flag values can be found in the `os` module (`RTLD_xxx` constants, e.g. `os.RTLD_LAZY`).

Availability: Unix.

sys.set_int_max_str_digits(*maxdigits*)

Set the integer string conversion length limitation used by this interpreter. See also `get_int_max_str_digits()`.

Added in version 3.11.

sys.setprofile(*profilefunc*)

Set the system’s profile function, which allows you to implement a Python source code profiler in Python. See chapter The Python Profilers for more information on the Python profiler. The system’s profile function is called similarly to the system’s trace function (see `settrace()`), but it is called with different events, for example it isn’t called for each executed line of code (only on call and return, but the return event is reported even when an exception has been set). The function is thread-specific, but there is no way for the profiler to know about context switches between threads, so it does not make sense to use this in the presence of multiple threads. Also, its return value is not used, so it can simply return `None`. Error in the profile function will cause itself unset.

Note

The same tracing mechanism is used for **`setprofile()`** as `settrace()`. To trace calls with **`setprofile()`** inside a tracing function (e.g. in a debugger breakpoint), see `call_tracing()`.

Profile functions should have three arguments: *frame*, *event*, and *arg*. *frame* is the current stack frame. *event* is a string: `'call'`, `'return'`, `'c_call'`, `'c_return'`, or `'c_exception'`. *arg* depends on the event type.

The events have the following meaning:

`'call'`
A function is called (or some other code block entered). The profile function is called; *arg* is `None`.

`'return'`
A function (or other code block) is about to return. The profile function is called; *arg* is the value that will be returned, or `None` if the event is caused by an exception being raised.

`'c_call'`
A C function is about to be called. This may be an extension function or a built-in. *arg* is the C function object.

`'c_return'`
A C function has returned. *arg* is the C function object.

`'c_exception'`
A C function has raised an exception. *arg* is the C function object.

Raises an auditing event `sys.setprofile` with no arguments.

sys.setrecursionlimit(*limit*)

Set the maximum depth of the Python interpreter stack to *limit*. This limit prevents infinite recursion from causing an overflow of the C stack and crashing Python.

The highest possible limit is platform-dependent. A user may need to set the limit higher when they have a program that requires deep recursion and a platform that supports a higher limit. This should be done with care, because a too-high limit can lead to a crash.

If the new limit is too low at the current recursion depth, a **`RecursionError`** exception is raised.

Changed in version 3.5.1: A **`RecursionError`** exception is now raised if the new limit is too low at the current recursion depth.

sys.setswitchinterval(*interval*)

Set the interpreter’s thread switch interval (in seconds). This floating-point value determines the ideal duration of the “timeslices” allocated to concurrently running Python threads. Please note that the actual value can be higher, especially if long-running internal functions or methods are used. Also, which thread becomes scheduled at the end of the interval is the operating system’s decision. The interpreter doesn’t have its own scheduler.

Added in version 3.2.

sys.settrace(*tracefunc*)

Set the system’s trace function, which allows you to implement a Python source code debugger in Python. The function is thread-specific; for a debugger to support multiple threads, it must register a trace function using **`settrace()`** for each thread being debugged or use **`threading.settrace()`**.

Trace functions should have three arguments: *frame*, *event*, and *arg*. *frame* is the current stack frame. *event* is a string: `'call'`, `'line'`, `'return'`, `'exception'` or `'opcode'`. *arg* depends on the event type.

The trace function is invoked (with *event* set to `'call'`) whenever a new local scope is entered; it should return a reference to a local trace function to be used for the new scope, or `None` if the scope shouldn’t be traced.

The local trace function should return a reference to itself, or to another function which would then be used as the local trace function for the scope.

If there is any error occurred in the trace function, it will be unset, just like `settrace(None)` is called.

Note

Tracing is disabled while calling the trace function (e.g. a function set by **`settrace()`**). For recursive tracing see **`call_tracing()`**.

The events have the following meaning:

`'call'`
A function is called (or some other code block entered). The global trace function is called; *arg* is `None`; the return value specifies the local trace function.

`'line'`
The interpreter is about to execute a new line of code or re-execute the condition of a loop. The local trace function is called; *arg* is `None`; the return value specifies the new local trace function. See `Objects/lnotab_notes.txt` for a detailed explanation of how this works. Per-line events may be disabled for a frame by setting **`f_trace_lines`** to **`False`** on that frame.

`'return'`
A function (or other code block) is about to return. The local trace function is called; *arg* is the value that will be returned, or `None` if the event is caused by an exception being raised. The trace function’s return value is ignored.

`'exception'`

An exception has occurred. The local trace function is called; *arg* is a tuple `(exception, value, traceback)`; the return value specifies the new local trace function.

`'opcode'`

The interpreter is about to execute a new opcode (see **`dis`** for opcode details). The local trace function is called; *arg* is `None`; the return value specifies the new local trace function. Per-opcode events are not emitted by default: they must be explicitly requested by setting **`f_trace_opcodes`** to **`True`** on the frame.

Note that as an exception is propagated down the chain of callers, an `'exception'` event is generated at each level.

For more fine-grained usage, it's possible to set a trace function by assigning `frame.f_trace = tracefunc` explicitly, rather than relying on it being set indirectly via the return value from an already installed trace function. This is also required for activating the trace function on the current frame, which **`settrace()`** doesn't do. Note that in order for this to work, a global tracing function must have been installed with **`settrace()`** in order to enable the runtime tracing machinery, but it doesn't need to be the same tracing function (e.g. it could be a low overhead tracing function that simply returns `None` to disable itself immediately on each frame).

For more information on code and frame objects, refer to The standard type hierarchy.

Raises an auditing event `sys.settrace` with no arguments.

CPython implementation detail: The **`settrace()`** function is intended only for implementing debuggers, profilers, coverage tools and the like. Its behavior is part of the implementation platform, rather than part of the language definition, and thus may not be available in all Python implementations.

Changed in version 3.7: `'opcode'` event type added; **`f_trace_lines`** and **`f_trace_opcodes`** attributes added to frames

sys.set_asyncgen_hooks(*[firstiter] [, finalizer]*)

Accepts two optional keyword arguments which are callables that accept an asynchronous generator iterator as an argument. The *firstiter* callable will be called when an asynchronous generator is iterated for the first time. The *finalizer* will be called when an asynchronous generator is about to be garbage collected.

Raises an auditing event `sys.set_asyncgen_hooks_firstiter` with no arguments.

Raises an auditing event `sys.set_asyncgen_hooks_finalizer` with no arguments.

Two auditing events are raised because the underlying API consists of two calls, each of which must raise its own event.

Added in version 3.6: See **PEP 525** [https://peps.python.org/pep-0525/] for more details, and for a reference example of a *finalizer* method see the implementation of `asyncio.Loop.shutdown_asyncgens` in Lib/asyncio/base_events.py [https://github.com/python/cpython/tree/3.14/Lib/asyncio/base_events.py]

Note

This function has been added on a provisional basis (see **PEP 411** [https://peps.python.org/pep-0411/] for details.)

sys.set_coroutine_origin_tracking_depth(*depth*)

Allows enabling or disabling coroutine origin tracking. When enabled, the `cr_origin` attribute on coroutine objects will contain a tuple of (filename, line number, function name) tuples describing the traceback where the coroutine object was created, with the most recent call first. When disabled, `cr_origin` will be `None`.

To enable, pass a *depth* value greater than zero; this sets the number of frames whose information will be captured. To disable, set *depth* to zero.

This setting is thread-specific.

Added in version 3.7.

Note

This function has been added on a provisional basis (see **PEP 411** [https://peps.python.org/pep-0411/] for details.) Use it only for debugging purposes.

sys.activate_stack_trampoline(*backend*, */*)

Activate the stack profiler trampoline *backend*. The only supported backend is `"perf"`.

Stack trampolines cannot be activated if the JIT is active.

Availability: Linux.

Added in version 3.12.

See also

- Python support for the Linux perf profiler
- https://perf.wiki.kernel.org

sys.deactivate_stack_trampoline()

Deactivate the current stack profiler trampoline backend.

If no stack profiler is activated, this function has no effect.

Availability: Linux.

Added in version 3.12.

sys.is_stack_trampoline_active()

Return `True` if a stack profiler trampoline is active.

Availability: Linux.

Added in version 3.12.

sys.remote_exec(*pid*, *script*)

Executes *script*, a file containing Python code in the remote process with the given *pid*.

This function returns immediately, and the code will be executed by the target process’s main thread at the next available opportunity, similarly to how signals are handled. There is no interface to determine when the code has been executed. The caller is responsible for making sure that the file still exists whenever the remote process tries to read it and that it hasn’t been overwritten.

The remote process must be running a CPython interpreter of the same major and minor version as the local process. If either the local or remote interpreter is pre-release (alpha, beta, or release candidate) then the local and remote interpreters must be the same exact version.

When the code is executed in the remote process, an auditing event `sys.remote_exec` is raised with the *pid* and the path to the script file. This event is raised in the process that called **`sys.remote_exec()`**.

When the script is executed in the remote process, an auditing event `cpython.remote_debugger_script` is raised with the path in the remote process. This event is raised in the remote process, not the one that called **`sys.remote_exec()`**.

Availability: Unix, Windows.

Added in version 3.14.

sys._enablelegacywindowsfsencoding()

Changes the filesystem encoding and error handler to 'mbcs' and 'replace' respectively, for consistency with versions of Python prior to 3.6.

This is equivalent to defining the **`PYTHONLEGACYWINDOWSFSENCODING`** environment variable before launching Python.

See also **`sys.getfilesystemencoding()`** and **`sys.getfilesystemencodeerrors()`**.

Availability: Windows.

Note

Changing the filesystem encoding after Python startup is risky because the old fsencoding or paths encoded by the old fsencoding may be cached somewhere. Use **`PYTHONLEGACYWINDOWSFSENCODING`** instead.

Added in version 3.6: See **PEP 529** [https://peps.python.org/pep-0529/] for more details.

Deprecated since version 3.13, will be removed in version 3.16: Use **`PYTHONLEGACYWINDOWSFSENCODING`** instead.

sys.stdin
sys.stdout

sys.stderr

File objects used by the interpreter for standard input, output and errors:

- `stdin` is used for all interactive input (including calls to **`input()`**);
- `stdout` is used for the output of **`print()`** and expression statements and for the prompts of **`input()`**;
- The interpreter's own prompts and its error messages go to `stderr`.

These streams are regular text files like those returned by the **`open()`** function. Their parameters are chosen as follows:

- The encoding and error handling are is initialized from **`PyConfig.stdio_encoding`** and **`PyConfig.stdio_errors`**.

  On Windows, UTF-8 is used for the console device. Non-character devices such as disk files and pipes use the system locale encoding (i.e. the ANSI codepage). Non-console character devices such as NUL (i.e. where `isatty()` returns `True`) use the value of the console input and output codepages at startup, respectively for stdin and stdout/stderr. This defaults to the system locale encoding if the process is not initially attached to a console.

  The special behaviour of the console can be overridden by setting the environment variable PYTHONLEGACYWINDOWSSTDIO before starting Python. In that case, the console codepages are used as for any other character device.

  Under all platforms, you can override the character encoding by setting the **`PYTHONIOENCODING`** environment variable before starting Python or by using the new **`-X`** `utf8` command line option and **`PYTHONUTF8`** environment variable. However, for the Windows console, this only applies when `PYTHONLEGACYWINDOWSSTDIO` is also set.

- When interactive, the `stdout` stream is line-buffered. Otherwise, it is block-buffered like regular text files. The `stderr` stream is line-buffered in both cases. You can make both streams unbuffered by passing the `-u` command-line option or setting the `PYTHONUNBUFFERED` environment variable.

Changed in version 3.9: Non-interactive `stderr` is now line-buffered instead of fully buffered.

Note

To write or read binary data from/to the standard streams, use the underlying binary `buffer` object. For example, to write bytes to `stdout`, use `sys.stdout.buffer.write(b'abc')`.

However, if you are writing a library (and do not control in which context its code will be executed), be aware that the standard streams may be replaced with file-like objects like `io.StringIO` which do not support the **`buffer`** attribute.

sys.__stdin__
sys.__stdout__
sys.__stderr__

These objects contain the original values of `stdin`, `stderr` and `stdout` at the start of the program. They are used during finalization, and could be useful to print to the actual standard stream no matter if the `sys.std*` object has been redirected.

It can also be used to restore the actual files to known working file objects in case they have been overwritten with a broken object. However, the preferred way to do this is to explicitly save the previous stream before replacing it, and restore the saved object.

Note

Under some conditions `stdin`, `stdout` and `stderr` as well as the original values `__stdin__`, `__stdout__` and `__stderr__` can be `None`. It is usually the case for Windows GUI apps that aren't connected to a console and Python apps started with **pythonw**.

sys.stdlib_module_names

A frozenset of strings containing the names of standard library modules.

It is the same on all platforms. Modules which are not available on some platforms and modules disabled at Python build are also listed. All module kinds are listed: pure Python, built-in, frozen and extension modules. Test modules are excluded.

For packages, only the main package is listed: sub-packages and sub-modules are not listed. For example, the `email` package is listed, but the `email.mime` sub-package and the `email.message` sub-module are not listed.

See also the **`sys.builtin_module_names`** list.

Added in version 3.10.

sys.thread_info

A named tuple holding information about the thread implementation.

thread_info.name

The name of the thread implementation:

- `"nt"`: Windows threads
- `"pthread"`: POSIX threads
- `"pthread-stubs"`: stub POSIX threads (on WebAssembly platforms without threading support)
- `"solaris"`: Solaris threads

thread_info.lock

The name of the lock implementation:

- `"semaphore"`: a lock uses a semaphore
- `"mutex+cond"`: a lock uses a mutex and a condition variable
- `None` if this information is unknown

thread_info.version

The name and version of the thread library. It is a string, or `None` if this information is unknown.

Added in version 3.3.

sys.tracebacklimit

When this variable is set to an integer value, it determines the maximum number of levels of traceback information printed when an unhandled exception occurs. The default is `1000`. When set to `0` or less, all traceback information is suppressed and only the exception type and value are printed.

sys.unraisablehook(*unraisable*, */*)

Handle an unraisable exception.

Called when an exception has occurred but there is no way for Python to handle it. For example, when a destructor raises an exception or during garbage collection (`gc.collect()`).

The *unraisable* argument has the following attributes:

- **`exc_type`**: Exception type.
- **`exc_value`**: Exception value, can be `None`.
- **`exc_traceback`**: Exception traceback, can be `None`.
- **`err_msg`**: Error message, can be `None`.
- **`object`**: Object causing the exception, can be `None`.

The default hook formats **`err_msg`** and **`object`** as: `f'{err_msg}: {object!r}'`; use “Exception ignored in” error message if **`err_msg`** is `None`.

`sys.unraisablehook()` can be overridden to control how unraisable exceptions are handled.

See also

`excepthook()` which handles uncaught exceptions.

Warning

Storing **`exc_value`** using a custom hook can create a reference cycle. It should be cleared explicitly to break the reference cycle when the exception is no longer needed.

Storing **`object`** using a custom hook can resurrect it if it is set to an object which is being finalized. Avoid storing **`object`** after the custom hook completes to avoid resurrecting objects.

Raise an auditing event `sys.unraisablehook` with arguments *hook*, *unraisable* when an exception that cannot be handled occurs. The *unraisable* object is the same as what will be passed to the hook. If no hook has been set, *hook* may be `None`.

Added in version 3.8.

sys.version

A string containing the version number of the Python interpreter plus additional information on the build number and compiler used. This string is displayed when the interactive interpreter is started. Do not extract version information out of it, rather, use **`version_info`** and the functions provided by the **`platform`** module.

sys.api_version

The C API version, equivalent to the C macro **`PYTHON_API_VERSION`**. Defined for backwards compatibility.

Currently, this constant is not updated in new Python versions, and is not useful for versioning. This may change in the future.

sys.version_info

A tuple containing the five components of the version number: *major*, *minor*, *micro*, *releaselevel*, and *serial*. All values except *releaselevel* are integers; the release level is `'alpha'`, `'beta'`, `'candidate'`, or `'final'`. The `version_info` value corresponding to the Python version 2.0 is `(2, 0, 0, 'final', 0)`. The components can also be accessed by name, so `sys.version_info[0]` is equivalent to `sys.version_info.major` and so on.

Changed in version 3.1: Added named component attributes.

sys.warnoptions

This is an implementation detail of the warnings framework; do not modify this value. Refer to the **`warnings`** module for more information on the warnings framework.

sys.winver

The version number used to form registry keys on Windows platforms. This is stored as string resource 1000 in the Python DLL. The value is normally the major and minor versions of the running Python interpreter. It is provided in the `sys` module for informational purposes; modifying this value has no effect on the registry keys used by Python.

Availability: Windows.

sys.monitoring

Namespace containing functions and constants for register callbacks and controlling monitoring events. See `sys.monitoring` for details.

sys._xoptions

A dictionary of the various implementation-specific flags passed through the `-X` command-line option. Option names are either mapped to their values, if given explicitly, or to `True`. Example:

```
$ ./python -Xa=b -Xc
Python 3.2a3+ (py3k, Oct 16 2010, 20:14:50)
[GCC 4.4.3] on linux2
Type "help", "copyright", "credits" or "license" for more
information.
>>> import sys
>>> sys._xoptions
{'a': 'b', 'c': True}
```

CPython implementation detail: This is a CPython-specific way of accessing options passed through `-X`. Other implementations may export them through other means, or not at all.

Added in version 3.2.

Citations

[C99]

ISO/IEC 9899:1999. “Programming languages – C.” A public draft of this standard is available at https://www.open-std.org/jtc1/sc22/wg14/www/docs/n1256.pdf.

`sys.monitoring` — Execution event monitoring

Added in version 3.12.

Note

`sys.monitoring` is a namespace within the `sys` module, not an independent module, so there is no need to `import sys.monitoring`, simply `import sys` and then use `sys.monitoring`.

This namespace provides access to the functions and constants necessary to activate and control event monitoring.

As programs execute, events occur that might be of interest to tools that monitor execution. The `sys.monitoring` namespace provides means to receive callbacks when events of interest occur.

The monitoring API consists of three components:

- Tool identifiers
- Events
- Callbacks

Tool identifiers

A tool identifier is an integer and the associated name. Tool identifiers are used to discourage tools from interfering with each other and to allow multiple tools to operate at the same time. Currently tools are completely independent and cannot be used to monitor each other. This restriction may be lifted in the future.

Before registering or activating events, a tool should choose an identifier. Identifiers are integers in the range 0 to 5 inclusive.

Registering and using tools

sys.monitoring.use_tool_id(*tool_id: int*, *name: str*, */*) → None

Must be called before *tool_id* can be used. *tool_id* must be in the range 0 to 5 inclusive. Raises a **`ValueError`** if *tool_id* is in use.

sys.monitoring.clear_tool_id(*tool_id: int*, */*) → None

Unregister all events and callback functions associated with *tool_id*.

sys.monitoring.free_tool_id(*tool_id: int*, */*) → None

Should be called once a tool no longer requires *tool_id*. Will call **`clear_tool_id()`** before releasing *tool_id*.

sys.monitoring.get_tool(*tool_id: int*, */*) → str | None

Returns the name of the tool if *tool_id* is in use, otherwise it returns `None`. *tool_id* must be in the range 0 to 5 inclusive.

All IDs are treated the same by the VM with regard to events, but the following IDs are pre-defined to make co-operation of tools easier:

```
sys.monitoring.DEBUGGER_ID = 0
sys.monitoring.COVERAGE_ID = 1
sys.monitoring.PROFILER_ID = 2
sys.monitoring.OPTIMIZER_ID = 5
```

Events

The following events are supported:

sys.monitoring.events.BRANCH_LEFT

A conditional branch goes left.

It is up to the tool to determine how to present “left” and “right” branches. There is no guarantee which branch is “left” and which is “right”, except that it will be consistent for the duration of the program.

sys.monitoring.events.BRANCH_RIGHT

A conditional branch goes right.

sys.monitoring.events.CALL

A call in Python code (event occurs before the call).

sys.monitoring.events.C_RAISE

An exception raised from any callable, except for Python functions (event occurs after the exit).

sys.monitoring.events.C_RETURN

Return from any callable, except for Python functions (event occurs after the return).

sys.monitoring.events.EXCEPTION_HANDLED

An exception is handled.

sys.monitoring.events.INSTRUCTION

A VM instruction is about to be executed.

sys.monitoring.events.JUMP

An unconditional jump in the control flow graph is made.

sys.monitoring.events.LINE

An instruction is about to be executed that has a different line number from the preceding instruction.

sys.monitoring.events.PY_RESUME

Resumption of a Python function (for generator and coroutine functions), except for `throw()` calls.

sys.monitoring.events.PY_RETURN

Return from a Python function (occurs immediately before the return, the callee’s frame will be on the stack).

sys.monitoring.events.PY_START

Start of a Python function (occurs immediately after the call, the callee’s frame will be on the stack)

sys.monitoring.events.PY_THROW

A Python function is resumed by a `throw()` call.

sys.monitoring.events.PY_UNWIND

Exit from a Python function during exception unwinding. This includes exceptions raised directly within the function and that are allowed to continue to propagate.

sys.monitoring.events.PY_YIELD

Yield from a Python function (occurs immediately before the yield, the callee’s frame will be on the stack).

sys.monitoring.events.RAISE

An exception is raised, except those that cause a **`STOP_ITERATION`** event.

sys.monitoring.events.RERAISE

An exception is re-raised, for example at the end of a **`finally`** block.

sys.monitoring.events.STOP_ITERATION

An artificial **`StopIteration`** is raised; see the STOP_ITERATION event.

More events may be added in the future.

These events are attributes of the **`sys.monitoring.events`** namespace. Each event is represented as a power-of-2 integer constant. To define a set of events, simply bitwise OR the individual events together. For example, to specify both **`PY_RETURN`** and **`PY_START`** events, use the expression `PY_RETURN | PY_START`.

sys.monitoring.events.NO_EVENTS

An alias for `0` so users can do explicit comparisons like:

```
if get_events(DEBUGGER_ID) == NO_EVENTS:
    ...
```

Setting this event deactivates all events.

Local events

Local events are associated with normal execution of the program and happen at clearly defined locations. All local events can be disabled. The local events are:

- **`PY_START`**
- **`PY_RESUME`**
- **`PY_RETURN`**
- **`PY_YIELD`**
- **`CALL`**
- **`LINE`**
- **`INSTRUCTION`**

- `JUMP`
- `BRANCH_LEFT`
- `BRANCH_RIGHT`
- `STOP_ITERATION`

Deprecated event

- `BRANCH`

The `BRANCH` event is deprecated in 3.14. Using `BRANCH_LEFT` and `BRANCH_RIGHT` events will give much better performance as they can be disabled independently.

Ancillary events

Ancillary events can be monitored like other events, but are controlled by another event:

- `C_RAISE`
- `C_RETURN`

The `C_RETURN` and `C_RAISE` events are controlled by the `CALL` event. `C_RETURN` and `C_RAISE` events will only be seen if the corresponding `CALL` event is being monitored.

Other events

Other events are not necessarily tied to a specific location in the program and cannot be individually disabled via `DISABLE`.

The other events that can be monitored are:

- `PY_THROW`
- `PY_UNWIND`
- `RAISE`
- `EXCEPTION_HANDLED`

The STOP_ITERATION event

PEP 380 [https://peps.python.org/pep-0380/#use-of-stopiteration-to-return-values] specifies that a `StopIteration` exception is raised when returning a value from a generator or coroutine. However, this is a very inefficient way to return a value, so some Python implementations, notably CPython 3.12+, do not raise an exception unless it would be visible to other code.

To allow tools to monitor for real exceptions without slowing down generators and coroutines, the `STOP_ITERATION` event is provided. `STOP_ITERATION` can be locally disabled, unlike `RAISE`.

Note that the `STOP_ITERATION` event and the `RAISE` event for a `StopIteration` exception are equivalent, and are treated as interchangeable when generating events. Implementations will favor `STOP_ITERATION` for performance reasons, but may generate a `RAISE` event with a `StopIteration`.

Turning events on and off

In order to monitor an event, it must be turned on and a corresponding callback must be registered. Events can be turned on or off by setting the events either globally and/or for a particular code object. An event will trigger only once, even if it is turned on both globally and locally.

Setting events globally

Events can be controlled globally by modifying the set of events being monitored.

sys.monitoring.get_events(*tool_id: int*, */*) → int

Returns the `int` representing all the active events.

sys.monitoring.set_events(*tool_id: int*, *event_set: int*, */*) → None

Activates all events which are set in *event_set*. Raises a **`ValueError`** if *tool_id* is not in use.

No events are active by default.

Per code object events

Events can also be controlled on a per code object basis. The functions defined below which accept a **`types.CodeType`** should be prepared to accept a look-alike object from functions which are not defined in Python (see Monitoring C API).

sys.monitoring.get_local_events(*tool_id: int*, *code: CodeType*, */*) → int

Returns all the local events for *code*

sys.monitoring.set_local_events(*tool_id: int*, *code: CodeType*, *event_set: int*, */*) → None

Activates all the local events for *code* which are set in *event_set*. Raises a **`ValueError`** if *tool_id* is not in use.

Disabling events

sys.monitoring.DISABLE

A special value that can be returned from a callback function to disable events for the current code location.

Local events can be disabled for a specific code location by returning **`sys.monitoring.DISABLE`** from a callback function. This does not change which events are set, or any other code locations for the same event.

Disabling events for specific locations is very important for high performance monitoring. For example, a program can be run under a debugger with no overhead if the debugger disables all monitoring except for a few breakpoints.

If **`DISABLE`** is returned by a callback for a global event, **`ValueError`** will be raised by the interpreter in a non-specific location (that is, no traceback will be provided).

sys.monitoring.restart_events() → None

Enable all the events that were disabled by **`sys.monitoring.DISABLE`** for all tools.

Registering callback functions

sys.monitoring.register_callback(*tool_id: int*, *event: int*, *func: Callable* | *None*, */*) → Callable | None

Registers the callable *func* for the *event* with the given *tool_id*

If another callback was registered for the given *tool_id* and *event*, it is unregistered and returned. Otherwise **`register_callback()`** returns `None`.

Raises an auditing event `sys.monitoring.register_callback` with argument `func`.

Functions can be unregistered by calling `sys.monitoring.register_callback(tool_id, event, None)`.

Callback functions can be registered and unregistered at any time.

Callbacks are called only once regardless if the event is turned on both globally and locally. As such, if an event could be turned on for both global and local events by your code then the callback needs to be written to handle either trigger.

Callback function arguments

sys.monitoring.MISSING

> A special value that is passed to a callback function to indicate that there are no arguments to the call.

When an active event occurs, the registered callback function is called. Callback functions returning an object other than `DISABLE` will have no effect. Different events will provide the callback function with different arguments, as follows:

- `PY_START` and `PY_RESUME`:

  ```
  func(code: CodeType, instruction_offset: int) -> object
  ```

- `PY_RETURN` and `PY_YIELD`:

  ```
  func(code: CodeType, instruction_offset: int, retval: object) -> object
  ```

- `CALL`, `C_RAISE` and `C_RETURN` (*arg0* can be `MISSING` specifically):

  ```
  func(code: CodeType, instruction_offset: int, callable: object, arg0: object) -> object
  ```

  code represents the code object where the call is being made, while *callable* is the object that is about to be called (and thus triggered the event). If there are no arguments, *arg0* is set to `sys.monitoring.MISSING`.

  For instance methods, *callable* will be the function object as found on the class with *arg0* set to the instance (i.e. the `self` argument to the method).

- **`RAISE`**, **`RERAISE`**, **`EXCEPTION_HANDLED`**, **`PY_UNWIND`**, **`PY_THROW`** and **`STOP_ITERATION`**:

  ```
  func(code: CodeType, instruction_offset: int, exception: BaseException) -> object
  ```

- **`LINE`**:

  ```
  func(code: CodeType, line_number: int) -> object
  ```

- **`BRANCH_LEFT`**, **`BRANCH_RIGHT`** and **`JUMP`**:

  ```
  func(code: CodeType, instruction_offset: int, destination_offset: int) -> object
  ```

  Note that the *destination_offset* is where the code will next execute.

- **`INSTRUCTION`**:

  ```
  func(code: CodeType, instruction_offset: int) -> object
  ```

`sysconfig` — Provide access to Python’s configuration information

Added in version 3.2.

Source code: Lib/sysconfig
[https://github.com/python/cpython/tree/3.14/Lib/sysconfig]

The **`sysconfig`** module provides access to Python’s configuration information like the list of installation paths and the configuration variables relevant for the current platform.

Configuration variables

A Python distribution contains a `Makefile` and a `pyconfig.h` header file that are necessary to build both the Python binary itself and third-party C extensions compiled using `setuptools`.

`sysconfig` puts all variables found in these files in a dictionary that can be accessed using **`get_config_vars()`** or **`get_config_var()`**.

Notice that on Windows, it’s a much smaller set.

sysconfig.get_config_vars(**args*)

With no arguments, return a dictionary of all configuration variables relevant for the current platform.

With arguments, return a list of values that result from looking up each argument in the configuration variable dictionary.

For each argument, if the value is not found, return `None`.

sysconfig.get_config_var(*name*)

Return the value of a single variable *name*. Equivalent to `get_config_vars().get(name)`.

If *name* is not found, return `None`.

Example of usage:

```
>>> import sysconfig
>>> sysconfig.get_config_var('Py_ENABLE_SHARED')
0
>>> sysconfig.get_config_var('LIBDIR')
'/usr/local/lib'
>>> sysconfig.get_config_vars('AR', 'CXX')
['ar', 'g++']
```

Installation paths

Python uses an installation scheme that differs depending on the platform and on the installation options. These schemes are stored in `sysconfig` under unique identifiers based on the value returned by `os.name`. The schemes are used by package installers to determine where to copy files to.

Python currently supports nine schemes:

- *posix_prefix*: scheme for POSIX platforms like Linux or macOS. This is the default scheme used when Python or a component is installed.

- *posix_home*: scheme for POSIX platforms, when the *home* option is used. This scheme defines paths located under a specific home prefix.
- *posix_user*: scheme for POSIX platforms, when the *user* option is used. This scheme defines paths located under the user's home directory (`site.USER_BASE`).
- *posix_venv*: scheme for `Python virtual environments` on POSIX platforms; by default it is the same as *posix_prefix*.
- *nt*: scheme for Windows. This is the default scheme used when Python or a component is installed.
- *nt_user*: scheme for Windows, when the *user* option is used.
- *nt_venv*: scheme for `Python virtual environments` on Windows; by default it is the same as *nt*.
- *venv*: a scheme with values from either *posix_venv* or *nt_venv* depending on the platform Python runs on.
- *osx_framework_user*: scheme for macOS, when the *user* option is used.

Each scheme is itself composed of a series of paths and each path has a unique identifier. Python currently uses eight paths:

- *stdlib*: directory containing the standard Python library files that are not platform-specific.
- *platstdlib*: directory containing the standard Python library files that are platform-specific.
- *platlib*: directory for site-specific, platform-specific files.
- *purelib*: directory for site-specific, non-platform-specific files ('pure' Python).
- *include*: directory for non-platform-specific header files for the Python C-API.
- *platinclude*: directory for platform-specific header files for the Python C-API.
- *scripts*: directory for script files.

- *data*: directory for data files.

User scheme

This scheme is designed to be the most convenient solution for users that don’t have write permission to the global site-packages directory or don’t want to install into it.

Files will be installed into subdirectories of `site.USER_BASE` (written as *`userbase`* hereafter). This scheme installs pure Python modules and extension modules in the same location (also known as `site.USER_SITE`).

`posix_user`

Path	Installation directory
stdlib	`userbase/lib/pythonX.Y`
platstdlib	`userbase/lib/pythonX.Y`
platlib	`userbase/lib/pythonX.Y/site-packages`
purelib	`userbase/lib/pythonX.Y/site-packages`
include	`userbase/include/pythonX.Y`
scripts	`userbase/bin`

Path	**Installation directory**
data	`userbase`

`nt_user`

Path	**Installation directory**
stdlib	`userbase\PythonXY`
platstdlib	`userbase\PythonXY`
platlib	`userbase\PythonXY\site-packages`
purelib	`userbase\PythonXY\site-packages`
include	`userbase\PythonXY\Include`
scripts	`userbase\PythonXY\Scripts`
data	`userbase`

`osx_framework_user`

Path	Installation directory
stdlib	`userbase/lib/python`
platstdlib	`userbase/lib/python`
platlib	`userbase/lib/python/site-packages`
purelib	`userbase/lib/python/site-packages`
include	`userbase/include/pythonX.Y`
scripts	`userbase/bin`
data	`userbase`

Home scheme

The idea behind the “home scheme” is that you build and maintain a personal stash of Python modules. This scheme’s name is derived from the idea of a “home” directory on Unix, since it’s not unusual for a Unix user to make their home directory have a layout similar to `/usr/` or `/usr/local/`. This scheme can be used by anyone, regardless of the operating system they are installing for.

`posix_home`

Path	Installation directory
stdlib	`home/lib/python`
platstdlib	`home/lib/python`
platlib	`home/lib/python`
purelib	`home/lib/python`
include	`home/include/python`
platinclude	`home/include/python`
scripts	`home/bin`
data	`home`

Prefix scheme

The “prefix scheme” is useful when you wish to use one Python installation to perform the build/install (i.e., to run the setup script), but install modules into the third-party module directory of a different Python installation (or something that looks like a different Python installation). If this sounds a trifle unusual, it is—that’s why the user and home schemes come before. However, there are at least two known cases where the prefix scheme will be useful.

First, consider that many Linux distributions put Python in `/usr`, rather than the more traditional `/usr/local`. This is entirely appropriate, since in those cases Python is part of “the system” rather than a local add-on. However, if you are installing Python modules from source, you probably want them to go in `/usr/local/lib/python2.X` rather than `/usr/lib/python2.X`.

Another possibility is a network filesystem where the name used to write to a remote directory is different from the name used to read it: for example, the Python interpreter accessed as `/usr/local/bin/python` might search for modules in `/usr/local/lib/python2.X`, but those modules would have to be installed to, say, `/mnt/@server/export/lib/python2.X`.

`posix_prefix`

Path	Installation directory
stdlib	`prefix/lib/pythonX.Y`
platstdlib	`prefix/lib/pythonX.Y`
platlib	`prefix/lib/pythonX.Y/site-packages`
purelib	`prefix/lib/pythonX.Y/site-packages`
include	`prefix/include/pythonX.Y`

Path	Installation directory
platinclude	*prefix*`/include/python`*X.Y*
scripts	*prefix*`/bin`
data	*prefix*

`nt`

Path	Installation directory
stdlib	*prefix*`\Lib`
platstdlib	*prefix*`\Lib`
platlib	*prefix*`\Lib\site-packages`
purelib	*prefix*`\Lib\site-packages`
include	*prefix*`\Include`
platinclude	*prefix*`\Include`
scripts	*prefix*`\Scripts`

Path	Installation directory
data	*`prefix`*

Installation path functions

`sysconfig` provides some functions to determine these installation paths.

sysconfig.get_scheme_names()

Return a tuple containing all schemes currently supported in `sysconfig`.

sysconfig.get_default_scheme()

Return the default scheme name for the current platform.

Added in version 3.10: This function was previously named `_get_default_scheme()` and considered an implementation detail.

Changed in version 3.11: When Python runs from a virtual environment, the *venv* scheme is returned.

sysconfig.get_preferred_scheme(*key*)

Return a preferred scheme name for an installation layout specified by *key*.

key must be either `"prefix"`, `"home"`, or `"user"`.

The return value is a scheme name listed in `get_scheme_names()`. It can be passed to `sysconfig` functions that take a *scheme* argument, such as `get_paths()`.

Added in version 3.10.

Changed in version 3.11: When Python runs from a virtual environment and `key="prefix"`, the *venv* scheme is returned.

sysconfig._get_preferred_schemes()

Return a dict containing preferred scheme names on the current platform. Python implementers and redistributors may add their preferred schemes to the `_INSTALL_SCHEMES` module-level global value, and modify this function to return those scheme names, to e.g. provide different schemes for system and language package managers to use, so packages installed by either do not mix with those by the other.

End users should not use this function, but **`get_default_scheme()`** and **`get_preferred_scheme()`** instead.

Added in version 3.10.

sysconfig.get_path_names()

Return a tuple containing all path names currently supported in **`sysconfig`**.

sysconfig.get_path(*name*[, *scheme*[, *vars*[, *expand*]]])

Return an installation path corresponding to the path *name*, from the install scheme named *scheme*.

name has to be a value from the list returned by **`get_path_names()`**.

`sysconfig` stores installation paths corresponding to each path name, for each platform, with variables to be expanded. For instance the *stdlib* path for the *nt* scheme is: `{base}/Lib`.

`get_path()` will use the variables returned by `get_config_vars()` to expand the path. All variables have default values for each platform so one may call this function and get the default value.

If *scheme* is provided, it must be a value from the list returned by `get_scheme_names()`. Otherwise, the default scheme for the current platform is used.

If *vars* is provided, it must be a dictionary of variables that will update the dictionary returned by `get_config_vars()`.

If *expand* is set to `False`, the path will not be expanded using the variables.

If *name* is not found, raise a `KeyError`.

sysconfig.get_paths([*scheme*[, *vars*[, *expand*]]])

Return a dictionary containing all installation paths corresponding to an installation scheme. See `get_path()` for more information.

If *scheme* is not provided, will use the default scheme for the current platform.

If *vars* is provided, it must be a dictionary of variables that will update the dictionary used to expand the paths.

If *expand* is set to false, the paths will not be expanded.

If *scheme* is not an existing scheme, `get_paths()` will raise a `KeyError`.

Other functions

sysconfig.get_python_version()

Return the `MAJOR.MINOR` Python version number as a string. Similar to `'%d.%d' % sys.version_info[:2]`.

sysconfig.get_platform()

Return a string that identifies the current platform.

This is used mainly to distinguish platform-specific build directories and platform-specific built distributions. Typically includes the OS name and version and the architecture (as supplied by `os.uname()`), although the exact information included depends on the OS; e.g., on Linux, the kernel version isn't particularly important.

Examples of returned values:

Windows:

- win-amd64 (64-bit Windows on AMD64, aka x86_64, Intel64, and EM64T)
- win-arm64 (64-bit Windows on ARM64, aka AArch64)
- win32 (all others - specifically, sys.platform is returned)

POSIX based OS:

- linux-x86_64
- macosx-15.5-arm64
- macosx-26.0-universal2 (macOS on Apple Silicon or Intel)
- android-24-arm64_v8a

For other non-POSIX platforms, currently just returns `sys.platform`.

sysconfig.is_python_build()

Return `True` if the running Python interpreter was built from source and is being run from its built location, and not from a location resulting from e.g. running `make install` or installing via a binary installer.

sysconfig.parse_config_h(*fp*[, *vars*])

Parse a `config.h`-style file.

fp is a file-like object pointing to the `config.h`-like file.

A dictionary containing name/value pairs is returned. If an optional dictionary is passed in as the second argument, it is used instead of a new dictionary, and updated with the values read in the file.

sysconfig.get_config_h_filename()

Return the path of `pyconfig.h`.

sysconfig.get_makefile_filename()

Return the path of `Makefile`.

Command-line usage

You can use **`sysconfig`** as a script with Python’s *-m* option:

```
$ python -m sysconfig
Platform: "macosx-10.4-i386"
Python version: "3.2"
Current installation scheme: "posix_prefix"

Paths:
        data = "/usr/local"
        include =
"/Users/tarek/Dev/svn.python.org/py3k/Include"
        platinclude = "."
        platlib = "/usr/local/lib/python3.2/site-packages"
        platstdlib = "/usr/local/lib/python3.2"
        purelib = "/usr/local/lib/python3.2/site-packages"
        scripts = "/usr/local/bin"
        stdlib = "/usr/local/lib/python3.2"

Variables:
        AC_APPLE_UNIVERSAL_BUILD = "0"
        AIX_GENUINE_CPLUSPLUS = "0"
        AR = "ar"
        ARFLAGS = "rc"
        ...
```

This call will print in the standard output the information returned by `get_platform()`, `get_python_version()`, `get_path()` and `get_config_vars()`.

`builtins` — Built-in objects

This module provides direct access to all ‘built-in’ identifiers of Python; for example, `builtins.open` is the full name for the built-in function `open()`.

This module is not normally accessed explicitly by most applications, but can be useful in modules that provide objects with the same name as a built-in value, but in which the built-in of that name is also needed. For example, in a module that wants to implement an `open()` function that wraps the built-in `open()`, this module can be used directly:

```
import builtins

def open(path):
    f = builtins.open(path, 'r')
    return UpperCaser(f)

class UpperCaser:
    '''Wrapper around a file that converts output to
uppercase.'''

    def __init__(self, f):
        self._f = f

    def read(self, count=-1):
        return self._f.read(count).upper()

    # ...
```

As an implementation detail, most modules have the name `__builtins__` made available as part of their globals. The value of `__builtins__` is normally either this module or the value of this module’s `__dict__` attribute. Since this is an implementation detail, it may not be used by alternate implementations of Python.

See also

- Built-in Constants
- Built-in Exceptions
- Built-in Functions
- Built-in Types

`__main__` — Top-level code environment

In Python, the special name `__main__` is used for two important constructs:

1. the name of the top-level environment of the program, which can be checked using the `__name__ == '__main__'` expression; and
2. the `__main__.py` file in Python packages.

Both of these mechanisms are related to Python modules; how users interact with them and how they interact with each other. They are explained in detail below. If you’re new to Python modules, see the tutorial section Modules for an introduction.

`__name__ == '__main__'`

When a Python module or package is imported, `__name__` is set to the module’s name. Usually, this is the name of the Python file itself without the `.py` extension:

```
>>> import configparser
>>> configparser.__name__
'configparser'
```

If the file is part of a package, `__name__` will also include the parent package’s path:

```
>>> from concurrent.futures import process
>>> process.__name__
'concurrent.futures.process'
```

However, if the module is executed in the top-level code environment, its `__name__` is set to the string `'__main__'`.

What is the “top-level code environment”?

`__main__` is the name of the environment where top-level code is run. “Top-level code” is the first user-specified Python module that starts running. It’s “top-level” because it imports all other modules that the program needs. Sometimes “top-level code” is called an *entry point* to the application.

The top-level code environment can be:

- the scope of an interactive prompt:

  ```
  >>> __name__
  '__main__'
  ```

- the Python module passed to the Python interpreter as a file argument:

  ```
  $ python helloworld.py
  Hello, world!
  ```

- the Python module or package passed to the Python interpreter with the `-m` argument:

  ```
  $ python -m tarfile
  usage: tarfile.py [-h] [-v] (...)
  ```

- Python code read by the Python interpreter from standard input:

  ```
  $ echo "import this" | python
  The Zen of Python, by Tim Peters

  Beautiful is better than ugly.
  ```

```
Explicit is better than implicit.
...
```

- Python code passed to the Python interpreter with the `-c` argument:

  ```
  $ python -c "import this"
  The Zen of Python, by Tim Peters

  Beautiful is better than ugly.
  Explicit is better than implicit.
  ...
  ```

In each of these situations, the top-level module's `__name__` is set to `'__main__'`.

As a result, a module can discover whether or not it is running in the top-level environment by checking its own `__name__`, which allows a common idiom for conditionally executing code when the module is not initialized from an import statement:

```
if __name__ == '__main__':
    # Execute when the module is not initialized from an import statement.
    ...
```

See also

For a more detailed look at how `__name__` is set in all situations, see the tutorial section Modules.

Idiomatic Usage

Some modules contain code that is intended for script use only, like parsing command-line arguments or fetching data from standard input. If a module like this was imported from a different module, for example to unit test it, the script code would unintentionally execute as well.

This is where using the `if __name__ == '__main__'` code block comes in handy. Code within this block won’t run unless the module is executed in the top-level environment.

Putting as few statements as possible in the block below `if __name__ == '__main__'` can improve code clarity and correctness. Most often, a function named `main` encapsulates the program’s primary behavior:

```
# echo.py

import shlex
import sys

def echo(phrase: str) -> None:
   """A dummy wrapper around print."""
   # for demonstration purposes, you can imagine that there is
some
   # valuable and reusable logic inside this function
   print(phrase)

def main() -> int:
    """Echo the input arguments to standard output"""
    phrase = shlex.join(sys.argv)
    echo(phrase)
    return 0

if __name__ == '__main__':
    sys.exit(main())  # next section explains the use of
sys.exit
```

Note that if the module didn’t encapsulate code inside the `main` function but instead put it directly within the `if __name__ == '__main__'` block, the `phrase` variable would be global to the entire module. This is error-prone as other functions within the module could be unintentionally using the global variable instead of a local name. A `main` function solves this problem.

Using a `main` function has the added benefit of the `echo` function itself being isolated and importable elsewhere. When `echo.py` is imported, the `echo` and `main` functions will be defined, but neither of them will be called, because `__name__ != '__main__'`.

Packaging Considerations

`main` functions are often used to create command-line tools by specifying them as entry points for console scripts. When this is done, pip [https://pip.pypa.io/] inserts the function call into a template script, where the return value of `main` is passed into `sys.exit()`. For example:

```
sys.exit(main())
```

Since the call to `main` is wrapped in `sys.exit()`, the expectation is that your function will return some value acceptable as an input to `sys.exit()`; typically, an integer or `None` (which is implicitly returned if your function does not have a return statement).

By proactively following this convention ourselves, our module will have the same behavior when run directly (i.e. `python echo.py`) as it will have if we later package it as a console script entry-point in a pip-installable package.

In particular, be careful about returning strings from your `main` function. `sys.exit()` will interpret a string argument as a failure message, so your program will have an exit code of `1`, indicating failure, and the string will be written to `sys.stderr`. The `echo.py` example from earlier exemplifies using the `sys.exit(main())` convention.

See also

Python Packaging User Guide [https://packaging.python.org/] contains a collection of tutorials and references on how to distribute and install Python packages with modern tools.

`__main__.py` in Python Packages

If you are not familiar with Python packages, see section Packages of the tutorial. Most commonly, the `__main__.py` file is used to provide a command-line interface for a package. Consider the following hypothetical package, “bandclass”:

```
bandclass
  ├── __init__.py
  ├── __main__.py
  └── student.py
```

`__main__.py` will be executed when the package itself is invoked directly from the command line using the `-m` flag. For example:

```
$ python -m bandclass
```

This command will cause `__main__.py` to run. How you utilize this mechanism will depend on the nature of the package you are writing, but in this hypothetical case, it might make sense to allow the teacher to search for students:

```
# bandclass/__main__.py

import sys
from .student import search_students

student_name = sys.argv[1] if len(sys.argv) >= 2 else ''
print(f'Found student: {search_students(student_name)}')
```

Note that `from .student import search_students` is an example of a relative import. This import style can be used when referencing modules within a package. For more details, see Intra-package References in the Modules section of the tutorial.

Idiomatic Usage

The content of `__main__.py` typically isn't fenced with an `if __name__ == '__main__'` block. Instead, those files are kept short and import functions to execute from other modules. Those other modules can then be easily unit-tested and are properly reusable.

If used, an `if __name__ == '__main__'` block will still work as expected for a `__main__.py` file within a package, because its `__name__` attribute will include the package's path if imported:

```
>>> import asyncio.__main__
>>> asyncio.__main__.__name__
'asyncio.__main__'
```

This won't work for `__main__.py` files in the root directory of a `.zip` file though. Hence, for consistency, a minimal `__main__.py` without a `__name__` check is preferred.

See also

See **`venv`** for an example of a package with a minimal `__main__.py` in the standard library. It doesn't contain a `if __name__ == '__main__'` block. You can invoke it with `python -m venv [directory]`.

See **`runpy`** for more details on the **`-m`** flag to the interpreter executable.

See `zipapp` for how to run applications packaged as *.zip* files. In this case Python looks for a `__main__.py` file in the root directory of the archive.

`import __main__`

Regardless of which module a Python program was started with, other modules running within that same program can import the top-level environment's scope (namespace) by importing the `__main__` module. This doesn't import a `__main__.py` file but rather whichever module that received the special name `'__main__'`.

Here is an example module that consumes the `__main__` namespace:

```
# namely.py

import __main__

def did_user_define_their_name():
    return 'my_name' in dir(__main__)

def print_user_name():
    if not did_user_define_their_name():
        raise ValueError('Define the variable `my_name`!')

    print(__main__.my_name)
```

Example usage of this module could be as follows:

```
# start.py

import sys

from namely import print_user_name

# my_name = "Dinsdale"
```

```
def main():
    try:
        print_user_name()
    except ValueError as ve:
        return str(ve)

if __name__ == "__main__":
    sys.exit(main())
```

Now, if we started our program, the result would look like this:

```
$ python start.py
Define the variable `my_name`!
```

The exit code of the program would be 1, indicating an error. Uncommenting the line with `my_name = "Dinsdale"` fixes the program and now it exits with status code 0, indicating success:

```
$ python start.py
Dinsdale
```

Note that importing `__main__` doesn't cause any issues with unintentionally running top-level code meant for script use which is put in the `if __name__ == "__main__"` block of the `start` module. Why does this work?

Python inserts an empty `__main__` module in **`sys.modules`** at interpreter startup, and populates it by running top-level code. In our example this is the `start` module which runs line by line and imports `namely`. In turn, `namely` imports `__main__` (which is really `start`). That's an import cycle! Fortunately, since the partially populated `__main__` module is present in **`sys.modules`**, Python passes that to `namely`. See Special considerations for __main__ in the import system's reference for details on how this works.

The Python REPL is another example of a “top-level environment”, so anything defined in the REPL becomes part of the `__main__` scope:

```
>>> import namely
>>> namely.did_user_define_their_name()
False
>>> namely.print_user_name()
Traceback (most recent call last):
...
ValueError: Define the variable `my_name`!
>>> my_name = 'Jabberwocky'
>>> namely.did_user_define_their_name()
True
>>> namely.print_user_name()
Jabberwocky
```

The `__main__` scope is used in the implementation of **`pdb`** and **`rlcompleter`**.

`warnings` — Warning control

Source code: Lib/warnings.py
[https://github.com/python/cpython/tree/3.14/Lib/warnings.py]

Warning messages are typically issued in situations where it is useful to alert the user of some condition in a program, where that condition (normally) doesn’t warrant raising an exception and terminating the program. For example, one might want to issue a warning when a program uses an obsolete module.

Python programmers issue warnings by calling the **`warn()`** function defined in this module. (C programmers use **`PyErr_WarnEx()`**; see Exception Handling for details).

Warning messages are normally written to **`sys.stderr`**, but their disposition can be changed flexibly, from ignoring all warnings to turning them into exceptions. The disposition of warnings can vary based on the warning category, the text of the warning message, and the source location where it is issued. Repetitions of a particular warning for the same source location are typically suppressed.

There are two stages in warning control: first, each time a warning is issued, a determination is made whether a message should be issued or not; next, if a message is to be issued, it is formatted and printed using a user-settable hook.

The determination whether to issue a warning message is controlled by the warning filter, which is a sequence of matching rules and actions. Rules can be added to the filter by calling `filterwarnings()` and reset to its default state by calling `resetwarnings()`.

The printing of warning messages is done by calling `showwarning()`, which may be overridden; the default implementation of this function formats the message by calling `formatwarning()`, which is also available for use by custom implementations.

See also

`logging.captureWarnings()` allows you to handle all warnings with the standard logging infrastructure.

Warning Categories

There are a number of built-in exceptions that represent warning categories. This categorization is useful to be able to filter out groups of warnings.

While these are technically built-in exceptions, they are documented here, because conceptually they belong to the warnings mechanism.

User code can define additional warning categories by subclassing one of the standard warning categories. A warning category must always be a subclass of the `Warning` class.

The following warnings category classes are currently defined:

Class	Description
`Warning`	This is the base class of all warning category classes. It is a subclass of `Exception`.
`UserWarning`	The default category for `warn()`.
`DeprecationWarning`	Base category for warnings about deprecated features when those warnings are intended for other Python developers (ignored by default, unless triggered by code in `__main__`).
`SyntaxWarning`	Base category for warnings about dubious syntactic features (typically emitted when compiling Python source code, and hence may not be suppressed by runtime filters)
`RuntimeWarning`	Base category for warnings about dubious runtime features.
`FutureWarning`	Base category for warnings about deprecated features when those warnings are intended for end users of applications that are written in Python.

Class	Description
`PendingDeprecationWarning`	Base category for warnings about features that will be deprecated in the future (ignored by default).
`ImportWarning`	Base category for warnings triggered during the process of importing a module (ignored by default).
`UnicodeWarning`	Base category for warnings related to Unicode.
`BytesWarning`	Base category for warnings related to `bytes` and `bytearray`.
`ResourceWarning`	Base category for warnings related to resource usage (ignored by default).

Changed in version 3.7: Previously `DeprecationWarning` and `FutureWarning` were distinguished based on whether a feature was being removed entirely or changing its behaviour. They are now distinguished based on their intended audience and the way they’re handled by the default warnings filters.

The Warnings Filter

The warnings filter controls whether warnings are ignored, displayed, or turned into errors (raising an exception).

Conceptually, the warnings filter maintains an ordered list of filter specifications; any specific warning is matched against each filter specification in the list in turn until a match is found; the filter determines the disposition of the match. Each entry is a tuple of the form (*action*, *message*, *category*, *module*, *lineno*), where:

- *action* is one of the following strings:

Value	**Disposition**
`"default"`	print the first occurrence of matching warnings for each location (module + line number) where the warning is issued
`"error"`	turn matching warnings into exceptions
`"ignore"`	never print matching warnings
`"always"`	always print matching warnings
`"all"`	alias to “always”
`"module"`	print the first occurrence of matching warnings for each module where the warning is issued (regardless of line number)

Value	Disposition
`"once"`	print only the first occurrence of matching warnings, regardless of location

- *message* is a string containing a regular expression that the start of the warning message must match, case-insensitively. In `-W` and `PYTHONWARNINGS`, *message* is a literal string that the start of the warning message must contain (case-insensitively), ignoring any whitespace at the start or end of *message*.
- *category* is a class (a subclass of `Warning`) of which the warning category must be a subclass in order to match.
- *module* is a string containing a regular expression that the start of the fully qualified module name must match, case-sensitively. In `-W` and `PYTHONWARNINGS`, *module* is a literal string that the fully qualified module name must be equal to (case-sensitively), ignoring any whitespace at the start or end of *module*.
- *lineno* is an integer that the line number where the warning occurred must match, or `0` to match all line numbers.

Since the `Warning` class is derived from the built-in `Exception` class, to turn a warning into an error we simply raise `category(message)`.

If a warning is reported and doesn't match any registered filter then the "default" action is applied (hence its name).

Repeated Warning Suppression Criteria

The filters that suppress repeated warnings apply the following criteria to determine if a warning is considered a repeat:

- `"default"`: A warning is considered a repeat only if the (*message*, *category*, *module*, *lineno*) are all the same.
- `"module"`: A warning is considered a repeat if the (*message*, *category*, *module*) are the same, ignoring the line number.
- `"once"`: A warning is considered a repeat if the (*message*, *category*) are the same, ignoring the module and line number.

Describing Warning Filters

The warnings filter is initialized by `-W` options passed to the Python interpreter command line and the `PYTHONWARNINGS` environment variable. The interpreter saves the arguments for all supplied entries without interpretation in `sys.warnoptions`; the `warnings` module parses these when it is first imported (invalid options are ignored, after printing a message to `sys.stderr`).

Individual warnings filters are specified as a sequence of fields separated by colons:

```
action:message:category:module:line
```

The meaning of each of these fields is as described in The Warnings Filter. When listing multiple filters on a single line (as for `PYTHONWARNINGS`), the individual filters are separated by commas and the filters listed later take precedence over those listed before them (as they’re applied left-to-right, and the most recently applied filters take precedence over earlier ones).

Commonly used warning filters apply to either all warnings, warnings in a particular category, or warnings raised by particular modules or packages. Some examples:

```
default                      # Show all warnings (even those 
ignored by default)
ignore                       # Ignore all warnings
error                        # Convert all warnings to errors
error::ResourceWarning       # Treat ResourceWarning messages 
as errors
default::DeprecationWarning  # Show DeprecationWarning messages
ignore,default:::mymodule    # Only report warnings triggered 
by "mymodule"
error:::mymodule             # Convert warnings to errors in 
"mymodule"
```

Default Warning Filter

By default, Python installs several warning filters, which can be overridden by the `-W` command-line option, the `PYTHONWARNINGS` environment variable and calls to `filterwarnings()`.

In regular release builds, the default warning filter has the following entries (in order of precedence):

```
default::DeprecationWarning:__main__
ignore::DeprecationWarning
ignore::PendingDeprecationWarning
ignore::ImportWarning
ignore::ResourceWarning
```

In a debug build, the list of default warning filters is empty.

Changed in version 3.2: `DeprecationWarning` is now ignored by default in addition to `PendingDeprecationWarning`.

Changed in version 3.7: `DeprecationWarning` is once again shown by default when triggered directly by code in `__main__`.

Changed in version 3.7: `BytesWarning` no longer appears in the default filter list and is instead configured via `sys.warnoptions` when `-b` is specified twice.

Overriding the default filter

Developers of applications written in Python may wish to hide *all* Python level warnings from their users by default, and only display them when running tests or otherwise working on the application. The `sys.warnoptions` attribute used to pass filter configurations to the interpreter can be used as a marker to indicate whether or not warnings should be disabled:

```
import sys

if not sys.warnoptions:
    import warnings
    warnings.simplefilter("ignore")
```

Developers of test runners for Python code are advised to instead ensure that *all* warnings are displayed by default for the code under test, using code like:

```
import sys

if not sys.warnoptions:
    import os, warnings
    warnings.simplefilter("default") # Change the filter in
this process
    os.environ["PYTHONWARNINGS"] = "default" # Also affect
subprocesses
```

Finally, developers of interactive shells that run user code in a namespace other than `__main__` are advised to ensure that `DeprecationWarning` messages are made visible by default, using code like the following (where `user_ns` is the module used to execute code entered interactively):

```
import warnings
warnings.filterwarnings("default", category=DeprecationWarning,

module=user_ns.get("__name__"))
```

Temporarily Suppressing Warnings

If you are using code that you know will raise a warning, such as a deprecated function, but do not want to see the warning (even when warnings have been explicitly configured via the command line), then it is possible to suppress the warning using the **`catch_warnings`** context manager:

```
import warnings

def fxn():
    warnings.warn("deprecated", DeprecationWarning)

with warnings.catch_warnings():
    warnings.simplefilter("ignore")
    fxn()
```

While within the context manager all warnings will simply be ignored. This allows you to use known-deprecated code without having to see the warning while not suppressing the warning for other code that might not be aware of its use of deprecated code.

> **Note**
>
> See Concurrent safety of Context Managers for details on the concurrency-safety of the **`catch_warnings`** context

manager when used in programs using multiple threads or async functions.

Testing Warnings

To test warnings raised by code, use the `catch_warnings` context manager. With it you can temporarily mutate the warnings filter to facilitate your testing. For instance, do the following to capture all raised warnings to check:

```
import warnings

def fxn():
    warnings.warn("deprecated", DeprecationWarning)

with warnings.catch_warnings(record=True) as w:
    # Cause all warnings to always be triggered.
    warnings.simplefilter("always")
    # Trigger a warning.
    fxn()
    # Verify some things
    assert len(w) == 1
    assert issubclass(w[-1].category, DeprecationWarning)
    assert "deprecated" in str(w[-1].message)
```

One can also cause all warnings to be exceptions by using `error` instead of `always`. One thing to be aware of is that if a warning has already been raised because of a `once`/`default` rule, then no matter what filters are set the warning will not be seen again unless the warnings registry related to the warning has been cleared.

Once the context manager exits, the warnings filter is restored to its state when the context was entered. This prevents tests from changing the warnings filter in unexpected ways between tests and leading to indeterminate test results.

Note

See Concurrent safety of Context Managers for details on the concurrency-safety of the `catch_warnings` context manager when used in programs using multiple threads or async functions.

When testing multiple operations that raise the same kind of warning, it is important to test them in a manner that confirms each operation is raising a new warning (e.g. set warnings to be raised as exceptions and check the operations raise exceptions, check that the length of the warning list continues to increase after each operation, or else delete the previous entries from the warnings list before each new operation).

Updating Code For New Versions of Dependencies

Warning categories that are primarily of interest to Python developers (rather than end users of applications written in Python) are ignored by default.

Notably, this “ignored by default” list includes `DeprecationWarning` (for every module except `__main__`), which means developers should make sure to test their code with typically ignored warnings made visible in order to receive timely notifications of future breaking API changes (whether in the standard library or third party packages).

In the ideal case, the code will have a suitable test suite, and the test runner will take care of implicitly enabling all warnings when running tests (the test runner provided by the `unittest` module does this).

In less ideal cases, applications can be checked for use of deprecated interfaces by passing `-Wd` to the Python interpreter (this is shorthand for **`-W default`**) or setting `PYTHONWARNINGS=default` in the environment. This enables default handling for all warnings, including those that are ignored by default. To change what action is taken for encountered warnings you can change what argument is passed to `-W` (e.g. **`-W error`**). See the `-W` flag for more details on what is possible.

Available Functions

warnings.warn(*message*, *category=None*, *stacklevel=1*, *source=None*, ***, *skip_file_prefixes=()*)

Issue a warning, or maybe ignore it or raise an exception. The *category* argument, if given, must be a warning category class; it defaults to **`UserWarning`**. Alternatively, *message* can be a **`Warning`** instance, in which case *category* will be ignored and `message.__class__` will be used. In this case, the message text will be `str(message)`. This function raises an exception if the particular warning issued is changed into an error by the warnings filter. The *stacklevel* argument can be used by wrapper functions written in Python, like this:

```
def deprecated_api(message):
    warnings.warn(message, DeprecationWarning, stacklevel=2)
```

This makes the warning refer to `deprecated_api`’s caller, rather than to the source of `deprecated_api` itself (since the latter would defeat the purpose of the warning message).

The *skip_file_prefixes* keyword argument can be used to indicate which stack frames are ignored when counting stack levels. This can be useful when you want the warning to always appear at call sites outside of a package when a constant *stacklevel* does not fit all call paths or is otherwise challenging to maintain. If supplied, it must be a tuple of strings. When prefixes are supplied, stacklevel is implicitly overridden to be `max(2, stacklevel)`. To cause a warning to be attributed to the caller from outside of the current package you might write:

```
# example/lower.py
_warn_skips = (os.path.dirname(__file__),)

def one_way(r_luxury_yacht=None, t_wobbler_mangrove=None):
    if r_luxury_yacht:
        warnings.warn("Please migrate to 
t_wobbler_mangrove=.",
                      skip_file_prefixes=_warn_skips)

# example/higher.py
from . import lower

def another_way(**kw):
    lower.one_way(**kw)
```

This makes the warning refer to both the `example.lower.one_way()` and `example.higher.another_way()` call sites only from calling code living outside of `example` package.

source, if supplied, is the destroyed object which emitted a `ResourceWarning`.

Changed in version 3.6: Added *source* parameter.

Changed in version 3.12: Added *skip_file_prefixes*.

warnings.warn_explicit(*message*, *category*, *filename*, *lineno*, *module=None*, *registry=None*, *module_globals=None*, *source=None*)

This is a low-level interface to the functionality of `warn()`, passing in explicitly the message, category, filename and line number, and optionally other arguments. *message* must be a string and *category* a subclass of `Warning` or *message* may be a `Warning` instance, in which case *category* will be ignored.

module, if supplied, should be the module name. If no module is passed, the filename with `.py` stripped is used.

registry, if supplied, should be the `__warningregistry__` dictionary of the module. If no registry is passed, each warning is treated as the first occurrence, that is, filter actions `"default"`, `"module"` and `"once"` are handled as `"always"`.

module_globals, if supplied, should be the global namespace in use by the code for which the warning is issued. (This argument is used to support displaying source for modules found in zipfiles or other non-filesystem import sources).

source, if supplied, is the destroyed object which emitted a `ResourceWarning`.

Changed in version 3.6: Add the *source* parameter.

warnings.showwarning(*message*, *category*, *filename*, *lineno*, *file=None*, *line=None*)

Write a warning to a file. The default implementation calls `formatwarning(message, category, filename, lineno, line)` and writes the resulting string to *file*, which defaults to `sys.stderr`. You may replace this function with any callable by assigning to `warnings.showwarning`. *line* is a line of source code to be included in the warning message; if *line* is not supplied, `showwarning()` will try to read the line specified by *filename* and *lineno*.

warnings.formatwarning(*message*, *category*, *filename*, *lineno*, *line=None*)

Format a warning the standard way. This returns a string which may contain embedded newlines and ends in a newline. *line* is a line of source code to be included in the warning message; if *line* is not supplied, `formatwarning()` will try to read the line specified by *filename* and *lineno*.

warnings.filterwarnings(*action*, *message=''*, *category=Warning*, *module=''*, *lineno=0*, *append=False*)

Insert an entry into the list of warnings filter specifications. The entry is inserted at the front by default; if *append* is true, it is inserted at the end. This checks the types of the arguments, compiles the *message* and *module* regular expressions, and inserts them as a tuple in the list of warnings filters. Entries closer to the front of the list override entries later in the list, if both match a particular warning. Omitted arguments default to a value that matches everything.

warnings.simplefilter(*action*, *category=Warning*, *lineno=0*, *append=False*)

Insert a simple entry into the list of warnings filter specifications. The meaning of the function parameters is as for `filterwarnings()`, but regular expressions are not needed as the filter inserted always matches any message in any module as long as the category and line number match.

warnings.resetwarnings()

Reset the warnings filter. This discards the effect of all previous calls to `filterwarnings()`, including that of the `-W` command line options and calls to `simplefilter()`.

@warnings.deprecated(*msg*, *, *category=DeprecationWarning*, *stacklevel=1*)

Decorator to indicate that a class, function or overload is deprecated.

When this decorator is applied to an object, deprecation warnings may be emitted at runtime when the object is used. static type checkers will also generate a diagnostic on usage of the deprecated object.

Usage:

```
from warnings import deprecated
from typing import overload

@deprecated("Use B instead")
class A:
    pass

@deprecated("Use g instead")
def f():
    pass

@overload
@deprecated("int support is deprecated")
def g(x: int) -> int: ...
@overload
def g(x: str) -> int: ...
```

The warning specified by *category* will be emitted at runtime on use of deprecated objects. For functions, that happens on calls; for classes, on instantiation and on creation of subclasses. If the *category* is `None`, no warning is emitted at runtime. The *stacklevel* determines where the warning is emitted. If it is `1` (the default), the warning is emitted at the direct caller of the deprecated object; if it is higher, it is emitted further up the stack. Static type checker behavior is not affected by the *category* and *stacklevel* arguments.

The deprecation message passed to the decorator is saved in the `__deprecated__` attribute on the decorated object. If applied to an overload, the decorator must be after the [`@~typing.overload`] decorator for the attribute to exist on the overload as returned by [`typing.get_overloads()`].

Added in version 3.13: See **PEP 702** [https://peps.python.org/pep-0702/].

Available Context Managers

class warnings.catch_warnings(*, *record=False*, *module=None*, *action=None*, *category=Warning*, *lineno=0*, *append=False*)

A context manager that copies and, upon exit, restores the warnings filter and the `showwarning()` function. If the *record* argument is `False` (the default) the context manager returns `None` on entry. If *record* is `True`, a list is returned that is progressively populated with objects as seen by a custom `showwarning()` function (which also suppresses output to `sys.stdout`). Each object in the list has attributes with the same names as the arguments to `showwarning()`.

The *module* argument takes a module that will be used instead of the module returned when you import `warnings` whose filter will be protected. This argument exists primarily for testing the `warnings` module itself.

If the *action* argument is not `None`, the remaining arguments are passed to `simplefilter()` as if it were called immediately on entering the context.

See The Warnings Filter for the meaning of the *category* and *lineno* parameters.

Note

See Concurrent safety of Context Managers for details on the concurrency-safety of the `catch_warnings` context manager when used in programs using multiple threads or async functions.

Changed in version 3.11: Added the *action*, *category*, *lineno*, and *append* parameters.

Concurrent safety of Context Managers

The behavior of `catch_warnings` context manager depends on the `sys.flags.context_aware_warnings` flag. If the flag is true, the context manager behaves in a concurrent-safe fashion and otherwise not. Concurrent-safe means that it is both thread-safe and safe to use within asyncio coroutines and tasks. Being thread-safe means that behavior is predictable in a multi-threaded program. The flag defaults to true for free-threaded builds and false otherwise.

If the `context_aware_warnings` flag is false, then `catch_warnings` will modify the global attributes of the `warnings` module. This is not safe if used within a concurrent program (using multiple threads or using asyncio coroutines). For example, if two or more threads use the `catch_warnings` class at the same time, the behavior is undefined.

If the flag is true, `catch_warnings` will not modify global attributes and will instead use a `ContextVar` to store the newly established warning filtering state. A context variable provides thread-local storage and it makes the use of `catch_warnings` thread-safe.

The *record* parameter of the context handler also behaves differently depending on the value of the flag. When *record* is true and the flag is false, the context manager works by replacing and then later restoring the module’s `showwarning()` function. That is not concurrent-safe.

When *record* is true and the flag is true, the `showwarning()` function is not replaced. Instead, the recording status is indicated by an internal property in the context variable. In this case, the `showwarning()` function will not be restored when exiting the context handler.

The `context_aware_warnings` flag can be set the `-X context_aware_warnings` command-line option or by the `PYTHON_CONTEXT_AWARE_WARNINGS` environment variable.

> **Note**
>
> It is likely that most programs that desire thread-safe behaviour of the warnings module will also want to set the `thread_inherit_context` flag to true. That flag causes threads created by `threading.Thread` to start with a copy of the context variables from the thread starting it. When true, the context established by `catch_warnings` in one thread will also apply to new threads started by it. If false, new threads will start with an empty warnings context variable, meaning that any filtering that was established by a `catch_warnings` context manager will no longer be active.

Changed in version 3.14: Added the `sys.flags.context_aware_warnings` flag and the use of a context variable for `catch_warnings` if the flag is true. Previous versions of Python acted as if the flag was always set to false.

`dataclasses` — Data Classes

Source code: Lib/dataclasses.py
[https://github.com/python/cpython/tree/3.14/Lib/dataclasses.py]

This module provides a decorator and functions for automatically adding generated special methods such as `__init__()` and `__repr__()` to user-defined classes. It was originally described in **PEP 557** [https://peps.python.org/pep-0557/].

The member variables to use in these generated methods are defined using **PEP 526** [https://peps.python.org/pep-0526/] type annotations. For example, this code:

```
from dataclasses import dataclass

@dataclass
class InventoryItem:
    """Class for keeping track of an item in inventory."""
    name: str
    unit_price: float
    quantity_on_hand: int = 0

    def total_cost(self) -> float:
        return self.unit_price * self.quantity_on_hand
```

will add, among other things, a `__init__()` that looks like:

```
def __init__(self, name: str, unit_price: float,
quantity_on_hand: int = 0):
    self.name = name
    self.unit_price = unit_price
    self.quantity_on_hand = quantity_on_hand
```

Note that this method is automatically added to the class: it is not directly specified in the `InventoryItem` definition shown above.

Added in version 3.7.

Module contents

@dataclasses.dataclass(*, *init=True*, *repr=True*, *eq=True*, *order=False*, *unsafe_hash=False*, *frozen=False*, *match_args=True*, *kw_only=False*, *slots=False*, *weakref_slot=False*)

This function is a decorator that is used to add generated special methods to classes, as described below.

The `@dataclass` decorator examines the class to find `field`s. A `field` is defined as a class variable that has a type annotation. With two exceptions described below, nothing in `@dataclass` examines the type specified in the variable annotation.

The order of the fields in all of the generated methods is the order in which they appear in the class definition.

The `@dataclass` decorator will add various “dunder” methods to the class, described below. If any of the added methods already exist in the class, the behavior depends on the parameter, as documented below. The decorator returns the same class that it is called on; no new class is created.

If `@dataclass` is used just as a simple decorator with no parameters, it acts as if it has the default values documented in this signature. That is, these three uses of `@dataclass` are equivalent:

```
@dataclass
class C:
    ...

@dataclass()
class C:
    ...

@dataclass(init=True, repr=True, eq=True, order=False,
unsafe_hash=False, frozen=False,
           match_args=True, kw_only=False, slots=False,
weakref_slot=False)
class C:
    ...
```

The parameters to `@dataclass` are:

- *init*: If true (the default), a `__init__()` method will be generated.

  If the class already defines **`__init__()`**, this parameter is ignored.

- *repr*: If true (the default), a `__repr__()` method will be generated. The generated repr string will have the class name and the name and repr of each field, in the order they are defined in the class. Fields that are marked as being excluded from the repr are not included. For example: `InventoryItem(name='widget', unit_price=3.0, quantity_on_hand=10)`.

  If the class already defines **`__repr__()`**, this parameter is ignored.

- *eq*: If true (the default), an `__eq__()` method will be generated. This method compares the class as if it were a tuple of its fields, in order. Both instances in the comparison must be of the identical type.

  If the class already defines **`__eq__()`**, this parameter is ignored.

- *order*: If true (the default is `False`), `__lt__()`, `__le__()`, `__gt__()`, and `__ge__()` methods will be generated. These compare the class as if it were a tuple of its fields, in order. Both instances in the comparison must be of the identical type. If *order* is true and *eq* is false, a `ValueError` is raised.

  If the class already defines any of **`__lt__()`**, **`__le__()`**, **`__gt__()`**, or **`__ge__()`**, then `TypeError` is raised.

- *unsafe_hash*: If true, force `dataclasses` to create a `__hash__()` method, even though it may not be safe to do so. Otherwise, generate a `__hash__()` method according to how *eq* and *frozen* are set. The default value is `False`.

  `__hash__()` is used by built-in `hash()`, and when objects are added to hashed collections such as dictionaries and sets. Having a **`__hash__()`** implies that instances of the class are immutable. Mutability is a complicated property that depends on the programmer’s intent, the existence and behavior of **`__eq__()`**, and the values of the *eq* and *frozen* flags in the `@dataclass` decorator.

  By default, `@dataclass` will not implicitly add a `__hash__()` method unless it is safe to do so. Neither will it add or change an existing explicitly defined **`__hash__()`** method. Setting the class attribute `__hash__ = None` has a specific meaning to Python, as described in the **`__hash__()`** documentation.

  If **`__hash__()`** is not explicitly defined, or if it is set to `None`, then `@dataclass` *may* add an implicit **`__hash__()`** method. Although not recommended, you can force `@dataclass` to create a **`__hash__()`** method with `unsafe_hash=True`. This might be the case if your class is logically immutable but can still be mutated. This is a specialized use case and should be considered carefully.

Here are the rules governing implicit creation of a **`__hash__()`** method. Note that you cannot both have an explicit **`__hash__()`** method in your dataclass and set `unsafe_hash=True`; this will result in a **`TypeError`**.

If *eq* and *frozen* are both true, by default `@dataclass` will generate a **`__hash__()`** method for you. If *eq* is true and *frozen* is false, **`__hash__()`** will be set to `None`, marking it unhashable (which it is, since it is mutable). If *eq* is false, **`__hash__()`** will be left untouched meaning the **`__hash__()`** method of the superclass will be used (if the superclass is **`object`**, this means it will fall back to id-based hashing).

- *frozen*: If true (the default is `False`), assigning to fields will generate an exception. This emulates read-only frozen instances. See the discussion below.

  If **`__setattr__()`** or **`__delattr__()`** is defined in the class and *frozen* is true, then **`TypeError`** is raised.

- *match_args*: If true (the default is `True`), the **`__match_args__`** tuple will be created from the list of non keyword-only parameters to the generated **`__init__()`** method (even if **`__init__()`** is not generated, see above). If false, or if **`__match_args__`** is already defined in the class, then **`__match_args__`** will not be generated.

Added in version 3.10.

- *kw_only*: If true (the default value is `False`), then all fields will be marked as keyword-only. If a field is marked as keyword-only, then the only effect is that the `__init__()` parameter generated from a keyword-only field must be specified with a keyword when `__init__()` is called. See the parameter glossary entry for details. Also see the `KW_ONLY` section.

Keyword-only fields are not included in `__match_args__`.

Added in version 3.10.

- *slots*: If true (the default is `False`), `__slots__` attribute will be generated and new class will be returned instead of the original one. If `__slots__` is already defined in the class, then `TypeError` is raised.

Warning

Passing parameters to a base class `__init_subclass__()` when using `slots=True` will result in a `TypeError`. Either use `__init_subclass__` with no parameters or use default values as a workaround. See gh-91126 [https://github.com/python/cpython/issues/91126] for full details.

Added in version 3.10.

Changed in version 3.11: If a field name is already included in the `__slots__` of a base class, it will not be included in the generated `__slots__` to prevent overriding them. Therefore, do not use `__slots__` to retrieve the field names of a dataclass. Use `fields()` instead. To be able to determine inherited slots, base class `__slots__` may be any iterable, but *not* an iterator.

- *weakref_slot*: If true (the default is `False`), add a slot named "__weakref__", which is required to make an instance **weakref-able**. It is an error to specify `weakref_slot=True` without also specifying `slots=True`.

  Added in version 3.11.

`field`s may optionally specify a default value, using normal Python syntax:

```
@dataclass
class C:
    a: int       # 'a' has no default value
    b: int = 0   # assign a default value for 'b'
```

In this example, both **`a`** and **`b`** will be included in the added **`__init__()`** method, which will be defined as:

```
def __init__(self, a: int, b: int = 0):
```

`TypeError` will be raised if a field without a default value follows a field with a default value. This is true whether this occurs in a single class, or as a result of class inheritance.

dataclasses.field(*, *default=MISSING*, *default_factory=MISSING*, *init=True*, *repr=True*, *hash=None*, *compare=True*, *metadata=None*, *kw_only=MISSING*, *doc=None*)

For common and simple use cases, no other functionality is required. There are, however, some dataclass features that require additional per-field information. To satisfy this need for additional information, you can replace the default field value with a call to the provided **`field()`** function. For example:

```
@dataclass
class C:
    mylist: list[int] = field(default_factory=list)

c = C()
c.mylist += [1, 2, 3]
```

As shown above, the `MISSING` value is a sentinel object used to detect if some parameters are provided by the user. This sentinel is used because `None` is a valid value for some parameters with a distinct meaning. No code should directly use the `MISSING` value.

The parameters to **`field()`** are:

- *default*: If provided, this will be the default value for this field. This is needed because the **`field()`** call itself replaces the normal position of the default value.

- *default_factory*: If provided, it must be a zero-argument callable that will be called when a default value is needed for this field. Among other purposes, this can be used to specify fields with mutable default values, as discussed below. It is an error to specify both *default* and *default_factory*.

- *init*: If true (the default), this field is included as a parameter to the generated `__init__()` method.

- *repr*: If true (the default), this field is included in the string returned by the generated `__repr__()` method.

- *hash*: This can be a bool or `None`. If true, this field is included in the generated `__hash__()` method. If false, this field is excluded from the generated `__hash__()`. If `None` (the default), use the value of *compare*: this would normally be the expected behavior, since a field should be included in the hash if it’s used for comparisons. Setting this value to anything other than `None` is discouraged.

One possible reason to set `hash=False` but `compare=True` would be if a field is expensive to compute a hash value for, that field is needed for equality testing, and there are other fields that contribute to the type's hash value. Even if a field is excluded from the hash, it will still be used for comparisons.

- *compare*: If true (the default), this field is included in the generated equality and comparison methods (`__eq__()`, `__gt__()`, et al.).
- *metadata*: This can be a mapping or `None`. `None` is treated as an empty dict. This value is wrapped in `MappingProxyType()` to make it read-only, and exposed on the `Field` object. It is not used at all by Data Classes, and is provided as a third-party extension mechanism. Multiple third-parties can each have their own key, to use as a namespace in the metadata.
- *kw_only*: If true, this field will be marked as keyword-only. This is used when the generated `__init__()` method's parameters are computed.

  Keyword-only fields are also not included in **`__match_args__`**.

Added in version 3.10.

- *doc*: optional docstring for this field.

Added in version 3.14.

If the default value of a field is specified by a call to **`field()`**, then the class attribute for this field will be replaced by the specified *default* value. If *default* is not provided, then the class attribute will be deleted. The intent is that after the **`@dataclass`** decorator runs, the class attributes will all contain the default values for the fields, just as if the default value itself were specified. For example, after:

```
@dataclass
class C:
    x: int
    y: int = field(repr=False)
    z: int = field(repr=False, default=10)
    t: int = 20
```

The class attribute **`C.z`** will be `10`, the class attribute **`C.t`** will be `20`, and the class attributes **`C.x`** and **`C.y`** will not be set.

class dataclasses.Field

`Field` objects describe each defined field. These objects are created internally, and are returned by the **`fields()`** module-level method (see below). Users should never instantiate a **`Field`** object directly. Its documented attributes are:

- **`name`**: The name of the field.
- **`type`**: The type of the field.
- **`default`**, **`default_factory`**, **`init`**, **`repr`**, **`hash`**, **`compare`**, **`metadata`**, and **`kw_only`** have the identical meaning and values as they do in the **`field()`** function.

Other attributes may exist, but they are private and must not be inspected or relied on.

class dataclasses.InitVar

`InitVar[T]` type annotations describe variables that are init-only. Fields annotated with **`InitVar`** are considered pseudo-fields, and thus are neither returned by the `fields()` function nor used in any way except adding them as parameters to `__init__()` and an optional `__post_init__()`.

dataclasses.fields(*class_or_instance*)

Returns a tuple of `Field` objects that define the fields for this dataclass. Accepts either a dataclass, or an instance of a dataclass. Raises `TypeError` if not passed a dataclass or instance of one. Does not return pseudo-fields which are `ClassVar` or `InitVar`.

dataclasses.asdict(*obj*, *, *dict_factory=dict*)

Converts the dataclass *obj* to a dict (by using the factory function *dict_factory*). Each dataclass is converted to a dict of its fields, as `name: value` pairs. dataclasses, dicts, lists, and tuples are recursed into. Other objects are copied with `copy.deepcopy()`.

Example of using **`asdict()`** on nested dataclasses:

```
@dataclass
class Point:
     x: int
     y: int

@dataclass
class C:
     mylist: list[Point]

p = Point(10, 20)
assert asdict(p) == {'x': 10, 'y': 20}

c = C([Point(0, 0), Point(10, 4)])
assert asdict(c) == {'mylist': [{'x': 0, 'y': 0}, {'x': 10,
'y': 4}]}
```

To create a shallow copy, the following workaround may be used:

```
{field.name: getattr(obj, field.name) for field in
fields(obj)}
```

`asdict()` raises `TypeError` if *obj* is not a dataclass instance.

dataclasses.astuple(*obj*, ***, *tuple_factory=tuple*)

Converts the dataclass *obj* to a tuple (by using the factory function *tuple_factory*). Each dataclass is converted to a tuple of its field values. dataclasses, dicts, lists, and tuples are recursed into. Other objects are copied with `copy.deepcopy()`.

Continuing from the previous example:

```
assert astuple(p) == (10, 20)
assert astuple(c) == ([(0, 0), (10, 4)],)
```

To create a shallow copy, the following workaround may be used:

```
tuple(getattr(obj, field.name) for field in
dataclasses.fields(obj))
```

`astuple()` raises `TypeError` if *obj* is not a dataclass instance.

dataclasses.make_dataclass(*cls_name*, *fields*, ***, *bases=()*, *namespace=None*, *init=True*, *repr=True*, *eq=True*, *order=False*, *unsafe_hash=False*, *frozen=False*, *match_args=True*, *kw_only=False*, *slots=False*, *weakref_slot=False*, *module=None*, *decorator=dataclass*)

Creates a new dataclass with name *cls_name*, fields as defined in *fields*, base classes as given in *bases*, and initialized with a namespace as given in *namespace*. *fields* is an iterable whose elements are each either `name`, `(name, type)`, or `(name, type, Field)`. If just `name` is supplied, `typing.Any` is used for `type`. The values of *init*, *repr*, *eq*, *order*, *unsafe_hash*, *frozen*, *match_args*, *kw_only*, *slots*, and *weakref_slot* have the same meaning as they do in `@dataclass`.

If *module* is defined, the **`__module__`** attribute of the dataclass is set to that value. By default, it is set to the module name of the caller.

The *decorator* parameter is a callable that will be used to create the dataclass. It should take the class object as a first argument and the same keyword arguments as `@dataclass`. By default, the `@dataclass` function is used.

This function is not strictly required, because any Python mechanism for creating a new class with `__annotations__` can then apply the `@dataclass` function to convert that class to a dataclass. This function is provided as a convenience. For example:

```
C = make_dataclass('C',
                   [('x', int),
                     'y',
                    ('z', int, field(default=5))],
                   namespace={'add_one': lambda self: self.x
+ 1})
```

Is equivalent to:

```
@dataclass
class C:
    x: int
    y: 'typing.Any'
    z: int = 5

    def add_one(self):
        return self.x + 1
```

Added in version 3.14: Added the *decorator* parameter.

dataclasses.replace(*obj*, */*, ***changes*)

Creates a new object of the same type as *obj*, replacing fields with values from *changes*. If *obj* is not a Data Class, raises `TypeError`. If keys in *changes* are not field names of the given dataclass, raises `TypeError`.

The newly returned object is created by calling the `__init__()` method of the dataclass. This ensures that `__post_init__()`, if present, is also called.

Init-only variables without default values, if any exist, must be specified on the call to **`replace()`** so that they can be passed to **`__init__()`** and `__post_init__()`.

It is an error for *changes* to contain any fields that are defined as having `init=False`. A `ValueError` will be raised in this case.

Be forewarned about how `init=False` fields work during a call to **`replace()`**. They are not copied from the source object, but rather are initialized in `__post_init__()`, if they’re initialized at all. It is expected that `init=False` fields will be rarely and judiciously used. If they are used, it might be wise to have alternate class constructors, or perhaps a custom **`replace()`** (or similarly named) method which handles instance copying.

Dataclass instances are also supported by generic function `copy.replace()`.

dataclasses.is_dataclass(*obj*)

Return `True` if its parameter is a dataclass (including subclasses of a dataclass) or an instance of one, otherwise return `False`.

If you need to know if a class is an instance of a dataclass (and not a dataclass itself), then add a further check for `not isinstance(obj, type)`:

```
def is_dataclass_instance(obj):
    return is_dataclass(obj) and not isinstance(obj, type)
```

dataclasses.MISSING

A sentinel value signifying a missing default or default_factory.

dataclasses.KW_ONLY

A sentinel value used as a type annotation. Any fields after a pseudo-field with the type of **`KW_ONLY`** are marked as keyword-only fields. Note that a pseudo-field of type **`KW_ONLY`** is otherwise completely ignored. This includes the name of such a field. By convention, a name of `_` is used for a **`KW_ONLY`** field. Keyword-only fields signify **`__init__()`** parameters that must be specified as keywords when the class is instantiated.

In this example, the fields `y` and `z` will be marked as keyword-only fields:

```
@dataclass
class Point:
    x: float
    _: KW_ONLY
    y: float
    z: float

p = Point(0, y=1.5, z=2.0)
```

In a single dataclass, it is an error to specify more than one field whose type is **`KW_ONLY`**.

Added in version 3.10.

exception dataclasses.FrozenInstanceError

Raised when an implicitly defined **`__setattr__()`** or **`__delattr__()`** is called on a dataclass which was defined with `frozen=True`. It is a subclass of **`AttributeError`**.

Post-init processing

dataclasses.__post_init__()

When defined on the class, it will be called by the generated **`__init__()`**, normally as **`self.__post_init__()`**. However, if any `InitVar` fields are defined, they will also be passed to **`__post_init__()`** in the order they were defined in the class. If no **`__init__()`** method is generated, then **`__post_init__()`** will not automatically be called.

Among other uses, this allows for initializing field values that depend on one or more other fields. For example:

```
@dataclass
class C:
    a: float
    b: float
    c: float = field(init=False)

    def __post_init__(self):
        self.c = self.a + self.b
```

The **`__init__()`** method generated by **`@dataclass`** does not call base class **`__init__()`** methods. If the base class has an **`__init__()`** method that has to be called, it is common to call this method in a **`__post_init__()`** method:

```
class Rectangle:
    def __init__(self, height, width):
        self.height = height
        self.width = width

@dataclass
class Square(Rectangle):
    side: float

    def __post_init__(self):
        super().__init__(self.side, self.side)
```

Note, however, that in general the dataclass-generated **`__init__()`** methods don't need to be called, since the derived dataclass will take care of initializing all fields of any base class that is a dataclass itself.

See the section below on init-only variables for ways to pass parameters to **`__post_init__()`**. Also see the warning about how **`replace()`** handles `init=False` fields.

Class variables

One of the few places where **`@dataclass`** actually inspects the type of a field is to determine if a field is a class variable as defined in **PEP 526** [https://peps.python.org/pep-0526/]. It does this by checking if the type of the field is **`typing.ClassVar`**. If a field is a `ClassVar`, it is excluded from consideration as a field and is ignored by the dataclass mechanisms. Such `ClassVar` pseudo-fields are not returned by the module-level **`fields()`** function.

Init-only variables

Another place where **`@dataclass`** inspects a type annotation is to determine if a field is an init-only variable. It does this by seeing if the type of a field is of type **`InitVar`**. If a field is an **`InitVar`**, it is considered a pseudo-field called an init-only field. As it is not a true field, it is not returned by the module-level **`fields()`** function. Init-only fields are added as parameters to the generated **`__init__()`** method, and are passed to the optional **`__post_init__()`** method. They are not otherwise used by dataclasses.

For example, suppose a field will be initialized from a database, if a value is not provided when creating the class:

```
@dataclass
class C:
    i: int
    j: int | None = None
    database: InitVar[DatabaseType | None] = None

    def __post_init__(self, database):
        if self.j is None and database is not None:
            self.j = database.lookup('j')

c = C(10, database=my_database)
```

In this case, `fields()` will return `Field` objects for **`i`** and **`j`**, but not for **`database`**.

Frozen instances

It is not possible to create truly immutable Python objects. However, by passing `frozen=True` to the `@dataclass` decorator you can emulate immutability. In that case, dataclasses will add `__setattr__()` and `__delattr__()` methods to the class. These methods will raise a `FrozenInstanceError` when invoked.

There is a tiny performance penalty when using `frozen=True`: `__init__()` cannot use simple assignment to initialize fields, and must use **`object.__setattr__()`**.

Inheritance

When the dataclass is being created by the `@dataclass` decorator, it looks through all of the class's base classes in reverse MRO (that is, starting at `object`) and, for each dataclass that it finds, adds the fields from that base class to an ordered mapping of fields. After all of the base class fields are added, it adds its own fields to the ordered mapping. All of the generated methods will use this combined, calculated ordered mapping of fields. Because the fields are in insertion order, derived classes override base classes. An example:

```
@dataclass
class Base:
    x: Any = 15.0
    y: int = 0

@dataclass
class C(Base):
    z: int = 10
    x: int = 15
```

The final list of fields is, in order, **`x`**, **`y`**, **`z`**. The final type of **`x`** is [**`int`**](), as specified in class **`C`**.

The generated [**`__init__()`**]() method for **`C`** will look like:

```
def __init__(self, x: int = 15, y: int = 0, z: int = 10):
```

Re-ordering of keyword-only parameters in `__init__()`

After the parameters needed for [**`__init__()`**]() are computed, any keyword-only parameters are moved to come after all regular (non-keyword-only) parameters. This is a requirement of how keyword-only parameters are implemented in Python: they must come after non-keyword-only parameters.

In this example, **`Base.y`**, **`Base.w`**, and **`D.t`** are keyword-only fields, and **`Base.x`** and **`D.z`** are regular fields:

```
@dataclass
class Base:
    x: Any = 15.0
    _: KW_ONLY
    y: int = 0
    w: int = 1

@dataclass
class D(Base):
    z: int = 10
    t: int = field(kw_only=True, default=0)
```

The generated `__init__()` method for `D` will look like:

```
def __init__(self, x: Any = 15.0, z: int = 10, *, y: int = 0,
w: int = 1, t: int = 0):
```

Note that the parameters have been re-ordered from how they appear in the list of fields: parameters derived from regular fields are followed by parameters derived from keyword-only fields.

The relative ordering of keyword-only parameters is maintained in the re-ordered `__init__()` parameter list.

Default factory functions

If a `field()` specifies a *default_factory*, it is called with zero arguments when a default value for the field is needed. For example, to create a new instance of a list, use:

```
mylist: list = field(default_factory=list)
```

If a field is excluded from `__init__()` (using `init=False`) and the field also specifies *default_factory*, then the default factory function will always be called from the generated `__init__()` function. This happens because there is no other way to give the field an initial value.

Mutable default values

Python stores default member variable values in class attributes. Consider this example, not using dataclasses:

```
class C:
    x = []
    def add(self, element):
        self.x.append(element)

o1 = C()
o2 = C()
o1.add(1)
o2.add(2)
assert o1.x == [1, 2]
assert o1.x is o2.x
```

Note that the two instances of class **`C`** share the same class variable **`x`**, as expected.

Using dataclasses, *if* this code was valid:

```
@dataclass
class D:
    x: list = []      # This code raises ValueError
    def add(self, element):
        self.x.append(element)
```

it would generate code similar to:

```
class D:
    x = []
    def __init__(self, x=x):
        self.x = x
    def add(self, element):
        self.x.append(element)

assert D().x is D().x
```

This has the same issue as the original example using class **`C`**. That is, two instances of class **`D`** that do not specify a value for **`x`** when creating a class instance will share the same copy of **`x`**. Because dataclasses just use normal Python class creation they also share this behavior. There is no general way for Data Classes to detect this condition. Instead, the `@dataclass` decorator will raise a `ValueError` if it detects an unhashable default parameter. The assumption is that if a value is unhashable, it is mutable. This is a partial solution, but it does protect against many common errors.

Using default factory functions is a way to create new instances of mutable types as default values for fields:

```
@dataclass
class D:
    x: list = field(default_factory=list)

assert D().x is not D().x
```

Changed in version 3.11: Instead of looking for and disallowing objects of type `list`, `dict`, or `set`, unhashable objects are now not allowed as default values. Unhashability is used to approximate mutability.

Descriptor-typed fields

Fields that are assigned descriptor objects as their default value have the following special behaviors:

- The value for the field passed to the dataclass's `__init__()` method is passed to the descriptor's `__set__()` method rather than overwriting the descriptor object.
- Similarly, when getting or setting the field, the descriptor's `__get__()` or **`__set__()`** method is called rather than returning or overwriting the descriptor object.
- To determine whether a field contains a default value, `@dataclass` will call the descriptor's **`__get__()`** method using its class access form: `descriptor.__get__(obj=None, type=cls)`. If the descriptor returns a value in this case, it will be used as the field's default. On the other hand, if

the descriptor raises `AttributeError` in this situation, no default value will be provided for the field.

```
class IntConversionDescriptor:
    def __init__(self, *, default):
        self._default = default

    def __set_name__(self, owner, name):
        self._name = "_" + name

    def __get__(self, obj, type):
        if obj is None:
            return self._default

        return getattr(obj, self._name, self._default)

    def __set__(self, obj, value):
        setattr(obj, self._name, int(value))

@dataclass
class InventoryItem:
    quantity_on_hand: IntConversionDescriptor =
IntConversionDescriptor(default=100)

i = InventoryItem()
print(i.quantity_on_hand)   # 100
i.quantity_on_hand = 2.5    # calls __set__ with 2.5
print(i.quantity_on_hand)   # 2
```

Note that if a field is annotated with a descriptor type, but is not assigned a descriptor object as its default value, the field will act like a normal field.

`contextlib` — Utilities for `with`-statement contexts

Source code: Lib/contextlib.py
[https://github.com/python/cpython/tree/3.14/Lib/contextlib.py]

This module provides utilities for common tasks involving the `with` statement. For more information see also Context Manager Types and With Statement Context Managers.

Utilities

Functions and classes provided:

class contextlib.AbstractContextManager

An abstract base class for classes that implement `object.__enter__()` and `object.__exit__()`. A default implementation for `object.__enter__()` is provided which returns `self` while `object.__exit__()` is an abstract method which by default returns `None`. See also the definition of Context Manager Types.

Added in version 3.6.

class contextlib.AbstractAsyncContextManager

An abstract base class for classes that implement `object.__aenter__()` and `object.__aexit__()`. A default implementation for `object.__aenter__()` is provided which returns `self` while `object.__aexit__()` is an abstract method which by default returns `None`. See also the definition of Asynchronous Context Managers.

Added in version 3.7.

@contextlib.contextmanager

This function is a decorator that can be used to define a factory function for **`with`** statement context managers, without needing to create a class or separate **`__enter__()`** and **`__exit__()`** methods.

While many objects natively support use in with statements, sometimes a resource needs to be managed that isn't a context manager in its own right, and doesn't implement a `close()` method for use with `contextlib.closing`.

An abstract example would be the following to ensure correct resource management:

```
from contextlib import contextmanager

@contextmanager
def managed_resource(*args, **kwds):
    # Code to acquire resource, e.g.:
    resource = acquire_resource(*args, **kwds)
    try:
        yield resource
    finally:
        # Code to release resource, e.g.:
        release_resource(resource)
```

The function can then be used like this:

```
>>> with managed_resource(timeout=3600) as resource:
...     # Resource is released at the end of this block,
...     # even if code in the block raises an exception
```

The function being decorated must return a generator-iterator when called. This iterator must yield exactly one value, which will be bound to the targets in the **`with`** statement's **`as`** clause, if any.

At the point where the generator yields, the block nested in the `with` statement is executed. The generator is then resumed after the block is exited. If an unhandled exception occurs in the block, it is reraised inside the generator at the point where the yield occurred. Thus, you can use a `try`…`except`…`finally` statement to trap the error (if any), or ensure that some cleanup takes place. If an exception is trapped merely in order to log it or to perform some action (rather than to suppress it entirely), the generator must reraise that exception. Otherwise the generator context manager will indicate to the **`with`** statement that the exception has been handled, and execution will resume with the statement immediately following the **`with`** statement.

`contextmanager()` uses `ContextDecorator` so the context managers it creates can be used as decorators as well as in `with` statements. When used as a decorator, a new generator instance is implicitly created on each function call (this allows the otherwise “one-shot” context managers created by `contextmanager()` to meet the requirement that context managers support multiple invocations in order to be used as decorators).

Changed in version 3.2: Use of `ContextDecorator`.

@contextlib.asynccontextmanager

Similar to `contextmanager()`, but creates an asynchronous context manager.

This function is a decorator that can be used to define a factory function for `async with` statement asynchronous context managers, without needing to create a class or separate `__aenter__()` and `__aexit__()` methods. It must be applied to an asynchronous generator function.

A simple example:

```
from contextlib import asynccontextmanager

@asynccontextmanager
async def get_connection():
    conn = await acquire_db_connection()
    try:
        yield conn
    finally:
        await release_db_connection(conn)

async def get_all_users():
    async with get_connection() as conn:
        return conn.query('SELECT ...')
```

Added in version 3.7.

Context managers defined with `asynccontextmanager()` can be used either as decorators or with `async with` statements:

```
import time
from contextlib import asynccontextmanager

@asynccontextmanager
async def timeit():
    now = time.monotonic()
    try:
        yield
    finally:
        print(f'it took {time.monotonic() - now}s to run')

@timeit()
async def main():
    # ... async code ...
```

When used as a decorator, a new generator instance is implicitly created on each function call. This allows the otherwise “one-shot” context managers created by `asynccontextmanager()` to meet the requirement that context managers support multiple invocations in order to be used as decorators.

Changed in version 3.10: Async context managers created with `asynccontextmanager()` can be used as decorators.

contextlib.closing(*thing*)

Return a context manager that closes *thing* upon completion of the block. This is basically equivalent to:

```
from contextlib import contextmanager

@contextmanager
def closing(thing):
    try:
        yield thing
    finally:
        thing.close()
```

And lets you write code like this:

```
from contextlib import closing
from urllib.request import urlopen

with closing(urlopen('https://www.python.org')) as page:
    for line in page:
        print(line)
```

without needing to explicitly close `page`. Even if an error occurs, `page.close()` will be called when the `with` block is exited.

Note

Most types managing resources support the context manager protocol, which closes *thing* on leaving the `with` statement. As such, **`closing()`** is most useful for third party types that don’t support context managers.

This example is purely for illustration purposes, as `urlopen()` would normally be used in a context manager.

contextlib.aclosing(*thing*)

Return an async context manager that calls the `aclose()` method of *thing* upon completion of the block. This is basically equivalent to:

```
from contextlib import asynccontextmanager

@asynccontextmanager
async def aclosing(thing):
    try:
        yield thing
    finally:
        await thing.aclose()
```

Significantly, `aclosing()` supports deterministic cleanup of async generators when they happen to exit early by `break` or an exception. For example:

```
from contextlib import aclosing

async with aclosing(my_generator()) as values:
    async for value in values:
        if value == 42:
            break
```

This pattern ensures that the generator's async exit code is executed in the same context as its iterations (so that exceptions and context variables work as expected, and the exit code isn't run after the lifetime of some task it depends on).

Added in version 3.10.

contextlib.nullcontext(*enter_result=None*)

Return a context manager that returns *enter_result* from `__enter__`, but otherwise does nothing. It is intended to be used as a stand-in for an optional context manager, for example:

```
def myfunction(arg, ignore_exceptions=False):
    if ignore_exceptions:
        # Use suppress to ignore all exceptions.
        cm = contextlib.suppress(Exception)
    else:
        # Do not ignore any exceptions, cm has no effect.
        cm = contextlib.nullcontext()
    with cm:
        # Do something
```

An example using *enter_result*:

```
def process_file(file_or_path):
    if isinstance(file_or_path, str):
        # If string, open file
        cm = open(file_or_path)
    else:
        # Caller is responsible for closing file
        cm = nullcontext(file_or_path)

    with cm as file:
        # Perform processing on the file
```

It can also be used as a stand-in for asynchronous context managers:

```
async def send_http(session=None):
    if not session:
        # If no http session, create it with aiohttp
        cm = aiohttp.ClientSession()
    else:
        # Caller is responsible for closing the session
        cm = nullcontext(session)

    async with cm as session:
        # Send http requests with session
```

Added in version 3.7.

Changed in version 3.10: asynchronous context manager support was added.

contextlib.suppress(**exceptions*)

Return a context manager that suppresses any of the specified exceptions if they occur in the body of a **`with`** statement and then resumes execution with the first statement following the end of the **`with`** statement.

As with any other mechanism that completely suppresses exceptions, this context manager should be used only to cover very specific errors where silently continuing with program execution is known to be the right thing to do.

For example:

```
from contextlib import suppress

with suppress(FileNotFoundError):
    os.remove('somefile.tmp')

with suppress(FileNotFoundError):
    os.remove('someotherfile.tmp')
```

This code is equivalent to:

```
try:
    os.remove('somefile.tmp')
except FileNotFoundError:
    pass

try:
    os.remove('someotherfile.tmp')
except FileNotFoundError:
    pass
```

This context manager is reentrant.

If the code within the **`with`** block raises a `BaseExceptionGroup`, suppressed exceptions are removed from the group. Any exceptions of the group which are not suppressed are re-raised in a new group which is created using the original group's `derive()` method.

Added in version 3.4.

Changed in version 3.12: `suppress` now supports suppressing exceptions raised as part of a `BaseExceptionGroup`.

contextlib.redirect_stdout(*new_target*)

Context manager for temporarily redirecting `sys.stdout` to another file or file-like object.

This tool adds flexibility to existing functions or classes whose output is hardwired to stdout.

For example, the output of `help()` normally is sent to *sys.stdout*. You can capture that output in a string by redirecting the output to an `io.StringIO` object. The replacement stream is returned from the `__enter__` method and so is available as the target of the `with` statement:

```
with redirect_stdout(io.StringIO()) as f:
    help(pow)
s = f.getvalue()
```

To send the output of `help()` to a file on disk, redirect the output to a regular file:

```
with open('help.txt', 'w') as f:
    with redirect_stdout(f):
        help(pow)
```

To send the output of `help()` to *sys.stderr*:

```
with redirect_stdout(sys.stderr):
    help(pow)
```

Note that the global side effect on **`sys.stdout`** means that this context manager is not suitable for use in library code and most threaded applications. It also has no effect on the output of subprocesses. However, it is still a useful approach for many utility scripts.

This context manager is reentrant.

Added in version 3.4.

contextlib.redirect_stderr(*new_target*)

Similar to **`redirect_stdout()`** but redirecting **`sys.stderr`** to another file or file-like object.

This context manager is reentrant.

Added in version 3.5.

contextlib.chdir(*path*)

Non parallel-safe context manager to change the current working directory. As this changes a global state, the working directory, it is not suitable for use in most threaded or async contexts. It is also not suitable for most non-linear code execution, like generators, where the program execution is temporarily relinquished – unless explicitly desired, you should not yield when this context manager is active.

This is a simple wrapper around **`chdir()`**, it changes the current working directory upon entering and restores the old one on exit.

This context manager is reentrant.

Added in version 3.11.

class contextlib.ContextDecorator

A base class that enables a context manager to also be used as a decorator.

Context managers inheriting from `ContextDecorator` have to implement `__enter__` and `__exit__` as normal. `__exit__` retains its optional exception handling even when used as a decorator.

`ContextDecorator` is used by **`contextmanager()`**, so you get this functionality automatically.

Example of `ContextDecorator`:

```
from contextlib import ContextDecorator

class mycontext(ContextDecorator):
    def __enter__(self):
        print('Starting')
        return self

    def __exit__(self, *exc):
        print('Finishing')
        return False
```

The class can then be used like this:

```
>>> @mycontext()
... def function():
...     print('The bit in the middle')
...
>>> function()
Starting
The bit in the middle
Finishing

>>> with mycontext():
...     print('The bit in the middle')
```

```
...
Starting
The bit in the middle
Finishing
```

This change is just syntactic sugar for any construct of the following form:

```
def f():
    with cm():
        # Do stuff
```

`ContextDecorator` lets you instead write:

```
@cm()
def f():
    # Do stuff
```

It makes it clear that the `cm` applies to the whole function, rather than just a piece of it (and saving an indentation level is nice, too).

Existing context managers that already have a base class can be extended by using `ContextDecorator` as a mixin class:

```
from contextlib import ContextDecorator

class mycontext(ContextBaseClass, ContextDecorator):
    def __enter__(self):
        return self

    def __exit__(self, *exc):
        return False
```

Note

As the decorated function must be able to be called multiple times, the underlying context manager must support use in multiple `with` statements. If this is not the case, then the original construct with the explicit **`with`** statement inside the function should be used.

Added in version 3.2.

class contextlib.AsyncContextDecorator

Similar to [`ContextDecorator`] but only for asynchronous functions.

Example of `AsyncContextDecorator`:

```
from asyncio import run
from contextlib import AsyncContextDecorator

class mycontext(AsyncContextDecorator):
    async def __aenter__(self):
        print('Starting')
        return self

    async def __aexit__(self, *exc):
        print('Finishing')
        return False
```

The class can then be used like this:

```
>>> @mycontext()
... async def function():
...     print('The bit in the middle')
...
>>> run(function())
Starting
The bit in the middle
Finishing

>>> async def function():
...    async with mycontext():
...         print('The bit in the middle')
...
>>> run(function())
Starting
```

```
The bit in the middle
Finishing
```

Added in version 3.10.

class contextlib.ExitStack

A context manager that is designed to make it easy to programmatically combine other context managers and cleanup functions, especially those that are optional or otherwise driven by input data.

For example, a set of files may easily be handled in a single with statement as follows:

```
with ExitStack() as stack:
    files = [stack.enter_context(open(fname)) for fname in
filenames]
    # All opened files will automatically be closed at the
end of
    # the with statement, even if attempts to open files
later
    # in the list raise an exception
```

The `__enter__()` method returns the `ExitStack` instance, and performs no additional operations.

Each instance maintains a stack of registered callbacks that are called in reverse order when the instance is closed (either explicitly or implicitly at the end of a `with` statement). Note that callbacks are *not* invoked implicitly when the context stack instance is garbage collected.

This stack model is used so that context managers that acquire their resources in their `__init__` method (such as file objects) can be handled correctly.

Since registered callbacks are invoked in the reverse order of registration, this ends up behaving as if multiple nested `with` statements had been used with the registered set of callbacks. This even extends to exception handling - if an inner callback suppresses or replaces an exception, then outer callbacks will be passed arguments based on that updated state.

This is a relatively low level API that takes care of the details of correctly unwinding the stack of exit callbacks. It provides a suitable foundation for higher level context managers that manipulate the exit stack in application specific ways.

Added in version 3.3.

enter_context(*cm*)

Enters a new context manager and adds its `__exit__()` method to the callback stack. The return value is the result of the context manager's own `__enter__()` method.

These context managers may suppress exceptions just as they normally would if used directly as part of a `with` statement.

Changed in version 3.11: Raises `TypeError` instead of `AttributeError` if *cm* is not a context manager.

push(*exit*)

Adds a context manager's `__exit__()` method to the callback stack.

As `__enter__` is *not* invoked, this method can be used to cover part of an `__enter__()` implementation with a context manager's own `__exit__()` method.

If passed an object that is not a context manager, this method assumes it is a callback with the same signature as a context manager’s `__exit__()` method and adds it directly to the callback stack.

By returning true values, these callbacks can suppress exceptions the same way context manager `__exit__()` methods can.

The passed in object is returned from the function, allowing this method to be used as a function decorator.

callback(*callback*, */*, **args*, ***kwds*)

Accepts an arbitrary callback function and arguments and adds it to the callback stack.

Unlike the other methods, callbacks added this way cannot suppress exceptions (as they are never passed the exception details).

The passed in callback is returned from the function, allowing this method to be used as a function decorator.

pop_all()

Transfers the callback stack to a fresh `ExitStack` instance and returns it. No callbacks are invoked by this operation - instead, they will now be invoked when the new stack is closed (either explicitly or implicitly at the end of a `with` statement).

For example, a group of files can be opened as an “all or nothing” operation as follows:

```
with ExitStack() as stack:
    files = [stack.enter_context(open(fname)) for fname
in filenames]
    # Hold onto the close method, but don't call it yet.
    close_files = stack.pop_all().close
    # If opening any file fails, all previously opened
files will be
    # closed automatically. If all files are opened
successfully,
    # they will remain open even after the with statement
ends.
    # close_files() can then be invoked explicitly to
close them all.
```

close()

Immediately unwinds the callback stack, invoking callbacks in the reverse order of registration. For any context managers and exit callbacks registered, the arguments passed in will indicate that no exception occurred.

class contextlib.AsyncExitStack

An asynchronous context manager, similar to `ExitStack`, that supports combining both synchronous and asynchronous context managers, as well as having coroutines for cleanup logic.

The `close()` method is not implemented; `aclose()` must be used instead.

async enter_async_context(*cm*)

Similar to `ExitStack.enter_context()` but expects an asynchronous context manager.

Changed in version 3.11: Raises `TypeError` instead of `AttributeError` if *cm* is not an asynchronous context manager.

push_async_exit(*exit*)

Similar to `ExitStack.push()` but expects either an asynchronous context manager or a coroutine function.

push_async_callback(*callback*, */*, **args*, ***kwds*)

Similar to `ExitStack.callback()` but expects a coroutine function.

async aclose()

Similar to `ExitStack.close()` but properly handles awaitables.

Continuing the example for `asynccontextmanager()`:

```
async with AsyncExitStack() as stack:
    connections = [await
stack.enter_async_context(get_connection())
        for i in range(5)]
    # All opened connections will automatically be released
at the end of
    # the async with statement, even if attempts to open a
connection
    # later in the list raise an exception.
```

Added in version 3.7.

Examples and Recipes

This section describes some examples and recipes for making effective use of the tools provided by `contextlib`.

Supporting a variable number of context managers

The primary use case for `ExitStack` is the one given in the class documentation: supporting a variable number of context managers and other cleanup operations in a single `with` statement. The variability may come from the number of context managers needed being driven by user input (such as opening a user specified collection of files), or from some of the context managers being optional:

```
with ExitStack() as stack:
    for resource in resources:
        stack.enter_context(resource)
    if need_special_resource():
        special = acquire_special_resource()
        stack.callback(release_special_resource, special)
    # Perform operations that use the acquired resources
```

As shown, **`ExitStack`** also makes it quite easy to use **`with`** statements to manage arbitrary resources that don’t natively support the context management protocol.

Catching exceptions from `__enter__` methods

It is occasionally desirable to catch exceptions from an `__enter__` method implementation, *without* inadvertently catching exceptions from the **`with`** statement body or the context manager’s `__exit__` method. By using **`ExitStack`** the steps in the context management protocol can be separated slightly in order to allow this:

```
stack = ExitStack()
try:
    x = stack.enter_context(cm)
except Exception:
    # handle __enter__ exception
else:
    with stack:
        # Handle normal case
```

Actually needing to do this is likely to indicate that the underlying API should be providing a direct resource management interface for use with **`try`**/**`except`**/**`finally`** statements, but not all APIs are well designed in that regard. When a context manager is the only resource management API provided, then **`ExitStack`** can make it easier to handle various situations that can’t be handled directly in a `with` statement.

Cleaning up in an `__enter__` implementation

As noted in the documentation of `ExitStack.push()`, this method can be useful in cleaning up an already allocated resource if later steps in the `__enter__()` implementation fail.

Here’s an example of doing this for a context manager that accepts resource acquisition and release functions, along with an optional validation function, and maps them to the context management protocol:

```
from contextlib import contextmanager, AbstractContextManager,
ExitStack

class ResourceManager(AbstractContextManager):

    def __init__(self, acquire_resource, release_resource,
check_resource_ok=None):
        self.acquire_resource = acquire_resource
        self.release_resource = release_resource
        if check_resource_ok is None:
            def check_resource_ok(resource):
                return True
        self.check_resource_ok = check_resource_ok

    @contextmanager
    def _cleanup_on_error(self):
        with ExitStack() as stack:
            stack.push(self)
            yield
            # The validation check passed and didn't raise an
exception
            # Accordingly, we want to keep the resource, and
pass it
            # back to our caller
            stack.pop_all()
    def __enter__(self):
        resource = self.acquire_resource()
        with self._cleanup_on_error():
            if not self.check_resource_ok(resource):
                msg = "Failed validation for {!r}"
                raise RuntimeError(msg.format(resource))
        return resource

    def __exit__(self, *exc_details):
        # We don't need to duplicate any of our resource
release logic
        self.release_resource()
```

Replacing any use of `try-finally` and flag variables

A pattern you will sometimes see is a `try-finally` statement with a flag variable to indicate whether or not the body of the `finally` clause should be executed. In its simplest form (that can't already be handled just by using an `except` clause instead), it looks something like this:

```
cleanup_needed = True
try:
    result = perform_operation()
    if result:
        cleanup_needed = False
finally:
    if cleanup_needed:
        cleanup_resources()
```

As with any `try` statement based code, this can cause problems for development and review, because the setup code and the cleanup code can end up being separated by arbitrarily long sections of code.

`ExitStack` makes it possible to instead register a callback for execution at the end of a `with` statement, and then later decide to skip executing that callback:

```
from contextlib import ExitStack

with ExitStack() as stack:
    stack.callback(cleanup_resources)
    result = perform_operation()
    if result:
        stack.pop_all()
```

This allows the intended cleanup behaviour to be made explicit up front, rather than requiring a separate flag variable.

If a particular application uses this pattern a lot, it can be simplified even further by means of a small helper class:

```
from contextlib import ExitStack

class Callback(ExitStack):
    def __init__(self, callback, /, *args, **kwds):
        super().__init__()
        self.callback(callback, *args, **kwds)

    def cancel(self):
        self.pop_all()

with Callback(cleanup_resources) as cb:
    result = perform_operation()
    if result:
        cb.cancel()
```

If the resource cleanup isn't already neatly bundled into a standalone function, then it is still possible to use the decorator form of `ExitStack.callback()` to declare the resource cleanup in advance:

```
from contextlib import ExitStack

with ExitStack() as stack:
    @stack.callback
    def cleanup_resources():
        ...
    result = perform_operation()
    if result:
        stack.pop_all()
```

Due to the way the decorator protocol works, a callback function declared this way cannot take any parameters. Instead, any resources to be released must be accessed as closure variables.

Using a context manager as a function decorator

`ContextDecorator` makes it possible to use a context manager in both an ordinary `with` statement and also as a function decorator.

For example, it is sometimes useful to wrap functions or groups of statements with a logger that can track the time of entry and time of exit. Rather than writing both a function decorator and a context manager for the task, inheriting from **`ContextDecorator`** provides both capabilities in a single definition:

```
from contextlib import ContextDecorator
import logging

logging.basicConfig(level=logging.INFO)

class track_entry_and_exit(ContextDecorator):
    def __init__(self, name):
        self.name = name

    def __enter__(self):
        logging.info('Entering: %s', self.name)

    def __exit__(self, exc_type, exc, exc_tb):
        logging.info('Exiting: %s', self.name)
```

Instances of this class can be used as both a context manager:

```
with track_entry_and_exit('widget loader'):
    print('Some time consuming activity goes here')
    load_widget()
```

And also as a function decorator:

```
@track_entry_and_exit('widget loader')
def activity():
    print('Some time consuming activity goes here')
    load_widget()
```

Note that there is one additional limitation when using context managers as function decorators: there’s no way to access the return value of `__enter__()`. If that value is needed, then it is still necessary to use an explicit `with` statement.

See also

PEP 343 [https://peps.python.org/pep-0343/] - The “with” statement
The specification, background, and examples for the Python `with` statement.

Single use, reusable and reentrant context managers

Most context managers are written in a way that means they can only be used effectively in a `with` statement once. These single use context managers must be created afresh each time they’re used - attempting to use them a second time will trigger an exception or otherwise not work correctly.

This common limitation means that it is generally advisable to create context managers directly in the header of the `with` statement where they are used (as shown in all of the usage examples above).

Files are an example of effectively single use context managers, since the first `with` statement will close the file, preventing any further IO operations using that file object.

Context managers created using `contextmanager()` are also single use context managers, and will complain about the underlying generator failing to yield if an attempt is made to use them a second time:

```
>>> from contextlib import contextmanager
>>> @contextmanager
... def singleuse():
...     print("Before")
...     yield
...     print("After")
...
>>> cm = singleuse()
>>> with cm:
...     pass
...
Before
After
>>> with cm:
...     pass
...
Traceback (most recent call last):
    ...
RuntimeError: generator didn't yield
```

Reentrant context managers

More sophisticated context managers may be “reentrant”. These context managers can not only be used in multiple `with` statements, but may also be used *inside* a **`with`** statement that is already using the same context manager.

`threading.RLock` is an example of a reentrant context manager, as are `suppress()`, `redirect_stdout()`, and `chdir()`. Here’s a very simple example of reentrant use:

```
>>> from contextlib import redirect_stdout
>>> from io import StringIO
>>> stream = StringIO()
>>> write_to_stream = redirect_stdout(stream)
>>> with write_to_stream:
...     print("This is written to the stream rather than 
stdout")
...     with write_to_stream:
...         print("This is also written to the stream")
...
>>> print("This is written directly to stdout")
This is written directly to stdout
>>> print(stream.getvalue())
This is written to the stream rather than stdout
This is also written to the stream
```

Real world examples of reentrancy are more likely to involve multiple functions calling each other and hence be far more complicated than this example.

Note also that being reentrant is *not* the same thing as being thread safe. `redirect_stdout()`, for example, is definitely not thread safe, as it makes a global modification to the system state by binding `sys.stdout` to a different stream.

Reusable context managers

Distinct from both single use and reentrant context managers are “reusable” context managers (or, to be completely explicit, “reusable, but not reentrant” context managers, since reentrant context managers are also reusable). These context managers support being used multiple times, but will fail (or otherwise not work correctly) if the specific context manager instance has already been used in a containing with statement.

`threading.Lock` is an example of a reusable, but not reentrant, context manager (for a reentrant lock, it is necessary to use **`threading.RLock`** instead).

Another example of a reusable, but not reentrant, context manager is **`ExitStack`**, as it invokes *all* currently registered callbacks when leaving any with statement, regardless of where those callbacks were added:

```
>>> from contextlib import ExitStack
>>> stack = ExitStack()
>>> with stack:
...     stack.callback(print, "Callback: from first context")
...     print("Leaving first context")
...
Leaving first context
Callback: from first context
>>> with stack:
...     stack.callback(print, "Callback: from second context")
...     print("Leaving second context")
...
Leaving second context
Callback: from second context
>>> with stack:
...     stack.callback(print, "Callback: from outer context")
...     with stack:
...         stack.callback(print, "Callback: from inner
context")
...         print("Leaving inner context")
...     print("Leaving outer context")
...
Leaving inner context
Callback: from inner context
```

```
Callback: from outer context
Leaving outer context
```

As the output from the example shows, reusing a single stack object across multiple with statements works correctly, but attempting to nest them will cause the stack to be cleared at the end of the innermost with statement, which is unlikely to be desirable behaviour.

Using separate **`ExitStack`** instances instead of reusing a single instance avoids that problem:

```
>>> from contextlib import ExitStack
>>> with ExitStack() as outer_stack:
...     outer_stack.callback(print, "Callback: from outer
context")
...     with ExitStack() as inner_stack:
...         inner_stack.callback(print, "Callback: from inner
context")
...         print("Leaving inner context")
...     print("Leaving outer context")
...
Leaving inner context
Callback: from inner context
Leaving outer context
Callback: from outer context
```

`abc` — Abstract Base Classes

Source code: Lib/abc.py
[https://github.com/python/cpython/tree/3.14/Lib/abc.py]

This module provides the infrastructure for defining abstract base classes (ABCs) in Python, as outlined in **PEP 3119** [https://peps.python.org/pep-3119/]; see the PEP for why this was added to Python. (See also **PEP 3141** [https://peps.python.org/pep-3141/] and the **`numbers`** module regarding a type hierarchy for numbers based on ABCs.)

The **`collections`** module has some concrete classes that derive from ABCs; these can, of course, be further derived. In addition, the **`collections.abc`** submodule has some ABCs that can be used to test whether a class or instance provides a particular interface, for example, if it is hashable or if it is a mapping.

This module provides the metaclass **`ABCMeta`** for defining ABCs and a helper class **`ABC`** to alternatively define ABCs through inheritance:

class abc.ABC

A helper class that has **`ABCMeta`** as its metaclass. With this class, an abstract base class can be created by simply deriving from **`ABC`** avoiding sometimes confusing metaclass usage, for example:

```
from abc import ABC
```

```
class MyABC(ABC):
    pass
```

Note that the type of **`ABC`** is still [**`ABCMeta`**], therefore inheriting from **`ABC`** requires the usual precautions regarding metaclass usage, as multiple inheritance may lead to metaclass conflicts. One may also define an abstract base class by passing the metaclass keyword and using **`ABCMeta`** directly, for example:

```
from abc import ABCMeta

class MyABC(metaclass=ABCMeta):
    pass
```

Added in version 3.4.

class abc.ABCMeta

Metaclass for defining Abstract Base Classes (ABCs).

Use this metaclass to create an ABC. An ABC can be subclassed directly, and then acts as a mix-in class. You can also register unrelated concrete classes (even built-in classes) and unrelated ABCs as “virtual subclasses” – these and their descendants will be considered subclasses of the registering ABC by the built-in **`issubclass()`** function, but the registering ABC won’t show up in their MRO (Method Resolution Order) nor will method implementations defined by the registering ABC be callable (not even via **`super()`**). [[1]]

Classes created with a metaclass of **`ABCMeta`** have the following method:

register(*subclass*)

Register *subclass* as a “virtual subclass” of this ABC. For example:

```
from abc import ABC

class MyABC(ABC):
    pass

MyABC.register(tuple)

assert issubclass(tuple, MyABC)
assert isinstance((), MyABC)
```

Changed in version 3.3: Returns the registered subclass, to allow usage as a class decorator.

Changed in version 3.4: To detect calls to **`register()`**, you can use the `get_cache_token()` function.

You can also override this method in an abstract base class:

__subclasshook__(*subclass*)

(Must be defined as a class method.)

Check whether *subclass* is considered a subclass of this ABC. This means that you can customize the behavior of `issubclass()` further without the need to call `register()` on every class you want to consider a subclass of the ABC. (This class method is called from the `__subclasscheck__()` method of the ABC.)

This method should return `True`, `False` or `NotImplemented`. If it returns `True`, the *subclass* is considered a subclass of this ABC. If it returns `False`, the *subclass* is not considered a subclass of this ABC, even if it would normally be one. If it returns **`NotImplemented`**, the subclass check is continued with the usual mechanism.

For a demonstration of these concepts, look at this example ABC definition:

```
class Foo:
    def __getitem__(self, index):
        ...
    def __len__(self):
        ...
    def get_iterator(self):
        return iter(self)

class MyIterable(ABC):

    @abstractmethod
    def __iter__(self):
        while False:
            yield None

    def get_iterator(self):
        return self.__iter__()

    @classmethod
    def __subclasshook__(cls, C):
        if cls is MyIterable:
            if any("__iter__" in B.__dict__ for B in
C.__mro__):
                return True
        return NotImplemented

MyIterable.register(Foo)
```

The ABC `MyIterable` defines the standard iterable method, `__iter__()`, as an abstract method. The implementation given here can still be called from subclasses. The **`get_iterator()`** method is also part of the `MyIterable` abstract base class, but it does not have to be overridden in non-abstract derived classes.

The `__subclasshook__()` class method defined here says that any class that has an `__iter__()` method in its `__dict__` (or in that of one of its base classes, accessed via the `__mro__` list) is considered a `MyIterable` too.

Finally, the last line makes `Foo` a virtual subclass of `MyIterable`, even though it does not define an `__iter__()` method (it uses the old-style iterable protocol, defined in terms of `__len__()` and `__getitem__()`). Note that this will not make `get_iterator` available as a method of `Foo`, so it is provided separately.

The **`abc`** module also provides the following decorator:

@abc.abstractmethod

A decorator indicating abstract methods.

Using this decorator requires that the class's metaclass is `ABCMeta` or is derived from it. A class that has a metaclass derived from **`ABCMeta`** cannot be instantiated unless all of its abstract methods and properties are overridden. The abstract methods can be called using any of the normal 'super' call mechanisms. **`abstractmethod()`** may be used to declare abstract methods for properties and descriptors.

Dynamically adding abstract methods to a class, or attempting to modify the abstraction status of a method or class once it is created, are only supported using the `update_abstractmethods()` function. The **`abstractmethod()`** only affects subclasses derived using regular inheritance; "virtual subclasses" registered with the ABC's `register()` method are not affected.

When **`abstractmethod()`** is applied in combination with other method descriptors, it should be applied as the innermost decorator, as shown in the following usage examples:

```
class C(ABC):
    @abstractmethod
    def my_abstract_method(self, arg1):
        ...
    @classmethod
    @abstractmethod
    def my_abstract_classmethod(cls, arg2):
        ...
    @staticmethod
    @abstractmethod
    def my_abstract_staticmethod(arg3):
        ...

    @property
    @abstractmethod
    def my_abstract_property(self):
        ...
    @my_abstract_property.setter
    @abstractmethod
    def my_abstract_property(self, val):
        ...

    @abstractmethod
    def _get_x(self):
        ...
    @abstractmethod
    def _set_x(self, val):
        ...
    x = property(_get_x, _set_x)
```

In order to correctly interoperate with the abstract base class machinery, the descriptor must identify itself as abstract using **`__isabstractmethod__`**. In general, this attribute should be `True` if any of the methods used to compose the descriptor are abstract. For example, Python’s built-in **`property`** does the equivalent of:

```
class Descriptor:
    ...
    @property
    def __isabstractmethod__(self):
        return any(getattr(f, '__isabstractmethod__', False)
for
                   f in (self._fget, self._fset,
self._fdel))
```

Note

Unlike Java abstract methods, these abstract methods may have an implementation. This implementation can be called via the `super()` mechanism from the class that overrides it. This could be useful as an end-point for a super-call in a framework that uses cooperative multiple-inheritance.

The **`abc`** module also supports the following legacy decorators:

@abc.abstractclassmethod

Added in version 3.2.

Deprecated since version 3.3: It is now possible to use `classmethod` with `abstractmethod()`, making this decorator redundant.

A subclass of the built-in `classmethod()`, indicating an abstract classmethod. Otherwise it is similar to `abstractmethod()`.

This special case is deprecated, as the `classmethod()` decorator is now correctly identified as abstract when applied to an abstract method:

```
class C(ABC):
    @classmethod
    @abstractmethod
    def my_abstract_classmethod(cls, arg):
        ...
```

@abc.abstractstaticmethod

Added in version 3.2.

Deprecated since version 3.3: It is now possible to use `staticmethod` with `abstractmethod()`, making this decorator redundant.

A subclass of the built-in `staticmethod()`, indicating an abstract staticmethod. Otherwise it is similar to `abstractmethod()`.

This special case is deprecated, as the `staticmethod()` decorator is now correctly identified as abstract when applied to an abstract method:

```
class C(ABC):
    @staticmethod
    @abstractmethod
    def my_abstract_staticmethod(arg):
        ...
```

@abc.abstractproperty

Deprecated since version 3.3: It is now possible to use `property`, `property.getter()`, `property.setter()` and `property.deleter()` with `abstractmethod()`, making this decorator redundant.

A subclass of the built-in `property()`, indicating an abstract property.

This special case is deprecated, as the `property()` decorator is now correctly identified as abstract when applied to an abstract method:

```
class C(ABC):
    @property
    @abstractmethod
    def my_abstract_property(self):
        ...
```

The above example defines a read-only property; you can also define a read-write abstract property by appropriately marking one or more of the underlying methods as abstract:

```
class C(ABC):
    @property
    def x(self):
        ...

    @x.setter
    @abstractmethod
    def x(self, val):
        ...
```

If only some components are abstract, only those components need to be updated to create a concrete property in a subclass:

```
class D(C):
    @C.x.setter
    def x(self, val):
        ...
```

The **`abc`** module also provides the following functions:

abc.get_cache_token()

Returns the current abstract base class cache token.

The token is an opaque object (that supports equality testing) identifying the current version of the abstract base class cache for virtual subclasses. The token changes with every call to `ABCMeta.register()` on any ABC.

Added in version 3.4.

abc.update_abstractmethods(*cls*)

A function to recalculate an abstract class’s abstraction status. This function should be called if a class’s abstract methods have been implemented or changed after it was created. Usually, this function should be called from within a class decorator.

Returns *cls*, to allow usage as a class decorator.

If *cls* is not an instance of `ABCMeta`, does nothing.

Note

This function assumes that *cls*’s superclasses are already updated. It does not update any subclasses.

Added in version 3.10.

Footnotes

[1] C++ programmers should note that Python’s virtual base class concept is not the same as C++’s.

`atexit` — Exit handlers

The `atexit` module defines functions to register and unregister cleanup functions. Functions thus registered are automatically executed upon normal interpreter termination. `atexit` runs these functions in the *reverse* order in which they were registered; if you register `A`, `B`, and `C`, at interpreter termination time they will be run in the order `C`, `B`, `A`.

Note: The functions registered via this module are not called when the program is killed by a signal not handled by Python, when a Python fatal internal error is detected, or when `os._exit()` is called.

Note: The effect of registering or unregistering functions from within a cleanup function is undefined.

Changed in version 3.7: When used with C-API subinterpreters, registered functions are local to the interpreter they were registered in.

atexit.register(*func*, **args*, ***kwargs*)

Register *func* as a function to be executed at termination. Any optional arguments that are to be passed to *func* must be passed as arguments to `register()`. It is possible to register the same function and arguments more than once.

At normal program termination (for instance, if `sys.exit()` is called or the main module’s execution completes), all functions registered are called in last in, first out order. The assumption is that lower level modules will normally be imported before higher level modules and thus must be cleaned up later.

If an exception is raised during execution of the exit handlers, a traceback is printed (unless `SystemExit` is raised) and the exception information is saved. After all exit handlers have had a chance to run, the last exception to be raised is re-raised.

This function returns *func*, which makes it possible to use it as a decorator.

Warning

Starting new threads or calling `os.fork()` from a registered function can lead to race condition between the main Python runtime thread freeing thread states while internal `threading` routines or the new process try to use that state. This can lead to crashes rather than clean shutdown.

Changed in version 3.12: Attempts to start a new thread or `os.fork()` a new process in a registered function now leads to `RuntimeError`.

atexit.unregister(*func*)

Remove *func* from the list of functions to be run at interpreter shutdown. `unregister()` silently does nothing if *func* was not previously registered. If *func* has been registered more than once, every occurrence of that function in the `atexit` call stack will be removed. Equality comparisons (==) are used internally during unregistration, so function references do not need to have matching identities.

See also

Module `readline`

Useful example of `atexit` to read and write `readline` history files.

`atexit` Example

The following simple example demonstrates how a module can initialize a counter from a file when it is imported and save the counter’s updated value automatically when the program terminates without relying on the application making an explicit call into this module at termination.

```
try:
    with open('counterfile') as infile:
        _count = int(infile.read())
except FileNotFoundError:
    _count = 0

def incrcounter(n):
    global _count
    _count = _count + n

def savecounter():
    with open('counterfile', 'w') as outfile:
        outfile.write('%d' % _count)

import atexit

atexit.register(savecounter)
```

Positional and keyword arguments may also be passed to `register()` to be passed along to the registered function when it is called:

```
def goodbye(name, adjective):
    print('Goodbye %s, it was %s to meet you.' % (name,
adjective))
```

```
import atexit

atexit.register(goodbye, 'Donny', 'nice')
# or:
atexit.register(goodbye, adjective='nice', name='Donny')
```

Usage as a decorator:

```
import atexit

@atexit.register
def goodbye():
    print('You are now leaving the Python sector.')
```

This only works with functions that can be called without arguments.

`traceback` — Print or retrieve a stack traceback

Source code: Lib/traceback.py
[https://github.com/python/cpython/tree/3.14/Lib/traceback.py]

This module provides a standard interface to extract, format and print stack traces of Python programs. It is more flexible than the interpreter’s default traceback display, and therefore makes it possible to configure certain aspects of the output. Finally, it contains a utility for capturing enough information about an exception to print it later, without the need to save a reference to the actual exception. Since exceptions can be the roots of large objects graph, this utility can significantly improve memory management.

The module uses traceback objects — these are objects of type **`types.TracebackType`**, which are assigned to the **`__traceback__`** field of **`BaseException`** instances.

See also

Module `faulthandler`
Used to dump Python tracebacks explicitly, on a fault, after a timeout, or on a user signal.

Module `pdb`
Interactive source code debugger for Python programs.

The module’s API can be divided into two parts:

- Module-level functions offering basic functionality, which are useful for interactive inspection of exceptions and tracebacks.
- `TracebackException` class and its helper classes `StackSummary` and `FrameSummary`. These offer both more flexibility in the output generated and the ability to store the information necessary for later formatting without holding references to actual exception and traceback objects.

Added in version 3.13: Output is colorized by default and can be controlled using environment variables.

Module-Level Functions

traceback.print_tb(*tb*, *limit=None*, *file=None*)

Print up to *limit* stack trace entries from traceback object *tb* (starting from the caller's frame) if *limit* is positive. Otherwise, print the last `abs(limit)` entries. If *limit* is omitted or `None`, all entries are printed. If *file* is omitted or `None`, the output goes to `sys.stderr`; otherwise it should be an open file or file-like object to receive the output.

Note

The meaning of the *limit* parameter is different than the meaning of `sys.tracebacklimit`. A negative *limit* value corresponds to a positive value of `sys.tracebacklimit`, whereas the behaviour of a positive *limit* value cannot be achieved with `sys.tracebacklimit`.

Changed in version 3.5: Added negative *limit* support.

traceback.print_exception(*exc*, */*, [*value*, *tb*,]*limit=None*, *file=None*, *chain=True*)

Print exception information and stack trace entries from traceback object *tb* to *file*. This differs from `print_tb()` in the following ways:

- if *tb* is not `None`, it prints a header `Traceback (most recent call last):`
- it prints the exception type and *value* after the stack trace
- if *type(value)* is **`SyntaxError`** and *value* has the appropriate format, it prints the line where the syntax error occurred with a caret indicating the approximate position of the error.

Since Python 3.10, instead of passing *value* and *tb*, an exception object can be passed as the first argument. If *value* and *tb* are provided, the first argument is ignored in order to provide backwards compatibility.

The optional *limit* argument has the same meaning as for `print_tb()`. If *chain* is true (the default), then chained exceptions (the `__cause__` or `__context__` attributes of the exception) will be printed as well, like the interpreter itself does when printing an unhandled exception.

Changed in version 3.5: The *etype* argument is ignored and inferred from the type of *value*.

Changed in version 3.10: The *etype* parameter has been renamed to *exc* and is now positional-only.

traceback.print_exc(*limit=None*, *file=None*, *chain=True*)

This is a shorthand for `print_exception(sys.exception(), limit=limit, file=file, chain=chain)`.

traceback.print_last(*limit=None*, *file=None*, *chain=True*)

This is a shorthand for `print_exception(sys.last_exc, limit=limit, file=file, chain=chain)`. In general it will work only after an exception has reached an interactive prompt (see **`sys.last_exc`**).

traceback.print_stack(*f=None*, *limit=None*, *file=None*)

Print up to *limit* stack trace entries (starting from the invocation point) if *limit* is positive. Otherwise, print the last `abs(limit)` entries. If *limit* is omitted or `None`, all entries are printed. The optional *f* argument can be used to specify an alternate stack frame to start. The optional *file* argument has the same meaning as for **`print_tb()`**.

Changed in version 3.5: Added negative *limit* support.

traceback.extract_tb(*tb*, *limit=None*)

Return a **`StackSummary`** object representing a list of “pre-processed” stack trace entries extracted from the traceback object *tb*. It is useful for alternate formatting of stack traces. The optional *limit* argument has the same meaning as for **`print_tb()`**. A “pre-processed” stack trace entry is a **`FrameSummary`** object containing attributes **`filename`**, **`lineno`**, **`name`**, and **`line`** representing the information that is usually printed for a stack trace.

traceback.extract_stack(*f=None*, *limit=None*)

Extract the raw traceback from the current stack frame. The return value has the same format as for **`extract_tb()`**. The optional *f* and *limit* arguments have the same meaning as for **`print_stack()`**.

traceback.print_list(*extracted_list*, *file=None*)

Print the list of tuples as returned by **`extract_tb()`** or **`extract_stack()`** as a formatted stack trace to the given file. If *file* is `None`, the output is written to **`sys.stderr`**.

traceback.format_list(*extracted_list*)

Given a list of tuples or `FrameSummary` objects as returned by `extract_tb()` or `extract_stack()`, return a list of strings ready for printing. Each string in the resulting list corresponds to the item with the same index in the argument list. Each string ends in a newline; the strings may contain internal newlines as well, for those items whose source text line is not `None`.

traceback.format_exception_only(*exc*, */*, [*value*,]*, *show_group=False*)

Format the exception part of a traceback using an exception value such as given by `sys.last_value`. The return value is a list of strings, each ending in a newline. The list contains the exception's message, which is normally a single string; however, for `SyntaxError` exceptions, it contains several lines that (when printed) display detailed information about where the syntax error occurred. Following the message, the list contains the exception's `notes`.

Since Python 3.10, instead of passing *value*, an exception object can be passed as the first argument. If *value* is provided, the first argument is ignored in order to provide backwards compatibility.

When *show_group* is `True`, and the exception is an instance of `BaseExceptionGroup`, the nested exceptions are included as well, recursively, with indentation relative to their nesting depth.

Changed in version 3.10: The *etype* parameter has been renamed to *exc* and is now positional-only.

Changed in version 3.11: The returned list now includes any **`notes`** attached to the exception.

Changed in version 3.13: show_group parameter was added.

traceback.format_exception(*exc*, */*, [*value*, *tb*,]*limit=None*, *chain=True*)

Format a stack trace and the exception information. The arguments have the same meaning as the corresponding arguments to **`print_exception()`**. The return value is a list of strings, each ending in a newline and some containing internal newlines. When these lines are concatenated and printed, exactly the same text is printed as does **`print_exception()`**.

Changed in version 3.5: The *etype* argument is ignored and inferred from the type of *value*.

Changed in version 3.10: This function's behavior and signature were modified to match **`print_exception()`**.

traceback.format_exc(*limit=None*, *chain=True*)

This is like `print_exc(limit)` but returns a string instead of printing to a file.

traceback.format_tb(*tb*, *limit=None*)

A shorthand for `format_list(extract_tb(tb, limit))`.

traceback.format_stack(*f=None*, *limit=None*)

A shorthand for `format_list(extract_stack(f, limit))`.

traceback.clear_frames(*tb*)

Clears the local variables of all the stack frames in a traceback *tb* by calling the **`clear()`** method of each frame object.

Added in version 3.4.

traceback.walk_stack(*f*)

Walk a stack following `f.f_back` from the given frame, yielding the frame and line number for each frame. If *f* is `None`, the current stack is used. This helper is used with `StackSummary.extract()`.

Added in version 3.5.

Changed in version 3.14: This function previously returned a generator that would walk the stack when first iterated over. The generator returned now is the state of the stack when `walk_stack` is called.

traceback.walk_tb(*tb*)

Walk a traceback following `tb_next` yielding the frame and line number for each frame. This helper is used with `StackSummary.extract()`.

Added in version 3.5.

`TracebackException` Objects

Added in version 3.5.

`TracebackException` objects are created from actual exceptions to capture data for later printing. They offer a more lightweight method of storing this information by avoiding holding references to traceback and frame objects. In addition, they expose more options to configure the output compared to the module-level functions described above.

class traceback.TracebackException(*exc_type*, *exc_value*, *exc_traceback*, *, *limit=None*, *lookup_lines=True*, *capture_locals=False*, *compact=False*, *max_group_width=15*, *max_group_depth=10*)

Capture an exception for later rendering. The meaning of *limit*, *lookup_lines* and *capture_locals* are as for the `StackSummary` class.

If *compact* is true, only data that is required by **`TracebackException`**'s `format()` method is saved in the class attributes. In particular, the `__context__` field is calculated only if `__cause__` is `None` and `__suppress_context__` is false.

Note that when locals are captured, they are also shown in the traceback.

max_group_width and *max_group_depth* control the formatting of exception groups (see `BaseExceptionGroup`). The depth refers to the nesting level of the group, and the width refers to the size of a single exception group's exceptions array. The formatted output is truncated when either limit is exceeded.

Changed in version 3.10: Added the *compact* parameter.

Changed in version 3.11: Added the *max_group_width* and *max_group_depth* parameters.

__cause__

A **`TracebackException`** of the original `__cause__`.

__context__

A **`TracebackException`** of the original `__context__`.

exceptions

If `self` represents an `ExceptionGroup`, this field holds a list of **`TracebackException`** instances representing the nested exceptions. Otherwise it is `None`.

Added in version 3.11.

__suppress_context__

The `__suppress_context__` value from the original exception.

__notes__

The `__notes__` value from the original exception, or `None` if the exception does not have any notes. If it is not `None` is it formatted in the traceback after the exception string.

Added in version 3.11.

stack

A `StackSummary` representing the traceback.

exc_type

The class of the original traceback.

Deprecated since version 3.13.

exc_type_str

String display of the class of the original exception.

Added in version 3.13.

filename

For syntax errors - the file name where the error occurred.

lineno

For syntax errors - the line number where the error occurred.

end_lineno

For syntax errors - the end line number where the error occurred. Can be `None` if not present.

Added in version 3.10.

text

For syntax errors - the text where the error occurred.

offset

For syntax errors - the offset into the text where the error occurred.

end_offset

For syntax errors - the end offset into the text where the error occurred. Can be `None` if not present.

Added in version 3.10.

msg

For syntax errors - the compiler error message.

classmethod from_exception(*exc*, *, *limit=None*, *lookup_lines=True*, *capture_locals=False*)

Capture an exception for later rendering. *limit*, *lookup_lines* and *capture_locals* are as for the **`StackSummary`** class.

Note that when locals are captured, they are also shown in the traceback.

print(*, *file=None*, *chain=True*)

Print to *file* (default `sys.stderr`) the exception information returned by `format()`.

Added in version 3.11.

format(*, *chain=True*)

Format the exception.

If *chain* is not `True`, `__cause__` and `__context__` will not be formatted.

The return value is a generator of strings, each ending in a newline and some containing internal newlines. `print_exception()` is a wrapper around this method which just prints the lines to a file.

format_exception_only(*, *show_group=False*)

Format the exception part of the traceback.

The return value is a generator of strings, each ending in a newline.

When *show_group* is `False`, the generator emits the exception's message followed by its notes (if it has any). The exception message is normally a single string; however, for `SyntaxError` exceptions, it consists of several lines that (when printed) display detailed information about where the syntax error occurred.

When *show_group* is `True`, and the exception is an instance of `BaseExceptionGroup`, the nested exceptions are included as well, recursively, with indentation relative to their nesting depth.

Changed in version 3.11: The exception's `notes` are now included in the output.

Changed in version 3.13: Added the *show_group* parameter.

`StackSummary` Objects

Added in version 3.5.

`StackSummary` objects represent a call stack ready for formatting.

class traceback.StackSummary

classmethod extract(*frame_gen*, ***, *limit=None*, *lookup_lines=True*, *capture_locals=False*)

Construct a **`StackSummary`** object from a frame generator (such as is returned by `walk_stack()` or `walk_tb()`).

If *limit* is supplied, only this many frames are taken from *frame_gen*. If *lookup_lines* is `False`, the returned `FrameSummary` objects will not have read their lines in yet, making the cost of creating the **`StackSummary`** cheaper (which may be valuable if it may not actually get formatted). If *capture_locals* is `True` the local variables in each **`FrameSummary`** are captured as object representations.

Changed in version 3.12: Exceptions raised from `repr()` on a local variable (when *capture_locals* is `True`) are no longer propagated to the caller.

classmethod from_list(*a_list*)

Construct a **`StackSummary`** object from a supplied list of `FrameSummary` objects or old-style list of tuples. Each tuple should be a 4-tuple with *filename*, *lineno*, *name*, *line* as the elements.

format()

Returns a list of strings ready for printing. Each string in the resulting list corresponds to a single frame from the stack. Each string ends in a newline; the strings may contain internal newlines as well, for those items with source text lines.

For long sequences of the same frame and line, the first few repetitions are shown, followed by a summary line stating the exact number of further repetitions.

Changed in version 3.6: Long sequences of repeated frames are now abbreviated.

format_frame_summary(*frame_summary*)

Returns a string for printing one of the frames involved in the stack. This method is called for each **`FrameSummary`** object to be printed by **`StackSummary.format()`**. If it returns `None`, the frame is omitted from the output.

Added in version 3.11.

`FrameSummary` Objects

Added in version 3.5.

A **`FrameSummary`** object represents a single frame in a traceback.

class traceback.FrameSummary(*filename*, *lineno*, *name*, ***, *lookup_line=True*, *locals=None*, *line=None*, *end_lineno=None*, *colno=None*, *end_colno=None*)

Represents a single frame in the traceback or stack that is being formatted or printed. It may optionally have a stringified version of the frame's locals included in it. If *lookup_line* is `False`, the source code is not looked up until the **`FrameSummary`** has the **`line`** attribute accessed (which also happens when casting it to a **`tuple`**). **`line`** may be directly provided, and will prevent line lookups happening at all. *locals* is an optional local variable mapping, and if supplied the variable representations are stored in the summary for later display.

`FrameSummary` instances have the following attributes:

filename

The filename of the source code for this frame. Equivalent to accessing **`f.f_code.co_filename`** on a frame object *f*.

lineno

The line number of the source code for this frame.

name

Equivalent to accessing **`f.f_code.co_name`** on a frame object *f*.

line

A string representing the source code for this frame, with leading and trailing whitespace stripped. If the source is not available, it is `None`.

end_lineno

The last line number of the source code for this frame. By default, it is set to `lineno` and indexation starts from 1.

Changed in version 3.13: The default value changed from `None` to `lineno`.

colno

The column number of the source code for this frame. By default, it is `None` and indexation starts from 0.

end_colno

The last column number of the source code for this frame. By default, it is `None` and indexation starts from 0.

Examples of Using the Module-Level Functions

This simple example implements a basic read-eval-print loop, similar to (but less useful than) the standard Python interactive interpreter loop. For a more complete implementation of the interpreter loop, refer to the `code` module.

```
import sys, traceback

def run_user_code(envdir):
    source = input(">>> ")
    try:
        exec(source, envdir)
    except Exception:
        print("Exception in user code:")
        print("-"*60)
        traceback.print_exc(file=sys.stdout)
        print("-"*60)

envdir = {}
while True:
    run_user_code(envdir)
```

The following example demonstrates the different ways to print and format the exception and traceback:

```
import sys, traceback

def lumberjack():
    bright_side_of_life()

def bright_side_of_life():
    return tuple()[0]

try:
    lumberjack()
except IndexError as exc:
    print("*** print_tb:")
    traceback.print_tb(exc.__traceback__, limit=1,
file=sys.stdout)
    print("*** print_exception:")
    traceback.print_exception(exc, limit=2, file=sys.stdout)
    print("*** print_exc:")
    traceback.print_exc(limit=2, file=sys.stdout)
    print("*** format_exc, first and last line:")
    formatted_lines = traceback.format_exc().splitlines()
    print(formatted_lines[0])
    print(formatted_lines[-1])
    print("*** format_exception:")
    print(repr(traceback.format_exception(exc)))
    print("*** extract_tb:")
    print(repr(traceback.extract_tb(exc.__traceback__)))
    print("*** format_tb:")
    print(repr(traceback.format_tb(exc.__traceback__)))
    print("*** tb_lineno:", exc.__traceback__.tb_lineno)
```

The output for the example would look similar to this:

```
*** print_tb:
  File "<doctest...>", line 10, in <module>
    lumberjack()
    ~~~~~~~~~~^^
*** print_exception:
Traceback (most recent call last):
  File "<doctest...>", line 10, in <module>
```

```
    lumberjack()
    ~~~~~~~~~~^^
  File "<doctest...>", line 4, in lumberjack
    bright_side_of_life()
    ~~~~~~~~~~~~~~~~~~~^^
IndexError: tuple index out of range
*** print_exc:
Traceback (most recent call last):
  File "<doctest...>", line 10, in <module>
    lumberjack()
    ~~~~~~~~~~^^
  File "<doctest...>", line 4, in lumberjack
    bright_side_of_life()
    ~~~~~~~~~~~~~~~~~~~^^
IndexError: tuple index out of range
*** format_exc, first and last line:
Traceback (most recent call last):
IndexError: tuple index out of range
*** format_exception:
['Traceback (most recent call last):\n',
 '  File "<doctest default[0]>", line 10, in <module>\n
lumberjack()\n    ~~~~~~~~~~^^\n',
 '  File "<doctest default[0]>", line 4, in lumberjack\n
bright_side_of_life()\n    ~~~~~~~~~~~~~~~~~~~^^\n',
 '  File "<doctest default[0]>", line 7, in
bright_side_of_life\n    return tuple()[0]\n
~~~~~~~^^^\n',
 'IndexError: tuple index out of range\n']
*** extract_tb:
[<FrameSummary file <doctest...>, line 10 in <module>>,
 <FrameSummary file <doctest...>, line 4 in lumberjack>,
 <FrameSummary file <doctest...>, line 7 in
bright_side_of_life>]
*** format_tb:
['  File "<doctest default[0]>", line 10, in <module>\n
lumberjack()\n    ~~~~~~~~~~^^\n',
 '  File "<doctest default[0]>", line 4, in lumberjack\n
bright_side_of_life()\n    ~~~~~~~~~~~~~~~~~~~^^\n',
 '  File "<doctest default[0]>", line 7, in
bright_side_of_life\n    return tuple()[0]\n
~~~~~~~^^^\n']
*** tb_lineno: 10
```

The following example shows the different ways to print and format the stack:

```
>>> import traceback
>>> def another_function():
...     lumberstack()
...
>>> def lumberstack():
...     traceback.print_stack()
...     print(repr(traceback.extract_stack()))
...     print(repr(traceback.format_stack()))
...
>>> another_function()
  File "<doctest>", line 10, in <module>
    another_function()
  File "<doctest>", line 3, in another_function
    lumberstack()
  File "<doctest>", line 6, in lumberstack
    traceback.print_stack()
[('<doctest>', 10, '<module>', 'another_function()'),
 ('<doctest>', 3, 'another_function', 'lumberstack()'),
 ('<doctest>', 7, 'lumberstack',
'print(repr(traceback.extract_stack()))')]
['  File "<doctest>", line 10, in <module>\n
another_function()\n',
 '  File "<doctest>", line 3, in another_function\n
lumberstack()\n',
 '  File "<doctest>", line 8, in lumberstack\n
print(repr(traceback.format_stack()))\n']
```

This last example demonstrates the final few formatting functions:

```
>>> import traceback
>>> traceback.format_list([('spam.py', 3, '<module>',
'spam.eggs()'),
...                        ('eggs.py', 42, 'eggs', 'return
"bacon"')])
['  File "spam.py", line 3, in <module>\n    spam.eggs()\n',
 '  File "eggs.py", line 42, in eggs\n    return "bacon"\n']
>>> an_error = IndexError('tuple index out of range')
>>> traceback.format_exception_only(an_error)
['IndexError: tuple index out of range\n']
```

Examples of Using `TracebackException`

With the helper class, we have more options:

```
>>> import sys
>>> from traceback import TracebackException
>>>
>>> def lumberjack():
...     bright_side_of_life()
...
>>> def bright_side_of_life():
...     t = "bright", "side", "of", "life"
...     return t[5]
...
>>> try:
...     lumberjack()
... except IndexError as e:
...     exc = e
...
>>> try:
...     try:
...         lumberjack()
...     except:
...         1/0
... except Exception as e:
...     chained_exc = e
...
>>> # limit works as with the module-level functions
>>> TracebackException.from_exception(exc, limit=-2).print()
Traceback (most recent call last):
  File "<python-input-1>", line 6, in lumberjack
    bright_side_of_life()
    ~~~~~~~~~~~~~~~~~~~^^
  File "<python-input-1>", line 10, in bright_side_of_life
    return t[5]
           ~^^^
IndexError: tuple index out of range
```

```
>>> # capture_locals adds local variables in frames
>>> TracebackException.from_exception(exc, limit=-2,
capture_locals=True).print()
Traceback (most recent call last):
  File "<python-input-1>", line 6, in lumberjack
    bright_side_of_life()
    ~~~~~~~~~~~~~~~~~~~^^
  File "<python-input-1>", line 10, in bright_side_of_life
    return t[5]
           ~^^^
    t = ("bright", "side", "of", "life")
IndexError: tuple index out of range

>>> # The *chain* kwarg to print() controls whether chained
>>> # exceptions are displayed
>>> TracebackException.from_exception(chained_exc).print()
Traceback (most recent call last):
  File "<python-input-19>", line 4, in <module>
    lumberjack()
    ~~~~~~~~~~^^
  File "<python-input-8>", line 7, in lumberjack
    bright_side_of_life()
    ~~~~~~~~~~~~~~~~~~~^^
  File "<python-input-8>", line 11, in bright_side_of_life
    return t[5]
           ~^^^
IndexError: tuple index out of range

During handling of the above exception, another exception
occurred:

Traceback (most recent call last):
  File "<python-input-19>", line 6, in <module>
    1/0
    ~^~
ZeroDivisionError: division by zero

>>>
TracebackException.from_exception(chained_exc).print(chain=Fals
e)
Traceback (most recent call last):
  File "<python-input-19>", line 6, in <module>
    1/0
```

```
    ~^~
ZeroDivisionError: division by zero
```

`__future__` — Future statement definitions

Source code: Lib/__future__.py
[https://github.com/python/cpython/tree/3.14/Lib/__future__.py]

Imports of the form `from __future__ import feature` are called future statements. These are special-cased by the Python compiler to allow the use of new Python features in modules containing the future statement before the release in which the feature becomes standard.

While these future statements are given additional special meaning by the Python compiler, they are still executed like any other import statement and the `__future__` exists and is handled by the import system the same way any other Python module would be. This design serves three purposes:

- To avoid confusing existing tools that analyze import statements and expect to find the modules they're importing.
- To document when incompatible changes were introduced, and when they will be — or were — made mandatory. This is a form of executable documentation, and can be inspected programmatically via importing `__future__` and examining its contents.
- To ensure that future statements run under releases prior to Python 2.1 at least yield runtime exceptions (the import of `__future__` will fail, because there was no module of that name prior to 2.1).

Module Contents

No feature description will ever be deleted from `__future__`. Since its introduction in Python 2.1 the following features have found their way into the language using this mechanism:

feature	**optional in**	**mandatory in**	**effect**
__future__.nested_scopes	2.1.0b1	2.2	PEP 227: *Statically Nested Scopes*
__future__.generators	2.2.0a1	2.3	PEP 255: *Simple Generators*
__future__.division	2.2.0a2	3.0	PEP 238: *Changing the Division Operator*
__future__.absolute_import	2.5.0a1	3.0	PEP 328: *Imports: Multi-Line and Absolute/Relative*
__future__.with_statement	2.5.0a1	2.6	PEP 343: *The “with” Statement*
__future__.print_function	2.6.0a2	3.0	PEP 3105: *Make print a function*
__future__.unicode_literals	2.6.0a2	3.0	PEP 3112: *Bytes literals in Python 3000*
__future__.generator_stop	3.5.0b1	3.7	PEP 479: *StopIteration handling inside generators*

feature	optional in	mandatory in	effect
__future__.annotations	3.7.0b1	Never [[1]]	**PEP 563**: *Postponed evaluation of annotations*, **PEP 649**: *Deferred evaluation of annotations using descriptors*

class __future__._Feature

Each statement in `__future__.py` is of the form:

```
FeatureName = _Feature(OptionalRelease, MandatoryRelease,
                       CompilerFlag)
```

where, normally, *OptionalRelease* is less than *MandatoryRelease*, and both are 5-tuples of the same form as `sys.version_info`:

```
(PY_MAJOR_VERSION, # the 2 in 2.1.0a3; an int
 PY_MINOR_VERSION, # the 1; an int
 PY_MICRO_VERSION, # the 0; an int
 PY_RELEASE_LEVEL, # "alpha", "beta", "candidate" or "final"; string
 PY_RELEASE_SERIAL # the 3; an int
)
```

_Feature.getOptionalRelease()

OptionalRelease records the first release in which the feature was accepted.

_Feature.getMandatoryRelease()

In the case of a *MandatoryRelease* that has not yet occurred, *MandatoryRelease* predicts the release in which the feature will become part of the language.

Else *MandatoryRelease* records when the feature became part of the language; in releases at or after that, modules no longer need a future statement to use the feature in question, but may continue to use such imports.

MandatoryRelease may also be `None`, meaning that a planned feature got dropped or that it is not yet decided.

_Feature.compiler_flag

CompilerFlag is the (bitfield) flag that should be passed in the fourth argument to the built-in function `compile()` to enable the feature in dynamically compiled code. This flag is stored in the `_Feature.compiler_flag` attribute on `_Feature` instances.

[1]

`from __future__ import annotations` was previously scheduled to become mandatory in Python 3.10, but the change was delayed and ultimately canceled. This feature will eventually be deprecated and removed. See **PEP 649** [https://peps.python.org/pep-0649/] and **PEP 749** [https://peps.python.org/pep-0749/].

See also

Future statements
How the compiler treats future imports.

PEP 236 [https://peps.python.org/pep-0236/] - Back to the __future__
The original proposal for the __future__ mechanism.

`gc` — Garbage Collector interface

This module provides an interface to the optional garbage collector. It provides the ability to disable the collector, tune the collection frequency, and set debugging options. It also provides access to unreachable objects that the collector found but cannot free. Since the collector supplements the reference counting already used in Python, you can disable the collector if you are sure your program does not create reference cycles. Automatic collection can be disabled by calling `gc.disable()`. To debug a leaking program call `gc.set_debug(gc.DEBUG_LEAK)`. Notice that this includes `gc.DEBUG_SAVEALL`, causing garbage-collected objects to be saved in gc.garbage for inspection.

The **`gc`** module provides the following functions:

gc.enable()

Enable automatic garbage collection.

gc.disable()

Disable automatic garbage collection.

gc.isenabled()

Return `True` if automatic collection is enabled.

gc.collect(*generation=2*)

Perform a collection. The optional argument *generation* may be an integer specifying which generation to collect (from 0 to 2). A **`ValueError`** is raised if the generation number is invalid. The sum of collected objects and uncollectable objects is returned.

Calling `gc.collect(0)` will perform a GC collection on the young generation.

Calling `gc.collect(1)` will perform a GC collection on the young generation and an increment of the old generation.

Calling `gc.collect(2)` or `gc.collect()` performs a full collection

The free lists maintained for a number of built-in types are cleared whenever a full collection or collection of the highest generation (2) is run. Not all items in some free lists may be freed due to the particular implementation, in particular **`float`**.

The effect of calling `gc.collect()` while the interpreter is already performing a collection is undefined.

Changed in version 3.14: `generation=1` performs an increment of collection.

gc.set_debug(*flags*)

Set the garbage collection debugging flags. Debugging information will be written to `sys.stderr`. See below for a list of debugging flags which can be combined using bit operations to control debugging.

gc.get_debug()

Return the debugging flags currently set.

gc.get_objects(*generation=None*)

Returns a list of all objects tracked by the collector, excluding the list returned. If *generation* is not `None`, return only the objects as follows:

- 0: All objects in the young generation
- 1: No objects, as there is no generation 1 (as of Python 3.14)
- 2: All objects in the old generation

Changed in version 3.8: New *generation* parameter.

Changed in version 3.14: Generation 1 is removed

Raises an auditing event `gc.get_objects` with argument `generation`.

gc.get_stats()

Return a list of three per-generation dictionaries containing collection statistics since interpreter start. The number of keys may change in the future, but currently each dictionary will contain the following items:

- `collections` is the number of times this generation was collected;
- `collected` is the total number of objects collected inside this generation;
- `uncollectable` is the total number of objects which were found to be uncollectable (and were therefore moved to the **`garbage`** list) inside this generation.

Added in version 3.4.

gc.set_threshold(*threshold0*[, *threshold1*[, *threshold2*]])

Set the garbage collection thresholds (the collection frequency). Setting *threshold0* to zero disables collection.

The GC classifies objects into two generations depending on whether they have survived a collection. New objects are placed in the young generation. If an object survives a collection it is moved into the old generation.

In order to decide when to run, the collector keeps track of the number of object allocations and deallocations since the last collection. When the number of allocations minus the number of deallocations exceeds *threshold0*, collection starts. For each collection, all the objects in the young generation and some fraction of the old generation is collected.

In the free-threaded build, the increase in process memory usage is also checked before running the collector. If the memory usage has not increased by 10% since the last collection and the net number of object allocations has not exceeded 40 times *threshold0*, the collection is not run.

The fraction of the old generation that is collected is **inversely** proportional to *threshold1*. The larger *threshold1* is, the slower objects in the old generation are collected. For the default value of 10, 1% of the old generation is scanned during each collection.

threshold2 is ignored.

See Garbage collector design [https://github.com/python/cpython/blob/3.14/InternalDocs/garbage_collector.md] for more information.

Changed in version 3.14: threshold2 is ignored

gc.get_count()

Return the current collection counts as a tuple of `(count0, count1, count2)`.

gc.get_threshold()

Return the current collection thresholds as a tuple of `(threshold0, threshold1, threshold2)`.

gc.get_referrers(**objs*)

Return the list of objects that directly refer to any of objs. This function will only locate those containers which support garbage collection; extension types which do refer to other objects but do not support garbage collection will not be found.

Note that objects which have already been dereferenced, but which live in cycles and have not yet been collected by the garbage collector can be listed among the resulting referrers. To get only currently live objects, call `collect()` before calling `get_referrers()`.

Warning

Care must be taken when using objects returned by `get_referrers()` because some of them could still be under construction and hence in a temporarily invalid state. Avoid using `get_referrers()` for any purpose other than debugging.

Raises an auditing event `gc.get_referrers` with argument `objs`.

gc.get_referents(**objs*)

Return a list of objects directly referred to by any of the arguments. The referents returned are those objects visited by the arguments’ C-level `tp_traverse` methods (if any), and may not be all objects actually directly reachable. **`tp_traverse`** methods are supported only by objects that support garbage collection, and are only required to visit objects that may be involved in a cycle. So, for example, if an integer is directly reachable from an argument, that integer object may or may not appear in the result list.

Raises an auditing event `gc.get_referents` with argument `objs`.

gc.is_tracked(*obj*)

Returns `True` if the object is currently tracked by the garbage collector, `False` otherwise. As a general rule, instances of atomic types aren’t tracked and instances of non-atomic types (containers, user-defined objects…) are. However, some type-specific optimizations can be present in order to suppress the garbage collector footprint of simple instances (e.g. dicts containing only atomic keys and values):

```
>>> gc.is_tracked(0)
False
>>> gc.is_tracked("a")
False
>>> gc.is_tracked([])
True
>>> gc.is_tracked({})
False
>>> gc.is_tracked({"a": 1})
True
```

Added in version 3.1.

gc.is_finalized(*obj*)

Returns `True` if the given object has been finalized by the garbage collector, `False` otherwise.

```
>>> x = None
>>> class Lazarus:
...     def __del__(self):
...         global x
...         x = self
...
>>> lazarus = Lazarus()
>>> gc.is_finalized(lazarus)
False
>>> del lazarus
>>> gc.is_finalized(x)
True
```

Added in version 3.9.

gc.freeze()

Freeze all the objects tracked by the garbage collector; move them to a permanent generation and ignore them in all the future collections.

If a process will `fork()` without `exec()`, avoiding unnecessary copy-on-write in child processes will maximize memory sharing and reduce overall memory usage. This requires both avoiding creation of freed “holes” in memory pages in the parent process and ensuring that GC collections in child processes won’t touch the `gc_refs` counter of long-lived objects originating in the parent process. To accomplish both, call `gc.disable()` early in the parent process, `gc.freeze()` right before `fork()`, and `gc.enable()` early in child processes.

Added in version 3.7.

gc.unfreeze()

Unfreeze the objects in the permanent generation, put them back into the oldest generation.

Added in version 3.7.

gc.get_freeze_count()

Return the number of objects in the permanent generation.

Added in version 3.7.

The following variables are provided for read-only access (you can mutate the values but should not rebind them):

gc.garbage

A list of objects which the collector found to be unreachable but could not be freed (uncollectable objects). Starting with Python 3.4, this list should be empty most of the time, except when using instances of C extension types with a non-`NULL` `tp_del` slot.

If **`DEBUG_SAVEALL`** is set, then all unreachable objects will be added to this list rather than freed.

Changed in version 3.2: If this list is non-empty at interpreter shutdown, a **`ResourceWarning`** is emitted, which is silent by default. If **`DEBUG_UNCOLLECTABLE`** is set, in addition all uncollectable objects are printed.

Changed in version 3.4: Following **PEP 442** [https://peps.python.org/pep-0442/], objects with a **`__del__()`** method don’t end up in **`gc.garbage`** anymore.

gc.callbacks

A list of callbacks that will be invoked by the garbage collector before and after collection. The callbacks will be called with two arguments, *phase* and *info*.

phase can be one of two values:

“start”: The garbage collection is about to start.

“stop”: The garbage collection has finished.

info is a dict providing more information for the callback. The following keys are currently defined:

“generation”: The oldest generation being collected.

“collected”: When *phase* is “stop”, the number of objects successfully collected.

“uncollectable”: When *phase* is “stop”, the number of objects that could not be collected and were put in `garbage`.

Applications can add their own callbacks to this list. The primary use cases are:

Gathering statistics about garbage collection, such as how often various generations are collected, and how long the collection takes.

Allowing applications to identify and clear their own uncollectable types when they appear in `garbage`.

Added in version 3.3.

The following constants are provided for use with `set_debug()`:

gc.DEBUG_STATS

Print statistics during collection. This information can be useful when tuning the collection frequency.

gc.DEBUG_COLLECTABLE

Print information on collectable objects found.

gc.DEBUG_UNCOLLECTABLE

Print information of uncollectable objects found (objects which are not reachable but cannot be freed by the collector). These objects will be added to the `garbage` list.

Changed in version 3.2: Also print the contents of the **`garbage`** list at interpreter shutdown, if it isn't empty.

gc.DEBUG_SAVEALL

When set, all unreachable objects found will be appended to *garbage* rather than being freed. This can be useful for debugging a leaking program.

gc.DEBUG_LEAK

The debugging flags necessary for the collector to print information about a leaking program (equal to `DEBUG_COLLECTABLE | DEBUG_UNCOLLECTABLE | DEBUG_SAVEALL`).

`inspect` — Inspect live objects

Source code: Lib/inspect.py
[https://github.com/python/cpython/tree/3.14/Lib/inspect.py]

The **`inspect`** module provides several useful functions to help get information about live objects such as modules, classes, methods, functions, tracebacks, frame objects, and code objects. For example, it can help you examine the contents of a class, retrieve the source code of a method, extract and format the argument list for a function, or get all the information you need to display a detailed traceback.

There are four main kinds of services provided by this module: type checking, getting source code, inspecting classes and functions, and examining the interpreter stack.

Types and members

The **`getmembers()`** function retrieves the members of an object such as a class or module. The functions whose names begin with “is” are mainly provided as convenient choices for the second argument to **`getmembers()`**. They also help you determine when you can expect to find the following special attributes (see Import-related attributes on module objects for module attributes):

Type	Attribute	Description
class	__doc__	documentation string
	__name__	name with which this class was defined
	__qualname__	qualified name

Type	Attribute	Description
	__module__	name of module in which this class was defined
	__type_params__	A tuple containing the type parameters of a generic class
method	__doc__	documentation string
	__name__	name with which this method was defined
	__qualname__	qualified name
	__func__	function object containing implementation of method
	__self__	instance to which this method is bound, or `None`
	__module__	name of module in which this method was defined
function	__doc__	documentation string
	__name__	name with which this function was defined
	__qualname__	qualified name

Type	Attribute	Description
	__code__	code object containing compiled function bytecode
	__defaults__	tuple of any default values for positional or keyword parameters
	__kwdefaults__	mapping of any default values for keyword-only parameters
	__globals__	global namespace in which this function was defined
	__builtins__	builtins namespace
	__annotations__	mapping of parameters names to annotations; `"return"` key is reserved for return annotations.
	__type_params__	A tuple containing the type parameters of a generic function
	__module__	name of module in which this function was defined
traceback	tb_frame	frame object at this level
	tb_lasti	index of last attempted instruction in bytecode
	tb_lineno	current line number in Python source code

Type	Attribute	Description
	tb_next	next inner traceback object (called by this level)
frame	f_back	next outer frame object (this frame’s caller)
	f_builtins	builtins namespace seen by this frame
	f_code	code object being executed in this frame
	f_globals	global namespace seen by this frame
	f_lasti	index of last attempted instruction in bytecode
	f_lineno	current line number in Python source code
	f_locals	local namespace seen by this frame
	f_generator	returns the generator or coroutine object that owns this frame, or `None` if the frame is of a regular function
	f_trace	tracing function for this frame, or `None`
	f_trace_lines	indicate whether a tracing event is triggered for each source source line

Type	Attribute	Description
	f_trace_opcodes	indicate whether per-opcode events are requested
	clear()	used to clear all references to local variables
code	co_argcount	number of arguments (not including keyword only arguments, * or ** args)
	co_code	string of raw compiled bytecode
	co_cellvars	tuple of names of cell variables (referenced by containing scopes)
	co_consts	tuple of constants used in the bytecode
	co_filename	name of file in which this code object was created
	co_firstlineno	number of first line in Python source code
	co_flags	bitmap of `CO_*` flags, read more here
	co_lnotab	encoded mapping of line numbers to bytecode indices
	co_freevars	tuple of names of free variables (referenced via a function’s closure)

Type	Attribute	Description
	co_posonlyargcount	number of positional only arguments
	co_kwonlyargcount	number of keyword only arguments (not including ** arg)
	co_name	name with which this code object was defined
	co_qualname	fully qualified name with which this code object was defined
	co_names	tuple of names other than arguments and function locals
	co_nlocals	number of local variables
	co_stacksize	virtual machine stack space required
	co_varnames	tuple of names of arguments and local variables
	co_lines()	returns an iterator that yields successive bytecode ranges
	co_positions()	returns an iterator of source code positions for each bytecode instruction
	replace()	returns a copy of the code object with new values

Type	**Attribute**	**Description**
generator	__name__	name
	__qualname__	qualified name
	gi_frame	frame
	gi_running	is the generator running?
	gi_suspended	is the generator suspended?
	gi_code	code
	gi_yieldfrom	object being iterated by `yield from`, or `None`
async generator	__name__	name
	__qualname__	qualified name
	ag_await	object being awaited on, or `None`
	ag_frame	frame
	ag_running	is the generator running?
	ag_suspended	is the generator suspended?

Type	Attribute	Description
	ag_code	code
coroutine	__name__	name
	__qualname__	qualified name
	cr_await	object being awaited on, or `None`
	cr_frame	frame
	cr_running	is the coroutine running?
	cr_suspended	is the coroutine suspended?
	cr_code	code
	cr_origin	where coroutine was created, or `None`. See `sys.set_coroutine_origin_tracking_depth()`
builtin	__doc__	documentation string
	__name__	original name of this function or method
	__qualname__	qualified name
	__self__	instance to which a method is bound, or `None`

Type	Attribute	Description

Changed in version 3.5: Add `__qualname__` and `gi_yieldfrom` attributes to generators.

The `__name__` attribute of generators is now set from the function name, instead of the code name, and it can now be modified.

Changed in version 3.7: Add `cr_origin` attribute to coroutines.

Changed in version 3.10: Add `__builtins__` attribute to functions.

Changed in version 3.11: Add `gi_suspended` attribute to generators.

Changed in version 3.11: Add `cr_suspended` attribute to coroutines.

Changed in version 3.12: Add `ag_suspended` attribute to async generators.

Changed in version 3.14: Add `f_generator` attribute to frames.

inspect.getmembers(*object*[, *predicate*])

Return all the members of an object in a list of `(name, value)` pairs sorted by name. If the optional *predicate* argument—which will be called with the `value` object of each member—is supplied, only members for which the predicate returns a true value are included.

Note

`getmembers()` will only return class attributes defined in the metaclass when the argument is a class and those attributes have been listed in the metaclass' custom `__dir__()`.

inspect.getmembers_static(*object*[, *predicate*])

Return all the members of an object in a list of `(name, value)` pairs sorted by name without triggering dynamic lookup via the descriptor protocol, __getattr__ or __getattribute__. Optionally, only return members that satisfy a given predicate.

Note

`getmembers_static()` may not be able to retrieve all members that getmembers can fetch (like dynamically created attributes) and may find members that getmembers can't (like descriptors that raise AttributeError). It can also return descriptor objects instead of instance members in some cases.

Added in version 3.11.

inspect.getmodulename(*path*)

Return the name of the module named by the file *path*, without including the names of enclosing packages. The file extension is checked against all of the entries in `importlib.machinery.all_suffixes()`. If it matches, the final path component is returned with the extension removed. Otherwise, `None` is returned.

Note that this function *only* returns a meaningful name for actual Python modules - paths that potentially refer to Python packages will still return `None`.

Changed in version 3.3: The function is based directly on `importlib`.

inspect.ismodule(*object*)

Return `True` if the object is a module.

inspect.isclass(*object*)

Return `True` if the object is a class, whether built-in or created in Python code.

inspect.ismethod(*object*)

Return `True` if the object is a bound method written in Python.

inspect.ispackage(*object*)

Return `True` if the object is a package.

Added in version 3.14.

inspect.isfunction(*object*)

Return `True` if the object is a Python function, which includes functions created by a lambda expression.

inspect.isgeneratorfunction(*object*)

Return `True` if the object is a Python generator function.

Changed in version 3.8: Functions wrapped in **`functools.partial()`** now return `True` if the wrapped function is a Python generator function.

Changed in version 3.13: Functions wrapped in **`functools.partialmethod()`** now return `True` if the wrapped function is a Python generator function.

inspect.isgenerator(*object*)

Return `True` if the object is a generator.

inspect.iscoroutinefunction(*object*)

Return `True` if the object is a coroutine function (a function defined with an **`async def`** syntax), a **`functools.partial()`** wrapping a coroutine function, or a sync function marked with **`markcoroutinefunction()`**.

Added in version 3.5.

Changed in version 3.8: Functions wrapped in **`functools.partial()`** now return `True` if the wrapped function is a coroutine function.

Changed in version 3.12: Sync functions marked with **`markcoroutinefunction()`** now return `True`.

Changed in version 3.13: Functions wrapped in **`functools.partialmethod()`** now return `True` if the wrapped function is a coroutine function.

inspect.markcoroutinefunction(*func*)

Decorator to mark a callable as a coroutine function if it would not otherwise be detected by **`iscoroutinefunction()`**.

This may be of use for sync functions that return a coroutine, if the function is passed to an API that requires **`iscoroutinefunction()`**.

When possible, using an **`async def`** function is preferred. Also acceptable is calling the function and testing the return with **`iscoroutine()`**.

Added in version 3.12.

inspect.iscoroutine(*object*)

Return `True` if the object is a coroutine created by an **`async def`** function.

Added in version 3.5.

inspect.isawaitable(*object*)

Return `True` if the object can be used in **`await`** expression.

Can also be used to distinguish generator-based coroutines from regular generators:

```
import types

def gen():
    yield
@types.coroutine
def gen_coro():
    yield

assert not isawaitable(gen())
assert isawaitable(gen_coro())
```

Added in version 3.5.

inspect.isasyncgenfunction(*object*)

Return `True` if the object is an asynchronous generator function, for example:

```
>>> async def agen():
...     yield 1
...
>>> inspect.isasyncgenfunction(agen)
True
```

Added in version 3.6.

Changed in version 3.8: Functions wrapped in **`functools.partial()`** now return `True` if the wrapped function is an asynchronous generator function.

Changed in version 3.13: Functions wrapped in **`functools.partialmethod()`** now return `True` if the wrapped function is a coroutine function.

inspect.isasyncgen(*object*)

Return `True` if the object is an asynchronous generator iterator created by an asynchronous generator function.

Added in version 3.6.

inspect.istraceback(*object*)

Return `True` if the object is a traceback.

inspect.isframe(*object*)

Return `True` if the object is a frame.

inspect.iscode(*object*)

Return `True` if the object is a code.

inspect.isbuiltin(*object*)

Return `True` if the object is a built-in function or a bound built-in method.

inspect.ismethodwrapper(*object*)

Return `True` if the type of object is a **`MethodWrapperType`**.

These are instances of **`MethodWrapperType`**, such as **`__str__()`**, **`__eq__()`** and **`__repr__()`**.

Added in version 3.11.

inspect.isroutine(*object*)

Return `True` if the object is a user-defined or built-in function or method.

inspect.isabstract(*object*)

Return `True` if the object is an abstract base class.

inspect.ismethoddescriptor(*object*)

Return `True` if the object is a method descriptor, but not if **`ismethod()`**, **`isclass()`**, **`isfunction()`** or **`isbuiltin()`** are true.

This, for example, is true of `int.__add__`. An object passing this test has a **`__get__()`** method, but not a **`__set__()`** method or a **`__delete__()`** method. Beyond that, the set of attributes varies. A **`__name__`** attribute is usually sensible, and **`__doc__`** often is.

Methods implemented via descriptors that also pass one of the other tests return `False` from the `ismethoddescriptor()` test, simply because the other tests promise more – you can, e.g., count on having the `__func__` attribute (etc) when an object passes `ismethod()`.

Changed in version 3.13: This function no longer incorrectly reports objects with `__get__()` and `__delete__()`, but not `__set__()`, as being method descriptors (such objects are data descriptors, not method descriptors).

inspect.isdatadescriptor(*object*)

Return `True` if the object is a data descriptor.

Data descriptors have a `__set__` or a `__delete__` method. Examples are properties (defined in Python), getsets, and members. The latter two are defined in C and there are more specific tests available for those types, which is robust across Python implementations. Typically, data descriptors will also have `__name__` and `__doc__` attributes (properties, getsets, and members have both of these attributes), but this is not guaranteed.

inspect.isgetsetdescriptor(*object*)

Return `True` if the object is a getset descriptor.

CPython implementation detail: getsets are attributes defined in extension modules via `PyGetSetDef` structures. For Python implementations without such types, this method will always return `False`.

inspect.ismemberdescriptor(*object*)

Return `True` if the object is a member descriptor.

CPython implementation detail: Member descriptors are attributes defined in extension modules via `PyMemberDef` structures. For Python implementations without such types, this method will always return `False`.

Retrieving source code

inspect.getdoc(*object*)

Get the documentation string for an object, cleaned up with `cleandoc()`. If the documentation string for an object is not provided and the object is a class, a method, a property or a descriptor, retrieve the documentation string from the inheritance hierarchy. Return `None` if the documentation string is invalid or missing.

Changed in version 3.5: Documentation strings are now inherited if not overridden.

inspect.getcomments(*object*)

Return in a single string any lines of comments immediately preceding the object’s source code (for a class, function, or method), or at the top of the Python source file (if the object is a module). If the object’s source code is unavailable, return `None`. This could happen if the object has been defined in C or the interactive shell.

inspect.getfile(*object*)

Return the name of the (text or binary) file in which an object was defined. This will fail with a `TypeError` if the object is a built-in module, class, or function.

inspect.getmodule(*object*)

Try to guess which module an object was defined in. Return `None` if the module cannot be determined.

inspect.getsourcefile(*object*)

Return the name of the Python source file in which an object was defined or `None` if no way can be identified to get the source. This will fail with a `TypeError` if the object is a built-in module, class, or function.

inspect.getsourcelines(*object*)

Return a list of source lines and starting line number for an object. The argument may be a module, class, method, function, traceback, frame, or code object. The source code is returned as a list of the lines corresponding to the object and the line number indicates where in the original source file the first line of code was found. An `OSError` is raised if the source code cannot be retrieved. A `TypeError` is raised if the object is a built-in module, class, or function.

Changed in version 3.3: `OSError` is raised instead of `IOError`, now an alias of the former.

inspect.getsource(*object*)

Return the text of the source code for an object. The argument may be a module, class, method, function, traceback, frame, or code object. The source code is returned as a single string. An `OSError` is raised if the source code cannot be retrieved. A `TypeError` is raised if the object is a built-in module, class, or function.

Changed in version 3.3: `OSError` is raised instead of `IOError`, now an alias of the former.

inspect.cleandoc(*doc*)

Clean up indentation from docstrings that are indented to line up with blocks of code.

All leading whitespace is removed from the first line. Any leading whitespace that can be uniformly removed from the second line onwards is removed. Empty lines at the beginning and end are subsequently removed. Also, all tabs are expanded to spaces.

Introspecting callables with the Signature object

Added in version 3.3.

The `Signature` object represents the call signature of a callable object and its return annotation. To retrieve a **`Signature`** object, use the **`signature()`** function.

inspect.signature(*callable*, ***, *follow_wrapped=True*, *globals=None*, *locals=None*, *eval_str=False*, *annotation_format=Format.VALUE*)

Return a `Signature` object for the given *callable*:

```
>>> from inspect import signature
>>> def foo(a, *, b:int, **kwargs):
...     pass

>>> sig = signature(foo)

>>> str(sig)
'(a, *, b: int, **kwargs)'

>>> str(sig.parameters['b'])
'b: int'
```

```
>>> sig.parameters['b'].annotation
<class 'int'>
```

Accepts a wide range of Python callables, from plain functions and classes to **`functools.partial()`** objects.

If some of the annotations are strings (e.g., because `from __future__ import annotations` was used), **`signature()`** will attempt to automatically un-stringize the annotations using **`annotationlib.get_annotations()`**. The *globals*, *locals*, and *eval_str* parameters are passed into **`annotationlib.get_annotations()`** when resolving the annotations; see the documentation for **`annotationlib.get_annotations()`** for instructions on how to use these parameters. A member of the **`annotationlib.Format`** enum can be passed to the *annotation_format* parameter to control the format of the returned annotations. For example, use `annotation_format=annotationlib.Format.STRING` to return annotations in string format.

Raises **`ValueError`** if no signature can be provided, and **`TypeError`** if that type of object is not supported. Also, if the annotations are stringized, and *eval_str* is not false, the `eval()` call(s) to un-stringize the annotations in **`annotationlib.get_annotations()`** could potentially raise any kind of exception.

A slash (/) in the signature of a function denotes that the parameters prior to it are positional-only. For more info, see the FAQ entry on positional-only parameters.

Changed in version 3.5: The *follow_wrapped* parameter was added. Pass `False` to get a signature of *callable* specifically (`callable.__wrapped__` will not be used to unwrap decorated callables.)

Changed in version 3.10: The *globals*, *locals*, and *eval_str* parameters were added.

Changed in version 3.14: The *annotation_format* parameter was added.

Note

Some callables may not be introspectable in certain implementations of Python. For example, in CPython, some built-in functions defined in C provide no metadata about their arguments.

CPython implementation detail: If the passed object has a `__signature__` attribute, we may use it to create the signature. The exact semantics are an implementation detail and are subject to unannounced changes. Consult the source code for current semantics.

class inspect.Signature(*parameters=None*, ***, *return_annotation=Signature.empty*)

A `Signature` object represents the call signature of a function and its return annotation. For each parameter accepted by the function it stores a `Parameter` object in its `parameters` collection.

The optional *parameters* argument is a sequence of `Parameter` objects, which is validated to check that there are no parameters with duplicate names, and that the parameters are in the right order, i.e. positional-only first, then positional-or-keyword, and that parameters with defaults follow parameters without defaults.

The optional *return_annotation* argument can be an arbitrary Python object. It represents the “return” annotation of the callable.

`Signature` objects are *immutable*. Use `Signature.replace()` or `copy.replace()` to make a modified copy.

Changed in version 3.5: `Signature` objects are now picklable and hashable.

empty

A special class-level marker to specify absence of a return annotation.

parameters

An ordered mapping of parameters’ names to the corresponding `Parameter` objects. Parameters appear in strict definition order, including keyword-only parameters.

Changed in version 3.7: Python only explicitly guaranteed that it preserved the declaration order of keyword-only parameters as of version 3.7, although in practice this order had always been preserved in Python 3.

return_annotation

The “return” annotation for the callable. If the callable has no “return” annotation, this attribute is set to `Signature.empty`.

bind(**args*, ***kwargs*)

Create a mapping from positional and keyword arguments to parameters. Returns `BoundArguments` if `*args` and `**kwargs` match the signature, or raises a `TypeError`.

bind_partial(**args*, ***kwargs*)

Works the same way as `Signature.bind()`, but allows the omission of some required arguments (mimics `functools.partial()` behavior.) Returns `BoundArguments`, or raises a `TypeError` if the passed arguments do not match the signature.

replace(**[, parameters][, return_annotation]*)

Create a new `Signature` instance based on the instance `replace()` was invoked on. It is possible to pass different *parameters* and/or *return_annotation* to override the corresponding properties of the base signature. To remove `return_annotation` from the copied **`Signature`**, pass in `Signature.empty`.

```
>>> def test(a, b):
...     pass
...
>>> sig = signature(test)
>>> new_sig = sig.replace(return_annotation="new return anno")
>>> str(new_sig)
"(a, b) -> 'new return anno'"
```

`Signature` objects are also supported by the generic function `copy.replace()`.

format(***, *max_width=None*, *quote_annotation_strings=True*)

Create a string representation of the `Signature` object.

If *max_width* is passed, the method will attempt to fit the signature into lines of at most *max_width* characters. If the signature is longer than *max_width*, all parameters will be on separate lines.

If *quote_annotation_strings* is False, annotations in the signature are displayed without opening and closing quotation marks if they are strings. This is useful if the signature was created with the `STRING` format or if `from __future__ import annotations` was used.

Added in version 3.13.

Changed in version 3.14: The *unquote_annotations* parameter was added.

classmethod from_callable(*obj*, *, *follow_wrapped=True*, *globals=None*, *locals=None*, *eval_str=False*)

Return a **`Signature`** (or its subclass) object for a given callable *obj*.

This method simplifies subclassing of **`Signature`**:

```
class MySignature(Signature):
    pass
sig = MySignature.from_callable(sum)
assert isinstance(sig, MySignature)
```

Its behavior is otherwise identical to that of **`signature()`**.

Added in version 3.5.

Changed in version 3.10: The *globals*, *locals*, and *eval_str* parameters were added.

class inspect.Parameter(*name*, *kind*, *, *default=Parameter.empty*, *annotation=Parameter.empty*)

`Parameter` objects are *immutable*. Instead of modifying a **`Parameter`** object, you can use **`Parameter.replace()`** or **`copy.replace()`** to create a modified copy.

Changed in version 3.5: Parameter objects are now picklable and hashable.

empty

A special class-level marker to specify absence of default values and annotations.

name

The name of the parameter as a string. The name must be a valid Python identifier.

CPython implementation detail: CPython generates implicit parameter names of the form `.0` on the code objects used to implement comprehensions and generator expressions.

Changed in version 3.6: These parameter names are now exposed by this module as names like `implicit0`.

default

The default value for the parameter. If the parameter has no default value, this attribute is set to [`Parameter.empty`].

annotation

The annotation for the parameter. If the parameter has no annotation, this attribute is set to [`Parameter.empty`].

kind

Describes how argument values are bound to the parameter. The possible values are accessible via [`Parameter`] (like `Parameter.KEYWORD_ONLY`), and support comparison and ordering, in the following order:

Name	**Meaning**
POSITIONAL_ONLY	Value must be supplied as a positional argument. Positional only parameters are those which appear before a `/` entry (if present) in a Python function definition.
POSITIONAL_OR_KEYWORD	Value may be supplied as either a keyword or positional argument (this is the standard binding behaviour for functions implemented in Python.)
VAR_POSITIONAL	A tuple of positional arguments that aren’t bound to any other parameter. This corresponds to a `*args` parameter in a Python function definition.
KEYWORD_ONLY	Value must be supplied as a keyword argument. Keyword only parameters are those which appear after a `*` or `*args` entry in a Python function definition.

Name	**Meaning**
VAR_KEYWORD	A dict of keyword arguments that aren’t bound to any other parameter. This corresponds to a `**kwargs` parameter in a Python function definition.

Example: print all keyword-only arguments without default values:

```
>>> def foo(a, b, *, c, d=10):
...     pass

>>> sig = signature(foo)
>>> for param in sig.parameters.values():
...     if (param.kind == param.KEYWORD_ONLY and
...                        param.default is param.empty):
...         print('Parameter:', param)
Parameter: c
```

kind.description

Describes an enum value of [`Parameter.kind`].

Added in version 3.8.

Example: print all descriptions of arguments:

```
>>> def foo(a, b, *, c, d=10):
...     pass

>>> sig = signature(foo)
>>> for param in sig.parameters.values():
...     print(param.kind.description)
positional or keyword
positional or keyword
keyword-only
keyword-only
```

replace(**[, name][, kind][, default][, annotation]*)

Create a new [`Parameter`] instance based on the instance replaced was invoked on. To override a **`Parameter`** attribute, pass the corresponding argument. To remove a default value or/and an annotation from a **`Parameter`**, pass [`Parameter.empty`].

```
>>> from inspect import Parameter
>>> param = Parameter('foo', Parameter.KEYWORD_ONLY, default=42)
>>> str(param)
'foo=42'

>>> str(param.replace()) # Will create a shallow copy of 'param'
'foo=42'

>>> str(param.replace(default=Parameter.empty, annotation='spam'))
"foo: 'spam'"
```

`Parameter` objects are also supported by the generic function `copy.replace()`.

Changed in version 3.4: In Python 3.3 `Parameter` objects were allowed to have `name` set to `None` if their `kind` was set to `POSITIONAL_ONLY`. This is no longer permitted.

class inspect.BoundArguments

Result of a `Signature.bind()` or `Signature.bind_partial()` call. Holds the mapping of arguments to the function's parameters.

arguments

A mutable mapping of parameters' names to arguments' values. Contains only explicitly bound arguments. Changes in `arguments` will reflect in `args` and `kwargs`.

Should be used in conjunction with `Signature.parameters` for any argument processing purposes.

Note

Arguments for which `Signature.bind()` or `Signature.bind_partial()` relied on a default value are skipped. However, if needed, use `BoundArguments.apply_defaults()` to add them.

Changed in version 3.9: `arguments` is now of type `dict`. Formerly, it was of type `collections.OrderedDict`.

args

A tuple of positional arguments values. Dynamically computed from the `arguments` attribute.

kwargs

A dict of keyword arguments values. Dynamically computed from the `arguments` attribute. Arguments that can be passed positionally are included in `args` instead.

signature

A reference to the parent `Signature` object.

apply_defaults()

Set default values for missing arguments.

For variable-positional arguments (`*args`) the default is an empty tuple.

For variable-keyword arguments (`**kwargs`) the default is an empty dict.

```
>>> def foo(a, b='ham', *args): pass
>>> ba = inspect.signature(foo).bind('spam')
>>> ba.apply_defaults()
>>> ba.arguments
{'a': 'spam', 'b': 'ham', 'args': ()}
```

Added in version 3.5.

The `args` and `kwargs` properties can be used to invoke functions:

```
def test(a, *, b):
    ...

sig = signature(test)
ba = sig.bind(10, b=20)
test(*ba.args, **ba.kwargs)
```

See also

PEP 362 [https://peps.python.org/pep-0362/] - Function Signature Object.

The detailed specification, implementation details and examples.

Classes and functions

inspect.getclasstree(*classes*, *unique=False*)

Arrange the given list of classes into a hierarchy of nested lists. Where a nested list appears, it contains classes derived from the class whose entry immediately precedes the list. Each entry is a 2-tuple containing a class and a tuple of its base classes. If the *unique* argument is true, exactly one entry appears in the returned structure for each class in the given list. Otherwise, classes using multiple inheritance and their descendants will appear multiple times.

inspect.getfullargspec(*func*)

Get the names and default values of a Python function's parameters. A named tuple is returned:

`FullArgSpec(args, varargs, varkw, defaults, kwonlyargs, kwonlydefaults, annotations)`

args is a list of the positional parameter names. *varargs* is the name of the `*` parameter or `None` if arbitrary positional arguments are not accepted. *varkw* is the name of the `**` parameter or `None` if arbitrary keyword arguments are not accepted. *defaults* is an *n*-tuple of default argument values corresponding to the last *n* positional parameters, or `None` if there are no such defaults defined. *kwonlyargs* is a list of keyword-only parameter names in declaration order. *kwonlydefaults* is a dictionary mapping parameter names from *kwonlyargs* to the default values used if no argument is supplied. *annotations* is a dictionary mapping parameter names to annotations. The special key `"return"` is used to report the function return value annotation (if any).

Note that **`signature()`** and Signature Object provide the recommended API for callable introspection, and support additional behaviours (like positional-only arguments) that are sometimes encountered in extension module APIs. This function is retained primarily for use in code that needs to maintain compatibility with the Python 2 `inspect` module API.

Changed in version 3.4: This function is now based on **`signature()`**, but still ignores `__wrapped__` attributes and includes the already bound first parameter in the signature output for bound methods.

Changed in version 3.6: This method was previously documented as deprecated in favour of **`signature()`** in Python 3.5, but that decision has been reversed in order to restore a clearly supported standard interface for single-source Python 2/3 code migrating away from the legacy **`getargspec()`** API.

Changed in version 3.7: Python only explicitly guaranteed that it preserved the declaration order of keyword-only parameters as of version 3.7, although in practice this order had always been preserved in Python 3.

inspect.getargvalues(*frame*)

Get information about arguments passed into a particular frame. A named tuple `ArgInfo(args, varargs, keywords, locals)` is returned. *args* is a list of the argument names. *varargs* and *keywords* are the names of the `*` and `**` arguments or `None`. *locals* is the locals dictionary of the given frame.

Note

This function was inadvertently marked as deprecated in Python 3.5.

inspect.formatargvalues(*args*[, *varargs*, *varkw*, *locals*, *formatarg*, *formatvarargs*, *formatvarkw*, *formatvalue*])

Format a pretty argument spec from the four values returned by **`getargvalues()`**. The format* arguments are the corresponding optional formatting functions that are called to turn names and values into strings.

Note

This function was inadvertently marked as deprecated in Python 3.5.

inspect.getmro(*cls*)

Return a tuple of class cls’s base classes, including cls, in method resolution order. No class appears more than once in this tuple. Note that the method resolution order depends on cls’s type. Unless a very peculiar user-defined metatype is in use, cls will be the first element of the tuple.

inspect.getcallargs(*func*, /, **args*, ***kwds*)

Bind the *args* and *kwds* to the argument names of the Python function or method *func*, as if it was called with them. For bound methods, bind also the first argument (typically named `self`) to the associated instance. A dict is returned, mapping the argument names (including the names of the `*` and `**` arguments, if any) to their values from *args* and *kwds*. In case of invoking *func* incorrectly, i.e. whenever `func(*args, **kwds)` would raise an exception because of incompatible signature, an exception of the same type and the same or similar message is raised. For example:

```
>>> from inspect import getcallargs
>>> def f(a, b=1, *pos, **named):
...     pass
...
>>> getcallargs(f, 1, 2, 3) == {'a': 1, 'named': {}, 'b': 2, 'pos':
(3,)}
True
>>> getcallargs(f, a=2, x=4) == {'a': 2, 'named': {'x': 4}, 'b': 1,
'pos': ()}
True
>>> getcallargs(f)
Traceback (most recent call last):
...
TypeError: f() missing 1 required positional argument: 'a'
```

Added in version 3.2.

Deprecated since version 3.5: Use **`Signature.bind()`** and **`Signature.bind_partial()`** instead.

inspect.getclosurevars(*func*)

Get the mapping of external name references in a Python function or method *func* to their current values. A named tuple `ClosureVars(nonlocals, globals, builtins, unbound)` is returned. *nonlocals* maps referenced names to lexical closure variables, *globals* to the function’s module globals and *builtins* to the builtins visible from the function body. *unbound* is the set of names referenced in the function that could not be resolved at all given the current module globals and builtins.

`TypeError` is raised if *func* is not a Python function or method.

Added in version 3.3.

inspect.unwrap(*func*, ***, *stop=None*)

Get the object wrapped by *func*. It follows the chain of **`__wrapped__`** attributes returning the last object in the chain.

stop is an optional callback accepting an object in the wrapper chain as its sole argument that allows the unwrapping to be terminated early if the callback returns a true value. If the callback never returns a true value, the last object in the chain is returned as usual. For example, **`signature()`** uses this to stop unwrapping if any object in the chain has a `__signature__` attribute defined.

`ValueError` is raised if a cycle is encountered.

Added in version 3.4.

inspect.get_annotations(*obj*, *, *globals=None*, *locals=None*, *eval_str=False*, *format=annotationlib.Format.VALUE*)

Compute the annotations dict for an object.

This is an alias for **`annotationlib.get_annotations()`**; see the documentation of that function for more information.

Caution

This function may execute arbitrary code contained in annotations. See Security implications of introspecting annotations for more information.

Added in version 3.10.

Changed in version 3.14: This function is now an alias for **`annotationlib.get_annotations()`**. Calling it as `inspect.get_annotations` will continue to work.

The interpreter stack

Some of the following functions return **`FrameInfo`** objects. For backwards compatibility these objects allow tuple-like operations on all attributes except `positions`. This behavior is considered deprecated and may be removed in the future.

class inspect.FrameInfo

frame

The frame object that the record corresponds to.

filename

The file name associated with the code being executed by the frame this record corresponds to.

lineno

The line number of the current line associated with the code being executed by the frame this record corresponds to.

function

The function name that is being executed by the frame this record corresponds to.

code_context

A list of lines of context from the source code that’s being executed by the frame this record corresponds to.

index

The index of the current line being executed in the `code_context` list.

positions

A `dis.Positions` object containing the start line number, end line number, start column offset, and end column offset associated with the instruction being executed by the frame this record corresponds to.

Changed in version 3.5: Return a named tuple instead of a `tuple`.

Changed in version 3.11: **`FrameInfo`** is now a class instance (that is backwards compatible with the previous named tuple).

class inspect.Traceback

filename

The file name associated with the code being executed by the frame this traceback corresponds to.

lineno

The line number of the current line associated with the code being executed by the frame this traceback corresponds to.

function

The function name that is being executed by the frame this traceback corresponds to.

code_context

A list of lines of context from the source code that’s being executed by the frame this traceback corresponds to.

index
: The index of the current line being executed in the `code_context` list.

positions
: A `dis.Positions` object containing the start line number, end line number, start column offset, and end column offset associated with the instruction being executed by the frame this traceback corresponds to.

Changed in version 3.11: `Traceback` is now a class instance (that is backwards compatible with the previous named tuple).

Note

Keeping references to frame objects, as found in the first element of the frame records these functions return, can cause your program to create reference cycles. Once a reference cycle has been created, the lifespan of all objects which can be accessed from the objects which form the cycle can become much longer even if Python's optional cycle detector is enabled. If such cycles must be created, it is important to ensure they are explicitly broken to avoid the delayed destruction of objects and increased memory consumption which occurs.

Though the cycle detector will catch these, destruction of the frames (and local variables) can be made deterministic by removing the cycle in a `finally` clause. This is also important if the cycle detector was disabled when Python was compiled or using `gc.disable()`. For example:

```
def handle_stackframe_without_leak():
    frame = inspect.currentframe()
    try:
        # do something with the frame
    finally:
        del frame
```

If you want to keep the frame around (for example to print a traceback later), you can also break reference cycles by using the `frame.clear()` method.

The optional *context* argument supported by most of these functions specifies the number of lines of context to return, which are centered around the current line.

inspect.getframeinfo(*frame*, *context=1*)

Get information about a frame or traceback object. A **`Traceback`** object is returned.

Changed in version 3.11: A **`Traceback`** object is returned instead of a named tuple.

inspect.getouterframes(*frame*, *context=1*)

Get a list of **`FrameInfo`** objects for a frame and all outer frames. These frames represent the calls that lead to the creation of *frame*. The first entry in the returned list represents *frame*; the last entry represents the outermost call on *frame*'s stack.

Changed in version 3.5: A list of named tuples `FrameInfo(frame, filename, lineno, function, code_context, index)` is returned.

Changed in version 3.11: A list of **`FrameInfo`** objects is returned.

inspect.getinnerframes(*traceback*, *context=1*)

Get a list of **`FrameInfo`** objects for a traceback's frame and all inner frames. These frames represent calls made as a consequence of *frame*. The first entry in the list represents *traceback*; the last entry represents where the exception was raised.

Changed in version 3.5: A list of named tuples `FrameInfo(frame, filename, lineno, function, code_context, index)` is returned.

Changed in version 3.11: A list of **`FrameInfo`** objects is returned.

inspect.currentframe()

Return the frame object for the caller's stack frame.

CPython implementation detail: This function relies on Python stack frame support in the interpreter, which isn't guaranteed to exist in all implementations of Python. If running in an implementation without Python stack frame support this function returns `None`.

inspect.stack(*context=1*)

Return a list of `FrameInfo` objects for the caller's stack. The first entry in the returned list represents the caller; the last entry represents the outermost call on the stack.

Changed in version 3.5: A list of named tuples `FrameInfo(frame, filename, lineno, function, code_context, index)` is returned.

Changed in version 3.11: A list of `FrameInfo` objects is returned.

inspect.trace(*context=1*)

Return a list of `FrameInfo` objects for the stack between the current frame and the frame in which an exception currently being handled was raised in. The first entry in the list represents the caller; the last entry represents where the exception was raised.

Changed in version 3.5: A list of named tuples `FrameInfo(frame, filename, lineno, function, code_context, index)` is returned.

Changed in version 3.11: A list of `FrameInfo` objects is returned.

Fetching attributes statically

Both `getattr()` and `hasattr()` can trigger code execution when fetching or checking for the existence of attributes. Descriptors, like properties, will be invoked and `__getattr__()` and `__getattribute__()` may be called.

For cases where you want passive introspection, like documentation tools, this can be inconvenient. `getattr_static()` has the same signature as `getattr()` but avoids executing code when it fetches attributes.

inspect.getattr_static(*obj*, *attr*, *default=None*)

Retrieve attributes without triggering dynamic lookup via the descriptor protocol, `__getattr__()` or `__getattribute__()`.

Note: this function may not be able to retrieve all attributes that getattr can fetch (like dynamically created attributes) and may find attributes that getattr can't (like descriptors that raise AttributeError). It can also return descriptors objects instead of instance members.

If the instance `__dict__` is shadowed by another member (for example a property) then this function will be unable to find instance members.

Added in version 3.2.

`getattr_static()` does not resolve descriptors, for example slot descriptors or getset descriptors on objects implemented in C. The descriptor object is returned instead of the underlying attribute.

You can handle these with code like the following. Note that for arbitrary getset descriptors invoking these may trigger code execution:

```
# example code for resolving the builtin descriptor types
class _foo:
    __slots__ = ['foo']

slot_descriptor = type(_foo.foo)
getset_descriptor = type(type(open(__file__)).name)
wrapper_descriptor = type(str.__dict__['__add__'])
descriptor_types = (slot_descriptor, getset_descriptor, wrapper_descriptor)

result = getattr_static(some_object, 'foo')
if type(result) in descriptor_types:
    try:
        result = result.__get__()
    except AttributeError:
        # descriptors can raise AttributeError to
        # indicate there is no underlying value
        # in which case the descriptor itself will
        # have to do
        pass
```

Current State of Generators, Coroutines, and Asynchronous Generators

When implementing coroutine schedulers and for other advanced uses of generators, it is useful to determine whether a generator is currently executing, is waiting to start or resume or execution, or has already terminated. `getgeneratorstate()` allows the current state of a generator to be determined easily.

inspect.getgeneratorstate(*generator*)

Get current state of a generator-iterator.

Possible states are:

- GEN_CREATED: Waiting to start execution.
- GEN_RUNNING: Currently being executed by the interpreter.

- GEN_SUSPENDED: Currently suspended at a yield expression.
- GEN_CLOSED: Execution has completed.

Added in version 3.2.

inspect.getcoroutinestate(*coroutine*)

Get current state of a coroutine object. The function is intended to be used with coroutine objects created by `async def` functions, but will accept any coroutine-like object that has `cr_running` and `cr_frame` attributes.

Possible states are:

- CORO_CREATED: Waiting to start execution.
- CORO_RUNNING: Currently being executed by the interpreter.
- CORO_SUSPENDED: Currently suspended at an await expression.
- CORO_CLOSED: Execution has completed.

Added in version 3.5.

inspect.getasyncgenstate(*agen*)

Get current state of an asynchronous generator object. The function is intended to be used with asynchronous iterator objects created by `async def` functions which use the `yield` statement, but will accept any asynchronous generator-like object that has `ag_running` and `ag_frame` attributes.

Possible states are:

- AGEN_CREATED: Waiting to start execution.
- AGEN_RUNNING: Currently being executed by the interpreter.
- AGEN_SUSPENDED: Currently suspended at a yield expression.
- AGEN_CLOSED: Execution has completed.

Added in version 3.12.

The current internal state of the generator can also be queried. This is mostly useful for testing purposes, to ensure that internal state is being updated as expected:

inspect.getgeneratorlocals(*generator*)

Get the mapping of live local variables in *generator* to their current values. A dictionary is returned that maps from variable names to values. This is the equivalent of calling `locals()` in the body of the generator, and all the same caveats apply.

If *generator* is a generator with no currently associated frame, then an empty dictionary is returned. `TypeError` is raised if *generator* is not a Python generator object.

CPython implementation detail: This function relies on the generator exposing a Python stack frame for introspection, which isn’t guaranteed to be the case in all implementations of Python. In such cases, this function will always return an empty dictionary.

Added in version 3.3.

inspect.getcoroutinelocals(*coroutine*)

This function is analogous to `getgeneratorlocals()`, but works for coroutine objects created by `async def` functions.

Added in version 3.5.

inspect.getasyncgenlocals(*agen*)

This function is analogous to `getgeneratorlocals()`, but works for asynchronous generator objects created by `async def` functions which use the `yield` statement.

Added in version 3.12.

Code Objects Bit Flags

Python code objects have a `co_flags` attribute, which is a bitmap of the following flags:

inspect.CO_OPTIMIZED

The code object is optimized, using fast locals.

inspect.CO_NEWLOCALS

If set, a new dict will be created for the frame’s `f_locals` when the code object is executed.

inspect.CO_VARARGS

The code object has a variable positional parameter (`*args`-like).

inspect.CO_VARKEYWORDS

The code object has a variable keyword parameter (`**kwargs`-like).

inspect.CO_NESTED

The flag is set when the code object is a nested function.

inspect.CO_GENERATOR

The flag is set when the code object is a generator function, i.e. a generator object is returned when the code object is executed.

inspect.CO_COROUTINE

The flag is set when the code object is a coroutine function. When the code object is executed it returns a coroutine object. See **PEP 492** [https://peps.python.org/pep-0492/] for more details.

Added in version 3.5.

inspect.CO_ITERABLE_COROUTINE

The flag is used to transform generators into generator-based coroutines. Generator objects with this flag can be used in `await` expression, and can `yield from` coroutine objects. See **PEP 492** [https://peps.python.org/pep-0492/] for more details.

Added in version 3.5.

inspect.CO_ASYNC_GENERATOR

The flag is set when the code object is an asynchronous generator function. When the code object is executed it returns an asynchronous generator object. See **PEP 525** [https://peps.python.org/pep-0525/] for more details.

Added in version 3.6.

inspect.CO_HAS_DOCSTRING

The flag is set when there is a docstring for the code object in the source code. If set, it will be the first item in `co_consts`.

Added in version 3.14.

inspect.CO_METHOD

The flag is set when the code object is a function defined in class scope.

Added in version 3.14.

Note

The flags are specific to CPython, and may not be defined in other Python implementations. Furthermore, the flags are an implementation detail, and can be removed or deprecated in future Python releases. It’s recommended to use public APIs from the `inspect` module for any introspection needs.

Buffer flags

class inspect.BufferFlags

This is an `enum.IntFlag` that represents the flags that can be passed to the `__buffer__()` method of objects implementing the buffer protocol.

The meaning of the flags is explained at Buffer request types.

SIMPLE

WRITABLE

FORMAT

ND

STRIDES

C_CONTIGUOUS

F_CONTIGUOUS

ANY_CONTIGUOUS

INDIRECT

CONTIG

CONTIG_RO

STRIDED

STRIDED_RO

RECORDS

RECORDS_RO

FULL

FULL_RO

READ

WRITE

Added in version 3.12.

Command-line interface

The `inspect` module also provides a basic introspection capability from the command line.

By default, accepts the name of a module and prints the source of that module. A class or function within the module can be printed instead by appended a colon and the qualified name of the target object.

--details

Print information about the specified object rather than the source code

`annotationlib` — Functionality for introspecting annotations

Added in version 3.14.

Source code: Lib/annotationlib.py
[https://github.com/python/cpython/tree/3.14/Lib/annotationlib.py]

The **`annotationlib`** module provides tools for introspecting annotations on modules, classes, and functions.

Annotations are lazily evaluated and often contain forward references to objects that are not yet defined when the annotation is created. This module provides a set of low-level tools that can be used to retrieve annotations in a reliable way, even in the presence of forward references and other edge cases.

This module supports retrieving annotations in three main formats (see **`Format`**), each of which works best for different use cases:

- **`VALUE`** evaluates the annotations and returns their value. This is most straightforward to work with, but it may raise errors, for example if the annotations contain references to undefined names.
- **`FORWARDREF`** returns **`ForwardRef`** objects for annotations that cannot be resolved, allowing you to inspect the annotations without evaluating them. This is useful

when you need to work with annotations that may contain unresolved forward references.
- `STRING` returns the annotations as a string, similar to how it would appear in the source file. This is useful for documentation generators that want to display annotations in a readable way.

The `get_annotations()` function is the main entry point for retrieving annotations. Given a function, class, or module, it returns an annotations dictionary in the requested format. This module also provides functionality for working directly with the annotate function that is used to evaluate annotations, such as `get_annotate_from_class_namespace()` and `call_annotate_function()`, as well as the `call_evaluate_function()` function for working with evaluate functions.

Caution

Most functionality in this module can execute arbitrary code; see the security section for more information.

See also

PEP 649 [https://peps.python.org/pep-0649/] proposed the current model for how annotations work in Python.

PEP 749 [https://peps.python.org/pep-0749/] expanded on various aspects of **PEP 649** [https://peps.python.org/pep-0649/] and introduced the `annotationlib` module.

Annotations Best Practices provides best practices for working with annotations.

typing-extensions [https://pypi.org/project/typing-extensions/] provides a backport of `get_annotations()` that works on earlier versions of Python.

Annotation semantics

The way annotations are evaluated has changed over the history of Python 3, and currently still depends on a future import. There have been execution models for annotations:

- *Stock semantics* (default in Python 3.0 through 3.13; see **PEP 3107** [https://peps.python.org/pep-3107/] and **PEP 526** [https://peps.python.org/pep-0526/]): Annotations are evaluated eagerly, as they are encountered in the source code.
- *Stringified annotations* (used with `from __future__ import annotations` in Python 3.7 and newer; see **PEP 563** [https://peps.python.org/pep-0563/]): Annotations are stored as strings only.
- *Deferred evaluation* (default in Python 3.14 and newer; see **PEP 649** [https://peps.python.org/pep-0649/] and **PEP 749** [https://peps.python.org/pep-0749/]): Annotations are evaluated lazily, only when they are accessed.

As an example, consider the following program:

```
def func(a: Cls) -> None:
    print(a)

class Cls: pass

print(func.__annotations__)
```

This will behave as follows:

- Under stock semantics (Python 3.13 and earlier), it will throw a `NameError` at the line where `func` is defined, because `Cls` is an undefined name at that point.

- Under stringified annotations (if `from __future__ import annotations` is used), it will print `{'a': 'Cls', 'return': 'None'}`.
- Under deferred evaluation (Python 3.14 and later), it will print `{'a': <class 'Cls'>, 'return': None}`.

Stock semantics were used when function annotations were first introduced in Python 3.0 (by **PEP 3107** [https://peps.python.org/pep-3107/]) because this was the simplest, most obvious way to implement annotations. The same execution model was used when variable annotations were introduced in Python 3.6 (by **PEP 526** [https://peps.python.org/pep-0526/]). However, stock semantics caused problems when using annotations as type hints, such as a need to refer to names that are not yet defined when the annotation is encountered. In addition, there were performance problems with executing annotations at module import time. Therefore, in Python 3.7, **PEP 563** [https://peps.python.org/pep-0563/] introduced the ability to store annotations as strings using the `from __future__ import annotations` syntax. The plan at the time was to eventually make this behavior the default, but a problem appeared: stringified annotations are more difficult to process for those who introspect annotations at runtime. An alternative proposal, **PEP 649** [https://peps.python.org/pep-0649/], introduced the third execution model, deferred evaluation, and was implemented in Python 3.14. Stringified annotations are still used if `from __future__ import annotations` is present, but this behavior will eventually be removed.

Classes

class annotationlib.Format

An `IntEnum` describing the formats in which annotations can be returned. Members of the enum, or their equivalent integer values, can be passed to `get_annotations()` and other functions in this module, as well as to `__annotate__` functions.

VALUE = *1*

Values are the result of evaluating the annotation expressions.

VALUE_WITH_FAKE_GLOBALS = *2*

Special value used to signal that an annotate function is being evaluated in a special environment with fake globals. When passed this value, annotate functions should either return the same value as for the `Format.VALUE` format, or raise `NotImplementedError` to signal that they do not support execution in this environment. This format is only used internally and should not be passed to the functions in this module.

FORWARDREF = *3*

Values are real annotation values (as per `Format.VALUE` format) for defined values, and `ForwardRef` proxies for undefined values. Real objects may contain references to `ForwardRef` proxy objects.

STRING = *4*

Values are the text string of the annotation as it appears in the source code, up to modifications including, but not restricted to, whitespace normalizations and constant values optimizations.

The exact values of these strings may change in future versions of Python.

Added in version 3.14.

class annotationlib.ForwardRef

A proxy object for forward references in annotations.

Instances of this class are returned when the **`FORWARDREF`** format is used and annotations contain a name that cannot be resolved. This can happen when a forward reference is used in an annotation, such as when a class is referenced before it is defined.

__forward_arg__

A string containing the code that was evaluated to produce the **`ForwardRef`**. The string may not be exactly equivalent to the original source.

evaluate(*, *owner=None*, *globals=None*, *locals=None*, *type_params=None*, *format=Format.VALUE*)

Evaluate the forward reference, returning its value.

If the *format* argument is **`VALUE`** (the default), this method may throw an exception, such as **`NameError`**, if the forward reference refers to a name that cannot be resolved. The arguments to this method can be used to provide bindings for names that would otherwise be undefined. If the *format* argument is **`FORWARDREF`**, the method will never throw an exception, but may return a **`ForwardRef`** instance. For example, if the forward reference object contains the code `list[undefined]`, where `undefined` is a name that is not defined, evaluating it with the **`FORWARDREF`** format will return `list[ForwardRef('undefined')]`. If the *format* argument is **`STRING`**, the method will return **`__forward_arg__`**.

The *owner* parameter provides the preferred mechanism for passing scope information to this method. The owner of a **`ForwardRef`** is the object that contains the annotation from which the **`ForwardRef`** derives, such as a module object, type object, or function object.

The *globals*, *locals*, and *type_params* parameters provide a more precise mechanism for influencing the names that are available when the **`ForwardRef`** is evaluated. *globals* and *locals* are passed to **`eval()`**, representing the global and local namespaces in which the name is evaluated. The *type_params* parameter is relevant for objects created using the native syntax for generic classes and functions. It is a tuple of type parameters that are in scope while the forward reference is being evaluated. For example, if evaluating a **`ForwardRef`** retrieved from an annotation found in the class namespace of a generic class `C`, *type_params* should be set to `C.__type_params__`.

`ForwardRef` instances returned by **`get_annotations()`** retain references to information about the scope they originated from, so calling this method with no further arguments may be sufficient to evaluate such objects. **`ForwardRef`** instances created by other means may not have any information about their scope, so passing arguments to this method may be necessary to evaluate them successfully.

If no *owner*, *globals*, *locals*, or *type_params* are provided and the **`ForwardRef`** does not contain information about its origin, empty globals and locals dictionaries are used.

Added in version 3.14.

Functions

annotationlib.annotations_to_string(*annotations*)

Convert an annotations dict containing runtime values to a dict containing only strings. If the values are not already strings, they are converted using **`type_repr()`**. This is meant as a helper for user-provided annotate functions that support the **`STRING`** format but do not have access to the code creating the annotations.

For example, this is used to implement the **`STRING`** for **`typing.TypedDict`** classes created through the functional syntax:

```
>>> from typing import TypedDict
>>> Movie = TypedDict("movie", {"name": str, "year": int})
>>> get_annotations(Movie, format=Format.STRING)
{'name': 'str', 'year': 'int'}
```

Added in version 3.14.

annotationlib.call_annotate_function(*annotate*, *format*, *, *owner=None*)

Call the annotate function *annotate* with the given *format*, a member of the **`Format`** enum, and return the annotations dictionary produced by the function.

This helper function is required because annotate functions generated by the compiler for functions, classes, and modules only support the **`VALUE`** format when called directly. To support other formats, this function calls the annotate function in a special environment that allows it to produce annotations in the other formats. This is a useful building block when implementing functionality that needs to partially evaluate annotations while a class is being constructed.

owner is the object that owns the annotation function, usually a function, class, or module. If provided, it is used in the `FORWARDREF` format to produce a `ForwardRef` object that carries more information.

See also

PEP 649 [https://peps.python.org/pep-0649/#the-stringizer-and-the-fake-globals-environment] contains an explanation of the implementation technique used by this function.

Added in version 3.14.

annotationlib.call_evaluate_function(*evaluate*, *format*, *, *owner=None*)

Call the evaluate function *evaluate* with the given *format*, a member of the `Format` enum, and return the value produced by the function. This is similar to `call_annotate_function()`, but the latter always returns a dictionary mapping strings to annotations, while this function returns a single value.

This is intended for use with the evaluate functions generated for lazily evaluated elements related to type aliases and type parameters:

- `typing.TypeAliasType.evaluate_value()`, the value of type aliases
- `typing.TypeVar.evaluate_bound()`, the bound of type variables
- `typing.TypeVar.evaluate_constraints()`, the constraints of type variables
- `typing.TypeVar.evaluate_default()`, the default value of type variables
- `typing.ParamSpec.evaluate_default()`, the default value of parameter specifications
- `typing.TypeVarTuple.evaluate_default()`, the default value of type variable tuples

owner is the object that owns the evaluate function, such as the type alias or type variable object.

format can be used to control the format in which the value is returned:

```
>>> type Alias = undefined
>>> call_evaluate_function(Alias.evaluate_value,
Format.VALUE)
Traceback (most recent call last):
...
NameError: name 'undefined' is not defined
>>> call_evaluate_function(Alias.evaluate_value,
Format.FORWARDREF)
ForwardRef('undefined')
>>> call_evaluate_function(Alias.evaluate_value,
Format.STRING)
'undefined'
```

Added in version 3.14.

annotationlib.get_annotate_from_class_namespace(*namespace*)

Retrieve the annotate function from a class namespace dictionary *namespace*. Return **`None`** if the namespace does not contain an annotate function. This is primarily useful before the class has been fully created (e.g., in a metaclass); after the class exists, the annotate function can be retrieved with `cls.__annotate__`. See below for an example using this function in a metaclass.

Added in version 3.14.

annotationlib.get_annotations(*obj*, *, *globals=None*, *locals=None*, *eval_str=False*, *format=Format.VALUE*)

Compute the annotations dict for an object.

obj may be a callable, class, module, or other object with `__annotate__` or `__annotations__` attributes. Passing any other object raises `TypeError`.

The *format* parameter controls the format in which annotations are returned, and must be a member of the `Format` enum or its integer equivalent. The different formats work as follows:

- VALUE: `object.__annotations__` is tried first; if that does not exist, the `object.__annotate__` function is called if it exists.
- FORWARDREF: If `object.__annotations__` exists and can be evaluated successfully, it is used; otherwise, the `object.__annotate__` function is called. If it does not exist either, `object.__annotations__` is tried again and any error from accessing it is re-raised.
  - When calling `object.__annotate__` it is first called with `FORWARDREF`. If this is not implemented, it will then check if `VALUE_WITH_FAKE_GLOBALS` is supported and use that in the fake globals environment. If neither of these formats are supported, it will fall back to using `VALUE`. If `VALUE` fails, the error from this call will be raised.
- STRING: If `object.__annotate__` exists, it is called first; otherwise, `object.__annotations__` is used and stringified using `annotations_to_string()`.
  - When calling `object.__annotate__` it is first called with `STRING`. If this is not implemented, it will then check if `VALUE_WITH_FAKE_GLOBALS` is supported and use that in the fake globals environment. If neither of these formats are supported, it will fall back to using `VALUE` with the result converted using `annotations_to_string()`. If `VALUE` fails, the error from this call will be raised.

Returns a dict. **`get_annotations()`** returns a new dict every time it’s called; calling it twice on the same object will return two different but equivalent dicts.

This function handles several details for you:

- If *eval_str* is true, values of type **`str`** will be un-stringized using `eval()`. This is intended for use with stringized annotations (`from __future__ import annotations`). It is an error to set *eval_str* to true with formats other than **`Format.VALUE`**.
- If *obj* doesn’t have an annotations dict, returns an empty dict. (Functions and methods always have an annotations dict; classes, modules, and other types of callables may not.)
- Ignores inherited annotations on classes, as well as annotations on metaclasses. If a class doesn’t have its own annotations dict, returns an empty dict.
- All accesses to object members and dict values are done using `getattr()` and `dict.get()` for safety.

eval_str controls whether or not values of type **`str`** are replaced with the result of calling `eval()` on those values:

- If eval_str is true, `eval()` is called on values of type **`str`**. (Note that **`get_annotations()`** doesn’t catch exceptions; if `eval()` raises an exception, it will unwind the stack past the **`get_annotations()`** call.)
- If *eval_str* is false (the default), values of type **`str`** are unchanged.

globals and *locals* are passed in to `eval()`; see the documentation for `eval()` for more information. If *globals* or *locals* is **`None`**, this function may replace that value with a context-specific default, contingent on `type(obj)`:

- If *obj* is a module, *globals* defaults to `obj.__dict__`.

- If *obj* is a class, *globals* defaults to `sys.modules[obj.__module__].__dict__` and *locals* defaults to the *obj* class namespace.
- If *obj* is a callable, *globals* defaults to `obj.__globals__`, although if *obj* is a wrapped function (using `functools.update_wrapper()`) or a `functools.partial` object, it is unwrapped until a non-wrapped function is found.

Calling **`get_annotations()`** is best practice for accessing the annotations dict of any object. See Annotations Best Practices for more information on annotations best practices.

```
>>> def f(a: int, b: str) -> float:
...     pass
>>> get_annotations(f)
{'a': <class 'int'>, 'b': <class 'str'>, 'return': <class
'float'>}
```

Added in version 3.14.

annotationlib.type_repr(*value*)

Convert an arbitrary Python value to a format suitable for use by the `STRING` format. This calls `repr()` for most objects, but has special handling for some objects, such as type objects.

This is meant as a helper for user-provided annotate functions that support the `STRING` format but do not have access to the code creating the annotations. It can also be used to provide a user-friendly string representation for other objects that contain values that are commonly encountered in annotations.

Added in version 3.14.

Recipes

Using annotations in a metaclass

A metaclass may want to inspect or even modify the annotations in a class body during class creation. Doing so requires retrieving annotations from the class namespace dictionary. For classes created with `from __future__ import annotations`, the annotations will be in the `__annotations__` key of the dictionary. For other classes with annotations, **`get_annotate_from_class_namespace()`** can be used to get the annotate function, and **`call_annotate_function()`** can be used to call it and retrieve the annotations. Using the **`FORWARDREF`** format will usually be best, because this allows the annotations to refer to names that cannot yet be resolved when the class is created.

To modify the annotations, it is best to create a wrapper annotate function that calls the original annotate function, makes any necessary adjustments, and returns the result.

Below is an example of a metaclass that filters out all **`typing.ClassVar`** annotations from the class and puts them in a separate attribute:

```
import annotationlib
import typing

class ClassVarSeparator(type):
   def __new__(mcls, name, bases, ns):
      if "__annotations__" in ns:  # from __future__ import
annotations
         annotations = ns["__annotations__"]
         classvar_keys = {
            key for key, value in annotations.items()
            # Use string comparison for simplicity; a more
robust solution
            # could use annotationlib.ForwardRef.evaluate
            if value.startswith("ClassVar")
         }
         classvars = {key: annotations[key] for key in
classvar_keys}
         ns["__annotations__"] = {
            key: value for key, value in annotations.items()
            if key not in classvar_keys
         }
         wrapped_annotate = None
      elif annotate :=
annotationlib.get_annotate_from_class_namespace(ns):
         annotations = annotationlib.call_annotate_function(
            annotate, format=annotationlib.Format.FORWARDREF
         )
         classvar_keys = {
            key for key, value in annotations.items()
            if typing.get_origin(value) is typing.ClassVar
         }
         classvars = {key: annotations[key] for key in
classvar_keys}

         def wrapped_annotate(format):
            annos =
annotationlib.call_annotate_function(annotate, format,
owner=typ)
            return {key: value for key, value in annos.items()
if key not in classvar_keys}

      else:  # no annotations
         classvars = {}
         wrapped_annotate = None
      typ = super().__new__(mcls, name, bases, ns)

      if wrapped_annotate is not None:
         # Wrap the original __annotate__ with a wrapper that
removes ClassVars
         typ.__annotate__ = wrapped_annotate
      typ.classvars = classvars  # Store the ClassVars in a
separate attribute
      return typ
```

Limitations of the `STRING` format

The `STRING` format is meant to approximate the source code of the annotation, but the implementation strategy used means that it is not always possible to recover the exact source code.

First, the stringifier of course cannot recover any information that is not present in the compiled code, including comments, whitespace, parenthesization, and operations that get simplified by the compiler.

Second, the stringifier can intercept almost all operations that involve names looked up in some scope, but it cannot intercept operations that operate fully on constants. As a corollary, this also means it is not safe to request the `STRING` format on untrusted code: Python is powerful enough that it is possible to achieve arbitrary code execution even with no access to any globals or builtins. For example:

```
>>> def f(x: (1).__class__.__base__.__subclasses__()
[-1].__init__.__builtins__["print"]("Hello world")): pass
...
>>> annotationlib.get_annotations(f,
format=annotationlib.Format.STRING)
Hello world
{'x': 'None'}
```

Note

This particular example works as of the time of writing, but it relies on implementation details and is not guaranteed to work in the future.

Among the different kinds of expressions that exist in Python, as represented by the `ast` module, some expressions are supported, meaning that the `STRING` format can generally recover the original source code; others are unsupported, meaning that they may result in incorrect output or an error.

The following are supported (sometimes with caveats):

- `ast.BinOp`
- `ast.UnaryOp`
    - `ast.Invert` (`~`), `ast.UAdd` (`+`), and `ast.USub` (`-`) are supported
    - `ast.Not` (`not`) is not supported
- `ast.Dict` (except when using `**` unpacking)
- `ast.Set`
- `ast.Compare`
    - `ast.Eq` and `ast.NotEq` are supported
    - `ast.Lt`, `ast.LtE`, `ast.Gt`, and `ast.GtE` are supported, but the operand may be flipped
    - `ast.Is`, `ast.IsNot`, `ast.In`, and `ast.NotIn` are not supported
- `ast.Call` (except when using `**` unpacking)
- `ast.Constant` (though not the exact representation of the constant; for example, escape sequences in strings are lost; hexadecimal numbers are converted to decimal)
- `ast.Attribute` (assuming the value is not a constant)
- `ast.Subscript` (assuming the value is not a constant)
- `ast.Starred` (`*` unpacking)
- `ast.Name`
- `ast.List`
- `ast.Tuple`
- `ast.Slice`

The following are unsupported, but throw an informative error when encountered by the stringifier:

- `ast.FormattedValue` (f-strings; error is not detected if conversion specifiers like `!r` are used)
- `ast.JoinedStr` (f-strings)

The following are unsupported and result in incorrect output:

- `ast.BoolOp` (`and` and `or`)
- `ast.IfExp`
- `ast.Lambda`
- `ast.ListComp`
- `ast.SetComp`
- `ast.DictComp`
- `ast.GeneratorExp`

The following are disallowed in annotation scopes and therefore not relevant:

- `ast.NamedExpr` (`:=`)
- `ast.Await`
- `ast.Yield`
- `ast.YieldFrom`

Limitations of the `FORWARDREF` format

The `FORWARDREF` format aims to produce real values as much as possible, with anything that cannot be resolved replaced with `ForwardRef` objects. It is affected by broadly the same Limitations as the `STRING` format: annotations that perform operations on literals or that use unsupported expression types may raise exceptions when evaluated using the `FORWARDREF` format.

Below are a few examples of the behavior with unsupported expressions:

```
>>> from annotationlib import get_annotations, Format
>>> def zerodiv(x: 1 / 0): ...
>>> get_annotations(zerodiv, format=Format.STRING)
Traceback (most recent call last):
  ...
```

```
ZeroDivisionError: division by zero
>>> get_annotations(zerodiv, format=Format.FORWARDREF)
Traceback (most recent call last):
  ...
ZeroDivisionError: division by zero
>>> def ifexp(x: 1 if y else 0): ...
>>> get_annotations(ifexp, format=Format.STRING)
{'x': '1'}
```

Security implications of introspecting annotations

Much of the functionality in this module involves executing code related to annotations, which can then do arbitrary things. For example, `get_annotations()` may call an arbitrary annotate function, and `ForwardRef.evaluate()` may call `eval()` on an arbitrary string. Code contained in an annotation might make arbitrary system calls, enter an infinite loop, or perform any other operation. This is also true for any access of the `__annotations__` attribute, and for various functions in the `typing` module that work with annotations, such as `typing.get_type_hints()`.

Any security issue arising from this also applies immediately after importing code that may contain untrusted annotations: importing code can always cause arbitrary operations to be performed. However, it is unsafe to accept strings or other input from an untrusted source and pass them to any of the APIs for introspecting annotations, for example by editing an `__annotations__` dictionary or directly creating a `ForwardRef` object.

`site` — Site-specific configuration hook

Source code: Lib/site.py
[https://github.com/python/cpython/tree/3.14/Lib/site.py]

This module is automatically imported during initialization. The automatic import can be suppressed using the interpreter's **`-S`** option.

Importing this module normally appends site-specific paths to the module search path and adds callables, including **`help()`** to the built-in namespace. However, Python startup option **`-S`** blocks this and this module can be safely imported with no automatic modifications to the module search path or additions to the builtins. To explicitly trigger the usual site-specific additions, call the **`main()`** function.

Changed in version 3.3: Importing the module used to trigger paths manipulation even when using **`-S`**.

It starts by constructing up to four directories from a head and a tail part. For the head part, it uses `sys.prefix` and `sys.exec_prefix`; empty heads are skipped. For the tail part, it uses the empty string and then `lib/site-packages` (on Windows) or `lib/pythonX.Y[t]/site-packages` (on Unix and macOS). (The optional suffix "t" indicates the free threading build, and is appended if `"t"` is present in the **`sys.abiflags`** constant.) For each of the distinct head-tail combinations, it sees if it refers to an existing directory, and if so, adds it to `sys.path` and also inspects the newly added path for configuration files.

Changed in version 3.5: Support for the “site-python” directory has been removed.

Changed in version 3.13: On Unix, Free threading Python installations are identified by the “t” suffix in the version-specific directory name, such as `lib/python3.13t/`.

Changed in version 3.14: **`site`** is no longer responsible for updating **`sys.prefix`** and **`sys.exec_prefix`** on Virtual Environments. This is now done during the path initialization. As a result, under Virtual Environments, **`sys.prefix`** and **`sys.exec_prefix`** no longer depend on the **`site`** initialization, and are therefore unaffected by **`-S`**.

When running under a virtual environment, the `pyvenv.cfg` file in **`sys.prefix`** is checked for site-specific configurations. If the `include-system-site-packages` key exists and is set to `true` (case-insensitive), the system-level prefixes will be searched for site-packages, otherwise they won’t.

A path configuration file is a file whose name has the form *`name`*`.pth` and exists in one of the four directories mentioned above; its contents are additional items (one per line) to be added to `sys.path`. Non-existing items are never added to `sys.path`, and no check is made that the item refers to a directory rather than a file. No item is added to `sys.path` more than once. Blank lines and lines beginning with `#` are skipped. Lines starting with `import` (followed by space or tab) are executed.

Note

An executable line in a `.pth` file is run at every Python startup, regardless of whether a particular module is actually going to be used. Its impact should thus be kept to a minimum. The primary intended purpose of executable lines is to make the corresponding module(s) importable (load 3rd-party import hooks, adjust **PATH** etc). Any other initialization is supposed to be done upon a module's actual import, if and when it happens. Limiting a code chunk to a single line is a deliberate measure to discourage putting anything more complex here.

Changed in version 3.13: The `.pth` files are now decoded by UTF-8 at first and then by the locale encoding if it fails.

For example, suppose `sys.prefix` and `sys.exec_prefix` are set to `/usr/local`. The Python X.Y library is then installed in `/usr/local/lib/pythonX.Y`. Suppose this has a subdirectory `/usr/local/lib/pythonX.Y/site-packages` with three subsubdirectories, `foo`, `bar` and `spam`, and two path configuration files, `foo.pth` and `bar.pth`. Assume `foo.pth` contains the following:

```
# foo package configuration

foo
bar
bletch
```

and `bar.pth` contains:

```
# bar package configuration

bar
```

Then the following version-specific directories are added to `sys.path`, in this order:

```
/usr/local/lib/pythonX.Y/site-packages/bar
/usr/local/lib/pythonX.Y/site-packages/foo
```

Note that `bletch` is omitted because it doesn’t exist; the `bar` directory precedes the `foo` directory because `bar.pth` comes alphabetically before `foo.pth`; and `spam` is omitted because it is not mentioned in either path configuration file.

`sitecustomize`

After these path manipulations, an attempt is made to import a module named **`sitecustomize`**, which can perform arbitrary site-specific customizations. It is typically created by a system administrator in the site-packages directory. If this import fails with an **`ImportError`** or its subclass exception, and the exception’s **`name`** attribute equals to `'sitecustomize'`, it is silently ignored. If Python is started without output streams available, as with `pythonw.exe` on Windows (which is used by default to start IDLE), attempted output from **`sitecustomize`** is ignored. Any other exception causes a silent and perhaps mysterious failure of the process.

`usercustomize`

After this, an attempt is made to import a module named **`usercustomize`**, which can perform arbitrary user-specific customizations, if **`ENABLE_USER_SITE`** is true. This file is intended to be created in the user site-packages directory (see below), which is part of `sys.path` unless disabled by **`-s`**. If this import fails with an **`ImportError`** or its subclass exception, and the exception’s **`name`** attribute equals to `'usercustomize'`, it is silently ignored.

Note that for some non-Unix systems, `sys.prefix` and `sys.exec_prefix` are empty, and the path manipulations are skipped; however the import of **`sitecustomize`** and **`usercustomize`** is still attempted.

Readline configuration

On systems that support `readline`, this module will also import and configure the `rlcompleter` module, if Python is started in interactive mode and without the `-S` option. The default behavior is enable tab-completion and to use `~/.python_history` as the history save file. To disable it, delete (or override) the `sys.__interactivehook__` attribute in your `sitecustomize` or `usercustomize` module or your `PYTHONSTARTUP` file.

Changed in version 3.4: Activation of rlcompleter and history was made automatic.

Module contents

site.PREFIXES

A list of prefixes for site-packages directories.

site.ENABLE_USER_SITE

Flag showing the status of the user site-packages directory. `True` means that it is enabled and was added to `sys.path`. `False` means that it was disabled by user request (with `-s` or `PYTHONNOUSERSITE`). `None` means it was disabled for security reasons (mismatch between user or group id and effective id) or by an administrator.

site.USER_SITE

Path to the user site-packages for the running Python. Can be `None` if `getusersitepackages()` hasn't been called yet. Default value is `~/.local/lib/pythonX.Y[t]/site-packages` for UNIX and non-framework macOS builds, `~/Library/Python/X.Y/lib/python/site-packages` for macOS framework builds, and `%APPDATA%\Python\PythonXY\site-packages` on Windows. The optional "t" indicates the free-threaded build. This directory is a site directory, which means that `.pth` files in it will be processed.

site.USER_BASE

Path to the base directory for the user site-packages. Can be `None` if **`getuserbase()`** hasn’t been called yet. Default value is `~/.local` for UNIX and macOS non-framework builds, `~/Library/Python/X.Y` for macOS framework builds, and `%APPDATA%\Python` for Windows. This value is used to compute the installation directories for scripts, data files, Python modules, etc. for the user installation scheme. See also **`PYTHONUSERBASE`**.

site.main()

Adds all the standard site-specific directories to the module search path. This function is called automatically when this module is imported, unless the Python interpreter was started with the **`-S`** flag.

Changed in version 3.3: This function used to be called unconditionally.

site.addsitedir(*sitedir*, *known_paths=None*)

Add a directory to sys.path and process its `.pth` files. Typically used in **`sitecustomize`** or **`usercustomize`** (see above).

site.getsitepackages()

Return a list containing all global site-packages directories.

Added in version 3.2.

site.getuserbase()

Return the path of the user base directory, **`USER_BASE`**. If it is not initialized yet, this function will also set it, respecting `PYTHONUSERBASE`.

Added in version 3.2.

site.getusersitepackages()

Return the path of the user-specific site-packages directory, `USER_SITE`. If it is not initialized yet, this function will also set it, respecting `USER_BASE`. To determine if the user-specific site-packages was added to `sys.path` `ENABLE_USER_SITE` should be used.

Added in version 3.2.

Command-line interface

The `site` module also provides a way to get the user directories from the command line:

```
$ python -m site --user-site
/home/user/.local/lib/python3.11/site-packages
```

If it is called without arguments, it will print the contents of `sys.path` on the standard output, followed by the value of `USER_BASE` and whether the directory exists, then the same thing for `USER_SITE`, and finally the value of `ENABLE_USER_SITE`.

--user-base

Print the path to the user base directory.

--user-site

Print the path to the user site-packages directory.

If both options are given, user base and user site will be printed (always in this order), separated by `os.pathsep`.

If any option is given, the script will exit with one of these values: `0` if the user site-packages directory is enabled, `1` if it was disabled by the user, `2` if it is disabled for security reasons or by an administrator, and a value greater than 2 if there is an error.

See also

- **PEP 370** [https://peps.python.org/pep-0370/] – Per user site-packages directory
- The initialization of the sys.path module search path – The initialization of **`sys.path`**.

Custom Python Interpreters

The modules described in this chapter allow writing interfaces similar to Python’s interactive interpreter. If you want a Python interpreter that supports some special feature in addition to the Python language, you should look at the **`code`** module. (The **`codeop`** module is lower-level, used to support compiling a possibly incomplete chunk of Python code.)

The full list of modules described in this chapter is:

- **`code`** — Interpreter base classes
  - Interactive Interpreter Objects
  - Interactive Console Objects
- **`codeop`** — Compile Python code

`code` — Interpreter base classes

Source code: Lib/code.py [https://github.com/python/cpython/tree/3.14/Lib/code.py]

The `code` module provides facilities to implement read-eval-print loops in Python. Two classes and convenience functions are included which can be used to build applications which provide an interactive interpreter prompt.

class code.InteractiveInterpreter(*locals=None*)

This class deals with parsing and interpreter state (the user’s namespace); it does not deal with input buffering or prompting or input file naming (the filename is always passed in explicitly). The optional *locals* argument specifies a mapping to use as the namespace in which code will be executed; it defaults to a newly created dictionary with key `'__name__'` set to `'__console__'` and key `'__doc__'` set to `None`.

Note that functions and classes objects created under an **`InteractiveInterpreter`** instance will belong to the namespace specified by *locals*. They are only pickleable if *locals* is the namespace of an existing module.

class code.InteractiveConsole(*locals=None*, *filename='<console>'*, *local_exit=False*)

Closely emulate the behavior of the interactive Python interpreter. This class builds on **`InteractiveInterpreter`** and adds prompting using the familiar `sys.ps1` and `sys.ps2`, and input buffering. If *local_exit* is true, `exit()` and `quit()` in the console will not raise **`SystemExit`**, but instead return to the calling code.

Changed in version 3.13: Added *local_exit* parameter.

code.interact(*banner=None*, *readfunc=None*, *local=None*, *exitmsg=None*, *local_exit=False*)

Convenience function to run a read-eval-print loop. This creates a new instance of **`InteractiveConsole`** and sets *readfunc* to be used as the **`InteractiveConsole.raw_input()`** method, if provided. If *local* is provided, it is passed to the **`InteractiveConsole`** constructor for use as the default namespace for the interpreter loop. If *local_exit* is provided, it is passed to the **`InteractiveConsole`** constructor. The **`interact()`** method of the instance is then run with *banner* and *exitmsg* passed as the banner and exit message to use, if provided. The console object is discarded after use.

Changed in version 3.6: Added *exitmsg* parameter.

Changed in version 3.13: Added *local_exit* parameter.

code.compile_command(*source*, *filename*='*<input>*', *symbol*='*single*')

This function is useful for programs that want to emulate Python’s interpreter main loop (a.k.a. the read-eval-print loop). The tricky part is to determine when the user has entered an incomplete command that can be completed by entering more text (as opposed to a complete command or a syntax error). This function *almost* always makes the same decision as the real interpreter main loop.

source is the source string; *filename* is the optional filename from which source was read, defaulting to `'<input>'`; and *symbol* is the optional grammar start symbol, which should be `'single'` (the default), `'eval'` or `'exec'`.

Returns a code object (the same as `compile(source, filename, symbol)`) if the command is complete and valid; `None` if the command is incomplete; raises **`SyntaxError`** if the command is complete and contains a syntax error, or raises **`OverflowError`** or **`ValueError`** if the command contains an invalid literal.

Interactive Interpreter Objects

InteractiveInterpreter.runsource(*source*, *filename*='*<input>*', *symbol*='*single*')

Compile and run some source in the interpreter. Arguments are the same as for **`compile_command()`**; the default for *filename* is `'<input>'`, and for *symbol* is `'single'`. One of several things can happen:

- The input is incorrect; **`compile_command()`** raised an exception (**`SyntaxError`** or **`OverflowError`**). A syntax traceback will be printed by calling the **`showsyntaxerror()`** method. **`runsource()`** returns `False`.
- The input is incomplete, and more input is required; **`compile_command()`** returned `None`. **`runsource()`** returns `True`.
- The input is complete; **`compile_command()`** returned a code object. The code is executed by calling the **`runcode()`** (which also handles run-time exceptions, except for **`SystemExit`**). **`runsource()`** returns `False`.

The return value can be used to decide whether to use `sys.ps1` or `sys.ps2` to prompt the next line.

InteractiveInterpreter.runcode(*code*)

Execute a code object. When an exception occurs, `showtraceback()` is called to display a traceback. All exceptions are caught except `SystemExit`, which is allowed to propagate.

A note about `KeyboardInterrupt`: this exception may occur elsewhere in this code, and may not always be caught. The caller should be prepared to deal with it.

InteractiveInterpreter.showsyntaxerror(*filename=None*)

Display the syntax error that just occurred. This does not display a stack trace because there isn't one for syntax errors. If *filename* is given, it is stuffed into the exception instead of the default filename provided by Python's parser, because it always uses `'<string>'` when reading from a string. The output is written by the `write()` method.

InteractiveInterpreter.showtraceback()

Display the exception that just occurred. We remove the first stack item because it is within the interpreter object implementation. The output is written by the `write()` method.

Changed in version 3.5: The full chained traceback is displayed instead of just the primary traceback.

InteractiveInterpreter.write(*data*)

Write a string to the standard error stream (`sys.stderr`). Derived classes should override this to provide the appropriate output handling as needed.

Interactive Console Objects

The `InteractiveConsole` class is a subclass of `InteractiveInterpreter`, and so offers all the methods of the interpreter objects as well as the following additions.

InteractiveConsole.interact(*banner=None*, *exitmsg=None*)

> Closely emulate the interactive Python console. The optional *banner* argument specify the banner to print before the first interaction; by default it prints a banner similar to the one printed by the standard Python interpreter, followed by the class name of the console object in parentheses (so as not to confuse this with the real interpreter – since it's so close!).
>
> The optional *exitmsg* argument specifies an exit message printed when exiting. Pass the empty string to suppress the exit message. If *exitmsg* is not given or `None`, a default message is printed.
>
> *Changed in version 3.4:* To suppress printing any banner, pass an empty string.
>
> *Changed in version 3.6:* Print an exit message when exiting.

InteractiveConsole.push(*line*)

> Push a line of source text to the interpreter. The line should not have a trailing newline; it may have internal newlines. The line is appended to a buffer and the interpreter's `runsource()` method is called with the concatenated contents of the buffer as source. If this indicates that the command was executed or invalid, the buffer is reset; otherwise, the command is incomplete, and the buffer is left as it was after the line was appended. The return value is `True` if more input is required, `False` if the line was dealt with in some way (this is the same as **`runsource()`**).

InteractiveConsole.resetbuffer()

Remove any unhandled source text from the input buffer.

InteractiveConsole.raw_input(*prompt*='')

Write a prompt and read a line. The returned line does not include the trailing newline. When the user enters the EOF key sequence, **`EOFError`** is raised. The base implementation reads from `sys.stdin`; a subclass may replace this with a different implementation.

`codeop` — Compile Python code

Source code: Lib/codeop.py
[https://github.com/python/cpython/tree/3.14/Lib/codeop.py]

The **`codeop`** module provides utilities upon which the Python read-eval-print loop can be emulated, as is done in the **`code`** module. As a result, you probably don't want to use the module directly; if you want to include such a loop in your program you probably want to use the **`code`** module instead.

There are two parts to this job:

1. Being able to tell if a line of input completes a Python statement: in short, telling whether to print '`>>>`' or '`...`' next.
2. Remembering which future statements the user has entered, so subsequent input can be compiled with these in effect.

The **`codeop`** module provides a way of doing each of these things, and a way of doing them both.

To do just the former:

codeop.compile_command(*source*, *filename*='*<input>*', *symbol*='*single*')

Tries to compile *source*, which should be a string of Python code and return a code object if *source* is valid Python code. In that case, the filename attribute of the code object will be *filename*, which defaults to `'<input>'`.

Returns `None` if *source* is *not* valid Python code, but is a prefix of valid Python code.

If there is a problem with *source*, an exception will be raised. `SyntaxError` is raised if there is invalid Python syntax, and `OverflowError` or `ValueError` if there is an invalid literal.

The *symbol* argument determines whether *source* is compiled as a statement (`'single'`, the default), as a sequence of statement (`'exec'`) or as an expression (`'eval'`). Any other value will cause `ValueError` to be raised.

Note

It is possible (but not likely) that the parser stops parsing with a successful outcome before reaching the end of the source; in this case, trailing symbols may be ignored instead of causing an error. For example, a backslash followed by two newlines may be followed by arbitrary garbage. This will be fixed once the API for the parser is better.

class codeop.Compile

Instances of this class have `__call__()` methods identical in signature to the built-in function `compile()`, but with the difference that if the instance compiles program text containing a `__future__` statement, the instance ‘remembers’ and compiles all subsequent program texts with the statement in force.

class codeop.CommandCompiler

Instances of this class have `__call__()` methods identical in signature to `compile_command()`; the difference is that if the instance compiles program text containing a `__future__` statement, the instance ‘remembers’ and compiles all subsequent program texts with the statement in force.

Importing Modules

The modules described in this chapter provide new ways to import other Python modules and hooks for customizing the import process.

The full list of modules described in this chapter is:

- `zipimport` — Import modules from Zip archives
  - zipimporter Objects
  - Examples
- `pkgutil` — Package extension utility
- `modulefinder` — Find modules used by a script
  - Example usage of `ModuleFinder`
- `runpy` — Locating and executing Python modules
- `importlib` — The implementation of `import`
  - Introduction
  - Functions
  - `importlib.abc` – Abstract base classes related to import
  - `importlib.machinery` – Importers and path hooks
  - `importlib.util` – Utility code for importers
  - Examples
    - Importing programmatically
    - Checking if a module can be imported
    - Importing a source file directly
    - Implementing lazy imports
    - Setting up an importer
    - Approximating `importlib.import_module()`
- `importlib.resources` – Package resource reading, opening and access
  - Functional API
- `importlib.resources.abc` – Abstract base classes for resources

- `importlib.metadata` – Accessing package metadata
  - Overview
  - Functional API
    - Entry points
    - Distribution metadata
    - Distribution versions
    - Distribution files
    - Distribution requirements
    - Mapping import to distribution packages
  - Distributions
  - Distribution Discovery
  - Implementing Custom Providers
    - Example
- The initialization of the `sys.path` module search path
  - Virtual Environments
  - _pth files
  - Embedded Python

`zipimport` — Import modules from Zip archives

Source code: Lib/zipimport.py [https://github.com/python/cpython/tree/3.14/Lib/zipimport.py]

This module adds the ability to import Python modules (`*.py`, `*.pyc`) and packages from ZIP-format archives. It is usually not needed to use the **`zipimport`** module explicitly; it is automatically used by the built-in **`import`** mechanism for **`sys.path`** items that are paths to ZIP archives.

Typically, **`sys.path`** is a list of directory names as strings. This module also allows an item of **`sys.path`** to be a string naming a ZIP file archive. The ZIP archive can contain a subdirectory structure to support package imports, and a path within the archive can be specified to only import from a subdirectory. For example, the path `example.zip/lib/` would only import from the `lib/` subdirectory within the archive.

Any files may be present in the ZIP archive, but importers are only invoked for `.py` and `.pyc` files. ZIP import of dynamic modules (`.pyd`, `.so`) is disallowed. Note that if an archive only contains `.py` files, Python will not attempt to modify the archive by adding the corresponding `.pyc` file, meaning that if a ZIP archive doesn't contain `.pyc` files, importing may be rather slow.

Changed in version 3.13: ZIP64 is supported

Changed in version 3.8: Previously, ZIP archives with an archive comment were not supported.

See also

PKZIP Application Note [https://pkware.cachefly.net/webdocs/casestudies/APPNOTE.TXT]
Documentation on the ZIP file format by Phil Katz, the creator of the format and algorithms used.

PEP 273 [https://peps.python.org/pep-0273/] - Import Modules from Zip Archives
Written by James C. Ahlstrom, who also provided an implementation. Python 2.3 follows the specification in **PEP 273** [https://peps.python.org/pep-0273/], but uses an implementation written by Just van Rossum that uses the import hooks described in **PEP 302** [https://peps.python.org/pep-0302/].

`importlib` - The implementation of the import machinery
Package providing the relevant protocols for all importers to implement.

This module defines an exception:

exception zipimport.ZipImportError
Exception raised by zipimporter objects. It’s a subclass of `ImportError`, so it can be caught as `ImportError`, too.

zipimporter Objects

`zipimporter` is the class for importing ZIP files.

class zipimport.zipimporter(*archivepath*)
Create a new zipimporter instance. *archivepath* must be a path to a ZIP file, or to a specific path within a ZIP file.

For example, an *archivepath* of `foo/bar.zip/lib` will look for modules in the `lib` directory inside the ZIP file `foo/bar.zip` (provided that it exists).

`ZipImportError` is raised if *archivepath* doesn’t point to a valid ZIP archive.

Changed in version 3.12: Methods `find_loader()` and `find_module()`, deprecated in 3.10 are now removed. Use **`find_spec()`** instead.

create_module(*spec*)

Implementation of **`importlib.abc.Loader.create_module()`** that returns **`None`** to explicitly request the default semantics.

Added in version 3.10.

exec_module(*module*)

Implementation of **`importlib.abc.Loader.exec_module()`**.

Added in version 3.10.

find_spec(*fullname*, *target=None*)

An implementation of **`importlib.abc.PathEntryFinder.find_spec()`**.

Added in version 3.10.

get_code(*fullname*)

Return the code object for the specified module. Raise **`ZipImportError`** if the module couldn’t be imported.

get_data(*pathname*)

Return the data associated with *pathname*. Raise **`OSError`** if the file wasn’t found.

Changed in version 3.3: `IOError` used to be raised, it is now an alias of `OSError`.

get_filename(*fullname*)

Return the value `__file__` would be set to if the specified module was imported. Raise `ZipImportError` if the module couldn't be imported.

Added in version 3.1.

get_source(*fullname*)

Return the source code for the specified module. Raise `ZipImportError` if the module couldn't be found, return `None` if the archive does contain the module, but has no source for it.

is_package(*fullname*)

Return `True` if the module specified by *fullname* is a package. Raise `ZipImportError` if the module couldn't be found.

load_module(*fullname*)

Load the module specified by *fullname*. *fullname* must be the fully qualified (dotted) module name. Returns the imported module on success, raises `ZipImportError` on failure.

Deprecated since version 3.10, will be removed in version 3.15: Use `exec_module()` instead.

invalidate_caches()

Clear out the internal cache of information about files found within the ZIP archive.

Added in version 3.10.

archive

The file name of the importer’s associated ZIP file, without a possible subpath.

prefix

The subpath within the ZIP file where modules are searched. This is the empty string for zipimporter objects which point to the root of the ZIP file.

The `archive` and `prefix` attributes, when combined with a slash, equal the original *archivepath* argument given to the `zipimporter` constructor.

Examples

Here is an example that imports a module from a ZIP archive - note that the `zipimport` module is not explicitly used.

```
$ unzip -l example_archive.zip
Archive:  example_archive.zip
  Length     Date   Time    Name
 --------    ----   ----    ----
     8467  01-01-00 12:30   example.py
 --------                   -------
     8467                   1 file
```

```
>>> import sys
>>> # Add the archive to the front of the module search path
>>> sys.path.insert(0, 'example_archive.zip')
>>> import example
>>> example.__file__
'example_archive.zip/example.py'
```

`pkgutil` — Package extension utility

Source code: Lib/pkgutil.py
[https://github.com/python/cpython/tree/3.14/Lib/pkgutil.py]

This module provides utilities for the import system, in particular package support.

class pkgutil.ModuleInfo(*module_finder*, *name*, *ispkg*)

A namedtuple that holds a brief summary of a module's info.

Added in version 3.6.

pkgutil.extend_path(*path*, *name*)

Extend the search path for the modules which comprise a package. Intended use is to place the following code in a package's `__init__.py`:

```
from pkgutil import extend_path
__path__ = extend_path(__path__, __name__)
```

For each directory on **`sys.path`** that has a subdirectory that matches the package name, add the subdirectory to the package's **`__path__`**. This is useful if one wants to distribute different parts of a single logical package as multiple directories.

It also looks for `*.pkg` files beginning where `*` matches the *name* argument. This feature is similar to `*.pth` files (see the **`site`** module for more information), except that it doesn't special-case lines starting with `import`. A `*.pkg` file is trusted at face value: apart from skipping blank lines and ignoring comments, all entries found in a `*.pkg` file are added to the path, regardless of whether they exist on the filesystem (this is a feature).

If the input path is not a list (as is the case for frozen packages) it is returned unchanged. The input path is not modified; an extended copy is returned. Items are only appended to the copy at the end.

It is assumed that `sys.path` is a sequence. Items of `sys.path` that are not strings referring to existing directories are ignored. Unicode items on `sys.path` that cause errors when used as filenames may cause this function to raise an exception (in line with `os.path.isdir()` behavior).

pkgutil.get_importer(*path_item*)

Retrieve a finder for the given *path_item*.

The returned finder is cached in `sys.path_importer_cache` if it was newly created by a path hook.

The cache (or part of it) can be cleared manually if a rescan of `sys.path_hooks` is necessary.

Changed in version 3.3: Updated to be based directly on `importlib` rather than relying on the package internal **PEP 302** [https://peps.python.org/pep-0302/] import emulation.

pkgutil.iter_importers(*fullname*='')

Yield finder objects for the given module name.

If *fullname* contains a `'.'`, the finders will be for the package containing *fullname*, otherwise they will be all registered top level finders (i.e. those on both `sys.meta_path` and `sys.path_hooks`).

If the named module is in a package, that package is imported as a side effect of invoking this function.

If no module name is specified, all top level finders are produced.

Changed in version 3.3: Updated to be based directly on `importlib` rather than relying on the package internal **PEP 302** [https://peps.python.org/pep-0302/] import emulation.

pkgutil.iter_modules(*path=None*, *prefix=''*)

Yields `ModuleInfo` for all submodules on *path*, or, if *path* is `None`, all top-level modules on `sys.path`.

path should be either `None` or a list of paths to look for modules in.

prefix is a string to output on the front of every module name on output.

Note

Only works for a finder which defines an `iter_modules()` method. This interface is non-standard, so the module also provides implementations for `importlib.machinery.FileFinder` and `zipimport.zipimporter`.

Changed in version 3.3: Updated to be based directly on `importlib` rather than relying on the package internal **PEP 302** [https://peps.python.org/pep-0302/] import emulation.

pkgutil.walk_packages(*path=None*, *prefix=''*, *onerror=None*)

Yields **`ModuleInfo`** for all modules recursively on *path*, or, if *path* is `None`, all accessible modules.

path should be either `None` or a list of paths to look for modules in.

prefix is a string to output on the front of every module name on output.

Note that this function must import all *packages* (*not* all modules!) on the given *path*, in order to access the `__path__` attribute to find submodules.

onerror is a function which gets called with one argument (the name of the package which was being imported) if any exception occurs while trying to import a package. If no *onerror* function is supplied, **`ImportError`**s are caught and ignored, while all other exceptions are propagated, terminating the search.

Examples:

```
# list all modules python can access
walk_packages()

# list all submodules of ctypes
walk_packages(ctypes.__path__, ctypes.__name__ + '.')
```

Note

Only works for a finder which defines an `iter_modules()` method. This interface is non-standard, so the module also provides implementations for **`importlib.machinery.FileFinder`** and **`zipimport.zipimporter`**.

Changed in version 3.3: Updated to be based directly on **`importlib`** rather than relying on the package internal **PEP 302** [https://peps.python.org/pep-0302/] import emulation.

pkgutil.get_data(*package*, *resource*)

Get a resource from a package.

This is a wrapper for the loader **`get_data`** API. The *package* argument should be the name of a package, in standard module format (`foo.bar`). The *resource* argument should be in the form of a relative filename, using `/` as the path separator. The parent directory name `..` is not allowed, and nor is a rooted name (starting with a `/`).

The function returns a binary string that is the contents of the specified resource.

For packages located in the filesystem, which have already been imported, this is the rough equivalent of:

```
d = os.path.dirname(sys.modules[package].__file__)
data = open(os.path.join(d, resource), 'rb').read()
```

If the package cannot be located or loaded, or it uses a loader which does not support **`get_data`**, then `None` is returned. In particular, the loader for namespace packages does not support **`get_data`**.

pkgutil.resolve_name(*name*)

Resolve a name to an object.

This functionality is used in numerous places in the standard library (see bpo-12915 [https://bugs.python.org/issue?@action=redirect&bpo=12915]) - and equivalent functionality is also in widely used third-party packages such as setuptools, Django and Pyramid.

It is expected that *name* will be a string in one of the following formats, where W is shorthand for a valid Python identifier and dot stands for a literal period in these pseudo-regexes:

- `W(.W)*`
- `W(.W)*:(W(.W)*)?`

The first form is intended for backward compatibility only. It assumes that some part of the dotted name is a package, and the rest is an object somewhere within that package, possibly nested inside other objects. Because the place where the package stops and the object hierarchy starts can't be inferred by inspection, repeated attempts to import must be done with this form.

In the second form, the caller makes the division point clear through the provision of a single colon: the dotted name to the left of the colon is a package to be imported, and the dotted name to the right is the object hierarchy within that package. Only one import is needed in this form. If it ends with the colon, then a module object is returned.

The function will return an object (which might be a module), or raise one of the following exceptions:

`ValueError` – if *name* isn't in a recognised format.

`ImportError` – if an import failed when it shouldn't have.

`AttributeError` – If a failure occurred when traversing the object hierarchy within the imported package to get to the desired object.

Added in version 3.9.

`modulefinder` — Find modules used by a script

Source code: Lib/modulefinder.py
[https://github.com/python/cpython/tree/3.14/Lib/modulefinder.py]

This module provides a **`ModuleFinder`** class that can be used to determine the set of modules imported by a script. `modulefinder.py` can also be run as a script, giving the filename of a Python script as its argument, after which a report of the imported modules will be printed.

modulefinder.AddPackagePath(*pkg_name*, *path*)

Record that the package named *pkg_name* can be found in the specified *path*.

modulefinder.ReplacePackage(*oldname*, *newname*)

Allows specifying that the module named *oldname* is in fact the package named *newname*.

class modulefinder.ModuleFinder(*path=None*, *debug=0*, *excludes=[]*, *replace_paths=[]*)

This class provides **`run_script()`** and **`report()`** methods to determine the set of modules imported by a script. *path* can be a list of directories to search for modules; if not specified, `sys.path` is used. *debug* sets the debugging level; higher values make the class print debugging messages about what it’s doing. *excludes* is a list of module names to exclude from the analysis. *replace_paths* is a list of `(oldpath, newpath)` tuples that will be replaced in module paths.

report()

Print a report to standard output that lists the modules imported by the script and their paths, as well as modules that are missing or seem to be missing.

run_script(*pathname*)

Analyze the contents of the *pathname* file, which must contain Python code.

modules

A dictionary mapping module names to modules. See Example usage of ModuleFinder.

Example usage of `ModuleFinder`

The script that is going to get analyzed later on (bacon.py):

```
import re, itertools

try:
    import baconhameggs
except ImportError:
    pass

try:
    import guido.python.ham
except ImportError:
    pass
```

The script that will output the report of bacon.py:

```
from modulefinder import ModuleFinder

finder = ModuleFinder()
finder.run_script('bacon.py')

print('Loaded modules:')
for name, mod in finder.modules.items():
    print('%s: ' % name, end='')
    print(','.join(list(mod.globalnames.keys())[:3]))

print('-'*50)
print('Modules not imported:')
print('\n'.join(finder.badmodules.keys()))
```

Sample output (may vary depending on the architecture):

```
Loaded modules:
_types:
copyreg:  _inverted_registry,_slotnames,__all__
re._compiler:  isstring,_sre,_optimize_unicode
_sre:
re._constants:  REPEAT_ONE,makedict,AT_END_LINE
sys:
re:  __module__,finditer,_expand
itertools:
__main__:  re,itertools,baconhameggs
re._parser:  _PATTERNENDERS,SRE_FLAG_UNICODE
array:
types:  __module__,IntType,TypeType
---------------------------------------------------
Modules not imported:
guido.python.ham
baconhameggs
```

`runpy` — Locating and executing Python modules

Source code: Lib/runpy.py [https://github.com/python/cpython/tree/3.14/Lib/runpy.py]

The `runpy` module is used to locate and run Python modules without importing them first. Its main use is to implement the `-m` command line switch that allows scripts to be located using the Python module namespace rather than the filesystem.

Note that this is *not* a sandbox module - all code is executed in the current process, and any side effects (such as cached imports of other modules) will remain in place after the functions have returned.

Furthermore, any functions and classes defined by the executed code are not guaranteed to work correctly after a `runpy` function has returned. If that limitation is not acceptable for a given use case, `importlib` is likely to be a more suitable choice than this module.

The `runpy` module provides two functions:

runpy.run_module(*mod_name*, *init_globals=None*, *run_name=None*, *alter_sys=False*)

Execute the code of the specified module and return the resulting module's globals dictionary. The module's code is first located using the standard import mechanism (refer to **PEP 302** [https://peps.python.org/pep-0302/] for details) and then executed in a fresh module namespace.

The *mod_name* argument should be an absolute module name. If the module name refers to a package rather than a normal module, then that package is imported and the **`__main__`** submodule within that package is then executed and the resulting module globals dictionary returned.

The optional dictionary argument *init_globals* may be used to pre-populate the module's globals dictionary before the code is executed. *init_globals* will not be modified. If any of the special global variables below are defined in *init_globals*, those definitions are overridden by **`run_module()`**.

The special global variables `__name__`, `__spec__`, `__file__`, `__cached__`, `__loader__` and `__package__` are set in the globals dictionary before the module code is executed. (Note that this is a minimal set of variables - other variables may be set implicitly as an interpreter implementation detail.)

`__name__` is set to *run_name* if this optional argument is not **`None`**, to `mod_name + '.__main__'` if the named module is a package and to the *mod_name* argument otherwise.

`__spec__` will be set appropriately for the *actually* imported module (that is, `__spec__.name` will always be *mod_name* or `mod_name + '.__main__'`, never *run_name*).

`__file__`, `__cached__`, `__loader__` and `__package__` are set as normal based on the module spec.

If the argument *alter_sys* is supplied and evaluates to **`True`**, then `sys.argv[0]` is updated with the value of `__file__` and `sys.modules[__name__]` is updated with a temporary module object for the module being executed. Both `sys.argv[0]` and `sys.modules[__name__]` are restored to their original values before the function returns.

Note that this manipulation of `sys` is not thread-safe. Other threads may see the partially initialised module, as well as the altered list of arguments. It is recommended that the `sys` module be left alone when invoking this function from threaded code.

See also

The `-m` option offering equivalent functionality from the command line.

Changed in version 3.1: Added ability to execute packages by looking for a `__main__` submodule.

Changed in version 3.2: Added `__cached__` global variable (see **PEP 3147** [https://peps.python.org/pep-3147/]).

Changed in version 3.4: Updated to take advantage of the module spec feature added by **PEP 451** [https://peps.python.org/pep-0451/]. This allows `__cached__` to be set correctly for modules run this way, as well as ensuring the real module name is always accessible as `__spec__.name`.

Changed in version 3.12: The setting of `__cached__`, `__loader__`, and `__package__` are deprecated. See `ModuleSpec` for alternatives.

runpy.run_path(*path_name*, *init_globals=None*, *run_name=None*)

Execute the code at the named filesystem location and return the resulting module’s globals dictionary. As with a script name supplied to the CPython command line,

file_path may refer to a Python source file, a compiled bytecode file or a valid `sys.path` entry containing a `__main__` module (e.g. a zipfile containing a top-level `__main__.py` file).

For a simple script, the specified code is simply executed in a fresh module namespace. For a valid `sys.path` entry (typically a zipfile or directory), the entry is first added to the beginning of `sys.path`. The function then looks for and executes a `__main__` module using the updated path. Note that there is no special protection against invoking an existing `__main__` entry located elsewhere on `sys.path` if there is no such module at the specified location.

The optional dictionary argument *init_globals* may be used to pre-populate the module's globals dictionary before the code is executed. *init_globals* will not be modified. If any of the special global variables below are defined in *init_globals*, those definitions are overridden by `run_path()`.

The special global variables `__name__`, `__spec__`, `__file__`, `__cached__`, `__loader__` and `__package__` are set in the globals dictionary before the module code is executed. (Note that this is a minimal set of variables - other variables may be set implicitly as an interpreter implementation detail.)

`__name__` is set to *run_name* if this optional argument is not `None` and to `'<run_path>'` otherwise.

If *file_path* directly references a script file (whether as source or as precompiled byte code), then `__file__` will be set to *file_path*, and `__spec__`, `__cached__`, `__loader__` and `__package__` will all be set to `None`.

If *file_path* is a reference to a valid **`sys.path`** entry, then `__spec__` will be set appropriately for the imported **`__main__`** module (that is, `__spec__.name` will always be `__main__`). `__file__`, `__cached__`, `__loader__` and `__package__` will be set as normal based on the module spec.

A number of alterations are also made to the **`sys`** module. Firstly, **`sys.path`** may be altered as described above. `sys.argv[0]` is updated with the value of *file_path* and `sys.modules[__name__]` is updated with a temporary module object for the module being executed. All modifications to items in **`sys`** are reverted before the function returns.

Note that, unlike **`run_module()`**, the alterations made to **`sys`** are not optional in this function as these adjustments are essential to allowing the execution of **`sys.path`** entries. As the thread-safety limitations still apply, use of this function in threaded code should be either serialised with the import lock or delegated to a separate process.

See also

Interface options for equivalent functionality on the command line (`python path/to/script`).

Added in version 3.2.

Changed in version 3.4: Updated to take advantage of the module spec feature added by **PEP 451** [https://peps.python.org/pep-0451/]. This allows `__cached__` to be set correctly in the case where `__main__` is imported from a valid **`sys.path`** entry rather than being executed directly.

Changed in version 3.12: The setting of `__cached__`, `__loader__`, and `__package__` are deprecated.

See also

PEP 338 [https://peps.python.org/pep-0338/] – Executing modules as scripts
PEP written and implemented by Nick Coghlan.

PEP 366 [https://peps.python.org/pep-0366/] – Main module explicit relative imports
PEP written and implemented by Nick Coghlan.

PEP 451 [https://peps.python.org/pep-0451/] – A ModuleSpec Type for the Import System
PEP written and implemented by Eric Snow

Command line and environment - CPython command line details

The `importlib.import_module()` function

`importlib` — The implementation of `import`

Added in version 3.1.

Source code: Lib/importlib/__init__.py
[https://github.com/python/cpython/tree/3.14/Lib/importlib/__init__.py]

Introduction

The purpose of the **`importlib`** package is three-fold.

One is to provide the implementation of the **`import`** statement (and thus, by extension, the `__import__()` function) in Python source code. This provides an implementation of **`import`** which is portable to any Python interpreter. This also provides an implementation which is easier to comprehend than one implemented in a programming language other than Python.

Two, the components to implement **`import`** are exposed in this package, making it easier for users to create their own custom objects (known generically as an importer) to participate in the import process.

Three, the package contains modules exposing additional functionality for managing aspects of Python packages:

- **`importlib.metadata`** presents access to metadata from third-party distributions.
- **`importlib.resources`** provides routines for accessing non-code “resources” from Python packages.

See also

The import statement
The language reference for the `import` statement.

Packages specification **[https://www.python.org/doc/essays/packages/]**
Original specification of packages. Some semantics have changed since the writing of this document (e.g. redirecting based on `None` in `sys.modules`).

The `__import__()` function
The `import` statement is syntactic sugar for this function.

The initialization of the sys.path module search path
The initialization of `sys.path`.

PEP 235 **[https://peps.python.org/pep-0235/]**
Import on Case-Insensitive Platforms

PEP 263 **[https://peps.python.org/pep-0263/]**
Defining Python Source Code Encodings

PEP 302 **[https://peps.python.org/pep-0302/]**
New Import Hooks

PEP 328 **[https://peps.python.org/pep-0328/]**
Imports: Multi-Line and Absolute/Relative

PEP 366 **[https://peps.python.org/pep-0366/]**
Main module explicit relative imports

PEP 420 **[https://peps.python.org/pep-0420/]**
Implicit namespace packages

PEP 451 **[https://peps.python.org/pep-0451/]**

A ModuleSpec Type for the Import System

PEP 488 **[https://peps.python.org/pep-0488/]**
Elimination of PYO files

PEP 489 **[https://peps.python.org/pep-0489/]**
Multi-phase extension module initialization

PEP 552 **[https://peps.python.org/pep-0552/]**
Deterministic pycs

PEP 3120 **[https://peps.python.org/pep-3120/]**
Using UTF-8 as the Default Source Encoding

PEP 3147 **[https://peps.python.org/pep-3147/]**
PYC Repository Directories

Functions

importlib.__import__(*name*, *globals=None*, *locals=None*, *fromlist=()*, *level=0*)

An implementation of the built-in `__import__()` function.

Note

Programmatic importing of modules should use `import_module()` instead of this function.

importlib.import_module(*name*, *package=None*)

Import a module. The *name* argument specifies what module to import in absolute or relative terms (e.g. either `pkg.mod` or `..mod`). If the name is specified in relative terms, then the *package* argument must be set to the name of the package which is to act as the anchor for resolving the package name (e.g. `import_module('..mod', 'pkg.subpkg')` will import `pkg.mod`).

The `import_module()` function acts as a simplifying wrapper around `importlib.__import__()`. This means all semantics of the function are derived from `importlib.__import__()`. The most important difference between these two functions is that `import_module()` returns the specified package or module (e.g. `pkg.mod`), while `__import__()` returns the top-level package or module (e.g. `pkg`).

If you are dynamically importing a module that was created since the interpreter began execution (e.g., created a Python source file), you may need to call `invalidate_caches()` in order for the new module to be noticed by the import system.

Changed in version 3.3: Parent packages are automatically imported.

importlib.invalidate_caches()

Invalidate the internal caches of finders stored at `sys.meta_path`. If a finder implements `invalidate_caches()` then it will be called to perform the invalidation. This function should be called if any modules are created/installed while your program is running to guarantee all finders will notice the new module’s existence.

Added in version 3.3.

Changed in version 3.10: Namespace packages created/installed in a different `sys.path` location after the same namespace was already imported are noticed.

importlib.reload(*module*)

Reload a previously imported *module*. The argument must be a module object, so it must have been successfully imported before. This is useful if you have edited the module source file using an external editor and want to try out the new version without leaving the Python interpreter. The return value is the module object (which can be different if re-importing causes a different object to be placed in **`sys.modules`**).

When **`reload()`** is executed:

- Python module’s code is recompiled and the module-level code re-executed, defining a new set of objects which are bound to names in the module’s dictionary by reusing the loader which originally loaded the module. The `init` function of extension modules is not called a second time.
- As with all other objects in Python the old objects are only reclaimed after their reference counts drop to zero.
- The names in the module namespace are updated to point to any new or changed objects.
- Other references to the old objects (such as names external to the module) are not rebound to refer to the new objects and must be updated in each namespace where they occur if that is desired.

There are a number of other caveats:

When a module is reloaded, its dictionary (containing the module’s global variables) is retained. Redefinitions of names will override the old definitions, so this is generally not a problem. If the new version of a module does not define a name that was defined by the old version, the old definition remains. This feature can be used to the module’s advantage if it maintains a global table or cache of objects — with a **`try`** statement it can test for the table’s presence and skip its initialization if desired:

```
try:
    cache
except NameError:
    cache = {}
```

It is generally not very useful to reload built-in or dynamically loaded modules. Reloading `sys`, `__main__`, `builtins` and other key modules is not recommended. In many cases extension modules are not designed to be initialized more than once, and may fail in arbitrary ways when reloaded.

If a module imports objects from another module using `from` … `import` …, calling `reload()` for the other module does not redefine the objects imported from it — one way around this is to re-execute the **`from`** statement, another is to use **`import`** and qualified names (*module.name*) instead.

If a module instantiates instances of a class, reloading the module that defines the class does not affect the method definitions of the instances — they continue to use the old class definition. The same is true for derived classes.

Added in version 3.4.

Changed in version 3.7: `ModuleNotFoundError` is raised when the module being reloaded lacks a `ModuleSpec`.

Warning

This function is not thread-safe. Calling it from multiple threads can result in unexpected behavior. It’s recommended to use the `threading.Lock` or other synchronization primitives for thread-safe module reloading.

`importlib.abc` – Abstract base classes related to import

Source code: Lib/importlib/abc.py [https://github.com/python/cpython/tree/3.14/Lib/importlib/abc.py]

The `importlib.abc` module contains all of the core abstract base classes used by `import`. Some subclasses of the core abstract base classes are also provided to help in implementing the core ABCs.

ABC hierarchy:

```
object
 +-- MetaPathFinder
 +-- PathEntryFinder
 +-- Loader
      +-- ResourceLoader --------+
      +-- InspectLoader          |
           +-- ExecutionLoader --+
                                 +-- FileLoader
                                 +-- SourceLoader
```

class importlib.abc.MetaPathFinder

An abstract base class representing a meta path finder.

Added in version 3.3.

Changed in version 3.10: No longer a subclass of **`Finder`**.

find_spec(*fullname*, *path*, *target=None*)

An abstract method for finding a spec for the specified module. If this is a top-level import, *path* will be `None`. Otherwise, this is a search for a subpackage or module and *path* will be the value of **`__path__`** from the parent package. If a spec cannot be found, `None` is returned. When passed in, `target` is a module object that the finder may use to make a more educated guess about what spec to return. **`importlib.util.spec_from_loader()`** may be useful for implementing concrete `MetaPathFinders`.

Added in version 3.4.

invalidate_caches()

An optional method which, when called, should invalidate any internal cache used by the finder. Used by **`importlib.invalidate_caches()`** when invalidating the caches of all finders on **`sys.meta_path`**.

Changed in version 3.4: Returns `None` when called instead of **`NotImplemented`**.

class importlib.abc.PathEntryFinder

An abstract base class representing a path entry finder. Though it bears some similarities to **`MetaPathFinder`**, `PathEntryFinder` is meant for use only within the path-based import subsystem provided by **`importlib.machinery.PathFinder`**.

Added in version 3.3.

Changed in version 3.10: No longer a subclass of **`Finder`**.

find_spec(*fullname*, *target=None*)

An abstract method for finding a spec for the specified module. The finder will search for the module only within the path entry to which it is assigned. If a spec cannot be found, `None` is returned. When passed in, `target` is a module object that the finder may use to make a more educated guess about what spec to return. **`importlib.util.spec_from_loader()`** may be useful for implementing concrete `PathEntryFinders`.

Added in version 3.4.

invalidate_caches()

An optional method which, when called, should invalidate any internal cache used by the finder. Used by **`importlib.machinery.PathFinder.invalidate_caches()`** when invalidating the caches of all cached finders.

class importlib.abc.Loader

An abstract base class for a loader. See **PEP 302** [https://peps.python.org/pep-0302/] for the exact definition for a loader.

Loaders that wish to support resource reading should implement a **`get_resource_reader()`** method as specified by **`importlib.resources.abc.ResourceReader`**.

Changed in version 3.7: Introduced the optional **`get_resource_reader()`** method.

create_module(*spec*)

A method that returns the module object to use when importing a module. This method may return `None`, indicating that default module creation semantics should take place.

Added in version 3.4.

Changed in version 3.6: This method is no longer optional when `exec_module()` is defined.

exec_module(*module*)

An abstract method that executes the module in its own namespace when a module is imported or reloaded. The module should already be initialized when `exec_module()` is called. When this method exists, `create_module()` must be defined.

Added in version 3.4.

Changed in version 3.6: `create_module()` must also be defined.

load_module(*fullname*)

A legacy method for loading a module. If the module cannot be loaded, `ImportError` is raised, otherwise the loaded module is returned.

If the requested module already exists in `sys.modules`, that module should be used and reloaded. Otherwise the loader should create a new module and insert it into `sys.modules` before any loading begins, to prevent recursion from the import. If the loader inserted a module and the load fails, it must be removed by the loader from `sys.modules`; modules already in `sys.modules` before the loader began execution should be left alone.

The loader should set several attributes on the module (note that some of these attributes can change when a module is reloaded):

- `module.__name__`
- `module.__file__`

- `module.__cached__` *(deprecated)*
- `module.__path__`
- `module.__package__` *(deprecated)*
- `module.__loader__` *(deprecated)*

When `exec_module()` is available then backwards-compatible functionality is provided.

Changed in version 3.4: Raise `ImportError` when called instead of `NotImplementedError`. Functionality provided when `exec_module()` is available.

Deprecated since version 3.4, will be removed in version 3.15: The recommended API for loading a module is `exec_module()` (and `create_module()`). Loaders should implement it instead of `load_module()`. The import machinery takes care of all the other responsibilities of `load_module()` when `exec_module()` is implemented.

class importlib.abc.ResourceLoader

Superseded by TraversableResources

An abstract base class for a loader which implements the optional **PEP 302** [https://peps.python.org/pep-0302/] protocol for loading arbitrary resources from the storage back-end.

Deprecated since version 3.7: This ABC is deprecated in favour of supporting resource loading through `importlib.resources.abc.TraversableResources`. This class exists for backwards compatibility only with other ABCs in this module.

abstractmethod get_data(*path*)

An abstract method to return the bytes for the data located at *path*. Loaders that have a file-like storage back-end that allows storing arbitrary data can implement this abstract method to give direct access to the data stored. **`OSError`** is to be raised if the *path* cannot be found. The *path* is expected to be constructed using a module's **`__file__`** attribute or an item from a package's **`__path__`**.

Changed in version 3.4: Raises **`OSError`** instead of **`NotImplementedError`**.

class importlib.abc.InspectLoader

An abstract base class for a loader which implements the optional **PEP 302** [https://peps.python.org/pep-0302/] protocol for loaders that inspect modules.

get_code(*fullname*)

Return the code object for a module, or `None` if the module does not have a code object (as would be the case, for example, for a built-in module). Raise an **`ImportError`** if loader cannot find the requested module.

Note

While the method has a default implementation, it is suggested that it be overridden if possible for performance.

Changed in version 3.4: No longer abstract and a concrete implementation is provided.

abstractmethod get_source(*fullname*)

An abstract method to return the source of a module. It is returned as a text string using universal newlines, translating all recognized line separators into `'\n'` characters. Returns `None` if no source is available (e.g. a built-in module). Raises **`ImportError`** if the loader cannot find the module specified.

Changed in version 3.4: Raises **`ImportError`** instead of **`NotImplementedError`**.

is_package(*fullname*)

An optional method to return a true value if the module is a package, a false value otherwise. **`ImportError`** is raised if the loader cannot find the module.

Changed in version 3.4: Raises **`ImportError`** instead of **`NotImplementedError`**.

static source_to_code(*data*, *path*='<*string*>')

Create a code object from Python source.

The *data* argument can be whatever the **`compile()`** function supports (i.e. string or bytes). The *path* argument should be the “path” to where the source code originated from, which can be an abstract concept (e.g. location in a zip file).

With the subsequent code object one can execute it in a module by running `exec(code, module.__dict__)`.

Added in version 3.4.

Changed in version 3.5: Made the method static.

exec_module(*module*)

Implementation of `Loader.exec_module()`.

Added in version 3.4.

load_module(*fullname*)

Implementation of `Loader.load_module()`.

Deprecated since version 3.4, will be removed in version 3.15: use `exec_module()` instead.

class importlib.abc.ExecutionLoader

An abstract base class which inherits from `InspectLoader` that, when implemented, helps a module to be executed as a script. The ABC represents an optional **PEP 302** [https://peps.python.org/pep-0302/] protocol.

abstractmethod get_filename(*fullname*)

An abstract method that is to return the value of `__file__` for the specified module. If no path is available, `ImportError` is raised.

If source code is available, then the method should return the path to the source file, regardless of whether a bytecode was used to load the module.

Changed in version 3.4: Raises `ImportError` instead of `NotImplementedError`.

class importlib.abc.FileLoader(*fullname*, *path*)

An abstract base class which inherits from `ResourceLoader` and `ExecutionLoader`, providing concrete implementations of `ResourceLoader.get_data()` and `ExecutionLoader.get_filename()`.

The *fullname* argument is a fully resolved name of the module the loader is to handle. The *path* argument is the path to the file for the module.

Added in version 3.3.

name

The name of the module the loader can handle.

path

Path to the file of the module.

load_module(*fullname*)

Calls super's `load_module()`.

Deprecated since version 3.4, will be removed in version 3.15: Use `Loader.exec_module()` instead.

abstractmethod get_filename(*fullname*)

Returns `path`.

abstractmethod get_data(*path*)

Reads *path* as a binary file and returns the bytes from it.

class importlib.abc.SourceLoader

An abstract base class for implementing source (and optionally bytecode) file loading. The class inherits from both `ResourceLoader` and `ExecutionLoader`, requiring the implementation of:

- `ResourceLoader.get_data()`
- `ExecutionLoader.get_filename()`

  Should only return the path to the source file; sourceless loading is not supported.

The abstract methods defined by this class are to add optional bytecode file support. Not implementing these optional methods (or causing them to raise `NotImplementedError`) causes the loader to only work with source code. Implementing the methods allows the loader to work with source *and* bytecode files; it does not allow for *sourceless* loading where only bytecode is provided. Bytecode files are an optimization to speed up loading by removing the parsing step of Python's compiler, and so no bytecode-specific API is exposed.

path_stats(*path*)

Optional abstract method which returns a `dict` containing metadata about the specified path. Supported dictionary keys are:

- `'mtime'` (mandatory): an integer or floating-point number representing the modification time of the source code;
- `'size'` (optional): the size in bytes of the source code.

Any other keys in the dictionary are ignored, to allow for future extensions. If the path cannot be handled, `OSError` is raised.

Added in version 3.3.

Changed in version 3.4: Raise `OSError` instead of `NotImplementedError`.

path_mtime(*path*)

Optional abstract method which returns the modification time for the specified path.

Deprecated since version 3.3: This method is deprecated in favour of `path_stats()`. You don't have to implement it, but it is still available for compatibility purposes. Raise `OSError` if the path cannot be handled. *Changed in version 3.4:* Raise `OSError` instead of `NotImplementedError`.

set_data(*path*, *data*)

Optional abstract method which writes the specified bytes to a file path. Any intermediate directories which do not exist are to be created automatically.

When writing to the path fails because the path is read-only (`errno.EACCES`/`PermissionError`), do not propagate the exception.

Changed in version 3.4: No longer raises `NotImplementedError` when called.

get_code(*fullname*)

Concrete implementation of `InspectLoader.get_code()`.

exec_module(*module*)

Concrete implementation of `Loader.exec_module()`.

Added in version 3.4.

load_module(*fullname*)

Concrete implementation of `Loader.load_module()`.

Deprecated since version 3.4, will be removed in version 3.15: Use `exec_module()` instead.

get_source(*fullname*)

Concrete implementation of `InspectLoader.get_source()`.

is_package(*fullname*)

Concrete implementation of `InspectLoader.is_package()`. A module is determined to be a package if its file path (as provided by `ExecutionLoader.get_filename()`) is a file named `__init__` when the file extension is removed **and** the module name itself does not end in `__init__`.

class importlib.abc.ResourceReader

Superseded by TraversableResources

An abstract base class to provide the ability to read *resources*.

From the perspective of this ABC, a *resource* is a binary artifact that is shipped within a package. Typically this is something like a data file that lives next to the `__init__.py` file of the package. The purpose of this class is to help abstract out the accessing of such data files so that it does not matter if the package and its data file(s) are stored e.g. in a zip file versus on the file system.

For any of methods of this class, a *resource* argument is expected to be a path-like object which represents conceptually just a file name. This means that no subdirectory paths should be included in the *resource* argument. This is because the location of the package the reader is for, acts as the “directory”. Hence the metaphor for directories and file names is packages and resources, respectively. This is also why instances of this class are expected to directly correlate to a specific package (instead of potentially representing multiple packages or a module).

Loaders that wish to support resource reading are expected to provide a method called `get_resource_reader(fullname)` which returns an object implementing this ABC’s interface. If the module specified by fullname is not a package, this method should return `None`. An object compatible with this ABC should only be returned when the specified module is a package.

Added in version 3.7.

Deprecated since version 3.12, removed in version 3.14: Use `importlib.resources.abc.TraversableResources` instead.

abstractmethod open_resource(*resource*)

> Returns an opened, file-like object for binary reading of the *resource*.
>
> If the resource cannot be found, `FileNotFoundError` is raised.

abstractmethod resource_path(*resource*)

> Returns the file system path to the *resource*.
>
> If the resource does not concretely exist on the file system, raise `FileNotFoundError`.

abstractmethod is_resource(*name*)

> Returns `True` if the named *name* is considered a resource. `FileNotFoundError` is raised if *name* does not exist.

abstractmethod contents()

> Returns an iterable of strings over the contents of the package. Do note that it is not required that all names returned by the iterator be actual

resources, e.g. it is acceptable to return names for which `is_resource()` would be false.

Allowing non-resource names to be returned is to allow for situations where how a package and its resources are stored are known a priori and the non-resource names would be useful. For instance, returning subdirectory names is allowed so that when it is known that the package and resources are stored on the file system then those subdirectory names can be used directly.

The abstract method returns an iterable of no items.

class importlib.abc.Traversable

An object with a subset of `pathlib.Path` methods suitable for traversing directories and opening files.

For a representation of the object on the file-system, use `importlib.resources.as_file()`.

Added in version 3.9.

Deprecated since version 3.12, removed in version 3.14: Use `importlib.resources.abc.Traversable` instead.

name

Abstract. The base name of this object without any parent references.

abstractmethod iterdir()

Yield `Traversable` objects in `self`.

abstractmethod is_dir()

Return `True` if `self` is a directory.

abstractmethod is_file()

Return `True` if `self` is a file.

abstractmethod joinpath(*child*)

Return Traversable child in `self`.

abstractmethod __truediv__(*child*)

Return `Traversable` child in `self`.

abstractmethod open(*mode='r'*, **args*, ***kwargs*)

mode may be ‘r’ or ‘rb’ to open as text or binary. Return a handle suitable for reading (same as `pathlib.Path.open`).

When opening as text, accepts encoding parameters such as those accepted by `io.TextIOWrapper`.

read_bytes()

Read contents of `self` as bytes.

read_text(*encoding=None*)

Read contents of `self` as text.

class importlib.abc.TraversableResources

An abstract base class for resource readers capable of serving the `importlib.resources.files()` interface. Subclasses `importlib.resources.abc.ResourceReader` and provides concrete implementations of the `importlib.resources.abc.ResourceReader`’s abstract methods. Therefore, any loader supplying `importlib.abc.TraversableResources` also supplies ResourceReader.

Loaders that wish to support resource reading are expected to implement this interface.

Added in version 3.9.

Deprecated since version 3.12, removed in version 3.14: Use `importlib.resources.abc.TraversableResources` instead.

abstractmethod files()

Returns a `importlib.resources.abc.Traversable` object for the loaded package.

`importlib.machinery` – Importers and path hooks

Source code: Lib/importlib/machinery.py [https://github.com/python/cpython/tree/3.14/Lib/importlib/machinery.py]

This module contains the various objects that help `import` find and load modules.

importlib.machinery.SOURCE_SUFFIXES

A list of strings representing the recognized file suffixes for source modules.

Added in version 3.3.

importlib.machinery.DEBUG_BYTECODE_SUFFIXES

A list of strings representing the file suffixes for non-optimized bytecode modules.

Added in version 3.3.

Deprecated since version 3.5: Use `BYTECODE_SUFFIXES` instead.

importlib.machinery.OPTIMIZED_BYTECODE_SUFFIXES

A list of strings representing the file suffixes for optimized bytecode modules.

Added in version 3.3.

Deprecated since version 3.5: Use **`BYTECODE_SUFFIXES`** instead.

importlib.machinery.BYTECODE_SUFFIXES

A list of strings representing the recognized file suffixes for bytecode modules (including the leading dot).

Added in version 3.3.

Changed in version 3.5: The value is no longer dependent on `__debug__`.

importlib.machinery.EXTENSION_SUFFIXES

A list of strings representing the recognized file suffixes for extension modules.

Added in version 3.3.

importlib.machinery.all_suffixes()

Returns a combined list of strings representing all file suffixes for modules recognized by the standard import machinery. This is a helper for code which simply needs to know if a filesystem path potentially refers to a module without needing any details on the kind of module (for example, **`inspect.getmodulename()`**).

Added in version 3.3.

class importlib.machinery.BuiltinImporter

An importer for built-in modules. All known built-in modules are listed in **`sys.builtin_module_names`**. This class implements the `importlib.abc.MetaPathFinder` and `importlib.abc.InspectLoader` ABCs.

Only class methods are defined by this class to alleviate the need for instantiation.

Changed in version 3.5: As part of **PEP 489** [https://peps.python.org/pep-0489/], the builtin importer now implements `Loader.create_module()` and `Loader.exec_module()`

class importlib.machinery.FrozenImporter

An importer for frozen modules. This class implements the `importlib.abc.MetaPathFinder` and `importlib.abc.InspectLoader` ABCs.

Only class methods are defined by this class to alleviate the need for instantiation.

Changed in version 3.4: Gained `create_module()` and `exec_module()` methods.

class importlib.machinery.WindowsRegistryFinder

Finder for modules declared in the Windows registry. This class implements the `importlib.abc.MetaPathFinder` ABC.

Only class methods are defined by this class to alleviate the need for instantiation.

Added in version 3.3.

Deprecated since version 3.6: Use `site` configuration instead. Future versions of Python may not enable this finder by default.

class importlib.machinery.PathFinder

A Finder for `sys.path` and package `__path__` attributes. This class implements the `importlib.abc.MetaPathFinder` ABC.

Only class methods are defined by this class to alleviate the need for instantiation.

classmethod find_spec(*fullname*, *path=None*, *target=None*)

Class method that attempts to find a spec for the module specified by *fullname* on **`sys.path`** or, if defined, on *path*. For each path entry that is searched, **`sys.path_importer_cache`** is checked. If a non-false object is found then it is used as the path entry finder to look for the module being searched for. If no entry is found in **`sys.path_importer_cache`**, then **`sys.path_hooks`** is searched for a finder for the path entry and, if found, is stored in **`sys.path_importer_cache`** along with being queried about the module. If no finder is ever found then `None` is both stored in the cache and returned.

Added in version 3.4.

Changed in version 3.5: If the current working directory – represented by an empty string – is no longer valid then `None` is returned but no value is cached in **`sys.path_importer_cache`**.

classmethod invalidate_caches()

Calls **`importlib.abc.PathEntryFinder.invalidate_caches()`** on all finders stored in **`sys.path_importer_cache`** that define the method. Otherwise entries in **`sys.path_importer_cache`** set to `None` are deleted.

Changed in version 3.7: Entries of `None` in **`sys.path_importer_cache`** are deleted.

Changed in version 3.4: Calls objects in **`sys.path_hooks`** with the current working directory for `''` (i.e. the empty string).

class importlib.machinery.FileFinder(*path*, **loader_details*)

A concrete implementation of `importlib.abc.PathEntryFinder` which caches results from the file system.

The *path* argument is the directory for which the finder is in charge of searching.

The *loader_details* argument is a variable number of 2-item tuples each containing a loader and a sequence of file suffixes the loader recognizes. The loaders are expected to be callables which accept two arguments of the module's name and the path to the file found.

The finder will cache the directory contents as necessary, making stat calls for each module search to verify the cache is not outdated. Because cache staleness relies upon the granularity of the operating system's state information of the file system, there is a potential race condition of searching for a module, creating a new file, and then searching for the module the new file represents. If the operations happen fast enough to fit within the granularity of stat calls, then the module search will fail. To prevent this from happening, when you create a module dynamically, make sure to call `importlib.invalidate_caches()`.

Added in version 3.3.

path

The path the finder will search in.

find_spec(*fullname*, *target=None*)

Attempt to find the spec to handle *fullname* within `path`.

Added in version 3.4.

invalidate_caches()

Clear out the internal cache.

classmethod path_hook(**loader_details*)

A class method which returns a closure for use on `sys.path_hooks`. An instance of `FileFinder` is returned by the closure using the path argument given to the closure directly and *loader_details* indirectly.

If the argument to the closure is not an existing directory, `ImportError` is raised.

class importlib.machinery.SourceFileLoader(*fullname*, *path*)

A concrete implementation of `importlib.abc.SourceLoader` by subclassing `importlib.abc.FileLoader` and providing some concrete implementations of other methods.

Added in version 3.3.

name

The name of the module that this loader will handle.

path

The path to the source file.

is_package(*fullname*)

Return `True` if `path` appears to be for a package.

path_stats(*path*)

Concrete implementation of `importlib.abc.SourceLoader.path_stats()`.

set_data(*path*, *data*)

Concrete implementation of `importlib.abc.SourceLoader.set_data()`.

load_module(*name=None*)

Concrete implementation of `importlib.abc.Loader.load_module()` where specifying the name of the module to load is optional.

Deprecated since version 3.6, will be removed in version 3.15: Use `importlib.abc.Loader.exec_module()` instead.

class importlib.machinery.SourcelessFileLoader(*fullname*, *path*)

A concrete implementation of `importlib.abc.FileLoader` which can import bytecode files (i.e. no source code files exist).

Please note that direct use of bytecode files (and thus not source code files) inhibits your modules from being usable by all Python implementations or new versions of Python which change the bytecode format.

Added in version 3.3.

name

The name of the module the loader will handle.

path

The path to the bytecode file.

is_package(*fullname*)

Determines if the module is a package based on `path`.

get_code(*fullname*)

Returns the code object for `name` created from `path`.

get_source(*fullname*)

Returns `None` as bytecode files have no source when this loader is used.

load_module(*name=None*)

Concrete implementation of `importlib.abc.Loader.load_module()` where specifying the name of the module to load is optional.

Deprecated since version 3.6, will be removed in version 3.15: Use `importlib.abc.Loader.exec_module()` instead.

class importlib.machinery.ExtensionFileLoader(*fullname*, *path*)

A concrete implementation of `importlib.abc.ExecutionLoader` for extension modules.

The *fullname* argument specifies the name of the module the loader is to support. The *path* argument is the path to the extension module's file.

Note that, by default, importing an extension module will fail in subinterpreters if it doesn't implement multi-phase init (see **PEP 489** [https://peps.python.org/pep-0489/]), even if it would otherwise import successfully.

Added in version 3.3.

Changed in version 3.12: Multi-phase init is now required for use in subinterpreters.

name

Name of the module the loader supports.

path

Path to the extension module.

create_module(*spec*)

Creates the module object from the given specification in accordance with **PEP 489** [https://peps.python.org/pep-0489/].

Added in version 3.5.

exec_module(*module*)

Initializes the given module object in accordance with **PEP 489** [https://peps.python.org/pep-0489/].

Added in version 3.5.

is_package(*fullname*)

Returns `True` if the file path points to a package's `__init__` module based on `EXTENSION_SUFFIXES`.

get_code(*fullname*)

Returns `None` as extension modules lack a code object.

get_source(*fullname*)

Returns `None` as extension modules do not have source code.

get_filename(*fullname*)

Returns `path`.

Added in version 3.4.

class importlib.machinery.NamespaceLoader(*name*, *path*, *path_finder*)

A concrete implementation of `importlib.abc.InspectLoader` for namespace packages. This is an alias for a private class and is only made public for introspecting the `__loader__` attribute on namespace packages:

```
>>> from importlib.machinery import NamespaceLoader
>>> import my_namespace
>>> isinstance(my_namespace.__loader__, NamespaceLoader)
True
>>> import importlib.abc
>>> isinstance(my_namespace.__loader__,
importlib.abc.Loader)
True
```

Added in version 3.11.

class importlib.machinery.ModuleSpec(*name*, *loader*, *, *origin=None*, *loader_state=None*, *is_package=None*)

A specification for a module's import-system-related state. This is typically exposed as the module's **`__spec__`** attribute. Many of these attributes are also available directly on a module: for example, `module.__spec__.origin == module.__file__`. Note, however, that while the *values* are usually equivalent, they can differ since there is no synchronization between the two objects. For example, it is possible to update the module's **`__file__`** at runtime and this will not be automatically reflected in the module's **`__spec__.origin`**, and vice versa.

Added in version 3.4.

name

The module's fully qualified name (see **`module.__name__`**). The finder should always set this attribute to a non-empty string.

loader

The loader used to load the module (see **`module.__loader__`**). The finder should always set this attribute.

origin

The location the loader should use to load the module (see **`module.__file__`**). For example, for modules loaded from a `.py` file this is the filename. The finder should always set this attribute to a meaningful value for the loader to use. In the uncommon case that there is not one (like for namespace packages), it should be set to `None`.

submodule_search_locations

A (possibly empty) sequence of strings enumerating the locations in which a package's submodules will be found (see **`module.__path__`**). Most of the time there will only be a single directory in this list.

The finder should set this attribute to a sequence, even an empty one, to indicate to the import system that the module is a package. It should be set to `None` for non-package modules. It is set automatically later to a special object for namespace packages.

loader_state

The finder may set this attribute to an object containing additional, module-specific data to use when loading the module. Otherwise it should be set to `None`.

cached

The filename of a compiled version of the module's code (see **`module.__cached__`**). The finder should always set this attribute but it may be `None` for modules that do not need compiled code stored.

parent

(Read-only) The fully qualified name of the package the module is in (or the empty string for a top-level module). See `module.__package__`. If the module is a package then this is the same as `name`.

has_location

True if the spec's `origin` refers to a loadable location, `False` otherwise. This value impacts how **`origin`** is interpreted and how the module's `__file__` is populated.

class importlib.machinery.AppleFrameworkLoader(*name*, *path*)

A specialization of `importlib.machinery.ExtensionFileLoader` that is able to load extension modules in Framework format.

For compatibility with the iOS App Store, *all* binary modules in an iOS app must be dynamic libraries, contained in a framework with appropriate metadata, stored in the `Frameworks` folder of the packaged app. There can be only a single binary per framework, and there can be no executable binary material outside the Frameworks folder.

To accommodate this requirement, when running on iOS, extension module binaries are *not* packaged as `.so` files on `sys.path`, but as individual standalone frameworks. To discover those frameworks, this loader is registered against the `.fwork` file extension, with a `.fwork` file acting as a placeholder in the original location of the binary on `sys.path`. The `.fwork` file contains the path of the actual binary in the `Frameworks` folder, relative to the app bundle. To allow for resolving a framework-packaged binary back to the original location, the framework is expected to contain a `.origin` file that contains the location of the `.fwork` file, relative to the app bundle.

For example, consider the case of an import `from foo.bar import _whiz`, where `_whiz` is implemented with the binary module `sources/foo/bar/_whiz.abi3.so`, with `sources` being the location registered on `sys.path`, relative to the application bundle. This module *must* be distributed as `Frameworks/foo.bar._whiz.framework/foo.bar._whiz` (creating the framework name from the full import path of the module), with an `Info.plist` file in the `.framework` directory identifying the binary as a framework. The `foo.bar._whiz` module would be represented in the original location with a `sources/foo/bar/_whiz.abi3.fwork` marker file, containing the path `Frameworks/foo.bar._whiz/foo.bar._whiz`. The framework would also contain `Frameworks/foo.bar._whiz.framework/foo.bar._whiz.origin`, containing the path to the `.fwork` file.

When a module is loaded with this loader, the `__file__` for the module will report as the location of the `.fwork` file. This allows code to use the `__file__` of a module as an anchor for file system traversal. However, the spec origin will reference the location of the *actual* binary in the `.framework` folder.

The Xcode project building the app is responsible for converting any `.so` files from wherever they exist in the `PYTHONPATH` into frameworks in the `Frameworks` folder (including stripping extensions from the module file, the addition of framework metadata, and signing the resulting framework), and creating the `.fwork` and `.origin` files. This will usually be done with a build step in the Xcode project; see the iOS documentation for details on how to construct this build step.

Added in version 3.13.

Availability: iOS.

name

Name of the module the loader supports.

path

Path to the `.fwork` file for the extension module.

`importlib.util` – Utility code for importers

Source code: Lib/importlib/util.py [https://github.com/python/cpython/tree/3.14/Lib/importlib/util.py]

This module contains the various objects that help in the construction of an importer.

importlib.util.MAGIC_NUMBER

The bytes which represent the bytecode version number. If you need help with loading/writing bytecode then consider **`importlib.abc.SourceLoader`**.

Added in version 3.4.

importlib.util.cache_from_source(*path*, *debug_override=None*, *, *optimization=None*)

Return the **PEP 3147** [https://peps.python.org/pep-3147/]/**PEP 488** [https://peps.python.org/pep-0488/] path to the byte-compiled file associated with the source *path*. For example, if *path* is `/foo/bar/baz.py` the return value would be `/foo/bar/__pycache__/baz.cpython-32.pyc` for Python 3.2. The `cpython-32` string comes from the current magic tag (see **`get_tag()`**; if **`sys.implementation.cache_tag`** is not defined then **`NotImplementedError`** will be raised).

The *optimization* parameter is used to specify the optimization level of the bytecode file. An empty string represents no optimization, so `/foo/bar/baz.py` with an *optimization* of `''` will result in a bytecode path of `/foo/bar/__pycache__/baz.cpython-32.pyc`. `None` causes the interpreter's optimization level to be used. Any other value's string representation is used, so `/foo/bar/baz.py` with an *optimization* of `2` will lead to the bytecode path of `/foo/bar/__pycache__/baz.cpython-32.opt-2.pyc`. The string representation of *optimization* can only be alphanumeric, else **`ValueError`** is raised.

The *debug_override* parameter is deprecated and can be used to override the system's value for `__debug__`. A `True` value is the equivalent of setting *optimization* to the empty string. A `False` value is the same as setting *optimization* to `1`. If both *debug_override* an *optimization* are not `None` then **`TypeError`** is raised.

Added in version 3.4.

Changed in version 3.5: The *optimization* parameter was added and the *debug_override* parameter was deprecated.

Changed in version 3.6: Accepts a path-like object.

importlib.util.source_from_cache(*path*)

Given the *path* to a **PEP 3147** [https://peps.python.org/pep-3147/] file name, return the associated source code file path. For example, if *path* is `/foo/bar/__pycache__/baz.cpython-32.pyc` the returned path would be `/foo/bar/baz.py`. *path* need not exist, however if it does not conform to **PEP 3147** [https://peps.python.org/pep-3147/] or **PEP 488** [https://peps.python.org/pep-0488/] format, a **`ValueError`** is raised. If **`sys.implementation.cache_tag`** is not defined, **`NotImplementedError`** is raised.

Added in version 3.4.

Changed in version 3.6: Accepts a path-like object.

importlib.util.decode_source(*source_bytes*)

Decode the given bytes representing source code and return it as a string with universal newlines (as required by **`importlib.abc.InspectLoader.get_source()`**).

Added in version 3.4.

importlib.util.resolve_name(*name*, *package*)

Resolve a relative module name to an absolute one.

If **name** has no leading dots, then **name** is simply returned. This allows for usage such as `importlib.util.resolve_name('sys', __spec__.parent)` without doing a check to see if the **package** argument is needed.

`ImportError` is raised if **name** is a relative module name but **package** is a false value (e.g. `None` or the empty string). **`ImportError`** is also raised if a relative name would escape its containing package (e.g. requesting `..bacon` from within the `spam` package).

Added in version 3.3.

Changed in version 3.9: To improve consistency with import statements, raise **`ImportError`** instead of **`ValueError`** for invalid relative import attempts.

importlib.util.find_spec(*name*, *package=None*)

Find the spec for a module, optionally relative to the specified **package** name. If the module is in `sys.modules`, then `sys.modules[name].__spec__` is returned (unless the spec would be `None` or is not set, in which case `ValueError` is raised). Otherwise a search using `sys.meta_path` is done. `None` is returned if no spec is found.

If **name** is for a submodule (contains a dot), the parent module is automatically imported.

name and **package** work the same as for `import_module()`.

Added in version 3.4.

Changed in version 3.7: Raises `ModuleNotFoundError` instead of `AttributeError` if **package** is in fact not a package (i.e. lacks a `__path__` attribute).

importlib.util.module_from_spec(*spec*)

Create a new module based on **spec** and `spec.loader.create_module`.

If `spec.loader.create_module` does not return `None`, then any pre-existing attributes will not be reset. Also, no `AttributeError` will be raised if triggered while accessing **spec** or setting an attribute on the module.

This function is preferred over using `types.ModuleType` to create a new module as **spec** is used to set as many import-controlled attributes on the module as possible.

Added in version 3.5.

importlib.util.spec_from_loader(*name*, *loader*, *, *origin=None*, *is_package=None*)

A factory function for creating a **`ModuleSpec`** instance based on a loader. The parameters have the same meaning as they do for ModuleSpec. The function uses available loader APIs, such as **`InspectLoader.is_package()`**, to fill in any missing information on the spec.

Added in version 3.4.

importlib.util.spec_from_file_location(*name*, *location*, *, *loader=None*, *submodule_search_locations=None*)

A factory function for creating a **`ModuleSpec`** instance based on the path to a file. Missing information will be filled in on the spec by making use of loader APIs and by the implication that the module will be file-based.

Added in version 3.4.

Changed in version 3.6: Accepts a path-like object.

importlib.util.source_hash(*source_bytes*)

Return the hash of *source_bytes* as bytes. A hash-based `.pyc` file embeds the **`source_hash()`** of the corresponding source file's contents in its header.

Added in version 3.7.

importlib.util._incompatible_extension_module_restrictions(*, *disable_check*)

A context manager that can temporarily skip the compatibility check for extension modules. By default the check is enabled and will fail when a single-phase init module is imported in a subinterpreter. It will also fail for a multi-phase init module that doesn't explicitly support a per-interpreter GIL, when imported in an interpreter with its own GIL.

Note that this function is meant to accommodate an unusual case; one which is likely to eventually go away. There’s is a pretty good chance this is not what you were looking for.

You can get the same effect as this function by implementing the basic interface of multi-phase init (**PEP 489** [https://peps.python.org/pep-0489/]) and lying about support for multiple interpreters (or per-interpreter GIL).

Warning

Using this function to disable the check can lead to unexpected behavior and even crashes. It should only be used during extension module development.

Added in version 3.12.

class importlib.util.LazyLoader(*loader*)

A class which postpones the execution of the loader of a module until the module has an attribute accessed.

This class **only** works with loaders that define `exec_module()` as control over what module type is used for the module is required. For those same reasons, the loader’s `create_module()` method must return `None` or a type for which its `__class__` attribute can be mutated along with not using slots. Finally, modules which substitute the object placed into `sys.modules` will not work as there is no way to properly replace the module references throughout the interpreter safely; `ValueError` is raised if such a substitution is detected.

Note

For projects where startup time is critical, this class allows for potentially minimizing the cost of loading a module if it is never used. For projects where startup time is not essential then use of this class is **heavily** discouraged due to error messages created during loading being postponed and thus occurring out of context.

Added in version 3.5.

Changed in version 3.6: Began calling `create_module()`, removing the compatibility warning for `importlib.machinery.BuiltinImporter` and `importlib.machinery.ExtensionFileLoader`.

classmethod factory(*loader*)

A class method which returns a callable that creates a lazy loader. This is meant to be used in situations where the loader is passed by class instead of by instance.

```
suffixes = importlib.machinery.SOURCE_SUFFIXES
loader = importlib.machinery.SourceFileLoader
lazy_loader = importlib.util.LazyLoader.factory(loader)
finder = importlib.machinery.FileFinder(path,
(lazy_loader, suffixes))
```

Examples

Importing programmatically

To programmatically import a module, use `importlib.import_module()`.

```
import importlib

itertools = importlib.import_module('itertools')
```

Checking if a module can be imported

If you need to find out if a module can be imported without actually doing the import, then you should use `importlib.util.find_spec()`.

Note that if `name` is a submodule (contains a dot), `importlib.util.find_spec()` will import the parent module.

```
import importlib.util
import sys

# For illustrative purposes.
name = 'itertools'

if name in sys.modules:
    print(f"{name!r} already in sys.modules")
elif (spec := importlib.util.find_spec(name)) is not None:
    # If you chose to perform the actual import ...
    module = importlib.util.module_from_spec(spec)
    sys.modules[name] = module
    spec.loader.exec_module(module)
    print(f"{name!r} has been imported")
else:
    print(f"can't find the {name!r} module")
```

Importing a source file directly

This recipe should be used with caution: it is an approximation of an import statement where the file path is specified directly, rather than `sys.path` being searched. Alternatives should first be considered first, such as modifying `sys.path` when a proper module is required, or using `runpy.run_path()` when the global namespace resulting from running a Python file is appropriate.

To import a Python source file directly from a path, use the following recipe:

```
import importlib.util
import sys


def import_from_path(module_name, file_path):
    spec = importlib.util.spec_from_file_location(module_name,
file_path)
    module = importlib.util.module_from_spec(spec)
    sys.modules[module_name] = module
    spec.loader.exec_module(module)
    return module


# For illustrative purposes only (use of `json` is arbitrary).
import json
file_path = json.__file__
module_name = json.__name__

# Similar outcome as `import json`.
json = import_from_path(module_name, file_path)
```

Implementing lazy imports

The example below shows how to implement lazy imports:

```
>>> import importlib.util
>>> import sys
>>> def lazy_import(name):
...     spec = importlib.util.find_spec(name)
...     loader = importlib.util.LazyLoader(spec.loader)
...     spec.loader = loader
...     module = importlib.util.module_from_spec(spec)
...     sys.modules[name] = module
...     loader.exec_module(module)
...     return module
...
>>> lazy_typing = lazy_import("typing")
>>> #lazy_typing is a real module object,
>>> #but it is not loaded in memory yet.
>>> lazy_typing.TYPE_CHECKING
False
```

Setting up an importer

For deep customizations of import, you typically want to implement an importer. This means managing both the finder and loader side of things. For finders there are two flavours to choose from depending on your needs: a meta path finder or a path entry finder. The former is what you would put on **`sys.meta_path`** while the latter is what you create using a path entry hook on **`sys.path_hooks`** which works with **`sys.path`** entries to potentially create a finder. This example will show you how to register your own importers so that import will use them (for creating an importer for yourself, read the documentation for the appropriate classes defined within this package):

```
import importlib.machinery
import sys

# For illustrative purposes only.
SpamMetaPathFinder = importlib.machinery.PathFinder
SpamPathEntryFinder = importlib.machinery.FileFinder
loader_details = (importlib.machinery.SourceFileLoader,
                  importlib.machinery.SOURCE_SUFFIXES)

# Setting up a meta path finder.
# Make sure to put the finder in the proper location in the
list in terms of
# priority.
sys.meta_path.append(SpamMetaPathFinder)

# Setting up a path entry finder.
# Make sure to put the path hook in the proper location in the
list in terms
# of priority.
sys.path_hooks.append(SpamPathEntryFinder.path_hook(loader_deta
ils))
```

Approximating `importlib.import_module()`

Import itself is implemented in Python code, making it possible to expose most of the import machinery through importlib. The following helps illustrate the various APIs that importlib exposes by providing an approximate implementation of `importlib.import_module()`:

```
import importlib.util
import sys

def import_module(name, package=None):
    """An approximate implementation of import."""
    absolute_name = importlib.util.resolve_name(name, package)
    try:
        return sys.modules[absolute_name]
    except KeyError:
        pass

    path = None
    if '.' in absolute_name:
        parent_name, _, child_name =
absolute_name.rpartition('.')
        parent_module = import_module(parent_name)
        path =
parent_module.__spec__.submodule_search_locations
    for finder in sys.meta_path:
        spec = finder.find_spec(absolute_name, path)
        if spec is not None:
            break
    else:
        msg = f'No module named {absolute_name!r}'
        raise ModuleNotFoundError(msg, name=absolute_name)
    module = importlib.util.module_from_spec(spec)
    sys.modules[absolute_name] = module
    spec.loader.exec_module(module)
    if path is not None:
        setattr(parent_module, child_name, module)
    return module
```

`importlib.resources` – Package resource reading, opening and access

Source code: Lib/importlib/resources/__init__.py [https://github.com/python/cpython/tree/3.14/Lib/importlib/resources/__init__.py]

Added in version 3.7.

This module leverages Python’s import system to provide access to *resources* within *packages*.

“Resources” are file-like resources associated with a module or package in Python. The resources may be contained directly in a package, within a subdirectory contained in that package, or adjacent to modules outside a package. Resources may be text or binary. As a result, a package’s Python module sources (.py), compilation artifacts (pycache), and installation artifacts (like `reserved filenames` in directories) are technically de-facto resources of that package. In practice, however, resources are primarily those non-Python artifacts exposed specifically by the package author.

Resources can be opened or read in either binary or text mode.

Resources are roughly akin to files inside directories, though it’s important to keep in mind that this is just a metaphor. Resources and packages **do not** have to exist as physical files and directories on the file system: for example, a package and its resources can be imported from a zip file using **`zipimport`**.

Note

This module provides functionality similar to pkg_resources [https://setuptools.readthedocs.io/en/latest/pkg_resources.html] Basic Resource Access [https://setuptools.readthedocs.io/en/latest/pkg_resources.html#basic-resource-access] without the performance overhead of that package. This makes reading resources included in packages easier, with more stable and consistent semantics.

The standalone backport of this module provides more information on using importlib.resources [https://importlib-resources.readthedocs.io/en/latest/using.html] and migrating from pkg_resources to importlib.resources [https://importlib-resources.readthedocs.io/en/latest/migration.html].

`Loaders` that wish to support resource reading should implement a `get_resource_reader(fullname)` method as specified by **`importlib.resources.abc.ResourceReader`**.

class importlib.resources.Anchor

Represents an anchor for resources, either a **`module object`** or a module name as a string. Defined as `Union[str, ModuleType]`.

importlib.resources.files(*anchor: Anchor* | *None* = *None*)

Returns a **`Traversable`** object representing the resource container (think directory) and its resources (think files). A Traversable may contain other containers (think subdirectories).

anchor is an optional **`Anchor`**. If the anchor is a package, resources are resolved from that package. If a module, resources are resolved adjacent to that module (in the same package or the package root). If the anchor is omitted, the caller’s module is used.

Added in version 3.9.

Changed in version 3.12: package parameter was renamed to *anchor*. *anchor* can now be a non-package module and if omitted will default to the caller’s module. *package* is still accepted for compatibility but will raise a **`DeprecationWarning`**. Consider passing the anchor positionally or using `importlib_resources >= 5.10` for a compatible interface on older Pythons.

importlib.resources.as_file(*traversable*)

Given a **`Traversable`** object representing a file or directory, typically from **`importlib.resources.files()`**, return a context manager for use in a **`with`** statement. The context manager provides a **`pathlib.Path`** object.

Exiting the context manager cleans up any temporary file or directory created when the resource was extracted from e.g. a zip file.

Use `as_file` when the Traversable methods (`read_text`, etc) are insufficient and an actual file or directory on the file system is required.

Added in version 3.9.

Changed in version 3.12: Added support for *traversable* representing a directory.

Functional API

A set of simplified, backwards-compatible helpers is available. These allow common operations in a single function call.

For all the following functions:

- *anchor* is an **`Anchor`**, as in **`files()`**. Unlike in `files`, it may not be omitted.
- *path_names* are components of a resource’s path name, relative to the anchor. For example, to get the text of resource named `info.txt`, use:

  ```
  importlib.resources.read_text(my_module, "info.txt")
  ```

  Like **`Traversable.joinpath`**, The individual components should use forward slashes (`/`) as path separators. For example, the following are equivalent:

  ```
  importlib.resources.read_binary(my_module,
  "pics/painting.png")
  importlib.resources.read_binary(my_module, "pics",
  "painting.png")
  ```

  For backward compatibility reasons, functions that read text require an explicit *encoding* argument if multiple *path_names* are given. For example, to get the text of `info/chapter1.txt`, use:

  ```
  importlib.resources.read_text(my_module, "info",
  "chapter1.txt",
                                encoding='utf-8')
  ```

importlib.resources.open_binary(*anchor*, **path_names*)

Open the named resource for binary reading.

See the introduction for details on *anchor* and *path_names*.

This function returns a **`BinaryIO`** object, that is, a binary stream open for reading.

This function is roughly equivalent to:

```
files(anchor).joinpath(*path_names).open('rb')
```

Changed in version 3.13: Multiple *path_names* are accepted.

importlib.resources.open_text(*anchor*, **path_names*, *encoding='utf-8'*, *errors='strict'*)

Open the named resource for text reading. By default, the contents are read as strict UTF-8.

See the introduction for details on *anchor* and *path_names*. *encoding* and *errors* have the same meaning as in built-in **`open()`**.

For backward compatibility reasons, the *encoding* argument must be given explicitly if there are multiple *path_names*. This limitation is scheduled to be removed in Python 3.15.

This function returns a **`TextIO`** object, that is, a text stream open for reading.

This function is roughly equivalent to:

```
files(anchor).joinpath(*path_names).open('r',
encoding=encoding)
```

Changed in version 3.13: Multiple *path_names* are accepted. *encoding* and *errors* must be given as keyword arguments.

importlib.resources.read_binary(*anchor*, **path_names*)

Read and return the contents of the named resource as **`bytes`**.

See the introduction for details on *anchor* and *path_names*.

This function is roughly equivalent to:

```
files(anchor).joinpath(*path_names).read_bytes()
```

Changed in version 3.13: Multiple *path_names* are accepted.

importlib.resources.read_text(*anchor*, **path_names*, *encoding='utf-8'*, *errors='strict'*)

Read and return the contents of the named resource as **`str`**. By default, the contents are read as strict UTF-8.

See the introduction for details on *anchor* and *path_names*. *encoding* and *errors* have the same meaning as in built-in **`open()`**.

For backward compatibility reasons, the *encoding* argument must be given explicitly if there are multiple *path_names*. This limitation is scheduled to be removed in Python 3.15.

This function is roughly equivalent to:

```
files(anchor).joinpath(*path_names).read_text(encoding=encod
ing)
```

Changed in version 3.13: Multiple *path_names* are accepted. *encoding* and *errors* must be given as keyword arguments.

importlib.resources.path(*anchor*, **path_names*)

Provides the path to the *resource* as an actual file system path. This function returns a context manager for use in a **`with`** statement. The context manager provides a **`pathlib.Path`** object.

Exiting the context manager cleans up any temporary files created, e.g. when the resource needs to be extracted from a zip file.

For example, the **`stat()`** method requires an actual file system path; it can be used like this:

```
with importlib.resources.path(anchor, "resource.txt") as
fspath:
    result = fspath.stat()
```

See the introduction for details on *anchor* and *path_names*.

This function is roughly equivalent to:

```
as_file(files(anchor).joinpath(*path_names))
```

Changed in version 3.13: Multiple *path_names* are accepted. *encoding* and *errors* must be given as keyword arguments.

importlib.resources.is_resource(*anchor*, **path_names*)

Return `True` if the named resource exists, otherwise `False`. This function does not consider directories to be resources.

See the introduction for details on *anchor* and *path_names*.

This function is roughly equivalent to:

```
files(anchor).joinpath(*path_names).is_file()
```

Changed in version 3.13: Multiple *path_names* are accepted.

importlib.resources.contents(*anchor*, **path_names*)

Return an iterable over the named items within the package or path. The iterable returns names of resources (e.g. files) and non-resources (e.g. directories) as **`str`**. The iterable does not recurse into subdirectories.

See the introduction for details on *anchor* and *path_names*.

This function is roughly equivalent to:

```
for resource in
files(anchor).joinpath(*path_names).iterdir():
    yield resource.name
```

Deprecated since version 3.11: Prefer `iterdir()` as above, which offers more control over the results and richer functionality.

`importlib.resources.abc` – Abstract base classes for resources

Source code: Lib/importlib/resources/abc.py [https://github.com/python/cpython/tree/3.14/Lib/importlib/resources/abc.py]

Added in version 3.11.

class importlib.resources.abc.ResourceReader

Superseded by TraversableResources

An abstract base class to provide the ability to read *resources*.

From the perspective of this ABC, a *resource* is a binary artifact that is shipped within a package. Typically this is something like a data file that lives next to the `__init__.py` file of the package. The purpose of this class is to help abstract out the accessing of such data files so that it does not matter if the package and its data file(s) are stored e.g. in a zip file versus on the file system.

For any of methods of this class, a *resource* argument is expected to be a path-like object which represents conceptually just a file name. This means that no subdirectory paths should be included in the *resource* argument. This is because the location of the package the reader is for, acts as the “directory”. Hence the metaphor for directories and file names is packages and resources, respectively. This is also why instances of this class are expected to directly correlate to a specific package (instead of potentially representing multiple packages or a module).

Loaders that wish to support resource reading are expected to provide a method called `get_resource_reader(fullname)` which returns an object implementing this ABC’s interface. If the module specified by fullname is not a package, this method should return **`None`**. An object compatible with this ABC should only be returned when the specified module is a package.

Deprecated since version 3.12: Use **`importlib.resources.abc.TraversableResources`** instead.

abstractmethod open_resource(*resource*)

Returns an opened, file-like object for binary reading of the *resource*.

If the resource cannot be found, **`FileNotFoundError`** is raised.

abstractmethod resource_path(*resource*)

Returns the file system path to the *resource*.

If the resource does not concretely exist on the file system, raise **`FileNotFoundError`**.

abstractmethod is_resource(*path*)

Returns `True` if the named *path* is considered a resource. **`FileNotFoundError`** is raised if *path* does not exist.

Changed in version 3.10: The argument *name* was renamed to *path*.

abstractmethod contents()

Returns an iterable of strings over the contents of the package. Do note that it is not required that all names returned by the iterator be actual resources, e.g. it is acceptable to return names for which `is_resource()` would be false.

Allowing non-resource names to be returned is to allow for situations where how a package and its resources are stored are known a priori and the non-resource names would be useful. For instance, returning subdirectory names is allowed so that when it is known that the package and resources are stored on the file system then those subdirectory names can be used directly.

The abstract method returns an iterable of no items.

class importlib.resources.abc.Traversable

An object with a subset of `pathlib.Path` methods suitable for traversing directories and opening files.

For a representation of the object on the file-system, use `importlib.resources.as_file()`.

name

Abstract. The base name of this object without any parent references.

abstractmethod iterdir()

Yield Traversable objects in self.

abstractmethod is_dir()

Return `True` if self is a directory.

abstractmethod is_file()

Return `True` if self is a file.

abstractmethod joinpath(**pathsegments*)

Traverse directories according to *pathsegments* and return the result as **`Traversable`**.

Each *pathsegments* argument may contain multiple names separated by forward slashes (`/`, `posixpath.sep`). For example, the following are equivalent:

```
files.joinpath('subdir', 'subsuddir', 'file.txt')
files.joinpath('subdir/subsuddir/file.txt')
```

Note that some **`Traversable`** implementations might not be updated to the latest version of the protocol. For compatibility with such implementations, provide a single argument without path separators to each call to `joinpath`. For example:

```
files.joinpath('subdir').joinpath('subsubdir').joinpath('
file.txt')
```

Changed in version 3.11: `joinpath` accepts multiple *pathsegments*, and these segments may contain forward slashes as path separators. Previously, only a single *child* argument was accepted.

abstractmethod __truediv__(*child*)

Return Traversable child in self. Equivalent to `joinpath(child)`.

abstractmethod open(*mode='r'*, **args*, ***kwargs*)

mode may be ‘r’ or ‘rb’ to open as text or binary. Return a handle suitable for reading (same as **`pathlib.Path.open`**).

When opening as text, accepts encoding parameters such as those accepted by `io.TextIOWrapper`.

read_bytes()

Read contents of self as bytes.

read_text(*encoding=None*)

Read contents of self as text.

class importlib.resources.abc.TraversableResources

An abstract base class for resource readers capable of serving the `importlib.resources.files()` interface. Subclasses `ResourceReader` and provides concrete implementations of the **`ResourceReader`**’s abstract methods. Therefore, any loader supplying **`TraversableResources`** also supplies **`ResourceReader`**.

Loaders that wish to support resource reading are expected to implement this interface.

abstractmethod files()

Returns a `importlib.resources.abc.Traversable` object for the loaded package.

`importlib.metadata` – Accessing package metadata

Added in version 3.8.

Changed in version 3.10: `importlib.metadata` is no longer provisional.

Source code: Lib/importlib/metadata/__init__.py [https://github.com/python/cpython/tree/3.14/Lib/importlib/metadata/__init__.py]

`importlib.metadata` is a library that provides access to the metadata of an installed Distribution Package [https://packaging.python.org/en/latest/glossary/#term-Distribution-Package], such as its entry points or its top-level names (Import Package [https://packaging.python.org/en/latest/glossary/#term-Import-Package]s, modules, if any). Built in part on Python's import system, this library intends to replace similar functionality in the entry point API [https://setuptools.readthedocs.io/en/latest/pkg_resources.html#entry-points] and metadata API [https://setuptools.readthedocs.io/en/latest/pkg_resources.html#metadata-api] of `pkg_resources`. Along with **`importlib.resources`**, this package can eliminate the need to use the older and less efficient `pkg_resources` package.

`importlib.metadata` operates on third-party *distribution packages* installed into Python's `site-packages` directory via tools such as pip [https://pypi.org/project/pip/]. Specifically, it works with distributions with discoverable `dist-info` or `egg-info` directories, and metadata defined by the Core metadata specifications [https://packaging.python.org/en/latest/specifications/core-metadata/#core-metadata].

Important

These are *not* necessarily equivalent to or correspond 1:1 with the top-level *import package* names that can be imported inside Python code. One *distribution package* can contain multiple *import packages* (and single modules), and one top-level *import package* may map to multiple *distribution packages* if it is a namespace package. You can use packages_distributions() to get a mapping between them.

By default, distribution metadata can live on the file system or in zip archives on `sys.path`. Through an extension mechanism, the metadata can live almost anywhere.

See also

https://importlib-metadata.readthedocs.io/
The documentation for `importlib_metadata`, which supplies a backport of `importlib.metadata`. This includes an API reference [https://importlib-metadata.readthedocs.io/en/latest/api.html] for this module’s classes and functions, as well as a migration guide [https://importlib-metadata.readthedocs.io/en/latest/migration.html] for existing users of `pkg_resources`.

Overview

Let’s say you wanted to get the version string for a Distribution Package [https://packaging.python.org/en/latest/glossary/#term-Distribution-Package] you’ve installed using `pip`. We start by creating a virtual environment and installing something into it:

```
$ python -m venv example
$ source example/bin/activate
(example) $ python -m pip install wheel
```

You can get the version string for `wheel` by running the following:

```
(example) $ python
>>> from importlib.metadata import version
>>> version('wheel')
'0.32.3'
```

You can also get a collection of entry points selectable by properties of the EntryPoint (typically ‘group’ or ‘name’), such as `console_scripts`, `distutils.commands` and others. Each group contains a collection of EntryPoint objects.

You can get the metadata for a distribution:

```
>>> list(metadata('wheel'))
['Metadata-Version', 'Name', 'Version', 'Summary', 'Home-page',
'Author', 'Author-email', 'Maintainer', 'Maintainer-email',
'License', 'Project-URL', 'Project-URL', 'Project-URL',
'Keywords', 'Platform', 'Classifier', 'Classifier',
'Classifier', 'Classifier', 'Classifier', 'Classifier',
'Classifier', 'Classifier', 'Classifier', 'Classifier',
'Classifier', 'Classifier', 'Requires-Python', 'Provides-
Extra', 'Requires-Dist', 'Requires-Dist']
```

You can also get a distribution’s version number, list its constituent files, and get a list of the distribution’s Distribution requirements.

exception importlib.metadata.PackageNotFoundError

Subclass of `ModuleNotFoundError` raised by several functions in this module when queried for a distribution package which is not installed in the current Python environment.

Functional API

This package provides the following functionality via its public API.

Entry points

importlib.metadata.entry_points(***select_params*)

Returns a `EntryPoints` instance describing entry points for the current environment. Any given keyword parameters are passed to the **`select()`** method for comparison to the attributes of the individual entry point definitions.

Note: it is not currently possible to query for entry points based on their **`EntryPoint.dist`** attribute (as different **`Distribution`** instances do not currently compare equal, even if they have the same attributes)

class importlib.metadata.EntryPoints

Details of a collection of installed entry points.

Also provides a `.groups` attribute that reports all identified entry point groups, and a `.names` attribute that reports all identified entry point names.

class importlib.metadata.EntryPoint

Details of an installed entry point.

Each **`EntryPoint`** instance has `.name`, `.group`, and `.value` attributes and a `.load()` method to resolve the value. There are also `.module`, `.attr`, and `.extras` attributes for getting the components of the `.value` attribute, and `.dist` for obtaining information regarding the distribution package that provides the entry point.

Query all entry points:

```
>>> eps = entry_points()
```

The **`entry_points()`** function returns a **`EntryPoints`** object, a collection of all **`EntryPoint`** objects with `names` and `groups` attributes for convenience:

```
>>> sorted(eps.groups)
['console_scripts', 'distutils.commands',
'distutils.setup_keywords', 'egg_info.writers',
'setuptools.installation']
```

`EntryPoints` has a **`select()`** method to select entry points matching specific properties. Select entry points in the `console_scripts` group:

```
>>> scripts = eps.select(group='console_scripts')
```

Equivalently, since **`entry_points()`** passes keyword arguments through to select:

```
>>> scripts = entry_points(group='console_scripts')
```

Pick out a specific script named “wheel” (found in the wheel project):

```
>>> 'wheel' in scripts.names
True
>>> wheel = scripts['wheel']
```

Equivalently, query for that entry point during selection:

```
>>> (wheel,) = entry_points(group='console_scripts',
name='wheel')
```

```
>>> (wheel,) = entry_points().select(group='console_scripts',
name='wheel')
```

Inspect the resolved entry point:

```
>>> wheel
EntryPoint(name='wheel', value='wheel.cli:main',
group='console_scripts')
>>> wheel.module
'wheel.cli'
>>> wheel.attr
'main'
>>> wheel.extras
[]
>>> main = wheel.load()
>>> main
<function main at 0x103528488>
```

The `group` and `name` are arbitrary values defined by the package author and usually a client will wish to resolve all entry points for a particular group. Read the setuptools docs [https://setuptools.pypa.io/en/latest/userguide/entry_point.html] for more information on entry points, their definition, and usage.

Changed in version 3.12: The “selectable” entry points were introduced in `importlib_metadata` 3.6 and Python 3.10. Prior to those changes, `entry_points` accepted no parameters and always returned a dictionary of entry points, keyed by group. With `importlib_metadata` 5.0 and Python 3.12, `entry_points` always returns an `EntryPoints` object. See backports.entry_points_selectable [https://pypi.org/project/backports.entry_points_selectable/] for compatibility options.

Changed in version 3.13: `EntryPoint` objects no longer present a tuple-like interface (**`__getitem__()`**).

Distribution metadata

importlib.metadata.metadata(*distribution_name*)

Return the distribution metadata corresponding to the named distribution package as a **`PackageMetadata`** instance.

Raises **`PackageNotFoundError`** if the named distribution package is not installed in the current Python environment.

class importlib.metadata.PackageMetadata

A concrete implementation of the PackageMetadata protocol [https://importlib-metadata.readthedocs.io/en/latest/api.html#importlib_metadata.PackageMetadata].

In addition to providing the defined protocol methods and attributes, subscripting the instance is equivalent to calling the **`get()`** method.

Every Distribution Package [https://packaging.python.org/en/latest/glossary/#term-Distribution-Package] includes some metadata, which you can extract using the **`metadata()`** function:

```
>>> wheel_metadata = metadata('wheel')
```

The keys of the returned data structure name the metadata keywords, and the values are returned unparsed from the distribution metadata:

```
>>> wheel_metadata['Requires-Python']
'>=2.7, !=3.0.*, !=3.1.*, !=3.2.*, !=3.3.*'
```

`PackageMetadata` also presents a **`json`** attribute that returns all the metadata in a JSON-compatible form per **PEP 566** [https://peps.python.org/pep-0566/]:

```
>>> wheel_metadata.json['requires_python']
'>=2.7, !=3.0.*, !=3.1.*, !=3.2.*, !=3.3.*'
```

The full set of available metadata is not described here. See the PyPA Core metadata specification [https://packaging.python.org/en/latest/specifications/core-metadata/#core-metadata] for additional details.

Changed in version 3.10: The `Description` is now included in the metadata when presented through the payload. Line continuation characters have been removed.

The `json` attribute was added.

Distribution versions

importlib.metadata.version(*distribution_name*)

Return the installed distribution package version [https://packaging.python.org/en/latest/specifications/core-metadata/#version] for the named distribution package.

Raises **`PackageNotFoundError`** if the named distribution package is not installed in the current Python environment.

The **`version()`** function is the quickest way to get a Distribution Package [https://packaging.python.org/en/latest/glossary/#term-Distribution-Package]’s version number, as a string:

```
>>> version('wheel')
'0.32.3'
```

Distribution files

importlib.metadata.files(*distribution_name*)

Return the full set of files contained within the named distribution package.

Raises **`PackageNotFoundError`** if the named distribution package is not installed in the current Python environment.

Returns **`None`** if the distribution is found but the installation database records reporting the files associated with the distribution package are missing.

class importlib.metadata.PackagePath

A **`pathlib.PurePath`** derived object with additional `dist`, `size`, and `hash` properties corresponding to the distribution package's installation metadata for that file.

The **`files()`** function takes a Distribution Package [https://packaging.python.org/en/latest/glossary/#term-Distribution-Package] name and returns all of the files installed by this distribution. Each file is reported as a **`PackagePath`** instance. For example:

```
>>> util = [p for p in files('wheel') if 'util.py' in str(p)]
[0]
>>> util
PackagePath('wheel/util.py')
>>> util.size
859
>>> util.dist
<importlib.metadata._hooks.PathDistribution object at
0x101e0cef0>
>>> util.hash
<FileHash mode: sha256 value:
bYkw5oMccfazVCoYQwKkkemoVyMAFoR34mmKBx8R1NI>
```

Once you have the file, you can also read its contents:

```
>>> print(util.read_text())
import base64
import sys
...
def as_bytes(s):
    if isinstance(s, text_type):
        return s.encode('utf-8')
    return s
```

You can also use the **`locate()`** method to get the absolute path to the file:

```
>>> util.locate()
PosixPath('/home/gustav/example/lib/site-
packages/wheel/util.py')
```

In the case where the metadata file listing files (`RECORD` or `SOURCES.txt`) is missing, **`files()`** will return **`None`**. The caller may wish to wrap calls to **`files()`** in always_iterable [https://more-itertools.readthedocs.io/en/stable/api.html#more_itertools.always_iterable] or otherwise guard against this condition if the target distribution is not known to have the metadata present.

Distribution requirements

importlib.metadata.requires(*distribution_name*)

Return the declared dependency specifiers for the named distribution package.

Raises **`PackageNotFoundError`** if the named distribution package is not installed in the current Python environment.

To get the full set of requirements for a Distribution Package [https://packaging.python.org/en/latest/glossary/#term-Distribution-Package], use the **`requires()`** function:

```
>>> requires('wheel')
["pytest (>=3.0.0) ; extra == 'test'", "pytest-cov ; extra ==
'test'"]
```

Mapping import to distribution packages

importlib.metadata.packages_distributions()

Return a mapping from the top level module and import package names found via `sys.meta_path` to the names of the distribution packages (if any) that provide the corresponding files.

To allow for namespace packages (which may have members provided by multiple distribution packages), each top level import name maps to a list of distribution names rather than mapping directly to a single name.

A convenience method to resolve the Distribution Package [https://packaging.python.org/en/latest/glossary/#term-Distribution-Package] name (or names, in the case of a namespace package) that provide each importable top-level Python module or Import Package [https://packaging.python.org/en/latest/glossary/#term-Import-Package]:

```
>>> packages_distributions()
{'importlib_metadata': ['importlib-metadata'], 'yaml':
['PyYAML'], 'jaraco': ['jaraco.classes', 'jaraco.functools'],
...}
```

Some editable installs, do not supply top-level names [https://github.com/pypa/packaging-problems/issues/609], and thus this function is not reliable with such installs.

Added in version 3.10.

Distributions

importlib.metadata.distribution(*distribution_name*)

Return a `Distribution` instance describing the named distribution package.

Raises `PackageNotFoundError` if the named distribution package is not installed in the current Python environment.

class importlib.metadata.Distribution

Details of an installed distribution package.

Note: different `Distribution` instances do not currently compare equal, even if they relate to the same installed distribution and accordingly have the same attributes.

While the module level API described above is the most common and convenient usage, you can get all of that information from the `Distribution` class. `Distribution` is an abstract object that represents the metadata for a Python Distribution Package [https://packaging.python.org/en/latest/glossary/#term-Distribution-Package]. You can get the concrete `Distribution` subclass instance for an installed distribution package by calling the `distribution()` function:

```
>>> from importlib.metadata import distribution
>>> dist = distribution('wheel')
>>> type(dist)
<class 'importlib.metadata.PathDistribution'>
```

Thus, an alternative way to get the version number is through the `Distribution` instance:

```
>>> dist.version
'0.32.3'
```

There are all kinds of additional metadata available on `Distribution` instances:

```
>>> dist.metadata['Requires-Python']
'>=2.7, !=3.0.*, !=3.1.*, !=3.2.*, !=3.3.*'
>>> dist.metadata['License']
'MIT'
```

For editable packages, an `origin` property may present **PEP 610** [https://peps.python.org/pep-0610/] metadata:

```
>>> dist.origin.url
'file:///path/to/wheel-0.32.3.editable-py3-none-any.whl'
```

The full set of available metadata is not described here. See the PyPA Core metadata specification [https://packaging.python.org/en/latest/specifications/core-metadata/#core-metadata] for additional details.

Added in version 3.13: The `.origin` property was added.

Distribution Discovery

By default, this package provides built-in support for discovery of metadata for file system and zip file Distribution Package [https://packaging.python.org/en/latest/glossary/#term-Distribution-Package]s. This metadata finder search defaults to `sys.path`, but varies slightly in how it interprets those values from how other import machinery does. In particular:

- `importlib.metadata` does not honor **`bytes`** objects on `sys.path`.
- `importlib.metadata` will incidentally honor **`pathlib.Path`** objects on `sys.path` even though such values will be ignored for imports.

Implementing Custom Providers

`importlib.metadata` address two API surfaces, one for *consumers* and another for *providers*. Most users are consumers, consuming metadata provided by the packages. There are other use-cases, however, where users wish to expose metadata through some other mechanism, such as alongside a custom importer. Such a use case calls for a *custom provider*.

Because Distribution Package [https://packaging.python.org/en/latest/glossary/#term-Distribution-Package] metadata is not available through **`sys.path`** searches, or package loaders directly, the metadata for a distribution is found through import system finders. To find a distribution package’s metadata, `importlib.metadata` queries the list of meta path finders on **`sys.meta_path`**.

The implementation has hooks integrated into the `PathFinder`, serving metadata for distribution packages found on the file system.

The abstract class **`importlib.abc.MetaPathFinder`** defines the interface expected of finders by Python’s import system. `importlib.metadata` extends this protocol by looking for an optional `find_distributions` callable on the finders from **`sys.meta_path`** and presents this extended interface as the `DistributionFinder` abstract base class, which defines this abstract method:

```
@abc.abstractmethod
def find_distributions(context=DistributionFinder.Context()) ->
Iterable[Distribution]:
    """Return an iterable of all Distribution instances capable
of
    loading the metadata for packages for the indicated
``context``.
    """
```

The `DistributionFinder.Context` object provides `.path` and `.name` properties indicating the path to search and name to match and may supply other relevant context sought by the consumer.

In practice, to support finding distribution package metadata in locations other than the file system, subclass `Distribution` and implement the abstract methods. Then from a custom finder, return instances of this derived `Distribution` in the `find_distributions()` method.

Example

Imagine a custom finder that loads Python modules from a database:

```
class DatabaseImporter(importlib.abc.MetaPathFinder):
    def __init__(self, db):
        self.db = db

    def find_spec(self, fullname, target=None) -> ModuleSpec:
        return self.db.spec_from_name(fullname)

sys.meta_path.append(DatabaseImporter(connect_db(...)))
```

That importer now presumably provides importable modules from a database, but it provides no metadata or entry points. For this custom importer to provide metadata, it would also need to implement `DistributionFinder`:

```
from importlib.metadata import DistributionFinder

class DatabaseImporter(DistributionFinder):
    ...

    def find_distributions(self, 
context=DistributionFinder.Context()):
        query = dict(name=context.name) if context.name else {}
        for dist_record in self.db.query_distributions(query):
            yield DatabaseDistribution(dist_record)
```

In this way, `query_distributions` would return records for each distribution served by the database matching the query. For example, if `requests-1.0` is in the database, `find_distributions` would yield a `DatabaseDistribution` for `Context(name='requests')` or `Context(name=None)`.

For the sake of simplicity, this example ignores `context.path`. The `path` attribute defaults to `sys.path` and is the set of import paths to be considered in the search. A `DatabaseImporter` could potentially function without any concern for a search path. Assuming the importer does no partitioning, the “path” would be irrelevant. In order to illustrate the purpose of `path`, the example would need to illustrate a more complex `DatabaseImporter` whose behavior varied depending on `sys.path`/`PYTHONPATH`. In that case, the `find_distributions` should honor the `context.path` and only yield `Distribution`s pertinent to that path.

`DatabaseDistribution`, then, would look something like:

```
class DatabaseDistribution(importlib.metadata.Distribution):
    def __init__(self, record):
        self.record = record

    def read_text(self, filename):
        """
        Read a file like "METADATA" for the current
distribution.
        """
        if filename == "METADATA":
            return f"""Name: {self.record.name}
Version: {self.record.version}
"""
        if filename == "entry_points.txt":
            return "\n".join(
              f"""[{ep.group}]\n{ep.name}={ep.value}"""
            for ep in self.record.entry_points)

    def locate_file(self, path):
        raise RuntimeError("This distribution has no file
system")
```

This basic implementation should provide metadata and entry points for packages served by the `DatabaseImporter`, assuming that the `record` supplies suitable `.name`, `.version`, and `.entry_points` attributes.

The `DatabaseDistribution` may also provide other metadata files, like `RECORD` (required for `Distribution.files`) or override the implementation of `Distribution.files`. See the source for more inspiration.

The initialization of the `sys.path` module search path

A module search path is initialized when Python starts. This module search path may be accessed at `sys.path`.

The first entry in the module search path is the directory that contains the input script, if there is one. Otherwise, the first entry is the current directory, which is the case when executing the interactive shell, a `-c` command, or `-m` module.

The `PYTHONPATH` environment variable is often used to add directories to the search path. If this environment variable is found then the contents are added to the module search path.

Note

`PYTHONPATH` will affect all installed Python versions/environments. Be wary of setting this in your shell profile or global environment variables. The `site` module offers more nuanced techniques as mentioned below.

The next items added are the directories containing standard Python modules as well as any extension modules that these modules depend on. Extension modules are `.pyd` files on Windows and `.so` files on other platforms. The directory with the platform-independent Python modules is called `prefix`. The directory with the extension modules is called `exec_prefix`.

The **`PYTHONHOME`** environment variable may be used to set the `prefix` and `exec_prefix` locations. Otherwise these directories are found by using the Python executable as a starting point and then looking for various ‘landmark’ files and directories. Note that any symbolic links are followed so the real Python executable location is used as the search starting point. The Python executable location is called `home`.

Once `home` is determined, the `prefix` directory is found by first looking for `python`*`majorversionminorversion`*`.zip` (`python311.zip`). On Windows the zip archive is searched for in `home` and on Unix the archive is expected to be in `lib`. Note that the expected zip archive location is added to the module search path even if the archive does not exist. If no archive was found, Python on Windows will continue the search for `prefix` by looking for `Lib\os.py`. Python on Unix will look for `lib/python`*`majorversion`*`.`*`minorversion`*`/os.py` (`lib/python3.11/os.py`). On Windows `prefix` and `exec_prefix` are the same, however on other platforms `lib/python`*`majorversion`*`.`*`minorversion`*`/lib-dynload` (`lib/python3.11/lib-dynload`) is searched for and used as an anchor for `exec_prefix`. On some platforms `lib` may be `lib64` or another value, see **`sys.platlibdir`** and **`PYTHONPLATLIBDIR`**.

Once found, `prefix` and `exec_prefix` are available at **`sys.base_prefix`** and **`sys.base_exec_prefix`** respectively.

If **`PYTHONHOME`** is not set, and a `pyvenv.cfg` file is found alongside the main executable, or in its parent directory, **`sys.prefix`** and **`sys.exec_prefix`** get set to the directory containing `pyvenv.cfg`, otherwise they are set to the same value as **`sys.base_prefix`** and **`sys.base_exec_prefix`**, respectively. This is used by Virtual Environments.

Finally, the **`site`** module is processed and `site-packages` directories are added to the module search path. A common way to customize the search path is to create `sitecustomize` or `usercustomize` modules as described in the `site` module documentation.

Note

Certain command line options may further affect path calculations. See `-E`, `-I`, `-s` and `-S` for further details.

Changed in version 3.14: `sys.prefix` and `sys.exec_prefix` are now set to the `pyvenv.cfg` directory during the path initialization. This was previously done by `site`, therefore affected by `-S`.

Virtual Environments

Virtual environments place a `pyvenv.cfg` file in their prefix, which causes `sys.prefix` and `sys.exec_prefix` to point to them, instead of the base installation.

The `prefix` and `exec_prefix` values of the base installation are available at `sys.base_prefix` and `sys.base_exec_prefix`.

As well as being used as a marker to identify virtual environments, `pyvenv.cfg` may also be used to configure the `site` initialization. Please refer to `site`’s virtual environments documentation.

Note

`PYTHONHOME` overrides the `pyvenv.cfg` detection.

Note

There are other ways how “virtual environments” could be implemented, this documentation refers implementations based on the `pyvenv.cfg` mechanism, such as **`venv`**. Most virtual environment implementations follow the model set by **`venv`**, but there may be exotic implementations that diverge from it.

_pth files

To completely override **`sys.path`** create a `._pth` file with the same name as the shared library or executable (`python._pth` or `python311._pth`). The shared library path is always known on Windows, however it may not be available on other platforms. In the `._pth` file specify one line for each path to add to **`sys.path`**. The file based on the shared library name overrides the one based on the executable, which allows paths to be restricted for any program loading the runtime if desired.

When the file exists, all registry and environment variables are ignored, isolated mode is enabled, and **`site`** is not imported unless one line in the file specifies `import site`. Blank paths and lines starting with `#` are ignored. Each path may be absolute or relative to the location of the file. Import statements other than to `site` are not permitted, and arbitrary code cannot be specified.

Note that `.pth` files (without leading underscore) will be processed normally by the **`site`** module when `import site` has been specified.

Embedded Python

If Python is embedded within another application `Py_InitializeFromConfig()` and the `PyConfig` structure can be used to initialize Python. The path specific details are described at Python Path Configuration.

See also

- Finding modules for detailed Windows notes.
- Using Python on Unix platforms for Unix details.

Python Language Services

Python provides a number of modules to assist in working with the Python language. These modules support tokenizing, parsing, syntax analysis, bytecode disassembly, and various other facilities.

These modules include:

- `ast` — Abstract syntax trees
  - Abstract grammar
  - Node classes
    - Root nodes
    - Literals
    - Variables
    - Expressions
      - Subscripting
      - Comprehensions
    - Statements
      - Imports
    - Control flow
    - Pattern matching
    - Type annotations
    - Type parameters
    - Function and class definitions
    - Async and await
  - `ast` helpers
  - Compiler flags
  - Command-line usage
- `symtable` — Access to the compiler’s symbol tables
  - Generating Symbol Tables
  - Examining Symbol Tables
  - Command-Line Usage
- `token` — Constants used with Python parse trees

- `keyword` — Testing for Python keywords
- `tokenize` — Tokenizer for Python source
  - Tokenizing Input
  - Command-Line Usage
  - Examples
- `tabnanny` — Detection of ambiguous indentation
- `pyclbr` — Python module browser support
  - Function Objects
  - Class Objects
- `py_compile` — Compile Python source files
  - Command-Line Interface
- `compileall` — Byte-compile Python libraries
  - Command-line use
  - Public functions
- `dis` — Disassembler for Python bytecode
  - Command-line interface
  - Bytecode analysis
  - Analysis functions
  - Python Bytecode Instructions
  - Opcode collections
- `pickletools` — Tools for pickle developers
  - Command-line usage
    - Command-line options
  - Programmatic interface

`ast` — Abstract syntax trees

Source code: Lib/ast.py [https://github.com/python/cpython/tree/3.14/Lib/ast.py]

The `ast` module helps Python applications to process trees of the Python abstract syntax grammar. The abstract syntax itself might change with each Python release; this module helps to find out programmatically what the current grammar looks like.

An abstract syntax tree can be generated by passing `ast.PyCF_ONLY_AST` as a flag to the `compile()` built-in function, or using the `parse()` helper provided in this module. The result will be a tree of objects whose classes all inherit from `ast.AST`. An abstract syntax tree can be compiled into a Python code object using the built-in `compile()` function.

Abstract grammar

The abstract grammar is currently defined as follows:

```
-- ASDL's 4 builtin types are:
-- identifier, int, string, constant

module Python
{
    mod = Module(stmt* body, type_ignore* type_ignores)
        | Interactive(stmt* body)
        | Expression(expr body)
        | FunctionType(expr* argtypes, expr returns)
    stmt = FunctionDef(identifier name, arguments args,
                       stmt* body, expr* decorator_list, expr?
returns,
                       string? type_comment, type_param*
type_params)
          | AsyncFunctionDef(identifier name, arguments args,
                             stmt* body, expr* decorator_list,
expr? returns,
                             string? type_comment, type_param*
type_params)

          | ClassDef(identifier name,
             expr* bases,
             keyword* keywords,
             stmt* body,
             expr* decorator_list,
             type_param* type_params)
          | Return(expr? value)

          | Delete(expr* targets)
          | Assign(expr* targets, expr value, string?
type_comment)
          | TypeAlias(expr name, type_param* type_params, expr
value)
          | AugAssign(expr target, operator op, expr value)
          -- 'simple' indicates that we annotate simple name
without parens
          | AnnAssign(expr target, expr annotation, expr?
value, int simple)

          -- use 'orelse' because else is a keyword in target
languages
          | For(expr target, expr iter, stmt* body, stmt*
orelse, string? type_comment)
          | AsyncFor(expr target, expr iter, stmt* body, stmt*
orelse, string? type_comment)
          | While(expr test, stmt* body, stmt* orelse)
          | If(expr test, stmt* body, stmt* orelse)
          | With(withitem* items, stmt* body, string?
type_comment)
          | AsyncWith(withitem* items, stmt* body, string?
type_comment)
```

```
          | Match(expr subject, match_case* cases)

          | Raise(expr? exc, expr? cause)
          | Try(stmt* body, excepthandler* handlers, stmt*
orelse, stmt* finalbody)
          | TryStar(stmt* body, excepthandler* handlers, stmt*
orelse, stmt* finalbody)
          | Assert(expr test, expr? msg)

          | Import(alias* names)
          | ImportFrom(identifier? module, alias* names, int?
level)

          | Global(identifier* names)
          | Nonlocal(identifier* names)
          | Expr(expr value)
          | Pass | Break | Continue

          -- col_offset is the byte offset in the utf8 string
the parser uses
          attributes (int lineno, int col_offset, int?
end_lineno, int? end_col_offset)

          -- BoolOp() can use left & right?
    expr = BoolOp(boolop op, expr* values)
         | NamedExpr(expr target, expr value)
         | BinOp(expr left, operator op, expr right)
         | UnaryOp(unaryop op, expr operand)
         | Lambda(arguments args, expr body)
         | IfExp(expr test, expr body, expr orelse)
         | Dict(expr?* keys, expr* values)
         | Set(expr* elts)
         | ListComp(expr elt, comprehension* generators)
         | SetComp(expr elt, comprehension* generators)
         | DictComp(expr key, expr value, comprehension*
generators)
         | GeneratorExp(expr elt, comprehension* generators)
         -- the grammar constrains where yield expressions can
occur
         | Await(expr value)
         | Yield(expr? value)
         | YieldFrom(expr value)
         -- need sequences for compare to distinguish between
         -- x < 4 < 3 and (x < 4) < 3
         | Compare(expr left, cmpop* ops, expr* comparators)
         | Call(expr func, expr* args, keyword* keywords)
         | FormattedValue(expr value, int conversion, expr?
format_spec)
         | Interpolation(expr value, constant str, int
conversion, expr? format_spec)
         | JoinedStr(expr* values)
         | TemplateStr(expr* values)
         | Constant(constant value, string? kind)

         -- the following expression can appear in assignment
context
         | Attribute(expr value, identifier attr, expr_context
ctx)
         | Subscript(expr value, expr slice, expr_context ctx)
         | Starred(expr value, expr_context ctx)
         | Name(identifier id, expr_context ctx)
         | List(expr* elts, expr_context ctx)
         | Tuple(expr* elts, expr_context ctx)

         -- can appear only in Subscript
         | Slice(expr? lower, expr? upper, expr? step)

          -- col_offset is the byte offset in the utf8 string
the parser uses
          attributes (int lineno, int col_offset, int?
end_lineno, int? end_col_offset)

    expr_context = Load | Store | Del

    boolop = And | Or

    operator = Add | Sub | Mult | MatMult | Div | Mod | Pow |
LShift
                 | RShift | BitOr | BitXor | BitAnd | FloorDiv

    unaryop = Invert | Not | UAdd | USub

    cmpop = Eq | NotEq | Lt | LtE | Gt | GtE | Is | IsNot | In
| NotIn

    comprehension = (expr target, expr iter, expr* ifs, int
is_async)

    excepthandler = ExceptHandler(expr? type, identifier? name,
stmt* body)
                    attributes (int lineno, int col_offset,
int? end_lineno, int? end_col_offset)

    arguments = (arg* posonlyargs, arg* args, arg? vararg, arg*
kwonlyargs,
                 expr?* kw_defaults, arg? kwarg, expr*
defaults)

    arg = (identifier arg, expr? annotation, string?
type_comment)
           attributes (int lineno, int col_offset, int?
end_lineno, int? end_col_offset)

    -- keyword arguments supplied to call (NULL identifier for
**kwargs)
    keyword = (identifier? arg, expr value)
               attributes (int lineno, int col_offset, int?
end_lineno, int? end_col_offset)

    -- import name with optional 'as' alias.
    alias = (identifier name, identifier? asname)
             attributes (int lineno, int col_offset, int?
end_lineno, int? end_col_offset)

    withitem = (expr context_expr, expr? optional_vars)

    match_case = (pattern pattern, expr? guard, stmt* body)

    pattern = MatchValue(expr value)
            | MatchSingleton(constant value)
            | MatchSequence(pattern* patterns)
            | MatchMapping(expr* keys, pattern* patterns,
identifier? rest)
            | MatchClass(expr cls, pattern* patterns,
identifier* kwd_attrs, pattern* kwd_patterns)

            | MatchStar(identifier? name)
            -- The optional "rest" MatchMapping parameter
handles capturing extra mapping keys
```

```
            | MatchAs(pattern? pattern, identifier? name)
            | MatchOr(pattern* patterns)

             attributes (int lineno, int col_offset, int 
end_lineno, int end_col_offset)

    type_ignore = TypeIgnore(int lineno, string tag)

    type_param = TypeVar(identifier name, expr? bound, expr? 
default_value)
               | ParamSpec(identifier name, expr? 
default_value)
               | TypeVarTuple(identifier name, expr? 
default_value)
               attributes (int lineno, int col_offset, int 
end_lineno, int end_col_offset)
}
```

Node classes

class ast.AST

This is the base of all AST node classes. The actual node classes are derived from the `Parser/Python.asdl` file, which is reproduced above. They are defined in the `_ast` C module and re-exported in `ast`.

There is one class defined for each left-hand side symbol in the abstract grammar (for example, `ast.stmt` or `ast.expr`). In addition, there is one class defined for each constructor on the right-hand side; these classes inherit from the classes for the left-hand side trees. For example, `ast.BinOp` inherits from `ast.expr`. For production rules with alternatives (aka “sums”), the left-hand side class is abstract: only instances of specific constructor nodes are ever created.

_fields

Each concrete class has an attribute `_fields` which gives the names of all child nodes.

Each instance of a concrete class has one attribute for each child node, of the type as defined in the grammar. For example, `ast.BinOp` instances have an attribute **`left`** of type **`ast.expr`**.

If these attributes are marked as optional in the grammar (using a question mark), the value might be `None`. If the attributes can have zero-or-more values (marked with an asterisk), the values are represented as Python lists. All possible attributes must be present and have valid values when compiling an AST with `compile()`.

_field_types

The **`_field_types`** attribute on each concrete class is a dictionary mapping field names (as also listed in `_fields`) to their types.

```
>>> ast.TypeVar._field_types
{'name': <class 'str'>, 'bound': ast.expr | None,
'default_value': ast.expr | None}
```

Added in version 3.13.

lineno
col_offset
end_lineno
end_col_offset

Instances of **`ast.expr`** and **`ast.stmt`** subclasses have `lineno`, `col_offset`, `end_lineno`, and `end_col_offset` attributes. The `lineno` and `end_lineno` are the first and last line numbers of source text span (1-indexed so the first line is line 1) and the `col_offset` and `end_col_offset` are the corresponding UTF-8 byte offsets of the first and last tokens that generated the node. The UTF-8 offset is recorded because the parser uses UTF-8 internally.

Note that the end positions are not required by the compiler and are therefore optional. The end offset is *after* the last symbol, for example one can get the source segment of a one-line expression node using `source_line[node.col_offset : node.end_col_offset]`.

The constructor of a class **`ast.T`** parses its arguments as follows:

- If there are positional arguments, there must be as many as there are items in **`T._fields`**; they will be assigned as attributes of these names.
- If there are keyword arguments, they will set the attributes of the same names to the given values.

For example, to create and populate an `ast.UnaryOp` node, you could use

```
node = ast.UnaryOp(ast.USub(), ast.Constant(5, lineno=0,
col_offset=0),
                   lineno=0, col_offset=0)
```

If a field that is optional in the grammar is omitted from the constructor, it defaults to `None`. If a list field is omitted, it defaults to the empty list. If a field of type **`ast.expr_context`** is omitted, it defaults to `Load()`. If any other field is omitted, a `DeprecationWarning` is raised and the AST node will not have this field. In Python 3.15, this condition will raise an error.

Changed in version 3.8: Class `ast.Constant` is now used for all constants.

Changed in version 3.9: Simple indices are represented by their value, extended slices are represented as tuples.

Changed in version 3.14: The `__repr__()` output of `AST` nodes includes the values of the node fields.

Deprecated since version 3.8, removed in version 3.14: Previous versions of Python provided the AST classes **`ast.Num`**, **`ast.Str`**, **`ast.Bytes`**, **`ast.NameConstant`** and **`ast.Ellipsis`**, which were deprecated in Python 3.8. These classes were removed in Python 3.14, and their functionality has been replaced with `ast.Constant`.

Deprecated since version 3.9: Old classes **`ast.Index`** and **`ast.ExtSlice`** are still available, but they will be removed in future Python releases. In the meantime, instantiating them will return an instance of a different class.

Deprecated since version 3.13, will be removed in version 3.15: Previous versions of Python allowed the creation of AST nodes that were missing required fields. Similarly, AST node constructors allowed arbitrary keyword arguments that were set as attributes of the AST node, even if they did not match any of the fields of the AST node. This behavior is deprecated and will be removed in Python 3.15.

Note

The descriptions of the specific node classes displayed here were initially adapted from the fantastic Green Tree Snakes [https://greentreesnakes.readthedocs.io/en/latest/] project and all its contributors.

Root nodes

class ast.Module(*body*, *type_ignores*)

A Python module, as with file input. Node type generated by **`ast.parse()`** in the default `"exec"` *mode*.

`body` is a **`list`** of the module’s Statements.

`type_ignores` is a **`list`** of the module’s type ignore comments; see **`ast.parse()`** for more details.

```
>>> print(ast.dump(ast.parse('x = 1'), indent=4))
Module(
    body=[
        Assign(
            targets=[
                Name(id='x', ctx=Store())],
            value=Constant(value=1))])
```

class ast.Expression(*body*)

A single Python expression input. Node type generated by **`ast.parse()`** when *mode* is `"eval"`.

`body` is a single node, one of the expression types.

```
>>> print(ast.dump(ast.parse('123', mode='eval'), indent=4))
Expression(
    body=Constant(value=123))
```

class ast.Interactive(*body*)

A single interactive input, like in Interactive Mode. Node type generated by **`ast.parse()`** when *mode* is `"single"`.

`body` is a **`list`** of statement nodes.

```
>>> print(ast.dump(ast.parse('x = 1; y = 2', mode='single'),
indent=4))
Interactive(
    body=[
        Assign(
            targets=[
                Name(id='x', ctx=Store())],
            value=Constant(value=1)),
        Assign(
            targets=[
                Name(id='y', ctx=Store())],
            value=Constant(value=2))])
```

class ast.FunctionType(*argtypes*, *returns*)

A representation of an old-style type comments for functions, as Python versions prior to 3.5 didn't support **PEP 484** [https://peps.python.org/pep-0484/] annotations. Node type generated by **`ast.parse()`** when *mode* is `"func_type"`.

Such type comments would look like this:

```
def sum_two_number(a, b):
    # type: (int, int) -> int
    return a + b
```

`argtypes` is a **`list`** of expression nodes.

`returns` is a single expression node.

```
>>> print(ast.dump(ast.parse('(int, str) -> List[int]',
mode='func_type'), indent=4))
FunctionType(
    argtypes=[
        Name(id='int', ctx=Load()),
        Name(id='str', ctx=Load())],
    returns=Subscript(
        value=Name(id='List', ctx=Load()),
        slice=Name(id='int', ctx=Load()),
        ctx=Load()))
```

Added in version 3.8.

Literals

class ast.Constant(*value*)

A constant value. The `value` attribute of the `Constant` literal contains the Python object it represents. The values represented can be instances of **`str`**, **`bytes`**, **`int`**, **`float`**, **`complex`**, and **`bool`**, and the constants **`None`** and **`Ellipsis`**.

```
>>> print(ast.dump(ast.parse('123', mode='eval'), indent=4))
Expression(
    body=Constant(value=123))
```

class ast.FormattedValue(*value*, *conversion*, *format_spec*)

Node representing a single formatting field in an f-string. If the string contains a single formatting field and nothing else the node can be isolated otherwise it appears in **`JoinedStr`**.

- `value` is any expression node (such as a literal, a variable, or a function call).
- `conversion` is an integer:
    - -1: no formatting
    - 97 (`ord('a')`): `!a` **`ASCII`** formatting
    - 114 (`ord('r')`): `!r` **`repr()`** formatting
    - 115 (`ord('s')`): `!s` **`string`** formatting
- `format_spec` is a **`JoinedStr`** node representing the formatting of the value, or `None` if no format was specified. Both `conversion` and `format_spec` can be set at the same time.

class ast.JoinedStr(*values*)

An f-string, comprising a series of **`FormattedValue`** and **`Constant`** nodes.

```
>>> print(ast.dump(ast.parse('f"sin({a}) is {sin(a):.3}"',
mode='eval'), indent=4))
Expression(
    body=JoinedStr(
        values=[
            Constant(value='sin('),
            FormattedValue(
                value=Name(id='a', ctx=Load()),
                conversion=-1),
            Constant(value=') is '),
            FormattedValue(
                value=Call(
                    func=Name(id='sin', ctx=Load()),
                    args=[
                        Name(id='a', ctx=Load())]),
                conversion=-1,
                format_spec=JoinedStr(
                    values=[
                        Constant(value='.3')]))]))
```

class ast.TemplateStr(*values*, */*)

Added in version 3.14.

Node representing a template string literal, comprising a series of [`Interpolation`] and [`Constant`] nodes. These nodes may be any order, and do not need to be interleaved.

```
>>> expr = ast.parse('t"{name} finished {place:ordinal}"',
mode='eval')
>>> print(ast.dump(expr, indent=4))
Expression(
    body=TemplateStr(
        values=[
            Interpolation(
                value=Name(id='name', ctx=Load()),
                str='name',
                conversion=-1),
            Constant(value=' finished '),
            Interpolation(
                value=Name(id='place', ctx=Load()),
                str='place',
                conversion=-1,
                format_spec=JoinedStr(
                    values=[
                        Constant(value='ordinal')]))]))
```

class ast.Interpolation(*value*, *str*, *conversion*, *format_spec=None*)

Added in version 3.14.

Node representing a single interpolation field in a template string literal.

- `value` is any expression node (such as a literal, a variable, or a function call). This has the same meaning as `FormattedValue.value`.
- `str` is a constant containing the text of the interpolation expression.

  If `str` is set to `None`, then `value` is used to generate code when calling **`ast.unparse()`**. This no longer guarantees that the generated code is identical to the original and is intended for code generation.
- `conversion` is an integer:
  - -1: no conversion
  - 97 (`ord('a')`): `!a` **`ASCII`** conversion
  - 114 (`ord('r')`): `!r` **`repr()`** conversion
  - 115 (`ord('s')`): `!s` **`string`** conversion

  This has the same meaning as `FormattedValue.conversion`.
- `format_spec` is a **`JoinedStr`** node representing the formatting of the value, or `None` if no format was specified. Both `conversion` and `format_spec` can be set at the same time. This has the same meaning as `FormattedValue.format_spec`.

class ast.List(*elts*, *ctx*)
class ast.Tuple(*elts*, *ctx*)

A list or tuple. `elts` holds a list of nodes representing the elements. `ctx` is **`Store`** if the container is an assignment target (i.e. `(x,y)=something`), and `Load` otherwise.

```
>>> print(ast.dump(ast.parse('[1, 2, 3]', mode='eval'),
indent=4))
Expression(
    body=List(
        elts=[
            Constant(value=1),
            Constant(value=2),
            Constant(value=3)],
        ctx=Load()))
>>> print(ast.dump(ast.parse('(1, 2, 3)', mode='eval'),
indent=4))
Expression(
    body=Tuple(
        elts=[
            Constant(value=1),
            Constant(value=2),
            Constant(value=3)],
        ctx=Load()))
```

class ast.Set(*elts*)

A set. `elts` holds a list of nodes representing the set’s elements.

```
>>> print(ast.dump(ast.parse('{1, 2, 3}', mode='eval'),
indent=4))
Expression(
    body=Set(
        elts=[
            Constant(value=1),
            Constant(value=2),
            Constant(value=3)]))
```

class ast.Dict(*keys*, *values*)

A dictionary. `keys` and `values` hold lists of nodes representing the keys and the values respectively, in matching order (what would be returned when calling `dictionary.keys()` and `dictionary.values()`).

When doing dictionary unpacking using dictionary literals the expression to be expanded goes in the `values` list, with a `None` at the corresponding position in `keys`.

```
>>> print(ast.dump(ast.parse('{"a":1, **d}', mode='eval'),
indent=4))
Expression(
    body=Dict(
        keys=[
            Constant(value='a'),
            None],
        values=[
            Constant(value=1),
            Name(id='d', ctx=Load())]))
```

Variables

class ast.Name(*id*, *ctx*)

A variable name. `id` holds the name as a string, and `ctx` is one of the following types.

class ast.Load
class ast.Store
class ast.Del

Variable references can be used to load the value of a variable, to assign a new value to it, or to delete it. Variable references are given a context to distinguish these cases.

```
>>> print(ast.dump(ast.parse('a'), indent=4))
Module(
    body=[
        Expr(
            value=Name(id='a', ctx=Load()))])

>>> print(ast.dump(ast.parse('a = 1'), indent=4))
Module(
    body=[
        Assign(
            targets=[
                Name(id='a', ctx=Store())],
            value=Constant(value=1))])

>>> print(ast.dump(ast.parse('del a'), indent=4))
Module(
    body=[
        Delete(
            targets=[
                Name(id='a', ctx=Del())])])
```

class ast.Starred(*value*, *ctx*)

A `*var` variable reference. `value` holds the variable, typically a `Name` node. This type must be used when building a `Call` node with `*args`.

```
>>> print(ast.dump(ast.parse('a, *b = it'), indent=4))
Module(
    body=[
        Assign(
            targets=[
                Tuple(
                    elts=[
                        Name(id='a', ctx=Store()),
                        Starred(
                            value=Name(id='b', ctx=Store()),
                            ctx=Store())],
                    ctx=Store())],
            value=Name(id='it', ctx=Load()))])
```

Expressions

class ast.Expr(*value*)

When an expression, such as a function call, appears as a statement by itself with its return value not used or stored, it is wrapped in this container. `value` holds one of the other nodes in this section, a `Constant`, a `Name`, a `Lambda`, a `Yield` or `YieldFrom` node.

```
>>> print(ast.dump(ast.parse('-a'), indent=4))
Module(
    body=[
        Expr(
            value=UnaryOp(
                op=USub(),
                operand=Name(id='a', ctx=Load())))])
```

class ast.UnaryOp(*op*, *operand*)

A unary operation. `op` is the operator, and `operand` any expression node.

class ast.UAdd
class ast.USub
class ast.Not
class ast.Invert

Unary operator tokens. **`Not`** is the `not` keyword, **`Invert`** is the `~` operator.

```
>>> print(ast.dump(ast.parse('not x', mode='eval'),
indent=4))
Expression(
    body=UnaryOp(
        op=Not(),
        operand=Name(id='x', ctx=Load())))
```

class ast.BinOp(*left*, *op*, *right*)

A binary operation (like addition or division). `op` is the operator, and `left` and `right` are any expression nodes.

```
>>> print(ast.dump(ast.parse('x + y', mode='eval'),
indent=4))
Expression(
    body=BinOp(
        left=Name(id='x', ctx=Load()),
        op=Add(),
        right=Name(id='y', ctx=Load())))
```

class ast.Add

class ast.Sub
class ast.Mult
class ast.Div
class ast.FloorDiv
class ast.Mod
class ast.Pow
class ast.LShift
class ast.RShift
class ast.BitOr
class ast.BitXor
class ast.BitAnd
class ast.MatMult

Binary operator tokens.

class ast.BoolOp(*op*, *values*)

A boolean operation, ‘or’ or ‘and’. `op` is **`Or`** or **`And`**. `values` are the values involved. Consecutive operations with the same operator, such as `a or b or c`, are collapsed into one node with several values.

This doesn’t include `not`, which is a **`UnaryOp`**.

```
>>> print(ast.dump(ast.parse('x or y', mode='eval'),
indent=4))
Expression(
    body=BoolOp(
        op=Or(),
        values=[
            Name(id='x', ctx=Load()),
            Name(id='y', ctx=Load())]))
```

class ast.And
class ast.Or

Boolean operator tokens.

class ast.Compare(*left*, *ops*, *comparators*)

A comparison of two or more values. `left` is the first value in the comparison, `ops` the list of operators, and `comparators` the list of values after the first element in the comparison.

```
>>> print(ast.dump(ast.parse('1 <= a < 10', mode='eval'),
indent=4))
Expression(
    body=Compare(
        left=Constant(value=1),
        ops=[
            LtE(),
            Lt()],
        comparators=[
            Name(id='a', ctx=Load()),
            Constant(value=10)]))
```

class ast.Eq
class ast.NotEq
class ast.Lt
class ast.LtE
class ast.Gt
class ast.GtE
class ast.Is
class ast.IsNot
class ast.In
class ast.NotIn

Comparison operator tokens.

class ast.Call(*func*, *args*, *keywords*)

A function call. `func` is the function, which will often be a **`Name`** or **`Attribute`** object. Of the arguments:

- `args` holds a list of the arguments passed by position.
- `keywords` holds a list of **`keyword`** objects representing arguments passed by keyword.

The `args` and `keywords` arguments are optional and default to empty lists.

```
>>> print(ast.dump(ast.parse('func(a, b=c, *d, **e)',
mode='eval'), indent=4))
Expression(
    body=Call(
        func=Name(id='func', ctx=Load()),
        args=[
            Name(id='a', ctx=Load()),
            Starred(
                value=Name(id='d', ctx=Load()),
                ctx=Load())],
        keywords=[
            keyword(
                arg='b',
                value=Name(id='c', ctx=Load())),
            keyword(
                value=Name(id='e', ctx=Load()))]))
```

class ast.keyword(*arg*, *value*)

A keyword argument to a function call or class definition. `arg` is a raw string of the parameter name, `value` is a node to pass in.

class ast.IfExp(*test*, *body*, *orelse*)

An expression such as `a if b else c`. Each field holds a single node, so in the following example, all three are `Name` nodes.

```
>>> print(ast.dump(ast.parse('a if b else c', mode='eval'),
indent=4))
Expression(
    body=IfExp(
        test=Name(id='b', ctx=Load()),
        body=Name(id='a', ctx=Load()),
        orelse=Name(id='c', ctx=Load())))
```

class ast.Attribute(*value*, *attr*, *ctx*)

Attribute access, e.g. `d.keys`. `value` is a node, typically a **`Name`**. `attr` is a bare string giving the name of the attribute, and `ctx` is **`Load`**, **`Store`** or **`Del`** according to how the attribute is acted on.

```
>>> print(ast.dump(ast.parse('snake.colour', mode='eval'),
indent=4))
Expression(
    body=Attribute(
        value=Name(id='snake', ctx=Load()),
        attr='colour',
        ctx=Load()))
```

class ast.NamedExpr(*target*, *value*)

A named expression. This AST node is produced by the assignment expressions operator (also known as the walrus operator). As opposed to the **`Assign`** node in which the first argument can be multiple nodes, in this case both `target` and `value` must be single nodes.

```
>>> print(ast.dump(ast.parse('(x := 4)', mode='eval'),
indent=4))
Expression(
    body=NamedExpr(
        target=Name(id='x', ctx=Store()),
        value=Constant(value=4)))
```

Added in version 3.8.

Subscripting

class ast.Subscript(*value*, *slice*, *ctx*)

A subscript, such as `l[1]`. `value` is the subscripted object (usually sequence or mapping). `slice` is an index, slice or key. It can be a **`Tuple`** and contain a **`Slice`**. `ctx` is **`Load`**, **`Store`** or **`Del`** according to the action performed with the subscript.

```
>>> print(ast.dump(ast.parse('l[1:2, 3]', mode='eval'),
indent=4))
Expression(
    body=Subscript(
        value=Name(id='l', ctx=Load()),
        slice=Tuple(
            elts=[
                Slice(
                    lower=Constant(value=1),
                    upper=Constant(value=2)),
                Constant(value=3)],
            ctx=Load()),
        ctx=Load()))
```

class ast.Slice(*lower*, *upper*, *step*)

Regular slicing (on the form `lower:upper` or `lower:upper:step`). Can occur only inside the *slice* field of [`Subscript`](), either directly or as an element of [`Tuple`]().

```
>>> print(ast.dump(ast.parse('l[1:2]', mode='eval'),
indent=4))
Expression(
    body=Subscript(
        value=Name(id='l', ctx=Load()),
        slice=Slice(
            lower=Constant(value=1),
            upper=Constant(value=2)),
        ctx=Load()))
```

Comprehensions

class ast.ListComp(*elt*, *generators*)
class ast.SetComp(*elt*, *generators*)
class ast.GeneratorExp(*elt*, *generators*)
class ast.DictComp(*key*, *value*, *generators*)

List and set comprehensions, generator expressions, and dictionary comprehensions. `elt` (or `key` and `value`) is a single node representing the part that will be evaluated for each item.

`generators` is a list of **`comprehension`** nodes.

```
>>> print(ast.dump(
...     ast.parse('[x for x in numbers]', mode='eval'),
...     indent=4,
... ))
Expression(
    body=ListComp(
        elt=Name(id='x', ctx=Load()),
        generators=[
            comprehension(
                target=Name(id='x', ctx=Store()),
                iter=Name(id='numbers', ctx=Load()),
                is_async=0)]))
>>> print(ast.dump(
...     ast.parse('{x: x**2 for x in numbers}',
mode='eval'),
...     indent=4,
... ))
Expression(
    body=DictComp(
        key=Name(id='x', ctx=Load()),
        value=BinOp(
            left=Name(id='x', ctx=Load()),
            op=Pow(),
            right=Constant(value=2)),
        generators=[
            comprehension(
                target=Name(id='x', ctx=Store()),
                iter=Name(id='numbers', ctx=Load()),
                is_async=0)]))
>>> print(ast.dump(
...     ast.parse('{x for x in numbers}', mode='eval'),
...     indent=4,
... ))
Expression(
    body=SetComp(
        elt=Name(id='x', ctx=Load()),
        generators=[
                comprehension(
                    target=Name(id='x', ctx=Store()),
                    iter=Name(id='numbers', ctx=Load()),
                    is_async=0)]))
```

class ast.comprehension(*target*, *iter*, *ifs*, *is_async*)

One `for` clause in a comprehension. `target` is the reference to use for each element - typically a **`Name`** or **`Tuple`** node. `iter` is the object to iterate over. `ifs` is a list of test expressions: each `for` clause can have multiple `ifs`.

`is_async` indicates a comprehension is asynchronous (using an `async for` instead of `for`). The value is an integer (0 or 1).

```
>>> print(ast.dump(ast.parse('[ord(c) for line in file for c
in line]', mode='eval'),
...                 indent=4)) # Multiple comprehensions in
one.
Expression(
    body=ListComp(
        elt=Call(
            func=Name(id='ord', ctx=Load()),
            args=[
                Name(id='c', ctx=Load())]),
        generators=[
            comprehension(
                target=Name(id='line', ctx=Store()),
                iter=Name(id='file', ctx=Load()),
                is_async=0),
            comprehension(
                target=Name(id='c', ctx=Store()),
                iter=Name(id='line', ctx=Load()),
                is_async=0)]))

>>> print(ast.dump(ast.parse('(n**2 for n in it if n>5 if
n<10)', mode='eval'),
...                 indent=4)) # generator comprehension
Expression(
    body=GeneratorExp(
        elt=BinOp(
                left=Name(id='n', ctx=Load()),
                op=Pow(),
                right=Constant(value=2)),
            generators=[
                comprehension(
                    target=Name(id='n', ctx=Store()),
                    iter=Name(id='it', ctx=Load()),
                    ifs=[
                        Compare(
                            left=Name(id='n', ctx=Load()),
                            ops=[
                                Gt()],
                            comparators=[
                                Constant(value=5)]),
                        Compare(
                            left=Name(id='n', ctx=Load()),
                            ops=[
                                Lt()],
                            comparators=[
                                Constant(value=10)])],
                    is_async=0)]))

>>> print(ast.dump(ast.parse('[i async for i in soc]',
mode='eval'),
...                 indent=4)) # Async comprehension
Expression(
    body=ListComp(
        elt=Name(id='i', ctx=Load()),
        generators=[
            comprehension(
                target=Name(id='i', ctx=Store()),
                iter=Name(id='soc', ctx=Load()),
                is_async=1)]))
```

Statements

class ast.Assign(*targets*, *value*, *type_comment*)

An assignment. `targets` is a list of nodes, and `value` is a single node.

Multiple nodes in `targets` represents assigning the same value to each. Unpacking is represented by putting a **`Tuple`** or **`List`** within `targets`.

type_comment

`type_comment` is an optional string with the type annotation as a comment.

```
>>> print(ast.dump(ast.parse('a = b = 1'), indent=4)) #
Multiple assignment
Module(
    body=[
        Assign(
            targets=[
                Name(id='a', ctx=Store()),
                Name(id='b', ctx=Store())],
            value=Constant(value=1))])

>>> print(ast.dump(ast.parse('a,b = c'), indent=4)) #
Unpacking
Module(
    body=[
        Assign(
            targets=[
                Tuple(
                    elts=[
                        Name(id='a', ctx=Store()),
                        Name(id='b', ctx=Store())],
                    ctx=Store())],
            value=Name(id='c', ctx=Load()))])
```

class ast.AnnAssign(*target*, *annotation*, *value*, *simple*)

An assignment with a type annotation. `target` is a single node and can be a **`Name`**, an **`Attribute`** or a **`Subscript`**. `annotation` is the annotation, such as a **`Constant`** or **`Name`** node. `value` is a single optional node.

`simple` is always either 0 (indicating a “complex” target) or 1 (indicating a “simple” target). A “simple” target consists solely of a `Name` node that does not appear between parentheses; all other targets are considered complex. Only simple targets appear in the `__annotations__` dictionary of modules and classes.

```
>>> print(ast.dump(ast.parse('c: int'), indent=4))
Module(
    body=[
        AnnAssign(
            target=Name(id='c', ctx=Store()),
            annotation=Name(id='int', ctx=Load()),
            simple=1)])

>>> print(ast.dump(ast.parse('(a): int = 1'), indent=4)) #
Annotation with parenthesis
Module(
    body=[
        AnnAssign(
            target=Name(id='a', ctx=Store()),
            annotation=Name(id='int', ctx=Load()),
            value=Constant(value=1),
            simple=0)])

>>> print(ast.dump(ast.parse('a.b: int'), indent=4)) #
Attribute annotation
Module(
    body=[
        AnnAssign(
            target=Attribute(
                value=Name(id='a', ctx=Load()),
                attr='b',
                ctx=Store()),
            annotation=Name(id='int', ctx=Load()),
            simple=0)])

>>> print(ast.dump(ast.parse('a[1]: int'), indent=4)) #
Subscript annotation
Module(
    body=[
        AnnAssign(
            target=Subscript(
                value=Name(id='a', ctx=Load()),
                slice=Constant(value=1),
                ctx=Store()),
            annotation=Name(id='int', ctx=Load()),
            simple=0)])
```

class ast.AugAssign(*target*, *op*, *value*)

Augmented assignment, such as `a += 1`. In the following example, `target` is a **`Name`** node for `x` (with the **`Store`** context), `op` is **`Add`**, and `value` is a **`Constant`** with value for 1.

The `target` attribute cannot be of class **`Tuple`** or **`List`**, unlike the targets of **`Assign`**.

```
>>> print(ast.dump(ast.parse('x += 2'), indent=4))
Module(
    body=[
        AugAssign(
            target=Name(id='x', ctx=Store()),
            op=Add(),
            value=Constant(value=2))])
```

class ast.Raise(*exc*, *cause*)

A `raise` statement. `exc` is the exception object to be raised, normally a **`Call`** or **`Name`**, or `None` for a standalone `raise`. `cause` is the optional part for `y` in `raise x from y`.

```
>>> print(ast.dump(ast.parse('raise x from y'), indent=4))
Module(
    body=[
        Raise(
            exc=Name(id='x', ctx=Load()),
            cause=Name(id='y', ctx=Load()))])
```

class ast.Assert(*test*, *msg*)

An assertion. `test` holds the condition, such as a **`Compare`** node. `msg` holds the failure message.

```
>>> print(ast.dump(ast.parse('assert x,y'), indent=4))
Module(
    body=[
        Assert(
            test=Name(id='x', ctx=Load()),
            msg=Name(id='y', ctx=Load()))])
```

class ast.Delete(*targets*)

Represents a `del` statement. `targets` is a list of nodes, such as **`Name`**, **`Attribute`** or **`Subscript`** nodes.

```
>>> print(ast.dump(ast.parse('del x,y,z'), indent=4))
Module(
    body=[
        Delete(
            targets=[
                Name(id='x', ctx=Del()),
                Name(id='y', ctx=Del()),
                Name(id='z', ctx=Del())])])
```

class ast.Pass

A `pass` statement.

```
>>> print(ast.dump(ast.parse('pass'), indent=4))
Module(
    body=[
        Pass()])
```

class ast.TypeAlias(*name*, *type_params*, *value*)

A type alias created through the **`type`** statement. `name` is the name of the alias, `type_params` is a list of type parameters, and `value` is the value of the type alias.

```
>>> print(ast.dump(ast.parse('type Alias = int'), indent=4))
Module(
    body=[
        TypeAlias(
            name=Name(id='Alias', ctx=Store()),
            value=Name(id='int', ctx=Load()))])
```

Added in version 3.12.

Other statements which are only applicable inside functions or loops are described in other sections.

Imports

class ast.Import(*names*)

An import statement. `names` is a list of `alias` nodes.

```
>>> print(ast.dump(ast.parse('import x,y,z'), indent=4))
Module(
    body=[
        Import(
            names=[
                alias(name='x'),
                alias(name='y'),
                alias(name='z')])])
```

class ast.ImportFrom(*module*, *names*, *level*)

Represents `from x import y`. `module` is a raw string of the ‘from’ name, without any leading dots, or `None` for statements such as `from . import foo`. `level` is an integer holding the level of the relative import (0 means absolute import).

```
>>> print(ast.dump(ast.parse('from y import x,y,z'),
indent=4))
Module(
    body=[
        ImportFrom(
            module='y',
            names=[
                alias(name='x'),
                alias(name='y'),
                alias(name='z')],
            level=0)])
```

class ast.alias(*name*, *asname*)

Both parameters are raw strings of the names. `asname` can be `None` if the regular name is to be used.

```
>>> print(ast.dump(ast.parse('from ..foo.bar import a as b,
c'), indent=4))
Module(
    body=[
        ImportFrom(
            module='foo.bar',
            names=[
                alias(name='a', asname='b'),
                alias(name='c')],
            level=2)])
```

Control flow

Note

Optional clauses such as `else` are stored as an empty list if they’re not present.

class ast.If(*test*, *body*, *orelse*)

An `if` statement. `test` holds a single node, such as a [`Compare`] node. `body` and `orelse` each hold a list of nodes.

`elif` clauses don’t have a special representation in the AST, but rather appear as extra [`If`] nodes within the `orelse` section of the previous one.

```
>>> print(ast.dump(ast.parse("""
... if x:
...    ...
... elif y:
...    ...
... else:
...    ...
... """), indent=4))
Module(
    body=[
        If(
            test=Name(id='x', ctx=Load()),
            body=[
                Expr(
                    value=Constant(value=Ellipsis))],
            orelse=[
                If(
                    test=Name(id='y', ctx=Load()),
                    body=[
                        Expr(

value=Constant(value=Ellipsis))],
                    orelse=[
                        Expr(

value=Constant(value=Ellipsis))])])])
```

class ast.For(*target*, *iter*, *body*, *orelse*, *type_comment*)

A `for` loop. `target` holds the variable(s) the loop assigns to, as a single `Name`, `Tuple`, `List`, `Attribute` or `Subscript` node. `iter` holds the item to be looped over, again as a single node. `body` and `orelse` contain lists of nodes to execute. Those in `orelse` are executed if the loop finishes normally, rather than via a `break` statement.

type_comment

`type_comment` is an optional string with the type annotation as a comment.

```
>>> print(ast.dump(ast.parse("""
... for x in y:
...     ...
... else:
...     ...
... """), indent=4))
Module(
    body=[
        For(
            target=Name(id='x', ctx=Store()),
            iter=Name(id='y', ctx=Load()),
            body=[
                Expr(
                    value=Constant(value=Ellipsis))],
            orelse=[
                Expr(
                    value=Constant(value=Ellipsis))])])
```

class ast.While(*test*, *body*, *orelse*)

A `while` loop. `test` holds the condition, such as a **`Compare`** node.

```
>>> print(ast.dump(ast.parse("""
... while x:
...    ...
... else:
...    ...
... """), indent=4))
Module(
    body=[
        While(
            test=Name(id='x', ctx=Load()),
            body=[
                Expr(
                    value=Constant(value=Ellipsis))],
            orelse=[
                Expr(
                    value=Constant(value=Ellipsis))])])
```

class ast.Break
class ast.Continue

The `break` and `continue` statements.

```
>>> print(ast.dump(ast.parse("""\
... for a in b:
...     if a > 5:
...         break
...     else:
...         continue
...
... """), indent=4))
```

```
Module(
    body=[
        For(
            target=Name(id='a', ctx=Store()),
            iter=Name(id='b', ctx=Load()),
            body=[
                If(
                    test=Compare(
                        left=Name(id='a', ctx=Load()),
                        ops=[
                            Gt()],
                        comparators=[
                            Constant(value=5)]),
                    body=[
                        Break()],
                    orelse=[
                        Continue()])])])
```

class ast.Try(*body*, *handlers*, *orelse*, *finalbody*)

`try` blocks. All attributes are list of nodes to execute, except for `handlers`, which is a list of **`ExceptHandler`** nodes.

```
>>> print(ast.dump(ast.parse("""
... try:
...    ...
... except Exception:
...    ...
... except OtherException as e:
...    ...
... else:
...    ...
... finally:
...    ...
... """), indent=4))
Module(
    body=[
        Try(
            body=[
                Expr(
                    value=Constant(value=Ellipsis))],
            handlers=[
                ExceptHandler(
                    type=Name(id='Exception', ctx=Load()),
                    body=[
                        Expr(

value=Constant(value=Ellipsis))]),
                ExceptHandler(
                    type=Name(id='OtherException',
ctx=Load()),
                    name='e',
                    body=[
                        Expr(

value=Constant(value=Ellipsis))])],
            orelse=[
                Expr(
                    value=Constant(value=Ellipsis))],
            finalbody=[
                Expr(
                    value=Constant(value=Ellipsis))])])
```

class ast.TryStar(*body*, *handlers*, *orelse*, *finalbody*)

`try` blocks which are followed by `except*` clauses. The attributes are the same as for **`Try`** but the **`ExceptHandler`** nodes in `handlers` are interpreted as `except*` blocks rather then `except`.

```
>>> print(ast.dump(ast.parse("""
... try:
...    ...
... except* Exception:
...    ...
... """), indent=4))
Module(
    body=[
        TryStar(
            body=[
                Expr(
                    value=Constant(value=Ellipsis))],
            handlers=[
                ExceptHandler(
                    type=Name(id='Exception', ctx=Load()),
                    body=[
                        Expr(

value=Constant(value=Ellipsis))])])])
```

Added in version 3.11.

class ast.ExceptHandler(*type*, *name*, *body*)

A single `except` clause. `type` is the exception type it will match, typically a [`Name`]() node (or `None` for a catch-all `except:` clause). `name` is a raw string for the name to hold the exception, or `None` if the clause doesn't have `as foo`. `body` is a list of nodes.

```
>>> print(ast.dump(ast.parse("""\
... try:
...     a + 1
... except TypeError:
...     pass
... """), indent=4))
Module(
    body=[
        Try(
            body=[
                Expr(
                    value=BinOp(
                        left=Name(id='a', ctx=Load()),
                        op=Add(),
                        right=Constant(value=1)))],
            handlers=[
                ExceptHandler(
                    type=Name(id='TypeError', ctx=Load()),
                    body=[
                        Pass()])])])
```

class ast.With(*items*, *body*, *type_comment*)

A `with` block. `items` is a list of [`withitem`]() nodes representing the context managers, and `body` is the indented block inside the context.

type_comment

`type_comment` is an optional string with the type annotation as a comment.

class ast.withitem(*context_expr*, *optional_vars*)

A single context manager in a `with` block. `context_expr` is the context manager, often a **`Call`** node. `optional_vars` is a **`Name`**, **`Tuple`** or **`List`** for the `as foo` part, or `None` if that isn't used.

```
>>> print(ast.dump(ast.parse("""\
... with a as b, c as d:
...    something(b, d)
... """), indent=4))
Module(
    body=[
        With(
            items=[
                withitem(
                    context_expr=Name(id='a', ctx=Load()),
                    optional_vars=Name(id='b',
ctx=Store())),
                withitem(
                    context_expr=Name(id='c', ctx=Load()),
                    optional_vars=Name(id='d',
ctx=Store()))],
            body=[
                Expr(
                    value=Call(
                        func=Name(id='something',
ctx=Load()),
                        args=[
                            Name(id='b', ctx=Load()),
                            Name(id='d', ctx=Load())]))])])
```

Pattern matching

class ast.Match(*subject*, *cases*)

A `match` statement. `subject` holds the subject of the match (the object that is being matched against the cases) and `cases` contains an iterable of [`match_case`] nodes with the different cases.

Added in version 3.10.

class ast.match_case(*pattern*, *guard*, *body*)

A single case pattern in a `match` statement. `pattern` contains the match pattern that the subject will be matched against. Note that the [`AST`] nodes produced for patterns differ from those produced for expressions, even when they share the same syntax.

The `guard` attribute contains an expression that will be evaluated if the pattern matches the subject.

`body` contains a list of nodes to execute if the pattern matches and the result of evaluating the guard expression is true.

```
>>> print(ast.dump(ast.parse("""
... match x:
...     case [x] if x>0:
...         ...
...     case tuple():
...         ...
... """), indent=4))
Module(
    body=[
        Match(
            subject=Name(id='x', ctx=Load()),
            cases=[
                match_case(
                    pattern=MatchSequence(
                        patterns=[
                            MatchAs(name='x')]),
                    guard=Compare(
                        left=Name(id='x', ctx=Load()),
                        ops=[
                            Gt()],
                        comparators=[
                                Constant(value=0)]),
                        body=[
                            Expr(

value=Constant(value=Ellipsis))]),
                    match_case(
                        pattern=MatchClass(
                            cls=Name(id='tuple', ctx=Load())),
                        body=[
                            Expr(

value=Constant(value=Ellipsis))])])])
```

Added in version 3.10.

class ast.MatchValue(*value*)

A match literal or value pattern that compares by equality. `value` is an expression node. Permitted value nodes are restricted as described in the match statement documentation. This pattern succeeds if the match subject is equal to the evaluated value.

```
>>> print(ast.dump(ast.parse("""
... match x:
...     case "Relevant":
...         ...
... """), indent=4))
Module(
    body=[
        Match(
            subject=Name(id='x', ctx=Load()),
            cases=[
                match_case(
                    pattern=MatchValue(
                        value=Constant(value='Relevant')),
                    body=[
                        Expr(

value=Constant(value=Ellipsis))])])])
```

Added in version 3.10.

class ast.MatchSingleton(*value*)

A match literal pattern that compares by identity. `value` is the singleton to be compared against: `None`, `True`, or `False`. This pattern succeeds if the match subject is the given constant.

```
>>> print(ast.dump(ast.parse("""
... match x:
...     case None:
...         ...
... """), indent=4))
Module(
    body=[
        Match(
            subject=Name(id='x', ctx=Load()),
            cases=[
                match_case(
                    pattern=MatchSingleton(value=None),
                    body=[
                        Expr(

value=Constant(value=Ellipsis))])])])
```

Added in version 3.10.

class ast.MatchSequence(*patterns*)

A match sequence pattern. `patterns` contains the patterns to be matched against the subject elements if the subject is a sequence. Matches a variable length sequence if one of the subpatterns is a `MatchStar` node, otherwise matches a fixed length sequence.

```
>>> print(ast.dump(ast.parse("""
... match x:
...     case [1, 2]:
...         ...
... """), indent=4))
Module(
    body=[
        Match(
            subject=Name(id='x', ctx=Load()),
            cases=[
                match_case(
                    pattern=MatchSequence(
                        patterns=[
                            MatchValue(
                                value=Constant(value=1)),
                            MatchValue(
                                value=Constant(value=2))]),
                    body=[
                        Expr(

value=Constant(value=Ellipsis))])])])
```

Added in version 3.10.

class ast.MatchStar(*name*)

Matches the rest of the sequence in a variable length match sequence pattern. If `name` is not `None`, a list containing the remaining sequence elements is bound to that name if the overall sequence pattern is successful.

```
>>> print(ast.dump(ast.parse("""
... match x:
...     case [1, 2, *rest]:
...         ...
...     case [*_]:
...         ...
... """), indent=4))
Module(
    body=[
        Match(
            subject=Name(id='x', ctx=Load()),
            cases=[
                match_case(
                    pattern=MatchSequence(
                        patterns=[
                            MatchValue(
                                value=Constant(value=1)),
                            MatchValue(
                                value=Constant(value=2)),
                            MatchStar(name='rest')]),
                    body=[
                        Expr(

value=Constant(value=Ellipsis))]),
                match_case(
                    pattern=MatchSequence(
                        patterns=[
                            MatchStar()]),
                    body=[
                        Expr(

value=Constant(value=Ellipsis))])])])
```

Added in version 3.10.

class ast.MatchMapping(*keys*, *patterns*, *rest*)

A match mapping pattern. `keys` is a sequence of expression nodes. `patterns` is a corresponding sequence of pattern nodes. `rest` is an optional name that can be specified to capture the remaining mapping elements. Permitted key expressions are restricted as described in the match statement documentation.

This pattern succeeds if the subject is a mapping, all evaluated key expressions are present in the mapping, and the value corresponding to each key matches the corresponding subpattern. If `rest` is not `None`, a dict containing the remaining mapping elements is bound to that name if the overall mapping pattern is successful.

```
>>> print(ast.dump(ast.parse("""
... match x:
...     case {1: _, 2: _}:
...         ...
...     case {**rest}:
...         ...
... """), indent=4))
Module(
    body=[
        Match(
            subject=Name(id='x', ctx=Load()),
            cases=[
                match_case(
                    pattern=MatchMapping(
                        keys=[
                            Constant(value=1),
                            Constant(value=2)],
                        patterns=[
                            MatchAs(),
                            MatchAs()]),
                    body=[
                        Expr(

value=Constant(value=Ellipsis))]),
                match_case(
                    pattern=MatchMapping(rest='rest'),
                    body=[
                        Expr(

value=Constant(value=Ellipsis))])])])
```

Added in version 3.10.

class ast.MatchClass(*cls*, *patterns*, *kwd_attrs*, *kwd_patterns*)

A match class pattern. `cls` is an expression giving the nominal class to be matched. `patterns` is a sequence of pattern nodes to be matched against the class defined sequence of pattern matching attributes. `kwd_attrs` is a sequence of additional attributes to be matched (specified as keyword arguments in the class pattern), `kwd_patterns` are the corresponding patterns (specified as keyword values in the class pattern).

This pattern succeeds if the subject is an instance of the nominated class, all positional patterns match the corresponding class-defined attributes, and any specified keyword attributes match their corresponding pattern.

Note: classes may define a property that returns self in order to match a pattern node against the instance being matched. Several builtin types are also matched that way, as described in the match statement documentation.

```
>>> print(ast.dump(ast.parse("""
... match x:
...     case Point2D(0, 0):
...         ...
...     case Point3D(x=0, y=0, z=0):
...         ...
... """), indent=4))
Module(
    body=[
        Match(
            subject=Name(id='x', ctx=Load()),
            cases=[
                match_case(
                    pattern=MatchClass(
                        cls=Name(id='Point2D', ctx=Load()),
                        patterns=[
                            MatchValue(
                                value=Constant(value=0)),
                            MatchValue(
                                value=Constant(value=0))]),
                    body=[
                        Expr(

value=Constant(value=Ellipsis))]),
                match_case(
                    pattern=MatchClass(
                        cls=Name(id='Point3D', ctx=Load()),
                        kwd_attrs=[
                            'x',
                            'y',
                            'z'],
                        kwd_patterns=[
                            MatchValue(
                                value=Constant(value=0)),
                            MatchValue(
                                value=Constant(value=0)),
                            MatchValue(
                                value=Constant(value=0))]),
                    body=[
                        Expr(

value=Constant(value=Ellipsis))])])])
```

Added in version 3.10.

class ast.MatchAs(*pattern*, *name*)

A match “as-pattern”, capture pattern or wildcard pattern. `pattern` contains the match pattern that the subject will be matched against. If the pattern is `None`, the node represents a capture pattern (i.e a bare name) and will always succeed.

The `name` attribute contains the name that will be bound if the pattern is successful. If `name` is `None`, `pattern` must also be `None` and the node represents the wildcard pattern.

```
>>> print(ast.dump(ast.parse("""
... match x:
...     case [x] as y:
...         ...
...     case _:
...         ...
... """), indent=4))
Module(
    body=[
        Match(
            subject=Name(id='x', ctx=Load()),
            cases=[
                match_case(
                    pattern=MatchAs(
                        pattern=MatchSequence(
                            patterns=[
                                MatchAs(name='x')]),
                        name='y'),
                    body=[
                        Expr(

value=Constant(value=Ellipsis))]),
                match_case(
                    pattern=MatchAs(),
                    body=[
                        Expr(

value=Constant(value=Ellipsis))])])])
```

Added in version 3.10.

class ast.MatchOr(*patterns*)

A match “or-pattern”. An or-pattern matches each of its subpatterns in turn to the subject, until one succeeds. The or-pattern is then deemed to succeed. If none of the subpatterns succeed the or-pattern fails. The `patterns` attribute contains a list of match pattern nodes that will be matched against the subject.

```
>>> print(ast.dump(ast.parse("""
... match x:
...     case [x] | (y):
...         ...
... """), indent=4))
Module(
    body=[
        Match(
            subject=Name(id='x', ctx=Load()),
            cases=[
                match_case(
                    pattern=MatchOr(
                        patterns=[
                            MatchSequence(
                                patterns=[
                                    MatchAs(name='x')]),
                            MatchAs(name='y')]),
                    body=[
                        Expr(

value=Constant(value=Ellipsis))])])])
```

Added in version 3.10.

Type annotations

class ast.TypeIgnore(*lineno*, *tag*)

A `# type: ignore` comment located at *lineno*. *tag* is the optional tag specified by the form `# type: ignore <tag>`.

```
>>> print(ast.dump(ast.parse('x = 1 # type: ignore',
type_comments=True), indent=4))
Module(
    body=[
        Assign(
            targets=[
                Name(id='x', ctx=Store())],
            value=Constant(value=1))],
    type_ignores=[
        TypeIgnore(lineno=1, tag='')])
>>> print(ast.dump(ast.parse('x: bool = 1 # type:
ignore[assignment]', type_comments=True), indent=4))
Module(
    body=[
        AnnAssign(
            target=Name(id='x', ctx=Store()),
            annotation=Name(id='bool', ctx=Load()),
            value=Constant(value=1),
            simple=1)],
    type_ignores=[
        TypeIgnore(lineno=1, tag='[assignment]')])
```

Note

`TypeIgnore` nodes are not generated when the *type_comments* parameter is set to `False` (default). See **`ast.parse()`** for more details.

Added in version 3.8.

Type parameters

Type parameters can exist on classes, functions, and type aliases.

class ast.TypeVar(*name*, *bound*, *default_value*)

A [`typing.TypeVar`](). `name` is the name of the type variable. `bound` is the bound or constraints, if any. If `bound` is a [`Tuple`](), it represents constraints; otherwise it represents the bound. `default_value` is the default value; if the **`TypeVar`** has no default, this attribute will be set to `None`.

```
>>> print(ast.dump(ast.parse("type Alias[T: int = bool] =
list[T]"), indent=4))
Module(
    body=[
        TypeAlias(
            name=Name(id='Alias', ctx=Store()),
            type_params=[
                TypeVar(
                    name='T',
                    bound=Name(id='int', ctx=Load()),
                    default_value=Name(id='bool',
ctx=Load()))],
            value=Subscript(
                value=Name(id='list', ctx=Load()),
                slice=Name(id='T', ctx=Load()),
                ctx=Load()))])
```

Added in version 3.12.

Changed in version 3.13: Added the *default_value* parameter.

class ast.ParamSpec(*name*, *default_value*)

A [`typing.ParamSpec`](). `name` is the name of the parameter specification. `default_value` is the default value; if the **`ParamSpec`** has no default, this attribute will be set to `None`.

```
>>> print(ast.dump(ast.parse("type Alias[**P = [int, str]] =
Callable[P, int]"), indent=4))
Module(
    body=[
        TypeAlias(
            name=Name(id='Alias', ctx=Store()),
            type_params=[
                ParamSpec(
                    name='P',
                    default_value=List(
                        elts=[
                            Name(id='int', ctx=Load()),
                            Name(id='str', ctx=Load())],
                        ctx=Load()))],
            value=Subscript(
                value=Name(id='Callable', ctx=Load()),
                slice=Tuple(
                    elts=[
                        Name(id='P', ctx=Load()),
                        Name(id='int', ctx=Load())],
                    ctx=Load()),
                ctx=Load()))])
```

Added in version 3.12.

Changed in version 3.13: Added the *default_value* parameter.

class ast.TypeVarTuple(*name*, *default_value*)

A **typing.TypeVarTuple**. `name` is the name of the type variable tuple. `default_value` is the default value; if the **`TypeVarTuple`** has no default, this attribute will be set to `None`.

```
>>> print(ast.dump(ast.parse("type Alias[*Ts = ()] =
tuple[*Ts]"), indent=4))
Module(
    body=[
        TypeAlias(
            name=Name(id='Alias', ctx=Store()),
            type_params=[
                TypeVarTuple(
                    name='Ts',
                    default_value=Tuple(ctx=Load()))],
            value=Subscript(
                value=Name(id='tuple', ctx=Load()),
                slice=Tuple(
                    elts=[
                        Starred(
                            value=Name(id='Ts', ctx=Load()),
                            ctx=Load())],
                    ctx=Load()),
                ctx=Load()))])
```

Added in version 3.12.

Changed in version 3.13: Added the *default_value* parameter.

Function and class definitions

class ast.FunctionDef(*name*, *args*, *body*, *decorator_list*, *returns*, *type_comment*, *type_params*)

A function definition.

- `name` is a raw string of the function name.
- `args` is an **`arguments`** node.
- `body` is the list of nodes inside the function.
- `decorator_list` is the list of decorators to be applied, stored outermost first (i.e. the first in the list will be applied last).
- `returns` is the return annotation.
- `type_params` is a list of type parameters.

type_comment

`type_comment` is an optional string with the type annotation as a comment.

Changed in version 3.12: Added `type_params`.

class ast.Lambda(*args*, *body*)

`lambda` is a minimal function definition that can be used inside an expression. Unlike **`FunctionDef`**, `body` holds a single node.

```
>>> print(ast.dump(ast.parse('lambda x,y: ...'), indent=4))
Module(
    body=[
        Expr(
            value=Lambda(
                args=arguments(
                    args=[
                        arg(arg='x'),
                        arg(arg='y')]),
                body=Constant(value=Ellipsis)))])
```

class ast.arguments(*posonlyargs*, *args*, *vararg*, *kwonlyargs*, *kw_defaults*, *kwarg*, *defaults*)

The arguments for a function.

- `posonlyargs`, `args` and `kwonlyargs` are lists of **`arg`** nodes.
- `vararg` and `kwarg` are single **`arg`** nodes, referring to the `*args, **kwargs` parameters.
- `kw_defaults` is a list of default values for keyword-only arguments. If one is `None`, the corresponding argument is required.
- `defaults` is a list of default values for arguments that can be passed positionally. If there are fewer defaults, they correspond to the last n arguments.

class ast.arg(*arg*, *annotation*, *type_comment*)

A single argument in a list. `arg` is a raw string of the argument name; `annotation` is its annotation, such as a **`Name`** node.

type_comment

`type_comment` is an optional string with the type annotation as a comment

```
>>> print(ast.dump(ast.parse("""\
... @decorator1
... @decorator2
... def f(a: 'annotation', b=1, c=2, *d, e, f=3, **g) ->
'return annotation':
...     pass
... """), indent=4))
Module(
    body=[
        FunctionDef(
            name='f',
            args=arguments(
                args=[
                    arg(
                        arg='a',

annotation=Constant(value='annotation')),
                    arg(arg='b'),
                    arg(arg='c')],
                vararg=arg(arg='d'),
                kwonlyargs=[
                    arg(arg='e'),
                    arg(arg='f')],
                kw_defaults=[
                    None,
                    Constant(value=3)],
                kwarg=arg(arg='g'),
                defaults=[
                    Constant(value=1),
                    Constant(value=2)]),
            body=[
                Pass()],
            decorator_list=[
                Name(id='decorator1', ctx=Load()),
                Name(id='decorator2', ctx=Load())],
            returns=Constant(value='return annotation'))])
```

class ast.Return(*value*)

A `return` statement.

```
>>> print(ast.dump(ast.parse('return 4'), indent=4))
Module(
    body=[
        Return(
            value=Constant(value=4))])
```

class ast.Yield(*value*)
class ast.YieldFrom(*value*)

A `yield` or `yield from` expression. Because these are expressions, they must be wrapped in an `Expr` node if the value sent back is not used.

```
>>> print(ast.dump(ast.parse('yield x'), indent=4))
Module(
    body=[
        Expr(
            value=Yield(
                value=Name(id='x', ctx=Load())))])

>>> print(ast.dump(ast.parse('yield from x'), indent=4))
Module(
    body=[
        Expr(
            value=YieldFrom(
                value=Name(id='x', ctx=Load())))])
```

class ast.Global(*names*)
class ast.Nonlocal(*names*)

`global` and `nonlocal` statements. `names` is a list of raw strings.

```
>>> print(ast.dump(ast.parse('global x,y,z'), indent=4))
Module(
    body=[
        Global(
            names=[
                'x',
                'y',
                'z'])])

>>> print(ast.dump(ast.parse('nonlocal x,y,z'), indent=4))
Module(
    body=[
        Nonlocal(
            names=[
                'x',
                    'y',
                    'z'])])
```

class ast.ClassDef(*name*, *bases*, *keywords*, *body*, *decorator_list*, *type_params*)

A class definition.

- `name` is a raw string for the class name
- `bases` is a list of nodes for explicitly specified base classes.
- `keywords` is a list of **`keyword`** nodes, principally for ‘metaclass’. Other keywords will be passed to the metaclass, as per **PEP 3115** [https://peps.python.org/pep-3115/].
- `body` is a list of nodes representing the code within the class definition.
- `decorator_list` is a list of nodes, as in **`FunctionDef`**.
- `type_params` is a list of type parameters.

```
>>> print(ast.dump(ast.parse("""\
... @decorator1
... @decorator2
... class Foo(base1, base2, metaclass=meta):
...     pass
... """), indent=4))
Module(
    body=[
        ClassDef(
            name='Foo',
            bases=[
                Name(id='base1', ctx=Load()),
                Name(id='base2', ctx=Load())],
            keywords=[
                keyword(
                    arg='metaclass',
                    value=Name(id='meta', ctx=Load()))],
            body=[
                Pass()],
            decorator_list=[
                   Name(id='decorator1', ctx=Load()),
                   Name(id='decorator2', ctx=Load())])])
```

Changed in version 3.12: Added `type_params`.

Async and await

class ast.AsyncFunctionDef(*name*, *args*, *body*, *decorator_list*, *returns*, *type_comment*, *type_params*)

An `async def` function definition. Has the same fields as **`FunctionDef`**.

Changed in version 3.12: Added `type_params`.

class ast.Await(*value*)

An `await` expression. `value` is what it waits for. Only valid in the body of an **`AsyncFunctionDef`**.

```
>>> print(ast.dump(ast.parse("""\
... async def f():
...     await other_func()
... """), indent=4))
Module(
    body=[
        AsyncFunctionDef(
            name='f',
            args=arguments(),
            body=[
                Expr(
                    value=Await(
                        value=Call(
                            func=Name(id='other_func',
ctx=Load())))))])])
```

class ast.AsyncFor(*target*, *iter*, *body*, *orelse*, *type_comment*)
class ast.AsyncWith(*items*, *body*, *type_comment*)

`async for` loops and `async with` context managers. They have the same fields as **`For`** and **`With`**, respectively. Only valid in the body of an **`AsyncFunctionDef`**.

Note

When a string is parsed by `ast.parse()`, operator nodes (subclasses of **`ast.operator`**, **`ast.unaryop`**, **`ast.cmpop`**, **`ast.boolop`** and **`ast.expr_context`**) on the returned tree will be singletons. Changes to one will be reflected in all other occurrences of the same value (for example, `ast.Add`).

`ast` helpers

Apart from the node classes, the `ast` module defines these utility functions and classes for traversing abstract syntax trees:

ast.parse(*source*, *filename='<unknown>'*, *mode='exec'*, *, *type_comments=False*, *feature_version=None*, *optimize=-1*)

Parse the source into an AST node. Equivalent to `compile(source, filename, mode, flags=FLAGS_VALUE, optimize=optimize)`, where `FLAGS_VALUE` is `ast.PyCF_ONLY_AST` if `optimize <= 0` and `ast.PyCF_OPTIMIZED_AST` otherwise.

If `type_comments=True` is given, the parser is modified to check and return type comments as specified by **PEP 484** [https://peps.python.org/pep-0484/] and **PEP 526** [https://peps.python.org/pep-0526/]. This is equivalent to adding `ast.PyCF_TYPE_COMMENTS` to the flags passed to `compile()`. This will report syntax errors for misplaced type comments. Without this flag, type comments will be ignored, and the `type_comment` field on selected AST nodes will always be `None`. In addition, the locations of `# type: ignore` comments will be returned as the `type_ignores` attribute of `Module` (otherwise it is always an empty list).

In addition, if `mode` is `'func_type'`, the input syntax is modified to correspond to **PEP 484** [https://peps.python.org/pep-0484/] “signature type comments”, e.g. `(str, int) -> List[str]`.

Setting `feature_version` to a tuple `(major, minor)` will result in a “best-effort” attempt to parse using that Python version’s grammar. For example, setting `feature_version=(3, 9)` will attempt to disallow parsing of **`match`** statements. Currently `major` must equal to `3`. The lowest supported version is `(3, 7)` (and this may increase in future Python versions); the highest is `sys.version_info[0:2]`. “Best-effort” attempt means there is no guarantee that the parse (or success of the parse) is the same as when run on the Python version corresponding to `feature_version`.

If source contains a null character (`\0`), **`ValueError`** is raised.

Warning

Note that successfully parsing source code into an AST object doesn’t guarantee that the source code provided is valid Python code that can be executed as the compilation step can raise further **`SyntaxError`** exceptions. For instance, the source `return 42` generates a valid AST node for a return statement, but it cannot be compiled alone (it needs to be inside a function node).

In particular, **`ast.parse()`** won’t do any scoping checks, which the compilation step does.

Warning

It is possible to crash the Python interpreter with a sufficiently large/complex string due to stack depth limitations in Python’s AST compiler.

Changed in version 3.8: Added `type_comments`, `mode='func_type'` and `feature_version`.

Changed in version 3.13: The minimum supported version for `feature_version` is now `(3, 7)`. The `optimize` argument was added.

ast.unparse(*ast_obj*)

Unparse an `ast.AST` object and generate a string with code that would produce an equivalent `ast.AST` object if parsed back with `ast.parse()`.

Warning

The produced code string will not necessarily be equal to the original code that generated the `ast.AST` object (without any compiler optimizations, such as constant tuples/frozensets).

Warning

Trying to unparse a highly complex expression would result with `RecursionError`.

Added in version 3.9.

ast.literal_eval(*node_or_string*)

Evaluate an expression node or a string containing only a Python literal or container display. The string or node provided may only consist of the following Python literal structures: strings, bytes, numbers, tuples, lists, dicts, sets, booleans, `None` and `Ellipsis`.

This can be used for evaluating strings containing Python values without the need to parse the values oneself. It is not capable of evaluating arbitrarily complex expressions, for example involving operators or indexing.

This function had been documented as “safe” in the past without defining what that meant. That was misleading. This is specifically designed not to execute Python code, unlike the more general `eval()`. There is no namespace, no name lookups, or ability to call out. But it is not free from attack: A relatively small input can lead to memory exhaustion or to C stack exhaustion, crashing the process. There is also the possibility for excessive CPU consumption denial of service on some inputs. Calling it on untrusted data is thus not recommended.

Warning

It is possible to crash the Python interpreter due to stack depth limitations in Python’s AST compiler.

It can raise `ValueError`, `TypeError`, `SyntaxError`, `MemoryError` and `RecursionError` depending on the malformed input.

Changed in version 3.2: Now allows bytes and set literals.

Changed in version 3.9: Now supports creating empty sets with `'set()'`.

Changed in version 3.10: For string inputs, leading spaces and tabs are now stripped.

ast.get_docstring(*node*, *clean=True*)

Return the docstring of the given *node* (which must be a `FunctionDef`, `AsyncFunctionDef`, `ClassDef`, or `Module` node), or `None` if it has no docstring. If *clean* is true, clean up the docstring's indentation with `inspect.cleandoc()`.

Changed in version 3.5: `AsyncFunctionDef` is now supported.

ast.get_source_segment(*source*, *node*, *, *padded=False*)

Get source code segment of the *source* that generated *node*. If some location information (`lineno`, `end_lineno`, `col_offset`, or `end_col_offset`) is missing, return `None`.

If *padded* is `True`, the first line of a multi-line statement will be padded with spaces to match its original position.

Added in version 3.8.

ast.fix_missing_locations(*node*)

When you compile a node tree with `compile()`, the compiler expects `lineno` and `col_offset` attributes for every node that supports them. This is rather tedious to fill in for generated nodes, so this helper adds these attributes recursively where not already set, by setting them to the values of the parent node. It works recursively starting at *node*.

ast.increment_lineno(*node*, *n=1*)

Increment the line number and end line number of each node in the tree starting at *node* by *n*. This is useful to “move code” to a different location in a file.

ast.copy_location(*new_node*, *old_node*)

Copy source location (`lineno`, `col_offset`, `end_lineno`, and `end_col_offset`) from *old_node* to *new_node* if possible, and return *new_node*.

ast.iter_fields(*node*)

Yield a tuple of `(fieldname, value)` for each field in `node._fields` that is present on *node*.

ast.iter_child_nodes(*node*)

Yield all direct child nodes of *node*, that is, all fields that are nodes and all items of fields that are lists of nodes.

ast.walk(*node*)

Recursively yield all descendant nodes in the tree starting at *node* (including *node* itself), in no specified order. This is useful if you only want to modify nodes in place and don't care about the context.

class ast.NodeVisitor

A node visitor base class that walks the abstract syntax tree and calls a visitor function for every node found. This function may return a value which is forwarded by the **`visit()`** method.

This class is meant to be subclassed, with the subclass adding visitor methods.

visit(*node*)

Visit a node. The default implementation calls the method called `self.visit_classname` where *classname* is the name of the node class, or **`generic_visit()`** if that method doesn't exist.

generic_visit(*node*)

This visitor calls **`visit()`** on all children of the node.

Note that child nodes of nodes that have a custom visitor method won't be visited unless the visitor calls **`generic_visit()`** or visits them itself.

visit_Constant(*node*)

Handles all constant nodes.

Don’t use the `NodeVisitor` if you want to apply changes to nodes during traversal. For this a special visitor exists (`NodeTransformer`) that allows modifications.

Deprecated since version 3.8, removed in version 3.14: Methods **`visit_Num()`**, **`visit_Str()`**, **`visit_Bytes()`**, **`visit_NameConstant()`** and **`visit_Ellipsis()`** will not be called in Python 3.14+. Add the `visit_Constant()` method instead to handle all constant nodes.

class ast.NodeTransformer

A `NodeVisitor` subclass that walks the abstract syntax tree and allows modification of nodes.

The `NodeTransformer` will walk the AST and use the return value of the visitor methods to replace or remove the old node. If the return value of the visitor method is `None`, the node will be removed from its location, otherwise it is replaced with the return value. The return value may be the original node in which case no replacement takes place.

Here is an example transformer that rewrites all occurrences of name lookups (`foo`) to `data['foo']`:

```
class RewriteName(NodeTransformer):

    def visit_Name(self, node):
        return Subscript(
            value=Name(id='data', ctx=Load()),
            slice=Constant(value=node.id),
            ctx=node.ctx
        )
```

Keep in mind that if the node you're operating on has child nodes you must either transform the child nodes yourself or call the `generic_visit()` method for the node first.

For nodes that were part of a collection of statements (that applies to all statement nodes), the visitor may also return a list of nodes rather than just a single node.

If `NodeTransformer` introduces new nodes (that weren't part of original tree) without giving them location information (such as `lineno`), `fix_missing_locations()` should be called with the new sub-tree to recalculate the location information:

```
tree = ast.parse('foo', mode='eval')
new_tree = fix_missing_locations(RewriteName().visit(tree))
```

Usually you use the transformer like this:

```
node = YourTransformer().visit(node)
```

ast.dump(*node*, *annotate_fields=True*, *include_attributes=False*, *, *indent=None*, *show_empty=False*)

Return a formatted dump of the tree in *node*. This is mainly useful for debugging purposes. If *annotate_fields* is true (by default), the returned string will show the names and the values for fields. If *annotate_fields* is false, the result string will be more compact by omitting unambiguous field names. Attributes such as line numbers and column offsets are not dumped by default. If this is wanted, *include_attributes* can be set to true.

If *indent* is a non-negative integer or string, then the tree will be pretty-printed with that indent level. An indent level of 0, negative, or `""` will only insert newlines. `None` (the default) selects the single line representation. Using a positive integer indent indents that many spaces per level. If *indent* is a string (such as `"\t"`), that string is used to indent each level.

If *show_empty* is false (the default), optional empty lists will be omitted from the output. Optional `None` values are always omitted.

Changed in version 3.9: Added the *indent* option.

Changed in version 3.13: Added the *show_empty* option.

```
>>> print(ast.dump(ast.parse("""\
... async def f():
...     await other_func()
... """), indent=4, show_empty=True))
Module(
    body=[
        AsyncFunctionDef(
            name='f',
            args=arguments(
                posonlyargs=[],
                args=[],
                kwonlyargs=[],
                kw_defaults=[],
                defaults=[]),
            body=[
                Expr(
                    value=Await(
                        value=Call(
                            func=Name(id='other_func',
ctx=Load()),
                            args=[],
                            keywords=[])))],
            decorator_list=[],
            type_params=[])],
    type_ignores=[])
```

Compiler flags

The following flags may be passed to `compile()` in order to change effects on the compilation of a program:

ast.PyCF_ALLOW_TOP_LEVEL_AWAIT

Enables support for top-level `await`, `async for`, `async with` and async comprehensions.

Added in version 3.8.

ast.PyCF_ONLY_AST

Generates and returns an abstract syntax tree instead of returning a compiled code object.

ast.PyCF_OPTIMIZED_AST

The returned AST is optimized according to the *optimize* argument in `compile()` or `ast.parse()`.

Added in version 3.13.

ast.PyCF_TYPE_COMMENTS

Enables support for **PEP 484** [https://peps.python.org/pep-0484/] and **PEP 526** [https://peps.python.org/pep-0526/] style type comments (`# type: <type>`, `# type: ignore <stuff>`).

Added in version 3.8.

ast.compare(*a*, *b*, /, *, *compare_attributes=False*)

Recursively compares two ASTs.

compare_attributes affects whether AST attributes are considered in the comparison. If *compare_attributes* is `False` (default), then attributes are ignored. Otherwise they must all be equal. This option is useful to check whether the ASTs are structurally equal but differ in whitespace or similar details. Attributes include line numbers and column offsets.

Added in version 3.14.

Command-line usage

Added in version 3.9.

The `ast` module can be executed as a script from the command line. It is as simple as:

```
python -m ast [-m <mode>] [-a] [infile]
```

The following options are accepted:

-h, --help

Show the help message and exit.

-m <mode>
--mode <mode>

Specify what kind of code must be compiled, like the *mode* argument in `parse()`.

--no-type-comments

Don't parse type comments.

-a, --include-attributes

Include attributes such as line numbers and column offsets.

-i <indent>
--indent <indent>

Indentation of nodes in AST (number of spaces).

--feature-version <version>

Python version in the format 3.x (for example, 3.10). Defaults to the current version of the interpreter.

Added in version 3.14.

-O <level>
--optimize <level>

Optimization level for parser. Defaults to no optimization.

Added in version 3.14.

--show-empty

Show empty lists and fields that are `None`. Defaults to not showing empty objects.

Added in version 3.14.

If `infile` is specified its contents are parsed to AST and dumped to stdout. Otherwise, the content is read from stdin.

See also

Green Tree Snakes [https://greentreesnakes.readthedocs.io/], an external documentation resource, has good details on working with Python ASTs.

ASTTokens [https://asttokens.readthedocs.io/en/latest/user-guide.html] annotates Python ASTs with the positions of tokens and text in the source code that generated them. This is helpful for tools that make source code transformations.

leoAst.py [https://leo-editor.github.io/leo-editor/appendices.html#leoast-py] unifies the token-based and parse-tree-based views of python programs by inserting two-way links between tokens and ast nodes.

LibCST [https://libcst.readthedocs.io/] parses code as a Concrete Syntax Tree that looks like an ast tree and keeps all formatting details. It’s useful for building automated refactoring (codemod) applications and linters.

Parso [https://parso.readthedocs.io] is a Python parser that supports error recovery and round-trip parsing for different Python versions (in multiple Python versions). Parso is also able to list multiple syntax errors in your Python file.

`symtable` — Access to the compiler’s symbol tables

Source code: Lib/symtable.py
[https://github.com/python/cpython/tree/3.14/Lib/symtable.py]

Symbol tables are generated by the compiler from AST just before bytecode is generated. The symbol table is responsible for calculating the scope of every identifier in the code. **`symtable`** provides an interface to examine these tables.

Generating Symbol Tables

symtable.symtable(*code*, *filename*, *compile_type*)

Return the toplevel **`SymbolTable`** for the Python source *code*. *filename* is the name of the file containing the code. *compile_type* is like the *mode* argument to **`compile()`**.

Examining Symbol Tables

class symtable.SymbolTableType

An enumeration indicating the type of a **`SymbolTable`** object.

MODULE = "*module*"

Used for the symbol table of a module.

FUNCTION = "*function*"

Used for the symbol table of a function.

CLASS = "*class*"

Used for the symbol table of a class.

The following members refer to different flavors of annotation scopes.

ANNOTATION = "*annotation*"

Used for annotations if `from __future__ import annotations` is active.

TYPE_ALIAS = "*type alias*"

Used for the symbol table of **`type`** constructions.

TYPE_PARAMETERS = "*type parameters*"

Used for the symbol table of generic functions or generic classes.

TYPE_VARIABLE = "*type variable*"

Used for the symbol table of the bound, the constraint tuple or the default value of a single type variable in the formal sense, i.e., a TypeVar, a TypeVarTuple or a ParamSpec object (the latter two do not support a bound or a constraint tuple).

Added in version 3.13.

class symtable.SymbolTable

A namespace table for a block. The constructor is not public.

get_type()

Return the type of the symbol table. Possible values are members of the **`SymbolTableType`** enumeration.

Changed in version 3.12: Added `'annotation'`, `'TypeVar bound'`, `'type alias'`, and `'type parameter'` as possible return values.

Changed in version 3.13: Return values are members of the **`SymbolTableType`** enumeration.

The exact values of the returned string may change in the future, and thus, it is recommended to use **`SymbolTableType`** members instead of hard-coded strings.

get_id()

Return the table’s identifier.

get_name()

Return the table’s name. This is the name of the class if the table is for a class, the name of the function if the table is for a function, or `'top'` if the table is global (**`get_type()`** returns `'module'`). For type parameter scopes (which are used for generic classes, functions, and type aliases), it is the name of the underlying class, function, or type alias. For type alias scopes, it is the name of the type alias. For **`TypeVar`** bound scopes, it is the name of the `TypeVar`.

get_lineno()

Return the number of the first line in the block this table represents.

is_optimized()

Return `True` if the locals in this table can be optimized.

is_nested()

Return `True` if the block is a nested class or function.

has_children()

Return `True` if the block has nested namespaces within it. These can be obtained with **`get_children()`**.

get_identifiers()

Return a view object containing the names of symbols in the table. See the documentation of view objects.

lookup(*name*)

Lookup *name* in the table and return a **`Symbol`** instance.

get_symbols()

Return a list of **`Symbol`** instances for names in the table.

get_children()

Return a list of the nested symbol tables.

class symtable.Function

A namespace for a function or method. This class inherits from **`SymbolTable`**.

get_parameters()

Return a tuple containing names of parameters to this function.

get_locals()

Return a tuple containing names of locals in this function.

get_globals()

Return a tuple containing names of globals in this function.

get_nonlocals()

Return a tuple containing names of explicitly declared nonlocals in this function.

get_frees()

Return a tuple containing names of free (closure) variables in this function.

class symtable.Class

A namespace of a class. This class inherits from **`SymbolTable`**.

get_methods()

Return a tuple containing the names of method-like functions declared in the class.

Here, the term ‘method’ designates *any* function defined in the class body via **`def`** or **`async def`**.

Functions defined in a deeper scope (e.g., in an inner class) are not picked up by **`get_methods()`**.

For example:

```
>>> import symtable
>>> st = symtable.symtable('''
... def outer(): pass
...
... class A:
...    def f():
...        def w(): pass
...
...    def g(self): pass
...
...    @classmethod
...    async def h(cls): pass
...
...    global outer
...    def outer(self): pass
... ''', 'test', 'exec')
```

```
>>> class_A = st.get_children()[2]
>>> class_A.get_methods()
('f', 'g', 'h')
```

Although `A().f()` raises [`TypeError`] at runtime, `A.f` is still considered as a method-like function.

Deprecated since version 3.14, will be removed in version 3.16.

class symtable.Symbol

An entry in a [`SymbolTable`] corresponding to an identifier in the source. The constructor is not public.

get_name()

Return the symbol’s name.

is_referenced()

Return `True` if the symbol is used in its block.

is_imported()

Return `True` if the symbol is created from an import statement.

is_parameter()

Return `True` if the symbol is a parameter.

is_type_parameter()

Return `True` if the symbol is a type parameter.

Added in version 3.14.

is_global()

Return `True` if the symbol is global.

is_nonlocal()

Return `True` if the symbol is nonlocal.

is_declared_global()

Return `True` if the symbol is declared global with a global statement.

is_local()

Return `True` if the symbol is local to its block.

is_annotated()

Return `True` if the symbol is annotated.

Added in version 3.6.

is_free()

Return `True` if the symbol is referenced in its block, but not assigned to.

is_free_class()

Return *True* if a class-scoped symbol is free from the perspective of a method.

Consider the following example:

```
def f():
    x = 1  # function-scoped
    class C:
        x = 2  # class-scoped
        def method(self):
            return x
```

In this example, the class-scoped symbol `x` is considered to be free from the perspective of `C.method`, thereby allowing the latter to return *1* at runtime and not *2*.

Added in version 3.14.

is_assigned()

Return `True` if the symbol is assigned to in its block.

is_comp_iter()

Return `True` if the symbol is a comprehension iteration variable.

Added in version 3.14.

is_comp_cell()

Return `True` if the symbol is a cell in an inlined comprehension.

Added in version 3.14.

is_namespace()

Return `True` if name binding introduces new namespace.

If the name is used as the target of a function or class statement, this will be true.

For example:

```
>>> table = symtable.symtable("def some_func(): pass",
"string", "exec")
>>> table.lookup("some_func").is_namespace()
True
```

Note that a single name can be bound to multiple objects. If the result is `True`, the name may also be bound to other objects, like an int or list, that does not introduce a new namespace.

get_namespaces()

Return a list of namespaces bound to this name.

get_namespace()

Return the namespace bound to this name. If more than one or no namespace is bound to this name, a `ValueError` is raised.

Command-Line Usage

Added in version 3.13.

The `symtable` module can be executed as a script from the command line.

```
python -m symtable [infile...]
```

Symbol tables are generated for the specified Python source files and dumped to stdout. If no input file is specified, the content is read from stdin.

`token` — Constants used with Python parse trees

Source code: Lib/token.py [https://github.com/python/cpython/tree/3.14/Lib/token.py]

This module provides constants which represent the numeric values of leaf nodes of the parse tree (terminal tokens). Refer to the file `Grammar/Tokens` in the Python distribution for the definitions of the names in the context of the language grammar. The specific numeric values which the names map to may change between Python versions.

The module also provides a mapping from numeric codes to names and some functions. The functions mirror definitions in the Python C header files.

Note that a token’s value may depend on tokenizer options. For example, a `"+"` token may be reported as either **`PLUS`** or **`OP`**, or a `"match"` token may be either **`NAME`** or **`SOFT_KEYWORD`**.

token.tok_name

Dictionary mapping the numeric values of the constants defined in this module back to name strings, allowing more human-readable representation of parse trees to be generated.

token.ISTERMINAL(*x*)

Return `True` for terminal token values.

token.ISNONTERMINAL(*x*)

Return `True` for non-terminal token values.

token.ISEOF(*x*)

Return `True` if *x* is the marker indicating the end of input.

The token constants are:

token.NAME

Token value that indicates an identifier. Note that keywords are also initially tokenized as `NAME` tokens.

token.NUMBER

Token value that indicates a numeric literal

token.STRING

Token value that indicates a string or byte literal, excluding formatted string literals. The token string is not interpreted: it includes the surrounding quotation marks and the prefix (if given); backslashes are included literally, without processing escape sequences.

token.OP

A generic token value that indicates an operator or delimiter.

CPython implementation detail: This value is only reported by the **`tokenize`** module. Internally, the tokenizer uses exact token types instead.

token.COMMENT

Token value used to indicate a comment. The parser ignores **`COMMENT`** tokens.

token.NEWLINE

Token value that indicates the end of a logical line.

token.NL

Token value used to indicate a non-terminating newline. **`NL`** tokens are generated when a logical line of code is continued over multiple physical lines. The parser ignores **`NL`** tokens.

token.INDENT

Token value used at the beginning of a logical line to indicate the start of an indented block.

token.DEDENT

Token value used at the beginning of a logical line to indicate the end of an indented block.

token.FSTRING_START

Token value used to indicate the beginning of an f-string literal.

CPython implementation detail: The token string includes the prefix and the opening quote(s), but none of the contents of the literal.

token.FSTRING_MIDDLE

Token value used for literal text inside an f-string literal, including format specifications.

CPython implementation detail: Replacement fields (that is, the non-literal parts of f-strings) use the same tokens as other expressions, and are delimited by **`LBRACE`**, **`RBRACE`**, **`EXCLAMATION`** and **`COLON`** tokens.

token.FSTRING_END

Token value used to indicate the end of a f-string.

CPython implementation detail: The token string contains the closing quote(s).

token.TSTRING_START

Token value used to indicate the beginning of a template string literal.

CPython implementation detail: The token string includes the prefix and the opening quote(s), but none of the contents of the literal.

Added in version 3.14.

token.TSTRING_MIDDLE

Token value used for literal text inside a template string literal including format specifications.

CPython implementation detail: Replacement fields (that is, the non-literal parts of t-strings) use the same tokens as other expressions, and are delimited by **`LBRACE`**, **`RBRACE`**, **`EXCLAMATION`** and **`COLON`** tokens.

Added in version 3.14.

token.TSTRING_END

Token value used to indicate the end of a template string literal.

CPython implementation detail: The token string contains the closing quote(s).

Added in version 3.14.

token.ENDMARKER

Token value that indicates the end of input. Used in top-level grammar rules.

token.ENCODING

Token value that indicates the encoding used to decode the source bytes into text. The first token returned by `tokenize.tokenize()` will always be an `ENCODING` token.

CPython implementation detail: This token type isn't used by the C tokenizer but is needed for the `tokenize` module.

The following token types are not produced by the `tokenize` module, and are defined for special uses in the tokenizer or parser:

token.TYPE_IGNORE

Token value indicating that a `type: ignore` comment was recognized. Such tokens are produced instead of regular `COMMENT` tokens only with the `PyCF_TYPE_COMMENTS` flag.

token.TYPE_COMMENT

Token value indicating that a type comment was recognized. Such tokens are produced instead of regular `COMMENT` tokens only with the `PyCF_TYPE_COMMENTS` flag.

token.SOFT_KEYWORD

Token value indicating a soft keyword.

The tokenizer never produces this value. To check for a soft keyword, pass a `NAME` token's string to `keyword.issoftkeyword()`.

token.ERRORTOKEN

Token value used to indicate wrong input.

The `tokenize` module generally indicates errors by raising exceptions instead of emitting this token. It can also emit tokens such as `OP` or `NAME` with strings that are later rejected by the parser.

The remaining tokens represent specific operators and delimiters. (The `tokenize` module reports these as `OP`; see `exact_type` in the **`tokenize`** documentation for details.)

Token	**Value**
token.LPAR	`"("`
token.RPAR	`")"`
token.LSQB	`"["`
token.RSQB	`"]"`
token.COLON	`":"`
token.COMMA	`","`
token.SEMI	`";"`
token.PLUS	`"+"`
token.MINUS	`"-"`
token.STAR	`"*"`
token.SLASH	`"/"`

Token	Value
token.VBAR	`"|"`
token.AMPER	`"&"`
token.LESS	`"<"`
token.GREATER	`">"`
token.EQUAL	`"="`
token.DOT	`"."`
token.PERCENT	`"%"`
token.LBRACE	`"{"`
token.RBRACE	`"}"`
token.EQEQUAL	`"=="`
token.NOTEQUAL	`"!="`
token.LESSEQUAL	`"<="`

Token	**Value**
token.GREATEREQUAL	`">="`
token.TILDE	`"~"`
token.CIRCUMFLEX	`"^"`
token.LEFTSHIFT	`"<<"`
token.RIGHTSHIFT	`">>"`
token.DOUBLESTAR	`"**"`
token.PLUSEQUAL	`"+="`
token.MINEQUAL	`"-="`
token.STAREQUAL	`"*="`
token.SLASHEQUAL	`"/="`
token.PERCENTEQUAL	`"%="`
token.AMPEREQUAL	`"&="`

Token	Value
token.VBAREQUAL	`"|="`
token.CIRCUMFLEXEQUAL	`"^="`
token.LEFTSHIFTEQUAL	`"<<="`
token.RIGHTSHIFTEQUAL	`">>="`
token.DOUBLESTAREQUAL	`"**="`
token.DOUBLESLASH	`"//"`
token.DOUBLESLASHEQUAL	`"//="`
token.AT	`"@"`
token.ATEQUAL	`"@="`
token.RARROW	`"->"`
token.ELLIPSIS	`"..."`
token.COLONEQUAL	`":="`

Token	**Value**
token.EXCLAMATION	`"!"`

The following non-token constants are provided:

token.N_TOKENS

The number of token types defined in this module.

token.EXACT_TOKEN_TYPES

A dictionary mapping the string representation of a token to its numeric code.

Added in version 3.8.

Changed in version 3.5: Added **`AWAIT`** and **`ASYNC`** tokens.

Changed in version 3.7: Added `COMMENT`, `NL` and `ENCODING` tokens.

Changed in version 3.7: Removed **`AWAIT`** and **`ASYNC`** tokens. “async” and “await” are now tokenized as `NAME` tokens.

Changed in version 3.8: Added `TYPE_COMMENT`, `TYPE_IGNORE`, `COLONEQUAL`. Added **`AWAIT`** and **`ASYNC`** tokens back (they’re needed to support parsing older Python versions for `ast.parse()` with `feature_version` set to 6 or lower).

Changed in version 3.12: Added `EXCLAMATION`.

Changed in version 3.13: Removed **`AWAIT`** and **`ASYNC`** tokens again.

`keyword` — Testing for Python keywords

Source code: Lib/keyword.py [https://github.com/python/cpython/tree/3.14/Lib/keyword.py]

This module allows a Python program to determine if a string is a keyword or soft keyword.

keyword.iskeyword(*s*)

Return `True` if *s* is a Python keyword.

keyword.kwlist

Sequence containing all the keywords defined for the interpreter. If any keywords are defined to only be active when particular **`__future__`** statements are in effect, these will be included as well.

keyword.issoftkeyword(*s*)

Return `True` if *s* is a Python soft keyword.

Added in version 3.9.

keyword.softkwlist

Sequence containing all the soft keywords defined for the interpreter. If any soft keywords are defined to only be active when particular **`__future__`** statements are in effect, these will be included as well.

Added in version 3.9.

`tokenize` — Tokenizer for Python source

Source code: Lib/tokenize.py
[https://github.com/python/cpython/tree/3.14/Lib/tokenize.py]

The **`tokenize`** module provides a lexical scanner for Python source code, implemented in Python. The scanner in this module returns comments as tokens as well, making it useful for implementing “pretty-printers”, including colorizers for on-screen displays.

To simplify token stream handling, all operator and delimiter tokens and **`Ellipsis`** are returned using the generic **`OP`** token type. The exact type can be determined by checking the `exact_type` property on the named tuple returned from **`tokenize.tokenize()`**.

Warning

Note that the functions in this module are only designed to parse syntactically valid Python code (code that does not raise when parsed using **`ast.parse()`**). The behavior of the functions in this module is **undefined** when providing invalid Python code and it can change at any point.

Tokenizing Input

The primary entry point is a generator:

tokenize.tokenize(*readline*)

The **`tokenize()`** generator requires one argument, *readline*, which must be a callable object which provides the same interface as the **`io.IOBase.readline()`** method of file objects. Each call to the function should return one line of input as bytes.

The generator produces 5-tuples with these members: the token type; the token string; a 2-tuple `(srow, scol)` of ints specifying the row and column where the token begins in the source; a 2-tuple `(erow, ecol)` of ints specifying the row and column where the token ends in the source; and the line on which the token was found. The line passed (the last tuple item) is the *physical* line. The 5 tuple is returned as a named tuple with the field names: `type string start end line`.

The returned named tuple has an additional property named `exact_type` that contains the exact operator type for **`OP`** tokens. For all other token types `exact_type` equals the named tuple `type` field.

Changed in version 3.1: Added support for named tuples.

Changed in version 3.3: Added support for `exact_type`.

`tokenize()` determines the source encoding of the file by looking for a UTF-8 BOM or encoding cookie, according to **PEP 263** [https://peps.python.org/pep-0263/].

tokenize.generate_tokens(*readline*)

Tokenize a source reading unicode strings instead of bytes.

Like **`tokenize()`**, the *readline* argument is a callable returning a single line of input. However, **`generate_tokens()`**

expects *readline* to return a str object rather than bytes.

The result is an iterator yielding named tuples, exactly like `tokenize()`. It does not yield an `ENCODING` token.

All constants from the `token` module are also exported from `tokenize`.

Another function is provided to reverse the tokenization process. This is useful for creating tools that tokenize a script, modify the token stream, and write back the modified script.

tokenize.untokenize(*iterable*)

Converts tokens back into Python source code. The *iterable* must return sequences with at least two elements, the token type and the token string. Any additional sequence elements are ignored.

The result is guaranteed to tokenize back to match the input so that the conversion is lossless and round-trips are assured. The guarantee applies only to the token type and token string as the spacing between tokens (column positions) may change.

It returns bytes, encoded using the `ENCODING` token, which is the first token sequence output by `tokenize()`. If there is no encoding token in the input, it returns a str instead.

`tokenize()` needs to detect the encoding of source files it tokenizes. The function it uses to do this is available:

tokenize.detect_encoding(*readline*)

The `detect_encoding()` function is used to detect the encoding that should be used to decode a Python source file. It requires one argument, readline, in the same way as the `tokenize()` generator.

It will call readline a maximum of twice, and return the encoding used (as a string) and a list of any lines (not decoded from bytes) it has read in.

It detects the encoding from the presence of a UTF-8 BOM or an encoding cookie as specified in **PEP 263** [https://peps.python.org/pep-0263/]. If both a BOM and a cookie are present, but disagree, a `SyntaxError` will be raised. Note that if the BOM is found, `'utf-8-sig'` will be returned as an encoding.

If no encoding is specified, then the default of `'utf-8'` will be returned.

Use `open()` to open Python source files: it uses `detect_encoding()` to detect the file encoding.

tokenize.open(*filename*)

Open a file in read only mode using the encoding detected by `detect_encoding()`.

Added in version 3.2.

exception tokenize.TokenError

Raised when either a docstring or expression that may be split over several lines is not completed anywhere in the file, for example:

```
"""Beginning of
docstring
```

or:

```
[1,
 2,
    3
```

Command-Line Usage

Added in version 3.3.

The **`tokenize`** module can be executed as a script from the command line. It is as simple as:

```
python -m tokenize [-e] [filename.py]
```

The following options are accepted:

-h, --help
    show this help message and exit

-e, --exact
    display token names using the exact type

If `filename.py` is specified its contents are tokenized to stdout. Otherwise, tokenization is performed on stdin.

Examples

Example of a script rewriter that transforms float literals into Decimal objects:

```
from tokenize import tokenize, untokenize, NUMBER, STRING,
NAME, OP
from io import BytesIO

def decistmt(s):
    """Substitute Decimals for floats in a string of
statements.

    >>> from decimal import Decimal
    >>> s = 'print(+21.3e-5*-.1234/81.7)'
    >>> decistmt(s)
    "print (+Decimal ('21.3e-5')*-Decimal ('.1234')/Decimal
('81.7'))"

    The format of the exponent is inherited from the platform C
library.
    Known cases are "e-007" (Windows) and "e-07" (not Windows).
Since
    we're only showing 12 digits, and the 13th isn't close to
5, the
    rest of the output should be platform-independent.

    >>> exec(s)  #doctest: +ELLIPSIS
    -3.21716034272e-0...7

    Output from calculations with Decimal should be identical
across all
    platforms.

    >>> exec(decistmt(s))
    -3.217160342717258261933904529E-7
    """
    result = []
    g = tokenize(BytesIO(s.encode('utf-8')).readline)  #
tokenize the string
    for toknum, tokval, _, _, _ in g:
        if toknum == NUMBER and '.' in tokval:  # replace
NUMBER tokens
            result.extend([
                (NAME, 'Decimal'),
                (OP, '('),
                (STRING, repr(tokval)),
                (OP, ')')
            ])
        else:
            result.append((toknum, tokval))
    return untokenize(result).decode('utf-8')
```

Example of tokenizing from the command line. The script:

```
def say_hello():
    print("Hello, World!")

say_hello()
```

will be tokenized to the following output where the first column is the range of the line/column coordinates where the token is found, the second column is the name of the token, and the final column is the value of the token (if any)

```
$ python -m tokenize hello.py
0,0-0,0:            ENCODING       'utf-8'
1,0-1,3:            NAME           'def'
1,4-1,13:           NAME           'say_hello'
1,13-1,14:          OP             '('
1,14-1,15:          OP             ')'
1,15-1,16:          OP             ':'
1,16-1,17:          NEWLINE        '\n'
2,0-2,4:            INDENT         '    '
2,4-2,9:            NAME           'print'
2,9-2,10:           OP             '('
2,10-2,25:          STRING         '"Hello, World!"'
2,25-2,26:          OP             ')'
2,26-2,27:          NEWLINE        '\n'
3,0-3,1:            NL             '\n'
4,0-4,0:            DEDENT         ''
4,0-4,9:            NAME           'say_hello'
4,9-4,10:           OP             '('
4,10-4,11:          OP             ')'
4,11-4,12:          NEWLINE        '\n'
5,0-5,0:            ENDMARKER      ''
```

The exact token type names can be displayed using the **`-e`** option:

```
$ python -m tokenize -e hello.py
0,0-0,0:            ENCODING       'utf-8'
1,0-1,3:            NAME           'def'
1,4-1,13:           NAME           'say_hello'
1,13-1,14:          LPAR           '('
1,14-1,15:          RPAR           ')'
1,15-1,16:          COLON          ':'
1,16-1,17:          NEWLINE        '\n'
2,0-2,4:            INDENT         '    '
2,4-2,9:            NAME           'print'
2,9-2,10:           LPAR           '('
2,10-2,25:          STRING         '"Hello, World!"'
2,25-2,26:          RPAR           ')'
2,26-2,27:          NEWLINE        '\n'
3,0-3,1:            NL             '\n'
4,0-4,0:            DEDENT         ''
4,0-4,9:            NAME           'say_hello'
4,9-4,10:           LPAR           '('
4,10-4,11:          RPAR           ')'
4,11-4,12:          NEWLINE        '\n'
5,0-5,0:            ENDMARKER      ''
```

Example of tokenizing a file programmatically, reading unicode strings instead of bytes with **`generate_tokens()`**:

```
import tokenize

with tokenize.open('hello.py') as f:
    tokens = tokenize.generate_tokens(f.readline)
    for token in tokens:
        print(token)
```

Or reading bytes directly with **`tokenize()`**:

```
import tokenize

with open('hello.py', 'rb') as f:
    tokens = tokenize.tokenize(f.readline)
    for token in tokens:
        print(token)
```

`tabnanny` — Detection of ambiguous indentation

Source code: Lib/tabnanny.py
[https://github.com/python/cpython/tree/3.14/Lib/tabnanny.py]

For the time being this module is intended to be called as a script. However it is possible to import it into an IDE and use the function **`check()`** described below.

Note

The API provided by this module is likely to change in future releases; such changes may not be backward compatible.

tabnanny.check(*file_or_dir*)

If *file_or_dir* is a directory and not a symbolic link, then recursively descend the directory tree named by *file_or_dir*, checking all `.py` files along the way. If *file_or_dir* is an ordinary Python source file, it is checked for whitespace related problems. The diagnostic messages are written to standard output using the **`print()`** function.

tabnanny.verbose

Flag indicating whether to print verbose messages. This is incremented by the `-v` option if called as a script.

tabnanny.filename_only

Flag indicating whether to print only the filenames of files containing whitespace related problems. This is set to true by the `-q` option if called as a script.

exception tabnanny.NannyNag

Raised by `process_tokens()` if detecting an ambiguous indent. Captured and handled in `check()`.

tabnanny.process_tokens(*tokens*)

This function is used by `check()` to process tokens generated by the `tokenize` module.

See also

Module `tokenize`

Lexical scanner for Python source code.

`pyclbr` — Python module browser support

Source code: Lib/pyclbr.py
[https://github.com/python/cpython/tree/3.14/Lib/pyclbr.py]

The `pyclbr` module provides limited information about the functions, classes, and methods defined in a Python-coded module. The information is sufficient to implement a module browser. The information is extracted from the Python source code rather than by importing the module, so this module is safe to use with untrusted code. This restriction makes it impossible to use this module with modules not implemented in Python, including all standard and optional extension modules.

pyclbr.readmodule(*module*, *path=None*)

Return a dictionary mapping module-level class names to class descriptors. If possible, descriptors for imported base classes are included. Parameter *module* is a string with the name of the module to read; it may be the name of a module within a package. If given, *path* is a sequence of directory paths prepended to `sys.path`, which is used to locate the module source code.

This function is the original interface and is only kept for back compatibility. It returns a filtered version of the following.

pyclbr.readmodule_ex(*module*, *path=None*)

Return a dictionary-based tree containing a function or class descriptors for each function and class defined in the module with a `def` or `class` statement. The returned dictionary maps module-level function and class names to their descriptors. Nested objects are entered into the children dictionary of their parent. As with readmodule, *module* names the module to be read and *path* is prepended to sys.path. If the module being read is a package, the returned dictionary has a key `'__path__'` whose value is a list containing the package search path.

Added in version 3.7: Descriptors for nested definitions. They are accessed through the new children attribute. Each has a new parent attribute.

The descriptors returned by these functions are instances of Function and Class classes. Users are not expected to create instances of these classes.

Function Objects

class pyclbr.Function

Class **`Function`** instances describe functions defined by def statements. They have the following attributes:

file

Name of the file in which the function is defined.

module

The name of the module defining the function described.

name

The name of the function.

lineno

The line number in the file where the definition starts.

parent

For top-level functions, `None`. For nested functions, the parent.

Added in version 3.7.

children

A **`dictionary`** mapping names to descriptors for nested functions and classes.

Added in version 3.7.

is_async

`True` for functions that are defined with the **`async`** prefix, `False` otherwise.

Added in version 3.10.

Class Objects

class pyclbr.Class

Class **`Class`** instances describe classes defined by class statements. They have the same attributes as **`Functions`** and two more.

file

Name of the file in which the class is defined.

module

The name of the module defining the class described.

name

The name of the class.

lineno

The line number in the file where the definition starts.

parent

For top-level classes, `None`. For nested classes, the parent.

Added in version 3.7.

children

A dictionary mapping names to descriptors for nested functions and classes.

Added in version 3.7.

super

A list of **`Class`** objects which describe the immediate base classes of the class being described. Classes which are named as superclasses but which are not discoverable by `readmodule_ex()` are listed as a string with the class name instead of as **`Class`** objects.

methods

A `dictionary` mapping method names to line numbers. This can be derived from the newer `children` dictionary, but remains for back-compatibility.

`py_compile` — Compile Python source files

Source code: Lib/py_compile.py [https://github.com/python/cpython/tree/3.14/Lib/py_compile.py]

The `py_compile` module provides a function to generate a byte-code file from a source file, and another function used when the module source file is invoked as a script.

Though not often needed, this function can be useful when installing modules for shared use, especially if some of the users may not have permission to write the byte-code cache files in the directory containing the source code.

exception py_compile.PyCompileError

Exception raised when an error occurs while attempting to compile the file.

py_compile.compile(*file*, *cfile=None*, *dfile=None*, *doraise=False*, *optimize=-1*, *invalidation_mode=PycInvalidationMode.TIMESTAMP*, *quiet=0*)

Compile a source file to byte-code and write out the byte-code cache file. The source code is loaded from the file named *file*. The byte-code is written to *cfile*, which defaults to the **PEP 3147** [https://peps.python.org/pep-3147/]/**PEP 488** [https://peps.python.org/pep-0488/] path, ending in `.pyc`. For example, if *file* is `/foo/bar/baz.py` *cfile* will default to `/foo/bar/__pycache__/baz.cpython-32.pyc` for Python 3.2. If *dfile* is specified, it is used instead of *file* as the name of the source file from which source lines are obtained for display in exception tracebacks. If *doraise* is true, a `PyCompileError` is raised when an error is encountered while compiling *file*. If *doraise* is false (the default), an error string is written to `sys.stderr`, but no exception is raised. This function returns the path to byte-compiled file, i.e. whatever *cfile* value was used.

The *doraise* and *quiet* arguments determine how errors are handled while compiling file. If *quiet* is 0 or 1, and *doraise* is false, the default behaviour is enabled: an error string is written to `sys.stderr`, and the function returns `None` instead of a path. If *doraise* is true, a `PyCompileError` is raised instead. However if *quiet* is 2, no message is written, and *doraise* has no effect.

If the path that *cfile* becomes (either explicitly specified or computed) is a symlink or non-regular file, `FileExistsError` will be raised. This is to act as a warning that import will turn those paths into regular files if it is allowed to write byte-compiled files to those paths. This is a side-effect of import using file renaming to place the final byte-compiled file into place to prevent concurrent file writing issues.

optimize controls the optimization level and is passed to the built-in `compile()` function. The default of `-1` selects the optimization level of the current interpreter.

invalidation_mode should be a member of the `PycInvalidationMode` enum and controls how the generated bytecode cache is invalidated at runtime. The default is `PycInvalidationMode.CHECKED_HASH` if the **`SOURCE_DATE_EPOCH`** environment variable is set, otherwise the default is `PycInvalidationMode.TIMESTAMP`.

Changed in version 3.2: Changed default value of *cfile* to be **PEP 3147** [https://peps.python.org/pep-3147/]-compliant. Previous default was *file* + `'c'` (`'o'` if optimization was enabled). Also added the *optimize* parameter.

Changed in version 3.4: Changed code to use **`importlib`** for the byte-code cache file writing. This means file creation/writing semantics now match what **`importlib`** does, e.g. permissions, write-and-move semantics, etc. Also added the caveat that **`FileExistsError`** is raised if *cfile* is a symlink or non-regular file.

Changed in version 3.7: The *invalidation_mode* parameter was added as specified in **PEP 552** [https://peps.python.org/pep-0552/]. If the **`SOURCE_DATE_EPOCH`** environment variable is set, *invalidation_mode* will be forced to **`PycInvalidationMode.CHECKED_HASH`**.

Changed in version 3.7.2: The **`SOURCE_DATE_EPOCH`** environment variable no longer overrides the value of the *invalidation_mode* argument, and determines its default value instead.

Changed in version 3.8: The *quiet* parameter was added.

class py_compile.PycInvalidationMode

An enumeration of possible methods the interpreter can use to determine whether a bytecode file is up to date with a source file. The `.pyc` file indicates the desired invalidation mode in its header. See Cached bytecode invalidation for more information on how Python invalidates `.pyc` files at runtime.

Added in version 3.7.

TIMESTAMP

The `.pyc` file includes the timestamp and size of the source file, which Python will compare against the metadata of the source file at runtime to determine if the `.pyc` file needs to be regenerated.

CHECKED_HASH

The `.pyc` file includes a hash of the source file content, which Python will compare against the source at runtime to determine if the `.pyc` file needs to be regenerated.

UNCHECKED_HASH

Like **`CHECKED_HASH`**, the `.pyc` file includes a hash of the source file content. However, Python will at runtime assume the `.pyc` file is up to date and not validate the `.pyc` against the source file at all.

This option is useful when the `.pycs` are kept up to date by some system external to Python like a build system.

Command-Line Interface

This module can be invoked as a script to compile several source files. The files named in *filenames* are compiled and the resulting bytecode is cached in the normal manner. This program does not search a directory structure to locate source files; it only compiles files named explicitly. The exit status is nonzero if one of the files could not be compiled.

<file> ... <fileN>
-

Positional arguments are files to compile. If `-` is the only parameter, the list of files is taken from standard input.

-q, --quiet

Suppress errors output.

Changed in version 3.2: Added support for `-`.

Changed in version 3.10: Added support for `-q`.

See also

Module `compileall`

Utilities to compile all Python source files in a directory tree.

`compileall` — Byte-compile Python libraries

Source code: Lib/compileall.py
[https://github.com/python/cpython/tree/3.14/Lib/compileall.py]

This module provides some utility functions to support installing Python libraries. These functions compile Python source files in a directory tree. This module can be used to create the cached byte-code files at library installation time, which makes them available for use even by users who don’t have write permission to the library directories.

Availability: not WASI.

This module does not work or is not available on WebAssembly. See WebAssembly platforms for more information.

Command-line use

This module can work as a script (using **python -m compileall**) to compile Python sources.

directory ...
file ...

Positional arguments are files to compile or directories that contain source files, traversed recursively. If no argument is given, behave as if the command line was `-l` *`<directories from sys.path>`*.

-l

Do not recurse into subdirectories, only compile source code files directly contained in the named or implied directories.

-f

Force rebuild even if timestamps are up-to-date.

-q

Do not print the list of files compiled. If passed once, error messages will still be printed. If passed twice (`-qq`), all output is suppressed.

-d destdir

Directory prepended to the path to each file being compiled. This will appear in compilation time tracebacks, and is also compiled in to the byte-code file, where it will be used in tracebacks and other messages in cases where the source file does not exist at the time the byte-code file is executed.

-s strip_prefix

Remove the given prefix from paths recorded in the `.pyc` files. Paths are made relative to the prefix.

This option can be used with `-p` but not with `-d`.

-p prepend_prefix

Prepend the given prefix to paths recorded in the `.pyc` files. Use `-p /` to make the paths absolute.

This option can be used with `-s` but not with `-d`.

-x regex

regex is used to search the full path to each file considered for compilation, and if the regex produces a match, the file is skipped.

-i list

Read the file `list` and add each line that it contains to the list of files and directories to compile. If `list` is `-`, read lines from `stdin`.

-b

Write the byte-code files to their legacy locations and names, which may overwrite byte-code files created by another version of Python. The default is to write files to their **PEP 3147** [https://peps.python.org/pep-3147/] locations and names, which allows byte-code files from multiple versions of Python to coexist.

-r

Control the maximum recursion level for subdirectories. If this is given, then `-l` option will not be taken into account. **python -m compileall <directory> -r 0** is equivalent to **python -m compileall <directory> -l**.

-j N

Use *N* workers to compile the files within the given directory. If `0` is used, then the result of `os.process_cpu_count()` will be used.

--invalidation-mode [timestamp|checked-hash|unchecked-hash]

Control how the generated byte-code files are invalidated at runtime. The `timestamp` value, means that `.pyc` files with the source timestamp and size embedded will be generated. The `checked-hash` and `unchecked-hash` values cause hash-based pycs to be generated. Hash-based pycs embed a hash of the source file contents rather than a timestamp. See Cached bytecode invalidation for more information on how Python validates bytecode cache files at runtime. The default is `timestamp` if the **`SOURCE_DATE_EPOCH`** environment variable is not set, and `checked-hash` if the `SOURCE_DATE_EPOCH` environment variable is set.

-o level

Compile with the given optimization level. May be used multiple times to compile for multiple levels at a time (for example, `compileall -o 1 -o 2`).

-e dir

Ignore symlinks pointing outside the given directory.

--hardlink-dupes

If two `.pyc` files with different optimization level have the same content, use hard links to consolidate duplicate files.

Changed in version 3.2: Added the `-i`, `-b` and `-h` options.

Changed in version 3.5: Added the `-j`, `-r`, and `-qq` options. `-q` option was changed to a multilevel value. `-b` will always produce a byte-code file ending in `.pyc`, never `.pyo`.

Changed in version 3.7: Added the `--invalidation-mode` option.

Changed in version 3.9: Added the `-s`, `-p`, `-e` and `--hardlink-dupes` options. Raised the default recursion limit from 10 to `sys.getrecursionlimit()`. Added the possibility to specify the `-o` option multiple times.

There is no command-line option to control the optimization level used by the `compile()` function, because the Python interpreter itself already provides the option: **python -O -m compileall**.

Similarly, the `compile()` function respects the `sys.pycache_prefix` setting. The generated bytecode cache will only be useful if `compile()` is run with the same `sys.pycache_prefix` (if any) that will be used at runtime.

Public functions

compileall.compile_dir(*dir*, *maxlevels=sys.getrecursionlimit()*, *ddir=None*, *force=False*, *rx=None*, *quiet=0*, *legacy=False*, *optimize=-1*, *workers=1*, *invalidation_mode=None*, ***, *stripdir=None*, *prependdir=None*, *limit_sl_dest=None*, *hardlink_dupes=False*)

Recursively descend the directory tree named by *dir*, compiling all `.py` files along the way. Return a true value if all the files compiled successfully, and a false value otherwise.

The *maxlevels* parameter is used to limit the depth of the recursion; it defaults to `sys.getrecursionlimit()`.

If *ddir* is given, it is prepended to the path to each file being compiled for use in compilation time tracebacks, and is also compiled in to the byte-code file, where it will be used in tracebacks and other messages in cases where the source file does not exist at the time the byte-code file is executed.

If *force* is true, modules are re-compiled even if the timestamps are up to date.

If *rx* is given, its `search` method is called on the complete path to each file considered for compilation, and if it returns a true value, the file is skipped. This can be used to exclude files matching a regular expression, given as a re.Pattern object.

If *quiet* is `False` or `0` (the default), the filenames and other information are printed to standard out. Set to `1`, only errors are printed. Set to `2`, all output is suppressed.

If *legacy* is true, byte-code files are written to their legacy locations and names, which may overwrite byte-code files created by another version of Python. The default is to write files to their **PEP 3147** [https://peps.python.org/pep-3147/] locations and names, which allows byte-code files from multiple versions of Python to coexist.

optimize specifies the optimization level for the compiler. It is passed to the built-in `compile()` function. Accepts also a sequence of optimization levels which lead to multiple compilations of one `.py` file in one call.

The argument *workers* specifies how many workers are used to compile files in parallel. The default is to not use multiple workers. If the platform can't use multiple workers and *workers* argument is given, then sequential compilation will be used as a fallback. If *workers* is 0, the number of cores in the system is used. If *workers* is lower than `0`, a `ValueError` will be raised.

invalidation_mode should be a member of the `py_compile.PycInvalidationMode` enum and controls how the generated pycs are invalidated at runtime.

The *stripdir*, *prependdir* and *limit_sl_dest* arguments correspond to the `-s`, `-p` and `-e` options described above. They may be specified as `str` or `os.PathLike`.

If *hardlink_dupes* is true and two `.pyc` files with different optimization level have the same content, use hard links to consolidate duplicate files.

Changed in version 3.2: Added the *legacy* and *optimize* parameter.

Changed in version 3.5: Added the *workers* parameter.

Changed in version 3.5: *quiet* parameter was changed to a multilevel value.

Changed in version 3.5: The *legacy* parameter only writes out `.pyc` files, not `.pyo` files no matter what the value of *optimize* is.

Changed in version 3.6: Accepts a path-like object.

Changed in version 3.7: The *invalidation_mode* parameter was added.

Changed in version 3.7.2: The *invalidation_mode* parameter's default value is updated to `None`.

Changed in version 3.8: Setting *workers* to 0 now chooses the optimal number of cores.

Changed in version 3.9: Added *stripdir*, *prependdir*, *limit_sl_dest* and *hardlink_dupes* arguments. Default value of *maxlevels* was changed from `10` to `sys.getrecursionlimit()`

compileall.compile_file(*fullname*, *ddir=None*, *force=False*, *rx=None*, *quiet=0*, *legacy=False*, *optimize=-1*, *invalidation_mode=None*, *, *stripdir=None*, *prependdir=None*, *limit_sl_dest=None*, *hardlink_dupes=False*)

Compile the file with path *fullname*. Return a true value if the file compiled successfully, and a false value otherwise.

If *ddir* is given, it is prepended to the path to the file being compiled for use in compilation time tracebacks, and is also compiled in to the byte-code file, where it will be used in tracebacks and other messages in cases where the source file does not exist at the time the byte-code file is executed.

If *rx* is given, its `search` method is passed the full path name to the file being compiled, and if it returns a true value, the file is not compiled and `True` is returned. This can be used to exclude files matching a regular expression, given as a re.Pattern object.

If *quiet* is `False` or `0` (the default), the filenames and other information are printed to standard out. Set to `1`, only errors are printed. Set to `2`, all output is suppressed.

If *legacy* is true, byte-code files are written to their legacy locations and names, which may overwrite byte-code files created by another version of Python. The default is to write files to their **PEP 3147** [https://peps.python.org/pep-3147/] locations and names, which allows byte-code files from multiple versions of Python to coexist.

optimize specifies the optimization level for the compiler. It is passed to the built-in `compile()` function. Accepts also a sequence of optimization levels which lead to multiple compilations of one `.py` file in one call.

invalidation_mode should be a member of the `py_compile.PycInvalidationMode` enum and controls how the generated pycs are invalidated at runtime.

The *stripdir*, *prependdir* and *limit_sl_dest* arguments correspond to the `-s`, `-p` and `-e` options described above. They may be specified as `str` or `os.PathLike`.

If *hardlink_dupes* is true and two `.pyc` files with different optimization level have the same content, use hard links to consolidate duplicate files.

Added in version 3.2.

Changed in version 3.5: quiet parameter was changed to a multilevel value.

Changed in version 3.5: The *legacy* parameter only writes out `.pyc` files, not `.pyo` files no matter what the value of *optimize* is.

Changed in version 3.7: The *invalidation_mode* parameter was added.

Changed in version 3.7.2: The *invalidation_mode* parameter's default value is updated to `None`.

Changed in version 3.9: Added *stripdir*, *prependdir*, *limit_sl_dest* and *hardlink_dupes* arguments.

compileall.compile_path(*skip_curdir=True*, *maxlevels=0*, *force=False*, *quiet=0*, *legacy=False*, *optimize=-1*, *invalidation_mode=None*)

Byte-compile all the `.py` files found along `sys.path`. Return a true value if all the files compiled successfully, and a false value otherwise.

If *skip_curdir* is true (the default), the current directory is not included in the search. All other parameters are passed to the **`compile_dir()`** function. Note that unlike the other compile functions, `maxlevels` defaults to `0`.

Changed in version 3.2: Added the *legacy* and *optimize* parameter.

Changed in version 3.5: quiet parameter was changed to a multilevel value.

Changed in version 3.5: The *legacy* parameter only writes out `.pyc` files, not `.pyo` files no matter what the value of *optimize* is.

Changed in version 3.7: The *invalidation_mode* parameter was added.

Changed in version 3.7.2: The *invalidation_mode* parameter's default value is updated to `None`.

To force a recompile of all the `.py` files in the `Lib/` subdirectory and all its subdirectories:

```
import compileall

compileall.compile_dir('Lib/', force=True)

# Perform same compilation, excluding files in .svn
directories.
import re
compileall.compile_dir('Lib/', rx=re.compile(r'[/\\][.]svn'),
force=True)

# pathlib.Path objects can also be used.
import pathlib
compileall.compile_dir(pathlib.Path('Lib/'), force=True)
```

See also

Module `py_compile`
Byte-compile a single source file.

`dis` — Disassembler for Python bytecode

Source code: Lib/dis.py [https://github.com/python/cpython/tree/3.14/Lib/dis.py]

The **`dis`** module supports the analysis of CPython bytecode by disassembling it. The CPython bytecode which this module takes as an input is defined in the file `Include/opcode.h` and used by the compiler and the interpreter.

CPython implementation detail: Bytecode is an implementation detail of the CPython interpreter. No guarantees are made that bytecode will not be added, removed, or changed between versions of Python. Use of this module should not be considered to work across Python VMs or Python releases.

Changed in version 3.6: Use 2 bytes for each instruction. Previously the number of bytes varied by instruction.

Changed in version 3.10: The argument of jump, exception handling and loop instructions is now the instruction offset rather than the byte offset.

Changed in version 3.11: Some instructions are accompanied by one or more inline cache entries, which take the form of **`CACHE`** instructions. These instructions are hidden by default, but can be shown by passing `show_caches=True` to any **`dis`** utility. Furthermore, the interpreter now adapts the bytecode to specialize it for different runtime conditions. The adaptive bytecode can be shown by passing `adaptive=True`.

Changed in version 3.12: The argument of a jump is the offset of the target instruction relative to the instruction that appears immediately after the jump instruction’s **`CACHE`** entries.

As a consequence, the presence of the **`CACHE`** instructions is transparent for forward jumps but needs to be taken into account when reasoning about backward jumps.

Changed in version 3.13: The output shows logical labels rather than instruction offsets for jump targets and exception handlers. The `-O` command line option and the `show_offsets` argument were added.

Changed in version 3.14: The **`-P`** command-line option and the `show_positions` argument were added.

The **`-S`** command-line option is added.

Example: Given the function **`myfunc()`**:

```
def myfunc(alist):
    return len(alist)
```

the following command can be used to display the disassembly of **`myfunc()`**:

```
>>> dis.dis(myfunc)
  2           RESUME                   0

  3           LOAD_GLOBAL              1 (len + NULL)
              LOAD_FAST_BORROW         0 (alist)
              CALL                     1
              RETURN_VALUE
```

(The “2” is a line number).

Command-line interface

The `dis` module can be invoked as a script from the command line:

```
python -m dis [-h] [-C] [-O] [-P] [-S] [infile]
```

The following options are accepted:

-h, --help
: Display usage and exit.

-C, --show-caches
: Show inline caches.

  Added in version 3.13.

-O, --show-offsets
: Show offsets of instructions.

  Added in version 3.13.

-P, --show-positions
: Show positions of instructions in the source code.

  Added in version 3.14.

-S, --specialized
: Show specialized bytecode.

  Added in version 3.14.

If `infile` is specified, its disassembled code will be written to stdout. Otherwise, disassembly is performed on compiled source code received from stdin.

Bytecode analysis

Added in version 3.4.

The bytecode analysis API allows pieces of Python code to be wrapped in a `Bytecode` object that provides easy access to details of the compiled code.

class dis.Bytecode(*x*, *, *first_line=None*, *current_offset=None*, *show_caches=False*, *adaptive=False*, *show_offsets=False*, *show_positions=False*)

Analyse the bytecode corresponding to a function, generator, asynchronous generator, coroutine, method, string of source code, or a code object (as returned by `compile()`).

This is a convenience wrapper around many of the functions listed below, most notably `get_instructions()`, as iterating over a `Bytecode` instance yields the bytecode operations as `Instruction` instances.

If *first_line* is not `None`, it indicates the line number that should be reported for the first source line in the disassembled code. Otherwise, the source line information (if any) is taken directly from the disassembled code object.

If *current_offset* is not `None`, it refers to an instruction offset in the disassembled code. Setting this means `dis()` will display a “current instruction” marker against the specified opcode.

If *show_caches* is `True`, `dis()` will display inline cache entries used by the interpreter to specialize the bytecode.

If *adaptive* is `True`, **`dis()`** will display specialized bytecode that may be different from the original bytecode.

If *show_offsets* is `True`, **`dis()`** will include instruction offsets in the output.

If *show_positions* is `True`, **`dis()`** will include instruction source code positions in the output.

classmethod from_traceback(*tb*, *, *show_caches=False*)

Construct a **`Bytecode`** instance from the given traceback, setting *current_offset* to the instruction responsible for the exception.

codeobj

The compiled code object.

first_line

The first source line of the code object (if available)

dis()

Return a formatted view of the bytecode operations (the same as printed by **`dis.dis()`**, but returned as a multi-line string).

info()

Return a formatted multi-line string with detailed information about the code object, like **`code_info()`**.

Changed in version 3.7: This can now handle coroutine and asynchronous generator objects.

Changed in version 3.11: Added the *show_caches* and *adaptive* parameters.

Changed in version 3.13: Added the *show_offsets* parameter

Changed in version 3.14: Added the *show_positions* parameter.

Example:

```
>>> bytecode = dis.Bytecode(myfunc)
>>> for instr in bytecode:
...     print(instr.opname)
...
RESUME
LOAD_GLOBAL
LOAD_FAST_BORROW
CALL
RETURN_VALUE
```

Analysis functions

The `dis` module also defines the following analysis functions that convert the input directly to the desired output. They can be useful if only a single operation is being performed, so the intermediate analysis object isn't useful:

dis.code_info(*x*)

Return a formatted multi-line string with detailed code object information for the supplied function, generator, asynchronous generator, coroutine, method, source code string or code object.

Note that the exact contents of code info strings are highly implementation dependent and they may change arbitrarily across Python VMs or Python releases.

Added in version 3.2.

Changed in version 3.7: This can now handle coroutine and asynchronous generator objects.

dis.show_code(*x*, *, *file=None*)

Print detailed code object information for the supplied function, method, source code string or code object to *file* (or `sys.stdout` if *file* is not specified).

This is a convenient shorthand for `print(code_info(x), file=file)`, intended for interactive exploration at the interpreter prompt.

Added in version 3.2.

Changed in version 3.4: Added *file* parameter.

dis.dis(*x=None*, *, *file=None*, *depth=None*, *show_caches=False*, *adaptive=False*, *show_offsets=False*, *show_positions=False*)

Disassemble the *x* object. *x* can denote either a module, a class, a method, a function, a generator, an asynchronous generator, a coroutine, a code object, a string of source code or a byte sequence of raw bytecode. For a module, it disassembles all functions. For a class, it disassembles all methods (including class and static methods). For a code object or sequence of raw bytecode, it prints one line per bytecode instruction. It also recursively disassembles nested code objects. These can include generator expressions, nested functions, the bodies of nested classes, and the code objects used for annotation scopes. Strings are first compiled to code objects with the `compile()` built-in function before being disassembled. If no object is provided, this function disassembles the last traceback.

The disassembly is written as text to the supplied *file* argument if provided and to `sys.stdout` otherwise.

The maximal depth of recursion is limited by *depth* unless it is `None`. `depth=0` means no recursion.

If *show_caches* is `True`, this function will display inline cache entries used by the interpreter to specialize the bytecode.

If *adaptive* is `True`, this function will display specialized bytecode that may be different from the original bytecode.

Changed in version 3.4: Added *file* parameter.

Changed in version 3.7: Implemented recursive disassembling and added *depth* parameter.

Changed in version 3.7: This can now handle coroutine and asynchronous generator objects.

Changed in version 3.11: Added the *show_caches* and *adaptive* parameters.

Changed in version 3.13: Added the *show_offsets* parameter.

Changed in version 3.14: Added the *show_positions* parameter.

dis.distb(*tb=None*, *, *file=None*, *show_caches=False*, *adaptive=False*, *show_offset=False*, *show_positions=False*)

Disassemble the top-of-stack function of a traceback, using the last traceback if none was passed. The instruction causing the exception is indicated.

The disassembly is written as text to the supplied *file* argument if provided and to `sys.stdout` otherwise.

Changed in version 3.4: Added *file* parameter.

Changed in version 3.11: Added the *show_caches* and *adaptive* parameters.

Changed in version 3.13: Added the *show_offsets* parameter.

Changed in version 3.14: Added the *show_positions* parameter.

dis.disassemble(*code*, *lasti=-1*, ***, *file=None*, *show_caches=False*, *adaptive=False*, *show_offsets=False*, *show_positions=False*)
dis.disco(*code*, *lasti=-1*, ***, *file=None*, *show_caches=False*, *adaptive=False*, *show_offsets=False*, *show_positions=False*)

Disassemble a code object, indicating the last instruction if *lasti* was provided. The output is divided in the following columns:

1. the source code location of the instruction. Complete location information is shown if *show_positions* is true. Otherwise (the default) only the line number is displayed.
2. the current instruction, indicated as `-->`,
3. a labelled instruction, indicated with `>>`,
4. the address of the instruction,
5. the operation code name,
6. operation parameters, and
7. interpretation of the parameters in parentheses.

The parameter interpretation recognizes local and global variable names, constant values, branch targets, and compare operators.

The disassembly is written as text to the supplied *file* argument if provided and to `sys.stdout` otherwise.

Changed in version 3.4: Added *file* parameter.

Changed in version 3.11: Added the *show_caches* and *adaptive* parameters.

Changed in version 3.13: Added the *show_offsets* parameter.

Changed in version 3.14: Added the *show_positions* parameter.

dis.get_instructions(*x*, *, *first_line=None*, *show_caches=False*, *adaptive=False*)

Return an iterator over the instructions in the supplied function, method, source code string or code object.

The iterator generates a series of `Instruction` named tuples giving the details of each operation in the supplied code.

If *first_line* is not `None`, it indicates the line number that should be reported for the first source line in the disassembled code. Otherwise, the source line information (if any) is taken directly from the disassembled code object.

The *adaptive* parameter works as it does in `dis()`.

Added in version 3.4.

Changed in version 3.11: Added the *show_caches* and *adaptive* parameters.

Changed in version 3.13: The *show_caches* parameter is deprecated and has no effect. The iterator generates the **`Instruction`** instances with the *cache_info* field populated (regardless of the value of *show_caches*) and it no longer generates separate items for the cache entries.

dis.findlinestarts(*code*)

This generator function uses the **`co_lines()`** method of the code object *code* to find the offsets which are starts of lines in the source code. They are generated as `(offset, lineno)` pairs.

Changed in version 3.6: Line numbers can be decreasing. Before, they were always increasing.

Changed in version 3.10: The **PEP 626** [https://peps.python.org/pep-0626/] **`co_lines()`** method is used instead of the **`co_firstlineno`** and **`co_lnotab`** attributes of the code object.

Changed in version 3.13: Line numbers can be `None` for bytecode that does not map to source lines.

dis.findlabels(*code*)

Detect all offsets in the raw compiled bytecode string *code* which are jump targets, and return a list of these offsets.

dis.stack_effect(*opcode*, *oparg=None*, *, *jump=None*)

Compute the stack effect of *opcode* with argument *oparg*.

If the code has a jump target and *jump* is `True`, **`stack_effect()`** will return the stack effect of jumping. If *jump* is `False`, it will return the stack effect of not jumping. And if *jump* is `None` (default), it will return the maximal stack effect of both cases.

Added in version 3.4.

Changed in version 3.8: Added *jump* parameter.

Changed in version 3.13: If `oparg` is omitted (or `None`), the stack effect is now returned for `oparg=0`. Previously this was an error for opcodes that use their arg. It is also no longer an error to pass an integer `oparg` when the `opcode` does not use it; the `oparg` in this case is ignored.

Python Bytecode Instructions

The **`get_instructions()`** function and **`Bytecode`** class provide details of bytecode instructions as **`Instruction`** instances:

class dis.Instruction

Details for a bytecode operation

opcode

numeric code for operation, corresponding to the opcode values listed below and the bytecode values in the Opcode collections.

opname

human readable name for operation

baseopcode

numeric code for the base operation if operation is specialized; otherwise equal to **`opcode`**

baseopname

human readable name for the base operation if operation is specialized; otherwise equal to **`opname`**

arg
: numeric argument to operation (if any), otherwise `None`

oparg
: alias for [`arg`]

argval
: resolved arg value (if any), otherwise `None`

argrepr
: human readable description of operation argument (if any), otherwise an empty string.

offset
: start index of operation within bytecode sequence

start_offset
: start index of operation within bytecode sequence, including prefixed `EXTENDED_ARG` operations if present; otherwise equal to [`offset`]

cache_offset
: start index of the cache entries following the operation

end_offset
: end index of the cache entries following the operation

starts_line
: `True` if this opcode starts a source line, otherwise `False`

line_number
: source line number associated with this opcode (if any), otherwise `None`

is_jump_target

`True` if other code jumps to here, otherwise `False`

jump_target

bytecode index of the jump target if this is a jump operation, otherwise `None`

positions

`dis.Positions` object holding the start and end locations that are covered by this instruction.

cache_info

Information about the cache entries of this instruction, as triplets of the form `(name, size, data)`, where the `name` and `size` describe the cache format and data is the contents of the cache. `cache_info` is `None` if the instruction does not have caches.

Added in version 3.4.

Changed in version 3.11: Field `positions` is added.

Changed in version 3.13: Changed field `starts_line`.

Added fields `start_offset`, `cache_offset`, `end_offset`, `baseopname`, `baseopcode`, `jump_target`, `oparg`, `line_number` and `cache_info`.

class dis.Positions

In case the information is not available, some fields might be `None`.

lineno

end_lineno

col_offset

end_col_offset

Added in version 3.11.

The Python compiler currently generates the following bytecode instructions.

General instructions

In the following, We will refer to the interpreter stack as `STACK` and describe operations on it as if it was a Python list. The top of the stack corresponds to `STACK[-1]` in this language.

NOP

Do nothing code. Used as a placeholder by the bytecode optimizer, and to generate line tracing events.

NOT_TAKEN

Do nothing code. Used by the interpreter to record `BRANCH_LEFT` and `BRANCH_RIGHT` events for `sys.monitoring`.

Added in version 3.14.

POP_ITER

Removes the iterator from the top of the stack.

Added in version 3.14.

POP_TOP

Removes the top-of-stack item:

```
STACK.pop()
```

END_FOR

Removes the top-of-stack item. Equivalent to `POP_TOP`. Used to clean up at the end of loops, hence the name.

Added in version 3.12.

END_SEND

Implements `del STACK[-2]`. Used to clean up when a generator exits.

Added in version 3.12.

COPY(*i*)

Push the i-th item to the top of the stack without removing it from its original location:

```
assert i > 0
STACK.append(STACK[-i])
```

Added in version 3.11.

SWAP(*i*)

Swap the top of the stack with the i-th element:

```
STACK[-i], STACK[-1] = STACK[-1], STACK[-i]
```

Added in version 3.11.

CACHE

Rather than being an actual instruction, this opcode is used to mark extra space for the interpreter to cache useful data directly in the bytecode itself. It is automatically hidden by all `dis` utilities, but can be viewed with `show_caches=True`.

Logically, this space is part of the preceding instruction. Many opcodes expect to be followed by an exact number of caches, and will instruct the interpreter to skip over them at runtime.

Populated caches can look like arbitrary instructions, so great care should be taken when reading or modifying raw, adaptive bytecode containing quickened data.

Added in version 3.11.

Unary operations

Unary operations take the top of the stack, apply the operation, and push the result back on the stack.

UNARY_NEGATIVE
: Implements `STACK[-1] = -STACK[-1]`.

UNARY_NOT
: Implements `STACK[-1] = not STACK[-1]`.

  Changed in version 3.13: This instruction now requires an exact **`bool`** operand.

UNARY_INVERT
: Implements `STACK[-1] = ~STACK[-1]`.

GET_ITER
: Implements `STACK[-1] = iter(STACK[-1])`.

GET_YIELD_FROM_ITER
: If `STACK[-1]` is a generator iterator or coroutine object it is left as is. Otherwise, implements `STACK[-1] = iter(STACK[-1])`.

  Added in version 3.5.

TO_BOOL
: Implements `STACK[-1] = bool(STACK[-1])`.

  Added in version 3.13.

Binary and in-place operations

Binary operations remove the top two items from the stack (`STACK[-1]` and `STACK[-2]`). They perform the operation, then put the result back on the stack.

In-place operations are like binary operations, but the operation is done in-place when `STACK[-2]` supports it, and the resulting `STACK[-1]` may be (but does not have to be) the original `STACK[-2]`.

BINARY_OP(*op*)

Implements the binary and in-place operators (depending on the value of *op*):

```
rhs = STACK.pop()
lhs = STACK.pop()
STACK.append(lhs op rhs)
```

Added in version 3.11.

Changed in version 3.14: With oparg :`NB_SUBSCR`, implements binary subscript (replaces opcode `BINARY_SUBSCR`)

STORE_SUBSCR

Implements:

```
key = STACK.pop()
container = STACK.pop()
value = STACK.pop()
container[key] = value
```

DELETE_SUBSCR

Implements:

```
key = STACK.pop()
container = STACK.pop()
del container[key]
```

BINARY_SLICE

Implements:

```
end = STACK.pop()
start = STACK.pop()
container = STACK.pop()
STACK.append(container[start:end])
```

Added in version 3.12.

STORE_SLICE

Implements:

```
end = STACK.pop()
start = STACK.pop()
container = STACK.pop()
value = STACK.pop()
container[start:end] = value
```

Added in version 3.12.

Coroutine opcodes

GET_AWAITABLE(*where*)

Implements `STACK[-1] = get_awaitable(STACK[-1])`, where `get_awaitable(o)` returns `o` if `o` is a coroutine object or a generator object with the **`CO_ITERABLE_COROUTINE`** flag, or resolves `o.__await__`.

If the `where` operand is nonzero, it indicates where the instruction occurs:

- `1`: After a call to `__aenter__`
- `2`: After a call to `__aexit__`

Added in version 3.5.

Changed in version 3.11: Previously, this instruction did not have an oparg.

GET_AITER

Implements `STACK[-1] = STACK[-1].__aiter__()`.

Added in version 3.5.

Changed in version 3.7: Returning awaitable objects from `__aiter__` is no longer supported.

GET_ANEXT

Implement `STACK.append(get_awaitable(STACK[-1].__anext__()))` to the stack. See `GET_AWAITABLE` for details about `get_awaitable`.

Added in version 3.5.

END_ASYNC_FOR

Terminates an **`async for`** loop. Handles an exception raised when awaiting a next item. The stack contains the async iterable in `STACK[-2]` and the raised exception in `STACK[-1]`. Both are popped. If the exception is not **`StopAsyncIteration`**, it is re-raised.

Added in version 3.8.

Changed in version 3.11: Exception representation on the stack now consist of one, not three, items.

CLEANUP_THROW

Handles an exception raised during a **`throw()`** or **`close()`** call through the current frame. If `STACK[-1]` is an instance of **`StopIteration`**, pop three values from the stack and push its `value` member. Otherwise, re-raise `STACK[-1]`.

Added in version 3.12.

Miscellaneous opcodes

SET_ADD(*i*)

Implements:

```
item = STACK.pop()
set.add(STACK[-i], item)
```

Used to implement set comprehensions.

LIST_APPEND(*i*)

Implements:

```
item = STACK.pop()
list.append(STACK[-i], item)
```

Used to implement list comprehensions.

MAP_ADD(*i*)

Implements:

```
value = STACK.pop()
key = STACK.pop()
dict.__setitem__(STACK[-i], key, value)
```

Used to implement dict comprehensions.

Added in version 3.1.

Changed in version 3.8: Map value is `STACK[-1]` and map key is `STACK[-2]`. Before, those were reversed.

For all of the **`SET_ADD`**, **`LIST_APPEND`** and **`MAP_ADD`** instructions, while the added value or key/value pair is popped off, the container object remains on the stack so that it is available for further iterations of the loop.

RETURN_VALUE

Returns with `STACK[-1]` to the caller of the function.

YIELD_VALUE

Yields `STACK.pop()` from a generator.

Changed in version 3.11: oparg set to be the stack depth.

Changed in version 3.12: oparg set to be the exception block depth, for efficient closing of generators.

Changed in version 3.13: oparg is `1` if this instruction is part of a yield-from or await, and `0` otherwise.

SETUP_ANNOTATIONS

Checks whether `__annotations__` is defined in `locals()`, if not it is set up to an empty `dict`. This opcode is only emitted if a class or module body contains variable annotations statically.

Added in version 3.6.

POP_EXCEPT

Pops a value from the stack, which is used to restore the exception state.

Changed in version 3.11: Exception representation on the stack now consist of one, not three, items.

RERAISE

Re-raises the exception currently on top of the stack. If oparg is non-zero, pops an additional value from the stack which is used to set **`f_lasti`** of the current frame.

Added in version 3.9.

Changed in version 3.11: Exception representation on the stack now consist of one, not three, items.

PUSH_EXC_INFO

Pops a value from the stack. Pushes the current exception to the top of the stack. Pushes the value originally popped back to the stack. Used in exception handlers.

Added in version 3.11.

CHECK_EXC_MATCH

Performs exception matching for `except`. Tests whether the `STACK[-2]` is an exception matching `STACK[-1]`. Pops `STACK[-1]` and pushes the boolean result of the test.

Added in version 3.11.

CHECK_EG_MATCH

Performs exception matching for `except*`. Applies `split(STACK[-1])` on the exception group representing `STACK[-2]`.

In case of a match, pops two items from the stack and pushes the non-matching subgroup (`None` in case of full match) followed by the matching subgroup. When there is no match, pops one item (the match type) and pushes `None`.

Added in version 3.11.

WITH_EXCEPT_START

Calls the function in position 4 on the stack with arguments (type, val, tb) representing the exception at the top of the stack. Used to implement the call `context_manager.__exit__(*exc_info())` when an exception has occurred in a **`with`** statement.

Added in version 3.9.

Changed in version 3.11: The `__exit__` function is in position 4 of the stack rather than 7. Exception representation on the stack now consist of one, not three, items.

LOAD_COMMON_CONSTANT

Pushes a common constant onto the stack. The interpreter contains a hardcoded list of constants supported by this instruction. Used by the **`assert`** statement to load **`AssertionError`**.

Added in version 3.14.

LOAD_BUILD_CLASS

Pushes **`builtins.__build_class__()`** onto the stack. It is later called to construct a class.

GET_LEN

Perform `STACK.append(len(STACK[-1]))`. Used in **`match`** statements where comparison with structure of pattern is needed.

Added in version 3.10.

MATCH_MAPPING

If `STACK[-1]` is an instance of **`collections.abc.Mapping`** (or, more technically: if it has the **`Py_TPFLAGS_MAPPING`** flag set in its **`tp_flags`**), push `True` onto the stack. Otherwise, push `False`.

Added in version 3.10.

MATCH_SEQUENCE

If `STACK[-1]` is an instance of **`collections.abc.Sequence`** and is *not* an instance of **`str`**/**`bytes`**/**`bytearray`** (or, more technically: if it has the **`Py_TPFLAGS_SEQUENCE`** flag set in its **`tp_flags`**), push `True` onto the stack. Otherwise, push `False`.

Added in version 3.10.

MATCH_KEYS

`STACK[-1]` is a tuple of mapping keys, and `STACK[-2]` is the match subject. If `STACK[-2]` contains all of the keys in `STACK[-1]`, push a **`tuple`** containing the corresponding values. Otherwise, push `None`.

Added in version 3.10.

Changed in version 3.11: Previously, this instruction also pushed a boolean value indicating success (`True`) or failure (`False`).

STORE_NAME(*namei*)

Implements `name = STACK.pop()`. *namei* is the index of *name* in the attribute **`co_names`** of the code object. The compiler tries to use **`STORE_FAST`** or **`STORE_GLOBAL`** if possible.

DELETE_NAME(*namei*)

Implements `del name`, where *namei* is the index into **`co_names`** attribute of the code object.

UNPACK_SEQUENCE(*count*)

Unpacks `STACK[-1]` into *count* individual values, which are put onto the stack right-to-left. Require there to be exactly *count* values.:

```
assert(len(STACK[-1]) == count)
STACK.extend(STACK.pop()[:-count-1:-1])
```

UNPACK_EX(*counts*)

Implements assignment with a starred target: Unpacks an iterable in `STACK[-1]` into individual values, where the total number of values can be smaller than the number of items in the iterable: one of the new values will be a list of all leftover items.

The number of values before and after the list value is limited to 255.

The number of values before the list value is encoded in the argument of the opcode. The number of values after the list if any is encoded using an `EXTENDED_ARG`. As a consequence, the argument can be seen as a two bytes values where the low byte of *counts* is the number of values before the list value, the high byte of *counts* the number of values after it.

The extracted values are put onto the stack right-to-left, i.e. `a, *b, c = d` will be stored after execution as `STACK.extend((a, b, c))`.

STORE_ATTR(*namei*)

Implements:

```
obj = STACK.pop()
value = STACK.pop()
obj.name = value
```

where *namei* is the index of name in **`co_names`** of the code object.

DELETE_ATTR(*namei*)

Implements:

```
obj = STACK.pop()
del obj.name
```

where *namei* is the index of name into **`co_names`** of the code object.

STORE_GLOBAL(*namei*)

Works as **`STORE_NAME`**, but stores the name as a global.

DELETE_GLOBAL(*namei*)

Works as **`DELETE_NAME`**, but deletes a global name.

LOAD_CONST(*consti*)

Pushes `co_consts[consti]` onto the stack.

LOAD_SMALL_INT(*i*)

Pushes the integer `i` onto the stack. `i` must be in `range(256)`

Added in version 3.14.

LOAD_NAME(*namei*)

Pushes the value associated with `co_names[namei]` onto the stack. The name is looked up within the locals, then the globals, then the builtins.

LOAD_LOCALS

Pushes a reference to the locals dictionary onto the stack. This is used to prepare namespace dictionaries for **`LOAD_FROM_DICT_OR_DEREF`** and **`LOAD_FROM_DICT_OR_GLOBALS`**.

Added in version 3.12.

LOAD_FROM_DICT_OR_GLOBALS(*i*)

Pops a mapping off the stack and looks up the value for `co_names[namei]`. If the name is not found there, looks it up in the globals and then the builtins, similar to **`LOAD_GLOBAL`**.

This is used for loading global variables in annotation scopes within class bodies.

Added in version 3.12.

BUILD_TEMPLATE

Constructs a new **`Template`** instance from a tuple of strings and a tuple of interpolations and pushes the resulting object onto the stack:

```
interpolations = STACK.pop()
strings = STACK.pop()
STACK.append(_build_template(strings, interpolations))
```

Added in version 3.14.

BUILD_INTERPOLATION(*format*)

Constructs a new **`Interpolation`** instance from a value and its source expression and pushes the resulting object onto the stack.

If no conversion or format specification is present, `format` is set to `2`.

If the low bit of `format` is set, it indicates that the interpolation contains a format specification.

If `format >> 2` is non-zero, it indicates that the interpolation contains a conversion. The value of `format >> 2` is the conversion type (`0` for no conversion, `1` for `!s`, `2` for `!r`, and `3` for `!a`):

```
conversion = format >> 2
if format & 1:
    format_spec = STACK.pop()
else:
    format_spec = None
expression = STACK.pop()
```

```
value = STACK.pop()
STACK.append(_build_interpolation(value, expression,
conversion, format_spec))
```

Added in version 3.14.

BUILD_TUPLE(*count*)

Creates a tuple consuming *count* items from the stack, and pushes the resulting tuple onto the stack:

```
if count == 0:
    value = ()
else:
    value = tuple(STACK[-count:])
    STACK = STACK[:-count]

STACK.append(value)
```

BUILD_LIST(*count*)

Works as **`BUILD_TUPLE`**, but creates a list.

BUILD_SET(*count*)

Works as **`BUILD_TUPLE`**, but creates a set.

BUILD_MAP(*count*)

Pushes a new dictionary object onto the stack. Pops `2 * count` items so that the dictionary holds *count* entries: `{..., STACK[-4]: STACK[-3], STACK[-2]: STACK[-1]}`.

Changed in version 3.5: The dictionary is created from stack items instead of creating an empty dictionary pre-sized to hold *count* items.

BUILD_STRING(*count*)

Concatenates *count* strings from the stack and pushes the resulting string onto the stack.

Added in version 3.6.

LIST_EXTEND(*i*)

Implements:

```
seq = STACK.pop()
list.extend(STACK[-i], seq)
```

Used to build lists.

Added in version 3.9.

SET_UPDATE(*i*)

Implements:

```
seq = STACK.pop()
set.update(STACK[-i], seq)
```

Used to build sets.

Added in version 3.9.

DICT_UPDATE(*i*)

Implements:

```
map = STACK.pop()
dict.update(STACK[-i], map)
```

Used to build dicts.

Added in version 3.9.

DICT_MERGE(*i*)

Like **`DICT_UPDATE`** but raises an exception for duplicate keys.

Added in version 3.9.

LOAD_ATTR(*namei*)

If the low bit of `namei` is not set, this replaces `STACK[-1]` with `getattr(STACK[-1], co_names[namei>>1])`.

If the low bit of `namei` is set, this will attempt to load a method named `co_names[namei>>1]` from the `STACK[-1]` object. `STACK[-1]` is popped. This bytecode distinguishes two cases: if `STACK[-1]` has a method with the correct name, the bytecode pushes the unbound method and `STACK[-1]`. `STACK[-1]` will be used as the first argument (`self`) by **`CALL`** or **`CALL_KW`** when calling the unbound method. Otherwise, `NULL` and the object returned by the attribute lookup are pushed.

Changed in version 3.12: If the low bit of `namei` is set, then a `NULL` or `self` is pushed to the stack before the attribute or unbound method respectively.

LOAD_SUPER_ATTR(*namei*)

This opcode implements **`super()`**, both in its zero-argument and two-argument forms (e.g. `super().method()`, `super().attr` and `super(cls, self).method()`, `super(cls, self).attr`).

It pops three values from the stack (from top of stack down):

- `self`: the first argument to the current method
- `cls`: the class within which the current method was defined
- the global `super`

With respect to its argument, it works similarly to **`LOAD_ATTR`**, except that `namei` is shifted left by 2 bits instead of 1.

The low bit of `namei` signals to attempt a method load, as with **`LOAD_ATTR`**, which results in pushing `NULL` and the loaded method. When it is unset a single value is pushed to the stack.

The second-low bit of `namei`, if set, means that this was a two-argument call to **`super()`** (unset means zero-argument).

Added in version 3.12.

COMPARE_OP(*opname*)

Performs a Boolean operation. The operation name can be found in `cmp_op[opname >> 5]`. If the fifth-lowest bit of `opname` is set (`opname & 16`), the result should be coerced to `bool`.

Changed in version 3.13: The fifth-lowest bit of the oparg now indicates a forced conversion to **`bool`**.

IS_OP(*invert*)

Performs `is` comparison, or `is not` if `invert` is 1.

Added in version 3.9.

CONTAINS_OP(*invert*)

Performs `in` comparison, or `not in` if `invert` is 1.

Added in version 3.9.

IMPORT_NAME(*namei*)

Imports the module `co_names[namei]`. `STACK[-1]` and `STACK[-2]` are popped and provide the *fromlist* and *level* arguments of **`__import__()`**. The module object is pushed onto the stack. The current namespace is not affected: for a proper import statement, a subsequent **`STORE_FAST`** instruction modifies the namespace.

IMPORT_FROM(*namei*)

Loads the attribute `co_names[namei]` from the module found in `STACK[-1]`. The resulting object is pushed onto the stack, to be subsequently stored by a **`STORE_FAST`** instruction.

JUMP_FORWARD(*delta*)

Increments bytecode counter by *delta*.

JUMP_BACKWARD(*delta*)

Decrements bytecode counter by *delta*. Checks for interrupts.

Added in version 3.11.

JUMP_BACKWARD_NO_INTERRUPT(*delta*)

Decrements bytecode counter by *delta*. Does not check for interrupts.

Added in version 3.11.

POP_JUMP_IF_TRUE(*delta*)

If `STACK[-1]` is true, increments the bytecode counter by *delta*. `STACK[-1]` is popped.

Changed in version 3.11: The oparg is now a relative delta rather than an absolute target. This opcode is a pseudo-instruction, replaced in final bytecode by the directed versions (forward/backward).

Changed in version 3.12: This is no longer a pseudo-instruction.

Changed in version 3.13: This instruction now requires an exact **`bool`** operand.

POP_JUMP_IF_FALSE(*delta*)

If `STACK[-1]` is false, increments the bytecode counter by *delta*. `STACK[-1]` is popped.

Changed in version 3.11: The oparg is now a relative delta rather than an absolute target. This opcode is a pseudo-instruction, replaced in final bytecode by the directed versions (forward/backward).

Changed in version 3.12: This is no longer a pseudo-instruction.

Changed in version 3.13: This instruction now requires an exact **`bool`** operand.

POP_JUMP_IF_NOT_NONE(*delta*)

If `STACK[-1]` is not `None`, increments the bytecode counter by *delta*. `STACK[-1]` is popped.

Added in version 3.11.

Changed in version 3.12: This is no longer a pseudo-instruction.

POP_JUMP_IF_NONE(*delta*)

If `STACK[-1]` is `None`, increments the bytecode counter by *delta*. `STACK[-1]` is popped.

Added in version 3.11.

Changed in version 3.12: This is no longer a pseudo-instruction.

FOR_ITER(*delta*)

`STACK[-1]` is an iterator. Call its **`__next__()`** method. If this yields a new value, push it on the stack (leaving the iterator below it). If the iterator indicates it is exhausted then the byte code counter is incremented by *delta*.

Changed in version 3.12: Up until 3.11 the iterator was popped when it was exhausted.

LOAD_GLOBAL(*namei*)

Loads the global named `co_names[namei>>1]` onto the stack.

Changed in version 3.11: If the low bit of `namei` is set, then a `NULL` is pushed to the stack before the global variable.

LOAD_FAST(*var_num*)

Pushes a reference to the local `co_varnames[var_num]` onto the stack.

Changed in version 3.12: This opcode is now only used in situations where the local variable is guaranteed to be initialized. It cannot raise **`UnboundLocalError`**.

LOAD_FAST_BORROW(*var_num*)

Pushes a borrowed reference to the local `co_varnames[var_num]` onto the stack.

Added in version 3.14.

LOAD_FAST_LOAD_FAST(*var_nums*)

Pushes references to `co_varnames[var_nums >> 4]` and `co_varnames[var_nums & 15]` onto the stack.

Added in version 3.13.

LOAD_FAST_BORROW_LOAD_FAST_BORROW(*var_nums*)

Pushes borrowed references to `co_varnames[var_nums >> 4]` and `co_varnames[var_nums & 15]` onto the stack.

Added in version 3.14.

LOAD_FAST_CHECK(*var_num*)

Pushes a reference to the local `co_varnames[var_num]` onto the stack, raising an **`UnboundLocalError`** if the local variable has not been initialized.

Added in version 3.12.

LOAD_FAST_AND_CLEAR(*var_num*)

Pushes a reference to the local `co_varnames[var_num]` onto the stack (or pushes `NULL` onto the stack if the local variable has not been initialized) and sets `co_varnames[var_num]` to `NULL`.

Added in version 3.12.

STORE_FAST(*var_num*)

Stores `STACK.pop()` into the local `co_varnames[var_num]`.

STORE_FAST_STORE_FAST(*var_nums*)

Stores `STACK[-1]` into `co_varnames[var_nums >> 4]` and `STACK[-2]` into `co_varnames[var_nums & 15]`.

Added in version 3.13.

STORE_FAST_LOAD_FAST(*var_nums*)

Stores `STACK.pop()` into the local `co_varnames[var_nums >> 4]` and pushes a reference to the local `co_varnames[var_nums & 15]` onto the stack.

Added in version 3.13.

DELETE_FAST(*var_num*)

Deletes local `co_varnames[var_num]`.

MAKE_CELL(*i*)

Creates a new cell in slot `i`. If that slot is nonempty then that value is stored into the new cell.

Added in version 3.11.

LOAD_DEREF(*i*)

Loads the cell contained in slot `i` of the “fast locals” storage. Pushes a reference to the object the cell contains on the stack.

Changed in version 3.11: `i` is no longer offset by the length of **`co_varnames`**.

LOAD_FROM_DICT_OR_DEREF(*i*)

Pops a mapping off the stack and looks up the name associated with slot `i` of the “fast locals” storage in this mapping. If the name is not found there, loads it from the cell contained in slot `i`, similar to **`LOAD_DEREF`**. This is used for loading closure variables in class bodies (which previously used **`LOAD_CLASSDEREF`**) and in annotation scopes within class bodies.

Added in version 3.12.

STORE_DEREF(*i*)

Stores `STACK.pop()` into the cell contained in slot `i` of the “fast locals” storage.

Changed in version 3.11: `i` is no longer offset by the length of **`co_varnames`**.

DELETE_DEREF(*i*)

Empties the cell contained in slot `i` of the “fast locals” storage. Used by the **`del`** statement.

Added in version 3.2.

Changed in version 3.11: `i` is no longer offset by the length of **`co_varnames`**.

COPY_FREE_VARS(*n*)

Copies the `n` free (closure) variables from the closure into the frame. Removes the need for special code on the caller's side when calling closures.

Added in version 3.11.

RAISE_VARARGS(*argc*)

Raises an exception using one of the 3 forms of the `raise` statement, depending on the value of *argc*:

- 0: `raise` (re-raise previous exception)
- 1: `raise STACK[-1]` (raise exception instance or type at `STACK[-1]`)
- 2: `raise STACK[-2] from STACK[-1]` (raise exception instance or type at `STACK[-2]` with `__cause__` set to `STACK[-1]`)

CALL(*argc*)

Calls a callable object with the number of arguments specified by `argc`. On the stack are (in ascending order):

- The callable
- `self` or `NULL`
- The remaining positional arguments

`argc` is the total of the positional arguments, excluding `self`.

`CALL` pops all arguments and the callable object off the stack, calls the callable object with those arguments, and pushes the return value returned by the callable object.

Added in version 3.11.

Changed in version 3.13: The callable now always appears at the same position on the stack.

Changed in version 3.13: Calls with keyword arguments are now handled by [`CALL_KW`](#).

CALL_KW(*argc*)

Calls a callable object with the number of arguments specified by `argc`, including one or more named arguments. On the stack are (in ascending order):

- The callable
- `self` or `NULL`
- The remaining positional arguments
- The named arguments
- A `tuple` of keyword argument names

`argc` is the total of the positional and named arguments, excluding `self`. The length of the tuple of keyword argument names is the number of named arguments.

`CALL_KW` pops all arguments, the keyword names, and the callable object off the stack, calls the callable object with those arguments, and pushes the return value returned by the callable object.

Added in version 3.13.

CALL_FUNCTION_EX(*flags*)

Calls a callable object with variable set of positional and keyword arguments. If the lowest bit of *flags* is set, the top of the stack contains a mapping object containing additional keyword arguments. Before the callable is called, the mapping object and iterable object are each “unpacked” and their contents passed in as keyword and positional arguments respectively. `CALL_FUNCTION_EX` pops all arguments and the callable object off the stack, calls the callable object with those arguments, and pushes the return value returned by the callable object.

Added in version 3.6.

PUSH_NULL

Pushes a `NULL` to the stack. Used in the call sequence to match the `NULL` pushed by **`LOAD_METHOD`** for non-method calls.

Added in version 3.11.

MAKE_FUNCTION

Pushes a new function object on the stack built from the code object at `STACK[-1]`.

Changed in version 3.10: Flag value `0x04` is a tuple of strings instead of dictionary

Changed in version 3.11: Qualified name at `STACK[-1]` was removed.

Changed in version 3.13: Extra function attributes on the stack, signaled by oparg flags, were removed. They now use **`SET_FUNCTION_ATTRIBUTE`**.

SET_FUNCTION_ATTRIBUTE(*flag*)

Sets an attribute on a function object. Expects the function at `STACK[-1]` and the attribute value to set at `STACK[-2]`; consumes both and leaves the function at `STACK[-1]`. The flag determines which attribute to set:

- `0x01` a tuple of default values for positional-only and positional-or-keyword parameters in positional order

- `0x02` a dictionary of keyword-only parameters’ default values
- `0x04` a tuple of strings containing parameters’ annotations
- `0x08` a tuple containing cells for free variables, making a closure
- `0x10` the annotate function for the function object

Added in version 3.13.

Changed in version 3.14: Added `0x10` to indicate the annotate function for the function object.

BUILD_SLICE(*argc*)

Pushes a slice object on the stack. *argc* must be 2 or 3. If it is 2, implements:

```
end = STACK.pop()
start = STACK.pop()
STACK.append(slice(start, end))
```

if it is 3, implements:

```
step = STACK.pop()
end = STACK.pop()
start = STACK.pop()
STACK.append(slice(start, end, step))
```

See the **`slice()`** built-in function for more information.

EXTENDED_ARG(*ext*)

Prefixes any opcode which has an argument too big to fit into the default one byte. *ext* holds an additional byte which act as higher bits in the argument. For each opcode, at most three prefixal `EXTENDED_ARG` are allowed, forming an argument from two-byte to four-byte.

CONVERT_VALUE(*oparg*)

Convert value to a string, depending on `oparg`:

```
value = STACK.pop()
result = func(value)
STACK.append(result)
```

- `oparg == 1`: call **`str()`** on *value*
- `oparg == 2`: call **`repr()`** on *value*
- `oparg == 3`: call **`ascii()`** on *value*

Used for implementing formatted string literals (f-strings).

Added in version 3.13.

FORMAT_SIMPLE

Formats the value on top of stack:

```
value = STACK.pop()
result = value.__format__("")
STACK.append(result)
```

Used for implementing formatted string literals (f-strings).

Added in version 3.13.

FORMAT_WITH_SPEC

Formats the given value with the given format spec:

```
spec = STACK.pop()
value = STACK.pop()
result = value.__format__(spec)
STACK.append(result)
```

Used for implementing formatted string literals (f-strings).

Added in version 3.13.

MATCH_CLASS(*count*)

`STACK[-1]` is a tuple of keyword attribute names, `STACK[-2]` is the class being matched against, and `STACK[-3]` is the match subject. *count* is the number of positional sub-patterns.

Pop `STACK[-1]`, `STACK[-2]`, and `STACK[-3]`. If `STACK[-3]` is an instance of `STACK[-2]` and has the positional and keyword attributes required by *count* and `STACK[-1]`, push a tuple of extracted attributes. Otherwise, push `None`.

Added in version 3.10.

Changed in version 3.11: Previously, this instruction also pushed a boolean value indicating success (`True`) or failure (`False`).

RESUME(*context*)

A no-op. Performs internal tracing, debugging and optimization checks.

The `context` operand consists of two parts. The lowest two bits indicate where the `RESUME` occurs:

- `0` The start of a function, which is neither a generator, coroutine nor an async generator
- `1` After a `yield` expression
- `2` After a `yield from` expression
- `3` After an `await` expression

The next bit is `1` if the RESUME is at except-depth `1`, and `0` otherwise.

Added in version 3.11.

Changed in version 3.13: The oparg value changed to include information about except-depth

RETURN_GENERATOR

Create a generator, coroutine, or async generator from the current frame. Used as first opcode of in code object for the above mentioned callables. Clear the current frame and return the newly created generator.

Added in version 3.11.

SEND(*delta*)

Equivalent to `STACK[-1] = STACK[-2].send(STACK[-1])`. Used in `yield from` and `await` statements.

If the call raises **`StopIteration`**, pop the top value from the stack, push the exception’s `value` attribute, and increment the bytecode counter by *delta*.

Added in version 3.11.

HAVE_ARGUMENT

This is not really an opcode. It identifies the dividing line between opcodes in the range [0,255] which don’t use their argument and those that do (`< HAVE_ARGUMENT` and `>= HAVE_ARGUMENT`, respectively).

If your application uses pseudo instructions or specialized instructions, use the **`hasarg`** collection instead.

Changed in version 3.6: Now every instruction has an argument, but opcodes `< HAVE_ARGUMENT` ignore it. Before, only opcodes `>= HAVE_ARGUMENT` had an argument.

Changed in version 3.12: Pseudo instructions were added to the **`dis`** module, and for them it is not true that comparison with `HAVE_ARGUMENT` indicates whether they use their arg.

Deprecated since version 3.13: Use **`hasarg`** instead.

CALL_INTRINSIC_1

Calls an intrinsic function with one argument. Passes `STACK[-1]` as the argument and sets `STACK[-1]` to the result. Used to implement functionality that is not performance critical.

The operand determines which intrinsic function is called:

Operand	**Description**
`INTRINSIC_1_INVALID`	Not valid
`INTRINSIC_PRINT`	Prints the argument to standard out. Used in the REPL.
`INTRINSIC_IMPORT_STAR`	Performs `import *` for the named module.
`INTRINSIC_STOPITERATION_ERROR`	Extracts the return value from a `StopIteration` exception.
`INTRINSIC_ASYNC_GEN_WRAP`	Wraps an async generator value
`INTRINSIC_UNARY_POSITIVE`	Performs the unary `+` operation
`INTRINSIC_LIST_TO_TUPLE`	Converts a list to a tuple
`INTRINSIC_TYPEVAR`	Creates a `typing.TypeVar`
`INTRINSIC_PARAMSPEC`	Creates a `typing.ParamSpec`

Operand	**Description**
`INTRINSIC_TYPEVARTUPLE`	Creates a `typing.TypeVarTuple`
`INTRINSIC_SUBSCRIPT_GENERIC`	Returns `typing.Generic` subscripted with the argument
`INTRINSIC_TYPEALIAS`	Creates a `typing.TypeAliasType`; used in the `type` statement. The argument is a tuple of the type alias’s name, type parameters, and value.

Added in version 3.12.

CALL_INTRINSIC_2

Calls an intrinsic function with two arguments. Used to implement functionality that is not performance critical:

```
arg2 = STACK.pop()
arg1 = STACK.pop()
result = intrinsic2(arg1, arg2)
STACK.append(result)
```

The operand determines which intrinsic function is called:

Operand	**Description**
`INTRINSIC_2_INVALID`	Not valid
`INTRINSIC_PREP_RERAISE_STAR`	Calculates the `ExceptionGroup` to raise from a `try-except*`.

Operand	Description
`INTRINSIC_TYPEVAR_WITH_BOUND`	Creates a `typing.TypeVar` with a bound.
`INTRINSIC_TYPEVAR_WITH_CONSTRAINTS`	Creates a `typing.TypeVar` with constraints.
`INTRINSIC_SET_FUNCTION_TYPE_PARAMS`	Sets the `__type_params__` attribute of a function.

Added in version 3.12.

LOAD_SPECIAL

Performs special method lookup on `STACK[-1]`. If `type(STACK[-1]).__xxx__` is a method, leave `type(STACK[-1]).__xxx__; STACK[-1]` on the stack. If `type(STACK[-1]).__xxx__` is not a method, leave `STACK[-1].__xxx__; NULL` on the stack.

Added in version 3.14.

Pseudo-instructions

These opcodes do not appear in Python bytecode. They are used by the compiler but are replaced by real opcodes or removed before bytecode is generated.

SETUP_FINALLY(*target*)

Set up an exception handler for the following code block. If an exception occurs, the value stack level is restored to its current state and control is transferred to the exception handler at `target`.

SETUP_CLEANUP(*target*)

Like `SETUP_FINALLY`, but in case of an exception also pushes the last instruction (`lasti`) to the stack so that `RERAISE` can restore it. If an exception occurs, the value stack level and the last instruction on the frame are restored to their current state, and control is transferred to the exception handler at `target`.

SETUP_WITH(*target*)

Like `SETUP_CLEANUP`, but in case of an exception one more item is popped from the stack before control is transferred to the exception handler at `target`.

This variant is used in **`with`** and **`async with`** constructs, which push the return value of the context manager's **`__enter__()`** or **`__aenter__()`** to the stack.

POP_BLOCK

Marks the end of the code block associated with the last `SETUP_FINALLY`, `SETUP_CLEANUP` or `SETUP_WITH`.

LOAD_CONST_IMMORTAL(*consti*)

Works as **`LOAD_CONST`**, but is more efficient for immortal objects.

JUMP
JUMP_NO_INTERRUPT

Undirected relative jump instructions which are replaced by their directed (forward/backward) counterparts by the assembler.

JUMP_IF_TRUE
JUMP_IF_FALSE

Conditional jumps which do not impact the stack. Replaced by the sequence `COPY 1`, `TO_BOOL`, `POP_JUMP_IF_TRUE/FALSE`.

LOAD_CLOSURE(*i*)

Pushes a reference to the cell contained in slot `i` of the "fast locals" storage.

Note that `LOAD_CLOSURE` is replaced with `LOAD_FAST` in the assembler.

Changed in version 3.13: This opcode is now a pseudo-instruction.

Opcode collections

These collections are provided for automatic introspection of bytecode instructions:

Changed in version 3.12: The collections now contain pseudo instructions and instrumented instructions as well. These are opcodes with values `>= MIN_PSEUDO_OPCODE` and `>= MIN_INSTRUMENTED_OPCODE`.

dis.opname

Sequence of operation names, indexable using the bytecode.

dis.opmap

Dictionary mapping operation names to bytecodes.

dis.cmp_op

Sequence of all compare operation names.

dis.hasarg

Sequence of bytecodes that use their argument.

Added in version 3.12.

dis.hasconst

Sequence of bytecodes that access a constant.

dis.hasfree

Sequence of bytecodes that access a free (closure) variable. ‘free’ in this context refers to names in the current scope that are referenced by inner scopes or names in outer scopes that are referenced from this scope. It does *not* include references to global or builtin scopes.

dis.hasname

Sequence of bytecodes that access an attribute by name.

dis.hasjump

Sequence of bytecodes that have a jump target. All jumps are relative.

Added in version 3.13.

dis.haslocal

Sequence of bytecodes that access a local variable.

dis.hascompare

Sequence of bytecodes of Boolean operations.

dis.hasexc

Sequence of bytecodes that set an exception handler.

Added in version 3.12.

dis.hasjrel

Sequence of bytecodes that have a relative jump target.

Deprecated since version 3.13: All jumps are now relative. Use **`hasjump`**.

dis.hasjabs

Sequence of bytecodes that have an absolute jump target.

Deprecated since version 3.13: All jumps are now relative. This list is empty.

`pickletools` — Tools for pickle developers

Source code: Lib/pickletools.py
[https://github.com/python/cpython/tree/3.14/Lib/pickletools.py]

This module contains various constants relating to the intimate details of the **`pickle`** module, some lengthy comments about the implementation, and a few useful functions for analyzing pickled data. The contents of this module are useful for Python core developers who are working on the **`pickle`**; ordinary users of the **`pickle`** module probably won't find the **`pickletools`** module relevant.

Command-line usage

Added in version 3.2.

When invoked from the command line, `python -m pickletools` will disassemble the contents of one or more pickle files. Note that if you want to see the Python object stored in the pickle rather than the details of pickle format, you may want to use `-m pickle` instead. However, when the pickle file that you want to examine comes from an untrusted source, `-m pickletools` is a safer option because it does not execute pickle bytecode.

For example, with a tuple `(1, 2)` pickled in file `x.pickle`:

```
$ python -m pickle x.pickle
(1, 2)
```

```
$ python -m pickletools x.pickle
    0: \x80 PROTO      3
    2: K    BININT1    1
    4: K    BININT1    2
    6: \x86 TUPLE2
    7: q    BINPUT     0
    9: .    STOP
highest protocol among opcodes = 2
```

Command-line options

-a, --annotate

Annotate each line with a short opcode description.

-o, --output=<file>

Name of a file where the output should be written.

-l, --indentlevel=<num>

The number of blanks by which to indent a new MARK level.

-m, --memo

When multiple objects are disassembled, preserve memo between disassemblies.

-p, --preamble=<preamble>

When more than one pickle file is specified, print given preamble before each disassembly.

pickle_file

A pickle file to read, or `-` to indicate reading from standard input.

Programmatic interface

pickletools.dis(*pickle*, *out=None*, *memo=None*, *indentlevel=4*, *annotate=0*)

Outputs a symbolic disassembly of the pickle to the file-like object *out*, defaulting to `sys.stdout`. *pickle* can be a string or a file-like object. *memo* can be a Python dictionary that will be used as the pickle's memo; it can be used to perform disassemblies across multiple pickles created by the same pickler. Successive levels, indicated by `MARK` opcodes in the stream, are indented by *indentlevel* spaces. If a nonzero value is given to *annotate*, each opcode in the output is annotated with a short description. The value of *annotate* is used as a hint for the column where annotation should start.

Changed in version 3.2: Added the *annotate* parameter.

pickletools.genops(*pickle*)

Provides an iterator over all of the opcodes in a pickle, returning a sequence of `(opcode, arg, pos)` triples. *opcode* is an instance of an **`OpcodeInfo`** class; *arg* is the decoded value, as a Python object, of the opcode's argument; *pos* is the position at which this opcode is located. *pickle* can be a string or a file-like object.

pickletools.optimize(*picklestring*)

Returns a new equivalent pickle string after eliminating unused `PUT` opcodes. The optimized pickle is shorter, takes less transmission time, requires less storage space, and unpickles more efficiently.

MS Windows Specific Services

This chapter describes modules that are only available on MS Windows platforms.

- **`msvcrt`** — Useful routines from the MS VC++ runtime
  - File Operations
  - Console I/O
  - Other Functions
- **`winreg`** — Windows registry access
  - Functions
  - Constants
    - HKEY_* Constants
    - Access Rights
      - 64-bit Specific
    - Value Types
  - Registry Handle Objects
- **`winsound`** — Sound-playing interface for Windows

`msvcrt` — Useful routines from the MS VC++ runtime

These functions provide access to some useful capabilities on Windows platforms. Some higher-level modules use these functions to build the Windows implementations of their services. For example, the `getpass` module uses this in the implementation of the `getpass()` function.

Further documentation on these functions can be found in the Platform API documentation.

The module implements both the normal and wide char variants of the console I/O api. The normal API deals only with ASCII characters and is of limited use for internationalized applications. The wide char API should be used where ever possible.

Availability: Windows.

Changed in version 3.3: Operations in this module now raise `OSError` where `IOError` was raised.

File Operations

msvcrt.locking(*fd*, *mode*, *nbytes*)

Lock part of a file based on file descriptor *fd* from the C runtime. Raises `OSError` on failure. The locked region of the file extends from the current file position for *nbytes*

bytes, and may continue beyond the end of the file. *mode* must be one of the **`LK_*`** constants listed below. Multiple regions in a file may be locked at the same time, but may not overlap. Adjacent regions are not merged; they must be unlocked individually.

Raises an auditing event `msvcrt.locking` with arguments `fd`, `mode`, `nbytes`.

msvcrt.LK_LOCK
msvcrt.LK_RLCK

Locks the specified bytes. If the bytes cannot be locked, the program immediately tries again after 1 second. If, after 10 attempts, the bytes cannot be locked, **`OSError`** is raised.

msvcrt.LK_NBLCK
msvcrt.LK_NBRLCK

Locks the specified bytes. If the bytes cannot be locked, **`OSError`** is raised.

msvcrt.LK_UNLCK

Unlocks the specified bytes, which must have been previously locked.

msvcrt.setmode(*fd*, *flags*)

Set the line-end translation mode for the file descriptor *fd*. To set it to text mode, *flags* should be **`os.O_TEXT`**; for binary, it should be **`os.O_BINARY`**.

msvcrt.open_osfhandle(*handle*, *flags*)

Create a C runtime file descriptor from the file handle *handle*. The *flags* parameter should be a bitwise OR of **`os.O_APPEND`**, **`os.O_RDONLY`**, **`os.O_TEXT`** and **`os.O_NOINHERIT`**. The returned file descriptor may be used as a parameter to **`os.fdopen()`** to create a file object.

The file descriptor is inheritable by default. Pass **`os.O_NOINHERIT`** flag to make it non inheritable.

Raises an auditing event `msvcrt.open_osfhandle` with arguments `handle`, `flags`.

msvcrt.get_osfhandle(*fd*)

Return the file handle for the file descriptor *fd*. Raises **`OSError`** if *fd* is not recognized.

Raises an auditing event `msvcrt.get_osfhandle` with argument `fd`.

Console I/O

msvcrt.kbhit()

Returns a nonzero value if a keypress is waiting to be read. Otherwise, return 0.

msvcrt.getch()

Read a keypress and return the resulting character as a byte string. Nothing is echoed to the console. This call will block if a keypress is not already available, but will not wait for `Enter` to be pressed. If the pressed key was a special function key, this will return `'\000'` or `'\xe0'`; the next call will return the keycode. The `Control-C` keypress cannot be read with this function.

msvcrt.getwch()

Wide char variant of **`getch()`**, returning a Unicode value.

msvcrt.getche()

Similar to **`getch()`**, but the keypress will be echoed if it represents a printable character.

msvcrt.getwche()

Wide char variant of `getche()`, returning a Unicode value.

msvcrt.putch(*char*)

Print the byte string *char* to the console without buffering.

msvcrt.putwch(*unicode_char*)

Wide char variant of `putch()`, accepting a Unicode value.

msvcrt.ungetch(*char*)

Cause the byte string *char* to be “pushed back” into the console buffer; it will be the next character read by `getch()` or `getche()`.

msvcrt.ungetwch(*unicode_char*)

Wide char variant of `ungetch()`, accepting a Unicode value.

Other Functions

msvcrt.heapmin()

Force the **`malloc()`** heap to clean itself up and return unused blocks to the operating system. On failure, this raises `OSError`.

msvcrt.set_error_mode(*mode*)

Changes the location where the C runtime writes an error message for an error that might end the program. *mode* must be one of the **`OUT_*`** constants listed below or `REPORT_ERRMODE`. Returns the old setting or -1 if an error occurs. Only available in debug build of Python.

msvcrt.OUT_TO_DEFAULT

Error sink is determined by the app’s type. Only available in debug build of Python.

msvcrt.OUT_TO_STDERR

Error sink is a standard error. Only available in debug build of Python.

msvcrt.OUT_TO_MSGBOX

Error sink is a message box. Only available in debug build of Python.

msvcrt.REPORT_ERRMODE

Report the current error mode value. Only available in debug build of Python.

msvcrt.CrtSetReportMode(*type*, *mode*)

Specifies the destination or destinations for a specific report type generated by **`_CrtDbgReport()`** in the MS VC++ runtime. *type* must be one of the **`CRT_*`** constants listed below. *mode* must be one of the **`CRTDBG_*`** constants listed below. Only available in debug build of Python.

msvcrt.CrtSetReportFile(*type*, *file*)

After you use **`CrtSetReportMode()`** to specify **`CRTDBG_MODE_FILE`**, you can specify the file handle to receive the message text. *type* must be one of the **`CRT_*`** constants listed below. *file* should be the file handle your want specified. Only available in debug build of Python.

msvcrt.CRT_WARN

Warnings, messages, and information that doesn’t need immediate attention.

msvcrt.CRT_ERROR

Errors, unrecoverable problems, and issues that require immediate attention.

msvcrt.CRT_ASSERT

Assertion failures.

msvcrt.CRTDBG_MODE_DEBUG

Writes the message to the debugger’s output window.

msvcrt.CRTDBG_MODE_FILE

Writes the message to a user-supplied file handle. `CrtSetReportFile()` should be called to define the specific file or stream to use as the destination.

msvcrt.CRTDBG_MODE_WNDW

Creates a message box to display the message along with the `Abort`, `Retry`, and `Ignore` buttons.

msvcrt.CRTDBG_REPORT_MODE

Returns current *mode* for the specified *type*.

msvcrt.CRT_ASSEMBLY_VERSION

The CRT Assembly version, from the `crtassem.h` header file.

msvcrt.VC_ASSEMBLY_PUBLICKEYTOKEN

The VC Assembly public key token, from the `crtassem.h` header file.

msvcrt.LIBRARIES_ASSEMBLY_NAME_PREFIX

The Libraries Assembly name prefix, from the `crtassem.h` header file.

`winreg` — Windows registry access

These functions expose the Windows registry API to Python. Instead of using an integer as the registry handle, a handle object is used to ensure that the handles are closed correctly, even if the programmer neglects to explicitly close them.

Availability: Windows.

Changed in version 3.3: Several functions in this module used to raise a `WindowsError`, which is now an alias of `OSError`.

Functions

This module offers the following functions:

winreg.CloseKey(*hkey*)

Closes a previously opened registry key. The *hkey* argument specifies a previously opened key.

> **Note**
>
> If *hkey* is not closed using this method (or via `hkey.Close()`), it is closed when the *hkey* object is destroyed by Python.

winreg.ConnectRegistry(*computer_name*, *key*)

Establishes a connection to a predefined registry handle on another computer, and returns a handle object.

computer_name is the name of the remote computer, of the form `r"\\computername"`. If `None`, the local computer is used.

key is the predefined handle to connect to.

The return value is the handle of the opened key. If the function fails, an **`OSError`** exception is raised.

Raises an auditing event `winreg.ConnectRegistry` with arguments `computer_name`, `key`.

Changed in version 3.3: See above.

winreg.CreateKey(*key*, *sub_key*)

Creates or opens the specified key, returning a handle object.

key is an already open key, or one of the predefined HKEY_* constants.

sub_key is a string that names the key this method opens or creates.

If *key* is one of the predefined keys, *sub_key* may be `None`. In that case, the handle returned is the same key handle passed in to the function.

If the key already exists, this function opens the existing key.

The return value is the handle of the opened key. If the function fails, an **`OSError`** exception is raised.

Raises an auditing event `winreg.CreateKey` with arguments `key`, `sub_key`, `access`.

Raises an auditing event `winreg.OpenKey/result` with argument `key`.

Changed in version 3.3: See above.

winreg.CreateKeyEx(*key*, *sub_key*, *reserved=0*, *access=KEY_WRITE*)

Creates or opens the specified key, returning a handle object.

key is an already open key, or one of the predefined HKEY_* constants.

sub_key is a string that names the key this method opens or creates.

reserved is a reserved integer, and must be zero. The default is zero.

access is an integer that specifies an access mask that describes the desired security access for the key. Default is **`KEY_WRITE`**. See Access Rights for other allowed values.

If *key* is one of the predefined keys, *sub_key* may be `None`. In that case, the handle returned is the same key handle passed in to the function.

If the key already exists, this function opens the existing key.

The return value is the handle of the opened key. If the function fails, an **`OSError`** exception is raised.

Raises an auditing event `winreg.CreateKey` with arguments `key`, `sub_key`, `access`.

Raises an auditing event `winreg.OpenKey/result` with argument `key`.

Added in version 3.2.

Changed in version 3.3: See above.

winreg.DeleteKey(*key*, *sub_key*)

Deletes the specified key.

key is an already open key, or one of the predefined HKEY_* constants.

sub_key is a string that must be a subkey of the key identified by the *key* parameter. This value must not be `None`, and the key may not have subkeys.

This method can not delete keys with subkeys.

If the method succeeds, the entire key, including all of its values, is removed. If the method fails, an **`OSError`** exception is raised.

Raises an auditing event `winreg.DeleteKey` with arguments `key`, `sub_key`, `access`.

Changed in version 3.3: See above.

winreg.DeleteKeyEx(*key*, *sub_key*, *access=KEY_WOW64_64KEY*, *reserved=0*)

Deletes the specified key.

key is an already open key, or one of the predefined HKEY_* constants.

sub_key is a string that must be a subkey of the key identified by the *key* parameter. This value must not be

`None`, and the key may not have subkeys.

reserved is a reserved integer, and must be zero. The default is zero.

access is an integer that specifies an access mask that describes the desired security access for the key. Default is **`KEY_WOW64_64KEY`**. On 32-bit Windows, the WOW64 constants are ignored. See Access Rights for other allowed values.

This method can not delete keys with subkeys.

If the method succeeds, the entire key, including all of its values, is removed. If the method fails, an **`OSError`** exception is raised.

On unsupported Windows versions, **`NotImplementedError`** is raised.

Raises an auditing event `winreg.DeleteKey` with arguments `key`, `sub_key`, `access`.

Added in version 3.2.

Changed in version 3.3: See above.

winreg.DeleteValue(*key*, *value*)

Removes a named value from a registry key.

key is an already open key, or one of the predefined HKEY_* constants.

value is a string that identifies the value to remove.

Raises an auditing event `winreg.DeleteValue` with arguments `key`, `value`.

winreg.EnumKey(*key*, *index*)

Enumerates subkeys of an open registry key, returning a string.

key is an already open key, or one of the predefined HKEY_* constants.

index is an integer that identifies the index of the key to retrieve.

The function retrieves the name of one subkey each time it is called. It is typically called repeatedly until an **`OSError`** exception is raised, indicating, no more values are available.

Raises an auditing event `winreg.EnumKey` with arguments `key`, `index`.

Changed in version 3.3: See above.

winreg.EnumValue(*key*, *index*)

Enumerates values of an open registry key, returning a tuple.

key is an already open key, or one of the predefined HKEY_* constants.

index is an integer that identifies the index of the value to retrieve.

The function retrieves the name of one subkey each time it is called. It is typically called repeatedly, until an **`OSError`** exception is raised, indicating no more values.

The result is a tuple of 3 items:

Index	Meaning
`0`	A string that identifies the value name
`1`	An object that holds the value data, and whose type depends on the underlying registry type
`2`	An integer that identifies the type of the value data (see table in docs for `SetValueEx()`)

Raises an auditing event `winreg.EnumValue` with arguments `key`, `index`.

Changed in version 3.3: See above.

winreg.ExpandEnvironmentStrings(*str*)

Expands environment variable placeholders `%NAME%` in strings like **`REG_EXPAND_SZ`**:

```
>>> ExpandEnvironmentStrings('%windir%')
'C:\\Windows'
```

Raises an auditing event `winreg.ExpandEnvironmentStrings` with argument `str`.

winreg.FlushKey(*key*)

Writes all the attributes of a key to the registry.

key is an already open key, or one of the predefined HKEY_* constants.

It is not necessary to call **`FlushKey()`** to change a key. Registry changes are flushed to disk by the registry using its lazy flusher. Registry changes are also flushed to disk at system shutdown. Unlike **`CloseKey()`**, the **`FlushKey()`** method returns only when all the data has been written to the registry. An application should only call `FlushKey()` if it requires absolute certainty that registry changes are on disk.

Note

If you don’t know whether a `FlushKey()` call is required, it probably isn’t.

winreg.LoadKey(*key*, *sub_key*, *file_name*)

Creates a subkey under the specified key and stores registration information from a specified file into that subkey.

key is a handle returned by `ConnectRegistry()` or one of the constants `HKEY_USERS` or `HKEY_LOCAL_MACHINE`.

sub_key is a string that identifies the subkey to load.

file_name is the name of the file to load registry data from. This file must have been created with the `SaveKey()` function. Under the file allocation table (FAT) file system, the filename may not have an extension.

A call to `LoadKey()` fails if the calling process does not have the **`SE_RESTORE_PRIVILEGE`** privilege. Note that privileges are different from permissions – see the RegLoadKey documentation [https://msdn.microsoft.com/en-us/library/ms724889%28v=VS.85%29.aspx] for more details.

If *key* is a handle returned by `ConnectRegistry()`, then the path specified in *file_name* is relative to the remote computer.

Raises an auditing event `winreg.LoadKey` with arguments `key`, `sub_key`, `file_name`.

winreg.OpenKey(*key*, *sub_key*, *reserved=0*, *access=KEY_READ*)
winreg.OpenKeyEx(*key*, *sub_key*, *reserved=0*, *access=KEY_READ*)

Opens the specified key, returning a handle object.

key is an already open key, or one of the predefined HKEY_* constants.

sub_key is a string that identifies the sub_key to open.

reserved is a reserved integer, and must be zero. The default is zero.

access is an integer that specifies an access mask that describes the desired security access for the key. Default is **`KEY_READ`**. See Access Rights for other allowed values.

The result is a new handle to the specified key.

If the function fails, **`OSError`** is raised.

Raises an auditing event `winreg.OpenKey` with arguments `key`, `sub_key`, `access`.

Raises an auditing event `winreg.OpenKey/result` with argument `key`.

Changed in version 3.2: Allow the use of named arguments.

Changed in version 3.3: See above.

winreg.QueryInfoKey(*key*)

Returns information about a key, as a tuple.

key is an already open key, or one of the predefined HKEY_* constants.

The result is a tuple of 3 items:

Index	Meaning
`0`	An integer giving the number of sub keys this key has.
`1`	An integer giving the number of values this key has.
`2`	An integer giving when the key was last modified (if available) as 100's of nanoseconds since Jan 1, 1601.

Raises an auditing event `winreg.QueryInfoKey` with argument `key`.

winreg.QueryValue(*key*, *sub_key*)

Retrieves the unnamed value for a key, as a string.

key is an already open key, or one of the predefined HKEY_* constants.

sub_key is a string that holds the name of the subkey with which the value is associated. If this parameter is `None` or empty, the function retrieves the value set by the **`SetValue()`** method for the key identified by *key*.

Values in the registry have name, type, and data components. This method retrieves the data for a key's first value that has a `NULL` name. But the underlying API call doesn't return the type, so always use **`QueryValueEx()`** if possible.

Raises an auditing event `winreg.QueryValue` with arguments `key`, `sub_key`, `value_name`.

winreg.QueryValueEx(*key*, *value_name*)

Retrieves the type and data for a specified value name associated with an open registry key.

key is an already open key, or one of the predefined HKEY_* constants.

value_name is a string indicating the value to query.

The result is a tuple of 2 items:

Index	**Meaning**
`0`	The value of the registry item.
`1`	An integer giving the registry type for this value (see table in docs for **`SetValueEx()`**)

Raises an auditing event `winreg.QueryValue` with arguments `key`, `sub_key`, `value_name`.

winreg.SaveKey(*key*, *file_name*)

Saves the specified key, and all its subkeys to the specified file.

key is an already open key, or one of the predefined HKEY_* constants.

file_name is the name of the file to save registry data to. This file cannot already exist. If this filename includes an extension, it cannot be used on file allocation table (FAT) file systems by the **`LoadKey()`** method.

If *key* represents a key on a remote computer, the path described by *file_name* is relative to the remote computer. The caller of this method must possess the **SeBackupPrivilege** security privilege. Note that privileges are different than permissions – see the Conflicts Between User Rights and Permissions documentation [https://msdn.microsoft.com/en-us/library/ms724878%28v=VS.85%29.aspx] for more details.

This function passes `NULL` for *security_attributes* to the API.

Raises an auditing event `winreg.SaveKey` with arguments `key`, `file_name`.

winreg.SetValue(*key*, *sub_key*, *type*, *value*)

Associates a value with a specified key.

key is an already open key, or one of the predefined HKEY_* constants.

sub_key is a string that names the subkey with which the value is associated.

type is an integer that specifies the type of the data. Currently this must be **`REG_SZ`**, meaning only strings are supported. Use the **`SetValueEx()`** function for support for other data types.

value is a string that specifies the new value.

If the key specified by the *sub_key* parameter does not exist, the SetValue function creates it.

Value lengths are limited by available memory. Long values (more than 2048 bytes) should be stored as files with the filenames stored in the configuration registry. This helps the registry perform efficiently.

The key identified by the *key* parameter must have been opened with **`KEY_SET_VALUE`** access.

Raises an auditing event `winreg.SetValue` with arguments `key`, `sub_key`, `type`, `value`.

winreg.SetValueEx(*key*, *value_name*, *reserved*, *type*, *value*)

Stores data in the value field of an open registry key.

key is an already open key, or one of the predefined HKEY_* constants.

value_name is a string that names the subkey with which the value is associated.

reserved can be anything – zero is always passed to the API.

type is an integer that specifies the type of the data. See Value Types for the available types.

value is a string that specifies the new value.

This method can also set additional value and type information for the specified key. The key identified by the key parameter must have been opened with **`KEY_SET_VALUE`** access.

To open the key, use the **`CreateKey()`** or **`OpenKey()`** methods.

Value lengths are limited by available memory. Long values (more than 2048 bytes) should be stored as files with the filenames stored in the configuration registry. This helps the registry perform efficiently.

Raises an auditing event `winreg.SetValue` with arguments `key`, `sub_key`, `type`, `value`.

winreg.DisableReflectionKey(*key*)

Disables registry reflection for 32-bit processes running on a 64-bit operating system.

key is an already open key, or one of the predefined HKEY_* constants.

Will generally raise **`NotImplementedError`** if executed on a 32-bit operating system.

If the key is not on the reflection list, the function succeeds but has no effect. Disabling reflection for a key does not affect reflection of any subkeys.

Raises an auditing event `winreg.DisableReflectionKey` with argument `key`.

winreg.EnableReflectionKey(*key*)

Restores registry reflection for the specified disabled key.

key is an already open key, or one of the predefined HKEY_* constants.

Will generally raise **`NotImplementedError`** if executed on a 32-bit operating system.

Restoring reflection for a key does not affect reflection of any subkeys.

Raises an auditing event `winreg.EnableReflectionKey` with argument `key`.

winreg.QueryReflectionKey(*key*)

Determines the reflection state for the specified key.

key is an already open key, or one of the predefined HKEY_* constants.

Returns `True` if reflection is disabled.

Will generally raise **`NotImplementedError`** if executed on a 32-bit operating system.

Raises an auditing event `winreg.QueryReflectionKey` with argument `key`.

Constants

The following constants are defined for use in many **`winreg`** functions.

HKEY_* Constants

winreg.HKEY_CLASSES_ROOT

Registry entries subordinate to this key define types (or classes) of documents and the properties associated with those types. Shell and COM applications use the information stored under this key.

winreg.HKEY_CURRENT_USER

Registry entries subordinate to this key define the preferences of the current user. These preferences include the settings of environment variables, data about program groups, colors, printers, network connections, and application preferences.

winreg.HKEY_LOCAL_MACHINE

Registry entries subordinate to this key define the physical state of the computer, including data about the bus type, system memory, and installed hardware and software.

winreg.HKEY_USERS

Registry entries subordinate to this key define the default user configuration for new users on the local computer and the user configuration for the current user.

winreg.HKEY_PERFORMANCE_DATA

Registry entries subordinate to this key allow you to access performance data. The data is not actually stored in the registry; the registry functions cause the system to collect the data from its source.

winreg.HKEY_CURRENT_CONFIG

Contains information about the current hardware profile of the local computer system.

winreg.HKEY_DYN_DATA

This key is not used in versions of Windows after 98.

Access Rights

For more information, see Registry Key Security and Access [https://msdn.microsoft.com/en-us/library/ms724878%28v=VS.85%29.aspx].

winreg.KEY_ALL_ACCESS

Combines the STANDARD_RIGHTS_REQUIRED, **`KEY_QUERY_VALUE`**, **`KEY_SET_VALUE`**, **`KEY_CREATE_SUB_KEY`**, **`KEY_ENUMERATE_SUB_KEYS`**, **`KEY_NOTIFY`**, and **`KEY_CREATE_LINK`** access rights.

winreg.KEY_WRITE

Combines the STANDARD_RIGHTS_WRITE, **`KEY_SET_VALUE`**, and **`KEY_CREATE_SUB_KEY`** access rights.

winreg.KEY_READ

Combines the STANDARD_RIGHTS_READ, **`KEY_QUERY_VALUE`**, **`KEY_ENUMERATE_SUB_KEYS`**, and **`KEY_NOTIFY`** values.

winreg.KEY_EXECUTE

Equivalent to `KEY_READ`.

winreg.KEY_QUERY_VALUE

Required to query the values of a registry key.

winreg.KEY_SET_VALUE

Required to create, delete, or set a registry value.

winreg.KEY_CREATE_SUB_KEY

Required to create a subkey of a registry key.

winreg.KEY_ENUMERATE_SUB_KEYS

Required to enumerate the subkeys of a registry key.

winreg.KEY_NOTIFY

Required to request change notifications for a registry key or for subkeys of a registry key.

winreg.KEY_CREATE_LINK

Reserved for system use.

64-bit Specific

For more information, see Accessing an Alternate Registry View [https://msdn.microsoft.com/en-us/library/aa384129(v=VS.85).aspx].

winreg.KEY_WOW64_64KEY

Indicates that an application on 64-bit Windows should operate on the 64-bit registry view. On 32-bit Windows, this constant is ignored.

winreg.KEY_WOW64_32KEY

Indicates that an application on 64-bit Windows should operate on the 32-bit registry view. On 32-bit Windows, this constant is ignored.

Value Types

For more information, see Registry Value Types [https://msdn.microsoft.com/en-us/library/ms724884%28v=VS.85%29.aspx].

winreg.REG_BINARY
Binary data in any form.

winreg.REG_DWORD
32-bit number.

winreg.REG_DWORD_LITTLE_ENDIAN
A 32-bit number in little-endian format. Equivalent to `REG_DWORD`.

winreg.REG_DWORD_BIG_ENDIAN
A 32-bit number in big-endian format.

winreg.REG_EXPAND_SZ
Null-terminated string containing references to environment variables (`%PATH%`).

winreg.REG_LINK
A Unicode symbolic link.

winreg.REG_MULTI_SZ
A sequence of null-terminated strings, terminated by two null characters. (Python handles this termination automatically.)

winreg.REG_NONE
No defined value type.

winreg.REG_QWORD

A 64-bit number.

Added in version 3.6.

winreg.REG_QWORD_LITTLE_ENDIAN
A 64-bit number in little-endian format. Equivalent to `REG_QWORD`.

Added in version 3.6.

winreg.REG_RESOURCE_LIST
A device-driver resource list.

winreg.REG_FULL_RESOURCE_DESCRIPTOR
A hardware setting.

winreg.REG_RESOURCE_REQUIREMENTS_LIST
A hardware resource list.

winreg.REG_SZ
A null-terminated string.

Registry Handle Objects

This object wraps a Windows HKEY object, automatically closing it when the object is destroyed. To guarantee cleanup, you can call either the `Close()` method on the object, or the `CloseKey()` function.

All registry functions in this module return one of these objects.

All registry functions in this module which accept a handle object also accept an integer, however, use of the handle object is encouraged.

Handle objects provide semantics for `__bool__()` – thus

```
if handle:
    print("Yes")
```

will print `Yes` if the handle is currently valid (has not been closed or detached).

Handle objects can be converted to an integer (e.g., using the built-in `int()` function), in which case the underlying Windows handle value is returned. You can also use the `Detach()` method to return the integer handle, and also disconnect the Windows handle from the handle object.

PyHKEY.Close()

Closes the underlying Windows handle.

If the handle is already closed, no error is raised.

PyHKEY.Detach()

Detaches the Windows handle from the handle object.

The result is an integer that holds the value of the handle before it is detached. If the handle is already detached or closed, this will return zero.

After calling this function, the handle is effectively invalidated, but the handle is not closed. You would call this function when you need the underlying Win32 handle to exist beyond the lifetime of the handle object.

Raises an auditing event `winreg.PyHKEY.Detach` with argument `key`.

PyHKEY.__enter__()
PyHKEY.__exit__(**exc_info*)

The HKEY object implements `__enter__()` and `__exit__()` and thus supports the context protocol for the `with` statement:

```
with OpenKey(HKEY_LOCAL_MACHINE, "foo") as key:
    ...  # work with key
```

will automatically close *key* when control leaves the `with` block.

`winsound` — Sound-playing interface for Windows

The `winsound` module provides access to the basic sound-playing machinery provided by Windows platforms. It includes functions and several constants.

Availability: Windows.

winsound.Beep(*frequency*, *duration*)

Beep the PC’s speaker. The *frequency* parameter specifies frequency, in hertz, of the sound, and must be in the range 37 through 32,767. The *duration* parameter specifies the number of milliseconds the sound should last. If the system is not able to beep the speaker, `RuntimeError` is raised.

winsound.PlaySound(*sound*, *flags*)

Call the underlying `PlaySound()` function from the Platform API. The *sound* parameter may be a filename, a system sound alias, audio data as a bytes-like object, or `None`. Its interpretation depends on the value of *flags*, which can be a bitwise ORed combination of the constants described below. If the *sound* parameter is `None`, any currently playing waveform sound is stopped. If the system indicates an error, `RuntimeError` is raised.

winsound.MessageBeep(*type=MB_OK*)

Call the underlying `MessageBeep()` function from the Platform API. This plays a sound as specified in the registry. The *type* argument specifies which sound to play; possible values are `-1`, `MB_ICONASTERISK`, `MB_ICONEXCLAMATION`, `MB_ICONHAND`, `MB_ICONQUESTION`, and `MB_OK`, all described below. The value `-1` produces a “simple beep”; this is the final fallback if a sound cannot be played otherwise. If the system indicates an error, `RuntimeError` is raised.

winsound.SND_FILENAME

The *sound* parameter is the name of a WAV file. Do not use with `SND_ALIAS`.

winsound.SND_ALIAS

The *sound* parameter is a sound association name from the registry. If the registry contains no such name, play the system default sound unless `SND_NODEFAULT` is also specified. If no default sound is registered, raise `RuntimeError`. Do not use with `SND_FILENAME`.

All Win32 systems support at least the following; most systems support many more:

`PlaySound()` ***name***	**Corresponding Control Panel Sound name**
`'SystemAsterisk'`	Asterisk
`'SystemExclamation'`	Exclamation
`'SystemExit'`	Exit Windows
`'SystemHand'`	Critical Stop
`'SystemQuestion'`	Question

For example:

```
import winsound
# Play Windows exit sound.
winsound.PlaySound("SystemExit", winsound.SND_ALIAS)

# Probably play Windows default sound, if any is registered
(because
# "*" probably isn't the registered name of any sound).
winsound.PlaySound("*", winsound.SND_ALIAS)
```

winsound.SND_LOOP

Play the sound repeatedly. The `SND_ASYNC` flag must also be used to avoid blocking. Cannot be used with `SND_MEMORY`.

winsound.SND_MEMORY

The *sound* parameter to `PlaySound()` is a memory image of a WAV file, as a bytes-like object.

Note

This module does not support playing from a memory image asynchronously, so a combination of this flag and `SND_ASYNC` will raise `RuntimeError`.

winsound.SND_PURGE

Stop playing all instances of the specified sound.

Note

This flag is not supported on modern Windows platforms.

winsound.SND_ASYNC

Return immediately, allowing sounds to play asynchronously.

winsound.SND_NODEFAULT

If the specified sound cannot be found, do not play the system default sound.

winsound.SND_NOSTOP

Do not interrupt sounds currently playing.

winsound.SND_NOWAIT

Return immediately if the sound driver is busy.

Note

This flag is not supported on modern Windows platforms.

winsound.SND_APPLICATION

The *sound* parameter is an application-specific alias in the registry. This flag can be combined with the `SND_ALIAS` flag to specify an application-defined sound alias.

winsound.SND_SENTRY

Triggers a SoundSentry event when the sound is played.

Added in version 3.14.

winsound.SND_SYNC

The sound is played synchronously. This is the default behavior.

Added in version 3.14.

winsound.SND_SYSTEM

Assign the sound to the audio session for system notification sounds.

Added in version 3.14.

winsound.MB_ICONASTERISK
Play the `SystemDefault` sound.

winsound.MB_ICONEXCLAMATION
Play the `SystemExclamation` sound.

winsound.MB_ICONHAND
Play the `SystemHand` sound.

winsound.MB_ICONQUESTION
Play the `SystemQuestion` sound.

winsound.MB_OK
Play the `SystemDefault` sound.

winsound.MB_ICONERROR
Play the `SystemHand` sound.

Added in version 3.14.

winsound.MB_ICONINFORMATION
Play the `SystemDefault` sound.

Added in version 3.14.

winsound.MB_ICONSTOP
Play the `SystemHand` sound.

Added in version 3.14.

winsound.MB_ICONWARNING
Play the `SystemExclamation` sound.

Added in version 3.14.

Unix-specific services

The modules described in this chapter provide interfaces to features that are unique to the Unix operating system, or in some cases to some or many variants of it. Here’s an overview:

- **`shlex`** — Simple lexical analysis
  - shlex Objects
  - Parsing Rules
  - Improved Compatibility with Shells
- **`posix`** — The most common POSIX system calls
  - Large File Support
  - Notable Module Contents
- **`pwd`** — The password database
- **`grp`** — The group database
- **`termios`** — POSIX style tty control
  - Example
- **`tty`** — Terminal control functions
- **`pty`** — Pseudo-terminal utilities
  - Example
- **`fcntl`** — The **`fcntl`** and **`ioctl`** system calls
- **`resource`** — Resource usage information
  - Resource Limits
  - Resource Usage
- **`syslog`** — Unix syslog library routines
  - Examples
    - Simple example

`shlex` — Simple lexical analysis

Source code: Lib/shlex.py
[https://github.com/python/cpython/tree/3.14/Lib/shlex.py]

The **`shlex`** class makes it easy to write lexical analyzers for simple syntaxes resembling that of the Unix shell. This will often be useful for writing minilanguages, (for example, in run control files for Python applications) or for parsing quoted strings.

The **`shlex`** module defines the following functions:

shlex.split(*s*, *comments=False*, *posix=True*)

Split the string *s* using shell-like syntax. If *comments* is **`False`** (the default), the parsing of comments in the given string will be disabled (setting the **`commenters`** attribute of the **`shlex`** instance to the empty string). This function operates in POSIX mode by default, but uses non-POSIX mode if the *posix* argument is false.

Changed in version 3.12: Passing `None` for *s* argument now raises an exception, rather than reading **`sys.stdin`**.

shlex.join(*split_command*)

Concatenate the tokens of the list *split_command* and return a string. This function is the inverse of **`split()`**.

```
>>> from shlex import join
>>> print(join(['echo', '-n', 'Multiple words']))
echo -n 'Multiple words'
```

The returned value is shell-escaped to protect against injection vulnerabilities (see `quote()`).

Added in version 3.8.

shlex.quote(*s*)

Return a shell-escaped version of the string *s*. The returned value is a string that can safely be used as one token in a shell command line, for cases where you cannot use a list.

Warning

The `shlex` module is **only designed for Unix shells**.

The `quote()` function is not guaranteed to be correct on non-POSIX compliant shells or shells from other operating systems such as Windows. Executing commands quoted by this module on such shells can open up the possibility of a command injection vulnerability.

Consider using functions that pass command arguments with lists such as `subprocess.run()` with `shell=False`.

This idiom would be unsafe:

```
>>> filename = 'somefile; rm -rf ~'
>>> command = 'ls -l {}'.format(filename)
>>> print(command)  # executed by a shell: boom!
ls -l somefile; rm -rf ~
```

`quote()` lets you plug the security hole:

```
>>> from shlex import quote
>>> command = 'ls -l {}'.format(quote(filename))
>>> print(command)
```

```
ls -l 'somefile; rm -rf ~'
>>> remote_command = 'ssh home {}'.format(quote(command))
>>> print(remote_command)
ssh home 'ls -l '"'"'somefile; rm -rf ~'"'"''
```

The quoting is compatible with UNIX shells and with `split()`:

```
>>> from shlex import split
>>> remote_command = split(remote_command)
>>> remote_command
['ssh', 'home', "ls -l 'somefile; rm -rf ~'"]
>>> command = split(remote_command[-1])
>>> command
['ls', '-l', 'somefile; rm -rf ~']
```

Added in version 3.3.

The `shlex` module defines the following class:

class shlex.shlex(*instream=None*, *infile=None*, *posix=False*, *punctuation_chars=False*)

A `shlex` instance or subclass instance is a lexical analyzer object. The initialization argument, if present, specifies where to read characters from. It must be a file-/stream-like object with `read()` and `readline()` methods, or a string. If no argument is given, input will be taken from `sys.stdin`. The second optional argument is a filename string, which sets the initial value of the `infile` attribute. If the *instream* argument is omitted or equal to `sys.stdin`, this second argument defaults to “stdin”. The *posix* argument defines the operational mode: when *posix* is not true (default), the `shlex` instance will operate in compatibility mode. When operating in POSIX mode, `shlex` will try to be as close as possible to the POSIX shell parsing rules. The *punctuation_chars* argument provides a way to make the behaviour even closer to how real shells parse. This can take a number of values: the default value, `False`, preserves the behaviour seen under

Python 3.5 and earlier. If set to `True`, then parsing of the characters `();<>|&` is changed: any run of these characters (considered punctuation characters) is returned as a single token. If set to a non-empty string of characters, those characters will be used as the punctuation characters. Any characters in the **`wordchars`** attribute that appear in *punctuation_chars* will be removed from **`wordchars`**. See Improved Compatibility with Shells for more information. *punctuation_chars* can be set only upon **`shlex`** instance creation and can't be modified later.

Changed in version 3.6: The *punctuation_chars* parameter was added.

See also

Module **`configparser`**

Parser for configuration files similar to the Windows `.ini` files.

shlex Objects

A **`shlex`** instance has the following methods:

shlex.get_token()

Return a token. If tokens have been stacked using **`push_token()`**, pop a token off the stack. Otherwise, read one from the input stream. If reading encounters an immediate end-of-file, **`eof`** is returned (the empty string (`''`) in non-POSIX mode, and `None` in POSIX mode).

shlex.push_token(*str*)

Push the argument onto the token stack.

shlex.read_token()

Read a raw token. Ignore the pushback stack, and do not interpret source requests. (This is not ordinarily a useful entry point, and is documented here only for the sake of completeness.)

shlex.sourcehook(*filename*)

When **`shlex`** detects a source request (see **`source`** below) this method is given the following token as argument, and expected to return a tuple consisting of a filename and an open file-like object.

Normally, this method first strips any quotes off the argument. If the result is an absolute pathname, or there was no previous source request in effect, or the previous source was a stream (such as `sys.stdin`), the result is left alone. Otherwise, if the result is a relative pathname, the directory part of the name of the file immediately before it on the source inclusion stack is prepended (this behavior is like the way the C preprocessor handles `#include "file.h"`).

The result of the manipulations is treated as a filename, and returned as the first component of the tuple, with **`open()`** called on it to yield the second component. (Note: this is the reverse of the order of arguments in instance initialization!)

This hook is exposed so that you can use it to implement directory search paths, addition of file extensions, and other namespace hacks. There is no corresponding ‘close’ hook, but a shlex instance will call the **`close()`** method of the sourced input stream when it returns EOF.

For more explicit control of source stacking, use the **`push_source()`** and **`pop_source()`** methods.

shlex.push_source(*newstream*, *newfile=None*)

Push an input source stream onto the input stack. If the filename argument is specified it will later be available for use in error messages. This is the same method used internally by the `sourcehook()` method.

shlex.pop_source()

Pop the last-pushed input source from the input stack. This is the same method used internally when the lexer reaches EOF on a stacked input stream.

shlex.error_leader(*infile=None*, *lineno=None*)

This method generates an error message leader in the format of a Unix C compiler error label; the format is `'"%s", line %d: '`, where the `%s` is replaced with the name of the current source file and the `%d` with the current input line number (the optional arguments can be used to override these).

This convenience is provided to encourage `shlex` users to generate error messages in the standard, parseable format understood by Emacs and other Unix tools.

Instances of `shlex` subclasses have some public instance variables which either control lexical analysis or can be used for debugging:

shlex.commenters

The string of characters that are recognized as comment beginners. All characters from the comment beginner to end of line are ignored. Includes just `'#'` by default.

shlex.wordchars

The string of characters that will accumulate into multi-character tokens. By default, includes all ASCII alphanumerics and underscore. In POSIX mode, the accented characters in the Latin-1 set are also included. If `punctuation_chars` is not empty, the characters `~-./*?=`, which can appear in filename specifications and command line parameters, will also be included in this attribute, and any characters which appear in `punctuation_chars` will be removed from `wordchars` if they are present there. If `whitespace_split` is set to `True`, this will have no effect.

shlex.whitespace

Characters that will be considered whitespace and skipped. Whitespace bounds tokens. By default, includes space, tab, linefeed and carriage-return.

shlex.escape

Characters that will be considered as escape. This will be only used in POSIX mode, and includes just `'\'` by default.

shlex.quotes

Characters that will be considered string quotes. The token accumulates until the same quote is encountered again (thus, different quote types protect each other as in the shell.) By default, includes ASCII single and double quotes.

shlex.escapedquotes

Characters in `quotes` that will interpret escape characters defined in `escape`. This is only used in POSIX mode, and includes just `'"'` by default.

shlex.whitespace_split

If `True`, tokens will only be split in whitespaces. This is useful, for example, for parsing command lines with `shlex`, getting tokens in a similar way to shell arguments. When used in combination with `punctuation_chars`, tokens will be split on whitespace in addition to those characters.

Changed in version 3.8: The **`punctuation_chars`** attribute was made compatible with the **`whitespace_split`** attribute.

shlex.infile

The name of the current input file, as initially set at class instantiation time or stacked by later source requests. It may be useful to examine this when constructing error messages.

shlex.instream

The input stream from which this **`shlex`** instance is reading characters.

shlex.source

This attribute is `None` by default. If you assign a string to it, that string will be recognized as a lexical-level inclusion request similar to the `source` keyword in various shells. That is, the immediately following token will be opened as a filename and input will be taken from that stream until EOF, at which point the **`close()`** method of that stream will be called and the input source will again become the original input stream. Source requests may be stacked any number of levels deep.

shlex.debug

If this attribute is numeric and `1` or more, a **`shlex`** instance will print verbose progress output on its behavior. If you need to use this, you can read the module source code to learn the details.

shlex.lineno

Source line number (count of newlines seen so far plus one).

shlex.token

The token buffer. It may be useful to examine this when catching exceptions.

shlex.eof

Token used to determine end of file. This will be set to the empty string (`''`), in non-POSIX mode, and to `None` in POSIX mode.

shlex.punctuation_chars

A read-only property. Characters that will be considered punctuation. Runs of punctuation characters will be returned as a single token. However, note that no semantic validity checking will be performed: for example, ‘>>>’ could be returned as a token, even though it may not be recognised as such by shells.

Added in version 3.6.

Parsing Rules

When operating in non-POSIX mode, **`shlex`** will try to obey to the following rules.

- Quote characters are not recognized within words (`Do"Not"Separate` is parsed as the single word `Do"Not"Separate`);
- Escape characters are not recognized;
- Enclosing characters in quotes preserve the literal value of all characters within the quotes;
- Closing quotes separate words (`"Do"Separate` is parsed as `"Do"` and `Separate`);
- If **`whitespace_split`** is `False`, any character not declared to be a word character, whitespace, or a quote will be

returned as a single-character token. If it is `True`, `shlex` will only split words in whitespaces;
- EOF is signaled with an empty string (`''`);
- It’s not possible to parse empty strings, even if quoted.

When operating in POSIX mode, `shlex` will try to obey to the following parsing rules.

- Quotes are stripped out, and do not separate words (`"Do"Not"Separate"` is parsed as the single word `DoNotSeparate`);
- Non-quoted escape characters (e.g. `'\'`) preserve the literal value of the next character that follows;
- Enclosing characters in quotes which are not part of `escapedquotes` (e.g. `"'"`) preserve the literal value of all characters within the quotes;
- Enclosing characters in quotes which are part of `escapedquotes` (e.g. `'"'`) preserves the literal value of all characters within the quotes, with the exception of the characters mentioned in `escape`. The escape characters retain its special meaning only when followed by the quote in use, or the escape character itself. Otherwise the escape character will be considered a normal character.
- EOF is signaled with a `None` value;
- Quoted empty strings (`''`) are allowed.

Improved Compatibility with Shells

Added in version 3.6.

The `shlex` class provides compatibility with the parsing performed by common Unix shells like `bash`, `dash`, and `sh`. To take advantage of this compatibility, specify the `punctuation_chars` argument in the constructor. This defaults to `False`, which preserves pre-3.6 behaviour. However, if it is set to `True`, then parsing of the characters `();<>|&` is changed: any run of these characters is returned as a single token. While this is short of a full parser for shells (which would be out of scope for the standard library, given the multiplicity of shells out there), it does allow you to perform processing of command lines more easily than you could otherwise. To illustrate, you can see the difference in the following snippet:

```
>>> import shlex
>>> text = "a && b; c && d || e; f >'abc'; (def \"ghi\")"
>>> s = shlex.shlex(text, posix=True)
>>> s.whitespace_split = True
>>> list(s)
['a', '&&', 'b;', 'c', '&&', 'd', '||', 'e;', 'f', '>abc;',
'(def', 'ghi)']
>>> s = shlex.shlex(text, posix=True, punctuation_chars=True)
>>> s.whitespace_split = True
>>> list(s)
['a', '&&', 'b', ';', 'c', '&&', 'd', '||', 'e', ';', 'f', '>',
'abc', ';',
'(', 'def', 'ghi', ')']
```

Of course, tokens will be returned which are not valid for shells, and you’ll need to implement your own error checks on the returned tokens.

Instead of passing `True` as the value for the punctuation_chars parameter, you can pass a string with specific characters, which will be used to determine which characters constitute punctuation. For example:

```
>>> import shlex
>>> s = shlex.shlex("a && b || c", punctuation_chars="|")
>>> list(s)
['a', '&', '&', 'b', '||', 'c']
```

Note

When `punctuation_chars` is specified, the **`wordchars`** attribute is augmented with the characters `~-./*?=`. That is because these characters can appear in file names (including wildcards) and command-line arguments (e.g. `--color=auto`). Hence:

```
>>> import shlex
>>> s = shlex.shlex('~/a && b-c --color=auto || d *.py?',
...                 punctuation_chars=True)
>>> list(s)
['~/a', '&&', 'b-c', '--color=auto', '||', 'd', '*.py?']
```

However, to match the shell as closely as possible, it is recommended to always use `posix` and **`whitespace_split`** when using **`punctuation_chars`**, which will negate **`wordchars`** entirely.

For best effect, `punctuation_chars` should be set in conjunction with `posix=True`. (Note that `posix=False` is the default for **`shlex`**.)

`posix` — The most common POSIX system calls

This module provides access to operating system functionality that is standardized by the C Standard and the POSIX standard (a thinly disguised Unix interface).

Availability: Unix.

Do not import this module directly. Instead, import the module `os`, which provides a *portable* version of this interface. On Unix, the `os` module provides a superset of the `posix` interface. On non-Unix operating systems the `posix` module is not available, but a subset is always available through the `os` interface. Once `os` is imported, there is *no* performance penalty in using it instead of `posix`. In addition, `os` provides some additional functionality, such as automatically calling `putenv()` when an entry in `os.environ` is changed.

Errors are reported as exceptions; the usual exceptions are given for type errors, while errors reported by the system calls raise `OSError`.

Large File Support

Several operating systems (including AIX and Solaris) provide support for files that are larger than 2 GiB from a C programming model where int and long are 32-bit values. This is typically accomplished by defining the relevant size and offset types as 64-bit values. Such files are sometimes referred to as *large files*.

Large file support is enabled in Python when the size of an **`off_t`** is larger than a long and the long long is at least as large as an **`off_t`**. It may be necessary to configure and compile Python with certain compiler flags to enable this mode. For example, with Solaris 2.6 and 2.7 you need to do something like:

```
CFLAGS="`getconf LFS_CFLAGS`" OPT="-g -O2 $CFLAGS" \
        ./configure
```

On large-file-capable Linux systems, this might work:

```
CFLAGS='-D_LARGEFILE64_SOURCE -D_FILE_OFFSET_BITS=64' OPT="-g -
O2 $CFLAGS" \
        ./configure
```

Notable Module Contents

In addition to many functions described in the **`os`** module documentation, **`posix`** defines the following data item:

posix.environ

A dictionary representing the string environment at the time the interpreter was started. Keys and values are bytes on Unix and str on Windows. For example, `environ[b'HOME']` (`environ['HOME']` on Windows) is the pathname of your home directory, equivalent to `getenv("HOME")` in C.

Modifying this dictionary does not affect the string environment passed on by **`execv()`**, **`popen()`** or **`system()`**; if you need to change the environment, pass `environ` to **`execve()`** or add variable assignments and export statements to the command string for **`system()`** or **`popen()`**.

Changed in version 3.2: On Unix, keys and values are bytes.

Note

The `os` module provides an alternate implementation of `environ` which updates the environment on modification. Note also that updating `os.environ` will render this dictionary obsolete. Use of the `os` module version of this is recommended over direct access to the `posix` module.

`pwd` — The password database

This module provides access to the Unix user account and password database. It is available on all Unix versions.

Availability: Unix, not WASI, not iOS.

Password database entries are reported as a tuple-like object, whose attributes correspond to the members of the `passwd` structure (Attribute field below, see `<pwd.h>`):

Index	Attribute	Meaning
0	`pw_name`	Login name
1	`pw_passwd`	Optional encrypted password
2	`pw_uid`	Numerical user ID
3	`pw_gid`	Numerical group ID
4	`pw_gecos`	User name or comment field
5	`pw_dir`	User home directory

Index	Attribute	Meaning
6	`pw_shell`	User command interpreter

The uid and gid items are integers, all others are strings. `KeyError` is raised if the entry asked for cannot be found.

Note

In traditional Unix the field `pw_passwd` usually contains a password encrypted with a DES derived algorithm. However most modern unices use a so-called *shadow password* system. On those unices the *pw_passwd* field only contains an asterisk (`'*'`) or the letter `'x'` where the encrypted password is stored in a file `/etc/shadow` which is not world readable. Whether the *pw_passwd* field contains anything useful is system-dependent.

It defines the following items:

pwd.getpwuid(*uid*)

Return the password database entry for the given numeric user ID.

pwd.getpwnam(*name*)

Return the password database entry for the given user name.

pwd.getpwall()

Return a list of all available password database entries, in arbitrary order.

See also

Module `grp`

An interface to the group database, similar to this.

`grp` — The group database

This module provides access to the Unix group database. It is available on all Unix versions.

Availability: Unix, not WASI, not Android, not iOS.

Group database entries are reported as a tuple-like object, whose attributes correspond to the members of the `group` structure (Attribute field below, see `<grp.h>`):

Index	Attribute	Meaning
0	gr_name	the name of the group
1	gr_passwd	the (encrypted) group password; often empty
2	gr_gid	the numerical group ID
3	gr_mem	all the group member’s user names

The gid is an integer, name and password are strings, and the member list is a list of strings. (Note that most users are not explicitly listed as members of the group they are in according to the password database. Check both databases to get complete membership information. Also note that a `gr_name` that starts with a `+` or `-` is likely to be a YP/NIS reference and may not be accessible via `getgrnam()` or `getgrgid()`.)

It defines the following items:

grp.getgrgid(*id*)

Return the group database entry for the given numeric group ID. `KeyError` is raised if the entry asked for cannot be found.

Changed in version 3.10: `TypeError` is raised for non-integer arguments like floats or strings.

grp.getgrnam(*name*)

Return the group database entry for the given group name. `KeyError` is raised if the entry asked for cannot be found.

grp.getgrall()

Return a list of all available group entries, in arbitrary order.

See also

Module `pwd`

An interface to the user database, similar to this.

`termios` — POSIX style tty control

This module provides an interface to the POSIX calls for tty I/O control. For a complete description of these calls, see termios(3) *[https://manpages.debian.org/termios(3)]* Unix manual page. It is only available for those Unix versions that support POSIX *termios* style tty I/O control configured during installation.

Availability: Unix.

All functions in this module take a file descriptor *fd* as their first argument. This can be an integer file descriptor, such as returned by `sys.stdin.fileno()`, or a file object, such as `sys.stdin` itself.

This module also defines all the constants needed to work with the functions provided here; these have the same name as their counterparts in C. Please refer to your system documentation for more information on using these terminal control interfaces.

The module defines the following functions:

termios.tcgetattr(*fd*)

Return a list containing the tty attributes for file descriptor *fd*, as follows: `[iflag, oflag, cflag, lflag, ispeed, ospeed, cc]` where *cc* is a list of the tty special characters (each a string of length 1, except the items with indices **`VMIN`** and **`VTIME`**, which are integers when these fields are defined). The interpretation of the flags and the speeds as well as the indexing in the *cc* array must be done using the symbolic constants defined in the `termios` module.

termios.tcsetattr(*fd*, *when*, *attributes*)

Set the tty attributes for file descriptor *fd* from the *attributes*, which is a list like the one returned by `tcgetattr()`. The *when* argument determines when the attributes are changed:

termios.TCSANOW

Change attributes immediately.

termios.TCSADRAIN

Change attributes after transmitting all queued output.

termios.TCSAFLUSH

Change attributes after transmitting all queued output and discarding all queued input.

termios.tcsendbreak(*fd*, *duration*)

Send a break on file descriptor *fd*. A zero *duration* sends a break for 0.25–0.5 seconds; a nonzero *duration* has a system dependent meaning.

termios.tcdrain(*fd*)

Wait until all output written to file descriptor *fd* has been transmitted.

termios.tcflush(*fd*, *queue*)

Discard queued data on file descriptor *fd*. The *queue* selector specifies which queue: **`TCIFLUSH`** for the input queue, **`TCOFLUSH`** for the output queue, or **`TCIOFLUSH`** for both queues.

termios.tcflow(*fd*, *action*)

Suspend or resume input or output on file descriptor *fd*. The *action* argument can be **`TCOOFF`** to suspend output, **`TCOON`** to restart output, **`TCIOFF`** to suspend input, or **`TCION`** to restart input.

termios.tcgetwinsize(*fd*)

Return a tuple `(ws_row, ws_col)` containing the tty window size for file descriptor *fd*. Requires **`termios.TIOCGWINSZ`** or **`termios.TIOCGSIZE`**.

Added in version 3.11.

termios.tcsetwinsize(*fd*, *winsize*)

Set the tty window size for file descriptor *fd* from *winsize*, which is a two-item tuple `(ws_row, ws_col)` like the one returned by **`tcgetwinsize()`**. Requires at least one of the pairs (**`termios.TIOCGWINSZ`**, **`termios.TIOCSWINSZ`**); (**`termios.TIOCGSIZE`**, **`termios.TIOCSSIZE`**) to be defined.

Added in version 3.11.

See also

Module `tty`

Convenience functions for common terminal control operations.

Example

Here’s a function that prompts for a password with echoing turned off. Note the technique using a separate **`tcgetattr()`** call and a **`try`** … **`finally`** statement to ensure that the old tty attributes are restored exactly no matter what happens:

```
def getpass(prompt="Password: "):
    import termios, sys
    fd = sys.stdin.fileno()
    old = termios.tcgetattr(fd)
    new = termios.tcgetattr(fd)
    new[3] = new[3] & ~termios.ECHO          # lflags
    try:
        termios.tcsetattr(fd, termios.TCSADRAIN, new)
        passwd = input(prompt)
    finally:
        termios.tcsetattr(fd, termios.TCSADRAIN, old)
    return passwd
```

`tty` — Terminal control functions

Source code: Lib/tty.py
[https://github.com/python/cpython/tree/3.14/Lib/tty.py]

The **`tty`** module defines functions for putting the tty into cbreak and raw modes.

Availability: Unix.

Because it requires the **`termios`** module, it will work only on Unix.

The **`tty`** module defines the following functions:

tty.cfmakeraw(*mode*)

Convert the tty attribute list *mode*, which is a list like the one returned by **`termios.tcgetattr()`**, to that of a tty in raw mode.

Added in version 3.12.

tty.cfmakecbreak(*mode*)

Convert the tty attribute list *mode*, which is a list like the one returned by **`termios.tcgetattr()`**, to that of a tty in cbreak mode.

This clears the `ECHO` and `ICANON` local mode flags in *mode* as well as setting the minimum input to 1 byte with no delay.

Added in version 3.12.

Changed in version 3.12.2: The `ICRNL` flag is no longer cleared. This matches Linux and macOS `stty cbreak` behavior and what `setcbreak()` historically did.

tty.setraw(*fd*, *when=termios.TCSAFLUSH*)

Change the mode of the file descriptor *fd* to raw. If *when* is omitted, it defaults to `termios.TCSAFLUSH`, and is passed to `termios.tcsetattr()`. The return value of `termios.tcgetattr()` is saved before setting *fd* to raw mode; this value is returned.

Changed in version 3.12: The return value is now the original tty attributes, instead of `None`.

tty.setcbreak(*fd*, *when=termios.TCSAFLUSH*)

Change the mode of file descriptor *fd* to cbreak. If *when* is omitted, it defaults to `termios.TCSAFLUSH`, and is passed to `termios.tcsetattr()`. The return value of `termios.tcgetattr()` is saved before setting *fd* to cbreak mode; this value is returned.

This clears the `ECHO` and `ICANON` local mode flags as well as setting the minimum input to 1 byte with no delay.

Changed in version 3.12: The return value is now the original tty attributes, instead of `None`.

Changed in version 3.12.2: The `ICRNL` flag is no longer cleared. This restores the behavior of Python 3.11 and earlier as well as matching what Linux, macOS, & BSDs describe in their `stty(1)` man pages regarding cbreak mode.

See also

Module `termios`

Low-level terminal control interface.

`pty` — Pseudo-terminal utilities

Source code: Lib/pty.py [https://github.com/python/cpython/tree/3.14/Lib/pty.py]

The **`pty`** module defines operations for handling the pseudo-terminal concept: starting another process and being able to write to and read from its controlling terminal programmatically.

Availability: Unix.

Pseudo-terminal handling is highly platform dependent. This code is mainly tested on Linux, FreeBSD, and macOS (it is supposed to work on other POSIX platforms but it’s not been thoroughly tested).

The **`pty`** module defines the following functions:

pty.fork()

Fork. Connect the child’s controlling terminal to a pseudo-terminal. Return value is `(pid, fd)`. Note that the child gets *pid* 0, and the *fd* is *invalid*. The parent’s return value is the *pid* of the child, and *fd* is a file descriptor connected to the child’s controlling terminal (and also to the child’s standard input and output).

Warning

On macOS the use of this function is unsafe when mixed with using higher-level system APIs, and that includes using `urllib.request`.

pty.openpty()

Open a new pseudo-terminal pair, using `os.openpty()` if possible, or emulation code for generic Unix systems. Return a pair of file descriptors `(master, slave)`, for the master and the slave end, respectively.

pty.spawn(*argv*[, *master_read*[, *stdin_read*]])

Spawn a process, and connect its controlling terminal with the current process's standard io. This is often used to baffle programs which insist on reading from the controlling terminal. It is expected that the process spawned behind the pty will eventually terminate, and when it does *spawn* will return.

A loop copies STDIN of the current process to the child and data received from the child to STDOUT of the current process. It is not signaled to the child if STDIN of the current process closes down.

The functions *master_read* and *stdin_read* are passed a file descriptor which they should read from, and they should always return a byte string. In order to force spawn to return before the child process exits an empty byte array should be returned to signal end of file.

The default implementation for both functions will read and return up to 1024 bytes each time the function is called. The *master_read* callback is passed the pseudoterminal's master file descriptor to read output from the child process, and *stdin_read* is passed file descriptor 0, to read from the parent process's standard input.

Returning an empty byte string from either callback is interpreted as an end-of-file (EOF) condition, and that callback will not be called after that. If *stdin_read* signals EOF the controlling terminal can no longer communicate with the parent process OR the child process. Unless the child process will quit without any input, *spawn* will then loop forever. If *master_read* signals EOF the same behavior results (on linux at least).

Return the exit status value from **`os.waitpid()`** on the child process.

`os.waitstatus_to_exitcode()` can be used to convert the exit status into an exit code.

Raises an auditing event `pty.spawn` with argument `argv`.

Changed in version 3.4: **`spawn()`** now returns the status value from **`os.waitpid()`** on the child process.

Example

The following program acts like the Unix command *script(1)* *[https://manpages.debian.org/script(1)]*, using a pseudo-terminal to record all input and output of a terminal session in a "typescript".

```
import argparse
import os
import pty
import sys
import time

parser = argparse.ArgumentParser()
```

```
parser.add_argument('-a', dest='append', action='store_true')
parser.add_argument('-p', dest='use_python',
action='store_true')
parser.add_argument('filename', nargs='?',
default='typescript')
options = parser.parse_args()

shell = sys.executable if options.use_python else
os.environ.get('SHELL', 'sh')
filename = options.filename
mode = 'ab' if options.append else 'wb'

with open(filename, mode) as script:
    def read(fd):
        data = os.read(fd, 1024)
        script.write(data)
        return data

    print('Script started, file is', filename)
    script.write(('Script started on %s\n' %
time.asctime()).encode())

    pty.spawn(shell, read)

    script.write(('Script done on %s\n' %
time.asctime()).encode())
    print('Script done, file is', filename)
```

`fcntl` — The `fcntl` and `ioctl` system calls

This module performs file and I/O control on file descriptors. It is an interface to the **`fcntl()`** and **`ioctl()`** Unix routines. See the *fcntl(2)* *[https://manpages.debian.org/fcntl(2)]* and *ioctl(2)* *[https://manpages.debian.org/ioctl(2)]* Unix manual pages for full details.

Availability: Unix, not WASI.

All functions in this module take a file descriptor *fd* as their first argument. This can be an integer file descriptor, such as returned by `sys.stdin.fileno()`, or an **`io.IOBase`** object, such as `sys.stdin` itself, which provides a **`fileno()`** that returns a genuine file descriptor.

Changed in version 3.3: Operations in this module used to raise an **`IOError`** where they now raise an **`OSError`**.

Changed in version 3.8: The **`fcntl`** module now contains `F_ADD_SEALS`, `F_GET_SEALS`, and `F_SEAL_*` constants for sealing of **`os.memfd_create()`** file descriptors.

Changed in version 3.9: On macOS, the **`fcntl`** module exposes the `F_GETPATH` constant, which obtains the path of a file from a file descriptor. On Linux(>=3.15), the **`fcntl`** module exposes the `F_OFD_GETLK`, `F_OFD_SETLK` and `F_OFD_SETLKW` constants, which are used when working with open file description locks.

Changed in version 3.10: On Linux >= 2.6.11, the **fcntl** module exposes the `F_GETPIPE_SZ` and `F_SETPIPE_SZ` constants, which allow to check and modify a pipe’s size respectively.

Changed in version 3.11: On FreeBSD, the **fcntl** module exposes the `F_DUP2FD` and `F_DUP2FD_CLOEXEC` constants, which allow to duplicate a file descriptor, the latter setting `FD_CLOEXEC` flag in addition.

Changed in version 3.12: On Linux >= 4.5, the **fcntl** module exposes the `FICLONE` and `FICLONERANGE` constants, which allow to share some data of one file with another file by reflinking on some filesystems (e.g., btrfs, OCFS2, and XFS). This behavior is commonly referred to as “copy-on-write”.

Changed in version 3.13: On Linux >= 2.6.32, the **fcntl** module exposes the `F_GETOWN_EX`, `F_SETOWN_EX`, `F_OWNER_TID`, `F_OWNER_PID`, `F_OWNER_PGRP` constants, which allow to direct I/O availability signals to a specific thread, process, or process group. On Linux >= 4.13, the **fcntl** module exposes the `F_GET_RW_HINT`, `F_SET_RW_HINT`, `F_GET_FILE_RW_HINT`, `F_SET_FILE_RW_HINT`, and `RWH_WRITE_LIFE_*` constants, which allow to inform the kernel about the relative expected lifetime of writes on a given inode or via a particular open file description. On Linux >= 5.1 and NetBSD, the **fcntl** module exposes the `F_SEAL_FUTURE_WRITE` constant for use with `F_ADD_SEALS` and `F_GET_SEALS` operations. On FreeBSD, the **fcntl** module exposes the `F_READAHEAD`, `F_ISUNIONSTACK`, and `F_KINFO` constants. On macOS and FreeBSD, the **fcntl** module exposes the `F_RDAHEAD` constant. On NetBSD and AIX, the **fcntl** module exposes the `F_CLOSEM` constant. On NetBSD, the **fcntl** module exposes the `F_MAXFD` constant. On macOS and NetBSD, the **fcntl** module exposes the `F_GETNOSIGPIPE` and `F_SETNOSIGPIPE` constant.

Changed in version 3.14: On Linux >= 6.1, the **`fcntl`** module exposes the `F_DUPFD_QUERY` to query a file descriptor pointing to the same file.

The module defines the following functions:

fcntl.fcntl(*fd*, *cmd*, *arg=0*, */*)

Perform the operation *cmd* on file descriptor *fd* (file objects providing a **`fileno()`** method are accepted as well). The values used for *cmd* are operating system dependent, and are available as constants in the **`fcntl`** module, using the same names as used in the relevant C header files. The argument *arg* can either be an integer value, a bytes-like object, or a string. The type and size of *arg* must match the type and size of the argument of the operation as specified in the relevant C documentation.

When *arg* is an integer, the function returns the integer return value of the C **`fcntl()`** call.

When the argument is bytes-like object, it represents a binary structure, for example, created by **`struct.pack()`**. A string value is encoded to binary using the UTF-8 encoding. The binary data is copied to a buffer whose address is passed to the C **`fcntl()`** call. The return value after a successful call is the contents of the buffer, converted to a **`bytes`** object. The length of the returned object will be the same as the length of the *arg* argument. This is limited to 1024 bytes.

If the **`fcntl()`** call fails, an **`OSError`** is raised.

Note

If the type or the size of *arg* does not match the type or size of the argument of the operation (for example, if an integer is passed when a pointer is expected, or the information returned in the buffer by the operating system is larger than 1024 bytes), this is most likely to result in a segmentation violation or a more subtle data corruption.

Raises an auditing event `fcntl.fcntl` with arguments `fd`, `cmd`, `arg`.

Changed in version 3.14: Add support of arbitrary bytes-like objects, not only **`bytes`**.

fcntl.ioctl(*fd*, *request*, *arg=0*, *mutate_flag=True*, */*)

This function is identical to the **`fcntl()`** function, except that the argument handling is even more complicated.

The *request* parameter is limited to values that can fit in 32-bits or 64-bits, depending on the platform. Additional constants of interest for use as the *request* argument can be found in the **`termios`** module, under the same names as used in the relevant C header files.

The parameter *arg* can be an integer, a bytes-like object, or a string. The type and size of *arg* must match the type and size of the argument of the operation as specified in the relevant C documentation.

If *arg* does not support the read-write buffer interface or the *mutate_flag* is false, behavior is as for the **`fcntl()`** function.

If *arg* supports the read-write buffer interface (like **`bytearray`**) and *mutate_flag* is true (the default), then the buffer is (in effect) passed to the underlying **`ioctl()`** system call, the latter's return code is passed back to the calling Python, and the buffer's new contents reflect the action of the **`ioctl()`**. This is a slight simplification, because if the supplied buffer is less than 1024 bytes long it is first copied into a static buffer 1024 bytes long which is then passed to `ioctl()` and copied back into the supplied buffer.

If the **`ioctl()`** call fails, an `OSError` exception is raised.

Note

If the type or size of *arg* does not match the type or size of the operation’s argument (for example, if an integer is passed when a pointer is expected, or the information returned in the buffer by the operating system is larger than 1024 bytes, or the size of the mutable bytes-like object is too small), this is most likely to result in a segmentation violation or a more subtle data corruption.

An example:

```
>>> import array, fcntl, struct, termios, os
>>> os.getpgrp()
13341
>>> struct.unpack('h', fcntl.ioctl(0, termios.TIOCGPGRP, "
"))[0]
13341
>>> buf = array.array('h', [0])
>>> fcntl.ioctl(0, termios.TIOCGPGRP, buf, 1)
0
>>> buf
array('h', [13341])
```

Raises an auditing event `fcntl.ioctl` with arguments `fd`, `request`, `arg`.

Changed in version 3.14: The GIL is always released during a system call. System calls failing with EINTR are automatically retried.

fcntl.flock(*fd*, *operation*, */*)

Perform the lock operation *operation* on file descriptor *fd* (file objects providing a **`fileno()`** method are accepted as well). See the Unix manual *flock(2)* *[https://manpages.debian.org/flock(2)]* for details. (On some systems, this function is emulated using **`fcntl()`**.)

If the **`flock()`** call fails, an **`OSError`** exception is raised.

Raises an auditing event `fcntl.flock` with arguments `fd`, `operation`.

fcntl.lockf(*fd*, *cmd*, *len=0*, *start=0*, *whence=0*, */*)

This is essentially a wrapper around the **`fcntl()`** locking calls. *fd* is the file descriptor (file objects providing a **`fileno()`** method are accepted as well) of the file to lock or unlock, and *cmd* is one of the following values:

fcntl.LOCK_UN

Release an existing lock.

fcntl.LOCK_SH

Acquire a shared lock.

fcntl.LOCK_EX

Acquire an exclusive lock.

fcntl.LOCK_NB

Bitwise OR with any of the other three `LOCK_*` constants to make the request non-blocking.

If **`LOCK_NB`** is used and the lock cannot be acquired, an **`OSError`** will be raised and the exception will have an *errno* attribute set to **`EACCES`** or **`EAGAIN`** (depending on the operating system; for portability, check for both values). On at least some systems, **`LOCK_EX`** can only be used if the file descriptor refers to a file opened for writing.

len is the number of bytes to lock, *start* is the byte offset at which the lock starts, relative to *whence*, and *whence* is as with **`io.IOBase.seek()`**, specifically:

- `0` – relative to the start of the file (**`os.SEEK_SET`**)
- `1` – relative to the current buffer position (**`os.SEEK_CUR`**)
- `2` – relative to the end of the file (**`os.SEEK_END`**)

The default for *start* is 0, which means to start at the beginning of the file. The default for *len* is 0 which means to lock to the end of the file. The default for *whence* is also 0.

Raises an auditing event `fcntl.lockf` with arguments `fd`, `cmd`, `len`, `start`, `whence`.

Examples (all on a SVR4 compliant system):

```
import struct, fcntl, os

f = open(...)
rv = fcntl.fcntl(f, fcntl.F_SETFL, os.O_NDELAY)

lockdata = struct.pack('hhllhh', fcntl.F_WRLCK, 0, 0, 0, 0, 0)
rv = fcntl.fcntl(f, fcntl.F_SETLKW, lockdata)
```

Note that in the first example the return value variable *rv* will hold an integer value; in the second example it will hold a **`bytes`** object. The structure lay-out for the *lockdata* variable is system dependent — therefore using the **`flock()`** call may be better.

See also

Module `os`

If the locking flags `O_SHLOCK` and `O_EXLOCK` are present in the `os` module (on BSD only), the `os.open()` function provides an alternative to the `lockf()` and `flock()` functions.

`resource` — Resource usage information

This module provides basic mechanisms for measuring and controlling system resources utilized by a program.

Availability: Unix, not WASI.

Symbolic constants are used to specify particular system resources and to request usage information about either the current process or its children.

An **`OSError`** is raised on syscall failure.

exception resource.error

A deprecated alias of **`OSError`**.

Changed in version 3.3: Following **PEP 3151** [https://peps.python.org/pep-3151/], this class was made an alias of **`OSError`**.

Resource Limits

Resources usage can be limited using the **`setrlimit()`** function described below. Each resource is controlled by a pair of limits: a soft limit and a hard limit. The soft limit is the current limit, and may be lowered or raised by a process over time. The soft limit can never exceed the hard limit. The hard limit can be lowered to any value greater than the soft limit, but not raised. (Only processes with the effective UID of the super-user can raise a hard limit.)

The specific resources that can be limited are system dependent. They are described in the *getrlimit(2)* *[https://manpages.debian.org/getrlimit(2)]* man page. The resources listed below are supported when the underlying operating system supports them; resources which cannot be checked or controlled by the operating system are not defined in this module for those platforms.

resource.RLIM_INFINITY

Constant used to represent the limit for an unlimited resource.

resource.getrlimit(*resource*)

Returns a tuple `(soft, hard)` with the current soft and hard limits of *resource*. Raises **`ValueError`** if an invalid resource is specified, or **`error`** if the underlying system call fails unexpectedly.

resource.setrlimit(*resource*, *limits*)

Sets new limits of consumption of *resource*. The *limits* argument must be a tuple `(soft, hard)` of two integers describing the new limits. A value of **`RLIM_INFINITY`** can be used to request a limit that is unlimited.

Raises **`ValueError`** if an invalid resource is specified, if the new soft limit exceeds the hard limit, or if a process tries to raise its hard limit. Specifying a limit of **`RLIM_INFINITY`** when the hard or system limit for that resource is not unlimited will result in a **`ValueError`**. A process with the effective UID of super-user can request any valid limit value, including unlimited, but **`ValueError`** will still be raised if the requested limit exceeds the system imposed limit.

`setrlimit` may also raise **`error`** if the underlying system call fails.

VxWorks only supports setting **`RLIMIT_NOFILE`**.

Raises an auditing event `resource.setrlimit` with arguments `resource`, `limits`.

resource.prlimit(*pid*, *resource*[, *limits*])

Combines **`setrlimit()`** and **`getrlimit()`** in one function and supports to get and set the resources limits of an arbitrary process. If *pid* is 0, then the call applies to the current process. *resource* and *limits* have the same meaning as in **`setrlimit()`**, except that *limits* is optional.

When *limits* is not given the function returns the *resource* limit of the process *pid*. When *limits* is given the *resource* limit of the process is set and the former resource limit is returned.

Raises **`ProcessLookupError`** when *pid* can’t be found and **`PermissionError`** when the user doesn’t have `CAP_SYS_RESOURCE` for the process.

Raises an auditing event `resource.prlimit` with arguments `pid`, `resource`, `limits`.

Availability: Linux >= 2.6.36 with glibc >= 2.13.

Added in version 3.4.

These symbols define resources whose consumption can be controlled using the **`setrlimit()`** and **`getrlimit()`** functions described below. The values of these symbols are exactly the constants used by C programs.

The Unix man page for *getrlimit(2)* *[https://manpages.debian.org/getrlimit(2)]* lists the available resources. Note that not all systems use the same symbol or same value to denote the same resource. This module does not attempt to mask platform differences — symbols not defined for a platform will not be available from this module on that platform.

resource.RLIMIT_CORE

The maximum size (in bytes) of a core file that the current process can create. This may result in the creation of a partial core file if a larger core would be required to contain the entire process image.

resource.RLIMIT_CPU

The maximum amount of processor time (in seconds) that a process can use. If this limit is exceeded, a `SIGXCPU` signal is sent to the process. (See the `signal` module documentation for information about how to catch this signal and do something useful, e.g. flush open files to disk.)

resource.RLIMIT_FSIZE

The maximum size of a file which the process may create.

resource.RLIMIT_DATA

The maximum size (in bytes) of the process’s heap.

resource.RLIMIT_STACK

The maximum size (in bytes) of the call stack for the current process. This only affects the stack of the main thread in a multi-threaded process.

resource.RLIMIT_RSS

The maximum resident set size that should be made available to the process.

resource.RLIMIT_NPROC

The maximum number of processes the current process may create.

resource.RLIMIT_NOFILE

The maximum number of open file descriptors for the current process.

resource.RLIMIT_OFILE

The BSD name for **`RLIMIT_NOFILE`**.

resource.RLIMIT_MEMLOCK

The maximum address space which may be locked in memory.

resource.RLIMIT_VMEM

The largest area of mapped memory which the process may occupy. Usually an alias of **`RLIMIT_AS`**.

Availability: Solaris, FreeBSD, NetBSD.

resource.RLIMIT_AS

The maximum area (in bytes) of address space which may be taken by the process.

resource.RLIMIT_MSGQUEUE

The number of bytes that can be allocated for POSIX message queues.

Availability: Linux >= 2.6.8.

Added in version 3.4.

resource.RLIMIT_NICE

The ceiling for the process’s nice level (calculated as 20 - rlim_cur).

Availability: Linux >= 2.6.12.

Added in version 3.4.

resource.RLIMIT_RTPRIO

The ceiling of the real-time priority.

Availability: Linux >= 2.6.12.

Added in version 3.4.

resource.RLIMIT_RTTIME

The time limit (in microseconds) on CPU time that a process can spend under real-time scheduling without making a blocking syscall.

Availability: Linux >= 2.6.25.

Added in version 3.4.

resource.RLIMIT_SIGPENDING

The number of signals which the process may queue.

Availability: Linux >= 2.6.8.

Added in version 3.4.

resource.RLIMIT_SBSIZE

The maximum size (in bytes) of socket buffer usage for this user. This limits the amount of network memory, and hence the amount of mbufs, that this user may hold at any time.

Availability: FreeBSD, NetBSD.

Added in version 3.4.

resource.RLIMIT_SWAP

The maximum size (in bytes) of the swap space that may be reserved or used by all of this user id's processes. This limit is enforced only if bit 1 of the vm.overcommit sysctl is set. Please see tuning(7) [https://man.freebsd.org/cgi/man.cgi?query=tuning&sektion=7] for a complete description of this sysctl.

Availability: FreeBSD >= 8.

Added in version 3.4.

resource.RLIMIT_NPTS

The maximum number of pseudo-terminals created by this user id.

Availability: FreeBSD >= 8.

Added in version 3.4.

resource.RLIMIT_KQUEUES

The maximum number of kqueues this user id is allowed to create.

Availability: FreeBSD >= 11.

Added in version 3.10.

Resource Usage

These functions are used to retrieve resource usage information:

resource.getrusage(*who*)

This function returns an object that describes the resources consumed by either the current process or its children, as specified by the *who* parameter. The *who* parameter should be specified using one of the **`RUSAGE_*`** constants described below.

A simple example:

```
from resource import *
import time

# a non CPU-bound task
time.sleep(3)
print(getrusage(RUSAGE_SELF))

# a CPU-bound task
for i in range(10 ** 8):
   _ = 1 + 1
print(getrusage(RUSAGE_SELF))
```

The fields of the return value each describe how a particular system resource has been used, e.g. amount of time spent running is user mode or number of times the process was swapped out of main memory. Some values are dependent on the clock tick internal, e.g. the amount of memory the process is using.

For backward compatibility, the return value is also accessible as a tuple of 16 elements.

The fields **`ru_utime`** and **`ru_stime`** of the return value are floating-point values representing the amount of time spent executing in user mode and the amount of time spent executing in system mode, respectively. The remaining values are integers. Consult the *getrusage(2)* *[https://manpages.debian.org/getrusage(2)]* man page for detailed information about these values. A brief summary is presented here:

Index	**Field**	**Resource**
`0`	**`ru_utime`**	time in user mode (float seconds)
`1`	**`ru_stime`**	time in system mode (float seconds)
`2`	**`ru_maxrss`**	maximum resident set size
`3`	**`ru_ixrss`**	shared memory size
`4`	**`ru_idrss`**	unshared memory size
`5`	**`ru_isrss`**	unshared stack size
`6`	**`ru_minflt`**	page faults not requiring I/O
`7`	**`ru_majflt`**	page faults requiring I/O
`8`	**`ru_nswap`**	number of swap outs
`9`	**`ru_inblock`**	block input operations
`10`	**`ru_oublock`**	block output operations
`11`	**`ru_msgsnd`**	messages sent
`12`	**`ru_msgrcv`**	messages received
`13`	**`ru_nsignals`**	signals received
`14`	**`ru_nvcsw`**	voluntary context switches
`15`	**`ru_nivcsw`**	involuntary context switches

This function will raise a `ValueError` if an invalid *who* parameter is specified. It may also raise `error` exception in unusual circumstances.

resource.getpagesize()

Returns the number of bytes in a system page. (This need not be the same as the hardware page size.)

The following **`RUSAGE_*`** symbols are passed to the `getrusage()` function to specify which processes information should be provided for.

resource.RUSAGE_SELF

Pass to `getrusage()` to request resources consumed by the calling process, which is the sum of resources used by all threads in the process.

resource.RUSAGE_CHILDREN

Pass to `getrusage()` to request resources consumed by child processes of the calling process which have been terminated and waited for.

resource.RUSAGE_BOTH

Pass to `getrusage()` to request resources consumed by both the current process and child processes. May not be available on all systems.

resource.RUSAGE_THREAD

Pass to `getrusage()` to request resources consumed by the current thread. May not be available on all systems.

Added in version 3.2.

`syslog` — Unix syslog library routines

This module provides an interface to the Unix `syslog` library routines. Refer to the Unix manual pages for a detailed description of the `syslog` facility.

Availability: Unix, not WASI, not iOS.

This module wraps the system `syslog` family of routines. A pure Python library that can speak to a syslog server is available in the **`logging.handlers`** module as **`SysLogHandler`**.

The module defines the following functions:

syslog.syslog(*message*)
syslog.syslog(*priority*, *message*)

> Send the string *message* to the system logger. A trailing newline is added if necessary. Each message is tagged with a priority composed of a *facility* and a *level*. The optional *priority* argument, which defaults to **`LOG_INFO`**, determines the message priority. If the facility is not encoded in *priority* using logical-or (`LOG_INFO | LOG_USER`), the value given in the **`openlog()`** call is used.
>
> If **`openlog()`** has not been called prior to the call to **`syslog()`**, **`openlog()`** will be called with no arguments.
>
> Raises an auditing event `syslog.syslog` with arguments `priority`, `message`.

Changed in version 3.2: In previous versions, `openlog()` would not be called automatically if it wasn't called prior to the call to `syslog()`, deferring to the syslog implementation to call `openlog()`.

Changed in version 3.12: This function is restricted in subinterpreters. (Only code that runs in multiple interpreters is affected and the restriction is not relevant for most users.) `openlog()` must be called in the main interpreter before `syslog()` may be used in a subinterpreter. Otherwise it will raise `RuntimeError`.

syslog.openlog([*ident*[, *logoption*[, *facility*]]])

Logging options of subsequent `syslog()` calls can be set by calling `openlog()`. `syslog()` will call `openlog()` with no arguments if the log is not currently open.

The optional *ident* keyword argument is a string which is prepended to every message, and defaults to `sys.argv[0]` with leading path components stripped. The optional *logoption* keyword argument (default is 0) is a bit field – see below for possible values to combine. The optional *facility* keyword argument (default is `LOG_USER`) sets the default facility for messages which do not have a facility explicitly encoded.

Raises an auditing event `syslog.openlog` with arguments `ident`, `logoption`, `facility`.

Changed in version 3.2: In previous versions, keyword arguments were not allowed, and *ident* was required.

Changed in version 3.12: This function is restricted in subinterpreters. (Only code that runs in multiple interpreters is affected and the restriction is not relevant for most users.) This may only be called in the main interpreter. It will raise `RuntimeError` if called in a subinterpreter.

syslog.closelog()

Reset the syslog module values and call the system library `closelog()`.

This causes the module to behave as it does when initially imported. For example, `openlog()` will be called on the first `syslog()` call (if `openlog()` hasn't already been called), and *ident* and other `openlog()` parameters are reset to defaults.

Raises an auditing event `syslog.closelog` with no arguments.

Changed in version 3.12: This function is restricted in subinterpreters. (Only code that runs in multiple interpreters is affected and the restriction is not relevant for most users.) This may only be called in the main interpreter. It will raise `RuntimeError` if called in a subinterpreter.

syslog.setlogmask(*maskpri*)

Set the priority mask to *maskpri* and return the previous mask value. Calls to `syslog()` with a priority level not set in *maskpri* are ignored. The default is to log all priorities. The function `LOG_MASK(pri)` calculates the mask for the individual priority *pri*. The function `LOG_UPTO(pri)` calculates the mask for all priorities up to and including *pri*.

Raises an auditing event `syslog.setlogmask` with argument `maskpri`.

The module defines the following constants:

syslog.LOG_EMERG
syslog.LOG_ALERT
syslog.LOG_CRIT
syslog.LOG_ERR
syslog.LOG_WARNING
syslog.LOG_NOTICE
syslog.LOG_INFO
syslog.LOG_DEBUG

Priority levels (high to low).

syslog.LOG_AUTH
syslog.LOG_AUTHPRIV
syslog.LOG_CRON
syslog.LOG_DAEMON
syslog.LOG_FTP
syslog.LOG_INSTALL
syslog.LOG_KERN
syslog.LOG_LAUNCHD
syslog.LOG_LPR
syslog.LOG_MAIL
syslog.LOG_NETINFO
syslog.LOG_NEWS
syslog.LOG_RAS
syslog.LOG_REMOTEAUTH
syslog.LOG_SYSLOG
syslog.LOG_USER
syslog.LOG_UUCP
syslog.LOG_LOCAL0
syslog.LOG_LOCAL1
syslog.LOG_LOCAL2
syslog.LOG_LOCAL3
syslog.LOG_LOCAL4
syslog.LOG_LOCAL5
syslog.LOG_LOCAL6
syslog.LOG_LOCAL7

Facilities, depending on availability in `<syslog.h>` for `LOG_AUTHPRIV`, `LOG_FTP`, `LOG_NETINFO`, `LOG_REMOTEAUTH`, `LOG_INSTALL` and `LOG_RAS`.

Changed in version 3.13: Added `LOG_FTP`, `LOG_NETINFO`, `LOG_REMOTEAUTH`, `LOG_INSTALL`, `LOG_RAS`, and `LOG_LAUNCHD`.

syslog.LOG_PID
syslog.LOG_CONS
syslog.LOG_NDELAY
syslog.LOG_ODELAY
syslog.LOG_NOWAIT
syslog.LOG_PERROR

Log options, depending on availability in `<syslog.h>` for `LOG_ODELAY`, `LOG_NOWAIT` and `LOG_PERROR`.

Examples

Simple example

A simple set of examples:

```
import syslog

syslog.syslog('Processing started')
if error:
    syslog.syslog(syslog.LOG_ERR, 'Processing started')
```

An example of setting some log options, these would include the process ID in logged messages, and write the messages to the destination facility used for mail logging:

```
syslog.openlog(logoption=syslog.LOG_PID,
facility=syslog.LOG_MAIL)
syslog.syslog('E-mail processing initiated...')
```

Modules command-line interface (CLI)

The following modules have a command-line interface.

- ast
- asyncio
- `base64`
- calendar
- `code`
- compileall
- `cProfile`: see profile
- dis
- doctest
- `encodings.rot_13`
- ensurepip
- `filecmp`
- `fileinput`
- `ftplib`
- gzip
- http.server
- idlelib
- inspect
- json
- mimetypes
- pdb
- pickle
- pickletools
- platform
- `poplib`
- profile
- `pstats`

- py_compile
- `pyclbr`
- `pydoc`
- `quopri`
- random
- `runpy`
- site
- sqlite3
- symtable
- sysconfig
- `tabnanny`
- tarfile
- `this`
- timeit
- tokenize
- trace
- `turtledemo`
- unittest
- uuid
- venv
- webbrowser
- zipapp
- zipfile

See also the Python command-line interface.

Superseded Modules

The modules described in this chapter have been superseded by other modules for most use cases, and are retained primarily to preserve backwards compatibility.

Modules may appear in this chapter because they only cover a limited subset of a problem space, and a more generally applicable solution is available elsewhere in the standard library (for example, `getopt` covers the very specific task of “mimic the C `getopt()` API in Python”, rather than the broader command line option parsing and argument parsing capabilities offered by `optparse` and `argparse`).

Alternatively, modules may appear in this chapter because they are deprecated outright, and awaiting removal in a future release, or they are soft deprecated and their use is actively discouraged in new projects. With the removal of various obsolete modules through **PEP 594** [https://peps.python.org/pep-0594/], there are currently no modules in this latter category.

- `getopt` — C-style parser for command line options

`getopt` — C-style parser for command line options

Source code: Lib/getopt.py [https://github.com/python/cpython/tree/3.14/Lib/getopt.py]

Note

This module is considered feature complete. A more declarative and extensible alternative to this API is provided in the **`optparse`** module. Further functional enhancements for command line parameter processing are provided either as third party modules on PyPI, or else as features in the **`argparse`** module.

This module helps scripts to parse the command line arguments in `sys.argv`. It supports the same conventions as the Unix **`getopt()`** function (including the special meanings of arguments of the form '`-`' and '`--`'). Long options similar to those supported by GNU software may be used as well via an optional third argument.

Users who are unfamiliar with the Unix **`getopt()`** function should consider using the **`argparse`** module instead. Users who are familiar with the Unix **`getopt()`** function, but would like to get equivalent behavior while writing less code and getting better help and error messages should consider using the **`optparse`** module. See Choosing an argument parsing library for additional details.

This module provides two functions and an exception:

getopt.getopt(*args*, *shortopts*, *longopts=[]*)

Parses command line options and parameter list. *args* is the argument list to be parsed, without the leading reference to the running program. Typically, this means `sys.argv[1:]`. *shortopts* is the string of option letters that the script wants to recognize, with options that require an argument followed by a colon (`':'`) and options that accept an optional argument followed by two colons (`'::'`); i.e., the same format that Unix **`getopt()`** uses.

Note

Unlike GNU **`getopt()`**, after a non-option argument, all further arguments are considered also non-options. This is similar to the way non-GNU Unix systems work.

longopts, if specified, must be a list of strings with the names of the long options which should be supported. The leading `'--'` characters should not be included in the option name. Long options which require an argument should be followed by an equal sign (`'='`). Long options which accept an optional argument should be followed by an equal sign and question mark (`'=?'`). To accept only long options, *shortopts* should be an empty string. Long options on the command line can be recognized so long as they provide a prefix of the option name that matches exactly one of the accepted options. For example, if *longopts* is `['foo', 'frob']`, the option `--fo` will match as `--foo`, but `--f` will not match uniquely, so **`GetoptError`** will be raised.

The return value consists of two elements: the first is a list of `(option, value)` pairs; the second is the list of program arguments left after the option list was stripped (this is a trailing slice of *args*). Each option-and-value pair returned has the option as its first element, prefixed with a hyphen for short options (e.g., `'-x'`) or two hyphens for long options (e.g., `'--long-option'`), and the option argument as its second element, or an empty string if the option has no argument. The options occur in the list in the same order in which they were found, thus allowing multiple occurrences. Long and short options may be mixed.

Changed in version 3.14: Optional arguments are supported.

getopt.gnu_getopt(*args*, *shortopts*, *longopts=[]*)

This function works like **`getopt()`**, except that GNU style scanning mode is used by default. This means that option and non-option arguments may be intermixed. The **`getopt()`** function stops processing options as soon as a non-option argument is encountered.

If the first character of the option string is `'+'`, or if the environment variable **`POSIXLY_CORRECT`** is set, then option processing stops as soon as a non-option argument is encountered.

If the first character of the option string is `'-'`, non-option arguments that are followed by options are added to the list of option-and-value pairs as a pair that has `None` as its first element and the list of non-option arguments as its second element. The second element of the **`gnu_getopt()`** result is a list of program arguments after the last option.

Changed in version 3.14: Support for returning intermixed options and non-option arguments in order.

exception getopt.GetoptError

This is raised when an unrecognized option is found in the argument list or when an option requiring an argument is given none. The argument to the exception is a string indicating the cause of the error. For long options, an argument given to an option which does not require one will also cause this exception to be raised. The attributes **`msg`** and **`opt`** give the error message and related option; if there is no specific option to which the exception relates, **`opt`** is an empty string.

exception getopt.error

Alias for `GetoptError`; for backward compatibility.

An example using only Unix style options:

```
>>> import getopt
>>> args = '-a -b -cfoo -d bar a1 a2'.split()
>>> args
['-a', '-b', '-cfoo', '-d', 'bar', 'a1', 'a2']
>>> optlist, args = getopt.getopt(args, 'abc:d:')
>>> optlist
[('-a', ''), ('-b', ''), ('-c', 'foo'), ('-d', 'bar')]
>>> args
['a1', 'a2']
```

Using long option names is equally easy:

```
>>> s = '--condition=foo --testing --output-file abc.def -x a1
a2'
>>> args = s.split()
>>> args
['--condition=foo', '--testing', '--output-file', 'abc.def', '-
x', 'a1', 'a2']
>>> optlist, args = getopt.getopt(args, 'x', [
...     'condition=', 'output-file=', 'testing'])
>>> optlist
[('--condition', 'foo'), ('--testing', ''), ('--output-file',
'abc.def'), ('-x', '')]
>>> args
['a1', 'a2']
```

Optional arguments should be specified explicitly:

```
>>> s = '-Con -C --color=off --color a1 a2'
>>> args = s.split()
>>> args
['-Con', '-C', '--color=off', '--color', 'a1', 'a2']
>>> optlist, args = getopt.getopt(args, 'C::', ['color=?'])
>>> optlist
[('-C', 'on'), ('-C', ''), ('--color', 'off'), ('--color', '')]
>>> args
['a1', 'a2']
```

The order of options and non-option arguments can be preserved:

```
>>> s = 'a1 -x a2 a3 a4 --long a5 a6'
>>> args = s.split()
>>> args
['a1', '-x', 'a2', 'a3', 'a4', '--long', 'a5', 'a6']
>>> optlist, args = getopt.gnu_getopt(args, '-x:', ['long='])
>>> optlist
[(None, ['a1']), ('-x', 'a2'), (None, ['a3', 'a4']), ('--long',
'a5')]
>>> args
['a6']
```

In a script, typical usage is something like this:

```
import getopt, sys

def main():
    try:
        opts, args = getopt.getopt(sys.argv[1:], "ho:v",
["help", "output="])
    except getopt.GetoptError as err:
        # print help information and exit:
        print(err)  # will print something like "option -a not
recognized"
        usage()
        sys.exit(2)
    output = None
    verbose = False
    for o, a in opts:
        if o == "-v":
            verbose = True
        elif o in ("-h", "--help"):
            usage()
            sys.exit()
        elif o in ("-o", "--output"):
            output = a
        else:
            assert False, "unhandled option"
    process(args, output=output, verbose=verbose)

if __name__ == "__main__":
    main()
```

Note that an equivalent command line interface could be produced with less code and more informative help and error messages by using the `optparse` module:

```
import optparse

if __name__ == '__main__':
    parser = optparse.OptionParser()
    parser.add_option('-o', '--output')
    parser.add_option('-v', dest='verbose',
action='store_true')
    opts, args = parser.parse_args()
    process(args, output=opts.output, verbose=opts.verbose)
```

A roughly equivalent command line interface for this case can also be produced by using the `argparse` module:

```
import argparse

if __name__ == '__main__':
    parser = argparse.ArgumentParser()
    parser.add_argument('-o', '--output')
    parser.add_argument('-v', dest='verbose',
action='store_true')
    parser.add_argument('rest', nargs='*')
    args = parser.parse_args()
    process(args.rest, output=args.output,
verbose=args.verbose)
```

See Choosing an argument parsing library for details on how the `argparse` version of this code differs in behaviour from the `optparse` (and `getopt`) version.

See also

Module `optparse`
Declarative command line option parsing.

Module `argparse`
More opinionated command line option and argument parsing library.

Removed Modules

The modules described in this chapter have been removed from the Python standard library. They are documented here to help people find replacements.

- **`aifc`** — Read and write AIFF and AIFC files
- **`asynchat`** — Asynchronous socket command/response handler
- **`asyncore`** — Asynchronous socket handler
- **`audioop`** — Manipulate raw audio data
- **`cgi`** — Common Gateway Interface support
- **`cgitb`** — Traceback manager for CGI scripts
- **`chunk`** — Read IFF chunked data
- **`crypt`** — Function to check Unix passwords
- **`distutils`** — Building and installing Python modules
- **`imghdr`** — Determine the type of an image
- **`imp`** — Access the import internals
- **`mailcap`** — Mailcap file handling
- **`msilib`** — Read and write Microsoft Installer files
- **`nis`** — Interface to Sun’s NIS (Yellow Pages)
- **`nntplib`** — NNTP protocol client
- **`ossaudiodev`** — Access to OSS-compatible audio devices
- **`pipes`** — Interface to shell pipelines
- **`smtpd`** — SMTP Server
- **`sndhdr`** — Determine type of sound file
- **`spwd`** — The shadow password database
- **`sunau`** — Read and write Sun AU files
- **`telnetlib`** — Telnet client
- **`uu`** — Encode and decode uuencode files
- **`xdrlib`** — Encode and decode XDR data

Security Considerations

The following modules have specific security considerations:

- **`base64`**: base64 security considerations in **RFC 4648** [https://datatracker.ietf.org/doc/html/rfc4648.html]
- **`hashlib`**: all constructors take a “usedforsecurity” keyword-only argument disabling known insecure and blocked algorithms
- **`http.server`** is not suitable for production use, only implementing basic security checks. See the security considerations.
- **`logging`**: Logging configuration uses eval()
- **`multiprocessing`**: Connection.recv() uses pickle
- **`pickle`**: Restricting globals in pickle
- **`random`** shouldn’t be used for security purposes, use **`secrets`** instead
- **`shelve`**: shelve is based on pickle and thus unsuitable for dealing with untrusted sources
- **`ssl`**: SSL/TLS security considerations
- **`subprocess`**: Subprocess security considerations
- **`tempfile`**: mktemp is deprecated due to vulnerability to race conditions
- **`xml`**: XML security
- **`zipfile`**: maliciously prepared .zip files can cause disk volume exhaustion

The **`-I`** command line option can be used to run Python in isolated mode. When it cannot be used, the **`-P`** option or the **`PYTHONSAFEPATH`** environment variable can be used to not prepend a potentially unsafe path to **`sys.path`** such as the current directory, the script’s directory or an empty string.

Extending and Embedding the Python Interpreter

This document describes how to write modules in C or C++ to extend the Python interpreter with new modules. Those modules can not only define new functions but also new object types and their methods. The document also describes how to embed the Python interpreter in another application, for use as an extension language. Finally, it shows how to compile and link extension modules so that they can be loaded dynamically (at run time) into the interpreter, if the underlying operating system supports this feature.

This document assumes basic knowledge about Python. For an informal introduction to the language, see The Python Tutorial. The Python Language Reference gives a more formal definition of the language. The Python Standard Library documents the existing object types, functions and modules (both built-in and written in Python) that give the language its wide application range.

For a detailed description of the whole Python/C API, see the separate Python/C API Reference Manual.

Recommended third party tools

This guide only covers the basic tools for creating extensions provided as part of this version of CPython. Some third party tools offer both simpler and more sophisticated approaches to creating C and C++ extensions for Python.

Creating extensions without third party tools

This section of the guide covers creating C and C++ extensions without assistance from third party tools. It is intended primarily for creators of those tools, rather than being a recommended way to create your own C extensions.

See also

PEP 489 [https://peps.python.org/pep-0489/] – Multi-phase extension module initialization

- 1. Extending Python with C or C++
  - 1.1. A Simple Example
  - 1.2. Intermezzo: Errors and Exceptions
  - 1.3. Back to the Example
  - 1.4. The Module’s Method Table and Initialization Function
  - 1.5. Compilation and Linkage
  - 1.6. Calling Python Functions from C
  - 1.7. Extracting Parameters in Extension Functions
  - 1.8. Keyword Parameters for Extension Functions
  - 1.9. Building Arbitrary Values
  - 1.10. Reference Counts
  - 1.11. Writing Extensions in C++
  - 1.12. Providing a C API for an Extension Module
- 2. Defining Extension Types: Tutorial
  - 2.1. The Basics
  - 2.2. Adding data and methods to the Basic example
  - 2.3. Providing finer control over data attributes
  - 2.4. Supporting cyclic garbage collection
  - 2.5. Subclassing other types
- 3. Defining Extension Types: Assorted Topics

    - 3.1. Finalization and De-allocation
    - 3.2. Object Presentation
    - 3.3. Attribute Management
    - 3.4. Object Comparison
    - 3.5. Abstract Protocol Support
    - 3.6. Weak Reference Support
    - 3.7. More Suggestions
- 4. Building C and C++ Extensions
    - 4.1. Building C and C++ Extensions with setuptools
- 5. Building C and C++ Extensions on Windows
    - 5.1. A Cookbook Approach
    - 5.2. Differences Between Unix and Windows
    - 5.3. Using DLLs in Practice

Embedding the CPython runtime in a larger application

Sometimes, rather than creating an extension that runs inside the Python interpreter as the main application, it is desirable to instead embed the CPython runtime inside a larger application. This section covers some of the details involved in doing that successfully.

- 1. Embedding Python in Another Application
    - 1.1. Very High Level Embedding
    - 1.2. Beyond Very High Level Embedding: An overview
    - 1.3. Pure Embedding
    - 1.4. Extending Embedded Python
    - 1.5. Embedding Python in C++
    - 1.6. Compiling and Linking under Unix-like systems

1. Extending Python with C or C++

It is quite easy to add new built-in modules to Python, if you know how to program in C. Such *extension modules* can do two things that can’t be done directly in Python: they can implement new built-in object types, and they can call C library functions and system calls.

To support extensions, the Python API (Application Programmers Interface) defines a set of functions, macros and variables that provide access to most aspects of the Python run-time system. The Python API is incorporated in a C source file by including the header `"Python.h"`.

The compilation of an extension module depends on its intended use as well as on your system setup; details are given in later chapters.

Note

The C extension interface is specific to CPython, and extension modules do not work on other Python implementations. In many cases, it is possible to avoid writing C extensions and preserve portability to other implementations. For example, if your use case is calling C library functions or system calls, you should consider using the **`ctypes`** module or the cffi [https://cffi.readthedocs.io/] library rather than writing custom C code. These modules let you write Python code to interface with C code and are more portable between implementations of Python than writing and compiling a C extension module.

1.1. A Simple Example

Let’s create an extension module called `spam` (the favorite food of Monty Python fans…) and let’s say we want to create a Python interface to the C library function **`system()`** [[1]]. This function takes a null-terminated character string as argument and returns an integer. We want this function to be callable from Python as follows:

```
>>> import spam
>>> status = spam.system("ls -l")
```

Begin by creating a file `spammodule.c`. (Historically, if a module is called `spam`, the C file containing its implementation is called `spammodule.c`; if the module name is very long, like `spammify`, the module name can be just `spammify.c`.)

The first two lines of our file can be:

```
#define PY_SSIZE_T_CLEAN
#include <Python.h>
```

which pulls in the Python API (you can add a comment describing the purpose of the module and a copyright notice if you like).

Note

Since Python may define some pre-processor definitions which affect the standard headers on some systems, you *must* include `Python.h` before any standard headers are included.

`#define PY_SSIZE_T_CLEAN` was used to indicate that `Py_ssize_t` should be used in some APIs instead of `int`. It is not necessary since Python 3.13, but we keep it here for backward compatibility. See Strings and buffers for a description of this macro.

All user-visible symbols defined by `Python.h` have a prefix of `Py` or `PY`, except those defined in standard header files.

Tip

For backward compatibility, `Python.h` includes several standard header files. C extensions should include the standard headers that they use, and should not rely on these implicit includes. If using the limited C API version 3.13 or newer, the implicit includes are:

- `<assert.h>`
- `<intrin.h>` (on Windows)
- `<inttypes.h>`
- `<limits.h>`
- `<math.h>`
- `<stdarg.h>`
- `<wchar.h>`
- `<sys/types.h>` (if present)

If **`Py_LIMITED_API`** is not defined, or is set to version 3.12 or older, the headers below are also included:

- `<ctype.h>`
- `<unistd.h>` (on POSIX)

If **`Py_LIMITED_API`** is not defined, or is set to version 3.10 or older, the headers below are also included:

- `<errno.h>`
- `<stdio.h>`
- `<stdlib.h>`

- `<string.h>`

The next thing we add to our module file is the C function that will be called when the Python expression `spam.system(string)` is evaluated (we’ll see shortly how it ends up being called):

```
static PyObject *
spam_system(PyObject *self, PyObject *args)
{
    const char *command;
    int sts;

    if (!PyArg_ParseTuple(args, "s", &command))
        return NULL;
    sts = system(command);
    return PyLong_FromLong(sts);
}
```

There is a straightforward translation from the argument list in Python (for example, the single expression `"ls -l"`) to the arguments passed to the C function. The C function always has two arguments, conventionally named *self* and *args*.

The *self* argument points to the module object for module-level functions; for a method it would point to the object instance.

The *args* argument will be a pointer to a Python tuple object containing the arguments. Each item of the tuple corresponds to an argument in the call’s argument list. The arguments are Python objects — in order to do anything with them in our C function we have to convert them to C values. The function **`PyArg_ParseTuple()`** in the Python API checks the argument types and converts them to C values. It uses a template string to determine the required types of the arguments as well as the types of the C variables into which to store the converted values. More about this later.

`PyArg_ParseTuple()` returns true (nonzero) if all arguments have the right type and its components have been stored in the variables whose addresses are passed. It returns false (zero) if an invalid argument list was passed. In the latter case it also raises an appropriate exception so the calling function can return `NULL` immediately (as we saw in the example).

1.2. Intermezzo: Errors and Exceptions

An important convention throughout the Python interpreter is the following: when a function fails, it should set an exception condition and return an error value (usually `-1` or a `NULL` pointer). Exception information is stored in three members of the interpreter's thread state. These are `NULL` if there is no exception. Otherwise they are the C equivalents of the members of the Python tuple returned by `sys.exc_info()`. These are the exception type, exception instance, and a traceback object. It is important to know about them to understand how errors are passed around.

The Python API defines a number of functions to set various types of exceptions.

The most common one is `PyErr_SetString()`. Its arguments are an exception object and a C string. The exception object is usually a predefined object like `PyExc_ZeroDivisionError`. The C string indicates the cause of the error and is converted to a Python string object and stored as the "associated value" of the exception.

Another useful function is `PyErr_SetFromErrno()`, which only takes an exception argument and constructs the associated value by inspection of the global variable **`errno`**. The most general function is `PyErr_SetObject()`, which takes two object arguments, the exception and its associated value. You don't need to `Py_INCREF()` the objects passed to any of these functions.

You can test non-destructively whether an exception has been set with `PyErr_Occurred()`. This returns the current exception object, or `NULL` if no exception has occurred. You normally don't need to call `PyErr_Occurred()` to see whether an error occurred in a function call, since you should be able to tell from the return value.

When a function *f* that calls another function *g* detects that the latter fails, *f* should itself return an error value (usually `NULL` or `-1`). It should *not* call one of the `PyErr_*` functions — one has already been called by *g*. *f*'s caller is then supposed to also return an error indication to *its* caller, again *without* calling `PyErr_*`, and so on — the most detailed cause of the error was already reported by the function that first detected it. Once the error reaches the Python interpreter's main loop, this aborts the currently executing Python code and tries to find an exception handler specified by the Python programmer.

(There are situations where a module can actually give a more detailed error message by calling another `PyErr_*` function, and in such cases it is fine to do so. As a general rule, however, this is not necessary, and can cause information about the cause of the error to be lost: most operations can fail for a variety of reasons.)

To ignore an exception set by a function call that failed, the exception condition must be cleared explicitly by calling

`PyErr_Clear()`. The only time C code should call `PyErr_Clear()` is if it doesn't want to pass the error on to the interpreter but wants to handle it completely by itself (possibly by trying something else, or pretending nothing went wrong).

Every failing **`malloc()`** call must be turned into an exception — the direct caller of **`malloc()`** (or **`realloc()`**) must call `PyErr_NoMemory()` and return a failure indicator itself. All the object-creating functions (for example, `PyLong_FromLong()`) already do this, so this note is only relevant to those who call **`malloc()`** directly.

Also note that, with the important exception of `PyArg_ParseTuple()` and friends, functions that return an integer status usually return a positive value or zero for success and `-1` for failure, like Unix system calls.

Finally, be careful to clean up garbage (by making `Py_XDECREF()` or `Py_DECREF()` calls for objects you have already created) when you return an error indicator!

The choice of which exception to raise is entirely yours. There are predeclared C objects corresponding to all built-in Python exceptions, such as `PyExc_ZeroDivisionError`, which you can use directly. Of course, you should choose exceptions wisely — don't use `PyExc_TypeError` to mean that a file couldn't be opened (that should probably be `PyExc_OSError`). If something's wrong with the argument list, the `PyArg_ParseTuple()` function usually raises `PyExc_TypeError`. If you have an argument whose value must be in a particular range or must satisfy other conditions, `PyExc_ValueError` is appropriate.

You can also define a new exception that is unique to your module. The simplest way to do this is to declare a static global object variable at the beginning of the file:

```
static PyObject *SpamError = NULL;
```

and initialize it by calling `PyErr_NewException()` in the module's `Py_mod_exec` function (**`spam_module_exec()`**):

```
SpamError = PyErr_NewException("spam.error", NULL, NULL);
```

Since **`SpamError`** is a global variable, it will be overwritten every time the module is reinitialized, when the `Py_mod_exec` function is called.

For now, let's avoid the issue: we will block repeated initialization by raising an `ImportError`:

```
static PyObject *SpamError = NULL;

static int
spam_module_exec(PyObject *m)
{
    if (SpamError != NULL) {
        PyErr_SetString(PyExc_ImportError,
                        "cannot initialize spam module more
than once");
        return -1;
    }
    SpamError = PyErr_NewException("spam.error", NULL, NULL);
    if (PyModule_AddObjectRef(m, "SpamError", SpamError) < 0) {
        return -1;
    }

    return 0;
}

static PyModuleDef_Slot spam_module_slots[] = {
    {Py_mod_exec, spam_module_exec},
    {0, NULL}
};

static struct PyModuleDef spam_module = {
    .m_base = PyModuleDef_HEAD_INIT,
    .m_name = "spam",
    .m_size = 0,  // non-negative
    .m_slots = spam_module_slots,
};

PyMODINIT_FUNC
PyInit_spam(void)
{
    return PyModuleDef_Init(&spam_module);
}
```

Note that the Python name for the exception object is **`spam.error`**. The **`PyErr_NewException()`** function may create a class with the base class being **`Exception`** (unless another class is passed in instead of `NULL`), described in Built-in Exceptions.

Note also that the **`SpamError`** variable retains a reference to the newly created exception class; this is intentional! Since the exception could be removed from the module by external code, an owned reference to the class is needed to ensure that it will not be discarded, causing **`SpamError`** to become a dangling pointer. Should it become a dangling pointer, C code which raises the exception could cause a core dump or other unintended side effects.

For now, the **`Py_DECREF()`** call to remove this reference is missing. Even when the Python interpreter shuts down, the global **`SpamError`** variable will not be garbage-collected. It will “leak”. We did, however, ensure that this will happen at most once per process.

We discuss the use of **`PyMODINIT_FUNC`** as a function return type later in this sample.

The **`spam.error`** exception can be raised in your extension module using a call to **`PyErr_SetString()`** as shown below:

```
static PyObject *
spam_system(PyObject *self, PyObject *args)
{
    const char *command;
    int sts;

    if (!PyArg_ParseTuple(args, "s", &command))
        return NULL;
    sts = system(command);
    if (sts < 0) {
        PyErr_SetString(SpamError, "System command failed");
        return NULL;
    }
    return PyLong_FromLong(sts);
}
```

1.3. Back to the Example

Going back to our example function, you should now be able to understand this statement:

```
if (!PyArg_ParseTuple(args, "s", &command))
    return NULL;
```

It returns `NULL` (the error indicator for functions returning object pointers) if an error is detected in the argument list, relying on the exception set by **`PyArg_ParseTuple()`**. Otherwise the string value of the argument has been copied to the local variable **`command`**. This is a pointer assignment and you are not supposed to modify the string to which it points (so in Standard C, the variable **`command`** should properly be declared as `const char *command`).

The next statement is a call to the Unix function **`system()`**, passing it the string we just got from **`PyArg_ParseTuple()`**:

```
sts = system(command);
```

Our **`spam.system()`** function must return the value of **`sts`** as a Python object. This is done using the function **`PyLong_FromLong()`**.

```
return PyLong_FromLong(sts);
```

In this case, it will return an integer object. (Yes, even integers are objects on the heap in Python!)

If you have a C function that returns no useful argument (a function returning void), the corresponding Python function must return `None`. You need this idiom to do so (which is implemented by the **`Py_RETURN_NONE`** macro):

```
Py_INCREF(Py_None);
return Py_None;
```

`Py_None` is the C name for the special Python object `None`. It is a genuine Python object rather than a `NULL` pointer, which means “error” in most contexts, as we have seen.

1.4. The Module’s Method Table and Initialization Function

I promised to show how **`spam_system()`** is called from Python programs. First, we need to list its name and address in a “method table”:

```
static PyMethodDef spam_methods[] = {
    ...
    {"system",  spam_system, METH_VARARGS,
     "Execute a shell command."},
    ...
    {NULL, NULL, 0, NULL}        /* Sentinel */
};
```

Note the third entry (`METH_VARARGS`). This is a flag telling the interpreter the calling convention to be used for the C function. It should normally always be `METH_VARARGS` or `METH_VARARGS | METH_KEYWORDS`; a value of `0` means that an obsolete variant of `PyArg_ParseTuple()` is used.

When using only `METH_VARARGS`, the function should expect the Python-level parameters to be passed in as a tuple acceptable for parsing via `PyArg_ParseTuple()`; more information on this function is provided below.

The **`METH_KEYWORDS`** bit may be set in the third field if keyword arguments should be passed to the function. In this case, the C function should accept a third `PyObject *` parameter which will be a dictionary of keywords. Use `PyArg_ParseTupleAndKeywords()` to parse the arguments to such a function.

The method table must be referenced in the module definition structure:

```
static struct PyModuleDef spam_module = {
    ...
    .m_methods = spam_methods,
    ...
};
```

This structure, in turn, must be passed to the interpreter in the module’s initialization function. The initialization function must be named **`PyInit_name()`**, where *name* is the name of the module, and should be the only non-`static` item defined in the module file:

```
PyMODINIT_FUNC
PyInit_spam(void)
{
    return PyModuleDef_Init(&spam_module);
}
```

Note that `PyMODINIT_FUNC` declares the function as `PyObject *` return type, declares any special linkage declarations required by the platform, and for C++ declares the function as `extern "C"`.

`PyInit_spam()` is called when each interpreter imports its module **`spam`** for the first time. (See below for comments about embedding Python.) A pointer to the module definition must be returned via `PyModuleDef_Init()`, so that the import machinery can create the module and store it in `sys.modules`.

When embedding Python, the **`PyInit_spam()`** function is not called automatically unless there's an entry in the `PyImport_Inittab` table. To add the module to the initialization table, use `PyImport_AppendInittab()`, optionally followed by an import of the module:

```
#define PY_SSIZE_T_CLEAN
#include <Python.h>

int
main(int argc, char *argv[])
{
    PyStatus status;
    PyConfig config;
    PyConfig_InitPythonConfig(&config);

    /* Add a built-in module, before Py_Initialize */
    if (PyImport_AppendInittab("spam", PyInit_spam) == -1) {
        fprintf(stderr, "Error: could not extend in-built
modules table\n");
        exit(1);
    }

    /* Pass argv[0] to the Python interpreter */
    status = PyConfig_SetBytesString(&config,
&config.program_name, argv[0]);
    if (PyStatus_Exception(status)) {
        goto exception;
    }

    /* Initialize the Python interpreter.  Required.
       If this step fails, it will be a fatal error. */
    status = Py_InitializeFromConfig(&config);
    if (PyStatus_Exception(status)) {
        goto exception;
    }
    PyConfig_Clear(&config);

    /* Optionally import the module; alternatively,
       import can be deferred until the embedded script
       imports it. */
    PyObject *pmodule = PyImport_ImportModule("spam");
    if (!pmodule) {
        PyErr_Print();
        fprintf(stderr, "Error: could not import module
'spam'\n");
    }

    // ... use Python C API here ...

    return 0;

  exception:
     PyConfig_Clear(&config);
     Py_ExitStatusException(status);
}
```

Note

If you declare a global variable or a local static one, the module may experience unintended side-effects on re-initialisation, for example when removing entries from `sys.modules` or importing compiled modules into multiple interpreters within a process (or following a **`fork()`** without an intervening **`exec()`**). If module state is not yet fully isolated, authors should consider marking the module as having no support for subinterpreters (via **`Py_MOD_MULTIPLE_INTERPRETERS_NOT_SUPPORTED`**).

A more substantial example module is included in the Python source distribution as `Modules/xxlimited.c`. This file may be used as a template or simply read as an example.

1.5. Compilation and Linkage

There are two more things to do before you can use your new extension: compiling and linking it with the Python system. If you use dynamic loading, the details may depend on the style of dynamic loading your system uses; see the chapters about building extension modules (chapter Building C and C++ Extensions) and additional information that pertains only to building on Windows (chapter Building C and C++ Extensions on Windows) for more information about this.

If you can’t use dynamic loading, or if you want to make your module a permanent part of the Python interpreter, you will have to change the configuration setup and rebuild the interpreter. Luckily, this is very simple on Unix: just place your file (`spammodule.c` for example) in the `Modules/` directory of an unpacked source distribution, add a line to the file `Modules/Setup.local` describing your file:

```
spam spammodule.o
```

and rebuild the interpreter by running **make** in the toplevel directory. You can also run **make** in the `Modules/` subdirectory, but then you must first rebuild `Makefile` there by running ‘**make** Makefile’. (This is necessary each time you change the `Setup` file.)

If your module requires additional libraries to link with, these can be listed on the line in the configuration file as well, for instance:

```
spam spammodule.o -lX11
```

1.6. Calling Python Functions from C

So far we have concentrated on making C functions callable from Python. The reverse is also useful: calling Python functions from C. This is especially the case for libraries that support so-called “callback” functions. If a C interface makes use of callbacks, the equivalent Python often needs to provide a callback mechanism to the Python programmer; the implementation will require calling the Python callback functions from a C callback. Other uses are also imaginable.

Fortunately, the Python interpreter is easily called recursively, and there is a standard interface to call a Python function. (I won’t dwell on how to call the Python parser with a particular string as input — if you’re interested, have a look at the implementation of the `-c` command line option in `Modules/main.c` from the Python source code.)

Calling a Python function is easy. First, the Python program must somehow pass you the Python function object. You should provide a function (or some other interface) to do this. When this function is called, save a pointer to the Python function object (be careful to `Py_INCREF()` it!) in a global variable — or wherever you see fit. For example, the following function might be part of a module definition:

```
static PyObject *my_callback = NULL;

static PyObject *
my_set_callback(PyObject *dummy, PyObject *args)
{
    PyObject *result = NULL;
    PyObject *temp;
    if (PyArg_ParseTuple(args, "O:set_callback", &temp)) {
        if (!PyCallable_Check(temp)) {
            PyErr_SetString(PyExc_TypeError, "parameter must be
callable");
            return NULL;
        }
        Py_XINCREF(temp);         /* Add a reference to new
callback */
        Py_XDECREF(my_callback);  /* Dispose of previous
callback */
        my_callback = temp;       /* Remember new callback */
        /* Boilerplate to return "None" */
        Py_INCREF(Py_None);
        result = Py_None;
    }
    return result;
}
```

This function must be registered with the interpreter using the **`METH_VARARGS`** flag; this is described in section The Module’s Method Table and Initialization Function. The **`PyArg_ParseTuple()`** function and its arguments are documented in section Extracting Parameters in Extension Functions.

The macros **`Py_XINCREF()`** and **`Py_XDECREF()`** increment/decrement the reference count of an object and are safe in the presence of `NULL` pointers (but note that *temp* will not be `NULL` in this context). More info on them in section Reference Counts.

Later, when it is time to call the function, you call the C function **`PyObject_CallObject()`**. This function has two arguments, both pointers to arbitrary Python objects: the Python function, and the argument list. The argument list must always be a tuple object, whose length is the number of arguments. To call the Python function with no arguments, pass in `NULL`, or an empty tuple; to call it with one argument, pass a singleton tuple. **`Py_BuildValue()`** returns a tuple when its format string consists of zero or more format codes between parentheses. For example:

```
int arg;
PyObject *arglist;
PyObject *result;
...
arg = 123;
...
/* Time to call the callback */
arglist = Py_BuildValue("(i)", arg);
result = PyObject_CallObject(my_callback, arglist);
Py_DECREF(arglist);
```

`PyObject_CallObject()` returns a Python object pointer: this is the return value of the Python function. `PyObject_CallObject()` is “reference-count-neutral” with respect to its arguments. In the example a new tuple was created to serve as the argument list, which is `Py_DECREF()`-ed immediately after the `PyObject_CallObject()` call.

The return value of `PyObject_CallObject()` is “new”: either it is a brand new object, or it is an existing object whose reference count has been incremented. So, unless you want to save it in a global variable, you should somehow `Py_DECREF()` the result, even (especially!) if you are not interested in its value.

Before you do this, however, it is important to check that the return value isn’t `NULL`. If it is, the Python function terminated by raising an exception. If the C code that called `PyObject_CallObject()` is called from Python, it should now return an error indication to its Python caller, so the interpreter can print a stack trace, or the calling Python code can handle the exception. If this is not possible or desirable, the exception should be cleared by calling `PyErr_Clear()`. For example:

```
if (result == NULL)
    return NULL; /* Pass error back */
...use result...
Py_DECREF(result);
```

Depending on the desired interface to the Python callback function, you may also have to provide an argument list to `PyObject_CallObject()`. In some cases the argument list is also provided by the Python program, through the same interface that specified the callback function. It can then be saved and used in the same manner as the function object. In other cases, you may have to construct a new tuple to pass as the argument list. The simplest way to do this is to call `Py_BuildValue()`. For example, if you want to pass an integral event code, you might use the following code:

```
PyObject *arglist;
...
arglist = Py_BuildValue("(l)", eventcode);
result = PyObject_CallObject(my_callback, arglist);
Py_DECREF(arglist);
if (result == NULL)
    return NULL; /* Pass error back */
/* Here maybe use the result */
Py_DECREF(result);
```

Note the placement of `Py_DECREF(arglist)` immediately after the call, before the error check! Also note that strictly speaking this code is not complete: `Py_BuildValue()` may run out of memory, and this should be checked.

You may also call a function with keyword arguments by using `PyObject_Call()`, which supports arguments and keyword arguments. As in the above example, we use `Py_BuildValue()` to construct the dictionary.

```
PyObject *dict;
...
dict = Py_BuildValue("{s:i}", "name", val);
```

```
result = PyObject_Call(my_callback, NULL, dict);
Py_DECREF(dict);
if (result == NULL)
    return NULL; /* Pass error back */
/* Here maybe use the result */
Py_DECREF(result);
```

1.7. Extracting Parameters in Extension Functions

The **`PyArg_ParseTuple()`** function is declared as follows:

```
int PyArg_ParseTuple(PyObject *arg, const char *format, ...);
```

The *arg* argument must be a tuple object containing an argument list passed from Python to a C function. The *format* argument must be a format string, whose syntax is explained in Parsing arguments and building values in the Python/C API Reference Manual. The remaining arguments must be addresses of variables whose type is determined by the format string.

Note that while **`PyArg_ParseTuple()`** checks that the Python arguments have the required types, it cannot check the validity of the addresses of C variables passed to the call: if you make mistakes there, your code will probably crash or at least overwrite random bits in memory. So be careful!

Note that any Python object references which are provided to the caller are *borrowed* references; do not decrement their reference count!

Some example calls:

```
#define PY_SSIZE_T_CLEAN
#include <Python.h>
```

```
int ok;
int i, j;
long k, l;
const char *s;
Py_ssize_t size;

ok = PyArg_ParseTuple(args, ""); /* No arguments */
    /* Python call: f() */

ok = PyArg_ParseTuple(args, "s", &s); /* A string */
    /* Possible Python call: f('whoops!') */

ok = PyArg_ParseTuple(args, "lls", &k, &l, &s); /* Two longs
and a string */
    /* Possible Python call: f(1, 2, 'three') */

ok = PyArg_ParseTuple(args, "(ii)s#", &i, &j, &s, &size);
    /* A pair of ints and a string, whose size is also returned
*/
    /* Possible Python call: f((1, 2), 'three') */

{
    const char *file;
    const char *mode = "r";
    int bufsize = 0;
    ok = PyArg_ParseTuple(args, "s|si", &file, &mode,
&bufsize);
    /* A string, and optionally another string and an integer
*/
    /* Possible Python calls:
       f('spam')
       f('spam', 'w')
       f('spam', 'wb', 100000) */
}

{
    int left, top, right, bottom, h, v;
    ok = PyArg_ParseTuple(args, "((ii)(ii))(ii)",
             &left, &top, &right, &bottom, &h, &v);
    /* A rectangle and a point */
    /* Possible Python call:
       f(((0, 0), (400, 300)), (10, 10)) */
}
```

```
{
    Py_complex c;
    ok = PyArg_ParseTuple(args, "D:myfunction", &c);
    /* a complex, also providing a function name for errors */
    /* Possible Python call: myfunction(1+2j) */
}
```

1.8. Keyword Parameters for Extension Functions

The `PyArg_ParseTupleAndKeywords()` function is declared as follows:

```
int PyArg_ParseTupleAndKeywords(PyObject *arg, PyObject
*kwdict,
                                const char *format, char *
const *kwlist, ...);
```

The *arg* and *format* parameters are identical to those of the `PyArg_ParseTuple()` function. The *kwdict* parameter is the dictionary of keywords received as the third parameter from the Python runtime. The *kwlist* parameter is a `NULL`-terminated list of strings which identify the parameters; the names are matched with the type information from *format* from left to right. On success, `PyArg_ParseTupleAndKeywords()` returns true, otherwise it returns false and raises an appropriate exception.

Note

Nested tuples cannot be parsed when using keyword arguments! Keyword parameters passed in which are not present in the *kwlist* will cause `TypeError` to be raised.

Here is an example module which uses keywords, based on an example by Geoff Philbrick (philbrick@hks.com):

```
#define PY_SSIZE_T_CLEAN
#include <Python.h>

static PyObject *
keywdarg_parrot(PyObject *self, PyObject *args, PyObject
*keywds)
{
    int voltage;
    const char *state = "a stiff";
    const char *action = "voom";
    const char *type = "Norwegian Blue";

    static char *kwlist[] = {"voltage", "state", "action",
"type", NULL};

    if (!PyArg_ParseTupleAndKeywords(args, keywds, "i|sss",
kwlist,
                                     &voltage, &state, &action,
&type))
        return NULL;

    printf("-- This parrot wouldn't %s if you put %i Volts
through it.\n",
           action, voltage);
    printf("-- Lovely plumage, the %s -- It's %s!\n", type,
state);

    Py_RETURN_NONE;
}

static PyMethodDef keywdarg_methods[] = {
    /* The cast of the function is necessary since PyCFunction
values
     * only take two PyObject* parameters, and
keywdarg_parrot() takes
     * three.
     */
    {"parrot", (PyCFunction)(void(*)(void))keywdarg_parrot,
METH_VARARGS | METH_KEYWORDS,
     "Print a lovely skit to standard output."},
    {NULL, NULL, 0, NULL}   /* sentinel */
};
```

```
static struct PyModuleDef keywdarg_module = {
    .m_base = PyModuleDef_HEAD_INIT,
    .m_name = "keywdarg",
    .m_size = 0,
    .m_methods = keywdarg_methods,
};

PyMODINIT_FUNC
PyInit_keywdarg(void)
{
    return PyModuleDef_Init(&keywdarg_module);
}
```

1.9. Building Arbitrary Values

This function is the counterpart to `PyArg_ParseTuple()`. It is declared as follows:

```
PyObject *Py_BuildValue(const char *format, ...);
```

It recognizes a set of format units similar to the ones recognized by `PyArg_ParseTuple()`, but the arguments (which are input to the function, not output) must not be pointers, just values. It returns a new Python object, suitable for returning from a C function called from Python.

One difference with `PyArg_ParseTuple()`: while the latter requires its first argument to be a tuple (since Python argument lists are always represented as tuples internally), `Py_BuildValue()` does not always build a tuple. It builds a tuple only if its format string contains two or more format units. If the format string is empty, it returns `None`; if it contains exactly one format unit, it returns whatever object is described by that format unit. To force it to return a tuple of size 0 or one, parenthesize the format string.

Examples (to the left the call, to the right the resulting Python value):

```
Py_BuildValue("")                        None
Py_BuildValue("i", 123)                  123
Py_BuildValue("iii", 123, 456, 789)      (123, 456, 789)
Py_BuildValue("s", "hello")              'hello'
Py_BuildValue("y", "hello")              b'hello'
Py_BuildValue("ss", "hello", "world")    ('hello', 'world')
Py_BuildValue("s#", "hello", 4)          'hell'
Py_BuildValue("y#", "hello", 4)          b'hell'
Py_BuildValue("()")                      ()
Py_BuildValue("(i)", 123)                (123,)
Py_BuildValue("(ii)", 123, 456)          (123, 456)
Py_BuildValue("(i,i)", 123, 456)         (123, 456)
Py_BuildValue("[i,i]", 123, 456)         [123, 456]
Py_BuildValue("{s:i,s:i}",
              "abc", 123, "def", 456)    {'abc': 123, 'def':
456}
Py_BuildValue("((ii)(ii)) (ii)",
              1, 2, 3, 4, 5, 6)          (((1, 2), (3, 4)), (5,
6))
```

1.10. Reference Counts

In languages like C or C++, the programmer is responsible for dynamic allocation and deallocation of memory on the heap. In C, this is done using the functions **`malloc()`** and **`free()`**. In C++, the operators `new` and `delete` are used with essentially the same meaning and we’ll restrict the following discussion to the C case.

Every block of memory allocated with **`malloc()`** should eventually be returned to the pool of available memory by exactly one call to **`free()`**. It is important to call **`free()`** at the right time. If a block’s address is forgotten but **`free()`** is not called for it, the memory it occupies cannot be reused until the program terminates. This is called a *memory leak*. On the other hand, if a program calls **`free()`** for a block and then continues to use the block, it creates a conflict with reuse of the block through another **`malloc()`** call. This is called *using* *freed memory*. It has the same bad consequences as referencing uninitialized data — core dumps, wrong results, mysterious crashes.

Common causes of memory leaks are unusual paths through the code. For instance, a function may allocate a block of memory, do some calculation, and then free the block again. Now a change in the requirements for the function may add a test to the calculation that detects an error condition and can return prematurely from the function. It’s easy to forget to free the allocated memory block when taking this premature exit, especially when it is added later to the code. Such leaks, once introduced, often go undetected for a long time: the error exit is taken only in a small fraction of all calls, and most modern machines have plenty of virtual memory, so the leak only becomes apparent in a long-running process that uses the leaking function frequently. Therefore, it’s important to prevent leaks from happening by having a coding convention or strategy that minimizes this kind of errors.

Since Python makes heavy use of **`malloc()`** and **`free()`**, it needs a strategy to avoid memory leaks as well as the use of freed memory. The chosen method is called *reference counting*. The principle is simple: every object contains a counter, which is incremented when a reference to the object is stored somewhere, and which is decremented when a reference to it is deleted. When the counter reaches zero, the last reference to the object has been deleted and the object is freed.

An alternative strategy is called *automatic garbage collection*. (Sometimes, reference counting is also referred to as a garbage collection strategy, hence my use of “automatic” to distinguish the two.) The big advantage of automatic garbage collection is that the user doesn’t need to call **`free()`** explicitly. (Another claimed advantage is an improvement in speed or memory usage — this is no hard fact however.) The disadvantage is that for C, there is no truly portable automatic garbage collector, while reference counting can be implemented portably (as long as the functions **`malloc()`** and **`free()`** are available — which the C Standard guarantees). Maybe some day a sufficiently portable automatic garbage collector will be available for C. Until then, we'll have to live with reference counts.

While Python uses the traditional reference counting implementation, it also offers a cycle detector that works to detect reference cycles. This allows applications to not worry about creating direct or indirect circular references; these are the weakness of garbage collection implemented using only reference counting. Reference cycles consist of objects which contain (possibly indirect) references to themselves, so that each object in the cycle has a reference count which is non-zero. Typical reference counting implementations are not able to reclaim the memory belonging to any objects in a reference cycle, or referenced from the objects in the cycle, even though there are no further references to the cycle itself.

The cycle detector is able to detect garbage cycles and can reclaim them. The `gc` module exposes a way to run the detector (the `collect()` function), as well as configuration interfaces and the ability to disable the detector at runtime.

1.10.1. Reference Counting in Python

There are two macros, `Py_INCREF(x)` and `Py_DECREF(x)`, which handle the incrementing and decrementing of the reference count. `Py_DECREF()` also frees the object when the count reaches zero. For flexibility, it doesn't call **`free()`** directly — rather, it makes a call through a function pointer in the object’s *type object*. For this purpose (and others), every object also contains a pointer to its type object.

The big question now remains: when to use `Py_INCREF(x)` and `Py_DECREF(x)`? Let’s first introduce some terms. Nobody “owns” an object; however, you can *own a reference* to an object. An object’s reference count is now defined as the number of owned references to it. The owner of a reference is responsible for calling **`Py_DECREF()`** when the reference is no longer needed. Ownership of a reference can be transferred. There are three ways to dispose of an owned reference: pass it on, store it, or call **`Py_DECREF()`**. Forgetting to dispose of an owned reference creates a memory leak.

It is also possible to *borrow* [[2]] a reference to an object. The borrower of a reference should not call **`Py_DECREF()`**. The borrower must not hold on to the object longer than the owner from which it was borrowed. Using a borrowed reference after the owner has disposed of it risks using freed memory and should be avoided completely [[3]].

The advantage of borrowing over owning a reference is that you don’t need to take care of disposing of the reference on all possible paths through the code — in other words, with a borrowed reference you don’t run the risk of leaking when a premature exit is taken. The disadvantage of borrowing over owning is that there are some subtle situations where in seemingly correct code a borrowed reference can be used after the owner from which it was borrowed has in fact disposed of it.

A borrowed reference can be changed into an owned reference by calling **`Py_INCREF()`**. This does not affect the status of the owner from which the reference was borrowed — it creates a new owned reference, and gives full owner responsibilities (the new owner must dispose of the reference properly, as well as the previous owner).

1.10.2. Ownership Rules

Whenever an object reference is passed into or out of a function, it is part of the function’s interface specification whether ownership is transferred with the reference or not.

Most functions that return a reference to an object pass on ownership with the reference. In particular, all functions whose function it is to create a new object, such as `PyLong_FromLong()` and `Py_BuildValue()`, pass ownership to the receiver. Even if the object is not actually new, you still receive ownership of a new reference to that object. For instance, `PyLong_FromLong()` maintains a cache of popular values and can return a reference to a cached item.

Many functions that extract objects from other objects also transfer ownership with the reference, for instance `PyObject_GetAttrString()`. The picture is less clear, here, however, since a few common routines are exceptions: `PyTuple_GetItem()`, `PyList_GetItem()`, `PyDict_GetItem()`, and `PyDict_GetItemString()` all return references that you borrow from the tuple, list or dictionary.

The function `PyImport_AddModule()` also returns a borrowed reference, even though it may actually create the object it returns: this is possible because an owned reference to the object is stored in `sys.modules`.

When you pass an object reference into another function, in general, the function borrows the reference from you — if it needs to store it, it will use `Py_INCREF()` to become an independent owner. There are exactly two important exceptions to this rule: `PyTuple_SetItem()` and `PyList_SetItem()`.

These functions take over ownership of the item passed to them — even if they fail! (Note that `PyDict_SetItem()` and friends don't take over ownership — they are "normal.")

When a C function is called from Python, it borrows references to its arguments from the caller. The caller owns a reference to the object, so the borrowed reference's lifetime is guaranteed until the function returns. Only when such a borrowed reference must be stored or passed on, it must be turned into an owned reference by calling `Py_INCREF()`.

The object reference returned from a C function that is called from Python must be an owned reference — ownership is transferred from the function to its caller.

1.10.3. Thin Ice

There are a few situations where seemingly harmless use of a borrowed reference can lead to problems. These all have to do with implicit invocations of the interpreter, which can cause the owner of a reference to dispose of it.

The first and most important case to know about is using `Py_DECREF()` on an unrelated object while borrowing a reference to a list item. For instance:

```
void
bug(PyObject *list)
{
    PyObject *item = PyList_GetItem(list, 0);

    PyList_SetItem(list, 1, PyLong_FromLong(0L));
    PyObject_Print(item, stdout, 0); /* BUG! */
}
```

This function first borrows a reference to `list[0]`, then replaces `list[1]` with the value `0`, and finally prints the borrowed reference. Looks harmless, right? But it’s not!

Let’s follow the control flow into `PyList_SetItem()`. The list owns references to all its items, so when item 1 is replaced, it has to dispose of the original item 1. Now let’s suppose the original item 1 was an instance of a user-defined class, and let’s further suppose that the class defined a **`__del__()`** method. If this class instance has a reference count of 1, disposing of it will call its **`__del__()`** method. Internally, `PyList_SetItem()` calls `Py_DECREF()` on the replaced item, which invokes replaced item’s corresponding **`tp_dealloc`** function. During deallocation, **`tp_dealloc`** calls **`tp_finalize`**, which is mapped to the **`__del__()`** method for class instances (see **PEP 442** [https://peps.python.org/pep-0442/]). This entire sequence happens synchronously within the `PyList_SetItem()` call.

Since it is written in Python, the **`__del__()`** method can execute arbitrary Python code. Could it perhaps do something to invalidate the reference to `item` in **`bug()`**? You bet! Assuming that the list passed into **`bug()`** is accessible to the **`__del__()`** method, it could execute a statement to the effect of `del list[0]`, and assuming this was the last reference to that object, it would free the memory associated with it, thereby invalidating `item`.

The solution, once you know the source of the problem, is easy: temporarily increment the reference count. The correct version of the function reads:

```
void
no_bug(PyObject *list)
{
    PyObject *item = PyList_GetItem(list, 0);

    Py_INCREF(item);
    PyList_SetItem(list, 1, PyLong_FromLong(0L));
    PyObject_Print(item, stdout, 0);
    Py_DECREF(item);
}
```

This is a true story. An older version of Python contained variants of this bug and someone spent a considerable amount of time in a C debugger to figure out why his **`__del__()`** methods would fail…

The second case of problems with a borrowed reference is a variant involving threads. Normally, multiple threads in the Python interpreter can’t get in each other’s way, because there is a global lock protecting Python’s entire object space. However, it is possible to temporarily release this lock using the macro **`Py_BEGIN_ALLOW_THREADS`**, and to re-acquire it using **`Py_END_ALLOW_THREADS`**. This is common around blocking I/O calls, to let other threads use the processor while waiting for the I/O to complete. Obviously, the following function has the same problem as the previous one:

```
void
bug(PyObject *list)
{
    PyObject *item = PyList_GetItem(list, 0);
    Py_BEGIN_ALLOW_THREADS
    ...some blocking I/O call...
    Py_END_ALLOW_THREADS
    PyObject_Print(item, stdout, 0); /* BUG! */
}
```

1.10.4. NULL Pointers

In general, functions that take object references as arguments do not expect you to pass them `NULL` pointers, and will dump core (or cause later core dumps) if you do so. Functions that return object references generally return `NULL` only to indicate that an exception occurred. The reason for not testing for `NULL` arguments is that functions often pass the objects they receive on to other function — if each function were to test for `NULL`, there would be a lot of redundant tests and the code would run more slowly.

It is better to test for `NULL` only at the “source:” when a pointer that may be `NULL` is received, for example, from **`malloc()`** or from a function that may raise an exception.

The macros **`Py_INCREF()`** and **`Py_DECREF()`** do not check for `NULL` pointers — however, their variants **`Py_XINCREF()`** and **`Py_XDECREF()`** do.

The macros for checking for a particular object type (`Pytype_Check()`) don’t check for `NULL` pointers — again, there is much code that calls several of these in a row to test an object against various different expected types, and this would generate redundant tests. There are no variants with `NULL` checking.

The C function calling mechanism guarantees that the argument list passed to C functions (`args` in the examples) is never `NULL` — in fact it guarantees that it is always a tuple [[4]].

It is a severe error to ever let a `NULL` pointer “escape” to the Python user.

1.11. Writing Extensions in C++

It is possible to write extension modules in C++. Some restrictions apply. If the main program (the Python interpreter) is compiled and linked by the C compiler, global or static objects with constructors cannot be used. This is not a problem if the main program is linked by the C++ compiler. Functions that will be called by the Python interpreter (in particular, module initialization functions) have to be declared using `extern "C"`. It is unnecessary to enclose the Python header files in `extern "C" {...}` — they use this form already if the symbol `__cplusplus` is defined (all recent C++ compilers define this symbol).

1.12. Providing a C API for an Extension Module

Many extension modules just provide new functions and types to be used from Python, but sometimes the code in an extension module can be useful for other extension modules. For example, an extension module could implement a type “collection” which works like lists without order. Just like the standard Python list type has a C API which permits extension modules to create and manipulate lists, this new collection type should have a set of C functions for direct manipulation from other extension modules.

At first sight this seems easy: just write the functions (without declaring them `static`, of course), provide an appropriate header file, and document the C API. And in fact this would work if all extension modules were always linked statically with the Python interpreter. When modules are used as shared libraries, however, the symbols defined in one module may not be visible to another module. The details of visibility depend on the operating system; some systems use one global namespace for the Python interpreter and all extension modules (Windows, for example), whereas others require an explicit list of imported symbols at module link time (AIX is one example), or offer a choice of different strategies (most Unices). And even if symbols are globally visible, the module whose functions one wishes to call might not have been loaded yet!

Portability therefore requires not to make any assumptions about symbol visibility. This means that all symbols in extension modules should be declared `static`, except for the module's initialization function, in order to avoid name clashes with other extension modules (as discussed in section The Module's Method Table and Initialization Function). And it means that symbols that *should* be accessible from other extension modules must be exported in a different way.

Python provides a special mechanism to pass C-level information (pointers) from one extension module to another one: Capsules. A Capsule is a Python data type which stores a pointer (void*). Capsules can only be created and accessed via their C API, but they can be passed around like any other Python object. In particular, they can be assigned to a name in an extension module's namespace. Other extension modules can then import this module, retrieve the value of this name, and then retrieve the pointer from the Capsule.

There are many ways in which Capsules can be used to export the C API of an extension module. Each function could get its own Capsule, or all C API pointers could be stored in an array whose address is published in a Capsule. And the various tasks of storing and retrieving the pointers can be distributed in different ways between the module providing the code and the client modules.

Whichever method you choose, it's important to name your Capsules properly. The function **`PyCapsule_New()`** takes a name parameter (const char*); you're permitted to pass in a `NULL` name, but we strongly encourage you to specify a name. Properly named Capsules provide a degree of runtime type-safety; there is no feasible way to tell one unnamed Capsule from another.

In particular, Capsules used to expose C APIs should be given a name following this convention:

```
modulename.attributename
```

The convenience function `PyCapsule_Import()` makes it easy to load a C API provided via a Capsule, but only if the Capsule’s name matches this convention. This behavior gives C API users a high degree of certainty that the Capsule they load contains the correct C API.

The following example demonstrates an approach that puts most of the burden on the writer of the exporting module, which is appropriate for commonly used library modules. It stores all C API pointers (just one in the example!) in an array of void pointers which becomes the value of a Capsule. The header file corresponding to the module provides a macro that takes care of importing the module and retrieving its C API pointers; client modules only have to call this macro before accessing the C API.

The exporting module is a modification of the **`spam`** module from section A Simple Example. The function **`spam.system()`** does not call the C library function **`system()`** directly, but a function **`PySpam_System()`**, which would of course do something more complicated in reality (such as adding “spam” to every command). This function **`PySpam_System()`** is also exported to other extension modules.

The function **`PySpam_System()`** is a plain C function, declared `static` like everything else:

```
static int
PySpam_System(const char *command)
{
    return system(command);
}
```

The function **`spam_system()`** is modified in a trivial way:

```
static PyObject *
spam_system(PyObject *self, PyObject *args)
{
    const char *command;
    int sts;

    if (!PyArg_ParseTuple(args, "s", &command))
        return NULL;
    sts = PySpam_System(command);
    return PyLong_FromLong(sts);
}
```

In the beginning of the module, right after the line

```
#include <Python.h>
```

two more lines must be added:

```
#define SPAM_MODULE
#include "spammodule.h"
```

The `#define` is used to tell the header file that it is being included in the exporting module, not a client module. Finally, the module’s **`mod_exec`** function must take care of initializing the C API pointer array:

```
static int
spam_module_exec(PyObject *m)
{
    static void *PySpam_API[PySpam_API_pointers];
    PyObject *c_api_object;

    /* Initialize the C API pointer array */
    PySpam_API[PySpam_System_NUM] = (void *)PySpam_System;

    /* Create a Capsule containing the API pointer array's
address */
    c_api_object = PyCapsule_New((void *)PySpam_API,
"spam._C_API", NULL);
```

```
    if (PyModule_Add(m, "_C_API", c_api_object) < 0) {
        return -1;
    }

    return 0;
}
```

Note that `PySpam_API` is declared `static`; otherwise the pointer array would disappear when **`PyInit_spam()`** terminates!

The bulk of the work is in the header file `spammodule.h`, which looks like this:

```
#ifndef Py_SPAMMODULE_H
#define Py_SPAMMODULE_H
#ifdef __cplusplus
extern "C" {
#endif

/* Header file for spammodule */

/* C API functions */
#define PySpam_System_NUM 0
#define PySpam_System_RETURN int
#define PySpam_System_PROTO (const char *command)

/* Total number of C API pointers */
#define PySpam_API_pointers 1


#ifdef SPAM_MODULE
/* This section is used when compiling spammodule.c */

static PySpam_System_RETURN PySpam_System PySpam_System_PROTO;

#else
/* This section is used in modules that use spammodule's API */

static void **PySpam_API;

#define PySpam_System \
 (*(PySpam_System_RETURN (*)PySpam_System_PROTO)
PySpam_API[PySpam_System_NUM])

/* Return -1 on error, 0 on success.
 * PyCapsule_Import will set an exception if there's an error.
 */
static int
import_spam(void)
{
    PySpam_API = (void **)PyCapsule_Import("spam._C_API", 0);
    return (PySpam_API != NULL) ? 0 : -1;
}

#endif

#ifdef __cplusplus
}
#endif

#endif /* !defined(Py_SPAMMODULE_H) */
```

All that a client module must do in order to have access to the function **`PySpam_System()`** is to call the function (or rather macro) **`import_spam()`** in its **`mod_exec`** function:

```
static int
client_module_exec(PyObject *m)
{
    if (import_spam() < 0) {
        return -1;
    }
    /* additional initialization can happen here */
    return 0;
}
```

The main disadvantage of this approach is that the file `spammodule.h` is rather complicated. However, the basic structure is the same for each function that is exported, so it has to be learned only once.

Finally it should be mentioned that Capsules offer additional functionality, which is especially useful for memory allocation and deallocation of the pointer stored in a Capsule. The details are described in the Python/C API Reference Manual in the section Capsules and in the implementation of Capsules (files `Include/pycapsule.h` and `Objects/pycapsule.c` in the Python source code distribution).

Footnotes

[1]

An interface for this function already exists in the standard module **`os`** — it was chosen as a simple and straightforward example.

[2]

The metaphor of “borrowing” a reference is not completely correct: the owner still has a copy of the reference.

[3]

Checking that the reference count is at least 1 **does not work** — the reference count itself could be in freed memory and may thus be reused for another object!

[4]

These guarantees don’t hold when you use the “old” style calling convention — this is still found in much existing code.

2. Defining Extension Types: Tutorial

Python allows the writer of a C extension module to define new types that can be manipulated from Python code, much like the built-in `str` and `list` types. The code for all extension types follows a pattern, but there are some details that you need to understand before you can get started. This document is a gentle introduction to the topic.

2.1. The Basics

The CPython runtime sees all Python objects as variables of type PyObject*, which serves as a “base type” for all Python objects. The `PyObject` structure itself only contains the object’s reference count and a pointer to the object’s “type object”. This is where the action is; the type object determines which (C) functions get called by the interpreter when, for instance, an attribute gets looked up on an object, a method called, or it is multiplied by another object. These C functions are called “type methods”.

So, if you want to define a new extension type, you need to create a new type object.

This sort of thing can only be explained by example, so here’s a minimal, but complete, module that defines a new type named `Custom` inside a C extension module `custom`:

Note

What we’re showing here is the traditional way of defining *static* extension types. It should be adequate for most uses. The C API also allows defining heap-allocated extension types using the `PyType_FromSpec()` function, which isn’t covered in this tutorial.

```
#define PY_SSIZE_T_CLEAN
#include <Python.h>

typedef struct {
    PyObject_HEAD
    /* Type-specific fields go here. */
} CustomObject;

static PyTypeObject CustomType = {
    .ob_base = PyVarObject_HEAD_INIT(NULL, 0)
    .tp_name = "custom.Custom",
    .tp_doc = PyDoc_STR("Custom objects"),
    .tp_basicsize = sizeof(CustomObject),
    .tp_itemsize = 0,
    .tp_flags = Py_TPFLAGS_DEFAULT,
    .tp_new = PyType_GenericNew,
};

static int
custom_module_exec(PyObject *m)
{
    if (PyType_Ready(&CustomType) < 0) {
        return -1;
    }

    if (PyModule_AddObjectRef(m, "Custom", (PyObject *)
&CustomType) < 0) {
        return -1;
    }

    return 0;
}

static PyModuleDef_Slot custom_module_slots[] = {
    {Py_mod_exec, custom_module_exec},
    // Just use this while using static types
    {Py_mod_multiple_interpreters,
Py_MOD_MULTIPLE_INTERPRETERS_NOT_SUPPORTED},
    {0, NULL}
};

static PyModuleDef custom_module = {
    .m_base = PyModuleDef_HEAD_INIT,
    .m_name = "custom",
    .m_doc = "Example module that creates an extension type.",
    .m_size = 0,
    .m_slots = custom_module_slots,
};

PyMODINIT_FUNC
PyInit_custom(void)
{
    return PyModuleDef_Init(&custom_module);
}
```

Now that’s quite a bit to take in at once, but hopefully bits will seem familiar from the previous chapter. This file defines three things:

1. What a **`Custom`** **object** contains: this is the `CustomObject` struct, which is allocated once for each **`Custom`** instance.
2. How the **`Custom`** **type** behaves: this is the `CustomType` struct, which defines a set of flags and function pointers that the interpreter inspects when specific operations are requested.
3. How to define and execute the **`custom`** module: this is the `PyInit_custom` function and the associated `custom_module` struct for defining the module, and the `custom_module_exec` function to set up a fresh module object.

The first bit is:

```
typedef struct {
    PyObject_HEAD
} CustomObject;
```

This is what a Custom object will contain. `PyObject_HEAD` is mandatory at the start of each object struct and defines a field called `ob_base` of type **`PyObject`**, containing a pointer to a type object and a reference count (these can be accessed using the macros **`Py_TYPE`** and **`Py_REFCNT`** respectively). The reason for the macro is to abstract away the layout and to enable additional fields in debug builds.

Note

There is no semicolon above after the **`PyObject_HEAD`** macro. Be wary of adding one by accident: some compilers will complain.

Of course, objects generally store additional data besides the standard `PyObject_HEAD` boilerplate; for example, here is the definition for standard Python floats:

```
typedef struct {
    PyObject_HEAD
    double ob_fval;
} PyFloatObject;
```

The second bit is the definition of the type object.

```
static PyTypeObject CustomType = {
    .ob_base = PyVarObject_HEAD_INIT(NULL, 0)
    .tp_name = "custom.Custom",
    .tp_doc = PyDoc_STR("Custom objects"),
    .tp_basicsize = sizeof(CustomObject),
    .tp_itemsize = 0,
    .tp_flags = Py_TPFLAGS_DEFAULT,
    .tp_new = PyType_GenericNew,
};
```

Note

We recommend using C99-style designated initializers as above, to avoid listing all the **`PyTypeObject`** fields that you don’t care about and also to avoid caring about the fields’ declaration order.

The actual definition of **`PyTypeObject`** in `object.h` has many more fields than the definition above. The remaining fields will be filled with zeros by the C compiler, and it’s common practice to not specify them explicitly unless you need them.

We’re going to pick it apart, one field at a time:

```
.ob_base = PyVarObject_HEAD_INIT(NULL, 0)
```

This line is mandatory boilerplate to initialize the `ob_base` field mentioned above.

```
.tp_name = "custom.Custom",
```

The name of our type. This will appear in the default textual representation of our objects and in some error messages, for example:

```
>>> "" + custom.Custom()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: can only concatenate str (not "custom.Custom") to
str
```

Note that the name is a dotted name that includes both the module name and the name of the type within the module. The module in this case is **`custom`** and the type is **`Custom`**, so we set the type name to **`custom.Custom`**. Using the real dotted import path is important to make your type compatible with the **`pydoc`** and **`pickle`** modules.

```
.tp_basicsize = sizeof(CustomObject),
.tp_itemsize = 0,
```

This is so that Python knows how much memory to allocate when creating new `Custom` instances. `tp_itemsize` is only used for variable-sized objects and should otherwise be zero.

Note

If you want your type to be subclassable from Python, and your type has the same `tp_basicsize` as its base type, you may have problems with multiple inheritance. A Python subclass of your type will have to list your type first in its `__bases__`, or else it will not be able to call your type's `__new__()` method without getting an error. You can avoid this problem by ensuring that your type has a larger value for `tp_basicsize` than its base type does. Most of the time, this will be true anyway, because either your base type will be `object`, or else you will be adding data members to your base type, and therefore increasing its size.

We set the class flags to `Py_TPFLAGS_DEFAULT`.

```
.tp_flags = Py_TPFLAGS_DEFAULT,
```

All types should include this constant in their flags. It enables all of the members defined until at least Python 3.3. If you need further members, you will need to OR the corresponding flags.

We provide a doc string for the type in `tp_doc`.

```
.tp_doc = PyDoc_STR("Custom objects"),
```

To enable object creation, we have to provide a `tp_new` handler. This is the equivalent of the Python method `__new__()`, but has to be specified explicitly. In this case, we can just use the default implementation provided by the API function `PyType_GenericNew()`.

```
.tp_new = PyType_GenericNew,
```

Everything else in the file should be familiar, except for some code in **`custom_module_exec()`**:

```
if (PyType_Ready(&CustomType) < 0) {
    return -1;
}
```

This initializes the **`Custom`** type, filling in a number of members to the appropriate default values, including `ob_type` that we initially set to `NULL`.

```
if (PyModule_AddObjectRef(m, "Custom", (PyObject *)
&CustomType) < 0) {
    return -1;
}
```

This adds the type to the module dictionary. This allows us to create **`Custom`** instances by calling the **`Custom`** class:

```
>>> import custom
>>> mycustom = custom.Custom()
```

That’s it! All that remains is to build it; put the above code in a file called `custom.c`,

```
[build-system]
requires = ["setuptools"]
build-backend = "setuptools.build_meta"

[project]
name = "custom"
version = "1"
```

in a file called `pyproject.toml`, and

```
from setuptools import Extension, setup
setup(ext_modules=[Extension("custom", ["custom.c"])])
```

in a file called `setup.py`; then typing

```
$ python -m pip install .
```

in a shell should produce a file `custom.so` in a subdirectory and install it; now fire up Python — you should be able to `import custom` and play around with `Custom` objects.

That wasn’t so hard, was it?

Of course, the current Custom type is pretty uninteresting. It has no data and doesn’t do anything. It can’t even be subclassed.

2.2. Adding data and methods to the Basic example

Let’s extend the basic example to add some data and methods. Let’s also make the type usable as a base class. We’ll create a new module, **`custom2`** that adds these capabilities:

```
#define PY_SSIZE_T_CLEAN
#include <Python.h>
#include <stddef.h> /* for offsetof() */

typedef struct {
    PyObject_HEAD
    PyObject *first; /* first name */
    PyObject *last;  /* last name */
    int number;
} CustomObject;

static void
Custom_dealloc(PyObject *op)
{
    CustomObject *self = (CustomObject *) op;
    Py_XDECREF(self->first);
    Py_XDECREF(self->last);
    Py_TYPE(self)->tp_free(self);
}

static PyObject *
Custom_new(PyTypeObject *type, PyObject *args, PyObject *kwds)
{
    CustomObject *self;
    self = (CustomObject *) type->tp_alloc(type, 0);
    if (self != NULL) {
        self->first = Py_GetConstant(Py_CONSTANT_EMPTY_STR);
        if (self->first == NULL) {
            Py_DECREF(self);
            return NULL;
        }
        self->last = Py_GetConstant(Py_CONSTANT_EMPTY_STR);
        if (self->last == NULL) {
            Py_DECREF(self);
            return NULL;
        }
        self->number = 0;
    }
    return (PyObject *) self;
}

static int
Custom_init(PyObject *op, PyObject *args, PyObject *kwds)
{
    CustomObject *self = (CustomObject *) op;
    static char *kwlist[] = {"first", "last", "number", NULL};
    PyObject *first = NULL, *last = NULL;

    if (!PyArg_ParseTupleAndKeywords(args, kwds, "|OOi",
kwlist,
                                     &first, &last,
                                     &self->number))
        return -1;

    if (first) {
        Py_XSETREF(self->first, Py_NewRef(first));
    }
    if (last) {
        Py_XSETREF(self->last, Py_NewRef(last));
    }
    return 0;
}

static PyMemberDef Custom_members[] = {
    {"first", Py_T_OBJECT_EX, offsetof(CustomObject, first), 0,
     "first name"},
    {"last", Py_T_OBJECT_EX, offsetof(CustomObject, last), 0,
     "last name"},
    {"number", Py_T_INT, offsetof(CustomObject, number), 0,
     "custom number"},
    {NULL}  /* Sentinel */
};

static PyObject *
Custom_name(PyObject *op, PyObject *Py_UNUSED(dummy))
{
    CustomObject *self = (CustomObject *) op;
    if (self->first == NULL) {
        PyErr_SetString(PyExc_AttributeError, "first");
        return NULL;
    }
    if (self->last == NULL) {
        PyErr_SetString(PyExc_AttributeError, "last");
        return NULL;
    }
    return PyUnicode_FromFormat("%S %S", self->first, self-
>last);
}

static PyMethodDef Custom_methods[] = {
    {"name", Custom_name, METH_NOARGS,
     "Return the name, combining the first and last name"
    },
    {NULL}  /* Sentinel */
};

static PyTypeObject CustomType = {
    .ob_base = PyVarObject_HEAD_INIT(NULL, 0)
    .tp_name = "custom2.Custom",
    .tp_doc = PyDoc_STR("Custom objects"),
    .tp_basicsize = sizeof(CustomObject),
    .tp_itemsize = 0,
    .tp_flags = Py_TPFLAGS_DEFAULT | Py_TPFLAGS_BASETYPE,
    .tp_new = Custom_new,
    .tp_init = Custom_init,
    .tp_dealloc = Custom_dealloc,
    .tp_members = Custom_members,
    .tp_methods = Custom_methods,
};

static int
custom_module_exec(PyObject *m)
{
    if (PyType_Ready(&CustomType) < 0) {
        return -1;
    }

    if (PyModule_AddObjectRef(m, "Custom", (PyObject *)
&CustomType) < 0) {
        return -1;
    }

    return 0;
}

static PyModuleDef_Slot custom_module_slots[] = {
    {Py_mod_exec, custom_module_exec},
    {Py_mod_multiple_interpreters,
Py_MOD_MULTIPLE_INTERPRETERS_NOT_SUPPORTED},
    {0, NULL}
};

static PyModuleDef custom_module = {
    .m_base = PyModuleDef_HEAD_INIT,
    .m_name = "custom2",
    .m_doc = "Example module that creates an extension type.",
    .m_size = 0,
    .m_slots = custom_module_slots,
};

PyMODINIT_FUNC
PyInit_custom2(void)
{
    return PyModuleDef_Init(&custom_module);
}
```

This version of the module has a number of changes.

The **`Custom`** type now has three data attributes in its C struct, *first*, *last*, and *number*. The *first* and *last* variables are Python strings containing first and last names. The *number* attribute is a C integer.

The object structure is updated accordingly:

```
typedef struct {
    PyObject_HEAD
    PyObject *first; /* first name */
    PyObject *last;  /* last name */
    int number;
} CustomObject;
```

Because we now have data to manage, we have to be more careful about object allocation and deallocation. At a minimum, we need a deallocation method:

```
static void
Custom_dealloc(PyObject *op)
{
    CustomObject *self = (CustomObject *) op;
    Py_XDECREF(self->first);
    Py_XDECREF(self->last);
    Py_TYPE(self)->tp_free(self);
}
```

which is assigned to the **`tp_dealloc`** member:

```
.tp_dealloc = Custom_dealloc,
```

This method first clears the reference counts of the two Python attributes. **`Py_XDECREF()`** correctly handles the case where its argument is `NULL` (which might happen here if `tp_new` failed midway). It then calls the **`tp_free`** member of the object’s type (computed by `Py_TYPE(self)`) to free the object’s memory. Note that the object’s type might not be **`CustomType`**, because the object may be an instance of a subclass.

Note

The explicit cast to `CustomObject *` above is needed because we defined `Custom_dealloc` to take a `PyObject *` argument, as the `tp_dealloc` function pointer expects to receive a `PyObject *` argument. By assigning to the `tp_dealloc` slot of a type, we declare that it can only be called with instances of our `CustomObject` class, so the cast to `(CustomObject *)` is safe. This is object-oriented polymorphism, in C!

In existing code, or in previous versions of this tutorial, you might see similar functions take a pointer to the subtype object structure (`CustomObject*`) directly, like this:

```
Custom_dealloc(CustomObject *self)
{
    Py_XDECREF(self->first);
    Py_XDECREF(self->last);
    Py_TYPE(self)->tp_free((PyObject *) self);
}
...
.tp_dealloc = (destructor) Custom_dealloc,
```

This does the same thing on all architectures that CPython supports, but according to the C standard, it invokes undefined behavior.

We want to make sure that the first and last names are initialized to empty strings, so we provide a `tp_new` implementation:

```
static PyObject *
Custom_new(PyTypeObject *type, PyObject *args, PyObject *kwds)
{
    CustomObject *self;
    self = (CustomObject *) type->tp_alloc(type, 0);
    if (self != NULL) {
        self->first = PyUnicode_FromString("");
        if (self->first == NULL) {
            Py_DECREF(self);
            return NULL;
        }
        self->last = PyUnicode_FromString("");
        if (self->last == NULL) {
            Py_DECREF(self);
            return NULL;
        }
        self->number = 0;
    }
    return (PyObject *) self;
}
```

and install it in the **`tp_new`** member:

```
.tp_new = Custom_new,
```

The `tp_new` handler is responsible for creating (as opposed to initializing) objects of the type. It is exposed in Python as the `__new__()` method. It is not required to define a `tp_new` member, and indeed many extension types will simply reuse `PyType_GenericNew()` as done in the first version of the **`Custom`** type above. In this case, we use the `tp_new` handler to initialize the `first` and `last` attributes to non-`NULL` default values.

`tp_new` is passed the type being instantiated (not necessarily `CustomType`, if a subclass is instantiated) and any arguments passed when the type was called, and is expected to return the instance created. `tp_new` handlers always accept positional and keyword arguments, but they often ignore the arguments, leaving the argument handling to initializer (a.k.a. `tp_init` in C or `__init__` in Python) methods.

Note

`tp_new` shouldn't call `tp_init` explicitly, as the interpreter will do it itself.

The `tp_new` implementation calls the `tp_alloc` slot to allocate memory:

```
self = (CustomObject *) type->tp_alloc(type, 0);
```

Since memory allocation may fail, we must check the `tp_alloc` result against `NULL` before proceeding.

Note

We didn't fill the `tp_alloc` slot ourselves. Rather `PyType_Ready()` fills it for us by inheriting it from our base class, which is `object` by default. Most types use the default allocation strategy.

Note

If you are creating a co-operative `tp_new` (one that calls a base type's `tp_new` or `__new__()`), you must *not* try to determine what method to call using method resolution order at runtime. Always statically determine what type you are going to call, and call its `tp_new` directly, or via `type->tp_base->tp_new`. If you do not do this, Python subclasses of your type that also inherit from other Python-defined classes may not work correctly. (Specifically, you may not be able to create instances of such subclasses without getting a `TypeError`.)

We also define an initialization function which accepts arguments to provide initial values for our instance:

```
static int
Custom_init(PyObject *op, PyObject *args, PyObject *kwds)
{
    CustomObject *self = (CustomObject *) op;
    static char *kwlist[] = {"first", "last", "number", NULL};
    PyObject *first = NULL, *last = NULL, *tmp;

    if (!PyArg_ParseTupleAndKeywords(args, kwds, "|OOi",
kwlist,
                                     &first, &last,
                                     &self->number))
        return -1;

    if (first) {
        tmp = self->first;
        Py_INCREF(first);
        self->first = first;
        Py_XDECREF(tmp);
    }
    if (last) {
        tmp = self->last;
        Py_INCREF(last);
        self->last = last;
        Py_XDECREF(tmp);
    }
    return 0;
}
```

by filling the `tp_init` slot.

```
.tp_init = Custom_init,
```

The `tp_init` slot is exposed in Python as the `__init__()` method. It is used to initialize an object after it’s created.

Initializers always accept positional and keyword arguments, and they should return either `0` on success or `-1` on error.

Unlike the `tp_new` handler, there is no guarantee that `tp_init` is called at all (for example, the **`pickle`** module by default doesn't call **`__init__()`** on unpickled instances). It can also be called multiple times. Anyone can call the **`__init__()`** method on our objects. For this reason, we have to be extra careful when assigning the new attribute values. We might be tempted, for example to assign the `first` member like this:

```
if (first) {
    Py_XDECREF(self->first);
    Py_INCREF(first);
    self->first = first;
}
```

But this would be risky. Our type doesn't restrict the type of the `first` member, so it could be any kind of object. It could have a destructor that causes code to be executed that tries to access the `first` member; or that destructor could detach the thread state and let arbitrary code run in other threads that accesses and modifies our object.

To be paranoid and protect ourselves against this possibility, we almost always reassign members before decrementing their reference counts. When don't we have to do this?

- when we absolutely know that the reference count is greater than 1;
- when we know that deallocation of the object [[1]] will neither detach the thread state nor cause any calls back into our type's code;
- when decrementing a reference count in a **`tp_dealloc`** handler on a type which doesn't support cyclic garbage collection [[2]].

We want to expose our instance variables as attributes. There are a number of ways to do that. The simplest way is to define member definitions:

```
static PyMemberDef Custom_members[] = {
    {"first", Py_T_OBJECT_EX, offsetof(CustomObject, first), 0,
     "first name"},
    {"last", Py_T_OBJECT_EX, offsetof(CustomObject, last), 0,
     "last name"},
    {"number", Py_T_INT, offsetof(CustomObject, number), 0,
     "custom number"},
    {NULL}  /* Sentinel */
};
```

and put the definitions in the **`tp_members`** slot:

```
.tp_members = Custom_members,
```

Each member definition has a member name, type, offset, access flags and documentation string. See the Generic Attribute Management section below for details.

A disadvantage of this approach is that it doesn’t provide a way to restrict the types of objects that can be assigned to the Python attributes. We expect the first and last names to be strings, but any Python objects can be assigned. Further, the attributes can be deleted, setting the C pointers to `NULL`. Even though we can make sure the members are initialized to non-`NULL` values, the members can be set to `NULL` if the attributes are deleted.

We define a single method, **`Custom.name()`**, that outputs the objects name as the concatenation of the first and last names.

```
static PyObject *
Custom_name(PyObject *op, PyObject *Py_UNUSED(dummy))
{
    CustomObject *self = (CustomObject *) op;
    if (self->first == NULL) {
        PyErr_SetString(PyExc_AttributeError, "first");
        return NULL;
    }
    if (self->last == NULL) {
        PyErr_SetString(PyExc_AttributeError, "last");
        return NULL;
    }
    return PyUnicode_FromFormat("%S %S", self->first, self-
>last);
}
```

The method is implemented as a C function that takes a **`Custom`** (or **`Custom`** subclass) instance as the first argument. Methods always take an instance as the first argument. Methods often take positional and keyword arguments as well, but in this case we don't take any and don't need to accept a positional argument tuple or keyword argument dictionary. This method is equivalent to the Python method:

```
def name(self):
    return "%s %s" % (self.first, self.last)
```

Note that we have to check for the possibility that our **`first`** and **`last`** members are `NULL`. This is because they can be deleted, in which case they are set to `NULL`. It would be better to prevent deletion of these attributes and to restrict the attribute values to be strings. We'll see how to do that in the next section.

Now that we've defined the method, we need to create an array of method definitions:

```
static PyMethodDef Custom_methods[] = {
    {"name", Custom_name, METH_NOARGS,
     "Return the name, combining the first and last name"
    },
    {NULL}  /* Sentinel */
};
```

(note that we used the `METH_NOARGS` flag to indicate that the method is expecting no arguments other than *self*)

and assign it to the `tp_methods` slot:

```
.tp_methods = Custom_methods,
```

Finally, we'll make our type usable as a base class for subclassing. We've written our methods carefully so far so that they don't make any assumptions about the type of the object being created or used, so all we need to do is to add the `Py_TPFLAGS_BASETYPE` to our class flag definition:

```
.tp_flags = Py_TPFLAGS_DEFAULT | Py_TPFLAGS_BASETYPE,
```

We rename **`PyInit_custom()`** to **`PyInit_custom2()`**, update the module name in the `PyModuleDef` struct, and update the full class name in the `PyTypeObject` struct.

Finally, we update our `setup.py` file to include the new module,

```
from setuptools import Extension, setup
setup(ext_modules=[
    Extension("custom", ["custom.c"]),
    Extension("custom2", ["custom2.c"]),
])
```

and then we re-install so that we can `import custom2`:

```
$ python -m pip install .
```

2.3. Providing finer control over data attributes

In this section, we'll provide finer control over how the **`first`** and **`last`** attributes are set in the **`Custom`** example. In the previous version of our module, the instance variables **`first`** and **`last`** could be set to non-string values or even deleted. We want to make sure that these attributes always contain strings.

```
#define PY_SSIZE_T_CLEAN
#include <Python.h>
#include <stddef.h> /* for offsetof() */

typedef struct {
    PyObject_HEAD
    PyObject *first; /* first name */
    PyObject *last;  /* last name */
    int number;
} CustomObject;

static void
Custom_dealloc(PyObject *op)
{
    CustomObject *self = (CustomObject *) op;
    Py_XDECREF(self->first);
    Py_XDECREF(self->last);
    Py_TYPE(self)->tp_free(self);
}

static PyObject *
Custom_new(PyTypeObject *type, PyObject *args, PyObject *kwds)
{
    CustomObject *self;
    self = (CustomObject *) type->tp_alloc(type, 0);
    if (self != NULL) {
        self->first = Py_GetConstant(Py_CONSTANT_EMPTY_STR);
        if (self->first == NULL) {
            Py_DECREF(self);
            return NULL;
        }
        self->last = Py_GetConstant(Py_CONSTANT_EMPTY_STR);
        if (self->last == NULL) {
            Py_DECREF(self);
            return NULL;
        }
        self->number = 0;
    }
    return (PyObject *) self;
}

static int
Custom_init(PyObject *op, PyObject *args, PyObject *kwds)
{
    CustomObject *self = (CustomObject *) op;
    static char *kwlist[] = {"first", "last", "number", NULL};
    PyObject *first = NULL, *last = NULL;

    if (!PyArg_ParseTupleAndKeywords(args, kwds, "|UUi",
kwlist,
                                     &first, &last,
                                     &self->number))
        return -1;

    if (first) {
        Py_SETREF(self->first, Py_NewRef(first));
    }
    if (last) {
        Py_SETREF(self->last, Py_NewRef(last));
    }
    return 0;
}

static PyMemberDef Custom_members[] = {
    {"number", Py_T_INT, offsetof(CustomObject, number), 0,
     "custom number"},
    {NULL}  /* Sentinel */
};

static PyObject *
Custom_getfirst(PyObject *op, void *closure)
{
    CustomObject *self = (CustomObject *) op;
    return Py_NewRef(self->first);
}

static int
Custom_setfirst(PyObject *op, PyObject *value, void *closure)
{
    CustomObject *self = (CustomObject *) op;
    if (value == NULL) {
        PyErr_SetString(PyExc_TypeError, "Cannot delete the
first attribute");
        return -1;
    }
    if (!PyUnicode_Check(value)) {
        PyErr_SetString(PyExc_TypeError,
                        "The first attribute value must be a
string");
        return -1;
    }
    Py_SETREF(self->first, Py_NewRef(value));
    return 0;
}

static PyObject *
Custom_getlast(PyObject *op, void *closure)
{
    CustomObject *self = (CustomObject *) op;
    return Py_NewRef(self->last);
}

static int
Custom_setlast(PyObject *op, PyObject *value, void *closure)
{
    CustomObject *self = (CustomObject *) op;
    if (value == NULL) {
        PyErr_SetString(PyExc_TypeError, "Cannot delete the
last attribute");
        return -1;
    }
    if (!PyUnicode_Check(value)) {
        PyErr_SetString(PyExc_TypeError,
                        "The last attribute value must be a
string");
        return -1;
    }
    Py_SETREF(self->last, Py_NewRef(value));
    return 0;
}

static PyGetSetDef Custom_getsetters[] = {
    {"first", Custom_getfirst, Custom_setfirst,
     "first name", NULL},
    {"last", Custom_getlast, Custom_setlast,
     "last name", NULL},
    {NULL}  /* Sentinel */
};

static PyObject *
Custom_name(PyObject *op, PyObject *Py_UNUSED(dummy))
{
    CustomObject *self = (CustomObject *) op;
    return PyUnicode_FromFormat("%S %S", self->first, self-
>last);
}

static PyMethodDef Custom_methods[] = {
    {"name", Custom_name, METH_NOARGS,
     "Return the name, combining the first and last name"
    },
    {NULL}  /* Sentinel */
};

static PyTypeObject CustomType = {
    .ob_base = PyVarObject_HEAD_INIT(NULL, 0)
    .tp_name = "custom3.Custom",
    .tp_doc = PyDoc_STR("Custom objects"),
    .tp_basicsize = sizeof(CustomObject),
    .tp_itemsize = 0,
    .tp_flags = Py_TPFLAGS_DEFAULT | Py_TPFLAGS_BASETYPE,
    .tp_new = Custom_new,
    .tp_init = Custom_init,
    .tp_dealloc = Custom_dealloc,
    .tp_members = Custom_members,
    .tp_methods = Custom_methods,
    .tp_getset = Custom_getsetters,
};

static int
custom_module_exec(PyObject *m)
{
    if (PyType_Ready(&CustomType) < 0) {
        return -1;
    }

    if (PyModule_AddObjectRef(m, "Custom", (PyObject *)
&CustomType) < 0) {
        return -1;
    }

    return 0;
}

static PyModuleDef_Slot custom_module_slots[] = {
    {Py_mod_exec, custom_module_exec},
    {Py_mod_multiple_interpreters,
Py_MOD_MULTIPLE_INTERPRETERS_NOT_SUPPORTED},
    {0, NULL}
};

static PyModuleDef custom_module = {
    .m_base = PyModuleDef_HEAD_INIT,
    .m_name = "custom3",
    .m_doc = "Example module that creates an extension type.",
    .m_size = 0,
    .m_slots = custom_module_slots,
};

PyMODINIT_FUNC
PyInit_custom3(void)
{
    return PyModuleDef_Init(&custom_module);
}
```

To provide greater control, over the **`first`** and **`last`** attributes, we’ll use custom getter and setter functions. Here are the functions for getting and setting the **`first`** attribute:

```
static PyObject *
Custom_getfirst(PyObject *op, void *closure)
{
    CustomObject *self = (CustomObject *) op;
    Py_INCREF(self->first);
    return self->first;
}

static int
Custom_setfirst(PyObject *op, PyObject *value, void *closure)
```

```
{
    CustomObject *self = (CustomObject *) op;
    PyObject *tmp;
    if (value == NULL) {
        PyErr_SetString(PyExc_TypeError, "Cannot delete the
first attribute");
        return -1;
    }
    if (!PyUnicode_Check(value)) {
        PyErr_SetString(PyExc_TypeError,
                        "The first attribute value must be a
string");
        return -1;
    }
    tmp = self->first;
    Py_INCREF(value);
    self->first = value;
    Py_DECREF(tmp);
    return 0;
}
```

The getter function is passed a **`Custom`** object and a “closure”, which is a void pointer. In this case, the closure is ignored. (The closure supports an advanced usage in which definition data is passed to the getter and setter. This could, for example, be used to allow a single set of getter and setter functions that decide the attribute to get or set based on data in the closure.)

The setter function is passed the **`Custom`** object, the new value, and the closure. The new value may be `NULL`, in which case the attribute is being deleted. In our setter, we raise an error if the attribute is deleted or if its new value is not a string.

We create an array of **`PyGetSetDef`** structures:

```
static PyGetSetDef Custom_getsetters[] = {
    {"first", Custom_getfirst, Custom_setfirst,
     "first name", NULL},
    {"last", Custom_getlast, Custom_setlast,
     "last name", NULL},
    {NULL}  /* Sentinel */
};
```

and register it in the **`tp_getset`** slot:

```
.tp_getset = Custom_getsetters,
```

The last item in a **`PyGetSetDef`** structure is the “closure” mentioned above. In this case, we aren’t using a closure, so we just pass `NULL`.

We also remove the member definitions for these attributes:

```
static PyMemberDef Custom_members[] = {
    {"number", Py_T_INT, offsetof(CustomObject, number), 0,
     "custom number"},
    {NULL}  /* Sentinel */
};
```

We also need to update the **`tp_init`** handler to only allow strings [[3]] to be passed:

```
static int
Custom_init(PyObject *op, PyObject *args, PyObject *kwds)
{
    CustomObject *self = (CustomObject *) op;
    static char *kwlist[] = {"first", "last", "number", NULL};
    PyObject *first = NULL, *last = NULL, *tmp;

    if (!PyArg_ParseTupleAndKeywords(args, kwds, "|UUi",
kwlist,
                                     &first, &last,
                                     &self->number))
        return -1;

    if (first) {
        tmp = self->first;
        Py_INCREF(first);
        self->first = first;
        Py_DECREF(tmp);
    }
    if (last) {
        tmp = self->last;
        Py_INCREF(last);
        self->last = last;
        Py_DECREF(tmp);
    }
    return 0;
}
```

With these changes, we can assure that the `first` and `last` members are never `NULL` so we can remove checks for `NULL` values in almost all cases. This means that most of the `Py_XDECREF()` calls can be converted to `Py_DECREF()` calls. The only place we can't change these calls is in the `tp_dealloc` implementation, where there is the possibility that the initialization of these members failed in `tp_new`.

We also rename the module initialization function and module name in the initialization function, as we did before, and we add an extra definition to the `setup.py` file.

2.4. Supporting cyclic garbage collection

Python has a cyclic garbage collector (GC) that can identify unneeded objects even when their reference counts are not zero. This can happen when objects are involved in cycles. For example, consider:

```
>>> l = []
>>> l.append(l)
>>> del l
```

In this example, we create a list that contains itself. When we delete it, it still has a reference from itself. Its reference count doesn't drop to zero. Fortunately, Python's cyclic garbage collector will eventually figure out that the list is garbage and free it.

In the second version of the **`Custom`** example, we allowed any kind of object to be stored in the **`first`** or **`last`** attributes [[4]]. Besides, in the second and third versions, we allowed subclassing **`Custom`**, and subclasses may add arbitrary attributes. For any of those two reasons, **`Custom`** objects can participate in cycles:

```
>>> import custom3
>>> class Derived(custom3.Custom): pass
...
>>> n = Derived()
>>> n.some_attribute = n
```

To allow a **`Custom`** instance participating in a reference cycle to be properly detected and collected by the cyclic GC, our **`Custom`** type needs to fill two additional slots and to enable a flag that enables these slots:

```
#define PY_SSIZE_T_CLEAN
#include <Python.h>
#include <stddef.h> /* for offsetof() */

typedef struct {
    PyObject_HEAD
    PyObject *first; /* first name */
    PyObject *last;  /* last name */
    int number;
} CustomObject;

static int
Custom_traverse(PyObject *op, visitproc visit, void *arg)
{
    CustomObject *self = (CustomObject *) op;
    Py_VISIT(self->first);
    Py_VISIT(self->last);
    return 0;
}
```

```
static int
Custom_clear(PyObject *op)
{
    CustomObject *self = (CustomObject *) op;
    Py_CLEAR(self->first);
    Py_CLEAR(self->last);
    return 0;
}

static void
Custom_dealloc(PyObject *op)
{
    PyObject_GC_UnTrack(op);
    (void)Custom_clear(op);
    Py_TYPE(op)->tp_free(op);
}

static PyObject *
Custom_new(PyTypeObject *type, PyObject *args, PyObject *kwds)
{
    CustomObject *self;
    self = (CustomObject *) type->tp_alloc(type, 0);
    if (self != NULL) {
        self->first = Py_GetConstant(Py_CONSTANT_EMPTY_STR);
        if (self->first == NULL) {
            Py_DECREF(self);
            return NULL;
        }
        self->last = Py_GetConstant(Py_CONSTANT_EMPTY_STR);
        if (self->last == NULL) {
            Py_DECREF(self);
            return NULL;
        }
        self->number = 0;
    }
    return (PyObject *) self;
}

static int
Custom_init(PyObject *op, PyObject *args, PyObject *kwds)
{
    CustomObject *self = (CustomObject *) op;
    static char *kwlist[] = {"first", "last", "number", NULL};
    PyObject *first = NULL, *last = NULL;

    if (!PyArg_ParseTupleAndKeywords(args, kwds, "|UUi",
kwlist,
                                     &first, &last,
                                     &self->number))
        return -1;

    if (first) {
        Py_SETREF(self->first, Py_NewRef(first));
    }
    if (last) {
        Py_SETREF(self->last, Py_NewRef(last));
    }
    return 0;
}

static PyMemberDef Custom_members[] = {
    {"number", Py_T_INT, offsetof(CustomObject, number), 0,
     "custom number"},
    {NULL}  /* Sentinel */
};

static PyObject *
Custom_getfirst(PyObject *op, void *closure)
{
    CustomObject *self = (CustomObject *) op;
    return Py_NewRef(self->first);
}

static int
Custom_setfirst(PyObject *op, PyObject *value, void *closure)
{
    CustomObject *self = (CustomObject *) op;
    if (value == NULL) {
        PyErr_SetString(PyExc_TypeError, "Cannot delete the
first attribute");
        return -1;
    }
    if (!PyUnicode_Check(value)) {
        PyErr_SetString(PyExc_TypeError,
                        "The first attribute value must be a
string");
        return -1;
    }
    Py_XSETREF(self->first, Py_NewRef(value));
    return 0;
}

static PyObject *
Custom_getlast(PyObject *op, void *closure)
{
    CustomObject *self = (CustomObject *) op;
    return Py_NewRef(self->last);
}

static int
Custom_setlast(PyObject *op, PyObject *value, void *closure)
{
    CustomObject *self = (CustomObject *) op;
    if (value == NULL) {
        PyErr_SetString(PyExc_TypeError, "Cannot delete the
last attribute");
        return -1;
    }
    if (!PyUnicode_Check(value)) {
        PyErr_SetString(PyExc_TypeError,
                        "The last attribute value must be a
string");
        return -1;
    }
    Py_XSETREF(self->last, Py_NewRef(value));
    return 0;
}

static PyGetSetDef Custom_getsetters[] = {
    {"first", Custom_getfirst, Custom_setfirst,
     "first name", NULL},
    {"last", Custom_getlast, Custom_setlast,
     "last name", NULL},
    {NULL}  /* Sentinel */
};

static PyObject *
Custom_name(PyObject *op, PyObject *Py_UNUSED(dummy))
```

```
{
    CustomObject *self = (CustomObject *) op;
    return PyUnicode_FromFormat("%S %S", self->first, self-
>last);
}

static PyMethodDef Custom_methods[] = {
    {"name", Custom_name, METH_NOARGS,
     "Return the name, combining the first and last name"
    },
    {NULL}  /* Sentinel */
};

static PyTypeObject CustomType = {
    .ob_base = PyVarObject_HEAD_INIT(NULL, 0)
    .tp_name = "custom4.Custom",
    .tp_doc = PyDoc_STR("Custom objects"),
    .tp_basicsize = sizeof(CustomObject),
    .tp_itemsize = 0,
    .tp_flags = Py_TPFLAGS_DEFAULT | Py_TPFLAGS_BASETYPE |
Py_TPFLAGS_HAVE_GC,
    .tp_new = Custom_new,
    .tp_init = Custom_init,
    .tp_dealloc = Custom_dealloc,
    .tp_traverse = Custom_traverse,
    .tp_clear = Custom_clear,
    .tp_members = Custom_members,
    .tp_methods = Custom_methods,
    .tp_getset = Custom_getsetters,
};

static int
custom_module_exec(PyObject *m)
{
    if (PyType_Ready(&CustomType) < 0) {
        return -1;
    }

    if (PyModule_AddObjectRef(m, "Custom", (PyObject *)
&CustomType) < 0) {
        return -1;
    }
    return 0;
}

static PyModuleDef_Slot custom_module_slots[] = {
    {Py_mod_exec, custom_module_exec},
    {Py_mod_multiple_interpreters,
Py_MOD_MULTIPLE_INTERPRETERS_NOT_SUPPORTED},
    {0, NULL}
};

static PyModuleDef custom_module = {
    .m_base = PyModuleDef_HEAD_INIT,
    .m_name = "custom4",
    .m_doc = "Example module that creates an extension type.",
    .m_size = 0,
    .m_slots = custom_module_slots,
};

PyMODINIT_FUNC
PyInit_custom4(void)
{
    return PyModuleDef_Init(&custom_module);
}
```

First, the traversal method lets the cyclic GC know about subobjects that could participate in cycles:

```
static int
Custom_traverse(PyObject *op, visitproc visit, void *arg)
{
    CustomObject *self = (CustomObject *) op;
    int vret;
    if (self->first) {
        vret = visit(self->first, arg);
        if (vret != 0)
            return vret;
    }
    if (self->last) {
        vret = visit(self->last, arg);
        if (vret != 0)
            return vret;
    }
    return 0;
}
```

For each subobject that can participate in cycles, we need to call the **`visit()`** function, which is passed to the traversal method. The **`visit()`** function takes as arguments the subobject and the extra argument *arg* passed to the traversal method. It returns an integer value that must be returned if it is non-zero.

Python provides a **`Py_VISIT()`** macro that automates calling visit functions. With **`Py_VISIT()`**, we can minimize the amount of boilerplate in `Custom_traverse`:

```
static int
Custom_traverse(PyObject *op, visitproc visit, void *arg)
{
    CustomObject *self = (CustomObject *) op;
    Py_VISIT(self->first);
    Py_VISIT(self->last);
    return 0;
}
```

Note

The **`tp_traverse`** implementation must name its arguments exactly *visit* and *arg* in order to use **`Py_VISIT()`**.

Second, we need to provide a method for clearing any subobjects that can participate in cycles:

```
static int
Custom_clear(PyObject *op)
{
    CustomObject *self = (CustomObject *) op;
    Py_CLEAR(self->first);
    Py_CLEAR(self->last);
    return 0;
}
```

Notice the use of the `Py_CLEAR()` macro. It is the recommended and safe way to clear data attributes of arbitrary types while decrementing their reference counts. If you were to call `Py_XDECREF()` instead on the attribute before setting it to `NULL`, there is a possibility that the attribute's destructor would call back into code that reads the attribute again (*especially* if there is a reference cycle).

Note

You could emulate `Py_CLEAR()` by writing:

```
PyObject *tmp;
tmp = self->first;
self->first = NULL;
Py_XDECREF(tmp);
```

Nevertheless, it is much easier and less error-prone to always use `Py_CLEAR()` when deleting an attribute. Don't try to micro-optimize at the expense of robustness!

The deallocator `Custom_dealloc` may call arbitrary code when clearing attributes. It means the circular GC can be triggered inside the function. Since the GC assumes reference count is not zero, we need to untrack the object from the GC by calling `PyObject_GC_UnTrack()` before clearing members. Here is our reimplemented deallocator using `PyObject_GC_UnTrack()` and `Custom_clear`:

```
static void
Custom_dealloc(PyObject *op)
{
    PyObject_GC_UnTrack(op);
    (void)Custom_clear(op);
    Py_TYPE(op)->tp_free(op);
}
```

Finally, we add the `Py_TPFLAGS_HAVE_GC` flag to the class flags:

```
.tp_flags = Py_TPFLAGS_DEFAULT | Py_TPFLAGS_BASETYPE |
Py_TPFLAGS_HAVE_GC,
```

That’s pretty much it. If we had written custom `tp_alloc` or `tp_free` handlers, we’d need to modify them for cyclic garbage collection. Most extensions will use the versions automatically provided.

2.5. Subclassing other types

It is possible to create new extension types that are derived from existing types. It is easiest to inherit from the built in types, since an extension can easily use the `PyTypeObject` it needs. It can be difficult to share these `PyTypeObject` structures between extension modules.

In this example we will create a **`SubList`** type that inherits from the built-in `list` type. The new type will be completely compatible with regular lists, but will have an additional **`increment()`** method that increases an internal counter:

```
>>> import sublist
>>> s = sublist.SubList(range(3))
>>> s.extend(s)
>>> print(len(s))
6
>>> print(s.increment())
1
>>> print(s.increment())
2
```

```
#define PY_SSIZE_T_CLEAN
#include <Python.h>
```

```
typedef struct {
    PyListObject list;
    int state;
} SubListObject;

static PyObject *
SubList_increment(PyObject *op, PyObject *Py_UNUSED(dummy))
{
    SubListObject *self = (SubListObject *) op;
    self->state++;
    return PyLong_FromLong(self->state);
}

static PyMethodDef SubList_methods[] = {
    {"increment", SubList_increment, METH_NOARGS,
     PyDoc_STR("increment state counter")},
    {NULL},
};

static int
SubList_init(PyObject *op, PyObject *args, PyObject *kwds)
{
    SubListObject *self = (SubListObject *) op;
    if (PyList_Type.tp_init(op, args, kwds) < 0)
        return -1;
    self->state = 0;
    return 0;
}

static PyTypeObject SubListType = {
    .ob_base = PyVarObject_HEAD_INIT(NULL, 0)
    .tp_name = "sublist.SubList",
    .tp_doc = PyDoc_STR("SubList objects"),
    .tp_basicsize = sizeof(SubListObject),
    .tp_itemsize = 0,
    .tp_flags = Py_TPFLAGS_DEFAULT | Py_TPFLAGS_BASETYPE,
    .tp_init = SubList_init,
    .tp_methods = SubList_methods,
};

static int
sublist_module_exec(PyObject *m)
{
    SubListType.tp_base = &PyList_Type;
    if (PyType_Ready(&SubListType) < 0) {
        return -1;
    }

    if (PyModule_AddObjectRef(m, "SubList", (PyObject *)
&SubListType) < 0) {
        return -1;
    }

    return 0;
}

static PyModuleDef_Slot sublist_module_slots[] = {
    {Py_mod_exec, sublist_module_exec},
    {Py_mod_multiple_interpreters,
Py_MOD_MULTIPLE_INTERPRETERS_NOT_SUPPORTED},
    {0, NULL}
};

static PyModuleDef sublist_module = {
    .m_base = PyModuleDef_HEAD_INIT,
    .m_name = "sublist",
    .m_doc = "Example module that creates an extension type.",
    .m_size = 0,
    .m_slots = sublist_module_slots,
};

PyMODINIT_FUNC
PyInit_sublist(void)
{
    return PyModuleDef_Init(&sublist_module);
}
```

As you can see, the source code closely resembles the **`Custom`** examples in previous sections. We will break down the main differences between them.

```
typedef struct {
    PyListObject list;
    int state;
} SubListObject;
```

The primary difference for derived type objects is that the base type’s object structure must be the first value. The base type will already include the `PyObject_HEAD()` at the beginning of its structure.

When a Python object is a **`SubList`** instance, its `PyObject *` pointer can be safely cast to both `PyListObject *` and `SubListObject *`:

```
static int
SubList_init(PyObject *op, PyObject *args, PyObject *kwds)
{
    SubListObject *self = (SubListObject *) op;
    if (PyList_Type.tp_init(op, args, kwds) < 0)
        return -1;
    self->state = 0;
    return 0;
}
```

We see above how to call through to the `__init__()` method of the base type.

This pattern is important when writing a type with custom `tp_new` and `tp_dealloc` members. The `tp_new` handler should not actually create the memory for the object with its `tp_alloc`, but let the base class handle it by calling its own `tp_new`.

The `PyTypeObject` struct supports a `tp_base` specifying the type’s concrete base class. Due to cross-platform compiler issues, you can’t fill that field directly with a reference to `PyList_Type`; it should be done in the `Py_mod_exec` function:

```
static int
sublist_module_exec(PyObject *m)
{
    SubListType.tp_base = &PyList_Type;
    if (PyType_Ready(&SubListType) < 0) {
        return -1;
    }

    if (PyModule_AddObjectRef(m, "SubList", (PyObject *)
&SubListType) < 0) {
        return -1;
    }

    return 0;
}
```

Before calling `PyType_Ready()`, the type structure must have the `tp_base` slot filled in. When we are deriving an existing type, it is not necessary to fill out the `tp_alloc` slot with `PyType_GenericNew()` – the allocation function from the base type will be inherited.

After that, calling `PyType_Ready()` and adding the type object to the module is the same as with the basic **`Custom`** examples.

Footnotes

[1]

This is true when we know that the object is a basic type, like a string or a float.

[2]

We relied on this in the `tp_dealloc` handler in this example, because our type doesn't support garbage collection.

[3]

We now know that the first and last members are strings, so perhaps we could be less careful about decrementing their reference counts, however, we accept instances of string subclasses. Even though deallocating normal strings won't

call back into our objects, we can’t guarantee that deallocating an instance of a string subclass won’t call back into our objects.
[4]

Also, even with our attributes restricted to strings instances, the user could pass arbitrary `str` subclasses and therefore still create reference cycles.

3. Defining Extension Types: Assorted Topics

This section aims to give a quick fly-by on the various type methods you can implement and what they do.

Here is the definition of **`PyTypeObject`**, with some fields only used in debug builds omitted:

```
typedef struct _typeobject {
    PyObject_VAR_HEAD
    const char *tp_name; /* For printing, in format "<module>.
<name>" */
    Py_ssize_t tp_basicsize, tp_itemsize; /* For allocation */

    /* Methods to implement standard operations */

    destructor tp_dealloc;
    Py_ssize_t tp_vectorcall_offset;
    getattrfunc tp_getattr;
    setattrfunc tp_setattr;
    PyAsyncMethods *tp_as_async; /* formerly known as
tp_compare (Python 2)
                                    or tp_reserved (Python 3)
*/
    reprfunc tp_repr;

    /* Method suites for standard classes */

    PyNumberMethods *tp_as_number;
    PySequenceMethods *tp_as_sequence;
    PyMappingMethods *tp_as_mapping;

    /* More standard operations (here for binary compatibility)
*/

    hashfunc tp_hash;
    ternaryfunc tp_call;
    reprfunc tp_str;
    getattrofunc tp_getattro;
    setattrofunc tp_setattro;

    /* Functions to access object as input/output buffer */
    PyBufferProcs *tp_as_buffer;

    /* Flags to define presence of optional/expanded features
*/
    unsigned long tp_flags;

    const char *tp_doc; /* Documentation string */

    /* Assigned meaning in release 2.0 */
    /* call function for all accessible objects */
    traverseproc tp_traverse;

    /* delete references to contained objects */
    inquiry tp_clear;

    /* Assigned meaning in release 2.1 */
    /* rich comparisons */
    richcmpfunc tp_richcompare;

    /* weak reference enabler */
    Py_ssize_t tp_weaklistoffset;

    /* Iterators */
    getiterfunc tp_iter;
    iternextfunc tp_iternext;

    /* Attribute descriptor and subclassing stuff */
    PyMethodDef *tp_methods;
    PyMemberDef *tp_members;
    PyGetSetDef *tp_getset;
    // Strong reference on a heap type, borrowed reference on a
static type
    PyTypeObject *tp_base;
    PyObject *tp_dict;
    descrgetfunc tp_descr_get;
    descrsetfunc tp_descr_set;
    Py_ssize_t tp_dictoffset;
    initproc tp_init;
    allocfunc tp_alloc;
    newfunc tp_new;
    freefunc tp_free; /* Low-level free-memory routine */
    inquiry tp_is_gc; /* For PyObject_IS_GC */
    PyObject *tp_bases;
    PyObject *tp_mro; /* method resolution order */
    PyObject *tp_cache; /* no longer used */
    void *tp_subclasses;  /* for static builtin types this is
an index */
    PyObject *tp_weaklist; /* not used for static builtin types
*/
    destructor tp_del;

    /* Type attribute cache version tag. Added in version 2.6.
     * If zero, the cache is invalid and must be initialized.
     */
    unsigned int tp_version_tag;

    destructor tp_finalize;
    vectorcallfunc tp_vectorcall;

    /* bitset of which type-watchers care about this type */
    unsigned char tp_watched;

    /* Number of tp_version_tag values used.
     * Set to _Py_ATTR_CACHE_UNUSED if the attribute cache is
     * disabled for this type (e.g. due to custom MRO entries).
     * Otherwise, limited to MAX_VERSIONS_PER_CLASS (defined
elsewhere).
     */
    uint16_t tp_versions_used;
} PyTypeObject;
```

Now that’s a *lot* of methods. Don’t worry too much though – if you have a type you want to define, the chances are very good that you will only implement a handful of these.

As you probably expect by now, we’re going to go over this and give more information about the various handlers. We won’t go in the order they are defined in the structure, because there is a lot of historical baggage that impacts the ordering of the fields. It’s often easiest to find an example that includes the fields you need and then change the values to suit your new type.

```
const char *tp_name; /* For printing */
```

The name of the type – as mentioned in the previous chapter, this will appear in various places, almost entirely for diagnostic purposes. Try to choose something that will be helpful in such a situation!

```
Py_ssize_t tp_basicsize, tp_itemsize; /* For allocation */
```

These fields tell the runtime how much memory to allocate when new objects of this type are created. Python has some built-in support for variable length structures (think: strings, tuples) which is where the `tp_itemsize` field comes in. This will be dealt with later.

```
const char *tp_doc;
```

Here you can put a string (or its address) that you want returned when the Python script references `obj.__doc__` to retrieve the doc string.

Now we come to the basic type methods – the ones most extension types will implement.

3.1. Finalization and De-allocation

```
destructor tp_dealloc;
```

This function is called when the reference count of the instance of your type is reduced to zero and the Python interpreter wants to reclaim it. If your type has memory to free or other clean-up to perform, you can put it here. The object itself needs to be freed here as well. Here is an example of this function:

```
static void
newdatatype_dealloc(PyObject *op)
{
    newdatatypeobject *self = (newdatatypeobject *) op;
    free(self->obj_UnderlyingDatatypePtr);
    Py_TYPE(self)->tp_free(self);
}
```

If your type supports garbage collection, the destructor should call `PyObject_GC_UnTrack()` before clearing any member fields:

```
static void
newdatatype_dealloc(PyObject *op)
{
    newdatatypeobject *self = (newdatatypeobject *) op;
    PyObject_GC_UnTrack(op);
    Py_CLEAR(self->other_obj);
    ...
    Py_TYPE(self)->tp_free(self);
}
```

One important requirement of the deallocator function is that it leaves any pending exceptions alone. This is important since deallocators are frequently called as the interpreter unwinds the Python stack; when the stack is unwound due to an exception (rather than normal returns), nothing is done to protect the deallocators from seeing that an exception has already been set. Any actions which a deallocator performs which may cause additional Python code to be executed may detect that an exception has been set. This can lead to misleading errors from the interpreter. The proper way to protect against this is to save a pending exception before performing the unsafe action, and restoring it when done. This can be done using the `PyErr_Fetch()` and `PyErr_Restore()` functions:

```
static void
my_dealloc(PyObject *obj)
{
    MyObject *self = (MyObject *) obj;
    PyObject *cbresult;

    if (self->my_callback != NULL) {
        PyObject *err_type, *err_value, *err_traceback;

        /* This saves the current exception state */
        PyErr_Fetch(&err_type, &err_value, &err_traceback);

        cbresult = PyObject_CallNoArgs(self->my_callback);
        if (cbresult == NULL) {
           PyErr_WriteUnraisable(self->my_callback);
        }
        else {
            Py_DECREF(cbresult);
        }

        /* This restores the saved exception state */
        PyErr_Restore(err_type, err_value, err_traceback);

        Py_DECREF(self->my_callback);
    }
    Py_TYPE(self)->tp_free(self);
}
```

Note

There are limitations to what you can safely do in a deallocator function. First, if your type supports garbage collection (using `tp_traverse` and/or `tp_clear`), some of the object's members can have been cleared or finalized by the time `tp_dealloc` is called. Second, in `tp_dealloc`, your object is in an unstable state: its reference count is equal to zero. Any call to a non-trivial object or API (as in the example above) might end up calling `tp_dealloc` again, causing a double free and a crash.

Starting with Python 3.4, it is recommended not to put any complex finalization code in `tp_dealloc`, and instead use the new `tp_finalize` type method.

See also

PEP 442 [https://peps.python.org/pep-0442/] explains the new finalization scheme.

3.2. Object Presentation

In Python, there are two ways to generate a textual representation of an object: the `repr()` function, and the `str()` function. (The `print()` function just calls `str()`.) These handlers are both optional.

```
reprfunc tp_repr;
reprfunc tp_str;
```

The `tp_repr` handler should return a string object containing a representation of the instance for which it is called. Here is a simple example:

```
static PyObject *
newdatatype_repr(PyObject *op)
{
    newdatatypeobject *self = (newdatatypeobject *) op;
    return PyUnicode_FromFormat("Repr-
ified_newdatatype{{size:%d}}",
                                self-
>obj_UnderlyingDatatypePtr->size);
}
```

If no `tp_repr` handler is specified, the interpreter will supply a representation that uses the type’s `tp_name` and a uniquely identifying value for the object.

The `tp_str` handler is to `str()` what the `tp_repr` handler described above is to `repr()`; that is, it is called when Python code calls `str()` on an instance of your object. Its implementation is very similar to the `tp_repr` function, but the resulting string is intended for human consumption. If `tp_str` is not specified, the `tp_repr` handler is used instead.

Here is a simple example:

```
static PyObject *
newdatatype_str(PyObject *op)
{
    newdatatypeobject *self = (newdatatypeobject *) op;
    return
PyUnicode_FromFormat("Stringified_newdatatype{{size:%d}}",
                                self-
>obj_UnderlyingDatatypePtr->size);
}
```

3.3. Attribute Management

For every object which can support attributes, the corresponding type must provide the functions that control how the attributes are resolved. There needs to be a function which can retrieve attributes (if any are defined), and another to set attributes (if setting attributes is allowed). Removing an attribute is a special case, for which the new value passed to the handler is `NULL`.

Python supports two pairs of attribute handlers; a type that supports attributes only needs to implement the functions for one pair. The difference is that one pair takes the name of the attribute as a char*, while the other accepts a PyObject*. Each type can use whichever pair makes more sense for the implementation’s convenience.

```
getattrfunc  tp_getattr;        /* char * version */
setattrfunc  tp_setattr;
```

```
/* ... */
getattrofunc tp_getattro;       /* PyObject * version */
setattrofunc tp_setattro;
```

If accessing attributes of an object is always a simple operation (this will be explained shortly), there are generic implementations which can be used to provide the PyObject* version of the attribute management functions. The actual need for type-specific attribute handlers almost completely disappeared starting with Python 2.2, though there are many examples which have not been updated to use some of the new generic mechanism that is available.

3.3.1. Generic Attribute Management

Most extension types only use *simple* attributes. So, what makes the attributes simple? There are only a couple of conditions that must be met:

1. The name of the attributes must be known when **`PyType_Ready()`** is called.
2. No special processing is needed to record that an attribute was looked up or set, nor do actions need to be taken based on the value.

Note that this list does not place any restrictions on the values of the attributes, when the values are computed, or how relevant data is stored.

When **`PyType_Ready()`** is called, it uses three tables referenced by the type object to create descriptors which are placed in the dictionary of the type object. Each descriptor controls access to one attribute of the instance object. Each of the tables is optional; if all three are `NULL`, instances of the type will only have attributes that are inherited from their base type, and should leave the **`tp_getattro`** and **`tp_setattro`** fields `NULL` as well, allowing the base type to handle attributes.

The tables are declared as three fields of the type object:

```
struct PyMethodDef *tp_methods;
struct PyMemberDef *tp_members;
struct PyGetSetDef *tp_getset;
```

If **`tp_methods`** is not `NULL`, it must refer to an array of **`PyMethodDef`** structures. Each entry in the table is an instance of this structure:

```
typedef struct PyMethodDef {
    const char  *ml_name;       /* method name */
    PyCFunction  ml_meth;       /* implementation function */
    int          ml_flags;      /* flags */
    const char  *ml_doc;        /* docstring */
} PyMethodDef;
```

One entry should be defined for each method provided by the type; no entries are needed for methods inherited from a base type. One additional entry is needed at the end; it is a sentinel that marks the end of the array. The **`ml_name`** field of the sentinel must be `NULL`.

The second table is used to define attributes which map directly to data stored in the instance. A variety of primitive C types are supported, and access may be read-only or read-write. The structures in the table are defined as:

```
typedef struct PyMemberDef {
    const char *name;
    int         type;
    int         offset;
    int         flags;
    const char *doc;
} PyMemberDef;
```

For each entry in the table, a descriptor will be constructed and added to the type which will be able to extract a value from the instance structure. The **`type`** field should contain a type code like `Py_T_INT` or `Py_T_DOUBLE`; the value will be used to determine how to convert Python values to and from C values. The `flags` field is used to store flags which control how the attribute can be accessed: you can set it to `Py_READONLY` to prevent Python code from setting it.

An interesting advantage of using the `tp_members` table to build descriptors that are used at runtime is that any attribute defined this way can have an associated doc string simply by providing the text in the table. An application can use the introspection API to retrieve the descriptor from the class object, and get the doc string using its `__doc__` attribute.

As with the `tp_methods` table, a sentinel entry with a `ml_name` value of `NULL` is required.

3.3.2. Type-specific Attribute Management

For simplicity, only the char* version will be demonstrated here; the type of the name parameter is the only difference between the char* and PyObject* flavors of the interface. This example effectively does the same thing as the generic example above, but does not use the generic support added in Python 2.2. It explains how the handler functions are called, so that if you do need to extend their functionality, you'll understand what needs to be done.

The `tp_getattr` handler is called when the object requires an attribute look-up. It is called in the same situations where the `__getattr__()` method of a class would be called.

Here is an example:

```
static PyObject *
newdatatype_getattr(PyObject *op, char *name)
{
    newdatatypeobject *self = (newdatatypeobject *) op;
    if (strcmp(name, "data") == 0) {
        return PyLong_FromLong(self->data);
    }

    PyErr_Format(PyExc_AttributeError,
                 "'%.100s' object has no attribute '%.400s'",
                 Py_TYPE(self)->tp_name, name);
    return NULL;
}
```

The **`tp_setattr`** handler is called when the `__setattr__()` or `__delattr__()` method of a class instance would be called. When an attribute should be deleted, the third parameter will be `NULL`. Here is an example that simply raises an exception; if this were really all you wanted, the **`tp_setattr`** handler should be set to `NULL`.

```
static int
newdatatype_setattr(PyObject *op, char *name, PyObject *v)
{
    PyErr_Format(PyExc_RuntimeError, "Read-only attribute: %s",
name);
    return -1;
}
```

3.4. Object Comparison

```
richcmpfunc tp_richcompare;
```

The **`tp_richcompare`** handler is called when comparisons are needed. It is analogous to the rich comparison methods, like **`__lt__()`**, and also called by **`PyObject_RichCompare()`** and **`PyObject_RichCompareBool()`**.

This function is called with two Python objects and the operator as arguments, where the operator is one of `Py_EQ`, `Py_NE`, `Py_LE`, `Py_GE`, `Py_LT` or `Py_GT`. It should compare the two objects with respect to the specified operator and return

`Py_True` or `Py_False` if the comparison is successful, `Py_NotImplemented` to indicate that comparison is not implemented and the other object's comparison method should be tried, or `NULL` if an exception was set.

Here is a sample implementation, for a datatype that is considered equal if the size of an internal pointer is equal:

```
static PyObject *
newdatatype_richcmp(PyObject *lhs, PyObject *rhs, int op)
{
    newdatatypeobject *obj1 = (newdatatypeobject *) lhs;
    newdatatypeobject *obj2 = (newdatatypeobject *) rhs;
    PyObject *result;
    int c, size1, size2;

    /* code to make sure that both arguments are of type
       newdatatype omitted */

    size1 = obj1->obj_UnderlyingDatatypePtr->size;
    size2 = obj2->obj_UnderlyingDatatypePtr->size;

    switch (op) {
    case Py_LT: c = size1 <  size2; break;
    case Py_LE: c = size1 <= size2; break;
    case Py_EQ: c = size1 == size2; break;
    case Py_NE: c = size1 != size2; break;
    case Py_GT: c = size1 >  size2; break;
    case Py_GE: c = size1 >= size2; break;
    }
    result = c ? Py_True : Py_False;
    return Py_NewRef(result);
 }
```

3.5. Abstract Protocol Support

Python supports a variety of *abstract* 'protocols;' the specific interfaces provided to use these interfaces are documented in Abstract Objects Layer.

A number of these abstract interfaces were defined early in the development of the Python implementation. In particular, the number, mapping, and sequence protocols have been part of Python since the beginning. Other protocols have been added over time. For protocols which depend on several handler routines from the type implementation, the older protocols have been defined as optional blocks of handlers referenced by the type object. For newer protocols there are additional slots in the main type object, with a flag bit being set to indicate that the slots are present and should be checked by the interpreter. (The flag bit does not indicate that the slot values are non-`NULL`. The flag may be set to indicate the presence of a slot, but a slot may still be unfilled.)

```
PyNumberMethods   *tp_as_number;
PySequenceMethods *tp_as_sequence;
PyMappingMethods  *tp_as_mapping;
```

If you wish your object to be able to act like a number, a sequence, or a mapping object, then you place the address of a structure that implements the C type **`PyNumberMethods`**, **`PySequenceMethods`**, or **`PyMappingMethods`**, respectively. It is up to you to fill in this structure with appropriate values. You can find examples of the use of each of these in the `Objects` directory of the Python source distribution.

```
hashfunc tp_hash;
```

This function, if you choose to provide it, should return a hash number for an instance of your data type. Here is a simple example:

```
static Py_hash_t
newdatatype_hash(PyObject *op)
{
    newdatatypeobject *self = (newdatatypeobject *) op;
    Py_hash_t result;
    result = self->some_size + 32767 * self->some_number;
    if (result == -1) {
        result = -2;
    }
    return result;
}
```

`Py_hash_t` is a signed integer type with a platform-varying width. Returning `-1` from `tp_hash` indicates an error, which is why you should be careful to avoid returning it when hash computation is successful, as seen above.

```
ternaryfunc tp_call;
```

This function is called when an instance of your data type is “called”, for example, if `obj1` is an instance of your data type and the Python script contains `obj1('hello')`, the `tp_call` handler is invoked.

This function takes three arguments:

1. *self* is the instance of the data type which is the subject of the call. If the call is `obj1('hello')`, then *self* is `obj1`.
2. *args* is a tuple containing the arguments to the call. You can use `PyArg_ParseTuple()` to extract the arguments.
3. *kwds* is a dictionary of keyword arguments that were passed. If this is non-`NULL` and you support keyword arguments, use `PyArg_ParseTupleAndKeywords()` to extract the arguments. If you do not want to support keyword arguments and this is non-`NULL`, raise a `TypeError` with a message saying that keyword arguments are not supported.

Here is a toy `tp_call` implementation:

```
static PyObject *
newdatatype_call(PyObject *op, PyObject *args, PyObject *kwds)
{
    newdatatypeobject *self = (newdatatypeobject *) op;
    PyObject *result;
    const char *arg1;
    const char *arg2;
    const char *arg3;

    if (!PyArg_ParseTuple(args, "sss:call", &arg1, &arg2,
&arg3)) {
        return NULL;
    }
    result = PyUnicode_FromFormat(
        "Returning -- value: [%d] arg1: [%s] arg2: [%s] arg3:
[%s]\n",
        self->obj_UnderlyingDatatypePtr->size,
        arg1, arg2, arg3);
    return result;
}
```

```
/* Iterators */
getiterfunc tp_iter;
iternextfunc tp_iternext;
```

These functions provide support for the iterator protocol. Both handlers take exactly one parameter, the instance for which they are being called, and return a new reference. In the case of an error, they should set an exception and return `NULL`. **`tp_iter`** corresponds to the Python **`__iter__()`** method, while **`tp_iternext`** corresponds to the Python **`__next__()`** method.

Any iterable object must implement the **`tp_iter`** handler, which must return an iterator object. Here the same guidelines apply as for Python classes:

- For collections (such as lists and tuples) which can support multiple independent iterators, a new iterator should be created and returned by each call to **`tp_iter`**.
- Objects which can only be iterated over once (usually due to side effects of iteration, such as file objects) can implement **`tp_iter`** by returning a new reference to

themselves – and should also therefore implement the **`tp_iternext`** handler.

Any iterator object should implement both **`tp_iter`** and **`tp_iternext`**. An iterator's **`tp_iter`** handler should return a new reference to the iterator. Its **`tp_iternext`** handler should return a new reference to the next object in the iteration, if there is one. If the iteration has reached the end, **`tp_iternext`** may return `NULL` without setting an exception, or it may set **`StopIteration`** *in addition* to returning `NULL`; avoiding the exception can yield slightly better performance. If an actual error occurs, **`tp_iternext`** should always set an exception and return `NULL`.

3.6. Weak Reference Support

One of the goals of Python's weak reference implementation is to allow any type to participate in the weak reference mechanism without incurring the overhead on performance-critical objects (such as numbers).

See also

Documentation for the **`weakref`** module.

For an object to be weakly referenceable, the extension type must set the `Py_TPFLAGS_MANAGED_WEAKREF` bit of the **`tp_flags`** field. The legacy **`tp_weaklistoffset`** field should be left as zero.

Concretely, here is how the statically declared type object would look:

```
static PyTypeObject TrivialType = {
    PyVarObject_HEAD_INIT(NULL, 0)
    /* ... other members omitted for brevity ... */
    .tp_flags = Py_TPFLAGS_MANAGED_WEAKREF | ...,
};
```

The only further addition is that `tp_dealloc` needs to clear any weak references (by calling **`PyObject_ClearWeakRefs()`**):

```
static void
Trivial_dealloc(PyObject *op)
{
    /* Clear weakrefs first before calling any destructors */
    PyObject_ClearWeakRefs(op);
    /* ... remainder of destruction code omitted for brevity
... */
    Py_TYPE(op)->tp_free(op);
}
```

3.7. More Suggestions

In order to learn how to implement any specific method for your new data type, get the CPython source code. Go to the `Objects` directory, then search the C source files for `tp_` plus the function you want (for example, `tp_richcompare`). You will find examples of the function you want to implement.

When you need to verify that an object is a concrete instance of the type you are implementing, use the **`PyObject_TypeCheck()`** function. A sample of its use might be something like the following:

```
if (!PyObject_TypeCheck(some_object, &MyType)) {
    PyErr_SetString(PyExc_TypeError, "arg #1 not a mything");
    return NULL;
}
```

See also

Download CPython source releases.
https://www.python.org/downloads/source/

The CPython project on GitHub, where the CPython source code is developed.

https://github.com/python/cpython

4. Building C and C++ Extensions

A C extension for CPython is a shared library (for example, a `.so` file on Linux, `.pyd` on Windows), which exports an *initialization function*.

See Defining extension modules for details.

4.1. Building C and C++ Extensions with setuptools

Building, packaging and distributing extension modules is best done with third-party tools, and is out of scope of this document. One suitable tool is Setuptools, whose documentation can be found at https://setuptools.pypa.io/en/latest/setuptools.html.

The **`distutils`** module, which was included in the standard library until Python 3.12, is now maintained as part of Setuptools.

5. Building C and C++ Extensions on Windows

This chapter briefly explains how to create a Windows extension module for Python using Microsoft Visual C++, and follows with more detailed background information on how it works. The explanatory material is useful for both the Windows programmer learning to build Python extensions and the Unix programmer interested in producing software which can be successfully built on both Unix and Windows.

Module authors are encouraged to use the distutils approach for building extension modules, instead of the one described in this section. You will still need the C compiler that was used to build Python; typically Microsoft Visual C++.

Note

This chapter mentions a number of filenames that include an encoded Python version number. These filenames are represented with the version number shown as XY; in practice, `'X'` will be the major version number and `'Y'` will be the minor version number of the Python release you’re working with. For example, if you are using Python 2.2.1, XY will actually be 22.

5.1. A Cookbook Approach

There are two approaches to building extension modules on Windows, just as there are on Unix: use the `setuptools` package to control the build process, or do things manually. The setuptools approach works well for most extensions; documentation on using `setuptools` to build and package extension modules is available in Building C and C++ Extensions with setuptools. If you find you really need to do things manually, it may be instructive to study the project file for the winsound [https://github.com/python/cpython/tree/3.14/PCbuild/winsound.vcxproj] standard library module.

5.2. Differences Between Unix and Windows

Unix and Windows use completely different paradigms for run-time loading of code. Before you try to build a module that can be dynamically loaded, be aware of how your system works.

In Unix, a shared object (`.so`) file contains code to be used by the program, and also the names of functions and data that it expects to find in the program. When the file is joined to the program, all references to those functions and data in the file's code are changed to point to the actual locations in the program where the functions and data are placed in memory. This is basically a link operation.

In Windows, a dynamic-link library (`.dll`) file has no dangling references. Instead, an access to functions or data goes through a lookup table. So the DLL code does not have to be fixed up at runtime to refer to the program's memory;

instead, the code already uses the DLL's lookup table, and the lookup table is modified at runtime to point to the functions and data.

In Unix, there is only one type of library file (`.a`) which contains code from several object files (`.o`). During the link step to create a shared object file (`.so`), the linker may find that it doesn't know where an identifier is defined. The linker will look for it in the object files in the libraries; if it finds it, it will include all the code from that object file.

In Windows, there are two types of library, a static library and an import library (both called `.lib`). A static library is like a Unix `.a` file; it contains code to be included as necessary. An import library is basically used only to reassure the linker that a certain identifier is legal, and will be present in the program when the DLL is loaded. So the linker uses the information from the import library to build the lookup table for using identifiers that are not included in the DLL. When an application or a DLL is linked, an import library may be generated, which will need to be used for all future DLLs that depend on the symbols in the application or DLL.

Suppose you are building two dynamic-load modules, B and C, which should share another block of code A. On Unix, you would *not* pass `A.a` to the linker for `B.so` and `C.so`; that would cause it to be included twice, so that B and C would each have their own copy. In Windows, building `A.dll` will also build `A.lib`. You *do* pass `A.lib` to the linker for B and C. `A.lib` does not contain code; it just contains information which will be used at runtime to access A's code.

In Windows, using an import library is sort of like using `import spam`; it gives you access to spam's names, but does not create a separate copy. On Unix, linking with a library is more like `from spam import *`; it does create a separate copy.

Py_NO_LINK_LIB

Turn off the implicit, `#pragma`-based linkage with the Python library, performed inside CPython header files.

Added in version 3.14.

5.3. Using DLLs in Practice

Windows Python is built in Microsoft Visual C++; using other compilers may or may not work. The rest of this section is MSVC++ specific.

When creating DLLs in Windows, you can use the CPython library in two ways:

1. By default, inclusion of `PC/pyconfig.h` directly or via `Python.h` triggers an implicit, configure-aware link with the library. The header file chooses `pythonXY_d.lib` for Debug, `pythonXY.lib` for Release, and `pythonX.lib` for Release with the Limited API enabled.

   To build two DLLs, spam and ni (which uses C functions found in spam), you could use these commands:

   ```
   cl /LD /I/python/include spam.c
   cl /LD /I/python/include ni.c spam.lib
   ```

   The first command created three files: `spam.obj`, `spam.dll` and `spam.lib`. `Spam.dll` does not contain any Python functions (such as **`PyArg_ParseTuple()`**), but it does know how to find the Python code thanks to the implicitly linked `pythonXY.lib`.

The second command created `ni.dll` (and `.obj` and `.lib`), which knows how to find the necessary functions from spam, and also from the Python executable.

2. Manually by defining `Py_NO_LINK_LIB` macro before including `Python.h`. You must pass `pythonXY.lib` to the linker.

   To build two DLLs, spam and ni (which uses C functions found in spam), you could use these commands:

   ```
   cl /LD /DPy_NO_LINK_LIB /I/python/include spam.c
   ../libs/pythonXY.lib
   cl /LD /DPy_NO_LINK_LIB /I/python/include ni.c spam.lib
   ../libs/pythonXY.lib
   ```

   The first command created three files: `spam.obj`, `spam.dll` and `spam.lib`. `Spam.dll` does not contain any Python functions (such as `PyArg_ParseTuple()`), but it does know how to find the Python code thanks to `pythonXY.lib`.

   The second command created `ni.dll` (and `.obj` and `.lib`), which knows how to find the necessary functions from spam, and also from the Python executable.

Not every identifier is exported to the lookup table. If you want any other modules (including Python) to be able to see your identifiers, you have to say `_declspec(dllexport)`, as in `void _declspec(dllexport) initspam(void)` or `PyObject _declspec(dllexport) *NiGetSpamData(void)`.

Developer Studio will throw in a lot of import libraries that you do not really need, adding about 100K to your executable. To get rid of them, use the Project Settings dialog, Link tab, to specify *ignore default libraries*. Add the correct `msvcrtxx.lib` to the list of libraries.

1. Embedding Python in Another Application

The previous chapters discussed how to extend Python, that is, how to extend the functionality of Python by attaching a library of C functions to it. It is also possible to do it the other way around: enrich your C/C++ application by embedding Python in it. Embedding provides your application with the ability to implement some of the functionality of your application in Python rather than C or C++. This can be used for many purposes; one example would be to allow users to tailor the application to their needs by writing some scripts in Python. You can also use it yourself if some of the functionality can be written in Python more easily.

Embedding Python is similar to extending it, but not quite. The difference is that when you extend Python, the main program of the application is still the Python interpreter, while if you embed Python, the main program may have nothing to do with Python — instead, some parts of the application occasionally call the Python interpreter to run some Python code.

So if you are embedding Python, you are providing your own main program. One of the things this main program has to do is initialize the Python interpreter. At the very least, you have to call the function `Py_Initialize()`. There are optional calls to pass command line arguments to Python. Then later you can call the interpreter from any part of the application.

There are several different ways to call the interpreter: you can pass a string containing Python statements to

`PyRun_SimpleString()`, or you can pass a stdio file pointer and a file name (for identification in error messages only) to `PyRun_SimpleFile()`. You can also call the lower-level operations described in the previous chapters to construct and use Python objects.

See also

Python/C API Reference Manual
The details of Python's C interface are given in this manual. A great deal of necessary information can be found here.

1.1. Very High Level Embedding

The simplest form of embedding Python is the use of the very high level interface. This interface is intended to execute a Python script without needing to interact with the application directly. This can for example be used to perform some operation on a file.

```
#define PY_SSIZE_T_CLEAN
#include <Python.h>

int
main(int argc, char *argv[])
{
    PyStatus status;
    PyConfig config;
    PyConfig_InitPythonConfig(&config);

    /* optional but recommended */
    status = PyConfig_SetBytesString(&config,
&config.program_name, argv[0]);
    if (PyStatus_Exception(status)) {
        goto exception;
    }
    status = Py_InitializeFromConfig(&config);
    if (PyStatus_Exception(status)) {
        goto exception;
    }
    PyConfig_Clear(&config);

    PyRun_SimpleString("from time import time,ctime\n"
                       "print('Today is', ctime(time()))\n");
    if (Py_FinalizeEx() < 0) {
        exit(120);
    }
    return 0;

  exception:
     PyConfig_Clear(&config);
     Py_ExitStatusException(status);
}
```

Note

`#define PY_SSIZE_T_CLEAN` was used to indicate that `Py_ssize_t` should be used in some APIs instead of `int`. It is not necessary since Python 3.13, but we keep it here for backward compatibility. See Strings and buffers for a description of this macro.

Setting `PyConfig.program_name` should be called before `Py_InitializeFromConfig()` to inform the interpreter about paths to Python run-time libraries. Next, the Python interpreter is initialized with `Py_Initialize()`, followed by the execution of a hard-coded Python script that prints the date and time. Afterwards, the `Py_FinalizeEx()` call shuts the interpreter down, followed by the end of the program. In a real program, you may want to get the Python script from another source, perhaps a text-editor routine, a file, or a database. Getting the Python code from a file can better be done by using the `PyRun_SimpleFile()` function, which saves you the trouble of allocating memory space and loading the file contents.

1.2. Beyond Very High Level Embedding: An overview

The high level interface gives you the ability to execute arbitrary pieces of Python code from your application, but exchanging data values is quite cumbersome to say the least. If you want that, you should use lower level calls. At the cost of having to write more C code, you can achieve almost anything.

It should be noted that extending Python and embedding Python is quite the same activity, despite the different intent. Most topics discussed in the previous chapters are still valid. To show this, consider what the extension code from Python to C really does:

1. Convert data values from Python to C,
2. Perform a function call to a C routine using the converted values, and
3. Convert the data values from the call from C to Python.

When embedding Python, the interface code does:

1. Convert data values from C to Python,
2. Perform a function call to a Python interface routine using the converted values, and
3. Convert the data values from the call from Python to C.

As you can see, the data conversion steps are simply swapped to accommodate the different direction of the cross-language transfer. The only difference is the routine that you call between both data conversions. When extending, you call a C routine, when embedding, you call a Python routine.

This chapter will not discuss how to convert data from Python to C and vice versa. Also, proper use of references and dealing with errors is assumed to be understood. Since these aspects do not differ from extending the interpreter, you can refer to earlier chapters for the required information.

1.3. Pure Embedding

The first program aims to execute a function in a Python script. Like in the section about the very high level interface, the Python interpreter does not directly interact with the application (but that will change in the next section).

The code to run a function defined in a Python script is:

```
#define PY_SSIZE_T_CLEAN
#include <Python.h>

int
main(int argc, char *argv[])
{
    PyObject *pName, *pModule, *pFunc;
    PyObject *pArgs, *pValue;
    int i;

    if (argc < 3) {
        fprintf(stderr,"Usage: call pythonfile funcname
[args]\n");
        return 1;
    }

    Py_Initialize();
    pName = PyUnicode_DecodeFSDefault(argv[1]);
    /* Error checking of pName left out */
    pModule = PyImport_Import(pName);
    Py_DECREF(pName);

    if (pModule != NULL) {
        pFunc = PyObject_GetAttrString(pModule, argv[2]);
        /* pFunc is a new reference */

        if (pFunc && PyCallable_Check(pFunc)) {
            pArgs = PyTuple_New(argc - 3);
            for (i = 0; i < argc - 3; ++i) {
                pValue = PyLong_FromLong(atoi(argv[i + 3]));
                if (!pValue) {
                    Py_DECREF(pArgs);
                    Py_DECREF(pModule);
                    fprintf(stderr, "Cannot convert
argument\n");
                    return 1;
                }
                /* pValue reference stolen here: */
                PyTuple_SetItem(pArgs, i, pValue);
            }
            pValue = PyObject_CallObject(pFunc, pArgs);
            Py_DECREF(pArgs);
            if (pValue != NULL) {
                printf("Result of call: %ld\n",
PyLong_AsLong(pValue));
                Py_DECREF(pValue);
            }
            else {
                Py_DECREF(pFunc);
                Py_DECREF(pModule);
                PyErr_Print();
                fprintf(stderr,"Call failed\n");
                return 1;
            }
        }
        else {
            if (PyErr_Occurred())
                PyErr_Print();
            fprintf(stderr, "Cannot find function \"%s\"\n",
argv[2]);
        }
        Py_XDECREF(pFunc);
        Py_DECREF(pModule);
    }
    else {
        PyErr_Print();
        fprintf(stderr, "Failed to load \"%s\"\n", argv[1]);
        return 1;
    }
    if (Py_FinalizeEx() < 0) {
        return 120;
    }
    return 0;
}
```

This code loads a Python script using `argv[1]`, and calls the function named in `argv[2]`. Its integer arguments are the other values of the `argv` array. If you compile and link this program (let’s call the finished executable **call**), and use it to execute a Python script, such as:

```
def multiply(a,b):
    print("Will compute", a, "times", b)
    c = 0
    for i in range(0, a):
        c = c + b
    return c
```

then the result should be:

```
$ call multiply multiply 3 2
Will compute 3 times 2
Result of call: 6
```

Although the program is quite large for its functionality, most of the code is for data conversion between Python and C, and for error reporting. The interesting part with respect to embedding Python starts with

```
Py_Initialize();
pName = PyUnicode_DecodeFSDefault(argv[1]);
/* Error checking of pName left out */
pModule = PyImport_Import(pName);
```

After initializing the interpreter, the script is loaded using `PyImport_Import()`. This routine needs a Python string as its argument, which is constructed using the `PyUnicode_DecodeFSDefault()` data conversion routine.

```
pFunc = PyObject_GetAttrString(pModule, argv[2]);
/* pFunc is a new reference */

if (pFunc && PyCallable_Check(pFunc)) {
    ...
}
Py_XDECREF(pFunc);
```

Once the script is loaded, the name we’re looking for is retrieved using `PyObject_GetAttrString()`. If the name exists, and the object returned is callable, you can safely assume that it is a function. The program then proceeds by constructing a tuple of arguments as normal. The call to the Python function is then made with:

```
pValue = PyObject_CallObject(pFunc, pArgs);
```

Upon return of the function, `pValue` is either `NULL` or it contains a reference to the return value of the function. Be sure to release the reference after examining the value.

1.4. Extending Embedded Python

Until now, the embedded Python interpreter had no access to functionality from the application itself. The Python API allows this by extending the embedded interpreter. That is, the embedded interpreter gets extended with routines provided by the application. While it sounds complex, it is not so bad. Simply forget for a while that the application starts the Python interpreter. Instead, consider the application to be a set of subroutines, and write some glue code that gives Python access to those routines, just like you would write a normal Python extension. For example:

```
static int numargs=0;

/* Return the number of arguments of the application command
line */
static PyObject*
emb_numargs(PyObject *self, PyObject *args)
{
    if(!PyArg_ParseTuple(args, ":numargs"))
        return NULL;
    return PyLong_FromLong(numargs);
}

static PyMethodDef emb_module_methods[] = {
    {"numargs", emb_numargs, METH_VARARGS,
     "Return the number of arguments received by the
process."},
    {NULL, NULL, 0, NULL}
};

static struct PyModuleDef emb_module = {
    .m_base = PyModuleDef_HEAD_INIT,
    .m_name = "emb",
    .m_size = 0,
    .m_methods = emb_module_methods,
};

static PyObject*
PyInit_emb(void)
{
    return PyModuleDef_Init(&emb_module);
}
```

Insert the above code just above the **`main()`** function. Also, insert the following two statements before the call to **`Py_Initialize()`**:

```
numargs = argc;
PyImport_AppendInittab("emb", &PyInit_emb);
```

These two lines initialize the `numargs` variable, and make the **`emb.numargs()`** function accessible to the embedded Python interpreter. With these extensions, the Python script can do things like

```
import emb
print("Number of arguments", emb.numargs())
```

In a real application, the methods will expose an API of the application to Python.

1.5. Embedding Python in C++

It is also possible to embed Python in a C++ program; precisely how this is done will depend on the details of the C++ system used; in general you will need to write the main program in C++, and use the C++ compiler to compile and link your program. There is no need to recompile Python itself using C++.

1.6. Compiling and Linking under Unix-like systems

It is not necessarily trivial to find the right flags to pass to your compiler (and linker) in order to embed the Python interpreter into your application, particularly because Python needs to load library modules implemented as C dynamic extensions (`.so` files) linked against it.

To find out the required compiler and linker flags, you can execute the `python`*`X.Y`*`-config` script which is generated as part of the installation process (a `python3-config` script may also be available). This script has several options, of which the following will be directly useful to you:

- `pythonX.Y-config --cflags` will give you the recommended flags when compiling:

  ```
  $ /opt/bin/python3.11-config --cflags
  -I/opt/include/python3.11 -I/opt/include/python3.11 -Wsign-compare  -DNDEBUG -g -fwrapv -O3 -Wall
  ```

- `pythonX.Y-config --ldflags --embed` will give you the recommended flags when linking:

  ```
  $ /opt/bin/python3.11-config --ldflags --embed
  -L/opt/lib/python3.11/config-3.11-x86_64-linux-gnu -L/opt/lib -lpython3.11 -lpthread -ldl  -lutil -lm
  ```

Note

To avoid confusion between several Python installations (and especially between the system Python and your own compiled Python), it is recommended that you use the absolute path to `python`*`X.Y`*`-config`, as in the above example.

If this procedure doesn't work for you (it is not guaranteed to work for all Unix-like platforms; however, we welcome bug reports) you will have to read your system's documentation about dynamic linking and/or examine Python's `Makefile` (use **`sysconfig.get_makefile_filename()`** to find its location) and compilation options. In this case, the **`sysconfig`** module is a useful tool to programmatically extract the configuration values that you will want to combine together. For example:

```
>>> import sysconfig
>>> sysconfig.get_config_var('LIBS')
'-lpthread -ldl  -lutil'
>>> sysconfig.get_config_var('LINKFORSHARED')
'-Xlinker -export-dynamic'
```

Python/C API Reference Manual

This manual documents the API used by C and C++ programmers who want to write extension modules or embed Python. It is a companion to Extending and Embedding the Python Interpreter, which describes the general principles of extension writing but does not document the API functions in detail.

- Introduction
    - Language version compatibility
    - Coding standards
    - Include Files
    - Useful macros
    - Objects, Types and Reference Counts
    - Exceptions
    - Embedding Python
    - Debugging Builds
    - Recommended third party tools
- C API Stability
    - Unstable C API
    - Stable Application Binary Interface
    - Platform Considerations
    - Contents of Limited API
- The Very High Level Layer
    - Available start symbols
    - Stack Effects
- Reference Counting
- Exception Handling
    - Printing and clearing
    - Raising exceptions

  - Issuing warnings
  - Querying the error indicator
  - Signal Handling
  - Exception Classes
  - Exception Objects
  - Unicode Exception Objects
  - Recursion Control
  - Exception and warning types
  - Tracebacks
- Defining extension modules
  - Multiple module instances
  - Initialization function
  - Multi-phase initialization
  - Legacy single-phase initialization
- Utilities
  - Operating System Utilities
  - System Functions
  - Process Control
  - Importing Modules
  - Data marshalling support
  - Parsing arguments and building values
  - String conversion and formatting
  - Character classification and conversion
  - PyHash API
  - Reflection
  - Codec registry and support functions
  - PyTime C API
  - Support for Perf Maps
- Abstract Objects Layer
  - Object Protocol
  - Call Protocol
  - Number Protocol
  - Sequence Protocol
  - Mapping Protocol
  - Iterator Protocol
  - Buffer Protocol

- Concrete Objects Layer
  - Fundamental Objects
  - Numeric Objects
  - Sequence Objects
  - Container Objects
  - Function Objects
  - Other Objects
  - C API for extension modules
- Initialization, Finalization, and Threads
  - Before Python Initialization
  - Global configuration variables
  - Initializing and finalizing the interpreter
  - Process-wide parameters
  - Thread State and the Global Interpreter Lock
  - Sub-interpreter support
  - Asynchronous Notifications
  - Profiling and Tracing
  - Reference tracing
  - Advanced Debugger Support
  - Thread Local Storage Support
  - Synchronization Primitives
  - Operating System Thread APIs
- Python Initialization Configuration
  - PyInitConfig C API
  - Configuration Options
  - Runtime Python configuration API
  - PyConfig C API
  - Py_GetArgcArgv()
  - Delaying main module execution
- Memory Management
  - Overview
  - Allocator Domains
  - Raw Memory Interface
  - Memory Interface
  - Object allocators
  - Default Memory Allocators

  - Customize Memory Allocators
  - Debug hooks on the Python memory allocators
  - The pymalloc allocator
  - The mimalloc allocator
  - tracemalloc C API
  - Examples
- Object Implementation Support
  - Allocating Objects on the Heap
  - Object Life Cycle
  - Common Object Structures
  - Type Object Structures
  - Supporting Cyclic Garbage Collection
- API and ABI Versioning
  - Build-time version constants
  - Run-time version
  - Bit-packing macros
- Monitoring C API
- Generating Execution Events
  - Managing the Monitoring State

Introduction

The Application Programmer's Interface to Python gives C and C++ programmers access to the Python interpreter at a variety of levels. The API is equally usable from C++, but for brevity it is generally referred to as the Python/C API. There are two fundamentally different reasons for using the Python/C API. The first reason is to write *extension modules* for specific purposes; these are C modules that extend the Python interpreter. This is probably the most common use. The second reason is to use Python as a component in a larger application; this technique is generally referred to as *embedding* Python in an application.

Writing an extension module is a relatively well-understood process, where a "cookbook" approach works well. There are several tools that automate the process to some extent. While people have embedded Python in other applications since its early existence, the process of embedding Python is less straightforward than writing an extension.

Many API functions are useful independent of whether you're embedding or extending Python; moreover, most applications that embed Python will need to provide a custom extension as well, so it's probably a good idea to become familiar with writing an extension before attempting to embed Python in a real application.

Language version compatibility

Python's C API is compatible with C11 and C++11 versions of C and C++.

This is a lower limit: the C API does not require features from later C/C++ versions. You do *not* need to enable your compiler's "c11 mode".

Coding standards

If you're writing C code for inclusion in CPython, you **must** follow the guidelines and standards defined in **PEP 7** [https://peps.python.org/pep-0007/]. These guidelines apply regardless of the version of Python you are contributing to. Following these conventions is not necessary for your own third party extension modules, unless you eventually expect to contribute them to Python.

Include Files

All function, type and macro definitions needed to use the Python/C API are included in your code by the following line:

```
#define PY_SSIZE_T_CLEAN
#include <Python.h>
```

This implies inclusion of the following standard headers: `<stdio.h>`, `<string.h>`, `<errno.h>`, `<limits.h>`, `<assert.h>` and `<stdlib.h>` (if available).

Note

Since Python may define some pre-processor definitions which affect the standard headers on some systems, you *must* include `Python.h` before any standard headers are included.

It is recommended to always define `PY_SSIZE_T_CLEAN` before including `Python.h`. See Parsing arguments and building values for a description of this macro.

All user visible names defined by Python.h (except those defined by the included standard headers) have one of the prefixes `Py` or `_Py`. Names beginning with `_Py` are for internal use by the Python implementation and should not be used by extension writers. Structure member names do not have a reserved prefix.

Note

User code should never define names that begin with `Py` or `_Py`. This confuses the reader, and jeopardizes the portability of the user code to future Python versions, which may define additional names beginning with one of these prefixes.

The header files are typically installed with Python. On Unix, these are located in the directories `prefix/include/pythonversion/` and `exec_prefix/include/pythonversion/`, where **`prefix`** and **`exec_prefix`** are defined by the corresponding parameters to Python's **configure** script and *version* is `'%d.%d' % sys.version_info[:2]`. On Windows, the headers are installed in `prefix/include`, where `prefix` is the installation directory specified to the installer.

To include the headers, place both directories (if different) on your compiler's search path for includes. Do *not* place the parent directories on the search path and then use `#include <pythonX.Y/Python.h>`; this will break on multi-platform builds since the platform independent headers under `prefix` include the platform specific headers from `exec_prefix`.

C++ users should note that although the API is defined entirely using C, the header files properly declare the entry points to be `extern "C"`. As a result, there is no need to do anything special to use the API from C++.

Useful macros

Several useful macros are defined in the Python header files. Many are defined closer to where they are useful (for example, `Py_RETURN_NONE`, `PyMODINIT_FUNC`). Others of a more general utility are defined here. This is not necessarily a complete listing.

Py_ABS(x)

Return the absolute value of `x`.

If the result cannot be represented (for example, if `x` has **`INT_MIN`** value for int type), the behavior is undefined.

Added in version 3.3.

Py_ALWAYS_INLINE

Ask the compiler to always inline a static inline function. The compiler can ignore it and decide to not inline the function.

It can be used to inline performance critical static inline functions when building Python in debug mode with function inlining disabled. For example, MSC disables function inlining when building in debug mode.

Marking blindly a static inline function with Py_ALWAYS_INLINE can result in worse performances (due to increased code size for example). The compiler is usually smarter than the developer for the cost/benefit analysis.

If Python is built in debug mode (if the **`Py_DEBUG`** macro is defined), the **`Py_ALWAYS_INLINE`** macro does nothing.

It must be specified before the function return type. Usage:

```
static inline Py_ALWAYS_INLINE int random(void) { return 4;
}
```

Added in version 3.11.

Py_CHARMASK(c)

Argument must be a character or an integer in the range [-128, 127] or [0, 255]. This macro returns `c` cast to an `unsigned char`.

Py_DEPRECATED(version)

Use this for deprecated declarations. The macro must be placed before the symbol name.

Example:

```
Py_DEPRECATED(3.8) PyAPI_FUNC(int) Py_OldFunction(void);
```

Changed in version 3.8: MSVC support was added.

Py_GETENV(s)

Like `getenv(s)`, but returns `NULL` if **`-E`** was passed on the command line (see **`PyConfig.use_environment`**).

Py_LOCAL(type)

Declare a function returning the specified *type* using a fast-calling qualifier for functions that are local to the current file. Semantically, this is equivalent to `static type`.

Py_LOCAL_INLINE(type)

Equivalent to **`Py_LOCAL`** but additionally requests the function be inlined.

Py_LOCAL_SYMBOL

Macro used to declare a symbol as local to the shared library (hidden). On supported platforms, it ensures the symbol is not exported.

On compatible versions of GCC/Clang, it expands to `__attribute__((visibility("hidden")))`.

Py_MAX(x, y)

Return the maximum value between `x` and `y`.

Added in version 3.3.

Py_MEMBER_SIZE(type, member)

Return the size of a structure (`type`) `member` in bytes.

Added in version 3.6.

Py_MEMCPY(dest, src, n)

This is a soft deprecated alias to **`memcpy()`**. Use **`memcpy()`** directly instead.

Deprecated since version 3.14: The macro is soft deprecated.

Py_MIN(x, y)

Return the minimum value between `x` and `y`.

Added in version 3.3.

Py_NO_INLINE

Disable inlining on a function. For example, it reduces the C stack consumption: useful on LTO+PGO builds which heavily inline code (see bpo-33720 [https://bugs.python.org/issue?@action=redirect&bpo=33720]).

Usage:

```
Py_NO_INLINE static int random(void) { return 4; }
```

Added in version 3.11.

Py_STRINGIFY(x)

Convert `x` to a C string. E.g. `Py_STRINGIFY(123)` returns `"123"`.

Added in version 3.4.

Py_UNREACHABLE()

Use this when you have a code path that cannot be reached by design. For example, in the `default:` clause in a `switch` statement for which all possible values are covered in `case` statements. Use this in places where you might be tempted to put an `assert(0)` or `abort()` call.

In release mode, the macro helps the compiler to optimize the code, and avoids a warning about unreachable code. For example, the macro is implemented with `__builtin_unreachable()` on GCC in release mode.

A use for `Py_UNREACHABLE()` is following a call a function that never returns but that is not declared **`_Py_NO_RETURN`**.

If a code path is very unlikely code but can be reached under exceptional case, this macro must not be used. For example, under low memory condition or if a system call returns a value out of the expected range. In this case, it's better to report the error to the caller. If the error cannot be reported to caller, `Py_FatalError()` can be used.

Added in version 3.7.

Py_UNUSED(arg)

Use this for unused arguments in a function definition to silence compiler warnings. Example: `int func(int a, int Py_UNUSED(b)) { return a; }`.

Added in version 3.4.

Py_BUILD_ASSERT(cond)

Asserts a compile-time condition *cond*, as a statement. The build will fail if the condition is false or cannot be evaluated at compile time.

For example:

```
Py_BUILD_ASSERT(sizeof(PyTime_t) == sizeof(int64_t));
```

Added in version 3.3.

Py_BUILD_ASSERT_EXPR(cond)

Asserts a compile-time condition *cond*, as an expression that evaluates to `0`. The build will fail if the condition is false or cannot be evaluated at compile time.

For example:

```
#define foo_to_char(foo) \
    ((char *)(foo) + Py_BUILD_ASSERT_EXPR(offsetof(struct
foo, string) == 0))
```

Added in version 3.3.

PyDoc_STRVAR(name, str)

Creates a variable with name *name* that can be used in docstrings. If Python is built without docstrings, the value will be empty.

Use **`PyDoc_STRVAR`** for docstrings to support building Python without docstrings, as specified in **PEP 7** [https://peps.python.org/pep-0007/].

Example:

```
PyDoc_STRVAR(pop_doc, "Remove and return the rightmost
element.");

static PyMethodDef deque_methods[] = {
    // ...
    {"pop", (PyCFunction)deque_pop, METH_NOARGS, pop_doc},
    // ...
}
```

PyDoc_STR(str)

Creates a docstring for the given input string or an empty string if docstrings are disabled.

Use **`PyDoc_STR`** in specifying docstrings to support building Python without docstrings, as specified in **PEP 7** [https://peps.python.org/pep-0007/].

Example:

```
static PyMethodDef pysqlite_row_methods[] = {
    {"keys", (PyCFunction)pysqlite_row_keys, METH_NOARGS,
        PyDoc_STR("Returns the keys of the row.")},
    {NULL, NULL}
};
```

PyDoc_VAR(name)

Declares a static character array variable with the given name *name*.

For example:

```
PyDoc_VAR(python_doc) = PyDoc_STR("A genus of constricting
snakes in the Pythonidae family native "
                                  "to the tropics and
subtropics of the Eastern Hemisphere.");
```

Py_ARRAY_LENGTH(array)

Compute the length of a statically allocated C array at compile time.

The *array* argument must be a C array with a size known at compile time. Passing an array with an unknown size, such as a heap-allocated array, will result in a compilation error on some compilers, or otherwise produce incorrect results.

This is roughly equivalent to:

```
sizeof(array) / sizeof((array)[0])
```

Py_EXPORTED_SYMBOL

Macro used to declare a symbol (function or data) as exported. On Windows, this expands to `__declspec(dllexport)`. On compatible versions of GCC/Clang, it expands to `__attribute__((visibility("default")))`. This macro is for defining the C API itself; extension modules should not use it.

Py_IMPORTED_SYMBOL

Macro used to declare a symbol as imported. On Windows, this expands to `__declspec(dllimport)`. This macro is for defining the C API itself; extension modules should not use it.

PyAPI_FUNC(type)

Macro used by CPython to declare a function as part of the C API. Its expansion depends on the platform and build configuration. This macro is intended for defining CPython's C API itself; extension modules should not use it for their own symbols.

PyAPI_DATA(type)

Macro used by CPython to declare a public global variable as part of the C API. Its expansion depends on the platform and build configuration. This macro is intended for defining CPython's C API itself; extension modules should not use it for their own symbols.

Objects, Types and Reference Counts

Most Python/C API functions have one or more arguments as well as a return value of type PyObject*. This type is a pointer to an opaque data type representing an arbitrary Python object. Since all Python object types are treated the same way by the Python language in most situations (e.g., assignments, scope rules, and argument passing), it is only fitting that they should be represented by a single C type. Almost all Python objects live on the heap: you never declare an automatic or static variable of type **`PyObject`**, only pointer variables of type PyObject* can be declared. The sole exception are the type objects; since these must never be deallocated, they are typically static **`PyTypeObject`** objects.

All Python objects (even Python integers) have a *type* and a *reference count*. An object's type determines what kind of object it is (e.g., an integer, a list, or a user-defined function; there are many more as explained in The standard type hierarchy). For each of the well-known types there is a macro to check whether an object is of that type; for instance, `PyList_Check(a)` is true if (and only if) the object pointed to by *a* is a Python list.

Reference Counts

The reference count is important because today's computers have a finite (and often severely limited) memory size; it counts how many different places there are that have a strong reference to an object. Such a place could be another object, or a global (or static) C variable, or a local variable in some C function. When the last strong reference to an object is released (i.e. its reference count becomes zero), the object is deallocated. If it contains references to other objects, those references are released. Those other objects may be deallocated in turn, if there are no more references to them, and so on. (There's an obvious problem with objects that reference each other here; for now, the solution is "don't do that.")

Reference counts are always manipulated explicitly. The normal way is to use the macro **`Py_INCREF()`** to take a new reference to an object (i.e. increment its reference count by one), and **`Py_DECREF()`** to release that reference (i.e. decrement the reference count by one). The **`Py_DECREF()`** macro is considerably more complex than the incref one, since it must check whether the reference count becomes zero and then cause the object's deallocator to be called. The deallocator is a function pointer contained in the object's type structure. The type-specific deallocator takes care of releasing references for other objects contained in the object if this is a compound object type, such as a list, as well as performing any additional finalization that's needed. There's no chance that the reference count can overflow; at least as many bits are used to hold the reference count as there are distinct memory locations in virtual memory (assuming `sizeof(Py_ssize_t) >=`

`sizeof(void*)`). Thus, the reference count increment is a simple operation.

It is not necessary to hold a strong reference (i.e. increment the reference count) for every local variable that contains a pointer to an object. In theory, the object’s reference count goes up by one when the variable is made to point to it and it goes down by one when the variable goes out of scope. However, these two cancel each other out, so at the end the reference count hasn’t changed. The only real reason to use the reference count is to prevent the object from being deallocated as long as our variable is pointing to it. If we know that there is at least one other reference to the object that lives at least as long as our variable, there is no need to take a new strong reference (i.e. increment the reference count) temporarily. An important situation where this arises is in objects that are passed as arguments to C functions in an extension module that are called from Python; the call mechanism guarantees to hold a reference to every argument for the duration of the call.

However, a common pitfall is to extract an object from a list and hold on to it for a while without taking a new reference. Some other operation might conceivably remove the object from the list, releasing that reference, and possibly deallocating it. The real danger is that innocent-looking operations may invoke arbitrary Python code which could do this; there is a code path which allows control to flow back to the user from a **`Py_DECREF()`**, so almost any operation is potentially dangerous.

A safe approach is to always use the generic operations (functions whose name begins with `PyObject_`, `PyNumber_`, `PySequence_` or `PyMapping_`). These operations always create a new strong reference (i.e. increment the reference count) of the object they return. This leaves the caller with the responsibility to call `Py_DECREF()` when they are done with the result; this soon becomes second nature.

Reference Count Details

The reference count behavior of functions in the Python/C API is best explained in terms of *ownership of references*. Ownership pertains to references, never to objects (objects are not owned: they are always shared). “Owning a reference” means being responsible for calling Py_DECREF on it when the reference is no longer needed. Ownership can also be transferred, meaning that the code that receives ownership of the reference then becomes responsible for eventually releasing it by calling `Py_DECREF()` or `Py_XDECREF()` when it’s no longer needed—or passing on this responsibility (usually to its caller). When a function passes ownership of a reference on to its caller, the caller is said to receive a *new* reference. When no ownership is transferred, the caller is said to *borrow* the reference. Nothing needs to be done for a borrowed reference.

Conversely, when a calling function passes in a reference to an object, there are two possibilities: the function *steals* a reference to the object, or it does not. *Stealing a reference* means that when you pass a reference to a function, that function assumes that it now owns that reference, and you are not responsible for it any longer.

Few functions steal references; the two notable exceptions are `PyList_SetItem()` and `PyTuple_SetItem()`, which steal a reference to the item (but not to the tuple or list into which the item is put!). These functions were designed to steal a reference because of a common idiom for populating a tuple or list with newly created objects; for example, the code to create the tuple `(1, 2, "three")` could look like this (forgetting about error handling for the moment; a better way to code this is shown below):

```
PyObject *t;

t = PyTuple_New(3);
PyTuple_SetItem(t, 0, PyLong_FromLong(1L));
PyTuple_SetItem(t, 1, PyLong_FromLong(2L));
PyTuple_SetItem(t, 2, PyUnicode_FromString("three"));
```

Here, `PyLong_FromLong()` returns a new reference which is immediately stolen by `PyTuple_SetItem()`. When you want to keep using an object although the reference to it will be stolen, use `Py_INCREF()` to grab another reference before calling the reference-stealing function.

Incidentally, `PyTuple_SetItem()` is the *only* way to set tuple items; `PySequence_SetItem()` and `PyObject_SetItem()` refuse to do this since tuples are an immutable data type. You should only use `PyTuple_SetItem()` for tuples that you are creating yourself.

Equivalent code for populating a list can be written using `PyList_New()` and `PyList_SetItem()`.

However, in practice, you will rarely use these ways of creating and populating a tuple or list. There’s a generic function, `Py_BuildValue()`, that can create most common objects from C values, directed by a *format string*. For example, the above two blocks of code could be replaced by the following (which also takes care of the error checking):

```
PyObject *tuple, *list;

tuple = Py_BuildValue("(iis)", 1, 2, "three");
list = Py_BuildValue("[iis]", 1, 2, "three");
```

It is much more common to use **`PyObject_SetItem()`** and friends with items whose references you are only borrowing, like arguments that were passed in to the function you are writing. In that case, their behaviour regarding references is much saner, since you don't have to take a new reference just so you can give that reference away ("have it be stolen"). For example, this function sets all items of a list (actually, any mutable sequence) to a given item:

```
int
set_all(PyObject *target, PyObject *item)
{
    Py_ssize_t i, n;

    n = PyObject_Length(target);
    if (n < 0)
        return -1;
    for (i = 0; i < n; i++) {
        PyObject *index = PyLong_FromSsize_t(i);
        if (!index)
            return -1;
        if (PyObject_SetItem(target, index, item) < 0) {
            Py_DECREF(index);
            return -1;
        }
        Py_DECREF(index);
    }
    return 0;
}
```

The situation is slightly different for function return values. While passing a reference to most functions does not change your ownership responsibilities for that reference, many functions that return a reference to an object give you ownership of the reference. The reason is simple: in many cases, the returned object is created on the fly, and the reference you get is the only reference to the object. Therefore, the generic functions that return object references, like **`PyObject_GetItem()`** and **`PySequence_GetItem()`**, always return a new reference (the caller becomes the owner of the reference).

It is important to realize that whether you own a reference returned by a function depends on which function you call only — *the plumage* (the type of the object passed as an argument to the function) *doesn’t enter into it!* Thus, if you extract an item from a list using `PyList_GetItem()`, you don’t own the reference — but if you obtain the same item from the same list using `PySequence_GetItem()` (which happens to take exactly the same arguments), you do own a reference to the returned object.

Here is an example of how you could write a function that computes the sum of the items in a list of integers; once using `PyList_GetItem()`, and once using `PySequence_GetItem()`.

```
long
sum_list(PyObject *list)
{
    Py_ssize_t i, n;
    long total = 0, value;
    PyObject *item;

    n = PyList_Size(list);
    if (n < 0)
        return -1; /* Not a list */
    for (i = 0; i < n; i++) {
        item = PyList_GetItem(list, i); /* Can't fail */
        if (!PyLong_Check(item)) continue; /* Skip non-integers
*/
        value = PyLong_AsLong(item);
        if (value == -1 && PyErr_Occurred())
            /* Integer too big to fit in a C long, bail out */
            return -1;
        total += value;
    }
    return total;
}
```

```
long
sum_sequence(PyObject *sequence)
{
    Py_ssize_t i, n;
    long total = 0, value;
    PyObject *item;
    n = PySequence_Length(sequence);
    if (n < 0)
        return -1; /* Has no length */
    for (i = 0; i < n; i++) {
        item = PySequence_GetItem(sequence, i);
        if (item == NULL)
            return -1; /* Not a sequence, or other failure */
        if (PyLong_Check(item)) {
            value = PyLong_AsLong(item);
            Py_DECREF(item);
            if (value == -1 && PyErr_Occurred())
                /* Integer too big to fit in a C long, bail out
*/
                return -1;
            total += value;
        }
        else {
            Py_DECREF(item); /* Discard reference ownership */
        }
    }
    return total;
}
```

Types

There are few other data types that play a significant role in the Python/C API; most are simple C types such as int, long, double and char*. A few structure types are used to describe static tables used to list the functions exported by a module or the data attributes of a new object type, and another is used to describe the value of a complex number. These will be discussed together with the functions that use them.

type Py_ssize_t

Part of the Stable ABI.

A signed integral type such that `sizeof(Py_ssize_t) == sizeof(size_t)`. C99 doesn't define such a thing directly (size_t is an unsigned integral type). See **PEP 353** [https://peps.python.org/pep-0353/] for details. `PY_SSIZE_T_MAX` is the largest positive value of type `Py_ssize_t`.

Exceptions

The Python programmer only needs to deal with exceptions if specific error handling is required; unhandled exceptions are automatically propagated to the caller, then to the caller's caller, and so on, until they reach the top-level interpreter, where they are reported to the user accompanied by a stack traceback.

For C programmers, however, error checking always has to be explicit. All functions in the Python/C API can raise exceptions, unless an explicit claim is made otherwise in a function's documentation. In general, when a function encounters an error, it sets an exception, discards any object references that it owns, and returns an error indicator. If not documented otherwise, this indicator is either `NULL` or `-1`, depending on the function's return type. A few functions return a Boolean true/false result, with false indicating an error. Very few functions return no explicit error indicator or have an ambiguous return value, and require explicit testing for errors with `PyErr_Occurred()`. These exceptions are always explicitly documented.

Exception state is maintained in per-thread storage (this is equivalent to using global storage in an unthreaded application). A thread can be in one of two states: an exception has occurred, or not. The function `PyErr_Occurred()`

can be used to check for this: it returns a borrowed reference to the exception type object when an exception has occurred, and `NULL` otherwise. There are a number of functions to set the exception state: **`PyErr_SetString()`** is the most common (though not the most general) function to set the exception state, and **`PyErr_Clear()`** clears the exception state.

The full exception state consists of three objects (all of which can be `NULL`): the exception type, the corresponding exception value, and the traceback. These have the same meanings as the Python result of `sys.exc_info()`; however, they are not the same: the Python objects represent the last exception being handled by a Python **`try`** … **`except`** statement, while the C level exception state only exists while an exception is being passed on between C functions until it reaches the Python bytecode interpreter's main loop, which takes care of transferring it to `sys.exc_info()` and friends.

Note that starting with Python 1.5, the preferred, thread-safe way to access the exception state from Python code is to call the function **`sys.exc_info()`**, which returns the per-thread exception state for Python code. Also, the semantics of both ways to access the exception state have changed so that a function which catches an exception will save and restore its thread's exception state so as to preserve the exception state of its caller. This prevents common bugs in exception handling code caused by an innocent-looking function overwriting the exception being handled; it also reduces the often unwanted lifetime extension for objects that are referenced by the stack frames in the traceback.

As a general principle, a function that calls another function to perform some task should check whether the called function raised an exception, and if so, pass the exception state on to its caller. It should discard any object references that it owns, and return an error indicator, but it should *not* set another exception — that would overwrite the exception that was just raised, and lose important information about the exact cause of the error.

A simple example of detecting exceptions and passing them on is shown in the **`sum_sequence()`** example above. It so happens that this example doesn't need to clean up any owned references when it detects an error. The following example function shows some error cleanup. First, to remind you why you like Python, we show the equivalent Python code:

```
def incr_item(dict, key):
    try:
        item = dict[key]
    except KeyError:
        item = 0
    dict[key] = item + 1
```

Here is the corresponding C code, in all its glory:

```
int
incr_item(PyObject *dict, PyObject *key)
{
    /* Objects all initialized to NULL for Py_XDECREF */
    PyObject *item = NULL, *const_one = NULL, *incremented_item
= NULL;
    int rv = -1; /* Return value initialized to -1 (failure) */

    item = PyObject_GetItem(dict, key);
    if (item == NULL) {
        /* Handle KeyError only: */
        if (!PyErr_ExceptionMatches(PyExc_KeyError))
            goto error;

        /* Clear the error and use zero: */
        PyErr_Clear();
        item = PyLong_FromLong(0L);
        if (item == NULL)
            goto error;
    }
    const_one = PyLong_FromLong(1L);
    if (const_one == NULL)
        goto error;

    incremented_item = PyNumber_Add(item, const_one);
    if (incremented_item == NULL)
        goto error;

    if (PyObject_SetItem(dict, key, incremented_item) < 0)
        goto error;
    rv = 0; /* Success */
    /* Continue with cleanup code */

 error:
    /* Cleanup code, shared by success and failure path */

    /* Use Py_XDECREF() to ignore NULL references */
    Py_XDECREF(item);
    Py_XDECREF(const_one);
    Py_XDECREF(incremented_item);

    return rv; /* -1 for error, 0 for success */
}
```

This example represents an endorsed use of the `goto` statement in C! It illustrates the use of **`PyErr_ExceptionMatches()`** and **`PyErr_Clear()`** to handle specific exceptions, and the use of **`Py_XDECREF()`** to dispose of owned references that may be `NULL` (note the `'X'` in the name; **`Py_DECREF()`** would crash when confronted with a `NULL` reference). It is important that the variables used to hold owned references are initialized to `NULL` for this to work; likewise, the proposed return value is initialized to `-1` (failure) and only set to success after the final call made is successful.

Embedding Python

The one important task that only embedders (as opposed to extension writers) of the Python interpreter have to worry about is the initialization, and possibly the finalization, of the Python interpreter. Most functionality of the interpreter can only be used after the interpreter has been initialized.

The basic initialization function is **`Py_Initialize()`**. This initializes the table of loaded modules, and creates the fundamental modules **`builtins`**, **`__main__`**, and **`sys`**. It also initializes the module search path (`sys.path`).

`Py_Initialize()` does not set the “script argument list” (`sys.argv`). If this variable is needed by Python code that will be executed later, setting **`PyConfig.argv`** and **`PyConfig.parse_argv`** must be set: see Python Initialization Configuration.

On most systems (in particular, on Unix and Windows, although the details are slightly different), **`Py_Initialize()`** calculates the module search path based upon its best guess for the location of the standard Python interpreter executable, assuming that the Python library is found in a fixed location relative to the Python interpreter executable. In particular, it looks for a directory named `lib/python`*`X.Y`* relative to the parent directory where the executable named `python` is found on the shell command search path (the environment variable **`PATH`**).

For instance, if the Python executable is found in `/usr/local/bin/python`, it will assume that the libraries are in `/usr/local/lib/python`*`X.Y`*. (In fact, this particular path is also the “fallback” location, used when no executable file named `python` is found along **`PATH`**.) The user can override this behavior by setting the environment variable **`PYTHONHOME`**, or insert additional directories in front of the standard path by setting **`PYTHONPATH`**.

The embedding application can steer the search by setting `PyConfig.program_name` *before* calling `Py_InitializeFromConfig()`. Note that `PYTHONHOME` still overrides this and `PYTHONPATH` is still inserted in front of the standard path. An application that requires total control has to provide its own implementation of `Py_GetPath()`, `Py_GetPrefix()`, `Py_GetExecPrefix()`, and `Py_GetProgramFullPath()` (all defined in `Modules/getpath.c`).

Sometimes, it is desirable to “uninitialize” Python. For instance, the application may want to start over (make another call to `Py_Initialize()`) or the application is simply done with its use of Python and wants to free memory allocated by Python. This can be accomplished by calling `Py_FinalizeEx()`. The function `Py_IsInitialized()` returns true if Python is currently in the initialized state. More information about these functions is given in a later chapter. Notice that `Py_FinalizeEx()` does *not* free all memory allocated by the Python interpreter, e.g. memory allocated by extension modules currently cannot be released.

Debugging Builds

Python can be built with several macros to enable extra checks of the interpreter and extension modules. These checks tend to add a large amount of overhead to the runtime so they are not enabled by default.

A full list of the various types of debugging builds is in the file `Misc/SpecialBuilds.txt` in the Python source distribution. Builds are available that support tracing of reference counts, debugging the memory allocator, or low-level profiling of the main interpreter loop. Only the most frequently used builds will be described in the remainder of this section.

Py_DEBUG

Compiling the interpreter with the **`Py_DEBUG`** macro defined produces what is generally meant by a debug build of Python. **`Py_DEBUG`** is enabled in the Unix build by adding **`--with-pydebug`** to the `./configure` command. It is also implied by the presence of the not-Python-specific **`_DEBUG`** macro. When **`Py_DEBUG`** is enabled in the Unix build, compiler optimization is disabled.

In addition to the reference count debugging described below, extra checks are performed, see Python Debug Build.

Defining **`Py_TRACE_REFS`** enables reference tracing (see the **`configure --with-trace-refs option`**). When defined, a circular doubly linked list of active objects is maintained by adding two extra fields to every **`PyObject`**. Total allocations are tracked as well. Upon exit, all existing references are printed. (In interactive mode this happens after every statement run by the interpreter.)

Please refer to `Misc/SpecialBuilds.txt` in the Python source distribution for more detailed information.

Recommended third party tools

The following third party tools offer both simpler and more sophisticated approaches to creating C, C++ and Rust extensions for Python:

- Cython [https://cython.org/]
- cffi [https://cffi.readthedocs.io]
- HPy [https://hpyproject.org/]
- nanobind [https://github.com/wjakob/nanobind] (C++)
- Numba [https://numba.pydata.org/]
- pybind11 [https://pybind11.readthedocs.io/] (C++)
- PyO3 [https://pyo3.rs/] (Rust)
- SWIG [https://www.swig.org]

Using tools such as these can help avoid writing code that is tightly bound to a particular version of CPython, avoid reference counting errors, and focus more on your own code than on using the CPython API. In general, new versions of Python can be supported by updating the tool, and your code will often use newer and more efficient APIs automatically. Some tools also support compiling for other implementations of Python from a single set of sources.

These projects are not supported by the same people who maintain Python, and issues need to be raised with the projects directly. Remember to check that the project is still maintained and supported, as the list above may become outdated.

See also

Python Packaging User Guide: Binary Extensions

[https://packaging.python.org/guides/packaging-binary-extensions/]

The Python Packaging User Guide not only covers several available tools that simplify the creation of binary extensions, but also discusses the various reasons why creating an extension module may be desirable in the first place.

C API Stability

Unless documented otherwise, Python’s C API is covered by the Backwards Compatibility Policy, **PEP 387** [https://peps.python.org/pep-0387/]. Most changes to it are source-compatible (typically by only adding new API). Changing existing API or removing API is only done after a deprecation period or to fix serious issues.

CPython’s Application Binary Interface (ABI) is forward- and backwards-compatible across a minor release (if these are compiled the same way; see Platform Considerations below). So, code compiled for Python 3.10.0 will work on 3.10.8 and vice versa, but will need to be compiled separately for 3.9.x and 3.11.x.

There are two tiers of C API with different stability expectations:

- Unstable API, may change in minor versions without a deprecation period. It is marked by the `PyUnstable` prefix in names.
- Limited API, is compatible across several minor releases. When **`Py_LIMITED_API`** is defined, only this subset is exposed from `Python.h`.

These are discussed in more detail below.

Names prefixed by an underscore, such as `_Py_InternalState`, are private API that can change without notice even in patch releases. If you need to use this API, consider reaching out to CPython developers [https://discuss.python.org/c/core-dev/c-api/30] to discuss adding public API for your use case.

Unstable C API

Any API named with the `PyUnstable` prefix exposes CPython implementation details, and may change in every minor release (e.g. from 3.9 to 3.10) without any deprecation warnings. However, it will not change in a bugfix release (e.g. from 3.10.0 to 3.10.1).

It is generally intended for specialized, low-level tools like debuggers.

Projects that use this API are expected to follow CPython development and spend extra effort adjusting to changes.

Stable Application Binary Interface

For simplicity, this document talks about *extensions*, but the Limited API and Stable ABI work the same way for all uses of the API – for example, embedding Python.

Limited C API

Python 3.2 introduced the *Limited API*, a subset of Python's C API. Extensions that only use the Limited API can be compiled once and be loaded on multiple versions of Python. Contents of the Limited API are listed below.

Py_LIMITED_API

Define this macro before including `Python.h` to opt in to only use the Limited API, and to select the Limited API version.

Define `Py_LIMITED_API` to the value of **`PY_VERSION_HEX`** corresponding to the lowest Python version your extension supports. The extension will be ABI-compatible with all Python 3 releases from the specified one onward, and can use Limited API introduced up to that version.

Rather than using the `PY_VERSION_HEX` macro directly, hardcode a minimum minor version (e.g. `0x030A0000` for Python 3.10) for stability when compiling with future Python versions.

You can also define `Py_LIMITED_API` to `3`. This works the same as `0x03020000` (Python 3.2, the version that introduced Limited API).

Stable ABI

To enable this, Python provides a *Stable ABI*: a set of symbols that will remain ABI-compatible across Python 3.x versions.

Note

The Stable ABI prevents ABI issues, like linker errors due to missing symbols or data corruption due to changes in structure layouts or function signatures. However, other changes in Python can change the *behavior* of extensions. See Python's Backwards Compatibility Policy (**PEP 387** [https://peps.python.org/pep-0387/]) for details.

The Stable ABI contains symbols exposed in the Limited API, but also other ones – for example, functions necessary to support older versions of the Limited API.

On Windows, extensions that use the Stable ABI should be linked against `python3.dll` rather than a version-specific library such as `python39.dll`.

On some platforms, Python will look for and load shared library files named with the `abi3` tag (e.g. `mymodule.abi3.so`). It does not check if such extensions conform to a Stable ABI. The user (or their packaging tools) need to ensure that, for example, extensions built with the 3.10+ Limited API are not installed for lower versions of Python.

All functions in the Stable ABI are present as functions in Python’s shared library, not solely as macros. This makes them usable from languages that don’t use the C preprocessor.

Limited API Scope and Performance

The goal for the Limited API is to allow everything that is possible with the full C API, but possibly with a performance penalty.

For example, while `PyList_GetItem()` is available, its “unsafe” macro variant `PyList_GET_ITEM()` is not. The macro can be faster because it can rely on version-specific implementation details of the list object.

Without `Py_LIMITED_API` defined, some C API functions are inlined or replaced by macros. Defining `Py_LIMITED_API` disables this inlining, allowing stability as Python’s data structures are improved, but possibly reducing performance.

By leaving out the `Py_LIMITED_API` definition, it is possible to compile a Limited API extension with a version-specific ABI. This can improve performance for that Python version, but will limit compatibility. Compiling with `Py_LIMITED_API` will then yield an extension that can be distributed where a version-specific one is not available – for example, for prereleases of an upcoming Python version.

Limited API Caveats

Note that compiling with `Py_LIMITED_API` is *not* a complete guarantee that code conforms to the Limited API or the Stable ABI. `Py_LIMITED_API` only covers definitions, but an API also includes other issues, such as expected semantics.

One issue that `Py_LIMITED_API` does not guard against is calling a function with arguments that are invalid in a lower Python version. For example, consider a function that starts accepting `NULL` for an argument. In Python 3.9, `NULL` now selects a default behavior, but in Python 3.8, the argument will be used directly, causing a `NULL` dereference and crash. A similar argument works for fields of structs.

Another issue is that some struct fields are currently not hidden when `Py_LIMITED_API` is defined, even though they're part of the Limited API.

For these reasons, we recommend testing an extension with *all* minor Python versions it supports, and preferably to build with the *lowest* such version.

We also recommend reviewing documentation of all used API to check if it is explicitly part of the Limited API. Even with `Py_LIMITED_API` defined, a few private declarations are exposed for technical reasons (or even unintentionally, as bugs).

Also note that the Limited API is not necessarily stable: compiling with `Py_LIMITED_API` with Python 3.8 means that the extension will run with Python 3.12, but it will not necessarily *compile* with Python 3.12. In particular, parts of the Limited API may be deprecated and removed, provided that the Stable ABI stays stable.

Platform Considerations

ABI stability depends not only on Python, but also on the compiler used, lower-level libraries and compiler options. For the purposes of the Stable ABI, these details define a “platform”. They usually depend on the OS type and processor architecture

It is the responsibility of each particular distributor of Python to ensure that all Python versions on a particular platform are built in a way that does not break the Stable ABI. This is the case with Windows and macOS releases from `python.org` and many third-party distributors.

Contents of Limited API

Currently, the Limited API includes the following items:

- `METH_CLASS`
- `METH_COEXIST`
- `METH_FASTCALL`
- `METH_METHOD`
- `METH_NOARGS`
- `METH_O`
- `METH_STATIC`
- `METH_VARARGS`
- `PY_VECTORCALL_ARGUMENTS_OFFSET`
- `PyAIter_Check()`
- `PyArg_Parse()`
- `PyArg_ParseTuple()`
- `PyArg_ParseTupleAndKeywords()`
- `PyArg_UnpackTuple()`
- `PyArg_VaParse()`
- `PyArg_VaParseTupleAndKeywords()`
- `PyArg_ValidateKeywordArguments()`

- PyBUF_ANY_CONTIGUOUS
- PyBUF_CONTIG
- PyBUF_CONTIG_RO
- PyBUF_C_CONTIGUOUS
- PyBUF_FORMAT
- PyBUF_FULL
- PyBUF_FULL_RO
- PyBUF_F_CONTIGUOUS
- PyBUF_INDIRECT
- PyBUF_MAX_NDIM
- PyBUF_ND
- PyBUF_READ
- PyBUF_RECORDS
- PyBUF_RECORDS_RO
- PyBUF_SIMPLE
- PyBUF_STRIDED
- PyBUF_STRIDED_RO
- PyBUF_STRIDES
- PyBUF_WRITABLE
- PyBUF_WRITE
- PyBaseObject_Type
- PyBool_FromLong()
- PyBool_Type
- PyBuffer_FillContiguousStrides()
- PyBuffer_FillInfo()
- PyBuffer_FromContiguous()
- PyBuffer_GetPointer()
- PyBuffer_IsContiguous()
- PyBuffer_Release()
- PyBuffer_SizeFromFormat()
- PyBuffer_ToContiguous()
- PyByteArrayIter_Type
- PyByteArray_AsString()
- PyByteArray_Concat()
- PyByteArray_FromObject()
- PyByteArray_FromStringAndSize()

- PyByteArray_Resize()
- PyByteArray_Size()
- PyByteArray_Type
- PyBytesIter_Type
- PyBytes_AsString()
- PyBytes_AsStringAndSize()
- PyBytes_Concat()
- PyBytes_ConcatAndDel()
- PyBytes_DecodeEscape()
- PyBytes_FromFormat()
- PyBytes_FromFormatV()
- PyBytes_FromObject()
- PyBytes_FromString()
- PyBytes_FromStringAndSize()
- PyBytes_Repr()
- PyBytes_Size()
- PyBytes_Type
- PyCFunction
- PyCFunctionFast
- PyCFunctionFastWithKeywords
- PyCFunctionWithKeywords
- PyCFunction_GetFlags()
- PyCFunction_GetFunction()
- PyCFunction_GetSelf()
- PyCFunction_New()
- PyCFunction_NewEx()
- PyCFunction_Type
- PyCMethod_New()
- PyCallIter_New()
- PyCallIter_Type
- PyCallable_Check()
- PyCapsule_Destructor
- PyCapsule_GetContext()
- PyCapsule_GetDestructor()
- PyCapsule_GetName()
- PyCapsule_GetPointer()

- `PyCapsule_Import()`
- `PyCapsule_IsValid()`
- `PyCapsule_New()`
- `PyCapsule_SetContext()`
- `PyCapsule_SetDestructor()`
- `PyCapsule_SetName()`
- `PyCapsule_SetPointer()`
- `PyCapsule_Type`
- `PyClassMethodDescr_Type`
- `PyCodec_BackslashReplaceErrors()`
- `PyCodec_Decode()`
- `PyCodec_Decoder()`
- `PyCodec_Encode()`
- `PyCodec_Encoder()`
- `PyCodec_IgnoreErrors()`
- `PyCodec_IncrementalDecoder()`
- `PyCodec_IncrementalEncoder()`
- `PyCodec_KnownEncoding()`
- `PyCodec_LookupError()`
- `PyCodec_NameReplaceErrors()`
- `PyCodec_Register()`
- `PyCodec_RegisterError()`
- `PyCodec_ReplaceErrors()`
- `PyCodec_StreamReader()`
- `PyCodec_StreamWriter()`
- `PyCodec_StrictErrors()`
- `PyCodec_Unregister()`
- `PyCodec_XMLCharRefReplaceErrors()`
- `PyComplex_FromDoubles()`
- `PyComplex_ImagAsDouble()`
- `PyComplex_RealAsDouble()`
- `PyComplex_Type`
- `PyDescr_NewClassMethod()`
- `PyDescr_NewGetSet()`
- `PyDescr_NewMember()`
- `PyDescr_NewMethod()`

- PyDictItems_Type
- PyDictIterItem_Type
- PyDictIterKey_Type
- PyDictIterValue_Type
- PyDictKeys_Type
- PyDictProxy_New()
- PyDictProxy_Type
- PyDictRevIterItem_Type
- PyDictRevIterKey_Type
- PyDictRevIterValue_Type
- PyDictValues_Type
- PyDict_Clear()
- PyDict_Contains()
- PyDict_Copy()
- PyDict_DelItem()
- PyDict_DelItemString()
- PyDict_GetItem()
- PyDict_GetItemRef()
- PyDict_GetItemString()
- PyDict_GetItemStringRef()
- PyDict_GetItemWithError()
- PyDict_Items()
- PyDict_Keys()
- PyDict_Merge()
- PyDict_MergeFromSeq2()
- PyDict_New()
- PyDict_Next()
- PyDict_SetItem()
- PyDict_SetItemString()
- PyDict_Size()
- PyDict_Type
- PyDict_Update()
- PyDict_Values()
- PyEllipsis_Type
- PyEnum_Type
- PyErr_BadArgument()

- `PyErr_BadInternalCall()`
- `PyErr_CheckSignals()`
- `PyErr_Clear()`
- `PyErr_Display()`
- `PyErr_DisplayException()`
- `PyErr_ExceptionMatches()`
- `PyErr_Fetch()`
- `PyErr_Format()`
- `PyErr_FormatV()`
- `PyErr_GetExcInfo()`
- `PyErr_GetHandledException()`
- `PyErr_GetRaisedException()`
- `PyErr_GivenExceptionMatches()`
- `PyErr_NewException()`
- `PyErr_NewExceptionWithDoc()`
- `PyErr_NoMemory()`
- `PyErr_NormalizeException()`
- `PyErr_Occurred()`
- `PyErr_Print()`
- `PyErr_PrintEx()`
- `PyErr_ProgramText()`
- `PyErr_ResourceWarning()`
- `PyErr_Restore()`
- `PyErr_SetExcFromWindowsErr()`
- `PyErr_SetExcFromWindowsErrWithFilename()`
- `PyErr_SetExcFromWindowsErrWithFilenameObject()`
- `PyErr_SetExcFromWindowsErrWithFilenameObjects()`
- `PyErr_SetExcInfo()`
- `PyErr_SetFromErrno()`
- `PyErr_SetFromErrnoWithFilename()`
- `PyErr_SetFromErrnoWithFilenameObject()`
- `PyErr_SetFromErrnoWithFilenameObjects()`
- `PyErr_SetFromWindowsErr()`
- `PyErr_SetFromWindowsErrWithFilename()`
- `PyErr_SetHandledException()`
- `PyErr_SetImportError()`

- PyErr_SetImportErrorSubclass()
- PyErr_SetInterrupt()
- PyErr_SetInterruptEx()
- PyErr_SetNone()
- PyErr_SetObject()
- PyErr_SetRaisedException()
- PyErr_SetString()
- PyErr_SyntaxLocation()
- PyErr_SyntaxLocationEx()
- PyErr_WarnEx()
- PyErr_WarnExplicit()
- PyErr_WarnFormat()
- PyErr_WriteUnraisable()
- PyEval_AcquireThread()
- PyEval_EvalCode()
- PyEval_EvalCodeEx()
- PyEval_EvalFrame()
- PyEval_EvalFrameEx()
- PyEval_GetBuiltins()
- PyEval_GetFrame()
- PyEval_GetFrameBuiltins()
- PyEval_GetFrameGlobals()
- PyEval_GetFrameLocals()
- PyEval_GetFuncDesc()
- PyEval_GetFuncName()
- PyEval_GetGlobals()
- PyEval_GetLocals()
- PyEval_InitThreads()
- PyEval_ReleaseThread()
- PyEval_RestoreThread()
- PyEval_SaveThread()
- PyExc_ArithmeticError
- PyExc_AssertionError
- PyExc_AttributeError
- PyExc_BaseException
- PyExc_BaseExceptionGroup

- PyExc_BlockingIOError
- PyExc_BrokenPipeError
- PyExc_BufferError
- PyExc_BytesWarning
- PyExc_ChildProcessError
- PyExc_ConnectionAbortedError
- PyExc_ConnectionError
- PyExc_ConnectionRefusedError
- PyExc_ConnectionResetError
- PyExc_DeprecationWarning
- PyExc_EOFError
- PyExc_EncodingWarning
- PyExc_EnvironmentError
- PyExc_Exception
- PyExc_FileExistsError
- PyExc_FileNotFoundError
- PyExc_FloatingPointError
- PyExc_FutureWarning
- PyExc_GeneratorExit
- PyExc_IOError
- PyExc_ImportError
- PyExc_ImportWarning
- PyExc_IndentationError
- PyExc_IndexError
- PyExc_InterruptedError
- PyExc_IsADirectoryError
- PyExc_KeyError
- PyExc_KeyboardInterrupt
- PyExc_LookupError
- PyExc_MemoryError
- PyExc_ModuleNotFoundError
- PyExc_NameError
- PyExc_NotADirectoryError
- PyExc_NotImplementedError
- PyExc_OSError
- PyExc_OverflowError

- PyExc_PendingDeprecationWarning
- PyExc_PermissionError
- PyExc_ProcessLookupError
- PyExc_RecursionError
- PyExc_ReferenceError
- PyExc_ResourceWarning
- PyExc_RuntimeError
- PyExc_RuntimeWarning
- PyExc_StopAsyncIteration
- PyExc_StopIteration
- PyExc_SyntaxError
- PyExc_SyntaxWarning
- PyExc_SystemError
- PyExc_SystemExit
- PyExc_TabError
- PyExc_TimeoutError
- PyExc_TypeError
- PyExc_UnboundLocalError
- PyExc_UnicodeDecodeError
- PyExc_UnicodeEncodeError
- PyExc_UnicodeError
- PyExc_UnicodeTranslateError
- PyExc_UnicodeWarning
- PyExc_UserWarning
- PyExc_ValueError
- PyExc_Warning
- PyExc_WindowsError
- PyExc_ZeroDivisionError
- PyExceptionClass_Name()
- PyException_GetArgs()
- PyException_GetCause()
- PyException_GetContext()
- PyException_GetTraceback()
- PyException_SetArgs()
- PyException_SetCause()
- PyException_SetContext()

- PyException_SetTraceback()
- PyFile_FromFd()
- PyFile_GetLine()
- PyFile_WriteObject()
- PyFile_WriteString()
- PyFilter_Type
- PyFloat_AsDouble()
- PyFloat_FromDouble()
- PyFloat_FromString()
- PyFloat_GetInfo()
- PyFloat_GetMax()
- PyFloat_GetMin()
- PyFloat_Type
- PyFrameObject
- PyFrame_GetCode()
- PyFrame_GetLineNumber()
- PyFrozenSet_New()
- PyFrozenSet_Type
- PyGC_Collect()
- PyGC_Disable()
- PyGC_Enable()
- PyGC_IsEnabled()
- PyGILState_Ensure()
- PyGILState_GetThisThreadState()
- PyGILState_Release()
- PyGILState_STATE
- PyGetSetDef
- PyGetSetDescr_Type
- PyImport_AddModule()
- PyImport_AddModuleObject()
- PyImport_AddModuleRef()
- PyImport_AppendInittab()
- PyImport_ExecCodeModule()
- PyImport_ExecCodeModuleEx()
- PyImport_ExecCodeModuleObject()
- PyImport_ExecCodeModuleWithPathnames()

- PyImport_GetImporter()
- PyImport_GetMagicNumber()
- PyImport_GetMagicTag()
- PyImport_GetModule()
- PyImport_GetModuleDict()
- PyImport_Import()
- PyImport_ImportFrozenModule()
- PyImport_ImportFrozenModuleObject()
- PyImport_ImportModule()
- PyImport_ImportModuleLevel()
- PyImport_ImportModuleLevelObject()
- PyImport_ImportModuleNoBlock()
- PyImport_ReloadModule()
- PyIndex_Check()
- PyInterpreterState
- PyInterpreterState_Clear()
- PyInterpreterState_Delete()
- PyInterpreterState_Get()
- PyInterpreterState_GetDict()
- PyInterpreterState_GetID()
- PyInterpreterState_New()
- PyIter_Check()
- PyIter_Next()
- PyIter_NextItem()
- PyIter_Send()
- PyListIter_Type
- PyListRevIter_Type
- PyList_Append()
- PyList_AsTuple()
- PyList_GetItem()
- PyList_GetItemRef()
- PyList_GetSlice()
- PyList_Insert()
- PyList_New()
- PyList_Reverse()
- PyList_SetItem()

- PyList_SetSlice()
- PyList_Size()
- PyList_Sort()
- PyList_Type
- PyLongObject
- PyLongRangeIter_Type
- PyLong_AsDouble()
- PyLong_AsInt()
- PyLong_AsInt32()
- PyLong_AsInt64()
- PyLong_AsLong()
- PyLong_AsLongAndOverflow()
- PyLong_AsLongLong()
- PyLong_AsLongLongAndOverflow()
- PyLong_AsNativeBytes()
- PyLong_AsSize_t()
- PyLong_AsSsize_t()
- PyLong_AsUInt32()
- PyLong_AsUInt64()
- PyLong_AsUnsignedLong()
- PyLong_AsUnsignedLongLong()
- PyLong_AsUnsignedLongLongMask()
- PyLong_AsUnsignedLongMask()
- PyLong_AsVoidPtr()
- PyLong_FromDouble()
- PyLong_FromInt32()
- PyLong_FromInt64()
- PyLong_FromLong()
- PyLong_FromLongLong()
- PyLong_FromNativeBytes()
- PyLong_FromSize_t()
- PyLong_FromSsize_t()
- PyLong_FromString()
- PyLong_FromUInt32()
- PyLong_FromUInt64()
- PyLong_FromUnsignedLong()

- PyLong_FromUnsignedLongLong()
- PyLong_FromUnsignedNativeBytes()
- PyLong_FromVoidPtr()
- PyLong_GetInfo()
- PyLong_Type
- PyMap_Type
- PyMapping_Check()
- PyMapping_GetItemString()
- PyMapping_GetOptionalItem()
- PyMapping_GetOptionalItemString()
- PyMapping_HasKey()
- PyMapping_HasKeyString()
- PyMapping_HasKeyStringWithError()
- PyMapping_HasKeyWithError()
- PyMapping_Items()
- PyMapping_Keys()
- PyMapping_Length()
- PyMapping_SetItemString()
- PyMapping_Size()
- PyMapping_Values()
- PyMem_Calloc()
- PyMem_Free()
- PyMem_Malloc()
- PyMem_RawCalloc()
- PyMem_RawFree()
- PyMem_RawMalloc()
- PyMem_RawRealloc()
- PyMem_Realloc()
- PyMemberDef
- PyMemberDescr_Type
- PyMember_GetOne()
- PyMember_SetOne()
- PyMemoryView_FromBuffer()
- PyMemoryView_FromMemory()
- PyMemoryView_FromObject()
- PyMemoryView_GetContiguous()

- PyMemoryView_Type
- PyMethodDef
- PyMethodDescr_Type
- PyModuleDef
- PyModuleDef_Base
- PyModuleDef_Init()
- PyModuleDef_Slot
- PyModuleDef_Type
- PyModule_Add()
- PyModule_AddFunctions()
- PyModule_AddIntConstant()
- PyModule_AddObject()
- PyModule_AddObjectRef()
- PyModule_AddStringConstant()
- PyModule_AddType()
- PyModule_Create2()
- PyModule_ExecDef()
- PyModule_FromDefAndSpec2()
- PyModule_GetDef()
- PyModule_GetDict()
- PyModule_GetFilename()
- PyModule_GetFilenameObject()
- PyModule_GetName()
- PyModule_GetNameObject()
- PyModule_GetState()
- PyModule_New()
- PyModule_NewObject()
- PyModule_SetDocString()
- PyModule_Type
- PyNumber_Absolute()
- PyNumber_Add()
- PyNumber_And()
- PyNumber_AsSsize_t()
- PyNumber_Check()
- PyNumber_Divmod()
- PyNumber_Float()

- PyNumber_FloorDivide()
- PyNumber_InPlaceAdd()
- PyNumber_InPlaceAnd()
- PyNumber_InPlaceFloorDivide()
- PyNumber_InPlaceLshift()
- PyNumber_InPlaceMatrixMultiply()
- PyNumber_InPlaceMultiply()
- PyNumber_InPlaceOr()
- PyNumber_InPlacePower()
- PyNumber_InPlaceRemainder()
- PyNumber_InPlaceRshift()
- PyNumber_InPlaceSubtract()
- PyNumber_InPlaceTrueDivide()
- PyNumber_InPlaceXor()
- PyNumber_Index()
- PyNumber_Invert()
- PyNumber_Long()
- PyNumber_Lshift()
- PyNumber_MatrixMultiply()
- PyNumber_Multiply()
- PyNumber_Negative()
- PyNumber_Or()
- PyNumber_Positive()
- PyNumber_Power()
- PyNumber_Remainder()
- PyNumber_Rshift()
- PyNumber_Subtract()
- PyNumber_ToBase()
- PyNumber_TrueDivide()
- PyNumber_Xor()
- PyOS_AfterFork()
- PyOS_AfterFork_Child()
- PyOS_AfterFork_Parent()
- PyOS_BeforeFork()
- PyOS_CheckStack()
- PyOS_FSPath()

- PyOS_InputHook
- PyOS_InterruptOccurred()
- PyOS_double_to_string()
- PyOS_getsig()
- PyOS_mystricmp()
- PyOS_mystrnicmp()
- PyOS_setsig()
- PyOS_sighandler_t
- PyOS_snprintf()
- PyOS_string_to_double()
- PyOS_strtol()
- PyOS_strtoul()
- PyOS_vsnprintf()
- PyObject
- PyObject.ob_refcnt
- PyObject.ob_type
- PyObject_ASCII()
- PyObject_AsFileDescriptor()
- PyObject_Bytes()
- PyObject_Call()
- PyObject_CallFunction()
- PyObject_CallFunctionObjArgs()
- PyObject_CallMethod()
- PyObject_CallMethodObjArgs()
- PyObject_CallNoArgs()
- PyObject_CallObject()
- PyObject_Calloc()
- PyObject_CheckBuffer()
- PyObject_ClearWeakRefs()
- PyObject_CopyData()
- PyObject_DelAttr()
- PyObject_DelAttrString()
- PyObject_DelItem()
- PyObject_DelItemString()
- PyObject_Dir()
- PyObject_Format()

- `PyObject_Free()`
- `PyObject_GC_Del()`
- `PyObject_GC_IsFinalized()`
- `PyObject_GC_IsTracked()`
- `PyObject_GC_Track()`
- `PyObject_GC_UnTrack()`
- `PyObject_GenericGetAttr()`
- `PyObject_GenericGetDict()`
- `PyObject_GenericSetAttr()`
- `PyObject_GenericSetDict()`
- `PyObject_GetAIter()`
- `PyObject_GetAttr()`
- `PyObject_GetAttrString()`
- `PyObject_GetBuffer()`
- `PyObject_GetItem()`
- `PyObject_GetIter()`
- `PyObject_GetOptionalAttr()`
- `PyObject_GetOptionalAttrString()`
- `PyObject_GetTypeData()`
- `PyObject_HasAttr()`
- `PyObject_HasAttrString()`
- `PyObject_HasAttrStringWithError()`
- `PyObject_HasAttrWithError()`
- `PyObject_Hash()`
- `PyObject_HashNotImplemented()`
- `PyObject_Init()`
- `PyObject_InitVar()`
- `PyObject_IsInstance()`
- `PyObject_IsSubclass()`
- `PyObject_IsTrue()`
- `PyObject_Length()`
- `PyObject_Malloc()`
- `PyObject_Not()`
- `PyObject_Realloc()`
- `PyObject_Repr()`
- `PyObject_RichCompare()`

- PyObject_RichCompareBool()
- PyObject_SelfIter()
- PyObject_SetAttr()
- PyObject_SetAttrString()
- PyObject_SetItem()
- PyObject_Size()
- PyObject_Str()
- PyObject_Type()
- PyObject_Vectorcall()
- PyObject_VectorcallMethod()
- PyProperty_Type
- PyRangeIter_Type
- PyRange_Type
- PyReversed_Type
- PySeqIter_New()
- PySeqIter_Type
- PySequence_Check()
- PySequence_Concat()
- PySequence_Contains()
- PySequence_Count()
- PySequence_DelItem()
- PySequence_DelSlice()
- PySequence_Fast()
- PySequence_GetItem()
- PySequence_GetSlice()
- PySequence_In()
- PySequence_InPlaceConcat()
- PySequence_InPlaceRepeat()
- PySequence_Index()
- PySequence_Length()
- PySequence_List()
- PySequence_Repeat()
- PySequence_SetItem()
- PySequence_SetSlice()
- PySequence_Size()
- PySequence_Tuple()

- PySetIter_Type
- PySet_Add()
- PySet_Clear()
- PySet_Contains()
- PySet_Discard()
- PySet_New()
- PySet_Pop()
- PySet_Size()
- PySet_Type
- PySlice_AdjustIndices()
- PySlice_GetIndices()
- PySlice_GetIndicesEx()
- PySlice_New()
- PySlice_Type
- PySlice_Unpack()
- PyState_AddModule()
- PyState_FindModule()
- PyState_RemoveModule()
- PyStructSequence_Desc
- PyStructSequence_Field
- PyStructSequence_GetItem()
- PyStructSequence_New()
- PyStructSequence_NewType()
- PyStructSequence_SetItem()
- PyStructSequence_UnnamedField
- PySuper_Type
- PySys_Audit()
- PySys_AuditTuple()
- PySys_FormatStderr()
- PySys_FormatStdout()
- PySys_GetObject()
- PySys_GetXOptions()
- PySys_ResetWarnOptions()
- PySys_SetArgv()
- PySys_SetArgvEx()
- PySys_SetObject()

- PySys_WriteStderr()
- PySys_WriteStdout()
- PyThreadState
- PyThreadState_Clear()
- PyThreadState_Delete()
- PyThreadState_Get()
- PyThreadState_GetDict()
- PyThreadState_GetFrame()
- PyThreadState_GetID()
- PyThreadState_GetInterpreter()
- PyThreadState_New()
- PyThreadState_SetAsyncExc()
- PyThreadState_Swap()
- PyThread_GetInfo()
- PyThread_ReInitTLS()
- PyThread_acquire_lock()
- PyThread_acquire_lock_timed()
- PyThread_allocate_lock()
- PyThread_create_key()
- PyThread_delete_key()
- PyThread_delete_key_value()
- PyThread_exit_thread()
- PyThread_free_lock()
- PyThread_get_key_value()
- PyThread_get_stacksize()
- PyThread_get_thread_ident()
- PyThread_get_thread_native_id()
- PyThread_init_thread()
- PyThread_release_lock()
- PyThread_set_key_value()
- PyThread_set_stacksize()
- PyThread_start_new_thread()
- PyThread_tss_alloc()
- PyThread_tss_create()
- PyThread_tss_delete()
- PyThread_tss_free()

- PyThread_tss_get()
- PyThread_tss_is_created()
- PyThread_tss_set()
- PyTraceBack_Here()
- PyTraceBack_Print()
- PyTraceBack_Type
- PyTupleIter_Type
- PyTuple_GetItem()
- PyTuple_GetSlice()
- PyTuple_New()
- PyTuple_Pack()
- PyTuple_SetItem()
- PyTuple_Size()
- PyTuple_Type
- PyTypeObject
- PyType_ClearCache()
- PyType_Freeze()
- PyType_FromMetaclass()
- PyType_FromModuleAndSpec()
- PyType_FromSpec()
- PyType_FromSpecWithBases()
- PyType_GenericAlloc()
- PyType_GenericNew()
- PyType_GetBaseByToken()
- PyType_GetFlags()
- PyType_GetFullyQualifiedName()
- PyType_GetModule()
- PyType_GetModuleByDef()
- PyType_GetModuleName()
- PyType_GetModuleState()
- PyType_GetName()
- PyType_GetQualName()
- PyType_GetSlot()
- PyType_GetTypeDataSize()
- PyType_IsSubtype()
- PyType_Modified()

- PyType_Ready()
- PyType_Slot
- PyType_Spec
- PyType_Type
- PyUnicodeDecodeError_Create()
- PyUnicodeDecodeError_GetEncoding()
- PyUnicodeDecodeError_GetEnd()
- PyUnicodeDecodeError_GetObject()
- PyUnicodeDecodeError_GetReason()
- PyUnicodeDecodeError_GetStart()
- PyUnicodeDecodeError_SetEnd()
- PyUnicodeDecodeError_SetReason()
- PyUnicodeDecodeError_SetStart()
- PyUnicodeEncodeError_GetEncoding()
- PyUnicodeEncodeError_GetEnd()
- PyUnicodeEncodeError_GetObject()
- PyUnicodeEncodeError_GetReason()
- PyUnicodeEncodeError_GetStart()
- PyUnicodeEncodeError_SetEnd()
- PyUnicodeEncodeError_SetReason()
- PyUnicodeEncodeError_SetStart()
- PyUnicodeIter_Type
- PyUnicodeTranslateError_GetEnd()
- PyUnicodeTranslateError_GetObject()
- PyUnicodeTranslateError_GetReason()
- PyUnicodeTranslateError_GetStart()
- PyUnicodeTranslateError_SetEnd()
- PyUnicodeTranslateError_SetReason()
- PyUnicodeTranslateError_SetStart()
- PyUnicode_Append()
- PyUnicode_AppendAndDel()
- PyUnicode_AsASCIIString()
- PyUnicode_AsCharmapString()
- PyUnicode_AsDecodedObject()
- PyUnicode_AsDecodedUnicode()
- PyUnicode_AsEncodedObject()

- PyUnicode_AsEncodedString()
- PyUnicode_AsEncodedUnicode()
- PyUnicode_AsLatin1String()
- PyUnicode_AsMBCSString()
- PyUnicode_AsRawUnicodeEscapeString()
- PyUnicode_AsUCS4()
- PyUnicode_AsUCS4Copy()
- PyUnicode_AsUTF16String()
- PyUnicode_AsUTF32String()
- PyUnicode_AsUTF8AndSize()
- PyUnicode_AsUTF8String()
- PyUnicode_AsUnicodeEscapeString()
- PyUnicode_AsWideChar()
- PyUnicode_AsWideCharString()
- PyUnicode_BuildEncodingMap()
- PyUnicode_Compare()
- PyUnicode_CompareWithASCIIString()
- PyUnicode_Concat()
- PyUnicode_Contains()
- PyUnicode_Count()
- PyUnicode_Decode()
- PyUnicode_DecodeASCII()
- PyUnicode_DecodeCharmap()
- PyUnicode_DecodeCodePageStateful()
- PyUnicode_DecodeFSDefault()
- PyUnicode_DecodeFSDefaultAndSize()
- PyUnicode_DecodeLatin1()
- PyUnicode_DecodeLocale()
- PyUnicode_DecodeLocaleAndSize()
- PyUnicode_DecodeMBCS()
- PyUnicode_DecodeMBCSStateful()
- PyUnicode_DecodeRawUnicodeEscape()
- PyUnicode_DecodeUTF16()
- PyUnicode_DecodeUTF16Stateful()
- PyUnicode_DecodeUTF32()
- PyUnicode_DecodeUTF32Stateful()

- `PyUnicode_DecodeUTF7()`
- `PyUnicode_DecodeUTF7Stateful()`
- `PyUnicode_DecodeUTF8()`
- `PyUnicode_DecodeUTF8Stateful()`
- `PyUnicode_DecodeUnicodeEscape()`
- `PyUnicode_EncodeCodePage()`
- `PyUnicode_EncodeFSDefault()`
- `PyUnicode_EncodeLocale()`
- `PyUnicode_Equal()`
- `PyUnicode_EqualToUTF8()`
- `PyUnicode_EqualToUTF8AndSize()`
- `PyUnicode_FSConverter()`
- `PyUnicode_FSDecoder()`
- `PyUnicode_Find()`
- `PyUnicode_FindChar()`
- `PyUnicode_Format()`
- `PyUnicode_FromEncodedObject()`
- `PyUnicode_FromFormat()`
- `PyUnicode_FromFormatV()`
- `PyUnicode_FromObject()`
- `PyUnicode_FromOrdinal()`
- `PyUnicode_FromString()`
- `PyUnicode_FromStringAndSize()`
- `PyUnicode_FromWideChar()`
- `PyUnicode_GetDefaultEncoding()`
- `PyUnicode_GetLength()`
- `PyUnicode_InternFromString()`
- `PyUnicode_InternInPlace()`
- `PyUnicode_IsIdentifier()`
- `PyUnicode_Join()`
- `PyUnicode_Partition()`
- `PyUnicode_RPartition()`
- `PyUnicode_RSplit()`
- `PyUnicode_ReadChar()`
- `PyUnicode_Replace()`
- `PyUnicode_Resize()`

- PyUnicode_RichCompare()
- PyUnicode_Split()
- PyUnicode_Splitlines()
- PyUnicode_Substring()
- PyUnicode_Tailmatch()
- PyUnicode_Translate()
- PyUnicode_Type
- PyUnicode_WriteChar()
- PyVarObject
- PyVarObject.ob_base
- PyVarObject.ob_size
- PyVectorcall_Call()
- PyVectorcall_NARGS()
- PyWeakReference
- PyWeakref_GetObject()
- PyWeakref_GetRef()
- PyWeakref_NewProxy()
- PyWeakref_NewRef()
- PyWrapperDescr_Type
- PyWrapper_New()
- PyZip_Type
- Py_ASNATIVEBYTES_ALLOW_INDEX
- Py_ASNATIVEBYTES_BIG_ENDIAN
- Py_ASNATIVEBYTES_DEFAULTS
- Py_ASNATIVEBYTES_LITTLE_ENDIAN
- Py_ASNATIVEBYTES_NATIVE_ENDIAN
- Py_ASNATIVEBYTES_REJECT_NEGATIVE
- Py_ASNATIVEBYTES_UNSIGNED_BUFFER
- Py_AUDIT_READ
- Py_AddPendingCall()
- Py_AtExit()
- Py_BEGIN_ALLOW_THREADS
- Py_BLOCK_THREADS
- Py_BuildValue()
- Py_BytesMain()
- Py_CompileString()

- Py_DecRef()
- Py_DecodeLocale()
- Py_END_ALLOW_THREADS
- Py_EncodeLocale()
- Py_EndInterpreter()
- Py_EnterRecursiveCall()
- Py_Exit()
- Py_FatalError()
- Py_FileSystemDefaultEncodeErrors
- Py_FileSystemDefaultEncoding
- Py_Finalize()
- Py_FinalizeEx()
- Py_GenericAlias()
- Py_GenericAliasType
- Py_GetBuildInfo()
- Py_GetCompiler()
- Py_GetConstant()
- Py_GetConstantBorrowed()
- Py_GetCopyright()
- Py_GetExecPrefix()
- Py_GetPath()
- Py_GetPlatform()
- Py_GetPrefix()
- Py_GetProgramFullPath()
- Py_GetProgramName()
- Py_GetPythonHome()
- Py_GetRecursionLimit()
- Py_GetVersion()
- Py_HasFileSystemDefaultEncoding
- Py_IncRef()
- Py_Initialize()
- Py_InitializeEx()
- Py_Is()
- Py_IsFalse()
- Py_IsFinalizing()
- Py_IsInitialized()

- Py_IsNone()
- Py_IsTrue()
- Py_LeaveRecursiveCall()
- Py_Main()
- Py_MakePendingCalls()
- Py_NewInterpreter()
- Py_NewRef()
- Py_PACK_FULL_VERSION()
- Py_PACK_VERSION()
- Py_READONLY
- Py_REFCNT()
- Py_RELATIVE_OFFSET
- Py_ReprEnter()
- Py_ReprLeave()
- Py_SetProgramName()
- Py_SetPythonHome()
- Py_SetRecursionLimit()
- Py_TPFLAGS_BASETYPE
- Py_TPFLAGS_DEFAULT
- Py_TPFLAGS_HAVE_GC
- Py_TPFLAGS_HAVE_VECTORCALL
- Py_TPFLAGS_ITEMS_AT_END
- Py_TPFLAGS_METHOD_DESCRIPTOR
- Py_TP_USE_SPEC
- Py_TYPE()
- Py_T_BOOL
- Py_T_BYTE
- Py_T_CHAR
- Py_T_DOUBLE
- Py_T_FLOAT
- Py_T_INT
- Py_T_LONG
- Py_T_LONGLONG
- Py_T_OBJECT_EX
- Py_T_PYSSIZET
- Py_T_SHORT

- Py_T_STRING
- Py_T_STRING_INPLACE
- Py_T_UBYTE
- Py_T_UINT
- Py_T_ULONG
- Py_T_ULONGLONG
- Py_T_USHORT
- Py_UCS4
- Py_UNBLOCK_THREADS
- Py_UTF8Mode
- Py_VaBuildValue()
- Py_Version
- Py_XNewRef()
- Py_am_aiter
- Py_am_anext
- Py_am_await
- Py_am_send
- Py_bf_getbuffer
- Py_bf_releasebuffer
- Py_buffer
- Py_intptr_t
- Py_mod_create
- Py_mod_exec
- Py_mod_gil
- Py_mod_multiple_interpreters
- Py_mp_ass_subscript
- Py_mp_length
- Py_mp_subscript
- Py_nb_absolute
- Py_nb_add
- Py_nb_and
- Py_nb_bool
- Py_nb_divmod
- Py_nb_float
- Py_nb_floor_divide
- Py_nb_index

- Py_nb_inplace_add
- Py_nb_inplace_and
- Py_nb_inplace_floor_divide
- Py_nb_inplace_lshift
- Py_nb_inplace_matrix_multiply
- Py_nb_inplace_multiply
- Py_nb_inplace_or
- Py_nb_inplace_power
- Py_nb_inplace_remainder
- Py_nb_inplace_rshift
- Py_nb_inplace_subtract
- Py_nb_inplace_true_divide
- Py_nb_inplace_xor
- Py_nb_int
- Py_nb_invert
- Py_nb_lshift
- Py_nb_matrix_multiply
- Py_nb_multiply
- Py_nb_negative
- Py_nb_or
- Py_nb_positive
- Py_nb_power
- Py_nb_remainder
- Py_nb_rshift
- Py_nb_subtract
- Py_nb_true_divide
- Py_nb_xor
- Py_sq_ass_item
- Py_sq_concat
- Py_sq_contains
- Py_sq_inplace_concat
- Py_sq_inplace_repeat
- Py_sq_item
- Py_sq_length
- Py_sq_repeat
- Py_ssize_t

- Py_tp_alloc
- Py_tp_base
- Py_tp_bases
- Py_tp_call
- Py_tp_clear
- Py_tp_dealloc
- Py_tp_del
- Py_tp_descr_get
- Py_tp_descr_set
- Py_tp_doc
- Py_tp_finalize
- Py_tp_free
- Py_tp_getattr
- Py_tp_getattro
- Py_tp_getset
- Py_tp_hash
- Py_tp_init
- Py_tp_is_gc
- Py_tp_iter
- Py_tp_iternext
- Py_tp_members
- Py_tp_methods
- Py_tp_new
- Py_tp_repr
- Py_tp_richcompare
- Py_tp_setattr
- Py_tp_setattro
- Py_tp_str
- Py_tp_token
- Py_tp_traverse
- Py_tp_vectorcall
- Py_uintptr_t
- allocfunc
- binaryfunc
- descrgetfunc
- descrsetfunc

- destructor
- getattrfunc
- getattrofunc
- getbufferproc
- getiterfunc
- getter
- hashfunc
- initproc
- inquiry
- iternextfunc
- lenfunc
- newfunc
- objobjargproc
- objobjproc
- releasebufferproc
- reprfunc
- richcmpfunc
- setattrfunc
- setattrofunc
- setter
- ssizeargfunc
- ssizeobjargproc
- ssizessizeargfunc
- ssizessizeobjargproc
- symtable
- ternaryfunc
- traverseproc
- unaryfunc
- vectorcallfunc
- visitproc

The Very High Level Layer

The functions in this chapter will let you execute Python source code given in a file or a buffer, but they will not let you interact in a more detailed way with the interpreter.

Several of these functions accept a start symbol from the grammar as a parameter. The available start symbols are `Py_eval_input`, `Py_file_input`, `Py_single_input`, and `Py_func_type_input`. These are described following the functions which accept them as parameters.

Note also that several of these functions take FILE* parameters. One particular issue which needs to be handled carefully is that the `FILE` structure for different C libraries can be different and incompatible. Under Windows (at least), it is possible for dynamically linked extensions to actually use different libraries, so care should be taken that FILE* parameters are only passed to these functions if it is certain that they were created by the same library that the Python runtime is using.

int PyRun_AnyFile(FILE *fp, const char *filename)

This is a simplified interface to `PyRun_AnyFileExFlags()` below, leaving *closeit* set to `0` and *flags* set to `NULL`.

int PyRun_AnyFileFlags(FILE *fp, const char *filename, PyCompilerFlags *flags)

This is a simplified interface to `PyRun_AnyFileExFlags()` below, leaving the *closeit* argument set to `0`.

int PyRun_AnyFileEx(FILE *fp, const char *filename, int closeit)

This is a simplified interface to `PyRun_AnyFileExFlags()` below, leaving the *flags* argument set to `NULL`.

int PyRun_AnyFileExFlags(FILE *fp, const char *filename, int closeit, PyCompilerFlags *flags)

If *fp* refers to a file associated with an interactive device (console or terminal input or Unix pseudo-terminal), return the value of `PyRun_InteractiveLoop()`, otherwise return the result of `PyRun_SimpleFile()`. *filename* is decoded from the filesystem encoding (`sys.getfilesystemencoding()`). If *filename* is `NULL`, this function uses `"???"` as the filename. If *closeit* is true, the file is closed before `PyRun_SimpleFileExFlags()` returns.

int PyRun_SimpleString(const char *command)

This is a simplified interface to `PyRun_SimpleStringFlags()` below, leaving the `PyCompilerFlags`* argument set to `NULL`.

int PyRun_SimpleStringFlags(const char *command, PyCompilerFlags *flags)

Executes the Python source code from *command* in the `__main__` module according to the *flags* argument. If `__main__` does not already exist, it is created. Returns `0` on success or `-1` if an exception was raised. If there was an error, there is no way to get the exception information. For the meaning of *flags*, see below.

Note that if an otherwise unhandled `SystemExit` is raised, this function will not return `-1`, but exit the process, as long as `PyConfig.inspect` is zero.

int PyRun_SimpleFile(FILE *fp, const char *filename)

This is a simplified interface to `PyRun_SimpleFileExFlags()` below, leaving *closeit* set to `0` and *flags* set to `NULL`.

int PyRun_SimpleFileEx(FILE *fp, const char *filename, int closeit)

This is a simplified interface to **`PyRun_SimpleFileExFlags()`** below, leaving *flags* set to `NULL`.

int PyRun_SimpleFileExFlags(FILE *fp, const char *filename, int closeit, PyCompilerFlags *flags)

Similar to **`PyRun_SimpleStringFlags()`**, but the Python source code is read from *fp* instead of an in-memory string. *filename* should be the name of the file, it is decoded from filesystem encoding and error handler. If *closeit* is true, the file is closed before `PyRun_SimpleFileExFlags()` returns.

Note

On Windows, *fp* should be opened as binary mode (e.g. `fopen(filename, "rb")`). Otherwise, Python may not handle script file with LF line ending correctly.

int PyRun_InteractiveOneObject(FILE *fp, PyObject *filename, PyCompilerFlags *flags)

Read and execute a single statement from a file associated with an interactive device according to the *flags* argument. The user will be prompted using `sys.ps1` and `sys.ps2`. *filename* must be a Python **`str`** object.

Returns `0` when the input was executed successfully, `-1` if there was an exception, or an error code from the `errcode.h` include file distributed as part of Python if there was a parse error. (Note that `errcode.h` is not included by `Python.h`, so must be included specifically if needed.)

int PyRun_InteractiveOne(FILE *fp, const char *filename)

This is a simplified interface to **`PyRun_InteractiveOneFlags()`** below, leaving *flags* set to `NULL`.

int PyRun_InteractiveOneFlags(FILE *fp, const char *filename, PyCompilerFlags *flags)

Similar to **`PyRun_InteractiveOneObject()`**, but *filename* is a const char*, which is decoded from the filesystem encoding and error handler.

int PyRun_InteractiveLoop(FILE *fp, const char *filename)

This is a simplified interface to **`PyRun_InteractiveLoopFlags()`** below, leaving *flags* set to `NULL`.

int PyRun_InteractiveLoopFlags(FILE *fp, const char *filename, PyCompilerFlags *flags)

Read and execute statements from a file associated with an interactive device until EOF is reached. The user will be prompted using `sys.ps1` and `sys.ps2`. *filename* is decoded from the filesystem encoding and error handler. Returns `0` at EOF or a negative number upon failure.

int (*PyOS_InputHook)(void)

Part of the Stable ABI.

Can be set to point to a function with the prototype `int func(void)`. The function will be called when Python’s interpreter prompt is about to become idle and wait for user input from the terminal. The return value is ignored. Overriding this hook can be used to integrate the interpreter’s prompt with other event loops, as done in `Modules/_tkinter.c` in the Python source code.

Changed in version 3.12: This function is only called from the main interpreter.

char *(*PyOS_ReadlineFunctionPointer)(FILE*, FILE*, const char*)

Can be set to point to a function with the prototype `char *func(FILE *stdin, FILE *stdout, char *prompt)`, overriding the default function used to read a single line of input at the interpreter's prompt. The function is expected to output the string *prompt* if it's not `NULL`, and then read a line of input from the provided standard input file, returning the resulting string. For example, The **`readline`** module sets this hook to provide line-editing and tab-completion features.

The result must be a string allocated by **`PyMem_RawMalloc()`** or **`PyMem_RawRealloc()`**, or `NULL` if an error occurred.

Changed in version 3.4: The result must be allocated by **`PyMem_RawMalloc()`** or **`PyMem_RawRealloc()`**, instead of being allocated by **`PyMem_Malloc()`** or **`PyMem_Realloc()`**.

Changed in version 3.12: This function is only called from the main interpreter.

PyObject *PyRun_String(const char *str, int start, PyObject *globals, PyObject *locals)

Return value: New reference.
This is a simplified interface to **`PyRun_StringFlags()`** below, leaving *flags* set to `NULL`.

PyObject *PyRun_StringFlags(const char *str, int start, PyObject *globals, PyObject *locals, PyCompilerFlags *flags)

Return value: New reference.
Execute Python source code from *str* in the context specified by the objects *globals* and *locals* with the compiler flags specified by *flags*. *globals* must be a dictionary; *locals* can be any object that implements the

mapping protocol. The parameter *start* specifies the start symbol and must one of the available start symbols.

Returns the result of executing the code as a Python object, or `NULL` if an exception was raised.

PyObject *PyRun_File(FILE *fp, const char *filename, int start, PyObject *globals, PyObject *locals)

Return value: New reference.
This is a simplified interface to **`PyRun_FileExFlags()`** below, leaving *closeit* set to `0` and *flags* set to `NULL`.

PyObject *PyRun_FileEx(FILE *fp, const char *filename, int start, PyObject *globals, PyObject *locals, int closeit)

Return value: New reference.
This is a simplified interface to **`PyRun_FileExFlags()`** below, leaving *flags* set to `NULL`.

PyObject *PyRun_FileFlags(FILE *fp, const char *filename, int start, PyObject *globals, PyObject *locals, PyCompilerFlags *flags)

Return value: New reference.
This is a simplified interface to **`PyRun_FileExFlags()`** below, leaving *closeit* set to `0`.

PyObject *PyRun_FileExFlags(FILE *fp, const char *filename, int start, PyObject *globals, PyObject *locals, int closeit, PyCompilerFlags *flags)

Return value: New reference.
Similar to **`PyRun_StringFlags()`**, but the Python source code is read from *fp* instead of an in-memory string. *filename* should be the name of the file, it is decoded from the filesystem encoding and error handler. If *closeit* is true, the file is closed before **`PyRun_FileExFlags()`** returns.

PyObject *Py_CompileString(const char *str, const char *filename, int start)

Return value: New reference. Part of the Stable ABI.
This is a simplified interface to **`Py_CompileStringFlags()`** below, leaving *flags* set to `NULL`.

PyObject *Py_CompileStringFlags(const char *str, const char *filename, int start, PyCompilerFlags *flags)

Return value: New reference.
This is a simplified interface to **`Py_CompileStringExFlags()`** below, with *optimize* set to `-1`.

PyObject *Py_CompileStringObject(const char *str, PyObject *filename, int start, PyCompilerFlags *flags, int optimize)

Return value: New reference.
Parse and compile the Python source code in *str*, returning the resulting code object. The start symbol is given by *start*; this can be used to constrain the code which can be compiled and should be available start symbols. The filename specified by *filename* is used to construct the code object and may appear in tracebacks or **`SyntaxError`** exception messages. This returns `NULL` if the code cannot be parsed or compiled.

The integer *optimize* specifies the optimization level of the compiler; a value of `-1` selects the optimization level of the interpreter as given by **`-O`** options. Explicit levels are `0` (no optimization; `__debug__` is true), `1` (asserts are removed, `__debug__` is false) or `2` (docstrings are removed too).

Added in version 3.4.

PyObject *Py_CompileStringExFlags(const char *str, const char *filename, int start, PyCompilerFlags *flags, int optimize)

Return value: New reference.
Like `Py_CompileStringObject()`, but *filename* is a byte string decoded from the filesystem encoding and error handler.

Added in version 3.2.

PyObject *PyEval_EvalCode(PyObject *co, PyObject *globals, PyObject *locals)

Return value: New reference. Part of the Stable ABI.
This is a simplified interface to `PyEval_EvalCodeEx()`, with just the code object, and global and local variables. The other arguments are set to `NULL`.

PyObject *PyEval_EvalCodeEx(PyObject *co, PyObject *globals, PyObject *locals, PyObject *const *args, int argcount, PyObject *const *kws, int kwcount, PyObject *const *defs, int defcount, PyObject *kwdefs, PyObject *closure)

Return value: New reference. Part of the Stable ABI.
Evaluate a precompiled code object, given a particular environment for its evaluation. This environment consists of a dictionary of global variables, a mapping object of local variables, arrays of arguments, keywords and defaults, a dictionary of default values for keyword-only arguments and a closure tuple of cells.

PyObject *PyEval_EvalFrame(PyFrameObject *f)

Return value: New reference. Part of the Stable ABI.
Evaluate an execution frame. This is a simplified interface to `PyEval_EvalFrameEx()`, for backward compatibility.

PyObject *PyEval_EvalFrameEx(PyFrameObject *f, int throwflag)

Return value: New reference. Part of the Stable ABI.
This is the main, unvarnished function of Python interpretation. The code object associated with the execution frame *f* is executed, interpreting bytecode and executing calls as needed. The additional *throwflag* parameter can mostly be ignored - if true, then it causes an exception to immediately be thrown; this is used for the **`throw()`** methods of generator objects.

Changed in version 3.4: This function now includes a debug assertion to help ensure that it does not silently discard an active exception.

int PyEval_MergeCompilerFlags(PyCompilerFlags *cf)

This function changes the flags of the current evaluation frame, and returns true on success, false on failure.

struct PyCompilerFlags

This is the structure used to hold compiler flags. In cases where code is only being compiled, it is passed as `int flags`, and in cases where code is being executed, it is passed as `PyCompilerFlags *flags`. In this case, `from __future__ import` can modify *flags*.

Whenever `PyCompilerFlags *flags` is `NULL`, **`cf_flags`** is treated as equal to `0`, and any modification due to `from __future__ import` is discarded.

int cf_flags

Compiler flags.

int cf_feature_version

cf_feature_version is the minor Python version. It should be initialized to `PY_MINOR_VERSION`.

The field is ignored by default, it is used if and only if `PyCF_ONLY_AST` flag is set in **`cf_flags`**.

Changed in version 3.8: Added *cf_feature_version* field.

The available compiler flags are accessible as macros:

PyCF_ALLOW_TOP_LEVEL_AWAIT
PyCF_ONLY_AST
PyCF_OPTIMIZED_AST
PyCF_TYPE_COMMENTS

See compiler flags in documentation of the **`ast`** Python module, which exports these constants under the same names.

The "`PyCF`" flags above can be combined with "`CO_FUTURE`" flags such as **`CO_FUTURE_ANNOTATIONS`** to enable features normally selectable using future statements. See Code Object Flags for a complete list.

Available start symbols

int Py_eval_input

The start symbol from the Python grammar for isolated expressions; for use with **`Py_CompileString()`**.

int Py_file_input

The start symbol from the Python grammar for sequences of statements as read from a file or other source; for use with **`Py_CompileString()`**. This is the symbol to use when compiling arbitrarily long Python source code.

int Py_single_input

The start symbol from the Python grammar for a single statement; for use with **`Py_CompileString()`**. This is the symbol used for the interactive interpreter loop.

int Py_func_type_input

The start symbol from the Python grammar for a function type; for use with `Py_CompileString()`. This is used to parse “signature type comments” from **PEP 484** [https://peps.python.org/pep-0484/].

This requires the `PyCF_ONLY_AST` flag to be set.

See also

- `ast.FunctionType`
- **PEP 484** [https://peps.python.org/pep-0484/]

Added in version 3.8.

Stack Effects

See also

`dis.stack_effect()`

PY_INVALID_STACK_EFFECT

Sentinel value representing an invalid stack effect.

This is currently equivalent to `INT_MAX`.

Added in version 3.8.

int PyCompile_OpcodeStackEffect(int opcode, int oparg)

Compute the stack effect of *opcode* with argument *oparg*.

On success, this function returns the stack effect; on failure, this returns `PY_INVALID_STACK_EFFECT`.

Added in version 3.4.

int PyCompile_OpcodeStackEffectWithJump(int opcode, int oparg, int jump)

Similar to `PyCompile_OpcodeStackEffect()`, but don't include the stack effect of jumping if *jump* is zero.

If *jump* is `0`, this will not include the stack effect of jumping, but if *jump* is `1` or `-1`, this will include it.

On success, this function returns the stack effect; on failure, this returns `PY_INVALID_STACK_EFFECT`.

Added in version 3.8.

Reference Counting

The functions and macros in this section are used for managing reference counts of Python objects.

Py_ssize_t Py_REFCNT(PyObject *o)

Part of the Stable ABI since version 3.14.
Get the reference count of the Python object *o*.

Note that the returned value may not actually reflect how many references to the object are actually held. For example, some objects are immortal and have a very high refcount that does not reflect the actual number of references. Consequently, do not rely on the returned value to be accurate, other than a value of 0 or 1.

Use the `Py_SET_REFCNT()` function to set an object reference count.

Note

On free threaded builds of Python, returning 1 isn’t sufficient to determine if it’s safe to treat *o* as having no access by other threads. Use `PyUnstable_Object_IsUniquelyReferenced()` for that instead.

See also the function `PyUnstable_Object_IsUniqueReferencedTemporary()`.

Changed in version 3.10: `Py_REFCNT()` is changed to the inline static function.

Changed in version 3.11: The parameter type is no longer const PyObject*.

void Py_SET_REFCNT(PyObject *o, Py_ssize_t refcnt)

Set the object *o* reference counter to *refcnt*.

On Python build with Free Threading, if *refcnt* is larger than `UINT32_MAX`, the object is made immortal.

This function has no effect on immortal objects.

Added in version 3.9.

Changed in version 3.12: Immortal objects are not modified.

void Py_INCREF(PyObject *o)

Indicate taking a new strong reference to object *o*, indicating it is in use and should not be destroyed.

This function has no effect on immortal objects.

This function is usually used to convert a borrowed reference to a strong reference in-place. The **`Py_NewRef()`** function can be used to create a new strong reference.

When done using the object, release is by calling **`Py_DECREF()`**.

The object must not be `NULL`; if you aren't sure that it isn't `NULL`, use **`Py_XINCREF()`**.

Do not expect this function to actually modify *o* in any way. For at least **some objects** [https://peps.python.org/pep-0683/], this function has no effect.

Changed in version 3.12: Immortal objects are not modified.

void Py_XINCREF(PyObject *o)

Similar to **`Py_INCREF()`**, but the object *o* can be `NULL`, in which case this has no effect.

See also **`Py_XNewRef()`**.

PyObject *Py_NewRef(PyObject *o)

Part of the Stable ABI since version 3.10.
Create a new strong reference to an object: call **`Py_INCREF()`** on *o* and return the object *o*.

When the strong reference is no longer needed, **`Py_DECREF()`** should be called on it to release the reference.

The object *o* must not be `NULL`; use **`Py_XNewRef()`** if *o* can be `NULL`.

For example:

```
Py_INCREF(obj);
self->attr = obj;
```

can be written as:

```
self->attr = Py_NewRef(obj);
```

See also **`Py_INCREF()`**.

Added in version 3.10.

PyObject *Py_XNewRef(PyObject *o)

Part of the Stable ABI since version 3.10.
Similar to **`Py_NewRef()`**, but the object *o* can be NULL.

If the object *o* is `NULL`, the function just returns `NULL`.

Added in version 3.10.

void Py_DECREF(PyObject *o)

Release a strong reference to object *o*, indicating the reference is no longer used.

This function has no effect on immortal objects.

Once the last strong reference is released (i.e. the object's reference count reaches 0), the object's type's deallocation function (which must not be `NULL`) is invoked.

This function is usually used to delete a strong reference before exiting its scope.

The object must not be `NULL`; if you aren't sure that it isn't `NULL`, use **`Py_XDECREF()`**.

Do not expect this function to actually modify *o* in any way. For at least **some objects** [https://peps.python.org/pep-0683/], this function has no effect.

Warning

The deallocation function can cause arbitrary Python code to be invoked (e.g. when a class instance with a **`__del__()`** method is deallocated). While exceptions in such code are not propagated, the executed code has free access to all Python global variables. This means that any object that is reachable from a global variable should be in a consistent state before **`Py_DECREF()`** is invoked. For example, code to delete an object from a list should copy a reference to the deleted object in a temporary variable, update the list data structure, and then call **`Py_DECREF()`** for the temporary variable.

Changed in version 3.12: Immortal objects are not modified.

void Py_XDECREF(PyObject *o)

Similar to **`Py_DECREF()`**, but the object *o* can be `NULL`, in which case this has no effect. The same warning from **`Py_DECREF()`** applies here as well.

void Py_CLEAR(PyObject *o)

Release a strong reference for object *o*. The object may be `NULL`, in which case the macro has no effect; otherwise the effect is the same as for **`Py_DECREF()`**, except that the argument is also set to `NULL`. The warning for **`Py_DECREF()`** does not apply with respect to the object passed because the macro carefully uses a temporary variable and sets the argument to `NULL` before releasing the reference.

It is a good idea to use this macro whenever releasing a reference to an object that might be traversed during garbage collection.

Changed in version 3.12: The macro argument is now only evaluated once. If the argument has side effects, these are no longer duplicated.

void Py_IncRef(PyObject *o)

Part of the Stable ABI.
Indicate taking a new strong reference to object *o*. A function version of **`Py_XINCREF()`**. It can be used for runtime dynamic embedding of Python.

void Py_DecRef(PyObject *o)

Part of the Stable ABI.
Release a strong reference to object *o*. A function version of **`Py_XDECREF()`**. It can be used for runtime dynamic embedding of Python.

Py_SETREF(dst, src)

Macro safely releasing a strong reference to object *dst* and setting *dst* to *src*.

As in case of `Py_CLEAR()`, “the obvious” code can be deadly:

```
Py_DECREF(dst);
dst = src;
```

The safe way is:

```
Py_SETREF(dst, src);
```

That arranges to set *dst* to *src* *before* releasing the reference to the old value of *dst*, so that any code triggered as a side-effect of *dst* getting torn down no longer believes *dst* points to a valid object.

Added in version 3.6.

Changed in version 3.12: The macro arguments are now only evaluated once. If an argument has side effects, these are no longer duplicated.

Py_XSETREF(dst, src)

Variant of `Py_SETREF` macro that uses `Py_XDECREF()` instead of `Py_DECREF()`.

Added in version 3.6.

Changed in version 3.12: The macro arguments are now only evaluated once. If an argument has side effects, these are no longer duplicated.

Exception Handling

The functions described in this chapter will let you handle and raise Python exceptions. It is important to understand some of the basics of Python exception handling. It works somewhat like the POSIX **`errno`** variable: there is a global indicator (per thread) of the last error that occurred. Most C API functions don’t clear this on success, but will set it to indicate the cause of the error on failure. Most C API functions also return an error indicator, usually `NULL` if they are supposed to return a pointer, or `-1` if they return an integer (exception: the `PyArg_*` functions return `1` for success and `0` for failure).

Concretely, the error indicator consists of three object pointers: the exception’s type, the exception’s value, and the traceback object. Any of those pointers can be `NULL` if non-set (although some combinations are forbidden, for example you can’t have a non-`NULL` traceback if the exception type is `NULL`).

When a function must fail because some function it called failed, it generally doesn’t set the error indicator; the function it called already set it. It is responsible for either handling the error and clearing the exception or returning after cleaning up any resources it holds (such as object references or memory allocations); it should *not* continue normally if it is not prepared to handle the error. If returning due to an error, it is important to indicate to the caller that an error has been set. If the error is not handled or carefully propagated, additional calls into the Python/C API may not behave as intended and may fail in mysterious ways.

Note

The error indicator is **not** the result of `sys.exc_info()`. The former corresponds to an exception that is not yet caught (and is therefore still propagating), while the latter returns an exception after it is caught (and has therefore stopped propagating).

Printing and clearing

void PyErr_Clear()

Part of the Stable ABI.
Clear the error indicator. If the error indicator is not set, there is no effect.

void PyErr_PrintEx(int set_sys_last_vars)

Part of the Stable ABI.
Print a standard traceback to `sys.stderr` and clear the error indicator. **Unless** the error is a `SystemExit`, in that case no traceback is printed and the Python process will exit with the error code specified by the `SystemExit` instance.

Call this function **only** when the error indicator is set. Otherwise it will cause a fatal error!

If *set_sys_last_vars* is nonzero, the variable `sys.last_exc` is set to the printed exception. For backwards compatibility, the deprecated variables `sys.last_type`, `sys.last_value` and `sys.last_traceback` are also set to the type, value and traceback of this exception, respectively.

Changed in version 3.12: The setting of `sys.last_exc` was added.

void PyErr_Print()

Part of the Stable ABI.
Alias for `PyErr_PrintEx(1)`.

void PyErr_WriteUnraisable(PyObject *obj)

Part of the Stable ABI.

Call `sys.unraisablehook()` using the current exception and *obj* argument.

This utility function prints a warning message to `sys.stderr` when an exception has been set but it is impossible for the interpreter to actually raise the exception. It is used, for example, when an exception occurs in an `__del__()` method.

The function is called with a single argument *obj* that identifies the context in which the unraisable exception occurred. If possible, the repr of *obj* will be printed in the warning message. If *obj* is `NULL`, only the traceback is printed.

An exception must be set when calling this function.

Changed in version 3.4: Print a traceback. Print only traceback if *obj* is `NULL`.

Changed in version 3.8: Use `sys.unraisablehook()`.

void PyErr_FormatUnraisable(const char *format, ...)

Similar to `PyErr_WriteUnraisable()`, but the *format* and subsequent parameters help format the warning message; they have the same meaning and values as in `PyUnicode_FromFormat()`. `PyErr_WriteUnraisable(obj)` is roughly equivalent to `PyErr_FormatUnraisable("Exception ignored in: %R", obj)`. If *format* is `NULL`, only the traceback is printed.

Added in version 3.13.

void PyErr_DisplayException(PyObject *exc)

Part of the Stable ABI since version 3.12.

Print the standard traceback display of `exc` to `sys.stderr`, including chained exceptions and notes.

Added in version 3.12.

Raising exceptions

These functions help you set the current thread's error indicator. For convenience, some of these functions will always return a `NULL` pointer for use in a `return` statement.

void PyErr_SetString(PyObject *type, const char *message)

Part of the Stable ABI.
This is the most common way to set the error indicator. The first argument specifies the exception type; it is normally one of the standard exceptions, e.g. **`PyExc_RuntimeError`**. You need not create a new strong reference to it (e.g. with **`Py_INCREF()`**). The second argument is an error message; it is decoded from `'utf-8'`.

void PyErr_SetObject(PyObject *type, PyObject *value)

Part of the Stable ABI.
This function is similar to **`PyErr_SetString()`** but lets you specify an arbitrary Python object for the "value" of the exception.

PyObject *PyErr_Format(PyObject *exception, const char *format, ...)

Return value: Always NULL. Part of the Stable ABI.
This function sets the error indicator and returns `NULL`. *exception* should be a Python exception class. The *format* and subsequent parameters help format the error message; they have the same meaning and values as in **`PyUnicode_FromFormat()`**. *format* is an ASCII-encoded string.

PyObject *PyErr_FormatV(PyObject *exception, const char *format, va_list vargs)

Return value: Always NULL. Part of the Stable ABI since version 3.5.
Same as **`PyErr_Format()`**, but taking a **`va_list`** argument rather than a variable number of arguments.

Added in version 3.5.

void PyErr_SetNone(PyObject *type)

Part of the Stable ABI.
This is a shorthand for `PyErr_SetObject(type, Py_None)`.

int PyErr_BadArgument()

Part of the Stable ABI.
This is a shorthand for `PyErr_SetString(PyExc_TypeError, message)`, where *message* indicates that a built-in operation was invoked with an illegal argument. It is mostly for internal use.

PyObject *PyErr_NoMemory()

Return value: Always NULL. Part of the Stable ABI.
This is a shorthand for `PyErr_SetNone(PyExc_MemoryError)`; it returns `NULL` so an object allocation function can write `return PyErr_NoMemory();` when it runs out of memory.

PyObject *PyErr_SetFromErrno(PyObject *type)

Return value: Always NULL. Part of the Stable ABI.
This is a convenience function to raise an exception when a C library function has returned an error and set the C variable **`errno`**. It constructs a tuple object whose first item is the integer **`errno`** value and whose second item is the corresponding error message (gotten from **`strerror()`**), and then calls `PyErr_SetObject(type, object)`. On Unix, when the **`errno`** value is **`EINTR`**, indicating an interrupted system call, this calls **`PyErr_CheckSignals()`**, and if that set the error

indicator, leaves it set to that. The function always returns `NULL`, so a wrapper function around a system call can write `return PyErr_SetFromErrno(type);` when the system call returns an error.

PyObject *PyErr_SetFromErrnoWithFilenameObject(PyObject *type, PyObject *filenameObject)

Return value: Always NULL. Part of the Stable ABI.
Similar to **`PyErr_SetFromErrno()`**, with the additional behavior that if *filenameObject* is not `NULL`, it is passed to the constructor of *type* as a third parameter. In the case of **`OSError`** exception, this is used to define the **`filename`** attribute of the exception instance.

PyObject *PyErr_SetFromErrnoWithFilenameObjects(PyObject *type, PyObject *filenameObject, PyObject *filenameObject2)

Return value: Always NULL. Part of the Stable ABI since version 3.7.
Similar to **`PyErr_SetFromErrnoWithFilenameObject()`**, but takes a second filename object, for raising errors when a function that takes two filenames fails.

Added in version 3.4.

PyObject *PyErr_SetFromErrnoWithFilename(PyObject *type, const char *filename)

Return value: Always NULL. Part of the Stable ABI.
Similar to **`PyErr_SetFromErrnoWithFilenameObject()`**, but the filename is given as a C string. *filename* is decoded from the filesystem encoding and error handler.

PyObject *PyErr_SetFromWindowsErr(int ierr)

Return value: Always NULL. Part of the Stable ABI on Windows since version 3.7.
This is a convenience function to raise **`OSError`**. If called with *ierr* of `0`, the error code returned by a call to

`GetLastError()` is used instead. It calls the Win32 function **`FormatMessage()`** to retrieve the Windows description of error code given by *ierr* or **`GetLastError()`**, then it constructs a **`OSError`** object with the **`winerror`** attribute set to the error code, the **`strerror`** attribute set to the corresponding error message (gotten from **`FormatMessage()`**), and then calls `PyErr_SetObject(PyExc_OSError, object)`. This function always returns `NULL`.

Availability: Windows.

PyObject *PyErr_SetExcFromWindowsErr(PyObject *type, int ierr)

Return value: Always NULL. Part of the Stable ABI on Windows since version 3.7.
Similar to **`PyErr_SetFromWindowsErr()`**, with an additional parameter specifying the exception type to be raised.

Availability: Windows.

PyObject *PyErr_SetFromWindowsErrWithFilename(int ierr, const char *filename)

Return value: Always NULL. Part of the Stable ABI on Windows since version 3.7.
Similar to **`PyErr_SetFromWindowsErr()`**, with the additional behavior that if *filename* is not `NULL`, it is decoded from the filesystem encoding (**`os.fsdecode()`**) and passed to the constructor of **`OSError`** as a third parameter to be used to define the **`filename`** attribute of the exception instance.

Availability: Windows.

PyObject *PyErr_SetExcFromWindowsErrWithFilenameObject(PyObject *type, int ierr, PyObject *filename)

Return value: Always NULL. Part of the Stable ABI on Windows since version 3.7.
Similar to `PyErr_SetExcFromWindowsErr()`, with the additional behavior that if *filename* is not `NULL`, it is passed to the constructor of `OSError` as a third parameter to be used to define the **`filename`** attribute of the exception instance.

Availability: Windows.

PyObject *PyErr_SetExcFromWindowsErrWithFilenameObjects(PyObject *type, int ierr, PyObject *filename, PyObject *filename2)

Return value: Always NULL. Part of the Stable ABI on Windows since version 3.7.
Similar to `PyErr_SetExcFromWindowsErrWithFilenameObject()`, but accepts a second filename object.

Availability: Windows.

Added in version 3.4.

PyObject *PyErr_SetExcFromWindowsErrWithFilename(PyObject *type, int ierr, const char *filename)

Return value: Always NULL. Part of the Stable ABI on Windows since version 3.7.
Similar to `PyErr_SetFromWindowsErrWithFilename()`, with an additional parameter specifying the exception type to be raised.

Availability: Windows.

PyObject *PyErr_SetImportError(PyObject *msg, PyObject *name, PyObject *path)

Return value: Always NULL. Part of the Stable ABI since version 3.7.

This is a convenience function to raise `ImportError`. *msg* will be set as the exception's message string. *name* and *path*, both of which can be `NULL`, will be set as the `ImportError`'s respective `name` and `path` attributes.

Added in version 3.3.

PyObject *PyErr_SetImportErrorSubclass(PyObject *exception, PyObject *msg, PyObject *name, PyObject *path)

Return value: Always NULL. Part of the Stable ABI since version 3.6.

Much like `PyErr_SetImportError()` but this function allows for specifying a subclass of `ImportError` to raise.

Added in version 3.6.

void PyErr_SyntaxLocationObject(PyObject *filename, int lineno, int col_offset)

Set file, line, and offset information for the current exception. If the current exception is not a `SyntaxError`, then it sets additional attributes, which make the exception printing subsystem think the exception is a `SyntaxError`.

Added in version 3.4.

void PyErr_RangedSyntaxLocationObject(PyObject *filename, int lineno, int col_offset, int end_lineno, int end_col_offset)

Similar to `PyErr_SyntaxLocationObject()`, but also sets the *end_lineno* and *end_col_offset* information for the current exception.

Added in version 3.10.

void PyErr_SyntaxLocationEx(const char *filename, int lineno, int col_offset)

Part of the Stable ABI since version 3.7.
Like **`PyErr_SyntaxLocationObject()`**, but *filename* is a byte string decoded from the filesystem encoding and error handler.

Added in version 3.2.

void PyErr_SyntaxLocation(const char *filename, int lineno)

Part of the Stable ABI.
Like **`PyErr_SyntaxLocationEx()`**, but the *col_offset* parameter is omitted.

void PyErr_BadInternalCall()

Part of the Stable ABI.
This is a shorthand for `PyErr_SetString(PyExc_SystemError, message)`, where *message* indicates that an internal operation (e.g. a Python/C API function) was invoked with an illegal argument. It is mostly for internal use.

PyObject *PyErr_ProgramTextObject(PyObject *filename, int lineno)

Get the source line in *filename* at line *lineno*. *filename* should be a Python **`str`** object.

On success, this function returns a Python string object with the found line. On failure, this function returns `NULL` without an exception set.

PyObject *PyErr_ProgramText(const char *filename, int lineno)

Part of the Stable ABI.
Similar to **`PyErr_ProgramTextObject()`**, but *filename* is a const char*, which is decoded with the filesystem encoding and error handler, instead of a Python object reference.

Issuing warnings

Use these functions to issue warnings from C code. They mirror similar functions exported by the Python **`warnings`** module. They normally print a warning message to *sys.stderr*; however, it is also possible that the user has specified that warnings are to be turned into errors, and in that case they will raise an exception. It is also possible that the functions raise an exception because of a problem with the warning machinery. The return value is `0` if no exception is raised, or `-1` if an exception is raised. (It is not possible to determine whether a warning message is actually printed, nor what the reason is for the exception; this is intentional.) If an exception is raised, the caller should do its normal exception handling (for example, **`Py_DECREF()`** owned references and return an error value).

int PyErr_WarnEx(PyObject *category, const char *message, Py_ssize_t stack_level)

Part of the Stable ABI.

Issue a warning message. The *category* argument is a warning category (see below) or `NULL`; the *message* argument is a UTF-8 encoded string. *stack_level* is a positive number giving a number of stack frames; the warning will be issued from the currently executing line of code in that stack frame. A *stack_level* of 1 is the function calling **`PyErr_WarnEx()`**, 2 is the function above that, and so forth.

Warning categories must be subclasses of **`PyExc_Warning`**; **`PyExc_Warning`** is a subclass of **`PyExc_Exception`**; the default warning category is **`PyExc_RuntimeWarning`**. The standard Python warning categories are available as global variables whose names are enumerated at Warning types.

For information about warning control, see the documentation for the **`warnings`** module and the **`-W`** option in the command line documentation. There is no C API for warning control.

int PyErr_WarnExplicitObject(PyObject *category, PyObject *message, PyObject *filename, int lineno, PyObject *module, PyObject *registry)

Issue a warning message with explicit control over all warning attributes. This is a straightforward wrapper around the Python function **`warnings.warn_explicit()`**; see there for more information. The *module* and *registry* arguments may be set to `NULL` to get the default effect described there.

Added in version 3.4.

int PyErr_WarnExplicit(PyObject *category, const char *message, const char *filename, int lineno, const char *module, PyObject *registry)

Part of the Stable ABI.
Similar to **`PyErr_WarnExplicitObject()`** except that *message* and *module* are UTF-8 encoded strings, and *filename* is decoded from the filesystem encoding and error handler.

int PyErr_WarnFormat(PyObject *category, Py_ssize_t stack_level, const char *format, ...)

Part of the Stable ABI.
Function similar to **`PyErr_WarnEx()`**, but use **`PyUnicode_FromFormat()`** to format the warning message. *format* is an ASCII-encoded string.

Added in version 3.2.

int PyErr_WarnExplicitFormat(PyObject *category, const char *filename, int lineno, const char *module, PyObject *registry, const char *format, ...)

Similar to `PyErr_WarnExplicit()`, but uses `PyUnicode_FromFormat()` to format the warning message. *format* is an ASCII-encoded string.

Added in version 3.2.

int PyErr_ResourceWarning(PyObject *source, Py_ssize_t stack_level, const char *format, ...)

Part of the Stable ABI since version 3.6.
Function similar to `PyErr_WarnFormat()`, but *category* is `ResourceWarning` and it passes *source* to **`warnings.WarningMessage`**.

Added in version 3.6.

Querying the error indicator

PyObject *PyErr_Occurred()

Return value: Borrowed reference. Part of the Stable ABI.
Test whether the error indicator is set. If set, return the exception *type* (the first argument to the last call to one of the `PyErr_Set*` functions or to `PyErr_Restore()`). If not set, return `NULL`. You do not own a reference to the return value, so you do not need to `Py_DECREF()` it.

The caller must have an attached thread state.

Note

Do not compare the return value to a specific exception; use `PyErr_ExceptionMatches()` instead, shown below. (The comparison could easily fail since the exception may be an instance instead of a class, in the case of a class exception, or it may be a subclass of the expected exception.)

int PyErr_ExceptionMatches(PyObject *exc)

Part of the Stable ABI.

Equivalent to `PyErr_GivenExceptionMatches(PyErr_Occurred(), exc)`. This should only be called when an exception is actually set; a memory access violation will occur if no exception has been raised.

int PyErr_GivenExceptionMatches(PyObject *given, PyObject *exc)

Part of the Stable ABI.

Return true if the *given* exception matches the exception type in *exc*. If *exc* is a class object, this also returns true when *given* is an instance of a subclass. If *exc* is a tuple, all exception types in the tuple (and recursively in subtuples) are searched for a match.

PyObject *PyErr_GetRaisedException(void)

Return value: New reference. Part of the Stable ABI since version 3.12.

Return the exception currently being raised, clearing the error indicator at the same time. Return `NULL` if the error indicator is not set.

This function is used by code that needs to catch exceptions, or code that needs to save and restore the error indicator temporarily.

For example:

```
{
   PyObject *exc = PyErr_GetRaisedException();

   /* ... code that might produce other errors ... */

   PyErr_SetRaisedException(exc);
}
```

See also

[`PyErr_GetHandledException()`], to save the exception currently being handled.

Added in version 3.12.

void PyErr_SetRaisedException(PyObject *exc)

Part of the Stable ABI since version 3.12.
Set *exc* as the exception currently being raised, clearing the existing exception if one is set.

Warning

This call steals a reference to *exc*, which must be a valid exception.

Added in version 3.12.

void PyErr_Fetch(PyObject **ptype, PyObject **pvalue, PyObject **ptraceback)

Part of the Stable ABI.
Deprecated since version 3.12: Use `PyErr_GetRaisedException()` instead.

Retrieve the error indicator into three variables whose addresses are passed. If the error indicator is not set, set all three variables to `NULL`. If it is set, it will be cleared and you own a reference to each object retrieved. The value and traceback object may be `NULL` even when the type object is not.

Note

This function is normally only used by legacy code that needs to catch exceptions or save and restore the error indicator temporarily.

For example:

```
{
   PyObject *type, *value, *traceback;
   PyErr_Fetch(&type, &value, &traceback);

   /* ... code that might produce other errors ... */

   PyErr_Restore(type, value, traceback);
}
```

void PyErr_Restore(PyObject *type, PyObject *value, PyObject *traceback)

Part of the Stable ABI.
Deprecated since version 3.12: Use `PyErr_SetRaisedException()` instead.

Set the error indicator from the three objects, *type*, *value*, and *traceback*, clearing the existing exception if one is set. If the objects are `NULL`, the error indicator is cleared. Do not pass a `NULL` type and non-`NULL` value or traceback. The exception type should be a class. Do not pass an invalid exception type or value. (Violating these rules will cause subtle problems later.) This call takes away a reference to each object: you must own a reference to each object before the call and after the call you no longer own these references. (If you don't understand this, don't use this function. I warned you.)

Note

This function is normally only used by legacy code that needs to save and restore the error indicator temporarily. Use `PyErr_Fetch()` to save the current error indicator.

void PyErr_NormalizeException(PyObject **exc, PyObject **val, PyObject **tb)

Part of the Stable ABI.

Deprecated since version 3.12: Use **`PyErr_GetRaisedException()`** instead, to avoid any possible de-normalization.

Under certain circumstances, the values returned by **`PyErr_Fetch()`** below can be “unnormalized”, meaning that `*exc` is a class object but `*val` is not an instance of the same class. This function can be used to instantiate the class in that case. If the values are already normalized, nothing happens. The delayed normalization is implemented to improve performance.

Note

This function *does not* implicitly set the **`__traceback__`** attribute on the exception value. If setting the traceback appropriately is desired, the following additional snippet is needed:

```
if (tb != NULL) {
  PyException_SetTraceback(val, tb);
}
```

PyObject *PyErr_GetHandledException(void)

Part of the Stable ABI since version 3.11.

Retrieve the active exception instance, as would be returned by **`sys.exception()`**. This refers to an exception that was *already caught*, not to an exception that was freshly raised. Returns a new reference to the exception or `NULL`. Does not modify the interpreter’s exception state.

Note

This function is not normally used by code that wants to handle exceptions. Rather, it can be used when code needs to save and restore the exception state temporarily. Use **`PyErr_SetHandledException()`** to restore or clear the exception state.

Added in version 3.11.

void PyErr_SetHandledException(PyObject *exc)

Part of the Stable ABI since version 3.11.

Set the active exception, as known from `sys.exception()`. This refers to an exception that was *already caught*, not to an exception that was freshly raised. To clear the exception state, pass `NULL`.

Note

This function is not normally used by code that wants to handle exceptions. Rather, it can be used when code needs to save and restore the exception state temporarily. Use **`PyErr_GetHandledException()`** to get the exception state.

Added in version 3.11.

void PyErr_GetExcInfo(PyObject **ptype, PyObject **pvalue, PyObject **ptraceback)

Part of the Stable ABI since version 3.7.

Retrieve the old-style representation of the exception info, as known from **`sys.exc_info()`**. This refers to an exception that was *already caught*, not to an exception that was freshly raised. Returns new references for the three objects, any of which may be `NULL`. Does not modify the exception info state. This function is kept for backwards compatibility. Prefer using **`PyErr_GetHandledException()`**.

Note

This function is not normally used by code that wants to handle exceptions. Rather, it can be used when code needs to save and restore the exception state temporarily. Use **`PyErr_SetExcInfo()`** to restore or clear the exception state.

Added in version 3.3.

void PyErr_SetExcInfo(PyObject *type, PyObject *value, PyObject *traceback)

Part of the Stable ABI since version 3.7.

Set the exception info, as known from `sys.exc_info()`. This refers to an exception that was *already caught*, not to an exception that was freshly raised. This function steals the references of the arguments. To clear the exception state, pass `NULL` for all three arguments. This function is kept for backwards compatibility. Prefer using **`PyErr_SetHandledException()`**.

Note

This function is not normally used by code that wants to handle exceptions. Rather, it can be used when code needs to save and restore the exception state temporarily. Use **`PyErr_GetExcInfo()`** to read the exception state.

Added in version 3.3.

Changed in version 3.11: The `type` and `traceback` arguments are no longer used and can be NULL. The interpreter now derives them from the exception instance (the `value` argument). The function still steals references of all three arguments.

Signal Handling

int PyErr_CheckSignals()

Part of the Stable ABI.

This function interacts with Python’s signal handling.

If the function is called from the main thread and under the main Python interpreter, it checks whether a signal has been sent to the processes and if so, invokes the corresponding signal handler. If the **`signal`** module is supported, this can invoke a signal handler written in Python.

The function attempts to handle all pending signals, and then returns `0`. However, if a Python signal handler raises an exception, the error indicator is set and the function returns `-1` immediately (such that other pending signals may not have been handled yet: they will be on the next **`PyErr_CheckSignals()`** invocation).

If the function is called from a non-main thread, or under a non-main Python interpreter, it does nothing and returns `0`.

This function can be called by long-running C code that wants to be interruptible by user requests (such as by pressing Ctrl-C).

Note

The default Python signal handler for **`SIGINT`** raises the **`KeyboardInterrupt`** exception.

void PyErr_SetInterrupt()

Part of the Stable ABI.

Simulate the effect of a **`SIGINT`** signal arriving. This is equivalent to `PyErr_SetInterruptEx(SIGINT)`.

Note

This function is async-signal-safe. It can be called without an attached thread state and from a C signal handler.

int PyErr_SetInterruptEx(int signum)

Part of the Stable ABI since version 3.10.
Simulate the effect of a signal arriving. The next time `PyErr_CheckSignals()` is called, the Python signal handler for the given signal number will be called.

This function can be called by C code that sets up its own signal handling and wants Python signal handlers to be invoked as expected when an interruption is requested (for example when the user presses Ctrl-C to interrupt an operation).

If the given signal isn't handled by Python (it was set to `signal.SIG_DFL` or `signal.SIG_IGN`), it will be ignored.

If *signum* is outside of the allowed range of signal numbers, `-1` is returned. Otherwise, `0` is returned. The error indicator is never changed by this function.

Note

This function is async-signal-safe. It can be called without an attached thread state and from a C signal handler.

Added in version 3.10.

int PySignal_SetWakeupFd(int fd)

This utility function specifies a file descriptor to which the signal number is written as a single byte whenever a signal is received. *fd* must be non-blocking. It returns the previous such file descriptor.

The value `-1` disables the feature; this is the initial state. This is equivalent to **`signal.set_wakeup_fd()`** in Python, but without any error checking. *fd* should be a valid file descriptor. The function should only be called from the main thread.

Changed in version 3.5: On Windows, the function now also supports socket handles.

Exception Classes

PyObject *PyErr_NewException(const char *name, PyObject *base, PyObject *dict)

Return value: New reference. Part of the Stable ABI.
This utility function creates and returns a new exception class. The *name* argument must be the name of the new exception, a C string of the form `module.classname`. The *base* and *dict* arguments are normally `NULL`. This creates a class object derived from **`Exception`** (accessible in C as **`PyExc_Exception`**).

The **`__module__`** attribute of the new class is set to the first part (up to the last dot) of the *name* argument, and the class name is set to the last part (after the last dot). The *base* argument can be used to specify alternate base classes; it can either be only one class or a tuple of classes. The *dict* argument can be used to specify a dictionary of class variables and methods.

PyObject *PyErr_NewExceptionWithDoc(const char *name, const char *doc, PyObject *base, PyObject *dict)

Return value: New reference. Part of the Stable ABI.
Same as **`PyErr_NewException()`**, except that the new exception class can easily be given a docstring: If *doc* is non-`NULL`, it will be used as the docstring for the exception class.

Added in version 3.2.

int PyExceptionClass_Check(PyObject *ob)

Return non-zero if *ob* is an exception class, zero otherwise. This function always succeeds.

const char *PyExceptionClass_Name(PyObject *ob)

Part of the Stable ABI since version 3.8.
Return **`tp_name`** of the exception class *ob*.

Exception Objects

int PyExceptionInstance_Check(PyObject *op)

Return true if *op* is an instance of **`BaseException`**, false otherwise. This function always succeeds.

PyExceptionInstance_Class(op)

Equivalent to **`Py_TYPE(op)`**.

PyObject *PyException_GetTraceback(PyObject *ex)

Return value: New reference. Part of the Stable ABI.
Return the traceback associated with the exception as a new reference, as accessible from Python through the **`__traceback__`** attribute. If there is no traceback associated, this returns `NULL`.

int PyException_SetTraceback(PyObject *ex, PyObject *tb)

Part of the Stable ABI.
Set the traceback associated with the exception to *tb*. Use `Py_None` to clear it.

PyObject *PyException_GetContext(PyObject *ex)

Return value: New reference. Part of the Stable ABI.
Return the context (another exception instance during whose handling *ex* was raised) associated with the

exception as a new reference, as accessible from Python through the `__context__` attribute. If there is no context associated, this returns `NULL`.

void PyException_SetContext(PyObject *ex, PyObject *ctx)

Part of the Stable ABI.
Set the context associated with the exception to *ctx*. Use `NULL` to clear it. There is no type check to make sure that *ctx* is an exception instance. This steals a reference to *ctx*.

PyObject *PyException_GetCause(PyObject *ex)

Return value: New reference. Part of the Stable ABI.
Return the cause (either an exception instance, or `None`, set by `raise ... from ...`) associated with the exception as a new reference, as accessible from Python through the `__cause__` attribute.

void PyException_SetCause(PyObject *ex, PyObject *cause)

Part of the Stable ABI.
Set the cause associated with the exception to *cause*. Use `NULL` to clear it. There is no type check to make sure that *cause* is either an exception instance or `None`. This steals a reference to *cause*.

The `__suppress_context__` attribute is implicitly set to `True` by this function.

PyObject *PyException_GetArgs(PyObject *ex)

Return value: New reference. Part of the Stable ABI since version 3.12.
Return `args` of exception *ex*.

void PyException_SetArgs(PyObject *ex, PyObject *args)

Part of the Stable ABI since version 3.12.
Set `args` of exception *ex* to *args*.

PyObject *PyUnstable_Exc_PrepReraiseStar(PyObject *orig, PyObject *excs)

This is Unstable API. It may change without warning in minor releases.

Implement part of the interpreter's implementation of **`except*`**. *orig* is the original exception that was caught, and *excs* is the list of the exceptions that need to be raised. This list contains the unhandled part of *orig*, if any, as well as the exceptions that were raised from the **`except*`** clauses (so they have a different traceback from *orig*) and those that were reraised (and have the same traceback as *orig*). Return the **`ExceptionGroup`** that needs to be reraised in the end, or `None` if there is nothing to reraise.

Added in version 3.12.

Unicode Exception Objects

The following functions are used to create and modify Unicode exceptions from C.

PyObject *PyUnicodeDecodeError_Create(const char *encoding, const char *object, Py_ssize_t length, Py_ssize_t start, Py_ssize_t end, const char *reason)

Return value: New reference. Part of the Stable ABI.
Create a **`UnicodeDecodeError`** object with the attributes *encoding*, *object*, *length*, *start*, *end* and *reason*. *encoding* and *reason* are UTF-8 encoded strings.

PyObject *PyUnicodeDecodeError_GetEncoding(PyObject *exc)
PyObject *PyUnicodeEncodeError_GetEncoding(PyObject *exc)

Return value: New reference. Part of the Stable ABI.

Return the *encoding* attribute of the given exception object.

PyObject *PyUnicodeDecodeError_GetObject(PyObject *exc)
PyObject *PyUnicodeEncodeError_GetObject(PyObject *exc)
PyObject *PyUnicodeTranslateError_GetObject(PyObject *exc)

Return value: New reference. Part of the Stable ABI.
Return the *object* attribute of the given exception object.

int PyUnicodeDecodeError_GetStart(PyObject *exc, Py_ssize_t *start)
int PyUnicodeEncodeError_GetStart(PyObject *exc, Py_ssize_t *start)
int PyUnicodeTranslateError_GetStart(PyObject *exc, Py_ssize_t *start)

Part of the Stable ABI.
Get the *start* attribute of the given exception object and place it into **start*. *start* must not be `NULL`. Return `0` on success, `-1` on failure.

If the **`UnicodeError.object`** is an empty sequence, the resulting *start* is `0`. Otherwise, it is clipped to `[0, len(object) - 1]`.

See also

`UnicodeError.start`

int PyUnicodeDecodeError_SetStart(PyObject *exc, Py_ssize_t start)
int PyUnicodeEncodeError_SetStart(PyObject *exc, Py_ssize_t start)
int PyUnicodeTranslateError_SetStart(PyObject *exc, Py_ssize_t start)

Part of the Stable ABI.
Set the *start* attribute of the given exception object to *start*. Return `0` on success, `-1` on failure.

Note

While passing a negative *start* does not raise an exception, the corresponding getters will not consider it as a relative offset.

int PyUnicodeDecodeError_GetEnd(PyObject *exc, Py_ssize_t *end)
int PyUnicodeEncodeError_GetEnd(PyObject *exc, Py_ssize_t *end)
int PyUnicodeTranslateError_GetEnd(PyObject *exc, Py_ssize_t *end)

Part of the Stable ABI.
Get the *end* attribute of the given exception object and place it into **end*. *end* must not be `NULL`. Return `0` on success, `-1` on failure.

If the **`UnicodeError.object`** is an empty sequence, the resulting *end* is `0`. Otherwise, it is clipped to `[1, len(object)]`.

int PyUnicodeDecodeError_SetEnd(PyObject *exc, Py_ssize_t end)
int PyUnicodeEncodeError_SetEnd(PyObject *exc, Py_ssize_t end)
int PyUnicodeTranslateError_SetEnd(PyObject *exc, Py_ssize_t end)

Part of the Stable ABI.
Set the *end* attribute of the given exception object to *end*. Return `0` on success, `-1` on failure.

See also

`UnicodeError.end`

PyObject *PyUnicodeDecodeError_GetReason(PyObject *exc)
PyObject *PyUnicodeEncodeError_GetReason(PyObject *exc)
PyObject *PyUnicodeTranslateError_GetReason(PyObject *exc)

Return value: New reference. Part of the Stable ABI.
Return the *reason* attribute of the given exception object.

int PyUnicodeDecodeError_SetReason(PyObject *exc, const char *reason)
int PyUnicodeEncodeError_SetReason(PyObject *exc, const char *reason)
int PyUnicodeTranslateError_SetReason(PyObject *exc, const char *reason)

Part of the Stable ABI.
Set the *reason* attribute of the given exception object to *reason*. Return `0` on success, `-1` on failure.

Recursion Control

These two functions provide a way to perform safe recursive calls at the C level, both in the core and in extension modules. They are needed if the recursive code does not necessarily invoke Python code (which tracks its recursion depth automatically). They are also not needed for *tp_call* implementations because the call protocol takes care of recursion handling.

int Py_EnterRecursiveCall(const char *where)

Part of the Stable ABI since version 3.9.

Marks a point where a recursive C-level call is about to be performed.

The function then checks if the stack limit is reached. If this is the case, a `RecursionError` is set and a nonzero value is returned. Otherwise, zero is returned.

where should be a UTF-8 encoded string such as `" in instance check"` to be concatenated to the `RecursionError` message caused by the recursion depth limit.

See also

The `PyUnstable_ThreadState_SetStackProtection()` function.

Changed in version 3.9: This function is now also available in the limited API.

void Py_LeaveRecursiveCall(void)

Part of the Stable ABI since version 3.9.

Ends a `Py_EnterRecursiveCall()`. Must be called once for each *successful* invocation of `Py_EnterRecursiveCall()`.

Changed in version 3.9: This function is now also available in the limited API.

Properly implementing `tp_repr` for container types requires special recursion handling. In addition to protecting the stack, `tp_repr` also needs to track objects to prevent cycles. The following two functions facilitate this functionality. Effectively, these are the C equivalent to `reprlib.recursive_repr()`.

int Py_ReprEnter(PyObject *object)

Part of the Stable ABI.

Called at the beginning of the `tp_repr` implementation to detect cycles.

If the object has already been processed, the function returns a positive integer. In that case the `tp_repr` implementation should return a string object indicating a cycle. As examples, `dict` objects return `{...}` and `list` objects return `[...]`.

The function will return a negative integer if the recursion limit is reached. In that case the `tp_repr` implementation should typically return `NULL`.

Otherwise, the function returns zero and the `tp_repr` implementation can continue normally.

void Py_ReprLeave(PyObject *object)

Part of the Stable ABI.
Ends a `Py_ReprEnter()`. Must be called once for each invocation of `Py_ReprEnter()` that returns zero.

int Py_GetRecursionLimit(void)

Part of the Stable ABI.
Get the recursion limit for the current interpreter. It can be set with `Py_SetRecursionLimit()`. The recursion limit prevents the Python interpreter stack from growing infinitely.

This function cannot fail, and the caller must hold an attached thread state.

See also

`sys.getrecursionlimit()`

void Py_SetRecursionLimit(int new_limit)

Part of the Stable ABI.
Set the recursion limit for the current interpreter.

This function cannot fail, and the caller must hold an attached thread state.

See also

`sys.setrecursionlimit()`

Exception and warning types

All standard Python exceptions and warning categories are available as global variables whose names are `PyExc_` followed by the Python exception name. These have the type PyObject*; they are all class objects.

For completeness, here are all the variables:

Exception types

C name	Python name
PyObject *PyExc_BaseException *Part of the Stable ABI.*	`BaseException`
PyObject *PyExc_BaseExceptionGroup *Part of the Stable ABI since version 3.11.*	`BaseExceptionGroup`
PyObject *PyExc_Exception *Part of the Stable ABI.*	`Exception`
PyObject *PyExc_ArithmeticError	`ArithmeticError`

C name	**Python name**
Part of the Stable ABI.	
PyObject *PyExc_AssertionError *Part of the Stable ABI.*	`AssertionError`
PyObject *PyExc_AttributeError *Part of the Stable ABI.*	`AttributeError`
PyObject *PyExc_BlockingIOError *Part of the Stable ABI since version 3.7.*	`BlockingIOError`
PyObject *PyExc_BrokenPipeError *Part of the Stable ABI since version 3.7.*	`BrokenPipeError`
PyObject *PyExc_BufferError *Part of the Stable ABI.*	`BufferError`
PyObject *PyExc_ChildProcessError *Part of the Stable ABI since version 3.7.*	`ChildProcessError`
PyObject *PyExc_ConnectionAbortedError *Part of the Stable ABI since version 3.7.*	`ConnectionAbortedError`

C name	Python name
PyObject *PyExc_ConnectionError *Part of the Stable ABI since version 3.7.*	`ConnectionError`
PyObject *PyExc_ConnectionRefusedError *Part of the Stable ABI since version 3.7.*	`ConnectionRefusedError`
PyObject *PyExc_ConnectionResetError *Part of the Stable ABI since version 3.7.*	`ConnectionResetError`
PyObject *PyExc_EOFError *Part of the Stable ABI.*	`EOFError`
PyObject *PyExc_FileExistsError *Part of the Stable ABI since version 3.7.*	`FileExistsError`
PyObject *PyExc_FileNotFoundError *Part of the Stable ABI since version 3.7.*	`FileNotFoundError`
PyObject *PyExc_FloatingPointError *Part of the Stable ABI.*	`FloatingPointError`

C name	Python name
PyObject *PyExc_GeneratorExit *Part of the Stable ABI.*	`GeneratorExit`
PyObject *PyExc_ImportError *Part of the Stable ABI.*	`ImportError`
PyObject *PyExc_IndentationError *Part of the Stable ABI.*	`IndentationError`
PyObject *PyExc_IndexError *Part of the Stable ABI.*	`IndexError`
PyObject *PyExc_InterruptedError *Part of the Stable ABI since version 3.7.*	`InterruptedError`
PyObject *PyExc_IsADirectoryError *Part of the Stable ABI since version 3.7.*	`IsADirectoryError`
PyObject *PyExc_KeyError *Part of the Stable ABI.*	`KeyError`
PyObject *PyExc_KeyboardInterrupt *Part of the Stable ABI.*	`KeyboardInterrupt`
PyObject *PyExc_LookupError	`LookupError`

C name	**Python name**
Part of the Stable ABI.	
PyObject *PyExc_MemoryError *Part of the Stable ABI.*	`MemoryError`
PyObject *PyExc_ModuleNotFoundError *Part of the Stable ABI since version 3.6.*	`ModuleNotFoundError`
PyObject *PyExc_NameError *Part of the Stable ABI.*	`NameError`
PyObject *PyExc_NotADirectoryError *Part of the Stable ABI since version 3.7.*	`NotADirectoryError`
PyObject *PyExc_NotImplementedError *Part of the Stable ABI.*	`NotImplementedError`
PyObject *PyExc_OSError *Part of the Stable ABI.*	`OSError`
PyObject *PyExc_OverflowError *Part of the Stable ABI.*	`OverflowError`
PyObject *PyExc_PermissionError	`PermissionError`

C name	**Python name**
Part of the Stable ABI since version 3.7.	
PyObject *PyExc_ProcessLookupError *Part of the Stable ABI since version 3.7.*	`ProcessLookupError`
PyObject *PyExc_PythonFinalizationError	`PythonFinalizationError`
PyObject *PyExc_RecursionError *Part of the Stable ABI since version 3.7.*	`RecursionError`
PyObject *PyExc_ReferenceError *Part of the Stable ABI.*	`ReferenceError`
PyObject *PyExc_RuntimeError *Part of the Stable ABI.*	`RuntimeError`
PyObject *PyExc_StopAsyncIteration *Part of the Stable ABI since version 3.7.*	`StopAsyncIteration`
PyObject *PyExc_StopIteration *Part of the Stable ABI.*	`StopIteration`
PyObject *PyExc_SyntaxError	`SyntaxError`

C name	Python name
Part of the Stable ABI.	
PyObject *PyExc_SystemError *Part of the Stable ABI.*	`SystemError`
PyObject *PyExc_SystemExit *Part of the Stable ABI.*	`SystemExit`
PyObject *PyExc_TabError *Part of the Stable ABI.*	`TabError`
PyObject *PyExc_TimeoutError *Part of the Stable ABI since version 3.7.*	`TimeoutError`
PyObject *PyExc_TypeError *Part of the Stable ABI.*	`TypeError`
PyObject *PyExc_UnboundLocalError *Part of the Stable ABI.*	`UnboundLocalError`
PyObject *PyExc_UnicodeDecodeError *Part of the Stable ABI.*	`UnicodeDecodeError`
PyObject *PyExc_UnicodeEncodeError *Part of the Stable ABI.*	`UnicodeEncodeError`

C name	Python name
PyObject *PyExc_UnicodeError *Part of the Stable ABI.*	`UnicodeError`
PyObject *PyExc_UnicodeTranslateError *Part of the Stable ABI.*	`UnicodeTranslateError`
PyObject *PyExc_ValueError *Part of the Stable ABI.*	`ValueError`
PyObject *PyExc_ZeroDivisionError *Part of the Stable ABI.*	`ZeroDivisionError`

Added in version 3.3: `PyExc_BlockingIOError`, `PyExc_BrokenPipeError`, `PyExc_ChildProcessError`, `PyExc_ConnectionError`, `PyExc_ConnectionAbortedError`, `PyExc_ConnectionRefusedError`, `PyExc_ConnectionResetError`, `PyExc_FileExistsError`, `PyExc_FileNotFoundError`, `PyExc_InterruptedError`, `PyExc_IsADirectoryError`, `PyExc_NotADirectoryError`, `PyExc_PermissionError`, `PyExc_ProcessLookupError` and `PyExc_TimeoutError` were introduced following **PEP 3151** [https://peps.python.org/pep-3151/].

Added in version 3.5: `PyExc_StopAsyncIteration` and `PyExc_RecursionError`.

Added in version 3.6: `PyExc_ModuleNotFoundError`.

Added in version 3.11: `PyExc_BaseExceptionGroup`.

OSError aliases

The following are a compatibility aliases to `PyExc_OSError`.

Changed in version 3.3: These aliases used to be separate exception types.

C name	Python name	Notes
PyObject *PyExc_EnvironmentError *Part of the Stable ABI.*	`OSError`	
PyObject *PyExc_IOError *Part of the Stable ABI.*	`OSError`	
PyObject *PyExc_WindowsError *Part of the Stable ABI on Windows since version 3.7.*	`OSError`	[win]

Notes:

[win]

`PyExc_WindowsError` is only defined on Windows; protect code that uses this by testing that the preprocessor macro `MS_WINDOWS` is defined.

Warning types

C name	Python name
PyObject *PyExc_Warning *Part of the Stable ABI.*	`Warning`
PyObject *PyExc_BytesWarning *Part of the Stable ABI.*	`BytesWarning`
PyObject *PyExc_DeprecationWarning *Part of the Stable ABI.*	`DeprecationWarning`
PyObject *PyExc_EncodingWarning *Part of the Stable ABI since version 3.10.*	`EncodingWarning`
PyObject *PyExc_FutureWarning *Part of the Stable ABI.*	`FutureWarning`
PyObject *PyExc_ImportWarning *Part of the Stable ABI.*	`ImportWarning`
PyObject *PyExc_PendingDeprecationWarning *Part of the Stable ABI.*	`PendingDeprecationWarning`
PyObject *PyExc_ResourceWarning *Part of the Stable ABI since version 3.7.*	`ResourceWarning`

C name	**Python name**
PyObject *PyExc_RuntimeWarning *Part of the Stable ABI.*	`RuntimeWarning`
PyObject *PyExc_SyntaxWarning *Part of the Stable ABI.*	`SyntaxWarning`
PyObject *PyExc_UnicodeWarning *Part of the Stable ABI.*	`UnicodeWarning`
PyObject *PyExc_UserWarning *Part of the Stable ABI.*	`UserWarning`

Added in version 3.2: `PyExc_ResourceWarning`.

Added in version 3.10: `PyExc_EncodingWarning`.

Tracebacks

PyTypeObject PyTraceBack_Type

Part of the Stable ABI.

Type object for traceback objects. This is available as `types.TracebackType` in the Python layer.

int PyTraceBack_Check(PyObject *op)

Return true if *op* is a traceback object, false otherwise. This function does not account for subtypes.

int PyTraceBack_Here(PyFrameObject *f)

Part of the Stable ABI.

Replace the `__traceback__` attribute on the current exception with a new traceback prepending *f* to the existing chain.

Calling this function without an exception set is undefined behavior.

This function returns `0` on success, and returns `-1` with an exception set on failure.

int PyTraceBack_Print(PyObject *tb, PyObject *f)

Part of the Stable ABI.
Write the traceback *tb* into the file *f*.

This function returns `0` on success, and returns `-1` with an exception set on failure.

Defining extension modules

A C extension for CPython is a shared library (for example, a `.so` file on Linux, `.pyd` DLL on Windows), which is loadable into the Python process (for example, it is compiled with compatible compiler settings), and which exports an initialization function.

To be importable by default (that is, by `importlib.machinery.ExtensionFileLoader`), the shared library must be available on `sys.path`, and must be named after the module name plus an extension listed in `importlib.machinery.EXTENSION_SUFFIXES`.

Note

Building, packaging and distributing extension modules is best done with third-party tools, and is out of scope of this document. One suitable tool is Setuptools, whose documentation can be found at https://setuptools.pypa.io/en/latest/setuptools.html.

Normally, the initialization function returns a module definition initialized using `PyModuleDef_Init()`. This allows splitting the creation process into several phases:

- Before any substantial code is executed, Python can determine which capabilities the module supports, and it can adjust the environment or refuse loading an incompatible extension.

- By default, Python itself creates the module object – that is, it does the equivalent of `object.__new__()` for classes. It also sets initial attributes like `__package__` and `__loader__`.
- Afterwards, the module object is initialized using extension-specific code – the equivalent of `__init__()` on classes.

This is called *multi-phase initialization* to distinguish it from the legacy (but still supported) *single-phase initialization* scheme, where the initialization function returns a fully constructed module. See the single-phase-initialization section below for details.

Changed in version 3.5: Added support for multi-phase initialization (**PEP 489** [https://peps.python.org/pep-0489/]).

Multiple module instances

By default, extension modules are not singletons. For example, if the `sys.modules` entry is removed and the module is re-imported, a new module object is created, and typically populated with fresh method and type objects. The old module is subject to normal garbage collection. This mirrors the behavior of pure-Python modules.

Additional module instances may be created in sub-interpreters or after Python runtime reinitialization (`Py_Finalize()` and `Py_Initialize()`). In these cases, sharing Python objects between module instances would likely cause crashes or undefined behavior.

To avoid such issues, each instance of an extension module should be *isolated*: changes to one instance should not implicitly affect the others, and all state owned by the module, including references to Python objects, should be specific to a particular module instance. See Isolating Extension Modules for more details and a practical guide.

A simpler way to avoid these issues is raising an error on repeated initialization.

All modules are expected to support sub-interpreters, or otherwise explicitly signal a lack of support. This is usually achieved by isolation or blocking repeated initialization, as above. A module may also be limited to the main interpreter using the **`Py_mod_multiple_interpreters`** slot.

Initialization function

The initialization function defined by an extension module has the following signature:

PyObject *PyInit_modulename(void)

Its name should be `PyInit_<name>`, with `<name>` replaced by the name of the module.

For modules with ASCII-only names, the function must instead be named `PyInit_<name>`, with `<name>` replaced by the name of the module. When using Multi-phase initialization, non-ASCII module names are allowed. In this case, the initialization function name is `PyInitU_<name>`, with `<name>` encoded using Python’s *punycode* encoding with hyphens replaced by underscores. In Python:

```
def initfunc_name(name):
    try:
        suffix = b'_' + name.encode('ascii')
    except UnicodeEncodeError:
        suffix = b'U_' + name.encode('punycode').replace(b'-',
b'_')
    return b'PyInit' + suffix
```

It is recommended to define the initialization function using a helper macro:

PyMODINIT_FUNC

Declare an extension module initialization function. This macro:

- specifies the PyObject* return type,
- adds any special linkage declarations required by the platform, and
- for C++, declares the function as `extern "C"`.

For example, a module called `spam` would be defined like this:

```
static struct PyModuleDef spam_module = {
    .m_base = PyModuleDef_HEAD_INIT,
    .m_name = "spam",
    ...
};

PyMODINIT_FUNC
PyInit_spam(void)
{
    return PyModuleDef_Init(&spam_module);
}
```

It is possible to export multiple modules from a single shared library by defining multiple initialization functions. However, importing them requires using symbolic links or a custom importer, because by default only the function corresponding to the filename is found. See the Multiple modules in one library [https://peps.python.org/pep-0489/#multiple-modules-in-one-library] section in **PEP 489** [https://peps.python.org/pep-0489/] for details.

The initialization function is typically the only non-`static` item defined in the module’s C source.

Multi-phase initialization

Normally, the initialization function (`PyInit_modulename`) returns a **`PyModuleDef`** instance with non-`NULL` **`m_slots`**. Before it is returned, the `PyModuleDef` instance must be initialized using the following function:

PyObject *PyModuleDef_Init(PyModuleDef *def)

Part of the Stable ABI since version 3.5.

Ensure a module definition is a properly initialized Python object that correctly reports its type and a reference count.

Return *def* cast to `PyObject*`, or `NULL` if an error occurred.

Calling this function is required for Multi-phase initialization. It should not be used in other contexts.

Note that Python assumes that `PyModuleDef` structures are statically allocated. This function may return either a new reference or a borrowed one; this reference must not be released.

Added in version 3.5.

Legacy single-phase initialization

Attention

Single-phase initialization is a legacy mechanism to initialize extension modules, with known drawbacks and design flaws. Extension module authors are encouraged to use multi-phase initialization instead.

In single-phase initialization, the initialization function (`PyInit_modulename`) should create, populate and return a module object. This is typically done using **`PyModule_Create()`** and functions like **`PyModule_AddObjectRef()`**.

Single-phase initialization differs from the default in the following ways:

- Single-phase modules are, or rather *contain*, “singletons”.

  When the module is first initialized, Python saves the contents of the module’s `__dict__` (that is, typically, the module’s functions and types).

  For subsequent imports, Python does not call the initialization function again. Instead, it creates a new module object with a new `__dict__`, and copies the saved contents to it. For example, given a single-phase module `_testsinglephase` [[1]] that defines a function `sum` and an exception class `error`:

  ```
  >>> import sys
  >>> import _testsinglephase as one
  >>> del sys.modules['_testsinglephase']
  >>> import _testsinglephase as two
  >>> one is two
  False
  >>> one.__dict__ is two.__dict__
  False
  >>> one.sum is two.sum
  True
  >>> one.error is two.error
  True
  ```

  The exact behavior should be considered a CPython implementation detail.

- To work around the fact that `PyInit_modulename` does not take a *spec* argument, some state of the import machinery is saved and applied to the first suitable module created during the `PyInit_modulename` call. Specifically, when a sub-module is imported, this mechanism prepends the parent package name to the name of the module.

  A single-phase `PyInit_modulename` function should create "its" module object as soon as possible, before any other module objects can be created.

- Non-ASCII module names (`PyInitU_modulename`) are not supported.

- Single-phase modules support module lookup functions like **`PyState_FindModule()`**.

[1]

`_testsinglephase` is an internal module used in CPython's self-test suite; your installation may or may not include it.

Utilities

The functions in this chapter perform various utility tasks, ranging from helping C code be more portable across platforms, using Python modules from C, and parsing function arguments and constructing Python values from C values.

- Operating System Utilities
- System Functions
- Process Control
- Importing Modules
- Data marshalling support
- Parsing arguments and building values
  - Parsing arguments
    - Strings and buffers
    - Numbers
    - Other objects
    - API Functions
  - Building values
- String conversion and formatting
- Character classification and conversion
- PyHash API
- Reflection
- Codec registry and support functions
  - Codec lookup API
  - Registry API for Unicode encoding error handlers
  - Codec utility variables
- PyTime C API
  - Types
  - Clock Functions
  - Raw Clock Functions
  - Conversion functions

- Support for Perf Maps

Operating System Utilities

PyObject *PyOS_FSPath(PyObject *path)

Return value: New reference. Part of the Stable ABI since version 3.6.

Return the file system representation for *path*. If the object is a **`str`** or **`bytes`** object, then a new strong reference is returned. If the object implements the **`os.PathLike`** interface, then **`__fspath__()`** is returned as long as it is a **`str`** or **`bytes`** object. Otherwise **`TypeError`** is raised and `NULL` is returned.

Added in version 3.6.

int Py_FdIsInteractive(FILE *fp, const char *filename)

Return true (nonzero) if the standard I/O file *fp* with name *filename* is deemed interactive. This is the case for files for which `isatty(fileno(fp))` is true. If the **`PyConfig.interactive`** is non-zero, this function also returns true if the *filename* pointer is `NULL` or if the name is equal to one of the strings `'<stdin>'` or `'???'`.

This function must not be called before Python is initialized.

void PyOS_BeforeFork()

Part of the Stable ABI on platforms with fork() since version 3.7.

Function to prepare some internal state before a process fork. This should be called before calling **`fork()`** or any similar function that clones the current process. Only available on systems where **`fork()`** is defined.

Warning

The C **`fork()`** call should only be made from the “main” thread (of the “main” interpreter). The same is true for `PyOS_BeforeFork()`.

Added in version 3.7.

void PyOS_AfterFork_Parent()

Part of the Stable ABI on platforms with fork() since version 3.7.

Function to update some internal state after a process fork. This should be called from the parent process after calling **`fork()`** or any similar function that clones the current process, regardless of whether process cloning was successful. Only available on systems where **`fork()`** is defined.

Warning

The C **`fork()`** call should only be made from the “main” thread (of the “main” interpreter). The same is true for `PyOS_AfterFork_Parent()`.

Added in version 3.7.

void PyOS_AfterFork_Child()

Part of the Stable ABI on platforms with fork() since version 3.7.

Function to update internal interpreter state after a process fork. This must be called from the child process after calling **`fork()`**, or any similar function that clones the current process, if there is any chance the process will call back into the Python interpreter. Only available on systems where **`fork()`** is defined.

Warning

The C **`fork()`** call should only be made from the “main” thread (of the “main” interpreter). The same is true for `PyOS_AfterFork_Child()`.

Added in version 3.7.

See also

`os.register_at_fork()` allows registering custom Python functions to be called by **`PyOS_BeforeFork()`**, **`PyOS_AfterFork_Parent()`** and **`PyOS_AfterFork_Child()`**.

void PyOS_AfterFork()

Part of the Stable ABI on platforms with fork().
Function to update some internal state after a process fork; this should be called in the new process if the Python interpreter will continue to be used. If a new executable is loaded into the new process, this function does not need to be called.

Deprecated since version 3.7: This function is superseded by **`PyOS_AfterFork_Child()`**.

int PyOS_CheckStack()

Part of the Stable ABI on platforms with USE_STACKCHECK since version 3.7.
Return true when the interpreter runs out of stack space. This is a reliable check, but is only available when **`USE_STACKCHECK`** is defined (currently on certain versions of Windows using the Microsoft Visual C++ compiler). **`USE_STACKCHECK`** will be defined automatically; you should never change the definition in your own code.

typedef void (*PyOS_sighandler_t)(int)

Part of the Stable ABI.

PyOS_sighandler_t PyOS_getsig(int i)

Part of the Stable ABI.

Return the current signal handler for signal *i*. This is a thin wrapper around either **`sigaction()`** or **`signal()`**. Do not call those functions directly!

PyOS_sighandler_t PyOS_setsig(int i, PyOS_sighandler_t h)

Part of the Stable ABI.

Set the signal handler for signal *i* to be *h*; return the old signal handler. This is a thin wrapper around either **`sigaction()`** or **`signal()`**. Do not call those functions directly!

int PyOS_InterruptOccurred(void)

Part of the Stable ABI.

Check if a **`SIGINT`** signal has been received.

Returns `1` if a **`SIGINT`** has occurred and clears the signal flag, or `0` otherwise.

In most cases, you should prefer **`PyErr_CheckSignals()`** over this function. **`PyErr_CheckSignals()`** invokes the appropriate signal handlers for all pending signals, allowing Python code to handle the signal properly. This function only detects **`SIGINT`** and does not invoke any Python signal handlers.

This function is async-signal-safe and this function cannot fail. The caller must hold an attached thread state.

wchar_t *Py_DecodeLocale(const char *arg, size_t *size)

Part of the Stable ABI since version 3.7.

Warning

This function should not be called directly: use the `PyConfig` API with the `PyConfig_SetBytesString()` function which ensures that Python is preinitialized.

This function must not be called before Python is preinitialized and so that the LC_CTYPE locale is properly configured: see the `Py_PreInitialize()` function.

Decode a byte string from the filesystem encoding and error handler. If the error handler is surrogateescape error handler, undecodable bytes are decoded as characters in range U+DC80..U+DCFF; and if a byte sequence can be decoded as a surrogate character, the bytes are escaped using the surrogateescape error handler instead of decoding them.

Return a pointer to a newly allocated wide character string, use `PyMem_RawFree()` to free the memory. If size is not `NULL`, write the number of wide characters excluding the null character into `*size`

Return `NULL` on decoding error or memory allocation error. If *size* is not `NULL`, `*size` is set to `(size_t)-1` on memory error or set to `(size_t)-2` on decoding error.

The filesystem encoding and error handler are selected by `PyConfig_Read()`: see `filesystem_encoding` and `filesystem_errors` members of `PyConfig`.

Decoding errors should never happen, unless there is a bug in the C library.

Use the `Py_EncodeLocale()` function to encode the character string back to a byte string.

See also

The `PyUnicode_DecodeFSDefaultAndSize()` and `PyUnicode_DecodeLocaleAndSize()` functions.

Added in version 3.5.

Changed in version 3.7: The function now uses the UTF-8 encoding in the Python UTF-8 Mode.

Changed in version 3.8: The function now uses the UTF-8 encoding on Windows if `PyPreConfig.legacy_windows_fs_encoding` is zero;

char *Py_EncodeLocale(const wchar_t *text, size_t *error_pos)

Part of the Stable ABI since version 3.7.

Encode a wide character string to the filesystem encoding and error handler. If the error handler is surrogateescape error handler, surrogate characters in the range U+DC80..U+DCFF are converted to bytes 0x80..0xFF.

Return a pointer to a newly allocated byte string, use `PyMem_Free()` to free the memory. Return `NULL` on encoding error or memory allocation error.

If error_pos is not `NULL`, `*error_pos` is set to `(size_t)-1` on success, or set to the index of the invalid character on encoding error.

The filesystem encoding and error handler are selected by `PyConfig_Read()`: see `filesystem_encoding` and `filesystem_errors` members of `PyConfig`.

Use the `Py_DecodeLocale()` function to decode the bytes string back to a wide character string.

Warning

This function must not be called before Python is preinitialized and so that the LC_CTYPE locale is properly configured: see the `Py_PreInitialize()` function.

See also

The `PyUnicode_EncodeFSDefault()` and `PyUnicode_EncodeLocale()` functions.

Added in version 3.5.

Changed in version 3.7: The function now uses the UTF-8 encoding in the Python UTF-8 Mode.

Changed in version 3.8: The function now uses the UTF-8 encoding on Windows if `PyPreConfig.legacy_windows_fs_encoding` is zero.

FILE *Py_fopen(PyObject *path, const char *mode)

Similar to `fopen()`, but *path* is a Python object and an exception is set on error.

path must be a `str` object, a `bytes` object, or a path-like object.

On success, return the new file pointer. On error, set an exception and return `NULL`.

The file must be closed by `Py_fclose()` rather than calling directly `fclose()`.

The file descriptor is created non-inheritable (**PEP 446** [https://peps.python.org/pep-0446/]).

The caller must have an attached thread state.

Added in version 3.14.

int Py_fclose(FILE *file)

Close a file that was opened by **`Py_fopen()`**.

On success, return `0`. On error, return `EOF` and `errno` is set to indicate the error. In either case, any further access (including another call to **`Py_fclose()`**) to the stream results in undefined behavior.

Added in version 3.14.

System Functions

These are utility functions that make functionality from the **`sys`** module accessible to C code. They all work with the current interpreter thread's **`sys`** module's dict, which is contained in the internal thread state structure.

PyObject *PySys_GetObject(const char *name)

Return value: Borrowed reference. Part of the Stable ABI.
Return the object *name* from the **`sys`** module or `NULL` if it does not exist, without setting an exception.

int PySys_SetObject(const char *name, PyObject *v)

Part of the Stable ABI.
Set *name* in the **`sys`** module to *v* unless *v* is `NULL`, in which case *name* is deleted from the sys module. Returns `0` on success, `-1` on error.

void PySys_ResetWarnOptions()

Part of the Stable ABI.

Reset `sys.warnoptions` to an empty list. This function may be called prior to `Py_Initialize()`.

Deprecated since version 3.13, will be removed in version 3.15: Clear `sys.warnoptions` and **`warnings.filters`** instead.

void PySys_WriteStdout(const char *format, ...)

Part of the Stable ABI.

Write the output string described by *format* to `sys.stdout`. No exceptions are raised, even if truncation occurs (see below).

format should limit the total size of the formatted output string to 1000 bytes or less – after 1000 bytes, the output string is truncated. In particular, this means that no unrestricted “%s” formats should occur; these should be limited using “%.<N>s” where <N> is a decimal number calculated so that <N> plus the maximum size of other formatted text does not exceed 1000 bytes. Also watch out for “%f”, which can print hundreds of digits for very large numbers.

If a problem occurs, or `sys.stdout` is unset, the formatted message is written to the real (C level) *stdout*.

void PySys_WriteStderr(const char *format, ...)

Part of the Stable ABI.

As `PySys_WriteStdout()`, but write to `sys.stderr` or *stderr* instead.

void PySys_FormatStdout(const char *format, ...)

Part of the Stable ABI.

Function similar to PySys_WriteStdout() but format the message using `PyUnicode_FromFormatV()` and don’t truncate the message to an arbitrary length.

Added in version 3.2.

void PySys_FormatStderr(const char *format, ...)

Part of the Stable ABI.

As `PySys_FormatStdout()`, but write to `sys.stderr` or *stderr* instead.

Added in version 3.2.

PyObject *PySys_GetXOptions()

Return value: Borrowed reference. Part of the Stable ABI since version 3.7.

Return the current dictionary of `-X` options, similarly to `sys._xoptions`. On error, `NULL` is returned and an exception is set.

Added in version 3.2.

int PySys_Audit(const char *event, const char *format, ...)

Part of the Stable ABI since version 3.13.

Raise an auditing event with any active hooks. Return zero for success and non-zero with an exception set on failure.

The *event* string argument must not be *NULL*.

If any hooks have been added, *format* and other arguments will be used to construct a tuple to pass. Apart from `N`, the same format characters as used in `Py_BuildValue()` are available. If the built value is not a tuple, it will be added into a single-element tuple.

The `N` format option must not be used. It consumes a reference, but since there is no way to know whether arguments to this function will be consumed, using it may cause reference leaks.

Note that `#` format characters should always be treated as **`Py_ssize_t`**, regardless of whether `PY_SSIZE_T_CLEAN` was defined.

`sys.audit()` performs the same function from Python code.

See also **`PySys_AuditTuple()`**.

Added in version 3.8.

Changed in version 3.8.2: Require **`Py_ssize_t`** for `#` format characters. Previously, an unavoidable deprecation warning was raised.

int PySys_AuditTuple(const char *event, PyObject *args)

Part of the Stable ABI since version 3.13.
Similar to **`PySys_Audit()`**, but pass arguments as a Python object. *args* must be a **`tuple`**. To pass no arguments, *args* can be *NULL*.

Added in version 3.13.

int PySys_AddAuditHook(Py_AuditHookFunction hook, void *userData)

Append the callable *hook* to the list of active auditing hooks. Return zero on success and non-zero on failure. If the runtime has been initialized, also set an error on failure. Hooks added through this API are called for all interpreters created by the runtime.

The *userData* pointer is passed into the hook function. Since hook functions may be called from different runtimes, this pointer should not refer directly to Python state.

This function is safe to call before **`Py_Initialize()`**. When called after runtime initialization, existing audit hooks are notified and may silently abort the operation by raising an error subclassed from **`Exception`** (other errors will not be silenced).

The hook function is always called with an attached thread state by the Python interpreter that raised the event.

See **PEP 578** [https://peps.python.org/pep-0578/] for a detailed description of auditing. Functions in the runtime and standard library that raise events are listed in the audit events table. Details are in each function's documentation.

If the interpreter is initialized, this function raises an auditing event `sys.addaudithook` with no arguments. If any existing hooks raise an exception derived from **`Exception`**, the new hook will not be added and the exception is cleared. As a result, callers cannot assume that their hook has been added unless they control all existing hooks.

typedef int (*Py_AuditHookFunction)(const char *event, PyObject *args, void *userData)

> The type of the hook function. *event* is the C string event argument passed to **`PySys_Audit()`** or **`PySys_AuditTuple()`**. *args* is guaranteed to be a **`PyTupleObject`**. *userData* is the argument passed to PySys_AddAuditHook().

Added in version 3.8.

Process Control

void Py_FatalError(const char *message)

Part of the Stable ABI.

Print a fatal error message and kill the process. No cleanup is performed. This function should only be invoked when a condition is detected that would make it dangerous to continue using the Python interpreter; e.g., when the object administration appears to be corrupted. On Unix, the standard C library function **`abort()`** is called which will attempt to produce a `core` file.

The `Py_FatalError()` function is replaced with a macro which logs automatically the name of the current function, unless the `Py_LIMITED_API` macro is defined.

Changed in version 3.9: Log the function name automatically.

void Py_Exit(int status)

Part of the Stable ABI.

Exit the current process. This calls **`Py_FinalizeEx()`** and then calls the standard C library function `exit(status)`. If **`Py_FinalizeEx()`** indicates an error, the exit status is set to 120.

Changed in version 3.6: Errors from finalization no longer ignored.

int Py_AtExit(void (*func)())

Part of the Stable ABI.

Register a cleanup function to be called by `Py_FinalizeEx()`. The cleanup function will be called with no arguments and should return no value. At most 32 cleanup functions can be registered. When the registration is successful, `Py_AtExit()` returns `0`; on failure, it returns `-1`. The cleanup function registered last is called first. Each cleanup function will be called at most once. Since Python’s internal finalization will have completed before the cleanup function, no Python APIs should be called by *func*.

See also

`PyUnstable_AtExit()` for passing a `void *data` argument.

Importing Modules

PyObject *PyImport_ImportModule(const char *name)

Return value: New reference. Part of the Stable ABI.
This is a wrapper around `PyImport_Import()` which takes a const char* as an argument instead of a PyObject*.

PyObject *PyImport_ImportModuleNoBlock(const char *name)

Return value: New reference. Part of the Stable ABI.
This function is a deprecated alias of `PyImport_ImportModule()`.

Changed in version 3.3: This function used to fail immediately when the import lock was held by another thread. In Python 3.3 though, the locking scheme switched to per-module locks for most purposes, so this function's special behaviour isn't needed anymore.

Deprecated since version 3.13, will be removed in version 3.15: Use `PyImport_ImportModule()` instead.

PyObject *PyImport_ImportModuleEx(const char *name, PyObject *globals, PyObject *locals, PyObject *fromlist)

Return value: New reference.
Import a module. This is best described by referring to the built-in Python function `__import__()`.

The return value is a new reference to the imported module or top-level package, or `NULL` with an exception set on failure. Like for `__import__()`, the return value when a submodule of a package was requested is normally the top-level package, unless a non-empty *fromlist* was given.

Failing imports remove incomplete module objects, like with **`PyImport_ImportModule()`**.

PyObject *PyImport_ImportModuleLevelObject(PyObject *name, PyObject *globals, PyObject *locals, PyObject *fromlist, int level)

Return value: New reference. Part of the Stable ABI since version 3.7.
Import a module. This is best described by referring to the built-in Python function **`__import__()`**, as the standard **`__import__()`** function calls this function directly.

The return value is a new reference to the imported module or top-level package, or `NULL` with an exception set on failure. Like for **`__import__()`**, the return value when a submodule of a package was requested is normally the top-level package, unless a non-empty *fromlist* was given.

Added in version 3.3.

PyObject *PyImport_ImportModuleLevel(const char *name, PyObject *globals, PyObject *locals, PyObject *fromlist, int level)

Return value: New reference. Part of the Stable ABI.
Similar to **`PyImport_ImportModuleLevelObject()`**, but the name is a UTF-8 encoded string instead of a Unicode object.

Changed in version 3.3: Negative values for *level* are no longer accepted.

PyObject *PyImport_Import(PyObject *name)

Return value: New reference. Part of the Stable ABI.

This is a higher-level interface that calls the current “import hook function” (with an explicit *level* of 0, meaning absolute import). It invokes the `__import__()` function from the `__builtins__` of the current globals. This means that the import is done using whatever import hooks are installed in the current environment.

This function always uses absolute imports.

PyObject *PyImport_ReloadModule(PyObject *m)

Return value: New reference. Part of the Stable ABI.
Reload a module. Return a new reference to the reloaded module, or `NULL` with an exception set on failure (the module still exists in this case).

PyObject *PyImport_AddModuleRef(const char *name)

Return value: New reference. Part of the Stable ABI since version 3.13.
Return the module object corresponding to a module name.

The *name* argument may be of the form `package.module`. First check the modules dictionary if there’s one there, and if not, create a new one and insert it in the modules dictionary.

Return a strong reference to the module on success. Return `NULL` with an exception set on failure.

The module name *name* is decoded from UTF-8.

This function does not load or import the module; if the module wasn’t already loaded, you will get an empty module object. Use `PyImport_ImportModule()` or one of its variants to import a module. Package structures implied by a dotted name for *name* are not created if not already present.

Added in version 3.13.

PyObject *PyImport_AddModuleObject(PyObject *name)

Return value: Borrowed reference. Part of the Stable ABI since version 3.7.
Similar to **`PyImport_AddModuleRef()`**, but return a borrowed reference and *name* is a Python **`str`** object.

Added in version 3.3.

PyObject *PyImport_AddModule(const char *name)

Return value: Borrowed reference. Part of the Stable ABI.
Similar to **`PyImport_AddModuleRef()`**, but return a borrowed reference.

PyObject *PyImport_ExecCodeModule(const char *name, PyObject *co)

Return value: New reference. Part of the Stable ABI.
Given a module name (possibly of the form `package.module`) and a code object read from a Python bytecode file or obtained from the built-in function **`compile()`**, load the module. Return a new reference to the module object, or `NULL` with an exception set if an error occurred. *name* is removed from **`sys.modules`** in error cases, even if *name* was already in **`sys.modules`** on entry to **`PyImport_ExecCodeModule()`**. Leaving incompletely initialized modules in **`sys.modules`** is dangerous, as imports of such modules have no way to know that the module object is an unknown (and probably damaged with respect to the module author's intents) state.

The module's **`__spec__`** and **`__loader__`** will be set, if not set already, with the appropriate values. The spec's loader will be set to the module’s `__loader__` (if set) and to an instance of `SourceFileLoader` otherwise.

The module’s `__file__` attribute will be set to the code object’s `co_filename`. If applicable, `__cached__` will also be set.

This function will reload the module if it was already imported. See `PyImport_ReloadModule()` for the intended way to reload a module.

If *name* points to a dotted name of the form `package.module`, any package structures not already created will still not be created.

See also `PyImport_ExecCodeModuleEx()` and `PyImport_ExecCodeModuleWithPathnames()`.

Changed in version 3.12: The setting of `__cached__` and `__loader__` is deprecated. See `ModuleSpec` for alternatives.

PyObject *PyImport_ExecCodeModuleEx(const char *name, PyObject *co, const char *pathname)

Return value: New reference. Part of the Stable ABI.
Like `PyImport_ExecCodeModule()`, but the `__file__` attribute of the module object is set to *pathname* if it is non-`NULL`.

See also `PyImport_ExecCodeModuleWithPathnames()`.

PyObject *PyImport_ExecCodeModuleObject(PyObject *name, PyObject *co, PyObject *pathname, PyObject *cpathname)

Return value: New reference. Part of the Stable ABI since version 3.7.
Like `PyImport_ExecCodeModuleEx()`, but the `__cached__` attribute of the module object is set to *cpathname* if it is

non-`NULL`. Of the three functions, this is the preferred one to use.

Added in version 3.3.

Changed in version 3.12: Setting **`__cached__`** is deprecated. See **`ModuleSpec`** for alternatives.

PyObject *PyImport_ExecCodeModuleWithPathnames(const char *name, PyObject *co, const char *pathname, const char *cpathname)

Return value: New reference. Part of the Stable ABI.
Like **`PyImport_ExecCodeModuleObject()`**, but *name*, *pathname* and *cpathname* are UTF-8 encoded strings. Attempts are also made to figure out what the value for *pathname* should be from *cpathname* if the former is set to `NULL`.

Added in version 3.2.

Changed in version 3.3: Uses **`imp.source_from_cache()`** in calculating the source path if only the bytecode path is provided.

Changed in version 3.12: No longer uses the removed **`imp`** module.

long PyImport_GetMagicNumber()

Part of the Stable ABI.
Return the magic number for Python bytecode files (a.k.a. `.pyc` file). The magic number should be present in the first four bytes of the bytecode file, in little-endian byte order. Returns `-1` on error.

Changed in version 3.3: Return value of `-1` upon failure.

const char *PyImport_GetMagicTag()

Part of the Stable ABI.
Return the magic tag string for **PEP 3147** [https://peps.python.org/pep-3147/] format Python bytecode file names. Keep in mind that the value at `sys.implementation.cache_tag` is authoritative and should be used instead of this function.

Added in version 3.2.

PyObject *PyImport_GetModuleDict()

Return value: Borrowed reference. Part of the Stable ABI.
Return the dictionary used for the module administration (a.k.a. `sys.modules`). Note that this is a per-interpreter variable.

PyObject *PyImport_GetModule(PyObject *name)

Return value: New reference. Part of the Stable ABI since version 3.8.
Return the already imported module with the given name. If the module has not been imported yet then returns `NULL` but does not set an error. Returns `NULL` and sets an error if the lookup failed.

Added in version 3.7.

PyObject *PyImport_GetImporter(PyObject *path)

Return value: New reference. Part of the Stable ABI.
Return a finder object for a **`sys.path`**/**`pkg.__path__`** item *path*, possibly by fetching it from the **`sys.path_importer_cache`** dict. If it wasn’t yet cached, traverse **`sys.path_hooks`** until a hook is found that can handle the path item. Return `None` if no hook could; this tells our caller that the path based finder could not find a finder for this path item. Cache the result in **`sys.path_importer_cache`**. Return a new reference to the finder object.

int PyImport_ImportFrozenModuleObject(PyObject *name)

Part of the Stable ABI since version 3.7.

Load a frozen module named *name*. Return `1` for success, `0` if the module is not found, and `-1` with an exception set if the initialization failed. To access the imported module on a successful load, use **`PyImport_ImportModule()`**. (Note the misnomer — this function would reload the module if it was already imported.)

Added in version 3.3.

Changed in version 3.4: The `__file__` attribute is no longer set on the module.

int PyImport_ImportFrozenModule(const char *name)

Part of the Stable ABI.

Similar to **`PyImport_ImportFrozenModuleObject()`**, but the name is a UTF-8 encoded string instead of a Unicode object.

struct _frozen

This is the structure type definition for frozen module descriptors, as generated by the **freeze** utility (see `Tools/freeze/` in the Python source distribution). Its definition, found in `Include/import.h`, is:

```
struct _frozen {
    const char *name;
    const unsigned char *code;
    int size;
    bool is_package;
};
```

Changed in version 3.11: The new `is_package` field indicates whether the module is a package or not. This replaces setting the `size` field to a negative value.

const struct _frozen *PyImport_FrozenModules

This pointer is initialized to point to an array of `_frozen` records, terminated by one whose members are all `NULL` or zero. When a frozen module is imported, it is searched in this table. Third-party code could play tricks with this to provide a dynamically created collection of frozen modules.

int PyImport_AppendInittab(const char *name, PyObject *(*initfunc)(void))

Part of the Stable ABI.

Add a single module to the existing table of built-in modules. This is a convenience wrapper around `PyImport_ExtendInittab()`, returning `-1` if the table could not be extended. The new module can be imported by the name *name*, and uses the function *initfunc* as the initialization function called on the first attempted import. This should be called before `Py_Initialize()`.

struct _inittab

Structure describing a single entry in the list of built-in modules. Programs which embed Python may use an array of these structures in conjunction with `PyImport_ExtendInittab()` to provide additional built-in modules. The structure consists of two members:

const char *name

The module name, as an ASCII encoded string.

PyObject *(*initfunc)(void)

Initialization function for a module built into the interpreter.

int PyImport_ExtendInittab(struct _inittab *newtab)

Add a collection of modules to the table of built-in modules. The *newtab* array must end with a sentinel entry which contains `NULL` for the `name` field; failure to provide the sentinel value can result in a memory fault. Returns `0` on success or `-1` if insufficient memory could be allocated to extend the internal table. In the event of failure, no modules are added to the internal table. This must be called before `Py_Initialize()`.

If Python is initialized multiple times, `PyImport_AppendInittab()` or `PyImport_ExtendInittab()` must be called before each Python initialization.

struct _inittab *PyImport_Inittab

The table of built-in modules used by Python initialization. Do not use this directly; use `PyImport_AppendInittab()` and `PyImport_ExtendInittab()` instead.

PyObject *PyImport_ImportModuleAttr(PyObject *mod_name, PyObject *attr_name)

Return value: New reference.
Import the module *mod_name* and get its attribute *attr_name*.

Names must be Python `str` objects.

Helper function combining `PyImport_Import()` and `PyObject_GetAttr()`. For example, it can raise `ImportError` if the module is not found, and `AttributeError` if the attribute doesn't exist.

Added in version 3.14.

PyObject *PyImport_ImportModuleAttrString(const char *mod_name, const char *attr_name)

Return value: New reference.
Similar to `PyImport_ImportModuleAttr()`, but names are UTF-8 encoded strings instead of Python `str` objects.

Added in version 3.14.

Data marshalling support

These routines allow C code to work with serialized objects using the same data format as the `marshal` module. There are functions to write data into the serialization format, and additional functions that can be used to read the data back. Files used to store marshalled data must be opened in binary mode.

Numeric values are stored with the least significant byte first.

The module supports several versions of the data format; see the `Python module documentation` for details.

Py_MARSHAL_VERSION

The current format version. See `marshal.version`.

void PyMarshal_WriteLongToFile(long value, FILE *file, int version)

Marshal a long integer, *value*, to *file*. This will only write the least-significant 32 bits of *value*; regardless of the size of the native long type. *version* indicates the file format.

This function can fail, in which case it sets the error indicator. Use `PyErr_Occurred()` to check for that.

void PyMarshal_WriteObjectToFile(PyObject *value, FILE *file, int version)

Marshal a Python object, *value*, to *file*. *version* indicates the file format.

This function can fail, in which case it sets the error indicator. Use **`PyErr_Occurred()`** to check for that.

PyObject *PyMarshal_WriteObjectToString(PyObject *value, int version)

Return value: New reference.
Return a bytes object containing the marshalled representation of *value*. *version* indicates the file format.

The following functions allow marshalled values to be read back in.

long PyMarshal_ReadLongFromFile(FILE *file)

Return a C long from the data stream in a FILE* opened for reading. Only a 32-bit value can be read in using this function, regardless of the native size of long.

On error, sets the appropriate exception (**`EOFError`**) and returns `-1`.

int PyMarshal_ReadShortFromFile(FILE *file)

Return a C short from the data stream in a FILE* opened for reading. Only a 16-bit value can be read in using this function, regardless of the native size of short.

On error, sets the appropriate exception (**`EOFError`**) and returns `-1`.

PyObject *PyMarshal_ReadObjectFromFile(FILE *file)

Return value: New reference.
Return a Python object from the data stream in a FILE* opened for reading.

On error, sets the appropriate exception (**`EOFError`**, **`ValueError`** or **`TypeError`**) and returns `NULL`.

PyObject *PyMarshal_ReadLastObjectFromFile(FILE *file)

Return value: New reference.
Return a Python object from the data stream in a FILE* opened for reading. Unlike **`PyMarshal_ReadObjectFromFile()`**, this function assumes that no further objects will be read from the file, allowing it to aggressively load file data into memory so that the de-serialization can operate from data in memory rather than reading a byte at a time from the file. Only use this variant if you are certain that you won’t be reading anything else from the file.

On error, sets the appropriate exception (**`EOFError`**, **`ValueError`** or **`TypeError`**) and returns `NULL`.

PyObject *PyMarshal_ReadObjectFromString(const char *data, Py_ssize_t len)

Return value: New reference.
Return a Python object from the data stream in a byte buffer containing *len* bytes pointed to by *data*.

On error, sets the appropriate exception (**`EOFError`**, **`ValueError`** or **`TypeError`**) and returns `NULL`.

Parsing arguments and building values

These functions are useful when creating your own extension functions and methods. Additional information and examples are available in Extending and Embedding the Python Interpreter.

The first three of these functions described, `PyArg_ParseTuple()`, `PyArg_ParseTupleAndKeywords()`, and `PyArg_Parse()`, all use *format strings* which are used to tell the function about the expected arguments. The format strings use the same syntax for each of these functions.

Parsing arguments

A format string consists of zero or more “format units.” A format unit describes one Python object; it is usually a single character or a parenthesized sequence of format units. With a few exceptions, a format unit that is not a parenthesized sequence normally corresponds to a single address argument to these functions. In the following description, the quoted form is the format unit; the entry in (round) parentheses is the Python object type that matches the format unit; and the entry in [square] brackets is the type of the C variable(s) whose address should be passed.

Strings and buffers

Note

On Python 3.12 and older, the macro **`PY_SSIZE_T_CLEAN`** must be defined before including `Python.h` to use all `#` variants of formats (`s#`, `y#`, etc.) explained below. This is not necessary on Python 3.13 and later.

These formats allow accessing an object as a contiguous chunk of memory. You don't have to provide raw storage for the returned unicode or bytes area.

Unless otherwise stated, buffers are not NUL-terminated.

There are three ways strings and buffers can be converted to C:

- Formats such as `y*` and `s*` fill a `Py_buffer` structure. This locks the underlying buffer so that the caller can subsequently use the buffer even inside a `Py_BEGIN_ALLOW_THREADS` block without the risk of mutable data being resized or destroyed. As a result, **you have to call** `PyBuffer_Release()` after you have finished processing the data (or in any early abort case).

- The `es`, `es#`, `et` and `et#` formats allocate the result buffer. **You have to call** `PyMem_Free()` after you have finished processing the data (or in any early abort case).

- Other formats take a `str` or a read-only bytes-like object, such as `bytes`, and provide a `const char *` pointer to its buffer. In this case the buffer is "borrowed": it is managed by the corresponding Python object, and shares the lifetime of this object. You won't have to release any memory yourself.

  To ensure that the underlying buffer may be safely borrowed, the object's `PyBufferProcs.bf_releasebuffer` field must be `NULL`. This disallows common mutable objects such as `bytearray`, but also some read-only objects such as `memoryview` of `bytes`.

Besides this `bf_releasebuffer` requirement, there is no check to verify whether the input object is immutable (e.g. whether it would honor a request for a writable buffer, or whether another thread can mutate the data).

`s` (`str`) [const char *]

Convert a Unicode object to a C pointer to a character string. A pointer to an existing string is stored in the character pointer variable whose address you pass. The C string is NUL-terminated. The Python string must not contain embedded null code points; if it does, a `ValueError` exception is raised. Unicode objects are converted to C strings using `'utf-8'` encoding. If this conversion fails, a `UnicodeError` is raised.

Note

This format does not accept bytes-like objects. If you want to accept filesystem paths and convert them to C character strings, it is preferable to use the `O&` format with `PyUnicode_FSConverter()` as *converter*.

Changed in version 3.5: Previously, `TypeError` was raised when embedded null code points were encountered in the Python string.

`s*` (`str` or bytes-like object) [Py_buffer]

This format accepts Unicode objects as well as bytes-like objects. It fills a `Py_buffer` structure provided by the caller. In this case the resulting C string may contain embedded NUL bytes. Unicode objects are converted to C strings using `'utf-8'` encoding.

`s#` (**`str`**, read-only bytes-like object) [const char *, **`Py_ssize_t`**]

Like `s*`, except that it provides a borrowed buffer. The result is stored into two C variables, the first one a pointer to a C string, the second one its length. The string may contain embedded null bytes. Unicode objects are converted to C strings using `'utf-8'` encoding.

`z` (**`str`** or `None`) [const char *]

Like `s`, but the Python object may also be `None`, in which case the C pointer is set to `NULL`.

`z*` (**`str`**, bytes-like object or `None`) [Py_buffer]

Like `s*`, but the Python object may also be `None`, in which case the `buf` member of the **`Py_buffer`** structure is set to `NULL`.

`z#` (**`str`**, read-only bytes-like object or `None`) [const char *, **`Py_ssize_t`**]

Like `s#`, but the Python object may also be `None`, in which case the C pointer is set to `NULL`.

`y` (read-only bytes-like object) [const char *]

This format converts a bytes-like object to a C pointer to a borrowed character string; it does not accept Unicode objects. The bytes buffer must not contain embedded null bytes; if it does, a **`ValueError`** exception is raised.

Changed in version 3.5: Previously, **`TypeError`** was raised when embedded null bytes were encountered in the bytes buffer.

`y*` (bytes-like object) [Py_buffer]

This variant on `s*` doesn’t accept Unicode objects, only bytes-like objects. **This is the recommended way to accept binary data.**

`y#` (read-only bytes-like object) [const char *, `Py_ssize_t`]

This variant on `s#` doesn't accept Unicode objects, only bytes-like objects.

`S` (`bytes`) [PyBytesObject *]

Requires that the Python object is a `bytes` object, without attempting any conversion. Raises `TypeError` if the object is not a bytes object. The C variable may also be declared as PyObject*.

`Y` (`bytearray`) [PyByteArrayObject *]

Requires that the Python object is a `bytearray` object, without attempting any conversion. Raises `TypeError` if the object is not a `bytearray` object. The C variable may also be declared as PyObject*.

`U` (`str`) [PyObject *]

Requires that the Python object is a Unicode object, without attempting any conversion. Raises `TypeError` if the object is not a Unicode object. The C variable may also be declared as PyObject*.

`w*` (read-write bytes-like object) [Py_buffer]

This format accepts any object which implements the read-write buffer interface. It fills a `Py_buffer` structure provided by the caller. The buffer may contain embedded null bytes. The caller has to call `PyBuffer_Release()` when it is done with the buffer.

`es` (`str`) [const char *encoding, char **buffer]

This variant on `s` is used for encoding Unicode into a character buffer. It only works for encoded data without embedded NUL bytes.

This format requires two arguments. The first is only used as input, and must be a const char* which points to the name of an encoding as a NUL-terminated string, or `NULL`, in which case `'utf-8'` encoding is used. An exception is raised if the named encoding is not known to Python. The second argument must be a char**; the value of the pointer it references will be set to a buffer with the contents of the argument text. The text will be encoded in the encoding specified by the first argument.

`PyArg_ParseTuple()` will allocate a buffer of the needed size, copy the encoded data into this buffer and adjust **buffer* to reference the newly allocated storage. The caller is responsible for calling `PyMem_Free()` to free the allocated buffer after use.

`et` (`str`, `bytes` or `bytearray`) [const char *encoding, char **buffer]

Same as `es` except that byte string objects are passed through without recoding them. Instead, the implementation assumes that the byte string object uses the encoding passed in as parameter.

`es#` (`str`) [const char *encoding, char **buffer, `Py_ssize_t` *buffer_length]

This variant on `s#` is used for encoding Unicode into a character buffer. Unlike the `es` format, this variant allows input data which contains NUL characters.

It requires three arguments. The first is only used as input, and must be a const char* which points to the name of an encoding as a NUL-terminated string, or `NULL`, in which case `'utf-8'` encoding is used. An exception is raised if the named encoding is not known to Python. The second argument must be a char**; the value of the pointer it references will be set to a buffer with the contents of the argument text. The text will be encoded in the encoding specified by the first argument. The third argument must be a pointer to an integer; the referenced integer will be set to the number of bytes in the output buffer.

There are two modes of operation:

If **buffer* points a `NULL` pointer, the function will allocate a buffer of the needed size, copy the encoded data into this buffer and set **buffer* to reference the newly allocated storage. The caller is responsible for calling `PyMem_Free()` to free the allocated buffer after usage.

If **buffer* points to a non-`NULL` pointer (an already allocated buffer), `PyArg_ParseTuple()` will use this location as the buffer and interpret the initial value of **buffer_length* as the buffer size. It will then copy the encoded data into the buffer and NUL-terminate it. If the buffer is not large enough, a `ValueError` will be set.

In both cases, **buffer_length* is set to the length of the encoded data without the trailing NUL byte.

`et#` (`str`, `bytes` or `bytearray`) [const char *encoding, char **buffer, `Py_ssize_t` *buffer_length]

Same as `es#` except that byte string objects are passed through without recoding them. Instead, the implementation assumes that the byte string object uses the encoding passed in as parameter.

Changed in version 3.12: `u`, `u#`, `Z`, and `Z#` are removed because they used a legacy `Py_UNICODE*` representation.

Numbers

These formats allow representing Python numbers or single characters as C numbers. Formats that require `int`, `float` or `complex` can also use the corresponding special methods `__index__()`, `__float__()` or `__complex__()` to convert the Python object to the required type.

For signed integer formats, `OverflowError` is raised if the value is out of range for the C type. For unsigned integer formats, no range checking is done — the most significant bits are silently truncated when the receiving field is too small to receive the value.

`b` (`int`) [unsigned char]
: Convert a nonnegative Python integer to an unsigned tiny integer, stored in a C unsigned char.

`B` (`int`) [unsigned char]
: Convert a Python integer to a tiny integer without overflow checking, stored in a C unsigned char.

`h` (`int`) [short int]
: Convert a Python integer to a C short int.

`H` (`int`) [unsigned short int]
: Convert a Python integer to a C unsigned short int, without overflow checking.

`i` (`int`) [int]
: Convert a Python integer to a plain C int.

`I` (`int`) [unsigned int]
: Convert a Python integer to a C unsigned int, without overflow checking.

`l` (`int`) [long int]
: Convert a Python integer to a C long int.

`k` (`int`) [unsigned long]

Convert a Python integer to a C unsigned long without overflow checking.

Changed in version 3.14: Use `__index__()` if available.

`L` (`int`) [long long]
Convert a Python integer to a C long long.

`K` (`int`) [unsigned long long]
Convert a Python integer to a C unsigned long long without overflow checking.

Changed in version 3.14: Use `__index__()` if available.

`n` (`int`) [`Py_ssize_t`]
Convert a Python integer to a C `Py_ssize_t`.

`c` (`bytes` or `bytearray` of length 1) [char]
Convert a Python byte, represented as a `bytes` or `bytearray` object of length 1, to a C char.

Changed in version 3.3: Allow `bytearray` objects.

`C` (`str` of length 1) [int]
Convert a Python character, represented as a `str` object of length 1, to a C int.

`f` (`float`) [float]
Convert a Python floating-point number to a C float.

`d` (`float`) [double]
Convert a Python floating-point number to a C double.

`D` (`complex`) [Py_complex]
Convert a Python complex number to a C `Py_complex` structure.

Other objects

`O` (object) [PyObject *]

Store a Python object (without any conversion) in a C object pointer. The C program thus receives the actual object that was passed. A new strong reference to the object is not created (i.e. its reference count is not increased). The pointer stored is not `NULL`.

`O!` (object) [*typeobject*, PyObject *]

Store a Python object in a C object pointer. This is similar to `O`, but takes two C arguments: the first is the address of a Python type object, the second is the address of the C variable (of type PyObject*) into which the object pointer is stored. If the Python object does not have the required type, **`TypeError`** is raised.

`O&` (object) [*converter*, *address*]

Convert a Python object to a C variable through a *converter* function. This takes two arguments: the first is a function, the second is the address of a C variable (of arbitrary type), converted to void*. The *converter* function in turn is called as follows:

```
status = converter(object, address);
```

where *object* is the Python object to be converted and *address* is the void* argument that was passed to the `PyArg_Parse*` function. The returned *status* should be `1` for a successful conversion and `0` if the conversion has failed. When the conversion fails, the *converter* function should raise an exception and leave the content of *address* unmodified.

If the *converter* returns **`Py_CLEANUP_SUPPORTED`**, it may get called a second time if the argument parsing eventually fails, giving the converter a chance to release any memory that it had already allocated. In this second call, the *object* parameter will be `NULL`; *address* will have the same value as in the original call.

Examples of converters: **`PyUnicode_FSConverter()`** and **`PyUnicode_FSDecoder()`**.

Changed in version 3.1: **`Py_CLEANUP_SUPPORTED`** was added.

`p` (**`bool`**) [int]

Tests the value passed in for truth (a boolean **p**redicate) and converts the result to its equivalent C true/false integer value. Sets the int to `1` if the expression was true and `0` if it was false. This accepts any valid Python value. See Truth Value Testing for more information about how Python tests values for truth.

Added in version 3.3.

`(items)` (sequence) [*matching-items*]

The object must be a Python sequence (except **`str`**, **`bytes`** or **`bytearray`**) whose length is the number of format units in *items*. The C arguments must correspond to the individual format units in *items*. Format units for sequences may be nested.

If *items* contains format units which store a borrowed buffer (`s`, `s#`, `z`, `z#`, `y`, or `y#`) or a borrowed reference (`S`, `Y`, `U`, `O`, or `O!`), the object must be a Python tuple. The *converter* for the `O&` format unit in *items* must not store a borrowed buffer or a borrowed reference.

Changed in version 3.14: **`str`** and **`bytearray`** objects no longer accepted as a sequence.

Deprecated since version 3.14: Non-tuple sequences are deprecated if *items* contains format units which store a borrowed buffer or a borrowed reference.

A few other characters have a meaning in a format string. These may not occur inside nested parentheses. They are:

|

Indicates that the remaining arguments in the Python argument list are optional. The C variables corresponding to optional arguments should be initialized to their default value — when an optional argument is not specified, `PyArg_ParseTuple()` does not touch the contents of the corresponding C variable(s).

$

`PyArg_ParseTupleAndKeywords()` only: Indicates that the remaining arguments in the Python argument list are keyword-only. Currently, all keyword-only arguments must also be optional arguments, so `|` must always be specified before `$` in the format string.

Added in version 3.3.

:

The list of format units ends here; the string after the colon is used as the function name in error messages (the “associated value” of the exception that `PyArg_ParseTuple()` raises).

;

The list of format units ends here; the string after the semicolon is used as the error message *instead* of the default error message. `:` and `;` mutually exclude each other.

Note that any Python object references which are provided to the caller are *borrowed* references; do not release them (i.e. do not decrement their reference count)!

Additional arguments passed to these functions must be addresses of variables whose type is determined by the format string; these are used to store values from the input tuple. There are a few cases, as described in the list of format units above, where these parameters are used as input values; they should match what is specified for the corresponding format unit in that case.

For the conversion to succeed, the *arg* object must match the format and the format must be exhausted. On success, the `PyArg_Parse*` functions return true, otherwise they return false and raise an appropriate exception. When the `PyArg_Parse*` functions fail due to conversion failure in one of the format units, the variables at the addresses corresponding to that and the following format units are left untouched.

API Functions

int PyArg_ParseTuple(PyObject *args, const char *format, ...)

Part of the Stable ABI.

Parse the parameters of a function that takes only positional parameters into local variables. Returns true on success; on failure, it returns false and raises the appropriate exception.

int PyArg_VaParse(PyObject *args, const char *format, va_list vargs)

Part of the Stable ABI.

Identical to **`PyArg_ParseTuple()`**, except that it accepts a va_list rather than a variable number of arguments.

int PyArg_ParseTupleAndKeywords(PyObject *args, PyObject *kw, const char *format, char *const *keywords, ...)

Part of the Stable ABI.
Parse the parameters of a function that takes both positional and keyword parameters into local variables. The *keywords* argument is a `NULL`-terminated array of keyword parameter names specified as null-terminated ASCII or UTF-8 encoded C strings. Empty names denote positional-only parameters. Returns true on success; on failure, it returns false and raises the appropriate exception.

Note

The *keywords* parameter declaration is char *const* in C and const char *const* in C++. This can be overridden with the **`PY_CXX_CONST`** macro.

Changed in version 3.6: Added support for positional-only parameters.

Changed in version 3.13: The *keywords* parameter has now type char *const* in C and const char *const* in C++, instead of char**. Added support for non-ASCII keyword parameter names.

int PyArg_VaParseTupleAndKeywords(PyObject *args, PyObject *kw, const char *format, char *const *keywords, va_list vargs)

Part of the Stable ABI.
Identical to **`PyArg_ParseTupleAndKeywords()`**, except that it accepts a va_list rather than a variable number of arguments.

int PyArg_ValidateKeywordArguments(PyObject*)

Part of the Stable ABI.
Ensure that the keys in the keywords argument dictionary are strings. This is only needed if **`PyArg_ParseTupleAndKeywords()`** is not used, since the latter already does this check.

Added in version 3.2.

int PyArg_Parse(PyObject *args, const char *format, ...)

Part of the Stable ABI.
Parse the parameter of a function that takes a single positional parameter into a local variable. Returns true on success; on failure, it returns false and raises the appropriate exception.

Example:

```
// Function using METH_O calling convention
static PyObject*
my_function(PyObject *module, PyObject *arg)
{
    int value;
    if (!PyArg_Parse(arg, "i:my_function", &value)) {
        return NULL;
    }
    // ... use value ...
}
```

int PyArg_UnpackTuple(PyObject *args, const char *name, Py_ssize_t min, Py_ssize_t max, ...)

Part of the Stable ABI.
A simpler form of parameter retrieval which does not use a format string to specify the types of the arguments. Functions which use this method to retrieve their parameters should be declared as **`METH_VARARGS`** in function or method tables. The tuple containing the actual parameters should be passed as *args*; it must actually be

a tuple. The length of the tuple must be at least *min* and no more than *max*; *min* and *max* may be equal. Additional arguments must be passed to the function, each of which should be a pointer to a PyObject* variable; these will be filled in with the values from *args*; they will contain borrowed references. The variables which correspond to optional parameters not given by *args* will not be filled in; these should be initialized by the caller. This function returns true on success and false if *args* is not a tuple or contains the wrong number of elements; an exception will be set if there was a failure.

This is an example of the use of this function, taken from the sources for the **`_weakref`** helper module for weak references:

```
static PyObject *
weakref_ref(PyObject *self, PyObject *args)
{
    PyObject *object;
    PyObject *callback = NULL;
    PyObject *result = NULL;

    if (PyArg_UnpackTuple(args, "ref", 1, 2, &object,
&callback)) {
        result = PyWeakref_NewRef(object, callback);
    }
    return result;
}
```

The call to **`PyArg_UnpackTuple()`** in this example is entirely equivalent to this call to **`PyArg_ParseTuple()`**:

```
PyArg_ParseTuple(args, "O|O:ref", &object, &callback)
```

PY_CXX_CONST

The value to be inserted, if any, before char *const* in the *keywords* parameter declaration of **`PyArg_ParseTupleAndKeywords()`** and

`PyArg_VaParseTupleAndKeywords()`. Default empty for C and `const` for C++ (const char *const*). To override, define it to the desired value before including `Python.h`.

Added in version 3.13.

Building values

PyObject *Py_BuildValue(const char *format, ...)

Return value: New reference. Part of the Stable ABI. Create a new value based on a format string similar to those accepted by the `PyArg_Parse*` family of functions and a sequence of values. Returns the value or `NULL` in the case of an error; an exception will be raised if `NULL` is returned.

`Py_BuildValue()` does not always build a tuple. It builds a tuple only if its format string contains two or more format units. If the format string is empty, it returns `None`; if it contains exactly one format unit, it returns whatever object is described by that format unit. To force it to return a tuple of size 0 or one, parenthesize the format string.

When memory buffers are passed as parameters to supply data to build objects, as for the `s` and `s#` formats, the required data is copied. Buffers provided by the caller are never referenced by the objects created by **`Py_BuildValue()`**. In other words, if your code invokes **`malloc()`** and passes the allocated memory to **`Py_BuildValue()`**, your code is responsible for calling **`free()`** for that memory once **`Py_BuildValue()`** returns.

In the following description, the quoted form is the format unit; the entry in (round) parentheses is the Python object type that the format unit will return; and the entry in [square] brackets is the type of the C value(s) to be passed.

The characters space, tab, colon and comma are ignored in format strings (but not within format units such as `s#`). This can be used to make long format strings a tad more readable.

`s` (**`str`** or `None`) [const char *]
: Convert a null-terminated C string to a Python **`str`** object using `'utf-8'` encoding. If the C string pointer is `NULL`, `None` is used.

`s#` (**`str`** or `None`) [const char *, **`Py_ssize_t`**]
: Convert a C string and its length to a Python **`str`** object using `'utf-8'` encoding. If the C string pointer is `NULL`, the length is ignored and `None` is returned.

`y` (**`bytes`**) [const char *]
: This converts a C string to a Python **`bytes`** object. If the C string pointer is `NULL`, `None` is returned.

`y#` (**`bytes`**) [const char *, **`Py_ssize_t`**]
: This converts a C string and its lengths to a Python object. If the C string pointer is `NULL`, `None` is returned.

`z` (**`str`** or `None`) [const char *]
: Same as `s`.

`z#` (**`str`** or `None`) [const char *, **`Py_ssize_t`**]
: Same as `s#`.

`u` (**`str`**) [const wchar_t *]
: Convert a null-terminated **`wchar_t`** buffer of Unicode (UTF-16 or UCS-4) data to a Python Unicode object. If the Unicode buffer pointer is `NULL`, `None` is returned.

`u#` (str) [const wchar_t *, Py_ssize_t]
Convert a Unicode (UTF-16 or UCS-4) data buffer and its length to a Python Unicode object. If the Unicode buffer pointer is `NULL`, the length is ignored and `None` is returned.

`U` (str or `None`) [const char *]
Same as `s`.

`U#` (str or `None`) [const char *, Py_ssize_t]
Same as `s#`.

`i` (int) [int]
Convert a plain C int to a Python integer object.

`b` (int) [char]
Convert a plain C char to a Python integer object.

`h` (int) [short int]
Convert a plain C short int to a Python integer object.

`l` (int) [long int]
Convert a C long int to a Python integer object.

`B` (int) [unsigned char]
Convert a C unsigned char to a Python integer object.

`H` (int) [unsigned short int]
Convert a C unsigned short int to a Python integer object.

`I` (int) [unsigned int]
Convert a C unsigned int to a Python integer object.

`k` (int) [unsigned long]
Convert a C unsigned long to a Python integer object.

`L` (**int**) [long long]
: Convert a C long long to a Python integer object.

`K` (**int**) [unsigned long long]
: Convert a C unsigned long long to a Python integer object.

`n` (**int**) [**Py_ssize_t**]
: Convert a C **Py_ssize_t** to a Python integer.

`p` (**bool**) [int]
: Convert a C int to a Python **bool** object.

Be aware that this format requires an `int` argument. Unlike most other contexts in C, variadic arguments are not coerced to a suitable type automatically. You can convert another type (for example, a pointer or a float) to a suitable `int` value using `(x) ? 1 : 0` or `!!x`.

Added in version 3.14.

`c` (**bytes** of length 1) [char]
: Convert a C int representing a byte to a Python **bytes** object of length 1.

`C` (**str** of length 1) [int]
: Convert a C int representing a character to Python **str** object of length 1.

`d` (**float**) [double]
: Convert a C double to a Python floating-point number.

`f` (**float**) [float]
: Convert a C float to a Python floating-point number.

`D` (**complex**) [Py_complex *]

Convert a C **`Py_complex`** structure to a Python complex number.

`O` (object) [PyObject *]
: Pass a Python object untouched but create a new strong reference to it (i.e. its reference count is incremented by one). If the object passed in is a `NULL` pointer, it is assumed that this was caused because the call producing the argument found an error and set an exception. Therefore, **`Py_BuildValue()`** will return `NULL` but won't raise an exception. If no exception has been raised yet, **`SystemError`** is set.

`S` (object) [PyObject *]
: Same as `O`.

`N` (object) [PyObject *]
: Same as `O`, except it doesn't create a new strong reference. Useful when the object is created by a call to an object constructor in the argument list.

`O&` (object) [*converter*, *anything*]
: Convert *anything* to a Python object through a *converter* function. The function is called with *anything* (which should be compatible with void*) as its argument and should return a "new" Python object, or `NULL` if an error occurred.

`(items)` (**`tuple`**) [*matching-items*]
: Convert a sequence of C values to a Python tuple with the same number of items.

`[items]` (**`list`**) [*matching-items*]
: Convert a sequence of C values to a Python list with the same number of items.

`{items}` (`dict`) [*matching-items*]

Convert a sequence of C values to a Python dictionary. Each pair of consecutive C values adds one item to the dictionary, serving as key and value, respectively.

If there is an error in the format string, the `SystemError` exception is set and `NULL` returned.

PyObject *Py_VaBuildValue(const char *format, va_list vargs)

Return value: New reference. Part of the Stable ABI. Identical to `Py_BuildValue()`, except that it accepts a va_list rather than a variable number of arguments.

String conversion and formatting

Functions for number conversion and formatted string output.

int PyOS_snprintf(char *str, size_t size, const char *format, ...)

Part of the Stable ABI.
Output not more than *size* bytes to *str* according to the format string *format* and the extra arguments. See the Unix man page *snprintf(3)* *[https://manpages.debian.org/snprintf(3)]*.

int PyOS_vsnprintf(char *str, size_t size, const char *format, va_list va)

Part of the Stable ABI.
Output not more than *size* bytes to *str* according to the format string *format* and the variable argument list *va*. Unix man page *vsnprintf(3)* *[https://manpages.debian.org/vsnprintf(3)]*.

`PyOS_snprintf()` and **`PyOS_vsnprintf()`** wrap the Standard C library functions **`snprintf()`** and **`vsnprintf()`**. Their purpose is to guarantee consistent behavior in corner cases, which the Standard C functions do not.

The wrappers ensure that `str[size-1]` is always `'\0'` upon return. They never write more than *size* bytes (including the trailing `'\0'`) into str. Both functions require that `str != NULL`, `size > 0`, `format != NULL` and `size < INT_MAX`. Note that this means there is no equivalent to the C99 `n = snprintf(NULL, 0, ...)` which would determine the necessary buffer size.

The return value (*rv*) for these functions should be interpreted as follows:

- When `0 <= rv < size`, the output conversion was successful and *rv* characters were written to *str* (excluding the trailing `'\0'` byte at `str[rv]`).
- When `rv >= size`, the output conversion was truncated and a buffer with `rv + 1` bytes would have been needed to succeed. `str[size-1]` is `'\0'` in this case.
- When `rv < 0`, the output conversion failed and `str[size-1]` is `'\0'` in this case too, but the rest of *str* is undefined. The exact cause of the error depends on the underlying platform.

The following functions provide locale-independent string to number conversions.

unsigned long PyOS_strtoul(const char *str, char **ptr, int base)

Part of the Stable ABI.
Convert the initial part of the string in `str` to an unsigned long value according to the given `base`, which must be between `2` and `36` inclusive, or be the special value `0`.

Leading white space and case of characters are ignored. If `base` is zero it looks for a leading `0b`, `0o` or `0x` to tell which base. If these are absent it defaults to `10`. Base must be 0 or between 2 and 36 (inclusive). If `ptr` is non-`NULL` it will contain a pointer to the end of the scan.

If the converted value falls out of range of corresponding return type, range error occurs (**`errno`** is set to **`ERANGE`**) and

`ULONG_MAX` is returned. If no conversion can be performed, `0` is returned.

See also the Unix man page *strtoul(3)* *[https://manpages.debian.org/strtoul(3)]*.

Added in version 3.2.

long PyOS_strtol(const char *str, char **ptr, int base)

Part of the Stable ABI.
Convert the initial part of the string in `str` to an long value according to the given `base`, which must be between `2` and `36` inclusive, or be the special value `0`.

Same as **`PyOS_strtoul()`**, but return a long value instead and **`LONG_MAX`** on overflows.

See also the Unix man page *strtol(3)* *[https://manpages.debian.org/strtol(3)]*.

Added in version 3.2.

double PyOS_string_to_double(const char *s, char **endptr, PyObject *overflow_exception)

Part of the Stable ABI.
Convert a string `s` to a double, raising a Python exception on failure. The set of accepted strings corresponds to the set of strings accepted by Python's **`float()`** constructor, except that `s` must not have leading or trailing whitespace. The conversion is independent of the current locale.

If `endptr` is `NULL`, convert the whole string. Raise **`ValueError`** and return `-1.0` if the string is not a valid representation of a floating-point number.

If endptr is not `NULL`, convert as much of the string as possible and set `*endptr` to point to the first unconverted character. If no initial segment of the string is the valid representation of a floating-point number, set `*endptr` to point to the beginning of the string, raise ValueError, and return `-1.0`.

If `s` represents a value that is too large to store in a float (for example, `"1e500"` is such a string on many platforms) then if `overflow_exception` is `NULL` return `Py_INFINITY` (with an appropriate sign) and don’t set any exception. Otherwise, `overflow_exception` must point to a Python exception object; raise that exception and return `-1.0`. In both cases, set `*endptr` to point to the first character after the converted value.

If any other error occurs during the conversion (for example an out-of-memory error), set the appropriate Python exception and return `-1.0`.

Added in version 3.1.

char *PyOS_double_to_string(double val, char format_code, int precision, int flags, int *ptype)

Part of the Stable ABI.
Convert a double *val* to a string using supplied *format_code*, *precision*, and *flags*.

format_code must be one of `'e'`, `'E'`, `'f'`, `'F'`, `'g'`, `'G'` or `'r'`. For `'r'`, the supplied *precision* must be 0 and is ignored. The `'r'` format code specifies the standard **`repr()`** format.

flags can be zero or more of the following values or-ed together:

Py_DTSF_SIGN

Always precede the returned string with a sign character, even if *val* is non-negative.

Py_DTSF_ADD_DOT_0

Ensure that the returned string will not look like an integer.

Py_DTSF_ALT

Apply “alternate” formatting rules. See the documentation for the **`PyOS_snprintf()`** `'#'` specifier for details.

Py_DTSF_NO_NEG_0

Negative zero is converted to positive zero.

Added in version 3.11.

If *ptype* is non-`NULL`, then the value it points to will be set to one of `Py_DTST_FINITE`, `Py_DTST_INFINITE`, or `Py_DTST_NAN`, signifying that *val* is a finite number, an infinite number, or not a number, respectively.

The return value is a pointer to *buffer* with the converted string or `NULL` if the conversion failed. The caller is responsible for freeing the returned string by calling **`PyMem_Free()`**.

Added in version 3.1.

int PyOS_mystricmp(const char *str1, const char *str2)
int PyOS_mystrnicmp(const char *str1, const char *str2, Py_ssize_t size)

Part of the Stable ABI.
Case insensitive comparison of strings. These functions work almost identically to **`strcmp()`** and **`strncmp()`**

(respectively), except that they ignore the case of ASCII characters.

Return `0` if the strings are equal, a negative value if *str1* sorts lexicographically before *str2*, or a positive value if it sorts after.

In the *str1* or *str2* arguments, a NUL byte marks the end of the string. For **`PyOS_mystrnicmp()`**, the *size* argument gives the maximum size of the string, as if NUL was present at the index given by *size*.

These functions do not use the locale.

int PyOS_stricmp(const char *str1, const char *str2)
int PyOS_strnicmp(const char *str1, const char *str2, Py_ssize_t size)

Case insensitive comparison of strings.

On Windows, these are aliases of **`stricmp()`** and **`strnicmp()`**, respectively.

On other platforms, they are aliases of **`PyOS_mystricmp()`** and **`PyOS_mystrnicmp()`**, respectively.

Character classification and conversion

The following macros provide locale-independent (unlike the C standard library `ctype.h`) character classification and conversion. The argument must be a signed or unsigned char.

Py_ISALNUM(c)

Return true if the character *c* is an alphanumeric character.

Py_ISALPHA(c)

Return true if the character *c* is an alphabetic character (`a-z` and `A-Z`).

Py_ISDIGIT(c)

Return true if the character *c* is a decimal digit (`0-9`).

Py_ISLOWER(c)

Return true if the character *c* is a lowercase ASCII letter (`a-z`).

Py_ISUPPER(c)

Return true if the character *c* is an uppercase ASCII letter (`A-Z`).

Py_ISSPACE(c)

Return true if the character *c* is a whitespace character (space, tab, carriage return, newline, vertical tab, or form feed).

Py_ISXDIGIT(c)

Return true if the character *c* is a hexadecimal digit (`0-9`, `a-f`, and `A-F`).

Py_TOLOWER(c)

Return the lowercase equivalent of the character *c*.

Py_TOUPPER(c)

Return the uppercase equivalent of the character *c*.

PyHash API

See also the `PyTypeObject.tp_hash` member and Hashing of numeric types.

type Py_hash_t

Hash value type: signed integer.

Added in version 3.2.

type Py_uhash_t

Hash value type: unsigned integer.

Added in version 3.2.

Py_HASH_ALGORITHM

A numerical value indicating the algorithm for hashing of `str`, `bytes`, and `memoryview`.

The algorithm name is exposed by `sys.hash_info.algorithm`.

Added in version 3.4.

Py_HASH_FNV
Py_HASH_SIPHASH24
Py_HASH_SIPHASH13

Numerical values to compare to `Py_HASH_ALGORITHM` to determine which algorithm is used for hashing. The hash algorithm can be configured via the configure `--with-hash-algorithm` option.

Added in version 3.4: Add **`Py_HASH_FNV`** and **`Py_HASH_SIPHASH24`**.

Added in version 3.11: Add **`Py_HASH_SIPHASH13`**.

Py_HASH_CUTOFF

Buffers of length in range `[1, Py_HASH_CUTOFF)` are hashed using DJBX33A instead of the algorithm described by **`Py_HASH_ALGORITHM`**.

- A **`Py_HASH_CUTOFF`** of 0 disables the optimization.
- **`Py_HASH_CUTOFF`** must be non-negative and less or equal than 7.

32-bit platforms should use a cutoff smaller than 64-bit platforms because it is easier to create colliding strings. A cutoff of 7 on 64-bit platforms and 5 on 32-bit platforms should provide a decent safety margin.

This corresponds to the **`sys.hash_info.cutoff`** constant.

Added in version 3.4.

PyHASH_MODULUS

The Mersenne prime [https://en.wikipedia.org/wiki/Mersenne_prime] `P = 2**n -1`, used for numeric hash scheme.

This corresponds to the **`sys.hash_info.modulus`** constant.

Added in version 3.13.

PyHASH_BITS

The exponent `n` of `P` in **`PyHASH_MODULUS`**.

Added in version 3.13.

PyHASH_MULTIPLIER

Prime multiplier used in string and various other hashes.

Added in version 3.13.

PyHASH_INF

The hash value returned for a positive infinity.

This corresponds to the `sys.hash_info.inf` constant.

Added in version 3.13.

PyHASH_IMAG

The multiplier used for the imaginary part of a complex number.

This corresponds to the `sys.hash_info.imag` constant.

Added in version 3.13.

type PyHash_FuncDef

Hash function definition used by `PyHash_GetFuncDef()`.

Py_hash_t (*const hash)(const void*, Py_ssize_t)

Hash function.

const char *name

Hash function name (UTF-8 encoded string).

This corresponds to the `sys.hash_info.algorithm` constant.

const int hash_bits

Internal size of the hash value in bits.

This corresponds to the `sys.hash_info.hash_bits` constant.

const int seed_bits

Size of seed input in bits.

This corresponds to the `sys.hash_info.seed_bits` constant.

Added in version 3.4.

PyHash_FuncDef *PyHash_GetFuncDef(void)

Get the hash function definition.

See also

PEP 456 [https://peps.python.org/pep-0456/] “Secure and interchangeable hash algorithm”.

Added in version 3.4.

Py_hash_t Py_HashPointer(const void *ptr)

Hash a pointer value: process the pointer value as an integer (cast it to `uintptr_t` internally). The pointer is not dereferenced.

The function cannot fail: it cannot return `-1`.

Added in version 3.13.

Py_hash_t Py_HashBuffer(const void *ptr, Py_ssize_t len)

Compute and return the hash value of a buffer of *len* bytes starting at address *ptr*. The hash is guaranteed to match that of `bytes`, `memoryview`, and other built-in objects that implement the buffer protocol.

Use this function to implement hashing for immutable objects whose `tp_richcompare` function compares to another object’s buffer.

len must be greater than or equal to `0`.

This function always succeeds.

Added in version 3.14.

Py_hash_t PyObject_GenericHash(PyObject *obj)

Generic hashing function that is meant to be put into a type object's `tp_hash` slot. Its result only depends on the object's identity.

CPython implementation detail: In CPython, it is equivalent to **`Py_HashPointer()`**.

Added in version 3.13.

Reflection

PyObject *PyEval_GetBuiltins(void)

Return value: Borrowed reference. Part of the Stable ABI.
Deprecated since version 3.13: Use `PyEval_GetFrameBuiltins()` instead.

Return a dictionary of the builtins in the current execution frame, or the interpreter of the thread state if no frame is currently executing.

PyObject *PyEval_GetLocals(void)

Return value: Borrowed reference. Part of the Stable ABI.
Deprecated since version 3.13: Use either `PyEval_GetFrameLocals()` to obtain the same behaviour as calling `locals()` in Python code, or else call `PyFrame_GetLocals()` on the result of `PyEval_GetFrame()` to access the `f_locals` attribute of the currently executing frame.

Return a mapping providing access to the local variables in the current execution frame, or `NULL` if no frame is currently executing.

Refer to `locals()` for details of the mapping returned at different scopes.

As this function returns a borrowed reference, the dictionary returned for optimized scopes is cached on the frame object and will remain alive as long as the frame object does. Unlike `PyEval_GetFrameLocals()` and `locals()`, subsequent calls to this function in the same frame will update the contents of the cached dictionary to reflect changes in the state of the local variables rather than returning a new snapshot.

Changed in version 3.13: As part of **PEP 667** [https://peps.python.org/pep-0667/], **`PyFrame_GetLocals()`**, **`locals()`**, and **`FrameType.f_locals`** no longer make use of the shared cache dictionary. Refer to the What's New entry for additional details.

PyObject *PyEval_GetGlobals(void)

Return value: Borrowed reference. Part of the Stable ABI.
Deprecated since version 3.13: Use **`PyEval_GetFrameGlobals()`** instead.

Return a dictionary of the global variables in the current execution frame, or `NULL` if no frame is currently executing.

PyFrameObject *PyEval_GetFrame(void)

Return value: Borrowed reference. Part of the Stable ABI.
Return the attached thread state's frame, which is `NULL` if no frame is currently executing.

See also **`PyThreadState_GetFrame()`**.

PyObject *PyEval_GetFrameBuiltins(void)

Return value: New reference. Part of the Stable ABI since version 3.13.
Return a dictionary of the builtins in the current execution frame, or the interpreter of the thread state if no frame is currently executing.

Added in version 3.13.

PyObject *PyEval_GetFrameLocals(void)

Return value: New reference. Part of the Stable ABI since version 3.13.
Return a dictionary of the local variables in the current execution frame, or `NULL` if no frame is currently executing. Equivalent to calling **`locals()`** in Python code.

To access **`f_locals`** on the current frame without making an independent snapshot in optimized scopes, call **`PyFrame_GetLocals()`** on the result of **`PyEval_GetFrame()`**.

Added in version 3.13.

PyObject *PyEval_GetFrameGlobals(void)

Return value: New reference. Part of the Stable ABI since version 3.13.
Return a dictionary of the global variables in the current execution frame, or `NULL` if no frame is currently executing. Equivalent to calling **`globals()`** in Python code.

Added in version 3.13.

const char *PyEval_GetFuncName(PyObject *func)

Part of the Stable ABI.
Return the name of *func* if it is a function, class or instance object, else the name of *func*s type.

const char *PyEval_GetFuncDesc(PyObject *func)

Part of the Stable ABI.
Return a description string, depending on the type of *func*. Return values include “()” for functions and methods, “ constructor”, “ instance”, and “ object”. Concatenated with the result of **`PyEval_GetFuncName()`**, the result will be a description of *func*.

Codec registry and support functions

int PyCodec_Register(PyObject *search_function)

Part of the Stable ABI.

Register a new codec search function.

As a side effect, this tries to load the `encodings` package, if not yet done, to make sure that it is always first in the list of search functions.

int PyCodec_Unregister(PyObject *search_function)

Part of the Stable ABI since version 3.10.

Unregister a codec search function and clear the registry’s cache. If the search function is not registered, do nothing. Return 0 on success. Raise an exception and return -1 on error.

Added in version 3.10.

int PyCodec_KnownEncoding(const char *encoding)

Part of the Stable ABI.

Return `1` or `0` depending on whether there is a registered codec for the given *encoding*. This function always succeeds.

PyObject *PyCodec_Encode(PyObject *object, const char *encoding, const char *errors)

Return value: New reference. Part of the Stable ABI.

Generic codec based encoding API.

object is passed through the encoder function found for the given *encoding* using the error handling method defined by *errors*. *errors* may be `NULL` to use the default method defined for the codec. Raises a **`LookupError`** if no encoder can be found.

PyObject *PyCodec_Decode(PyObject *object, const char *encoding, const char *errors)

Return value: New reference. Part of the Stable ABI.
Generic codec based decoding API.

object is passed through the decoder function found for the given *encoding* using the error handling method defined by *errors*. *errors* may be `NULL` to use the default method defined for the codec. Raises a **`LookupError`** if no decoder can be found.

Codec lookup API

In the following functions, the *encoding* string is looked up converted to all lower-case characters, which makes encodings looked up through this mechanism effectively case-insensitive. If no codec is found, a **`KeyError`** is set and `NULL` returned.

PyObject *PyCodec_Encoder(const char *encoding)

Return value: New reference. Part of the Stable ABI.
Get an encoder function for the given *encoding*.

PyObject *PyCodec_Decoder(const char *encoding)

Return value: New reference. Part of the Stable ABI.
Get a decoder function for the given *encoding*.

PyObject *PyCodec_IncrementalEncoder(const char *encoding, const char *errors)

Return value: New reference. Part of the Stable ABI.
Get an `IncrementalEncoder` object for the given *encoding*.

PyObject *PyCodec_IncrementalDecoder(const char *encoding, const char *errors)

Return value: New reference. Part of the Stable ABI.
Get an `IncrementalDecoder` object for the given *encoding*.

PyObject *PyCodec_StreamReader(const char *encoding, PyObject *stream, const char *errors)

Return value: New reference. Part of the Stable ABI.
Get a `StreamReader` factory function for the given *encoding*.

PyObject *PyCodec_StreamWriter(const char *encoding, PyObject *stream, const char *errors)

Return value: New reference. Part of the Stable ABI.
Get a `StreamWriter` factory function for the given *encoding*.

Registry API for Unicode encoding error handlers

int PyCodec_RegisterError(const char *name, PyObject *error)

Part of the Stable ABI.
Register the error handling callback function *error* under the given *name*. This callback function will be called by a codec when it encounters unencodable characters/undecodable bytes and *name* is specified as the error parameter in the call to the encode/decode function.

The callback gets a single argument, an instance of `UnicodeEncodeError`, `UnicodeDecodeError` or `UnicodeTranslateError` that holds information about the problematic sequence of characters or bytes and their offset in the original string (see Unicode Exception Objects for functions to extract this information). The callback must either raise the given exception, or return a two-item tuple containing the replacement for the problematic sequence, and an integer giving the offset in the original string at which encoding/decoding should be resumed.

Return `0` on success, `-1` on error.

PyObject *PyCodec_LookupError(const char *name)

Return value: New reference. Part of the Stable ABI.
Lookup the error handling callback function registered under *name*. As a special case `NULL` can be passed, in which case the error handling callback for “strict” will be returned.

PyObject *PyCodec_StrictErrors(PyObject *exc)

Return value: Always NULL. Part of the Stable ABI.
Raise *exc* as an exception.

PyObject *PyCodec_IgnoreErrors(PyObject *exc)

Return value: New reference. Part of the Stable ABI.
Ignore the unicode error, skipping the faulty input.

PyObject *PyCodec_ReplaceErrors(PyObject *exc)

Return value: New reference. Part of the Stable ABI.
Replace the unicode encode error with `?` or `U+FFFD`.

PyObject *PyCodec_XMLCharRefReplaceErrors(PyObject *exc)

Return value: New reference. Part of the Stable ABI.
Replace the unicode encode error with XML character references.

PyObject *PyCodec_BackslashReplaceErrors(PyObject *exc)

Return value: New reference. Part of the Stable ABI.

Replace the unicode encode error with backslash escapes (`\x`, `\u` and `\U`).

PyObject *PyCodec_NameReplaceErrors(PyObject *exc)

Return value: New reference. Part of the Stable ABI since version 3.7.

Replace the unicode encode error with `\N{...}` escapes.

Added in version 3.5.

Codec utility variables

const char *Py_hexdigits

A string constant containing the lowercase hexadecimal digits: `"0123456789abcdef"`.

Added in version 3.3.

PyTime C API

Added in version 3.13.

The clock C API provides access to system clocks. It is similar to the Python **`time`** module.

For C API related to the **`datetime`** module, see DateTime Objects.

Types

type PyTime_t

A timestamp or duration in nanoseconds, represented as a signed 64-bit integer.

The reference point for timestamps depends on the clock used. For example, **`PyTime_Time()`** returns timestamps relative to the UNIX epoch.

The supported range is around [-292.3 years; +292.3 years]. Using the Unix epoch (January 1st, 1970) as reference, the supported date range is around [1677-09-21; 2262-04-11]. The exact limits are exposed as constants:

PyTime_t PyTime_MIN

Minimum value of **`PyTime_t`**.

PyTime_t PyTime_MAX

Maximum value of **`PyTime_t`**.

Clock Functions

The following functions take a pointer to a PyTime_t that they set to the value of a particular clock. Details of each clock are given in the documentation of the corresponding Python function.

The functions return `0` on success, or `-1` (with an exception set) on failure.

On integer overflow, they set the **`PyExc_OverflowError`** exception and set `*result` to the value clamped to the `[PyTime_MIN; PyTime_MAX]` range. (On current systems, integer overflows are likely caused by misconfigured system time.)

As any other C API (unless otherwise specified), the functions must be called with an attached thread state.

int PyTime_Monotonic(PyTime_t *result)

Read the monotonic clock. See **`time.monotonic()`** for important details on this clock.

int PyTime_PerfCounter(PyTime_t *result)

Read the performance counter. See **`time.perf_counter()`** for important details on this clock.

int PyTime_Time(PyTime_t *result)

Read the “wall clock” time. See **`time.time()`** for details important on this clock.

Raw Clock Functions

Similar to clock functions, but don’t set an exception on error and don’t require the caller to have an attached thread state.

On success, the functions return `0`.

On failure, they set `*result` to `0` and return `-1`, *without* setting an exception. To get the cause of the error, attach a thread state, and call the regular (non-`Raw`) function. Note that the regular function may succeed after the `Raw` one failed.

int PyTime_MonotonicRaw(PyTime_t *result)

Similar to **`PyTime_Monotonic()`**, but don't set an exception on error and don't require an attached thread state.

int PyTime_PerfCounterRaw(PyTime_t *result)

Similar to **`PyTime_PerfCounter()`**, but don't set an exception on error and don't require an attached thread state.

int PyTime_TimeRaw(PyTime_t *result)

Similar to **`PyTime_Time()`**, but don't set an exception on error and don't require an attached thread state.

Conversion functions

double PyTime_AsSecondsDouble(PyTime_t t)

Convert a timestamp to a number of seconds as a C double.

The function cannot fail, but note that double has limited accuracy for large values.

Support for Perf Maps

On supported platforms (as of this writing, only Linux), the runtime can take advantage of *perf map files* to make Python functions visible to an external profiling tool (such as perf [https://perf.wiki.kernel.org/index.php/Main_Page]). A running process may create a file in the `/tmp` directory, which contains entries that can map a section of executable code to a name. This interface is described in the documentation of the Linux Perf tool [https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/tree/tools/perf/Documentation/jit-interface.txt].

In Python, these helper APIs can be used by libraries and features that rely on generating machine code on the fly.

Note that holding an attached thread state is not required for these APIs.

int PyUnstable_PerfMapState_Init(void)

> *This is Unstable API. It may change without warning in minor releases.*

Open the `/tmp/perf-$pid.map` file, unless it's already opened, and create a lock to ensure thread-safe writes to the file (provided the writes are done through **`PyUnstable_WritePerfMapEntry()`**). Normally, there's no need to call this explicitly; just use **`PyUnstable_WritePerfMapEntry()`** and it will initialize the state on first call.

Returns `0` on success, `-1` on failure to create/open the perf map file, or `-2` on failure to create a lock. Check `errno` for more information about the cause of a failure.

int PyUnstable_WritePerfMapEntry(const void *code_addr, unsigned int code_size, const char *entry_name)

This is Unstable API. It may change without warning in minor releases.

Write one single entry to the `/tmp/perf-$pid.map` file. This function is thread safe. Here is what an example entry looks like:

```
# address      size  name
7f3529fcf759 b     py::bar:/run/t.py
```

Will call `PyUnstable_PerfMapState_Init()` before writing the entry, if the perf map file is not already opened. Returns `0` on success, or the same error codes as `PyUnstable_PerfMapState_Init()` on failure.

void PyUnstable_PerfMapState_Fini(void)

This is Unstable API. It may change without warning in minor releases.

Close the perf map file opened by `PyUnstable_PerfMapState_Init()`. This is called by the runtime itself during interpreter shut-down. In general, there shouldn't be a reason to explicitly call this, except to handle specific scenarios such as forking.

Abstract Objects Layer

The functions in this chapter interact with Python objects regardless of their type, or with wide classes of object types (e.g. all numerical types, or all sequence types). When used on object types for which they do not apply, they will raise a Python exception.

It is not possible to use these functions on objects that are not properly initialized, such as a list object that has been created by `PyList_New()`, but whose items have not been set to some non-`NULL` value yet.

- Object Protocol
- Call Protocol
  - The *tp_call* Protocol
  - The Vectorcall Protocol
    - Recursion Control
    - Vectorcall Support API
  - Object Calling API
  - Call Support API
- Number Protocol
- Sequence Protocol
- Mapping Protocol
- Iterator Protocol
- Buffer Protocol
  - Buffer structure
  - Buffer request types
    - request-independent fields
    - readonly, format
    - shape, strides, suboffsets
    - contiguity requests
    - compound requests
  - Complex arrays

    - NumPy-style: shape and strides
    - PIL-style: shape, strides and suboffsets
- Buffer-related functions

Object Protocol

PyObject *Py_GetConstant(unsigned int constant_id)

Part of the Stable ABI since version 3.13.
Get a strong reference to a constant.

Set an exception and return `NULL` if *constant_id* is invalid.

constant_id must be one of these constant identifiers:

Constant Identifier	Value	Returned object
Py_CONSTANT_NONE	`0`	`None`
Py_CONSTANT_FALSE	`1`	`False`
Py_CONSTANT_TRUE	`2`	`True`
Py_CONSTANT_ELLIPSIS	`3`	`Ellipsis`
Py_CONSTANT_NOT_IMPLEMENTED	`4`	`NotImplemented`
Py_CONSTANT_ZERO	`5`	`0`
Py_CONSTANT_ONE	`6`	`1`
Py_CONSTANT_EMPTY_STR	`7`	`''`
Py_CONSTANT_EMPTY_BYTES	`8`	`b''`
Py_CONSTANT_EMPTY_TUPLE	`9`	`()`

Numeric values are only given for projects which cannot use the constant identifiers.

Added in version 3.13.

CPython implementation detail: In CPython, all of these constants are immortal.

PyObject *Py_GetConstantBorrowed(unsigned int constant_id)

Part of the Stable ABI since version 3.13.
Similar to `Py_GetConstant()`, but return a borrowed reference.

This function is primarily intended for backwards compatibility: using `Py_GetConstant()` is recommended for new code.

The reference is borrowed from the interpreter, and is valid until the interpreter finalization.

Added in version 3.13.

PyObject *Py_NotImplemented

The `NotImplemented` singleton, used to signal that an operation is not implemented for the given type combination.

Py_RETURN_NOTIMPLEMENTED

Properly handle returning `Py_NotImplemented` from within a C function (that is, create a new strong reference to `NotImplemented` and return it).

Py_PRINT_RAW

Flag to be used with multiple functions that print the object (like `PyObject_Print()` and `PyFile_WriteObject()`). If passed, these functions use the `str()` of the object instead of the `repr()`.

int PyObject_Print(PyObject *o, FILE *fp, int flags)

Print an object *o*, on file *fp*. Returns `-1` on error. The flags argument is used to enable certain printing options. The only option currently supported is `Py_PRINT_RAW`; if given, the `str()` of the object is written instead of the `repr()`.

int PyObject_HasAttrWithError(PyObject *o, PyObject *attr_name)

Part of the Stable ABI since version 3.13.
Returns `1` if *o* has the attribute *attr_name*, and `0` otherwise. This is equivalent to the Python expression `hasattr(o, attr_name)`. On failure, return `-1`.

Added in version 3.13.

int PyObject_HasAttrStringWithError(PyObject *o, const char *attr_name)

Part of the Stable ABI since version 3.13.
This is the same as `PyObject_HasAttrWithError()`, but *attr_name* is specified as a const char* UTF-8 encoded bytes string, rather than a PyObject*.

Added in version 3.13.

int PyObject_HasAttr(PyObject *o, PyObject *attr_name)

Part of the Stable ABI.
Returns `1` if *o* has the attribute *attr_name*, and `0` otherwise. This function always succeeds.

Note

Exceptions that occur when this calls `__getattr__()` and `__getattribute__()` methods aren’t propagated, but instead given to `sys.unraisablehook()`. For proper error handling, use `PyObject_HasAttrWithError()`, `PyObject_GetOptionalAttr()` or `PyObject_GetAttr()` instead.

int PyObject_HasAttrString(PyObject *o, const char *attr_name)

Part of the Stable ABI.

This is the same as `PyObject_HasAttr()`, but *attr_name* is specified as a const char* UTF-8 encoded bytes string, rather than a PyObject*.

Note

Exceptions that occur when this calls `__getattr__()` and `__getattribute__()` methods or while creating the temporary `str` object are silently ignored. For proper error handling, use `PyObject_HasAttrStringWithError()`, `PyObject_GetOptionalAttrString()` or `PyObject_GetAttrString()` instead.

PyObject *PyObject_GetAttr(PyObject *o, PyObject *attr_name)

Return value: New reference. Part of the Stable ABI.

Retrieve an attribute named *attr_name* from object *o*. Returns the attribute value on success, or `NULL` on failure. This is the equivalent of the Python expression `o.attr_name`.

If the missing attribute should not be treated as a failure, you can use `PyObject_GetOptionalAttr()` instead.

PyObject *PyObject_GetAttrString(PyObject *o, const char *attr_name)

Return value: New reference. Part of the Stable ABI.
This is the same as **`PyObject_GetAttr()`**, but *attr_name* is specified as a const char* UTF-8 encoded bytes string, rather than a PyObject*.

If the missing attribute should not be treated as a failure, you can use **`PyObject_GetOptionalAttrString()`** instead.

int PyObject_GetOptionalAttr(PyObject *obj, PyObject *attr_name, PyObject **result);

Part of the Stable ABI since version 3.13.
Variant of **`PyObject_GetAttr()`** which doesn't raise **`AttributeError`** if the attribute is not found.

If the attribute is found, return `1` and set **result* to a new strong reference to the attribute. If the attribute is not found, return `0` and set **result* to `NULL`; the **`AttributeError`** is silenced. If an error other than **`AttributeError`** is raised, return `-1` and set **result* to `NULL`.

Added in version 3.13.

int PyObject_GetOptionalAttrString(PyObject *obj, const char *attr_name, PyObject **result);

Part of the Stable ABI since version 3.13.
This is the same as **`PyObject_GetOptionalAttr()`**, but *attr_name* is specified as a const char* UTF-8 encoded bytes string, rather than a PyObject*.

Added in version 3.13.

PyObject *PyObject_GenericGetAttr(PyObject *o, PyObject *name)

Return value: New reference. Part of the Stable ABI.
Generic attribute getter function that is meant to be put into a type object's `tp_getattro` slot. It looks for a

descriptor in the dictionary of classes in the object's MRO as well as an attribute in the object's `__dict__` (if present). As outlined in Implementing Descriptors, data descriptors take preference over instance attributes, while non-data descriptors don't. Otherwise, an `AttributeError` is raised.

int PyObject_SetAttr(PyObject *o, PyObject *attr_name, PyObject *v)

Part of the Stable ABI.
Set the value of the attribute named *attr_name*, for object *o*, to the value *v*. Raise an exception and return `-1` on failure; return `0` on success. This is the equivalent of the Python statement `o.attr_name = v`.

If *v* is `NULL`, the attribute is deleted. This behaviour is deprecated in favour of using `PyObject_DelAttr()`, but there are currently no plans to remove it.

int PyObject_SetAttrString(PyObject *o, const char *attr_name, PyObject *v)

Part of the Stable ABI.
This is the same as `PyObject_SetAttr()`, but *attr_name* is specified as a const char* UTF-8 encoded bytes string, rather than a PyObject*.

If *v* is `NULL`, the attribute is deleted, but this feature is deprecated in favour of using `PyObject_DelAttrString()`.

The number of different attribute names passed to this function should be kept small, usually by using a statically allocated string as *attr_name*. For attribute names that aren't known at compile time, prefer calling `PyUnicode_FromString()` and `PyObject_SetAttr()` directly. For more details, see `PyUnicode_InternFromString()`, which may be used internally to create a key object.

int PyObject_GenericSetAttr(PyObject *o, PyObject *name, PyObject *value)

Part of the Stable ABI.

Generic attribute setter and deleter function that is meant to be put into a type object's **`tp_setattro`** slot. It looks for a data descriptor in the dictionary of classes in the object's MRO, and if found it takes preference over setting or deleting the attribute in the instance dictionary. Otherwise, the attribute is set or deleted in the object's **`__dict__`** (if present). On success, `0` is returned, otherwise an **`AttributeError`** is raised and `-1` is returned.

int PyObject_DelAttr(PyObject *o, PyObject *attr_name)

Part of the Stable ABI since version 3.13.

Delete attribute named *attr_name*, for object *o*. Returns `-1` on failure. This is the equivalent of the Python statement `del o.attr_name`.

int PyObject_DelAttrString(PyObject *o, const char *attr_name)

Part of the Stable ABI since version 3.13.

This is the same as **`PyObject_DelAttr()`**, but *attr_name* is specified as a const char* UTF-8 encoded bytes string, rather than a PyObject*.

The number of different attribute names passed to this function should be kept small, usually by using a statically allocated string as *attr_name*. For attribute names that aren't known at compile time, prefer calling **`PyUnicode_FromString()`** and **`PyObject_DelAttr()`** directly. For more details, see **`PyUnicode_InternFromString()`**, which may be used internally to create a key object for lookup.

PyObject *PyObject_GenericGetDict(PyObject *o, void *context)

Return value: New reference. Part of the Stable ABI since version 3.10.

A generic implementation for the getter of a `__dict__` descriptor. It creates the dictionary if necessary.

This function may also be called to get the **`__dict__`** of the object *o*. Pass `NULL` for *context* when calling it. Since this function may need to allocate memory for the dictionary, it may be more efficient to call **`PyObject_GetAttr()`** when accessing an attribute on the object.

On failure, returns `NULL` with an exception set.

Added in version 3.3.

int PyObject_GenericSetDict(PyObject *o, PyObject *value, void *context)

Part of the Stable ABI since version 3.7.

A generic implementation for the setter of a `__dict__` descriptor. This implementation does not allow the dictionary to be deleted.

Added in version 3.3.

PyObject **_PyObject_GetDictPtr(PyObject *obj)

Return a pointer to **`__dict__`** of the object *obj*. If there is no `__dict__`, return `NULL` without setting an exception.

This function may need to allocate memory for the dictionary, so it may be more efficient to call **`PyObject_GetAttr()`** when accessing an attribute on the object.

PyObject *PyObject_RichCompare(PyObject *o1, PyObject *o2, int opid)

Return value: New reference. Part of the Stable ABI.

Compare the values of *o1* and *o2* using the operation specified by *opid*, which must be one of **`Py_LT`**, **`Py_LE`**, **`Py_EQ`**, **`Py_NE`**, **`Py_GT`**, or **`Py_GE`**, corresponding to `<`, `<=`, `==`, `!=`, `>`, or `>=` respectively. This is the equivalent of the Python expression `o1 op o2`, where `op` is the operator corresponding to *opid*. Returns the value of the comparison on success, or `NULL` on failure.

int PyObject_RichCompareBool(PyObject *o1, PyObject *o2, int opid)

Part of the Stable ABI.

Compare the values of *o1* and *o2* using the operation specified by *opid*, like **`PyObject_RichCompare()`**, but returns `-1` on error, `0` if the result is false, `1` otherwise.

Note

If *o1* and *o2* are the same object, **`PyObject_RichCompareBool()`** will always return `1` for **`Py_EQ`** and `0` for **`Py_NE`**.

PyObject *PyObject_Format(PyObject *obj, PyObject *format_spec)

Part of the Stable ABI.

Format *obj* using *format_spec*. This is equivalent to the Python expression `format(obj, format_spec)`.

format_spec may be `NULL`. In this case the call is equivalent to `format(obj)`. Returns the formatted string on success, `NULL` on failure.

PyObject *PyObject_Repr(PyObject *o)

Return value: New reference. Part of the Stable ABI. Compute a string representation of object *o*. Returns the string representation on success, `NULL` on failure. This is the equivalent of the Python expression `repr(o)`. Called by the **`repr()`** built-in function.

Changed in version 3.4: This function now includes a debug assertion to help ensure that it does not silently discard an active exception.

PyObject *PyObject_ASCII(PyObject *o)

Return value: New reference. Part of the Stable ABI. As **`PyObject_Repr()`**, compute a string representation of object *o*, but escape the non-ASCII characters in the string returned by **`PyObject_Repr()`** with `\x`, `\u` or `\U` escapes. This generates a string similar to that returned by **`PyObject_Repr()`** in Python 2. Called by the **`ascii()`** built-in function.

PyObject *PyObject_Str(PyObject *o)

Return value: New reference. Part of the Stable ABI. Compute a string representation of object *o*. Returns the string representation on success, `NULL` on failure. This is the equivalent of the Python expression `str(o)`. Called by the **`str()`** built-in function and, therefore, by the **`print()`** function.

Changed in version 3.4: This function now includes a debug assertion to help ensure that it does not silently discard an active exception.

PyObject *PyObject_Bytes(PyObject *o)

Return value: New reference. Part of the Stable ABI. Compute a bytes representation of object *o*. `NULL` is returned on failure and a bytes object on success. This is equivalent to the Python expression `bytes(o)`, when *o* is not an integer. Unlike `bytes(o)`, a TypeError is raised when *o* is an integer instead of a zero-initialized bytes object.

int PyObject_IsSubclass(PyObject *derived, PyObject *cls)

Part of the Stable ABI.

Return `1` if the class *derived* is identical to or derived from the class *cls*, otherwise return `0`. In case of an error, return `-1`.

If *cls* is a tuple, the check will be done against every entry in *cls*. The result will be `1` when at least one of the checks returns `1`, otherwise it will be `0`.

If *cls* has a **`__subclasscheck__()`** method, it will be called to determine the subclass status as described in **PEP 3119** [https://peps.python.org/pep-3119/]. Otherwise, *derived* is a subclass of *cls* if it is a direct or indirect subclass, i.e. contained in **`cls.__mro__`**.

Normally only class objects, i.e. instances of **`type`** or a derived class, are considered classes. However, objects can override this by having a **`__bases__`** attribute (which must be a tuple of base classes).

int PyObject_IsInstance(PyObject *inst, PyObject *cls)

Part of the Stable ABI.

Return `1` if *inst* is an instance of the class *cls* or a subclass of *cls*, or `0` if not. On error, returns `-1` and sets an exception.

If *cls* is a tuple, the check will be done against every entry in *cls*. The result will be `1` when at least one of the checks returns `1`, otherwise it will be `0`.

If *cls* has a **`__instancecheck__()`** method, it will be called to determine the subclass status as described in **PEP 3119** [https://peps.python.org/pep-3119/]. Otherwise, *inst* is an instance of *cls* if its class is a subclass of *cls*.

An instance *inst* can override what is considered its class by having a **`__class__`** attribute.

An object *cls* can override if it is considered a class, and what its base classes are, by having a **`__bases__`** attribute (which must be a tuple of base classes).

Py_hash_t PyObject_Hash(PyObject *o)

Part of the Stable ABI.
Compute and return the hash value of an object *o*. On failure, return `-1`. This is the equivalent of the Python expression `hash(o)`.

Changed in version 3.2: The return type is now Py_hash_t. This is a signed integer the same size as **`Py_ssize_t`**.

Py_hash_t PyObject_HashNotImplemented(PyObject *o)

Part of the Stable ABI.
Set a **`TypeError`** indicating that `type(o)` is not hashable and return `-1`. This function receives special treatment when stored in a `tp_hash` slot, allowing a type to explicitly indicate to the interpreter that it is not hashable.

int PyObject_IsTrue(PyObject *o)

Part of the Stable ABI.
Returns `1` if the object *o* is considered to be true, and `0` otherwise. This is equivalent to the Python expression `not not o`. On failure, return `-1`.

int PyObject_Not(PyObject *o)

Part of the Stable ABI.

Returns `0` if the object *o* is considered to be true, and `1` otherwise. This is equivalent to the Python expression `not o`. On failure, return `-1`.

PyObject *PyObject_Type(PyObject *o)

Return value: New reference. Part of the Stable ABI.
When *o* is non-`NULL`, returns a type object corresponding to the object type of object *o*. On failure, raises **`SystemError`** and returns `NULL`. This is equivalent to the Python expression `type(o)`. This function creates a new strong reference to the return value. There's really no reason to use this function instead of the **`Py_TYPE()`** function, which returns a pointer of type PyTypeObject*, except when a new strong reference is needed.

int PyObject_TypeCheck(PyObject *o, PyTypeObject *type)

Return non-zero if the object *o* is of type *type* or a subtype of *type*, and `0` otherwise. Both parameters must be non-`NULL`.

Py_ssize_t PyObject_Size(PyObject *o)
Py_ssize_t PyObject_Length(PyObject *o)

Part of the Stable ABI.
Return the length of object *o*. If the object *o* provides either the sequence and mapping protocols, the sequence length is returned. On error, `-1` is returned. This is the equivalent to the Python expression `len(o)`.

Py_ssize_t PyObject_LengthHint(PyObject *o, Py_ssize_t defaultvalue)

Return an estimated length for the object *o*. First try to return its actual length, then an estimate using **`__length_hint__()`**, and finally return the default value. On error return `-1`. This is the equivalent to the Python expression `operator.length_hint(o, defaultvalue)`.

Added in version 3.4.

PyObject *PyObject_GetItem(PyObject *o, PyObject *key)

Return value: New reference. Part of the Stable ABI.

Return element of *o* corresponding to the object *key* or `NULL` on failure. This is the equivalent of the Python expression `o[key]`.

int PyObject_SetItem(PyObject *o, PyObject *key, PyObject *v)

Part of the Stable ABI.

Map the object *key* to the value *v*. Raise an exception and return `-1` on failure; return `0` on success. This is the equivalent of the Python statement `o[key] = v`. This function *does not* steal a reference to *v*.

int PyObject_DelItem(PyObject *o, PyObject *key)

Part of the Stable ABI.

Remove the mapping for the object *key* from the object *o*. Return `-1` on failure. This is equivalent to the Python statement `del o[key]`.

int PyObject_DelItemString(PyObject *o, const char *key)

Part of the Stable ABI.

This is the same as **`PyObject_DelItem()`**, but *key* is specified as a const char* UTF-8 encoded bytes string, rather than a PyObject*.

PyObject *PyObject_Dir(PyObject *o)

Return value: New reference. Part of the Stable ABI.

This is equivalent to the Python expression `dir(o)`, returning a (possibly empty) list of strings appropriate for the object argument, or `NULL` if there was an error. If the argument is `NULL`, this is like the Python `dir()`, returning the names of the current locals; in this case, if no execution frame is active then `NULL` is returned but **`PyErr_Occurred()`** will return false.

PyObject *PyObject_GetIter(PyObject *o)

Return value: New reference. Part of the Stable ABI.
This is equivalent to the Python expression `iter(o)`. It returns a new iterator for the object argument, or the object itself if the object is already an iterator. Raises **`TypeError`** and returns `NULL` if the object cannot be iterated.

PyObject *PyObject_SelfIter(PyObject *obj)

Return value: New reference. Part of the Stable ABI.
This is equivalent to the Python `__iter__(self): return self` method. It is intended for iterator types, to be used in the **`PyTypeObject.tp_iter`** slot.

PyObject *PyObject_GetAIter(PyObject *o)

Return value: New reference. Part of the Stable ABI since version 3.10.
This is the equivalent to the Python expression `aiter(o)`. Takes an **`AsyncIterable`** object and returns an **`AsyncIterator`** for it. This is typically a new iterator but if the argument is an **`AsyncIterator`**, this returns itself. Raises **`TypeError`** and returns `NULL` if the object cannot be iterated.

Added in version 3.10.

void *PyObject_GetTypeData(PyObject *o, PyTypeObject *cls)

Part of the Stable ABI since version 3.12.
Get a pointer to subclass-specific data reserved for *cls*.

The object *o* must be an instance of *cls*, and *cls* must have been created using negative **`PyType_Spec.basicsize`**. Python does not check this.

On error, set an exception and return `NULL`.

Added in version 3.12.

Py_ssize_t PyType_GetTypeDataSize(PyTypeObject *cls)

Part of the Stable ABI since version 3.12.
Return the size of the instance memory space reserved for *cls*, i.e. the size of the memory **`PyObject_GetTypeData()`** returns.

This may be larger than requested using **`-PyType_Spec.basicsize`**; it is safe to use this larger size (e.g. with **`memset()`**).

The type *cls* **must** have been created using negative **`PyType_Spec.basicsize`**. Python does not check this.

On error, set an exception and return a negative value.

Added in version 3.12.

void *PyObject_GetItemData(PyObject *o)

Get a pointer to per-item data for a class with **`Py_TPFLAGS_ITEMS_AT_END`**.

On error, set an exception and return `NULL`. **`TypeError`** is raised if *o* does not have **`Py_TPFLAGS_ITEMS_AT_END`** set.

Added in version 3.12.

int PyObject_VisitManagedDict(PyObject *obj, visitproc visit, void *arg)

Visit the managed dictionary of *obj*.

This function must only be called in a traverse function of the type which has the **`Py_TPFLAGS_MANAGED_DICT`** flag set.

Added in version 3.13.

void PyObject_ClearManagedDict(PyObject *obj)

Clear the managed dictionary of *obj*.

This function must only be called in a clear function of the type which has the `Py_TPFLAGS_MANAGED_DICT` flag set.

Added in version 3.13.

int PyUnstable_Object_EnableDeferredRefcount(PyObject *obj)

This is Unstable API. It may change without warning in minor releases.

Enable deferred reference counting [https://peps.python.org/pep-0703/#deferred-reference-counting] on *obj*, if supported by the runtime. In the free-threaded build, this allows the interpreter to avoid reference count adjustments to *obj*, which may improve multi-threaded performance. The tradeoff is that *obj* will only be deallocated by the tracing garbage collector, and not when the interpreter no longer has any references to it.

This function returns `1` if deferred reference counting is enabled on *obj*, and `0` if deferred reference counting is not supported or if the hint was ignored by the interpreter, such as when deferred reference counting is already enabled on *obj*. This function is thread-safe, and cannot fail.

This function does nothing on builds with the GIL enabled, which do not support deferred reference counting. This also does nothing if *obj* is not an object tracked by the garbage collector (see `gc.is_tracked()` and `PyObject_GC_IsTracked()`).

This function is intended to be used soon after *obj* is created, by the code that creates it, such as in the object's **`tp_new`** slot.

Added in version 3.14.

int PyUnstable_Object_IsUniqueReferencedTemporary(PyObject *obj)

This is Unstable API. It may change without warning in minor releases.

Check if *obj* is a unique temporary object. Returns `1` if *obj* is known to be a unique temporary object, and `0` otherwise. This function cannot fail, but the check is conservative, and may return `0` in some cases even if *obj* is a unique temporary object.

If an object is a unique temporary, it is guaranteed that the current code has the only reference to the object. For arguments to C functions, this should be used instead of checking if the reference count is `1`. Starting with Python 3.14, the interpreter internally avoids some reference count modifications when loading objects onto the operands stack by borrowing references when possible, which means that a reference count of `1` by itself does not guarantee that a function argument uniquely referenced.

In the example below, `my_func` is called with a unique temporary object as its argument:

```
my_func([1, 2, 3])
```

In the example below, `my_func` is **not** called with a unique temporary object as its argument, even if its refcount is

```
1:

my_list = [1, 2, 3]
my_func(my_list)
```

See also the function `Py_REFCNT()`.

Added in version 3.14.

int PyUnstable_IsImmortal(PyObject *obj)

This is Unstable API. It may change without warning in minor releases.

This function returns non-zero if *obj* is immortal, and zero otherwise. This function cannot fail.

Note

Objects that are immortal in one CPython version are not guaranteed to be immortal in another.

Added in version 3.14.

int PyUnstable_TryIncRef(PyObject *obj)

This is Unstable API. It may change without warning in minor releases.

Increments the reference count of *obj* if it is not zero. Returns `1` if the object’s reference count was successfully incremented. Otherwise, this function returns `0`.

`PyUnstable_EnableTryIncRef()` must have been called earlier on *obj* or this function may spuriously return `0` in the free threading build.

This function is logically equivalent to the following C code, except that it behaves atomically in the free threading build:

```
if (Py_REFCNT(op) > 0) {
   Py_INCREF(op);
   return 1;
}
return 0;
```

This is intended as a building block for managing weak references without the overhead of a Python weak reference object.

Typically, correct use of this function requires support from *obj*’s deallocator (**`tp_dealloc`**). For example, the following sketch could be adapted to implement a “weakmap” that works like a **`WeakValueDictionary`** for a specific type:

```
PyMutex mutex;

PyObject *
add_entry(weakmap_key_type *key, PyObject *value)
{
    PyUnstable_EnableTryIncRef(value);
    weakmap_type weakmap = ...;
    PyMutex_Lock(&mutex);
    weakmap_add_entry(weakmap, key, value);
    PyMutex_Unlock(&mutex);
    Py_RETURN_NONE;
}

PyObject *
get_value(weakmap_key_type *key)
{
    weakmap_type weakmap = ...;
    PyMutex_Lock(&mutex);
    PyObject *result = weakmap_find(weakmap, key);
    if (PyUnstable_TryIncRef(result)) {
        // `result` is safe to use
        PyMutex_Unlock(&mutex);
        return result;
    }
    // if we get here, `result` is starting to be garbage-
collected,
    // but has not been removed from the weakmap yet
    PyMutex_Unlock(&mutex);
    return NULL;
}

// tp_dealloc function for weakmap values
void
value_dealloc(PyObject *value)
{
    weakmap_type weakmap = ...;
    PyMutex_Lock(&mutex);
    weakmap_remove_value(weakmap, value);

    ...
    PyMutex_Unlock(&mutex);
}
```

Added in version 3.14.

void PyUnstable_EnableTryIncRef(PyObject *obj)

This is Unstable API. It may change without warning in minor releases.

Enables subsequent uses of **`PyUnstable_TryIncRef()`** on *obj*. The caller must hold a strong reference to *obj* when calling this.

Added in version 3.14.

int PyUnstable_Object_IsUniquelyReferenced(PyObject *op)

This is Unstable API. It may change without warning in minor releases.

Determine if *op* only has one reference.

On GIL-enabled builds, this function is equivalent to Py_REFCNT(op) == 1.

On a free threaded build, this checks if *op*’s reference count is equal to one and additionally checks if *op* is only used by this thread. Py_REFCNT(op) == 1 is **not** thread-safe on free threaded builds; prefer this function.

The caller must hold an attached thread state, despite the fact that this function doesn’t call into the Python interpreter. This function cannot fail.

Added in version 3.14.

Call Protocol

CPython supports two different calling protocols: *tp_call* and vectorcall.

The *tp_call* Protocol

Instances of classes that set **`tp_call`** are callable. The signature of the slot is:

```
PyObject *tp_call(PyObject *callable, PyObject *args, PyObject
*kwargs);
```

A call is made using a tuple for the positional arguments and a dict for the keyword arguments, similarly to `callable(*args, **kwargs)` in Python code. *args* must be non-NULL (use an empty tuple if there are no arguments) but *kwargs* may be *NULL* if there are no keyword arguments.

This convention is not only used by *tp_call*: **`tp_new`** and **`tp_init`** also pass arguments this way.

To call an object, use **`PyObject_Call()`** or another call API.

The Vectorcall Protocol

Added in version 3.9.

The vectorcall protocol was introduced in **PEP 590** [https://peps.python.org/pep-0590/] as an additional protocol for making calls more efficient.

As rule of thumb, CPython will prefer the vectorcall for internal calls if the callable supports it. However, this is not a hard rule. Additionally, some third-party extensions use *tp_call* directly (rather than using `PyObject_Call()`). Therefore, a class supporting vectorcall must also implement `tp_call`. Moreover, the callable must behave the same regardless of which protocol is used. The recommended way to achieve this is by setting `tp_call` to `PyVectorcall_Call()`. This bears repeating:

Warning

A class supporting vectorcall **must** also implement `tp_call` with the same semantics.

Changed in version 3.12: The `Py_TPFLAGS_HAVE_VECTORCALL` flag is now removed from a class when the class's `__call__()` method is reassigned. (This internally sets `tp_call` only, and thus may make it behave differently than the vectorcall function.) In earlier Python versions, vectorcall should only be used with `immutable` or static types.

A class should not implement vectorcall if that would be slower than *tp_call*. For example, if the callee needs to convert the arguments to an args tuple and kwargs dict anyway, then there is no point in implementing vectorcall.

Classes can implement the vectorcall protocol by enabling the `Py_TPFLAGS_HAVE_VECTORCALL` flag and setting `tp_vectorcall_offset` to the offset inside the object structure where a *vectorcallfunc* appears. This is a pointer to a function with the following signature:

typedef PyObject *(*vectorcallfunc)(PyObject *callable, PyObject *const *args, size_t nargsf, PyObject *kwnames)

Part of the Stable ABI since version 3.12.

- *callable* is the object being called.
- *args* is a C array consisting of the positional arguments followed by the
  values of the keyword arguments. This can be *NULL* if there are no arguments.

- *nargsf* is the number of positional arguments plus possibly the
  `PY_VECTORCALL_ARGUMENTS_OFFSET` flag. To get the actual number of positional arguments from *nargsf*, use **`PyVectorcall_NARGS()`**.

- *kwnames* is a tuple containing the names of the keyword arguments;
  in other words, the keys of the kwargs dict. These names must be strings (instances of `str` or a subclass) and they must be unique. If there are no keyword arguments, then *kwnames* can instead be *NULL*.

PY_VECTORCALL_ARGUMENTS_OFFSET

Part of the Stable ABI since version 3.12.
If this flag is set in a vectorcall *nargsf* argument, the callee is allowed to temporarily change `args[-1]`. In other words, *args* points to argument 1 (not 0) in the allocated vector. The callee must restore the value of `args[-1]` before returning.

For **`PyObject_VectorcallMethod()`**, this flag means instead that `args[0]` may be changed.

Whenever they can do so cheaply (without additional allocation), callers are encouraged to use **`PY_VECTORCALL_ARGUMENTS_OFFSET`**. Doing so will allow callables such as bound methods to make their onward calls (which include a prepended *self* argument) very efficiently.

Added in version 3.8.

To call an object that implements vectorcall, use a call API function as with any other callable. `PyObject_Vectorcall()` will usually be most efficient.

Recursion Control

When using *tp_call*, callees do not need to worry about recursion: CPython uses `Py_EnterRecursiveCall()` and `Py_LeaveRecursiveCall()` for calls made using *tp_call*.

For efficiency, this is not the case for calls done using vectorcall: the callee should use *Py_EnterRecursiveCall* and *Py_LeaveRecursiveCall* if needed.

Vectorcall Support API

Py_ssize_t PyVectorcall_NARGS(size_t nargsf)

Part of the Stable ABI since version 3.12.
Given a vectorcall *nargsf* argument, return the actual number of arguments. Currently equivalent to:

```
(Py_ssize_t)(nargsf & ~PY_VECTORCALL_ARGUMENTS_OFFSET)
```

However, the function `PyVectorcall_NARGS` should be used to allow for future extensions.

Added in version 3.8.

vectorcallfunc PyVectorcall_Function(PyObject *op)

If *op* does not support the vectorcall protocol (either because the type does not or because the specific instance does not), return *NULL*. Otherwise, return the vectorcall function pointer stored in *op*. This function never raises an exception.

This is mostly useful to check whether or not *op* supports vectorcall, which can be done by checking

`PyVectorcall_Function(op) != NULL`.

Added in version 3.9.

PyObject *PyVectorcall_Call(PyObject *callable, PyObject *tuple, PyObject *dict)

Part of the Stable ABI since version 3.12.
Call *callable*'s **`vectorcallfunc`** with positional and keyword arguments given in a tuple and dict, respectively.

This is a specialized function, intended to be put in the **`tp_call`** slot or be used in an implementation of `tp_call`. It does not check the **`Py_TPFLAGS_HAVE_VECTORCALL`** flag and it does not fall back to `tp_call`.

Added in version 3.8.

Object Calling API

Various functions are available for calling a Python object. Each converts its arguments to a convention supported by the called object – either *tp_call* or vectorcall. In order to do as little conversion as possible, pick one that best fits the format of data you have available.

The following table summarizes the available functions; please see individual documentation for details.

Function	callable	args	kwargs
`PyObject_Call()`	`PyObject *`	tuple	dict/`NULL`

Function	callable	args	kwargs
`PyObject_CallNoArgs()`	`PyObject *`	—	—
`PyObject_CallOneArg()`	`PyObject *`	1 object	—
`PyObject_CallObject()`	`PyObject *`	tuple/`NULL`	—
`PyObject_CallFunction()`	`PyObject *`	format	—
`PyObject_CallMethod()`	obj + `char*`	format	—
`PyObject_CallFunctionObjArgs()`	`PyObject *`	variadic	—
`PyObject_CallMethodObjArgs()`	obj + name	variadic	—
`PyObject_CallMethodNoArgs()`	obj + name	—	—
`PyObject_CallMethodOneArg()`	obj + name	1 object	—

Function	**callable**	**args**	**kwargs**
`PyObject_Vectorcall()`	`PyObject *`	vectorcall	vectorcall
`PyObject_VectorcallDict()`	`PyObject *`	vectorcall	dict/`NULL`
`PyObject_VectorcallMethod()`	arg + name	vectorcall	vectorcall

PyObject *PyObject_Call(PyObject *callable, PyObject *args, PyObject *kwargs)

Return value: New reference. Part of the Stable ABI.
Call a callable Python object *callable*, with arguments given by the tuple *args*, and named arguments given by the dictionary *kwargs*.

args must not be *NULL*; use an empty tuple if no arguments are needed. If no named arguments are needed, *kwargs* can be *NULL*.

Return the result of the call on success, or raise an exception and return *NULL* on failure.

This is the equivalent of the Python expression: `callable(*args, **kwargs)`.

PyObject *PyObject_CallNoArgs(PyObject *callable)

Return value: New reference. Part of the Stable ABI since version 3.10.

Call a callable Python object *callable* without any arguments. It is the most efficient way to call a callable Python object without any argument.

Return the result of the call on success, or raise an exception and return *NULL* on failure.

Added in version 3.9.

PyObject *PyObject_CallOneArg(PyObject *callable, PyObject *arg)

Return value: New reference.
Call a callable Python object *callable* with exactly 1 positional argument *arg* and no keyword arguments.

Return the result of the call on success, or raise an exception and return *NULL* on failure.

Added in version 3.9.

PyObject *PyObject_CallObject(PyObject *callable, PyObject *args)

Return value: New reference. Part of the Stable ABI.
Call a callable Python object *callable*, with arguments given by the tuple *args*. If no arguments are needed, then *args* can be *NULL*.

Return the result of the call on success, or raise an exception and return *NULL* on failure.

This is the equivalent of the Python expression: `callable(*args)`.

PyObject *PyObject_CallFunction(PyObject *callable, const char *format, ...)

Return value: New reference. Part of the Stable ABI.

Call a callable Python object *callable*, with a variable number of C arguments. The C arguments are described using a **`Py_BuildValue()`** style format string. The format can be *NULL*, indicating that no arguments are provided.

Return the result of the call on success, or raise an exception and return *NULL* on failure.

This is the equivalent of the Python expression: `callable(*args)`.

Note that if you only pass PyObject* args, **`PyObject_CallFunctionObjArgs()`** is a faster alternative.

Changed in version 3.4: The type of *format* was changed from `char *`.

PyObject *PyObject_CallMethod(PyObject *obj, const char *name, const char *format, ...)

Return value: New reference. Part of the Stable ABI.
Call the method named *name* of object *obj* with a variable number of C arguments. The C arguments are described by a **`Py_BuildValue()`** format string that should produce a tuple.

The format can be *NULL*, indicating that no arguments are provided.

Return the result of the call on success, or raise an exception and return *NULL* on failure.

This is the equivalent of the Python expression: `obj.name(arg1, arg2, ...)`.

Note that if you only pass PyObject* args, **`PyObject_CallMethodObjArgs()`** is a faster alternative.

Changed in version 3.4: The types of *name* and *format* were changed from `char *`.

PyObject *PyObject_CallFunctionObjArgs(PyObject *callable, ...)

Return value: New reference. Part of the Stable ABI.
Call a callable Python object *callable*, with a variable number of PyObject* arguments. The arguments are provided as a variable number of parameters followed by *NULL*.

Return the result of the call on success, or raise an exception and return *NULL* on failure.

This is the equivalent of the Python expression: `callable(arg1, arg2, ...)`.

PyObject *PyObject_CallMethodObjArgs(PyObject *obj, PyObject *name, ...)

Return value: New reference. Part of the Stable ABI.
Call a method of the Python object *obj*, where the name of the method is given as a Python string object in *name*. It is called with a variable number of PyObject* arguments. The arguments are provided as a variable number of parameters followed by *NULL*.

Return the result of the call on success, or raise an exception and return *NULL* on failure.

PyObject *PyObject_CallMethodNoArgs(PyObject *obj, PyObject *name)

Call a method of the Python object *obj* without arguments, where the name of the method is given as a Python string object in *name*.

Return the result of the call on success, or raise an exception and return *NULL* on failure.

Added in version 3.9.

PyObject *PyObject_CallMethodOneArg(PyObject *obj, PyObject *name, PyObject *arg)

Call a method of the Python object *obj* with a single positional argument *arg*, where the name of the method is given as a Python string object in *name*.

Return the result of the call on success, or raise an exception and return *NULL* on failure.

Added in version 3.9.

PyObject *PyObject_Vectorcall(PyObject *callable, PyObject *const *args, size_t nargsf, PyObject *kwnames)

Part of the Stable ABI since version 3.12.
Call a callable Python object *callable*. The arguments are the same as for **`vectorcallfunc`**. If *callable* supports vectorcall, this directly calls the vectorcall function stored in *callable*.

Return the result of the call on success, or raise an exception and return *NULL* on failure.

Added in version 3.8: as `_PyObject_Vectorcall`

Changed in version 3.9: Renamed to the current name, without the leading underscore. The old provisional name is soft deprecated.

PyObject *PyObject_VectorcallDict(PyObject *callable, PyObject *const *args, size_t nargsf, PyObject *kwdict)

Call *callable* with positional arguments passed exactly as in the vectorcall protocol, but with keyword arguments passed as a dictionary *kwdict*. The *args* array contains only the positional arguments.

Regardless of which protocol is used internally, a conversion of arguments needs to be done. Therefore, this function should only be used if the caller already has a dictionary ready to use for the keyword arguments, but not a tuple for the positional arguments.

Added in version 3.9.

PyObject *PyObject_VectorcallMethod(PyObject *name, PyObject *const *args, size_t nargsf, PyObject *kwnames)

Part of the Stable ABI since version 3.12.
Call a method using the vectorcall calling convention. The name of the method is given as a Python string *name*. The object whose method is called is *args[0]*, and the *args* array starting at *args[1]* represents the arguments of the call. There must be at least one positional argument. *nargsf* is the number of positional arguments including *args[0]*, plus **`PY_VECTORCALL_ARGUMENTS_OFFSET`** if the value of `args[0]` may temporarily be changed. Keyword arguments can be passed just like in **`PyObject_Vectorcall()`**.

If the object has the **`Py_TPFLAGS_METHOD_DESCRIPTOR`** feature, this will call the unbound method object with the full *args* vector as arguments.

Return the result of the call on success, or raise an exception and return *NULL* on failure.

Added in version 3.9.

Call Support API

int PyCallable_Check(PyObject *o)

Part of the Stable ABI.

Determine if the object *o* is callable. Return `1` if the object is callable and `0` otherwise. This function always succeeds.

Number Protocol

int PyNumber_Check(PyObject *o)

Part of the Stable ABI.

Returns `1` if the object *o* provides numeric protocols, and false otherwise. This function always succeeds.

Changed in version 3.8: Returns `1` if *o* is an index integer.

PyObject *PyNumber_Add(PyObject *o1, PyObject *o2)

Return value: New reference. Part of the Stable ABI.

Returns the result of adding *o1* and *o2*, or `NULL` on failure. This is the equivalent of the Python expression `o1 + o2`.

PyObject *PyNumber_Subtract(PyObject *o1, PyObject *o2)

Return value: New reference. Part of the Stable ABI.

Returns the result of subtracting *o2* from *o1*, or `NULL` on failure. This is the equivalent of the Python expression `o1 - o2`.

PyObject *PyNumber_Multiply(PyObject *o1, PyObject *o2)

Return value: New reference. Part of the Stable ABI.

Returns the result of multiplying *o1* and *o2*, or `NULL` on failure. This is the equivalent of the Python expression `o1 * o2`.

PyObject *PyNumber_MatrixMultiply(PyObject *o1, PyObject *o2)

Return value: New reference. Part of the Stable ABI since version 3.7.

Returns the result of matrix multiplication on *o1* and *o2*, or `NULL` on failure. This is the equivalent of the Python expression `o1 @ o2`.

Added in version 3.5.

PyObject *PyNumber_FloorDivide(PyObject *o1, PyObject *o2)

Return value: New reference. Part of the Stable ABI.
Return the floor of *o1* divided by *o2*, or `NULL` on failure. This is the equivalent of the Python expression `o1 // o2`.

PyObject *PyNumber_TrueDivide(PyObject *o1, PyObject *o2)

Return value: New reference. Part of the Stable ABI.
Return a reasonable approximation for the mathematical value of *o1* divided by *o2*, or `NULL` on failure. The return value is “approximate” because binary floating-point numbers are approximate; it is not possible to represent all real numbers in base two. This function can return a floating-point value when passed two integers. This is the equivalent of the Python expression `o1 / o2`.

PyObject *PyNumber_Remainder(PyObject *o1, PyObject *o2)

Return value: New reference. Part of the Stable ABI.
Returns the remainder of dividing *o1* by *o2*, or `NULL` on failure. This is the equivalent of the Python expression `o1 % o2`.

PyObject *PyNumber_Divmod(PyObject *o1, PyObject *o2)

Return value: New reference. Part of the Stable ABI.
See the built-in function **`divmod()`**. Returns `NULL` on failure. This is the equivalent of the Python expression `divmod(o1, o2)`.

PyObject *PyNumber_Power(PyObject *o1, PyObject *o2, PyObject *o3)

Return value: New reference. Part of the Stable ABI.

See the built-in function **`pow()`**. Returns `NULL` on failure. This is the equivalent of the Python expression `pow(o1, o2, o3)`, where *o3* is optional. If *o3* is to be ignored, pass **`Py_None`** in its place (passing `NULL` for *o3* would cause an illegal memory access).

PyObject *PyNumber_Negative(PyObject *o)

Return value: New reference. Part of the Stable ABI. Returns the negation of *o* on success, or `NULL` on failure. This is the equivalent of the Python expression `-o`.

PyObject *PyNumber_Positive(PyObject *o)

Return value: New reference. Part of the Stable ABI. Returns *o* on success, or `NULL` on failure. This is the equivalent of the Python expression `+o`.

PyObject *PyNumber_Absolute(PyObject *o)

Return value: New reference. Part of the Stable ABI. Returns the absolute value of *o*, or `NULL` on failure. This is the equivalent of the Python expression `abs(o)`.

PyObject *PyNumber_Invert(PyObject *o)

Return value: New reference. Part of the Stable ABI. Returns the bitwise negation of *o* on success, or `NULL` on failure. This is the equivalent of the Python expression `~o`.

PyObject *PyNumber_Lshift(PyObject *o1, PyObject *o2)

Return value: New reference. Part of the Stable ABI. Returns the result of left shifting *o1* by *o2* on success, or `NULL` on failure. This is the equivalent of the Python expression `o1 << o2`.

PyObject *PyNumber_Rshift(PyObject *o1, PyObject *o2)

Return value: New reference. Part of the Stable ABI.

Returns the result of right shifting *o1* by *o2* on success, or `NULL` on failure. This is the equivalent of the Python expression `o1 >> o2`.

PyObject *PyNumber_And(PyObject *o1, PyObject *o2)

Return value: New reference. Part of the Stable ABI. Returns the “bitwise and” of *o1* and *o2* on success and `NULL` on failure. This is the equivalent of the Python expression `o1 & o2`.

PyObject *PyNumber_Xor(PyObject *o1, PyObject *o2)

Return value: New reference. Part of the Stable ABI. Returns the “bitwise exclusive or” of *o1* by *o2* on success, or `NULL` on failure. This is the equivalent of the Python expression `o1 ^ o2`.

PyObject *PyNumber_Or(PyObject *o1, PyObject *o2)

Return value: New reference. Part of the Stable ABI. Returns the “bitwise or” of *o1* and *o2* on success, or `NULL` on failure. This is the equivalent of the Python expression `o1 | o2`.

PyObject *PyNumber_InPlaceAdd(PyObject *o1, PyObject *o2)

Return value: New reference. Part of the Stable ABI. Returns the result of adding *o1* and *o2*, or `NULL` on failure. The operation is done *in-place* when *o1* supports it. This is the equivalent of the Python statement `o1 += o2`.

PyObject *PyNumber_InPlaceSubtract(PyObject *o1, PyObject *o2)

Return value: New reference. Part of the Stable ABI. Returns the result of subtracting *o2* from *o1*, or `NULL` on failure. The operation is done *in-place* when *o1* supports it. This is the equivalent of the Python statement `o1 -= o2`.

PyObject *PyNumber_InPlaceMultiply(PyObject *o1, PyObject *o2)

Return value: New reference. Part of the Stable ABI.
Returns the result of multiplying *o1* and *o2*, or `NULL` on failure. The operation is done *in-place* when *o1* supports it. This is the equivalent of the Python statement `o1 *= o2`.

PyObject *PyNumber_InPlaceMatrixMultiply(PyObject *o1, PyObject *o2)

Return value: New reference. Part of the Stable ABI since version 3.7.
Returns the result of matrix multiplication on *o1* and *o2*, or `NULL` on failure. The operation is done *in-place* when *o1* supports it. This is the equivalent of the Python statement `o1 @= o2`.

Added in version 3.5.

PyObject *PyNumber_InPlaceFloorDivide(PyObject *o1, PyObject *o2)

Return value: New reference. Part of the Stable ABI.
Returns the mathematical floor of dividing *o1* by *o2*, or `NULL` on failure. The operation is done *in-place* when *o1* supports it. This is the equivalent of the Python statement `o1 //= o2`.

PyObject *PyNumber_InPlaceTrueDivide(PyObject *o1, PyObject *o2)

Return value: New reference. Part of the Stable ABI.
Return a reasonable approximation for the mathematical value of *o1* divided by *o2*, or `NULL` on failure. The return value is “approximate” because binary floating-point numbers are approximate; it is not possible to represent all real numbers in base two. This function can return a floating-point value when passed two integers. The

operation is done *in-place* when *o1* supports it. This is the equivalent of the Python statement `o1 /= o2`.

PyObject *PyNumber_InPlaceRemainder(PyObject *o1, PyObject *o2)

Return value: New reference. Part of the Stable ABI.
Returns the remainder of dividing *o1* by *o2*, or `NULL` on failure. The operation is done *in-place* when *o1* supports it. This is the equivalent of the Python statement `o1 %= o2`.

PyObject *PyNumber_InPlacePower(PyObject *o1, PyObject *o2, PyObject *o3)

Return value: New reference. Part of the Stable ABI.
See the built-in function **`pow()`**. Returns `NULL` on failure. The operation is done *in-place* when *o1* supports it. This is the equivalent of the Python statement `o1 **= o2` when o3 is **`Py_None`**, or an in-place variant of `pow(o1, o2, o3)` otherwise. If *o3* is to be ignored, pass **`Py_None`** in its place (passing `NULL` for *o3* would cause an illegal memory access).

PyObject *PyNumber_InPlaceLshift(PyObject *o1, PyObject *o2)

Return value: New reference. Part of the Stable ABI.
Returns the result of left shifting *o1* by *o2* on success, or `NULL` on failure. The operation is done *in-place* when *o1* supports it. This is the equivalent of the Python statement `o1 <<= o2`.

PyObject *PyNumber_InPlaceRshift(PyObject *o1, PyObject *o2)

Return value: New reference. Part of the Stable ABI.
Returns the result of right shifting *o1* by *o2* on success, or `NULL` on failure. The operation is done *in-place* when *o1*

supports it. This is the equivalent of the Python statement `o1 >>= o2`.

PyObject *PyNumber_InPlaceAnd(PyObject *o1, PyObject *o2)

Return value: New reference. Part of the Stable ABI. Returns the “bitwise and” of *o1* and *o2* on success and `NULL` on failure. The operation is done *in-place* when *o1* supports it. This is the equivalent of the Python statement `o1 &= o2`.

PyObject *PyNumber_InPlaceXor(PyObject *o1, PyObject *o2)

Return value: New reference. Part of the Stable ABI. Returns the “bitwise exclusive or” of *o1* by *o2* on success, or `NULL` on failure. The operation is done *in-place* when *o1* supports it. This is the equivalent of the Python statement `o1 ^= o2`.

PyObject *PyNumber_InPlaceOr(PyObject *o1, PyObject *o2)

Return value: New reference. Part of the Stable ABI. Returns the “bitwise or” of *o1* and *o2* on success, or `NULL` on failure. The operation is done *in-place* when *o1* supports it. This is the equivalent of the Python statement `o1 |= o2`.

PyObject *PyNumber_Long(PyObject *o)

Return value: New reference. Part of the Stable ABI. Returns the *o* converted to an integer object on success, or `NULL` on failure. This is the equivalent of the Python expression `int(o)`.

PyObject *PyNumber_Float(PyObject *o)

Return value: New reference. Part of the Stable ABI.

Returns the *o* converted to a float object on success, or `NULL` on failure. This is the equivalent of the Python expression `float(o)`.

PyObject *PyNumber_Index(PyObject *o)

Return value: New reference. Part of the Stable ABI.
Returns the *o* converted to a Python int on success or `NULL` with a **`TypeError`** exception raised on failure.

Changed in version 3.10: The result always has exact type **`int`**. Previously, the result could have been an instance of a subclass of `int`.

PyObject *PyNumber_ToBase(PyObject *n, int base)

Return value: New reference. Part of the Stable ABI.
Returns the integer *n* converted to base *base* as a string. The *base* argument must be one of 2, 8, 10, or 16. For base 2, 8, or 16, the returned string is prefixed with a base marker of `'0b'`, `'0o'`, or `'0x'`, respectively. If *n* is not a Python int, it is converted with **`PyNumber_Index()`** first.

Py_ssize_t PyNumber_AsSsize_t(PyObject *o, PyObject *exc)

Part of the Stable ABI.
Returns *o* converted to a **`Py_ssize_t`** value if *o* can be interpreted as an integer. If the call fails, an exception is raised and `-1` is returned.

If *o* can be converted to a Python int but the attempt to convert to a **`Py_ssize_t`** value would raise an **`OverflowError`**, then the *exc* argument is the type of exception that will be raised (usually **`IndexError`** or **`OverflowError`**). If *exc* is `NULL`, then the exception is cleared and the value is clipped to `PY_SSIZE_T_MIN` for a negative integer or `PY_SSIZE_T_MAX` for a positive integer.

int PyIndex_Check(PyObject *o)

Part of the Stable ABI since version 3.8.
Returns `1` if *o* is an index integer (has the `nb_index` slot of the `tp_as_number` structure filled in), and `0` otherwise. This function always succeeds.

Sequence Protocol

int PySequence_Check(PyObject *o)

Part of the Stable ABI.

Return `1` if the object provides the sequence protocol, and `0` otherwise. Note that it returns `1` for Python classes with a **`__getitem__()`** method, unless they are **`dict`** subclasses, since in general it is impossible to determine what type of keys the class supports. This function always succeeds.

Py_ssize_t PySequence_Size(PyObject *o)
Py_ssize_t PySequence_Length(PyObject *o)

Part of the Stable ABI.

Returns the number of objects in sequence *o* on success, and `-1` on failure. This is equivalent to the Python expression `len(o)`.

PyObject *PySequence_Concat(PyObject *o1, PyObject *o2)

Return value: New reference. Part of the Stable ABI.

Return the concatenation of *o1* and *o2* on success, and `NULL` on failure. This is the equivalent of the Python expression `o1 + o2`.

PyObject *PySequence_Repeat(PyObject *o, Py_ssize_t count)

Return value: New reference. Part of the Stable ABI.

Return the result of repeating sequence object *o count* times, or `NULL` on failure. This is the equivalent of the Python expression `o * count`.

PyObject *PySequence_InPlaceConcat(PyObject *o1, PyObject *o2)

Return value: New reference. Part of the Stable ABI.
Return the concatenation of *o1* and *o2* on success, and `NULL` on failure. The operation is done *in-place* when *o1* supports it. This is the equivalent of the Python expression `o1 += o2`.

PyObject *PySequence_InPlaceRepeat(PyObject *o, Py_ssize_t count)

Return value: New reference. Part of the Stable ABI.
Return the result of repeating sequence object *o count* times, or `NULL` on failure. The operation is done *in-place* when *o* supports it. This is the equivalent of the Python expression `o *= count`.

PyObject *PySequence_GetItem(PyObject *o, Py_ssize_t i)

Return value: New reference. Part of the Stable ABI.
Return the *i*th element of *o*, or `NULL` on failure. This is the equivalent of the Python expression `o[i]`.

PyObject *PySequence_GetSlice(PyObject *o, Py_ssize_t i1, Py_ssize_t i2)

Return value: New reference. Part of the Stable ABI.
Return the slice of sequence object *o* between *i1* and *i2*, or `NULL` on failure. This is the equivalent of the Python expression `o[i1:i2]`.

int PySequence_SetItem(PyObject *o, Py_ssize_t i, PyObject *v)

Part of the Stable ABI.
Assign object *v* to the *i*th element of *o*. Raise an exception and return `-1` on failure; return `0` on success.

This is the equivalent of the Python statement `o[i] = v`. This function *does not* steal a reference to *v*.

If *v* is `NULL`, the element is deleted, but this feature is deprecated in favour of using **`PySequence_DelItem()`**.

int PySequence_DelItem(PyObject *o, Py_ssize_t i)

Part of the Stable ABI.
Delete the *i*th element of object *o*. Returns `-1` on failure. This is the equivalent of the Python statement `del o[i]`.

int PySequence_SetSlice(PyObject *o, Py_ssize_t i1, Py_ssize_t i2, PyObject *v)

Part of the Stable ABI.
Assign the sequence object *v* to the slice in sequence object *o* from *i1* to *i2*. This is the equivalent of the Python statement `o[i1:i2] = v`.

int PySequence_DelSlice(PyObject *o, Py_ssize_t i1, Py_ssize_t i2)

Part of the Stable ABI.
Delete the slice in sequence object *o* from *i1* to *i2*. Returns `-1` on failure. This is the equivalent of the Python statement `del o[i1:i2]`.

Py_ssize_t PySequence_Count(PyObject *o, PyObject *value)

Part of the Stable ABI.
Return the number of occurrences of *value* in *o*, that is, return the number of keys for which `o[key] == value`. On failure, return `-1`. This is equivalent to the Python expression `o.count(value)`.

int PySequence_Contains(PyObject *o, PyObject *value)

Part of the Stable ABI.

Determine if *o* contains *value*. If an item in *o* is equal to *value*, return `1`, otherwise return `0`. On error, return `-1`. This is equivalent to the Python expression `value in o`.

int PySequence_In(PyObject *o, PyObject *value)

Part of the Stable ABI.

Alias for **`PySequence_Contains()`**.

Deprecated since version 3.14: The function is soft deprecated and should no longer be used to write new code.

Py_ssize_t PySequence_Index(PyObject *o, PyObject *value)

Part of the Stable ABI.

Return the first index *i* for which `o[i] == value`. On error, return `-1`. This is equivalent to the Python expression `o.index(value)`.

PyObject *PySequence_List(PyObject *o)

Return value: New reference. Part of the Stable ABI.

Return a list object with the same contents as the sequence or iterable *o*, or `NULL` on failure. The returned list is guaranteed to be new. This is equivalent to the Python expression `list(o)`.

PyObject *PySequence_Tuple(PyObject *o)

Return value: New reference. Part of the Stable ABI.

Return a tuple object with the same contents as the sequence or iterable *o*, or `NULL` on failure. If *o* is a tuple, a new reference will be returned, otherwise a tuple will be constructed with the appropriate contents. This is equivalent to the Python expression `tuple(o)`.

PyObject *PySequence_Fast(PyObject *o, const char *m)

Return value: New reference. Part of the Stable ABI.

Return the sequence or iterable *o* as an object usable by the other `PySequence_Fast*` family of functions. If the object is not a sequence or iterable, raises **`TypeError`** with *m* as the message text. Returns `NULL` on failure.

The `PySequence_Fast*` functions are thus named because they assume *o* is a **`PyTupleObject`** or a **`PyListObject`** and access the data fields of *o* directly.

As a CPython implementation detail, if *o* is already a sequence or list, it will be returned.

Py_ssize_t PySequence_Fast_GET_SIZE(PyObject *o)

Returns the length of *o*, assuming that *o* was returned by **`PySequence_Fast()`** and that *o* is not `NULL`. The size can also be retrieved by calling **`PySequence_Size()`** on *o*, but **`PySequence_Fast_GET_SIZE()`** is faster because it can assume *o* is a list or tuple.

PyObject *PySequence_Fast_GET_ITEM(PyObject *o, Py_ssize_t i)

Return value: Borrowed reference.
Return the *i*th element of *o*, assuming that *o* was returned by **`PySequence_Fast()`**, *o* is not `NULL`, and that *i* is within bounds.

PyObject **PySequence_Fast_ITEMS(PyObject *o)

Return the underlying array of PyObject pointers. Assumes that *o* was returned by **`PySequence_Fast()`** and *o* is not `NULL`.

Note, if a list gets resized, the reallocation may relocate the items array. So, only use the underlying array pointer in contexts where the sequence cannot change.

PyObject *PySequence_ITEM(PyObject *o, Py_ssize_t i)

Return value: New reference.
Return the *i*th element of *o* or `NULL` on failure. Faster form of **`PySequence_GetItem()`** but without checking that **`PySequence_Check()`** on *o* is true and without adjustment for negative indices.

Mapping Protocol

See also `PyObject_GetItem()`, `PyObject_SetItem()` and `PyObject_DelItem()`.

int PyMapping_Check(PyObject *o)

Part of the Stable ABI.

Return `1` if the object provides the mapping protocol or supports slicing, and `0` otherwise. Note that it returns `1` for Python classes with a `__getitem__()` method, since in general it is impossible to determine what type of keys the class supports. This function always succeeds.

Py_ssize_t PyMapping_Size(PyObject *o)
Py_ssize_t PyMapping_Length(PyObject *o)

Part of the Stable ABI.

Returns the number of keys in object *o* on success, and `-1` on failure. This is equivalent to the Python expression `len(o)`.

PyObject *PyMapping_GetItemString(PyObject *o, const char *key)

Return value: New reference. Part of the Stable ABI.

This is the same as `PyObject_GetItem()`, but *key* is specified as a const char* UTF-8 encoded bytes string, rather than a PyObject*.

int PyMapping_GetOptionalItem(PyObject *obj, PyObject *key, PyObject **result)

Part of the Stable ABI since version 3.13.

Variant of `PyObject_GetItem()` which doesn't raise `KeyError` if the key is not found.

If the key is found, return `1` and set **result* to a new strong reference to the corresponding value. If the key is not found, return `0` and set **result* to `NULL`; the **`KeyError`** is silenced. If an error other than **`KeyError`** is raised, return `-1` and set **result* to `NULL`.

Added in version 3.13.

int PyMapping_GetOptionalItemString(PyObject *obj, const char *key, PyObject **result)

Part of the Stable ABI since version 3.13.
This is the same as **`PyMapping_GetOptionalItem()`**, but *key* is specified as a const char* UTF-8 encoded bytes string, rather than a PyObject*.

Added in version 3.13.

int PyMapping_SetItemString(PyObject *o, const char *key, PyObject *v)

Part of the Stable ABI.
This is the same as **`PyObject_SetItem()`**, but *key* is specified as a const char* UTF-8 encoded bytes string, rather than a PyObject*.

int PyMapping_DelItem(PyObject *o, PyObject *key)

This is an alias of **`PyObject_DelItem()`**.

int PyMapping_DelItemString(PyObject *o, const char *key)

This is the same as **`PyObject_DelItem()`**, but *key* is specified as a const char* UTF-8 encoded bytes string, rather than a PyObject*.

int PyMapping_HasKeyWithError(PyObject *o, PyObject *key)

Part of the Stable ABI since version 3.13.

Return `1` if the mapping object has the key *key* and `0` otherwise. This is equivalent to the Python expression `key in o`. On failure, return `-1`.

Added in version 3.13.

int PyMapping_HasKeyStringWithError(PyObject *o, const char *key)

Part of the Stable ABI since version 3.13.

This is the same as **`PyMapping_HasKeyWithError()`**, but *key* is specified as a const char* UTF-8 encoded bytes string, rather than a PyObject*.

Added in version 3.13.

int PyMapping_HasKey(PyObject *o, PyObject *key)

Part of the Stable ABI.

Return `1` if the mapping object has the key *key* and `0` otherwise. This is equivalent to the Python expression `key in o`. This function always succeeds.

Note

Exceptions which occur when this calls the **`__getitem__()`** method are silently ignored. For proper error handling, use **`PyMapping_HasKeyWithError()`**, **`PyMapping_GetOptionalItem()`** or **`PyObject_GetItem()`** instead.

int PyMapping_HasKeyString(PyObject *o, const char *key)

Part of the Stable ABI.

This is the same as **`PyMapping_HasKey()`**, but *key* is specified as a const char* UTF-8 encoded bytes string, rather than a PyObject*.

Note

Exceptions that occur when this calls the `__getitem__()` method or while creating the temporary `str` object are silently ignored. For proper error handling, use `PyMapping_HasKeyStringWithError()`, `PyMapping_GetOptionalItemString()` or `PyMapping_GetItemString()` instead.

PyObject *PyMapping_Keys(PyObject *o)

Return value: New reference. Part of the Stable ABI.
On success, return a list of the keys in object *o*. On failure, return `NULL`.

Changed in version 3.7: Previously, the function returned a list or a tuple.

PyObject *PyMapping_Values(PyObject *o)

Return value: New reference. Part of the Stable ABI.
On success, return a list of the values in object *o*. On failure, return `NULL`.

Changed in version 3.7: Previously, the function returned a list or a tuple.

PyObject *PyMapping_Items(PyObject *o)

Return value: New reference. Part of the Stable ABI.
On success, return a list of the items in object *o*, where each item is a tuple containing a key-value pair. On failure, return `NULL`.

Changed in version 3.7: Previously, the function returned a list or a tuple.

Iterator Protocol

There are two functions specifically for working with iterators.

int PyIter_Check(PyObject *o)

Part of the Stable ABI since version 3.8.
Return non-zero if the object *o* can be safely passed to **`PyIter_NextItem()`** and `0` otherwise. This function always succeeds.

int PyAIter_Check(PyObject *o)

Part of the Stable ABI since version 3.10.
Return non-zero if the object *o* provides the **`AsyncIterator`** protocol, and `0` otherwise. This function always succeeds.

Added in version 3.10.

int PyIter_NextItem(PyObject *iter, PyObject **item)

Part of the Stable ABI since version 3.14.
Return `1` and set *item* to a strong reference of the next value of the iterator *iter* on success. Return `0` and set *item* to `NULL` if there are no remaining values. Return `-1`, set *item* to `NULL` and set an exception on error.

Added in version 3.14.

PyObject *PyIter_Next(PyObject *o)

Return value: New reference. Part of the Stable ABI.
This is an older version of **`PyIter_NextItem()`**, which is retained for backwards compatibility. Prefer **`PyIter_NextItem()`**.

Return the next value from the iterator *o*. The object must be an iterator according to `PyIter_Check()` (it is up to the caller to check this). If there are no remaining values, returns `NULL` with no exception set. If an error occurs while retrieving the item, returns `NULL` and passes along the exception.

type PySendResult

The enum value used to represent different results of `PyIter_Send()`.

Added in version 3.10.

PySendResult PyIter_Send(PyObject *iter, PyObject *arg, PyObject **presult)

Part of the Stable ABI since version 3.10.

Sends the *arg* value into the iterator *iter*. Returns:

- `PYGEN_RETURN` if iterator returns. Return value is returned via *presult*.
- `PYGEN_NEXT` if iterator yields. Yielded value is returned via *presult*.
- `PYGEN_ERROR` if iterator has raised and exception. *presult* is set to `NULL`.

Added in version 3.10.

Buffer Protocol

Certain objects available in Python wrap access to an underlying memory array or *buffer*. Such objects include the built-in `bytes` and `bytearray`, and some extension types like `array.array`. Third-party libraries may define their own types for special purposes, such as image processing or numeric analysis.

While each of these types have their own semantics, they share the common characteristic of being backed by a possibly large memory buffer. It is then desirable, in some situations, to access that buffer directly and without intermediate copying.

Python provides such a facility at the C and Python level in the form of the buffer protocol. This protocol has two sides:

- on the producer side, a type can export a “buffer interface” which allows objects of that type to expose information about their underlying buffer. This interface is described in the section Buffer Object Structures; for Python see Emulating buffer types.
- on the consumer side, several means are available to obtain a pointer to the raw underlying data of an object (for example a method parameter). For Python see `memoryview`.

Simple objects such as `bytes` and `bytearray` expose their underlying buffer in byte-oriented form. Other forms are possible; for example, the elements exposed by an `array.array` can be multi-byte values.

An example consumer of the buffer interface is the `write()` method of file objects: any object that can export a series of bytes through the buffer interface can be written to a file. While `write()` only needs read-only access to the internal contents of the object passed to it, other methods such as `readinto()` need write access to the contents of their argument. The buffer interface allows objects to selectively allow or reject exporting of read-write and read-only buffers.

There are two ways for a consumer of the buffer interface to acquire a buffer over a target object:

- call `PyObject_GetBuffer()` with the right parameters;
- call `PyArg_ParseTuple()` (or one of its siblings) with one of the `y*`, `w*` or `s*` format codes.

In both cases, `PyBuffer_Release()` must be called when the buffer isn’t needed anymore. Failure to do so could lead to various issues such as resource leaks.

Added in version 3.12: The buffer protocol is now accessible in Python, see Emulating buffer types and `memoryview`.

Buffer structure

Buffer structures (or simply “buffers”) are useful as a way to expose the binary data from another object to the Python programmer. They can also be used as a zero-copy slicing mechanism. Using their ability to reference a block of memory, it is possible to expose any data to the Python programmer quite easily. The memory could be a large, constant array in a C extension, it could be a raw block of memory for manipulation before passing to an operating system library, or it could be used to pass around structured data in its native, in-memory format.

Contrary to most data types exposed by the Python interpreter, buffers are not `PyObject` pointers but rather simple C structures. This allows them to be created and copied very simply. When a generic wrapper around a buffer is needed, a memoryview object can be created.

For short instructions how to write an exporting object, see Buffer Object Structures. For obtaining a buffer, see `PyObject_GetBuffer()`.

type Py_buffer

Part of the Stable ABI (including all members) since version 3.11.

void *buf

A pointer to the start of the logical structure described by the buffer fields. This can be any location within the underlying physical memory block of the exporter. For example, with negative `strides` the value may point to the end of the memory block.

For contiguous arrays, the value points to the beginning of the memory block.

PyObject *obj

A new reference to the exporting object. The reference is owned by the consumer and automatically released (i.e. reference count decremented) and set to `NULL` by `PyBuffer_Release()`. The field is the equivalent of the return value of any standard C-API function.

As a special case, for *temporary* buffers that are wrapped by `PyMemoryView_FromBuffer()` or `PyBuffer_FillInfo()` this field is `NULL`. In general, exporting objects MUST NOT use this scheme.

Py_ssize_t len

`product(shape) * itemsize`. For contiguous arrays, this is the length of the underlying memory block. For non-contiguous arrays, it is the length that the logical structure would have if it were copied to a contiguous representation.

Accessing `((char *)buf)[0] up to ((char *)buf)[len-1]` is only valid if the buffer has been obtained by a request that guarantees contiguity. In most cases such a request will be **`PyBUF_SIMPLE`** or **`PyBUF_WRITABLE`**.

int readonly

An indicator of whether the buffer is read-only. This field is controlled by the **`PyBUF_WRITABLE`** flag.

Py_ssize_t itemsize

Item size in bytes of a single element. Same as the value of **`struct.calcsize()`** called on non-`NULL` **`format`** values.

Important exception: If a consumer requests a buffer without the **`PyBUF_FORMAT`** flag, **`format`** will be set to `NULL`, but **`itemsize`** still has the value for the original format.

If **`shape`** is present, the equality `product(shape) * itemsize == len` still holds and the consumer can use **`itemsize`** to navigate the buffer.

If **`shape`** is `NULL` as a result of a **`PyBUF_SIMPLE`** or a **`PyBUF_WRITABLE`** request, the consumer must disregard **`itemsize`** and assume `itemsize == 1`.

char *format

A *NULL* terminated string in **`struct`** module style syntax describing the contents of a single item. If this is `NULL`, `"B"` (unsigned bytes) is assumed.

This field is controlled by the **`PyBUF_FORMAT`** flag.

int ndim

The number of dimensions the memory represents as an n-dimensional array. If it is `0`, **`buf`** points to a single item representing a scalar. In this case, **`shape`**, **`strides`** and **`suboffsets`** MUST be `NULL`. The maximum number of dimensions is given by **`PyBUF_MAX_NDIM`**.

Py_ssize_t *shape

An array of **`Py_ssize_t`** of length **`ndim`** indicating the shape of the memory as an n-dimensional array. Note that `shape[0] * ... * shape[ndim-1] * itemsize` MUST be equal to **`len`**.

Shape values are restricted to `shape[n] >= 0`. The case `shape[n] == 0` requires special attention. See complex arrays for further information.

The shape array is read-only for the consumer.

Py_ssize_t *strides

An array of **`Py_ssize_t`** of length **`ndim`** giving the number of bytes to skip to get to a new element in each dimension.

Stride values can be any integer. For regular arrays, strides are usually positive, but a consumer MUST be able to handle the case `strides[n] <= 0`. See complex arrays for further information.

The strides array is read-only for the consumer.

Py_ssize_t *suboffsets

An array of **`Py_ssize_t`** of length **`ndim`**. If `suboffsets[n] >= 0`, the values stored along the nth dimension are pointers and the suboffset value dictates how many bytes to add to each pointer after de-referencing. A suboffset value that is negative indicates that no de-referencing should occur (striding in a contiguous memory block).

If all suboffsets are negative (i.e. no de-referencing is needed), then this field must be `NULL` (the default value).

This type of array representation is used by the Python Imaging Library (PIL). See complex arrays for further information how to access elements of such an array.

The suboffsets array is read-only for the consumer.

void *internal

This is for use internally by the exporting object. For example, this might be re-cast as an integer by the exporter and used to store flags about whether or not the shape, strides, and suboffsets arrays must be freed when the buffer is released. The consumer MUST NOT alter this value.

Constants:

PyBUF_MAX_NDIM

Part of the Stable ABI since version 3.11.
The maximum number of dimensions the memory represents. Exporters MUST respect this limit, consumers of multi-dimensional buffers SHOULD be able to handle up to **`PyBUF_MAX_NDIM`** dimensions. Currently set to 64.

Buffer request types

Buffers are usually obtained by sending a buffer request to an exporting object via **`PyObject_GetBuffer()`**. Since the complexity of the logical structure of the memory can vary drastically, the consumer uses the *flags* argument to specify the exact buffer type it can handle.

All **`Py_buffer`** fields are unambiguously defined by the request type.

request-independent fields

The following fields are not influenced by *flags* and must always be filled in with the correct values: `obj`, `buf`, `len`, `itemsize`, `ndim`.

readonly, format

PyBUF_WRITABLE

Part of the Stable ABI since version 3.11.
Controls the `readonly` field. If set, the exporter MUST provide a writable buffer or else report failure. Otherwise, the exporter MAY provide either a read-only or writable buffer, but the choice MUST be consistent for all consumers. For example, PyBUF_SIMPLE | PyBUF_WRITABLE can be used to request a simple writable buffer.

PyBUF_WRITEABLE

This is a soft deprecated alias to `PyBUF_WRITABLE`.

PyBUF_FORMAT

Part of the Stable ABI since version 3.11.
Controls the `format` field. If set, this field MUST be filled in correctly. Otherwise, this field MUST be `NULL`.

`PyBUF_WRITABLE` can be |'d to any of the flags in the next section. Since `PyBUF_SIMPLE` is defined as 0, `PyBUF_WRITABLE` can be used as a stand-alone flag to request a simple writable buffer.

`PyBUF_FORMAT` must be |'d to any of the flags except `PyBUF_SIMPLE`, because the latter already implies format `B` (unsigned bytes). **`PyBUF_FORMAT`** cannot be used on its own.

shape, strides, suboffsets

The flags that control the logical structure of the memory are listed in decreasing order of complexity. Note that each flag contains all bits of the flags below it.

Request	shape	strides	suboffsets
PyBUF_INDIRECT *Part of the Stable ABI since version 3.11.*	yes	yes	if needed
PyBUF_STRIDES *Part of the Stable ABI since version 3.11.*	yes	yes	NULL

Request	shape	strides	suboffsets
PyBUF_ND *Part of the Stable ABI since version 3.11.*	yes	NULL	NULL
PyBUF_SIMPLE *Part of the Stable ABI since version 3.11.*	NULL	NULL	NULL

contiguity requests

C or Fortran contiguity can be explicitly requested, with and without stride information. Without stride information, the buffer must be C-contiguous.

Request	shape	strides	suboffsets	contig
PyBUF_C_CONTIGUOUS *Part of the Stable ABI since version 3.11.*	yes	yes	NULL	C
PyBUF_F_CONTIGUOUS *Part of the Stable ABI since version 3.11.*	yes	yes	NULL	F
PyBUF_ANY_CONTIGUOUS *Part of the Stable ABI since version 3.11.*	yes	yes	NULL	C or F
`PyBUF_ND`	yes	NULL	NULL	C

compound requests

All possible requests are fully defined by some combination of the flags in the previous section. For convenience, the buffer protocol provides frequently used combinations as single flags.

In the following table *U* stands for undefined contiguity. The consumer would have to call `PyBuffer_IsContiguous()` to determine contiguity.

Request	**shape**	**strides**	**suboffsets**	**contig**	**readonly**	**format**
PyBUF_FULL *Part of the Stable ABI since version 3.11.*	yes	yes	if needed	U	0	yes
PyBUF_FULL_RO *Part of the Stable ABI since version 3.11.*	yes	yes	if needed	U	1 or 0	yes
PyBUF_RECORDS *Part of the Stable ABI since version 3.11.*	yes	yes	NULL	U	0	yes
PyBUF_RECORDS_RO *Part of the Stable ABI since version 3.11.*	yes	yes	NULL	U	1 or 0	yes
PyBUF_STRIDED *Part of the Stable ABI since version 3.11.*	yes	yes	NULL	U	0	NULL
PyBUF_STRIDED_RO *Part of the Stable ABI since version 3.11.*	yes	yes	NULL	U	1 or 0	NULL
PyBUF_CONTIG *Part of the Stable ABI since version 3.11.*	yes	NULL	NULL	C	0	NULL
PyBUF_CONTIG_RO *Part of the Stable ABI since version*	yes	NULL	NULL	C	1 or 0	NULL

Request	shape	strides	suboffsets	contig	readonly	format
3.11.						

Complex arrays

NumPy-style: shape and strides

The logical structure of NumPy-style arrays is defined by `itemsize`, `ndim`, `shape` and `strides`.

If `ndim == 0`, the memory location pointed to by `buf` is interpreted as a scalar of size `itemsize`. In that case, both `shape` and `strides` are `NULL`.

If `strides` is `NULL`, the array is interpreted as a standard n-dimensional C-array. Otherwise, the consumer must access an n-dimensional array as follows:

```
ptr = (char *)buf + indices[0] * strides[0] + ... + indices[n-1] * strides[n-1];
item = *((typeof(item) *)ptr);
```

As noted above, `buf` can point to any location within the actual memory block. An exporter can check the validity of a buffer with this function:

```
def verify_structure(memlen, itemsize, ndim, shape, strides, offset):
    """Verify that the parameters represent a valid array within
       the bounds of the allocated memory:
           char *mem: start of the physical memory block
           memlen: length of the physical memory block
           offset: (char *)buf - mem
    """
    if offset % itemsize:
        return False
    if offset < 0 or offset+itemsize > memlen:
        return False
    if any(v % itemsize for v in strides):
        return False

    if ndim <= 0:
        return ndim == 0 and not shape and not strides
    if 0 in shape:
        return True

    imin = sum(strides[j]*(shape[j]-1) for j in range(ndim)
               if strides[j] <= 0)
    imax = sum(strides[j]*(shape[j]-1) for j in range(ndim)
               if strides[j] > 0)

    return 0 <= offset+imin and offset+imax+itemsize <= memlen
```

PIL-style: shape, strides and suboffsets

In addition to the regular items, PIL-style arrays can contain pointers that must be followed in order to get to the next element in a dimension. For example, the regular three-dimensional C-array `char v[2][2][3]` can also be viewed as an array of 2 pointers to 2 two-dimensional arrays: `char (*v[2])[2][3]`. In suboffsets representation, those two pointers can be embedded at the start of **`buf`**, pointing to two `char x[2][3]` arrays that can be located anywhere in memory.

Here is a function that returns a pointer to the element in an N-D array pointed to by an N-dimensional index when there are both non-`NULL` strides and suboffsets:

```
void *get_item_pointer(int ndim, void *buf, Py_ssize_t *strides,
                       Py_ssize_t *suboffsets, Py_ssize_t *indices) {
    char *pointer = (char*)buf;
    int i;
    for (i = 0; i < ndim; i++) {
        pointer += strides[i] * indices[i];
        if (suboffsets[i] >=0 ) {
            pointer = *((char**)pointer) + suboffsets[i];
        }
    }
    return (void*)pointer;
}
```

Buffer-related functions

int PyObject_CheckBuffer(PyObject *obj)

Part of the Stable ABI since version 3.11.
Return `1` if *obj* supports the buffer interface otherwise `0`. When `1` is returned, it doesn't guarantee that **`PyObject_GetBuffer()`** will succeed. This function always succeeds.

int PyObject_GetBuffer(PyObject *exporter, Py_buffer *view, int flags)

Part of the Stable ABI since version 3.11.
Send a request to *exporter* to fill in *view* as specified by *flags*. If the exporter cannot provide a buffer of the exact type, it MUST raise **`BufferError`**, set `view->obj` to `NULL` and return `-1`.

On success, fill in *view*, set `view->obj` to a new reference to *exporter* and return 0. In the case of chained buffer providers that redirect requests to a single object, `view->obj` MAY refer to this object instead of *exporter* (See Buffer Object Structures).

Successful calls to **`PyObject_GetBuffer()`** must be paired with calls to **`PyBuffer_Release()`**, similar to **`malloc()`** and **`free()`**. Thus, after the consumer is done with the buffer, **`PyBuffer_Release()`** must be called exactly once.

void PyBuffer_Release(Py_buffer *view)

Part of the Stable ABI since version 3.11.

Release the buffer *view* and release the strong reference (i.e. decrement the reference count) to the view's supporting object, `view->obj`. This function MUST be called when the buffer is no longer being used, otherwise reference leaks may occur.

It is an error to call this function on a buffer that was not obtained via **`PyObject_GetBuffer()`**.

Py_ssize_t PyBuffer_SizeFromFormat(const char *format)

Part of the Stable ABI since version 3.11.
Return the implied **`itemsize`** from **`format`**. On error, raise an exception and return -1.

Added in version 3.9.

int PyBuffer_IsContiguous(const Py_buffer *view, char order)

Part of the Stable ABI since version 3.11.
Return `1` if the memory defined by the *view* is C-style (*order* is `'C'`) or Fortran-style (*order* is `'F'`) contiguous or either one (*order* is `'A'`). Return `0` otherwise. This function always succeeds.

void *PyBuffer_GetPointer(const Py_buffer *view, const Py_ssize_t *indices)

Part of the Stable ABI since version 3.11.
Get the memory area pointed to by the *indices* inside the given *view*. *indices* must point to an array of `view->ndim` indices.

int PyBuffer_FromContiguous(const Py_buffer *view, const void *buf, Py_ssize_t len, char fort)

Part of the Stable ABI since version 3.11.
Copy contiguous *len* bytes from *buf* to *view*. *fort* can be `'C'` or `'F'` (for C-style or Fortran-style ordering). `0` is returned on success, `-1` on error.

int PyBuffer_ToContiguous(void *buf, const Py_buffer *src, Py_ssize_t len, char order)

Part of the Stable ABI since version 3.11.
Copy *len* bytes from *src* to its contiguous representation in *buf*. *order* can be `'C'` or `'F'` or `'A'` (for C-style or Fortran-style ordering or either one). `0` is returned on success, `-1` on error.

This function fails if *len* != *src->len*.

int PyObject_CopyData(PyObject *dest, PyObject *src)

Part of the Stable ABI since version 3.11.
Copy data from *src* to *dest* buffer. Can convert between C-style and or Fortran-style buffers.

`0` is returned on success, `-1` on error.

void PyBuffer_FillContiguousStrides(int ndims, Py_ssize_t *shape, Py_ssize_t *strides, int itemsize, char order)

Part of the Stable ABI since version 3.11.
Fill the *strides* array with byte-strides of a contiguous (C-style if *order* is `'C'` or Fortran-style if *order* is `'F'`) array of the given shape with the given number of bytes per element.

int PyBuffer_FillInfo(Py_buffer *view, PyObject *exporter, void *buf, Py_ssize_t len, int readonly, int flags)

Part of the Stable ABI since version 3.11.
Handle buffer requests for an exporter that wants to expose *buf* of size *len* with writability set according to *readonly*. *buf* is interpreted as a sequence of unsigned bytes.

The *flags* argument indicates the request type. This function always fills in *view* as specified by flags, unless *buf* has been designated as read-only and **`PyBUF_WRITABLE`** is set in *flags*.

On success, set `view->obj` to a new reference to *exporter* and return 0. Otherwise, raise **`BufferError`**, set `view->obj` to `NULL` and return `-1`;

If this function is used as part of a getbufferproc, *exporter* MUST be set to the exporting object and *flags* must be passed unmodified. Otherwise, *exporter* MUST be `NULL`.

Concrete Objects Layer

The functions in this chapter are specific to certain Python object types. Passing them an object of the wrong type is not a good idea; if you receive an object from a Python program and you are not sure that it has the right type, you must perform a type check first; for example, to check that an object is a dictionary, use `PyDict_Check()`. The chapter is structured like the “family tree” of Python object types.

Warning

While the functions described in this chapter carefully check the type of the objects which are passed in, many of them do not check for `NULL` being passed instead of a valid object. Allowing `NULL` to be passed in can cause memory access violations and immediate termination of the interpreter.

Fundamental Objects

This section describes Python type objects and the singleton object `None`.

- Type Objects
    - Creating Heap-Allocated Types
- The `None` Object

Numeric Objects

- Integer Objects
    - Export API

    - PyLongWriter API
    - Deprecated API
- Boolean Objects
- Floating-Point Objects
    - Pack and Unpack functions
        - Pack functions
        - Unpack functions
- Complex Number Objects
    - Complex Numbers as C Structures
    - Complex Numbers as Python Objects

Sequence Objects

Generic operations on sequence objects were discussed in the previous chapter; this section deals with the specific kinds of sequence objects that are intrinsic to the Python language.

- Bytes Objects
- Byte Array Objects
    - Type check macros
    - Direct API functions
    - Macros
- Unicode Objects and Codecs
    - Unicode Objects
        - Unicode Type
        - Unicode Character Properties
        - Creating and accessing Unicode strings
        - Locale Encoding
        - File System Encoding
        - wchar_t Support
    - Built-in Codecs
        - Generic Codecs
        - UTF-8 Codecs
        - UTF-32 Codecs

        - UTF-16 Codecs
        - UTF-7 Codecs
        - Unicode-Escape Codecs
        - Raw-Unicode-Escape Codecs
        - Latin-1 Codecs
        - ASCII Codecs
        - Character Map Codecs
        - MBCS codecs for Windows
    - Methods and Slot Functions
    - PyUnicodeWriter
    - Deprecated API
- Tuple Objects
- Struct Sequence Objects
- List Objects

Container Objects

- Dictionary Objects
    - Dictionary View Objects
    - Ordered Dictionaries
- Set Objects
    - Deprecated API

Function Objects

- Function Objects
- Instance Method Objects
- Method Objects
- Cell Objects
- Code Objects
- Code Object Flags
- Extra information

Other Objects

- File Objects
    - Deprecated API
- Module Objects
- Module definitions
    - Module slots
- Creating extension modules dynamically
- Support functions
    - Module lookup (single-phase initialization)
- Iterator Objects
    - Range Objects
    - Builtin Iterator Types
    - Other Iterator Objects
- Descriptor Objects
    - Built-in descriptors
- Slice Objects
    - Ellipsis Object
- MemoryView objects
- Pickle buffer objects
- Weak Reference Objects
- Capsules
- Frame Objects
    - Frame Locals Proxies
    - Legacy Local Variable APIs
    - Internal Frames
- Generator Objects
    - Asynchronous Generator Objects
    - Deprecated API
- Coroutine Objects
- Context Variables Objects
- Objects for Type Hinting

C API for extension modules

- Curses C API
- Internal data

- DateTime Objects
- Internal data

Type Objects

type PyTypeObject

Part of the Limited API (as an opaque struct).
The C structure of the objects used to describe built-in types.

PyTypeObject PyType_Type

Part of the Stable ABI.
This is the type object for type objects; it is the same object as **`type`** in the Python layer.

int PyType_Check(PyObject *o)

Return non-zero if the object *o* is a type object, including instances of types derived from the standard type object. Return 0 in all other cases. This function always succeeds.

int PyType_CheckExact(PyObject *o)

Return non-zero if the object *o* is a type object, but not a subtype of the standard type object. Return 0 in all other cases. This function always succeeds.

unsigned int PyType_ClearCache()

Part of the Stable ABI.
Clear the internal lookup cache. Return the current version tag.

unsigned long PyType_GetFlags(PyTypeObject *type)

Part of the Stable ABI.
Return the **`tp_flags`** member of *type*. This function is primarily meant for use with `Py_LIMITED_API`; the individual flag bits are guaranteed to be stable across Python

releases, but access to **`tp_flags`** itself is not part of the limited API.

Added in version 3.2.

Changed in version 3.4: The return type is now `unsigned long` rather than `long`.

PyObject *PyType_GetDict(PyTypeObject *type)

Return the type object's internal namespace, which is otherwise only exposed via a read-only proxy (**`cls.__dict__`**). This is a replacement for accessing **`tp_dict`** directly. The returned dictionary must be treated as read-only.

This function is meant for specific embedding and language-binding cases, where direct access to the dict is necessary and indirect access (e.g. via the proxy or **`PyObject_GetAttr()`**) isn't adequate.

Extension modules should continue to use `tp_dict`, directly or indirectly, when setting up their own types.

Added in version 3.12.

void PyType_Modified(PyTypeObject *type)

Part of the Stable ABI.
Invalidate the internal lookup cache for the type and all of its subtypes. This function must be called after any manual modification of the attributes or base classes of the type.

int PyType_AddWatcher(PyType_WatchCallback callback)

Register *callback* as a type watcher. Return a non-negative integer ID which must be passed to future calls to **`PyType_Watch()`**. In case of error (e.g. no more watcher IDs available), return `-1` and set an exception.

In free-threaded builds, **`PyType_AddWatcher()`** is not thread-safe, so it must be called at start up (before spawning the first thread).

Added in version 3.12.

int PyType_ClearWatcher(int watcher_id)

Clear watcher identified by *watcher_id* (previously returned from **`PyType_AddWatcher()`**). Return `0` on success, `-1` on error (e.g. if *watcher_id* was never registered.)

An extension should never call `PyType_ClearWatcher` with a *watcher_id* that was not returned to it by a previous call to **`PyType_AddWatcher()`**.

Added in version 3.12.

int PyType_Watch(int watcher_id, PyObject *type)

Mark *type* as watched. The callback granted *watcher_id* by **`PyType_AddWatcher()`** will be called whenever **`PyType_Modified()`** reports a change to *type*. (The callback may be called only once for a series of consecutive modifications to *type*, if **`_PyType_Lookup()`** is not called on *type* between the modifications; this is an implementation detail and subject to change.)

An extension should never call `PyType_Watch` with a *watcher_id* that was not returned to it by a previous call to **`PyType_AddWatcher()`**.

Added in version 3.12.

int PyType_Unwatch(int watcher_id, PyObject *type)

Mark *type* as not watched. This undoes a previous call to **`PyType_Watch()`**. *type* must not be `NULL`.

An extension should never call this function with a *watcher_id* that was not returned to it by a previous call to **`PyType_AddWatcher()`**.

On success, this function returns `0`. On failure, this function returns `-1` with an exception set.

Added in version 3.12.

typedef int (*PyType_WatchCallback)(PyObject *type)

Type of a type-watcher callback function.

The callback must not modify *type* or cause **`PyType_Modified()`** to be called on *type* or any type in its MRO; violating this rule could cause infinite recursion.

Added in version 3.12.

int PyType_HasFeature(PyTypeObject *o, int feature)

Return non-zero if the type object *o* sets the feature *feature*. Type features are denoted by single bit flags.

int PyType_FastSubclass(PyTypeObject *type, int flag)

Return non-zero if the type object *type* sets the subclass flag *flag*. Subclass flags are denoted by **`Py_TPFLAGS_*_SUBCLASS`**. This function is used by many `_Check` functions for common types.

See also

`PyObject_TypeCheck()`, which is used as a slower alternative in `_Check` functions for types that don’t come with subclass flags.

int PyType_IS_GC(PyTypeObject *o)

Return true if the type object includes support for the cycle detector; this tests the type flag **`Py_TPFLAGS_HAVE_GC`**.

int PyType_IsSubtype(PyTypeObject *a, PyTypeObject *b)

Part of the Stable ABI.
Return true if *a* is a subtype of *b*.

This function only checks for actual subtypes, which means that **`__subclasscheck__()`** is not called on *b*. Call **`PyObject_IsSubclass()`** to do the same check that **`issubclass()`** would do.

PyObject *PyType_GenericAlloc(PyTypeObject *type, Py_ssize_t nitems)

Return value: New reference. Part of the Stable ABI.
Generic handler for the **`tp_alloc`** slot of a type object. Uses Python's default memory allocation mechanism to allocate memory for a new instance, zeros the memory, then initializes the memory as if by calling **`PyObject_Init()`** or **`PyObject_InitVar()`**.

Do not call this directly to allocate memory for an object; call the type's **`tp_alloc`** slot instead.

For types that support garbage collection (i.e., the **`Py_TPFLAGS_HAVE_GC`** flag is set), this function behaves like **`PyObject_GC_New`** or **`PyObject_GC_NewVar`** (except the memory is guaranteed to be zeroed before initialization), and should be paired with **`PyObject_GC_Del()`** in **`tp_free`**. Otherwise, it behaves like **`PyObject_New`** or **`PyObject_NewVar`** (except the memory is guaranteed to be zeroed before initialization) and should be paired with **`PyObject_Free()`** in **`tp_free`**.

PyObject *PyType_GenericNew(PyTypeObject *type, PyObject *args, PyObject *kwds)

Return value: New reference. Part of the Stable ABI.
Generic handler for the `tp_new` slot of a type object. Creates a new instance using the type’s `tp_alloc` slot and returns the resulting object.

int PyType_Ready(PyTypeObject *type)

Part of the Stable ABI.
Finalize a type object. This should be called on all type objects to finish their initialization. This function is responsible for adding inherited slots from a type’s base class. Return `0` on success, or return `-1` and sets an exception on error.

Note

If some of the base classes implements the GC protocol and the provided type does not include the `Py_TPFLAGS_HAVE_GC` in its flags, then the GC protocol will be automatically implemented from its parents. On the contrary, if the type being created does include `Py_TPFLAGS_HAVE_GC` in its flags then it **must** implement the GC protocol itself by at least implementing the `tp_traverse` handle.

PyObject *PyType_GetName(PyTypeObject *type)

Return value: New reference. Part of the Stable ABI since version 3.11.
Return the type’s name. Equivalent to getting the type’s `__name__` attribute.

Added in version 3.11.

PyObject *PyType_GetQualName(PyTypeObject *type)

Return value: New reference. Part of the Stable ABI since version 3.11.
Return the type’s qualified name. Equivalent to getting the type’s **`__qualname__`** attribute.

Added in version 3.11.

PyObject *PyType_GetFullyQualifiedName(PyTypeObject *type)

Part of the Stable ABI since version 3.13.
Return the type’s fully qualified name. Equivalent to `f"{type.__module__}.{type.__qualname__}"`, or **`type.__qualname__`** if **`type.__module__`** is not a string or is equal to `"builtins"`.

Added in version 3.13.

PyObject *PyType_GetModuleName(PyTypeObject *type)

Part of the Stable ABI since version 3.13.
Return the type’s module name. Equivalent to getting the **`type.__module__`** attribute.

Added in version 3.13.

void *PyType_GetSlot(PyTypeObject *type, int slot)

Part of the Stable ABI since version 3.4.
Return the function pointer stored in the given slot. If the result is `NULL`, this indicates that either the slot is `NULL`, or that the function was called with invalid parameters. Callers will typically cast the result pointer into the appropriate function type.

See **`PyType_Slot.slot`** for possible values of the *slot* argument.

Added in version 3.4.

Changed in version 3.10: **`PyType_GetSlot()`** can now accept all types. Previously, it was limited to heap types.

PyObject *PyType_GetModule(PyTypeObject *type)

Part of the Stable ABI since version 3.10.
Return the module object associated with the given type when the type was created using **`PyType_FromModuleAndSpec()`**.

If no module is associated with the given type, sets **`TypeError`** and returns `NULL`.

This function is usually used to get the module in which a method is defined. Note that in such a method, `PyType_GetModule(Py_TYPE(self))` may not return the intended result. `Py_TYPE(self)` may be a *subclass* of the intended class, and subclasses are not necessarily defined in the same module as their superclass. See **`PyCMethod`** to get the class that defines the method. See **`PyType_GetModuleByDef()`** for cases when **`PyCMethod`** cannot be used.

Added in version 3.9.

void *PyType_GetModuleState(PyTypeObject *type)

Part of the Stable ABI since version 3.10.
Return the state of the module object associated with the given type. This is a shortcut for calling **`PyModule_GetState()`** on the result of **`PyType_GetModule()`**.

If no module is associated with the given type, sets **`TypeError`** and returns `NULL`.

If the *type* has an associated module but its state is `NULL`, returns `NULL` without setting an exception.

Added in version 3.9.

PyObject *PyType_GetModuleByDef(PyTypeObject *type, struct PyModuleDef *def)

Return value: Borrowed reference. Part of the Stable ABI since version 3.13.
Find the first superclass whose module was created from the given **`PyModuleDef`** *def*, and return that module.

If no module is found, raises a **`TypeError`** and returns `NULL`.

This function is intended to be used together with **`PyModule_GetState()`** to get module state from slot methods (such as **`tp_init`** or **`nb_add`**) and other places where a method's defining class cannot be passed using the **`PyCMethod`** calling convention.

The returned reference is borrowed from *type*, and will be valid as long as you hold a reference to *type*. Do not release it with **`Py_DECREF()`** or similar.

Added in version 3.11.

int PyType_GetBaseByToken(PyTypeObject *type, void *token, PyTypeObject **result)

Part of the Stable ABI since version 3.14.
Find the first superclass in *type*'s method resolution order whose **`Py_tp_token`** token is equal to the given one.

- If found, set **result* to a new strong reference to it and return `1`.
- If not found, set **result* to `NULL` and return `0`.
- On error, set **result* to `NULL` and return `-1` with an exception set.

The *result* argument may be `NULL`, in which case **result* is not set. Use this if you need only the return value.

The *token* argument may not be `NULL`.

Added in version 3.14.

int PyUnstable_Type_AssignVersionTag(PyTypeObject *type)

This is Unstable API. It may change without warning in minor releases.

Attempt to assign a version tag to the given type.

Returns 1 if the type already had a valid version tag or a new one was assigned, or 0 if a new tag could not be assigned.

Added in version 3.12.

int PyType_SUPPORTS_WEAKREFS(PyTypeObject *type)

Return true if instances of *type* support creating weak references, false otherwise. This function always succeeds. *type* must not be `NULL`.

See also

- Weak Reference Objects
- **`weakref`**

Creating Heap-Allocated Types

The following functions and structs are used to create heap types.

PyObject *PyType_FromMetaclass(PyTypeObject *metaclass, PyObject *module, PyType_Spec *spec, PyObject *bases)

Part of the Stable ABI since version 3.12.

Create and return a heap type from the *spec* (see **`Py_TPFLAGS_HEAPTYPE`**).

The metaclass *metaclass* is used to construct the resulting type object. When *metaclass* is `NULL`, the metaclass is derived from *bases* (or *Py_tp_base[s]* slots if *bases* is `NULL`, see below).

Metaclasses that override **`tp_new`** are not supported, except if `tp_new` is `NULL`.

The *bases* argument can be used to specify base classes; it can either be only one class or a tuple of classes. If *bases* is `NULL`, the **`Py_tp_bases`** slot is used instead. If that also is `NULL`, the **`Py_tp_base`** slot is used instead. If that also is `NULL`, the new type derives from **`object`**.

The *module* argument can be used to record the module in which the new class is defined. It must be a module object or `NULL`. If not `NULL`, the module is associated with the new type and can later be retrieved with **`PyType_GetModule()`**. The associated module is not inherited by subclasses; it must be specified for each class individually.

This function calls **`PyType_Ready()`** on the new type.

Note that this function does *not* fully match the behavior of calling **`type()`** or using the **`class`** statement. With user-provided base types or metaclasses, prefer calling **`type`** (or the metaclass) over `PyType_From*` functions. Specifically:

- **`__new__()`** is not called on the new class (and it must be set to `type.__new__`).
- **`__init__()`** is not called on the new class.
- **`__init_subclass__()`** is not called on any bases.

- **`__set_name__()`** is not called on new descriptors.

  Added in version 3.12.

PyObject *PyType_FromModuleAndSpec(PyObject *module, PyType_Spec *spec, PyObject *bases)

Return value: New reference. Part of the Stable ABI since version 3.10.
Equivalent to `PyType_FromMetaclass(NULL, module, spec, bases)`.

Added in version 3.9.

Changed in version 3.10: The function now accepts a single class as the *bases* argument and `NULL` as the `tp_doc` slot.

Changed in version 3.12: The function now finds and uses a metaclass corresponding to the provided base classes. Previously, only **`type`** instances were returned.

The **`tp_new`** of the metaclass is *ignored*. which may result in incomplete initialization. Creating classes whose metaclass overrides **`tp_new`** is deprecated.

Changed in version 3.14: Creating classes whose metaclass overrides **`tp_new`** is no longer allowed.

PyObject *PyType_FromSpecWithBases(PyType_Spec *spec, PyObject *bases)

Return value: New reference. Part of the Stable ABI since version 3.3.
Equivalent to `PyType_FromMetaclass(NULL, NULL, spec, bases)`.

Added in version 3.3.

Changed in version 3.12: The function now finds and uses a metaclass corresponding to the provided base classes. Previously, only **`type`** instances were returned.

The **`tp_new`** of the metaclass is *ignored*. which may result in incomplete initialization. Creating classes whose metaclass overrides **`tp_new`** is deprecated.

Changed in version 3.14: Creating classes whose metaclass overrides **`tp_new`** is no longer allowed.

PyObject *PyType_FromSpec(PyType_Spec *spec)

Return value: New reference. Part of the Stable ABI.
Equivalent to `PyType_FromMetaclass(NULL, NULL, spec, NULL)`.

Changed in version 3.12: The function now finds and uses a metaclass corresponding to the base classes provided in *Py_tp_base[s]* slots. Previously, only **`type`** instances were returned.

The **`tp_new`** of the metaclass is *ignored*. which may result in incomplete initialization. Creating classes whose metaclass overrides **`tp_new`** is deprecated.

Changed in version 3.14: Creating classes whose metaclass overrides **`tp_new`** is no longer allowed.

int PyType_Freeze(PyTypeObject *type)

Part of the Stable ABI since version 3.14.
Make a type immutable: set the **`Py_TPFLAGS_IMMUTABLETYPE`** flag.

All base classes of *type* must be immutable.

On success, return `0`. On error, set an exception and return `-1`.

The type must not be used before it’s made immutable. For example, type instances must not be created before the type is made immutable.

Added in version 3.14.

type PyType_Spec

Part of the Stable ABI (including all members).
Structure defining a type’s behavior.

const char *name

Name of the type, used to set `PyTypeObject.tp_name`.

int basicsize

If positive, specifies the size of the instance in bytes. It is used to set `PyTypeObject.tp_basicsize`.

If zero, specifies that `tp_basicsize` should be inherited.

If negative, the absolute value specifies how much space instances of the class need *in addition* to the superclass. Use `PyObject_GetTypeData()` to get a pointer to subclass-specific memory reserved this way. For negative **`basicsize`**, Python will insert padding when needed to meet `tp_basicsize`’s alignment requirements.

Changed in version 3.12: Previously, this field could not be negative.

int itemsize

Size of one element of a variable-size type, in bytes. Used to set `PyTypeObject.tp_itemsize`. See `tp_itemsize` documentation for caveats.

If zero, `tp_itemsize` is inherited. Extending arbitrary variable-sized classes is dangerous, since some types use a fixed offset for variable-sized memory, which can then overlap fixed-sized memory used by a subclass. To help prevent mistakes, inheriting `itemsize` is only possible in the following situations:

- The base is not variable-sized (its **`tp_itemsize`**).
- The requested **`PyType_Spec.basicsize`** is positive, suggesting that the memory layout of the base class is known.
- The requested **`PyType_Spec.basicsize`** is zero, suggesting that the subclass does not access the instance's memory directly.
- With the **`Py_TPFLAGS_ITEMS_AT_END`** flag.

unsigned int flags

Type flags, used to set **`PyTypeObject.tp_flags`**.

If the `Py_TPFLAGS_HEAPTYPE` flag is not set, **`PyType_FromSpecWithBases()`** sets it automatically.

PyType_Slot *slots

Array of **`PyType_Slot`** structures. Terminated by the special slot value `{0, NULL}`.

Each slot ID should be specified at most once.

type PyType_Slot

Part of the Stable ABI (including all members).
Structure defining optional functionality of a type, containing a slot ID and a value pointer.

int slot

A slot ID.

Slot IDs are named like the field names of the structures **`PyTypeObject`**, **`PyNumberMethods`**, `PySequenceMethods`, `PyMappingMethods` and `PyAsyncMethods` with an added `Py_` prefix. For example, use:

- `Py_tp_dealloc` to set `PyTypeObject.tp_dealloc`
- `Py_nb_add` to set `PyNumberMethods.nb_add`
- `Py_sq_length` to set `PySequenceMethods.sq_length`

An additional slot is supported that does not correspond to a **`PyTypeObject`** struct field:

- `Py_tp_token`

The following “offset” fields cannot be set using `PyType_Slot`:

- `tp_weaklistoffset` (use `Py_TPFLAGS_MANAGED_WEAKREF` instead if possible)
- `tp_dictoffset` (use `Py_TPFLAGS_MANAGED_DICT` instead if possible)
- `tp_vectorcall_offset` (use `"__vectorcalloffset__"` in PyMemberDef)

If it is not possible to switch to a `MANAGED` flag (for example, for vectorcall or to support Python older than 3.12), specify the offset in `Py_tp_members`. See PyMemberDef documentation for details.

The following internal fields cannot be set at all when creating a heap type:

- `tp_dict`, `tp_mro`, `tp_cache`, `tp_subclasses`, and `tp_weaklist`.

Setting `Py_tp_bases` or `Py_tp_base` may be problematic on some platforms. To avoid issues, use the *bases* argument of `PyType_FromSpecWithBases()` instead.

Changed in version 3.9: Slots in `PyBufferProcs` may be set in the unlimited API.

Changed in version 3.11: **`bf_getbuffer`** and **`bf_releasebuffer`** are now available under the limited API.

Changed in version 3.14: The field **`tp_vectorcall`** can now be set using **`Py_tp_vectorcall`**. See the field’s documentation for details.

void *pfunc

The desired value of the slot. In most cases, this is a pointer to a function.

pfunc values may not be `NULL`, except for the following slots:

- **`Py_tp_doc`**
- **`Py_tp_token`** (for clarity, prefer **`Py_TP_USE_SPEC`** rather than `NULL`)

Py_tp_token

Part of the Stable ABI since version 3.14.
A **`slot`** that records a static memory layout ID for a class.

If the **`PyType_Spec`** of the class is statically allocated, the token can be set to the spec using the special value **`Py_TP_USE_SPEC`**:

```
static PyType_Slot foo_slots[] = {
   {Py_tp_token, Py_TP_USE_SPEC},
```

It can also be set to an arbitrary pointer, but you must ensure that:

- The pointer outlives the class, so it’s not reused for something else while the class exists.
- It “belongs” to the extension module where the class lives, so it will not clash with other extensions.

Use **`PyType_GetBaseByToken()`** to check if a class's superclass has a given token – that is, check whether the memory layout is compatible.

To get the token for a given class (without considering superclasses), use **`PyType_GetSlot()`** with `Py_tp_token`.

Added in version 3.14.

Py_TP_USE_SPEC

Part of the Stable ABI since version 3.14.

Used as a value with **`Py_tp_token`** to set the token to the class's **`PyType_Spec`**. Expands to `NULL`.

Added in version 3.14.

The `None` Object

Note that the **`PyTypeObject`** for `None` is not directly exposed in the Python/C API. Since `None` is a singleton, testing for object identity (using `==` in C) is sufficient. There is no **`PyNone_Check()`** function for the same reason.

PyObject *Py_None

The Python `None` object, denoting lack of value. This object has no methods and is immortal.

Changed in version 3.12: **`Py_None`** is immortal.

Py_RETURN_NONE

Return **`Py_None`** from a function.

Integer Objects

All integers are implemented as “long” integer objects of arbitrary size.

On error, most `PyLong_As*` APIs return `(return type)-1` which cannot be distinguished from a number. Use **`PyErr_Occurred()`** to disambiguate.

type PyLongObject

Part of the Limited API (as an opaque struct).
This subtype of **`PyObject`** represents a Python integer object.

PyTypeObject PyLong_Type

Part of the Stable ABI.
This instance of **`PyTypeObject`** represents the Python integer type. This is the same object as **`int`** in the Python layer.

int PyLong_Check(PyObject *p)

Return true if its argument is a **`PyLongObject`** or a subtype of **`PyLongObject`**. This function always succeeds.

int PyLong_CheckExact(PyObject *p)

Return true if its argument is a **`PyLongObject`**, but not a subtype of **`PyLongObject`**. This function always succeeds.

PyObject *PyLong_FromLong(long v)

Return value: New reference. Part of the Stable ABI.
Return a new **`PyLongObject`** object from *v*, or `NULL` on failure.

CPython implementation detail: CPython keeps an array of integer objects for all integers between `-5` and `256`. When you create an int in that range you actually just get back a reference to the existing object.

PyObject *PyLong_FromUnsignedLong(unsigned long v)

Return value: New reference. Part of the Stable ABI.
Return a new **`PyLongObject`** object from a C unsigned long, or `NULL` on failure.

PyObject *PyLong_FromSsize_t(Py_ssize_t v)

Return value: New reference. Part of the Stable ABI.
Return a new **`PyLongObject`** object from a C **`Py_ssize_t`**, or `NULL` on failure.

PyObject *PyLong_FromSize_t(size_t v)

Return value: New reference. Part of the Stable ABI.
Return a new **`PyLongObject`** object from a C **`size_t`**, or `NULL` on failure.

PyObject *PyLong_FromLongLong(long long v)

Return value: New reference. Part of the Stable ABI.
Return a new **`PyLongObject`** object from a C long long, or `NULL` on failure.

PyObject *PyLong_FromInt32(int32_t value)
PyObject *PyLong_FromInt64(int64_t value)

Part of the Stable ABI since version 3.14.
Return a new **`PyLongObject`** object from a signed C int32_t or int64_t, or `NULL` with an exception set on failure.

Added in version 3.14.

PyObject *PyLong_FromUnsignedLongLong(unsigned long long v)

Return value: New reference. Part of the Stable ABI.
Return a new `PyLongObject` object from a C unsigned long long, or `NULL` on failure.

PyObject *PyLong_FromUInt32(uint32_t value)
PyObject *PyLong_FromUInt64(uint64_t value)

Part of the Stable ABI since version 3.14.
Return a new `PyLongObject` object from an unsigned C uint32_t or uint64_t, or `NULL` with an exception set on failure.

Added in version 3.14.

PyObject *PyLong_FromDouble(double v)

Return value: New reference. Part of the Stable ABI.
Return a new `PyLongObject` object from the integer part of *v*, or `NULL` on failure.

PyObject *PyLong_FromString(const char *str, char **pend, int base)

Return value: New reference. Part of the Stable ABI.
Return a new `PyLongObject` based on the string value in *str*, which is interpreted according to the radix in *base*, or `NULL` on failure. If *pend* is non-`NULL`, **pend* will point to the end of *str* on success or to the first character that could not be processed on error. If *base* is `0`, *str* is interpreted using the Integer literals definition; in this case, leading zeros in a non-zero decimal number raises a `ValueError`. If *base* is not `0`, it must be between `2` and `36`, inclusive. Leading and trailing whitespace and single underscores after a base specifier and between digits are ignored. If there are no digits or *str* is not NULL-terminated following the digits and trailing whitespace, `ValueError` will be raised.

See also

`PyLong_AsNativeBytes()` and `PyLong_FromNativeBytes()` functions can be used to convert a `PyLongObject` to/from an array of bytes in base `256`.

PyObject *PyLong_FromUnicodeObject(PyObject *u, int base)

Return value: New reference.
Convert a sequence of Unicode digits in the string *u* to a Python integer value.

Added in version 3.3.

PyObject *PyLong_FromVoidPtr(void *p)

Return value: New reference. Part of the Stable ABI.
Create a Python integer from the pointer *p*. The pointer value can be retrieved from the resulting value using `PyLong_AsVoidPtr()`.

PyObject *PyLong_FromNativeBytes(const void *buffer, size_t n_bytes, int flags)

Part of the Stable ABI since version 3.14.
Create a Python integer from the value contained in the first *n_bytes* of *buffer*, interpreted as a two's-complement signed number.

flags are as for `PyLong_AsNativeBytes()`. Passing `-1` will select the native endian that CPython was compiled with and assume that the most-significant bit is a sign bit. Passing `Py_ASNATIVEBYTES_UNSIGNED_BUFFER` will produce the same result as calling `PyLong_FromUnsignedNativeBytes()`. Other flags are ignored.

Added in version 3.13.

PyObject *PyLong_FromUnsignedNativeBytes(const void *buffer, size_t n_bytes, int flags)

Part of the Stable ABI since version 3.14.
Create a Python integer from the value contained in the first *n_bytes* of *buffer*, interpreted as an unsigned number.

flags are as for `PyLong_AsNativeBytes()`. Passing `-1` will select the native endian that CPython was compiled with and assume that the most-significant bit is not a sign bit. Flags other than endian are ignored.

Added in version 3.13.

PyLong_FromPid(pid)

Macro for creating a Python integer from a process identifier.

This can be defined as an alias to `PyLong_FromLong()` or `PyLong_FromLongLong()`, depending on the size of the system's PID type.

Added in version 3.2.

long PyLong_AsLong(PyObject *obj)

Part of the Stable ABI.
Return a C long representation of *obj*. If *obj* is not an instance of `PyLongObject`, first call its `__index__()` method (if present) to convert it to a `PyLongObject`.

Raise `OverflowError` if the value of *obj* is out of range for a long.

Returns `-1` on error. Use `PyErr_Occurred()` to disambiguate.

Changed in version 3.8: Use `__index__()` if available.

Changed in version 3.10: This function will no longer use `__int__()`.

long PyLong_AS_LONG(PyObject *obj)

A soft deprecated alias. Exactly equivalent to the preferred `PyLong_AsLong`. In particular, it can fail with **`OverflowError`** or another exception.

Deprecated since version 3.14: The function is soft deprecated.

int PyLong_AsInt(PyObject *obj)

Part of the Stable ABI since version 3.13.
Similar to `PyLong_AsLong()`, but store the result in a C int instead of a C long.

Added in version 3.13.

long PyLong_AsLongAndOverflow(PyObject *obj, int *overflow)

Part of the Stable ABI.
Return a C long representation of *obj*. If *obj* is not an instance of **`PyLongObject`**, first call its `__index__()` method (if present) to convert it to a **`PyLongObject`**.

If the value of *obj* is greater than **`LONG_MAX`** or less than **`LONG_MIN`**, set **overflow* to `1` or `-1`, respectively, and return `-1`; otherwise, set **overflow* to `0`. If any other exception occurs set **overflow* to `0` and return `-1` as usual.

Returns `-1` on error. Use `PyErr_Occurred()` to disambiguate.

Changed in version 3.8: Use `__index__()` if available.

Changed in version 3.10: This function will no longer use `__int__()`.

long long PyLong_AsLongLong(PyObject *obj)

Part of the Stable ABI.

Return a C long long representation of *obj*. If *obj* is not an instance of `PyLongObject`, first call its `__index__()` method (if present) to convert it to a `PyLongObject`.

Raise `OverflowError` if the value of *obj* is out of range for a long long.

Returns `-1` on error. Use `PyErr_Occurred()` to disambiguate.

Changed in version 3.8: Use `__index__()` if available.

Changed in version 3.10: This function will no longer use `__int__()`.

long long PyLong_AsLongLongAndOverflow(PyObject *obj, int *overflow)

Part of the Stable ABI.

Return a C long long representation of *obj*. If *obj* is not an instance of `PyLongObject`, first call its `__index__()` method (if present) to convert it to a `PyLongObject`.

If the value of *obj* is greater than **`LLONG_MAX`** or less than **`LLONG_MIN`**, set **overflow* to `1` or `-1`, respectively, and return `-1`; otherwise, set **overflow* to `0`. If any other exception occurs set **overflow* to `0` and return `-1` as usual.

Returns `-1` on error. Use `PyErr_Occurred()` to disambiguate.

Added in version 3.2.

Changed in version 3.8: Use `__index__()` if available.

Changed in version 3.10: This function will no longer use `__int__()`.

Py_ssize_t PyLong_AsSsize_t(PyObject *pylong)

Part of the Stable ABI.

Return a C `Py_ssize_t` representation of *pylong*. *pylong* must be an instance of `PyLongObject`.

Raise `OverflowError` if the value of *pylong* is out of range for a `Py_ssize_t`.

Returns `-1` on error. Use `PyErr_Occurred()` to disambiguate.

unsigned long PyLong_AsUnsignedLong(PyObject *pylong)

Part of the Stable ABI.

Return a C unsigned long representation of *pylong*. *pylong* must be an instance of `PyLongObject`.

Raise `OverflowError` if the value of *pylong* is out of range for a unsigned long.

Returns `(unsigned long)-1` on error. Use `PyErr_Occurred()` to disambiguate.

size_t PyLong_AsSize_t(PyObject *pylong)

Part of the Stable ABI.

Return a C **`size_t`** representation of *pylong*. *pylong* must be an instance of `PyLongObject`.

Raise `OverflowError` if the value of *pylong* is out of range for a **`size_t`**.

Returns `(size_t)-1` on error. Use `PyErr_Occurred()` to disambiguate.

unsigned long long PyLong_AsUnsignedLongLong(PyObject *pylong)

Part of the Stable ABI.

Return a C unsigned long long representation of *pylong*. *pylong* must be an instance of **`PyLongObject`**.

Raise **`OverflowError`** if the value of *pylong* is out of range for an unsigned long long.

Returns `(unsigned long long)-1` on error. Use **`PyErr_Occurred()`** to disambiguate.

Changed in version 3.1: A negative *pylong* now raises **`OverflowError`**, not **`TypeError`**.

unsigned long PyLong_AsUnsignedLongMask(PyObject *obj)

Part of the Stable ABI.

Return a C unsigned long representation of *obj*. If *obj* is not an instance of **`PyLongObject`**, first call its **`__index__()`** method (if present) to convert it to a **`PyLongObject`**.

If the value of *obj* is out of range for an unsigned long, return the reduction of that value modulo `ULONG_MAX + 1`.

Returns `(unsigned long)-1` on error. Use **`PyErr_Occurred()`** to disambiguate.

Changed in version 3.8: Use **`__index__()`** if available.

Changed in version 3.10: This function will no longer use **`__int__()`**.

unsigned long long PyLong_AsUnsignedLongLongMask(PyObject *obj)

Part of the Stable ABI.

Return a C unsigned long long representation of *obj*. If *obj* is not an instance of **`PyLongObject`**, first call its **`__index__()`** method (if present) to convert it to a **`PyLongObject`**.

If the value of *obj* is out of range for an unsigned long long, return the reduction of that value modulo `ULLONG_MAX + 1`.

Returns `(unsigned long long)-1` on error. Use **`PyErr_Occurred()`** to disambiguate.

Changed in version 3.8: Use **`__index__()`** if available.

Changed in version 3.10: This function will no longer use **`__int__()`**.

int PyLong_AsInt32(PyObject *obj, int32_t *value)
int PyLong_AsInt64(PyObject *obj, int64_t *value)

Part of the Stable ABI since version 3.14.
Set **value* to a signed C int32_t or int64_t representation of *obj*.

If *obj* is not an instance of **`PyLongObject`**, first call its **`__index__()`** method (if present) to convert it to a **`PyLongObject`**.

If the *obj* value is out of range, raise an **`OverflowError`**.

Set **value* and return `0` on success. Set an exception and return `-1` on error.

value must not be `NULL`.

Added in version 3.14.

int PyLong_AsUInt32(PyObject *obj, uint32_t *value)
int PyLong_AsUInt64(PyObject *obj, uint64_t *value)

Part of the Stable ABI since version 3.14.
Set **value* to an unsigned C uint32_t or uint64_t representation of *obj*.

If *obj* is not an instance of **`PyLongObject`**, first call its **`__index__()`** method (if present) to convert it to a **`PyLongObject`**.

- If *obj* is negative, raise a **`ValueError`**.
- If the *obj* value is out of range, raise an **`OverflowError`**.

Set **value* and return `0` on success. Set an exception and return `-1` on error.

value must not be `NULL`.

Added in version 3.14.

double PyLong_AsDouble(PyObject *pylong)

Part of the Stable ABI.
Return a C double representation of *pylong*. *pylong* must be an instance of **`PyLongObject`**.

Raise **`OverflowError`** if the value of *pylong* is out of range for a double.

Returns `-1.0` on error. Use **`PyErr_Occurred()`** to disambiguate.

void *PyLong_AsVoidPtr(PyObject *pylong)

Part of the Stable ABI.
Convert a Python integer *pylong* to a C void pointer. If *pylong* cannot be converted, an **`OverflowError`** will be raised. This is only assured to produce a usable void pointer for values created with **`PyLong_FromVoidPtr()`**.

Returns `NULL` on error. Use **`PyErr_Occurred()`** to disambiguate.

Py_ssize_t PyLong_AsNativeBytes(PyObject *pylong, void *buffer, Py_ssize_t n_bytes, int flags)

Part of the Stable ABI since version 3.14.
Copy the Python integer value *pylong* to a native *buffer* of size *n_bytes*. The *flags* can be set to `-1` to behave similarly to a C cast, or to values documented below to control the behavior.

Returns `-1` with an exception raised on error. This may happen if *pylong* cannot be interpreted as an integer, or if *pylong* was negative and the `Py_ASNATIVEBYTES_REJECT_NEGATIVE` flag was set.

Otherwise, returns the number of bytes required to store the value. If this is equal to or less than *n_bytes*, the entire value was copied. All *n_bytes* of the buffer are written: remaining bytes filled by copies of the sign bit.

If the returned value is greater than *n_bytes*, the value was truncated: as many of the lowest bits of the value as could fit are written, and the higher bits are ignored. This matches the typical behavior of a C-style downcast.

Note

Overflow is not considered an error. If the returned value is larger than *n_bytes*, most significant bits were discarded.

`0` will never be returned.

Values are always copied as two’s-complement.

Usage example:

```
int32_t value;
Py_ssize_t bytes = PyLong_AsNativeBytes(pylong, &value,
sizeof(value), -1);
if (bytes < 0) {
    // Failed. A Python exception was set with the reason.
    return NULL;
}
else if (bytes <= (Py_ssize_t)sizeof(value)) {
    // Success!
}
else {
    // Overflow occurred, but 'value' contains the truncated
    // lowest bits of pylong.
}
```

Passing zero to *n_bytes* will return the size of a buffer that would be large enough to hold the value. This may be larger than technically necessary, but not unreasonably so. If *n_bytes=0*, *buffer* may be `NULL`.

Note

Passing *n_bytes=0* to this function is not an accurate way to determine the bit length of the value.

To get at the entire Python value of an unknown size, the function can be called twice: first to determine the buffer size, then to fill it:

```
// Ask how much space we need.
Py_ssize_t expected = PyLong_AsNativeBytes(pylong, NULL, 0,
-1);
if (expected < 0) {
    // Failed. A Python exception was set with the reason.
    return NULL;
}
assert(expected != 0);  // Impossible per the API
definition.
uint8_t *bignum = malloc(expected);
if (!bignum) {
    PyErr_SetString(PyExc_MemoryError, "bignum malloc
failed.");
    return NULL;
}
// Safely get the entire value.
```

```
Py_ssize_t bytes = PyLong_AsNativeBytes(pylong, bignum,
expected, -1);
if (bytes < 0) {  // Exception has been set.
    free(bignum);
    return NULL;
}
else if (bytes > expected) {  // This should not be
possible.
    PyErr_SetString(PyExc_RuntimeError,
        "Unexpected bignum truncation after a size check.");
    free(bignum);
    return NULL;
}
// The expected success given the above pre-check.
// ... use bignum ...
free(bignum);
```

flags is either `-1` (`Py_ASNATIVEBYTES_DEFAULTS`) to select defaults that behave most like a C cast, or a combination of the other flags in the table below. Note that `-1` cannot be combined with other flags.

Currently, `-1` corresponds to `Py_ASNATIVEBYTES_NATIVE_ENDIAN | Py_ASNATIVEBYTES_UNSIGNED_BUFFER`.

Flag	Value
Py_ASNATIVEBYTES_DEFAULTS *Part of the Stable ABI since version 3.14.*	`-1`
Py_ASNATIVEBYTES_BIG_ENDIAN *Part of the Stable ABI since version 3.14.*	`0`
Py_ASNATIVEBYTES_LITTLE_ENDIAN *Part of the Stable ABI since version 3.14.*	`1`
Py_ASNATIVEBYTES_NATIVE_ENDIAN *Part of the Stable ABI since version 3.14.*	`3`
Py_ASNATIVEBYTES_UNSIGNED_BUFFER	`4`

Flag	Value
Part of the Stable ABI since version 3.14.	
Py_ASNATIVEBYTES_REJECT_NEGATIVE *Part of the Stable ABI since version 3.14.*	8
Py_ASNATIVEBYTES_ALLOW_INDEX *Part of the Stable ABI since version 3.14.*	16

Specifying `Py_ASNATIVEBYTES_NATIVE_ENDIAN` will override any other endian flags. Passing `2` is reserved.

By default, sufficient buffer will be requested to include a sign bit. For example, when converting 128 with *n_bytes=1*, the function will return 2 (or more) in order to store a zero sign bit.

If `Py_ASNATIVEBYTES_UNSIGNED_BUFFER` is specified, a zero sign bit will be omitted from size calculations. This allows, for example, 128 to fit in a single-byte buffer. If the destination buffer is later treated as signed, a positive input value may become negative. Note that the flag does not affect handling of negative values: for those, space for a sign bit is always requested.

Specifying `Py_ASNATIVEBYTES_REJECT_NEGATIVE` causes an exception to be set if *pylong* is negative. Without this flag, negative values will be copied provided there is enough space for at least one sign bit, regardless of whether `Py_ASNATIVEBYTES_UNSIGNED_BUFFER` was specified.

If `Py_ASNATIVEBYTES_ALLOW_INDEX` is specified and a non-integer value is passed, its `__index__()` method will be called first. This may result in Python code executing and other threads being allowed to run, which could cause changes to other objects or values in use. When *flags* is `-1`, this option is not set, and non-integer values will raise `TypeError`.

Note

With the default *flags* (`-1`, or *UNSIGNED_BUFFER* without *REJECT_NEGATIVE*), multiple Python integers can map to a single value without overflow. For example, both `255` and `-1` fit a single-byte buffer and set all its bits. This matches typical C cast behavior.

Added in version 3.13.

PyLong_AsPid(pid)

Macro for converting a Python integer into a process identifier.

This can be defined as an alias to `PyLong_AsLong()`, `PyLong_FromLongLong()`, or `PyLong_AsInt()`, depending on the size of the system's PID type.

Added in version 3.2.

int PyLong_GetSign(PyObject *obj, int *sign)

Get the sign of the integer object *obj*.

On success, set **sign* to the integer sign (0, -1 or +1 for zero, negative or positive integer, respectively) and return 0.

On failure, return -1 with an exception set. This function always succeeds if *obj* is a `PyLongObject` or its subtype.

Added in version 3.14.

int PyLong_IsPositive(PyObject *obj)

Check if the integer object *obj* is positive (`obj > 0`).

If *obj* is an instance of **`PyLongObject`** or its subtype, return `1` when it's positive and `0` otherwise. Else set an exception and return `-1`.

Added in version 3.14.

int PyLong_IsNegative(PyObject *obj)

Check if the integer object *obj* is negative (`obj < 0`).

If *obj* is an instance of **`PyLongObject`** or its subtype, return `1` when it's negative and `0` otherwise. Else set an exception and return `-1`.

Added in version 3.14.

int PyLong_IsZero(PyObject *obj)

Check if the integer object *obj* is zero.

If *obj* is an instance of **`PyLongObject`** or its subtype, return `1` when it's zero and `0` otherwise. Else set an exception and return `-1`.

Added in version 3.14.

PyObject *PyLong_GetInfo(void)

Part of the Stable ABI.

On success, return a read only named tuple, that holds information about Python's internal representation of integers. See **`sys.int_info`** for description of individual fields.

On failure, return `NULL` with an exception set.

Added in version 3.1.

int PyUnstable_Long_IsCompact(const PyLongObject *op)

This is Unstable API. It may change without warning in minor releases.

Return 1 if *op* is compact, 0 otherwise.

This function makes it possible for performance-critical code to implement a “fast path” for small integers. For compact values use `PyUnstable_Long_CompactValue()`; for others fall back to a `PyLong_As*` function or `PyLong_AsNativeBytes()`.

The speedup is expected to be negligible for most users.

Exactly what values are considered compact is an implementation detail and is subject to change.

Added in version 3.12.

Py_ssize_t PyUnstable_Long_CompactValue(const PyLongObject *op)

This is Unstable API. It may change without warning in minor releases.

If *op* is compact, as determined by `PyUnstable_Long_IsCompact()`, return its value.

Otherwise, the return value is undefined.

Added in version 3.12.

Export API

Added in version 3.14.

struct PyLongLayout

Layout of an array of “digits” (“limbs” in the GMP terminology), used to represent absolute value for arbitrary precision integers.

Use `PyLong_GetNativeLayout()` to get the native layout of Python `int` objects, used internally for integers with “big enough” absolute value.

See also `sys.int_info` which exposes similar information in Python.

uint8_t bits_per_digit

Bits per digit. For example, a 15 bit digit means that bits 0-14 contain meaningful information.

uint8_t digit_size

Digit size in bytes. For example, a 15 bit digit will require at least 2 bytes.

int8_t digits_order

Digits order:

- `1` for most significant digit first
- `-1` for least significant digit first

int8_t digit_endianness

Digit endianness:

- `1` for most significant byte first (big endian)
- `-1` for least significant byte first (little endian)

const PyLongLayout *PyLong_GetNativeLayout(void)

Get the native layout of Python `int` objects.

See the `PyLongLayout` structure.

The function must not be called before Python initialization nor after Python finalization. The returned layout is valid until Python is finalized. The layout is the same for all Python sub-interpreters in a process, and so it can be cached.

struct PyLongExport

Export of a Python `int` object.

There are two cases:

- If `digits` is `NULL`, only use the `value` member.
- If `digits` is not `NULL`, use `negative`, `ndigits` and `digits` members.

int64_t value

The native integer value of the exported `int` object. Only valid if `digits` is `NULL`.

uint8_t negative

`1` if the number is negative, `0` otherwise. Only valid if `digits` is not `NULL`.

Py_ssize_t ndigits

Number of digits in `digits` array. Only valid if `digits` is not `NULL`.

const void *digits

Read-only array of unsigned digits. Can be `NULL`.

int PyLong_Export(PyObject *obj, PyLongExport *export_long)

Export a Python `int` object.

export_long must point to a **`PyLongExport`** structure allocated by the caller. It must not be `NULL`.

On success, fill in **export_long* and return `0`. On error, set an exception and return `-1`.

`PyLong_FreeExport()` must be called when the export is no longer needed.

> **CPython implementation detail:** This function always succeeds if *obj* is a Python **`int`** object or a subclass.

void PyLong_FreeExport(PyLongExport *export_long)

Release the export *export_long* created by **`PyLong_Export()`**.

CPython implementation detail: Calling **`PyLong_FreeExport()`** is optional if *export_long->digits* is `NULL`.

PyLongWriter API

The **`PyLongWriter`** API can be used to import an integer.

Added in version 3.14.

struct PyLongWriter

A Python **`int`** writer instance.

The instance must be destroyed by **`PyLongWriter_Finish()`** or **`PyLongWriter_Discard()`**.

PyLongWriter *PyLongWriter_Create(int negative, Py_ssize_t ndigits, void **digits)

Create a **`PyLongWriter`**.

On success, allocate **digits* and return a writer. On error, set an exception and return `NULL`.

negative is `1` if the number is negative, or `0` otherwise.

ndigits is the number of digits in the *digits* array. It must be greater than 0.

digits must not be NULL.

After a successful call to this function, the caller should fill in the array of digits *digits* and then call **`PyLongWriter_Finish()`** to get a Python **`int`**. The layout of *digits* is described by **`PyLong_GetNativeLayout()`**.

Digits must be in the range [`0`; `(1 << bits_per_digit) - 1`] (where the **`bits_per_digit`** is the number of bits per digit). Any unused most significant digits must be set to `0`.

Alternately, call **`PyLongWriter_Discard()`** to destroy the writer instance without creating an **`int`** object.

PyObject *PyLongWriter_Finish(PyLongWriter *writer)

Return value: New reference.
Finish a **`PyLongWriter`** created by **`PyLongWriter_Create()`**.

On success, return a Python **`int`** object. On error, set an exception and return `NULL`.

The function takes care of normalizing the digits and converts the object to a compact integer if needed.

The writer instance and the *digits* array are invalid after the call.

void PyLongWriter_Discard(PyLongWriter *writer)

Discard a **`PyLongWriter`** created by **`PyLongWriter_Create()`**.

If *writer* is `NULL`, no operation is performed.

The writer instance and the *digits* array are invalid after the call.

Deprecated API

These macros are soft deprecated. They describe parameters of the internal representation of **`PyLongObject`** instances.

Use **`PyLong_GetNativeLayout()`** instead, along with **`PyLong_Export()`** to read integer data or **`PyLongWriter`** to write it. These currently use the same layout, but are designed to continue working correctly even if CPython's internal integer representation changes.

PyLong_SHIFT

This is equivalent to **`bits_per_digit`** in the output of **`PyLong_GetNativeLayout()`**.

PyLong_BASE

This is currently equivalent to 1 << PyLong_SHIFT.

PyLong_MASK

This is currently equivalent to (1 << PyLong_SHIFT) - 1

Boolean Objects

Booleans in Python are implemented as a subclass of integers. There are only two booleans, **`Py_False`** and **`Py_True`**. As such, the normal creation and deletion functions don't apply to booleans. The following macros are available, however.

PyTypeObject PyBool_Type

Part of the Stable ABI.

This instance of **`PyTypeObject`** represents the Python boolean type; it is the same object as **`bool`** in the Python layer.

int PyBool_Check(PyObject *o)

Return true if *o* is of type **`PyBool_Type`**. This function always succeeds.

PyObject *Py_False

The Python `False` object. This object has no methods and is immortal.

Changed in version 3.12: **`Py_False`** is immortal.

PyObject *Py_True

The Python `True` object. This object has no methods and is immortal.

Changed in version 3.12: **`Py_True`** is immortal.

Py_RETURN_FALSE

Return **`Py_False`** from a function.

Py_RETURN_TRUE

Return `Py_True` from a function.

PyObject *PyBool_FromLong(long v)

Return value: New reference. Part of the Stable ABI.

Return `Py_True` or `Py_False`, depending on the truth value of *v*.

Floating-Point Objects

type PyFloatObject

This subtype of **`PyObject`** represents a Python floating-point object.

PyTypeObject PyFloat_Type

Part of the Stable ABI.

This instance of **`PyTypeObject`** represents the Python floating-point type. This is the same object as **`float`** in the Python layer.

int PyFloat_Check(PyObject *p)

Return true if its argument is a **`PyFloatObject`** or a subtype of **`PyFloatObject`**. This function always succeeds.

int PyFloat_CheckExact(PyObject *p)

Return true if its argument is a **`PyFloatObject`**, but not a subtype of **`PyFloatObject`**. This function always succeeds.

PyObject *PyFloat_FromString(PyObject *str)

Return value: New reference. Part of the Stable ABI.

Create a **`PyFloatObject`** object based on the string value in *str*, or `NULL` on failure.

PyObject *PyFloat_FromDouble(double v)

Return value: New reference. Part of the Stable ABI.

Create a **`PyFloatObject`** object from *v*, or `NULL` on failure.

double PyFloat_AsDouble(PyObject *pyfloat)

Part of the Stable ABI.

Return a C double representation of the contents of *pyfloat*. If *pyfloat* is not a Python floating-point object but has a `__float__()` method, this method will first be called to convert *pyfloat* into a float. If `__float__()` is not defined then it falls back to `__index__()`. This method returns `-1.0` upon failure, so one should call `PyErr_Occurred()` to check for errors.

Changed in version 3.8: Use `__index__()` if available.

double PyFloat_AS_DOUBLE(PyObject *pyfloat)

Return a C double representation of the contents of *pyfloat*, but without error checking.

PyObject *PyFloat_GetInfo(void)

Return value: New reference. Part of the Stable ABI.
Return a structseq instance which contains information about the precision, minimum and maximum values of a float. It's a thin wrapper around the header file `float.h`.

double PyFloat_GetMax()

Part of the Stable ABI.
Return the maximum representable finite float *DBL_MAX* as C double.

double PyFloat_GetMin()

Part of the Stable ABI.
Return the minimum normalized positive float *DBL_MIN* as C double.

Py_INFINITY

This macro expands a to constant expression of type double, that represents the positive infinity.

On most platforms, this is equivalent to the **`INFINITY`** macro from the C11 standard `<math.h>` header.

Py_NAN

This macro expands a to constant expression of type double, that represents a quiet not-a-number (qNaN) value.

On most platforms, this is equivalent to the **`NAN`** macro from the C11 standard `<math.h>` header.

Py_HUGE_VAL

Equivalent to **`INFINITY`**.

Deprecated since version 3.14: The macro is soft deprecated.

Py_MATH_E

The definition (accurate for a double type) of the **`math.e`** constant.

Py_MATH_El

High precision (long double) definition of **`e`** constant.

Py_MATH_PI

The definition (accurate for a double type) of the **`math.pi`** constant.

Py_MATH_PIl

High precision (long double) definition of **`pi`** constant.

Py_MATH_TAU

The definition (accurate for a double type) of the **`math.tau`** constant.

Added in version 3.6.

Py_RETURN_NAN

Return **`math.nan`** from a function.

On most platforms, this is equivalent to `return PyFloat_FromDouble(NAN)`.

Py_RETURN_INF(sign)

Return **`math.inf`** or **`-math.inf`** from a function, depending on the sign of *sign*.

On most platforms, this is equivalent to the following:

```
return PyFloat_FromDouble(copysign(INFINITY, sign));
```

Py_IS_FINITE(X)

Return `1` if the given floating-point number *X* is finite, that is, it is normal, subnormal or zero, but not infinite or NaN. Return `0` otherwise.

Deprecated since version 3.14: The macro is soft deprecated. Use **`isfinite`** instead.

Py_IS_INFINITY(X)

Return `1` if the given floating-point number *X* is positive or negative infinity. Return `0` otherwise.

Deprecated since version 3.14: The macro is soft deprecated. Use **`isinf`** instead.

Py_IS_NAN(X)

Return `1` if the given floating-point number *X* is a not-a-number (NaN) value. Return `0` otherwise.

Deprecated since version 3.14: The macro is soft deprecated. Use **`isnan`** instead.

Pack and Unpack functions

The pack and unpack functions provide an efficient platform-independent way to store floating-point values as byte strings. The Pack routines produce a bytes string from a C double, and the Unpack routines produce a C double from such a bytes string. The suffix (2, 4 or 8) specifies the number of bytes in the bytes string.

On platforms that appear to use IEEE 754 formats these functions work by copying bits. On other platforms, the 2-byte format is identical to the IEEE 754 binary16 half-precision format, the 4-byte format (32-bit) is identical to the IEEE 754 binary32 single precision format, and the 8-byte format to the IEEE 754 binary64 double precision format, although the packing of INFs and NaNs (if such things exist on the platform) isn't handled correctly, and attempting to unpack a bytes string containing an IEEE INF or NaN will raise an exception.

Note that NaNs type may not be preserved on IEEE platforms (signaling NaN become quiet NaN), for example on x86 systems in 32-bit mode.

On non-IEEE platforms with more precision, or larger dynamic range, than IEEE 754 supports, not all values can be packed; on non-IEEE platforms with less precision, or smaller dynamic range, not all values can be unpacked. What happens in such cases is partly accidental (alas).

Added in version 3.11.

Pack functions

The pack routines write 2, 4 or 8 bytes, starting at *p*. *le* is an int argument, non-zero if you want the bytes string in little-endian format (exponent last, at `p+1`, `p+3`, or `p+6` `p+7`), zero if you want big-endian format (exponent first, at *p*). The

`PY_BIG_ENDIAN` constant can be used to use the native endian: it is equal to `1` on big endian processor, or `0` on little endian processor.

Return value: `0` if all is OK, `-1` if error (and an exception is set, most likely `OverflowError`).

There are two problems on non-IEEE platforms:

- What this does is undefined if *x* is a NaN or infinity.
- `-0.0` and `+0.0` produce the same bytes string.

int PyFloat_Pack2(double x, char *p, int le)

Pack a C double as the IEEE 754 binary16 half-precision format.

int PyFloat_Pack4(double x, char *p, int le)

Pack a C double as the IEEE 754 binary32 single precision format.

int PyFloat_Pack8(double x, char *p, int le)

Pack a C double as the IEEE 754 binary64 double precision format.

Unpack functions

The unpack routines read 2, 4 or 8 bytes, starting at *p*. *le* is an int argument, non-zero if the bytes string is in little-endian format (exponent last, at `p+1`, `p+3` or `p+6` and `p+7`), zero if big-endian (exponent first, at *p*). The **`PY_BIG_ENDIAN`** constant can be used to use the native endian: it is equal to `1` on big endian processor, or `0` on little endian processor.

Return value: The unpacked double. On error, this is `-1.0` and `PyErr_Occurred()` is true (and an exception is set, most likely `OverflowError`).

Note that on a non-IEEE platform this will refuse to unpack a bytes string that represents a NaN or infinity.

double PyFloat_Unpack2(const char *p, int le)

Unpack the IEEE 754 binary16 half-precision format as a C double.

double PyFloat_Unpack4(const char *p, int le)

Unpack the IEEE 754 binary32 single precision format as a C double.

double PyFloat_Unpack8(const char *p, int le)

Unpack the IEEE 754 binary64 double precision format as a C double.

Complex Number Objects

Python’s complex number objects are implemented as two distinct types when viewed from the C API: one is the Python object exposed to Python programs, and the other is a C structure which represents the actual complex number value. The API provides functions for working with both.

Complex Numbers as C Structures

Note that the functions which accept these structures as parameters and return them as results do so *by value* rather than dereferencing them through pointers. This is consistent throughout the API.

type Py_complex

The C structure which corresponds to the value portion of a Python complex number object. Most of the functions for dealing with complex number objects use structures of this type as input or output values, as appropriate.

double real
double imag

The structure is defined as:

```
typedef struct {
    double real;
    double imag;
} Py_complex;
```

Py_complex _Py_c_sum(Py_complex left, Py_complex right)

Return the sum of two complex numbers, using the C **`Py_complex`** representation.

Py_complex _Py_c_diff(Py_complex left, Py_complex right)

Return the difference between two complex numbers, using the C `Py_complex` representation.

Py_complex _Py_c_neg(Py_complex num)

Return the negation of the complex number *num*, using the C `Py_complex` representation.

Py_complex _Py_c_prod(Py_complex left, Py_complex right)

Return the product of two complex numbers, using the C `Py_complex` representation.

Py_complex _Py_c_quot(Py_complex dividend, Py_complex divisor)

Return the quotient of two complex numbers, using the C `Py_complex` representation.

If *divisor* is null, this method returns zero and sets `errno` to `EDOM`.

Py_complex _Py_c_pow(Py_complex num, Py_complex exp)

Return the exponentiation of *num* by *exp*, using the C `Py_complex` representation.

If *num* is null and *exp* is not a positive real number, this method returns zero and sets `errno` to `EDOM`.

Set `errno` to `ERANGE` on overflows.

Complex Numbers as Python Objects

type PyComplexObject

This subtype of `PyObject` represents a Python complex number object.

PyTypeObject PyComplex_Type

Part of the Stable ABI.

This instance of `PyTypeObject` represents the Python complex number type. It is the same object as `complex` in the Python layer.

int PyComplex_Check(PyObject *p)

Return true if its argument is a `PyComplexObject` or a subtype of `PyComplexObject`. This function always succeeds.

int PyComplex_CheckExact(PyObject *p)

Return true if its argument is a `PyComplexObject`, but not a subtype of `PyComplexObject`. This function always succeeds.

PyObject *PyComplex_FromCComplex(Py_complex v)

Return value: New reference.

Create a new Python complex number object from a C `Py_complex` value. Return `NULL` with an exception set on error.

PyObject *PyComplex_FromDoubles(double real, double imag)

Return value: New reference. Part of the Stable ABI.

Return a new `PyComplexObject` object from *real* and *imag*. Return `NULL` with an exception set on error.

double PyComplex_RealAsDouble(PyObject *op)

Part of the Stable ABI.

Return the real part of *op* as a C double.

If *op* is not a Python complex number object but has a `__complex__()` method, this method will first be called to convert *op* to a Python complex number object. If `__complex__()` is not defined then it falls back to call `PyFloat_AsDouble()` and returns its result.

Upon failure, this method returns `-1.0` with an exception set, so one should call **`PyErr_Occurred()`** to check for errors.

Changed in version 3.13: Use **`__complex__()`** if available.

double PyComplex_ImagAsDouble(PyObject *op)

Part of the Stable ABI.

Return the imaginary part of *op* as a C double.

If *op* is not a Python complex number object but has a **`__complex__()`** method, this method will first be called to convert *op* to a Python complex number object. If **`__complex__()`** is not defined then it falls back to call **`PyFloat_AsDouble()`** and returns `0.0` on success.

Upon failure, this method returns `-1.0` with an exception set, so one should call **`PyErr_Occurred()`** to check for errors.

Changed in version 3.13: Use **`__complex__()`** if available.

Py_complex PyComplex_AsCComplex(PyObject *op)

Return the **`Py_complex`** value of the complex number *op*.

If *op* is not a Python complex number object but has a **`__complex__()`** method, this method will first be called to convert *op* to a Python complex number object. If **`__complex__()`** is not defined then it falls back to **`__float__()`**. If **`__float__()`** is not defined then it falls back to **`__index__()`**.

Upon failure, this method returns **`Py_complex`** with **`real`** set to `-1.0` and with an exception set, so one should call **`PyErr_Occurred()`** to check for errors.

Changed in version 3.8: Use **`__index__()`** if available.

Bytes Objects

These functions raise `TypeError` when expecting a bytes parameter and called with a non-bytes parameter.

type PyBytesObject

This subtype of `PyObject` represents a Python bytes object.

PyTypeObject PyBytes_Type

Part of the Stable ABI.

This instance of `PyTypeObject` represents the Python bytes type; it is the same object as `bytes` in the Python layer.

int PyBytes_Check(PyObject *o)

Return true if the object *o* is a bytes object or an instance of a subtype of the bytes type. This function always succeeds.

int PyBytes_CheckExact(PyObject *o)

Return true if the object *o* is a bytes object, but not an instance of a subtype of the bytes type. This function always succeeds.

PyObject *PyBytes_FromString(const char *v)

Return value: New reference. Part of the Stable ABI.

Return a new bytes object with a copy of the string *v* as value on success, and `NULL` on failure. The parameter *v* must not be `NULL`; it will not be checked.

PyObject *PyBytes_FromStringAndSize(const char *v, Py_ssize_t len)

Return value: New reference. Part of the Stable ABI.

Return a new bytes object with a copy of the string *v* as value and length *len* on success, and `NULL` on failure. If *v* is `NULL`, the contents of the bytes object are uninitialized.

PyObject *PyBytes_FromFormat(const char *format, ...)

Return value: New reference. Part of the Stable ABI.
Take a C **`printf()`**-style *format* string and a variable number of arguments, calculate the size of the resulting Python bytes object and return a bytes object with the values formatted into it. The variable arguments must be C types and must correspond exactly to the format characters in the *format* string. The following format characters are allowed:

Format Characters	**Type**	**Comment**
`%%`	*n/a*	The literal % character.
`%c`	int	A single byte, represented as a C int.
`%d`	int	Equivalent to `printf("%d")`. [[1]]
`%u`	unsigned int	Equivalent to `printf("%u")`. [[1]]
`%ld`	long	Equivalent to `printf("%ld")`. [[1]]
`%lu`	unsigned long	Equivalent to `printf("%lu")`. [[1]]
`%zd`	**`Py_ssize_t`**	Equivalent to `printf("%zd")`. [[1]]

Format Characters	Type	Comment
`%zu`	size_t	Equivalent to `printf("%zu")`. [[1]]
`%i`	int	Equivalent to `printf("%i")`. [[1]]
`%x`	int	Equivalent to `printf("%x")`. [[1]]
`%s`	const char*	A null-terminated C character array.
`%p`	const void*	The hex representation of a C pointer. Mostly equivalent to `printf("%p")` except that it is guaranteed to start with the literal `0x` regardless of what the platform's `printf` yields.

An unrecognized format character causes all the rest of the format string to be copied as-is to the result object, and any extra arguments discarded.

[1] (1,2,3,4,5,6,7,8)
For integer specifiers (d, u, ld, lu, zd, zu, i, x): the 0-conversion flag has effect even when a precision is given.

PyObject *PyBytes_FromFormatV(const char *format, va_list vargs)

Return value: New reference. Part of the Stable ABI.
Identical to **`PyBytes_FromFormat()`** except that it takes exactly two arguments.

PyObject *PyBytes_FromObject(PyObject *o)

Return value: New reference. Part of the Stable ABI.
Return the bytes representation of object *o* that implements the buffer protocol.

Py_ssize_t PyBytes_Size(PyObject *o)

Part of the Stable ABI.
Return the length of the bytes in bytes object *o*.

Py_ssize_t PyBytes_GET_SIZE(PyObject *o)

Similar to **`PyBytes_Size()`**, but without error checking.

char *PyBytes_AsString(PyObject *o)

Part of the Stable ABI.
Return a pointer to the contents of *o*. The pointer refers to the internal buffer of *o*, which consists of `len(o) + 1` bytes. The last byte in the buffer is always null, regardless of whether there are any other null bytes. The data must not be modified in any way, unless the object was just created using `PyBytes_FromStringAndSize(NULL, size)`. It must not be deallocated. If *o* is not a bytes object at all, **`PyBytes_AsString()`** returns `NULL` and raises **`TypeError`**.

char *PyBytes_AS_STRING(PyObject *string)

Similar to **`PyBytes_AsString()`**, but without error checking.

int PyBytes_AsStringAndSize(PyObject *obj, char **buffer, Py_ssize_t *length)

Part of the Stable ABI.
Return the null-terminated contents of the object *obj* through the output variables *buffer* and *length*. Returns `0` on success.

If *length* is `NULL`, the bytes object may not contain embedded null bytes; if it does, the function returns `-1` and a `ValueError` is raised.

The buffer refers to an internal buffer of *obj*, which includes an additional null byte at the end (not counted in *length*). The data must not be modified in any way, unless the object was just created using `PyBytes_FromStringAndSize(NULL, size)`. It must not be deallocated. If *obj* is not a bytes object at all, `PyBytes_AsStringAndSize()` returns `-1` and raises `TypeError`.

Changed in version 3.5: Previously, `TypeError` was raised when embedded null bytes were encountered in the bytes object.

void PyBytes_Concat(PyObject **bytes, PyObject *newpart)

Part of the Stable ABI.

Create a new bytes object in **bytes* containing the contents of *newpart* appended to *bytes*; the caller will own the new reference. The reference to the old value of *bytes* will be stolen. If the new object cannot be created, the old reference to *bytes* will still be discarded and the value of **bytes* will be set to `NULL`; the appropriate exception will be set.

void PyBytes_ConcatAndDel(PyObject **bytes, PyObject *newpart)

Part of the Stable ABI.

Create a new bytes object in **bytes* containing the contents of *newpart* appended to *bytes*. This version releases the strong reference to *newpart* (i.e. decrements its reference count).

PyObject *PyBytes_Join(PyObject *sep, PyObject *iterable)

Similar to `sep.join(iterable)` in Python.

sep must be Python **`bytes`** object. (Note that **`PyUnicode_Join()`** accepts `NULL` separator and treats it as a space, whereas **`PyBytes_Join()`** doesn't accept `NULL` separator.)

iterable must be an iterable object yielding objects that implement the buffer protocol.

On success, return a new **`bytes`** object. On error, set an exception and return `NULL`.

Added in version 3.14.

int _PyBytes_Resize(PyObject **bytes, Py_ssize_t newsize)

Resize a bytes object. *newsize* will be the new length of the bytes object. You can think of it as creating a new bytes object and destroying the old one, only more efficiently. Pass the address of an existing bytes object as an lvalue (it may be written into), and the new size desired. On success, **bytes* holds the resized bytes object and `0` is returned; the address in **bytes* may differ from its input value. If the reallocation fails, the original bytes object at **bytes* is deallocated, **bytes* is set to `NULL`, **`MemoryError`** is set, and `-1` is returned.

PyObject *PyBytes_Repr(PyObject *bytes, int smartquotes)

Part of the Stable ABI.

Get the string representation of *bytes*. This function is currently used to implement **`bytes.__repr__()`** in Python.

This function does not do type checking; it is undefined behavior to pass *bytes* as a non-bytes object or `NULL`.

If *smartquotes* is true, the representation will use a double-quoted string instead of single-quoted string when single-quotes are present in *bytes*. For example, the byte string `'Python'` would be represented as `b"'Python'"` when *smartquotes* is true, or `b'\'Python\''` when it is false.

On success, this function returns a strong reference to a **`str`** object containing the representation. On failure, this returns `NULL` with an exception set.

PyObject *PyBytes_DecodeEscape(const char *s, Py_ssize_t len, const char *errors, Py_ssize_t unicode, const char *recode_encoding)

Part of the Stable ABI.
Unescape a backslash-escaped string *s*. *s* must not be `NULL`. *len* must be the size of *s*.

errors must be one of `"strict"`, `"replace"`, or `"ignore"`. If *errors* is `NULL`, then `"strict"` is used by default.

On success, this function returns a strong reference to a Python **`bytes`** object containing the unescaped string. On failure, this function returns `NULL` with an exception set.

Changed in version 3.9: unicode and *recode_encoding* are now unused.

Byte Array Objects

type PyByteArrayObject

This subtype of **`PyObject`** represents a Python bytearray object.

PyTypeObject PyByteArray_Type

Part of the Stable ABI.

This instance of **`PyTypeObject`** represents the Python bytearray type; it is the same object as **`bytearray`** in the Python layer.

Type check macros

int PyByteArray_Check(PyObject *o)

Return true if the object *o* is a bytearray object or an instance of a subtype of the bytearray type. This function always succeeds.

int PyByteArray_CheckExact(PyObject *o)

Return true if the object *o* is a bytearray object, but not an instance of a subtype of the bytearray type. This function always succeeds.

Direct API functions

PyObject *PyByteArray_FromObject(PyObject *o)

Return value: New reference. Part of the Stable ABI.

Return a new bytearray object from any object, *o*, that implements the buffer protocol.

On failure, return `NULL` with an exception set.

PyObject *PyByteArray_FromStringAndSize(const char *string, Py_ssize_t len)

Return value: New reference. Part of the Stable ABI.
Create a new bytearray object from *string* and its length, *len*.

On failure, return `NULL` with an exception set.

PyObject *PyByteArray_Concat(PyObject *a, PyObject *b)

Return value: New reference. Part of the Stable ABI.
Concat bytearrays *a* and *b* and return a new bytearray with the result.

On failure, return `NULL` with an exception set.

Py_ssize_t PyByteArray_Size(PyObject *bytearray)

Part of the Stable ABI.
Return the size of *bytearray* after checking for a `NULL` pointer.

char *PyByteArray_AsString(PyObject *bytearray)

Part of the Stable ABI.
Return the contents of *bytearray* as a char array after checking for a `NULL` pointer. The returned array always has an extra null byte appended.

int PyByteArray_Resize(PyObject *bytearray, Py_ssize_t len)

Part of the Stable ABI.
Resize the internal buffer of *bytearray* to *len*. Failure is a `-1` return with an exception set.

Changed in version 3.14: A negative *len* will now result in an exception being set and -1 returned.

Macros

These macros trade safety for speed and they don’t check pointers.

char *PyByteArray_AS_STRING(PyObject *bytearray)

Similar to `PyByteArray_AsString()`, but without error checking.

Py_ssize_t PyByteArray_GET_SIZE(PyObject *bytearray)

Similar to `PyByteArray_Size()`, but without error checking.

Unicode Objects and Codecs

Unicode Objects

Since the implementation of **PEP 393** [https://peps.python.org/pep-0393/] in Python 3.3, Unicode objects internally use a variety of representations, in order to allow handling the complete range of Unicode characters while staying memory efficient. There are special cases for strings where all code points are below 128, 256, or 65536; otherwise, code points must be below 1114112 (which is the full Unicode range).

UTF-8 representation is created on demand and cached in the Unicode object.

Note

The `Py_UNICODE` representation has been removed since Python 3.12 with deprecated APIs. See **PEP 623** [https://peps.python.org/pep-0623/] for more information.

Unicode Type

These are the basic Unicode object types used for the Unicode implementation in Python:

PyTypeObject PyUnicode_Type

Part of the Stable ABI.
This instance of `PyTypeObject` represents the Python Unicode type. It is exposed to Python code as `str`.

PyTypeObject PyUnicodeIter_Type

Part of the Stable ABI.

This instance of `PyTypeObject` represents the Python Unicode iterator type. It is used to iterate over Unicode string objects.

type Py_UCS4
type Py_UCS2
type Py_UCS1

Part of the Stable ABI.

These types are typedefs for unsigned integer types wide enough to contain characters of 32 bits, 16 bits and 8 bits, respectively. When dealing with single Unicode characters, use `Py_UCS4`.

Added in version 3.3.

type PyASCIIObject
type PyCompactUnicodeObject
type PyUnicodeObject

These subtypes of `PyObject` represent a Python Unicode object. In almost all cases, they shouldn't be used directly, since all API functions that deal with Unicode objects take and return `PyObject` pointers.

Added in version 3.3.

The structure of a particular object can be determined using the following macros. The macros cannot fail; their behavior is undefined if their argument is not a Python Unicode object.

PyUnicode_IS_COMPACT(o)

True if *o* uses the `PyCompactUnicodeObject` structure.

Added in version 3.3.

PyUnicode_IS_COMPACT_ASCII(o)

True if *o* uses the **`PyASCIIObject`** structure.

Added in version 3.3.

The following APIs are C macros and static inlined functions for fast checks and access to internal read-only data of Unicode objects:

int PyUnicode_Check(PyObject *obj)

Return true if the object *obj* is a Unicode object or an instance of a Unicode subtype. This function always succeeds.

int PyUnicode_CheckExact(PyObject *obj)

Return true if the object *obj* is a Unicode object, but not an instance of a subtype. This function always succeeds.

Py_ssize_t PyUnicode_GET_LENGTH(PyObject *unicode)

Return the length of the Unicode string, in code points. *unicode* has to be a Unicode object in the “canonical” representation (not checked).

Added in version 3.3.

Py_UCS1 *PyUnicode_1BYTE_DATA(PyObject *unicode)
Py_UCS2 *PyUnicode_2BYTE_DATA(PyObject *unicode)
Py_UCS4 *PyUnicode_4BYTE_DATA(PyObject *unicode)

Return a pointer to the canonical representation cast to UCS1, UCS2 or UCS4 integer types for direct character access. No checks are performed if the canonical representation has the correct character size; use **`PyUnicode_KIND()`** to select the right function.

Added in version 3.3.

PyUnicode_1BYTE_KIND

PyUnicode_2BYTE_KIND
PyUnicode_4BYTE_KIND

Return values of the `PyUnicode_KIND()` macro.

Added in version 3.3.

Changed in version 3.12: `PyUnicode_WCHAR_KIND` has been removed.

int PyUnicode_KIND(PyObject *unicode)

Return one of the PyUnicode kind constants (see above) that indicate how many bytes per character this Unicode object uses to store its data. *unicode* has to be a Unicode object in the “canonical” representation (not checked).

Added in version 3.3.

void *PyUnicode_DATA(PyObject *unicode)

Return a void pointer to the raw Unicode buffer. *unicode* has to be a Unicode object in the “canonical” representation (not checked).

Added in version 3.3.

void PyUnicode_WRITE(int kind, void *data, Py_ssize_t index, Py_UCS4 value)

Write the code point *value* to the given zero-based *index* in a string.

The *kind* value and *data* pointer must have been obtained from a string using `PyUnicode_KIND()` and `PyUnicode_DATA()` respectively. You must hold a reference to that string while calling **`PyUnicode_WRITE()`**. All requirements of `PyUnicode_WriteChar()` also apply.

The function performs no checks for any of its requirements, and is intended for usage in loops.

Added in version 3.3.

Py_UCS4 PyUnicode_READ(int kind, void *data, Py_ssize_t index)

Read a code point from a canonical representation *data* (as obtained with `PyUnicode_DATA()`). No checks or ready calls are performed.

Added in version 3.3.

Py_UCS4 PyUnicode_READ_CHAR(PyObject *unicode, Py_ssize_t index)

Read a character from a Unicode object *unicode*, which must be in the “canonical” representation. This is less efficient than `PyUnicode_READ()` if you do multiple consecutive reads.

Added in version 3.3.

Py_UCS4 PyUnicode_MAX_CHAR_VALUE(PyObject *unicode)

Return the maximum code point that is suitable for creating another string based on *unicode*, which must be in the “canonical” representation. This is always an approximation but more efficient than iterating over the string.

Added in version 3.3.

int PyUnicode_IsIdentifier(PyObject *unicode)

Part of the Stable ABI.

Return `1` if the string is a valid identifier according to the language definition, section Names (identifiers and keywords). Return `0` otherwise.

Changed in version 3.9: The function does not call `Py_FatalError()` anymore if the string is not ready.

unsigned int PyUnicode_IS_ASCII(PyObject *unicode)

Return true if the string only contains ASCII characters. Equivalent to `str.isascii()`.

Added in version 3.2.

Unicode Character Properties

Unicode provides many different character properties. The most often needed ones are available through these macros which are mapped to C functions depending on the Python configuration.

int Py_UNICODE_ISSPACE(Py_UCS4 ch)

Return `1` or `0` depending on whether *ch* is a whitespace character.

int Py_UNICODE_ISLOWER(Py_UCS4 ch)

Return `1` or `0` depending on whether *ch* is a lowercase character.

int Py_UNICODE_ISUPPER(Py_UCS4 ch)

Return `1` or `0` depending on whether *ch* is an uppercase character.

int Py_UNICODE_ISTITLE(Py_UCS4 ch)

Return `1` or `0` depending on whether *ch* is a titlecase character.

int Py_UNICODE_ISLINEBREAK(Py_UCS4 ch)

Return `1` or `0` depending on whether *ch* is a linebreak character.

int Py_UNICODE_ISDECIMAL(Py_UCS4 ch)

Return `1` or `0` depending on whether *ch* is a decimal character.

int Py_UNICODE_ISDIGIT(Py_UCS4 ch)

Return `1` or `0` depending on whether *ch* is a digit character.

int Py_UNICODE_ISNUMERIC(Py_UCS4 ch)

Return `1` or `0` depending on whether *ch* is a numeric character.

int Py_UNICODE_ISALPHA(Py_UCS4 ch)

Return `1` or `0` depending on whether *ch* is an alphabetic character.

int Py_UNICODE_ISALNUM(Py_UCS4 ch)

Return `1` or `0` depending on whether *ch* is an alphanumeric character.

int Py_UNICODE_ISPRINTABLE(Py_UCS4 ch)

Return `1` or `0` depending on whether *ch* is a printable character, in the sense of **`str.isprintable()`**.

These APIs can be used for fast direct character conversions:

Py_UCS4 Py_UNICODE_TOLOWER(Py_UCS4 ch)

Return the character *ch* converted to lower case.

Py_UCS4 Py_UNICODE_TOUPPER(Py_UCS4 ch)

Return the character *ch* converted to upper case.

Py_UCS4 Py_UNICODE_TOTITLE(Py_UCS4 ch)

Return the character *ch* converted to title case.

int Py_UNICODE_TODECIMAL(Py_UCS4 ch)

Return the character *ch* converted to a decimal positive integer. Return `-1` if this is not possible. This function does not raise exceptions.

int Py_UNICODE_TODIGIT(Py_UCS4 ch)

Return the character *ch* converted to a single digit integer. Return `-1` if this is not possible. This function does not raise exceptions.

double Py_UNICODE_TONUMERIC(Py_UCS4 ch)

Return the character *ch* converted to a double. Return `-1.0` if this is not possible. This function does not raise exceptions.

These APIs can be used to work with surrogates:

int Py_UNICODE_IS_SURROGATE(Py_UCS4 ch)

Check if *ch* is a surrogate (`0xD800 <= ch <= 0xDFFF`).

int Py_UNICODE_IS_HIGH_SURROGATE(Py_UCS4 ch)

Check if *ch* is a high surrogate (`0xD800 <= ch <= 0xDBFF`).

int Py_UNICODE_IS_LOW_SURROGATE(Py_UCS4 ch)

Check if *ch* is a low surrogate (`0xDC00 <= ch <= 0xDFFF`).

Py_UCS4 Py_UNICODE_HIGH_SURROGATE(Py_UCS4 ch)

Return the high UTF-16 surrogate (`0xD800` to `0xDBFF`) for a Unicode code point in the range `[0x10000; 0x10FFFF]`.

Py_UCS4 Py_UNICODE_LOW_SURROGATE(Py_UCS4 ch)

Return the low UTF-16 surrogate (`0xDC00` to `0xDFFF`) for a Unicode code point in the range `[0x10000; 0x10FFFF]`.

Py_UCS4 Py_UNICODE_JOIN_SURROGATES(Py_UCS4 high, Py_UCS4 low)

Join two surrogate code points and return a single **`Py_UCS4`** value. *high* and *low* are respectively the leading and trailing surrogates in a surrogate pair. *high* must be in the range `[0xD800; 0xDBFF]` and *low* must be in the range `[0xDC00; 0xDFFF]`.

Creating and accessing Unicode strings

To create Unicode objects and access their basic sequence properties, use these APIs:

PyObject *PyUnicode_New(Py_ssize_t size, Py_UCS4 maxchar)

Return value: New reference.
Create a new Unicode object. *maxchar* should be the true maximum code point to be placed in the string. As an approximation, it can be rounded up to the nearest value in the sequence 127, 255, 65535, 1114111.

On error, set an exception and return `NULL`.

After creation, the string can be filled by **`PyUnicode_WriteChar()`**, **`PyUnicode_CopyCharacters()`**, **`PyUnicode_Fill()`**, **`PyUnicode_WRITE()`** or similar. Since strings are supposed to be immutable, take care to not “use” the result while it is being modified. In particular, before it’s filled with its final contents, a string:

- must not be hashed,
- must not be **`converted to UTF-8`**, or another non-“canonical” representation,
- must not have its reference count changed,
- must not be shared with code that might do one of the above.

This list is not exhaustive. Avoiding these uses is your responsibility; Python does not always check these requirements.

To avoid accidentally exposing a partially-written string object, prefer using the **`PyUnicodeWriter`** API, or one of the `PyUnicode_From*` functions below.

Added in version 3.3.

PyObject *PyUnicode_FromKindAndData(int kind, const void *buffer, Py_ssize_t size)

Return value: New reference.
Create a new Unicode object with the given *kind* (possible values are **`PyUnicode_1BYTE_KIND`** etc., as returned by **`PyUnicode_KIND()`**). The *buffer* must point to an array of *size* units of 1, 2 or 4 bytes per character, as given by the kind.

If necessary, the input *buffer* is copied and transformed into the canonical representation. For example, if the *buffer* is a UCS4 string (**`PyUnicode_4BYTE_KIND`**) and it consists only of codepoints in the UCS1 range, it will be transformed into UCS1 (**`PyUnicode_1BYTE_KIND`**).

Added in version 3.3.

PyObject *PyUnicode_FromStringAndSize(const char *str, Py_ssize_t size)

Return value: New reference. Part of the Stable ABI.
Create a Unicode object from the char buffer *str*. The bytes will be interpreted as being UTF-8 encoded. The buffer is copied into the new object. The return value might be a shared object, i.e. modification of the data is not allowed.

This function raises **`SystemError`** when:

- *size* < 0,
- *str* is `NULL` and *size* > 0

Changed in version 3.12: str == `NULL` with *size* > 0 is not allowed anymore.

PyObject *PyUnicode_FromString(const char *str)

Return value: New reference. Part of the Stable ABI.
Create a Unicode object from a UTF-8 encoded null-terminated char buffer *str*.

PyObject *PyUnicode_FromFormat(const char *format, ...)

Return value: New reference. Part of the Stable ABI.

Take a C **`printf()`**-style *format* string and a variable number of arguments, calculate the size of the resulting Python Unicode string and return a string with the values formatted into it. The variable arguments must be C types and must correspond exactly to the format characters in the *format* ASCII-encoded string.

A conversion specifier contains two or more characters and has the following components, which must occur in this order:

1. The `'%'` character, which marks the start of the specifier.
2. Conversion flags (optional), which affect the result of some conversion types.
3. Minimum field width (optional). If specified as an `'*'` (asterisk), the actual width is given in the next argument, which must be of type int, and the object to convert comes after the minimum field width and optional precision.
4. Precision (optional), given as a `'.'` (dot) followed by the precision. If specified as `'*'` (an asterisk), the actual precision is given in the next argument, which must be of type int, and the value to convert comes after the precision.
5. Length modifier (optional).
6. Conversion type.

The conversion flag characters are:

Flag	Meaning
`0`	The conversion will be zero padded for numeric values.

Flag	Meaning
`-`	The converted value is left adjusted (overrides the `0` flag if both are given).

The length modifiers for following integer conversions (`d`, `i`, `o`, `u`, `x`, or `X`) specify the type of the argument (int by default):

Modifier	Types
`l`	long or unsigned long
`ll`	long long or unsigned long long
`j`	**`intmax_t`** or **`uintmax_t`**
`z`	**`size_t`** or **`ssize_t`**
`t`	**`ptrdiff_t`**

The length modifier `l` for following conversions `s` or `V` specify that the type of the argument is const wchar_t*.

The conversion specifiers are:

Conversion Specifier	Type	Comment
`%`	*n/a*	The literal `%` character.
`d`, `i`	Specified by the length modifier	The decimal representation of a signed C integer.
`u`	Specified by the length	The decimal representation of an unsigned C integer.

Conversion Specifier	Type	Comment
	modifier	
`o`	Specified by the length modifier	The octal representation of an unsigned C integer.
`x`	Specified by the length modifier	The hexadecimal representation of an unsigned C integer (lowercase).
`X`	Specified by the length modifier	The hexadecimal representation of an unsigned C integer (uppercase).
`c`	int	A single character.
`s`	const char* or const wchar_t*	A null-terminated C character array.
`p`	const void*	The hex representation of a C pointer. Mostly equivalent to `printf("%p")` except that it is guaranteed to start with the literal `0x` regardless of what the platform's `printf` yields.
`A`	PyObject*	The result of calling **`ascii()`**.
`U`	PyObject*	A Unicode object.

Conversion Specifier	Type	Comment
`V`	PyObject*, const char* or const wchar_t*	A Unicode object (which may be `NULL`) and a null-terminated C character array as a second parameter (which will be used, if the first parameter is `NULL`).
`S`	PyObject*	The result of calling **`PyObject_Str()`**.
`R`	PyObject*	The result of calling **`PyObject_Repr()`**.
`T`	PyObject*	Get the fully qualified name of an object type; call **`PyType_GetFullyQualifiedName()`**.
`#T`	PyObject*	Similar to `T` format, but use a colon (`:`) as separator between the module name and the qualified name.
`N`	PyTypeObject*	Get the fully qualified name of a type; call **`PyType_GetFullyQualifiedName()`**.
`#N`	PyTypeObject*	Similar to `N` format, but use a colon (`:`) as separator between the module name and the qualified name.

Note

The width formatter unit is number of characters rather than bytes. The precision formatter unit is number of bytes or **`wchar_t`** items (if the length modifier `l` is used) for `"%s"` and `"%V"` (if the `PyObject*` argument is `NULL`), and a number of characters for `"%A"`, `"%U"`, `"%S"`, `"%R"` and `"%V"` (if the `PyObject*` argument is not `NULL`).

Note

Unlike to C **`printf()`** the `0` flag has effect even when a precision is given for integer conversions (`d`, `i`, `u`, `o`, `x`, or `X`).

Changed in version 3.2: Support for `"%lld"` and `"%llu"` added.

Changed in version 3.3: Support for `"%li"`, `"%lli"` and `"%zi"` added.

Changed in version 3.4: Support width and precision formatter for `"%s"`, `"%A"`, `"%U"`, `"%V"`, `"%S"`, `"%R"` added.

Changed in version 3.12: Support for conversion specifiers `o` and `X`. Support for length modifiers `j` and `t`. Length modifiers are now applied to all integer conversions. Length modifier `l` is now applied to conversion specifiers `s` and `V`. Support for variable width and precision `*`. Support for flag `-`.

An unrecognized format character now sets a `SystemError`. In previous versions it caused all the rest of the format string to be copied as-is to the result string, and any extra arguments discarded.

Changed in version 3.13: Support for `%T`, `%#T`, `%N` and `%#N` formats added.

PyObject *PyUnicode_FromFormatV(const char *format, va_list vargs)

Return value: New reference. Part of the Stable ABI.
Identical to **`PyUnicode_FromFormat()`** except that it takes exactly two arguments.

PyObject *PyUnicode_FromObject(PyObject *obj)

Return value: New reference. Part of the Stable ABI.
Copy an instance of a Unicode subtype to a new true Unicode object if necessary. If *obj* is already a true Unicode object (not a subtype), return a new strong reference to the object.

Objects other than Unicode or its subtypes will cause a **`TypeError`**.

PyObject *PyUnicode_FromOrdinal(int ordinal)

Return value: New reference. Part of the Stable ABI.
Create a Unicode Object from the given Unicode code point *ordinal*.

The ordinal must be in `range(0x110000)`. A **`ValueError`** is raised in the case it is not.

PyObject *PyUnicode_FromEncodedObject(PyObject *obj, const char *encoding, const char *errors)

Return value: New reference. Part of the Stable ABI.
Decode an encoded object *obj* to a Unicode object.

`bytes`, **`bytearray`** and other bytes-like objects are decoded according to the given *encoding* and using the error handling defined by *errors*. Both can be `NULL` to have the interface use the default values (see Built-in Codecs for details).

All other objects, including Unicode objects, cause a `TypeError` to be set.

The API returns `NULL` if there was an error. The caller is responsible for decref'ing the returned objects.

void PyUnicode_Append(PyObject **p_left, PyObject *right)

Part of the Stable ABI.

Append the string *right* to the end of *p_left*. *p_left* must point to a strong reference to a Unicode object; **`PyUnicode_Append()`** releases ("steals") this reference.

On error, set **p_left* to `NULL` and set an exception.

On success, set **p_left* to a new strong reference to the result.

void PyUnicode_AppendAndDel(PyObject **p_left, PyObject *right)

Part of the Stable ABI.

The function is similar to `PyUnicode_Append()`, with the only difference being that it decrements the reference count of *right* by one.

PyObject *PyUnicode_BuildEncodingMap(PyObject *string)

Return value: New reference. Part of the Stable ABI.

Return a mapping suitable for decoding a custom single-byte encoding. Given a Unicode string *string* of up to 256 characters representing an encoding table, returns either a compact internal mapping object or a dictionary mapping character ordinals to byte values. Raises a `TypeError` and return `NULL` on invalid input.

Added in version 3.2.

const char *PyUnicode_GetDefaultEncoding(void)

Part of the Stable ABI.

Return the name of the default string encoding, `"utf-8"`. See `sys.getdefaultencoding()`.

The returned string does not need to be freed, and is valid until interpreter shutdown.

Py_ssize_t PyUnicode_GetLength(PyObject *unicode)

Part of the Stable ABI since version 3.7.
Return the length of the Unicode object, in code points.

On error, set an exception and return `-1`.

Added in version 3.3.

Py_ssize_t PyUnicode_CopyCharacters(PyObject *to, Py_ssize_t to_start, PyObject *from, Py_ssize_t from_start, Py_ssize_t how_many)

Copy characters from one Unicode object into another. This function performs character conversion when necessary and falls back to **`memcpy()`** if possible. Returns `-1` and sets an exception on error, otherwise returns the number of copied characters.

The string must not have been “used” yet. See `PyUnicode_New()` for details.

Added in version 3.3.

int PyUnicode_Resize(PyObject **unicode, Py_ssize_t length);

Part of the Stable ABI.
Resize a Unicode object **unicode* to the new *length* in code points.

Try to resize the string in place (which is usually faster than allocating a new string and copying characters), or create a new string.

**unicode* is modified to point to the new (resized) object and `0` is returned on success. Otherwise, `-1` is returned and an exception is set, and **unicode* is left untouched.

The function doesn’t check string content, the result may not be a string in canonical representation.

Py_ssize_t PyUnicode_Fill(PyObject *unicode, Py_ssize_t start, Py_ssize_t length, Py_UCS4 fill_char)

Fill a string with a character: write *fill_char* into `unicode[start:start+length]`.

Fail if *fill_char* is bigger than the string maximum character, or if the string has more than 1 reference.

The string must not have been “used” yet. See **`PyUnicode_New()`** for details.

Return the number of written character, or return `-1` and raise an exception on error.

Added in version 3.3.

int PyUnicode_WriteChar(PyObject *unicode, Py_ssize_t index, Py_UCS4 character)

Part of the Stable ABI since version 3.7.
Write a *character* to the string *unicode* at the zero-based *index*. Return `0` on success, `-1` on error with an exception set.

This function checks that *unicode* is a Unicode object, that the index is not out of bounds, and that the object’s reference count is one. See **`PyUnicode_WRITE()`** for a version that skips these checks, making them your responsibility.

The string must not have been “used” yet. See **`PyUnicode_New()`** for details.

Added in version 3.3.

Py_UCS4 PyUnicode_ReadChar(PyObject *unicode, Py_ssize_t index)

Part of the Stable ABI since version 3.7.
Read a character from a string. This function checks that *unicode* is a Unicode object and the index is not out of bounds, in contrast to **`PyUnicode_READ_CHAR()`**, which performs no error checking.

Return character on success, `-1` on error with an exception set.

Added in version 3.3.

PyObject *PyUnicode_Substring(PyObject *unicode, Py_ssize_t start, Py_ssize_t end)

Return value: New reference. Part of the Stable ABI since version 3.7.
Return a substring of *unicode*, from character index *start* (included) to character index *end* (excluded). Negative indices are not supported. On error, set an exception and return `NULL`.

Added in version 3.3.

Py_UCS4 *PyUnicode_AsUCS4(PyObject *unicode, Py_UCS4 *buffer, Py_ssize_t buflen, int copy_null)

Part of the Stable ABI since version 3.7.
Copy the string *unicode* into a UCS4 buffer, including a null character, if *copy_null* is set. Returns `NULL` and sets an exception on error (in particular, a **`SystemError`** if *buflen* is smaller than the length of *unicode*). *buffer* is returned on success.

Added in version 3.3.

Py_UCS4 *PyUnicode_AsUCS4Copy(PyObject *unicode)

Part of the Stable ABI since version 3.7.
Copy the string *unicode* into a new UCS4 buffer that is allocated using **`PyMem_Malloc()`**. If this fails, `NULL` is returned with a **`MemoryError`** set. The returned buffer always has an extra null code point appended.

Added in version 3.3.

Locale Encoding

The current locale encoding can be used to decode text from the operating system.

PyObject *PyUnicode_DecodeLocaleAndSize(const char *str, Py_ssize_t length, const char *errors)

Return value: New reference. Part of the Stable ABI since version 3.7.
Decode a string from UTF-8 on Android and VxWorks, or from the current locale encoding on other platforms. The supported error handlers are `"strict"` and `"surrogateescape"` (**PEP 383** [https://peps.python.org/pep-0383/]). The decoder uses `"strict"` error handler if *errors* is `NULL`. *str* must end with a null character but cannot contain embedded null characters.

Use **`PyUnicode_DecodeFSDefaultAndSize()`** to decode a string from the filesystem encoding and error handler.

This function ignores the Python UTF-8 Mode.

See also

The **`Py_DecodeLocale()`** function.

Added in version 3.3.

Changed in version 3.7: The function now also uses the current locale encoding for the `surrogateescape` error handler, except on Android. Previously, **`Py_DecodeLocale()`** was used for the `surrogateescape`, and the current locale encoding was used for `strict`.

PyObject *PyUnicode_DecodeLocale(const char *str, const char *errors)

Return value: New reference. Part of the Stable ABI since version 3.7.
Similar to **`PyUnicode_DecodeLocaleAndSize()`**, but compute the string length using **`strlen()`**.

Added in version 3.3.

PyObject *PyUnicode_EncodeLocale(PyObject *unicode, const char *errors)

Return value: New reference. Part of the Stable ABI since version 3.7.
Encode a Unicode object to UTF-8 on Android and VxWorks, or to the current locale encoding on other platforms. The supported error handlers are `"strict"` and `"surrogateescape"` (**PEP 383** [https://peps.python.org/pep-0383/]). The encoder uses `"strict"` error handler if *errors* is `NULL`. Return a **`bytes`** object. *unicode* cannot contain embedded null characters.

Use **`PyUnicode_EncodeFSDefault()`** to encode a string to the filesystem encoding and error handler.

This function ignores the Python UTF-8 Mode.

See also

The **`Py_EncodeLocale()`** function.

Added in version 3.3.

Changed in version 3.7: The function now also uses the current locale encoding for the `surrogateescape` error handler, except on Android. Previously, **`Py_EncodeLocale()`** was used for the `surrogateescape`, and the current locale encoding was used for `strict`.

File System Encoding

Functions encoding to and decoding from the filesystem encoding and error handler (**PEP 383** [https://peps.python.org/pep-0383/] and **PEP 529** [https://peps.python.org/pep-0529/]).

To encode file names to **`bytes`** during argument parsing, the `"O&"` converter should be used, passing **`PyUnicode_FSConverter()`** as the conversion function:

int PyUnicode_FSConverter(PyObject *obj, void *result)

Part of the Stable ABI.

PyArg_Parse* converter: encode **`str`** objects – obtained directly or through the **`os.PathLike`** interface – to **`bytes`** using **`PyUnicode_EncodeFSDefault()`**; **`bytes`** objects are output as-is. *result* must be an address of a C variable of type PyObject* (or PyBytesObject*). On success, set the variable to a new strong reference to a bytes object which must be released when it is no longer used and return a non-zero value (**`Py_CLEANUP_SUPPORTED`**). Embedded null bytes are not allowed in the result. On failure, return `0` with an exception set.

If *obj* is `NULL`, the function releases a strong reference stored in the variable referred by *result* and returns `1`.

Added in version 3.1.

Changed in version 3.6: Accepts a path-like object.

To decode file names to **`str`** during argument parsing, the `"O&"` converter should be used, passing **`PyUnicode_FSDecoder()`** as the conversion function:

int PyUnicode_FSDecoder(PyObject *obj, void *result)

Part of the Stable ABI.

PyArg_Parse* converter: decode `bytes` objects – obtained either directly or indirectly through the `os.PathLike` interface – to `str` using `PyUnicode_DecodeFSDefaultAndSize()`; `str` objects are output as-is. *result* must be an address of a C variable of type PyObject* (or PyUnicodeObject*). On success, set the variable to a new strong reference to a Unicode object which must be released when it is no longer used and return a non-zero value (`Py_CLEANUP_SUPPORTED`). Embedded null characters are not allowed in the result. On failure, return `0` with an exception set.

If *obj* is `NULL`, release the strong reference to the object referred to by *result* and return `1`.

Added in version 3.2.

Changed in version 3.6: Accepts a path-like object.

PyObject *PyUnicode_DecodeFSDefaultAndSize(const char *str, Py_ssize_t size)

Return value: New reference. Part of the Stable ABI.

Decode a string from the filesystem encoding and error handler.

If you need to decode a string from the current locale encoding, use `PyUnicode_DecodeLocaleAndSize()`.

See also

The `Py_DecodeLocale()` function.

Changed in version 3.6: The filesystem error handler is now used.

PyObject *PyUnicode_DecodeFSDefault(const char *str)

Return value: New reference. Part of the Stable ABI.
Decode a null-terminated string from the filesystem encoding and error handler.

If the string length is known, use `PyUnicode_DecodeFSDefaultAndSize()`.

Changed in version 3.6: The filesystem error handler is now used.

PyObject *PyUnicode_EncodeFSDefault(PyObject *unicode)

Return value: New reference. Part of the Stable ABI.
Encode a Unicode object to the filesystem encoding and error handler, and return `bytes`. Note that the resulting `bytes` object can contain null bytes.

If you need to encode a string to the current locale encoding, use `PyUnicode_EncodeLocale()`.

See also

The `Py_EncodeLocale()` function.

Added in version 3.2.

Changed in version 3.6: The filesystem error handler is now used.

wchar_t Support

`wchar_t` support for platforms which support it:

PyObject *PyUnicode_FromWideChar(const wchar_t *wstr, Py_ssize_t size)

Return value: New reference. Part of the Stable ABI.

Create a Unicode object from the **`wchar_t`** buffer *wstr* of the given *size*. Passing `-1` as the *size* indicates that the function must itself compute the length, using **`wcslen()`**. Return `NULL` on failure.

Py_ssize_t PyUnicode_AsWideChar(PyObject *unicode, wchar_t *wstr, Py_ssize_t size)

Part of the Stable ABI.

Copy the Unicode object contents into the **`wchar_t`** buffer *wstr*. At most *size* **`wchar_t`** characters are copied (excluding a possibly trailing null termination character). Return the number of **`wchar_t`** characters copied or `-1` in case of an error.

When *wstr* is `NULL`, instead return the *size* that would be required to store all of *unicode* including a terminating null.

Note that the resulting wchar_t* string may or may not be null-terminated. It is the responsibility of the caller to make sure that the wchar_t* string is null-terminated in case this is required by the application. Also, note that the wchar_t* string might contain null characters, which would cause the string to be truncated when used with most C functions.

wchar_t *PyUnicode_AsWideCharString(PyObject *unicode, Py_ssize_t *size)

Part of the Stable ABI since version 3.7.

Convert the Unicode object to a wide character string. The output string always ends with a null character. If *size* is not `NULL`, write the number of wide characters (excluding the trailing null termination character) into **size*. Note that the resulting **`wchar_t`** string might contain null characters, which would cause the string to be truncated when used with most C functions. If *size* is `NULL` and the wchar_t* string contains null characters a **`ValueError`** is raised.

Returns a buffer allocated by `PyMem_New` (use `PyMem_Free()` to free it) on success. On error, returns `NULL` and **size* is undefined. Raises a `MemoryError` if memory allocation is failed.

Added in version 3.2.

Changed in version 3.7: Raises a `ValueError` if *size* is `NULL` and the wchar_t* string contains null characters.

Built-in Codecs

Python provides a set of built-in codecs which are written in C for speed. All of these codecs are directly usable via the following functions.

Many of the following APIs take two arguments encoding and errors, and they have the same semantics as the ones of the built-in `str()` string object constructor.

Setting encoding to `NULL` causes the default encoding to be used which is UTF-8. The file system calls should use `PyUnicode_FSConverter()` for encoding file names. This uses the filesystem encoding and error handler internally.

Error handling is set by errors which may also be set to `NULL` meaning to use the default handling defined for the codec. Default error handling for all built-in codecs is “strict” (`ValueError` is raised).

The codecs all use a similar interface. Only deviations from the following generic ones are documented for simplicity.

Generic Codecs

The following macro is provided:

Py_UNICODE_REPLACEMENT_CHARACTER

The Unicode code point `U+FFFD` (replacement character).

This Unicode character is used as the replacement character during decoding if the *errors* argument is set to “replace”.

These are the generic codec APIs:

PyObject *PyUnicode_Decode(const char *str, Py_ssize_t size, const char *encoding, const char *errors)

Return value: New reference. Part of the Stable ABI. Create a Unicode object by decoding *size* bytes of the encoded string *str*. *encoding* and *errors* have the same meaning as the parameters of the same name in the **`str()`** built-in function. The codec to be used is looked up using the Python codec registry. Return `NULL` if an exception was raised by the codec.

PyObject *PyUnicode_AsEncodedString(PyObject *unicode, const char *encoding, const char *errors)

Return value: New reference. Part of the Stable ABI. Encode a Unicode object and return the result as Python bytes object. *encoding* and *errors* have the same meaning as the parameters of the same name in the Unicode **`encode()`** method. The codec to be used is looked up using the Python codec registry. Return `NULL` if an exception was raised by the codec.

UTF-8 Codecs

These are the UTF-8 codec APIs:

PyObject *PyUnicode_DecodeUTF8(const char *str, Py_ssize_t size, const char *errors)

Return value: New reference. Part of the Stable ABI.

Create a Unicode object by decoding *size* bytes of the UTF-8 encoded string *str*. Return `NULL` if an exception was raised by the codec.

PyObject *PyUnicode_DecodeUTF8Stateful(const char *str, Py_ssize_t size, const char *errors, Py_ssize_t *consumed)

Return value: New reference. Part of the Stable ABI.
If *consumed* is `NULL`, behave like **`PyUnicode_DecodeUTF8()`**. If *consumed* is not `NULL`, trailing incomplete UTF-8 byte sequences will not be treated as an error. Those bytes will not be decoded and the number of bytes that have been decoded will be stored in *consumed*.

PyObject *PyUnicode_AsUTF8String(PyObject *unicode)

Return value: New reference. Part of the Stable ABI.
Encode a Unicode object using UTF-8 and return the result as Python bytes object. Error handling is “strict”. Return `NULL` if an exception was raised by the codec.

The function fails if the string contains surrogate code points (`U+D800` - `U+DFFF`).

const char *PyUnicode_AsUTF8AndSize(PyObject *unicode, Py_ssize_t *size)

Part of the Stable ABI since version 3.10.
Return a pointer to the UTF-8 encoding of the Unicode object, and store the size of the encoded representation (in bytes) in *size*. The *size* argument can be `NULL`; in this case no size will be stored. The returned buffer always has an extra null byte appended (not included in *size*), regardless of whether there are any other null code points.

On error, set an exception, set *size* to `-1` (if it’s not NULL) and return `NULL`.

The function fails if the string contains surrogate code points (`U+D800` - `U+DFFF`).

This caches the UTF-8 representation of the string in the Unicode object, and subsequent calls will return a pointer to the same buffer. The caller is not responsible for deallocating the buffer. The buffer is deallocated and pointers to it become invalid when the Unicode object is garbage collected.

Added in version 3.3.

Changed in version 3.7: The return type is now `const char *` rather of `char *`.

Changed in version 3.10: This function is a part of the limited API.

const char *PyUnicode_AsUTF8(PyObject *unicode)

As **`PyUnicode_AsUTF8AndSize()`**, but does not store the size.

Warning

This function does not have any special behavior for null characters [https://en.wikipedia.org/wiki/Null_character] embedded within *unicode*. As a result, strings containing null characters will remain in the returned string, which some C functions might interpret as the end of the string, leading to truncation. If truncation is an issue, it is recommended to use **`PyUnicode_AsUTF8AndSize()`** instead.

Added in version 3.3.

Changed in version 3.7: The return type is now `const char *` rather of `char *`.

UTF-32 Codecs

These are the UTF-32 codec APIs:

PyObject *PyUnicode_DecodeUTF32(const char *str, Py_ssize_t size, const char *errors, int *byteorder)

Return value: New reference. Part of the Stable ABI.

Decode *size* bytes from a UTF-32 encoded buffer string and return the corresponding Unicode object. *errors* (if non-`NULL`) defines the error handling. It defaults to “strict”.

If *byteorder* is non-`NULL`, the decoder starts decoding using the given byte order:

```
*byteorder == -1: little endian
*byteorder == 0:  native order
*byteorder == 1:  big endian
```

If `*byteorder` is zero, and the first four bytes of the input data are a byte order mark (BOM), the decoder switches to this byte order and the BOM is not copied into the resulting Unicode string. If `*byteorder` is `-1` or `1`, any byte order mark is copied to the output.

After completion, **byteorder* is set to the current byte order at the end of input data.

If *byteorder* is `NULL`, the codec starts in native order mode.

Return `NULL` if an exception was raised by the codec.

PyObject *PyUnicode_DecodeUTF32Stateful(const char *str, Py_ssize_t size, const char *errors, int *byteorder, Py_ssize_t *consumed)

Return value: New reference. Part of the Stable ABI.

If *consumed* is `NULL`, behave like **`PyUnicode_DecodeUTF32()`**. If *consumed* is not `NULL`, **`PyUnicode_DecodeUTF32Stateful()`** will not treat trailing incomplete UTF-32 byte sequences (such as a number of bytes not divisible by four) as an error. Those bytes will not be decoded and the number of bytes that have been decoded will be stored in *consumed*.

PyObject *PyUnicode_AsUTF32String(PyObject *unicode)

Return value: New reference. Part of the Stable ABI.
Return a Python byte string using the UTF-32 encoding in native byte order. The string always starts with a BOM mark. Error handling is “strict”. Return `NULL` if an exception was raised by the codec.

UTF-16 Codecs

These are the UTF-16 codec APIs:

PyObject *PyUnicode_DecodeUTF16(const char *str, Py_ssize_t size, const char *errors, int *byteorder)

Return value: New reference. Part of the Stable ABI.
Decode *size* bytes from a UTF-16 encoded buffer string and return the corresponding Unicode object. *errors* (if non-`NULL`) defines the error handling. It defaults to “strict”.

If *byteorder* is non-`NULL`, the decoder starts decoding using the given byte order:

```
*byteorder == -1: little endian
*byteorder == 0:  native order
*byteorder == 1:  big endian
```

If `*byteorder` is zero, and the first two bytes of the input data are a byte order mark (BOM), the decoder switches to this byte order and the BOM is not copied into the resulting Unicode string. If `*byteorder` is `-1` or `1`, any byte order mark is copied to the output (where it will result in either a `\ufeff` or a `\ufffe` character).

After completion, `*byteorder` is set to the current byte order at the end of input data.

If *byteorder* is `NULL`, the codec starts in native order mode.

Return `NULL` if an exception was raised by the codec.

PyObject *PyUnicode_DecodeUTF16Stateful(const char *str, Py_ssize_t size, const char *errors, int *byteorder, Py_ssize_t *consumed)

Return value: New reference. Part of the Stable ABI.
If *consumed* is `NULL`, behave like **`PyUnicode_DecodeUTF16()`**. If *consumed* is not `NULL`, **`PyUnicode_DecodeUTF16Stateful()`** will not treat trailing incomplete UTF-16 byte sequences (such as an odd number of bytes or a split surrogate pair) as an error. Those bytes will not be decoded and the number of bytes that have been decoded will be stored in *consumed*.

PyObject *PyUnicode_AsUTF16String(PyObject *unicode)

Return value: New reference. Part of the Stable ABI.
Return a Python byte string using the UTF-16 encoding in native byte order. The string always starts with a BOM mark. Error handling is “strict”. Return `NULL` if an exception was raised by the codec.

UTF-7 Codecs

These are the UTF-7 codec APIs:

PyObject *PyUnicode_DecodeUTF7(const char *str, Py_ssize_t size, const char *errors)

Return value: New reference. Part of the Stable ABI.
Create a Unicode object by decoding *size* bytes of the UTF-7 encoded string *str*. Return `NULL` if an exception was raised by the codec.

PyObject *PyUnicode_DecodeUTF7Stateful(const char *str, Py_ssize_t size, const char *errors, Py_ssize_t *consumed)

Return value: New reference. Part of the Stable ABI.
If *consumed* is `NULL`, behave like **`PyUnicode_DecodeUTF7()`**. If *consumed* is not `NULL`, trailing incomplete UTF-7 base-64 sections will not be treated as an error. Those bytes will not

be decoded and the number of bytes that have been decoded will be stored in *consumed*.

Unicode-Escape Codecs

These are the “Unicode Escape” codec APIs:

PyObject *PyUnicode_DecodeUnicodeEscape(const char *str, Py_ssize_t size, const char *errors)

Return value: New reference. Part of the Stable ABI.
Create a Unicode object by decoding *size* bytes of the Unicode-Escape encoded string *str*. Return `NULL` if an exception was raised by the codec.

PyObject *PyUnicode_AsUnicodeEscapeString(PyObject *unicode)

Return value: New reference. Part of the Stable ABI.
Encode a Unicode object using Unicode-Escape and return the result as a bytes object. Error handling is “strict”. Return `NULL` if an exception was raised by the codec.

Raw-Unicode-Escape Codecs

These are the “Raw Unicode Escape” codec APIs:

PyObject *PyUnicode_DecodeRawUnicodeEscape(const char *str, Py_ssize_t size, const char *errors)

Return value: New reference. Part of the Stable ABI.
Create a Unicode object by decoding *size* bytes of the Raw-Unicode-Escape encoded string *str*. Return `NULL` if an exception was raised by the codec.

PyObject *PyUnicode_AsRawUnicodeEscapeString(PyObject *unicode)

Return value: New reference. Part of the Stable ABI.

Encode a Unicode object using Raw-Unicode-Escape and return the result as a bytes object. Error handling is “strict”. Return `NULL` if an exception was raised by the codec.

Latin-1 Codecs

These are the Latin-1 codec APIs: Latin-1 corresponds to the first 256 Unicode ordinals and only these are accepted by the codecs during encoding.

PyObject *PyUnicode_DecodeLatin1(const char *str, Py_ssize_t size, const char *errors)

Return value: New reference. Part of the Stable ABI.
Create a Unicode object by decoding *size* bytes of the Latin-1 encoded string *str*. Return `NULL` if an exception was raised by the codec.

PyObject *PyUnicode_AsLatin1String(PyObject *unicode)

Return value: New reference. Part of the Stable ABI.
Encode a Unicode object using Latin-1 and return the result as Python bytes object. Error handling is “strict”. Return `NULL` if an exception was raised by the codec.

ASCII Codecs

These are the ASCII codec APIs. Only 7-bit ASCII data is accepted. All other codes generate errors.

PyObject *PyUnicode_DecodeASCII(const char *str, Py_ssize_t size, const char *errors)

Return value: New reference. Part of the Stable ABI.
Create a Unicode object by decoding *size* bytes of the ASCII encoded string *str*. Return `NULL` if an exception was raised by the codec.

PyObject *PyUnicode_AsASCIIString(PyObject *unicode)

Return value: New reference. Part of the Stable ABI.
Encode a Unicode object using ASCII and return the result as Python bytes object. Error handling is “strict”. Return `NULL` if an exception was raised by the codec.

Character Map Codecs

This codec is special in that it can be used to implement many different codecs (and this is in fact what was done to obtain most of the standard codecs included in the **`encodings`** package). The codec uses mappings to encode and decode characters. The mapping objects provided must support the **`__getitem__()`** mapping interface; dictionaries and sequences work well.

These are the mapping codec APIs:

PyObject *PyUnicode_DecodeCharmap(const char *str, Py_ssize_t length, PyObject *mapping, const char *errors)

Return value: New reference. Part of the Stable ABI.
Create a Unicode object by decoding *size* bytes of the encoded string *str* using the given *mapping* object. Return `NULL` if an exception was raised by the codec.

If *mapping* is `NULL`, Latin-1 decoding will be applied. Else *mapping* must map bytes ordinals (integers in the range from 0 to 255) to Unicode strings, integers (which are then interpreted as Unicode ordinals) or `None`. Unmapped data bytes – ones which cause a **`LookupError`**, as well as ones which get mapped to `None`, `0xFFFE` or `'\ufffe'`, are treated as undefined mappings and cause an error.

PyObject *PyUnicode_AsCharmapString(PyObject *unicode, PyObject *mapping)

Return value: New reference. Part of the Stable ABI.

Encode a Unicode object using the given *mapping* object and return the result as a bytes object. Error handling is "strict". Return `NULL` if an exception was raised by the codec.

The *mapping* object must map Unicode ordinal integers to bytes objects, integers in the range from 0 to 255 or `None`. Unmapped character ordinals (ones which cause a **`LookupError`**) as well as mapped to `None` are treated as "undefined mapping" and cause an error.

The following codec API is special in that maps Unicode to Unicode.

PyObject *PyUnicode_Translate(PyObject *unicode, PyObject *table, const char *errors)

Return value: New reference. Part of the Stable ABI.
Translate a string by applying a character mapping table to it and return the resulting Unicode object. Return `NULL` if an exception was raised by the codec.

The mapping table must map Unicode ordinal integers to Unicode ordinal integers or `None` (causing deletion of the character).

Mapping tables need only provide the **`__getitem__()`** interface; dictionaries and sequences work well. Unmapped character ordinals (ones which cause a **`LookupError`**) are left untouched and are copied as-is.

errors has the usual meaning for codecs. It may be `NULL` which indicates to use the default error handling.

MBCS codecs for Windows

These are the MBCS codec APIs. They are currently only available on Windows and use the Win32 MBCS converters to implement the conversions. Note that MBCS (or DBCS) is a class of encodings, not just one. The target encoding is defined by the user settings on the machine running the codec.

PyObject *PyUnicode_DecodeMBCS(const char *str, Py_ssize_t size, const char *errors)

Return value: New reference. Part of the Stable ABI on Windows since version 3.7.

Create a Unicode object by decoding *size* bytes of the MBCS encoded string *str*. Return `NULL` if an exception was raised by the codec.

PyObject *PyUnicode_DecodeMBCSStateful(const char *str, Py_ssize_t size, const char *errors, Py_ssize_t *consumed)

Return value: New reference. Part of the Stable ABI on Windows since version 3.7.

If *consumed* is `NULL`, behave like **`PyUnicode_DecodeMBCS()`**. If *consumed* is not `NULL`, **`PyUnicode_DecodeMBCSStateful()`** will not decode trailing lead byte and the number of bytes that have been decoded will be stored in *consumed*.

PyObject *PyUnicode_DecodeCodePageStateful(int code_page, const char *str, Py_ssize_t size, const char *errors, Py_ssize_t *consumed)

Return value: New reference. Part of the Stable ABI on Windows since version 3.7.

Similar to **`PyUnicode_DecodeMBCSStateful()`**, except uses the code page specified by *code_page*.

PyObject *PyUnicode_AsMBCSString(PyObject *unicode)

Return value: New reference. Part of the Stable ABI on Windows since version 3.7.

Encode a Unicode object using MBCS and return the result as Python bytes object. Error handling is “strict”. Return `NULL` if an exception was raised by the codec.

PyObject *PyUnicode_EncodeCodePage(int code_page, PyObject *unicode, const char *errors)

Return value: New reference. Part of the Stable ABI on Windows since version 3.7.
Encode the Unicode object using the specified code page and return a Python bytes object. Return `NULL` if an exception was raised by the codec. Use **`CP_ACP`** code page to get the MBCS encoder.

Added in version 3.3.

Methods and Slot Functions

The following APIs are capable of handling Unicode objects and strings on input (we refer to them as strings in the descriptions) and return Unicode objects or integers as appropriate.

They all return `NULL` or `-1` if an exception occurs.

PyObject *PyUnicode_Concat(PyObject *left, PyObject *right)

Return value: New reference. Part of the Stable ABI.
Concat two strings giving a new Unicode string.

PyObject *PyUnicode_Split(PyObject *unicode, PyObject *sep, Py_ssize_t maxsplit)

Return value: New reference. Part of the Stable ABI.
Split a string giving a list of Unicode strings. If *sep* is `NULL`, splitting will be done at all whitespace substrings. Otherwise, splits occur at the given separator. At most *maxsplit* splits will be done. If negative, no limit is set. Separators are not included in the resulting list.

On error, return `NULL` with an exception set.

Equivalent to **`str.split()`**.

PyObject *PyUnicode_RSplit(PyObject *unicode, PyObject *sep, Py_ssize_t maxsplit)

Return value: New reference. Part of the Stable ABI.
Similar to `PyUnicode_Split()`, but splitting will be done beginning at the end of the string.

On error, return `NULL` with an exception set.

Equivalent to `str.rsplit()`.

PyObject *PyUnicode_Splitlines(PyObject *unicode, int keepends)

Return value: New reference. Part of the Stable ABI.
Split a Unicode string at line breaks, returning a list of Unicode strings. CRLF is considered to be one line break. If *keepends* is `0`, the Line break characters are not included in the resulting strings.

PyObject *PyUnicode_Partition(PyObject *unicode, PyObject *sep)

Return value: New reference. Part of the Stable ABI.
Split a Unicode string at the first occurrence of *sep*, and return a 3-tuple containing the part before the separator, the separator itself, and the part after the separator. If the separator is not found, return a 3-tuple containing the string itself, followed by two empty strings.

sep must not be empty.

On error, return `NULL` with an exception set.

Equivalent to `str.partition()`.

PyObject *PyUnicode_RPartition(PyObject *unicode, PyObject *sep)

Return value: New reference. Part of the Stable ABI.

Similar to `PyUnicode_Partition()`, but split a Unicode string at the last occurrence of *sep*. If the separator is not found, return a 3-tuple containing two empty strings, followed by the string itself.

sep must not be empty.

On error, return `NULL` with an exception set.

Equivalent to `str.rpartition()`.

PyObject *PyUnicode_Join(PyObject *separator, PyObject *seq)

Return value: New reference. Part of the Stable ABI.
Join a sequence of strings using the given *separator* and return the resulting Unicode string.

Py_ssize_t PyUnicode_Tailmatch(PyObject *unicode, PyObject *substr, Py_ssize_t start, Py_ssize_t end, int direction)

Part of the Stable ABI.
Return `1` if *substr* matches `unicode[start:end]` at the given tail end (*direction* == `-1` means to do a prefix match, *direction* == `1` a suffix match), `0` otherwise. Return `-1` if an error occurred.

Py_ssize_t PyUnicode_Find(PyObject *unicode, PyObject *substr, Py_ssize_t start, Py_ssize_t end, int direction)

Part of the Stable ABI.
Return the first position of *substr* in `unicode[start:end]` using the given *direction* (*direction* == `1` means to do a forward search, *direction* == `-1` a backward search). The return value is the index of the first match; a value of `-1` indicates that no match was found, and `-2` indicates that an error occurred and an exception has been set.

Py_ssize_t PyUnicode_FindChar(PyObject *unicode, Py_UCS4 ch, Py_ssize_t start, Py_ssize_t end, int direction)

Part of the Stable ABI since version 3.7.
Return the first position of the character *ch* in `unicode[start:end]` using the given *direction* (*direction* == `1` means to do a forward search, *direction* == `-1` a backward search). The return value is the index of the first match; a value of `-1` indicates that no match was found, and `-2` indicates that an error occurred and an exception has been set.

Added in version 3.3.

Changed in version 3.7: start and *end* are now adjusted to behave like `unicode[start:end]`.

Py_ssize_t PyUnicode_Count(PyObject *unicode, PyObject *substr, Py_ssize_t start, Py_ssize_t end)

Part of the Stable ABI.
Return the number of non-overlapping occurrences of *substr* in `unicode[start:end]`. Return `-1` if an error occurred.

PyObject *PyUnicode_Replace(PyObject *unicode, PyObject *substr, PyObject *replstr, Py_ssize_t maxcount)

Return value: New reference. Part of the Stable ABI.
Replace at most *maxcount* occurrences of *substr* in *unicode* with *replstr* and return the resulting Unicode object. *maxcount* == `-1` means replace all occurrences.

int PyUnicode_Compare(PyObject *left, PyObject *right)

Part of the Stable ABI.
Compare two strings and return `-1`, `0`, `1` for less than, equal, and greater than, respectively.

This function returns `-1` upon failure, so one should call **`PyErr_Occurred()`** to check for errors.

See also

The `PyUnicode_Equal()` function.

int PyUnicode_Equal(PyObject *a, PyObject *b)

Part of the Stable ABI since version 3.14.

Test if two strings are equal:

- Return `1` if *a* is equal to *b*.
- Return `0` if *a* is not equal to *b*.
- Set a `TypeError` exception and return `-1` if *a* or *b* is not a `str` object.

The function always succeeds if *a* and *b* are `str` objects.

The function works for `str` subclasses, but does not honor custom `__eq__()` method.

See also

The `PyUnicode_Compare()` function.

Added in version 3.14.

int PyUnicode_EqualToUTF8AndSize(PyObject *unicode, const char *string, Py_ssize_t size)

Part of the Stable ABI since version 3.13.

Compare a Unicode object with a char buffer which is interpreted as being UTF-8 or ASCII encoded and return true (`1`) if they are equal, or false (`0`) otherwise. If the Unicode object contains surrogate code points (`U+D800` - `U+DFFF`) or the C string is not valid UTF-8, false (`0`) is returned.

This function does not raise exceptions.

Added in version 3.13.

int PyUnicode_EqualToUTF8(PyObject *unicode, const char *string)

Part of the Stable ABI since version 3.13.
Similar to **`PyUnicode_EqualToUTF8AndSize()`**, but compute *string* length using **`strlen()`**. If the Unicode object contains null characters, false (`0`) is returned.

Added in version 3.13.

int PyUnicode_CompareWithASCIIString(PyObject *unicode, const char *string)

Part of the Stable ABI.
Compare a Unicode object, *unicode*, with *string* and return `-1`, `0`, `1` for less than, equal, and greater than, respectively. It is best to pass only ASCII-encoded strings, but the function interprets the input string as ISO-8859-1 if it contains non-ASCII characters.

This function does not raise exceptions.

PyObject *PyUnicode_RichCompare(PyObject *left, PyObject *right, int op)

Return value: New reference. Part of the Stable ABI.
Rich compare two Unicode strings and return one of the following:

- `NULL` in case an exception was raised
- **`Py_True`** or **`Py_False`** for successful comparisons
- **`Py_NotImplemented`** in case the type combination is unknown

Possible values for *op* are **`Py_GT`**, **`Py_GE`**, **`Py_EQ`**, **`Py_NE`**, **`Py_LT`**, and **`Py_LE`**.

PyObject *PyUnicode_Format(PyObject *format, PyObject *args)

Return value: New reference. Part of the Stable ABI.

Return a new string object from *format* and *args*; this is analogous to `format % args`.

int PyUnicode_Contains(PyObject *unicode, PyObject *substr)

Part of the Stable ABI.

Check whether *substr* is contained in *unicode* and return true or false accordingly.

substr has to coerce to a one element Unicode string. `-1` is returned if there was an error.

void PyUnicode_InternInPlace(PyObject **p_unicode)

Part of the Stable ABI.

Intern the argument *p_unicode in place. The argument must be the address of a pointer variable pointing to a Python Unicode string object. If there is an existing interned string that is the same as *p_unicode, it sets *p_unicode to it (releasing the reference to the old string object and creating a new strong reference to the interned string object), otherwise it leaves *p_unicode alone and interns it.

(Clarification: even though there is a lot of talk about references, think of this function as reference-neutral. You must own the object you pass in; after the call you no longer own the passed-in reference, but you newly own the result.)

This function never raises an exception. On error, it leaves its argument unchanged without interning it.

Instances of subclasses of **`str`** may not be interned, that is, PyUnicode_CheckExact(*p_unicode) must be true. If it is not, then – as with any other error – the argument is left unchanged.

Note that interned strings are not “immortal”. You must keep a reference to the result to benefit from interning.

PyObject *PyUnicode_InternFromString(const char *str)

Return value: New reference. Part of the Stable ABI.

A combination of `PyUnicode_FromString()` and `PyUnicode_InternInPlace()`, meant for statically allocated strings.

Return a new (“owned”) reference to either a new Unicode string object that has been interned, or an earlier interned string object with the same value.

Python may keep a reference to the result, or make it immortal, preventing it from being garbage-collected promptly. For interning an unbounded number of different strings, such as ones coming from user input, prefer calling `PyUnicode_FromString()` and `PyUnicode_InternInPlace()` directly.

unsigned int PyUnicode_CHECK_INTERNED(PyObject *str)

Return a non-zero value if *str* is interned, zero if not. The *str* argument must be a string; this is not checked. This function always succeeds.

CPython implementation detail: A non-zero return value may carry additional information about *how* the string is interned. The meaning of such non-zero values, as well as each specific string’s intern-related details, may change between CPython versions.

PyUnicodeWriter

The `PyUnicodeWriter` API can be used to create a Python `str` object.

Added in version 3.14.

type PyUnicodeWriter

A Unicode writer instance.

The instance must be destroyed by `PyUnicodeWriter_Finish()` on success, or `PyUnicodeWriter_Discard()` on error.

PyUnicodeWriter *PyUnicodeWriter_Create(Py_ssize_t length)

Create a Unicode writer instance.

length must be greater than or equal to `0`.

If *length* is greater than `0`, preallocate an internal buffer of *length* characters.

Set an exception and return `NULL` on error.

PyObject *PyUnicodeWriter_Finish(PyUnicodeWriter *writer)

Return the final Python `str` object and destroy the writer instance.

Set an exception and return `NULL` on error.

The writer instance is invalid after this call.

void PyUnicodeWriter_Discard(PyUnicodeWriter *writer)

Discard the internal Unicode buffer and destroy the writer instance.

If *writer* is `NULL`, no operation is performed.

The writer instance is invalid after this call.

int PyUnicodeWriter_WriteChar(PyUnicodeWriter *writer, Py_UCS4 ch)

Write the single Unicode character *ch* into *writer*.

On success, return `0`. On error, set an exception, leave the writer unchanged, and return `-1`.

int PyUnicodeWriter_WriteUTF8(PyUnicodeWriter *writer, const char *str, Py_ssize_t size)

Decode the string *str* from UTF-8 in strict mode and write the output into *writer*.

size is the string length in bytes. If *size* is equal to `-1`, call `strlen(str)` to get the string length.

On success, return `0`. On error, set an exception, leave the writer unchanged, and return `-1`.

See also **`PyUnicodeWriter_DecodeUTF8Stateful()`**.

int PyUnicodeWriter_WriteASCII(PyUnicodeWriter *writer, const char *str, Py_ssize_t size)

Write the ASCII string *str* into *writer*.

size is the string length in bytes. If *size* is equal to `-1`, call `strlen(str)` to get the string length.

str must only contain ASCII characters. The behavior is undefined if *str* contains non-ASCII characters.

On success, return `0`. On error, set an exception, leave the writer unchanged, and return `-1`.

Added in version 3.14.

int PyUnicodeWriter_WriteWideChar(PyUnicodeWriter *writer, const wchar_t *str, Py_ssize_t size)

Write the wide string *str* into *writer*.

size is a number of wide characters. If *size* is equal to `-1`, call `wcslen(str)` to get the string length.

On success, return `0`. On error, set an exception, leave the writer unchanged, and return `-1`.

int PyUnicodeWriter_WriteUCS4(PyUnicodeWriter *writer, Py_UCS4 *str, Py_ssize_t size)

Writer the UCS4 string *str* into *writer*.

size is a number of UCS4 characters.

On success, return `0`. On error, set an exception, leave the writer unchanged, and return `-1`.

int PyUnicodeWriter_WriteStr(PyUnicodeWriter *writer, PyObject *obj)

Call **`PyObject_Str()`** on *obj* and write the output into *writer*.

On success, return `0`. On error, set an exception, leave the writer unchanged, and return `-1`.

int PyUnicodeWriter_WriteRepr(PyUnicodeWriter *writer, PyObject *obj)

Call **`PyObject_Repr()`** on *obj* and write the output into *writer*.

On success, return `0`. On error, set an exception, leave the writer unchanged, and return `-1`.

int PyUnicodeWriter_WriteSubstring(PyUnicodeWriter *writer, PyObject *str, Py_ssize_t start, Py_ssize_t end)

Write the substring `str[start:end]` into *writer*.

str must be Python **`str`** object. *start* must be greater than or equal to 0, and less than or equal to *end*. *end* must be less than or equal to *str* length.

On success, return `0`. On error, set an exception, leave the writer unchanged, and return `-1`.

int PyUnicodeWriter_Format(PyUnicodeWriter *writer, const char *format, ...)

Similar to `PyUnicode_FromFormat()`, but write the output directly into *writer*.

On success, return `0`. On error, set an exception, leave the writer unchanged, and return `-1`.

int PyUnicodeWriter_DecodeUTF8Stateful(PyUnicodeWriter *writer, const char *string, Py_ssize_t length, const char *errors, Py_ssize_t *consumed)

Decode the string *str* from UTF-8 with *errors* error handler and write the output into *writer*.

size is the string length in bytes. If *size* is equal to `-1`, call `strlen(str)` to get the string length.

errors is an error handler name, such as `"replace"`. If *errors* is `NULL`, use the strict error handler.

If *consumed* is not `NULL`, set **consumed* to the number of decoded bytes on success. If *consumed* is `NULL`, treat trailing incomplete UTF-8 byte sequences as an error.

On success, return `0`. On error, set an exception, leave the writer unchanged, and return `-1`.

See also `PyUnicodeWriter_WriteUTF8()`.

Deprecated API

The following API is deprecated.

type Py_UNICODE

This is a typedef of **`wchar_t`**, which is a 16-bit type or 32-bit type depending on the platform. Please use **`wchar_t`** directly instead.

Changed in version 3.3: In previous versions, this was a 16-bit type or a 32-bit type depending on whether you selected a “narrow” or “wide” Unicode version of Python at build time.

Deprecated since version 3.13, will be removed in version 3.15.

int PyUnicode_READY(PyObject *unicode)

Do nothing and return `0`. This API is kept only for backward compatibility, but there are no plans to remove it.

Added in version 3.3.

Deprecated since version 3.10: This API does nothing since Python 3.12. Previously, this needed to be called for each string created using the old API (**`PyUnicode_FromUnicode()`** or similar).

unsigned int PyUnicode_IS_READY(PyObject *unicode)

Do nothing and return `1`. This API is kept only for backward compatibility, but there are no plans to remove it.

Added in version 3.3.

Deprecated since version 3.14: This API does nothing since Python 3.12. Previously, this could be called to check if **`PyUnicode_READY()`** is necessary.

Tuple Objects

type PyTupleObject

This subtype of **`PyObject`** represents a Python tuple object.

PyTypeObject PyTuple_Type

Part of the Stable ABI.

This instance of **`PyTypeObject`** represents the Python tuple type; it is the same object as **`tuple`** in the Python layer.

int PyTuple_Check(PyObject *p)

Return true if *p* is a tuple object or an instance of a subtype of the tuple type. This function always succeeds.

int PyTuple_CheckExact(PyObject *p)

Return true if *p* is a tuple object, but not an instance of a subtype of the tuple type. This function always succeeds.

PyObject *PyTuple_New(Py_ssize_t len)

Return value: New reference. Part of the Stable ABI.

Return a new tuple object of size *len*, or `NULL` with an exception set on failure.

PyObject *PyTuple_Pack(Py_ssize_t n, ...)

Return value: New reference. Part of the Stable ABI.

Return a new tuple object of size *n*, or `NULL` with an exception set on failure. The tuple values are initialized to the subsequent *n* C arguments pointing to Python objects. `PyTuple_Pack(2, a, b)` is equivalent to `Py_BuildValue("(OO)", a, b)`.

Py_ssize_t PyTuple_Size(PyObject *p)

Part of the Stable ABI.

Take a pointer to a tuple object, and return the size of that tuple. On error, return `-1` with an exception set.

Py_ssize_t PyTuple_GET_SIZE(PyObject *p)

Like **`PyTuple_Size()`**, but without error checking.

PyObject *PyTuple_GetItem(PyObject *p, Py_ssize_t pos)

Return value: Borrowed reference. Part of the Stable ABI.
Return the object at position *pos* in the tuple pointed to by *p*. If *pos* is negative or out of bounds, return `NULL` and set an **`IndexError`** exception.

The returned reference is borrowed from the tuple *p* (that is: it is only valid as long as you hold a reference to *p*). To get a strong reference, use **`Py_NewRef(PyTuple_GetItem(...))`** or **`PySequence_GetItem()`**.

PyObject *PyTuple_GET_ITEM(PyObject *p, Py_ssize_t pos)

Return value: Borrowed reference.
Like **`PyTuple_GetItem()`**, but does no checking of its arguments.

PyObject *PyTuple_GetSlice(PyObject *p, Py_ssize_t low, Py_ssize_t high)

Return value: New reference. Part of the Stable ABI.
Return the slice of the tuple pointed to by *p* between *low* and *high*, or `NULL` with an exception set on failure.

This is the equivalent of the Python expression `p[low:high]`. Indexing from the end of the tuple is not supported.

int PyTuple_SetItem(PyObject *p, Py_ssize_t pos, PyObject *o)

Part of the Stable ABI.

Insert a reference to object *o* at position *pos* of the tuple pointed to by *p*. Return `0` on success. If *pos* is out of bounds, return `-1` and set an **`IndexError`** exception.

Note

This function “steals” a reference to *o* and discards a reference to an item already in the tuple at the affected position.

void PyTuple_SET_ITEM(PyObject *p, Py_ssize_t pos, PyObject *o)

Like **`PyTuple_SetItem()`**, but does no error checking, and should *only* be used to fill in brand new tuples.

Bounds checking is performed as an assertion if Python is built in debug mode or **`with assertions`**.

Note

This function “steals” a reference to *o*, and, unlike **`PyTuple_SetItem()`**, does *not* discard a reference to any item that is being replaced; any reference in the tuple at position *pos* will be leaked.

Warning

This macro should *only* be used on tuples that are newly created. Using this macro on a tuple that is already in use (or in other words, has a refcount > 1) could lead to undefined behavior.

int _PyTuple_Resize(PyObject **p, Py_ssize_t newsize)

Can be used to resize a tuple. *newsize* will be the new length of the tuple. Because tuples are *supposed* to be immutable, this should only be used if there is only one reference to the object. Do *not* use this if the tuple may already be known to some other part of the code. The tuple will always grow or shrink at the end. Think of this as destroying the old tuple and creating a new one, only more efficiently. Returns `0` on success. Client code should never assume that the resulting value of `*p` will be the same as before calling this function. If the object referenced by `*p` is replaced, the original `*p` is destroyed. On failure, returns `-1` and sets `*p` to `NULL`, and raises **`MemoryError`** or **`SystemError`**.

Struct Sequence Objects

Struct sequence objects are the C equivalent of **`namedtuple()`** objects, i.e. a sequence whose items can also be accessed through attributes. To create a struct sequence, you first have to create a specific struct sequence type.

PyTypeObject *PyStructSequence_NewType(PyStructSequence_Desc *desc)

Return value: New reference. Part of the Stable ABI.
Create a new struct sequence type from the data in *desc*, described below. Instances of the resulting type can be created with **`PyStructSequence_New()`**.

Return `NULL` with an exception set on failure.

void PyStructSequence_InitType(PyTypeObject *type, PyStructSequence_Desc *desc)

Initializes a struct sequence type *type* from *desc* in place.

int PyStructSequence_InitType2(PyTypeObject *type, PyStructSequence_Desc *desc)

Like `PyStructSequence_InitType()`, but returns `0` on success and `-1` with an exception set on failure.

Added in version 3.4.

type PyStructSequence_Desc

Part of the Stable ABI (including all members).
Contains the meta information of a struct sequence type to create.

const char *name

Fully qualified name of the type; null-terminated UTF-8 encoded. The name must contain the module name.

const char *doc

Pointer to docstring for the type or `NULL` to omit.

PyStructSequence_Field *fields

Pointer to `NULL`-terminated array with field names of the new type.

int n_in_sequence

Number of fields visible to the Python side (if used as tuple).

type PyStructSequence_Field

Part of the Stable ABI (including all members).
Describes a field of a struct sequence. As a struct sequence is modeled as a tuple, all fields are typed as PyObject*. The index in the `fields` array of the `PyStructSequence_Desc` determines which field of the struct sequence is described.

const char *name

Name for the field or `NULL` to end the list of named fields, set to **`PyStructSequence_UnnamedField`** to leave unnamed.

const char *doc

Field docstring or `NULL` to omit.

const char *const PyStructSequence_UnnamedField

Part of the Stable ABI since version 3.11.

Special value for a field name to leave it unnamed.

Changed in version 3.9: The type was changed from `char *`.

PyObject *PyStructSequence_New(PyTypeObject *type)

Return value: New reference. Part of the Stable ABI.

Creates an instance of *type*, which must have been created with **`PyStructSequence_NewType()`**.

Return `NULL` with an exception set on failure.

PyObject *PyStructSequence_GetItem(PyObject *p, Py_ssize_t pos)

Return value: Borrowed reference. Part of the Stable ABI.

Return the object at position *pos* in the struct sequence pointed to by *p*.

Bounds checking is performed as an assertion if Python is built in debug mode or **`with assertions`**.

PyObject *PyStructSequence_GET_ITEM(PyObject *p, Py_ssize_t pos)

Return value: Borrowed reference.

Alias to **`PyStructSequence_GetItem()`**.

Changed in version 3.13: Now implemented as an alias to `PyStructSequence_GetItem()`.

void PyStructSequence_SetItem(PyObject *p, Py_ssize_t pos, PyObject *o)

Part of the Stable ABI.

Sets the field at index *pos* of the struct sequence *p* to value *o*. Like `PyTuple_SET_ITEM()`, this should only be used to fill in brand new instances.

Bounds checking is performed as an assertion if Python is built in debug mode or `with assertions`.

Note

This function “steals” a reference to *o*.

void PyStructSequence_SET_ITEM(PyObject *p, Py_ssize_t *pos, PyObject *o)

Alias to `PyStructSequence_SetItem()`.

Changed in version 3.13: Now implemented as an alias to `PyStructSequence_SetItem()`.

List Objects

type PyListObject

This subtype of `PyObject` represents a Python list object.

PyTypeObject PyList_Type

Part of the Stable ABI.

This instance of `PyTypeObject` represents the Python list type. This is the same object as `list` in the Python layer.

int PyList_Check(PyObject *p)

Return true if *p* is a list object or an instance of a subtype of the list type. This function always succeeds.

int PyList_CheckExact(PyObject *p)

Return true if *p* is a list object, but not an instance of a subtype of the list type. This function always succeeds.

PyObject *PyList_New(Py_ssize_t len)

Return value: New reference. Part of the Stable ABI.

Return a new list of length *len* on success, or `NULL` on failure.

Note

If *len* is greater than zero, the returned list object's items are set to `NULL`. Thus you cannot use abstract API functions such as `PySequence_SetItem()` or expose the object to Python code before setting all items to a real object with `PyList_SetItem()` or `PyList_SET_ITEM()`. The following APIs are safe APIs before the list is fully initialized: `PyList_SetItem()` and `PyList_SET_ITEM()`.

Py_ssize_t PyList_Size(PyObject *list)

Part of the Stable ABI.

Return the length of the list object in *list*; this is equivalent to `len(list)` on a list object.

Py_ssize_t PyList_GET_SIZE(PyObject *list)

Similar to **`PyList_Size()`**, but without error checking.

PyObject *PyList_GetItemRef(PyObject *list, Py_ssize_t index)

Return value: New reference. Part of the Stable ABI since version 3.13.

Return the object at position *index* in the list pointed to by *list*. The position must be non-negative; indexing from the end of the list is not supported. If *index* is out of bounds (`<0 or >=len(list)`), return `NULL` and set an **`IndexError`** exception.

Added in version 3.13.

PyObject *PyList_GetItem(PyObject *list, Py_ssize_t index)

Return value: Borrowed reference. Part of the Stable ABI.

Like **`PyList_GetItemRef()`**, but returns a borrowed reference instead of a strong reference.

PyObject *PyList_GET_ITEM(PyObject *list, Py_ssize_t i)

Return value: Borrowed reference.

Similar to **`PyList_GetItem()`**, but without error checking.

int PyList_SetItem(PyObject *list, Py_ssize_t index, PyObject *item)

Part of the Stable ABI.

Set the item at index *index* in list to *item*. Return `0` on success. If *index* is out of bounds, return `-1` and set an **`IndexError`** exception.

Note

This function “steals” a reference to *item* and discards a reference to an item already in the list at the affected position.

void PyList_SET_ITEM(PyObject *list, Py_ssize_t i, PyObject *o)

Macro form of **`PyList_SetItem()`** without error checking. This is normally only used to fill in new lists where there is no previous content.

Bounds checking is performed as an assertion if Python is built in debug mode or **`with assertions`**.

Note

This macro “steals” a reference to *item*, and, unlike **`PyList_SetItem()`**, does *not* discard a reference to any item that is being replaced; any reference in *list* at position *i* will be leaked.

int PyList_Insert(PyObject *list, Py_ssize_t index, PyObject *item)

Part of the Stable ABI.

Insert the item *item* into list *list* in front of index *index*. Return `0` if successful; return `-1` and set an exception if unsuccessful. Analogous to `list.insert(index, item)`.

int PyList_Append(PyObject *list, PyObject *item)

Part of the Stable ABI.

Append the object *item* at the end of list *list*. Return `0` if successful; return `-1` and set an exception if unsuccessful. Analogous to `list.append(item)`.

PyObject *PyList_GetSlice(PyObject *list, Py_ssize_t low, Py_ssize_t high)

Return value: New reference. Part of the Stable ABI.
Return a list of the objects in *list* containing the objects *between low* and *high*. Return `NULL` and set an exception if unsuccessful. Analogous to `list[low:high]`. Indexing from the end of the list is not supported.

int PyList_SetSlice(PyObject *list, Py_ssize_t low, Py_ssize_t high, PyObject *itemlist)

Part of the Stable ABI.
Set the slice of *list* between *low* and *high* to the contents of *itemlist*. Analogous to `list[low:high] = itemlist`. The *itemlist* may be `NULL`, indicating the assignment of an empty list (slice deletion). Return `0` on success, `-1` on failure. Indexing from the end of the list is not supported.

int PyList_Extend(PyObject *list, PyObject *iterable)

Extend *list* with the contents of *iterable*. This is the same as `PyList_SetSlice(list, PY_SSIZE_T_MAX, PY_SSIZE_T_MAX, iterable)` and analogous to `list.extend(iterable)` or `list += iterable`.

Raise an exception and return `-1` if *list* is not a **`list`** object. Return 0 on success.

Added in version 3.13.

int PyList_Clear(PyObject *list)

Remove all items from *list*. This is the same as `PyList_SetSlice(list, 0, PY_SSIZE_T_MAX, NULL)` and analogous to `list.clear()` or `del list[:]`.

Raise an exception and return `-1` if *list* is not a **`list`** object. Return 0 on success.

Added in version 3.13.

int PyList_Sort(PyObject *list)

Part of the Stable ABI.

Sort the items of *list* in place. Return `0` on success, `-1` on failure. This is equivalent to `list.sort()`.

int PyList_Reverse(PyObject *list)

Part of the Stable ABI.

Reverse the items of *list* in place. Return `0` on success, `-1` on failure. This is the equivalent of `list.reverse()`.

PyObject *PyList_AsTuple(PyObject *list)

Return value: New reference. Part of the Stable ABI.

Return a new tuple object containing the contents of *list*; equivalent to `tuple(list)`.

Dictionary Objects

type PyDictObject

This subtype of `PyObject` represents a Python dictionary object.

PyTypeObject PyDict_Type

Part of the Stable ABI.

This instance of `PyTypeObject` represents the Python dictionary type. This is the same object as `dict` in the Python layer.

int PyDict_Check(PyObject *p)

Return true if *p* is a dict object or an instance of a subtype of the dict type. This function always succeeds.

int PyDict_CheckExact(PyObject *p)

Return true if *p* is a dict object, but not an instance of a subtype of the dict type. This function always succeeds.

PyObject *PyDict_New()

Return value: New reference. Part of the Stable ABI.

Return a new empty dictionary, or `NULL` on failure.

PyObject *PyDictProxy_New(PyObject *mapping)

Return value: New reference. Part of the Stable ABI.

Return a `types.MappingProxyType` object for a mapping which enforces read-only behavior. This is normally used to create a view to prevent modification of the dictionary for non-dynamic class types.

PyTypeObject PyDictProxy_Type

Part of the Stable ABI.

The type object for mapping proxy objects created by `PyDictProxy_New()` and for the read-only `__dict__` attribute of many built-in types. A `PyDictProxy_Type` instance provides a dynamic, read-only view of an underlying dictionary: changes to the underlying dictionary are reflected in the proxy, but the proxy itself does not support mutation operations. This corresponds to `types.MappingProxyType` in Python.

void PyDict_Clear(PyObject *p)

Part of the Stable ABI.
Empty an existing dictionary of all key-value pairs.

int PyDict_Contains(PyObject *p, PyObject *key)

Part of the Stable ABI.
Determine if dictionary *p* contains *key*. If an item in *p* matches *key*, return `1`, otherwise return `0`. On error, return `-1`. This is equivalent to the Python expression `key in p`.

int PyDict_ContainsString(PyObject *p, const char *key)

This is the same as `PyDict_Contains()`, but *key* is specified as a const char* UTF-8 encoded bytes string, rather than a PyObject*.

Added in version 3.13.

PyObject *PyDict_Copy(PyObject *p)

Return value: New reference. Part of the Stable ABI.
Return a new dictionary that contains the same key-value pairs as *p*.

int PyDict_SetItem(PyObject *p, PyObject *key, PyObject *val)

Part of the Stable ABI.

Insert *val* into the dictionary *p* with a key of *key*. *key* must be hashable; if it isn't, **`TypeError`** will be raised. Return `0` on success or `-1` on failure. This function *does not* steal a reference to *val*.

int PyDict_SetItemString(PyObject *p, const char *key, PyObject *val)

Part of the Stable ABI.

This is the same as **`PyDict_SetItem()`**, but *key* is specified as a const char* UTF-8 encoded bytes string, rather than a PyObject*.

int PyDict_DelItem(PyObject *p, PyObject *key)

Part of the Stable ABI.

Remove the entry in dictionary *p* with key *key*. *key* must be hashable; if it isn't, **`TypeError`** is raised. If *key* is not in the dictionary, **`KeyError`** is raised. Return `0` on success or `-1` on failure.

int PyDict_DelItemString(PyObject *p, const char *key)

Part of the Stable ABI.

This is the same as **`PyDict_DelItem()`**, but *key* is specified as a const char* UTF-8 encoded bytes string, rather than a PyObject*.

int PyDict_GetItemRef(PyObject *p, PyObject *key, PyObject **result)

Part of the Stable ABI since version 3.13.

Return a new strong reference to the object from dictionary *p* which has a key *key*:

- If the key is present, set **result* to a new strong reference to the value and return `1`.
- If the key is missing, set **result* to `NULL` and return `0`.
- On error, raise an exception and return `-1`.

Added in version 3.13.

See also the `PyObject_GetItem()` function.

PyObject *PyDict_GetItem(PyObject *p, PyObject *key)

Return value: Borrowed reference. Part of the Stable ABI. Return a borrowed reference to the object from dictionary *p* which has a key *key*. Return `NULL` if the key *key* is missing *without* setting an exception.

Note

Exceptions that occur while this calls `__hash__()` and `__eq__()` methods are silently ignored. Prefer the `PyDict_GetItemWithError()` function instead.

Changed in version 3.10: Calling this API without an attached thread state had been allowed for historical reason. It is no longer allowed.

PyObject *PyDict_GetItemWithError(PyObject *p, PyObject *key)

Return value: Borrowed reference. Part of the Stable ABI. Variant of `PyDict_GetItem()` that does not suppress exceptions. Return `NULL` **with** an exception set if an exception occurred. Return `NULL` **without** an exception set if the key wasn't present.

PyObject *PyDict_GetItemString(PyObject *p, const char *key)

Return value: Borrowed reference. Part of the Stable ABI. This is the same as `PyDict_GetItem()`, but *key* is specified as a const char* UTF-8 encoded bytes string, rather than a PyObject*.

Note

Exceptions that occur while this calls `__hash__()` and `__eq__()` methods or while creating the temporary `str` object are silently ignored. Prefer using the `PyDict_GetItemWithError()` function with your own `PyUnicode_FromString()` *key* instead.

int PyDict_GetItemStringRef(PyObject *p, const char *key, PyObject **result)

Part of the Stable ABI since version 3.13.
Similar to `PyDict_GetItemRef()`, but *key* is specified as a const char* UTF-8 encoded bytes string, rather than a PyObject*.

Added in version 3.13.

PyObject *PyDict_SetDefault(PyObject *p, PyObject *key, PyObject *defaultobj)

Return value: Borrowed reference.
This is the same as the Python-level `dict.setdefault()`. If present, it returns the value corresponding to *key* from the dictionary *p*. If the key is not in the dict, it is inserted with value *defaultobj* and *defaultobj* is returned. This function evaluates the hash function of *key* only once, instead of evaluating it independently for the lookup and the insertion.

Added in version 3.4.

int PyDict_SetDefaultRef(PyObject *p, PyObject *key, PyObject *default_value, PyObject **result)

Inserts *default_value* into the dictionary *p* with a key of *key* if the key is not already present in the dictionary. If *result* is not `NULL`, then **result* is set to a strong reference to either *default_value*, if the key was not present, or the existing value, if *key* was already present in the dictionary. Returns `1` if the key was present and *default_value* was not inserted, or `0` if the key was not present and *default_value* was inserted. On failure, returns `-1`, sets an exception, and sets `*result` to `NULL`.

For clarity: if you have a strong reference to *default_value* before calling this function, then after it returns, you hold a strong reference to both *default_value* and **result* (if it's not `NULL`). These may refer to the same object: in that case you hold two separate references to it.

Added in version 3.13.

int PyDict_Pop(PyObject *p, PyObject *key, PyObject **result)

Remove *key* from dictionary *p* and optionally return the removed value. Do not raise **`KeyError`** if the key is missing.

- If the key is present, set **result* to a new reference to the removed value if *result* is not `NULL`, and return `1`.
- If the key is missing, set **result* to `NULL` if *result* is not `NULL`, and return `0`.
- On error, raise an exception and return `-1`.

Similar to **`dict.pop()`**, but without the default value and not raising **`KeyError`** if the key is missing.

Added in version 3.13.

int PyDict_PopString(PyObject *p, const char *key, PyObject **result)

Similar to **`PyDict_Pop()`**, but *key* is specified as a const char* UTF-8 encoded bytes string, rather than a

PyObject*.

Added in version 3.13.

PyObject *PyDict_Items(PyObject *p)

Return value: New reference. Part of the Stable ABI.

Return a **`PyListObject`** containing all the items from the dictionary.

PyObject *PyDict_Keys(PyObject *p)

Return value: New reference. Part of the Stable ABI.

Return a **`PyListObject`** containing all the keys from the dictionary.

PyObject *PyDict_Values(PyObject *p)

Return value: New reference. Part of the Stable ABI.

Return a **`PyListObject`** containing all the values from the dictionary *p*.

Py_ssize_t PyDict_Size(PyObject *p)

Part of the Stable ABI.

Return the number of items in the dictionary. This is equivalent to `len(p)` on a dictionary.

Py_ssize_t PyDict_GET_SIZE(PyObject *p)

Similar to **`PyDict_Size()`**, but without error checking.

int PyDict_Next(PyObject *p, Py_ssize_t *ppos, PyObject **pkey, PyObject **pvalue)

Part of the Stable ABI.

Iterate over all key-value pairs in the dictionary *p*. The **`Py_ssize_t`** referred to by *ppos* must be initialized to `0` prior to the first call to this function to start the iteration; the function returns true for each pair in the dictionary, and false once all pairs have been reported. The parameters *pkey* and *pvalue* should either point to PyObject* variables that will be filled in with each key and value, respectively, or may be `NULL`. Any references returned through them are borrowed. *ppos* should not be altered during iteration. Its value represents offsets within the internal dictionary structure, and since the structure is sparse, the offsets are not consecutive.

For example:

```
PyObject *key, *value;
Py_ssize_t pos = 0;

while (PyDict_Next(self->dict, &pos, &key, &value)) {
    /* do something interesting with the values... */
    ...
}
```

The dictionary *p* should not be mutated during iteration. It is safe to modify the values of the keys as you iterate over the dictionary, but only so long as the set of keys does not change. For example:

```
PyObject *key, *value;
Py_ssize_t pos = 0;

while (PyDict_Next(self->dict, &pos, &key, &value)) {
    long i = PyLong_AsLong(value);
    if (i == -1 && PyErr_Occurred()) {
        return -1;
    }
    PyObject *o = PyLong_FromLong(i + 1);
    if (o == NULL)
        return -1;
    if (PyDict_SetItem(self->dict, key, o) < 0) {
        Py_DECREF(o);
        return -1;
    }
    Py_DECREF(o);
}
```

The function is not thread-safe in the free-threaded build without external synchronization. You can use `Py_BEGIN_CRITICAL_SECTION` to lock the dictionary while iterating over it:

```
Py_BEGIN_CRITICAL_SECTION(self->dict);
while (PyDict_Next(self->dict, &pos, &key, &value)) {
    ...
}
Py_END_CRITICAL_SECTION();
```

Note

On the free-threaded build, this function can be used safely inside a critical section. However, the references returned for *pkey* and *pvalue* are borrowed and are only valid while the critical section is held. If you need to use these objects outside the critical section or when the critical section can be suspended, create a strong reference (for example, using `Py_NewRef()`).

int PyDict_Merge(PyObject *a, PyObject *b, int override)

Part of the Stable ABI.

Iterate over mapping object *b* adding key-value pairs to dictionary *a*. *b* may be a dictionary, or any object supporting `PyMapping_Keys()` and `PyObject_GetItem()`. If *override* is true, existing pairs in *a* will be replaced if a matching key is found in *b*, otherwise pairs will only be added if there is not a matching key in *a*. Return `0` on success or `-1` if an exception was raised.

int PyDict_Update(PyObject *a, PyObject *b)

Part of the Stable ABI.

This is the same as `PyDict_Merge(a, b, 1)` in C, and is similar to `a.update(b)` in Python except that `PyDict_Update()` doesn’t fall back to the iterating over a sequence of key value pairs if the second argument has no “keys” attribute. Return `0` on success or `-1` if an exception was raised.

int PyDict_MergeFromSeq2(PyObject *a, PyObject *seq2, int override)

Part of the Stable ABI.

Update or merge into dictionary *a*, from the key-value pairs in *seq2*. *seq2* must be an iterable object producing iterable objects of length 2, viewed as key-value pairs. In case of duplicate keys, the last wins if *override* is true, else the first wins. Return `0` on success or `-1` if an exception was raised. Equivalent Python (except for the return value):

```
def PyDict_MergeFromSeq2(a, seq2, override):
    for key, value in seq2:
        if override or key not in a:
            a[key] = value
```

int PyDict_AddWatcher(PyDict_WatchCallback callback)

Register *callback* as a dictionary watcher. Return a non-negative integer id which must be passed to future calls to **`PyDict_Watch()`**. In case of error (e.g. no more watcher IDs available), return `-1` and set an exception.

Added in version 3.12.

int PyDict_ClearWatcher(int watcher_id)

Clear watcher identified by *watcher_id* previously returned from **`PyDict_AddWatcher()`**. Return `0` on success, `-1` on error (e.g. if the given *watcher_id* was never registered.)

Added in version 3.12.

int PyDict_Watch(int watcher_id, PyObject *dict)

Mark dictionary *dict* as watched. The callback granted *watcher_id* by **`PyDict_AddWatcher()`** will be called when *dict* is modified or deallocated. Return `0` on success or `-1` on error.

Added in version 3.12.

int PyDict_Unwatch(int watcher_id, PyObject *dict)

Mark dictionary *dict* as no longer watched. The callback granted *watcher_id* by **`PyDict_AddWatcher()`** will no longer be called when *dict* is modified or deallocated. The dict must previously have been watched by this watcher. Return `0` on success or `-1` on error.

Added in version 3.12.

type PyDict_WatchEvent

Enumeration of possible dictionary watcher events: `PyDict_EVENT_ADDED`, `PyDict_EVENT_MODIFIED`, `PyDict_EVENT_DELETED`, `PyDict_EVENT_CLONED`, `PyDict_EVENT_CLEARED`, or `PyDict_EVENT_DEALLOCATED`.

Added in version 3.12.

typedef int (*PyDict_WatchCallback)(PyDict_WatchEvent event, PyObject *dict, PyObject *key, PyObject *new_value)

Type of a dict watcher callback function.

If *event* is `PyDict_EVENT_CLEARED` or `PyDict_EVENT_DEALLOCATED`, both *key* and *new_value* will be `NULL`. If *event* is `PyDict_EVENT_ADDED` or `PyDict_EVENT_MODIFIED`, *new_value* will be the new value for *key*. If *event* is `PyDict_EVENT_DELETED`, *key* is being deleted from the dictionary and *new_value* will be `NULL`.

`PyDict_EVENT_CLONED` occurs when *dict* was previously empty and another dict is merged into it. To maintain efficiency of this operation, per-key `PyDict_EVENT_ADDED` events are not issued in this case; instead a single `PyDict_EVENT_CLONED` is issued, and *key* will be the source dictionary.

The callback may inspect but must not modify *dict*; doing so could have unpredictable effects, including infinite recursion. Do not trigger Python code execution in the callback, as it could modify the dict as a side effect.

If *event* is `PyDict_EVENT_DEALLOCATED`, taking a new reference in the callback to the about-to-be-destroyed dictionary will resurrect it and prevent it from being freed at this time. When the resurrected object is destroyed later, any watcher callbacks active at that time will be called again.

Callbacks occur before the notified modification to *dict* takes place, so the prior state of *dict* can be inspected.

If the callback sets an exception, it must return `-1`; this exception will be printed as an unraisable exception using **`PyErr_WriteUnraisable()`**. Otherwise it should return `0`.

There may already be a pending exception set on entry to the callback. In this case, the callback should return `0` with the same exception still set. This means the callback may not call any other API that can set an exception unless it saves and clears the exception state first, and restores it before returning.

Added in version 3.12.

Dictionary View Objects

int PyDictViewSet_Check(PyObject *op)

Return true if *op* is a view of a set inside a dictionary. This is currently equivalent to PyDictKeys_Check(op) || PyDictItems_Check(op). This function always succeeds.

PyTypeObject PyDictKeys_Type

Part of the Stable ABI.

Type object for a view of dictionary keys. In Python, this is the type of the object returned by **`dict.keys()`**.

int PyDictKeys_Check(PyObject *op)

Return true if *op* is an instance of a dictionary keys view. This function always succeeds.

PyTypeObject PyDictValues_Type

Part of the Stable ABI.

Type object for a view of dictionary values. In Python, this is the type of the object returned by **`dict.values()`**.

int PyDictValues_Check(PyObject *op)

Return true if *op* is an instance of a dictionary values view. This function always succeeds.

PyTypeObject PyDictItems_Type

Part of the Stable ABI.

Type object for a view of dictionary items. In Python, this is the type of the object returned by **`dict.items()`**.

int PyDictItems_Check(PyObject *op)

Return true if *op* is an instance of a dictionary items view. This function always succeeds.

Ordered Dictionaries

Python’s C API provides interface for **`collections.OrderedDict`** from C. Since Python 3.7, dictionaries are ordered by default, so there is usually little need for these functions; prefer `PyDict*` where possible.

PyTypeObject PyODict_Type

Type object for ordered dictionaries. This is the same object as **`collections.OrderedDict`** in the Python layer.

int PyODict_Check(PyObject *od)

Return true if *od* is an ordered dictionary object or an instance of a subtype of the **`OrderedDict`** type. This function always succeeds.

int PyODict_CheckExact(PyObject *od)

Return true if *od* is an ordered dictionary object, but not an instance of a subtype of the **`OrderedDict`** type. This function always succeeds.

PyTypeObject PyODictKeys_Type

Analogous to **`PyDictKeys_Type`** for ordered dictionaries.

PyTypeObject PyODictValues_Type

Analogous to **`PyDictValues_Type`** for ordered dictionaries.

PyTypeObject PyODictItems_Type

Analogous to **`PyDictItems_Type`** for ordered dictionaries.

PyObject *PyODict_New(void)

Return a new empty ordered dictionary, or `NULL` on failure.

This is analogous to **`PyDict_New()`**.

int PyODict_SetItem(PyObject *od, PyObject *key, PyObject *value)

Insert *value* into the ordered dictionary *od* with a key of *key*. Return `0` on success or `-1` with an exception set on failure.

This is analogous to **`PyDict_SetItem()`**.

int PyODict_DelItem(PyObject *od, PyObject *key)

Remove the entry in the ordered dictionary *od* with key *key*. Return `0` on success or `-1` with an exception set on failure.

This is analogous to **`PyDict_DelItem()`**.

These are soft deprecated aliases to `PyDict` APIs:

`PyODict`	**`PyDict`**
PyODict_GetItem(od, key)	**`PyDict_GetItem()`**
PyODict_GetItemWithError(od, key)	**`PyDict_GetItemWithError()`**
PyODict_GetItemString(od, key)	**`PyDict_GetItemString()`**
PyODict_Contains(od, key)	**`PyDict_Contains()`**
PyODict_Size(od)	**`PyDict_Size()`**
PyODict_SIZE(od)	**`PyDict_GET_SIZE()`**

Set Objects

This section details the public API for `set` and `frozenset` objects. Any functionality not listed below is best accessed using either the abstract object protocol (including `PyObject_CallMethod()`, `PyObject_RichCompareBool()`, `PyObject_Hash()`, `PyObject_Repr()`, `PyObject_IsTrue()`, `PyObject_Print()`, and `PyObject_GetIter()`) or the abstract number protocol (including `PyNumber_And()`, `PyNumber_Subtract()`, `PyNumber_Or()`, `PyNumber_Xor()`, `PyNumber_InPlaceAnd()`, `PyNumber_InPlaceSubtract()`, `PyNumber_InPlaceOr()`, and `PyNumber_InPlaceXor()`).

type PySetObject

This subtype of `PyObject` is used to hold the internal data for both `set` and `frozenset` objects. It is like a `PyDictObject` in that it is a fixed size for small sets (much like tuple storage) and will point to a separate, variable sized block of memory for medium and large sized sets (much like list storage). None of the fields of this structure should be considered public and all are subject to change. All access should be done through the documented API rather than by manipulating the values in the structure.

PyTypeObject PySet_Type

Part of the Stable ABI.

This is an instance of `PyTypeObject` representing the Python `set` type.

PyTypeObject PyFrozenSet_Type

Part of the Stable ABI.

This is an instance of `PyTypeObject` representing the Python `frozenset` type.

The following type check macros work on pointers to any Python object. Likewise, the constructor functions work with any iterable Python object.

int PySet_Check(PyObject *p)

Return true if *p* is a **`set`** object or an instance of a subtype. This function always succeeds.

int PyFrozenSet_Check(PyObject *p)

Return true if *p* is a **`frozenset`** object or an instance of a subtype. This function always succeeds.

int PyAnySet_Check(PyObject *p)

Return true if *p* is a **`set`** object, a **`frozenset`** object, or an instance of a subtype. This function always succeeds.

int PySet_CheckExact(PyObject *p)

Return true if *p* is a **`set`** object but not an instance of a subtype. This function always succeeds.

Added in version 3.10.

int PyAnySet_CheckExact(PyObject *p)

Return true if *p* is a **`set`** object or a **`frozenset`** object but not an instance of a subtype. This function always succeeds.

int PyFrozenSet_CheckExact(PyObject *p)

Return true if *p* is a **`frozenset`** object but not an instance of a subtype. This function always succeeds.

PyObject *PySet_New(PyObject *iterable)

Return value: New reference. Part of the Stable ABI.
Return a new **`set`** containing objects returned by the *iterable*. The *iterable* may be `NULL` to create a new empty

set. Return the new set on success or `NULL` on failure. Raise **`TypeError`** if *iterable* is not actually iterable. The constructor is also useful for copying a set (`c=set(s)`).

PyObject *PyFrozenSet_New(PyObject *iterable)

Return value: New reference. Part of the Stable ABI.
Return a new **`frozenset`** containing objects returned by the *iterable*. The *iterable* may be `NULL` to create a new empty frozenset. Return the new set on success or `NULL` on failure. Raise **`TypeError`** if *iterable* is not actually iterable.

The following functions and macros are available for instances of **`set`** or **`frozenset`** or instances of their subtypes.

Py_ssize_t PySet_Size(PyObject *anyset)

Part of the Stable ABI.
Return the length of a **`set`** or **`frozenset`** object. Equivalent to `len(anyset)`. Raises a **`SystemError`** if *anyset* is not a **`set`**, **`frozenset`**, or an instance of a subtype.

Py_ssize_t PySet_GET_SIZE(PyObject *anyset)

Macro form of **`PySet_Size()`** without error checking.

int PySet_Contains(PyObject *anyset, PyObject *key)

Part of the Stable ABI.
Return `1` if found, `0` if not found, and `-1` if an error is encountered. Unlike the Python **`__contains__()`** method, this function does not automatically convert unhashable sets into temporary frozensets. Raise a **`TypeError`** if the *key* is unhashable. Raise **`SystemError`** if *anyset* is not a **`set`**, **`frozenset`**, or an instance of a subtype.

int PySet_Add(PyObject *set, PyObject *key)

Part of the Stable ABI.

Add *key* to a **`set`** instance. Also works with **`frozenset`** instances (like **`PyTuple_SetItem()`** it can be used to fill in the values of brand new frozensets before they are exposed to other code). Return `0` on success or `-1` on failure. Raise a **`TypeError`** if the *key* is unhashable. Raise a **`MemoryError`** if there is no room to grow. Raise a **`SystemError`** if *set* is not an instance of **`set`** or its subtype.

The following functions are available for instances of **`set`** or its subtypes but not for instances of **`frozenset`** or its subtypes.

int PySet_Discard(PyObject *set, PyObject *key)

Part of the Stable ABI.

Return `1` if found and removed, `0` if not found (no action taken), and `-1` if an error is encountered. Does not raise **`KeyError`** for missing keys. Raise a **`TypeError`** if the *key* is unhashable. Unlike the Python **`discard()`** method, this function does not automatically convert unhashable sets into temporary frozensets. Raise **`SystemError`** if *set* is not an instance of **`set`** or its subtype.

PyObject *PySet_Pop(PyObject *set)

Return value: New reference. Part of the Stable ABI.

Return a new reference to an arbitrary object in the *set*, and removes the object from the *set*. Return `NULL` on failure. Raise **`KeyError`** if the set is empty. Raise a **`SystemError`** if *set* is not an instance of **`set`** or its subtype.

int PySet_Clear(PyObject *set)

Part of the Stable ABI.

Empty an existing set of all elements. Return `0` on success. Return `-1` and raise **`SystemError`** if *set* is not an instance of **`set`** or its subtype.

Deprecated API

PySet_MINSIZE

A soft deprecated constant representing the size of an internal preallocated table inside **`PySetObject`** instances.

This is documented solely for completeness, as there are no guarantees that a given version of CPython uses preallocated tables with a fixed size. In code that does not deal with unstable set internals, **`PySet_MINSIZE`** can be replaced with a small constant like `8`.

If looking for the size of a set, use **`PySet_Size()`** instead.

Function Objects

There are a few functions specific to Python functions.

type PyFunctionObject

The C structure used for functions.

PyTypeObject PyFunction_Type

This is an instance of **`PyTypeObject`** and represents the Python function type. It is exposed to Python programmers as `types.FunctionType`.

int PyFunction_Check(PyObject *o)

Return true if *o* is a function object (has type **`PyFunction_Type`**). The parameter must not be `NULL`. This function always succeeds.

PyObject *PyFunction_New(PyObject *code, PyObject *globals)

Return value: New reference.
Return a new function object associated with the code object *code*. *globals* must be a dictionary with the global variables accessible to the function.

The function's docstring and name are retrieved from the code object. **`__module__`** is retrieved from *globals*. The argument defaults, annotations and closure are set to `NULL`. **`__qualname__`** is set to the same value as the code object's **`co_qualname`** field.

PyObject *PyFunction_NewWithQualName(PyObject *code, PyObject *globals, PyObject *qualname)

Return value: New reference.

As `PyFunction_New()`, but also allows setting the function object's `__qualname__` attribute. *qualname* should be a unicode object or `NULL`; if `NULL`, the **`__qualname__`** attribute is set to the same value as the code object's `co_qualname` field.

Added in version 3.3.

PyObject *PyFunction_GetCode(PyObject *op)

Return value: Borrowed reference.
Return the code object associated with the function object *op*.

PyObject *PyFunction_GetGlobals(PyObject *op)

Return value: Borrowed reference.
Return the globals dictionary associated with the function object *op*.

PyObject *PyFunction_GetModule(PyObject *op)

Return value: Borrowed reference.
Return a borrowed reference to the `__module__` attribute of the function object *op*. It can be *NULL*.

This is normally a `string` containing the module name, but can be set to any other object by Python code.

PyObject *PyFunction_GetDefaults(PyObject *op)

Return value: Borrowed reference.
Return the argument default values of the function object *op*. This can be a tuple of arguments or `NULL`.

int PyFunction_SetDefaults(PyObject *op, PyObject *defaults)

Set the argument default values for the function object *op*. *defaults* must be `Py_None` or a tuple.

Raises **`SystemError`** and returns `-1` on failure.

void PyFunction_SetVectorcall(PyFunctionObject *func, vectorcallfunc vectorcall)

Set the vectorcall field of a given function object *func*.

Warning: extensions using this API must preserve the behavior of the unaltered (default) vectorcall function!

Added in version 3.12.

PyObject *PyFunction_GetKwDefaults(PyObject *op)

Return value: Borrowed reference.
Return the keyword-only argument default values of the function object *op*. This can be a dictionary of arguments or `NULL`.

int PyFunction_SetKwDefaults(PyObject *op, PyObject *defaults)

Set the keyword-only argument default values of the function object *op*. *defaults* must be a dictionary of keyword-only arguments or `Py_None`.

This function returns `0` on success, and returns `-1` with an exception set on failure.

PyObject *PyFunction_GetClosure(PyObject *op)

Return value: Borrowed reference.
Return the closure associated with the function object *op*. This can be `NULL` or a tuple of cell objects.

int PyFunction_SetClosure(PyObject *op, PyObject *closure)

Set the closure associated with the function object *op*. *closure* must be `Py_None` or a tuple of cell objects.

Raises **`SystemError`** and returns `-1` on failure.

PyObject *PyFunction_GetAnnotations(PyObject *op)

Return value: Borrowed reference.
Return the annotations of the function object *op*. This can be a mutable dictionary or `NULL`.

int PyFunction_SetAnnotations(PyObject *op, PyObject *annotations)

Set the annotations for the function object *op*. *annotations* must be a dictionary or `Py_None`.

Raises **`SystemError`** and returns `-1` on failure.

PyObject *PyFunction_GET_CODE(PyObject *op)
PyObject *PyFunction_GET_GLOBALS(PyObject *op)
PyObject *PyFunction_GET_MODULE(PyObject *op)
PyObject *PyFunction_GET_DEFAULTS(PyObject *op)
PyObject *PyFunction_GET_KW_DEFAULTS(PyObject *op)
PyObject *PyFunction_GET_CLOSURE(PyObject *op)
PyObject *PyFunction_GET_ANNOTATIONS(PyObject *op)

Return value: Borrowed reference.
These functions are similar to their `PyFunction_Get*` counterparts, but do not do type checking. Passing anything other than an instance of **`PyFunction_Type`** is undefined behavior.

int PyFunction_AddWatcher(PyFunction_WatchCallback callback)

Register *callback* as a function watcher for the current interpreter. Return an ID which may be passed to **`PyFunction_ClearWatcher()`**. In case of error (e.g. no more watcher IDs available), return `-1` and set an exception.

Added in version 3.12.

int PyFunction_ClearWatcher(int watcher_id)

Clear watcher identified by *watcher_id* previously returned from **`PyFunction_AddWatcher()`** for the current interpreter. Return `0` on success, or `-1` and set an exception on error (e.g. if the given *watcher_id* was never registered.)

Added in version 3.12.

type PyFunction_WatchEvent

Enumeration of possible function watcher events:

- `PyFunction_EVENT_CREATE`
- `PyFunction_EVENT_DESTROY`
- `PyFunction_EVENT_MODIFY_CODE`
- `PyFunction_EVENT_MODIFY_DEFAULTS`
- `PyFunction_EVENT_MODIFY_KWDEFAULTS`

Added in version 3.12.

typedef int (*PyFunction_WatchCallback)(PyFunction_WatchEvent event, PyFunctionObject *func, PyObject *new_value)

Type of a function watcher callback function.

If *event* is `PyFunction_EVENT_CREATE` or `PyFunction_EVENT_DESTROY` then *new_value* will be `NULL`. Otherwise, *new_value* will hold a borrowed reference to the new value that is about to be stored in *func* for the attribute that is being modified.

The callback may inspect but must not modify *func*; doing so could have unpredictable effects, including infinite recursion.

If *event* is `PyFunction_EVENT_CREATE`, then the callback is invoked after *func* has been fully initialized. Otherwise, the callback is invoked before the modification to *func* takes place, so the prior state of *func* can be inspected. The runtime is permitted to optimize away the creation of function objects when possible. In such cases no event will be emitted. Although this creates the possibility of an observable difference of runtime behavior depending on optimization decisions, it does not change the semantics of the Python code being executed.

If *event* is `PyFunction_EVENT_DESTROY`, Taking a reference in the callback to the about-to-be-destroyed function will resurrect it, preventing it from being freed at this time. When the resurrected object is destroyed later, any watcher callbacks active at that time will be called again.

If the callback sets an exception, it must return `-1`; this exception will be printed as an unraisable exception using **`PyErr_WriteUnraisable()`**. Otherwise it should return `0`.

There may already be a pending exception set on entry to the callback. In this case, the callback should return `0` with the same exception still set. This means the callback may not call any other API that can set an exception unless it saves and clears the exception state first, and restores it before returning.

Added in version 3.12.

Instance Method Objects

An instance method is a wrapper for a **`PyCFunction`** and the new way to bind a **`PyCFunction`** to a class object. It replaces the former call `PyMethod_New(func, NULL, class)`.

PyTypeObject PyInstanceMethod_Type

This instance of **`PyTypeObject`** represents the Python instance method type. It is not exposed to Python programs.

int PyInstanceMethod_Check(PyObject *o)

Return true if *o* is an instance method object (has type **`PyInstanceMethod_Type`**). The parameter must not be `NULL`. This function always succeeds.

PyObject *PyInstanceMethod_New(PyObject *func)

Return value: New reference.
Return a new instance method object, with *func* being any callable object. *func* is the function that will be called when the instance method is called.

PyObject *PyInstanceMethod_Function(PyObject *im)

Return value: Borrowed reference.
Return the function object associated with the instance method *im*.

PyObject *PyInstanceMethod_GET_FUNCTION(PyObject *im)

Return value: Borrowed reference.
Macro version of **`PyInstanceMethod_Function()`** which avoids error checking.

Method Objects

Methods are bound function objects. Methods are always bound to an instance of a user-defined class. Unbound methods (methods bound to a class object) are no longer available.

PyTypeObject PyMethod_Type

This instance of **`PyTypeObject`** represents the Python method type. This is exposed to Python programs as `types.MethodType`.

int PyMethod_Check(PyObject *o)

Return true if *o* is a method object (has type **`PyMethod_Type`**). The parameter must not be `NULL`. This function always succeeds.

PyObject *PyMethod_New(PyObject *func, PyObject *self)

Return value: New reference.
Return a new method object, with *func* being any callable object and *self* the instance the method should be bound. *func* is the function that will be called when the method is called. *self* must not be `NULL`.

PyObject *PyMethod_Function(PyObject *meth)

Return value: Borrowed reference.
Return the function object associated with the method *meth*.

PyObject *PyMethod_GET_FUNCTION(PyObject *meth)

Return value: Borrowed reference.
Macro version of **`PyMethod_Function()`** which avoids error checking.

PyObject *PyMethod_Self(PyObject *meth)

Return value: Borrowed reference.
Return the instance associated with the method *meth*.

PyObject *PyMethod_GET_SELF(PyObject *meth)

Return value: Borrowed reference.
Macro version of **`PyMethod_Self()`** which avoids error checking.

Cell Objects

“Cell” objects are used to implement variables referenced by multiple scopes. For each such variable, a cell object is created to store the value; the local variables of each stack frame that references the value contain a reference to the cells from outer scopes which also use that variable. When the value is accessed, the value contained in the cell is used instead of the cell object itself. This de-referencing of the cell object requires support from the generated byte-code; these are not automatically de-referenced when accessed. Cell objects are not likely to be useful elsewhere.

type PyCellObject

The C structure used for cell objects.

PyTypeObject PyCell_Type

The type object corresponding to cell objects.

int PyCell_Check(PyObject *ob)

Return true if *ob* is a cell object; *ob* must not be `NULL`. This function always succeeds.

PyObject *PyCell_New(PyObject *ob)

Return value: New reference.
Create and return a new cell object containing the value *ob*. The parameter may be `NULL`.

PyObject *PyCell_Get(PyObject *cell)

Return value: New reference.
Return the contents of the cell *cell*, which can be `NULL`. If *cell* is not a cell object, returns `NULL` with an exception set.

PyObject *PyCell_GET(PyObject *cell)

Return value: Borrowed reference.
Return the contents of the cell *cell*, but without checking that *cell* is non-`NULL` and a cell object.

int PyCell_Set(PyObject *cell, PyObject *value)

Set the contents of the cell object *cell* to *value*. This releases the reference to any current content of the cell. *value* may be `NULL`. *cell* must be non-`NULL`.

On success, return `0`. If *cell* is not a cell object, set an exception and return `-1`.

void PyCell_SET(PyObject *cell, PyObject *value)

Sets the value of the cell object *cell* to *value*. No reference counts are adjusted, and no checks are made for safety; *cell* must be non-`NULL` and must be a cell object.

Code Objects

Code objects are a low-level detail of the CPython implementation. Each one represents a chunk of executable code that hasn’t yet been bound into a function.

type PyCodeObject

The C structure of the objects used to describe code objects. The fields of this type are subject to change at any time.

PyTypeObject PyCode_Type

This is an instance of **`PyTypeObject`** representing the Python code object.

int PyCode_Check(PyObject *co)

Return true if *co* is a code object. This function always succeeds.

Py_ssize_t PyCode_GetNumFree(PyCodeObject *co)

Return the number of free (closure) variables in a code object.

int PyUnstable_Code_GetFirstFree(PyCodeObject *co)

This is Unstable API. It may change without warning in minor releases.

Return the position of the first free (closure) variable in a code object.

Changed in version 3.13: Renamed from `PyCode_GetFirstFree` as part of Unstable C API. The old name is deprecated, but will remain available until the signature changes again.

PyCodeObject *PyUnstable_Code_New(int argcount, int kwonlyargcount, int nlocals, int stacksize, int flags, PyObject *code, PyObject *consts, PyObject *names, PyObject *varnames, PyObject *freevars, PyObject *cellvars, PyObject *filename, PyObject *name, PyObject *qualname, int firstlineno, PyObject *linetable, PyObject *exceptiontable)

> *This is Unstable API. It may change without warning in minor releases.*

> Return a new code object. If you need a dummy code object to create a frame, use **`PyCode_NewEmpty()`** instead.
>
> Since the definition of the bytecode changes often, calling **`PyUnstable_Code_New()`** directly can bind you to a precise Python version.
>
> The many arguments of this function are inter-dependent in complex ways, meaning that subtle changes to values are likely to result in incorrect execution or VM crashes. Use this function only with extreme care.
>
> *Changed in version 3.11:* Added `qualname` and `exceptiontable` parameters.
>
> *Changed in version 3.12:* Renamed from `PyCode_New` as part of Unstable C API. The old name is deprecated, but will remain available until the signature changes again.

PyCodeObject *PyUnstable_Code_NewWithPosOnlyArgs(int argcount, int posonlyargcount, int kwonlyargcount, int nlocals, int stacksize, int flags, PyObject *code, PyObject *consts, PyObject *names, PyObject *varnames, PyObject *freevars, PyObject *cellvars, PyObject *filename, PyObject *name, PyObject *qualname, int firstlineno, PyObject *linetable, PyObject *exceptiontable)

This is Unstable API. It may change without warning in minor releases.

Similar to `PyUnstable_Code_New()`, but with an extra "posonlyargcount" for positional-only arguments. The same caveats that apply to `PyUnstable_Code_New` also apply to this function.

Added in version 3.8: as `PyCode_NewWithPosOnlyArgs`

Changed in version 3.11: Added `qualname` and `exceptiontable` parameters.

Changed in version 3.12: Renamed to `PyUnstable_Code_NewWithPosOnlyArgs`. The old name is deprecated, but will remain available until the signature changes again.

PyCodeObject *PyCode_NewEmpty(const char *filename, const char *funcname, int firstlineno)

Return value: New reference.
Return a new empty code object with the specified filename, function name, and first line number. The resulting code object will raise an `Exception` if executed.

int PyCode_Addr2Line(PyCodeObject *co, int byte_offset)

Return the line number of the instruction that occurs on or before `byte_offset` and ends after it. If you just need the line number of a frame, use `PyFrame_GetLineNumber()` instead.

For efficiently iterating over the line numbers in a code object, use **the API described in PEP 626** [https://peps.python.org/pep-0626/#out-of-process-debuggers-and-profilers].

int PyCode_Addr2Location(PyObject *co, int byte_offset, int *start_line, int *start_column, int *end_line, int *end_column)

Sets the passed `int` pointers to the source code line and column numbers for the instruction at `byte_offset`. Sets the value to `0` when information is not available for any particular element.

Returns `1` if the function succeeds and 0 otherwise.

Added in version 3.11.

PyObject *PyCode_GetCode(PyCodeObject *co)

Equivalent to the Python code `getattr(co, 'co_code')`. Returns a strong reference to a **`PyBytesObject`** representing the bytecode in a code object. On error, `NULL` is returned and an exception is raised.

This `PyBytesObject` may be created on-demand by the interpreter and does not necessarily represent the bytecode actually executed by CPython. The primary use case for this function is debuggers and profilers.

Added in version 3.11.

PyObject *PyCode_GetVarnames(PyCodeObject *co)

Equivalent to the Python code `getattr(co, 'co_varnames')`. Returns a new reference to a **`PyTupleObject`** containing the names of the local variables. On error, `NULL` is returned and an exception is raised.

Added in version 3.11.

PyObject *PyCode_GetCellvars(PyCodeObject *co)

Equivalent to the Python code `getattr(co, 'co_cellvars')`. Returns a new reference to a **`PyTupleObject`** containing the names of the local variables that are referenced by nested functions. On error, `NULL` is returned and an exception is raised.

Added in version 3.11.

PyObject *PyCode_GetFreevars(PyCodeObject *co)

Equivalent to the Python code `getattr(co, 'co_freevars')`. Returns a new reference to a **`PyTupleObject`** containing the names of the free (closure) variables. On error, `NULL` is returned and an exception is raised.

Added in version 3.11.

int PyCode_AddWatcher(PyCode_WatchCallback callback)

Register *callback* as a code object watcher for the current interpreter. Return an ID which may be passed to **`PyCode_ClearWatcher()`**. In case of error (e.g. no more watcher IDs available), return `-1` and set an exception.

Added in version 3.12.

int PyCode_ClearWatcher(int watcher_id)

Clear watcher identified by *watcher_id* previously returned from **`PyCode_AddWatcher()`** for the current interpreter. Return `0` on success, or `-1` and set an exception on error (e.g. if the given *watcher_id* was never registered.)

Added in version 3.12.

type PyCodeEvent

Enumeration of possible code object watcher events: - `PY_CODE_EVENT_CREATE` - `PY_CODE_EVENT_DESTROY`

Added in version 3.12.

typedef int (*PyCode_WatchCallback)(PyCodeEvent event, PyCodeObject *co)

Type of a code object watcher callback function.

If *event* is `PY_CODE_EVENT_CREATE`, then the callback is invoked after *co* has been fully initialized. Otherwise, the callback is invoked before the destruction of *co* takes place, so the prior state of *co* can be inspected.

If *event* is `PY_CODE_EVENT_DESTROY`, taking a reference in the callback to the about-to-be-destroyed code object will resurrect it and prevent it from being freed at this time. When the resurrected object is destroyed later, any watcher callbacks active at that time will be called again.

Users of this API should not rely on internal runtime implementation details. Such details may include, but are not limited to, the exact order and timing of creation and destruction of code objects. While changes in these details may result in differences observable by watchers (including whether a callback is invoked or not), it does not change the semantics of the Python code being executed.

If the callback sets an exception, it must return `-1`; this exception will be printed as an unraisable exception using **`PyErr_WriteUnraisable()`**. Otherwise it should return `0`.

There may already be a pending exception set on entry to the callback. In this case, the callback should return `0` with the same exception still set. This means the callback may not call any other API that can set an exception unless it saves and clears the exception state first, and restores it before returning.

Added in version 3.12.

PyObject *PyCode_Optimize(PyObject *code, PyObject *consts, PyObject *names, PyObject *lnotab_obj)

This is a soft deprecated function that does nothing.

Prior to Python 3.10, this function would perform basic optimizations to a code object.

Changed in version 3.10: This function now does nothing.

Code Object Flags

Code objects contain a bit-field of flags, which can be retrieved as the `co_flags` Python attribute (for example using `PyObject_GetAttrString()`), and set using a *flags* argument to `PyUnstable_Code_New()` and similar functions.

Flags whose names start with `CO_FUTURE_` correspond to features normally selectable by future statements. These flags can be used in `PyCompilerFlags.cf_flags`. Note that many `CO_FUTURE_` flags are mandatory in current versions of Python, and setting them has no effect.

The following flags are available. For their meaning, see the linked documentation of their Python equivalents.

Flag	**Meaning**
CO_OPTIMIZED	`inspect.CO_OPTIMIZED`
CO_NEWLOCALS	`inspect.CO_NEWLOCALS`
CO_VARARGS	`inspect.CO_VARARGS`
CO_VARKEYWORDS	`inspect.CO_VARKEYWORDS`

Flag	Meaning
CO_NESTED	`inspect.CO_NESTED`
CO_GENERATOR	`inspect.CO_GENERATOR`
CO_COROUTINE	`inspect.CO_COROUTINE`
CO_ITERABLE_COROUTINE	`inspect.CO_ITERABLE_COROUTINE`
CO_ASYNC_GENERATOR	`inspect.CO_ASYNC_GENERATOR`
CO_HAS_DOCSTRING	`inspect.CO_HAS_DOCSTRING`
CO_METHOD	`inspect.CO_METHOD`
CO_FUTURE_DIVISION	no effect (`__future__.division`)
CO_FUTURE_ABSOLUTE_IMPORT	no effect (`__future__.absolute_import`)
CO_FUTURE_WITH_STATEMENT	no effect (`__future__.with_statement`)
CO_FUTURE_PRINT_FUNCTION	no effect (`__future__.print_function`)

Flag	Meaning
CO_FUTURE_UNICODE_LITERALS	no effect (`__future__.unicode_literals`)
CO_FUTURE_GENERATOR_STOP	no effect (`__future__.generator_stop`)
CO_FUTURE_ANNOTATIONS	`__future__.annotations`

Extra information

To support low-level extensions to frame evaluation, such as external just-in-time compilers, it is possible to attach arbitrary extra data to code objects.

These functions are part of the unstable C API tier: this functionality is a CPython implementation detail, and the API may change without deprecation warnings.

Py_ssize_t PyUnstable_Eval_RequestCodeExtraIndex(freefunc free)

> *This is Unstable API. It may change without warning in minor releases.*

> Return a new opaque index value used to adding data to code objects.
>
> You generally call this function once (per interpreter) and use the result with `PyCode_GetExtra` and `PyCode_SetExtra` to

manipulate data on individual code objects.

If *free* is not `NULL`: when a code object is deallocated, *free* will be called on non-`NULL` data stored under the new index. Use **`Py_DecRef()`** when storing **`PyObject`**.

Added in version 3.6: as `_PyEval_RequestCodeExtraIndex`

Changed in version 3.12: Renamed to `PyUnstable_Eval_RequestCodeExtraIndex`. The old private name is deprecated, but will be available until the API changes.

int PyUnstable_Code_GetExtra(PyObject *code, Py_ssize_t index, void **extra)

This is Unstable API. It may change without warning in minor releases.

Set *extra* to the extra data stored under the given index. Return 0 on success. Set an exception and return -1 on failure.

If no data was set under the index, set *extra* to `NULL` and return 0 without setting an exception.

Added in version 3.6: as `_PyCode_GetExtra`

Changed in version 3.12: Renamed to `PyUnstable_Code_GetExtra`. The old private name is deprecated, but will be available until the API changes.

int PyUnstable_Code_SetExtra(PyObject *code, Py_ssize_t index, void *extra)

This is Unstable API. It may change without warning in minor releases.

Set the extra data stored under the given index to *extra*. Return 0 on success. Set an exception and return -1 on failure.

Added in version 3.6: as `_PyCode_SetExtra`

Changed in version 3.12: Renamed to `PyUnstable_Code_SetExtra`. The old private name is deprecated, but will be available until the API changes.

File Objects

These APIs are a minimal emulation of the Python 2 C API for built-in file objects, which used to rely on the buffered I/O (FILE*) support from the C standard library. In Python 3, files and streams use the new `io` module, which defines several layers over the low-level unbuffered I/O of the operating system. The functions described below are convenience C wrappers over these new APIs, and meant mostly for internal error reporting in the interpreter; third-party code is advised to access the `io` APIs instead.

PyObject *PyFile_FromFd(int fd, const char *name, const char *mode, int buffering, const char *encoding, const char *errors, const char *newline, int closefd)

Return value: New reference. Part of the Stable ABI. Create a Python file object from the file descriptor of an already opened file *fd*. The arguments *name*, *encoding*, *errors* and *newline* can be `NULL` to use the defaults; *buffering* can be *-1* to use the default. *name* is ignored and kept for backward compatibility. Return `NULL` on failure. For a more comprehensive description of the arguments, please refer to the `io.open()` function documentation.

Warning

Since Python streams have their own buffering layer, mixing them with OS-level file descriptors can produce various issues (such as unexpected ordering of data).

Changed in version 3.2: Ignore *name* attribute.

int PyObject_AsFileDescriptor(PyObject *p)

Part of the Stable ABI.

Return the file descriptor associated with *p* as an int. If the object is an integer, its value is returned. If not, the object’s **`fileno()`** method is called if it exists; the method must return an integer, which is returned as the file descriptor value. Sets an exception and returns `-1` on failure.

PyObject *PyFile_GetLine(PyObject *p, int n)

Return value: New reference. Part of the Stable ABI.

Equivalent to `p.readline([n])`, this function reads one line from the object *p*. *p* may be a file object or any object with a **`readline()`** method. If *n* is `0`, exactly one line is read, regardless of the length of the line. If *n* is greater than `0`, no more than *n* bytes will be read from the file; a partial line can be returned. In both cases, an empty string is returned if the end of the file is reached immediately. If *n* is less than `0`, however, one line is read regardless of length, but **`EOFError`** is raised if the end of the file is reached immediately.

int PyFile_SetOpenCodeHook(Py_OpenCodeHookFunction handler)

Overrides the normal behavior of **`io.open_code()`** to pass its parameter through the provided handler.

The *handler* is a function of type:

typedef PyObject *(*Py_OpenCodeHookFunction)(PyObject*, void*)

Equivalent of PyObject *(*)(PyObject *path, void *userData), where *path* is guaranteed to be **`PyUnicodeObject`**.

The *userData* pointer is passed into the hook function. Since hook functions may be called from different runtimes, this pointer should not refer directly to Python state.

As this hook is intentionally used during import, avoid importing new modules during its execution unless they are known to be frozen or available in `sys.modules`.

Once a hook has been set, it cannot be removed or replaced, and later calls to **`PyFile_SetOpenCodeHook()`** will fail. On failure, the function returns -1 and sets an exception if the interpreter has been initialized.

This function is safe to call before **`Py_Initialize()`**.

Raises an auditing event `setopencodehook` with no arguments.

Added in version 3.8.

PyObject *PyFile_OpenCodeObject(PyObject *path)

Open *path* with the mode `'rb'`. *path* must be a Python **`str`** object. The behavior of this function may be overridden by **`PyFile_SetOpenCodeHook()`** to allow for some preprocessing of the text.

This is analogous to **`io.open_code()`** in Python.

On success, this function returns a strong reference to a Python file object. On failure, this function returns `NULL` with an exception set.

Added in version 3.8.

PyObject *PyFile_OpenCode(const char *path)

Similar to **`PyFile_OpenCodeObject()`**, but *path* is a UTF-8 encoded const char*.

Added in version 3.8.

int PyFile_WriteObject(PyObject *obj, PyObject *p, int flags)

Part of the Stable ABI.

Write object *obj* to file object *p*. The only supported flag for *flags* is **`Py_PRINT_RAW`**; if given, the **`str()`** of the object is written instead of the **`repr()`**. Return `0` on success or `-1` on failure; the appropriate exception will be set.

int PyFile_WriteString(const char *s, PyObject *p)

Part of the Stable ABI.

Write string *s* to file object *p*. Return `0` on success or `-1` on failure; the appropriate exception will be set.

Deprecated API

These are soft deprecated APIs that were included in Python's C API by mistake. They are documented solely for completeness; use other `PyFile*` APIs instead.

PyObject *PyFile_NewStdPrinter(int fd)

Use **`PyFile_FromFd()`** with defaults (`fd, NULL, "w", -1, NULL, NULL, NULL, 0`) instead.

PyTypeObject PyStdPrinter_Type

Type of file-like objects used internally at Python startup when **`io`** is not yet available. Use Python **`open()`** or **`PyFile_FromFd()`** to create file objects instead.

Module Objects

PyTypeObject PyModule_Type

Part of the Stable ABI.

This instance of `PyTypeObject` represents the Python module type. This is exposed to Python programs as `types.ModuleType`.

int PyModule_Check(PyObject *p)

Return true if *p* is a module object, or a subtype of a module object. This function always succeeds.

int PyModule_CheckExact(PyObject *p)

Return true if *p* is a module object, but not a subtype of `PyModule_Type`. This function always succeeds.

PyObject *PyModule_NewObject(PyObject *name)

Return value: New reference. Part of the Stable ABI since version 3.7.

Return a new module object with `module.__name__` set to *name*. The module's `__name__`, `__doc__`, `__package__` and `__loader__` attributes are filled in (all but `__name__` are set to `None`). The caller is responsible for setting a `__file__` attribute.

Return `NULL` with an exception set on error.

Added in version 3.3.

Changed in version 3.4: `__package__` and `__loader__` are now set to `None`.

PyObject *PyModule_New(const char *name)

Return value: New reference. Part of the Stable ABI.
Similar to `PyModule_NewObject()`, but the name is a UTF-8 encoded string instead of a Unicode object.

PyObject *PyModule_GetDict(PyObject *module)

Return value: Borrowed reference. Part of the Stable ABI.
Return the dictionary object that implements *module*'s namespace; this object is the same as the `__dict__` attribute of the module object. If *module* is not a module object (or a subtype of a module object), `SystemError` is raised and `NULL` is returned.

It is recommended extensions use other `PyModule_*` and `PyObject_*` functions rather than directly manipulate a module's `__dict__`.

The returned reference is borrowed from the module; it is valid until the module is destroyed.

PyObject *PyModule_GetNameObject(PyObject *module)

Return value: New reference. Part of the Stable ABI since version 3.7.
Return *module*'s `__name__` value. If the module does not provide one, or if it is not a string, `SystemError` is raised and `NULL` is returned.

Added in version 3.3.

const char *PyModule_GetName(PyObject *module)

Part of the Stable ABI.
Similar to `PyModule_GetNameObject()` but return the name encoded to `'utf-8'`.

The returned buffer is only valid until the module is renamed or destroyed. Note that Python code may rename a module by setting its `__name__` attribute.

void *PyModule_GetState(PyObject *module)

Part of the Stable ABI.
Return the “state” of the module, that is, a pointer to the block of memory allocated at module creation time, or `NULL`. See **`PyModuleDef.m_size`**.

PyModuleDef *PyModule_GetDef(PyObject *module)

Part of the Stable ABI.
Return a pointer to the **`PyModuleDef`** struct from which the module was created, or `NULL` if the module wasn’t created from a definition.

On error, return `NULL` with an exception set. Use **`PyErr_Occurred()`** to tell this case apart from a missing **`PyModuleDef`**.

PyObject *PyModule_GetFilenameObject(PyObject *module)

Return value: New reference. Part of the Stable ABI.
Return the name of the file from which *module* was loaded using *module*’s **`__file__`** attribute. If this is not defined, or if it is not a string, raise **`SystemError`** and return `NULL`; otherwise return a reference to a Unicode object.

Added in version 3.2.

const char *PyModule_GetFilename(PyObject *module)

Part of the Stable ABI.
Similar to **`PyModule_GetFilenameObject()`** but return the filename encoded to ‘utf-8’.

The returned buffer is only valid until the module’s **`__file__`** attribute is reassigned or the module is destroyed.

Deprecated since version 3.2: **`PyModule_GetFilename()`** raises **`UnicodeEncodeError`** on unencodable filenames, use

`PyModule_GetFilenameObject()` instead.

Module definitions

The functions in the previous section work on any module object, including modules imported from Python code.

Modules defined using the C API typically use a *module definition*, `PyModuleDef` – a statically allocated, constant “description” of how a module should be created.

The definition is usually used to define an extension’s “main” module object (see Defining extension modules for details). It is also used to create extension modules dynamically.

Unlike `PyModule_New()`, the definition allows management of *module state* – a piece of memory that is allocated and cleared together with the module object. Unlike the module’s Python attributes, Python code cannot replace or delete data stored in module state.

type PyModuleDef

Part of the Stable ABI (including all members).
The module definition struct, which holds all information needed to create a module object. This structure must be statically allocated (or be otherwise guaranteed to be valid while any modules created from it exist). Usually, there is only one variable of this type for each extension module.

PyModuleDef_Base m_base

Always initialize this member to **`PyModuleDef_HEAD_INIT`**.

const char *m_name

Name for the new module.

const char *m_doc

Docstring for the module; usually a docstring variable created with **`PyDoc_STRVAR`** is used.

Py_ssize_t m_size

Module state may be kept in a per-module memory area that can be retrieved with **`PyModule_GetState()`**, rather than in static globals. This makes modules safe for use in multiple sub-interpreters.

This memory area is allocated based on *m_size* on module creation, and freed when the module object is deallocated, after the **`m_free`** function has been called, if present.

Setting it to a non-negative value means that the module can be re-initialized and specifies the additional amount of memory it requires for its state.

Setting `m_size` to `-1` means that the module does not support sub-interpreters, because it has global state. Negative `m_size` is only allowed when using legacy single-phase initialization or when creating modules dynamically.

See **PEP 3121** [https://peps.python.org/pep-3121/] for more details.

PyMethodDef *m_methods

A pointer to a table of module-level functions, described by **`PyMethodDef`** values. Can be `NULL` if no functions are present.

PyModuleDef_Slot *m_slots

An array of slot definitions for multi-phase initialization, terminated by a `{0, NULL}` entry. When using legacy single-phase initialization, *m_slots* must be `NULL`.

Changed in version 3.5: Prior to version 3.5, this member was always set to `NULL`, and was defined as:

inquiry m_reload

traverseproc m_traverse

A traversal function to call during GC traversal of the module object, or `NULL` if not needed.

This function is not called if the module state was requested but is not allocated yet. This is the case immediately after the module is created and before the module is executed (**`Py_mod_exec`** function). More precisely, this function is not called if **`m_size`** is greater than 0 and the module state (as returned by **`PyModule_GetState()`**) is `NULL`.

Changed in version 3.9: No longer called before the module state is allocated.

inquiry m_clear

A clear function to call during GC clearing of the module object, or `NULL` if not needed.

This function is not called if the module state was requested but is not allocated yet. This is the case immediately after the module is created and before the module is executed (**`Py_mod_exec`** function). More precisely, this function is not called if **`m_size`** is greater than 0 and the module state (as returned by **`PyModule_GetState()`**) is `NULL`.

Like `PyTypeObject.tp_clear`, this function is not *always* called before a module is deallocated. For example, when reference counting is enough to determine that an object is no longer used, the cyclic garbage collector is not involved and `m_free` is called directly.

Changed in version 3.9: No longer called before the module state is allocated.

freefunc m_free

A function to call during deallocation of the module object, or `NULL` if not needed.

This function is not called if the module state was requested but is not allocated yet. This is the case immediately after the module is created and before the module is executed (`Py_mod_exec` function). More precisely, this function is not called if `m_size` is greater than 0 and the module state (as returned by `PyModule_GetState()`) is `NULL`.

Changed in version 3.9: No longer called before the module state is allocated.

Module slots

type PyModuleDef_Slot

Part of the Stable ABI (including all members) since version 3.5.

int slot

A slot ID, chosen from the available values explained below.

void *value

Value of the slot, whose meaning depends on the slot ID.

Added in version 3.5.

The available slot types are:

Py_mod_create

Part of the Stable ABI since version 3.5.

Specifies a function that is called to create the module object itself. The *value* pointer of this slot must point to a function of the signature:

PyObject *create_module(PyObject *spec, PyModuleDef *def)

The function receives a **`ModuleSpec`** instance, as defined in **PEP 451** [https://peps.python.org/pep-0451/], and the module definition. It should return a new module object, or set an error and return `NULL`.

This function should be kept minimal. In particular, it should not call arbitrary Python code, as trying to import the same module again may result in an infinite loop.

Multiple `Py_mod_create` slots may not be specified in one module definition.

If `Py_mod_create` is not specified, the import machinery will create a normal module object using **`PyModule_New()`**. The name is taken from *spec*, not the definition, to allow extension modules to dynamically adjust to their place in the module hierarchy and be imported under different names through symlinks, all while sharing a single module definition.

There is no requirement for the returned object to be an instance of **`PyModule_Type`**. Any type can be used, as long as it supports setting and getting import-related attributes. However, only `PyModule_Type` instances may be returned if the `PyModuleDef` has non-`NULL` `m_traverse`, `m_clear`, `m_free`; non-zero `m_size`; or slots other than `Py_mod_create`.

Added in version 3.5.

Py_mod_exec

Part of the Stable ABI since version 3.5.

Specifies a function that is called to *execute* the module. This is equivalent to executing the code of a Python module: typically, this function adds classes and constants to the module. The signature of the function is:

int exec_module(PyObject *module)

If multiple `Py_mod_exec` slots are specified, they are processed in the order they appear in the *m_slots* array.

Added in version 3.5.

Py_mod_multiple_interpreters

Part of the Stable ABI since version 3.12.

Specifies one of the following values:

Py_MOD_MULTIPLE_INTERPRETERS_NOT_SUPPORTED

The module does not support being imported in subinterpreters.

Py_MOD_MULTIPLE_INTERPRETERS_SUPPORTED

The module supports being imported in subinterpreters, but only when they share the main interpreter’s GIL. (See Isolating Extension Modules.)

Py_MOD_PER_INTERPRETER_GIL_SUPPORTED

The module supports being imported in subinterpreters, even when they have their own GIL. (See Isolating Extension Modules.)

This slot determines whether or not importing this module in a subinterpreter will fail.

Multiple `Py_mod_multiple_interpreters` slots may not be specified in one module definition.

If `Py_mod_multiple_interpreters` is not specified, the import machinery defaults to `Py_MOD_MULTIPLE_INTERPRETERS_SUPPORTED`.

Added in version 3.12.

Py_mod_gil

Part of the Stable ABI since version 3.13.
Specifies one of the following values:

Py_MOD_GIL_USED

The module depends on the presence of the global interpreter lock (GIL), and may access global state without synchronization.

Py_MOD_GIL_NOT_USED

The module is safe to run without an active GIL.

This slot is ignored by Python builds not configured with **`--disable-gil`**. Otherwise, it determines whether or not importing this module will cause the GIL to be automatically enabled. See Free-threaded CPython for more detail.

Multiple `Py_mod_gil` slots may not be specified in one module definition.

If `Py_mod_gil` is not specified, the import machinery defaults to `Py_MOD_GIL_USED`.

Added in version 3.13.

Creating extension modules dynamically

The following functions may be used to create a module outside of an extension's initialization function. They are also used in single-phase initialization.

PyObject *PyModule_Create(PyModuleDef *def)

Return value: New reference.
Create a new module object, given the definition in *def*. This is a macro that calls **`PyModule_Create2()`** with *module_api_version* set to **`PYTHON_API_VERSION`**, or to **`PYTHON_ABI_VERSION`** if using the limited API.

PyObject *PyModule_Create2(PyModuleDef *def, int module_api_version)

Return value: New reference. Part of the Stable ABI.
Create a new module object, given the definition in *def*, assuming the API version *module_api_version*. If that version does not match the version of the running interpreter, a **`RuntimeWarning`** is emitted.

Return `NULL` with an exception set on error.

This function does not support slots. The **`m_slots`** member of *def* must be `NULL`.

Note

Most uses of this function should be using **`PyModule_Create()`** instead; only use this if you are sure you need it.

PyObject *PyModule_FromDefAndSpec(PyModuleDef *def, PyObject *spec)

Return value: New reference.
This macro calls `PyModule_FromDefAndSpec2()` with *module_api_version* set to `PYTHON_API_VERSION`, or to `PYTHON_ABI_VERSION` if using the limited API.

Added in version 3.5.

PyObject *PyModule_FromDefAndSpec2(PyModuleDef *def, PyObject *spec, int module_api_version)

Return value: New reference. Part of the Stable ABI since version 3.7.
Create a new module object, given the definition in *def* and the ModuleSpec *spec*, assuming the API version *module_api_version*. If that version does not match the version of the running interpreter, a `RuntimeWarning` is emitted.

Return `NULL` with an exception set on error.

Note that this does not process execution slots (`Py_mod_exec`). Both `PyModule_FromDefAndSpec` and `PyModule_ExecDef` must be called to fully initialize a module.

Note

Most uses of this function should be using `PyModule_FromDefAndSpec()` instead; only use this if you are sure you need it.

Added in version 3.5.

int PyModule_ExecDef(PyObject *module, PyModuleDef *def)

Part of the Stable ABI since version 3.7.
Process any execution slots (`Py_mod_exec`) given in *def*.

Added in version 3.5.

PYTHON_API_VERSION

The C API version. Defined for backwards compatibility.

Currently, this constant is not updated in new Python versions, and is not useful for versioning. This may change in the future.

PYTHON_ABI_VERSION

Defined as `3` for backwards compatibility.

Currently, this constant is not updated in new Python versions, and is not useful for versioning. This may change in the future.

Support functions

The following functions are provided to help initialize a module state. They are intended for a module's execution slots (`Py_mod_exec`), the initialization function for legacy single-phase initialization, or code that creates modules dynamically.

int PyModule_AddObjectRef(PyObject *module, const char *name, PyObject *value)

Part of the Stable ABI since version 3.10.
Add an object to *module* as *name*. This is a convenience function which can be used from the module's initialization function.

On success, return `0`. On error, raise an exception and return `-1`.

Example usage:

```
static int
add_spam(PyObject *module, int value)
{
    PyObject *obj = PyLong_FromLong(value);
    if (obj == NULL) {
        return -1;
    }
    int res = PyModule_AddObjectRef(module, "spam", obj);
    Py_DECREF(obj);
    return res;
 }
```

To be convenient, the function accepts `NULL` *value* with an exception set. In this case, return `-1` and just leave the raised exception unchanged.

The example can also be written without checking explicitly if *obj* is `NULL`:

```
static int
add_spam(PyObject *module, int value)
{
    PyObject *obj = PyLong_FromLong(value);
    int res = PyModule_AddObjectRef(module, "spam", obj);
    Py_XDECREF(obj);
    return res;
 }
```

Note that `Py_XDECREF()` should be used instead of `Py_DECREF()` in this case, since *obj* can be `NULL`.

The number of different *name* strings passed to this function should be kept small, usually by only using statically allocated strings as *name*. For names that aren’t known at compile time, prefer calling

`PyUnicode_FromString()` and `PyObject_SetAttr()` directly. For more details, see `PyUnicode_InternFromString()`, which may be used internally to create a key object.

Added in version 3.10.

int PyModule_Add(PyObject *module, const char *name, PyObject *value)

Part of the Stable ABI since version 3.13.
Similar to `PyModule_AddObjectRef()`, but “steals” a reference to *value*. It can be called with a result of function that returns a new reference without bothering to check its result or even saving it to a variable.

Example usage:

```
if (PyModule_Add(module, "spam", PyBytes_FromString(value))
< 0) {
    goto error;
}
```

Added in version 3.13.

int PyModule_AddObject(PyObject *module, const char *name, PyObject *value)

Part of the Stable ABI.
Similar to `PyModule_AddObjectRef()`, but steals a reference to *value* on success (if it returns `0`).

The new `PyModule_Add()` or `PyModule_AddObjectRef()` functions are recommended, since it is easy to introduce reference leaks by misusing the `PyModule_AddObject()` function.

Note

Unlike other functions that steal references, `PyModule_AddObject()` only releases the reference to *value*

on success.

This means that its return value must be checked, and calling code must **`Py_XDECREF()`** *value* manually on error.

Example usage:

```
PyObject *obj = PyBytes_FromString(value);
if (PyModule_AddObject(module, "spam", obj) < 0) {
    // If 'obj' is not NULL and PyModule_AddObject() failed,
    // 'obj' strong reference must be deleted with
Py_XDECREF().
    // If 'obj' is NULL, Py_XDECREF() does nothing.
    Py_XDECREF(obj);
    goto error;
}
// PyModule_AddObject() stole a reference to obj:
// Py_XDECREF(obj) is not needed here.
```

Deprecated since version 3.13: **`PyModule_AddObject()`** is soft deprecated.

int PyModule_AddIntConstant(PyObject *module, const char *name, long value)

Part of the Stable ABI.
Add an integer constant to *module* as *name*. This convenience function can be used from the module’s initialization function. Return `-1` with an exception set on error, `0` on success.

This is a convenience function that calls **`PyLong_FromLong()`** and **`PyModule_AddObjectRef()`**; see their documentation for details.

int PyModule_AddStringConstant(PyObject *module, const char *name, const char *value)

Part of the Stable ABI.

Add a string constant to *module* as *name*. This convenience function can be used from the module's initialization function. The string *value* must be `NULL`-terminated. Return `-1` with an exception set on error, `0` on success.

This is a convenience function that calls **`PyUnicode_InternFromString()`** and **`PyModule_AddObjectRef()`**; see their documentation for details.

PyModule_AddIntMacro(module, macro)

Add an int constant to *module*. The name and the value are taken from *macro*. For example `PyModule_AddIntMacro(module, AF_INET)` adds the int constant *AF_INET* with the value of *AF_INET* to *module*. Return `-1` with an exception set on error, `0` on success.

PyModule_AddStringMacro(module, macro)

Add a string constant to *module*.

int PyModule_AddType(PyObject *module, PyTypeObject *type)

Part of the Stable ABI since version 3.10.
Add a type object to *module*. The type object is finalized by calling internally **`PyType_Ready()`**. The name of the type object is taken from the last component of **`tp_name`** after dot. Return `-1` with an exception set on error, `0` on success.

Added in version 3.9.

int PyModule_AddFunctions(PyObject *module, PyMethodDef *functions)

Part of the Stable ABI since version 3.7.

Add the functions from the `NULL` terminated *functions* array to *module*. Refer to the **`PyMethodDef`** documentation for details on individual entries (due to the lack of a shared module namespace, module level “functions” implemented in C typically receive the module as their first parameter, making them similar to instance methods on Python classes).

This function is called automatically when creating a module from `PyModuleDef` (such as when using Multi-phase initialization, `PyModule_Create`, or `PyModule_FromDefAndSpec`). Some module authors may prefer defining functions in multiple **`PyMethodDef`** arrays; in that case they should call this function directly.

The *functions* array must be statically allocated (or otherwise guaranteed to outlive the module object).

Added in version 3.5.

int PyModule_SetDocString(PyObject *module, const char *docstring)

Part of the Stable ABI since version 3.7.
Set the docstring for *module* to *docstring*. This function is called automatically when creating a module from `PyModuleDef` (such as when using Multi-phase initialization, `PyModule_Create`, or `PyModule_FromDefAndSpec`).

Added in version 3.5.

int PyUnstable_Module_SetGIL(PyObject *module, void *gil)

This is Unstable API. It may change without warning in minor releases.

Indicate that *module* does or does not support running without the global interpreter lock (GIL), using one of the values from **`Py_mod_gil`**. It must be called during *module*’s initialization function when using Legacy single-phase initialization. If this function is not called during module initialization, the import machinery assumes the module does not support running without the GIL. This function is only available in Python builds configured with **`--disable-gil`**. Return `-1` with an exception set on error, `0` on success.

Added in version 3.13.

Module lookup (single-phase initialization)

The legacy single-phase initialization initialization scheme creates singleton modules that can be looked up in the context of the current interpreter. This allows the module object to be retrieved later with only a reference to the module definition.

These functions will not work on modules created using multi-phase initialization, since multiple such modules can be created from a single definition.

PyObject *PyState_FindModule(PyModuleDef *def)

Return value: Borrowed reference. Part of the Stable ABI. Returns the module object that was created from *def* for the current interpreter. This method requires that the module object has been attached to the interpreter state with **`PyState_AddModule()`** beforehand. In case the corresponding module object is not found or has not been attached to the interpreter state yet, it returns `NULL`.

int PyState_AddModule(PyObject *module, PyModuleDef *def)

Part of the Stable ABI since version 3.3.
Attaches the module object passed to the function to the interpreter state. This allows the module object to be accessible via **`PyState_FindModule()`**.

Only effective on modules created using single-phase initialization.

Python calls `PyState_AddModule` automatically after importing a module that uses single-phase initialization, so it is unnecessary (but harmless) to call it from module initialization code. An explicit call is needed only if the module's own init code subsequently calls `PyState_FindModule`. The function is mainly intended for implementing alternative import mechanisms (either by calling it directly, or by referring to its implementation for details of the required state updates).

If a module was attached previously using the same *def*, it is replaced by the new *module*.

The caller must have an attached thread state.

Return `-1` with an exception set on error, `0` on success.

Added in version 3.3.

int PyState_RemoveModule(PyModuleDef *def)

Part of the Stable ABI since version 3.3.
Removes the module object created from *def* from the interpreter state. Return `-1` with an exception set on error, `0` on success.

The caller must have an attached thread state.

Added in version 3.3.

Iterator Objects

Python provides two general-purpose iterator objects. The first, a sequence iterator, works with an arbitrary sequence supporting the **`__getitem__()`** method. The second works with a callable object and a sentinel value, calling the callable for each item in the sequence, and ending the iteration when the sentinel value is returned.

PyTypeObject PySeqIter_Type

Part of the Stable ABI.
Type object for iterator objects returned by **`PySeqIter_New()`** and the one-argument form of the **`iter()`** built-in function for built-in sequence types.

int PySeqIter_Check(PyObject *op)

Return true if the type of *op* is **`PySeqIter_Type`**. This function always succeeds.

PyObject *PySeqIter_New(PyObject *seq)

Return value: New reference. Part of the Stable ABI.
Return an iterator that works with a general sequence object, *seq*. The iteration ends when the sequence raises **`IndexError`** for the subscripting operation.

PyTypeObject PyCallIter_Type

Part of the Stable ABI.
Type object for iterator objects returned by **`PyCallIter_New()`** and the two-argument form of the **`iter()`** built-in function.

int PyCallIter_Check(PyObject *op)

Return true if the type of *op* is `PyCallIter_Type`. This function always succeeds.

PyObject *PyCallIter_New(PyObject *callable, PyObject *sentinel)

Return value: New reference. Part of the Stable ABI.
Return a new iterator. The first parameter, *callable*, can be any Python callable object that can be called with no parameters; each call to it should return the next item in the iteration. When *callable* returns a value equal to *sentinel*, the iteration will be terminated.

Range Objects

PyTypeObject PyRange_Type

Part of the Stable ABI.
The type object for `range` objects.

int PyRange_Check(PyObject *o)

Return true if the object *o* is an instance of a `range` object. This function always succeeds.

Builtin Iterator Types

These are built-in iteration types that are included in Python’s C API, but provide no additional functions. They are here for completeness.

C type	**Python type**
PyTypeObject PyEnum_Type *Part of the Stable ABI.*	`enumerate`

C type	Python type
PyTypeObject PyFilter_Type *Part of the Stable ABI.*	`filter`
PyTypeObject PyMap_Type *Part of the Stable ABI.*	`map`
PyTypeObject PyReversed_Type *Part of the Stable ABI.*	`reversed`
PyTypeObject PyZip_Type *Part of the Stable ABI.*	`zip`

Other Iterator Objects

PyTypeObject PyByteArrayIter_Type
Part of the Stable ABI.

PyTypeObject PyBytesIter_Type
Part of the Stable ABI.

PyTypeObject PyListIter_Type
Part of the Stable ABI.

PyTypeObject PyListRevIter_Type
Part of the Stable ABI.

PyTypeObject PySetIter_Type
Part of the Stable ABI.

PyTypeObject PyTupleIter_Type

Part of the Stable ABI.

PyTypeObject PyRangeIter_Type

Part of the Stable ABI.

PyTypeObject PyLongRangeIter_Type

Part of the Stable ABI.

PyTypeObject PyDictIterKey_Type

Part of the Stable ABI.

PyTypeObject PyDictRevIterKey_Type

Part of the Stable ABI since version 3.8.

PyTypeObject PyDictIterValue_Type

Part of the Stable ABI.

PyTypeObject PyDictRevIterValue_Type

Part of the Stable ABI since version 3.8.

PyTypeObject PyDictIterItem_Type

Part of the Stable ABI.

PyTypeObject PyDictRevIterItem_Type

Part of the Stable ABI since version 3.8.

PyTypeObject PyODictIter_Type

Type objects for iterators of various built-in objects.

Do not create instances of these directly; prefer calling `PyObject_GetIter()` instead.

Note that there is no guarantee that a given built-in type uses a given iterator type. For example, iterating over `range` will use one of two iterator types depending on the size of the range. Other types may start using a similar scheme in the future, without warning.

Descriptor Objects

“Descriptors” are objects that describe some attribute of an object. They are found in the dictionary of type objects.

PyTypeObject PyProperty_Type

Part of the Stable ABI.

The type object for the built-in descriptor types.

PyObject *PyDescr_NewGetSet(PyTypeObject *type, struct PyGetSetDef *getset)

Return value: New reference. Part of the Stable ABI.

PyObject *PyDescr_NewMember(PyTypeObject *type, struct PyMemberDef *meth)

Return value: New reference. Part of the Stable ABI.

PyTypeObject PyMemberDescr_Type

Part of the Stable ABI.

The type object for member descriptor objects created from `PyMemberDef` structures. These descriptors expose fields of a C struct as attributes on a type, and correspond to `types.MemberDescriptorType` objects in Python.

PyTypeObject PyGetSetDescr_Type

Part of the Stable ABI.

The type object for get/set descriptor objects created from `PyGetSetDef` structures. These descriptors implement attributes whose value is computed by C getter and setter functions, and are used for many built-in type attributes.

PyObject *PyDescr_NewMethod(PyTypeObject *type, struct PyMethodDef *meth)

Return value: New reference. Part of the Stable ABI.

PyTypeObject PyMethodDescr_Type

Part of the Stable ABI.

The type object for method descriptor objects created from **`PyMethodDef`** structures. These descriptors expose C functions as methods on a type, and correspond to **`types.MemberDescriptorType`** objects in Python.

PyObject *PyDescr_NewWrapper(PyTypeObject *type, struct wrapperbase *wrapper, void *wrapped)

Return value: New reference.

PyTypeObject PyWrapperDescr_Type

Part of the Stable ABI.

The type object for wrapper descriptor objects created by **`PyDescr_NewWrapper()`** and **`PyWrapper_New()`**. Wrapper descriptors are used internally to expose special methods implemented via wrapper structures, and appear in Python as **`types.WrapperDescriptorType`** objects.

PyObject *PyDescr_NewClassMethod(PyTypeObject *type, PyMethodDef *method)

Return value: New reference. Part of the Stable ABI.

int PyDescr_IsData(PyObject *descr)

Return non-zero if the descriptor object *descr* describes a data attribute, or `0` if it describes a method. *descr* must be a descriptor object; there is no error checking.

PyObject *PyWrapper_New(PyObject*, PyObject*)

Return value: New reference. Part of the Stable ABI.

Built-in descriptors

PyTypeObject PySuper_Type

Part of the Stable ABI.

The type object for super objects. This is the same object as **`super`** in the Python layer.

PyTypeObject PyClassMethod_Type

The type of class method objects. This is the same object as **`classmethod`** in the Python layer.

PyTypeObject PyClassMethodDescr_Type

Part of the Stable ABI.

The type object for C-level class method descriptor objects. This is the type of the descriptors created for **`classmethod()`** defined in C extension types, and is the same object as **`classmethod`** in Python.

PyObject *PyClassMethod_New(PyObject *callable)

Create a new **`classmethod`** object wrapping *callable*. *callable* must be a callable object and must not be `NULL`.

On success, this function returns a strong reference to a new class method descriptor. On failure, this function returns `NULL` with an exception set.

PyTypeObject PyStaticMethod_Type

The type of static method objects. This is the same object as **`staticmethod`** in the Python layer.

PyObject *PyStaticMethod_New(PyObject *callable)

Create a new **`staticmethod`** object wrapping *callable*. *callable* must be a callable object and must not be `NULL`.

On success, this function returns a strong reference to a new static method descriptor. On failure, this function returns `NULL` with an exception set.

Slice Objects

PyTypeObject PySlice_Type

Part of the Stable ABI.

The type object for slice objects. This is the same as `slice` in the Python layer.

int PySlice_Check(PyObject *ob)

Return true if *ob* is a slice object; *ob* must not be `NULL`. This function always succeeds.

PyObject *PySlice_New(PyObject *start, PyObject *stop, PyObject *step)

Return value: New reference. Part of the Stable ABI.

Return a new slice object with the given values. The *start*, *stop*, and *step* parameters are used as the values of the slice object attributes of the same names. Any of the values may be `NULL`, in which case the `None` will be used for the corresponding attribute.

Return `NULL` with an exception set if the new object could not be allocated.

int PySlice_GetIndices(PyObject *slice, Py_ssize_t length, Py_ssize_t *start, Py_ssize_t *stop, Py_ssize_t *step)

Part of the Stable ABI.

Retrieve the start, stop and step indices from the slice object *slice*, assuming a sequence of length *length*. Treats indices greater than *length* as errors.

Returns `0` on success and `-1` on error with no exception set (unless one of the indices was not `None` and failed to be converted to an integer, in which case `-1` is returned with an exception set).

You probably do not want to use this function.

Changed in version 3.2: The parameter type for the *slice* parameter was `PySliceObject*` before.

int PySlice_GetIndicesEx(PyObject *slice, Py_ssize_t length, Py_ssize_t *start, Py_ssize_t *stop, Py_ssize_t *step, Py_ssize_t *slicelength)

Part of the Stable ABI.

Usable replacement for **`PySlice_GetIndices()`**. Retrieve the start, stop, and step indices from the slice object *slice* assuming a sequence of length *length*, and store the length of the slice in *slicelength*. Out of bounds indices are clipped in a manner consistent with the handling of normal slices.

Return `0` on success and `-1` on error with an exception set.

Note

This function is considered not safe for resizable sequences. Its invocation should be replaced by a combination of **`PySlice_Unpack()`** and **`PySlice_AdjustIndices()`** where

```
if (PySlice_GetIndicesEx(slice, length, &start, &stop,
&step, &slicelength) < 0) {
    // return error
}
```

is replaced by

```
if (PySlice_Unpack(slice, &start, &stop, &step) < 0) {
    // return error
}
slicelength = PySlice_AdjustIndices(length, &start, &stop,
step);
```

Changed in version 3.2: The parameter type for the *slice* parameter was `PySliceObject*` before.

Changed in version 3.6.1: If `Py_LIMITED_API` is not set or set to the value between `0x03050400` and `0x03060000` (not including) or `0x03060100` or higher **`PySlice_GetIndicesEx()`** is implemented as a macro using **`PySlice_Unpack()`** and **`PySlice_AdjustIndices()`**. Arguments *start*, *stop* and *step* are evaluated more than once.

Deprecated since version 3.6.1: If `Py_LIMITED_API` is set to the value less than `0x03050400` or between `0x03060000` and `0x03060100` (not including) **`PySlice_GetIndicesEx()`** is a deprecated function.

int PySlice_Unpack(PyObject *slice, Py_ssize_t *start, Py_ssize_t *stop, Py_ssize_t *step)

Part of the Stable ABI since version 3.7.
Extract the start, stop and step data members from a slice object as C integers. Silently reduce values larger than `PY_SSIZE_T_MAX` to `PY_SSIZE_T_MAX`, silently boost the start and stop values less than `PY_SSIZE_T_MIN` to `PY_SSIZE_T_MIN`, and silently boost the step values less than `-PY_SSIZE_T_MAX` to `-PY_SSIZE_T_MAX`.

Return `-1` with an exception set on error, `0` on success.

Added in version 3.6.1.

Py_ssize_t PySlice_AdjustIndices(Py_ssize_t length, Py_ssize_t *start, Py_ssize_t *stop, Py_ssize_t step)

Part of the Stable ABI since version 3.7.

Adjust start/end slice indices assuming a sequence of the specified length. Out of bounds indices are clipped in a manner consistent with the handling of normal slices.

Return the length of the slice. Always successful. Doesn't call Python code.

Added in version 3.6.1.

Ellipsis Object

PyTypeObject PyEllipsis_Type

Part of the Stable ABI.
The type of Python **`Ellipsis`** object. Same as **`types.EllipsisType`** in the Python layer.

PyObject *Py_Ellipsis

The Python `Ellipsis` object. This object has no methods. Like **`Py_None`**, it is an immortal singleton object.

Changed in version 3.12: **`Py_Ellipsis`** is immortal.

MemoryView objects

A `memoryview` object exposes the C level buffer interface as a Python object which can then be passed around like any other object.

PyTypeObject PyMemoryView_Type

Part of the Stable ABI.
This instance of `PyTypeObject` represents the Python memoryview type. This is the same object as `memoryview` in the Python layer.

PyObject *PyMemoryView_FromObject(PyObject *obj)

Return value: New reference. Part of the Stable ABI.
Create a memoryview object from an object that provides the buffer interface. If *obj* supports writable buffer exports, the memoryview object will be read/write, otherwise it may be either read-only or read/write at the discretion of the exporter.

PyBUF_READ

Part of the Stable ABI since version 3.11.
Flag to request a readonly buffer.

PyBUF_WRITE

Part of the Stable ABI since version 3.11.
Flag to request a writable buffer.

PyObject *PyMemoryView_FromMemory(char *mem, Py_ssize_t size, int flags)

Return value: New reference. Part of the Stable ABI since version 3.7.

Create a memoryview object using *mem* as the underlying buffer. *flags* can be one of **`PyBUF_READ`** or **`PyBUF_WRITE`**.

Added in version 3.3.

PyObject *PyMemoryView_FromBuffer(const Py_buffer *view)

Return value: New reference. Part of the Stable ABI since version 3.11.
Create a memoryview object wrapping the given buffer structure *view*. For simple byte buffers, **`PyMemoryView_FromMemory()`** is the preferred function.

PyObject *PyMemoryView_GetContiguous(PyObject *obj, int buffertype, char order)

Return value: New reference. Part of the Stable ABI.
Create a memoryview object to a contiguous chunk of memory (in either 'C' or 'F'ortran *order*) from an object that defines the buffer interface. If memory is contiguous, the memoryview object points to the original memory. Otherwise, a copy is made and the memoryview points to a new bytes object.

buffertype can be one of **`PyBUF_READ`** or **`PyBUF_WRITE`**.

int PyMemoryView_Check(PyObject *obj)

Return true if the object *obj* is a memoryview object. It is not currently allowed to create subclasses of **`memoryview`**. This function always succeeds.

Py_buffer *PyMemoryView_GET_BUFFER(PyObject *mview)

Return a pointer to the memoryview's private copy of the exporter's buffer. *mview* **must** be a memoryview instance; this macro doesn’t check its type, you must do it yourself or you will risk crashes.

PyObject *PyMemoryView_GET_BASE(PyObject *mview)

Return either a pointer to the exporting object that the memoryview is based on or `NULL` if the memoryview has been created by one of the functions `PyMemoryView_FromMemory()` or `PyMemoryView_FromBuffer()`. *mview* **must** be a memoryview instance.

Pickle buffer objects

Added in version 3.8.

A **`pickle.PickleBuffer`** object wraps a buffer-providing object for out-of-band data transfer with the **`pickle`** module.

PyTypeObject PyPickleBuffer_Type

This instance of **`PyTypeObject`** represents the Python pickle buffer type. This is the same object as **`pickle.PickleBuffer`** in the Python layer.

int PyPickleBuffer_Check(PyObject *op)

Return true if *op* is a pickle buffer instance. This function always succeeds.

PyObject *PyPickleBuffer_FromObject(PyObject *obj)

Create a pickle buffer from the object *obj*.

This function will fail if *obj* doesn’t support the buffer protocol.

On success, return a new pickle buffer instance. On failure, set an exception and return `NULL`.

Analogous to calling **`pickle.PickleBuffer`** with *obj* in Python.

const Py_buffer *PyPickleBuffer_GetBuffer(PyObject *picklebuf)

Get a pointer to the underlying **`Py_buffer`** that the pickle buffer wraps.

The returned pointer is valid as long as *picklebuf* is alive and has not been released. The caller must not modify or free the returned **`Py_buffer`**. If the pickle buffer has been released, raise **`ValueError`**.

On success, return a pointer to the buffer view. On failure, set an exception and return `NULL`.

int PyPickleBuffer_Release(PyObject *picklebuf)

Release the underlying buffer held by the pickle buffer.

Return `0` on success. On failure, set an exception and return `-1`.

Analogous to calling **`pickle.PickleBuffer.release()`** in Python.

Weak Reference Objects

Python supports *weak references* as first-class objects. There are two specific object types which directly implement weak references. The first is a simple reference object, and the second acts as a proxy for the original object as much as it can.

int PyWeakref_Check(PyObject *ob)

Return non-zero if *ob* is either a reference or proxy object. This function always succeeds.

int PyWeakref_CheckRef(PyObject *ob)

Return non-zero if *ob* is a reference object or a subclass of the reference type. This function always succeeds.

int PyWeakref_CheckRefExact(PyObject *ob)

Return non-zero if *ob* is a reference object, but not a subclass of the reference type. This function always succeeds.

int PyWeakref_CheckProxy(PyObject *ob)

Return non-zero if *ob* is a proxy object. This function always succeeds.

PyObject *PyWeakref_NewRef(PyObject *ob, PyObject *callback)

Return value: New reference. Part of the Stable ABI. Return a weak reference object for the object *ob*. This will always return a new reference, but is not guaranteed to create a new object; an existing reference object may be returned. The second parameter, *callback*, can be a callable object that receives notification when *ob* is garbage collected; it should accept a single parameter, which will be the weak reference object itself. *callback* may also be `None` or `NULL`. If *ob* is not a weakly referenceable object, or if *callback* is not callable, `None`, or `NULL`, this will return `NULL` and raise **`TypeError`**.

See also

`PyType_SUPPORTS_WEAKREFS()` for checking if *ob* is weakly referenceable.

PyObject *PyWeakref_NewProxy(PyObject *ob, PyObject *callback)

Return value: New reference. Part of the Stable ABI.
Return a weak reference proxy object for the object *ob*. This will always return a new reference, but is not guaranteed to create a new object; an existing proxy object may be returned. The second parameter, *callback*, can be a callable object that receives notification when *ob* is garbage collected; it should accept a single parameter, which will be the weak reference object itself. *callback* may also be `None` or `NULL`. If *ob* is not a weakly referenceable object, or if *callback* is not callable, `None`, or `NULL`, this will return `NULL` and raise **`TypeError`**.

See also

`PyType_SUPPORTS_WEAKREFS()` for checking if *ob* is weakly referenceable.

int PyWeakref_GetRef(PyObject *ref, PyObject **pobj)

Part of the Stable ABI since version 3.13.
Get a strong reference to the referenced object from a weak reference, *ref*, into **pobj*.

- On success, set **pobj* to a new strong reference to the referenced object and return 1.
- If the reference is dead, set **pobj* to `NULL` and return 0.
- On error, raise an exception and return -1.

Added in version 3.13.

PyObject *PyWeakref_GetObject(PyObject *ref)

Return value: Borrowed reference. Part of the Stable ABI.
Return a borrowed reference to the referenced object from a weak reference, *ref*. If the referent is no longer live, returns `Py_None`.

Note

This function returns a borrowed reference to the referenced object. This means that you should always call **`Py_INCREF()`** on the object except when it cannot be destroyed before the last usage of the borrowed reference.

Deprecated since version 3.13, will be removed in version 3.15: Use **`PyWeakref_GetRef()`** instead.

PyObject *PyWeakref_GET_OBJECT(PyObject *ref)

Return value: Borrowed reference.
Similar to **`PyWeakref_GetObject()`**, but does no error checking.

Deprecated since version 3.13, will be removed in version 3.15: Use **`PyWeakref_GetRef()`** instead.

int PyWeakref_IsDead(PyObject *ref)

Test if the weak reference *ref* is dead. Returns 1 if the reference is dead, 0 if it is alive, and -1 with an error set if *ref* is not a weak reference object.

Added in version 3.14.

void PyObject_ClearWeakRefs(PyObject *object)

Part of the Stable ABI.

This function is called by the **`tp_dealloc`** handler to clear weak references.

This iterates through the weak references for *object* and calls callbacks for those references which have one. It returns when all callbacks have been attempted.

void PyUnstable_Object_ClearWeakRefsNoCallbacks(PyObject *object)

> *This is Unstable API. It may change without warning in minor releases.*

Clears the weakrefs for *object* without calling the callbacks.

This function is called by the **`tp_dealloc`** handler for types with finalizers (i.e., **`__del__()`**). The handler for those objects first calls **`PyObject_ClearWeakRefs()`** to clear weakrefs and call their callbacks, then the finalizer, and finally this function to clear any weakrefs that may have been created by the finalizer.

In most circumstances, it’s more appropriate to use **`PyObject_ClearWeakRefs()`** to clear weakrefs instead of this function.

Added in version 3.13.

Capsules

Refer to Providing a C API for an Extension Module for more information on using these objects.

Added in version 3.1.

type PyCapsule

This subtype of **`PyObject`** represents an opaque value, useful for C extension modules which need to pass an opaque value (as a void* pointer) through Python code to other C code. It is often used to make a C function pointer defined in one module available to other modules, so the regular import mechanism can be used to access C APIs defined in dynamically loaded modules.

PyTypeObject PyCapsule_Type

Part of the Stable ABI.
The type object corresponding to capsule objects. This is the same object as **`types.CapsuleType`** in the Python layer.

type PyCapsule_Destructor

Part of the Stable ABI.
The type of a destructor callback for a capsule. Defined as:

```
typedef void (*PyCapsule_Destructor)(PyObject *);
```

See **`PyCapsule_New()`** for the semantics of PyCapsule_Destructor callbacks.

int PyCapsule_CheckExact(PyObject *p)

Return true if its argument is a **`PyCapsule`**. This function always succeeds.

PyObject *PyCapsule_New(void *pointer, const char *name, PyCapsule_Destructor destructor)

Return value: New reference. Part of the Stable ABI.
Create a **`PyCapsule`** encapsulating the *pointer*. The *pointer* argument may not be `NULL`.

On failure, set an exception and return `NULL`.

The *name* string may either be `NULL` or a pointer to a valid C string. If non-`NULL`, this string must outlive the capsule. (Though it is permitted to free it inside the *destructor*.)

If the *destructor* argument is not `NULL`, it will be called with the capsule as its argument when it is destroyed.

If this capsule will be stored as an attribute of a module, the *name* should be specified as `modulename.attributename`. This will enable other modules to import the capsule using **`PyCapsule_Import()`**.

void *PyCapsule_GetPointer(PyObject *capsule, const char *name)

Part of the Stable ABI.
Retrieve the *pointer* stored in the capsule. On failure, set an exception and return `NULL`.

The *name* parameter must compare exactly to the name stored in the capsule. If the name stored in the capsule is `NULL`, the *name* passed in must also be `NULL`. Python uses the C function **`strcmp()`** to compare capsule names.

PyCapsule_Destructor PyCapsule_GetDestructor(PyObject *capsule)

Part of the Stable ABI.
Return the current destructor stored in the capsule. On failure, set an exception and return `NULL`.

It is legal for a capsule to have a `NULL` destructor. This makes a `NULL` return code somewhat ambiguous; use **`PyCapsule_IsValid()`** or **`PyErr_Occurred()`** to disambiguate.

void *PyCapsule_GetContext(PyObject *capsule)

Part of the Stable ABI.
Return the current context stored in the capsule. On failure, set an exception and return `NULL`.

It is legal for a capsule to have a `NULL` context. This makes a `NULL` return code somewhat ambiguous; use **`PyCapsule_IsValid()`** or **`PyErr_Occurred()`** to disambiguate.

const char *PyCapsule_GetName(PyObject *capsule)

Part of the Stable ABI.
Return the current name stored in the capsule. On failure, set an exception and return `NULL`.

It is legal for a capsule to have a `NULL` name. This makes a `NULL` return code somewhat ambiguous; use **`PyCapsule_IsValid()`** or **`PyErr_Occurred()`** to disambiguate.

void *PyCapsule_Import(const char *name, int no_block)

Part of the Stable ABI.
Import a pointer to a C object from a capsule attribute in a module. The *name* parameter should specify the full name to the attribute, as in `module.attribute`. The *name* stored in the capsule must match this string exactly.

This function splits *name* on the `.` character, and imports the first element. It then processes further elements using attribute lookups.

Return the capsule's internal *pointer* on success. On failure, set an exception and return `NULL`.

Note

If *name* points to an attribute of some submodule or subpackage, this submodule or subpackage must be previously imported using other means (for example, by using **`PyImport_ImportModule()`**) for the attribute lookups to succeed.

Changed in version 3.3: no_block has no effect anymore.

int PyCapsule_IsValid(PyObject *capsule, const char *name)

Part of the Stable ABI.

Determines whether or not *capsule* is a valid capsule. A valid capsule is non-`NULL`, passes **`PyCapsule_CheckExact()`**, has a non-`NULL` pointer stored in it, and its internal name matches the *name* parameter. (See **`PyCapsule_GetPointer()`** for information on how capsule names are compared.)

In other words, if **`PyCapsule_IsValid()`** returns a true value, calls to any of the accessors (any function starting with `PyCapsule_Get`) are guaranteed to succeed.

Return a nonzero value if the object is valid and matches the name passed in. Return `0` otherwise. This function will not fail.

int PyCapsule_SetContext(PyObject *capsule, void *context)

Part of the Stable ABI.

Set the context pointer inside *capsule* to *context*.

Return `0` on success. Return nonzero and set an exception on failure.

int PyCapsule_SetDestructor(PyObject *capsule, PyCapsule_Destructor destructor)

Part of the Stable ABI.

Set the destructor inside *capsule* to *destructor*.

Return `0` on success. Return nonzero and set an exception on failure.

int PyCapsule_SetName(PyObject *capsule, const char *name)

Part of the Stable ABI.

Set the name inside *capsule* to *name*. If non-`NULL`, the name must outlive the capsule. If the previous *name* stored in the capsule was not `NULL`, no attempt is made to free it.

Return `0` on success. Return nonzero and set an exception on failure.

int PyCapsule_SetPointer(PyObject *capsule, void *pointer)

Part of the Stable ABI.

Set the void pointer inside *capsule* to *pointer*. The pointer may not be `NULL`.

Return `0` on success. Return nonzero and set an exception on failure.

Frame Objects

type PyFrameObject

Part of the Limited API (as an opaque struct).
The C structure of the objects used to describe frame objects.

There are no public members in this structure.

Changed in version 3.11: The members of this structure were removed from the public C API. Refer to the What’s New entry for details.

The **`PyEval_GetFrame()`** and **`PyThreadState_GetFrame()`** functions can be used to get a frame object.

See also Reflection.

PyTypeObject PyFrame_Type

The type of frame objects. It is the same object as **`types.FrameType`** in the Python layer.

Changed in version 3.11: Previously, this type was only available after including `<frameobject.h>`.

PyFrameObject *PyFrame_New(PyThreadState *tstate, PyCodeObject *code, PyObject *globals, PyObject *locals)

Create a new frame object. This function returns a strong reference to the new frame object on success, and returns `NULL` with an exception set on failure.

int PyFrame_Check(PyObject *obj)

Return non-zero if *obj* is a frame object.

Changed in version 3.11: Previously, this function was only available after including `<frameobject.h>`.

PyFrameObject *PyFrame_GetBack(PyFrameObject *frame)

Return value: New reference.
Get the *frame* next outer frame.

Return a strong reference, or `NULL` if *frame* has no outer frame.

Added in version 3.9.

PyObject *PyFrame_GetBuiltins(PyFrameObject *frame)

Return value: New reference.
Get the *frame*’s **`f_builtins`** attribute.

Return a strong reference. The result cannot be `NULL`.

Added in version 3.11.

PyCodeObject *PyFrame_GetCode(PyFrameObject *frame)

Return value: New reference. Part of the Stable ABI since version 3.10.
Get the *frame* code.

Return a strong reference.

The result (frame code) cannot be `NULL`.

Added in version 3.9.

PyObject *PyFrame_GetGenerator(PyFrameObject *frame)

Return value: New reference.
Get the generator, coroutine, or async generator that owns this frame, or `NULL` if this frame is not owned by a

generator. Does not raise an exception, even if the return value is `NULL`.

Return a strong reference, or `NULL`.

Added in version 3.11.

PyObject *PyFrame_GetGlobals(PyFrameObject *frame)

Return value: New reference.
Get the *frame*’s **`f_globals`** attribute.

Return a strong reference. The result cannot be `NULL`.

Added in version 3.11.

int PyFrame_GetLasti(PyFrameObject *frame)

Get the *frame*’s **`f_lasti`** attribute.

Returns -1 if `frame.f_lasti` is `None`.

Added in version 3.11.

PyObject *PyFrame_GetVar(PyFrameObject *frame, PyObject *name)

Return value: New reference.
Get the variable *name* of *frame*.

- Return a strong reference to the variable value on success.
- Raise **`NameError`** and return `NULL` if the variable does not exist.
- Raise an exception and return `NULL` on error.

name type must be a **`str`**.

Added in version 3.12.

PyObject *PyFrame_GetVarString(PyFrameObject *frame, const char *name)

Return value: New reference.
Similar to **`PyFrame_GetVar()`**, but the variable name is a C string encoded in UTF-8.

Added in version 3.12.

PyObject *PyFrame_GetLocals(PyFrameObject *frame)

Return value: New reference.
Get the *frame*’s **`f_locals`** attribute. If the frame refers to an optimized scope, this returns a write-through proxy object that allows modifying the locals. In all other cases (classes, modules, **`exec()`**, **`eval()`**) it returns the mapping representing the frame locals directly (as described for **`locals()`**).

Return a strong reference.

Added in version 3.11.

Changed in version 3.13: As part of **PEP 667** [https://peps.python.org/pep-0667/], return an instance of **`PyFrameLocalsProxy_Type`**.

int PyFrame_GetLineNumber(PyFrameObject *frame)

Part of the Stable ABI since version 3.10.
Return the line number that *frame* is currently executing.

Frame Locals Proxies

Added in version 3.13.

The **`f_locals`** attribute on a frame object is an instance of a “frame-locals proxy”. The proxy object exposes a write-through view of the underlying locals dictionary for the frame. This ensures that the variables exposed by `f_locals` are always up to date with the live local variables in the frame itself.

See **PEP 667** [https://peps.python.org/pep-0667/] for more information.

PyTypeObject PyFrameLocalsProxy_Type

The type of frame **`locals()`** proxy objects.

int PyFrameLocalsProxy_Check(PyObject *obj)

Return non-zero if *obj* is a frame **`locals()`** proxy.

Legacy Local Variable APIs

These APIs are soft deprecated. As of Python 3.13, they do nothing. They exist solely for backwards compatibility.

void PyFrame_LocalsToFast(PyFrameObject *f, int clear)

This function is soft deprecated and does nothing.

Prior to Python 3.13, this function would copy the **`f_locals`** attribute of *f* to the internal "fast" array of local variables, allowing changes in frame objects to be visible to the interpreter. If *clear* was true, this function would process variables that were unset in the locals dictionary.

Changed in version 3.13: This function now does nothing.

void PyFrame_FastToLocals(PyFrameObject *f)

This function is soft deprecated and does nothing.

Prior to Python 3.13, this function would copy the internal "fast" array of local variables (which is used by the interpreter) to the `f_locals` attribute of *f*, allowing changes in local variables to be visible to frame objects.

Changed in version 3.13: This function now does nothing.

int PyFrame_FastToLocalsWithError(PyFrameObject *f)

This function is soft deprecated and does nothing.

Prior to Python 3.13, this function was similar to `PyFrame_FastToLocals()`, but would return `0` on success, and `-1` with an exception set on failure.

Changed in version 3.13: This function now does nothing.

See also

PEP 667 [https://peps.python.org/pep-0667/]

Internal Frames

Unless using **PEP 523** [https://peps.python.org/pep-0523/], you will not need this.

struct _PyInterpreterFrame

The interpreter's internal frame representation.

Added in version 3.11.

PyObject *PyUnstable_InterpreterFrame_GetCode(struct _PyInterpreterFrame *frame);

This is Unstable API. It may change without warning in minor releases.

Return a strong reference to the code object for the frame.

Added in version 3.12.

int PyUnstable_InterpreterFrame_GetLasti(struct _PyInterpreterFrame *frame);

This is Unstable API. It may change without warning in minor releases.

Return the byte offset into the last executed instruction.

Added in version 3.12.

int PyUnstable_InterpreterFrame_GetLine(struct _PyInterpreterFrame *frame);

This is Unstable API. It may change without warning in minor releases.

Return the currently executing line number, or -1 if there is no line number.

Added in version 3.12.

Generator Objects

Generator objects are what Python uses to implement generator iterators. They are normally created by iterating over a function that yields values, rather than explicitly calling **`PyGen_New()`** or **`PyGen_NewWithQualName()`**.

type PyGenObject

The C structure used for generator objects.

PyTypeObject PyGen_Type

The type object corresponding to generator objects.

int PyGen_Check(PyObject *ob)

Return true if *ob* is a generator object; *ob* must not be `NULL`. This function always succeeds.

int PyGen_CheckExact(PyObject *ob)

Return true if *ob*’s type is **`PyGen_Type`**; *ob* must not be `NULL`. This function always succeeds.

PyObject *PyGen_New(PyFrameObject *frame)

Return value: New reference.
Create and return a new generator object based on the *frame* object. A reference to *frame* is stolen by this function. The argument must not be `NULL`.

PyObject *PyGen_NewWithQualName(PyFrameObject *frame, PyObject *name, PyObject *qualname)

Return value: New reference.
Create and return a new generator object based on the *frame* object, with `__name__` and `__qualname__` set to *name*

and *qualname*. A reference to *frame* is stolen by this function. The *frame* argument must not be `NULL`.

PyCodeObject *PyGen_GetCode(PyGenObject *gen)

Return a new strong reference to the code object wrapped by *gen*. This function always succeeds.

Asynchronous Generator Objects

See also

PEP 525 [https://peps.python.org/pep-0525/]

PyTypeObject PyAsyncGen_Type

The type object corresponding to asynchronous generator objects. This is available as **`types.AsyncGeneratorType`** in the Python layer.

Added in version 3.6.

PyObject *PyAsyncGen_New(PyFrameObject *frame, PyObject *name, PyObject *qualname)

Create a new asynchronous generator wrapping *frame*, with `__name__` and `__qualname__` set to *name* and *qualname*. *frame* is stolen by this function and must not be `NULL`.

On success, this function returns a strong reference to the new asynchronous generator. On failure, this function returns `NULL` with an exception set.

Added in version 3.6.

int PyAsyncGen_CheckExact(PyObject *op)

Return true if *op* is an asynchronous generator object, false otherwise. This function always succeeds.

Added in version 3.6.

Deprecated API

PyAsyncGenASend_CheckExact(op)

This is a soft deprecated API that was included in Python’s C API by mistake.

It is solely here for completeness; do not use this API.

Coroutine Objects

Added in version 3.5.

Coroutine objects are what functions declared with an `async` keyword return.

type PyCoroObject

The C structure used for coroutine objects.

PyTypeObject PyCoro_Type

The type object corresponding to coroutine objects.

int PyCoro_CheckExact(PyObject *ob)

Return true if *ob*’s type is **`PyCoro_Type`**; *ob* must not be `NULL`. This function always succeeds.

PyObject *PyCoro_New(PyFrameObject *frame, PyObject *name, PyObject *qualname)

Return value: New reference.

Create and return a new coroutine object based on the *frame* object, with `__name__` and `__qualname__` set to *name* and *qualname*. A reference to *frame* is stolen by this function. The *frame* argument must not be `NULL`.

Context Variables Objects

Added in version 3.7.

Changed in version 3.7.1:

Note

In Python 3.7.1 the signatures of all context variables C APIs were **changed** to use `PyObject` pointers instead of `PyContext`, `PyContextVar`, and `PyContextToken`, e.g.:

```
// in 3.7.0:
PyContext *PyContext_New(void);

// in 3.7.1+:
PyObject *PyContext_New(void);
```

See bpo-34762 [https://bugs.python.org/issue?@action=redirect&bpo=34762] for more details.

This section details the public C API for the `contextvars` module.

type PyContext

The C structure used to represent a `contextvars.Context` object.

type PyContextVar

The C structure used to represent a `contextvars.ContextVar` object.

type PyContextToken

The C structure used to represent a `contextvars.Token` object.

PyTypeObject PyContext_Type

The type object representing the *context* type.

PyTypeObject PyContextVar_Type

The type object representing the *context variable* type.

PyTypeObject PyContextToken_Type

The type object representing the *context variable token* type.

Type-check macros:

int PyContext_CheckExact(PyObject *o)

Return true if *o* is of type `PyContext_Type`. *o* must not be `NULL`. This function always succeeds.

int PyContextVar_CheckExact(PyObject *o)

Return true if *o* is of type `PyContextVar_Type`. *o* must not be `NULL`. This function always succeeds.

int PyContextToken_CheckExact(PyObject *o)

Return true if *o* is of type `PyContextToken_Type`. *o* must not be `NULL`. This function always succeeds.

Context object management functions:

PyObject *PyContext_New(void)

Return value: New reference.
Create a new empty context object. Returns `NULL` if an error has occurred.

PyObject *PyContext_Copy(PyObject *ctx)

Return value: New reference.

Create a shallow copy of the passed *ctx* context object. Returns `NULL` if an error has occurred.

PyObject *PyContext_CopyCurrent(void)

Return value: New reference.
Create a shallow copy of the current thread context. Returns `NULL` if an error has occurred.

int PyContext_Enter(PyObject *ctx)

Set *ctx* as the current context for the current thread. Returns `0` on success, and `-1` on error.

int PyContext_Exit(PyObject *ctx)

Deactivate the *ctx* context and restore the previous context as the current context for the current thread. Returns `0` on success, and `-1` on error.

int PyContext_AddWatcher(PyContext_WatchCallback callback)

Register *callback* as a context object watcher for the current interpreter. Return an ID which may be passed to **`PyContext_ClearWatcher()`**. In case of error (e.g. no more watcher IDs available), return `-1` and set an exception.

Added in version 3.14.

int PyContext_ClearWatcher(int watcher_id)

Clear watcher identified by *watcher_id* previously returned from **`PyContext_AddWatcher()`** for the current interpreter. Return `0` on success, or `-1` and set an exception on error (e.g. if the given *watcher_id* was never registered.)

Added in version 3.14.

type PyContextEvent

Enumeration of possible context object watcher events:

- `Py_CONTEXT_SWITCHED`: The current context has switched to a different context. The object passed to the watch callback is the now-current **`contextvars.Context`** object, or None if no context is current.

Added in version 3.14.

typedef int (*PyContext_WatchCallback)(PyContextEvent event, PyObject *obj)

Context object watcher callback function. The object passed to the callback is event-specific; see **`PyContextEvent`** for details.

If the callback returns with an exception set, it must return `-1`; this exception will be printed as an unraisable exception using **`PyErr_FormatUnraisable()`**. Otherwise it should return `0`.

There may already be a pending exception set on entry to the callback. In this case, the callback should return `0` with the same exception still set. This means the callback may not call any other API that can set an exception unless it saves and clears the exception state first, and restores it before returning.

Added in version 3.14.

Context variable functions:

PyObject *PyContextVar_New(const char *name, PyObject *def)

Return value: New reference.
Create a new `ContextVar` object. The *name* parameter is used for introspection and debug purposes. The *def*

parameter specifies a default value for the context variable, or `NULL` for no default. If an error has occurred, this function returns `NULL`.

int PyContextVar_Get(PyObject *var, PyObject *default_value, PyObject **value)

Get the value of a context variable. Returns `-1` if an error has occurred during lookup, and `0` if no error occurred, whether or not a value was found.

If the context variable was found, *value* will be a pointer to it. If the context variable was *not* found, *value* will point to:

- *default_value*, if not `NULL`;
- the default value of *var*, if not `NULL`;
- `NULL`

Except for `NULL`, the function returns a new reference.

PyObject *PyContextVar_Set(PyObject *var, PyObject *value)

Return value: New reference.
Set the value of *var* to *value* in the current context. Returns a new token object for this change, or `NULL` if an error has occurred.

int PyContextVar_Reset(PyObject *var, PyObject *token)

Reset the state of the *var* context variable to that it was in before **`PyContextVar_Set()`** that returned the *token* was called. This function returns `0` on success and `-1` on error.

Objects for Type Hinting

Various built-in types for type hinting are provided. Currently, two types exist – GenericAlias and Union. Only `GenericAlias` is exposed to C.

PyObject *Py_GenericAlias(PyObject *origin, PyObject *args)

Part of the Stable ABI since version 3.9.
Create a GenericAlias object. Equivalent to calling the Python class **`types.GenericAlias`**. The *origin* and *args* arguments set the `GenericAlias`'s `__origin__` and `__args__` attributes respectively. *origin* should be a PyTypeObject*, and *args* can be a PyTupleObject* or any `PyObject*`. If *args* passed is not a tuple, a 1-tuple is automatically constructed and `__args__` is set to `(args,)`. Minimal checking is done for the arguments, so the function will succeed even if *origin* is not a type. The `GenericAlias`'s `__parameters__` attribute is constructed lazily from `__args__`. On failure, an exception is raised and `NULL` is returned.

Here's an example of how to make an extension type generic:

```
...
static PyMethodDef my_obj_methods[] = {
    // Other methods.
    ...
    {"__class_getitem__", Py_GenericAlias,
METH_O|METH_CLASS, "See PEP 585"}
    ...
}
```

See also

The data model method **`__class_getitem__()`**.

Added in version 3.9.

PyTypeObject Py_GenericAliasType

Part of the Stable ABI since version 3.9.

The C type of the object returned by **`Py_GenericAlias()`**. Equivalent to **`types.GenericAlias`** in Python.

Added in version 3.9.

Curses C API

`curses` exposes a small C interface for extension modules. Consumers must include the header file `py_curses.h` (which is not included by default by `Python.h`) and `import_curses()` must be invoked, usually as part of the module initialisation function, to populate `PyCurses_API`.

Warning

Neither the C API nor the pure Python `curses` module are compatible with subinterpreters.

import_curses()

Import the curses C API. The macro does not need a semi-colon to be called.

On success, populate the `PyCurses_API` pointer.

On failure, set `PyCurses_API` to NULL and set an exception. The caller must check if an error occurred via `PyErr_Occurred()`:

```
import_curses();  // semi-colon is optional but recommended
if (PyErr_Occurred()) { /* cleanup */ }
```

void **PyCurses_API

Dynamically allocated object containing the curses C API. This variable is only available once `import_curses` succeeds.

`PyCurses_API[0]` corresponds to `PyCursesWindow_Type`.

`PyCurses_API[1]`, `PyCurses_API[2]`, and `PyCurses_API[3]` are pointers to predicate functions of type `int (*)(void)`.

When called, these predicates return whether **`curses.setupterm()`**, **`curses.initscr()`**, and **`curses.start_color()`** have been called respectively.

See also the convenience macros **`PyCursesSetupTermCalled`**, **`PyCursesInitialised`**, and **`PyCursesInitialisedColor`**.

Note

The number of entries in this structure is subject to changes. Consider using **`PyCurses_API_pointers`** to check if new fields are available or not.

PyCurses_API_pointers

The number of accessible fields (`4`) in **`PyCurses_API`**. This number is incremented whenever new fields are added.

PyTypeObject PyCursesWindow_Type

The heap type corresponding to **`curses.window`**.

int PyCursesWindow_Check(PyObject *op)

Return true if *op* is a **`curses.window`** instance, false otherwise.

The following macros are convenience macros expanding into C statements. In particular, they can only be used as `macro;` or `macro`, but not `macro()` or `macro();`.

PyCursesSetupTermCalled

Macro checking if **`curses.setupterm()`** has been called.

The macro expansion is roughly equivalent to:

```
{
    typedef int (*predicate_t)(void);
    predicate_t was_setupterm_called =
(predicate_t)PyCurses_API[1];
    if (!was_setupterm_called()) {
        return NULL;
    }
}
```

PyCursesInitialised

Macro checking if `curses.initscr()` has been called.

The macro expansion is roughly equivalent to:

```
{
    typedef int (*predicate_t)(void);
    predicate_t was_initscr_called =
(predicate_t)PyCurses_API[2];
    if (!was_initscr_called()) {
        return NULL;
    }
}
```

PyCursesInitialisedColor

Macro checking if `curses.start_color()` has been called.

The macro expansion is roughly equivalent to:

```
{
    typedef int (*predicate_t)(void);
    predicate_t was_start_color_called =
(predicate_t)PyCurses_API[3];
    if (!was_start_color_called()) {
        return NULL;
    }
}
```

Internal data

The following objects are exposed by the C API but should be considered internal-only.

PyCurses_CAPSULE_NAME

Name of the curses capsule to pass to `PyCapsule_Import()`.

Internal usage only. Use `import_curses` instead.

DateTime Objects

Various date and time objects are supplied by the **`datetime`** module. Before using any of these functions, the header file `datetime.h` must be included in your source (note that this is not included by `Python.h`), and the macro **`PyDateTime_IMPORT`** must be invoked, usually as part of the module initialisation function. The macro puts a pointer to a C structure into a static variable, **`PyDateTimeAPI`**, that is used by the following macros.

PyDateTime_IMPORT()

Import the datetime C API.

On success, populate the **`PyDateTimeAPI`** pointer. On failure, set **`PyDateTimeAPI`** to `NULL` and set an exception. The caller must check if an error occurred via **`PyErr_Occurred()`**:

```
PyDateTime_IMPORT;
if (PyErr_Occurred()) { /* cleanup */ }
```

Warning

This is not compatible with subinterpreters.

type PyDateTime_CAPI

Structure containing the fields for the datetime C API.

The fields of this structure are private and subject to change.

Do not use this directly; prefer `PyDateTime_*` APIs instead.

PyDateTime_CAPI *PyDateTimeAPI

Dynamically allocated object containing the datetime C API.

This variable is only available once `PyDateTime_IMPORT` succeeds.

type PyDateTime_Date

This subtype of `PyObject` represents a Python date object.

type PyDateTime_DateTime

This subtype of `PyObject` represents a Python datetime object.

type PyDateTime_Time

This subtype of `PyObject` represents a Python time object.

type PyDateTime_Delta

This subtype of `PyObject` represents the difference between two datetime values.

PyTypeObject PyDateTime_DateType

This instance of `PyTypeObject` represents the Python date type; it is the same object as `datetime.date` in the Python layer.

PyTypeObject PyDateTime_DateTimeType

This instance of `PyTypeObject` represents the Python datetime type; it is the same object as `datetime.datetime` in the Python layer.

PyTypeObject PyDateTime_TimeType

This instance of `PyTypeObject` represents the Python time type; it is the same object as `datetime.time` in the Python layer.

PyTypeObject PyDateTime_DeltaType

This instance of `PyTypeObject` represents the Python type for the difference between two datetime values; it is the same object as `datetime.timedelta` in the Python layer.

PyTypeObject PyDateTime_TZInfoType

This instance of `PyTypeObject` represents the Python time zone info type; it is the same object as `datetime.tzinfo` in the Python layer.

Macro for access to the UTC singleton:

PyObject *PyDateTime_TimeZone_UTC

Returns the time zone singleton representing UTC, the same object as `datetime.timezone.utc`.

Added in version 3.7.

Type-check macros:

int PyDate_Check(PyObject *ob)

Return true if *ob* is of type `PyDateTime_DateType` or a subtype of **`PyDateTime_DateType`**. *ob* must not be `NULL`. This function always succeeds.

int PyDate_CheckExact(PyObject *ob)

Return true if *ob* is of type `PyDateTime_DateType`. *ob* must not be `NULL`. This function always succeeds.

int PyDateTime_Check(PyObject *ob)

Return true if *ob* is of type `PyDateTime_DateTimeType` or a subtype of **`PyDateTime_DateTimeType`**. *ob* must not be `NULL`. This function always succeeds.

int PyDateTime_CheckExact(PyObject *ob)

Return true if *ob* is of type `PyDateTime_DateTimeType`. *ob* must not be `NULL`. This function always succeeds.

int PyTime_Check(PyObject *ob)

Return true if *ob* is of type **`PyDateTime_TimeType`** or a subtype of **`PyDateTime_TimeType`**. *ob* must not be `NULL`. This function always succeeds.

int PyTime_CheckExact(PyObject *ob)

Return true if *ob* is of type **`PyDateTime_TimeType`**. *ob* must not be `NULL`. This function always succeeds.

int PyDelta_Check(PyObject *ob)

Return true if *ob* is of type **`PyDateTime_DeltaType`** or a subtype of **`PyDateTime_DeltaType`**. *ob* must not be `NULL`. This function always succeeds.

int PyDelta_CheckExact(PyObject *ob)

Return true if *ob* is of type **`PyDateTime_DeltaType`**. *ob* must not be `NULL`. This function always succeeds.

int PyTZInfo_Check(PyObject *ob)

Return true if *ob* is of type **`PyDateTime_TZInfoType`** or a subtype of **`PyDateTime_TZInfoType`**. *ob* must not be `NULL`. This function always succeeds.

int PyTZInfo_CheckExact(PyObject *ob)

Return true if *ob* is of type **`PyDateTime_TZInfoType`**. *ob* must not be `NULL`. This function always succeeds.

Macros to create objects:

PyObject *PyDate_FromDate(int year, int month, int day)

Return value: New reference.

Return a **`datetime.date`** object with the specified year, month and day.

PyObject *PyDateTime_FromDateAndTime(int year, int month, int day, int hour, int minute, int second, int usecond)

Return value: New reference.
Return a `datetime.datetime` object with the specified year, month, day, hour, minute, second and microsecond.

PyObject *PyDateTime_FromDateAndTimeAndFold(int year, int month, int day, int hour, int minute, int second, int usecond, int fold)

Return value: New reference.
Return a `datetime.datetime` object with the specified year, month, day, hour, minute, second, microsecond and fold.

Added in version 3.6.

PyObject *PyTime_FromTime(int hour, int minute, int second, int usecond)

Return value: New reference.
Return a `datetime.time` object with the specified hour, minute, second and microsecond.

PyObject *PyTime_FromTimeAndFold(int hour, int minute, int second, int usecond, int fold)

Return value: New reference.
Return a `datetime.time` object with the specified hour, minute, second, microsecond and fold.

Added in version 3.6.

PyObject *PyDelta_FromDSU(int days, int seconds, int useconds)

Return value: New reference.
Return a `datetime.timedelta` object representing the given number of days, seconds and microseconds. Normalization is performed so that the resulting number

of microseconds and seconds lie in the ranges documented for **`datetime.timedelta`** objects.

PyObject *PyTimeZone_FromOffset(PyObject *offset)

Return value: New reference.
Return a **`datetime.timezone`** object with an unnamed fixed offset represented by the *offset* argument.

Added in version 3.7.

PyObject *PyTimeZone_FromOffsetAndName(PyObject *offset, PyObject *name)

Return value: New reference.
Return a **`datetime.timezone`** object with a fixed offset represented by the *offset* argument and with tzname *name*.

Added in version 3.7.

Macros to extract fields from date objects. The argument must be an instance of **`PyDateTime_Date`**, including subclasses (such as **`PyDateTime_DateTime`**). The argument must not be `NULL`, and the type is not checked:

int PyDateTime_GET_YEAR(PyDateTime_Date *o)

Return the year, as a positive int.

int PyDateTime_GET_MONTH(PyDateTime_Date *o)

Return the month, as an int from 1 through 12.

int PyDateTime_GET_DAY(PyDateTime_Date *o)

Return the day, as an int from 1 through 31.

Macros to extract fields from datetime objects. The argument must be an instance of **`PyDateTime_DateTime`**, including subclasses. The argument must not be `NULL`, and the type is not checked:

int PyDateTime_DATE_GET_HOUR(PyDateTime_DateTime *o)

Return the hour, as an int from 0 through 23.

int PyDateTime_DATE_GET_MINUTE(PyDateTime_DateTime *o)

Return the minute, as an int from 0 through 59.

int PyDateTime_DATE_GET_SECOND(PyDateTime_DateTime *o)

Return the second, as an int from 0 through 59.

int PyDateTime_DATE_GET_MICROSECOND(PyDateTime_DateTime *o)

Return the microsecond, as an int from 0 through 999999.

int PyDateTime_DATE_GET_FOLD(PyDateTime_DateTime *o)

Return the fold, as an int from 0 through 1.

Added in version 3.6.

PyObject *PyDateTime_DATE_GET_TZINFO(PyDateTime_DateTime *o)

Return the tzinfo (which may be `None`).

Added in version 3.10.

Macros to extract fields from time objects. The argument must be an instance of **`PyDateTime_Time`**, including subclasses. The argument must not be `NULL`, and the type is not checked:

int PyDateTime_TIME_GET_HOUR(PyDateTime_Time *o)

Return the hour, as an int from 0 through 23.

int PyDateTime_TIME_GET_MINUTE(PyDateTime_Time *o)

Return the minute, as an int from 0 through 59.

int PyDateTime_TIME_GET_SECOND(PyDateTime_Time *o)

Return the second, as an int from 0 through 59.

int PyDateTime_TIME_GET_MICROSECOND(PyDateTime_Time *o)

Return the microsecond, as an int from 0 through 999999.

int PyDateTime_TIME_GET_FOLD(PyDateTime_Time *o)

Return the fold, as an int from 0 through 1.

Added in version 3.6.

PyObject *PyDateTime_TIME_GET_TZINFO(PyDateTime_Time *o)

Return the tzinfo (which may be `None`).

Added in version 3.10.

Macros to extract fields from time delta objects. The argument must be an instance of **`PyDateTime_Delta`**, including subclasses. The argument must not be `NULL`, and the type is not checked:

int PyDateTime_DELTA_GET_DAYS(PyDateTime_Delta *o)

Return the number of days, as an int from -999999999 to 999999999.

Added in version 3.3.

int PyDateTime_DELTA_GET_SECONDS(PyDateTime_Delta *o)

Return the number of seconds, as an int from 0 through 86399.

Added in version 3.3.

int PyDateTime_DELTA_GET_MICROSECONDS(PyDateTime_Delta *o)

Return the number of microseconds, as an int from 0 through 999999.

Added in version 3.3.

Macros for the convenience of modules implementing the DB API:

PyObject *PyDateTime_FromTimestamp(PyObject *args)

Return value: New reference.
Create and return a new **`datetime.datetime`** object given an argument tuple suitable for passing to **`datetime.datetime.fromtimestamp()`**.

PyObject *PyDate_FromTimestamp(PyObject *args)

Return value: New reference.
Create and return a new **`datetime.date`** object given an argument tuple suitable for passing to **`datetime.date.fromtimestamp()`**.

Internal data

The following symbols are exposed by the C API but should be considered internal-only.

PyDateTime_CAPSULE_NAME

Name of the datetime capsule to pass to `PyCapsule_Import()`.

Internal usage only. Use `PyDateTime_IMPORT` instead.

Initialization, Finalization, and Threads

See Python Initialization Configuration for details on how to configure the interpreter prior to initialization.

Before Python Initialization

In an application embedding Python, the **`Py_Initialize()`** function must be called before using any other Python/C API functions; with the exception of a few functions and the global configuration variables.

The following functions can be safely called before Python is initialized:

- Functions that initialize the interpreter:
  - **`Py_Initialize()`**
  - **`Py_InitializeEx()`**
  - **`Py_InitializeFromConfig()`**
  - **`Py_BytesMain()`**
  - **`Py_Main()`**
  - the runtime pre-initialization functions covered in Python Initialization Configuration
- Configuration functions:
  - **`PyImport_AppendInittab()`**
  - **`PyImport_ExtendInittab()`**
  - **`PyInitFrozenExtensions()`**
  - **`PyMem_SetAllocator()`**
  - **`PyMem_SetupDebugHooks()`**
  - **`PyObject_SetArenaAllocator()`**
  - **`Py_SetProgramName()`**

  - `Py_SetPythonHome()`
  - `PySys_ResetWarnOptions()`
  - the configuration functions covered in Python Initialization Configuration
- Informative functions:
  - `Py_IsInitialized()`
  - `PyMem_GetAllocator()`
  - `PyObject_GetArenaAllocator()`
  - `Py_GetBuildInfo()`
  - `Py_GetCompiler()`
  - `Py_GetCopyright()`
  - `Py_GetPlatform()`
  - `Py_GetVersion()`
  - `Py_IsInitialized()`
- Utilities:
  - `Py_DecodeLocale()`
  - the status reporting and utility functions covered in Python Initialization Configuration
- Memory allocators:
  - `PyMem_RawMalloc()`
  - `PyMem_RawRealloc()`
  - `PyMem_RawCalloc()`
  - `PyMem_RawFree()`
- Synchronization:
  - `PyMutex_Lock()`
  - `PyMutex_Unlock()`

Note

Despite their apparent similarity to some of the functions listed above, the following functions **should not be called** before the interpreter has been initialized: `Py_EncodeLocale()`, `Py_GetPath()`, `Py_GetPrefix()`, `Py_GetExecPrefix()`, `Py_GetProgramFullPath()`,

`Py_GetPythonHome()`, **`Py_GetProgramName()`**, **`PyEval_InitThreads()`**, and **`Py_RunMain()`**.

Global configuration variables

Python has variables for the global configuration to control different features and options. By default, these flags are controlled by command line options.

When a flag is set by an option, the value of the flag is the number of times that the option was set. For example, `-b` sets **`Py_BytesWarningFlag`** to 1 and `-bb` sets **`Py_BytesWarningFlag`** to 2.

int Py_BytesWarningFlag

This API is kept for backward compatibility: setting **`PyConfig.bytes_warning`** should be used instead, see Python Initialization Configuration.

Issue a warning when comparing **`bytes`** or **`bytearray`** with **`str`** or **`bytes`** with **`int`**. Issue an error if greater or equal to `2`.

Set by the **`-b`** option.

Deprecated since version 3.12, will be removed in version 3.15.

int Py_DebugFlag

This API is kept for backward compatibility: setting **`PyConfig.parser_debug`** should be used instead, see Python Initialization Configuration.

Turn on parser debugging output (for expert only, depending on compilation options).

Set by the `-d` option and the `PYTHONDEBUG` environment variable.

Deprecated since version 3.12, will be removed in version 3.15.

int Py_DontWriteBytecodeFlag

This API is kept for backward compatibility: setting `PyConfig.write_bytecode` should be used instead, see Python Initialization Configuration.

If set to non-zero, Python won't try to write `.pyc` files on the import of source modules.

Set by the `-B` option and the `PYTHONDONTWRITEBYTECODE` environment variable.

Deprecated since version 3.12, will be removed in version 3.15.

int Py_FrozenFlag

This API is kept for backward compatibility: setting `PyConfig.pathconfig_warnings` should be used instead, see Python Initialization Configuration.

Suppress error messages when calculating the module search path in `Py_GetPath()`.

Private flag used by `_freeze_module` and `frozenmain` programs.

Deprecated since version 3.12, will be removed in version 3.15.

int Py_HashRandomizationFlag

This API is kept for backward compatibility: setting `PyConfig.hash_seed` and `PyConfig.use_hash_seed` should be used instead, see Python Initialization Configuration.

Set to `1` if the **`PYTHONHASHSEED`** environment variable is set to a non-empty string.

If the flag is non-zero, read the **`PYTHONHASHSEED`** environment variable to initialize the secret hash seed.

Deprecated since version 3.12, will be removed in version 3.15.

int Py_IgnoreEnvironmentFlag

This API is kept for backward compatibility: setting **`PyConfig.use_environment`** should be used instead, see Python Initialization Configuration.

Ignore all **`PYTHON*`** environment variables, e.g. **`PYTHONPATH`** and **`PYTHONHOME`**, that might be set.

Set by the **`-E`** and **`-I`** options.

Deprecated since version 3.12, will be removed in version 3.15.

int Py_InspectFlag

This API is kept for backward compatibility: setting **`PyConfig.inspect`** should be used instead, see Python Initialization Configuration.

When a script is passed as first argument or the **`-c`** option is used, enter interactive mode after executing the script or the command, even when **`sys.stdin`** does not appear to be a terminal.

Set by the **`-i`** option and the **`PYTHONINSPECT`** environment variable.

Deprecated since version 3.12, will be removed in version 3.15.

int Py_InteractiveFlag

This API is kept for backward compatibility: setting `PyConfig.interactive` should be used instead, see Python Initialization Configuration.

Set by the `-i` option.

Deprecated since version 3.12, will be removed in version 3.15.

int Py_IsolatedFlag

This API is kept for backward compatibility: setting `PyConfig.isolated` should be used instead, see Python Initialization Configuration.

Run Python in isolated mode. In isolated mode `sys.path` contains neither the script's directory nor the user's site-packages directory.

Set by the `-I` option.

Added in version 3.4.

Deprecated since version 3.12, will be removed in version 3.15.

int Py_LegacyWindowsFSEncodingFlag

This API is kept for backward compatibility: setting `PyPreConfig.legacy_windows_fs_encoding` should be used instead, see Python Initialization Configuration.

If the flag is non-zero, use the `mbcs` encoding with `replace` error handler, instead of the UTF-8 encoding with `surrogatepass` error handler, for the filesystem encoding and error handler.

Set to `1` if the **`PYTHONLEGACYWINDOWSFSENCODING`** environment variable is set to a non-empty string.

See **PEP 529** [https://peps.python.org/pep-0529/] for more details.

Availability: Windows.

Deprecated since version 3.12, will be removed in version 3.15.

int Py_LegacyWindowsStdioFlag

This API is kept for backward compatibility: setting **`PyConfig.legacy_windows_stdio`** should be used instead, see Python Initialization Configuration.

If the flag is non-zero, use **`io.FileIO`** instead of **`io._WindowsConsoleIO`** for **`sys`** standard streams.

Set to `1` if the **`PYTHONLEGACYWINDOWSSTDIO`** environment variable is set to a non-empty string.

See **PEP 528** [https://peps.python.org/pep-0528/] for more details.

Availability: Windows.

Deprecated since version 3.12, will be removed in version 3.15.

int Py_NoSiteFlag

This API is kept for backward compatibility: setting **`PyConfig.site_import`** should be used instead, see Python Initialization Configuration.

Disable the import of the module `site` and the site-dependent manipulations of `sys.path` that it entails. Also disable these manipulations if `site` is explicitly imported later (call `site.main()` if you want them to be triggered).

Set by the `-S` option.

Deprecated since version 3.12, will be removed in version 3.15.

int Py_NoUserSiteDirectory

This API is kept for backward compatibility: setting `PyConfig.user_site_directory` should be used instead, see Python Initialization Configuration.

Don't add the `user site-packages directory` to `sys.path`.

Set by the `-s` and `-I` options, and the `PYTHONNOUSERSITE` environment variable.

Deprecated since version 3.12, will be removed in version 3.15.

int Py_OptimizeFlag

This API is kept for backward compatibility: setting `PyConfig.optimization_level` should be used instead, see Python Initialization Configuration.

Set by the `-O` option and the `PYTHONOPTIMIZE` environment variable.

Deprecated since version 3.12, will be removed in version 3.15.

int Py_QuietFlag

This API is kept for backward compatibility: setting `PyConfig.quiet` should be used instead, see Python

Initialization Configuration.

Don’t display the copyright and version messages even in interactive mode.

Set by the `-q` option.

Added in version 3.2.

Deprecated since version 3.12, will be removed in version 3.15.

int Py_UnbufferedStdioFlag

This API is kept for backward compatibility: setting `PyConfig.buffered_stdio` should be used instead, see Python Initialization Configuration.

Force the stdout and stderr streams to be unbuffered.

Set by the `-u` option and the `PYTHONUNBUFFERED` environment variable.

Deprecated since version 3.12, will be removed in version 3.15.

int Py_VerboseFlag

This API is kept for backward compatibility: setting `PyConfig.verbose` should be used instead, see Python Initialization Configuration.

Print a message each time a module is initialized, showing the place (filename or built-in module) from which it is loaded. If greater or equal to `2`, print a message for each file that is checked for when searching for a module. Also provides information on module cleanup at exit.

Set by the **`-v`** option and the **`PYTHONVERBOSE`** environment variable.

Deprecated since version 3.12, will be removed in version 3.15.

Initializing and finalizing the interpreter

void Py_Initialize()

Part of the Stable ABI.
Initialize the Python interpreter. In an application embedding Python, this should be called before using any other Python/C API functions; see Before Python Initialization for the few exceptions.

This initializes the table of loaded modules (`sys.modules`), and creates the fundamental modules **`builtins`**, **`__main__`** and **`sys`**. It also initializes the module search path (`sys.path`). It does not set `sys.argv`; use the Python Initialization Configuration API for that. This is a no-op when called for a second time (without calling **`Py_FinalizeEx()`** first). There is no return value; it is a fatal error if the initialization fails.

Use **`Py_InitializeFromConfig()`** to customize the Python Initialization Configuration.

Note

On Windows, changes the console mode from `O_TEXT` to `O_BINARY`, which will also affect non-Python uses of the console using the C Runtime.

void Py_InitializeEx(int initsigs)

Part of the Stable ABI.
This function works like **`Py_Initialize()`** if *initsigs* is `1`. If *initsigs* is `0`, it skips initialization registration of signal handlers, which may be useful when CPython is embedded as part of a larger application.

Use **`Py_InitializeFromConfig()`** to customize the Python Initialization Configuration.

PyStatus Py_InitializeFromConfig(const PyConfig *config)

Initialize Python from *config* configuration, as described in Initialization with PyConfig.

See the Python Initialization Configuration section for details on pre-initializing the interpreter, populating the runtime configuration structure, and querying the returned status structure.

int Py_IsInitialized()

Part of the Stable ABI.
Return true (nonzero) when the Python interpreter has been initialized, false (zero) if not. After **`Py_FinalizeEx()`** is called, this returns false until **`Py_Initialize()`** is called again.

int Py_IsFinalizing()

Part of the Stable ABI since version 3.13.
Return true (non-zero) if the main Python interpreter is shutting down. Return false (zero) otherwise.

Added in version 3.13.

int Py_FinalizeEx()

Part of the Stable ABI since version 3.6.
Undo all initializations made by **`Py_Initialize()`** and subsequent use of Python/C API functions, and destroy all

sub-interpreters (see `Py_NewInterpreter()` below) that were created and not yet destroyed since the last call to `Py_Initialize()`. This is a no-op when called for a second time (without calling `Py_Initialize()` again first).

Since this is the reverse of `Py_Initialize()`, it should be called in the same thread with the same interpreter active. That means the main thread and the main interpreter. This should never be called while `Py_RunMain()` is running.

Normally the return value is `0`. If there were errors during finalization (flushing buffered data), `-1` is returned.

Note that Python will do a best effort at freeing all memory allocated by the Python interpreter. Therefore, any C-Extension should make sure to correctly clean up all of the preveiously allocated PyObjects before using them in subsequent calls to `Py_Initialize()`. Otherwise it could introduce vulnerabilities and incorrect behavior.

This function is provided for a number of reasons. An embedding application might want to restart Python without having to restart the application itself. An application that has loaded the Python interpreter from a dynamically loadable library (or DLL) might want to free all memory allocated by Python before unloading the DLL. During a hunt for memory leaks in an application a developer might want to free all memory allocated by Python before exiting from the application.

Bugs and caveats: The destruction of modules and objects in modules is done in random order; this may cause destructors (`__del__()` methods) to fail when they depend on other objects (even functions) or modules. Dynamically loaded extension modules loaded by Python are not unloaded. Small amounts of memory allocated by the Python interpreter may not be freed (if you find a leak, please report it). Memory tied up in circular references between objects is not freed. Interned strings will all be deallocated regardless of their reference count. Some memory allocated by extension modules may not be freed. Some extensions may not work properly if their initialization routine is called more than once; this can happen if an application calls **`Py_Initialize()`** and **`Py_FinalizeEx()`** more than once. **`Py_FinalizeEx()`** must not be called recursively from within itself. Therefore, it must not be called by any code that may be run as part of the interpreter shutdown process, such as **`atexit`** handlers, object finalizers, or any code that may be run while flushing the stdout and stderr files.

Raises an auditing event `cpython._PySys_ClearAuditHooks` with no arguments.

Added in version 3.6.

void Py_Finalize()

Part of the Stable ABI.
This is a backwards-compatible version of **`Py_FinalizeEx()`** that disregards the return value.

int Py_BytesMain(int argc, char **argv)

Part of the Stable ABI since version 3.8.
Similar to **`Py_Main()`** but *argv* is an array of bytes strings, allowing the calling application to delegate the text decoding step to the CPython runtime.

Added in version 3.8.

int Py_Main(int argc, wchar_t **argv)

Part of the Stable ABI.

The main program for the standard interpreter, encapsulating a full initialization/finalization cycle, as well as additional behaviour to implement reading configurations settings from the environment and command line, and then executing `__main__` in accordance with Command line.

This is made available for programs which wish to support the full CPython command line interface, rather than just embedding a Python runtime in a larger application.

The *argc* and *argv* parameters are similar to those which are passed to a C program’s **`main()`** function, except that the *argv* entries are first converted to `wchar_t` using **`Py_DecodeLocale()`**. It is also important to note that the argument list entries may be modified to point to strings other than those passed in (however, the contents of the strings pointed to by the argument list are not modified).

The return value is `2` if the argument list does not represent a valid Python command line, and otherwise the same as **`Py_RunMain()`**.

In terms of the CPython runtime configuration APIs documented in the runtime configuration section (and without accounting for error handling), `Py_Main` is approximately equivalent to:

```
PyConfig config;
PyConfig_InitPythonConfig(&config);
PyConfig_SetArgv(&config, argc, argv);
Py_InitializeFromConfig(&config);
PyConfig_Clear(&config);

Py_RunMain();
```

In normal usage, an embedding application will call this function *instead* of calling `Py_Initialize()`, `Py_InitializeEx()` or `Py_InitializeFromConfig()` directly, and all settings will be applied as described elsewhere in this documentation. If this function is instead called *after* a preceding runtime initialization API call, then exactly which environmental and command line configuration settings will be updated is version dependent (as it depends on which settings correctly support being modified after they have already been set once when the runtime was first initialized).

int Py_RunMain(void)

Executes the main module in a fully configured CPython runtime.

Executes the command (`PyConfig.run_command`), the script (`PyConfig.run_filename`) or the module (`PyConfig.run_module`) specified on the command line or in the configuration. If none of these values are set, runs the interactive Python prompt (REPL) using the `__main__` module's global namespace.

If `PyConfig.inspect` is not set (the default), the return value will be `0` if the interpreter exits normally (that is, without raising an exception), the exit status of an unhandled `SystemExit`, or `1` for any other unhandled exception.

If `PyConfig.inspect` is set (such as when the `-i` option is used), rather than returning when the interpreter exits, execution will instead resume in an interactive Python prompt (REPL) using the `__main__` module's global namespace. If the interpreter exited with an exception, it is immediately raised in the REPL session. The function return value is then determined by the way the *REPL* *session* terminates: `0`, `1`, or the status of a **`SystemExit`**, as specified above.

This function always finalizes the Python interpreter before it returns.

See Python Configuration for an example of a customized Python that always runs in isolated mode using **`Py_RunMain()`**.

int PyUnstable_AtExit(PyInterpreterState *interp, void (*func)(void*), void *data)

This is Unstable API. It may change without warning in minor releases.

Register an **`atexit`** callback for the target interpreter *interp*. This is similar to **`Py_AtExit()`**, but takes an explicit interpreter and data pointer for the callback.

There must be an attached thread state for *interp*.

Added in version 3.13.

Process-wide parameters

void Py_SetProgramName(const wchar_t *name)

Part of the Stable ABI.
This API is kept for backward compatibility: setting **`PyConfig.program_name`** should be used instead, see Python Initialization Configuration.

This function should be called before **`Py_Initialize()`** is called for the first time, if it is called at all. It tells the interpreter the value of the `argv[0]` argument to the **`main()`** function of the program (converted to wide characters).

This is used by `Py_GetPath()` and some other functions below to find the Python run-time libraries relative to the interpreter executable. The default value is `'python'`. The argument should point to a zero-terminated wide character string in static storage whose contents will not change for the duration of the program's execution. No code in the Python interpreter will change the contents of this storage.

Use `Py_DecodeLocale()` to decode a bytes string to get a wchar_t* string.

Deprecated since version 3.11, will be removed in version 3.15.

wchar_t *Py_GetProgramName()

Part of the Stable ABI.
Return the program name set with `PyConfig.program_name`, or the default. The returned string points into static storage; the caller should not modify its value.

This function should not be called before `Py_Initialize()`, otherwise it returns `NULL`.

Changed in version 3.10: It now returns `NULL` if called before `Py_Initialize()`.

Deprecated since version 3.13, will be removed in version 3.15: Use `PyConfig_Get("executable")` (`sys.executable`) instead.

wchar_t *Py_GetPrefix()

Part of the Stable ABI.
Return the *prefix* for installed platform-independent files. This is derived through a number of complicated rules from the program name set with `PyConfig.program_name` and

some environment variables; for example, if the program name is `'/usr/local/bin/python'`, the prefix is `'/usr/local'`. The returned string points into static storage; the caller should not modify its value. This corresponds to the **prefix** variable in the top-level `Makefile` and the `--prefix` argument to the **configure** script at build time. The value is available to Python code as `sys.base_prefix`. It is only useful on Unix. See also the next function.

This function should not be called before `Py_Initialize()`, otherwise it returns `NULL`.

Changed in version 3.10: It now returns `NULL` if called before `Py_Initialize()`.

Deprecated since version 3.13, will be removed in version 3.15: Use `PyConfig_Get("base_prefix")` (`sys.base_prefix`) instead. Use `PyConfig_Get("prefix")` (`sys.prefix`) if virtual environments need to be handled.

wchar_t *Py_GetExecPrefix()

Part of the Stable ABI.
Return the *exec-prefix* for installed platform-*dependent* files. This is derived through a number of complicated rules from the program name set with `PyConfig.program_name` and some environment variables; for example, if the program name is `'/usr/local/bin/python'`, the exec-prefix is `'/usr/local'`. The returned string points into static storage; the caller should not modify its value. This corresponds to the **exec_prefix** variable in the top-level `Makefile` and the `--exec-prefix` argument to the **configure** script at build time. The value is available to Python code as `sys.base_exec_prefix`. It is only useful on Unix.

Background: The exec-prefix differs from the prefix when platform dependent files (such as executables and shared libraries) are installed in a different directory tree. In a typical installation, platform dependent files may be installed in the `/usr/local/plat` subtree while platform independent may be installed in `/usr/local`.

Generally speaking, a platform is a combination of hardware and software families, e.g. Sparc machines running the Solaris 2.x operating system are considered the same platform, but Intel machines running Solaris 2.x are another platform, and Intel machines running Linux are yet another platform. Different major revisions of the same operating system generally also form different platforms. Non-Unix operating systems are a different story; the installation strategies on those systems are so different that the prefix and exec-prefix are meaningless, and set to the empty string. Note that compiled Python bytecode files are platform independent (but not independent from the Python version by which they were compiled!).

System administrators will know how to configure the **mount** or **automount** programs to share `/usr/local` between platforms while having `/usr/local/plat` be a different filesystem for each platform.

This function should not be called before `Py_Initialize()`, otherwise it returns `NULL`.

Changed in version 3.10: It now returns `NULL` if called before `Py_Initialize()`.

Deprecated since version 3.13, will be removed in version 3.15: Use `PyConfig_Get("base_exec_prefix")` (`sys.base_exec_prefix`) instead. Use

`PyConfig_Get("exec_prefix")` (`sys.exec_prefix`) if virtual environments need to be handled.

wchar_t *Py_GetProgramFullPath()

Part of the Stable ABI.
Return the full program name of the Python executable; this is computed as a side-effect of deriving the default module search path from the program name (set by `PyConfig.program_name`). The returned string points into static storage; the caller should not modify its value. The value is available to Python code as `sys.executable`.

This function should not be called before `Py_Initialize()`, otherwise it returns `NULL`.

Changed in version 3.10: It now returns `NULL` if called before `Py_Initialize()`.

Deprecated since version 3.13, will be removed in version 3.15: Use `PyConfig_Get("executable")` (`sys.executable`) instead.

wchar_t *Py_GetPath()

Part of the Stable ABI.
Return the default module search path; this is computed from the program name (set by `PyConfig.program_name`) and some environment variables. The returned string consists of a series of directory names separated by a platform dependent delimiter character. The delimiter character is `':'` on Unix and macOS, `';'` on Windows. The returned string points into static storage; the caller should not modify its value. The list `sys.path` is initialized with this value on interpreter startup; it can be (and usually is) modified later to change the search path for loading modules.

This function should not be called before **`Py_Initialize()`**, otherwise it returns `NULL`.

Changed in version 3.10: It now returns `NULL` if called before **`Py_Initialize()`**.

Deprecated since version 3.13, will be removed in version 3.15: Use **`PyConfig_Get("module_search_paths")`** (**`sys.path`**) instead.

const char *Py_GetVersion()

Part of the Stable ABI.

Return the version of this Python interpreter. This is a string that looks something like

```
"3.0a5+ (py3k:63103M, May 12 2008, 00:53:55) \n[GCC 4.2.3]"
```

The first word (up to the first space character) is the current Python version; the first characters are the major and minor version separated by a period. The returned string points into static storage; the caller should not modify its value. The value is available to Python code as **`sys.version`**.

See also the **`Py_Version`** constant.

const char *Py_GetPlatform()

Part of the Stable ABI.

Return the platform identifier for the current platform. On Unix, this is formed from the "official" name of the operating system, converted to lower case, followed by the major revision number; e.g., for Solaris 2.x, which is also known as SunOS 5.x, the value is `'sunos5'`. On macOS, it is `'darwin'`. On Windows, it is `'win'`. The returned string points into static storage; the caller should not modify its value. The value is available to Python code as `sys.platform`.

const char *Py_GetCopyright()

Part of the Stable ABI.

Return the official copyright string for the current Python version, for example

```
'Copyright 1991-1995 Stichting Mathematisch Centrum,
Amsterdam'
```

The returned string points into static storage; the caller should not modify its value. The value is available to Python code as `sys.copyright`.

const char *Py_GetCompiler()

Part of the Stable ABI.

Return an indication of the compiler used to build the current Python version, in square brackets, for example:

```
"[GCC 2.7.2.2]"
```

The returned string points into static storage; the caller should not modify its value. The value is available to Python code as part of the variable `sys.version`.

const char *Py_GetBuildInfo()

Part of the Stable ABI.

Return information about the sequence number and build date and time of the current Python interpreter instance, for example

```
"#67, Aug  1 1997, 22:34:28"
```

The returned string points into static storage; the caller should not modify its value. The value is available to Python code as part of the variable `sys.version`.

void PySys_SetArgvEx(int argc, wchar_t **argv, int updatepath)

Part of the Stable ABI.
This API is kept for backward compatibility: setting **`PyConfig.argv`**, **`PyConfig.parse_argv`** and **`PyConfig.safe_path`** should be used instead, see Python Initialization Configuration.

Set **`sys.argv`** based on *argc* and *argv*. These parameters are similar to those passed to the program's **`main()`** function with the difference that the first entry should refer to the script file to be executed rather than the executable hosting the Python interpreter. If there isn't a script that will be run, the first entry in *argv* can be an empty string. If this function fails to initialize **`sys.argv`**, a fatal condition is signalled using **`Py_FatalError()`**.

If *updatepath* is zero, this is all the function does. If *updatepath* is non-zero, the function also modifies **`sys.path`** according to the following algorithm:

- If the name of an existing script is passed in `argv[0]`, the absolute path of the directory where the script is located is prepended to **`sys.path`**.
- Otherwise (that is, if *argc* is `0` or `argv[0]` doesn't point to an existing file name), an empty string is prepended to **`sys.path`**, which is the same as prepending the current working directory (`"."`).

Use **`Py_DecodeLocale()`** to decode a bytes string to get a wchar_t* string.

See also **`PyConfig.orig_argv`** and **`PyConfig.argv`** members of the Python Initialization Configuration.

Note

It is recommended that applications embedding the Python interpreter for purposes other than executing a single script pass `0` as *updatepath*, and update `sys.path` themselves if desired. See **CVE 2008-5983** [https://www.cve.org/CVERecord?id=CVE-2008-5983].

On versions before 3.1.3, you can achieve the same effect by manually popping the first `sys.path` element after having called `PySys_SetArgv()`, for example using:

```
PyRun_SimpleString("import sys; sys.path.pop(0)\n");
```

Added in version 3.1.3.

Deprecated since version 3.11, will be removed in version 3.15.

void PySys_SetArgv(int argc, wchar_t **argv)

Part of the Stable ABI.

This API is kept for backward compatibility: setting `PyConfig.argv` and `PyConfig.parse_argv` should be used instead, see Python Initialization Configuration.

This function works like `PySys_SetArgvEx()` with *updatepath* set to `1` unless the **python** interpreter was started with the `-I`.

Use `Py_DecodeLocale()` to decode a bytes string to get a wchar_t* string.

See also `PyConfig.orig_argv` and `PyConfig.argv` members of the Python Initialization Configuration.

Changed in version 3.4: The *updatepath* value depends on `-I`.

Deprecated since version 3.11, will be removed in version 3.15.

void Py_SetPythonHome(const wchar_t *home)

Part of the Stable ABI.

This API is kept for backward compatibility: setting `PyConfig.home` should be used instead, see Python Initialization Configuration.

Set the default “home” directory, that is, the location of the standard Python libraries. See `PYTHONHOME` for the meaning of the argument string.

The argument should point to a zero-terminated character string in static storage whose contents will not change for the duration of the program’s execution. No code in the Python interpreter will change the contents of this storage.

Use `Py_DecodeLocale()` to decode a bytes string to get a wchar_t* string.

Deprecated since version 3.11, will be removed in version 3.15.

wchar_t *Py_GetPythonHome()

Part of the Stable ABI.

Return the default “home”, that is, the value set by `PyConfig.home`, or the value of the `PYTHONHOME` environment variable if it is set.

This function should not be called before `Py_Initialize()`, otherwise it returns `NULL`.

Changed in version 3.10: It now returns `NULL` if called before `Py_Initialize()`.

Deprecated since version 3.13, will be removed in version 3.15: Use `PyConfig_Get("home")` or the `PYTHONHOME` environment variable instead.

Thread State and the Global Interpreter Lock

Unless on a free-threaded build of CPython, the Python interpreter is not fully thread-safe. In order to support multi-threaded Python programs, there’s a global lock, called the global interpreter lock or GIL, that must be held by the current thread before it can safely access Python objects. Without the lock, even the simplest operations could cause problems in a multi-threaded program: for example, when two threads simultaneously increment the reference count of the same object, the reference count could end up being incremented only once instead of twice.

Therefore, the rule exists that only the thread that has acquired the GIL may operate on Python objects or call Python/C API functions. In order to emulate concurrency of execution, the interpreter regularly tries to switch threads (see **`sys.setswitchinterval()`**). The lock is also released around potentially blocking I/O operations like reading or writing a file, so that other Python threads can run in the meantime.

The Python interpreter keeps some thread-specific bookkeeping information inside a data structure called **`PyThreadState`**, known as a thread state. Each OS thread has a thread-local pointer to a **`PyThreadState`**; a thread state referenced by this pointer is considered to be attached.

A thread can only have one attached thread state at a time. An attached thread state is typically analogous with holding the GIL, except on free-threaded builds. On builds with the GIL enabled, attaching a thread state will block until the GIL can be acquired. However, even on builds with the GIL disabled, it is still required to have an attached thread state to call most of the C API.

In general, there will always be an attached thread state when using Python's C API. Only in some specific cases (such as in a **`Py_BEGIN_ALLOW_THREADS`** block) will the thread not have an attached thread state. If uncertain, check if **`PyThreadState_GetUnchecked()`** returns `NULL`.

Detaching the thread state from extension code

Most extension code manipulating the thread state has the following simple structure:

```
Save the thread state in a local variable.
... Do some blocking I/O operation ...
Restore the thread state from the local variable.
```

This is so common that a pair of macros exists to simplify it:

```
Py_BEGIN_ALLOW_THREADS
... Do some blocking I/O operation ...
Py_END_ALLOW_THREADS
```

The **`Py_BEGIN_ALLOW_THREADS`** macro opens a new block and declares a hidden local variable; the **`Py_END_ALLOW_THREADS`** macro closes the block.

The block above expands to the following code:

```
PyThreadState *_save;

_save = PyEval_SaveThread();
... Do some blocking I/O operation ...
PyEval_RestoreThread(_save);
```

Here is how these functions work:

The attached thread state holds the GIL for the entire interpreter. When detaching the attached thread state, the GIL is released, allowing other threads to attach a thread state to their own thread, thus getting the GIL and can start executing. The pointer to the prior attached thread state is stored as a local variable. Upon reaching `Py_END_ALLOW_THREADS`, the thread state that was previously attached is passed to `PyEval_RestoreThread()`. This function will block until another releases its thread state, thus allowing the old thread state to get re-attached and the C API can be called again.

For free-threaded builds, the GIL is normally out of the question, but detaching the thread state is still required for blocking I/O and long operations. The difference is that threads don’t have to wait for the GIL to be released to attach their thread state, allowing true multi-core parallelism.

Note

Calling system I/O functions is the most common use case for detaching the thread state, but it can also be useful before calling long-running computations which don’t need access to Python objects, such as compression or cryptographic functions operating over memory buffers. For example, the standard `zlib` and `hashlib` modules detach the thread state when compressing or hashing data.

Non-Python created threads

When threads are created using the dedicated Python APIs (such as the `threading` module), a thread state is automatically associated to them and the code showed above is therefore correct. However, when threads are created from C (for example by a third-party library with its own thread management), they don’t hold the GIL, because they don’t have an attached thread state.

If you need to call Python code from these threads (often this will be part of a callback API provided by the aforementioned third-party library), you must first register these threads with the interpreter by creating an attached thread state before you can start using the Python/C API. When you are done, you should detach the thread state, and finally free it.

The **`PyGILState_Ensure()`** and **`PyGILState_Release()`** functions do all of the above automatically. The typical idiom for calling into Python from a C thread is:

```
PyGILState_STATE gstate;
gstate = PyGILState_Ensure();

/* Perform Python actions here. */
result = CallSomeFunction();
/* evaluate result or handle exception */

/* Release the thread. No Python API allowed beyond this point.
*/
PyGILState_Release(gstate);
```

Note that the `PyGILState_*` functions assume there is only one global interpreter (created automatically by **`Py_Initialize()`**). Python supports the creation of additional interpreters (using **`Py_NewInterpreter()`**), but mixing multiple interpreters and the `PyGILState_*` API is unsupported. This is because **`PyGILState_Ensure()`** and similar functions default to attaching a thread state for the main interpreter, meaning that the thread can’t safely interact with the calling subinterpreter.

Supporting subinterpreters in non-Python threads

If you would like to support subinterpreters with non-Python created threads, you must use the `PyThreadState_*` API instead of the traditional `PyGILState_*` API.

In particular, you must store the interpreter state from the calling function and pass it to **`PyThreadState_New()`**, which will ensure that the thread state is targeting the correct interpreter:

```
/* The return value of PyInterpreterState_Get() from the
   function that created this thread. */
PyInterpreterState *interp = ThreadData->interp;
PyThreadState *tstate = PyThreadState_New(interp);
PyThreadState_Swap(tstate);

/* GIL of the subinterpreter is now held.
   Perform Python actions here. */
result = CallSomeFunction();
/* evaluate result or handle exception */

/* Destroy the thread state. No Python API allowed beyond this
point. */
PyThreadState_Clear(tstate);
PyThreadState_DeleteCurrent();
```

Cautions about fork()

Another important thing to note about threads is their behaviour in the face of the C **`fork()`** call. On most systems with **`fork()`**, after a process forks only the thread that issued the fork will exist. This has a concrete impact both on how locks must be handled and on all stored state in CPython’s runtime.

The fact that only the “current” thread remains means any locks held by other threads will never be released. Python solves this for `os.fork()` by acquiring the locks it uses internally before the fork, and releasing them afterwards. In addition, it resets any Lock objects in the child. When extending or embedding Python, there is no way to inform Python of additional (non-Python) locks that need to be acquired before or reset after a fork. OS facilities such as `pthread_atfork()` would need to be used to accomplish the same thing. Additionally, when extending or embedding Python, calling `fork()` directly rather than through `os.fork()` (and returning to or calling into Python) may result in a deadlock by one of Python’s internal locks being held by a thread that is defunct after the fork. `PyOS_AfterFork_Child()` tries to reset the necessary locks, but is not always able to.

The fact that all other threads go away also means that CPython’s runtime state there must be cleaned up properly, which `os.fork()` does. This means finalizing all other `PyThreadState` objects belonging to the current interpreter and all other `PyInterpreterState` objects. Due to this and the special nature of the “main” interpreter, `fork()` should only be called in that interpreter’s “main” thread, where the CPython global runtime was originally initialized. The only exception is if `exec()` will be called immediately after.

Cautions regarding runtime finalization

In the late stage of interpreter shutdown, after attempting to wait for non-daemon threads to exit (though this can be interrupted by `KeyboardInterrupt`) and running the `atexit` functions, the runtime is marked as *finalizing*: `Py_IsFinalizing()` and `sys.is_finalizing()` return true. At this point, only the *finalization thread* that initiated finalization (typically the main thread) is allowed to acquire the GIL.

If any thread, other than the finalization thread, attempts to attach a thread state during finalization, either explicitly or implicitly, the thread enters **a permanently blocked state** where it remains until the program exits. In most cases this is harmless, but this can result in deadlock if a later stage of finalization attempts to acquire a lock owned by the blocked thread, or otherwise waits on the blocked thread.

Gross? Yes. This prevents random crashes and/or unexpectedly skipped C++ finalizations further up the call stack when such threads were forcibly exited here in CPython 3.13 and earlier. The CPython runtime thread state C APIs have never had any error reporting or handling expectations at thread state attachment time that would've allowed for graceful exit from this situation. Changing that would require new stable C APIs and rewriting the majority of C code in the CPython ecosystem to use those with error handling.

High-level API

These are the most commonly used types and functions when writing C extension code, or when embedding the Python interpreter:

type PyInterpreterState

Part of the Limited API (as an opaque struct).
This data structure represents the state shared by a number of cooperating threads. Threads belonging to the same interpreter share their module administration and a few other internal items. There are no public members in this structure.

Threads belonging to different interpreters initially share nothing, except process state like available memory, open file descriptors and such. The global interpreter lock is also shared by all threads, regardless of to which interpreter they belong.

Changed in version 3.12: **PEP 684** [https://peps.python.org/pep-0684/] introduced the possibility of a per-interpreter GIL. See **`Py_NewInterpreterFromConfig()`**.

type PyThreadState

Part of the Limited API (as an opaque struct).

This data structure represents the state of a single thread. The only public data member is:

PyInterpreterState *interp

This thread's interpreter state.

void PyEval_InitThreads()

Part of the Stable ABI.

Deprecated function which does nothing.

In Python 3.6 and older, this function created the GIL if it didn't exist.

Changed in version 3.9: The function now does nothing.

Changed in version 3.7: This function is now called by **`Py_Initialize()`**, so you don't have to call it yourself anymore.

Changed in version 3.2: This function cannot be called before **`Py_Initialize()`** anymore.

Deprecated since version 3.9.

PyThreadState *PyEval_SaveThread()

Part of the Stable ABI.

Detach the attached thread state and return it. The thread will have no thread state upon returning.

void PyEval_RestoreThread(PyThreadState *tstate)

Part of the Stable ABI.

Set the attached thread state to *tstate*. The passed thread state **should not** be attached, otherwise deadlock ensues. *tstate* will be attached upon returning.

Note

Calling this function from a thread when the runtime is finalizing will hang the thread until the program exits, even if the thread was not created by Python. Refer to Cautions regarding runtime finalization for more details.

Changed in version 3.14: Hangs the current thread, rather than terminating it, if called while the interpreter is finalizing.

PyThreadState *PyThreadState_Get()

Part of the Stable ABI.

Return the attached thread state. If the thread has no attached thread state, (such as when inside of **`Py_BEGIN_ALLOW_THREADS`** block), then this issues a fatal error (so that the caller needn't check for `NULL`).

See also **`PyThreadState_GetUnchecked()`**.

PyThreadState *PyThreadState_GetUnchecked()

Similar to **`PyThreadState_Get()`**, but don't kill the process with a fatal error if it is NULL. The caller is responsible to check if the result is NULL.

Added in version 3.13: In Python 3.5 to 3.12, the function was private and known as `_PyThreadState_UncheckedGet()`.

PyThreadState *PyThreadState_Swap(PyThreadState *tstate)

Part of the Stable ABI.

Set the attached thread state to *tstate*, and return the thread state that was attached prior to calling.

This function is safe to call without an attached thread state; it will simply return `NULL` indicating that there was no prior thread state.

See also

`PyEval_ReleaseThread()`

Note

Similar to **`PyGILState_Ensure()`**, this function will hang the thread if the runtime is finalizing.

The following functions use thread-local storage, and are not compatible with sub-interpreters:

type PyGILState_STATE

Part of the Stable ABI.

The type of the value returned by **`PyGILState_Ensure()`** and passed to **`PyGILState_Release()`**.

enumerator PyGILState_LOCKED

The GIL was already held when **`PyGILState_Ensure()`** was called.

enumerator PyGILState_UNLOCKED

The GIL was not held when **`PyGILState_Ensure()`** was called.

PyGILState_STATE PyGILState_Ensure()

Part of the Stable ABI.

Ensure that the current thread is ready to call the Python C API regardless of the current state of Python, or of the attached thread state. This may be called as many times as desired by a thread as long as each call is matched with a call to **`PyGILState_Release()`**. In general, other thread-related APIs may be used between **`PyGILState_Ensure()`** and **`PyGILState_Release()`** calls as long as the thread state is restored to its previous state before the Release(). For example, normal usage of the **`Py_BEGIN_ALLOW_THREADS`** and **`Py_END_ALLOW_THREADS`** macros is acceptable.

The return value is an opaque “handle” to the attached thread state when **`PyGILState_Ensure()`** was called, and must be passed to **`PyGILState_Release()`** to ensure Python is left in the same state. Even though recursive calls are allowed, these handles *cannot* be shared - each unique call to **`PyGILState_Ensure()`** must save the handle for its call to **`PyGILState_Release()`**.

When the function returns, there will be an attached thread state and the thread will be able to call arbitrary Python code. Failure is a fatal error.

Warning

Calling this function when the runtime is finalizing is unsafe. Doing so will either hang the thread until the program ends, or fully crash the interpreter in rare cases. Refer to Cautions regarding runtime finalization for more details.

Changed in version 3.14: Hangs the current thread, rather than terminating it, if called while the interpreter is finalizing.

void PyGILState_Release(PyGILState_STATE)

Part of the Stable ABI.

Release any resources previously acquired. After this call, Python’s state will be the same as it was prior to the corresponding **`PyGILState_Ensure()`** call (but generally this state will be unknown to the caller, hence the use of the GILState API).

Every call to **`PyGILState_Ensure()`** must be matched by a call to **`PyGILState_Release()`** on the same thread.

PyThreadState *PyGILState_GetThisThreadState()

Part of the Stable ABI.

Get the attached thread state for this thread. May return `NULL` if no GILState API has been used on the current thread. Note that the main thread always has such a thread-state, even if no auto-thread-state call has been made on the main thread. This is mainly a helper/diagnostic function.

Note

This function may return non-`NULL` even when the thread state is detached. Prefer **`PyThreadState_Get()`** or **`PyThreadState_GetUnchecked()`** for most cases.

See also

`PyThreadState_Get()`

int PyGILState_Check()

Return `1` if the current thread is holding the GIL and `0` otherwise. This function can be called from any thread at any time. Only if it has had its thread state initialized via `PyGILState_Ensure()` will it return `1`. This is mainly a helper/diagnostic function. It can be useful for example in callback contexts or memory allocation functions when knowing that the GIL is locked can allow the caller to perform sensitive actions or otherwise behave differently.

Note

If the current Python process has ever created a subinterpreter, this function will *always* return `1`. Prefer `PyThreadState_GetUnchecked()` for most cases.

Added in version 3.4.

The following macros are normally used without a trailing semicolon; look for example usage in the Python source distribution.

Py_BEGIN_ALLOW_THREADS

Part of the Stable ABI.

This macro expands to `{ PyThreadState *_save; _save = PyEval_SaveThread();`. Note that it contains an opening brace; it must be matched with a following `Py_END_ALLOW_THREADS` macro. See above for further discussion of this macro.

Py_END_ALLOW_THREADS

Part of the Stable ABI.

This macro expands to `PyEval_RestoreThread(_save); }`. Note that it contains a closing brace; it must be matched with an earlier `Py_BEGIN_ALLOW_THREADS` macro. See above for further discussion of this macro.

Py_BLOCK_THREADS

Part of the Stable ABI.

This macro expands to `PyEval_RestoreThread(_save);`: it is equivalent to **`Py_END_ALLOW_THREADS`** without the closing brace.

Py_UNBLOCK_THREADS

Part of the Stable ABI.

This macro expands to `_save = PyEval_SaveThread();`: it is equivalent to **`Py_BEGIN_ALLOW_THREADS`** without the opening brace and variable declaration.

Low-level API

All of the following functions must be called after **`Py_Initialize()`**.

Changed in version 3.7: **`Py_Initialize()`** now initializes the GIL and sets an attached thread state.

PyInterpreterState *PyInterpreterState_New()

Part of the Stable ABI.

Create a new interpreter state object. An attached thread state is not needed, but may optionally exist if it is necessary to serialize calls to this function.

Raises an auditing event `cpython.PyInterpreterState_New` with no arguments.

void PyInterpreterState_Clear(PyInterpreterState *interp)

Part of the Stable ABI.

Reset all information in an interpreter state object. There must be an attached thread state for the interpreter.

Raises an auditing event `cpython.PyInterpreterState_Clear` with no arguments.

void PyInterpreterState_Delete(PyInterpreterState *interp)

Part of the Stable ABI.
Destroy an interpreter state object. There **should not** be an attached thread state for the target interpreter. The interpreter state must have been reset with a previous call to `PyInterpreterState_Clear()`.

PyThreadState *PyThreadState_New(PyInterpreterState *interp)

Part of the Stable ABI.
Create a new thread state object belonging to the given interpreter object. An attached thread state is not needed.

void PyThreadState_Clear(PyThreadState *tstate)

Part of the Stable ABI.
Reset all information in a thread state object. *tstate* must be attached

Changed in version 3.9: This function now calls the `PyThreadState.on_delete` callback. Previously, that happened in `PyThreadState_Delete()`.

Changed in version 3.13: The `PyThreadState.on_delete` callback was removed.

void PyThreadState_Delete(PyThreadState *tstate)

Part of the Stable ABI.
Destroy a thread state object. *tstate* should not be attached to any thread. *tstate* must have been reset with a previous call to `PyThreadState_Clear()`.

void PyThreadState_DeleteCurrent(void)

Detach the attached thread state (which must have been reset with a previous call to `PyThreadState_Clear()`) and then destroy it.

No thread state will be attached upon returning.

PyFrameObject *PyThreadState_GetFrame(PyThreadState *tstate)

Part of the Stable ABI since version 3.10.
Get the current frame of the Python thread state *tstate*.

Return a strong reference. Return `NULL` if no frame is currently executing.

See also **`PyEval_GetFrame()`**.

tstate must not be `NULL`, and must be attached.

Added in version 3.9.

uint64_t PyThreadState_GetID(PyThreadState *tstate)

Part of the Stable ABI since version 3.10.
Get the unique thread state identifier of the Python thread state *tstate*.

tstate must not be `NULL`, and must be attached.

Added in version 3.9.

PyInterpreterState *PyThreadState_GetInterpreter(PyThreadState *tstate)

Part of the Stable ABI since version 3.10.
Get the interpreter of the Python thread state *tstate*.

tstate must not be `NULL`, and must be attached.

Added in version 3.9.

void PyThreadState_EnterTracing(PyThreadState *tstate)

Suspend tracing and profiling in the Python thread state *tstate*.

Resume them using the **`PyThreadState_LeaveTracing()`** function.

Added in version 3.11.

void PyThreadState_LeaveTracing(PyThreadState *tstate)

Resume tracing and profiling in the Python thread state *tstate* suspended by the **`PyThreadState_EnterTracing()`** function.

See also **`PyEval_SetTrace()`** and **`PyEval_SetProfile()`** functions.

Added in version 3.11.

int PyUnstable_ThreadState_SetStackProtection(PyThreadState *tstate, void *stack_start_addr, size_t stack_size)

This is Unstable API. It may change without warning in minor releases.

Set the stack protection start address and stack protection size of a Python thread state.

On success, return `0`. On failure, set an exception and return `-1`.

CPython implements recursion control for C code by raising **`RecursionError`** when it notices that the machine execution stack is close to overflow. See for example the **`Py_EnterRecursiveCall()`** function. For this, it needs to know the location of the current thread’s stack, which it normally gets from the operating system. When the stack is changed, for example using context switching techniques like the Boost library’s `boost::context`, you must call **`PyUnstable_ThreadState_SetStackProtection()`** to inform CPython of the change.

Call **`PyUnstable_ThreadState_SetStackProtection()`** either before or after changing the stack. Do not call any other Python C API between the call and the stack change.

See **`PyUnstable_ThreadState_ResetStackProtection()`** for undoing this operation.

Added in version 3.14.1:

Warning

This function was added in a bugfix release, and extensions that use it will be incompatible with Python 3.14.0. Most packaging tools for Python are not able to handle this incompatibility automatically, and will need explicit configuration. When using PyPA standards (wheels and source distributions), specify `Requires-Python: != 3.14.0.*` in core metadata [https://packaging.python.org/en/latest/specifications/core-metadata/#requires-python].

void PyUnstable_ThreadState_ResetStackProtection(PyThreadState *tstate)

This is Unstable API. It may change without warning in minor releases.

Reset the stack protection start address and stack protection size of a Python thread state to the operating system defaults.

See `PyUnstable_ThreadState_SetStackProtection()` for an explanation.

Added in version 3.14.1:

Warning

This function was added in a bugfix release, and extensions that use it will be incompatible with Python 3.14.0. Most packaging tools for Python are not able to handle this incompatibility automatically, and will need explicit configuration. When using PyPA standards (wheels and source distributions), specify `Requires-Python: != 3.14.0.*` in core metadata [https://packaging.python.org/en/latest/specifications/core-metadata/#requires-python].

PyInterpreterState *PyInterpreterState_Get(void)

Part of the Stable ABI since version 3.9.
Get the current interpreter.

Issue a fatal error if there no attached thread state. It cannot return NULL.

Added in version 3.9.

int64_t PyInterpreterState_GetID(PyInterpreterState *interp)

Part of the Stable ABI since version 3.7.
Return the interpreter’s unique ID. If there was any error in doing so then `-1` is returned and an error is set.

The caller must have an attached thread state.

Added in version 3.7.

PyObject *PyInterpreterState_GetDict(PyInterpreterState *interp)

Return value: Borrowed reference. Part of the Stable ABI since version 3.8.
Return a dictionary in which interpreter-specific data may be stored. If this function returns `NULL` then no exception has been raised and the caller should assume no interpreter-specific dict is available.

This is not a replacement for **`PyModule_GetState()`**, which extensions should use to store interpreter-specific state information.

The returned dictionary is borrowed from the interpreter and is valid until interpreter shutdown.

Added in version 3.8.

typedef PyObject *(*_PyFrameEvalFunction)(PyThreadState *tstate, _PyInterpreterFrame *frame, int throwflag)

Type of a frame evaluation function.

The *throwflag* parameter is used by the `throw()` method of generators: if non-zero, handle the current exception.

Changed in version 3.9: The function now takes a *tstate* parameter.

Changed in version 3.11: The *frame* parameter changed from `PyFrameObject*` to `_PyInterpreterFrame*`.

_PyFrameEvalFunction _PyInterpreterState_GetEvalFrameFunc(PyInterpreterState *interp)

Get the frame evaluation function.

See the **PEP 523** [https://peps.python.org/pep-0523/] “Adding a frame evaluation API to CPython”.

Added in version 3.9.

void _PyInterpreterState_SetEvalFrameFunc(PyInterpreterState *interp, _PyFrameEvalFunction eval_frame)

Set the frame evaluation function.

See the **PEP 523** [https://peps.python.org/pep-0523/] “Adding a frame evaluation API to CPython”.

Added in version 3.9.

PyObject *PyThreadState_GetDict()

Return value: Borrowed reference. Part of the Stable ABI.
Return a dictionary in which extensions can store thread-specific state information. Each extension should use a unique key to use to store state in the dictionary. It is okay to call this function when no thread state is attached. If this function returns `NULL`, no exception has been raised and the caller should assume no thread state is attached.

int PyThreadState_SetAsyncExc(unsigned long id, PyObject *exc)

Part of the Stable ABI.
Asynchronously raise an exception in a thread. The *id* argument is the thread id of the target thread; *exc* is the exception object to be raised. This function does not steal any references to *exc*. To prevent naive misuse, you must write your own C extension to call this. Must be called with an attached thread state. Returns the number of thread states modified; this is normally one, but will be zero if the thread id isn’t found. If *exc* is `NULL`, the pending

exception (if any) for the thread is cleared. This raises no exceptions.

Changed in version 3.7: The type of the *id* parameter changed from long to unsigned long.

void PyEval_AcquireThread(PyThreadState *tstate)

Part of the Stable ABI.
Attach *tstate* to the current thread, which must not be `NULL` or already attached.

The calling thread must not already have an attached thread state.

Note

Calling this function from a thread when the runtime is finalizing will hang the thread until the program exits, even if the thread was not created by Python. Refer to Cautions regarding runtime finalization for more details.

Changed in version 3.8: Updated to be consistent with `PyEval_RestoreThread()`, `Py_END_ALLOW_THREADS()`, and `PyGILState_Ensure()`, and terminate the current thread if called while the interpreter is finalizing.

Changed in version 3.14: Hangs the current thread, rather than terminating it, if called while the interpreter is finalizing.

`PyEval_RestoreThread()` is a higher-level function which is always available (even when threads have not been initialized).

void PyEval_ReleaseThread(PyThreadState *tstate)

Part of the Stable ABI.

Detach the attached thread state. The *tstate* argument, which must not be `NULL`, is only used to check that it represents the attached thread state — if it isn’t, a fatal error is reported.

`PyEval_SaveThread()` is a higher-level function which is always available (even when threads have not been initialized).

Sub-interpreter support

While in most uses, you will only embed a single Python interpreter, there are cases where you need to create several independent interpreters in the same process and perhaps even in the same thread. Sub-interpreters allow you to do that.

The “main” interpreter is the first one created when the runtime initializes. It is usually the only Python interpreter in a process. Unlike sub-interpreters, the main interpreter has unique process-global responsibilities like signal handling. It is also responsible for execution during runtime initialization and is usually the active interpreter during runtime finalization. The `PyInterpreterState_Main()` function returns a pointer to its state.

You can switch between sub-interpreters using the `PyThreadState_Swap()` function. You can create and destroy them using the following functions:

type PyInterpreterConfig

Structure containing most parameters to configure a sub-interpreter. Its values are used only in `Py_NewInterpreterFromConfig()` and never modified by the runtime.

Added in version 3.12.

Structure fields:

int use_main_obmalloc

If this is `0` then the sub-interpreter will use its own "object" allocator state. Otherwise it will use (share) the main interpreter's.

If this is `0` then `check_multi_interp_extensions` must be `1` (non-zero). If this is `1` then `gil` must not be `PyInterpreterConfig_OWN_GIL`.

int allow_fork

If this is `0` then the runtime will not support forking the process in any thread where the sub-interpreter is currently active. Otherwise fork is unrestricted.

Note that the `subprocess` module still works when fork is disallowed.

int allow_exec

If this is `0` then the runtime will not support replacing the current process via exec (e.g. `os.execv()`) in any thread where the sub-interpreter is currently active. Otherwise exec is unrestricted.

Note that the `subprocess` module still works when exec is disallowed.

int allow_threads

If this is `0` then the sub-interpreter's `threading` module won't create threads. Otherwise threads are allowed.

int allow_daemon_threads

If this is `0` then the sub-interpreter’s `threading` module won’t create daemon threads. Otherwise daemon threads are allowed (as long as `allow_threads` is non-zero).

int check_multi_interp_extensions

If this is `0` then all extension modules may be imported, including legacy (single-phase init) modules, in any thread where the sub-interpreter is currently active. Otherwise only multi-phase init extension modules (see **PEP 489** [https://peps.python.org/pep-0489/]) may be imported. (Also see `Py_mod_multiple_interpreters`.)

This must be `1` (non-zero) if `use_main_obmalloc` is `0`.

int gil

This determines the operation of the GIL for the sub-interpreter. It may be one of the following:

PyInterpreterConfig_DEFAULT_GIL

Use the default selection (`PyInterpreterConfig_SHARED_GIL`).

PyInterpreterConfig_SHARED_GIL

Use (share) the main interpreter’s GIL.

PyInterpreterConfig_OWN_GIL

Use the sub-interpreter’s own GIL.

If this is `PyInterpreterConfig_OWN_GIL` then `PyInterpreterConfig.use_main_obmalloc` must be `0`.

PyStatus Py_NewInterpreterFromConfig(PyThreadState **tstate_p, const PyInterpreterConfig *config)

Create a new sub-interpreter. This is an (almost) totally separate environment for the execution of Python code. In particular, the new interpreter has separate, independent versions of all imported modules, including the fundamental modules **`builtins`**, **`__main__`** and **`sys`**. The table of loaded modules (`sys.modules`) and the module search path (`sys.path`) are also separate. The new environment has no `sys.argv` variable. It has new standard I/O stream file objects `sys.stdin`, `sys.stdout` and `sys.stderr` (however these refer to the same underlying file descriptors).

The given *config* controls the options with which the interpreter is initialized.

Upon success, *tstate_p* will be set to the first thread state created in the new sub-interpreter. This thread state is attached. Note that no actual thread is created; see the discussion of thread states below. If creation of the new interpreter is unsuccessful, *tstate_p* is set to `NULL`; no exception is set since the exception state is stored in the attached thread state, which might not exist.

Like all other Python/C API functions, an attached thread state must be present before calling this function, but it might be detached upon returning. On success, the returned thread state will be attached. If the sub-interpreter is created with its own GIL then the attached thread state of the calling interpreter will be detached. When the function returns, the new interpreter's thread state will be attached to the current thread and the previous interpreter's attached thread state will remain detached.

Added in version 3.12.

Sub-interpreters are most effective when isolated from each other, with certain functionality restricted:

```
PyInterpreterConfig config = {
    .use_main_obmalloc = 0,
    .allow_fork = 0,
    .allow_exec = 0,
    .allow_threads = 1,
    .allow_daemon_threads = 0,
    .check_multi_interp_extensions = 1,
    .gil = PyInterpreterConfig_OWN_GIL,
};
PyThreadState *tstate = NULL;
PyStatus status = Py_NewInterpreterFromConfig(&tstate,
&config);
if (PyStatus_Exception(status)) {
    Py_ExitStatusException(status);
}
```

Note that the config is used only briefly and does not get modified. During initialization the config's values are converted into various `PyInterpreterState` values. A read-only copy of the config may be stored internally on the `PyInterpreterState`.

Extension modules are shared between (sub-)interpreters as follows:

- For modules using multi-phase initialization, e.g. `PyModule_FromDefAndSpec()`, a separate module object is created and initialized for each interpreter. Only C-level static and global variables are shared between these module objects.

- For modules using single-phase initialization, e.g. `PyModule_Create()`, the first time a particular extension is imported, it is initialized normally, and a (shallow) copy of its module's dictionary is squirreled away. When the same extension is imported by another (sub-)interpreter, a new module is initialized and filled with the contents of this copy; the extension's `init` function is not called. Objects in the module's dictionary thus end up shared across (sub-)interpreters, which might cause unwanted behavior (see Bugs and caveats below).

Note that this is different from what happens when an extension is imported after the interpreter has been completely re-initialized by calling `Py_FinalizeEx()` and `Py_Initialize()`; in that case, the extension's `initmodule` function *is* called again. As with multi-phase initialization, this means that only C-level static and global variables are shared between these modules.

PyThreadState *Py_NewInterpreter(void)

Part of the Stable ABI.
Create a new sub-interpreter. This is essentially just a wrapper around `Py_NewInterpreterFromConfig()` with a config that preserves the existing behavior. The result is an unisolated sub-interpreter that shares the main interpreter's GIL, allows fork/exec, allows daemon threads, and allows single-phase init modules.

void Py_EndInterpreter(PyThreadState *tstate)

Part of the Stable ABI.
Destroy the (sub-)interpreter represented by the given thread state. The given thread state must be attached. When the call returns, there will be no attached thread state. All thread states associated with this interpreter are destroyed.

`Py_FinalizeEx()` will destroy all sub-interpreters that haven't been explicitly destroyed at that point.

A Per-Interpreter GIL

Using `Py_NewInterpreterFromConfig()` you can create a sub-interpreter that is completely isolated from other interpreters, including having its own GIL. The most important benefit of this isolation is that such an interpreter can execute Python code without being blocked by other interpreters or blocking any others. Thus a single Python process can truly take advantage of multiple CPU cores when running Python code. The isolation also encourages a different approach to concurrency than that of just using threads. (See **PEP 554** [https://peps.python.org/pep-0554/] and **PEP 684** [https://peps.python.org/pep-0684/].)

Using an isolated interpreter requires vigilance in preserving that isolation. That especially means not sharing any objects or mutable state without guarantees about thread-safety. Even objects that are otherwise immutable (e.g. `None`, `(1, 5)`) can't normally be shared because of the refcount. One simple but less-efficient approach around this is to use a global lock around all use of some state (or object). Alternately, effectively immutable objects (like integers or strings) can be made safe in spite of their refcounts by making them immortal. In fact, this has been done for the builtin singletons, small integers, and a number of other builtin objects.

If you preserve isolation then you will have access to proper multi-core computing without the complications that come with free-threading. Failure to preserve isolation will expose you to the full consequences of free-threading, including races and hard-to-debug crashes.

Aside from that, one of the main challenges of using multiple isolated interpreters is how to communicate between them safely (not break isolation) and efficiently.

The runtime and stdlib do not provide any standard approach to this yet. A future stdlib module would help mitigate the effort of preserving isolation and expose effective tools for communicating (and sharing) data between interpreters.

Added in version 3.12.

Bugs and caveats

Because sub-interpreters (and the main interpreter) are part of the same process, the insulation between them isn't perfect — for example, using low-level file operations like `os.close()` they can (accidentally or maliciously) affect each other's open files. Because of the way extensions are shared between (sub-)interpreters, some extensions may not work properly; this is especially likely when using single-phase initialization or (static) global variables. It is possible to insert objects created in one sub-interpreter into a namespace of another (sub-)interpreter; this should be avoided if possible.

Special care should be taken to avoid sharing user-defined functions, methods, instances or classes between sub-interpreters, since import operations executed by such objects may affect the wrong (sub-)interpreter's dictionary of loaded modules. It is equally important to avoid sharing objects from which the above are reachable.

Also note that combining this functionality with `PyGILState_*` APIs is delicate, because these APIs assume a bijection between Python thread states and OS-level threads, an assumption broken by the presence of sub-interpreters. It is highly recommended that you don't switch sub-interpreters between a pair of matching `PyGILState_Ensure()` and `PyGILState_Release()` calls. Furthermore, extensions (such as `ctypes`) using these APIs to allow calling of Python code from non-Python created threads will probably be broken when using sub-interpreters.

Asynchronous Notifications

A mechanism is provided to make asynchronous notifications to the main interpreter thread. These notifications take the form of a function pointer and a void pointer argument.

int Py_AddPendingCall(int (*func)(void*), void *arg)

Part of the Stable ABI.

Schedule a function to be called from the main interpreter thread. On success, `0` is returned and *func* is queued for being called in the main thread. On failure, `-1` is returned without setting any exception.

When successfully queued, *func* will be *eventually* called from the main interpreter thread with the argument *arg*. It will be called asynchronously with respect to normally running Python code, but with both these conditions met:

- on a bytecode boundary;
- with the main thread holding an attached thread state (*func* can therefore use the full C API).

func must return `0` on success, or `-1` on failure with an exception set. *func* won’t be interrupted to perform another asynchronous notification recursively, but it can still be interrupted to switch threads if the thread state is detached.

This function doesn’t need an attached thread state. However, to call this function in a subinterpreter, the caller must have an attached thread state. Otherwise, the function *func* can be scheduled to be called from the wrong interpreter.

Warning

This is a low-level function, only useful for very special cases. There is no guarantee that *func* will be called as quick as possible. If the main thread is busy executing a system call, *func* won’t be called before the system call returns. This function is generally **not** suitable for calling Python code from arbitrary C threads. Instead, use the PyGILState API.

Added in version 3.1.

Changed in version 3.9: If this function is called in a subinterpreter, the function *func* is now scheduled to be called from the subinterpreter, rather than being called from the main interpreter. Each subinterpreter now has its own list of scheduled calls.

Changed in version 3.12: This function now always schedules *func* to be run in the main interpreter.

int Py_MakePendingCalls(void)

Part of the Stable ABI.

Execute all pending calls. This is usually executed automatically by the interpreter.

This function returns `0` on success, and returns `-1` with an exception set on failure.

If this is not called in the main thread of the main interpreter, this function does nothing and returns `0`. The caller must hold an attached thread state.

Added in version 3.1.

Changed in version 3.12: This function only runs pending calls in the main interpreter.

Profiling and Tracing

The Python interpreter provides some low-level support for attaching profiling and execution tracing facilities. These are used for profiling, debugging, and coverage analysis tools.

This C interface allows the profiling or tracing code to avoid the overhead of calling through Python-level callable objects, making a direct C function call instead. The essential attributes of the facility have not changed; the interface allows trace functions to be installed per-thread, and the basic events reported to the trace function are the same as had been reported to the Python-level trace functions in previous versions.

typedef int (*Py_tracefunc)(PyObject *obj, PyFrameObject *frame, int what, PyObject *arg)

The type of the trace function registered using `PyEval_SetProfile()` and `PyEval_SetTrace()`. The first parameter is the object passed to the registration function as *obj*, *frame* is the frame object to which the event pertains, *what* is one of the constants `PyTrace_CALL`, `PyTrace_EXCEPTION`, `PyTrace_LINE`, `PyTrace_RETURN`, `PyTrace_C_CALL`, `PyTrace_C_EXCEPTION`, `PyTrace_C_RETURN`, or `PyTrace_OPCODE`, and *arg* depends on the value of *what*:

Value of *what*	**Meaning of *arg***
`PyTrace_CALL`	Always `Py_None`.

Value of *what*	**Meaning of *arg***
`PyTrace_EXCEPTION`	Exception information as returned by `sys.exc_info()`.
`PyTrace_LINE`	Always `Py_None`.
`PyTrace_RETURN`	Value being returned to the caller, or `NULL` if caused by an exception.
`PyTrace_C_CALL`	Function object being called.
`PyTrace_C_EXCEPTION`	Function object being called.
`PyTrace_C_RETURN`	Function object being called.
`PyTrace_OPCODE`	Always `Py_None`.

int PyTrace_CALL

The value of the *what* parameter to a `Py_tracefunc` function when a new call to a function or method is being reported, or a new entry into a generator. Note that the creation of the iterator for a generator function is not reported as there is no control transfer to the Python bytecode in the corresponding frame.

int PyTrace_EXCEPTION

The value of the *what* parameter to a `Py_tracefunc` function when an exception has been raised. The callback function is called with this value for *what* when after any bytecode is processed after which the exception becomes set within the frame being executed. The effect of this is that as exception propagation causes the Python stack to unwind, the callback is called upon return to each frame as the exception propagates. Only trace functions receives these events; they are not needed by the profiler.

int PyTrace_LINE

The value passed as the *what* parameter to a `Py_tracefunc` function (but not a profiling function) when a line-number event is being reported. It may be disabled for a frame by setting `f_trace_lines` to *0* on that frame.

int PyTrace_RETURN

The value for the *what* parameter to `Py_tracefunc` functions when a call is about to return.

int PyTrace_C_CALL

The value for the *what* parameter to `Py_tracefunc` functions when a C function is about to be called.

int PyTrace_C_EXCEPTION

The value for the *what* parameter to `Py_tracefunc` functions when a C function has raised an exception.

int PyTrace_C_RETURN

The value for the *what* parameter to `Py_tracefunc` functions when a C function has returned.

int PyTrace_OPCODE

The value for the *what* parameter to `Py_tracefunc` functions (but not profiling functions) when a new opcode is about to be executed. This event is not emitted by default: it must be explicitly requested by setting `f_trace_opcodes` to *1* on the frame.

void PyEval_SetProfile(Py_tracefunc func, PyObject *obj)

Set the profiler function to *func*. The *obj* parameter is passed to the function as its first parameter, and may be any Python object, or `NULL`. If the profile function needs to maintain state, using a different value for *obj* for each thread provides a convenient and thread-safe place to store it. The profile function is called for all monitored events except `PyTrace_LINE` `PyTrace_OPCODE` and `PyTrace_EXCEPTION`.

See also the `sys.setprofile()` function.

The caller must have an attached thread state.

void PyEval_SetProfileAllThreads(Py_tracefunc func, PyObject *obj)

Like `PyEval_SetProfile()` but sets the profile function in all running threads belonging to the current interpreter instead of the setting it only on the current thread.

The caller must have an attached thread state.

As `PyEval_SetProfile()`, this function ignores any exceptions raised while setting the profile functions in all threads.

Added in version 3.12.

void PyEval_SetTrace(Py_tracefunc func, PyObject *obj)

Set the tracing function to *func*. This is similar to `PyEval_SetProfile()`, except the tracing function does receive line-number events and per-opcode events, but does not receive any event related to C function objects being called. Any trace function registered using `PyEval_SetTrace()` will not receive `PyTrace_C_CALL`, `PyTrace_C_EXCEPTION` or `PyTrace_C_RETURN` as a value for the *what* parameter.

See also the `sys.settrace()` function.

The caller must have an attached thread state.

void PyEval_SetTraceAllThreads(Py_tracefunc func, PyObject *obj)

Like **`PyEval_SetTrace()`** but sets the tracing function in all running threads belonging to the current interpreter instead of the setting it only on the current thread.

The caller must have an attached thread state.

As **`PyEval_SetTrace()`**, this function ignores any exceptions raised while setting the trace functions in all threads.

Added in version 3.12.

Reference tracing

Added in version 3.13.

typedef int (*PyRefTracer)(PyObject*, int event, void *data)

The type of the trace function registered using **`PyRefTracer_SetTracer()`**. The first parameter is a Python object that has been just created (when **event** is set to **`PyRefTracer_CREATE`**) or about to be destroyed (when **event** is set to **`PyRefTracer_DESTROY`**). The **data** argument is the opaque pointer that was provided when **`PyRefTracer_SetTracer()`** was called.

Added in version 3.13.

int PyRefTracer_CREATE

The value for the *event* parameter to **`PyRefTracer`** functions when a Python object has been created.

int PyRefTracer_DESTROY

The value for the *event* parameter to **`PyRefTracer`** functions when a Python object has been destroyed.

int PyRefTracer_SetTracer(PyRefTracer tracer, void *data)

Register a reference tracer function. The function will be called when a new Python has been created or when an object is going to be destroyed. If **data** is provided it must be an opaque pointer that will be provided when the tracer function is called. Return `0` on success. Set an exception and return `-1` on error.

Not that tracer functions **must not** create Python objects inside or otherwise the call will be re-entrant. The tracer also **must not** clear any existing exception or set an exception. A thread state will be active every time the tracer function is called.

There must be an attached thread state when calling this function.

Added in version 3.13.

PyRefTracer PyRefTracer_GetTracer(void **data)

Get the registered reference tracer function and the value of the opaque data pointer that was registered when **`PyRefTracer_SetTracer()`** was called. If no tracer was registered this function will return NULL and will set the **data** pointer to NULL.

There must be an attached thread state when calling this function.

Added in version 3.13.

Advanced Debugger Support

These functions are only intended to be used by advanced debugging tools.

PyInterpreterState *PyInterpreterState_Head()

Return the interpreter state object at the head of the list of all such objects.

PyInterpreterState *PyInterpreterState_Main()

Return the main interpreter state object.

PyInterpreterState *PyInterpreterState_Next(PyInterpreterState *interp)

Return the next interpreter state object after *interp* from the list of all such objects.

PyThreadState *PyInterpreterState_ThreadHead(PyInterpreterState *interp)

Return the pointer to the first **`PyThreadState`** object in the list of threads associated with the interpreter *interp*.

PyThreadState *PyThreadState_Next(PyThreadState *tstate)

Return the next thread state object after *tstate* from the list of all such objects belonging to the same **`PyInterpreterState`** object.

Thread Local Storage Support

The Python interpreter provides low-level support for thread-local storage (TLS) which wraps the underlying native TLS implementation to support the Python-level thread local storage API (**`threading.local`**). The CPython C level APIs are similar to those offered by pthreads and Windows: use a thread key and functions to associate a void* value per thread.

A thread state does *not* need to be attached when calling these functions; they suppl their own locking.

Note that `Python.h` does not include the declaration of the TLS APIs, you need to include `pythread.h` to use thread-local storage.

Note

None of these API functions handle memory management on behalf of the void* values. You need to allocate and deallocate them yourself. If the void* values happen to be PyObject*, these functions don't do refcount operations on them either.

Thread Specific Storage (TSS) API

TSS API is introduced to supersede the use of the existing TLS API within the CPython interpreter. This API uses a new type **`Py_tss_t`** instead of int to represent thread keys.

Added in version 3.7.

See also

"A New C-API for Thread-Local Storage in CPython" (**PEP 539** [https://peps.python.org/pep-0539/])

type Py_tss_t

This data structure represents the state of a thread key, the definition of which may depend on the underlying TLS implementation, and it has an internal field representing the key's initialization state. There are no public members in this structure.

When Py_LIMITED_API is not defined, static allocation of this type by `Py_tss_NEEDS_INIT` is allowed.

Py_tss_NEEDS_INIT

This macro expands to the initializer for `Py_tss_t` variables. Note that this macro won’t be defined with Py_LIMITED_API.

Dynamic Allocation

Dynamic allocation of the `Py_tss_t`, required in extension modules built with Py_LIMITED_API, where static allocation of this type is not possible due to its implementation being opaque at build time.

Py_tss_t *PyThread_tss_alloc()

Part of the Stable ABI since version 3.7.
Return a value which is the same state as a value initialized with `Py_tss_NEEDS_INIT`, or `NULL` in the case of dynamic allocation failure.

void PyThread_tss_free(Py_tss_t *key)

Part of the Stable ABI since version 3.7.
Free the given *key* allocated by `PyThread_tss_alloc()`, after first calling `PyThread_tss_delete()` to ensure any associated thread locals have been unassigned. This is a no-op if the *key* argument is `NULL`.

Note

A freed key becomes a dangling pointer. You should reset the key to `NULL`.

Methods

The parameter *key* of these functions must not be `NULL`. Moreover, the behaviors of **`PyThread_tss_set()`** and **`PyThread_tss_get()`** are undefined if the given **`Py_tss_t`** has not been initialized by **`PyThread_tss_create()`**.

int PyThread_tss_is_created(Py_tss_t *key)

Part of the Stable ABI since version 3.7.
Return a non-zero value if the given **`Py_tss_t`** has been initialized by **`PyThread_tss_create()`**.

int PyThread_tss_create(Py_tss_t *key)

Part of the Stable ABI since version 3.7.
Return a zero value on successful initialization of a TSS key. The behavior is undefined if the value pointed to by the *key* argument is not initialized by **`Py_tss_NEEDS_INIT`**. This function can be called repeatedly on the same key – calling it on an already initialized key is a no-op and immediately returns success.

void PyThread_tss_delete(Py_tss_t *key)

Part of the Stable ABI since version 3.7.
Destroy a TSS key to forget the values associated with the key across all threads, and change the key's initialization state to uninitialized. A destroyed key is able to be initialized again by **`PyThread_tss_create()`**. This function can be called repeatedly on the same key – calling it on an already destroyed key is a no-op.

int PyThread_tss_set(Py_tss_t *key, void *value)

Part of the Stable ABI since version 3.7.
Return a zero value to indicate successfully associating a void* value with a TSS key in the current thread. Each thread has a distinct mapping of the key to a void* value.

void *PyThread_tss_get(Py_tss_t *key)

Part of the Stable ABI since version 3.7.
Return the void* value associated with a TSS key in the current thread. This returns `NULL` if no value is associated with the key in the current thread.

Thread Local Storage (TLS) API

Deprecated since version 3.7: This API is superseded by Thread Specific Storage (TSS) API.

Note

This version of the API does not support platforms where the native TLS key is defined in a way that cannot be safely cast to `int`. On such platforms, **`PyThread_create_key()`** will return immediately with a failure status, and the other TLS functions will all be no-ops on such platforms.

Due to the compatibility problem noted above, this version of the API should not be used in new code.

int PyThread_create_key()
Part of the Stable ABI.

void PyThread_delete_key(int key)
Part of the Stable ABI.

int PyThread_set_key_value(int key, void *value)
Part of the Stable ABI.

void *PyThread_get_key_value(int key)
Part of the Stable ABI.

void PyThread_delete_key_value(int key)

Part of the Stable ABI.

void PyThread_ReInitTLS()

Part of the Stable ABI.

Synchronization Primitives

The C-API provides a basic mutual exclusion lock.

type PyMutex

A mutual exclusion lock. The **`PyMutex`** should be initialized to zero to represent the unlocked state. For example:

```
PyMutex mutex = {0};
```

Instances of **`PyMutex`** should not be copied or moved. Both the contents and address of a **`PyMutex`** are meaningful, and it must remain at a fixed, writable location in memory.

Note

A **`PyMutex`** currently occupies one byte, but the size should be considered unstable. The size may change in future Python releases without a deprecation period.

Added in version 3.13.

void PyMutex_Lock(PyMutex *m)

Lock mutex *m*. If another thread has already locked it, the calling thread will block until the mutex is unlocked. While blocked, the thread will temporarily detach the thread state if one exists.

Added in version 3.13.

void PyMutex_Unlock(PyMutex *m)

Unlock mutex *m*. The mutex must be locked — otherwise, the function will issue a fatal error.

Added in version 3.13.

int PyMutex_IsLocked(PyMutex *m)

Returns non-zero if the mutex *m* is currently locked, zero otherwise.

Note

This function is intended for use in assertions and debugging only and should not be used to make concurrency control decisions, as the lock state may change immediately after the check.

Added in version 3.14.

Python Critical Section API

The critical section API provides a deadlock avoidance layer on top of per-object locks for free-threaded CPython. They are intended to replace reliance on the global interpreter lock, and are no-ops in versions of Python with the global interpreter lock.

Critical sections are intended to be used for custom types implemented in C-API extensions. They should generally not be used with built-in types like `list` and `dict` because their public C-APIs already use critical sections internally, with the notable exception of `PyDict_Next()`, which requires critical section to be acquired externally.

Critical sections avoid deadlocks by implicitly suspending active critical sections, hence, they do not provide exclusive access such as provided by traditional locks like `PyMutex`. When a critical section is started, the per-object lock for the object is acquired. If the code executed inside the critical section calls C-API functions then it can suspend the critical section thereby releasing the per-object lock, so other threads can acquire the per-object lock for the same object.

Variants that accept `PyMutex` pointers rather than Python objects are also available. Use these variants to start a critical section in a situation where there is no `PyObject` – for example, when working with a C type that does not extend or wrap `PyObject` but still needs to call into the C API in a manner that might lead to deadlocks.

The functions and structs used by the macros are exposed for cases where C macros are not available. They should only be used as in the given macro expansions. Note that the sizes and contents of the structures may change in future Python versions.

Note

Operations that need to lock two objects at once must use `Py_BEGIN_CRITICAL_SECTION2`. You *cannot* use nested critical sections to lock more than one object at once, because the inner critical section may suspend the outer critical sections. This API does not provide a way to lock more than two objects at once.

Example usage:

```
static PyObject *
set_field(MyObject *self, PyObject *value)
{
   Py_BEGIN_CRITICAL_SECTION(self);
   Py_SETREF(self->field, Py_XNewRef(value));
```

```
    Py_END_CRITICAL_SECTION();
    Py_RETURN_NONE;
}
```

In the above example, **`Py_SETREF`** calls **`Py_DECREF`**, which can call arbitrary code through an object's deallocation function. The critical section API avoids potential deadlocks due to reentrancy and lock ordering by allowing the runtime to temporarily suspend the critical section if the code triggered by the finalizer blocks and calls **`PyEval_SaveThread()`**.

Py_BEGIN_CRITICAL_SECTION(op)

Acquires the per-object lock for the object *op* and begins a critical section.

In the free-threaded build, this macro expands to:

```
{
    PyCriticalSection _py_cs;
    PyCriticalSection_Begin(&_py_cs, (PyObject*)(op))
```

In the default build, this macro expands to `{`.

Added in version 3.13.

Py_BEGIN_CRITICAL_SECTION_MUTEX(m)

Locks the mutex *m* and begins a critical section.

In the free-threaded build, this macro expands to:

```
{
     PyCriticalSection _py_cs;
     PyCriticalSection_BeginMutex(&_py_cs, m)
```

Note that unlike **`Py_BEGIN_CRITICAL_SECTION`**, there is no cast for the argument of the macro - it must be a **`PyMutex`** pointer.

On the default build, this macro expands to `{`.

Added in version 3.14.

Py_END_CRITICAL_SECTION()

Ends the critical section and releases the per-object lock.

In the free-threaded build, this macro expands to:

```
    PyCriticalSection_End(&_py_cs);
}
```

In the default build, this macro expands to `}`.

Added in version 3.13.

Py_BEGIN_CRITICAL_SECTION2(a, b)

Acquires the per-objects locks for the objects *a* and *b* and begins a critical section. The locks are acquired in a consistent order (lowest address first) to avoid lock ordering deadlocks.

In the free-threaded build, this macro expands to:

```
{
    PyCriticalSection2 _py_cs2;
    PyCriticalSection2_Begin(&_py_cs2, (PyObject*)(a), 
(PyObject*)(b))
```

In the default build, this macro expands to `{`.

Added in version 3.13.

Py_BEGIN_CRITICAL_SECTION2_MUTEX(m1, m2)

Locks the mutexes *m1* and *m2* and begins a critical section.

In the free-threaded build, this macro expands to:

```
{
     PyCriticalSection2 _py_cs2;
```

```
    PyCriticalSection2_BeginMutex(&_py_cs2, m1, m2)
```

Note that unlike `Py_BEGIN_CRITICAL_SECTION2`, there is no cast for the arguments of the macro - they must be `PyMutex` pointers.

On the default build, this macro expands to `{`.

Added in version 3.14.

Py_END_CRITICAL_SECTION2()

Ends the critical section and releases the per-object locks.

In the free-threaded build, this macro expands to:

```
    PyCriticalSection2_End(&_py_cs2);
}
```

In the default build, this macro expands to `}`.

Added in version 3.13.

Legacy Locking APIs

These APIs are obsolete since Python 3.13 with the introduction of `PyMutex`.

Changed in version 3.15: These APIs are now a simple wrapper around `PyMutex`.

type PyThread_type_lock

A pointer to a mutual exclusion lock.

type PyLockStatus

The result of acquiring a lock with a timeout.

enumerator PY_LOCK_FAILURE

Failed to acquire the lock.

enumerator PY_LOCK_ACQUIRED

The lock was successfully acquired.

enumerator PY_LOCK_INTR

The lock was interrupted by a signal.

PyThread_type_lock PyThread_allocate_lock(void)

Part of the Stable ABI.

Allocate a new lock.

On success, this function returns a lock; on failure, this function returns `0` without an exception set.

The caller does not need to hold an attached thread state.

Changed in version 3.15: This function now always uses **`PyMutex`**. In prior versions, this would use a lock provided by the operating system.

void PyThread_free_lock(PyThread_type_lock lock)

Part of the Stable ABI.

Destroy *lock*. The lock should not be held by any thread when calling this.

The caller does not need to hold an attached thread state.

PyLockStatus PyThread_acquire_lock_timed(PyThread_type_lock lock, long long microseconds, int intr_flag)

Part of the Stable ABI.

Acquire *lock* with a timeout.

This will wait for *microseconds* microseconds to acquire the lock. If the timeout expires, this function returns **`PY_LOCK_FAILURE`**. If *microseconds* is `-1`, this will wait indefinitely until the lock has been released.

If *intr_flag* is `1`, acquiring the lock may be interrupted by a signal, in which case this function returns **`PY_LOCK_INTR`**. Upon interruption, it's generally expected that the caller makes a call to **`Py_MakePendingCalls()`** to propagate an exception to Python code.

If the lock is successfully acquired, this function returns **`PY_LOCK_ACQUIRED`**.

The caller does not need to hold an attached thread state.

int PyThread_acquire_lock(PyThread_type_lock lock, int waitflag)

Part of the Stable ABI.
Acquire *lock*.

If *waitflag* is `1` and another thread currently holds the lock, this function will wait until the lock can be acquired and will always return `1`.

If *waitflag* is `0` and another thread holds the lock, this function will not wait and instead return `0`. If the lock is not held by any other thread, then this function will acquire it and return `1`.

Unlike **`PyThread_acquire_lock_timed()`**, acquiring the lock cannot be interrupted by a signal.

The caller does not need to hold an attached thread state.

int PyThread_release_lock(PyThread_type_lock lock)

Part of the Stable ABI.

Release *lock*. If *lock* is not held, then this function issues a fatal error.

The caller does not need to hold an attached thread state.

Operating System Thread APIs

PYTHREAD_INVALID_THREAD_ID

Sentinel value for an invalid thread ID.

This is currently equivalent to `(unsigned long)-1`.

unsigned long PyThread_start_new_thread(void (*func)(void*), void *arg)

Part of the Stable ABI.

Start function *func* in a new thread with argument *arg*. The resulting thread is not intended to be joined.

func must not be `NULL`, but *arg* may be `NULL`.

On success, this function returns the identifier of the new thread; on failure, this returns **`PYTHREAD_INVALID_THREAD_ID`**.

The caller does not need to hold an attached thread state.

unsigned long PyThread_get_thread_ident(void)

Part of the Stable ABI.

Return the identifier of the current thread, which will never be zero.

This function cannot fail, and the caller does not need to hold an attached thread state.

See also

`threading.get_ident()`

PyObject *PyThread_GetInfo(void)

Part of the Stable ABI since version 3.3.
Get general information about the current thread in the form of a struct sequence object. This information is accessible as `sys.thread_info` in Python.

On success, this returns a new strong reference to the thread information; on failure, this returns `NULL` with an exception set.

The caller must hold an attached thread state.

PY_HAVE_THREAD_NATIVE_ID

This macro is defined when the system supports native thread IDs.

unsigned long PyThread_get_thread_native_id(void)

Part of the Stable ABI on platforms with native thread IDs.
Get the native identifier of the current thread as it was assigned by the operating system’s kernel, which will never be less than zero.

This function is only available when `PY_HAVE_THREAD_NATIVE_ID` is defined.

This function cannot fail, and the caller does not need to hold an attached thread state.

See also

`threading.get_native_id()`

void PyThread_exit_thread(void)

Part of the Stable ABI.

Terminate the current thread. This function is generally considered unsafe and should be avoided. It is kept solely for backwards compatibility.

This function is only safe to call if all functions in the full call stack are written to safely allow it.

Warning

If the current system uses POSIX threads (also known as “pthreads”), this calls *pthread_exit(3)* *[https://manpages.debian.org/pthread_exit(3)]*, which attempts to unwind the stack and call C++ destructors on some libc implementations. However, if a `noexcept` function is reached, it may terminate the process. Other systems, such as macOS, do unwinding.

On Windows, this function calls `_endthreadex()`, which kills the thread without calling C++ destructors.

In any case, there is a risk of corruption on the thread’s stack.

Deprecated since version 3.14.

void PyThread_init_thread(void)

Part of the Stable ABI.

Initialize `PyThread*` APIs. Python executes this function automatically, so there’s little need to call it from an extension module.

int PyThread_set_stacksize(size_t size)

Part of the Stable ABI.

Set the stack size of the current thread to *size* bytes.

This function returns `0` on success, `-1` if *size* is invalid, or `-2` if the system does not support changing the stack size. This function does not set exceptions.

The caller does not need to hold an attached thread state.

size_t PyThread_get_stacksize(void)

Part of the Stable ABI.

Return the stack size of the current thread in bytes, or `0` if the system’s default stack size is in use.

The caller does not need to hold an attached thread state.

Python Initialization Configuration

PyInitConfig C API

Added in version 3.14.

Python can be initialized with `Py_InitializeFromInitConfig()`.

The `Py_RunMain()` function can be used to write a customized Python program.

See also Initialization, Finalization, and Threads.

See also

PEP 741 [https://peps.python.org/pep-0741/] “Python Configuration C API”.

Example

Example of customized Python always running with the Python Development Mode enabled; return `-1` on error:

```
int init_python(void)
{
    PyInitConfig *config = PyInitConfig_Create();
    if (config == NULL) {
        printf("PYTHON INIT ERROR: memory allocation failed\n");
        return -1;
    }

    // Enable the Python Development Mode
    if (PyInitConfig_SetInt(config, "dev_mode", 1) < 0) {
        goto error;
    }

    // Initialize Python with the configuration
    if (Py_InitializeFromInitConfig(config) < 0) {
        goto error;
    }
    PyInitConfig_Free(config);
    return 0;
error:
    {
        // Display the error message.
        //
        // This uncommon braces style is used, because you cannot make
        // goto targets point to variable declarations.
        const char *err_msg;
        (void)PyInitConfig_GetError(config, &err_msg);
        printf("PYTHON INIT ERROR: %s\n", err_msg);
        PyInitConfig_Free(config);
        return -1;
    }
}
```

Create Config

struct PyInitConfig

Opaque structure to configure the Python initialization.

PyInitConfig *PyInitConfig_Create(void)

Create a new initialization configuration using Isolated Configuration default values.

It must be freed by **`PyInitConfig_Free()`**.

Return `NULL` on memory allocation failure.

void PyInitConfig_Free(PyInitConfig *config)

Free memory of the initialization configuration *config*.

If *config* is `NULL`, no operation is performed.

Error Handling

int PyInitConfig_GetError(PyInitConfig *config, const char **err_msg)

Get the *config* error message.

- Set **err_msg* and return `1` if an error is set.
- Set **err_msg* to `NULL` and return `0` otherwise.

An error message is a UTF-8 encoded string.

If *config* has an exit code, format the exit code as an error message.

The error message remains valid until another `PyInitConfig` function is called with *config*. The caller doesn’t have to free the error message.

int PyInitConfig_GetExitCode(PyInitConfig *config, int *exitcode)

Get the *config* exit code.

- Set **exitcode* and return `1` if *config* has an exit code set.
- Return `0` if *config* has no exit code set.

Only the `Py_InitializeFromInitConfig()` function can set an exit code if the `parse_argv` option is non-zero.

An exit code can be set when parsing the command line failed (exit code `2`) or when a command line option asks to display the command line help (exit code `0`).

Get Options

The configuration option *name* parameter must be a non-NULL null-terminated UTF-8 encoded string. See Configuration Options.

int PyInitConfig_HasOption(PyInitConfig *config, const char *name)

Test if the configuration has an option called *name*.

Return `1` if the option exists, or return `0` otherwise.

int PyInitConfig_GetInt(PyInitConfig *config, const char *name, int64_t *value)

Get an integer configuration option.

- Set **value*, and return `0` on success.
- Set an error in *config* and return `-1` on error.

int PyInitConfig_GetStr(PyInitConfig *config, const char *name, char **value)

Get a string configuration option as a null-terminated UTF-8 encoded string.

- Set **value*, and return `0` on success.
- Set an error in *config* and return `-1` on error.

**value* can be set to `NULL` if the option is an optional string and the option is unset.

On success, the string must be released with `free(value)` if it’s not `NULL`.

int PyInitConfig_GetStrList(PyInitConfig *config, const char *name, size_t *length, char ***items)

Get a string list configuration option as an array of null-terminated UTF-8 encoded strings.

- Set **length* and **value*, and return `0` on success.
- Set an error in *config* and return `-1` on error.

On success, the string list must be released with `PyInitConfig_FreeStrList(length, items)`.

void PyInitConfig_FreeStrList(size_t length, char **items)

Free memory of a string list created by `PyInitConfig_GetStrList()`.

Set Options

The configuration option *name* parameter must be a non-NULL null-terminated UTF-8 encoded string. See Configuration Options.

Some configuration options have side effects on other options. This logic is only implemented when `Py_InitializeFromInitConfig()` is called, not by the “Set” functions below. For example, setting `dev_mode` to `1` does not set `faulthandler` to `1`.

int PyInitConfig_SetInt(PyInitConfig *config, const char *name, int64_t value)

Set an integer configuration option.

- Return `0` on success.
- Set an error in *config* and return `-1` on error.

int PyInitConfig_SetStr(PyInitConfig *config, const char *name, const char *value)

Set a string configuration option from a null-terminated UTF-8 encoded string. The string is copied.

- Return `0` on success.
- Set an error in *config* and return `-1` on error.

int PyInitConfig_SetStrList(PyInitConfig *config, const char *name, size_t length, char *const *items)

Set a string list configuration option from an array of null-terminated UTF-8 encoded strings. The string list is copied.

- Return `0` on success.
- Set an error in *config* and return `-1` on error.

Module

int PyInitConfig_AddModule(PyInitConfig *config, const char *name, PyObject *(*initfunc)(void))

Add a built-in extension module to the table of built-in modules.

The new module can be imported by the name *name*, and uses the function *initfunc* as the initialization function called on the first attempted import.

- Return `0` on success.
- Set an error in *config* and return `-1` on error.

If Python is initialized multiple times, `PyInitConfig_AddModule()` must be called at each Python initialization.

Similar to the **`PyImport_AppendInittab()`** function.

Initialize Python

int Py_InitializeFromInitConfig(PyInitConfig *config)

Initialize Python from the initialization configuration.

- Return `0` on success.
- Set an error in *config* and return `-1` on error.
- Set an exit code in *config* and return `-1` if Python wants to exit.

See `PyInitConfig_GetExitcode()` for the exit code case.

Configuration Options

Option	PyConfig/PyPreConfig member	Type	Visibility
`"allocator"`	**`allocator`**	`int`	Read-only
`"argv"`	**`argv`**	`list[str]`	Public
`"base_exec_prefix"`	**`base_exec_prefix`**	`str`	Public

Option	PyConfig/PyPreConfig member	Type	Visibility
`"base_executable"`	`base_executable`	`str`	Public
`"base_prefix"`	`base_prefix`	`str`	Public
`"buffered_stdio"`	`buffered_stdio`	`bool`	Read-only
`"bytes_warning"`	`bytes_warning`	`int`	Public
`"check_hash_pycs_mode"`	`check_hash_pycs_mode`	`str`	Read-only
`"code_debug_ranges"`	`code_debug_ranges`	`bool`	Read-only
`"coerce_c_locale"`	`coerce_c_locale`	`bool`	Read-only
`"coerce_c_locale_warn"`	`coerce_c_locale_warn`	`bool`	Read-only
`"configure_c_stdio"`	`configure_c_stdio`	`bool`	Read-only
`"configure_locale"`	`configure_locale`	`bool`	Read-only
`"cpu_count"`	`cpu_count`	`int`	Public

Option	PyConfig/PyPreConfig member	Type	Visibility
`"dev_mode"`	`dev_mode`	`bool`	Read-only
`"dump_refs"`	`dump_refs`	`bool`	Read-only
`"dump_refs_file"`	`dump_refs_file`	`str`	Read-only
`"exec_prefix"`	`exec_prefix`	`str`	Public
`"executable"`	`executable`	`str`	Public
`"faulthandler"`	`faulthandler`	`bool`	Read-only
`"filesystem_encoding"`	`filesystem_encoding`	`str`	Read-only
`"filesystem_errors"`	`filesystem_errors`	`str`	Read-only
`"hash_seed"`	`hash_seed`	`int`	Read-only
`"home"`	`home`	`str`	Read-only
`"import_time"`	`import_time`	`int`	Read-only

Option	PyConfig/PyPreConfig member	Type	Visibility
`"inspect"`	`inspect`	`bool`	Public
`"install_signal_handlers"`	`install_signal_handlers`	`bool`	Read-only
`"int_max_str_digits"`	`int_max_str_digits`	`int`	Public
`"interactive"`	`interactive`	`bool`	Public
`"isolated"`	`isolated`	`bool`	Read-only
`"legacy_windows_fs_encoding"`	`legacy_windows_fs_encoding`	`bool`	Read-only
`"legacy_windows_stdio"`	`legacy_windows_stdio`	`bool`	Read-only
`"malloc_stats"`	`malloc_stats`	`bool`	Read-only
`"module_search_paths"`	`module_search_paths`	`list[str]`	Public
`"optimization_level"`	`optimization_level`	`int`	Public
`"orig_argv"`	`orig_argv`	`list[str]`	Read-only
`"parse_argv"`	`parse_argv`	`bool`	Read-only

Option	PyConfig/PyPreConfig member	Type	Visibility
`"parser_debug"`	`parser_debug`	`bool`	Public
`"pathconfig_warnings"`	`pathconfig_warnings`	`bool`	Read-only
`"perf_profiling"`	`perf_profiling`	`bool`	Read-only
`"platlibdir"`	`platlibdir`	`str`	Public
`"prefix"`	`prefix`	`str`	Public
`"program_name"`	`program_name`	`str`	Read-only
`"pycache_prefix"`	`pycache_prefix`	`str`	Public
`"quiet"`	`quiet`	`bool`	Public
`"run_command"`	`run_command`	`str`	Read-only
`"run_filename"`	`run_filename`	`str`	Read-only
`"run_module"`	`run_module`	`str`	Read-only
`"run_presite"`	`run_presite`	`str`	Read-only

Option	**PyConfig/PyPreConfig member**	**Type**	**Visibility**
`"safe_path"`	`safe_path`	`bool`	Read-only
`"show_ref_count"`	`show_ref_count`	`bool`	Read-only
`"site_import"`	`site_import`	`bool`	Read-only
`"skip_source_first_line"`	`skip_source_first_line`	`bool`	Read-only
`"stdio_encoding"`	`stdio_encoding`	`str`	Read-only
`"stdio_errors"`	`stdio_errors`	`str`	Read-only
`"stdlib_dir"`	`stdlib_dir`	`str`	Public
`"tracemalloc"`	`tracemalloc`	`int`	Read-only
`"use_environment"`	`use_environment`	`bool`	Public
`"use_frozen_modules"`	`use_frozen_modules`	`bool`	Read-only
`"use_hash_seed"`	`use_hash_seed`	`bool`	Read-only

Option	PyConfig/PyPreConfig member	Type	Visibility
`"use_system_logger"`	`use_system_logger`	`bool`	Read-only
`"user_site_directory"`	`user_site_directory`	`bool`	Read-only
`"utf8_mode"`	`utf8_mode`	`bool`	Read-only
`"verbose"`	`verbose`	`int`	Public
`"warn_default_encoding"`	`warn_default_encoding`	`bool`	Read-only
`"warnoptions"`	`warnoptions`	`list[str]`	Public
`"write_bytecode"`	`write_bytecode`	`bool`	Public
`"xoptions"`	`xoptions`	`dict[str, str]`	Public
`"_pystats"`	`_pystats`	`bool`	Read-only

Visibility:

- Public: Can by get by `PyConfig_Get()` and set by `PyConfig_Set()`.
- Read-only: Can by get by `PyConfig_Get()`, but cannot be set by `PyConfig_Set()`.

Runtime Python configuration API

At runtime, it’s possible to get and set configuration options using **`PyConfig_Get()`** and **`PyConfig_Set()`** functions.

The configuration option *name* parameter must be a non-NULL null-terminated UTF-8 encoded string. See Configuration Options.

Some options are read from the **`sys`** attributes. For example, the option `"argv"` is read from **`sys.argv`**.

PyObject *PyConfig_Get(const char *name)

Get the current runtime value of a configuration option as a Python object.

- Return a new reference on success.
- Set an exception and return `NULL` on error.

The object type depends on the configuration option. It can be:

- `bool`
- `int`
- `str`
- `list[str]`
- `dict[str, str]`

The caller must have an attached thread state. The function cannot be called before Python initialization nor after Python finalization.

Added in version 3.14.

int PyConfig_GetInt(const char *name, int *value)

Similar to **`PyConfig_Get()`**, but get the value as a C int.

- Return `0` on success.
- Set an exception and return `-1` on error.

Added in version 3.14.

PyObject *PyConfig_Names(void)

Get all configuration option names as a `frozenset`.

- Return a new reference on success.
- Set an exception and return `NULL` on error.

The caller must have an attached thread state. The function cannot be called before Python initialization nor after Python finalization.

Added in version 3.14.

int PyConfig_Set(const char *name, PyObject *value)

Set the current runtime value of a configuration option.

- Raise a `ValueError` if there is no option *name*.
- Raise a `ValueError` if *value* is an invalid value.
- Raise a `ValueError` if the option is read-only (cannot be set).
- Raise a `TypeError` if *value* has not the proper type.

The caller must have an attached thread state. The function cannot be called before Python initialization nor after Python finalization.

Raises an auditing event `cpython.PyConfig_Set` with arguments `name`, `value`.

Added in version 3.14.

PyConfig C API

Added in version 3.8.

Python can be initialized with `Py_InitializeFromConfig()` and the `PyConfig` structure. It can be preinitialized with `Py_PreInitialize()` and the `PyPreConfig` structure.

There are two kinds of configuration:

- The Python Configuration can be used to build a customized Python which behaves as the regular Python. For example, environment variables and command line arguments are used to configure Python.
- The Isolated Configuration can be used to embed Python into an application. It isolates Python from the system. For example, environment variables are ignored, the LC_CTYPE locale is left unchanged and no signal handler is registered.

The `Py_RunMain()` function can be used to write a customized Python program.

See also Initialization, Finalization, and Threads.

See also

PEP 587 [https://peps.python.org/pep-0587/] “Python Initialization Configuration”.

Example

Example of customized Python always running in isolated mode:

```
int main(int argc, char **argv)
{
    PyStatus status;

    PyConfig config;
    PyConfig_InitPythonConfig(&config);
    config.isolated = 1;

    /* Decode command line arguments.
       Implicitly preinitialize Python (in isolated mode). */
    status = PyConfig_SetBytesArgv(&config, argc, argv);
    if (PyStatus_Exception(status)) {
        goto exception;
    }

    status = Py_InitializeFromConfig(&config);
    if (PyStatus_Exception(status)) {
        goto exception;
    }
    PyConfig_Clear(&config);

    return Py_RunMain();

exception:
    PyConfig_Clear(&config);
    if (PyStatus_IsExit(status)) {
        return status.exitcode;
    }
    /* Display the error message and exit the process with
       non-zero exit code */
    Py_ExitStatusException(status);
}
```

PyWideStringList

type PyWideStringList

List of `wchar_t*` strings.

If *length* is non-zero, *items* must be non-`NULL` and all strings must be non-`NULL`.

Methods:

PyStatus PyWideStringList_Append(PyWideStringList *list, const wchar_t *item)

Append *item* to *list*.

Python must be preinitialized to call this function.

PyStatus PyWideStringList_Insert(PyWideStringList *list, Py_ssize_t index, const wchar_t *item)

Insert *item* into *list* at *index*.

If *index* is greater than or equal to *list* length, append *item* to *list*.

index must be greater than or equal to `0`.

Python must be preinitialized to call this function.

Structure fields:

Py_ssize_t length

List length.

wchar_t **items

List items.

PyStatus

type PyStatus

Structure to store an initialization function status: success, error or exit.

For an error, it can store the C function name which created the error.

Structure fields:

int exitcode

Exit code. Argument passed to `exit()`.

const char *err_msg

Error message.

const char *func

Name of the function which created an error, can be `NULL`.

Functions to create a status:

PyStatus PyStatus_Ok(void)

Success.

PyStatus PyStatus_Error(const char *err_msg)

Initialization error with a message.

err_msg must not be `NULL`.

PyStatus PyStatus_NoMemory(void)

Memory allocation failure (out of memory).

PyStatus PyStatus_Exit(int exitcode)

Exit Python with the specified exit code.

Functions to handle a status:

int PyStatus_Exception(PyStatus status)

Is the status an error or an exit? If true, the exception must be handled; by calling **`Py_ExitStatusException()`** for example.

int PyStatus_IsError(PyStatus status)

Is the result an error?

int PyStatus_IsExit(PyStatus status)

Is the result an exit?

void Py_ExitStatusException(PyStatus status)

Call `exit(exitcode)` if *status* is an exit. Print the error message and exit with a non-zero exit code if *status* is an error. Must only be called if `PyStatus_Exception(status)` is non-zero.

Note

Internally, Python uses macros which set `PyStatus.func`, whereas functions to create a status set `func` to `NULL`.

Example:

```
PyStatus alloc(void **ptr, size_t size)
{
    *ptr = PyMem_RawMalloc(size);
    if (*ptr == NULL) {
        return PyStatus_NoMemory();
    }
    return PyStatus_Ok();
}
```

```
int main(int argc, char **argv)
{
    void *ptr;
    PyStatus status = alloc(&ptr, 16);
    if (PyStatus_Exception(status)) {
        Py_ExitStatusException(status);
    }
    PyMem_Free(ptr);
    return 0;
}
```

PyPreConfig

type PyPreConfig

Structure used to preinitialize Python.

Function to initialize a preconfiguration:

void PyPreConfig_InitPythonConfig(PyPreConfig *preconfig)

Initialize the preconfiguration with Python Configuration.

void PyPreConfig_InitIsolatedConfig(PyPreConfig *preconfig)

Initialize the preconfiguration with Isolated Configuration.

Structure fields:

int allocator

Name of the Python memory allocators:

- `PYMEM_ALLOCATOR_NOT_SET` (`0`): don’t change memory allocators (use defaults).
- `PYMEM_ALLOCATOR_DEFAULT` (`1`): default memory allocators.
- `PYMEM_ALLOCATOR_DEBUG` (`2`): default memory allocators with debug hooks.
- `PYMEM_ALLOCATOR_MALLOC` (`3`): use `malloc()` of the C library.
- `PYMEM_ALLOCATOR_MALLOC_DEBUG` (`4`): force usage of `malloc()` with debug hooks.
- `PYMEM_ALLOCATOR_PYMALLOC` (`5`): Python pymalloc memory allocator.
- `PYMEM_ALLOCATOR_PYMALLOC_DEBUG` (`6`): Python pymalloc memory allocator with debug hooks.
- `PYMEM_ALLOCATOR_MIMALLOC` (`6`): use `mimalloc`, a fast malloc replacement.
- `PYMEM_ALLOCATOR_MIMALLOC_DEBUG` (`7`): use `mimalloc`, a fast malloc replacement with debug hooks.

`PYMEM_ALLOCATOR_PYMALLOC` and `PYMEM_ALLOCATOR_PYMALLOC_DEBUG` are not supported if Python is **`configured using --without-pymalloc`**.

`PYMEM_ALLOCATOR_MIMALLOC` and `PYMEM_ALLOCATOR_MIMALLOC_DEBUG` are not supported if Python is **`configured using --without-mimalloc`** or if the underlying atomic support isn't available.

See Memory Management.

Default: `PYMEM_ALLOCATOR_NOT_SET`.

int configure_locale

Set the LC_CTYPE locale to the user preferred locale.

If equals to `0`, set **`coerce_c_locale`** and **`coerce_c_locale_warn`** members to `0`.

See the locale encoding.

Default: `1` in Python config, `0` in isolated config.

int coerce_c_locale

If equals to `2`, coerce the C locale.

If equals to `1`, read the LC_CTYPE locale to decide if it should be coerced.

See the locale encoding.

Default: `-1` in Python config, `0` in isolated config.

int coerce_c_locale_warn

If non-zero, emit a warning if the C locale is coerced.

Default: `-1` in Python config, `0` in isolated config.

int dev_mode

Python Development Mode: see **`PyConfig.dev_mode`**.

Default: `-1` in Python mode, `0` in isolated mode.

int isolated

Isolated mode: see **`PyConfig.isolated`**.

Default: `0` in Python mode, `1` in isolated mode.

int legacy_windows_fs_encoding

If non-zero:

- Set **`PyPreConfig.utf8_mode`** to `0`,
- Set **`PyConfig.filesystem_encoding`** to `"mbcs"`,
- Set **`PyConfig.filesystem_errors`** to `"replace"`.

Initialized from the **`PYTHONLEGACYWINDOWSFSENCODING`** environment variable value.

Only available on Windows. `#ifdef MS_WINDOWS` macro can be used for Windows specific code.

Default: `0`.

int parse_argv

If non-zero, **`Py_PreInitializeFromArgs()`** and **`Py_PreInitializeFromBytesArgs()`** parse their `argv` argument the same way the regular Python parses command line arguments: see Command Line Arguments.

Default: `1` in Python config, `0` in isolated config.

int use_environment

Use environment variables? See **`PyConfig.use_environment`**.

Default: `1` in Python config and `0` in isolated config.

int utf8_mode

If non-zero, enable the Python UTF-8 Mode.

Set to `0` or `1` by the **`-X utf8`** command line option and the **`PYTHONUTF8`** environment variable.

Also set to `1` if the `LC_CTYPE` locale is `C` or `POSIX`.

Default: `-1` in Python config and `0` in isolated config.

Preinitialize Python with PyPreConfig

The preinitialization of Python:

- Set the Python memory allocators (**`PyPreConfig.allocator`**)
- Configure the LC_CTYPE locale (locale encoding)
- Set the Python UTF-8 Mode (**`PyPreConfig.utf8_mode`**)

The current preconfiguration (`PyPreConfig` type) is stored in `_PyRuntime.preconfig`.

Functions to preinitialize Python:

PyStatus Py_PreInitialize(const PyPreConfig *preconfig)

Preinitialize Python from *preconfig* preconfiguration.

preconfig must not be `NULL`.

PyStatus Py_PreInitializeFromBytesArgs(const PyPreConfig *preconfig, int argc, char *const *argv)

Preinitialize Python from *preconfig* preconfiguration.

Parse *argv* command line arguments (bytes strings) if **`parse_argv`** of *preconfig* is non-zero.

preconfig must not be `NULL`.

PyStatus Py_PreInitializeFromArgs(const PyPreConfig *preconfig, int argc, wchar_t *const *argv)

Preinitialize Python from *preconfig* preconfiguration.

Parse *argv* command line arguments (wide strings) if **`parse_argv`** of *preconfig* is non-zero.

preconfig must not be `NULL`.

The caller is responsible to handle exceptions (error or exit) using **`PyStatus_Exception()`** and **`Py_ExitStatusException()`**.

For Python Configuration (**`PyPreConfig_InitPythonConfig()`**), if Python is initialized with command line arguments, the command line arguments must also be passed to preinitialize Python, since they have an effect on the pre-configuration like encodings. For example, the **`-X utf8`** command line option enables the Python UTF-8 Mode.

`PyMem_SetAllocator()` can be called after **`Py_PreInitialize()`** and before **`Py_InitializeFromConfig()`** to install a custom memory allocator. It can be called before **`Py_PreInitialize()`** if **`PyPreConfig.allocator`** is set to `PYMEM_ALLOCATOR_NOT_SET`.

Python memory allocation functions like **`PyMem_RawMalloc()`** must not be used before the Python preinitialization, whereas calling directly `malloc()`

and `free()` is always safe. **`Py_DecodeLocale()`** must not be called before the Python preinitialization.

Example using the preinitialization to enable the Python UTF-8 Mode:

```
PyStatus status;
PyPreConfig preconfig;
PyPreConfig_InitPythonConfig(&preconfig);

preconfig.utf8_mode = 1;

status = Py_PreInitialize(&preconfig);
if (PyStatus_Exception(status)) {
    Py_ExitStatusException(status);
}

/* at this point, Python speaks UTF-8 */

Py_Initialize();
/* ... use Python API here ... */
Py_Finalize();
```

PyConfig

type PyConfig

Structure containing most parameters to configure Python.

When done, the **`PyConfig_Clear()`** function must be used to release the configuration memory.

Structure methods:

void PyConfig_InitPythonConfig(PyConfig *config)

Initialize configuration with the Python Configuration.

void PyConfig_InitIsolatedConfig(PyConfig *config)

Initialize configuration with the Isolated Configuration.

PyStatus PyConfig_SetString(PyConfig *config, wchar_t *const *config_str, const wchar_t *str)

Copy the wide character string *str* into `*config_str`.

Preinitialize Python if needed.

PyStatus PyConfig_SetBytesString(PyConfig *config, wchar_t *const *config_str, const char *str)

Decode *str* using **`Py_DecodeLocale()`** and set the result into `*config_str`.

Preinitialize Python if needed.

PyStatus PyConfig_SetArgv(PyConfig *config, int argc, wchar_t *const *argv)

Set command line arguments (**`argv`** member of *config*) from the *argv* list of wide character strings.

Preinitialize Python if needed.

PyStatus PyConfig_SetBytesArgv(PyConfig *config, int argc, char *const *argv)

Set command line arguments (**`argv`** member of *config*) from the *argv* list of bytes strings. Decode bytes using **`Py_DecodeLocale()`**.

Preinitialize Python if needed.

PyStatus PyConfig_SetWideStringList(PyConfig *config, PyWideStringList *list, Py_ssize_t length, wchar_t **items)

Set the list of wide strings *list* to *length* and *items*.

Preinitialize Python if needed.

PyStatus PyConfig_Read(PyConfig *config)

Read all Python configuration.

Fields which are already initialized are left unchanged.

Fields for path configuration are no longer calculated or modified when calling this function, as of Python 3.11.

The **`PyConfig_Read()`** function only parses **`PyConfig.argv`** arguments once: **`PyConfig.parse_argv`** is set to `2` after arguments are parsed. Since Python arguments are stripped from **`PyConfig.argv`**, parsing arguments twice would parse the application options as Python options.

Preinitialize Python if needed.

Changed in version 3.10: The **`PyConfig.argv`** arguments are now only parsed once, **`PyConfig.parse_argv`** is set to `2` after arguments are parsed, and arguments are only parsed if **`PyConfig.parse_argv`** equals `1`.

Changed in version 3.11: `PyConfig_Read()` no longer calculates all paths, and so fields listed under Python Path Configuration may no longer be updated until `Py_InitializeFromConfig()` is called.

void PyConfig_Clear(PyConfig *config)

Release configuration memory.

Most `PyConfig` methods preinitialize Python if needed. In that case, the Python preinitialization configuration (`PyPreConfig`) in based on the `PyConfig`. If configuration fields which are in common with `PyPreConfig` are tuned, they must be set before calling a `PyConfig` method:

- `PyConfig.dev_mode`
- `PyConfig.isolated`
- `PyConfig.parse_argv`
- `PyConfig.use_environment`

Moreover, if `PyConfig_SetArgv()` or `PyConfig_SetBytesArgv()` is used, this method must be called before other methods, since the preinitialization configuration depends on command line arguments (if `parse_argv` is non-zero).

The caller of these methods is responsible to handle exceptions (error or exit) using `PyStatus_Exception()` and `Py_ExitStatusException()`.

Structure fields:

PyWideStringList argv

Set `sys.argv` command line arguments based on `argv`. These parameters are similar to those passed to the program's **`main()`** function with the difference that the first entry should refer to the script file to be executed rather than the executable hosting the Python interpreter. If there isn't a script that will be run, the first entry in `argv` can be an empty string.

Set `parse_argv` to `1` to parse `argv` the same way the regular Python parses Python command line arguments and then to strip Python arguments from `argv`.

If `argv` is empty, an empty string is added to ensure that `sys.argv` always exists and is never empty.

Default: `NULL`.

See also the `orig_argv` member.

int safe_path

If equals to zero, `Py_RunMain()` prepends a potentially unsafe path to **`sys.path`** at startup:

- If **`argv[0]`** is equal to `L"-m"` (`python -m module`), prepend the current working directory.
- If running a script (`python script.py`), prepend the script's directory. If it's a symbolic link, resolve symbolic links.
- Otherwise (`python -c code` and `python`), prepend an empty string, which means the current working directory.

Set to `1` by the **`-P`** command line option and the **`PYTHONSAFEPATH`** environment variable.

Default: `0` in Python config, `1` in isolated config.

Added in version 3.11.

wchar_t *base_exec_prefix

`sys.base_exec_prefix`.

Default: `NULL`.

Part of the Python Path Configuration output.

See also **`PyConfig.exec_prefix`**.

wchar_t *base_executable

Python base executable: **`sys._base_executable`**.

Set by the **`__PYVENV_LAUNCHER__`** environment variable.

Set from **`PyConfig.executable`** if `NULL`.

Default: `NULL`.

Part of the Python Path Configuration output.

See also **`PyConfig.executable`**.

wchar_t *base_prefix

`sys.base_prefix`.

Default: `NULL`.

Part of the Python Path Configuration output.

See also `PyConfig.prefix`.

int buffered_stdio

If equals to `0` and `configure_c_stdio` is non-zero, disable buffering on the C streams stdout and stderr.

Set to `0` by the `-u` command line option and the `PYTHONUNBUFFERED` environment variable.

stdin is always opened in buffered mode.

Default: `1`.

int bytes_warning

If equals to `1`, issue a warning when comparing `bytes` or `bytearray` with `str`, or comparing `bytes` with `int`.

If equal or greater to `2`, raise a `BytesWarning` exception in these cases.

Incremented by the `-b` command line option.

Default: `0`.

int warn_default_encoding

If non-zero, emit a `EncodingWarning` warning when `io.TextIOWrapper` uses its default encoding. See Opt-in EncodingWarning for details.

Default: `0`.

Added in version 3.10.

int code_debug_ranges

If equals to `0`, disables the inclusion of the end line and column mappings in code objects. Also disables traceback printing carets to specific error locations.

Set to `0` by the `PYTHONNODEBUGRANGES` environment variable and by the `-X no_debug_ranges` command line option.

Default: `1`.

Added in version 3.11.

wchar_t *check_hash_pycs_mode

Control the validation behavior of hash-based `.pyc` files: value of the `--check-hash-based-pycs` command line option.

Valid values:

- `L"always"`: Hash the source file for invalidation regardless of value of the ‘check_source’ flag.
- `L"never"`: Assume that hash-based pycs always are valid.
- `L"default"`: The ‘check_source’ flag in hash-based pycs determines invalidation.

Default: `L"default"`.

See also **PEP 552** [https://peps.python.org/pep-0552/] “Deterministic pycs”.

int configure_c_stdio

If non-zero, configure C standard streams:

- On Windows, set the binary mode (`O_BINARY`) on stdin, stdout and stderr.
- If **`buffered_stdio`** equals zero, disable buffering of stdin, stdout and stderr streams.
- If **`interactive`** is non-zero, enable stream buffering on stdin and stdout (only stdout on Windows).

Default: `1` in Python config, `0` in isolated config.

int dev_mode

If non-zero, enable the Python Development Mode.

Set to `1` by the **`-X dev`** option and the **`PYTHONDEVMODE`** environment variable.

Default: `-1` in Python mode, `0` in isolated mode.

int dump_refs

Dump Python references?

If non-zero, dump all objects which are still alive at exit.

Set to `1` by the **`PYTHONDUMPREFS`** environment variable.

Needs a special build of Python with the `Py_TRACE_REFS` macro defined: see the **`configure --with-trace-refs option`**.

Default: `0`.

wchar_t *dump_refs_file

Filename where to dump Python references.

Set by the **`PYTHONDUMPREFSFILE`** environment variable.

Default: `NULL`.

Added in version 3.11.

wchar_t *exec_prefix

The site-specific directory prefix where the platform-dependent Python files are installed: **`sys.exec_prefix`**.

Default: `NULL`.

Part of the Python Path Configuration output.

See also **`PyConfig.base_exec_prefix`**.

wchar_t *executable

The absolute path of the executable binary for the Python interpreter: **`sys.executable`**.

Default: `NULL`.

Part of the Python Path Configuration output.

See also **`PyConfig.base_executable`**.

int faulthandler

Enable faulthandler?

If non-zero, call **`faulthandler.enable()`** at startup.

Set to `1` by **`-X faulthandler`** and the **`PYTHONFAULTHANDLER`** environment variable.

Default: `-1` in Python mode, `0` in isolated mode.

wchar_t *filesystem_encoding

Filesystem encoding: **`sys.getfilesystemencoding()`**.

On macOS, Android and VxWorks: use `"utf-8"` by default.

On Windows: use `"utf-8"` by default, or `"mbcs"` if **`legacy_windows_fs_encoding`** of **`PyPreConfig`** is non-zero.

Default encoding on other platforms:

- `"utf-8"` if **`PyPreConfig.utf8_mode`** is non-zero.
- `"ascii"` if Python detects that `nl_langinfo(CODESET)` announces the ASCII encoding, whereas the `mbstowcs()` function decodes from a different encoding (usually Latin1).
- `"utf-8"` if `nl_langinfo(CODESET)` returns an empty string.
- Otherwise, use the locale encoding: `nl_langinfo(CODESET)` result.

At Python startup, the encoding name is normalized to the Python codec name. For example, `"ANSI_X3.4-1968"` is replaced with `"ascii"`.

See also the **`filesystem_errors`** member.

wchar_t *filesystem_errors

Filesystem error handler: **`sys.getfilesystemencodeerrors()`**.

On Windows: use `"surrogatepass"` by default, or `"replace"` if **`legacy_windows_fs_encoding`** of **`PyPreConfig`** is non-zero.

On other platforms: use `"surrogateescape"` by default.

Supported error handlers:

- `"strict"`
- `"surrogateescape"`
- `"surrogatepass"` (only supported with the UTF-8 encoding)

See also the **`filesystem_encoding`** member.

int use_frozen_modules

If non-zero, use frozen modules.

Set by the **`PYTHON_FROZEN_MODULES`** environment variable.

Default: `1` in a release build, or `0` in a debug build.

unsigned long hash_seed

int use_hash_seed

Randomized hash function seed.

If **`use_hash_seed`** is zero, a seed is chosen randomly at Python startup, and **`hash_seed`** is ignored.

Set by the **`PYTHONHASHSEED`** environment variable.

Default *use_hash_seed* value: `-1` in Python mode, `0` in isolated mode.

wchar_t *home

Set the default Python “home” directory, that is, the location of the standard Python libraries (see **`PYTHONHOME`**).

Set by the **`PYTHONHOME`** environment variable.

Default: `NULL`.

Part of the Python Path Configuration input.

int import_time

If `1`, profile import time. If `2`, include additional output that indicates when an imported module has already been loaded.

Set by the **`-X importtime`** option and the **`PYTHONPROFILEIMPORTTIME`** environment variable.

Default: `0`.

Changed in version 3.14: Added support for `import_time = 2`

int inspect

Enter interactive mode after executing a script or a command.

If greater than `0`, enable inspect: when a script is passed as first argument or the -c option is used, enter interactive mode after executing the script or the command, even when **`sys.stdin`** does not appear to be a terminal.

Incremented by the **`-i`** command line option. Set to `1` if the **`PYTHONINSPECT`** environment variable is non-empty.

Default: `0`.

int install_signal_handlers

Install Python signal handlers?

Default: `1` in Python mode, `0` in isolated mode.

int interactive

If greater than `0`, enable the interactive mode (REPL).

Incremented by the **`-i`** command line option.

Default: `0`.

int int_max_str_digits

Configures the integer string conversion length limitation. An initial value of `-1` means the value will be taken from the command line or environment or otherwise default to 4300 (`sys.int_info.default_max_str_digits`). A value of `0` disables the limitation. Values greater than zero but less than 640 (`sys.int_info.str_digits_check_threshold`) are unsupported and will produce an error.

Configured by the `-X int_max_str_digits` command line flag or the `PYTHONINTMAXSTRDIGITS` environment variable.

Default: `-1` in Python mode. 4300 (`sys.int_info.default_max_str_digits`) in isolated mode.

Added in version 3.12.

int cpu_count

If the value of `cpu_count` is not `-1` then it will override the return values of `os.cpu_count()`, `os.process_cpu_count()`, and `multiprocessing.cpu_count()`.

Configured by the `-X cpu_count=n|default` command line flag or the `PYTHON_CPU_COUNT` environment variable.

Default: `-1`.

Added in version 3.13.

int isolated

If greater than `0`, enable isolated mode:

- Set `safe_path` to `1`: don't prepend a potentially unsafe path to `sys.path` at Python startup, such as the current directory, the script's directory or an empty string.
- Set `use_environment` to `0`: ignore `PYTHON` environment variables.
- Set `user_site_directory` to `0`: don't add the user site directory to `sys.path`.
- Python REPL doesn't import `readline` nor enable default readline configuration on interactive prompts.

Set to `1` by the `-I` command line option.

Default: `0` in Python mode, `1` in isolated mode.

See also the Isolated Configuration and `PyPreConfig.isolated`.

int legacy_windows_stdio

If non-zero, use `io.FileIO` instead of **`io._WindowsConsoleIO`** for `sys.stdin`, `sys.stdout` and `sys.stderr`.

Set to `1` if the `PYTHONLEGACYWINDOWSSTDIO` environment variable is set to a non-empty string.

Only available on Windows. `#ifdef MS_WINDOWS` macro can be used for Windows specific code.

Default: `0`.

See also the **PEP 528** [https://peps.python.org/pep-0528/] (Change Windows console encoding to UTF-8).

int malloc_stats

If non-zero, dump statistics on Python pymalloc memory allocator at exit.

Set to `1` by the `PYTHONMALLOCSTATS` environment variable.

The option is ignored if Python is `configured using the --without-pymalloc option`.

Default: `0`.

wchar_t *platlibdir

Platform library directory name: `sys.platlibdir`.

Set by the `PYTHONPLATLIBDIR` environment variable.

Default: value of the `PLATLIBDIR` macro which is set by the `configure --with-platlibdir option` (default: `"lib"`, or `"DLLs"` on Windows).

Part of the Python Path Configuration input.

Added in version 3.9.

Changed in version 3.11: This macro is now used on Windows to locate the standard library extension modules, typically under `DLLs`. However, for compatibility, note that this value is ignored for any non-standard layouts, including in-tree builds and virtual environments.

wchar_t *pythonpath_env

Module search paths (**`sys.path`**) as a string separated by `DELIM` (**`os.pathsep`**).

Set by the **`PYTHONPATH`** environment variable.

Default: `NULL`.

Part of the Python Path Configuration input.

PyWideStringList module_search_paths

int module_search_paths_set

Module search paths: **`sys.path`**.

If **`module_search_paths_set`** is equal to `0`, **`Py_InitializeFromConfig()`** will replace **`module_search_paths`** and sets **`module_search_paths_set`** to `1`.

Default: empty list (`module_search_paths`) and `0` (`module_search_paths_set`).

Part of the Python Path Configuration output.

int optimization_level

Compilation optimization level:

- `0`: Peephole optimizer, set `__debug__` to `True`.
- `1`: Level 0, remove assertions, set `__debug__` to `False`.
- `2`: Level 1, strip docstrings.

Incremented by the **`-O`** command line option. Set to the **`PYTHONOPTIMIZE`** environment variable value.

Default: `0`.

PyWideStringList orig_argv

The list of the original command line arguments passed to the Python executable: **`sys.orig_argv`**.

If **`orig_argv`** list is empty and **`argv`** is not a list only containing an empty string, **`PyConfig_Read()`** copies **`argv`** into **`orig_argv`** before modifying **`argv`** (if **`parse_argv`** is non-zero).

See also the **`argv`** member and the **`Py_GetArgcArgv()`** function.

Default: empty list.

Added in version 3.10.

int parse_argv

Parse command line arguments?

If equals to `1`, parse **`argv`** the same way the regular Python parses command line arguments, and strip Python arguments from **`argv`**.

The **`PyConfig_Read()`** function only parses **`PyConfig.argv`** arguments once: **`PyConfig.parse_argv`** is set to `2` after arguments are parsed. Since Python arguments are stripped from **`PyConfig.argv`**, parsing arguments twice would parse the application options as Python options.

Default: `1` in Python mode, `0` in isolated mode.

Changed in version 3.10: The **`PyConfig.argv`** arguments are now only parsed if **`PyConfig.parse_argv`** equals to `1`.

int parser_debug

Parser debug mode. If greater than `0`, turn on parser debugging output (for expert only, depending on compilation options).

Incremented by the **`-d`** command line option. Set to the **`PYTHONDEBUG`** environment variable value.

Needs a debug build of Python (the `Py_DEBUG` macro must be defined).

Default: `0`.

int pathconfig_warnings

If non-zero, calculation of path configuration is allowed to log warnings into `stderr`. If equals to `0`, suppress these warnings.

Default: `1` in Python mode, `0` in isolated mode.

Part of the Python Path Configuration input.

Changed in version 3.11: Now also applies on Windows.

wchar_t *prefix

The site-specific directory prefix where the platform independent Python files are installed: **`sys.prefix`**.

Default: `NULL`.

Part of the Python Path Configuration output.

See also `PyConfig.base_prefix`.

wchar_t *program_name

Program name used to initialize `executable` and in early error messages during Python initialization.

- On macOS, use `PYTHONEXECUTABLE` environment variable if set.
- If the `WITH_NEXT_FRAMEWORK` macro is defined, use **`__PYVENV_LAUNCHER__`** environment variable if set.
- Use `argv[0]` of `argv` if available and non-empty.
- Otherwise, use `L"python"` on Windows, or `L"python3"` on other platforms.

Default: `NULL`.

Part of the Python Path Configuration input.

wchar_t *pycache_prefix

Directory where cached `.pyc` files are written: `sys.pycache_prefix`.

Set by the `-X pycache_prefix=PATH` command line option and the `PYTHONPYCACHEPREFIX` environment variable. The command-line option takes precedence.

If `NULL`, `sys.pycache_prefix` is set to `None`.

Default: `NULL`.

int quiet

Quiet mode. If greater than `0`, don’t display the copyright and version at Python startup in interactive mode.

Incremented by the `-q` command line option.

Default: `0`.

wchar_t *run_command

Value of the `-c` command line option.

Used by `Py_RunMain()`.

Default: `NULL`.

wchar_t *run_filename

Filename passed on the command line: trailing command line argument without `-c` or `-m`. It is used by the `Py_RunMain()` function.

For example, it is set to `script.py` by the `python3 script.py arg` command line.

See also the **`PyConfig.skip_source_first_line`** option.

Default: `NULL`.

wchar_t *run_module

Value of the **`-m`** command line option.

Used by **`Py_RunMain()`**.

Default: `NULL`.

wchar_t *run_presite

`package.module` path to module that should be imported before `site.py` is run.

Set by the **`-X presite=package.module`** command-line option and the **`PYTHON_PRESITE`** environment variable. The command-line option takes precedence.

Needs a debug build of Python (the `Py_DEBUG` macro must be defined).

Default: `NULL`.

int show_ref_count

Show total reference count at exit (excluding immortal objects)?

Set to `1` by **`-X showrefcount`** command line option.

Needs a debug build of Python (the `Py_REF_DEBUG` macro must be defined).

Default: `0`.

int site_import

Import the **`site`** module at startup?

If equal to zero, disable the import of the module site and the site-dependent manipulations of **`sys.path`** that it entails.

Also disable these manipulations if the **`site`** module is explicitly imported later (call **`site.main()`** if you want them to be triggered).

Set to `0` by the **`-S`** command line option.

`sys.flags.no_site` is set to the inverted value of **`site_import`**.

Default: `1`.

int skip_source_first_line

If non-zero, skip the first line of the **`PyConfig.run_filename`** source.

It allows the usage of non-Unix forms of `#!cmd`. This is intended for a DOS specific hack only.

Set to `1` by the **`-x`** command line option.

Default: `0`.

wchar_t *stdio_encoding

wchar_t *stdio_errors

Encoding and encoding errors of **`sys.stdin`**, **`sys.stdout`** and **`sys.stderr`** (but **`sys.stderr`** always uses `"backslashreplace"` error handler).

Use the **`PYTHONIOENCODING`** environment variable if it is non-empty.

Default encoding:

- `"UTF-8"` if **`PyPreConfig.utf8_mode`** is non-zero.
- Otherwise, use the locale encoding.

Default error handler:

- On Windows: use `"surrogateescape"`.
- `"surrogateescape"` if **`PyPreConfig.utf8_mode`** is non-zero, or if the LC_CTYPE locale is “C” or “POSIX”.
- `"strict"` otherwise.

See also **`PyConfig.legacy_windows_stdio`**.

int tracemalloc

Enable tracemalloc?

If non-zero, call **`tracemalloc.start()`** at startup.

Set by **`-X tracemalloc=N`** command line option and by the **`PYTHONTRACEMALLOC`** environment variable.

Default: `-1` in Python mode, `0` in isolated mode.

int perf_profiling

Enable the Linux `perf` profiler support?

If equals to `1`, enable support for the Linux `perf` profiler.

If equals to `2`, enable support for the Linux `perf` profiler with DWARF JIT support.

Set to `1` by **`-X perf`** command-line option and the **`PYTHONPERFSUPPORT`** environment variable.

Set to `2` by the **`-X perf_jit`** command-line option and the **`PYTHON_PERF_JIT_SUPPORT`** environment variable.

Default: `-1`.

See also

See Python support for the Linux perf profiler for more information.

Added in version 3.12.

wchar_t *stdlib_dir

Directory of the Python standard library.

Default: `NULL`.

Added in version 3.11.

int use_environment

Use environment variables?

If equals to zero, ignore the environment variables.

Set to `0` by the **`-E`** environment variable.

Default: `1` in Python config and `0` in isolated config.

int use_system_logger

If non-zero, `stdout` and `stderr` will be redirected to the system log.

Only available on macOS 10.12 and later, and on iOS.

Default: `0` (don’t use the system log) on macOS; `1` on iOS (use the system log).

Added in version 3.14.

int user_site_directory

If non-zero, add the user site directory to `sys.path`.

Set to `0` by the `-s` and `-I` command line options.

Set to `0` by the `PYTHONNOUSERSITE` environment variable.

Default: `1` in Python mode, `0` in isolated mode.

int verbose

Verbose mode. If greater than `0`, print a message each time a module is imported, showing the place (filename or built-in module) from which it is loaded.

If greater than or equal to `2`, print a message for each file that is checked for when searching for a module. Also provides information on module cleanup at exit.

Incremented by the `-v` command line option.

Set by the `PYTHONVERBOSE` environment variable value.

Default: `0`.

PyWideStringList warnoptions

Options of the `warnings` module to build warnings filters, lowest to highest priority: `sys.warnoptions`.

The `warnings` module adds `sys.warnoptions` in the reverse order: the last `PyConfig.warnoptions` item becomes the first item of **`warnings.filters`** which is checked first (highest priority).

The `-W` command line options adds its value to `warnoptions`, it can be used multiple times.

The `PYTHONWARNINGS` environment variable can also be used to add warning options. Multiple options can be specified, separated by commas (`,`).

Default: empty list.

int write_bytecode

If equal to `0`, Python won’t try to write `.pyc` files on the import of source modules.

Set to `0` by the **`-B`** command line option and the **`PYTHONDONTWRITEBYTECODE`** environment variable.

`sys.dont_write_bytecode` is initialized to the inverted value of **`write_bytecode`**.

Default: `1`.

PyWideStringList xoptions

Values of the **`-X`** command line options: **`sys._xoptions`**.

Default: empty list.

int _pystats

If non-zero, write performance statistics at Python exit.

Need a special build with the `Py_STATS` macro: see **`--enable-pystats`**.

Default: `0`.

If **`parse_argv`** is non-zero, **`argv`** arguments are parsed the same way the regular Python parses command line arguments, and Python arguments are stripped from **`argv`**.

The **`xoptions`** options are parsed to set other options: see the **`-X`** command line option.

Changed in version 3.9: The `show_alloc_count` field has been removed.

Initialization with PyConfig

Initializing the interpreter from a populated configuration struct is handled by calling **`Py_InitializeFromConfig()`**.

The caller is responsible to handle exceptions (error or exit) using **`PyStatus_Exception()`** and **`Py_ExitStatusException()`**.

If **`PyImport_FrozenModules()`**, **`PyImport_AppendInittab()`** or **`PyImport_ExtendInittab()`** are used, they must be set or called after Python preinitialization and before the Python initialization. If Python is initialized multiple times, **`PyImport_AppendInittab()`** or **`PyImport_ExtendInittab()`** must be called before each Python initialization.

The current configuration (`PyConfig` type) is stored in `PyInterpreterState.config`.

Example setting the program name:

```
void init_python(void)
{
    PyStatus status;

    PyConfig config;
    PyConfig_InitPythonConfig(&config);

    /* Set the program name. Implicitly preinitialize Python. */
    status = PyConfig_SetString(&config, &config.program_name,
                                L"/path/to/my_program");
    if (PyStatus_Exception(status)) {
        goto exception;
    }

    status = Py_InitializeFromConfig(&config);
    if (PyStatus_Exception(status)) {
        goto exception;
    }
    PyConfig_Clear(&config);
    return;

exception:
    PyConfig_Clear(&config);
    Py_ExitStatusException(status);
}
```

More complete example modifying the default configuration, read the configuration, and then override some parameters. Note that since 3.11, many parameters are not calculated until initialization, and so values cannot be read from the configuration structure. Any values set before initialize is called will be left unchanged by initialization:

```
PyStatus init_python(const char *program_name)
{
    PyStatus status;

    PyConfig config;
    PyConfig_InitPythonConfig(&config);

    /* Set the program name before reading the configuration
       (decode byte string from the locale encoding).

       Implicitly preinitialize Python. */
    status = PyConfig_SetBytesString(&config, &config.program_name,
                                     program_name);
    if (PyStatus_Exception(status)) {
        goto done;
    }
    /* Read all configuration at once */
    status = PyConfig_Read(&config);
    if (PyStatus_Exception(status)) {
        goto done;
    }

    /* Specify sys.path explicitly */
    /* If you want to modify the default set of paths, finish
       initialization first and then use PySys_GetObject("path") */
    config.module_search_paths_set = 1;
    status = PyWideStringList_Append(&config.module_search_paths,
                                     L"/path/to/stdlib");
    if (PyStatus_Exception(status)) {
        goto done;
    }
    status = PyWideStringList_Append(&config.module_search_paths,
                                     L"/path/to/more/modules");
    if (PyStatus_Exception(status)) {
        goto done;
    }

    /* Override executable computed by PyConfig_Read() */
    status = PyConfig_SetString(&config, &config.executable,
                                L"/path/to/my_executable");
    if (PyStatus_Exception(status)) {
        goto done;
    }

    status = Py_InitializeFromConfig(&config);

done:
    PyConfig_Clear(&config);
    return status;
}
```

Isolated Configuration

`PyPreConfig_InitIsolatedConfig()` and `PyConfig_InitIsolatedConfig()` functions create a configuration to isolate Python from the system. For example, to embed Python into an application.

This configuration ignores global configuration variables, environment variables, command line arguments (`PyConfig.argv` is not parsed) and user site directory. The C standard streams (ex: `stdout`) and the LC_CTYPE locale are left unchanged. Signal handlers are not installed.

Configuration files are still used with this configuration to determine paths that are unspecified. Ensure `PyConfig.home` is specified to avoid computing the default path configuration.

Python Configuration

`PyPreConfig_InitPythonConfig()` and `PyConfig_InitPythonConfig()` functions create a configuration to build a customized Python which behaves as the regular Python.

Environments variables and command line arguments are used to configure Python, whereas global configuration variables are ignored.

This function enables C locale coercion (**PEP 538** [https://peps.python.org/pep-0538/]) and Python UTF-8 Mode (**PEP 540** [https://peps.python.org/pep-0540/]) depending on the LC_CTYPE locale, `PYTHONUTF8` and `PYTHONCOERCECLOCALE` environment variables.

Python Path Configuration

`PyConfig` contains multiple fields for the path configuration:

- Path configuration inputs:
    - `PyConfig.home`
    - `PyConfig.platlibdir`
    - `PyConfig.pathconfig_warnings`
    - `PyConfig.program_name`
    - `PyConfig.pythonpath_env`
    - current working directory: to get absolute paths
    - `PATH` environment variable to get the program full path (from `PyConfig.program_name`)
    - `__PYVENV_LAUNCHER__` environment variable
    - (Windows only) Application paths in the registry under “SoftwarePythonPythonCoreX.YPythonPath” of HKEY_CURRENT_USER and HKEY_LOCAL_MACHINE (where X.Y is the Python version).
- Path configuration output fields:
    - `PyConfig.base_exec_prefix`
    - `PyConfig.base_executable`
    - `PyConfig.base_prefix`
    - `PyConfig.exec_prefix`
    - `PyConfig.executable`
    - `PyConfig.module_search_paths_set`, `PyConfig.module_search_paths`
    - `PyConfig.prefix`

If at least one “output field” is not set, Python calculates the path configuration to fill unset fields. If `module_search_paths_set` is equal to `0`, `module_search_paths` is overridden and `module_search_paths_set` is set to `1`.

It is possible to completely ignore the function calculating the default path configuration by setting explicitly all path configuration output fields listed above. A string is considered as set even if it is non-empty. `module_search_paths` is considered as set if `module_search_paths_set` is set to `1`. In this case, `module_search_paths` will be used without modification.

Set **`pathconfig_warnings`** to `0` to suppress warnings when calculating the path configuration (Unix only, Windows does not log any warning).

If **`base_prefix`** or **`base_exec_prefix`** fields are not set, they inherit their value from **`prefix`** and **`exec_prefix`** respectively.

`Py_RunMain()` and **`Py_Main()`** modify **`sys.path`**:

- If **`run_filename`** is set and is a directory which contains a `__main__.py` script, prepend **`run_filename`** to **`sys.path`**.
- If **`isolated`** is zero:
    - If **`run_module`** is set, prepend the current directory to **`sys.path`**. Do nothing if the current directory cannot be read.
    - If **`run_filename`** is set, prepend the directory of the filename to **`sys.path`**.
    - Otherwise, prepend an empty string to **`sys.path`**.

If **`site_import`** is non-zero, **`sys.path`** can be modified by the **`site`** module. If **`user_site_directory`** is non-zero and the user's site-package directory exists, the **`site`** module appends the user's site-package directory to **`sys.path`**.

The following configuration files are used by the path configuration:

- `pyvenv.cfg`
- `._pth` file (ex: `python._pth`)
- `pybuilddir.txt` (Unix only)

If a `._pth` file is present:

- Set **`isolated`** to `1`.
- Set **`use_environment`** to `0`.
- Set **`site_import`** to `0`.
- Set **`safe_path`** to `1`.

If **`home`** is not set and a `pyvenv.cfg` file is present in the same directory as **`executable`**, or its parent, **`prefix`** and **`exec_prefix`** are set that location. When this happens, **`base_prefix`** and **`base_exec_prefix`** still keep their value, pointing to the base installation. See Virtual Environments for more information.

The `__PYVENV_LAUNCHER__` environment variable is used to set `PyConfig.base_executable`.

Changed in version 3.14: `prefix`, and `exec_prefix`, are now set to the `pyvenv.cfg` directory. This was previously done by `site`, therefore affected by `-S`.

Py_GetArgcArgv()

void Py_GetArgcArgv(int *argc, wchar_t ***argv)

Get the original command line arguments, before Python modified them.

See also `PyConfig.orig_argv` member.

Delaying main module execution

In some embedding use cases, it may be desirable to separate interpreter initialization from the execution of the main module.

This separation can be achieved by setting `PyConfig.run_command` to the empty string during initialization (to prevent the interpreter from dropping into the interactive prompt), and then subsequently executing the desired main module code using `__main__.__dict__` as the global namespace.

Memory Management

Overview

Memory management in Python involves a private heap containing all Python objects and data structures. The management of this private heap is ensured internally by the *Python memory manager*. The Python memory manager has different components which deal with various dynamic storage management aspects, like sharing, segmentation, preallocation or caching.

At the lowest level, a raw memory allocator ensures that there is enough room in the private heap for storing all Python-related data by interacting with the memory manager of the operating system. On top of the raw memory allocator, several object-specific allocators operate on the same heap and implement distinct memory management policies adapted to the peculiarities of every object type. For example, integer objects are managed differently within the heap than strings, tuples or dictionaries because integers imply different storage requirements and speed/space tradeoffs. The Python memory manager thus delegates some of the work to the object-specific allocators, but ensures that the latter operate within the bounds of the private heap.

It is important to understand that the management of the Python heap is performed by the interpreter itself and that the user has no control over it, even if they regularly manipulate object pointers to memory blocks inside that heap. The allocation of heap space for Python objects and other internal buffers is performed on demand by the Python memory manager through the Python/C API functions listed in this document.

To avoid memory corruption, extension writers should never try to operate on Python objects with the functions exported by the C library: **`malloc()`**, **`calloc()`**, **`realloc()`** and **`free()`**. This will result in mixed calls between the C allocator and the Python memory manager with fatal consequences, because they implement different algorithms and operate on different heaps. However, one may safely allocate and release memory blocks with the C library allocator for individual purposes, as shown in the following example:

```
PyObject *res;
char *buf = (char *) malloc(BUFSIZ); /* for I/O */

if (buf == NULL)
    return PyErr_NoMemory();
...Do some I/O operation involving buf...
res = PyBytes_FromString(buf);
free(buf); /* malloc'ed */
return res;
```

In this example, the memory request for the I/O buffer is handled by the C library allocator. The Python memory manager is involved only in the allocation of the bytes object returned as a result.

In most situations, however, it is recommended to allocate memory from the Python heap specifically because the latter is under control of the Python memory manager. For example, this is required when the interpreter is extended with new object types written in C. Another reason for using the Python heap is the desire to *inform* the Python memory manager about the memory needs of the extension module. Even when the requested memory is used exclusively for internal, highly specific purposes, delegating all memory requests to the Python memory manager causes the interpreter to have a more accurate image of its memory footprint as a whole. Consequently, under certain circumstances, the Python memory manager may or may not trigger appropriate actions, like garbage collection, memory compaction or other preventive procedures. Note that by using the C library allocator as shown in the previous example, the allocated memory for the I/O buffer escapes completely the Python memory manager.

See also

The **`PYTHONMALLOC`** environment variable can be used to configure the memory allocators used by Python.

The **`PYTHONMALLOCSTATS`** environment variable can be used to print statistics of the pymalloc memory allocator every time a new pymalloc object arena is created, and on shutdown.

Allocator Domains

All allocating functions belong to one of three different “domains” (see also **`PyMemAllocatorDomain`**). These domains represent different allocation strategies and are optimized for different purposes. The specific details on how every domain allocates memory or what internal functions each domain calls is considered an implementation detail, but for debugging purposes a simplified table can be found at Default Memory Allocators. The APIs used to allocate and free a block of memory must be from the same domain. For example, **`PyMem_Free()`** must be used to free memory allocated using **`PyMem_Malloc()`**.

The three allocation domains are:

- Raw domain: intended for allocating memory for general-purpose memory buffers where the allocation *must* go to the system allocator or where the allocator can operate without an attached thread state. The memory is requested directly from the system. See Raw Memory Interface.
- “Mem” domain: intended for allocating memory for Python buffers and general-purpose memory buffers where the allocation must be performed with an attached thread state. The memory is taken from the Python private heap. See Memory Interface.
- Object domain: intended for allocating memory for Python objects. The memory is taken from the Python private heap. See Object allocators.

Note

The free-threaded build requires that only Python objects are allocated using the “object” domain and that all Python objects are allocated using that domain. This differs from the prior Python versions, where this was only a best practice and not a hard requirement.

For example, buffers (non-Python objects) should be allocated using **`PyMem_Malloc()`**, **`PyMem_RawMalloc()`**, or **`malloc()`**, but not **`PyObject_Malloc()`**.

See Memory Allocation APIs.

Raw Memory Interface

The following function sets are wrappers to the system allocator. These functions are thread-safe, so a thread state does not need to be attached.

The default raw memory allocator uses the following functions: **`malloc()`**, **`calloc()`**, **`realloc()`** and **`free()`**; call `malloc(1)` (or `calloc(1, 1)`) when requesting zero bytes.

Added in version 3.4.

void *PyMem_RawMalloc(size_t n)

Part of the Stable ABI since version 3.13.
Allocates *n* bytes and returns a pointer of type void* to the allocated memory, or `NULL` if the request fails.

Requesting zero bytes returns a distinct non-`NULL` pointer if possible, as if `PyMem_RawMalloc(1)` had been called instead. The memory will not have been initialized in any way.

void *PyMem_RawCalloc(size_t nelem, size_t elsize)

Part of the Stable ABI since version 3.13.
Allocates *nelem* elements each whose size in bytes is *elsize* and returns a pointer of type void* to the allocated memory, or `NULL` if the request fails. The memory is initialized to zeros.

Requesting zero elements or elements of size zero bytes returns a distinct non-`NULL` pointer if possible, as if `PyMem_RawCalloc(1, 1)` had been called instead.

Added in version 3.5.

void *PyMem_RawRealloc(void *p, size_t n)

Part of the Stable ABI since version 3.13.
Resizes the memory block pointed to by *p* to *n* bytes. The contents will be unchanged to the minimum of the old and the new sizes.

If *p* is `NULL`, the call is equivalent to `PyMem_RawMalloc(n)`; else if *n* is equal to zero, the memory block is resized but is not freed, and the returned pointer is non-`NULL`.

Unless *p* is `NULL`, it must have been returned by a previous call to **`PyMem_RawMalloc()`**, **`PyMem_RawRealloc()`** or **`PyMem_RawCalloc()`**.

If the request fails, **`PyMem_RawRealloc()`** returns `NULL` and *p* remains a valid pointer to the previous memory area.

void PyMem_RawFree(void *p)

Part of the Stable ABI since version 3.13.
Frees the memory block pointed to by *p*, which must have been returned by a previous call to **`PyMem_RawMalloc()`**, **`PyMem_RawRealloc()`** or **`PyMem_RawCalloc()`**. Otherwise, or if `PyMem_RawFree(p)` has been called before, undefined behavior occurs.

If *p* is `NULL`, no operation is performed.

Memory Interface

The following function sets, modeled after the ANSI C standard, but specifying behavior when requesting zero bytes, are available for allocating and releasing memory from the

Python heap.

The default memory allocator uses the pymalloc memory allocator.

Warning

There must be an attached thread state when using these functions.

Changed in version 3.6: The default allocator is now pymalloc instead of system **`malloc()`**.

void *PyMem_Malloc(size_t n)

Part of the Stable ABI.

Allocates *n* bytes and returns a pointer of type void* to the allocated memory, or `NULL` if the request fails.

Requesting zero bytes returns a distinct non-`NULL` pointer if possible, as if `PyMem_Malloc(1)` had been called instead. The memory will not have been initialized in any way.

void *PyMem_Calloc(size_t nelem, size_t elsize)

Part of the Stable ABI since version 3.7.

Allocates *nelem* elements each whose size in bytes is *elsize* and returns a pointer of type void* to the allocated memory, or `NULL` if the request fails. The memory is initialized to zeros.

Requesting zero elements or elements of size zero bytes returns a distinct non-`NULL` pointer if possible, as if `PyMem_Calloc(1, 1)` had been called instead.

Added in version 3.5.

void *PyMem_Realloc(void *p, size_t n)

Part of the Stable ABI.

Resizes the memory block pointed to by *p* to *n* bytes. The contents will be unchanged to the minimum of the old and the new sizes.

If *p* is `NULL`, the call is equivalent to `PyMem_Malloc(n)`; else if *n* is equal to zero, the memory block is resized but is not freed, and the returned pointer is non-`NULL`.

Unless *p* is `NULL`, it must have been returned by a previous call to **`PyMem_Malloc()`**, **`PyMem_Realloc()`** or **`PyMem_Calloc()`**.

If the request fails, **`PyMem_Realloc()`** returns `NULL` and *p* remains a valid pointer to the previous memory area.

void PyMem_Free(void *p)

Part of the Stable ABI.

Frees the memory block pointed to by *p*, which must have been returned by a previous call to **`PyMem_Malloc()`**, **`PyMem_Realloc()`** or **`PyMem_Calloc()`**. Otherwise, or if `PyMem_Free(p)` has been called before, undefined behavior occurs.

If *p* is `NULL`, no operation is performed.

The following type-oriented macros are provided for convenience. Note that *TYPE* refers to any C type.

PyMem_New(TYPE, n)

Same as **`PyMem_Malloc()`**, but allocates `(n * sizeof(TYPE))` bytes of memory. Returns a pointer cast to `TYPE*`. The memory will not have been initialized in any way.

PyMem_Resize(p, TYPE, n)

Same as **`PyMem_Realloc()`**, but the memory block is resized to `(n * sizeof(TYPE))` bytes. Returns a pointer cast to `TYPE*`. On return, *p* will be a pointer to the new memory area, or `NULL` in the event of failure.

This is a C preprocessor macro; *p* is always reassigned. Save the original value of *p* to avoid losing memory when handling errors.

void PyMem_Del(void *p)

Same as **`PyMem_Free()`**.

In addition, the following macro sets are provided for calling the Python memory allocator directly, without involving the C API functions listed above. However, note that their use does not preserve binary compatibility across Python versions and is therefore deprecated in extension modules.

- `PyMem_MALLOC(size)`
- `PyMem_NEW(type, size)`
- `PyMem_REALLOC(ptr, size)`
- `PyMem_RESIZE(ptr, type, size)`
- `PyMem_FREE(ptr)`
- `PyMem_DEL(ptr)`

Object allocators

The following function sets, modeled after the ANSI C standard, but specifying behavior when requesting zero bytes, are available for allocating and releasing memory from the Python heap.

Note

There is no guarantee that the memory returned by these allocators can be successfully cast to a Python object when intercepting the allocating functions in this domain by the methods described in the Customize Memory Allocators section.

The default object allocator uses the pymalloc memory allocator.

Warning

There must be an attached thread state when using these functions.

void *PyObject_Malloc(size_t n)

Part of the Stable ABI.
Allocates *n* bytes and returns a pointer of type void* to the allocated memory, or `NULL` if the request fails.

Requesting zero bytes returns a distinct non-`NULL` pointer if possible, as if `PyObject_Malloc(1)` had been called instead. The memory will not have been initialized in any way.

void *PyObject_Calloc(size_t nelem, size_t elsize)

Part of the Stable ABI since version 3.7.
Allocates *nelem* elements each whose size in bytes is *elsize* and returns a pointer of type void* to the allocated memory, or `NULL` if the request fails. The memory is initialized to zeros.

Requesting zero elements or elements of size zero bytes returns a distinct non-`NULL` pointer if possible, as if `PyObject_Calloc(1, 1)` had been called instead.

Added in version 3.5.

void *PyObject_Realloc(void *p, size_t n)

Part of the Stable ABI.
Resizes the memory block pointed to by *p* to *n* bytes. The contents will be unchanged to the minimum of the old and the new sizes.

If *p* is `NULL`, the call is equivalent to `PyObject_Malloc(n)`; else if *n* is equal to zero, the memory block is resized but is not freed, and the returned pointer is non-`NULL`.

Unless *p* is `NULL`, it must have been returned by a previous call to **`PyObject_Malloc()`**, **`PyObject_Realloc()`** or **`PyObject_Calloc()`**.

If the request fails, **`PyObject_Realloc()`** returns `NULL` and *p* remains a valid pointer to the previous memory area.

void PyObject_Free(void *p)

Part of the Stable ABI.
Frees the memory block pointed to by *p*, which must have been returned by a previous call to **`PyObject_Malloc()`**, **`PyObject_Realloc()`** or **`PyObject_Calloc()`**. Otherwise, or if `PyObject_Free(p)` has been called before, undefined behavior occurs.

If *p* is `NULL`, no operation is performed.

Do not call this directly to free an object’s memory; call the type’s **`tp_free`** slot instead.

Do not use this for memory allocated by **`PyObject_GC_New`** or **`PyObject_GC_NewVar`**; use **`PyObject_GC_Del()`** instead.

See also

- **`PyObject_GC_Del()`** is the equivalent of this function for memory allocated by types that support garbage collection.
- **`PyObject_Malloc()`**
- **`PyObject_Realloc()`**
- **`PyObject_Calloc()`**
- **`PyObject_New`**
- **`PyObject_NewVar`**
- **`PyType_GenericAlloc()`**
- **`tp_free`**

Default Memory Allocators

Default memory allocators:

Configuration	**Name**	**PyMem_RawMalloc**	**PyMem_Malloc**	**PyObject_Malloc**
Release build	`"pymalloc"`	`malloc`	`pymalloc`	`pymalloc`
Debug build	`"pymalloc_debug"`	`malloc` + debug	`pymalloc` + debug	`pymalloc` + debug
Release build, without pymalloc	`"malloc"`	`malloc`	`malloc`	`malloc`
Debug build, without pymalloc	`"malloc_debug"`	`malloc` + debug	`malloc` + debug	`malloc` + debug

Legend:

- Name: value for **`PYTHONMALLOC`** environment variable.
- `malloc`: system allocators from the standard C library, C functions: **`malloc()`**, **`calloc()`**, **`realloc()`** and **`free()`**.
- `pymalloc`: pymalloc memory allocator.
- `mimalloc`: mimalloc memory allocator. The pymalloc allocator will be used if mimalloc support isn't available.
- "+ debug": with debug hooks on the Python memory allocators.
- "Debug build": Python build in debug mode.

Customize Memory Allocators

Added in version 3.4.

type PyMemAllocatorEx

Structure used to describe a memory block allocator. The structure has the following fields:

Field	**Meaning**
`void *ctx`	user context passed as first argument
`void* malloc(void *ctx, size_t size)`	allocate a memory block
`void* calloc(void *ctx, size_t nelem, size_t elsize)`	allocate a memory block initialized with zeros
`void* realloc(void *ctx, void *ptr, size_t new_size)`	allocate or resize a memory block

Field	**Meaning**
`void free(void *ctx, void *ptr)`	free a memory block

Changed in version 3.5: The **`PyMemAllocator`** structure was renamed to **`PyMemAllocatorEx`** and a new `calloc` field was added.

type PyMemAllocatorDomain

Enum used to identify an allocator domain. Domains:

PYMEM_DOMAIN_RAW

Functions:

- **`PyMem_RawMalloc()`**
- **`PyMem_RawRealloc()`**
- **`PyMem_RawCalloc()`**
- **`PyMem_RawFree()`**

PYMEM_DOMAIN_MEM

Functions:

- **`PyMem_Malloc()`**,
- **`PyMem_Realloc()`**
- **`PyMem_Calloc()`**
- **`PyMem_Free()`**

PYMEM_DOMAIN_OBJ

Functions:

- **`PyObject_Malloc()`**
- **`PyObject_Realloc()`**
- **`PyObject_Calloc()`**
- **`PyObject_Free()`**

void PyMem_GetAllocator(PyMemAllocatorDomain domain, PyMemAllocatorEx *allocator)

Get the memory block allocator of the specified domain.

void PyMem_SetAllocator(PyMemAllocatorDomain domain, PyMemAllocatorEx *allocator)

Set the memory block allocator of the specified domain.

The new allocator must return a distinct non-`NULL` pointer when requesting zero bytes.

For the **`PYMEM_DOMAIN_RAW`** domain, the allocator must be thread-safe: a thread state is not attached when the allocator is called.

For the remaining domains, the allocator must also be thread-safe: the allocator may be called in different interpreters that do not share a GIL.

If the new allocator is not a hook (does not call the previous allocator), the **`PyMem_SetupDebugHooks()`** function must be called to reinstall the debug hooks on top on the new allocator.

See also **`PyPreConfig.allocator`** and Preinitialize Python with PyPreConfig.

Warning

`PyMem_SetAllocator()` does have the following contract:

- It can be called after `Py_PreInitialize()` and before `Py_InitializeFromConfig()` to install a custom memory allocator. There are no restrictions over the installed allocator other than the ones imposed by the domain (for instance, the Raw Domain allows the allocator to be called without an attached thread state). See the section on allocator domains for more information.
- If called after Python has finish initializing (after `Py_InitializeFromConfig()` has been called) the allocator **must** wrap the existing allocator. Substituting the current allocator for some other arbitrary one is **not supported**.

Changed in version 3.12: All allocators must be thread-safe.

void PyMem_SetupDebugHooks(void)

Setup debug hooks in the Python memory allocators to detect memory errors.

Debug hooks on the Python memory allocators

When Python is built in debug mode, the `PyMem_SetupDebugHooks()` function is called at the Python preinitialization to setup debug hooks on Python memory allocators to detect memory errors.

The `PYTHONMALLOC` environment variable can be used to install debug hooks on a Python compiled in release mode (ex: `PYTHONMALLOC=debug`).

The `PyMem_SetupDebugHooks()` function can be used to set debug hooks after calling `PyMem_SetAllocator()`.

These debug hooks fill dynamically allocated memory blocks with special, recognizable bit patterns. Newly allocated memory is filled with the byte `0xCD` (`PYMEM_CLEANBYTE`), freed memory is filled with the byte `0xDD` (`PYMEM_DEADBYTE`). Memory blocks are surrounded by “forbidden bytes” filled with the byte `0xFD` (`PYMEM_FORBIDDENBYTE`). Strings of these bytes are unlikely to be valid addresses, floats, or ASCII strings.

Runtime checks:

- Detect API violations. For example, detect if `PyObject_Free()` is called on a memory block allocated by `PyMem_Malloc()`.
- Detect write before the start of the buffer (buffer underflow).
- Detect write after the end of the buffer (buffer overflow).
- Check that there is an attached thread state when allocator functions of `PYMEM_DOMAIN_OBJ` (ex: `PyObject_Malloc()`) and `PYMEM_DOMAIN_MEM` (ex: `PyMem_Malloc()`) domains are called.

On error, the debug hooks use the `tracemalloc` module to get the traceback where a memory block was allocated. The traceback is only displayed if `tracemalloc` is tracing Python memory allocations and the memory block was traced.

Let *S* = `sizeof(size_t)`. `2*S` bytes are added at each end of each block of *N* bytes requested. The memory layout is like so, where p represents the address returned by a malloc-like or realloc-like function (`p[i:j]` means the slice of bytes from `*(p+i)` inclusive up to `*(p+j)` exclusive; note that the treatment of negative indices differs from a Python slice):

`p[-2*S:-S]`
Number of bytes originally asked for. This is a size_t, big-endian (easier to read in a memory dump).

`p[-S]`
API identifier (ASCII character):

- `'r'` for **`PYMEM_DOMAIN_RAW`**.
- `'m'` for **`PYMEM_DOMAIN_MEM`**.
- `'o'` for **`PYMEM_DOMAIN_OBJ`**.

`p[-S+1:0]`
Copies of PYMEM_FORBIDDENBYTE. Used to catch under- writes and reads.

`p[0:N]`
The requested memory, filled with copies of PYMEM_CLEANBYTE, used to catch reference to uninitialized memory. When a realloc-like function is called requesting a larger memory block, the new excess bytes are also filled with PYMEM_CLEANBYTE. When a free-like function is called, these are overwritten with PYMEM_DEADBYTE, to catch reference to freed memory. When a realloc- like function is called requesting a smaller memory block, the excess old bytes are also filled with PYMEM_DEADBYTE.

`p[N:N+S]`
Copies of PYMEM_FORBIDDENBYTE. Used to catch over- writes and reads.

`p[N+S:N+2*S]`
Only used if the `PYMEM_DEBUG_SERIALNO` macro is defined (not defined by default).

A serial number, incremented by 1 on each call to a malloc-like or realloc-like function. Big-endian **`size_t`**. If “bad memory” is detected later, the serial number gives an excellent way to set a breakpoint on the next run, to capture the instant at which this block was passed out. The static function bumpserialno() in obmalloc.c is the only place the serial number is incremented, and exists so you can set such a breakpoint easily.

A realloc-like or free-like function first checks that the PYMEM_FORBIDDENBYTE bytes at each end are intact. If they’ve been altered, diagnostic output is written to stderr, and the program is aborted via Py_FatalError(). The other main failure mode is provoking a memory error when a program reads up one of the special bit patterns and tries to use it as an address. If you get in a debugger then and look at the object, you’re likely to see that it’s entirely filled with PYMEM_DEADBYTE (meaning freed memory is getting used) or PYMEM_CLEANBYTE (meaning uninitialized memory is getting used).

Changed in version 3.6: The **`PyMem_SetupDebugHooks()`** function now also works on Python compiled in release mode. On error, the debug hooks now use **`tracemalloc`** to get the traceback where a memory block was allocated. The debug hooks now also check if there is an attached thread state when functions of **`PYMEM_DOMAIN_OBJ`** and **`PYMEM_DOMAIN_MEM`** domains are called.

Changed in version 3.8: Byte patterns `0xCB` (`PYMEM_CLEANBYTE`), `0xDB` (`PYMEM_DEADBYTE`) and `0xFB` (`PYMEM_FORBIDDENBYTE`) have been replaced with `0xCD`, `0xDD` and `0xFD` to use the same values than Windows CRT debug `malloc()` and `free()`.

The pymalloc allocator

Python has a *pymalloc* allocator optimized for small objects (smaller or equal to 512 bytes) with a short lifetime. It uses memory mappings called “arenas” with a fixed size of either 256 KiB on 32-bit platforms or 1 MiB on 64-bit platforms. It falls back to `PyMem_RawMalloc()` and `PyMem_RawRealloc()` for allocations larger than 512 bytes.

pymalloc is the default allocator of the `PYMEM_DOMAIN_MEM` (ex: `PyMem_Malloc()`) and `PYMEM_DOMAIN_OBJ` (ex: `PyObject_Malloc()`) domains.

The arena allocator uses the following functions:

- `VirtualAlloc()` and `VirtualFree()` on Windows,
- `mmap()` and `munmap()` if available,
- `malloc()` and `free()` otherwise.

This allocator is disabled if Python is configured with the `--without-pymalloc` option. It can also be disabled at runtime using the `PYTHONMALLOC` environment variable (ex: `PYTHONMALLOC=malloc`).

Typically, it makes sense to disable the pymalloc allocator when building Python with AddressSanitizer (`--with-address-sanitizer`) which helps uncover low level bugs within the C code.

Customize pymalloc Arena Allocator

Added in version 3.4.

type PyObjectArenaAllocator

Structure used to describe an arena allocator. The structure has three fields:

Field	**Meaning**
`void *ctx`	user context passed as first argument
`void* alloc(void *ctx, size_t size)`	allocate an arena of size bytes
`void free(void *ctx, void *ptr, size_t size)`	free an arena

void PyObject_GetArenaAllocator(PyObjectArenaAllocator *allocator)

Get the arena allocator.

void PyObject_SetArenaAllocator(PyObjectArenaAllocator *allocator)

Set the arena allocator.

The mimalloc allocator

Added in version 3.13.

Python supports the mimalloc allocator when the underlying platform support is available. mimalloc “is a general purpose allocator with excellent performance characteristics. Initially developed by Daan Leijen for the runtime systems of the Koka and Lean languages.”

tracemalloc C API

Added in version 3.7.

int PyTraceMalloc_Track(unsigned int domain, uintptr_t ptr, size_t size)

Track an allocated memory block in the **`tracemalloc`** module.

Return `0` on success, return `-1` on error (failed to allocate memory to store the trace). Return `-2` if tracemalloc is disabled.

If memory block is already tracked, update the existing trace.

int PyTraceMalloc_Untrack(unsigned int domain, uintptr_t ptr)

Untrack an allocated memory block in the **`tracemalloc`** module. Do nothing if the block was not tracked.

Return `-2` if tracemalloc is disabled, otherwise return `0`.

Examples

Here is the example from section Overview, rewritten so that the I/O buffer is allocated from the Python heap by using the first function set:

```
PyObject *res;
char *buf = (char *) PyMem_Malloc(BUFSIZ); /* for I/O */

if (buf == NULL)
    return PyErr_NoMemory();
/* ...Do some I/O operation involving buf... */
res = PyBytes_FromString(buf);
PyMem_Free(buf); /* allocated with PyMem_Malloc */
return res;
```

The same code using the type-oriented function set:

```
PyObject *res;
char *buf = PyMem_New(char, BUFSIZ); /* for I/O */

if (buf == NULL)
    return PyErr_NoMemory();
/* ...Do some I/O operation involving buf... */
res = PyBytes_FromString(buf);
PyMem_Free(buf); /* allocated with PyMem_New */
return res;
```

Note that in the two examples above, the buffer is always manipulated via functions belonging to the same set. Indeed, it is required to use the same memory API family for a given memory block, so that the risk of mixing different allocators is reduced to a minimum. The following code sequence contains two errors, one of which is labeled as *fatal* because it mixes two different allocators operating on different heaps.

```
char *buf1 = PyMem_New(char, BUFSIZ);
char *buf2 = (char *) malloc(BUFSIZ);
char *buf3 = (char *) PyMem_Malloc(BUFSIZ);
...
PyMem_Del(buf3);  /* Wrong -- should be PyMem_Free() */
free(buf2);       /* Right -- allocated via malloc() */
free(buf1);       /* Fatal -- should be PyMem_Free()  */
```

In addition to the functions aimed at handling raw memory blocks from the Python heap, objects in Python are allocated and released with **`PyObject_New`**, **`PyObject_NewVar`** and

`PyObject_Free()`.

These will be explained in the next chapter on defining and implementing new object types in C.

Object Implementation Support

This chapter describes the functions, types, and macros used when defining new object types.

- Allocating Objects on the Heap
  - Deprecated aliases
- Object Life Cycle
  - Life Events
  - Cyclic Isolate Destruction
  - Functions
- Common Object Structures
  - Base object types and macros
  - Implementing functions and methods
  - Accessing attributes of extension types
    - Member flags
    - Member types
    - Defining Getters and Setters
- Type Object Structures
  - Quick Reference
    - “tp slots”
    - sub-slots
    - slot typedefs
  - PyTypeObject Definition
  - PyObject Slots
  - PyVarObject Slots
  - PyTypeObject Slots
  - Static Types
  - Heap Types
  - Number Object Structures
  - Mapping Object Structures

  - Sequence Object Structures
  - Buffer Object Structures
  - Async Object Structures
  - Slot Type typedefs
  - Examples
- Supporting Cyclic Garbage Collection
  - Controlling the Garbage Collector State
  - Querying Garbage Collector State

Allocating Objects on the Heap

PyObject *_PyObject_New(PyTypeObject *type)

Return value: New reference.

PyVarObject *_PyObject_NewVar(PyTypeObject *type, Py_ssize_t size)

Return value: New reference.

PyObject *PyObject_Init(PyObject *op, PyTypeObject *type)

Return value: Borrowed reference. Part of the Stable ABI.

Initialize a newly allocated object *op* with its type and initial reference. Returns the initialized object. Other fields of the object are not initialized. Despite its name, this function is unrelated to the object's `__init__()` method (`tp_init` slot). Specifically, this function does **not** call the object's `__init__()` method.

In general, consider this function to be a low-level routine. Use `tp_alloc` where possible. For implementing `tp_alloc` for your type, prefer `PyType_GenericAlloc()` or `PyObject_New()`.

Note

This function only initializes the object's memory corresponding to the initial `PyObject` structure. It does not zero the rest.

PyVarObject *PyObject_InitVar(PyVarObject *op, PyTypeObject *type, Py_ssize_t size)

Return value: Borrowed reference. Part of the Stable ABI.
This does everything `PyObject_Init()` does, and also initializes the length information for a variable-size object.

Note

This function only initializes some of the object's memory. It does not zero the rest.

PyObject_New(TYPE, typeobj)

Allocates a new Python object using the C structure type *TYPE* and the Python type object *typeobj* (`PyTypeObject*`) by calling `PyObject_Malloc()` to allocate memory and initializing it like `PyObject_Init()`. The caller will own the only reference to the object (i.e. its reference count will be one).

Avoid calling this directly to allocate memory for an object; call the type's `tp_alloc` slot instead.

When populating a type's `tp_alloc` slot, `PyType_GenericAlloc()` is preferred over a custom function that simply calls this macro.

This macro does not call `tp_alloc`, `tp_new` (`__new__()`), or `tp_init` (`__init__()`).

This cannot be used for objects with `Py_TPFLAGS_HAVE_GC` set in `tp_flags`; use `PyObject_GC_New` instead.

Memory allocated by this macro must be freed with `PyObject_Free()` (usually called via the object's `tp_free` slot).

Note

The returned memory is not guaranteed to have been completely zeroed before it was initialized.

Note

This macro does not construct a fully initialized object of the given type; it merely allocates memory and prepares it for further initialization by **`tp_init`**. To construct a fully initialized object, call *typeobj* instead. For example:

```
PyObject *foo = PyObject_CallNoArgs((PyObject
*)&PyFoo_Type);
```

See also

- **`PyObject_Free()`**
- **`PyObject_GC_New`**
- **`PyType_GenericAlloc()`**
- **`tp_alloc`**

PyObject_NewVar(TYPE, typeobj, size)

Like **`PyObject_New`** except:

- It allocates enough memory for the *TYPE* structure plus *size* (`Py_ssize_t`) fields of the size given by the **`tp_itemsize`** field of *typeobj*.
- The memory is initialized like **`PyObject_InitVar()`**.

This is useful for implementing objects like tuples, which are able to determine their size at construction time. Embedding the array of fields into the same allocation decreases the number of allocations, improving the memory management efficiency.

Avoid calling this directly to allocate memory for an object; call the type’s `tp_alloc` slot instead.

When populating a type’s `tp_alloc` slot, `PyType_GenericAlloc()` is preferred over a custom function that simply calls this macro.

This cannot be used for objects with `Py_TPFLAGS_HAVE_GC` set in `tp_flags`; use `PyObject_GC_NewVar` instead.

Memory allocated by this function must be freed with `PyObject_Free()` (usually called via the object’s `tp_free` slot).

Note

The returned memory is not guaranteed to have been completely zeroed before it was initialized.

Note

This macro does not construct a fully initialized object of the given type; it merely allocates memory and prepares it for further initialization by `tp_init`. To construct a fully initialized object, call *typeobj* instead. For example:

```
PyObject *list_instance = PyObject_CallNoArgs((PyObject
*)&PyList_Type);
```

See also

- `PyObject_Free()`
- `PyObject_GC_NewVar`

- `PyType_GenericAlloc()`
- `tp_alloc`

PyObject _Py_NoneStruct

Object which is visible in Python as `None`. This should only be accessed using the `Py_None` macro, which evaluates to a pointer to this object.

See also

Module Objects

To allocate and create extension modules.

Deprecated aliases

These are soft deprecated aliases to existing functions and macros. They exist solely for backwards compatibility.

Deprecated alias	Function
PyObject_NEW(type, typeobj)	`PyObject_New`
PyObject_NEW_VAR(type, typeobj, n)	`PyObject_NewVar`
PyObject_INIT(op, typeobj)	`PyObject_Init()`
PyObject_INIT_VAR(op, typeobj, n)	`PyObject_InitVar()`
PyObject_MALLOC(n)	`PyObject_Malloc()`

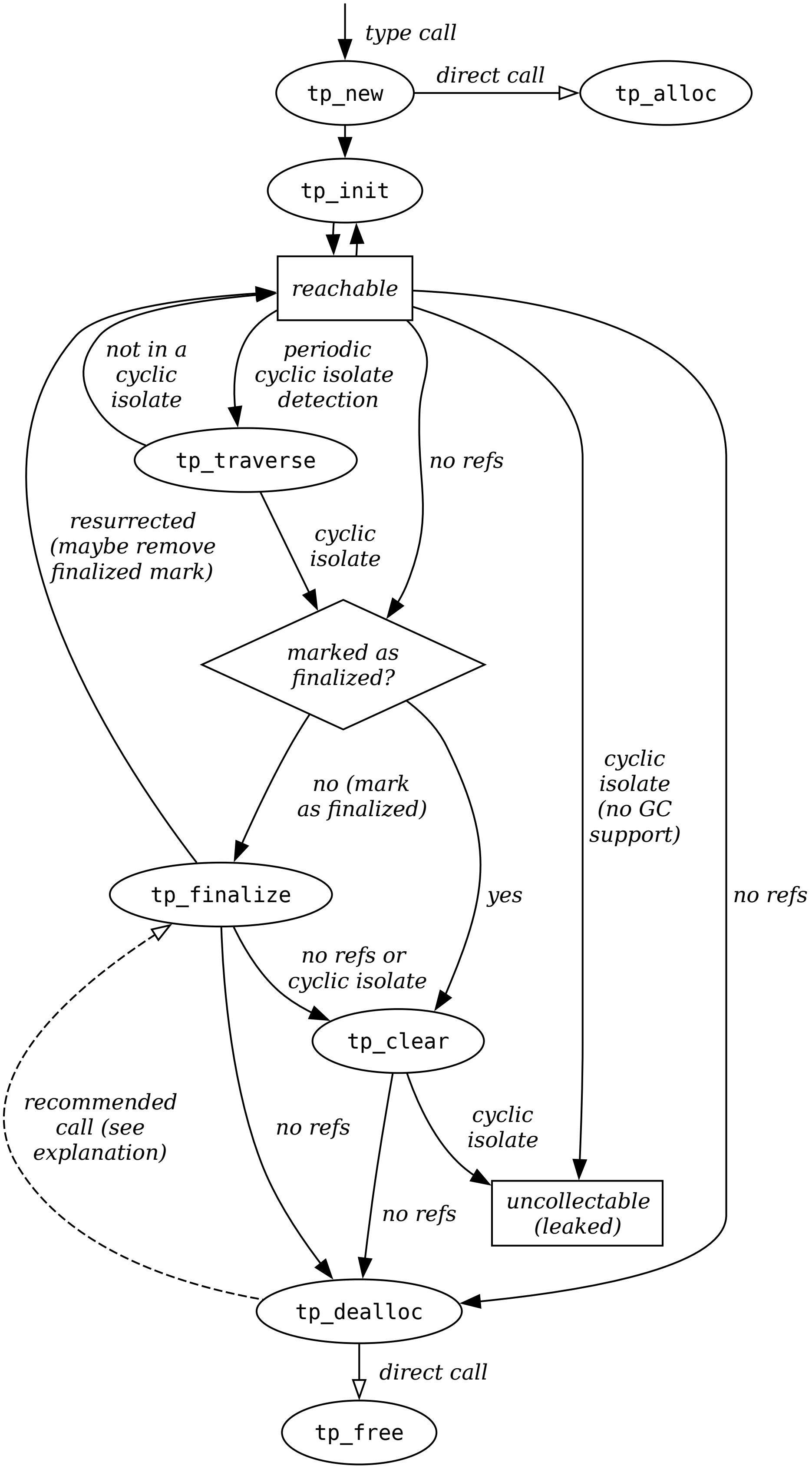
Deprecated alias	Function
PyObject_REALLOC(p, n)	`PyObject_Realloc()`
PyObject_FREE(p)	`PyObject_Free()`
PyObject_DEL(p)	`PyObject_Free()`
PyObject_Del(p)	`PyObject_Free()`

Object Life Cycle

This section explains how a type's slots relate to each other throughout the life of an object. It is not intended to be a complete canonical reference for the slots; instead, refer to the slot-specific documentation in Type Object Structures for details about a particular slot.

Life Events

The figure below illustrates the order of events that can occur throughout an object's life. An arrow from *A* to *B* indicates that event *B* can occur after event *A* has occurred, with the arrow's label indicating the condition that must be true for *B* to occur after *A*.


type call
tp_new
direct call
tp_alloc
tp_init
reachable
not in a cyclic isolate
periodic cyclic isolate detection
no refs
tp_traverse
resurrected (maybe remove finalized mark)
cyclic isolate
marked as finalized?
no (mark as finalized)
yes
cyclic isolate (no GC support)
no refs
tp_finalize
no refs or cyclic isolate
tp_clear
recommended call (see explanation)
no refs
cyclic isolate
no refs
uncollectable (leaked)
tp_dealloc
direct call
tp_free

Explanation:

- When a new object is constructed by calling its type:
  1. `tp_new` is called to create a new object.
  2. `tp_alloc` is directly called by `tp_new` to allocate the memory for the new object.
  3. `tp_init` initializes the newly created object. **`tp_init`** can be called again to re-initialize an object, if desired. The **`tp_init`** call can also be skipped entirely, for example by Python code calling `__new__()`.
- After **`tp_init`** completes, the object is ready to use.
- Some time after the last reference to an object is removed:
  1. If an object is not marked as *finalized*, it might be finalized by marking it as *finalized* and calling its `tp_finalize` function. Python does *not* finalize an object when the last reference to it is deleted; use `PyObject_CallFinalizerFromDealloc()` to ensure that `tp_finalize` is always called.
  2. If the object is marked as finalized, `tp_clear` might be called by the garbage collector to clear references held by the object. It is *not* called when the object's reference count reaches zero.
  3. `tp_dealloc` is called to destroy the object. To avoid code duplication, `tp_dealloc` typically calls into `tp_clear` to free up the object's references.
  4. When `tp_dealloc` finishes object destruction, it directly calls `tp_free` (usually set to `PyObject_Free()` or `PyObject_GC_Del()` automatically as appropriate for the type) to deallocate the memory.
- The `tp_finalize` function is permitted to add a reference to the object if desired. If it does, the object is *resurrected*, preventing its pending destruction. (Only **`tp_finalize`** is allowed to resurrect an object; `tp_clear` and `tp_dealloc` cannot without calling into **`tp_finalize`**.)

  Resurrecting an object may or may not cause the object's *finalized* mark to be removed. Currently, Python does not remove the *finalized* mark from a resurrected object if it supports garbage collection (i.e., the `Py_TPFLAGS_HAVE_GC` flag is set) but does remove the mark if the object does not support garbage collection; either or both of these behaviors may change in the future.
- `tp_dealloc` can optionally call `tp_finalize` via `PyObject_CallFinalizerFromDealloc()` if it wishes to reuse that code to help with object destruction. This is recommended because it guarantees that `tp_finalize` is always called before destruction. See the `tp_dealloc` documentation for example code.
- If the object is a member of a cyclic isolate and either `tp_clear` fails to break the reference cycle or the cyclic isolate is not detected (perhaps `gc.disable()` was called, or the `Py_TPFLAGS_HAVE_GC` flag was erroneously omitted in one of the involved types), the objects remain indefinitely uncollectable (they "leak"). See `gc.garbage`.

If the object is marked as supporting garbage collection (the `Py_TPFLAGS_HAVE_GC` flag is set in `tp_flags`), the following events are also possible:

- The garbage collector occasionally calls `tp_traverse` to identify cyclic isolates.
- When the garbage collector discovers a cyclic isolate, it finalizes one of the objects in the group by marking it as *finalized* and calling its `tp_finalize` function, if it has one. This repeats until the cyclic isolate doesn't exist or all of the objects have been finalized.
- `tp_finalize` is permitted to resurrect the object by adding a reference from outside the cyclic isolate. The new reference causes the group of objects to no longer form a cyclic isolate (the reference cycle may still exist, but if it does the objects are no longer isolated).

- When the garbage collector discovers a cyclic isolate and all of the objects in the group have already been marked as *finalized*, the garbage collector clears one or more of the uncleared objects in the group (possibly concurrently) by calling each's `tp_clear` function. This repeats as long as the cyclic isolate still exists and not all of the objects have been cleared.

Cyclic Isolate Destruction

Listed below are the stages of life of a hypothetical cyclic isolate that continues to exist after each member object is finalized or cleared. It is a memory leak if a cyclic isolate progresses through all of these stages; it should vanish once all objects are cleared, if not sooner. A cyclic isolate can vanish either because the reference cycle is broken or because the objects are no longer isolated due to finalizer resurrection (see `tp_finalize`).

0. **Reachable** (not yet a cyclic isolate): All objects are in their normal, reachable state. A reference cycle could exist, but an external reference means the objects are not yet isolated.
1. **Unreachable but consistent:** The final reference from outside the cyclic group of objects has been removed, causing the objects to become isolated (thus a cyclic isolate is born). None of the group's objects have been finalized or cleared yet. The cyclic isolate remains at this stage until some future run of the garbage collector (not necessarily the next run because the next run might not scan every object).
2. **Mix of finalized and not finalized:** Objects in a cyclic isolate are finalized one at a time, which means that there is a period of time when the cyclic isolate is composed of a mix of finalized and non-finalized

objects. Finalization order is unspecified, so it can appear random. A finalized object must behave in a sane manner when non-finalized objects interact with it, and a non-finalized object must be able to tolerate the finalization of an arbitrary subset of its referents.

3. **All finalized:** All objects in a cyclic isolate are finalized before any of them are cleared.
4. **Mix of finalized and cleared:** The objects can be cleared serially or concurrently (but with the GIL held); either way, some will finish before others. A finalized object must be able to tolerate the clearing of a subset of its referents. **PEP 442** [https://peps.python.org/pep-0442/] calls this stage “cyclic trash”.
5. **Leaked:** If a cyclic isolate still exists after all objects in the group have been finalized and cleared, then the objects remain indefinitely uncollectable (see `gc.garbage`). It is a bug if a cyclic isolate reaches this stage—it means the `tp_clear` methods of the participating objects have failed to break the reference cycle as required.

If `tp_clear` did not exist, then Python would have no way to safely break a reference cycle. Simply destroying an object in a cyclic isolate would result in a dangling pointer, triggering undefined behavior when an object referencing the destroyed object is itself destroyed. The clearing step makes object destruction a two-phase process: first `tp_clear` is called to partially destroy the objects enough to detangle them from each other, then `tp_dealloc` is called to complete the destruction.

Unlike clearing, finalization is not a phase of destruction. A finalized object must still behave properly by continuing to fulfill its design contracts. An object’s finalizer is allowed to execute arbitrary Python code, and is even allowed to prevent the impending destruction by adding a reference.

The finalizer is only related to destruction by call order—if it runs, it runs before destruction, which starts with **`tp_clear`** (if called) and concludes with **`tp_dealloc`**.

The finalization step is not necessary to safely reclaim the objects in a cyclic isolate, but its existence makes it easier to design types that behave in a sane manner when objects are cleared. Clearing an object might necessarily leave it in a broken, partially destroyed state—it might be unsafe to call any of the cleared object's methods or access any of its attributes. With finalization, only finalized objects can possibly interact with cleared objects; non-finalized objects are guaranteed to interact with only non-cleared (but potentially finalized) objects.

To summarize the possible interactions:

- A non-finalized object might have references to or from non-finalized and finalized objects, but not to or from cleared objects.
- A finalized object might have references to or from non-finalized, finalized, and cleared objects.
- A cleared object might have references to or from finalized and cleared objects, but not to or from non-finalized objects.

Without any reference cycles, an object can be simply destroyed once its last reference is deleted; the finalization and clearing steps are not necessary to safely reclaim unused objects. However, it can be useful to automatically call **`tp_finalize`** and **`tp_clear`** before destruction anyway because type design is simplified when all objects always experience the same series of events regardless of whether they participated in a cyclic isolate. Python currently only calls **`tp_finalize`** and **`tp_clear`** as needed to destroy a cyclic isolate; this may change in a future version.

Functions

To allocate and free memory, see Allocating Objects on the Heap.

void PyObject_CallFinalizer(PyObject *op)

Finalizes the object as described in `tp_finalize`. Call this function (or `PyObject_CallFinalizerFromDealloc()`) instead of calling `tp_finalize` directly because this function may deduplicate multiple calls to `tp_finalize`. Currently, calls are only deduplicated if the type supports garbage collection (i.e., the `Py_TPFLAGS_HAVE_GC` flag is set); this may change in the future.

int PyObject_CallFinalizerFromDealloc(PyObject *op)

Same as `PyObject_CallFinalizer()` but meant to be called at the beginning of the object's destructor (`tp_dealloc`). There must not be any references to the object. If the object's finalizer resurrects the object, this function returns -1; no further destruction should happen. Otherwise, this function returns 0 and destruction can continue normally.

See also

`tp_dealloc` for example code.

Common Object Structures

There are a large number of structures which are used in the definition of object types for Python. This section describes these structures and how they are used.

Base object types and macros

All Python objects ultimately share a small number of fields at the beginning of the object's representation in memory. These are represented by the **`PyObject`** and **`PyVarObject`** types, which are defined, in turn, by the expansions of some macros also used, whether directly or indirectly, in the definition of all other Python objects. Additional macros can be found under reference counting.

type PyObject

Part of the Limited API. (Only some members are part of the stable ABI.)
All object types are extensions of this type. This is a type which contains the information Python needs to treat a pointer to an object as an object. In a normal "release" build, it contains only the object's reference count and a pointer to the corresponding type object. Nothing is actually declared to be a **`PyObject`**, but every pointer to a Python object can be cast to a PyObject*.

The members must not be accessed directly; instead use macros such as **`Py_REFCNT`** and **`Py_TYPE`**.

Py_ssize_t ob_refcnt

Part of the Stable ABI.

The object’s reference count, as returned by **`Py_REFCNT`**. Do not use this field directly; instead use functions and macros such as **`Py_REFCNT`**, **`Py_INCREF()`** and **`Py_DecRef()`**.

The field type may be different from `Py_ssize_t`, depending on build configuration and platform.

PyTypeObject *ob_type

Part of the Stable ABI.
The object’s type. Do not use this field directly; use **`Py_TYPE`** and **`Py_SET_TYPE()`** instead.

type PyVarObject

Part of the Limited API. (Only some members are part of the stable ABI.)
An extension of **`PyObject`** that adds the **`ob_size`** field. This is intended for objects that have some notion of *length*.

As with **`PyObject`**, the members must not be accessed directly; instead use macros such as **`Py_SIZE`**, **`Py_REFCNT`** and **`Py_TYPE`**.

Py_ssize_t ob_size

Part of the Stable ABI.
A size field, whose contents should be considered an object’s internal implementation detail.

Do not use this field directly; use **`Py_SIZE`** instead.

Object creation functions such as **`PyObject_NewVar()`** will generally set this field to the requested size (number of items). After creation, arbitrary values can be stored in **`ob_size`** using **`Py_SET_SIZE`**.

To get an object’s publicly exposed length, as returned by the Python function **`len()`**, use **`PyObject_Length()`**

instead.

PyObject_HEAD

This is a macro used when declaring new types which represent objects without a varying length. The PyObject_HEAD macro expands to:

```
PyObject ob_base;
```

See documentation of **`PyObject`** above.

PyObject_VAR_HEAD

This is a macro used when declaring new types which represent objects with a length that varies from instance to instance. The PyObject_VAR_HEAD macro expands to:

```
PyVarObject ob_base;
```

See documentation of **`PyVarObject`** above.

PyTypeObject PyBaseObject_Type

Part of the Stable ABI.
The base class of all other objects, the same as **`object`** in Python.

int Py_Is(PyObject *x, PyObject *y)

Part of the Stable ABI since version 3.10.
Test if the *x* object is the *y* object, the same as `x is y` in Python.

Added in version 3.10.

int Py_IsNone(PyObject *x)

Part of the Stable ABI since version 3.10.
Test if an object is the `None` singleton, the same as `x is None` in Python.

Added in version 3.10.

int Py_IsTrue(PyObject *x)

Part of the Stable ABI since version 3.10.
Test if an object is the `True` singleton, the same as `x is True` in Python.

Added in version 3.10.

int Py_IsFalse(PyObject *x)

Part of the Stable ABI since version 3.10.
Test if an object is the `False` singleton, the same as `x is False` in Python.

Added in version 3.10.

PyTypeObject *Py_TYPE(PyObject *o)

Return value: Borrowed reference. Part of the Stable ABI since version 3.14.
Get the type of the Python object *o*.

The returned reference is borrowed from *o*. Do not release it with **`Py_DECREF()`** or similar.

Changed in version 3.11: **`Py_TYPE()`** is changed to an inline static function. The parameter type is no longer const PyObject*.

int Py_IS_TYPE(PyObject *o, PyTypeObject *type)

Return non-zero if the object *o* type is *type*. Return zero otherwise. Equivalent to: `Py_TYPE(o) == type`.

Added in version 3.9.

void Py_SET_TYPE(PyObject *o, PyTypeObject *type)

Set the type of object *o* to *type*, without any checking or reference counting.

This is a very low-level operation. Consider instead setting the Python attribute **`__class__`** using **`PyObject_SetAttrString()`** or similar.

Note that assigning an incompatible type can lead to undefined behavior.

If *type* is a heap type, the caller must create a new reference to it. Similarly, if the old type of *o* is a heap type, the caller must release a reference to that type.

Added in version 3.9.

Py_ssize_t Py_SIZE(PyVarObject *o)

Get the **`ob_size`** field of *o*.

Changed in version 3.11: **`Py_SIZE()`** is changed to an inline static function. The parameter type is no longer const PyVarObject*.

void Py_SET_SIZE(PyVarObject *o, Py_ssize_t size)

Set the **`ob_size`** field of *o* to *size*.

Added in version 3.9.

PyObject_HEAD_INIT(type)

This is a macro which expands to initialization values for a new **`PyObject`** type. This macro expands to:

```
_PyObject_EXTRA_INIT
1, type,
```

PyVarObject_HEAD_INIT(type, size)

This is a macro which expands to initialization values for a new **`PyVarObject`** type, including the **`ob_size`** field. This macro expands to:

```
_PyObject_EXTRA_INIT
1, type, size,
```

Implementing functions and methods

type PyCFunction

Part of the Stable ABI.

Type of the functions used to implement most Python callables in C. Functions of this type take two PyObject* parameters and return one such value. If the return value is `NULL`, an exception shall have been set. If not `NULL`, the return value is interpreted as the return value of the function as exposed in Python. The function must return a new reference.

The function signature is:

```
PyObject *PyCFunction(PyObject *self,
                      PyObject *args);
```

type PyCFunctionWithKeywords

Part of the Stable ABI.

Type of the functions used to implement Python callables in C with signature METH_VARARGS | METH_KEYWORDS. The function signature is:

```
PyObject *PyCFunctionWithKeywords(PyObject *self,
                                  PyObject *args,
                                  PyObject *kwargs);
```

type PyCFunctionFast

Part of the Stable ABI since version 3.13.

Type of the functions used to implement Python callables in C with signature **`METH_FASTCALL`**. The function signature is:

```
PyObject *PyCFunctionFast(PyObject *self,
                          PyObject *const *args,
                          Py_ssize_t nargs);
```

type PyCFunctionFastWithKeywords

Part of the Stable ABI since version 3.13.
Type of the functions used to implement Python callables in C with signature METH_FASTCALL | METH_KEYWORDS. The function signature is:

```
PyObject *PyCFunctionFastWithKeywords(PyObject *self,
                                      PyObject *const *args,
                                      Py_ssize_t nargs,
                                      PyObject *kwnames);
```

type PyCMethod

Type of the functions used to implement Python callables in C with signature METH_METHOD | METH_FASTCALL | METH_KEYWORDS. The function signature is:

```
PyObject *PyCMethod(PyObject *self,
                    PyTypeObject *defining_class,
                    PyObject *const *args,
                    Py_ssize_t nargs,
                    PyObject *kwnames)
```

Added in version 3.9.

type PyMethodDef

Part of the Stable ABI (including all members).
Structure used to describe a method of an extension type. This structure has four fields:

const char *ml_name

Name of the method.

PyCFunction ml_meth

Pointer to the C implementation.

int ml_flags

Flags bits indicating how the call should be constructed.

const char *ml_doc

Points to the contents of the docstring.

The **`ml_meth`** is a C function pointer. The functions may be of different types, but they always return PyObject*. If the function is not of the **`PyCFunction`**, the compiler will require a cast in the method table. Even though **`PyCFunction`** defines the first parameter as PyObject*, it is common that the method implementation uses the specific C type of the *self* object.

The **`ml_flags`** field is a bitfield which can include the following flags. The individual flags indicate either a calling convention or a binding convention.

There are these calling conventions:

METH_VARARGS

Part of the Stable ABI.

This is the typical calling convention, where the methods have the type **`PyCFunction`**. The function expects two PyObject* values. The first one is the *self* object for methods; for module functions, it is the module object. The second parameter (often called *args*) is a tuple object representing all arguments. This parameter is typically processed using **`PyArg_ParseTuple()`** or **`PyArg_UnpackTuple()`**.

METH_KEYWORDS

Can only be used in certain combinations with other flags: METH_VARARGS | METH_KEYWORDS, METH_FASTCALL | METH_KEYWORDS and METH_METHOD | METH_FASTCALL | METH_KEYWORDS.

METH_VARARGS | METH_KEYWORDS

Methods with these flags must be of type `PyCFunctionWithKeywords`. The function expects three parameters: *self*, *args*, *kwargs* where *kwargs* is a dictionary of all the keyword arguments or possibly `NULL` if there are no keyword arguments. The parameters are typically processed using `PyArg_ParseTupleAndKeywords()`.

METH_FASTCALL

Part of the Stable ABI since version 3.7.
Fast calling convention supporting only positional arguments. The methods have the type `PyCFunctionFast`. The first parameter is *self*, the second parameter is a C array of PyObject* values indicating the arguments and the third parameter is the number of arguments (the length of the array).

Added in version 3.7.

Changed in version 3.10: `METH_FASTCALL` is now part of the stable ABI.

METH_FASTCALL | METH_KEYWORDS

Extension of `METH_FASTCALL` supporting also keyword arguments, with methods of type `PyCFunctionFastWithKeywords`. Keyword arguments are passed the same way as in the vectorcall protocol: there is an additional fourth PyObject* parameter which is a tuple representing the names of the keyword arguments (which are guaranteed to be strings) or possibly `NULL` if there are no keywords. The values of the keyword arguments are stored in the *args* array, after the positional arguments.

Added in version 3.7.

METH_METHOD

Part of the Stable ABI since version 3.7.
Can only be used in the combination with other flags: METH_METHOD | METH_FASTCALL | METH_KEYWORDS.

METH_METHOD | METH_FASTCALL | METH_KEYWORDS

Extension of METH_FASTCALL | METH_KEYWORDS supporting the *defining class*, that is, the class that contains the method in question. The defining class might be a superclass of `Py_TYPE(self)`.

The method needs to be of type **`PyCMethod`**, the same as for `METH_FASTCALL | METH_KEYWORDS` with `defining_class` argument added after `self`.

Added in version 3.9.

METH_NOARGS

Part of the Stable ABI.
Methods without parameters don’t need to check whether arguments are given if they are listed with the **`METH_NOARGS`** flag. They need to be of type **`PyCFunction`**. The first parameter is typically named *self* and will hold a reference to the module or object instance. In all cases the second parameter will be `NULL`.

The function must have 2 parameters. Since the second parameter is unused, **`Py_UNUSED`** can be used to prevent a compiler warning.

METH_O

Part of the Stable ABI.
Methods with a single object argument can be listed with the **`METH_O`** flag, instead of invoking **`PyArg_ParseTuple()`** with a `"O"` argument. They have the type **`PyCFunction`**, with the *self* parameter, and a PyObject* parameter representing the single argument.

These two constants are not used to indicate the calling convention but the binding when use with methods of classes. These may not be used for functions defined for modules. At most one of these flags may be set for any given method.

METH_CLASS

Part of the Stable ABI.
The method will be passed the type object as the first parameter rather than an instance of the type. This is used to create *class methods*, similar to what is created when using the **`classmethod()`** built-in function.

METH_STATIC

Part of the Stable ABI.
The method will be passed `NULL` as the first parameter rather than an instance of the type. This is used to create *static methods*, similar to what is created when using the **`staticmethod()`** built-in function.

One other constant controls whether a method is loaded in place of another definition with the same method name.

METH_COEXIST

Part of the Stable ABI.
The method will be loaded in place of existing definitions. Without *METH_COEXIST*, the default is to skip repeated definitions. Since slot wrappers are loaded before the method table, the existence of a *sq_contains* slot, for

example, would generate a wrapped method named `__contains__()` and preclude the loading of a corresponding PyCFunction with the same name. With the flag defined, the PyCFunction will be loaded in place of the wrapper object and will co-exist with the slot. This is helpful because calls to PyCFunctions are optimized more than wrapper object calls.

PyTypeObject PyCMethod_Type

The type object corresponding to Python C method objects. This is available as **`types.BuiltinMethodType`** in the Python layer.

int PyCMethod_Check(PyObject *op)

Return true if *op* is an instance of the **`PyCMethod_Type`** type or a subtype of it. This function always succeeds.

int PyCMethod_CheckExact(PyObject *op)

This is the same as **`PyCMethod_Check()`**, but does not account for subtypes.

PyObject *PyCMethod_New(PyMethodDef *ml, PyObject *self, PyObject *module, PyTypeObject *cls)

Return value: New reference. Part of the Stable ABI since version 3.9.
Turn *ml* into a Python callable object. The caller must ensure that *ml* outlives the callable. Typically, *ml* is defined as a static variable.

The *self* parameter will be passed as the *self* argument to the C function in `ml->ml_meth` when invoked. *self* can be `NULL`.

The callable object's `__module__` attribute can be set from the given *module* argument. *module* should be a Python string, which will be used as name of the module the function is defined in. If unavailable, it can be set to `None` or `NULL`.

See also

`function.__module__`

The *cls* parameter will be passed as the *defining_class* argument to the C function. Must be set if `METH_METHOD` is set on `ml->ml_flags`.

Added in version 3.9.

PyTypeObject PyCFunction_Type

Part of the Stable ABI.

The type object corresponding to Python C function objects. This is available as `types.BuiltinFunctionType` in the Python layer.

int PyCFunction_Check(PyObject *op)

Return true if *op* is an instance of the `PyCFunction_Type` type or a subtype of it. This function always succeeds.

int PyCFunction_CheckExact(PyObject *op)

This is the same as `PyCFunction_Check()`, but does not account for subtypes.

PyObject *PyCFunction_NewEx(PyMethodDef *ml, PyObject *self, PyObject *module)

Return value: New reference. Part of the Stable ABI.

Equivalent to `PyCMethod_New(ml, self, module, NULL)`.

PyObject *PyCFunction_New(PyMethodDef *ml, PyObject *self)

Return value: New reference. Part of the Stable ABI since version 3.4.
Equivalent to `PyCMethod_New(ml, self, NULL, NULL)`.

int PyCFunction_GetFlags(PyObject *func)

Part of the Stable ABI.
Get the function's flags on *func* as they were passed to **`ml_flags`**.

If *func* is not a C function object, this fails with an exception. *func* must not be `NULL`.

This function returns the function's flags on success, and `-1` with an exception set on failure.

int PyCFunction_GET_FLAGS(PyObject *func)

This is the same as **`PyCFunction_GetFlags()`**, but without error or type checking.

PyCFunction PyCFunction_GetFunction(PyObject *func)

Part of the Stable ABI.
Get the function pointer on *func* as it was passed to **`ml_meth`**.

If *func* is not a C function object, this fails with an exception. *func* must not be `NULL`.

This function returns the function pointer on success, and `NULL` with an exception set on failure.

int PyCFunction_GET_FUNCTION(PyObject *func)

This is the same as **`PyCFunction_GetFunction()`**, but without error or type checking.

PyObject *PyCFunction_GetSelf(PyObject *func)

Part of the Stable ABI.

Get the “self” object on *func*. This is the object that would be passed to the first argument of a `PyCFunction`. For C function objects created through a `PyMethodDef` on a `PyModuleDef`, this is the resulting module object.

If *func* is not a C function object, this fails with an exception. *func* must not be `NULL`.

This function returns a borrowed reference to the “self” object on success, and `NULL` with an exception set on failure.

PyObject *PyCFunction_GET_SELF(PyObject *func)

This is the same as `PyCFunction_GetSelf()`, but without error or type checking.

Accessing attributes of extension types

type PyMemberDef

Part of the Stable ABI (including all members).
Structure which describes an attribute of a type which corresponds to a C struct member. When defining a class, put a NULL-terminated array of these structures in the `tp_members` slot.

Its fields are, in order:

const char *name

Name of the member. A NULL value marks the end of a `PyMemberDef[]` array.

The string should be static, no copy is made of it.

int type

The type of the member in the C struct. See Member types for the possible values.

Py_ssize_t offset

The offset in bytes that the member is located on the type's object struct.

int flags

Zero or more of the Member flags, combined using bitwise OR.

const char *doc

The docstring, or NULL. The string should be static, no copy is made of it. Typically, it is defined using **`PyDoc_STR`**.

By default (when **`flags`** is `0`), members allow both read and write access. Use the **`Py_READONLY`** flag for read-only access. Certain types, like **`Py_T_STRING`**, imply **`Py_READONLY`**. Only **`Py_T_OBJECT_EX`** (and legacy **`T_OBJECT`**) members can be deleted.

For heap-allocated types (created using **`PyType_FromSpec()`** or similar), `PyMemberDef` may contain a definition for the special member `"__vectorcalloffset__"`, corresponding to **`tp_vectorcall_offset`** in type objects. This member must be defined with `Py_T_PYSSIZET`, and either `Py_READONLY` or `Py_READONLY | Py_RELATIVE_OFFSET`. For example:

```
static PyMemberDef spam_type_members[] = {
    {"__vectorcalloffset__", Py_T_PYSSIZET,
     offsetof(Spam_object, vectorcall), Py_READONLY},
    {NULL}  /* Sentinel */
};
```

(You may need to `#include <stddef.h>` for **`offsetof()`**.)

The legacy offsets **`tp_dictoffset`** and **`tp_weaklistoffset`** can be defined similarly using `"__dictoffset__"` and `"__weaklistoffset__"` members, but extensions are strongly encouraged to use **`Py_TPFLAGS_MANAGED_DICT`** and **`Py_TPFLAGS_MANAGED_WEAKREF`** instead.

Changed in version 3.12: `PyMemberDef` is always available. Previously, it required including `"structmember.h"`.

Changed in version 3.14: **`Py_RELATIVE_OFFSET`** is now allowed for `"__vectorcalloffset__"`, `"__dictoffset__"` and `"__weaklistoffset__"`.

PyObject *PyMember_GetOne(const char *obj_addr, struct PyMemberDef *m)

Part of the Stable ABI.
Get an attribute belonging to the object at address *obj_addr*. The attribute is described by `PyMemberDef` *m*. Returns `NULL` on error.

Changed in version 3.12: `PyMember_GetOne` is always available. Previously, it required including `"structmember.h"`.

int PyMember_SetOne(char *obj_addr, struct PyMemberDef *m, PyObject *o)

Part of the Stable ABI.
Set an attribute belonging to the object at address *obj_addr* to object *o*. The attribute to set is described by `PyMemberDef` *m*. Returns `0` if successful and a negative value on failure.

Changed in version 3.12: `PyMember_SetOne` is always available. Previously, it required including `"structmember.h"`.

Member flags

The following flags can be used with **`PyMemberDef.flags`**:

Py_READONLY

Part of the Stable ABI since version 3.12.
Not writable.

Py_AUDIT_READ

Part of the Stable ABI since version 3.12.
Emit an `object.__getattr__` audit event before reading.

Py_RELATIVE_OFFSET

Part of the Stable ABI since version 3.12.
Indicates that the **`offset`** of this `PyMemberDef` entry indicates an offset from the subclass-specific data, rather than from `PyObject`.

Can only be used as part of the **`Py_tp_members`** **`slot`** when creating a class using negative **`basicsize`**. It is mandatory in that case. When setting **`tp_members`** from the slot during class creation, Python clears the flag and sets **`PyMemberDef.offset`** to the offset from the `PyObject` struct.

Changed in version 3.10: The **`RESTRICTED`**, **`READ_RESTRICTED`** and **`WRITE_RESTRICTED`** macros available with `#include "structmember.h"` are deprecated. **`READ_RESTRICTED`** and **`RESTRICTED`** are equivalent to **`Py_AUDIT_READ`**; **`WRITE_RESTRICTED`** does nothing.

Changed in version 3.12: The **`READONLY`** macro was renamed to **`Py_READONLY`**. The **`PY_AUDIT_READ`** macro was renamed with the `Py_` prefix. The new names are now always available. Previously, these required `#include "structmember.h"`. The header is still available and it provides the old names.

Member types

`PyMemberDef.type` can be one of the following macros corresponding to various C types. When the member is accessed in Python, it will be converted to the equivalent Python type. When it is set from Python, it will be converted back to the C type. If that is not possible, an exception such as `TypeError` or `ValueError` is raised.

Unless marked (D), attributes defined this way cannot be deleted using e.g. `del` or `delattr()`.

Macro name	C type	Python type
Py_T_BYTE *Part of the Stable ABI since version 3.12.*	char	`int`
Py_T_SHORT *Part of the Stable ABI since version 3.12.*	short	`int`
Py_T_INT *Part of the Stable ABI since version 3.12.*	int	`int`
Py_T_LONG *Part of the Stable ABI since version 3.12.*	long	`int`

Macro name	C type	Python type
Py_T_LONGLONG *Part of the Stable ABI since version 3.12.*	long long	`int`
Py_T_UBYTE *Part of the Stable ABI since version 3.12.*	unsigned char	`int`
Py_T_UINT *Part of the Stable ABI since version 3.12.*	unsigned int	`int`
Py_T_USHORT *Part of the Stable ABI since version 3.12.*	unsigned short	`int`
Py_T_ULONG *Part of the Stable ABI since version 3.12.*	unsigned long	`int`
Py_T_ULONGLONG *Part of the Stable ABI since version 3.12.*	unsigned long long	`int`
Py_T_PYSSIZET	Py_ssize_t	`int`

Macro name	**C type**	**Python type**
Part of the Stable ABI since version 3.12.		
Py_T_FLOAT *Part of the Stable ABI since version 3.12.*	float	`float`
Py_T_DOUBLE *Part of the Stable ABI since version 3.12.*	double	`float`
Py_T_BOOL *Part of the Stable ABI since version 3.12.*	char (written as 0 or 1)	`bool`
Py_T_STRING *Part of the Stable ABI since version 3.12.*	const char* (*)	`str` (RO)
Py_T_STRING_INPLACE *Part of the Stable ABI since version 3.12.*	const char[] (*)	`str` (RO)
Py_T_CHAR *Part of the Stable ABI since version 3.12.*	char (0-127)	`str` (**)

Macro name	**C type**	**Python type**
Py_T_OBJECT_EX *Part of the Stable ABI since version 3.12.*	PyObject*	**`object`** (D)

(*): Zero-terminated, UTF8-encoded C string. With **`Py_T_STRING`** the C representation is a pointer; with **`Py_T_STRING_INPLACE`** the string is stored directly in the structure.

(**): String of length 1. Only ASCII is accepted.

(RO): Implies **`Py_READONLY`**.

(D): Can be deleted, in which case the pointer is set to `NULL`. Reading a `NULL` pointer raises **`AttributeError`**.

Added in version 3.12: In previous versions, the macros were only available with `#include "structmember.h"` and were named without the `Py_` prefix (e.g. as `T_INT`). The header is still available and contains the old names, along with the following deprecated types:

T_OBJECT

Like `Py_T_OBJECT_EX`, but `NULL` is converted to `None`. This results in surprising behavior in Python: deleting the attribute effectively sets it to `None`.

T_NONE

Always `None`. Must be used with **`Py_READONLY`**.

Defining Getters and Setters

type PyGetSetDef

Part of the Stable ABI (including all members).
Structure to define property-like access for a type. See also description of the `PyTypeObject.tp_getset` slot.

const char *name

attribute name

getter get

C function to get the attribute.

setter set

Optional C function to set or delete the attribute. If `NULL`, the attribute is read-only.

const char *doc

optional docstring

void *closure

Optional user data pointer, providing additional data for getter and setter.

typedef PyObject *(*getter)(PyObject*, void*)

Part of the Stable ABI.
The `get` function takes one PyObject* parameter (the instance) and a user data pointer (the associated `closure`):

It should return a new reference on success or `NULL` with a set exception on failure.

typedef int (*setter)(PyObject*, PyObject*, void*)

Part of the Stable ABI.
`set` functions take two PyObject* parameters (the instance and the value to be set) and a user data pointer (the associated `closure`):

In case the attribute should be deleted the second parameter is `NULL`. Should return `0` on success or `-1` with a set exception on failure.

Type Object Structures

Perhaps one of the most important structures of the Python object system is the structure that defines a new type: the `PyTypeObject` structure. Type objects can be handled using any of the `PyObject_*` or `PyType_*` functions, but do not offer much that's interesting to most Python applications. These objects are fundamental to how objects behave, so they are very important to the interpreter itself and to any extension module that implements new types.

Type objects are fairly large compared to most of the standard types. The reason for the size is that each type object stores a large number of values, mostly C function pointers, each of which implements a small part of the type's functionality. The fields of the type object are examined in detail in this section. The fields will be described in the order in which they occur in the structure.

In addition to the following quick reference, the Examples section provides at-a-glance insight into the meaning and use of `PyTypeObject`.

Quick Reference

"tp slots"

PyTypeObject Slot [[1]]	**Type**	**special methods/attrs**	**Info [[2]]**			
			O	**T**	**D**	**I**
<R> `tp_name`	const char *	__name__	X	X		
`tp_basicsize`	`Py_ssize_t`		X	X		X

PyTypeObject Slot [[1]]	Type	special methods/attrs	Info [[2]]			
			O	T	D	I
`tp_itemsize`	`Py_ssize_t`			X		X
`tp_dealloc`	`destructor`		X	X		X
`tp_vectorcall_offset`	`Py_ssize_t`			X		X
(`tp_getattr`)	`getattrfunc`	__getattribute__, __getattr__				G
(`tp_setattr`)	`setattrfunc`	__setattr__, __delattr__				G
`tp_as_async`	`PyAsyncMethods` *	sub-slots				%
`tp_repr`	`reprfunc`	__repr__	X	X		X
`tp_as_number`	`PyNumberMethods` *	sub-slots				%
`tp_as_sequence`	`PySequenceMethods` *	sub-slots				%
`tp_as_mapping`	`PyMappingMethods` *	sub-slots				%
`tp_hash`	`hashfunc`	__hash__	X			G

PyTypeObject Slot [[1]]	Type	special methods/attrs	Info [[2]]			
			O	T	D	I
tp_call	ternaryfunc	__call__		X		X
tp_str	reprfunc	__str__	X			X
tp_getattro	getattrofunc	__getattribute__, __getattr__	X	X		G
tp_setattro	setattrofunc	__setattr__, __delattr__	X	X		G
tp_as_buffer	PyBufferProcs *	sub-slots				%
tp_flags	unsigned long		X	X		?
tp_doc	const char *	__doc__	X	X		
tp_traverse	traverseproc			X		G
tp_clear	inquiry			X		G
tp_richcompare	richcmpfunc	__lt__, __le__, __eq__, __ne__, __gt__, __ge__	X			G

PyTypeObject Slot [[1]]	Type	special methods/attrs	Info [[2]]			
			O	T	D	I
(tp_weaklistoffset)	Py_ssize_t			X		?
tp_iter	getiterfunc	__iter__				X
tp_iternext	iternextfunc	__next__				X
tp_methods	PyMethodDef []		X	X		
tp_members	PyMemberDef []			X		
tp_getset	PyGetSetDef []		X	X		
tp_base	PyTypeObject *	__base__			X	
tp_dict	PyObject *	__dict__			?	
tp_descr_get	descrgetfunc	__get__				X
tp_descr_set	descrsetfunc	__set__, __delete__				X
(tp_dictoffset)	Py_ssize_t			X		?
tp_init	initproc	__init__	X	X		X

PyTypeObject Slot [[1]]	Type	special methods/attrs	Info [[2]]			
			O	T	D	I
tp_alloc	allocfunc		X		?	?
tp_new	newfunc	__new__	X	X	?	?
tp_free	freefunc		X	X	?	?
tp_is_gc	inquiry			X		X
<tp_bases>	PyObject *	__bases__			~	
<tp_mro>	PyObject *	__mro__			~	
[tp_cache]	PyObject *					
[tp_subclasses]	void *	__subclasses__				
[tp_weaklist]	PyObject *					
(tp_del)	destructor					
[tp_version_tag]	unsigned int					
tp_finalize	destructor	__del__				X

PyTypeObject Slot [[1]]	Type	special methods/attrs	Info [[2]]			
			O	T	D	I
tp_vectorcall	vectorcallfunc					
[tp_watched]	unsigned char					

[1]

(): A slot name in parentheses indicates it is (effectively) deprecated.

<>: Names in angle brackets should be initially set to `NULL` and treated as read-only.

[]: Names in square brackets are for internal use only.

<R> (as a prefix) means the field is required (must be non-`NULL`).

[2]

Columns:

“O”: set on `PyBaseObject_Type`

“T”: set on `PyType_Type`

“D”: default (if slot is set to `NULL`)

```
X - PyType_Ready sets this value if it is NULL
~ - PyType_Ready always sets this value (it should be NULL)
? - PyType_Ready may set this value depending on other slots

Also see the inheritance column ("I").
```

“I”: inheritance

```
X - type slot is inherited via *PyType_Ready* if defined with a
*NULL* value
% - the slots of the sub-struct are inherited individually
G - inherited, but only in combination with other slots; see the
slot's description
? - it's complicated; see the slot's description
```

Note that some slots are effectively inherited through the normal attribute lookup chain.

sub-slots

Slot	Type	special methods
`am_await`	`unaryfunc`	__await__
`am_aiter`	`unaryfunc`	__aiter__
`am_anext`	`unaryfunc`	__anext__
`am_send`	`sendfunc`	
`nb_add`	`binaryfunc`	__add__ __radd__
`nb_inplace_add`	`binaryfunc`	__iadd__
`nb_subtract`	`binaryfunc`	__sub__ __rsub__
`nb_inplace_subtract`	`binaryfunc`	__isub__
`nb_multiply`	`binaryfunc`	__mul__ __rmul__

Slot	Type	special methods
`nb_inplace_multiply`	`binaryfunc`	__imul__
`nb_remainder`	`binaryfunc`	__mod__ __rmod__
`nb_inplace_remainder`	`binaryfunc`	__imod__
`nb_divmod`	`binaryfunc`	__divmod__ __rdivmod__
`nb_power`	`ternaryfunc`	__pow__ __rpow__
`nb_inplace_power`	`ternaryfunc`	__ipow__
`nb_negative`	`unaryfunc`	__neg__
`nb_positive`	`unaryfunc`	__pos__
`nb_absolute`	`unaryfunc`	__abs__
`nb_bool`	`inquiry`	__bool__
`nb_invert`	`unaryfunc`	__invert__
`nb_lshift`	`binaryfunc`	__lshift__ __rlshift__

Slot	Type	special methods
nb_inplace_lshift	binaryfunc	__ilshift__
nb_rshift	binaryfunc	__rshift__ __rrshift__
nb_inplace_rshift	binaryfunc	__irshift__
nb_and	binaryfunc	__and__ __rand__
nb_inplace_and	binaryfunc	__iand__
nb_xor	binaryfunc	__xor__ __rxor__
nb_inplace_xor	binaryfunc	__ixor__
nb_or	binaryfunc	__or__ __ror__
nb_inplace_or	binaryfunc	__ior__
nb_int	unaryfunc	__int__
nb_reserved	void *	
nb_float	unaryfunc	__float__

Slot	Type	special methods
nb_floor_divide	binaryfunc	__floordiv__
nb_inplace_floor_divide	binaryfunc	__ifloordiv__
nb_true_divide	binaryfunc	__truediv__
nb_inplace_true_divide	binaryfunc	__itruediv__
nb_index	unaryfunc	__index__
nb_matrix_multiply	binaryfunc	__matmul__ __rmatmul__
nb_inplace_matrix_multiply	binaryfunc	__imatmul__
mp_length	lenfunc	__len__
mp_subscript	binaryfunc	__getitem__
mp_ass_subscript	objobjargproc	__setitem__, __delitem__
sq_length	lenfunc	__len__
sq_concat	binaryfunc	__add__

Slot	Type	special methods
sq_repeat	ssizeargfunc	__mul__
sq_item	ssizeargfunc	__getitem__
sq_ass_item	ssizeobjargproc	__setitem__ __delitem__
sq_contains	objobjproc	__contains__
sq_inplace_concat	binaryfunc	__iadd__
sq_inplace_repeat	ssizeargfunc	__imul__
bf_getbuffer	getbufferproc()	__buffer__
bf_releasebuffer	releasebufferproc()	__release_buffer__

slot typedefs

typedef	Parameter Types	Return Type
allocfunc	PyTypeObject * Py_ssize_t	PyObject *
destructor	PyObject *	void

typedef	Parameter Types	Return Type
`freefunc`	void *	void
`traverseproc`	`PyObject` * `visitproc` void *	int
`newfunc`	`PyTypeObject` * `PyObject` * `PyObject` *	`PyObject` *
`initproc`	`PyObject` * `PyObject` * `PyObject` *	int
`reprfunc`	`PyObject` *	`PyObject` *
`getattrfunc`	`PyObject` * const char *	`PyObject` *
`setattrfunc`	`PyObject` * const char * `PyObject` *	int
`getattrofunc`	`PyObject` * `PyObject` *	`PyObject` *
`setattrofunc`	`PyObject` * `PyObject` * `PyObject` *	int

typedef	Parameter Types	Return Type
`descrgetfunc`	`PyObject` * `PyObject` * `PyObject` *	`PyObject` *
`descrsetfunc`	`PyObject` * `PyObject` * `PyObject` *	int
`hashfunc`	`PyObject` *	Py_hash_t
`richcmpfunc`	`PyObject` * `PyObject` * int	`PyObject` *
`getiterfunc`	`PyObject` *	`PyObject` *
`iternextfunc`	`PyObject` *	`PyObject` *
`lenfunc`	`PyObject` *	`Py_ssize_t`
`getbufferproc`	`PyObject` * `Py_buffer` * int	int
`releasebufferproc`	`PyObject` * `Py_buffer` *	void
`inquiry`	`PyObject` *	int

typedef	Parameter Types	Return Type
`unaryfunc`	`PyObject` *	`PyObject` *
`binaryfunc`	`PyObject` * `PyObject` *	`PyObject` *
`ternaryfunc`	`PyObject` * `PyObject` * `PyObject` *	`PyObject` *
`ssizeargfunc`	`PyObject` * `Py_ssize_t`	`PyObject` *
`ssizeobjargproc`	`PyObject` * `Py_ssize_t` `PyObject` *	int
`objobjproc`	`PyObject` * `PyObject` *	int
`objobjargproc`	`PyObject` * `PyObject` * `PyObject` *	int

See Slot Type typedefs below for more detail.

PyTypeObject Definition

The structure definition for `PyTypeObject` can be found in `Include/cpython/object.h`. For convenience of reference, this repeats the definition found there:

```
typedef struct _typeobject {
    PyObject_VAR_HEAD
    const char *tp_name; /* For printing, in format "<module>.<name>"
*/
    Py_ssize_t tp_basicsize, tp_itemsize; /* For allocation */

    /* Methods to implement standard operations */

    destructor tp_dealloc;
    Py_ssize_t tp_vectorcall_offset;
    getattrfunc tp_getattr;
    setattrfunc tp_setattr;
    PyAsyncMethods *tp_as_async; /* formerly known as tp_compare
(Python 2)
                                    or tp_reserved (Python 3) */
    reprfunc tp_repr;

    /* Method suites for standard classes */

    PyNumberMethods *tp_as_number;
    PySequenceMethods *tp_as_sequence;
    PyMappingMethods *tp_as_mapping;

    /* More standard operations (here for binary compatibility) */

    hashfunc tp_hash;
    ternaryfunc tp_call;
    reprfunc tp_str;
    getattrofunc tp_getattro;
    setattrofunc tp_setattro;

    /* Functions to access object as input/output buffer */
    PyBufferProcs *tp_as_buffer;

    /* Flags to define presence of optional/expanded features */
    unsigned long tp_flags;

    const char *tp_doc; /* Documentation string */

    /* Assigned meaning in release 2.0 */
    /* call function for all accessible objects */
    traverseproc tp_traverse;

    /* delete references to contained objects */
    inquiry tp_clear;
    /* Assigned meaning in release 2.1 */
    /* rich comparisons */
    richcmpfunc tp_richcompare;

    /* weak reference enabler */
    Py_ssize_t tp_weaklistoffset;

    /* Iterators */
    getiterfunc tp_iter;
    iternextfunc tp_iternext;

    /* Attribute descriptor and subclassing stuff */
    PyMethodDef *tp_methods;
    PyMemberDef *tp_members;
    PyGetSetDef *tp_getset;
    // Strong reference on a heap type, borrowed reference on a
static type
    PyTypeObject *tp_base;
    PyObject *tp_dict;
    descrgetfunc tp_descr_get;
    descrsetfunc tp_descr_set;
    Py_ssize_t tp_dictoffset;
    initproc tp_init;
    allocfunc tp_alloc;
    newfunc tp_new;
    freefunc tp_free; /* Low-level free-memory routine */
    inquiry tp_is_gc; /* For PyObject_IS_GC */
    PyObject *tp_bases;
    PyObject *tp_mro; /* method resolution order */
    PyObject *tp_cache; /* no longer used */
    void *tp_subclasses;  /* for static builtin types this is an
index */
    PyObject *tp_weaklist; /* not used for static builtin types */
    destructor tp_del;

    /* Type attribute cache version tag. Added in version 2.6.
     * If zero, the cache is invalid and must be initialized.
     */
    unsigned int tp_version_tag;

    destructor tp_finalize;
    vectorcallfunc tp_vectorcall;

    /* bitset of which type-watchers care about this type */
    unsigned char tp_watched;
    /* Number of tp_version_tag values used.
     * Set to _Py_ATTR_CACHE_UNUSED if the attribute cache is
     * disabled for this type (e.g. due to custom MRO entries).
     * Otherwise, limited to MAX_VERSIONS_PER_CLASS (defined
elsewhere).
     */
    uint16_t tp_versions_used;
} PyTypeObject;
```

PyObject Slots

The type object structure extends the **`PyVarObject`** structure. The **`ob_size`** field is used for dynamic types (created by **`type_new()`**, usually called from a class statement). Note that **`PyType_Type`** (the metatype) initializes **`tp_itemsize`**, which means that its instances (i.e. type objects) *must* have the **`ob_size`** field.

`PyObject.ob_refcnt`

The type object's reference count is initialized to `1` by the `PyObject_HEAD_INIT` macro. Note that for statically allocated type objects, the type's instances (objects whose **`ob_type`** points back to the type) do *not* count as references. But for dynamically allocated type objects, the instances *do* count as references.

Inheritance:

This field is not inherited by subtypes.

`PyObject.ob_type`

This is the type's type, in other words its metatype. It is initialized by the argument to the `PyObject_HEAD_INIT` macro, and its value should normally be `&PyType_Type`. However, for dynamically loadable extension modules that must be usable on Windows (at least), the compiler complains that this is not a valid initializer. Therefore, the convention is to pass `NULL` to the `PyObject_HEAD_INIT` macro and to initialize this field explicitly at the start of the module's initialization function, before doing anything else. This is typically done like this:

```
Foo_Type.ob_type = &PyType_Type;
```

This should be done before any instances of the type are created. **`PyType_Ready()`** checks if **`ob_type`** is `NULL`, and if so, initializes it to the **`ob_type`** field of the base class. **`PyType_Ready()`** will not change this field if it is non-zero.

Inheritance:

This field is inherited by subtypes.

PyVarObject Slots

`PyVarObject.ob_size`

For statically allocated type objects, this should be initialized to zero. For dynamically allocated type objects, this field has a special internal meaning.

This field should be accessed using the **`Py_SIZE()`** macro.

Inheritance:

This field is not inherited by subtypes.

PyTypeObject Slots

Each slot has a section describing inheritance. If **`PyType_Ready()`** may set a value when the field is set to `NULL` then there will also be a “Default” section. (Note that many fields set on **`PyBaseObject_Type`** and **`PyType_Type`** effectively act as defaults.)

const char *PyTypeObject.tp_name

Pointer to a NUL-terminated string containing the name of the type. For types that are accessible as module globals, the string should be the full module name, followed by a dot, followed by the type name; for built-in types, it should be just the type name. If the module is a submodule of a package, the full package name is part of the full module name. For example, a type named `T` defined in module `M` in subpackage `Q` in package `P` should have the `tp_name` initializer `"P.Q.M.T"`.

For dynamically allocated type objects, this should just be the type name, and the module name explicitly stored in the type dict as the value for key `'__module__'`.

For statically allocated type objects, the *tp_name* field should contain a dot. Everything before the last dot is made accessible as the `__module__` attribute, and everything after the last dot is made accessible as the `__name__` attribute.

If no dot is present, the entire `tp_name` field is made accessible as the `__name__` attribute, and the `__module__` attribute is undefined (unless explicitly set in the dictionary, as explained above). This means your type will be impossible to pickle. Additionally, it will not be listed in module documentations created with pydoc.

This field must not be `NULL`. It is the only required field in `PyTypeObject()` (other than potentially `tp_itemsize`).

Inheritance:

This field is not inherited by subtypes.

Py_ssize_t PyTypeObject.tp_basicsize
Py_ssize_t PyTypeObject.tp_itemsize

These fields allow calculating the size in bytes of instances of the type.

There are two kinds of types: types with fixed-length instances have a zero `tp_itemsize` field, types with variable-length instances have a non-zero `tp_itemsize` field. For a type with fixed-length instances, all instances have the same size, given in `tp_basicsize`. (Exceptions to this rule can be made using `PyUnstable_Object_GC_NewWithExtraData()`.)

For a type with variable-length instances, the instances must have an `ob_size` field, and the instance size is `tp_basicsize` plus N times `tp_itemsize`, where N is the “length” of the object.

Functions like `PyObject_NewVar()` will take the value of N as an argument, and store in the instance's `ob_size` field. Note that the `ob_size` field may later be used for other purposes. For example, `int` instances use the bits of `ob_size` in an implementation-defined way; the underlying storage and its size should be accessed using `PyLong_Export()`.

Note

The `ob_size` field should be accessed using the `Py_SIZE()` and `Py_SET_SIZE()` macros.

Also, the presence of an `ob_size` field in the instance layout doesn't mean that the instance structure is variable-length. For example, the `list` type has fixed-length instances, yet those instances have a `ob_size` field. (As with `int`, avoid reading lists' **`ob_size`** directly. Call `PyList_Size()` instead.)

The **`tp_basicsize`** includes size needed for data of the type's `tp_base`, plus any extra data needed by each instance.

The correct way to set **`tp_basicsize`** is to use the `sizeof` operator on the struct used to declare the instance layout. This struct must include the struct used to declare the base type. In other words, **`tp_basicsize`** must be greater than or equal to the base's **`tp_basicsize`**.

Since every type is a subtype of `object`, this struct must include `PyObject` or `PyVarObject` (depending on whether `ob_size` should be included). These are usually defined by the macro `PyObject_HEAD` or `PyObject_VAR_HEAD`, respectively.

The basic size does not include the GC header size, as that header is not part of `PyObject_HEAD`.

For cases where struct used to declare the base type is unknown, see `PyType_Spec.basicsize` and `PyType_FromMetaclass()`.

Notes about alignment:

- **`tp_basicsize`** must be a multiple of `_Alignof(PyObject)`. When using `sizeof` on a `struct` that includes **`PyObject_HEAD`**, as recommended, the compiler ensures this. When not using a C `struct`, or when using compiler extensions like `__attribute__((packed))`, it is up to you.
- If the variable items require a particular alignment, **`tp_basicsize`** and **`tp_itemsize`** must each be a multiple of that alignment. For example, if a type's variable part stores a `double`, it is your responsibility that both fields are a multiple of `_Alignof(double)`.

Inheritance:

These fields are inherited separately by subtypes. (That is, if the field is set to zero, **`PyType_Ready()`** will copy the value from the base type, indicating that the instances do not need additional storage.)

If the base type has a non-zero **`tp_itemsize`**, it is generally not safe to set **`tp_itemsize`** to a different non-zero value in a subtype (though this depends on the implementation of the base type).

destructor PyTypeObject.tp_dealloc

The corresponding slot ID `Py_tp_dealloc` is part of the Stable ABI.

A pointer to the instance destructor function. The function signature is:

```
void tp_dealloc(PyObject *self);
```

The destructor function should remove all references which the instance owns (e.g., call **`Py_CLEAR()`**), free all memory buffers owned by the instance, and call the type's **`tp_free`** function to free the object itself.

If you may call functions that may set the error indicator, you must use **`PyErr_GetRaisedException()`** and **`PyErr_SetRaisedException()`** to ensure you don't clobber a preexisting error indicator (the deallocation could have occurred while processing a different error):

```
static void
foo_dealloc(foo_object *self)
{
    PyObject *et, *ev, *etb;
    PyObject *exc = PyErr_GetRaisedException();
    ...
    PyErr_SetRaisedException(exc);
}
```

The dealloc handler itself must not raise an exception; if it hits an error case it should call `PyErr_FormatUnraisable()` to log (and clear) an unraisable exception.

No guarantees are made about when an object is destroyed, except:

- Python will destroy an object immediately or some time after the final reference to the object is deleted, unless its finalizer (`tp_finalize`) subsequently resurrects the object.
- An object will not be destroyed while it is being automatically finalized (`tp_finalize`) or automatically cleared (`tp_clear`).

CPython currently destroys an object immediately from `Py_DECREF()` when the new reference count is zero, but this may change in a future version.

It is recommended to call `PyObject_CallFinalizerFromDealloc()` at the beginning of **`tp_dealloc`** to guarantee that the object is always finalized before destruction.

If the type supports garbage collection (the `Py_TPFLAGS_HAVE_GC` flag is set), the destructor should call `PyObject_GC_UnTrack()` before clearing any member fields.

It is permissible to call `tp_clear` from **`tp_dealloc`** to reduce code duplication and to guarantee that the object is always cleared before destruction. Beware that **`tp_clear`** might have already been called.

If the type is heap allocated (`Py_TPFLAGS_HEAPTYPE`), the deallocator should release the owned reference to its type object (via `Py_DECREF()`) after calling the type deallocator. See the example code below.:

```
static void
foo_dealloc(PyObject *op)
{
   foo_object *self = (foo_object *) op;
   PyObject_GC_UnTrack(self);
   Py_CLEAR(self->ref);
   Py_TYPE(self)->tp_free(self);
}
```

`tp_dealloc` must leave the exception status unchanged. If it needs to call something that might raise an exception, the exception state must be backed up first and restored later (after logging any exceptions with `PyErr_WriteUnraisable()`).

Example:

```
static void
foo_dealloc(PyObject *self)
{
    PyObject *exc = PyErr_GetRaisedException();

    if (PyObject_CallFinalizerFromDealloc(self) < 0) {
        // self was resurrected.
        goto done;
    }

    PyTypeObject *tp = Py_TYPE(self);

    if (tp->tp_flags & Py_TPFLAGS_HAVE_GC) {
        PyObject_GC_UnTrack(self);
    }

    // Optional, but convenient to avoid code duplication.
    if (tp->tp_clear && tp->tp_clear(self) < 0) {
        PyErr_WriteUnraisable(self);
    }

    // Any additional destruction goes here.

    tp->tp_free(self);
    self = NULL;  // In case PyErr_WriteUnraisable() is called
below.
```

```
    if (tp->tp_flags & Py_TPFLAGS_HEAPTYPE) {
        Py_CLEAR(tp);
    }

done:
    // Optional, if something was called that might have raised an
    // exception.
    if (PyErr_Occurred()) {
        PyErr_WriteUnraisable(self);
    }
    PyErr_SetRaisedException(exc);
}
```

`tp_dealloc` may be called from any Python thread, not just the thread which created the object (if the object becomes part of a refcount cycle, that cycle might be collected by a garbage collection on any thread). This is not a problem for Python API calls, since the thread on which **`tp_dealloc`** is called with an attached thread state. However, if the object being destroyed in turn destroys objects from some other C library, care should be taken to ensure that destroying those objects on the thread which called **`tp_dealloc`** will not violate any assumptions of the library.

Inheritance:

This field is inherited by subtypes.

See also

Object Life Cycle for details about how this slot relates to other slots.

Py_ssize_t PyTypeObject.tp_vectorcall_offset

An optional offset to a per-instance function that implements calling the object using the vectorcall protocol, a more efficient alternative of the simpler **`tp_call`**.

This field is only used if the flag **`Py_TPFLAGS_HAVE_VECTORCALL`** is set. If so, this must be a positive integer containing the offset in the instance of a **`vectorcallfunc`** pointer.

The *vectorcallfunc* pointer may be `NULL`, in which case the instance behaves as if **`Py_TPFLAGS_HAVE_VECTORCALL`** was not set: calling the instance falls back to **`tp_call`**.

Any class that sets `Py_TPFLAGS_HAVE_VECTORCALL` must also set **`tp_call`** and make sure its behaviour is consistent with the *vectorcallfunc* function. This can be done by setting *tp_call* to **`PyVectorcall_Call()`**.

Changed in version 3.8: Before version 3.8, this slot was named `tp_print`. In Python 2.x, it was used for printing to a file. In Python 3.0 to 3.7, it was unused.

Changed in version 3.12: Before version 3.12, it was not recommended for mutable heap types to implement the vectorcall protocol. When a user sets **`__call__`** in Python code, only *tp_call* is updated, likely making it inconsistent with the vectorcall function. Since 3.12, setting `__call__` will disable vectorcall optimization by clearing the **`Py_TPFLAGS_HAVE_VECTORCALL`** flag.

Inheritance:

This field is always inherited. However, the **`Py_TPFLAGS_HAVE_VECTORCALL`** flag is not always inherited. If it's not set, then the subclass won't use vectorcall, except when **`PyVectorcall_Call()`** is explicitly called.

getattrfunc PyTypeObject.tp_getattr

The corresponding slot ID `Py_tp_getattr` is part of the Stable ABI.

An optional pointer to the get-attribute-string function.

This field is deprecated. When it is defined, it should point to a function that acts the same as the **`tp_getattro`** function, but taking a C string instead of a Python string object to give the attribute name.

Inheritance:

Group: **`tp_getattr`**, **`tp_getattro`**

This field is inherited by subtypes together with **`tp_getattro`**: a subtype inherits both **`tp_getattr`** and **`tp_getattro`** from its base type when the subtype's **`tp_getattr`** and **`tp_getattro`** are both `NULL`.

setattrfunc PyTypeObject.tp_setattr

The corresponding slot ID `Py_tp_setattr` is part of the Stable ABI.

An optional pointer to the function for setting and deleting attributes.

This field is deprecated. When it is defined, it should point to a function that acts the same as the **`tp_setattro`** function, but taking a C string instead of a Python string object to give the attribute name.

Inheritance:

Group: **`tp_setattr`**, **`tp_setattro`**

This field is inherited by subtypes together with **`tp_setattro`**: a subtype inherits both **`tp_setattr`** and **`tp_setattro`** from its base type when the subtype's **`tp_setattr`** and **`tp_setattro`** are both `NULL`.

PyAsyncMethods *PyTypeObject.tp_as_async

Pointer to an additional structure that contains fields relevant only to objects which implement awaitable and asynchronous iterator protocols at the C-level. See Async Object Structures for details.

Added in version 3.5: Formerly known as `tp_compare` and `tp_reserved`.

Inheritance:

The **`tp_as_async`** field is not inherited, but the contained fields are inherited individually.

reprfunc PyTypeObject.tp_repr

The corresponding slot ID `Py_tp_repr` is part of the Stable ABI.

An optional pointer to a function that implements the built-in function `repr()`.

The signature is the same as for `PyObject_Repr()`:

```
PyObject *tp_repr(PyObject *self);
```

The function must return a string or a Unicode object. Ideally, this function should return a string that, when passed to `eval()`, given a suitable environment, returns an object with the same value. If this is not feasible, it should return a string starting with `'<'` and ending with `'>'` from which both the type and the value of the object can be deduced.

Inheritance:

This field is inherited by subtypes.

Default:

When this field is not set, a string of the form `<%s object at %p>` is returned, where `%s` is replaced by the type name, and `%p` by the object’s memory address.

PyNumberMethods *PyTypeObject.tp_as_number

Pointer to an additional structure that contains fields relevant only to objects which implement the number protocol. These fields are documented in Number Object Structures.

Inheritance:

The `tp_as_number` field is not inherited, but the contained fields are inherited individually.

PySequenceMethods *PyTypeObject.tp_as_sequence

Pointer to an additional structure that contains fields relevant only to objects which implement the sequence protocol. These fields are documented in Sequence Object Structures.

Inheritance:

The `tp_as_sequence` field is not inherited, but the contained fields are inherited individually.

PyMappingMethods *PyTypeObject.tp_as_mapping

Pointer to an additional structure that contains fields relevant only to objects which implement the mapping protocol. These fields are documented in Mapping Object Structures.

Inheritance:

The `tp_as_mapping` field is not inherited, but the contained fields are inherited individually.

hashfunc PyTypeObject.tp_hash

The corresponding slot ID `Py_tp_hash` *is part of the Stable ABI.*

An optional pointer to a function that implements the built-in function `hash()`.

The signature is the same as for `PyObject_Hash()`:

```
Py_hash_t tp_hash(PyObject *);
```

The value `-1` should not be returned as a normal return value; when an error occurs during the computation of the hash value, the function should set an exception and return `-1`.

When this field is not set (*and* `tp_richcompare` is not set), an attempt to take the hash of the object raises `TypeError`. This is the same as setting it to `PyObject_HashNotImplemented()`.

This field can be set explicitly to `PyObject_HashNotImplemented()` to block inheritance of the hash method from a parent type. This is interpreted as the equivalent of `__hash__ = None` at the Python level, causing `isinstance(o, collections.Hashable)` to correctly return `False`. Note that the converse is also true - setting `__hash__ = None` on a class at the Python level will result in the `tp_hash` slot being set to `PyObject_HashNotImplemented()`.

Inheritance:

Group: `tp_hash`, `tp_richcompare`

This field is inherited by subtypes together with `tp_richcompare`: a subtype inherits both of `tp_richcompare` and `tp_hash`, when the subtype's `tp_richcompare` and `tp_hash` are both `NULL`.

Default:

`PyBaseObject_Type` uses `PyObject_GenericHash()`.

ternaryfunc PyTypeObject.tp_call

The corresponding slot ID `Py_tp_call` is part of the Stable ABI.

An optional pointer to a function that implements calling the object. This should be `NULL` if the object is not callable. The signature is the same as for `PyObject_Call()`:

```
PyObject *tp_call(PyObject *self, PyObject *args, PyObject
*kwargs);
```

Inheritance:

This field is inherited by subtypes.

reprfunc PyTypeObject.tp_str

The corresponding slot ID `Py_tp_str` is part of the Stable ABI.

An optional pointer to a function that implements the built-in operation `str()`. (Note that `str` is a type now, and `str()` calls the constructor for that type. This constructor calls `PyObject_Str()` to do the actual work, and `PyObject_Str()` will call this handler.)

The signature is the same as for `PyObject_Str()`:

```
PyObject *tp_str(PyObject *self);
```

The function must return a string or a Unicode object. It should be a “friendly” string representation of the object, as this is the representation that will be used, among other things, by the `print()` function.

Inheritance:

This field is inherited by subtypes.

Default:

When this field is not set, **`PyObject_Repr()`** is called to return a string representation.

getattrofunc PyTypeObject.tp_getattro

The corresponding slot ID `Py_tp_getattro` is part of the Stable ABI.

An optional pointer to the get-attribute function.

The signature is the same as for **`PyObject_GetAttr()`**:

```
PyObject *tp_getattro(PyObject *self, PyObject *attr);
```

It is usually convenient to set this field to **`PyObject_GenericGetAttr()`**, which implements the normal way of looking for object attributes.

Inheritance:

Group: **`tp_getattr`**, **`tp_getattro`**

This field is inherited by subtypes together with **`tp_getattr`**: a subtype inherits both **`tp_getattr`** and **`tp_getattro`** from its base type when the subtype's **`tp_getattr`** and **`tp_getattro`** are both `NULL`.

Default:

`PyBaseObject_Type` uses **`PyObject_GenericGetAttr()`**.

setattrofunc PyTypeObject.tp_setattro

The corresponding slot ID `Py_tp_setattro` is part of the Stable ABI.

An optional pointer to the function for setting and deleting attributes.

The signature is the same as for **`PyObject_SetAttr()`**:

```
int tp_setattro(PyObject *self, PyObject *attr, PyObject *value);
```

In addition, setting *value* to `NULL` to delete an attribute must be supported. It is usually convenient to set this field to `PyObject_GenericSetAttr()`, which implements the normal way of setting object attributes.

Inheritance:

Group: `tp_setattr`, `tp_setattro`

This field is inherited by subtypes together with `tp_setattr`: a subtype inherits both `tp_setattr` and `tp_setattro` from its base type when the subtype's `tp_setattr` and `tp_setattro` are both `NULL`.

Default:

`PyBaseObject_Type` uses `PyObject_GenericSetAttr()`.

PyBufferProcs *PyTypeObject.tp_as_buffer

Pointer to an additional structure that contains fields relevant only to objects which implement the buffer interface. These fields are documented in Buffer Object Structures.

Inheritance:

The `tp_as_buffer` field is not inherited, but the contained fields are inherited individually.

unsigned long PyTypeObject.tp_flags

This field is a bit mask of various flags. Some flags indicate variant semantics for certain situations; others are used to indicate that certain fields in the type object (or in the extension structures referenced via `tp_as_number`, `tp_as_sequence`, `tp_as_mapping`, and `tp_as_buffer`) that were historically not always present are valid; if such a flag bit is clear, the type fields it guards must not be accessed and must be considered to have a zero or `NULL` value instead.

Inheritance:

Inheritance of this field is complicated. Most flag bits are inherited individually, i.e. if the base type has a flag bit set, the subtype inherits this flag bit. The flag bits that pertain to extension structures are strictly inherited if the extension structure is inherited, i.e. the base type's value of the flag bit is copied into the subtype together with a pointer to the extension structure. The `Py_TPFLAGS_HAVE_GC` flag bit is inherited together with the `tp_traverse` and `tp_clear` fields, i.e. if the `Py_TPFLAGS_HAVE_GC` flag bit is clear in the subtype and the `tp_traverse` and `tp_clear` fields in the subtype exist and have `NULL` values.

Default:

`PyBaseObject_Type` uses `Py_TPFLAGS_DEFAULT | Py_TPFLAGS_BASETYPE`.

Bit Masks:

The following bit masks are currently defined; these can be ORed together using the `|` operator to form the value of the `tp_flags` field. The macro `PyType_HasFeature()` takes a type and a flags value, *tp* and *f*, and checks whether `tp->tp_flags & f` is non-zero.

Py_TPFLAGS_HEAPTYPE

This bit is set when the type object itself is allocated on the heap, for example, types created dynamically using `PyType_FromSpec()`. In this case, the `ob_type` field of its instances is considered a reference to the type, and the type object is INCREF'ed when a new instance is created, and DECREF'ed when an instance is destroyed (this does not apply to instances of subtypes; only the type referenced by the instance's ob_type gets INCREF'ed or DECREF'ed). Heap types should also support garbage collection as they can form a reference cycle with their own module object.

Inheritance:

???

Py_TPFLAGS_BASETYPE

Part of the Stable ABI.

This bit is set when the type can be used as the base type of another type. If this bit is clear, the type cannot be subtyped (similar to a “final” class in Java).

Inheritance:

???

Py_TPFLAGS_READY

This bit is set when the type object has been fully initialized by `PyType_Ready()`.

Inheritance:

???

Py_TPFLAGS_READYING

This bit is set while `PyType_Ready()` is in the process of initializing the type object.

Inheritance:

???

Py_TPFLAGS_HAVE_GC

Part of the Stable ABI.

This bit is set when the object supports garbage collection. If this bit is set, memory for new instances (see `tp_alloc`) must be allocated using `PyObject_GC_New` or `PyType_GenericAlloc()` and deallocated (see `tp_free`) using `PyObject_GC_Del()`. More information in section Supporting Cyclic Garbage Collection.

Inheritance:

Group: `Py_TPFLAGS_HAVE_GC`, `tp_traverse`, `tp_clear`

The `Py_TPFLAGS_HAVE_GC` flag bit is inherited together with the `tp_traverse` and `tp_clear` fields, i.e. if the `Py_TPFLAGS_HAVE_GC` flag bit is clear in the subtype and the **`tp_traverse`** and **`tp_clear`** fields in the subtype exist and have `NULL` values.

Py_TPFLAGS_DEFAULT

Part of the Stable ABI.

This is a bitmask of all the bits that pertain to the existence of certain fields in the type object and its extension structures. Currently, it includes the following bits: **`Py_TPFLAGS_HAVE_STACKLESS_EXTENSION`**.

Inheritance:

???

Py_TPFLAGS_METHOD_DESCRIPTOR

Part of the Stable ABI since version 3.8.

This bit indicates that objects behave like unbound methods.

If this flag is set for `type(meth)`, then:

- `meth.__get__(obj, cls)(*args, **kwds)` (with `obj` not None) must be equivalent to `meth(obj, *args, **kwds)`.
- `meth.__get__(None, cls)(*args, **kwds)` must be equivalent to `meth(*args, **kwds)`.

This flag enables an optimization for typical method calls like `obj.meth()`: it avoids creating a temporary “bound method” object for `obj.meth`.

Added in version 3.8.

Inheritance:

This flag is never inherited by types without the **`Py_TPFLAGS_IMMUTABLETYPE`** flag set. For extension types, it is inherited whenever **`tp_descr_get`** is inherited.

Py_TPFLAGS_MANAGED_DICT

This bit indicates that instances of the class have a **`__dict__`** attribute, and that the space for the dictionary is managed by the VM.

If this flag is set, `Py_TPFLAGS_HAVE_GC` should also be set.

The type traverse function must call `PyObject_VisitManagedDict()` and its clear function must call `PyObject_ClearManagedDict()`.

Added in version 3.12.

Inheritance:

This flag is inherited unless the `tp_dictoffset` field is set in a superclass.

Py_TPFLAGS_MANAGED_WEAKREF

This bit indicates that instances of the class should be weakly referenceable.

Added in version 3.12.

Inheritance:

This flag is inherited unless the `tp_weaklistoffset` field is set in a superclass.

Py_TPFLAGS_ITEMS_AT_END

Part of the Stable ABI since version 3.12.
Only usable with variable-size types, i.e. ones with non-zero `tp_itemsize`.

Indicates that the variable-sized portion of an instance of this type is at the end of the instance’s memory area, at an offset of `Py_TYPE(obj)->tp_basicsize` (which may be different in each subclass).

When setting this flag, be sure that all superclasses either use this memory layout, or are not variable-sized. Python does not check this.

Added in version 3.12.

Inheritance:

This flag is inherited.

Py_TPFLAGS_LONG_SUBCLASS

Py_TPFLAGS_LIST_SUBCLASS

Py_TPFLAGS_TUPLE_SUBCLASS

Py_TPFLAGS_BYTES_SUBCLASS

Py_TPFLAGS_UNICODE_SUBCLASS

Py_TPFLAGS_DICT_SUBCLASS

Py_TPFLAGS_BASE_EXC_SUBCLASS

Py_TPFLAGS_TYPE_SUBCLASS

Functions such as `PyLong_Check()` will call `PyType_FastSubclass()` with one of these flags to quickly determine if a type is a subclass of a built-in type; such specific checks are faster than a generic check, like `PyObject_IsInstance()`. Custom types that inherit from built-ins should have their `tp_flags` set appropriately, or the code that interacts with such types will behave differently depending on what kind of check is used.

Py_TPFLAGS_HAVE_FINALIZE

This bit is set when the `tp_finalize` slot is present in the type structure.

Added in version 3.4.

Deprecated since version 3.8: This flag isn't necessary anymore, as the interpreter assumes the `tp_finalize` slot is always present in the type structure.

Py_TPFLAGS_HAVE_VECTORCALL

Part of the Stable ABI since version 3.12.

This bit is set when the class implements the vectorcall protocol. See `tp_vectorcall_offset` for details.

Inheritance:

This bit is inherited if `tp_call` is also inherited.

Added in version 3.8: as `_Py_TPFLAGS_HAVE_VECTORCALL`

Changed in version 3.9.

Renamed to the current name, without the leading underscore. The old provisional name is soft deprecated.

Changed in version 3.12: This flag is now removed from a class when the class's `__call__()` method is reassigned.

This flag can now be inherited by mutable classes.

Py_TPFLAGS_IMMUTABLETYPE

This bit is set for type objects that are immutable: type attributes cannot be set nor deleted.

`PyType_Ready()` automatically applies this flag to static types.

Inheritance:

This flag is not inherited.

Added in version 3.10.

Py_TPFLAGS_DISALLOW_INSTANTIATION

Disallow creating instances of the type: set `tp_new` to NULL and don't create the `__new__` key in the type dictionary.

The flag must be set before creating the type, not after. For example, it must be set before `PyType_Ready()` is called on the type.

The flag is set automatically on static types if `tp_base` is NULL or `&PyBaseObject_Type` and `tp_new` is NULL.

Inheritance:

This flag is not inherited. However, subclasses will not be instantiable unless they provide a non-NULL `tp_new` (which is only possible via the C API).

Note

To disallow instantiating a class directly but allow instantiating its subclasses (e.g. for an abstract base class), do not use this flag. Instead, make `tp_new` only succeed for subclasses.

Added in version 3.10.

Py_TPFLAGS_MAPPING

This bit indicates that instances of the class may match mapping patterns when used as the subject of a `match` block. It is automatically set when registering or subclassing `collections.abc.Mapping`, and unset when registering `collections.abc.Sequence`.

Note

`Py_TPFLAGS_MAPPING` and `Py_TPFLAGS_SEQUENCE` are mutually exclusive; it is an error to enable both flags simultaneously.

Inheritance:

This flag is inherited by types that do not already set `Py_TPFLAGS_SEQUENCE`.

See also

PEP 634 [https://peps.python.org/pep-0634/] – Structural Pattern Matching: Specification

Added in version 3.10.

Py_TPFLAGS_SEQUENCE

This bit indicates that instances of the class may match sequence patterns when used as the subject of a `match` block. It is automatically set when registering or subclassing `collections.abc.Sequence`, and unset when registering `collections.abc.Mapping`.

Note

`Py_TPFLAGS_MAPPING` and `Py_TPFLAGS_SEQUENCE` are mutually exclusive; it is an error to enable both flags simultaneously.

Inheritance:

This flag is inherited by types that do not already set `Py_TPFLAGS_MAPPING`.

See also

PEP 634 [https://peps.python.org/pep-0634/] – Structural Pattern Matching: Specification

Added in version 3.10.

Py_TPFLAGS_VALID_VERSION_TAG

Internal. Do not set or unset this flag. To indicate that a class has changed call `PyType_Modified()`

Warning

This flag is present in header files, but is not be used. It will be removed in a future version of CPython

const char *PyTypeObject.tp_doc

The corresponding slot ID `Py_tp_doc` is part of the Stable ABI.

An optional pointer to a NUL-terminated C string giving the docstring for this type object. This is exposed as the `__doc__` attribute on the type and instances of the type.

Inheritance:

This field is *not* inherited by subtypes.

traverseproc PyTypeObject.tp_traverse

The corresponding slot ID `Py_tp_traverse` is part of the Stable ABI.

An optional pointer to a traversal function for the garbage collector. This is only used if the **`Py_TPFLAGS_HAVE_GC`** flag bit is set. The signature is:

```
int tp_traverse(PyObject *self, visitproc visit, void *arg);
```

More information about Python's garbage collection scheme can be found in section Supporting Cyclic Garbage Collection.

The **`tp_traverse`** pointer is used by the garbage collector to detect reference cycles. A typical implementation of a **`tp_traverse`** function simply calls **`Py_VISIT()`** on each of the instance's members that are Python objects that the instance owns. For example, this is function **`local_traverse()`** from the **`_thread`** extension module:

```
static int
local_traverse(PyObject *op, visitproc visit, void *arg)
{
    localobject *self = (localobject *) op;
    Py_VISIT(self->args);
    Py_VISIT(self->kw);
    Py_VISIT(self->dict);
    return 0;
}
```

Note that **`Py_VISIT()`** is called only on those members that can participate in reference cycles. Although there is also a `self->key` member, it can only be `NULL` or a Python string and therefore cannot be part of a reference cycle.

On the other hand, even if you know a member can never be part of a cycle, as a debugging aid you may want to visit it anyway just so the **`gc`** module's **`get_referents()`** function will include it.

Heap types (**`Py_TPFLAGS_HEAPTYPE`**) must visit their type with:

```
Py_VISIT(Py_TYPE(self));
```

It is only needed since Python 3.9. To support Python 3.8 and older, this line must be conditional:

```
#if PY_VERSION_HEX >= 0x03090000
    Py_VISIT(Py_TYPE(self));
#endif
```

If the **`Py_TPFLAGS_MANAGED_DICT`** bit is set in the **`tp_flags`** field, the traverse function must call **`PyObject_VisitManagedDict()`** like this:

```
PyObject_VisitManagedDict((PyObject*)self, visit, arg);
```

Warning

When implementing **`tp_traverse`**, only the members that the instance *owns* (by having strong references to them) must be visited. For instance, if an object supports weak references via the **`tp_weaklist`** slot, the pointer supporting the linked list (what *tp_weaklist* points to) must **not** be visited as the instance does not directly own the weak references to itself (the weakreference list is there to support the weak reference machinery, but the instance has no strong reference to the elements inside it, as they are allowed to be removed even if the instance is still alive).

Warning

The traversal function must not have any side effects. It must not modify the reference counts of any Python objects nor create or destroy any Python objects.

Note that **`Py_VISIT()`** requires the *visit* and *arg* parameters to **`local_traverse()`** to have these specific names; don't name them just anything.

Instances of heap-allocated types hold a reference to their type. Their traversal function must therefore either visit **`Py_TYPE(self)`**, or delegate this responsibility by calling `tp_traverse` of another heap-allocated type (such as a heap-allocated superclass). If they do not, the type object may not be garbage-collected.

Note

The **`tp_traverse`** function can be called from any thread.

Changed in version 3.9: Heap-allocated types are expected to visit `Py_TYPE(self)` in `tp_traverse`. In earlier versions of Python, due to bug 40217 [https://bugs.python.org/issue40217], doing this may lead to crashes in subclasses.

Inheritance:

Group: **`Py_TPFLAGS_HAVE_GC`**, **`tp_traverse`**, **`tp_clear`**

This field is inherited by subtypes together with **`tp_clear`** and the **`Py_TPFLAGS_HAVE_GC`** flag bit: the flag bit, **`tp_traverse`**, and **`tp_clear`** are all inherited from the base type if they are all zero in the subtype.

inquiry PyTypeObject.tp_clear

The corresponding slot ID `Py_tp_clear` is part of the Stable ABI.

An optional pointer to a clear function. The signature is:

```
int tp_clear(PyObject *);
```

The purpose of this function is to break reference cycles that are causing a cyclic isolate so that the objects can be safely destroyed. A cleared object is a partially destroyed object; the object is not obligated to satisfy design invariants held during normal use.

`tp_clear` does not need to delete references to objects that can’t participate in reference cycles, such as Python strings or Python integers. However, it may be convenient to clear all references, and write the type’s **`tp_dealloc`** function to invoke **`tp_clear`** to avoid code duplication. (Beware that **`tp_clear`** might have already been called. Prefer calling idempotent functions like **`Py_CLEAR()`**.)

Any non-trivial cleanup should be performed in **`tp_finalize`** instead of **`tp_clear`**.

Note

If `tp_clear` fails to break a reference cycle then the objects in the cyclic isolate may remain indefinitely uncollectable (“leak”). See `gc.garbage`.

Note

Referents (direct and indirect) might have already been cleared; they are not guaranteed to be in a consistent state.

Note

The `tp_clear` function can be called from any thread.

Note

An object is not guaranteed to be automatically cleared before its destructor (`tp_dealloc`) is called.

This function differs from the destructor (`tp_dealloc`) in the following ways:

- The purpose of clearing an object is to remove references to other objects that might participate in a reference cycle. The purpose of the destructor, on the other hand, is a superset: it must release *all* resources it owns, including references to objects that cannot participate in a reference cycle (e.g., integers) as well as the object’s own memory (by calling `tp_free`).
- When `tp_clear` is called, other objects might still hold references to the object being cleared. Because of this, `tp_clear` must not deallocate the object’s own memory (`tp_free`). The destructor, on the other hand, is only called when no (strong) references exist, and as such, must safely destroy the object itself by deallocating it.
- `tp_clear` might never be automatically called. An object’s destructor, on the other hand, will be automatically called some time after the object becomes unreachable (i.e.,

either there are no references to the object or the object is a member of a cyclic isolate).

No guarantees are made about when, if, or how often Python automatically clears an object, except:

- Python will not automatically clear an object if it is reachable, i.e., there is a reference to it and it is not a member of a cyclic isolate.
- Python will not automatically clear an object if it has not been automatically finalized (see **`tp_finalize`**). (If the finalizer resurrected the object, the object may or may not be automatically finalized again before it is cleared.)
- If an object is a member of a cyclic isolate, Python will not automatically clear it if any member of the cyclic isolate has not yet been automatically finalized (**`tp_finalize`**).
- Python will not destroy an object until after any automatic calls to its **`tp_clear`** function have returned. This ensures that the act of breaking a reference cycle does not invalidate the `self` pointer while **`tp_clear`** is still executing.
- Python will not automatically call **`tp_clear`** multiple times concurrently.

CPython currently only automatically clears objects as needed to break reference cycles in a cyclic isolate, but future versions might clear objects regularly before their destruction.

Taken together, all **`tp_clear`** functions in the system must combine to break all reference cycles. This is subtle, and if in any doubt supply a **`tp_clear`** function. For example, the tuple type does not implement a **`tp_clear`** function, because it's possible to prove that no reference cycle can be composed entirely of tuples. Therefore the **`tp_clear`** functions of other types are responsible for breaking any cycle containing a tuple. This isn't immediately obvious, and there's rarely a good reason to avoid implementing **`tp_clear`**.

Implementations of **`tp_clear`** should drop the instance's references to those of its members that may be Python objects, and set its pointers to those members to `NULL`, as in the following example:

```
static int
local_clear(PyObject *op)
{
    localobject *self = (localobject *) op;
    Py_CLEAR(self->key);
    Py_CLEAR(self->args);
    Py_CLEAR(self->kw);
    Py_CLEAR(self->dict);
    return 0;
}
```

The `Py_CLEAR()` macro should be used, because clearing references is delicate: the reference to the contained object must not be released (via `Py_DECREF()`) until after the pointer to the contained object is set to `NULL`. This is because releasing the reference may cause the contained object to become trash, triggering a chain of reclamation activity that may include invoking arbitrary Python code (due to finalizers, or weakref callbacks, associated with the contained object). If it's possible for such code to reference *self* again, it's important that the pointer to the contained object be `NULL` at that time, so that *self* knows the contained object can no longer be used. The `Py_CLEAR()` macro performs the operations in a safe order.

If the `Py_TPFLAGS_MANAGED_DICT` bit is set in the `tp_flags` field, the clear function must call `PyObject_ClearManagedDict()` like this:

```
PyObject_ClearManagedDict((PyObject*)self);
```

More information about Python's garbage collection scheme can be found in section Supporting Cyclic Garbage Collection.

Inheritance:

Group: `Py_TPFLAGS_HAVE_GC`, `tp_traverse`, `tp_clear`

This field is inherited by subtypes together with `tp_traverse` and the `Py_TPFLAGS_HAVE_GC` flag bit: the flag bit, `tp_traverse`, and `tp_clear` are all inherited from the base type if they are all zero in the subtype.

See also

Object Life Cycle for details about how this slot relates to other slots.

richcmpfunc PyTypeObject.tp_richcompare

The corresponding slot ID `Py_tp_richcompare` *is part of the Stable ABI.*

An optional pointer to the rich comparison function, whose signature is:

```
PyObject *tp_richcompare(PyObject *self, PyObject *other, int op);
```

The first parameter is guaranteed to be an instance of the type that is defined by **`PyTypeObject`**.

The function should return the result of the comparison (usually `Py_True` or `Py_False`). If the comparison is undefined, it must return `Py_NotImplemented`, if another error occurred it must return `NULL` and set an exception condition.

The following constants are defined to be used as the third argument for **`tp_richcompare`** and for **`PyObject_RichCompare()`**:

Constant	Comparison
Py_LT	`<`
Py_LE	`<=`
Py_EQ	`==`
Py_NE	`!=`
Py_GT	`>`
Py_GE	`>=`

The following macro is defined to ease writing rich comparison functions:

Py_RETURN_RICHCOMPARE(VAL_A, VAL_B, op)

Return `Py_True` or `Py_False` from the function, depending on the result of a comparison. VAL_A and VAL_B must be orderable by C comparison operators (for example, they may be C ints or floats). The third argument specifies the requested operation, as for **`PyObject_RichCompare()`**.

The returned value is a new strong reference.

On error, sets an exception and returns `NULL` from the function.

Added in version 3.7.

Inheritance:

Group: **`tp_hash`**, **`tp_richcompare`**

This field is inherited by subtypes together with **`tp_hash`**: a subtype inherits **`tp_richcompare`** and **`tp_hash`** when the subtype's **`tp_richcompare`** and **`tp_hash`** are both `NULL`.

Default:

`PyBaseObject_Type` provides a **`tp_richcompare`** implementation, which may be inherited. However, if only **`tp_hash`** is defined, not even the inherited function is used and instances of the type will not be able to participate in any comparisons.

Py_ssize_t PyTypeObject.tp_weaklistoffset

While this field is still supported, **`Py_TPFLAGS_MANAGED_WEAKREF`** should be used instead, if at all possible.

If the instances of this type are weakly referenceable, this field is greater than zero and contains the offset in the instance structure of the weak reference list head (ignoring the GC header, if present); this offset is used by **`PyObject_ClearWeakRefs()`** and the `PyWeakref_*` functions. The instance structure needs to include a field of type PyObject* which is initialized to `NULL`.

Do not confuse this field with **`tp_weaklist`**; that is the list head for weak references to the type object itself.

It is an error to set both the **`Py_TPFLAGS_MANAGED_WEAKREF`** bit and **`tp_weaklistoffset`**.

Inheritance:

This field is inherited by subtypes, but see the rules listed below. A subtype may override this offset; this means that the subtype uses a different weak reference list head than the base type. Since the list head is always found via **`tp_weaklistoffset`**, this should not be a problem.

Default:

If the **`Py_TPFLAGS_MANAGED_WEAKREF`** bit is set in the **`tp_flags`** field, then **`tp_weaklistoffset`** will be set to a negative value, to indicate that it is unsafe to use this field.

getiterfunc PyTypeObject.tp_iter

The corresponding slot ID `Py_tp_iter` is part of the Stable ABI.

An optional pointer to a function that returns an iterator for the object. Its presence normally signals that the instances of this type are iterable (although sequences may be iterable without this function).

This function has the same signature as **`PyObject_GetIter()`**:

```
PyObject *tp_iter(PyObject *self);
```

Inheritance:

This field is inherited by subtypes.

iternextfunc PyTypeObject.tp_iternext

The corresponding slot ID `Py_tp_iternext` is part of the Stable ABI.

An optional pointer to a function that returns the next item in an iterator. The signature is:

```
PyObject *tp_iternext(PyObject *self);
```

When the iterator is exhausted, it must return `NULL`; a **`StopIteration`** exception may or may not be set. When another error occurs, it must return `NULL` too. Its presence signals that the instances of this type are iterators.

Iterator types should also define the **`tp_iter`** function, and that function should return the iterator instance itself (not a new iterator instance).

This function has the same signature as **`PyIter_Next()`**.

Inheritance:

This field is inherited by subtypes.

struct PyMethodDef *PyTypeObject.tp_methods

The corresponding slot ID `Py_tp_methods` is part of the Stable ABI.

An optional pointer to a static `NULL`-terminated array of **`PyMethodDef`** structures, declaring regular methods of this type.

For each entry in the array, an entry is added to the type's dictionary (see **`tp_dict`** below) containing a method descriptor.

Inheritance:

This field is not inherited by subtypes (methods are inherited through a different mechanism).

struct PyMemberDef *PyTypeObject.tp_members

The corresponding slot ID `Py_tp_members` is part of the Stable ABI.

An optional pointer to a static `NULL`-terminated array of **`PyMemberDef`** structures, declaring regular data members (fields or slots) of instances of this type.

For each entry in the array, an entry is added to the type's dictionary (see `tp_dict` below) containing a member descriptor.

Inheritance:

This field is not inherited by subtypes (members are inherited through a different mechanism).

struct PyGetSetDef *PyTypeObject.tp_getset

The corresponding slot ID `Py_tp_getset` is part of the Stable ABI.

An optional pointer to a static `NULL`-terminated array of **`PyGetSetDef`** structures, declaring computed attributes of instances of this type.

For each entry in the array, an entry is added to the type's dictionary (see `tp_dict` below) containing a getset descriptor.

Inheritance:

This field is not inherited by subtypes (computed attributes are inherited through a different mechanism).

PyTypeObject *PyTypeObject.tp_base

The corresponding slot ID `Py_tp_base` is part of the Stable ABI.

An optional pointer to a base type from which type properties are inherited. At this level, only single inheritance is supported; multiple inheritance require dynamically creating a type object by calling the metatype.

Note

Slot initialization is subject to the rules of initializing globals. C99 requires the initializers to be "address constants". Function designators like `PyType_GenericNew()`, with implicit conversion to a pointer, are valid C99 address constants.

However, the unary '&' operator applied to a non-static variable like `PyBaseObject_Type` is not required to produce an address constant. Compilers may support this (gcc does), MSVC does not. Both compilers are strictly standard conforming in this particular behavior.

Consequently, `tp_base` should be set in the extension module's init function.

Inheritance:

This field is not inherited by subtypes (obviously).

Default:

This field defaults to `&PyBaseObject_Type` (which to Python programmers is known as the type `object`).

PyObject *PyTypeObject.tp_dict

The type's dictionary is stored here by `PyType_Ready()`.

This field should normally be initialized to `NULL` before PyType_Ready is called; it may also be initialized to a dictionary containing initial attributes for the type. Once `PyType_Ready()` has initialized the type, extra attributes for the type may be added to this dictionary only if they don't correspond to overloaded operations (like `__add__()`). Once initialization for the type has finished, this field should be treated as read-only.

Some types may not store their dictionary in this slot. Use `PyType_GetDict()` to retrieve the dictionary for an arbitrary type.

Changed in version 3.12: Internals detail: For static builtin types, this is always `NULL`. Instead, the dict for such types is stored on `PyInterpreterState`. Use `PyType_GetDict()` to get the dict for an arbitrary type.

Inheritance:

This field is not inherited by subtypes (though the attributes defined in here are inherited through a different mechanism).

Default:

If this field is `NULL`, **`PyType_Ready()`** will assign a new dictionary to it.

> **Warning**
>
> It is not safe to use **`PyDict_SetItem()`** on or otherwise modify **`tp_dict`** with the dictionary C-API.

descrgetfunc PyTypeObject.tp_descr_get

The corresponding slot ID `Py_tp_descr_get` is part of the Stable ABI.

An optional pointer to a “descriptor get” function.

The function signature is:

```
PyObject * tp_descr_get(PyObject *self, PyObject *obj, PyObject
*type);
```

Inheritance:

This field is inherited by subtypes.

descrsetfunc PyTypeObject.tp_descr_set

The corresponding slot ID `Py_tp_descr_set` is part of the Stable ABI.

An optional pointer to a function for setting and deleting a descriptor’s value.

The function signature is:

```
int tp_descr_set(PyObject *self, PyObject *obj, PyObject *value);
```

The *value* argument is set to `NULL` to delete the value.

Inheritance:

This field is inherited by subtypes.

Py_ssize_t PyTypeObject.tp_dictoffset

While this field is still supported, `Py_TPFLAGS_MANAGED_DICT` should be used instead, if at all possible.

If the instances of this type have a dictionary containing instance variables, this field is non-zero and contains the offset in the instances of the type of the instance variable dictionary; this offset is used by `PyObject_GenericGetAttr()`.

Do not confuse this field with `tp_dict`; that is the dictionary for attributes of the type object itself.

The value specifies the offset of the dictionary from the start of the instance structure.

The `tp_dictoffset` should be regarded as write-only. To get the pointer to the dictionary call `PyObject_GenericGetDict()`. Calling `PyObject_GenericGetDict()` may need to allocate memory for the dictionary, so it is may be more efficient to call `PyObject_GetAttr()` when accessing an attribute on the object.

It is an error to set both the `Py_TPFLAGS_MANAGED_DICT` bit and `tp_dictoffset`.

Inheritance:

This field is inherited by subtypes. A subtype should not override this offset; doing so could be unsafe, if C code tries to access the dictionary at the previous offset. To properly support inheritance, use `Py_TPFLAGS_MANAGED_DICT`.

Default:

This slot has no default. For static types, if the field is `NULL` then no `__dict__` gets created for instances.

If the `Py_TPFLAGS_MANAGED_DICT` bit is set in the `tp_flags` field, then `tp_dictoffset` will be set to `-1`, to indicate that it is unsafe to use this field.

initproc PyTypeObject.tp_init

The corresponding slot ID `Py_tp_init` *is part of the Stable ABI.*

An optional pointer to an instance initialization function.

This function corresponds to the `__init__()` method of classes. Like `__init__()`, it is possible to create an instance without calling `__init__()`, and it is possible to reinitialize an instance by calling its `__init__()` method again.

The function signature is:

```
int tp_init(PyObject *self, PyObject *args, PyObject *kwds);
```

The self argument is the instance to be initialized; the *args* and *kwds* arguments represent positional and keyword arguments of the call to `__init__()`.

The `tp_init` function, if not `NULL`, is called when an instance is created normally by calling its type, after the type's `tp_new` function has returned an instance of the type. If the `tp_new` function returns an instance of some other type that is not a subtype of the original type, no `tp_init` function is called; if `tp_new` returns an instance of a subtype of the original type, the subtype's `tp_init` is called.

Returns `0` on success, `-1` and sets an exception on error.

Inheritance:

This field is inherited by subtypes.

Default:

For static types this field does not have a default.

allocfunc PyTypeObject.tp_alloc

The corresponding slot ID `Py_tp_alloc` is part of the Stable ABI.

An optional pointer to an instance allocation function.

The function signature is:

```
PyObject *tp_alloc(PyTypeObject *self, Py_ssize_t nitems);
```

Inheritance:

Static subtypes inherit this slot, which will be `PyType_GenericAlloc()` if inherited from `object`.

Heap subtypes do not inherit this slot.

Default:

For heap subtypes, this field is always set to `PyType_GenericAlloc()`.

For static subtypes, this slot is inherited (see above).

newfunc PyTypeObject.tp_new

The corresponding slot ID `Py_tp_new` *is part of the Stable ABI.*

An optional pointer to an instance creation function.

The function signature is:

```
PyObject *tp_new(PyTypeObject *subtype, PyObject *args, PyObject
*kwds);
```

The *subtype* argument is the type of the object being created; the *args* and *kwds* arguments represent positional and keyword arguments of the call to the type. Note that *subtype* doesn't have to equal the type whose `tp_new` function is called; it may be a subtype of that type (but not an unrelated type).

The `tp_new` function should call `subtype->tp_alloc(subtype, nitems)` to allocate space for the object, and then do only as much further initialization as is absolutely necessary. Initialization that can safely be ignored or repeated should be placed in the `tp_init` handler. A good rule of thumb is that for immutable types, all initialization should take place in `tp_new`, while for mutable types, most initialization should be deferred to `tp_init`.

Set the `Py_TPFLAGS_DISALLOW_INSTANTIATION` flag to disallow creating instances of the type in Python.

Inheritance:

This field is inherited by subtypes, except it is not inherited by static types whose **`tp_base`** is `NULL` or `&PyBaseObject_Type`.

Default:

For static types this field has no default. This means if the slot is defined as `NULL`, the type cannot be called to create new instances; presumably there is some other way to create instances, like a factory function.

freefunc PyTypeObject.tp_free

The corresponding slot ID `Py_tp_free` is part of the Stable ABI.

An optional pointer to an instance deallocation function. Its signature is:

```
void tp_free(void *self);
```

This function must free the memory allocated by **`tp_alloc`**.

Inheritance:

Static subtypes inherit this slot, which will be **`PyObject_Free()`** if inherited from **`object`**. Exception: If the type supports garbage collection (i.e., the **`Py_TPFLAGS_HAVE_GC`** flag is set in **`tp_flags`**) and it would inherit **`PyObject_Free()`**, then this slot is not inherited but instead defaults to **`PyObject_GC_Del()`**.

Heap subtypes do not inherit this slot.

Default:

For heap subtypes, this slot defaults to a deallocator suitable to match **`PyType_GenericAlloc()`** and the value of the **`Py_TPFLAGS_HAVE_GC`** flag.

For static subtypes, this slot is inherited (see above).

inquiry PyTypeObject.tp_is_gc

The corresponding slot ID `Py_tp_is_gc` is part of the Stable ABI.

An optional pointer to a function called by the garbage collector.

The garbage collector needs to know whether a particular object is collectible or not. Normally, it is sufficient to look at the object's type's **`tp_flags`** field, and check the **`Py_TPFLAGS_HAVE_GC`** flag bit. But some types have a mixture of statically and dynamically allocated instances, and the statically allocated instances are not collectible. Such types should define this function; it should return `1` for a collectible instance, and `0` for a non-collectible instance. The signature is:

```
int tp_is_gc(PyObject *self);
```

(The only example of this are types themselves. The metatype, **`PyType_Type`**, defines this function to distinguish between statically and dynamically allocated types.)

Inheritance:

This field is inherited by subtypes.

Default:

This slot has no default. If this field is `NULL`, **`Py_TPFLAGS_HAVE_GC`** is used as the functional equivalent.

PyObject *PyTypeObject.tp_bases

The corresponding slot ID `Py_tp_bases` is part of the Stable ABI.

Tuple of base types.

This field should be set to `NULL` and treated as read-only. Python will fill it in when the type is **`initialized`**.

For dynamically created classes, the **`Py_tp_bases`** **`slot`** can be used instead of the *bases* argument of **`PyType_FromSpecWithBases()`**. The argument form is preferred.

Warning

Multiple inheritance does not work well for statically defined types. If you set `tp_bases` to a tuple, Python will not raise an error, but some slots will only be inherited from the first base.

Inheritance:

This field is not inherited.

PyObject *PyTypeObject.tp_mro

Tuple containing the expanded set of base types, starting with the type itself and ending with **`object`**, in Method Resolution Order.

This field should be set to `NULL` and treated as read-only. Python will fill it in when the type is **`initialized`**.

Inheritance:

This field is not inherited; it is calculated fresh by **`PyType_Ready()`**.

PyObject *PyTypeObject.tp_cache

Unused. Internal use only.

Inheritance:

This field is not inherited.

void *PyTypeObject.tp_subclasses

A collection of subclasses. Internal use only. May be an invalid pointer.

To get a list of subclasses, call the Python method **`__subclasses__()`**.

Changed in version 3.12: For some types, this field does not hold a valid PyObject*. The type was changed to void* to indicate this.

Inheritance:

This field is not inherited.

PyObject *PyTypeObject.tp_weaklist

Weak reference list head, for weak references to this type object. Not inherited. Internal use only.

Changed in version 3.12: Internals detail: For the static builtin types this is always `NULL`, even if weakrefs are added. Instead, the weakrefs for each are stored on `PyInterpreterState`. Use the public C-API or the internal `_PyObject_GET_WEAKREFS_LISTPTR()` macro to avoid the distinction.

Inheritance:

This field is not inherited.

destructor PyTypeObject.tp_del

The corresponding slot ID `Py_tp_del` *is part of the Stable ABI.*

This field is deprecated. Use **`tp_finalize`** instead.

unsigned int PyTypeObject.tp_version_tag

Used to index into the method cache. Internal use only.

Inheritance:

This field is not inherited.

destructor PyTypeObject.tp_finalize

The corresponding slot ID `Py_tp_finalize` *is part of the Stable ABI since version 3.5.*

An optional pointer to an instance finalization function. This is the C implementation of the **`__del__()`** special method. Its signature is:

```
void tp_finalize(PyObject *self);
```

The primary purpose of finalization is to perform any non-trivial cleanup that must be performed before the object is destroyed, while the object and any other objects it directly or indirectly references are still in a consistent state. The finalizer is allowed to execute arbitrary Python code.

Before Python automatically finalizes an object, some of the object's direct or indirect referents might have themselves been automatically finalized. However, none of the referents will have been automatically cleared (`tp_clear`) yet.

Other non-finalized objects might still be using a finalized object, so the finalizer must leave the object in a sane state (e.g., invariants are still met).

Note

After Python automatically finalizes an object, Python might start automatically clearing (`tp_clear`) the object and its referents (direct and indirect). Cleared objects are not guaranteed to be in a consistent state; a finalized object must be able to tolerate cleared referents.

Note

An object is not guaranteed to be automatically finalized before its destructor (`tp_dealloc`) is called. It is recommended to call `PyObject_CallFinalizerFromDealloc()` at the beginning of **`tp_dealloc`** to guarantee that the object is always finalized before destruction.

Note

The `tp_finalize` function can be called from any thread, although the GIL will be held.

Note

The **`tp_finalize`** function can be called during shutdown, after some global variables have been deleted. See the documentation of the `__del__()` method for details.

When Python finalizes an object, it behaves like the following algorithm:

1. Python might mark the object as *finalized*. Currently, Python always marks objects whose type supports garbage collection (i.e., the `Py_TPFLAGS_HAVE_GC` flag is set in `tp_flags`) and never marks other types of objects; this might change in a future version.
2. If the object is not marked as *finalized* and its **`tp_finalize`** finalizer function is non-`NULL`, the finalizer function is called.
3. If the finalizer function was called and the finalizer made the object reachable (i.e., there is a reference to the object and it is not a member of a cyclic isolate), then the finalizer is said to have *resurrected* the object. It is unspecified whether the finalizer can also resurrect the object by adding a new reference to the object that does not make it reachable, i.e., the object is (still) a member of a cyclic isolate.
4. If the finalizer resurrected the object, the object's pending destruction is canceled and the object's *finalized* mark might be removed if present. Currently, Python never removes the *finalized* mark; this might change in a future version.

Automatic finalization refers to any finalization performed by Python except via calls to `PyObject_CallFinalizer()` or `PyObject_CallFinalizerFromDealloc()`. No guarantees are made about when, if, or how often an object is automatically finalized, except:

- Python will not automatically finalize an object if it is reachable, i.e., there is a reference to it and it is not a member of a cyclic isolate.
- Python will not automatically finalize an object if finalizing it would not mark the object as *finalized*. Currently, this applies to objects whose type does not support garbage collection, i.e., the `Py_TPFLAGS_HAVE_GC` flag is not set. Such objects can still be manually finalized by calling `PyObject_CallFinalizer()` or `PyObject_CallFinalizerFromDealloc()`.

- Python will not automatically finalize any two members of a cyclic isolate concurrently.
- Python will not automatically finalize an object after it has automatically cleared (`tp_clear`) the object.
- If an object is a member of a cyclic isolate, Python will not automatically finalize it after automatically clearing (see `tp_clear`) any other member.
- Python will automatically finalize every member of a cyclic isolate before it automatically clears (see `tp_clear`) any of them.
- If Python is going to automatically clear an object (`tp_clear`), it will automatically finalize the object first.

Python currently only automatically finalizes objects that are members of a cyclic isolate, but future versions might finalize objects regularly before their destruction.

To manually finalize an object, do not call this function directly; call `PyObject_CallFinalizer()` or `PyObject_CallFinalizerFromDealloc()` instead.

`tp_finalize` should leave the current exception status unchanged. The recommended way to write a non-trivial finalizer is to back up the exception at the beginning by calling `PyErr_GetRaisedException()` and restore the exception at the end by calling `PyErr_SetRaisedException()`. If an exception is encountered in the middle of the finalizer, log and clear it with `PyErr_WriteUnraisable()` or `PyErr_FormatUnraisable()`. For example:

```
static void
foo_finalize(PyObject *self)
{
    // Save the current exception, if any.
    PyObject *exc = PyErr_GetRaisedException();

    // ...

    if (do_something_that_might_raise() != success_indicator) {
        PyErr_WriteUnraisable(self);
        goto done;
    }

done:
    // Restore the saved exception.  This silently discards any
exception
    // raised above, so be sure to call PyErr_WriteUnraisable
first if
    // necessary.
    PyErr_SetRaisedException(exc);
}
```

Inheritance:

This field is inherited by subtypes.

Added in version 3.4.

Changed in version 3.8: Before version 3.8 it was necessary to set the **`Py_TPFLAGS_HAVE_FINALIZE`** flags bit in order for this field to be used. This is no longer required.

See also

- **PEP 442** [https://peps.python.org/pep-0442/]: “Safe object finalization”
- Object Life Cycle for details about how this slot relates to other slots.
- **`PyObject_CallFinalizer()`**
- **`PyObject_CallFinalizerFromDealloc()`**

vectorcallfunc PyTypeObject.tp_vectorcall

The corresponding slot ID `Py_tp_vectorcall` is part of the Stable ABI since version 3.14.

A vectorcall function to use for calls of this type object (rather than instances). In other words, `tp_vectorcall` can be used to optimize `type.__call__`, which typically returns a new instance of *type*.

As with any vectorcall function, if `tp_vectorcall` is `NULL`, the *tp_call* protocol (`Py_TYPE(type)->tp_call`) is used instead.

Note

The vectorcall protocol requires that the vectorcall function has the same behavior as the corresponding `tp_call`. This means that `type->tp_vectorcall` must match the behavior of `Py_TYPE(type)->tp_call`.

Specifically, if *type* uses the default metaclass, `type->tp_vectorcall` must behave the same as PyType_Type->tp_call, which:

- calls `type->tp_new`,
- if the result is a subclass of *type*, calls `type->tp_init` on the result of `tp_new`, and
- returns the result of `tp_new`.

Typically, `tp_vectorcall` is overridden to optimize this process for specific **`tp_new`** and **`tp_init`**. When doing this for user-subclassable types, note that both can be overridden (using **`__new__()`** and **`__init__()`**, respectively).

Inheritance:

This field is never inherited.

Added in version 3.9: (the field exists since 3.8 but it’s only used since 3.9)

unsigned char PyTypeObject.tp_watched

Internal. Do not use.

Added in version 3.12.

Static Types

Traditionally, types defined in C code are *static*, that is, a static **`PyTypeObject`** structure is defined directly in code and initialized using **`PyType_Ready()`**.

This results in types that are limited relative to types defined in Python:

- Static types are limited to one base, i.e. they cannot use multiple inheritance.

- Static type objects (but not necessarily their instances) are immutable. It is not possible to add or modify the type object’s attributes from Python.
- Static type objects are shared across sub-interpreters, so they should not include any subinterpreter-specific state.

Also, since **`PyTypeObject`** is only part of the Limited API as an opaque struct, any extension modules using static types must be compiled for a specific Python minor version.

Heap Types

An alternative to static types is *heap-allocated types*, or *heap types* for short, which correspond closely to classes created by Python’s `class` statement. Heap types have the **`Py_TPFLAGS_HEAPTYPE`** flag set.

This is done by filling a **`PyType_Spec`** structure and calling **`PyType_FromSpec()`**, **`PyType_FromSpecWithBases()`**, **`PyType_FromModuleAndSpec()`**, or **`PyType_FromMetaclass()`**.

Number Object Structures

type PyNumberMethods

This structure holds pointers to the functions which an object uses to implement the number protocol. Each function is used by the function of similar name documented in the Number Protocol section.

Here is the structure definition:

```
typedef struct {
     binaryfunc nb_add;
     binaryfunc nb_subtract;
     binaryfunc nb_multiply;
     binaryfunc nb_remainder;
     binaryfunc nb_divmod;
     ternaryfunc nb_power;
     unaryfunc nb_negative;
     unaryfunc nb_positive;
     unaryfunc nb_absolute;
```

```
    inquiry nb_bool;
    unaryfunc nb_invert;
    binaryfunc nb_lshift;
    binaryfunc nb_rshift;
    binaryfunc nb_and;
    binaryfunc nb_xor;
    binaryfunc nb_or;
    unaryfunc nb_int;
    void *nb_reserved;
    unaryfunc nb_float;

    binaryfunc nb_inplace_add;
    binaryfunc nb_inplace_subtract;
    binaryfunc nb_inplace_multiply;
    binaryfunc nb_inplace_remainder;
    ternaryfunc nb_inplace_power;
    binaryfunc nb_inplace_lshift;
    binaryfunc nb_inplace_rshift;
    binaryfunc nb_inplace_and;
    binaryfunc nb_inplace_xor;
    binaryfunc nb_inplace_or;

    binaryfunc nb_floor_divide;
    binaryfunc nb_true_divide;
    binaryfunc nb_inplace_floor_divide;
    binaryfunc nb_inplace_true_divide;

    unaryfunc nb_index;

    binaryfunc nb_matrix_multiply;
    binaryfunc nb_inplace_matrix_multiply;
} PyNumberMethods;
```

Note

Binary and ternary functions must check the type of all their operands, and implement the necessary conversions (at least one of the operands is an instance of the defined type). If the operation is not defined for the given operands, binary and ternary functions must return `Py_NotImplemented`, if another error occurred they must return `NULL` and set an exception.

Note

The **`nb_reserved`** field should always be `NULL`. It was previously called **`nb_long`**, and was renamed in Python 3.0.1.

binaryfunc PyNumberMethods.nb_add

The corresponding slot ID `Py_nb_add` is part of the Stable ABI.

binaryfunc PyNumberMethods.nb_subtract

The corresponding slot ID `Py_nb_subtract` is part of the Stable ABI.

binaryfunc PyNumberMethods.nb_multiply

The corresponding slot ID `Py_nb_multiply` is part of the Stable ABI.

binaryfunc PyNumberMethods.nb_remainder

The corresponding slot ID `Py_nb_remainder` is part of the Stable ABI.

binaryfunc PyNumberMethods.nb_divmod

The corresponding slot ID `Py_nb_divmod` is part of the Stable ABI.

ternaryfunc PyNumberMethods.nb_power

The corresponding slot ID `Py_nb_power` is part of the Stable ABI.

unaryfunc PyNumberMethods.nb_negative

The corresponding slot ID `Py_nb_negative` is part of the Stable ABI.

unaryfunc PyNumberMethods.nb_positive

The corresponding slot ID `Py_nb_positive` is part of the Stable ABI.

unaryfunc PyNumberMethods.nb_absolute

The corresponding slot ID `Py_nb_absolute` is part of the Stable ABI.

inquiry PyNumberMethods.nb_bool

The corresponding slot ID `Py_nb_bool` is part of the Stable ABI.

unaryfunc PyNumberMethods.nb_invert

The corresponding slot ID `Py_nb_invert` *is part of the Stable ABI.*

binaryfunc PyNumberMethods.nb_lshift

The corresponding slot ID `Py_nb_lshift` *is part of the Stable ABI.*

binaryfunc PyNumberMethods.nb_rshift

The corresponding slot ID `Py_nb_rshift` *is part of the Stable ABI.*

binaryfunc PyNumberMethods.nb_and

The corresponding slot ID `Py_nb_and` *is part of the Stable ABI.*

binaryfunc PyNumberMethods.nb_xor

The corresponding slot ID `Py_nb_xor` *is part of the Stable ABI.*

binaryfunc PyNumberMethods.nb_or

The corresponding slot ID `Py_nb_or` *is part of the Stable ABI.*

unaryfunc PyNumberMethods.nb_int

The corresponding slot ID `Py_nb_int` *is part of the Stable ABI.*

void *PyNumberMethods.nb_reserved

unaryfunc PyNumberMethods.nb_float

The corresponding slot ID `Py_nb_float` *is part of the Stable ABI.*

binaryfunc PyNumberMethods.nb_inplace_add

The corresponding slot ID `Py_nb_inplace_add` *is part of the Stable ABI.*

binaryfunc PyNumberMethods.nb_inplace_subtract

The corresponding slot ID `Py_nb_inplace_subtract` *is part of the Stable ABI.*

binaryfunc PyNumberMethods.nb_inplace_multiply

The corresponding slot ID `Py_nb_inplace_multiply` *is part of the Stable ABI.*

binaryfunc PyNumberMethods.nb_inplace_remainder

The corresponding slot ID `Py_nb_inplace_remainder` is part of the Stable ABI.

ternaryfunc PyNumberMethods.nb_inplace_power

The corresponding slot ID `Py_nb_inplace_power` is part of the Stable ABI.

binaryfunc PyNumberMethods.nb_inplace_lshift

The corresponding slot ID `Py_nb_inplace_lshift` is part of the Stable ABI.

binaryfunc PyNumberMethods.nb_inplace_rshift

The corresponding slot ID `Py_nb_inplace_rshift` is part of the Stable ABI.

binaryfunc PyNumberMethods.nb_inplace_and

The corresponding slot ID `Py_nb_inplace_and` is part of the Stable ABI.

binaryfunc PyNumberMethods.nb_inplace_xor

The corresponding slot ID `Py_nb_inplace_xor` is part of the Stable ABI.

binaryfunc PyNumberMethods.nb_inplace_or

The corresponding slot ID `Py_nb_inplace_or` is part of the Stable ABI.

binaryfunc PyNumberMethods.nb_floor_divide

The corresponding slot ID `Py_nb_floor_divide` is part of the Stable ABI.

binaryfunc PyNumberMethods.nb_true_divide

The corresponding slot ID `Py_nb_true_divide` is part of the Stable ABI.

binaryfunc PyNumberMethods.nb_inplace_floor_divide

The corresponding slot ID `Py_nb_inplace_floor_divide` is part of the Stable ABI.

binaryfunc PyNumberMethods.nb_inplace_true_divide

The corresponding slot ID `Py_nb_inplace_true_divide` is part of the Stable ABI.

unaryfunc PyNumberMethods.nb_index

The corresponding slot ID `Py_nb_index` is part of the Stable ABI.

binaryfunc PyNumberMethods.nb_matrix_multiply

The corresponding slot ID `Py_nb_matrix_multiply` is part of the Stable ABI since version 3.5.

binaryfunc PyNumberMethods.nb_inplace_matrix_multiply

The corresponding slot ID `Py_nb_inplace_matrix_multiply` is part of the Stable ABI since version 3.5.

Mapping Object Structures

type PyMappingMethods

This structure holds pointers to the functions which an object uses to implement the mapping protocol. It has three members:

lenfunc PyMappingMethods.mp_length

The corresponding slot ID `Py_mp_length` is part of the Stable ABI.

This function is used by **`PyMapping_Size()`** and **`PyObject_Size()`**, and has the same signature. This slot may be set to `NULL` if the object has no defined length.

binaryfunc PyMappingMethods.mp_subscript

The corresponding slot ID `Py_mp_subscript` is part of the Stable ABI.

This function is used by **`PyObject_GetItem()`** and **`PySequence_GetSlice()`**, and has the same signature as **`PyObject_GetItem()`**. This slot must be filled for the **`PyMapping_Check()`** function to return `1`, it can be `NULL` otherwise.

objobjargproc PyMappingMethods.mp_ass_subscript

The corresponding slot ID `Py_mp_ass_subscript` is part of the Stable ABI.

This function is used by **`PyObject_SetItem()`**, **`PyObject_DelItem()`**, **`PySequence_SetSlice()`** and **`PySequence_DelSlice()`**. It has the same signature as **`PyObject_SetItem()`**, but *v* can also be set to `NULL` to delete an item. If this slot is `NULL`, the object does not support item assignment and deletion.

Sequence Object Structures

type PySequenceMethods

This structure holds pointers to the functions which an object uses to implement the sequence protocol.

lenfunc PySequenceMethods.sq_length

The corresponding slot ID `Py_sq_length` is part of the Stable ABI.

This function is used by **`PySequence_Size()`** and **`PyObject_Size()`**, and has the same signature. It is also used for handling negative indices via the **`sq_item`** and the **`sq_ass_item`** slots.

binaryfunc PySequenceMethods.sq_concat

The corresponding slot ID `Py_sq_concat` is part of the Stable ABI.

This function is used by **`PySequence_Concat()`** and has the same signature. It is also used by the `+` operator, after trying the numeric addition via the **`nb_add`** slot.

ssizeargfunc PySequenceMethods.sq_repeat

The corresponding slot ID `Py_sq_repeat` is part of the Stable ABI.

This function is used by **`PySequence_Repeat()`** and has the same signature. It is also used by the `*` operator, after trying numeric multiplication via the **`nb_multiply`** slot.

ssizeargfunc PySequenceMethods.sq_item

The corresponding slot ID `Py_sq_item` is part of the Stable ABI.

This function is used by **`PySequence_GetItem()`** and has the same signature. It is also used by **`PyObject_GetItem()`**, after trying the subscription via the **`mp_subscript`** slot. This slot must be filled for the `PySequence_Check()` function to return `1`, it can be `NULL` otherwise.

Negative indexes are handled as follows: if the `sq_length` slot is filled, it is called and the sequence length is used to compute a positive index which is passed to `sq_item`. If **`sq_length`** is `NULL`, the index is passed as is to the function.

ssizeobjargproc PySequenceMethods.sq_ass_item

The corresponding slot ID `Py_sq_ass_item` is part of the Stable ABI.

This function is used by `PySequence_SetItem()` and has the same signature. It is also used by `PyObject_SetItem()` and `PyObject_DelItem()`, after trying the item assignment and deletion via the `mp_ass_subscript` slot. This slot may be left to `NULL` if the object does not support item assignment and deletion.

objobjproc PySequenceMethods.sq_contains

The corresponding slot ID `Py_sq_contains` is part of the Stable ABI.

This function may be used by `PySequence_Contains()` and has the same signature. This slot may be left to `NULL`, in this case **`PySequence_Contains()`** simply traverses the sequence until it finds a match.

binaryfunc PySequenceMethods.sq_inplace_concat

The corresponding slot ID `Py_sq_inplace_concat` is part of the Stable ABI.

This function is used by `PySequence_InPlaceConcat()` and has the same signature. It should modify its first operand, and return it. This slot may be left to `NULL`, in this case **`PySequence_InPlaceConcat()`** will fall back to `PySequence_Concat()`. It is also used by the augmented assignment `+=`, after trying numeric in-place addition via the `nb_inplace_add` slot.

ssizeargfunc PySequenceMethods.sq_inplace_repeat

The corresponding slot ID `Py_sq_inplace_repeat` is part of the Stable ABI.

This function is used by **`PySequence_InPlaceRepeat()`** and has the same signature. It should modify its first operand, and return it. This slot may be left to `NULL`, in this case **`PySequence_InPlaceRepeat()`** will fall back to **`PySequence_Repeat()`**. It is also used by the augmented assignment `*=`, after trying numeric in-place multiplication via the **`nb_inplace_multiply`** slot.

Buffer Object Structures

type PyBufferProcs

This structure holds pointers to the functions required by the Buffer protocol. The protocol defines how an exporter object can expose its internal data to consumer objects.

getbufferproc PyBufferProcs.bf_getbuffer

The corresponding slot ID `Py_bf_getbuffer` is part of the Stable ABI since version 3.11.

The signature of this function is:

```
int (PyObject *exporter, Py_buffer *view, int flags);
```

Handle a request to *exporter* to fill in *view* as specified by *flags*. Except for point (3), an implementation of this function MUST take these steps:

1. Check if the request can be met. If not, raise **`BufferError`**, set view->obj to `NULL` and return `-1`.
2. Fill in the requested fields.
3. Increment an internal counter for the number of exports.
4. Set view->obj to *exporter* and increment view->obj.
5. Return `0`.

If *exporter* is part of a chain or tree of buffer providers, two main schemes can be used:

- Re-export: Each member of the tree acts as the exporting object and sets view->obj to a new reference to itself.

- Redirect: The buffer request is redirected to the root object of the tree. Here, view->obj will be a new reference to the root object.

The individual fields of *view* are described in section Buffer structure, the rules how an exporter must react to specific requests are in section Buffer request types.

All memory pointed to in the **`Py_buffer`** structure belongs to the exporter and must remain valid until there are no consumers left. **`format`**, **`shape`**, **`strides`**, **`suboffsets`** and **`internal`** are read-only for the consumer.

`PyBuffer_FillInfo()` provides an easy way of exposing a simple bytes buffer while dealing correctly with all request types.

`PyObject_GetBuffer()` is the interface for the consumer that wraps this function.

releasebufferproc PyBufferProcs.bf_releasebuffer

The corresponding slot ID `Py_bf_releasebuffer` *is part of the Stable ABI since version 3.11.*

The signature of this function is:

```
void (PyObject *exporter, Py_buffer *view);
```

Handle a request to release the resources of the buffer. If no resources need to be released, **`PyBufferProcs.bf_releasebuffer`** may be `NULL`. Otherwise, a standard implementation of this function will take these optional steps:

1. Decrement an internal counter for the number of exports.
2. If the counter is `0`, free all memory associated with *view*.

The exporter MUST use the **`internal`** field to keep track of buffer-specific resources. This field is guaranteed to remain constant, while a consumer MAY pass a copy of the original buffer as the *view* argument.

This function MUST NOT decrement view->obj, since that is done automatically in **`PyBuffer_Release()`** (this scheme is useful for breaking reference cycles).

`PyBuffer_Release()` is the interface for the consumer that wraps this function.

Async Object Structures

Added in version 3.5.

type PyAsyncMethods

This structure holds pointers to the functions required to implement awaitable and asynchronous iterator objects.

Here is the structure definition:

```
typedef struct {
    unaryfunc am_await;
    unaryfunc am_aiter;
    unaryfunc am_anext;
    sendfunc am_send;
} PyAsyncMethods;
```

unaryfunc PyAsyncMethods.am_await

The corresponding slot ID `Py_am_await` *is part of the Stable ABI since version 3.5.*

The signature of this function is:

```
PyObject *am_await(PyObject *self);
```

The returned object must be an iterator, i.e. `PyIter_Check()` must return `1` for it.

This slot may be set to `NULL` if an object is not an awaitable.

unaryfunc PyAsyncMethods.am_aiter

The corresponding slot ID `Py_am_aiter` *is part of the Stable ABI since version 3.5.*

The signature of this function is:

```
PyObject *am_aiter(PyObject *self);
```

Must return an asynchronous iterator object. See **`__anext__()`** for details.

This slot may be set to `NULL` if an object does not implement asynchronous iteration protocol.

unaryfunc PyAsyncMethods.am_anext

The corresponding slot ID `Py_am_anext` is part of the Stable ABI since version 3.5.

The signature of this function is:

```
PyObject *am_anext(PyObject *self);
```

Must return an awaitable object. See **`__anext__()`** for details. This slot may be set to `NULL`.

sendfunc PyAsyncMethods.am_send

The corresponding slot ID `Py_am_send` is part of the Stable ABI since version 3.10.

The signature of this function is:

```
PySendResult am_send(PyObject *self, PyObject *arg, PyObject
**result);
```

See **`PyIter_Send()`** for details. This slot may be set to `NULL`.

Added in version 3.10.

Slot Type typedefs

typedef PyObject *(*allocfunc)(PyTypeObject *cls, Py_ssize_t nitems)

Part of the Stable ABI.

The purpose of this function is to separate memory allocation from memory initialization. It should return a pointer to a block of memory of adequate length for the instance, suitably aligned, and initialized to zeros, but with **`ob_refcnt`** set to `1` and **`ob_type`** set to the type argument. If the type's **`tp_itemsize`** is non-zero, the object's **`ob_size`** field should be initialized to *nitems* and the length of the allocated memory block should be `tp_basicsize + nitems*tp_itemsize`, rounded up to a multiple of `sizeof(void*)`; otherwise, *nitems* is not used and the length of the block should be **`tp_basicsize`**.

This function should not do any other instance initialization, not even to allocate additional memory; that should be done by **`tp_new`**.

typedef void (*destructor)(PyObject*)

Part of the Stable ABI.

typedef void (*freefunc)(void*)

See **`tp_free`**.

typedef PyObject *(*newfunc)(PyTypeObject*, PyObject*, PyObject*)

Part of the Stable ABI.
See **`tp_new`**.

typedef int (*initproc)(PyObject*, PyObject*, PyObject*)

Part of the Stable ABI.
See **`tp_init`**.

typedef PyObject *(*reprfunc)(PyObject*)

Part of the Stable ABI.
See **`tp_repr`**.

typedef PyObject *(*getattrfunc)(PyObject *self, char *attr)

Part of the Stable ABI.
Return the value of the named attribute for the object.

typedef int (*setattrfunc)(PyObject *self, char *attr, PyObject *value)

Part of the Stable ABI.
Set the value of the named attribute for the object. The value argument is set to `NULL` to delete the attribute.

typedef PyObject *(*getattrofunc)(PyObject *self, PyObject *attr)

Part of the Stable ABI.

Return the value of the named attribute for the object.

See **`tp_getattro`**.

typedef int (*setattrofunc)(PyObject *self, PyObject *attr, PyObject *value)

Part of the Stable ABI.
Set the value of the named attribute for the object. The value argument is set to `NULL` to delete the attribute.

See **`tp_setattro`**.

typedef PyObject *(*descrgetfunc)(PyObject*, PyObject*, PyObject*)

Part of the Stable ABI.
See **`tp_descr_get`**.

typedef int (*descrsetfunc)(PyObject*, PyObject*, PyObject*)

Part of the Stable ABI.
See **`tp_descr_set`**.

typedef Py_hash_t (*hashfunc)(PyObject*)

Part of the Stable ABI.
See **`tp_hash`**.

typedef PyObject *(*richcmpfunc)(PyObject*, PyObject*, int)

Part of the Stable ABI.
See **`tp_richcompare`**.

typedef PyObject *(*getiterfunc)(PyObject*)

Part of the Stable ABI.
See **`tp_iter`**.

typedef PyObject *(*iternextfunc)(PyObject*)

Part of the Stable ABI.
See **`tp_iternext`**.

typedef Py_ssize_t (*lenfunc)(PyObject*)

Part of the Stable ABI.

typedef int (*getbufferproc)(PyObject*, Py_buffer*, int)

Part of the Stable ABI since version 3.12.

typedef void (*releasebufferproc)(PyObject*, Py_buffer*)

Part of the Stable ABI since version 3.12.

typedef PyObject *(*unaryfunc)(PyObject*)

Part of the Stable ABI.

typedef PyObject *(*binaryfunc)(PyObject*, PyObject*)

Part of the Stable ABI.

typedef PySendResult (*sendfunc)(PyObject*, PyObject*, PyObject**)

See **`am_send`**.

typedef PyObject *(*ternaryfunc)(PyObject*, PyObject*, PyObject*)

Part of the Stable ABI.

typedef PyObject *(*ssizeargfunc)(PyObject*, Py_ssize_t)

Part of the Stable ABI.

typedef int (*ssizeobjargproc)(PyObject*, Py_ssize_t, PyObject*)

Part of the Stable ABI.

typedef int (*objobjproc)(PyObject*, PyObject*)

Part of the Stable ABI.

typedef int (*objobjargproc)(PyObject*, PyObject*, PyObject*)

Part of the Stable ABI.

Examples

The following are simple examples of Python type definitions. They include common usage you may encounter. Some demonstrate tricky corner cases. For more examples, practical info, and a tutorial, see Defining Extension Types: Tutorial and Defining Extension Types: Assorted Topics.

A basic static type:

```
typedef struct {
    PyObject_HEAD
    const char *data;
} MyObject;

static PyTypeObject MyObject_Type = {
    PyVarObject_HEAD_INIT(NULL, 0)
    .tp_name = "mymod.MyObject",
    .tp_basicsize = sizeof(MyObject),
    .tp_doc = PyDoc_STR("My objects"),
    .tp_new = myobj_new,
    .tp_dealloc = (destructor)myobj_dealloc,
    .tp_repr = (reprfunc)myobj_repr,
};
```

You may also find older code (especially in the CPython code base) with a more verbose initializer:

```
static PyTypeObject MyObject_Type = {
    PyVarObject_HEAD_INIT(NULL, 0)
    "mymod.MyObject",               /* tp_name */
    sizeof(MyObject),               /* tp_basicsize */
    0,                              /* tp_itemsize */
    (destructor)myobj_dealloc,      /* tp_dealloc */
    0,                              /* tp_vectorcall_offset */
    0,                              /* tp_getattr */
    0,                              /* tp_setattr */
    0,                              /* tp_as_async */
    (reprfunc)myobj_repr,           /* tp_repr */
    0,                              /* tp_as_number */
    0,                              /* tp_as_sequence */
    0,                              /* tp_as_mapping */
    0,                              /* tp_hash */
    0,                              /* tp_call */
    0,                              /* tp_str */
    0,                              /* tp_getattro */
    0,                              /* tp_setattro */
    0,                              /* tp_as_buffer */
    0,                              /* tp_flags */
    PyDoc_STR("My objects"),        /* tp_doc */
    0,                              /* tp_traverse */
    0,                              /* tp_clear */
    0,                              /* tp_richcompare */
    0,                              /* tp_weaklistoffset */
    0,                              /* tp_iter */
    0,                              /* tp_iternext */
    0,                              /* tp_methods */
    0,                              /* tp_members */
    0,                              /* tp_getset */
    0,                              /* tp_base */
    0,                              /* tp_dict */
    0,                              /* tp_descr_get */
    0,                              /* tp_descr_set */
    0,                              /* tp_dictoffset */
    0,                              /* tp_init */
    0,                              /* tp_alloc */
    myobj_new,                      /* tp_new */
};
```

A type that supports weakrefs, instance dicts, and hashing:

```
typedef struct {
    PyObject_HEAD
    const char *data;
} MyObject;

static PyTypeObject MyObject_Type = {
    PyVarObject_HEAD_INIT(NULL, 0)
    .tp_name = "mymod.MyObject",
    .tp_basicsize = sizeof(MyObject),
    .tp_doc = PyDoc_STR("My objects"),
    .tp_flags = Py_TPFLAGS_DEFAULT | Py_TPFLAGS_BASETYPE |
         Py_TPFLAGS_HAVE_GC | Py_TPFLAGS_MANAGED_DICT |
         Py_TPFLAGS_MANAGED_WEAKREF,
    .tp_new = myobj_new,
    .tp_traverse = (traverseproc)myobj_traverse,
    .tp_clear = (inquiry)myobj_clear,
    .tp_alloc = PyType_GenericNew,
    .tp_dealloc = (destructor)myobj_dealloc,
    .tp_repr = (reprfunc)myobj_repr,
    .tp_hash = (hashfunc)myobj_hash,
    .tp_richcompare = PyBaseObject_Type.tp_richcompare,
};
```

A str subclass that cannot be subclassed and cannot be called to create instances (e.g. uses a separate factory func) using `Py_TPFLAGS_DISALLOW_INSTANTIATION` flag:

```
typedef struct {
    PyUnicodeObject raw;
    char *extra;
} MyStr;

static PyTypeObject MyStr_Type = {
    PyVarObject_HEAD_INIT(NULL, 0)
    .tp_name = "mymod.MyStr",
    .tp_basicsize = sizeof(MyStr),
    .tp_base = NULL,  // set to &PyUnicode_Type in module init
    .tp_doc = PyDoc_STR("my custom str"),
    .tp_flags = Py_TPFLAGS_DEFAULT |
Py_TPFLAGS_DISALLOW_INSTANTIATION,
    .tp_repr = (reprfunc)myobj_repr,
};
```

The simplest static type with fixed-length instances:

```
typedef struct {
    PyObject_HEAD
} MyObject;

static PyTypeObject MyObject_Type = {
    PyVarObject_HEAD_INIT(NULL, 0)
    .tp_name = "mymod.MyObject",
};
```

The simplest static type with variable-length instances:

```
typedef struct {
    PyObject_VAR_HEAD
    const char *data[1];
} MyObject;

static PyTypeObject MyObject_Type = {
    PyVarObject_HEAD_INIT(NULL, 0)
    .tp_name = "mymod.MyObject",
    .tp_basicsize = sizeof(MyObject) - sizeof(char *),
    .tp_itemsize = sizeof(char *),
};
```

Supporting Cyclic Garbage Collection

Python’s support for detecting and collecting garbage which involves circular references requires support from object types which are “containers” for other objects which may also be containers. Types which do not store references to other objects, or which only store references to atomic types (such as numbers or strings), do not need to provide any explicit support for garbage collection.

To create a container type, the `tp_flags` field of the type object must include the `Py_TPFLAGS_HAVE_GC` and provide an implementation of the `tp_traverse` handler. If instances of the type are mutable, a `tp_clear` implementation must also be provided.

`Py_TPFLAGS_HAVE_GC`

Objects with a type with this flag set must conform with the rules documented here. For convenience these objects will be referred to as container objects.

Constructors for container types must conform to two rules:

1. The memory for the object must be allocated using `PyObject_GC_New` or `PyObject_GC_NewVar`.
2. Once all the fields which may contain references to other containers are initialized, it must call `PyObject_GC_Track()`.

Similarly, the deallocator for the object must conform to a similar pair of rules:

1. Before fields which refer to other containers are invalidated, `PyObject_GC_UnTrack()` must be called.

2. The object’s memory must be deallocated using `PyObject_GC_Del()`.

Warning

If a type adds the Py_TPFLAGS_HAVE_GC, then it *must* implement at least a `tp_traverse` handler or explicitly use one from its subclass or subclasses.

When calling `PyType_Ready()` or some of the APIs that indirectly call it like `PyType_FromSpecWithBases()` or `PyType_FromSpec()` the interpreter will automatically populate the `tp_flags`, `tp_traverse` and `tp_clear` fields if the type inherits from a class that implements the garbage collector protocol and the child class does *not* include the `Py_TPFLAGS_HAVE_GC` flag.

PyObject_GC_New(TYPE, typeobj)

Analogous to `PyObject_New` but for container objects with the `Py_TPFLAGS_HAVE_GC` flag set.

Do not call this directly to allocate memory for an object; call the type’s `tp_alloc` slot instead.

When populating a type’s `tp_alloc` slot, `PyType_GenericAlloc()` is preferred over a custom function that simply calls this macro.

Memory allocated by this macro must be freed with `PyObject_GC_Del()` (usually called via the object’s `tp_free` slot).

See also

- `PyObject_GC_Del()`
- `PyObject_New`
- `PyType_GenericAlloc()`
- `tp_alloc`

PyObject_GC_NewVar(TYPE, typeobj, size)

Analogous to `PyObject_NewVar` but for container objects with the `Py_TPFLAGS_HAVE_GC` flag set.

Do not call this directly to allocate memory for an object; call the type's `tp_alloc` slot instead.

When populating a type's `tp_alloc` slot, `PyType_GenericAlloc()` is preferred over a custom function that simply calls this macro.

Memory allocated by this macro must be freed with `PyObject_GC_Del()` (usually called via the object's `tp_free` slot).

See also

- `PyObject_GC_Del()`
- `PyObject_NewVar`
- `PyType_GenericAlloc()`
- `tp_alloc`

PyObject *PyUnstable_Object_GC_NewWithExtraData(PyTypeObject *type, size_t extra_size)

This is Unstable API. It may change without warning in minor releases.

Analogous to `PyObject_GC_New` but allocates *extra_size* bytes at the end of the object (at offset `tp_basicsize`). The allocated memory is initialized to zeros, except for the `Python object header`.

The extra data will be deallocated with the object, but otherwise it is not managed by Python.

Memory allocated by this function must be freed with `PyObject_GC_Del()` (usually called via the object's `tp_free` slot).

Warning

The function is marked as unstable because the final mechanism for reserving extra data after an instance is not yet decided. For allocating a variable number of fields, prefer using `PyVarObject` and `tp_itemsize` instead.

Added in version 3.12.

PyObject_GC_Resize(TYPE, op, newsize)

Resize an object allocated by `PyObject_NewVar`. Returns the resized object of type `TYPE*` (refers to any C type) or `NULL` on failure.

op must be of type PyVarObject* and must not be tracked by the collector yet. *newsize* must be of type `Py_ssize_t`.

void PyObject_GC_Track(PyObject *op)

Part of the Stable ABI.

Adds the object *op* to the set of container objects tracked by the collector. The collector can run at unexpected times so objects must be valid while being tracked. This should be called once all the fields followed by the `tp_traverse` handler become valid, usually near the end of the constructor.

int PyObject_IS_GC(PyObject *obj)

Returns non-zero if the object implements the garbage collector protocol, otherwise returns 0.

The object cannot be tracked by the garbage collector if this function returns 0.

int PyObject_GC_IsTracked(PyObject *op)

Part of the Stable ABI since version 3.9.
Returns 1 if the object type of *op* implements the GC protocol and *op* is being currently tracked by the garbage collector and 0 otherwise.

This is analogous to the Python function `gc.is_tracked()`.

Added in version 3.9.

int PyObject_GC_IsFinalized(PyObject *op)

Part of the Stable ABI since version 3.9.
Returns 1 if the object type of *op* implements the GC protocol and *op* has been already finalized by the garbage collector and 0 otherwise.

This is analogous to the Python function `gc.is_finalized()`.

Added in version 3.9.

void PyObject_GC_Del(void *op)

Part of the Stable ABI.
Releases memory allocated to an object using `PyObject_GC_New` or `PyObject_GC_NewVar`.

Do not call this directly to free an object’s memory; call the type’s `tp_free` slot instead.

Do not use this for memory allocated by `PyObject_New`, `PyObject_NewVar`, or related allocation functions; use `PyObject_Free()` instead.

See also

- `PyObject_Free()` is the non-GC equivalent of this function.
- `PyObject_GC_New`
- `PyObject_GC_NewVar`
- `PyType_GenericAlloc()`
- `tp_free`

void PyObject_GC_UnTrack(void *op)

Part of the Stable ABI.
Remove the object *op* from the set of container objects tracked by the collector. Note that `PyObject_GC_Track()` can be called again on this object to add it back to the set of tracked objects. The deallocator (`tp_dealloc` handler) should call this for the object before any of the fields used by the `tp_traverse` handler become invalid.

Changed in version 3.8: The `_PyObject_GC_TRACK()` and `_PyObject_GC_UNTRACK()` macros have been removed from the public C API.

The `tp_traverse` handler accepts a function parameter of this type:

typedef int (*visitproc)(PyObject *object, void *arg)

Part of the Stable ABI.
Type of the visitor function passed to the `tp_traverse` handler. The function should be called with an object to

traverse as *object* and the third parameter to the `tp_traverse` handler as *arg*. The Python core uses several visitor functions to implement cyclic garbage detection; it’s not expected that users will need to write their own visitor functions.

The `tp_traverse` handler must have the following type:

typedef int (*traverseproc)(PyObject *self, visitproc visit, void *arg)

Part of the Stable ABI.
Traversal function for a container object. Implementations must call the *visit* function for each object directly contained by *self*, with the parameters to *visit* being the contained object and the *arg* value passed to the handler. The *visit* function must not be called with a `NULL` object argument. If *visit* returns a non-zero value that value should be returned immediately.

The traversal function must not have any side effects. Implementations may not modify the reference counts of any Python objects nor create or destroy any Python objects.

To simplify writing `tp_traverse` handlers, a `Py_VISIT()` macro is provided. In order to use this macro, the `tp_traverse` implementation must name its arguments exactly *visit* and *arg*:

Py_VISIT(o)

If the PyObject* *o* is not `NULL`, call the *visit* callback, with arguments *o* and *arg*. If *visit* returns a non-zero value, then return it. Using this macro, `tp_traverse` handlers look like:

```
static int
my_traverse(Noddy *self, visitproc visit, void *arg)
{
    Py_VISIT(self->foo);
    Py_VISIT(self->bar);
    return 0;
}
```

The `tp_clear` handler must be of the `inquiry` type, or `NULL` if the object is immutable.

typedef int (*inquiry)(PyObject *self)

Part of the Stable ABI.
Drop references that may have created reference cycles. Immutable objects do not have to define this method since they can never directly create reference cycles. Note that the object must still be valid after calling this method (don’t just call `Py_DECREF()` on a reference). The collector will call this method if it detects that this object is involved in a reference cycle.

Controlling the Garbage Collector State

The C-API provides the following functions for controlling garbage collection runs.

Py_ssize_t PyGC_Collect(void)

Part of the Stable ABI.
Perform a full garbage collection, if the garbage collector is enabled. (Note that `gc.collect()` runs it unconditionally.)

Returns the number of collected + unreachable objects which cannot be collected. If the garbage collector is disabled or already collecting, returns `0` immediately. Errors during garbage collection are passed to `sys.unraisablehook`. This function does not raise exceptions.

int PyGC_Enable(void)

Part of the Stable ABI since version 3.10.
Enable the garbage collector: similar to `gc.enable()`. Returns the previous state, 0 for disabled and 1 for enabled.

Added in version 3.10.

int PyGC_Disable(void)

Part of the Stable ABI since version 3.10.
Disable the garbage collector: similar to `gc.disable()`. Returns the previous state, 0 for disabled and 1 for enabled.

Added in version 3.10.

int PyGC_IsEnabled(void)

Part of the Stable ABI since version 3.10.
Query the state of the garbage collector: similar to `gc.isenabled()`. Returns the current state, 0 for disabled and 1 for enabled.

Added in version 3.10.

Querying Garbage Collector State

The C-API provides the following interface for querying information about the garbage collector.

void PyUnstable_GC_VisitObjects(gcvisitobjects_t callback, void *arg)

This is Unstable API. It may change without warning in minor releases.

Run supplied *callback* on all live GC-capable objects. *arg* is passed through to all invocations of *callback*.

Warning

If new objects are (de)allocated by the callback it is undefined if they will be visited.

Garbage collection is disabled during operation. Explicitly running a collection in the callback may lead to undefined behaviour e.g. visiting the same objects multiple times or not at all.

Added in version 3.12.

typedef int (*gcvisitobjects_t)(PyObject *object, void *arg)

Type of the visitor function to be passed to **`PyUnstable_GC_VisitObjects()`**. *arg* is the same as the *arg* passed to `PyUnstable_GC_VisitObjects`. Return `1` to continue iteration, return `0` to stop iteration. Other return values are reserved for now so behavior on returning anything else is undefined.

Added in version 3.12.

API and ABI Versioning

Build-time version constants

CPython exposes its version number in the following macros. Note that these correspond to the version code is **built** with. See `Py_Version` for the version used at **run time**.

See C API Stability for a discussion of API and ABI stability across versions.

PY_MAJOR_VERSION

The `3` in `3.4.1a2`.

PY_MINOR_VERSION

The `4` in `3.4.1a2`.

PY_MICRO_VERSION

The `1` in `3.4.1a2`.

PY_RELEASE_LEVEL

The `a` in `3.4.1a2`. This can be `0xA` for alpha, `0xB` for beta, `0xC` for release candidate or `0xF` for final.

PY_RELEASE_SERIAL

The `2` in `3.4.1a2`. Zero for final releases.

PY_VERSION_HEX

The Python version number encoded in a single integer. See `Py_PACK_FULL_VERSION()` for the encoding details.

Use this for numeric comparisons, for example, `#if PY_VERSION_HEX >= ...`.

These macros are defined in Include/patchlevel.h [https://github.com/python/cpython/tree/3.14/Include/patchlevel.h].

Run-time version

const unsigned long Py_Version

Part of the Stable ABI since version 3.11.
The Python runtime version number encoded in a single constant integer. See **`Py_PACK_FULL_VERSION()`** for the encoding details. This contains the Python version used at run time.

Use this for numeric comparisons, for example, `if (Py_Version >= ...)`.

Added in version 3.11.

Bit-packing macros

uint32_t Py_PACK_FULL_VERSION(int major, int minor, int micro, int release_level, int release_serial)

Part of the Stable ABI since version 3.14.
Return the given version, encoded as a single 32-bit integer with the following structure:

Argument	**No. of bits**	**Bit mask**	**Bit shift**	**Example values**	
				`3.4.1a2`	**`3.10.0`**
major	8	`0xFF000000`	24	`0x03`	`0x03`
minor	8	`0x00FF0000`	16	`0x04`	`0x0A`
micro	8	`0x0000FF00`	8	`0x01`	`0x00`

Argument	**No. of bits**	**Bit mask**	**Bit shift**	**Example values**	
				`3.4.1a2`	**`3.10.0`**
release_level	4	`0x000000F0`	4	`0xA`	`0xF`
release_serial	4	`0x0000000F`	0	`0x2`	`0x0`

For example:

Version	**`Py_PACK_FULL_VERSION` arguments**	**Encoded version**
`3.4.1a2`	`(3, 4, 1, 0xA, 2)`	`0x030401a2`
`3.10.0`	`(3, 10, 0, 0xF, 0)`	`0x030a00f0`

Out-of range bits in the arguments are ignored. That is, the macro can be defined as:

```
#ifndef Py_PACK_FULL_VERSION
#define Py_PACK_FULL_VERSION(X, Y, Z, LEVEL, SERIAL) ( \
   (((X) & 0xff) << 24) |                              \
   (((Y) & 0xff) << 16) |                              \
   (((Z) & 0xff) << 8) |                               \
   (((LEVEL) & 0xf) << 4) |                            \
   (((SERIAL) & 0xf) << 0))
#endif
```

`Py_PACK_FULL_VERSION` is primarily a macro, intended for use in `#if` directives, but it is also available as an exported function.

Added in version 3.14.

uint32_t Py_PACK_VERSION(int major, int minor)

Part of the Stable ABI since version 3.14.
Equivalent to `Py_PACK_FULL_VERSION(major, minor, 0, 0, 0)`. The result does not correspond to any Python release, but is useful in numeric comparisons.

Added in version 3.14.

Monitoring C API

Added in version 3.13.

An extension may need to interact with the event monitoring system. Subscribing to events and registering callbacks can be done via the Python API exposed in `sys.monitoring`.

Generating Execution Events

The functions below make it possible for an extension to fire monitoring events as it emulates the execution of Python code. Each of these functions accepts a `PyMonitoringState` struct which contains concise information about the activation state of events, as well as the event arguments, which include a `PyObject*` representing the code object, the instruction offset and sometimes additional, event-specific arguments (see `sys.monitoring` for details about the signatures of the different event callbacks). The `codelike` argument should be an instance of `types.CodeType` or of a type that emulates it.

The VM disables tracing when firing an event, so there is no need for user code to do that.

Monitoring functions should not be called with an exception set, except those listed below as working with the current exception.

type PyMonitoringState

Representation of the state of an event type. It is allocated by the user while its contents are maintained by the monitoring API functions described below.

All of the functions below return 0 on success and -1 (with an exception set) on error.

See `sys.monitoring` for descriptions of the events.

int PyMonitoring_FirePyStartEvent(PyMonitoringState *state, PyObject *codelike, int32_t offset)

Fire a `PY_START` event.

int PyMonitoring_FirePyResumeEvent(PyMonitoringState *state, PyObject *codelike, int32_t offset)

Fire a `PY_RESUME` event.

int PyMonitoring_FirePyReturnEvent(PyMonitoringState *state, PyObject *codelike, int32_t offset, PyObject *retval)

Fire a `PY_RETURN` event.

int PyMonitoring_FirePyYieldEvent(PyMonitoringState *state, PyObject *codelike, int32_t offset, PyObject *retval)

Fire a `PY_YIELD` event.

int PyMonitoring_FireCallEvent(PyMonitoringState *state, PyObject *codelike, int32_t offset, PyObject *callable, PyObject *arg0)

Fire a `CALL` event.

int PyMonitoring_FireLineEvent(PyMonitoringState *state, PyObject *codelike, int32_t offset, int lineno)

Fire a `LINE` event.

int PyMonitoring_FireJumpEvent(PyMonitoringState *state, PyObject *codelike, int32_t offset, PyObject *target_offset)

Fire a `JUMP` event.

int PyMonitoring_FireBranchLeftEvent(PyMonitoringState *state, PyObject *codelike, int32_t offset, PyObject *target_offset)

Fire a `BRANCH_LEFT` event.

int PyMonitoring_FireBranchRightEvent(PyMonitoringState *state, PyObject *codelike, int32_t offset, PyObject *target_offset)

Fire a `BRANCH_RIGHT` event.

int PyMonitoring_FireCReturnEvent(PyMonitoringState *state, PyObject *codelike, int32_t offset, PyObject *retval)

Fire a `C_RETURN` event.

int PyMonitoring_FirePyThrowEvent(PyMonitoringState *state, PyObject *codelike, int32_t offset)

Fire a `PY_THROW` event with the current exception (as returned by **`PyErr_GetRaisedException()`**).

int PyMonitoring_FireRaiseEvent(PyMonitoringState *state, PyObject *codelike, int32_t offset)

Fire a `RAISE` event with the current exception (as returned by **`PyErr_GetRaisedException()`**).

int PyMonitoring_FireCRaiseEvent(PyMonitoringState *state, PyObject *codelike, int32_t offset)

Fire a `C_RAISE` event with the current exception (as returned by **`PyErr_GetRaisedException()`**).

int PyMonitoring_FireReraiseEvent(PyMonitoringState *state, PyObject *codelike, int32_t offset)

Fire a `RERAISE` event with the current exception (as returned by **`PyErr_GetRaisedException()`**).

int PyMonitoring_FireExceptionHandledEvent(PyMonitoringState *state, PyObject *codelike, int32_t offset)

Fire an `EXCEPTION_HANDLED` event with the current exception (as returned by **`PyErr_GetRaisedException()`**).

int PyMonitoring_FirePyUnwindEvent(PyMonitoringState *state, PyObject *codelike, int32_t offset)

Fire a `PY_UNWIND` event with the current exception (as returned by **`PyErr_GetRaisedException()`**).

int PyMonitoring_FireStopIterationEvent(PyMonitoringState *state, PyObject *codelike, int32_t offset, PyObject *value)

Fire a `STOP_ITERATION` event. If `value` is an instance of **`StopIteration`**, it is used. Otherwise, a new **`StopIteration`** instance is created with `value` as its argument.

Managing the Monitoring State

Monitoring states can be managed with the help of monitoring scopes. A scope would typically correspond to a Python function.

int PyMonitoring_EnterScope(PyMonitoringState *state_array, uint64_t *version, const uint8_t *event_types, Py_ssize_t length)

Enter a monitored scope. `event_types` is an array of the event IDs for events that may be fired from the scope. For example, the ID of a `PY_START` event is the value `PY_MONITORING_EVENT_PY_START`, which is numerically equal to the base-2 logarithm of `sys.monitoring.events.PY_START`. `state_array` is an array with a monitoring state entry for each event in `event_types`, it is allocated by the user but populated by **`PyMonitoring_EnterScope()`** with information about the activation state of the event. The size of `event_types` (and hence also of `state_array`) is given in `length`.

The `version` argument is a pointer to a value which should be allocated by the user together with `state_array` and initialized to 0, and then set only by **`PyMonitoring_EnterScope()`** itself. It allows this function to determine whether event states have changed since the previous call, and to return quickly if they have not.

The scopes referred to here are lexical scopes: a function, class or method. **`PyMonitoring_EnterScope()`** should be called whenever the lexical scope is entered. Scopes can be reentered, reusing the same *state_array* and *version*, in situations like when emulating a recursive Python function. When a code-like’s execution is paused, such as when emulating a generator, the scope needs to be exited and re-entered.

The macros for *event_types* are:

Macro	Event
PY_MONITORING_EVENT_BRANCH_LEFT	**`BRANCH_LEFT`**
PY_MONITORING_EVENT_BRANCH_RIGHT	**`BRANCH_RIGHT`**
PY_MONITORING_EVENT_CALL	**`CALL`**

Macro	**Event**
PY_MONITORING_EVENT_C_RAISE	`C_RAISE`
PY_MONITORING_EVENT_C_RETURN	`C_RETURN`
PY_MONITORING_EVENT_EXCEPTION_HANDLED	`EXCEPTION_HANDLED`
PY_MONITORING_EVENT_INSTRUCTION	`INSTRUCTION`
PY_MONITORING_EVENT_JUMP	`JUMP`
PY_MONITORING_EVENT_LINE	`LINE`
PY_MONITORING_EVENT_PY_RESUME	`PY_RESUME`
PY_MONITORING_EVENT_PY_RETURN	`PY_RETURN`
PY_MONITORING_EVENT_PY_START	`PY_START`
PY_MONITORING_EVENT_PY_THROW	`PY_THROW`
PY_MONITORING_EVENT_PY_UNWIND	`PY_UNWIND`
PY_MONITORING_EVENT_PY_YIELD	`PY_YIELD`
PY_MONITORING_EVENT_RAISE	`RAISE`
PY_MONITORING_EVENT_RERAISE	`RERAISE`
PY_MONITORING_EVENT_STOP_ITERATION	`STOP_ITERATION`

int PyMonitoring_ExitScope(void)

Exit the last scope that was entered with `PyMonitoring_EnterScope()`.

int PY_MONITORING_IS_INSTRUMENTED_EVENT(uint8_t ev)

Return true if the event corresponding to the event ID *ev* is a local event.

Added in version 3.13.

Deprecated since version 3.14: This function is soft deprecated.

Installing Python Modules

Email:: distutils-sig@python.org

As a popular open source development project, Python has an active supporting community of contributors and users that also make their software available for other Python developers to use under open source license terms.

This allows Python users to share and collaborate effectively, benefiting from the solutions others have already created to common (and sometimes even rare!) problems, as well as potentially contributing their own solutions to the common pool.

This guide covers the installation part of the process. For a guide to creating and sharing your own Python projects, refer to the Python packaging user guide [https://packaging.python.org/en/latest/tutorials/packaging-projects/].

Note

For corporate and other institutional users, be aware that many organisations have their own policies around using and contributing to open source software. Please take such policies into account when making use of the distribution and installation tools provided with Python.

Key terms

- `pip` is the preferred installer program. Starting with Python 3.4, it is included by default with the Python binary installers.

- A *virtual environment* is a semi-isolated Python environment that allows packages to be installed for use by a particular application, rather than being installed system wide.
- `venv` is the standard tool for creating virtual environments, and has been part of Python since Python 3.3. Starting with Python 3.4, it defaults to installing `pip` into all created virtual environments.
- `virtualenv` is a third party alternative (and predecessor) to `venv`. It allows virtual environments to be used on versions of Python prior to 3.4, which either don't provide `venv` at all, or aren't able to automatically install `pip` into created environments.
- The Python Package Index [https://pypi.org] is a public repository of open source licensed packages made available for use by other Python users.
- the Python Packaging Authority [https://www.pypa.io/] is the group of developers and documentation authors responsible for the maintenance and evolution of the standard packaging tools and the associated metadata and file format standards. They maintain a variety of tools, documentation, and issue trackers on GitHub [https://github.com/pypa].
- `distutils` is the original build and distribution system first added to the Python standard library in 1998. While direct use of `distutils` is being phased out, it still laid the foundation for the current packaging and distribution infrastructure, and it not only remains part of the standard library, but its name lives on in other ways (such as the name of the mailing list used to coordinate Python packaging standards development).

Changed in version 3.5: The use of `venv` is now recommended for creating virtual environments.

See also

Python Packaging User Guide: Creating and using virtual environments [https://packaging.python.org/installing/#creating-virtual-environments]

Basic usage

The standard packaging tools are all designed to be used from the command line.

The following command will install the latest version of a module and its dependencies from the Python Package Index:

```
python -m pip install SomePackage
```

Note

For POSIX users (including macOS and Linux users), the examples in this guide assume the use of a virtual environment.

For Windows users, the examples in this guide assume that the option to adjust the system PATH environment variable was selected when installing Python.

It’s also possible to specify an exact or minimum version directly on the command line. When using comparator operators such as `>`, `<` or some other special character which get interpreted by shell, the package name and the version should be enclosed within double quotes:

```
python -m pip install SomePackage==1.0.4    # specific version
python -m pip install "SomePackage>=1.0.4"  # minimum version
```

Normally, if a suitable module is already installed, attempting to install it again will have no effect. Upgrading existing modules must be requested explicitly:

```
python -m pip install --upgrade SomePackage
```

More information and resources regarding `pip` and its capabilities can be found in the Python Packaging User Guide [https://packaging.python.org].

Creation of virtual environments is done through the **`venv`** module. Installing packages into an active virtual environment uses the commands shown above.

See also

Python Packaging User Guide: Installing Python Distribution Packages [https://packaging.python.org/installing/]

How do I …?

These are quick answers or links for some common tasks.

… install `pip` in versions of Python prior to Python 3.4?

Python only started bundling `pip` with Python 3.4. For earlier versions, `pip` needs to be “bootstrapped” as described in the Python Packaging User Guide.

See also

Python Packaging User Guide: Requirements for Installing Packages [https://packaging.python.org/installing/#requirements-for-installing-packages]

… install packages just for the current user?

Passing the `--user` option to `python -m pip install` will install a package just for the current user, rather than for all users of the system.

… install scientific Python packages?

A number of scientific Python packages have complex binary dependencies, and aren’t currently easy to install using `pip` directly. At this point in time, it will often be easier for users to install these packages by other means [https://packaging.python.org/science/] rather than attempting to install them with `pip`.

See also

Python Packaging User Guide: Installing Scientific Packages [https://packaging.python.org/science/]

… work with multiple versions of Python installed in parallel?

On Linux, macOS, and other POSIX systems, use the versioned Python commands in combination with the `-m` switch to run the appropriate copy of `pip`:

```
python2   -m pip install SomePackage  # default Python 2
python2.7 -m pip install SomePackage  # specifically Python 2.7
```

```
python3   -m pip install SomePackage  # default Python 3
python3.4 -m pip install SomePackage  # specifically Python 3.4
```

Appropriately versioned `pip` commands may also be available.

On Windows, use the `py` Python launcher in combination with the `-m` switch:

```
py -2   -m pip install SomePackage  # default Python 2
py -2.7 -m pip install SomePackage  # specifically Python 2.7
py -3   -m pip install SomePackage  # default Python 3
py -3.4 -m pip install SomePackage  # specifically Python 3.4
```

Common installation issues

Installing into the system Python on Linux

On Linux systems, a Python installation will typically be included as part of the distribution. Installing into this Python installation requires root access to the system, and may interfere with the operation of the system package manager and other components of the system if a component is unexpectedly upgraded using `pip`.

On such systems, it is often better to use a virtual environment or a per-user installation when installing packages with `pip`.

Pip not installed

It is possible that `pip` does not get installed by default. One potential fix is:

```
python -m ensurepip --default-pip
```

There are also additional resources for installing pip. [https://packaging.python.org/en/latest/tutorials/installing-packages/#ensure-pip-setuptools-and-wheel-are-up-to-date]

Installing binary extensions

Python has typically relied heavily on source based distribution, with end users being expected to compile extension modules from source as part of the installation process.

With the introduction of support for the binary `wheel` format, and the ability to publish wheels for at least Windows and macOS through the Python Package Index, this problem is expected to diminish over time, as users are more regularly able to install pre-built extensions rather than needing to build them themselves.

Some of the solutions for installing scientific software [https://packaging.python.org/science/] that are not yet available as pre-built `wheel` files may also help with obtaining other binary extensions without needing to build them locally.

See also

Python Packaging User Guide: Binary Extensions [https://packaging.python.org/extensions/]

Python HOWTOs

Python HOWTOs are documents that cover a specific topic in-depth. Modeled on the Linux Documentation Project's HOWTO collection, this collection is an effort to foster documentation that's more detailed than the Python Library Reference.

General:

- A Conceptual Overview of asyncio
- Annotations Best Practices
- Argparse Tutorial
- Descriptor Guide
- Enum HOWTO
- Functional Programming HOWTO
- An introduction to the ipaddress module
- Logging HOWTO
- Logging Cookbook
- Regular Expression HOWTO
- Sorting Techniques
- Unicode HOWTO
- HOWTO Fetch Internet Resources Using The urllib Package

Advanced development:

- Curses Programming with Python
- Python support for free threading
- C API Extension Support for Free Threading
- Isolating Extension Modules
- The Python 2.3 Method Resolution Order
- Socket Programming HOWTO
- timer file descriptor HOWTO

- Porting Extension Modules to Python 3

Debugging and profiling:

- Debugging C API extensions and CPython Internals with GDB
- Instrumenting CPython with DTrace and SystemTap
- Python support for the Linux perf profiler
- Remote debugging attachment protocol

A Conceptual Overview of `asyncio`

This HOWTO article seeks to help you build a sturdy mental model of how **`asyncio`** fundamentally works, helping you understand the how and why behind the recommended patterns.

You might be curious about some key **`asyncio`** concepts. By the end of this article, you'll be able to comfortably answer these questions:

- What's happening behind the scenes when an object is awaited?
- How does **`asyncio`** differentiate between a task which doesn't need CPU time (such as a network request or file read) as opposed to a task that does (such as computing n-factorial)?
- How to write an asynchronous variant of an operation, such as an async sleep or database request.

See also

- The guide [https://github.com/anordin95/a-conceptual-overview-of-asyncio/tree/main] that inspired this HOWTO article, by Alexander Nordin.
- This in-depth YouTube tutorial series [https://www.youtube.com/watch?v=Xbl7XjFYsN4&list=PLhNSoGM2ik6SIkVGXWBwerucXjgP1rHmB] on `asyncio` created by Python core team member, Łukasz Langa.
- 500 Lines or Less: A Web Crawler With asyncio Coroutines [https://aosabook.org/en/500L/a-web-crawler-with-asyncio-

coroutines.html] by A. Jesse Jiryu Davis and Guido van Rossum.

A conceptual overview part 1: the high-level

In part 1, we’ll cover the main, high-level building blocks of **`asyncio`**: the event loop, coroutine functions, coroutine objects, tasks, and `await`.

Event Loop

Everything in **`asyncio`** happens relative to the event loop. It’s the star of the show. It’s like an orchestra conductor. It’s behind the scenes managing resources. Some power is explicitly granted to it, but a lot of its ability to get things done comes from the respect and cooperation of its worker bees.

In more technical terms, the event loop contains a collection of jobs to be run. Some jobs are added directly by you, and some indirectly by **`asyncio`**. The event loop takes a job from its backlog of work and invokes it (or “gives it control”), similar to calling a function, and then that job runs. Once it pauses or completes, it returns control to the event loop. The event loop will then select another job from its pool and invoke it. You can *roughly* think of the collection of jobs as a queue: jobs are added and then processed one at a time, generally (but not always) in order. This process repeats indefinitely, with the event loop cycling endlessly onwards. If there are no more jobs pending execution, the event loop is smart enough to rest and avoid needlessly wasting CPU cycles, and will come back when there’s more work to be done.

Effective execution relies on jobs sharing well and cooperating; a greedy job could hog control and leave the other jobs to starve, rendering the overall event loop approach rather useless.

```
import asyncio

# This creates an event loop and indefinitely cycles through
# its collection of jobs.
event_loop = asyncio.new_event_loop()
event_loop.run_forever()
```

Asynchronous functions and coroutines

This is a basic, boring Python function:

```
def hello_printer():
    print(
        "Hi, I am a lowly, simple printer, though I have all I
"
        "need in life -- \nfresh paper and my dearly beloved
octopus "
        "partner in crime."
    )
```

Calling a regular function invokes its logic or body:

```
>>> hello_printer()
Hi, I am a lowly, simple printer, though I have all I need in
life --
fresh paper and my dearly beloved octopus partner in crime.
```

The async def, as opposed to just a plain `def`, makes this an asynchronous function (or “coroutine function”). Calling it creates and returns a coroutine object.

```
async def loudmouth_penguin(magic_number: int):
    print(
     "I am a super special talking penguin. Far cooler than
that printer. "
     f"By the way, my lucky number is: {magic_number}."
    )
```

Calling the async function, `loudmouth_penguin`, does not execute the print statement; instead, it creates a coroutine object:

```
>>> loudmouth_penguin(magic_number=3)
<coroutine object loudmouth_penguin at 0x104ed2740>
```

The terms “coroutine function” and “coroutine object” are often conflated as coroutine. That can be confusing! In this article, coroutine specifically refers to a coroutine object, or more precisely, an instance of **`types.CoroutineType`** (native coroutine). Note that coroutines can also exist as instances of **`collections.abc.Coroutine`** – a distinction that matters for type checking.

A coroutine represents the function’s body or logic. A coroutine has to be explicitly started; again, merely creating the coroutine does not start it. Notably, the coroutine can be paused and resumed at various points within the function’s body. That pausing and resuming ability is what allows for asynchronous behavior!

Coroutines and coroutine functions were built by leveraging the functionality of generators and generator functions. Recall, a generator function is a function that **`yield`**s, like this one:

```
def get_random_number():
    # This would be a bad random number generator!
    print("Hi")
    yield 1
    print("Hello")
    yield 7
    print("Howdy")
    yield 4
    ...
```

Similar to a coroutine function, calling a generator function does not run it. Instead, it creates a generator object:

```
>>> get_random_number()
<generator object get_random_number at 0x1048671c0>
```

You can proceed to the next `yield` of a generator by using the built-in function `next()`. In other words, the generator runs, then pauses. For example:

```
>>> generator = get_random_number()
>>> next(generator)
Hi
1
>>> next(generator)
Hello
7
```

Tasks

Roughly speaking, tasks are coroutines (not coroutine functions) tied to an event loop. A task also maintains a list of callback functions whose importance will become clear in a moment when we discuss `await`. The recommended way to create tasks is via `asyncio.create_task()`.

Creating a task automatically schedules it for execution (by adding a callback to run it in the event loop's to-do list, that is, collection of jobs).

`asyncio` automatically associates tasks with the event loop for you. This automatic association was purposely designed into **`asyncio`** for the sake of simplicity. Without it, you'd have to keep track of the event loop object and pass it to any coroutine function that wants to create tasks, adding redundant clutter to your code.

```
coroutine = loudmouth_penguin(magic_number=5)
# This creates a Task object and schedules its execution via
the event loop.
task = asyncio.create_task(coroutine)
```

Earlier, we manually created the event loop and set it to run forever. In practice, it’s recommended to use (and common to see) [`asyncio.run()`](), which takes care of managing the event loop and ensuring the provided coroutine finishes before advancing. For example, many async programs follow this setup:

```
import asyncio

async def main():
    # Perform all sorts of wacky, wild asynchronous things...
    ...

if __name__ == "__main__":
    asyncio.run(main())
    # The program will not reach the following print statement
until the
    # coroutine main() finishes.
    print("coroutine main() is done!")
```

It’s important to be aware that the task itself is not added to the event loop, only a callback to the task is. This matters if the task object you created is garbage collected before it’s called by the event loop. For example, consider this program:

```
async def hello():
    print("hello!")

async def main():
    asyncio.create_task(hello())
    # Other asynchronous instructions which run for a while
    # and cede control to the event loop...
    ...

asyncio.run(main())
```

Because there’s no reference to the task object created on line 5, it *might* be garbage collected before the event loop invokes it. Later instructions in the coroutine `main()` hand control back to the event loop so it can invoke other jobs. When the event loop eventually tries to run the task, it might fail and discover the task object does not exist! This can also happen even if a coroutine keeps a reference to a task but completes before that task finishes. When the coroutine exits, local variables go out of scope and may be subject to garbage collection. In practice, `asyncio` and Python’s garbage collector work pretty hard to ensure this sort of thing doesn’t happen. But that’s no reason to be reckless!

await

`await` is a Python keyword that’s commonly used in one of two different ways:

```
await task
await coroutine
```

In a crucial way, the behavior of `await` depends on the type of object being awaited.

Awaiting a task will cede control from the current task or coroutine to the event loop. In the process of relinquishing control, a few important things happen. We’ll use the following code example to illustrate:

```
async def plant_a_tree():
    dig_the_hole_task = asyncio.create_task(dig_the_hole())
    await dig_the_hole_task

    # Other instructions associated with planting a tree.
    ...
```

In this example, imagine the event loop has passed control to the start of the coroutine `plant_a_tree()`. As seen above, the coroutine creates a task and then awaits it. The `await dig_the_hole_task` instruction adds a callback (which will resume `plant_a_tree()`) to the `dig_the_hole_task` object's list of callbacks. And then, the instruction cedes control to the event loop. Some time later, the event loop will pass control to `dig_the_hole_task` and the task will finish whatever it needs to do. Once the task finishes, it will add its various callbacks to the event loop, in this case, a call to resume `plant_a_tree()`.

Generally speaking, when the awaited task finishes (`dig_the_hole_task`), the original task or coroutine (`plant_a_tree()`) is added back to the event loop's to-do list to be resumed.

This is a basic, yet reliable mental model. In practice, the control handoffs are slightly more complex, but not by much. In part 2, we'll walk through the details that make this possible.

Unlike tasks, awaiting a coroutine does not hand control back to the event loop! Wrapping a coroutine in a task first, then awaiting that would cede control. The behavior of `await coroutine` is effectively the same as invoking a regular, synchronous Python function. Consider this program:

```
import asyncio

async def coro_a():
   print("I am coro_a(). Hi!")

async def coro_b():
   print("I am coro_b(). I sure hope no one hogs the event
loop...")
```

```
async def main():
   task_b = asyncio.create_task(coro_b())
   num_repeats = 3
   for _ in range(num_repeats):
      await coro_a()
   await task_b

asyncio.run(main())
```

The first statement in the coroutine `main()` creates `task_b` and schedules it for execution via the event loop. Then, `coro_a()` is repeatedly awaited. Control never cedes to the event loop, which is why we see the output of all three `coro_a()` invocations before `coro_b()`'s output:

```
I am coro_a(). Hi!
I am coro_a(). Hi!
I am coro_a(). Hi!
I am coro_b(). I sure hope no one hogs the event loop...
```

If we change `await coro_a()` to `await asyncio.create_task(coro_a())`, the behavior changes. The coroutine `main()` cedes control to the event loop with that statement. The event loop then proceeds through its backlog of work, calling `task_b` and then the task which wraps `coro_a()` before resuming the coroutine `main()`.

```
I am coro_b(). I sure hope no one hogs the event loop...
I am coro_a(). Hi!
I am coro_a(). Hi!
I am coro_a(). Hi!
```

This behavior of `await coroutine` can trip a lot of people up! That example highlights how using only `await coroutine` could unintentionally hog control from other tasks and effectively stall the event loop. **`asyncio.run()`** can help you detect such occurrences via the `debug=True` flag, which enables debug mode. Among other things, it will log any coroutines that monopolize execution for 100ms or longer.

The design intentionally trades off some conceptual clarity around usage of `await` for improved performance. Each time a task is awaited, control needs to be passed all the way up the call stack to the event loop. That might sound minor, but in a large program with many `await` statements and a deep call stack, that overhead can add up to a meaningful performance drag.

A conceptual overview part 2: the nuts and bolts

Part 2 goes into detail on the mechanisms **`asyncio`** uses to manage control flow. This is where the magic happens. You’ll come away from this section knowing what `await` does behind the scenes and how to make your own asynchronous operators.

The inner workings of coroutines

`asyncio` leverages four components to pass around control.

`coroutine.send(arg)` is the method used to start or resume a coroutine. If the coroutine was paused and is now being resumed, the argument `arg` will be sent in as the return value of the `yield` statement which originally paused it. If the coroutine is being used for the first time (as opposed to being resumed), `arg` must be `None`.

```
class Rock:
    def __await__(self):
        value_sent_in = yield 7
        print(f"Rock.__await__ resuming with value: {value_sent_in}.")
        return value_sent_in

async def main():
    print("Beginning coroutine main().")
    rock = Rock()
    print("Awaiting rock...")
    value_from_rock = await rock
    print(f"Coroutine received value: {value_from_rock} from rock.")
    return 23

coroutine = main()
```

```
16 intermediate_result = coroutine.send(None)
17 print(f"Coroutine paused and returned intermediate value: {intermediate_result}.")
18
19 print(f"Resuming coroutine and sending in value: 42.")
20 try:
21     coroutine.send(42)
22 except StopIteration as e:
23     returned_value = e.value
24 print(f"Coroutine main() finished and provided value: {returned_value}.")
```

yield, as usual, pauses execution and returns control to the caller. In the example above, the `yield`, on line 3, is called by `... = await rock` on line 11. More broadly speaking, `await` calls the **`__await__()`** method of the given object. `await` also does one more very special thing: it propagates (or “passes along”) any `yield`s it receives up the call chain. In this case, that’s back to `... = coroutine.send(None)` on line 16.

The coroutine is resumed via the `coroutine.send(42)` call on line 21. The coroutine picks back up from where it `yield`ed (or paused) on line 3 and executes the remaining statements in its body. When a coroutine finishes, it raises a **`StopIteration`** exception with the return value attached in the **`value`** attribute.

That snippet produces this output:

```
Beginning coroutine main().
Awaiting rock...
Coroutine paused and returned intermediate value: 7.
Resuming coroutine and sending in value: 42.
Rock.__await__ resuming with value: 42.
Coroutine received value: 42 from rock.
Coroutine main() finished and provided value: 23.
```

It’s worth pausing for a moment here and making sure you followed the various ways that control flow and values were passed. A lot of important ideas were covered and it's worth ensuring your understanding is firm.

The only way to yield (or effectively cede control) from a coroutine is to `await` an object that `yield`s in its `__await__` method. That might sound odd to you. You might be thinking:

1. What about a `yield` directly within the coroutine function? The coroutine function becomes an async generator function, a different beast entirely.

2. What about a yield from within the coroutine function to a (plain) generator? That causes the error: `SyntaxError: yield from not allowed in a coroutine`. This was intentionally designed for the sake of simplicity – mandating only one way of using coroutines. Initially `yield` was barred as well, but was re-accepted to allow for async generators. Despite that, `yield from` and `await` effectively do the same thing.

Futures

A future is an object meant to represent a computation's status and result. The term is a nod to the idea of something still to come or not yet happened, and the object is a way to keep an eye on that something.

A future has a few important attributes. One is its state, which can be either "pending", "cancelled", or "done". Another is its result, which is set when the state transitions to done. Unlike a coroutine, a future does not represent the actual computation to be done; instead, it represents the status and result of that computation, kind of like a status light (red, yellow, or green) or indicator.

`asyncio.Task` subclasses `asyncio.Future` in order to gain these various capabilities. The prior section said tasks store a list of callbacks, which wasn’t entirely accurate. It’s actually the `Future` class that implements this logic, which `Task` inherits.

Futures may also be used directly (not via tasks). Tasks mark themselves as done when their coroutine is complete. Futures are much more versatile and will be marked as done when you say so. In this way, they’re the flexible interface for you to make your own conditions for waiting and resuming.

A homemade asyncio.sleep

We’ll go through an example of how you could leverage a future to create your own variant of asynchronous sleep (`async_sleep`) which mimics `asyncio.sleep()`.

This snippet registers a few tasks with the event loop and then awaits the task created by `asyncio.create_task`, which wraps the `async_sleep(3)` coroutine. We want that task to finish only after three seconds have elapsed, but without preventing other tasks from running.

```
async def other_work():
    print("I like work. Work work.")

async def main():
    # Add a few other tasks to the event loop, so there's
something
    # to do while asynchronously sleeping.
    work_tasks = [
        asyncio.create_task(other_work()),
        asyncio.create_task(other_work()),
        asyncio.create_task(other_work())
    ]
    print(
        "Beginning asynchronous sleep at time: "
        f"{datetime.datetime.now().strftime("%H:%M:%S")}."
    )
    await asyncio.create_task(async_sleep(3))
    print(
        "Done asynchronous sleep at time: "
        f"{datetime.datetime.now().strftime("%H:%M:%S")}."
    )
    # asyncio.gather effectively awaits each task in the
collection.
    await asyncio.gather(*work_tasks)
```

Below, we use a future to enable custom control over when that task will be marked as done. If `future.set_result()` (the method responsible for marking that future as done) is never called, then this task will never finish. We've also enlisted the help of another task, which we'll see in a moment, that will monitor how much time has elapsed and, accordingly, call `future.set_result()`.

```
async def async_sleep(seconds: float):
    future = asyncio.Future()
    time_to_wake = time.time() + seconds
    # Add the watcher-task to the event loop.
    watcher_task = asyncio.create_task(_sleep_watcher(future,
time_to_wake))
    # Block until the future is marked as done.
    await future
```

Below, we use a rather bare `YieldToEventLoop()` object to `yield` from its `__await__` method, ceding control to the event loop. This is effectively the same as calling `asyncio.sleep(0)`, but this approach offers more clarity, not to mention it's somewhat cheating to use `asyncio.sleep` when showcasing how to implement it!

As usual, the event loop cycles through its tasks, giving them control and receiving control back when they pause or finish. The `watcher_task`, which runs the coroutine `_sleep_watcher(...)`, will be invoked once per full cycle of the event loop. On each resumption, it'll check the time and if not enough has elapsed, then it'll pause once again and hand control back to the event loop. Once enough time has elapsed, `_sleep_watcher(...)` marks the future as done and completes by exiting its infinite `while` loop. Given this helper task is only invoked once per cycle of the event loop, you'd be correct to note that this asynchronous sleep will sleep *at least* three seconds, rather than exactly three seconds. Note this is also true of `asyncio.sleep`.

```
class YieldToEventLoop:
    def __await__(self):
        yield

async def _sleep_watcher(future, time_to_wake):
    while True:
        if time.time() >= time_to_wake:
            # This marks the future as done.
            future.set_result(None)
            break
        else:
            await YieldToEventLoop()
```

Here is the full program's output:

```
$ python custom-async-sleep.py
Beginning asynchronous sleep at time: 14:52:22.
I like work. Work work.
I like work. Work work.
I like work. Work work.
Done asynchronous sleep at time: 14:52:25.
```

You might feel this implementation of asynchronous sleep was unnecessarily convoluted. And, well, it was. The example was meant to showcase the versatility of futures with a simple example that could be mimicked for more complex needs. For reference, you could implement it without futures, like so:

```
async def simpler_async_sleep(seconds):
    time_to_wake = time.time() + seconds
    while True:
        if time.time() >= time_to_wake:
            return
        else:
            await YieldToEventLoop()
```

But that’s all for now. Hopefully you’re ready to more confidently dive into some async programming or check out advanced topics in the **`rest of the documentation`**.

Porting Extension Modules to Python 3

We recommend the following resources for porting extension modules to Python 3:

- The Migrating C extensions [http://python3porting.com/cextensions.html] chapter from *Supporting Python 3: An in-depth guide*, a book on moving from Python 2 to Python 3 in general, guides the reader through porting an extension module.
- The Porting guide [https://py3c.readthedocs.io/en/latest/guide.html] from the *py3c* project provides opinionated suggestions with supporting code.
- Recommended third party tools offer abstractions over the Python's C API. Extensions generally need to be re-written to use one of them, but the library then handles differences between various Python versions and implementations.

Curses Programming with Python

Author:: A.M. Kuchling, Eric S. Raymond

Release:: 2.04

Abstract

This document describes how to use the `curses` extension module to control text-mode displays.

What is curses?

The curses library supplies a terminal-independent screen-painting and keyboard-handling facility for text-based terminals; such terminals include VT100s, the Linux console, and the simulated terminal provided by various programs. Display terminals support various control codes to perform common operations such as moving the cursor, scrolling the screen, and erasing areas. Different terminals use widely differing codes, and often have their own minor quirks.

In a world of graphical displays, one might ask “why bother”? It’s true that character-cell display terminals are an obsolete technology, but there are niches in which being able to do fancy things with them are still valuable. One niche is on small-footprint or embedded Unixes that don’t run an X server. Another is tools such as OS installers and kernel configurators that may have to run before any graphical support is available.

The curses library provides fairly basic functionality, providing the programmer with an abstraction of a display containing multiple non-overlapping windows of text. The contents of a window can be changed in various ways—adding text, erasing it, changing its appearance—and the curses library will figure out what control codes need to be sent to the terminal to produce the right output. curses doesn't provide many user-interface concepts such as buttons, checkboxes, or dialogs; if you need such features, consider a user interface library such as Urwid [https://pypi.org/project/Urwid/].

The curses library was originally written for BSD Unix; the later System V versions of Unix from AT&T added many enhancements and new functions. BSD curses is no longer maintained, having been replaced by ncurses, which is an open-source implementation of the AT&T interface. If you're using an open-source Unix such as Linux or FreeBSD, your system almost certainly uses ncurses. Since most current commercial Unix versions are based on System V code, all the functions described here will probably be available. The older versions of curses carried by some proprietary Unixes may not support everything, though.

The Windows version of Python doesn't include the **`curses`** module. A ported version called UniCurses [https://pypi.org/project/UniCurses/] is available.

The Python curses module

The Python module is a fairly simple wrapper over the C functions provided by curses; if you're already familiar with curses programming in C, it's really easy to transfer that knowledge to Python. The biggest difference is that the Python interface makes things simpler by merging different C functions such as **`addstr()`**, **`mvaddstr()`**, and **`mvwaddstr()`** into a single **`addstr()`** method. You’ll see this covered in more detail later.

This HOWTO is an introduction to writing text-mode programs with curses and Python. It doesn’t attempt to be a complete guide to the curses API; for that, see the Python library guide’s section on ncurses, and the C manual pages for ncurses. It will, however, give you the basic ideas.

Starting and ending a curses application

Before doing anything, curses must be initialized. This is done by calling the **`initscr()`** function, which will determine the terminal type, send any required setup codes to the terminal, and create various internal data structures. If successful, **`initscr()`** returns a window object representing the entire screen; this is usually called `stdscr` after the name of the corresponding C variable.

```
import curses
stdscr = curses.initscr()
```

Usually curses applications turn off automatic echoing of keys to the screen, in order to be able to read keys and only display them under certain circumstances. This requires calling the **`noecho()`** function.

```
curses.noecho()
```

Applications will also commonly need to react to keys instantly, without requiring the Enter key to be pressed; this is called cbreak mode, as opposed to the usual buffered input mode.

```
curses.cbreak()
```

Terminals usually return special keys, such as the cursor keys or navigation keys such as Page Up and Home, as a multibyte escape sequence. While you could write your application to expect such sequences and process them accordingly, curses can do it for you, returning a special value such as `curses.KEY_LEFT`. To get curses to do the job, you'll have to enable keypad mode.

```
stdscr.keypad(True)
```

Terminating a curses application is much easier than starting one. You'll need to call:

```
curses.nocbreak()
stdscr.keypad(False)
curses.echo()
```

to reverse the curses-friendly terminal settings. Then call the `endwin()` function to restore the terminal to its original operating mode.

```
curses.endwin()
```

A common problem when debugging a curses application is to get your terminal messed up when the application dies without restoring the terminal to its previous state. In Python this commonly happens when your code is buggy and raises an uncaught exception. Keys are no longer echoed to the screen when you type them, for example, which makes using the shell difficult.

In Python you can avoid these complications and make debugging much easier by importing the `curses.wrapper()` function and using it like this:

```
from curses import wrapper

def main(stdscr):
    # Clear screen
    stdscr.clear()

    # This raises ZeroDivisionError when i == 10.
    for i in range(0, 11):
        v = i-10
        stdscr.addstr(i, 0, '10 divided by {} is {}'.format(v,
10/v))

        stdscr.refresh()
        stdscr.getkey()

wrapper(main)
```

The `wrapper()` function takes a callable object and does the initializations described above, also initializing colors if color support is present. **`wrapper()`** then runs your provided callable. Once the callable returns, **`wrapper()`** will restore the original state of the terminal. The callable is called inside a `try`…`except` that catches exceptions, restores the state of the terminal, and then re-raises the exception. Therefore your terminal won’t be left in a funny state on exception and you’ll be able to read the exception’s message and traceback.

Windows and Pads

Windows are the basic abstraction in curses. A window object represents a rectangular area of the screen, and supports methods to display text, erase it, allow the user to input strings, and so forth.

The `stdscr` object returned by the `initscr()` function is a window object that covers the entire screen. Many programs may need only this single window, but you might wish to divide the screen into smaller windows, in order to redraw or clear them separately. The `newwin()` function creates a new window of a given size, returning the new window object.

```
begin_x = 20; begin_y = 7
height = 5; width = 40
win = curses.newwin(height, width, begin_y, begin_x)
```

Note that the coordinate system used in curses is unusual. Coordinates are always passed in the order *y,x*, and the top-left corner of a window is coordinate (0,0). This breaks the normal convention for handling coordinates where the *x* coordinate comes first. This is an unfortunate difference from most other computer applications, but it's been part of curses since it was first written, and it's too late to change things now.

Your application can determine the size of the screen by using the `curses.LINES` and `curses.COLS` variables to obtain the *y* and *x* sizes. Legal coordinates will then extend from `(0,0)` to `(curses.LINES - 1, curses.COLS - 1)`.

When you call a method to display or erase text, the effect doesn't immediately show up on the display. Instead you must call the `refresh()` method of window objects to update the screen.

This is because curses was originally written with slow 300-baud terminal connections in mind; with these terminals, minimizing the time required to redraw the screen was very important. Instead curses accumulates changes to the screen and displays them in the most efficient manner when you call **`refresh()`**. For example, if your program displays some text in a window and then clears the window, there's no need to send the original text because they're never visible.

In practice, explicitly telling curses to redraw a window doesn't really complicate programming with curses much. Most programs go into a flurry of activity, and then pause waiting for a keypress or some other action on the part of the user. All you have to do is to be sure that the screen has been redrawn before pausing to wait for user input, by first calling **`stdscr.refresh()`** or the **`refresh()`** method of some other relevant window.

A pad is a special case of a window; it can be larger than the actual display screen, and only a portion of the pad displayed at a time. Creating a pad requires the pad’s height and width, while refreshing a pad requires giving the coordinates of the on-screen area where a subsection of the pad will be displayed.

```
pad = curses.newpad(100, 100)
# These loops fill the pad with letters; addch() is
# explained in the next section
for y in range(0, 99):
    for x in range(0, 99):
        pad.addch(y,x, ord('a') + (x*x+y*y) % 26)

# Displays a section of the pad in the middle of the screen.
# (0,0) : coordinate of upper-left corner of pad area to
display.
# (5,5) : coordinate of upper-left corner of window area to be
filled
#         with pad content.
# (20, 75) : coordinate of lower-right corner of window area to
be
#          : filled with pad content.
pad.refresh( 0,0, 5,5, 20,75)
```

The **`refresh()`** call displays a section of the pad in the rectangle extending from coordinate (5,5) to coordinate (20,75) on the screen; the upper left corner of the displayed section is coordinate (0,0) on the pad. Beyond that difference, pads are exactly like ordinary windows and support the same methods.

If you have multiple windows and pads on screen there is a more efficient way to update the screen and prevent annoying screen flicker as each part of the screen gets updated. **`refresh()`** actually does two things:

1. Calls the `noutrefresh()` method of each window to update an underlying data structure representing the desired state of the screen.
2. Calls the function `doupdate()` function to change the physical screen to match the desired state recorded in the data structure.

Instead you can call **`noutrefresh()`** on a number of windows to update the data structure, and then call **`doupdate()`** to update the screen.

Displaying Text

From a C programmer's point of view, curses may sometimes look like a twisty maze of functions, all subtly different. For example, **`addstr()`** displays a string at the current cursor location in the `stdscr` window, while **`mvaddstr()`** moves to a given y,x coordinate first before displaying the string. **`waddstr()`** is just like **`addstr()`**, but allows specifying a window to use instead of using `stdscr` by default. **`mvwaddstr()`** allows specifying both a window and a coordinate.

Fortunately the Python interface hides all these details. `stdscr` is a window object like any other, and methods such as `addstr()` accept multiple argument forms. Usually there are four different forms.

Form	Description
str or *ch*	Display the string *str* or character *ch* at the current position

Form	Description
str or *ch*, *attr*	Display the string *str* or character *ch*, using attribute *attr* at the current position
y, *x*, *str* or *ch*	Move to position *y,x* within the window, and display *str* or *ch*
y, *x*, *str* or *ch*, *attr*	Move to position *y,x* within the window, and display *str* or *ch*, using attribute *attr*

Attributes allow displaying text in highlighted forms such as boldface, underline, reverse code, or in color. They’ll be explained in more detail in the next subsection.

The `addstr()` method takes a Python string or bytestring as the value to be displayed. The contents of bytestrings are sent to the terminal as-is. Strings are encoded to bytes using the value of the window’s `encoding` attribute; this defaults to the default system encoding as returned by `locale.getencoding()`.

The `addch()` methods take a character, which can be either a string of length 1, a bytestring of length 1, or an integer.

Constants are provided for extension characters; these constants are integers greater than 255. For example, `ACS_PLMINUS` is a +/- symbol, and `ACS_ULCORNER` is the upper left corner of a box (handy for drawing borders). You can also use the appropriate Unicode character.

Windows remember where the cursor was left after the last operation, so if you leave out the *y,x* coordinates, the string or character will be displayed wherever the last operation left off. You can also move the cursor with the `move(y,x)` method. Because some terminals always display a flashing cursor, you may want to ensure that the cursor is positioned in some location where it won’t be distracting; it can be confusing to have the cursor blinking at some apparently random location.

If your application doesn’t need a blinking cursor at all, you can call `curs_set(False)` to make it invisible. For compatibility with older curses versions, there’s a `leaveok(bool)` function that’s a synonym for **`curs_set()`**. When *bool* is true, the curses library will attempt to suppress the flashing cursor, and you won’t need to worry about leaving it in odd locations.

Attributes and Color

Characters can be displayed in different ways. Status lines in a text-based application are commonly shown in reverse video, or a text viewer may need to highlight certain words. curses supports this by allowing you to specify an attribute for each cell on the screen.

An attribute is an integer, each bit representing a different attribute. You can try to display text with multiple attribute bits set, but curses doesn’t guarantee that all the possible combinations are available, or that they’re all visually distinct. That depends on the ability of the terminal being used, so it’s safest to stick to the most commonly available attributes, listed here.

Attribute	Description
`A_BLINK`	Blinking text
`A_BOLD`	Extra bright or bold text
`A_DIM`	Half bright text
`A_REVERSE`	Reverse-video text
`A_STANDOUT`	The best highlighting mode available
`A_UNDERLINE`	Underlined text

So, to display a reverse-video status line on the top line of the screen, you could code:

```
stdscr.addstr(0, 0, "Current mode: Typing mode",
              curses.A_REVERSE)
stdscr.refresh()
```

The curses library also supports color on those terminals that provide it. The most common such terminal is probably the Linux console, followed by color xterms.

To use color, you must call the `start_color()` function soon after calling `initscr()`, to initialize the default color set (the `curses.wrapper()` function does this automatically). Once that’s done, the `has_colors()` function returns TRUE if the terminal in use can actually display color. (Note: curses uses the American spelling ‘color’, instead of the Canadian/British spelling ‘colour’. If you’re used to the British spelling, you’ll have to resign yourself to misspelling it for the sake of these functions.)

The curses library maintains a finite number of color pairs, containing a foreground (or text) color and a background color. You can get the attribute value corresponding to a color pair with the `color_pair()` function; this can be bitwise-OR’ed with other attributes such as `A_REVERSE`, but again, such combinations are not guaranteed to work on all terminals.

An example, which displays a line of text using color pair 1:

```
stdscr.addstr("Pretty text", curses.color_pair(1))
stdscr.refresh()
```

As I said before, a color pair consists of a foreground and background color. The `init_pair(n, f, b)` function changes the definition of color pair *n*, to foreground color f and background color b. Color pair 0 is hard-wired to white on black, and cannot be changed.

Colors are numbered, and `start_color()` initializes 8 basic colors when it activates color mode. They are: 0:black, 1:red, 2:green, 3:yellow, 4:blue, 5:magenta, 6:cyan, and 7:white. The `curses` module defines named constants for each of these colors: `curses.COLOR_BLACK`, `curses.COLOR_RED`, and so forth.

Let’s put all this together. To change color 1 to red text on a white background, you would call:

```
curses.init_pair(1, curses.COLOR_RED, curses.COLOR_WHITE)
```

When you change a color pair, any text already displayed using that color pair will change to the new colors. You can also display new text in this color with:

```
stdscr.addstr(0,0, "RED ALERT!", curses.color_pair(1))
```

Very fancy terminals can change the definitions of the actual colors to a given RGB value. This lets you change color 1, which is usually red, to purple or blue or any other color you like. Unfortunately, the Linux console doesn't support this, so I'm unable to try it out, and can't provide any examples. You can check if your terminal can do this by calling **`can_change_color()`**, which returns `True` if the capability is there. If you're lucky enough to have such a talented terminal, consult your system's man pages for more information.

User Input

The C curses library offers only very simple input mechanisms. Python's **`curses`** module adds a basic text-input widget. (Other libraries such as Urwid [https://pypi.org/project/Urwid/] have more extensive collections of widgets.)

There are two methods for getting input from a window:

- **`getch()`** refreshes the screen and then waits for the user to hit a key, displaying the key if **`echo()`** has been called earlier. You can optionally specify a coordinate to which the cursor should be moved before pausing.
- **`getkey()`** does the same thing but converts the integer to a string. Individual characters are returned as 1-character strings, and special keys such as function keys return longer strings containing a key name such as `KEY_UP` or `^G`.

It’s possible to not wait for the user using the `nodelay()` window method. After `nodelay(True)`, **`getch()`** and **`getkey()`** for the window become non-blocking. To signal that no input is ready, **`getch()`** returns `curses.ERR` (a value of -1) and **`getkey()`** raises an exception. There’s also a `halfdelay()` function, which can be used to (in effect) set a timer on each **`getch()`**; if no input becomes available within a specified delay (measured in tenths of a second), curses raises an exception.

The **`getch()`** method returns an integer; if it’s between 0 and 255, it represents the ASCII code of the key pressed. Values greater than 255 are special keys such as Page Up, Home, or the cursor keys. You can compare the value returned to constants such as `curses.KEY_PPAGE`, `curses.KEY_HOME`, or `curses.KEY_LEFT`. The main loop of your program may look something like this:

```
while True:
    c = stdscr.getch()
    if c == ord('p'):
        PrintDocument()
    elif c == ord('q'):
        break  # Exit the while loop
    elif c == curses.KEY_HOME:
        x = y = 0
```

The `curses.ascii` module supplies ASCII class membership functions that take either integer or 1-character string arguments; these may be useful in writing more readable tests for such loops. It also supplies conversion functions that take either integer or 1-character-string arguments and return the same type. For example, `curses.ascii.ctrl()` returns the control character corresponding to its argument.

There’s also a method to retrieve an entire string, `getstr()`. It isn’t used very often, because its functionality is quite limited; the only editing keys available are the backspace key and the Enter key, which terminates the string. It can optionally be limited to a fixed number of characters.

```
curses.echo()            # Enable echoing of characters

# Get a 15-character string, with the cursor on the top line
s = stdscr.getstr(0,0, 15)
```

The `curses.textpad` module supplies a text box that supports an Emacs-like set of keybindings. Various methods of the `Textbox` class support editing with input validation and gathering the edit results either with or without trailing spaces. Here’s an example:

```
import curses
from curses.textpad import Textbox, rectangle

def main(stdscr):
    stdscr.addstr(0, 0, "Enter IM message: (hit Ctrl-G to
send)")

    editwin = curses.newwin(5,30, 2,1)
    rectangle(stdscr, 1,0, 1+5+1, 1+30+1)
    stdscr.refresh()

    box = Textbox(editwin)

    # Let the user edit until Ctrl-G is struck.
    box.edit()

    # Get resulting contents
    message = box.gather()
```

See the library documentation on `curses.textpad` for more details.

For More Information

This HOWTO doesn’t cover some advanced topics, such as reading the contents of the screen or capturing mouse events from an xterm instance, but the Python library page for the `curses` module is now reasonably complete. You should browse it next.

If you’re in doubt about the detailed behavior of the curses functions, consult the manual pages for your curses implementation, whether it’s ncurses or a proprietary Unix vendor’s. The manual pages will document any quirks, and provide complete lists of all the functions, attributes, and ACS_* characters available to you.

Because the curses API is so large, some functions aren’t supported in the Python interface. Often this isn’t because they’re difficult to implement, but because no one has needed them yet. Also, Python doesn’t yet support the menu library associated with ncurses. Patches adding support for these would be welcome; see the Python Developer’s Guide [https://devguide.python.org/] to learn more about submitting patches to Python.

- Writing Programs with NCURSES [https://invisible-island.net/ncurses/ncurses-intro.html]: a lengthy tutorial for C programmers.
- The ncurses man page [https://linux.die.net/man/3/ncurses]
- The ncurses FAQ [https://invisible-island.net/ncurses/ncurses.faq.html]
- “Use curses… don’t swear” [https://www.youtube.com/watch?v=eN1eZtjLEnU]: video of a PyCon 2013 talk on controlling terminals using curses or Urwid.
- “Console Applications with Urwid” [https://pyvideo.org/video/1568/console-applications-with-urwid]: video of a PyCon CA 2012 talk demonstrating some applications written using Urwid.

Descriptor Guide

Author:: Raymond Hettinger
Contact:: <python at rcn dot com>

Contents

- Descriptor Guide
    - Primer
        - Simple example: A descriptor that returns a constant
        - Dynamic lookups
        - Managed attributes
        - Customized names
        - Closing thoughts
    - Complete Practical Example
        - Validator class
        - Custom validators
        - Practical application
    - Technical Tutorial
        - Abstract
        - Definition and introduction
        - Descriptor protocol
        - Overview of descriptor invocation
        - Invocation from an instance
        - Invocation from a class
        - Invocation from super
        - Summary of invocation logic
        - Automatic name notification
        - ORM example
    - Pure Python Equivalents
        - Properties
        - Functions and methods

    - Kinds of methods
    - Static methods
    - Class methods
    - Member objects and __slots__

Descriptors let objects customize attribute lookup, storage, and deletion.

This guide has four major sections:

1. The “primer” gives a basic overview, moving gently from simple examples, adding one feature at a time. Start here if you’re new to descriptors.
2. The second section shows a complete, practical descriptor example. If you already know the basics, start there.
3. The third section provides a more technical tutorial that goes into the detailed mechanics of how descriptors work. Most people don’t need this level of detail.
4. The last section has pure Python equivalents for built-in descriptors that are written in C. Read this if you’re curious about how functions turn into bound methods or about the implementation of common tools like `classmethod()`, `staticmethod()`, `property()`, and __slots__.

Primer

In this primer, we start with the most basic possible example and then we’ll add new capabilities one by one.

Simple example: A descriptor that returns a constant

The **`Ten`** class is a descriptor whose `__get__()` method always returns the constant `10`:

```
class Ten:
    def __get__(self, obj, objtype=None):
        return 10
```

To use the descriptor, it must be stored as a class variable in another class:

```
class A:
    x = 5                       # Regular class attribute
    y = Ten()                   # Descriptor instance
```

An interactive session shows the difference between normal attribute lookup and descriptor lookup:

```
>>> a = A()                     # Make an instance of class A
>>> a.x                         # Normal attribute lookup
5
>>> a.y                         # Descriptor lookup
10
```

In the `a.x` attribute lookup, the dot operator finds `'x': 5` in the class dictionary. In the `a.y` lookup, the dot operator finds a descriptor instance, recognized by its `__get__` method. Calling that method returns `10`.

Note that the value `10` is not stored in either the class dictionary or the instance dictionary. Instead, the value `10` is computed on demand.

This example shows how a simple descriptor works, but it isn’t very useful. For retrieving constants, normal attribute lookup would be better.

In the next section, we’ll create something more useful, a dynamic lookup.

Dynamic lookups

Interesting descriptors typically run computations instead of returning constants:

```
import os

class DirectorySize:

    def __get__(self, obj, objtype=None):
        return len(os.listdir(obj.dirname))

class Directory:

    size = DirectorySize()              # Descriptor instance

    def __init__(self, dirname):
        self.dirname = dirname          # Regular instance
attribute
```

An interactive session shows that the lookup is dynamic — it computes different, updated answers each time:

```
>>> s = Directory('songs')
>>> g = Directory('games')
>>> s.size                              # The songs directory
has twenty files
20
>>> g.size                              # The games directory
has three files
3
>>> os.remove('games/chess')            # Delete a game
>>> g.size                              # File count is
automatically updated
2
```

Besides showing how descriptors can run computations, this example also reveals the purpose of the parameters to `__get__()`. The *self* parameter is *size*, an instance of *DirectorySize*. The *obj* parameter is either *g* or *s*, an instance of *Directory*. It is the *obj* parameter that lets the `__get__()` method learn the target directory. The *objtype* parameter is the class *Directory*.

Managed attributes

A popular use for descriptors is managing access to instance data. The descriptor is assigned to a public attribute in the class dictionary while the actual data is stored as a private attribute in the instance dictionary. The descriptor’s `__get__()` and `__set__()` methods are triggered when the public attribute is accessed.

In the following example, *age* is the public attribute and _*age* is the private attribute. When the public attribute is accessed, the descriptor logs the lookup or update:

```
import logging

logging.basicConfig(level=logging.INFO)

class LoggedAgeAccess:

    def __get__(self, obj, objtype=None):
        value = obj._age
        logging.info('Accessing %r giving %r', 'age', value)
        return value

    def __set__(self, obj, value):
        logging.info('Updating %r to %r', 'age', value)
        obj._age = value

class Person:

    age = LoggedAgeAccess()             # Descriptor instance

    def __init__(self, name, age):
        self.name = name                # Regular instance
attribute
        self.age = age                  # Calls __set__()

    def birthday(self):
        self.age += 1                   # Calls both __get__()
and __set__()
```

An interactive session shows that all access to the managed attribute *age* is logged, but that the regular attribute *name* is not logged:

```
>>> mary = Person('Mary M', 30)         # The initial age
update is logged
INFO:root:Updating 'age' to 30
>>> dave = Person('David D', 40)
INFO:root:Updating 'age' to 40

>>> vars(mary)                          # The actual data is in
a private attribute
{'name': 'Mary M', '_age': 30}
>>> vars(dave)
{'name': 'David D', '_age': 40}

>>> mary.age                            # Access the data and
log the lookup
INFO:root:Accessing 'age' giving 30
30
>>> mary.birthday()                     # Updates are logged as
well
INFO:root:Accessing 'age' giving 30
INFO:root:Updating 'age' to 31

>>> dave.name                           # Regular attribute
lookup isn't logged
'David D'
>>> dave.age                            # Only the managed
attribute is logged
INFO:root:Accessing 'age' giving 40
40
```

One major issue with this example is that the private name *_age* is hardwired in the *LoggedAgeAccess* class. That means that each instance can only have one logged attribute and that its name is unchangeable. In the next example, we’ll fix that problem.

Customized names

When a class uses descriptors, it can inform each descriptor about which variable name was used.

In this example, the **`Person`** class has two descriptor instances, *name* and *age*. When the **`Person`** class is defined, it makes a callback to `__set_name__()` in *LoggedAccess* so that the field names can be recorded, giving each descriptor its own *public_name* and *private_name*:

```
import logging

logging.basicConfig(level=logging.INFO)

class LoggedAccess:

    def __set_name__(self, owner, name):
        self.public_name = name
        self.private_name = '_' + name

    def __get__(self, obj, objtype=None):
        value = getattr(obj, self.private_name)
        logging.info('Accessing %r giving %r',
self.public_name, value)
        return value

    def __set__(self, obj, value):
        logging.info('Updating %r to %r', self.public_name,
value)
        setattr(obj, self.private_name, value)

class Person:

    name = LoggedAccess()                # First descriptor
instance
    age = LoggedAccess()                 # Second descriptor
instance

    def __init__(self, name, age):
        self.name = name                 # Calls the first
descriptor
        self.age = age                   # Calls the second
```

```
descriptor

    def birthday(self):
        self.age += 1
```

An interactive session shows that the **`Person`** class has called **`__set_name__()`** so that the field names would be recorded. Here we call **`vars()`** to look up the descriptor without triggering it:

```
>>> vars(vars(Person)['name'])
{'public_name': 'name', 'private_name': '_name'}
>>> vars(vars(Person)['age'])
{'public_name': 'age', 'private_name': '_age'}
```

The new class now logs access to both *name* and *age*:

```
>>> pete = Person('Peter P', 10)
INFO:root:Updating 'name' to 'Peter P'
INFO:root:Updating 'age' to 10
>>> kate = Person('Catherine C', 20)
INFO:root:Updating 'name' to 'Catherine C'
INFO:root:Updating 'age' to 20
```

The two *Person* instances contain only the private names:

```
>>> vars(pete)
{'_name': 'Peter P', '_age': 10}
>>> vars(kate)
{'_name': 'Catherine C', '_age': 20}
```

Closing thoughts

A descriptor is what we call any object that defines **`__get__()`**, **`__set__()`**, or **`__delete__()`**.

Optionally, descriptors can have a **`__set_name__()`** method. This is only used in cases where a descriptor needs to know either the class where it was created or the name of class variable it was assigned to. (This method, if present, is called even if the class is not a descriptor.)

Descriptors get invoked by the dot operator during attribute lookup. If a descriptor is accessed indirectly with `vars(some_class)[descriptor_name]`, the descriptor instance is returned without invoking it.

Descriptors only work when used as class variables. When put in instances, they have no effect.

The main motivation for descriptors is to provide a hook allowing objects stored in class variables to control what happens during attribute lookup.

Traditionally, the calling class controls what happens during lookup. Descriptors invert that relationship and allow the data being looked-up to have a say in the matter.

Descriptors are used throughout the language. It is how functions turn into bound methods. Common tools like `classmethod()`, `staticmethod()`, `property()`, and `functools.cached_property()` are all implemented as descriptors.

Complete Practical Example

In this example, we create a practical and powerful tool for locating notoriously hard to find data corruption bugs.

Validator class

A validator is a descriptor for managed attribute access. Prior to storing any data, it verifies that the new value meets various type and range restrictions. If those restrictions aren’t met, it raises an exception to prevent data corruption at its source.

This **`Validator`** class is both an abstract base class and a managed attribute descriptor:

```
from abc import ABC, abstractmethod

class Validator(ABC):

    def __set_name__(self, owner, name):
        self.private_name = '_' + name

    def __get__(self, obj, objtype=None):
        return getattr(obj, self.private_name)

    def __set__(self, obj, value):
        self.validate(value)
        setattr(obj, self.private_name, value)

    @abstractmethod
    def validate(self, value):
        pass
```

Custom validators need to inherit from **`Validator`** and must supply a **`validate()`** method to test various restrictions as needed.

Custom validators

Here are three practical data validation utilities:

1. **`OneOf`** verifies that a value is one of a restricted set of options.
2. **`Number`** verifies that a value is either an **`int`** or **`float`**. Optionally, it verifies that a value is between a given minimum or maximum.
3. **`String`** verifies that a value is a **`str`**. Optionally, it validates a given minimum or maximum length. It can

validate a user-defined predicate [https://en.wikipedia.org/wiki/Predicate_(mathematical_logic)] as well.

```
class OneOf(Validator):

    def __init__(self, *options):
        self.options = set(options)

    def validate(self, value):
        if value not in self.options:
            raise ValueError(
                f'Expected {value!r} to be one of
{self.options!r}'
            )

class Number(Validator):

    def __init__(self, minvalue=None, maxvalue=None):
        self.minvalue = minvalue
        self.maxvalue = maxvalue

    def validate(self, value):
        if not isinstance(value, (int, float)):
            raise TypeError(f'Expected {value!r} to be an int
or float')
        if self.minvalue is not None and value < self.minvalue:
            raise ValueError(
                f'Expected {value!r} to be at least
{self.minvalue!r}'
            )
        if self.maxvalue is not None and value > self.maxvalue:
            raise ValueError(
                f'Expected {value!r} to be no more than
{self.maxvalue!r}'
            )

class String(Validator):

    def __init__(self, minsize=None, maxsize=None,
predicate=None):
        self.minsize = minsize
        self.maxsize = maxsize
        self.predicate = predicate

    def validate(self, value):
        if not isinstance(value, str):
            raise TypeError(f'Expected {value!r} to be a str')
        if self.minsize is not None and len(value) <
self.minsize:
            raise ValueError(
                f'Expected {value!r} to be no smaller than
{self.minsize!r}'
            )
        if self.maxsize is not None and len(value) >
self.maxsize:
            raise ValueError(
                f'Expected {value!r} to be no bigger than
{self.maxsize!r}'
            )
        if self.predicate is not None and not
self.predicate(value):
            raise ValueError(
                f'Expected {self.predicate} to be true for
{value!r}'
            )
```

Practical application

Here’s how the data validators can be used in a real class:

```
class Component:

    name = String(minsize=3, maxsize=10, predicate=str.isupper)
    kind = OneOf('wood', 'metal', 'plastic')
    quantity = Number(minvalue=0)

    def __init__(self, name, kind, quantity):
        self.name = name
        self.kind = kind
        self.quantity = quantity
```

The descriptors prevent invalid instances from being created:

```
>>> Component('Widget', 'metal', 5)      # Blocked: 'Widget' is
not all uppercase
Traceback (most recent call last):
    ...
ValueError: Expected <method 'isupper' of 'str' objects> to be
true for 'Widget'

>>> Component('WIDGET', 'metle', 5)      # Blocked: 'metle' is
misspelled
Traceback (most recent call last):
    ...
ValueError: Expected 'metle' to be one of {'metal', 'plastic',
'wood'}

>>> Component('WIDGET', 'metal', -5)     # Blocked: -5 is
negative
Traceback (most recent call last):
    ...
ValueError: Expected -5 to be at least 0

>>> Component('WIDGET', 'metal', 'V')    # Blocked: 'V' isn't a
number
Traceback (most recent call last):
    ...
TypeError: Expected 'V' to be an int or float

>>> c = Component('WIDGET', 'metal', 5)  # Allowed:  The inputs
are valid
```

Technical Tutorial

What follows is a more technical tutorial for the mechanics and details of how descriptors work.

Abstract

Defines descriptors, summarizes the protocol, and shows how descriptors are called. Provides an example showing how object relational mappings work.

Learning about descriptors not only provides access to a larger toolset, it creates a deeper understanding of how Python works.

Definition and introduction

In general, a descriptor is an attribute value that has one of the methods in the descriptor protocol. Those methods are `__get__()`, `__set__()`, and `__delete__()`. If any of those methods are defined for an attribute, it is said to be a descriptor.

The default behavior for attribute access is to get, set, or delete the attribute from an object's dictionary. For instance, `a.x` has a lookup chain starting with `a.__dict__['x']`, then `type(a).__dict__['x']`, and continuing through the method resolution order of `type(a)`. If the looked-up value is an object defining one of the descriptor methods, then Python may override the default behavior and invoke the descriptor method instead. Where this occurs in the precedence chain depends on which descriptor methods were defined.

Descriptors are a powerful, general purpose protocol. They are the mechanism behind properties, methods, static methods, class methods, and `super()`. They are used throughout Python itself. Descriptors simplify the underlying C code and offer a flexible set of new tools for everyday Python programs.

Descriptor protocol

```
descr.__get__(self, obj, type=None)
```

```
descr.__set__(self, obj, value)
```

```
descr.__delete__(self, obj)
```

That is all there is to it. Define any of these methods and an object is considered a descriptor and can override default behavior upon being looked up as an attribute.

If an object defines `__set__()` or `__delete__()`, it is considered a data descriptor. Descriptors that only define `__get__()` are called non-data descriptors (they are often used for methods but other uses are possible).

Data and non-data descriptors differ in how overrides are calculated with respect to entries in an instance's dictionary. If an instance's dictionary has an entry with the same name as a data descriptor, the data descriptor takes precedence. If an instance's dictionary has an entry with the same name as a non-data descriptor, the dictionary entry takes precedence.

To make a read-only data descriptor, define both `__get__()` and `__set__()` with the `__set__()` raising an `AttributeError` when called. Defining the `__set__()` method with an exception raising placeholder is enough to make it a data descriptor.

Overview of descriptor invocation

A descriptor can be called directly with `desc.__get__(obj)` or `desc.__get__(None, cls)`.

But it is more common for a descriptor to be invoked automatically from attribute access.

The expression `obj.x` looks up the attribute `x` in the chain of namespaces for `obj`. If the search finds a descriptor outside of the instance `__dict__`, its `__get__()` method is invoked according to the precedence rules listed below.

The details of invocation depend on whether `obj` is an object, class, or instance of super.

Invocation from an instance

Instance lookup scans through a chain of namespaces giving data descriptors the highest priority, followed by instance variables, then non-data descriptors, then class variables, and lastly `__getattr__()` if it is provided.

If a descriptor is found for `a.x`, then it is invoked with: `desc.__get__(a, type(a))`.

The logic for a dotted lookup is in `object.__getattribute__()`. Here is a pure Python equivalent:

```
def find_name_in_mro(cls, name, default):
    "Emulate _PyType_Lookup() in Objects/typeobject.c"
    for base in cls.__mro__:
        if name in vars(base):
            return vars(base)[name]
    return default

def object_getattribute(obj, name):
    "Emulate PyObject_GenericGetAttr() in Objects/object.c"
    null = object()
    objtype = type(obj)
    cls_var = find_name_in_mro(objtype, name, null)
    descr_get = getattr(type(cls_var), '__get__', null)
    if descr_get is not null:
        if (hasattr(type(cls_var), '__set__')
            or hasattr(type(cls_var), '__delete__')):
            return descr_get(cls_var, obj, objtype)     # data
descriptor
    if hasattr(obj, '__dict__') and name in vars(obj):
        return vars(obj)[name]                          #
instance variable
    if descr_get is not null:
        return descr_get(cls_var, obj, objtype)         # non-
data descriptor
```

```
    if cls_var is not null:
        return cls_var                                   # class
variable
    raise AttributeError(name)
```

Note, there is no `__getattr__()` hook in the `__getattribute__()` code. That is why calling `__getattribute__()` directly or with `super().__getattribute__` will bypass `__getattr__()` entirely.

Instead, it is the dot operator and the `getattr()` function that are responsible for invoking `__getattr__()` whenever `__getattribute__()` raises an `AttributeError`. Their logic is encapsulated in a helper function:

```
def getattr_hook(obj, name):
    "Emulate slot_tp_getattr_hook() in Objects/typeobject.c"
    try:
        return obj.__getattribute__(name)
    except AttributeError:
        if not hasattr(type(obj), '__getattr__'):
            raise
    return type(obj).__getattr__(obj, name)             #
__getattr__
```

Invocation from a class

The logic for a dotted lookup such as `A.x` is in **`type.__getattribute__()`**. The steps are similar to those for **`object.__getattribute__()`** but the instance dictionary lookup is replaced by a search through the class’s method resolution order.

If a descriptor is found, it is invoked with `desc.__get__(None, A)`.

The full C implementation can be found in **`type_getattro()`** and **`_PyType_Lookup()`** in Objects/typeobject.c [https://github.com/python/cpython/tree/3.14/Objects/typeobject.c].

Invocation from super

The logic for super’s dotted lookup is in the `__getattribute__()` method for object returned by `super()`.

A dotted lookup such as `super(A, obj).m` searches `obj.__class__.__mro__` for the base class `B` immediately following `A` and then returns `B.__dict__['m'].__get__(obj, A)`. If not a descriptor, `m` is returned unchanged.

The full C implementation can be found in **`super_getattro()`** in Objects/typeobject.c [https://github.com/python/cpython/tree/3.14/Objects/typeobject.c]. A pure Python equivalent can be found in Guido’s Tutorial [https://www.python.org/download/releases/2.2.3/descrintro/#cooperation].

Summary of invocation logic

The mechanism for descriptors is embedded in the `__getattribute__()` methods for `object`, `type`, and `super()`.

The important points to remember are:

- Descriptors are invoked by the `__getattribute__()` method.
- Classes inherit this machinery from `object`, `type`, or `super()`.
- Overriding `__getattribute__()` prevents automatic descriptor calls because all the descriptor logic is in that method.
- **`object.__getattribute__()`** and **`type.__getattribute__()`** make different calls to `__get__()`. The first includes the instance and may include the class. The second puts in `None` for the instance and always includes the class.
- Data descriptors always override instance dictionaries.

- Non-data descriptors may be overridden by instance dictionaries.

Automatic name notification

Sometimes it is desirable for a descriptor to know what class variable name it was assigned to. When a new class is created, the `type` metaclass scans the dictionary of the new class. If any of the entries are descriptors and if they define `__set_name__()`, that method is called with two arguments. The *owner* is the class where the descriptor is used, and the *name* is the class variable the descriptor was assigned to.

The implementation details are in **`type_new()`** and **`set_names()`** in Objects/typeobject.c [https://github.com/python/cpython/tree/3.14/Objects/typeobject.c].

Since the update logic is in **`type.__new__()`**, notifications only take place at the time of class creation. If descriptors are added to the class afterwards, `__set_name__()` will need to be called manually.

ORM example

The following code is a simplified skeleton showing how data descriptors could be used to implement an object relational mapping [https://en.wikipedia.org/wiki/Object%E2%80%93relational_mapping].

The essential idea is that the data is stored in an external database. The Python instances only hold keys to the database’s tables. Descriptors take care of lookups or updates:

```
class Field:

    def __set_name__(self, owner, name):
        self.fetch = f'SELECT {name} FROM {owner.table} WHERE
{owner.key}=?;'
        self.store = f'UPDATE {owner.table} SET {name}=? WHERE
{owner.key}=?;'

    def __get__(self, obj, objtype=None):
        return conn.execute(self.fetch, [obj.key]).fetchone()
[0]

    def __set__(self, obj, value):
        conn.execute(self.store, [value, obj.key])
        conn.commit()
```

We can use the **`Field`** class to define models [https://en.wikipedia.org/wiki/Database_model] that describe the schema for each table in a database:

```
class Movie:
    table = 'Movies'                    # Table name
    key = 'title'                       # Primary key
    director = Field()
    year = Field()

    def __init__(self, key):
        self.key = key

class Song:
    table = 'Music'
    key = 'title'
    artist = Field()
    year = Field()
    genre = Field()

    def __init__(self, key):
        self.key = key
```

To use the models, first connect to the database:

```
>>> import sqlite3
>>> conn = sqlite3.connect('entertainment.db')
```

An interactive session shows how data is retrieved from the database and how it can be updated:

```
>>> Movie('Star Wars').director
'George Lucas'
>>> jaws = Movie('Jaws')
>>> f'Released in {jaws.year} by {jaws.director}'
'Released in 1975 by Steven Spielberg'

>>> Song('Country Roads').artist
'John Denver'

>>> Movie('Star Wars').director = 'J.J. Abrams'
>>> Movie('Star Wars').director
'J.J. Abrams'
```

Pure Python Equivalents

The descriptor protocol is simple and offers exciting possibilities. Several use cases are so common that they have been prepackaged into built-in tools. Properties, bound methods, static methods, class methods, and __slots__ are all based on the descriptor protocol.

Properties

Calling `property()` is a succinct way of building a data descriptor that triggers a function call upon access to an attribute. Its signature is:

```
property(fget=None, fset=None, fdel=None, doc=None) -> property
```

The documentation shows a typical use to define a managed attribute `x`:

```
class C:
    def getx(self): return self.__x
    def setx(self, value): self.__x = value
    def delx(self): del self.__x
    x = property(getx, setx, delx, "I'm the 'x' property.")
```

To see how `property()` is implemented in terms of the descriptor protocol, here is a pure Python equivalent that implements most of the core functionality:

```
class Property:
    "Emulate PyProperty_Type() in Objects/descrobject.c"

    def __init__(self, fget=None, fset=None, fdel=None,
doc=None):
        self.fget = fget
        self.fset = fset
        self.fdel = fdel
        if doc is None and fget is not None:
            doc = fget.__doc__
        self.__doc__ = doc

    def __set_name__(self, owner, name):
        self.__name__ = name

    def __get__(self, obj, objtype=None):
        if obj is None:
            return self
        if self.fget is None:
            raise AttributeError
        return self.fget(obj)

    def __set__(self, obj, value):
        if self.fset is None:
            raise AttributeError
        self.fset(obj, value)

    def __delete__(self, obj):
        if self.fdel is None:
            raise AttributeError
        self.fdel(obj)

    def getter(self, fget):
        return type(self)(fget, self.fset, self.fdel,
self.__doc__)
```

```
    def setter(self, fset):
        return type(self)(self.fget, fset, self.fdel,
self.__doc__)

    def deleter(self, fdel):
        return type(self)(self.fget, self.fset, fdel,
self.__doc__)
```

The `property()` builtin helps whenever a user interface has granted attribute access and then subsequent changes require the intervention of a method.

For instance, a spreadsheet class may grant access to a cell value through `Cell('b10').value`. Subsequent improvements to the program require the cell to be recalculated on every access; however, the programmer does not want to affect existing client code accessing the attribute directly. The solution is to wrap access to the value attribute in a property data descriptor:

```
class Cell:
    ...

    @property
    def value(self):
        "Recalculate the cell before returning value"
        self.recalc()
        return self._value
```

Either the built-in `property()` or our **`Property()`** equivalent would work in this example.

Functions and methods

Python’s object oriented features are built upon a function based environment. Using non-data descriptors, the two are merged seamlessly.

Functions stored in class dictionaries get turned into methods when invoked. Methods only differ from regular functions in that the object instance is prepended to the other arguments. By convention, the instance is called *self* but could be called *this* or any other variable name.

Methods can be created manually with `types.MethodType` which is roughly equivalent to:

```
class MethodType:
    "Emulate PyMethod_Type in Objects/classobject.c"

    def __init__(self, func, obj):
        self.__func__ = func
        self.__self__ = obj

    def __call__(self, *args, **kwargs):
        func = self.__func__
        obj = self.__self__
        return func(obj, *args, **kwargs)

    def __getattribute__(self, name):
        "Emulate method_getset() in Objects/classobject.c"
        if name == '__doc__':
            return self.__func__.__doc__
        return object.__getattribute__(self, name)

    def __getattr__(self, name):
        "Emulate method_getattro() in Objects/classobject.c"
        return getattr(self.__func__, name)

    def __get__(self, obj, objtype=None):
        "Emulate method_descr_get() in Objects/classobject.c"
        return self
```

To support automatic creation of methods, functions include the `__get__()` method for binding methods during attribute access. This means that functions are non-data descriptors that return bound methods during dotted lookup from an instance. Here’s how it works:

```
class Function:
    ...

    def __get__(self, obj, objtype=None):
        "Simulate func_descr_get() in Objects/funcobject.c"
        if obj is None:
            return self
        return MethodType(self, obj)
```

Running the following class in the interpreter shows how the function descriptor works in practice:

```
class D:
    def f(self):
         return self

class D2:
    pass
```

The function has a qualified name attribute to support introspection:

```
>>> D.f.__qualname__
'D.f'
```

Accessing the function through the class dictionary does not invoke **`__get__()`**. Instead, it just returns the underlying function object:

```
>>> D.__dict__['f']
<function D.f at 0x00C45070>
```

Dotted access from a class calls **`__get__()`** which just returns the underlying function unchanged:

```
>>> D.f
<function D.f at 0x00C45070>
```

The interesting behavior occurs during dotted access from an instance. The dotted lookup calls **`__get__()`** which returns a bound method object:

```
>>> d = D()
>>> d.f
<bound method D.f of <__main__.D object at 0x00B18C90>>
```

Internally, the bound method stores the underlying function and the bound instance:

```
>>> d.f.__func__
<function D.f at 0x00C45070>

>>> d.f.__self__
<__main__.D object at 0x00B18C90>
```

If you have ever wondered where *self* comes from in regular methods or where *cls* comes from in class methods, this is it!

Kinds of methods

Non-data descriptors provide a simple mechanism for variations on the usual patterns of binding functions into methods.

To recap, functions have a `__get__()` method so that they can be converted to a method when accessed as attributes. The non-data descriptor transforms an `obj.f(*args)` call into `f(obj, *args)`. Calling `cls.f(*args)` becomes `f(*args)`.

This chart summarizes the binding and its two most useful variants:

Transformation	Called from an object	Called from a class
function	f(obj, *args)	f(*args)
staticmethod	f(*args)	f(*args)
classmethod	f(type(obj), *args)	f(cls, *args)

Static methods

Static methods return the underlying function without changes. Calling either `c.f` or `C.f` is the equivalent of a direct lookup into `object.__getattribute__(c, "f")` or `object.__getattribute__(C, "f")`. As a result, the function becomes identically accessible from either an object or a class.

Good candidates for static methods are methods that do not reference the `self` variable.

For instance, a statistics package may include a container class for experimental data. The class provides normal methods for computing the average, mean, median, and other descriptive statistics that depend on the data. However, there may be useful functions which are conceptually related but do not depend on the data. For instance, `erf(x)` is handy conversion routine that comes up in statistical work but does not directly depend on a particular dataset. It can be called either from an object or the class: `s.erf(1.5) --> 0.9332` or `Sample.erf(1.5) --> 0.9332`.

Since static methods return the underlying function with no changes, the example calls are unexciting:

```
class E:
    @staticmethod
    def f(x):
        return x * 10
```

```
>>> E.f(3)
30
>>> E().f(3)
30
```

Using the non-data descriptor protocol, a pure Python version of `staticmethod()` would look like this:

```
import functools

class StaticMethod:
    "Emulate PyStaticMethod_Type() in Objects/funcobject.c"

    def __init__(self, f):
        self.f = f
        functools.update_wrapper(self, f)

    def __get__(self, obj, objtype=None):
        return self.f

    def __call__(self, *args, **kwds):
        return self.f(*args, **kwds)

    @property
    def __annotations__(self):
        return self.f.__annotations__
```

The `functools.update_wrapper()` call adds a `__wrapped__` attribute that refers to the underlying function. Also it carries forward the attributes necessary to make the wrapper look like the wrapped function, including `__name__`, `__qualname__`, and `__doc__`.

Class methods

Unlike static methods, class methods prepend the class reference to the argument list before calling the function. This format is the same for whether the caller is an object or a class:

```
class F:
    @classmethod
    def f(cls, x):
        return cls.__name__, x
```

```
>>> F.f(3)
('F', 3)
>>> F().f(3)
('F', 3)
```

This behavior is useful whenever the method only needs to have a class reference and does not rely on data stored in a specific instance. One use for class methods is to create alternate class constructors. For example, the classmethod `dict.fromkeys()` creates a new dictionary from a list of keys. The pure Python equivalent is:

```
class Dict(dict):
    @classmethod
    def fromkeys(cls, iterable, value=None):
        "Emulate dict_fromkeys() in Objects/dictobject.c"
        d = cls()
        for key in iterable:
            d[key] = value
        return d
```

Now a new dictionary of unique keys can be constructed like this:

```
>>> d = Dict.fromkeys('abracadabra')
>>> type(d) is Dict
True
```

```
>>> d
{'a': None, 'b': None, 'r': None, 'c': None, 'd': None}
```

Using the non-data descriptor protocol, a pure Python version of `classmethod()` would look like this:

```
import functools

class ClassMethod:
    "Emulate PyClassMethod_Type() in Objects/funcobject.c"

    def __init__(self, f):
        self.f = f
        functools.update_wrapper(self, f)

    def __get__(self, obj, cls=None):
        if cls is None:
            cls = type(obj)
        return MethodType(self.f, cls)
```

The `functools.update_wrapper()` call in `ClassMethod` adds a `__wrapped__` attribute that refers to the underlying function. Also it carries forward the attributes necessary to make the wrapper look like the wrapped function: `__name__`, `__qualname__`, `__doc__`, and `__annotations__`.

Member objects and __slots__

When a class defines `__slots__`, it replaces instance dictionaries with a fixed-length array of slot values. From a user point of view that has several effects:

1. Provides immediate detection of bugs due to misspelled attribute assignments. Only attribute names specified in `__slots__` are allowed:

```
class Vehicle:
    __slots__ = ('id_number', 'make', 'model')
```

```
>>> auto = Vehicle()
>>> auto.id_nubmer = 'VYE483814LQEX'
Traceback (most recent call last):
    ...
AttributeError: 'Vehicle' object has no attribute 'id_nubmer'
```

2. Helps create immutable objects where descriptors manage access to private attributes stored in `__slots__`:

```
class Immutable:

    __slots__ = ('_dept', '_name')          # Replace the
instance dictionary

    def __init__(self, dept, name):
        self._dept = dept                   # Store to private
attribute
        self._name = name                   # Store to private
attribute

    @property                               # Read-only
descriptor
    def dept(self):
        return self._dept

    @property
    def name(self):                         # Read-only
descriptor
        return self._name
```

```
>>> mark = Immutable('Botany', 'Mark Watney')
>>> mark.dept
'Botany'
>>> mark.dept = 'Space Pirate'
Traceback (most recent call last):
    ...
AttributeError: property 'dept' of 'Immutable' object has no
setter
>>> mark.location = 'Mars'
Traceback (most recent call last):
    ...
AttributeError: 'Immutable' object has no attribute 'location'
```

3. Saves memory. On a 64-bit Linux build, an instance with two attributes takes 48 bytes with `__slots__` and 152 bytes without. This flyweight design pattern [https://en.wikipedia.org/wiki/Flyweight_pattern] likely only matters when a large number of instances are going to be created.

4. Improves speed. Reading instance variables is 35% faster with `__slots__` (as measured with Python 3.10 on an Apple M1 processor).

5. Blocks tools like **`functools.cached_property()`** which require an instance dictionary to function correctly:

```
from functools import cached_property

class CP:
    __slots__ = ()                          # Eliminates the
instance dict

    @cached_property                        # Requires an
instance dict
    def pi(self):
        return 4 * sum((-1.0)**n / (2.0*n + 1.0)
                       for n in reversed(range(100_000)))

>>> CP().pi
Traceback (most recent call last):
  ...
TypeError: No '__dict__' attribute on 'CP' instance to cache
'pi' property.
```

It is not possible to create an exact drop-in pure Python version of `__slots__` because it requires direct access to C structures and control over object memory allocation. However, we can build a mostly faithful simulation where the actual C structure for slots is emulated by a private `_slotvalues` list. Reads and writes to that private structure are managed by member descriptors:

```
null = object()

class Member:

    def __init__(self, name, clsname, offset):
        'Emulate PyMemberDef in Include/structmember.h'
        # Also see descr_new() in Objects/descrobject.c
        self.name = name
        self.clsname = clsname
        self.offset = offset

    def __get__(self, obj, objtype=None):
        'Emulate member_get() in Objects/descrobject.c'
        # Also see PyMember_GetOne() in Python/structmember.c
        if obj is None:
            return self
        value = obj._slotvalues[self.offset]
        if value is null:
            raise AttributeError(self.name)
        return value

    def __set__(self, obj, value):
        'Emulate member_set() in Objects/descrobject.c'
        obj._slotvalues[self.offset] = value

    def __delete__(self, obj):
        'Emulate member_delete() in Objects/descrobject.c'
        value = obj._slotvalues[self.offset]
        if value is null:
            raise AttributeError(self.name)
        obj._slotvalues[self.offset] = null

    def __repr__(self):
        'Emulate member_repr() in Objects/descrobject.c'
        return f'<Member {self.name!r} of {self.clsname!r}>'
```

The **`type.__new__()`** method takes care of adding member objects to class variables:

```
class Type(type):
    'Simulate how the type metaclass adds member objects for 
slots'
```

```
    def __new__(mcls, clsname, bases, mapping, **kwargs):
        'Emulate type_new() in Objects/typeobject.c'
        # type_new() calls PyTypeReady() which calls
add_methods()
        slot_names = mapping.get('slot_names', [])
        for offset, name in enumerate(slot_names):
            mapping[name] = Member(name, clsname, offset)
        return type.__new__(mcls, clsname, bases, mapping,
**kwargs)
```

The `object.__new__()` method takes care of creating instances that have slots instead of an instance dictionary. Here is a rough simulation in pure Python:

```
class Object:
    'Simulate how object.__new__() allocates memory for
__slots__'

    def __new__(cls, *args, **kwargs):
        'Emulate object_new() in Objects/typeobject.c'
        inst = super().__new__(cls)
        if hasattr(cls, 'slot_names'):
            empty_slots = [null] * len(cls.slot_names)
            object.__setattr__(inst, '_slotvalues',
empty_slots)
        return inst

    def __setattr__(self, name, value):
        'Emulate _PyObject_GenericSetAttrWithDict()
Objects/object.c'
        cls = type(self)
        if hasattr(cls, 'slot_names') and name not in
cls.slot_names:
            raise AttributeError(
                f'{cls.__name__!r} object has no attribute
{name!r}'
            )
        super().__setattr__(name, value)

    def __delattr__(self, name):
        'Emulate _PyObject_GenericSetAttrWithDict()
```

```
Objects/object.c'
        cls = type(self)
        if hasattr(cls, 'slot_names') and name not in
cls.slot_names:
            raise AttributeError(
                f'{cls.__name__!r} object has no attribute
{name!r}'
            )
        super().__delattr__(name)
```

To use the simulation in a real class, just inherit from **`Object`** and set the metaclass to **`Type`**:

```
class H(Object, metaclass=Type):
    'Instance variables stored in slots'

    slot_names = ['x', 'y']

    def __init__(self, x, y):
        self.x = x
        self.y = y
```

At this point, the metaclass has loaded member objects for *x* and *y*:

```
>>> from pprint import pp
>>> pp(dict(vars(H)))
{'__module__': '__main__',
 '__doc__': 'Instance variables stored in slots',
 'slot_names': ['x', 'y'],
 '__init__': <function H.__init__ at 0x7fb5d302f9d0>,
 'x': <Member 'x' of 'H'>,
 'y': <Member 'y' of 'H'>}
```

When instances are created, they have a `slot_values` list where the attributes are stored:

```
>>> h = H(10, 20)
>>> vars(h)
{'_slotvalues': [10, 20]}
>>> h.x = 55
>>> vars(h)
{'_slotvalues': [55, 20]}
```

Misspelled or unassigned attributes will raise an exception:

```
>>> h.xz
Traceback (most recent call last):
    ...
AttributeError: 'H' object has no attribute 'xz'
```

Debugging C API extensions and CPython Internals with GDB

This document explains how the Python GDB extension, `python-gdb.py`, can be used with the GDB debugger to debug CPython extensions and the CPython interpreter itself.

When debugging low-level problems such as crashes or deadlocks, a low-level debugger, such as GDB, is useful to diagnose and correct the issue. By default, GDB (or any of its front-ends) doesn’t support high-level information specific to the CPython interpreter.

The `python-gdb.py` extension adds CPython interpreter information to GDB. The extension helps introspect the stack of currently executing Python functions. Given a Python object represented by a PyObject* pointer, the extension surfaces the type and value of the object.

Developers who are working on CPython extensions or tinkering with parts of CPython that are written in C can use this document to learn how to use the `python-gdb.py` extension with GDB.

Note

This document assumes that you are familiar with the basics of GDB and the CPython C API. It consolidates guidance from the devguide [https://devguide.python.org] and the Python wiki [https://wiki.python.org/moin/DebuggingWithGdb].

Prerequisites

You need to have:

- GDB 7 or later. (For earlier versions of GDB, see `Misc/gdbinit` in the sources of Python 3.11 or earlier.)
- GDB-compatible debugging information for Python and any extension you are debugging.
- The `python-gdb.py` extension.

The extension is built with Python, but might be distributed separately or not at all. Below, we include tips for a few common systems as examples. Note that even if the instructions match your system, they might be outdated.

Setup with Python built from source

When you build CPython from source, debugging information should be available, and the build should add a `python-gdb.py` file to the root directory of your repository.

To activate support, you must add the directory containing `python-gdb.py` to GDB’s “auto-load-safe-path”. If you haven’t done this, recent versions of GDB will print out a warning with instructions on how to do this.

Note

If you do not see instructions for your version of GDB, put this in your configuration file (`~/.gdbinit` or `~/.config/gdb/gdbinit`):

```
add-auto-load-safe-path /path/to/cpython
```

You can also add multiple paths, separated by `:`.

Setup for Python from a Linux distro

Most Linux systems provide debug information for the system Python in a package called `python-debuginfo`, `python-dbg` or similar. For example:

- Fedora:

  ```
  sudo dnf install gdb
  sudo dnf debuginfo-install python3
  ```

- Ubuntu:

  ```
  sudo apt install gdb python3-dbg
  ```

On several recent Linux systems, GDB can download debugging symbols automatically using *debuginfod*. However, this will not install the `python-gdb.py` extension; you generally do need to install the debug info package separately.

Using the Debug build and Development mode

For easier debugging, you might want to:

- Use a debug build of Python. (When building from source, use `configure --with-pydebug`. On Linux distros, install and run a package like `python-debug` or `python-dbg`, if available.)
- Use the runtime development mode (`-X dev`).

Both enable extra assertions and disable some optimizations. Sometimes this hides the bug you are trying to find, but in most cases they make the process easier.

Using the `python-gdb` extension

When the extension is loaded, it provides two main features: pretty printers for Python values, and additional commands.

Pretty-printers

This is what a GDB backtrace looks like (truncated) when this extension is enabled:

```
#0  0x000000000041a6b1 in PyObject_Malloc (nbytes=Cannot access
memory at address 0x7fffff7fefe8
) at Objects/obmalloc.c:748
#1  0x000000000041b7c0 in _PyObject_DebugMallocApi (id=111 'o',
nbytes=24) at Objects/obmalloc.c:1445
#2  0x000000000041b717 in _PyObject_DebugMalloc (nbytes=24) at
Objects/obmalloc.c:1412
#3  0x000000000044060a in _PyUnicode_New (length=11) at
Objects/unicodeobject.c:346
#4  0x00000000004466aa in PyUnicodeUCS2_DecodeUTF8Stateful
(s=0x5c2b8d "__lltrace__", size=11, errors=0x0, consumed=
    0x0) at Objects/unicodeobject.c:2531
#5  0x0000000000446647 in PyUnicodeUCS2_DecodeUTF8 (s=0x5c2b8d
"__lltrace__", size=11, errors=0x0)
    at Objects/unicodeobject.c:2495
#6  0x0000000000440d1b in PyUnicodeUCS2_FromStringAndSize
(u=0x5c2b8d "__lltrace__", size=11)
    at Objects/unicodeobject.c:551
#7  0x0000000000440d94 in PyUnicodeUCS2_FromString (u=0x5c2b8d
"__lltrace__") at Objects/unicodeobject.c:569
#8  0x0000000000584abd in PyDict_GetItemString (v=
    {'Yuck': <type at remote 0xad4730>, '__builtins__': <module
at remote 0x7ffff7fd5ee8>, '__file__':
'Lib/test/crashers/nasty_eq_vs_dict.py', '__package__': None,
'y': <Yuck(i=0) at remote 0xaacd80>, 'dict': {0: 0, 1: 1, 2: 2,
3: 3}, '__cached__': None, '__name__': '__main__', 'z':
<Yuck(i=0) at remote 0xaace60>, '__doc__': None}, key=
    0x5c2b8d "__lltrace__") at Objects/dictobject.c:2171
```

Notice how the dictionary argument to `PyDict_GetItemString` is displayed as its `repr()`, rather than an opaque `PyObject *` pointer.

The extension works by supplying a custom printing routine for values of type `PyObject *`. If you need to access lower-level details of an object, then cast the value to a pointer of the appropriate type. For example:

```
(gdb) p globals
$1 = {'__builtins__': <module at remote 0x7ffff7fb1868>,
'__name__':
'__main__', 'ctypes': <module at remote 0x7ffff7f14360>,
'__doc__': None,
'__package__': None}

(gdb) p *(PyDictObject*)globals
$2 = {ob_refcnt = 3, ob_type = 0x3dbdf85820, ma_fill = 5,
ma_used = 5,
ma_mask = 7, ma_table = 0x63d0f8, ma_lookup = 0x3dbdc7ea70
<lookdict_string>, ma_smalltable = {{me_hash =
7065186196740147912,
me_key = '__builtins__', me_value = <module at remote
0x7ffff7fb1868>},
{me_hash = -368181376027291943, me_key = '__name__',
me_value ='__main__'}, {me_hash = 0, me_key = 0x0, me_value =
0x0},
{me_hash = 0, me_key = 0x0, me_value = 0x0},
{me_hash = -9177857982131165996, me_key = 'ctypes',
me_value = <module at remote 0x7ffff7f14360>},
{me_hash = -8518757509529533123, me_key = '__doc__', me_value =
None},
{me_hash = 0, me_key = 0x0, me_value = 0x0}, {
  me_hash = 6614918939584953775, me_key = '__package__',
me_value = None}}}
```

Note that the pretty-printers do not actually call `repr()`. For basic types, they try to match its result closely.

An area that can be confusing is that the custom printer for some types look a lot like GDB’s built-in printer for standard types. For example, the pretty-printer for a Python `int` (PyLongObject*) gives a representation that is not distinguishable from one of a regular machine-level integer:

```
(gdb) p some_machine_integer
$3 = 42

(gdb) p some_python_integer
$4 = 42
```

The internal structure can be revealed with a cast to PyLongObject*:

```
(gdb) p *(PyLongObject*)some_python_integer
$5 = {ob_base = {ob_base = {ob_refcnt = 8, ob_type =
0x3dad39f5e0}, ob_size = 1},
ob_digit = {42}}
```

A similar confusion can arise with the `str` type, where the output looks a lot like gdb’s built-in printer for `char *`:

```
(gdb) p ptr_to_python_str
$6 = '__builtins__'
```

The pretty-printer for `str` instances defaults to using single-quotes (as does Python’s `repr` for strings) whereas the standard printer for `char *` values uses double-quotes and contains a hexadecimal address:

```
(gdb) p ptr_to_char_star
$7 = 0x6d72c0 "hello world"
```

Again, the implementation details can be revealed with a cast to PyUnicodeObject*:

```
(gdb) p *(PyUnicodeObject*)$6
$8 = {ob_base = {ob_refcnt = 33, ob_type = 0x3dad3a95a0},
length = 12,
str = 0x7ffff2128500, hash = 7065186196740147912, state = 1,
defenc = 0x0}
```

`py-list`

The extension adds a `py-list` command, which lists the Python source code (if any) for the current frame in the selected thread. The current line is marked with a “>”:

```
(gdb) py-list
 901        if options.profile:
 902            options.profile = False
 903            profile_me()
 904            return
 905
>906        u = UI()
 907        if not u.quit:
 908            try:
 909                gtk.main()
 910            except KeyboardInterrupt:
 911                # properly quit on a keyboard
interrupt...
```

Use `py-list START` to list at a different line number within the Python source, and `py-list START,END` to list a specific range of lines within the Python source.

`py-up` and `py-down`

The `py-up` and `py-down` commands are analogous to GDB’s regular `up` and `down` commands, but try to move at the level of CPython frames, rather than C frames.

GDB is not always able to read the relevant frame information, depending on the optimization level with which CPython was compiled. Internally, the commands look for C frames that are executing the default frame evaluation function (that is, the core bytecode interpreter loop within CPython) and look up the value of the related `PyFrameObject *`.

They emit the frame number (at the C level) within the thread.

For example:

```
(gdb) py-up
#37 Frame 0x9420b04, for file /usr/lib/python2.6/site-
packages/
gnome_sudoku/main.py, line 906, in start_game ()
    u = UI()
(gdb) py-up
#40 Frame 0x948e82c, for file /usr/lib/python2.6/site-
packages/
gnome_sudoku/gnome_sudoku.py, line 22, in start_game(main=
<module at remote 0xb771b7f4>)
    main.start_game()
(gdb) py-up
Unable to find an older python frame
```

so we’re at the top of the Python stack.

The frame numbers correspond to those displayed by GDB’s standard `backtrace` command. The command skips C frames which are not executing Python code.

Going back down:

```
(gdb) py-down
#37 Frame 0x9420b04, for file /usr/lib/python2.6/site-
packages/gnome_sudoku/main.py, line 906, in start_game ()
    u = UI()
(gdb) py-down
#34 (unable to read python frame information)
(gdb) py-down
#23 (unable to read python frame information)
(gdb) py-down
#19 (unable to read python frame information)
(gdb) py-down
#14 Frame 0x99262ac, for file /usr/lib/python2.6/site-
packages/gnome_sudoku/game_selector.py, line 201, in
run_swallowed_dialog (self=
<NewOrSavedGameSelector(new_game_model=<gtk.ListStore at
remote 0x98fab44>, puzzle=None, saved_games=
[{'gsd.auto_fills': 0, 'tracking': {}, 'trackers': {},
'notes': [], 'saved_at': 1270084485, 'game': '7 8 0 0 0 0 0
5 6 0 0 9 0 8 0 1 0 0 0 4 6 0 0 0 0 7 0 6 5 0 0 0 4 7 9 2 0
0 0 9 0 1 0 0 0 3 9 7 6 0 0 0 1 8 0 6 0 0 0 0 2 8 0 0 0 5 0
4 0 6 0 0 2 1 0 0 0 0 0 4 5\n7 8 0 0 0 0 0 5 6 0 0 9 0 8 0
1 0 0 0 4 6 0 0 0 0 7 0 6 5 1 8 3 4 7 9 2 0 0 0 9 0 1 0 0 0
3 9 7 6 0 0 0 1 8 0 6 0 0 0 0 2 8 0 0 0 5 0 4 0 6 0 0 2 1 0
0 0 0 0 4 5', 'gsd.impossible_hints': 0,
'timer.__absolute_start_time__': <float at remote
0x984b474>, 'gsd.hints': 0, 'timer.active_time': <float at
remote 0x984b494>, 'timer.total_time': <float at remote
0x984b464>}], dialog=<gtk.Dialog at remote 0x98faaa4>,
saved_game_model=<gtk.ListStore at remote 0x98fad24>,
sudoku_maker=<SudokuMaker(terminated=False, played=[],
batch_siz...(truncated)
            swallower.run_dialog(self.dialog)
(gdb) py-down
#11 Frame 0x9aead74, for file /usr/lib/python2.6/site-
packages/gnome_sudoku/dialog_swallower.py, line 48, in
run_dialog (self=<SwappableArea(running=<gtk.Dialog at
remote 0x98faaa4>, main_page=0) at remote 0x98fa6e4>, d=
<gtk.Dialog at remote 0x98faaa4>)
            gtk.main()
(gdb) py-down
#8 (unable to read python frame information)
(gdb) py-down
Unable to find a newer python frame
```

and we’re at the bottom of the Python stack.

Note that in Python 3.12 and newer, the same C stack frame can be used for multiple Python stack frames. This means that `py-up` and `py-down` may move multiple Python frames at once. For example:

```
(gdb) py-up
#6 Frame 0x7ffff7fb62b0, for file /tmp/rec.py, line 5, in
recursive_function (n=0)
   time.sleep(5)
#6 Frame 0x7ffff7fb6240, for file /tmp/rec.py, line 7, in
```

```
recursive_function (n=1)
   recursive_function(n-1)
#6 Frame 0x7ffff7fb61d0, for file /tmp/rec.py, line 7, in
recursive_function (n=2)
   recursive_function(n-1)
#6 Frame 0x7ffff7fb6160, for file /tmp/rec.py, line 7, in
recursive_function (n=3)
   recursive_function(n-1)
#6 Frame 0x7ffff7fb60f0, for file /tmp/rec.py, line 7, in
recursive_function (n=4)
   recursive_function(n-1)
#6 Frame 0x7ffff7fb6080, for file /tmp/rec.py, line 7, in
recursive_function (n=5)
   recursive_function(n-1)
#6 Frame 0x7ffff7fb6020, for file /tmp/rec.py, line 9, in
<module> ()
   recursive_function(5)
(gdb) py-up
Unable to find an older python frame
```

py-bt

The `py-bt` command attempts to display a Python-level backtrace of the current thread.

For example:

```
(gdb) py-bt
#8 (unable to read python frame information)
#11 Frame 0x9aead74, for file /usr/lib/python2.6/site-
packages/gnome_sudoku/dialog_swallower.py, line 48, in
run_dialog (self=<SwappableArea(running=<gtk.Dialog at
remote 0x98faaa4>, main_page=0) at remote 0x98fa6e4>, d=
<gtk.Dialog at remote 0x98faaa4>)
            gtk.main()
#14 Frame 0x99262ac, for file /usr/lib/python2.6/site-
packages/gnome_sudoku/game_selector.py, line 201, in
run_swallowed_dialog (self=
<NewOrSavedGameSelector(new_game_model=<gtk.ListStore at
remote 0x98fab44>, puzzle=None, saved_games=
[{'gsd.auto_fills': 0, 'tracking': {}, 'trackers': {},
'notes': [], 'saved_at': 1270084485, 'game': '7 8 0 0 0 0 0
5 6 0 0 9 0 8 0 1 0 0 0 4 6 0 0 0 0 7 0 6 5 0 0 0 4 7 9 2 0
0 0 9 0 1 0 0 0 3 9 7 6 0 0 0 1 8 0 6 0 0 0 0 2 8 0 0 0 5 0
4 0 6 0 0 2 1 0 0 0 0 0 4 5\n7 8 0 0 0 0 0 5 6 0 0 9 0 8 0
1 0 0 0 4 6 0 0 0 0 7 0 6 5 1 8 3 4 7 9 2 0 0 0 9 0 1 0 0 0
3 9 7 6 0 0 0 1 8 0 6 0 0 0 0 2 8 0 0 0 5 0 4 0 6 0 0 2 1 0
0 0 0 0 4 5', 'gsd.impossible_hints': 0,
'timer.__absolute_start_time__': <float at remote
0x984b474>, 'gsd.hints': 0, 'timer.active_time': <float at
remote 0x984b494>, 'timer.total_time': <float at remote
0x984b464>}], dialog=<gtk.Dialog at remote 0x98faaa4>,
saved_game_model=<gtk.ListStore at remote 0x98fad24>,
sudoku_maker=<SudokuMaker(terminated=False, played=[],
batch_siz...(truncated)
            swallower.run_dialog(self.dialog)
#19 (unable to read python frame information)
#23 (unable to read python frame information)
#34 (unable to read python frame information)
#37 Frame 0x9420b04, for file /usr/lib/python2.6/site-
packages/gnome_sudoku/main.py, line 906, in start_game ()
    u = UI()
#40 Frame 0x948e82c, for file /usr/lib/python2.6/site-
packages/gnome_sudoku/gnome_sudoku.py, line 22, in
start_game (main=<module at remote 0xb771b7f4>)
    main.start_game()
```

The frame numbers correspond to those displayed by GDB’s standard `backtrace` command.

py-print

The `py-print` command looks up a Python name and tries to print it. It looks in locals within the current thread, then globals, then finally builtins:

```
(gdb) py-print self
local 'self' = <SwappableArea(running=<gtk.Dialog at remote
0x98faaa4>,
main_page=0) at remote 0x98fa6e4>
(gdb) py-print __name__
global '__name__' = 'gnome_sudoku.dialog_swallower'
(gdb) py-print len
```

```
builtin 'len' = <built-in function len>
(gdb) py-print scarlet_pimpernel
'scarlet_pimpernel' not found
```

If the current C frame corresponds to multiple Python frames, `py-print` only considers the first one.

`py-locals`

The `py-locals` command looks up all Python locals within the current Python frame in the selected thread, and prints their representations:

```
(gdb) py-locals
self = <SwappableArea(running=<gtk.Dialog at remote
0x98faaa4>,
main_page=0) at remote 0x98fa6e4>
d = <gtk.Dialog at remote 0x98faaa4>
```

If the current C frame corresponds to multiple Python frames, locals from all of them will be shown:

```
(gdb) py-locals
Locals for recursive_function
n = 0
Locals for recursive_function
n = 1
Locals for recursive_function
n = 2
Locals for recursive_function
n = 3
Locals for recursive_function
n = 4
Locals for recursive_function
n = 5
Locals for <module>
```

Use with GDB commands

The extension commands complement GDB's built-in commands. For example, you can use a frame numbers shown by `py-bt` with the `frame` command to go a specific frame within the selected thread, like this:

```
(gdb) py-bt
(output snipped)
#68 Frame 0xaa4560, for file Lib/test/regrtest.py, line 1548,
in <module> ()
        main()
(gdb) frame 68
#68 0x00000000004cd1e6 in PyEval_EvalFrameEx (f=Frame 0xaa4560,
for file Lib/test/regrtest.py, line 1548, in <module> (),
throwflag=0) at Python/ceval.c:2665
2665                            x = call_function(&sp, oparg);
(gdb) py-list
1543        # Run the tests in a context manager that temporary
changes the CWD to a
1544        # temporary and writable directory. If it's not
possible to create or
1545        # change the CWD, the original CWD will be used.
The original CWD is
1546        # available from test_support.SAVEDCWD.
1547        with test_support.temp_cwd(TESTCWD, quiet=True):
>1548            main()
```

The `info threads` command will give you a list of the threads within the process, and you can use the `thread` command to select a different one:

```
(gdb) info threads
  105 Thread 0x7fffefa18710 (LWP 10260)  sem_wait () at
../nptl/sysdeps/unix/sysv/linux/x86_64/sem_wait.S:86
  104 Thread 0x7fffdf5fe710 (LWP 10259)  sem_wait () at
../nptl/sysdeps/unix/sysv/linux/x86_64/sem_wait.S:86
* 1 Thread 0x7ffff7fe2700 (LWP 10145)  0x00000038e46d73e3 in
select () at ../sysdeps/unix/syscall-template.S:82
```

You can use `thread apply all COMMAND` or (`t a a COMMAND` for short) to run a command on all threads. With `py-bt`, this lets you see what every thread is doing at the Python level:

```
(gdb) t a a py-bt

Thread 105 (Thread 0x7fffefa18710 (LWP 10260)):
#5 Frame 0x7fffd00019d0, for file /home/david/coding/python-
svn/Lib/threading.py, line 155, in _acquire_restore (self=
<_RLock(_Verbose__verbose=False, _RLock__owner=140737354016512,
_RLock__block=<thread.lock at remote 0x858770>,
_RLock__count=1) at remote 0xd7ff40>, count_owner=(1,
140737213728528), count=1, owner=140737213728528)
        self.__block.acquire()
#8 Frame 0x7fffac001640, for file /home/david/coding/python-
svn/Lib/threading.py, line 269, in wait (self=
<_Condition(_Condition__lock=<_RLock(_Verbose__verbose=False,
_RLock__owner=140737354016512, _RLock__block=<thread.lock at
remote 0x858770>, _RLock__count=1) at remote 0xd7ff40>,
acquire=<instancemethod at remote 0xd80260>, _is_owned=
<instancemethod at remote 0xd80160>, _release_save=
<instancemethod at remote 0xd803e0>, release=<instancemethod at
remote 0xd802e0>, _acquire_restore=<instancemethod at remote
0xd7ee60>, _Verbose__verbose=False, _Condition__waiters=[]) at
remote 0xd7fd10>, timeout=None, waiter=<thread.lock at remote
0x858a90>, saved_state=(1, 140737213728528))
            self._acquire_restore(saved_state)
#12 Frame 0x7fffb8001a10, for file /home/david/coding/python-
svn/Lib/test/lock_tests.py, line 348, in f ()
            cond.wait()
#16 Frame 0x7fffb8001c40, for file /home/david/coding/python-
svn/Lib/test/lock_tests.py, line 37, in task
(tid=140737213728528)
                f()

Thread 104 (Thread 0x7fffdf5fe710 (LWP 10259)):
#5 Frame 0x7fffe4001580, for file /home/david/coding/python-
svn/Lib/threading.py, line 155, in _acquire_restore (self=
<_RLock(_Verbose__verbose=False, _RLock__owner=140737354016512,
_RLock__block=<thread.lock at remote 0x858770>,
_RLock__count=1) at remote 0xd7ff40>, count_owner=(1,
140736940992272), count=1, owner=140736940992272)
        self.__block.acquire()
#8 Frame 0x7fffc8002090, for file /home/david/coding/python-
svn/Lib/threading.py, line 269, in wait (self=
<_Condition(_Condition__lock=<_RLock(_Verbose__verbose=False,
_RLock__owner=140737354016512, _RLock__block=<thread.lock at
```

```
remote 0x858770>, _RLock__count=1) at remote 0xd7ff40>,
acquire=<instancemethod at remote 0xd80260>, _is_owned=
<instancemethod at remote 0xd80160>, _release_save=
<instancemethod at remote 0xd803e0>, release=<instancemethod at
remote 0xd802e0>, _acquire_restore=<instancemethod at remote
0xd7ee60>, _Verbose__verbose=False, _Condition__waiters=[]) at
remote 0xd7fd10>, timeout=None, waiter=<thread.lock at remote
0x858860>, saved_state=(1, 140736940992272))
            self._acquire_restore(saved_state)
#12 Frame 0x7fffac001c90, for file /home/david/coding/python-
svn/Lib/test/lock_tests.py, line 348, in f ()
            cond.wait()
#16 Frame 0x7fffac0011c0, for file /home/david/coding/python-
svn/Lib/test/lock_tests.py, line 37, in task
(tid=140736940992272)
                f()

Thread 1 (Thread 0x7ffff7fe2700 (LWP 10145)):
#5 Frame 0xcb5380, for file /home/david/coding/python-
svn/Lib/test/lock_tests.py, line 16, in _wait ()
    time.sleep(0.01)
#8 Frame 0x7fffd00024a0, for file /home/david/coding/python-
svn/Lib/test/lock_tests.py, line 378, in _check_notify (self=
<ConditionTests(_testMethodName='test_notify',
_resultForDoCleanups=<TestResult(_original_stdout=
<cStringIO.StringO at remote 0xc191e0>, skipped=[],
_mirrorOutput=False, testsRun=39, buffer=False,
_original_stderr=<file at remote 0x7ffff7fc6340>,
_stdout_buffer=<cStringIO.StringO at remote 0xc9c7f8>,
_stderr_buffer=<cStringIO.StringO at remote 0xc9c790>,
_moduleSetUpFailed=False, expectedFailures=[], errors=[],
_previousTestClass=<type at remote 0x928310>,
unexpectedSuccesses=[], failures=[], shouldStop=False,
failfast=False) at remote 0xc185a0>, _threads=(0,), _cleanups=
[], _type_equality_funcs={<type at remote 0x7eba00>:
<instancemethod at remote 0xd750e0>, <type at remote 0x7e7820>:
<instancemethod at remote 0xd75160>, <type at remote 0x7e30e0>:
<instancemethod at remote 0xd75060>, <type at remote 0x7e7d20>:
<instancemethod at remote 0xd751e0>, <type at remote
0x7f19e0...(truncated)
        _wait()
```

Enum HOWTO

An `Enum` is a set of symbolic names bound to unique values. They are similar to global variables, but they offer a more useful `repr()`, grouping, type-safety, and a few other features.

They are most useful when you have a variable that can take one of a limited selection of values. For example, the days of the week:

```
>>> from enum import Enum
>>> class Weekday(Enum):
...     MONDAY = 1
...     TUESDAY = 2
...     WEDNESDAY = 3
...     THURSDAY = 4
...     FRIDAY = 5
...     SATURDAY = 6
...     SUNDAY = 7
```

Or perhaps the RGB primary colors:

```
>>> from enum import Enum
>>> class Color(Enum):
...     RED = 1
...     GREEN = 2
...     BLUE = 3
```

As you can see, creating an `Enum` is as simple as writing a class that inherits from `Enum` itself.

Note

Case of Enum Members

Because Enums are used to represent constants, and to help avoid issues with name clashes between mixin-class methods/attributes and enum names, we strongly recommend using UPPER_CASE names for members, and will be using that style in our examples.

Depending on the nature of the enum a member’s value may or may not be important, but either way that value can be used to get the corresponding member:

```
>>> Weekday(3)
<Weekday.WEDNESDAY: 3>
```

As you can see, the `repr()` of a member shows the enum name, the member name, and the value. The `str()` of a member shows only the enum name and member name:

```
>>> print(Weekday.THURSDAY)
Weekday.THURSDAY
```

The *type* of an enumeration member is the enum it belongs to:

```
>>> type(Weekday.MONDAY)
<enum 'Weekday'>
>>> isinstance(Weekday.FRIDAY, Weekday)
True
```

Enum members have an attribute that contains just their **`name`**:

```
>>> print(Weekday.TUESDAY.name)
TUESDAY
```

Likewise, they have an attribute for their **`value`**:

```
>>> Weekday.WEDNESDAY.value
3
```

Unlike many languages that treat enumerations solely as name/value pairs, Python Enums can have behavior added. For example, `datetime.date` has two methods for returning the weekday: `weekday()` and `isoweekday()`. The difference is that one of them counts from 0-6 and the other from 1-7. Rather than keep track of that ourselves we can add a method to the **`Weekday`** enum to extract the day from the `date` instance and return the matching enum member:

```
@classmethod
def from_date(cls, date):
    return cls(date.isoweekday())
```

The complete **`Weekday`** enum now looks like this:

```
>>> class Weekday(Enum):
...     MONDAY = 1
...     TUESDAY = 2
...     WEDNESDAY = 3
...     THURSDAY = 4
...     FRIDAY = 5
...     SATURDAY = 6
...     SUNDAY = 7
...     #
...     @classmethod
...     def from_date(cls, date):
...         return cls(date.isoweekday())
```

Now we can find out what today is! Observe:

```
>>> from datetime import date
>>> Weekday.from_date(date.today())
<Weekday.TUESDAY: 2>
```

Of course, if you’re reading this on some other day, you’ll see that day instead.

This **`Weekday`** enum is great if our variable only needs one day, but what if we need several? Maybe we’re writing a function to plot chores during a week, and don’t want to use a `list` – we could use a different type of `Enum`:

```
>>> from enum import Flag
>>> class Weekday(Flag):
...     MONDAY = 1
...     TUESDAY = 2
...     WEDNESDAY = 4
...     THURSDAY = 8
...     FRIDAY = 16
...     SATURDAY = 32
...     SUNDAY = 64
```

We’ve changed two things: we’re inherited from `Flag`, and the values are all powers of 2.

Just like the original **`Weekday`** enum above, we can have a single selection:

```
>>> first_week_day = Weekday.MONDAY
>>> first_week_day
<Weekday.MONDAY: 1>
```

But `Flag` also allows us to combine several members into a single variable:

```
>>> weekend = Weekday.SATURDAY | Weekday.SUNDAY
>>> weekend
<Weekday.SATURDAY|SUNDAY: 96>
```

You can even iterate over a `Flag` variable:

```
>>> for day in weekend:
...     print(day)
Weekday.SATURDAY
Weekday.SUNDAY
```

Okay, let’s get some chores set up:

```
>>> chores_for_ethan = {
...     'feed the cat': Weekday.MONDAY | Weekday.WEDNESDAY |
Weekday.FRIDAY,
...     'do the dishes': Weekday.TUESDAY | Weekday.THURSDAY,
...     'answer SO questions': Weekday.SATURDAY,
...     }
```

And a function to display the chores for a given day:

```
>>> def show_chores(chores, day):
...     for chore, days in chores.items():
...         if day in days:
...             print(chore)
...
>>> show_chores(chores_for_ethan, Weekday.SATURDAY)
answer SO questions
```

In cases where the actual values of the members do not matter, you can save yourself some work and use `auto()` for the values:

```
>>> from enum import auto
>>> class Weekday(Flag):
...     MONDAY = auto()
...     TUESDAY = auto()
...     WEDNESDAY = auto()
...     THURSDAY = auto()
...     FRIDAY = auto()
...     SATURDAY = auto()
...     SUNDAY = auto()
...     WEEKEND = SATURDAY | SUNDAY
```

Programmatic access to enumeration members and their attributes

Sometimes it’s useful to access members in enumerations programmatically (i.e. situations where `Color.RED` won’t do because the exact color is not known at program-writing time). `Enum` allows such access:

```
>>> Color(1)
<Color.RED: 1>
>>> Color(3)
<Color.BLUE: 3>
```

If you want to access enum members by *name*, use item access:

```
>>> Color['RED']
<Color.RED: 1>
>>> Color['GREEN']
<Color.GREEN: 2>
```

If you have an enum member and need its **`name`** or **`value`**:

```
>>> member = Color.RED
>>> member.name
'RED'
>>> member.value
1
```

Duplicating enum members and values

Having two enum members with the same name is invalid:

```
>>> class Shape(Enum):
...     SQUARE = 2
...     SQUARE = 3
...
Traceback (most recent call last):
...
TypeError: 'SQUARE' already defined as 2
```

However, an enum member can have other names associated with it. Given two entries `A` and `B` with the same value (and `A` defined first), `B` is an alias for the member `A`. By-value lookup of the value of `A` will return the member `A`.

By-name lookup of `A` will return the member `A`. By-name lookup of `B` will also return the member `A`:

```
>>> class Shape(Enum):
...     SQUARE = 2
...     DIAMOND = 1
...     CIRCLE = 3
...     ALIAS_FOR_SQUARE = 2
...
>>> Shape.SQUARE
<Shape.SQUARE: 2>
>>> Shape.ALIAS_FOR_SQUARE
<Shape.SQUARE: 2>
>>> Shape(2)
<Shape.SQUARE: 2>
```

Note

Attempting to create a member with the same name as an already defined attribute (another member, a method, etc.) or attempting to create an attribute with the same name as a member is not allowed.

Ensuring unique enumeration values

By default, enumerations allow multiple names as aliases for the same value. When this behavior isn’t desired, you can use the `unique()` decorator:

```
>>> from enum import Enum, unique
>>> @unique
... class Mistake(Enum):
...     ONE = 1
...     TWO = 2
...     THREE = 3
...     FOUR = 3
...
Traceback (most recent call last):
```

```
...
ValueError: duplicate values found in <enum 'Mistake'>: FOUR ->
THREE
```

Using automatic values

If the exact value is unimportant you can use `auto`:

```
>>> from enum import Enum, auto
>>> class Color(Enum):
...     RED = auto()
...     BLUE = auto()
...     GREEN = auto()
...
>>> [member.value for member in Color]
[1, 2, 3]
```

The values are chosen by `_generate_next_value_()`, which can be overridden:

```
>>> class AutoName(Enum):
...     @staticmethod
...     def _generate_next_value_(name, start, count,
last_values):
...         return name
...
>>> class Ordinal(AutoName):
...     NORTH = auto()
...     SOUTH = auto()
...     EAST = auto()
...     WEST = auto()
...
>>> [member.value for member in Ordinal]
['NORTH', 'SOUTH', 'EAST', 'WEST']
```

Note

The `_generate_next_value_()` method must be defined before any members.

Iteration

Iterating over the members of an enum does not provide the aliases:

```
>>> list(Shape)
[<Shape.SQUARE: 2>, <Shape.DIAMOND: 1>, <Shape.CIRCLE: 3>]
>>> list(Weekday)
[<Weekday.MONDAY: 1>, <Weekday.TUESDAY: 2>, <Weekday.WEDNESDAY:
4>, <Weekday.THURSDAY: 8>, <Weekday.FRIDAY: 16>,
<Weekday.SATURDAY: 32>, <Weekday.SUNDAY: 64>]
```

Note that the aliases `Shape.ALIAS_FOR_SQUARE` and `Weekday.WEEKEND` aren't shown.

The special attribute `__members__` is a read-only ordered mapping of names to members. It includes all names defined in the enumeration, including the aliases:

```
>>> for name, member in Shape.__members__.items():
...     name, member
...
('SQUARE', <Shape.SQUARE: 2>)
('DIAMOND', <Shape.DIAMOND: 1>)
('CIRCLE', <Shape.CIRCLE: 3>)
('ALIAS_FOR_SQUARE', <Shape.SQUARE: 2>)
```

The `__members__` attribute can be used for detailed programmatic access to the enumeration members. For example, finding all the aliases:

```
>>> [name for name, member in Shape.__members__.items() if
member.name != name]
['ALIAS_FOR_SQUARE']
```

Note

Aliases for flags include values with multiple flags set, such as `3`, and no flags set, i.e. `0`.

Comparisons

Enumeration members are compared by identity:

```
>>> Color.RED is Color.RED
True
>>> Color.RED is Color.BLUE
False
>>> Color.RED is not Color.BLUE
True
```

Ordered comparisons between enumeration values are *not* supported. Enum members are not integers (but see IntEnum below):

```
>>> Color.RED < Color.BLUE
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: '<' not supported between instances of 'Color' and
'Color'
```

Equality comparisons are defined though:

```
>>> Color.BLUE == Color.RED
False
>>> Color.BLUE != Color.RED
True
>>> Color.BLUE == Color.BLUE
True
```

Comparisons against non-enumeration values will always compare not equal (again, **`IntEnum`** was explicitly designed to behave differently, see below):

```
>>> Color.BLUE == 2
False
```

Warning

It is possible to reload modules – if a reloaded module contains enums, they will be recreated, and the new members may not compare identical/equal to the original members.

Allowed members and attributes of enumerations

Most of the examples above use integers for enumeration values. Using integers is short and handy (and provided by default by the Functional API), but not strictly enforced. In the vast majority of use-cases, one doesn't care what the actual value of an enumeration is. But if the value *is* important, enumerations can have arbitrary values.

Enumerations are Python classes, and can have methods and special methods as usual. If we have this enumeration:

```
>>> class Mood(Enum):
...     FUNKY = 1
...     HAPPY = 3
...
...     def describe(self):
...         # self is the member here
...         return self.name, self.value
...
...     def __str__(self):
...         return 'my custom str! {0}'.format(self.value)
...
...     @classmethod
...     def favorite_mood(cls):
...         # cls here is the enumeration
...         return cls.HAPPY
...
```

Then:

```
>>> Mood.favorite_mood()
<Mood.HAPPY: 3>
>>> Mood.HAPPY.describe()
('HAPPY', 3)
>>> str(Mood.FUNKY)
'my custom str! 1'
```

The rules for what is allowed are as follows: names that start and end with a single underscore are reserved by enum and cannot be used; all other attributes defined within an enumeration will become members of this enumeration, with the exception of special methods (`__str__()`, `__add__()`, etc.), descriptors (methods are also descriptors), and variable names listed in `_ignore_`.

Note: if your enumeration defines `__new__()` and/or `__init__()`, any value(s) given to the enum member will be passed into those methods. See Planet for an example.

Note

The `__new__()` method, if defined, is used during creation of the Enum members; it is then replaced by Enum’s `__new__()` which is used after class creation for lookup of existing members. See When to use __new__() vs. __init__() for more details.

Restricted Enum subclassing

A new `Enum` class must have one base enum class, up to one concrete data type, and as many `object`-based mixin classes as needed. The order of these base classes is:

```
class EnumName([mix-in, ...,] [data-type,] base-enum):
    pass
```

Also, subclassing an enumeration is allowed only if the enumeration does not define any members. So this is forbidden:

```
>>> class MoreColor(Color):
...     PINK = 17
...
Traceback (most recent call last):
...
TypeError: <enum 'MoreColor'> cannot extend <enum 'Color'>
```

But this is allowed:

```
>>> class Foo(Enum):
...     def some_behavior(self):
...         pass
...
>>> class Bar(Foo):
...     HAPPY = 1
...     SAD = 2
...
```

Allowing subclassing of enums that define members would lead to a violation of some important invariants of types and instances. On the other hand, it makes sense to allow sharing some common behavior between a group of enumerations. (See OrderedEnum for an example.)

Dataclass support

When inheriting from a **`dataclass`**, the **`__repr__()`** omits the inherited class' name. For example:

```
>>> from dataclasses import dataclass, field
>>> @dataclass
... class CreatureDataMixin:
...     size: str
...     legs: int
...     tail: bool = field(repr=False, default=True)
...
>>> class Creature(CreatureDataMixin, Enum):
...     BEETLE = 'small', 6
...     DOG = 'medium', 4
...
>>> Creature.DOG
<Creature.DOG: size='medium', legs=4>
```

Use the **`dataclass()`** argument `repr=False` to use the standard **`repr()`**.

Changed in version 3.12: Only the dataclass fields are shown in the value area, not the dataclass' name.

Note

Adding **`dataclass()`** decorator to **`Enum`** and its subclasses is not supported. It will not raise any errors, but it will produce very strange results at runtime, such as members being equal to each other:

```
>>> @dataclass               # don't do this: it does not
make any sense
... class Color(Enum):
...    RED = 1
...    BLUE = 2
...
>>> Color.RED is Color.BLUE
False
>>> Color.RED == Color.BLUE  # problem is here: they should
not be equal
True
```

Pickling

Enumerations can be pickled and unpickled:

```
>>> from test.test_enum import Fruit
>>> from pickle import dumps, loads
>>> Fruit.TOMATO is loads(dumps(Fruit.TOMATO))
True
```

The usual restrictions for pickling apply: picklable enums must be defined in the top level of a module, since unpickling requires them to be importable from that module.

Note

With pickle protocol version 4 it is possible to easily pickle enums nested in other classes.

It is possible to modify how enum members are pickled/unpickled by defining `__reduce_ex__()` in the enumeration class. The default method is by-value, but enums with complicated values may want to use by-name:

```
>>> import enum
>>> class MyEnum(enum.Enum):
...     __reduce_ex__ = enum.pickle_by_enum_name
```

Note

Using by-name for flags is not recommended, as unnamed aliases will not unpickle.

Functional API

The **`Enum`** class is callable, providing the following functional API:

```
>>> Animal = Enum('Animal', 'ANT BEE CAT DOG')
>>> Animal
<enum 'Animal'>
>>> Animal.ANT
<Animal.ANT: 1>
>>> list(Animal)
[<Animal.ANT: 1>, <Animal.BEE: 2>, <Animal.CAT: 3>,
<Animal.DOG: 4>]
```

The semantics of this API resemble **`namedtuple`**. The first argument of the call to **`Enum`** is the name of the enumeration.

The second argument is the *source* of enumeration member names. It can be a whitespace-separated string of names, a sequence of names, a sequence of 2-tuples with key/value pairs, or a mapping (e.g. dictionary) of names to values. The last two options enable assigning arbitrary values to enumerations; the others auto-assign increasing integers starting with 1 (use the `start` parameter to specify a different starting value). A new class derived from **`Enum`** is returned. In other words, the above assignment to **`Animal`** is equivalent to:

```
>>> class Animal(Enum):
...     ANT = 1
...     BEE = 2
...     CAT = 3
...     DOG = 4
...
```

The reason for defaulting to `1` as the starting number and not `0` is that `0` is `False` in a boolean sense, but by default enum members all evaluate to `True`.

Pickling enums created with the functional API can be tricky as frame stack implementation details are used to try and figure out which module the enumeration is being created in (e.g. it will fail if you use a utility function in a separate module, and also may not work on IronPython or Jython). The solution is to specify the module name explicitly as follows:

```
>>> Animal = Enum('Animal', 'ANT BEE CAT DOG', module=__name__)
```

Warning

If `module` is not supplied, and Enum cannot determine what it is, the new Enum members will not be unpicklable; to keep errors closer to the source, pickling will be disabled.

The new pickle protocol 4 also, in some circumstances, relies on `__qualname__` being set to the location where pickle will be able to find the class. For example, if the class was made available in class SomeData in the global scope:

```
>>> Animal = Enum('Animal', 'ANT BEE CAT DOG',
qualname='SomeData.Animal')
```

The complete signature is:

```
Enum(
    value='NewEnumName',
    names=<...>,
    *,
    module='...',
    qualname='...',
    type=<mixed-in class>,
    start=1,
    )
```

- *value*: What the new enum class will record as its name.
- *names*: The enum members. This can be a whitespace- or comma-separated string (values will start at 1 unless

otherwise specified):

`'RED GREEN BLUE' | 'RED,GREEN,BLUE' | 'RED, GREEN, BLUE'`

or an iterator of names:

`['RED', 'GREEN', 'BLUE']`

or an iterator of (name, value) pairs:

`[('CYAN', 4), ('MAGENTA', 5), ('YELLOW', 6)]`

or a mapping:

`{'CHARTREUSE': 7, 'SEA_GREEN': 11, 'ROSEMARY': 42}`

- *module*: name of module where new enum class can be found.
- *qualname*: where in module new enum class can be found.
- *type*: type to mix in to new enum class.
- *start*: number to start counting at if only names are passed in.

Changed in version 3.5: The *start* parameter was added.

Derived Enumerations

IntEnum

The first variation of `Enum` that is provided is also a subclass of `int`. Members of an `IntEnum` can be compared to integers; by extension, integer enumerations of different types can also be compared to each other:

```
>>> from enum import IntEnum
>>> class Shape(IntEnum):
...     CIRCLE = 1
...     SQUARE = 2
...
>>> class Request(IntEnum):
...     POST = 1
...     GET = 2
...
>>> Shape == 1
False
>>> Shape.CIRCLE == 1
True
>>> Shape.CIRCLE == Request.POST
True
```

However, they still can’t be compared to standard `Enum` enumerations:

```
>>> class Shape(IntEnum):
...     CIRCLE = 1
...     SQUARE = 2
...
>>> class Color(Enum):
...     RED = 1
...     GREEN = 2
...
>>> Shape.CIRCLE == Color.RED
False
```

`IntEnum` values behave like integers in other ways you’d expect:

```
>>> int(Shape.CIRCLE)
1
>>> ['a', 'b', 'c'][Shape.CIRCLE]
'b'
>>> [i for i in range(Shape.SQUARE)]
[0, 1]
```

StrEnum

The second variation of `Enum` that is provided is also a subclass of `str`. Members of a `StrEnum` can be compared to strings; by extension, string enumerations of different types can also be compared to each other.

Added in version 3.11.

IntFlag

The next variation of `Enum` provided, `IntFlag`, is also based on `int`. The difference being `IntFlag` members can be combined using the bitwise operators (&, |, ^, ~) and the result is still an `IntFlag` member, if possible. Like `IntEnum`, `IntFlag` members are also integers and can be used wherever an `int` is used.

Note

Any operation on an `IntFlag` member besides the bit-wise operations will lose the `IntFlag` membership.

Bit-wise operations that result in invalid `IntFlag` values will lose the `IntFlag` membership. See `FlagBoundary` for details.

Added in version 3.6.

Changed in version 3.11.

Sample `IntFlag` class:

```
>>> from enum import IntFlag
>>> class Perm(IntFlag):
...     R = 4
...     W = 2
...     X = 1
...
>>> Perm.R | Perm.W
```

```
<Perm.R|W: 6>
>>> Perm.R + Perm.W
6
>>> RW = Perm.R | Perm.W
>>> Perm.R in RW
True
```

It is also possible to name the combinations:

```
>>> class Perm(IntFlag):
...     R = 4
...     W = 2
...     X = 1
...     RWX = 7
...
>>> Perm.RWX
<Perm.RWX: 7>
>>> ~Perm.RWX
<Perm: 0>
>>> Perm(7)
<Perm.RWX: 7>
```

Note

Named combinations are considered aliases. Aliases do not show up during iteration, but can be returned from by-value lookups.

Changed in version 3.11.

Another important difference between `IntFlag` and `Enum` is that if no flags are set (the value is 0), its boolean evaluation is `False`:

```
>>> Perm.R & Perm.X
<Perm: 0>
>>> bool(Perm.R & Perm.X)
False
```

Because `IntFlag` members are also subclasses of `int` they can be combined with them (but may lose `IntFlag` membership:

```
>>> Perm.X | 4
<Perm.R|X: 5>

>>> Perm.X + 8
9
```

Note

The negation operator, ~, always returns an `IntFlag` member with a positive value:

```
>>> (~Perm.X).value == (Perm.R|Perm.W).value == 6
True
```

`IntFlag` members can also be iterated over:

```
>>> list(RW)
[<Perm.R: 4>, <Perm.W: 2>]
```

Added in version 3.11.

Flag

The last variation is `Flag`. Like `IntFlag`, `Flag` members can be combined using the bitwise operators (&, |, ^, ~). Unlike `IntFlag`, they cannot be combined with, nor compared against, any other `Flag` enumeration, nor `int`. While it is possible to specify the values directly it is recommended to use `auto` as the value and let `Flag` select an appropriate value.

Added in version 3.6.

Like `IntFlag`, if a combination of `Flag` members results in no flags being set, the boolean evaluation is `False`:

```
>>> from enum import Flag, auto
>>> class Color(Flag):
...     RED = auto()
...     BLUE = auto()
...     GREEN = auto()
...
>>> Color.RED & Color.GREEN
<Color: 0>
>>> bool(Color.RED & Color.GREEN)
False
```

Individual flags should have values that are powers of two (1, 2, 4, 8, …), while combinations of flags will not:

```
>>> class Color(Flag):
...     RED = auto()
...     BLUE = auto()
...     GREEN = auto()
...     WHITE = RED | BLUE | GREEN
...
>>> Color.WHITE
<Color.WHITE: 7>
```

Giving a name to the “no flags set” condition does not change its boolean value:

```
>>> class Color(Flag):
...     BLACK = 0
...     RED = auto()
...     BLUE = auto()
...     GREEN = auto()
...
>>> Color.BLACK
<Color.BLACK: 0>
>>> bool(Color.BLACK)
False
```

`Flag` members can also be iterated over:

```
>>> purple = Color.RED | Color.BLUE
>>> list(purple)
[<Color.RED: 1>, <Color.BLUE: 2>]
```

Added in version 3.11.

Note

For the majority of new code, `Enum` and `Flag` are strongly recommended, since `IntEnum` and `IntFlag` break some semantic promises of an enumeration (by being comparable to integers, and thus by transitivity to other unrelated enumerations). `IntEnum` and `IntFlag` should be used only in cases where `Enum` and `Flag` will not do; for example, when integer constants are replaced with enumerations, or for interoperability with other systems.

Others

While `IntEnum` is part of the `enum` module, it would be very simple to implement independently:

```
class IntEnum(int, ReprEnum):   # or Enum instead of ReprEnum
    pass
```

This demonstrates how similar derived enumerations can be defined; for example a **`FloatEnum`** that mixes in `float` instead of `int`.

Some rules:

1. When subclassing `Enum`, mix-in types must appear before the `Enum` class itself in the sequence of bases, as in the `IntEnum` example above.
2. Mix-in types must be subclassable. For example, `bool` and `range` are not subclassable and will throw an error

during Enum creation if used as the mix-in type.

3. While `Enum` can have members of any type, once you mix in an additional type, all the members must have values of that type, e.g. `int` above. This restriction does not apply to mix-ins which only add methods and don't specify another type.
4. When another data type is mixed in, the `value` attribute is *not the same* as the enum member itself, although it is equivalent and will compare equal.
5. A `data type` is a mixin that defines `__new__()`, or a `dataclass`
6. %-style formatting: `%s` and `%r` call the `Enum` class's `__str__()` and `__repr__()` respectively; other codes (such as `%i` or `%h` for IntEnum) treat the enum member as its mixed-in type.
7. Formatted string literals, `str.format()`, and `format()` will use the enum's `__str__()` method.

Note

Because `IntEnum`, `IntFlag`, and `StrEnum` are designed to be drop-in replacements for existing constants, their `__str__()` method has been reset to their data types' `__str__()` method.

When to use `__new__()` vs. `__init__()`

`__new__()` must be used whenever you want to customize the actual value of the `Enum` member. Any other modifications may go in either `__new__()` or `__init__()`, with `__init__()` being preferred.

For example, if you want to pass several items to the constructor, but only want one of them to be the value:

```
>>> class Coordinate(bytes, Enum):
...     """
...     Coordinate with binary codes that can be indexed by the
int code.
...     """
...     def __new__(cls, value, label, unit):
...         obj = bytes.__new__(cls, [value])
...         obj._value_ = value
...         obj.label = label
...         obj.unit = unit
...         return obj
...     PX = (0, 'P.X', 'km')
...     PY = (1, 'P.Y', 'km')
...     VX = (2, 'V.X', 'km/s')
...     VY = (3, 'V.Y', 'km/s')
...

>>> print(Coordinate['PY'])
Coordinate.PY

>>> print(Coordinate(3))
Coordinate.VY
```

Warning

Do not call `super().__new__()`, as the lookup-only `__new__` is the one that is found; instead, use the data type directly.

Finer Points

Supported `__dunder__` names

`__members__` is a read-only ordered mapping of `member_name`:`member` items. It is only available on the class.

`__new__()`, if specified, must create and return the enum members; it is also a very good idea to set the member’s `_value_` appropriately. Once all the members are created it is no longer used.

Supported `_sunder_` names

- `_name_` – name of the member
- `_value_` – value of the member; can be set in `__new__`
- `_missing_()` – a lookup function used when a value is not found; may be overridden
- `_ignore_` – a list of names, either as a `list` or a `str`, that will not be transformed into members, and will be removed from the final class
- `_generate_next_value_()` – used to get an appropriate value for an enum member; may be overridden
- `_add_alias_()` – adds a new name as an alias to an existing member.
- `_add_value_alias_()` – adds a new value as an alias to an existing member. See MultiValueEnum for an example.

  Note

  For standard `Enum` classes the next value chosen is the highest value seen incremented by one.

  For `Flag` classes the next value chosen will be the next highest power-of-two.

Changed in version 3.13: Prior versions would use the last seen value instead of the highest value.

Added in version 3.6: `_missing_`, `_order_`, `_generate_next_value_`

Added in version 3.7: `_ignore_`

Added in version 3.13: `_add_alias_`, `_add_value_alias_`

To help keep Python 2 / Python 3 code in sync an **`_order_`** attribute can be provided. It will be checked against the actual order of the enumeration and raise an error if the two do not match:

```
>>> class Color(Enum):
...     _order_ = 'RED GREEN BLUE'
...     RED = 1
...     BLUE = 3
...     GREEN = 2
...
Traceback (most recent call last):
...
TypeError: member order does not match _order_:
  ['RED', 'BLUE', 'GREEN']
  ['RED', 'GREEN', 'BLUE']
```

Note

In Python 2 code the **`_order_`** attribute is necessary as definition order is lost before it can be recorded.

_Private__names

Private names are not converted to enum members, but remain normal attributes.

Changed in version 3.11.

`Enum` member type

Enum members are instances of their enum class, and are normally accessed as `EnumClass.member`. In certain situations, such as writing custom enum behavior, being able to access one member directly from another is useful, and is supported; however, in order to avoid name clashes between member names and attributes/methods from mixed-in classes, upper-case names are strongly recommended.

Changed in version 3.5.

Creating members that are mixed with other data types

When subclassing other data types, such as **`int`** or **`str`**, with an **`Enum`**, all values after the `=` are passed to that data type's constructor. For example:

```
>>> class MyEnum(IntEnum):      # help(int) -> int(x, base=10)
-> integer
...     example = '11', 16      # so x='11' and base=16
...
>>> MyEnum.example.value        # and hex(11) is...
17
```

Boolean value of `Enum` classes and members

Enum classes that are mixed with non-**`Enum`** types (such as **`int`**, **`str`**, etc.) are evaluated according to the mixed-in type's rules; otherwise, all members evaluate as **`True`**. To make your own enum's boolean evaluation depend on the member's value add the following to your class:

```
def __bool__(self):
    return bool(self.value)
```

Plain **`Enum`** classes always evaluate as **`True`**.

`Enum` classes with methods

If you give your enum subclass extra methods, like the Planet class below, those methods will show up in a **`dir()`** of the member, but not of the class:

```
>>> dir(Planet)
['EARTH', 'JUPITER', 'MARS', 'MERCURY', 'NEPTUNE', 'SATURN',
'URANUS', 'VENUS', '__class__', '__doc__', '__members__',
'__module__']
>>> dir(Planet.EARTH)
['__class__', '__doc__', '__module__', 'mass', 'name',
'radius', 'surface_gravity', 'value']
```

Combining members of `Flag`

Iterating over a combination of **`Flag`** members will only return the members that are comprised of a single bit:

```
>>> class Color(Flag):
...     RED = auto()
...     GREEN = auto()
...     BLUE = auto()
...     MAGENTA = RED | BLUE
...     YELLOW = RED | GREEN
...     CYAN = GREEN | BLUE
...
>>> Color(3)  # named combination
<Color.YELLOW: 3>
>>> Color(7)      # not named combination
<Color.RED|GREEN|BLUE: 7>
```

`Flag` and `IntFlag` minutia

Using the following snippet for our examples:

```
>>> class Color(IntFlag):
...     BLACK = 0
...     RED = 1
...     GREEN = 2
...     BLUE = 4
...     PURPLE = RED | BLUE
...     WHITE = RED | GREEN | BLUE
...
```

the following are true:

- single-bit flags are canonical
- multi-bit and zero-bit flags are aliases
- only canonical flags are returned during iteration:

  ```
  >>> list(Color.WHITE)
  [<Color.RED: 1>, <Color.GREEN: 2>, <Color.BLUE: 4>]
  ```

- negating a flag or flag set returns a new flag/flag set with the corresponding positive integer value:

  ```
  >>> Color.BLUE
  <Color.BLUE: 4>

  >>> ~Color.BLUE
  <Color.RED|GREEN: 3>
  ```

- names of pseudo-flags are constructed from their members' names:

  ```
  >>> (Color.RED | Color.GREEN).name
  'RED|GREEN'

  >>> class Perm(IntFlag):
  ...     R = 4
  ...     W = 2
  ...     X = 1
  ...
  >>> (Perm.R & Perm.W).name is None  # effectively Perm(0)
  True
  ```

- multi-bit flags, aka aliases, can be returned from operations:

  ```
  >>> Color.RED | Color.BLUE
  <Color.PURPLE: 5>

  >>> Color(7)  # or Color(-1)
  <Color.WHITE: 7>

  >>> Color(0)
  <Color.BLACK: 0>
  ```

- membership / containment checking: zero-valued flags are always considered to be contained:

  ```
  >>> Color.BLACK in Color.WHITE
  True
  ```

  otherwise, only if all bits of one flag are in the other flag will True be returned:

  ```
  >>> Color.PURPLE in Color.WHITE
  True

  >>> Color.GREEN in Color.PURPLE
  False
  ```

There is a new boundary mechanism that controls how out-of-range / invalid bits are handled: `STRICT`, `CONFORM`, `EJECT`, and `KEEP`:

- STRICT –> raises an exception when presented with invalid values
- CONFORM –> discards any invalid bits
- EJECT –> lose Flag status and become a normal int with the given value
- KEEP –> keep the extra bits
  - keeps Flag status and extra bits
  - extra bits do not show up in iteration

  - extra bits do show up in repr() and str()

The default for Flag is `STRICT`, the default for `IntFlag` is `EJECT`, and the default for `_convert_` is `KEEP` (see `ssl.Options` for an example of when `KEEP` is needed).

How are Enums and Flags different?

Enums have a custom metaclass that affects many aspects of both derived `Enum` classes and their instances (members).

Enum Classes

The `EnumType` metaclass is responsible for providing the `__contains__()`, `__dir__()`, `__iter__()` and other methods that allow one to do things with an `Enum` class that fail on a typical class, such as `list(Color)` or `some_enum_var in Color`. `EnumType` is responsible for ensuring that various other methods on the final `Enum` class are correct (such as `__new__()`, `__getnewargs__()`, `__str__()` and `__repr__()`).

Flag Classes

Flags have an expanded view of aliasing: to be canonical, the value of a flag needs to be a power-of-two value, and not a duplicate name. So, in addition to the `Enum` definition of alias, a flag with no value (a.k.a. `0`) or with more than one power-of-two value (e.g. `3`) is considered an alias.

Enum Members (aka instances)

The most interesting thing about enum members is that they are singletons. `EnumType` creates them all while it is creating the enum class itself, and then puts a custom `__new__()` in place to ensure that no new ones are ever instantiated by returning only the existing member instances.

Flag Members

Flag members can be iterated over just like the `Flag` class, and only the canonical members will be returned. For example:

```
>>> list(Color)
[<Color.RED: 1>, <Color.GREEN: 2>, <Color.BLUE: 4>]
```

(Note that `BLACK`, `PURPLE`, and `WHITE` do not show up.)

Inverting a flag member returns the corresponding positive value, rather than a negative value — for example:

```
>>> ~Color.RED
<Color.GREEN|BLUE: 6>
```

Flag members have a length corresponding to the number of power-of-two values they contain. For example:

```
>>> len(Color.PURPLE)
2
```

Enum Cookbook

While `Enum`, `IntEnum`, `StrEnum`, `Flag`, and `IntFlag` are expected to cover the majority of use-cases, they cannot cover them all. Here are recipes for some different types of enumerations that can be used directly, or as examples for creating one's own.

Omitting values

In many use-cases, one doesn't care what the actual value of an enumeration is. There are several ways to define this type of simple enumeration:

- use instances of `auto` for the value
- use instances of `object` as the value
- use a descriptive string as the value
- use a tuple as the value and a custom `__new__()` to replace the tuple with an `int` value

Using any of these methods signifies to the user that these values are not important, and also enables one to add, remove, or reorder members without having to renumber the remaining members.

Using `auto`

Using `auto` would look like:

```
>>> class Color(Enum):
...     RED = auto()
...     BLUE = auto()
...     GREEN = auto()
...
>>> Color.GREEN
<Color.GREEN: 3>
```

Using `object`

Using `object` would look like:

```
>>> class Color(Enum):
...     RED = object()
...     GREEN = object()
...     BLUE = object()
...
>>> Color.GREEN
<Color.GREEN: <object object at 0x...>>
```

This is also a good example of why you might want to write your own `__repr__()`:

```
>>> class Color(Enum):
...     RED = object()
...     GREEN = object()
...     BLUE = object()
...     def __repr__(self):
...         return "<%s.%s>" % (self.__class__.__name__,
self._name_)
...
>>> Color.GREEN
<Color.GREEN>
```

Using a descriptive string

Using a string as the value would look like:

```
>>> class Color(Enum):
...     RED = 'stop'
...     GREEN = 'go'
...     BLUE = 'too fast!'
...
>>> Color.GREEN
<Color.GREEN: 'go'>
```

Using a custom `__new__()`

Using an auto-numbering `__new__()` would look like:

```
>>> class AutoNumber(Enum):
...     def __new__(cls):
...         value = len(cls.__members__) + 1
...         obj = object.__new__(cls)
...         obj._value_ = value
...         return obj
...
>>> class Color(AutoNumber):
...     RED = ()
...     GREEN = ()
...     BLUE = ()
...
>>> Color.GREEN
<Color.GREEN: 2>
```

To make a more general purpose `AutoNumber`, add `*args` to the signature:

```
>>> class AutoNumber(Enum):
...     def __new__(cls, *args):      # this is the only change
from above
...         value = len(cls.__members__) + 1
...         obj = object.__new__(cls)
...         obj._value_ = value
...         return obj
...
```

Then when you inherit from `AutoNumber` you can write your own `__init__` to handle any extra arguments:

```
>>> class Swatch(AutoNumber):
...     def __init__(self, pantone='unknown'):
...         self.pantone = pantone
...     AUBURN = '3497'
...     SEA_GREEN = '1246'
...     BLEACHED_CORAL = () # New color, no Pantone code yet!
...
>>> Swatch.SEA_GREEN
<Swatch.SEA_GREEN: 2>
>>> Swatch.SEA_GREEN.pantone
'1246'
>>> Swatch.BLEACHED_CORAL.pantone
'unknown'
```

Note

The `__new__()` method, if defined, is used during creation of the Enum members; it is then replaced by Enum’s `__new__()`

which is used after class creation for lookup of existing members.

Warning

Do not call `super().__new__()`, as the lookup-only `__new__` is the one that is found; instead, use the data type directly – e.g.:

```
obj = int.__new__(cls, value)
```

OrderedEnum

An ordered enumeration that is not based on **`IntEnum`** and so maintains the normal **`Enum`** invariants (such as not being comparable to other enumerations):

```
>>> class OrderedEnum(Enum):
...     def __ge__(self, other):
...         if self.__class__ is other.__class__:
...             return self.value >= other.value
...         return NotImplemented
...     def __gt__(self, other):
...         if self.__class__ is other.__class__:
...             return self.value > other.value
...         return NotImplemented
...     def __le__(self, other):
...         if self.__class__ is other.__class__:
...             return self.value <= other.value
...         return NotImplemented
...     def __lt__(self, other):
...         if self.__class__ is other.__class__:
...             return self.value < other.value
...         return NotImplemented
...
>>> class Grade(OrderedEnum):
...     A = 5
...     B = 4
...     C = 3
...     D = 2
...     F = 1
...
>>> Grade.C < Grade.A
True
```

DuplicateFreeEnum

Raises an error if a duplicate member value is found instead of creating an alias:

```
>>> class DuplicateFreeEnum(Enum):
...     def __init__(self, *args):
...         cls = self.__class__
...         if any(self.value == e.value for e in cls):
...             a = self.name
...             e = cls(self.value).name
...             raise ValueError(
...                 "aliases not allowed in DuplicateFreeEnum:
%r --> %r"
...                 % (a, e))
...
>>> class Color(DuplicateFreeEnum):
...     RED = 1
...     GREEN = 2
...     BLUE = 3
...     GRENE = 2
...
Traceback (most recent call last):
  ...
ValueError: aliases not allowed in DuplicateFreeEnum:  'GRENE'
--> 'GREEN'
```

Note

This is a useful example for subclassing Enum to add or change other behaviors as well as disallowing aliases. If the only desired change is disallowing aliases, the `unique()` decorator can be used instead.

MultiValueEnum

Supports having more than one value per member:

```
>>> class MultiValueEnum(Enum):
...     def __new__(cls, value, *values):
...         self = object.__new__(cls)
...         self._value_ = value
...         for v in values:
...             self._add_value_alias_(v)
...         return self
...
>>> class DType(MultiValueEnum):
...     float32 = 'f', 8
...     double64 = 'd', 9
...
>>> DType('f')
<DType.float32: 'f'>
>>> DType(9)
<DType.double64: 'd'>
```

Planet

If `__new__()` or `__init__()` is defined, the value of the enum member will be passed to those methods:

```
>>> class Planet(Enum):
...     MERCURY = (3.303e+23, 2.4397e6)
...     VENUS   = (4.869e+24, 6.0518e6)
...     EARTH   = (5.976e+24, 6.37814e6)
...     MARS    = (6.421e+23, 3.3972e6)
...     JUPITER = (1.9e+27,   7.1492e7)
...     SATURN  = (5.688e+26, 6.0268e7)
...     URANUS  = (8.686e+25, 2.5559e7)
...     NEPTUNE = (1.024e+26, 2.4746e7)
...     def __init__(self, mass, radius):
```

```
...         self.mass = mass       # in kilograms
...         self.radius = radius   # in meters
...     @property
...     def surface_gravity(self):
...         # universal gravitational constant  (m3 kg-1 s-2)
...         G = 6.67300E-11
...         return G * self.mass / (self.radius * self.radius)
...
>>> Planet.EARTH.value
(5.976e+24, 6378140.0)
>>> Planet.EARTH.surface_gravity
9.802652743337129
```

TimePeriod

An example to show the `_ignore_` attribute in use:

```
>>> from datetime import timedelta
>>> class Period(timedelta, Enum):
...     "different lengths of time"
...     _ignore_ = 'Period i'
...     Period = vars()
...     for i in range(367):
...         Period['day_%d' % i] = i
...
>>> list(Period)[:2]
[<Period.day_0: datetime.timedelta(0)>, <Period.day_1:
datetime.timedelta(days=1)>]
>>> list(Period)[-2:]
[<Period.day_365: datetime.timedelta(days=365)>,
<Period.day_366: datetime.timedelta(days=366)>]
```

Subclassing EnumType

While most enum needs can be met by customizing `Enum` subclasses, either with class decorators or custom functions, `EnumType` can be subclassed to provide a different Enum experience.

Functional Programming HOWTO

Author:: A. M. Kuchling
Release:: 0.32

In this document, we’ll take a tour of Python’s features suitable for implementing programs in a functional style. After an introduction to the concepts of functional programming, we’ll look at language features such as iterators and generators and relevant library modules such as **`itertools`** and **`functools`**.

Introduction

This section explains the basic concept of functional programming; if you’re just interested in learning about Python language features, skip to the next section on Iterators.

Programming languages support decomposing problems in several different ways:

- Most programming languages are **procedural**: programs are lists of instructions that tell the computer what to do with the program’s input. C, Pascal, and even Unix shells are procedural languages.
- In **declarative** languages, you write a specification that describes the problem to be solved, and the language implementation figures out how to perform the computation efficiently. SQL is the declarative language you’re most likely to be familiar with; a SQL query

describes the data set you want to retrieve, and the SQL engine decides whether to scan tables or use indexes, which subclauses should be performed first, etc.

- **Object-oriented** programs manipulate collections of objects. Objects have internal state and support methods that query or modify this internal state in some way. Smalltalk and Java are object-oriented languages. C++ and Python are languages that support object-oriented programming, but don’t force the use of object-oriented features.
- **Functional** programming decomposes a problem into a set of functions. Ideally, functions only take inputs and produce outputs, and don’t have any internal state that affects the output produced for a given input. Well-known functional languages include the ML family (Standard ML, OCaml, and other variants) and Haskell.

The designers of some computer languages choose to emphasize one particular approach to programming. This often makes it difficult to write programs that use a different approach. Other languages are multi-paradigm languages that support several different approaches. Lisp, C++, and Python are multi-paradigm; you can write programs or libraries that are largely procedural, object-oriented, or functional in all of these languages. In a large program, different sections might be written using different approaches; the GUI might be object-oriented while the processing logic is procedural or functional, for example.

In a functional program, input flows through a set of functions. Each function operates on its input and produces some output. Functional style discourages functions with side effects that modify internal state or make other changes that aren’t visible in the function’s return value. Functions that have no side effects at all are called **purely functional**. Avoiding side effects means not using data structures that get updated as a program runs; every function’s output must only depend on its input.

Some languages are very strict about purity and don’t even have assignment statements such as `a=3` or `c = a + b`, but it’s difficult to avoid all side effects, such as printing to the screen or writing to a disk file. Another example is a call to the `print()` or `time.sleep()` function, neither of which returns a useful value. Both are called only for their side effects of sending some text to the screen or pausing execution for a second.

Python programs written in functional style usually won’t go to the extreme of avoiding all I/O or all assignments; instead, they’ll provide a functional-appearing interface but will use non-functional features internally. For example, the implementation of a function will still use assignments to local variables, but won’t modify global variables or have other side effects.

Functional programming can be considered the opposite of object-oriented programming. Objects are little capsules containing some internal state along with a collection of method calls that let you modify this state, and programs consist of making the right set of state changes. Functional programming wants to avoid state changes as much as possible and works with data flowing between functions. In Python you might combine the two approaches by writing functions that take and return instances representing objects in your application (e-mail messages, transactions, etc.).

Functional design may seem like an odd constraint to work under. Why should you avoid objects and side effects? There are theoretical and practical advantages to the functional style:

- Formal provability.
- Modularity.
- Composability.
- Ease of debugging and testing.

Formal provability

A theoretical benefit is that it’s easier to construct a mathematical proof that a functional program is correct.

For a long time researchers have been interested in finding ways to mathematically prove programs correct. This is different from testing a program on numerous inputs and concluding that its output is usually correct, or reading a program’s source code and concluding that the code looks right; the goal is instead a rigorous proof that a program produces the right result for all possible inputs.

The technique used to prove programs correct is to write down **invariants**, properties of the input data and of the program’s variables that are always true. For each line of code, you then show that if invariants X and Y are true **before** the line is executed, the slightly different invariants X’ and Y’ are true **after** the line is executed. This continues until you reach the end of the program, at which point the invariants should match the desired conditions on the program’s output.

Functional programming’s avoidance of assignments arose because assignments are difficult to handle with this technique; assignments can break invariants that were true before the assignment without producing any new invariants that can be propagated onward.

Unfortunately, proving programs correct is largely impractical and not relevant to Python software. Even trivial programs require proofs that are several pages long; the proof of correctness for a moderately complicated program would be enormous, and few or none of the programs you use daily (the Python interpreter, your XML parser, your web browser) could be proven correct. Even if you wrote down or generated a proof, there would then be the question of verifying the proof; maybe there’s an error in it, and you wrongly believe you’ve proved the program correct.

Modularity

A more practical benefit of functional programming is that it forces you to break apart your problem into small pieces. Programs are more modular as a result. It’s easier to specify and write a small function that does one thing than a large function that performs a complicated transformation. Small functions are also easier to read and to check for errors.

Ease of debugging and testing

Testing and debugging a functional-style program is easier.

Debugging is simplified because functions are generally small and clearly specified. When a program doesn’t work, each function is an interface point where you can check that the data are correct. You can look at the intermediate inputs and outputs to quickly isolate the function that’s responsible for a bug.

Testing is easier because each function is a potential subject for a unit test. Functions don’t depend on system state that needs to be replicated before running a test; instead you only have to synthesize the right input and then check that the output matches expectations.

Composability

As you work on a functional-style program, you'll write a number of functions with varying inputs and outputs. Some of these functions will be unavoidably specialized to a particular application, but others will be useful in a wide variety of programs. For example, a function that takes a directory path and returns all the XML files in the directory, or a function that takes a filename and returns its contents, can be applied to many different situations.

Over time you'll form a personal library of utilities. Often you'll assemble new programs by arranging existing functions in a new configuration and writing a few functions specialized for the current task.

Iterators

I'll start by looking at a Python language feature that's an important foundation for writing functional-style programs: iterators.

An iterator is an object representing a stream of data; this object returns the data one element at a time. A Python iterator must support a method called `__next__()` that takes no arguments and always returns the next element of the stream. If there are no more elements in the stream, `__next__()` must raise the `StopIteration` exception. Iterators don't have to be finite, though; it's perfectly reasonable to write an iterator that produces an infinite stream of data.

The built-in `iter()` function takes an arbitrary object and tries to return an iterator that will return the object's contents or elements, raising `TypeError` if the object doesn't support iteration. Several of Python's built-in data types support iteration, the most common being lists and dictionaries. An object is called iterable if you can get an iterator for it.

You can experiment with the iteration interface manually:

```
>>> L = [1, 2, 3]
>>> it = iter(L)
>>> it
<...iterator object at ...>
>>> it.__next__()  # same as next(it)
1
>>> next(it)
2
>>> next(it)
3
>>> next(it)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
>>>
```

Python expects iterable objects in several different contexts, the most important being the **`for`** statement. In the statement `for X in Y`, Y must be an iterator or some object for which **`iter()`** can create an iterator. These two statements are equivalent:

```
for i in iter(obj):
    print(i)

for i in obj:
    print(i)
```

Iterators can be materialized as lists or tuples by using the **`list()`** or **`tuple()`** constructor functions:

```
>>> L = [1, 2, 3]
>>> iterator = iter(L)
>>> t = tuple(iterator)
>>> t
(1, 2, 3)
```

Sequence unpacking also supports iterators: if you know an iterator will return N elements, you can unpack them into an

N-tuple:

```
>>> L = [1, 2, 3]
>>> iterator = iter(L)
>>> a, b, c = iterator
>>> a, b, c
(1, 2, 3)
```

Built-in functions such as `max()` and `min()` can take a single iterator argument and will return the largest or smallest element. The `"in"` and `"not in"` operators also support iterators: `X in iterator` is true if X is found in the stream returned by the iterator. You'll run into obvious problems if the iterator is infinite; `max()`, `min()` will never return, and if the element X never appears in the stream, the `"in"` and `"not in"` operators won't return either.

Note that you can only go forward in an iterator; there's no way to get the previous element, reset the iterator, or make a copy of it. Iterator objects can optionally provide these additional capabilities, but the iterator protocol only specifies the `__next__()` method. Functions may therefore consume all of the iterator's output, and if you need to do something different with the same stream, you'll have to create a new iterator.

Data Types That Support Iterators

We've already seen how lists and tuples support iterators. In fact, any Python sequence type, such as strings, will automatically support creation of an iterator.

Calling `iter()` on a dictionary returns an iterator that will loop over the dictionary's keys:

```
>>> m = {'Jan': 1, 'Feb': 2, 'Mar': 3, 'Apr': 4, 'May': 5,
'Jun': 6,
...      'Jul': 7, 'Aug': 8, 'Sep': 9, 'Oct': 10, 'Nov': 11,
'Dec': 12}
>>> for key in m:
...     print(key, m[key])
Jan 1
Feb 2
Mar 3
Apr 4
May 5
Jun 6
Jul 7
Aug 8
Sep 9
Oct 10
Nov 11
Dec 12
```

Note that starting with Python 3.7, dictionary iteration order is guaranteed to be the same as the insertion order. In earlier versions, the behaviour was unspecified and could vary between implementations.

Applying `iter()` to a dictionary always loops over the keys, but dictionaries have methods that return other iterators. If you want to iterate over values or key/value pairs, you can explicitly call the `values()` or `items()` methods to get an appropriate iterator.

The `dict()` constructor can accept an iterator that returns a finite stream of `(key, value)` tuples:

```
>>> L = [('Italy', 'Rome'), ('France', 'Paris'), ('US',
'Washington DC')]
>>> dict(iter(L))
{'Italy': 'Rome', 'France': 'Paris', 'US': 'Washington DC'}
```

Files also support iteration by calling the `readline()` method until there are no more lines in the file. This means you can read each line of a file like this:

```
for line in file:
    # do something for each line
    ...
```

Sets can take their contents from an iterable and let you iterate over the set’s elements:

```
>>> S = {2, 3, 5, 7, 11, 13}
>>> for i in S:
...     print(i)
2
3
5
7
11
13
```

Generator expressions and list comprehensions

Two common operations on an iterator’s output are 1) performing some operation for every element, 2) selecting a subset of elements that meet some condition. For example, given a list of strings, you might want to strip off trailing whitespace from each line or extract all the strings containing a given substring.

List comprehensions and generator expressions (short form: “listcomps” and “genexps”) are a concise notation for such operations, borrowed from the functional programming language Haskell (https://www.haskell.org/). You can strip all the whitespace from a stream of strings with the following code:

```
>>> line_list = ['  line 1\n', 'line 2  \n', ' \n', '']

>>> # Generator expression -- returns iterator
>>> stripped_iter = (line.strip() for line in line_list)
>>> # List comprehension -- returns list
>>> stripped_list = [line.strip() for line in line_list]
```

You can select only certain elements by adding an `"if"` condition:

```
>>> stripped_list = [line.strip() for line in line_list
...                  if line != ""]
```

With a list comprehension, you get back a Python list; `stripped_list` is a list containing the resulting lines, not an iterator. Generator expressions return an iterator that computes the values as necessary, not needing to materialize all the values at once. This means that list comprehensions aren’t useful if you’re working with iterators that return an infinite stream or a very large amount of data. Generator expressions are preferable in these situations.

Generator expressions are surrounded by parentheses (“()”) and list comprehensions are surrounded by square brackets (“[]”). Generator expressions have the form:

```
( expression for expr in sequence1
             if condition1
             for expr2 in sequence2
             if condition2
             for expr3 in sequence3
             ...
             if condition3
             for exprN in sequenceN
             if conditionN )
```

Again, for a list comprehension only the outside brackets are different (square brackets instead of parentheses).

The elements of the generated output will be the successive values of `expression`. The `if` clauses are all optional; if present, `expression` is only evaluated and added to the result when `condition` is true.

Generator expressions always have to be written inside parentheses, but the parentheses signalling a function call also count. If you want to create an iterator that will be immediately passed to a function you can write:

```
obj_total = sum(obj.count for obj in list_all_objects())
```

The `for...in` clauses contain the sequences to be iterated over. The sequences do not have to be the same length, because they are iterated over from left to right, **not** in parallel. For each element in `sequence1`, `sequence2` is looped over from the beginning. `sequence3` is then looped over for each resulting pair of elements from `sequence1` and `sequence2`.

To put it another way, a list comprehension or generator expression is equivalent to the following Python code:

```
for expr1 in sequence1:
    if not (condition1):
        continue   # Skip this element
    for expr2 in sequence2:
        if not (condition2):
            continue   # Skip this element
        ...
        for exprN in sequenceN:
            if not (conditionN):
                continue   # Skip this element

            # Output the value of
            # the expression.
```

This means that when there are multiple `for...in` clauses but no `if` clauses, the length of the resulting output will be equal to the product of the lengths of all the sequences. If you have two lists of length 3, the output list is 9 elements long:

```
>>> seq1 = 'abc'
>>> seq2 = (1, 2, 3)
>>> [(x, y) for x in seq1 for y in seq2]
[('a', 1), ('a', 2), ('a', 3),
 ('b', 1), ('b', 2), ('b', 3),
 ('c', 1), ('c', 2), ('c', 3)]
```

To avoid introducing an ambiguity into Python’s grammar, if `expression` is creating a tuple, it must be surrounded with parentheses. The first list comprehension below is a syntax error, while the second one is correct:

```
# Syntax error
[x, y for x in seq1 for y in seq2]
# Correct
[(x, y) for x in seq1 for y in seq2]
```

Generators

Generators are a special class of functions that simplify the task of writing iterators. Regular functions compute a value and return it, but generators return an iterator that returns a stream of values.

You’re doubtless familiar with how regular function calls work in Python or C. When you call a function, it gets a private namespace where its local variables are created. When the function reaches a `return` statement, the local variables are destroyed and the value is returned to the caller. A later call to the same function creates a new private namespace and a fresh set of local variables. But, what if the local variables weren’t thrown away on exiting a function? What if you could later resume the function where it left off? This is what generators provide; they can be thought of as resumable functions.

Here’s the simplest example of a generator function:

```
>>> def generate_ints(N):
...    for i in range(N):
...        yield i
```

Any function containing a **`yield`** keyword is a generator function; this is detected by Python's bytecode compiler which compiles the function specially as a result.

When you call a generator function, it doesn't return a single value; instead it returns a generator object that supports the iterator protocol. On executing the `yield` expression, the generator outputs the value of `i`, similar to a `return` statement. The big difference between `yield` and a `return` statement is that on reaching a `yield` the generator's state of execution is suspended and local variables are preserved. On the next call to the generator's **`__next__()`** method, the function will resume executing.

Here's a sample usage of the `generate_ints()` generator:

```
>>> gen = generate_ints(3)
>>> gen
<generator object generate_ints at ...>
>>> next(gen)
0
>>> next(gen)
1
>>> next(gen)
2
>>> next(gen)
Traceback (most recent call last):
  File "stdin", line 1, in <module>
  File "stdin", line 2, in generate_ints
StopIteration
```

You could equally write `for i in generate_ints(5)`, or `a, b, c = generate_ints(3)`.

Inside a generator function, `return value` causes `StopIteration(value)` to be raised from the `__next__()` method. Once this happens, or the bottom of the function is reached, the procession of values ends and the generator cannot yield any further values.

You could achieve the effect of generators manually by writing your own class and storing all the local variables of the generator as instance variables. For example, returning a list of integers could be done by setting `self.count` to 0, and having the `__next__()` method increment `self.count` and return it. However, for a moderately complicated generator, writing a corresponding class can be much messier.

The test suite included with Python's library, Lib/test/test_generators.py [https://github.com/python/cpython/tree/3.14/Lib/test/test_generators.py], contains a number of more interesting examples. Here's one generator that implements an in-order traversal of a tree using generators recursively.

```
# A recursive generator that generates Tree leaves in in-order.
def inorder(t):
    if t:
        for x in inorder(t.left):
            yield x

        yield t.label

        for x in inorder(t.right):
            yield x
```

Two other examples in `test_generators.py` produce solutions for the N-Queens problem (placing N queens on an NxN chess board so that no queen threatens another) and the Knight's Tour (finding a route that takes a knight to every square of an NxN chessboard without visiting any square twice).

Passing values into a generator

In Python 2.4 and earlier, generators only produced output. Once a generator's code was invoked to create an iterator, there was no way to pass any new information into the function when its execution is resumed. You could hack together this ability by making the generator look at a global variable or by passing in some mutable object that callers then modify, but these approaches are messy.

In Python 2.5 there's a simple way to pass values into a generator. `yield` became an expression, returning a value that can be assigned to a variable or otherwise operated on:

```
val = (yield i)
```

I recommend that you **always** put parentheses around a `yield` expression when you're doing something with the returned value, as in the above example. The parentheses aren't always necessary, but it's easier to always add them instead of having to remember when they're needed.

(**PEP 342** [https://peps.python.org/pep-0342/] explains the exact rules, which are that a `yield`-expression must always be parenthesized except when it occurs at the top-level expression on the right-hand side of an assignment. This means you can write `val = yield i` but have to use parentheses when there's an operation, as in `val = (yield i) + 12`.)

Values are sent into a generator by calling its `send(value)` method. This method resumes the generator's code and the `yield` expression returns the specified value. If the regular `__next__()` method is called, the `yield` returns `None`.

Here's a simple counter that increments by 1 and allows changing the value of the internal counter.

```
def counter(maximum):
    i = 0
    while i < maximum:
        val = (yield i)
        # If value provided, change counter
        if val is not None:
            i = val
        else:
            i += 1
```

And here’s an example of changing the counter:

```
>>> it = counter(10)
>>> next(it)
0
>>> next(it)
1
>>> it.send(8)
8
>>> next(it)
9
>>> next(it)
Traceback (most recent call last):
  File "t.py", line 15, in <module>
    it.next()
StopIteration
```

Because `yield` will often be returning `None`, you should always check for this case. Don’t just use its value in expressions unless you’re sure that the `send()` method will be the only method used to resume your generator function.

In addition to `send()`, there are two other methods on generators:

- `throw(value)` is used to raise an exception inside the generator; the exception is raised by the `yield` expression where the generator’s execution is paused.

- `close()` sends a `GeneratorExit` exception to the generator to terminate the iteration. On receiving this exception, the generator's code must either raise `GeneratorExit` or `StopIteration`; catching the exception and doing anything else is illegal and will trigger a `RuntimeError`. `close()` will also be called by Python's garbage collector when the generator is garbage-collected.

  If you need to run cleanup code when a `GeneratorExit` occurs, I suggest using a `try: ... finally:` suite instead of catching `GeneratorExit`.

The cumulative effect of these changes is to turn generators from one-way producers of information into both producers and consumers.

Generators also become **coroutines**, a more generalized form of subroutines. Subroutines are entered at one point and exited at another point (the top of the function, and a `return` statement), but coroutines can be entered, exited, and resumed at many different points (the `yield` statements).

Built-in functions

Let's look in more detail at built-in functions often used with iterators.

Two of Python's built-in functions, `map()` and `filter()` duplicate the features of generator expressions:

`map(f, iterA, iterB, ...)` returns an iterator over the sequence

```
f(iterA[0], iterB[0]), f(iterA[1], iterB[1]), f(iterA[2], iterB[2]), ....
```

```
>>> def upper(s):
...     return s.upper()

>>> list(map(upper, ['sentence', 'fragment']))
['SENTENCE', 'FRAGMENT']
>>> [upper(s) for s in ['sentence', 'fragment']]
['SENTENCE', 'FRAGMENT']
```

You can of course achieve the same effect with a list comprehension.

`filter(predicate, iter)` returns an iterator over all the sequence elements that meet a certain condition, and is similarly duplicated by list comprehensions. A **predicate** is a function that returns the truth value of some condition; for use with `filter()`, the predicate must take a single value.

```
>>> def is_even(x):
...     return (x % 2) == 0

>>> list(filter(is_even, range(10)))
[0, 2, 4, 6, 8]
```

This can also be written as a list comprehension:

```
>>> list(x for x in range(10) if is_even(x))
[0, 2, 4, 6, 8]
```

`enumerate(iter, start=0)` counts off the elements in the iterable returning 2-tuples containing the count (from *start*) and each element.

```
>>> for item in enumerate(['subject', 'verb', 'object']):
...     print(item)
(0, 'subject')
(1, 'verb')
(2, 'object')
```

`enumerate()` is often used when looping through a list and recording the indexes at which certain conditions are met:

```
f = open('data.txt', 'r')
for i, line in enumerate(f):
    if line.strip() == '':
        print('Blank line at line #%i' % i)
```

`sorted(iterable, key=None, reverse=False)` collects all the elements of the iterable into a list, sorts the list, and returns the sorted result. The *key* and *reverse* arguments are passed through to the constructed list's `sort()` method.

```
>>> import random
>>> # Generate 8 random numbers between [0, 10000)
>>> rand_list = random.sample(range(10000), 8)
>>> rand_list
[769, 7953, 9828, 6431, 8442, 9878, 6213, 2207]
>>> sorted(rand_list)
[769, 2207, 6213, 6431, 7953, 8442, 9828, 9878]
>>> sorted(rand_list, reverse=True)
[9878, 9828, 8442, 7953, 6431, 6213, 2207, 769]
```

(For a more detailed discussion of sorting, see the Sorting Techniques.)

The `any(iter)` and `all(iter)` built-ins look at the truth values of an iterable's contents. `any()` returns `True` if any element in the iterable is a true value, and `all()` returns `True` if all of the elements are true values:

```
>>> any([0, 1, 0])
True
>>> any([0, 0, 0])
False
>>> any([1, 1, 1])
True
>>> all([0, 1, 0])
False
>>> all([0, 0, 0])
False
>>> all([1, 1, 1])
True
```

`zip(iterA, iterB, ...)` takes one element from each iterable and returns them in a tuple:

```
zip(['a', 'b', 'c'], (1, 2, 3)) =>
  ('a', 1), ('b', 2), ('c', 3)
```

It doesn’t construct an in-memory list and exhaust all the input iterators before returning; instead tuples are constructed and returned only if they’re requested. (The technical term for this behaviour is lazy evaluation [https://en.wikipedia.org/wiki/Lazy_evaluation].)

This iterator is intended to be used with iterables that are all of the same length. If the iterables are of different lengths, the resulting stream will be the same length as the shortest iterable.

```
zip(['a', 'b'], (1, 2, 3)) =>
  ('a', 1), ('b', 2)
```

You should avoid doing this, though, because an element may be taken from the longer iterators and discarded. This means you can’t go on to use the iterators further because you risk skipping a discarded element.

The itertools module

The `itertools` module contains a number of commonly used iterators as well as functions for combining several iterators. This section will introduce the module’s contents by showing small examples.

The module’s functions fall into a few broad classes:

- Functions that create a new iterator based on an existing iterator.

- Functions for treating an iterator’s elements as function arguments.
- Functions for selecting portions of an iterator’s output.
- A function for grouping an iterator’s output.

Creating new iterators

`itertools.count(start, step)` returns an infinite stream of evenly spaced values. You can optionally supply the starting number, which defaults to 0, and the interval between numbers, which defaults to 1:

```
itertools.count() =>
  0, 1, 2, 3, 4, 5, 6, 7, 8, 9, ...
itertools.count(10) =>
  10, 11, 12, 13, 14, 15, 16, 17, 18, 19, ...
itertools.count(10, 5) =>
  10, 15, 20, 25, 30, 35, 40, 45, 50, 55, ...
```

`itertools.cycle(iter)` saves a copy of the contents of a provided iterable and returns a new iterator that returns its elements from first to last. The new iterator will repeat these elements infinitely.

```
itertools.cycle([1, 2, 3, 4, 5]) =>
  1, 2, 3, 4, 5, 1, 2, 3, 4, 5, ...
```

`itertools.repeat(elem, [n])` returns the provided element *n* times, or returns the element endlessly if *n* is not provided.

```
itertools.repeat('abc') =>
  abc, abc, abc, abc, abc, abc, abc, abc, abc, abc, ...
itertools.repeat('abc', 5) =>
  abc, abc, abc, abc, abc
```

`itertools.chain(iterA, iterB, ...)` takes an arbitrary number of iterables as input, and returns all the elements of the first iterator, then all the elements of the second, and so on, until all of the iterables have been exhausted.

```
itertools.chain(['a', 'b', 'c'], (1, 2, 3)) =>
  a, b, c, 1, 2, 3
```

`itertools.islice(iter, [start], stop, [step])` returns a stream that's a slice of the iterator. With a single *stop* argument, it will return the first *stop* elements. If you supply a starting index, you'll get *stop-start* elements, and if you supply a value for *step*, elements will be skipped accordingly. Unlike Python's string and list slicing, you can't use negative values for *start*, *stop*, or *step*.

```
itertools.islice(range(10), 8) =>
  0, 1, 2, 3, 4, 5, 6, 7
itertools.islice(range(10), 2, 8) =>
  2, 3, 4, 5, 6, 7
itertools.islice(range(10), 2, 8, 2) =>
  2, 4, 6
```

`itertools.tee(iter, [n])` replicates an iterator; it returns *n* independent iterators that will all return the contents of the source iterator. If you don't supply a value for *n*, the default is 2. Replicating iterators requires saving some of the contents of the source iterator, so this can consume significant memory if the iterator is large and one of the new iterators is consumed more than the others.

```
itertools.tee( itertools.count() ) =>
   iterA, iterB

where iterA ->
   0, 1, 2, 3, 4, 5, 6, 7, 8, 9, ...

and   iterB ->
   0, 1, 2, 3, 4, 5, 6, 7, 8, 9, ...
```

Calling functions on elements

The `operator` module contains a set of functions corresponding to Python's operators. Some examples are `operator.add(a, b)` (adds two values), `operator.ne(a, b)` (same as `a != b`), and `operator.attrgetter('id')` (returns a callable that fetches the `.id` attribute).

`itertools.starmap(func, iter)` assumes that the iterable will return a stream of tuples, and calls *func* using these tuples as the arguments:

```
itertools.starmap(os.path.join,
                  [('/bin', 'python'), ('/usr', 'bin', 'java'),
                   ('/usr', 'bin', 'perl'), ('/usr', 'bin',
'ruby')])
=>
  /bin/python, /usr/bin/java, /usr/bin/perl, /usr/bin/ruby
```

Selecting elements

Another group of functions chooses a subset of an iterator's elements based on a predicate.

`itertools.filterfalse(predicate, iter)` is the opposite of `filter()`, returning all elements for which the predicate returns false:

```
itertools.filterfalse(is_even, itertools.count()) =>
  1, 3, 5, 7, 9, 11, 13, 15, ...
```

`itertools.takewhile(predicate, iter)` returns elements for as long as the predicate returns true. Once the predicate returns false, the iterator will signal the end of its results.

```
def less_than_10(x):
    return x < 10

itertools.takewhile(less_than_10, itertools.count()) =>
  0, 1, 2, 3, 4, 5, 6, 7, 8, 9

itertools.takewhile(is_even, itertools.count()) =>
  0
```

`itertools.dropwhile(predicate, iter)` discards elements while the predicate returns true, and then returns the rest of the iterable's results.

```
itertools.dropwhile(less_than_10, itertools.count()) =>
  10, 11, 12, 13, 14, 15, 16, 17, 18, 19, ...

itertools.dropwhile(is_even, itertools.count()) =>
  1, 2, 3, 4, 5, 6, 7, 8, 9, 10, ...
```

`itertools.compress(data, selectors)` takes two iterators and returns only those elements of *data* for which the corresponding element of *selectors* is true, stopping whenever either one is exhausted:

```
itertools.compress([1, 2, 3, 4, 5], [True, True, False, False,
True]) =>
   1, 2, 5
```

Combinatoric functions

The `itertools.combinations(iterable, r)` returns an iterator giving all possible *r*-tuple combinations of the elements contained in *iterable*.

```
itertools.combinations([1, 2, 3, 4, 5], 2) =>
  (1, 2), (1, 3), (1, 4), (1, 5),
  (2, 3), (2, 4), (2, 5),
  (3, 4), (3, 5),
  (4, 5)

itertools.combinations([1, 2, 3, 4, 5], 3) =>
  (1, 2, 3), (1, 2, 4), (1, 2, 5), (1, 3, 4), (1, 3, 5), (1, 4,
5),
  (2, 3, 4), (2, 3, 5), (2, 4, 5),
  (3, 4, 5)
```

The elements within each tuple remain in the same order as *iterable* returned them. For example, the number 1 is always before 2, 3, 4, or 5 in the examples above. A similar function, [`itertools.permutations(iterable, r=None)`](), removes this constraint on the order, returning all possible arrangements of length *r*:

```
itertools.permutations([1, 2, 3, 4, 5], 2) =>
  (1, 2), (1, 3), (1, 4), (1, 5),
  (2, 1), (2, 3), (2, 4), (2, 5),
  (3, 1), (3, 2), (3, 4), (3, 5),
  (4, 1), (4, 2), (4, 3), (4, 5),
  (5, 1), (5, 2), (5, 3), (5, 4)

itertools.permutations([1, 2, 3, 4, 5]) =>
  (1, 2, 3, 4, 5), (1, 2, 3, 5, 4), (1, 2, 4, 3, 5),
  ...
  (5, 4, 3, 2, 1)
```

If you don’t supply a value for *r* the length of the iterable is used, meaning that all the elements are permuted.

Note that these functions produce all of the possible combinations by position and don’t require that the contents of *iterable* are unique:

```
itertools.permutations('aba', 3) =>
  ('a', 'b', 'a'), ('a', 'a', 'b'), ('b', 'a', 'a'),
  ('b', 'a', 'a'), ('a', 'a', 'b'), ('a', 'b', 'a')
```

The identical tuple `('a', 'a', 'b')` occurs twice, but the two ‘a’ strings came from different positions.

The [`itertools.combinations_with_replacement(iterable, r)`]() function relaxes a different constraint: elements can be repeated within a single tuple. Conceptually an element is selected for the first position of each tuple and then is replaced before the second element is selected.

```
itertools.combinations_with_replacement([1, 2, 3, 4, 5], 2) =>
  (1, 1), (1, 2), (1, 3), (1, 4), (1, 5),
  (2, 2), (2, 3), (2, 4), (2, 5),
  (3, 3), (3, 4), (3, 5),
  (4, 4), (4, 5),
  (5, 5)
```

Grouping elements

The last function I’ll discuss, `itertools.groupby(iter, key_func=None)`, is the most complicated. `key_func(elem)` is a function that can compute a key value for each element returned by the iterable. If you don’t supply a key function, the key is simply each element itself.

`groupby()` collects all the consecutive elements from the underlying iterable that have the same key value, and returns a stream of 2-tuples containing a key value and an iterator for the elements with that key.

```
city_list = [('Decatur', 'AL'), ('Huntsville', 'AL'), ('Selma',
'AL'),
             ('Anchorage', 'AK'), ('Nome', 'AK'),
             ('Flagstaff', 'AZ'), ('Phoenix', 'AZ'), ('Tucson',
'AZ'),
             ...
            ]

def get_state(city_state):
    return city_state[1]

itertools.groupby(city_list, get_state) =>
  ('AL', iterator-1),
  ('AK', iterator-2),
  ('AZ', iterator-3), ...

where
iterator-1 =>
  ('Decatur', 'AL'), ('Huntsville', 'AL'), ('Selma', 'AL')
iterator-2 =>
  ('Anchorage', 'AK'), ('Nome', 'AK')
iterator-3 =>
  ('Flagstaff', 'AZ'), ('Phoenix', 'AZ'), ('Tucson', 'AZ')
```

`groupby()` assumes that the underlying iterable's contents will already be sorted based on the key. Note that the returned iterators also use the underlying iterable, so you have to consume the results of iterator-1 before requesting iterator-2 and its corresponding key.

The functools module

The `functools` module contains some higher-order functions. A **higher-order function** takes one or more functions as input and returns a new function. The most useful tool in this module is the `functools.partial()` function.

For programs written in a functional style, you'll sometimes want to construct variants of existing functions that have some of the parameters filled in. Consider a Python function `f(a, b, c)`; you may wish to create a new function `g(b, c)` that's equivalent to `f(1, b, c)`; you're filling in a value for one of `f()`'s parameters. This is called "partial function application".

The constructor for `partial()` takes the arguments `(function, arg1, arg2, ..., kwarg1=value1, kwarg2=value2)`. The resulting object is callable, so you can just call it to invoke `function` with the filled-in arguments.

Here's a small but realistic example:

```
import functools

def log(message, subsystem):
    """Write the contents of 'message' to the specified
subsystem."""
    print('%s: %s' % (subsystem, message))
    ...
```

```
server_log = functools.partial(log, subsystem='server')
server_log('Unable to open socket')
```

`functools.reduce(func, iter, [initial_value])` cumulatively performs an operation on all the iterable's elements and, therefore, can't be applied to infinite iterables. *func* must be a function that takes two elements and returns a single value. **`functools.reduce()`** takes the first two elements A and B returned by the iterator and calculates `func(A, B)`. It then requests the third element, C, calculates `func(func(A, B), C)`, combines this result with the fourth element returned, and continues until the iterable is exhausted. If the iterable returns no values at all, a **`TypeError`** exception is raised. If the initial value is supplied, it's used as a starting point and `func(initial_value, A)` is the first calculation.

```
>>> import operator, functools
>>> functools.reduce(operator.concat, ['A', 'BB', 'C'])
'ABBC'
>>> functools.reduce(operator.concat, [])
Traceback (most recent call last):
  ...
TypeError: reduce() of empty sequence with no initial value
>>> functools.reduce(operator.mul, [1, 2, 3], 1)
6
>>> functools.reduce(operator.mul, [], 1)
1
```

If you use **`operator.add()`** with **`functools.reduce()`**, you'll add up all the elements of the iterable. This case is so common that there's a special built-in called **`sum()`** to compute it:

```
>>> import functools, operator
>>> functools.reduce(operator.add, [1, 2, 3, 4], 0)
10
>>> sum([1, 2, 3, 4])
10
>>> sum([])
0
```

For many uses of `functools.reduce()`, though, it can be clearer to just write the obvious `for` loop:

```
import functools
# Instead of:
product = functools.reduce(operator.mul, [1, 2, 3], 1)

# You can write:
product = 1
for i in [1, 2, 3]:
    product *= i
```

A related function is `itertools.accumulate(iterable, func=operator.add)`. It performs the same calculation, but instead of returning only the final result, `accumulate()` returns an iterator that also yields each partial result:

```
itertools.accumulate([1, 2, 3, 4, 5]) =>
  1, 3, 6, 10, 15

itertools.accumulate([1, 2, 3, 4, 5], operator.mul) =>
  1, 2, 6, 24, 120
```

The operator module

The `operator` module was mentioned earlier. It contains a set of functions corresponding to Python’s operators. These functions are often useful in functional-style code because they save you from writing trivial functions that perform a single operation.

Some of the functions in this module are:

- Math operations: `add()`, `sub()`, `mul()`, `floordiv()`, `abs()`, …
- Logical operations: `not_()`, `truth()`.
- Bitwise operations: `and_()`, `or_()`, `invert()`.
- Comparisons: `eq()`, `ne()`, `lt()`, `le()`, `gt()`, and `ge()`.
- Object identity: `is_()`, `is_not()`.

Consult the operator module’s documentation for a complete list.

Small functions and the lambda expression

When writing functional-style programs, you’ll often need little functions that act as predicates or that combine elements in some way.

If there’s a Python built-in or a module function that’s suitable, you don’t need to define a new function at all:

```
stripped_lines = [line.strip() for line in lines]
existing_files = filter(os.path.exists, file_list)
```

If the function you need doesn’t exist, you need to write it. One way to write small functions is to use the **`lambda`** expression. `lambda` takes a number of parameters and an expression combining these parameters, and creates an anonymous function that returns the value of the expression:

```
adder = lambda x, y: x+y

print_assign = lambda name, value: name + '=' + str(value)
```

An alternative is to just use the `def` statement and define a function in the usual way:

```
def adder(x, y):
    return x + y

def print_assign(name, value):
    return name + '=' + str(value)
```

Which alternative is preferable? That’s a style question; my usual course is to avoid using `lambda`.

One reason for my preference is that `lambda` is quite limited in the functions it can define. The result has to be computable as a single expression, which means you can't have multiway `if... elif... else` comparisons or `try... except` statements. If you try to do too much in a `lambda` statement, you'll end up with an overly complicated expression that's hard to read. Quick, what's the following code doing?

```
import functools
total = functools.reduce(lambda a, b: (0, a[1] + b[1]), items)
[1]
```

You can figure it out, but it takes time to disentangle the expression to figure out what's going on. Using a short nested `def` statements makes things a little bit better:

```
import functools
def combine(a, b):
    return 0, a[1] + b[1]

total = functools.reduce(combine, items)[1]
```

But it would be best of all if I had simply used a `for` loop:

```
total = 0
for a, b in items:
    total += b
```

Or the `sum()` built-in and a generator expression:

```
total = sum(b for a, b in items)
```

Many uses of `functools.reduce()` are clearer when written as `for` loops.

Fredrik Lundh once suggested the following set of rules for refactoring uses of `lambda`:

1. Write a lambda function.
2. Write a comment explaining what the heck that lambda does.
3. Study the comment for a while, and think of a name that captures the essence of the comment.
4. Convert the lambda to a def statement, using that name.
5. Remove the comment.

I really like these rules, but you’re free to disagree about whether this lambda-free style is better.

Revision History and Acknowledgements

The author would like to thank the following people for offering suggestions, corrections and assistance with various drafts of this article: Ian Bicking, Nick Coghlan, Nick Efford, Raymond Hettinger, Jim Jewett, Mike Krell, Leandro Lameiro, Jussi Salmela, Collin Winter, Blake Winton.

Version 0.1: posted June 30 2006.

Version 0.11: posted July 1 2006. Typo fixes.

Version 0.2: posted July 10 2006. Merged genexp and listcomp sections into one. Typo fixes.

Version 0.21: Added more references suggested on the tutor mailing list.

Version 0.30: Adds a section on the `functional` module written by Collin Winter; adds short section on the operator module; a few other edits.

References

General

Structure and Interpretation of Computer Programs, by Harold Abelson and Gerald Jay Sussman with Julie Sussman. The book can be found at https://mitpress.mit.edu/sicp. In this classic textbook of computer science, chapters 2 and 3 discuss the use of sequences and streams to organize the data flow inside a program. The book uses Scheme for its examples, but many of the design approaches described in these chapters are applicable to functional-style Python code.

https://defmacro.org/2006/06/19/fp.html: A general introduction to functional programming that uses Java examples and has a lengthy historical introduction.

https://en.wikipedia.org/wiki/Functional_programming: General Wikipedia entry describing functional programming.

https://en.wikipedia.org/wiki/Coroutine: Entry for coroutines.

https://en.wikipedia.org/wiki/Partial_application: Entry for the concept of partial function application.

https://en.wikipedia.org/wiki/Currying: Entry for the concept of currying.

Python-specific

https://gnosis.cx/TPiP/: The first chapter of David Mertz's book *Text Processing in Python* discusses functional programming for text processing, in the section titled "Utilizing Higher-Order Functions in Text Processing".

Mertz also wrote a 3-part series of articles on functional programming for IBM’s DeveloperWorks site; see part 1 [https://developer.ibm.com/articles/l-prog/], part 2 [https://developer.ibm.com/tutorials/l-prog2/], and part 3 [https://developer.ibm.com/tutorials/l-prog3/],

Python documentation

Documentation for the `itertools` module.

Documentation for the `functools` module.

Documentation for the `operator` module.

PEP 289 [https://peps.python.org/pep-0289/]: “Generator Expressions”

PEP 342 [https://peps.python.org/pep-0342/]: “Coroutines via Enhanced Generators” describes the new generator features in Python 2.5.

Logging HOWTO

Author:: Vinay Sajip <vinay_sajip at red-dove dot com>

This page contains tutorial information. For links to reference information and a logging cookbook, please see Other resources.

Basic Logging Tutorial

Logging is a means of tracking events that happen when some software runs. The software's developer adds logging calls to their code to indicate that certain events have occurred. An event is described by a descriptive message which can optionally contain variable data (i.e. data that is potentially different for each occurrence of the event). Events also have an importance which the developer ascribes to the event; the importance can also be called the *level* or *severity*.

When to use logging

You can access logging functionality by creating a logger via `logger = getLogger(__name__)`, and then calling the logger's `debug()`, `info()`, `warning()`, `error()` and `critical()` methods. To determine when to use logging, and to see which logger methods to use when, see the table below. It states, for each of a set of common tasks, the best tool to use for that task.

Task you want to perform	The best tool for the task
Display console output for ordinary usage of a command line script or program	`print()`
Report events that occur during normal operation of a program (e.g. for status monitoring or fault investigation)	A logger's `info()` (or `debug()` method for very detailed output for diagnostic purposes)
Issue a warning regarding a particular runtime event	`warnings.warn()` in library code if the issue is avoidable and the client application should be modified to eliminate the warning A logger's `warning()` method if there is nothing the client application can do

Task you want to perform	The best tool for the task
	about the situation, but the event should still be noted
Report an error regarding a particular runtime event	Raise an exception
Report suppression of an error without raising an exception (e.g. error handler in a long-running server process)	A logger’s `error()`, `exception()` or `critical()` method as appropriate for the specific error and application domain

The logger methods are named after the level or severity of the events they are used to track. The standard levels and their applicability are described below (in increasing order of severity):

Level	When it’s used
`DEBUG`	Detailed information, typically of interest only when diagnosing problems.
`INFO`	Confirmation that things are working as expected.
`WARNING`	An indication that something unexpected happened, or indicative of some problem in the near future (e.g. ‘disk space low’). The software is still working as expected.
`ERROR`	Due to a more serious problem, the software has not been able to perform some function.
`CRITICAL`	A serious error, indicating that the program itself may be unable to continue running.

The default level is `WARNING`, which means that only events of this severity and higher will be tracked, unless the logging package is configured to do otherwise.

Events that are tracked can be handled in different ways. The simplest way of handling tracked events is to print them to the console. Another common way is to write them to a disk file.

A simple example

A very simple example is:

```
import logging
logging.warning('Watch out!')  # will print a message to the console
logging.info('I told you so')  # will not print anything
```

If you type these lines into a script and run it, you'll see:

```
WARNING:root:Watch out!
```

printed out on the console. The `INFO` message doesn't appear because the default level is `WARNING`. The printed message includes the indication of the level and the description of the event provided in the logging call, i.e. 'Watch out!'. The actual output can be formatted quite flexibly if you need that; formatting options will also be explained later.

Notice that in this example, we use functions directly on the `logging` module, like `logging.debug`, rather than creating a logger and calling functions on it. These functions operate on the root logger, but can be useful as they will call **`basicConfig()`** for you if it has not been called yet, like in this example. In larger programs you'll usually want to control the logging configuration explicitly however - so for that reason as well as others, it's better to create loggers and call their methods.

Logging to a file

A very common situation is that of recording logging events in a file, so let's look at that next. Be sure to try the following in a newly started Python interpreter, and don't just continue from the session described above:

```
import logging
logger = logging.getLogger(__name__)
logging.basicConfig(filename='example.log', encoding='utf-8', level=logging.DEBUG)
logger.debug('This message should go to the log file')
logger.info('So should this')
logger.warning('And this, too')
logger.error('And non-ASCII stuff, too, like Øresund and Malmö')
```

Changed in version 3.9: The *encoding* argument was added. In earlier Python versions, or if not specified, the encoding used is the default value used by **`open()`**. While not shown in the above example, an *errors* argument can also now be passed, which determines how encoding errors are handled. For available values and the default, see the documentation for **`open()`**.

And now if we open the file and look at what we have, we should find the log messages:

```
DEBUG:__main__:This message should go to the log file
INFO:__main__:So should this
WARNING:__main__:And this, too
ERROR:__main__:And non-ASCII stuff, too, like Øresund and Malmö
```

This example also shows how you can set the logging level which acts as the threshold for tracking. In this case, because we set the threshold to `DEBUG`, all of the messages were printed.

If you want to set the logging level from a command-line option such as:

```
--log=INFO
```

and you have the value of the parameter passed for `--log` in some variable *loglevel*, you can use:

```
getattr(logging, loglevel.upper())
```

to get the value which you'll pass to `basicConfig()` via the *level* argument. You may want to error check any user input value, perhaps as in the following example:

```
# assuming loglevel is bound to the string value obtained from the
# command line argument. Convert to upper case to allow the user to
# specify --log=DEBUG or --log=debug
numeric_level = getattr(logging, loglevel.upper(), None)
if not isinstance(numeric_level, int):
    raise ValueError('Invalid log level: %s' % loglevel)
logging.basicConfig(level=numeric_level, ...)
```

The call to `basicConfig()` should come *before* any calls to a logger's methods such as `debug()`, `info()`, etc. Otherwise, that logging event may not be handled in the desired manner.

If you run the above script several times, the messages from successive runs are appended to the file *example.log*. If you want each run to start afresh, not remembering the messages from earlier runs, you can specify the *filemode* argument, by changing the call in the above example to:

```
logging.basicConfig(filename='example.log', filemode='w', level=logging.DEBUG)
```

The output will be the same as before, but the log file is no longer appended to, so the messages from earlier runs are lost.

Logging variable data

To log variable data, use a format string for the event description message and append the variable data as arguments. For example:

```
import logging
logging.warning('%s before you %s', 'Look', 'leap!')
```

will display:

```
WARNING:root:Look before you leap!
```

As you can see, merging of variable data into the event description message uses the old, %-style of string formatting. This is for backwards compatibility: the logging package pre-dates newer formatting options such as **`str.format()`** and **`string.Template`**. These newer formatting options *are* supported, but exploring them is outside the scope of this tutorial: see Using particular formatting styles throughout your application for more information.

Changing the format of displayed messages

To change the format which is used to display messages, you need to specify the format you want to use:

```
import logging
logging.basicConfig(format='%(levelname)s:%(message)s', level=logging.DEBUG)
logging.debug('This message should appear on the console')
logging.info('So should this')
logging.warning('And this, too')
```

which would print:

```
DEBUG:This message should appear on the console
INFO:So should this
WARNING:And this, too
```

Notice that the 'root' which appeared in earlier examples has disappeared. For a full set of things that can appear in format strings, you can refer to the documentation for LogRecord attributes, but for simple usage, you just need the *levelname* (severity), *message* (event description, including variable data) and perhaps to display when the event occurred. This is described in the next section.

Displaying the date/time in messages

To display the date and time of an event, you would place '%(asctime)s' in your format string:

```
import logging
logging.basicConfig(format='%(asctime)s %(message)s')
logging.warning('is when this event was logged.')
```

which should print something like this:

```
2010-12-12 11:41:42,612 is when this event was logged.
```

The default format for date/time display (shown above) is like ISO8601 or **RFC 3339** [https://datatracker.ietf.org/doc/html/rfc3339.html]. If you need more control over the formatting of the date/time, provide a *datefmt* argument to `basicConfig`, as in this example:

```
import logging
logging.basicConfig(format='%(asctime)s %(message)s', datefmt='%m/%d/%Y %I:%M:%S %p')
logging.warning('is when this event was logged.')
```

which would display something like this:

```
12/12/2010 11:46:36 AM is when this event was logged.
```

The format of the *datefmt* argument is the same as supported by `time.strftime()`.

Next Steps

That concludes the basic tutorial. It should be enough to get you up and running with logging. There’s a lot more that the logging package offers, but to get the best out of it, you’ll need to invest a little more of your time in reading the following sections. If you’re ready for that, grab some of your favourite beverage and carry on.

If your logging needs are simple, then use the above examples to incorporate logging into your own scripts, and if you run into problems or don’t understand something, please post a question in the Help category of the Python discussion forum [https://discuss.python.org/c/help/7] and you should receive help before too long.

Still here? You can carry on reading the next few sections, which provide a slightly more advanced/in-depth tutorial than the basic one above. After that, you can take a look at the Logging Cookbook.

Advanced Logging Tutorial

The logging library takes a modular approach and offers several categories of components: loggers, handlers, filters, and formatters.

- Loggers expose the interface that application code directly uses.
- Handlers send the log records (created by loggers) to the appropriate destination.
- Filters provide a finer grained facility for determining which log records to output.
- Formatters specify the layout of log records in the final output.

Log event information is passed between loggers, handlers, filters and formatters in a `LogRecord` instance.

Logging is performed by calling methods on instances of the `Logger` class (hereafter called *loggers*). Each instance has a name, and they are conceptually arranged in a namespace hierarchy using dots (periods) as separators. For example, a logger named ‘scan’ is the parent of loggers ‘scan.text’, ‘scan.html’ and ‘scan.pdf’. Logger names can be anything you want, and indicate the area of an application in which a logged message originates.

A good convention to use when naming loggers is to use a module-level logger, in each module which uses logging, named as follows:

```
logger = logging.getLogger(__name__)
```

This means that logger names track the package/module hierarchy, and it’s intuitively obvious where events are logged just from the logger name.

The root of the hierarchy of loggers is called the root logger. That’s the logger used by the functions **`debug()`**, **`info()`**, **`warning()`**, **`error()`** and **`critical()`**, which just call the same-named method of the root logger. The functions and the methods have the same signatures. The root logger’s name is printed as ‘root’ in the logged output.

It is, of course, possible to log messages to different destinations. Support is included in the package for writing log messages to files, HTTP GET/POST locations, email via SMTP, generic sockets, queues, or OS-specific logging mechanisms such as syslog or the Windows NT event log. Destinations are served by *handler* classes. You can create your own log destination class if you have special requirements not met by any of the built-in handler classes.

By default, no destination is set for any logging messages. You can specify a destination (such as console or file) by using **`basicConfig()`** as in the tutorial examples. If you call the functions **`debug()`**, **`info()`**, **`warning()`**, **`error()`** and **`critical()`**, they will check to see if no destination is set; and if one is not set, they will set a destination of the console (`sys.stderr`) and a default format for the displayed message before delegating to the root logger to do the actual message output.

The default format set by **`basicConfig()`** for messages is:

```
severity:logger name:message
```

You can change this by passing a format string to **`basicConfig()`** with the *format* keyword argument. For all options regarding how a format string is constructed, see Formatter Objects.

Logging Flow

The flow of log event information in loggers and handlers is illustrated in the following diagram.

Loggers

`Logger` objects have a threefold job. First, they expose several methods to application code so that applications can log messages at runtime. Second, logger objects determine which log messages to act upon based upon severity (the default filtering facility) or filter objects. Third, logger objects pass along relevant log messages to all interested log handlers.

The most widely used methods on logger objects fall into two categories: configuration and message sending.

These are the most common configuration methods:

- `Logger.setLevel()` specifies the lowest-severity log message a logger will handle, where debug is the lowest built-in severity level and critical is the highest built-in severity. For example, if the severity level is INFO, the logger will handle only INFO, WARNING, ERROR, and CRITICAL messages and will ignore DEBUG messages.
- `Logger.addHandler()` and `Logger.removeHandler()` add and remove handler objects from the logger object. Handlers are covered in more detail in Handlers.
- `Logger.addFilter()` and `Logger.removeFilter()` add and remove filter objects from the logger object. Filters are covered in more detail in Filter Objects.

You don’t need to always call these methods on every logger you create. See the last two paragraphs in this section.

With the logger object configured, the following methods create log messages:

- `Logger.debug()`, `Logger.info()`, `Logger.warning()`, `Logger.error()`, and `Logger.critical()` all create log records with a message and a level that corresponds to their respective method names. The message is actually a format string, which may contain the standard string substitution syntax of `%s`, `%d`, `%f`, and so on. The rest of their arguments is a list of objects that correspond with the substitution fields in the message. With regard to `**kwargs`, the logging methods care only about a keyword of `exc_info` and use it to determine whether to log exception information.
- `Logger.exception()` creates a log message similar to `Logger.error()`. The difference is that `Logger.exception()` dumps a stack trace along with it. Call this method only from an exception handler.
- `Logger.log()` takes a log level as an explicit argument. This is a little more verbose for logging messages than using the log level convenience methods listed above, but this is how to log at custom log levels.

`getLogger()` returns a reference to a logger instance with the specified name if it is provided, or `root` if not. The names are period-separated hierarchical structures. Multiple calls to `getLogger()` with the same name will return a reference to the same logger object. Loggers that are further down in the hierarchical list are children of loggers higher up in the list. For example, given a logger with a name of `foo`, loggers with names of `foo.bar`, `foo.bar.baz`, and `foo.bam` are all descendants of `foo`.

Loggers have a concept of *effective level*. If a level is not explicitly set on a logger, the level of its parent is used instead as its effective level. If the parent has no explicit level set, *its* parent is examined, and so on - all ancestors are searched until an explicitly set level is found. The root logger always has an explicit level set (`WARNING` by default). When deciding whether to process an event, the effective level of the logger is used to determine whether the event is passed to the logger’s handlers.

Child loggers propagate messages up to the handlers associated with their ancestor loggers. Because of this, it is unnecessary to define and configure handlers for all the loggers an application uses. It is sufficient to configure handlers for a top-level logger and create child loggers as needed. (You can, however, turn off propagation by setting the *propagate* attribute of a logger to `False`.)

Handlers

`Handler` objects are responsible for dispatching the appropriate log messages (based on the log messages' severity) to the handler's specified destination. **`Logger`** objects can add zero or more handler objects to themselves with an **`addHandler()`** method. As an example scenario, an application may want to send all log messages to a log file, all log messages of error or higher to stdout, and all messages of critical to an email address. This scenario requires three individual handlers where each handler is responsible for sending messages of a specific severity to a specific location.

The standard library includes quite a few handler types (see Useful Handlers); the tutorials use mainly **`StreamHandler`** and **`FileHandler`** in its examples.

There are very few methods in a handler for application developers to concern themselves with. The only handler methods that seem relevant for application developers who are using the built-in handler objects (that is, not creating custom handlers) are the following configuration methods:

- The **`setLevel()`** method, just as in logger objects, specifies the lowest severity that will be dispatched to the appropriate destination. Why are there two **`setLevel()`** methods? The level set in the logger determines which severity of messages it will pass to its handlers. The level set in each handler determines which messages that handler will send on.
- **`setFormatter()`** selects a Formatter object for this handler to use.
- **`addFilter()`** and **`removeFilter()`** respectively configure and deconfigure filter objects on handlers.

Application code should not directly instantiate and use instances of **`Handler`**. Instead, the **`Handler`** class is a base class that defines the interface that all handlers should have and establishes some default behavior that child classes can use (or override).

Formatters

Formatter objects configure the final order, structure, and contents of the log message. Unlike the base **`logging.Handler`** class, application code may instantiate formatter classes, although you could likely subclass the formatter if your application needs special behavior. The constructor takes three optional arguments – a message format string, a date format string and a style indicator.

logging.Formatter.__init__(*fmt=None*, *datefmt=None*, *style='%'*)

If there is no message format string, the default is to use the raw message. If there is no date format string, the default date format is:

```
%Y-%m-%d %H:%M:%S
```

with the milliseconds tacked on at the end. The `style` is one of `'%'`, `'{'`, or `'$'`. If one of these is not specified, then `'%'` will be used.

If the `style` is `'%'`, the message format string uses `%(<dictionary key>)s` styled string substitution; the possible keys are documented in LogRecord attributes. If the style is `'{'`, the message format string is assumed to be compatible with **`str.format()`** (using keyword arguments), while if the style is `'$'` then the message format string should conform to what is expected by **`string.Template.substitute()`**.

Changed in version 3.2: Added the `style` parameter.

The following message format string will log the time in a human-readable format, the severity of the message, and the contents of the message, in that order:

```
'%(asctime)s - %(levelname)s - %(message)s'
```

Formatters use a user-configurable function to convert the creation time of a record to a tuple. By default, **`time.localtime()`** is used; to change this for a particular formatter instance, set the `converter` attribute of the instance to a function with the same signature as **`time.localtime()`** or **`time.gmtime()`**. To change it for all formatters, for example if you want all logging times to be shown in GMT, set the `converter` attribute in the Formatter class (to `time.gmtime` for GMT display).

Configuring Logging

Programmers can configure logging in three ways:

1. Creating loggers, handlers, and formatters explicitly using Python code that calls the configuration methods listed above.
2. Creating a logging config file and reading it using the **`fileConfig()`** function.
3. Creating a dictionary of configuration information and passing it to the **`dictConfig()`** function.

For the reference documentation on the last two options, see Configuration functions. The following example configures a very simple logger, a console handler, and a simple formatter using Python code:

```
import logging

# create logger
logger = logging.getLogger('simple_example')
logger.setLevel(logging.DEBUG)

# create console handler and set level to debug
```

```
ch = logging.StreamHandler()
ch.setLevel(logging.DEBUG)

# create formatter
formatter = logging.Formatter('%(asctime)s - %(name)s - %(levelname)s - %(message)s')

# add formatter to ch
ch.setFormatter(formatter)

# add ch to logger
logger.addHandler(ch)

# 'application' code
logger.debug('debug message')
logger.info('info message')
logger.warning('warn message')
logger.error('error message')
logger.critical('critical message')
```

Running this module from the command line produces the following output:

```
$ python simple_logging_module.py
2005-03-19 15:10:26,618 - simple_example - DEBUG - debug message
2005-03-19 15:10:26,620 - simple_example - INFO - info message
2005-03-19 15:10:26,695 - simple_example - WARNING - warn message
2005-03-19 15:10:26,697 - simple_example - ERROR - error message
2005-03-19 15:10:26,773 - simple_example - CRITICAL - critical message
```

The following Python module creates a logger, handler, and formatter nearly identical to those in the example listed above, with the only difference being the names of the objects:

```
import logging
import logging.config

logging.config.fileConfig('logging.conf')

# create logger
logger = logging.getLogger('simpleExample')

# 'application' code
logger.debug('debug message')
logger.info('info message')
logger.warning('warn message')
logger.error('error message')
logger.critical('critical message')
```

Here is the logging.conf file:

```
[loggers]
keys=root,simpleExample

[handlers]
keys=consoleHandler

[formatters]
keys=simpleFormatter
```

```
[logger_root]
level=DEBUG
handlers=consoleHandler

[logger_simpleExample]
level=DEBUG
handlers=consoleHandler
qualname=simpleExample
propagate=0

[handler_consoleHandler]
class=StreamHandler
level=DEBUG
formatter=simpleFormatter
args=(sys.stdout,)

[formatter_simpleFormatter]
format=%(asctime)s - %(name)s - %(levelname)s - %(message)s
```

The output is nearly identical to that of the non-config-file-based example:

```
$ python simple_logging_config.py
2005-03-19 15:38:55,977 - simpleExample - DEBUG - debug message
2005-03-19 15:38:55,979 - simpleExample - INFO - info message
2005-03-19 15:38:56,054 - simpleExample - WARNING - warn message
2005-03-19 15:38:56,055 - simpleExample - ERROR - error message
2005-03-19 15:38:56,130 - simpleExample - CRITICAL - critical message
```

You can see that the config file approach has a few advantages over the Python code approach, mainly separation of configuration and code and the ability of noncoders to easily modify the logging properties.

Warning

The **`fileConfig()`** function takes a default parameter, `disable_existing_loggers`, which defaults to `True` for reasons of backward compatibility. This may or may not be what you want, since it will cause any non-root loggers existing before the **`fileConfig()`** call to be disabled unless they (or an ancestor) are explicitly named in the configuration. Please refer to the reference documentation for more information, and specify `False` for this parameter if you wish.

The dictionary passed to **`dictConfig()`** can also specify a Boolean value with key `disable_existing_loggers`, which if not specified explicitly in the dictionary also defaults to being interpreted as `True`. This leads to the logger-disabling behaviour described above, which may not be what you want - in which case, provide the key explicitly with a value of `False`.

Note that the class names referenced in config files need to be either relative to the logging module, or absolute values which can be resolved using normal import mechanisms. Thus, you could use either **`WatchedFileHandler`** (relative to the logging module) or `mypackage.mymodule.MyHandler` (for a class defined in package `mypackage` and module `mymodule`, where `mypackage` is available on the Python import path).

In Python 3.2, a new means of configuring logging has been introduced, using dictionaries to hold configuration information. This provides a superset of the functionality of the config-file-based approach outlined above, and is the recommended configuration method for new applications and deployments. Because a Python dictionary is used to hold configuration information, and since you can populate that dictionary using different means, you have more options for configuration. For example, you can use a configuration file in JSON format, or, if you have access to YAML processing functionality, a file in YAML format, to populate the configuration dictionary. Or, of course, you can construct the dictionary in Python code, receive it in pickled form over a socket, or use whatever approach makes sense for your application.

Here’s an example of the same configuration as above, in YAML format for the new dictionary-based approach:

```
version: 1
formatters:
  simple:
    format: '%(asctime)s - %(name)s - %(levelname)s - %(message)s'
handlers:
  console:
    class: logging.StreamHandler
    level: DEBUG
    formatter: simple
    stream: ext://sys.stdout
loggers:
  simpleExample:
    level: DEBUG
    handlers: [console]
    propagate: no
root:
  level: DEBUG
  handlers: [console]
```

For more information about logging using a dictionary, see Configuration functions.

What happens if no configuration is provided

If no logging configuration is provided, it is possible to have a situation where a logging event needs to be output, but no handlers can be found to output the event.

The event is output using a ‘handler of last resort’, stored in **`lastResort`**. This internal handler is not associated with any logger, and acts like a **`StreamHandler`** which writes the event description message to the current value of `sys.stderr` (therefore respecting any redirections which may be in effect). No formatting is done on the message - just the bare event description message is printed. The handler’s level is set to `WARNING`, so all events at this and greater severities will be output.

Changed in version 3.2: For versions of Python prior to 3.2, the behaviour is as follows:

- If **`raiseExceptions`** is `False` (production mode), the event is silently dropped.
- If **`raiseExceptions`** is `True` (development mode), a message ‘No handlers could be found for logger X.Y.Z’ is printed once.

To obtain the pre-3.2 behaviour, **`lastResort`** can be set to `None`.

Configuring Logging for a Library

When developing a library which uses logging, you should take care to document how the library uses logging - for example, the names of loggers used. Some consideration also needs to be given to its logging configuration. If the using application does not use logging, and library code makes logging calls, then (as described in the previous section) events of severity `WARNING` and greater will be printed to `sys.stderr`. This is regarded as the best default behaviour.

If for some reason you *don’t* want these messages printed in the absence of any logging configuration, you can attach a do-nothing handler to the top-level logger for your library. This avoids the message being printed, since a handler will always be found for the library’s events: it just doesn’t produce any output. If the library user configures logging for application use, presumably that configuration will add some handlers, and if levels are suitably configured then logging calls made in library code will send output to those handlers, as normal.

A do-nothing handler is included in the logging package: **`NullHandler`** (since Python 3.1). An instance of this handler could be added to the top-level logger of the logging namespace used by the library (*if* you want to prevent your library’s logged events being output to `sys.stderr` in the absence of logging configuration). If all logging by a library *foo* is done using loggers with names matching ‘foo.x’, ‘foo.x.y’, etc. then the code:

```
import logging
logging.getLogger('foo').addHandler(logging.NullHandler())
```

should have the desired effect. If an organisation produces a number of libraries, then the logger name specified can be ‘orgname.foo’ rather than just ‘foo’.

Note

It is strongly advised that you *do not log to the root logger* in your library. Instead, use a logger with a unique and easily identifiable name, such as the `__name__` for your library’s top-level package or module. Logging to the root logger will make it difficult or impossible for the application developer to configure the logging verbosity or handlers of your library as they wish.

Note

It is strongly advised that you *do not add any handlers other than* `NullHandler` *to your library's loggers*. This is because the configuration of handlers is the prerogative of the application developer who uses your library. The application developer knows their target audience and what handlers are most appropriate for their application: if you add handlers 'under the hood', you might well interfere with their ability to carry out unit tests and deliver logs which suit their requirements.

Logging Levels

The numeric values of logging levels are given in the following table. These are primarily of interest if you want to define your own levels, and need them to have specific values relative to the predefined levels. If you define a level with the same numeric value, it overwrites the predefined value; the predefined name is lost.

Level	Numeric value
`CRITICAL`	50
`ERROR`	40
`WARNING`	30
`INFO`	20
`DEBUG`	10
`NOTSET`	0

Levels can also be associated with loggers, being set either by the developer or through loading a saved logging configuration. When a logging method is called on a logger, the logger compares its own level with the level associated with the method call. If the logger's level is higher than the method call's, no logging message is actually generated. This is the basic mechanism controlling the verbosity of logging output.

Logging messages are encoded as instances of the `LogRecord` class. When a logger decides to actually log an event, a `LogRecord` instance is created from the logging message.

Logging messages are subjected to a dispatch mechanism through the use of *handlers*, which are instances of subclasses of the `Handler` class. Handlers are responsible for ensuring that a logged message (in the form of a `LogRecord`) ends up in a particular location (or set of locations) which is useful for the target audience for that message (such as end users, support desk staff, system administrators, developers). Handlers are passed `LogRecord` instances intended for particular destinations. Each logger can have zero, one or more handlers associated with it (via the `addHandler()` method of `Logger`). In addition to any handlers directly associated with a logger, *all handlers associated with all ancestors of the logger* are called to dispatch the message (unless the *propagate* flag for a logger is set to a false value, at which point the passing to ancestor handlers stops).

Just as for loggers, handlers can have levels associated with them. A handler's level acts as a filter in the same way as a logger's level does. If a handler decides to actually dispatch an event, the `emit()` method is used to send the message to its destination. Most user-defined subclasses of `Handler` will need to override this `emit()`.

Custom Levels

Defining your own levels is possible, but should not be necessary, as the existing levels have been chosen on the basis of practical experience. However, if you are convinced that you need custom levels, great care should be exercised when doing this, and it is possibly *a very bad idea to define custom levels if you are developing a library*. That's because if multiple library authors all define their own custom levels, there is a chance that the logging output from such multiple libraries used together will be difficult for the using developer to control and/or interpret, because a given numeric value might mean different things for different libraries.

Useful Handlers

In addition to the base `Handler` class, many useful subclasses are provided:

1. `StreamHandler` instances send messages to streams (file-like objects).
2. `FileHandler` instances send messages to disk files.
3. `BaseRotatingHandler` is the base class for handlers that rotate log files at a certain point. It is not meant to be instantiated directly. Instead, use `RotatingFileHandler` or `TimedRotatingFileHandler`.

4. `RotatingFileHandler` instances send messages to disk files, with support for maximum log file sizes and log file rotation.
5. `TimedRotatingFileHandler` instances send messages to disk files, rotating the log file at certain timed intervals.
6. `SocketHandler` instances send messages to TCP/IP sockets. Since 3.4, Unix domain sockets are also supported.
7. `DatagramHandler` instances send messages to UDP sockets. Since 3.4, Unix domain sockets are also supported.
8. `SMTPHandler` instances send messages to a designated email address.
9. `SysLogHandler` instances send messages to a Unix syslog daemon, possibly on a remote machine.
10. `NTEventLogHandler` instances send messages to a Windows NT/2000/XP event log.
11. `MemoryHandler` instances send messages to a buffer in memory, which is flushed whenever specific criteria are met.
12. `HTTPHandler` instances send messages to an HTTP server using either `GET` or `POST` semantics.
13. `WatchedFileHandler` instances watch the file they are logging to. If the file changes, it is closed and reopened using the file name. This handler is only useful on Unix-like systems; Windows does not support the underlying mechanism used.
14. `QueueHandler` instances send messages to a queue, such as those implemented in the `queue` or `multiprocessing` modules.
15. `NullHandler` instances do nothing with error messages. They are used by library developers who want to use logging, but want to avoid the ‘No handlers could be found for logger *XXX*’ message which can be displayed if the library user has not configured logging. See Configuring Logging for a Library for more information.

Added in version 3.1: The `NullHandler` class.

Added in version 3.2: The `QueueHandler` class.

The `NullHandler`, `StreamHandler` and `FileHandler` classes are defined in the core logging package. The other handlers are defined in a sub-module, `logging.handlers`. (There is also another sub-module, `logging.config`, for configuration functionality.)

Logged messages are formatted for presentation through instances of the `Formatter` class. They are initialized with a format string suitable for use with the % operator and a dictionary.

For formatting multiple messages in a batch, instances of `BufferingFormatter` can be used. In addition to the format string (which is applied to each message in the batch), there is provision for header and trailer format strings.

When filtering based on logger level and/or handler level is not enough, instances of `Filter` can be added to both `Logger` and `Handler` instances (through their `addFilter()` method). Before deciding to process a message further, both loggers and handlers consult all their filters for permission. If any filter returns a false value, the message is not processed further.

The basic `Filter` functionality allows filtering by specific logger name. If this feature is used, messages sent to the named logger and its children are allowed through the filter, and all others dropped.

Exceptions raised during logging

The logging package is designed to swallow exceptions which occur while logging in production. This is so that errors which occur while handling logging events - such as logging misconfiguration, network or other similar errors - do not cause the application using logging to terminate prematurely.

`SystemExit` and `KeyboardInterrupt` exceptions are never swallowed. Other exceptions which occur during the `emit()` method of a `Handler` subclass are passed to its `handleError()` method.

The default implementation of `handleError()` in `Handler` checks to see if a module-level variable, `raiseExceptions`, is set. If set, a traceback is printed to `sys.stderr`. If not set, the exception is swallowed.

Note

The default value of `raiseExceptions` is `True`. This is because during development, you typically want to be notified of any exceptions that occur. It’s advised that you set `raiseExceptions` to `False` for production usage.

Using arbitrary objects as messages

In the preceding sections and examples, it has been assumed that the message passed when logging the event is a string. However, this is not the only possibility. You can pass an arbitrary object as a message, and its `__str__()` method will be called when the logging system needs to convert it to a string representation. In fact, if you want to, you can avoid computing a string representation altogether - for example, the `SocketHandler` emits an event by pickling it and sending it over the wire.

Optimization

Formatting of message arguments is deferred until it cannot be avoided. However, computing the arguments passed to the logging method can also be expensive, and you may want to avoid doing it if the logger will just throw away your event. To decide what to do, you can call the `isEnabledFor()` method which takes a level argument and returns true if the event would be created by the Logger for that level of call. You can write code like this:

```
if logger.isEnabledFor(logging.DEBUG):
    logger.debug('Message with %s, %s', expensive_func1(),
                                        expensive_func2())
```

so that if the logger's threshold is set above `DEBUG`, the calls to `expensive_func1` and `expensive_func2` are never made.

Note

In some cases, `isEnabledFor()` can itself be more expensive than you'd like (e.g. for deeply nested loggers where an explicit level is only set high up in the logger hierarchy). In such cases (or if you want to avoid calling a method in tight loops), you can cache the result of a call to `isEnabledFor()` in a local or instance variable, and use that instead of calling the method each time. Such a cached value would only need to be recomputed when the logging configuration changes dynamically while the application is running (which is not all that common).

There are other optimizations which can be made for specific applications which need more precise control over what logging information is collected. Here's a list of things you can do to avoid processing during logging which you don't need:

What you don't want to collect	How to avoid collecting it
Information about where calls were made from.	Set `logging._srcfile` to `None`. This avoids calling `sys._getframe()`, which may help to speed up your code in environments like PyPy (which can't speed up code that uses `sys._getframe()`).
Threading information.	Set `logging.logThreads` to `False`.
Current process ID (`os.getpid()`)	Set `logging.logProcesses` to `False`.
Current process name when using `multiprocessing` to manage multiple processes.	Set `logging.logMultiprocessing` to `False`.
Current `asyncio.Task` name when using `asyncio`.	Set `logging.logAsyncioTasks` to `False`.

Also note that the core logging module only includes the basic handlers. If you don’t import `logging.handlers` and `logging.config`, they won’t take up any memory.

Other resources

See also

Module `logging`
API reference for the logging module.

Module `logging.config`
Configuration API for the logging module.

Module `logging.handlers`
Useful handlers included with the logging module.

A logging cookbook

Logging Cookbook

Author:: Vinay Sajip <vinay_sajip at red-dove dot com>

This page contains a number of recipes related to logging, which have been found useful in the past. For links to tutorial and reference information, please see Other resources.

Using logging in multiple modules

Multiple calls to `logging.getLogger('someLogger')` return a reference to the same logger object. This is true not only within the same module, but also across modules as long as it is in the same Python interpreter process. It is true for references to the same object; additionally, application code can define and configure a parent logger in one module and create (but not configure) a child logger in a separate module, and all logger calls to the child will pass up to the parent. Here is a main module:

```
import logging
import auxiliary_module

# create logger with 'spam_application'
logger = logging.getLogger('spam_application')
logger.setLevel(logging.DEBUG)
# create file handler which logs even debug messages
fh = logging.FileHandler('spam.log')
fh.setLevel(logging.DEBUG)
# create console handler with a higher log level
ch = logging.StreamHandler()
ch.setLevel(logging.ERROR)
# create formatter and add it to the handlers
formatter = logging.Formatter('%(asctime)s - %(name)s - %
(levelname)s - %(message)s')
fh.setFormatter(formatter)
ch.setFormatter(formatter)
# add the handlers to the logger
logger.addHandler(fh)
logger.addHandler(ch)

logger.info('creating an instance of
auxiliary_module.Auxiliary')
a = auxiliary_module.Auxiliary()
logger.info('created an instance of
auxiliary_module.Auxiliary')
logger.info('calling auxiliary_module.Auxiliary.do_something')
a.do_something()
logger.info('finished auxiliary_module.Auxiliary.do_something')
logger.info('calling auxiliary_module.some_function()')
auxiliary_module.some_function()
logger.info('done with auxiliary_module.some_function()')
```

Here is the auxiliary module:

```
import logging

# create logger
module_logger = logging.getLogger('spam_application.auxiliary')

class Auxiliary:
    def __init__(self):
        self.logger =
logging.getLogger('spam_application.auxiliary.Auxiliary')
        self.logger.info('creating an instance of Auxiliary')

    def do_something(self):
        self.logger.info('doing something')
        a = 1 + 1
        self.logger.info('done doing something')

def some_function():
    module_logger.info('received a call to "some_function"')
```

The output looks like this:

```
2005-03-23 23:47:11,663 - spam_application - INFO -
   creating an instance of auxiliary_module.Auxiliary
2005-03-23 23:47:11,665 - spam_application.auxiliary.Auxiliary
- INFO -
   creating an instance of Auxiliary
2005-03-23 23:47:11,665 - spam_application - INFO -
   created an instance of auxiliary_module.Auxiliary
2005-03-23 23:47:11,668 - spam_application - INFO -
   calling auxiliary_module.Auxiliary.do_something
2005-03-23 23:47:11,668 - spam_application.auxiliary.Auxiliary
- INFO -
   doing something
2005-03-23 23:47:11,669 - spam_application.auxiliary.Auxiliary
- INFO -
   done doing something
2005-03-23 23:47:11,670 - spam_application - INFO -
   finished auxiliary_module.Auxiliary.do_something
2005-03-23 23:47:11,671 - spam_application - INFO -
   calling auxiliary_module.some_function()
2005-03-23 23:47:11,672 - spam_application.auxiliary - INFO -
   received a call to 'some_function'
2005-03-23 23:47:11,673 - spam_application - INFO -
   done with auxiliary_module.some_function()
```

Logging from multiple threads

Logging from multiple threads requires no special effort. The following example shows logging from the main (initial) thread and another thread:

```
import logging
import threading
import time

def worker(arg):
    while not arg['stop']:
        logging.debug('Hi from myfunc')
        time.sleep(0.5)

def main():
    logging.basicConfig(level=logging.DEBUG, format='%
(relativeCreated)6d %(threadName)s %(message)s')
    info = {'stop': False}
    thread = threading.Thread(target=worker, args=(info,))
    thread.start()
    while True:
        try:
            logging.debug('Hello from main')
            time.sleep(0.75)
        except KeyboardInterrupt:
            info['stop'] = True
            break
    thread.join()

if __name__ == '__main__':
    main()
```

When run, the script should print something like the following:

```
   0 Thread-1 Hi from myfunc
   3 MainThread Hello from main
 505 Thread-1 Hi from myfunc
 755 MainThread Hello from main
1007 Thread-1 Hi from myfunc
1507 MainThread Hello from main
1508 Thread-1 Hi from myfunc
2010 Thread-1 Hi from myfunc
2258 MainThread Hello from main
2512 Thread-1 Hi from myfunc
3009 MainThread Hello from main
3013 Thread-1 Hi from myfunc
3515 Thread-1 Hi from myfunc
3761 MainThread Hello from main
4017 Thread-1 Hi from myfunc
4513 MainThread Hello from main
4518 Thread-1 Hi from myfunc
```

This shows the logging output interspersed as one might expect. This approach works for more threads than shown here, of course.

Multiple handlers and formatters

Loggers are plain Python objects. The `addHandler()` method has no minimum or maximum quota for the number of handlers you may add. Sometimes it will be beneficial for an application to log all messages of all severities to a text file while simultaneously logging errors or above to the console. To set this up, simply configure the appropriate handlers. The logging calls in the application code will remain unchanged. Here is a slight modification to the previous simple module-based configuration example:

```
import logging

logger = logging.getLogger('simple_example')
logger.setLevel(logging.DEBUG)
# create file handler which logs even debug messages
fh = logging.FileHandler('spam.log')
fh.setLevel(logging.DEBUG)
# create console handler with a higher log level
ch = logging.StreamHandler()
ch.setLevel(logging.ERROR)
# create formatter and add it to the handlers
formatter = logging.Formatter('%(asctime)s - %(name)s - %
(levelname)s - %(message)s')
ch.setFormatter(formatter)
fh.setFormatter(formatter)
# add the handlers to logger
logger.addHandler(ch)
logger.addHandler(fh)

# 'application' code
logger.debug('debug message')
logger.info('info message')
logger.warning('warn message')
logger.error('error message')
logger.critical('critical message')
```

Notice that the ‘application’ code does not care about multiple handlers. All that changed was the addition and configuration of a new handler named *fh*.

The ability to create new handlers with higher- or lower-severity filters can be very helpful when writing and testing an application. Instead of using many `print` statements for debugging, use `logger.debug`: Unlike the print statements, which you will have to delete or comment out later, the logger.debug statements can remain intact in the source code and remain dormant until you need them again. At that time, the only change that needs to happen is to modify the severity level of the logger and/or handler to debug.

Logging to multiple destinations

Let’s say you want to log to console and file with different message formats and in differing circumstances. Say you want to log messages with levels of DEBUG and higher to file, and those messages at level INFO and higher to the console. Let’s also assume that the file should contain timestamps, but the console messages should not. Here’s how you can achieve this:

```
import logging

# set up logging to file - see previous section for more
details
logging.basicConfig(level=logging.DEBUG,
                    format='%(asctime)s %(name)-12s %
(levelname)-8s %(message)s',
                    datefmt='%m-%d %H:%M',
                    filename='/tmp/myapp.log',
                    filemode='w')
# define a Handler which writes INFO messages or higher to the
sys.stderr
console = logging.StreamHandler()
console.setLevel(logging.INFO)
# set a format which is simpler for console use
```

```
formatter = logging.Formatter('%(name)-12s: %(levelname)-8s %
(message)s')
# tell the handler to use this format
console.setFormatter(formatter)
# add the handler to the root logger
logging.getLogger('').addHandler(console)

# Now, we can log to the root logger, or any other logger.
First the root...
logging.info('Jackdaws love my big sphinx of quartz.')

# Now, define a couple of other loggers which might represent
areas in your
# application:

logger1 = logging.getLogger('myapp.area1')
logger2 = logging.getLogger('myapp.area2')

logger1.debug('Quick zephyrs blow, vexing daft Jim.')
logger1.info('How quickly daft jumping zebras vex.')
logger2.warning('Jail zesty vixen who grabbed pay from quack.')
logger2.error('The five boxing wizards jump quickly.')
```

When you run this, on the console you will see

```
root        : INFO     Jackdaws love my big sphinx of quartz.
myapp.area1 : INFO     How quickly daft jumping zebras vex.
myapp.area2 : WARNING  Jail zesty vixen who grabbed pay from
quack.
myapp.area2 : ERROR    The five boxing wizards jump quickly.
```

and in the file you will see something like

```
10-22 22:19 root         INFO     Jackdaws love my big sphinx
of quartz.
10-22 22:19 myapp.area1  DEBUG    Quick zephyrs blow, vexing
daft Jim.
10-22 22:19 myapp.area1  INFO     How quickly daft jumping
zebras vex.
10-22 22:19 myapp.area2  WARNING  Jail zesty vixen who grabbed
pay from quack.
```

```
10-22 22:19 myapp.area2  ERROR    The five boxing wizards jump
quickly.
```

As you can see, the DEBUG message only shows up in the file. The other messages are sent to both destinations.

This example uses console and file handlers, but you can use any number and combination of handlers you choose.

Note that the above choice of log filename `/tmp/myapp.log` implies use of a standard location for temporary files on POSIX systems. On Windows, you may need to choose a different directory name for the log - just ensure that the directory exists and that you have the permissions to create and update files in it.

Custom handling of levels

Sometimes, you might want to do something slightly different from the standard handling of levels in handlers, where all levels above a threshold get processed by a handler. To do this, you need to use filters. Let’s look at a scenario where you want to arrange things as follows:

- Send messages of severity `INFO` and `WARNING` to `sys.stdout`
- Send messages of severity `ERROR` and above to `sys.stderr`
- Send messages of severity `DEBUG` and above to file `app.log`

Suppose you configure logging with the following JSON:

```
{
    "version": 1,
    "disable_existing_loggers": false,
    "formatters": {
        "simple": {
            "format": "%(levelname)-8s - %(message)s"
        }
    },
    "handlers": {
        "stdout": {
            "class": "logging.StreamHandler",
            "level": "INFO",
            "formatter": "simple",
            "stream": "ext://sys.stdout"
        },
        "stderr": {
            "class": "logging.StreamHandler",
            "level": "ERROR",
            "formatter": "simple",
            "stream": "ext://sys.stderr"
        },
        "file": {
            "class": "logging.FileHandler",
            "formatter": "simple",
            "filename": "app.log",
            "mode": "w"
        }
    },
    "root": {
        "level": "DEBUG",
        "handlers": [
            "stderr",
            "stdout",
            "file"
        ]
    }
}
```

This configuration does *almost* what we want, except that `sys.stdout` would show messages of severity `ERROR` and only events of this severity and higher will be tracked as well as `INFO` and `WARNING` messages. To prevent this, we can set up a filter which excludes those messages and add it to the relevant handler. This can be configured by adding a `filters` section parallel to `formatters` and `handlers`:

```
{
    "filters": {
        "warnings_and_below": {
            "()" : "__main__.filter_maker",
            "level": "WARNING"
        }
    }
}
```

and changing the section on the `stdout` handler to add it:

```
{
    "stdout": {
        "class": "logging.StreamHandler",
        "level": "INFO",
        "formatter": "simple",
        "stream": "ext://sys.stdout",
        "filters": ["warnings_and_below"]
    }
}
```

A filter is just a function, so we can define the `filter_maker` (a factory function) as follows:

```
def filter_maker(level):
    level = getattr(logging, level)

    def filter(record):
        return record.levelno <= level

    return filter
```

This converts the string argument passed in to a numeric level, and returns a function which only returns `True` if the level of the passed in record is at or below the specified level. Note that in this example I have defined the `filter_maker` in a test script `main.py` that I run from the command line, so its module will be `__main__` - hence the `__main__.filter_maker` in the filter configuration. You will need to change that if you define it in a different module.

With the filter added, we can run `main.py`, which in full is:

```
import json
import logging
import logging.config

CONFIG = '''
{
    "version": 1,
    "disable_existing_loggers": false,
    "formatters": {
        "simple": {
            "format": "%(levelname)-8s - %(message)s"
        }
    },
    "filters": {
        "warnings_and_below": {
            "()" : "__main__.filter_maker",
            "level": "WARNING"
        }
    },
    "handlers": {
        "stdout": {
            "class": "logging.StreamHandler",
            "level": "INFO",
            "formatter": "simple",
            "stream": "ext://sys.stdout",
            "filters": ["warnings_and_below"]
        },
        "stderr": {
            "class": "logging.StreamHandler",
            "level": "ERROR",
            "formatter": "simple",
            "stream": "ext://sys.stderr"
        },
        "file": {
            "class": "logging.FileHandler",
            "formatter": "simple",
            "filename": "app.log",
            "mode": "w"
        }
    },
    "root": {
        "level": "DEBUG",
        "handlers": [
            "stderr",
            "stdout",
            "file"
        ]
    }
}
'''

def filter_maker(level):
    level = getattr(logging, level)

    def filter(record):
        return record.levelno <= level

    return filter

logging.config.dictConfig(json.loads(CONFIG))
logging.debug('A DEBUG message')
logging.info('An INFO message')
logging.warning('A WARNING message')
logging.error('An ERROR message')
logging.critical('A CRITICAL message')
```

And after running it like this:

```
python main.py 2>stderr.log >stdout.log
```

We can see the results are as expected:

```
$ more *.log
::::::::::::::
app.log
::::::::::::::
DEBUG    - A DEBUG message
INFO     - An INFO message
WARNING  - A WARNING message
ERROR    - An ERROR message
CRITICAL - A CRITICAL message
::::::::::::::
stderr.log
::::::::::::::
ERROR    - An ERROR message
```

```
CRITICAL - A CRITICAL message
::::::::::::::
stdout.log
::::::::::::::
INFO     - An INFO message
WARNING  - A WARNING message
```

Configuration server example

Here is an example of a module using the logging configuration server:

```
import logging
import logging.config
import time
import os

# read initial config file
logging.config.fileConfig('logging.conf')

# create and start listener on port 9999
t = logging.config.listen(9999)
t.start()

logger = logging.getLogger('simpleExample')

try:
    # loop through logging calls to see the difference
    # new configurations make, until Ctrl+C is pressed
    while True:
        logger.debug('debug message')
        logger.info('info message')
        logger.warning('warn message')
        logger.error('error message')
        logger.critical('critical message')
        time.sleep(5)
except KeyboardInterrupt:
    # cleanup
    logging.config.stopListening()
    t.join()
```

And here is a script that takes a filename and sends that file to the server, properly preceded with the binary-encoded length, as the new logging configuration:

```
#!/usr/bin/env python
import socket, sys, struct

with open(sys.argv[1], 'rb') as f:
    data_to_send = f.read()

HOST = 'localhost'
PORT = 9999
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
print('connecting...')
s.connect((HOST, PORT))
print('sending config...')
s.send(struct.pack('>L', len(data_to_send)))
s.send(data_to_send)
s.close()
print('complete')
```

Dealing with handlers that block

Sometimes you have to get your logging handlers to do their work without blocking the thread you’re logging from. This is common in web applications, though of course it also occurs in other scenarios.

A common culprit which demonstrates sluggish behaviour is the `SMTPHandler`: sending emails can take a long time, for a number of reasons outside the developer’s control (for example, a poorly performing mail or network infrastructure). But almost any network-based handler can block: Even a `SocketHandler` operation may do a DNS query under the hood which is too slow (and this query can be deep in the socket library code, below the Python layer, and outside your control).

One solution is to use a two-part approach. For the first part, attach only a [`QueueHandler`] to those loggers which are accessed from performance-critical threads. They simply write to their queue, which can be sized to a large enough capacity or initialized with no upper bound to their size. The write to the queue will typically be accepted quickly, though you will probably need to catch the [`queue.Full`] exception as a precaution in your code. If you are a library developer who has performance-critical threads in their code, be sure to document this (together with a suggestion to attach only `QueueHandlers` to your loggers) for the benefit of other developers who will use your code.

The second part of the solution is [`QueueListener`], which has been designed as the counterpart to [`QueueHandler`]. A [`QueueListener`] is very simple: it’s passed a queue and some handlers, and it fires up an internal thread which listens to its queue for LogRecords sent from `QueueHandlers` (or any other source of `LogRecords`, for that matter). The `LogRecords` are removed from the queue and passed to the handlers for processing.

The advantage of having a separate [`QueueListener`] class is that you can use the same instance to service multiple `QueueHandlers`. This is more resource-friendly than, say, having threaded versions of the existing handler classes, which would eat up one thread per handler for no particular benefit.

An example of using these two classes follows (imports omitted):

```
que = queue.Queue(-1)  # no limit on size
queue_handler = QueueHandler(que)
handler = logging.StreamHandler()
listener = QueueListener(que, handler)
root = logging.getLogger()
```

```
root.addHandler(queue_handler)
formatter = logging.Formatter('%(threadName)s: %(message)s')
handler.setFormatter(formatter)
listener.start()
# The log output will display the thread which generated
# the event (the main thread) rather than the internal
# thread which monitors the internal queue. This is what
# you want to happen.
root.warning('Look out!')
listener.stop()
```

which, when run, will produce:

```
MainThread: Look out!
```

Note

Although the earlier discussion wasn't specifically talking about async code, but rather about slow logging handlers, it should be noted that when logging from async code, network and even file handlers could lead to problems (blocking the event loop) because some logging is done from **`asyncio`** internals. It might be best, if any async code is used in an application, to use the above approach for logging, so that any blocking code runs only in the `QueueListener` thread.

Changed in version 3.5: Prior to Python 3.5, the **`QueueListener`** always passed every message received from the queue to every handler it was initialized with. (This was because it was assumed that level filtering was all done on the other side, where the queue is filled.) From 3.5 onwards, this behaviour can be changed by passing a keyword argument `respect_handler_level=True` to the listener's constructor. When this is done, the listener compares the level of each message with the handler's level, and only passes a message to a handler if it's appropriate to do so.

Changed in version 3.14: The **`QueueListener`** can be started (and stopped) via the **`with`** statement. For example:

```
with QueueListener(que, handler) as listener:
    # The queue listener automatically starts
    # when the 'with' block is entered.
    pass
# The queue listener automatically stops once
# the 'with' block is exited.
```

Sending and receiving logging events across a network

Let’s say you want to send logging events across a network, and handle them at the receiving end. A simple way of doing this is attaching a **`SocketHandler`** instance to the root logger at the sending end:

```
import logging, logging.handlers

rootLogger = logging.getLogger('')
rootLogger.setLevel(logging.DEBUG)
socketHandler = logging.handlers.SocketHandler('localhost',
                    logging.handlers.DEFAULT_TCP_LOGGING_PORT)
# don't bother with a formatter, since a socket handler sends
the event as
# an unformatted pickle
rootLogger.addHandler(socketHandler)

# Now, we can log to the root logger, or any other logger.
First the root...
logging.info('Jackdaws love my big sphinx of quartz.')

# Now, define a couple of other loggers which might represent
areas in your
# application:

logger1 = logging.getLogger('myapp.area1')
logger2 = logging.getLogger('myapp.area2')
```

```
logger1.debug('Quick zephyrs blow, vexing daft Jim.')
logger1.info('How quickly daft jumping zebras vex.')
logger2.warning('Jail zesty vixen who grabbed pay from quack.')
logger2.error('The five boxing wizards jump quickly.')
```

At the receiving end, you can set up a receiver using the `socketserver` module. Here is a basic working example:

```
import pickle
import logging
import logging.handlers
import socketserver
import struct


class
LogRecordStreamHandler(socketserver.StreamRequestHandler):
    """Handler for a streaming logging request.

    This basically logs the record using whatever logging
policy is
    configured locally.
    """

    def handle(self):
        """
        Handle multiple requests - each expected to be a 4-byte
length,
        followed by the LogRecord in pickle format. Logs the
record
        according to whatever policy is configured locally.
        """
        while True:
            chunk = self.connection.recv(4)
            if len(chunk) < 4:
                break
            slen = struct.unpack('>L', chunk)[0]
            chunk = self.connection.recv(slen)
            while len(chunk) < slen:
                chunk = chunk + self.connection.recv(slen -
len(chunk))
            obj = self.unPickle(chunk)
            record = logging.makeLogRecord(obj)
            self.handleLogRecord(record)

    def unPickle(self, data):
        return pickle.loads(data)

    def handleLogRecord(self, record):
        # if a name is specified, we use the named logger
rather than the one
        # implied by the record.
        if self.server.logname is not None:
            name = self.server.logname
        else:
            name = record.name
        logger = logging.getLogger(name)
        # N.B. EVERY record gets logged. This is because
Logger.handle
        # is normally called AFTER logger-level filtering. If
you want
        # to do filtering, do it at the client end to save
wasting
        # cycles and network bandwidth!
        logger.handle(record)

class LogRecordSocketReceiver(socketserver.ThreadingTCPServer):
    """
    Simple TCP socket-based logging receiver suitable for
testing.
    """

    allow_reuse_address = True

    def __init__(self, host='localhost',

port=logging.handlers.DEFAULT_TCP_LOGGING_PORT,
                 handler=LogRecordStreamHandler):
        socketserver.ThreadingTCPServer.__init__(self, (host,
port), handler)
        self.abort = 0
        self.timeout = 1
        self.logname = None

    def serve_until_stopped(self):
        import select
        abort = 0
        while not abort:
            rd, wr, ex = select.select([self.socket.fileno()],
                                       [], [],
                                       self.timeout)
            if rd:
                self.handle_request()
            abort = self.abort

def main():
    logging.basicConfig(
        format='%(relativeCreated)5d %(name)-15s %
(levelname)-8s %(message)s')
    tcpserver = LogRecordSocketReceiver()
    print('About to start TCP server...')
    tcpserver.serve_until_stopped()

if __name__ == '__main__':
    main()
```

First run the server, and then the client. On the client side, nothing is printed on the console; on the server side, you should see something like:

```
About to start TCP server...
   59 root            INFO     Jackdaws love my big sphinx of
quartz.
   59 myapp.area1     DEBUG    Quick zephyrs blow, vexing daft
Jim.
   69 myapp.area1     INFO     How quickly daft jumping zebras
vex.
   69 myapp.area2     WARNING  Jail zesty vixen who grabbed pay
from quack.
   69 myapp.area2     ERROR    The five boxing wizards jump
quickly.
```

Note that there are some security issues with pickle in some scenarios. If these affect you, you can use an alternative serialization scheme by overriding the `makePickle()` method and implementing your alternative there, as well as adapting the above script to use your alternative serialization.

Running a logging socket listener in production

To run a logging listener in production, you may need to use a process-management tool such as Supervisor [http://supervisord.org/]. Here is a Gist [https://gist.github.com/vsajip/4b227eeec43817465ca835ca66f75e2b] which provides the bare-bones files to run the above functionality using Supervisor. It consists of the following files:

File	**Purpose**
`prepare.sh`	A Bash script to prepare the environment for testing
`supervisor.conf`	The Supervisor configuration file, which has entries for the listener and a multi-process web application
`ensure_app.sh`	A Bash script to ensure that Supervisor is running with the above configuration
`log_listener.py`	The socket listener program which receives log events and records them to a file
`main.py`	A simple web application which performs logging via a socket connected to the listener

File	**Purpose**
`webapp.json`	A JSON configuration file for the web application
`client.py`	A Python script to exercise the web application

The web application uses Gunicorn [https://gunicorn.org/], which is a popular web application server that starts multiple worker processes to handle requests. This example setup shows how the workers can write to the same log file without conflicting with one another — they all go through the socket listener.

To test these files, do the following in a POSIX environment:

1. Download the Gist [https://gist.github.com/vsajip/4b227eeec43817465ca835ca66f75e2b] as a ZIP archive using the *Download ZIP* button.
2. Unzip the above files from the archive into a scratch directory.
3. In the scratch directory, run `bash prepare.sh` to get things ready. This creates a `run` subdirectory to contain Supervisor-related and log files, and a `venv` subdirectory to contain a virtual environment into which `bottle`, `gunicorn` and `supervisor` are installed.
4. Run `bash ensure_app.sh` to ensure that Supervisor is running with the above configuration.
5. Run `venv/bin/python client.py` to exercise the web application, which will lead to records being written to the log.

6. Inspect the log files in the `run` subdirectory. You should see the most recent log lines in files matching the pattern `app.log*`. They won't be in any particular order, since they have been handled concurrently by different worker processes in a non-deterministic way.
7. You can shut down the listener and the web application by running `venv/bin/supervisorctl -c supervisor.conf shutdown`.

You may need to tweak the configuration files in the unlikely event that the configured ports clash with something else in your test environment.

The default configuration uses a TCP socket on port 9020. You can use a Unix Domain socket instead of a TCP socket by doing the following:

1. In `listener.json`, add a `socket` key with the path to the domain socket you want to use. If this key is present, the listener listens on the corresponding domain socket and not on a TCP socket (the `port` key is ignored).
2. In `webapp.json`, change the socket handler configuration dictionary so that the `host` value is the path to the domain socket, and set the `port` value to `null`.

Adding contextual information to your logging output

Sometimes you want logging output to contain contextual information in addition to the parameters passed to the logging call. For example, in a networked application, it may be desirable to log client-specific information in the log (e.g. remote client's username, or IP address). Although you could use the *extra* parameter to achieve this, it's not always convenient to pass the information in this way. While it might be tempting to create `Logger` instances on a per-connection basis, this is not a good idea because these instances are not garbage collected. While this is not a problem in practice, when the number of `Logger` instances is dependent on the level of granularity you want to use in logging an application, it could be hard to manage if the number of `Logger` instances becomes effectively unbounded.

Using LoggerAdapters to impart contextual information

An easy way in which you can pass contextual information to be output along with logging event information is to use the `LoggerAdapter` class. This class is designed to look like a `Logger`, so that you can call `debug()`, `info()`, `warning()`, `error()`, `exception()`, `critical()` and `log()`. These methods have the same signatures as their counterparts in `Logger`, so you can use the two types of instances interchangeably.

When you create an instance of `LoggerAdapter`, you pass it a `Logger` instance and a dict-like object which contains your contextual information. When you call one of the logging methods on an instance of `LoggerAdapter`, it delegates the call to the underlying instance of `Logger` passed to its constructor, and arranges to pass the contextual information in the delegated call. Here’s a snippet from the code of `LoggerAdapter`:

```
def debug(self, msg, /, *args, **kwargs):
    """
    Delegate a debug call to the underlying logger, after
adding
    contextual information from this adapter instance.
    """
    msg, kwargs = self.process(msg, kwargs)
    self.logger.debug(msg, *args, **kwargs)
```

The `process()` method of `LoggerAdapter` is where the contextual information is added to the logging output. It's passed the message and keyword arguments of the logging call, and it passes back (potentially) modified versions of these to use in the call to the underlying logger. The default implementation of this method leaves the message alone, but inserts an 'extra' key in the keyword argument whose value is the dict-like object passed to the constructor. Of course, if you had passed an 'extra' keyword argument in the call to the adapter, it will be silently overwritten.

The advantage of using 'extra' is that the values in the dict-like object are merged into the `LogRecord` instance's __dict__, allowing you to use customized strings with your `Formatter` instances which know about the keys of the dict-like object. If you need a different method, e.g. if you want to prepend or append the contextual information to the message string, you just need to subclass `LoggerAdapter` and override `process()` to do what you need. Here is a simple example:

```
class CustomAdapter(logging.LoggerAdapter):
    """
    This example adapter expects the passed in dict-like object
to have a
    'connid' key, whose value in brackets is prepended to the
log message.
    """
    def process(self, msg, kwargs):
        return '[%s] %s' % (self.extra['connid'], msg), kwargs
```

which you can use like this:

```
logger = logging.getLogger(__name__)
adapter = CustomAdapter(logger, {'connid': some_conn_id})
```

Then any events that you log to the adapter will have the value of `some_conn_id` prepended to the log messages.

Using objects other than dicts to pass contextual information

You don’t need to pass an actual dict to a **`LoggerAdapter`** - you could pass an instance of a class which implements `__getitem__` and `__iter__` so that it looks like a dict to logging. This would be useful if you want to generate values dynamically (whereas the values in a dict would be constant).

Using Filters to impart contextual information

You can also add contextual information to log output using a user-defined **`Filter`**. `Filter` instances are allowed to modify the `LogRecords` passed to them, including adding additional attributes which can then be output using a suitable format string, or if needed a custom **`Formatter`**.

For example in a web application, the request being processed (or at least, the interesting parts of it) can be stored in a threadlocal (**`threading.local`**) variable, and then accessed from a `Filter` to add, say, information from the request - say, the remote IP address and remote user’s username - to the `LogRecord`, using the attribute names ‘ip’ and ‘user’ as in the `LoggerAdapter` example above. In that case, the same format string can be used to get similar output to that shown above. Here’s an example script:

```
import logging
from random import choice

class ContextFilter(logging.Filter):
    """
    This is a filter which injects contextual information into
the log.

    Rather than use actual contextual information, we just use
```

```
random
    data in this demo.
    """

    USERS = ['jim', 'fred', 'sheila']
    IPS = ['123.231.231.123', '127.0.0.1', '192.168.0.1']

    def filter(self, record):

        record.ip = choice(ContextFilter.IPS)
        record.user = choice(ContextFilter.USERS)
        return True

if __name__ == '__main__':
    levels = (logging.DEBUG, logging.INFO, logging.WARNING,
logging.ERROR, logging.CRITICAL)
    logging.basicConfig(level=logging.DEBUG,
                        format='%(asctime)-15s %(name)-5s %
(levelname)-8s IP: %(ip)-15s User: %(user)-8s %(message)s')
    a1 = logging.getLogger('a.b.c')
    a2 = logging.getLogger('d.e.f')

    f = ContextFilter()
    a1.addFilter(f)
    a2.addFilter(f)
    a1.debug('A debug message')
    a1.info('An info message with %s', 'some parameters')
    for x in range(10):
        lvl = choice(levels)
        lvlname = logging.getLevelName(lvl)
        a2.log(lvl, 'A message at %s level with %d %s',
lvlname, 2, 'parameters')
```

which, when run, produces something like:

```
2010-09-06 22:38:15,292 a.b.c DEBUG    IP: 123.231.231.123
User: fred     A debug message
2010-09-06 22:38:15,300 a.b.c INFO     IP: 192.168.0.1
User: sheila   An info message with some parameters
2010-09-06 22:38:15,300 d.e.f CRITICAL IP: 127.0.0.1
User: sheila   A message at CRITICAL level with 2 parameters
2010-09-06 22:38:15,300 d.e.f ERROR    IP: 127.0.0.1
User: jim      A message at ERROR level with 2 parameters
```

```
2010-09-06 22:38:15,300 d.e.f DEBUG    IP: 127.0.0.1
User: sheila   A message at DEBUG level with 2 parameters
2010-09-06 22:38:15,300 d.e.f ERROR    IP: 123.231.231.123
User: fred     A message at ERROR level with 2 parameters
2010-09-06 22:38:15,300 d.e.f CRITICAL IP: 192.168.0.1
User: jim      A message at CRITICAL level with 2 parameters
2010-09-06 22:38:15,300 d.e.f CRITICAL IP: 127.0.0.1
User: sheila   A message at CRITICAL level with 2 parameters
2010-09-06 22:38:15,300 d.e.f DEBUG    IP: 192.168.0.1
User: jim      A message at DEBUG level with 2 parameters
2010-09-06 22:38:15,301 d.e.f ERROR    IP: 127.0.0.1
User: sheila   A message at ERROR level with 2 parameters
2010-09-06 22:38:15,301 d.e.f DEBUG    IP: 123.231.231.123
User: fred     A message at DEBUG level with 2 parameters
2010-09-06 22:38:15,301 d.e.f INFO     IP: 123.231.231.123
User: fred     A message at INFO level with 2 parameters
```

Use of `contextvars`

Since Python 3.7, the `contextvars` module has provided context-local storage which works for both `threading` and `asyncio` processing needs. This type of storage may thus be generally preferable to thread-locals. The following example shows how, in a multi-threaded environment, logs can populated with contextual information such as, for example, request attributes handled by web applications.

For the purposes of illustration, say that you have different web applications, each independent of the other but running in the same Python process and using a library common to them. How can each of these applications have their own log, where all logging messages from the library (and other request processing code) are directed to the appropriate application’s log file, while including in the log additional contextual information such as client IP, HTTP request method and client username?

Let’s assume that the library can be simulated by the following code:

```
# webapplib.py
import logging
import time

logger = logging.getLogger(__name__)

def useful():
    # Just a representative event logged from the library
    logger.debug('Hello from webapplib!')
    # Just sleep for a bit so other threads get to run
    time.sleep(0.01)
```

We can simulate the multiple web applications by means of two simple classes, `Request` and `WebApp`. These simulate how real threaded web applications work - each request is handled by a thread:

```
# main.py
import argparse
from contextvars import ContextVar
import logging
import os
from random import choice
import threading
import webapplib

logger = logging.getLogger(__name__)
root = logging.getLogger()
root.setLevel(logging.DEBUG)

class Request:
    """
    A simple dummy request class which just holds dummy HTTP
request method,
    client IP address and client username
    """
    def __init__(self, method, ip, user):
        self.method = method
        self.ip = ip
        self.user = user

# A dummy set of requests which will be used in the simulation
- we'll just pick
# from this list randomly. Note that all GET requests are from
192.168.2.XXX
# addresses, whereas POST requests are from 192.16.3.XXX
addresses. Three users
# are represented in the sample requests.

REQUESTS = [
    Request('GET', '192.168.2.20', 'jim'),
    Request('POST', '192.168.3.20', 'fred'),
    Request('GET', '192.168.2.21', 'sheila'),
    Request('POST', '192.168.3.21', 'jim'),
    Request('GET', '192.168.2.22', 'fred'),
    Request('POST', '192.168.3.22', 'sheila'),
]

# Note that the format string includes references to request
context information
# such as HTTP method, client IP and username

formatter = logging.Formatter('%(threadName)-11s %(appName)s %
(name)-9s %(user)-6s %(ip)s %(method)-4s %(message)s')

# Create our context variables. These will be filled at the
start of request
# processing, and used in the logging that happens during that
processing

ctx_request = ContextVar('request')
ctx_appname = ContextVar('appname')

class InjectingFilter(logging.Filter):
    """
    A filter which injects context-specific information into
logs and ensures
    that only information for a specific webapp is included in
its log
    """
    def __init__(self, app):
        self.app = app
    def filter(self, record):
        request = ctx_request.get()
        record.method = request.method
        record.ip = request.ip
        record.user = request.user
        record.appName = appName = ctx_appname.get()
        return appName == self.app.name

class WebApp:
    """
    A dummy web application class which has its own handler and
filter for a
    webapp-specific log.
    """
    def __init__(self, name):
        self.name = name
        handler = logging.FileHandler(name + '.log', 'w')
        f = InjectingFilter(self)
        handler.setFormatter(formatter)
        handler.addFilter(f)
        root.addHandler(handler)
        self.num_requests = 0

    def process_request(self, request):
        """
        This is the dummy method for processing a request. It's
called on a
        different thread for every request. We store the
context information into
        the context vars before doing anything else.
        """
        ctx_request.set(request)
        ctx_appname.set(self.name)
        self.num_requests += 1
        logger.debug('Request processing started')
        webapplib.useful()
        logger.debug('Request processing finished')

def main():
    fn = os.path.splitext(os.path.basename(__file__))[0]
    adhf = argparse.ArgumentDefaultsHelpFormatter
    ap = argparse.ArgumentParser(formatter_class=adhf, prog=fn,
                                    description='Simulate a couple
of web '
                                                'applications
handling some '
                                                'requests, showing
how request '
                                                'context can be
used to '
                                                'populate logs')
    aa = ap.add_argument
    aa('--count', '-c', type=int, default=100, help='How many
requests to simulate')
    options = ap.parse_args()

    # Create the dummy webapps and put them in a list which we
can use to select
    # from randomly
    app1 = WebApp('app1')
    app2 = WebApp('app2')
    apps = [app1, app2]
    threads = []
    # Add a common handler which will capture all events
    handler = logging.FileHandler('app.log', 'w')
    handler.setFormatter(formatter)
    root.addHandler(handler)

    # Generate calls to process requests
    for i in range(options.count):
        try:
            # Pick an app at random and a request for it to
process
            app = choice(apps)
            request = choice(REQUESTS)
            # Process the request in its own thread
            t = threading.Thread(target=app.process_request,
args=(request,))
            threads.append(t)
            t.start()
        except KeyboardInterrupt:
            break

    # Wait for the threads to terminate
    for t in threads:
        t.join()

    for app in apps:
        print('%s processed %s requests' % (app.name,
app.num_requests))

if __name__ == '__main__':
    main()
```

If you run the above, you should find that roughly half the requests go into `app1.log` and the rest into `app2.log`, and the all the requests are logged to `app.log`. Each webapp-specific log will contain only log entries for only that webapp, and the request information will be displayed consistently in the log (i.e. the information in each dummy request will always appear together in a log line). This is illustrated by the following shell output:

```
~/logging-contextual-webapp$ python main.py
app1 processed 51 requests
app2 processed 49 requests
~/logging-contextual-webapp$ wc -l *.log
  153 app1.log
  147 app2.log
  300 app.log
  600 total
~/logging-contextual-webapp$ head -3 app1.log
Thread-3 (process_request) app1 __main__  jim    192.168.3.21
POST Request processing started
Thread-3 (process_request) app1 webapplib jim    192.168.3.21
POST Hello from webapplib!
Thread-5 (process_request) app1 __main__  jim    192.168.3.21
POST Request processing started
~/logging-contextual-webapp$ head -3 app2.log
Thread-1 (process_request) app2 __main__  sheila 192.168.2.21
GET  Request processing started
Thread-1 (process_request) app2 webapplib sheila 192.168.2.21
GET  Hello from webapplib!
Thread-2 (process_request) app2 __main__  jim    192.168.2.20
GET  Request processing started
~/logging-contextual-webapp$ head app.log
```

```
Thread-1 (process_request) app2 __main__  sheila 192.168.2.21
GET  Request processing started
Thread-1 (process_request) app2 webapplib sheila 192.168.2.21
GET  Hello from webapplib!
Thread-2 (process_request) app2 __main__  jim    192.168.2.20
GET  Request processing started
Thread-3 (process_request) app1 __main__  jim    192.168.3.21
POST Request processing started
Thread-2 (process_request) app2 webapplib jim    192.168.2.20
GET  Hello from webapplib!
Thread-3 (process_request) app1 webapplib jim    192.168.3.21
POST Hello from webapplib!
Thread-4 (process_request) app2 __main__  fred   192.168.2.22
GET  Request processing started
Thread-5 (process_request) app1 __main__  jim    192.168.3.21
POST Request processing started
Thread-4 (process_request) app2 webapplib fred   192.168.2.22
GET  Hello from webapplib!
Thread-6 (process_request) app1 __main__  jim    192.168.3.21
POST Request processing started
~/logging-contextual-webapp$ grep app1 app1.log | wc -l
153
~/logging-contextual-webapp$ grep app2 app2.log | wc -l
147
~/logging-contextual-webapp$ grep app1 app.log | wc -l
153
~/logging-contextual-webapp$ grep app2 app.log | wc -l
147
```

Imparting contextual information in handlers

Each `Handler` has its own chain of filters. If you want to add contextual information to a `LogRecord` without leaking it to other handlers, you can use a filter that returns a new `LogRecord` instead of modifying it in-place, as shown in the following script:

```
import copy
import logging
```

```
def filter(record: logging.LogRecord):
    record = copy.copy(record)
    record.user = 'jim'
    return record

if __name__ == '__main__':
    logger = logging.getLogger()
    logger.setLevel(logging.INFO)
    handler = logging.StreamHandler()
    formatter = logging.Formatter('%(message)s from %
(user)-8s')
    handler.setFormatter(formatter)
    handler.addFilter(filter)
    logger.addHandler(handler)

    logger.info('A log message')
```

Logging to a single file from multiple processes

Although logging is thread-safe, and logging to a single file from multiple threads in a single process *is* supported, logging to a single file from *multiple processes* is *not* supported, because there is no standard way to serialize access to a single file across multiple processes in Python. If you need to log to a single file from multiple processes, one way of doing this is to have all the processes log to a `SocketHandler`, and have a separate process which implements a socket server which reads from the socket and logs to file. (If you prefer, you can dedicate one thread in one of the existing processes to perform this function.) This section documents this approach in more detail and includes a working socket receiver which can be used as a starting point for you to adapt in your own applications.

You could also write your own handler which uses the `Lock` class from the `multiprocessing` module to serialize access to the file from your processes. The stdlib **`FileHandler`** and subclasses do not make use of **`multiprocessing`**.

Alternatively, you can use a `Queue` and a **`QueueHandler`** to send all logging events to one of the processes in your multi-process application. The following example script demonstrates how you can do this; in the example a separate listener process listens for events sent by other processes and logs them according to its own logging configuration. Although the example only demonstrates one way of doing it (for example, you may want to use a listener thread rather than a separate listener process – the implementation would be analogous) it does allow for completely different logging configurations for the listener and the other processes in your application, and can be used as the basis for code meeting your own specific requirements:

```
# You'll need these imports in your own code
import logging
import logging.handlers
import multiprocessing

# Next two import lines for this demo only
from random import choice, random
import time

#
# Because you'll want to define the logging configurations for
listener and workers, the
# listener and worker process functions take a configurer
parameter which is a callable
# for configuring logging for that process. These functions are
also passed the queue,
# which they use for communication.
#
# In practice, you can configure the listener however you want,
but note that in this
# simple example, the listener does not apply level or filter
logic to received records.
```

```
# In practice, you would probably want to do this logic in the
worker processes, to avoid
# sending events which would be filtered out between processes.
#
# The size of the rotated files is made small so you can see
the results easily.
def listener_configurer():
    root = logging.getLogger()
    h = logging.handlers.RotatingFileHandler('mptest.log', 'a',
300, 10)
    f = logging.Formatter('%(asctime)s %(processName)-10s %
(name)s %(levelname)-8s %(message)s')
    h.setFormatter(f)
    root.addHandler(h)

# This is the listener process top-level loop: wait for logging
events
# (LogRecords)on the queue and handle them, quit when you get a
None for a
# LogRecord.
def listener_process(queue, configurer):
    configurer()
    while True:
        try:
            record = queue.get()
            if record is None:  # We send this as a sentinel to
tell the listener to quit.
                break
            logger = logging.getLogger(record.name)
            logger.handle(record)  # No level or filter logic
applied - just do it!
        except Exception:
            import sys, traceback
            print('Whoops! Problem:', file=sys.stderr)
            traceback.print_exc(file=sys.stderr)

# Arrays used for random selections in this demo

LEVELS = [logging.DEBUG, logging.INFO, logging.WARNING,
          logging.ERROR, logging.CRITICAL]

LOGGERS = ['a.b.c', 'd.e.f']
```

```
MESSAGES = [
    'Random message #1',
    'Random message #2',
    'Random message #3',
]

# The worker configuration is done at the start of the worker
process run.
# Note that on Windows you can't rely on fork semantics, so
each process
# will run the logging configuration code when it starts.
def worker_configurer(queue):
    h = logging.handlers.QueueHandler(queue)  # Just the one
handler needed
    root = logging.getLogger()
    root.addHandler(h)
    # send all messages, for demo; no other level or filter
logic applied.
    root.setLevel(logging.DEBUG)

# This is the worker process top-level loop, which just logs
ten events with
# random intervening delays before terminating.
# The print messages are just so you know it's doing something!
def worker_process(queue, configurer):
    configurer(queue)
    name = multiprocessing.current_process().name
    print('Worker started: %s' % name)
    for i in range(10):
        time.sleep(random())
        logger = logging.getLogger(choice(LOGGERS))
        level = choice(LEVELS)
        message = choice(MESSAGES)
        logger.log(level, message)
    print('Worker finished: %s' % name)

# Here's where the demo gets orchestrated. Create the queue,
create and start
# the listener, create ten workers and start them, wait for
them to finish,
# then send a None to the queue to tell the listener to finish.
def main():
    queue = multiprocessing.Queue(-1)
    listener = multiprocessing.Process(target=listener_process,
                                       args=(queue,
listener_configurer))
    listener.start()
    workers = []
    for i in range(10):
        worker = multiprocessing.Process(target=worker_process,
                                         args=(queue,
worker_configurer))
        workers.append(worker)
        worker.start()
    for w in workers:
        w.join()
    queue.put_nowait(None)
    listener.join()

if __name__ == '__main__':
    main()
```

A variant of the above script keeps the logging in the main process, in a separate thread:

```
import logging
import logging.config
import logging.handlers
from multiprocessing import Process, Queue
import random
import threading
import time

def logger_thread(q):
    while True:
        record = q.get()
        if record is None:
            break
        logger = logging.getLogger(record.name)
        logger.handle(record)


def worker_process(q):
    qh = logging.handlers.QueueHandler(q)
    root = logging.getLogger()
    root.setLevel(logging.DEBUG)
    root.addHandler(qh)
    levels = [logging.DEBUG, logging.INFO, logging.WARNING,
logging.ERROR,
              logging.CRITICAL]
    loggers = ['foo', 'foo.bar', 'foo.bar.baz',
               'spam', 'spam.ham', 'spam.ham.eggs']
    for i in range(100):
        lvl = random.choice(levels)
        logger = logging.getLogger(random.choice(loggers))
        logger.log(lvl, 'Message no. %d', i)

if __name__ == '__main__':
    q = Queue()
    d = {
        'version': 1,
        'formatters': {
            'detailed': {
                'class': 'logging.Formatter',
                'format': '%(asctime)s %(name)-15s %
(levelname)-8s %(processName)-10s %(message)s'
            }
        },
        'handlers': {
            'console': {
                'class': 'logging.StreamHandler',
                'level': 'INFO',
            },
            'file': {
                'class': 'logging.FileHandler',
                'filename': 'mplog.log',
                'mode': 'w',
                'formatter': 'detailed',
            },
            'foofile': {
                'class': 'logging.FileHandler',
                'filename': 'mplog-foo.log',
                'mode': 'w',
                'formatter': 'detailed',
            },
            'errors': {
                'class': 'logging.FileHandler',
                'filename': 'mplog-errors.log',
                'mode': 'w',
                'level': 'ERROR',
                'formatter': 'detailed',
            },
        },
        'loggers': {
            'foo': {
                'handlers': ['foofile']
            }
        },
        'root': {
            'level': 'DEBUG',
            'handlers': ['console', 'file', 'errors']
        },
    }
    workers = []
    for i in range(5):
        wp = Process(target=worker_process, name='worker %d' %
(i + 1), args=(q,))
        workers.append(wp)
        wp.start()
    logging.config.dictConfig(d)
    lp = threading.Thread(target=logger_thread, args=(q,))
    lp.start()
    # At this point, the main process could do some useful work
of its own
    # Once it's done that, it can wait for the workers to
terminate...
    for wp in workers:
        wp.join()
    # And now tell the logging thread to finish up, too
    q.put(None)
    lp.join()
```

This variant shows how you can e.g. apply configuration for particular loggers - e.g. the `foo` logger has a special handler which stores all events in the `foo` subsystem in a file `mplog-foo.log`. This will be used by the logging machinery in the main process (even though the logging events are generated in the worker processes) to direct the messages to the appropriate destinations.

Using concurrent.futures.ProcessPoolExecutor

If you want to use `concurrent.futures.ProcessPoolExecutor` to start your worker processes, you need to create the queue slightly differently. Instead of

```
queue = multiprocessing.Queue(-1)
```

you should use

```
queue = multiprocessing.Manager().Queue(-1)  # also works with
the examples above
```

and you can then replace the worker creation from this:

```
workers = []
for i in range(10):
    worker = multiprocessing.Process(target=worker_process,
                                     args=(queue,
worker_configurer))
    workers.append(worker)
    worker.start()
for w in workers:
    w.join()
```

to this (remembering to first import `concurrent.futures`):

```
with concurrent.futures.ProcessPoolExecutor(max_workers=10) as
executor:
    for i in range(10):
        executor.submit(worker_process, queue,
worker_configurer)
```

Deploying Web applications using Gunicorn and uWSGI

When deploying Web applications using Gunicorn [https://gunicorn.org/] or uWSGI [https://uwsgi-docs.readthedocs.io/en/latest/] (or similar), multiple worker processes are created to handle client requests. In such environments, avoid creating file-based handlers directly in your web application. Instead, use a **`SocketHandler`** to log from the web application to a listener in a separate process. This can be set up using a process management tool such as Supervisor - see Running a logging socket listener in production for more details.

Using file rotation

Sometimes you want to let a log file grow to a certain size, then open a new file and log to that. You may want to keep a certain number of these files, and when that many files have been created, rotate the files so that the number of files and the size of the files both remain bounded. For this usage pattern, the logging package provides a **`RotatingFileHandler`**:

```
import glob
import logging
import logging.handlers

LOG_FILENAME = 'logging_rotatingfile_example.out'

# Set up a specific logger with our desired output level
my_logger = logging.getLogger('MyLogger')
my_logger.setLevel(logging.DEBUG)

# Add the log message handler to the logger
handler = logging.handlers.RotatingFileHandler(
              LOG_FILENAME, maxBytes=20, backupCount=5)

my_logger.addHandler(handler)

# Log some messages
for i in range(20):
    my_logger.debug('i = %d' % i)

# See what files are created
```

```
logfiles = glob.glob('%s*' % LOG_FILENAME)

for filename in logfiles:
    print(filename)
```

The result should be 6 separate files, each with part of the log history for the application:

```
logging_rotatingfile_example.out
logging_rotatingfile_example.out.1
logging_rotatingfile_example.out.2
logging_rotatingfile_example.out.3
logging_rotatingfile_example.out.4
logging_rotatingfile_example.out.5
```

The most current file is always `logging_rotatingfile_example.out`, and each time it reaches the size limit it is renamed with the suffix `.1`. Each of the existing backup files is renamed to increment the suffix (`.1` becomes `.2`, etc.) and the `.6` file is erased.

Obviously this example sets the log length much too small as an extreme example. You would want to set *maxBytes* to an appropriate value.

Use of alternative formatting styles

When logging was added to the Python standard library, the only way of formatting messages with variable content was to use the %-formatting method. Since then, Python has gained two new formatting approaches: **`string.Template`** (added in Python 2.4) and **`str.format()`** (added in Python 2.6).

Logging (as of 3.2) provides improved support for these two additional formatting styles. The **`Formatter`** class been enhanced to take an additional, optional keyword parameter named `style`. This defaults to `'%'`, but other possible values are `'{'` and `'$'`, which correspond to the other two formatting styles. Backwards compatibility is maintained by default (as you would expect), but by explicitly specifying a style parameter, you get the ability to specify format strings which work with **`str.format()`** or **`string.Template`**. Here’s an example console session to show the possibilities:

```
>>> import logging
>>> root = logging.getLogger()
>>> root.setLevel(logging.DEBUG)
>>> handler = logging.StreamHandler()
>>> bf = logging.Formatter('{asctime} {name} {levelname:8s}
{message}',
...                        style='{')
>>> handler.setFormatter(bf)
>>> root.addHandler(handler)
>>> logger = logging.getLogger('foo.bar')
>>> logger.debug('This is a DEBUG message')
2010-10-28 15:11:55,341 foo.bar DEBUG    This is a DEBUG
message
>>> logger.critical('This is a CRITICAL message')
2010-10-28 15:12:11,526 foo.bar CRITICAL This is a CRITICAL
message
>>> df = logging.Formatter('$asctime $name ${levelname}
$message',
...                        style='$')
>>> handler.setFormatter(df)
>>> logger.debug('This is a DEBUG message')
2010-10-28 15:13:06,924 foo.bar DEBUG This is a DEBUG message
>>> logger.critical('This is a CRITICAL message')
2010-10-28 15:13:11,494 foo.bar CRITICAL This is a CRITICAL
message
>>>
```

Note that the formatting of logging messages for final output to logs is completely independent of how an individual logging message is constructed. That can still use %-formatting, as shown here:

```
>>> logger.error('This is an%s %s %s', 'other,', 'ERROR,',
'message')
```

```
2010-10-28 15:19:29,833 foo.bar ERROR This is another, ERROR,
message
>>>
```

Logging calls (`logger.debug()`, `logger.info()` etc.) only take positional parameters for the actual logging message itself, with keyword parameters used only for determining options for how to handle the actual logging call (e.g. the `exc_info` keyword parameter to indicate that traceback information should be logged, or the `extra` keyword parameter to indicate additional contextual information to be added to the log). So you cannot directly make logging calls using **`str.format()`** or **`string.Template`** syntax, because internally the logging package uses %-formatting to merge the format string and the variable arguments. There would be no changing this while preserving backward compatibility, since all logging calls which are out there in existing code will be using %-format strings.

There is, however, a way that you can use {}- and $-formatting to construct your individual log messages. Recall that for a message you can use an arbitrary object as a message format string, and that the logging package will call `str()` on that object to get the actual format string. Consider the following two classes:

```
class BraceMessage:
    def __init__(self, fmt, /, *args, **kwargs):
        self.fmt = fmt
        self.args = args
        self.kwargs = kwargs

    def __str__(self):
        return self.fmt.format(*self.args, **self.kwargs)

class DollarMessage:
    def __init__(self, fmt, /, **kwargs):
        self.fmt = fmt
        self.kwargs = kwargs
    def __str__(self):
        from string import Template
        return Template(self.fmt).substitute(**self.kwargs)
```

Either of these can be used in place of a format string, to allow {}- or $-formatting to be used to build the actual “message” part which appears in the formatted log output in place of “%(message)s” or “{message}” or “$message”. It’s a little unwieldy to use the class names whenever you want to log something, but it’s quite palatable if you use an alias such as __ (double underscore — not to be confused with _, the single underscore used as a synonym/alias for `gettext.gettext()` or its brethren).

The above classes are not included in Python, though they’re easy enough to copy and paste into your own code. They can be used as follows (assuming that they’re declared in a module called `wherever`):

```
>>> from wherever import BraceMessage as __
>>> print(__('Message with {0} {name}', 2,
name='placeholders'))
Message with 2 placeholders
>>> class Point: pass
...
>>> p = Point()
>>> p.x = 0.5
>>> p.y = 0.5
>>> print(__('Message with coordinates: ({point.x:.2f},
{point.y:.2f})',
...          point=p))
Message with coordinates: (0.50, 0.50)
>>> from wherever import DollarMessage as __
>>> print(__('Message with $num $what', num=2,
what='placeholders'))
Message with 2 placeholders
>>>
```

While the above examples use `print()` to show how the formatting works, you would of course use `logger.debug()` or similar to actually log using this approach.

One thing to note is that you pay no significant performance penalty with this approach: the actual formatting happens not when you make the logging call, but when (and if) the logged message is actually about to be output to a log by a handler. So the only slightly unusual thing which might trip you up is that the parentheses go around the format string and the arguments, not just the format string. That’s because the __ notation is just syntax sugar for a constructor call to one of the *XXX*`Message` classes.

If you prefer, you can use a [`LoggerAdapter`] to achieve a similar effect to the above, as in the following example:

```
import logging

class Message:
    def __init__(self, fmt, args):
        self.fmt = fmt
        self.args = args

    def __str__(self):
        return self.fmt.format(*self.args)

class StyleAdapter(logging.LoggerAdapter):
    def log(self, level, msg, /, *args, stacklevel=1,
**kwargs):
        if self.isEnabledFor(level):
            msg, kwargs = self.process(msg, kwargs)
            self.logger.log(level, Message(msg, args),
**kwargs,
                            stacklevel=stacklevel+1)

logger = StyleAdapter(logging.getLogger(__name__))

def main():
    logger.debug('Hello, {}', 'world!')
```

```
if __name__ == '__main__':
    logging.basicConfig(level=logging.DEBUG)
    main()
```

The above script should log the message `Hello, world!` when run with Python 3.8 or later.

Customizing `LogRecord`

Every logging event is represented by a `LogRecord` instance. When an event is logged and not filtered out by a logger's level, a `LogRecord` is created, populated with information about the event and then passed to the handlers for that logger (and its ancestors, up to and including the logger where further propagation up the hierarchy is disabled). Before Python 3.2, there were only two places where this creation was done:

- `Logger.makeRecord()`, which is called in the normal process of logging an event. This invoked `LogRecord` directly to create an instance.
- `makeLogRecord()`, which is called with a dictionary containing attributes to be added to the LogRecord. This is typically invoked when a suitable dictionary has been received over the network (e.g. in pickle form via a `SocketHandler`, or in JSON form via an `HTTPHandler`).

This has usually meant that if you need to do anything special with a `LogRecord`, you've had to do one of the following.

- Create your own `Logger` subclass, which overrides `Logger.makeRecord()`, and set it using `setLoggerClass()` before any loggers that you care about are instantiated.

- Add a `Filter` to a logger or handler, which does the necessary special manipulation you need when its `filter()` method is called.

The first approach would be a little unwieldy in the scenario where (say) several different libraries wanted to do different things. Each would attempt to set its own `Logger` subclass, and the one which did this last would win.

The second approach works reasonably well for many cases, but does not allow you to e.g. use a specialized subclass of `LogRecord`. Library developers can set a suitable filter on their loggers, but they would have to remember to do this every time they introduced a new logger (which they would do simply by adding new packages or modules and doing

```
logger = logging.getLogger(__name__)
```

at module level). It’s probably one too many things to think about. Developers could also add the filter to a `NullHandler` attached to their top-level logger, but this would not be invoked if an application developer attached a handler to a lower-level library logger — so output from that handler would not reflect the intentions of the library developer.

In Python 3.2 and later, `LogRecord` creation is done through a factory, which you can specify. The factory is just a callable you can set with `setLogRecordFactory()`, and interrogate with `getLogRecordFactory()`. The factory is invoked with the same signature as the `LogRecord` constructor, as `LogRecord` is the default setting for the factory.

This approach allows a custom factory to control all aspects of LogRecord creation. For example, you could return a subclass, or just add some additional attributes to the record once created, using a pattern similar to this:

```
old_factory = logging.getLogRecordFactory()

def record_factory(*args, **kwargs):
    record = old_factory(*args, **kwargs)
    record.custom_attribute = 0xdecafbad
    return record

logging.setLogRecordFactory(record_factory)
```

This pattern allows different libraries to chain factories together, and as long as they don’t overwrite each other’s attributes or unintentionally overwrite the attributes provided as standard, there should be no surprises. However, it should be borne in mind that each link in the chain adds run-time overhead to all logging operations, and the technique should only be used when the use of a `Filter` does not provide the desired result.

Subclassing QueueHandler and QueueListener- a ZeroMQ example

Subclass `QueueHandler`

You can use a `QueueHandler` subclass to send messages to other kinds of queues, for example a ZeroMQ ‘publish’ socket. In the example below,the socket is created separately and passed to the handler (as its ‘queue’):

```
import zmq   # using pyzmq, the Python binding for ZeroMQ
import json  # for serializing records portably

ctx = zmq.Context()
sock = zmq.Socket(ctx, zmq.PUB)  # or zmq.PUSH, or other
suitable value
sock.bind('tcp://*:5556')        # or wherever

class ZeroMQSocketHandler(QueueHandler):
    def enqueue(self, record):
        self.queue.send_json(record.__dict__)


handler = ZeroMQSocketHandler(sock)
```

Of course there are other ways of organizing this, for example passing in the data needed by the handler to create the socket:

```
class ZeroMQSocketHandler(QueueHandler):
    def __init__(self, uri, socktype=zmq.PUB, ctx=None):
        self.ctx = ctx or zmq.Context()
        socket = zmq.Socket(self.ctx, socktype)
        socket.bind(uri)
        super().__init__(socket)

    def enqueue(self, record):
        self.queue.send_json(record.__dict__)

    def close(self):
        self.queue.close()
```

Subclass `QueueListener`

You can also subclass `QueueListener` to get messages from other kinds of queues, for example a ZeroMQ ‘subscribe’ socket. Here’s an example:

```
class ZeroMQSocketListener(QueueListener):
    def __init__(self, uri, /, *handlers, **kwargs):
        self.ctx = kwargs.get('ctx') or zmq.Context()
        socket = zmq.Socket(self.ctx, zmq.SUB)
        socket.setsockopt_string(zmq.SUBSCRIBE, '')  #
subscribe to everything
        socket.connect(uri)
        super().__init__(socket, *handlers, **kwargs)

    def dequeue(self):
        msg = self.queue.recv_json()
        return logging.makeLogRecord(msg)
```

Subclassing QueueHandler and QueueListener- a `pynng` example

In a similar way to the above section, we can implement a listener and handler using pynng [https://pypi.org/project/pynng/], which is a Python binding to NNG [https://nng.nanomsg.org/], billed as a spiritual successor to ZeroMQ. The following snippets illustrate – you can test them in an environment which has `pynng` installed. Just for variety, we present the listener first.

Subclass `QueueListener`

```
# listener.py
import json
import logging
import logging.handlers

import pynng

DEFAULT_ADDR = "tcp://localhost:13232"

interrupted = False

class NNGSocketListener(logging.handlers.QueueListener):

    def __init__(self, uri, /, *handlers, **kwargs):
        # Have a timeout for interruptability, and open a
        # subscriber socket
        socket = pynng.Sub0(listen=uri, recv_timeout=500)
        # The b'' subscription matches all topics
        topics = kwargs.pop('topics', None) or b''
        socket.subscribe(topics)
        # We treat the socket as a queue
        super().__init__(socket, *handlers, **kwargs)

    def dequeue(self, block):
        data = None
        # Keep looping while not interrupted and no data
received over the
```

```
        # socket
        while not interrupted:
            try:
                data = self.queue.recv(block=block)
                break
            except pynng.Timeout:
                pass
            except pynng.Closed:  # sometimes happens when you
hit Ctrl-C
                break
        if data is None:
            return None
        # Get the logging event sent from a publisher
        event = json.loads(data.decode('utf-8'))
        return logging.makeLogRecord(event)

    def enqueue_sentinel(self):
        # Not used in this implementation, as the socket isn't
really a
        # queue
        pass

logging.getLogger('pynng').propagate = False
listener = NNGSocketListener(DEFAULT_ADDR,
logging.StreamHandler(), topics=b'')
listener.start()
print('Press Ctrl-C to stop.')
try:
    while True:
        pass
except KeyboardInterrupt:
    interrupted = True
finally:
    listener.stop()
```

Subclass `QueueHandler`

```
# sender.py
import json
import logging
import logging.handlers
import time
import random

import pynng

DEFAULT_ADDR = "tcp://localhost:13232"

class NNGSocketHandler(logging.handlers.QueueHandler):

    def __init__(self, uri):
        socket = pynng.Pub0(dial=uri, send_timeout=500)
        super().__init__(socket)

    def enqueue(self, record):
        # Send the record as UTF-8 encoded JSON
        d = dict(record.__dict__)
        data = json.dumps(d)
        self.queue.send(data.encode('utf-8'))

    def close(self):
        self.queue.close()

logging.getLogger('pynng').propagate = False
handler = NNGSocketHandler(DEFAULT_ADDR)
# Make sure the process ID is in the output
logging.basicConfig(level=logging.DEBUG,
                    handlers=[logging.StreamHandler(),
handler],
                    format='%(levelname)-8s %(name)10s %
(process)6s %(message)s')
levels = (logging.DEBUG, logging.INFO, logging.WARNING,
logging.ERROR,
          logging.CRITICAL)
logger_names = ('myapp', 'myapp.lib1', 'myapp.lib2')
msgno = 1
while True:
    # Just randomly select some loggers and levels and log away
    level = random.choice(levels)
    logger = logging.getLogger(random.choice(logger_names))
    logger.log(level, 'Message no. %5d' % msgno)
    msgno += 1
    delay = random.random() * 2 + 0.5
    time.sleep(delay)
```

You can run the above two snippets in separate command shells. If we run the listener in one shell and run the sender in two separate shells, we should see something like the following. In the first sender shell:

```
$ python sender.py
DEBUG         myapp    613 Message no.     1
WARNING  myapp.lib2    613 Message no.     2
CRITICAL myapp.lib2    613 Message no.     3
WARNING  myapp.lib2    613 Message no.     4
CRITICAL myapp.lib1    613 Message no.     5
DEBUG         myapp    613 Message no.     6
CRITICAL myapp.lib1    613 Message no.     7
INFO     myapp.lib1    613 Message no.     8
(and so on)
```

In the second sender shell:

```
$ python sender.py
INFO     myapp.lib2    657 Message no.     1
CRITICAL myapp.lib2    657 Message no.     2
CRITICAL      myapp    657 Message no.     3
CRITICAL myapp.lib1    657 Message no.     4
INFO     myapp.lib1    657 Message no.     5
WARNING  myapp.lib2    657 Message no.     6
CRITICAL      myapp    657 Message no.     7
DEBUG    myapp.lib1    657 Message no.     8
(and so on)
```

In the listener shell:

```
$ python listener.py
Press Ctrl-C to stop.
DEBUG         myapp    613 Message no.     1
WARNING  myapp.lib2    613 Message no.     2
INFO     myapp.lib2    657 Message no.     1
CRITICAL myapp.lib2    613 Message no.     3
CRITICAL myapp.lib2    657 Message no.     2
CRITICAL      myapp    657 Message no.     3
WARNING  myapp.lib2    613 Message no.     4
CRITICAL myapp.lib1    613 Message no.     5
CRITICAL myapp.lib1    657 Message no.     4
```

```
INFO     myapp.lib1    657 Message no.     5
DEBUG         myapp    613 Message no.     6
WARNING  myapp.lib2    657 Message no.     6
CRITICAL      myapp    657 Message no.     7
CRITICAL myapp.lib1    613 Message no.     7
INFO     myapp.lib1    613 Message no.     8
DEBUG    myapp.lib1    657 Message no.     8
(and so on)
```

As you can see, the logging from the two sender processes is interleaved in the listener's output.

An example dictionary-based configuration

Below is an example of a logging configuration dictionary - it's taken from the documentation on the Django project [https://docs.djangoproject.com/en/stable/topics/logging/#configuring-logging]. This dictionary is passed to **`dictConfig()`** to put the configuration into effect:

```
LOGGING = {
    'version': 1,
    'disable_existing_loggers': False,
    'formatters': {
        'verbose': {
            'format': '{levelname} {asctime} {module}
{process:d} {thread:d} {message}',
            'style': '{',
        },
        'simple': {
            'format': '{levelname} {message}',
            'style': '{',
        },
    },
    'filters': {
        'special': {
            '()': 'project.logging.SpecialFilter',
            'foo': 'bar',
        },
    },
    'handlers': {
        'console': {
            'level': 'INFO',
            'class': 'logging.StreamHandler',
            'formatter': 'simple',
        },
        'mail_admins': {
            'level': 'ERROR',
            'class': 'django.utils.log.AdminEmailHandler',
            'filters': ['special']
        }
    },
    'loggers': {
        'django': {
            'handlers': ['console'],
            'propagate': True,
        },
        'django.request': {
            'handlers': ['mail_admins'],
            'level': 'ERROR',
            'propagate': False,
        },
        'myproject.custom': {
            'handlers': ['console', 'mail_admins'],
            'level': 'INFO',
            'filters': ['special']
        }
    }
}
```

For more information about this configuration, you can see the relevant section [https://docs.djangoproject.com/en/stable/topics/logging/#configuring-logging] of the Django documentation.

Using a rotator and namer to customize log rotation processing

An example of how you can define a namer and rotator is given in the following runnable script, which shows gzip compression of the log file:

```
import gzip
import logging
import logging.handlers
import os
import shutil

def namer(name):
    return name + ".gz"

def rotator(source, dest):
    with open(source, 'rb') as f_in:
        with gzip.open(dest, 'wb') as f_out:
            shutil.copyfileobj(f_in, f_out)
    os.remove(source)


rh = logging.handlers.RotatingFileHandler('rotated.log',
maxBytes=128, backupCount=5)
rh.rotator = rotator
rh.namer = namer

root = logging.getLogger()
root.setLevel(logging.INFO)
root.addHandler(rh)
f = logging.Formatter('%(asctime)s %(message)s')
rh.setFormatter(f)
for i in range(1000):
    root.info(f'Message no. {i + 1}')
```

After running this, you will see six new files, five of which are compressed:

```
$ ls rotated.log*
rotated.log       rotated.log.2.gz  rotated.log.4.gz
rotated.log.1.gz  rotated.log.3.gz  rotated.log.5.gz
$ zcat rotated.log.1.gz
2023-01-20 02:28:17,767 Message no. 996
```

```
2023-01-20 02:28:17,767 Message no. 997
2023-01-20 02:28:17,767 Message no. 998
```

A more elaborate multiprocessing example

The following working example shows how logging can be used with multiprocessing using configuration files. The configurations are fairly simple, but serve to illustrate how more complex ones could be implemented in a real multiprocessing scenario.

In the example, the main process spawns a listener process and some worker processes. Each of the main process, the listener and the workers have three separate configurations (the workers all share the same configuration). We can see logging in the main process, how the workers log to a QueueHandler and how the listener implements a QueueListener and a more complex logging configuration, and arranges to dispatch events received via the queue to the handlers specified in the configuration. Note that these configurations are purely illustrative, but you should be able to adapt this example to your own scenario.

Here’s the script - the docstrings and the comments hopefully explain how it works:

```
import logging
import logging.config
import logging.handlers
from multiprocessing import Process, Queue, Event,
current_process
import os
import random
import time

class MyHandler:
    """
    A simple handler for logging events. It runs in the
listener process and
    dispatches events to loggers based on the name in the
received record,
    which then get dispatched, by the logging system, to the
handlers
    configured for those loggers.
    """

    def handle(self, record):
        if record.name == "root":
            logger = logging.getLogger()
        else:
            logger = logging.getLogger(record.name)

        if logger.isEnabledFor(record.levelno):
            # The process name is transformed just to show that
it's the listener
            # doing the logging to files and console
            record.processName = '%s (for %s)' %
(current_process().name, record.processName)
            logger.handle(record)

def listener_process(q, stop_event, config):
    """
    This could be done in the main process, but is just done in
a separate
    process for illustrative purposes.

    This initialises logging according to the specified
configuration,
    starts the listener and waits for the main process to
signal completion
    via the event. The listener is then stopped, and the
process exits.
    """
    logging.config.dictConfig(config)
    listener = logging.handlers.QueueListener(q, MyHandler())
    listener.start()
    if os.name == 'posix':
        # On POSIX, the setup logger will have been configured
in the
```

```
        # parent process, but should have been disabled
following the
        # dictConfig call.
        # On Windows, since fork isn't used, the setup logger
won't
        # exist in the child, so it would be created and the
message
        # would appear - hence the "if posix" clause.
        logger = logging.getLogger('setup')
        logger.critical('Should not appear, because of disabled
logger ...')
    stop_event.wait()
    listener.stop()

def worker_process(config):
    """
    A number of these are spawned for the purpose of
illustration. In
    practice, they could be a heterogeneous bunch of processes
rather than
    ones which are identical to each other.

    This initialises logging according to the specified
configuration,
    and logs a hundred messages with random levels to randomly
selected
    loggers.

    A small sleep is added to allow other processes a chance to
run. This
    is not strictly needed, but it mixes the output from the
different
    processes a bit more than if it's left out.
    """
    logging.config.dictConfig(config)
    levels = [logging.DEBUG, logging.INFO, logging.WARNING,
logging.ERROR,
              logging.CRITICAL]
    loggers = ['foo', 'foo.bar', 'foo.bar.baz',
               'spam', 'spam.ham', 'spam.ham.eggs']
    if os.name == 'posix':
        # On POSIX, the setup logger will have been configured
in the
```

```
        # parent process, but should have been disabled
following the
        # dictConfig call.
        # On Windows, since fork isn't used, the setup logger
won't
        # exist in the child, so it would be created and the
message
        # would appear - hence the "if posix" clause.
        logger = logging.getLogger('setup')
        logger.critical('Should not appear, because of disabled
logger ...')
    for i in range(100):
        lvl = random.choice(levels)
        logger = logging.getLogger(random.choice(loggers))
        logger.log(lvl, 'Message no. %d', i)
        time.sleep(0.01)

def main():
    q = Queue()
    # The main process gets a simple configuration which prints
to the console.
    config_initial = {
        'version': 1,
        'handlers': {
            'console': {
                'class': 'logging.StreamHandler',
                'level': 'INFO'
            }
        },
        'root': {
            'handlers': ['console'],
            'level': 'DEBUG'
        }
    }
    # The worker process configuration is just a QueueHandler
attached to the
    # root logger, which allows all messages to be sent to the
queue.
    # We disable existing loggers to disable the "setup" logger
used in the
    # parent process. This is needed on POSIX because the
logger will
    # be there in the child following a fork().
    config_worker = {
        'version': 1,
        'disable_existing_loggers': True,
        'handlers': {
            'queue': {
                'class': 'logging.handlers.QueueHandler',
                'queue': q
            }
        },
        'root': {
            'handlers': ['queue'],
            'level': 'DEBUG'
        }
    }
    # The listener process configuration shows that the full
flexibility of
    # logging configuration is available to dispatch events to
handlers however
    # you want.
    # We disable existing loggers to disable the "setup" logger
used in the
    # parent process. This is needed on POSIX because the
logger will
    # be there in the child following a fork().
    config_listener = {
        'version': 1,
        'disable_existing_loggers': True,
        'formatters': {
            'detailed': {
                'class': 'logging.Formatter',
                'format': '%(asctime)s %(name)-15s %
(levelname)-8s %(processName)-10s %(message)s'
            },
            'simple': {
                'class': 'logging.Formatter',
                'format': '%(name)-15s %(levelname)-8s %
(processName)-10s %(message)s'
            }
        },
        'handlers': {
            'console': {
                'class': 'logging.StreamHandler',
                'formatter': 'simple',
                'level': 'INFO'
            },
            'file': {
                'class': 'logging.FileHandler',
                'filename': 'mplog.log',
                'mode': 'w',
                'formatter': 'detailed'
            },
            'foofile': {
                'class': 'logging.FileHandler',
                'filename': 'mplog-foo.log',
                'mode': 'w',
                'formatter': 'detailed'
            },
            'errors': {
                'class': 'logging.FileHandler',
                'filename': 'mplog-errors.log',
                'mode': 'w',
                'formatter': 'detailed',
                'level': 'ERROR'
            }
        },
        'loggers': {
            'foo': {
                'handlers': ['foofile']
            }
        },
        'root': {
            'handlers': ['console', 'file', 'errors'],
            'level': 'DEBUG'
        }
    }
    # Log some initial events, just to show that logging in the
parent works
    # normally.
    logging.config.dictConfig(config_initial)
    logger = logging.getLogger('setup')
    logger.info('About to create workers ...')
    workers = []
    for i in range(5):
        wp = Process(target=worker_process, name='worker %d' %
(i + 1),
                     args=(config_worker,))
        workers.append(wp)
        wp.start()
        logger.info('Started worker: %s', wp.name)
    logger.info('About to create listener ...')
    stop_event = Event()
    lp = Process(target=listener_process, name='listener',
                 args=(q, stop_event, config_listener))
    lp.start()
    logger.info('Started listener')
    # We now hang around for the workers to finish their work.
    for wp in workers:
        wp.join()
    # Workers all done, listening can now stop.
    # Logging in the parent still works normally.
    logger.info('Telling listener to stop ...')
    stop_event.set()
    lp.join()
    logger.info('All done.')

if __name__ == '__main__':
    main()
```

Inserting a BOM into messages sent to a SysLogHandler

RFC 5424 [https://datatracker.ietf.org/doc/html/rfc5424.html] requires that a Unicode message be sent to a syslog daemon as a set of bytes which have the following structure: an optional pure-ASCII component, followed by a UTF-8 Byte Order Mark (BOM), followed by Unicode encoded using UTF-8. (See the **relevant section of the specification** [https://datatracker.ietf.org/doc/html/rfc5424.html#section-6].)

In Python 3.1, code was added to `SysLogHandler` to insert a BOM into the message, but unfortunately, it was implemented incorrectly, with the BOM appearing at the beginning of the message and hence not allowing any pure-ASCII component to appear before it.

As this behaviour is broken, the incorrect BOM insertion code is being removed from Python 3.2.4 and later. However, it is not being replaced, and if you want to produce **RFC 5424** [https://datatracker.ietf.org/doc/html/rfc5424.html]-compliant messages which include a BOM, an optional pure-ASCII sequence before it and arbitrary Unicode after it, encoded using UTF-8, then you need to do the following:

1. Attach a **`Formatter`** instance to your **`SysLogHandler`** instance, with a format string such as:

   ```
   'ASCII section\ufeffUnicode section'
   ```

   The Unicode code point U+FEFF, when encoded using UTF-8, will be encoded as a UTF-8 BOM – the byte-string `b'\xef\xbb\xbf'`.

2. Replace the ASCII section with whatever placeholders you like, but make sure that the data that appears in there after substitution is always ASCII (that way, it will remain unchanged after UTF-8 encoding).

3. Replace the Unicode section with whatever placeholders you like; if the data which appears there after substitution contains characters outside the ASCII range, that’s fine – it will be encoded using UTF-8.

The formatted message *will* be encoded using UTF-8 encoding by `SysLogHandler`. If you follow the above rules, you should be able to produce **RFC 5424** [https://datatracker.ietf.org/doc/html/rfc5424.html]-compliant messages. If you don’t, logging may not complain, but your messages will not be RFC 5424-compliant, and your syslog daemon may complain.

Implementing structured logging

Although most logging messages are intended for reading by humans, and thus not readily machine-parseable, there might be circumstances where you want to output messages in a structured format which *is* capable of being parsed by a program (without needing complex regular expressions to parse the log message). This is straightforward to achieve using the logging package. There are a number of ways in which this could be achieved, but the following is a simple approach which uses JSON to serialise the event in a machine-parseable manner:

```
import json
import logging

class StructuredMessage:
    def __init__(self, message, /, **kwargs):
        self.message = message
        self.kwargs = kwargs

    def __str__(self):
        return '%s >>> %s' % (self.message,
json.dumps(self.kwargs))

_ = StructuredMessage   # optional, to improve readability

logging.basicConfig(level=logging.INFO, format='%(message)s')
logging.info(_('message 1', foo='bar', bar='baz', num=123,
fnum=123.456))
```

If the above script is run, it prints:

```
message 1 >>> {"fnum": 123.456, "num": 123, "bar": "baz",
"foo": "bar"}
```

Note that the order of items might be different according to the version of Python used.

If you need more specialised processing, you can use a custom JSON encoder, as in the following complete example:

```
import json
import logging


class Encoder(json.JSONEncoder):
    def default(self, o):
        if isinstance(o, set):
            return tuple(o)
        elif isinstance(o, str):
            return o.encode('unicode_escape').decode('ascii')
        return super().default(o)

class StructuredMessage:
    def __init__(self, message, /, **kwargs):
        self.message = message
        self.kwargs = kwargs

    def __str__(self):
        s = Encoder().encode(self.kwargs)
        return '%s >>> %s' % (self.message, s)

_ = StructuredMessage   # optional, to improve readability

def main():
    logging.basicConfig(level=logging.INFO, format='%(message)s')
    logging.info(_('message 1', set_value={1, 2, 3}, snowman='\u2603'))

if __name__ == '__main__':
    main()
```

When the above script is run, it prints:

```
message 1 >>> {"snowman": "\u2603", "set_value": [1, 2, 3]}
```

Note that the order of items might be different according to the version of Python used.

Customizing handlers with `dictConfig()`

There are times when you want to customize logging handlers in particular ways, and if you use `dictConfig()` you may be able to do this without subclassing. As an example, consider that you may want to set the ownership of a log file. On POSIX, this is easily done using `shutil.chown()`, but the file handlers in the stdlib don't offer built-in support. You can customize handler creation using a plain function such as:

```
def owned_file_handler(filename, mode='a', encoding=None,
owner=None):
    if owner:
        if not os.path.exists(filename):
            open(filename, 'a').close()
        shutil.chown(filename, *owner)
    return logging.FileHandler(filename, mode, encoding)
```

You can then specify, in a logging configuration passed to `dictConfig()`, that a logging handler be created by calling this function:

```
LOGGING = {
    'version': 1,
    'disable_existing_loggers': False,
    'formatters': {
        'default': {
            'format': '%(asctime)s %(levelname)s %(name)s %
(message)s'
        },
    },
    'handlers': {
        'file':{
            # The values below are popped from this dictionary
and
            # used to create the handler, set the handler's
level and
            # its formatter.
            '()': owned_file_handler,
            'level':'DEBUG',
            'formatter': 'default',
            # The values below are passed to the handler
creator callable
            # as keyword arguments.
            'owner': ['pulse', 'pulse'],
            'filename': 'chowntest.log',
            'mode': 'w',
            'encoding': 'utf-8',
        },
    },
    'root': {
        'handlers': ['file'],
        'level': 'DEBUG',
    },
}
```

In this example I am setting the ownership using the `pulse` user and group, just for the purposes of illustration. Putting it together into a working script, `chowntest.py`:

```
import logging, logging.config, os, shutil

def owned_file_handler(filename, mode='a', encoding=None,
owner=None):
    if owner:
        if not os.path.exists(filename):
            open(filename, 'a').close()
        shutil.chown(filename, *owner)
    return logging.FileHandler(filename, mode, encoding)

LOGGING = {
    'version': 1,
    'disable_existing_loggers': False,
    'formatters': {
        'default': {
            'format': '%(asctime)s %(levelname)s %(name)s %
(message)s'
        },
    },
    'handlers': {
        'file':{
            # The values below are popped from this dictionary
and
            # used to create the handler, set the handler's
level and
            # its formatter.
            '()': owned_file_handler,
            'level':'DEBUG',
            'formatter': 'default',
            # The values below are passed to the handler
creator callable
            # as keyword arguments.
            'owner': ['pulse', 'pulse'],
            'filename': 'chowntest.log',
            'mode': 'w',
            'encoding': 'utf-8',
        },
    },
    'root': {
        'handlers': ['file'],
        'level': 'DEBUG',
    },
}

logging.config.dictConfig(LOGGING)
logger = logging.getLogger('mylogger')
logger.debug('A debug message')
```

To run this, you will probably need to run as `root`:

```
$ sudo python3.3 chowntest.py
$ cat chowntest.log
2013-11-05 09:34:51,128 DEBUG mylogger A debug message
$ ls -l chowntest.log
-rw-r--r-- 1 pulse pulse 55 2013-11-05 09:34 chowntest.log
```

Note that this example uses Python 3.3 because that’s where **`shutil.chown()`** makes an appearance. This approach should work with any Python version that supports **`dictConfig()`** - namely, Python 2.7, 3.2 or later. With pre-3.3 versions, you would need to implement the actual ownership change using e.g. **`os.chown()`**.

In practice, the handler-creating function may be in a utility module somewhere in your project. Instead of the line in the configuration:

```
'()': owned_file_handler,
```

you could use e.g.:

```
'()': 'ext://project.util.owned_file_handler',
```

where `project.util` can be replaced with the actual name of the package where the function resides. In the above working script, using `'ext://__main__.owned_file_handler'` should work. Here, the actual callable is resolved by **`dictConfig()`** from the `ext://` specification.

This example hopefully also points the way to how you could implement other types of file change - e.g. setting specific POSIX permission bits - in the same way, using **`os.chmod()`**.

Of course, the approach could also be extended to types of handler other than a **`FileHandler`** - for example, one of the rotating file handlers, or a different type of handler altogether.

Using particular formatting styles throughout your application

In Python 3.2, the **`Formatter`** gained a `style` keyword parameter which, while defaulting to `%` for backward compatibility, allowed the specification of `{` or `$` to support the formatting approaches supported by **`str.format()`** and **`string.Template`**. Note that this governs the formatting of logging messages for final output to logs, and is completely orthogonal to how an individual logging message is constructed.

Logging calls (`debug()`, `info()` etc.) only take positional parameters for the actual logging message itself, with keyword parameters used only for determining options for how to handle the logging call (e.g. the `exc_info` keyword parameter to indicate that traceback information should be logged, or the `extra` keyword parameter to indicate additional contextual information to be added to the log). So you cannot directly make logging calls using `str.format()` or `string.Template` syntax, because internally the logging package uses %-formatting to merge the format string and the variable arguments. There would be no changing this while preserving backward compatibility, since all logging calls which are out there in existing code will be using %-format strings.

There have been suggestions to associate format styles with specific loggers, but that approach also runs into backward compatibility problems because any existing code could be using a given logger name and using %-formatting.

For logging to work interoperably between any third-party libraries and your code, decisions about formatting need to be made at the level of the individual logging call. This opens up a couple of ways in which alternative formatting styles can be accommodated.

Using LogRecord factories

In Python 3.2, along with the `Formatter` changes mentioned above, the logging package gained the ability to allow users to set their own `LogRecord` subclasses, using the `setLogRecordFactory()` function. You can use this to set your own subclass of `LogRecord`, which does the Right Thing by overriding the `getMessage()` method. The base class implementation of this method is where the `msg % args` formatting happens, and where you can substitute your alternate formatting; however, you should be careful to support all formatting styles and allow %-formatting as the default, to ensure interoperability with other code. Care should also be taken to call `str(self.msg)`, just as the base implementation does.

Refer to the reference documentation on **`setLogRecordFactory()`** and **`LogRecord`** for more information.

Using custom message objects

There is another, perhaps simpler way that you can use {}- and $- formatting to construct your individual log messages. You may recall (from Using arbitrary objects as messages) that when logging you can use an arbitrary object as a message format string, and that the logging package will call **`str()`** on that object to get the actual format string. Consider the following two classes:

```
class BraceMessage:
    def __init__(self, fmt, /, *args, **kwargs):
        self.fmt = fmt
        self.args = args
        self.kwargs = kwargs

    def __str__(self):
        return self.fmt.format(*self.args, **self.kwargs)

class DollarMessage:
    def __init__(self, fmt, /, **kwargs):
        self.fmt = fmt
        self.kwargs = kwargs

    def __str__(self):
        from string import Template
        return Template(self.fmt).substitute(**self.kwargs)
```

Either of these can be used in place of a format string, to allow {}- or $-formatting to be used to build the actual “message” part which appears in the formatted log output in place of “%(message)s” or “{message}” or “$message”. If you find it a little unwieldy to use the class names whenever you want to log something, you can make it more palatable if you use an alias such as `M` or `_` for the message (or perhaps `__`, if you are using `_` for localization).

Examples of this approach are given below. Firstly, formatting with [`str.format()`](#):

```
>>> __ = BraceMessage
>>> print(__('Message with {0} {1}', 2, 'placeholders'))
Message with 2 placeholders
>>> class Point: pass
...
>>> p = Point()
>>> p.x = 0.5
>>> p.y = 0.5
>>> print(__('Message with coordinates: ({point.x:.2f},
{point.y:.2f})', point=p))
Message with coordinates: (0.50, 0.50)
```

Secondly, formatting with [`string.Template`](#):

```
>>> __ = DollarMessage
>>> print(__('Message with $num $what', num=2,
what='placeholders'))
Message with 2 placeholders
>>>
```

One thing to note is that you pay no significant performance penalty with this approach: the actual formatting happens not when you make the logging call, but when (and if) the logged message is actually about to be output to a log by a handler. So the only slightly unusual thing which might trip you up is that the parentheses go around the format string and the arguments, not just the format string. That’s because the `__` notation is just syntax sugar for a constructor call to one of the *XXX*`Message` classes shown above.

Configuring filters with `dictConfig()`

You *can* configure filters using `dictConfig()`, though it might not be obvious at first glance how to do it (hence this recipe). Since `Filter` is the only filter class included in the standard library, and it is unlikely to cater to many requirements (it’s only there as a base class), you will typically need to define your own `Filter` subclass with an overridden `filter()` method. To do this, specify the `()` key in the configuration dictionary for the filter, specifying a callable which will be used to create the filter (a class is the most obvious, but you can provide any callable which returns a `Filter` instance). Here is a complete example:

```
import logging
import logging.config
import sys

class MyFilter(logging.Filter):
    def __init__(self, param=None):
        self.param = param

    def filter(self, record):
        if self.param is None:
            allow = True
        else:
            allow = self.param not in record.msg
        if allow:
            record.msg = 'changed: ' + record.msg
        return allow

LOGGING = {
    'version': 1,
    'filters': {
        'myfilter': {
            '()': MyFilter,
            'param': 'noshow',
        }
    },
    'handlers': {
        'console': {
            'class': 'logging.StreamHandler',
            'filters': ['myfilter']
        }
    },
    'root': {
        'level': 'DEBUG',
        'handlers': ['console']
    },
}

if __name__ == '__main__':
    logging.config.dictConfig(LOGGING)
    logging.debug('hello')
    logging.debug('hello - noshow')
```

This example shows how you can pass configuration data to the callable which constructs the instance, in the form of keyword parameters. When run, the above script will print:

```
changed: hello
```

which shows that the filter is working as configured.

A couple of extra points to note:

- If you can't refer to the callable directly in the configuration (e.g. if it lives in a different module, and you can't import it directly where the configuration dictionary is), you can use the form `ext://...` as described in Access to external objects. For example, you could have used the text `'ext://__main__.MyFilter'` instead of `MyFilter` in the above example.
- As well as for filters, this technique can also be used to configure custom handlers and formatters. See User-defined objects for more information on how logging

supports using user-defined objects in its configuration, and see the other cookbook recipe Customizing handlers with dictConfig() above.

Customized exception formatting

There might be times when you want to do customized exception formatting - for argument’s sake, let’s say you want exactly one line per logged event, even when exception information is present. You can do this with a custom formatter class, as shown in the following example:

```
import logging

class OneLineExceptionFormatter(logging.Formatter):
    def formatException(self, exc_info):
        """
        Format an exception so that it prints on a single line.
        """
        result = super().formatException(exc_info)
        return repr(result)  # or format into one line however
you want to

    def format(self, record):
        s = super().format(record)
        if record.exc_text:
            s = s.replace('\n', '') + '|'
        return s

def configure_logging():
    fh = logging.FileHandler('output.txt', 'w')
    f = OneLineExceptionFormatter('%(asctime)s|%(levelname)s|%
(message)s|',
                                  '%d/%m/%Y %H:%M:%S')
    fh.setFormatter(f)
    root = logging.getLogger()
    root.setLevel(logging.DEBUG)
    root.addHandler(fh)

def main():
    configure_logging()
    logging.info('Sample message')
    try:
        x = 1 / 0
    except ZeroDivisionError as e:
        logging.exception('ZeroDivisionError: %s', e)

if __name__ == '__main__':
    main()
```

When run, this produces a file with exactly two lines:

```
28/01/2015 07:21:23|INFO|Sample message|
28/01/2015 07:21:23|ERROR|ZeroDivisionError: division by
zero|'Traceback (most recent call last):\n  File "logtest7.py",
line 30, in main\n    x = 1 / 0\nZeroDivisionError: division by
zero'|
```

While the above treatment is simplistic, it points the way to how exception information can be formatted to your liking. The **`traceback`** module may be helpful for more specialized needs.

Speaking logging messages

There might be situations when it is desirable to have logging messages rendered in an audible rather than a visible format. This is easy to do if you have text-to-speech (TTS) functionality available in your system, even if it doesn’t have a Python binding. Most TTS systems have a command line program you can run, and this can be invoked from a handler using **`subprocess`**. It’s assumed here that TTS command line programs won’t expect to interact with users or take a long time to complete, and that the frequency of logged messages will be not so high as to swamp the user with messages, and that it’s acceptable to have the messages spoken one at a time rather than concurrently, The example implementation below waits for one message to be spoken before the next is processed, and this might cause other handlers to be kept waiting. Here is a short example showing the approach, which assumes that the `espeak` TTS package is available:

```
import logging
import subprocess
import sys

class TTSHandler(logging.Handler):
    def emit(self, record):
        msg = self.format(record)
        # Speak slowly in a female English voice
        cmd = ['espeak', '-s150', '-ven+f3', msg]
        p = subprocess.Popen(cmd, stdout=subprocess.PIPE,
                             stderr=subprocess.STDOUT)
        # wait for the program to finish
        p.communicate()

def configure_logging():
    h = TTSHandler()
    root = logging.getLogger()
    root.addHandler(h)
    # the default formatter just returns the message
    root.setLevel(logging.DEBUG)

def main():
    logging.info('Hello')
    logging.debug('Goodbye')

if __name__ == '__main__':
    configure_logging()
    sys.exit(main())
```

When run, this script should say “Hello” and then “Goodbye” in a female voice.

The above approach can, of course, be adapted to other TTS systems and even other systems altogether which can process messages via external programs run from a command line.

Buffering logging messages and outputting them conditionally

There might be situations where you want to log messages in a temporary area and only output them if a certain condition occurs. For example, you may want to start logging debug events in a function, and if the function completes without errors, you don’t want to clutter the log with the collected debug information, but if there is an error, you want all the debug information to be output as well as the error.

Here is an example which shows how you could do this using a decorator for your functions where you want logging to behave this way. It makes use of the **`logging.handlers.MemoryHandler`**, which allows buffering of logged events until some condition occurs, at which point the buffered events are `flushed` - passed to another handler (the `target` handler) for processing. By default, the `MemoryHandler` flushed when its buffer gets filled up or an event whose level is greater than or equal to a specified threshold is seen. You can use this recipe with a more specialised subclass of `MemoryHandler` if you want custom flushing behavior.

The example script has a simple function, `foo`, which just cycles through all the logging levels, writing to `sys.stderr` to say what level it’s about to log at, and then actually logging a message at that level. You can pass a parameter to `foo` which, if true, will log at ERROR and CRITICAL levels - otherwise, it only logs at DEBUG, INFO and WARNING levels.

The script just arranges to decorate `foo` with a decorator which will do the conditional logging that’s required. The decorator takes a logger as a parameter and attaches a memory handler for the duration of the call to the decorated function. The decorator can be additionally parameterised using a target handler, a level at which flushing should occur, and a capacity for the buffer (number of records buffered). These default to a `StreamHandler` which writes to `sys.stderr`, `logging.ERROR` and `100` respectively.

Here's the script:

```
import logging
from logging.handlers import MemoryHandler
import sys

logger = logging.getLogger(__name__)
logger.addHandler(logging.NullHandler())

def log_if_errors(logger, target_handler=None,
flush_level=None, capacity=None):
    if target_handler is None:
        target_handler = logging.StreamHandler()
    if flush_level is None:
        flush_level = logging.ERROR
    if capacity is None:
        capacity = 100
    handler = MemoryHandler(capacity, flushLevel=flush_level,
target=target_handler)

    def decorator(fn):
        def wrapper(*args, **kwargs):
            logger.addHandler(handler)
            try:
                return fn(*args, **kwargs)
            except Exception:
                logger.exception('call failed')
                raise
            finally:
                super(MemoryHandler, handler).flush()
                logger.removeHandler(handler)
        return wrapper

    return decorator
```

```
def write_line(s):
    sys.stderr.write('%s\n' % s)

def foo(fail=False):
    write_line('about to log at DEBUG ...')
    logger.debug('Actually logged at DEBUG')
    write_line('about to log at INFO ...')
    logger.info('Actually logged at INFO')
    write_line('about to log at WARNING ...')
    logger.warning('Actually logged at WARNING')
    if fail:
        write_line('about to log at ERROR ...')
        logger.error('Actually logged at ERROR')
        write_line('about to log at CRITICAL ...')
        logger.critical('Actually logged at CRITICAL')
    return fail

decorated_foo = log_if_errors(logger)(foo)

if __name__ == '__main__':
    logger.setLevel(logging.DEBUG)
    write_line('Calling undecorated foo with False')
    assert not foo(False)
    write_line('Calling undecorated foo with True')
    assert foo(True)
    write_line('Calling decorated foo with False')
    assert not decorated_foo(False)
    write_line('Calling decorated foo with True')
    assert decorated_foo(True)
```

When this script is run, the following output should be observed:

```
Calling undecorated foo with False
about to log at DEBUG ...
about to log at INFO ...
about to log at WARNING ...
Calling undecorated foo with True
about to log at DEBUG ...
about to log at INFO ...
about to log at WARNING ...
about to log at ERROR ...
about to log at CRITICAL ...
Calling decorated foo with False
about to log at DEBUG ...
about to log at INFO ...
about to log at WARNING ...
Calling decorated foo with True
about to log at DEBUG ...
about to log at INFO ...
about to log at WARNING ...
about to log at ERROR ...
Actually logged at DEBUG
Actually logged at INFO
Actually logged at WARNING
Actually logged at ERROR
about to log at CRITICAL ...
Actually logged at CRITICAL
```

As you can see, actual logging output only occurs when an event is logged whose severity is ERROR or greater, but in that case, any previous events at lower severities are also logged.

You can of course use the conventional means of decoration:

```
@log_if_errors(logger)
def foo(fail=False):
    ...
```

Sending logging messages to email, with buffering

To illustrate how you can send log messages via email, so that a set number of messages are sent per email, you can subclass `BufferingHandler`. In the following example, which you can adapt to suit your specific needs, a simple test harness is provided which allows you to run the script with command line arguments specifying what you typically need to send things via SMTP. (Run the downloaded script with the `-h` argument to see the required and optional arguments.)

```
import logging
import logging.handlers
import smtplib

class BufferingSMTPHandler(logging.handlers.BufferingHandler):
    def __init__(self, mailhost, port, username, password,
fromaddr, toaddrs,
                 subject, capacity):
        logging.handlers.BufferingHandler.__init__(self,
capacity)
        self.mailhost = mailhost
        self.mailport = port
        self.username = username
        self.password = password
        self.fromaddr = fromaddr
        if isinstance(toaddrs, str):
            toaddrs = [toaddrs]
        self.toaddrs = toaddrs
        self.subject = subject
        self.setFormatter(logging.Formatter("%(asctime)s %
(levelname)-5s %(message)s"))

    def flush(self):
        if len(self.buffer) > 0:
            try:
                smtp = smtplib.SMTP(self.mailhost,
self.mailport)
                smtp.starttls()
                smtp.login(self.username, self.password)
                msg = "From: %s\r\nTo: %s\r\nSubject:
%s\r\n\r\n" % (self.fromaddr, ','.join(self.toaddrs),
self.subject)
                for record in self.buffer:
                    s = self.format(record)
                    msg = msg + s + "\r\n"
                smtp.sendmail(self.fromaddr, self.toaddrs, msg)
                smtp.quit()
            except Exception:
                if logging.raiseExceptions:
                    raise
            self.buffer = []

if __name__ == '__main__':
    import argparse

    ap = argparse.ArgumentParser()
    aa = ap.add_argument
    aa('host', metavar='HOST', help='SMTP server')
    aa('--port', '-p', type=int, default=587, help='SMTP port')
    aa('user', metavar='USER', help='SMTP username')
    aa('password', metavar='PASSWORD', help='SMTP password')
    aa('to', metavar='TO', help='Addressee for emails')
    aa('sender', metavar='SENDER', help='Sender email address')
    aa('--subject', '-s',
       default='Test Logging email from Python logging module
(buffering)',
       help='Subject of email')
    options = ap.parse_args()
    logger = logging.getLogger()
    logger.setLevel(logging.DEBUG)
    h = BufferingSMTPHandler(options.host, options.port,
options.user,
                             options.password, options.sender,
                             options.to, options.subject, 10)
    logger.addHandler(h)
    for i in range(102):
        logger.info("Info index = %d", i)
    h.flush()
    h.close()
```

If you run this script and your SMTP server is correctly set up, you should find that it sends eleven emails to the addressee you specify. The first ten emails will each have ten log messages, and the eleventh will have two messages. That makes up 102 messages as specified in the script.

Formatting times using UTC (GMT) via configuration

Sometimes you want to format times using UTC, which can be done using a class such as `UTCFormatter`, shown below:

```
import logging
import time

class UTCFormatter(logging.Formatter):
    converter = time.gmtime
```

and you can then use the `UTCFormatter` in your code instead of **`Formatter`**. If you want to do that via configuration, you can use the **`dictConfig()`** API with an approach illustrated by the following complete example:

```
import logging
import logging.config
import time

class UTCFormatter(logging.Formatter):
    converter = time.gmtime

LOGGING = {
    'version': 1,
    'disable_existing_loggers': False,
    'formatters': {
        'utc': {
            '()': UTCFormatter,
            'format': '%(asctime)s %(message)s',
        },
        'local': {
            'format': '%(asctime)s %(message)s',
        }
    },
    'handlers': {
        'console1': {
            'class': 'logging.StreamHandler',
            'formatter': 'utc',
        },
        'console2': {
            'class': 'logging.StreamHandler',
            'formatter': 'local',
        },
    },
    'root': {
        'handlers': ['console1', 'console2'],
   }
}

if __name__ == '__main__':
    logging.config.dictConfig(LOGGING)
    logging.warning('The local time is %s', time.asctime())
```

When this script is run, it should print something like:

```
2015-10-17 12:53:29,501 The local time is Sat Oct 17 13:53:29
2015
2015-10-17 13:53:29,501 The local time is Sat Oct 17 13:53:29
2015
```

showing how the time is formatted both as local time and UTC, one for each handler.

Using a context manager for selective logging

There are times when it would be useful to temporarily change the logging configuration and revert it back after doing something. For this, a context manager is the most obvious way of saving and restoring the logging context. Here is a simple example of such a context manager, which allows you to optionally change the logging level and add a logging handler purely in the scope of the context manager:

```
import logging
import sys

class LoggingContext:
    def __init__(self, logger, level=None, handler=None,
close=True):
        self.logger = logger
        self.level = level
        self.handler = handler
        self.close = close

    def __enter__(self):
        if self.level is not None:
            self.old_level = self.logger.level
            self.logger.setLevel(self.level)
        if self.handler:
            self.logger.addHandler(self.handler)

    def __exit__(self, et, ev, tb):
        if self.level is not None:
            self.logger.setLevel(self.old_level)
        if self.handler:
            self.logger.removeHandler(self.handler)
        if self.handler and self.close:
            self.handler.close()
        # implicit return of None => don't swallow exceptions
```

If you specify a level value, the logger’s level is set to that value in the scope of the with block covered by the context manager. If you specify a handler, it is added to the logger on entry to the block and removed on exit from the block. You can also ask the manager to close the handler for you on block exit - you could do this if you don’t need the handler any more.

To illustrate how it works, we can add the following block of code to the above:

```
if __name__ == '__main__':
    logger = logging.getLogger('foo')
    logger.addHandler(logging.StreamHandler())
    logger.setLevel(logging.INFO)
    logger.info('1. This should appear just once on stderr.')
    logger.debug('2. This should not appear.')
    with LoggingContext(logger, level=logging.DEBUG):
        logger.debug('3. This should appear once on stderr.')
    logger.debug('4. This should not appear.')
    h = logging.StreamHandler(sys.stdout)
    with LoggingContext(logger, level=logging.DEBUG, handler=h,
close=True):
        logger.debug('5. This should appear twice - once on
stderr and once on stdout.')
    logger.info('6. This should appear just once on stderr.')
    logger.debug('7. This should not appear.')
```

We initially set the logger’s level to `INFO`, so message #1 appears and message #2 doesn’t. We then change the level to `DEBUG` temporarily in the following `with` block, and so message #3 appears. After the block exits, the logger’s level is restored to `INFO` and so message #4 doesn’t appear. In the next `with` block, we set the level to `DEBUG` again but also add a handler writing to `sys.stdout`. Thus, message #5 appears twice on the console (once via `stderr` and once via `stdout`). After the `with` statement’s completion, the status is as it was before so message #6 appears (like message #1) whereas message #7 doesn’t (just like message #2).

If we run the resulting script, the result is as follows:

```
$ python logctx.py
1. This should appear just once on stderr.
3. This should appear once on stderr.
5. This should appear twice - once on stderr and once on
stdout.
5. This should appear twice - once on stderr and once on
stdout.
6. This should appear just once on stderr.
```

If we run it again, but pipe `stderr` to `/dev/null`, we see the following, which is the only message written to `stdout`:

```
$ python logctx.py 2>/dev/null
5. This should appear twice - once on stderr and once on
stdout.
```

Once again, but piping `stdout` to `/dev/null`, we get:

```
$ python logctx.py >/dev/null
1. This should appear just once on stderr.
```

```
3. This should appear once on stderr.
5. This should appear twice - once on stderr and once on
stdout.
6. This should appear just once on stderr.
```

In this case, the message #5 printed to `stdout` doesn't appear, as expected.

Of course, the approach described here can be generalised, for example to attach logging filters temporarily. Note that the above code works in Python 2 as well as Python 3.

A CLI application starter template

Here's an example which shows how you can:

- Use a logging level based on command-line arguments
- Dispatch to multiple subcommands in separate files, all logging at the same level in a consistent way
- Make use of simple, minimal configuration

Suppose we have a command-line application whose job is to stop, start or restart some services. This could be organised for the purposes of illustration as a file `app.py` that is the main script for the application, with individual commands implemented in `start.py`, `stop.py` and `restart.py`. Suppose further that we want to control the verbosity of the application via a command-line argument, defaulting to `logging.INFO`. Here's one way that `app.py` could be written:

```
import argparse
import importlib
import logging
import os
import sys

def main(args=None):
    scriptname = os.path.basename(__file__)
    parser = argparse.ArgumentParser(scriptname)
    levels = ('DEBUG', 'INFO', 'WARNING', 'ERROR', 'CRITICAL')
    parser.add_argument('--log-level', default='INFO',
choices=levels)
    subparsers = parser.add_subparsers(dest='command',
                                       help='Available
commands:')
    start_cmd = subparsers.add_parser('start', help='Start a
service')
    start_cmd.add_argument('name', metavar='NAME',
                           help='Name of service to start')
    stop_cmd = subparsers.add_parser('stop',
                                     help='Stop one or more
services')
    stop_cmd.add_argument('names', metavar='NAME', nargs='+',
                          help='Name of service to stop')
    restart_cmd = subparsers.add_parser('restart',
                                        help='Restart one or
more services')
    restart_cmd.add_argument('names', metavar='NAME',
nargs='+',
                             help='Name of service to restart')
    options = parser.parse_args()
    # the code to dispatch commands could all be in this file.
For the purposes
    # of illustration only, we implement each command in a
separate module.
    try:
        mod = importlib.import_module(options.command)
        cmd = getattr(mod, 'command')
    except (ImportError, AttributeError):
        print('Unable to find the code for command \'%s\'' %
options.command)
        return 1
    # Could get fancy here and load configuration from file or
dictionary
    logging.basicConfig(level=options.log_level,
                        format='%(levelname)s %(name)s %
(message)s')
    cmd(options)

if __name__ == '__main__':
    sys.exit(main())
```

And the `start`, `stop` and `restart` commands can be implemented in separate modules, like so for starting:

```
# start.py
import logging

logger = logging.getLogger(__name__)

def command(options):
    logger.debug('About to start %s', options.name)
    # actually do the command processing here ...
    logger.info('Started the \'%s\' service.', options.name)
```

and thus for stopping:

```
# stop.py
import logging

logger = logging.getLogger(__name__)

def command(options):
    n = len(options.names)
    if n == 1:
        plural = ''
        services = '\'%s\'' % options.names[0]
    else:
        plural = 's'
        services = ', '.join('\'%s\'' % name for name in 
options.names)
        i = services.rfind(', ')
        services = services[:i] + ' and ' + services[i + 2:]
    logger.debug('About to stop %s', services)
    # actually do the command processing here ...
    logger.info('Stopped the %s service%s.', services, plural)
```

and similarly for restarting:

```
# restart.py
import logging

logger = logging.getLogger(__name__)
```

```
def command(options):
    n = len(options.names)
    if n == 1:
        plural = ''
        services = '\'%s\'' % options.names[0]
    else:
        plural = 's'
        services = ', '.join('\'%s\'' % name for name in
options.names)
        i = services.rfind(', ')
        services = services[:i] + ' and ' + services[i + 2:]
    logger.debug('About to restart %s', services)
    # actually do the command processing here ...
    logger.info('Restarted the %s service%s.', services,
plural)
```

If we run this application with the default log level, we get output like this:

```
$ python app.py start foo
INFO start Started the 'foo' service.

$ python app.py stop foo bar
INFO stop Stopped the 'foo' and 'bar' services.

$ python app.py restart foo bar baz
INFO restart Restarted the 'foo', 'bar' and 'baz' services.
```

The first word is the logging level, and the second word is the module or package name of the place where the event was logged.

If we change the logging level, then we can change the information sent to the log. For example, if we want more information:

```
$ python app.py --log-level DEBUG start foo
DEBUG start About to start foo
INFO start Started the 'foo' service.

$ python app.py --log-level DEBUG stop foo bar
```

```
DEBUG stop About to stop 'foo' and 'bar'
INFO stop Stopped the 'foo' and 'bar' services.

$ python app.py --log-level DEBUG restart foo bar baz
DEBUG restart About to restart 'foo', 'bar' and 'baz'
INFO restart Restarted the 'foo', 'bar' and 'baz' services.
```

And if we want less:

```
$ python app.py --log-level WARNING start foo
$ python app.py --log-level WARNING stop foo bar
$ python app.py --log-level WARNING restart foo bar baz
```

In this case, the commands don’t print anything to the console, since nothing at `WARNING` level or above is logged by them.

A Qt GUI for logging

A question that comes up from time to time is about how to log to a GUI application. The Qt [https://www.qt.io/] framework is a popular cross-platform UI framework with Python bindings using PySide2 [https://pypi.org/project/PySide2/] or PyQt5 [https://pypi.org/project/PyQt5/] libraries.

The following example shows how to log to a Qt GUI. This introduces a simple `QtHandler` class which takes a callable, which should be a slot in the main thread that does GUI updates. A worker thread is also created to show how you can log to the GUI from both the UI itself (via a button for manual logging) as well as a worker thread doing work in the background (here, just logging messages at random levels with random short delays in between).

The worker thread is implemented using Qt’s `QThread` class rather than the **`threading`** module, as there are circumstances where one has to use `QThread`, which offers better integration with other `Qt` components.

The code should work with recent releases of any of `PySide6`, `PyQt6`, `PySide2` or `PyQt5`. You should be able to adapt the approach to earlier versions of Qt. Please refer to the comments in the code snippet for more detailed information.

```
import datetime
import logging
import random
import sys
import time

# Deal with minor differences between different Qt packages
try:
    from PySide6 import QtCore, QtGui, QtWidgets
    Signal = QtCore.Signal
    Slot = QtCore.Slot
except ImportError:
    try:
        from PyQt6 import QtCore, QtGui, QtWidgets
        Signal = QtCore.pyqtSignal
        Slot = QtCore.pyqtSlot
    except ImportError:
        try:
            from PySide2 import QtCore, QtGui, QtWidgets
            Signal = QtCore.Signal
            Slot = QtCore.Slot
        except ImportError:
            from PyQt5 import QtCore, QtGui, QtWidgets
            Signal = QtCore.pyqtSignal
            Slot = QtCore.pyqtSlot

logger = logging.getLogger(__name__)


#
# Signals need to be contained in a QObject or subclass in
order to be correctly
```

```
# initialized.
#
class Signaller(QtCore.QObject):
    signal = Signal(str, logging.LogRecord)

#
# Output to a Qt GUI is only supposed to happen on the main
thread. So, this
# handler is designed to take a slot function which is set up
to run in the main
# thread. In this example, the function takes a string argument
which is a
# formatted log message, and the log record which generated it.
The formatted
# string is just a convenience - you could format a string for
output any way
# you like in the slot function itself.
#
# You specify the slot function to do whatever GUI updates you
want. The handler
# doesn't know or care about specific UI elements.
#
class QtHandler(logging.Handler):
    def __init__(self, slotfunc, *args, **kwargs):
        super().__init__(*args, **kwargs)
        self.signaller = Signaller()
        self.signaller.signal.connect(slotfunc)

    def emit(self, record):
        s = self.format(record)
        self.signaller.signal.emit(s, record)

#
# This example uses QThreads, which means that the threads at
the Python level
# are named something like "Dummy-1". The function below gets
the Qt name of the
# current thread.
#
def ctname():
    return QtCore.QThread.currentThread().objectName()
```

```
#
# Used to generate random levels for logging.
#
LEVELS = (logging.DEBUG, logging.INFO, logging.WARNING, 
logging.ERROR,
          logging.CRITICAL)

#
# This worker class represents work that is done in a thread 
separate to the
# main thread. The way the thread is kicked off to do work is 
via a button press
# that connects to a slot in the worker.
#
# Because the default threadName value in the LogRecord isn't 
much use, we add
# a qThreadName which contains the QThread name as computed 
above, and pass that
# value in an "extra" dictionary which is used to update the 
LogRecord with the
# QThread name.
#
# This example worker just outputs messages sequentially, 
interspersed with
# random delays of the order of a few seconds.
#
class Worker(QtCore.QObject):
    @Slot()
    def start(self):
        extra = {'qThreadName': ctname() }
        logger.debug('Started work', extra=extra)
        i = 1
        # Let the thread run until interrupted. This allows 
reasonably clean
        # thread termination.
        while not 
QtCore.QThread.currentThread().isInterruptionRequested():
            delay = 0.5 + random.random() * 2
            time.sleep(delay)
            try:
                if random.random() < 0.1:
                    raise ValueError('Exception raised: %d' % 
i)
```

```
                else:
                    level = random.choice(LEVELS)
                    logger.log(level, 'Message after delay of
%3.1f: %d', delay, i, extra=extra)
            except ValueError as e:
                logger.exception('Failed: %s', e, extra=extra)
            i += 1

#
# Implement a simple UI for this cookbook example. This
contains:
#
# * A read-only text edit window which holds formatted log
messages
# * A button to start work and log stuff in a separate thread
# * A button to log something from the main thread
# * A button to clear the log window
#
class Window(QtWidgets.QWidget):

    COLORS = {
        logging.DEBUG: 'black',
        logging.INFO: 'blue',
        logging.WARNING: 'orange',
        logging.ERROR: 'red',
        logging.CRITICAL: 'purple',
    }

    def __init__(self, app):
        super().__init__()
        self.app = app
        self.textedit = te = QtWidgets.QPlainTextEdit(self)
        # Set whatever the default monospace font is for the
platform
        f = QtGui.QFont('nosuchfont')
        if hasattr(f, 'Monospace'):
            f.setStyleHint(f.Monospace)
        else:
            f.setStyleHint(f.StyleHint.Monospace)  # for Qt6
        te.setFont(f)
        te.setReadOnly(True)
        PB = QtWidgets.QPushButton
        self.work_button = PB('Start background work', self)
        self.log_button = PB('Log a message at a random level',
self)
        self.clear_button = PB('Clear log window', self)
        self.handler = h = QtHandler(self.update_status)
        # Remember to use qThreadName rather than threadName in
the format string.
        fs = '%(asctime)s %(qThreadName)-12s %(levelname)-8s %
(message)s'
        formatter = logging.Formatter(fs)
        h.setFormatter(formatter)
        logger.addHandler(h)
        # Set up to terminate the QThread when we exit
        app.aboutToQuit.connect(self.force_quit)

        # Lay out all the widgets
        layout = QtWidgets.QVBoxLayout(self)
        layout.addWidget(te)
        layout.addWidget(self.work_button)
        layout.addWidget(self.log_button)
        layout.addWidget(self.clear_button)
        self.setFixedSize(900, 400)

        # Connect the non-worker slots and signals
        self.log_button.clicked.connect(self.manual_update)
        self.clear_button.clicked.connect(self.clear_display)

        # Start a new worker thread and connect the slots for
the worker
        self.start_thread()
        self.work_button.clicked.connect(self.worker.start)
        # Once started, the button should be disabled
        self.work_button.clicked.connect(lambda :
self.work_button.setEnabled(False))

    def start_thread(self):
        self.worker = Worker()
        self.worker_thread = QtCore.QThread()
        self.worker.setObjectName('Worker')
        self.worker_thread.setObjectName('WorkerThread')  # for
qThreadName
        self.worker.moveToThread(self.worker_thread)
        # This will start an event loop in the worker thread
        self.worker_thread.start()
    def kill_thread(self):
        # Just tell the worker to stop, then tell it to quit
and wait for that
        # to happen
        self.worker_thread.requestInterruption()
        if self.worker_thread.isRunning():
            self.worker_thread.quit()
            self.worker_thread.wait()
        else:
            print('worker has already exited.')

    def force_quit(self):
        # For use when the window is closed
        if self.worker_thread.isRunning():
            self.kill_thread()

    # The functions below update the UI and run in the main
thread because
    # that's where the slots are set up

    @Slot(str, logging.LogRecord)
    def update_status(self, status, record):
        color = self.COLORS.get(record.levelno, 'black')
        s = '<pre><font color="%s">%s</font></pre>' % (color,
status)
        self.textedit.appendHtml(s)

    @Slot()
    def manual_update(self):
        # This function uses the formatted message passed in,
but also uses
        # information from the record to format the message in
an appropriate
        # color according to its severity (level).
        level = random.choice(LEVELS)
        extra = {'qThreadName': ctname() }
        logger.log(level, 'Manually logged!', extra=extra)

    @Slot()
    def clear_display(self):
        self.textedit.clear()
```

```
def main():
    QtCore.QThread.currentThread().setObjectName('MainThread')
    logging.getLogger().setLevel(logging.DEBUG)
    app = QtWidgets.QApplication(sys.argv)
    example = Window(app)
    example.show()
    if hasattr(app, 'exec'):
        rc = app.exec()
    else:
        rc = app.exec_()
    sys.exit(rc)

if __name__=='__main__':
    main()
```

Logging to syslog with RFC5424 support

Although **RFC 5424** [https://datatracker.ietf.org/doc/html/rfc5424.html] dates from 2009, most syslog servers are configured by default to use the older **RFC 3164** [https://datatracker.ietf.org/doc/html/rfc3164.html], which hails from 2001. When `logging` was added to Python in 2003, it supported the earlier (and only existing) protocol at the time. Since RFC5424 came out, as there has not been widespread deployment of it in syslog servers, the **`SysLogHandler`** functionality has not been updated.

RFC 5424 contains some useful features such as support for structured data, and if you need to be able to log to a syslog server with support for it, you can do so with a subclassed handler which looks something like this:

```
import datetime
import logging.handlers
import re
import socket
import time

class SysLogHandler5424(logging.handlers.SysLogHandler):

    tz_offset = re.compile(r'([+-]\d{2})(\d{2})$')
    escaped = re.compile(r'([\]"\\])')

    def __init__(self, *args, **kwargs):
        self.msgid = kwargs.pop('msgid', None)
        self.appname = kwargs.pop('appname', None)
        super().__init__(*args, **kwargs)

    def format(self, record):
        version = 1
        asctime =
datetime.datetime.fromtimestamp(record.created).isoformat()
        m = self.tz_offset.match(time.strftime('%z'))
        has_offset = False
        if m and time.timezone:
            hrs, mins = m.groups()
            if int(hrs) or int(mins):
                has_offset = True
        if not has_offset:
            asctime += 'Z'
        else:
            asctime += f'{hrs}:{mins}'
        try:
            hostname = socket.gethostname()
        except Exception:
            hostname = '-'
        appname = self.appname or '-'
        procid = record.process
        msgid = '-'
        msg = super().format(record)
        sdata = '-'
        if hasattr(record, 'structured_data'):
            sd = record.structured_data
            # This should be a dict where the keys are SD-ID
and the value is a
            # dict mapping PARAM-NAME to PARAM-VALUE (refer to
the RFC for what these
            # mean)
            # There's no error checking here - it's purely for
```

```
illustration, and you
            # can adapt this code for use in production
environments
            parts = []

            def replacer(m):
                g = m.groups()
                return '\\' + g[0]

            for sdid, dv in sd.items():
                part = f'[{sdid}'
                for k, v in dv.items():
                    s = str(v)
                    s = self.escaped.sub(replacer, s)
                    part += f' {k}="{s}"'
                part += ']'
                parts.append(part)
            sdata = ''.join(parts)
        return f'{version} {asctime} {hostname} {appname}
{procid} {msgid} {sdata} {msg}'
```

You’ll need to be familiar with RFC 5424 to fully understand the above code, and it may be that you have slightly different needs (e.g. for how you pass structural data to the log). Nevertheless, the above should be adaptable to your speciric needs. With the above handler, you’d pass structured data using something like this:

```
sd = {
    'foo@12345': {'bar': 'baz', 'baz': 'bozz', 'fizz':
r'buzz'},
    'foo@54321': {'rab': 'baz', 'zab': 'bozz', 'zzif': r'buzz'}
}
extra = {'structured_data': sd}
i = 1
logger.debug('Message %d', i, extra=extra)
```

How to treat a logger like an output stream

Sometimes, you need to interface to a third-party API which expects a file-like object to write to, but you want to direct the API’s output to a logger. You can do this using a class which wraps a logger with a file-like API. Here’s a short script illustrating such a class:

```
import logging

class LoggerWriter:
    def __init__(self, logger, level):
        self.logger = logger
        self.level = level

    def write(self, message):
        if message != '\n':  # avoid printing bare newlines, if
you like
            self.logger.log(self.level, message)

    def flush(self):
        # doesn't actually do anything, but might be expected
of a file-like
        # object - so optional depending on your situation
        pass

    def close(self):
        # doesn't actually do anything, but might be expected
of a file-like
        # object - so optional depending on your situation. You
might want
        # to set a flag so that later calls to write raise an
exception
        pass

def main():
    logging.basicConfig(level=logging.DEBUG)
    logger = logging.getLogger('demo')
    info_fp = LoggerWriter(logger, logging.INFO)
    debug_fp = LoggerWriter(logger, logging.DEBUG)
    print('An INFO message', file=info_fp)
    print('A DEBUG message', file=debug_fp)
```

```
if __name__ == "__main__":
    main()
```

When this script is run, it prints

```
INFO:demo:An INFO message
DEBUG:demo:A DEBUG message
```

You could also use `LoggerWriter` to redirect `sys.stdout` and `sys.stderr` by doing something like this:

```
import sys

sys.stdout = LoggerWriter(logger, logging.INFO)
sys.stderr = LoggerWriter(logger, logging.WARNING)
```

You should do this *after* configuring logging for your needs. In the above example, the **`basicConfig()`** call does this (using the `sys.stderr` value *before* it is overwritten by a `LoggerWriter` instance). Then, you'd get this kind of result:

```
>>> print('Foo')
INFO:demo:Foo
>>> print('Bar', file=sys.stderr)
WARNING:demo:Bar
>>>
```

Of course, the examples above show output according to the format used by **`basicConfig()`**, but you can use a different formatter when you configure logging.

Note that with the above scheme, you are somewhat at the mercy of buffering and the sequence of write calls which you are intercepting. For example, with the definition of `LoggerWriter` above, if you have the snippet

```
sys.stderr = LoggerWriter(logger, logging.WARNING)
1 / 0
```

then running the script results in

```
WARNING:demo:Traceback (most recent call last):

WARNING:demo:  File "/home/runner/cookbook-
loggerwriter/test.py", line 53, in <module>

WARNING:demo:
WARNING:demo:main()
WARNING:demo:  File "/home/runner/cookbook-
loggerwriter/test.py", line 49, in main

WARNING:demo:
WARNING:demo:1 / 0
WARNING:demo:ZeroDivisionError
WARNING:demo::
WARNING:demo:division by zero
```

As you can see, this output isn’t ideal. That’s because the underlying code which writes to `sys.stderr` makes multiple writes, each of which results in a separate logged line (for example, the last three lines above). To get around this problem, you need to buffer things and only output log lines when newlines are seen. Let’s use a slightly better implementation of `LoggerWriter`:

```
class BufferingLoggerWriter(LoggerWriter):
    def __init__(self, logger, level):
        super().__init__(logger, level)
        self.buffer = ''

    def write(self, message):
        if '\n' not in message:
            self.buffer += message
        else:
            parts = message.split('\n')
            if self.buffer:
                s = self.buffer + parts.pop(0)
                self.logger.log(self.level, s)
            self.buffer = parts.pop()
            for part in parts:
                self.logger.log(self.level, part)
```

This just buffers up stuff until a newline is seen, and then logs complete lines. With this approach, you get better output:

```
WARNING:demo:Traceback (most recent call last):
WARNING:demo:  File "/home/runner/cookbook-
loggerwriter/main.py", line 55, in <module>
WARNING:demo:    main()
WARNING:demo:  File "/home/runner/cookbook-
loggerwriter/main.py", line 52, in main
WARNING:demo:    1/0
WARNING:demo:ZeroDivisionError: division by zero
```

How to uniformly handle newlines in logging output

Usually, messages that are logged (say to console or file) consist of a single line of text. However, sometimes there is a need to handle messages with multiple lines - whether because a logging format string contains newlines, or logged data contains newlines. If you want to handle such messages uniformly, so that each line in the logged message appears uniformly formatted as if it was logged separately, you can do this using a handler mixin, as in the following snippet:

```
# Assume this is in a module mymixins.py
import copy

class MultilineMixin:
    def emit(self, record):
        s = record.getMessage()
        if '\n' not in s:
            super().emit(record)
        else:
            lines = s.splitlines()
            rec = copy.copy(record)
            rec.args = None
            for line in lines:
                rec.msg = line
                super().emit(rec)
```

You can use the mixin as in the following script:

```
import logging

from mymixins import MultilineMixin

logger = logging.getLogger(__name__)

class StreamHandler(MultilineMixin, logging.StreamHandler):
    pass

if __name__ == '__main__':
    logging.basicConfig(level=logging.DEBUG, format='%
(asctime)s %(levelname)-9s %(message)s',
                        handlers = [StreamHandler()])
    logger.debug('Single line')
    logger.debug('Multiple lines:\nfool me once ...')
    logger.debug('Another single line')
    logger.debug('Multiple lines:\n%s', 'fool me ...\ncan\'t
get fooled again')
```

The script, when run, prints something like:

```
2025-07-02 13:54:47,234 DEBUG     Single line
2025-07-02 13:54:47,234 DEBUG     Multiple lines:
2025-07-02 13:54:47,234 DEBUG     fool me once ...
2025-07-02 13:54:47,234 DEBUG     Another single line
2025-07-02 13:54:47,234 DEBUG     Multiple lines:
2025-07-02 13:54:47,234 DEBUG     fool me ...
2025-07-02 13:54:47,234 DEBUG     can't get fooled again
```

If, on the other hand, you are concerned about log injection [https://owasp.org/www-community/attacks/Log_Injection], you can use a formatter which escapes newlines, as per the following example:

```
import logging

logger = logging.getLogger(__name__)
```

```
class EscapingFormatter(logging.Formatter):
    def format(self, record):
        s = super().format(record)
        return s.replace('\n', r'\n')

if __name__ == '__main__':
    h = logging.StreamHandler()
    h.setFormatter(EscapingFormatter('%(asctime)s %
(levelname)-9s %(message)s'))
    logging.basicConfig(level=logging.DEBUG, handlers = [h])
    logger.debug('Single line')
    logger.debug('Multiple lines:\nfool me once ...')
    logger.debug('Another single line')
    logger.debug('Multiple lines:\n%s', 'fool me ...\ncan\'t
get fooled again')
```

You can, of course, use whatever escaping scheme makes the most sense for you. The script, when run, should produce output like this:

```
2025-07-09 06:47:33,783 DEBUG     Single line
2025-07-09 06:47:33,783 DEBUG     Multiple lines:\nfool me once
...
2025-07-09 06:47:33,783 DEBUG     Another single line
2025-07-09 06:47:33,783 DEBUG     Multiple lines:\nfool me
...\ncan't get fooled again
```

Escaping behaviour can’t be the stdlib default , as it would break backwards compatibility.

Patterns to avoid

Although the preceding sections have described ways of doing things you might need to do or deal with, it is worth mentioning some usage patterns which are *unhelpful*, and which should therefore be avoided in most cases. The following sections are in no particular order.

Opening the same log file multiple times

On Windows, you will generally not be able to open the same file multiple times as this will lead to a “file is in use by another process” error. However, on POSIX platforms you’ll not get any errors if you open the same file multiple times. This could be done accidentally, for example by:

- Adding a file handler more than once which references the same file (e.g. by a copy/paste/forget-to-change error).
- Opening two files that look different, as they have different names, but are the same because one is a symbolic link to the other.
- Forking a process, following which both parent and child have a reference to the same file. This might be through use of the `multiprocessing` module, for example.

Opening a file multiple times might *appear* to work most of the time, but can lead to a number of problems in practice:

- Logging output can be garbled because multiple threads or processes try to write to the same file. Although logging guards against concurrent use of the same handler instance by multiple threads, there is no such protection if concurrent writes are attempted by two different threads using two different handler instances which happen to point to the same file.
- An attempt to delete a file (e.g. during file rotation) silently fails, because there is another reference pointing to it. This can lead to confusion and wasted debugging time - log entries end up in unexpected places, or are lost altogether. Or a file that was supposed to be moved remains in place, and grows in size unexpectedly despite size-based rotation being supposedly in place.

Use the techniques outlined in Logging to a single file from multiple processes to circumvent such issues.

Using loggers as attributes in a class or passing them as parameters

While there might be unusual cases where you’ll need to do this, in general there is no point because loggers are singletons. Code can always access a given logger instance by name using `logging.getLogger(name)`, so passing instances around and holding them as instance attributes is pointless. Note that in other languages such as Java and C#, loggers are often static class attributes. However, this pattern doesn’t make sense in Python, where the module (and not the class) is the unit of software decomposition.

Adding handlers other than `NullHandler` to a logger in a library

Configuring logging by adding handlers, formatters and filters is the responsibility of the application developer, not the library developer. If you are maintaining a library, ensure that you don’t add handlers to any of your loggers other than a **`NullHandler`** instance.

Creating a lot of loggers

Loggers are singletons that are never freed during a script execution, and so creating lots of loggers will use up memory which can’t then be freed. Rather than create a logger per e.g. file processed or network connection made, use the existing mechanisms for passing contextual information into your logs and restrict the loggers created to those describing areas within your application (generally modules, but occasionally slightly more fine-grained than that).

Other resources

See also

Module `logging`
API reference for the logging module.

Module `logging.config`
Configuration API for the logging module.

Module `logging.handlers`
Useful handlers included with the logging module.

Basic Tutorial

Advanced Tutorial

Regular Expression HOWTO

Author:: A.M. Kuchling <amk@amk.ca>

Abstract

This document is an introductory tutorial to using regular expressions in Python with the **re** module. It provides a gentler introduction than the corresponding section in the Library Reference.

Introduction

Regular expressions (called REs, or regexes, or regex patterns) are essentially a tiny, highly specialized programming language embedded inside Python and made available through the **re** module. Using this little language, you specify the rules for the set of possible strings that you want to match; this set might contain English sentences, or e-mail addresses, or TeX commands, or anything you like. You can then ask questions such as “Does this string match the pattern?”, or “Is there a match for the pattern anywhere in this string?”. You can also use REs to modify a string or to split it apart in various ways.

Regular expression patterns are compiled into a series of bytecodes which are then executed by a matching engine written in C. For advanced use, it may be necessary to pay careful attention to how the engine will execute a given RE, and write the RE in a certain way in order to produce bytecode that runs faster. Optimization isn't covered in this document, because it requires that you have a good understanding of the matching engine's internals.

The regular expression language is relatively small and restricted, so not all possible string processing tasks can be done using regular expressions. There are also tasks that *can* be done with regular expressions, but the expressions turn out to be very complicated. In these cases, you may be better off writing Python code to do the processing; while Python code will be slower than an elaborate regular expression, it will also probably be more understandable.

Simple Patterns

We'll start by learning about the simplest possible regular expressions. Since regular expressions are used to operate on strings, we'll begin with the most common task: matching characters.

For a detailed explanation of the computer science underlying regular expressions (deterministic and non-deterministic finite automata), you can refer to almost any textbook on writing compilers.

Matching Characters

Most letters and characters will simply match themselves. For example, the regular expression `test` will match the string `test` exactly. (You can enable a case-insensitive mode that would let this RE match `Test` or `TEST` as well; more about this later.)

There are exceptions to this rule; some characters are special *metacharacters*, and don't match themselves. Instead, they signal that some out-of-the-ordinary thing should be matched, or they affect other portions of the RE by repeating them or changing their meaning. Much of this document is devoted to discussing various metacharacters and what they do.

Here’s a complete list of the metacharacters; their meanings will be discussed in the rest of this HOWTO.

```
. ^ $ * + ? { } [ ] \ | ( )
```

The first metacharacters we’ll look at are `[` and `]`. They’re used for specifying a character class, which is a set of characters that you wish to match. Characters can be listed individually, or a range of characters can be indicated by giving two characters and separating them by a `'-'`. For example, `[abc]` will match any of the characters `a`, `b`, or `c`; this is the same as `[a-c]`, which uses a range to express the same set of characters. If you wanted to match only lowercase letters, your RE would be `[a-z]`.

Metacharacters (except `\`) are not active inside classes. For example, `[akm$]` will match any of the characters `'a'`, `'k'`, `'m'`, or `'$'`; `'$'` is usually a metacharacter, but inside a character class it’s stripped of its special nature.

You can match the characters not listed within the class by *complementing* the set. This is indicated by including a `'^'` as the first character of the class. For example, `[^5]` will match any character except `'5'`. If the caret appears elsewhere in a character class, it does not have special meaning. For example: `[5^]` will match either a `'5'` or a `'^'`.

Perhaps the most important metacharacter is the backslash, `\`. As in Python string literals, the backslash can be followed by various characters to signal various special sequences. It’s also used to escape all the metacharacters so you can still match them in patterns; for example, if you need to match a `[` or `\`, you can precede them with a backslash to remove their special meaning: `\[` or `\\`.

Some of the special sequences beginning with `'\'` represent predefined sets of characters that are often useful, such as the set of digits, the set of letters, or the set of anything that isn’t whitespace.

Let’s take an example: `\w` matches any alphanumeric character. If the regex pattern is expressed in bytes, this is equivalent to the class `[a-zA-Z0-9_]`. If the regex pattern is a string, `\w` will match all the characters marked as letters in the Unicode database provided by the **`unicodedata`** module. You can use the more restricted definition of `\w` in a string pattern by supplying the **`re.ASCII`** flag when compiling the regular expression.

The following list of special sequences isn’t complete. For a complete list of sequences and expanded class definitions for Unicode string patterns, see the last part of Regular Expression Syntax in the Standard Library reference. In general, the Unicode versions match any character that’s in the appropriate category in the Unicode database.

`\d`

Matches any decimal digit; this is equivalent to the class `[0-9]`.

`\D`

Matches any non-digit character; this is equivalent to the class `[^0-9]`.

`\s`

Matches any whitespace character; this is equivalent to the class `[ \t\n\r\f\v]`.

`\S`

Matches any non-whitespace character; this is equivalent to the class `[^ \t\n\r\f\v]`.

`\w`

Matches any alphanumeric character; this is equivalent to the class `[a-zA-Z0-9_]`.

`\W`

Matches any non-alphanumeric character; this is equivalent to the class `[^a-zA-Z0-9_]`.

These sequences can be included inside a character class. For example, `[\s,.]` is a character class that will match any whitespace character, or `','` or `'.'`.

The final metacharacter in this section is `.`. It matches anything except a newline character, and there's an alternate mode (**`re.DOTALL`**) where it will match even a newline. `.` is often used where you want to match "any character".

Repeating Things

Being able to match varying sets of characters is the first thing regular expressions can do that isn't already possible with the methods available on strings. However, if that was the only additional capability of regexes, they wouldn't be much of an advance. Another capability is that you can specify that portions of the RE must be repeated a certain number of times.

The first metacharacter for repeating things that we'll look at is `*`. `*` doesn't match the literal character `'*'`; instead, it specifies that the previous character can be matched zero or more times, instead of exactly once.

For example, `ca*t` will match `'ct'` (0 `'a'` characters), `'cat'` (1 `'a'`), `'caaat'` (3 `'a'` characters), and so forth.

Repetitions such as `*` are *greedy*; when repeating a RE, the matching engine will try to repeat it as many times as possible. If later portions of the pattern don’t match, the matching engine will then back up and try again with fewer repetitions.

A step-by-step example will make this more obvious. Let’s consider the expression `a[bcd]*b`. This matches the letter `'a'`, zero or more letters from the class `[bcd]`, and finally ends with a `'b'`. Now imagine matching this RE against the string `'abcbd'`.

Step	**Matched**	**Explanation**
1	`a`	The `a` in the RE matches.
2	`abcbd`	The engine matches `[bcd]*`, going as far as it can, which is to the end of the string.
3	*Failure*	The engine tries to match `b`, but the current position is at the end of the string, so it fails.
4	`abcb`	Back up, so that `[bcd]*` matches one less character.
5	*Failure*	Try `b` again, but the current position is at the last character, which is a `'d'`.

Step	Matched	Explanation
6	`abc`	Back up again, so that `[bcd]*` is only matching `bc`.
6	`abcb`	Try `b` again. This time the character at the current position is `'b'`, so it succeeds.

The end of the RE has now been reached, and it has matched `'abcb'`. This demonstrates how the matching engine goes as far as it can at first, and if no match is found it will then progressively back up and retry the rest of the RE again and again. It will back up until it has tried zero matches for `[bcd]*`, and if that subsequently fails, the engine will conclude that the string doesn’t match the RE at all.

Another repeating metacharacter is `+`, which matches one or more times. Pay careful attention to the difference between `*` and `+`; `*` matches *zero* or more times, so whatever’s being repeated may not be present at all, while `+` requires at least *one* occurrence. To use a similar example, `ca+t` will match `'cat'` (1 `'a'`), `'caaat'` (3 `'a'`s), but won’t match `'ct'`.

There are two more repeating operators or quantifiers. The question mark character, `?`, matches either once or zero times; you can think of it as marking something as being optional. For example, `home-?brew` matches either `'homebrew'` or `'home-brew'`.

The most complicated quantifier is `{m,n}`, where *m* and *n* are decimal integers. This quantifier means there must be at least *m* repetitions, and at most *n*. For example, `a/{1,3}b` will match `'a/b'`, `'a//b'`, and `'a///b'`. It won't match `'ab'`, which has no slashes, or `'a////b'`, which has four.

You can omit either *m* or *n*; in that case, a reasonable value is assumed for the missing value. Omitting *m* is interpreted as a lower limit of 0, while omitting *n* results in an upper bound of infinity.

The simplest case `{m}` matches the preceding item exactly *m* times. For example, `a/{2}b` will only match `'a//b'`.

Readers of a reductionist bent may notice that the three other quantifiers can all be expressed using this notation. `{0,}` is the same as `*`, `{1,}` is equivalent to `+`, and `{0,1}` is the same as `?`. It's better to use `*`, `+`, or `?` when you can, simply because they're shorter and easier to read.

Using Regular Expressions

Now that we've looked at some simple regular expressions, how do we actually use them in Python? The **`re`** module provides an interface to the regular expression engine, allowing you to compile REs into objects and then perform matches with them.

Compiling Regular Expressions

Regular expressions are compiled into pattern objects, which have methods for various operations such as searching for pattern matches or performing string substitutions.

```
>>> import re
>>> p = re.compile('ab*')
>>> p
re.compile('ab*')
```

`re.compile()` also accepts an optional *flags* argument, used to enable various special features and syntax variations. We'll go over the available settings later, but for now a single example will do:

```
>>> p = re.compile('ab*', re.IGNORECASE)
```

The RE is passed to `re.compile()` as a string. REs are handled as strings because regular expressions aren't part of the core Python language, and no special syntax was created for expressing them. (There are applications that don't need REs at all, so there's no need to bloat the language specification by including them.) Instead, the `re` module is simply a C extension module included with Python, just like the `socket` or `zlib` modules.

Putting REs in strings keeps the Python language simpler, but has one disadvantage which is the topic of the next section.

The Backslash Plague

As stated earlier, regular expressions use the backslash character (`'\'`) to indicate special forms or to allow special characters to be used without invoking their special meaning. This conflicts with Python's usage of the same character for the same purpose in string literals.

Let's say you want to write a RE that matches the string `\section`, which might be found in a LaTeX file. To figure out what to write in the program code, start with the desired string to be matched. Next, you must escape any backslashes and other metacharacters by preceding them with a backslash, resulting in the string `\\section`. The resulting string that must be passed to `re.compile()` must be

`\\section`. However, to express this as a Python string literal, both backslashes must be escaped *again*.

Characters	Stage
`\section`	Text string to be matched
`\\section`	Escaped backslash for `re.compile()`
`"\\\\section"`	Escaped backslashes for a string literal

In short, to match a literal backslash, one has to write `'\\\\'` as the RE string, because the regular expression must be `\\`, and each backslash must be expressed as `\\` inside a regular Python string literal. In REs that feature backslashes repeatedly, this leads to lots of repeated backslashes and makes the resulting strings difficult to understand.

The solution is to use Python's raw string notation for regular expressions; backslashes are not handled in any special way in a string literal prefixed with `'r'`, so `r"\n"` is a two-character string containing `'\'` and `'n'`, while `"\n"` is a one-character string containing a newline. Regular expressions will often be written in Python code using this raw string notation.

In addition, special escape sequences that are valid in regular expressions, but not valid as Python string literals, now result in a `DeprecationWarning` and will eventually become a `SyntaxError`, which means the sequences will be invalid if raw string notation or escaping the backslashes isn't used.

Regular String	Raw string
`"ab*"`	`r"ab*"`
`"\\\\section"`	`r"\\section"`
`"\\w+\\s+\\1"`	`r"\w+\s+\1"`

Performing Matches

Once you have an object representing a compiled regular expression, what do you do with it? Pattern objects have several methods and attributes. Only the most significant ones will be covered here; consult the `re` docs for a complete listing.

Method/Attribute	Purpose
`match()`	Determine if the RE matches at the beginning of the string.
`search()`	Scan through a string, looking for any location where this RE matches.
`findall()`	Find all substrings where the RE matches, and returns them as a list.

Method/Attribute	Purpose
`finditer()`	Find all substrings where the RE matches, and returns them as an iterator.

`match()` and **`search()`** return `None` if no match can be found. If they're successful, a match object instance is returned, containing information about the match: where it starts and ends, the substring it matched, and more.

You can learn about this by interactively experimenting with the **`re`** module.

This HOWTO uses the standard Python interpreter for its examples. First, run the Python interpreter, import the **`re`** module, and compile a RE:

```
>>> import re
>>> p = re.compile('[a-z]+')
>>> p
re.compile('[a-z]+')
```

Now, you can try matching various strings against the RE `[a-z]+`. An empty string shouldn't match at all, since `+` means 'one or more repetitions'. **`match()`** should return `None` in this case, which will cause the interpreter to print no output. You can explicitly print the result of **`match()`** to make this clear.

```
>>> p.match("")
>>> print(p.match(""))
None
```

Now, let's try it on a string that it should match, such as `tempo`. In this case, **`match()`** will return a match object, so you should store the result in a variable for later use.

```
>>> m = p.match('tempo')
>>> m
<re.Match object; span=(0, 5), match='tempo'>
```

Now you can query the match object for information about the matching string. Match object instances also have several methods and attributes; the most important ones are:

Method/Attribute	Purpose
`group()`	Return the string matched by the RE
`start()`	Return the starting position of the match
`end()`	Return the ending position of the match
`span()`	Return a tuple containing the (start, end) positions of the match

Trying these methods will soon clarify their meaning:

```
>>> m.group()
'tempo'
>>> m.start(), m.end()
(0, 5)
>>> m.span()
(0, 5)
```

`group()` returns the substring that was matched by the RE. **`start()`** and **`end()`** return the starting and ending index of the match. **`span()`** returns both start and end indexes in a single tuple. Since the **`match()`** method only checks if the RE matches at the start of a string, **`start()`** will always be zero. However, the **`search()`** method of patterns scans through the string, so the match may not start at zero in that case.

```
>>> print(p.match('::: message'))
None
>>> m = p.search('::: message'); print(m)
<re.Match object; span=(4, 11), match='message'>
>>> m.group()
'message'
>>> m.span()
(4, 11)
```

In actual programs, the most common style is to store the match object in a variable, and then check if it was `None`. This usually looks like:

```
p = re.compile( ... )
m = p.match( 'string goes here' )
if m:
    print('Match found: ', m.group())
else:
    print('No match')
```

Two pattern methods return all of the matches for a pattern. **`findall()`** returns a list of matching strings:

```
>>> p = re.compile(r'\d+')
>>> p.findall('12 drummers drumming, 11 pipers piping, 10 lords
a-leaping')
['12', '11', '10']
```

The `r` prefix, making the literal a raw string literal, is needed in this example because escape sequences in a normal “cooked” string literal that are not recognized by Python, as opposed to regular expressions, now result in a

`DeprecationWarning` and will eventually become a **`SyntaxError`**. See The Backslash Plague.

`findall()` has to create the entire list before it can be returned as the result. The **`finditer()`** method returns a sequence of match object instances as an iterator:

```
>>> iterator = p.finditer('12 drummers drumming, 11 ... 10
...')
>>> iterator
<callable_iterator object at 0x...>
>>> for match in iterator:
...     print(match.span())
...
(0, 2)
(22, 24)
(29, 31)
```

Module-Level Functions

You don't have to create a pattern object and call its methods; the **`re`** module also provides top-level functions called **`match()`**, **`search()`**, **`findall()`**, **`sub()`**, and so forth. These functions take the same arguments as the corresponding pattern method with the RE string added as the first argument, and still return either `None` or a match object instance.

```
>>> print(re.match(r'From\s+', 'Fromage amk'))
None
>>> re.match(r'From\s+', 'From amk Thu May 14 19:12:10 1998')
<re.Match object; span=(0, 5), match='From '>
```

Under the hood, these functions simply create a pattern object for you and call the appropriate method on it. They also store the compiled object in a cache, so future calls using the same RE won't need to parse the pattern again and again.

Should you use these module-level functions, or should you get the pattern and call its methods yourself? If you’re accessing a regex within a loop, pre-compiling it will save a few function calls. Outside of loops, there’s not much difference thanks to the internal cache.

Compilation Flags

Compilation flags let you modify some aspects of how regular expressions work. Flags are available in the `re` module under two names, a long name such as `IGNORECASE` and a short, one-letter form such as `I`. (If you’re familiar with Perl’s pattern modifiers, the one-letter forms use the same letters; the short form of `re.VERBOSE` is `re.X`, for example.) Multiple flags can be specified by bitwise OR-ing them; `re.I | re.M` sets both the `I` and `M` flags, for example.

Here’s a table of the available flags, followed by a more detailed explanation of each one.

Flag	Meaning
`ASCII`, `A`	Makes several escapes like `\w`, `\b`, `\s` and `\d` match only on ASCII characters with the respective property.
`DOTALL`, `S`	Make `.` match any character, including newlines.
`IGNORECASE`, `I`	Do case-insensitive matches.
`LOCALE`, `L`	Do a locale-aware match.

Flag	**Meaning**
`MULTILINE`, `M`	Multi-line matching, affecting `^` and `$`.
`VERBOSE`, `X` (for ‘extended’)	Enable verbose REs, which can be organized more cleanly and understandably.

re.I
re.IGNORECASE

Perform case-insensitive matching; character class and literal strings will match letters by ignoring case. For example, `[A-Z]` will match lowercase letters, too. Full Unicode matching also works unless the `ASCII` flag is used to disable non-ASCII matches. When the Unicode patterns `[a-z]` or `[A-Z]` are used in combination with the `IGNORECASE` flag, they will match the 52 ASCII letters and 4 additional non-ASCII letters: ‘İ’ (U+0130, Latin capital letter I with dot above), ‘ı’ (U+0131, Latin small letter dotless i), ‘ſ’ (U+017F, Latin small letter long s) and ‘K’ (U+212A, Kelvin sign). `Spam` will match `'Spam'`, `'spam'`, `'spAM'`, or `'ſpam'` (the latter is matched only in Unicode mode). This lowercasing doesn’t take the current locale into account; it will if you also set the `LOCALE` flag.

re.L
re.LOCALE

Make `\w`, `\W`, `\b`, `\B` and case-insensitive matching dependent on the current locale instead of the Unicode database.

Locales are a feature of the C library intended to help in writing programs that take account of language differences. For example, if you're processing encoded French text, you'd want to be able to write `\w+` to match words, but `\w` only matches the character class `[A-Za-z]` in bytes patterns; it won't match bytes corresponding to `é` or `ç`. If your system is configured properly and a French locale is selected, certain C functions will tell the program that the byte corresponding to `é` should also be considered a letter. Setting the **`LOCALE`** flag when compiling a regular expression will cause the resulting compiled object to use these C functions for `\w`; this is slower, but also enables `\w+` to match French words as you'd expect. The use of this flag is discouraged in Python 3 as the locale mechanism is very unreliable, it only handles one "culture" at a time, and it only works with 8-bit locales. Unicode matching is already enabled by default in Python 3 for Unicode (str) patterns, and it is able to handle different locales/languages.

re.M
re.MULTILINE

(`^` and `$` haven't been explained yet; they'll be introduced in section More Metacharacters.)

Usually `^` matches only at the beginning of the string, and `$` matches only at the end of the string and immediately before the newline (if any) at the end of the string. When this flag is specified, `^` matches at the beginning of the string and at the beginning of each line within the string, immediately following each newline. Similarly, the `$` metacharacter matches either at the end of the string and at the end of each line (immediately preceding each newline).

re.S

re.DOTALL

Makes the `'.'` special character match any character at all, including a newline; without this flag, `'.'` will match anything *except* a newline.

re.A
re.ASCII

Make `\w`, `\W`, `\b`, `\B`, `\s` and `\S` perform ASCII-only matching instead of full Unicode matching. This is only meaningful for Unicode patterns, and is ignored for byte patterns.

re.X
re.VERBOSE

This flag allows you to write regular expressions that are more readable by granting you more flexibility in how you can format them. When this flag has been specified, whitespace within the RE string is ignored, except when the whitespace is in a character class or preceded by an unescaped backslash; this lets you organize and indent the RE more clearly. This flag also lets you put comments within a RE that will be ignored by the engine; comments are marked by a `'#'` that's neither in a character class or preceded by an unescaped backslash.

For example, here's a RE that uses **`re.VERBOSE`**; see how much easier it is to read?

```
charref = re.compile(r"""
 &[#]                # Start of a numeric entity reference
 (
     0[0-7]+         # Octal form
   | [0-9]+          # Decimal form
   | x[0-9a-fA-F]+   # Hexadecimal form
 )
 ;                   # Trailing semicolon
""", re.VERBOSE)
```

Without the verbose setting, the RE would look like this:

```
charref = re.compile("&#(0[0-7]+"
                     "|[0-9]+"
                     "|x[0-9a-fA-F]+);")
```

In the above example, Python’s automatic concatenation of string literals has been used to break up the RE into smaller pieces, but it’s still more difficult to understand than the version using **`re.VERBOSE`**.

More Pattern Power

So far we’ve only covered a part of the features of regular expressions. In this section, we’ll cover some new metacharacters, and how to use groups to retrieve portions of the text that was matched.

More Metacharacters

There are some metacharacters that we haven’t covered yet. Most of them will be covered in this section.

Some of the remaining metacharacters to be discussed are *zero-width assertions*. They don’t cause the engine to advance through the string; instead, they consume no characters at all, and simply succeed or fail. For example, `\b` is an assertion that the current position is located at a word boundary; the position isn’t changed by the `\b` at all. This means that zero-width assertions should never be repeated, because if they match once at a given location, they can obviously be matched an infinite number of times.

`|`

Alternation, or the “or” operator. If *A* and *B* are regular expressions, `A|B` will match any string that matches either *A* or *B*. `|` has very low precedence in order to make it work reasonably when you’re alternating multi-character strings. `Crow|Servo` will match either `'Crow'` or `'Servo'`, not `'Cro'`, a `'w'` or an `'S'`, and `'ervo'`.

To match a literal `'|'`, use `\|`, or enclose it inside a character class, as in `[|]`.

`^`

Matches at the beginning of lines. Unless the **`MULTILINE`** flag has been set, this will only match at the beginning of the string. In **`MULTILINE`** mode, this also matches immediately after each newline within the string.

For example, if you wish to match the word `From` only at the beginning of a line, the RE to use is `^From`.

```
>>> print(re.search('^From', 'From Here to Eternity'))
<re.Match object; span=(0, 4), match='From'>
>>> print(re.search('^From', 'Reciting From Memory'))
None
```

To match a literal `'^'`, use `\^`.

`$`

Matches at the end of a line, which is defined as either the end of the string, or any location followed by a newline character.

```
>>> print(re.search('}$', '{block}'))
<re.Match object; span=(6, 7), match='}'>
>>> print(re.search('}$', '{block} '))
None
>>> print(re.search('}$', '{block}\n'))
<re.Match object; span=(6, 7), match='}'>
```

To match a literal `'$'`, use `\$` or enclose it inside a character class, as in `[$]`.

`\A`

Matches only at the start of the string. When not in **`MULTILINE`** mode, `\A` and `^` are effectively the same. In **`MULTILINE`** mode, they're different: `\A` still matches only at the beginning of the string, but `^` may match at any location inside the string that follows a newline character.

`\z`

Matches only at the end of the string.

`\Z`

The same as `\z`. For compatibility with old Python versions.

`\b`

Word boundary. This is a zero-width assertion that matches only at the beginning or end of a word. A word is defined as a sequence of alphanumeric characters, so the end of a word is indicated by whitespace or a non-alphanumeric character.

The following example matches `class` only when it's a complete word; it won't match when it's contained inside another word.

```
>>> p = re.compile(r'\bclass\b')
>>> print(p.search('no class at all'))
<re.Match object; span=(3, 8), match='class'>
>>> print(p.search('the declassified algorithm'))
None
>>> print(p.search('one subclass is'))
None
```

There are two subtleties you should remember when using this special sequence. First, this is the worst collision between Python's string literals and regular expression sequences. In Python's string literals, `\b` is the backspace character, ASCII value 8. If you're not using raw strings, then Python will convert the `\b` to a backspace, and your RE won’t match as you expect it to. The following example looks the same as our previous RE, but omits the `'r'` in front of the RE string.

```
>>> p = re.compile('\bclass\b')
>>> print(p.search('no class at all'))
None
>>> print(p.search('\b' + 'class' + '\b'))
<re.Match object; span=(0, 7), match='\x08class\x08'>
```

Second, inside a character class, where there’s no use for this assertion, `\b` represents the backspace character, for compatibility with Python’s string literals.

`\B`

Another zero-width assertion, this is the opposite of `\b`, only matching when the current position is not at a word boundary.

Grouping

Frequently you need to obtain more information than just whether the RE matched or not. Regular expressions are often used to dissect strings by writing a RE divided into several subgroups which match different components of interest. For example, an RFC-822 header line is divided into a header name and a value, separated by a `':'`, like this:

```
From: author@example.com
User-Agent: Thunderbird 1.5.0.9 (X11/20061227)
MIME-Version: 1.0
To: editor@example.com
```

This can be handled by writing a regular expression which matches an entire header line, and has one group which matches the header name, and another group which matches the header’s value.

Groups are marked by the `'('`, `')'` metacharacters. `'('` and `')'` have much the same meaning as they do in mathematical expressions; they group together the expressions contained inside them, and you can repeat the contents of a group with a quantifier, such as `*`, `+`, `?`, or `{m,n}`. For example, `(ab)*` will match zero or more repetitions of `ab`.

```
>>> p = re.compile('(ab)*')
>>> print(p.match('ababababab').span())
(0, 10)
```

Groups indicated with `'('`, `')'` also capture the starting and ending index of the text that they match; this can be retrieved by passing an argument to **`group()`**, **`start()`**, **`end()`**, and **`span()`**. Groups are numbered starting with 0. Group 0 is always present; it's the whole RE, so match object methods all have group 0 as their default argument. Later we'll see how to express groups that don't capture the span of text that they match.

```
>>> p = re.compile('(a)b')
>>> m = p.match('ab')
>>> m.group()
'ab'
>>> m.group(0)
'ab'
```

Subgroups are numbered from left to right, from 1 upward. Groups can be nested; to determine the number, just count the opening parenthesis characters, going from left to right.

```
>>> p = re.compile('(a(b)c)d')
>>> m = p.match('abcd')
>>> m.group(0)
'abcd'
>>> m.group(1)
'abc'
>>> m.group(2)
'b'
```

`group()` can be passed multiple group numbers at a time, in which case it will return a tuple containing the corresponding values for those groups.

```
>>> m.group(2,1,2)
('b', 'abc', 'b')
```

The `groups()` method returns a tuple containing the strings for all the subgroups, from 1 up to however many there are.

```
>>> m.groups()
('abc', 'b')
```

Backreferences in a pattern allow you to specify that the contents of an earlier capturing group must also be found at the current location in the string. For example, `\1` will succeed if the exact contents of group 1 can be found at the current position, and fails otherwise. Remember that Python’s string literals also use a backslash followed by numbers to allow including arbitrary characters in a string, so be sure to use a raw string when incorporating backreferences in a RE.

For example, the following RE detects doubled words in a string.

```
>>> p = re.compile(r'\b(\w+)\s+\1\b')
>>> p.search('Paris in the the spring').group()
'the the'
```

Backreferences like this aren’t often useful for just searching through a string — there are few text formats which repeat data in this way — but you’ll soon find out that they’re *very* useful when performing string substitutions.

Non-capturing and Named Groups

Elaborate REs may use many groups, both to capture substrings of interest, and to group and structure the RE itself. In complex REs, it becomes difficult to keep track of the group numbers. There are two features which help with this problem. Both of them use a common syntax for regular expression extensions, so we'll look at that first.

Perl 5 is well known for its powerful additions to standard regular expressions. For these new features the Perl developers couldn't choose new single-keystroke metacharacters or new special sequences beginning with `\` without making Perl's regular expressions confusingly different from standard REs. If they chose `&` as a new metacharacter, for example, old expressions would be assuming that `&` was a regular character and wouldn't have escaped it by writing `\&` or `[&]`.

The solution chosen by the Perl developers was to use `(?...)` as the extension syntax. `?` immediately after a parenthesis was a syntax error because the `?` would have nothing to repeat, so this didn't introduce any compatibility problems. The characters immediately after the `?` indicate what extension is being used, so `(?=foo)` is one thing (a positive lookahead assertion) and `(?:foo)` is something else (a non-capturing group containing the subexpression `foo`).

Python supports several of Perl's extensions and adds an extension syntax to Perl's extension syntax. If the first character after the question mark is a `P`, you know that it's an extension that's specific to Python.

Now that we've looked at the general extension syntax, we can return to the features that simplify working with groups in complex REs.

Sometimes you’ll want to use a group to denote a part of a regular expression, but aren’t interested in retrieving the group’s contents. You can make this fact explicit by using a non-capturing group: `(?:...)`, where you can replace the `...` with any other regular expression.

```
>>> m = re.match("([abc])+", "abc")
>>> m.groups()
('c',)
>>> m = re.match("(?:[abc])+", "abc")
>>> m.groups()
()
```

Except for the fact that you can’t retrieve the contents of what the group matched, a non-capturing group behaves exactly the same as a capturing group; you can put anything inside it, repeat it with a repetition metacharacter such as `*`, and nest it within other groups (capturing or non-capturing). `(?:...)` is particularly useful when modifying an existing pattern, since you can add new groups without changing how all the other groups are numbered. It should be mentioned that there’s no performance difference in searching between capturing and non-capturing groups; neither form is any faster than the other.

A more significant feature is named groups: instead of referring to them by numbers, groups can be referenced by a name.

The syntax for a named group is one of the Python-specific extensions: `(?P<name>...)`. *name* is, obviously, the name of the group. Named groups behave exactly like capturing groups, and additionally associate a name with a group. The match object methods that deal with capturing groups all accept either integers that refer to the group by number or strings that contain the desired group’s name. Named groups are still given numbers, so you can retrieve information about a group in two ways:

```
>>> p = re.compile(r'(?P<word>\b\w+\b)')
>>> m = p.search( '(((( Lots of punctuation )))' )
>>> m.group('word')
'Lots'
>>> m.group(1)
'Lots'
```

Additionally, you can retrieve named groups as a dictionary with **`groupdict()`**:

```
>>> m = re.match(r'(?P<first>\w+) (?P<last>\w+)', 'Jane Doe')
>>> m.groupdict()
{'first': 'Jane', 'last': 'Doe'}
```

Named groups are handy because they let you use easily remembered names, instead of having to remember numbers. Here’s an example RE from the **`imaplib`** module:

```
InternalDate = re.compile(r'INTERNALDATE "'
        r'(?P<day>[ 123][0-9])-(?P<mon>[A-Z][a-z][a-z])-'
        r'(?P<year>[0-9][0-9][0-9][0-9])'
        r' (?P<hour>[0-9][0-9]):(?P<min>[0-9][0-9]):(?P<sec>[0-
9][0-9])'
        r' (?P<zonen>[-+])(?P<zoneh>[0-9][0-9])(?P<zonem>[0-9]
[0-9])'
        r'"')
```

It’s obviously much easier to retrieve `m.group('zonem')`, instead of having to remember to retrieve group 9.

The syntax for backreferences in an expression such as `(...)\1` refers to the number of the group. There’s naturally a variant that uses the group name instead of the number. This is another Python extension: `(?P=name)` indicates that the contents of the group called *name* should again be matched at the current point. The regular expression for finding doubled words, `\b(\w+)\s+\1\b` can also be written as `\b(?P<word>\w+)\s+(?P=word)\b`:

```
>>> p = re.compile(r'\b(?P<word>\w+)\s+(?P=word)\b')
>>> p.search('Paris in the the spring').group()
'the the'
```

Lookahead Assertions

Another zero-width assertion is the lookahead assertion. Lookahead assertions are available in both positive and negative form, and look like this:

`(?=...)`
: Positive lookahead assertion. This succeeds if the contained regular expression, represented here by `...`, successfully matches at the current location, and fails otherwise. But, once the contained expression has been tried, the matching engine doesn't advance at all; the rest of the pattern is tried right where the assertion started.

`(?!...)`
: Negative lookahead assertion. This is the opposite of the positive assertion; it succeeds if the contained expression *doesn't* match at the current position in the string.

To make this concrete, let's look at a case where a lookahead is useful. Consider a simple pattern to match a filename and split it apart into a base name and an extension, separated by a `.`. For example, in `news.rc`, `news` is the base name, and `rc` is the filename's extension.

The pattern to match this is quite simple:

```
.*[.].*$
```

Notice that the `.` needs to be treated specially because it's a metacharacter, so it's inside a character class to only match that specific character. Also notice the trailing `$`; this is added to ensure that all the rest of the string must be included in the extension. This regular expression matches `foo.bar` and `autoexec.bat` and `sendmail.cf` and `printers.conf`.

Now, consider complicating the problem a bit; what if you want to match filenames where the extension is not `bat`? Some incorrect attempts:

```
.*[.][^b].*$
```

The first attempt above tries to exclude `bat` by requiring that the first character of the extension is not a `b`. This is wrong, because the pattern also doesn't match `foo.bar`.

```
.*[.]([^b]..|.[^a].|..[^t])$
```

The expression gets messier when you try to patch up the first solution by requiring one of the following cases to match: the first character of the extension isn't `b`; the second character isn't `a`; or the third character isn't `t`. This accepts `foo.bar` and rejects `autoexec.bat`, but it requires a three-letter extension and won't accept a filename with a two-letter extension such as `sendmail.cf`. We'll complicate the pattern again in an effort to fix it.

```
.*[.]([^b].?.?|.[^a]?.?|..?[^t]?)$
```

In the third attempt, the second and third letters are all made optional in order to allow matching extensions shorter than three characters, such as `sendmail.cf`.

The pattern's getting really complicated now, which makes it hard to read and understand. Worse, if the problem changes and you want to exclude both `bat` and `exe` as extensions, the pattern would get even more complicated and confusing.

A negative lookahead cuts through all this confusion:

```
.*[.](?!bat$)[^.]*$
```

The negative lookahead means: if the expression `bat` doesn't match at this point, try the rest of the pattern; if `bat$` does match, the whole pattern will fail. The trailing `$` is required to ensure that something like `sample.batch`, where the extension only starts with `bat`, will be allowed. The `[^.]*` makes sure that the pattern works when there are multiple dots in the filename.

Excluding another filename extension is now easy; simply add it as an alternative inside the assertion. The following pattern excludes filenames that end in either `bat` or `exe`:

```
.*[.](?!bat$|exe$)[^.]*$
```

Modifying Strings

Up to this point, we've simply performed searches against a static string. Regular expressions are also commonly used to modify strings in various ways, using the following pattern methods:

Method/Attribute	Purpose
`split()`	Split the string into a list, splitting it wherever the RE matches

Method/Attribute	Purpose
`sub()`	Find all substrings where the RE matches, and replace them with a different string
`subn()`	Does the same thing as **`sub()`**, but returns the new string and the number of replacements

Splitting Strings

The `split()` method of a pattern splits a string apart wherever the RE matches, returning a list of the pieces. It’s similar to the `split()` method of strings but provides much more generality in the delimiters that you can split by; string **`split()`** only supports splitting by whitespace or by a fixed string. As you’d expect, there’s a module-level `re.split()` function, too.

.split(*string*[, *maxsplit=0*])

> Split *string* by the matches of the regular expression. If capturing parentheses are used in the RE, then their contents will also be returned as part of the resulting list. If *maxsplit* is nonzero, at most *maxsplit* splits are performed.

You can limit the number of splits made, by passing a value for *maxsplit*. When *maxsplit* is nonzero, at most *maxsplit* splits will be made, and the remainder of the string is returned as the final element of the list. In the following example, the delimiter is any sequence of non-alphanumeric characters.

```
>>> p = re.compile(r'\W+')
>>> p.split('This is a test, short and sweet, of split().')
['This', 'is', 'a', 'test', 'short', 'and', 'sweet', 'of',
'split', '']
>>> p.split('This is a test, short and sweet, of split().', 3)
['This', 'is', 'a', 'test, short and sweet, of split().']
```

Sometimes you’re not only interested in what the text between delimiters is, but also need to know what the delimiter was. If capturing parentheses are used in the RE, then their values are also returned as part of the list. Compare the following calls:

```
>>> p = re.compile(r'\W+')
>>> p2 = re.compile(r'(\W+)')
>>> p.split('This... is a test.')
['This', 'is', 'a', 'test', '']
>>> p2.split('This... is a test.')
['This', '... ', 'is', ' ', 'a', ' ', 'test', '.', '']
```

The module-level function `re.split()` adds the RE to be used as the first argument, but is otherwise the same.

```
>>> re.split(r'[\W]+', 'Words, words, words.')
['Words', 'words', 'words', '']
>>> re.split(r'([\W]+)', 'Words, words, words.')
['Words', ', ', 'words', ', ', 'words', '.', '']
>>> re.split(r'[\W]+', 'Words, words, words.', 1)
['Words', 'words, words.']
```

Search and Replace

Another common task is to find all the matches for a pattern, and replace them with a different string. The `sub()` method takes a replacement value, which can be either a string or a function, and the string to be processed.

.sub(*replacement*, *string*[, *count=0*])

Returns the string obtained by replacing the leftmost non-overlapping occurrences of the RE in *string* by the replacement *replacement*. If the pattern isn’t found, *string* is returned unchanged.

The optional argument *count* is the maximum number of pattern occurrences to be replaced; *count* must be a non-negative integer. The default value of 0 means to replace all occurrences.

Here’s a simple example of using the **`sub()`** method. It replaces colour names with the word `colour`:

```
>>> p = re.compile('(blue|white|red)')
>>> p.sub('colour', 'blue socks and red shoes')
'colour socks and colour shoes'
>>> p.sub('colour', 'blue socks and red shoes', count=1)
'colour socks and red shoes'
```

The **`subn()`** method does the same work, but returns a 2-tuple containing the new string value and the number of replacements that were performed:

```
>>> p = re.compile('(blue|white|red)')
>>> p.subn('colour', 'blue socks and red shoes')
('colour socks and colour shoes', 2)
>>> p.subn('colour', 'no colours at all')
('no colours at all', 0)
```

Empty matches are replaced only when they’re not adjacent to a previous empty match.

```
>>> p = re.compile('x*')
>>> p.sub('-', 'abxd')
'-a-b--d-'
```

If *replacement* is a string, any backslash escapes in it are processed. That is, `\n` is converted to a single newline character, `\r` is converted to a carriage return, and so forth. Unknown escapes such as `\&` are left alone. Backreferences, such as `\6`, are replaced with the substring matched by the corresponding group in the RE. This lets you incorporate portions of the original text in the resulting replacement string.

This example matches the word `section` followed by a string enclosed in `{`, `}`, and changes `section` to `subsection`:

```
>>> p = re.compile('section{ ( [^}]* ) }', re.VERBOSE)
>>> p.sub(r'subsection{\1}','section{First} section{second}')
'subsection{First} subsection{second}'
```

There's also a syntax for referring to named groups as defined by the `(?P<name>...)` syntax. `\g<name>` will use the substring matched by the group named `name`, and `\g<number>` uses the corresponding group number. `\g<2>` is therefore equivalent to `\2`, but isn't ambiguous in a replacement string such as `\g<2>0`. (`\20` would be interpreted as a reference to group 20, not a reference to group 2 followed by the literal character `'0'`.) The following substitutions are all equivalent, but use all three variations of the replacement string.

```
>>> p = re.compile('section{ (?P<name> [^}]* ) }', re.VERBOSE)
>>> p.sub(r'subsection{\1}','section{First}')
'subsection{First}'
>>> p.sub(r'subsection{\g<1>}','section{First}')
'subsection{First}'
>>> p.sub(r'subsection{\g<name>}','section{First}')
'subsection{First}'
```

replacement can also be a function, which gives you even more control. If *replacement* is a function, the function is called for every non-overlapping occurrence of *pattern*. On each call, the function is passed a match object argument for the match and can use this information to compute the desired replacement string and return it.

In the following example, the replacement function translates decimals into hexadecimal:

```
>>> def hexrepl(match):
...     "Return the hex string for a decimal number"
...     value = int(match.group())
...     return hex(value)
...
>>> p = re.compile(r'\d+')
>>> p.sub(hexrepl, 'Call 65490 for printing, 49152 for user
code.')
'Call 0xffd2 for printing, 0xc000 for user code.'
```

When using the module-level `re.sub()` function, the pattern is passed as the first argument. The pattern may be provided as an object or as a string; if you need to specify regular expression flags, you must either use a pattern object as the first parameter, or use embedded modifiers in the pattern string, e.g. `sub("(?i)b+", "x", "bbbb BBBB")` returns `'x x'`.

Common Problems

Regular expressions are a powerful tool for some applications, but in some ways their behaviour isn’t intuitive and at times they don’t behave the way you may expect them to. This section will point out some of the most common pitfalls.

Use String Methods

Sometimes using the `re` module is a mistake. If you’re matching a fixed string, or a single character class, and you’re not using any `re` features such as the `IGNORECASE` flag, then the full power of regular expressions may not be required. Strings have several methods for performing operations with fixed strings and they’re usually much faster, because the implementation is a single small C loop that’s been optimized for the purpose, instead of the large, more generalized regular expression engine.

One example might be replacing a single fixed string with another one; for example, you might replace `word` with `deed`. **`re.sub()`** seems like the function to use for this, but consider the **`replace()`** method. Note that **`replace()`** will also replace `word` inside words, turning `swordfish` into `sdeedfish`, but the naive RE `word` would have done that, too. (To avoid performing the substitution on parts of words, the pattern would have to be `\bword\b`, in order to require that `word` have a word boundary on either side. This takes the job beyond **`replace()`**’s abilities.)

Another common task is deleting every occurrence of a single character from a string or replacing it with another single character. You might do this with something like `re.sub('\n', ' ', S)`, but **`translate()`** is capable of doing both tasks and will be faster than any regular expression operation can be.

In short, before turning to the **`re`** module, consider whether your problem can be solved with a faster and simpler string method.

match() versus search()

The **`match()`** function only checks if the RE matches at the beginning of the string while **`search()`** will scan forward through the string for a match. It’s important to keep this distinction in mind. Remember, **`match()`** will only report a successful match which will start at 0; if the match wouldn’t start at zero, **`match()`** will *not* report it.

```
>>> print(re.match('super', 'superstition').span())
(0, 5)
>>> print(re.match('super', 'insuperable'))
None
```

On the other hand, `search()` will scan forward through the string, reporting the first match it finds.

```
>>> print(re.search('super', 'superstition').span())
(0, 5)
>>> print(re.search('super', 'insuperable').span())
(2, 7)
```

Sometimes you’ll be tempted to keep using `re.match()`, and just add `.*` to the front of your RE. Resist this temptation and use `re.search()` instead. The regular expression compiler does some analysis of REs in order to speed up the process of looking for a match. One such analysis figures out what the first character of a match must be; for example, a pattern starting with `Crow` must match starting with a `'C'`. The analysis lets the engine quickly scan through the string looking for the starting character, only trying the full match if a `'C'` is found.

Adding `.*` defeats this optimization, requiring scanning to the end of the string and then backtracking to find a match for the rest of the RE. Use `re.search()` instead.

Greedy versus Non-Greedy

When repeating a regular expression, as in `a*`, the resulting action is to consume as much of the pattern as possible. This fact often bites you when you’re trying to match a pair of balanced delimiters, such as the angle brackets surrounding an HTML tag. The naive pattern for matching a single HTML tag doesn't work because of the greedy nature of `.*`.

```
>>> s = '<html><head><title>Title</title>'
>>> len(s)
32
>>> print(re.match('<.*>', s).span())
(0, 32)
>>> print(re.match('<.*>', s).group())
<html><head><title>Title</title>
```

The RE matches the `'<'` in `'<html>'`, and the `.*` consumes the rest of the string. There's still more left in the RE, though, and the `>` can't match at the end of the string, so the regular expression engine has to backtrack character by character until it finds a match for the `>`. The final match extends from the `'<'` in `'<html>'` to the `'>'` in `'</title>'`, which isn't what you want.

In this case, the solution is to use the non-greedy quantifiers `*?`, `+?`, `??`, or `{m,n}?`, which match as *little* text as possible. In the above example, the `'>'` is tried immediately after the first `'<'` matches, and when it fails, the engine advances a character at a time, retrying the `'>'` at every step. This produces just the right result:

```
>>> print(re.match('<.*?>', s).group())
<html>
```

(Note that parsing HTML or XML with regular expressions is painful. Quick-and-dirty patterns will handle common cases, but HTML and XML have special cases that will break the obvious regular expression; by the time you've written a regular expression that handles all of the possible cases, the patterns will be *very* complicated. Use an HTML or XML parser module for such tasks.)

Using re.VERBOSE

By now you’ve probably noticed that regular expressions are a very compact notation, but they’re not terribly readable. REs of moderate complexity can become lengthy collections of backslashes, parentheses, and metacharacters, making them difficult to read and understand.

For such REs, specifying the `re.VERBOSE` flag when compiling the regular expression can be helpful, because it allows you to format the regular expression more clearly.

The `re.VERBOSE` flag has several effects. Whitespace in the regular expression that *isn’t* inside a character class is ignored. This means that an expression such as `dog | cat` is equivalent to the less readable `dog|cat`, but `[a b]` will still match the characters `'a'`, `'b'`, or a space. In addition, you can also put comments inside a RE; comments extend from a `#` character to the next newline. When used with triple-quoted strings, this enables REs to be formatted more neatly:

```
pat = re.compile(r"""
 \s*                 # Skip leading whitespace
 (?P<header>[^:]+)   # Header name
 \s* :               # Whitespace, and a colon
 (?P<value>.*?)      # The header's value -- *? used to
                     # lose the following trailing whitespace
 \s*$                # Trailing whitespace to end-of-line
""", re.VERBOSE)
```

This is far more readable than:

```
pat = re.compile(r"\s*(?P<header>[^:]+)\s*:(?P<value>.*?)\s*$")
```

Feedback

Regular expressions are a complicated topic. Did this document help you understand them? Were there parts that were unclear, or Problems you encountered that weren’t covered here? If so, please send suggestions for improvements to the author.

The most complete book on regular expressions is almost certainly Jeffrey Friedl’s Mastering Regular Expressions, published by O’Reilly. Unfortunately, it exclusively concentrates on Perl and Java’s flavours of regular expressions, and doesn’t contain any Python material at all, so it won’t be useful as a reference for programming in Python. (The first edition covered Python’s now-removed `regex` module, which won’t help you much.) Consider checking it out from your library.

Socket Programming HOWTO

Author:: Gordon McMillan

Abstract

Sockets are used nearly everywhere, but are one of the most severely misunderstood technologies around. This is a 10,000 foot overview of sockets. It’s not really a tutorial - you’ll still have work to do in getting things operational. It doesn’t cover the fine points (and there are a lot of them), but I hope it will give you enough background to begin using them decently.

Sockets

I’m only going to talk about INET (i.e. IPv4) sockets, but they account for at least 99% of the sockets in use. And I’ll only talk about STREAM (i.e. TCP) sockets - unless you really know what you’re doing (in which case this HOWTO isn’t for you!), you’ll get better behavior and performance from a STREAM socket than anything else. I will try to clear up the mystery of what a socket is, as well as some hints on how to work with blocking and non-blocking sockets. But I’ll start by talking about blocking sockets. You’ll need to know how they work before dealing with non-blocking sockets.

Part of the trouble with understanding these things is that “socket” can mean a number of subtly different things, depending on context. So first, let’s make a distinction between a “client” socket - an endpoint of a conversation, and a “server” socket, which is more like a switchboard operator. The client application (your browser, for example) uses “client” sockets exclusively; the web server it’s talking to uses both “server” sockets and “client” sockets.

History

Of the various forms of IPC, sockets are by far the most popular. On any given platform, there are likely to be other forms of IPC that are faster, but for cross-platform communication, sockets are about the only game in town.

They were invented in Berkeley as part of the BSD flavor of Unix. They spread like wildfire with the internet. With good reason — the combination of sockets with INET makes talking to arbitrary machines around the world unbelievably easy (at least compared to other schemes).

Creating a Socket

Roughly speaking, when you clicked on the link that brought you to this page, your browser did something like the following:

```
# create an INET, STREAMing socket
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
# now connect to the web server on port 80 - the normal http
port
s.connect(("www.python.org", 80))
```

When the `connect` completes, the socket `s` can be used to send in a request for the text of the page. The same socket will read the reply, and then be destroyed. That’s right, destroyed. Client sockets are normally only used for one exchange (or a small set of sequential exchanges).

What happens in the web server is a bit more complex. First, the web server creates a “server socket”:

```
# create an INET, STREAMing socket
serversocket = socket.socket(socket.AF_INET,
socket.SOCK_STREAM)
# bind the socket to a public host, and a well-known port
serversocket.bind((socket.gethostname(), 80))
# become a server socket
serversocket.listen(5)
```

A couple things to notice: we used `socket.gethostname()` so that the socket would be visible to the outside world. If we had used `s.bind(('localhost', 80))` or `s.bind(('127.0.0.1', 80))` we would still have a “server” socket, but one that was only visible within the same machine. `s.bind(('', 80))` specifies that the socket is reachable by any address the machine happens to have.

A second thing to note: low number ports are usually reserved for “well known” services (HTTP, SNMP etc). If you’re playing around, use a nice high number (4 digits).

Finally, the argument to `listen` tells the socket library that we want it to queue up as many as 5 connect requests (the normal max) before refusing outside connections. If the rest of the code is written properly, that should be plenty.

Now that we have a “server” socket, listening on port 80, we can enter the mainloop of the web server:

```
while True:
    # accept connections from outside
    (clientsocket, address) = serversocket.accept()
    # now do something with the clientsocket
    # in this case, we'll pretend this is a threaded server
    ct = make_client_thread(clientsocket)
    ct.start()
```

There’s actually 3 general ways in which this loop could work - dispatching a thread to handle `clientsocket`, create a new process to handle `clientsocket`, or restructure this app to use non-blocking sockets, and multiplex between our “server” socket and any active `clientsocket`s using `select`. More about that later. The important thing to understand now is this: this is *all* a “server” socket does. It doesn’t send any data. It doesn’t receive any data. It just produces “client” sockets. Each `clientsocket` is created in response to some *other* “client” socket doing a `connect()` to the host and port we’re bound to. As soon as we’ve created that `clientsocket`, we go back to listening for more connections. The two “clients” are free to chat it up - they are using some dynamically allocated port which will be recycled when the conversation ends.

IPC

If you need fast IPC between two processes on one machine, you should look into pipes or shared memory. If you do decide to use AF_INET sockets, bind the “server” socket to `'localhost'`. On most platforms, this will take a shortcut around a couple of layers of network code and be quite a bit faster.

See also

The **`multiprocessing`** integrates cross-platform IPC into a higher-level API.

Using a Socket

The first thing to note, is that the web browser’s “client” socket and the web server’s “client” socket are identical beasts. That is, this is a “peer to peer” conversation. Or to put it another way, *as the designer, you will have to decide what the rules of etiquette are for a conversation*. Normally, the `connect`ing socket starts the conversation, by sending in a request, or perhaps a signon. But that’s a design decision - it’s not a rule of sockets.

Now there are two sets of verbs to use for communication. You can use `send` and `recv`, or you can transform your client socket into a file-like beast and use `read` and `write`. The latter is the way Java presents its sockets. I’m not going to talk about it here, except to warn you that you need to use `flush` on sockets. These are buffered “files”, and a common mistake is to `write` something, and then `read` for a reply. Without a `flush` in there, you may wait forever for the reply, because the request may still be in your output buffer.

Now we come to the major stumbling block of sockets - `send` and `recv` operate on the network buffers. They do not necessarily handle all the bytes you hand them (or expect from them), because their major focus is handling the network buffers. In general, they return when the associated network buffers have been filled (`send`) or emptied (`recv`). They then tell you how many bytes they handled. It is *your* responsibility to call them again until your message has been completely dealt with.

When a `recv` returns 0 bytes, it means the other side has closed (or is in the process of closing) the connection. You will not receive any more data on this connection. Ever. You may be able to send data successfully; I’ll talk more about this later.

A protocol like HTTP uses a socket for only one transfer. The client sends a request, then reads a reply. That’s it. The socket is discarded. This means that a client can detect the end of the reply by receiving 0 bytes.

But if you plan to reuse your socket for further transfers, you need to realize that *there is no* EOT *on a socket.* I repeat: if a socket `send` or `recv` returns after handling 0 bytes, the connection has been broken. If the connection has *not* been broken, you may wait on a `recv` forever, because the socket will *not* tell you that there’s nothing more to read (for now). Now if you think about that a bit, you’ll come to realize a fundamental truth of sockets: *messages must either be fixed length* (yuck), *or be delimited* (shrug), *or indicate how long they are* (much better), *or end by shutting down the connection*. The choice is entirely yours, (but some ways are righter than others).

Assuming you don’t want to end the connection, the simplest solution is a fixed length message:

```
class MySocket:
    """demonstration class only
      - coded for clarity, not efficiency
    """

    def __init__(self, sock=None):
        if sock is None:
            self.sock = socket.socket(
                            socket.AF_INET, socket.SOCK_STREAM)
        else:
            self.sock = sock

    def connect(self, host, port):
        self.sock.connect((host, port))

    def mysend(self, msg):
        totalsent = 0
        while totalsent < MSGLEN:
            sent = self.sock.send(msg[totalsent:])
            if sent == 0:
                raise RuntimeError("socket connection broken")
            totalsent = totalsent + sent

    def myreceive(self):
        chunks = []
        bytes_recd = 0
        while bytes_recd < MSGLEN:
            chunk = self.sock.recv(min(MSGLEN - bytes_recd,
2048))
            if chunk == b'':
                raise RuntimeError("socket connection broken")
            chunks.append(chunk)
            bytes_recd = bytes_recd + len(chunk)
        return b''.join(chunks)
```

The sending code here is usable for almost any messaging scheme - in Python you send strings, and you can use `len()` to determine its length (even if it has embedded `\0` characters). It’s mostly the receiving code that gets more complex. (And in C, it’s not much worse, except you can’t use `strlen` if the message has embedded `\0`s.)

The easiest enhancement is to make the first character of the message an indicator of message type, and have the type determine the length. Now you have two `recv`s - the first to get (at least) that first character so you can look up the length, and the second in a loop to get the rest. If you decide to go the delimited route, you’ll be receiving in some arbitrary chunk size, (4096 or 8192 is frequently a good match for network buffer sizes), and scanning what you’ve received for a delimiter.

One complication to be aware of: if your conversational protocol allows multiple messages to be sent back to back (without some kind of reply), and you pass `recv` an arbitrary chunk size, you may end up reading the start of a following message. You’ll need to put that aside and hold onto it, until it’s needed.

Prefixing the message with its length (say, as 5 numeric characters) gets more complex, because (believe it or not), you may not get all 5 characters in one `recv`. In playing around, you'll get away with it; but in high network loads, your code will very quickly break unless you use two `recv` loops - the first to determine the length, the second to get the data part of the message. Nasty. This is also when you'll discover that `send` does not always manage to get rid of everything in one pass. And despite having read this, you will eventually get bit by it!

In the interests of space, building your character, (and preserving my competitive position), these enhancements are left as an exercise for the reader. Lets move on to cleaning up.

Binary Data

It is perfectly possible to send binary data over a socket. The major problem is that not all machines use the same formats for binary data. For example, network byte order [https://en.wikipedia.org/wiki/Endianness#Networking] is big-endian, with the most significant byte first, so a 16 bit integer with the value `1` would be the two hex bytes `00 01`. However, most common processors (x86/AMD64, ARM, RISC-V), are little-endian, with the least significant byte first - that same `1` would be `01 00`.

Socket libraries have calls for converting 16 and 32 bit integers - `ntohl, htonl, ntohs, htons` where "n" means *network* and "h" means *host*, "s" means *short* and "l" means *long*. Where network order is host order, these do nothing, but where the machine is byte-reversed, these swap the bytes around appropriately.

In these days of 64-bit machines, the ASCII representation of binary data is frequently smaller than the binary representation. That’s because a surprising amount of the time, most integers have the value 0, or maybe 1. The string `"0"` would be two bytes, while a full 64-bit integer would be 8. Of course, this doesn’t fit well with fixed-length messages. Decisions, decisions.

Disconnecting

Strictly speaking, you’re supposed to use `shutdown` on a socket before you `close` it. The `shutdown` is an advisory to the socket at the other end. Depending on the argument you pass it, it can mean “I’m not going to send anymore, but I’ll still listen”, or “I’m not listening, good riddance!”. Most socket libraries, however, are so used to programmers neglecting to use this piece of etiquette that normally a `close` is the same as `shutdown(); close()`. So in most situations, an explicit `shutdown` is not needed.

One way to use `shutdown` effectively is in an HTTP-like exchange. The client sends a request and then does a `shutdown(1)`. This tells the server “This client is done sending, but can still receive.” The server can detect “EOF” by a receive of 0 bytes. It can assume it has the complete request. The server sends a reply. If the `send` completes successfully then, indeed, the client was still receiving.

Python takes the automatic shutdown a step further, and says that when a socket is garbage collected, it will automatically do a `close` if it’s needed. But relying on this is a very bad habit. If your socket just disappears without doing a `close`, the socket at the other end may hang indefinitely, thinking you’re just being slow. *Please* `close` your sockets when you’re done.

When Sockets Die

Probably the worst thing about using blocking sockets is what happens when the other side comes down hard (without doing a `close`). Your socket is likely to hang. TCP is a reliable protocol, and it will wait a long, long time before giving up on a connection. If you're using threads, the entire thread is essentially dead. There's not much you can do about it. As long as you aren't doing something dumb, like holding a lock while doing a blocking read, the thread isn't really consuming much in the way of resources. Do *not* try to kill the thread - part of the reason that threads are more efficient than processes is that they avoid the overhead associated with the automatic recycling of resources. In other words, if you do manage to kill the thread, your whole process is likely to be screwed up.

Non-blocking Sockets

If you've understood the preceding, you already know most of what you need to know about the mechanics of using sockets. You'll still use the same calls, in much the same ways. It's just that, if you do it right, your app will be almost inside-out.

In Python, you use `socket.setblocking(False)` to make it non-blocking. In C, it's more complex, (for one thing, you'll need to choose between the BSD flavor `O_NONBLOCK` and the almost indistinguishable POSIX flavor `O_NDELAY`, which is completely different from `TCP_NODELAY`), but it's the exact same idea. You do this after creating the socket, but before using it. (Actually, if you're nuts, you can switch back and forth.)

The major mechanical difference is that `send`, `recv`, `connect` and `accept` can return without having done anything. You have (of course) a number of choices. You can check return code and error codes and generally drive yourself crazy. If you don’t believe me, try it sometime. Your app will grow large, buggy and suck CPU. So let’s skip the brain-dead solutions and do it right.

Use `select`.

In C, coding `select` is fairly complex. In Python, it’s a piece of cake, but it’s close enough to the C version that if you understand `select` in Python, you’ll have little trouble with it in C:

```
ready_to_read, ready_to_write, in_error = \
               select.select(
                  potential_readers,
                  potential_writers,
                  potential_errs,
                  timeout)
```

You pass `select` three lists: the first contains all sockets that you might want to try reading; the second all the sockets you might want to try writing to, and the last (normally left empty) those that you want to check for errors. You should note that a socket can go into more than one list. The `select` call is blocking, but you can give it a timeout. This is generally a sensible thing to do - give it a nice long timeout (say a minute) unless you have good reason to do otherwise.

In return, you will get three lists. They contain the sockets that are actually readable, writable and in error. Each of these lists is a subset (possibly empty) of the corresponding list you passed in.

If a socket is in the output readable list, you can be as-close-to-certain-as-we-ever-get-in-this-business that a `recv` on that socket will return *something*. Same idea for the writable list.

You’ll be able to send *something*. Maybe not all you want to, but *something* is better than nothing. (Actually, any reasonably healthy socket will return as writable - it just means outbound network buffer space is available.)

If you have a “server” socket, put it in the potential_readers list. If it comes out in the readable list, your `accept` will (almost certainly) work. If you have created a new socket to `connect` to someone else, put it in the potential_writers list. If it shows up in the writable list, you have a decent chance that it has connected.

Actually, `select` can be handy even with blocking sockets. It’s one way of determining whether you will block - the socket returns as readable when there’s something in the buffers. However, this still doesn’t help with the problem of determining whether the other end is done, or just busy with something else.

Portability alert: On Unix, `select` works both with the sockets and files. Don’t try this on Windows. On Windows, `select` works with sockets only. Also note that in C, many of the more advanced socket options are done differently on Windows. In fact, on Windows I usually use threads (which work very, very well) with my sockets.

Sorting Techniques

Author:: Andrew Dalke and Raymond Hettinger

Python lists have a built-in `list.sort()` method that modifies the list in-place. There is also a `sorted()` built-in function that builds a new sorted list from an iterable.

In this document, we explore the various techniques for sorting data using Python.

Sorting Basics

A simple ascending sort is very easy: just call the `sorted()` function. It returns a new sorted list:

```
>>> sorted([5, 2, 3, 1, 4])
[1, 2, 3, 4, 5]
```

You can also use the `list.sort()` method. It modifies the list in-place (and returns `None` to avoid confusion). Usually it’s less convenient than `sorted()` - but if you don’t need the original list, it’s slightly more efficient.

```
>>> a = [5, 2, 3, 1, 4]
>>> a.sort()
>>> a
[1, 2, 3, 4, 5]
```

Another difference is that the `list.sort()` method is only defined for lists. In contrast, the `sorted()` function accepts any iterable.

```
>>> sorted({1: 'D', 2: 'B', 3: 'B', 4: 'E', 5: 'A'})
[1, 2, 3, 4, 5]
```

Key Functions

The `list.sort()` method and the functions `sorted()`, `min()`, `max()`, `heapq.nsmallest()`, and `heapq.nlargest()` have a *key* parameter to specify a function (or other callable) to be called on each list element prior to making comparisons.

For example, here's a case-insensitive string comparison using `str.casefold()`:

```
>>> sorted("This is a test string from Andrew".split(),
key=str.casefold)
['a', 'Andrew', 'from', 'is', 'string', 'test', 'This']
```

The value of the *key* parameter should be a function (or other callable) that takes a single argument and returns a key to use for sorting purposes. This technique is fast because the key function is called exactly once for each input record.

A common pattern is to sort complex objects using some of the object's indices as keys. For example:

```
>>> student_tuples = [
...     ('john', 'A', 15),
...     ('jane', 'B', 12),
...     ('dave', 'B', 10),
... ]
>>> sorted(student_tuples, key=lambda student: student[2])   #
sort by age
[('dave', 'B', 10), ('jane', 'B', 12), ('john', 'A', 15)]
```

The same technique works for objects with named attributes. For example:

```
>>> class Student:
...     def __init__(self, name, grade, age):
...         self.name = name
...         self.grade = grade
...         self.age = age
...     def __repr__(self):
...         return repr((self.name, self.grade, self.age))

>>> student_objects = [
...     Student('john', 'A', 15),
...     Student('jane', 'B', 12),
...     Student('dave', 'B', 10),
... ]
>>> sorted(student_objects, key=lambda student: student.age)
# sort by age
[('dave', 'B', 10), ('jane', 'B', 12), ('john', 'A', 15)]
```

Objects with named attributes can be made by a regular class as shown above, or they can be instances of **`dataclass`** or a named tuple.

Operator Module Functions and Partial Function Evaluation

The key function patterns shown above are very common, so Python provides convenience functions to make accessor functions easier and faster. The **`operator`** module has **`itemgetter()`**, **`attrgetter()`**, and a **`methodcaller()`** function.

Using those functions, the above examples become simpler and faster:

```
>>> from operator import itemgetter, attrgetter

>>> sorted(student_tuples, key=itemgetter(2))
[('dave', 'B', 10), ('jane', 'B', 12), ('john', 'A', 15)]

>>> sorted(student_objects, key=attrgetter('age'))
[('dave', 'B', 10), ('jane', 'B', 12), ('john', 'A', 15)]
```

The operator module functions allow multiple levels of sorting. For example, to sort by *grade* then by *age*:

```
>>> sorted(student_tuples, key=itemgetter(1,2))
[('john', 'A', 15), ('dave', 'B', 10), ('jane', 'B', 12)]

>>> sorted(student_objects, key=attrgetter('grade', 'age'))
[('john', 'A', 15), ('dave', 'B', 10), ('jane', 'B', 12)]
```

The [`functools`] module provides another helpful tool for making key-functions. The [`partial()`] function can reduce the arity [https://en.wikipedia.org/wiki/Arity] of a multi-argument function making it suitable for use as a key-function.

```
>>> from functools import partial
>>> from unicodedata import normalize

>>> names = 'Zoë Åbjørn Núñez Élana Zeke Abe Nubia
Eloise'.split()

>>> sorted(names, key=partial(normalize, 'NFD'))
['Abe', 'Åbjørn', 'Eloise', 'Élana', 'Nubia', 'Núñez', 'Zeke',
'Zoë']

>>> sorted(names, key=partial(normalize, 'NFC'))
['Abe', 'Eloise', 'Nubia', 'Núñez', 'Zeke', 'Zoë', 'Åbjørn',
'Élana']
```

Ascending and Descending

Both `list.sort()` and `sorted()` accept a *reverse* parameter with a boolean value. This is used to flag descending sorts. For example, to get the student data in reverse *age* order:

```
>>> sorted(student_tuples, key=itemgetter(2), reverse=True)
[('john', 'A', 15), ('jane', 'B', 12), ('dave', 'B', 10)]

>>> sorted(student_objects, key=attrgetter('age'),
reverse=True)
[('john', 'A', 15), ('jane', 'B', 12), ('dave', 'B', 10)]
```

Sort Stability and Complex Sorts

Sorts are guaranteed to be stable [https://en.wikipedia.org/wiki/Sorting_algorithm#Stability]. That means that when multiple records have the same key, their original order is preserved.

```
>>> data = [('red', 1), ('blue', 1), ('red', 2), ('blue', 2)]
>>> sorted(data, key=itemgetter(0))
[('blue', 1), ('blue', 2), ('red', 1), ('red', 2)]
```

Notice how the two records for *blue* retain their original order so that `('blue', 1)` is guaranteed to precede `('blue', 2)`.

This wonderful property lets you build complex sorts in a series of sorting steps. For example, to sort the student data by descending *grade* and then ascending *age*, do the *age* sort first and then sort again using *grade*:

```
>>> s = sorted(student_objects, key=attrgetter('age'))     # 
sort on secondary key
>>> sorted(s, key=attrgetter('grade'), reverse=True)       # 
now sort on primary key, descending
[('dave', 'B', 10), ('jane', 'B', 12), ('john', 'A', 15)]
```

This can be abstracted out into a wrapper function that can take a list and tuples of field and order to sort them on multiple passes.

```
>>> def multisort(xs, specs):
...     for key, reverse in reversed(specs):
...         xs.sort(key=attrgetter(key), reverse=reverse)
...     return xs

>>> multisort(list(student_objects), (('grade', True), ('age', 
False)))
[('dave', 'B', 10), ('jane', 'B', 12), ('john', 'A', 15)]
```

The Timsort [https://en.wikipedia.org/wiki/Timsort] algorithm used in Python does multiple sorts efficiently because it can take advantage of any ordering already present in a dataset.

Decorate-Sort-Undecorate

This idiom is called Decorate-Sort-Undecorate after its three steps:

- First, the initial list is decorated with new values that control the sort order.
- Second, the decorated list is sorted.
- Finally, the decorations are removed, creating a list that contains only the initial values in the new order.

For example, to sort the student data by *grade* using the DSU approach:

```
>>> decorated = [(student.grade, i, student) for i, student in
enumerate(student_objects)]
>>> decorated.sort()
>>> [student for grade, i, student in decorated]
# undecorate
[('john', 'A', 15), ('jane', 'B', 12), ('dave', 'B', 10)]
```

This idiom works because tuples are compared lexicographically; the first items are compared; if they are the same then the second items are compared, and so on.

It is not strictly necessary in all cases to include the index *i* in the decorated list, but including it gives two benefits:

- The sort is stable – if two items have the same key, their order will be preserved in the sorted list.
- The original items do not have to be comparable because the ordering of the decorated tuples will be determined by at most the first two items. So for example the original list could contain complex numbers which cannot be sorted directly.

Another name for this idiom is Schwartzian transform [https://en.wikipedia.org/wiki/Schwartzian_transform], after Randal L. Schwartz, who popularized it among Perl programmers.

Now that Python sorting provides key-functions, this technique is not often needed.

Comparison Functions

Unlike key functions that return an absolute value for sorting, a comparison function computes the relative ordering for two inputs.

For example, a balance scale [https://upload.wikimedia.org/wikipedia/commons/1/17/Balance_à_tabac_1850.JPG] compares two samples giving a relative ordering: lighter, equal, or heavier. Likewise, a comparison function such as `cmp(a, b)` will return a negative value for less-than, zero if the inputs are equal, or a positive value for greater-than.

It is common to encounter comparison functions when translating algorithms from other languages. Also, some libraries provide comparison functions as part of their API. For example, **`locale.strcoll()`** is a comparison function.

To accommodate those situations, Python provides **`functools.cmp_to_key`** to wrap the comparison function to make it usable as a key function:

```
sorted(words, key=cmp_to_key(strcoll))  # locale-aware sort
order
```

Strategies For Unorderable Types and Values

A number of type and value issues can arise when sorting. Here are some strategies that can help:

- Convert non-comparable input types to strings prior to sorting:

```
>>> data = ['twelve', '11', 10]
>>> sorted(map(str, data))
['10', '11', 'twelve']
```

This is needed because most cross-type comparisons raise a `TypeError`.

- Remove special values prior to sorting:

```
>>> from math import isnan
>>> from itertools import filterfalse
>>> data = [3.3, float('nan'), 1.1, 2.2]
>>> sorted(filterfalse(isnan, data))
[1.1, 2.2, 3.3]
```

This is needed because the IEEE-754 standard [https://en.wikipedia.org/wiki/IEEE_754] specifies that, “Every NaN shall compare unordered with everything, including itself.”

Likewise, `None` can be stripped from datasets as well:

```
>>> data = [3.3, None, 1.1, 2.2]
>>> sorted(x for x in data if x is not None)
[1.1, 2.2, 3.3]
```

This is needed because `None` is not comparable to other types.

- Convert mapping types into sorted item lists before sorting:

```
>>> data = [{'a': 1}, {'b': 2}]
>>> sorted(data, key=lambda d: sorted(d.items()))
[{'a': 1}, {'b': 2}]
```

This is needed because dict-to-dict comparisons raise a `TypeError`.

- Convert set types into sorted lists before sorting:

```
>>> data = [{'a', 'b', 'c'}, {'b', 'c', 'd'}]
>>> sorted(map(sorted, data))
[['a', 'b', 'c'], ['b', 'c', 'd']]
```

This is needed because the elements contained in set types do not have a deterministic order. For example, `list({'a', 'b'})` may produce either `['a', 'b']` or `['b', 'a']`.

Odds and Ends

- For locale aware sorting, use `locale.strxfrm()` for a key function or `locale.strcoll()` for a comparison function. This is necessary because “alphabetical” sort orderings can vary across cultures even if the underlying alphabet is the same.

- The *reverse* parameter still maintains sort stability (so that records with equal keys retain the original order). Interestingly, that effect can be simulated without the parameter by using the builtin `reversed()` function twice:

  ```
  >>> data = [('red', 1), ('blue', 1), ('red', 2), ('blue', 2)]
  >>> standard_way = sorted(data, key=itemgetter(0), reverse=True)
  >>> double_reversed = list(reversed(sorted(reversed(data), key=itemgetter(0))))
  >>> assert standard_way == double_reversed
  >>> standard_way
  [('red', 1), ('red', 2), ('blue', 1), ('blue', 2)]
  ```

- The sort routines use `<` when making comparisons between two objects. So, it is easy to add a standard sort order to a class by defining an `__lt__()` method:

```
>>> Student.__lt__ = lambda self, other: self.age <
other.age
>>> sorted(student_objects)
[('dave', 'B', 10), ('jane', 'B', 12), ('john', 'A', 15)]
```

However, note that < can fall back to using `__gt__()` if `__lt__()` is not implemented (see `object.__lt__()` for details on the mechanics). To avoid surprises, **PEP 8** [https://peps.python.org/pep-0008/] recommends that all six comparison methods be implemented. The `total_ordering()` decorator is provided to make that task easier.

- Key functions need not depend directly on the objects being sorted. A key function can also access external resources. For instance, if the student grades are stored in a dictionary, they can be used to sort a separate list of student names:

```
>>> students = ['dave', 'john', 'jane']
>>> newgrades = {'john': 'F', 'jane':'A', 'dave': 'C'}
>>> sorted(students, key=newgrades.__getitem__)
['jane', 'dave', 'john']
```

Partial Sorts

Some applications require only some of the data to be ordered. The standard library provides several tools that do less work than a full sort:

- `min()` and `max()` return the smallest and largest values, respectively. These functions make a single pass over the input data and require almost no auxiliary memory.
- `heapq.nsmallest()` and `heapq.nlargest()` return the *n* smallest and largest values, respectively. These

functions make a single pass over the data keeping only *n* elements in memory at a time. For values of *n* that are small relative to the number of inputs, these functions make far fewer comparisons than a full sort.

- **`heapq.heappush()`** and **`heapq.heappop()`** create and maintain a partially sorted arrangement of data that keeps the smallest element at position `0`. These functions are suitable for implementing priority queues which are commonly used for task scheduling.

Unicode HOWTO

Release:: 1.12

This HOWTO discusses Python's support for the Unicode specification for representing textual data, and explains various problems that people commonly encounter when trying to work with Unicode.

Introduction to Unicode

Definitions

Today's programs need to be able to handle a wide variety of characters. Applications are often internationalized to display messages and output in a variety of user-selectable languages; the same program might need to output an error message in English, French, Japanese, Hebrew, or Russian. Web content can be written in any of these languages and can also include a variety of emoji symbols. Python's string type uses the Unicode Standard for representing characters, which lets Python programs work with all these different possible characters.

Unicode (https://www.unicode.org/) is a specification that aims to list every character used by human languages and give each character its own unique code. The Unicode specifications are continually revised and updated to add new languages and symbols.

A **character** is the smallest possible component of a text. 'A', 'B', 'C', etc., are all different characters. So are 'È' and 'Í'. Characters vary depending on the language or context you’re talking about. For example, there’s a character for “Roman Numeral One”, ‘Ⅰ’, that’s separate from the uppercase letter ‘I’. They’ll usually look the same, but these are two different characters that have different meanings.

The Unicode standard describes how characters are represented by **code points**. A code point value is an integer in the range 0 to 0x10FFFF (about 1.1 million values, the actual number assigned [https://www.unicode.org/versions/latest/#Summary] is less than that). In the standard and in this document, a code point is written using the notation `U+265E` to mean the character with value `0x265e` (9,822 in decimal).

The Unicode standard contains a lot of tables listing characters and their corresponding code points:

```
0061    'a'; LATIN SMALL LETTER A
0062    'b'; LATIN SMALL LETTER B
0063    'c'; LATIN SMALL LETTER C
...
007B    '{'; LEFT CURLY BRACKET
...
2167    'Ⅷ'; ROMAN NUMERAL EIGHT
2168    'Ⅸ'; ROMAN NUMERAL NINE
...
265E    '♞'; BLACK CHESS KNIGHT
265F    '♟'; BLACK CHESS PAWN
...
1F600   '😀'; GRINNING FACE
1F609   '😉'; WINKING FACE
...
```

Strictly, these definitions imply that it’s meaningless to say ‘this is character `U+265E`’. `U+265E` is a code point, which represents some particular character; in this case, it represents the character ‘BLACK CHESS KNIGHT’, ‘♞’. In informal contexts, this distinction between code points and characters will sometimes be forgotten.

A character is represented on a screen or on paper by a set of graphical elements that’s called a **glyph**. The glyph for an uppercase A, for example, is two diagonal strokes and a horizontal stroke, though the exact details will depend on the font being used. Most Python code doesn’t need to worry about glyphs; figuring out the correct glyph to display is generally the job of a GUI toolkit or a terminal’s font renderer.

Encodings

To summarize the previous section: a Unicode string is a sequence of code points, which are numbers from 0 through `0x10FFFF` (1,114,111 decimal). This sequence of code points needs to be represented in memory as a set of **code units**, and **code units** are then mapped to 8-bit bytes. The rules for translating a Unicode string into a sequence of bytes are called a **character encoding**, or just an **encoding**.

The first encoding you might think of is using 32-bit integers as the code unit, and then using the CPU’s representation of 32-bit integers. In this representation, the string “Python” might look like this:

```
   P           y           t           h           o
n
0x50 00 00 00 79 00 00 00 74 00 00 00 68 00 00 00 6f 00 00 00
6e 00 00 00
   0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19
20 21 22 23
```

This representation is straightforward but using it presents a number of problems.

1. It’s not portable; different processors order the bytes differently.

2. It’s very wasteful of space. In most texts, the majority of the code points are less than 127, or less than 255, so a lot of space is occupied by `0x00` bytes. The above string takes 24 bytes compared to the 6 bytes needed for an ASCII representation. Increased RAM usage doesn’t matter too much (desktop computers have gigabytes of RAM, and strings aren’t usually that large), but expanding our usage of disk and network bandwidth by a factor of 4 is intolerable.
3. It’s not compatible with existing C functions such as `strlen()`, so a new family of wide string functions would need to be used.

Therefore this encoding isn’t used very much, and people instead choose other encodings that are more efficient and convenient, such as UTF-8.

UTF-8 is one of the most commonly used encodings, and Python often defaults to using it. UTF stands for “Unicode Transformation Format”, and the ‘8’ means that 8-bit values are used in the encoding. (There are also UTF-16 and UTF-32 encodings, but they are less frequently used than UTF-8.) UTF-8 uses the following rules:

1. If the code point is < 128, it’s represented by the corresponding byte value.
2. If the code point is >= 128, it’s turned into a sequence of two, three, or four bytes, where each byte of the sequence is between 128 and 255.

UTF-8 has several convenient properties:

1. It can handle any Unicode code point.
2. A Unicode string is turned into a sequence of bytes that contains embedded zero bytes only where they represent the null character (U+0000). This means that

UTF-8 strings can be processed by C functions such as `strcpy()` and sent through protocols that can't handle zero bytes for anything other than end-of-string markers.
3. A string of ASCII text is also valid UTF-8 text.
4. UTF-8 is fairly compact; the majority of commonly used characters can be represented with one or two bytes.
5. If bytes are corrupted or lost, it's possible to determine the start of the next UTF-8-encoded code point and resynchronize. It's also unlikely that random 8-bit data will look like valid UTF-8.
6. UTF-8 is a byte oriented encoding. The encoding specifies that each character is represented by a specific sequence of one or more bytes. This avoids the byte-ordering issues that can occur with integer and word oriented encodings, like UTF-16 and UTF-32, where the sequence of bytes varies depending on the hardware on which the string was encoded.

References

The Unicode Consortium site [https://www.unicode.org] has character charts, a glossary, and PDF versions of the Unicode specification. Be prepared for some difficult reading. A chronology [https://www.unicode.org/history/] of the origin and development of Unicode is also available on the site.

On the Computerphile Youtube channel, Tom Scott briefly discusses the history of Unicode and UTF-8 [https://www.youtube.com/watch?v=MijmeoH9LT4] (9 minutes 36 seconds).

To help understand the standard, Jukka Korpela has written an introductory guide [https://jkorpela.fi/unicode/guide.html] to reading the Unicode character tables.

Another good introductory article [https://www.joelonsoftware.com/2003/10/08/the-absolute-minimum-every-software-developer-absolutely-positively-must-know-about-unicode-and-character-sets-no-excuses/] was written by Joel Spolsky. If this introduction didn’t make things clear to you, you should try reading this alternate article before continuing.

Wikipedia entries are often helpful; see the entries for “character encoding [https://en.wikipedia.org/wiki/Character_encoding]” and UTF-8 [https://en.wikipedia.org/wiki/UTF-8], for example.

Python’s Unicode Support

Now that you’ve learned the rudiments of Unicode, we can look at Python’s Unicode features.

The String Type

Since Python 3.0, the language’s **`str`** type contains Unicode characters, meaning any string created using `"unicode rocks!"`, `'unicode rocks!'`, or the triple-quoted string syntax is stored as Unicode.

The default encoding for Python source code is UTF-8, so you can simply include a Unicode character in a string literal:

```
try:
    with open('/tmp/input.txt', 'r') as f:
        ...
except OSError:
    # 'File not found' error message.
    print("Fichier non trouvé")
```

Side note: Python 3 also supports using Unicode characters in identifiers:

```
répertoire = "/tmp/records.log"
with open(répertoire, "w") as f:
    f.write("test\n")
```

If you can't enter a particular character in your editor or want to keep the source code ASCII-only for some reason, you can also use escape sequences in string literals. (Depending on your system, you may see the actual capital-delta glyph instead of a u escape.)

```
>>> "\N{GREEK CAPITAL LETTER DELTA}"  # Using the character
name
'\u0394'
>>> "\u0394"                          # Using a 16-bit hex
value
'\u0394'
>>> "\U00000394"                      # Using a 32-bit hex
value
'\u0394'
```

In addition, one can create a string using the `decode()` method of `bytes`. This method takes an *encoding* argument, such as `UTF-8`, and optionally an *errors* argument.

The *errors* argument specifies the response when the input string can't be converted according to the encoding's rules. Legal values for this argument are `'strict'` (raise a `UnicodeDecodeError` exception), `'replace'` (use `U+FFFD`, `REPLACEMENT CHARACTER`), `'ignore'` (just leave the character out of the Unicode result), or `'backslashreplace'` (inserts a `\xNN` escape sequence). The following examples show the differences:

```
>>> b'\x80abc'.decode("utf-8", "strict")
Traceback (most recent call last):
    ...
UnicodeDecodeError: 'utf-8' codec can't decode byte 0x80 in
position 0:
  invalid start byte
>>> b'\x80abc'.decode("utf-8", "replace")
```

```
'\ufffdabc'
>>> b'\x80abc'.decode("utf-8", "backslashreplace")
'\\x80abc'
>>> b'\x80abc'.decode("utf-8", "ignore")
'abc'
```

Encodings are specified as strings containing the encoding’s name. Python comes with roughly 100 different encodings; see the Python Library Reference at Standard Encodings for a list. Some encodings have multiple names; for example, `'latin-1'`, `'iso_8859_1'` and `'8859`’ are all synonyms for the same encoding.

One-character Unicode strings can also be created with the **`chr()`** built-in function, which takes integers and returns a Unicode string of length 1 that contains the corresponding code point. The reverse operation is the built-in **`ord()`** function that takes a one-character Unicode string and returns the code point value:

```
>>> chr(57344)
'\ue000'
>>> ord('\ue000')
57344
```

Converting to Bytes

The opposite method of **`bytes.decode()`** is **`str.encode()`**, which returns a **`bytes`** representation of the Unicode string, encoded in the requested *encoding*.

The *errors* parameter is the same as the parameter of the **`decode()`** method but supports a few more possible handlers. As well as `'strict'`, `'ignore'`, and `'replace'` (which in this case inserts a question mark instead of the unencodable character), there is also `'xmlcharrefreplace'` (inserts an XML character reference), `backslashreplace` (inserts a `\uNNNN` escape sequence) and `namereplace` (inserts a `\N{...}` escape sequence).

The following example shows the different results:

```
>>> u = chr(40960) + 'abcd' + chr(1972)
>>> u.encode('utf-8')
b'\xea\x80\x80abcd\xde\xb4'
>>> u.encode('ascii')
Traceback (most recent call last):
    ...
UnicodeEncodeError: 'ascii' codec can't encode character
'\ua000' in
  position 0: ordinal not in range(128)
>>> u.encode('ascii', 'ignore')
b'abcd'
>>> u.encode('ascii', 'replace')
b'?abcd?'
>>> u.encode('ascii', 'xmlcharrefreplace')
b'&#40960;abcd&#1972;'
>>> u.encode('ascii', 'backslashreplace')
b'\\ua000abcd\\u07b4'
>>> u.encode('ascii', 'namereplace')
b'\\N{YI SYLLABLE IT}abcd\\u07b4'
```

The low-level routines for registering and accessing the available encodings are found in the `codecs` module. Implementing new encodings also requires understanding the `codecs` module. However, the encoding and decoding functions returned by this module are usually more low-level than is comfortable, and writing new encodings is a specialized task, so the module won’t be covered in this HOWTO.

Unicode Literals in Python Source Code

In Python source code, specific Unicode code points can be written using the `\u` escape sequence, which is followed by four hex digits giving the code point. The `\U` escape sequence is similar, but expects eight hex digits, not four:

```
>>> s = "a\xac\u1234\u20ac\U00008000"
... #     ^^^^ two-digit hex escape
... #         ^^^^^^ four-digit Unicode escape
... #                     ^^^^^^^^^^ eight-digit Unicode escape
>>> [ord(c) for c in s]
[97, 172, 4660, 8364, 32768]
```

Using escape sequences for code points greater than 127 is fine in small doses, but becomes an annoyance if you're using many accented characters, as you would in a program with messages in French or some other accent-using language. You can also assemble strings using the `chr()` built-in function, but this is even more tedious.

Ideally, you'd want to be able to write literals in your language's natural encoding. You could then edit Python source code with your favorite editor which would display the accented characters naturally, and have the right characters used at runtime.

Python supports writing source code in UTF-8 by default, but you can use almost any encoding if you declare the encoding being used. This is done by including a special comment as either the first or second line of the source file:

```
#!/usr/bin/env python
# -*- coding: latin-1 -*-

u = 'abcdé'
print(ord(u[-1]))
```

The syntax is inspired by Emacs's notation for specifying variables local to a file. Emacs supports many different variables, but Python only supports 'coding'. The `-*-` symbols indicate to Emacs that the comment is special;

they have no significance to Python but are a convention. Python looks for `coding: name` or `coding=name` in the comment.

If you don’t include such a comment, the default encoding used will be UTF-8 as already mentioned. See also **PEP 263** [https://peps.python.org/pep-0263/] for more information.

Unicode Properties

The Unicode specification includes a database of information about code points. For each defined code point, the information includes the character’s name, its category, the numeric value if applicable (for characters representing numeric concepts such as the Roman numerals, fractions such as one-third and four-fifths, etc.). There are also display-related properties, such as how to use the code point in bidirectional text.

The following program displays some information about several characters, and prints the numeric value of one particular character:

```
import unicodedata

u = chr(233) + chr(0x0bf2) + chr(3972) + chr(6000) + chr(13231)

for i, c in enumerate(u):
    print(i, '%04x' % ord(c), unicodedata.category(c), end=" ")
    print(unicodedata.name(c))

# Get numeric value of second character
print(unicodedata.numeric(u[1]))
```

When run, this prints:

```
0 00e9 Ll LATIN SMALL LETTER E WITH ACUTE
1 0bf2 No TAMIL NUMBER ONE THOUSAND
2 0f84 Mn TIBETAN MARK HALANTA
3 1770 Lo TAGBANWA LETTER SA
```

```
4 33af So SQUARE RAD OVER S SQUARED
1000.0
```

The category codes are abbreviations describing the nature of the character. These are grouped into categories such as “Letter”, “Number”, “Punctuation”, or “Symbol”, which in turn are broken up into subcategories. To take the codes from the above output, `'Ll'` means ‘Letter, lowercase’, `'No'` means “Number, other”, `'Mn'` is “Mark, nonspacing”, and `'So'` is “Symbol, other”. See the General Category Values section of the Unicode Character Database documentation [https://www.unicode.org/reports/tr44/#General_Category_Values] for a list of category codes.

Comparing Strings

Unicode adds some complication to comparing strings, because the same set of characters can be represented by different sequences of code points. For example, a letter like ‘ê’ can be represented as a single code point U+00EA, or as U+0065 U+0302, which is the code point for ‘e’ followed by a code point for ‘COMBINING CIRCUMFLEX ACCENT’. These will produce the same output when printed, but one is a string of length 1 and the other is of length 2.

One tool for a case-insensitive comparison is the **`casefold()`** string method that converts a string to a case-insensitive form following an algorithm described by the Unicode Standard. This algorithm has special handling for characters such as the German letter ‘ß’ (code point U+00DF), which becomes the pair of lowercase letters ‘ss’.

```
>>> street = 'Gürzenichstraße'
>>> street.casefold()
'gürzenichstrasse'
```

A second tool is the `unicodedata` module's `normalize()` function that converts strings to one of several normal forms, where letters followed by a combining character are replaced with single characters. `normalize()` can be used to perform string comparisons that won't falsely report inequality if two strings use combining characters differently:

```
import unicodedata

def compare_strs(s1, s2):
    def NFD(s):
        return unicodedata.normalize('NFD', s)

    return NFD(s1) == NFD(s2)

single_char = 'ê'
multiple_chars = '\N{LATIN SMALL LETTER E}\N{COMBINING
CIRCUMFLEX ACCENT}'
print('length of first string=', len(single_char))
print('length of second string=', len(multiple_chars))
print(compare_strs(single_char, multiple_chars))
```

When run, this outputs:

```
$ python compare-strs.py
length of first string= 1
length of second string= 2
True
```

The first argument to the `normalize()` function is a string giving the desired normalization form, which can be one of 'NFC', 'NFKC', 'NFD', and 'NFKD'.

The Unicode Standard also specifies how to do caseless comparisons:

```
import unicodedata

def compare_caseless(s1, s2):
    def NFD(s):
        return unicodedata.normalize('NFD', s)

    return NFD(NFD(s1).casefold()) == NFD(NFD(s2).casefold())

# Example usage
single_char = 'ê'
multiple_chars = '\N{LATIN CAPITAL LETTER E}\N{COMBINING
CIRCUMFLEX ACCENT}'

print(compare_caseless(single_char, multiple_chars))
```

This will print `True`. (Why is **`NFD()`** invoked twice? Because there are a few characters that make **`casefold()`** return a non-normalized string, so the result needs to be normalized again. See section 3.13 of the Unicode Standard for a discussion and an example.)

Unicode Regular Expressions

The regular expressions supported by the **`re`** module can be provided either as bytes or strings. Some of the special character sequences such as `\d` and `\w` have different meanings depending on whether the pattern is supplied as bytes or a string. For example, `\d` will match the characters `[0-9]` in bytes but in strings will match any character that’s in the `'Nd'` category.

The string in this example has the number 57 written in both Thai and Arabic numerals:

```
import re
p = re.compile(r'\d+')

s = "Over \u0e55\u0e57 57 flavours"
m = p.search(s)
print(repr(m.group()))
```

When executed, `\d+` will match the Thai numerals and print them out. If you supply the **`re.ASCII`** flag to **`compile()`**, `\d+` will match the substring “57” instead.

Similarly, `\w` matches a wide variety of Unicode characters but only `[a-zA-Z0-9_]` in bytes or if **`re.ASCII`** is supplied, and `\s` will match either Unicode whitespace characters or `[ \t\n\r\f\v]`.

References

Some good alternative discussions of Python’s Unicode support are:

- Processing Text Files in Python 3 [https://python-notes.curiousefficiency.org/en/latest/python3/text_file_processing.html], by Nick Coghlan.
- Pragmatic Unicode [https://nedbatchelder.com/text/unipain.html], a PyCon 2012 presentation by Ned Batchelder.

The **`str`** type is described in the Python library reference at Text Sequence Type — str.

The documentation for the **`unicodedata`** module.

The documentation for the **`codecs`** module.

Marc-André Lemburg gave a presentation titled “Python and Unicode” (PDF slides) [https://downloads.egenix.com/python/Unicode-EPC2002-Talk.pdf] at EuroPython 2002. The slides are an excellent overview of the design of Python 2’s Unicode features (where the Unicode string type is called `unicode` and literals start with `u`).

Reading and Writing Unicode Data

Once you’ve written some code that works with Unicode data, the next problem is input/output. How do you get

Unicode strings into your program, and how do you convert Unicode into a form suitable for storage or transmission?

It’s possible that you may not need to do anything depending on your input sources and output destinations; you should check whether the libraries used in your application support Unicode natively. XML parsers often return Unicode data, for example. Many relational databases also support Unicode-valued columns and can return Unicode values from an SQL query.

Unicode data is usually converted to a particular encoding before it gets written to disk or sent over a socket. It’s possible to do all the work yourself: open a file, read an 8-bit bytes object from it, and convert the bytes with `bytes.decode(encoding)`. However, the manual approach is not recommended.

One problem is the multi-byte nature of encodings; one Unicode character can be represented by several bytes. If you want to read the file in arbitrary-sized chunks (say, 1024 or 4096 bytes), you need to write error-handling code to catch the case where only part of the bytes encoding a single Unicode character are read at the end of a chunk. One solution would be to read the entire file into memory and then perform the decoding, but that prevents you from working with files that are extremely large; if you need to read a 2 GiB file, you need 2 GiB of RAM. (More, really, since for at least a moment you’d need to have both the encoded string and its Unicode version in memory.)

The solution would be to use the low-level decoding interface to catch the case of partial coding sequences. The work of implementing this has already been done for you: the built-in `open()` function can return a file-like object that assumes the file’s contents are in a specified encoding and accepts Unicode parameters for methods such as `read()` and `write()`. This works through `open()`'s *encoding* and *errors* parameters which are interpreted just like those in `str.encode()` and `bytes.decode()`.

Reading Unicode from a file is therefore simple:

```
with open('unicode.txt', encoding='utf-8') as f:
    for line in f:
        print(repr(line))
```

It’s also possible to open files in update mode, allowing both reading and writing:

```
with open('test', encoding='utf-8', mode='w+') as f:
    f.write('\u4500 blah blah blah\n')
    f.seek(0)
    print(repr(f.readline()[:1]))
```

The Unicode character `U+FEFF` is used as a byte-order mark (BOM), and is often written as the first character of a file in order to assist with autodetection of the file’s byte ordering. Some encodings, such as UTF-16, expect a BOM to be present at the start of a file; when such an encoding is used, the BOM will be automatically written as the first character and will be silently dropped when the file is read. There are variants of these encodings, such as ‘utf-16-le’ and ‘utf-16-be’ for little-endian and big-endian encodings, that specify one particular byte ordering and don’t skip the BOM.

In some areas, it is also convention to use a “BOM” at the start of UTF-8 encoded files; the name is misleading since UTF-8 is not byte-order dependent. The mark simply announces that the file is encoded in UTF-8. For reading such files, use the ‘utf-8-sig’ codec to automatically skip the mark if present.

Unicode filenames

Most of the operating systems in common use today support filenames that contain arbitrary Unicode characters. Usually this is implemented by converting the Unicode string into some encoding that varies depending on the system. Today Python is converging on using UTF-8: Python on MacOS has used UTF-8 for several versions, and Python 3.6 switched to using UTF-8 on Windows as well. On Unix systems, there will only be a filesystem encoding. if you've set the `LANG` or `LC_CTYPE` environment variables; if you haven't, the default encoding is again UTF-8.

The **`sys.getfilesystemencoding()`** function returns the encoding to use on your current system, in case you want to do the encoding manually, but there's not much reason to bother. When opening a file for reading or writing, you can usually just provide the Unicode string as the filename, and it will be automatically converted to the right encoding for you:

```
filename = 'filename\u4500abc'
with open(filename, 'w') as f:
    f.write('blah\n')
```

Functions in the **`os`** module such as **`os.stat()`** will also accept Unicode filenames.

The **`os.listdir()`** function returns filenames, which raises an issue: should it return the Unicode version of filenames, or should it return bytes containing the encoded versions? **`os.listdir()`** can do both, depending on whether you provided the directory path as bytes or a Unicode string. If you pass a Unicode string as the path, filenames will be decoded using the filesystem's encoding and a list of Unicode strings will be returned, while passing a byte path will return the filenames as bytes. For example, assuming the default filesystem encoding is UTF-8, running the following program:

```
fn = 'filename\u4500abc'
f = open(fn, 'w')
f.close()

import os
print(os.listdir(b'.'))
print(os.listdir('.'))
```

will produce the following output:

```
$ python listdir-test.py
[b'filename\xe4\x94\x80abc', ...]
['filename\u4500abc', ...]
```

The first list contains UTF-8-encoded filenames, and the second list contains the Unicode versions.

Note that on most occasions, you should can just stick with using Unicode with these APIs. The bytes APIs should only be used on systems where undecodable file names can be present; that’s pretty much only Unix systems now.

Tips for Writing Unicode-aware Programs

This section provides some suggestions on writing software that deals with Unicode.

The most important tip is:

> Software should only work with Unicode strings internally, decoding the input data as soon as possible and encoding the output only at the end.

If you attempt to write processing functions that accept both Unicode and byte strings, you will find your program vulnerable to bugs wherever you combine the two different kinds of strings. There is no automatic encoding or decoding: if you do e.g. `str + bytes`, a **`TypeError`** will be raised.

When using data coming from a web browser or some other untrusted source, a common technique is to check for illegal characters in a string before using the string in a generated command line or storing it in a database. If you’re doing this, be careful to check the decoded string, not the encoded bytes data; some encodings may have interesting properties, such as not being bijective or not being fully ASCII-compatible. This is especially true if the input data also specifies the encoding, since the attacker can then choose a clever way to hide malicious text in the encoded bytestream.

Converting Between File Encodings

The **`StreamRecoder`** class can transparently convert between encodings, taking a stream that returns data in encoding #1 and behaving like a stream returning data in encoding #2.

For example, if you have an input file *f* that’s in Latin-1, you can wrap it with a **`StreamRecoder`** to return bytes encoded in UTF-8:

```
new_f = codecs.StreamRecoder(f,
    # en/decoder: used by read() to encode its results and
    # by write() to decode its input.
    codecs.getencoder('utf-8'), codecs.getdecoder('utf-8'),

    # reader/writer: used to read and write to the stream.
    codecs.getreader('latin-1'), codecs.getwriter('latin-1') )
```

Files in an Unknown Encoding

What can you do if you need to make a change to a file, but don’t know the file’s encoding? If you know the encoding is

ASCII-compatible and only want to examine or modify the ASCII parts, you can open the file with the `surrogateescape` error handler:

```
with open(fname, 'r', encoding="ascii",
errors="surrogateescape") as f:
    data = f.read()

# make changes to the string 'data'

with open(fname + '.new', 'w',
          encoding="ascii", errors="surrogateescape") as f:
    f.write(data)
```

The `surrogateescape` error handler will decode any non-ASCII bytes as code points in a special range running from U+DC80 to U+DCFF. These code points will then turn back into the same bytes when the `surrogateescape` error handler is used to encode the data and write it back out.

References

One section of Mastering Python 3 Input/Output [https://pyvideo.org/video/289/pycon-2010--mastering-python-3-i-o], a PyCon 2010 talk by David Beazley, discusses text processing and binary data handling.

The PDF slides for Marc-André Lemburg’s presentation “Writing Unicode-aware Applications in Python” [https://downloads.egenix.com/python/LSM2005-Developing-Unicode-aware-applications-in-Python.pdf] discuss questions of character encodings as well as how to internationalize and localize an application. These slides cover Python 2.x only.

The Guts of Unicode in Python [https://pyvideo.org/video/1768/the-guts-of-unicode-in-python] is a PyCon 2013 talk by Benjamin Peterson that discusses the internal Unicode representation in Python 3.3.

Acknowledgements

The initial draft of this document was written by Andrew Kuchling. It has since been revised further by Alexander Belopolsky, Georg Brandl, Andrew Kuchling, and Ezio Melotti.

Thanks to the following people who have noted errors or offered suggestions on this article: Éric Araujo, Nicholas Bastin, Nick Coghlan, Marius Gedminas, Kent Johnson, Ken Krugler, Marc-André Lemburg, Martin von Löwis, Terry J. Reedy, Serhiy Storchaka, Eryk Sun, Chad Whitacre, Graham Wideman.

HOWTO Fetch Internet Resources Using The urllib Package

Author:: Michael Foord [https://agileabstractions.com/]

Introduction

Related Articles

You may also find useful the following article on fetching web resources with Python:

- Basic Authentication [https://web.archive.org/web/20201215133350/http://www.voidspace.org.uk/python/articles/authentication.shtml]

  A tutorial on *Basic Authentication*, with examples in Python.

urllib.request is a Python module for fetching URLs (Uniform Resource Locators). It offers a very simple interface, in the form of the *urlopen* function. This is capable of fetching URLs using a variety of different protocols. It also offers a slightly more complex interface for handling common situations - like basic authentication, cookies, proxies and so on. These are provided by objects called handlers and openers.

urllib.request supports fetching URLs for many “URL schemes” (identified by the string before the `":"` in URL - for example `"ftp"` is the URL scheme of `"ftp://python.org/"`) using their associated network protocols (e.g. FTP, HTTP). This tutorial focuses on the most common case, HTTP.

For straightforward situations *urlopen* is very easy to use. But as soon as you encounter errors or non-trivial cases when opening HTTP URLs, you will need some understanding of the HyperText Transfer Protocol. The most comprehensive and authoritative reference to HTTP is **RFC 2616** [https://datatracker.ietf.org/doc/html/rfc2616.html]. This is a technical document and not intended to be easy to read. This HOWTO aims to illustrate using *urllib*, with enough detail about HTTP to help you through. It is not intended to replace the **`urllib.request`** docs, but is supplementary to them.

Fetching URLs

The simplest way to use urllib.request is as follows:

```
import urllib.request
with urllib.request.urlopen('http://python.org/') as response:
   html = response.read()
```

If you wish to retrieve a resource via URL and store it in a temporary location, you can do so via the **`shutil.copyfileobj()`** and **`tempfile.NamedTemporaryFile()`** functions:

```
import shutil
import tempfile
import urllib.request

with urllib.request.urlopen('http://python.org/') as response:
    with tempfile.NamedTemporaryFile(delete=False) as tmp_file:
        shutil.copyfileobj(response, tmp_file)

with open(tmp_file.name) as html:
    pass
```

Many uses of urllib will be that simple (note that instead of an ‘http:’ URL we could have used a URL starting with ‘ftp:’, ‘file:’, etc.). However, it’s the purpose of this tutorial to explain the more complicated cases, concentrating on HTTP.

HTTP is based on requests and responses - the client makes requests and servers send responses. urllib.request mirrors this with a `Request` object which represents the HTTP request you are making. In its simplest form you create a Request object that specifies the URL you want to fetch. Calling `urlopen` with this Request object returns a response object for the URL requested. This response is a file-like object, which means you can for example call `.read()` on the response:

```
import urllib.request

req = urllib.request.Request('http://python.org/')
with urllib.request.urlopen(req) as response:
   the_page = response.read()
```

Note that urllib.request makes use of the same Request interface to handle all URL schemes. For example, you can make an FTP request like so:

```
req = urllib.request.Request('ftp://example.com/')
```

In the case of HTTP, there are two extra things that Request objects allow you to do: First, you can pass data to be sent to the server. Second, you can pass extra information (“metadata”) *about* the data or about the request itself, to the server - this information is sent as HTTP “headers”. Let’s look at each of these in turn.

Data

Sometimes you want to send data to a URL (often the URL will refer to a CGI (Common Gateway Interface) script or other web application). With HTTP, this is often done using what's known as a **POST** request. This is often what your browser does when you submit a HTML form that you filled in on the web. Not all POSTs have to come from forms: you can use a POST to transmit arbitrary data to your own application. In the common case of HTML forms, the data needs to be encoded in a standard way, and then passed to the Request object as the `data` argument. The encoding is done using a function from the **`urllib.parse`** library.

```
import urllib.parse
import urllib.request

url = 'http://www.someserver.com/cgi-bin/register.cgi'
values = {'name' : 'Michael Foord',
          'location' : 'Northampton',
          'language' : 'Python' }

data = urllib.parse.urlencode(values)
data = data.encode('ascii') # data should be bytes
req = urllib.request.Request(url, data)
with urllib.request.urlopen(req) as response:
   the_page = response.read()
```

Note that other encodings are sometimes required (e.g. for file upload from HTML forms - see HTML Specification, Form Submission [https://www.w3.org/TR/REC-html40/interact/forms.html#h-17.13] for more details).

If you do not pass the `data` argument, urllib uses a **GET** request. One way in which GET and POST requests differ is that POST requests often have "side-effects": they change the state of the system in some way (for example by placing an order with the website for a hundredweight of tinned spam to be delivered to your door). Though the HTTP standard makes it clear that POSTs are intended to *always* cause side-effects, and GET requests *never* to cause side-effects, nothing prevents a GET request from having side-effects, nor a POST requests from having no side-effects. Data can also be passed in an HTTP GET request by encoding it in the URL itself.

This is done as follows:

```
>>> import urllib.request
>>> import urllib.parse
>>> data = {}
>>> data['name'] = 'Somebody Here'
>>> data['location'] = 'Northampton'
>>> data['language'] = 'Python'
>>> url_values = urllib.parse.urlencode(data)
>>> print(url_values)  # The order may differ from below.
name=Somebody+Here&language=Python&location=Northampton
>>> url = 'http://www.example.com/example.cgi'
>>> full_url = url + '?' + url_values
>>> data = urllib.request.urlopen(full_url)
```

Notice that the full URL is created by adding a `?` to the URL, followed by the encoded values.

Headers

We’ll discuss here one particular HTTP header, to illustrate how to add headers to your HTTP request.

Some websites [[1]] dislike being browsed by programs, or send different versions to different browsers [[2]]. By default urllib identifies itself as `Python-urllib/x.y` (where `x` and `y` are the major and minor version numbers of the Python release, e.g. `Python-urllib/2.5`), which may confuse the site, or just plain not work. The way a browser identifies itself is through the `User-Agent` header [[3]]. When you create a Request object you can pass a dictionary of headers in. The following example makes the same request as above, but identifies itself as a version of Internet Explorer [[4]].

```
import urllib.parse
import urllib.request

url = 'http://www.someserver.com/cgi-bin/register.cgi'
user_agent = 'Mozilla/5.0 (Windows NT 6.1; Win64; x64)'
values = {'name': 'Michael Foord',
          'location': 'Northampton',
          'language': 'Python' }
headers = {'User-Agent': user_agent}

data = urllib.parse.urlencode(values)
data = data.encode('ascii')
req = urllib.request.Request(url, data, headers)
with urllib.request.urlopen(req) as response:
   the_page = response.read()
```

The response also has two useful methods. See the section on info and geturl which comes after we have a look at what happens when things go wrong.

Handling Exceptions

urlopen raises `URLError` when it cannot handle a response (though as usual with Python APIs, built-in exceptions such as `ValueError`, `TypeError` etc. may also be raised).

`HTTPError` is the subclass of `URLError` raised in the specific case of HTTP URLs.

The exception classes are exported from the `urllib.error` module.

URLError

Often, URLError is raised because there is no network connection (no route to the specified server), or the specified server doesn’t exist. In this case, the exception raised will have a ‘reason’ attribute, which is a tuple containing an error code and a text error message.

e.g.

```
>>> req =
urllib.request.Request('http://www.pretend_server.org')
>>> try: urllib.request.urlopen(req)
... except urllib.error.URLError as e:
...     print(e.reason)
...
(4, 'getaddrinfo failed')
```

HTTPError

Every HTTP response from the server contains a numeric “status code”. Sometimes the status code indicates that the server is unable to fulfil the request. The default handlers will handle some of these responses for you (for example, if the response is a “redirection” that requests the client fetch the document from a different URL, urllib will handle that for you). For those it can’t handle, urlopen will raise an `HTTPError`. Typical errors include ‘404’ (page not found), ‘403’ (request forbidden), and ‘401’ (authentication required).

See section 10 of **RFC 2616** [https://datatracker.ietf.org/doc/html/rfc2616.html] for a reference on all the HTTP error codes.

The `HTTPError` instance raised will have an integer ‘code’ attribute, which corresponds to the error sent by the server.

Error Codes

Because the default handlers handle redirects (codes in the 300 range), and codes in the 100–299 range indicate success, you will usually only see error codes in the 400–599 range.

`http.server.BaseHTTPRequestHandler.responses` is a useful dictionary of response codes that shows all the response codes used by **RFC 2616** [https://datatracker.ietf.org/doc/html/rfc2616.html]. An excerpt from the dictionary is shown below

```
responses = {
    ...
    <HTTPStatus.OK: 200>: ('OK', 'Request fulfilled, document
follows'),
    ...
    <HTTPStatus.FORBIDDEN: 403>: ('Forbidden',
                                  'Request forbidden --
authorization will '
                                  'not help'),
    <HTTPStatus.NOT_FOUND: 404>: ('Not Found',
                                  'Nothing matches the given
URI'),
    ...
    <HTTPStatus.IM_A_TEAPOT: 418>: ("I'm a Teapot",
                                    'Server refuses to brew
coffee because '
                                    'it is a teapot'),
    ...
    <HTTPStatus.SERVICE_UNAVAILABLE: 503>: ('Service
Unavailable',
                                            'The server cannot
process the '
                                            'request due to a
high load'),
    ...
    }
```

When an error is raised the server responds by returning an HTTP error code *and* an error page. You can use the `HTTPError` instance as a response on the page returned. This means that as well as the code attribute, it also has read, geturl, and info, methods as returned by the `urllib.response` module:

```
>>> req =
urllib.request.Request('http://www.python.org/fish.html')
>>> try:
...     urllib.request.urlopen(req)
... except urllib.error.HTTPError as e:
...     print(e.code)
...     print(e.read())
...
404
b'<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0
Transitional//EN"
  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-
transitional.dtd">\n\n\n<html
  ...
  <title>Page Not Found</title>\n
  ...
```

Wrapping it Up

So if you want to be prepared for **`HTTPError`** *or* **`URLError`** there are two basic approaches. I prefer the second approach.

Number 1

```
from urllib.request import Request, urlopen
from urllib.error import URLError, HTTPError
req = Request(someurl)
try:
    response = urlopen(req)
except HTTPError as e:
    print('The server couldn\'t fulfill the request.')
    print('Error code: ', e.code)
except URLError as e:
    print('We failed to reach a server.')
    print('Reason: ', e.reason)
else:
    # everything is fine
```

Note

The `except HTTPError` *must* come first, otherwise `except URLError` will *also* catch an `HTTPError`.

Number 2

```
from urllib.request import Request, urlopen
from urllib.error import URLError
req = Request(someurl)
try:
    response = urlopen(req)
except URLError as e:
    if hasattr(e, 'reason'):
        print('We failed to reach a server.')
        print('Reason: ', e.reason)
    elif hasattr(e, 'code'):
        print('The server couldn\'t fulfill the request.')
        print('Error code: ', e.code)
else:
    # everything is fine
```

info and geturl

The response returned by urlopen (or the `HTTPError` instance) has two useful methods **`info()`** and **`geturl()`** and is defined in the module `urllib.response`.

- **geturl** - this returns the real URL of the page fetched. This is useful because `urlopen` (or the opener object used) may have followed a redirect. The URL of the page fetched may not be the same as the URL requested.
- **info** - this returns a dictionary-like object that describes the page fetched, particularly the headers sent by the server. It is currently an `http.client.HTTPMessage` instance.

Typical headers include ‘Content-length’, ‘Content-type’, and so on. See the Quick Reference to HTTP Headers [https://jkorpela.fi/http.html] for a useful listing of HTTP headers with brief explanations of their meaning and use.

Openers and Handlers

When you fetch a URL you use an opener (an instance of the perhaps confusingly named **`urllib.request.OpenerDirector`**). Normally we have been using the default opener - via `urlopen` - but you can create custom openers. Openers use handlers. All the “heavy lifting” is done by the handlers. Each handler knows how to open URLs for a particular URL scheme (http, ftp, etc.), or how to handle an aspect of URL opening, for example HTTP redirections or HTTP cookies.

You will want to create openers if you want to fetch URLs with specific handlers installed, for example to get an opener that handles cookies, or to get an opener that does not handle redirections.

To create an opener, instantiate an `OpenerDirector`, and then call `.add_handler(some_handler_instance)` repeatedly.

Alternatively, you can use `build_opener`, which is a convenience function for creating opener objects with a single function call. `build_opener` adds several handlers by default, but provides a quick way to add more and/or override the default handlers.

Other sorts of handlers you might want to can handle proxies, authentication, and other common but slightly specialised situations.

`install_opener` can be used to make an `opener` object the (global) default opener. This means that calls to `urlopen` will use the opener you have installed.

Opener objects have an `open` method, which can be called directly to fetch urls in the same way as the `urlopen` function: there’s no need to call `install_opener`, except as a convenience.

Basic Authentication

To illustrate creating and installing a handler we will use the `HTTPBasicAuthHandler`. For a more detailed discussion of this subject – including an explanation of how Basic Authentication works - see the Basic Authentication Tutorial [https://web.archive.org/web/20201215133350/http://www.voidspace.org.uk/python/articles/authentication.shtml].

When authentication is required, the server sends a header (as well as the 401 error code) requesting authentication. This specifies the authentication scheme and a ‘realm’. The header looks like: `WWW-Authenticate: SCHEME realm="REALM"`.

e.g.

```
WWW-Authenticate: Basic realm="cPanel Users"
```

The client should then retry the request with the appropriate name and password for the realm included as a header in the request. This is ‘basic authentication’. In order to simplify this process we can create an instance of `HTTPBasicAuthHandler` and an opener to use this handler.

The `HTTPBasicAuthHandler` uses an object called a password manager to handle the mapping of URLs and realms to passwords and usernames. If you know what the realm is (from the authentication header sent by the server), then you can use a `HTTPPasswordMgr`. Frequently one doesn’t care what the realm is. In that case, it is convenient to use `HTTPPasswordMgrWithDefaultRealm`. This allows you to specify a default username and password for a URL. This will be supplied in the absence of you providing an alternative combination for a specific realm. We indicate this by providing `None` as the realm argument to the `add_password` method.

The top-level URL is the first URL that requires authentication. URLs “deeper” than the URL you pass to .add_password() will also match.

```
# create a password manager
password_mgr = urllib.request.HTTPPasswordMgrWithDefaultRealm()

# Add the username and password.
# If we knew the realm, we could use it instead of None.
top_level_url = "http://example.com/foo/"
password_mgr.add_password(None, top_level_url, username,
password)

handler = urllib.request.HTTPBasicAuthHandler(password_mgr)

# create "opener" (OpenerDirector instance)
opener = urllib.request.build_opener(handler)

# use the opener to fetch a URL
opener.open(a_url)

# Install the opener.
# Now all calls to urllib.request.urlopen use our opener.
urllib.request.install_opener(opener)
```

Note

In the above example we only supplied our `HTTPBasicAuthHandler` to `build_opener`. By default openers have the handlers for normal situations – `ProxyHandler` (if a proxy setting such as an **`http_proxy`** environment variable is set), `UnknownHandler`, `HTTPHandler`, `HTTPDefaultErrorHandler`, `HTTPRedirectHandler`, `FTPHandler`, `FileHandler`, `DataHandler`, `HTTPErrorProcessor`.

`top_level_url` is in fact *either* a full URL (including the 'http:' scheme component and the hostname and optionally the port number) e.g. `"http://example.com/"` *or* an "authority" (i.e. the hostname, optionally including the port number) e.g. `"example.com"` or `"example.com:8080"` (the latter example includes a port number). The authority, if present, must NOT contain the "userinfo" component - for example `"joe:password@example.com"` is not correct.

Proxies

urllib will auto-detect your proxy settings and use those. This is through the `ProxyHandler`, which is part of the normal handler chain when a proxy setting is detected. Normally that's a good thing, but there are occasions when it may not be helpful [[5]]. One way to do this is to setup our own `ProxyHandler`, with no proxies defined. This is done using similar steps to setting up a Basic Authentication handler:

```
>>> proxy_support = urllib.request.ProxyHandler({})
>>> opener = urllib.request.build_opener(proxy_support)
>>> urllib.request.install_opener(opener)
```

Note

Currently `urllib.request` *does not* support fetching of `https` locations through a proxy. However, this can be enabled by extending urllib.request as shown in the recipe [[6]].

Note

`HTTP_PROXY` will be ignored if a variable `REQUEST_METHOD` is set; see the documentation on `getproxies()`.

Sockets and Layers

The Python support for fetching resources from the web is layered. urllib uses the `http.client` library, which in turn uses the socket library.

As of Python 2.3 you can specify how long a socket should wait for a response before timing out. This can be useful in applications which have to fetch web pages. By default the socket module has *no timeout* and can hang. Currently, the socket timeout is not exposed at the http.client or urllib.request levels. However, you can set the default timeout globally for all sockets using

```
import socket
import urllib.request

# timeout in seconds
timeout = 10
socket.setdefaulttimeout(timeout)

# this call to urllib.request.urlopen now uses the default
timeout
# we have set in the socket module
req = urllib.request.Request('http://www.voidspace.org.uk')
response = urllib.request.urlopen(req)
```

Footnotes

This document was reviewed and revised by John Lee.

[1]

Google for example.
[2]

Browser sniffing is a very bad practice for website design - building sites using web standards is much more sensible. Unfortunately a lot of sites still send different versions to different browsers.

[3]

The user agent for MSIE 6 is *‘Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.1; SV1; .NET CLR 1.1.4322)’*

[4]

For details of more HTTP request headers, see Quick Reference to HTTP Headers [https://jkorpela.fi/http.html].

[5]

In my case I have to use a proxy to access the internet at work. If you attempt to fetch *localhost* URLs through this proxy it blocks them. IE is set to use the proxy, which urllib picks up on. In order to test scripts with a localhost server, I have to prevent urllib from using the proxy.

[6]

urllib opener for SSL proxy (CONNECT method): ASPN Cookbook Recipe [https://code.activestate.com/recipes/456195-urrlib2-opener-for-ssl-proxy-connect-method/].

An introduction to the ipaddress module

author:: Peter Moody
author:: Nick Coghlan

Overview

This document aims to provide a gentle introduction to the `ipaddress` module. It is aimed primarily at users that aren't already familiar with IP networking terminology, but may also be useful to network engineers wanting an overview of how `ipaddress` represents IP network addressing concepts.

Creating Address/Network/Interface objects

Since `ipaddress` is a module for inspecting and manipulating IP addresses, the first thing you'll want to do is create some objects. You can use `ipaddress` to create objects from strings and integers.

A Note on IP Versions

For readers that aren't particularly familiar with IP addressing, it's important to know that the Internet Protocol (IP) is currently in the process of moving from version 4 of the protocol to version 6. This transition is occurring largely because version 4 of the protocol doesn't provide enough addresses to handle the needs of the whole world, especially given the increasing number of devices with direct connections to the internet.

Explaining the details of the differences between the two versions of the protocol is beyond the scope of this introduction, but readers need to at least be aware that these two versions exist, and it will sometimes be necessary to force the use of one version or the other.

IP Host Addresses

Addresses, often referred to as “host addresses” are the most basic unit when working with IP addressing. The simplest way to create addresses is to use the `ipaddress.ip_address()` factory function, which automatically determines whether to create an IPv4 or IPv6 address based on the passed in value:

```
>>> ipaddress.ip_address('192.0.2.1')
IPv4Address('192.0.2.1')
>>> ipaddress.ip_address('2001:DB8::1')
IPv6Address('2001:db8::1')
```

Addresses can also be created directly from integers. Values that will fit within 32 bits are assumed to be IPv4 addresses:

```
>>> ipaddress.ip_address(3221225985)
IPv4Address('192.0.2.1')
>>>
ipaddress.ip_address(42540766411282592856903984951653826561)
IPv6Address('2001:db8::1')
```

To force the use of IPv4 or IPv6 addresses, the relevant classes can be invoked directly. This is particularly useful to force creation of IPv6 addresses for small integers:

```
>>> ipaddress.ip_address(1)
IPv4Address('0.0.0.1')
>>> ipaddress.IPv4Address(1)
IPv4Address('0.0.0.1')
>>> ipaddress.IPv6Address(1)
IPv6Address('::1')
```

Defining Networks

Host addresses are usually grouped together into IP networks, so **`ipaddress`** provides a way to create, inspect and manipulate network definitions. IP network objects are constructed from strings that define the range of host addresses that are part of that network. The simplest form for that information is a “network address/network prefix” pair, where the prefix defines the number of leading bits that are compared to determine whether or not an address is part of the network and the network address defines the expected value of those bits.

As for addresses, a factory function is provided that determines the correct IP version automatically:

```
>>> ipaddress.ip_network('192.0.2.0/24')
IPv4Network('192.0.2.0/24')
>>> ipaddress.ip_network('2001:db8::0/96')
IPv6Network('2001:db8::/96')
```

Network objects cannot have any host bits set. The practical effect of this is that `192.0.2.1/24` does not describe a network. Such definitions are referred to as interface objects since the ip-on-a-network notation is commonly used to describe network interfaces of a computer on a given network and are described further in the next section.

By default, attempting to create a network object with host bits set will result in **`ValueError`** being raised. To request that the additional bits instead be coerced to zero, the flag `strict=False` can be passed to the constructor:

```
>>> ipaddress.ip_network('192.0.2.1/24')
Traceback (most recent call last):
   ...
ValueError: 192.0.2.1/24 has host bits set
>>> ipaddress.ip_network('192.0.2.1/24', strict=False)
IPv4Network('192.0.2.0/24')
```

While the string form offers significantly more flexibility, networks can also be defined with integers, just like host addresses. In this case, the network is considered to contain only the single address identified by the integer, so the network prefix includes the entire network address:

```
>>> ipaddress.ip_network(3221225984)
IPv4Network('192.0.2.0/32')
>>>
ipaddress.ip_network(42540766411282592856903984951653826560)
IPv6Network('2001:db8::/128')
```

As with addresses, creation of a particular kind of network can be forced by calling the class constructor directly instead of using the factory function.

Host Interfaces

As mentioned just above, if you need to describe an address on a particular network, neither the address nor the network classes are sufficient. Notation like `192.0.2.1/24` is commonly used by network engineers and the people who write tools for firewalls and routers as shorthand for “the host `192.0.2.1` on the network `192.0.2.0/24`”, Accordingly, **`ipaddress`** provides a set of hybrid classes that associate an address with a particular network. The interface for creation is identical to that for defining network objects, except that the address portion isn’t constrained to being a network address.

```
>>> ipaddress.ip_interface('192.0.2.1/24')
IPv4Interface('192.0.2.1/24')
>>> ipaddress.ip_interface('2001:db8::1/96')
IPv6Interface('2001:db8::1/96')
```

Integer inputs are accepted (as with networks), and use of a particular IP version can be forced by calling the relevant constructor directly.

Inspecting Address/Network/Interface Objects

You’ve gone to the trouble of creating an IPv(4|6) (Address|Network|Interface) object, so you probably want to get information about it. **`ipaddress`** tries to make doing this easy and intuitive.

Extracting the IP version:

```
>>> addr4 = ipaddress.ip_address('192.0.2.1')
>>> addr6 = ipaddress.ip_address('2001:db8::1')
>>> addr6.version
6
>>> addr4.version
4
```

Obtaining the network from an interface:

```
>>> host4 = ipaddress.ip_interface('192.0.2.1/24')
>>> host4.network
IPv4Network('192.0.2.0/24')
>>> host6 = ipaddress.ip_interface('2001:db8::1/96')
>>> host6.network
IPv6Network('2001:db8::/96')
```

Finding out how many individual addresses are in a network:

```
>>> net4 = ipaddress.ip_network('192.0.2.0/24')
>>> net4.num_addresses
```

```
256
>>> net6 = ipaddress.ip_network('2001:db8::0/96')
>>> net6.num_addresses
4294967296
```

Iterating through the “usable” addresses on a network:

```
>>> net4 = ipaddress.ip_network('192.0.2.0/24')
>>> for x in net4.hosts():
...     print(x)
192.0.2.1
192.0.2.2
192.0.2.3
192.0.2.4
...
192.0.2.252
192.0.2.253
192.0.2.254
```

Obtaining the netmask (i.e. set bits corresponding to the network prefix) or the hostmask (any bits that are not part of the netmask):

```
>>> net4 = ipaddress.ip_network('192.0.2.0/24')
>>> net4.netmask
IPv4Address('255.255.255.0')
>>> net4.hostmask
IPv4Address('0.0.0.255')
>>> net6 = ipaddress.ip_network('2001:db8::0/96')
>>> net6.netmask
IPv6Address('ffff:ffff:ffff:ffff:ffff:ffff::')
>>> net6.hostmask
IPv6Address('::ffff:ffff')
```

Exploding or compressing the address:

```
>>> addr6.exploded
'2001:0db8:0000:0000:0000:0000:0000:0001'
>>> addr6.compressed
'2001:db8::1'
>>> net6.exploded
'2001:0db8:0000:0000:0000:0000:0000:0000/96'
```

```
>>> net6.compressed
'2001:db8::/96'
```

While IPv4 doesn’t support explosion or compression, the associated objects still provide the relevant properties so that version neutral code can easily ensure the most concise or most verbose form is used for IPv6 addresses while still correctly handling IPv4 addresses.

Networks as lists of Addresses

It’s sometimes useful to treat networks as lists. This means it is possible to index them like this:

```
>>> net4[1]
IPv4Address('192.0.2.1')
>>> net4[-1]
IPv4Address('192.0.2.255')
>>> net6[1]
IPv6Address('2001:db8::1')
>>> net6[-1]
IPv6Address('2001:db8::ffff:ffff')
```

It also means that network objects lend themselves to using the list membership test syntax like this:

```
if address in network:
    # do something
```

Containment testing is done efficiently based on the network prefix:

```
>>> addr4 = ipaddress.ip_address('192.0.2.1')
>>> addr4 in ipaddress.ip_network('192.0.2.0/24')
True
>>> addr4 in ipaddress.ip_network('192.0.3.0/24')
False
```

Comparisons

`ipaddress` provides some simple, hopefully intuitive ways to compare objects, where it makes sense:

```
>>> ipaddress.ip_address('192.0.2.1') <
ipaddress.ip_address('192.0.2.2')
True
```

A **`TypeError`** exception is raised if you try to compare objects of different versions or different types.

Using IP Addresses with other modules

Other modules that use IP addresses (such as **`socket`**) usually won’t accept objects from this module directly. Instead, they must be coerced to an integer or string that the other module will accept:

```
>>> addr4 = ipaddress.ip_address('192.0.2.1')
>>> str(addr4)
'192.0.2.1'
>>> int(addr4)
3221225985
```

Getting more detail when instance creation fails

When creating address/network/interface objects using the version-agnostic factory functions, any errors will be reported as **`ValueError`** with a generic error message that simply says the passed in value was not recognized as an object of that type. The lack of a specific error is because it’s necessary to know whether the value is *supposed* to be IPv4 or IPv6 in order to provide more detail on why it has been rejected.

To support use cases where it is useful to have access to this additional detail, the individual class constructors actually raise the `ValueError` subclasses `ipaddress.AddressValueError` and `ipaddress.NetmaskValueError` to indicate exactly which part of the definition failed to parse correctly.

The error messages are significantly more detailed when using the class constructors directly. For example:

```
>>> ipaddress.ip_address("192.168.0.256")
Traceback (most recent call last):
  ...
ValueError: '192.168.0.256' does not appear to be an IPv4 or
IPv6 address
>>> ipaddress.IPv4Address("192.168.0.256")
Traceback (most recent call last):
  ...
ipaddress.AddressValueError: Octet 256 (> 255) not permitted in
'192.168.0.256'

>>> ipaddress.ip_network("192.168.0.1/64")
Traceback (most recent call last):
  ...
ValueError: '192.168.0.1/64' does not appear to be an IPv4 or
IPv6 network
>>> ipaddress.IPv4Network("192.168.0.1/64")
Traceback (most recent call last):
  ...
ipaddress.NetmaskValueError: '64' is not a valid netmask
```

However, both of the module specific exceptions have `ValueError` as their parent class, so if you’re not concerned with the particular type of error, you can still write code like the following:

```
try:
    network = ipaddress.IPv4Network(address)
except ValueError:
    print('address/netmask is invalid for IPv4:', address)
```

Instrumenting CPython with DTrace and SystemTap

author:: David Malcolm
author:: Łukasz Langa

DTrace and SystemTap are monitoring tools, each providing a way to inspect what the processes on a computer system are doing. They both use domain-specific languages allowing a user to write scripts which:

- filter which processes are to be observed
- gather data from the processes of interest
- generate reports on the data

As of Python 3.6, CPython can be built with embedded “markers”, also known as “probes”, that can be observed by a DTrace or SystemTap script, making it easier to monitor what the CPython processes on a system are doing.

CPython implementation detail: DTrace markers are implementation details of the CPython interpreter. No guarantees are made about probe compatibility between versions of CPython. DTrace scripts can stop working or work incorrectly without warning when changing CPython versions.

Enabling the static markers

macOS comes with built-in support for DTrace. On Linux, in order to build CPython with the embedded markers for

SystemTap, the SystemTap development tools must be installed.

On a Linux machine, this can be done via:

```
$ yum install systemtap-sdt-devel
```

or:

```
$ sudo apt-get install systemtap-sdt-dev
```

CPython must then be **configured with the `--with-dtrace` option**:

```
checking for --with-dtrace... yes
```

On macOS, you can list available DTrace probes by running a Python process in the background and listing all probes made available by the Python provider:

```
$ python3.6 -q &
$ sudo dtrace -l -P python$!  # or: dtrace -l -m python3.6

   ID   PROVIDER            MODULE
FUNCTION NAME
29564 python18035        python3.6
_PyEval_EvalFrameDefault function-entry
29565 python18035        python3.6
dtrace_function_entry function-entry
29566 python18035        python3.6
_PyEval_EvalFrameDefault function-return
29567 python18035        python3.6
dtrace_function_return function-return
29568 python18035        python3.6
collect gc-done
29569 python18035        python3.6
collect gc-start
29570 python18035        python3.6
_PyEval_EvalFrameDefault line
29571 python18035        python3.6
maybe_dtrace_line line
```

On Linux, you can verify if the SystemTap static markers are present in the built binary by seeing if it contains a “.note.stapsdt” section.

```
$ readelf -S ./python | grep .note.stapsdt
[30] .note.stapsdt        NOTE         0000000000000000
00308d78
```

If you’ve built Python as a shared library (with the [`--enable-shared`]() configure option), you need to look instead within the shared library. For example:

```
$ readelf -S libpython3.3dm.so.1.0 | grep .note.stapsdt
[29] .note.stapsdt        NOTE         0000000000000000
00365b68
```

Sufficiently modern readelf can print the metadata:

```
$ readelf -n ./python

Displaying notes found at file offset 0x00000254 with length
0x00000020:
    Owner                 Data size          Description
    GNU                  0x00000010          NT_GNU_ABI_TAG
(ABI version tag)
        OS: Linux, ABI: 2.6.32

Displaying notes found at file offset 0x00000274 with length
0x00000024:
    Owner                 Data size          Description
    GNU                  0x00000014          NT_GNU_BUILD_ID
(unique build ID bitstring)
        Build ID: df924a2b08a7e89f6e11251d4602022977af2670

Displaying notes found at file offset 0x002d6c30 with length
0x00000144:
    Owner                 Data size          Description
    stapsdt              0x00000031          NT_STAPSDT
(SystemTap probe descriptors)
        Provider: python
        Name: gc__start
        Location: 0x00000000004371c3, Base: 0x0000000000630ce2,
Semaphore: 0x00000000008d6bf6
        Arguments: -4@%ebx
    stapsdt              0x00000030          NT_STAPSDT
(SystemTap probe descriptors)
        Provider: python
        Name: gc__done
        Location: 0x00000000004374e1, Base: 0x0000000000630ce2,
Semaphore: 0x00000000008d6bf8
        Arguments: -8@%rax
    stapsdt              0x00000045          NT_STAPSDT
(SystemTap probe descriptors)
        Provider: python
        Name: function__entry
        Location: 0x000000000053db6c, Base: 0x0000000000630ce2,
Semaphore: 0x00000000008d6be8
        Arguments: 8@%rbp 8@%r12 -4@%eax
    stapsdt              0x00000046          NT_STAPSDT
(SystemTap probe descriptors)
        Provider: python
        Name: function__return
        Location: 0x000000000053dba8, Base: 0x0000000000630ce2,
Semaphore: 0x00000000008d6bea
        Arguments: 8@%rbp 8@%r12 -4@%eax
```

The above metadata contains information for SystemTap describing how it can patch strategically placed machine code instructions to enable the tracing hooks used by a SystemTap script.

Static DTrace probes

The following example DTrace script can be used to show the call/return hierarchy of a Python script, only tracing within the invocation of a function called “start”. In other words, import-time function invocations are not going to be listed:

```
self int indent;
```

```
python$target:::function-entry
/copyinstr(arg1) == "start"/
{
        self->trace = 1;
}

python$target:::function-entry
/self->trace/
{
        printf("%d\t%*s:", timestamp, 15, probename);
        printf("%*s", self->indent, "");
        printf("%s:%s:%d\n", basename(copyinstr(arg0)),
copyinstr(arg1), arg2);
        self->indent++;
}

python$target:::function-return
/self->trace/
{
        self->indent--;
        printf("%d\t%*s:", timestamp, 15, probename);
        printf("%*s", self->indent, "");
        printf("%s:%s:%d\n", basename(copyinstr(arg0)),
copyinstr(arg1), arg2);
}

python$target:::function-return
/copyinstr(arg1) == "start"/
{
        self->trace = 0;
}
```

It can be invoked like this:

```
$ sudo dtrace -q -s call_stack.d -c "python3.6 script.py"
```

The output looks like this:

```
156641360502280  function-entry:call_stack.py:start:23
156641360518804  function-entry: call_stack.py:function_1:1
156641360532797  function-entry:  call_stack.py:function_3:9
156641360546807 function-return:  call_stack.py:function_3:10
```

```
156641360563367 function-return: call_stack.py:function_1:2
156641360578365  function-entry: call_stack.py:function_2:5
156641360591757  function-entry:  call_stack.py:function_1:1
156641360605556  function-entry:   call_stack.py:function_3:9
156641360617482 function-return:   call_stack.py:function_3:10
156641360629814 function-return:  call_stack.py:function_1:2
156641360642285 function-return: call_stack.py:function_2:6
156641360656770  function-entry: call_stack.py:function_3:9
156641360669707 function-return: call_stack.py:function_3:10
156641360687853  function-entry: call_stack.py:function_4:13
156641360700719 function-return: call_stack.py:function_4:14
156641360719640  function-entry: call_stack.py:function_5:18
156641360732567 function-return: call_stack.py:function_5:21
156641360747370 function-return:call_stack.py:start:28
```

Static SystemTap markers

The low-level way to use the SystemTap integration is to use the static markers directly. This requires you to explicitly state the binary file containing them.

For example, this SystemTap script can be used to show the call/return hierarchy of a Python script:

```
probe process("python").mark("function__entry") {
     filename = user_string($arg1);
     funcname = user_string($arg2);
     lineno = $arg3;

     printf("%s => %s in %s:%d\\n",
            thread_indent(1), funcname, filename, lineno);
}

probe process("python").mark("function__return") {
    filename = user_string($arg1);
    funcname = user_string($arg2);
    lineno = $arg3;

    printf("%s <= %s in %s:%d\\n",
           thread_indent(-1), funcname, filename, lineno);
}
```

It can be invoked like this:

```
$ stap \
  show-call-hierarchy.stp \
  -c "./python test.py"
```

The output looks like this:

```
11408 python(8274):        => __contains__ in
Lib/_abcoll.py:362
11414 python(8274):         => __getitem__ in Lib/os.py:425
11418 python(8274):          => encode in Lib/os.py:490
11424 python(8274):          <= encode in Lib/os.py:493
11428 python(8274):         <= __getitem__ in Lib/os.py:426
11433 python(8274):        <= __contains__ in
Lib/_abcoll.py:366
```

where the columns are:

- time in microseconds since start of script
- name of executable
- PID of process

and the remainder indicates the call/return hierarchy as the script executes.

For a **`--enable-shared`** build of CPython, the markers are contained within the libpython shared library, and the probe’s dotted path needs to reflect this. For example, this line from the above example:

```
probe process("python").mark("function__entry") {
```

should instead read:

```
probe
process("python").library("libpython3.6dm.so.1.0").mark("functi
on__entry") {
```

(assuming a debug build of CPython 3.6)

Available static markers

function__entry(str filename, str funcname, int lineno)

This marker indicates that execution of a Python function has begun. It is only triggered for pure-Python (bytecode) functions.

The filename, function name, and line number are provided back to the tracing script as positional arguments, which must be accessed using `$arg1`, `$arg2`, `$arg3`:

- `$arg1` : `(const char *)` filename, accessible using `user_string($arg1)`
- `$arg2` : `(const char *)` function name, accessible using `user_string($arg2)`
- `$arg3` : `int` line number

function__return(str filename, str funcname, int lineno)

This marker is the converse of **`function__entry()`**, and indicates that execution of a Python function has ended (either via `return`, or via an exception). It is only triggered for pure-Python (bytecode) functions.

The arguments are the same as for **`function__entry()`**

line(str filename, str funcname, int lineno)

This marker indicates a Python line is about to be executed. It is the equivalent of line-by-line tracing with a Python profiler. It is not triggered within C functions.

The arguments are the same as for **`function__entry()`**.

gc__start(int generation)

Fires when the Python interpreter starts a garbage collection cycle. `arg0` is the generation to scan, like `gc.collect()`.

gc__done(long collected)

Fires when the Python interpreter finishes a garbage collection cycle. `arg0` is the number of collected objects.

import__find__load__start(str modulename)

Fires before `importlib` attempts to find and load the module. `arg0` is the module name.

Added in version 3.7.

import__find__load__done(str modulename, int found)

Fires after `importlib`'s find_and_load function is called. `arg0` is the module name, `arg1` indicates if module was successfully loaded.

Added in version 3.7.

audit(str event, void *tuple)

Fires when `sys.audit()` or `PySys_Audit()` is called. `arg0` is the event name as C string, `arg1` is a `PyObject` pointer to a tuple object.

Added in version 3.8.

SystemTap Tapsets

The higher-level way to use the SystemTap integration is to use a “tapset”: SystemTap’s equivalent of a library, which hides some of the lower-level details of the static markers.

Here is a tapset file, based on a non-shared build of CPython:

```
/*
   Provide a higher-level wrapping around the function__entry
and
   function__return markers:
 \*/
probe python.function.entry =
process("python").mark("function__entry")
{
    filename = user_string($arg1);
    funcname = user_string($arg2);
    lineno = $arg3;
    frameptr = $arg4
}
probe python.function.return =
process("python").mark("function__return")
{
    filename = user_string($arg1);
    funcname = user_string($arg2);
    lineno = $arg3;
    frameptr = $arg4
}
```

If this file is installed in SystemTap's tapset directory (e.g. `/usr/share/systemtap/tapset`), then these additional probepoints become available:

python.function.entry(str filename, str funcname, int lineno, frameptr)

> This probe point indicates that execution of a Python function has begun. It is only triggered for pure-Python (bytecode) functions.

python.function.return(str filename, str funcname, int lineno, frameptr)

> This probe point is the converse of `python.function.return`, and indicates that execution of a Python function has ended (either via `return`, or via an exception). It is only triggered for pure-Python (bytecode) functions.

Examples

This SystemTap script uses the tapset above to more cleanly implement the example given above of tracing the Python function-call hierarchy, without needing to directly name the static markers:

```
probe python.function.entry
{
  printf("%s => %s in %s:%d\n",
         thread_indent(1), funcname, filename, lineno);
}

probe python.function.return
{
  printf("%s <= %s in %s:%d\n",
         thread_indent(-1), funcname, filename, lineno);
}
```

The following script uses the tapset above to provide a top-like view of all running CPython code, showing the top 20 most frequently entered bytecode frames, each second, across the whole system:

```
global fn_calls;

probe python.function.entry
{
    fn_calls[pid(), filename, funcname, lineno] += 1;
}

probe timer.ms(1000) {
    printf("\033[2J\033[1;1H") /* clear screen \*/
    printf("%6s %80s %6s %30s %6s\n",
           "PID", "FILENAME", "LINE", "FUNCTION", "CALLS")
    foreach ([pid, filename, funcname, lineno] in fn_calls-
limit 20) {
        printf("%6d %80s %6d %30s %6d\n",
            pid, filename, lineno, funcname,
            fn_calls[pid, filename, funcname, lineno]);
    }
    delete fn_calls;
}
```

Python support for the Linux `perf` profiler

author:: Pablo Galindo

The Linux perf profiler [https://perf.wiki.kernel.org] is a very powerful tool that allows you to profile and obtain information about the performance of your application. `perf` also has a very vibrant ecosystem of tools that aid with the analysis of the data that it produces.

The main problem with using the `perf` profiler with Python applications is that `perf` only gets information about native symbols, that is, the names of functions and procedures written in C. This means that the names and file names of Python functions in your code will not appear in the output of `perf`.

Since Python 3.12, the interpreter can run in a special mode that allows Python functions to appear in the output of the `perf` profiler. When this mode is enabled, the interpreter will interpose a small piece of code compiled on the fly before the execution of every Python function and it will teach `perf` the relationship between this piece of code and the associated Python function using perf map files.

Note

Support for the `perf` profiler is currently only available for Linux on select architectures. Check the output of the `configure` build step or check the output of `python -m` `sysconfig | grep HAVE_PERF_TRAMPOLINE` to see if your system is supported.

For example, consider the following script:

```
def foo(n):
    result = 0
    for _ in range(n):
        result += 1
    return result

def bar(n):
    foo(n)

def baz(n):
    bar(n)

if __name__ == "__main__":
    baz(1000000)
```

We can run `perf` to sample CPU stack traces at 9999 hertz:

```
$ perf record -F 9999 -g -o perf.data python my_script.py
```

Then we can use `perf report` to analyze the data:

```
$ perf report --stdio -n -g

# Children      Self       Samples  Command     Shared Object
Symbol
# ........  ........  ...........  ..........
..................  ..........................................
#
    91.08%     0.00%             0  python.exe  python.exe
[.] _start
            |
            ---_start
            |
                --90.71%--__libc_start_main
                        Py_BytesMain
                        |
                        |--56.88%--
pymain_run_python.constprop.0
                        |          |
                        |          |--56.13%--
_PyRun_AnyFileObject
                        |          |
_PyRun_SimpleFileObject
                        |          |          |
                        |          |          |--55.02%--
run_mod
                        |          |          |          |
                        |          |          |           -
-54.65%--PyEval_EvalCode
                        |          |          |
_PyEval_EvalFrameDefault
                        |          |          |
PyObject_Vectorcall
                        |          |          |
_PyEval_Vector
                        |          |          |
_PyEval_EvalFrameDefault
                        |          |          |
PyObject_Vectorcall
                        |          |          |
_PyEval_Vector
                        |          |          |
_PyEval_EvalFrameDefault
                        |          |          |
PyObject_Vectorcall
                        |          |          |
_PyEval_Vector
                        |          |          |
|
                        |          |          |
|--51.67%--_PyEval_EvalFrameDefault
                        |          |          |
|          |
                        |          |          |
|          |--11.52%--_PyLong_Add
                        |          |          |
|          |          |
                        |          |          |
|           |           |--2.97%--_PyObject_Malloc
...
```

As you can see, the Python functions are not shown in the output, only `_PyEval_EvalFrameDefault` (the function that evaluates the Python bytecode) shows up. Unfortunately that's not very useful because all Python functions use the same C function to evaluate bytecode so we cannot know which Python function corresponds to which bytecode-evaluating function.

Instead, if we run the same experiment with `perf` support enabled we get:

```
$ perf report --stdio -n -g

# Children      Self       Samples  Command     Shared Object
Symbol
# ........  ........  ............  ..........
..................
...............................................................
......
#
    90.58%     0.36%             1  python.exe  python.exe
[.] _start
            |
            ---_start
            |
                --89.86%--__libc_start_main
                        Py_BytesMain
                        |
                        |--55.43%--
pymain_run_python.constprop.0
                        |          |
                        |          |--54.71%--
_PyRun_AnyFileObject
                        |          |
_PyRun_SimpleFileObject
                        |          |          |
                        |          |          |--53.62%--
run_mod
```

```
                        |          |          |          |
                        |          |          |           -
-53.26%--PyEval_EvalCode
                        |          |          |
py::<module>:/src/script.py
                        |          |          |
_PyEval_EvalFrameDefault
                        |          |          |
PyObject_Vectorcall
                        |          |          |
_PyEval_Vector
                        |          |          |
py::baz:/src/script.py
                        |          |          |
_PyEval_EvalFrameDefault
                        |          |          |
PyObject_Vectorcall
                        |          |          |
_PyEval_Vector
                        |          |          |
py::bar:/src/script.py
                        |          |          |
_PyEval_EvalFrameDefault
                        |          |          |
PyObject_Vectorcall
                        |          |          |
_PyEval_Vector
                        |          |          |
py::foo:/src/script.py
                        |          |          |
|
                        |          |          |
|--51.81%--_PyEval_EvalFrameDefault
                        |          |          |
|          |
                        |          |          |
|          |--13.77%--_PyLong_Add
                        |          |          |
|          |          |
                        |          |          |
|          |          |--3.26%--_PyObject_Malloc
```

How to enable `perf` profiling support

`perf` profiling support can be enabled either from the start using the environment variable `PYTHONPERFSUPPORT` or the `-X perf` option, or dynamically using `sys.activate_stack_trampoline()` and `sys.deactivate_stack_trampoline()`.

The `sys` functions take precedence over the `-X` option, the `-X` option takes precedence over the environment variable.

Example, using the environment variable:

```
$ PYTHONPERFSUPPORT=1 perf record -F 9999 -g -o perf.data
python my_script.py
$ perf report -g -i perf.data
```

Example, using the `-X` option:

```
$ perf record -F 9999 -g -o perf.data python -X perf
my_script.py
$ perf report -g -i perf.data
```

Example, using the `sys` APIs in file `example.py`:

```
import sys

sys.activate_stack_trampoline("perf")
do_profiled_stuff()
sys.deactivate_stack_trampoline()

non_profiled_stuff()
```

…then:

```
$ perf record -F 9999 -g -o perf.data python ./example.py
$ perf report -g -i perf.data
```

How to obtain the best results

For best results, Python should be compiled with `CFLAGS="-fno-omit-frame-pointer -mno-omit-leaf-frame-pointer"` as this allows profilers to unwind using only the frame pointer and not on DWARF debug information. This is because as the code that is interposed to allow `perf` support is dynamically generated it doesn’t have any DWARF debugging information available.

You can check if your system has been compiled with this flag by running:

```
$ python -m sysconfig | grep 'no-omit-frame-pointer'
```

If you don’t see any output it means that your interpreter has not been compiled with frame pointers and therefore it may not be able to show Python functions in the output of `perf`.

How to work without frame pointers

If you are working with a Python interpreter that has been compiled without frame pointers, you can still use the `perf` profiler, but the overhead will be a bit higher because Python needs to generate unwinding information for every Python function call on the fly. Additionally, `perf` will take more time to process the data because it will need to use the DWARF debugging information to unwind the stack and this is a slow process.

To enable this mode, you can use the environment variable `PYTHON_PERF_JIT_SUPPORT` or the `-X perf_jit` option, which will enable the JIT mode for the `perf` profiler.

Note

Due to a bug in the `perf` tool, only `perf` versions higher than v6.8 will work with the JIT mode. The fix was also backported to the v6.7.2 version of the tool.

Note that when checking the version of the `perf` tool (which can be done by running `perf version`) you must take into account that some distros add some custom version numbers including a `-` character. This means that `perf 6.7-3` is not necessarily `perf 6.7.3`.

When using the perf JIT mode, you need an extra step before you can run `perf report`. You need to call the `perf inject` command to inject the JIT information into the `perf.data` file.:

```
$ perf record -F 9999 -g -k 1 --call-graph dwarf -o perf.data
python -Xperf_jit my_script.py
$ perf inject -i perf.data --jit --output perf.jit.data
$ perf report -g -i perf.jit.data
```

or using the environment variable:

```
$ PYTHON_PERF_JIT_SUPPORT=1 perf record -F 9999 -g --call-graph
dwarf -o perf.data python my_script.py
$ perf inject -i perf.data --jit --output perf.jit.data
$ perf report -g -i perf.jit.data
```

`perf inject --jit` command will read `perf.data`, automatically pick up the perf dump file that Python creates (in `/tmp/perf-$PID.dump`), and then create `perf.jit.data` which merges all the JIT information together. It should also create a lot of `jitted-XXXX-N.so` files in the current directory which are ELF images for all the JIT trampolines that were created by Python.

Warning

When using `--call-graph dwarf`, the `perf` tool will take snapshots of the stack of the process being profiled and save the information in the `perf.data` file. By default, the size of the stack dump is 8192 bytes, but you can change the size by passing it after a comma like `--call-graph dwarf,16384`.

The size of the stack dump is important because if the size is too small `perf` will not be able to unwind the stack and the output will be incomplete. On the other hand, if the size is too big, then `perf` won’t be able to sample the process as frequently as it would like as the overhead will be higher.

The stack size is particularly important when profiling Python code compiled with low optimization levels (like `-O0`), as these builds tend to have larger stack frames. If you are compiling Python with `-O0` and not seeing Python functions in your profiling output, try increasing the stack dump size to 65528 bytes (the maximum):

```
$ perf record -F 9999 -g -k 1 --call-graph dwarf,65528 -o
perf.data python -Xperf_jit my_script.py
```

Different compilation flags can significantly impact stack sizes:

- Builds with `-O0` typically have much larger stack frames than those with `-O1` or higher
- Adding optimizations (`-O1`, `-O2`, etc.) typically reduces stack size
- Frame pointers (`-fno-omit-frame-pointer`) generally provide more reliable stack unwinding

Annotations Best Practices

author:: Larry Hastings

Abstract

This document is designed to encapsulate the best practices for working with annotations dicts. If you write Python code that examines `__annotations__` on Python objects, we encourage you to follow the guidelines described below.

The document is organized into four sections: best practices for accessing the annotations of an object in Python versions 3.10 and newer, best practices for accessing the annotations of an object in Python versions 3.9 and older, other best practices for `__annotations__` that apply to any Python version, and quirks of `__annotations__`.

Note that this document is specifically about working with `__annotations__`, not uses *for* annotations. If you're looking for information on how to use "type hints" in your code, please see the `typing` module.

Accessing The Annotations Dict Of An Object In Python 3.10 And Newer

Python 3.10 adds a new function to the standard library: `inspect.get_annotations()`. In Python versions 3.10 through 3.13, calling this function is the best practice for accessing the annotations dict of any object that supports annotations.

This function can also “un-stringize” stringized annotations for you.

In Python 3.14, there is a new `annotationlib` module with functionality for working with annotations. This includes a `annotationlib.get_annotations()` function, which supersedes `inspect.get_annotations()`.

If for some reason `inspect.get_annotations()` isn’t viable for your use case, you may access the `__annotations__` data member manually. Best practice for this changed in Python 3.10 as well: as of Python 3.10, `o.__annotations__` is guaranteed to *always* work on Python functions, classes, and modules. If you’re certain the object you’re examining is one of these three *specific* objects, you may simply use `o.__annotations__` to get at the object’s annotations dict.

However, other types of callables–for example, callables created by `functools.partial()`–may not have an `__annotations__` attribute defined. When accessing the `__annotations__` of a possibly unknown object, best practice in Python versions 3.10 and newer is to call `getattr()` with three arguments, for example `getattr(o, '__annotations__', None)`.

Before Python 3.10, accessing `__annotations__` on a class that defines no annotations but that has a parent class with annotations would return the parent’s `__annotations__`. In Python 3.10 and newer, the child class’s annotations will be an empty dict instead.

Accessing The Annotations Dict Of An Object In Python 3.9 And Older

In Python 3.9 and older, accessing the annotations dict of an object is much more complicated than in newer versions. The problem is a design flaw in these older versions of Python, specifically to do with class annotations.

Best practice for accessing the annotations dict of other objects–functions, other callables, and modules–is the same as best practice for 3.10, assuming you aren't calling `inspect.get_annotations()`: you should use three-argument `getattr()` to access the object's `__annotations__` attribute.

Unfortunately, this isn't best practice for classes. The problem is that, since `__annotations__` is optional on classes, and because classes can inherit attributes from their base classes, accessing the `__annotations__` attribute of a class may inadvertently return the annotations dict of a *base class.* As an example:

```
class Base:
    a: int = 3
    b: str = 'abc'

class Derived(Base):
    pass

print(Derived.__annotations__)
```

This will print the annotations dict from `Base`, not `Derived`.

Your code will have to have a separate code path if the object you're examining is a class (`isinstance(o, type)`). In that case, best practice relies on an implementation detail of Python 3.9 and before: if a class has annotations defined, they are stored in the class's `__dict__` dictionary. Since the class may or may not have annotations defined, best practice is to call the `get()` method on the class dict.

To put it all together, here is some sample code that safely accesses the `__annotations__` attribute on an arbitrary object in Python 3.9 and before:

```
if isinstance(o, type):
    ann = o.__dict__.get('__annotations__', None)
else:
    ann = getattr(o, '__annotations__', None)
```

After running this code, `ann` should be either a dictionary or `None`. You’re encouraged to double-check the type of `ann` using **`isinstance()`** before further examination.

Note that some exotic or malformed type objects may not have a **`__dict__`** attribute, so for extra safety you may also wish to use **`getattr()`** to access **`__dict__`**.

Manually Un-Stringizing Stringized Annotations

In situations where some annotations may be “stringized”, and you wish to evaluate those strings to produce the Python values they represent, it really is best to call **`inspect.get_annotations()`** to do this work for you.

If you’re using Python 3.9 or older, or if for some reason you can’t use **`inspect.get_annotations()`**, you’ll need to duplicate its logic. You’re encouraged to examine the implementation of **`inspect.get_annotations()`** in the current Python version and follow a similar approach.

In a nutshell, if you wish to evaluate a stringized annotation on an arbitrary object `o`:

- If `o` is a module, use `o.__dict__` as the `globals` when calling **`eval()`**.

- If `o` is a class, use `sys.modules[o.__module__].__dict__` as the `globals`, and `dict(vars(o))` as the `locals`, when calling `eval()`.
- If `o` is a wrapped callable using `functools.update_wrapper()`, `functools.wraps()`, or `functools.partial()`, iteratively unwrap it by accessing either `o.__wrapped__` or `o.func` as appropriate, until you have found the root unwrapped function.
- If `o` is a callable (but not a class), use `o.__globals__` as the globals when calling `eval()`.

However, not all string values used as annotations can be successfully turned into Python values by `eval()`. String values could theoretically contain any valid string, and in practice there are valid use cases for type hints that require annotating with string values that specifically *can't* be evaluated. For example:

- **PEP 604** [https://peps.python.org/pep-0604/] union types using `|`, before support for this was added to Python 3.10.
- Definitions that aren't needed at runtime, only imported when `typing.TYPE_CHECKING` is true.

If `eval()` attempts to evaluate such values, it will fail and raise an exception. So, when designing a library API that works with annotations, it's recommended to only attempt to evaluate string values when explicitly requested to by the caller.

Best Practices For `__annotations__` In Any Python Version

- You should avoid assigning to the `__annotations__` member of objects directly. Let Python manage setting `__annotations__`.

- If you do assign directly to the `__annotations__` member of an object, you should always set it to a `dict` object.
- You should avoid accessing `__annotations__` directly on any object. Instead, use `annotationlib.get_annotations()` (Python 3.14+) or `inspect.get_annotations()` (Python 3.10+).
- If you do directly access the `__annotations__` member of an object, you should ensure that it's a dictionary before attempting to examine its contents.
- You should avoid modifying `__annotations__` dicts.
- You should avoid deleting the `__annotations__` attribute of an object.

`__annotations__` Quirks

In all versions of Python 3, function objects lazy-create an annotations dict if no annotations are defined on that object. You can delete the `__annotations__` attribute using `del fn.__annotations__`, but if you then access `fn.__annotations__` the object will create a new empty dict that it will store and return as its annotations. Deleting the annotations on a function before it has lazily created its annotations dict will throw an `AttributeError`; using `del fn.__annotations__` twice in a row is guaranteed to always throw an `AttributeError`.

Everything in the above paragraph also applies to class and module objects in Python 3.10 and newer.

In all versions of Python 3, you can set `__annotations__` on a function object to `None`. However, subsequently accessing the annotations on that object using `fn.__annotations__` will lazy-create an empty dictionary as per the first paragraph of this section. This is *not* true of modules and classes, in any Python version; those objects permit setting `__annotations__` to any Python value, and will retain whatever value is set.

If Python stringizes your annotations for you (using `from __future__ import annotations`), and you specify a string as an annotation, the string will itself be quoted. In effect the annotation is quoted *twice.* For example:

```
from __future__ import annotations
def foo(a: "str"): pass

print(foo.__annotations__)
```

This prints `{'a': "'str'"}`. This shouldn't really be considered a "quirk"; it's mentioned here simply because it might be surprising.

If you use a class with a custom metaclass and access `__annotations__` on the class, you may observe unexpected behavior; see **749** [https://peps.python.org/pep-0749/#pep749-metaclasses] for some examples. You can avoid these quirks by using **`annotationlib.get_annotations()`** on Python 3.14+ or **`inspect.get_annotations()`** on Python 3.10+. On earlier versions of Python, you can avoid these bugs by accessing the annotations from the class's **`__dict__`** (for example, `cls.__dict__.get('__annotations__', None)`).

In some versions of Python, instances of classes may have an `__annotations__` attribute. However, this is not supported functionality. If you need the annotations of an instance, you can use **`type()`** to access its class (for example, `annotationlib.get_annotations(type(myinstance))` on Python 3.14+).

Isolating Extension Modules

Abstract

Traditionally, state belonging to Python extension modules was kept in C `static` variables, which have process-wide scope. This document describes problems of such per-process state and shows a safer way: per-module state.

The document also describes how to switch to per-module state where possible. This transition involves allocating space for that state, potentially switching from static types to heap types, and—perhaps most importantly—accessing per-module state from code.

Who should read this

This guide is written for maintainers of C-API extensions who would like to make that extension safer to use in applications where Python itself is used as a library.

Background

An *interpreter* is the context in which Python code runs. It contains configuration (e.g. the import path) and runtime state (e.g. the set of imported modules).

Python supports running multiple interpreters in one process. There are two cases to think about—users may run interpreters:

- in sequence, with several `Py_InitializeEx()`/`Py_FinalizeEx()` cycles, and
- in parallel, managing “sub-interpreters” using `Py_NewInterpreter()`/`Py_EndInterpreter()`.

Both cases (and combinations of them) would be most useful when embedding Python within a library. Libraries generally shouldn’t make assumptions about the application that uses them, which include assuming a process-wide “main Python interpreter”.

Historically, Python extension modules don’t handle this use case well. Many extension modules (and even some stdlib modules) use *per-process* global state, because C `static` variables are extremely easy to use. Thus, data that should be specific to an interpreter ends up being shared between interpreters. Unless the extension developer is careful, it is very easy to introduce edge cases that lead to crashes when a module is loaded in more than one interpreter in the same process.

Unfortunately, *per-interpreter* state is not easy to achieve. Extension authors tend to not keep multiple interpreters in mind when developing, and it is currently cumbersome to test the behavior.

Enter Per-Module State

Instead of focusing on per-interpreter state, Python’s C API is evolving to better support the more granular *per-module* state. This means that C-level data should be attached to a *module object*. Each interpreter creates its own module object, keeping the data separate. For testing the isolation, multiple module objects corresponding to a single extension can even be loaded in a single interpreter.

Per-module state provides an easy way to think about lifetime and resource ownership: the extension module will initialize when a module object is created, and clean up when it’s freed. In this regard, a module is just like any other PyObject*; there are no “on interpreter shutdown” hooks to think—or forget—about.

Note that there are use cases for different kinds of “globals”: per-process, per-interpreter, per-thread or per-task state. With per-module state as the default, these are still possible, but you should treat them as exceptional cases: if you need them, you should give them additional care and testing. (Note that this guide does not cover them.)

Isolated Module Objects

The key point to keep in mind when developing an extension module is that several module objects can be created from a single shared library. For example:

```
>>> import sys
>>> import binascii
>>> old_binascii = binascii
>>> del sys.modules['binascii']
>>> import binascii  # create a new module object
>>> old_binascii == binascii
False
```

As a rule of thumb, the two modules should be completely independent. All objects and state specific to the module should be encapsulated within the module object, not shared with other module objects, and cleaned up when the module object is deallocated. Since this just is a rule of thumb, exceptions are possible (see Managing Global State), but they will need more thought and attention to edge cases.

While some modules could do with less stringent restrictions, isolated modules make it easier to set clear expectations and guidelines that work across a variety of use cases.

Surprising Edge Cases

Note that isolated modules do create some surprising edge cases. Most notably, each module object will typically not share its classes and exceptions with other similar modules. Continuing from the example above, note that `old_binascii.Error` and `binascii.Error` are separate objects. In the following code, the exception is *not* caught:

```
>>> old_binascii.Error == binascii.Error
False
>>> try:
...     old_binascii.unhexlify(b'qwertyuiop')
... except binascii.Error:
...     print('boo')
...
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
binascii.Error: Non-hexadecimal digit found
```

This is expected. Notice that pure-Python modules behave the same way: it is a part of how Python works.

The goal is to make extension modules safe at the C level, not to make hacks behave intuitively. Mutating `sys.modules` “manually” counts as a hack.

Making Modules Safe with Multiple Interpreters

Managing Global State

Sometimes, the state associated with a Python module is not specific to that module, but to the entire process (or something else “more global” than a module). For example:

- The `readline` module manages *the* terminal.
- A module running on a circuit board wants to control *the* on-board LED.

In these cases, the Python module should provide *access* to the global state, rather than *own* it. If possible, write the module so that multiple copies of it can access the state independently (along with other libraries, whether for Python or other languages). If that is not possible, consider explicit locking.

If it is necessary to use process-global state, the simplest way to avoid issues with multiple interpreters is to explicitly prevent a module from being loaded more than once per process—see Opt-Out: Limiting to One Module Object per Process.

Managing Per-Module State

To use per-module state, use multi-phase extension module initialization. This signals that your module supports multiple interpreters correctly.

Set `PyModuleDef.m_size` to a positive number to request that many bytes of storage local to the module. Usually, this will be set to the size of some module-specific `struct`, which can store all of the module’s C-level state. In particular, it is where you should put pointers to classes (including exceptions, but excluding static types) and settings (e.g. `csv`’s **`field_size_limit`**) which the C code needs to function.

Note

Another option is to store state in the module’s `__dict__`, but you must avoid crashing when users modify `__dict__` from Python code. This usually means error- and type-checking at the C level, which is easy to get wrong and hard to test sufficiently.

However, if module state is not needed in C code, storing it in `__dict__` only is a good idea.

If the module state includes `PyObject` pointers, the module object must hold references to those objects and implement the module-level hooks `m_traverse`, `m_clear` and `m_free`. These work like `tp_traverse`, `tp_clear` and `tp_free` of a class. Adding them will require some work and make the code longer; this is the price for modules which can be unloaded cleanly.

An example of a module with per-module state is currently available as xxlimited [https://github.com/python/cpython/blob/master/Modules/xxlimited.c]; example module initialization shown at the bottom of the file.

Opt-Out: Limiting to One Module Object per Process

A non-negative `PyModuleDef.m_size` signals that a module supports multiple interpreters correctly. If this is not yet the case for your module, you can explicitly make your module loadable only once per process. For example:

```
// A process-wide flag
static int loaded = 0;

// Mutex to provide thread safety (only needed for free-
threaded Python)
static PyMutex modinit_mutex = {0};

static int
exec_module(PyObject* module)
{
    PyMutex_Lock(&modinit_mutex);
    if (loaded) {
        PyMutex_Unlock(&modinit_mutex);
        PyErr_SetString(PyExc_ImportError,
                        "cannot load module more than once per
process");
        return -1;
    }
    loaded = 1;
    PyMutex_Unlock(&modinit_mutex);
    // ... rest of initialization
}
```

If your module's `PyModuleDef.m_clear` function is able to prepare for future re-initialization, it should clear the `loaded` flag. In this case, your module won't support multiple instances existing *concurrently*, but it will, for example, support being loaded after Python runtime shutdown (`Py_FinalizeEx()`) and re-initialization (`Py_Initialize()`).

Module State Access from Functions

Accessing the state from module-level functions is straightforward. Functions get the module object as their first argument; for extracting the state, you can use `PyModule_GetState`:

```
static PyObject *
func(PyObject *module, PyObject *args)
{
    my_struct *state = (my_struct*)PyModule_GetState(module);
    if (state == NULL) {
        return NULL;
    }
    // ... rest of logic
}
```

Note

`PyModule_GetState` may return `NULL` without setting an exception if there is no module state, i.e. `PyModuleDef.m_size` was zero. In your own module, you're in control of `m_size`, so this is easy to prevent.

Heap Types

Traditionally, types defined in C code are *static*; that is, `static PyTypeObject` structures defined directly in code and initialized using `PyType_Ready()`.

Such types are necessarily shared across the process. Sharing them between module objects requires paying attention to any state they own or access. To limit the possible issues, static types are immutable at the Python level: for example, you can't set `str.myattribute = 123`.

CPython implementation detail: Sharing truly immutable objects between interpreters is fine, as long as they don't provide access to mutable objects. However, in CPython, every Python object has a mutable implementation detail: the reference count. Changes to the refcount are guarded by the GIL. Thus, code that shares any Python objects across interpreters implicitly depends on CPython’s current, process-wide GIL.

Because they are immutable and process-global, static types cannot access “their” module state. If any method of such a type requires access to module state, the type must be converted to a *heap-allocated type*, or *heap type* for short. These correspond more closely to classes created by Python’s `class` statement.

For new modules, using heap types by default is a good rule of thumb.

Changing Static Types to Heap Types

Static types can be converted to heap types, but note that the heap type API was not designed for “lossless” conversion from static types—that is, creating a type that works exactly like a given static type. So, when rewriting the class definition in a new API, you are likely to unintentionally change a few details (e.g. pickleability or inherited slots). Always test the details that are important to you.

Watch out for the following two points in particular (but note that this is not a comprehensive list):

- Unlike static types, heap type objects are mutable by default. Use the `Py_TPFLAGS_IMMUTABLETYPE` flag to prevent mutability.
- Heap types inherit `tp_new` by default, so it may become possible to instantiate them from Python code. You can prevent this with the `Py_TPFLAGS_DISALLOW_INSTANTIATION` flag.

Defining Heap Types

Heap types can be created by filling a `PyType_Spec` structure, a description or “blueprint” of a class, and calling `PyType_FromModuleAndSpec()` to construct a new class object.

Note

Other functions, like `PyType_FromSpec()`, can also create heap types, but `PyType_FromModuleAndSpec()` associates the module with the class, allowing access to the module state from methods.

The class should generally be stored in *both* the module state (for safe access from C) and the module’s `__dict__` (for access from Python code).

Garbage-Collection Protocol

Instances of heap types hold a reference to their type. This ensures that the type isn’t destroyed before all its instances are, but may result in reference cycles that need to be broken by the garbage collector.

To avoid memory leaks, instances of heap types must implement the garbage collection protocol. That is, heap types should:

- Have the `Py_TPFLAGS_HAVE_GC` flag.
- Define a traverse function using `Py_tp_traverse`, which visits the type (e.g. using `Py_VISIT(Py_TYPE(self))`).

Please refer to the documentation of `Py_TPFLAGS_HAVE_GC` and `tp_traverse` for additional considerations.

The API for defining heap types grew organically, leaving it somewhat awkward to use in its current state. The following sections will guide you through common issues.

`tp_traverse` in Python 3.8 and lower

The requirement to visit the type from `tp_traverse` was added in Python 3.9. If you support Python 3.8 and lower, the traverse function must *not* visit the type, so it must be more complicated:

```
static int my_traverse(PyObject *self, visitproc visit, void
*arg)
{
    if (Py_Version >= 0x03090000) {
        Py_VISIT(Py_TYPE(self));
    }
    return 0;
}
```

Unfortunately, `Py_Version` was only added in Python 3.11. As a replacement, use:

- `PY_VERSION_HEX`, if not using the stable ABI, or
- `sys.version_info` (via `PySys_GetObject()` and `PyArg_ParseTuple()`).

Delegating `tp_traverse`

If your traverse function delegates to the `tp_traverse` of its base class (or another type), ensure that `Py_TYPE(self)` is visited only once. Note that only heap type are expected to visit the type in `tp_traverse`.

For example, if your traverse function includes:

```
base->tp_traverse(self, visit, arg)
```

…and `base` may be a static type, then it should also include:

```
if (base->tp_flags & Py_TPFLAGS_HEAPTYPE) {
    // a heap type's tp_traverse already visited Py_TYPE(self)
} else {
    if (Py_Version >= 0x03090000) {
        Py_VISIT(Py_TYPE(self));
    }
}
```

It is not necessary to handle the type’s reference count in **`tp_new`** and **`tp_clear`**.

Defining `tp_dealloc`

If your type has a custom **`tp_dealloc`** function, it needs to:

- call **`PyObject_GC_UnTrack()`** before any fields are invalidated, and
- decrement the reference count of the type.

To keep the type valid while `tp_free` is called, the type’s refcount needs to be decremented *after* the instance is deallocated. For example:

```
static void my_dealloc(PyObject *self)
{
    PyObject_GC_UnTrack(self);
    ...
    PyTypeObject *type = Py_TYPE(self);
    type->tp_free(self);
    Py_DECREF(type);
}
```

The default `tp_dealloc` function does this, so if your type does *not* override `tp_dealloc` you don’t need to add it.

Not overriding `tp_free`

The `tp_free` slot of a heap type must be set to `PyObject_GC_Del()`. This is the default; do not override it.

Avoiding `PyObject_New`

GC-tracked objects need to be allocated using GC-aware functions.

If you use `PyObject_New()` or `PyObject_NewVar()`:

- Get and call type's `tp_alloc` slot, if possible. That is, replace `TYPE *o = PyObject_New(TYPE, typeobj)` with:

  ```
  TYPE *o = typeobj->tp_alloc(typeobj, 0);
  ```

  Replace `o = PyObject_NewVar(TYPE, typeobj, size)` with the same, but use size instead of the 0.

- If the above is not possible (e.g. inside a custom `tp_alloc`), call `PyObject_GC_New()` or `PyObject_GC_NewVar()`:

  ```
  TYPE *o = PyObject_GC_New(TYPE, typeobj);
  ```

  ```
  TYPE *o = PyObject_GC_NewVar(TYPE, typeobj, size);
  ```

Module State Access from Classes

If you have a type object defined with `PyType_FromModuleAndSpec()`, you can call `PyType_GetModule()` to get the associated module, and then `PyModule_GetState()` to get the module's state.

To save a some tedious error-handling boilerplate code, you can combine these two steps with `PyType_GetModuleState()`, resulting in:

```
my_struct *state = (my_struct*)PyType_GetModuleState(type);
if (state == NULL) {
```

```
    return NULL;
}
```

Module State Access from Regular Methods

Accessing the module-level state from methods of a class is somewhat more complicated, but is possible thanks to API introduced in Python 3.9. To get the state, you need to first get the *defining class*, and then get the module state from it.

The largest roadblock is getting *the class a method was defined in*, or that method's "defining class" for short. The defining class can have a reference to the module it is part of.

Do not confuse the defining class with `Py_TYPE(self)`. If the method is called on a *subclass* of your type, `Py_TYPE(self)` will refer to that subclass, which may be defined in different module than yours.

Note

The following Python code can illustrate the concept. `Base.get_defining_class` returns `Base` even if `type(self) == Sub`:

```
class Base:
    def get_type_of_self(self):
        return type(self)

    def get_defining_class(self):
        return __class__

class Sub(Base):
    pass
```

For a method to get its “defining class”, it must use the METH_METHOD | METH_FASTCALL | METH_KEYWORDS `calling convention` and the corresponding `PyCMethod` signature:

```
PyObject *PyCMethod(
    PyObject *self,               // object the method was
called on
    PyTypeObject *defining_class, // defining class
    PyObject *const *args,        // C array of arguments
    Py_ssize_t nargs,             // length of "args"
    PyObject *kwnames)            // NULL, or dict of keyword
arguments
```

Once you have the defining class, call `PyType_GetModuleState()` to get the state of its associated module.

For example:

```
static PyObject *
example_method(PyObject *self,
        PyTypeObject *defining_class,
        PyObject *const *args,
        Py_ssize_t nargs,
        PyObject *kwnames)
{
    my_struct *state =
(my_struct*)PyType_GetModuleState(defining_class);
    if (state == NULL) {
        return NULL;
    }
    ... // rest of logic
}

PyDoc_STRVAR(example_method_doc, "...");

static PyMethodDef my_methods[] = {
    {"example_method",
      (PyCFunction)(void(*)(void))example_method,
      METH_METHOD|METH_FASTCALL|METH_KEYWORDS,
      example_method_doc}
    {NULL},
}
```

Module State Access from Slot Methods, Getters and Setters

Note

This is new in Python 3.11.

Slot methods—the fast C equivalents for special methods, such as `nb_add` for `__add__` or `tp_new` for initialization—have a very simple API that doesn't allow passing in the defining class, unlike with `PyCMethod`. The same goes for getters and setters defined with `PyGetSetDef`.

To access the module state in these cases, use the `PyType_GetModuleByDef()` function, and pass in the module definition. Once you have the module, call `PyModule_GetState()` to get the state:

```
PyObject *module = PyType_GetModuleByDef(Py_TYPE(self),
&module_def);
my_struct *state = (my_struct*)PyModule_GetState(module);
if (state == NULL) {
    return NULL;
}
```

`PyType_GetModuleByDef()` works by searching the method resolution order (i.e. all superclasses) for the first superclass that has a corresponding module.

Note

In very exotic cases (inheritance chains spanning multiple modules created from the same definition), **`PyType_GetModuleByDef()`** might not return the module of the true defining class. However, it will always return a module with the same definition, ensuring a compatible C memory layout.

Lifetime of the Module State

When a module object is garbage-collected, its module state is freed. For each pointer to (a part of) the module state, you must hold a reference to the module object.

Usually this is not an issue, because types created with `PyType_FromModuleAndSpec()`, and their instances, hold a reference to the module. However, you must be careful in reference counting when you reference module state from other places, such as callbacks for external libraries.

Open Issues

Several issues around per-module state and heap types are still open.

Discussions about improving the situation are best held on the discuss forum under c-api tag [https://discuss.python.org/c/core-dev/c-api/30].

Per-Class Scope

It is currently (as of Python 3.11) not possible to attach state to individual *types* without relying on CPython implementation details (which may change in the future—perhaps, ironically, to allow a proper solution for per-class scope).

Lossless Conversion to Heap Types

The heap type API was not designed for “lossless” conversion from static types; that is, creating a type that works exactly like a given static type.

timer file descriptor HOWTO

Release:: 1.13

This HOWTO discusses Python’s support for the linux timer file descriptor.

Examples

The following example shows how to use a timer file descriptor to execute a function twice a second:

```
# Practical scripts should use really use a non-blocking timer,
# we use a blocking timer here for simplicity.
import os, time

# Create the timer file descriptor
fd = os.timerfd_create(time.CLOCK_REALTIME)

# Start the timer in 1 second, with an interval of half a
second
os.timerfd_settime(fd, initial=1, interval=0.5)

try:
    # Process timer events four times.
    for _ in range(4):
        # read() will block until the timer expires
        _ = os.read(fd, 8)
        print("Timer expired")
finally:
    # Remember to close the timer file descriptor!
    os.close(fd)
```

To avoid the precision loss caused by the `float` type, timer file descriptors allow specifying initial expiration and interval in integer nanoseconds with `_ns` variants of the functions.

This example shows how `epoll()` can be used with timer file descriptors to wait until the file descriptor is ready for reading:

```
import os, time, select, socket, sys

# Create an epoll object
ep = select.epoll()

# In this example, use loopback address to send "stop" command
to the server.
#
# $ telnet 127.0.0.1 1234
# Trying 127.0.0.1...
# Connected to 127.0.0.1.
# Escape character is '^]'.
# stop
# Connection closed by foreign host.
#
sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
sock.bind(("127.0.0.1", 1234))
sock.setblocking(False)
sock.listen(1)
ep.register(sock, select.EPOLLIN)

# Create timer file descriptors in non-blocking mode.
num = 3
fds = []
for _ in range(num):
    fd = os.timerfd_create(time.CLOCK_REALTIME,
flags=os.TFD_NONBLOCK)
    fds.append(fd)
    # Register the timer file descriptor for read events
    ep.register(fd, select.EPOLLIN)

# Start the timer with os.timerfd_settime_ns() in nanoseconds.
# Timer 1 fires every 0.25 seconds; timer 2 every 0.5 seconds;
etc
for i, fd in enumerate(fds, start=1):
    one_sec_in_nsec = 10**9
    i = i * one_sec_in_nsec
    os.timerfd_settime_ns(fd, initial=i//4, interval=i//4)

timeout = 3
try:
    conn = None
    is_active = True
    while is_active:
        # Wait for the timer to expire for 3 seconds.
        # epoll.poll() returns a list of (fd, event) pairs.
        # fd is a file descriptor.
        # sock and conn[=returned value of socket.accept()] are
socket objects, not file descriptors.
        # So use sock.fileno() and conn.fileno() to get the
file descriptors.
        events = ep.poll(timeout)

        # If more than one timer file descriptors are ready for
reading at once,
        # epoll.poll() returns a list of (fd, event) pairs.
        #
        # In this example settings,
        #    1st timer fires every 0.25 seconds in 0.25
seconds. (0.25, 0.5, 0.75, 1.0, ...)
        #    2nd timer every 0.5 seconds in 0.5 seconds. (0.5,
1.0, 1.5, 2.0, ...)
        #    3rd timer every 0.75 seconds in 0.75 seconds.
(0.75, 1.5, 2.25, 3.0, ...)
        #
        #    In 0.25 seconds, only 1st timer fires.
        #    In 0.5 seconds, 1st timer and 2nd timer fires at
once.
        #    In 0.75 seconds, 1st timer and 3rd timer fires at
once.
        #    In 1.5 seconds, 1st timer, 2nd timer and 3rd timer
fires at once.
        #
        # If a timer file descriptor is signaled more than once
since
        # the last os.read() call, os.read() returns the number
of signaled
        # as host order of class bytes.
        print(f"Signaled events={events}")
        for fd, event in events:
            if event & select.EPOLLIN:
                if fd == sock.fileno():
                    # Check if there is a connection request.
                    print(f"Accepting connection {fd}")
                    conn, addr = sock.accept()
                    conn.setblocking(False)
                    print(f"Accepted connection {conn} from
{addr}")
                    ep.register(conn, select.EPOLLIN)
                elif conn and fd == conn.fileno():
                    # Check if there is data to read.
                    print(f"Reading data {fd}")
                    data = conn.recv(1024)
                    if data:
                        # You should catch UnicodeDecodeError
exception for safety.
                        cmd = data.decode()
                        if cmd.startswith("stop"):
                            print(f"Stopping server")
                            is_active = False
                        else:
                            print(f"Unknown command: {cmd}")
                    else:
                        # No more data, close connection
                        print(f"Closing connection {fd}")
                        ep.unregister(conn)
                        conn.close()
                        conn = None
                elif fd in fds:
                    print(f"Reading timer {fd}")
                    count = int.from_bytes(os.read(fd, 8),
byteorder=sys.byteorder)
                    print(f"Timer {fds.index(fd) + 1} expired
{count} times")
                else:
                    print(f"Unknown file descriptor {fd}")
finally:
    for fd in fds:
        ep.unregister(fd)
        os.close(fd)
    ep.close()
```

This example shows how `select()` can be used with timer file descriptors to wait until the file descriptor is ready for reading:

```
import os, time, select, socket, sys

# In this example, use loopback address to send "stop" command
to the server.
#
# $ telnet 127.0.0.1 1234
# Trying 127.0.0.1...
# Connected to 127.0.0.1.
# Escape character is '^]'.
# stop
# Connection closed by foreign host.
#
sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
sock.bind(("127.0.0.1", 1234))
sock.setblocking(False)
sock.listen(1)

# Create timer file descriptors in non-blocking mode.
num = 3
fds = [os.timerfd_create(time.CLOCK_REALTIME,
flags=os.TFD_NONBLOCK)
       for _ in range(num)]
select_fds = fds + [sock]

# Start the timers with os.timerfd_settime() in seconds.
# Timer 1 fires every 0.25 seconds; timer 2 every 0.5 seconds;
etc
for i, fd in enumerate(fds, start=1):
   os.timerfd_settime(fd, initial=i/4, interval=i/4)

timeout = 3
try:
    conn = None
    is_active = True
    while is_active:
       # Wait for the timer to expire for 3 seconds.
       # select.select() returns a list of file descriptors or
objects.
```

```
       rfd, wfd, xfd = select.select(select_fds, select_fds,
select_fds, timeout)
       for fd in rfd:
           if fd == sock:
               # Check if there is a connection request.
               print(f"Accepting connection {fd}")
               conn, addr = sock.accept()
               conn.setblocking(False)
               print(f"Accepted connection {conn} from {addr}")
               select_fds.append(conn)
           elif conn and fd == conn:
               # Check if there is data to read.
               print(f"Reading data {fd}")
               data = conn.recv(1024)
               if data:
                   # You should catch UnicodeDecodeError
exception for safety.
                   cmd = data.decode()
                   if cmd.startswith("stop"):
                       print(f"Stopping server")
                       is_active = False
                   else:
                       print(f"Unknown command: {cmd}")
               else:
                   # No more data, close connection
                   print(f"Closing connection {fd}")
                   select_fds.remove(conn)
                   conn.close()
                   conn = None
           elif fd in fds:
               print(f"Reading timer {fd}")
               count = int.from_bytes(os.read(fd, 8),
byteorder=sys.byteorder)
               print(f"Timer {fds.index(fd) + 1} expired
{count} times")
           else:
               print(f"Unknown file descriptor {fd}")
finally:
    for fd in fds:
       os.close(fd)
    sock.close()
    sock = None
```

The Python 2.3 Method Resolution Order

Note

This is a historical document, provided as an appendix to the official documentation. The Method Resolution Order discussed here was *introduced* in Python 2.3, but it is still used in later versions – including Python 3.

By Michele Simionato [https://www.phyast.pitt.edu/~micheles/].

Abstract:: *This document is intended for Python programmers who want to understand the C3 Method Resolution Order used in Python 2.3. Although it is not intended for newbies, it is quite pedagogical with many worked out examples. I am not aware of other publicly available documents with the same scope, therefore it should be useful.*

Disclaimer:

> *I donate this document to the Python Software Foundation, under the Python 2.3 license. As usual in these circumstances, I warn the reader that what follows* should *be correct, but I don't give any warranty. Use it at your own risk and peril!*

Acknowledgments:

> *All the people of the Python mailing list who sent me their support. Paul Foley who pointed out various imprecisions and made me to add the part on local precedence ordering. David Goodger for help with the formatting in reStructuredText. David Mertz for help with the editing. Finally, Guido van Rossum who enthusiastically added this document to the official Python 2.3 home-page.*

The beginning

> *Felix qui potuit rerum cognoscere causas* – Virgilius

Everything started with a post by Samuele Pedroni to the Python development mailing list [[1]]. In his post, Samuele showed that the Python 2.2 method resolution order is not monotonic and he proposed to replace it with the C3 method resolution order. Guido agreed with his arguments and therefore now Python 2.3 uses C3. The C3 method itself has nothing to do with Python, since it was invented by people working on Dylan and it is described in a paper intended for lispers [[2]]. The present paper gives a (hopefully) readable discussion of the C3 algorithm for Pythonistas who want to understand the reasons for the change.

First of all, let me point out that what I am going to say only applies to the *new style classes* introduced in Python 2.2: *classic classes* maintain their old method resolution order, depth first and then left to right. Therefore, there is no breaking of old code for classic classes; and even if in principle there could be breaking of code for Python 2.2 new style classes, in practice the cases in which the C3 resolution order differs from the Python 2.2 method resolution order are so rare that no real breaking of code is expected. Therefore:

> *Don’t be scared!*

Moreover, unless you make strong use of multiple inheritance and you have non-trivial hierarchies, you don’t need to understand the C3 algorithm, and you can easily skip this paper. On the other hand, if you really want to know how multiple inheritance works, then this paper is for you. The good news is that things are not as complicated as you might expect.

Let me begin with some basic definitions.

1. Given a class C in a complicated multiple inheritance hierarchy, it is a non-trivial task to specify the order in which methods are overridden, i.e. to specify the order of the ancestors of C.
2. The list of the ancestors of a class C, including the class itself, ordered from the nearest ancestor to the furthest, is called the class precedence list or the *linearization* of C.
3. The *Method Resolution Order* (MRO) is the set of rules that construct the linearization. In the Python literature, the idiom “the MRO of C” is also used as a synonymous for the linearization of the class C.
4. For instance, in the case of single inheritance hierarchy, if C is a subclass of C1, and C1 is a subclass of C2, then the linearization of C is simply the list [C, C1 , C2]. However, with multiple inheritance hierarchies, the construction of the linearization is more cumbersome, since it is more difficult to construct a linearization that respects *local precedence ordering* and *monotonicity*.
5. I will discuss the local precedence ordering later, but I can give the definition of monotonicity here. A MRO is

monotonic when the following is true: *if C1 precedes C2 in the linearization of C, then C1 precedes C2 in the linearization of any subclass of C*. Otherwise, the innocuous operation of deriving a new class could change the resolution order of methods, potentially introducing very subtle bugs. Examples where this happens will be shown later.

6. Not all classes admit a linearization. There are cases, in complicated hierarchies, where it is not possible to derive a class such that its linearization respects all the desired properties.

Here I give an example of this situation. Consider the hierarchy

```
>>> O = object
>>> class X(O): pass
>>> class Y(O): pass
>>> class A(X,Y): pass
>>> class B(Y,X): pass
```

which can be represented with the following inheritance graph, where I have denoted with O the `object` class, which is the beginning of any hierarchy for new style classes:

```
   -----------
  |           |
  |    O      |
  |  /   \    |
   - X    Y  /
     |  / | /
     | /  |/
     A    B
     \   /
       ?
```

In this case, it is not possible to derive a new class C from A and B, since X precedes Y in A, but Y precedes X in B, therefore the method resolution order would be ambiguous in C.

Python 2.3 raises an exception in this situation (TypeError: MRO conflict among bases Y, X) forbidding the naive programmer from creating ambiguous hierarchies. Python 2.2 instead does not raise an exception, but chooses an *ad hoc* ordering (CABXYO in this case).

The C3 Method Resolution Order

Let me introduce a few simple notations which will be useful for the following discussion. I will use the shortcut notation:

```
C1 C2 ... CN
```

to indicate the list of classes [C1, C2, … , CN].

The *head* of the list is its first element:

```
head = C1
```

whereas the *tail* is the rest of the list:

```
tail = C2 ... CN.
```

I shall also use the notation:

```
C + (C1 C2 ... CN) = C C1 C2 ... CN
```

to denote the sum of the lists [C] + [C1, C2, … ,CN].

Now I can explain how the MRO works in Python 2.3.

Consider a class C in a multiple inheritance hierarchy, with C inheriting from the base classes B1, B2, … , BN. We want to compute the linearization L[C] of the class C. The rule is the following:

> *the linearization of C is the sum of C plus the merge of the linearizations of the parents and the list of the parents.*

In symbolic notation:

```
L[C(B1 ... BN)] = C + merge(L[B1] ... L[BN], B1 ... BN)
```

In particular, if C is the `object` class, which has no parents, the linearization is trivial:

```
L[object] = object.
```

However, in general one has to compute the merge according to the following prescription:

> *take the head of the first list, i.e L[B1][0]; if this head is not in the tail of any of the other lists, then add it to the linearization of C and remove it from the lists in the merge, otherwise look at the head of the next list and take it, if it is a good head. Then repeat the operation until all the class are removed or it is impossible to find good heads. In this case, it is impossible to construct the merge, Python 2.3 will refuse to create the class C and will raise an exception.*

This prescription ensures that the merge operation *preserves* the ordering, if the ordering can be preserved. On the other hand, if the order cannot be preserved (as in the example of serious order disagreement discussed above) then the merge cannot be computed.

The computation of the merge is trivial if C has only one parent (single inheritance); in this case:

```
L[C(B)] = C + merge(L[B],B) = C + L[B]
```

However, in the case of multiple inheritance things are more cumbersome and I don’t expect you can understand the rule without a couple of examples ;-)

Examples

First example. Consider the following hierarchy:

```
>>> O = object
>>> class F(O): pass
>>> class E(O): pass
>>> class D(O): pass
>>> class C(D,F): pass
>>> class B(D,E): pass
>>> class A(B,C): pass
```

In this case the inheritance graph can be drawn as:

```
                            6
                           ---
  Level 3                 | O |                  (more
  general)
                        /  ---  \
                       /    |    \                      |
                      /     |     \                     |
                     /      |      \                    |
                    ---    ---    ---                   |
  Level 2        3 | D | 4| E |  | F | 5                |
                    ---    ---    ---                   |
                     \  \ _ /       |                   |
                      \    / \ _    |                   |
                       \  /      \  |                   |
                        ---      ---                    |
  Level 1            1 | B |    | C | 2                 |
                        ---      ---                    |
                          \      /                      |
                           \    /                      \ /
                             ---
  Level 0                 0 | A |                (more
  specialized)
                             ---
```

The linearizations of O,D,E and F are trivial:

```
L[O] = O
L[D] = D O
L[E] = E O
L[F] = F O
```

The linearization of B can be computed as:

```
L[B] = B + merge(DO, EO, DE)
```

We see that D is a good head, therefore we take it and we are reduced to compute `merge(O,EO,E)`. Now O is not a good head, since it is in the tail of the sequence EO. In this case the rule says that we have to skip to the next sequence. Then we see that E is a good head; we take it and we are reduced to compute `merge(O,O)` which gives O. Therefore:

```
L[B] =  B D E O
```

Using the same procedure one finds:

```
L[C] = C + merge(DO,FO,DF)
     = C + D + merge(O,FO,F)
     = C + D + F + merge(O,O)
     = C D F O
```

Now we can compute:

```
L[A] = A + merge(BDEO,CDFO,BC)
     = A + B + merge(DEO,CDFO,C)
     = A + B + C + merge(DEO,DFO)
     = A + B + C + D + merge(EO,FO)
     = A + B + C + D + E + merge(O,FO)
     = A + B + C + D + E + F + merge(O,O)
     = A B C D E F O
```

In this example, the linearization is ordered in a pretty nice way according to the inheritance level, in the sense that lower levels (i.e. more specialized classes) have higher precedence (see the inheritance graph). However, this is not the general case.

I leave as an exercise for the reader to compute the linearization for my second example:

```
>>> O = object
>>> class F(O): pass
>>> class E(O): pass
>>> class D(O): pass
>>> class C(D,F): pass
>>> class B(E,D): pass
>>> class A(B,C): pass
```

The only difference with the previous example is the change B(D,E) –> B(E,D); however even such a little modification completely changes the ordering of the hierarchy:

```
                             6
                            ---
  Level 3                  | O |
                         /  ---  \
                        /    |    \
                       /     |     \
                      /      |      \
                    ---     ---    ---
  Level 2        2 | E | 4 | D |  | F | 5
                    ---     ---    ---
                     \      / \     /
                      \    /   \   /
                       \  /     \ /
                        ---     ---
  Level 1            1 | B |   | C | 3
                        ---     ---
                         \       /
                          \     /
                            ---
  Level 0                0 | A |
                            ---
```

Notice that the class E, which is in the second level of the hierarchy, precedes the class C, which is in the first level of the hierarchy, i.e. E is more specialized than C, even if it is in a higher level.

A lazy programmer can obtain the MRO directly from Python 2.2, since in this case it coincides with the Python 2.3 linearization. It is enough to invoke the `mro()` method of class A:

```
>>> A.mro()
[<class 'A'>, <class 'B'>, <class 'E'>,
<class 'C'>, <class 'D'>, <class 'F'>,
<class 'object'>]
```

Finally, let me consider the example discussed in the first section, involving a serious order disagreement. In this case, it is straightforward to compute the linearizations of O, X, Y, A and B:

```
  L[O] = 0
  L[X] = X O
  L[Y] = Y O
  L[A] = A X Y O
  L[B] = B Y X O
```

However, it is impossible to compute the linearization for a class C that inherits from A and B:

```
L[C] = C + merge(AXYO, BYXO, AB)
     = C + A + merge(XYO, BYXO, B)
     = C + A + B + merge(XYO, YXO)
```

At this point we cannot merge the lists XYO and YXO, since X is in the tail of YXO whereas Y is in the tail of XYO: therefore there are no good heads and the C3 algorithm stops. Python 2.3 raises an error and refuses to create the class C.

Bad Method Resolution Orders

A MRO is *bad* when it breaks such fundamental properties as local precedence ordering and monotonicity. In this section, I will show that both the MRO for classic classes and the MRO for new style classes in Python 2.2 are bad.

It is easier to start with the local precedence ordering. Consider the following example:

```
>>> F=type('Food',(),{'remember2buy':'spam'})
>>> E=type('Eggs',(F,),{'remember2buy':'eggs'})
>>> G=type('GoodFood',(F,E),{}) # under Python 2.3 this is an
error!
```

with inheritance diagram

```
               O
               |
  (buy spam)   F
               | \
               | E   (buy eggs)
               | /
               G

        (buy eggs or spam ?)
```

We see that class G inherits from F and E, with F *before* E: therefore we would expect the attribute *G.remember2buy* to be inherited by *F.remember2buy* and not by *E.remember2buy*: nevertheless Python 2.2 gives

```
>>> G.remember2buy
'eggs'
```

This is a breaking of local precedence ordering since the order in the local precedence list, i.e. the list of the parents of G, is not preserved in the Python 2.2 linearization of G:

```
L[G,P22]= G E F object   # F *follows* E
```

One could argue that the reason why F follows E in the Python 2.2 linearization is that F is less specialized than E, since F is the superclass of E; nevertheless the breaking of local precedence ordering is quite non-intuitive and error prone. This is particularly true since it is a different from old style classes:

```
>>> class F: remember2buy='spam'
>>> class E(F): remember2buy='eggs'
>>> class G(F,E): pass
>>> G.remember2buy
'spam'
```

In this case the MRO is GFEF and the local precedence ordering is preserved.

As a general rule, hierarchies such as the previous one should be avoided, since it is unclear if F should override E or vice-versa. Python 2.3 solves the ambiguity by raising an exception in the creation of class G, effectively stopping the programmer from generating ambiguous hierarchies. The reason for that is that the C3 algorithm fails when the merge:

```
merge(FO,EFO,FE)
```

cannot be computed, because F is in the tail of EFO and E is in the tail of FE.

The real solution is to design a non-ambiguous hierarchy, i.e. to derive G from E and F (the more specific first) and not from F and E; in this case the MRO is GEF without any doubt.

```
             O
             |
             F (spam)
           / |
  (eggs)   E |
           \ |
             G
               (eggs, no doubt)
```

Python 2.3 forces the programmer to write good hierarchies (or, at least, less error-prone ones).

On a related note, let me point out that the Python 2.3 algorithm is smart enough to recognize obvious mistakes, as the duplication of classes in the list of parents:

```
>>> class A(object): pass
>>> class C(A,A): pass # error
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
TypeError: duplicate base class A
```

Python 2.2 (both for classic classes and new style classes) in this situation, would not raise any exception.

Finally, I would like to point out two lessons we have learned from this example:

1. despite the name, the MRO determines the resolution order of attributes, not only of methods;
2. the default food for Pythonistas is spam ! (but you already knew that ;-)

Having discussed the issue of local precedence ordering, let me now consider the issue of monotonicity. My goal is to show that neither the MRO for classic classes nor that for Python 2.2 new style classes is monotonic.

To prove that the MRO for classic classes is non-monotonic is rather trivial, it is enough to look at the diamond diagram:

```
     C
    / \
   /   \
  A     B
   \   /
    \ /
     D
```

One easily discerns the inconsistency:

```
L[B,P21] = B C        # B precedes C : B's methods win
L[D,P21] = D A C B C  # B follows C  : C's methods win!
```

On the other hand, there are no problems with the Python 2.2 and 2.3 MROs, they give both:

```
L[D] = D A B C
```

Guido points out in his essay [[3]] that the classic MRO is not so bad in practice, since one can typically avoids diamonds for classic classes. But all new style classes inherit from `object`, therefore diamonds are unavoidable and inconsistencies shows up in every multiple inheritance graph.

The MRO of Python 2.2 makes breaking monotonicity difficult, but not impossible. The following example, originally provided by Samuele Pedroni, shows that the MRO of Python 2.2 is non-monotonic:

```
>>> class A(object): pass
>>> class B(object): pass
>>> class C(object): pass
>>> class D(object): pass
>>> class E(object): pass
>>> class K1(A,B,C): pass
>>> class K2(D,B,E): pass
>>> class K3(D,A):   pass
>>> class Z(K1,K2,K3): pass
```

Here are the linearizations according to the C3 MRO (the reader should verify these linearizations as an exercise and draw the inheritance diagram ;-)

```
L[A] = A O
L[B] = B O
L[C] = C O
L[D] = D O
L[E] = E O
L[K1]= K1 A B C O
L[K2]= K2 D B E O
L[K3]= K3 D A O
L[Z] = Z K1 K2 K3 D A B C E O
```

Python 2.2 gives exactly the same linearizations for A, B, C, D, E, K1, K2 and K3, but a different linearization for Z:

```
L[Z,P22] = Z K1 K3 A K2 D B C E O
```

It is clear that this linearization is *wrong*, since A comes before D whereas in the linearization of K3 A comes *after* D. In other words, in K3 methods derived by D override methods derived by A, but in Z, which still is a subclass of K3, methods derived by A override methods derived by D! This is a violation of monotonicity. Moreover, the Python 2.2 linearization of Z is also inconsistent with local precedence ordering, since the local precedence list of the class Z is [K1, K2, K3] (K2 precedes K3), whereas in the linearization of Z K2 *follows* K3. These problems explain why the 2.2 rule has been dismissed in favor of the C3 rule.

The end

This section is for the impatient reader, who skipped all the previous sections and jumped immediately to the end. This section is for the lazy programmer too, who didn’t want to exercise her/his brain. Finally, it is for the programmer with some hubris, otherwise s/he would not be reading a paper on the C3 method resolution order in multiple inheritance hierarchies ;-) These three virtues taken all together (and *not* separately) deserve a prize: the prize is a short Python 2.2 script that allows you to compute the 2.3 MRO without risk to your brain. Simply change the last line to play with the various examples I have discussed in this paper.:

```
#<mro.py>

"""C3 algorithm by Samuele Pedroni (with readability enhanced
by me)."""

class __metaclass__(type):
    "All classes are metamagically modified to be nicely
printed"
    __repr__ = lambda cls: cls.__name__

class ex_2:
    "Serious order disagreement" #From Guido
    class O: pass
    class X(O): pass
    class Y(O): pass
    class A(X,Y): pass
    class B(Y,X): pass
    try:
        class Z(A,B): pass #creates Z(A,B) in Python 2.2
    except TypeError:
        pass # Z(A,B) cannot be created in Python 2.3

class ex_5:
    "My first example"
    class O: pass
    class F(O): pass
    class E(O): pass
    class D(O): pass
    class C(D,F): pass
    class B(D,E): pass
    class A(B,C): pass

class ex_6:
    "My second example"
    class O: pass
    class F(O): pass
    class E(O): pass
    class D(O): pass
    class C(D,F): pass
    class B(E,D): pass
    class A(B,C): pass

class ex_9:
    "Difference between Python 2.2 MRO and C3" #From Samuele
    class O: pass
    class A(O): pass
    class B(O): pass
    class C(O): pass
    class D(O): pass
    class E(O): pass
    class K1(A,B,C): pass
    class K2(D,B,E): pass
    class K3(D,A): pass
    class Z(K1,K2,K3): pass

def merge(seqs):
    print '\n\nCPL[%s]=%s' % (seqs[0][0],seqs),
    res = []; i=0
    while 1:
      nonemptyseqs=[seq for seq in seqs if seq]
      if not nonemptyseqs: return res
      i+=1; print '\n',i,'round: candidates...',
      for seq in nonemptyseqs: # find merge candidates among
seq heads
          cand = seq[0]; print ' ',cand,
          nothead=[s for s in nonemptyseqs if cand in s[1:]]
          if nothead: cand=None #reject candidate
          else: break
      if not cand: raise "Inconsistent hierarchy"
      res.append(cand)
      for seq in nonemptyseqs: # remove cand
          if seq[0] == cand: del seq[0]

def mro(C):
    "Compute the class precedence list (mro) according to C3"
    return merge([[C]]+map(mro,C.__bases__)+
[list(C.__bases__)])

def print_mro(C):
    print '\nMRO[%s]=%s' % (C,mro(C))
    print '\nP22 MRO[%s]=%s' % (C,C.mro())

print_mro(ex_9.Z)

#</mro.py>
```

That’s all folks,

enjoy !

Resources

[1]

The thread on python-dev started by Samuele Pedroni: https://mail.python.org/pipermail/python-dev/2002-October/029035.html

[2]

The paper *A Monotonic Superclass Linearization for Dylan*: https://doi.org/10.1145/236337.236343

[3]

Guido van Rossum’s essay, *Unifying types and classes in Python 2.2*: https://web.archive.org/web/20140210194412/http://www.python.org/download/releases/2.2.2/descrintro

Python support for free threading

Starting with the 3.13 release, CPython has support for a build of Python called free threading where the global interpreter lock (GIL) is disabled. Free-threaded execution allows for full utilization of the available processing power by running threads in parallel on available CPU cores. While not all software will benefit from this automatically, programs designed with threading in mind will run faster on multi-core hardware.

Some third-party packages, in particular ones with an extension module, may not be ready for use in a free-threaded build, and will re-enable the GIL.

This document describes the implications of free threading for Python code. See C API Extension Support for Free Threading for information on how to write C extensions that support the free-threaded build.

See also

PEP 703 [https://peps.python.org/pep-0703/] – Making the Global Interpreter Lock Optional in CPython for an overall description of free-threaded Python.

Installation

Starting with Python 3.13, the official macOS and Windows installers optionally support installing free-threaded Python binaries. The installers are available at https://www.python.org/downloads/.

For information on other platforms, see the Installing a Free-Threaded Python [https://py-free-threading.github.io/installing-cpython/], a community-maintained installation guide for installing free-threaded Python.

When building CPython from source, the `--disable-gil` configure option should be used to build a free-threaded Python interpreter.

Identifying free-threaded Python

To check if the current interpreter supports free-threading, `python -VV` and `sys.version` contain “free-threading build”. The new `sys._is_gil_enabled()` function can be used to check whether the GIL is actually disabled in the running process.

The `sysconfig.get_config_var("Py_GIL_DISABLED")` configuration variable can be used to determine whether the build supports free threading. If the variable is set to `1`, then the build supports free threading. This is the recommended mechanism for decisions related to the build configuration.

The global interpreter lock in free-threaded Python

Free-threaded builds of CPython support optionally running with the GIL enabled at runtime using the environment variable `PYTHON_GIL` or the command-line option `-X gil`.

The GIL may also automatically be enabled when importing a C-API extension module that is not explicitly marked as supporting free threading. A warning will be printed in this case.

In addition to individual package documentation, the following websites track the status of popular packages support for free threading:

- https://py-free-threading.github.io/tracking/
- https://hugovk.github.io/free-threaded-wheels/

Thread safety

The free-threaded build of CPython aims to provide similar thread-safety behavior at the Python level to the default GIL-enabled build. Built-in types like `dict`, `list`, and `set` use internal locks to protect against concurrent modifications in ways that behave similarly to the GIL. However, Python has not historically guaranteed specific behavior for concurrent modifications to these built-in types, so this should be treated as a description of the current implementation, not a guarantee of current or future behavior.

Note

It’s recommended to use the `threading.Lock` or other synchronization primitives instead of relying on the internal locks of built-in types, when possible.

Known limitations

This section describes known limitations of the free-threaded CPython build.

Immortalization

In the free-threaded build, some objects are immortal. Immortal objects are not deallocated and have reference counts that are never modified. This is done to avoid reference count contention that would prevent efficient multi-threaded scaling.

As of the 3.14 release, immortalization is limited to:

- Code constants: numeric literals, string literals, and tuple literals composed of other constants.
- Strings interned by `sys.intern()`.

Frame objects

It is not safe to access `frame.f_locals` from a frame object if that frame is currently executing in another thread, and doing so may crash the interpreter.

Iterators

It is generally not thread-safe to access the same iterator object from multiple threads concurrently, and threads may see duplicate or missing elements.

Single-threaded performance

The free-threaded build has additional overhead when executing Python code compared to the default GIL-enabled build. The amount of overhead depends on the workload and hardware. On the pyperformance benchmark suite, the average overhead ranges from about 1% on macOS aarch64 to 8% on x86-64 Linux systems.

Behavioral changes

This section describes CPython behavioural changes with the free-threaded build.

Context variables

In the free-threaded build, the flag `thread_inherit_context` is set to true by default which causes threads created with `threading.Thread` to start with a copy of the `Context()` of the caller of `start()`. In the default GIL-enabled build, the flag defaults to false so threads start with an empty `Context()`.

Warning filters

In the free-threaded build, the flag `context_aware_warnings` is set to true by default. In the default GIL-enabled build, the flag defaults to false. If the flag is true then the `warnings.catch_warnings` context manager uses a context variable for warning filters. If the flag is false then `catch_warnings` modifies the global filters list, which is not thread-safe. See the `warnings` module for more details.

C API Extension Support for Free Threading

Starting with the 3.13 release, CPython has support for running with the global interpreter lock (GIL) disabled in a configuration called free threading. This document describes how to adapt C API extensions to support free threading.

Identifying the Free-Threaded Build in C

The CPython C API exposes the `Py_GIL_DISABLED` macro: in the free-threaded build it's defined to `1`, and in the regular build it's not defined. You can use it to enable code that only runs under the free-threaded build:

```
#ifdef Py_GIL_DISABLED
/* code that only runs in the free-threaded build */
#endif
```

Note

On Windows, this macro is not defined automatically, but must be specified to the compiler when building. The `sysconfig.get_config_var()` function can be used to determine whether the current running interpreter had the macro defined.

Module Initialization

Extension modules need to explicitly indicate that they support running with the GIL disabled; otherwise importing the extension will raise a warning and enable the GIL at runtime.

There are two ways to indicate that an extension module supports running with the GIL disabled depending on whether the extension uses multi-phase or single-phase initialization.

Multi-Phase Initialization

Extensions that use multi-phase initialization (i.e., `PyModuleDef_Init()`) should add a `Py_mod_gil` slot in the module definition. If your extension supports older versions of CPython, you should guard the slot with a `PY_VERSION_HEX` check.

```
static struct PyModuleDef_Slot module_slots[] = {
    ...
#if PY_VERSION_HEX >= 0x030D0000
    {Py_mod_gil, Py_MOD_GIL_NOT_USED},
#endif
    {0, NULL}
};

static struct PyModuleDef moduledef = {
    PyModuleDef_HEAD_INIT,
    .m_slots = module_slots,
    ...
};
```

Single-Phase Initialization

Extensions that use single-phase initialization (i.e., `PyModule_Create()`) should call `PyUnstable_Module_SetGIL()` to indicate that they support running with the GIL disabled. The function is only defined in the free-threaded build, so you should guard the call with `#ifdef Py_GIL_DISABLED` to avoid compilation errors in the regular build.

```
static struct PyModuleDef moduledef = {
    PyModuleDef_HEAD_INIT,
    ...
};

PyMODINIT_FUNC
PyInit_mymodule(void)
{
    PyObject *m = PyModule_Create(&moduledef);
    if (m == NULL) {
        return NULL;
    }
#ifdef Py_GIL_DISABLED
    PyUnstable_Module_SetGIL(m, Py_MOD_GIL_NOT_USED);
#endif
    return m;
}
```

General API Guidelines

Most of the C API is thread-safe, but there are some exceptions.

- **Struct Fields**: Accessing fields in Python C API objects or structs directly is not thread-safe if the field may be concurrently modified.
- **Macros**: Accessor macros like `PyList_GET_ITEM`, `PyList_SET_ITEM`, and macros like `PySequence_Fast_GET_SIZE` that use the object returned by `PySequence_Fast()` do not perform any error checking or locking. These macros are not thread-safe if the container object may be modified concurrently.
- **Borrowed References**: C API functions that return borrowed references may not be thread-safe if the

containing object is modified concurrently. See the section on borrowed references for more information.

Container Thread Safety

Containers like `PyListObject`, `PyDictObject`, and `PySetObject` perform internal locking in the free-threaded build. For example, the `PyList_Append()` will lock the list before appending an item.

PyDict_Next

A notable exception is `PyDict_Next()`, which does not lock the dictionary. You should use `Py_BEGIN_CRITICAL_SECTION` to protect the dictionary while iterating over it if the dictionary may be concurrently modified:

```
Py_BEGIN_CRITICAL_SECTION(dict);
PyObject *key, *value;
Py_ssize_t pos = 0;
while (PyDict_Next(dict, &pos, &key, &value)) {
    ...
}
Py_END_CRITICAL_SECTION();
```

Borrowed References

Some C API functions return borrowed references. These APIs are not thread-safe if the containing object is modified concurrently. For example, it’s not safe to use `PyList_GetItem()` if the list may be modified concurrently.

The following table lists some borrowed reference APIs and their replacements that return strong references.

Borrowed reference API	Strong reference API
`PyList_GetItem()`	`PyList_GetItemRef()`
`PyList_GET_ITEM()`	`PyList_GetItemRef()`
`PyDict_GetItem()`	`PyDict_GetItemRef()`
`PyDict_GetItemWithError()`	`PyDict_GetItemRef()`
`PyDict_GetItemString()`	`PyDict_GetItemStringRef()`
`PyDict_SetDefault()`	`PyDict_SetDefaultRef()`
`PyDict_Next()`	none (see PyDict_Next)
`PyWeakref_GetObject()`	`PyWeakref_GetRef()`
`PyWeakref_GET_OBJECT()`	`PyWeakref_GetRef()`
`PyImport_AddModule()`	`PyImport_AddModuleRef()`
`PyCell_GET()`	`PyCell_Get()`

Not all APIs that return borrowed references are problematic. For example, `PyTuple_GetItem()` is safe because tuples are immutable. Similarly, not all uses of the above APIs are problematic. For example, `PyDict_GetItem()` is often used for parsing keyword argument dictionaries in function calls; those keyword argument dictionaries are effectively private (not accessible by other threads), so using borrowed references in that context is safe.

Some of these functions were added in Python 3.13. You can use the pythoncapi-compat [https://github.com/python/pythoncapi-compat] package to provide implementations of these functions for older Python versions.

Memory Allocation APIs

Python’s memory management C API provides functions in three different allocation domains: “raw”, “mem”, and “object”. For thread-safety, the free-threaded build requires that only Python objects are allocated using the object domain, and that all Python objects are allocated using that domain. This differs from the prior Python versions, where this was only a best practice and not a hard requirement.

Note

Search for uses of `PyObject_Malloc()` in your extension and check that the allocated memory is used for Python objects. Use `PyMem_Malloc()` to allocate buffers instead of `PyObject_Malloc()`.

Thread State and GIL APIs

Python provides a set of functions and macros to manage thread state and the GIL, such as:

- `PyGILState_Ensure()` and `PyGILState_Release()`
- `PyEval_SaveThread()` and `PyEval_RestoreThread()`
- `Py_BEGIN_ALLOW_THREADS` and `Py_END_ALLOW_THREADS`

These functions should still be used in the free-threaded build to manage thread state even when the GIL is disabled. For example, if you create a thread outside of Python, you must call `PyGILState_Ensure()` before calling into the Python API to ensure that the thread has a valid Python thread state.

You should continue to call `PyEval_SaveThread()` or `Py_BEGIN_ALLOW_THREADS` around blocking operations, such as I/O or lock acquisitions, to allow other threads to run the cyclic garbage collector.

Protecting Internal Extension State

Your extension may have internal state that was previously protected by the GIL. You may need to add locking to protect this state. The approach will depend on your extension, but some common patterns include:

- **Caches**: global caches are a common source of shared state. Consider using a lock to protect the cache or disabling it in the free-threaded build if the cache is not critical for performance.
- **Global State**: global state may need to be protected by a lock or moved to thread local storage. C11 and C++11 provide the `thread_local` or `_Thread_local` for thread-local storage [https://en.cppreference.com/w/c/language/storage_duration].

Critical Sections

In the free-threaded build, CPython provides a mechanism called “critical sections” to protect data that would otherwise be protected by the GIL. While extension authors may not interact with the internal critical section implementation directly, understanding their behavior is crucial when using certain C API functions or managing shared state in the free-threaded build.

What Are Critical Sections?

Conceptually, critical sections act as a deadlock avoidance layer built on top of simple mutexes. Each thread maintains a stack of active critical sections. When a thread needs to acquire a lock associated with a critical section (e.g., implicitly when calling a thread-safe C API function like `PyDict_SetItem()`, or explicitly using macros), it attempts to acquire the underlying mutex.

Using Critical Sections

The primary APIs for using critical sections are:

- `Py_BEGIN_CRITICAL_SECTION` and `Py_END_CRITICAL_SECTION` - For locking a single object
- `Py_BEGIN_CRITICAL_SECTION2` and `Py_END_CRITICAL_SECTION2` - For locking two objects simultaneously

These macros must be used in matching pairs and must appear in the same C scope, since they establish a new local scope. These macros are no-ops in non-free-threaded builds, so they can be safely added to code that needs to support both build types.

A common use of a critical section would be to lock an object while accessing an internal attribute of it. For example, if an extension type has an internal count field, you could use a critical section while reading or writing that field:

```
// read the count, returns new reference to internal count
value
PyObject *result;
Py_BEGIN_CRITICAL_SECTION(obj);
result = Py_NewRef(obj->count);
Py_END_CRITICAL_SECTION();
return result;

// write the count, consumes reference from new_count
Py_BEGIN_CRITICAL_SECTION(obj);
obj->count = new_count;
Py_END_CRITICAL_SECTION();
```

How Critical Sections Work

Unlike traditional locks, critical sections do not guarantee exclusive access throughout their entire duration. If a thread would block while holding a critical section (e.g., by acquiring another lock or performing I/O), the critical section is temporarily suspended—all locks are released—and then resumed when the blocking operation completes.

This behavior is similar to what happens with the GIL when a thread makes a blocking call. The key differences are:

- Critical sections operate on a per-object basis rather than globally
- Critical sections follow a stack discipline within each thread (the “begin” and “end” macros enforce this since they must be paired and within the same scope)
- Critical sections automatically release and reacquire locks around potential blocking operations

Deadlock Avoidance

Critical sections help avoid deadlocks in two ways:

1. If a thread tries to acquire a lock that's already held by another thread, it first suspends all of its active critical sections, temporarily releasing their locks
2. When the blocking operation completes, only the top-most critical section is reacquired first

This means you cannot rely on nested critical sections to lock multiple objects at once, as the inner critical section may suspend the outer ones. Instead, use `Py_BEGIN_CRITICAL_SECTION2` to lock two objects simultaneously.

Note that the locks described above are only `PyMutex` based locks. The critical section implementation does not know about or affect other locking mechanisms that might be in use, like POSIX mutexes. Also note that while blocking on any `PyMutex` causes the critical sections to be suspended, only the mutexes that are part of the critical sections are released. If `PyMutex` is used without a critical section, it will not be released and therefore does not get the same deadlock avoidance.

Important Considerations

- Critical sections may temporarily release their locks, allowing other threads to modify the protected data. Be careful about making assumptions about the state of the data after operations that might block.
- Because locks can be temporarily released (suspended), entering a critical section does not guarantee exclusive access to the protected resource throughout the section's duration. If code within a critical section calls another function that blocks (e.g., acquires another lock,

performs blocking I/O), all locks held by the thread via critical sections will be released. This is similar to how the GIL can be released during blocking calls.
- Only the lock(s) associated with the most recently entered (top-most) critical section are guaranteed to be held at any given time. Locks for outer, nested critical sections might have been suspended.
- You can lock at most two objects simultaneously with these APIs. If you need to lock more objects, you'll need to restructure your code.
- While critical sections will not deadlock if you attempt to lock the same object twice, they are less efficient than purpose-built reentrant locks for this use case.
- When using **`Py_BEGIN_CRITICAL_SECTION2`**, the order of the objects doesn't affect correctness (the implementation handles deadlock avoidance), but it's good practice to always lock objects in a consistent order.
- Remember that the critical section macros are primarily for protecting access to *Python objects* that might be involved in internal CPython operations susceptible to the deadlock scenarios described above. For protecting purely internal extension state, standard mutexes or other synchronization primitives might be more appropriate.

Building Extensions for the Free-Threaded Build

C API extensions need to be built specifically for the free-threaded build. The wheels, shared libraries, and binaries are indicated by a `t` suffix.

- pypa/manylinux [https://github.com/pypa/manylinux] supports the free-threaded build, with the `t` suffix, such as `python3.13t`.

- pypa/cibuildwheel [https://github.com/pypa/cibuildwheel] supports the free-threaded build on Python 3.13 and 3.14. On Python 3.14, free-threaded wheels will be built by default. On Python 3.13, you will need to set CIBW_ENABLE to cpython-freethreading [https://cibuildwheel.pypa.io/en/stable/options/#enable].

Limited C API and Stable ABI

The free-threaded build does not currently support the Limited C API or the stable ABI. If you use setuptools [https://setuptools.pypa.io/en/latest/setuptools.html] to build your extension and currently set `py_limited_api=True` you can use `py_limited_api=not sysconfig.get_config_var("Py_GIL_DISABLED")` to opt out of the limited API when building with the free-threaded build.

Note

You will need to build separate wheels specifically for the free-threaded build. If you currently use the stable ABI, you can continue to build a single wheel for multiple non-free-threaded Python versions.

Windows

Due to a limitation of the official Windows installer, you will need to manually define `Py_GIL_DISABLED=1` when building extensions from source.

See also

Porting Extension Modules to Support Free-Threading [https://py-free-threading.github.io/porting/]: A community-maintained porting guide for extension authors.

Remote debugging attachment protocol

This protocol enables external tools to attach to a running CPython process and execute Python code remotely.

Most platforms require elevated privileges to attach to another Python process.

Permission requirements

Attaching to a running Python process for remote debugging requires elevated privileges on most platforms. The specific requirements and troubleshooting steps depend on your operating system:

Linux

The tracer process must have the `CAP_SYS_PTRACE` capability or equivalent privileges. You can only trace processes you own and can signal. Tracing may fail if the process is already being traced, or if it is running with set-user-ID or set-group-ID. Security modules like Yama may further restrict tracing.

To temporarily relax ptrace restrictions (until reboot), run:

```
echo 0 | sudo tee /proc/sys/kernel/yama/ptrace_scope
```

Note

Disabling `ptrace_scope` reduces system hardening and should only be done in trusted environments.

If running inside a container, use `--cap-add=SYS_PTRACE` or `--privileged`, and run as root if needed.

Try re-running the command with elevated privileges:

```
sudo -E !!
```

macOS

To attach to another process, you typically need to run your debugging tool with elevated privileges. This can be done by using `sudo` or running as root.

Even when attaching to processes you own, macOS may block debugging unless the debugger is run with root privileges due to system security restrictions.

Windows

To attach to another process, you usually need to run your debugging tool with administrative privileges. Start the command prompt or terminal as Administrator.

Some processes may still be inaccessible even with Administrator rights, unless you have the `SeDebugPrivilege` privilege enabled.

To resolve file or folder access issues, adjust the security permissions:

1. Right-click the file or folder and select **Properties**.

2. Go to the **Security** tab to view users and groups with access.
3. Click **Edit** to modify permissions.
4. Select your user account.
5. In **Permissions**, check **Read** or **Full control** as needed.
6. Click **Apply**, then **OK** to confirm.

Note

Ensure you’ve satisfied all Permission requirements before proceeding.

This section describes the low-level protocol that enables external tools to inject and execute a Python script within a running CPython process.

This mechanism forms the basis of the `sys.remote_exec()` function, which instructs a remote Python process to execute a `.py` file. However, this section does not document the usage of that function. Instead, it provides a detailed explanation of the underlying protocol, which takes as input the `pid` of a target Python process and the path to a Python source file to be executed. This information supports independent reimplementation of the protocol, regardless of programming language.

Warning

The execution of the injected script depends on the interpreter reaching a safe evaluation point. As a result, execution may be delayed depending on the runtime state of the target process.

Once injected, the script is executed by the interpreter within the target process the next time a safe evaluation point is reached. This approach enables remote execution capabilities without modifying the behavior or structure of the running Python application.

Subsequent sections provide a step-by-step description of the protocol, including techniques for locating interpreter structures in memory, safely accessing internal fields, and triggering code execution. Platform-specific variations are noted where applicable, and example implementations are included to clarify each operation.

Locating the PyRuntime structure

CPython places the `PyRuntime` structure in a dedicated binary section to help external tools find it at runtime. The name and format of this section vary by platform. For example, `.PyRuntime` is used on ELF systems, and `__DATA,__PyRuntime` is used on macOS. Tools can find the offset of this structure by examining the binary on disk.

The `PyRuntime` structure contains CPython’s global interpreter state and provides access to other internal data, including the list of interpreters, thread states, and debugger support fields.

To work with a remote Python process, a debugger must first find the memory address of the `PyRuntime` structure in the target process. This address can’t be hardcoded or calculated from a symbol name, because it depends on where the operating system loaded the binary.

The method for finding `PyRuntime` depends on the platform, but the steps are the same in general:

1. Find the base address where the Python binary or shared library was loaded in the target process.
2. Use the on-disk binary to locate the offset of the `.PyRuntime` section.
3. Add the section offset to the base address to compute the address in memory.

The sections below explain how to do this on each supported platform and include example code.

Linux (ELF)

To find the `PyRuntime` structure on Linux:

1. Read the process's memory map (for example, `/proc/<pid>/maps`) to find the address where the Python executable or `libpython` was loaded.
2. Parse the ELF section headers in the binary to get the offset of the `.PyRuntime` section.
3. Add that offset to the base address from step 1 to get the memory address of `PyRuntime`.

The following is an example implementation:

```
def find_py_runtime_linux(pid: int) -> int:
    # Step 1: Try to find the Python executable in memory
    binary_path, base_address = find_mapped_binary(
        pid, name_contains="python"
    )

    # Step 2: Fallback to shared library if executable is not
found
    if binary_path is None:
        binary_path, base_address = find_mapped_binary(
            pid, name_contains="libpython"
        )

    # Step 3: Parse ELF headers to get .PyRuntime section
offset
    section_offset = parse_elf_section_offset(
        binary_path, ".PyRuntime"
    )

    # Step 4: Compute PyRuntime address in memory
    return base_address + section_offset
```

On Linux systems, there are two main approaches to read memory from another process. The first is through the `/proc` filesystem, specifically by reading from `/proc/[pid]/mem` which provides direct access to the process's memory. This requires appropriate permissions - either being the same user as the target process or having root access. The second approach is using the `process_vm_readv()` system call which provides a more efficient way to copy memory between processes. While ptrace's `PTRACE_PEEKTEXT` operation can also be used to read memory, it is significantly slower as it only reads one word at a time and requires multiple context switches between the tracer and tracee processes.

For parsing ELF sections, the process involves reading and interpreting the ELF file format structures from the binary file on disk. The ELF header contains a pointer to the section header table. Each section header contains metadata about a section including its name (stored in a separate string table), offset, and size. To find a specific section like .PyRuntime, you need to walk through these headers and match the section name. The section header then provides the offset where that section exists in the file, which can be used to calculate its runtime address when the binary is loaded into memory.

You can read more about the ELF file format in the ELF specification [https://en.wikipedia.org/wiki/Executable_and_Linkable_Format].

macOS (Mach-O)

To find the `PyRuntime` structure on macOS:

1. Call `task_for_pid()` to get the `mach_port_t` task port for the target process. This handle is needed to read memory using APIs like `mach_vm_read_overwrite` and `mach_vm_region`.
2. Scan the memory regions to find the one containing the Python executable or `libpython`.
3. Load the binary file from disk and parse the Mach-O headers to find the section named `PyRuntime` in the `__DATA` segment. On macOS, symbol names are automatically prefixed with an underscore, so the `PyRuntime` symbol appears as `_PyRuntime` in the symbol table, but the section name is not affected.

The following is an example implementation:

```
def find_py_runtime_macos(pid: int) -> int:
    # Step 1: Get access to the process's memory
    handle = get_memory_access_handle(pid)

    # Step 2: Try to find the Python executable in memory
    binary_path, base_address = find_mapped_binary(
        handle, name_contains="python"
    )

    # Step 3: Fallback to libpython if the executable is not
found
    if binary_path is None:
        binary_path, base_address = find_mapped_binary(
            handle, name_contains="libpython"
        )

    # Step 4: Parse Mach-O headers to get __DATA,__PyRuntime
section offset
    section_offset = parse_macho_section_offset(
        binary_path, "__DATA", "__PyRuntime"
    )

    # Step 5: Compute the PyRuntime address in memory
    return base_address + section_offset
```

On macOS, accessing another process’s memory requires using Mach-O specific APIs and file formats. The first step is obtaining a `task_port` handle via `task_for_pid()`, which provides access to the target process’s memory space. This handle enables memory operations through APIs like `mach_vm_read_overwrite()`.

The process memory can be examined using `mach_vm_region()` to scan through the virtual memory space, while `proc_regionfilename()` helps identify which binary files are loaded at each memory region. When the Python binary or library is found, its Mach-O headers need to be parsed to locate the `PyRuntime` structure.

The Mach-O format organizes code and data into segments and sections. The `PyRuntime` structure lives in a section named `__PyRuntime` within the `__DATA` segment. The actual runtime address calculation involves finding the `__TEXT` segment which serves as the binary’s base address, then locating the `__DATA` segment containing our target section. The final address is computed by combining the base address with the appropriate section offsets from the Mach-O headers.

Note that accessing another process’s memory on macOS typically requires elevated privileges - either root access or special security entitlements granted to the debugging process.

Windows (PE)

To find the `PyRuntime` structure on Windows:

1. Use the ToolHelp API to enumerate all modules loaded in the target process. This is done using functions such as CreateToolhelp32Snapshot [https://learn.microsoft.com/en-us/windows/win32/api/tlhelp32/nf-tlhelp32-createtoolhelp32snapshot], Module32First [https://learn.microsoft.com/en-us/windows/win32/api/tlhelp32/nf-tlhelp32-module32first], and Module32Next [https://learn.microsoft.com/en-us/windows/win32/api/tlhelp32/nf-tlhelp32-module32next].
2. Identify the module corresponding to `python.exe` or `python`*`XY`*`.dll`, where `X` and `Y` are the major and minor version numbers of the Python version, and record its base address.
3. Locate the `PyRuntim` section. Due to the PE format's 8-character limit on section names (defined as `IMAGE_SIZEOF_SHORT_NAME`), the original name `PyRuntime` is truncated. This section contains the `PyRuntime` structure.
4. Retrieve the section's relative virtual address (RVA) and add it to the base address of the module.

The following is an example implementation:

```
def find_py_runtime_windows(pid: int) -> int:
    # Step 1: Try to find the Python executable in memory
    binary_path, base_address = find_loaded_module(
        pid, name_contains="python"
    )

    # Step 2: Fallback to shared pythonXY.dll if the executable
is not
    # found
    if binary_path is None:
        binary_path, base_address = find_loaded_module(
            pid, name_contains="python3"
        )

    # Step 3: Parse PE section headers to get the RVA of the
PyRuntime
```

```
    # section. The section name appears as "PyRuntim" due to
the
    # 8-character limit defined by the PE format
(IMAGE_SIZEOF_SHORT_NAME).
    section_rva = parse_pe_section_offset(binary_path,
"PyRuntim")

    # Step 4: Compute PyRuntime address in memory
    return base_address + section_rva
```

On Windows, accessing another process’s memory requires using the Windows API functions like `CreateToolhelp32Snapshot()` and `Module32First()/Module32Next()` to enumerate loaded modules. The `OpenProcess()` function provides a handle to access the target process’s memory space, enabling memory operations through `ReadProcessMemory()`.

The process memory can be examined by enumerating loaded modules to find the Python binary or DLL. When found, its PE headers need to be parsed to locate the `PyRuntime` structure.

The PE format organizes code and data into sections. The `PyRuntime` structure lives in a section named “PyRuntim” (truncated from “PyRuntime” due to PE’s 8-character name limit). The actual runtime address calculation involves finding the module’s base address from the module entry, then locating our target section in the PE headers. The final address is computed by combining the base address with the section’s virtual address from the PE section headers.

Note that accessing another process’s memory on Windows typically requires appropriate privileges - either administrative access or the `SeDebugPrivilege` privilege granted to the debugging process.

Reading _Py_DebugOffsets

Once the address of the `PyRuntime` structure has been determined, the next step is to read the `_Py_DebugOffsets` structure located at the beginning of the `PyRuntime` block.

This structure provides version-specific field offsets that are needed to safely read interpreter and thread state memory. These offsets vary between CPython versions and must be checked before use to ensure they are compatible.

To read and check the debug offsets, follow these steps:

1. Read memory from the target process starting at the `PyRuntime` address, covering the same number of bytes as the `_Py_DebugOffsets` structure. This structure is located at the very start of the `PyRuntime` memory block. Its layout is defined in CPython’s internal headers and stays the same within a given minor version, but may change in major versions.
2. Check that the structure contains valid data:
    - The `cookie` field must match the expected debug marker.
    - The `version` field must match the version of the Python interpreter used by the debugger.
    - If either the debugger or the target process is using a pre-release version (for example, an alpha, beta, or release candidate), the versions must match exactly.
    - The `free_threaded` field must have the same value in both the debugger and the target process.
3. If the structure is valid, the offsets it contains can be used to locate fields in memory. If any check fails, the debugger should stop the operation to avoid reading memory in the wrong format.

The following is an example implementation that reads and checks `_Py_DebugOffsets`:

```
def read_debug_offsets(pid: int, py_runtime_addr: int) -> DebugOffsets:
    # Step 1: Read memory from the target process at the PyRuntime address
    data = read_process_memory(
        pid, address=py_runtime_addr, size=DEBUG_OFFSETS_SIZE
    )

    # Step 2: Deserialize the raw bytes into a _Py_DebugOffsets structure
    debug_offsets = parse_debug_offsets(data)

    # Step 3: Validate the contents of the structure
    if debug_offsets.cookie != EXPECTED_COOKIE:
        raise RuntimeError("Invalid or missing debug cookie")
    if debug_offsets.version != LOCAL_PYTHON_VERSION:
        raise RuntimeError(
            "Mismatch between caller and target Python versions"
        )
    if debug_offsets.free_threaded != LOCAL_FREE_THREADED:
        raise RuntimeError("Mismatch in free-threaded configuration")

    return debug_offsets
```

Warning

Process suspension recommended

To avoid race conditions and ensure memory consistency, it is strongly recommended that the target process be suspended before performing any operations that read or write internal interpreter state. The Python runtime may concurrently mutate interpreter data structures—such as creating or destroying threads—during normal execution. This can result in invalid memory reads or writes.

A debugger may suspend execution by attaching to the process with `ptrace` or by sending a `SIGSTOP` signal. Execution should only be resumed after debugger-side memory operations are complete.

Note

Some tools, such as profilers or sampling-based debuggers, may operate on a running process without suspension. In such cases, tools must be explicitly designed to handle partially updated or inconsistent memory. For most debugger implementations, suspending the process remains the safest and most robust approach.

Locating the interpreter and thread state

Before code can be injected and executed in a remote Python process, the debugger must choose a thread in which to schedule execution. This is necessary because the control fields used to perform remote code injection are located in the `_PyRemoteDebuggerSupport` structure, which is embedded in a `PyThreadState` object. These fields are modified by the debugger to request execution of injected scripts.

The `PyThreadState` structure represents a thread running inside a Python interpreter. It maintains the thread's evaluation context and contains the fields required for debugger coordination. Locating a valid `PyThreadState` is therefore a key prerequisite for triggering execution remotely.

A thread is typically selected based on its role or ID. In most cases, the main thread is used, but some tools may target a specific thread by its native thread ID. Once the target thread is chosen, the debugger must locate both the interpreter and the associated thread state structures in memory.

The relevant internal structures are defined as follows:

- `PyInterpreterState` represents an isolated Python interpreter instance. Each interpreter maintains its own set of imported modules, built-in state, and thread state list. Although most Python applications use a single interpreter, CPython supports multiple interpreters in the same process.
- `PyThreadState` represents a thread running within an interpreter. It contains execution state and the control fields used by the debugger.

To locate a thread:

1. Use the offset `runtime_state.interpreters_head` to obtain the address of the first interpreter in the `PyRuntime` structure. This is the entry point to the linked list of active interpreters.
2. Use the offset `interpreter_state.threads_main` to access the main thread state associated with the selected interpreter. This is typically the most reliable thread to target.
3. Optionally, use the offset `interpreter_state.threads_head` to iterate through the linked list of all thread states. Each `PyThreadState` structure contains a `native_thread_id`

field, which may be compared to a target thread ID to find a specific thread.
4. Once a valid `PyThreadState` has been found, its address can be used in later steps of the protocol, such as writing debugger control fields and scheduling execution.

The following is an example implementation that locates the main thread state:

```
def find_main_thread_state(
    pid: int, py_runtime_addr: int, debug_offsets:
DebugOffsets,
) -> int:
    # Step 1: Read interpreters_head from PyRuntime
    interp_head_ptr = (
        py_runtime_addr +
debug_offsets.runtime_state.interpreters_head
    )
    interp_addr = read_pointer(pid, interp_head_ptr)
    if interp_addr == 0:
        raise RuntimeError("No interpreter found in the target
process")

    # Step 2: Read the threads_main pointer from the
interpreter
    threads_main_ptr = (
        interp_addr +
debug_offsets.interpreter_state.threads_main
    )
    thread_state_addr = read_pointer(pid, threads_main_ptr)
    if thread_state_addr == 0:
        raise RuntimeError("Main thread state is not
available")

    return thread_state_addr
```

The following example demonstrates how to locate a thread by its native thread ID:

```
def find_thread_by_id(
    pid: int,
    interp_addr: int,
    debug_offsets: DebugOffsets,
    target_tid: int,
) -> int:
    # Start at threads_head and walk the linked list
    thread_ptr = read_pointer(
        pid,
        interp_addr +
debug_offsets.interpreter_state.threads_head
    )

    while thread_ptr:
        native_tid_ptr = (
            thread_ptr +
debug_offsets.thread_state.native_thread_id
        )
        native_tid = read_int(pid, native_tid_ptr)
        if native_tid == target_tid:
            return thread_ptr
        thread_ptr = read_pointer(
            pid,
            thread_ptr + debug_offsets.thread_state.next
        )

    raise RuntimeError("Thread with the given ID was not
found")
```

Once a valid thread state has been located, the debugger can proceed with modifying its control fields and scheduling execution, as described in the next section.

Writing control information

Once a valid `PyThreadState` structure has been identified, the debugger may modify control fields within it to schedule the execution of a specified Python script. These control fields are checked periodically by the interpreter, and when set correctly, they trigger the execution of remote code at a safe point in the evaluation loop.

Each `PyThreadState` contains a `_PyRemoteDebuggerSupport` structure used for communication between the debugger and the interpreter. The locations of its fields are defined by the `_Py_DebugOffsets` structure and include the following:

- `debugger_script_path`: A fixed-size buffer that holds the full path to a Python source file (`.py`). This file must be accessible and readable by the target process when execution is triggered.
- `debugger_pending_call`: An integer flag. Setting this to `1` tells the interpreter that a script is ready to be executed.
- `eval_breaker`: A field checked by the interpreter during execution. Setting bit 5 (`_PY_EVAL_PLEASE_STOP_BIT`, value `1U << 5`) in this field causes the interpreter to pause and check for debugger activity.

To complete the injection, the debugger must perform the following steps:

1. Write the full script path into the `debugger_script_path` buffer.
2. Set `debugger_pending_call` to `1`.
3. Read the current value of `eval_breaker`, set bit 5 (`_PY_EVAL_PLEASE_STOP_BIT`), and write the updated value back. This signals the interpreter to check for debugger activity.

The following is an example implementation:

```
def inject_script(
    pid: int,
    thread_state_addr: int,
    debug_offsets: DebugOffsets,
    script_path: str
) -> None:
    # Compute the base offset of _PyRemoteDebuggerSupport
    support_base = (
        thread_state_addr +
        debug_offsets.debugger_support.remote_debugger_support
    )

    # Step 1: Write the script path into debugger_script_path
    script_path_ptr = (
        support_base +
        debug_offsets.debugger_support.debugger_script_path
    )
    write_string(pid, script_path_ptr, script_path)

    # Step 2: Set debugger_pending_call to 1
    pending_ptr = (
        support_base +
        debug_offsets.debugger_support.debugger_pending_call
    )
    write_int(pid, pending_ptr, 1)

    # Step 3: Set _PY_EVAL_PLEASE_STOP_BIT (bit 5, value 1 <<
5) in
    # eval_breaker
    eval_breaker_ptr = (
        thread_state_addr +
        debug_offsets.debugger_support.eval_breaker
    )
    breaker = read_int(pid, eval_breaker_ptr)
    breaker |= (1 << 5)
    write_int(pid, eval_breaker_ptr, breaker)
```

Once these fields are set, the debugger may resume the process (if it was suspended). The interpreter will process the request at the next safe evaluation point, load the script from disk, and execute it.

It is the responsibility of the debugger to ensure that the script file remains present and accessible to the target process during execution.

Note

Script execution is asynchronous. The script file cannot be deleted immediately after injection. The debugger should wait until the injected script has produced an observable effect before removing the file. This effect depends on what the script is designed to do. For example, a debugger might wait until the remote process connects back to a socket before removing the script. Once such an effect is observed, it is safe to assume the file is no longer needed.

Summary

To inject and execute a Python script in a remote process:

1. Locate the `PyRuntime` structure in the target process's memory.
2. Read and validate the `_Py_DebugOffsets` structure at the beginning of `PyRuntime`.
3. Use the offsets to locate a valid `PyThreadState`.
4. Write the path to a Python script into `debugger_script_path`.
5. Set the `debugger_pending_call` flag to `1`.
6. Set `_PY_EVAL_PLEASE_STOP_BIT` in the `eval_breaker` field.
7. Resume the process (if suspended). The script will execute at the next safe evaluation point.

Python Frequently Asked Questions

- General Python FAQ
- Programming FAQ
- Design and History FAQ
- Library and Extension FAQ
- Extending/Embedding FAQ
- Python on Windows FAQ
- Graphic User Interface FAQ
- “Why is Python Installed on my Computer?” FAQ

General Python FAQ

Contents

- General Python FAQ
    - General Information
        - What is Python?
        - What is the Python Software Foundation?
        - Are there copyright restrictions on the use of Python?
        - Why was Python created in the first place?
        - What is Python good for?
        - How does the Python version numbering scheme work?
        - How do I obtain a copy of the Python source?
        - How do I get documentation on Python?
        - I’ve never programmed before. Is there a Python tutorial?
        - Is there a newsgroup or mailing list devoted to Python?
        - How do I get a beta test version of Python?
        - How do I submit bug reports and patches for Python?
        - Are there any published articles about Python that I can reference?
        - Are there any books on Python?
        - Where in the world is www.python.org located?
        - Why is it called Python?
        - Do I have to like “Monty Python’s Flying Circus”?
    - Python in the real world
        - How stable is Python?

- How many people are using Python?
- Have any significant projects been done in Python?
- What new developments are expected for Python in the future?
- Is it reasonable to propose incompatible changes to Python?
- Is Python a good language for beginning programmers?

General Information

What is Python?

Python is an interpreted, interactive, object-oriented programming language. It incorporates modules, exceptions, dynamic typing, very high level dynamic data types, and classes. It supports multiple programming paradigms beyond object-oriented programming, such as procedural and functional programming. Python combines remarkable power with very clear syntax. It has interfaces to many system calls and libraries, as well as to various window systems, and is extensible in C or C++. It is also usable as an extension language for applications that need a programmable interface. Finally, Python is portable: it runs on many Unix variants including Linux and macOS, and on Windows.

To find out more, start with The Python Tutorial. The Beginner’s Guide to Python [https://wiki.python.org/moin/BeginnersGuide] links to other introductory tutorials and resources for learning Python.

What is the Python Software Foundation?

The Python Software Foundation is an independent non-profit organization that holds the copyright on Python versions 2.1 and newer. The PSF's mission is to advance open source technology related to the Python programming language and to publicize the use of Python. The PSF's home page is at https://www.python.org/psf/.

Donations to the PSF are tax-exempt in the US. If you use Python and find it helpful, please contribute via the PSF donation page [https://www.python.org/psf/donations/].

Are there copyright restrictions on the use of Python?

You can do anything you want with the source, as long as you leave the copyrights in and display those copyrights in any documentation about Python that you produce. If you honor the copyright rules, it's OK to use Python for commercial use, to sell copies of Python in source or binary form (modified or unmodified), or to sell products that incorporate Python in some form. We would still like to know about all commercial use of Python, of course.

See the license page [https://docs.python.org/3/license.html] to find further explanations and the full text of the PSF License.

The Python logo is trademarked, and in certain cases permission is required to use it. Consult the Trademark Usage Policy [https://www.python.org/psf/trademarks/] for more information.

Why was Python created in the first place?

Here's a *very* brief summary of what started it all, written by Guido van Rossum:

I had extensive experience with implementing an interpreted language in the ABC group at CWI, and from working with this group I had learned a lot about language design. This is the origin of many Python features, including the use of indentation for statement grouping and the inclusion of very-high-level data types (although the details are all different in Python).

I had a number of gripes about the ABC language, but also liked many of its features. It was impossible to extend the ABC language (or its implementation) to remedy my complaints – in fact its lack of extensibility was one of its biggest problems. I had some experience with using Modula-2+ and talked with the designers of Modula-3 and read the Modula-3 report. Modula-3 is the origin of the syntax and semantics used for exceptions, and some other Python features.

I was working in the Amoeba distributed operating system group at CWI. We needed a better way to do system administration than by writing either C programs or Bourne shell scripts, since Amoeba had its own system call interface which wasn't easily accessible from the Bourne shell. My experience with error handling in Amoeba made me acutely aware of the importance of exceptions as a programming language feature.

It occurred to me that a scripting language with a syntax like ABC but with access to the Amoeba system calls would fill the need. I realized that it would be foolish to write an Amoeba-specific language, so I decided that I needed a language that was generally extensible.

> During the 1989 Christmas holidays, I had a lot of time on my hand, so I decided to give it a try. During the next year, while still mostly working on it in my own time, Python was used in the Amoeba project with increasing success, and the feedback from colleagues made me add many early improvements.
>
> In February 1991, after just over a year of development, I decided to post to USENET. The rest is in the `Misc/HISTORY` file.

What is Python good for?

Python is a high-level general-purpose programming language that can be applied to many different classes of problems.

The language comes with a large standard library that covers areas such as string processing (regular expressions, Unicode, calculating differences between files), internet protocols (HTTP, FTP, SMTP, XML-RPC, POP, IMAP), software engineering (unit testing, logging, profiling, parsing Python code), and operating system interfaces (system calls, filesystems, TCP/IP sockets). Look at the table of contents for The Python Standard Library to get an idea of what's available. A wide variety of third-party extensions are also available. Consult the Python Package Index [https://pypi.org] to find packages of interest to you.

How does the Python version numbering scheme work?

Python versions are numbered "A.B.C" or "A.B":

- *A* is the major version number – it is only incremented for really major changes in the language.

- *B* is the minor version number – it is incremented for less earth-shattering changes.
- *C* is the micro version number – it is incremented for each bugfix release.

Not all releases are bugfix releases. In the run-up to a new feature release, a series of development releases are made, denoted as alpha, beta, or release candidate. Alphas are early releases in which interfaces aren't yet finalized; it's not unexpected to see an interface change between two alpha releases. Betas are more stable, preserving existing interfaces but possibly adding new modules, and release candidates are frozen, making no changes except as needed to fix critical bugs.

Alpha, beta and release candidate versions have an additional suffix:

- The suffix for an alpha version is “aN” for some small number *N*.
- The suffix for a beta version is “bN” for some small number *N*.
- The suffix for a release candidate version is “rcN” for some small number *N*.

In other words, all versions labeled *2.0aN* precede the versions labeled *2.0bN*, which precede versions labeled *2.0rcN*, and *those* precede 2.0.

You may also find version numbers with a “+” suffix, e.g. “2.2+”. These are unreleased versions, built directly from the CPython development repository. In practice, after a final minor release is made, the version is incremented to the next minor version, which becomes the “a0” version, e.g. “2.4a0”.

See the Developer’s Guide [https://devguide.python.org/developer-workflow/development-cycle/] for more information about the development cycle, and **PEP 387** [https://peps.python.org/pep-0387/] to learn more about Python’s backward compatibility policy. See also the documentation for `sys.version`, `sys.hexversion`, and `sys.version_info`.

How do I obtain a copy of the Python source?

The latest Python source distribution is always available from python.org, at https://www.python.org/downloads/. The latest development sources can be obtained at https://github.com/python/cpython/.

The source distribution is a gzipped tar file containing the complete C source, Sphinx-formatted documentation, Python library modules, example programs, and several useful pieces of freely distributable software. The source will compile and run out of the box on most UNIX platforms.

Consult the Getting Started section of the Python Developer’s Guide [https://devguide.python.org/setup/] for more information on getting the source code and compiling it.

How do I get documentation on Python?

The standard documentation for the current stable version of Python is available at https://docs.python.org/3/. EPUB, plain text, and downloadable HTML versions are also available at https://docs.python.org/3/download.html.

The documentation is written in reStructuredText and processed by the Sphinx documentation tool [https://www.sphinx-doc.org/]. The reStructuredText source for the documentation is part of the Python source distribution.

I’ve never programmed before. Is there a Python tutorial?

There are numerous tutorials and books available. The standard documentation includes The Python Tutorial.

Consult the Beginner’s Guide [https://wiki.python.org/moin/BeginnersGuide] to find information for beginning Python programmers, including lists of tutorials.

Is there a newsgroup or mailing list devoted to Python?

There is a newsgroup, *comp.lang.python*, and a mailing list, python-list [https://mail.python.org/mailman/listinfo/python-list]. The newsgroup and mailing list are gatewayed into each other – if you can read news it’s unnecessary to subscribe to the mailing list. *comp.lang.python* is high-traffic, receiving hundreds of postings every day, and Usenet readers are often more able to cope with this volume.

Announcements of new software releases and events can be found in comp.lang.python.announce, a low-traffic moderated list that receives about five postings per day. It’s available as the python-announce mailing list [https://mail.python.org/mailman3/lists/python-announce-list.python.org/].

More info about other mailing lists and newsgroups can be found at https://www.python.org/community/lists/.

How do I get a beta test version of Python?

Alpha and beta releases are available from https://www.python.org/downloads/. All releases are announced on the comp.lang.python and comp.lang.python.announce newsgroups and on the Python home page at https://www.python.org/; an RSS feed of news is available.

You can also access the development version of Python through Git. See The Python Developer's Guide [https://devguide.python.org/] for details.

How do I submit bug reports and patches for Python?

To report a bug or submit a patch, use the issue tracker at https://github.com/python/cpython/issues.

For more information on how Python is developed, consult the Python Developer's Guide [https://devguide.python.org/].

Are there any published articles about Python that I can reference?

It's probably best to cite your favorite book about Python.

The very first article [https://ir.cwi.nl/pub/18204] about Python was written in 1991 and is now quite outdated.

> Guido van Rossum and Jelke de Boer, "Interactively Testing Remote Servers Using the Python Programming Language", CWI Quarterly, Volume 4, Issue 4 (December 1991), Amsterdam, pp 283–303.

Are there any books on Python?

Yes, there are many, and more are being published. See the python.org wiki at https://wiki.python.org/moin/PythonBooks for a list.

You can also search online bookstores for “Python” and filter out the Monty Python references; or perhaps search for “Python” and “language”.

Where in the world is www.python.org located?

The Python project’s infrastructure is located all over the world and is managed by the Python Infrastructure Team. Details here [https://infra.psf.io].

Why is it called Python?

When he began implementing Python, Guido van Rossum was also reading the published scripts from “Monty Python’s Flying Circus” [https://en.wikipedia.org/wiki/Monty_Python], a BBC comedy series from the 1970s. Van Rossum thought he needed a name that was short, unique, and slightly mysterious, so he decided to call the language Python.

Do I have to like “Monty Python’s Flying Circus”?

No, but it helps. :)

Python in the real world

How stable is Python?

Very stable. New, stable releases have been coming out roughly every 6 to 18 months since 1991, and this seems likely to continue. As of version 3.9, Python will have a new feature release every 12 months (**PEP 602** [https://peps.python.org/pep-0602/]).

The developers issue bugfix releases of older versions, so the stability of existing releases gradually improves. Bugfix releases, indicated by a third component of the version number (e.g. 3.5.3, 3.6.2), are managed for stability; only fixes for known problems are included in a bugfix release, and it's guaranteed that interfaces will remain the same throughout a series of bugfix releases.

The latest stable releases can always be found on the Python download page [https://www.python.org/downloads/]. Python 3.x is the recommended version and supported by most widely used libraries. Python 2.x **is not maintained anymore** [https://peps.python.org/pep-0373/].

How many people are using Python?

There are probably millions of users, though it's difficult to obtain an exact count.

Python is available for free download, so there are no sales figures, and it's available from many different sites and packaged with many Linux distributions, so download statistics don't tell the whole story either.

The comp.lang.python newsgroup is very active, but not all Python users post to the group or even read it.

Have any significant projects been done in Python?

See https://www.python.org/about/success for a list of projects that use Python. Consulting the proceedings for past Python conferences [https://www.python.org/community/workshops/] will reveal contributions from many different companies and organizations.

High-profile Python projects include the Mailman mailing list manager [https://www.list.org] and the Zope application server [https://www.zope.dev]. Several Linux distributions, most notably Red Hat [https://www.redhat.com], have written part or all of their installer and system administration software in Python. Companies that use Python internally include Google, Yahoo, and Lucasfilm Ltd.

What new developments are expected for Python in the future?

See https://peps.python.org/ for the Python Enhancement Proposals (PEPs). PEPs are design documents describing a suggested new feature for Python, providing a concise technical specification and a rationale. Look for a PEP titled “Python X.Y Release Schedule”, where X.Y is a version that hasn’t been publicly released yet.

New development is discussed on the python-dev mailing list [https://mail.python.org/mailman3/lists/python-dev.python.org/].

Is it reasonable to propose incompatible changes to Python?

In general, no. There are already millions of lines of Python code around the world, so any change in the language that invalidates more than a very small fraction of existing programs has to be frowned upon. Even if you can provide a conversion program, there’s still the problem of updating all documentation; many books have been written about Python, and we don’t want to invalidate them all at a single stroke.

Providing a gradual upgrade path is necessary if a feature has to be changed. **PEP 5** [https://peps.python.org/pep-0005/]

describes the procedure followed for introducing backward-incompatible changes while minimizing disruption for users.

Is Python a good language for beginning programmers?

Yes.

It is still common to start students with a procedural and statically typed language such as Pascal, C, or a subset of C++ or Java. Students may be better served by learning Python as their first language. Python has a very simple and consistent syntax and a large standard library and, most importantly, using Python in a beginning programming course lets students concentrate on important programming skills such as problem decomposition and data type design. With Python, students can be quickly introduced to basic concepts such as loops and procedures. They can probably even work with user-defined objects in their very first course.

For a student who has never programmed before, using a statically typed language seems unnatural. It presents additional complexity that the student must master and slows the pace of the course. The students are trying to learn to think like a computer, decompose problems, design consistent interfaces, and encapsulate data. While learning to use a statically typed language is important in the long term, it is not necessarily the best topic to address in the students’ first programming course.

Many other aspects of Python make it a good first language. Like Java, Python has a large standard library so that students can be assigned programming projects very early in the course that *do* something. Assignments aren’t restricted to the standard four-function calculator and check balancing programs. By using the standard library, students can gain the satisfaction of working on realistic applications as they learn the fundamentals of programming. Using the standard library also teaches students about code reuse. Third-party modules such as PyGame are also helpful in extending the students’ reach.

Python’s interactive interpreter enables students to test language features while they’re programming. They can keep a window with the interpreter running while they enter their program’s source in another window. If they can’t remember the methods for a list, they can do something like this:

```
>>> L = []
>>> dir(L)
['__add__', '__class__', '__contains__', '__delattr__',
'__delitem__',
'__dir__', '__doc__', '__eq__', '__format__', '__ge__',
'__getattribute__', '__getitem__', '__gt__', '__hash__',
'__iadd__',
'__imul__', '__init__', '__iter__', '__le__', '__len__',
'__lt__',
'__mul__', '__ne__', '__new__', '__reduce__', '__reduce_ex__',
'__repr__', '__reversed__', '__rmul__', '__setattr__',
'__setitem__',
'__sizeof__', '__str__', '__subclasshook__', 'append', 'clear',
'copy', 'count', 'extend', 'index', 'insert', 'pop', 'remove',
'reverse', 'sort']
>>> [d for d in dir(L) if '__' not in d]
['append', 'clear', 'copy', 'count', 'extend', 'index',
'insert', 'pop', 'remove', 'reverse', 'sort']

>>> help(L.append)
Help on built-in function append:

append(...)
    L.append(object) -> None -- append object to end

>>> L.append(1)
>>> L
[1]
```

With the interpreter, documentation is never far from the student as they are programming.

There are also good IDEs for Python. IDLE is a cross-platform IDE for Python that is written in Python using Tkinter. Emacs users will be happy to know that there is a very good Python mode for Emacs. All of these programming environments provide syntax highlighting, auto-indenting, and access to the interactive interpreter while coding. Consult the Python wiki [https://wiki.python.org/moin/PythonEditors] for a full list of Python editing environments.

If you want to discuss Python’s use in education, you may be interested in joining the edu-sig mailing list [https://www.python.org/community/sigs/current/edu-sig].

Programming FAQ

Contents

- Programming FAQ
    - General Questions
        - Is there a source code level debugger with breakpoints, single-stepping, etc.?
        - Are there tools to help find bugs or perform static analysis?
        - How can I create a stand-alone binary from a Python script?
        - Are there coding standards or a style guide for Python programs?
    - Core Language
        - Why am I getting an UnboundLocalError when the variable has a value?
        - What are the rules for local and global variables in Python?
        - Why do lambdas defined in a loop with different values all return the same result?
        - How do I share global variables across modules?
        - What are the “best practices” for using import in a module?
        - Why are default values shared between objects?
        - How can I pass optional or keyword parameters from one function to another?
        - What is the difference between arguments and parameters?
        - Why did changing list ‘y’ also change list ‘x’?

    - How do I write a function with output parameters (call by reference)?
    - How do you make a higher order function in Python?
    - How do I copy an object in Python?
    - How can I find the methods or attributes of an object?
    - How can my code discover the name of an object?
    - What’s up with the comma operator’s precedence?
    - Is there an equivalent of C’s “?:” ternary operator?
    - Is it possible to write obfuscated one-liners in Python?
    - What does the slash(/) in the parameter list of a function mean?
- Numbers and strings
    - How do I specify hexadecimal and octal integers?
    - Why does -22 // 10 return -3?
    - How do I get int literal attribute instead of SyntaxError?
    - How do I convert a string to a number?
    - How do I convert a number to a string?
    - How do I modify a string in place?
    - How do I use strings to call functions/methods?
    - Is there an equivalent to Perl’s `chomp()` for removing trailing newlines from strings?
    - Is there a `scanf()` or `sscanf()` equivalent?
    - What does `UnicodeDecodeError` or `UnicodeEncodeError` error mean?
    - Can I end a raw string with an odd number of backslashes?
- Performance
    - My program is too slow. How do I speed it up?

    - What is the most efficient way to concatenate many strings together?
  - Sequences (Tuples/Lists)
    - How do I convert between tuples and lists?
    - What’s a negative index?
    - How do I iterate over a sequence in reverse order?
    - How do you remove duplicates from a list?
    - How do you remove multiple items from a list
    - How do you make an array in Python?
    - How do I create a multidimensional list?
    - How do I apply a method or function to a sequence of objects?
    - Why does a_tuple[i] += [‘item’] raise an exception when the addition works?
    - I want to do a complicated sort: can you do a Schwartzian Transform in Python?
    - How can I sort one list by values from another list?
  - Objects
    - What is a class?
    - What is a method?
    - What is self?
    - How do I check if an object is an instance of a given class or of a subclass of it?
    - What is delegation?
    - How do I call a method defined in a base class from a derived class that extends it?
    - How can I organize my code to make it easier to change the base class?
    - How do I create static class data and static class methods?
    - How can I overload constructors (or methods) in Python?
    - I try to use __spam and I get an error about _SomeClassName__spam.

    - My class defines __del__ but it is not called when I delete the object.
    - How do I get a list of all instances of a given class?
    - Why does the result of `id()` appear to be not unique?
    - When can I rely on identity tests with the *is* operator?
    - How can a subclass control what data is stored in an immutable instance?
    - How do I cache method calls?
  - Modules
    - How do I create a .pyc file?
    - How do I find the current module name?
    - How can I have modules that mutually import each other?
    - __import__('x.y.z') returns <module 'x'>; how do I get z?
    - When I edit an imported module and reimport it, the changes don't show up. Why does this happen?

General Questions

Is there a source code level debugger with breakpoints, single-stepping, etc.?

Yes.

Several debuggers for Python are described below, and the built-in function `breakpoint()` allows you to drop into any of them.

The pdb module is a simple but adequate console-mode debugger for Python. It is part of the standard Python library, and is **`documented in the Library Reference Manual`**. You can also write your own debugger by using the code for pdb as an example.

The IDLE interactive development environment, which is part of the standard Python distribution (normally available as Tools/scripts/idle3 [https://github.com/python/cpython/blob/main/Tools/scripts/idle3]), includes a graphical debugger.

PythonWin is a Python IDE that includes a GUI debugger based on pdb. The PythonWin debugger colors breakpoints and has quite a few cool features such as debugging non-PythonWin programs. PythonWin is available as part of pywin32 [https://github.com/mhammond/pywin32] project and as a part of the ActivePython [https://www.activestate.com/products/python/] distribution.

Eric [https://eric-ide.python-projects.org/] is an IDE built on PyQt and the Scintilla editing component.

trepan3k [https://github.com/rocky/python3-trepan/] is a gdb-like debugger.

Visual Studio Code [https://code.visualstudio.com/] is an IDE with debugging tools that integrates with version-control software.

There are a number of commercial Python IDEs that include graphical debuggers. They include:

- Wing IDE [https://wingware.com/]
- Komodo IDE [https://www.activestate.com/products/komodo-ide/]
- PyCharm [https://www.jetbrains.com/pycharm/]

Are there tools to help find bugs or perform static analysis?

Yes.

Pylint [https://pylint.pycqa.org/en/latest/index.html] and Pyflakes [https://github.com/PyCQA/pyflakes] do basic checking that will help you catch bugs sooner.

Static type checkers such as Mypy [https://mypy-lang.org/], Pyre [https://pyre-check.org/], and Pytype [https://github.com/google/pytype] can check type hints in Python source code.

How can I create a stand-alone binary from a Python script?

You don’t need the ability to compile Python to C code if all you want is a stand-alone program that users can download and run without having to install the Python distribution first. There are a number of tools that determine the set of modules required by a program and bind these modules together with a Python binary to produce a single executable.

One is to use the freeze tool, which is included in the Python source tree as Tools/freeze [https://github.com/python/cpython/tree/main/Tools/freeze]. It converts Python byte code to C arrays; with a C compiler you can embed all your modules into a new program, which is then linked with the standard Python modules.

It works by scanning your source recursively for import statements (in both forms) and looking for the modules in the standard Python path as well as in the source directory (for built-in modules). It then turns the bytecode for modules written in Python into C code (array initializers that can be turned into code objects using the marshal module) and creates a custom-made config file that only contains those built-in modules which are actually used in the program. It then compiles the generated C code and links it with the rest of the Python interpreter to form a self-contained binary which acts exactly like your script.

The following packages can help with the creation of console and GUI executables:

- Nuitka [https://nuitka.net/] (Cross-platform)
- PyInstaller [https://pyinstaller.org/] (Cross-platform)
- PyOxidizer [https://pyoxidizer.readthedocs.io/en/stable/] (Cross-platform)
- cx_Freeze [https://marcelotduarte.github.io/cx_Freeze/] (Cross-platform)
- py2app [https://github.com/ronaldoussoren/py2app] (macOS only)
- py2exe [https://www.py2exe.org/] (Windows only)

Are there coding standards or a style guide for Python programs?

Yes. The coding style required for standard library modules is documented as **PEP 8** [https://peps.python.org/pep-0008/].

Core Language

Why am I getting an UnboundLocalError when the variable has a value?

It can be a surprise to get the `UnboundLocalError` in previously working code when it is modified by adding an assignment statement somewhere in the body of a function.

This code:

```
>>> x = 10
>>> def bar():
...     print(x)
...
>>> bar()
10
```

works, but this code:

```
>>> x = 10
>>> def foo():
...     print(x)
...     x += 1
```

results in an **`UnboundLocalError`**:

```
>>> foo()
Traceback (most recent call last):
  ...
UnboundLocalError: local variable 'x' referenced before
assignment
```

This is because when you make an assignment to a variable in a scope, that variable becomes local to that scope and shadows any similarly named variable in the outer scope. Since the last statement in foo assigns a new value to `x`, the compiler recognizes it as a local variable. Consequently when the earlier `print(x)` attempts to print the uninitialized local variable and an error results.

In the example above you can access the outer scope variable by declaring it global:

```
>>> x = 10
>>> def foobar():
...     global x
...     print(x)
...     x += 1
...
>>> foobar()
10
```

This explicit declaration is required in order to remind you that (unlike the superficially analogous situation with class and instance variables) you are actually modifying the value of the variable in the outer scope:

```
>>> print(x)
11
```

You can do a similar thing in a nested scope using the `nonlocal` keyword:

```
>>> def foo():
...    x = 10
...    def bar():
...        nonlocal x
...        print(x)
...        x += 1
...    bar()
...    print(x)
...
>>> foo()
10
11
```

What are the rules for local and global variables in Python?

In Python, variables that are only referenced inside a function are implicitly global. If a variable is assigned a value anywhere within the function's body, it's assumed to be a local unless explicitly declared as global.

Though a bit surprising at first, a moment's consideration explains this. On one hand, requiring `global` for assigned variables provides a bar against unintended side-effects. On the other hand, if `global` was required for all global references, you'd be using `global` all the time. You'd have to declare as global every reference to a built-in function or to a component of an imported module. This clutter would defeat the usefulness of the `global` declaration for identifying side-effects.

Why do lambdas defined in a loop with different values all return the same result?

Assume you use a for loop to define a few different lambdas (or even plain functions), e.g.:

```
>>> squares = []
>>> for x in range(5):
...     squares.append(lambda: x**2)
```

This gives you a list that contains 5 lambdas that calculate `x**2`. You might expect that, when called, they would return, respectively, `0`, `1`, `4`, `9`, and `16`. However, when you actually try you will see that they all return `16`:

```
>>> squares[2]()
16
>>> squares[4]()
16
```

This happens because `x` is not local to the lambdas, but is defined in the outer scope, and it is accessed when the lambda is called — not when it is defined. At the end of the loop, the value of `x` is `4`, so all the functions now return `4**2`, i.e. `16`. You can also verify this by changing the value of `x` and see how the results of the lambdas change:

```
>>> x = 8
>>> squares[2]()
64
```

In order to avoid this, you need to save the values in variables local to the lambdas, so that they don't rely on the value of the global `x`:

```
>>> squares = []
>>> for x in range(5):
...     squares.append(lambda n=x: n**2)
```

Here, `n=x` creates a new variable `n` local to the lambda and computed when the lambda is defined so that it has the same value that `x` had at that point in the loop. This means that the value of `n` will be `0` in the first lambda, `1` in the second, `2` in the third, and so on. Therefore each lambda will now return the correct result:

```
>>> squares[2]()
4
>>> squares[4]()
16
```

Note that this behaviour is not peculiar to lambdas, but applies to regular functions too.

How do I share global variables across modules?

The canonical way to share information across modules within a single program is to create a special module (often called config or cfg). Just import the config module in all modules of your application; the module then becomes available as a global name. Because there is only one instance of each module, any changes made to the module object get reflected everywhere. For example:

config.py:

```
x = 0   # Default value of the 'x' configuration setting
```

mod.py:

```
import config
config.x = 1
```

main.py:

```
import config
import mod
print(config.x)
```

Note that using a module is also the basis for implementing the singleton design pattern, for the same reason.

What are the “best practices” for using import in a module?

In general, don’t use `from modulename import *`. Doing so clutters the importer’s namespace, and makes it much harder for linters to detect undefined names.

Import modules at the top of a file. Doing so makes it clear what other modules your code requires and avoids questions of whether the module name is in scope. Using one import per line makes it easy to add and delete module imports, but using multiple imports per line uses less screen space.

It’s good practice if you import modules in the following order:

1. standard library modules – e.g. `sys`, `os`, `argparse`, `re`
2. third-party library modules (anything installed in Python’s site-packages directory) – e.g. **`dateutil`**, **`requests`**, **`PIL.Image`**
3. locally developed modules

It is sometimes necessary to move imports to a function or class to avoid problems with circular imports. Gordon McMillan says:

> Circular imports are fine where both modules use the “import <module>” form of import. They fail when the 2nd module wants to grab a name out of the first (“from module import name”) and the import is at the top level. That’s because names in the 1st are not yet available, because the first module is busy importing the 2nd.

In this case, if the second module is only used in one function, then the import can easily be moved into that function. By the time the import is called, the first module will have finished initializing, and the second module can do its import.

It may also be necessary to move imports out of the top level of code if some of the modules are platform-specific. In that case, it may not even be possible to import all of the modules at the top of the file. In this case, importing the correct modules in the corresponding platform-specific code is a good option.

Only move imports into a local scope, such as inside a function definition, if it’s necessary to solve a problem such as avoiding a circular import or are trying to reduce the initialization time of a module. This technique is especially helpful if many of the imports are unnecessary depending on how the program executes. You may also want to move imports into a function if the modules are only ever used in that function. Note that loading a module the first time may be expensive because of the one time initialization of the module, but loading a module multiple times is virtually free, costing only a couple of dictionary lookups. Even if the module name has gone out of scope, the module is probably available in `sys.modules`.

Why are default values shared between objects?

This type of bug commonly bites neophyte programmers. Consider this function:

```
def foo(mydict={}):  # Danger: shared reference to one dict for
all calls
    ... compute something ...
    mydict[key] = value
    return mydict
```

The first time you call this function, `mydict` contains a single item. The second time, `mydict` contains two items because when `foo()` begins executing, `mydict` starts out with an item already in it.

It is often expected that a function call creates new objects for default values. This is not what happens. Default values are created exactly once, when the function is defined. If that object is changed, like the dictionary in this example, subsequent calls to the function will refer to this changed object.

By definition, immutable objects such as numbers, strings, tuples, and `None`, are safe from change. Changes to mutable objects such as dictionaries, lists, and class instances can lead to confusion.

Because of this feature, it is good programming practice to not use mutable objects as default values. Instead, use `None` as the default value and inside the function, check if the parameter is `None` and create a new list/dictionary/whatever if it is. For example, don’t write:

```
def foo(mydict={}):
    ...
```

but:

```
def foo(mydict=None):
    if mydict is None:
        mydict = {}  # create a new dict for local namespace
```

This feature can be useful. When you have a function that's time-consuming to compute, a common technique is to cache the parameters and the resulting value of each call to the function, and return the cached value if the same value is requested again. This is called "memoizing", and can be implemented like this:

```
# Callers can only provide two parameters and optionally pass
_cache by keyword
def expensive(arg1, arg2, *, _cache={}):
    if (arg1, arg2) in _cache:
        return _cache[(arg1, arg2)]

    # Calculate the value
    result = ... expensive computation ...
    _cache[(arg1, arg2)] = result           # Store result in
the cache
    return result
```

You could use a global variable containing a dictionary instead of the default value; it's a matter of taste.

How can I pass optional or keyword parameters from one function to another?

Collect the arguments using the `*` and `**` specifiers in the function's parameter list; this gives you the positional arguments as a tuple and the keyword arguments as a dictionary. You can then pass these arguments when calling another function by using `*` and `**`:

```
def f(x, *args, **kwargs):
    ...
    kwargs['width'] = '14.3c'
    ...
    g(x, *args, **kwargs)
```

What is the difference between arguments and parameters?

Parameters are defined by the names that appear in a function definition, whereas arguments are the values actually passed to a function when calling it. Parameters define what kind of arguments a function can accept. For example, given the function definition:

```
def func(foo, bar=None, **kwargs):
    pass
```

foo, *bar* and *kwargs* are parameters of `func`. However, when calling `func`, for example:

```
func(42, bar=314, extra=somevar)
```

the values `42`, `314`, and `somevar` are arguments.

Why did changing list ‘y’ also change list ‘x’?

If you wrote code like:

```
>>> x = []
>>> y = x
>>> y.append(10)
>>> y
[10]
>>> x
[10]
```

you might be wondering why appending an element to `y` changed `x` too.

There are two factors that produce this result:

1. Variables are simply names that refer to objects. Doing `y = x` doesn’t create a copy of the list – it creates a new variable `y` that refers to the same object `x` refers to. This means that there is only one object (the list), and both `x` and `y` refer to it.
2. Lists are mutable, which means that you can change their content.

After the call to **`append()`**, the content of the mutable object has changed from `[]` to `[10]`. Since both the variables refer to the same object, using either name accesses the modified value `[10]`.

If we instead assign an immutable object to `x`:

```
>>> x = 5  # ints are immutable
>>> y = x
>>> x = x + 1  # 5 can't be mutated, we are creating a new
object here
>>> x
6
>>> y
5
```

we can see that in this case `x` and `y` are not equal anymore. This is because integers are immutable, and when we do `x = x + 1` we are not mutating the int `5` by incrementing its value; instead, we are creating a new object (the int `6`) and assigning it to `x` (that is, changing which object `x` refers to). After this assignment we have two objects (the ints `6` and `5`) and two variables that refer to them (`x` now refers to `6` but `y` still refers to `5`).

Some operations (for example `y.append(10)` and `y.sort()`) mutate the object, whereas superficially similar operations (for example `y = y + [10]` and **`sorted(y)`**) create a new object. In general in Python (and in all cases in the standard library) a method that mutates an object will return `None` to help avoid getting the two types of operations confused. So if you mistakenly write `y.sort()` thinking it will give you a sorted copy of `y`, you'll instead end up with `None`, which will likely cause your program to generate an easily diagnosed error.

However, there is one class of operations where the same operation sometimes has different behaviors with different types: the augmented assignment operators. For example, `+=` mutates lists but not tuples or ints (`a_list += [1, 2, 3]` is equivalent to `a_list.extend([1, 2, 3])` and mutates `a_list`, whereas `some_tuple += (1, 2, 3)` and `some_int += 1` create new objects).

In other words:

- If we have a mutable object (**`list`**, **`dict`**, **`set`**, etc.), we can use some specific operations to mutate it and all the variables that refer to it will see the change.
- If we have an immutable object (**`str`**, **`int`**, **`tuple`**, etc.), all the variables that refer to it will always see the same value, but operations that transform that value into a new value always return a new object.

If you want to know if two variables refer to the same object or not, you can use the **`is`** operator, or the built-in function **`id()`**.

How do I write a function with output parameters (call by reference)?

Remember that arguments are passed by assignment in Python. Since assignment just creates references to objects, there's no alias between an argument name in the caller and callee, and so no call-by-reference per se. You can achieve the desired effect in a number of ways.

1. By returning a tuple of the results:

   ```
   >>> def func1(a, b):
   ...     a = 'new-value'        # a and b are local names
   ...     b = b + 1              # assigned to new objects
   ...     return a, b            # return new values
   ...
   >>> x, y = 'old-value', 99
   >>> func1(x, y)
   ('new-value', 100)
   ```

   This is almost always the clearest solution.

2. By using global variables. This isn’t thread-safe, and is not recommended.

3. By passing a mutable (changeable in-place) object:

   ```
   >>> def func2(a):
   ...     a[0] = 'new-value'     # 'a' references a mutable
   list
   ...     a[1] = a[1] + 1        # changes a shared object
   ...
   >>> args = ['old-value', 99]
   >>> func2(args)
   >>> args
   ['new-value', 100]
   ```

4. By passing in a dictionary that gets mutated:

   ```
   >>> def func3(args):
   ...     args['a'] = 'new-value'     # args is a mutable
   dictionary
   ...     args['b'] = args['b'] + 1   # change it in-place
   ...
   >>> args = {'a': 'old-value', 'b': 99}
   >>> func3(args)
   >>> args
   {'a': 'new-value', 'b': 100}
   ```

5. Or bundle up values in a class instance:

```
>>> class Namespace:
...     def __init__(self, /, **args):
...         for key, value in args.items():
...             setattr(self, key, value)
...
>>> def func4(args):
...     args.a = 'new-value'        # args is a mutable
Namespace
...     args.b = args.b + 1         # change object in-
place
...
>>> args = Namespace(a='old-value', b=99)
>>> func4(args)
>>> vars(args)
{'a': 'new-value', 'b': 100}
```

There's almost never a good reason to get this complicated.

Your best choice is to return a tuple containing the multiple results.

How do you make a higher order function in Python?

You have two choices: you can use nested scopes or you can use callable objects. For example, suppose you wanted to define `linear(a,b)` which returns a function `f(x)` that computes the value `a*x+b`. Using nested scopes:

```
def linear(a, b):
    def result(x):
        return a * x + b
    return result
```

Or using a callable object:

```
class linear:

    def __init__(self, a, b):
        self.a, self.b = a, b

    def __call__(self, x):
        return self.a * x + self.b
```

In both cases,

```
taxes = linear(0.3, 2)
```

gives a callable object where `taxes(10e6) == 0.3 * 10e6 + 2`.

The callable object approach has the disadvantage that it is a bit slower and results in slightly longer code. However, note that a collection of callables can share their signature via inheritance:

```
class exponential(linear):
    # __init__ inherited
    def __call__(self, x):
        return self.a * (x ** self.b)
```

Object can encapsulate state for several methods:

```
class counter:

    value = 0

    def set(self, x):
        self.value = x

    def up(self):
        self.value = self.value + 1

    def down(self):
        self.value = self.value - 1

count = counter()
inc, dec, reset = count.up, count.down, count.set
```

Here `inc()`, `dec()` and `reset()` act like functions which share the same counting variable.

How do I copy an object in Python?

In general, try `copy.copy()` or `copy.deepcopy()` for the general case. Not all objects can be copied, but most can.

Some objects can be copied more easily. Dictionaries have a `copy()` method:

```
newdict = olddict.copy()
```

Sequences can be copied by slicing:

```
new_l = l[:]
```

How can I find the methods or attributes of an object?

For an instance `x` of a user-defined class, `dir(x)` returns an alphabetized list of the names containing the instance attributes and methods and attributes defined by its class.

How can my code discover the name of an object?

Generally speaking, it can’t, because objects don’t really have names. Essentially, assignment always binds a name to a value; the same is true of `def` and `class` statements, but in that case the value is a callable. Consider the following code:

```
>>> class A:
...     pass
...
>>> B = A
>>> a = B()
>>> b = a
>>> print(b)
<__main__.A object at 0x16D07CC>
```

```
>>> print(a)
<__main__.A object at 0x16D07CC>
```

Arguably the class has a name: even though it is bound to two names and invoked through the name `B` the created instance is still reported as an instance of class `A`. However, it is impossible to say whether the instance's name is `a` or `b`, since both names are bound to the same value.

Generally speaking it should not be necessary for your code to "know the names" of particular values. Unless you are deliberately writing introspective programs, this is usually an indication that a change of approach might be beneficial.

In comp.lang.python, Fredrik Lundh once gave an excellent analogy in answer to this question:

> The same way as you get the name of that cat you found on your porch: the cat (object) itself cannot tell you its name, and it doesn't really care – so the only way to find out what it's called is to ask all your neighbours (namespaces) if it's their cat (object)…
>
> ….and don't be surprised if you'll find that it's known by many names, or no name at all!

What's up with the comma operator's precedence?

Comma is not an operator in Python. Consider this session:

```
>>> "a" in "b", "a"
(False, 'a')
```

Since the comma is not an operator, but a separator between expressions the above is evaluated as if you had entered:

```
("a" in "b"), "a"
```

not:

```
"a" in ("b", "a")
```

The same is true of the various assignment operators (`=`, `+=` etc). They are not truly operators but syntactic delimiters in assignment statements.

Is there an equivalent of C’s “?:” ternary operator?

Yes, there is. The syntax is as follows:

```
[on_true] if [expression] else [on_false]

x, y = 50, 25
small = x if x < y else y
```

Before this syntax was introduced in Python 2.5, a common idiom was to use logical operators:

```
[expression] and [on_true] or [on_false]
```

However, this idiom is unsafe, as it can give wrong results when *on_true* has a false boolean value. Therefore, it is always better to use the `... if ... else ...` form.

Is it possible to write obfuscated one-liners in Python?

Yes. Usually this is done by nesting `lambda` within **`lambda`**. See the following three examples, slightly adapted from Ulf Bartelt:

```
from functools import reduce
```

```
# Primes < 1000
print(list(filter(None,map(lambda y:y*reduce(lambda x,y:x*y!=0,
map(lambda
x,y=y:y%x,range(2,int(pow(y,0.5)+1))),1),range(2,1000)))))

# First 10 Fibonacci numbers
print(list(map(lambda x,f=lambda x,f:(f(x-1,f)+f(x-2,f)) if x>1
else 1:
f(x,f), range(10))))

# Mandelbrot set
print((lambda Ru,Ro,Iu,Io,IM,Sx,Sy:reduce(lambda
x,y:x+'\n'+y,map(lambda y,
Iu=Iu,Io=Io,Ru=Ru,Ro=Ro,Sy=Sy,L=lambda
yc,Iu=Iu,Io=Io,Ru=Ru,Ro=Ro,i=IM,
Sx=Sx,Sy=Sy:reduce(lambda x,y:x+y,map(lambda
x,xc=Ru,yc=yc,Ru=Ru,Ro=Ro,
i=i,Sx=Sx,F=lambda xc,yc,x,y,k,f=lambda xc,yc,x,y,k,f:(k<=0)or
(x*x+y*y
>=4.0) or 1+f(xc,yc,x*x-y*y+xc,2.0*x*y+yc,k-
1,f):f(xc,yc,x,y,k,f):chr(
64+F(Ru+x*(Ro-Ru)/Sx,yc,0,0,i)),range(Sx))):L(Iu+y*(Io-
Iu)/Sy),range(Sy
))))(-2.1, 0.7, -1.2, 1.2, 30, 80, 24))
#    \___ ___/  \___ ___/  |   |   |__ lines on screen
#        V          V      |   |______ columns on screen
#        |          |      |__________ maximum of "iterations"
#        |          |_________________ range on y axis
#        |____________________________ range on x axis
```

Don’t try this at home, kids!

What does the slash(/) in the parameter list of a function mean?

A slash in the argument list of a function denotes that the parameters prior to it are positional-only. Positional-only parameters are the ones without an externally usable name. Upon calling a function that accepts positional-only parameters, arguments are mapped to parameters based solely on their position. For example, `divmod()` is a function that accepts positional-only parameters. Its documentation looks like this:

```
>>> help(divmod)
Help on built-in function divmod in module builtins:

divmod(x, y, /)
    Return the tuple (x//y, x%y).  Invariant: div*y + mod == x.
```

The slash at the end of the parameter list means that both parameters are positional-only. Thus, calling `divmod()` with keyword arguments would lead to an error:

```
>>> divmod(x=3, y=4)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: divmod() takes no keyword arguments
```

Numbers and strings

How do I specify hexadecimal and octal integers?

To specify an octal digit, precede the octal value with a zero, and then a lower or uppercase “o”. For example, to set the variable “a” to the octal value “10” (8 in decimal), type:

```
>>> a = 0o10
>>> a
8
```

Hexadecimal is just as easy. Simply precede the hexadecimal number with a zero, and then a lower or uppercase “x”. Hexadecimal digits can be specified in lower or uppercase. For example, in the Python interpreter:

```
>>> a = 0xa5
>>> a
165
>>> b = 0XB2
>>> b
178
```

Why does -22 // 10 return -3?

It’s primarily driven by the desire that `i % j` have the same sign as `j`. If you want that, and also want:

```
i == (i // j) * j + (i % j)
```

then integer division has to return the floor. C also requires that identity to hold, and then compilers that truncate `i // j` need to make `i % j` have the same sign as `i`.

There are few real use cases for `i % j` when `j` is negative. When `j` is positive, there are many, and in virtually all of them it’s more useful for `i % j` to be `>= 0`. If the clock says 10 now, what did it say 200 hours ago? `-190 % 12 == 2` is useful; `-190 % 12 == -10` is a bug waiting to bite.

How do I get int literal attribute instead of SyntaxError?

Trying to lookup an `int` literal attribute in the normal manner gives a `SyntaxError` because the period is seen as a decimal point:

```
>>> 1.__class__
  File "<stdin>", line 1
  1.__class__
   ^
SyntaxError: invalid decimal literal
```

The solution is to separate the literal from the period with either a space or parentheses.

```
>>> 1 .__class__
<class 'int'>
>>> (1).__class__
<class 'int'>
```

How do I convert a string to a number?

For integers, use the built-in `int()` type constructor, e.g. `int('144') == 144`. Similarly, `float()` converts to a floating-point number, e.g. `float('144') == 144.0`.

By default, these interpret the number as decimal, so that `int('0144') == 144` holds true, and `int('0x144')` raises `ValueError`. `int(string, base)` takes the base to convert from as a second optional argument, so `int( '0x144', 16) == 324`. If the base is specified as 0, the number is interpreted using Python’s rules: a leading ‘0o’ indicates octal, and ‘0x’ indicates a hex number.

Do not use the built-in function `eval()` if all you need is to convert strings to numbers. `eval()` will be significantly slower and it presents a security risk: someone could pass you a Python expression that might have unwanted side effects. For example, someone could pass `__import__('os').system("rm -rf $HOME")` which would erase your home directory.

`eval()` also has the effect of interpreting numbers as Python expressions, so that e.g. `eval('09')` gives a syntax error because Python does not allow leading ‘0’ in a decimal number (except ‘0’).

How do I convert a number to a string?

To convert, e.g., the number `144` to the string `'144'`, use the built-in type constructor **`str()`**. If you want a hexadecimal or octal representation, use the built-in functions **`hex()`** or **`oct()`**. For fancy formatting, see the f-strings and Format String Syntax sections, e.g. `"{:04d}".format(144)` yields `'0144'` and `"{:.3f}".format(1.0/3.0)` yields `'0.333'`.

How do I modify a string in place?

You can’t, because strings are immutable. In most situations, you should simply construct a new string from the various parts you want to assemble it from. However, if you need an object with the ability to modify in-place unicode data, try using an **`io.StringIO`** object or the **`array`** module:

```
>>> import io
>>> s = "Hello, world"
>>> sio = io.StringIO(s)
>>> sio.getvalue()
'Hello, world'
>>> sio.seek(7)
7
>>> sio.write("there!")
6
>>> sio.getvalue()
'Hello, there!'

>>> import array
>>> a = array.array('w', s)
>>> print(a)
array('w', 'Hello, world')
>>> a[0] = 'y'
>>> print(a)
array('w', 'yello, world')
>>> a.tounicode()
'yello, world'
```

How do I use strings to call functions/methods?

There are various techniques.

- The best is to use a dictionary that maps strings to functions. The primary advantage of this technique is that the strings do not need to match the names of the functions. This is also the primary technique used to emulate a case construct:

  ```
  def a():
      pass

  def b():
      pass

  dispatch = {'go': a, 'stop': b}  # Note lack of parens for
  funcs

  dispatch[get_input()]()  # Note trailing parens to call
  function
  ```

- Use the built-in function `getattr()`:

  ```
  import foo
  getattr(foo, 'bar')()
  ```

  Note that `getattr()` works on any object, including classes, class instances, modules, and so on.

  This is used in several places in the standard library, like this:

  ```
  class Foo:
      def do_foo(self):
          ...

      def do_bar(self):
          ...

  f = getattr(foo_instance, 'do_' + opname)
  f()
  ```

- Use `locals()` to resolve the function name:

  ```
  def myFunc():
      print("hello")

  fname = "myFunc"

  f = locals()[fname]
  f()
  ```

Is there an equivalent to Perl's `chomp()` for removing trailing newlines from strings?

You can use `S.rstrip("\r\n")` to remove all occurrences of any line terminator from the end of the string `S` without removing other trailing whitespace. If the string `S` represents more than one line, with several empty lines at the end, the line terminators for all the blank lines will be removed:

```
>>> lines = ("line 1 \r\n"
...          "\r\n"
...          "\r\n")
>>> lines.rstrip("\n\r")
'line 1 '
```

Since this is typically only desired when reading text one line at a time, using `S.rstrip()` this way works well.

Is there a `scanf()` or `sscanf()` equivalent?

Not as such.

For simple input parsing, the easiest approach is usually to split the line into whitespace-delimited words using the `split()` method of string objects and then convert decimal strings to numeric values using `int()` or `float()`. **`split()`** supports an optional "sep" parameter which is useful if the line uses something other than whitespace as a separator.

For more complicated input parsing, regular expressions are more powerful than C’s `sscanf` and better suited for the task.

What does `UnicodeDecodeError` or `UnicodeEncodeError` error mean?

See the Unicode HOWTO.

Can I end a raw string with an odd number of backslashes?

A raw string ending with an odd number of backslashes will escape the string’s quote:

```
>>> r'C:\this\will\not\work\'
  File "<stdin>", line 1
    r'C:\this\will\not\work\'
    ^
SyntaxError: unterminated string literal (detected at line 1)
```

There are several workarounds for this. One is to use regular strings and double the backslashes:

```
>>> 'C:\\this\\will\\work\\'
'C:\\this\\will\\work\\'
```

Another is to concatenate a regular string containing an escaped backslash to the raw string:

```
>>> r'C:\this\will\work' '\\'
'C:\\this\\will\\work\\'
```

It is also possible to use `os.path.join()` to append a backslash on Windows:

```
>>> os.path.join(r'C:\this\will\work', '')
'C:\\this\\will\\work\\'
```

Note that while a backslash will “escape” a quote for the purposes of determining where the raw string ends, no escaping occurs when interpreting the value of the raw string. That is, the backslash remains present in the value of the raw string:

```
>>> r'backslash\'preserved'
"backslash\\'preserved"
```

Also see the specification in the language reference.

Performance

My program is too slow. How do I speed it up?

That’s a tough one, in general. First, here are a list of things to remember before diving further:

- Performance characteristics vary across Python implementations. This FAQ focuses on CPython.
- Behaviour can vary across operating systems, especially when talking about I/O or multi-threading.
- You should always find the hot spots in your program *before* attempting to optimize any code (see the **`profile`** module).
- Writing benchmark scripts will allow you to iterate quickly when searching for improvements (see the **`timeit`** module).
- It is highly recommended to have good code coverage (through unit testing or any other technique) before potentially introducing regressions hidden in sophisticated optimizations.

That being said, there are many tricks to speed up Python code. Here are some general principles which go a long way towards reaching acceptable performance levels:

- Making your algorithms faster (or changing to faster ones) can yield much larger benefits than trying to sprinkle micro-optimization tricks all over your code.
- Use the right data structures. Study documentation for the Built-in Types and the `collections` module.
- When the standard library provides a primitive for doing something, it is likely (although not guaranteed) to be faster than any alternative you may come up with. This is doubly true for primitives written in C, such as builtins and some extension types. For example, be sure to use either the `list.sort()` built-in method or the related `sorted()` function to do sorting (and see the Sorting Techniques for examples of moderately advanced usage).
- Abstractions tend to create indirections and force the interpreter to work more. If the levels of indirection outweigh the amount of useful work done, your program will be slower. You should avoid excessive abstraction, especially under the form of tiny functions or methods (which are also often detrimental to readability).

If you have reached the limit of what pure Python can allow, there are tools to take you further away. For example, Cython [https://cython.org] can compile a slightly modified version of Python code into a C extension, and can be used on many different platforms. Cython can take advantage of compilation (and optional type annotations) to make your code significantly faster than when interpreted. If you are confident in your C programming skills, you can also write a C extension module yourself.

See also

The wiki page devoted to performance tips [https://wiki.python.org/moin/PythonSpeed/PerformanceTips].

What is the most efficient way to concatenate many strings together?

`str` and `bytes` objects are immutable, therefore concatenating many strings together is inefficient as each concatenation creates a new object. In the general case, the total runtime cost is quadratic in the total string length.

To accumulate many `str` objects, the recommended idiom is to place them into a list and call `str.join()` at the end:

```
chunks = []
for s in my_strings:
    chunks.append(s)
result = ''.join(chunks)
```

(another reasonably efficient idiom is to use `io.StringIO`)

To accumulate many `bytes` objects, the recommended idiom is to extend a `bytearray` object using in-place concatenation (the `+=` operator):

```
result = bytearray()
for b in my_bytes_objects:
    result += b
```

Sequences (Tuples/Lists)

How do I convert between tuples and lists?

The type constructor `tuple(seq)` converts any sequence (actually, any iterable) into a tuple with the same items in the same order.

For example, `tuple([1, 2, 3])` yields `(1, 2, 3)` and `tuple('abc')` yields `('a', 'b', 'c')`. If the argument is a tuple, it does not make a copy but returns the same object, so it is cheap to call `tuple()` when you aren't sure that an object is already a tuple.

The type constructor `list(seq)` converts any sequence or iterable into a list with the same items in the same order. For example, `list((1, 2, 3))` yields `[1, 2, 3]` and `list('abc')` yields `['a', 'b', 'c']`. If the argument is a list, it makes a copy just like `seq[:]` would.

What's a negative index?

Python sequences are indexed with positive numbers and negative numbers. For positive numbers 0 is the first index 1 is the second index and so forth. For negative indices -1 is the last index and -2 is the penultimate (next to last) index and so forth. Think of `seq[-n]` as the same as `seq[len(seq)-n]`.

Using negative indices can be very convenient. For example `S[:-1]` is all of the string except for its last character, which is useful for removing the trailing newline from a string.

How do I iterate over a sequence in reverse order?

Use the `reversed()` built-in function:

```
for x in reversed(sequence):
    ...  # do something with x ...
```

This won't touch your original sequence, but build a new copy with reversed order to iterate over.

How do you remove duplicates from a list?

See the Python Cookbook for a long discussion of many ways to do this:

https://code.activestate.com/recipes/52560/

If you don’t mind reordering the list, sort it and then scan from the end of the list, deleting duplicates as you go:

```
if mylist:
    mylist.sort()
    last = mylist[-1]
    for i in range(len(mylist)-2, -1, -1):
        if last == mylist[i]:
            del mylist[i]
        else:
            last = mylist[i]
```

If all elements of the list may be used as set keys (i.e. they are all hashable) this is often faster

```
mylist = list(set(mylist))
```

This converts the list into a set, thereby removing duplicates, and then back into a list.

How do you remove multiple items from a list

As with removing duplicates, explicitly iterating in reverse with a delete condition is one possibility. However, it is easier and faster to use slice replacement with an implicit or explicit forward iteration. Here are three variations.:

```
mylist[:] = filter(keep_function, mylist)
mylist[:] = (x for x in mylist if keep_condition)
mylist[:] = [x for x in mylist if keep_condition]
```

The list comprehension may be fastest.

How do you make an array in Python?

Use a list:

```
["this", 1, "is", "an", "array"]
```

Lists are equivalent to C or Pascal arrays in their time complexity; the primary difference is that a Python list can contain objects of many different types.

The `array` module also provides methods for creating arrays of fixed types with compact representations, but they are slower to index than lists. Also note that NumPy [https://numpy.org/] and other third party packages define array-like structures with various characteristics as well.

To get Lisp-style linked lists, you can emulate *cons cells* using tuples:

```
lisp_list = ("like",  ("this",  ("example", None) ) )
```

If mutability is desired, you could use lists instead of tuples. Here the analogue of a Lisp *car* is `lisp_list[0]` and the analogue of *cdr* is `lisp_list[1]`. Only do this if you’re sure you really need to, because it’s usually a lot slower than using Python lists.

How do I create a multidimensional list?

You probably tried to make a multidimensional array like this:

```
>>> A = [[None] * 2] * 3
```

This looks correct if you print it:

```
>>> A
[[None, None], [None, None], [None, None]]
```

But when you assign a value, it shows up in multiple places:

```
>>> A[0][0] = 5
>>> A
[[5, None], [5, None], [5, None]]
```

The reason is that replicating a list with `*` doesn't create copies, it only creates references to the existing objects. The `*3` creates a list containing 3 references to the same list of length two. Changes to one row will show in all rows, which is almost certainly not what you want.

The suggested approach is to create a list of the desired length first and then fill in each element with a newly created list:

```
A = [None] * 3
for i in range(3):
    A[i] = [None] * 2
```

This generates a list containing 3 different lists of length two. You can also use a list comprehension:

```
w, h = 2, 3
A = [[None] * w for i in range(h)]
```

Or, you can use an extension that provides a matrix datatype; NumPy [https://numpy.org/] is the best known.

How do I apply a method or function to a sequence of objects?

To call a method or function and accumulate the return values is a list, a list comprehension is an elegant solution:

```
result = [obj.method() for obj in mylist]

result = [function(obj) for obj in mylist]
```

To just run the method or function without saving the return values, a plain `for` loop will suffice:

```
for obj in mylist:
    obj.method()

for obj in mylist:
    function(obj)
```

Why does a_tuple[i] += [‘item’] raise an exception when the addition works?

This is because of a combination of the fact that augmented assignment operators are *assignment* operators, and the difference between mutable and immutable objects in Python.

This discussion applies in general when augmented assignment operators are applied to elements of a tuple that point to mutable objects, but we’ll use a `list` and `+=` as our exemplar.

If you wrote:

```
>>> a_tuple = (1, 2)
>>> a_tuple[0] += 1
Traceback (most recent call last):
   ...
TypeError: 'tuple' object does not support item assignment
```

The reason for the exception should be immediately clear: `1` is added to the object `a_tuple[0]` points to (`1`), producing the result object, `2`, but when we attempt to assign the result of the computation, `2`, to element `0` of the tuple, we get an error because we can’t change what an element of a tuple points to.

Under the covers, what this augmented assignment statement is doing is approximately this:

```
>>> result = a_tuple[0] + 1
>>> a_tuple[0] = result
Traceback (most recent call last):
  ...
TypeError: 'tuple' object does not support item assignment
```

It is the assignment part of the operation that produces the error, since a tuple is immutable.

When you write something like:

```
>>> a_tuple = (['foo'], 'bar')
>>> a_tuple[0] += ['item']
Traceback (most recent call last):
  ...
TypeError: 'tuple' object does not support item assignment
```

The exception is a bit more surprising, and even more surprising is the fact that even though there was an error, the append worked:

```
>>> a_tuple[0]
['foo', 'item']
```

To see why this happens, you need to know that (a) if an object implements an `__iadd__()` magic method, it gets called when the `+=` augmented assignment is executed, and its return value is what gets used in the assignment statement; and (b) for lists, **`__iadd__()`** is equivalent to calling `extend()` on the list and returning the list. That’s why we say that for lists, `+=` is a “shorthand” for `list.extend()`:

```
>>> a_list = []
>>> a_list += [1]
>>> a_list
[1]
```

This is equivalent to:

```
>>> result = a_list.__iadd__([1])
>>> a_list = result
```

The object pointed to by a_list has been mutated, and the pointer to the mutated object is assigned back to `a_list`. The end result of the assignment is a no-op, since it is a pointer to the same object that `a_list` was previously pointing to, but the assignment still happens.

Thus, in our tuple example what is happening is equivalent to:

```
>>> result = a_tuple[0].__iadd__(['item'])
>>> a_tuple[0] = result
Traceback (most recent call last):
  ...
TypeError: 'tuple' object does not support item assignment
```

The **`__iadd__()`** succeeds, and thus the list is extended, but even though `result` points to the same object that `a_tuple[0]` already points to, that final assignment still results in an error, because tuples are immutable.

I want to do a complicated sort: can you do a Schwartzian Transform in Python?

The technique, attributed to Randal Schwartz of the Perl community, sorts the elements of a list by a metric which maps each element to its “sort value”. In Python, use the `key` argument for the `list.sort()` method:

```
Isorted = L[:]
Isorted.sort(key=lambda s: int(s[10:15]))
```

How can I sort one list by values from another list?

Merge them into an iterator of tuples, sort the resulting list, and then pick out the element you want.

```
>>> list1 = ["what", "I'm", "sorting", "by"]
>>> list2 = ["something", "else", "to", "sort"]
>>> pairs = zip(list1, list2)
>>> pairs = sorted(pairs)
>>> pairs
[("I'm", 'else'), ('by', 'sort'), ('sorting', 'to'), ('what',
'something')]
>>> result = [x[1] for x in pairs]
>>> result
['else', 'sort', 'to', 'something']
```

Objects

What is a class?

A class is the particular object type created by executing a class statement. Class objects are used as templates to create instance objects, which embody both the data (attributes) and code (methods) specific to a datatype.

A class can be based on one or more other classes, called its base class(es). It then inherits the attributes and methods of its base classes. This allows an object model to be successively refined by inheritance. You might have a generic `Mailbox` class that provides basic accessor methods for a mailbox, and subclasses such as `MboxMailbox`, `MaildirMailbox`, `OutlookMailbox` that handle various specific mailbox formats.

What is a method?

A method is a function on some object `x` that you normally call as `x.name(arguments...)`. Methods are defined as functions inside the class definition:

```
class C:
    def meth(self, arg):
        return arg * 2 + self.attribute
```

What is self?

Self is merely a conventional name for the first argument of a method. A method defined as `meth(self, a, b, c)` should be called as `x.meth(a, b, c)` for some instance `x` of the class in which the definition occurs; the called method will think it is called as `meth(x, a, b, c)`.

See also Why must 'self' be used explicitly in method definitions and calls?.

How do I check if an object is an instance of a given class or of a subclass of it?

Use the built-in function **`isinstance(obj, cls)`**. You can check if an object is an instance of any of a number of classes by providing a tuple instead of a single class, e.g. `isinstance(obj, (class1, class2, ...))`, and can also check whether an object is one of Python's built-in types, e.g. `isinstance(obj, str)` or `isinstance(obj, (int, float, complex))`.

Note that **`isinstance()`** also checks for virtual inheritance from an abstract base class. So, the test will return `True` for a registered class even if hasn't directly or indirectly inherited from it. To test for "true inheritance", scan the MRO of the class:

```
from collections.abc import Mapping

class P:
     pass

class C(P):
    pass
```

```
Mapping.register(P)

>>> c = C()
>>> isinstance(c, C)        # direct
True
>>> isinstance(c, P)        # indirect
True
>>> isinstance(c, Mapping)  # virtual
True

# Actual inheritance chain
>>> type(c).__mro__
(<class 'C'>, <class 'P'>, <class 'object'>)

# Test for "true inheritance"
>>> Mapping in type(c).__mro__
False
```

Note that most programs do not use [`isinstance()`] on user-defined classes very often. If you are developing the classes yourself, a more proper object-oriented style is to define methods on the classes that encapsulate a particular behaviour, instead of checking the object's class and doing a different thing based on what class it is. For example, if you have a function that does something:

```
def search(obj):
    if isinstance(obj, Mailbox):
        ...  # code to search a mailbox
    elif isinstance(obj, Document):
        ...  # code to search a document
    elif ...
```

A better approach is to define a `search()` method on all the classes and just call it:

```
class Mailbox:
    def search(self):
        ...  # code to search a mailbox
```

```
class Document:
    def search(self):
        ...  # code to search a document

obj.search()
```

What is delegation?

Delegation is an object oriented technique (also called a design pattern). Let’s say you have an object `x` and want to change the behaviour of just one of its methods. You can create a new class that provides a new implementation of the method you’re interested in changing and delegates all other methods to the corresponding method of `x`.

Python programmers can easily implement delegation. For example, the following class implements a class that behaves like a file but converts all written data to uppercase:

```
class UpperOut:

    def __init__(self, outfile):
        self._outfile = outfile

    def write(self, s):
        self._outfile.write(s.upper())

    def __getattr__(self, name):
        return getattr(self._outfile, name)
```

Here the `UpperOut` class redefines the `write()` method to convert the argument string to uppercase before calling the underlying `self._outfile.write()` method. All other methods are delegated to the underlying `self._outfile` object. The delegation is accomplished via the `__getattr__()` method; consult the language reference for more information about controlling attribute access.

Note that for more general cases delegation can get trickier. When attributes must be set as well as retrieved, the class must define a `__setattr__()` method too, and it must do so carefully. The basic implementation of `__setattr__()` is roughly equivalent to the following:

```
class X:
    ...
    def __setattr__(self, name, value):
        self.__dict__[name] = value
    ...
```

Many `__setattr__()` implementations call `object.__setattr__()` to set an attribute on self without causing infinite recursion:

```
class X:
    def __setattr__(self, name, value):
        # Custom logic here...
        object.__setattr__(self, name, value)
```

Alternatively, it is possible to set attributes by inserting entries into `self.__dict__` directly.

How do I call a method defined in a base class from a derived class that extends it?

Use the built-in `super()` function:

```
class Derived(Base):
    def meth(self):
        super().meth()  # calls Base.meth
```

In the example, `super()` will automatically determine the instance from which it was called (the `self` value), look up the method resolution order (MRO) with `type(self).__mro__`, and return the next in line after `Derived` in the MRO: `Base`.

How can I organize my code to make it easier to change the base class?

You could assign the base class to an alias and derive from the alias. Then all you have to change is the value assigned to the alias. Incidentally, this trick is also handy if you want to decide dynamically (e.g. depending on availability of resources) which base class to use. Example:

```
class Base:
    ...

BaseAlias = Base

class Derived(BaseAlias):
    ...
```

How do I create static class data and static class methods?

Both static data and static methods (in the sense of C++ or Java) are supported in Python.

For static data, simply define a class attribute. To assign a new value to the attribute, you have to explicitly use the class name in the assignment:

```
class C:
    count = 0   # number of times C.__init__ called

    def __init__(self):
        C.count = C.count + 1

    def getcount(self):
        return C.count  # or return self.count
```

`c.count` also refers to `C.count` for any `c` such that `isinstance(c, C)` holds, unless overridden by `c` itself or by some class on the base-class search path from `c.__class__` back to `C`.

Caution: within a method of C, an assignment like `self.count = 42` creates a new and unrelated instance named “count” in `self`’s own dict. Rebinding of a class-static data name must always specify the class whether inside a method or not:

```
C.count = 314
```

Static methods are possible:

```
class C:
    @staticmethod
    def static(arg1, arg2, arg3):
        # No 'self' parameter!
        ...
```

However, a far more straightforward way to get the effect of a static method is via a simple module-level function:

```
def getcount():
    return C.count
```

If your code is structured so as to define one class (or tightly related class hierarchy) per module, this supplies the desired encapsulation.

How can I overload constructors (or methods) in Python?

This answer actually applies to all methods, but the question usually comes up first in the context of constructors.

In C++ you’d write

```
class C {
    C() { cout << "No arguments\n"; }
    C(int i) { cout << "Argument is " << i << "\n"; }
}
```

In Python you have to write a single constructor that catches all cases using default arguments. For example:

```
class C:
    def __init__(self, i=None):
        if i is None:
            print("No arguments")
        else:
            print("Argument is", i)
```

This is not entirely equivalent, but close enough in practice.

You could also try a variable-length argument list, e.g.

```
def __init__(self, *args):
    ...
```

The same approach works for all method definitions.

I try to use __spam and I get an error about _SomeClassName__spam.

Variable names with double leading underscores are “mangled” to provide a simple but effective way to define class private variables. Any identifier of the form `__spam` (at least two leading underscores, at most one trailing underscore) is textually replaced with `_classname__spam`, where `classname` is the current class name with any leading underscores stripped.

The identifier can be used unchanged within the class, but to access it outside the class, the mangled name must be used:

```
class A:
    def __one(self):
        return 1
    def two(self):
        return 2 * self.__one()
```

```
class B(A):
    def three(self):
        return 3 * self._A__one()

four = 4 * A()._A__one()
```

In particular, this does not guarantee privacy since an outside user can still deliberately access the private attribute; many Python programmers never bother to use private variable names at all.

See also

The private name mangling specifications for details and special cases.

My class defines __del__ but it is not called when I delete the object.

There are several possible reasons for this.

The `del` statement does not necessarily call `__del__()` – it simply decrements the object's reference count, and if this reaches zero `__del__()` is called.

If your data structures contain circular links (e.g. a tree where each child has a parent reference and each parent has a list of children) the reference counts will never go back to zero. Once in a while Python runs an algorithm to detect such cycles, but the garbage collector might run some time after the last reference to your data structure vanishes, so your `__del__()` method may be called at an inconvenient and random time. This is inconvenient if you're trying to reproduce a problem. Worse, the order in which object's `__del__()` methods are executed is arbitrary. You can run `gc.collect()` to force a collection, but there *are* pathological cases where objects will never be collected.

Despite the cycle collector, it’s still a good idea to define an explicit `close()` method on objects to be called whenever you’re done with them. The `close()` method can then remove attributes that refer to subobjects. Don’t call **`__del__()`** directly – **`__del__()`** should call `close()` and `close()` should make sure that it can be called more than once for the same object.

Another way to avoid cyclical references is to use the `weakref` module, which allows you to point to objects without incrementing their reference count. Tree data structures, for instance, should use weak references for their parent and sibling references (if they need them!).

Finally, if your **`__del__()`** method raises an exception, a warning message is printed to `sys.stderr`.

How do I get a list of all instances of a given class?

Python does not keep track of all instances of a class (or of a built-in type). You can program the class’s constructor to keep track of all instances by keeping a list of weak references to each instance.

Why does the result of `id()` appear to be not unique?

The `id()` builtin returns an integer that is guaranteed to be unique during the lifetime of the object. Since in CPython, this is the object’s memory address, it happens frequently that after an object is deleted from memory, the next freshly created object is allocated at the same position in memory. This is illustrated by this example:

```
>>> id(1000)
13901272
>>> id(2000)
13901272
```

The two ids belong to different integer objects that are created before, and deleted immediately after execution of the `id()` call. To be sure that objects whose id you want to examine are still alive, create another reference to the object:

```
>>> a = 1000; b = 2000
>>> id(a)
13901272
>>> id(b)
13891296
```

When can I rely on identity tests with the *is* operator?

The `is` operator tests for object identity. The test `a is b` is equivalent to `id(a) == id(b)`.

The most important property of an identity test is that an object is always identical to itself, `a is a` always returns `True`. Identity tests are usually faster than equality tests. And unlike equality tests, identity tests are guaranteed to return a boolean `True` or `False`.

However, identity tests can *only* be substituted for equality tests when object identity is assured. Generally, there are three circumstances where identity is guaranteed:

1. Assignments create new names but do not change object identity. After the assignment `new = old`, it is

guaranteed that `new is old`.
2. Putting an object in a container that stores object references does not change object identity. After the list assignment `s[0] = x`, it is guaranteed that `s[0] is x`.
3. If an object is a singleton, it means that only one instance of that object can exist. After the assignments `a = None` and `b = None`, it is guaranteed that `a is b` because `None` is a singleton.

In most other circumstances, identity tests are inadvisable and equality tests are preferred. In particular, identity tests should not be used to check constants such as **`int`** and **`str`** which aren't guaranteed to be singletons:

```
>>> a = 1000
>>> b = 500
>>> c = b + 500
>>> a is c
False

>>> a = 'Python'
>>> b = 'Py'
>>> c = b + 'thon'
>>> a is c
False
```

Likewise, new instances of mutable containers are never identical:

```
>>> a = []
>>> b = []
>>> a is b
False
```

In the standard library code, you will see several common patterns for correctly using identity tests:

1. As recommended by **PEP 8** [https://peps.python.org/pep-0008/], an identity test is the preferred way to check for `None`.

This reads like plain English in code and avoids confusion with other objects that may have boolean values that evaluate to false.

2. Detecting optional arguments can be tricky when `None` is a valid input value. In those situations, you can create a singleton sentinel object guaranteed to be distinct from other objects. For example, here is how to implement a method that behaves like **`dict.pop()`**:

   ```
   _sentinel = object()

   def pop(self, key, default=_sentinel):
       if key in self:
           value = self[key]
           del self[key]
           return value
       if default is _sentinel:
           raise KeyError(key)
       return default
   ```

3. Container implementations sometimes need to augment equality tests with identity tests. This prevents the code from being confused by objects such as `float('NaN')` that are not equal to themselves.

For example, here is the implementation of **`collections.abc.Sequence.__contains__()`**:

```
def __contains__(self, value):
    for v in self:
        if v is value or v == value:
            return True
    return False
```

How can a subclass control what data is stored in an immutable instance?

When subclassing an immutable type, override the `__new__()` method instead of the `__init__()` method. The latter only runs *after* an instance is created, which is too late to alter data in an immutable instance.

All of these immutable classes have a different signature than their parent class:

```
from datetime import date

class FirstOfMonthDate(date):
    "Always choose the first day of the month"
    def __new__(cls, year, month, day):
        return super().__new__(cls, year, month, 1)

class NamedInt(int):
    "Allow text names for some numbers"
    xlat = {'zero': 0, 'one': 1, 'ten': 10}
    def __new__(cls, value):
        value = cls.xlat.get(value, value)
        return super().__new__(cls, value)

class TitleStr(str):
    "Convert str to name suitable for a URL path"
    def __new__(cls, s):
        s = s.lower().replace(' ', '-')
        s = ''.join([c for c in s if c.isalnum() or c == '-'])
        return super().__new__(cls, s)
```

The classes can be used like this:

```
>>> FirstOfMonthDate(2012, 2, 14)
FirstOfMonthDate(2012, 2, 1)
>>> NamedInt('ten')
10
>>> NamedInt(20)
20
>>> TitleStr('Blog: Why Python Rocks')
'blog-why-python-rocks'
```

How do I cache method calls?

The two principal tools for caching methods are `functools.cached_property()` and `functools.lru_cache()`. The former stores results at the instance level and the latter at the class level.

The *cached_property* approach only works with methods that do not take any arguments. It does not create a reference to the instance. The cached method result will be kept only as long as the instance is alive.

The advantage is that when an instance is no longer used, the cached method result will be released right away. The disadvantage is that if instances accumulate, so too will the accumulated method results. They can grow without bound.

The *lru_cache* approach works with methods that have hashable arguments. It creates a reference to the instance unless special efforts are made to pass in weak references.

The advantage of the least recently used algorithm is that the cache is bounded by the specified *maxsize*. The disadvantage is that instances are kept alive until they age out of the cache or until the cache is cleared.

This example shows the various techniques:

```
class Weather:
    "Lookup weather information on a government website"

    def __init__(self, station_id):
        self._station_id = station_id
        # The _station_id is private and immutable

    def current_temperature(self):
        "Latest hourly observation"
        # Do not cache this because old results
        # can be out of date.

    @cached_property
    def location(self):
        "Return the longitude/latitude coordinates of the
station"
        # Result only depends on the station_id

    @lru_cache(maxsize=20)
    def historic_rainfall(self, date, units='mm'):
        "Rainfall on a given date"
        # Depends on the station_id, date, and units.
```

The above example assumes that the *station_id* never changes. If the relevant instance attributes are mutable, the *cached_property* approach can’t be made to work because it cannot detect changes to the attributes.

To make the *lru_cache* approach work when the *station_id* is mutable, the class needs to define the `__eq__()` and `__hash__()` methods so that the cache can detect relevant attribute updates:

```
class Weather:
    "Example with a mutable station identifier"

    def __init__(self, station_id):
        self.station_id = station_id

    def change_station(self, station_id):
        self.station_id = station_id

    def __eq__(self, other):
        return self.station_id == other.station_id

    def __hash__(self):
        return hash(self.station_id)

    @lru_cache(maxsize=20)
    def historic_rainfall(self, date, units='cm'):
        'Rainfall on a given date'
        # Depends on the station_id, date, and units.
```

Modules

How do I create a .pyc file?

When a module is imported for the first time (or when the source file has changed since the current compiled file was created) a `.pyc` file containing the compiled code should be created in a `__pycache__` subdirectory of the directory containing the `.py` file. The `.pyc` file will have a filename that starts with the same name as the `.py` file, and ends with `.pyc`, with a middle component that depends on the particular `python` binary that created it. (See **PEP 3147** [https://peps.python.org/pep-3147/] for details.)

One reason that a `.pyc` file may not be created is a permissions problem with the directory containing the source file, meaning that the `__pycache__` subdirectory cannot be created. This can happen, for example, if you develop as one user but run as another, such as if you are testing with a web server.

Unless the `PYTHONDONTWRITEBYTECODE` environment variable is set, creation of a .pyc file is automatic if you're importing a module and Python has the ability (permissions, free space, etc…) to create a `__pycache__` subdirectory and write the compiled module to that subdirectory.

Running Python on a top level script is not considered an import and no `.pyc` will be created. For example, if you have a top-level module `foo.py` that imports another module `xyz.py`, when you run `foo` (by typing `python foo.py` as a shell command), a `.pyc` will be created for `xyz` because `xyz` is imported, but no `.pyc` file will be created for `foo` since `foo.py` isn't being imported.

If you need to create a `.pyc` file for `foo` – that is, to create a `.pyc` file for a module that is not imported – you can, using the **`py_compile`** and **`compileall`** modules.

The **`py_compile`** module can manually compile any module. One way is to use the `compile()` function in that module interactively:

```
>>> import py_compile
>>> py_compile.compile('foo.py')
```

This will write the `.pyc` to a `__pycache__` subdirectory in the same location as `foo.py` (or you can override that with the optional parameter `cfile`).

You can also automatically compile all files in a directory or directories using the **`compileall`** module. You can do it from the shell prompt by running `compileall.py` and providing the path of a directory containing Python files to compile:

```
python -m compileall .
```

How do I find the current module name?

A module can find out its own module name by looking at the predefined global variable `__name__`. If this has the value `'__main__'`, the program is running as a script. Many modules that are usually used by importing them also provide a command-line interface or a self-test, and only execute this code after checking `__name__`:

```
def main():
    print('Running test...')
    ...

if __name__ == '__main__':
    main()
```

How can I have modules that mutually import each other?

Suppose you have the following modules:

`foo.py`:

```
from bar import bar_var
foo_var = 1
```

`bar.py`:

```
from foo import foo_var
bar_var = 2
```

The problem is that the interpreter will perform the following steps:

- main imports `foo`
- Empty globals for `foo` are created
- `foo` is compiled and starts executing
- `foo` imports `bar`
- Empty globals for `bar` are created
- `bar` is compiled and starts executing
- `bar` imports `foo` (which is a no-op since there already is a module named `foo`)
- The import mechanism tries to read `foo_var` from `foo` globals, to set `bar.foo_var = foo.foo_var`

The last step fails, because Python isn’t done with interpreting `foo` yet and the global symbol dictionary for `foo` is still empty.

The same thing happens when you use `import foo`, and then try to access `foo.foo_var` in global code.

There are (at least) three possible workarounds for this problem.

Guido van Rossum recommends avoiding all uses of `from <module> import ...`, and placing all code inside functions. Initializations of global variables and class variables should use constants or built-in functions only. This means everything from an imported module is referenced as `<module>.<name>`.

Jim Roskind suggests performing steps in the following order in each module:

- exports (globals, functions, and classes that don't need imported base classes)
- `import` statements
- active code (including globals that are initialized from imported values).

Van Rossum doesn't like this approach much because the imports appear in a strange place, but it does work.

Matthias Urlichs recommends restructuring your code so that the recursive import is not necessary in the first place.

These solutions are not mutually exclusive.

__import__('x.y.z') returns <module 'x'>; how do I get z?

Consider using the convenience function `import_module()` from `importlib` instead:

```
z = importlib.import_module('x.y.z')
```

When I edit an imported module and reimport it, the changes don't show up. Why does this

happen?

For reasons of efficiency as well as consistency, Python only reads the module file on the first time a module is imported. If it didn't, in a program consisting of many modules where each one imports the same basic module, the basic module would be parsed and re-parsed many times. To force re-reading of a changed module, do this:

```
import importlib
import modname
importlib.reload(modname)
```

Warning: this technique is not 100% fool-proof. In particular, modules containing statements like

```
from modname import some_objects
```

will continue to work with the old version of the imported objects. If the module contains class definitions, existing class instances will *not* be updated to use the new class definition. This can result in the following paradoxical behaviour:

```
>>> import importlib
>>> import cls
>>> c = cls.C()                # Create an instance of C
>>> importlib.reload(cls)
<module 'cls' from 'cls.py'>
>>> isinstance(c, cls.C)       # isinstance is false?!?
False
```

The nature of the problem is made clear if you print out the "identity" of the class objects:

```
>>> hex(id(c.__class__))
'0x7352a0'
>>> hex(id(cls.C))
'0x4198d0'
```

Design and History FAQ

Contents

- Design and History FAQ
  - Why does Python use indentation for grouping of statements?
  - Why am I getting strange results with simple arithmetic operations?
  - Why are floating-point calculations so inaccurate?
  - Why are Python strings immutable?
  - Why must ‘self’ be used explicitly in method definitions and calls?
  - Why can’t I use an assignment in an expression?
  - Why does Python use methods for some functionality (e.g. list.index()) but functions for other (e.g. len(list))?
  - Why is join() a string method instead of a list or tuple method?
  - How fast are exceptions?
  - Why isn’t there a switch or case statement in Python?
  - Can’t you emulate threads in the interpreter instead of relying on an OS-specific thread implementation?
  - Why can’t lambda expressions contain statements?
  - Can Python be compiled to machine code, C or some other language?
  - How does Python manage memory?
  - Why doesn’t CPython use a more traditional garbage collection scheme?

- Why isn’t all memory freed when CPython exits?
- Why are there separate tuple and list data types?
- How are lists implemented in CPython?
- How are dictionaries implemented in CPython?
- Why must dictionary keys be immutable?
- Why doesn’t list.sort() return the sorted list?
- How do you specify and enforce an interface spec in Python?
- Why is there no goto?
- Why can’t raw strings (r-strings) end with a backslash?
- Why doesn’t Python have a “with” statement for attribute assignments?
- Why don’t generators support the with statement?
- Why are colons required for the if/while/def/class statements?
- Why does Python allow commas at the end of lists and tuples?

Why does Python use indentation for grouping of statements?

Guido van Rossum believes that using indentation for grouping is extremely elegant and contributes a lot to the clarity of the average Python program. Most people learn to love this feature after a while.

Since there are no begin/end brackets there cannot be a disagreement between grouping perceived by the parser and the human reader. Occasionally C programmers will encounter a fragment of code like this:

```
if (x <= y)
        x++;
        y--;
z++;
```

Only the `x++` statement is executed if the condition is true, but the indentation leads many to believe otherwise. Even experienced C programmers will sometimes stare at it a long time wondering as to why `y` is being decremented even for `x > y`.

Because there are no begin/end brackets, Python is much less prone to coding-style conflicts. In C there are many different ways to place the braces. After becoming used to reading and writing code using a particular style, it is normal to feel somewhat uneasy when reading (or being required to write) in a different one.

Many coding styles place begin/end brackets on a line by themselves. This makes programs considerably longer and wastes valuable screen space, making it harder to get a good overview of a program. Ideally, a function should fit on one screen (say, 20–30 lines). 20 lines of Python can do a lot more work than 20 lines of C. This is not solely due to the lack of begin/end brackets – the lack of declarations and the high-level data types are also responsible – but the indentation-based syntax certainly helps.

Why am I getting strange results with simple arithmetic operations?

See the next question.

Why are floating-point calculations so inaccurate?

Users are often surprised by results like this:

```
>>> 1.2 - 1.0
0.19999999999999996
```

and think it is a bug in Python. It’s not. This has little to do with Python, and much more to do with how the underlying platform handles floating-point numbers.

The **`float`** type in CPython uses a C `double` for storage. A **`float`** object’s value is stored in binary floating-point with a fixed precision (typically 53 bits) and Python uses C operations, which in turn rely on the hardware implementation in the processor, to perform floating-point operations. This means that as far as floating-point operations are concerned, Python behaves like many popular languages including C and Java.

Many numbers that can be written easily in decimal notation cannot be expressed exactly in binary floating point. For example, after:

```
>>> x = 1.2
```

the value stored for `x` is a (very good) approximation to the decimal value `1.2`, but is not exactly equal to it. On a typical machine, the actual stored value is:

```
1.0011001100110011001100110011001100110011001100110011 (binary)
```

which is exactly:

```
1.1999999999999999555910790149937383830547332763671875
(decimal)
```

The typical precision of 53 bits provides Python floats with 15–16 decimal digits of accuracy.

For a fuller explanation, please see the floating-point arithmetic chapter in the Python tutorial.

Why are Python strings immutable?

There are several advantages.

One is performance: knowing that a string is immutable means we can allocate space for it at creation time, and the storage requirements are fixed and unchanging. This is also one of the reasons for the distinction between tuples and lists.

Another advantage is that strings in Python are considered as “elemental” as numbers. No amount of activity will change the value 8 to anything else, and in Python, no amount of activity will change the string “eight” to anything else.

Why must ‘self’ be used explicitly in method definitions and calls?

The idea was borrowed from Modula-3. It turns out to be very useful, for a variety of reasons.

First, it’s more obvious that you are using a method or instance attribute instead of a local variable. Reading `self.x` or `self.meth()` makes it absolutely clear that an instance variable or method is used even if you don’t know the class definition by heart. In C++, you can sort of tell by the lack of a local variable declaration (assuming globals are rare or easily recognizable) – but in Python, there are no local variable declarations, so you’d have to look up the class definition to be sure. Some C++ and Java coding standards call for instance attributes to have an `m_` prefix, so this explicitness is still useful in those languages, too.

Second, it means that no special syntax is necessary if you want to explicitly reference or call the method from a particular class. In C++, if you want to use a method from a base class which is overridden in a derived class, you have to use the `::` operator – in Python you can write `baseclass.methodname(self, <argument list>)`. This is particularly useful for `__init__()` methods, and in general in cases where a derived class method wants to extend the base class method of the same name and thus has to call the base class method somehow.

Finally, for instance variables it solves a syntactic problem with assignment: since local variables in Python are (by definition!) those variables to which a value is assigned in a function body (and that aren't explicitly declared global), there has to be some way to tell the interpreter that an assignment was meant to assign to an instance variable instead of to a local variable, and it should preferably be syntactic (for efficiency reasons). C++ does this through declarations, but Python doesn't have declarations and it would be a pity having to introduce them just for this purpose. Using the explicit `self.var` solves this nicely. Similarly, for using instance variables, having to write `self.var` means that references to unqualified names inside a method don't have to search the instance's directories. To put it another way, local variables and instance variables live in two different namespaces, and you need to tell Python which namespace to use.

Why can't I use an assignment in an expression?

Starting in Python 3.8, you can!

Assignment expressions using the walrus operator `:=` assign a variable in an expression:

```
while chunk := fp.read(200):
   print(chunk)
```

See **PEP 572** [https://peps.python.org/pep-0572/] for more information.

Why does Python use methods for some functionality (e.g. list.index()) but functions for other (e.g. len(list))?

As Guido said:

> (a) For some operations, prefix notation just reads better than postfix – prefix (and infix!) operations have a long tradition in mathematics which likes notations where the visuals help the mathematician thinking about a problem. Compare the easy with which we rewrite a formula like x*(a+b) into x*a + x*b to the clumsiness of doing the same thing using a raw OO notation.
>
> (b) When I read code that says len(x) I *know* that it is asking for the length of something. This tells me two things: the result is an integer, and the argument is some kind of container. To the contrary, when I read x.len(), I have to already know that x is some kind of container implementing an interface or inheriting from a class that has a standard len(). Witness the confusion we occasionally have when a class that is not implementing a mapping has a get() or keys() method, or something that isn't a file has a write() method.

—https://mail.python.org/pipermail/python-3000/2006-November/004643.html

Why is join() a string method instead of a list or tuple method?

Strings became much more like other standard types starting in Python 1.6, when methods were added which give the same functionality that has always been available using the functions of the string module. Most of these new methods have been widely accepted, but the one which appears to make some programmers feel uncomfortable is:

```
", ".join(['1', '2', '4', '8', '16'])
```

which gives the result:

```
"1, 2, 4, 8, 16"
```

There are two common arguments against this usage.

The first runs along the lines of: “It looks really ugly using a method of a string literal (string constant)”, to which the answer is that it might, but a string literal is just a fixed value. If the methods are to be allowed on names bound to strings there is no logical reason to make them unavailable on literals.

The second objection is typically cast as: “I am really telling a sequence to join its members together with a string constant”. Sadly, you aren’t. For some reason there seems to be much less difficulty with having `split()` as a string method, since in that case it is easy to see that

```
"1, 2, 4, 8, 16".split(", ")
```

is an instruction to a string literal to return the substrings delimited by the given separator (or, by default, arbitrary runs of white space).

`join()` is a string method because in using it you are telling the separator string to iterate over a sequence of strings and insert itself between adjacent elements. This method can be used with any argument which obeys the rules for sequence objects, including any new classes you might define yourself. Similar methods exist for bytes and bytearray objects.

How fast are exceptions?

A `try`/`except` block is extremely efficient if no exceptions are raised. Actually catching an exception is expensive. In versions of Python prior to 2.0 it was common to use this idiom:

```
try:
    value = mydict[key]
except KeyError:
    mydict[key] = getvalue(key)
    value = mydict[key]
```

This only made sense when you expected the dict to have the key almost all the time. If that wasn't the case, you coded it like this:

```
if key in mydict:
    value = mydict[key]
else:
    value = mydict[key] = getvalue(key)
```

For this specific case, you could also use `value = dict.setdefault(key, getvalue(key))`, but only if the `getvalue()` call is cheap enough because it is evaluated in all cases.

Why isn’t there a switch or case statement in Python?

In general, structured switch statements execute one block of code when an expression has a particular value or set of values. Since Python 3.10 one can easily match literal values, or constants within a namespace, with a `match ... case` statement. An older alternative is a sequence of `if... elif... elif... else`.

For cases where you need to choose from a very large number of possibilities, you can create a dictionary mapping case values to functions to call. For example:

```
functions = {'a': function_1,
             'b': function_2,
             'c': self.method_1}

func = functions[value]
func()
```

For calling methods on objects, you can simplify yet further by using the `getattr()` built-in to retrieve methods with a particular name:

```
class MyVisitor:
    def visit_a(self):
        ...

    def dispatch(self, value):
        method_name = 'visit_' + str(value)
        method = getattr(self, method_name)
        method()
```

It’s suggested that you use a prefix for the method names, such as `visit_` in this example. Without such a prefix, if values are coming from an untrusted source, an attacker would be able to call any method on your object.

Imitating switch with fallthrough, as with C's switch-case-default, is possible, much harder, and less needed.

Can't you emulate threads in the interpreter instead of relying on an OS-specific thread implementation?

Answer 1: Unfortunately, the interpreter pushes at least one C stack frame for each Python stack frame. Also, extensions can call back into Python at almost random moments. Therefore, a complete threads implementation requires thread support for C.

Answer 2: Fortunately, there is Stackless Python [https://github.com/stackless-dev/stackless/wiki], which has a completely redesigned interpreter loop that avoids the C stack.

Why can't lambda expressions contain statements?

Python lambda expressions cannot contain statements because Python's syntactic framework can't handle statements nested inside expressions. However, in Python, this is not a serious problem. Unlike lambda forms in other languages, where they add functionality, Python lambdas are only a shorthand notation if you're too lazy to define a function.

Functions are already first class objects in Python, and can be declared in a local scope. Therefore the only advantage of using a lambda instead of a locally defined function is that you don't need to invent a name for the function – but that's just a local variable to which the function object (which is exactly the same type of object that a lambda expression yields) is assigned!

Can Python be compiled to machine code, C or some other language?

Cython [https://cython.org/] compiles a modified version of Python with optional annotations into C extensions. Nuitka [https://nuitka.net/] is an up-and-coming compiler of Python into C++ code, aiming to support the full Python language.

How does Python manage memory?

The details of Python memory management depend on the implementation. The standard implementation of Python, CPython, uses reference counting to detect inaccessible objects, and another mechanism to collect reference cycles, periodically executing a cycle detection algorithm which looks for inaccessible cycles and deletes the objects involved. The **`gc`** module provides functions to perform a garbage collection, obtain debugging statistics, and tune the collector's parameters.

Other implementations (such as Jython [https://www.jython.org] or PyPy [https://pypy.org]), however, can rely on a different mechanism such as a full-blown garbage collector. This difference can cause some subtle porting problems if your Python code depends on the behavior of the reference counting implementation.

In some Python implementations, the following code (which is fine in CPython) will probably run out of file descriptors:

```
for file in very_long_list_of_files:
    f = open(file)
    c = f.read(1)
```

Indeed, using CPython's reference counting and destructor scheme, each new assignment to `f` closes the previous file. With a traditional GC, however, those file objects will only get collected (and closed) at varying and possibly long intervals.

If you want to write code that will work with any Python implementation, you should explicitly close the file or use the **`with`** statement; this will work regardless of memory management scheme:

```
for file in very_long_list_of_files:
    with open(file) as f:
        c = f.read(1)
```

Why doesn't CPython use a more traditional garbage collection scheme?

For one thing, this is not a C standard feature and hence it's not portable. (Yes, we know about the Boehm GC library. It has bits of assembler code for *most* common platforms, not for all of them, and although it is mostly transparent, it isn't completely transparent; patches are required to get Python to work with it.)

Traditional GC also becomes a problem when Python is embedded into other applications. While in a standalone Python it's fine to replace the standard `malloc()` and `free()` with versions provided by the GC library, an application embedding Python may want to have its *own* substitute for `malloc()` and `free()`, and may not want Python's. Right now, CPython works with anything that implements `malloc()` and `free()` properly.

Why isn’t all memory freed when CPython exits?

Objects referenced from the global namespaces of Python modules are not always deallocated when Python exits. This may happen if there are circular references. There are also certain bits of memory that are allocated by the C library that are impossible to free (e.g. a tool like Purify will complain about these). Python is, however, aggressive about cleaning up memory on exit and does try to destroy every single object.

If you want to force Python to delete certain things on deallocation use the `atexit` module to run a function that will force those deletions.

Why are there separate tuple and list data types?

Lists and tuples, while similar in many respects, are generally used in fundamentally different ways. Tuples can be thought of as being similar to Pascal `records` or C `structs`; they’re small collections of related data which may be of different types which are operated on as a group. For example, a Cartesian coordinate is appropriately represented as a tuple of two or three numbers.

Lists, on the other hand, are more like arrays in other languages. They tend to hold a varying number of objects all of which have the same type and which are operated on one-by-one. For example, `os.listdir('.')` returns a list of strings representing the files in the current directory. Functions which operate on this output would generally not break if you added another file or two to the directory.

Tuples are immutable, meaning that once a tuple has been created, you can’t replace any of its elements with a new value. Lists are mutable, meaning that you can always change a list’s elements. Only immutable elements can be used as dictionary keys, and hence only tuples and not lists can be used as keys.

How are lists implemented in CPython?

CPython’s lists are really variable-length arrays, not Lisp-style linked lists. The implementation uses a contiguous array of references to other objects, and keeps a pointer to this array and the array’s length in a list head structure.

This makes indexing a list `a[i]` an operation whose cost is independent of the size of the list or the value of the index.

When items are appended or inserted, the array of references is resized. Some cleverness is applied to improve the performance of appending items repeatedly; when the array must be grown, some extra space is allocated so the next few times don’t require an actual resize.

How are dictionaries implemented in CPython?

CPython’s dictionaries are implemented as resizable hash tables. Compared to B-trees, this gives better performance for lookup (the most common operation by far) under most circumstances, and the implementation is simpler.

Dictionaries work by computing a hash code for each key stored in the dictionary using the `hash()` built-in function.

The hash code varies widely depending on the key and a per-process seed; for example, `'Python'` could hash to `-539294296` while `'python'`, a string that differs by a single bit, could hash to `1142331976`. The hash code is then used to calculate a location in an internal array where the value will be stored. Assuming that you're storing keys that all have different hash values, this means that dictionaries take constant time – *O*(1), in Big-O notation – to retrieve a key.

Why must dictionary keys be immutable?

The hash table implementation of dictionaries uses a hash value calculated from the key value to find the key. If the key were a mutable object, its value could change, and thus its hash could also change. But since whoever changes the key object can't tell that it was being used as a dictionary key, it can't move the entry around in the dictionary. Then, when you try to look up the same object in the dictionary it won't be found because its hash value is different. If you tried to look up the old value it wouldn't be found either, because the value of the object found in that hash bin would be different.

If you want a dictionary indexed with a list, simply convert the list to a tuple first; the function `tuple(L)` creates a tuple with the same entries as the list `L`. Tuples are immutable and can therefore be used as dictionary keys.

Some unacceptable solutions that have been proposed:

- Hash lists by their address (object ID). This doesn't work because if you construct a new list with the same value it won't be found; e.g.:

```
mydict = {[1, 2]: '12'}
print(mydict[[1, 2]])
```

would raise a `KeyError` exception because the id of the `[1, 2]` used in the second line differs from that in the first line. In other words, dictionary keys should be compared using `==`, not using `is`.

- Make a copy when using a list as a key. This doesn’t work because the list, being a mutable object, could contain a reference to itself, and then the copying code would run into an infinite loop.

- Allow lists as keys but tell the user not to modify them. This would allow a class of hard-to-track bugs in programs when you forgot or modified a list by accident. It also invalidates an important invariant of dictionaries: every value in `d.keys()` is usable as a key of the dictionary.

- Mark lists as read-only once they are used as a dictionary key. The problem is that it’s not just the top-level object that could change its value; you could use a tuple containing a list as a key. Entering anything as a key into a dictionary would require marking all objects reachable from there as read-only – and again, self-referential objects could cause an infinite loop.

There is a trick to get around this if you need to, but use it at your own risk: You can wrap a mutable structure inside a class instance which has both a `__eq__()` and a `__hash__()` method. You must then make sure that the hash value for all such wrapper objects that reside in a dictionary (or other hash based structure), remain fixed while the object is in the dictionary (or other structure).

```
class ListWrapper:
    def __init__(self, the_list):
        self.the_list = the_list

    def __eq__(self, other):
        return self.the_list == other.the_list

    def __hash__(self):
        l = self.the_list
        result = 98767 - len(l)*555
        for i, el in enumerate(l):
            try:
                result = result + (hash(el) % 9999999) * 1001 +
i
            except Exception:
                result = (result % 7777777) + i * 333
        return result
```

Note that the hash computation is complicated by the possibility that some members of the list may be unhashable and also by the possibility of arithmetic overflow.

Furthermore it must always be the case that if `o1 == o2` (ie `o1.__eq__(o2) is True`) then `hash(o1) == hash(o2)` (ie, `o1.__hash__() == o2.__hash__()`), regardless of whether the object is in a dictionary or not. If you fail to meet these restrictions dictionaries and other hash based structures will misbehave.

In the case of **`ListWrapper`**, whenever the wrapper object is in a dictionary the wrapped list must not change to avoid anomalies. Don’t do this unless you are prepared to think hard about the requirements and the consequences of not meeting them correctly. Consider yourself warned.

Why doesn’t list.sort() return the sorted list?

In situations where performance matters, making a copy of the list just to sort it would be wasteful. Therefore, `list.sort()` sorts the list in place. In order to remind you of that fact, it does not return the sorted list. This way, you won't be fooled into accidentally overwriting a list when you need a sorted copy but also need to keep the unsorted version around.

If you want to return a new list, use the built-in `sorted()` function instead. This function creates a new list from a provided iterable, sorts it and returns it. For example, here's how to iterate over the keys of a dictionary in sorted order:

```
for key in sorted(mydict):
    ...  # do whatever with mydict[key]...
```

How do you specify and enforce an interface spec in Python?

An interface specification for a module as provided by languages such as C++ and Java describes the prototypes for the methods and functions of the module. Many feel that compile-time enforcement of interface specifications helps in the construction of large programs.

Python 2.6 adds an `abc` module that lets you define Abstract Base Classes (ABCs). You can then use `isinstance()` and `issubclass()` to check whether an instance or a class implements a particular ABC. The `collections.abc` module defines a set of useful ABCs such as `Iterable`, `Container`, and `MutableMapping`.

For Python, many of the advantages of interface specifications can be obtained by an appropriate test discipline for components.

A good test suite for a module can both provide a regression test and serve as a module interface specification and a set of examples. Many Python modules can be run as a script to provide a simple “self test.” Even modules which use complex external interfaces can often be tested in isolation using trivial “stub” emulations of the external interface. The `doctest` and `unittest` modules or third-party test frameworks can be used to construct exhaustive test suites that exercise every line of code in a module.

An appropriate testing discipline can help build large complex applications in Python as well as having interface specifications would. In fact, it can be better because an interface specification cannot test certain properties of a program. For example, the `list.append()` method is expected to add new elements to the end of some internal list; an interface specification cannot test that your `list.append()` implementation will actually do this correctly, but it’s trivial to check this property in a test suite.

Writing test suites is very helpful, and you might want to design your code to make it easily tested. One increasingly popular technique, test-driven development, calls for writing parts of the test suite first, before you write any of the actual code. Of course Python allows you to be sloppy and not write test cases at all.

Why is there no goto?

In the 1970s people realized that unrestricted goto could lead to messy “spaghetti” code that was hard to understand and revise. In a high-level language, it is also unneeded as long as there are ways to branch (in Python, with `if` statements and `or`, `and`, and `if`/`else` expressions) and loop (with `while` and `for` statements, possibly containing `continue` and `break`).

One can also use exceptions to provide a “structured goto” that works even across function calls. Many feel that exceptions can conveniently emulate all reasonable uses of the `go` or `goto` constructs of C, Fortran, and other languages. For example:

```
class label(Exception): pass  # declare a label

try:
    ...
    if condition: raise label()  # goto label
    ...
except label:  # where to goto
    pass
...
```

This doesn’t allow you to jump into the middle of a loop, but that’s usually considered an abuse of `goto` anyway. Use sparingly.

Why can’t raw strings (r-strings) end with a backslash?

More precisely, they can’t end with an odd number of backslashes: the unpaired backslash at the end escapes the closing quote character, leaving an unterminated string.

Raw strings were designed to ease creating input for processors (chiefly regular expression engines) that want to do their own backslash escape processing. Such processors consider an unmatched trailing backslash to be an error anyway, so raw strings disallow that. In return, they allow you to pass on the string quote character by escaping it with a backslash. These rules work well when r-strings are used for their intended purpose.

If you’re trying to build Windows pathnames, note that all Windows system calls accept forward slashes too:

```
f = open("/mydir/file.txt")  # works fine!
```

If you’re trying to build a pathname for a DOS command, try e.g. one of

```
dir = r"\this\is\my\dos\dir" "\\"
dir = r"\this\is\my\dos\dir\ "[:-1]
dir = "\\this\\is\\my\\dos\\dir\\"
```

Why doesn’t Python have a “with” statement for attribute assignments?

Python has a `with` statement that wraps the execution of a block, calling code on the entrance and exit from the block. Some languages have a construct that looks like this:

```
with obj:
    a = 1               # equivalent to obj.a = 1
    total = total + 1   # obj.total = obj.total + 1
```

In Python, such a construct would be ambiguous.

Other languages, such as Object Pascal, Delphi, and C++, use static types, so it’s possible to know, in an unambiguous way, what member is being assigned to. This is the main point of static typing – the compiler *always* knows the scope of every variable at compile time.

Python uses dynamic types. It is impossible to know in advance which attribute will be referenced at runtime. Member attributes may be added or removed from objects on the fly. This makes it impossible to know, from a simple reading, what attribute is being referenced: a local one, a global one, or a member attribute?

For instance, take the following incomplete snippet:

```
def foo(a):
    with a:
        print(x)
```

The snippet assumes that `a` must have a member attribute called `x`. However, there is nothing in Python that tells the interpreter this. What should happen if `a` is, let us say, an integer? If there is a global variable named `x`, will it be used inside the **`with`** block? As you see, the dynamic nature of Python makes such choices much harder.

The primary benefit of **`with`** and similar language features (reduction of code volume) can, however, easily be achieved in Python by assignment. Instead of:

```
function(args).mydict[index][index].a = 21
function(args).mydict[index][index].b = 42
function(args).mydict[index][index].c = 63
```

write this:

```
ref = function(args).mydict[index][index]
ref.a = 21
ref.b = 42
ref.c = 63
```

This also has the side-effect of increasing execution speed because name bindings are resolved at run-time in Python, and the second version only needs to perform the resolution once.

Similar proposals that would introduce syntax to further reduce code volume, such as using a ‘leading dot’, have been rejected in favour of explicitness (see https://mail.python.org/pipermail/python-ideas/2016-May/040070.html).

Why don’t generators support the with statement?

For technical reasons, a generator used directly as a context manager would not work correctly. When, as is most common, a generator is used as an iterator run to completion, no closing is needed. When it is, wrap it as `contextlib.closing(generator)` in the `with` statement.

Why are colons required for the if/while/def/class statements?

The colon is required primarily to enhance readability (one of the results of the experimental ABC language). Consider this:

```
if a == b
    print(a)
```

versus

```
if a == b:
    print(a)
```

Notice how the second one is slightly easier to read. Notice further how a colon sets off the example in this FAQ answer; it’s a standard usage in English.

Another minor reason is that the colon makes it easier for editors with syntax highlighting; they can look for colons to decide when indentation needs to be increased instead of having to do a more elaborate parsing of the program text.

Why does Python allow commas at the end of lists and tuples?

Python lets you add a trailing comma at the end of lists, tuples, and dictionaries:

```
[1, 2, 3,]
('a', 'b', 'c',)
d = {
    "A": [1, 5],
    "B": [6, 7],  # last trailing comma is optional but good
style
}
```

There are several reasons to allow this.

When you have a literal value for a list, tuple, or dictionary spread across multiple lines, it’s easier to add more elements because you don’t have to remember to add a comma to the previous line. The lines can also be reordered without creating a syntax error.

Accidentally omitting the comma can lead to errors that are hard to diagnose. For example:

```
x = [
  "fee",
  "fie"
  "foo",
  "fum"
]
```

This list looks like it has four elements, but it actually contains three: “fee”, “fiefoo” and “fum”. Always adding the comma avoids this source of error.

Allowing the trailing comma may also make programmatic code generation easier.

Library and Extension FAQ

Contents

- Library and Extension FAQ
  - General Library Questions
    - How do I find a module or application to perform task X?
    - Where is the math.py (socket.py, regex.py, etc.) source file?
    - How do I make a Python script executable on Unix?
    - Is there a curses/termcap package for Python?
    - Is there an equivalent to C’s onexit() in Python?
    - Why don’t my signal handlers work?
  - Common tasks
    - How do I test a Python program or component?
    - How do I create documentation from doc strings?
    - How do I get a single keypress at a time?
  - Threads
    - How do I program using threads?
    - None of my threads seem to run: why?
    - How do I parcel out work among a bunch of worker threads?
    - What kinds of global value mutation are thread-safe?
    - Can’t we get rid of the Global Interpreter Lock?
  - Input and Output

    - How do I delete a file? (And other file questions…)
    - How do I copy a file?
    - How do I read (or write) binary data?
    - I can’t seem to use os.read() on a pipe created with os.popen(); why?
    - How do I access the serial (RS232) port?
    - Why doesn’t closing sys.stdout (stdin, stderr) really close it?
  - Network/Internet Programming
    - What WWW tools are there for Python?
    - What module should I use to help with generating HTML?
    - How do I send mail from a Python script?
    - How do I avoid blocking in the connect() method of a socket?
  - Databases
    - Are there any interfaces to database packages in Python?
    - How do you implement persistent objects in Python?
  - Mathematics and Numerics
    - How do I generate random numbers in Python?

General Library Questions

How do I find a module or application to perform task X?

Check the Library Reference to see if there’s a relevant standard library module. (Eventually you’ll learn what’s in the standard library and will be able to skip this step.)

For third-party packages, search the Python Package Index [https://pypi.org] or try Google [https://www.google.com] or another web search engine. Searching for “Python” plus a keyword or two for your topic of interest will usually find something helpful.

Where is the math.py (socket.py, regex.py, etc.) source file?

If you can’t find a source file for a module it may be a built-in or dynamically loaded module implemented in C, C++ or other compiled language. In this case you may not have the source file or it may be something like `mathmodule.c`, somewhere in a C source directory (not on the Python Path).

There are (at least) three kinds of modules in Python:

1. modules written in Python (.py);
2. modules written in C and dynamically loaded (.dll, .pyd, .so, .sl, etc);
3. modules written in C and linked with the interpreter; to get a list of these, type:

   ```
   import sys
   print(sys.builtin_module_names)
   ```

How do I make a Python script executable on Unix?

You need to do two things: the script file’s mode must be executable and the first line must begin with `#!` followed by the path of the Python interpreter.

The first is done by executing `chmod +x scriptfile` or perhaps `chmod 755 scriptfile`.

The second can be done in a number of ways. The most straightforward way is to write

```
#!/usr/local/bin/python
```

as the very first line of your file, using the pathname for where the Python interpreter is installed on your platform.

If you would like the script to be independent of where the Python interpreter lives, you can use the **env** program. Almost all Unix variants support the following, assuming the Python interpreter is in a directory on the user's **`PATH`**:

```
#!/usr/bin/env python
```

Don't do this for CGI scripts. The **`PATH`** variable for CGI scripts is often very minimal, so you need to use the actual absolute pathname of the interpreter.

Occasionally, a user's environment is so full that the **/usr/bin/env** program fails; or there's no env program at all. In that case, you can try the following hack (due to Alex Rezinsky):

```
#! /bin/sh
""":"
exec python $0 ${1+"$@"}
"""
```

The minor disadvantage is that this defines the script's __doc__ string. However, you can fix that by adding

```
__doc__ = """...Whatever..."""
```

Is there a curses/termcap package for Python?

For Unix variants: The standard Python source distribution comes with a curses module in the Modules [https://github.com/python/cpython/tree/3.14/Modules] subdirectory, though it’s not compiled by default. (Note that this is not available in the Windows distribution – there is no curses module for Windows.)

The `curses` module supports basic curses features as well as many additional functions from ncurses and SYSV curses such as colour, alternative character set support, pads, and mouse support. This means the module isn’t compatible with operating systems that only have BSD curses, but there don’t seem to be any currently maintained OSes that fall into this category.

Is there an equivalent to C’s onexit() in Python?

The `atexit` module provides a register function that is similar to C’s **`onexit()`**.

Why don’t my signal handlers work?

The most common problem is that the signal handler is declared with the wrong argument list. It is called as

```
handler(signum, frame)
```

so it should be declared with two parameters:

```
def handler(signum, frame):
    ...
```

Common tasks

How do I test a Python program or component?

Python comes with two testing frameworks. The `doctest` module finds examples in the docstrings for a module and runs them, comparing the output with the expected output given in the docstring.

The `unittest` module is a fancier testing framework modelled on Java and Smalltalk testing frameworks.

To make testing easier, you should use good modular design in your program. Your program should have almost all functionality encapsulated in either functions or class methods – and this sometimes has the surprising and delightful effect of making the program run faster (because local variable accesses are faster than global accesses). Furthermore the program should avoid depending on mutating global variables, since this makes testing much more difficult to do.

The “global main logic” of your program may be as simple as

```
if __name__ == "__main__":
    main_logic()
```

at the bottom of the main module of your program.

Once your program is organized as a tractable collection of function and class behaviours, you should write test functions that exercise the behaviours. A test suite that automates a sequence of tests can be associated with each module. This sounds like a lot of work, but since Python is so terse and flexible it’s surprisingly easy. You can make coding much more pleasant and fun by writing your test functions in parallel with the “production code”, since this makes it easy to find bugs and even design flaws earlier.

“Support modules” that are not intended to be the main module of a program may include a self-test of the module.

```
if __name__ == "__main__":
    self_test()
```

Even programs that interact with complex external interfaces may be tested when the external interfaces are unavailable by using “fake” interfaces implemented in Python.

How do I create documentation from doc strings?

The `pydoc` module can create HTML from the doc strings in your Python source code. An alternative for creating API documentation purely from docstrings is epydoc [https://epydoc.sourceforge.net/]. Sphinx [https://www.sphinx-doc.org] can also include docstring content.

How do I get a single keypress at a time?

For Unix variants there are several solutions. It’s straightforward to do this using curses, but curses is a fairly large module to learn.

Threads

How do I program using threads?

Be sure to use the `threading` module and not the `_thread` module. The `threading` module builds convenient abstractions on top of the low-level primitives provided by the `_thread` module.

None of my threads seem to run: why?

As soon as the main thread exits, all threads are killed. Your main thread is running too quickly, giving the threads no time to do any work.

A simple fix is to add a sleep to the end of the program that’s long enough for all the threads to finish:

```
import threading, time

def thread_task(name, n):
    for i in range(n):
        print(name, i)

for i in range(10):
    T = threading.Thread(target=thread_task, args=(str(i), i))
    T.start()

time.sleep(10)  # <----------------------------!
```

But now (on many platforms) the threads don’t run in parallel, but appear to run sequentially, one at a time! The reason is that the OS thread scheduler doesn’t start a new thread until the previous thread is blocked.

A simple fix is to add a tiny sleep to the start of the run function:

```
def thread_task(name, n):
    time.sleep(0.001)  # <---------------------!
    for i in range(n):
        print(name, i)

for i in range(10):
    T = threading.Thread(target=thread_task, args=(str(i), i))
    T.start()

time.sleep(10)
```

Instead of trying to guess a good delay value for `time.sleep()`, it’s better to use some kind of semaphore mechanism. One idea is to use the `queue` module to create a queue object, let each thread append a token to the queue when it finishes, and let the main thread read as many tokens from the queue as there are threads.

How do I parcel out work among a bunch of worker threads?

The easiest way is to use the `concurrent.futures` module, especially the `ThreadPoolExecutor` class.

Or, if you want fine control over the dispatching algorithm, you can write your own logic manually. Use the `queue` module to create a queue containing a list of jobs. The `Queue` class maintains a list of objects and has a `.put(obj)` method that adds items to the queue and a `.get()` method to return them. The class will take care of the locking necessary to ensure that each job is handed out exactly once.

Here’s a trivial example:

```
import threading, queue, time

# The worker thread gets jobs off the queue.  When the queue is
empty, it
# assumes there will be no more work and exits.
# (Realistically workers will run until terminated.)
def worker():
    print('Running worker')
    time.sleep(0.1)
    while True:
        try:
            arg = q.get(block=False)
        except queue.Empty:
            print('Worker', threading.current_thread(), end='
')
            print('queue empty')
            break
        else:
            print('Worker', threading.current_thread(), end='
')
            print('running with argument', arg)
            time.sleep(0.5)

# Create queue
q = queue.Queue()

# Start a pool of 5 workers
for i in range(5):
    t = threading.Thread(target=worker, name='worker %i' %
(i+1))
    t.start()

# Begin adding work to the queue
for i in range(50):
    q.put(i)

# Give threads time to run
print('Main thread sleeping')
time.sleep(5)
```

When run, this will produce the following output:

```
Running worker
Running worker
Running worker
Running worker
Running worker
Main thread sleeping
Worker <Thread(worker 1, started 130283832797456)> running with
argument 0
Worker <Thread(worker 2, started 130283824404752)> running with
argument 1
Worker <Thread(worker 3, started 130283816012048)> running with
argument 2
Worker <Thread(worker 4, started 130283807619344)> running with
argument 3
Worker <Thread(worker 5, started 130283799226640)> running with
argument 4
Worker <Thread(worker 1, started 130283832797456)> running with
argument 5
...
```

Consult the module’s documentation for more details; the `Queue` class provides a featureful interface.

What kinds of global value mutation are thread-safe?

A global interpreter lock (GIL) is used internally to ensure that only one thread runs in the Python VM at a time. In general, Python offers to switch among threads only between bytecode instructions; how frequently it switches can be set via `sys.setswitchinterval()`. Each bytecode instruction and therefore all the C implementation code reached from each instruction is therefore atomic from the point of view of a Python program.

In theory, this means an exact accounting requires an exact understanding of the PVM bytecode implementation. In practice, it means that operations on shared variables of built-in data types (ints, lists, dicts, etc) that “look atomic” really are.

For example, the following operations are all atomic (L, L1, L2 are lists, D, D1, D2 are dicts, x, y are objects, i, j are ints):

```
L.append(x)
L1.extend(L2)
x = L[i]
x = L.pop()
L1[i:j] = L2
L.sort()
x = y
x.field = y
D[x] = y
D1.update(D2)
D.keys()
```

These aren’t:

```
i = i+1
L.append(L[-1])
L[i] = L[j]
D[x] = D[x] + 1
```

Operations that replace other objects may invoke those other objects' `__del__()` method when their reference count reaches zero, and that can affect things. This is especially true for the mass updates to dictionaries and lists. When in doubt, use a mutex!

Can't we get rid of the Global Interpreter Lock?

The global interpreter lock (GIL) is often seen as a hindrance to Python's deployment on high-end multiprocessor server machines, because a multi-threaded Python program effectively only uses one CPU, due to the insistence that (almost) all Python code can only run while the GIL is held.

With the approval of **PEP 703** [https://peps.python.org/pep-0703/] work is now underway to remove the GIL from the CPython implementation of Python. Initially it will be implemented as an optional compiler flag when building the interpreter, and so separate builds will be available with and without the GIL. Long-term, the hope is to settle on a single build, once the performance implications of removing the GIL are fully understood. Python 3.13 is likely to be the first release containing this work, although it may not be completely functional in this release.

The current work to remove the GIL is based on a fork of Python 3.9 with the GIL removed [https://github.com/colesbury/nogil] by Sam Gross. Prior to that, in the days of Python 1.5, Greg Stein actually implemented a comprehensive patch set (the "free threading" patches) that removed the GIL and replaced it with fine-grained locking. Adam Olsen did a similar experiment in his python-safethread [https://code.google.com/archive/p/python-safethread] project. Unfortunately, both of these earlier experiments exhibited a sharp drop in single-thread performance (at least 30% slower), due to the amount of fine-grained locking necessary to compensate for the removal of the GIL. The Python 3.9 fork is the first attempt at removing the GIL with an acceptable performance impact.

The presence of the GIL in current Python releases doesn't mean that you can't make good use of Python on multi-CPU machines! You just have to be creative with dividing the work up between multiple *processes* rather than multiple *threads*. The **`ProcessPoolExecutor`** class in the new **`concurrent.futures`** module provides an easy way of doing so; the **`multiprocessing`** module provides a lower-level API in case you want more control over dispatching of tasks.

Judicious use of C extensions will also help; if you use a C extension to perform a time-consuming task, the extension can release the GIL while the thread of execution is in the C code and allow other threads to get some work done. Some standard library modules such as **`zlib`** and **`hashlib`** already do this.

An alternative approach to reducing the impact of the GIL is to make the GIL a per-interpreter-state lock rather than truly global. This was first implemented in Python 3.12 and is available in the C API. A Python interface to it is expected in Python 3.13. The main limitation to it at the moment is likely to be 3rd party extension modules, since these must be written with multiple interpreters in mind in order to be usable, so many older extension modules will not be usable.

Input and Output

How do I delete a file? (And other file questions…)

Use `os.remove(filename)` or `os.unlink(filename)`; for documentation, see the **`os`** module. The two functions are identical; **`unlink()`** is simply the name of the Unix system call for this function.

To remove a directory, use **`os.rmdir()`**; use **`os.mkdir()`** to create one. `os.makedirs(path)` will create any intermediate directories in `path` that don't exist. `os.removedirs(path)` will remove intermediate directories as long as they're empty; if you want to delete an entire directory tree and its contents, use **`shutil.rmtree()`**.

To rename a file, use `os.rename(old_path, new_path)`.

To truncate a file, open it using `f = open(filename, "rb+")`, and use `f.truncate(offset)`; offset defaults to the current seek position. There's also `os.ftruncate(fd, offset)` for files opened with **`os.open()`**, where *fd* is the file descriptor (a small integer).

The **`shutil`** module also contains a number of functions to work on files including **`copyfile()`**, **`copytree()`**, and **`rmtree()`**.

How do I copy a file?

The **`shutil`** module contains a **`copyfile()`** function. Note that on Windows NTFS volumes, it does not copy alternate data streams [https://en.wikipedia.org/wiki/NTFS#Alternate_data_stream_(ADS)] nor resource forks [https://en.wikipedia.org/wiki/Resource_fork] on macOS HFS+ volumes, though both are now rarely used. It also doesn't copy file permissions and metadata, though using **`shutil.copy2()`** instead will preserve most (though not all) of it.

How do I read (or write) binary data?

To read or write complex binary data formats, it’s best to use the `struct` module. It allows you to take a string containing binary data (usually numbers) and convert it to Python objects; and vice versa.

For example, the following code reads two 2-byte integers and one 4-byte integer in big-endian format from a file:

```
import struct

with open(filename, "rb") as f:
    s = f.read(8)
    x, y, z = struct.unpack(">hhl", s)
```

The ‘>’ in the format string forces big-endian data; the letter ‘h’ reads one “short integer” (2 bytes), and ‘l’ reads one “long integer” (4 bytes) from the string.

For data that is more regular (e.g. a homogeneous list of ints or floats), you can also use the `array` module.

Note

To read and write binary data, it is mandatory to open the file in binary mode (here, passing `"rb"` to `open()`). If you use `"r"` instead (the default), the file will be open in text mode and `f.read()` will return `str` objects rather than `bytes` objects.

I can’t seem to use os.read() on a pipe created with os.popen(); why?

`os.read()` is a low-level function which takes a file descriptor, a small integer representing the opened file. `os.popen()`

creates a high-level file object, the same type returned by the built-in `open()` function. Thus, to read *n* bytes from a pipe *p* created with `os.popen()`, you need to use `p.read(n)`.

How do I access the serial (RS232) port?

For Win32, OSX, Linux, BSD, Jython, IronPython:

> pyserial [https://pypi.org/project/pyserial/]

For Unix, see a Usenet post by Mitch Chapman:

> https://groups.google.com/groups?selm=34A04430.CF9@ohioee.com

Why doesn't closing sys.stdout (stdin, stderr) really close it?

Python file objects are a high-level layer of abstraction on low-level C file descriptors.

For most file objects you create in Python via the built-in `open()` function, `f.close()` marks the Python file object as being closed from Python's point of view, and also arranges to close the underlying C file descriptor. This also happens automatically in `f`'s destructor, when `f` becomes garbage.

But stdin, stdout and stderr are treated specially by Python, because of the special status also given to them by C. Running `sys.stdout.close()` marks the Python-level file object as being closed, but does *not* close the associated C file descriptor.

To close the underlying C file descriptor for one of these three, you should first be sure that's what you really want to do (e.g., you may confuse extension modules trying to do I/O). If it is, use `os.close()`:

```
os.close(stdin.fileno())
os.close(stdout.fileno())
os.close(stderr.fileno())
```

Or you can use the numeric constants 0, 1 and 2, respectively.

Network/Internet Programming

What WWW tools are there for Python?

See the chapters titled Internet Protocols and Support and Internet Data Handling in the Library Reference Manual. Python has many modules that will help you build server-side and client-side web systems.

A summary of available frameworks is maintained by Paul Boddie at https://wiki.python.org/moin/WebProgramming.

What module should I use to help with generating HTML?

You can find a collection of useful links on the Web Programming wiki page [https://wiki.python.org/moin/WebProgramming].

How do I send mail from a Python script?

Use the standard library module `smtplib`.

Here’s a very simple interactive mail sender that uses it. This method will work on any host that supports an SMTP listener.

```
import sys, smtplib

fromaddr = input("From: ")
toaddrs  = input("To: ").split(',')
print("Enter message, end with ^D:")
msg = ''
while True:
    line = sys.stdin.readline()
    if not line:
        break
    msg += line

# The actual mail send
server = smtplib.SMTP('localhost')
server.sendmail(fromaddr, toaddrs, msg)
server.quit()
```

A Unix-only alternative uses sendmail. The location of the sendmail program varies between systems; sometimes it is `/usr/lib/sendmail`, sometimes `/usr/sbin/sendmail`. The sendmail manual page will help you out. Here's some sample code:

```
import os

SENDMAIL = "/usr/sbin/sendmail"  # sendmail location
p = os.popen("%s -t -i" % SENDMAIL, "w")
p.write("To: receiver@example.com\n")
p.write("Subject: test\n")
p.write("\n")  # blank line separating headers from body
p.write("Some text\n")
p.write("some more text\n")
sts = p.close()
if sts != 0:
    print("Sendmail exit status", sts)
```

How do I avoid blocking in the connect() method of a socket?

The `select` module is commonly used to help with asynchronous I/O on sockets.

To prevent the TCP connect from blocking, you can set the socket to non-blocking mode. Then when you do the `connect()`, you will either connect immediately (unlikely) or get an exception that contains the error number as `.errno`. `errno.EINPROGRESS` indicates that the connection is in progress, but hasn't finished yet. Different OSes will return different values, so you're going to have to check what's returned on your system.

You can use the `connect_ex()` method to avoid creating an exception. It will just return the errno value. To poll, you can call `connect_ex()` again later – `0` or `errno.EISCONN` indicate that you're connected – or you can pass this socket to `select.select()` to check if it's writable.

Note

The `asyncio` module provides a general purpose single-threaded and concurrent asynchronous library, which can be used for writing non-blocking network code. The third-party Twisted [https://twisted.org/] library is a popular and feature-rich alternative.

Databases

Are there any interfaces to database packages in Python?

Yes.

Interfaces to disk-based hashes such as **`DBM`** and **`GDBM`** are also included with standard Python. There is also the **`sqlite3`** module, which provides a lightweight disk-based relational database.

Support for most relational databases is available. See the DatabaseProgramming wiki page [https://wiki.python.org/moin/DatabaseProgramming] for details.

How do you implement persistent objects in Python?

The **`pickle`** library module solves this in a very general way (though you still can’t store things like open files, sockets or windows), and the **`shelve`** library module uses pickle and (g)dbm to create persistent mappings containing arbitrary Python objects.

Mathematics and Numerics

How do I generate random numbers in Python?

The standard module **`random`** implements a random number generator. Usage is simple:

```
import random
random.random()
```

This returns a random floating-point number in the range [0, 1).

There are also many other specialized generators in this module, such as:

- `randrange(a, b)` chooses an integer in the range [a, b).

- `uniform(a, b)` chooses a floating-point number in the range [a, b).
- `normalvariate(mean, sdev)` samples the normal (Gaussian) distribution.

Some higher-level functions operate on sequences directly, such as:

- `choice(S)` chooses a random element from a given sequence.
- `shuffle(L)` shuffles a list in-place, i.e. permutes it randomly.

There’s also a `Random` class you can instantiate to create independent multiple random number generators.

Extending/Embedding FAQ

Contents

- Extending/Embedding FAQ
    - Can I create my own functions in C?
    - Can I create my own functions in C++?
    - Writing C is hard; are there any alternatives?
    - How can I execute arbitrary Python statements from C?
    - How can I evaluate an arbitrary Python expression from C?
    - How do I extract C values from a Python object?
    - How do I use Py_BuildValue() to create a tuple of arbitrary length?
    - How do I call an object’s method from C?
    - How do I catch the output from PyErr_Print() (or anything that prints to stdout/stderr)?
    - How do I access a module written in Python from C?
    - How do I interface to C++ objects from Python?
    - I added a module using the Setup file and the make fails; why?
    - How do I debug an extension?
    - I want to compile a Python module on my Linux system, but some files are missing. Why?
    - How do I tell “incomplete input” from “invalid input”?
    - How do I find undefined g++ symbols __builtin_new or __pure_virtual?
    - Can I create an object class with some methods implemented in C and others in Python (e.g.

through inheritance)?

Can I create my own functions in C?

Yes, you can create built-in modules containing functions, variables, exceptions and even new types in C. This is explained in the document Extending and Embedding the Python Interpreter.

Most intermediate or advanced Python books will also cover this topic.

Can I create my own functions in C++?

Yes, using the C compatibility features found in C++. Place `extern "C" { ... }` around the Python include files and put `extern "C"` before each function that is going to be called by the Python interpreter. Global or static C++ objects with constructors are probably not a good idea.

Writing C is hard; are there any alternatives?

There are a number of alternatives to writing your own C extensions, depending on what you’re trying to do. Recommended third party tools offer both simpler and more sophisticated approaches to creating C and C++ extensions for Python.

How can I execute arbitrary Python statements from C?

The highest-level function to do this is `PyRun_SimpleString()` which takes a single string argument to be executed in the context of the module `__main__` and returns `0` for success and `-1` when an exception occurred (including `SyntaxError`). If you want more control, use `PyRun_String()`; see the source for `PyRun_SimpleString()` in `Python/pythonrun.c`.

How can I evaluate an arbitrary Python expression from C?

Call the function `PyRun_String()` from the previous question with the start symbol `Py_eval_input`; it parses an expression, evaluates it and returns its value.

How do I extract C values from a Python object?

That depends on the object's type. If it's a tuple, `PyTuple_Size()` returns its length and `PyTuple_GetItem()` returns the item at a specified index. Lists have similar functions, `PyList_Size()` and `PyList_GetItem()`.

For bytes, `PyBytes_Size()` returns its length and `PyBytes_AsStringAndSize()` provides a pointer to its value and its length. Note that Python bytes objects may contain null bytes so C's **`strlen()`** should not be used.

To test the type of an object, first make sure it isn't `NULL`, and then use `PyBytes_Check()`, `PyTuple_Check()`, `PyList_Check()`, etc.

There is also a high-level API to Python objects which is provided by the so-called 'abstract' interface – read `Include/abstract.h` for further details. It allows interfacing with any kind of Python sequence using calls like

`PySequence_Length()`, `PySequence_GetItem()`, etc. as well as many other useful protocols such as numbers (`PyNumber_Index()` et al.) and mappings in the PyMapping APIs.

How do I use Py_BuildValue() to create a tuple of arbitrary length?

You can’t. Use `PyTuple_Pack()` instead.

How do I call an object’s method from C?

The `PyObject_CallMethod()` function can be used to call an arbitrary method of an object. The parameters are the object, the name of the method to call, a format string like that used with `Py_BuildValue()`, and the argument values:

```
PyObject *
PyObject_CallMethod(PyObject *object, const char *method_name,
                    const char *arg_format, ...);
```

This works for any object that has methods – whether built-in or user-defined. You are responsible for eventually `Py_DECREF()`‘ing the return value.

To call, e.g., a file object’s “seek” method with arguments 10, 0 (assuming the file object pointer is “f”):

```
res = PyObject_CallMethod(f, "seek", "(ii)", 10, 0);
if (res == NULL) {
        ... an exception occurred ...
}
else {
        Py_DECREF(res);
}
```

Note that since `PyObject_CallObject()` *always* wants a tuple for the argument list, to call a function without arguments, pass “()” for the format, and to call a function with one argument, surround the argument in parentheses, e.g. “(i)”.

How do I catch the output from PyErr_Print() (or anything that prints to stdout/stderr)?

In Python code, define an object that supports the `write()` method. Assign this object to `sys.stdout` and `sys.stderr`. Call print_error, or just allow the standard traceback mechanism to work. Then, the output will go wherever your `write()` method sends it.

The easiest way to do this is to use the `io.StringIO` class:

```
>>> import io, sys
>>> sys.stdout = io.StringIO()
>>> print('foo')
>>> print('hello world!')
>>> sys.stderr.write(sys.stdout.getvalue())
foo
hello world!
```

A custom object to do the same would look like this:

```
>>> import io, sys
>>> class StdoutCatcher(io.TextIOBase):
...     def __init__(self):
...         self.data = []
...     def write(self, stuff):
...         self.data.append(stuff)
...
>>> import sys
>>> sys.stdout = StdoutCatcher()
>>> print('foo')
>>> print('hello world!')
>>> sys.stderr.write(''.join(sys.stdout.data))
foo
hello world!
```

How do I access a module written in Python from C?

You can get a pointer to the module object as follows:

```
module = PyImport_ImportModule("<modulename>");
```

If the module hasn’t been imported yet (i.e. it is not yet present in `sys.modules`), this initializes the module; otherwise it simply returns the value of `sys.modules["<modulename>"]`. Note that it doesn’t enter the module into any namespace – it only ensures it has been initialized and is stored in `sys.modules`.

You can then access the module’s attributes (i.e. any name defined in the module) as follows:

```
attr = PyObject_GetAttrString(module, "<attrname>");
```

Calling `PyObject_SetAttrString()` to assign to variables in the module also works.

How do I interface to C++ objects from Python?

Depending on your requirements, there are many approaches. To do this manually, begin by reading the “Extending and Embedding” document. Realize that for the Python run-time system, there isn’t a whole lot of difference between C and C++ – so the strategy of building a new Python type around a C structure (pointer) type will also work for C++ objects.

For C++ libraries, see Writing C is hard; are there any alternatives?.

I added a module using the Setup file and the make fails; why?

Setup must end in a newline, if there is no newline there, the build process fails. (Fixing this requires some ugly shell script hackery, and this bug is so minor that it doesn’t seem worth the effort.)

How do I debug an extension?

When using GDB with dynamically loaded extensions, you can’t set a breakpoint in your extension until your extension is loaded.

In your `.gdbinit` file (or interactively), add the command:

```
br _PyImport_LoadDynamicModule
```

Then, when you run GDB:

```
$ gdb /local/bin/python
gdb) run myscript.py
gdb) continue # repeat until your extension is loaded
gdb) finish   # so that your extension is loaded
gdb) br myfunction.c:50
gdb) continue
```

I want to compile a Python module on my Linux system, but some files are missing. Why?

Most packaged versions of Python omit some files required for compiling Python extensions.

For Red Hat, install the python3-devel RPM to get the necessary files.

For Debian, run `apt-get install python3-dev`.

How do I tell “incomplete input” from “invalid input”?

Sometimes you want to emulate the Python interactive interpreter’s behavior, where it gives you a continuation prompt when the input is incomplete (e.g. you typed the start of an “if” statement or you didn’t close your parentheses or triple string quotes), but it gives you a syntax error message immediately when the input is invalid.

In Python you can use the `codeop` module, which approximates the parser’s behavior sufficiently. IDLE uses this, for example.

The easiest way to do it in C is to call `PyRun_InteractiveLoop()` (perhaps in a separate thread) and let the Python interpreter handle the input for you. You can also set the `PyOS_ReadlineFunctionPointer()` to point at your custom input function. See `Modules/readline.c` and `Parser/myreadline.c` for more hints.

How do I find undefined g++ symbols __builtin_new or __pure_virtual?

To dynamically load g++ extension modules, you must recompile Python, relink it using g++ (change LINKCC in the

Python Modules Makefile), and link your extension module using g++ (e.g., `g++ -shared -o mymodule.so mymodule.o`).

Can I create an object class with some methods implemented in C and others in Python (e.g. through inheritance)?

Yes, you can inherit from built-in classes such as **`int`**, **`list`**, **`dict`**, etc.

The Boost Python Library (BPL, https://www.boost.org/libs/python/doc/index.html) provides a way of doing this from C++ (i.e. you can inherit from an extension class written in C++ using the BPL).

Python on Windows FAQ

Contents

- Python on Windows FAQ
  - How do I run a Python program under Windows?
  - How do I make Python scripts executable?
  - Why does Python sometimes take so long to start?
  - How do I make an executable from a Python script?
  - Is a `*.pyd` file the same as a DLL?
  - How can I embed Python into a Windows application?
  - How do I keep editors from inserting tabs into my Python source?
  - How do I check for a keypress without blocking?
  - How do I solve the missing api-ms-win-crt-runtime-l1-1-0.dll error?

How do I run a Python program under Windows?

This is not necessarily a straightforward question. If you are already familiar with running programs from the Windows command line then everything will seem obvious; otherwise, you might need a little more guidance.

Unless you use some sort of integrated development environment, you will end up *typing* Windows commands into what is referred to as a “Command prompt window”. Usually you can create such a window from your search bar by searching for `cmd`. You should be able to recognize when you have started such a window because you will see a Windows “command prompt”, which usually looks like this:

```
C:\>
```

The letter may be different, and there might be other things after it, so you might just as easily see something like:

```
D:\YourName\Projects\Python>
```

depending on how your computer has been set up and what else you have recently done with it. Once you have started such a window, you are well on the way to running Python programs.

You need to realize that your Python scripts have to be processed by another program called the Python *interpreter*. The interpreter reads your script, compiles it into bytecodes, and then executes the bytecodes to run your program. So, how do you arrange for the interpreter to handle your Python?

First, you need to make sure that your command window recognises the word “py” as an instruction to start the interpreter. If you have opened a command window, you should try entering the command `py` and hitting return:

```
C:\Users\YourName> py
```

You should then see something like:

```
Python 3.6.4 (v3.6.4:d48eceb, Dec 19 2017, 06:04:45) [MSC
v.1900 32 bit (Intel)] on win32
Type "help", "copyright", "credits" or "license" for more
information.
>>>
```

You have started the interpreter in “interactive mode”. That means you can enter Python statements or expressions interactively and have them executed or evaluated while you wait. This is one of Python’s strongest features. Check it by entering a few expressions of your choice and seeing the results:

```
>>> print("Hello")
Hello
>>> "Hello" * 3
'HelloHelloHello'
```

Many people use the interactive mode as a convenient yet highly programmable calculator. When you want to end your interactive Python session, call the `exit()` function or hold the `Ctrl` key down while you enter a `Z`, then hit the “`Enter`” key to get back to your Windows command prompt.

You may also find that you have a Start-menu entry such as *Start* ‣ *Programs* ‣ *Python 3.x* ‣ *Python (command line)* that results in you seeing the `>>>` prompt in a new window. If so, the window will disappear after you call the `exit()` function or enter the `Ctrl-Z` character; Windows is running a single “python” command in the window, and closes it when you terminate the interpreter.

Now that we know the `py` command is recognized, you can give your Python script to it. You’ll have to give either an absolute or a relative path to the Python script. Let’s say your Python script is located in your desktop and is named `hello.py`, and your command prompt is nicely opened in your home directory so you’re seeing something similar to:

```
C:\Users\YourName>
```

So now you’ll ask the `py` command to give your script to Python by typing `py` followed by your script path:

```
C:\Users\YourName> py Desktop\hello.py
hello
```

How do I make Python scripts executable?

On Windows, the standard Python installer already associates the .py extension with a file type (Python.File) and gives that file type an open command that runs the interpreter (`D:\Program Files\Python\python.exe "%1" %*`). This is enough to make scripts executable from the command prompt as ‘foo.py’. If you’d rather be able to execute the script by simple typing ‘foo’ with no extension you need to add .py to the PATHEXT environment variable.

Why does Python sometimes take so long to start?

Usually Python starts very quickly on Windows, but occasionally there are bug reports that Python suddenly begins to take a long time to start up. This is made even more puzzling because Python will work fine on other Windows systems which appear to be configured identically.

The problem may be caused by a misconfiguration of virus checking software on the problem machine. Some virus scanners have been known to introduce startup overhead of two orders of magnitude when the scanner is configured to monitor all reads from the filesystem. Try checking the configuration of virus scanning software on your systems to ensure that they are indeed configured identically. McAfee, when configured to scan all file system read activity, is a particular offender.

How do I make an executable from a Python script?

See How can I create a stand-alone binary from a Python script? for a list of tools that can be used to make executables.

Is a `*.pyd` file the same as a DLL?

Yes, .pyd files are dll's, but there are a few differences. If you have a DLL named `foo.pyd`, then it must have a function `PyInit_foo()`. You can then write Python "import foo", and Python will search for foo.pyd (as well as foo.py, foo.pyc) and if it finds it, will attempt to call `PyInit_foo()` to initialize it. You do not link your .exe with foo.lib, as that would cause Windows to require the DLL to be present.

Note that the search path for foo.pyd is PYTHONPATH, not the same as the path that Windows uses to search for foo.dll. Also, foo.pyd need not be present to run your program, whereas if you linked your program with a dll, the dll is required. Of course, foo.pyd is required if you want to say `import foo`. In a DLL, linkage is declared in the source code with `__declspec(dllexport)`. In a .pyd, linkage is defined in a list of available functions.

How can I embed Python into a Windows application?

Embedding the Python interpreter in a Windows app can be summarized as follows:

1. Do **not** build Python into your .exe file directly. On Windows, Python must be a DLL to handle importing

modules that are themselves DLL's. (This is the first key undocumented fact.) Instead, link to `python`*`NN`*`.dll`; it is typically installed in `C:\Windows\System`. *NN* is the Python version, a number such as "33" for Python 3.3.

You can link to Python in two different ways. Load-time linking means linking against `python`*`NN`*`.lib`, while run-time linking means linking against `python`*`NN`*`.dll`. (General note: `python`*`NN`*`.lib` is the so-called "import lib" corresponding to `python`*`NN`*`.dll`. It merely defines symbols for the linker.)

Run-time linking greatly simplifies link options; everything happens at run time. Your code must load `python`*`NN`*`.dll` using the Windows `LoadLibraryEx()` routine. The code must also use access routines and data in `python`*`NN`*`.dll` (that is, Python's C API's) using pointers obtained by the Windows `GetProcAddress()` routine. Macros can make using these pointers transparent to any C code that calls routines in Python's C API.

2. If you use SWIG, it is easy to create a Python "extension module" that will make the app's data and methods available to Python. SWIG will handle just about all the grungy details for you. The result is C code that you link *into* your .exe file (!) You do **not** have to create a DLL file, and this also simplifies linking.

3. SWIG will create an init function (a C function) whose name depends on the name of the extension module. For example, if the name of the module is leo, the init function will be called initleo(). If you use SWIG shadow classes, as you should, the init function will be called initleoc(). This initializes a mostly hidden helper class used by the shadow class.

The reason you can link the C code in step 2 into your .exe file is that calling the initialization function is equivalent to importing the module into Python! (This is the second key undocumented fact.)

4. In short, you can use the following code to initialize the Python interpreter with your extension module.

   ```
   #include <Python.h>
   ...
   Py_Initialize();  // Initialize Python.
   initmyAppc();  // Initialize (import) the helper class.
   PyRun_SimpleString("import myApp");  // Import the shadow
   class.
   ```

5. There are two problems with Python’s C API which will become apparent if you use a compiler other than MSVC, the compiler used to build pythonNN.dll.

   Problem 1: The so-called “Very High Level” functions that take `FILE *` arguments will not work in a multi-compiler environment because each compiler’s notion of a `struct FILE` will be different. From an implementation standpoint these are very low level functions.

   Problem 2: SWIG generates the following code when generating wrappers to void functions:

   ```
   Py_INCREF(Py_None);
   _resultobj = Py_None;
   return _resultobj;
   ```

   Alas, Py_None is a macro that expands to a reference to a complex data structure called _Py_NoneStruct inside pythonNN.dll. Again, this code will fail in a mult-compiler environment. Replace such code by:

   ```
   return Py_BuildValue("");
   ```

It may be possible to use SWIG's `%typemap` command to make the change automatically, though I have not been able to get this to work (I'm a complete SWIG newbie).

6. Using a Python shell script to put up a Python interpreter window from inside your Windows app is not a good idea; the resulting window will be independent of your app's windowing system. Rather, you (or the wxPythonWindow class) should create a “native” interpreter window. It is easy to connect that window to the Python interpreter. You can redirect Python's i/o to _any_ object that supports read and write, so all you need is a Python object (defined in your extension module) that contains read() and write() methods.

How do I keep editors from inserting tabs into my Python source?

The FAQ does not recommend using tabs, and the Python style guide, **PEP 8** [https://peps.python.org/pep-0008/], recommends 4 spaces for distributed Python code; this is also the Emacs python-mode default.

Under any editor, mixing tabs and spaces is a bad idea. MSVC is no different in this respect, and is easily configured to use spaces: Take *Tools* ‣ *Options* ‣ *Tabs*, and for file type “Default” set “Tab size” and “Indent size” to 4, and select the “Insert spaces” radio button.

Python raises `IndentationError` or `TabError` if mixed tabs and spaces are causing problems in leading whitespace. You may also run the `tabnanny` module to check a directory tree in batch mode.

How do I check for a keypress without blocking?

Use the `msvcrt` module. This is a standard Windows-specific extension module. It defines a function `kbhit()` which checks whether a keyboard hit is present, and `getch()` which gets one character without echoing it.

How do I solve the missing api-ms-win-crt-runtime-l1-1-0.dll error?

This can occur on Python 3.5 and later when using Windows 8.1 or earlier without all updates having been installed. First ensure your operating system is supported and is up to date, and if that does not resolve the issue, visit the Microsoft support page [https://support.microsoft.com/en-us/help/3118401/] for guidance on manually installing the C Runtime update.

Graphic User Interface FAQ

Contents

- Graphic User Interface FAQ
  - General GUI Questions
  - What GUI toolkits exist for Python?
  - Tkinter questions
    - How do I freeze Tkinter applications?
    - Can I have Tk events handled while waiting for I/O?
    - I can’t get key bindings to work in Tkinter: why?

General GUI Questions

What GUI toolkits exist for Python?

Standard builds of Python include an object-oriented interface to the Tcl/Tk widget set, called tkinter. This is probably the easiest to install (since it comes included with most binary distributions [https://www.python.org/downloads/] of Python) and use. For more info about Tk, including pointers to the source, see the Tcl/Tk home page [https://www.tcl.tk]. Tcl/Tk is fully portable to the macOS, Windows, and Unix platforms.

Depending on what platform(s) you are aiming at, there are also several alternatives. A list of cross-platform [https://wiki.python.org/moin/GuiProgramming#Cross-Platform_Frameworks] and platform-specific [https://wiki.python.org/moin/GuiProgramming#Platform-specific_Frameworks] GUI frameworks can be found on the python wiki.

Tkinter questions

How do I freeze Tkinter applications?

Freeze is a tool to create stand-alone applications. When freezing Tkinter applications, the applications will not be truly stand-alone, as the application will still need the Tcl and Tk libraries.

One solution is to ship the application with the Tcl and Tk libraries, and point to them at run-time using the `TCL_LIBRARY` and `TK_LIBRARY` environment variables.

Various third-party freeze libraries such as py2exe and cx_Freeze have handling for Tkinter applications built-in.

Can I have Tk events handled while waiting for I/O?

On platforms other than Windows, yes, and you don’t even need threads! But you’ll have to restructure your I/O code a bit. Tk has the equivalent of Xt’s `XtAddInput()` call, which allows you to register a callback function which will be called from the Tk mainloop when I/O is possible on a file descriptor. See File Handlers.

I can’t get key bindings to work in Tkinter: why?

An often-heard complaint is that event handlers bound to events with the **`bind()`** method don’t get handled even when the appropriate key is pressed.

The most common cause is that the widget to which the binding applies doesn’t have “keyboard focus”. Check out the Tk documentation for the focus command. Usually a widget is given the keyboard focus by clicking in it (but not for labels; see the takefocus option).

“Why is Python Installed on my Computer?” FAQ

What is Python?

Python is a programming language. It’s used for many different applications. It’s used in some high schools and colleges as an introductory programming language because Python is easy to learn, but it’s also used by professional software developers at places such as Google, NASA, and Lucasfilm Ltd.

If you wish to learn more about Python, start with the Beginner’s Guide to Python [https://wiki.python.org/moin/BeginnersGuide].

Why is Python installed on my machine?

If you find Python installed on your system but don’t remember installing it, there are several possible ways it could have gotten there.

- Perhaps another user on the computer wanted to learn programming and installed it; you’ll have to figure out who’s been using the machine and might have installed it.
- A third-party application installed on the machine might have been written in Python and included a Python installation. There are many such applications, from GUI programs to network servers and administrative scripts.

- Some Windows machines also have Python installed. At this writing we’re aware of computers from Hewlett-Packard and Compaq that include Python. Apparently some of HP/Compaq’s administrative tools are written in Python.
- Many Unix-compatible operating systems, such as macOS and some Linux distributions, have Python installed by default; it’s included in the base installation.

Can I delete Python?

That depends on where Python came from.

If someone installed it deliberately, you can remove it without hurting anything. On Windows, use the Add/Remove Programs icon in the Control Panel.

If Python was installed by a third-party application, you can also remove it, but that application will no longer work. You should use that application’s uninstaller rather than removing Python directly.

If Python came with your operating system, removing it is not recommended. If you remove it, whatever tools were written in Python will no longer run, and some of them might be important to you. Reinstalling the whole system would then be required to fix things again.

Deprecations

Pending removal in Python 3.15

- The import system:
    - Setting `__cached__` on a module while failing to set `__spec__.cached` is deprecated. In Python 3.15, **`__cached__`** will cease to be set or take into consideration by the import system or standard library. (gh-97879 [https://github.com/python/cpython/issues/97879])
    - Setting `__package__` on a module while failing to set `__spec__.parent` is deprecated. In Python 3.15, **`__package__`** will cease to be set or take into consideration by the import system or standard library. (gh-97879 [https://github.com/python/cpython/issues/97879])
- `ctypes`:
    - The undocumented **`ctypes.SetPointerType()`** function has been deprecated since Python 3.13.
- `http.server`:
    - The obsolete and rarely used `CGIHTTPRequestHandler` has been deprecated since Python 3.13. No direct replacement exists. *Anything* is better than CGI to interface a web server with a request handler.
    - The **`--cgi`** flag to the **python -m http.server** command-line interface has been deprecated since Python 3.13.
- `importlib`:
    - `load_module()` method: use `exec_module()` instead.
- `locale`:
    - The `getdefaultlocale()` function has been deprecated since Python 3.11. Its removal was originally

planned for Python 3.13 (gh-90817 [https://github.com/python/cpython/issues/90817]), but has been postponed to Python 3.15. Use **`getlocale()`**, **`setlocale()`**, and **`getencoding()`** instead. (Contributed by Hugo van Kemenade in gh-111187 [https://github.com/python/cpython/issues/111187].)

- **`pathlib`**:
  - **`PurePath.is_reserved()`** has been deprecated since Python 3.13. Use **`os.path.isreserved()`** to detect reserved paths on Windows.
- **`platform`**:
  - **`java_ver()`** has been deprecated since Python 3.13. This function is only useful for Jython support, has a confusing API, and is largely untested.
- **`sysconfig`**:
  - The *check_home* argument of **`sysconfig.is_python_build()`** has been deprecated since Python 3.12.
- **`threading`**:
  - **`RLock()`** will take no arguments in Python 3.15. Passing any arguments has been deprecated since Python 3.14, as the Python version does not permit any arguments, but the C version allows any number of positional or keyword arguments, ignoring every argument.
- **`types`**:
  - **`types.CodeType`**: Accessing **`co_lnotab`** was deprecated in **PEP 626** [https://peps.python.org/pep-0626/] since 3.10 and was planned to be removed in 3.12, but it only got a proper **`DeprecationWarning`** in 3.12. May be removed in 3.15. (Contributed by Nikita Sobolev in gh-101866 [https://github.com/python/cpython/issues/101866].)
- **`typing`**:
  - The undocumented keyword argument syntax for creating **`NamedTuple`** classes (for example, `Point = NamedTuple("Point", x=int, y=int)`) has been

    deprecated since Python 3.13. Use the class-based syntax or the functional syntax instead.
  - When using the functional syntax of `TypedDict`s, failing to pass a value to the *fields* parameter (`TD = TypedDict("TD")`) or passing `None` (`TD = TypedDict("TD", None)`) has been deprecated since Python 3.13. Use `class TD(TypedDict): pass` or `TD = TypedDict("TD", {})` to create a TypedDict with zero field.
  - The `typing.no_type_check_decorator()` decorator function has been deprecated since Python 3.13. After eight years in the `typing` module, it has yet to be supported by any major type checker.
- `wave`:
  - The `getmark()`, **`setmark()`**, and `getmarkers()` methods of the `Wave_read` and `Wave_write` classes have been deprecated since Python 3.13.
- `zipimport`:
  - `load_module()` has been deprecated since Python 3.10. Use `exec_module()` instead. (Contributed by Jiahao Li in gh-125746 [https://github.com/python/cpython/issues/125746].)

Pending removal in Python 3.16

- The import system:

  - Setting `__loader__` on a module while failing to set `__spec__.loader` is deprecated. In Python 3.16, **`__loader__`** will cease to be set or taken into consideration by the import system or the standard library.

- `array`:

  - The `'u'` format code (**`wchar_t`**) has been deprecated in documentation since Python 3.3 and at runtime since Python 3.13. Use the `'w'` format code (`Py_UCS4`) for Unicode characters instead.

- `asyncio`:
    - **`asyncio.iscoroutinefunction()`** is deprecated and will be removed in Python 3.16; use `inspect.iscoroutinefunction()` instead. (Contributed by Jiahao Li and Kumar Aditya in gh-122875 [https://github.com/python/cpython/issues/122875].)
    - `asyncio` policy system is deprecated and will be removed in Python 3.16. In particular, the following classes and functions are deprecated:
        - `asyncio.AbstractEventLoopPolicy`
        - `asyncio.DefaultEventLoopPolicy`
        - `asyncio.WindowsSelectorEventLoopPolicy`
        - `asyncio.WindowsProactorEventLoopPolicy`
        - `asyncio.get_event_loop_policy()`
        - `asyncio.set_event_loop_policy()`

      Users should use `asyncio.run()` or `asyncio.Runner` with *loop_factory* to use the desired event loop implementation.

      For example, to use `asyncio.SelectorEventLoop` on Windows:

      ```
      import asyncio

      async def main():
          ...

      asyncio.run(main(),
      loop_factory=asyncio.SelectorEventLoop)
      ```

(Contributed by Kumar Aditya in gh-127949 [https://github.com/python/cpython/issues/127949].)

- **`builtins`**:
  - Bitwise inversion on boolean types, `~True` or `~False` has been deprecated since Python 3.12, as it produces surprising and unintuitive results (`-2` and `-1`). Use `not x` instead for the logical negation of a Boolean. In the rare case that you need the bitwise inversion of the underlying integer, convert to `int` explicitly (`~int(x)`).
- **`functools`**:
  - Calling the Python implementation of **`functools.reduce()`** with *function* or *sequence* as keyword arguments has been deprecated since Python 3.14.
- **`logging`**:

  Support for custom logging handlers with the *strm* argument is deprecated and scheduled for removal in Python 3.16. Define handlers with the *stream* argument instead. (Contributed by Mariusz Felisiak in gh-115032 [https://github.com/python/cpython/issues/115032].)
- **`mimetypes`**:
  - Valid extensions start with a '.' or are empty for **`mimetypes.MimeTypes.add_type()`**. Undotted extensions are deprecated and will raise a **`ValueError`** in Python 3.16. (Contributed by Hugo van Kemenade in gh-75223 [https://github.com/python/cpython/issues/75223].)
- **`shutil`**:

  - The **`ExecError`** exception has been deprecated since Python 3.14. It has not been used by any function in **`shutil`** since Python 3.4, and is now an alias of [`RuntimeError`].

- [`symtable`]:

  - The [`Class.get_methods`] method has been deprecated since Python 3.14.

- [`sys`]:

  - The [`_enablelegacywindowsfsencoding()`] function has been deprecated since Python 3.13. Use the [`PYTHONLEGACYWINDOWSFSENCODING`] environment variable instead.

- [`sysconfig`]:

  - The **`sysconfig.expand_makefile_vars()`** function has been deprecated since Python 3.14. Use the `vars` argument of [`sysconfig.get_paths()`] instead.

- [`tarfile`]:

  - The undocumented and unused **`TarFile.tarfile`** attribute has been deprecated since Python 3.13.

Pending removal in Python 3.17

- [`collections.abc`]:

  - [`collections.abc.ByteString`] is scheduled for removal in Python 3.17.

    Use `isinstance(obj, collections.abc.Buffer)` to test if `obj` implements the [buffer protocol] at runtime. For use in type annotations, either use **`Buffer`** or a union that explicitly specifies the types your code supports (e.g., `bytes | bytearray | memoryview`).

`ByteString` was originally intended to be an abstract class that would serve as a supertype of both **`bytes`** and **`bytearray`**. However, since the ABC never had any methods, knowing that an object was an instance of **`ByteString`** never actually told you anything useful about the object. Other common buffer types such as **`memoryview`** were also never understood as subtypes of **`ByteString`** (either at runtime or by static type checkers).

See **PEP 688** [https://peps.python.org/pep-0688/#current-options] for more details. (Contributed by Shantanu Jain in gh-91896 [https://github.com/python/cpython/issues/91896].)

- **`typing`**:
  - Before Python 3.14, old-style unions were implemented using the private class `typing._UnionGenericAlias`. This class is no longer needed for the implementation, but it has been retained for backward compatibility, with removal scheduled for Python 3.17. Users should use documented introspection helpers like **`typing.get_origin()`** and **`typing.get_args()`** instead of relying on private implementation details.
  - **`typing.ByteString`**, deprecated since Python 3.9, is scheduled for removal in Python 3.17.

    Use `isinstance(obj, collections.abc.Buffer)` to test if `obj` implements the buffer protocol at runtime. For use in type annotations, either use **`Buffer`** or a union that explicitly specifies the types your code supports (e.g., `bytes | bytearray | memoryview`).

`ByteString` was originally intended to be an abstract class that would serve as a supertype of both **`bytes`** and **`bytearray`**. However, since the ABC never had any methods, knowing that an object was an instance of **`ByteString`** never actually told you anything useful about the object. Other common buffer types such as **`memoryview`** were also never understood as subtypes of **`ByteString`** (either at runtime or by static type checkers).

See **PEP 688** [https://peps.python.org/pep-0688/#current-options] for more details. (Contributed by Shantanu Jain in gh-91896 [https://github.com/python/cpython/issues/91896].)

Pending removal in Python 3.18

- **`decimal`**:
  - The non-standard and undocumented **`Decimal`** format specifier `'N'`, which is only supported in the **`decimal`** module’s C implementation, has been deprecated since Python 3.13. (Contributed by Serhiy Storchaka in gh-89902 [https://github.com/python/cpython/issues/89902].)

Pending removal in Python 3.19

- **`ctypes`**:
  - Implicitly switching to the MSVC-compatible struct layout by setting **`_pack_`** but not **`_layout_`** on non-Windows platforms.

Pending removal in future versions

The following APIs will be removed in the future, although there is currently no date scheduled for their removal.

- `argparse`:
    - Nesting argument groups and nesting mutually exclusive groups are deprecated.
    - Passing the undocumented keyword argument *prefix_chars* to `add_argument_group()` is now deprecated.
    - The `argparse.FileType` type converter is deprecated.
- `builtins`:
    - Generators: `throw(type, exc, tb)` and `athrow(type, exc, tb)` signature is deprecated: use `throw(exc)` and `athrow(exc)` instead, the single argument signature.
    - Currently Python accepts numeric literals immediately followed by keywords, for example `0in x`, `1or x`, `0if 1else 2`. It allows confusing and ambiguous expressions like `[0x1for x in y]` (which can be interpreted as `[0x1 for x in y]` or `[0x1f or x in y]`). A syntax warning is raised if the numeric literal is immediately followed by one of keywords **`and`**, **`else`**, **`for`**, **`if`**, **`in`**, **`is`** and **`or`**. In a future release it will be changed to a syntax error. (gh-87999 [https://github.com/python/cpython/issues/87999])
    - Support for `__index__()` and `__int__()` method returning non-int type: these methods will be required to return an instance of a strict subclass of `int`.
    - Support for `__float__()` method returning a strict subclass of `float`: these methods will be required to return an instance of `float`.
    - Support for `__complex__()` method returning a strict subclass of `complex`: these methods will be required to return an instance of `complex`.
    - Delegation of `int()` to `__trunc__()` method.

  - Passing a complex number as the *real* or *imag* argument in the **`complex()`** constructor is now deprecated; it should only be passed as a single positional argument. (Contributed by Serhiy Storchaka in gh-109218 [https://github.com/python/cpython/issues/109218].)
- **`calendar`**: `calendar.January` and `calendar.February` constants are deprecated and replaced by **`calendar.JANUARY`** and **`calendar.FEBRUARY`**. (Contributed by Prince Roshan in gh-103636 [https://github.com/python/cpython/issues/103636].)
- **`codecs`**: use **`open()`** instead of **`codecs.open()`**. (gh-133038 [https://github.com/python/cpython/issues/133038])
- **`codeobject.co_lnotab`**: use the **`codeobject.co_lines()`** method instead.
- **`datetime`**:
  - **`utcnow()`**: use `datetime.datetime.now(tz=datetime.UTC)`.
  - **`utcfromtimestamp()`**: use `datetime.datetime.fromtimestamp(timestamp, tz=datetime.UTC)`.
- **`gettext`**: Plural value must be an integer.
- **`importlib`**:
  - **`cache_from_source()`** *debug_override* parameter is deprecated: use the *optimization* parameter instead.
- **`importlib.metadata`**:
  - `EntryPoints` tuple interface.
  - Implicit `None` on return values.
- **`logging`**: the `warn()` method has been deprecated since Python 3.3, use **`warning()`** instead.
- **`mailbox`**: Use of StringIO input and text mode is deprecated, use BytesIO and binary mode instead.
- **`os`**: Calling **`os.register_at_fork()`** in multi-threaded process.
- **`pydoc.ErrorDuringImport`**: A tuple value for *exc_info* parameter is deprecated, use an exception instance.

- **re**: More strict rules are now applied for numerical group references and group names in regular expressions. Only sequence of ASCII digits is now accepted as a numerical reference. The group name in bytes patterns and replacement strings can now only contain ASCII letters and digits and underscore. (Contributed by Serhiy Storchaka in gh-91760 [https://github.com/python/cpython/issues/91760].)
- **`sre_compile`**, **`sre_constants`** and **`sre_parse`** modules.
- **`shutil`**: **`rmtree()`**'s *onerror* parameter is deprecated in Python 3.12; use the *onexc* parameter instead.
- **`ssl`** options and protocols:
  - **`ssl.SSLContext`** without protocol argument is deprecated.
  - **`ssl.SSLContext`**: **`set_npn_protocols()`** and **`selected_npn_protocol()`** are deprecated: use ALPN instead.
  - `ssl.OP_NO_SSL*` options
  - `ssl.OP_NO_TLS*` options
  - `ssl.PROTOCOL_SSLv3`
  - `ssl.PROTOCOL_TLS`
  - `ssl.PROTOCOL_TLSv1`
  - `ssl.PROTOCOL_TLSv1_1`
  - `ssl.PROTOCOL_TLSv1_2`
  - `ssl.TLSVersion.SSLv3`
  - `ssl.TLSVersion.TLSv1`
  - `ssl.TLSVersion.TLSv1_1`
- **`threading`** methods:
  - **`threading.Condition.notifyAll()`**: use **`notify_all()`**.
  - **`threading.Event.isSet()`**: use **`is_set()`**.
  - **`threading.Thread.isDaemon()`**, **`threading.Thread.setDaemon()`**: use **`threading.Thread.daemon`** attribute.
  - **`threading.Thread.getName()`**, **`threading.Thread.setName()`**: use **`threading.Thread.name`** attribute.

  - **`threading.currentThread()`**: use `threading.current_thread()`.
  - **`threading.activeCount()`**: use `threading.active_count()`.
- `typing.Text` (gh-92332 [https://github.com/python/cpython/issues/92332]).
- The internal class `typing._UnionGenericAlias` is no longer used to implement `typing.Union`. To preserve compatibility with users using this private class, a compatibility shim will be provided until at least Python 3.17. (Contributed by Jelle Zijlstra in gh-105499 [https://github.com/python/cpython/issues/105499].)
- `unittest.IsolatedAsyncioTestCase`: it is deprecated to return a value that is not `None` from a test case.
- `urllib.parse` deprecated functions: `urlparse()` instead
  - `splitattr()`
  - `splithost()`
  - `splitnport()`
  - `splitpasswd()`
  - `splitport()`
  - `splitquery()`
  - `splittag()`
  - `splittype()`
  - `splituser()`
  - `splitvalue()`
  - `to_bytes()`
- `wsgiref`: `SimpleHandler.stdout.write()` should not do partial writes.
- `xml.etree.ElementTree`: Testing the truth value of an `Element` is deprecated. In a future release it will always return `True`. Prefer explicit `len(elem)` or `elem is not None` tests instead.
- `sys._clear_type_cache()` is deprecated: use `sys._clear_internal_caches()` instead.

C API deprecations

Pending removal in Python 3.15

- The `PyImport_ImportModuleNoBlock()`: Use `PyImport_ImportModule()` instead.
- `PyWeakref_GetObject()` and `PyWeakref_GET_OBJECT()`: Use `PyWeakref_GetRef()` instead. The pythoncapi-compat project [https://github.com/python/pythoncapi-compat/] can be used to get `PyWeakref_GetRef()` on Python 3.12 and older.
- `Py_UNICODE` type and the **`Py_UNICODE_WIDE`** macro: Use **`wchar_t`** instead.
- **`PyUnicode_AsDecodedObject()`**: Use `PyCodec_Decode()` instead.
- **`PyUnicode_AsDecodedUnicode()`**: Use `PyCodec_Decode()` instead; Note that some codecs (for example, “base64”) may return a type other than `str`, such as `bytes`.
- **`PyUnicode_AsEncodedObject()`**: Use `PyCodec_Encode()` instead.
- **`PyUnicode_AsEncodedUnicode()`**: Use `PyCodec_Encode()` instead; Note that some codecs (for example, “base64”) may return a type other than `bytes`, such as `str`.
- Python initialization functions, deprecated in Python 3.13:
  - `Py_GetPath()`: Use `PyConfig_Get("module_search_paths")` (`sys.path`) instead.
  - `Py_GetPrefix()`: Use `PyConfig_Get("base_prefix")` (`sys.base_prefix`) instead. Use `PyConfig_Get("prefix")` (`sys.prefix`) if virtual environments need to be handled.
  - `Py_GetExecPrefix()`: Use `PyConfig_Get("base_exec_prefix")` (`sys.base_exec_prefix`) instead. Use `PyConfig_Get("exec_prefix")`

    (`sys.exec_prefix`) if virtual environments need to be handled.
  - `Py_GetProgramFullPath()`: Use `PyConfig_Get("executable")` (`sys.executable`) instead.
  - `Py_GetProgramName()`: Use `PyConfig_Get("executable")` (`sys.executable`) instead.
  - `Py_GetPythonHome()`: Use `PyConfig_Get("home")` or the `PYTHONHOME` environment variable instead.

  The pythoncapi-compat project [https://github.com/python/pythoncapi-compat/] can be used to get `PyConfig_Get()` on Python 3.13 and older.

- Functions to configure Python's initialization, deprecated in Python 3.11:

  - `PySys_SetArgvEx()`: Set `PyConfig.argv` instead.
  - `PySys_SetArgv()`: Set `PyConfig.argv` instead.
  - `Py_SetProgramName()`: Set `PyConfig.program_name` instead.
  - `Py_SetPythonHome()`: Set `PyConfig.home` instead.
  - `PySys_ResetWarnOptions()`: Clear `sys.warnoptions` and `warnings.filters` instead.

  The `Py_InitializeFromConfig()` API should be used with `PyConfig` instead.

- Global configuration variables:

  - `Py_DebugFlag`: Use `PyConfig.parser_debug` or `PyConfig_Get("parser_debug")` instead.
  - `Py_VerboseFlag`: Use `PyConfig.verbose` or `PyConfig_Get("verbose")` instead.
  - `Py_QuietFlag`: Use `PyConfig.quiet` or `PyConfig_Get("quiet")` instead.
  - `Py_InteractiveFlag`: Use `PyConfig.interactive` or `PyConfig_Get("interactive")` instead.

  - `Py_InspectFlag`: Use `PyConfig.inspect` or `PyConfig_Get("inspect")` instead.
  - `Py_OptimizeFlag`: Use `PyConfig.optimization_level` or `PyConfig_Get("optimization_level")` instead.
  - `Py_NoSiteFlag`: Use `PyConfig.site_import` or `PyConfig_Get("site_import")` instead.
  - `Py_BytesWarningFlag`: Use `PyConfig.bytes_warning` or `PyConfig_Get("bytes_warning")` instead.
  - `Py_FrozenFlag`: Use `PyConfig.pathconfig_warnings` or `PyConfig_Get("pathconfig_warnings")` instead.
  - `Py_IgnoreEnvironmentFlag`: Use `PyConfig.use_environment` or `PyConfig_Get("use_environment")` instead.
  - `Py_DontWriteBytecodeFlag`: Use `PyConfig.write_bytecode` or `PyConfig_Get("write_bytecode")` instead.
  - `Py_NoUserSiteDirectory`: Use `PyConfig.user_site_directory` or `PyConfig_Get("user_site_directory")` instead.
  - `Py_UnbufferedStdioFlag`: Use `PyConfig.buffered_stdio` or `PyConfig_Get("buffered_stdio")` instead.
  - `Py_HashRandomizationFlag`: Use `PyConfig.use_hash_seed` and `PyConfig.hash_seed` or `PyConfig_Get("hash_seed")` instead.
  - `Py_IsolatedFlag`: Use `PyConfig.isolated` or `PyConfig_Get("isolated")` instead.
  - `Py_LegacyWindowsFSEncodingFlag`: Use `PyPreConfig.legacy_windows_fs_encoding` or `PyConfig_Get("legacy_windows_fs_encoding")` instead.
  - `Py_LegacyWindowsStdioFlag`: Use `PyConfig.legacy_windows_stdio` or `PyConfig_Get("legacy_windows_stdio")` instead.
  - `Py_FileSystemDefaultEncoding`, `Py_HasFileSystemDefaultEncoding`: Use `PyConfig.filesystem_encoding` or `PyConfig_Get("filesystem_encoding")` instead.
  - `Py_FileSystemDefaultEncodeErrors`: Use `PyConfig.filesystem_errors` or

    `PyConfig_Get("filesystem_errors")` instead.
  - **`Py_UTF8Mode`**: Use `PyPreConfig.utf8_mode` or `PyConfig_Get("utf8_mode")` instead. (see `Py_PreInitialize()`)

  The `Py_InitializeFromConfig()` API should be used with `PyConfig` to set these options. Or `PyConfig_Get()` can be used to get these options at runtime.

Pending removal in Python 3.18

- The following private functions are deprecated and planned for removal in Python 3.18:
  - **`_PyBytes_Join()`**: use `PyBytes_Join()`.
  - **`_PyDict_GetItemStringWithError()`**: use `PyDict_GetItemStringRef()`.
  - **`_PyDict_Pop()`**: use `PyDict_Pop()`.
  - **`_PyLong_Sign()`**: use `PyLong_GetSign()`.
  - **`_PyLong_FromDigits()`** and **`_PyLong_New()`**: use `PyLongWriter_Create()`.
  - **`_PyThreadState_UncheckedGet()`**: use `PyThreadState_GetUnchecked()`.
  - **`_PyUnicode_AsString()`**: use `PyUnicode_AsUTF8()`.
  - **`_PyUnicodeWriter_Init()`**: replace `_PyUnicodeWriter_Init(&writer)` with `writer = PyUnicodeWriter_Create(0)`.
  - **`_PyUnicodeWriter_Finish()`**: replace `_PyUnicodeWriter_Finish(&writer)` with `PyUnicodeWriter_Finish(writer)`.
  - **`_PyUnicodeWriter_Dealloc()`**: replace `_PyUnicodeWriter_Dealloc(&writer)` with `PyUnicodeWriter_Discard(writer)`.
  - **`_PyUnicodeWriter_WriteChar()`**: replace `_PyUnicodeWriter_WriteChar(&writer, ch)` with `PyUnicodeWriter_WriteChar(writer, ch)`.

  - **`_PyUnicodeWriter_WriteStr()`**: replace `_PyUnicodeWriter_WriteStr(&writer, str)` with `PyUnicodeWriter_WriteStr(writer, str)`.
  - **`_PyUnicodeWriter_WriteSubstring()`**: replace `_PyUnicodeWriter_WriteSubstring(&writer, str, start, end)` with `PyUnicodeWriter_WriteSubstring(writer, str, start, end)`.
  - **`_PyUnicodeWriter_WriteASCIIString()`**: replace `_PyUnicodeWriter_WriteASCIIString(&writer, str)` with `PyUnicodeWriter_WriteASCII(writer, str)`.
  - **`_PyUnicodeWriter_WriteLatin1String()`**: replace `_PyUnicodeWriter_WriteLatin1String(&writer, str)` with `PyUnicodeWriter_WriteUTF8(writer, str)`.
  - **`_PyUnicodeWriter_Prepare()`**: (no replacement).
  - **`_PyUnicodeWriter_PrepareKind()`**: (no replacement).
  - **`_Py_HashPointer()`**: use `Py_HashPointer()`.
  - **`_Py_fopen_obj()`**: use `Py_fopen()`.

  The pythoncapi-compat project [https://github.com/python/pythoncapi-compat/] can be used to get these new public functions on Python 3.13 and older. (Contributed by Victor Stinner in gh-128863 [https://github.com/python/cpython/issues/128863].)

Pending removal in future versions

The following APIs are deprecated and will be removed, although there is currently no date scheduled for their removal.

- `Py_TPFLAGS_HAVE_FINALIZE`: Unneeded since Python 3.8.
- `PyErr_Fetch()`: Use `PyErr_GetRaisedException()` instead.
- `PyErr_NormalizeException()`: Use `PyErr_GetRaisedException()` instead.
- `PyErr_Restore()`: Use `PyErr_SetRaisedException()` instead.

- `PyModule_GetFilename()`: Use `PyModule_GetFilenameObject()` instead.
- `PyOS_AfterFork()`: Use `PyOS_AfterFork_Child()` instead.
- `PySlice_GetIndicesEx()`: Use `PySlice_Unpack()` and `PySlice_AdjustIndices()` instead.
- `PyUnicode_READY()`: Unneeded since Python 3.12
- `PyErr_Display()`: Use `PyErr_DisplayException()` instead.
- `_PyErr_ChainExceptions()`: Use `_PyErr_ChainExceptions1()` instead.
- `PyBytesObject.ob_shash` member: call `PyObject_Hash()` instead.
- Thread Local Storage (TLS) API:
  - `PyThread_create_key()`: Use `PyThread_tss_alloc()` instead.
  - `PyThread_delete_key()`: Use `PyThread_tss_free()` instead.
  - `PyThread_set_key_value()`: Use `PyThread_tss_set()` instead.
  - `PyThread_get_key_value()`: Use `PyThread_tss_get()` instead.
  - `PyThread_delete_key_value()`: Use `PyThread_tss_delete()` instead.
  - `PyThread_ReInitTLS()`: Unneeded since Python 3.7.

Glossary

>>>

The default Python prompt of the interactive shell. Often seen for code examples which can be executed interactively in the interpreter.

...

Can refer to:

- The default Python prompt of the interactive shell when entering the code for an indented code block, when within a pair of matching left and right delimiters (parentheses, square brackets, curly braces or triple quotes), or after specifying a decorator.
- The three dots form of the Ellipsis object.

abstract base class

Abstract base classes complement duck-typing by providing a way to define interfaces when other techniques like `hasattr()` would be clumsy or subtly wrong (for example with magic methods). ABCs introduce virtual subclasses, which are classes that don’t inherit from a class but are still recognized by `isinstance()` and `issubclass()`; see the `abc` module documentation. Python comes with many built-in ABCs for data structures (in the `collections.abc` module), numbers (in the `numbers` module), streams (in the `io` module), import finders and loaders (in the `importlib.abc` module). You can create your own ABCs with the `abc` module.

annotate function

A function that can be called to retrieve the annotations of an object. This function is accessible as the `__annotate__` attribute of functions, classes, and modules. Annotate functions are a subset of evaluate functions.

annotation

A label associated with a variable, a class attribute or a function parameter or return value, used by convention as a type hint.

Annotations of local variables cannot be accessed at runtime, but annotations of global variables, class attributes, and functions can be retrieved by calling `annotationlib.get_annotations()` on modules, classes, and functions, respectively.

See variable annotation, function annotation, **PEP 484** [https://peps.python.org/pep-0484/], **PEP 526** [https://peps.python.org/pep-0526/], and **PEP 649** [https://peps.python.org/pep-0649/], which describe this functionality. Also see Annotations Best Practices for best practices on working with annotations.

argument

A value passed to a function (or method) when calling the function. There are two kinds of argument:

- *keyword argument*: an argument preceded by an identifier (e.g. `name=`) in a function call or passed as a value in a dictionary preceded by `**`. For example, `3` and `5` are both keyword arguments in the following calls to `complex()`:

  ```
  complex(real=3, imag=5)
  complex(**{'real': 3, 'imag': 5})
  ```

- *positional argument*: an argument that is not a keyword argument. Positional arguments can appear at the beginning of an argument list and/or be passed as elements of an iterable preceded by `*`. For example, `3` and `5` are both positional arguments in the following calls:

  ```
  complex(3, 5)
  complex(*(3, 5))
  ```

Arguments are assigned to the named local variables in a function body. See the Calls section for the rules governing this assignment. Syntactically, any expression can be used to represent an argument; the evaluated value is assigned to the local variable.

See also the parameter glossary entry, the FAQ question on the difference between arguments and parameters, and **PEP 362** [https://peps.python.org/pep-0362/].

asynchronous context manager

An object which controls the environment seen in an **`async with`** statement by defining **`__aenter__()`** and **`__aexit__()`** methods. Introduced by **PEP 492** [https://peps.python.org/pep-0492/].

asynchronous generator

A function which returns an asynchronous generator iterator. It looks like a coroutine function defined with **`async def`** except that it contains **`yield`** expressions for producing a series of values usable in an **`async for`** loop.

Usually refers to an asynchronous generator function, but may refer to an *asynchronous generator iterator* in some contexts. In cases where the intended meaning isn’t clear, using the full terms avoids ambiguity.

An asynchronous generator function may contain `await` expressions as well as `async for`, and `async with` statements.

asynchronous generator iterator

An object created by an asynchronous generator function.

This is an asynchronous iterator which when called using the `__anext__()` method returns an awaitable object which will execute the body of the asynchronous generator function until the next `yield` expression.

Each `yield` temporarily suspends processing, remembering the execution state (including local variables and pending try-statements). When the *asynchronous generator iterator* effectively resumes with another awaitable returned by `__anext__()`, it picks up where it left off. See **PEP 492** [https://peps.python.org/pep-0492/] and **PEP 525** [https://peps.python.org/pep-0525/].

asynchronous iterable

An object, that can be used in an `async for` statement. Must return an asynchronous iterator from its `__aiter__()` method. Introduced by **PEP 492** [https://peps.python.org/pep-0492/].

asynchronous iterator

An object that implements the `__aiter__()` and `__anext__()` methods. `__anext__()` must return an awaitable object. `async for` resolves the awaitables returned by an asynchronous iterator's `__anext__()` method until it raises a `StopAsyncIteration` exception. Introduced by **PEP 492** [https://peps.python.org/pep-0492/].

atomic operation

An operation that appears to execute as a single, indivisible step: no other thread can observe it half-done, and its effects become visible all at once. Python does not guarantee that high-level statements are atomic (for example, `x += 1` performs multiple bytecode operations and is not atomic). Atomicity is only guaranteed where explicitly documented. See also race condition and data race.

attached thread state

A thread state that is active for the current OS thread.

When a thread state is attached, the OS thread has access to the full Python C API and can safely invoke the bytecode interpreter.

Unless a function explicitly notes otherwise, attempting to call the C API without an attached thread state will result in a fatal error or undefined behavior. A thread state can be attached and detached explicitly by the user through the C API, or implicitly by the runtime, including during blocking C calls and by the bytecode interpreter in between calls.

On most builds of Python, having an attached thread state implies that the caller holds the GIL for the current interpreter, so only one OS thread can have an attached thread state at a given moment. In free-threaded builds of Python, threads can concurrently hold an attached thread state, allowing for true parallelism of the bytecode interpreter.

attribute

A value associated with an object which is usually referenced by name using dotted expressions. For example, if an object *o* has an attribute *a* it would be referenced as *o.a*.

It is possible to give an object an attribute whose name is not an identifier as defined by Names (identifiers and keywords), for example using `setattr()`, if the object allows it. Such an attribute will not be accessible using a dotted expression, and would instead need to be retrieved with `getattr()`.

awaitable

An object that can be used in an `await` expression. Can be a coroutine or an object with an `__await__()` method. See also **PEP 492** [https://peps.python.org/pep-0492/].

BDFL

Benevolent Dictator For Life, a.k.a. Guido van Rossum [https://gvanrossum.github.io/], Python’s creator.

binary file

A file object able to read and write bytes-like objects. Examples of binary files are files opened in binary mode (`'rb'`, `'wb'` or `'rb+'`), `sys.stdin.buffer`, `sys.stdout.buffer`, and instances of `io.BytesIO` and `gzip.GzipFile`.

See also text file for a file object able to read and write `str` objects.

borrowed reference

In Python’s C API, a borrowed reference is a reference to an object, where the code using the object does not own the reference. It becomes a dangling pointer if the object is destroyed. For example, a garbage collection can remove the last strong reference to the object and so destroy it.

Calling `Py_INCREF()` on the borrowed reference is recommended to convert it to a strong reference in-place, except when the object cannot be destroyed before the last usage of the borrowed reference. The `Py_NewRef()` function can be used to create a new strong reference.

bytes-like object

An object that supports the Buffer Protocol and can export a C-contiguous buffer. This includes all `bytes`, `bytearray`, and `array.array` objects, as well as many common `memoryview` objects. Bytes-like objects can be used for various operations that work with binary data; these include compression, saving to a binary file, and sending over a socket.

Some operations need the binary data to be mutable. The documentation often refers to these as “read-write bytes-like objects”. Example mutable buffer objects include `bytearray` and a `memoryview` of a `bytearray`. Other operations require the binary data to be stored in immutable objects (“read-only bytes-like objects”); examples of these include `bytes` and a `memoryview` of a `bytes` object.

bytecode

Python source code is compiled into bytecode, the internal representation of a Python program in the CPython interpreter. The bytecode is also cached in `.pyc` files so that executing the same file is faster the second time (recompilation from source to bytecode can be avoided). This “intermediate language” is said to run on a virtual machine that executes the machine code corresponding to each bytecode. Do note that bytecodes are not expected to work between different Python virtual machines, nor to be stable between Python releases.

A list of bytecode instructions can be found in the documentation for the dis module.

callable

A callable is an object that can be called, possibly with a set of arguments (see argument), with the following syntax:

```
callable(argument1, argument2, argumentN)
```

A function, and by extension a method, is a callable. An instance of a class that implements the `__call__()` method is also a callable.

callback

A subroutine function which is passed as an argument to be executed at some point in the future.

class

A template for creating user-defined objects. Class definitions normally contain method definitions which operate on instances of the class.

class variable

A variable defined in a class and intended to be modified only at class level (i.e., not in an instance of the class).

closure variable

A free variable referenced from a nested scope that is defined in an outer scope rather than being resolved at runtime from the globals or builtin namespaces. May be explicitly defined with the `nonlocal` keyword to allow write access, or implicitly defined if the variable is only being read.

For example, in the `inner` function in the following code, both `x` and `print` are free variables, but only `x` is a *closure variable*:

```
def outer():
    x = 0
    def inner():
        nonlocal x
        x += 1
        print(x)
    return inner
```

Due to the **`codeobject.co_freevars`** attribute (which, despite its name, only includes the names of closure variables rather than listing all referenced free variables), the more general free variable term is sometimes used even when the intended meaning is to refer specifically to closure variables.

complex number

An extension of the familiar real number system in which all numbers are expressed as a sum of a real part and an imaginary part. Imaginary numbers are real multiples of the imaginary unit (the square root of `-1`), often written `i` in mathematics or `j` in engineering. Python has built-in support for complex numbers, which are written with this latter notation; the imaginary part is written with a `j` suffix, e.g., `3+1j`. To get access to complex equivalents of the **`math`** module, use **`cmath`**. Use of complex numbers is a fairly advanced mathematical feature. If you’re not aware of a need for them, it’s almost certain you can safely ignore them.

concurrency

The ability of a computer program to perform multiple tasks at the same time. Python provides libraries for writing programs that make use of different forms of concurrency. `asyncio` is a library for dealing with asynchronous tasks and coroutines. `threading` provides access to operating system threads and `multiprocessing` to operating system processes. Multi-core processors can execute threads and processes on different CPU cores at the same time (see parallelism).

concurrent modification

When multiple threads modify shared data at the same time. Concurrent modification without proper synchronization can cause race conditions, and might also trigger a data race, data corruption, or both.

context

This term has different meanings depending on where and how it is used. Some common meanings:

- The temporary state or environment established by a context manager via a `with` statement.
- The collection of keyvalue bindings associated with a particular `contextvars.Context` object and accessed via `ContextVar` objects. Also see context variable.
- A `contextvars.Context` object. Also see current context.

context management protocol

The `__enter__()` and `__exit__()` methods called by the `with` statement. See **PEP 343** [https://peps.python.org/pep-0343/].

context manager

An object which implements the context management protocol and controls the environment seen in a `with` statement. See **PEP 343** [https://peps.python.org/pep-0343/].

context variable

A variable whose value depends on which context is the current context. Values are accessed via `contextvars.ContextVar` objects. Context variables are primarily used to isolate state between concurrent asynchronous tasks.

contiguous

A buffer is considered contiguous exactly if it is either *C-contiguous* or *Fortran contiguous*. Zero-dimensional buffers are C and Fortran contiguous. In one-dimensional arrays, the items must be laid out in memory next to each other, in order of increasing indexes starting from zero. In multidimensional C-contiguous arrays, the last index varies the fastest when visiting items in order of memory address. However, in Fortran contiguous arrays, the first index varies the fastest.

coroutine

Coroutines are a more generalized form of subroutines. Subroutines are entered at one point and exited at another point. Coroutines can be entered, exited, and resumed at many different points. They can be implemented with the `async def` statement. See also **PEP 492** [https://peps.python.org/pep-0492/].

coroutine function

A function which returns a coroutine object. A coroutine function may be defined with the `async def` statement, and may contain `await`, `async for`, and `async with` keywords. These were introduced by **PEP 492** [https://peps.python.org/pep-0492/].

CPython

The canonical implementation of the Python programming language, as distributed on python.org [https://www.python.org]. The term “CPython” is used when necessary to distinguish this implementation from others such as Jython or IronPython.

current context

The context (`contextvars.Context` object) that is currently used by `ContextVar` objects to access (get or set) the values of context variables. Each thread has its own current context. Frameworks for executing asynchronous tasks (see `asyncio`) associate each task with a context which becomes the current context whenever the task starts or resumes execution.

cyclic isolate

A subgroup of one or more objects that reference each other in a reference cycle, but are not referenced by objects outside the group. The goal of the cyclic garbage collector is to identify these groups and break the reference cycles so that the memory can be reclaimed.

data race

A situation where multiple threads access the same memory location concurrently, at least one of the accesses is a write, and the threads do not use any synchronization to control their access. Data races lead to non-deterministic behavior and can cause data corruption. Proper use of locks and other synchronization primitives prevents data races. Note that data races can only happen in native code, but that native code might be exposed in a Python API. See also race condition and thread-safe.

deadlock

A situation in which two or more tasks (threads, processes, or coroutines) wait indefinitely for each other to release resources or complete actions, preventing any from making progress. For example, if thread A holds lock 1 and waits for lock 2, while thread B holds lock 2 and waits for lock 1, both threads will wait indefinitely. In Python this often arises from acquiring multiple locks in conflicting orders or from circular join/await dependencies. Deadlocks can be avoided by always acquiring multiple locks in a consistent order. See also lock and reentrant.

decorator

A function returning another function, usually applied as a function transformation using the `@wrapper` syntax. Common examples for decorators are **`classmethod()`** and **`staticmethod()`**.

The decorator syntax is merely syntactic sugar, the following two function definitions are semantically equivalent:

```
def f(arg):
    ...
f = staticmethod(f)

@staticmethod
def f(arg):
    ...
```

The same concept exists for classes, but is less commonly used there. See the documentation for function definitions and class definitions for more about decorators.

descriptor

Any object which defines the methods `__get__()`, `__set__()`, or `__delete__()`. When a class attribute is a descriptor, its special binding behavior is triggered upon attribute lookup. Normally, using *a.b* to get, set or delete an attribute looks up the object named *b* in the class dictionary for *a*, but if *b* is a descriptor, the respective descriptor method gets called. Understanding descriptors is a key to a deep understanding of Python because they are the basis for many features including functions, methods, properties, class methods, static methods, and reference to super classes.

For more information about descriptors' methods, see Implementing Descriptors or the Descriptor How To Guide.

dictionary

An associative array, where arbitrary keys are mapped to values. The keys can be any object with `__hash__()` and `__eq__()` methods. Called a hash in Perl.

dictionary comprehension

A compact way to process all or part of the elements in an iterable and return a dictionary with the results. `results = {n: n ** 2 for n in range(10)}` generates a dictionary containing key `n` mapped to value `n ** 2`. See Displays for lists, sets and dictionaries.

dictionary view

The objects returned from `dict.keys()`, `dict.values()`, and `dict.items()` are called dictionary views. They provide a dynamic view on the dictionary's entries, which means that when the dictionary changes, the view reflects these changes. To force the dictionary view to become a full list use `list(dictview)`. See Dictionary view objects.

docstring

A string literal which appears as the first expression in a class, function or module. While ignored when the suite is executed, it is recognized by the compiler and put into the `__doc__` attribute of the enclosing class, function or module. Since it is available via introspection, it is the canonical place for documentation of the object.

duck-typing

A programming style which does not look at an object's type to determine if it has the right interface; instead, the method or attribute is simply called or used ("If it looks like a duck and quacks like a duck, it must be a duck.") By emphasizing interfaces rather than specific types, well-designed code improves its flexibility by allowing polymorphic substitution. Duck-typing avoids tests using `type()` or `isinstance()`. (Note, however, that duck-typing can be complemented with abstract base classes.) Instead, it typically employs `hasattr()` tests or EAFP programming.

dunder

An informal short-hand for "double underscore", used when talking about a special method. For example, `__init__` is often pronounced "dunder init".

EAFP

Easier to ask for forgiveness than permission. This common Python coding style assumes the existence of valid keys or attributes and catches exceptions if the assumption proves false. This clean and fast style is characterized by the presence of many `try` and `except` statements. The technique contrasts with the LBYL style common to many other languages such as C.

evaluate function

A function that can be called to evaluate a lazily evaluated attribute of an object, such as the value of type aliases created with the `type` statement.

expression

A piece of syntax which can be evaluated to some value. In other words, an expression is an accumulation of expression elements like literals, names, attribute access, operators or function calls which all return a value. In contrast to many other languages, not all language constructs are expressions. There are also statements which cannot be used as expressions, such as `while`. Assignments are also statements, not expressions.

extension module

A module written in C or C++, using Python's C API to interact with the core and with user code.

f-string
f-strings

String literals prefixed with `f` or `F` are commonly called "f-strings" which is short for formatted string literals. See also **PEP 498** [https://peps.python.org/pep-0498/].

file object

An object exposing a file-oriented API (with methods such as `read()` or `write()`) to an underlying resource. Depending on the way it was created, a file object can mediate access to a real on-disk file or to another type of storage or communication device (for example standard input/output, in-memory buffers, sockets, pipes, etc.). File objects are also called *file-like objects* or *streams*.

There are actually three categories of file objects: raw binary files, buffered binary files and text files. Their interfaces are defined in the `io` module. The canonical way to create a file object is by using the `open()` function.

file-like object

A synonym for file object.

filesystem encoding and error handler

Encoding and error handler used by Python to decode bytes from the operating system and encode Unicode to the operating system.

The filesystem encoding must guarantee to successfully decode all bytes below 128. If the file system encoding fails to provide this guarantee, API functions can raise `UnicodeError`.

The `sys.getfilesystemencoding()` and `sys.getfilesystemencodeerrors()` functions can be used to get the filesystem encoding and error handler.

The filesystem encoding and error handler are configured at Python startup by the `PyConfig_Read()` function: see `filesystem_encoding` and `filesystem_errors` members of `PyConfig`.

See also the locale encoding.

finder

An object that tries to find the loader for a module that is being imported.

There are two types of finder: meta path finders for use with `sys.meta_path`, and path entry finders for use with `sys.path_hooks`.

See Finders and loaders and **`importlib`** for much more detail.

floor division

Mathematical division that rounds down to nearest integer. The floor division operator is `//`. For example, the expression `11 // 4` evaluates to `2` in contrast to the `2.75` returned by float true division. Note that `(-11) // 4` is `-3` because that is `-2.75` rounded *downward*. See **PEP 238** [https://peps.python.org/pep-0238/].

free threading

A threading model where multiple threads can run Python bytecode simultaneously within the same interpreter. This is in contrast to the global interpreter lock which allows only one thread to execute Python bytecode at a time. See **PEP 703** [https://peps.python.org/pep-0703/].

free variable

Formally, as defined in the language execution model, a free variable is any variable used in a namespace which is not a local variable in that namespace. See closure variable for an example. Pragmatically, due to the name of the **`codeobject.co_freevars`** attribute, the term is also sometimes used as a synonym for closure variable.

function

A series of statements which returns some value to a caller. It can also be passed zero or more arguments which may be used in the execution of the body. See also parameter, method, and the Function definitions section.

function annotation

An annotation of a function parameter or return value.

Function annotations are usually used for type hints: for example, this function is expected to take two **`int`** arguments and is also expected to have an **`int`** return value:

```
def sum_two_numbers(a: int, b: int) -> int:
   return a + b
```

Function annotation syntax is explained in section Function definitions.

See variable annotation and **PEP 484** [https://peps.python.org/pep-0484/], which describe this functionality. Also see Annotations Best Practices for best practices on working with annotations.

__future__

A future statement, `from __future__ import <feature>`, directs the compiler to compile the current module using syntax or semantics that will become standard in a future release of Python. The **`__future__`** module documents the possible values of *feature*. By importing this module and evaluating its variables, you can see when a new feature was first added to the language and when it will (or did) become the default:

```
>>> import __future__
>>> __future__.division
_Feature((2, 2, 0, 'alpha', 2), (3, 0, 0, 'alpha', 0), 8192)
```

garbage collection

The process of freeing memory when it is not used anymore. Python performs garbage collection via reference counting and a cyclic garbage collector that is able to detect and break reference cycles. The garbage collector can be controlled using the **`gc`** module.

generator

A function which returns a generator iterator. It looks like a normal function except that it contains **`yield`** expressions for producing a series of values usable in a for-loop or that can be retrieved one at a time with the **`next()`** function.

Usually refers to a generator function, but may refer to a *generator iterator* in some contexts. In cases where the intended meaning isn’t clear, using the full terms avoids ambiguity.

generator iterator

An object created by a generator function.

Each **`yield`** temporarily suspends processing, remembering the execution state (including local variables and pending try-statements). When the *generator iterator* resumes, it picks up where it left off (in contrast to functions which start fresh on every invocation).

generator expression

An expression that returns an iterator. It looks like a normal expression followed by a **`for`** clause defining a loop variable, range, and an optional **`if`** clause. The combined expression generates values for an enclosing function:

```
>>> sum(i*i for i in range(10))         # sum of squares 0,
1, 4, ... 81
285
```

generic function

A function composed of multiple functions implementing the same operation for different types. Which implementation should be used during a call is determined by the dispatch algorithm.

See also the single dispatch glossary entry, the `functools.singledispatch()` decorator, and **PEP 443** [https://peps.python.org/pep-0443/].

generic type

A type that can be parameterized; typically a container class such as `list` or `dict`. Used for type hints and annotations.

For more details, see generic alias types, **PEP 483** [https://peps.python.org/pep-0483/], **PEP 484** [https://peps.python.org/pep-0484/], **PEP 585** [https://peps.python.org/pep-0585/], and the `typing` module.

GIL

See global interpreter lock.

global interpreter lock

The mechanism used by the CPython interpreter to assure that only one thread executes Python bytecode at a time. This simplifies the CPython implementation by making the object model (including critical built-in types such as `dict`) implicitly safe against concurrent access. Locking the entire interpreter makes it easier for the interpreter to be multi-threaded, at the expense of much of the parallelism afforded by multi-processor machines.

However, some extension modules, either standard or third-party, are designed so as to release the GIL when doing computationally intensive tasks such as compression or hashing. Also, the GIL is always released when doing I/O.

As of Python 3.13, the GIL can be disabled using the `--disable-gil` build configuration. After building Python with this option, code must be run with `-X gil=0` or after setting the `PYTHON_GIL=0` environment variable. This feature enables improved performance for multi-threaded applications and makes it easier to use multi-core CPUs efficiently. For more details, see **PEP 703** [https://peps.python.org/pep-0703/].

In prior versions of Python's C API, a function might declare that it requires the GIL to be held in order to use it. This refers to having an attached thread state.

global state

Data that is accessible throughout a program, such as module-level variables, class variables, or C static variables in extension modules. In multi-threaded programs, global state shared between threads typically requires synchronization to avoid race conditions and data races.

hash-based pyc

A bytecode cache file that uses the hash rather than the last-modified time of the corresponding source file to determine its validity. See Cached bytecode invalidation.

hashable

An object is *hashable* if it has a hash value which never changes during its lifetime (it needs a `__hash__()` method), and can be compared to other objects (it needs an `__eq__()` method). Hashable objects which compare equal must have the same hash value.

Hashability makes an object usable as a dictionary key and a set member, because these data structures use the hash value internally.

Most of Python’s immutable built-in objects are hashable; mutable containers (such as lists or dictionaries) are not; immutable containers (such as tuples and frozensets) are only hashable if their elements are hashable. Objects which are instances of user-defined classes are hashable by default. They all compare unequal (except with themselves), and their hash value is derived from their `id()`.

IDLE

An Integrated Development and Learning Environment for Python. IDLE — Python editor and shell is a basic editor and interpreter environment which ships with the standard distribution of Python.

immortal

Immortal objects are a CPython implementation detail introduced in **PEP 683** [https://peps.python.org/pep-0683/].

If an object is immortal, its reference count is never modified, and therefore it is never deallocated while the interpreter is running. For example, `True` and `None` are immortal in CPython.

Immortal objects can be identified via `sys._is_immortal()`, or via `PyUnstable_IsImmortal()` in the C API.

immutable

An object with a fixed value. Immutable objects include numbers, strings and tuples. Such an object cannot be altered. A new object has to be created if a different value has to be stored. They play an important role in places where a constant hash value is needed, for example as a key in a dictionary. Immutable objects are inherently thread-safe because their state cannot be modified after creation, eliminating concerns about improperly synchronized concurrent modification.

import path

A list of locations (or path entries) that are searched by the path based finder for modules to import. During import, this list of locations usually comes from **`sys.path`**, but for subpackages it may also come from the parent package's `__path__` attribute.

importing

The process by which Python code in one module is made available to Python code in another module.

importer

An object that both finds and loads a module; both a finder and loader object.

interactive

Python has an interactive interpreter which means you can enter statements and expressions at the interpreter prompt, immediately execute them and see their results. Just launch `python` with no arguments (possibly by selecting it from your computer's main menu). It is a very powerful way to test out new ideas or inspect modules and packages (remember `help(x)`). For more on interactive mode, see Interactive Mode.

interpreted

Python is an interpreted language, as opposed to a compiled one, though the distinction can be blurry because of the presence of the bytecode compiler. This means that source files can be run directly without explicitly creating an executable which is then run. Interpreted languages typically have a shorter development/debug cycle than compiled ones, though their programs generally also run more slowly. See also interactive.

interpreter shutdown

When asked to shut down, the Python interpreter enters a special phase where it gradually releases all allocated resources, such as modules and various critical internal structures. It also makes several calls to the garbage collector. This can trigger the execution of code in user-defined destructors or weakref callbacks. Code executed during the shutdown phase can encounter various exceptions as the resources it relies on may not function anymore (common examples are library modules or the warnings machinery).

The main reason for interpreter shutdown is that the `__main__` module or the script being run has finished executing.

iterable

An object capable of returning its members one at a time. Examples of iterables include all sequence types (such as **`list`**, **`str`**, and **`tuple`**) and some non-sequence types like **`dict`**, file objects, and objects of any classes you define with an **`__iter__()`** method or with a **`__getitem__()`** method that implements sequence semantics.

Iterables can be used in a **`for`** loop and in many other places where a sequence is needed (**`zip()`**, **`map()`**, …). When an iterable object is passed as an argument to the built-in function **`iter()`**, it returns an iterator for the object. This iterator is good for one pass over the set of values. When using iterables, it is usually not necessary to call **`iter()`** or deal with iterator objects yourself. The **`for`** statement does that automatically for you, creating a temporary unnamed variable to hold the iterator for the duration of the loop. See also iterator, sequence, and generator.

iterator

An object representing a stream of data. Repeated calls to the iterator's `__next__()` method (or passing it to the built-in function `next()`) return successive items in the stream. When no more data are available a `StopIteration` exception is raised instead. At this point, the iterator object is exhausted and any further calls to its `__next__()` method just raise `StopIteration` again. Iterators are required to have an `__iter__()` method that returns the iterator object itself so every iterator is also iterable and may be used in most places where other iterables are accepted. One notable exception is code which attempts multiple iteration passes. A container object (such as a `list`) produces a fresh new iterator each time you pass it to the `iter()` function or use it in a `for` loop. Attempting this with an iterator will just return the same exhausted iterator object used in the previous iteration pass, making it appear like an empty container.

More information can be found in Iterator Types.

CPython implementation detail: CPython does not consistently apply the requirement that an iterator define `__iter__()`. And also please note that free-threaded CPython does not guarantee thread-safe behavior of iterator operations.

key function

A key function or collation function is a callable that returns a value used for sorting or ordering. For example, `locale.strxfrm()` is used to produce a sort key that is aware of locale specific sort conventions.

A number of tools in Python accept key functions to control how elements are ordered or grouped. They include **`min()`**, **`max()`**, **`sorted()`**, **`list.sort()`**, **`heapq.merge()`**, **`heapq.nsmallest()`**, **`heapq.nlargest()`**, and **`itertools.groupby()`**.

There are several ways to create a key function. For example. the **`str.casefold()`** method can serve as a key function for case insensitive sorts. Alternatively, a key function can be built from a **`lambda`** expression such as `lambda r: (r[0], r[2])`. Also, **`operator.attrgetter()`**, **`operator.itemgetter()`**, and **`operator.methodcaller()`** are three key function constructors. See the Sorting HOW TO for examples of how to create and use key functions.

keyword argument

See argument.

lambda

An anonymous inline function consisting of a single expression which is evaluated when the function is called. The syntax to create a lambda function is `lambda [parameters]: expression`

LBYL

Look before you leap. This coding style explicitly tests for pre-conditions before making calls or lookups. This style contrasts with the EAFP approach and is characterized by the presence of many **`if`** statements.

In a multi-threaded environment, the LBYL approach can risk introducing a race condition between “the looking” and “the leaping”. For example, the code, `if key in mapping: return mapping[key]` can fail if another thread removes *key* from *mapping* after the test, but before the lookup. This issue can be solved with locks or by using the EAFP approach. See also thread-safe.

lexical analyzer

Formal name for the *tokenizer*; see token.

list

A built-in Python sequence. Despite its name it is more akin to an array in other languages than to a linked list since access to elements is *O*(1).

list comprehension

A compact way to process all or part of the elements in a sequence and return a list with the results. `result = ['{:#04x}'.format(x) for x in range(256) if x % 2 == 0]` generates a list of strings containing even hex numbers (0x..) in the range from 0 to 255. The **`if`** clause is optional. If omitted, all elements in `range(256)` are processed.

lock

A synchronization primitive that allows only one thread at a time to access a shared resource. A thread must acquire a lock before accessing the protected resource and release it afterward. If a thread attempts to acquire a lock that is already held by another thread, it will block until the lock becomes available. Python’s **`threading`** module provides **`Lock`** (a basic lock) and **`RLock`** (a reentrant lock). Locks are used to prevent race conditions and ensure thread-safe access to shared data. Alternative design patterns to locks exist such as queues, producer/consumer patterns, and thread-local state. See also deadlock, and reentrant.

loader

An object that loads a module. It must define the **`exec_module()`** and **`create_module()`** methods to implement the `Loader` interface. A loader is typically returned by a finder. See also:

- Finders and loaders
- `importlib.abc.Loader`
- **PEP 302** [https://peps.python.org/pep-0302/]

locale encoding

On Unix, it is the encoding of the LC_CTYPE locale. It can be set with `locale.setlocale(locale.LC_CTYPE, new_locale)`.

On Windows, it is the ANSI code page (ex: `"cp1252"`).

On Android and VxWorks, Python uses `"utf-8"` as the locale encoding.

`locale.getencoding()` can be used to get the locale encoding.

See also the filesystem encoding and error handler.

magic method

An informal synonym for special method.

mapping

A container object that supports arbitrary key lookups and implements the methods specified in the `collections.abc.Mapping` or `collections.abc.MutableMapping` abstract base classes. Examples include `dict`, `collections.defaultdict`, `collections.OrderedDict` and `collections.Counter`.

meta path finder

A finder returned by a search of `sys.meta_path`. Meta path finders are related to, but different from path entry finders.

See `importlib.abc.MetaPathFinder` for the methods that meta path finders implement.

metaclass

The class of a class. Class definitions create a class name, a class dictionary, and a list of base classes. The metaclass is responsible for taking those three arguments and creating the class. Most object oriented programming languages provide a default implementation. What makes Python special is that it is possible to create custom metaclasses. Most users never need this tool, but when the need arises, metaclasses can provide powerful, elegant solutions. They have been used for logging attribute access, adding thread-safety, tracking object creation, implementing singletons, and many other tasks.

More information can be found in Metaclasses.

method

A function which is defined inside a class body. If called as an attribute of an instance of that class, the method will get the instance object as its first argument (which is usually called `self`). See function and nested scope.

method resolution order

Method Resolution Order is the order in which base classes are searched for a member during lookup. See The Python 2.3 Method Resolution Order for details of the algorithm used by the Python interpreter since the 2.3 release.

module

An object that serves as an organizational unit of Python code. Modules have a namespace containing arbitrary

Python objects. Modules are loaded into Python by the process of importing.

See also package.

module spec

A namespace containing the import-related information used to load a module. An instance of `importlib.machinery.ModuleSpec`.

See also Module specs.

MRO

See method resolution order.

mutable

An object with state that is allowed to change during the course of the program. In multi-threaded programs, mutable objects that are shared between threads require careful synchronization to avoid race conditions. See also immutable, thread-safe, and concurrent modification.

named tuple

The term “named tuple” applies to any type or class that inherits from tuple and whose indexable elements are also accessible using named attributes. The type or class may have other features as well.

Several built-in types are named tuples, including the values returned by `time.localtime()` and `os.stat()`. Another example is `sys.float_info`:

```
>>> sys.float_info[1]                   # indexed access
1024
>>> sys.float_info.max_exp              # named field access
1024
```

```
>>> isinstance(sys.float_info, tuple)   # kind of tuple
True
```

Some named tuples are built-in types (such as the above examples). Alternatively, a named tuple can be created from a regular class definition that inherits from **`tuple`** and that defines named fields. Such a class can be written by hand, or it can be created by inheriting **`typing.NamedTuple`**, or with the factory function **`collections.namedtuple()`**. The latter techniques also add some extra methods that may not be found in hand-written or built-in named tuples.

namespace

The place where a variable is stored. Namespaces are implemented as dictionaries. There are the local, global and built-in namespaces as well as nested namespaces in objects (in methods). Namespaces support modularity by preventing naming conflicts. For instance, the functions **`builtins.open`** and **`os.open()`** are distinguished by their namespaces. Namespaces also aid readability and maintainability by making it clear which module implements a function. For instance, writing **`random.seed()`** or **`itertools.islice()`** makes it clear that those functions are implemented by the **`random`** and **`itertools`** modules, respectively.

namespace package

A package which serves only as a container for subpackages. Namespace packages may have no physical representation, and specifically are not like a regular package because they have no `__init__.py` file.

Namespace packages allow several individually installable packages to have a common parent package. Otherwise, it is recommended to use a regular package.

For more information, see **PEP 420** [https://peps.python.org/pep-0420/] and Namespace packages.

See also module.

native code

Code that is compiled to machine instructions and runs directly on the processor, as opposed to code that is interpreted or runs in a virtual machine. In the context of Python, native code typically refers to C, C++, Rust or Fortran code in extension modules that can be called from Python. See also extension module.

nested scope

The ability to refer to a variable in an enclosing definition. For instance, a function defined inside another function can refer to variables in the outer function. Note that nested scopes by default work only for reference and not for assignment. Local variables both read and write in the innermost scope. Likewise, global variables read and write to the global namespace. The `nonlocal` allows writing to outer scopes.

new-style class

Old name for the flavor of classes now used for all class objects. In earlier Python versions, only new-style classes could use Python’s newer, versatile features like `__slots__`, descriptors, properties, `__getattribute__()`, class methods, and static methods.

non-deterministic

Behavior where the outcome of a program can vary between executions with the same inputs. In multi-threaded programs, non-deterministic behavior often results from race conditions where the relative timing or interleaving of threads affects the result. Proper synchronization using locks and other synchronization primitives helps ensure deterministic behavior.

object

Any data with state (attributes or value) and defined behavior (methods). Also the ultimate base class of any new-style class.

optimized scope

A scope where target local variable names are reliably known to the compiler when the code is compiled, allowing optimization of read and write access to these names. The local namespaces for functions, generators, coroutines, comprehensions, and generator expressions are optimized in this fashion. Note: most interpreter optimizations are applied to all scopes, only those relying on a known set of local and nonlocal variable names are restricted to optimized scopes.

optional module

An extension module that is part of the standard library, but may be absent in some builds of CPython, usually due to missing third-party libraries or because the module is not available for a given platform.

See Requirements for optional modules for a list of optional modules that require third-party libraries.

package

A Python module which can contain submodules or recursively, subpackages. Technically, a package is a Python module with a `__path__` attribute.

See also regular package and namespace package.

parallelism

Executing multiple operations at the same time (e.g. on multiple CPU cores). In Python builds with the global interpreter lock (GIL), only one thread runs Python bytecode at a time, so taking advantage of multiple CPU cores typically involves multiple processes (e.g. **`multiprocessing`**) or native extensions that release the GIL. In free-threaded Python, multiple Python threads can run Python code simultaneously on different cores.

parameter

A named entity in a function (or method) definition that specifies an argument (or in some cases, arguments) that the function can accept. There are five kinds of parameter:

- *positional-or-keyword*: specifies an argument that can be passed either positionally or as a keyword argument. This is the default kind of parameter, for example *foo* and *bar* in the following:

  ```
  def func(foo, bar=None): ...
  ```

- *positional-only*: specifies an argument that can be supplied only by position. Positional-only parameters can be defined by including a `/` character in the parameter list of the function definition after them, for example *posonly1* and *posonly2* in the following:

  ```
  def func(posonly1, posonly2, /, positional_or_keyword):
  ...
  ```

- *keyword-only*: specifies an argument that can be supplied only by keyword. Keyword-only parameters can be defined by including a single var-positional parameter or bare `*` in the parameter list of the function definition before them, for example *kw_only1* and *kw_only2* in the following:

```
def func(arg, *, kw_only1, kw_only2): ...
```

- *var-positional*: specifies that an arbitrary sequence of positional arguments can be provided (in addition to any positional arguments already accepted by other parameters). Such a parameter can be defined by prepending the parameter name with `*`, for example *args* in the following:

  ```
  def func(*args, **kwargs): ...
  ```

- *var-keyword*: specifies that arbitrarily many keyword arguments can be provided (in addition to any keyword arguments already accepted by other parameters). Such a parameter can be defined by prepending the parameter name with `**`, for example *kwargs* in the example above.

Parameters can specify both optional and required arguments, as well as default values for some optional arguments.

See also the argument glossary entry, the FAQ question on the difference between arguments and parameters, the **`inspect.Parameter`** class, the Function definitions section, and **PEP 362** [https://peps.python.org/pep-0362/].

path entry

A single location on the import path which the path based finder consults to find modules for importing.

path entry finder

A finder returned by a callable on **`sys.path_hooks`** (i.e. a path entry hook) which knows how to locate modules given a path entry.

See **`importlib.abc.PathEntryFinder`** for the methods that path entry finders implement.

path entry hook

A callable on the `sys.path_hooks` list which returns a path entry finder if it knows how to find modules on a specific path entry.

path based finder

One of the default meta path finders which searches an import path for modules.

path-like object

An object representing a file system path. A path-like object is either a `str` or `bytes` object representing a path, or an object implementing the `os.PathLike` protocol. An object that supports the `os.PathLike` protocol can be converted to a `str` or `bytes` file system path by calling the `os.fspath()` function; `os.fsdecode()` and `os.fsencode()` can be used to guarantee a `str` or `bytes` result instead, respectively. Introduced by **PEP 519** [https://peps.python.org/pep-0519/].

PEP

Python Enhancement Proposal. A PEP is a design document providing information to the Python community, or describing a new feature for Python or its processes or environment. PEPs should provide a concise technical specification and a rationale for proposed features.

PEPs are intended to be the primary mechanisms for proposing major new features, for collecting community input on an issue, and for documenting the design decisions that have gone into Python. The PEP author is responsible for building consensus within the community and documenting dissenting opinions.

See **PEP 1** [https://peps.python.org/pep-0001/].

portion

A set of files in a single directory (possibly stored in a zip file) that contribute to a namespace package, as defined in **PEP 420** [https://peps.python.org/pep-0420/].

positional argument

See argument.

provisional API

A provisional API is one which has been deliberately excluded from the standard library's backwards compatibility guarantees. While major changes to such interfaces are not expected, as long as they are marked provisional, backwards incompatible changes (up to and including removal of the interface) may occur if deemed necessary by core developers. Such changes will not be made gratuitously – they will occur only if serious fundamental flaws are uncovered that were missed prior to the inclusion of the API.

Even for provisional APIs, backwards incompatible changes are seen as a "solution of last resort" - every attempt will still be made to find a backwards compatible resolution to any identified problems.

This process allows the standard library to continue to evolve over time, without locking in problematic design errors for extended periods of time. See **PEP 411** [https://peps.python.org/pep-0411/] for more details.

provisional package

See provisional API.

Python 3000

Nickname for the Python 3.x release line (coined long ago when the release of version 3 was something in the distant future.) This is also abbreviated “Py3k”.

Pythonic

An idea or piece of code which closely follows the most common idioms of the Python language, rather than implementing code using concepts common to other languages. For example, a common idiom in Python is to loop over all elements of an iterable using a `for` statement. Many other languages don’t have this type of construct, so people unfamiliar with Python sometimes use a numerical counter instead:

```
for i in range(len(food)):
    print(food[i])
```

As opposed to the cleaner, Pythonic method:

```
for piece in food:
    print(piece)
```

qualified name

A dotted name showing the “path” from a module’s global scope to a class, function or method defined in that module, as defined in **PEP 3155** [https://peps.python.org/pep-3155/]. For top-level functions and classes, the qualified name is the same as the object’s name:

```
>>> class C:
...     class D:
...         def meth(self):
...             pass
...
>>> C.__qualname__
'C'
>>> C.D.__qualname__
'C.D'
```

```
>>> C.D.meth.__qualname__
'C.D.meth'
```

When used to refer to modules, the *fully qualified name* means the entire dotted path to the module, including any parent packages, e.g. `email.mime.text`:

```
>>> import email.mime.text
>>> email.mime.text.__name__
'email.mime.text'
```

race condition

A condition of a program where the its behavior depends on the relative timing or ordering of events, particularly in multi-threaded programs. Race conditions can lead to non-deterministic behavior and bugs that are difficult to reproduce. A data race is a specific type of race condition involving unsynchronized access to shared memory. The LBYL coding style is particularly susceptible to race conditions in multi-threaded code. Using locks and other synchronization primitives helps prevent race conditions.

reference count

The number of references to an object. When the reference count of an object drops to zero, it is deallocated. Some objects are immortal and have reference counts that are never modified, and therefore the objects are never deallocated. Reference counting is generally not visible to Python code, but it is a key element of the CPython implementation. Programmers can call the `sys.getrefcount()` function to return the reference count for a particular object.

In CPython, reference counts are not considered to be stable or well-defined values; the number of references to an object, and how that number is affected by Python code, may be different between versions.

regular package

A traditional package, such as a directory containing an `__init__.py` file.

See also namespace package.

reentrant

A property of a function or lock that allows it to be called or acquired multiple times by the same thread without causing errors or a deadlock.

For functions, reentrancy means the function can be safely called again before a previous invocation has completed, which is important when functions may be called recursively or from signal handlers. Thread-unsafe functions may be non-deterministic if they're called reentrantly in a multithreaded program.

For locks, Python's **`threading.RLock`** (reentrant lock) is reentrant, meaning a thread that already holds the lock can acquire it again without blocking. In contrast, **`threading.Lock`** is not reentrant - attempting to acquire it twice from the same thread will cause a deadlock.

See also lock and deadlock.

REPL

An acronym for the "read–eval–print loop", another name for the interactive interpreter shell.

__slots__

A declaration inside a class that saves memory by pre-declaring space for instance attributes and eliminating instance dictionaries. Though popular, the technique is somewhat tricky to get right and is best reserved for rare cases where there are large numbers of instances in a memory-critical application.

sequence

An iterable which supports efficient element access using integer indices via the `__getitem__()` special method and defines a `__len__()` method that returns the length of the sequence. Some built-in sequence types are `list`, `str`, `tuple`, and `bytes`. Note that `dict` also supports `__getitem__()` and `__len__()`, but is considered a mapping rather than a sequence because the lookups use arbitrary hashable keys rather than integers.

The `collections.abc.Sequence` abstract base class defines a much richer interface that goes beyond just `__getitem__()` and `__len__()`, adding `count()`, `index()`, `__contains__()`, and `__reversed__()`. Types that implement this expanded interface can be registered explicitly using `register()`. For more documentation on sequence methods generally, see Common Sequence Operations.

set comprehension

A compact way to process all or part of the elements in an iterable and return a set with the results. `results = {c for c in 'abracadabra' if c not in 'abc'}` generates the set of strings `{'r', 'd'}`. See Displays for lists, sets and dictionaries.

single dispatch

A form of generic function dispatch where the implementation is chosen based on the type of a single argument.

slice

An object usually containing a portion of a sequence. A slice is created using the subscript notation, `[]` with colons between numbers when several are given, such as in `variable_name[1:3:5]`. The bracket (subscript) notation uses **`slice`** objects internally.

soft deprecated

A soft deprecated API should not be used in new code, but it is safe for already existing code to use it. The API remains documented and tested, but will not be enhanced further.

Soft deprecation, unlike normal deprecation, does not plan on removing the API and will not emit warnings.

See PEP 387: Soft Deprecation [https://peps.python.org/pep-0387/#soft-deprecation].

special method

A method that is called implicitly by Python to execute a certain operation on a type, such as addition. Such methods have names starting and ending with double underscores. Special methods are documented in Special method names.

standard library

The collection of packages, modules and extension modules distributed as a part of the official Python interpreter package. The exact membership of the collection may vary based on platform, available system libraries, or other criteria. Documentation can be found at The Python Standard Library.

See also **`sys.stdlib_module_names`** for a list of all possible standard library module names.

statement

A statement is part of a suite (a “block” of code). A statement is either an expression or one of several constructs with a keyword, such as **`if`**, **`while`** or **`for`**.

static type checker

An external tool that reads Python code and analyzes it, looking for issues such as incorrect types. See also type hints and the **`typing`** module.

stdlib

An abbreviation of standard library.

strong reference

In Python’s C API, a strong reference is a reference to an object which is owned by the code holding the reference. The strong reference is taken by calling **`Py_INCREF()`** when the reference is created and released with **`Py_DECREF()`** when the reference is deleted.

The **`Py_NewRef()`** function can be used to create a strong reference to an object. Usually, the **`Py_DECREF()`** function must be called on the strong reference before exiting the scope of the strong reference, to avoid leaking one reference.

See also borrowed reference.

synchronization primitive

A basic building block for coordinating (synchronizing) the execution of multiple threads to ensure thread-safe access to shared resources. Python’s **`threading`** module provides several synchronization primitives including **`Lock`**, **`RLock`**, **`Semaphore`**, **`Condition`**, **`Event`**, and **`Barrier`**. Additionally, the **`queue`** module provides multi-producer, multi-consumer queues that are especially useful in multithreaded programs. These primitives help prevent race conditions and coordinate thread execution. See also lock.

t-string
t-strings

String literals prefixed with `t` or `T` are commonly called “t-strings” which is short for template string literals.

text encoding

A string in Python is a sequence of Unicode code points (in range `U+0000`–`U+10FFFF`). To store or transfer a string, it needs to be serialized as a sequence of bytes.

Serializing a string into a sequence of bytes is known as “encoding”, and recreating the string from the sequence of bytes is known as “decoding”.

There are a variety of different text serialization codecs, which are collectively referred to as “text encodings”.

text file

A file object able to read and write **`str`** objects. Often, a text file actually accesses a byte-oriented datastream and handles the text encoding automatically. Examples of text files are files opened in text mode (`'r'` or `'w'`), **`sys.stdin`**, **`sys.stdout`**, and instances of **`io.StringIO`**.

See also binary file for a file object able to read and write bytes-like objects.

thread state

The information used by the CPython runtime to run in an OS thread. For example, this includes the current exception, if any, and the state of the bytecode interpreter.

Each thread state is bound to a single OS thread, but threads may have many thread states available. At most, one of them may be attached at once.

An attached thread state is required to call most of Python’s C API, unless a function explicitly documents otherwise. The bytecode interpreter only runs under an attached thread state.

Each thread state belongs to a single interpreter, but each interpreter may have many thread states, including multiple for the same OS thread. Thread states from multiple interpreters may be bound to the same thread, but only one can be attached in that thread at any given moment.

See Thread State and the Global Interpreter Lock for more information.

thread-safe

A module, function, or class that behaves correctly when used by multiple threads concurrently. Thread-safe code uses appropriate synchronization primitives like locks to protect shared mutable state, or is designed to avoid shared mutable state entirely. In the free-threaded build, built-in types like `dict`, `list`, and `set` use internal locking to make many operations thread-safe, although thread safety is not necessarily guaranteed. Code that is not thread-safe may experience race conditions and data races when used in multi-threaded programs.

token

A small unit of source code, generated by the lexical analyzer (also called the *tokenizer*). Names, numbers, strings, operators, newlines and similar are represented by tokens.

The `tokenize` module exposes Python’s lexical analyzer. The `token` module contains information on the various types of tokens.

triple-quoted string

A string which is bound by three instances of either a quotation mark (”) or an apostrophe (‘). While they don’t provide any functionality not available with single-quoted strings, they are useful for a number of reasons. They allow you to include unescaped single and double quotes within a string and they can span multiple lines without the use of the continuation character, making them especially useful when writing docstrings.

type

The type of a Python object determines what kind of object it is; every object has a type. An object’s type is accessible as its `__class__` attribute or can be retrieved with `type(obj)`.

type alias

A synonym for a type, created by assigning the type to an identifier.

Type aliases are useful for simplifying type hints. For example:

```
def remove_gray_shades(
        colors: list[tuple[int, int, int]]) ->
list[tuple[int, int, int]]:
    pass
```

could be made more readable like this:

```
Color = tuple[int, int, int]

def remove_gray_shades(colors: list[Color]) -> list[Color]:
    pass
```

See **`typing`** and **PEP 484** [https://peps.python.org/pep-0484/], which describe this functionality.

type hint

An annotation that specifies the expected type for a variable, a class attribute, or a function parameter or return value.

Type hints are optional and are not enforced by Python but they are useful to static type checkers. They can also aid IDEs with code completion and refactoring.

Type hints of global variables, class attributes, and functions, but not local variables, can be accessed using **`typing.get_type_hints()`**.

See **`typing`** and **PEP 484** [https://peps.python.org/pep-0484/], which describe this functionality.

universal newlines

A manner of interpreting text streams in which all of the following are recognized as ending a line: the Unix end-of-line convention `'\n'`, the Windows convention `'\r\n'`, and the old Macintosh convention `'\r'`. See **PEP 278** [https://peps.python.org/pep-0278/] and **PEP 3116** [https://peps.python.org/pep-3116/], as well as **`bytes.splitlines()`** for an additional use.

variable annotation

An annotation of a variable or a class attribute.

When annotating a variable or a class attribute, assignment is optional:

```
class C:
    field: 'annotation'
```

Variable annotations are usually used for type hints: for example this variable is expected to take `int` values:

```
count: int = 0
```

Variable annotation syntax is explained in section Annotated assignment statements.

See function annotation, **PEP 484** [https://peps.python.org/pep-0484/] and **PEP 526** [https://peps.python.org/pep-0526/], which describe this functionality. Also see Annotations Best Practices for best practices on working with annotations.

virtual environment

A cooperatively isolated runtime environment that allows Python users and applications to install and upgrade Python distribution packages without interfering with the behaviour of other Python applications running on the same system.

See also `venv`.

virtual machine

A computer defined entirely in software. Python’s virtual machine executes the bytecode emitted by the bytecode compiler.

walrus operator

A light-hearted way to refer to the assignment expression operator `:=` because it looks a bit like a walrus if you turn your head.

Zen of Python

Listing of Python design principles and philosophies that are helpful in understanding and using the language. The listing can be found by typing “`import this`” at the interactive prompt.

About this documentation

Python's documentation is generated from reStructuredText [https://docutils.sourceforge.io/rst.html] sources using Sphinx [https://www.sphinx-doc.org/], a documentation generator originally created for Python and now maintained as an independent project.

Development of the documentation and its toolchain is an entirely volunteer effort, just like Python itself. If you want to contribute, please take a look at the Dealing with Bugs page for information on how to do so. New volunteers are always welcome!

Many thanks go to:

- Fred L. Drake, Jr., the creator of the original Python documentation toolset and author of much of the content;
- the Docutils [https://docutils.sourceforge.io/] project for creating reStructuredText and the Docutils suite;
- Fredrik Lundh for his Alternative Python Reference project from which Sphinx got many good ideas.

Contributors to the Python documentation

Many people have contributed to the Python language, the Python standard library, and the Python documentation. See Misc/ACKS [https://github.com/python/cpython/tree/3.14/Misc/ACKS] in the Python source distribution for a partial list of contributors.

It is only with the input and contributions of the Python community that Python has such wonderful documentation – Thank You!

Dealing with Bugs

Python is a mature programming language which has established a reputation for stability. In order to maintain this reputation, the developers would like to know of any deficiencies you find in Python.

It can be sometimes faster to fix bugs yourself and contribute patches to Python as it streamlines the process and involves less people. Learn how to contribute.

Documentation bugs

If you find a bug in this documentation or would like to propose an improvement, please submit a bug report on the issue tracker. If you have a suggestion on how to fix it, include that as well.

You can also open a discussion item on our Documentation Discourse forum [https://discuss.python.org/c/documentation/26].

If you find a bug in the theme (HTML / CSS / JavaScript) of the documentation, please submit a bug report on the python-doc-theme issue tracker [https://github.com/python/python-docs-theme].

See also

Documentation bugs [https://github.com/python/cpython/issues?q=is%3Aissue+is%3Aopen+label%3Adocs]
A list of documentation bugs that have been submitted to the Python issue tracker.

Issue Tracking **[https://devguide.python.org/tracker/]**
Overview of the process involved in reporting an improvement on the tracker.

Helping with Documentation **[https://devguide.python.org/docquality/#helping-with-documentation]**
Comprehensive guide for individuals that are interested in contributing to Python documentation.

Documentation Translations **[https://devguide.python.org/documentation/translating/]**
A list of GitHub pages for documentation translation and their primary contacts.

Using the Python issue tracker

Issue reports for Python itself should be submitted via the GitHub issues tracker (https://github.com/python/cpython/issues). The GitHub issues tracker offers a web form which allows pertinent information to be entered and submitted to the developers.

The first step in filing a report is to determine whether the problem has already been reported. The advantage in doing so, aside from saving the developers' time, is that you learn what has been done to fix it; it may be that the problem has already been fixed for the next release, or additional information is needed (in which case you are welcome to provide it if you can!). To do this, search the tracker using the search box at the top of the page.

If the problem you're reporting is not already in the list, log in to GitHub. If you don't already have a GitHub account, create a new account using the “Sign up” link. It is not possible to submit a bug report anonymously.

Being now logged in, you can submit an issue. Click on the “New issue” button in the top bar to report a new issue.

The submission form has two fields, “Title” and “Comment”.

For the “Title” field, enter a *very* short description of the problem; fewer than ten words is good.

In the “Comment” field, describe the problem in detail, including what you expected to happen and what did happen. Be sure to include whether any extension modules were involved, and what hardware and software platform you were using (including version information as appropriate).

Each issue report will be reviewed by a developer who will determine what needs to be done to correct the problem. You will receive an update each time an action is taken on the issue.

See also

How to Report Bugs Effectively [https://www.chiark.greenend.org.uk/~sgtatham/bugs.html]
Article which goes into some detail about how to create a useful bug report. This describes what kind of information is useful and why it is useful.

Bug Writing Guidelines [https://bugzilla.mozilla.org/page.cgi?id=bug-writing.html]

> Information about writing a good bug report. Some of this is specific to the Mozilla project, but describes general good practices.

Getting started contributing to Python yourself

Beyond just reporting bugs that you find, you are also welcome to submit patches to fix them. You can find more information on how to get started patching Python in the Python Developer's Guide [https://devguide.python.org/]. If you have questions, the core-mentorship mailing list [https://mail.python.org/mailman3/lists/core-mentorship.python.org/] is a friendly place to get answers to any and all questions pertaining to the process of fixing issues in Python.

Copyright

Python and this documentation is:

Copyright © 2001 Python Software Foundation. All rights reserved.

Copyright © 2000 BeOpen.com. All rights reserved.

Copyright © 1995-2000 Corporation for National Research Initiatives. All rights reserved.

Copyright © 1991-1995 Stichting Mathematisch Centrum. All rights reserved.

See History and License for complete license and permissions information.

History and License

History of the software

Python was created in the early 1990s by Guido van Rossum at Stichting Mathematisch Centrum (CWI, see https://www.cwi.nl) in the Netherlands as a successor of a language called ABC. Guido remains Python’s principal author, although it includes many contributions from others.

In 1995, Guido continued his work on Python at the Corporation for National Research Initiatives (CNRI, see https://www.cnri.reston.va.us) in Reston, Virginia where he released several versions of the software.

In May 2000, Guido and the Python core development team moved to BeOpen.com to form the BeOpen PythonLabs team. In October of the same year, the PythonLabs team moved to Digital Creations, which became Zope Corporation. In 2001, the Python Software Foundation (PSF, see https://www.python.org/psf/) was formed, a non-profit organization created specifically to own Python-related Intellectual Property. Zope Corporation was a sponsoring member of the PSF.

All Python releases are Open Source (see https://opensource.org for the Open Source Definition). Historically, most, but not all, Python releases have also been GPL-compatible; the table below summarizes the various releases.

Release	Derived from	Year	Owner	GPL-compatible? (1)
0.9.0 thru 1.2	n/a	1991-1995	CWI	yes
1.3 thru 1.5.2	1.2	1995-1999	CNRI	yes
1.6	1.5.2	2000	CNRI	no
2.0	1.6	2000	BeOpen.com	no
1.6.1	1.6	2001	CNRI	yes (2)
2.1	2.0+1.6.1	2001	PSF	no
2.0.1	2.0+1.6.1	2001	PSF	yes
2.1.1	2.1+2.0.1	2001	PSF	yes
2.1.2	2.1.1	2002	PSF	yes
2.1.3	2.1.2	2002	PSF	yes

Release	Derived from	Year	Owner	GPL-compatible? (1)
2.2 and above	2.1.1	2001-now	PSF	yes

Note

1. GPL-compatible doesn’t mean that we’re distributing Python under the GPL. All Python licenses, unlike the GPL, let you distribute a modified version without making your changes open source. The GPL-compatible licenses make it possible to combine Python with other software that is released under the GPL; the others don’t.
2. According to Richard Stallman, 1.6.1 is not GPL-compatible, because its license has a choice of law clause. According to CNRI, however, Stallman’s lawyer has told CNRI’s lawyer that 1.6.1 is “not incompatible” with the GPL.

Thanks to the many outside volunteers who have worked under Guido’s direction to make these releases possible.

Terms and conditions for accessing or otherwise using Python

Python software and documentation are licensed under the Python Software Foundation License Version 2.

Starting with Python 3.8.6, examples, recipes, and other code in the documentation are dual licensed under the PSF License Version 2 and the Zero-Clause BSD license.

Some software incorporated into Python is under different licenses. The licenses are listed with code falling under that license. See Licenses and Acknowledgements for Incorporated Software for an incomplete list of these licenses.

PYTHON SOFTWARE FOUNDATION LICENSE VERSION 2

```
1. This LICENSE AGREEMENT is between the Python Software
Foundation ("PSF"), and
   the Individual or Organization ("Licensee") accessing and
otherwise using this
   software ("Python") in source or binary form and its
associated documentation.

2. Subject to the terms and conditions of this License
Agreement, PSF hereby
   grants Licensee a nonexclusive, royalty-free, world-wide
license to reproduce,
   analyze, test, perform and/or display publicly, prepare
derivative works,
   distribute, and otherwise use Python alone or in any
derivative
   version, provided, however, that PSF's License Agreement and
PSF's notice of
   copyright, i.e., "Copyright © 2001 Python Software
Foundation; All Rights
   Reserved" are retained in Python alone or in any derivative
version
   prepared by Licensee.

3. In the event Licensee prepares a derivative work that is
based on or
   incorporates Python or any part thereof, and wants to make
the
```

derivative work available to others as provided herein, then Licensee hereby agrees to include in any such work a brief summary of the changes made to Python.

4. PSF is making Python available to Licensee on an "AS IS" basis. PSF MAKES NO REPRESENTATIONS OR WARRANTIES, EXPRESS OR IMPLIED. BY WAY OF EXAMPLE, BUT NOT LIMITATION, PSF MAKES NO AND DISCLAIMS ANY REPRESENTATION OR WARRANTY OF MERCHANTABILITY OR FITNESS FOR ANY PARTICULAR PURPOSE OR THAT THE USE OF PYTHON WILL NOT INFRINGE ANY THIRD PARTY RIGHTS.

5. PSF SHALL NOT BE LIABLE TO LICENSEE OR ANY OTHER USERS OF PYTHON FOR ANY INCIDENTAL, SPECIAL, OR CONSEQUENTIAL DAMAGES OR LOSS AS A RESULT OF MODIFYING, DISTRIBUTING, OR OTHERWISE USING PYTHON, OR ANY DERIVATIVE THEREOF, EVEN IF ADVISED OF THE POSSIBILITY THEREOF.

6. This License Agreement will automatically terminate upon a material breach of its terms and conditions.

7. Nothing in this License Agreement shall be deemed to create any relationship of agency, partnership, or joint venture between PSF and Licensee. This License Agreement does not grant permission to use PSF trademarks or trade name in a trademark sense to endorse or promote products or services of Licensee, or any third party.

8. By copying, installing or otherwise using Python, Licensee agrees to be bound by the terms and conditions of this License Agreement.

BEOPEN.COM LICENSE AGREEMENT FOR PYTHON 2.0

BEOPEN PYTHON OPEN SOURCE LICENSE AGREEMENT VERSION 1

1. This LICENSE AGREEMENT is between BeOpen.com ("BeOpen"), having an office at 160 Saratoga Avenue, Santa Clara, CA 95051, and the Individual or Organization ("Licensee") accessing and otherwise using this software in source or binary form and its associated documentation ("the Software").

2. Subject to the terms and conditions of this BeOpen Python License Agreement, BeOpen hereby grants Licensee a non-exclusive, royalty-free, world-wide license to reproduce, analyze, test, perform and/or display publicly, prepare derivative works, distribute, and otherwise use the Software alone or in any derivative version, provided, however, that the BeOpen Python License is retained in the Software, alone or in any derivative version prepared by Licensee.

3. BeOpen is making the Software available to Licensee on an "AS IS" basis. BEOPEN MAKES NO REPRESENTATIONS OR WARRANTIES, EXPRESS OR IMPLIED. BY WAY OF EXAMPLE, BUT NOT LIMITATION, BEOPEN MAKES NO AND DISCLAIMS ANY REPRESENTATION OR WARRANTY OF MERCHANTABILITY OR FITNESS FOR ANY PARTICULAR PURPOSE OR THAT THE USE OF THE SOFTWARE WILL NOT INFRINGE ANY THIRD PARTY RIGHTS.

4. BEOPEN SHALL NOT BE LIABLE TO LICENSEE OR ANY OTHER USERS OF THE SOFTWARE FOR ANY INCIDENTAL, SPECIAL, OR CONSEQUENTIAL DAMAGES OR LOSS AS

A RESULT OF USING, MODIFYING OR DISTRIBUTING THE SOFTWARE, OR ANY DERIVATIVE THEREOF, EVEN IF ADVISED OF THE POSSIBILITY THEREOF.

5. This License Agreement will automatically terminate upon a material breach of its terms and conditions.

6. This License Agreement shall be governed by and interpreted in all respects by the law of the State of California, excluding conflict of law provisions. Nothing in this License Agreement shall be deemed to create any relationship of agency, partnership, or joint venture between BeOpen and Licensee. This License Agreement does not grant permission to use BeOpen trademarks or trade names in a trademark sense to endorse or promote products or services of Licensee, or any third party. As an exception, the "BeOpen Python" logos available at http://www.pythonlabs.com/logos.html may be used according to the permissions granted on that web page.

7. By copying, installing or otherwise using the software, Licensee agrees to be bound by the terms and conditions of this License Agreement.

CNRI LICENSE AGREEMENT FOR PYTHON 1.6.1

1. This LICENSE AGREEMENT is between the Corporation for National Research Initiatives, having an office at 1895 Preston White Drive, Reston, VA 20191 ("CNRI"), and the Individual or Organization ("Licensee") accessing and otherwise using Python 1.6.1 software in source or binary form and its associated documentation.

2. Subject to the terms and conditions of this License Agreement, CNRI hereby grants Licensee a nonexclusive, royalty-free, world-wide license to reproduce, analyze, test, perform and/or display publicly, prepare derivative works, distribute, and otherwise use Python 1.6.1 alone or in any derivative version, provided, however, that CNRI's License Agreement and CNRI's notice of copyright, i.e., "Copyright © 1995-2001 Corporation for National Research Initiatives; All Rights Reserved" are retained in Python 1.6.1 alone or in any derivative version prepared by Licensee. Alternately, in lieu of CNRI's License Agreement, Licensee may substitute the following text (omitting the quotes): "Python 1.6.1 is made available subject to the terms and conditions in CNRI's License Agreement. This Agreement together with Python 1.6.1 may be located on the internet using the following unique, persistent identifier (known as a handle): 1895.22/1013. This Agreement may also be obtained from a proxy server on the internet using the following URL: http://hdl.handle.net/1895.22/1013".

3. In the event Licensee prepares a derivative work that is based on or incorporates Python 1.6.1 or any part thereof, and wants to make the derivative work available to others as provided herein, then Licensee hereby agrees to include in any such work a brief summary of the changes made to Python 1.6.1.

4. CNRI is making Python 1.6.1 available to Licensee on an "AS IS" basis. CNRI MAKES NO REPRESENTATIONS OR WARRANTIES, EXPRESS OR IMPLIED. BY WAY OF EXAMPLE,

BUT NOT LIMITATION, CNRI MAKES NO AND DISCLAIMS ANY REPRESENTATION OR WARRANTY OF MERCHANTABILITY OR FITNESS FOR ANY PARTICULAR PURPOSE OR THAT THE USE OF PYTHON 1.6.1 WILL NOT INFRINGE ANY THIRD PARTY RIGHTS.

5. CNRI SHALL NOT BE LIABLE TO LICENSEE OR ANY OTHER USERS OF PYTHON 1.6.1 FOR ANY INCIDENTAL, SPECIAL, OR CONSEQUENTIAL DAMAGES OR LOSS AS A RESULT OF MODIFYING, DISTRIBUTING, OR OTHERWISE USING PYTHON 1.6.1, OR ANY DERIVATIVE THEREOF, EVEN IF ADVISED OF THE POSSIBILITY THEREOF.

6. This License Agreement will automatically terminate upon a material breach of its terms and conditions.

7. This License Agreement shall be governed by the federal intellectual property law of the United States, including without limitation the federal copyright law, and, to the extent such U.S. federal law does not apply, by the law of the Commonwealth of Virginia, excluding Virginia's conflict of law provisions. Notwithstanding the foregoing, with regard to derivative works based on Python 1.6.1 that incorporate non-separable material that was previously distributed under the GNU General Public License (GPL), the law of the Commonwealth of Virginia shall govern this License Agreement only as to issues arising under or with respect to Paragraphs 4, 5, and 7 of this License Agreement. Nothing in this License Agreement shall be deemed to create any relationship of agency, partnership, or joint venture between CNRI and Licensee. This License Agreement does not grant permission to use CNRI trademarks or trade name in a trademark sense to endorse or promote products or services of

Licensee, or any third party.

8. By clicking on the "ACCEPT" button where indicated, or by copying, installing or otherwise using Python 1.6.1, Licensee agrees to be bound by the terms and conditions of this License Agreement.

CWI LICENSE AGREEMENT FOR PYTHON 0.9.0 THROUGH 1.2

Copyright © 1991 - 1995, Stichting Mathematisch Centrum Amsterdam, The Netherlands. All rights reserved.

Permission to use, copy, modify, and distribute this software and its documentation for any purpose and without fee is hereby granted, provided that the above copyright notice appear in all copies and that both that copyright notice and this permission notice appear in supporting documentation, and that the name of Stichting Mathematisch Centrum or CWI not be used in advertising or publicity pertaining to distribution of the software without specific, written prior permission.

STICHTING MATHEMATISCH CENTRUM DISCLAIMS ALL WARRANTIES WITH REGARD TO THIS SOFTWARE, INCLUDING ALL IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS, IN NO EVENT SHALL STICHTING MATHEMATISCH CENTRUM BE LIABLE FOR ANY SPECIAL, INDIRECT OR CONSEQUENTIAL DAMAGES OR ANY DAMAGES WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS, WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE OR

```
PERFORMANCE OF THIS
SOFTWARE.
```

ZERO-CLAUSE BSD LICENSE FOR CODE IN THE PYTHON DOCUMENTATION

```
Permission to use, copy, modify, and/or distribute this
software for any
purpose with or without fee is hereby granted.

THE SOFTWARE IS PROVIDED "AS IS" AND THE AUTHOR DISCLAIMS ALL
WARRANTIES WITH
REGARD TO THIS SOFTWARE INCLUDING ALL IMPLIED WARRANTIES OF
MERCHANTABILITY
AND FITNESS. IN NO EVENT SHALL THE AUTHOR BE LIABLE FOR ANY
SPECIAL, DIRECT,
INDIRECT, OR CONSEQUENTIAL DAMAGES OR ANY DAMAGES WHATSOEVER
RESULTING FROM
LOSS OF USE, DATA OR PROFITS, WHETHER IN AN ACTION OF CONTRACT,
NEGLIGENCE OR
OTHER TORTIOUS ACTION, ARISING OUT OF OR IN CONNECTION WITH THE
USE OR
PERFORMANCE OF THIS SOFTWARE.
```

Licenses and Acknowledgements for Incorporated Software

This section is an incomplete, but growing list of licenses and acknowledgements for third-party software incorporated in the Python distribution.

Mersenne Twister

The **`_random`** C extension underlying the **`random`** module includes code based on a download from http://www.math.sci.hiroshima-u.ac.jp/~m-mat/MT/MT2002/emt19937ar.html. The following are the verbatim comments from the original code:

A C-program for MT19937, with initialization improved 2002/1/26.
Coded by Takuji Nishimura and Makoto Matsumoto.

Before using, initialize the state by using init_genrand(seed) or init_by_array(init_key, key_length).

Copyright (C) 1997 - 2002, Makoto Matsumoto and Takuji Nishimura,
All rights reserved.

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

1. Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.

2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.

3. The names of its contributors may not be used to endorse or promote products derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR

```
CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL,
SPECIAL,
EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED
TO,
PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA,
OR
PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY
THEORY OF
LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT
(INCLUDING
NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF
THIS
SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.


Any feedback is very welcome.
http://www.math.sci.hiroshima-u.ac.jp/~m-mat/MT/emt.html
email: m-mat @ math.sci.hiroshima-u.ac.jp (remove space)
```

Sockets

The **`socket`** module uses the functions, **`getaddrinfo()`**, and **`getnameinfo()`**, which are coded in separate source files from the WIDE Project, https://www.wide.ad.jp/.

```
Copyright (C) 1995, 1996, 1997, and 1998 WIDE Project.
All rights reserved.

Redistribution and use in source and binary forms, with or
without
modification, are permitted provided that the following
conditions
are met:
1. Redistributions of source code must retain the above
copyright
   notice, this list of conditions and the following
disclaimer.
2. Redistributions in binary form must reproduce the above
copyright
   notice, this list of conditions and the following disclaimer
in the
   documentation and/or other materials provided with the
```

```
distribution.
3. Neither the name of the project nor the names of its
contributors
   may be used to endorse or promote products derived from this
software
   without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE PROJECT AND CONTRIBUTORS "AS
IS" AND
ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED
TO, THE
IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A
PARTICULAR PURPOSE
ARE DISCLAIMED.  IN NO EVENT SHALL THE PROJECT OR CONTRIBUTORS
BE LIABLE
FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR
CONSEQUENTIAL
DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF
SUBSTITUTE GOODS
OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS
INTERRUPTION)
HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN
CONTRACT, STRICT
LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING
IN ANY WAY
OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE
POSSIBILITY OF
SUCH DAMAGE.
```

Asynchronous socket services

The **`test.support.asynchat`** and **`test.support.asyncore`** modules contain the following notice:

```
Copyright 1996 by Sam Rushing

                        All Rights Reserved

Permission to use, copy, modify, and distribute this software
and
its documentation for any purpose and without fee is hereby
granted, provided that the above copyright notice appear in all
```

```
copies and that both that copyright notice and this permission
notice appear in supporting documentation, and that the name of
Sam
Rushing not be used in advertising or publicity pertaining to
distribution of the software without specific, written prior
permission.

SAM RUSHING DISCLAIMS ALL WARRANTIES WITH REGARD TO THIS
SOFTWARE,
INCLUDING ALL IMPLIED WARRANTIES OF MERCHANTABILITY AND
FITNESS, IN
NO EVENT SHALL SAM RUSHING BE LIABLE FOR ANY SPECIAL, INDIRECT
OR
CONSEQUENTIAL DAMAGES OR ANY DAMAGES WHATSOEVER RESULTING FROM
LOSS
OF USE, DATA OR PROFITS, WHETHER IN AN ACTION OF CONTRACT,
NEGLIGENCE OR OTHER TORTIOUS ACTION, ARISING OUT OF OR IN
CONNECTION WITH THE USE OR PERFORMANCE OF THIS SOFTWARE.
```

Cookie management

The `http.cookies` module contains the following notice:

```
Copyright 2000 by Timothy O'Malley <timo@alum.mit.edu>

               All Rights Reserved

Permission to use, copy, modify, and distribute this software
and its documentation for any purpose and without fee is hereby
granted, provided that the above copyright notice appear in all
copies and that both that copyright notice and this permission
notice appear in supporting documentation, and that the name of
Timothy O'Malley  not be used in advertising or publicity
pertaining to distribution of the software without specific,
written
prior permission.

Timothy O'Malley DISCLAIMS ALL WARRANTIES WITH REGARD TO THIS
SOFTWARE, INCLUDING ALL IMPLIED WARRANTIES OF MERCHANTABILITY
AND FITNESS, IN NO EVENT SHALL Timothy O'Malley BE LIABLE FOR
ANY SPECIAL, INDIRECT OR CONSEQUENTIAL DAMAGES OR ANY DAMAGES
WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS,
WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS
ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE OR
PERFORMANCE OF THIS SOFTWARE.
```

Execution tracing

The `trace` module contains the following notice:

```
portions copyright 2001, Autonomous Zones Industries, Inc., all
rights...
err...  reserved and offered to the public under the terms of
the
Python 2.2 license.
Author: Zooko O'Whielacronx
http://zooko.com/
mailto:zooko@zooko.com

Copyright 2000, Mojam Media, Inc., all rights reserved.
Author: Skip Montanaro

Copyright 1999, Bioreason, Inc., all rights reserved.
Author: Andrew Dalke

Copyright 1995-1997, Automatrix, Inc., all rights reserved.
Author: Skip Montanaro

Copyright 1991-1995, Stichting Mathematisch Centrum, all rights
reserved.


Permission to use, copy, modify, and distribute this Python
software and
its associated documentation for any purpose without fee is
hereby
granted, provided that the above copyright notice appears in
all copies,
and that both that copyright notice and this permission notice
appear in
supporting documentation, and that the name of neither
Automatrix,
Bioreason or Mojam Media be used in advertising or publicity
pertaining to
```

```
distribution of the software without specific, written prior
permission.
```

UUencode and UUdecode functions

The `uu` codec contains the following notice:

```
Copyright 1994 by Lance Ellinghouse
Cathedral City, California Republic, United States of America.
                       All Rights Reserved
Permission to use, copy, modify, and distribute this software
and its
documentation for any purpose and without fee is hereby
granted,
provided that the above copyright notice appear in all copies
and that
both that copyright notice and this permission notice appear in
supporting documentation, and that the name of Lance
Ellinghouse
not be used in advertising or publicity pertaining to
distribution
of the software without specific, written prior permission.
LANCE ELLINGHOUSE DISCLAIMS ALL WARRANTIES WITH REGARD TO
THIS SOFTWARE, INCLUDING ALL IMPLIED WARRANTIES OF
MERCHANTABILITY AND
FITNESS, IN NO EVENT SHALL LANCE ELLINGHOUSE CENTRUM BE LIABLE
FOR ANY SPECIAL, INDIRECT OR CONSEQUENTIAL DAMAGES OR ANY
DAMAGES
WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS, WHETHER
IN AN
ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS ACTION,
ARISING OUT
OF OR IN CONNECTION WITH THE USE OR PERFORMANCE OF THIS
SOFTWARE.

Modified by Jack Jansen, CWI, July 1995:
- Use binascii module to do the actual line-by-line conversion
  between ascii and binary. This results in a 1000-fold
speedup. The C
  version is still 5 times faster, though.
- Arguments more compliant with Python standard
```

XML Remote Procedure Calls

The `xmlrpc.client` module contains the following notice:

```
    The XML-RPC client interface is

Copyright (c) 1999-2002 by Secret Labs AB
Copyright (c) 1999-2002 by Fredrik Lundh

By obtaining, using, and/or copying this software and/or its
associated documentation, you agree that you have read,
understood,
and will comply with the following terms and conditions:

Permission to use, copy, modify, and distribute this software
and
its associated documentation for any purpose and without fee is
hereby granted, provided that the above copyright notice
appears in
all copies, and that both that copyright notice and this
permission
notice appear in supporting documentation, and that the name of
Secret Labs AB or the author not be used in advertising or
publicity
pertaining to distribution of the software without specific,
written
prior permission.

SECRET LABS AB AND THE AUTHOR DISCLAIMS ALL WARRANTIES WITH
REGARD
TO THIS SOFTWARE, INCLUDING ALL IMPLIED WARRANTIES OF MERCHANT-
ABILITY AND FITNESS.  IN NO EVENT SHALL SECRET LABS AB OR THE
AUTHOR
BE LIABLE FOR ANY SPECIAL, INDIRECT OR CONSEQUENTIAL DAMAGES OR
ANY
DAMAGES WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS,
WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS
ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE OR
PERFORMANCE
OF THIS SOFTWARE.
```

test_epoll

The **`test.test_epoll`** module contains the following notice:

```
Copyright (c) 2001-2006 Twisted Matrix Laboratories.

Permission is hereby granted, free of charge, to any person
obtaining
a copy of this software and associated documentation files (the
"Software"), to deal in the Software without restriction,
including
without limitation the rights to use, copy, modify, merge,
publish,
distribute, sublicense, and/or sell copies of the Software, and
to
permit persons to whom the Software is furnished to do so,
subject to
the following conditions:

The above copyright notice and this permission notice shall be
included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND,
EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES
OF
MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND
NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT
HOLDERS BE
LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN
ACTION
OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN
CONNECTION
WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.
```

Select kqueue

The **`select`** module contains the following notice for the kqueue interface:

```
Copyright (c) 2000 Doug White, 2006 James Knight, 2007
Christian Heimes
All rights reserved.

Redistribution and use in source and binary forms, with or
```

```
without
modification, are permitted provided that the following
conditions
are met:
1. Redistributions of source code must retain the above
copyright
   notice, this list of conditions and the following
disclaimer.
2. Redistributions in binary form must reproduce the above
copyright
   notice, this list of conditions and the following disclaimer
in the
   documentation and/or other materials provided with the
distribution.

THIS SOFTWARE IS PROVIDED BY THE AUTHOR AND CONTRIBUTORS "AS
IS" AND
ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED
TO, THE
IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A
PARTICULAR PURPOSE
ARE DISCLAIMED.  IN NO EVENT SHALL THE AUTHOR OR CONTRIBUTORS
BE LIABLE
FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR
CONSEQUENTIAL
DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF
SUBSTITUTE GOODS
OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS
INTERRUPTION)
HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN
CONTRACT, STRICT
LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING
IN ANY WAY
OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE
POSSIBILITY OF
SUCH DAMAGE.
```

SipHash24

The file `Python/pyhash.c` contains Marek Majkowski' implementation of Dan Bernstein's SipHash24 algorithm. It contains the following note:

```
<MIT License>
Copyright (c) 2013  Marek Majkowski <marek@popcount.org>

Permission is hereby granted, free of charge, to any person obtaining a copy
of this software and associated documentation files (the "Software"), to deal
in the Software without restriction, including without limitation the rights
to use, copy, modify, merge, publish, distribute, sublicense, and/or sell
copies of the Software, and to permit persons to whom the Software is
furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in
all copies or substantial portions of the Software.
</MIT License>

Original location:
   https://github.com/majek/csiphash/

Solution inspired by code from:
   Samuel Neves (supercop/crypto_auth/siphash24/little)
   djb (supercop/crypto_auth/siphash24/little2)
   Jean-Philippe Aumasson (https://131002.net/siphash/siphash24.c)
```

strtod and dtoa

The file `Python/dtoa.c`, which supplies C functions dtoa and strtod for conversion of C doubles to and from strings, is derived from the file of the same name by David M. Gay, currently available from https://web.archive.org/web/20220517033456/http://www.netlib.org/fp/dtoa.c. The original file, as retrieved on March 16, 2009, contains the following copyright and licensing notice:

```
/**************************************************************
**
 *
 * The author of this software is David M. Gay.
 *
 * Copyright (c) 1991, 2000, 2001 by Lucent Technologies.
 *
 * Permission to use, copy, modify, and distribute this
software for any
 * purpose without fee is hereby granted, provided that this
entire notice
 * is included in all copies of any software which is or
includes a copy
 * or modification of this software and in all copies of the
supporting
 * documentation for such software.
 *
 * THIS SOFTWARE IS BEING PROVIDED "AS IS", WITHOUT ANY EXPRESS
OR IMPLIED
 * WARRANTY.  IN PARTICULAR, NEITHER THE AUTHOR NOR LUCENT
MAKES ANY
 * REPRESENTATION OR WARRANTY OF ANY KIND CONCERNING THE
MERCHANTABILITY
 * OF THIS SOFTWARE OR ITS FITNESS FOR ANY PARTICULAR PURPOSE.
 *

***************************************************************
/
```

OpenSSL

The modules `hashlib`, `posix` and `ssl` use the OpenSSL library for added performance if made available by the operating system. Additionally, the Windows and macOS installers for Python may include a copy of the OpenSSL libraries, so we include a copy of the OpenSSL license here. For the OpenSSL 3.0 release, and later releases derived from that, the Apache License v2 applies:

```
                              Apache License
                        Version 2.0, January 2004
```

https://www.apache.org/licenses/

TERMS AND CONDITIONS FOR USE, REPRODUCTION, AND DISTRIBUTION

1. Definitions.

"License" shall mean the terms and conditions for use, reproduction, and distribution as defined by Sections 1 through 9 of this document.

"Licensor" shall mean the copyright owner or entity authorized by the copyright owner that is granting the License.

"Legal Entity" shall mean the union of the acting entity and all other entities that control, are controlled by, or are under common control with that entity. For the purposes of this definition, "control" means (i) the power, direct or indirect, to cause the direction or management of such entity, whether by contract or otherwise, or (ii) ownership of fifty percent (50%) or more of the outstanding shares, or (iii) beneficial ownership of such entity.

"You" (or "Your") shall mean an individual or Legal Entity exercising permissions granted by this License.

"Source" form shall mean the preferred form for making modifications, including but not limited to software source code, documentation source, and configuration files.

"Object" form shall mean any form resulting from mechanical transformation or translation of a Source form, including but not limited to compiled object code, generated documentation, and conversions to other media types.

"Work" shall mean the work of authorship, whether in Source or Object form, made available under the License, as indicated by a copyright notice that is included in or attached to the work (an example is provided in the Appendix below).

"Derivative Works" shall mean any work, whether in Source or Object form, that is based on (or derived from) the Work and for which the editorial revisions, annotations, elaborations, or other modifications represent, as a whole, an original work of authorship. For the purposes of this License, Derivative Works shall not include works that remain separable from, or merely link (or bind by name) to the interfaces of, the Work and Derivative Works thereof.

"Contribution" shall mean any work of authorship, including the original version of the Work and any modifications or additions to that Work or Derivative Works thereof, that is intentionally submitted to Licensor for inclusion in the Work by the copyright owner or by an individual or Legal Entity authorized to submit on behalf of the copyright owner. For the purposes of this definition, "submitted" means any form of electronic, verbal, or written communication sent to the Licensor or its representatives, including but not limited to communication on electronic mailing lists, source code control systems, and issue tracking systems that are managed by, or on behalf of, the

Licensor for the purpose of discussing and improving the Work, but excluding communication that is conspicuously marked or otherwise designated in writing by the copyright owner as "Not a Contribution."

"Contributor" shall mean Licensor and any individual or Legal Entity on behalf of whom a Contribution has been received by Licensor and subsequently incorporated within the Work.

2. Grant of Copyright License. Subject to the terms and conditions of this License, each Contributor hereby grants to You a perpetual, worldwide, non-exclusive, no-charge, royalty-free, irrevocable copyright license to reproduce, prepare Derivative Works of, publicly display, publicly perform, sublicense, and distribute the Work and such Derivative Works in Source or Object form.

3. Grant of Patent License. Subject to the terms and conditions of this License, each Contributor hereby grants to You a perpetual, worldwide, non-exclusive, no-charge, royalty-free, irrevocable (except as stated in this section) patent license to make, have made, use, offer to sell, sell, import, and otherwise transfer the Work, where such license applies only to those patent claims licensable by such Contributor that are necessarily infringed by their Contribution(s) alone or by combination of their Contribution(s) with the Work to which such Contribution(s) was submitted. If You institute patent litigation against any entity (including a cross-claim or counterclaim in a lawsuit) alleging that the

Work or a Contribution incorporated within the Work constitutes direct or contributory patent infringement, then any patent licenses granted to You under this License for that Work shall terminate as of the date such litigation is filed.

4. Redistribution. You may reproduce and distribute copies of the Work or Derivative Works thereof in any medium, with or without modifications, and in Source or Object form, provided that You meet the following conditions:

(a) You must give any other recipients of the Work or Derivative Works a copy of this License; and

(b) You must cause any modified files to carry prominent notices stating that You changed the files; and

(c) You must retain, in the Source form of any Derivative Works that You distribute, all copyright, patent, trademark, and attribution notices from the Source form of the Work, excluding those notices that do not pertain to any part of the Derivative Works; and

(d) If the Work includes a "NOTICE" text file as part of its distribution, then any Derivative Works that You distribute must include a readable copy of the attribution notices contained within such NOTICE file, excluding those notices that do not pertain to any part of the Derivative Works, in at least one of the following places: within a NOTICE text file distributed as part of the Derivative Works; within the Source form or documentation, if provided along with the Derivative Works; or, within a display generated by the Derivative Works, if and wherever such third-party notices normally appear. The contents of the NOTICE file are for informational purposes only and do not modify the License. You may add Your own attribution notices within Derivative Works that You distribute, alongside or as an addendum to the NOTICE text from the Work, provided that such additional attribution notices cannot be construed as modifying the License.

You may add Your own copyright statement to Your modifications and may provide additional or different license terms and conditions for use, reproduction, or distribution of Your modifications, or for any such Derivative Works as a whole, provided Your use, reproduction, and distribution of the Work otherwise complies with the conditions stated in this License.

5. Submission of Contributions. Unless You explicitly state otherwise, any Contribution intentionally submitted for inclusion in the Work by You to the Licensor shall be under the terms and conditions of this License, without any additional terms or conditions. Notwithstanding the above, nothing herein shall supersede or modify the terms of any separate license agreement you may have executed

with Licensor regarding such Contributions.

6. Trademarks. This License does not grant permission to use the trade names, trademarks, service marks, or product names of the Licensor, except as required for reasonable and customary use in describing the origin of the Work and reproducing the content of the NOTICE file.

7. Disclaimer of Warranty. Unless required by applicable law or agreed to in writing, Licensor provides the Work (and each Contributor provides its Contributions) on an "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied, including, without limitation, any warranties or conditions of TITLE, NON-INFRINGEMENT, MERCHANTABILITY, or FITNESS FOR A PARTICULAR PURPOSE. You are solely responsible for determining the appropriateness of using or redistributing the Work and assume any risks associated with Your exercise of permissions under this License.

8. Limitation of Liability. In no event and under no legal theory, whether in tort (including negligence), contract, or otherwise, unless required by applicable law (such as deliberate and grossly negligent acts) or agreed to in writing, shall any Contributor be liable to You for damages, including any direct, indirect, special, incidental, or consequential damages of any character arising as a result of this License or out of the use or inability to use the Work (including but not limited to damages for loss of goodwill,

work stoppage, computer failure or malfunction, or any and all other commercial damages or losses), even if such Contributor has been advised of the possibility of such damages.

9. Accepting Warranty or Additional Liability. While redistributing the Work or Derivative Works thereof, You may choose to offer, and charge a fee for, acceptance of support, warranty, indemnity, or other liability obligations and/or rights consistent with this License. However, in accepting such obligations, You may act only on Your own behalf and on Your sole responsibility, not on behalf of any other Contributor, and only if You agree to indemnify, defend, and hold each Contributor harmless for any liability incurred by, or claims asserted against, such Contributor by reason of your accepting any such warranty or additional liability.

END OF TERMS AND CONDITIONS

expat

The `pyexpat` extension is built using an included copy of the expat sources unless the build is configured `--with-system-expat`:

Copyright (c) 1998, 1999, 2000 Thai Open Source Software Center Ltd
and Clark Cooper

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including

```
without limitation the rights to use, copy, modify, merge,
publish,
distribute, sublicense, and/or sell copies of the Software, and
to
permit persons to whom the Software is furnished to do so,
subject to
the following conditions:

The above copyright notice and this permission notice shall be
included
in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND,
EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES
OF
MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND
NONINFRINGEMENT.
IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE
FOR ANY
CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF
CONTRACT,
TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH
THE
SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.
```

libffi

The **`_ctypes`** C extension underlying the **`ctypes`** module is built using an included copy of the libffi sources unless the build is configured `--with-system-libffi`:

```
Copyright (c) 1996-2008  Red Hat, Inc and others.

Permission is hereby granted, free of charge, to any person
obtaining
a copy of this software and associated documentation files (the
"Software"), to deal in the Software without restriction,
including
without limitation the rights to use, copy, modify, merge,
publish,
distribute, sublicense, and/or sell copies of the Software, and
to
```

permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

zlib

The **`zlib`** extension is built using an included copy of the zlib sources if the zlib version found on the system is too old to be used for the build:

Copyright (C) 1995-2011 Jean-loup Gailly and Mark Adler

This software is provided 'as-is', without any express or implied warranty. In no event will the authors be held liable for any damages arising from the use of this software.

Permission is granted to anyone to use this software for any purpose, including commercial applications, and to alter it and redistribute it freely, subject to the following restrictions:

1. The origin of this software must not be misrepresented; you must not claim that you wrote the original software. If you use this

```
software
   in a product, an acknowledgment in the product documentation
would be
   appreciated but is not required.

2. Altered source versions must be plainly marked as such, and
must not be
   misrepresented as being the original software.

3. This notice may not be removed or altered from any source
distribution.

Jean-loup Gailly        Mark Adler
jloup@gzip.org          madler@alumni.caltech.edu
```

cfuhash

The implementation of the hash table used by the **`tracemalloc`** is based on the cfuhash project:

```
Copyright (c) 2005 Don Owens
All rights reserved.

This code is released under the BSD license:

Redistribution and use in source and binary forms, with or
without
modification, are permitted provided that the following
conditions
are met:

  * Redistributions of source code must retain the above
copyright
    notice, this list of conditions and the following
disclaimer.

  * Redistributions in binary form must reproduce the above
    copyright notice, this list of conditions and the following
    disclaimer in the documentation and/or other materials
provided
    with the distribution.
  * Neither the name of the author nor the names of its
    contributors may be used to endorse or promote products
derived
    from this software without specific prior written
permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND
CONTRIBUTORS
"AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT
NOT
LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND
FITNESS
FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE
COPYRIGHT OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT,
INDIRECT,
INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES
(INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS
OR
SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS
INTERRUPTION)
HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN
CONTRACT,
STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE)
ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF
ADVISED
OF THE POSSIBILITY OF SUCH DAMAGE.
```

libmpdec

The **`_decimal`** C extension underlying the `decimal` module is built using an included copy of the libmpdec library unless the build is configured `--with-system-libmpdec`:

```
Copyright (c) 2008-2020 Stefan Krah. All rights reserved.

Redistribution and use in source and binary forms, with or
without
modification, are permitted provided that the following
conditions
are met:

1. Redistributions of source code must retain the above
```

```
copyright
   notice, this list of conditions and the following
disclaimer.

2. Redistributions in binary form must reproduce the above
copyright
   notice, this list of conditions and the following disclaimer
in the
   documentation and/or other materials provided with the
distribution.

THIS SOFTWARE IS PROVIDED BY THE AUTHOR AND CONTRIBUTORS "AS
IS" AND
ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED
TO, THE
IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A
PARTICULAR PURPOSE
ARE DISCLAIMED.  IN NO EVENT SHALL THE AUTHOR OR CONTRIBUTORS
BE LIABLE
FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR
CONSEQUENTIAL
DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF
SUBSTITUTE GOODS
OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS
INTERRUPTION)
HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN
CONTRACT, STRICT
LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING
IN ANY WAY
OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE
POSSIBILITY OF
SUCH DAMAGE.
```

W3C C14N test suite

The C14N 2.0 test suite in the **`test`** package (`Lib/test/xmltestdata/c14n-20/`) was retrieved from the W3C website at https://www.w3.org/TR/xml-c14n2-testcases/ and is distributed under the 3-clause BSD license:

Copyright (c) 2013 W3C(R) (MIT, ERCIM, Keio, Beihang),
All Rights Reserved.

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

* Redistributions of works must retain the original copyright notice, this list of conditions and the following disclaimer.
* Redistributions in binary form must reproduce the original copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.
* Neither the name of the W3C nor the names of its contributors may be used to endorse or promote products derived from this work without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF

THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

mimalloc

MIT License:

Copyright (c) 2018-2021 Microsoft Corporation, Daan Leijen

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

asyncio

Parts of the `asyncio` module are incorporated from uvloop 0.16 [https://github.com/MagicStack/uvloop/tree/v0.16.0], which is distributed under the MIT license:

```
Copyright (c) 2015-2021 MagicStack Inc.  http://magic.io

Permission is hereby granted, free of charge, to any person
obtaining
a copy of this software and associated documentation files (the
"Software"), to deal in the Software without restriction,
including
without limitation the rights to use, copy, modify, merge,
publish,
distribute, sublicense, and/or sell copies of the Software, and
to
permit persons to whom the Software is furnished to do so,
subject to
the following conditions:

The above copyright notice and this permission notice shall be
included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND,
EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES
OF
MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND
NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT
HOLDERS BE
LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN
ACTION
OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN
CONNECTION
WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.
```

Global Unbounded Sequences (GUS)

The file `Python/qsbr.c` is adapted from FreeBSD’s “Global Unbounded Sequences” safe memory reclamation scheme in subr_smr.c [https://github.com/freebsd/freebsd-src/blob/main/sys/kern/subr_smr.c]. The file is distributed under the 2-Clause BSD License:

Copyright (c) 2019,2020 Jeffrey Roberson <jeff@FreeBSD.org>

Redistribution and use in source and binary forms, with or without
modification, are permitted provided that the following conditions
are met:

1. Redistributions of source code must retain the above copyright
   notice unmodified, this list of conditions, and the following
   disclaimer.
2. Redistributions in binary form must reproduce the above copyright
   notice, this list of conditions and the following disclaimer in the
   documentation and/or other materials provided with the distribution.

THIS SOFTWARE IS PROVIDED BY THE AUTHOR "AS IS" AND ANY EXPRESS OR
IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES
OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED.
IN NO EVENT SHALL THE AUTHOR BE LIABLE FOR ANY DIRECT, INDIRECT,
INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT
NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE,
DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY
THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT
(INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF
THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

Zstandard bindings

Zstandard bindings in `Modules/_zstd` and `Lib/compression/zstd` are based on code from the pyzstd library [https://github.com/Rogdham/pyzstd/], copyright Ma Lin and contributors. The pyzstd code is distributed under the 3-Clause BSD License:

```
Copyright (c) 2020-present, Ma Lin and contributors.
All rights reserved.

Redistribution and use in source and binary forms, with or
without
modification, are permitted provided that the following
conditions are met:

1. Redistributions of source code must retain the above
copyright notice, this
   list of conditions and the following disclaimer.

2. Redistributions in binary form must reproduce the above
copyright notice,
   this list of conditions and the following disclaimer in the
documentation
   and/or other materials provided with the distribution.

3. Neither the name of the copyright holder nor the names of
its
   contributors may be used to endorse or promote products
derived from
   this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND
CONTRIBUTORS "AS IS"
AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT
LIMITED TO, THE
IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A
PARTICULAR PURPOSE ARE
DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT HOLDER OR
CONTRIBUTORS BE LIABLE
FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR
CONSEQUENTIAL
DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF
SUBSTITUTE GOODS OR
```

SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER
CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY,
OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE
OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

Python Module Index

_ | a | b | c | d | e | f | g | h | i | j | k | l | m | n | o | p | q | r | s | t | u | v | w | x | z

_	
`__future__`	*Future statement definitions*
`__main__`	*The environment where top-level code is run. Covers command-line interfaces, import-time behavior, and ``__name__ == '__main__'``.*
`_thread`	*Low-level threading API.*
`_tkinter`	*A binary module that contains the low-level interface to Tcl/Tk.*
a	
`abc`	*Abstract base classes according to :pep:`3119`.*
`aifc`	**Deprecated:** *Removed in 3.13.*
`annotationlib`	*Functionality for introspecting annotations*
`argparse`	*Command-line option and argument parsing library.*
`array`	*Space efficient arrays of uniformly typed numeric values.*

`ast`	*Abstract Syntax Tree classes and manipulation.*
`asynchat`	**Deprecated:** *Removed in 3.12.*
`asyncio`	*Asynchronous I/O.*
`asyncore`	**Deprecated:** *Removed in 3.12.*
`atexit`	*Register and execute cleanup functions.*
`audioop`	**Deprecated:** *Removed in 3.13.*
b	
`base64`	*RFC 4648: Base16, Base32, Base64 Data Encodings; Base85 and Ascii85*
`bdb`	*Debugger framework.*
`binascii`	*Tools for converting between binary and various ASCII-encoded binary representations.*
`bisect`	*Array bisection algorithms for binary searching.*
`builtins`	*The module that provides the built-in namespace.*
`bz2`	*Interfaces for bzip2 compression and decompression.*
c	

`calendar`	*Functions for working with calendars, including some emulation of the Unix cal program.*
`cgi`	**Deprecated:** *Removed in 3.13.*
`cgitb`	**Deprecated:** *Removed in 3.13.*
`chunk`	**Deprecated:** *Removed in 3.13.*
`cmath`	*Mathematical functions for complex numbers.*
`cmd`	*Build line-oriented command interpreters.*
`code`	*Facilities to implement read-eval-print loops.*
`codecs`	*Encode and decode data and streams.*
`codeop`	*Compile (possibly incomplete) Python code.*
`collections`	*Container datatypes*
`collections.abc`	*Abstract base classes for containers*
`colorsys`	*Conversion functions between RGB and other color systems.*
`compileall`	*Tools for byte-compiling all Python source files in a directory tree.*
`compression`	
`compression.zstd`	*Low-level interface to compression and*

	decompression routines in the zstd library.
`concurrent`	
`concurrent.futures`	*Execute computations concurrently using threads or processes.*
`concurrent.interpreters`	*Multiple interpreters in the same process*
`configparser`	*Configuration file parser.*
`contextlib`	*Utilities for with-statement contexts.*
`contextvars`	*Context Variables*
`copy`	*Shallow and deep copy operations.*
`copyreg`	*Register pickle support functions.*
`cProfile`	
`crypt`	**Deprecated:** *Removed in 3.13.*
`csv`	*Write and read tabular data to and from delimited files.*
`ctypes`	*A foreign function library for Python.*
`curses` *(Unix)*	*An interface to the curses library, providing portable terminal handling.*
`curses.ascii`	*Constants and set-membership functions for ASCII characters.*
`curses.panel`	*A panel stack extension that adds depth to curses windows.*

`curses.textpad`	*Emacs-like input editing in a curses window.*
d	
`dataclasses`	*Generate special methods on user-defined classes.*
`datetime`	*Basic date and time types.*
`dbm`	*Interfaces to various Unix "database" formats.*
`dbm.dumb`	*Portable implementation of the simple DBM interface.*
`dbm.gnu` *(Unix)*	*GNU database manager*
`dbm.ndbm` *(Unix)*	*The New Database Manager*
`dbm.sqlite3` *(All)*	*SQLite backend for dbm*
`decimal`	*Implementation of the General Decimal Arithmetic Specification.*
`difflib`	*Helpers for computing differences between objects.*
`dis`	*Disassembler for Python bytecode.*
`distutils`	**Deprecated:** *Removed in 3.12.*
`doctest`	*Test pieces of code within docstrings.*
e	
`email`	*Package supporting the parsing, manipulating, and generating email messages.*
`email.charset`	*Character Sets*

`email.contentmanager`	*Storing and Retrieving Content from MIME Parts*
`email.encoders`	*Encoders for email message payloads.*
`email.errors`	*The exception classes used by the email package.*
`email.generator`	*Generate flat text email messages from a message structure.*
`email.header`	*Representing non-ASCII headers*
`email.headerregistry`	*Automatic Parsing of headers based on the field name*
`email.iterators`	*Iterate over a message object tree.*
`email.message`	*The base class representing email messages.*
`email.mime`	*Build MIME messages.*
`email.mime.application`	
`email.mime.audio`	
`email.mime.base`	
`email.mime.image`	
`email.mime.message`	
`email.mime.multipart`	
`email.mime.nonmultipart`	
`email.mime.text`	
`email.parser`	*Parse flat text email messages to produce a message object structure.*
`email.policy`	*Controlling the parsing and generating of messages*

`email.utils`	*Miscellaneous email package utilities.*
`encodings`	*Encodings package*
`encodings.idna`	*Internationalized Domain Names implementation*
`encodings.mbcs`	*Windows ANSI codepage*
`encodings.utf_8_sig`	*UTF-8 codec with BOM signature*
`ensurepip`	*Bootstrapping the "pip" installer into an existing Python installation or virtual environment.*
`enum`	*Implementation of an enumeration class.*
`errno`	*Standard errno system symbols.*
f	
`faulthandler`	*Dump the Python traceback.*
`fcntl` *(Unix)*	*The fcntl() and ioctl() system calls.*
`filecmp`	*Compare files efficiently.*
`fileinput`	*Loop over standard input or a list of files.*
`fnmatch`	*Unix shell style filename pattern matching.*
`fractions`	*Rational numbers.*
`ftplib`	*FTP protocol client (requires sockets).*
`functools`	*Higher-order functions and operations on callable objects.*

g	
`gc`	*Interface to the cycle-detecting garbage collector.*
`getopt`	*Portable parser for command line options; support both short and long option names.*
`getpass`	*Portable reading of passwords and retrieval of the userid.*
`gettext`	*Multilingual internationalization services.*
`glob`	*Unix shell style pathname pattern expansion.*
`graphlib`	*Functionality to operate with graph-like structures*
`grp` *(Unix)*	*The group database (getgrnam() and friends).*
`gzip`	*Interfaces for gzip compression and decompression using file objects.*

h	
`hashlib`	*Secure hash and message digest algorithms.*
`heapq`	*Heap queue algorithm (a.k.a. priority queue).*
`hmac`	*Keyed-Hashing for Message Authentication (HMAC) implementation*
`html`	*Helpers for manipulating HTML.*

`html.entities`	*Definitions of HTML general entities.*
`html.parser`	*A simple parser that can handle HTML and XHTML.*
`http`	*HTTP status codes and messages*
`http.client`	*HTTP and HTTPS protocol client (requires sockets).*
`http.cookiejar`	*Classes for automatic handling of HTTP cookies.*
`http.cookies`	*Support for HTTP state management (cookies).*
`http.server`	*HTTP server and request handlers.*
i	
`idlelib`	*Implementation package for the IDLE shell/editor.*
`imaplib`	*IMAP4 protocol client (requires sockets).*
`imghdr`	**Deprecated:** *Removed in 3.13.*
`imp`	**Deprecated:** *Removed in 3.12.*
`importlib`	*The implementation of the import machinery.*
`importlib.abc`	*Abstract base classes related to import*
`importlib.machinery`	*Importers and path hooks*
`importlib.metadata`	*Accessing package metadata*

importlib.resources	*Package resource reading, opening, and access*
importlib.resources.abc	*Abstract base classes for resources*
importlib.util	*Utility code for importers*
inspect	*Extract information and source code from live objects.*
io	*Core tools for working with streams.*
ipaddress	*IPv4/IPv6 manipulation library.*
itertools	*Functions creating iterators for efficient looping.*
j	
json	*Encode and decode the JSON format.*
json.tool	*A command-line interface to validate and pretty-print JSON.*
k	
keyword	*Test whether a string is a keyword in Python.*
l	
linecache	*Provides random access to individual lines from text files.*
locale	*Internationalization services.*
logging	*Flexible event logging system for applications.*
logging.config	*Configuration of the logging module.*

`logging.handlers`	*Handlers for the logging module.*
`lzma`	*A Python wrapper for the liblzma compression library.*
m	
`mailbox`	*Manipulate mailboxes in various formats*
`mailcap`	**Deprecated:** *Removed in 3.13.*
`marshal`	*Convert Python objects to streams of bytes and back (with different constraints).*
`math`	*Mathematical functions (sin() etc.).*
`mimetypes`	*Mapping of filename extensions to MIME types.*
`mmap`	*Interface to memory-mapped files for Unix and Windows.*
`modulefinder`	*Find modules used by a script.*
`msilib`	**Deprecated:** *Removed in 3.13.*
`msvcrt` *(Windows)*	*Miscellaneous useful routines from the MS VC++ runtime.*
`multiprocessing`	*Process-based parallelism.*
`multiprocessing.connection`	*API for dealing with sockets.*
`multiprocessing.dummy`	*Dumb wrapper around threading.*
`multiprocessing.managers`	*Share data between process with shared objects.*
`multiprocessing.pool`	*Create pools of processes.*

multiprocessing.shared_memory	*Provides shared memory for direct access across processes.*
multiprocessing.sharedctypes	*Allocate ctypes objects from shared memory.*
n	
netrc	*Loading of .netrc files.*
nis	**Deprecated:** *Removed in 3.13.*
nntplib	**Deprecated:** *Removed in 3.13.*
numbers	*Numeric abstract base classes (Complex, Real, Integral, etc.).*
o	
operator	*Functions corresponding to the standard operators.*
optparse	*Command-line option parsing library.*
os	*Miscellaneous operating system interfaces.*
os.path	*Operations on pathnames.*
ossaudiodev	**Deprecated:** *Removed in 3.13.*
p	
pathlib	*Object-oriented filesystem paths*
pathlib.types	*pathlib types for static type checking*

`pdb`	*The Python debugger for interactive interpreters.*
`pickle`	*Convert Python objects to streams of bytes and back.*
`pickletools`	*Contains extensive comments about the pickle protocols and pickle-machine opcodes, as well as some useful functions.*
`pipes`	**Deprecated:** *Removed in 3.13.*
`pkgutil`	*Utilities for the import system.*
`platform`	*Retrieves as much platform identifying data as possible.*
`plistlib`	*Generate and parse Apple plist files.*
`poplib`	*POP3 protocol client (requires sockets).*
`posix` *(Unix)*	*The most common POSIX system calls (normally used via module os).*
`pprint`	*Data pretty printer.*
`profile`	*Python source profiler.*
`pstats`	*Statistics object for use with the profiler.*
`pty` *(Unix)*	*Pseudo-Terminal Handling for Unix.*
`pwd` *(Unix)*	*The password database (getpwnam() and friends).*
`py_compile`	*Generate byte-code files from Python source files.*
`pyclbr`	*Supports information extraction for a Python*

	module browser.
`pydoc`	*Documentation generator and online help system.*
q	
`queue`	*A synchronized queue class.*
`quopri`	*Encode and decode files using the MIME quoted-printable encoding.*
r	
`random`	*Generate pseudo-random numbers with various common distributions.*
`re`	*Regular expression operations.*
`readline` *(Unix)*	*GNU readline support for Python.*
`reprlib`	*Alternate repr() implementation with size limits.*
`resource` *(Unix)*	*An interface to provide resource usage information on the current process.*
`rlcompleter`	*Python identifier completion, suitable for the GNU readline library.*
`runpy`	*Locate and run Python modules without importing them first.*

s	
`sched`	*General purpose event scheduler.*
`secrets`	*Generate secure random numbers for managing secrets.*
`select`	*Wait for I/O completion on multiple streams.*
`selectors`	*High-level I/O multiplexing.*
`shelve`	*Python object persistence.*
`shlex`	*Simple lexical analysis for Unix shell-like languages.*
`shutil`	*High-level file operations, including copying.*
`signal`	*Set handlers for asynchronous events.*
`site`	*Module responsible for site-specific configuration.*
`sitecustomize`	
`smtpd`	**Deprecated:** *Removed in 3.12.*
`smtplib`	*SMTP protocol client (requires sockets).*
`sndhdr`	**Deprecated:** *Removed in 3.13.*
`socket`	*Low-level networking interface.*
`socketserver`	*A framework for network servers.*
`spwd`	**Deprecated:** *Removed in 3.13.*

`sqlite3`	*A DB-API 2.0 implementation using SQLite 3.x.*
`ssl`	*TLS/SSL wrapper for socket objects*
`stat`	*Utilities for interpreting the results of os.stat(), os.lstat() and os.fstat().*
`statistics`	*Mathematical statistics functions*
`string`	*Common string operations.*
`string.templatelib`	*Support for template string literals.*
`stringprep`	*String preparation, as per RFC 3453*
`struct`	*Interpret bytes as packed binary data.*
`subprocess`	*Subprocess management.*
`sunau`	**Deprecated:** *Removed in 3.13.*
`symtable`	*Interface to the compiler's internal symbol tables.*
`sys`	*Access system-specific parameters and functions.*
`sys.monitoring`	*Access and control event monitoring*
`sysconfig`	*Python's configuration information*
`syslog` *(Unix)*	*An interface to the Unix syslog library routines.*
t	

`tabnanny`	*Tool for detecting white space related problems in Python source files in a directory tree.*
`tarfile`	*Read and write tar-format archive files.*
`telnetlib`	**Deprecated:** *Removed in 3.13.*
`tempfile`	*Generate temporary files and directories.*
`termios` *(Unix)*	*POSIX style tty control.*
`test`	*Regression tests package containing the testing suite for Python.*
`test.regrtest`	*Drives the regression test suite.*
`test.support`	*Support for Python's regression test suite.*
`test.support.bytecode_helper`	*Support tools for testing correct bytecode generation.*
`test.support.import_helper`	*Support for import tests.*
`test.support.os_helper`	*Support for os tests.*
`test.support.script_helper`	*Support for Python's script execution tests.*
`test.support.socket_helper`	*Support for socket tests.*
`test.support.threading_helper`	*Support for threading tests.*
`test.support.warnings_helper`	*Support for warnings tests.*
`textwrap`	*Text wrapping and filling*
`threading`	*Thread-based parallelism.*
`time`	*Time access and conversions.*

`timeit`	*Measure the execution time of small code snippets.*
`tkinter`	*Interface to Tcl/Tk for graphical user interfaces*
`tkinter.colorchooser` *(Tk)*	*Color choosing dialog*
`tkinter.commondialog` *(Tk)*	*Tkinter base class for dialogs*
`tkinter.dnd` *(Tk)*	*Tkinter drag-and-drop interface*
`tkinter.filedialog` *(Tk)*	*Dialog classes for file selection*
`tkinter.font` *(Tk)*	*Tkinter font-wrapping class*
`tkinter.messagebox` *(Tk)*	*Various types of alert dialogs*
`tkinter.scrolledtext` *(Tk)*	*Text widget with a vertical scroll bar.*
`tkinter.simpledialog` *(Tk)*	*Simple dialog windows*
`tkinter.ttk`	*Tk themed widget set*
`token`	*Constants representing terminal nodes of the parse tree.*
`tokenize`	*Lexical scanner for Python source code.*
`tomllib`	*Parse TOML files.*
`trace`	*Trace or track Python statement execution.*
`traceback`	*Print or retrieve a stack traceback.*
`tracemalloc`	*Trace memory allocations.*
`tty` *(Unix)*	*Utility functions that perform common terminal control operations.*

`turtle`	*An educational framework for simple graphics applications*
`turtledemo`	*A viewer for example turtle scripts*
`types`	*Names for built-in types.*
`typing`	*Support for type hints (see :pep:`484`).*
u	
`unicodedata`	*Access the Unicode Database.*
`unittest`	*Unit testing framework for Python.*
`unittest.mock`	*Mock object library.*
`urllib`	
`urllib.error`	*Exception classes raised by urllib.request.*
`urllib.parse`	*Parse URLs into or assemble them from components.*
`urllib.request`	*Extensible library for opening URLs.*
`urllib.response`	*Response classes used by urllib.*
`urllib.robotparser`	*Load a robots.txt file and answer questions about fetchability of other URLs.*
`usercustomize`	
`uu`	**Deprecated:** *Removed in 3.13.*
`uuid`	*UUID objects (universally unique identifiers) according to RFC 9562*

v	
`venv`	*Creation of virtual environments.*
w	
`warnings`	*Issue warning messages and control their disposition.*
`wave`	*Provide an interface to the WAV sound format.*
`weakref`	*Support for weak references and weak dictionaries.*
`webbrowser`	*Easy-to-use controller for web browsers.*
`winreg` *(Windows)*	*Routines and objects for manipulating the Windows registry.*
`winsound` *(Windows)*	*Access to the sound-playing machinery for Windows.*
`wsgiref`	*WSGI Utilities and Reference Implementation.*
`wsgiref.handlers`	*WSGI server/gateway base classes.*
`wsgiref.headers`	*WSGI response header tools.*
`wsgiref.simple_server`	*A simple WSGI HTTP server.*
`wsgiref.types`	*WSGI types for static type checking*
`wsgiref.util`	*WSGI environment utilities.*
`wsgiref.validate`	*WSGI conformance checker.*
x	

`xdrlib`	**Deprecated:** *Removed in 3.13.*
`xml`	*Package containing XML processing modules*
`xml.dom`	*Document Object Model API for Python.*
`xml.dom.minidom`	*Minimal Document Object Model (DOM) implementation.*
`xml.dom.pulldom`	*Support for building partial DOM trees from SAX events.*
`xml.etree.ElementInclude`	
`xml.etree.ElementTree`	*Implementation of the ElementTree API.*
`xml.parsers.expat`	*An interface to the Expat non-validating XML parser.*
`xml.parsers.expat.errors`	
`xml.parsers.expat.model`	
`xml.sax`	*Package containing SAX2 base classes and convenience functions.*
`xml.sax.handler`	*Base classes for SAX event handlers.*
`xml.sax.saxutils`	*Convenience functions and classes for use with SAX.*
`xml.sax.xmlreader`	*Interface which SAX-compliant XML parsers must implement.*
`xmlrpc`	*Server and client modules implementing XML-RPC.*
`xmlrpc.client`	*XML-RPC client access.*
`xmlrpc.server`	*Basic XML-RPC server implementations.*

z	
`zipapp`	*Manage executable Python zip archives*
`zipfile`	*Read and write ZIP-format archive files.*
`zipimport`	*Support for importing Python modules from ZIP archives.*
`zlib`	*Low-level interface to compression and decompression routines compatible with gzip.*
`zoneinfo`	*IANA time zone support*

Pending removal in Python 3.14

- The `ma_version_tag` field in **`PyDictObject`** for extension modules (**PEP 699** [https://peps.python.org/pep-0699/]; gh-101193 [https://github.com/python/cpython/issues/101193]).
- Creating **`immutable types`** with mutable bases (gh-95388 [https://github.com/python/cpython/issues/95388]).

Pending removal in Python 3.15

- The `PyImport_ImportModuleNoBlock()`: Use `PyImport_ImportModule()` instead.
- `PyWeakref_GetObject()` and `PyWeakref_GET_OBJECT()`: Use `PyWeakref_GetRef()` instead. The pythoncapi-compat project [https://github.com/python/pythoncapi-compat/] can be used to get `PyWeakref_GetRef()` on Python 3.12 and older.
- `Py_UNICODE` type and the **`Py_UNICODE_WIDE`** macro: Use **`wchar_t`** instead.
- **`PyUnicode_AsDecodedObject()`**: Use `PyCodec_Decode()` instead.
- **`PyUnicode_AsDecodedUnicode()`**: Use `PyCodec_Decode()` instead; Note that some codecs (for example, “base64”) may return a type other than `str`, such as `bytes`.
- **`PyUnicode_AsEncodedObject()`**: Use `PyCodec_Encode()` instead.
- **`PyUnicode_AsEncodedUnicode()`**: Use `PyCodec_Encode()` instead; Note that some codecs (for example, “base64”) may return a type other than `bytes`, such as `str`.
- Python initialization functions, deprecated in Python 3.13:
  - `Py_GetPath()`: Use `PyConfig_Get("module_search_paths")` (`sys.path`) instead.
  - `Py_GetPrefix()`: Use `PyConfig_Get("base_prefix")` (`sys.base_prefix`) instead. Use `PyConfig_Get("prefix")`

    (`sys.prefix`) if virtual environments need to be handled.
  - `Py_GetExecPrefix()`: Use `PyConfig_Get("base_exec_prefix")` (`sys.base_exec_prefix`) instead. Use `PyConfig_Get("exec_prefix")` (`sys.exec_prefix`) if virtual environments need to be handled.
  - `Py_GetProgramFullPath()`: Use `PyConfig_Get("executable")` (`sys.executable`) instead.
  - `Py_GetProgramName()`: Use `PyConfig_Get("executable")` (`sys.executable`) instead.
  - `Py_GetPythonHome()`: Use `PyConfig_Get("home")` or the `PYTHONHOME` environment variable instead.

  The pythoncapi-compat project [https://github.com/python/pythoncapi-compat/] can be used to get `PyConfig_Get()` on Python 3.13 and older.

- Functions to configure Python's initialization, deprecated in Python 3.11:

  - `PySys_SetArgvEx()`: Set `PyConfig.argv` instead.
  - `PySys_SetArgv()`: Set `PyConfig.argv` instead.
  - `Py_SetProgramName()`: Set `PyConfig.program_name` instead.
  - `Py_SetPythonHome()`: Set `PyConfig.home` instead.
  - `PySys_ResetWarnOptions()`: Clear `sys.warnoptions` and `warnings.filters` instead.

  The `Py_InitializeFromConfig()` API should be used with `PyConfig` instead.

- Global configuration variables:

  - `Py_DebugFlag`: Use `PyConfig.parser_debug` or `PyConfig_Get("parser_debug")` instead.
  - `Py_VerboseFlag`: Use `PyConfig.verbose` or `PyConfig_Get("verbose")` instead.

  - `Py_QuietFlag`: Use `PyConfig.quiet` or `PyConfig_Get("quiet")` instead.
  - `Py_InteractiveFlag`: Use `PyConfig.interactive` or `PyConfig_Get("interactive")` instead.
  - `Py_InspectFlag`: Use `PyConfig.inspect` or `PyConfig_Get("inspect")` instead.
  - `Py_OptimizeFlag`: Use `PyConfig.optimization_level` or `PyConfig_Get("optimization_level")` instead.
  - `Py_NoSiteFlag`: Use `PyConfig.site_import` or `PyConfig_Get("site_import")` instead.
  - `Py_BytesWarningFlag`: Use `PyConfig.bytes_warning` or `PyConfig_Get("bytes_warning")` instead.
  - `Py_FrozenFlag`: Use `PyConfig.pathconfig_warnings` or `PyConfig_Get("pathconfig_warnings")` instead.
  - `Py_IgnoreEnvironmentFlag`: Use `PyConfig.use_environment` or `PyConfig_Get("use_environment")` instead.
  - `Py_DontWriteBytecodeFlag`: Use `PyConfig.write_bytecode` or `PyConfig_Get("write_bytecode")` instead.
  - `Py_NoUserSiteDirectory`: Use `PyConfig.user_site_directory` or `PyConfig_Get("user_site_directory")` instead.
  - `Py_UnbufferedStdioFlag`: Use `PyConfig.buffered_stdio` or `PyConfig_Get("buffered_stdio")` instead.
  - `Py_HashRandomizationFlag`: Use `PyConfig.use_hash_seed` and `PyConfig.hash_seed` or `PyConfig_Get("hash_seed")` instead.
  - `Py_IsolatedFlag`: Use `PyConfig.isolated` or `PyConfig_Get("isolated")` instead.
  - `Py_LegacyWindowsFSEncodingFlag`: Use `PyPreConfig.legacy_windows_fs_encoding` or `PyConfig_Get("legacy_windows_fs_encoding")` instead.
  - `Py_LegacyWindowsStdioFlag`: Use `PyConfig.legacy_windows_stdio` or `PyConfig_Get("legacy_windows_stdio")` instead.
  - `Py_FileSystemDefaultEncoding`, `Py_HasFileSystemDefaultEncoding`: Use

  `PyConfig.filesystem_encoding` or `PyConfig_Get("filesystem_encoding")` instead.
  - **`Py_FileSystemDefaultEncodeErrors`**: Use `PyConfig.filesystem_errors` or `PyConfig_Get("filesystem_errors")` instead.
  - **`Py_UTF8Mode`**: Use `PyPreConfig.utf8_mode` or `PyConfig_Get("utf8_mode")` instead. (see `Py_PreInitialize()`)

The `Py_InitializeFromConfig()` API should be used with `PyConfig` to set these options. Or `PyConfig_Get()` can be used to get these options at runtime.

Pending removal in Python 3.16

- The bundled copy of `libmpdec`.

Pending removal in Python 3.18

- The following private functions are deprecated and planned for removal in Python 3.18:
    - **`_PyBytes_Join()`**: use `PyBytes_Join()`.
    - **`_PyDict_GetItemStringWithError()`**: use `PyDict_GetItemStringRef()`.
    - **`_PyDict_Pop()`**: use `PyDict_Pop()`.
    - **`_PyLong_Sign()`**: use `PyLong_GetSign()`.
    - **`_PyLong_FromDigits()`** and **`_PyLong_New()`**: use `PyLongWriter_Create()`.
    - **`_PyThreadState_UncheckedGet()`**: use `PyThreadState_GetUnchecked()`.
    - **`_PyUnicode_AsString()`**: use `PyUnicode_AsUTF8()`.
    - **`_PyUnicodeWriter_Init()`**: replace `_PyUnicodeWriter_Init(&writer)` with `writer = PyUnicodeWriter_Create(0)`.
    - **`_PyUnicodeWriter_Finish()`**: replace `_PyUnicodeWriter_Finish(&writer)` with `PyUnicodeWriter_Finish(writer)`.
    - **`_PyUnicodeWriter_Dealloc()`**: replace `_PyUnicodeWriter_Dealloc(&writer)` with `PyUnicodeWriter_Discard(writer)`.
    - **`_PyUnicodeWriter_WriteChar()`**: replace `_PyUnicodeWriter_WriteChar(&writer, ch)` with `PyUnicodeWriter_WriteChar(writer, ch)`.
    - **`_PyUnicodeWriter_WriteStr()`**: replace `_PyUnicodeWriter_WriteStr(&writer, str)` with `PyUnicodeWriter_WriteStr(writer, str)`.

- **`_PyUnicodeWriter_WriteSubstring()`**: replace `_PyUnicodeWriter_WriteSubstring(&writer, str, start, end)` with **`PyUnicodeWriter_WriteSubstring(writer, str, start, end)`**.
- **`_PyUnicodeWriter_WriteASCIIString()`**: replace `_PyUnicodeWriter_WriteASCIIString(&writer, str)` with **`PyUnicodeWriter_WriteASCII(writer, str)`**.
- **`_PyUnicodeWriter_WriteLatin1String()`**: replace `_PyUnicodeWriter_WriteLatin1String(&writer, str)` with **`PyUnicodeWriter_WriteUTF8(writer, str)`**.
- **`_PyUnicodeWriter_Prepare()`**: (no replacement).
- **`_PyUnicodeWriter_PrepareKind()`**: (no replacement).
- **`_Py_HashPointer()`**: use **`Py_HashPointer()`**.
- **`_Py_fopen_obj()`**: use **`Py_fopen()`**.

The pythoncapi-compat project [https://github.com/python/pythoncapi-compat/] can be used to get these new public functions on Python 3.13 and older. (Contributed by Victor Stinner in gh-128863 [https://github.com/python/cpython/issues/128863].)

Pending removal in future versions

The following APIs are deprecated and will be removed, although there is currently no date scheduled for their removal.

- `Py_TPFLAGS_HAVE_FINALIZE`: Unneeded since Python 3.8.
- `PyErr_Fetch()`: Use `PyErr_GetRaisedException()` instead.
- `PyErr_NormalizeException()`: Use `PyErr_GetRaisedException()` instead.
- `PyErr_Restore()`: Use `PyErr_SetRaisedException()` instead.
- `PyModule_GetFilename()`: Use `PyModule_GetFilenameObject()` instead.
- `PyOS_AfterFork()`: Use `PyOS_AfterFork_Child()` instead.
- `PySlice_GetIndicesEx()`: Use `PySlice_Unpack()` and `PySlice_AdjustIndices()` instead.
- `PyUnicode_READY()`: Unneeded since Python 3.12
- `PyErr_Display()`: Use `PyErr_DisplayException()` instead.
- `_PyErr_ChainExceptions()`: Use `_PyErr_ChainExceptions1()` instead.
- `PyBytesObject.ob_shash` member: call `PyObject_Hash()` instead.
- Thread Local Storage (TLS) API:
  - `PyThread_create_key()`: Use `PyThread_tss_alloc()` instead.
  - `PyThread_delete_key()`: Use `PyThread_tss_free()` instead.
  - `PyThread_set_key_value()`: Use `PyThread_tss_set()` instead.
  - `PyThread_get_key_value()`: Use `PyThread_tss_get()` instead.

  - `PyThread_delete_key_value()`: Use `PyThread_tss_delete()` instead.
  - `PyThread_ReInitTLS()`: Unneeded since Python 3.7.

Pending removal in Python 3.13

Modules (see **PEP 594** [https://peps.python.org/pep-0594/]):

- **`aifc`**
- **`audioop`**
- **`cgi`**
- **`cgitb`**
- **`chunk`**
- **`crypt`**
- **`imghdr`**
- **`mailcap`**
- **`msilib`**
- **`nis`**
- **`nntplib`**
- **`ossaudiodev`**
- **`pipes`**
- **`sndhdr`**
- **`spwd`**
- **`sunau`**
- **`telnetlib`**
- **`uu`**
- **`xdrlib`**

Other modules:

- **`lib2to3`**, and the **2to3** program (gh-84540 [https://github.com/python/cpython/issues/84540])

APIs:

- **`configparser.LegacyInterpolation`** (gh-90765 [https://github.com/python/cpython/issues/90765])
- `locale.resetlocale()` (gh-90817 [https://github.com/python/cpython/issues/90817])
- **`turtle.RawTurtle.settiltangle()`** (gh-50096 [https://github.com/python/cpython/issues/50096])
- **`unittest.findTestCases()`** (gh-50096 [https://github.com/python/cpython/issues/50096])
- **`unittest.getTestCaseNames()`** (gh-50096 [https://github.com/python/cpython/issues/50096])
- **`unittest.makeSuite()`** (gh-50096 [https://github.com/python/cpython/issues/50096])
- **`unittest.TestProgram.usageExit()`** (gh-67048 [https://github.com/python/cpython/issues/67048])
- **`webbrowser.MacOSX`** (gh-86421 [https://github.com/python/cpython/issues/86421])
- **`classmethod`** descriptor chaining (gh-89519 [https://github.com/python/cpython/issues/89519])

Pending removal in Python 3.14

- `argparse`: The *type*, *choices*, and *metavar* parameters of `argparse.BooleanOptionalAction` are deprecated and will be removed in 3.14. (Contributed by Nikita Sobolev in gh-92248 [https://github.com/python/cpython/issues/92248].)
- `ast`: The following features have been deprecated in documentation since Python 3.8, now cause a `DeprecationWarning` to be emitted at runtime when they are accessed or used, and will be removed in Python 3.14:
  - `ast.Num`
  - `ast.Str`
  - `ast.Bytes`
  - `ast.NameConstant`
  - `ast.Ellipsis`

  Use `ast.Constant` instead. (Contributed by Serhiy Storchaka in gh-90953 [https://github.com/python/cpython/issues/90953].)
- `asyncio`:
  - The child watcher classes `asyncio.MultiLoopChildWatcher`, `asyncio.FastChildWatcher`, `asyncio.AbstractChildWatcher` and `asyncio.SafeChildWatcher` are deprecated and will be removed in Python 3.14. (Contributed by Kumar Aditya in gh-94597 [https://github.com/python/cpython/issues/94597].)

  - **`asyncio.set_child_watcher()`**, **`asyncio.get_child_watcher()`**, **`asyncio.AbstractEventLoopPolicy.set_child_watcher()`** and **`asyncio.AbstractEventLoopPolicy.get_child_watcher()`** are deprecated and will be removed in Python 3.14. (Contributed by Kumar Aditya in gh-94597 [https://github.com/python/cpython/issues/94597].)
  - The **`get_event_loop()`** method of the default event loop policy now emits a **`DeprecationWarning`** if there is no current event loop set and it decides to create one. (Contributed by Serhiy Storchaka and Guido van Rossum in gh-100160 [https://github.com/python/cpython/issues/100160].)

- **`email`**: Deprecated the *isdst* parameter in **`email.utils.localtime()`**. (Contributed by Alan Williams in gh-72346 [https://github.com/python/cpython/issues/72346].)

- **`importlib.abc`** deprecated classes:

  - **`importlib.abc.ResourceReader`**
  - **`importlib.abc.Traversable`**
  - **`importlib.abc.TraversableResources`**

  Use **`importlib.resources.abc`** classes instead:

  - **`importlib.resources.abc.Traversable`**
  - **`importlib.resources.abc.TraversableResources`**

  (Contributed by Jason R. Coombs and Hugo van Kemenade in gh-93963 [https://github.com/python/cpython/issues/93963].)

- **`itertools`** had undocumented, inefficient, historically buggy, and inconsistent support for copy, deepcopy, and pickle operations. This will be removed in 3.14 for a significant reduction in code volume and maintenance burden. (Contributed by Raymond Hettinger in gh-101588 [https://github.com/python/cpython/issues/101588].)

- **`multiprocessing`**: The default start method will change to a safer one on Linux, BSDs, and other non-macOS POSIX platforms where `'fork'` is currently the default (gh-84559 [https://github.com/python/cpython/issues/84559]). Adding a runtime warning about this was deemed too disruptive as the majority of code is not expected to care. Use the **`get_context()`** or **`set_start_method()`** APIs to explicitly specify when your code *requires* `'fork'`. See Contexts and start methods.

- **`pathlib`**: **`is_relative_to()`** and **`relative_to()`**: passing additional arguments is deprecated.

- **`pkgutil`**: **`pkgutil.find_loader()`** and **`pkgutil.get_loader()`** now raise **`DeprecationWarning`**; use **`importlib.util.find_spec()`** instead. (Contributed by Nikita Sobolev in gh-97850 [https://github.com/python/cpython/issues/97850].)

- **`pty`**:
  - `master_open()`: use **`pty.openpty()`**.
  - `slave_open()`: use **`pty.openpty()`**.

- **`sqlite3`**:
  - **`version`** and **`version_info`**.
  - **`execute()`** and **`executemany()`** if named placeholders are used and *parameters* is a sequence instead of a **`dict`**.

- **`urllib`**: **`urllib.parse.Quoter`** is deprecated: it was not intended to be a public API. (Contributed by Gregory P.

Smith in gh-88168 [https://github.com/python/cpython/issues/88168].)

Pending removal in Python 3.15

- The import system:
  - Setting `__cached__` on a module while failing to set `__spec__.cached` is deprecated. In Python 3.15, `__cached__` will cease to be set or take into consideration by the import system or standard library. (gh-97879 [https://github.com/python/cpython/issues/97879])
  - Setting `__package__` on a module while failing to set `__spec__.parent` is deprecated. In Python 3.15, `__package__` will cease to be set or take into consideration by the import system or standard library. (gh-97879 [https://github.com/python/cpython/issues/97879])
- `ctypes`:
  - The undocumented `ctypes.SetPointerType()` function has been deprecated since Python 3.13.
- `http.server`:
  - The obsolete and rarely used `CGIHTTPRequestHandler` has been deprecated since Python 3.13. No direct replacement exists. *Anything* is better than CGI to interface a web server with a request handler.
  - The `--cgi` flag to the **python -m http.server** command-line interface has been deprecated since Python 3.13.
- `importlib`:
  - `load_module()` method: use `exec_module()` instead.
- `locale`:
  - The `getdefaultlocale()` function has been deprecated since Python 3.11. Its removal was originally

    planned for Python 3.13 (gh-90817 [https://github.com/python/cpython/issues/90817]), but has been postponed to Python 3.15. Use **`getlocale()`**, **`setlocale()`**, and **`getencoding()`** instead. (Contributed by Hugo van Kemenade in gh-111187 [https://github.com/python/cpython/issues/111187].)
- **`pathlib`**:
    - **`PurePath.is_reserved()`** has been deprecated since Python 3.13. Use **`os.path.isreserved()`** to detect reserved paths on Windows.
- **`platform`**:
    - **`java_ver()`** has been deprecated since Python 3.13. This function is only useful for Jython support, has a confusing API, and is largely untested.
- **`sysconfig`**:
    - The *check_home* argument of **`sysconfig.is_python_build()`** has been deprecated since Python 3.12.
- **`threading`**:
    - **`RLock()`** will take no arguments in Python 3.15. Passing any arguments has been deprecated since Python 3.14, as the Python version does not permit any arguments, but the C version allows any number of positional or keyword arguments, ignoring every argument.
- **`types`**:
    - **`types.CodeType`**: Accessing **`co_lnotab`** was deprecated in **PEP 626** [https://peps.python.org/pep-0626/] since 3.10 and was planned to be removed in 3.12, but it only got a proper **`DeprecationWarning`** in 3.12. May be removed in 3.15. (Contributed by Nikita Sobolev in gh-101866 [https://github.com/python/cpython/issues/101866].)
- **`typing`**:
    - The undocumented keyword argument syntax for creating **`NamedTuple`** classes (for example, `Point = NamedTuple("Point", x=int, y=int)`) has been

deprecated since Python 3.13. Use the class-based syntax or the functional syntax instead.
  - When using the functional syntax of **`TypedDict`**s, failing to pass a value to the *fields* parameter (`TD = TypedDict("TD")`) or passing `None` (`TD = TypedDict("TD", None)`) has been deprecated since Python 3.13. Use `class TD(TypedDict): pass` or `TD = TypedDict("TD", {})` to create a TypedDict with zero field.
  - The **`typing.no_type_check_decorator()`** decorator function has been deprecated since Python 3.13. After eight years in the **`typing`** module, it has yet to be supported by any major type checker.
- **`wave`**:
  - The **`getmark()`**, **`setmark()`**, and **`getmarkers()`** methods of the **`Wave_read`** and **`Wave_write`** classes have been deprecated since Python 3.13.
- **`zipimport`**:
  - **`load_module()`** has been deprecated since Python 3.10. Use **`exec_module()`** instead. (Contributed by Jiahao Li in gh-125746 [https://github.com/python/cpython/issues/125746].)

Pending removal in Python 3.16

- The import system:
  - Setting `__loader__` on a module while failing to set `__spec__.loader` is deprecated. In Python 3.16, `__loader__` will cease to be set or taken into consideration by the import system or the standard library.
- `array`:
  - The `'u'` format code (`wchar_t`) has been deprecated in documentation since Python 3.3 and at runtime since Python 3.13. Use the `'w'` format code (`Py_UCS4`) for Unicode characters instead.
- `asyncio`:
  - `asyncio.iscoroutinefunction()` is deprecated and will be removed in Python 3.16; use `inspect.iscoroutinefunction()` instead. (Contributed by Jiahao Li and Kumar Aditya in gh-122875 [https://github.com/python/cpython/issues/122875].)
  - `asyncio` policy system is deprecated and will be removed in Python 3.16. In particular, the following classes and functions are deprecated:
    - `asyncio.AbstractEventLoopPolicy`
    - `asyncio.DefaultEventLoopPolicy`
    - `asyncio.WindowsSelectorEventLoopPolicy`

        - `asyncio.WindowsProactorEventLoopPolicy`
        - `asyncio.get_event_loop_policy()`
        - `asyncio.set_event_loop_policy()`

    Users should use `asyncio.run()` or `asyncio.Runner` with *loop_factory* to use the desired event loop implementation.

    For example, to use `asyncio.SelectorEventLoop` on Windows:

    ```
    import asyncio

    async def main():
        ...

    asyncio.run(main(),
    loop_factory=asyncio.SelectorEventLoop)
    ```

    (Contributed by Kumar Aditya in gh-127949 [https://github.com/python/cpython/issues/127949].)

- `builtins`:

    - Bitwise inversion on boolean types, `~True` or `~False` has been deprecated since Python 3.12, as it produces surprising and unintuitive results (`-2` and `-1`). Use `not x` instead for the logical negation of a Boolean. In the rare case that you need the bitwise inversion of the underlying integer, convert to `int` explicitly (`~int(x)`).

- `functools`:

    - Calling the Python implementation of `functools.reduce()` with *function* or *sequence* as keyword arguments has been deprecated since Python 3.14.

- `logging`:

  Support for custom logging handlers with the *strm* argument is deprecated and scheduled for removal in Python 3.16. Define handlers with the *stream* argument instead. (Contributed by Mariusz Felisiak in gh-115032 [https://github.com/python/cpython/issues/115032].)

- `mimetypes`:

  - Valid extensions start with a ‘.’ or are empty for `mimetypes.MimeTypes.add_type()`. Undotted extensions are deprecated and will raise a `ValueError` in Python 3.16. (Contributed by Hugo van Kemenade in gh-75223 [https://github.com/python/cpython/issues/75223].)

- `shutil`:

  - The **`ExecError`** exception has been deprecated since Python 3.14. It has not been used by any function in **`shutil`** since Python 3.4, and is now an alias of `RuntimeError`.

- `symtable`:

  - The `Class.get_methods` method has been deprecated since Python 3.14.

- `sys`:

  - The `_enablelegacywindowsfsencoding()` function has been deprecated since Python 3.13. Use the `PYTHONLEGACYWINDOWSFSENCODING` environment variable instead.

- `sysconfig`:

  - The `sysconfig.expand_makefile_vars()` function has been deprecated since Python 3.14. Use the `vars` argument of `sysconfig.get_paths()` instead.

- `tarfile`:

  - The undocumented and unused `TarFile.tarfile` attribute has been deprecated since Python 3.13.

Pending removal in Python 3.17

- `collections.abc`:
    - `collections.abc.ByteString` is scheduled for removal in Python 3.17.

      Use `isinstance(obj, collections.abc.Buffer)` to test if `obj` implements the buffer protocol at runtime. For use in type annotations, either use `Buffer` or a union that explicitly specifies the types your code supports (e.g., `bytes | bytearray | memoryview`).

      `ByteString` was originally intended to be an abstract class that would serve as a supertype of both `bytes` and `bytearray`. However, since the ABC never had any methods, knowing that an object was an instance of `ByteString` never actually told you anything useful about the object. Other common buffer types such as `memoryview` were also never understood as subtypes of `ByteString` (either at runtime or by static type checkers).

      See **PEP 688** [https://peps.python.org/pep-0688/#current-options] for more details. (Contributed by Shantanu Jain in gh-91896 [https://github.com/python/cpython/issues/91896].)
- `typing`:
    - Before Python 3.14, old-style unions were implemented using the private class

`typing._UnionGenericAlias`. This class is no longer needed for the implementation, but it has been retained for backward compatibility, with removal scheduled for Python 3.17. Users should use documented introspection helpers like **`typing.get_origin()`** and **`typing.get_args()`** instead of relying on private implementation details.

- **`typing.ByteString`**, deprecated since Python 3.9, is scheduled for removal in Python 3.17.

  Use `isinstance(obj, collections.abc.Buffer)` to test if `obj` implements the buffer protocol at runtime. For use in type annotations, either use **`Buffer`** or a union that explicitly specifies the types your code supports (e.g., `bytes | bytearray | memoryview`).

  `ByteString` was originally intended to be an abstract class that would serve as a supertype of both **`bytes`** and **`bytearray`**. However, since the ABC never had any methods, knowing that an object was an instance of **`ByteString`** never actually told you anything useful about the object. Other common buffer types such as **`memoryview`** were also never understood as subtypes of **`ByteString`** (either at runtime or by static type checkers).

  See **PEP 688** [https://peps.python.org/pep-0688/#current-options] for more details. (Contributed by Shantanu Jain in gh-91896 [https://github.com/python/cpython/issues/91896].)

Pending removal in Python 3.18

- `decimal`:
  - The non-standard and undocumented `Decimal` format specifier `'N'`, which is only supported in the **`decimal`** module’s C implementation, has been deprecated since Python 3.13. (Contributed by Serhiy Storchaka in gh-89902 [https://github.com/python/cpython/issues/89902].)

Pending removal in Python 3.19

- `ctypes`:
  - Implicitly switching to the MSVC-compatible struct layout by setting `_pack_` but not `_layout_` on non-Windows platforms.

Pending removal in future versions

The following APIs will be removed in the future, although there is currently no date scheduled for their removal.

- **`argparse`**:
  - Nesting argument groups and nesting mutually exclusive groups are deprecated.
  - Passing the undocumented keyword argument *prefix_chars* to **`add_argument_group()`** is now deprecated.
  - The **`argparse.FileType`** type converter is deprecated.
- **`builtins`**:
  - Generators: `throw(type, exc, tb)` and `athrow(type, exc, tb)` signature is deprecated: use `throw(exc)` and `athrow(exc)` instead, the single argument signature.
  - Currently Python accepts numeric literals immediately followed by keywords, for example `0in x`, `1or x`, `0if 1else 2`. It allows confusing and ambiguous expressions like `[0x1for x in y]` (which can be interpreted as `[0x1 for x in y]` or `[0x1f or x in y]`). A syntax warning is raised if the numeric literal is immediately followed by one of keywords **`and`**, **`else`**, **`for`**, **`if`**, **`in`**, **`is`** and **`or`**. In a future release it will be changed to a syntax error. (gh-87999 [https://github.com/python/cpython/issues/87999])
  - Support for `__index__()` and `__int__()` method returning non-int type: these methods will be required to return an instance of a strict subclass of **`int`**.

  - Support for `__float__()` method returning a strict subclass of **`float`**: these methods will be required to return an instance of **`float`**.
  - Support for `__complex__()` method returning a strict subclass of **`complex`**: these methods will be required to return an instance of **`complex`**.
  - Delegation of `int()` to `__trunc__()` method.
  - Passing a complex number as the *real* or *imag* argument in the **`complex()`** constructor is now deprecated; it should only be passed as a single positional argument. (Contributed by Serhiy Storchaka in gh-109218 [https://github.com/python/cpython/issues/109218].)
- **`calendar`**: `calendar.January` and `calendar.February` constants are deprecated and replaced by **`calendar.JANUARY`** and **`calendar.FEBRUARY`**. (Contributed by Prince Roshan in gh-103636 [https://github.com/python/cpython/issues/103636].)
- **`codecs`**: use **`open()`** instead of **`codecs.open()`**. (gh-133038 [https://github.com/python/cpython/issues/133038])
- **`codeobject.co_lnotab`**: use the **`codeobject.co_lines()`** method instead.
- **`datetime`**:
  - **`utcnow()`**: use `datetime.datetime.now(tz=datetime.UTC)`.
  - **`utcfromtimestamp()`**: use `datetime.datetime.fromtimestamp(timestamp, tz=datetime.UTC)`.
- **`gettext`**: Plural value must be an integer.
- **`importlib`**:
  - **`cache_from_source()`** *debug_override* parameter is deprecated: use the *optimization* parameter instead.
- **`importlib.metadata`**:
  - `EntryPoints` tuple interface.
  - Implicit `None` on return values.
- **`logging`**: the `warn()` method has been deprecated since Python 3.3, use **`warning()`** instead.

- `mailbox`: Use of StringIO input and text mode is deprecated, use BytesIO and binary mode instead.
- `os`: Calling `os.register_at_fork()` in multi-threaded process.
- **`pydoc.ErrorDuringImport`**: A tuple value for *exc_info* parameter is deprecated, use an exception instance.
- `re`: More strict rules are now applied for numerical group references and group names in regular expressions. Only sequence of ASCII digits is now accepted as a numerical reference. The group name in bytes patterns and replacement strings can now only contain ASCII letters and digits and underscore. (Contributed by Serhiy Storchaka in gh-91760 [https://github.com/python/cpython/issues/91760].)
- **`sre_compile`**, **`sre_constants`** and **`sre_parse`** modules.
- `shutil`: `rmtree()`’s *onerror* parameter is deprecated in Python 3.12; use the *onexc* parameter instead.
- `ssl` options and protocols:
    - `ssl.SSLContext` without protocol argument is deprecated.
    - `ssl.SSLContext`: `set_npn_protocols()` and **`selected_npn_protocol()`** are deprecated: use ALPN instead.
    - `ssl.OP_NO_SSL*` options
    - `ssl.OP_NO_TLS*` options
    - `ssl.PROTOCOL_SSLv3`
    - `ssl.PROTOCOL_TLS`
    - `ssl.PROTOCOL_TLSv1`
    - `ssl.PROTOCOL_TLSv1_1`
    - `ssl.PROTOCOL_TLSv1_2`
    - `ssl.TLSVersion.SSLv3`
    - `ssl.TLSVersion.TLSv1`
    - `ssl.TLSVersion.TLSv1_1`
- `threading` methods:
    - **`threading.Condition.notifyAll()`**: use `notify_all()`.
    - **`threading.Event.isSet()`**: use `is_set()`.

  - **`threading.Thread.isDaemon()`**, `threading.Thread.setDaemon()`: use `threading.Thread.daemon` attribute.
  - **`threading.Thread.getName()`**, `threading.Thread.setName()`: use `threading.Thread.name` attribute.
  - **`threading.currentThread()`**: use `threading.current_thread()`.
  - **`threading.activeCount()`**: use `threading.active_count()`.
- `typing.Text` (gh-92332 [https://github.com/python/cpython/issues/92332]).
- The internal class `typing._UnionGenericAlias` is no longer used to implement `typing.Union`. To preserve compatibility with users using this private class, a compatibility shim will be provided until at least Python 3.17. (Contributed by Jelle Zijlstra in gh-105499 [https://github.com/python/cpython/issues/105499].)
- `unittest.IsolatedAsyncioTestCase`: it is deprecated to return a value that is not `None` from a test case.
- `urllib.parse` deprecated functions: `urlparse()` instead
  - `splitattr()`
  - `splithost()`
  - `splitnport()`
  - `splitpasswd()`
  - `splitport()`
  - `splitquery()`
  - `splittag()`
  - `splittype()`
  - `splituser()`
  - `splitvalue()`
  - `to_bytes()`
- `wsgiref`: `SimpleHandler.stdout.write()` should not do partial writes.
- `xml.etree.ElementTree`: Testing the truth value of an `Element` is deprecated. In a future release it will always return `True`. Prefer explicit `len(elem)` or `elem is not None` tests instead.

- `sys._clear_type_cache()` is deprecated: use `sys._clear_internal_caches()` instead.

Distributing Python Modules

Note

Information and guidance on distributing Python modules and packages has been moved to the Python Packaging User Guide [https://packaging.python.org/], and the tutorial on packaging Python projects [https://packaging.python.org/en/latest/tutorials/packaging-projects/].

Argument Clinic How-To

Note

The Argument Clinic How-TO has been moved to the Python Developer’s Guide [https://devguide.python.org/development-tools/clinic/].

How to port Python 2 Code to Python 3

author:: Brett Cannon

Python 2 reached its official end-of-life at the start of 2020. This means that no new bug reports, fixes, or changes will be made to Python 2 - it’s no longer supported: see **PEP 373** [https://peps.python.org/pep-0373/] and status of Python versions [https://devguide.python.org/versions].

If you are looking to port an extension module instead of pure Python code, please see Porting Extension Modules to Python 3.

The archived python-porting [https://mail.python.org/pipermail/python-porting/] mailing list may contain some useful guidance.

Since Python 3.11 the original porting guide was discontinued. You can find the old guide in the archive [https://docs.python.org/3.10/howto/pyporting.html].

Third-party guides

There are also multiple third-party guides that might be useful:

- Guide by Fedora [https://portingguide.readthedocs.io]
- PyCon 2020 tutorial [https://www.youtube.com/watch?v=JgIgEjASOlk]
- Guide by DigitalOcean [https://www.digitalocean.com/community/tutorials/how-to-port-python-2-code-to-python-3]

- Guide by ActiveState [https://www.activestate.com/blog/how-to-migrate-python-2-applications-to-python-3]

`aifc` — Read and write AIFF and AIFC files

Deprecated since version 3.11, removed in version 3.13.

This module is no longer part of the Python standard library. It was removed in Python 3.13 after being deprecated in Python 3.11. The removal was decided in **PEP 594** [https://peps.python.org/pep-0594/].

The last version of Python that provided the **`aifc`** module was Python 3.12 [https://docs.python.org/3.12/library/aifc.html].

`asynchat` — Asynchronous socket command/response handler

Deprecated since version 3.6, removed in version 3.12.

This module is no longer part of the Python standard library. It was removed in Python 3.12 after being deprecated in Python 3.6. The removal was decided in **PEP 594** [https://peps.python.org/pep-0594/].

Applications should use the **`asyncio`** module instead.

The last version of Python that provided the **`asynchat`** module was Python 3.11 [https://docs.python.org/3.11/library/asynchat.html].

`asyncore` — Asynchronous socket handler

Deprecated since version 3.6, removed in version 3.12.

This module is no longer part of the Python standard library. It was removed in Python 3.12 after being deprecated in Python 3.6. The removal was decided in **PEP 594** [https://peps.python.org/pep-0594/].

Applications should use the **`asyncio`** module instead.

The last version of Python that provided the **`asyncore`** module was Python 3.11 [https://docs.python.org/3.11/library/asyncore.html].

`audioop` — Manipulate raw audio data

Deprecated since version 3.11, removed in version 3.13.

This module is no longer part of the Python standard library. It was removed in Python 3.13 after being deprecated in Python 3.11. The removal was decided in **PEP 594** [https://peps.python.org/pep-0594/].

The last version of Python that provided the **`audioop`** module was Python 3.12 [https://docs.python.org/3.12/library/audioop.html].

`cgi` — Common Gateway Interface support

Deprecated since version 3.11, removed in version 3.13.

This module is no longer part of the Python standard library. It was removed in Python 3.13 after being deprecated in Python 3.11. The removal was decided in **PEP 594** [https://peps.python.org/pep-0594/].

A fork of the module on PyPI can be used instead: legacy-cgi [https://pypi.org/project/legacy-cgi/]. This is a copy of the cgi module, no longer maintained or supported by the core Python team.

The last version of Python that provided the `cgi` module was Python 3.12 [https://docs.python.org/3.12/library/cgi.html].

`cgitb` — Traceback manager for CGI scripts

Deprecated since version 3.11, removed in version 3.13.

This module is no longer part of the Python standard library. It was removed in Python 3.13 after being deprecated in Python 3.11. The removal was decided in **PEP 594** [https://peps.python.org/pep-0594/].

A fork of the module on PyPI can now be used instead: legacy-cgi [https://pypi.org/project/legacy-cgi/]. This is a copy of the cgi module, no longer maintained or supported by the core Python team.

The last version of Python that provided the **`cgitb`** module was Python 3.12 [https://docs.python.org/3.12/library/cgitb.html].

`chunk` — Read IFF chunked data

Deprecated since version 3.11, removed in version 3.13.

This module is no longer part of the Python standard library. It was removed in Python 3.13 after being deprecated in Python 3.11. The removal was decided in **PEP 594** [https://peps.python.org/pep-0594/].

The last version of Python that provided the `chunk` module was Python 3.12 [https://docs.python.org/3.12/library/chunk.html].

`crypt` — Function to check Unix passwords

Deprecated since version 3.11, removed in version 3.13.

This module is no longer part of the Python standard library. It was removed in Python 3.13 after being deprecated in Python 3.11. The removal was decided in **PEP 594** [https://peps.python.org/pep-0594/].

Applications can use the **`hashlib`** module from the standard library. Other possible replacements are third-party libraries from PyPI: legacycrypt [https://pypi.org/project/legacycrypt/], bcrypt [https://pypi.org/project/bcrypt/], or argon2-cffi [https://pypi.org/project/argon2-cffi/]. These are not supported or maintained by the Python core team.

The last version of Python that provided the **`crypt`** module was Python 3.12 [https://docs.python.org/3.12/library/crypt.html].

`distutils` — Building and installing Python modules

Deprecated since version 3.10, removed in version 3.12.

This module is no longer part of the Python standard library. It was removed in Python 3.12 after being deprecated in Python 3.10. The removal was decided in **PEP 632** [https://peps.python.org/pep-0632/], which has migration advice [https://peps.python.org/pep-0632/#migration-advice].

The last version of Python that provided the **`distutils`** module was Python 3.11 [https://docs.python.org/3.11/library/distutils.html].

Program frameworks

This chapter is no longer maintained, and the modules it contained have been moved to their respective topical documentation.

- **`cmd`** — Command Line Interface Libraries
- **`shlex`** — Unix Specific Services
- **`turtle`** — Graphical User Interfaces with Tk

`imghdr` — Determine the type of an image

Deprecated since version 3.11, removed in version 3.13.

This module is no longer part of the Python standard library. It was removed in Python 3.13 after being deprecated in Python 3.11. The removal was decided in **PEP 594** [https://peps.python.org/pep-0594/].

Possible replacements are third-party libraries from PyPI: filetype [https://pypi.org/project/filetype/], puremagic [https://pypi.org/project/puremagic/], or python-magic [https://pypi.org/project/python-magic/]. These are not supported or maintained by the Python core team.

The last version of Python that provided the **`imghdr`** module was Python 3.12 [https://docs.python.org/3.12/library/imghdr.html].

`imp` — Access the import internals

Deprecated since version 3.4, removed in version 3.12.

This module is no longer part of the Python standard library. It was removed in Python 3.12 after being deprecated in Python 3.4.

The removal notice includes guidance for migrating code from **`imp`** to **`importlib`**.

The last version of Python that provided the **`imp`** module was Python 3.11 [https://docs.python.org/3.11/library/imp.html].

`mailcap` — Mailcap file handling

Deprecated since version 3.11, removed in version 3.13.

This module is no longer part of the Python standard library. It was removed in Python 3.13 after being deprecated in Python 3.11. The removal was decided in **PEP 594** [https://peps.python.org/pep-0594/].

The last version of Python that provided the **`mailcap`** module was Python 3.12 [https://docs.python.org/3.12/library/mailcap.html].

`msilib` — Read and write Microsoft Installer files

Deprecated since version 3.11, removed in version 3.13.

This module is no longer part of the Python standard library. It was removed in Python 3.13 after being deprecated in Python 3.11. The removal was decided in **PEP 594** [https://peps.python.org/pep-0594/].

The last version of Python that provided the **`msilib`** module was Python 3.12 [https://docs.python.org/3.12/library/msilib.html].

`nis` — Interface to Sun’s NIS (Yellow Pages)

Deprecated since version 3.11, removed in version 3.13.

This module is no longer part of the Python standard library. It was removed in Python 3.13 after being deprecated in Python 3.11. The removal was decided in **PEP 594** [https://peps.python.org/pep-0594/].

The last version of Python that provided the **`nis`** module was Python 3.12 [https://docs.python.org/3.12/library/nis.html].

`nntplib` — NNTP protocol client

Deprecated since version 3.11, removed in version 3.13.

This module is no longer part of the Python standard library. It was removed in Python 3.13 after being deprecated in Python 3.11. The removal was decided in **PEP 594** [https://peps.python.org/pep-0594/].

The last version of Python that provided the **`nntplib`** module was Python 3.12 [https://docs.python.org/3.12/library/nntplib.html].

`ossaudiodev` — Access to OSS-compatible audio devices

Deprecated since version 3.11, removed in version 3.13.

This module is no longer part of the Python standard library. It was removed in Python 3.13 after being deprecated in Python 3.11. The removal was decided in **PEP 594** [https://peps.python.org/pep-0594/].

The last version of Python that provided the **`ossaudiodev`** module was Python 3.12 [https://docs.python.org/3.12/library/ossaudiodev.html].

`pipes` — Interface to shell pipelines

Deprecated since version 3.11, removed in version 3.13.

This module is no longer part of the Python standard library. It was removed in Python 3.13 after being deprecated in Python 3.11. The removal was decided in **PEP 594** [https://peps.python.org/pep-0594/].

Applications should use the **`subprocess`** module instead.

The last version of Python that provided the **`pipes`** module was Python 3.12 [https://docs.python.org/3.12/library/pipes.html].

`smtpd` — SMTP Server

Deprecated since version 3.6, removed in version 3.12.

This module is no longer part of the Python standard library. It was removed in Python 3.12 after being deprecated in Python 3.6. The removal was decided in **PEP 594** [https://peps.python.org/pep-0594/].

A possible replacement is the third-party aiosmtpd [https://pypi.org/project/aiosmtpd/] library. This library is not maintained or supported by the Python core team.

The last version of Python that provided the **`smtpd`** module was Python 3.11 [https://docs.python.org/3.11/library/smtpd.html].

`sndhdr` — Determine type of sound file

Deprecated since version 3.11, removed in version 3.13.

This module is no longer part of the Python standard library. It was removed in Python 3.13 after being deprecated in Python 3.11. The removal was decided in **PEP 594** [https://peps.python.org/pep-0594/].

Possible replacements are third-party modules from PyPI: filetype [https://pypi.org/project/filetype/], puremagic [https://pypi.org/project/puremagic/], or python-magic [https://pypi.org/project/python-magic/]. These are not supported or maintained by the Python core team.

The last version of Python that provided the **`sndhdr`** module was Python 3.12 [https://docs.python.org/3.12/library/sndhdr.html].

`spwd` — The shadow password database

Deprecated since version 3.11, removed in version 3.13.

This module is no longer part of the Python standard library. It was removed in Python 3.13 after being deprecated in Python 3.11. The removal was decided in **PEP 594** [https://peps.python.org/pep-0594/].

A possible replacement is the third-party library python-pam [https://pypi.org/project/python-pam/]. This library is not supported or maintained by the Python core team.

The last version of Python that provided the **`spwd`** module was Python 3.12 [https://docs.python.org/3.12/library/spwd.html].

`sunau` — Read and write Sun AU files

Deprecated since version 3.11, removed in version 3.13.

This module is no longer part of the Python standard library. It was removed in Python 3.13 after being deprecated in Python 3.11. The removal was decided in **PEP 594** [https://peps.python.org/pep-0594/].

The last version of Python that provided the **`sunau`** module was Python 3.12 [https://docs.python.org/3.12/library/sunau.html].

`telnetlib` — Telnet client

Deprecated since version 3.11, removed in version 3.13.

This module is no longer part of the Python standard library. It was removed in Python 3.13 after being deprecated in Python 3.11. The removal was decided in **PEP 594** [https://peps.python.org/pep-0594/].

Possible replacements are third-party libraries from PyPI: telnetlib3 [https://pypi.org/project/telnetlib3/] or Exscript [https://pypi.org/project/Exscript/]. These are not supported or maintained by the Python core team.

The last version of Python that provided the `telnetlib` module was Python 3.12 [https://docs.python.org/3.12/library/telnetlib.html].

`uu` — Encode and decode uuencode files

Deprecated since version 3.11, removed in version 3.13.

This module is no longer part of the Python standard library. It was removed in Python 3.13 after being deprecated in Python 3.11. The removal was decided in **PEP 594** [https://peps.python.org/pep-0594/].

The last version of Python that provided the `uu` module was Python 3.12 [https://docs.python.org/3.12/library/uu.html].

`xdrlib` — Encode and decode XDR data

Deprecated since version 3.11, removed in version 3.13.

This module is no longer part of the Python standard library. It was removed in Python 3.13 after being deprecated in Python 3.11. The removal was decided in **PEP 594** [https://peps.python.org/pep-0594/].

The last version of Python that provided the **`xdrlib`** module was Python 3.12 [https://docs.python.org/3.12/library/xdrlib.html].